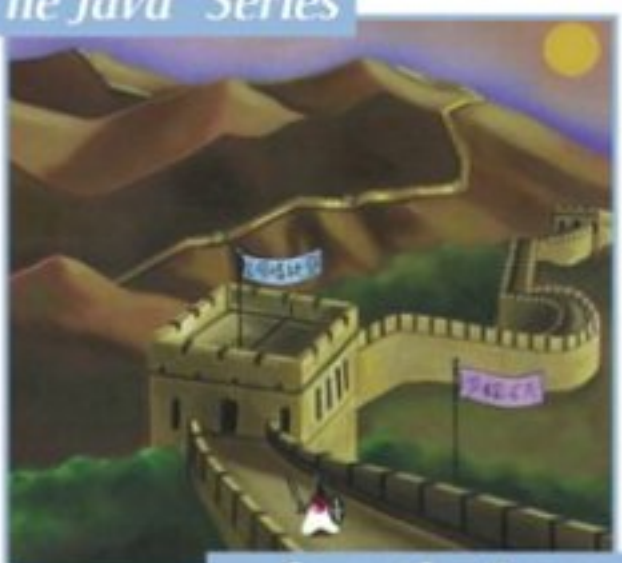
Li Gong • Gary Ellison • Mary Dageforde

Inside Java™ 2 Platform Security, Second Edition

Architecture, API Design, and Implementation

The Java™ Series



...from the Source

Sun Microsystems

JAVA

Preface

Inventing is a combination of brains and materials. The more brains you use, the less material you need.

—Charles Kettering

The phrases "computer security," "network security," and "information security" conjure up various notions and precepts to a given audience. Some people tend to envision technical measures, such as cryptography, as the sole means by which security is attained. Other people recognize the limitations of various technical measures and treat them as tools that, when used in combination with other technical measures, can accomplish the task at hand. The distinction is subtle but important. The phrase "platform security" reflects a holistic view of security, suggesting that the foundation is secure and can be relied on as is or used as a secure subsystem to leverage when building larger systems. Building a secure platform is a very difficult and exacting task that historically has been accomplished only when security is a design requirement that is taken into consideration at the onset. The idea that security can be "bolted on" has proved frail and wrought with failure modes, which has led to a mulititude of security breaches.

Java technology is possibly the only general-purpose secure computing platform to become commercially successful. This would never have happened had the designers not taken security seriously from the start. The security properties of Java technology are many, and the Java platform builds on itself to create a reliable and secure platform. The Java 2 security model would be impossible to make trustworthy if it were not for the safety net provided by the Java language itself. The Java language specifies the semantics to ensure type safety and referential integrity and yet would fail miserably if it were not for the enforcement and assurances the Java virtual machine provides. Thus, from these various secure subsystems, we have created a greater whole.

The target audience of this book is varied. We believe this book will be a useful resource to those seeking a general understanding of the security foundation the Java 2 security architecture provides and relies on. The book should also prove particularily useful to software practitioners building enterprise-class applications that must meet varied security requirements, ranging from authentication to authorization to information protection. This book provides insight into some of the design trade-offs we made as we developed the platform and the lessons we have learned as we continue to evolve and enhance the platform. We provide guidance to those needing to customize the security model for their specific purposes. We describe the inflection points we designed into the platform to accommodate those rare but critical customizations. Most of the aforementioned topics are targeted to system developers, yet we recognize that security is not limited to the implementation of an application. Equally important is the deployment of the application. For deployers, we supply descriptions ranging from expressing security policy to hardening the installation of the runtime environment.

This book does not explain to any level of detail the Java programming language. We recommend the book by Arnold and Gosling [3] as a good starting point. Also, we do not cover the various security APIs in their entirety, and thus we refer the reader to the Java 2 SDK documentation.

How This Book Is Organized

The text of this book is organized to cater to its various audiences. The first two chapters supply background information providing the basis for more specific topics covered in subsequent chapters. The reader need not be proficient in the Java language to understand these introductory chapters. Chapters 3 through 6 describe the Java 2 security architecture, starting with general concepts and ending with comprehensive coverage of security policy enforcement. Chapters 7 through 11 are targeted toward the enterprise application developer, covering topics ranging from trust establishment to cryptography and network security. For these chapters, Java language

proficiency is assumed. Chapter 12 is directly targeted toward deployers, who should also read Chapter 8 for additional details about trust establishment. It is our belief that deployers need not be proficient in the Java language and that they can ignore the sections of Chapter 8 describing APIs.

The content of each chapter of this book is as follows:

Chapter 1: A general background on computer, network, and information security

Chapter 2: A review of the Java security models, starting with the original *sandbox* and progressing to the fine-grained access control model

Chapter 3: An in-depth look at the Java 2 security architecture, which is policy driven and capable of enforcing fine-grained access controls

Chapter 4: Detailed coverage of class loading, including a description of the class loader inheritance hierarchy and the runtime delegation hierarchy

Chapter 5: An explanation of the security classes that supply the foundation for the enforcement of security policy at runtime

Chapter 6: Thorough coverage of the policy enforcement classes and the design of the Java 2 security architecture access control algorithm

Chapter 7: An explanation of the customization points provided for systems programmers who need to enhance the core security architecture

Chapter 8: An outline of the trust establishment capabilities and mechanisms supplied by the security architecture

Chapter 9: A presentation of common pitfalls and defensive programming strategies

Chapter 10: Comprehensive coverage of the cryptography-related APIs

Chapter 11: An operational overview of the APIs used to secure network protocols, including those for authentication, confidentiality, and integrity protection

Chapter 12: A presentation of the deployment options that may be used to securely deploy the Java runtime and Java technology-based applications

Chapter 13: A look at the various Java technology platforms and a glance toward the future of Java security

Acknowledgments

This project began as a casual conversation between Li Gong and me at the 2001 JavaOne conference in San Francisco. Prior to that conversation, Li had transitioned from the role of chief security architect for the Java 2 security development project to leading Project JXTA, whereas I had transitioned into the lead security architect role for the Java 2 development team near the end of the prior millennium. I mentioned to Li that the security architecture had evolved to the point that the first edition was no longer current and thus not an authoritative text.

Nearly two years later, the results of that conversation have come to fruition, and I can confidently state that we have come a long way to reach our goal of producing a book that thoroughly and

accurately describes the Java 2 security architecture. This clearly would not have been possible without Li's support, and I am grateful for having had the opportunity to work with Li in the past and especially on this project.

This book would probably be stuck in the starting blocks if it were not for the guidance and gentle nudging of Lisa Friendly, Manager of Software Technical Publications at Sun Microsystems. Lisa recognized early on that my commitment to the project was absolute but that my copious free time, which was allotted to this effort, fell between the hours of 10 P.M. and 2 A.M. Lisa quickly solved this problem by engaging Mary Dageforde as technical editor. I am forever grateful. Not only is Mary an excellent technical writer and editor who ended up writing enough to get coauthor billing, but she can code too! Mary truly made this project happen with her drive, dedication, and thoroughness. I cannot say enough about Mary, so I will keep it brief. Thank you, Mary.

Tim Lindholm was also an early inspiration, and I appreciate his support in helping me keep things in perspective. I also want to acknowledge the support of my management—Larry Abrahams, Maxine Erlund, Sharon Liu, and Stephen Pelletier—who understood how important this project was to me.

My peers in the Java security development team participated in this publication in many ways, and I wish to acknowledge them for their content contributions, insights, patience, camaraderie, constructive criticism, and most of all their friendship. Thank you, Alan Bateman, Jean-Christophe Collet, Jaya Hangal, Charlie Lai, Rosanna Lee, Jan Luehe, Seema Malkani, Ram Marti, Michael McMahon, Sean Mullan, Jeff Nisewanger, Yu-Ching Peng, Chok Poh, Vincent Ryan, Scott Seligman, Andreas Sterbenz, Mayank Upadhyay, Yingxian Wang, and Brad Wetmore.

Being a part of the team that created something that has had such a significant impact on computing is an honor not shared by many. The success of Java is obviously a result of the high caliber of people who made it a reality. I have had the luxury of working alongside many talented people, and I expressly want to thank Lars Bak, Josh Bloch, Gilad Bracha, Zhiqun Chen, Steffen Garup, James Gosling, Graham Hamilton, Mark Hapner, Stanley Ho, Peter Jones, Peter Kessler, Tim Lindholm, Ron Monzillo, Hans Muller, Hemma Prafullchandra, Mark Reinhold, Rene Schmidt, Bill Shannon, Bob Scheifler, Jim Waldo, and Ann Wollrath for the great experience, mentoring, and technical challenges.

Few people realize the existence and close working relationship the Java security development team at Sun Microsystems maintains with our peers in other organizations. I specifically wish to acknowledge the team at IBM, including Larry Koved, Marco Pistoia, Tony Nadalin, and Bruce Rich, who have been instrumental in enhancing the feature set of the Java 2 security architecture.

As new technologies emerge, we have worked closely with security researchers within Sun Labs to integrate and productize their output. I wish to acknowledge Anne Anderson, Whitfield Diffie, Steve Hanna, Susan Landau, and Radia Perlman for passing along best-in-breed security technology.

I also want to thank the many reviewers of this text and specifically recognize Gilad Bracha, Matt Curtin, James Hoburg, Peter Jones, Charlie Lai, Brian Larkins, Rosanna Lee, John Linn, Ram Marti, Doug Monroe, Sean Mullan, Shivaram Mysore, Vincent Ryan, Bob Scheifler, Andreas Sterbenz, Brad Wetmore, and Phil Yeater for the feedback they provided. I also wish to recognize Peter Jones and Shivaram Mysore for their content contributions.

Thanks also to Alan Sommerer, the Sun Microsystems Manager of Technical Publications for the Java platform, for his help in ushering this book to publication.

Finally, I want to express my gratitude to the production team. I thank the copy editor, Evelyn Pyle, and the production folks at Addison-Wesley for their support and effort in getting this book off my laptop and into print. Thanks to Marcy Barnes, Jacquelyn Doucette, Amy Fleischer, John

Fuller, Mike Hendrickson, Michael Mullen, and Ann Sellers. Also, I want to acknowledge Mary Darby and Amy Girard from Duarte Design for their innate ability to take my graphically challenged images and turn them into a thousand words.

Gary Ellison
San Mateo, California
March 2003

I am grateful to all past and current members of the Java Security and Networking group at Sun, as well as contributors from all over the world, who continue to strengthen Java's position as the premier computing platform in these areas. I am in debt to Gary Ellison and Mary Dageforde for their tremendous effort in producing this second edition which significantly expands the coverage of the first.

Li Gong
Beijing, China

It has been a pleasure working with Gary Ellison on this book. I thank him for his vision, dedication, encouragement, feedback, enormous effort in the face of multiple competing responsibilities, and sense of humor. It has also been my good fortune to work with Li Gong and members of the top-notch Java Security and Networking team at Sun at various times throughout the past several years. I thank them all. Thanks also to Lisa Friendly of Sun and Mike Hendrickson of Addison-Wesley for their support and their roles in facilitating publication of this book. Finally, I would like to thank the copy editor, the graphics designers, and the very helpful production folks at Addison-Wesley.

Mary Dageforde
Santa Clara, California

About the Authors

Li Gong is Managing Director of Sun Microsystems' Engineering and Research Institute in Beijing, China. Previously at Sun, he was engineering head of Java Security and Networking, Java Embedded Servers, and JXTA. He obtained B.S. and M.S. degrees from Tsinghua University, Beijing, and a Ph.D. from the University of Cambridge. He is Associate Editor-in-Chief of *IEEE Internet Computing*.

Gary Ellison is a Senior Staff Engineer at Sun Microsystems, where he designs secure network computing platforms. His primary role is focused on aspects of trust, security, and privacy. From 1999 through 2002, he led the architecture, design, and implementation of the security and networking components in the Java 2 Platform, Standard Edition. He holds a B.Sc. in Mathematics and Physical Science from The Ohio State University.

Mary Dageforde is a freelance consultant who writes software documentation for various Silicon Valley computer companies, including Sun Microsystems. She has an M.S. in Computer Science from Stanford University and a software design and development background encompassing compiler and interpreter implementation, language design, and database management. Since 1990, she has concentrated on documenting APIs, languages, tools, and systems. She wrote the Security trail of *The Java™ Tutorial Continued* (Addison-Wesley, 1999).

Preface to the First Edition

Give me a lever and a fulcrum, and I can move the globe.

—Archimedes

Since Java technology's inception, and especially its public debut in the spring of 1995, strong and growing interest has developed regarding the security of the Java platform, as well as new security issues raised by the deployment of Java technology. This level of attention to security is a fairly new phenomenon in computing history. Most new computing technologies tend to ignore security considerations when they emerge initially, and most are never made more secure thereafter. Attempts made to do so typically are not very successful, as it is now well known that retrofitting security is usually very difficult, if not impossible, and often causes backward compatibility problems.

Thus it is extremely fortunate that when Java technology burst on the Internet scene, security was one of its primary design goals. Its initial security model, although very simplistic, served as a great starting place, an Archimedean fulcrum. The engineering talents and strong management team at JavaSoft are the lever; together they made Java's extensive security architecture a reality.

From a technology provider's point of view, security on the Java platform focuses on two aspects. The first is to provide the Java platform, primarily through the Java Development Kit, as a secure platform on which to run Java-enabled applications in a secure fashion. The second is to provide security tools and services implemented in the Java programming language that enable a wider range of security-sensitive applications, for example, in the enterprise world.

I wrote this book with many purposes in mind. First, I wanted to equip the reader with a brief but clear understanding of the overall picture of systems and network security, especially in the context of the Internet environment within which Java technology plays a central role, and how various security technologies relate to each other.

Second, I wanted to provide a comprehensive description of the current security architecture on the Java platform. This includes language features, platform APIs, security policies, and their enforcement mechanisms. Whenever appropriate, I discuss not only how a feature functions, but also why it is designed in such a way and the alternative approaches that we—the Java security development team at Sun Microsystems—examined and rejected. When demonstrating the use of a class or its methods, I use real-world code examples whenever appropriate. Some of these examples are synthesized from the Java 2 SDK code source tree.

Third, I sought to tell the reader about security deployment issues, both how an individual or an enterprise manages security and how to customize, extend, and enrich the existing security architecture.

Finally, I wanted to help developers avoid programming errors by discussing a number of common mistakes and by providing tips for safe programming that can be immediately applied to ongoing projects.

Acknowledgments for the First Edition

It is a cliche to say that writing a book is not possible without the help of many others, but it is true. I am very grateful to Dick Neiss, my manager at JavaSoft, who encouraged me to write the book and regularly checked on my progress. Lisa Friendly, the Addison-Wesley Java series editor, helped by guiding me through the writing process while maintaining a constant but "friendly" pressure. The team at Addison-Wesley was tremendously helpful. I'd like particularly to thank Mike Hendrickson, Katherine Kwack, Marina Lang, Laura Michaels, Marty Rabinowitz, and Tracy Russ. They are always encouraging, kept faith in me, and rescued me whenever I encountered obstacles.

This book is centered around JDK 1.2 security development, a project that lasted fully two years, during which many people inside and outside of Sun Microsystems contributed in one way or another to the design, implementation, testing, and documentation of the final product. I would like to acknowledge Dirk Balfanz, Bob Blakley, Josh Bloch, David Bowen, Gilad Bracha, David Brownell, Eric Chu, David Connelly, Mary Dageforde, Drew Dean, Satya Dodda, Michal Geva, Gadi Guy, Graham Hamilton, Mimi Hills, Ted Jucevic, Larry Koved, Charlie Lai, Sheng Liang, Tim Lindholm, Jan Luehe, Gary McGraw, Marianne Mueller, Tony Nadalin, Don Neal, Jeff Nisewanger, Yu-Ching Peng, Hemma Prafullchandra, Benjamin Renaud, Roger Riggs, Jim Roskind, Nakul Saraiya, Roland Schemers, Bill Shannon, Vijay Srinivasan, Tom van Vleck, Dan Wallach, and Frank Yellin. I also appreciate the technical guidance from James Gosling and Jim Mitchell, as well as management support from Dick Neiss, Jon Kannegaard, and Alan Baratz. I have had the pleasure of chairing the Java Security Advisory Council, and I thank the external members, Ed Felten, Peter Neumann, Jerome Saltzer, Fred Schneider, and Michael Schroeder for their participation and superb insights into all matters that relate to computer security.

Isabel Cho, Lisa Friendly, Charlie Lai, Jan Luehe, Teresa Lunt, Laura Michaels, Stephen Northcutt, Peter Neumann, and a number of anonymous reviewers provided valuable comments on draft versions of this book.

G. H. Hardy once said that young men should prove theorems, while old men should write books. It is now time to prove some more theorems.

Li Gong
Los Altos, California
June 1999

Chapter 1. Computer and Network Security Fundamentals

The three golden rules to ensure computer security are: do not own a computer; do not power it on; and do not use it.

—Robert (Bob) T. Morris

Security is all about ensuring that bad things do not happen. This deceptively simple brief statement can in fact have very complicated interpretations. Exploring them can help in understanding what security really means.

Certain rule-of-thumb principles apply to the concept of security in general. Throughout this book, you will see that these heuristics apply equally well to computer security. First, security is always related to utility. To ensure that bad things do not happen, you can simply do nothing. For example, a car stored in a garage cannot cause a traffic accident. But doing nothing with the car is clearly not what is intended. The real goal is to ensure that bad things do not happen but that good things do get done.

Second, security is relative to the threat that one considers. For example, the effectiveness of your house's locked front door to prevent theft depends heavily on the types of thieves against which you are guarding. Although the lock might deter an amateur thief, it might not pose a problem for a sophisticated one equipped with the right tools.

Third, security must be considered from an overall system point of view. A system is only as secure as its weakest point. That is, it is not enough to secure only the front door. A skilled thief will try to enter the house from all potentially weak spots, especially those farthest away from where you have installed strong locks. It is of little use to install a deadbolt on a screen door.

Fourth, security must be easy to accomplish. If it takes 30 minutes and great effort to unlock a complicated lock, you will tend to leave the door unlocked.

Fifth, security must be affordable and cost-effective. For example, it clearly does not make sense to install a lock that is worth more than the contents it is guarding. This is made more difficult to gauge due to the fact that the value of something is subjective.

Last, but not least, security measures must be as simple as possible to comprehend because, as experience indicates, the more complex a system is, the more error-prone it tends to be. It is better to have something that is simple and trustworthy than something that is less dependable due to the complexity of building a comprehensive system.

1.1 Cryptography versus Computer Security

Before moving on to specific topics, we want to clarify that *cryptography* and *computer security* are two distinct subjects. Cryptography is the art of encoding information in a secret format such that only the intended recipient can access the information. Cryptography can also be applied to supply proofs of authenticity, integrity, and intent. The use of cryptography has progressed extensively over a long period of time, ranging from the ancient Caesar cipher to cipher machines widely used in World War II to modern cryptosystems implemented with computer hardware and software.

Computer security is the application of measures that ensure that information being processed, stored, or communicated is reliable and available to authorized entities. Computer security first became an issue only in the 1960s, when timesharing, multiuser computer operating systems, such

as Cambridge's early computing system [133] and MIT's Multics [110], were first built. After that, the field of computer security remained relatively obscure for years, apart from a brief active period in the mid-1970s [5, 51, 57, 116]. Security concerns then were based mostly on military requirements. Commercial security did not become fully mainstream until the Internet and electronic commerce (e-commerce)—and Java technology in particular—took center stage in the 1990s.

Security mechanisms often can benefit from the use of cryptography, such as when running a network-based user login protocol. However, they do not necessarily depend on the use of cryptography, such as when implementing UNIX-style access control on files.

Yet cryptography does not exist in a vacuum. Cryptographic algorithms are usually implemented in software or hardware; thus, their correct operation depends critically on whether there is an adequate level of system security. For example, if lack of access control means that an attacker can modify the software that implements the algorithm, the lack of security directly impacts the utilization of cryptography.

1.2 Threats and Protection

In computer security literature, threats or attacks are usually classified into three categories.

1. **Secrecy attacks.** The attacker attempts to steal confidential information, such as passwords, medical records, electronic mail (e-mail) logs, and payroll data. The methods of attack vary, from bribing a security guard to exploiting a security hole in the system or a weakness in a cryptographic algorithm.
2. **Integrity attacks.** The attacker attempts to alter parts of the system illegally. For example, a bank employee modifies the deposit system to transfer customer money into his own account, thus compromising transaction integrity [96]. Or, a college student breaks into the college administration system to raise her examination scores, thus compromising data integrity. An attacker might also try to erase system logs in order to hide his footprint.
3. **Availability attacks.** The attacker attempts to disrupt the normal operation of a system. These are also commonly called *denial-of-service attacks*. For example, bombarding a machine with a large number of IP (Internet Protocol) packets can effectively isolate the machine from the rest of the network. A cyberterrorist might attempt to bring down the national power grid or cause traffic accidents by compromising the computer-operated control systems.

These three categories of attacks are intricately related; that is, the techniques and results of attacks in one category can often be used to assist attacks in another. For example, by compromising secrecy, an attacker could obtain passwords and thus compromise integrity by gaining access to and then modifying system resources, which in turn could lead to successful denial-of-service attacks. When a system failure occurs during an attack, most systems are not *fail-safe*—that is, they do not enter into a state that is deemed secure—because they are not designed to do so [111]. For example, it has been shown that a system crash sometimes leads to a core dump in a publicly readable directory, where the core can contain sensitive information if the dump occurs at the right time.[1]

[1] Of course, attacks can be viewed from other perspectives. For example, there is widespread public concern about the privacy of the unregulated and sometimes illegal collection and distribution of personal data, such as birth dates and U.S. social security numbers.

Similarly, protection mechanisms against these types of attacks in general are related. Roughly speaking, the mechanisms are for one or more of the following purposes: attack prevention, detection, or recovery. Not all these purposes can be fulfilled by the same mechanisms, as explained later in this chapter.

To protect data secrecy, you can store the data in an obscure place in the hope that attackers will not find it. Or you can install strict access control procedures to guard against unauthorized access. Or you can use encryption technology to encrypt the data such that attackers cannot access real data unless they can steal the encryption key or can break the cryptosystem, which could be extremely difficult. Of course, multiple measures can be deployed at the same time. Note that, for secrecy, the most important technique is prevention. A loss of data is very difficult to detect, and lost data is impossible to recover.

To protect data integrity, you can use any or all the mechanisms mentioned previously. However, in this case, detection is easier, and recovery is often possible. For example, you could compute the hash value for a file x, using a wellknown one-way function $f()$, and store $f(x)$ separately. If x is then modified to be x', $f(x)$ very likely will not be equal to $f(x')$, according to the properties of $f()$. Thus, you can recompute the hash value and compare it with $f(x)$. A mismatch will indicate that integrity has been compromised. See Section 1.5.1 for more information on one-way hash functions.

Of course, if the corresponding $f(x)$ is also compromised, detection might not be possible. If the place to store $f(x)$ itself is not safe, you could use a keyed, oneway hash function and store $f(k, x)$ together with x. If k is kept secret, it will still be difficult for attackers to modify x and the hash value in such a way as to avoid detection [39, 83].

To be able to restore the data to its original form after an integrity compromise, you can back up data and store the backup in a secure place [96]. Or you can use more complicated distributed computing techniques to back up the data in an insecure network [53, 98, 114, 118].

Guarding against an availability attack is more complicated. The reason is that apart from applying the usual techniques of prevention and detection, surviving such attacks becomes critical. Here, computer security meets the field of faulttolerant computing. Some interesting research results in this combined topic area, sometimes called *dependable systems*, are available. For further reading, consult the papers and their citations at [24, 42, 99, 114].

1.3 Perimeter Defense

Because of the multitude of potential weaknesses and the essentially unlimited number of attack scenarios, whereby each scenario can be a combination of various attack techniques, securing an entire system can be daunting, especially when the system includes multiple host machines connected via a network. Because a system is only as secure as its weakest link, the security coverage must be comprehensive. The task is further complicated by the fact that a system—for example, the internal network deployed within a large enterprise—typically consists of machines of numerous brands and types. These machines run different operating systems and different application software and are connected with routers and other networking gears from various vendors offering differing features and capabilities. In such a heterogeneous and evolving environment, examining the entire system and securing all its components—if possible at all—takes a long time.

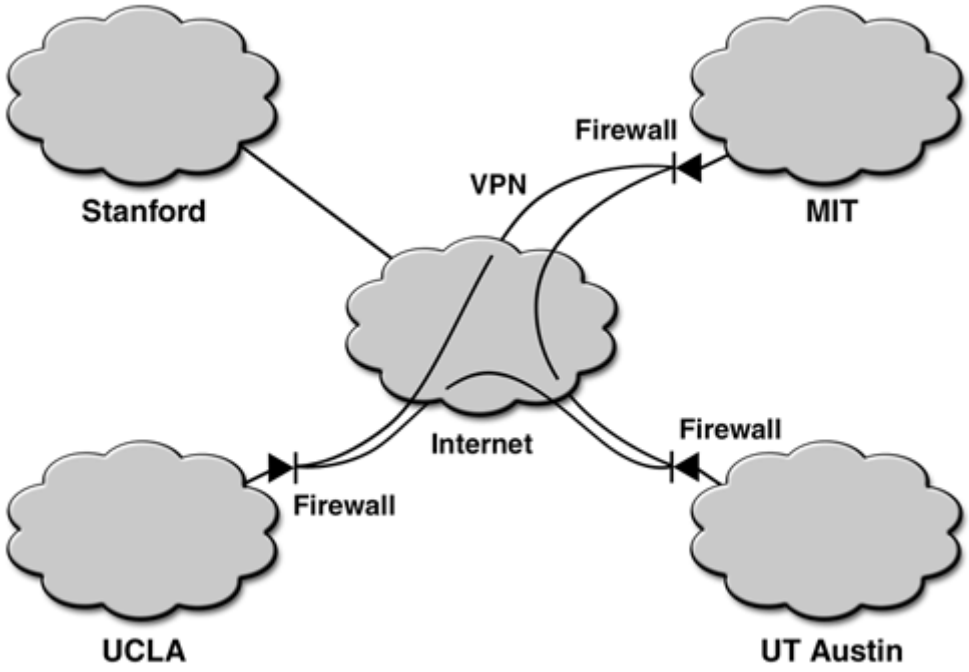
Faced with such a messy picture, it is no surprise that companies find it easier, both psychologically and physically, simply to divide the world into two camps: "us" and "them." "Us" includes all machines owned, operated, or, in general, trusted by the concerned enterprise, whereas "them" includes all other machines, which are potentially hostile and cannot be trusted. Once the border is drawn, it is a matter of keeping "them" out and "us" in. Such a defensive posture is often called *perimeter defense*.

One approach to constructing a perimeter defense is simply not to connect "us" with "them." Indeed, some military installations and commercial entities have internal networks that are entirely

separated from a wider area network: the Internet, for example. They might allow some isolated terminals or machines for outside connections, but these special machines are usually guarded to prevent their being connected to the internal network.

If the overall system contains machines scattered among physical or geographical locations, leased lines or dedicated network connections can link the sites to form a private network. If, however, the sites must communicate through the open network, encryption can be deployed between every two communicating sites so that they form a virtual private network (VPN). This is depicted in the fictitious scenario in Figure 1.1, where, although all four campuses are connected to the Internet, three sites (MIT, UT Austin, and UCLA) have firewalls deployed and have also formed a VPN so that network traffic among them is automatically protected from eavesdropping.

Figure 1.1. Perimeter defense

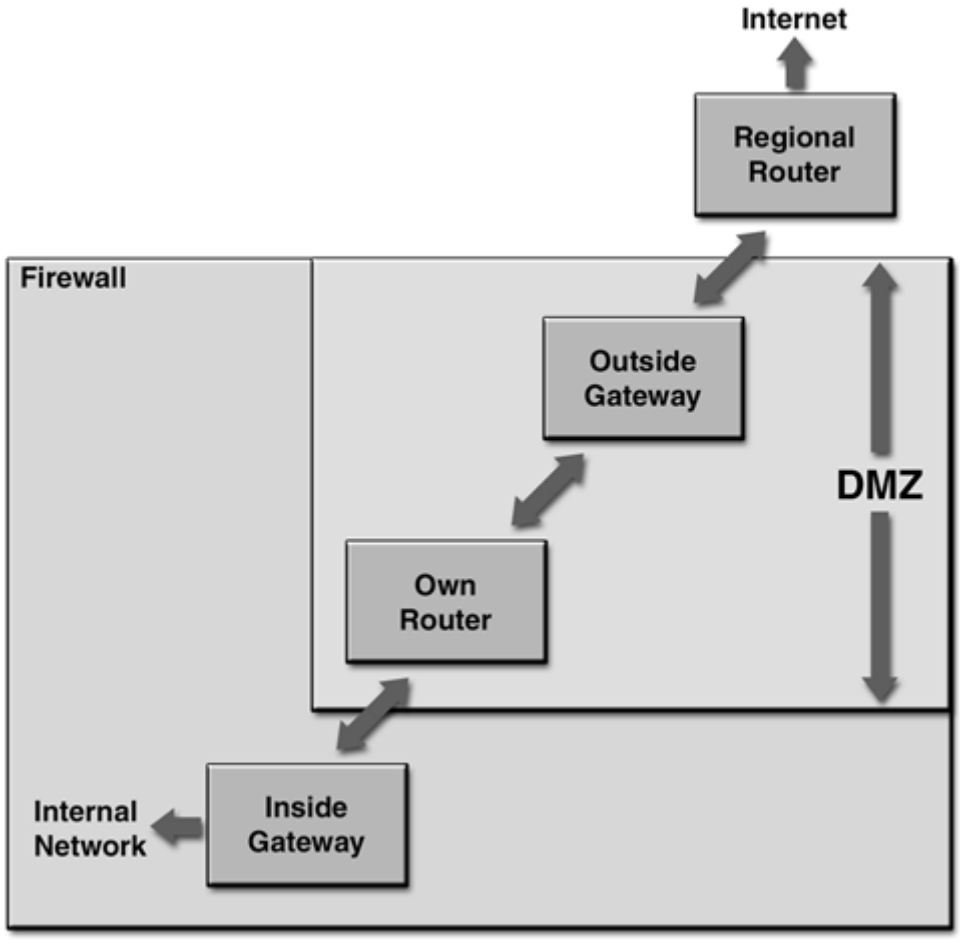


However, such total isolation from the outside does not always work well. For example, e-mail has become the "killer application" of the Internet as people increasingly demand the ability to communicate with the outside world via the Internet. The World Wide Web (the Web) has made the Internet even more popular, and—if used judiciously, of course—browsing the Web to locate information is important to productivity. These trends are driving previously closed enterprises to open up their border control selectively. Here is where firewalls play a critical role in constructing a more useful perimeter defense.

1.3.1 Firewalls

Firewalls come in different shapes and sizes [8]. Generally speaking, as illustrated in Figure 1.2, a firewall is a machine sitting between a private network and a public one. A firewall functions as a filter for network traffic, with the responsibility of selectively allowing certain traffic through, in each direction, based on a security policy. A security policy can be very simple or quite complicated. The reason is that filtering decisions are often based on, for example, the source and destination of the traffic, the protocols used, and the applications involved, among other factors. The firewall also might redirect traffic, act as a proxy server, or even manipulate the traffic content before allowing it to pass through. Furthermore, the firewall might encrypt traffic; indeed, encrypting firewalls can be used to form a VPN.

Figure 1.2. Firewall deployment



Perimeter defense as implemented by firewalls has been shown to be an effective security solution. A firewall provides a central point of control, so a corporate policy can be more easily implemented and updated. But a firewall has certain problems. First, firewalls cannot filter or stop all network traffic. In fact, traffic for such protocols as HTTP (Hypertext Transfer Protocol) is often deliberately let through firewalls. Generally, there is tension between the firewall and the utility the network provides. The firewall attempts to block or reduce unwanted traffic, whereas the primary benefit of the network is its ability to exchange all forms of traffic. A firewall can also be a bottleneck and a single point of communication failure. Moreover, many applications on the desktop have to be rewritten to use the firewall as a proxy. This problem is less severe for new applications, which often have built-in proxy support.

1.3.2 Inadequacies of Perimeter Defense Alone

Perimeter defense alone is not sufficient, however, as a total security solution, for several reasons. Locating and securing all perimeter points is quite difficult. In reported cases, direct telephone line–based connections are established, for diagnostic purposes, that can effectively puncture the perimeter defense [96]. Further, when an enterprise allows its employees to work remotely and from home, inspecting and ensuring that the remote entry points to the internal network are adequately protected may be impractical.

Even within an enterprise, controls are needed because not everything or everyone can be fully trusted. The most devastating attacks often occur from within. Such insider attacks usually incur comparatively large losses because insiders have a significant advantage over external attackers.

For example, the accounting department must be protected so that only authorized employees may issue purchase orders, whereas the patent department must be isolated to prevent information leaks to competitors.

The remainder of this chapter reviews security models and techniques that are useful both within the perimeter and across organizational boundaries.

1.4 Access Control and Security Models

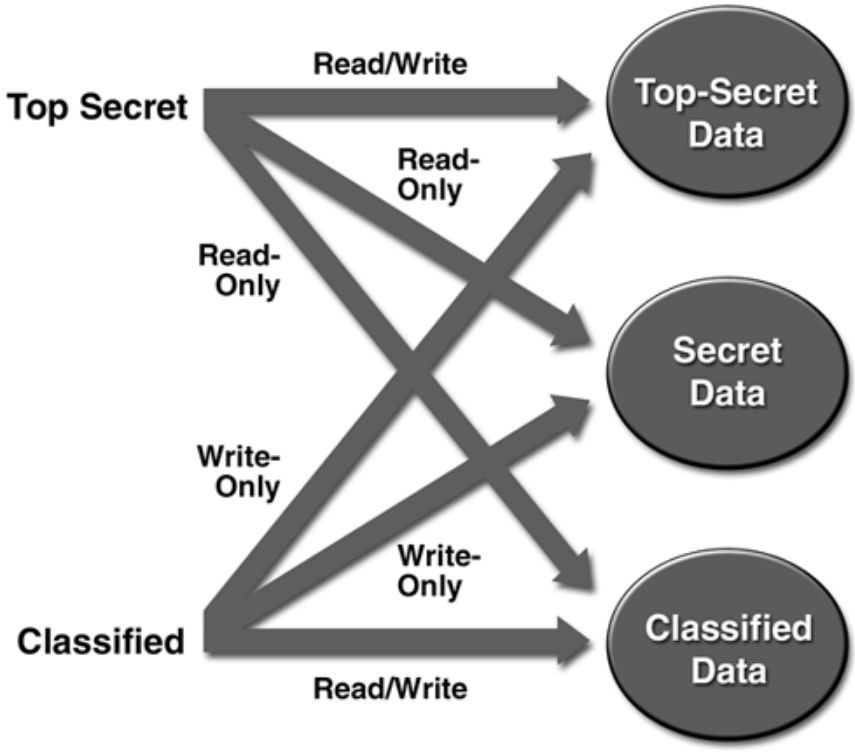
A security model is an abstraction of how one goes about controlling access to protected data. Like firewalls, security models come in various shapes and sizes because requirements for various applications and their environments can differ vastly. Multiple ways to classify security models are available, including the following:

- MAC and DAC models
- Data and information security models
- Static and dynamic models

1.4.1 MAC and DAC Models

One classification of security models centers on the concept of *mandatory access control*, or MAC. In a MAC security model, entities within a system are either *subjects*, roughly corresponding to the notions of users, processes, machines, and so on, or *objects*, roughly corresponding to the targets of control, such as files and data records. Each entity is assigned a sensitivity level. Such levels normally form a lattice over a "dominate" relationship so that, for example, if there are two levels, either one dominates the other or the two are incompatible. For example, levels of "classified," "secret," and "top-secret" could have the dominate relationship shown in Figure 1.3.

Figure 1.3. MAC security model



MAC models meeting the requirements of multilevel security are exemplified by the work of Bell and LaPadula [5], describing a mathematical model for the security of the Multics system [110]. In the Bell-LaPadula model, a subject may have read access to an object if, and only if, its level dominates that of the object and may have write access to an object if, and only if, its level is dominated by that of the object. This is called informally *read-down* and *write-up*, or more precisely, no *read-up* and no *write-down*. According to this model, two entities may communicate in both directions only when either they are at the same level or they do so via a trusted intermediary.

Non-MAC models are called *discretionary access control*, or DAC, models. The UNIX security model is similar to a DAC model in that the owner (user) of each file can determine who else can access it by setting the file's permission bits. Someone who can read a file can also make a copy of it and then let everyone read it. MAC models do not permit such discretionary decisions.

1.4.2 Access to Data and Information

So far, the discussion of access control has focused on models that specify explicit access to data, such as directly reading the content of a file stored on the file system. However, information can be transmitted implicitly, as experiences of human life can testify. In particular, cooperating parties can communicate through socalled *covert channels*, as compared to *overt channels*. For example, if two parties share the same disk partition and one party fills the disk to its full capacity, the other party can notice this fact when a new file creation fails due to lack of space. By filling or not filling the disk, one party can transmit a 1 or a 0 to the other party.

Investigation of this scenario began with Lampson's paper on the confinement problem [70]. In that paper, Lampson discussed the difficulty of restricting an application so that it cannot affect the outside world either directly or by transmitting information.

How critical this type of attack is hinges on the level of one's fear of infiltrationby the enemy and on the perceived potential for severe damage that an insider can cause. The mode of insider attack has a long tradition. The fall of Troy eventually led to the term "Trojan horse," which in the computer security field means any program that is planted on one's machine for the purpose of causing harm.

Early research into the confinement problem led to security models that are based on information flow instead of data access. In particular, the models put forward by Goguen and Meseguer served as the basis for extensive theory work in this area [37, 38]. Also, practical studies of covert-channel communication in real systems have been done. For example, a team at Digital Equipment Corporation constructed a case study in which two parties share the same disk. By placing files in strategic locations, one party can selectively read one file or another, which will cause a detectable delay when another party tries to read a third file. The delay is caused by the speed of the disk-arm movement, and the two different delay values can be interpreted as 1 and 0. The value of such practical studies is mostly in determining the capacity, and therefore the usability and threat, of covert channels. For example, the disk-arm covert channel is usually a lot faster than the fillupdisk-partition covert channel.

Note that the practical utility of covert channels is difficult to gauge. First, there is always the possibility of noise. In the disk-arm case, for example, a third party independently accessing various files on the disk could significantly reduce the bandwidth of the covert channel. However, for very secret materials, such as cryptographic keys, a slow covert channel is adequate for leaking those secrets. Second, covert channels are exploitable only when one can plant Trojan horse programs. When such penetration occurs, other forms of communication that are easier to exploit are often possible.

Moreover, defense against covert channels is effective only "within the system." For example, a computer system that does not allow an insider to signal to the outside world cannot prevent the

insider from memorizing the secrets and walking out with them. Nevertheless, some organizations, especially the U.S. government, take covert channels seriously. For example, researchers at the Naval Research Laboratory have been developing an extensive system, called the "Pump," for the sole purpose of transmitting information with no or limited leakage of information through covert channels. The Java platform does not comprehensively address the presence of covert channels.

1.4.3 Static versus Dynamic Models

At first glance, a security policy appears static. For example, an employee either can or cannot read file *x*. There is no third way, and that is that. But in reality, security policies are dynamic; they can change over time. When that employee transfers into a different department in the organization, she might then be given access to a file to which she was previously denied access. In the MAC model, the sensitivity level of the data and the clearance level of people can also change. Data can be upgraded or downgraded, or a person might gain or lose a particular level of security clearance.

Several notable security models exhibit this dynamism. One is the High-Watermark model [72], in which the sensitivity level of data keeps moving up according to the clearance level of the person who has had access to the data.

Another is the Chinese Wall model [18], which models the practice, especially in consulting firms and financial institutions, of erecting a Chinese Wall to avoid conflicts of interest. For example, a consultant in the oil industry is available to consult with oil company A or B, both of which are clients of the firm. Thus, the consultant potentially can access materials related to either A or B. However, once the consultant accesses A's materials, access to B's will be denied, due to conflict of interest. The Chinese Wall model attempts to represent such real-life policies.

A dynamic model that has its root in the financial industry is the Clark-Wilson integrity model [23], which can be used to model the security requirements for performing financial transactions. For example, transactions over a certain monetary limit must be cosigned by two people and in a particular order. This model was the first widely cited security model that clearly demonstrated the need for security models beyond those of interest to the military and to government agencies, which were primarily MAC security models.

1.4.4 Considerations for Using Security Models

A model can be used, for example, to drive or analyze the design of a computer system or to form the basis of a system's operation. These practical uses of models result in a number of interesting issues that have been studied to various degrees.

First is *decidability*. That is, when given a general security model of a real system and a particular requirement or condition of security—for example, an employee must not be allowed to access file *x* directly or indirectly—can you decide whether a system is secure? The answer to this question is no in the general case (see [51]). Later research to resolve this issue has primarily involved efforts to restrict the model's generality so that the issue becomes decidable. In such models, the computational complexity to answer the question is still NP-complete (Nondeterministic Polynomial time complete) [113].

The second issue is that it is often impossible or infeasible to model, specify, or analyze a system because a practical system tends to be fairly large. This has led to work with *composability*. Here, a security model is constructed such that if various components satisfy some set of security properties and are connected in some particular ways, the overall system automatically—via mathematical proof—satisfies another set of security properties [79, 88]. In practice, the ability to develop secure and composable systems is in the somewhat distant future.

Third, the need to retrofit security mechanisms into so-called legacy systems, or at least to connect systems together securely, means that the legacy systems must be securely *interoperable*. One definition of secure interoperability is that the security properties of each existing system must be preserved under their original definitions. However, in this case, deciding whether a particular interoperation is secure is often NP-complete [46] even under very simple models.

Finally, security does not mean only confidentiality. Modeling the integrity of a system is also critical. Examples of *integrity* models are the Bell-LaPadula confidentiality model described in Section 1.4.1 and an early integrity model by Biba [13], which is the mirror image of the Bell-LaPadula model; that is, it is a *read-up* and *write-down* rather than a *read-down* and *write-up* model. One can also view integrity as an aspect of dependability or correctness and thus can enlist the help of results from the field of fault tolerance.

1.5 Using Cryptography

Whereas cryptography pertains to the encoding and decoding of information, *cryptanalysis* is the reverse of cryptography and is the art of decoding, or breaking, secretly encoded information without knowledge of the encryption keys. The term *cryptology*, or crypto, refers to the whole subject field.

Security and cryptology are related but different fields, and many people confuse them. Both fields are orthogonal in the sense that each has its own utility without depending on the other, although technology from one can help the other. For example, all the security models discussed so far do not need to use crypto at all. Crypto can be used to enhance confidentiality and integrity and can be studied in the abstract, without reference to computer security. However, modern crypto exists largely in the context of a computer and a communications system, in which such features as access control are useful in protecting the access to cryptographic keys. In fact, the easiest way to attack a cryptosystem is to try to compromise its key-storage facility.

Among the most commonly used crypto concepts are

- One-way hash functions
- Symmetric ciphers
- Asymmetric ciphers

These concepts are discussed in the following subsections.

Another note about cryptosystems in general is that all, with the exception of one, are theoretically insecure, according to theorems by Claude Shannon, in the sense that an enemy with sufficient knowledge and computing power can always break the cryptosystem. The only exception is a system called one-time pad, in which the secret key is as long as the plaintext itself and is never reused. A one time pad system is practical only when the sender and recipient have a secure way to exchange the potentially very large key.

Comprehensive coverage of cryptography is available in the *Handbook of Applied Cryptography* [82]. For readers who do not want to dive into the deep background of cryptography and related research subjects, *Applied Cryptography* [115] may be a more suitable text.

1.5.1 One-Way Hash Functions

A one-way hash function is an important building block to help achieve data integrity. Such functions are often used to protect data both in storage and in transit.

According to Knuth [64], the idea of hashing originated in 1953 with two groups of IBM researchers. The earliest reference we can find to the concept of a one-way function was by

Wilkes in 1968 [133], when he referred to the invention of one-way functions for the Cambridge Time-Sharing Computer System by Needham.[2]

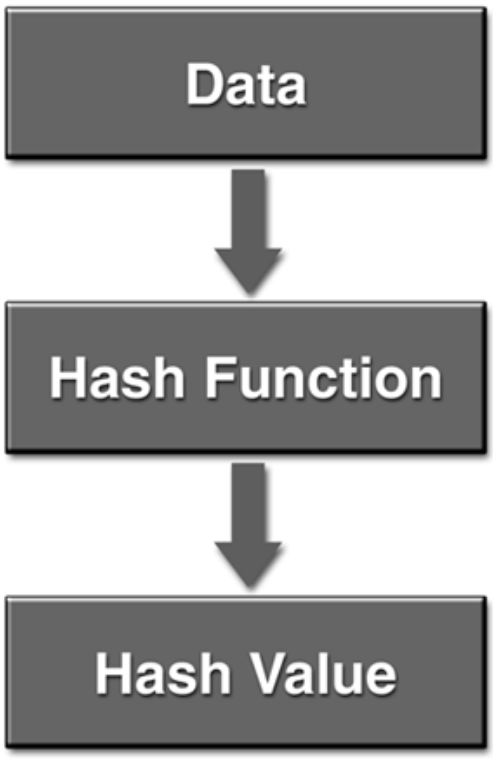
[2] Roger Needham later remembered that the idea was first discussed in The Eagle public house in Cambridge in 1967. He also noted that it is a compliment to the hospitality of the public house that nobody remembers exactly who made the suggestion.

The concept of one-way *hash* functions also dates back many years. A number of researchers, such as Merkle [83], Naor and Yung [90], and Damgård [26], have suggested definitions. Meyer and Schilling [85], Merkle [84], Rabin [98], Rivest [102], and others have presented practical designs for such functions.

Many terms relating to one-way hash functions have been introduced. Some of these terms are alternative names, and some are intended to emphasize differing assumptions. Examples are one-way (hash) function, collision-free (hash) function, fingerprinting function, modification detection code, and message authentication code.

Informally, a *one-way hash function* is a function that is easy to compute but difficult to reverse. Also, it is difficult to find two input values for which the function would compute the same output value. Such properties allow the protection of integrity as follows. Suppose that you store a file on a disk and suspect that it might be tampered with. Using the file content as input, you can compute the hash function value, which can be a lot shorter than the file content itself. Later, you can take the current content of the file and feed it into the hash function. If the new hash value is identical to the old hash value, it is highly likely that the file content has not been modified. In this case, the one-way hash function serves as an unforgeable link between the file content and its hash value. Figure 1.4 illustrates one-way hash functions.

Figure 1.4. One-way hash function

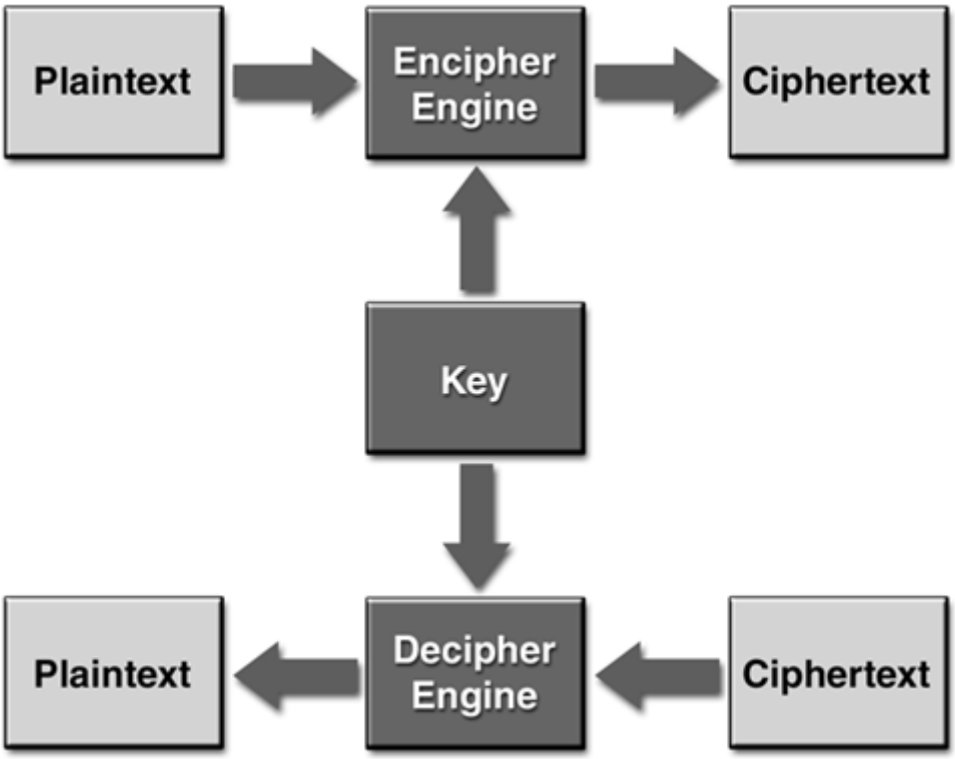


Designers often incorporate secret keys into the inputs of one-way hash functions such that the hash value cannot be correctly computed or predicted without knowing the secret keys. In this case, such a keyed, one-way hash function serves as an unforgeable link not only between the file content and its hash value but also between the secret keys used—and thus the entities that possess the keys—and the hash value.

1.5.2 Symmetric Ciphers

A symmetric cipher is a transformation, operated under a secret key, that can translate its input, called *plaintext*, to its output, called *ciphertext*, in such a way that, excluding cryptanalysis, only those entities possessing the secret key can recover the plaintext from the ciphertext (Figure 1.5).

Figure 1.5. Symmetric cipher



Symmetric ciphers have a long history. Their first known use dates from the early Caesar system [66]. Symmetric ciphers have been widely used; for example, the Data Encryption Standard (DES) [128] has been in use for nearly three decades. DES is no longer considered a strong encryption algorithm, due to advances in cryptanalysis and computing power. Thus, a replacement algorithm has been selected: the Advanced Encryption Standard (AES) [92].

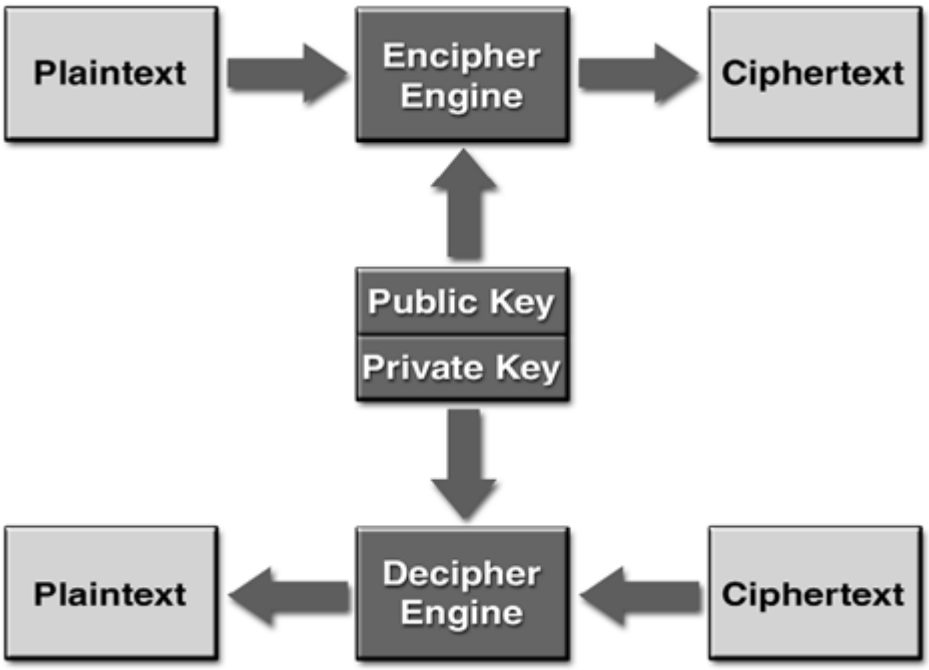
Symmetric ciphers are also called *secret-key ciphers* because the two communicating parties must share a secret key. This requirement creates some difficulties in key management and key distribution. Moreover, because each pair of communicating parties must share a distinct secret key, in theory an exponential number of secret keys is needed when a large group of parties talk to one another.

Symmetric ciphers can be operated in different modes, such as various feedback modes. Symmetric ciphers can also be stacked to improve the crypto strength of the whole system, such as in the case of triple-DES.

1.5.3 Asymmetric Ciphers

An asymmetric cipher is similar to a symmetric cipher, but instead it depends on a pair of keys rather than on only one key. The *public key* of the pair is used to encrypt plaintext. The *private key* of the pair is used to decrypt ciphertext. See Figure 1.6. The keys are generated such that it is easy to deduce the public key, given the private key; the reverse, however, is very difficult. This property enables people to exchange their public keys over public channels and still conduct private communications. Compare this with symmetric cipher systems, in which people must arrange a shared secret key via a private channel. Notable asymmetric systems include Diffie-Hellman [29] and RSA [104]. Asymmetric systems are often used to encrypt and exchange keys for symmetric systems.

Figure 1.6. Asymmetric cipher used for encryption and decryption



A distinct property of some asymmetric systems is that the encryption and decryption are *reversible*. This means that one can apply the decryption operation with the private key to the plaintext to get ciphertext, and one can recover the plaintext by applying the encryption operation with the public key to the ciphertext. In this case, because the public key is public, no confidentiality protection is provided. However, because only the holder of the private key can generate the ciphertext with these systems, the ciphertext can serve as a *digital signature* of the plaintext, and anyone with the public key can verify the authenticity of the signature. Thus, for example, Alice can sign a message M by applying the decryption operation to it using her private key to get S_M. She then sends M and S_M to Bob. Bob verifies that S_M is a valid signature for M by applying the encryption operation to S_M using Alice's public key. If the result is equal to M, the signature is valid, and Bob knows that the message came from Alice.

RSA (named for its creators—Rivest, Shamir, and Adleman) is perhaps the most widely used asymmetric system that can also be used to produce digital signatures. Another system, Digital Signature Algorithm (DSA) [91], can perform only digital signature functions; it cannot be used for encryption.

To prove that it is the real owner of a public key, one party can present a certificate for verification by the other party. A public-key certificate is a digitally signed statement from one entity, saying that the public key and some other information of another entity have some specific value. A

chain of certificates is possible, whereby each certificate contains a public key that is used to certify the public key in the succeeding certificate. The first certificate, often called the root certificate, does not have another public key to certify it. Thus, it normally is a *selfsigned certificate* in that its own public key is used to certify itself. Later chapters, especially Chapter 8, have more in-depth discussion about certificates.

1.6 Authentication

Another basic security issue is authentication. Authentication is the process of confirming the identity of an entity (a user, a machine, or a machine operating on behalf of a user). Authentication first became an issue when time-sharing systems began to be deployed and the system needed to know the identity of a user logging in to the system. This knowledge is critical for enforcing access control policies, as most of the security models mentioned previously are based on granting access to certain users and not to others.

The importance of authentication increased when networked computer systems started to surface. The network often is shared or public, so it is crucial to authenticate, or know the identity of, the user at the other end of the wire. It is equally important for the users to know the identity of the system they are connecting to.

Numerous authentication protocols exist, but many of them have subtle security flaws, discovered even after many years of scrutiny by experts. As a result, authentication has become a major study subject.

The basic approach is first to ask the user at the other end of the wire to present a name and a password and then to check these against system records. Such a simple-minded solution, which amazingly is still widely used when more secure solutions are available, is vulnerable to eavesdropping and guessing attacks [78]. Anyone who is monitoring network traffic can learn the password and use it later. Variations of this approach exist, such as one-time passwords [68] and an Internet Engineering Task Force (IETF) standard called OTP (One Time Password), which evolved from S/Key [50]. These are an improvement with limitations, because one can carry only a limited number of one-time passwords.

This basic approach can be generalized to one based on challenge and response. This approach can also be extended to perform the function of key distribution such that different entities need to share keys only with certain designated key-distribution centers. These centers can dynamically establish secret keys between any set of such entities that previously might not have communicated to each other. The earliest work in network-based authentication is the well-known Needham-Schroeder protocol [94], as illustrated in Figure 1.7. With such a protocol, two entities, referred to as Alice and Bob in the figure, can use the authentication server as a trusted third party to establish a short-term secure session. This protocol is the basis of the Kerberos system implemented as part of the MIT Project Athena and later adopted as part of the Open Group DCE (distributed computing environment) Security Service and as an IETF standard [87, 95].

Figure 1.7. Use of authentication server

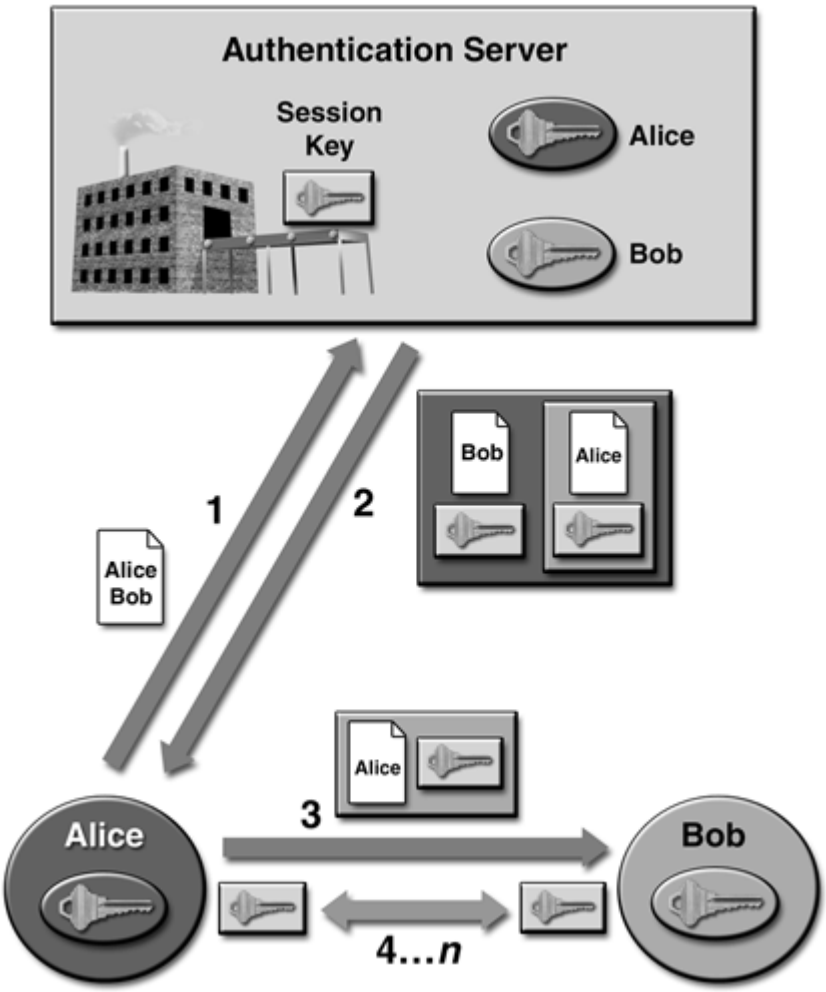


Figure 1.7 is a simplified representation of the use of an authentication server. In the figure, it is assumed that the authentication server shares a secret key with Alice and a different secret key with Bob. Whenever Alice and Bob want to communicate securely with each other, they can obtain a session key for that communication from the authentication server. A description of the steps depicted in the figure follows.

1. Alice sends to the authentication server a message with her name and Bob's name.
2. The authentication server creates a session key, creates a message for Bob with Alice's name and the session key, and encrypts this message, using the secret key it shares with Bob. The authentication server then encloses this message in another message it creates for Alice and encrypts this message with the secret key it shares with Alice. In addition to the message for Bob, this message includes Bob's name and the session key. The authentication server sends this message to Alice.
3. Alice decrypts the message, using her secret key. Now she has the session key and a message she can send to Bob so that he'll also know what the session key is. Alice sends this message to Bob.
4. Bob decrypts the message, using his secret key. Now he knows what the session key is, and Alice and Bob can securely exchange messages encrypted with the session key.

Protocol design is full of peril. The Needham-Schroeder protocol, among many others, was later shown to be defective in a number of aspects [19, 45]. Attacks on security protocols include replay attacks and interleaving attacks, whereby an attacker listens and records legitimate network traffic and then reuses these messages—sometimes after some skillful modifications—to defeat security. But these types of attacks can easily slip a protocol designer's mind and thereby lead to the possibility of attack later. As a result, formal and informal protocol analysis techniques have been

suggested and applied [19, 30, 45, 81, 86], including the fairly recent application of model-checking tools.

One especially serious issue involves authentication protocols designed for use by human beings. These protocols usually involve the use of passwords that people can remember. This approach has the disadvantage that such passwords are generally chosen from a fairly small space, such as all words in a dictionary, that can be mechanically searched and thus easily deduced. As examined by the security research community, all the authentication protocols that were published prior to 1989 suffer from this problem of easily guessed passwords. As a result, an attacker who has monitored the network traffic and obtained a running record of an authentication protocol can then guess each candidate password and verify whether the guess is correct, all offline and thus undetectable. Technical solutions to this problem started to appear in late 1989 [43, 78] and include EKE (Encrypted Key Exchange) and A-EKE (Augmented Encrypted Key Exchange) [10, 11]. Smart cards and other hardware-based security devices are often helpful in avoiding use of guessable passwords. Hardware tokens exhibit much stronger controls against guessing attacks by adding a second physical factor in the authentication, something possessed. Biometric devices can be used to add yet a third factor to the authentication: something inherent (that is, a personal physical characteristic).

1.7 Mobile Code

Mobile code is not a fundamentally new concept; anything that causes a remote system to behave differently can in theory be viewed as mobile code. Thus, the whole field of distributed computing works on the premise of mobile code. This includes data, such as Domain Name Service (DNS) information; remote commands, such as Remote Procedure Call (RPC); and executable scripts, such as remote shell on UNIX. This section focuses on the last category: executable scripts, code that travels from one machine to another and gets executed as it travels. Such mobile code is widespread, partly because it helps to distribute the computation load among client as well as server machines and partly because it helps to reduce demand on network bandwidth.

PostScript files belong to this category because when a PostScript file is displayed and viewed, it is the file content that is being executed. The same is true for Microsoft Word documents that contain macros; the macros are interpreted as the document is read. Another example is LISP. Many people read their e-mail from within EMACS, a powerful text editor. EMACS interprets LISP programs as it sees fit, so a LISP program segment embedded in an e-mail message can become active when viewed inside EMACS. Other kinds of active components include ActiveX controls and Java technology–based applets (Java applets).

Active contents do not pose a new category of threat. Instead, they help expose the inadequacies of commonly deployed security mechanisms. For example, when mobile code is a DNS update request, the interface is fairly narrow so that its security implication is more easily understood. However, when fully general mobile code, such as an ActiveX control, arrives, the interface becomes the entire Win32 APIs (Application Programming Interfaces), and any security holes in those APIs might be exploited.

The increasing use of mobile code has resulted in two responses. On the one hand, people try to enhance system security to better control and thus use the attractive aspects of mobile code. On the other hand, people get scared and want to block mobile code at their perimeters. The latter is at best a gapstopper, as mobile code and active contents can travel through multiple channels, such as e-mail, and filtering every e-mail message and removing parts of messages are often unacceptable to the e-mail users. One primary design goal of Java technology is to make it a secure platform for mobile code.

1.8 Where Java Technology–Based Security Fits In

The previous sections provided a broad overview of the large security jigsaw puzzle that today's systems use: from firewalls to access control, from encryption to authentication. Java technology–based security (Java security) is a very important piece of this puzzle because Java technology is pervasive both as a platform-independent technology and as the best vehicle to program mobile code and executable content for the Internet and the Web. The rate of adoption of Java technology is phenomenal. It is being deployed, for example, in financial institutions, in online e-commerce software, and as part of other critical infrastructure applications. Therefore, the Java platform must fulfill its promise as a safe Internet programming platform.[3]

[3] We should make it clear that JavaScript is not based on the Java programming language and is related to it only by name. JavaScript does not have the comprehensive security considerations and mechanisms that the Java platform has.

The Java platform can be viewed as a client-side application, such as when running it inside a browser; a server-side application, such as when running server software programmed in the Java programming language; or an operating system, such as when running the JavaOS directly on MS-DOS or bare hardware. Because different usage scenarios might require different or even conflicting security features, the Java 2 SDK (Software Development Kit) is designed to build in common functionality while leaving sufficient hooks so that it can be extensible to handle specific requirements.

When Java technology is available within an operating system, such as Solaris, Linux, or Microsoft Windows (Windows), its presence does not alter the basic security characteristics of the underlying system. On Solaris, for example, an instance of the virtual machine for the Java platform (Java virtual machine, or JVM) will have access only to resources that would be available to the user running the JVM. However, if the entire application interface is limited to Java APIs, usually the overall system security is improved.

Finally, security features on the Java 2 platform are not limited to what is available in the current shipping version. Further versions no doubt will continue to enrich the security features. Java technology is becoming not just one but rather many pieces of the security puzzle. Just as SSL/TLS (Secure Sockets Layer/Transport Layer Security) [28] and the browser finally brought cryptography to the mass market, Java technology has played an important role in pushing computer security into the technology mainstream.

Chapter 2. Basic Security for the Java Programming Language

Never forget class struggle.

—Mao Ze-Dong

Since the inception of Java technology [48, 74], strong and growing interest has centered on its security. Security has been publicized as one of its critical design goals and cited as a significant means of differentiating Java from other technologies.

A new technology rarely includes reasonably good security features in its initial-release. Thus, the positioning of Java as the best platform for secure Internet programming has attracted a lot of attention from both security professionals and the computer industry in general. Long-time security researchers, academics, and students have poured over design details and source code of the SDK, which was released by Sun Microsystems for just such purposes. Even the popular media have caught the frenzy; both the *Wall Street Journal* and the *New York Times* have covered it prominently.[1]

[1] Refer to [80] for some quotes and citations.

From a technology provider's point of view, Java security provides two features [41]:

- Primarily through the SDK, the Java platform as a secure, ready-made platform on which to run Java technology–enabled applications in a secure fashion
- Security tools and services, implemented in the Java programming language (Java language), that enable a wider range of security-sensitive applications

The deployment of Java technology also raised an array of interesting security issues, which are covered in later chapters. This chapter focuses primarily on the basic security features provided by the Java language and platform.

2.1 The Java Programming Language and Platform

The Java programming language [3] was designed originally for use in embedded consumer electronics applications, such as hand-held devices and set-top boxes. It is a general-purpose object-oriented programming language and is simple enough that many programmers can become fluent in it fairly quickly. It is specifically designed to be platform independent so that application developers can write a program once and then run it securely everywhere on the Internet. It is related to C and C++, but it is rather different, with a number of aspects of C and C++ omitted and a few ideas from other languages included.

The Java language is strongly typed. It does not include any unsafe constructs, such as array accesses without index checking, because such unsafe constructs might result in unspecified and unpredictable program behavior.[2] The Java language comes with automatic storage management, typically done by a garbage collector. Further, the Java language avoids the safety problems, such as those posed by C's `free` or C++'s `delete`, concerning the explicit deallocation of memory that is no longer needed.

[2] A study concluded that about 50 percent of all alerts issued by the Computer Emergency Response Team (CERT) are caused in part by buffer-overflow errors.

A program written in the Java programming language (Java program) is normally compiled to a bytecoded instruction set and binary format defined in the Java Virtual Machine Specification

[48]. The Java language also defines a number of packages for more complete programming support. A Java program is normally stored as binary files representing compiled classes and interfaces. The binary class files are loaded into a JVM and then linked, initialized, and executed. Here is an example of a simple program:

```
class Test {
   public static void main(String[] args) {
      for (int i = 0; i < args.length; i++)
         System.out.print(i == 0 ? args[i] : " " + args[i]);
      System.out.println();
   }
}
```

The class `Test`, stored in the file `Test.java`, can be compiled and executed by giving these commands:

```
javac Test.java
java Test Hello
```

The program will print out `Hello`.

The Java platform is network-centric and is born of the idea that the same software should run on many different kinds of computers, consumer gadgets, and other devices, such as smart cards. With Java technology, you can use the same application on a Sun Enterprise 15K running the Solaris operating system as on a personal computer (PC) running Windows.

The HotJava browser demonstrated the Java platform's power by making it possible to embed Java programs inside HTML pages. These programs, called *applets*, are transparently downloaded, to be run inside the browser. The Java platform has been integrated into most popular Web browsers. Java technology is also embedded in smart cards, cell phones, TV set-top boxes, and game consoles. Thus, Java programs need not depend on browser integration.

The Java platform consists of the Java language, the JVM, and the application programming interfaces (API libraries). The JVM is an abstract computing machine and does not assume any particular implementation technology or host platform. The JVM also knows nothing of the Java programming language but instead knows only of a particular file format: the *class file format*. A class file contains JVM instructions, or bytecodes, and a symbol table, as well as ancillary information. Bytecodes can be either interpreted or compiled for a native platform. The JVM may also be implemented either in microcode or directly in silicon.

2.2 Original Basic Security Architecture

In the original (1.0) release of the Java platform, the basic security architecture centered on allowing a user to import and run Java applets dynamically without undue risk to the user's system. An *applet* is loosely defined to be any code that does not reside on the local system and must be downloaded to be run. Code that does reside on the local system is commonly called a *Java application*, that is, a Java technology–based application. Because applets are downloaded dynamically and often without your awareness and because you may not know who the applets' authors are, you cannot blindly trust an applet not to be malicious. Thus, a downloaded applet's actions are restricted to a *sandbox*, an area of the Web browser allocated specifically to the applet. The applet may play around within its sandbox but cannot reach beyond it. For example, the applet cannot read or alter any file stored on the user's system. In this way, if a user accidentally imports a hostile applet, it cannot damage the user's system. Thus, this sandbox model provides a very restrictive environment in which to run untrusted code obtained from the open network. The

sandbox model is deployed through Java Development Kit (JDK) 1.0.*x* and is generally adopted by applications built with the JDK, including Java technology–enabled Web browsers.

In the original Java platform, all applications—as opposed to applets—are completely trusted to have full access to vital system resources, such as the file system. Security comes from maintaining physical control over the systems by, for example, preventing end users from installing hostile software. Note that the distinction between an applet and an application, or "outside" versus "inside," is not always absolute. With a network file system, a class file appearing to reside on the local file system might in fact be located thousands of miles away, whereas an applet can be downloaded from within the local area network (LAN), possibly from the same host on which the user is running it.

The original basic security architecture is enforced through a number of mechanisms. First, the Java language is designed to be type safe and easy to use. Thus, the programmer is less likely to make subtle mistakes, compared with those possible when using other programming languages, such as C or C++. Automatic memory management, garbage collection, and range checking on strings and arrays are examples of how the Java language helps the programmer to write correct and safer code.

Second, a bytecode verifier ensures that only legitimate code written in the Java programming language (Java code) is executed. A compiler translates Java programs into a machine-independent bytecode representation. Before a newly downloaded applet is run, a bytecode verifier is invoked to check that the applet conforms to the Java Language Specification and that there are no violations of the Java language rules or namespace restrictions. The reason is that, for the sake of security, the JVM imposes strong format and structural constraints on the code in a class file. The verifier also checks for violations of memory management, stack underflows or overflows, and illegal data type casts. Without these checks, a hostile applet could corrupt part of the security mechanism or replace part of the system with its own code. The bytecode verifier, together with the JVM, is designed to guarantee language type safety at runtime. To ensure complete type safety, for example, the JVM uses a runtime type check when storing references in arrays.

Runtime activities include the loading and linking of the classes needed for execution; optionally, machine code generation and dynamic optimization; and the program execution. During this process, a class loader defines a local namespace, which is used to ensure that untrusted code cannot interfere with the running of other Java programs.

Finally, access to crucial system resources is mediated by the JVM and is checked in advance by a security manager class that restricts to a minimum the actions of untrusted code. Class loader and security manager classes are discussed in greater detail later, in Chapters 4 and 6.

2.3 Bytecode Verification and Type Safety

Although a trustworthy compiler can ensure that Java language source code does not violate safety rules, someone could use a rigged compiler to produce code that does violate them. A Java technology–enabled Web browser that can import code fragments from anywhere does not know whether a code fragment comes from a trustworthy compiler. Thus, before executing any code fragment, the runtime system subjects it to a series of tests.

The tests range from verifying that the format of the fragment is correct to passing it through a simple theorem prover to establish that the code plays by the rules. Approximately, the code is checked to ensure that

- It does not do illegal data conversions, such as converting integers to pointers.
- It does not forge pointers. (There are no pointers in the Java language.)

- It does not violate access restrictions. For example, a private field should not be accessible from outside the object.
- It accesses objects as what they are. (For example, the tests ensure that `InputStream` objects are always used as `InputStream`s and never as anything else.)
- It calls methods with appropriate arguments of the appropriate type, there are no stack overflows, and methods return objects of the appropriate type.

Note that a static bytecode verifier is not strictly necessary to ensure type safety, because the JVM can, in theory, perform complete type checking during runtime. However, runtime checks often slow down the execution of a program significantly because such checks have to be done repeatedly for each method invocation. Thus, moving some checks up front to class loading time, where those checks are done only once, seems an appealing strategy. Knowing that any downloaded code satisfies these properties makes the runtime system operate much faster because it does not have to check for them. Note that the verifier is independent of the Java language or compiler and so can also examine bytecode that is generated from source languages other than Java.

The preceding five checked points do not tell the whole story and are not meant to be formal or precise. Space limitations do not permit a description of the considerable work that covers the finer details of the Java language design, the inner workings of the JVM, the background of flow analysis, and the art of theorem proving, all of which are necessary background for a complete understanding of how type safety is enforced.

For the present discussion, it is sufficient to understand the following points. The most fundamental goal of the Java security architecture is to ensure that the Java Language Specification and the Java Virtual Machine Specification are observed and implemented correctly. One way to think about this is to imagine that you are writing a calendar application. You typically will have interfaces that expect to take an integer between 1 and 12 to represent a month within the year. You might also have an initialization interface that prompts the user to type in the current month. Because your other interfaces assume that the month integer will be between 1 and 12, it is prudent that you check and ensure, from inside the initialization procedure, that the input is indeed a valid number. If you do not check for this and as a result do not reject invalid numbers, your calendar application might not work with an out-of-range month number and might behave in strange ways.

This same principle applies to the Java platform. The JVM expects the bytecode that it runs to have certain properties, and it is the job of the bytecode verifier to ensure that those properties are met. The JVM also decides to check additional properties itself, perhaps because these are difficult or impossible to analyze statically by the bytecode verifier. There is no mystery in ensuring type safety, just precise and judicious application of well-known principles.

You might ask what type safety has to do with computer security. Type safety contributes to program correctness. If a program that implements security functionality does not accomplish what is intended because the program cannot be correctly executed, the security functionality may not be provided. For example, a security decision may be embodied in an equality test of the following form:

```
if (name.equalsIgnoreCase("Chuck Jones")) {
   openDoorToHackersLounge()
} else {
   throwThemOut()
}
```

Here, security reasoning is written and performed in the Java language. Thus, it is critical that a yes answer is not possible when one string, such as `"Chuck Jones"`, is compared to a

different string, such as `"Tex Avery"`. Otherwise, a trivial incorrectness in string comparison leads to a security hole.

On the other hand, it is important to note that not all type-safety problems inevitably result in a security breach. For example, if a virtual machine implementation has a single bug that equates string `"acegikmoqsuwy"` with string `"bdfhjlnprtvxz"`, what security compromise this will cause is not immediately clear. Nevertheless, the type-safety issue needs close attention and should not be left to chance.

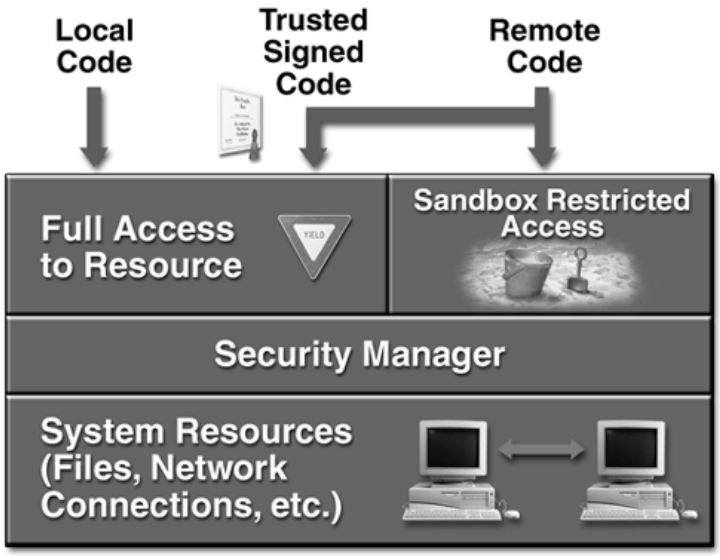
Yellin, in an early paper, included some details of the verifier and other typechecking mechanisms [135]. However, you need a fairly good understanding of the bytecode instructions in order to digest them fully. More recently, Liang and Bracha wrote about a new mechanism, implemented in the Java 2 SDK, Standard Edition (J2SDK), 1.2, that solves a problem with type safety regarding dynamic class loading [73]. This subject of bytecode verification is still evolving, with ongoing work occurring within the J2SDK development team, as well as at research labs and universities. A more formal and precise exposition of the entire language type-safety subject is anticipated for the future.

2.4 Signed Applets

JDK 1.1 introduced the concept of *signed applets*. Recall that in the original sandbox model, all remote code—that is, all applets—are automatically untrusted and are restricted to running inside the sandbox. Such restrictions, although contributing to a safe computing environment, are too limiting. Within a LAN, for example, a company might deploy an applet that is used to maintain employee pension data. An employee who downloads and runs the applet to change the plan allocation would want the applet to automatically update his own accounting record stored in his own file directory.

To facilitate such features, JDK 1.1 added support for digital signatures so that an applet's class files, after their development, could be signed and stored together with their signatures in the JAR (Java Archive) format. For each JDK installation, you can specify which signers, that is, which public keys, are trusted. When a correctly digitally signed applet is downloaded, and if its signers can be verified and recognized as trusted, the applet is treated as if it is trusted local code and is given full system access (Figure 2.1).

Figure 2.1. JDK 1.1 security model



2.5 Further Enhancements

Both the original sandbox model and the trusted-applet model were extended into the security architecture introduced in Java 2 SDK, Standard Edition (J2SDK)[3], 1.2. This security architecture implements fine-grained access control based on security policies and permissions. An overview of this architecture is given in Chapter 3 and covered in greater detail in Chapters 4 through 6.

[3] *Note*: A terminology shift occurred in December 1998. The Sun Microsystems product that implements the Java 2 Platform is referred to using "Java 2 SDK, Standard Edition," rather than "JDK."

Chapter 3. Java 2 Security Architecture

The state is nothing but an instrument of oppression of one class by another.

—Friedrich Engels

The need to support flexible and fine-grained access control security policies, with extensibility and scalability, called for an improved security architecture. The architecture introduced with the Java 2 Platform, Standard Edition (J2SE), 1.2, fulfills this goal. This chapter details why the security architecture changes were needed and then gives an overview of the Java 2 security architecture. The three subsequent chapters provide architecture details. Chapter 4 describes secure class loading, Chapter 5 tells how a security policy is specified and represented, and Chapter 6 tells how the security policy is enforced.

3.1 Security Architecture Requirements of Java 2

As discussed in the previous chapter, it was critical that the original release of JDK 1.0 consider security seriously and provide the sandbox security model. Not many technologies have security as a design goal, so Java technology, together with the Internet and the promise of e-commerce, helped to finally move security technology into the mainstream of the computer industry. Doing so was a significant achievement. The next step was to improve on the enhancements incorporated in JDK 1.1 to make the security solutions on the Java platform easy to use and more robust. The Java 2 security architecture corrects limitations of earlier platform versions.

3.1.1 Flexible Access Control

By default, the sandbox model severely restricts the kind of activities that an applet may perform. Although it was the catalyst that created the atmosphere for safe Internet computing, this model treats all applets as potentially malicious. Thus, some applets, such as those created by a corporation's finance group to handle internal transactions, are also limited in what they can do, even though they are likely to be more trustworthy than an arbitrary applet downloaded from an unfamiliar Web site.

Such a blanket restriction on all applets can be limiting. For example, suppose that a customer of a brokerage firm uses a stock-trading applet loaded from the brokerage's Web site. This customer may want to let the applet update local files that contain her stock portfolio. However, access to the client-side file system is prohibited by the sandbox model. Thus, this customer needs *flexible access control*, whereby certain applets can have access that is outside the sandbox. With JDK 1.1, the brokerage firm could sign the trading applet, and, assuming that the customer configured her Java runtime to recognize the brokerage firm to be a trusted signer, the applet could access resources outside the sandbox.

However, the customer may have installed on her local desktop financial management software that handles income tax issues. She might not feel comfortable letting the brokerage firm's applet have free rein on her entire desktop system. In this case, it may be best to confine the applet to limited file system access, perhaps only to the brokerage firm's file folder. What is needed is a model whereby the sandbox can be customized—for example, by the client system—to have flexible shapes and boundaries: in other words, *fine-grained access control*.

Prior to Java 2, one could, in theory, implement a more flexible and finergrained access control on the Java platform. To accomplish this, however, someone, such as an application writer, had to do substantial programming work by, for example, subclassing and customizing the `SecurityManager`, `ClassLoader`, and other classes. The HotJava browser was an example of such efforts; it had a limited range of user-definable security properties. However,

such extremely security-sensitive programming requires in-depth knowledge of computer security and robust programming skills.

The Java 2 security architecture eliminates the need to write custom security code for all but the most specialized environments, such as the military, which may require special security properties, such as multilevel security [72]. Even then, writing custom security code is simpler and safer than before.

3.1.2 Separation of Policy Expression from Policy Enforcement

As codified by the `java.lang.SecurityManager` class, the sandbox model implements a specific security policy that is expressed in the implementation of the software that does the policy enforcement. This means that to enforce a different security policy, a customized version of the software must be used—clearly, this is not desirable. Instead, what is needed is an infrastructure that supports a range of easily configurable security policies.

The Java 2 security architecture cleanly separates the enforcement mechanism from the description of security policy. In this way, application builders can configure security policies without having to write special programs.

3.1.3 Flexible and Extensible Policy Enforcement

Prior to Java 2, the Java security architecture hard coded the types of security checks performed. For example, to check whether a file can be opened for reading, you would call the `checkRead` method on the currently installed `SecurityManager`. Such a design is not easily extensible, because it does not accommodate the handling of new types of checks that are introduced as after-market add-ons to the Java runtime. It is also not very scalable. For example, to create a new access check, such as one that checks whether money can be withdrawn from a bank account, you would have to add a new `checkAccountWithdraw` method to the `SecurityManager` class or one of its subclasses. Thousands of various kinds of checks are possible. If methods were created for this large a number, they would clutter the `SecurityManager` class. In fact, because many checks are application specific, not of all them can be defined within the JDK. What is needed is an easily extensible access control structure.

To that end, the Java 2 architecture provides typed access control permissions and an automatic permission-handling mechanism to achieve extensibility and scalability. In theory, no new method ever needs to be added to the `SecurityManager` class. Thus far, throughout the multiple J2SE releases, we have not encountered a situation requiring a new method. Instead, the more general `checkPermission` method added in Java 2 has proved sufficient to handle all security checks. See Chapter 6 for more information.

3.1.4 Flexible and Customizable Security Policy

JDK 1.*x* had the built-in assumption that all locally installed Java applications were fully trusted and therefore should run with full privileges. As a result, the sandbox model applied only to downloaded unsigned applets. However, software installed locally should not be given full access to all parts of the system. For example, often a user installs a demo program on the local system and then tries it out. It is prudent to limit the potential damage such a demo program could cause, giving it less than full system access. In another example, caching applets on the local file system will improve performance, but caching should not change the security model by treating cached applets as trusted code, even though they now reside on the local system. Furthermore, the distinction between what is local code and what is remote code is blurry. In the modern world of software components, one application could use multiple components, such as JavaBeans, that

reside in all corners of the Internet. So security checks must be extended to all Java programs: to include applications as well as applets.

In Java 2, all code—whether local, remote, signed, or unsigned—is subjected to the same security controls. Thus, users can choose to give full or limited system access, based on the properties of the code and who is running the code. Such a choice is expressed by configuring a suitable security policy.

3.1.5 Robust and Simple Internal Security Mechanisms

In JDK 1.0 and JDK 1.1, a number of internal security mechanisms were designed and implemented, using techniques that were rather fragile. Although they worked reasonably well, maintaining and extending them proved difficult. For Java 2, we made important internal structural adjustments to reduce the risks of creating subtle security holes in the Java runtime and application programs. This involved revising the design and implementation of the `SecurityManager` and `ClassLoader` classes, as well as the underlying access control mechanism.

3.2 Overview of the Java 2 Security Architecture

The security architecture introduced in the Java 2 platform uses a security policy to decide which individual access permissions are granted to running code. These permissions are based on the code's characteristics, such as who is running the code, where it is coming from, whether it is digitally signed, and if so by whom. Attempts to access protected resources invoke security checks that compare the granted permissions with the ones needed for the attempted access. If a security policy is not explicitly given, the default policy is the classic sandbox policy as implemented in JDK 1.0 and JDK 1.1. The various caveats, refinements, and exceptions to this model are discussed in later chapters.

The Java 2 security architecture does not invent a new computer security theory, even though we had to design new ways of dealing with many subtle security issues that are unique to object-oriented systems. Instead, it offers a real-world example in which well-known security principles [36, 96, 127, 111] are put into engineering practice to construct a practical and widely deployed secure system.

Chapters 5 and 6 describe the details of the policy enforcement implementation classes. The major components of the security model are security policy, access permissions, protection domain, access control checking, privileged operation, and class loading and resolution. Security policy and access permissions define what actions are allowed, whereas protection domain and access control checking provide the enforcement. Privileged operation and class loading and resolution are valuable assistants in the overall protection mechanisms.

Beyond the security model itself, we describe the implementation classes that support authentication, integrity protection, and confidentiality controls. Additionally, we provide the details of how to customize the security architecture to enhance or modify the context evaluated when making a security decision. We describe best practices to minimize the risk of introducing a security vulnerability in custom code, and we tell how to deploy the Java runtime to ensure that it too is secure from unauthorized modification.

3.3 Architecture Summary

As a summary of the overall process of how the Java 2 security architecture works, this section takes you through the handling of an applet or application. The following steps occur when

viewing an applet, either through a Web browser or `appletviewer` or when running a Java application, possibly from the command line by invoking the program called `java`.

1. A class file is obtained and accepted if it passes preliminary bytecode verification.
2. The class's code source is determined. If the code appears to be signed, this step includes signature verification.
3. The set of static permissions, if any, to be granted to this class is determined, based on the class's code source.
4. A protection domain is created to mark the code source and to hold the statically bound permission set. Then the class is loaded and defined to be associated with the protection domain. *Note:* If a suitable domain has previously been created, that `ProtectionDomain` object is reused, and no new permission set is created.
5. The class may be instantiated into objects and their methods executed. The runtime type-safety check continues.
6. When a security check is invoked and one or more methods of this class are in the call chain, the access controller examines the protection domain. At this point, the security policy is consulted, and the set of permissions to be granted to this class is determined, based on the class's code source and principals, specifying who is running the code. In this step, the `Policy` object is constructed, if it has not been already. The `Policy` object maintains a runtime representation of the security policy.
7. Next, the permission set is evaluated to see whether sufficient permission has been granted for the requested access. If it has been granted, the execution continues. Otherwise, a security exception is thrown. (This check is done for all classes whose methods are involved in a thread. See Chapter 6 for the complete algorithm.)
8. When a security exception, which is a runtime exception, is thrown and not caught, the Java virtual machine aborts.

The delaying tactics described earlier help to reduce start-up time and the footprint of the runtime because objects are not instantiated until they must be used.

There are variations to this flow of actions. For example, in an eager approach, the creation of the `Policy` object and permissions can happen when classes are loaded into the runtime. This was the approach taken prior to Java 2 version 1.4.

The fundamental ideas adopted in the Java 2 security architecture have roots reaching into the past 40 years of computer security research: for example, the overall idea of the access control list [69]. We followed some of the UNIX conventions in specifying access permissions to the file system and other system resources. Most significantly, the design was inspired by the concept of protection domains and the work on handling mutually suspicious programs in Multics [110, 116] and "rights amplification" in Hydra [57, 134].

One novel feature not present in such operating systems as UNIX or Windows is the implementation of the least-privilege principle by automatically intersecting the sets of permissions granted to the protection domains involved in the call sequence. In this way, a programming error in system or application software is less likely to have an exploitable security hole.

Note that although it typically runs over a host operating system, such as Solaris, the Java virtual machine also may run directly over hardware, as in the case of the network computer JavaStation running JavaOS [101]. In general, to maintain platform independence, the Java 2 architecture does not depend on security features provided by an underlying operating system.

Furthermore, this architecture does not override the protection mechanisms in the underlying operating system. For example, by configuring a fine-grained access control policy, a user may grant specific permissions to certain software. This is effective, however, only if the underlying operating system itself has granted the user those permissions.

The protection mechanisms in Java 2 are language based and carried out within a single address space. This departure from more traditional operating systems is related to work on software-based protection and safe kernel extensions—for example, [12, 21, 117]—whereby research teams have targeted some of the same goals but by using different programming techniques. In a typical operating system, a cross-domain call tends to be quite expensive. In Java 2, a cross-domain call is just another method invocation.

The following are significant benefits of Java 2 platform security:

- The content of the security policy is totally separated from not only the implementation mechanism but also the interfaces. This leaves maximum room for evolution. It also allows the policy to be configured entirely separately from the runtime environment, thus reducing the complexity of system administration.
- The access control algorithm is cleanly separated from the semantics of the permissions it is checking. This allows the reuse of the access controller code with—perhaps application-specific—permission classes that are introduced after Java 2's release.
- The introduction of a hierarchy of permission classes brings the full power of object orientation, and especially encapsulation, to bear. This means that access control permissions can be expressed both statically and dynamically and that each `Permission` class may define its own semantics: for example, how it relates to a permission of its own type or of a different type or how to interpret wildcard symbols and other peculiarities that are specific to it.
- The secure class loading mechanism, coupled with the delegation mechanism, extends security coverage to Java applications, thus resulting in a uniform security architecture and policy for any and all Java code, whatever its origin or status.

3.4 Lessons Learned

Over the course of developing Java 2 platform security, great attention has been given to interface design, proper division of labor among classes, minimizing the number of classes and APIs, and, whenever possible, implementing private classes and methods for the sake of clarity and protection. Typically, we do not start to prototype code until we have a good grasp of the APIs. This was especially true when we first began developing Java 2. Throughout the development life cycle, we have responded to comments and suggestions and have made extensive revisions to APIs throughout the project, all without much difficulty and without jeopardizing code quality or project delivery. When we made the initial transition from JDK 1.*x* we superseded fragile features, such as those methods we deprecated in the `SecurityManager` class, with a more robust architecture.

We did encounter in JDK 1.0 two artifacts that, although inconvenient, were not changed. First, system classes have been traditionally loaded with a "primordial" class loader, which is now formally referred to as the bootstrap class loader. If you asked for the class loader of a system class, you were given a `null`. This became a sort of de facto API; that is, a class having a `null` class loader was a system class. Some programmers started to test for the existence or nonexistence of class loaders as a way to distinguish between system and nonsystem classes, especially as part of the security decision-making process. For backward compatibility, Java 2 provides that system classes are still loaded by the bootstrap class loader; if a `null` is returned as the class loader for a class, it means that the bootstrap class loader loaded the class. *Note:* System classes are now sometimes referred to as bootstrap classes, but we will continue to refer to them as system classes in this book.

This association between system classes and the `null` class loader, coupled with the difference in treatment of classes based on their class loader types, however, makes it difficult to subdivide system classes into various packages or JAR files and then give them separate sets of permissions. Such a subdivision can reduce the amount of code you need to trust completely, as well as reduce

the amount of trust in that code. In Java 2, application classes residing on the local file system must now be loaded with non-`null` class loaders. Further, a class being loaded with a non-`null` class loader does not say anything about its status, as the class might have been granted `AllPermission`. Hindsight tells us that it would have been much easier to evolve the design if all system classes had been originally loaded with a special, but non-`null`, class loader.

The second JDK 1.0 artifact that is not changed is that the runtime system does not always have a security manager installed, and in this case, a call to `System.getSecurityManager` results in a `null` security manager. Again, for backward compatibility, we have not changed this in Java 2. However, this oddity causes a few unnecessary complexities. For example, each invocation of a security check must be preceded with a test for a `null` security manager; this clutters code. Historically, programmers have tested for a `null` security manager as a means of determining the state of the universe, rather like trying to distinguish the world before and after the so-called big bang. This has led to unwarranted assumptions of how a virtual machine should behave when the security manager is `null`, partly because no security checks can be invoked on a `null` security manager. These assumptions should not have been permitted at a general level; nevertheless, they are being made by some programmers. The presence of such assumptions creates pressure on maintaining backward compatibility.

The `AccessController` class makes it possible to invoke security checks in the absence of a security manager, but such checks may need to be deployed gingerly for fear of breaking backward compatibility. It would have been easier for us if the security manager had always been installed—that is, immediately after the bootstrap process—even though its behavior might change over time.

The lesson learned from these two artifacts is that one cannot easily evolve the interface design of something that is `null`. Further, you definitely cannot invoke method calls on something that is `null`.

Chapter 4. Secure Class Loading

... sometimes it is necessary ... to consider men as a class, yet in the long run our safety lies in recognizing the individual's worth or lack of worth as the chief basis of action, and in shaping our whole conduct ... accordingly.

—Theodore Roosevelt

Dynamic class loading, a fundamental feature of the Java virtual machine, enables the Java platform to install software components at runtime [74]. This feature has a number of interesting characteristics. One is *lazy loading*, which means that classes are loaded on demand and as they are needed to resolve links. *Type-safe linkage* is the mechanism by which the Java virtual machine maintains type safety. It does so by adding link-time checks, which are performed only once, thus avoiding additional runtime checks. In addition, dynamic class loading in the Java platform supports the notion of *user-definable class loading policy*, whereby user-defined class loaders can customize the means by which classes are discovered and the security attributes assigned to classes loaded from particular sources. Finally, dynamic class loading supports the notion of *multiple namespaces*. For example, a browser can load applets from various Web pages into separate class loaders, thus maintaining a degree of isolation among those applet classes. In fact, those applets may contain classes of the same name; they are treated as distinct types by the Java virtual machine.

Section 2.3 touched on language type safety, which is enforced by a variety of techniques, including bytecode verification, class loading, and runtime checks. This chapter focuses on the algorithms and APIs for locating class files, determining the appropriate class loaders to use, assigning suitable security attributes to loaded classes, and associating the classes with protection domains, described in the next chapter.

4.1 Class Files, Types, and Defining Class Loaders

When a class loader loads Java software components, the smallest component unit is a class. A class is defined in a machine-independent binary representation called the *class file format* [74]. The representation of an individual class is called a *class file*, even though it need not be stored in an actual file. Typically, a class is in a file created as the result of compiling the source code for the class. A class file may contain bytecode, as well as symbolic references to fields, methods, and names of other classes. An example of a class is a class `C` declared as follows:

```
class C {
void f() {
D d = new D();
...
  }
}
```

The class file representing `C` contains a symbolic reference to class `D`. The symbolic reference is resolved to the actual class type when class `C` is linked. To do this, the Java virtual machine must ask a class loader to load the class file of `D` and create the class type. The Java virtual machine relies on the class loader that defined class `C` to be the *initiating loader* to find class `D`.

A class loader instance `L` that directly creates—loads and defines—class `C` is called the *defining loader* of the class. The actual *class type* is completely specified by both the class name `N` and the defining class loader. We symbolically denote this relationship as `C = <N, L>`. In other words, two class types in the Java virtual machine are equal if both the class names are equal and the classes have the same defining class loader.

4.2 Well-Known Class Loader Instances

Because each class loader is itself an instance of a class that must be loaded by another class loader, a chicken-and-egg question arises; that is, where does the first class loader come from? The answer is: A *"primordial" class loader* bootstraps the class loading process. This is formally referred to as the *bootstrap class loader* [48] and is sometimes written in a native language, such as C. The bootstrap class loader often loads classes from the local file system in a platform-dependent manner.

Some classes, such as those defined in the `java.*` packages, are essential for the correct functioning of the Java virtual machine and Java runtime environment. These core classes are loaded and defined by the bootstrap class loader. They are referred to as *system classes* in the seminal paper "Dynamic Class Loading in the Java Virtual Machine" by Bracha and Liang [73]. Classes loaded by the bootstrap class loader have a defining class loader that is represented as a `null` reference. The `null` reference is merely an artifact of the implementation and is not part of the platform specification.

To confuse matters, fairly late during Java 2 development, there was a terminology shift. All classes that reside on the `CLASSPATH` are called system classes, as they are loaded by a user-defined class loader known as the *system class loader*. A reference to this class loader can be obtained by invoking the method `java.lang.ClassLoader.getSystemClassLoader`. We will refer to this class loader instance as the *application class loader* and to the classes it defines as *application classes*. We will refer to classes defined by the bootstrap class loader as *system classes*.

Another well-known class loader instance is the extension class loader, which loads classes from the installed optional packages [122]. Optional packages were formerly known as standard extensions: hence the name extension class loader.

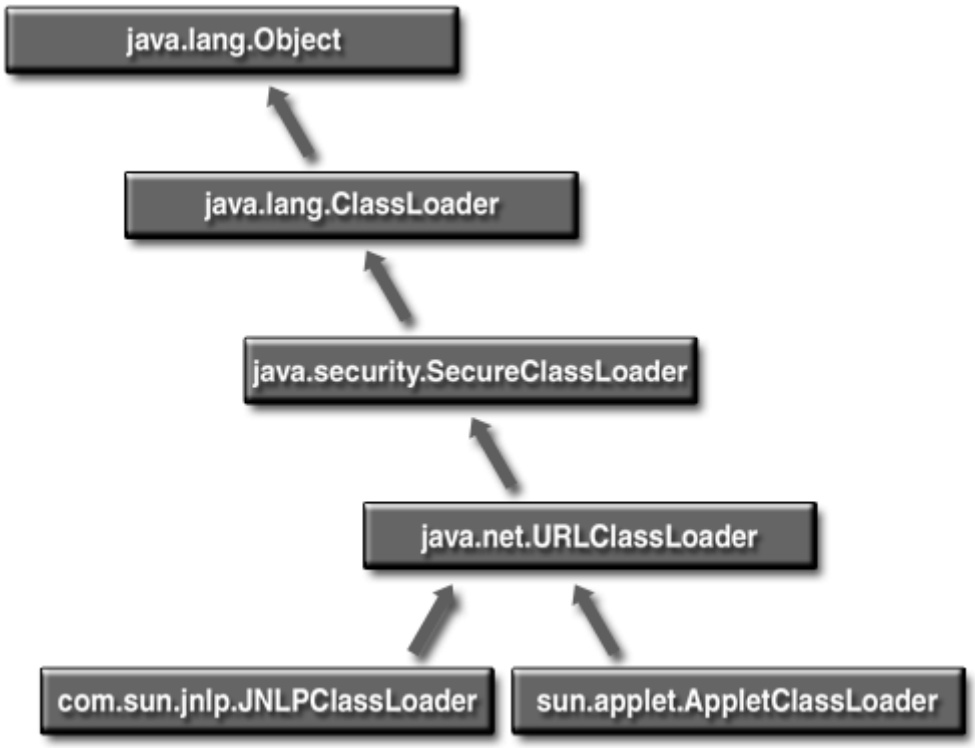
4.3 Class Loader Hierarchies

Java 2 has two orthogonal class loader hierarchies. The class definition hierarchy contains subclasses of `java.lang.ClassLoader`. Each subclass in this hierarchy builds on the behavior and semantics of its ancestors. This *inheritance hierarchy* is covered in Section 4.3.1. The second hierarchy is a dynamic one formed at runtime. This is the class loader *delegation hierarchy*. In the previous code example, class `C` constructs a new instance of class `D`. The class loader delegation hierarchy enables class `C` to reach class `D`, regardless of whether class `D` is a system, extension, application, or other category of class. The class loader delegation hierarchy is described in Section 4.3.2.

4.3.1 Class Loader Inheritance Hierarchy

Figure 4.1 shows part of the class loader inheritance hierarchy. Class loaders are ordinary objects. At the root of the class loader class hierarchy is the abstract class `java.lang.ClassLoader`, originally defined in JDK 1.0 and since expanded (see Section 4.4). The `java.security.SecureClassLoader` class, introduced in Java 2, is a subclass and a concrete implementation of the abstract `ClassLoader` class. The class `java.net.URLClassLoader`, also introduced in Java 2, is a subclass of `SecureClassLoader`. The `URLClassLoader` is a fully functional class loader, whereas its superclasses are either abstract or missing useful implementations of key methods.

Figure 4.1. Subclassing `Classleader`



The utility program `appletviewer` relies on a vendor-specific implementation class, `sun.applet.AppletClassLoader`, to load applets. In JDK 1.0, `AppletClassLoader` is a subclass and concrete implementation of `ClassLoader`; in Java 2, a subclass of `URLClassLoader`. Note that interposing new classes between an existing class and its superclass is binary backward compatible [48].

When creating a custom class loader class, you can subclass from all but one of the class loader classes mentioned in this section. Which one you subclass from depends on the particular needs of your custom class loader. The class you should not subclass from is the `AppletClassLoader` class. Because it is a vendor-specific implementation defined in the `sun.*` package, this class is not supported and is subject to change.

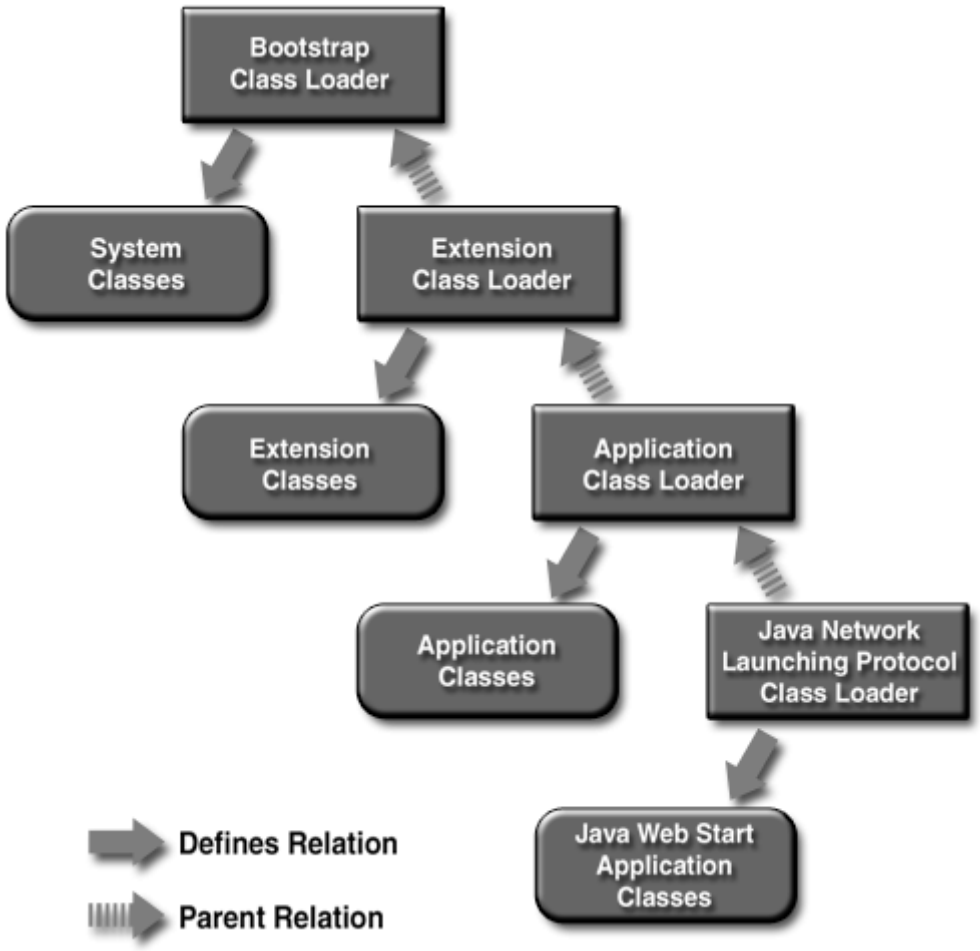
4.3.2 Class Loader Delegation Hierarchy

When it is the initiating loader and is asked to load a class, a class loader either loads the class itself or asks another class loader to do so, using the mechanisms described in Section 4.4. In other words, the first class loader can delegate to the second class loader. The delegation relationship is formed at runtime and has nothing to do with which class loader loads which other class loader. Instead, the delegation relationship is formed when `ClassLoader` objects are created; it takes the form of a parent-child relationship.

When it needs to discover a class, the Java virtual machine starts the search from the class loader that defined the class that triggered the class loading. The Java runtime environment creates a class loader delegation hierarchy with the application class loader typically at the top of the hierarchy. The formation of this hierarchy occurs early in the runtime's start-up sequence, at which point it creates the application class loader. The application class loader delegates to the extension class loader, and, finally, the extension class loader delegates to the bootstrap class loader. That is, the bootstrap class loader is the delegation root of all class loaders. Applet and application containers, such as the `appletviewer`, Java Plug-in, or Java Web Start, instantiate class loaders that delegate to the application class loader.

With all classes in one Java runtime environment, a class loading tree can easily be formed to reflect the class loading relationship (Figure 4.2). Each class that is not a class loader is a leaf node. The immediately preceding node of each class is its defining class loader, with the bootstrap class loader being the root class. Such a structure is a tree because there cannot be cycles; that is, a class loader cannot load its own ancestor class loader.

Figure 4.2. Class loading relationship



The constructors of the `ClassLoader` API build the delegation tree:

```
protected ClassLoader(ClassLoader parent)
protected ClassLoader()
```

The first constructor creates a class loader that has the supplied class loader as the delegation parent. The second constructor uses a default delegation parent. The default delegation parent is the class loader returned by a call to the `getSystemClassLoader` method of the `ClassLoader` class:

```
public static ClassLoader getSystemClassLoader()
```

You can obtain the parent of a class loader by calling the `getParent` method:

```
public ClassLoader getParent()
```

Because class loaders can perform sensitive operations, such as defining classes, restrictions are in place, when a security manager is present, to control the creation of class loaders by invoking a security check in the constructors. (The security manager is described in Chapter 6.) Because all class loaders are subclasses of `ClassLoader` and constructors in the subclasses always call the `super()` constructor, security checks placed here are always invoked. A security check involves ensuring that a particular permission has been granted, by the security policy, to all the code on the execution stack. In the case of the `ClassLoader` constructors, the permission that is required and checked for is the `RuntimePermission` with name `"createClassLoader"`. See Chapter 5 for information on permissions and security policy granting various permissions to code of varying locations, signers, and so on. The full access control—permission checking—algorithm, including specification of exactly which code must have the required permission, is described in Chapter 6.

Restrictions are also implemented to control who can successfully invoke the `getSystemClassLoader` and `getParent` methods. The reason is primarily because from within any object, you can call `this.getClass().getClassLoader()` to obtain its own defining class loader. With a reference to this class loader, you might attempt to "reach over" to its delegation parents and then invoke methods on them. Uncontrolled reach-over is clearly undesirable. Thus, if a security manager is present, the `getSystemClassLoader` and `getParent` methods will succeed only if the caller's class loader is the same as or is a delegation ancestor of the current class loader or if the code in the execution context has the `RuntimePermission` with name `"getClassLoader"`. Otherwise, a security exception will be thrown. Note that allowing a delegation ancestor to have access is reasonable because a delegation child, on its creation, must designate its delegation parent. Obviously, one has to be very careful about which class loader is specified as the delegation parent.

For similar reasons, the same security check is placed in the method call `Class.getClassLoader()` because you do not want anything with a reference to a `Class` object to reach over to its `ClassLoader` object. This security check is new to Java 2. The other security checks mentioned in this section were done in prior releases in the approach typical for those releases, that is, calling `check` methods in the `SecurityManager` class, as described in Chapter 6.

The use of delegation is described in the following sections.

4.4 Loading Classes

The following `ClassLoader` methods relate to class loading:

```
public Class loadClass(String name)
protected synchronized Class loadClass(String name,
                                       boolean resolve)
protected native final Class findLoadedClass(String name)
protected Class findClass(String name)
protected final void resolveClass(Class c)
```

The `loadClass` methods, called by the Java virtual machine to load classes, take a class name as argument and return a `Class` object. The loading process involves finding the binary form of the class—typically by retrieving the binary representation previously created by compiling the source code for the class—and then "defining" the class, that is, constructing from that binary form a `Class` object that represents the class.

4.4.1 Finding a Class

The default implementations of the `loadClass` methods, in the `ClassLoader` class, search for a class in the following order:

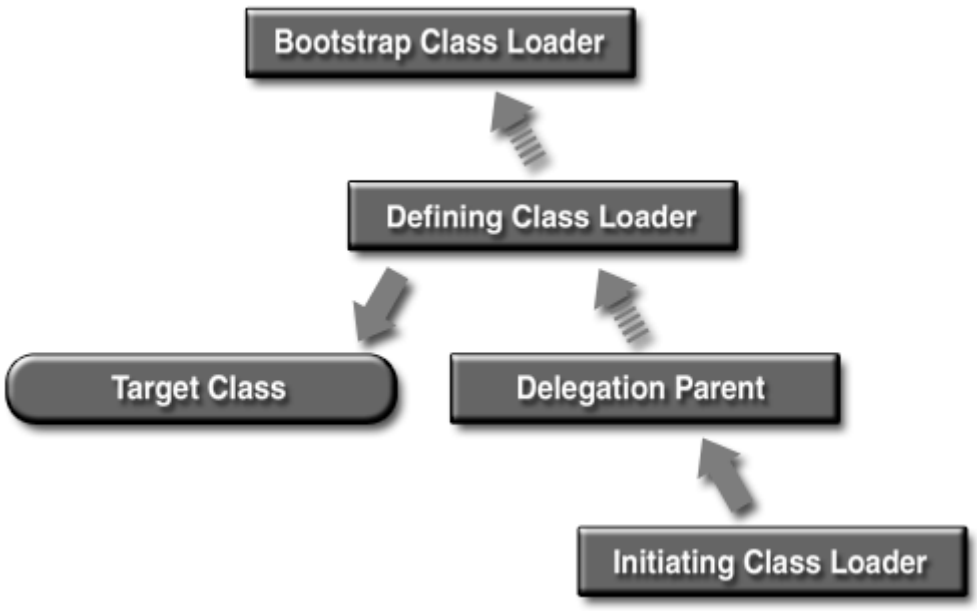
1. Call the `findLoadedClass` method to check whether the class has already been loaded. Otherwise,
2. If this class loader has a specified delegation parent, call the corresponding `loadClass` method of the parent, thereby delegating to the parent the task of loading the class.
3. If none of the class loaders in this class loader's delegation hierarchy loaded the class, invoke the `findClass` method to find the class using this class loader's implementation-specific mechanism to locate the class.

The `findLoadedClass` method looks into the Java virtual machine's loaded class cache to see whether the class has already been loaded. It is critical for type safety that the same class not be loaded more than once by the same class loader. Note that this method is `native final`, and it thus may never be overridden.

If the class is not among those already loaded, the current class loader will attempt to delegate the task to its parent class loader. This can occur recursively, ensuring that the appropriate class loader is used. For example, when searching for a system class, the delegation process continues until the bootstrap class loader is reached.

Note: Each class loader's `findClass` method attempts to locate only the classes the class loader is responsible for defining. Thus, the bootstrap class loader looks only for system classes—for example, classes in the runtime JAR file `rt.jar`—whereas the extensions class loader searches only for classes in the files of the extensions directory, and so on. If it doesn't find the class being searched for, a class loader throws an exception that is caught by the class loader that delegated to it, and then that class loader calls its own `findClass` method to attempt to find the class. The process is depicted in Figure 4.3.

Figure 4.3. `ClassLoader` searching for classes



In this figure, the solid arrows depict the parent-child delegation relationship between class loader instances, and the dashed arrows represent the possibly multiple class loader instances traversed to find the desired class. The *initiating class loader*, that is, the class loader initially asked to load the target class, delegates to its delegation parent, which delegates to its delegation parent, and so on, all the way up to the final delegation parent, the bootstrap class loader. The bootstrap class loader attempts to find the class. If it fails, the class loader that delegated to it attempts to find the class, and so on, until one of the class loaders finds the class, or it is not found at all. If the class is found, the class loader that found it must be the *defining class loader*, as described in Section 4.1.

The `findClass` method provides a way to customize the mechanism for looking for classes. Thus, a custom class loader can override this method to specify how a class should be discovered. For example, an applet class loader can override this method to go back to the applet host—origin server—to try to locate the class file and load it over the network.

If the class was found by using the previous steps and the `resolve` flag is `true`, the `loadClass` method then calls the `final resolveClass` method on the resulting `Class` object to link the class if it hasn't already been linked.

An issue about class loading to be clarified is which class loader does the Java virtual machine start with when trying to load a class, given the name of any class? Following are the rules implemented in Java 2:

- When the first class of an application is being loaded, a new instance of the application class loader, that is, the class loader returned by the `ClassLoader getSystemClassLoader` method, is used. In the Sun Microsystems implementation, this class loader is a subclass of `URLClassLoader`.
- When the first class of an applet is being loaded, a new instance of the `AppletClassLoader` is used.
- If the request to load a class is triggered by a reference to it from an existing class, the class loader for the existing class is asked to load the class.

The rules about the use of `URLClassLoader` and `AppletClassLoader` instances have exceptions and can vary, depending on the particular system environment. For example, a Web browser may choose to reuse an existing `AppletClassLoader` to load applet classes from the same Web page.

4.4.2 Defining the Class

Part of the operation of loading a class involves defining the class after the binary representation of the class is found by the steps listed in Section 4.4.1. The `defineClass` methods convert an array of bytes—containing the binary representation—into an instance of class `Class` and associate the class with its appropriate `ProtectionDomain`:

```
protected final Class defineClass(String name, byte[] b,
      int off, int len, ProtectionDomain protectionDomain)
protected final Class defineClass(String name, byte[] b,
      int off, int len)
```

As you will see in the next chapter, a `ProtectionDomain` is an object used for grouping together various characteristics applicable to all classes in the domain. During class definition, each class is assigned to its `ProtectionDomain`, based on characteristics of the `CodeSource` of the class. A `CodeSource` is described by two properties: the origin of the class, specified as a URL, and the signers of the code, if any, which are specified by the digital certificates corresponding to the private keys used to sign the class. (Code signing is described in

Section 8.3.) It is incumbent on the defining class loader to associate each class with the appropriate `ProtectionDomain`. At runtime, the Java virtual machine keeps track of the `ProtectionDomain`s of the class instances on the call stack. A resource access is basically granted if, and only if, all `ProtectionDomain`s of the class instances on the call stack are granted the requisite permission. (See Chapter 6 for full details.)

When defining a class, a class loader—instance of `ClassLoader`—must assign the class being defined to a `ProtectionDomain` based on the `CodeSource`. A class loader may implement an optimization that reduces the number of `ProtectionDomain`s constructed by caching unique `ProtectionDomain`s as they are created and reusing them when the same `CodeSource` is encountered for subsequent classes.

A class loader that directly finds or creates the `ProtectionDomain` then calls the `ClassLoader defineClass` method with the `ProtectionDomain` instance as a parameter. Some class loaders, such as `URLClassLoader`, don't search for or construct a `ProtectionDomain` but instead rely on the inherited `SecureClassLoader defineClass` method, which takes a `CodeSource` parameter. The `SecureClassLoader defineClass` method constructs or finds a `ProtectionDomain` instance and then passes it in a call to the `ClassLoader defineClass` method, as described in Section 4.5.

Note that the second `defineClass` method does not explicitly mention a `ProtectionDomain`, because this method existed before Java 2, when `ProtectionDomain`s did not exist. In this case, a default `ProtectionDomain` is used. This associates the class with the domain having a `CodeSource` of (`null, null`), that is, a `CodeSource` with `null` for both the URL and certificates. Such a class will be effectively granted the permissions that the security policy specifies for code whose origin and signing status—whether it was signed and by whom—do not matter.

The result of class definition is that a `Class` is created and marked as belonging to a specific protection domain. You can later query a class on its protection domain by calling the `Class.getProtectionDomain` method. Obviously, `ProtectionDomain` objects are security-sensitive objects, so you must be cautious about who can obtain references to them. If a security manager is present, the `getProtectionDomain` method invokes a security check to ensure that the code in the current execution context has the `RuntimePermission` with name `"getProtectionDomain"`. If it does not, a security exception is thrown.

When a class file is correctly signed with one or more digital signatures, the `defineClass` method that includes a `ProtectionDomain` argument calls the `ClassLoader.setSigners` method to associate the signers' certificates with the runtime `Class` created from the class file:

```
protected final void setSigners(Class c, Object[] signers)
```

You can query a class for its signers by calling the `Class.getSigners` method. There is no security check placed in this method, because it is usually not a security risk to reveal who signed the class.

4.4.3 Other `ClassLoader` Methods

The rest of the methods in the `ClassLoader` class are related mostly to finding resources and packaging. They are mentioned next with no further description:

```
protected String findLibrary(String libname)
public URL getResource(String name)
public final Enumeration getResources(String name)
public Enumeration findResources(String name)
public URL findResource(String name)
public static URL getSystemResource(String name)
public static Enumeration getSystemResources(String name)
public InputStream getResourceAsStream(String name)
public static InputStream
    getSystemResourceAsStream(String name)
protected Package definePackage(...)
protected Package getPackage(String name)
protected Package[] getPackages()
```

Refer to the javadocs for further explanation.

4.5 `SecureClassLoader` Details

The `java.security.SecureClassLoader` class extends `ClassLoader` with additional support for defining a class, given the code source of the class. During Java 2 development, this class initially had a richer design with a comprehensive set of method calls. Gradually, those functionalities have been moved either to the base class `ClassLoader` or to the newly created class `URLClassLoader`.

The two main methods of the `SecureClassLoader` class are the following:

```
protected final Class defineClass(String name, byte[] b,
                      int off, int len, CodeSource cs)
protected PermissionCollection getPermissions(CodeSource
                                    codesource)
```

The `defineClass` method defines a class from a particular code source. In some sense, this method duplicates certain functionality of the `ClassLoader` method `defineClass`, which takes a `ProtectionDomain` as an argument. However, sometimes it is convenient not to have to worry about protection domains. For example, the caller, a subclass, of this `SecureClassLoader` method, might not be able to determine which protection domain to use but still needs to define the class. In this case, `codesource` is the only piece of information available about the origin of the class that can be used to determine the permissions to be granted.

The `SecureClassLoader defineClass` method determines the `ProtectionDomain` for the class, first checking whether a `ProtectionDomain` has already been created for code from the code source of the class. If so, a reference to that `ProtectionDomain` is used. If such a `ProtectionDomain` has not yet been created, `SecureClassLoader` calls `getPermissions` to get the static permissions assigned to the specified code and then instantiates a `ProtectionDomain`, passing its constructor the `CodeSource`; the static permissions, permissions the loader assigns at class loading time; the class loader, `"this"`; and `null` for the `Principal`s array that may be set during execution to indicate who is executing the code. (A class loader should always assign `null` for the `Principal`s, as described in Section 5.3.) After the `ProtectionDomain` has been found or created, the `SecureClassLoader defineClass` method then calls the `ClassLoader defineClass` method that takes a `ProtectionDomain` as an argument.

The `getPermissions` method is intended to return the static permissions for the given `CodeSource` object. The default implementation of this method, as of J2SE 1.4, simply returns an empty new `Permissions` object. A class loader can override this method. For example, this method in the `AppletClassLoader` automatically grants a permission that allows the applet to connect back to the local host if the applet was loaded from the local file system, even though the security policy does not specify this permission. The `URLClassLoader` also customizes the `getPermissions` method, as described in the next section.

4.6 `URLClassLoader` Details

The `java.net.URLClassLoader` class extends `SecureClassLoader` and is used to load classes and resources from a search path of URLs referring to both JAR files and directories. Here are the two constructors:

```
public URLClassLoader(URL[] urls, ClassLoader parent)
public URLClassLoader(URL[] urls)
```

The first method constructs a new `URLClassLoader` for the given URLs and assigns it `parent` as its delegation parent. The second method constructs a new `URLClassLoader` for the specified URLs, using the default delegation parent class loader, which is the application class loader. The URLs will be searched in the order specified for classes and resources but only after `URLClassLoader` first delegates to its parent by searching in the specified parent class loader, as described in Section 4.4.1.

The `URLClassLoader` class overrides the `ClassLoader` method `findClass(String name)` and a few resource-related loading methods to find and load the class or resource with the specified name from the URL search path. Any URLs that refer to JAR files are loaded and opened as needed until the class is found.

More interesting from a security perspective, this class overrides the following method inherited from `SecureClassLoader`:

```
protected PermissionCollection getPermissions(CodeSource
                                              codesource)
```

This method, for returning the static permissions for the given `CodeSource` object, first calls `super.getPermissions()`. The method then adds permissions, based on the URL of the code source, according to the following rules:

- If the protocol specified by the URL is "file" and the path specifies a file, read permission to that file is granted.
- If the protocol specified by the URL is "file" and the path is a directory, read permission is granted to all files and, recursively, all files and subdirectories contained in that directory.
- If the protocol specified by the URL is not "file," a permission to connect to and accept connections from the URL's network host is granted.

In other words, classes loaded by a `URLClassLoader` are, by default, granted permission to access the URLs specified when the `URLClassLoader` was created.

Another distinguishing feature of `URLClassLoader` is the pair of static methods to create new `URLClassLoader` instances:

```
public static URLClassLoader newInstance(URL[] urls,
                                         ClassLoader parent)
public static URLClassLoader newInstance(URL[] urls)
```

As stated earlier in the chapter, security concerns compel severe restrictions on who can create `ClassLoader` instances. However, it is convenient to provide a mechanism for applications or applets to specify URL locations and to load classes or resources from them. These static `newInstance` methods allow any program to create instances of the `URLClassLoader` class, although not other types of class loaders. This is considered acceptable, given the available public methods and the delegation mechanism. Note that an application or applet still cannot call the protected methods in `URLClassLoader` or its superclasses.

Typically in a Web browser and specifically in `appletviewer` and the Java Plug-in, an applet class loader is used to load classes and resources needed for applets. In Java 2, this class is defined in the vendor-specific `sun.*` package and is a subclass of `URLClassLoader`. Additional `URLClassLoader` subclasses mentioned in Section 4.2 are the application class loader and the extension class loader.

4.7 Class Paths

The class loader classes described previously provide programmable ways to locate and load classes and resources. To simplify the task of installing software components on a system supporting the Java platform, common and user-specific places are available in which to put such components in order to allow them to be automatically discovered by the Java runtime system.

In JDK 1.0 and 1.1, a well-known, built-in, systemwide search path called `CLASSPATH` is set in a platform-specific way. On UNIX systems, for example, `CLASSPATH` can be set via the shell environment variable `CLASSPATH`. Essentially, all classes or JAR files containing classes on the local file system must reside on this path to be discovered. It also is where all system classes reside. As a result, all classes from the local file system are treated as system classes and are given full privileges to access all resources. In other words, those local classes that really belong to the system code are not distinguished from other local classes that are merely part of some locally installed applications.

This was clearly not perfect. One can imagine many scenarios in which a locally installed application should not be given full system privilege: for example, a demo program newly received in the mail. As another example, when displaying an important document, you might want to run the display application in read-only mode to ensure that the content of the document is not altered or lost due to software bugs in the application.

The security architecture in Java 2 includes provisions to treat locally resident classes in the same way as remotely downloaded applet classes, that is, by granting them specific and fine-grained permissions. For this to work, true system classes must be distinguishable from all other classes. The Java 2 approach is to have separate class paths: one for system classes and one for the rest.

The earliest design for this path separation, which was released in a beta version of Java 2 SDK, called for a search path—the application class path—in addition to the existing `CLASSPATH`. As with JDK 1.1, all classes on `CLASSPATH` were treated as system classes. All classes on the application class path were non-system classes, however, and were loaded with instances of the `SecureClassLoader`, which granted them permissions according to the security policy. The application class path could be specified by either setting a property called `java.app.class.path` or using a command line option when invoking the application. Command line options and other deployment issues are discussed in Chapter 12.

This design had the advantage of being fully backward compatible. An existing application could be migrated from sitting on `CLASSPATH` to the new application class path at its own pace and without affecting other installed software components. Before migration, the application ran exactly as in JDK 1.0 and JDK 1.1. Once migrated, the application became subject to fine-grained access control. However, it can be argued that such migration effort should not be placed on the shoulders of users. Also, the backward compatibility might have simply led users to do nothing at all; thus, they would miss out on a much better security architecture and a very powerful extensions mechanism.

Because of such concerns, in the eventual design of Java 2, `CLASSPATH` is interpreted as the application class path. Thus, deployed applications do not have to be moved. When Java 2 is installed, classes on this path are loaded by subclasses of `URLClassLoader`. The security policy can be configured to grant different permissions to different code sources on the application path.

As for the system class path, J2SE recognizes a nonstandard command line flag, `-Xbootclasspath`, which can be used to augment the locations the bootstrap class loader searches to discover classes. Users or developers should rarely or never have to install classes on this path. Note that this design might not provide full backward compatibility for some existing applications, even though the number of such applications is expected to be very small. The reason is that up to and including JDK 1.1, all classes on `CLASSPATH` were treated as system classes and were loaded with the `bootstrap` class loader. To reiterate, in J2SE, classes are loaded with an instance of `URLClassLoader`. Therefore, an application that checks for `bootstrap` class loaders might need to be upgraded to reflect the presence of `URLClassLoader`.

You might question why there remains a separate system class path. If system classes need all permissions, and they do, why not simply use the policy to grant them `AllPermission` and thus treat them as just a special kind of application? The real situation is somewhat more complicated than this. As noted, system classes are accustomed to being loaded by the `bootstrap` class loader. Determining whether a class is a system class by whether it has a `bootstrap` class loader is not good practice, yet there remains legacy code that is best not broken in the new security architecture. Moreover, bootstrapping and other issues could be technically solved, but the solutions were judged to be too destabilizing to attempt for Java 2. We hope that, in the future, different parts of the system classes can be granted only those fine-grained permissions that they really need. This subdivision of system classes will constrain the power of each system component and further reduce the consequences of programming errors in system classes.

Chapter 5. Elements of Security Policy

Policy must be clear, consistent, and confident.

—Dwight D. Eisenhower

A security policy, specified separately from the Java runtime system by a system administrator or user, indicates what security-sensitive system resources may be accessed by various groups of code. A security policy is essentially a mapping from a set of properties characterizing running code to a set of resource access permissions granted to the code. Some code will be granted a certain set of permissions, whereas other code will be granted its own set of permissions. In a Java runtime environment, the policy contents are represented in an implementation of the abstract `Policy` class.

This chapter documents the elements that comprise the interfaces to security policy. We describe classes relevant for representing the policy, running code, and the security-sensitive resource accesses the code requires in order to do specific operations.

The first section describes `Permission`s, instances of which represent access to specific security-sensitive resources. A `Policy` implementation usage of permissions is twofold. First, permissions are used to represent which resource accesses are granted to different groups of code. Second, when a security-sensitive operation is attempted, a relevant permission is constructed to indicate which permission must be granted by the security policy to the running code in order for the operation to be allowed. The `Policy` is then consulted to determine whether the specified permission has been granted.

Section 5.2.1 describes the `CodeSource` class, which holds two of the three properties used to characterize running code: its URL (origin) and digital certificates specifying the signers, if any, of the code. The third property, giving the Java runtime an indication of who is running the code, is discussed in Section 5.2.3 and further in Chapter 8.

In Section 5.3, we describe `ProtectionDomain`s. As mentioned in the preceding chapter, each class is associated—when it is defined—with an instance of `ProtectionDomain`. A `ProtectionDomain` is a convenience class for grouping the `CodeSource` of the class; static permissions, or ones assigned at class loading time; a reference to the defining class loader for the class; and a `Principal` array that may be set during execution to indicate on whose behalf the code is executing. All classes with the equivalent `CodeSource`, `Principal`s, and defining class loader belong to the same `ProtectionDomain`. The JVM tracks all the `ProtectionDomain`s for the code on the execution stack. As you will see in Chapter 6, a resource access is granted if every `ProtectionDomain` in the current execution context has been granted the permission required for that access, that is, if the code and `Principal`s specified by each `ProtectionDomain` are granted the permission.

Section 5.4 describes security policy and the abstract `Policy` class. At runtime, a concrete implementation of this class encapsulates the externally declared security policy, which is consulted whenever security checking is done. Section 5.5 depicts how permissions are granted to running code.

5.1 Permissions

The permission classes represent system resources and the operations supported on those resources. Each permission has a `String` name, and some permissions also have a `String` action list. An example of a permission is a `FilePermission` with name `"/tmp/abc"`

and action `"read"`, which specifies read access to the file `/tmp/abc`. Permissions are used to specify both what resource accesses are allowed by a policy and what permissions are required at runtime prior to performing security-sensitive operations.

Permission objects are used in a security policy to indicate which security-sensitive resource accesses are allowed for various groups of code, that is, for code in various locations, with various signers, executed by various users or services. Some code may be granted certain permissions, whereas other code is granted a different set of permissions.

A permission is also constructed at runtime as needed to indicate a permission that must be granted, by the security policy, to all the code in the current execution context in order for a particular resource access to be permitted. That is, a method responsible for performing a security-sensitive operation should always first construct an appropriate `Permission` and then test whether the callers on the call stack have been granted the specified permission by the policy. The method does this in the manner described in Chapter 6, typically by calling the `SecurityManager checkPermission` method, passing it the required `Permission`.

All permission classes are positive in that they represent approvals, rather than denials, of access. This design choice, discussed further in Section 5.1.5, greatly simplifies the implementation and improves efficiency.

The remainder of this section describes permissions. We first cover the design of the `Permission` class hierarchy and some of the specific permission classes. Then we discuss sets of permissions and provide details about what it means for one permission to "imply" another.

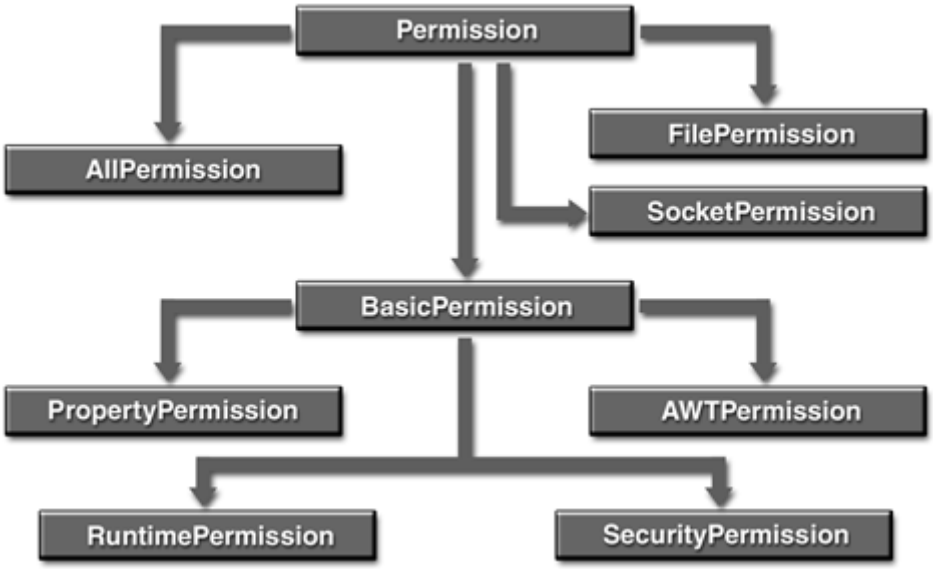
5.1.1 Permission Class Hierarchy

The root class of the `Permission` class hierarchy, `java.security.Permission`, is an abstract class and may be subclassed, as appropriate, to represent access to a specific resource. For example, `FilePermission` is a `Permission` subclass that is used to represent access to a file. The following code can be used to construct a permission that represents read access to the file named `abc` in the directory `/tmp`:

```
perm = new java.io.FilePermission("/tmp/abc", "read");
```

New permissions are subclassed from either the `Permission` class or one of its subclasses, such as `java.security.BasicPermission` (Figure 5.1). Subclassed permissions, other than `BasicPermission`, generally belong to their own packages. Thus, for example, `FilePermission` is found in the `java.io` package, which holds the APIs for file system access.

Figure 5.1. Common `Permission` subclasses



5.1.2 Common Permission Classes

Some of the more commonly encountered permission classes are `Permission`, `BasicPermission`, `AllPermission`, and `UnresolvedPermission`.

`java.security.Permission`

At the root of the `Permission` class hierarchy is the abstract `Permission` class, which has the following constructor and public methods:

```
public Permission(String name);
public abstract boolean implies(Permission permission);
public abstract boolean equals(Object obj);
public String toString();
public PermissionCollection newPermissionCollection();
```

Each permission has a target name, whose interpretation depends on the subclass. It is conceivable that for certain types of permissions, the target name is of no importance and is thus not interpreted.

Although the abstract `Permission` class has just a name argument, most `Permission` objects also include a list that gives the actions that are permitted on the permission target. For example, the permission name for a `java.io.FilePermission` object is the path to a file or directory, and the list of actions specifies which operations, such as `read` and `write`, are granted for the specified file or for files in the specified directory.

The actions list is optional for those `Permission` objects that do not need such a list. One example is `java.lang.RuntimePermission`, where the named permission, such as `"exitVM"`, is either granted or not. There is no further subdivision of different actions. Admittedly, for these special cases, quite often the name embodies both the target of the permission—for example, `VM` is the target from which to exit—and the action: `exit`. For simplicity, the target and action are merged as one string. Such permission classes typically subclass from `java.security.BasicPermission`

Each `Permission` instance is typically generated by passing one or more string parameters to the constructor. The first parameter is usually the name of the target, such as the name of a file for which the permission is aimed. The second parameter, if present, is the action, such as `"read"` for reading the file. Generally, a set of actions is specified as a comma-separated composite string.

`Permission` objects, like `String` objects, are immutable once they have been created. Subclasses should not provide methods that can change the state of a permission once it has been created.

Whether two `Permission` objects are considered `equal` is left entirely up to each subclass of the `Permission` class. In fact, it is up to each `Permission` subclass to determine the appropriate semantics for its implementation of each abstract `Permission` method.

One method that must be implemented by each subclass is the `implies` method, which is used to compare permissions. The semantics of the statement "permission `p1` implies permission `p2`" means that if you are granted permission `p1`, you are naturally granted permission `p2`. Thus, this is not really an equality test but rather more of a subset test.

It is important to remember that object equality differs from permission equivalence. Object equality is useful, for example, when you store objects in hash tables and later need to determine whether an entry already exists. This can be done by calling the `equals` method. Permission equivalence, on the other hand, means that two objects semantically represent the same permission. To determine permission equivalence, you must use the `implies` method and check to see whether one `Permission` object implies another, and vice versa.

Sometimes, it is desirable to present a permission's content in a human-readable fashion. The `toString` method returns a string describing the permission. The convention is to specify the class name, the permission name, and the actions in the following format:

```
(ClassName name actions)
```

For example, the following is returned as a result of a call to the `toString` method on a `java.io.FilePermission` that specifies both read and write access to the file */tmp/abc*:

```
(java.io.FilePermission /tmp/abc read,write)
```

As shown in Section 5.1.3, subclasses of `Permission` sometimes need to be stored in a specialized `PermissionCollection` object so as to provide the desired `PermissionCollection.implies` semantics. When this is the case, the `Permission.newPermissionCollection` method must be overridden such that the specialized `PermissionCollection` is returned. The default `newPermissionCollection` behavior is to return `null`.

The `Permission` class implements two interfaces: `java.security.Guard` and `java.io.Serializable`. For the latter, the intention is that `Permission` objects may be transported to remote machines, such as via Remote Method Invocation (RMI), and thus a `Serializable` representation is useful. `Guard`, which is related to the class `java.security.GuardedObject`, is discussed in Chapter 9.

Applications are free to introduce new categories of `Permission` classes beyond those that the system always supports. How to add application-specific permissions is discussed in Chapter 7.

java.security.BasicPermission

The `java.security.BasicPermission` class extends the `Permission` class and offers a very simple naming convention that is often encountered when creating permission classes. `BasicPermission` is commonly used as the base class for "named" permissions. A named permission contains a name, such as `"exitVM"`, `"setFactory"`, and `"queuePrintJob"`, but no actions list; either you have the named permission or you do not. The naming convention follows the hierarchical property naming convention, which is analogous to the package naming convention. An asterisk may appear at the end of the name, following a `"."` or by itself, to signify a wildcard match. For example, `java.*` and `*` are valid, but `*java` and `a*b` are invalid.

`BasicPermission` is an abstract class, so you cannot directly construct it and must construct one of its subclasses instead. Following are the subclasses of `BasicPermission` defined by the J2SE platform:

```
java.awt.AWTPermission
java.io.SerializablePermission
java.lang.RuntimePermission
java.lang.reflect.ReflectPermission
java.net.NetPermission
java.security.SecurityPermission
java.sound.sampled.AudioPermission
java.sql.SQLPermission
java.util.PropertyPermission
java.util.logging.LoggingPermission
javax.net.ssl.SSLPermission
javax.security.auth.AuthPermission
javax.security.auth.kerberos.DelegationPermission
```

Subclasses may implement actions on top of `BasicPermission`, if desired. In the preceding list of subclasses, only `java.util.PropertyPermission` currently does this. For such a permission, the name is the property name, such as `"java.home"`, and the actions list specifies whether you can set the specified property value—if the `actions` string is `"write"`—get the property value—if `actions` is `"read"`—or both—if `actions` includes both `"read"` and `"write"`, as in `"read, write"` or `"write, read"`.

Historically, all permission classes in the `Permission` class hierarchy below `Permission` have two constructors: one that takes just a `String` argument for the name, that is, the target, and another that additionally takes a `String` argument for the actions, regardless of whether the permission class uses actions. This was a limitation previously imposed by the implementation of the `UnresolvedPermission`, described later, and the default `Policy` provider. The limitation was removed with the release of J2SE 1.4. To remain backward compatible with existing permission classes, the two-argument constructor of a permission class may simply ignore the argument.

`BasicPermission` exposes the following two constructors:

```
public BasicPermission(String name);
public BasicPermission(String name, String actions);
```

The `BasicPermission` implementation of the `implies` method checks whether the permission parameter is of the same class type as this instance and, if so, whether its name is implied by the name of the comparing permission. Here, name string comparison takes into account wildcards, so that, for example, `"a.b.*"` implies `"a.b.c"`.

The BasicPermission implementation of the equals method simply checks whether the permission name strings are equal.

java.security.AllPermission

The java.security.AllPermission class represents a wildcard for all possible permissions. This class was introduced to simplify the work of system administrators who might need to perform multiple tasks that require all or numerous permissions. It would be inconvenient to require the security policy to iterate through all permissions while making a policy decision.

Because AllPermission does not care about the actual targets and actions, its constructors ignore all passed-in parameters. By definition, AllPermission permission implies all permissions. Moreover, two AllPermission objects are always considered equal. Thus, the AllPermission class implements the following two methods specially:

```
public boolean implies(Permission p) {
   return true;
}

public boolean equals(Object obj) {
   return (obj instanceof AllPermission);
}
```

Note that AllPermission also implies any permissions defined in the future. Clearly, granting this permission must be done with caution.

java.security.UnresolvedPermission

As discussed in Section 5.4, the security policy for a Java runtime environment—specifying which permissions are available for code from various code sources—is represented by a Policy object. In particular, the internal state of a security policy is normally represented by the Permission objects associated with each unique pair of CodeSource and Principals, whereby the Principals represent the entity running the code. Whenever a Policy object is initialized or refreshed, Permission objects of specific class types may need to be created for the permissions granted by the security policy.

Many Permission class types referenced by the policy configuration exist locally. That is, those classes can be discovered by the Policy provider's defining class loader or another loader it delegates to, such as the bootstrap class loader. Objects for such permissions can be instantiated during Policy initialization. For example, it is always possible to instantiate a java.io.FilePermission, as the FilePermission class is found on the bootstrap class path.

However, the dynamic nature of Java technology makes it possible that when the Policy object is constructed, the code that implements a particular Permission class has yet to be loaded or is not available for loading. For example, a referenced Permission class might be in a JAR file that will eventually be downloaded. In this case, the Permission class has yet to be defined within the Java runtime environment. For each such class, an UnresolvedPermission object is instantiated instead, as a placeholder that contains information about the permission. Thus, the UnresolvedPermission class is used to hold such "unresolved" permissions. Similarly, the class UnresolvedPermissionCollection stores a collection of UnresolvedPermission permissions.

Unresolved permissions of a particular type must be resolved before an access control decision can be made about a permission of the actual type. It is presumed that the permission class becomes available when the policy enforcement point constructs the requisite permission when determining whether to grant access to the resource. To resolve an `UnresolvedPermission`, the policy decision point must locate and instantiate the appropriate permission class type, based on the lexical information in the `UnresolvedPermission`. This new object then replaces the `UnresolvedPermission`. In the unlikely event that the permission is still unresolved, the policy implementation must ensure that the permission is not granted and deny the requested access.

Note that it is not necessary to instantiate all loadable permission classes at `Policy` initiation. Typically, only a small portion of the `Policy`'s contents is needed. Thus, it is quite legitimate, and even sometimes desirable for performance and efficiency, to make extensive use of `UnresolvedPermission` even when the `Permission` class is loadable, thus delaying the instantiation of the `Permission` objects until they are used. Nevertheless, `Policy` implementers must pay close attention to the complexity that `UnresolvedPermission` adds, especially when an `UnresolvedPermission` may resolve into different implementations at different times.

A few methods, explained later, in the `UnresolvedPermission` class are

```
public UnresolvedPermission(String type,
       String name, String actions,
       java.security.cert.Certificate certs[]);
public boolean implies(Permission p);
public boolean equals(Object obj);
```

Note that the constructor takes an array of certificates, which can be used to verify the signatures on the permission class. Remember that `UnresolvedPermission`s enable the deferred loading of permission classes so that a given permission class need not be defined until necessary. The very nature of such permission classes suggests that a more rigorous mechanism is required to ensure their authenticity. By signing a given permission class and specifying the signing requirement in the security policy, we have a foundation that can be used to assure us that the permission class respects the intentions of the root class `java.security.Permission` and that the implementation is not malicious. Of course, this assurance depends on the trust conveyed by the signature keys used to sign the class. However, without this mechanism, it would be up to the application to make this trust decision, which would be difficult, if not impossible, for the application to do.

The certificates also are useful when a `Permission` class does not reside locally and is downloaded each time it is used. On the one hand, ensuring that the same class file is downloaded each time is often desirable. However, this could be difficult to verify unless local storage is used to keep a copy or at least a fingerprint of a prior class file. On the other hand, software tends to get upgraded often, so it is not uncommon to expect the same named `Permission` class file to change over time, albeit in a consistent way. But again, this consistency is difficult to check by examining the class files. The certificates, which can be used to verify a class file's digital signature, normally change less often and can be managed more efficiently than the class files.

The by now familiar `implies` method always returns `false` for unresolved permissions. The reason is that an `UnresolvedPermission` is never considered to imply another permission.

Finally, when comparing two `UnresolvedPermission` objects for equality, the second `Permission` object must also be an `UnresolvedPermission`. It must also have the same class name, target, actions, and certificates as the first object, the one doing the comparison.

5.1.3 Permission Sets

It is often more convenient to work with sets of permissions than with one permission at a time. The abstract class `java.security.PermissionCollection` represents a collection—a set that allows duplicates—of a specific type of `Permission` objects, such as `FilePermission`. In other words, each `PermissionCollection` holds a homogeneous collection of permissions. Following are the more germane `PermissionCollection` methods:

```
public abstract void add(Permission permission);
public abstract boolean implies(Permission permission);
public abstract Enumeration elements();
public void setReadOnly();
public boolean isReadOnly();
```

The `add` method adds a `Permission` object to the current collection of `Permission` objects. How this is accomplished is left to the subclass implementation. For example, `FilePermission`s can be added to a `PermissionCollection` object in any order.

Similar to its purpose in the `Permission` class, the `implies` method here checks whether the specified permission is implied by one or more of the permissions in the `PermissionCollection` object. If so, we say that the permission is implied by the `PermissionCollection` object. Note that in this case, the specified permission, say to read and write file *x*, might not be implied by any single permission but rather by a composition of permissions in the `PermissionCollection` object, such as one permission to read file *x* and another to write file *x*. Thus, it is crucial that any concrete subclass of `PermissionCollection` ensures that the correct semantics are followed when the `implies` method is called.

The `setReadOnly` method marks the `PermissionCollection` object read-only, effectively making the collection immutable. Once a collection is marked read-only, any attempt to add a permission to the collection will result in a `SecurityException` being thrown.

To group together permission objects of the same type, the caller should first invoke the `Permission` object's `newPermissionCollection` method. The default behavior is simply to return `null`. However, to test whether a permission collection implies a given permission, the collection may at times implement specialized processing. To accomplish this, the implementation of the `PermissionCollection.implies` method ensures the semantically correct behavior and generally does not necessarily rely on the implementation of the permission class's `implies` method. To enable this, the implementation of the `Permission` class would override the `newPermissionCollection` method. When a non-`null` reference is returned from `newPermissionCollection`, the returned `PermissionCollection` must be used. If `null` is returned, the caller of `newPermissionCollection` is free to store permissions of the given type in any `PermissionCollection` it chooses, such as one that is backed by a `Hashtable`, a `Vector`, or whatever collection class is appropriate.

The `java.security.Permissions` class represents a heterogeneous collection of permissions. A `final` subclass of `PermissionCollection`, it is essentially a collection

of `PermissionCollection` objects. That is, it contains various types of `Permission` objects organized into `PermissionCollection`s. For example, any `java.io.FilePermission` objects added to an instance of a `Permissions` class are stored in a package private instance of the `FilePermissionCollection` class. This is the type of collection returned whenever the `FilePermission` `newPermissionCollection` method is invoked.

Following are the main public methods of the `Permissions` class:

```
public void add(Permission permission);
public boolean implies(Permission permission);
```

When the `add` method is called to add a `Permission`, the `Permission` is stored in the appropriate `PermissionCollection`. If no such collection yet exists, the `Permission` object's `newPermissionCollection` method is called to create the `PermissionCollection`, and then the `Permission` object is added to it. Finally, the `PermissionCollection` object is added to the `Permissions object`. If `newPermissionCollection` returns `null`, a default implementation-specific `PermissionCollection` that happens to use a hash table will be created and used. Each hash table entry has the `Permission` class's name as the key and the `Permission` object as the value.

Similar to the behavior of `PermissionCollection`, the `implies` method checks whether a `PermissionCollection` object is present for the supplied permission type and if so, whether it implies the permission passed in as a parameter. It returns `true` if the composition of permissions in the appropriate `PermissionCollection` imply the specified permission.

5.1.4 Implications of Permission Implications

Recall that permissions are often compared with one another. To facilitate such comparisons, each permission class must define an `implies` method that semantically represents how the particular permission class relates to other instances of the permission class. Obviously, `java.io.FilePermission("/tmp/*", "read")` implies `java.io.FilePermission("/tmp/abc", "read")`, but it does not imply any `java.net.NetPermission`. However, there is another, deeper implication that might not be immediately obvious to some readers.

Suppose that an applet is granted permission to write to the entire file system. Presumably, this allows the applet to replace the system software, including the Java virtual machine and runtime environment. This effectively means that the applet has been granted all permissions. Or suppose that an applet is granted runtime permission to create class loaders. It effectively is thereby granted many more permissions, as a class loader can perform sensitive operations, including assigning `AllPermission` to classes it defines.

Other permissions that are potentially dangerous to give out include:

- `AllPermission`—of course
- Those that allow the setting of system properties
- `java.lang.reflect.ReflectPermission`
- `java.security.SecurityPermission` with a target name of `"setPolicy"`

- Runtime permissions for defining packages and for loading native code libraries, because the Java security architecture is not designed to, and does not prevent, malicious behavior at the level of native code

It is common convention that the javadocs for a permission class give a description of the risks associated with granting a particular permission.

5.1.5 Positive versus Negative Permissions

The `Permission` class hierarchy denotes only positive permissions. This means that if a permission is present in the security policy, the said permission is granted. The denial of a permission is implicitly expressed by the absence of the said permission rather than by the presence of a "negative" permission.

The lack of negative permissions does not mean that they cannot be introduced in the future. However, restricting the design to only positive permissions has significant benefits for simplicity and good performance; no conflict can exist between two positive permissions, in the sense that there is no danger that access granted by one permission is denied by the other. Consequently, when you examine a security policy to decide what permissions to grant to some code, you do not need to check for conflicts. Without negative permissions, you can determine that the set implies the said permission as soon as you find one permission within the set that implies the said permission. These benefits to SDK implementation are also benefits to security policy administration.

The lack of negative permissions, on the other hand, does not allow you to specify a policy conveniently, such as "grant all file system access except for this particular file." However, this loss of convenience is not really a loss of functionality, because a negative can be expressed by the complement of a positive. It seems that with additional "syntactic sugar" in more powerful policy-processing tools, one can preprocess a policy with negative permissions and translate the policy into one with only positive permissions. How this issue plays out in practice remains to be seen.

Even though the permission classes within the Java runtime do not support a negative semantic, this does not preclude a custom security policy implementation from supporting such a semantic. One possibility is to implement a `Policy` provider and heterogeneous permission collection classes that support the notion of a constraint-based matching algorithm within the logic of the `implies` methods.

5.2 Describing Code

The security policy is essentially an access control matrix that describes code according to its characteristics and the permissions it is granted. The `CodeSource` and `Principal` classes are used to describe code.

5.2.1 `CodeSource`

Code is fully characterized by three things. One is its origin, or its location as specified by a URL. The second, applicable if the code is signed, is the set of digital certificates containing the public keys corresponding to the private keys used to sign the code. (*Note:* Digital certificates are described in Section 8.1.) The first two characteristics are captured in the class `java.security.CodeSource`, which can be viewed as a natural extension of the concept of a code base within HTML, although it is important not to confuse the `CodeSource` class with the `CODEBASE` tag in HTML. The third characteristic of code is who is running the code. This is discussed briefly in Section 5.2.3 and in greater detail in Chapter 8.

An example of a `CodeSource` URL, where the code is an applet packaged in a JAR file called `foo.jar` that resides at the Web address `http://java.sun.com/classes/`, would be the URL

```
http://java.sun.com/classes/foo.jar
```

If the JAR file is signed, it will contain digital signatures for individual entries in the JAR file or for the entire JAR file itself. In this case, the corresponding `CodeSource` will contain not only the code's URL but also the certificates that correspond to the signing keys. Actually, it may contain a chain of certificates. Certificate chains are described in Section 8.2. Note that if the signatures cannot be verified, the JAR file must be viewed as unsigned. Verification could fail either because the content of the JAR file was modified such that one or more entries no longer match its signature or because the signing keys are not trusted (refer to Section 8.3).

`CodeSource` objects are used by `Policy` implementations to specify code granted specific permissions by the security policy. `CodeSource` objects are also used to specify code that is running and that may need to be checked at some point for having the permissions required to perform security-sensitive operations.

Following are the most important methods in the `CodeSource` class:

```
CodeSource(URL url, java.security.cert.Certificate certs[]);
public boolean equals(Object obj);
public boolean implies(CodeSource codesource);
```

We intentionally made `CodeSource` immutable by including both the URL and the certificates in the constructor and by making copies of the certificates instead of merely keeping references to those certificate objects. Note that the URL itself is already immutable, so there is no need to make a clone of it. Making a `CodeSource` object immutable ensures that it can be passed around without its integrity being compromised. Its integrity is important because, as we discuss in the next chapter, access control decisions are made based partly on the `CodeSource` of running code. For example, a code fragment from a designated `CodeSource` may be allowed to write to the local file system, whereas code from other places is prohibited from local file system access. If a latter kind of `CodeSource` object could be illegally mutated to become identical to the former `CodeSource` object, code from the latter would gain illegal access to the local file system, thereby causing a security breach.

One might notice that only private keys are needed to create signatures and only public keys for verification. Thus, a syntactically complete certificate seems unnecessary. So why does the interface in `CodeSource` use only certificates and not raw public keys? The answer is twofold. For simplicity, and because of the security issues with relying on raw public keys, the best practice is to rely on digital certificates. In theory, both interfaces can exist, where one interface uses public keys and the other, certificates. But having both would be redundant and add complexity to the underlying algorithm and code.

Using certificates exclusively should not cause any problem, because given any public and private key pair, you can easily produce a self-signed certificate that encloses the public key. In fact, the tool used to generate keys in the Java 2 SDK, `keytool`, always generates a self-signed certificate when generating a key. A self-signed certificate normally would not convey any significance to the key enclosed inside, except to serve as a medium to transport the key.

Moreover, using certificates instead of public keys makes it easier to carry around important information that might be contained inside a certificate but that cannot be expressed by the public key itself. That is, a certificate contains more than just the public key, as described in Chapter 8.

For example, because `CodeSource` objects contain not only certificates but also their supporting certificate chains, one can validate an entire certificate chain all the way up to the root Certification Authority (CA). Such validation information is valuable for auditing purposes.

5.2.2 Testing for `CodeSource` Equality and Using Implication

Correctly testing for equality between two `CodeSource` objects is critically important because such a comparison is central to security policy decisions. Two `CodeSource` objects are considered equal if their URL locations are identical and if the two sets of certificates contained in the two objects are identical. Note that the two sets of certificates might not be stored in the same order in the array.

Sometimes, it is convenient to specify a first `CodeSource` object that is more general than a second `CodeSource` object so that any code qualified by the second will also be qualified by the first. In this case, the first `CodeSource` "implies" the second `CodeSource`. For example, a `CodeSource` with the origin `http://java.sun.com/classes/` is more general than one with the origin `http://java.sun.com/classes/foo.jar`.

With such a relationship based on "implication," security policy can be simplified by granting permissions to a general `CodeSource` object, which will implicitly grant the same permissions to any more specific `CodeSource` object. For example, you can give to `http://java.sun.com/classes/` permission to access the local file system, meaning that you give the same permission to all code residing on that Web page.

Note that URL matching here is lexical and does not deal with proxies or redirects. For example, a policy can include an entry that specifies a URL `http://java.sun.com`. Such an entry is useful only when you can obtain code directly from HTTP for execution. If the Web server redirects this URL to a different one, this policy entry has no effect.

To specify URLs for the local file system, you can use file URLs. For example, to specify files in the `/home/gong/temp` directory on a UNIX operating system, you can use `file:/home/gong/temp/*`. To specify files in the `temp` directory on the C drive on a Windows system, you can use `file:/c:/temp/*`.

Obviously, strict and precise rules must be followed in order to determine whether one `CodeSource` object implies another. When the `this.implies(CodeSource codesource)` method is called, it returns `true` if `this CodeSource` object implies the `codesource` instance passed in as the parameter. More specifically, this method makes the following checks, in the following order. If any check fails, it returns `false`. If they all succeed, it returns `true`.

1. `codesource` must not be `null`.
2. If this object's certificates are not `null`, all of them must be present in `codesource`'s certificates.
3. If this object's location (`getLocation()`) is not `null`, the following checks are made against its location and `codesource`'s location.
    a. `codesource`'s location must not be `null`.
    b. If this object's location equals `codesource`'s location, immediately return `true`. Otherwise, continue.
    c. This object's protocol (`getLocation().getProtocol()`) must be equal to `codesource`'s protocol.

d. If this object's host (`getLocation().getHost()`) is not `null`, the `SocketPermission` constructed with this object's host must imply the `SocketPermission` constructed with `codesource`'s host.
e. If this object's port (`getLocation().getPort()`) is not equal to -1, that is, if a port is specified, it must equal `codesource`'s port.
f. If this object's file (`getLocation().getFile()`) does not equal `codesource`'s file, the following checks are made:
   - If this object's file ends with a `/`, `codesource`'s file must contain this object's file as a prefix.
   - If this object's file ends with `/*`, `codesource`'s file must reference a class or JAR file in the directory pointed to by this object's file without the trailing `*`.
   - If this object's file ends with `/-`, `codesource`'s file must reference a class or JAR file in the directory pointed by this object's file without the trailing `*` or recursively any of its subdirectories.
   - In all other cases, `codesource`'s file must, as a prefix, contain this object's file with a `/` appended.
g. If this object's reference (`getLocation().getRef()`) is not `null`, it must equal `codesource`'s reference.

For example, consider `CodeSource` objects with the following locations and `null` certificates:

```
http:
http://*.sun.com/
http://java.sun.com/classes/
http://java.sun.com/classes/foo.jar
```

All these imply the `CodeSource` object with the location `http://java.sun.com/classes/foo.jar` and `null` certificates. The reason is that `http:`, `http://*.sun.com/`, and `http://java.sun.com/classes/` all include `http://java.sun.com/classes/foo.jar` as a special case.

Two different `CodeSource` objects refer to the same code source if they imply each other.

`CodeSource` implements the interface `java.io.Serializable`. Therefore, we provided customized private methods `writeObject` and `readObject` for serialization.

Following is a sample use of the `CodeSource` class. When defining a class that is loaded from either the local host or a remote host, you must calculate its code source. This requirement exists so that subsequent requests to the security policy can be made in order to determine the permissions accorded to the code. The `findClass` method in the `java.net.URLClassLoader` class (see Section 4.6) executes the following code segment:

```
...
// Construct a URLClassPath for the URLs this URLClassLoader
// uses to search for classes
URLClassPath ucp = new URLClassPath(urls);
...
// Find the class file named by path, where path is the
class
// name (passed to findClass) with ".class" appended and any
```

```
// periods replaced with forward slashes.
Resource res = ucp.getResource(path, false);
...
// Get the class file bytes, certificates, and URL
byte[] b = res.getBytes();
java.security.cert.Certificate[] certs =
res.getCertificates();
URL url = res.getCodeSourceURL();
// Construct a CodeSource from the URL and certificates
CodeSource cs = new CodeSource(url, certs);
...
// Call the superclass SecureClassLoader's defineClass
method
// to define the class and associate it with its appropriate
// protection domain. Here name is the class name passed to
// findClass.
return defineClass(name, b, 0, b.length, cs);
```

5.2.3 `Principal`

The `Principal` interface represents the abstract notion of a *principal*, which is any entity, such as an individual or a service, to which authorizations, and thus accountability, may be attributed. `Principal`s are used to represent a user or service when that user/service has been authenticated and then subsequently associated with the current execution context such that further execution is considered to be done on behalf of that user/service. Whereas a `CodeSource` indicates two of the three characteristics of running code—its location and signer certificates—`Principal`s specify the third characteristic: an indication of who is running the code. `Principal`s are described further in Section 8.4.

5.3 `ProtectionDomain`

To assign permissions to a class, one could follow the straightforward approach of encapsulating all the permissions granted to a class, represented by various `Permission` objects, to an instance of a `Permission`s class (described in Section 5.1.3 and then associating this permission set with the class via an interface in the base class `java.lang.Class`. However, linking a permission set so directly with a class would lead to a rigid API that could not easily be extended. For example, suppose that access control checks were to be performed based not only on permissions granted to the class but also on the name of the principal currently running the code. To do this, the `Class` class would have to be extended with additional interfaces, thus cluttering the base class. In fact, this feature of extending checks based on principals in addition to code characteristics was added after the initial Java 2 release without any impact on the Class class, as described in Section 5.3.1.

To facilitate extensibility, in the Java 2 SDK, each class is associated at class loading time with an instance of a `ProtectionDomain` class, which encapsulates the class characteristics, vis-à-vis a `CodeSource`. The permissions granted to the specified code may be statically bound in the `ProtectionDomain` instance or may be dynamically determined when an access control check is performed. Thus, classes belong to protection domains and are indirectly associated with the permissions granted to the code specified by their domains.

According to the classical definition of a protection domain [111], a domain is scoped by the set of objects currently directly accessible by a principal, where a principal is a computer system entity

to which authorizations—and as a result, accountability—are granted. Thus, the sandbox in JDK 1.0 is, in a sense, a protection domain with a fixed boundary.

In Java 2, each class belongs to a protection domain. The Java runtime maintains the mapping from code to protection domains. The mapping from a class to its domain is set only once, when the class is defined, and cannot be changed during the lifetime of the `Class` object.

The definition of the class `java.security.ProtectionDomain` is fairly straightforward. It encapsulates

- A `CodeSource` describing the code origin and signing certificates.
- A `Principal` array that may be set during execution to indicate who is executing the code.
- A `ClassLoader` reference, possibly `null`, to the class loader defining the class.
- A `PermissionCollection` containing permissions granted to the code statically when the class was loaded. The dynamic permissions for the protection domain are determined by consulting the policy.

The main `ProtectionDomain` methods are discussed in the following subsections.

5.3.1 `ProtectionDomain` Constructors

`ProtectionDomain` has two constructors:

```
public ProtectionDomain(CodeSource codesource,
    PermissionCollection permissions);
ProtectionDomain(CodeSource codesource, PermissionCollection
     permissions, ClassLoader classloader, Principal[]
principals)
```

Typically, a `ProtectionDomain` is instantiated by a class loader. The first time a given `CodeSource` is encountered during class loading, a `ProtectionDomain` is created. That is, for each class loader instance, all classes with the same `CodeSource` map to the same `ProtectionDomain`. Subsequent encounters with the same `CodeSource` by the class loader may reuse a reference to a cached `ProtectionDomain`.

The first constructor creates a `ProtectionDomain` with the specified `CodeSource` and `PermissionCollection`, a `null ClassLoader`, and an empty `Principal` array. If a non-`null PermissionCollection` is supplied, it is made immutable by setting it to read-only, to ensure its integrity and consistency. When we first developed Java 2, we made this design decision as an added integrity precaution; in retrospect, however, it complicated our ability to make security policy more flexible. When the `PermissionCollection` is not `null`, the permissions are statically granted to the `ProtectionDomain` and therefore the security `Policy` will not be consulted on subsequent permission checks against this `ProtectionDomain`. This behavior is consistent across all releases of J2SE. Only this first constructor existed when the Java 2 platform was introduced. In the releases prior to J2SE 1.4, permissions were granted to code solely on the basis of the characteristics encapsulated in a `CodeSource`. This constructor remains in order to maintain backward compatibility with versions prior to J2SE 1.4.

An optional package, the Java Authentication and Authorization Service (JAAS), was developed for use with J2SE 1.3. JAAS enabled services to authenticate and enforce access controls on users. JAAS extended the access control architecture of the Java 2 platform in a compatible fashion to

support granting permissions based not only on the code location and signers but also on *who* was executing the code. For example, code executed by one user may be granted certain permissions, whereas the same code executed by another user may be granted different permissions. JAAS, described further in Chapter 8, is no longer an optional package; it was integrated into the J2SE 1.4 release. To facilitate the integration, a second constructor was added so a `ProtectionDomain` could encapsulate the additional characteristic of a user (or service), as represented by one or more `Principal` objects. Additionally, a reference to the defining loader of the class is necessary to disambiguate two similar `ProtectionDomains`.

J2SE 1.4 greatly enhanced the support for dynamic policy, that is, for dynamically determining the permissions granted to code when a permission check is invoked, rather than simply when the code's `ProtectionDomain` was instantiated. To support dynamic policy, the `SecureClassLoader` (see Section 4.5) uses the second constructor to create a `ProtectionDomain`. No longer are all the objects that a `ProtectionDomain` encapsulates statically bound at class loading time. The `CodeSource` is, as before, but the permissions are not. The `SecureClassLoader` doesn't statically assign any permissions, although subclasses of `SecureClassLoader` might. For example, the implementation of `URLClassLoader` assigns permissions to enable the class to access resources from its point of origin.

During the process of defining a class, a `null` is specified for the `Principal`s. This is necessary to avoid granting the permissions to all the principals active on the execution thread at the time the class is defined and to all threads of execution that use instances of the class. That is, it is not possible to determine while loading classes which principal(s) should be trusted to execute code from the specified `CodeSource`.

The following code segment from `java.security.SecureClassLoader` demonstrates one example of how a protection domain is constructed:

```
PermissionCollection perms = getPermissions(cs);
ProtectionDomain pd = new ProtectionDomain(cs, perms, this,
null);
```

In this example, the `SecureClassLoader` first calls `getPermissions`, passing it the `CodeSource`, to compute the statically bound permissions to associate with the `ProtectionDomain`. The `SecureClassLoader getPermissions` method simply returns a new, empty `Permissions` object. Thus, the collection of static permissions should be an empty set unless a subclass of `SecureClassLoader` overrides the `getPermissions` method, as is done by `URLClassLoader` (Section 4.6). `SecureClassLoader` constructs the `ProtectionDomain`, based on the `CodeSource` of the class being defined, the static permissions supplied by calling `getPermissions`, a reference to `this` defining class loader (used to scope the permission grants) and a `null` for the array of `Principal`s.

5.3.2 `ProtectionDomain implies` Method

The `ProtectionDomain implies` method has the following signature:

```
public boolean implies(Permission permission);
```

This method is called for each `ProtectionDomain` on the call stack, whenever an access control decision is requested, to see whether the requisite permission is granted to the code—and

possibly `Principal`s—indicated by this `ProtectionDomain`. (The full access control checking behavior is described in Chapter 6.)

This method checks whether the `ProtectionDomain` "implies" the permission expressed by the `Permission` object. If the `ProtectionDomain` was assigned *just* static permissions—that is, the first constructor was called with a non-`null` `PermissionCollection`—the encapsulated `PermissionCollection`'s `implies` method (Section 5.1.3) is invoked. Otherwise, the current `Policy` is obtained and its `implies` method invoked, thus delegating the policy decision to the installed policy provider.

5.3.3 `ProtectionDomain` Finer Points

A `ProtectionDomain` object may contain sensitive information, so access to it is security checked with a runtime permission. The following code segment from `java.lang.Class` demonstrates how the method `getProtectionDomain` controls access to the protection domain of the class:

```
public java.security.ProtectionDomain getProtectionDomain()
{
   SecurityManager sm = System.getSecurityManager();
   if (sm != null) {
      getPDperm = new
RuntimePermission("getProtectionDomain");
      sm.checkPermission(getPDperm);
   }
   return protectiondomain;
}
```

As you will see in Chapter 6, this allows the `ProtectionDomain` of the class to be returned only if the code in the current execution context has been granted the `RuntimePermission` with the target name `"getProtectionDomain"`.

A number of additional points are worth discussing. First, protection domains in Java 2 are created on demand as new classes are loaded into the runtime. In the Java 2 default implementation, classes belonging to the same domain are loaded by the same class loader. This implementation detail is natural but not necessary. Classes belonging to the same domain are granted the same permissions, but the reverse is obviously not true, as there may be classes that have the same permissions but that are from different code sources and thus belong to different domains.

Second, out of the many protection domains created during the lifetime of the Java runtime, one protection domain is special: the system domain. The system domain comprises all classes that are considered part of the system core, trusted computing base, or kernel in operating system terminology. For historical reasons, system classes are always loaded by the bootstrap class loader. For the time being, you need remember only that code in the system domain is automatically granted all permissions. It is important that all protected external resources, such as the file system, the networking facility, and the screen and keyboard, are directly accessible only via system code, which mediates access requests made by less trustworthy code. Note that although system classes have a `null` class loader (Section 4.2), their protection domain is a non-`null` reference to a `ProtectionDomain` that has been statically assigned the `AllPermission` permission.

Moreover, the indirection between a class and its permissions via a protection domain has an interesting benefit of enabling Java virtual machine vendors to perform implementation

optimizations. For example, recall that it is desirable in some cases to change the permissions granted to some code during the lifetime of a Java runtime. By maintaining stability in the reference of the `ProtectionDomain` object, one can determine whether two classes belong to the same domain and then apply various optimization techniques.

Finally, note that the protection domains also serve as a convenient point for grouping and isolating units of protection within the Java runtime environment. For example, different domains may be prevented from interacting with each other. This could be done by using distinct class loaders to load classes belonging to different domains in such a way that any permitted interaction either must be through system code or explicitly allowed by the domains concerned. The reason is that in the Java virtual machine, a class is distinguished by itself and the `ClassLoader` instance that defined the class. Thus, a class loader defines a distinct namespace. It could be used to isolate and protect code within one protection domain by refusing to load code from different domains (and with different permissions).

This point raises the issue of accessibility, that is, what is visible to an object and what methods can an object invoke and on what other objects. In designing the Java 2 security architecture, we examined existing coding practices that use accessibility features that make one object visible to another. We found that accessibility needed to remain flexible, especially in server programs, without regard to the particular security policy being enforced. So we decided to maintain existing accessibility customs and rules, thus making accessibility orthogonal to security. In other words, it is up to the application programmer to decide whether and how objects and methods should be hidden from one another. In this sense, the Java security mechanism is much more than a classical capability system.

Note that, technically, we could have enforced stricter isolation between domains. However, this would have created a need for a new set of interfaces for interdomain communication, similar to interprocess communication (IPC). Also, existing applications would have had to be rewritten to use the new interfaces. To enforce complete isolation, we might have had to redesign some shared system classes and their static fields [52]. Thus, the decision to leave accessibility separate from security was the best available solution at the time. However, an effort is under way within the Java community to develop an application isolation API.

5.4 Security Policy

The security behavior of a Java runtime environment is specified by the security policy in effect during runtime. In abstract terms, the security policy is a typical access control matrix that says what system resources can be accessed, in what fashion, and under what circumstances. For example, one entry in the matrix shown in Figure 5.2 says something like, "When running an applet downloaded from `http://java.sun.com`, allow it to read the file *x*." More specifically, a security policy is a mapping from a set of properties that characterize running code to a set of access permissions granted to the code.

Figure 5.2. Policy matrix

User	Code	Permissions
	java.sun.com	read file x
admin	backup.sun.com	read /home/gfe/-
	CharlesSchwab Applets, Signed	read and write /home/gong/stock
	CharlesSchwab Applets, Signed and Unsigned	connect and accept bankofamerica.com

In J2SE, the expression of policy is declarative in nature, that is, nonprogrammatically specified external to the Java runtime environment. The advantages of this approach are many, but the most obvious is the flexibility it supplies to the deployer. The security policy can actually be composed of a *set* of policies that can be configured by a system administrator or user. There can be multiple forms of representation of a policy. For example, it may be represented as a flat ASCII file, as a serialized binary file representing a `Policy` subclass, or as a database. The default `Policy` provider implementation in J2SE expects the policy to be specified in a particular textual format in an ASCII file. The specification of this format is described in Chapter 12.

The source location(s) for the policy information used by the `Policy` object are expressed by URL(s) as values of one or more security properties. This is further described in Section 12.6. The design decision of using URLs to locate the external policy is flexible and should suffice for most deployment environments. For example, the policy can be retrieved via HTTP, LDAP (Lightweight Directory Access Protocol), or another protocol.

So that the security mechanism inside the Java runtime environment can consult the policy, the policy contents are necessarily represented internally by an instance of a `Policy` class that is a subclass of the abstract class `java.security.Policy`. Because there is no limitation on who can instantiate such an object, multiple instances of the `Policy` object could exist at the same time. Nevertheless, only one `Policy` object is in effect at any time, in the sense that it is the one consulted when making security policy decisions.

The security policy is represented by a `Policy` subclass that provides an implementation of the abstract methods in the `Policy` class. Following are `Policy`'s most important methods:

```
public static Policy getPolicy();
public static void setPolicy(Policy policy);
public abstract PermissionCollection
getPermissions(CodeSource

codesource);
public PermissionCollection getPermissions(ProtectionDomain
                                           domain);
public boolean implies(ProtectionDomain domain,
                       Permission permission);
public abstract void refresh();
```

The `Policy` object maintains a runtime representation of the policy and is typically instantiated either at the Java virtual machine start-up time or when the security policy is used for the first time. It may be changed later via a secure mechanism, such as by calling the static `setPolicy` method. The currently installed `Policy` object can be obtained by calling the static `getPolicy` method.

The `refresh` method causes the `Policy` object to refresh or reload its current configuration. How this is done is implementation dependent. For example, if the `Policy` object obtains its policy content from configuration files, a call to `refresh` will typically cause it to reread the policy configuration files. The description of `UnresolvedPermission`s in Section 5.1.2 portrays some policy update issues.

The `Policy` component is designed as a provider structure. That is, there is a default implementation, but an alternative implementation can be provided and installed, if desired. We designed the `Policy` component as a provider structure because we wanted to instill enough flexibility so that the policy content can be obtained in arbitrary ways. It would have been impossible to anticipate the various possible ways for doing this and then design sufficient APIs

for them. Refer to Section 12.6.5 for more information on deploying an alternative `Policy` provider.

The `getPolicy` method is `public static` so that anyone can call it, but policy content can be sensitive. Thus, a suitable security check is invoked inside the `getPolicy` method so that only code that has permission to obtain the policy—code granted the `SecurityPermission` with name `"getPolicy"`—can do so. Similarly, a security check is invoked, when the `setPolicy` method is called, to ensure that only code granted the `SecurityPermission` with name `"setPolicy"` will be able to change the `Policy` implementation to be used. If the calling code does not have the required permission, a `java.lang.SecurityException` is thrown, thereby indicating that a security-sensitive operation was attempted and then denied, due to insufficient access permission.

The following code segments demonstrate the use of the Java 2 permission model, whereby if a `SecurityManager` has been installed (see Section 6.1), a security-sensitive operation can be performed only if the required permission has been granted:

```
public static Policy getPolicy() {
   SecurityManager sm = System.getSecurityManager();
   if (sm != null) sm.checkPermission(new
      SecurityPermission("getPolicy"));
   return policy;
}

public static void setPolicy(Policy policy) {
   SecurityManager sm = System.getSecurityManager();
   if (sm != null) sm.checkPermission(new
      SecurityPermission("setPolicy");
Policy.policy = policy;
}
```

The `getPermissions(CodeSource)` and `refresh` methods are abstract. Therefore, they must be implemented by a subclass of the `Policy` class.

The contract for the `getPermissions` methods requires that the returned set of permissions must be an instance of a new mutable `PermissionCollection` object that must support heterogeneous `Permission` types.

The methods that take a `ProtectionDomain` as a parameter—`getPermissions(ProtectionDomain)` and `implies(ProtectionDomain)`—were introduced in J2SE 1.4 to support intrinsically the dynamic binding of policy to protection domains.

The default implementations of these methods in the `Policy` class preserve backward compatibility with legacy `Policy` providers, where permissions were statically determined by consulting the `Policy` when a class was defined and the class's protection domain instantiated. The concrete implementation of `Policy` from Sun Microsystems overrides these methods to handle the dynamic binding of policy to protection domains.

5.5 Assigning Permissions

Previous sections covered the basics of security policy, code source, and the `Permission` class hierarchy. This section discusses how permissions are computed for running code.

When loading a new class, the class loader assigns the class to its `ProtectionDomain`. The `ProtectionDomain` encapsulates the class characteristics, such as location and signers, and the static permissions granted to the specified code, as described in Section 5.3.

Permissions may be granted to a class before the class loader completes the class definition, but it is perfectly legitimate to delay the instantiation of the granted permissions until a security check occurs. This optimization allows a Java application that does not call for security checks to execute faster and with a smaller footprint. Even for a Java program that does trigger a security check, this optimization allows it to start up faster. Note that if the content of the policy is changed between the time the `Policy` class is instantiated and the time the first security check is invoked, the presence of this optimization technique may result in the `Policy` object having more up-to-date content.

Keeping the policy content up-to-date is, of course, a good thing. Section 5.4 discusses policy update using the `refresh` method. It is possible that the permissions already granted to a class will be changed or even withdrawn after the class is defined, such as during a revocation procedure after a security incident. Such alterations are considered legal, as long as they are also controlled with the appropriate permissions.

It must be emphasized that permissions are effectively granted to classes, which are static Java code, and not to objects, which are instances of classes. The primary reasons for this are to reduce complexity and increase manageability. Objects are runtime entities, so they do not exist in a static state. But the security policy must exist in a static state and independent of any particular Java runtime environment, so it cannot possibly refer to objects. Also, for the sort of security policies considered here, the same policy should be enforced no matter how objects are instantiated. In addition, the number of different classes tends to be a lot smaller than the number of different objects. Even if you want to support a security policy that is dependent on the runtime environment, the right way is not to grant permissions to objects but rather to perform security checks that take into account the runtime environment and perhaps who is running the code.[1] We return to this subject in Chapter 6.

[1] The security policy could grant permissions to interfaces, too, but this is immaterial, as interfaces alone do not get instantiated into objects that cause security checks to occur.

Finally, recall that the security policy, in essence, can be represented with a list of entries, each being of the form (`CodeSource`, `Principal`, `Permission`), thereby indicating that code from the named code source, executed by the specified `Principal`—user or service—is given the named permission. A `CodeSource` consists of a URL conveying the code location and certificates representing the code signers. All the code characteristics—URL, certificates, and `Principal`—in a policy entry are optional. If a characteristic is omitted, it indicates that the characteristic's value doesn't matter. If you don't specify a URL, for example, the specified permission is granted to code no matter where it is located. Clearly, for a given piece of code, its code source can match multiple entries in the policy. In this case, the code is granted the union of all permissions granted in each matched entry in the policy. In other words, permission assignment is *additive*. For example, if code that is signed by A gets permission X—no matter where the code is located or who is running it—and code signed by B gets permission Y—no matter where the code is

located or who is running it—code signed by both A and B gets permissions X and Y. Similarly, if the URL `http://java.sun.com` is given permission X and the URL `http://java.sun.com/people` is given permission Y, an applet from `http://java.sun.com/people/gong` gets both X and Y, assuming that the signers and users match. For further details of the matching algorithm, refer to the `implies` method in Section 5.2.1.

5.6 Dynamic Security Policy

The technique of deferred binding of permissions to protection domains is known as dynamic policy. However, even if the Java runtime environment subscribes to this technique, not all permissions accorded to a protection domain are computed dynamically. For instance, class loaders may statically bind permissions to a protection domain. An example of this is that code downloaded from an origin server is granted permission to connect back to the origin server. Specifically, the implementation of `URLClassLoader` adds into the static permission set of the `ProtectionDomain`—of the class being loaded—permissions based on the URL of the `CodeSource` of the class.

Prior to J2SE 1.4, all permissions assigned to a class were statically bound to its `ProtectionDomain` when the `ProtectionDomain` was created, that is, the first time the `CodeSource` for the class was encountered. This static binding restricted the effectiveness of refreshing security policy, calling the `Policy refresh` method. This depended on how the `Policy` provider and class loaders were implemented. For example, the `Policy` implementation might not have supported recomputing the permissions accorded a `ProtectionDomain` following a refresh. Also, a class loader might have statically bound permissions to protection domains and cached the protection domains as an optimization, thus preventing a refresh from having any effect.

J2SE 1.4 intrinsically supports dynamically determining the permissions granted to code, rather than statically (also known as eager binding) when the code's `ProtectionDomain` is created. The binding of the permissions specified in `Policy` is deferred until the `ProtectionDomain` is evaluated in the context of a security check. This change enables the `Policy` provider to dynamically derive the permissions granted to a class. The approach shifts responsibility for assigning permissions to `ProtectionDomains` from the `ClassLoader` to the `Policy` implementation, which enables the deployment of dynamically changing security policies. In hindsight, we should have done it this way from the beginning.

Now when the `SecureClassLoader` constructs a `ProtectionDomain` for a class, it no longer consults `Policy` within its implementation of the `getPermissions` method. Rather, the `SecureClassLoader getPermissions` method returns an empty `Permissions` (a subclass of `PermissionCollection`) object. To remain backward compatible, derivative `ClassLoaders` continue to add permissions to this collection. Note that these permissions are static and according to the specification are marked read-only once a `ProtectionDomain` is constructed with a non-`null PermissionCollection`. Our design retains this semantic.

Formerly, the `implies` method of `ProtectionDomain` returned the result of `PermissionCollection implies`. Now the implies method of `ProtectionDomain` must first test the static permissions for implication. If that fails and the `ProtectionDomain` was constructed for dynamic permissions, `ProtectionDomain` calls the implies method in `Policy`, passing this as the `ProtectionDomain` argument. The implies method in `Policy` collects the `PermissionCollection` for the given

`ProtectionDomain`, either from a cache of previous evaluations or from the external policy representation, and then evaluates whether the collection implies the requisite permission.

Adding support for dynamic policy has some important yet subtle distinct qualities. The former design assumed that a `Policy` provider implementation could compute a full enumeration of all the permissions granted to a given `CodeSource`. However, many access management systems tend to be able to answer whether a given security context has the requisite permission rather than being able to list all the permissions accorded a given security context. Also, the interface to the `Policy` class was too narrow to carry additional context, such as the `Principal`s associated with the current thread of execution. A consequence of enabling dynamic policy is that it potentially adds complexity to a proper implementation of a `Policy` provider. For example, a `Policy` provider must accommodate the case in which the actions of a requisite permission spans the collection statically bound to the protection domain and the permissions derived dynamically.

This chapter has detailed the security classes that provide the basic machinery to qualify system entities such that we can specify the security policies governing the resources the entities may access. The next chapter describes how this machinery is used to enforce security policy in the Java runtime.

Chapter 6. Enforcing Security Policy

If effectively enforced, the law confers a real and great good.

—Theodore Roosevelt

The previous chapter describes how security policy, telling what permissions are granted to various groups of code with various characteristics, is specified separately from the Java runtime environment and represented at runtime within a `Policy` object. That chapter documents classes used to characterize code, both code granted permissions by the policy and code that is running and whose permissions will need to be checked by consulting the policy whenever a security-sensitive resource access is attempted.

This chapter focuses on how the security policy is enforced. The chapter describes the classes involved in enforcement (`SecurityManager`, `AccessController`, `AccessControlContext`, `and DomainCombiner`) and the access control algorithm used to determine whether to allow a resource access, based on the current execution environment and the permissions granted by the policy.

6.1 `SecurityManager`

The `java.lang.SecurityManager` class, designed into the original release of JDK 1.0, is the focal point of access control. The security manager is called whenever a decision is needed to determine whether to grant or deny a request for accessing a sensitive resource. As an example of a `SecurityManager` class, the `sun.applet.AppletSecurity` class, a subclass of `SecurityManager`, implemented the sandbox security model in JDK 1.0. Recall from Section 2.2 that according to this model, applications—classes residing on the local file system—are given full system access, whereas applets—remote classes loaded over the network—are denied all but the most essential privileges.

6.1.1 Example Use of the Security Manager

In the Java 2 platform, code is allowed access to a security-sensitive resource only if that code has been explicitly granted the corresponding required permission by the security policy currently in effect. Policy enforcement may be accomplished by calling the `SecurityManager checkPermission` methods. Any code whose execution requires that a particular permission be granted may call `checkPermission` directly, after first checking to ensure that a `SecurityManager` is installed. For example, the following code segment from `java.lang.Class` demonstrates how its `getProtectionDomain` method restricts access to the protection domain of a class:

```
public java.security.ProtectionDomain getProtectionDomain()
{
   SecurityManager sm = System.getSecurityManager();
   if (sm != null) {
      getPDperm = new
RuntimePermission("getProtectionDomain");
      sm.checkPermission(getPDperm);
   }
   return protectiondomain;
}
```

6.1.2 `SecurityManager` API

Prior to Java 2, the `SecurityManager` class was abstract, so a vendor had to subclass it and create a concrete implementation. This was inconvenient. In Java 2, the class is concrete, with a public constructor. Following are the main `SecurityManager` APIs:

```
public SecurityManager()
protected native Class[] getClassContext()
public Object getSecurityContext()
public void checkPermission(Permission perm)
public void checkPermission(Permission perm, Object context)
```

A `SecurityManager` is itself considered a sensitive resource. Therefore, a security check is placed in the constructor to ensure that only code granted the `java.lang.RuntimePermission` with the name `"createSecurityManager"` can instantiate a `SecurityManager`.

The `getClassContext` method returns the current execution stack as an array of classes. The length of the array is the number of methods on the execution stack. The element at index `0` is the class of the currently executing method, the element at index `1` is the class of that method's caller, and so on. Such a context is useful for determining the current method calling sequence, which is essential knowledge for making an access control decision. This method is necessarily native because introspection should not disturb the Java runtime environment execution context.

The `getSecurityContext` method creates an object that encapsulates the current execution environment. This method's purpose is to create a snapshot of the context so that you can later query whether a security check would have passed if invoked within that context. The default implementation of this method returns an `AccessControlContext` object, which is explained in Section 6.4.6.

Recall that a systemwide security manager may be installed and that access control checking is often done only if that is the case. The `java.lang.System` class manages this security manager with the following relevant methods:

```
public static synchronized void
      setSecurityManager(SecurityManager s)
public static SecurityManager getSecurityManager()
```

In the `setSecurityManager` method, if a security manager has not yet been established, the argument passed in is established as the current security manager. This process is sometimes called *installing the security manager*. If the argument passed in is `null` and no security manager has been established, no action is taken, and the method simply returns. If a security manager has already been installed, a security check is invoked to see whether the caller has the `RuntimePermission` with name `"setSecurityManager"`. If it does, the passed-in argument is installed as the new security manager. Otherwise, a `SecurityException` is thrown.

The `getSecurityManager` method returns the installed security manager or, if no security manager has been installed, `null`. Allowing a security manager to be `null` is not a perfect design; its shortcomings are discussed later in the chapter.

6.1.3 The `checkPermission` Methods

Policy enforcement may be initiated by calling one of the `SecurityManager checkpermission` methods. The method signatures are

```
public void checkPermission(Permission perm)
public void checkPermission(Permission perm, Object context)
```

`The checkPermission` method with a single permission argument always performs security checks within the context of the currently executing thread. When a security check is being invoked within a given context—for example, from within a worker thread A—often the check should be done against a different context, such as thread B. In this case, the second `checkPermission` method should be used, with an appropriate context argument, such as the `AccessControlContext` of thread B.

The `checkPermission` methods were introduced with the release of J2SE 1.2. Prior to their introduction, specialized `check` methods were exposed for each type of resource access. These specialized `check` methods have been superseded by `checkPermission`. Additionally, we modified them to call `checkPermission` and, where practical, removed occurrences of `check` method overloading.

The first `SecurityManager checkPermission` method whose signature is shown earlier checks whether the requested access, specified by the given permission, is permitted, based on the current security policy. If it is permitted, the method returns silently; otherwise, it throws a security exception. The default implementation forwards all calls to the `SecurityManager checkPermission` method to the `java.security.AccessController checkPermission` method, described in Section 6.4.

The second `SecurityManager checkPermission` method checks whether the specified security context is granted access to the resource specified by the given permission, based on the current security policy. Recall that the `getSecurityContext` method creates an object that encapsulates the current execution environment in an `AccessControlContext`. If the context passed in is, as expected, an instance of `AccessControlContext`, the `checkPermission` method on that `AccessControlContext` object is called. A security exception is thrown if the context object is not an instance of `AccessControlContext` or if the resource access is denied.

The two `SecurityManager checkPermission` methods replace the myriad of `check` methods from JDK 1. *x*; thus, the semantics of the required check are no longer hard coded in the names and implementations of those `check` methods. Instead, they are encoded in the permission argument passed to the `checkPermission` method. This simple idea has a tremendous advantage. The implementation of the `checkPermission` call typically involves examining Java runtime internal state and performing complicated algorithms. That implementation can now be reused for all permission types, including those yet to be invented. Thus, to protect a new resource, one simply introduces a new `Permission` class and then places a `checkPermission` call in the appropriate place.

6.2 `AccessControlContext`

An `AccessControlContext` is used to make system resource access decisions based on the context it encapsulates. More specifically, it encapsulates a context and has a `checkPermission` method that is equivalent to the `checkPermission` method in the

`AccessController` class (refer to Section 6.4), with one difference: The `AccessControlContext checkPermission` method makes access decisions based on the context it encapsulates rather than on that of the current execution thread.

Thus, the purpose of `AccessControlContext` is for those situations in which a security check that should be made within a given context needs to be done from within a different context, for example, from within a worker thread.

`AccessControlContext` has the following APIs:

```
public AccessControlContext(ProtectionDomain context[])
public AccessControlContext(ProtectionDomain context[],
                            DomainCombiner combiner)
public void checkPermission(Permission perm)
public DomainCombiner getCombiner()
```

The public constructors create an `AccessControlContext` object with the given set of `ProtectionDomain` objects, thus mimicking the execution context in which objects, which instantiate classes from different protection domains, call each other in the sequence given in the array. The first element in the array corresponds to the protection domain of the most recent class. The constructor taking a `DomainCombiner` as a second argument allows for the customization of the access control machinery of the Java runtime.

6.3 `DomainCombiner`

The `DomainCombiner` interface enables an extrinsic implementation class to augment and dynamically update the `ProtectionDomain`s associated with the current `AccessControlContext`. This interface, along with the changes to `AccessControlContext` and `AccessController`, was added to Java 2 SDK, Standard Edition (J2SDK), 1.3, to support principal-based access control, which is described in greater detail in Section 8.4.7.

A `DomainCombiner` is passed as a parameter to the appropriate constructor for `AccessControlContext`, thus binding the implementation to the access control algorithm described in Section 6.4. The newly constructed context may then be passed to the `AccessController.doPrivileged` method to bind the provided context and associated `DomainCombiner` with the current execution thread. (See Section 6.4 for information on `AccessController` and its `doPrivileged` method.) Once the `DomainCombiner` is bound to the execution context, subsequent calls to `AccessController.getContext` or `AccessController.checkPermission` trigger the `DomainCombiner combine` method to be invoked.

Obviously, an object implementing the `DomainCombiner` interface wields considerable authority, as it participates in assembling the `AccessControlContext` evaluated during policy enforcement. Therefore, the ability to construct an `AccessControlContext` with a `DomainCombiner` requires that the caller be granted the `SecurityPermission` permission with the named target `"createAccessControlContext"`.

The `combine` method takes two arguments. The first argument represents an array of `ProtectionDomain`s from the current execution thread, up to and including the `ProtectionDomain` that issued the most recent call to

AccessController.doPrivileged. The second argument represents an array of inherited ProtectionDomains. The ProtectionDomains may be inherited from a parent thread or from a privileged context.

A prototypical combine method inspects the two input arrays of ProtectionDomains and returns a single array containing the updated ProtectionDomains. In the simplest case, the combine method merges the two stacks into one. In more complex scenarios, the combine method returns a modified stack of ProtectionDomains. The modification may have added new ProtectionDomains, removed certain ProtectionDomains, or simply updated existing ProtectionDomains. Reordering and other optimizations to the ProtectionDomains are also permitted. Typically, the combine method bases its updates on the information encapsulated within the DomainCombiner.

After it receives the combined stack of ProtectionDomains back from the DomainCombiner, the AccessController.getContext method returns a new AccessControlContext, which has the combined ProtectionDomains and the same instance of the DomainCombiner bound to it.

6.4 AccessController

Although the SecurityManager class defines the checkPermission and check methods as interfaces to invoke an appropriate security check, these interfaces do not specify how the security checks are done. In particular, they do not specify under what circumstances a request should be granted or denied. This is necessary because it is almost impossible to anticipate all reasonable ways to enforce a security check. For example, one application developer might want to implement a multilevel security policy [5], whereas another might want to implement support for separation-of-duty policies [72]. One way to achieve the goal of supporting multiple policies is to provide a Policy object with a sufficiently rich expressive power to include all possible policy specifications. This might not be possible, or at best, it might be very difficult. Another way is to override the check methods defined in SecurityManager to implement particular flavors of the security policy and to install the appropriate security managers, depending on the application environment.

Not fully specifying how security checks are done has its drawbacks. One is that developers might write security managers that have inconsistent behavior. For example, two custom SecurityManager classes might implement totally opposite semantics of a check method, thus resulting in inconsistent, adhoc, and possibly dangerous behavior. Another problem is that implementing a SecurityManager is difficult, especially for application developers who are not deeply versed in security. Some programmers tend to hard code a security policy in the check methods without leaving enough room for smooth evolution, whereas others may introduce subtle security bugs.

Thus, there is an urgent need to provide a default implementation that specifies a complete access control algorithm that is general enough to be used in a majority of applications. Developers can readily use such an implementation, and users can expect consistent behavior across different applications and platforms. The default implementation is the java.security.AccessController class. In other words, the SecurityManager checkPermission methods invoke by default the checkPermission method defined in AccessController, and thus SecurityManager essentially delegates security decision making to the AccessController.

The next subsection examines the interface design of `AccessController`. Later sections cover in detail the general access control algorithm that is embodied in this class.

6.4.1 Interface Design of `AccessController`

The `AccessController` class is declared `final`, so it cannot be further subclassed. It has no public constructor; thus, no one can instantiate it. It has only the following static methods:

```
public static void checkPermission(Permission perm)

public static native Object
    doPrivileged(PrivilegedAction action)
public static native Object
    doPrivileged(PrivilegedAction action,
        AccessControlContext context)
public static native Object
    doPrivileged(PrivilegedExceptionAction action)
        throws PrivilegedActionException

public static native Object
    doPrivileged(PrivilegedExceptionAction action,
        AccessControlContext context)
        throws PrivilegedActionException
public static AccessControlContext getContext()
```

The `checkPermission` method checks whether a requested access, as specified by the permission argument, is allowed in the current execution context. If it is, the method returns silently. Otherwise, it throws an `AccessControlException`—a subclass of `SecurityException`—that provides details of the reason for failure. The access control algorithm implemented by `checkPermission` is described in the following subsections. The other methods also are described in the following subsections: `getContext`, in Section 6.4.2; the `doPrivileged` methods, in Sections 6.4.4 through 6.4.6.

6.4.2 The Basic Access Control Algorithm

The decision as to whether to grant access to controlled resources can be made only within the right context, which must provide answers to such questions as who is requesting what and on whose behalf. Often, a thread is the right context for access control. Less often, access control decisions must be carried out among multiple threads that must cooperate in obtaining the right context information. A thread of execution may occur completely within a single protection domain. That is, all classes and objects involved in the thread belong to the identical protection domain. Alternatively, a thread of execution may involve multiple domains, such as an application domain and the system domain. For example, an application that prints a message will have to interact with the system domain, which is the only access point to an output stream.

The `AccessController getContext` method returns, in an `AccessControlContext` instance (Sections 6.2 and 6.4.6), the current execution context. The current execution context is entirely represented by its current sequence of method invocations, where each method is defined in a class, and each class belongs to a `ProtectionDomain`. Thus, you can form a sequence of protection domains for the execution context, and the `AccessControlContext` returned by `getContext` contains such a sequence.

The basic access control algorithm can be summarized in one sentence: *A request for access is granted if, and only if, every protection domain in the current execution context has been granted the said permission, that is, if the code and principals specified by each protection domain are granted the permission*. In other words, the required permission *p* must be an element of the intersection of the permissions within all the protection domains *D* in the current execution context:

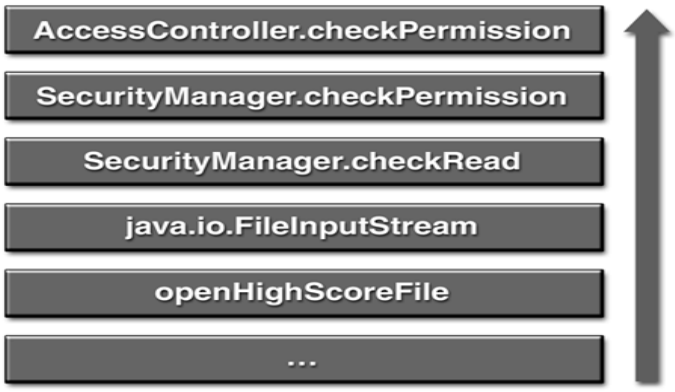
$$p \in \left\{ \bigcap_{i=1}^{n} D_i \right\}$$

The term *caller* is used to denote a protection domain within the context of the current execution, as a protection domain can be associated with multiple contexts. The basic access control algorithm can be expressed in the following constructive manner. Suppose that the current thread traversed `m` callers, in the order of caller 1 to caller 2 through to caller `m`. Then caller `m` invoked the `checkPermission` method, which determines whether access is granted or denied, based on the following algorithm:

```
i = m;
while (i > 0) {
   if (caller i's domain does not have the permission)
      throw AccessControlException;
      return;
   i = i - 1;
}
return;
```

To examine this basic algorithm, suppose that a game applet has a method named `openHighScoreFile` that calls the constructor of `FileInputStream` to open the high-score file, the file that keeps the scores of the top ten players of the game. The constructor calls `checkRead`, which in turn calls the `checkPermission` method inside the security manager. The security manager in turn calls the `checkPermission` method in `AccessController`. At this point, the execution context looks like the snapshot in Figure 6.1.

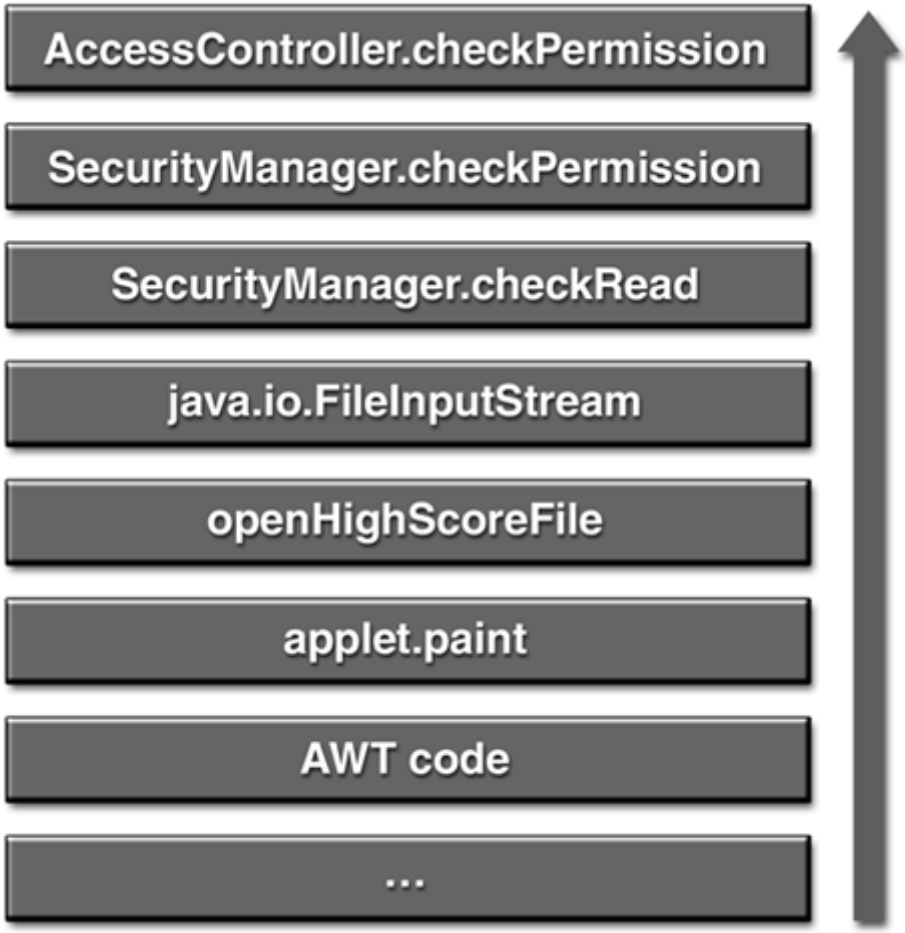
Figure 6.1. Stack frame snapshot



In this example, two distinct protection domains exist within the execution context: the system domain and the domain assigned to the application. The algorithm says that the file can be opened if, and only if, both domains have the `FilePermission` that allows reading the high-score file. Because the system domain by default has all permissions, the algorithm is reduced to checking whether the application has been granted the specified `FilePermission`. If the application has not been granted the permission, the file will not be opened, even though the application tries to enlist the help of system code to do so. This last point is critical because an application domain should not gain additional permissions simply as a result of calling the system domain.

In a reverse situation, a system domain invokes a method from an application domain. For example, the AWT (Abstract Window Toolkit) system code calls an applet's `paint` method to display the applet. Suppose that the applet then tries to open the high-score file from within `paint`. Figure 6.2 shows the execution context.

Figure 6.2. Stack frame execution context



Again, even though it appears that the AWT code triggers the call to `FileInputStream`, the file will not be opened if the applet has not been granted the necessary file permission. Otherwise, the applet would gain immense power simply because system code calls back to its own code. If this were not the case, the system domain would be vulnerable to a "luring" attack, and a serious security compromise could result. The access control algorithm built into the access controller in Java 2 prevents such mishaps.

Thus, code in a less powerful domain cannot gain additional permissions as a result of calling code in a more powerful domain, whereas code in a more powerful domain must lose its power when calling code in a less powerful domain. That is, code doesn't pass on its power to the less powerful domain. This *principle of least privilege* is applied to a thread that traverses multiple protection domains.

Prior to Java 2, any code that performed an access control decision relied explicitly on knowing its caller's status, that is, whether the caller was system code or applet code. This arrangement was fragile because knowing only the caller's status is often insufficiently secure. You frequently also need to know the status of the caller's caller, and so on. Placing this discovery process explicitly on the typical programmer becomes a serious burden and can be error-prone. It also means that the AWT code writer must worry about scenarios under which an applet might behave. The algorithm implemented in `AccessController` relieves this burden by automating the access-checking process.

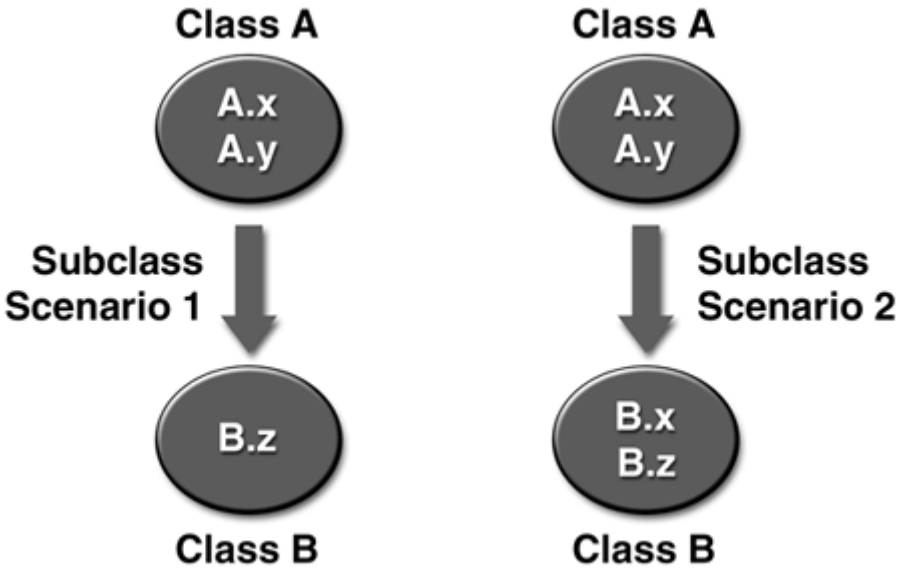
6.4.3 Method Inheritance

The subtle issue of method inheritance needs clarification. The basic access control algorithm, and its extended versions discussed later in the chapter, are defined in terms of a sequence of callers, each represented by a method invocation. The method invocation identifies the class in which the method is defined; the class is linked to the protection domain to which it belongs. Technically speaking, the code and principals the protection domain represents have been granted permissions against which an access control decision is made. Suppose that class B is a subclass of class A. Class A defines a method `x()`, which B inherits but does not override. Further assume that classes A and B belong to two different protection domains. When someone invokes a call on `B.x()`, which protection domain should be considered for this method invocation? Is it the protection domain of class A, which defined and implemented this method? Or is it the protection domain of class B, which simply inherited the method unchanged?

Either choice may seem reasonable under certain conditions. On balance, however, associating the protection domain according to where the method is implemented is more natural because a more powerful class can write its methods in a secure way that allows less powerful classes to inherit them and accomplish tasks for which they themselves would not have had the required permissions. Thus, in the scenario just given, class A's protection domain is examined for the necessary access permissions. Note that if in class B, method `x()` was overridden but otherwise does nothing more than delegate to its parent, super. `x()`, the protection domain to examine would be that of B instead of A, even though the override will not have changed the implementation of the method.

In fact, for this example, both protection domains would be examined, as both will appear as method invocations of the execution thread. The reason is that once a subclass overrides a method call, the superclass cannot be held responsible for the eventual implementation of the method call. In other words, B could have changed the implementation of `x()` in arbitrary ways, so its protection domain should be examined. Both of these scenarios are depicted in Figure 6.3.

Figure 6.3. Method inheritance



6.4.4 Extending the Basic Algorithm with Privileged Operations

The basic algorithm is simple and secure because all code involved in the computation must be granted sufficient permission for the requested access. However, the algorithm can be too restrictive. For example, consider a password-changing application. When a piece of user code calls this application, the user is prompted to type a new password twice—to ensure that the correct password is entered—and then to enter the old password. If the old password matches the one stored in the password file, the new password is stored in the updated password file. Note that the application needs to open the password file for read and write access; assume that the application has been granted sufficient access to do so. Under the basic algorithm, the application cannot open the password file, because it is called by the user code, which does not and should not have permission to access the password file directly. In this case, the application should be given a way to opt out of the basic algorithm in order to open the file, knowing full well what it is doing. In another, similar, example, an applet might not have direct access to certain system properties, but the system code servicing the applet may need to obtain some properties in order to complete its tasks.

To handle such exceptional cases, the `AccessController` class includes a static method, `doPrivileged`. A piece of code that calls `doPrivileged` is telling the Java runtime to ignore the status of its callers and that it itself is taking responsibility in exercising its own permissions. Following is an extended access control algorithm that takes into account privilege status:

```
i = m;
while (i > 0) {
   if (caller i's domain does not have the permission)
      throw AccessControlException;
   if (caller i is marked as privileged)
      return;
   i = i - 1;
}
return;
```

In this extended algorithm, callers must be checked in the same order that they call each other, starting with the most recent caller.

Armed with the call to "invoke one's own privilege," the password-changing application can use the following code segment to open the file, even if the calling code does not have access permission:

```
public void changePassword() {
   // Use own privilege to open the password file.
   AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         // Open file for reading and writing.
         ...
         return null;
      }
   });
   // Verify user's old and new passwords.
   ...
}
```

Here is a detailed look at `doPrivileged`. When executing this method, the Java virtual machine marks the calling thread's stack frame as privileged. In the previous example, the stack frame corresponding to the `changePassword` method is marked privileged. Just before completing the execution of the `doPrivileged` method, the Java virtual machine unmarks the calling thread's stack frame, thereby indicating that it is no longer privileged.

The `doPrivileged` method called in the preceding example takes an object of type `java.security.PrivilegedAction` and invokes its `run` method, which effectively asserts the privileges of the calling class such that all other classes on the execution thread whose methods were invoked prior to the calling class's method are not considered as part of the context of a security check. The implementation guarantees that the assertion of privileges will exist only while the `run` method is executed, even if execution of `doPrivileged` is interrupted by an asynchronous exception.

`PrivilegedAction` is an interface with a single method, named `run`, that returns an `Object`. The preceding example shows creation of an implementation of that interface, using an anonymous inner class; a concrete implementation of the `run` method is supplied. When the call to `doPrivileged` is made, an instance of the `PrivilegedAction` implementation is passed to it. The `doPrivileged` method calls the `run` method from the `PrivilegedAction` implementation after marking the current stack frame as privileged and returns the `run` method's return value as the `doPrivileged` return value, which is ignored in this example.

By calling `doPrivileged`, the caller is merely enabling privileges it already has. This is important to understand. A block of code never gains more permissions than the set of permissions it has been granted. Being privileged simply tells `AccessController` to ignore the callers of the method containing the privileged code. For example, `AccessController` can stop checking after it has already verified that the privileged code has the required permission.

Moreover, a privileged block is specific to the thread that enabled its privileges. That is, the effect of code being privileged in one thread does not have any impact on other concurrently running threads, even though those other threads might be executing code that belongs to the same protection domain.

Another subtlety to consider is that the `doPrivileged` method can be invoked reflectively by using `java.lang.reflect.Method.invoke`. In this case, the privileges granted in privileged mode are not those of `Method.invoke` but those of the nonreflective code that invoked it. Otherwise, system privileges could erroneously or maliciously be conferred on user code.

Let us dig a little deeper into the proper and careful use of `doPrivileged`. In the password-changing application example, suppose that the code to open the password file is in another method, named `openPasswordFile`, which opens the password file and returns the object reference to the file input stream. The example code would become the following:

```
public void changePassword() {
   // Use own privilege to open the password file.
   AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         // Open file for reading and writing.
         f = openPasswordFile();
         return null;
      }
   });
   // Verify user's old and new passwords.
   ...
}
```

This code should operate exactly as before. Calling `doPrivileged` from inside `openPasswordFile` instead would be a mistake. Why? Because the user code could then call it directly, and because of the privilege inside that method, the user code would get a reference to the password file, as `openPasswordFile` returns such a reference. The lesson here is that a method, such as `openPasswordFile`, that returns a resource whose access should be controlled should not assert its privilege if it does not know or have full control over who can call it. On the other hand, `changePassword` may safely assert its own privilege, even if anyone can call it. The reason is that it takes care not to reveal the password file to the outside world and will process it internally only after password checking succeeds.

Note that the design of the privilege feature is asymmetrical. That is, code may choose to exercise its privileges and tell the access controller to ignore those callers before the asserting method, but it cannot tell the access controller to ignore those callers that are subsequently called. Thus, calling a method whose corresponding protection domain does not have a requisite permission will not escalate the privileges of the called method. This asymmetry is deliberate. If the access controller also ignored those classes and methods that were subsequently called, then effectively the caller would have granted its permissions to any code it invoked. The code may have control over which classes and methods are directly invoked but cannot be expected to know what will be invoked later. Privileges and granted permissions could be misused or abused if any of those callers were malicious or incompetent. It is a very bad idea to trust a series of unknown parties. The algorithm is designed to protect code from accidentally falling into such traps.

6.4.5 Privileged Actions Programming Idioms

The code example in the previous section demonstrates the simplest usage of `doPrivileged` by passing in a `PrivilegedAction` interface as the argument. That usage pattern, repeated next, is useful only when the code within the privileged block does not need to return a value.

```
somemethod() {
   ...normal code here...
```

```
   AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         // Privileged code goes here, for example:
         System.loadLibrary("awt");
         return null; // Nothing to return.
      }
   });
   ...normal code here...
}
```

As previously mentioned, `PrivilegedAction` is an interface with a single method, named `run`, that returns an `Object`. The example shows a concrete implementation of the `run` method being supplied. When the call to `doPrivileged` is made, an instance of the `PrivilegedAction` implementation is passed to it. The `doPrivileged` method calls the `run` method from the `PrivilegedAction` implementation after enabling privileges and then returns the method's return value as the `doPrivileged` return value, which is ignored in this example.

If the code from within the privileged block needs to return a value, the following is one way to write the code:

```
somemethod() {
   ...normal code here...
   String user = (String) AccessController.doPrivileged(
       new PrivilegedAction() {
           public Object run() {
               return System.getProperty("user.name");
           }
   });
   ...normal code here...
}
```

This usage requires a dynamic cast on the value returned by `doPrivileged`.

An alternative is to use a `final` local variable, as follows:

```
somemethod() {
   ...normal code here...
   final String user[] = {null};
   AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         user[0] = System.getProperty("user.name");
         return null; // Still need this.
      }
   });
   ...normal code here...
}
```

Because the local variable, `user`, cannot be declared in the body of the anonymous inner class, it must be declared `final` and must be assigned before the body of the inner class. But by using an array, we still can assign an element to the array and not violate language semantics.

A third solution is to write a nonanonymous class that handles typing information safely, such as the following:

```
somemethod() {
   ...normal code here...
   GetPropertyAction gpa = new
GetPropertyAction("user.name");
   AccessController.doPrivileged(gpa);
   String user = gpa.getValue();
   ...normal code here...
}

class GetPropertyAction implements PrivilegedAction {
   private String property;
   private String value;

   public GetPropertyAction(String prop) {
      property = prop;
   }

   public Object run() {
      value = System.getProperty(property);
      return value;
   }

   public String getValue() {
      return value;
   }
}
```

In this example, there is no type cast. Because the `run` method returns a value, you can abbreviate `somemethod` to the following:

```
somemethod() {
   ...normal code here...
   String user = (String) AccessController.doPrivileged(new
                        GetPropertyAction("user.name"));
   ...normal code here...
}
```

Finally, the interface `PrivilegedAction` is for privileged code that does not throw checked exceptions, such as `FileNotFoundException`. If the code can throw a checked exception—one that must be listed in the `throws` clause of a method—then the alternative form of the `doPrivileged` method, which takes a `PrivilegedExceptionAction` rather than a `PrivilegedAction`, should be used.

```
somemethod() throws FileNotFoundException {
   ...normal code here...

   try {
      FileInputStream fis = (FileInputStream)
         AccessController.doPrivileged(
            new PrivilegedExceptionAction() {
```

```
                public Object run() throws
FileNotFoundException {
                    return new FileInputStream("someFile");
            }
        });
    } catch (PrivilegedActionException e) {
        // e.getException() should be an instance of
        // FileNotFoundException, as only "checked" exceptions
        // will be "wrapped" in a PrivilegedActionException.
        throw (FileNotFoundException) e.getException();
    }
    ...normal code here...
}
```

The use of `doPrivileged` may seem cumbersome, but there is a reason for this. The rationale behind this design choice is discussed in Section 6.4.10. Meanwhile, remember that privileged operations should be implemented with great care because they assert the code's granted permissions even when acting on behalf of untrusted code. The privileged code block should be as small as possible, and all code that can be executed outside the block should not be inside the block.

6.4.6 The Inherited Access Control Context

As previously mentioned, the `AccessController getContext` method computes a snapshot of the current execution context, places it in an `AccessControlContext` object, and returns that object. To facilitate the generation of the context, the JVM keeps track of the `ProtectionDomain` created for each class executing on the call stack.

In the Java runtime, code can start any number of child threads, which can then start their own child threads, and so on. When a thread is created, the Java virtual machine creates a fresh execution stack but ensures that the current execution context is inherited by the child thread. In other words, the security context of the child thread includes the security context of all its ancestors. More specifically, the snapshot of the current execution context includes the current thread's inherited `AccessControlContext`.

Note that, strictly speaking, the Java virtual machine does not have to force a thread to recursively inherit its parent context, as not inheriting it does not necessarily pose a security problem. However, our experience shows that a typical programmer expects the security context to be inherited, and surprising the programmer is undesirable. Automatic inheritance is in fact quite convenient in some cases. In a server application, for example, a master thread might create slave threads to handle individual incoming requests when it would have been a burden to manually write the code for the slave threads to take into account the master's security context.

Another point to emphasize is that the inherited context is the exact context in the parent thread at the moment when the child thread is created. The inherited context is essentially frozen for further references, and the parent thread is free to continue and change its context afterward without impacting the content of the inherited context.

The access control algorithm that takes the inherited context into account is similar to the previously shown 'extended access control algorithm,' with the addition of a step calling the inherited context's `checkPermission` method to evaluate whether the inherited context's domains have the required permission:

```
i = m;
```

```
while (i > 0) {
   if (caller i's domain does not have the permission)
      throw AccessControlException;
   if (caller i is marked as privileged)
      return;
   i = i - 1;
}

// Next, check the context inherited when the thread was
created.
// Whenever a new thread is created, the
AccessControlContext
// at that time is stored and associated with the new
thread,
// as the "inherited" context.

inheritedContext.checkPermission(permission);

return;
```

6.4.7 The Privileged Access Control Context

Sometimes, code needs to ensure that a privileged operation is limited to a reduced set of permissions or is bounded by a security context that was collected earlier in the life cycle of the Java runtime. To support this, the following two `doPrivileged` methods accept an `AccessControlContext` as an argument:

```
public static native Object
    doPrivileged(PrivilegedAction action,
        AccessControlContext context)
public static native Object
    doPrivileged(PrivilegedExceptionAction action,
        AccessControlContext context)
        throws PrivilegedActionException
```

The only difference between them is that one takes a `PrivilegedAction` argument, whereas the other takes a `PrivilegedExceptionAction`, needed if the method might throw a checked exception.

Each of these methods marks the calling method's stack frame as privileged and associates the given `AccessControlContext` with the privileged frame. The context will be included in all future access control checks and will be checked after the privileged frame's `ProtectionDomain` is checked. Understanding the use of this method might be easier after you read the full access control algorithm, discussed next. Its use is illustrated by the following code, where `acc` is the `AccessControlContext` object:

```
somemethod() {
   ...normal code here...
   AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         // Code goes here. Any permission checks from this
         // point forward require both the current context
         // and the snapshot's context to have the desired
```

```
            // permission.
         }
      }, acc);
      ...normal code here...
}
```

6.4.8 The Full Access Control Algorithm

The full access control algorithm is the following: Suppose that the current thread traversed `m` callers, in the order of caller 1 to caller 2 through to caller `m`. Then caller `m` invoked the `checkPermission` method. The `checkPermission` method determines whether access is granted or denied, based on the following algorithm:

```
i = m;
while (i > 0) {
   if (caller i's domain does not have the permission)
      throw AccessControlException;
   else if (caller i is marked as privileged) {
      if (a context was specified in the call to
doPrivileged)
         context.checkPermission(permission);
      return;
   }
   i = i - 1;
}

// Next, check the context inherited when the thread was
created.
// Whenever a new thread is created, the
AccessControlContext
// at that time is stored and associated with the new
thread,
// as the "inherited" context.

inheritedContext.checkPermission(permission);

return;
```

The full algorithm is slightly more complicated than the inherited context algorithm given in Section 6.4.6. They differ in only one way. When a privileged frame is being checked and an access control context was specified in the call to `doPrivileged`, the security check will pass if, and only if, the requested permission is allowable in the caller's frame and in the specified access control context.

One other subtlety is worth noting. If the code never invokes `doPrivileged`, it is critical that the inherited context be evaluated for the requisite permission so as to ensure that untrusted code cannot lurk behind a child thread. For example, if unprivileged code could start a "system" thread without being included in access control checks triggered by the "system" thread, the system code might be compromised. It is a common programming idiom for trusted code not to exercise its privileges when performing a security-sensitive operation on behalf of its caller. The intent of the idiom is to ensure that the calling code has sufficient privileges to access security-sensitive resources.

Thus, a call to a `doPrivileged` method that includes an `AccessControlContext` argument can be used to enable a privileged frame but only for those permissions that would have been granted in the specified access control context. In other words, this feature can be used to further restrict the extent of the privilege coverage. Without a context being specified, a privileged frame may exercise all the permissions granted to it. With a context specified, the exercisable permissions are further limited to those that would have been permitted within the specified context.

From a theoretical and abstract level, the access control algorithm says that, at any point in a thread of computation, the effective permission is the intersection of the permissions of all protection domains traversed by the execution thread with the privilege status and its associated access control context, if any, as well as inherited access control context taken into account. Many strategies for implementing this algorithm are possible. The two most obvious are discussed here: eager evaluation and lazy evaluation.

In an *eager evaluation* implementation, whenever a thread enters a new protection domain or exits from an existing one, the set of effective permissions is updated dynamically. The benefit is that checking whether a permission is allowed is simplified and can be faster in many cases. The disadvantage is that because permission checking occurs much less often than cross-domain calls, a large percentage of permission updates might be useless effort.

In the *lazy evaluation* implementation, which is what Java 2 uses, whenever permission checking is requested, the thread state, as reflected by the current thread stack or its equivalent, is examined, and a decision is reached either to deny or to grant the particular access requested. One potential downside of this approach is the performance penalty at permission-checking time. However, this penalty would be incurred as well in the eager evaluation implementation, albeit at earlier times and spread out among each cross-domain call. In the Java 2 implementation, performance of this algorithm is quite acceptable, so we feel that lazy evaluation is the most economical approach overall.[1]

[1] For details of the implementation of protection domains prior to J2SE 1.4 and a discussion on performance and optimization techniques, see [47].

Note that because access control is based on the protection domains associated with the current execution context, the context must be preserved when optimizing code across class boundaries. In particular, a static, or just-in-time (JIT), compiler or a particular implementation of the Java virtual machine must exact precautions not to optimize so aggressively as to lose security context. For example, method inlining must be done with care so that protection domain information is not lost and the `AccessController` class can be correctly implemented such that its effective behavior does not differ between optimized and unoptimized code.

6.4.9 `SecurityManager` versus `AccessController`

Recall from earlier in this chapter the difference, when invoking a security check, between calling `checkPermission` and calling the other `check` methods defined in the `SecurityManager` class. The choice then was contingent on whether you depended on any pre–Java 2 security manager classes. Now you have another choice: calling either the `checkPermission` method defined in `SecurityManager` or the one defined in `AccessController`. These methods differ in two major ways.

First, sometimes no installed `SecurityManager` exists, so you cannot invoke `check` or `checkPermission` methods on it. By contrast, the static methods in `AccessController` are always available to be called. Recall the following idiom for calling `SecurityManager`:

```
SecurityManager sm = System.getSecurityManager();
if (sm != null)
   sm.checkPermission(permission);
```

But you can always call

```
AccessController.checkPermission(permission);
```

Thus, regardless of whether a systemwide `SecurityManager` has been installed, if you want to ensure that your security check is always invoked, you should call `AccessController`. Note, however, that some existing applications test whether there is an installed instance of `SecurityManager`. Then, based on the result of this test, which signifies one or the other security states, these applications take different actions. For the backward compatibility of these applications, calling `SecurityManager` is more appropriate.

The second difference is that calling `SecurityManager` does not guarantee a particular access control algorithm; someone might have extended it and installed a custom security manager. By contrast, calling `AccessController` guarantees that the full access control algorithm specified earlier is used. Thus, if you do not want to delegate your security check to a custom security manager, you should call `AccessController` directly. Otherwise, call `SecurityManager`.

Also, be warned that because the `SecurityManager` class defines a general interface for security checks, it does not provide the privilege mechanism that `AccessController` has defined. In fact, if you use the privilege mechanism in your code but later call `SecurityManager` to perform a security check, the privilege status might not be taken into account if the security manager you installed is not the one provided by Java 2 and does not consult `AccessController` or its equivalent.

You might wonder why we provide these choices. Isn't one way of doing things good enough? These choices are based on experience. A balanced trade-off between generality and consistency is needed. In the long run, we expect that custom security managers will not often be needed and that, even when they are defined, they will be built on existing functionality in `AccessController`. In particular, they will provide additional functionality rather than promote incompatible behavior. Nevertheless, in a special environment in which a vastly different sort of security policy must be enforced, a customized security manager conceivably might not be able to use the algorithms implemented by `AccessController`.

6.4.10 A Brief History of Privileged Operations

To cap the discussion on the `AccessController` class, we now provide more background on how the design of the `doPrivileged` methods developed. The main goals were to help programmers write secure code and to guarantee security when a programmer makes a mistake.

It is helpful to compare the desired result with UNIX's `setuid` facility. Compared with the MS-DOS and Windows operating systems, UNIX has traditionally given security somewhat more comprehensive consideration. It limits what a userinvoked program/application may do to a user's privileges. In some cases, though, these limits are too restricting. The `setuid` mechanism is designed to circumvent those limits. However, the entire `setuid`-ed program is "armed," in that any software bug in a part of the often large program can potentially lead to a security hole. We wanted to avoid this possibility in Java 2, so we created APIs that enable a programmer to limit to just a few method calls either the scope of the dangerous operations or the duration of the "armed" period. In this way, bugs outside those sensitive methods are less likely to cause unintended harm.

We considered several design proposals. One was to extend the language with a method modifier, perhaps called "privileged." Privilege would be granted when entering the method and revoked on returning from it. This was by far the cleanest design, but it would have required a major addition to the Java language, which in turn would have required compiler vendors to update their compilers. Such a change cannot be made lightly. Moreover, a method modifier cannot take a context argument. So we decided against it. We also rejected a number of other proposals that would have either changed the existing semantics of non-security-related code or required Java virtual machine support that would have been difficult to implement on all platforms.

Up to Java 2 SDK 1.2 Beta 3, we went with a design by which we provided the following two method calls in the `AccessController` class:

```
public static native void beginPrivileged()
public static native void endPrivileged()
```

Declaring a block of code to be privileged was to occur as follows:

```
somemethod() {
   (normal code here)
   try {
      AccessController.beginPrivileged();
      // Privileged code goes here, for example:
      System.loadLibrary("awt");
   } finally {
      AccessController.endPrivileged();
   }
   (more normal code here)
}
```

This design had the advantage of being fairly simple to use within the wellknown `try-finally` block construct. Its downside was that the call to `endPrivileged` could have been made only in the same method frame as the `beginPrivileged` call and optimally would have been called as soon as the privilege was no longer needed. This limited the privilege period to one method invocation and ensured that the privilege was reversed as soon as possible. If a programmer accidentally forgot to call `endPrivileged`, we built in a number of measures and checks to prevent mismatch between invocations of these `begin` and `end` methods from within different frames. For example, we would have reversed a privilege status if it was clear that the programmer should have reversed it but forgot to do so. In the end, the requirement to match frames was considered difficult to specify and enforce precisely in a platform-independent manner, so we abandoned that design in favor of the `doPrivileged` interface.

The design we eventually adopted works reasonably well, except for slightly added complexity in programming. We expect to improve the design later, for example, when suitable language constructs are made available.

Chapter 7. Customizing the Security Architecture

The office of government is not to confer happiness, but to give men opportunity to work out happiness for themselves.

—William Ellery Channing

This chapter demonstrates ways to augment the security architecture. We explain how to develop custom implementations of the various security classes that support either extensibility or substitution mechanisms. We also describe the mechanics of implementing a custom `Permission` class, extending the functionality of the `SecurityManager` class, implementing a custom `Policy` provider, and implementing a `DomainCombiner` interface.

7.1 Creating New Permission Types

Recall from Section 5.1 that J2SDK 1.2 introduced a new hierarchy of typed and parameterized access permissions, rooted by an abstract class, `java.security.Permission`. Other permissions are subclassed from either the `Permission` class or one of its subclasses and appear in relevant packages. For example, the `FilePermission` permission representing file system access is located in the `java.io` package. Other permission classes are

- `java.net.SocketPermission` for access to network resources.
- `java.lang.RuntimePermission` for access to runtime system resources, such as class loaders and threads
- `java.lang.PropertyPermission` for access to system properties
- `java.awt.AWTPermission` for access to windowing resources

As this list illustrates, accesses to controlled resources, including properties and packages, are represented by the permission classes.

Applications are free to add new categories of permissions. However, it is essential that, apart from official releases, no one extend the permissions that are built into the SDK, either by adding new functionality or by introducing additional keywords into a class such as `java.lang.RuntimePermission`. Refraining from doing this maintains compatibility.

When creating a new permission, it is advisable also to declare the permission to be `final`. The rule of thumb is that if the permission will be granted in a security policy, it is probably best to declare it `final`. However, at times it may be necessary to create a class hierarchy for your custom permission. If this is the case, a couple of design heuristics are worth mentioning. First, if the abstract, or base, class of your permission or permission collection has a concrete implementation of the `implies` method, it is recommended that the `implies` method take the type of the permissions into consideration. For example, the `implies` method of the `BasicPermission` class has the following logic, which is similar to that of the `BasicPermissionCollection` class:

```
public boolean implies(Permission permission) {
    if (! (permission instanceof BasicPermission))
            return false;
    BasicPermission bp = (BasicPermission) permission;
    if (bp.getClass() != this.getClass())
            return false;
    ...
```

```
}
```

Pay particular attention to the second `if` statement, which enforces the type equality heuristic. Without this, the implementation may be exposed to a subtle security hole whereby the subclass may be able to interact in malicious ways with the superclass's implication checking.

The second design heuristic addresses whether a custom `PermissionCollection` class should be implemented. The general principle is that if the permission has complex processing semantics for either its name or the actions it specifies, it is usually necessary to create a custom `PermissionCollection` class. The Java security architecture specifically enables this by first delegating to the permission collection object for processing of a requisite permission when making a policy decision.

Perhaps these guidelines are best shown by an example. Suppose that you are an application developer from company MyPVR and want to create a customized permission to control access to the channel programming features of a personal video recorder. Further suppose that only three actions are to be controlled for any given channel of programming: the ability to `"view"`, to `"preview"`, and to `"record"`. The first question is, can you use an existing `Permission` object, such as the `BasicPermission` class, or do you need a custom permission class? Given the need to be able to control access based on the three actions, the `BasicPermission` class will not suffice. Therefore, a custom class needs to be designed.

Next, you must make sure that the `implies` method, among other methods, is correctly implemented. If you decide to support more elaborate channel-naming syntax for `PVRPermissions`, such as 1–10:13–20 or *, you may need to implement a custom permission collection class, `PVRPermissionCollection`. This custom class would be responsible for parsing the names and ensuring the proper semantics. Additionally, it may be necessary to perform the `implies` logic entirely within the implementation supplied by the permission collection. An example of when this is necessary is when the actions of a permission can be granted separately but tested in combination. For example, you may have two separate grants of a `PVRPermission` specified by the security policy: one for `"view"` and one for `"preview"`, yet the resource management code tests for them in combination, perhaps as an optimization. That is, in order to be able to `"preview"` a channel, one must also have the permission to `"view"` the channel.

Here are parts of the code for sample `com.mypvr.PVRPermission` and `com.mypvr.PVRPermissionCollection` classes:

```
public final class com.mypvr.PVRPermission
                extends java.security.BasicPermission
                implements java.io.Serializable {

    /* view channel */
    private final static int VIEW       = 0x1;
    /* preview a channel */
    private final static int PREVIEW    = 0x2;
    /* record a channel */
    private final static int RECORD     = 0x4;
    /* all actions */
    private final static int ALL        =
VIEW|PREVIEW|RECORD;
    /* the channel number */
    private transient String channel;
```

```
    /* the actions mask */
    private transient int actionMask;
    public PVRPermission(String channel, String actions) {
        super(channel, actions);
        this.channel = channel;
        this.actionMask = getMask(actions) // parse actions
    }
    ...
    /* for completeness we implement implies but given the
usage
      pattern the real work will be done in the permission
collection
    */
    public boolean implies(Permission p) {
        if (!(p instanceof PVRPermission))
            return false;
        PVRPermission that = (PVRPermission) p;

        if (this.channel.equals(that.channel) &&
            this.actions.equals(that.actions))
            return true;
        return false;
    }

    public PermissionCollection newPermissionCollection() {
        return new PVRPermissionCollection();
    }
}

final class PVRPermissionCollection extends
PermissionCollection
        implements java.io.Serializable {
    private Vector permissions;

    public PVRPermissionCollection() {
        permissions = new Vector();
    }
    ...
    public boolean implies(Permission permission) {
        if (! (permission instanceof PVRPermission))
                return false;
        PVRPermission np = (PVRPermission) permission;
        int desired = np.getMask();
        int effective = 0;
        int needed = desired;
        Enumeration e = permissions.elements();
        while (e.hasMoreElements()) {
            PVRPermission x = (PVRPermission)
e.nextElement();
            if (x.channel.equals(np.channel)) {
              if ((needed & x.getMask()) != 0)  {
                effective |=  x.getMask();
                if ((effective & desired) == desired)
```

```
                    return true;
                needed = (desired ∧ effective);
              }
            }
        }
        return false;
    }
}
```

Next, you want the application's resource management code, when checking whether an access should be granted, to call `SecurityManager`'s `checkPermission` method, using an instance of `com.mypvr.PVRPermission` as the parameter

```
public void previewChannel(int channel) {
  // in order to preview the channel you also need to be
able
  // to view the channel
  com.mypvr.PVRPermission tvperm =
        new
com.mypvr.PVRPermission(Integer.toString(channel),
                                      "view,preview");
  SecurityManager security = System.getSecurityManager();
  if (security != null) {
    security.checkPermission(tvperm);
  }
...
}
```

Finally, to grant this permission to applications and applets, you need to enter appropriate entries into the security policy. How to configure the policy is discussed in detail in Section 12.5. Basically, you put the string representation of this permission in the policy file so that this permission can be automatically configured for each domain granted the permission. An example of the policy file entry specifying permission for the user `"Duke"` to watch channel 5 is as follows, which grants to any code considered to be executed by `"Duke"` the privilege to view and preview channel 5:

```
grant principal javax.security.auth.x500.X500Principal
"cn=Duke"{
   permission com.mypvr.PVRPermission "5", "view";
   permission com.mypvr.PVRPermission "5", "preview";
}
```

To exercise the built-in access control algorithm, our code would typically invoke a permission check by directly calling the `checkPermission` method of the `SecurityManager` class, as shown. Generally, it is best to start up the access control machinery by calling the `SecurityManager.checkPermission` method as demonstrated. The default implementation of `SecurityManager.checkPermission` delegates to the `AccessController`. However, should a custom `SecurityManager` class be installed, there is no guarantee that the `AccessController.checkPermission` method will ever be invoked. Details of these classes are provided in Chapter 6, and the question of when to use `AccessController` versus `SecurityManager` is discussed in Section 6.4.9.

7.2 Customizing Security Policy

The security policy is first processed by the `Policy` object and then is enforced by the `SecurityManager` or `AccessController`, so customizing any of these classes would customize the behavior of the security policy. Beginning with J2SE 1.4, security policy decisions are lazily evaluated; prior to J2SE 1.4, security policy decisions were in effect statically computed in advance of enforcement. This section provides general descriptions of the various ways the policy enforcement and decision machinery can be augmented to supply specialized behavior. We first describe the extension points of the `SecurityManager` and then give guidance in implementing a custom `Policy` provider.

7.2.1 Customizing Security Policy Enforcement

As a first example, suppose that you want to allow file access only during office hours: 9 A.M. to 5 P.M. That is, during office hours, the security policy decides who can access what files. Outside of office hours, no one can access any file, no matter what the security policy says. To achieve this, you can implement a `TimeOfDaySecurityManager` class, as follows:

```
public class TimeOfDaySecurityManager extends
SecurityManager {
   public void checkPermission(Permission perm) {
      if (perm instanceof FilePermission) {
         Date d = new Date();
         int i = d.getHours();
         if ((i >= 9) && (i < 17))
            super.checkPermission(perm);
         else
           throw new SecurityException("Outside of office
hours");
      } else super.checkPermission(perm);
   }
}
```

The `TimeOfDaySecurityManager checkPermission` method checks whether the permission to be checked is a `FilePermission`. If it is, `checkPermission` computes the current time. If the time is within office hours, `checkPermission` invokes the `checkPermission` method from `TimeOfDaySecurityManager`'s `SecurityManager` superclass to check the security policy. Otherwise, `checkPermission` throws a security exception. An application that wishes to enforce the given office hour restriction should install this `TimeOfDaySecurityManager` in place of the built-in `SecurityManager`. A `SecurityManager` is installed by calling the `java.lang.System.setSecurityManager` method, as described in Section 6.1.2.

The next example concerns the need to keep a record of resource accesses that were granted or denied, for audit purposes later. Suppose that you design a simple `AuditSecurityManager` class as follows:

```
public class AuditSecurityManager extends SecurityManager {

    private static java.util.logging.Logger logger =

java.util.logging.Logger.getLogger("AuditSecurityManager");
```

```
    public void checkPermission(Permission perm) {
        logger.log(java.util.logging.Level.INFO,
perm.toString());
        super.checkPermission(perm);
   }
}
```

This class assumes that you also have an instance of the `java.util.logging.Logger` class, whose `log` method records permission checks in a safe place. In this example, the `log` method simply records the fact that a particular permission was checked. A variation would be to enter the audit record after `checkPermission` and specify a different logging level, contingent on the result of the access control decision. If the `checkPermission` call succeeds, the `log` method is called right after the `super.checkPermission` call, with a level of `INFO`. If the call fails, a `SecurityException` is thrown, and the `AuditSecurityManager checkPermission` catches the `SecurityException`, calls the `log` method with a level of `WARNING` to indicate the failure, and then rethrows the exception, as follows:

```
public class AuditSecurityManager extends SecurityManager {
   . . .
   public void checkPermission(Permission perm) {
      try {
         super.checkPermission(perm);
         logger.log(java.util.logging.Level.INFO,
perm.toString());
      } catch (SecurityException e) {
         logger.log(java.util.logging.Level.WARNING,
            perm.toString());
         throw e;
      }
   }
}
```

To implement complex security policies, you potentially need to spend more effort in the design. For example, if you wanted to enforce a multilevel security policy, you would first have to create sensitivity labels for each object.[1] The JVM would also have to keep track of the interaction between objects and might have to change object labels dynamically, as in a High-Watermark model. Then the `SecurityManager`'s `checkPermission` method would need to base its decision on the labels of the objects involved in the current thread of execution. As another example, to implement a Chinese Wall, or separation-of-duty, model, the JVM would need not only to monitor object interaction but also to keep a history of it. Much research and experimentation is needed in this area.

[1] This could be done perhaps most conveniently by adding a security level attribute to the base class, the `Object` class, but that would be a very significant change.

7.2.2 Customizing Security Policy Decisions

The wide variety of reasons for designing and implementing a custom `Policy` provider run the gamut from the need to support a specialized policy expression language to the need to support a custom policy store. An example might be a `Policy` provider that specifies policy according to the syntax and processing rules of KeyNote [14]. We introduced the role and structure of the

`Policy` class in Section 5.4. In this section, we illustrate quintessential and sometimes subtle design details necessary to implement a `Policy` provider correctly.

Locating Security Policy

When we designed the Java security policy interface, we recognized that it would be nearly impossible to specify adequately the storage and access requirements of security policy or the language by which security policy was expressed. Therefore, we incorporated two key abstractions in our design. The first is the notion of a pluggable provider, which is a standardized mechanism fully documented in Section 12.3.3. The second, location independence of the policy store, is not part of the Java platform specification, yet most platform vendors have adopted Sun's model. Our approach to referencing the policy store is to leverage the inherent generality and location independence Uniform Resource Locators (URLs) provide.

Usually, the deployment of the Java runtime is managed by a system administrator. At deployment time, the administrator has the latitude of changing the configuration of the Java runtime to point to the location of the security policy data. This change can be statically reflected in the security properties file, or it can be specified when the runtime is started, by specifying a URL value for the `java.security.policy` system property. This is described in much greater detail in Section 12.5. For purposes of this discussion, the salient point is that it is imperative for a `Policy` provider implementation to follow the directions of the deployer when locating and accessing the policy data. That is, a proper implementation will follow the hints given by the values for the `policy.url.n` properties in the security properties file, as well as the mechanism to override or augment security policy via the system property `java.security.policy`.

The first thing to consider is whether the deployment permits the override of the security policy location via the `java.security.policy` system property. This is captured in the `policy.allowSystemProperty` security property:

```
if ("true".equalsIgnoreCase

(Security.getProperty("policy.allowSystemProperty"))) {
    // process the override URL.
    ...
} else {
    ...
}
```

Next, assuming that the location can be overridden by the `java.security.policy` system property, determine whether the supplied URL is to be used in conjunction with the statically configured URLs or whether it is to be considered as the sole source of policy data. The former is indicated by preceding the specified location with an `=` sign, whereas the latter is indicated by preceding the specified location with a double equals (`==`). For example, specifying the following system property as a subpart of the command line invoking the Java runtime indicates that the policy at the given URL should be the only source of policy data:

```
-
Djava.security.policy==https://policy.example.com/security.p
olicy
```

Bootstrapping Security Policy

When we designed the `Policy` provider architecture, one of our design goals was to enable the installation of third-party security policy implementations. We also wanted to give deployers flexibility in installing and configuring the Java runtime environment. Section 12.4 details configuring and installing provider packages. As described in Chapter 12, the security `Policy` provider class can be statically configured to be the default provider. One advantage of statically configuring the default security provider is to avoid having to implement application code that dynamically installs the `Policy` provider by invoking the `setPolicy` method of the `Policy` class.

Generally, system or infrastructure software of this caliber is installed as an optional package [101]. However, by installing the `Policy` provider as an installed extension, the implementation of the `Policy` provider is faced with a chicken-and-egg problem. And by statically configuring the systemwide security policy class, the Java runtime is confronted with a chicken-and-egg problem of its own. Let's consider the runtime's problem first.

The Java runtime must be able to install a security `Policy` provider that is not part of the system domain. However, code that is outside the system domain is subject to security checks by the `SecurityManager` or `AccessController`. Therefore, how can the runtime enforce policy while in the process of installing the `Policy` provider? It is possible for the Java runtime to detect this conundrum by being selective as to which class loader is used to install the `Policy` provider. Once this recursion is detected, it is possible to bootstrap the installation of the configured `Policy` provider by relying on the Java platform vendor's default `Policy` implementation class. Once the configured provider is loaded, it can be installed as the systemwide `Policy` provider. The deployer may need to realize that the platform vendor's default policy must be configured to grant the thirdparty `Policy` provider sufficient permissions to be bootstrapped in this manner. This approach is by no means foolproof, given that the third-party implementation may trigger class loads and subsequent security checks of utility classes outside the system domain or its own protection domain.

The second circularity problem exists regardless of whether the security Policy provider is configured statically or installed dynamically. Again because the `Policy` provider is not part of the system domain, it is subject to security checks. Because access control decisions are passing through the `Policy` implementation via either the `getPermissions` method or the `implies` method, the `Policy` class's protection domain will be under scrutiny whenever it triggers a security check.

A rather rudimentary solution a `Policy` implementation can use is to cache a reference to its own protection domain within its constructor. Then whenever its `getPermissions` or `implies` method is invoked, the `Policy` implementation can treat its protection domain specially and assume it to have sufficient permission to access the resource. In other words, it is probably fair to assume that the `Policy` implementation has been granted `AllPermission`. Here is some sample code from the `Policy` implementation class:

```
// Domain of this provider
private ProtectionDomain providerDomain;

public CustomPolicy() {
    ...
    final Object p = this;
    providerDomain = (ProtectionDomain)
        AccessController.doPrivileged(
            new java.security.PrivilegedAction() {
                public Object run() {
```

```
                    return
p.getClass().getProtectionDomain();
                }
    });
}
...

public boolean implies(ProtectionDomain pd, Permission p) {
    ...
    if (providerDomain == pd) {
        return true;
    }
    ...
}

public PermissionCollection getPermissions(ProtectionDomain
domain)
    {
    Permissions perms = new Permissions();
    ...
    if (providerDomain == domain) {
        perms.add(new java.security.AllPermission());
    }
    ...
}
```

Some words of caution: If the `Policy` implementation is packaged with less trusted code and both are in the same protection domain, the less trusted code will be accorded the same permissions as the `Policy` implementation. Also, a call to the `refresh` method should be careful not to drop the cached protection domain.

Spanning Permissions

In Section 5.6, we introduced the merits of dynamic policy and alluded to the spanning permission problem. We also described this issue in Section 7.1. Essentially, the issue can be stated as follows: A `Policy` provider must be able to accommodate a policy decision query through its `implies` interface to determine whether a requisite permission is implied by the given protection domain. This must be derived even when the actions of the requisite permission span the permission collection encapsulated by the `ProtectionDomain` and `Policy` objects.

In our sample `PVRPermissionCollection` implementation, we made special provisions for the advent of this when the actions of the requisite permission were specified in separate `grant` statements. The same problem exists for the `Policy` provider, as the class loader may assign permissions to the `ProtectionDomain` of a class, and the provisioning of the security policy may also specify permissions of the same type and target but for different actions. The simplest approach is for the `Policy` provider to merge the permissions from the protection domain with the permissions it deems are granted by the policy for the given `ProtectionDomain` into a single `PermissionCollection`. The obvious place to implement this merge is in the `getPermissions` method of the `Policy` class.

7.3 Customizing the Access Control Context

When we first introduced the `DomainCombiner` in Section 6.3, we described the API and the relationship of a `DomainCombiner` to the access control machinery. The remainder of this chapter describes possible uses for a `DomainCombiner` and implementation strategies. We may use the term combiner as a shorthand for `DomainCombiner`.

Two steps must be taken to insert a combiner into the access control machinery of the Java runtime environment. The first is to construct an `AccessControlContext` with an instance of a `DomainCombiner`. The second step is to bind the security context with the execution context. This is accomplished by supplying the `AccessControlContext` to the appropriate `AccessController doPrivileged` method.

A real-world application of a `DomainCombiner` is the `javax.security.auth.SubjectDomainCombiner`. This particular implementation encapsulates an instance of a `javax.security.auth.Subject`. As described in Section 8.4.1, a `Subject` encapsulates a set of (ostensibly authenticated) principals. The role of the `SubjectDomainCombiner` is to augment the `ProtectionDomain`s of the current execution context with the principal information so that security policy can be based on who is running the code.

The `combine` method of a bound `DomainCombiner` is invoked as a result of a call to either the `checkPermission` or the `getContext AccessController` method. In the absence of a `DomainCombiner`, the `AccessController` optimizes the `AccessControlContext` object. When a `DomainCombiner` is present, it is up to the implementation of the installed combiner to perform any optimizations on the `AccessControlContext` returned from the `combine` method. That said, another possible application for a `DomainCombiner` is one that implements special optimizations. For example, suppose that the `DomainCombiner` encapsulated principal information analogous to the `SubjectDomainCombiner`. Additionally, the custom combiner makes special provisions for the administrative principal such that it is accorded `AllPermission`. Such a combiner can make a significant optimization to the `AccessControlContext` when it detects that it is executing as the administrative principal. One possibility is for the combiner to return an `AccessControlContext` with a single `ProtectionDomain`; in that case, the `PermissionCollection` encapsulated within the `ProtectionDomain` would include an instance of the `AllPermission` permission.

Chapter 8. Establishing Trust

Love all, trust a few, do wrong to none.

—William Shakespeare

The ability to establish trust is an essential ingredient when building reliable systems. A cornerstone of establishing a trust relationship is entity authentication, the process by which one or more authenticating parties determine the identity of a claimant party such that they have a reasonably strong assurance that the evidence presented proves the claimant's identity. Succinctly, an authenticated identity supplies assurance that the claimant is genuine. Once the identity of the entity has been determined, access control decisions can be made based on the perceived trustworthiness of the parties involved.

Entity authentication is accomplished with a variety of means. Generally speaking, it can be reduced to proving at least one of three basic factors: *something known*, *something possessed*, or *something inherent* in the claiming entity, commonly a biometric, such as a fingerprint. A system may derive different degrees of trust, depending on which or how many factors were corroborated during authentication. For example, an entity that authenticates with a reusable password may be restricted to have only read access to certain files. However, an entity that authenticated with two factors—the possession of a debit card and the correct PIN—may be permitted to withdraw funds from an automatic teller machine.

The Java 2 platform provides a robust infrastructure and rich set of APIs that, when applied, bolster application security. This chapter describes the mechanisms endemic to trust establishment within the context of the Java 2 platform. We begin by describing the foundation on which we establish trust in mobile code. The Java platform introduced the notion of code signing with the release of JDK 1.1. Java 2 relies on digital signatures and public-key certificates to determine the trust in a given piece of code. Code signing as a means of trust establishment is described in Section 8.3.

Section 8.1 provides a brief overview of *digital certificates* and their application in Java 2. The Java Certification Path (CertPath) API is covered in Section 8.2. CertPath specifically caters to application developers who need to implement customized trust management solutions based on PKI (public key infrastructure). Section 8.3 explains how digital certificates are used to sign code and includes a detailed discussion of signed JAR files and how signed code can affect access control decisions. Section 8.4 covers how user-centric access control can be accomplished with the Java Authentication and Authorization Service (JAAS). Section 8.5 explains how an application that requires peer-to-peer authentication can use the authentication and credential delegation capabilities of the Java Generic Security Services API (Java GSS-API) and the trust management capabilities of the Java Secure Socket Extension (JSSE).

Note: the Java Certification Path API and the Java GSS-API were added to the platform in Java 2 SDK, Standard Edition (J2SDK), 1.4. JAAS and JSSE were introduced as optional packages—extensions—to earlier releases of the J2SDK and subsequently integrated into J2SDK 1.4.

8.1 Digital Certificates

Users of public-key applications and systems must be confident that the public key of a subject—a user, organization, or other entity, such as a service—is genuine, that is, that the associated private key is owned by the subject. Public-key certificates are used to establish trust. A *public-key certificate* is a binding of a public key to a subject, whereby the certificate is digitally signed by the private key of another entity, often called a *Certification Authority (CA)*.

If the user does not have a trusted copy of the public key corresponding to the private key the CA used to sign the subject's public-key certificate, another public-key certificate vouching for the signing CA is required. This logic can be applied recursively, until a chain of certificates, or a *certification path*, is discovered from a *trust anchor*—often called the *root certificate*, or a *most-trusted CA*—to the target subject, commonly referred to as the end entity. See Section 8.2 for more information on certification paths.

To facilitate interoperability, the international body that sets communication standards—International Telecommunication Union (ITU), formerly Comité Consultatif International Téléphonique et Télégraphique—created a standard digital certificate format, ITU-T X.509, or ISO/IEC/ITU 9594-8. X.509 version 1 [20], or X.509 v1, was first published as part of the X.500 directory recommendations. X.500 was intended to define a global, distributed database of named entities.

An X.509 certificate binds a public key to a Distinguished Name (described in Section 8.1.1). Frequently, such a certificate is called an identity certificate. In the context of an X.500 directory, the bound public key is used to authenticate an entity attempting to modify a directory entry.

Other applications for X.509 certificates were developed outside the scope of the X.500 directory. For example, the initial specification for Privacy Enhancement for Internet Electronic Mail (PEM) [62] used X.509 version 1 certificates to ensure authenticity of message originators and recipients. Deployment of PEM proved difficult, and additions to the X.509 version 2 specification were used in a revision to the PEM specification [60].

Version 3, the most widely used version of the X.509 certificate format, is especially popular in Web browsers, such as Netscape Navigator and Internet Explorer, that support the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols.

8.1.1 X.500 Distinguished Names

X.500 Distinguished Names are used to identify entities for the subject and issuer (signer) fields of X.509 certificates. Following is a sample X.500 DN string:

```
"CN=Duke, OU=Java Software, O=Sun, L=Santa Clara, S=CA,
C=US"
```

The keywords `CN`, `OU`, and so on, are abbreviations:

- `CN` = Common name
- `OU` = Organization unit
- `O` = Organization name
- `L` = Locality name (indicating a city)
- `S` = State name (indicating a state or province)
- `C` = Country (expected to be a two-letter country code)

For more information about X.500 Distinguished Names, see [130].

8.1.2 X.509 Certificate Versions

Four versions of X.509 are available:

1. X.509 v1, available since 1988, is limited in capabilities and has been supplanted by X.509 v3.

2. X.509 v2 introduced the concept of subject and issuer unique identifiers to handle the possibility of reuse of subject and/or issuer names. Most certificate profile documents strongly recommend that names not be reused and that certificates not make use of unique identifiers. Version 2 certificates are not widely used.
3. X.509 v3 supports the notion of *extensions*. Anyone may define an extension and include it in the certificate. Some commonly used extensions are
    a. `KeyUsage`, which limits the use of the keys to particular purposes, such as signing only
    b. `SubjectAltName and IssuerAltName`, which allow other identities also to be associated with this public key—for example, DNS names, e-mail addresses, and IP addresses
    c. `BasicConstraints`, which identify whether the subject of the certificate is a CA and how deep a certification path may exist through that CA

    An extension can be marked "critical" to indicate that it should be checked and enforced/used. For example, if a certificate has the `KeyUsage` extension marked critical and only `digitalSignature` is asserted, the certificate must be used only in conjunction with digital signature mechanisms, such as entity authentication, for example, SSL/TLS communication.

4. X.509 v4 mainly adds support for attribute certificates. Unlike public-key certificates, which bind a name to a public key, attribute certificates bind one or more attributes to a name.

8.1.3 X.509 Certificate Contents

All X.509 certificates have the following data:

- **Version.** The X.509 version that applies to this certificate. The version affects what information can be specified in a certificate. Thus far, four versions are standardized.
- **Serial number.** A unique number, assigned by the entity that created the certificate so as to distinguish it from other certificates it issues. This information is used in numerous ways; for example, when a certificate is revoked, its serial number is placed in a *Certificate Revocation List* (CRL), a time-stamped list identifying revoked certificates. A CRL is signed by a CA and often made freely available in a public repository.
- **Signature algorithm identifier.** The algorithm used by the CA to sign the certificate, such as MD5RSA.
- **Issuer.** The X.500 name of the entity that signed the certificate. The entity is normally a CA.
- **Validity period.** The time period for which the certificate is valid. Each certificate is valid for only a limited amount of time. This period is described by a start date and time and an end date and time. This is the expected period during which the CA warrants that it will maintain information about the status of the certificate.
- **Subject name.** The name of the entity whose public key the certificate identifies. This name uses the X.500 standard, so it is intended to be unique across the Internet. It is the Distinguished Name of the entity.
- **Subject public-key information.** The public key of the entity being named (the subject), together with an algorithm identifier that specifies to which public-key cryptosystem this key belongs and any key parameters used by the algorithm.
- **Signature.** The signature is over the preceding fields, effectively binding the subject public key to the subject name. The signature uses the algorithm specified by the signature algorithm identifier.

All the data in a certificate is encoded using two related standards: Abstract Syntax Notation 1 (ASN.1) and Distinguished Encoding Rules (DER). ASN.1 describes data; DER describe a single way to store and transfer that data.

Certificates are often stored using the *Base64 encoding* [75]. Base64 encoding facilitates exporting certificates to other applications (for example, via e-mail). In its Base64 encoding, the encoded certificate is bounded at the beginning and the end, respectively, by

```
-----BEGIN CERTIFICATE-----
```

and

```
-----END CERTIFICATE-----
```

8.1.4 Obtaining Certificates

Certificates are available in a number of ways. A self-signed certificate can be generated by using the right tools, such as the `keytool` utility, which is provided with J2SDK and described in Section 12.8. A self-signed certificate is one signed using the private key corresponding to the public key in the certificate. However, some people will accept only certificates signed by a trusted CA. The value a CA provides is that of a neutral and trusted introduction service, based in part on its verification requirements, which are openly published in its Certification Practice Statement (CPS) [22].

A certificate can be requested from a CA. The `keytool` utility can assist in generating the request, called a Certificate Signing Request (CSR) [109]. Basically, to obtain a certificate from a CA, a matched pair of public and private keys are needed. At a minimum, the CA will need the public key and information about the entity being certified, such as the name and address for a person whose public key is to be certified. Normally, the CA will require the requester to provide proof as to the accuracy of this information. The information is submitted in a self-signed certificate so that the CA can verify its integrity. That is, the CA can verify that the private key used to sign the request corresponds to the public key in the request.

8.2 Establishing Trust with Certification Paths

As introduced in Section 8.1, a *certification path* is a chain of certificates from a trust anchor to the target subject, or *end entity*. After the first, each certificate in the chain authenticates the public key of the signer of the previous certificate in the chain. The trust anchor is usually specified by a certificate issued to a CA that the user relies on as a *Trusted Third Party* (TTP). Use of such a certificate implies that one trusts the entity that signed the certificate.

In general, a certification path is an ordered list of certificates, usually comprising the end entity's public-key certificate and zero or more additional certificates. A certification path typically has one or more encodings, allowing it to be safely transmitted across networks and to different operating system architectures.

Figure 8.1 illustrates a certification path from the public key of a trust anchor (CA1) to the target subject (Alice). The certification path establishes trust in Alice's public key through an intermediate CA named CA2.

Figure 8.1. Certification path

A certification path must be validated before it can be relied on to establish trust in a subject's public key. Validation can consist of various checks on the certificates contained in the certification path, such as verifying the signatures and checking that each certificate has not been revoked. The PKIX (Public Key Infrastructure) standards define an algorithm for validating certification paths consisting of X.509 certificates.

Often a user may not have a certification path from a trust anchor to the subject. Providing services to build or discover certification paths is an important feature of public-key-enabled systems. RFC (Request for Comments) 2587 [16] defines an LDAP schema definition that facilitates the discovery of X.509 certification paths using the LDAP directory service protocol.

Building and validating certification paths is an important part of many standard security protocols, such as SSL/TLS, S/MIME (Secure Multi-Purpose Internet Mail Extensions), and IPSec (Internet Protocol Security). The Java Certification Path (CertPath) API, added to the Java 2 platform 1.4, provides a set of classes and interfaces for developers who need to integrate this functionality into their applications. This API benefits two types of developers: those who need to write service provider implementations for specific certification path building or validation algorithms and those who need to access standard algorithms for creating, building, and validating certification paths in an implementation-independent manner.

The Java Certification Path API consists of interfaces and classes that support certification path functionality in an algorithm- and implementation-independent manner. The API also includes a set of algorithm-specific classes for the PKIX [54] standards. (See Section 8.2.6.) The API builds on and extends the previously existing J2SDK `java.security.cert` package for handling certificates. In the next subsection, we briefly describe the fundamental classes and interfaces that make up this API. The classes can be divided into five categories: core, basic, validation, building, and storage.

8.2.1 Core Certificate API

Java 2 contains a rich set of APIs for accessing and managing certificates. The certificate API includes the following classes:

- `Certificate`, an abstract class for managing a variety of certificates. This class is an abstraction for certificates that have different formats but important common uses. For example, different types of certificates, such as X.509 and PGP (Pretty Good Privacy) [136], share general certificate functionality, such as encoding and verifying, and some types of information, such as a public key. X.509, PGP, and SDSI (Simple Distributed Security Infrastructure) [103] certificates can all be implemented by subclassing the `Certificate` class, even though they contain different sets of information and store and retrieve the information in different ways.
- `CRL`, an abstraction of Certificate Revocation Lists (CRLs) that have different formats but important common uses. For example, all CRLs share the functionality of listing revoked certificates and can be queried on whether they list a given certificate.

- `CertificateFactory`, which defines the functionality of a certificate factory, used to generate certificate, certification path (CertPath), and Certificate Revocation List objects from their encodings.
- `X509Certificate`, an abstract class for X.509 certificates. This class provides a standard way to access all the attributes of an X.509 certificate.
- `X509Extension`, an interface for X.509 extensions. The extensions defined for X.509 v3 certificates and v2 Certificate Revocation Lists provide methods for associating additional attributes with users or public keys, for managing the certification hierarchy, and for managing CRL distribution. The X.509 extensions format also allows communities to define private extensions to carry information unique to those communities.
- `X509CRL`, an abstract class for an X.509 CRL.
- `X509CRLEntry`, an abstract class for a revoked certificate in a CRL.

In Section 8.1.3, we described the common contents of X.509 certificates. The following example demonstrates how to access the various attributes of an X.509 certificate:

```
FileInputStream fis = new FileInputStream(filename);
CertificateFactory cf =
CertificateFactory.getInstance("X.509");
X509Certificate cert =
    (X509Certificate)cf.generateCertificate(fis);
System.out.println("X.509 Certificate");
System.out.println("Version: " + cert.getVersion());
System.out.println("Serial number: " +
    cert.getSerialNumber().toString(16));
System.out.println("Signnature algorithm: " +
    cert.getSigAlgName());
System.out.println("Issuer name: " +
cert.getIssuerDN().getName());
System.out.println("Valid from: " + cert.getNotBefore());
System.out.println("Valid to: " + cert.getNotAfter());
System.out.println("Subject name: " +
    cert.getSubjectDN().getName());
System.out.println("Subject public key algorithm: " +
    cert.getPublicKey().getAlgorithm());
```

8.2.2 Basic Certification Path Classes

The basic certification path classes provide fundamental functionality for encoding and representing certification paths. The key class in the Java Certification Path API is `CertPath`, which encapsulates the universal aspects shared by all types of certification paths. An application uses an instance of the `CertificateFactory` class to create a `CertPath` object. The following example parses a PKCS (Public-Key Cryptography Standards) #7 [107] formatted certificate reply stored in a file and extracts all the certificates from it. (A certificate reply is what a CA sends in response to a certificate signing request.)

```
FileInputStream fis = new FileInputStream(filename);
CertificateFactory cf =
CertificateFactory.getInstance("X.509");
CertPath cp = cf.generateCertPath(fis);
```

8.2.3 Certification Path Validation Classes

The Java Certification Path API includes classes and interfaces for validating certification paths. An application uses an instance of the `CertPathValidator` engine class (see Section 10.2) to validate the certificates in a `CertPath` object. If successful, the result of the validation algorithm is returned in an object implementing the `CertPathValidatorResult` interface. Following is a simplified code sample that illustrates how to create a `CertPathValidator` that implements the `"PKIX"` algorithm and how to use it to validate the certification path in the `CertPath cp`:

```
CertPathValidator cpv =
CertPathValidator.getInstance("PKIX");
try {
    CertPathValidatorResult cpvResult = cpv.validate(cp,
params);
} catch (CertPathValidatorException cpve) {
    System.err.println("validation failed: " + cpve);
} catch (InvalidAlgorithmParameterException iape) {
    System.err.println("invalid parameter: " + iape);
}
```

Note that the `CertPath` and `CertPathParameters` passed to the `validate` method must each be of a type that is supported by the validation algorithm. Otherwise, a `java.security.InvalidAlgorithmParameterException` is thrown. For example, a `CertPathValidator` instance that implements the `"PKIX"` algorithm validates `CertPath` objects of type `"X.509"` and `CertPathParameters` that are instances of `PKIXParameters`. Typically, the `CertPathParameters` supply a set of trust anchors as instances of `TrustAnchor` classes. The information in a `TrustAnchor` includes the CA's Distinguished Name and public key and any constraints on the set of paths that may be validated using this key.

8.2.4 Certification Path Building Classes

The Java Certification Path API includes classes for building, or discovering, certification paths. An application uses an instance of the `CertPathBuilder` engine class (see Section 10.2) to build a `CertPath` object. If successful, the result of the build is returned in an object implementing the `CertPathBuilderResult` interface. The following code sample illustrates how to create a `CertPathBuilder` that implements the `"PKIX"` algorithm and how to use it to generate a certification path:

```
CertPathBuilder cpb = CertPathBuilder.getInstance("PKIX");
try {
    CertPathBuilderResult cpbResult = cpb.build(params);
    CertPath cp = cpbResult.getCertPath();
} catch (CertPathBuilderException cpbe) {
    System.err.println("build failed: " + cpbe);
} catch (InvalidAlgorithmParameterException iape) {
    System.err.println("invalid parameter: " + iape);
}
```

8.2.5 Certificate/CRL Storage Classes

The Java Certification Path API also includes the `CertStore` engine class (see Section 10.2) for retrieving certificates and CRLs from a repository. This class is useful because it allows a caller to specify the repository a `CertPathValidator` or `CertPathBuilder` implementation should use to find certificates and CRLs.

A `CertPathValidator` implementation may use the `CertStore` object that the caller specifies as a callback mechanism to fetch CRLs for performing revocation checks. Similarly, a `CertPathBuilder` may use the `CertStore` as a callback mechanism to fetch certificates and, if performing revocation checks, CRLs.

Unlike the `java.security.KeyStore` class, which provides access to a cache of private keys and trusted certificates, a `CertStore` is designed to provide access to a potentially vast repository of untrusted certificates and CRLs. For example, one could implement a `CertStore` that provides access to certificates and CRLs stored in one or more directories using LDAP. In fact, J2SE adds the convenience class `LDAPCertStoreParameters` to simplify the integration to an LDAP repository.

8.2.6 PKIX Classes

The Java Certification Path API also includes a set of algorithm-specific classes modeled for use with the PKIX certification path validation algorithm as defined in RFC 3280. Specifically, J2SE adds parameter convenience classes `PKIXParameters` and `PKIXBuilderParameters` to supply parameters for the `CertPathValidator` and `CertPathBuilder` algorithms, respectively. Specialized result classes `PKIXCertPathValidatorResult` and `PKIXCertPathBuilderResult` return the results of their respective PKIX certification path algorithm. Finally, J2SE defines the abstract `PKIXCertPathChecker` class, which can be extended to customize the PKIX certification path validation algorithm.

The following example is a specialized implementation of the path validation example from Section 8.2.3:

```
CertPathValidator cpv =
CertPathValidator.getInstance("PKIX");
KeyStore tks = KeyStore.getInstance("JKS");
tks.load(new FileInputStream("./keystore"),
"Three may keep a secret if two of them are
dead".toCharArray());
TrustAnchor anchor = new TrustAnchor((X509Certificate)
                         tks.getCertificate("ca"),null);
PKIXParameters params =
  new PKIXParameters(Collections.singleton(anchor));
params.setRevocationEnabled(false);
try {
  PKIXCertPathValidatorResult result =
    (PKIXCertPathValidatorResult) cpv.validate(cp, params);
} catch (CertPathValidatorException cpve) {
  System.err.println("validation failed: " + cpve);
} catch (InvalidAlgorithmParameterException iape) {
  System.err.println("invalid parameter: " + iape);
}
```

8.3 Establishing Trust in Signed Code

By building on the mechanisms described in the previous section, we now have the ability to establish trust in code. In Section 2.4, we described code signing, which was introduced with the release of JDK 1.1. Code signing provides the means to make trust decisions predicated on a trust relationship with the signer of the code. The code-signing model in JDK 1.1 effectively granted code that was signed by one or more trusted parties the permission to execute unencumbered. In other words, the code was not placed into the sandbox. In Java 2, we can be much more flexible and give only the minimum set of permissions necessary to accomplish the task. This is another example of how Java 2 subscribes to the principle of least privilege [111].

The remainder of this section describes the intricacies of code signing. First, we provide a brief overview describing the application of digital signatures to ensure the integrity of data. We cover the germane aspects of the JAR file format [121]. Then we tell how the integrity-protection mechanisms of the JAR file are relied on to ensure that the contents have not been modified either maliciously or otherwise. Finally, we describe how and when the Java runtime environment makes trust decisions based on the signers of the code with respect to the installed security policy.

8.3.1 Securing Messages with Digital Signatures

Digital signatures are a common technique used to ensure the integrity of a message. For the purposes of this discussion, we will describe only digital signatures that use the basic security primitives: public-key cryptography and one-way hash functions. Aside from the data-integrity property a digital signature provides, a digital signature may also be used for authentication. By combining these primitives, we can prove that a message has arrived intact. Also, one can be reasonably certain which key was used to sign the message. Thus, one can make a trust decision based on this knowledge and the knowledge that only authorized entities possess the signing key.

In general, digital signatures work as follows: The signer of a message computes a one-way hash of the message. The result of this computation, the *digest*, is then signed, that is, encrypted with the signer's private key. Thus, the digest is used to ensure the integrity of the message, and the signature protects the integrity of the digest. Refer to Section 10.6.3 for examples of using the `java.security.Signature` class.

To validate that the message has not been tampered with, the verifier computes the hash of the message, decrypts the signature with the signer's public key, and compares the two digest values. If they are the same, the verifier has proof that the message arrived unscathed. Also note that the verifier relies on the fact that only the private key that corresponds to the public key used to decrypt the signature could have been used to correctly encrypt the digest. This proves that the private key was used to sign the message. Possession of the private key is a primary factor used to authenticate a system entity. Therefore, if the public key is bound to a public-key certificate, the verifier can determine the Distinguished Name, or identity, of the signer. However, the presence of a public-key certificate is not sufficient evidence to trust the contents of the certificate. To accomplish this task, the certification path should be traversed to a trusted anchor (Section 8.2).

8.3.2 JAR File Format Overview

The JAR file format builds on the cross-platform ZIP archive format, defining a standard structure and archive entries that describe the contents of the archive, the integrity controls, and additional information necessary to describe the signatures and signers. This information is used to verify the integrity of the contents and subsequently to make trust decisions.

The remainder of this section describes the security elements of the JAR file format, their usage, and the relationships relevant to establishing trust in the code the JAR contains.

- **`META-INF` directory.** The `META-INF` directory contains all the ancillary control files used to describe the contents of the archive, including the information used to protect the integrity of the JAR contents. The `META-INF` directory is the parent to all the files that follow.
- **Manifest file.** The JAR archive contains a single manifest file, named `MANIFEST.MF`, which consists of sections that are entries for various files in the archive. Sections are separated from one another by empty lines. Not all files in the archive need to be listed in the manifest, but all files that are to be signed must be listed. Each manifest entry consists of attribute information about a file contained in the archive. The syntax of this information is modeled after RFC 822 [25] "name: value" pairs. In fact, all the control files in the JAR archive use this structural convention except for the digital signature files themselves. For each file that is signed, there is, at a minimum, a name entry whose value specifies a relative file path, or URL, and an attribute whose name specifies the digest algorithm used and whose value contains the Base64-encoded representation of the digest of the named file. Here is a sample section for the `DHPublicKey.class` file:
- `Name: com/sun/crypto/provider/DHPublicKey.class`
- `SHA1-Digest: A5o8kN0r/eqp2QiJDGLCFQdIRC4=`
- **Signature (instructions) file.** Whenever a JAR file is signed, a signature instructions file, sometimes referred to as a signature file, and a corresponding signature block file are created. There can be multiple signature instructions files in the archive, one for each signer. Each signature instructions file ends with a .SF file extension. This file looks similar to the manifest file, except that the digests in this file are calculated from the manifest file entries rather than from the contents of the specified files. That is, each individual entry must name a file and then specify the digest algorithm used and the digest of the section in the manifest file for the named file. The following sample entry specifies the SHA-1 digest of the section in the manifest file for `DHPublicKey.class`:
- `Name: com/sun/crypto/provider/DHPublicKey.class`
- `SHA1-Digest: fYx7UiXMdD2WWcgaYhLimHmoRbM=`
- **Signature block file.** A signature block file has the same file name as the corresponding signature instructions file but with a different extension. The extension varies, depending on the type of digital signature algorithm used. The format of the contents is specific to the signature algorithm. For example, both the DSA and RSA signature algorithms conform to the syntax and encoding specified by PKCS #7 [107].

Given the preceding description of the layout and semantics of a signed JAR, some important characteristics can be noted. The integrity of the files contained in the JAR archive is protected by the hashes, or digests, of the files. The digest for a file is stored in a named section of the manifest. This digest of the file and all the other attributes specified in its section of the manifest are protected by the digest within the signature instructions file of the named file's section in the manifest. Finally, the signature instructions file is protected by the signature block file. A noteworthy quality of this design is that it protects not only the contents of the JAR file but also any attributes specified in a named section of the manifest.

8.3.3 Runtime Trust Establishment

A Java runtime environment may defer initiating the verification of the JAR archive contents until the manifest must be parsed. At that time, only the signatures in the signature instructions files are verified, not the files in the archive. For efficiency, this verification can be remembered.

To validate a JAR file fully, a digest value in the signature instructions file is first compared with a digest calculated against the corresponding named entry in the manifest file. (This step may have been performed earlier, as described in the previous paragraph.) Then a digest value in the manifest file is compared to a digest calculated against the data referenced in the `"Name:"` attribute, which specifies either a relative file path or a URL.

During class loading, if the class coming from the JAR file verifies correctly, the defining class loader will set the signers of the class accordingly. The signers of the class will be taken into consideration by the security policy whenever the class is involved in a security check.

The security subsystem of the Java runtime environment requires consistency in what keys are used to sign classes within a JAR file. One very important restriction is the same-package, same-signers requirement. If any class file that belongs to a package in a JAR file is signed, all class files belonging to the same package must be signed by the same signers. JAR files may still contain unsigned packages. However, if any package contains signed classes, all class files of that package must be signed by the same signers. This constraint is enforced to thwart what is referred to as a *package-insertion attack*.

8.4 User-Centric Authentication and Authorization Using JAAS

When Java technology is used to construct a full-fledged distributed system, a whole new range of distributed systems security issues, such as those mentioned in Chapter 1, must be addressed. For example, additional mechanisms are needed to make Remote Method Invocation (RMI) secure in the presence of hostile network attacks. Jini Network Technology takes full advantage of RMI. Jini Network Technology enables digital devices to simply connect together, so service registration and location must be securely managed if the environment contains coexisting but potentially hostile parties. Thus, a full set of higher-level services must be secured, such as transactions for e-commerce. In addition, many lower-level security protocols can be leveraged, such as the network security protocols Kerberos [65] and IPSec [61]. This playing field is too large to speculate about in this short section, but a critical foundation for all these issues is a facility to authenticate users and to use this information to make access control decisions.

Java 2 relies on security policy to grant access permissions to running code. In the initial Java 2 releases, the decision depended on the characteristics of where the code was coming from and whether it was digitally signed and by whom. A *code-centric* style of access control is unusual. Traditional security measures, most commonly found in sophisticated operating systems, are *user-centric*, in that they apply control on the basis of who is running an application, not on which application is running. Code-centric access control was justified largely because as a user surfing the Web encounters executable content—for example, mobile code written in the Java language—the user essentially retains a constant identity. On the other hand, the user might trust one piece of mobile code more than others and would like to run this code with more privileges. Thus, it is natural to control the security of mobile code in a code-centric style.

Nevertheless, Java technology is widely used in multiuser environments, such as public Internet kiosks, enterprise payroll and calendar applications, and servers handling e-commerce transactions from numerous trading partners. All these examples must deal with different users, either concurrently or sequentially, and must grant them different privileges based on their identities.

The Java Authentication and Authorization Service (JAAS) was designed to provide a standard programming interface for authenticating users and for assigning privileges. JAAS was introduced as an optional package, or extension, to J2SE 1.3, and was subsequently integrated into J2SE 1.4. In these releases, an application can provide code-centric access control, user-centric access control, or a combination of both. JAAS also lays the groundwork to support a general mechanism for cross-protection domain authorization and the "running-on-behalf-of" style of delegation.

Authentication has been a topic of security research for decades. However, the Java runtime environment presents unique challenges. The design of JAAS was motivated by the following requirements:

- **Extensibility.** A need existed for a small but well-grounded set of Java application programming interfaces for authentication and authorization that can easily be extended.
- **Pluggability.** Different systems can easily incorporate their new or existing authentication capabilities into the JAAS framework.
- **Compatibility.** The initial code-based access control architecture, introduced in Java 2, and the user-based access control mechanism in JAAS could coexist independently and could also be seamlessly combined to implement sophisticated security policies.

Several existing standards support authentication, including the Generic Security Services Application Programmer's Interface (GSS-API) [77] and Simple Authentication and Security Layer Application Programmer's Interface (SASL) [89]. SASL represents a framework that provides authentication support for connection-based protocols, thereby catering to applications that perform network authentication. Like JAAS, SASL also has a modular architecture. GSS mechanisms, such as Kerberos [76] or the Simple Public Key Mechanism (SPKM) [1], may be plugged in under the SASL framework. JAAS, on the other hand, also supports local login. Thus, JAAS and SASL/GSS complement each other to provide both local and network-based support for authentication.

One scenario in which these architectures might coexist involves environments that rely on Kerberos (and possibly other services) for authentication. JAAS login modules could be plugged in under the login application to authenticate the user, when initially logging in, to both the underlying operating system and to Kerberos, to obtain the user's Kerberos Ticket Granting Ticket. By installing a Kerberos login module, the user would not have to perform additional steps, such as executing the command `kinit` at a later time to obtain the ticket. When the user executes client applications that are attempting to authenticate across the network to certain servers that use the Kerberos protocol, those applications could then use SASL, which would presumably have the appropriate Kerberos mechanism plugged in to perform the actual authentication.

8.4.1 Subjects and Principals

Users often depend on computing services to assist them in performing work. Furthermore, services themselves might subsequently interact with other services. JAAS uses the term *subject* to refer to a system entity, such as a user or a computing service. To identify the subjects with which it interacts, a computing service typically relies on names. However, a subject might not have the same name for each service and, in fact, may even have a different name for each individual service. The term *principal* represents a name associated with a subject [71]. Because a subject may have multiple names, potentially one for each service with which it interacts, a subject in JAAS comprises a set of principals.

Once a subject is authenticated, an instance of `javax.security.auth.Subject` is created to represent that subject and is populated with objects that implement the `java.security.Principal` interface. Authentication represents the process by which one system entity verifies the identity of another and must be performed in a secure fashion; otherwise, an intruder may impersonate others to gain access to a system. Authentication typically involves the subject demonstrating possession of some form of evidence to prove its identity. Such evidence may be information only the subject would be likely to know or have, such as a password or smart card, or that only the subject could produce, such as signed data using a private key.

When it attempts to authenticate to a service, a subject typically provides the proof of its identity along with its name. If the authentication attempt succeeds, the service associates a service-specific `Principal`, using the given name, with the `Subject`. Applications and services can determine the identity of the `Subject` simply by referencing the relevant `Principal` associated with that `Subject`.

Reliance on named principals usually derives from the fact that a service implements a conventional access control model of security [69]. This model allows a service to define a set of protected resources and the conditions under which named principals may access those resources. Both KeyNote [14] and SPKI [34] have focused on the limitations of using conventional names in large distributed systems for access control and note that public keys, instead, provide a more practical and scalable name representation. JAAS and SPKI do not impose any restrictions on principal names. Localized environments that have limited namespaces or that do not rely on public-key cryptography may define principals that have conventional names. Large-scale distributed systems may use principals that allow the principal name to be a public key.

8.4.2 Credentials

In addition to `Principal` information, some services may want to associate other security-related attributes and data with a `Subject`. JAAS calls such generic security-related attributes *credentials*. A credential may contain information that could be used to authenticate the subject to additional services. Some common types of credentials are passwords, Kerberos tickets [87] and public-key certificates. Many of these credential forms are used in environments that support single sign-on. Credentials may also contain data that simply enables the subject to perform certain activities. Cryptographic keys, for example, represent credentials that enable the subject to sign or encrypt data. In JAAS, credentials may be any type of object. Therefore, existing credential implementations, such as `java.security.cert.Certificate`, can be easily incorporated into JAAS. Third-party credential implementations may also be plugged in to the JAAS framework.

Although Kerberos tickets and cryptographic keys exemplify common types of credentials, credentials can represent a wider range of security-related data. Applications running on behalf of subjects must coordinate with the services on which they depend so as to agree on the kinds of credentials that are needed and recognized during their interactions. Thus, some credentials might be standard or well recognized, whereas others might be application and service specific. In addition, credential implementations do not necessarily have to contain the security-related data; they might simply reference that data. This occurs when the data must physically reside on a separate server or hardware device, such as private keys on a smart card.

A subject must successfully authenticate to a service to obtain credentials. On successful authentication, the service creates the appropriate credential object and associates it with the `Subject`. Once a `Subject` has been populated with credentials, applications considered to be running on behalf of the subject may, with the proper permissions, then access and use those credentials. JAAS does not impose any restrictions about credential delegation to third parties. Rather, JAAS either allows each credential implementation to specify its own delegation protocol, as Kerberos does, or leaves delegation decisions up to the applications.

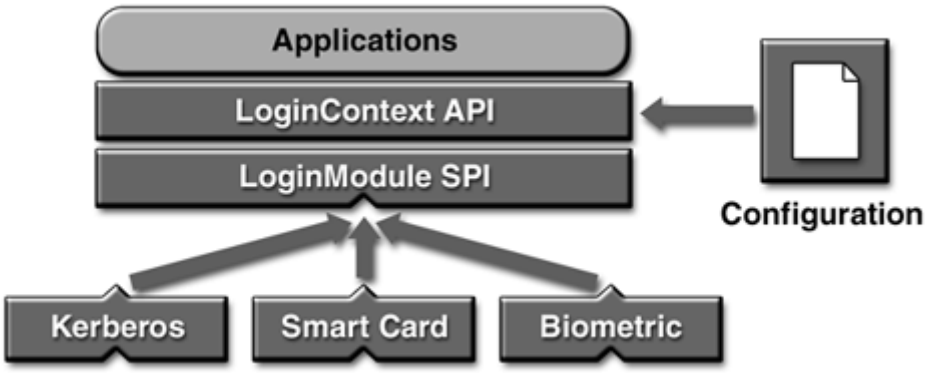
JAAS divides each `Subject`'s credentials into two sets. One set contains the subject's public credentials, such as public-key certificates. The other set stores the subject's private credentials, such as private keys, Kerberos tickets, encryption keys, passwords, and so on. To access a `Subject`'s public credentials, no permissions are required. However, access to a `Subject`'s private credential set requires the caller to have been granted a `PrivateCredentialPermission` for the corresponding credential class.

8.4.3 Pluggable and Stacked Authentication

To authenticate to conventional services, a subject is required to provide its name and some form of proof of its identity. Depending on the security parameters of the particular service, different kinds of proof may be required. The JAAS authentication framework is based on PAM (Pluggable Authentication Modules) [112] and thus supports an architecture that allows system administrators to plug in the appropriate authentication mechanisms to meet security requirements. The

architecture also enables applications to remain independent of the underlying authentication technology. Hence, as new authentication services become available or as current services are updated, system administrators can easily deploy them without having to modify or recompile existing applications. The authentication framework is depicted in Figure 8.2.

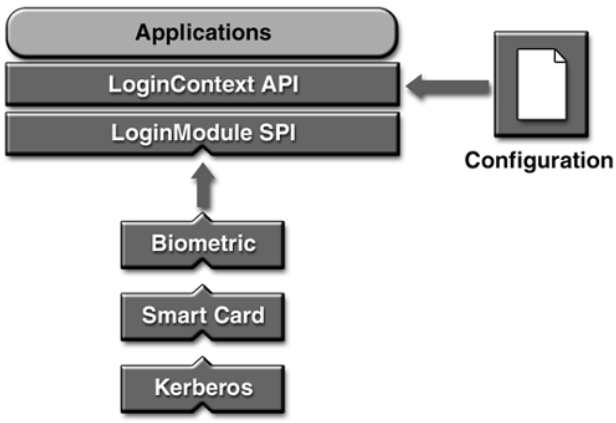
Figure 8.2. Pluggable authentication



When attempting to authenticate a subject, an application calls into the authentication framework, which JAAS defines as a *login context*. The `javax.security.auth.login.LoginContext` class provides the basic methods used to authenticate subjects and provides a way to develop applications independent of the underlying authentication technology. The `LoginContext` consults an instance of `javax.security.auth.login.Configuration` to determine the authentication mechanism(s), or *login module(s)*, configured for a particular application. Different implementations of the `javax.security.auth.spi.LoginModule` interface can be plugged in under an application without requiring any modifications to the application.

Each login module may authenticate a subject by using a different authentication technology. For example, a conventional password-based login module prompts for a user name and verifies a password; a smart card login module instructs the subject to insert the card into the card reader and verifies a personal identification number (PIN); and a biometric login module prompts for a user name and verifies some physical characteristic of the subject, such as a fingerprint or retina scan. Depending on the security requirements of the application, a system administrator configures the appropriate login module. In fact, system administrators may also plug in multiple login modules under an application. This type of stacked configuration is depicted in Figure 8.3.

Figure 8.3. Stacked authentication



A subject authenticates to the login modules in the order specified by the configuration. In general, regardless of whether a login module succeeds or fails, the subject continues to authenticate to the ensuing login modules on the stack. This helps hide the source of failure from potential attackers. Additional parameters within the configuration allow for exceptions to this rule and also determine which login modules must succeed for the overall authentication to succeed. Details on the login configuration syntax, semantics, and location appear in Section 12.7. The login context reports a successful authentication status back to the calling application only if all the necessary login modules, as determined by the configuration, succeed. To guarantee this, the login context performs the authentication steps in two phases. Both phases must complete successfully for the login context to return an overall authentication status noting success.

A typical application instantiates a `LoginContext` class, passing in a name and an instance of a class that implements the `javax.security.auth.callback.CallbackHandler` interface. `LoginContext` uses the name as the index into the configuration to determine which login modules should be used and which ones must succeed in order for the overall authentication to succeed. The `CallbackHandler` interface is described in Section 8.4.4.

The application then calls the login context's `login` method, which creates a new empty `Subject`—if no `Subject` was specified in the `LoginContext` constructor—and then does the following:

1. The login context invokes each configured login module and instructs it to verify the identity of the subject. If all the necessary login modules successfully pass this phase, the login context then enters the second phase.
2. The login context invokes each configured login module again, instructing it to formally commit the authentication process. During this phase, each login module associates any relevant `Principal`s, which hold the authenticated identities, and credentials with the `Subject`.

Thus, once the overall authentication process has completed, the calling application can enumerate the `Subject`'s collection of `Principal`s to obtain its various identities and can traverse through a `Subject`'s credentials to access supplementary data. Some login modules might associate only credentials, not principals, with the subject. A smart card login module, for example, might authenticate the subject by verifying a provided PIN and, on success, simply associate with the `Subject` a credential referencing a cryptographic key on the card. The smart card module in this case does not associate a `Principal` with the `Subject`.

If either phase fails, the login context invokes each configured login module and instructs it to abort the entire authentication attempt. Each login module then cleans up any relevant state it had associated with the authentication attempt.

During this two-phase process, if a particular login module fails, it does not sleep and does not attempt to retry the authentication. Otherwise, the entity attempting the authentication could detect which login module failed. The calling application owns the responsibility of performing such tasks as reattempting authentication and may elect to perform such tasks after each two-phase round of authentication.

During the authentication process, login modules have the choice and ability to share information with one another; whether one does depends on its security requirements. One motivation for sharing information is to help achieve single sign-on. For example, stacked login modules may share user name and password information, thereby enabling a user to enter that information only once but still get authenticated to multiple services. In the case of a subject having different user names and passwords for each service, login modules may also coordinate with one another to map such information into the relevant service-specific information. Thus, although the subject

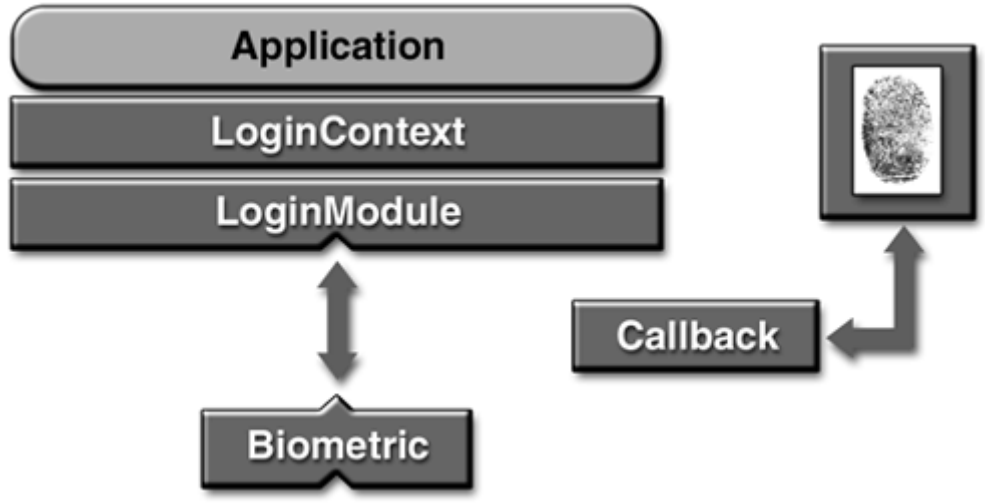
enters only a single user name and password, that information gets mapped into the respective service-specific user names and passwords, thereby enabling the subject to authenticate to multiple services again with relative ease.

8.4.4 Callbacks

By using the login context, applications remain independent from underlying login modules. Login modules may be plugged in under any type of application. A login module must be able to gather information from and display information to subjects via the calling application. For example, a login module that requires user name and password information needs the ability to prompt the subject for such information without knowing whether the calling application has a graphical user interface (GUI).

The login context achieves this independence by allowing applications to specify a *callback* that underlying login modules may use to interact with subjects. Applications provide a `CallbackHandler` implementation when a login context is instantiated. This `CallbackHandler` is passed to each login module. Login modules may then invoke the callback to gather or display the relevant information. The `CallbackHandler`, implemented by the application, inherently knows whether to construct a graphical window or simply to use a standard output stream. The callback design and usage is depicted in Figure 8.4.

Figure 8.4. Callback design and usage



8.4.5 Authorization

Authentication serves as the basis for *authorization* [71]. Specifically, once it knows the identity of a subject, an application may then specify what set of operations that subject may perform. A `Subject` simply represents a nameless container holding relevant information for a user (subject), whereas `Principal`s represent authenticated identities for that `Subject`. The set of permissions granted to a `Subject` depends on the `Principal`s associated with that `Subject`, not on the `Subject` itself. In other words, permissions are granted to a `Subject`, based on the authenticated `Principal`s it contains. This set of permissions granted can be configured within an external access control policy.

8.4.6 Principal-Based Access Control

Once we have authenticated a subject, the trust relationship can be enforced with Java 2 authorization mechanisms. Thus, we can impose access controls on the `Principal`s associated with the authenticated identities in the `Subject`. Principal-based access controls—access controls based on who is running the code—extend Java 2 code-source-based access controls.

8.4.7 Access Control Implementation

The Java 2 runtime environment enforces access controls via the currently installed `java.lang.SecurityManager` implementation, which is consulted any time code attempts to perform a security-sensitive operation. To determine whether the code has sufficient permissions, the default `SecurityManager` implementation delegates responsibility to the `java.security.AccessController`. The `AccessController` obtains a snapshot of the current `AccessControlContext` and then ensures that the code referenced by the `AccessControlContext` contains sufficient permissions for the operation to proceed. (See Chapter 6.)

JAAS enables the Java 2 security architecture to make subject-based authorizations with the `javax.security.auth.Subject.doAs` methods. These methods dynamically associate a `Subject` with the current `AccessControlContext`. Privileged code is then considered to be executed on behalf of the specified `Subject`. Hence, as subsequent access control checks are made, the `AccessController` can base its decisions on both the executing code itself and the `Principal`s associated with the `Subject`. The following static methods may be called to perform an action on behalf of a particular `Subject`:

```
public static Object
    doAs(Subject subject,
         java.security.PrivilegedAction action);

public static Object
    doAs(Subject subject,
         java.security.PrivilegedExceptionAction action)
      throws java.security.PrivilegedActionException;

public static Object
    doAsPrivileged(Subject subject,
                   java.security.PrivilegedAction action,
                   java.security.AccessControlContext acc);

public static Object
    doAsPrivileged(Subject subject,
                   java.security.PrivilegedExceptionAction
action,
                   java.security.AccessControlContext acc)
      throws java.security.PrivilegedActionException;
```

The `doAs` methods first associate the specified `Subject` with the current thread's `AccessControlContext` and then execute the action. This achieves the effect of having the action run as the `Subject`. This is accomplished by constructing an `AccessControlContext` with an implementation of `java.security.DomainCombiner` and then binding it to the current execution thread

by invoking a corresponding `AccessController.doPrivileged` method with the `AccessControlContext`. See Section 6.4.4.

The first method can throw runtime exceptions, although normal execution has it returning an `Object` from the `run` method of its `action` argument. The second method behaves similarly except that it can throw a checked exception from its `PrivilegedExceptionAction run` method.

The `doAsPrivileged` methods behave exactly the same as the `doAs` methods, except that instead of associating the provided `Subject` with the current `Thread`'s `AccessControlContext`, they use the provided `AccessControlContext`. In this way, actions can be restricted by `AccessControlContext`s different from the current one.

An `AccessControlContext` contains protection domains with information about all the code executed since the `AccessControlContext` was instantiated, including the code locations and the permissions the code is granted by the policy. In order for an access control check to succeed, the policy must grant each protection domain referenced by the `AccessControlContext` the required permissions.

If the `AccessControlContext` provided to `doAsPrivileged` is `null`, the action is not restricted by a separate `AccessControlContext`. This may be useful in a server environment, for example. A server may authenticate multiple incoming requests and perform a separate `doAs` operation for each request. To start each action "fresh," and without the restrictions of the current server `AccessControlContext`, the server can call `doAsPrivileged` and pass in a `null AccessControlContext`.

A `javax.security.auth.AuthPermission` with target name `"doAs"` is required to call the `doAs` methods, and an `AuthPermission` with target `"doAsPrivileged"` is required to call the `doAsPrivileged` methods. This ensures that only authorized entities can associate a subject with the execution context.

With these mechanisms in place, we can now inject additional context to the access control algorithm described in Section 6.4.8. Recall that the JVM keeps track of each `ProtectionDomain` for every class executing on the call stack. Whenever a `Subject` is associated with the execution context, each `ProtectionDomain` on the call stack is effectively assigned the `Principal`s from that `Subject`. This binding of the principals to a `ProtectionDomain` is accomplished by an implementation of a `DomainCombiner` and is discussed in Section 6.3. To reiterate, as of J2SE 1.4, when an access control (permission) check is invoked, the `Policy` is consulted to determine, for each `ProtectionDomain` on the call stack, whether the requisite permission is granted to the code and principals indicated by components of the `ProtectionDomain`. Both the dynamic permissions and the static permissions, assigned when the class was loaded, are considered when making the access control decision.

8.5 Distributed End-Entity Authentication

With the increasing use of distributed systems, users need to access resources that are often remote. Traditionally, users have had to sign on to multiple systems, each of which may involve different identities and authentication technologies. In contrast, with single sign-on, the user needs to authenticate only once, and the authenticated identity is securely carried across the network to access resources on behalf of the subject.

The Java 2 platform supplies rich APIs that, when used in conjunction with the authentication framework, integrate with standard distributed environments. An example in J2SE is a `LoginModule` implementing the Kerberos V5 protocol, `com.sun.security.auth.module.Krb5LoginModule`. An application that requires peer-to-peer authentication can use the inherent authentication and credential delegation capabilities of the Java Generic Security Services API (Java GSS-API) [58] to carry network authentication credentials to an end entity.

Another example of support for a distributed authentication protocol is the implementation of the Java Secure Socket Extension (JSSE), which enables secure Internet communications. JSSE provides a framework and an implementation for a Java language binding of the SSL and TLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Before we dive into the description of these frameworks, we describe an oft overlooked feature that can be used to accommodate client authentication to a distributed service: the `java.net.Authenticator`.

The remainder of this chapter describes key concepts of end-entity authentication with respect to Java GSS-API and the JSSE API. Descriptions of the details of these APIs and their use are deferred until Chapter 11 in order to present a cohesive view of the APIs with respect to not only authentication but also the confidentiality and integrity-protection mechanisms they provide.

8.5.1 `java.net.Authenticator`

The `Authenticator` class represents an object that knows how to obtain authentication information for a network connection. Usually, it will do this by prompting the user for information. Generally, the `Authenticator` is used for authentication protocols that rely on some form of user name and password; therefore, an implementation of `Authenticator` conveys the results of its interaction with the user via an instance of the `java.net.PasswordAuthentication` class. The `PasswordAuthentication` class is a data holder and is simply a repository for a user name and a password.

Applications use the `Authenticator` class by creating a subclass and registering an instance of that subclass with the system via the `setDefault` method. Note that currently this is a runtime-wide setting and therefore impacts all protocol handlers. When authentication is required, the system will invoke the static `requestPasswordAuthentication` method. This method populates the instance variables of the registered `Authenticator` and then dispatches into the `Authenticator`, typically with a call to the `getPasswordAuthentication` method. The subclass's `getPasswordAuthentication` method can call a number of inherited `getRequestingXXX` methods to perform queries about the authentication being requested and can then form an appropriate message for the user.

8.5.2 Single Sign-on in a Kerberized Environment

In this section, we provide a brief overview of how to accomplish single sign-on based on the Kerberos V5 protocol. JAAS can be used to authenticate a principal to a Kerberos authentication server and obtain credentials that prove the principal's identity. As one option, Sun's implementation of a Kerberos login module can be made to read credentials from an existing cache on platforms that contain native Kerberos support. Once the Kerberos credentials have been obtained, the Java GSS-API can be used to authenticate to a remote peer using the credentials. Here, we give brief descriptions of Kerberos V5, the Generic Security Service API, and the `Krb5LoginModule` from Sun. Details of Java GSS-API and its use are deferred until Chapter 11.

Kerberos V5

Kerberos V5 is a trusted third-party network authentication protocol designed to provide strong authentication using secret-key cryptography. When using Kerberos V5, the user's password is never sent across the network, not even in encrypted form. Kerberos was developed in the mid-1980s as part of MIT's Project Athena. A full description of the Kerberos V5 protocol is beyond the scope of this text. For more information on the Kerberos V5 protocol, refer to [65] and [95].

Generic Security Service API (GSS-API)

The Generic Security Service API, described in a language-independent form in RFC 2743 [77], offers security services such as authentication, message confidentiality and integrity, sequencing of protected messages, replay detection, and credential delegation. The underlying security technology, or security mechanism, being used has a choice of supporting one or more of these features beyond the essential one-way authentication.

The API is designed such that an implementation may support multiple mechanisms simultaneously, giving the application the ability to choose one at runtime. Mechanisms are identified by means of unique object identifiers (OIDs). For instance, the Kerberos V5 mechanism is identified by the OID {iso(1) member-body(2) United States(840) mit(113554) infosys(1) gssapi(2) krb5(2)}.

Another important feature of the API is that it is token based. That is, calls to the API generate opaque octets that the application must transport to its peer. This enables the API to be transport independent.

The Kerberos Login Module

The class `com.sun.security.auth.module.Krb5LoginModule` is Sun's implementation of a login module for the Kerberos V5 protocol. On successful authentication, the Ticket Granting Ticket (TGT) is stored in the `Subject`'s private credentials set, and the appropriate Kerberos `Principal` is stored in the `Subject`'s `Principal` set.

The `Krb5LoginModule` has been implemented such that it can use the operating environment's native credentials cache to acquire the TGT and/or use a keytab file containing the secret key so as to authenticate a principal implicitly. Due to the sensitivity of this information, it is imperative that the cached credentials not be handed out to untrusted code. Therefore, the ability of the login module to acquire these credentials is restricted to code possessing the appropriate `javax.security.auth.kerberos.ServicePermission`.

`ServicePermission`s protect Kerberos services and the credentials necessary to access those services. There is a one-to-one mapping of a service principal and the credential necessary to access the service. Therefore, granting access to a service principal implicitly grants access to the credential necessary to establish a security context with the service principal. This applies regardless of whether the credential is in a cache or acquired via an exchange with the KDC (Key Distribution Center). The credential can be a Ticket Granting Ticket, a service ticket, or a secret key from a key table.

A `ServicePermission` embodies the service's principal name and a list of actions that specify the context with which the credential can be used. Granting a `ServicePermission` implies that the caller can use a cached credential—TGT, service ticket, or secret key—within the context designated by the action. In the case of the TGT, granting this permission also implies that the TGT can be obtained by an Authentication Service exchange.

8.5.3 JSSE Authentication Mechanisms

The Java Secure Socket Extension (JSSE) includes machinery to authenticate end entities as part of its underlying SSL/TLS protocol. (Throughout this book, we use the abbreviations SSL and TLS interchangeably. Unless noted otherwise, we are referring to both the SSL and TLS protocols.) We defer discussion of the API until Chapter 11 so as to present a unified view of its full capabilities.

The intricacies of the TLS protocol message exchange are not germane to this discussion. However, during the TLS handshake, both the server and the client can solicit authentication credentials from the peer entity. The credentials used to authenticate the end entity are typically an X.509 certificate or a certificate chain. JSSE introduces the *key manager* and *trust manager* abstractions as the interfaces to the process of establishing trust. Key managers select keying material; trust managers decide whether to trust the received material.

Key managers and trust managers use keystores for their key material. A key manager administers a keystore and supplies public/private keys to the TLS mechanism as needed. A trust manager makes decisions about whom to trust, based on information in the truststore, a special type of keystore, it manages.

A *keystore* is a database of key material. Key material is used for a variety of purposes, including authentication and data integrity. Various types of keystores are available, including PKCS12 and Sun's JKS.

Keystores are described in detail in Section 12.8.1. Some of that information is summarized here, and the term *truststore* is also introduced to refer to a type of keystore that contains only certificates referring to trusted entities.

Generally speaking, keystore information can be grouped into two different categories: key entries and trusted certificate entries. A key entry consists of an entity's identity and its private key and can be used for a variety of cryptographic purposes. In contrast, a trusted certificate entry contains only a public key in addition to the entity's identity. Trusted certificate entries don't have private keys and are ignored by key managers. That is, a trusted certificate entry cannot be used where a private key is required, such as in a `javax.net.ssl.KeyManager`. In the J2SDK implementation of the JKS type of keystore, the keystore may contain both key entries and trusted certificate entries.

A *truststore* is a keystore used when making decisions about whom to trust. If data is received from an entity that is trusted and if the claimed identity of the sender can be verified, one can assume that the data originated from that entity.

An entry should be added to a truststore only if the user makes a decision to trust that entity. By importing a public key into a truststore, the user is explicitly making a trust decision, and thus the new entry in the store is considered a trusted entry.

It may be useful to have two different keystore files: one containing key entries and the other containing trusted certificate entries, including Certification Authority (CA) certificates. The former contains private information; the latter does not. More protection for the private keys can be provided if the private keys are stored in a keystore with restricted access, whereas the trusted certificates are supplied in a more publicly accessible keystore, if needed.

Authenticating the Server

In TLS, the authentication step is optional. However, common practice is always to authenticate the server. Authenticating the server to the client ensures that the server represents the end entity

the server claims to represent. To prove that a server belongs to the organization that it claims, the server presents its public-key certificate (identity certificate) to the client. The client uses the public key in the certificate to initiate the key-exchange algorithm with the server. Thus, the communicating peer must have the corresponding private key to engage in the TLS handshake successfully. If the handshake succeeds and the certificate is valid, the client can be certain of the identity of the server. Because the identity bound to the certificate is the DNS domain name of the server, this creates a convenient means by which to thwart DNS-based attacks.

Chapter 9. Object Security

If we cannot secure all our rights, let us secure what we can.

—Thomas Jefferson

As you develop applications using the Java language and platform, and especially when you consider security features, you knowingly or unknowingly depend on the underlying object orientation, such as data encapsulation, object namespace partition, and type safety. This dependence is also evident in the protection of the runtime's internal state, which is often represented and maintained as objects. For example, when using the Java Remote Method Invocation (Java RMI) package to build distributed Java applications that span multiple JVMs, you will sometimes find it convenient or even necessary to protect the state of an object for integrity and confidentiality when the state is transported from one machine to another. These security requirements exist when the objects of concern are inside a runtime system—for example, in memory—in transit—for example, stored in IP packets—or stored externally—for example, saved on disk.

These requirements entail a whole range of object-level security issues that must be correctly handled during system development, in addition to code-signing and policy-driven, fine-grained access control mechanisms. This chapter provides a number of secure-programming techniques to use when programming in the Java language. The chapter also describes three interfaces for signing, sealing—encrypting—and guarding objects. It begins by discussing some general practices.

9.1 Security Exceptions

It is not uncommon for a piece of code to catch an exception thrown from lowerlevel code and then either (1) mask this by translating the exception into a higher-level exception and rethrowing it or (2) do some processing that results in its "swallowing" the exception. For example, suppose that you write a class, `MyPasswordChecker`, that checks a user's password when the user logs in. If the password check fails because the user name does not exist, it is bad practice to let the user know that the name has been given wrongly, as doing this would help an attacker guess available user names. Instead, a more general error message, such as "login failed," should be given:

```
public class MyPasswordChecker {
   public void check(String name, String password)
            throws LoginFailureException {
      try {
      // Call the real password checking routine.
      ...
      } catch (NoSuchUserException e) {
         throw new LoginFailureException():
      }
   }
}
```

However, you should be extremely careful when writing code that masks or swallows security exceptions, such as `java.security.AccessControlException` and `java.lang.SecurityException`, because such an action could potentially mask serious security breaches. Sometimes, developers get annoyed by a security exception and take matters into their own hands by substituting their own security policy decision for that of `AccessController` or `SecurityManager`. This attitude of "just-make-the-code-work"

is very dangerous, especially if the code being developed might be run as system code and thus be fully trusted. Often, software design can be improved to avoid having to catch and swallow undesirable exceptions.

9.2 Fields and Methods

The Java language provides four access modes, which should be used properly: public, protected, private, and package private, which is the default if no mode is explicitly specified. A common example of improper use is the inexperienced programmer who, when writing a time zone class, mistakenly declares fields or variables that are publicly accessible:

```
public TimeZone defaultZone;
```

This design has a number of problems. First, any person or code, including untrusted code, can access this field and directly change the value of the default time zone. Second, because multiple threads can access this field, some synchronization is needed. Following is a better design:

```
private TimeZone defaultZone = null;
public synchronized void setDefault(TimeZone zone) {
   defaultZone = zone;
}
```

Suppose that after product release you decide that a security check is needed to guard against unauthorized modification to the value of the default time zone, `defaultZone`. For the next release of the product, you can define a new `TimeZonePermission` class—for example, as a subclass of `BasicPermission`—and add code like the following to allow setting the default zone if the `TimeZonePermission` with name `"setDefault"` has been granted:

```
private TimeZone defaultZone = null;
public synchronized void setDefault(TimeZone zone) {
   SecurityManager sm = System.getSecurityManager();
   if (sm != null) {
      sm.checkPermission(new
TimeZonePermission("setDefault"));
      defaultZone = zone;
   }
}
```

Permissions are described in Section 5.1; permission-checking, in Chapter 6.

A critical point about this product update is that it is done in a way that does not break backward compatibility. That is, a third-party application that runs on the earlier version of the release will still have the same API available when running on the new release. If the `TimeZone` was directly exposed as a `public` field, as in the first design, a security check or a synchronization feature could not be added without changing existing APIs.

To recap, never design `public` fields or variables that can be accessed directly. Instead, declare these fields as `private` and provide `public` accessor methods that mediate access to such fields. Moreover, decide carefully, *for every single public method*, if any such access is sensitive and might require a security check. If a field is intended to be a constant, it can be `public` but should be made `static` and `final`, as discussed in the next section.

Even when methods or fields are `protected`, a subclass can access them as long as the class is not `final`. Because an attacker can easily provide a subclass, security may be compromised. Similarly, package private methods and fields may be accessed by any class in the same package. Note that the JVM strictly controls access to package private members, based on its notion of a "runtime package." That is, a class is allowed access to package private members of another class at runtime only if both classes are in the same package and are both defined by the same class loader. Code signing adds an additional layer of protection from rogue classes impersonating as members of a package, due to the requirement that all class files belonging to the same package must be signed by the same signer(s). Regardless, always review all `protected` and package private methods and fields to see whether they should be made `private` and, if not, whether they should be accessed via accessor methods that perform security checks.

9.3 Static Fields

A *static field* is a per class field in the sense that its value is shared by all objects instantiated from the same class. Use of static fields is a minefield that can cause unintended interactions between supposedly independent subsystems. Static fields offer even less protection than per object fields; in the latter case, you must have an object's reference in order to access the field, whereas in the former case, anyone can access the field simply by using the class name directly.

As a result, directly exposed, non-`final public static` variables are extremely problematic for security. Never design a class with such variables. Instead, declare them as `static private`, with appropriate `public static` accessor methods. You still have to decide carefully whether these accessor methods should invoke security checks.

If you have a product already released with such dangerous variables, you should review all non-`final public static` variables and carefully assess the potential damage they can cause. You should eliminate the worst offenders even though doing so breaks backward compatibility. For the rest, if you must keep them for backward compatibility, you can only hope that no one can come up with a way to exploit them.

Another dangerous aspect of static fields is that they can create type-safety problems if used casually. For example, a part of the system code might be designed to share a static field, `Foo`, internally. If `Foo` is typed too loosely, an untrusted applet or application can plant an object of a subtype or a type that is incompatible with what the system programmer intended: for example, when `Foo` is declared to be of type `java.lang.Object`. This kind of substitution can create very subtle security problems that are difficult to detect and correct.

9.4 Private Object State and Object Immutability

Most objects have private, internal states that should not be randomly modified. Often, other objects need to query the state information. Many programmers implement a query method by using a simple `return` statement, as in the following example:

```
public MyClass {
   private boolean status = false;

   public void setStatus(boolean s) {
      status = s;
   }

   public boolean getStatus() {
```

```
      return status;
   }
}
```

No problem so far. However, if `status` is not a simple `boolean` but rather an array of `boolean`, serious problems can occur, as here:

```
public MyClass {
   private boolean[] status = null;
   . . .
   public boolean[] getStatus() {
      return status;
   }
}
```

In this example, once it obtains a reference to the private `status` member, another object can change the value of `status` without `MyClass`'s consent. The reason is that, unlike `boolean` or any other non-`final` primitive type, an *array* of `boolean`s—or an array of anything—is mutable. That is, the array reference returned refers to the internal instance within the `MyClass` object. Such a consequence might not be what the designer of `MyClass` intended, as uncontrolled modification to internal state can lead to incorrect or even malicious results.

There is a twist to this problem. In the example with an array of `boolean`s, the simplest way to implement the `setStatus` method is as follows:

```
public void setStatus(boolean[] s) {
   status = s;
}
```

Again, because `s` is mutable, even after `MyClass` has "taken possession of" it, the object that supplied `s` to `MyClass` can still change the value of `s`. Many programmers overlook this possibility.

In summary, when interacting with potentially malicious code, never return a reference to a mutable object, where changes to it would adversely affect internal object state. Further, you should never directly store a mutable object (by assigning the array reference to an internal variable) if the source of the object may be malicious. Because any code can potentially be malicious, the best practice is to clone or copy the objects before returning or storing them.

This discussion shows how important it is to be able to distinguish immutable objects from mutable ones and how beneficial it is to make objects immutable when possible. For example, `array, Vector`, and `Hashtable` are mutable. Even if an array contains only immutable objects, such as `String`, the array itself is still mutable, and anyone with a reference to the array can change entire objects contained in the array.

Figuring out whether an object is immutable is not always easy, as immutabilitydepends on what fields and methods are available and whether objects used in those cases are also immutable. This analysis might need to be done recursively, until all loose ends are tracked down and resolved to be immutable.

A final word on immutable objects: Because a password is typically seen as a string of characters, it is common to see Java programs in which a `String` is used to represent a password. Given that a `String` is immutable, however, there is no way for the application program to erase it

when it is no longer needed. Its fate is left entirely with the JVM's garbage collector. For better security, you should use `char[]` to represent passwords and wipe out the content of the array after use.

9.5 Privileged Code

Recall from Chapter 6 that a security-sensitive operation is typically allowed only if all the code in the execution environment has been granted the required permission by the security policy, but you can use an `AccessController.doPrivileged` method to mark a segment of code as privileged. The privileged code can then perform operations allowed for that code, whether or not its callers have been granted the required permissions. If a piece of trusted code (such as system code, which is granted all permissions) is privileged, it can load libraries (including native code), read any file, read system properties, and so on, independent of what its callers would normally be allowed to do. A privileged code segment is a critical region in which mistakes can be made and errors can be costly.

When writing privileged code, always try to keep it as short as possible. This practice not only reduces the chance of making mistakes but also makes auditing the code easier so as to ensure that it is accessing only the minimal amount of protected resources.

Also, watch out for the use of *tainted variable*s, ones that are set by the caller—passed in as parameters—and thus not under the control of the privileged code. For example, consider the following privileged code to open a file:

```
public FileInputStream getFileToRead(String filename) {
   FileInputStream fn =
      (FileInputStream) AccessController.doPrivileged(
         new PrivilegedAction() {
            public Object run() {
               return new FileInputStream(filename);
            }
         }
   );
}
```

This code can be used to open a font file when displaying images for applets, even though the original calling applet classes would not have access to the actual font file. However, this example has two flaws. One is that the method is public, so anyone can call it. The other is that there is no sanity check on the file name, so the code blindly opens any file requested by the caller. Either flaw alone can be a problem.[1] Combined, they create the worst possible situation, as now anyone can call this method to open any file desired, assuming that the privileged code has the appropriate permissions, which any system code does.

[1] A different set of problems can occur if untrusted code can take advantage of the first problem and cause a large number of arbitrary files to be opened.

The problem does not stop at public methods. Even if you change `getFileToRead` to be nonpublic, another public method can turn around and invoke `getFileToRead`. In this case, once again a tainted variable is used indirectly by privileged code and security could be compromised.

The most conservative way to design such methods is to make them private. That way, they are not callable from outside their own class.

9.6 Serialization

Serialization is a feature that allows an object's nontransient state to be stored in a "serialized" form, for example, for the purpose of transporting the object to another machine and then deserializing it, or reconstructing the object, at the destination. RMI uses serialization extensively, as do other packages. Objects are serialized and deserialized via output and input stream classes `ObjectOutputStream` and `ObjectInputStream`. Default implementations of two methods, `ObjectOutputStream.writeObject` and `ObjectInputStream.readObject`, are invoked for serialization and deserialization, respectively. You also can write, for a serializable class—one that implements the `java.io.Serializable` interface—`writeObject` and `readObject` methods to customize how serialization and deserialization are done. Following, we provide recommendations on defensively programming implementations of `Serializable`. Also, this topic is covered extensively in [15].

Security-conscious implementers should keep in mind that a serializable class's `readObject` method is, in effect, a public constructor that takes as input the values for each of the serialized object's nontransient fields and creates a new object instance. Under these circumstances, implementers should enforce the same restrictions that would be used if the object were created with a public constructor. Also, as the input to `readObject` can be provided by an adversary whose goal is to compromise the object under construction, you cannot safely assume that the input content was generated via the serialization of a properly constructed object of the correct type. As a result, if `readObject` blindly takes its input, various security problems can occur. This is true whether `readObject` is implicit—that is, the default provided by the JRE (Java Runtime Environment) implementation—or explicit, provided by the serializable class in question. In fact, the default implementation of `readObject` does no validity checking whatsoever.

In good defensive programming, if a class has any private or package private fields on which it maintains invariants, an explicit `readObject` method should be provided that checks that these invariants are satisfied, as in the following example:

```
private void readObject(ObjectInputStream s) throws
         IOException, ClassNotFoundException {
   s.defaultReadObject();
   if (<invariants are not satisfied>)
      throw new java.io.InvalidObjectException();
}
```

Further, if a class has any object reference fields that are private or package private and if the class depends on the fact that these object references are not available outside the class or package, the objects referenced by those fields must be defensively copied as part of the deserialization process. That is, the subobjects deserialized from the stream should be treated as untrusted input in that newly created objects, initialized to have the same values as the deserialized subobjects, should be substituted for the subobjects by the `readObject` method.

For example, suppose that an object has a private `byte` array field, `b`, that must remain private. Then `b` should be a clone of the result from `readObject`, as follows:

```
private void readObject(ObjectInputStream s) throws
         IOException, ClassNotFoundException {
   s.defaultReadObject();
   b = (byte[])b.clone();
   if (<invariants are not satisfied>)
```

```
        throw new java.io.InvalidObjectException();
}
```

Note that calling `clone` is not always the right way to copy a subobject defensively. If the `clone` method cannot be counted on to produce an independent copy and not to "steal" a reference to the copy, for example, when the class of the subobject is not `final`, an alternative way to produce the copy should be used. Also note that a way to avoid the overhead of copying and still have the same safety, as long as a shallow copy is OK, is either to (1) have a custom `readObject` method invoke the `ObjectInputStream`'s `readUnshared` method rather than its `readObject` method to read an object or (2) mark a serializable field as unshared with an explicit `serialPersistentFields` declaration, using the `ObjectStreamField` constructor that takes a `boolean`.

As a conservative alternative to using an explicit `readObject` method to ensure the integrity of deserialized objects, use a `readResolve` method instead, calling a public constructor from within that method. This absolutely guarantees that the deserialized object is one that could have been produced with a public constructor.

In J2SE 1.4, we added support for an additional private method that a serializable class can declare:

```
private void readObjectNoData() throws ObjectStreamException
```

This feature was added to allow a serializable class to perform validation that wouldn't otherwise be possible on deserialized instances. The class's `readObjectNoData` method, if declared, is invoked when deserializing an instance of a subclass, and the serialized data does not identify the class as a superclass of that subclass. The purpose of the `readObjectNoData` method is to validate and initialize the deserialized instance's state for the class when this situation occurs.

This situation may occur when the receiving party uses a different version of the deserialized instance's class than the sending party, and the receiver's version extends classes that are not extended by the sender's version. This may also occur if the serialization stream has been tampered with; hence, `readObjectNoData` is useful for initializing deserialized objects properly despite a "hostile" or incomplete source stream.

Prior to J2SE 1.4, if the object were deserialized, the object's fields declared by the class would just be assigned default values (`null`, `zero`, or `false`), and its `readObject` method would not be invoked. Although this may seem wrong, logically, `readObject` can't be invoked, because there is no instance data in the stream for the object. That is, `readObject` methods assume that they are passed the `ObjectInputStream` in a state in which their class's serialized instance data can be read, but that is not the case in the situation being discussed.

The intention of the pre–J2SE 1.4 behavior was to allow the addition of serializable superclasses as part of compatible class evolution. But in many cases, such behavior doesn't make sense, and it subverts the superclass's attempts at validation or other constructorlike activity. Therefore, in J2SE 1.4, when the serialization stream does not list the class containing the `readObjectNoData` method as a superclass of the object being deserialized, `ObjectInputStream` invokes the class's `readObjectNoData` method to allow it to do such things as validate the object, initialize the instance's specific state, and so on.

In general, if a non-`final` serializable class declares a `readObject` method to do something special in the event of all the class's serializable fields having their default values (`null`, `zero`, or `false`), it should also declare a `readObjectNoData` method to do the same thing. (In

many cases, it might be appropriate to declare a `readObjectNoData` method that throws `InvalidObjectException` unconditionally.)

Here are some more points to remember when implementing a `Serializable` interface. First, use the `transient` keyword for fields that contain direct handles to system resources and information relative to an address space. Otherwise, if a resource, such as a file handle, is not declared transient, when the object is serialized/deserialized, a part of the state probably won't get preserved/restored properly. This results in the object, after being deserialized, having improper access to resources and thereby causing security breaches or errors. In fact, for correctness, system-specific references should be declared transient, as they make no sense in a different environment in which the object is to be deserialized.

Second, as stated earlier, you should guarantee that a deserialized object does not have state that violates some invariants by having a class define its own deserializing methods. Because deserializing an object is a kind of object creation, if untrusted code has a restriction regarding object creation, you must ensure that that untrusted code has the same restriction when it deserializes the object. To illustrate the problem, consider the situation when an applet creates a frame. Security requires that a frame always be created to include a warning label: "This is an applet window." If the frame is serialized by anyone, including the applet, and then deserialized by an applet, you must ensure that the frame comes back up with the same warning banner.

Finally, when the state of a serialized object is outside the JVM, such as when being transported to another machine, the state can potentially be corrupted. Although such corruption cannot be directly prevented by the Java security system, measures can be taken to detect whether corruption has occurred. One way is to encrypt the byte stream produced by serialization. Another way is to use `SignedObject` and `SealedObject`, which are covered later in this chapter. However, such measures do not come free, as cryptographic keys must be managed, and this is far from a trivial task.

9.7 Inner Classes

Inner classes as currently defined have some security implications. Suppose that class A has a private field accessible only from within the class itself. Further suppose that A is rewritten to use inner classes and now encloses an inner class B that requires access to the private field. During compilation, the compiler automatically inserts into the definition of A a package private access method to the private field so that B can call this method. One side effect of this design is that any class in the same package as A and B will be able to call the access method and thus access the private field whose access had been forbidden to it prior to the use of inner classes. Transforming a field from private to package private does not naturally lead to security problems, but you must take care to examine the consequences of such transformations.

The use of inner classes has another design side effect. Suppose that class B is a protected member of class A. After compilation, B's class file defines itself as a public class, but an attribute in the class file correctly records the protection mode bit. Similarly, if B is a private member of A, B's class file defines itself as having package scope with an attribute that declares the true access protection mode. This side effect is not a problem by itself. However, any implementation of the JVM must perform the extra check and honor the true protection attributes.

9.8 Native Methods

Be careful when using native methods. Native methods, by definition, are outside the Java security system. Neither the security manager nor any other Java security mechanism is designed to control the behavior of native code. Thus, errors or security breaches in native code can be a lot more deadly. You should examine native methods for the parameters they take and the values they return. In particular, if a native method does something that bypasses Java security checks, you

must be very careful about the access mode of the method. If the mode is public, anyone can call the method. You must examine the consequences and decide whether that method should be made private.

9.9 Signing Objects

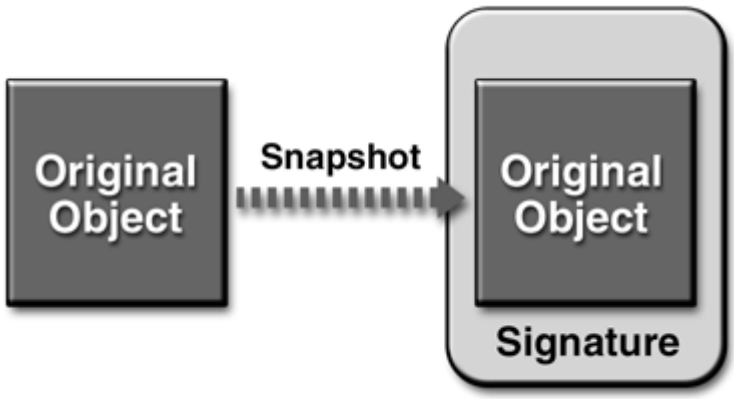
Recall the earlier discussion about the need to protect an object when it is in serialized state and during transit. In fact, quite a few situations exist in which the authenticity of an object and its state must be assured. Following are three examples.

- An object acting as an authentication or authorization token is passed around internally to any Java runtime as part of the security system functions. Such a token must be unforgeable, and any innocent or malicious modification to its state must be detected.
- An object is transported across machines (JVMs), and its authenticity still needs to be verified.
- An object's state is stored outside the Java runtime, for example, onto a disk for JVM restarting purposes.

The class `java.security.SignedObject` defines interfaces to sign objects. A series of nested `SignedObject`s can be used to construct a logical sequence of signatures that resemble a chain of authorization and delegation.

The object to be signed must be `Serializable`, that is, must implement the `Serializable` interface, because when a `SignedObject` is created, the object is always serialized before its digital signature is generated. A `SignedObject` contains the serialized object, the signature, and the name of the algorithm used to generate the signature, as depicted in Figure 9.1.

Figure 9.1. Signed object



The signed object is a "deep copy" (in serialized form) of an original object. Once the copy is made, further manipulation of the original object has no side effect on the copy. In fact, a `SignedObject` is immutable.

The `SignedObject` constructor and public methods follow; for brevity, exception declarations are not listed:

```
public SignedObject(Serializable object, PrivateKey
signingKey,
```

```
       Signature signingEngine)
public Object getObject();
public byte[] getSignature();
public String getAlgorithm();
public boolean verify(PublicKey verificationKey,
      Signature verificationEngine);
```

This class is intended to be subclassed in the future so as to allow multiple signatures on the same signed object. In this case, existing method calls in this base class are fully compatible semantically. In particular, any `get` method returns the unique value if there is only one signature; an arbitrary value from the set of signatures if there is more than one signature.

To create a `SignedObject`, first instantiate a `Signature` object that will be used to generate the signature. In creating the `Signature`, pass the name of the desired signature algorithm, such as `"SHA1withDSA"`, and the provider whose implementation of that signature algorithm is to be used. After the `Signature` is created, instantiate the `SignedObject`, passing it the `Serializable` object to be signed, the private key to be used to sign the object, and the `Signature` object to be used to generate the signature. Thus, typical code for creating a `SignedObject` follows:

```
Signature signingEngine =
      Signature.getInstance(algorithm, provider);
SignedObject so =
      new SignedObject(myobject, privatekey, signingEngine);
```

Typical code for verification and object retrieval is as follows, having received `SignedObject so`:

```
Signature verificationEngine =
      Signature.getInstance(algorithm, provider);
if (so.verify(publickey, verificationEngine))
   try {
      // call getObject to deserialize and return the object
      Object myobj = so.getObject();
   } catch (ClassNotFoundException e) {};
```

Obviously, for verification to succeed, the specified public key must be the one corresponding to the private key used to generate the signature, and the signature algorithm used for verification must be the same as the one used to generate the signature. Also, the security of `SignedObject` depends on the underlying digital signature algorithm and key management system not having been compromised. The signing, or verification, engine does not need to be initialized prior to creating a `SignedObject` or verifying the signature, as it will automatically be initialized by the `SignedObject` constructor or `verify` method.

The `getObject` method in a sense loses type information by returning an object of the type `Object`, so the signed objects likely will be used between collaborating parties so that the correct casting can be done. For example, the previous code can be changed as follows. Suppose that the object passed to the `SignedObject` constructor as shown earlier is the following:

```
String myobject = new String("Greetings.");
```

Then the signature can be verified, and the object retrieved, via the following:

```
if (so.verify(publickey, verificationEngine))
   try {
      String myobj = (String) so.getObject();
   } catch (ClassNotFoundException e) {};
```

In fact, it is probably more common to subclass `SignedObject` so that the correct casting is performed inside the subclass. In this way, static typing information is better preserved.

More important, for flexibility reasons, the `verify` method allows customized signature engines, which can implement signature algorithms that are not installed formally as part of a cryptography provider. However, it is crucial that the programmer writing the verifier code be aware what `Signature` engine is being used, as its own implementation of `verify` is invoked to verify a signature. In other words, a malicious `Signature` might choose to always return `true` on verification in an attempt to bypass security checks. For similar reasons, `verify` in the `SignedObject` class is `final`.

Note that signing objects is different from signing JAR files that contain class files. Signing *code* facilitates the authentication of static code—bytecode in the case of Java technology, native code in the case of Microsoft's Authenticode—whereas signing *objects* is done with objects that might represent a complex transaction application, complete with active state information.

9.10 Sealing Objects

The `SignedObject` class provides object authenticity, or integrity. The class `SealedObject`, on the other hand, protects an object's confidentiality.[2] These two classes may be combined to provide integrity and confidentiality at the same time.[3] In fact, from a technical design perspective, designing the two classes into one would have been a better choice. In reality, the `SealedObject` class is not even in the `java.security` package. Instead, it is included in the `javax.crypto` package, which was initially part of the Java Cryptography Extension (JCE) 1.2 and has been incorporated into J2SE 1.4. This design choice was influenced solely by U.S. regulations regarding the export of encryption software.[4]

[2] For those who are interested in researching the history of secure objects, earlier work on secure network objects using Modula-3 and Oblique [129] is related to `SignedObject` and `SealedObject` in that there was the high-level abstraction of secure remote object invocation. However, this abstraction was implemented by establishing a secure communication channel between the two end points and using this channel to send the plain object and data. In other words, there was no explicit concept of signing and sealing objects directly.

[3] Experience in security system design indicates that blindly signing encrypted data is sometimes dangerous. Thus, you should create and sign a `SignedObject` first and then use that `SignedObject` to create a `SealedObject`.

[4] In fact, when JCE was initially designed, the U.S. Commerce Department required encryption software to obtain the same kind of export license as that issued for munitions.

Given any `Serializable` object, a `SealedObject` that embeds in its content the original object, in serialized format, can be created. Then, a cryptographic algorithm, such as Blowfish, is applied to the content to protect its confidentiality. The encrypted content can later be decrypted by using the corresponding algorithm with the correct decryption key.

After decryption, the original content can be obtained in object form through deserialization. While encrypted, the content is not available to anyone who does not possess the correct decryption key, assuming that the cryptosystem is secure.

The `SealedObject` constructor and public methods are as follows, with exception declarations left out:

```
public SealedObject(Serializable object, Cipher c);
public final String getAlgorithm();
public final Object getObject(Cipher c);
public final Object getObject(Key k)
public final Object getObject(Key k, String provider);
```

A typical use of this class is illustrated with the following code segments; first, a DES key is generated and the DES cipher initialized:

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES");
SecretKey desKey = keyGen.generateKey();
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, desKey);
```

Next, a `SealedObject` is created and encrypted. Note that the `Cipher` object must be fully initialized with the correct algorithm, key, padding scheme, and so on, before being applied to a `SealedObject`:

```
String s = new String("Greetings");
SealedObject so = new SealedObject(s, cipher);
```

Later, the sealed object can be decrypted and the original object retrieved:

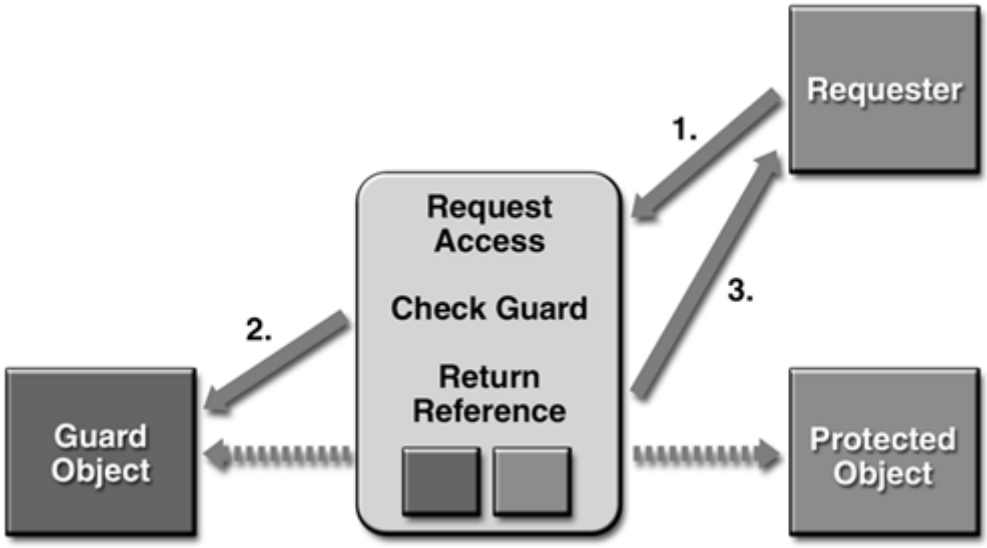
```
cipher.init(Cipher.DECRYPT_MODE, desKey);
try {
   String s = (String) so.getObject(cipher);
} catch (ClassNotFoundException e) {};
```

As is the case with `SignedObject`, `SealedObject` may be subclassed to provide better static typing information.

9.11 Guarding Objects

Apart from `SignedObject`, which provides object authenticity, J2SDK 1.2 introduced an interface, `java.security.Guard`, and a class, `java.security.GuardedObject`, that may be used for object-level access control. A `GuardedObject` is used to protect access to another object. A `GuardedObject` encapsulates a target object and a `Guard` object. Once a target object is encapsulated by a `GuardedObject`, access to that object is controlled by the `getObject` method. This method invokes the `checkGuard` method on the `java.security.Guard` object that is guarding access. If access is allowed, `checkGuard` returns silently; if access is not allowed, it throws a `SecurityException`. A `GuardedObject` and protection of access to the target object via the `Guard` object is illustrated in Figure 9.2, where solid lines represent method calls and dotted lines represent object references. Here, when a requester asks for an object that is guarded by a `GuardedObject` with a particular `Guard`, first the `Guard` is consulted, and then the reference to the desired object is returned to the requester, if the `Guard` allows it.

Figure 9.2. `Guard` and `GuardedObject`



One major motivation for having the `GuardedObject` class is that often the supplier of a resource is not in the same execution context, such as a thread, as the consumer of that resource. In this case, a security check within the security context of the supplier is often inappropriate because the check should occur within the security context of the consumer.

For example, when a file server thread responds to a request to open a file for reading and this request comes from a different environment, the decision to supply the file must take into account information about the requester, such as its `AccessControlContext`. (See Chapter 6.) Sometimes, however, the consumer cannot provide the supplier with such information, for several reasons.

- The consumer program does not always know ahead of time what information should be provided—quite possible in a dynamically changing environment—and it is undesirable—for example, for performance reasons—to engage in a dialogue or negotiation for each request.
- The consumer regards information about its execution environment as being too security sensitive to pass on to another party.
- There is too much information or data to pass on.
- Information about the execution environment of the consumer cannot be interpreted by the supplier.

To make access control in these situations more uniform and easier to program, `GuardedObject` was designed so that the supplier of the resource can create an object representing the resource and a `GuardedObject` containing the resource object and then provide the `GuardedObject` to the consumer. In creating the `GuardedObject`, the supplier also specifies a `Guard` object such that anyone, including the consumer, can obtain the resource object only when certain checks, such as security checks, inside the `Guard` are satisfied. `Guard` is an interface, so any object can choose to become a `Guard`.

Using `GuardedObject` has several benefits:

- You can correctly embed the protection mechanism together with the protected object so that access to the object is guaranteed to occur in a context in which the protection mechanism will allow it.
- You can delay an access control decision from time of request to time of actual access, thus simplifying server programs.

- You can replace often used access control lists with object stores and simply store a set of `GuardedObject`s.
- The designer of a class does not need to specify the class's protection semantics, as long as any object instantiated from this class is protected within a `GuardedObject` and the associated `Guard` object implements the correct security checks.
- The same programming pattern can be used to encapsulate an object's protection mechanisms, which can differ for the object's different method invocations, all inside a `Guard`.

Note that because the built-in base class `java.security.Permission` implements the `Guard` interface, all permissions of this type, including all permissions (on file, network, runtime, and other resources) defined in the SDK, are instantly usable as `Guard` objects.

The `Guard` interface contains only one method:

```
void checkGuard(Object object);
```

Following is the signature of the `GuardedObject` class.

```
public GuardedObject(Object object, Guard guard);
public Object getObject();
```

The following example uses `GuardedObject` to encapsulate an object's protection semantics completely inside an appropriate `Guard` object. Note that this is just an example. There is no plan to massively change such classes in the SDK to use `GuardedObject`.

An instance of a `java.io.FileInputStream` is used as an example of an object to be guarded, and a `java.io.FilePermission` is used as the `Guard`. For this example, a file input stream is created with a given file name, as follows:

```
FileInputStream fis = new FileInputStream("/a/b/c");
```

Normally, the implementation of this constructor must be aware that a security check needs to be done to ensure that permission to read the specified file has been granted, must understand what sort of check is appropriate, and must populate all constructors with the same or similar checks.

Such checks are included in the current `FileInputStream` implementation but are not needed within the constructors if access to a `FileInputStream` is instead controlled by a `Guard`. First, note that a `FilePermission` can be used as a `Guard` object, as its superclass, `java.security.Permission`, is a `Guard` object by virtue of implementing `Guard` and having a `checkGuard` method, defined as follows:

```
public abstract Permission implements Guard {
   . . .
   public void checkGuard(Object object) throws
SecurityException {
      SecurityManager sm = System.getSecurityManager();
      if (sm != null) sm.checkPermission(this);
   }
   . . .
}
```

This implementation ensures that a proper access control check takes place within the consumer context, when access to the stream is first requested.

Now the provider side of the code can simply be as follows:

```
FileInputStream fis = new FileInputStream("/a/b/c");
FilePermission p = new FilePermission("/a/b/c", "read");
GuardedObject g = new GuardedObject(fis, p);
```

After `GuardedObject g` is passed to the consumer, the following code will retrieve the `FileInputStream` but only if the consumer is permitted to obtain read access to the file `"/a/b/c"`:

```
FileInputStream fis = (FileInputStream) g.getObject();
```

Note that in this case, the implementation of `FileInputStream` itself need not be security aware, as long as it is always protected by a `GuardedObject`.

This design does not further perform security checks once a `FileInputStream` is returned to the consumer. This is the same behavior implemented in the `FileInputStream` class today. That is, the `FileInputStream` implementation does not itself assume that the `GuardedObject` approach is used; instead, its constructors have the appropriate permission checks. But once it returns a `FileInputStream` to the consumer, no further security checks are done.

Another potential application of `GuardedObject` is in the implementation of deferred object requests in the Java IDL (Interface Definition Language) or a similar product. The obvious implementation of this CORBA (Common Object Request Broker Architecture)-style API is to spin a separate thread in the ORB implementation to make the (deferred) request. This new thread is created by the ORB implementation, so any information about what code originated the request is lost, thereby making security checking difficult, if not impossible. With `GuardedObject`, the new thread can simply return a properly guarded object. This forces a security check to occur when the requester attempts to retrieve the object.

`Guard` and `GuardedObject` can be extended, or subclassed, to implement arbitrary guarding semantics. In fact, the guard concept in `GuardedObject` is similar to the well-known guard concept in programming language research. It has been used elsewhere, albeit mostly in specialized forms, for example as a pattern [35]. Its combination with `java.security.Permission` is a novel feature that makes `Guard` very powerful for access control on the Java platform.

As another example, we can, hypothetically, radically rewrite the `FileInputStream` class as follows. For every constructor that does not take a `Guard` object `g` as the parameter, a suitable `Guard` is automatically generated. For every access method, such as `read(bytes)`, the uniform security check in the form of `g.checkGuard()` is invoked first.

As with `SignedObject`, subclassing `GuardedObject` can better preserve static typing information, where the base classes are intended to be used between cooperating parties so that the receiving party should know what type of object to expect.

Chapter 10. Programming Cryptography

The secret of my influence has always been that it remained a secret.

—Salvador Dali

Earlier chapters briefly covered some of the basic concepts of cryptography, as well as code signing and the use of certificates, which depend on public-key cryptosystems. This chapter goes behind the scenes to look at the Java Cryptography Architecture (JCA), which underlies the APIs and the tools.

The JCA first appeared in JDK 1.1. It had fairly limited functionalities that included APIs for digital signatures and message digests. The Java 2 SDK significantly extended the JCA into a framework for accessing and developing cryptographic functionality for the Java platform. Loosely speaking, JCA encompasses the parts of the Java 2 SDK Security API that are related to cryptography. The JCA also includes a provider architecture that allows for multiple and interoperable cryptography implementations, as well as a set of associated conventions and specifications.

When the Java 2 SDK was first released, the JCA logically covered both the crypto APIs defined in the SDK and those defined in the Java Cryptography Extension (JCE) 1.2, an optional package that provided APIs for encryption, key exchange, MAC (message authentication code), and a number of other encryption-related features. Thus, the SDK and JCE together provided a comprehensive set of platform-independent cryptography APIs. JCE was released separately as an extension to the Java 2 SDK versions 1.2.*x* and 1.3. *x*, in accordance with U.S. regulations concerning the export of cryptography. As the regulations were subsequently modified, JCE was integrated into the Java 2 SDK 1.4.

This chapter covers architectural issues and classes that span the full JCA. First, the chapter defines cryptography terms and then presents the JCA design principles, including the notion of Cryptographic Service Providers. The cryptography classes are described in detail in Sections 10.4 (the core classes) and 10.5 (the additional JCE cryptography classes now integrated into the J2SDK). Section 10.6 provides several code examples illustrating use of the cryptography classes. Section 10.7 specifies the "standard names" to be used to refer to algorithms, types, and so on. Finally, Section 10.8 provides specifications for the various algorithms.

10.1 Cryptographic Concepts

This section provides a high-level description of the cryptographic terms used in the API specifications and this chapter.

A *digital signature* algorithm takes arbitrary-sized input and a private key and generates a relatively short, often fixed-size, string of bytes, called the *signature*, with the following two properties:

1. When the public key corresponding to the private key used to generate the signature is provided, it should be possible to verify the authenticity and integrity of the input.
2. The signature and the public key do not reveal anything about the private key.

A cryptographically secure *message digest* takes arbitrary-sized input—a `byte` array—and generates a fixed-size output, called a *digest*, or *hash*. It should be computationally infeasible to find two messages that hash to the same value, and the digest should not reveal anything about the input that was used to generate it. Thus, message digests are sometimes called the "digital fingerprints" of data.

Encryption is the process of taking data, called *cleartext*, and a short string—a key—and producing data, or *ciphertext*, meaningless to a third party who does not know the key. *Decryption* is the inverse process: taking ciphertext and a short key string and producing cleartext.

Password-based encryption (PBE) derives an encryption key from a password. In order to make the task of getting from password to key very time-consuming for an attacker, most PBE implementations mix in a random number, known as a *salt*, to create the key.

Encryption and decryption are done using a *cipher*. A cipher is an object capable of carrying out encryption and decryption according to an encryption scheme, or algorithm.

Key agreement is a protocol by which two or more parties can establish the same cryptographic keys. With such a protocol, the parties do not have to exchange any secret information.

A *message authentication code* (MAC) provides a way to check the integrity of information transmitted over or stored in an unreliable medium, based on a secret key. Typically, message authentication codes are used between two parties that share a secret key, in order to validate information transmitted between them.

A MAC mechanism that is based on cryptographic hash functions is referred to as HMAC. HMAC can be used with any cryptographic hash function, such as MD5 or SHA-1, in combination with a secret shared key. HMAC is specified in RFC 2104 [67].

10.2 Design Principles

The design of the Java Cryptography Architecture is guided by two principles: (1) algorithm independence and extensibility and (2) implementation independence and interoperability. The aim of the JCA is to let API users use cryptographic *services*, such as digital signatures and message digests, without concern for the implementations or even the algorithms being used to implement these services. At the same time, the JCA provides standardized APIs so that developers can request specific algorithms and specific implementations, if desired.

Algorithm independence is achieved by defining types of cryptographic "engines," or services, and defining classes that provide the functionality of these cryptographic engines. These classes are called *engine classes*; examples are the `MessageDigest`, `Signature`, `KeyFactory`, and `Cipher` classes.

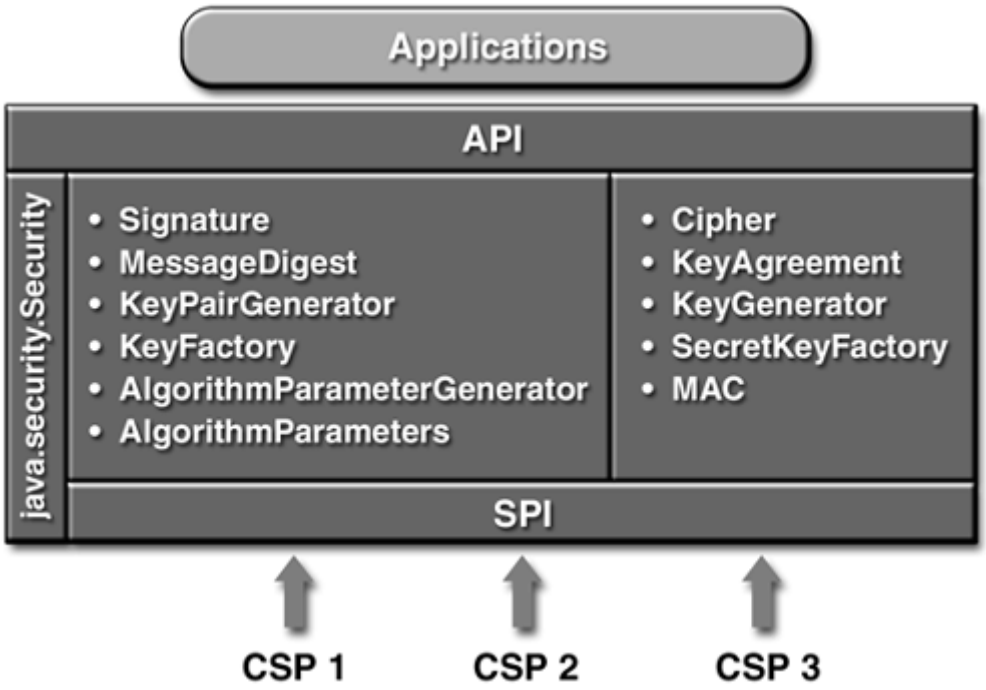
Implementation independence is achieved by using a *provider-based architecture*. A provider in the context of JCA means a *Cryptographic Service Provider* (CSP), or simply a *provider*, which is a package or set of packages that implement one or more JCA cryptographic services, such as digital signature algorithms, message digest algorithms, and key-conversion services. A program may simply request a particular type of object, such as a `Signature` object, that implements a particular service, such as the DSA signature algorithm, and receive an implementation from one of the installed providers. If desired, a program may instead request an implementation from a specific provider. Providers may be updated transparently to the application when, for example, faster or more secure versions are available.

Given the general nature of the API design, implementation interoperability is obtained in the sense that even though various implementations might have different characteristics, they can work with one another, such as using one another's keys or verifying one another's signatures. For example, when the appropriate key factory implementations are installed, a key generated by one provider would be usable by another for the same algorithm, and a signature generated by one provider would be verifiable by another. This would apply even though one provider might be implemented in software, while the other is implemented in hardware, and one might be platform

independent, while the other is platform specific. The interface design is also extensible in that new algorithms that fit in one of the supported engine classes can easily be added.

Figure 10.1 depicts the architectural outline of JCA. By following the two design principles given earlier, JCA brings major benefits to the cryptographic software market. On the one hand, application software developers have only one set of APIs (JCA) to worry about, no matter what algorithms they choose to use or what provider packages they install. On the other hand, crypto toolkit or library vendors can compete with one another in intellectual property—for example, patented algorithms and techniques—and performance optimization while maintaining full interoperability with one another at the level of JCA APIs.

Figure 10.1. JCA architecture



10.3 Cryptographic Services and Service Providers

As previously noted, the JCA incorporates the notion of a CSP, or provider, which is a package or a set of packages that supplies a concrete implementation of one or more cryptographic services. In JDK 1.1, a provider could supply an implementation of one or more digital signature algorithms, message digest algorithms, and key-generation algorithms. The following types of services were added in the Java 2 SDK 1.2:

- Key factories
- Keystore creation and key management
- Algorithm parameter management
- Algorithm parameter generation
- Certificate factories
- Random-number generation

JCE 1.2 was released separately as an extension to the Java 2 SDK 1.2.*x* and 1.3.*x*, in accordance with U.S. regulations concerning the export of cryptography, and it provided APIs for

- Encryption
- Key generation

- Key agreement
- MAC (message authentication code)

The regulations were subsequently modified, and JCE was integrated into the Java 2 SDK 1.4. In that release, the following types of services were added:

- Certification chain (path) builders
- Certification path validators
- Certificate stores for retrieving certificates

Details about most of the services that may be implemented are provided later in this chapter. The certification path and certificate store services are described in Section 8.2.

Each SDK installation typically has one or more provider packages installed, and users may add new providers statically or dynamically. Each provider is referred to by a unique name. Users may configure their runtimes with different providers and specify a preference order for each. JCA offers a set of APIs that allows users to query which providers are installed and what services they support. If the application requests a specific provider, only objects from that provider are returned. If no specific provider is given, a default provider is chosen. When multiple providers are available, a preference order is specified. This is the order in which providers are searched for requested services. When a requested service is not provided by the most preferred provider, the next provider in the preference order is examined, and so on.

For example, suppose that you have two providers installed in your JVM: Provider1 and Provider2. Further suppose that Provider1 implements the SHA1withDSA and MD5 algorithms, whereas Provider2 implements SHA1withDSA, MD5withRSA, and MD5. If Provider1 has preference order 1—the highest priority—and Provider2 has preference order 2 and you don't specify a particular provider when requesting an algorithm, the following behavior will occur.

- If you are looking for an MD5 implementation and both providers supply such an implementation, the Provider1 implementation is returned because it has the highest priority and thus is searched first.
- If you are looking for an MD5withRSA signature algorithm, Provider1 is searched first. No implementation is found, so Provider2 is searched. An implementation is found there and returned.
- If you are looking for a SHA1withRSA signature algorithm, neither installed provider implements it, so a `java.security.NoSuchAlgorithmException` is thrown.

An engine class defines a cryptographic service in an abstract fashion without a concrete implementation. A cryptographic service is always associated with a particular algorithm or type and does one of the following:

- Provides cryptographic operations, such as those for digital signatures and message digests
- Generates or supplies the cryptographic material—keys or parameters—required for cryptographic operations
- Generates data objects—keystores or certificates—that encapsulate cryptographic keys, which can be used in a cryptographic operation, in a secure fashion

For example, two of the engine classes are `Signature` and `KeyFactory`. The `Signature` class provides access to the functionality of a digital signature algorithm. A DSA `KeyFactory` supplies a DSA private or public key from its encoding or transparent specification in a format usable by the `initSign` or `initVerify` methods, respectively, of a DSA `Signature` object.

Programmers can request and use instances of the engine classes to carry out corresponding operations. The engine classes defined in the Java 2 SDK 1.4 are listed in Table 10.1.

Table 10.1. Java 2 SDK 1.4 Engine Classes

Engine Class	Used to	See Section
`MessageDigest`	Calculate the message digest (hash) of specified data.	10.4.3
`Signature`	Sign data and verify digital signatures.	10.4.4
`AlgorithmParameters`	Manage the parameters for a particular algorithm, including parameter encoding and decoding.	10.4.5
`AlgorithmParameterGenerator`	Generate a set of parameters suitable for a specified algorithm.	10.4.5
`KeyPairGenerator`	Generate a pair of public and private keys suitable for a specified algorithm.	10.4.6 and 10.4.8
`KeyFactory`	Convert opaque keys of type `Key` into key specifications—transparent representations of the underlying key material—and vice versa.	10.4.6 and 10.4.7
`CertificateFactory`	Create public-key certificates and CRLs.	10.4.7
`KeyStore`	Create and manage a keystore, a database of keys and certificates.	10.4.9
`SecureRandom`	Generate random or pseudo-random numbers.	10.4.10
`Cipher`	Encrypt and decrypt data.	10.5.1
`KeyGenerator`	Generate secret keys for symmetric algorithms.	10.5.2
`SecretKeyFactory`	Convert opaque keys of type `javax.crypto.SecretKey` into key specifications and vice versa.	10.5.3
`KeyAgreement`	Provide the functionality of a key-agreement protocol.	10.5.4
`Mac`	Provide the functionality of a message authentication code (MAC).	10.5.5
`CertPathValidator`	Validate certification paths.	8.2.3
`CertPathBuilder`	Build certificate chains, or certification paths.	8.2.4
`CertStore`	Retrieve certificates and CRLs from a repository.	8.2.5

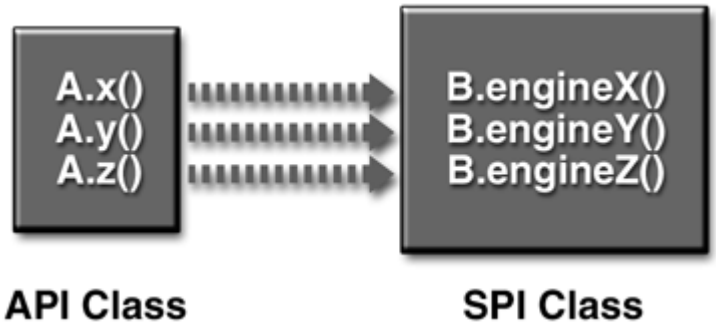
A generator and a factory differ within the JCA context. A generator creates objects with new contents, whereas a factory creates objects from existing material—for example, an encoding.

An engine class provides the interface to the functionality of a specific type of cryptographic service, independent of a particular cryptographic algorithm. An engine class defines APIs that allow applications to access the specific type of cryptographic service it provides. The implementations, from one or more providers, are those for specific algorithms. The `Signature` engine class, for example, provides access to the functionality of a digital signature algorithm. The implementation supplied in a `SignatureSpi` subclass would be that for a specific kind of signature algorithm, such as SHA1withDSA, SHA1withRSA, or MD5withRSA.

The application interfaces supplied by an engine class are implemented in terms of a *Service Provider Interface* (*SPI*). That is, for each engine class, there is a corresponding abstract SPI class that defines the SPI methods that cryptographic service providers must implement.

An instance—the API object—of an engine class encapsulates, as a private field, an instance—the SPI object—of the corresponding SPI class. All API methods of an API object are declared `final`, and their implementations invoke the corresponding SPI methods of the encapsulated SPI object. An instance of an engine class and of its corresponding SPI class is created by a call to a `getInstance` factory method of the engine class. Figure 10.2 depicts an API object and its corresponding SPI object.

Figure 10.2. API class and corresponding SPI class



The name of each SPI class is the same as that of the corresponding engine class, followed by `Spi`. For example, the SPI class corresponding to the `Signature` engine class is the `SignatureSpi` class. Each SPI class is abstract. To supply the implementation of a particular type of service, for a specific algorithm, a provider must subclass the corresponding SPI class and provide implementations for all the abstract methods.

Another example of an engine class is the `MessageDigest` class, which provides access to a message digest algorithm. Its implementations, in `MessageDigestSpi` subclasses, may be those of various message digest algorithms, such as SHA-1, MD5, or MD2.

As a final example, the `KeyFactory` engine class supports the conversion from opaque keys to transparent key specifications, and vice versa. The implementation supplied in a `KeyFactorySpi` subclass is for a specific type of key: for example, DSA public and private keys.

Implementations for various cryptography services are provided by JCA Cryptographic Service Providers, such as the SUN and SunJCE providers. Other providers may define their own implementations of those services or of other services not implemented by these providers, such as one of the RSA-based signature algorithms or the MD2 message digest algorithm. Section 12.5

explains how to install and register provider packages so that they are available for use by your programs.

The Sun Microsystems version of the Java runtime environment comes standard with a default provider, named SUN. Other Java runtime environments might not necessarily supply the SUN provider. The SUN provider package includes implementations of the following:

- The Digital Signature Algorithm (DSA) [91]
- The MD5 [102] and SHA-1 [93] message digest algorithms
- A DSA key-pair generator for generating a pair of public and private keys suitable for the DSA algorithm
- A DSA algorithm parameter generator
- A DSA algorithm parameter manager
- A DSA key factory providing bidirectional conversions between opaque DSA private- and public-key objects and their underlying key material
- A proprietary SHA1PRNG pseudo-random-number generation algorithm, following the recommendations in the IEEE P1363 standard
- A certificate factory for X.509 certificates and CRLs
- A certificate path builder and validator for PKIX, as defined in the Internet X.509 Public Key Infrastructure Certificate and CRL Profile [54]
- A certificate store implementation for retrieving certificates and CRLs from Collection and LDAP [16] directories
- A keystore for the proprietary keystore type named JKS

The Sun Microsystems version of the Java runtime environment also comes standard with a provider named SunJCE. This provider package supplies the following cryptographic services:

- An implementation of the DES [128], Triple DES, and Blowfish encryption algorithms in the Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Propagating Cipher Block Chaining (PCBC) modes (*Note:* Throughout this chapter, the terms Triple DES and DES-EDE are used interchangeably.)
- Key generators for generating keys suitable for the DES, Triple DES, Blowfish, HMAC-MD5, and HMAC-SHA1 algorithms
- An implementation of the MD5 with DES-CBC password-based encryption (PBE) algorithm defined in PKCS #5 [106]
- "Secret-key factories" providing bidirectional conversions between opaque DES, Triple DES, and PBE key objects and transparent representations of their underlying key material
- An implementation of the Diffie-Hellman key-agreement algorithm between two or more parties
- A Diffie-Hellman key-pair generator for generating a pair of public and private values suitable for the Diffie-Hellman algorithm
- A Diffie-Hellman algorithm parameter generator
- A Diffie-Hellman "key factory" providing bidirectional conversions between opaque Diffie-Hellman key objects and transparent representations of their underlying key material
- Algorithm parameter managers for Diffie-Hellman, DES, Triple DES, Blowfish, and PBE parameters
- An implementation of the HMAC-MD5 and HMAC-SHA1 keyed-hashing algorithms defined in RFC 2104 [67]
- An implementation of the padding scheme described in PKCS #5
- A keystore implementation for the proprietary keystore type named JCEKS

The lists of services supplied by the SUN and SunJCE providers will continue to expand. Consult the latest online Java Cryptography Architecture and Java Cryptography Extension documentation

to get up-to-date lists. This is available at `http://java.sun.com/j2se/m.n/docs/guide/security/`, where *m.n* refers to the release number. For example, the security documentation for the 1.4 release is at `http://java.sun.com/j2se/1.4/docs/guide/security/`.

10.4 Core Cryptography Classes

This section describes the design and usage of classes central to the Java Cryptography Architecture. Section 10.5 describes additional classes in Java 2 SDK 1.4 that were initially in the Java Cryptography Extension optional package.

10.4.1 `Security`

The `java.security.Security` class manages installed providers and security-wide properties. It contains only static methods and is never instantiated:

```
public static Provider[] getProviders()
public static Provider getProvider(String name)
public static int addProvider(Provider provider)
public static int insertProviderAt(Provider provider,
                                   int position)
public static void removeProvider(String name)
public static String getProperty(String key)
public static void setProperty(String key, String datum)
```

The `getProviders` method returns an array containing all the installed providers: technically, the `Provider` subclass for each package provider. The order of the providers in the array is their preference order, which is the order in which providers are searched for requested algorithms if no specific provider is requested. The `getProvider` method returns the `Provider` of the specified name. The `addProvider` method adds a provider to the end of the list of installed providers. It returns either the preference position at which the provider was added or -1, if the provider was not added because it was already installed.

The `insertProviderAt` method attempts to add a new provider at a specified position in the preference order. A provider cannot be added again if it is already installed. If the given provider gets installed at the requested position, the provider that used to be at that position, as well as all providers with a position greater than that position, are shifted down, toward the end of the list of installed providers. This method returns the preference position at which the provider was added or -1 if the provider was not added because it was already installed.

The `removeProvider` method removes the named provider. It returns silently if the `Provider` is not installed. When the specified provider is removed, all providers located at a position greater than where the specified provider was are shifted up one position, toward the head of the list of installed providers. To change the preference position of an installed provider, you must first remove it and then reinsert it at the new preference position.

The `Security` class maintains a list of systemwide security properties. These properties are accessible and settable via the `getProperty` and `setProperty` methods, respectively.

10.4.2 `Provider`

The term "Cryptographic Service Provider" (used interchangeably with "provider" in this book) refers to a package or set of packages that supplies a concrete implementation of a subset of the

cryptography aspects of the Java 2 SDK Security API. The `java.security.Provider` class is the interface to such a package or set of packages.

Each `Provider` class instance has a name, a version number, and a string description of the provider and its services. You can query the `Provider` instance for this information by calling the following methods:

```
public String getName()
public double getVersion()
public String getInfo()
```

Note that in addition to registering implementations of cryptographic services, the `Provider` class can also be used to register implementations of other security services that might get defined as part of the Java 2 SDK Security API.

To supply implementations of cryptographic or other services, an entity, such as a development group, writes the implementation code and creates a subclass of the `Provider` class. The constructor of the subclass sets the values of various properties that are required for the Java 2 SDK Security API to look up the services implemented by the provider. That is, the constructor specifies the fully qualified names of the classes implementing the services.

10.4.3 `MessageDigest`

The `java.security.MessageDigest` class is an engine class designed to provide the functionality of cryptographically secure message digests, such as SHA-1 or MD5. To compute a digest, you first create a `MessageDigest` instance. As with all engine classes, a `MessageDigest` object for a particular type of message digest algorithm is obtained by calling a `getInstance` static factory method on the `MessageDigest` class, such as the following:

```
public static MessageDigest getInstance(String algorithm)
```

The algorithm name is case insensitive. For example, all the following calls are equivalent:

```
MessageDigest.getInstance("SHA-1")
MessageDigest.getInstance("sha-1")
MessageDigest.getInstance("sHa-1")
```

A caller may optionally specify the currently case-sensitive name of a provider, or a `Provider` instance, which will guarantee that the implementation of the algorithm requested is from the specified provider:

```
public static MessageDigest getInstance(String algorithm,
                                        String provider)
public static MessageDigest getInstance(String algorithm,
                                        Provider provider)
```

A call to one of the `getInstance` methods returns an initialized `MessageDigest` object. Thus, it does not need further initialization.

Next, to calculate the digest of some data, you supply the data to the initialized message digest object. This is done by making one or more calls to one of the `update` methods:

```
public void update(byte input)
public void update(byte[] input)
public void update(byte[] input, int offset, int len)
```

After the data has been supplied by calls to `update` methods, the digest is computed, using a call to one of the `digest` methods:

```
public byte[] digest()
public byte[] digest(byte[] input)
public int digest(byte[] buf, int offset, int len)
```

The first two methods return the computed digest. The third stores the computed digest in the provided buffer `buf`, starting at `offset`. The `len` parameter tells the number of bytes in `buf` allotted for the digest. The method returns the number of bytes stored in `buf`. A call to the `digest` method that takes just an `input byte` array argument is equivalent to making a call to `public void update(byte[] input)` with the specified input, followed by a call to the `digest` method without any arguments.

Examples of computing message digests are shown in Section 10.6.1.

10.4.4 `Signature`

The `java.security.Signature` engine class is designed to provide the functionality of a cryptographic digital signature algorithm, such as SHA1withDSA or MD5withRSA. A `Signature` object can be used to generate a signature for data, and can also be used to verify whether an alleged signature is in fact the authentic signature of the data associated with it. `Signature` objects are modal objects. That is, a `Signature` object is always in a given state in which it may do only one type of operation.

Signature states are represented as final integer constants defined in the `Signature` class. A `Signature` object may have three states:

1. `UNINITIALIZED`
2. `SIGN`
3. `VERIFY`

To sign data or verify a signature, you first create a `Signature` instance. As with all engine classes, a `Signature` object for a particular type of signature algorithm is obtained by calling one of the `getInstance` static factory methods on the `Signature` class:

```
public static Signature getInstance(String algorithm)
public static Signature getInstance(String algorithm,
                                    String provider)
public static Signature getInstance(String algorithm,
                                    Provider provider)
```

A `Signature` object must be initialized before it can be used. When it is created, a `Signature` object is in the `UNINITIALIZED` state. The initialization method to be called depends on whether the object is going to be used for signing or for verification. If for signing, the object must first be initialized with the private key of the entity whose signature is going to be generated. This initialization is done by calling the `initSign` method:

```
public final void initSign(PrivateKey privateKey)
```

This method puts the `Signature` object in the `SIGN` state.

If the `Signature` object is going to be used for verification, it must be initialized with the public key of the entity whose signature is going to be verified. This initialization is done by calling one of the `initVerify` methods:

```
public final void initVerify(PublicKey publicKey)
public final void initVerify(Certificate certificate)
```

A call to an `initVerify` method puts the `Signature` object in the `VERIFY` state.

If the `Signature` object has been initialized for signing—if it is in the `SIGN` state—the data to be signed can then be supplied to the object. This is done by making one or more calls to one or more of the `update` methods:

```
public final void update(byte b)
public final void update(byte[] data)
public final void update(byte[] data, int off, int len)
```

Calls to the `update` method(s) should be made until all the data to be signed has been supplied to the `Signature` object.

To generate the signature, simply call one of the `sign` methods:

```
public final byte[] sign()
public final int sign(byte[] outbuf, int offset, int len)
```

The first method returns the signature result in a `byte` array. The second stores the signature result in the provided buffer `outbuf`, starting at `offset`. The `len` parameter is the number of bytes in `outbuf` allotted for the signature. The method returns the number of bytes stored. The signature encoding is algorithm specific. For example, a SHA1withDSA signature is encoded as a standard ASN.1 sequence of two integers: *r* and *s*.

When a `sign` method is called, it generates the signature and then resets the `Signature` object to the state it was in when previously initialized for signing, via a call to `initSign`. That is, the object is reset and available to generate another signature with the same private key, if desired, via new calls to `update` and `sign`. Alternatively, a new call can be made to `initSign`, specifying a different private key, or to `initVerify` to initialize the `Signature` object to verify a signature.

If the `Signature` object has been initialized for verification—it is in the `VERIFY` state—it can then verify whether an alleged signature is in fact the authentic signature of the data associated with it. The process begins by supplying the data to be verified, as opposed to the signature itself, to the object. This is done by making one or more calls to one or more of the `update` methods:

```
public final void update(byte b)
public final void update(byte[] data)
public final void update(byte[] data, int off, int len)
```

Calls to the `update` method(s) should be made until all the data has been supplied to the `Signature` object.

The signature can then be verified by calling one of the `verify` methods:

```
public final boolean verify(byte[] signature)
public final boolean verify(byte[] signature, int offset,
                            int length)
```

The first argument must be a byte array containing the signature. The `verify` method returns a `boolean` indicating whether the signature is the authentic signature of the data supplied to the `update` method(s).

When the `verify` method is called, it performs the verification and then resets the `Signature` object to the state it was in when previously initialized for verification, via a call to `initVerify`. That is, the object is reset and available to verify another signature from the identity whose public key was specified in the call to `initVerify`. Alternatively, a new call can be made either to `initVerify`, specifying a different public key to initialize the `Signature` object for verifying a signature from a different entity, or to `initSign`, to initialize the `Signature` object for generating a signature.

Examples of generating and verifying signatures are shown in Section 10.6.3.

10.4.5 Algorithm Parameters

JCA is designed to handle many crypto algorithms. These algorithms can be very different. In particular, each tends to have unique requirements with regard to various parameters, such as key size and defined constants. To organize these parameters, an algorithm parameter specification is defined for each algorithm, and all such specifications are divided into a small set of classes.

An algorithm parameter specification is a *transparent* representation of the sets of parameters used with an algorithm. This means that you can access each parameter value in the set individually, through one of the `get` methods defined in the corresponding specification class. For example, `DSAParameterSpec` defines `getP`, `getQ`, and `getG` methods, which access the *p*, *q*, and *g* parameter values, respectively. In an *opaque* representation, by contrast, as supplied by the `AlgorithmParameters` class, you have no direct access to the parameter fields. Rather, you can get only the name of the algorithm associated with the parameter set, via `getAlgorithm`, and some kind of encoding for the parameter set, via `getEncoded`. You can call the `AlgorithmParameters.getParameterSpec` method to convert an `AlgorithmParameters` object to a transparent specification. Examples of classes that may be initialized with `AlgorithmParameters` or `AlgorithmParameterSpec` objects are the `KeyPairGenerator` (Section 10.4.8) and `Cipher` (Section 10.5.1) classes.

The algorithm parameter and algorithm parameter specification interfaces and classes in the `java.security` and `java.security.spec` packages are

- `java.security.spec.AlgorithmParameterSpec`
- `java.security.spec.DSAParameterSpec`
- `java.security.AlgorithmParameters`
- `java.security.AlgorithmParameterGenerator`

`AlgorithmParameterSpec`

This interface is the base interface for the transparent specification of cryptographic parameters. It contains no methods or constants. Its only purpose is to group and provide type safety for all parameter specifications. All parameter specifications must implement this interface.

DSAParameterSpec

This class, which implements the `AlgorithmParameterSpec` interface, specifies the set of parameters used with the DSA algorithm. It has the following methods:

```
public BigInteger getP()
public BigInteger getQ()
public BigInteger getG()
```

These methods return the DSA algorithm parameters: the prime *p*, the subprime *q*, and the base *g*.

AlgorithmParameters

This engine class provides an opaque representation of cryptographic parameters. As with all engine classes, an `AlgorithmParameters` object for a particular type of algorithm is obtained by calling one of the `getInstance` static factory methods on the `AlgorithmParameters` class:

```
public static AlgorithmParameters getInstance(String
algorithm)
public static AlgorithmParameters getInstance(String
algorithm,
                                              String provider)
public static AlgorithmParameters getInstance(String
algorithm,
                                              Provider provider)
```

Once an `AlgorithmParameters` object is instantiated, it must be initialized via a call to `init`, using an appropriate parameter specification or parameter encoding:

```
public void init(AlgorithmParameterSpec paramSpec)
public void init(byte[] params)
public void init(byte[] params, String format)
```

In the preceding, `params` is an array containing the encoded parameters, and `format` is the name of the decoding format. In the `init` method with a `params` argument but no format argument, the primary decoding format for parameters is used. The primary decoding format is ASN.1, if an ASN.1 specification for the parameters exists. Note that `AlgorithmParameters` objects may be initialized only once and thus are not meant for reuse.

A byte encoding of the parameters represented in an `AlgorithmParameters` object may be obtained via a call to the `getEncoded` method:

```
public byte[] getEncoded()
```

This method returns the parameters in their primary encoding format.

To have the parameters returned in a specified encoding format, use this `getEncoded` method:

```
public byte[] getEncoded(String format)
```

If `format` is `null`, the primary encoding format for parameters is used, as in the other `getEncoded` method.[1]

[1] In the default `AlgorithmParameters` implementation, supplied by the SUN provider, the format argument is currently ignored.

A transparent parameter specification for the algorithm parameters may be obtained from an `AlgorithmParameters` object via a call to the `getParameterSpec` method:

```
public AlgorithmParameterSpec getParameterSpec(Class
paramSpec)
```

The `paramSpec` parameter identifies the specification class in which the parameters should be returned. That class could be, for example, `DSAParameterSpec.class` to indicate that the parameters should be returned in an instance of `DSAParameterSpec`, which is in the `java.security.spec` package and implements the `AlgorithmParameterSpec` interface.

AlgorithmParameterGenerator

This engine class generates a set of parameters suitable for the algorithm that is specified when an `AlgorithmParameterGenerator` instance is created. To get an `AlgorithmParameterGenerator` instance for a particular type of algorithm, call one of the `getInstance` static factory methods on the `AlgorithmParameterGenerator` class:

```
public static AlgorithmParameterGenerator
      getInstance(String algorithm)
public static AlgorithmParameterGenerator
      getInstance(String algorithm, String provider)
public static AlgorithmParameterGenerator
      getInstance(String algorithm, Provider provider)
```

The `AlgorithmParameterGenerator` object can be initialized in either of two ways:

- Algorithm independent
- Algorithm specific

The algorithm-independent approach uses the fact that all parameter generators share two concepts, those of a source of randomness and a size. Although the concept of size is universally shared by all algorithm parameters, it is interpreted differently for different algorithms. For example, in the case of parameters for the DSA algorithm, the size is the prime modulus, in bits. When the algorithm-independent approach is used, any algorithm-specific parameter-generation values default to some standard values.

An `init` method takes these two universally shared types of arguments. There is also one that takes just a size argument; it uses a system-provided source of randomness:

```
public void init(int size, SecureRandom random);
public void init(int size)
```

In the algorithm-specific approach, a parameter-generator object is initialized using algorithm-specific semantics, which are represented by a set of algorithm-specific parameter generation values supplied in an `AlgorithmParameterSpec object`:

```
public void init(AlgorithmParameterSpec genParamSpec,
                 SecureRandom random)
public void init(AlgorithmParameterSpec genParamSpec)
```

In the generation of the system parameters in, for example, the Diffie-Hellman scheme, the parameter-generation values usually consist of the size of the prime modulus and the size of the random exponent, both specified in bits. The source of randomness is explicitly specified if you call the first `init` method shown earlier or is system provided if you call the second `init` method.

Once you have created and initialized an `AlgorithmParameterGenerator` object, you can generate the algorithm parameters by using the `generateParameters` method:

```
public AlgorithmParameters generateParameters()
```

10.4.6 `Key` and `KeySpec`

This section describes the following key-related interfaces and their subinterfaces, as well as some classes implementing them:

- `Key`
- `PublicKey`
- `PrivateKey`
- `KeySpec`

`Key`

The top-level interface for all opaque keys is `java.security.Key`. It defines the functionality shared by all opaque key objects.

In an *opaque* key representation, you have no direct access to the key material that constitutes a key. In other words, the opaque representation gives you limited access to the key by calling the three methods defined by the `Key` interface: `getAlgorithm`, `getFormat`, and `getEncoded`. In a *transparent* representation, by contrast, you can access each key-material value individually, through one of the `get` methods defined in the corresponding specification class, as shown later, under `KeySpec`.

All opaque keys have three characteristics:

1. An algorithm—the key algorithm for that key
2. An encoded form
3. A format

The key algorithm is usually an encryption or asymmetric operation algorithm, such as `DSA` or `RSA`, that will work with those algorithms and with related algorithms, such as MD5withRSA and SHA1withRSA. The name of the algorithm of a key is obtained by using the `getAlgorithm` method:

```
public String getAlgorithm()
```

The encoded form is an external encoded form for the key used when a standard representation of the key is needed outside the JVM, as when transmitting the key to another party. The key is encoded according to a standard format, such as X.509 or PKCS #8 [108], and is returned using the `getEncoded` method:

```
public byte[] getEncoded()
```

The format is the name of the format of the encoded key and is returned by the `getFormat` method:

```
public String getFormat()
```

Keys are generally obtained through key generators, certificates, key specifications—using a `KeyFactory`—or a `KeyStore` implementation that accesses a keystore database used to manage keys. Using a `KeyFactory`, you can parse encoded keys in an algorithm-specific manner. Similarly, you can use `CertificateFactory` to parse certificates. `KeyFactory` and `CertificateFactory` are described in Section 10.4.7; `KeyStore`, in Section 10.4.9.

PublicKey and PrivateKey

The `java.security.PublicKey` and `java.security.PrivateKey` interfaces both extend the `Key` interface. They are methodless interfaces used for type safety and type identification.

KeySpec

A key specification is a transparent representation of the key material that constitutes a key. If it is stored on a hardware device, the key's specification may contain information that helps identify the key on the device. A key's being transparent means that you can access each key-material value individually, through one of the `get` methods defined in the corresponding specification class. For example, `DSAPrivateKeySpec` defines `getX`, `getP`, `getQ`, and `getG` methods to access the private key *x* and the DSA algorithm parameters used to calculate the key: the prime *p*, the subprime *q*, and the base *g*.

A key may be specified either in an algorithm-specific way or in an algorithm-independent encoding format, such as ASN.1. For example, a DSA private key may be specified by its components *x*, *p*, *q*, and *g* (see `DSAPrivateKeySpec`) or by using its DER encoding (see `PKCS8EncodedKeySpec`).

The `java.security.spec.KeySpec` interface contains no methods or constants. Its only purpose is to group and provide type safety for all key specifications. All key specifications must implement this interface. Descriptions of a number of classes implementing `KeySpec`, all in the `java.security.spec` package, follow. Code examples using `DSAPrivateKeySpec` and `DSAPublicKeySpec` appear in Section 10.6.3.

`DSAPrivateKeySpec`. This class implements the `KeySpec` interface, specifying a DSA private key with its associated parameters. This class has the following methods, which return the private key *x* and the DSA algorithm parameters used to calculate the key: the prime *p*, the subprime *q*, and the base *g*:

```
public BigInteger getX()
public BigInteger getP()
```

```
public BigInteger getQ()
public BigInteger getG()
```

`DSAPublicKeySpec`. This class implements the `KeySpec` interface, specifying a DSA public key with its associated parameters. This class has the following methods, which return the public key *y* and the DSA algorithm parameters used to calculate the key: the prime *p*, the subprime *q*, and the base *g*:

```
public BigInteger getY()
public BigInteger getP()
public BigInteger getQ()
public BigInteger getG()
```

`RSAPrivateKeySpec`. This class implements the `KeySpec` interface, specifying an RSA private key. This class has the following methods to return the RSA modulus *n* and private exponent *d* values, which constitute the RSA private key:

```
public BigInteger getModulus()
public BigInteger getPrivateExponent()
```

`RSAPrivateCrtKeySpec`. This class extends the `RSAPrivateKeySpec` class and specifies an RSA private key, as defined in the PKCS #1 [105] standard, using the Chinese Remainder Theorem (CRT) information values. This class has the following methods, in addition to those inherited from its `RSAPrivateKeySpec` superclass:

```
public BigInteger getPublicExponent()
public BigInteger getPrimeP()
public BigInteger getPrimeQ()
public BigInteger getPrimeExponentP()
public BigInteger getPrimeExponentQ()
public BigInteger getCrtCoefficient()
```

These methods return the public exponent *e* and the CRT information integers: the prime factor *p* of the modulus *n*, the prime factor *q* of *n*, the exponent *d mod (p-1)*, the exponent *d mod (q-1)*, and the CRT coefficient *(inverse of q) mod p*. An RSA private key logically consists of only the modulus and the private exponent. The presence of the CRT values is intended for efficiency.

`RSAMultiPrimePrivateCrtKeySpec`. This class extends the `RSAPrivateKeySpec` class and specifies an RSA multiprime private key, as defined in the PKCS #1 v2.1 standard, using the CRT information values. `RSAMultiPrimePrivateCrtKeySpec` has the following methods, in addition to the methods inherited from its superclass `RSAPrivateKeySpec`:

```
public BigInteger getPublicExponent()
public BigInteger getPrimeP()
public BigInteger getPrimeQ()
public BigInteger getPrimeExponentP()
public BigInteger getPrimeExponentQ()
public BigInteger getCrtCoefficient()
public RSAOtherPrimeInfo[] getOtherPrimeInfo()
```

These methods return the public exponent *e* and the CRT information integers: the prime factor *p* of the modulus *n*, the prime factor *q* of *n*, the exponent *d mod (p-1)*, the exponent *d mod (q-1)*, and

the CRT coefficient *(inverse of q) mod p*. Method `getOtherPrimeInfo` returns a copy of the `otherPrimeInfo`, as defined in PKCS #1 v2.1, or `null`, if there are only two prime factors (*p* and *q*).

An RSA private key logically consists of only the modulus and the private exponent. The presence of the CRT values is intended for efficiency.

`RSAPublicKeySpec`. This class implements the `KeySpec` interface and specifies an RSA public key. This class has the following methods, which return the RSA modulus *n* and public exponent *e* values that constitute the RSA public key:

```
public BigInteger getModulus()
public BigInteger getPublicExponent()
```

`EncodedKeySpec`. This abstract class implements the `KeySpec` interface and represents a public or private key in encoded format. The class's `getEncoded` and `getFormat` methods return the encoded key and the name of the encoding format, respectively:

```
public abstract byte[] getEncoded();
public abstract String getFormat();
```

`PKCS8EncodedKeySpec`. This subclass of `EncodedKeySpec` represents the DER encoding of a private key, according to the format specified in the PKCS #8 standard. The subclass's `getEncoded` method returns the key bytes, encoded according to the PKCS #8 standard, and its `getFormat` method returns the string `"PKCS #8"`.

`X509EncodedKeySpec`. This subclass of `EncodedKeySpec` represents the DER encoding of a public or private key, according to the format specified in the X.509 standard. The subclass's `getEncoded` method returns the key bytes, encoded according to the X.509 standard, and its `getFormat` method returns the string `"X.509"`.

10.4.7 `KeyFactory` and `CertificateFactory`

This section reviews the factory classes for generating keys and certificates.

`KeyFactory`

The `java.security.KeyFactory` class is an engine class designed to provide conversions between opaque cryptographic keys of type `Key` and key specifications, transparent representations of the underlying key material. Key factories are bidirectional. That is, you can build an opaque `Key` object from a given key specification—key material—or retrieve the underlying key material of a `Key` object.

Multiple compatible key specifications may exist for the same key. For example, a DSA public key may be specified by its components *y*, *p*, *q*, and *g* or by using its DER encoding according to the X.509 standard. A key factory can be used to translate between compatible key specifications. Key parsing can be achieved through translation between compatible key specifications. For example, when you translate from `X509EncodedKeySpec` to `DSAPublicKeySpec`, you basically are parsing the encoded key into its components.

As with all engine classes, a `KeyFactory` object for a particular type of key algorithm is obtained by calling one of the `getInstance` static factory methods on the `KeyFactory` class:

```
public static KeyFactory getInstance(String algorithm)
public static KeyFactory getInstance(String algorithm,
    String provider)
public static KeyFactory getInstance(String algorithm,
    Provider provider)
```

If you have a key specification for a public or private key, you can obtain an opaque `PublicKey` or `PrivateKey` object from the specification by using the `generatePublic` or `generatePrivate` method, respectively:

```
public PublicKey generatePublic(KeySpec keySpec)
public PrivateKey generatePrivate(KeySpec keySpec)
```

Conversely, if you have a `Key` object, you can get a corresponding `KeySpec` object by calling the `getKeySpec` method:

```
public KeySpec getKeySpec(Key key, Class keySpec)
```

The `keySpec` parameter identifies the specification class in which the key material should be returned. It could be, for example, `DSAPublicKeySpec.class` to indicate that the key material should be returned in an instance of the `DSAPublicKeySpec` class.

The use of a `KeyFactory` to obtain a `PrivateKey` from a key specification is is shown in Section 10.6.3.

CertificateFactory

The `java.security.cert.CertificateFactory` class is an engine class that defines the functionality of a certificate factory. A certificate factory is used to generate certificate and CRL objects from their encodings.

A certificate factory for an X.509 certificate must return certificates and CRLs that are instances of `java.security.cert.X509Certificate` and `java.security.cert.X509CRL`, respectively.

As with all engine classes, a `CertificateFactory` object for a particular certificate or CRL type is obtained by calling one of the `getInstance` static factory methods on the `CertificateFactory` class:

```
public static CertificateFactory getInstance(String type)
public static CertificateFactory getInstance(String type,
                                 String provider)
public static CertificateFactory getInstance(String type,
                                 Provider provider)
```

To generate a `Certificate` or `CRL` object and initialize it with the data read from an input stream, use the `generateCertificate` or `generateCRL` method, respectively:

```
public final Certificate generateCertificate(InputStream is)
public final CRL generateCRL(InputStream is)
```

To return a possibly empty collection view of the certificates or CRLs read from a given input stream, use the `generateCertificates` or `generateCRL`s method, respectively:

```
public final Collection generateCertificates(InputStream is)
public final Collection generateCRLs(InputStream is)
```

The use of a `CertificateFactory` to generate certificates read from an input stream is shown in Section 10.6.4.

10.4.8 `KeyPair` and `KeyPairGenerator`

The `java.security.KeyPair` class is a holder for a key pair: a public key and a private key. This class has two public methods, one each for returning the private key and the public key:

```
public PrivateKey getPrivate()
public PublicKey getPublic()
```

The `java.security.KeyPairGenerator` class is an engine class used to generate pairs of public and private keys. The generation can be algorithm independent or algorithm specific, depending on how the object is initialized.

All key-pair generation starts with a `KeyPairGenerator`. A key-pair generator for a particular algorithm creates a public/private key pair that can be used with this algorithm and also associates algorithm-specific parameters with each of the generated keys. To create a `KeyPairGenerator`, use one of the factory methods:

```
public static KeyPairGenerator getInstance(String algorithm)
public static KeyPairGenerator getInstance(String algorithm,
                                           String provider)
public static KeyPairGenerator getInstance(String algorithm,
                                           Provider provider)
```

A key-pair generator needs to be initialized before it can generate keys. In most cases, algorithm-independent initialization is sufficient. All key-pair generators share two concepts: those of a source of randomness and a key size. The key size is interpreted differently for different algorithms. For example, in the case of the DSA algorithm, the size is the length of the modulus.

One `initialize` method takes these two universally shared types of arguments, whereas another one takes just a key size argument because it uses a system-provided source of randomness:

```
public void initialize(int keysize, SecureRandom random)
public void initialize(int keysize)
```

Because no other parameters are specified when you call the preceding algorithm-independent `initialize` methods, the provider must decide what to do about any algorithm-specific parameters to be associated with each key. For example, if the algorithm is DSA and the modulus size (key size) is 512, 768, or 1,024, the SUN provider uses a set of precomputed values for the *p*, *q*, and *g* parameters. If the modulus size is not one of these values, the SUN provider creates a new set of parameters. Other providers might have precomputed parameter sets for more than just the three modulus sizes mentioned here. Still others might not have a list of precomputed parameters at all and instead always create new parameter sets.

In some cases, you need an algorithm-specific initialization: for example, when a set of algorithm-specific parameters already exists, as is the case for so-called community parameters in DSA. Two `initialize` methods take an `AlgorithmParameterSpec` argument (described in Section 10.4.5). One does not take a `SecureRandom` argument, in which case its source of randomness is provided by the system:

```
public void initialize(AlgorithmParameterSpec params,
                       SecureRandom random)
public void initialize(AlgorithmParameterSpec params)
```

To generate a key pair, call the following method from `KeyPairGenerator`:

```
public KeyPair generateKeyPair()
```

Multiple calls to `generateKeyPair` yield different key pairs.

Examples of generating a public/private key pair are shown in Section 10.6.2.

10.4.9 `KeyStore`

The `java.security.KeyStore` class is an engine class that defines interfaces to access and modify the information in a keystore. This section discusses `KeyStore`'s API design and implementation. Section 12.8.1 describes the keystore, which can be used to manage a repository of keys and certificates, and Section 12.8.2 demonstrates creation and manipulation of keystores by the `keytool` utility.

Two command line tools make use of `KeyStore`: `keytool` and `jarsigner`, as well as a GUI-based tool, `Policy Tool`. `KeyStore` is also used by the default `Policy` implementation when it processes policy files. SDK users can write additional security applications that use or extend `KeyStore`.

Multiple different concrete implementations are possible, where each implementation is for a particular type of keystore. For example, one implementation might provide persistent keystores, whereas another can use smart cards. Thus, keystore implementations of various types are not meant to be compatible.

A `KeyStore` implementation is provider based, as with all engine classes. The SDK has a built-in default implementation, provided by the SUN provider supplied by Sun Microsystems, that implements the keystore as a file, using a proprietary keystore type—format—named JKS.

The SunJCE provider supplied by Sun Microsystems also includes its own `KeyStore` implementation. The provider implements the keystore as a file, using a proprietary keystore type named JCEKS. This implementation uses a much stronger protection of private keys—using password-based encryption with Triple DES—than the keystore implementation from the SUN provider.

`KeyStore` represents an in-memory collection of keys and certificates and manages two types of entries: key entries and trusted certificate entries. (See Section 12.8.1.) To create a `KeyStore` object, you call one of the `getInstance` static factory methods on the `KeyStore` class, specifying the keystore type and, optionally, specifying the name of a provider or a `Provider` instance:

```
public static KeyStore getInstance(String type)
```

```
public static KeyStore getInstance(String type,
                                   String provider)
public static KeyStore getInstance(String type,
                                   Provider provider)
```

A keystore type defines the storage and data format of the keystore information, as well as the algorithms used to protect private keys in the keystore and the integrity of the keystore. The SDK default implementation of the keystore uses a proprietary keystore type named JKS. `String`s specifying types are not case sensitive; thus, `"jks"` would be considered the same as `"JKS"`.

The `KeyStore` class defines a static method `getDefaultType`, as follows:

```
public final static String getDefaultType()
```

This method returns the value of the `keystore.type` security property, or the default `"JKS"` value if no such security property value is defined. (Setting security property values is described in Section 12.3.3.) The following line of code creates a `KeyStore` instance of the default keystore type:

```
KeyStore ks =
KeyStore.getInstance(KeyStore.getDefaultType());
```

Before a `KeyStore` object can be used, the keystore data must be loaded into memory via the `load` method:

```
public final void load(InputStream stream, char[] password)
```

The optional `password` is used to check the integrity of the keystore data. If no password is supplied, no integrity check is performed. If you want to create an empty keystore, pass `null` as the `InputStream` argument to the `load` method.

Each entry in a keystore is identified by a unique alias string. An enumeration of the alias names present in the keystore can be obtained as follows:

```
public final Enumeration aliases()
```

The following methods determine whether the entry specified by the given alias is a key entry or a trusted certificate entry:

```
public final boolean isKeyEntry(String alias)
public final boolean isCertificateEntry(String alias)
```

The `setCertificateEntry` method assigns a certificate to a specified alias:

```
public final void setCertificateEntry(String alias,
                                      Certificate cert)
```

The `setKeyEntry` methods add—if `alias` does not yet exist—or set key entries:

```
public final void setKeyEntry(String alias, byte[] key,
                              Certificate[] chain)
public final void setKeyEntry(String alias, Key key,
```

```
                char[] password, Certificate[] chain)
```

The `deleteEntry` method deletes an entry:

```
public final void deleteEntry(String alias)
```

The `getKey` method returns the key associated with the given alias. The key is recovered by using the given password:

```
public final Key getKey(String alias, char[] password)
```

The following methods return the certificate, or certificate chain, respectively, associated with the given alias:

```
public final Certificate getCertificate(String alias)
public final Certificate[] getCertificateChain(String alias)
```

You can determine the name, or alias, of the first entry whose certificate matches a given certificate via the following:

```
public final String getCertificateAlias(Certificate cert)
```

The in-memory keystore can be saved via the `store` method:

```
public final void store(OutputStream stream, char[]
password)
```

10.4.10 Randomness and Seed Generators

A basic concept of cryptography is random-number generation. Randomness is the source of security in cryptography and is very useful, and sometimes essential, when generating keys and providing unique identifiers in, for example, challenge-response protocols.

The base class of a random-number generator is `java.util.Random`, introduced in JDK 1.0. The generator does not produce pure random numbers; rather, it produces *pseudo-random* numbers.

`Random` uses a 48-bit seed, which is modified using a linear congruent formula [63]. Here are its interfaces:

```
public Random()
public Random(long seed)
void setSeed(long seed)
protected int next(int bits)
boolean nextBoolean()
void nextBytes(byte[] bytes)
double nextDouble()
float nextFloat()
double nextGaussian()
int nextInt()
int nextInt(int n)
long nextLong()
```

You can construct a `Random` object and assign it a seed either in the constructor or via the `setSeed` method. If a seed is not assigned explicitly, it is by default a value based on the time at which the object is created. After the object has been initialized, various `next` methods can be called to obtain the next random number in different forms. The generator is deterministic in that if two instances of `Random` are created with the same seed and if the same sequence of method calls is made for each, both instances will generate and return identical sequences of numbers. Subclasses of `Random` are permitted to use other algorithms.

Security-savvy readers will have noticed by now that neither the default seeding scheme nor the subsequent number-generation algorithm produces numbers that are as unpredictable as a security application would normally require. Thus the need for `java.security.SecureRandom`: It provides a cryptographically strong pseudo-random-number generator (PRNG).

Like other algorithm-based classes in the SDK, the `java.security.SecureRandom` engine class provides implementation-independent algorithms, whereby an application requests a particular PRNG algorithm and is handed back a `SecureRandom` object for that algorithm. The application can also request a particular algorithm from a specific provider. For example, the default provider SUN supports a built-in algorithm named SHA1PRNG.

As with all engine classes, to get a `SecureRandom` instance for a particular type of PRNG algorithm, you call one of the `getInstance` static factory methods on the `SecureRandom` class:

```
public static SecureRandom getInstance(String algorithm)
public static SecureRandom getInstance(String algorithm,
                                       String provider)
public static SecureRandom getInstance(String algorithm,
                                       Provider provider)
```

Using `getInstance` is the preferred way to obtain `SecureRandom` objects, even though public constructors are still provided for backward compatibility. If these constructors are called, the default provider with the default algorithm is used:

```
public SecureRandom()
public SecureRandom(byte[] seed)
```

The `SecureRandom` implementation attempts to randomize completely the internal state of the generator itself. However, this seeding process does not happen until the first time that random output is needed, that is, when `nextBytes` is called. Thus, the caller can explicitly seed the `SecureRandom` object by calling one of the `setSeed` methods:

```
public void setSeed(byte[] seed)
public void setSeed(long seed)
```

Here is an example of calling `setSeed` right after `getInstance`:

```
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
random.setSeed(seed);
```

Once the `SecureRandom` object has been seeded, it attempts to produce bits as random as the original seeds. At any time, a `SecureRandom` object might be reseeded by using one of the `setSeed` methods. The newly given seed supplements rather than replaces the existing seed. Thus, repeated calls do not reduce randomness.

To get random bytes, a caller simply passes an array of any length, which is then filled with random bytes:

```
public void nextBytes(byte[] bytes)
```

`SecureRandom` itself can also help with seed generation—for example, for seeding another `SecureRandom` object—by calling the `generateSeed` method to generate a given number of seed bytes:

```
public byte[] generateSeed(int numBytes)
```

Examples of the use of `SecureRandom` to generate a random seed to be used by a `KeyPairGenerator` are shown in Section 10.6.2.

10.5 Additional Cryptography Classes

This section describes the design and usage of Java Cryptography Extension classes, which are now integrated with the rest of the Java Cryptography Architecture classes in the Java 2 SDK 1.4.

10.5.1 `Cipher`

The `javax.crypto.Cipher` class provides the functionality of a cryptographic cipher used for encryption and decryption. This class forms the core of the JCE framework.

Creating a Cipher Object

As with all engine classes, a `Cipher` object is created by calling one of the `getInstance` static factory methods on the `Cipher` class. To create a `Cipher` object, you must specify the transformation name. You may also specify which provider you want to supply the implementation of the requested transformation:

```
public static Cipher getInstance(String transformation);
public static Cipher getInstance(String transformation,
                                 String provider);
public static Cipher getInstance(String transformation,
                                 Provider provider);
```

A transformation is a string that describes the operation or set of operations to be performed on the given input to produce some output. A transformation always includes the name of a cryptographic algorithm, such as DES, and may be followed by a mode and padding scheme.

A transformation is of one of the following forms:

```
"algorithm/mode/padding"
"algorithm"
```

For example, the following are valid transformations:

```
"DES/CBC/PKCS5Padding"
"DES"
```

If no mode or padding has been specified, provider-specific default values for the mode and padding scheme are used. For example, the SunJCE provider uses ECB as the default mode and PKCS5Padding as the default padding scheme for DES, DES-EDE, and `Blowfish` ciphers. This means that in the case of the SunJCE provider, the following statements are equivalent:

```
Cipher c1 = Cipher.getInstance("DES/ECB/PKCS5Padding");

Cipher c1 = Cipher.getInstance("DES");
```

When requesting a block cipher in stream cipher mode, such as DES in CFB or OFB mode, you may, optionally, specify the number of bits to be processed at a time by appending this number to the mode name, as shown in the `"DES/CFB8/NoPadding"` and `"DES/OFB32/PKCS5Padding"` transformations. If no such number is specified, a provider-specific default is used. (For example, the SunJCE provider uses a default of 64 bits.) Section 10.7 lists standard names that can be used to specify the algorithm name, mode, and padding scheme components of a transformation.

The objects returned by the `getInstance` factory methods are uninitialized and must be initialized before they become usable.

Initializing a Cipher Object

A `Cipher` object obtained via `getInstance` must be initialized for one of four modes, which are defined as final integer constants in the `Cipher` class. The modes can be referenced by their symbolic names, which are shown here along with a description of the purpose of each mode:

- `ENCRYPT_MODE`, for encryption of data
- `DECRYPT_MODE`, for decryption of data
- `WRAP_MODE`, for wrapping a Key into bytes so that it can be securely transported
- `UNWRAP_MODE`, for unwrapping a previously wrapped key into a `java.security.Key` object

Each of the `Cipher` initialization methods takes a mode parameter (`opmode`) and initializes the `Cipher` object for that mode. Other parameters include the key (`key`) or certificate containing the key (`certificate`), algorithm parameters (`params`), and a source of randomness (`random`). To initialize a `Cipher` object, call one of the following `init` methods:

```
public void init(int opmode, Key key);
public void init(int opmode, Certificate certificate)
public void init(int opmode, Key key,
                 SecureRandom random);
public void init(int opmode, Certificate certificate,
                 SecureRandom random)
public void init(int opmode, Key key,
                 AlgorithmParameterSpec params);
public void init(int opmode, Key key,
                 AlgorithmParameterSpec params,
                 SecureRandom random);
public void init(int opmode, Key key,
                 AlgorithmParameters params)
```

```
public void init(int opmode, Key key, AlgorithmParameters
params,
                 SecureRandom random)
```

If a `Cipher` object that requires parameters, such as an initialization vector, is initialized for encryption and no parameters are supplied to the `init` method, the underlying cipher implementation is supposed to supply the required parameters itself, either by generating random parameters or by using a default, provider-specific set of parameters. However, if a `Cipher` object that requires parameters is initialized for decryption and no parameters are supplied to the `init` method, an `InvalidKeyException` or `InvalidAlgorithmParameterException` exception will be raised, depending on the `init` method that was used. Also note that the same parameters that were used for encryption must be used for decryption.

Note that when a `Cipher` object is initialized, it loses all previously acquired state. In other words, initializing a `Cipher` is equivalent to creating a new instance of that `Cipher` and initializing it. For example, if a `Cipher` is first initialized for decryption with a given key and then initialized for encryption, it will lose any state acquired while in decryption mode.

Encrypting and Decrypting Data

Data can be encrypted or decrypted in one step—single-part operation—or in multiple steps—multiple-part operation. A multiple-part operation is useful if you do not know in advance how long the data is going to be or if the data is too long to be stored in memory all at once.

To encrypt or decrypt data in a single step, call one of the `doFinal` methods:

```
public byte[] doFinal(byte[] input);
public byte[] doFinal(byte[] input, int inputOffset, int
inputLen);
public int doFinal(byte[] input, int inputOffset, int
inputLen,
                   byte[] output);
public int doFinal(byte[] input, int inputOffset, int
inputLen,
                   byte[] output, int outputOffset)
```

To encrypt or decrypt data in multiple steps, first make one or more calls to one or more of the `update` methods to supply all the data to be encrypted or decrypted:

```
public byte[] update(byte[] input);
public byte[] update(byte[] input, int inputOffset, int
inputLen);
public int update(byte[] input, int inputOffset, int
inputLen,
                  byte[] output);
public int update(byte[] input, int inputOffset, int
inputLen,
                  byte[] output, int outputOffset)
```

A multiple-part operation must be terminated by one of the preceding `doFinal` methods—if some input data is still left for the last step—or by one of the following `doFinal` methods—if no input data is left for the last step:

```
public byte[] doFinal();
public int doFinal(byte[] output, int outputOffset);
```

All the `doFinal` methods take care of any necessary padding or unpadding, if padding or unpadding was requested as part of the specified transformation.

A call to `doFinal` resets the `Cipher` object to the state it was in when initialized via a call to `init`. That is, the `Cipher` object is reset and available to encrypt or decrypt more data, depending on the operation mode that was specified in the call to `init`.

Sections 10.6.5 and 10.6.6 show coding examples for encrypting and decrypting data using a `Cipher` object.

Wrapping and Unwrapping Keys

Wrapping a key enables secure transfer of the key from one place to another. The wrap/unwrap API makes it more convenient to write code, as it works with key objects directly. These methods also enable the possibility of secure transfer of hardware-based keys.

To *wrap* a `Key`, first initialize the `Cipher` object for `WRAP_MODE`, and then call the following:

```
public final byte[] wrap(Key key);
```

If you are supplying the wrapped key bytes, the result of calling `wrap`, to someone else who will unwrap them, be sure also to send the following additional information the recipient will need in order to do the unwrap:

- The name of the key algorithm
- The type of the wrapped key (`Cipher.SECRET_KEY`, `Cipher.PRIVATE_KEY`, or `Cipher.PUBLIC_KEY`)

The key algorithm name can be determined by calling the `getAlgorithm` method from the `Key` interface:

```
public String getAlgorithm();
```

To *unwrap* the bytes returned by a previous call to `wrap`, first initialize a `Cipher` object for `UNWRAP_MODE`, and then call the following:

```
public final Key unwrap(byte[] wrappedKey,
                        String wrappedKeyAlgorithm,
                        int wrappedKeyType));
```

Here, `wrappedKey` is the bytes returned from the previous call to `wrap`, `wrappedKeyAlgorithm` is the algorithm associated with the wrapped key, and `wrappedKeyType` is the type of the wrapped key. This must be one of `Cipher.SECRET_KEY`, `Cipher.PRIVATE_KEY`, or `Cipher.PUBLIC_KEY`.

Managing Algorithm Parameters

The parameters being used by the underlying `Cipher` implementation, which were either explicitly passed to the `init` method by the application or generated by the underlying implementation itself, can be retrieved from the `Cipher` object by calling its `getParameters` method, which returns the parameters as a `java.security.AlgorithmParameters` object or, if no parameters are being used, `null`. If the parameter is an initialization vector (`IV`), it can also be retrieved by calling the `getIV` method:

```
public AlgorithmParameters getParameters()
public byte[] getIV()
```

In the following example, a `Cipher` object implementing password-based encryption is initialized with just a key and no parameters. However, the selected algorithm for password-based encryption requires two parameters: a salt and an iteration count. These will be generated by the underlying algorithm implementation itself. The application can retrieve the generated parameters from the `Cipher` object, as follows:

```
import javax.crypto.*;
import java.security.AlgorithmParameters;

// get cipher object for password-based encryption
Cipher c = Cipher.getInstance("PBEWithMD5AndDES");

// initialize cipher for encryption, without supplying
// any parameters. Here, "myKey" is assumed to refer
// to an already generated key.
c.init(Cipher.ENCRYPT_MODE, myKey);

// encrypt some data and store away ciphertext
// for later decryption
byte[] cipherText = c.doFinal("This is just an
example".getBytes());

// retrieve parameters generated by underlying cipher
// implementation
AlgorithmParameters algParams = c.getParameters();

// get parameter encoding and store it away
byte[] encodedAlgParams = algParams.getEncoded();
```

The same parameters that were used for encryption must be used for decryption. They can be instantiated from their encoding and used to initialize the corresponding `Cipher` object for decryption, as follows:

```
import javax.crypto.*;
import java.security.AlgorithmParameters;

// get parameter object for password-based encryption
AlgorithmParameters algParams;
algParams =
AlgorithmParameters.getInstance("PBEWithMD5AndDES");

// initialize with parameter encoding from above
```

```
algParams.init(encodedAlgParams);

// get cipher object for password-based encryption
Cipher c = Cipher.getInstance("PBEWithMD5AndDES");

// initialize cipher for decryption, using one of the
// init() methods that takes an AlgorithmParameters
// object, and pass it the algParams object from above
 c.init(Cipher.DECRYPT_MODE, myKey, algParams);
```

If you did not specify any parameters when you initialized a `Cipher` object and you are not sure whether the underlying implementation uses any parameters, you can find out by simply calling the `getParameters` method of your `Cipher` object and checking the value returned. A return value of `null` indicates that no parameters were used.

The following cipher algorithms implemented by the SunJCE provider use parameters:

- DES, DES-EDE, and `Blowfish`, when used in feedback—CBC, CFB, OFB, or PCBC—mode, use an initialization vector (`IV`). The class `javax.crypto.spec.IvParameterSpec` can be used to initialize a `Cipher` object with a given `IV`.
- `PBEWithMD5AndDES` uses a set of parameters, comprising a salt and an iteration count. The `javax.crypto.spec.PBEParameterSpec` class can be used to initialize a `Cipher` object implementing `PBEWithMD5AndDES` with a given salt and iteration count.

Note that you do not have to worry about storing or transferring any algorithm parameters for use by the decryption operation if you use the `SealedObject` class. This class attaches the parameters used for sealing, or encryption, to the encrypted object contents and uses the same parameters for unsealing, or decryption.

Cipher Output Considerations

Some of the `update` and `doFinal` methods of `Cipher` allow the caller to specify the output buffer into which to encrypt or decrypt the data. In these cases, it is important to pass a buffer that is large enough to hold the result of the encryption or decryption operation.

The following method in `Cipher` can be used to determine how big the output buffer should be:

```
public int getOutputSize(int inputLen)
```

10.5.2 `KeyGenerator`

A `javax.crypto.KeyGenerator` is used to generate secret keys for symmetric algorithms.

Creating a Key Generator

As with all engine classes, a `KeyGenerator` object is created by calling one of the `getInstance` static factory methods on the `KeyGenerator` class. To create a `KeyGenerator` object, you must specify the name of a symmetric algorithm for which a secret key is to be generated. You may also specify which provider you want to supply the implementation of the requested algorithm:

```
public static KeyGenerator getInstance(String algorithm);
public static KeyGenerator getInstance(String algorithm,
                                       String provider);
public static KeyGenerator getInstance(String algorithm,
                                       Provider provider);
```

Initializing a KeyGenerator Object

A key generator for a particular symmetric-key algorithm creates a symmetric key that can be used with that algorithm. A key generator also associates algorithm-specific parameters, if any, with the generated key.

A key can be generated in an algorithm-independent manner or in an algorithm-specific manner. The only difference between the two is the initialization of the object.

All key generators share the concepts of a key size and a source of randomness. An `init` method takes these two universally shared types of arguments. Another one takes just a `keysize` argument and uses a system-provided source of randomness, and yet another takes just a source of randomness:

```
public void init(SecureRandom random);
public void init(int keysize);
public void init(int keysize, SecureRandom random);
```

Because no other parameters are specified when you call these algorithm-independent `init` methods, it is up to the provider what to do about the algorithm-specific parameters, if any, to be associated with the generated key.

For situations in which a set of algorithm-specific parameters already exists, two `init` methods have an `AlgorithmParameterSpec` argument. One also has a `SecureRandom` argument, while the source of randomness is system-provided for the other:

```
public void init(AlgorithmParameterSpec params);
public void init(AlgorithmParameterSpec params,
                 SecureRandom random);
```

In case the client does not explicitly initialize the `KeyGenerator` via a call to an `init` method, each provider must supply and document a default initialization.

Creating a Key

The following method generates a secret key:

```
public SecretKey generateKey();
```

10.5.3 `SecretKeyFactory`

The `javax.crypto.SecretKeyFactory` class represents a factory for secret keys. Key factories are used to convert opaque cryptographic keys of type `java.security.Key` into key specifications, transparent representations of the underlying key material in a suitable format, and vice versa.

A `SecretKeyFactory` object operates only on secret, or symmetric, keys, whereas a `java.security.KeyFactory` object (Section 10.4.7) processes the public and private key components of a key pair.

Objects of type `java.security.Key`, of which `java.security.PublicKey`, `java.security.PrivateKey`, and `javax.crypto.SecretKey` are subclasses, are opaque key objects; that is, you cannot tell how they are implemented. The underlying implementation is provider dependent and may be software or hardware based. Key factories allow providers to supply their own implementations of cryptographic keys. For example, if you have a key specification for a Diffie-Hellman public key, consisting of the public value *y*, the prime modulus *p*, and the base *g*, and you feed the same specification to Diffie-Hellman key factories from different providers, the resulting `PublicKey` objects will most likely have different underlying implementations.

A provider should document the key specifications supported by its secret-key factory. For example, the `SecretKeyFactory` for DES keys supplied by the SunJCE provider supports `DESKeySpec` as a transparent representation of DES keys, the `SecretKeyFactory` for DES-EDE keys supports `DESedeKeySpec` as a transparent representation of DES-EDE keys, and the `SecretKeyFactory` for PBE supports `PBEKeySpec` as a transparent representation of the underlying password.

As with all engine classes, a `SecretKeyFactory` object for a particular type of secret-key algorithm is obtained by calling one of the `getInstance` static factory methods on the `SecretKeyFactory` class:

```
public static SecretKeyFactory getInstance(String
algorithm);
public static SecretKeyFactory getInstance(String algorithm,
                                           String provider);
public static SecretKeyFactory getInstance(String algorithm,
                                           Provider
provider);
```

The following is an example of how to use a `SecretKeyFactory` to convert secret-key data into a `SecretKey` object, which can be used for a subsequent `Cipher` operation:

```
// Note the following bytes are not realistic secret key
data
// bytes but are simply supplied as an illustration of using
data
// bytes (key material) you already have to build a
DESKeySpec.
byte[] desKeyData = { (byte)0x01, (byte)0x02, (byte)0x03,
   (byte)0x04, (byte)0x05, (byte)0x06, (byte)0x07,
(byte)0x08 };
DESKeySpec desKeySpec = new DESKeySpec(desKeyData);
SecretKeyFactory keyFactory =
SecretKeyFactory.getInstance("DES");
SecretKey secretKey = keyFactory.generateSecret(desKeySpec);
```

In this case, the underlying implementation of `secretKey` is based on the provider of `keyFactory`.

An alternative, provider-independent way of creating a functionally equivalent `SecretKey` object from the same key material is to use the `javax.crypto.spec.SecretKeySpec` class, which implements the `javax.crypto.SecretKey` interface:

```
byte[] desKeyData = { (byte)0x01, (byte)0x02, ...};
SecretKeySpec secretKey = new SecretKeySpec(desKeyData,
"DES");
```

10.5.4 `KeyAgreement`

The `javax.crypto.KeyAgreement` class provides the functionality of a key-agreement protocol. The keys involved in establishing a shared secret are created by `KeyPairGenerator` or `KeyGenerator`, a `KeyFactory`, or as a result from an intermediate phase of the key-agreement protocol.

Creating a `KeyAgreement` Object

Each party involved in the key agreement has to create a `KeyAgreement` object. As with all engine classes, a `KeyAgreement` object for a particular type of key-agreement algorithm is obtained by calling one of the `getInstance` static factory methods on the `KeyAgreement` class:

```
public static KeyAgreement getInstance(String algorithm);
public static KeyAgreement getInstance(String algorithm,
                                       String provider);
public static KeyAgreement getInstance(String algorithm,
                                       Provider provider);
```

Initializing a `KeyAgreement` Object

You initialize a `KeyAgreement` object with your private information. In the case of Diffie-Hellman, you initialize it with your Diffie-Hellman private key. Additional initialization information may contain a source of randomness and/or a set of algorithm parameters. Note that if the requested key-agreement algorithm requires the specification of algorithm parameters and if only a key but no parameters are provided to initialize the `KeyAgreement` object, the key must contain the required algorithm parameters. (For example, the Diffie-Hellman algorithm uses a prime modulus *p* and a base generator *g* as its parameters.)

To initialize a `KeyAgreement` object, call one of its `init` methods:

```
public void init(Key key);
public void init(Key key, SecureRandom random);
public void init(Key key, AlgorithmParameterSpec params);
public void init(Key key, AlgorithmParameterSpec params,
                 SecureRandom random);
```

Executing a `KeyAgreement` Phase

Every key-agreement protocol consists of a number of phases that need to be executed by each party involved in the key agreement. To execute the next phase in the key agreement, call the `doPhase` method:

```
public Key doPhase(Key key, boolean lastPhase);
```

The `key` parameter contains the key to be processed by that phase. In most cases, this is the public key of one of the other parties involved in the key agreement or an intermediate key that was generated by a previous phase. The `doPhase` method may return an intermediate key that you may have to send to the other parties of this key agreement so they can process it in a subsequent phase.

The `lastPhase` parameter specifies whether the phase to be executed is the last one in the key agreeement. A value of `FALSE` indicates that more phases of the key agreement are to follow, and a value of `TRUE` indicates that this is the last phase of the key agreement and that `generateSecret` can be called next.

In the example of Diffie-Hellman between two parties, you call `doPhase` once, with `lastPhase` set to `TRUE`. In the example of Diffie-Hellman among three parties, you call `doPhase` twice: the first time with `lastPhase` set to `FALSE` and the second time with `lastPhase` set to `TRUE`.

Generating the Shared Secret

After each party has executed all the required key-agreement phases, each party can compute the shared secret by calling one of the `generateSecret` methods:

```
public byte[] generateSecret();
public int generateSecret(byte[] sharedSecret, int offset);
public SecretKey generateSecret(String algorithm);
```

10.5.5 `Mac`

The `javax.crypto.Mac` class provides the functionality of a message authentication code (MAC).

Creating a `Mac` Object

As with all engine classes, a `Mac` object for a particular type of MAC algorithm is obtained by calling one of the `getInstance` static factory methods on the `Mac` class:

```
public static Mac getInstance(String algorithm);
public static Mac getInstance(String algorithm,
                              String provider);
public static Mac getInstance(String algorithm,
                              Provider provider);
```

Initializing a `Mac` Object

To initialize a `Mac` object, call one of its `init` methods. A `Mac` object is always initialized with a secret key and may, optionally, be initialized with a set of parameters, depending on the underlying MAC algorithm:

```
public void init(Key key);
public void init(Key key, AlgorithmParameterSpec params);
```

You can initialize your `Mac` object with any secret-key object that implements the `javax.crypto.SecretKey` interface. This could be an object returned by

`javax.crypto.KeyGenerator.generateKey()`, or one that is the result of a key-agreement protocol, as returned by `javax.crypto.KeyAgreement.generateSecret()`, or an instance of `javax.crypto.spec.SecretKeySpec`.

With some MAC algorithms, the secret-key algorithm associated with the secret-key object used to initialize the `Mac` object does not matter, as is the case with the HMAC-MD5 and HMAC-SHA1 implementations of the SunJCE provider. With others, however, the secret-key algorithm does matter, and an `InvalidKeyException` is thrown if a secret-key object with an inappropriate secret-key algorithm is used.

Computing a MAC

A MAC can be computed in one step—single-part operation—or in multiple steps—multiple-part operation. A multiple-part operation is useful if you do not know in advance how long the data is going to be or if the data is too long to be stored in memory all at once.

To compute the MAC of some data in a single step, call the following `doFinal` method:

```
public byte[] doFinal(byte[] input);
```

To compute the MAC of some data in multiple steps, first make one or more calls to one or more of the `update` methods to supply all the data:

```
public void update(byte input);
public void update(byte[] input);
public void update(byte[] input, int inputOffset, int
inputLen);
```

A multiple-part operation must be terminated by the preceding `doFinal` method if some input data is still left for the last step or by one of the following `doFinal` methods if no input data is left for the last step:

```
public byte[] doFinal();
public void doFinal(byte[] output, int outOffset);
```

10.6 Code Examples

This section presents several examples to illustrate further how you can use the classes discussed in this chapter.

10.6.1 Computing a Message Digest

The first example computes a message digest, or hash, using the SHA-1 algorithm. Suppose that you have a message composed of three `byte` arrays: `i1`, `i2`, and `i3`. First, you create a properly initialized message digest object. Then you run the three `byte` arrays through the message digest object to calculate the hash, as follows:

```
MessageDigest sha = MessageDigest.getInstance("SHA-1");
sha.update(i1);
sha.update(i2);
sha.update(i3);
byte[] hash = sha.digest();
```

The call to the `digest` method signals the end of the input message and causes the digest to be calculated. An alternative equivalent approach, calling a different `digest` method, which takes the last segment of the input as a parameter, is the following:

```
sha.update(i1);
sha.update(i2);
byte[] hash = sha.digest(i3);
```

After the message digest has been calculated, the message digest object is automatically reset and ready to receive new data and calculate its digest. All former state, that is, the data supplied to `update` calls, is lost.

In some hash implementations, you can obtain intermediate hash values through cloning. Suppose that you want to calculate separate hashes for three separate messages of this form:

`i1`

`i1` and `i2`

`i1`, `i2`, and `i3`

You can perform the computations as follows:

```
/* compute the hash for i1 */
sha.update(i1);
byte[] i1Hash = sha.clone().digest();

/* compute the hash for i1 and i2 */
sha.update(i2);
byte[] i12Hash = sha.clone().digest();

/* compute the hash for i1, i2 and i3 */
sha.update(i3);
byte[] i123hash = sha.digest();
```

This works only if the SHA-1 implementation is cloneable. One way to determine whether cloning is possible is to attempt to clone the `MessageDigest` object and see whether a `CloneNotSupportedException` is thrown.

10.6.2 Generating a Public/Private Key Pair

The second coding example generates a public/private key pair for the DSA algorithm. Keys are generated with a 1,024-bit modulus, using a user-derived seed: `userSeed`. First, you get a `KeyPairGenerator` object for generating keys for the DSA algorithm. Then, to initialize the `KeyPairGenerator`, you need a random seed, obtained from a `SecureRandom` object:

```
KeyPairGenerator keyGen =
KeyPairGenerator.getInstance("DSA");
SecureRandom random =
      SecureRandom.getInstance("SHA1PRNG", "SUN");
random.setSeed(userSeed);
keyGen.initialize(1024, random);
```

Suppose that you already have a set of DSA-specific parameters—*p*, *q*, and *g*—that you want to use to generate your key pair. Then the key-pair generator should be initialized differently, as in the following example:

```
KeyPairGenerator keyGen =
KeyPairGenerator.getInstance("DSA");
DSAParameterSpec dsaSpec = new DSAParameterSpec(p, q, g);
SecureRandom random =
      SecureRandom.getInstance("SHA1PRNG", "SUN");
random.setSeed(userSeed);
keyGen.initialize(dsaSpec, random);
```

Finally, you generate the key pair:

```
KeyPair pair = keyGen.generateKeyPair();
```

10.6.3 Generating and Verifying Signatures

This example generates and verifies a signature using the key pair generated in Section 10.6.2. First, you create a `Signature` object. Then, using the key pair generated in the preceding section, you initialize the object with the private key and sign the data to be signed, which is in a `byte` array called `data`:

```
Signature dsa = Signature.getInstance("SHA1withDSA");
/* Initialize the Signature object with a private key */
PrivateKey priv = pair.getPrivate();
dsa.initSign(priv);

/* Supply the data to be signed, and sign it */
dsa.update(data);
byte[] sig = dsa.sign();
```

Verifying the signature is straightforward:

```
/* Initialize the Signature object with the public key */
PublicKey pub = pair.getPublic();
dsa.initVerify(pub);

/* Supply the data to be verified, and verify it */
dsa.update(data);
boolean verifies = dsa.verify(sig);
System.out.println("signature verifies: " + verifies);
```

Suppose that, rather than having a public/private key pair, you have only the components of your DSA private key: *x* (the private key), *p* (the prime), *q* (the sub-prime), and *g* (the base). Further suppose that you want to use your private key to digitally sign some data, which is in a `byte` array named `someData`. Then the following code should be used:

```
DSAPrivateKeySpec dsaPrivKeySpec =
      new DSAPrivateKeySpec(x, p, q, g);

KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PrivateKey privKey =
```

```
        keyFactory.generatePrivate(dsaPrivKeySpec);

Signature sig = Signature.getInstance("SHA1withDSA");
sig.initSign(privKey);
sig.update(someData);
byte[] signature = sig.sign();
```

This code also illustrates how to create a key specification and use a key factory to obtain a `PrivateKey` from the key specification.

Now suppose that your personal attorney, Alice, wants to use the data you signed. For her to do so and so that she can verify your signature, you need to send her three things: the data, the signature, and the public key corresponding to the private key you used to sign the data. You can store the `someData` bytes in one file and the signature bytes in another and send both files to Alice. For the public key, assume, as in the previous signing example, that you have the components of the DSA public key corresponding to the DSA private key used to sign the data. Then you can create a `DSAPublicKeySpec` from those components:

```
DSAPublicKeySpec dsaPubKeySpec =
        new DSAPublicKeySpec(y, p, q, g);
```

You still need to extract the key bytes so that you can put them in a file. To do this, you first call the `generatePublic` method on the DSA key factory already created in the preceding example and then extract the encoded key bytes:

```
PublicKey pubKey = keyFactory.generatePublic(dsaPubKeySpec);
byte[] encKey = pubKey.getEncoded();
```

You now can store these bytes in a file and send it to Alice, along with the files containing the data and the signature.

Once Alice receives these files, she copies the data bytes from the data file to a `byte` array named `data`, the signature bytes from the signature file to a `byte` array named `signature`, and the encoded public-key bytes from the public-key file to a `byte` array named `encodedPubKey`. To verify the signature, she runs the following code, which illustrates how to use a key factory to instantiate a DSA public key from its encoding. (*Note:* `initVerify` requires a `PublicKey`.)

```
X509EncodedKeySpec pubKeySpec =
        new X509EncodedKeySpec(encodedPubKey);

KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

Signature sig = Signature.getInstance("SHA1withDSA");
sig.initVerify(pubKey);
sig.update(data);
sig.verify(signature);
```

Alice can also convert `pubKey` to a `DSAPublicKeySpec` in order to access the key components:

```
DSAPublicKeySpec dsaPubKeySpec =
```

```
      (DSAPublicKeySpec) keyFactory.getKeySpec(pubKey,
                           DSAPublicKeySpec.class)
BigInteger y = dsaPubKeySpec.getY();
BigInteger p = dsaPubKeySpec.getP();
BigInteger q = dsaPubKeySpec.getQ();
BigInteger g = dsaPubKeySpec.getG();
```

10.6.4 Reading a File That Contains Certificates

The examples in this section each read a file that contains certificates. In the first, the certificates are Base64-encoded. Such certificates are each bounded at the beginning and the end, respectively, by

```
-----BEGIN CERTIFICATE-----
```

and

```
-----END CERTIFICATE-----.
```

In the first example, we convert a `FileInputStream`, which does not support the `mark` and `reset` methods, to a `ByteArrayInputStream`, which does support those methods. We do this so that each call to `generateCertificate` consumes only one certificate and the read position of the input stream is positioned to the next certificate in the file:

```
FileInputStream fis = new FileInputStream(filename);
DataInputStream dis = new DataInputStream(fis);
CertificateFactory cf =
    CertificateFactory.getInstance("X.509");
byte[] bytes = new byte[dis.available()];
dis.readFully(bytes);
ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
while (bais.available() > 0) {
   Certificate cert = cf.generateCertificate(bais);
   System.out.println(cert.toString());
}
```

The second example shows how to parse a PKCS #7 [107] formatted certificate reply stored in a file and extract all the certificates from it. A certificate reply is received from a Certification Authority as a result of submitting a certificate signing request asking the CA to sign a certificate containing your public key. The CA authenticates your identity and returns one or more certificates authenticating that you are the owner of the public key. Suppose that `filename` is the name of the file containing the certificate(s). Here is the code:

```
FileInputStream fis = new FileInputStream(filename);
CertificateFactory cf =
    CertificateFactory.getInstance("X.509");
Collection c = cf.generateCertificates(fis);
Iterator i = c.iterator();
while (i.hasNext()) {
   Certificate cert = (Certificate) i.next();
   System.out.println(cert);
}
```

10.6.5 Using Encryption

This example takes you through the process of generating a key, creating and initializing a cipher object, encrypting a file, and then decrypting it. Throughout this example, we use the Data Encryption Standard (DES).

Generating a Key

To create a DES key, we have to instantiate a `KeyGenerator` for DES. We do not specify a provider, because we do not care about a particular DES key-generation implementation. Because we do not initialize the `KeyGenerator`, a system-provided source of randomness will be used to create the DES key:

```
KeyGenerator keygen = KeyGenerator.getInstance("DES");
SecretKey desKey = keygen.generateKey();
```

After the key has been generated, the same `KeyGenerator` object can be reused to create further keys.

Creating a Cipher

The next step is to create a `Cipher` instance. To do this, we use one of the `getInstance` factory methods of the `Cipher` class. We must specify the name of the requested transformation, which includes the following components, separated by slashes (/):

- The algorithm name
- The mode (optional)
- The padding scheme (optional)

In this example, we create a DES (Data Encryption Standard) cipher in Electronic Code Book mode, with PKCS #5–style padding. We do not specify a provider, because we do not care about a particular implementation of the requested transformation.

The standard algorithm name for DES is `"DES"`, the standard name for the Electronic Code Book mode is `"ECB"`, and the standard name for PKCS #5–style padding is `"PKCS5Padding"`:

```
Cipher desCipher;

// Create the cipher
desCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
```

We use the generated `desKey` to initialize the `Cipher` object for encryption:

```
// Initialize the cipher for encryption
desCipher.init(Cipher.ENCRYPT_MODE, desKey);

// Our cleartext
byte[] cleartext = "This is just an example".getBytes();

// Encrypt the cleartext
byte[] ciphertext = desCipher.doFinal(cleartext);
```

```
// Initialize the same cipher for decryption
desCipher.init(Cipher.DECRYPT_MODE, desKey);

// Decrypt the ciphertext
byte[] cleartext1 = desCipher.doFinal(ciphertext);
```

Note that `cleartext` and `cleartext1` are identical.

10.6.6 Using Password-Based Encryption

In this example, we prompt the user for a password from which we derive an encryption key. It would seem logical to collect and store the password in an object of type `java.lang.String`. However, here's the caveat: Objects of type `String` are immutable; that is, no methods are defined that allow you to change—overwrite—or zero out the contents of a `String` after usage. This feature makes `String` objects unsuitable for storing security-sensitive information, such as user passwords. You should always collect and store security-sensitive information in a `char` array instead. For that reason, the `javax.crypto.spec.PBEKeySpec` class takes and returns a password as a `char` array.

The following method is an example of how to collect a user password as a `char` array:

```
    /**
     * Reads user password from given input stream.
     */
    public char[] readPasswd(InputStream in) throws
IOException {
        char[] lineBuffer;
        char[] buf;
        int i;

        buf = lineBuffer = new char[128];

        int room = buf.length;
        int offset = 0;
        int c;

loop:   while (true) {
            switch (c = in.read()) {
              case -1:
              case '\n':
                break loop;

              case '\r':
                int c2 = in.read();
                if ((c2 != '\n') && (c2 != -1)) {
                    if (!(in instanceof
PushbackInputStream)) {
                        in = new PushbackInputStream(in);
                    }
                    ((PushbackInputStream)in).unread(c2);
                } else
                    break loop;
```

```
                default:
                  if (--room < 0) {
                      buf = new char[offset + 128];
                      room = buf.length - offset - 1;
                     System.arraycopy(lineBuffer, 0, buf, 0,
offset);
                      Arrays.fill(lineBuffer, ' ');
                      lineBuffer = buf;
                  }
                  buf[offset++] = (char) c;
                  break;
              }
          }
          if (offset == 0) {
              return null;
          }

          char[] ret = new char[offset];
          System.arraycopy(buf, 0, ret, 0, offset);
          Arrays.fill(buf, ' ');

          return ret;
      }
```

In order to use PBE as defined in PKCS #5, we have to specify a salt and an iteration count. The same salt and iteration count that are used for encryption must be used for decryption:

```
PBEKeySpec pbeKeySpec;
PBEParameterSpec pbeParamSpec;
SecretKeyFactory keyFac;

// Salt
byte[] salt = {
    (byte)0xc7, (byte)0x73, (byte)0x21, (byte)0x8c,
    (byte)0x7e, (byte)0xc8, (byte)0xee, (byte)0x99
};

// Iteration count
int count = 20;

// Create PBE parameter set
pbeParamSpec = new PBEParameterSpec(salt, count);

// Prompt user for encryption password.
// Collect user password as char array (using the
// "readPasswd" method from above), and convert
// it into a SecretKey object, using a PBE key
// factory.
System.out.print("Enter encryption password:  ");
System.out.flush();
pbeKeySpec = new PBEKeySpec(readPasswd(System.in));
keyFac = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
```

```
SecretKey pbeKey = keyFac.generateSecret(pbeKeySpec);

// Create PBE Cipher
Cipher pbeCipher = Cipher.getInstance("PBEWithMD5AndDES");

// Initialize PBE Cipher with key and parameters
pbeCipher.init(Cipher.ENCRYPT_MODE, pbeKey, pbeParamSpec);

// Our cleartext
byte[] cleartext = "This is another example".getBytes();

// Encrypt the cleartext
byte[] ciphertext = pbeCipher.doFinal(cleartext);
```

10.6.7 Additional Sample Programs

Additional sample programs are shown in the *JCE Reference Guide*, available with the rest of the security documentation at `http://java.sun.com/j2se/m.n/docs/guide/security/`, where *m.n* refers to the release number. For example, the security documentation for the 1.4 release is at `http://java.sun.com/j2se/1.4/docs/guide/security/`.

10.7 Standard Names

Whether in Java documentation or code, algorithms, certificates, and keystore types are referred to by specialized names. These names are not chosen randomly but rather according to adopted standards. This section lists the names used and explains their backgrounds.

In some cases, naming conventions are suggested for forming names that are not explicitly listed, to facilitate name consistency across provider implementations. Such suggestions use items in angle brackets, such as <digest> and <encryption>, as placeholders to be replaced by specific message digest, encryption algorithm, and other names.

Note: Algorithm, certificate, and keystore type names are treated case insensitively. Thus, for example, "dsa" and "DSA" are considered equivalent.

10.7.1 Message Digest Algorithms

Message digest algorithm names can be specified when generating an instance of `MessageDigest`.

- **MD2.** The MD2 message digest algorithm as defined in RFC 1319 [59].
- **MD5.** The MD5 message digest algorithm as defined in RFC 1321 [102].
- **SHA-1, SHA-256, SHA-384,** and **SHA-512.** Hash algorithms as defined in Secure Hash Standard, NIST FIPS 180-2 [93]. SHA-1 is a 160-bit hash function intended to provide 80 bits of security. SHA-256 is a 256-bit hash function intended to provide 128 bits of security against collision attacks. SHA-512 is a 512-bit hash function intended to provide 256 bits of security. A 384-bit hash may be obtained by truncating the SHA-512 output.

10.7.2 Key and Parameter Algorithms

Key and parameter algorithm names can be specified when generating an instance of `KeyPairGenerator`, `KeyFactory`, `KeyAgreement`, `AlgorithmParameterGenerator`, and `AlgorithmParameters`.

- **RSA.** The RSA encryption algorithm as defined in PKCS #1
- **DSA.** The Digital Signature Algorithm as defined in FIPS PUB 186
- **DiffieHellman.** Diffie-Hellman Key Agreement as defined in PKCS #3: Diffie-Hellman Key-Agreement Standard, RSA Laboratories, version 1.4, November 1993
- **Blowfish.** The block cipher designed by Bruce Schneier
- **DES.** The Data Encryption Standard as described in FIPS PUB 46-2
- **DESede.** Triple DES Encryption (DES-EDE)
- **PBE.** The password-based encryption algorithm (PKCS #5)

10.7.3 Digital Signature Algorithms

The following digital signature algorithm names can be specified when generating an instance of `Signature`:

- **SHA1withDSA.** The SHA-1 with DSA signature algorithm, which uses the SHA-1 digest algorithm and DSA to create and verify DSA digital signatures as defined in FIPS PUB 186.
- **MD2withRSA.** The MD2 with RSA encryption signature algorithm, which uses the MD2 digest algorithm and RSA to create and verify RSA digital signatures as defined in PKCS #1.
- **MD5withRSA.** The MD5 with RSA encryption signature algorithm, which uses the MD5 digest algorithm and RSA to create and verify RSA digital signatures as defined in PKCS #1.
- **SHA1withRSA.** The signature algorithm with SHA-1 and the RSA encryption algorithm as defined in the OSI Interoperability Workshop, using the padding conventions described in PKCS #1.
- **<digest>with<encryption>.** Used to form a name for a signature algorithm with a particular message digest, such as MD2 or MD5, and algorithm, such as RSA or DSA, just as was done for the explicitly defined standard names in this section, such as MD2withRSA. For the new signature schemes defined in PKCS #1 v2.0, for which the <digest>with<encryption> form is insufficient, <digest>with<encryption>and<mgf> can be used to form a name. Here, <mgf> should be replaced by a mask-generation function, such as MGF1, for example, MD5withRSAandMGF1.

10.7.4 Random-Number Generation Algorithms

Random-number generation algorithm names can be specified when generating an instance of `SecureRandom`.

- **SHA1PRNG.** The name of the pseudo-random-number generation (PRNG) algorithm supplied by the SUN provider. This implementation follows the IEEE P1363 standard, given in its Appendix G.7, Expansion of Source Bits, and uses SHA-1 as the foundation of the PRNG. It computes the SHA-1 hash over a true-random seed value concatenated with a 64-bit counter, which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used.

10.7.5 Certificate Types

Certificate types can be specified when generating an instance of `CertificateFactory`.

- **X.509.** The certificate type defined in X.509

10.7.6 Keystore Types

Keystore types can be specified when generating an instance of `KeyStore`.

- **JKS.** The name of the keystore implementation provided by the SUN provider
- **PKCS12.** The transfer syntax for personal identity information as defined in PKCS #12
- **JCEKS.** The name of the keystore implementation provided by the SunJCE provider

10.7.7 Service Attributes

A cryptographic service is always associated with a particular algorithm or type. For example, a digital signature service is always associated with a particular algorithm, such as DSA, and a `CertificateFactory` service is always associated with a particular certificate type, such as X.509.

The attributes in this section are for cryptographic services. The service attributes can be used as filters for selecting providers. Both the attribute name and value are case insensitive.

- **KeySize.** The maximum key size that the provider supports for the cryptographic service.
- **ImplementedIn.** Whether the implementation for the cryptographic service is done by software or hardware. The value of this attribute is `"software"` or `"hardware"`.

10.7.8 Cipher Algorithms, Modes, and Padding

Algorithms

The following names can be specified as the algorithm component in a transformation when requesting an instance of `Cipher`:

- **AES.** Advanced Encryption Standard as specified by NIST in a draft FIPS. Based on the Rijndael algorithm by Joan Daemen and Vincent Rijmen, AES is a 128-bit block cipher supporting keys of 128, 192, and 256 bits.
- **Blowfish.** The block cipher designed by Bruce Schneier.
- **DES.** The Digital Encryption Standard as described in FIPS PUB 46-2.
- **DESede.** Triple DES Encryption (DES-EDE).
- **PBEWith<digest>And<encryption>** or **PBEWith<prf>And<encryption>.** The password-based encryption algorithm (PKCS #5), using the specified message digest (<digest>) or pseudo-random function (<prf>) and encryption algorithm (<encryption>). Examples:
    - **PBEWithMD5AndDES.** The password-based encryption algorithm as defined in RSA Laboratories, "PKCS #5: Password-Based Encryption Standard," version 1.5, November 1993. Note that this algorithm implies CBC as the cipher mode and PKCS5Padding as the padding scheme and cannot be used with any other cipher modes or padding schemes.
    - **PBEWithHmacSHA1AndDESede.** The password-based encryption algorithm as defined in RSA Laboratories, "PKCS #5: Password-Based Cryptography Standard," version 2.0, March 1999.

- **RC2, RC4**, and **RC5.** Variable-key-size encryption algorithms developed by Ron Rivest for RSA Data Security, Inc.
- **RSA.** The RSA encryption algorithm as defined in PKCS #1.

Modes

The following names can be specified as the mode component in a transformation when requesting an instance of `Cipher`:

- **NONE.** No mode
- **CBC.** Cipher Block Chaining mode, as defined in FIPS PUB 81
- **CFB.** Cipher Feedback mode, as defined in FIPS PUB 81
- **ECB.** Electronic Code Book mode, as defined in The National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) PUB 81, "DES Modes of Operation," U.S. Department of Commerce, December 1980
- **OFB.** Output Feedback mode, as defined in FIPS PUB 81
- **PCBC.** Propagating Cipher Block Chaining, as defined by Kerberos V4

Padding

The following names can be specified as the padding component in a transformation when requesting an instance of `Cipher`:

- **NoPadding.** No padding.
- **OAEPWith<digest>And<mgf>Padding.** Optimal Asymmetric Encryption Padding scheme defined in PKCS #1, where <digest> should be replaced by the message digest and <mgf> by the mask-generation function. Example: OAEPWithMD5AndMGF1Padding.
- **PKCS5Padding.** The padding scheme described in RSA Laboratories, "PKCS #5: Password-Based Encryption Standard," version 1.5, November 1993.
- **SSL3Padding.** The padding scheme defined in the SSL Protocol version 3.0, November 18, 1996, section 5.2.3.2 (CBC block cipher):

```
block-ciphered struct {
    opaque content[SSLCompressed.length];
    opaque MAC[CipherSpec.hash_size];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
} GenericBlockCipher;
```

The size of an instance of a `GenericBlockCipher` must be a multiple of the block cipher's block length.

The padding length, which is always present, contributes to the padding, which implies that if

```
sizeof(content) + sizeof(MAC) % block_length = 0,
```

`padding` has to be `block_length-1` bytes long, because of the existence of `padding_length`.

This makes the padding scheme similar to PKCS5Padding, where the padding length is encoded in the padding and ranges from `1` to `block_length`. With the SSL scheme, the `sizeof(padding)` is encoded in the always present `padding_length` and therefore ranges from `0` to `block_length-1`.

Note that this padding mechanism is not supported by the SunJCE provider.

10.7.9 Key-Generator Algorithms

The following algorithm names can be specified when requesting an instance of `KeyGenerator`. Most of these algorithms were described in the previous section. The HmacMD5 and HmacSHA1 algorithms are described in Section 10.7.11.

- **AES**
- **Blowfish**
- **DES**
- **DESede**
- **HmacMD5**
- **HmacSHA1**

10.7.10 Secret-Key Algorithms

The following algorithm names can be specified when requesting an instance of `SecretKeyFactory`. These algorithms were described in Section 10.7.8.

- **AES**
- **DES**
- **DESede**
- **PBEWith<digest>And<encryption>** or **PBEWith<prf>And<encryption>.** Secret-key factory for use with PKCS #5 password-based encryption, where <digest> is a message digest, <prf> is a pseudo-random function, and <encryption> is an encryption algorithm. Examples: PBEWithMD5AndDES (PKCS #5, v1.5) and PBEWithHmacSHA1AndDESede (PKCS #5, v2.0). *Note:* Both of these use only the low-order 8 bits of each password character.

10.7.11 MAC Algorithms

The following algorithm names can be specified when requesting an instance of `Mac`:

- **HmacMD5.** The HMAC-MD5 keyed-hashing algorithm as defined in RFC 2104: "HMAC:Keyed-Hashing for Message Authentication" (February 1997).
- **HmacSHA1.** The HMAC-SHA1 keyed-hashing algorithm as defined in RFC 2104: "HMAC: Keyed-Hashing for Message Authentication" (February 1997).
- **PBEWith<mac>.** MAC for use with PKCS #5 v2.0 password-based message authentication standard, where <mac> is a message authentication code algorithm name. Example: PBEWithHmacSHA1.

10.8 Algorithm Specifications

When implementing crypto algorithms, a provider should comply with existing standard specifications. Following are some of these specifications and their relationships with SDK implementations. In particular, some or all the following fields are given for each algorithm specification:

- **Name.** The name by which the algorithm is known. This is the name passed to the `getInstance` method when requesting the algorithm and returned by the `getAlgorithm` method to determine the name of an existing algorithm object. These

methods are in the engine classes `Signature`, `MessageDigest`, `KeyPairGenerator`, and `AlgorithmParameterGenerator`.

- **Type.** The type of algorithm: `Signature`, `MessageDigest`, `KeyPairGenerator`, or `ParameterGenerator`.
- **Description.** General notes about the algorithm, including any standards implemented by the algorithm, applicable patents, and so on.
- **Key Pair Algorithm** (optional). The key pair algorithm for this algorithm.
- **Key Size** (optional). Legal key sizes for a keyed algorithm or key-generation algorithm.
- **Size** (optional). Legal sizes for algorithm parameter generation for an algorithm parameter generation algorithm.
- **Parameter Defaults** (optional). Default parameter values for a key-generation algorithm.
- **Signature Format** (optional). The format of the signature for a `Signature` algorithm, that is, the input and output of the `verify` and `sign` methods, respectively.

10.8.1 SHA-1 Message Digest Algorithm

Name: SHA-1

Type: `MessageDigest`

Description: The message digest algorithm as defined in NIST's FIPS 180-1. The output of this algorithm is a 160-bit digest.

10.8.2 MD2 Message Digest Algorithm

Name: MD2

Type: `MessageDigest`

Description: The message digest algorithm as defined in RFC 1319. The output of this algorithm is a 128-bit (16-byte) digest.

10.8.3 MD5 Message Digest Algorithm

Name: MD5

Type: `MessageDigest`

Description: The message digest algorithm as defined in RFC 1321. The output of this algorithm is a 128-bit (16-byte) digest.

10.8.4 Digital Signature Algorithm

Name: SHA1withDSA

Type: `Signature`

Description: The signature algorithm described in NIST FIPS 186, using DSA with the SHA-1 message digest algorithm.

Key Pair Algorithm: DSA

Signature Format: An ASN.1 sequence of two `INTEGER` values `r` and `s`, in that order:

```
SEQUENCE ::= { r INTEGER, s INTEGER }
```

10.8.5 RSA-Based Signature Algorithms

Names: MD2withRSA, MD5withRSA, and SHA1withRSA

Type: `Signature`

Description: The signature algorithms that use the MD2, MD5, and SHA-1 message digest algorithms, respectively, with RSA encryption.

Key Pair Algorithm: RSA

Signature Format: A DER-encoded PKCS #1 block as defined in RSA Laboratory's *Public Key Cryptography Standards Note #1*. The data encrypted is the digest of the data signed.

10.8.6 DSA Key-Pair Generation Algorithm

Name: DSA

Type: `KeyPairGenerator`

Description: The key-pair generation algorithm described in NIST FIPS 186 for DSA.

Key Size: The length, in bits, of the modulus *p*. This must range from 512 to 1,024 and must be a multiple of 64. The default key size is 1,024.

Parameter Defaults: The following default parameter values are used for key sizes of 512, 768, and 1,024 bits. The use of the parameter named `counter` is explained in the FIPS document.

For 512-bit key parameters:

```
SEED = b869c82b 35d70e1b 1ff91b28 e37a62ec dc34409b

counter = 123

p = fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e
    b4a208f3 ae1617ae 01f35b91 a47e6df6 3413c5e1 2ed0899b
    cd132acd 50d99151 bdc43ee7 37592e17

q = 962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5

g = 678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64
    86931d2d 14271b9e 35030b71 fd73da17 9069b32e 2935630e
    1c206235 4d0da20a 6c416e50 be794ca4
```

For 768-bit key parameters:

```
SEED = 77d0f8c4 dad15eb8 c4f2f8d6 726cefd9 6d5bb399

counter = 263
```

```
p = e9e64259 9d355f37 c97ffd35 67120b8e 25c9cd43 e927b3a9
    670fbec5 d8901419 22d2c3b3 ad248009 3799869d 1e846aab
    49fab0ad 26d2ce6a 22219d47 0bce7d77 7d4a21fb e9c270b5
    7f607002 f3cef839 3694cf45 ee3688c1 1a8c56ab 127a3daf

q = 9cdbd84c 9f1ac2f3 8d0f80f4 2ab952e7 338bf511

g = 30470ad5 a005fb14 ce2d9dcd 87e38bc7 d1b1c5fa cbaecbe9
    5f190aa7 a31d23c4 dbbcbe06 17454440 1a5b2c02 0965d8c2
    bd2171d3 66844577 1f74ba08 4d2029d8 3c1c1585 47f3a9f1
    a2715be2 3d51ae4d 3e5a1f6a 7064f316 933a346d 3f529252
```

For 1,024-bit key parameters:

```
SEED = 8d515589 4229d5e6 89ee01e6 018a237e 2cae64cd

counter = 92

p = fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400
    c31e3f80 b6512669 455d4022 51fb593d 8d58fabf c5f5ba30
    f6cb9b55 6cd7813b 801d346f f26660b7 6b9950a5 a49f9fe8
    047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
    83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2
    2203199d d14801c7

q = 9760508f 15230bcc b292b982 a2eb840b f0581cf5

g = f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82
    f9574c0b 3d078267 5159578e bad4594f e6710710 8180b449
    167123e8 4c281613 b7cf0932 8cc8a6e1 3c167a8b 547c8d28
    e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
    cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525
    64014c3b fecf492a
```

10.8.7 RSA Key-Pair Generation Algorithm

Name: RSA

Type: `KeyPairGenerator`

Description: The key-pair generation algorithm described in PKCS #1.

Key Size: Any integer that is a multiple of 8, greater than or equal to 512.

10.8.8 DSA Parameter-Generation Algorithm

Name: DSA

Type: `ParameterGenerator`

Description: The parameter-generation algorithm described in NIST FIPS 186 for DSA.

Size: The length, in bits, of the modulus *p*. This must range from 512 to 1,024 and must be a multiple of 64. The default size is 1,024.

Chapter 11. Network Security

Integrity without knowledge is weak and useless, and knowledge without integrity is dangerous and dreadful.

—Samuel Johnson

The importance of securing the communications path between distributed entities increases as our reliance on Internet technologies compounds. Today, transactions that only a few years ago we would trust being done only over private networks are being conducted using Internet technology. Data communications might commence within the confines of the corporate intranet, presumably a private network, or possibly traverse the open Internet. Applications, such as securities transactions, payments, online banking, payroll, and health care, barely scratch the surface of the types of things that we entrust to travel over open networks. Obviously, this data is extremely sensitive, and all sorts of mischief could ensue if it were to be exposed to the wrong party or if it were modified en route.

The Internet is a less than ideal medium to conduct such transactions. Version 4 of the Internet Protocol [97] is the most widely deployed network-layer protocol, yet it has a number of well-known security problems [6, 7]. To overcome these problems, security features have been added to the Internet Protocol [27, 61]. However, these protocols are not widely adopted and deployed. Over time, this will certainly change, but in the meantime, the most widely deployed alternative is to solve the security problems above the transport layer. The Java 2 platform provides APIs that support the two most commonly used upper-layer security protocols: Kerberos V5 and TLS 1.0.

In Section 8.5, we described the importance of authenticating communicating peers. Having a strong assurance of the identities participating in a dialogue is critical but does nothing to guarantee that the communication is kept confidential and that the messages have not been altered. In this chapter, we complete the description of the Java GSS-API and the JSSE API, focusing on the authentication, confidentiality, and integrity protection mechanisms they provide. To close the chapter, we describe the security properties of Remote Method Invocation.

11.1 Java GSS-API

When we introduced Java GSS-API in Section 8.5.2, we described its general facility for authenticating a distributed end entity. In this section, we go into the details of using the Java GSS-API. Typically, the establishment of a Java GSS-API security context required for securely exchanging messages between entities, such as a client and a server, encompasses authentication, confidentiality, and integrity controls at the time the security context is negotiated.

11.1.1 Using Kerberos Credentials with Java GSS-API

The Java GSS-API framework itself is quite thin, with all security-related functionality delegated to components obtained from the underlying mechanisms. The framework classes and interfaces are in the `org.ietf.jgss` package. The abstract `GSSManager` class serves as a factory for other Java GSS-API classes and also provides information about the mechanisms that are supported. The `GSSManager` class can create instances of classes implementing the following interfaces: `GSSName`, `GSSCredential`, and `GSSContext`. The class also has methods to query for the list of available mechanisms and the name types that each mechanism supports.

Here is sample code showing how the `GSSManager` may be used. First, an instance of the default `GSSManager` subclass may be obtained through the static method `getInstance`:

```
GSSManager manager = GSSManager.getInstance();
```

The GSSName interface encapsulates a single GSS-API principal entity. The application obtains an implementation of this interface through one of the createName methods in the GSSManager class. Conceptually, a GSSName contains many representations of the entity or many primitive name elements, one for each supported underlying mechanism. Different name formats and their definitions are identified with universal object identifiers (OIDs), represented by the Oid class. The format of a name can be derived based on the unique OID of its name type:

```
 Oid krb5Mechanism = new Oid("1.2.840.113554.1.2.2");
 Oid krb5PrincipalNameType = new
Oid("1.2.840.113554.1.2.2.1");
 // Identify who the client wishes to be
GSSName userName = manager.createName("gfe",
GSSName.NT_USER_NAME);
// Identify the name of the server. This uses a Kerberos-
specific
// name format.
GSSName serverName =
manager.createName("nfs/foo.example.com",

krb5PrincipalNameType);
```

The Kerberos V5 mechanism maps this last name to the Kerberos-specific form nfs/foo.example.com@EXAMPLE.COM, where EXAMPLE.COM is the realm of the principal. This principal represents the service nfs running on the host machine foo.example.com in the Kerberos realm EXAMPLE.COM.

The GSSCredential interface encapsulates the GSS-API credentials for an entity. A credential contains all the necessary cryptographic information to enable the creation of a GSS security context on behalf of the entity that it represents. It may contain multiple, distinct, mechanism-specific credential elements, each containing information for a specific security mechanism, but all referring to the same entity. A credential may be used to perform context initiation, acceptance, or both.

GSS-API credential creation is not intended to provide a "login to the network" function, as such a function would involve the creation of new credentials rather than merely acquiring a handle to existing credentials:

```
// Acquire credentials for the user
GSSCredential userCreds = manager.createCredential(userName,

GSSCredential.DEFAULT_LIFETIME,
                                   krb5Mechanism,

GSSCredential.INITIATE_ONLY);
```

The GSS-API itself does not dictate how an underlying mechanism obtains the credentials that are needed for authentication. It is assumed that prior to calling the GSS-API, these credentials are obtained and stored in a location that the mechanism provider is aware of. The default model in the Java 2 platform is that the mechanism providers must obtain credentials only from the private or public credential sets associated with the Subject in the current access control context. For detailed information about Subjects, see Section 8.4 and in particular Section 8.4.7 about dynamic association of a Subject with the current access control context.

In the preceding example, the Kerberos V5 mechanism will search for the required `INITIATE` credentials (`KerberosTicket`) in the private credential set, whereas another mechanism might look in the public set or in both sets. This model has the advantage that credential management is simple and predictable from the application's point of view. An application, given the right permissions, can purge the credentials in the `Subject` or renew them using standard Java APIs.

Security contexts are established between peers using locally acquired credentials. Multiple contexts may exist simultaneously between a pair of peers using the same or a different set of credentials. GSS-API functions in a manner independent of the underlying transport protocol and depends on its calling application to transport the tokens that are generated by the security context between the peers. The `GSSContext` interface provides the security services that are available over the context. Use a `GSSManager` to instantiate a `GSSContext`:

```
// Instantiate and initialize a security context that will
be
// established with the server
GSSContext context = manager.createContext(serverName,
    krb5Mechanism, userCreds, GSSContext.DEFAULT_LIFETIME);
```

This returns an initialized security context that is aware of the peer that it must communicate with, the server, and the mechanism that it must use to do so. The client's credentials (`userCreds`) are necessary to authenticate to the peer.

Before the `GSSContext` can be used for its security services, a security context-has to be established with an exchange of tokens between the two peers. Each call to a context establishment method will generate an opaque token that the application must somehow send to its peer, using a communication channel of its choice. The context establishment occurs in a loop where the initiator, such as a client, calls the `GSSContext initSecContext` method, and the accepter, such as a server, calls `acceptSecContext` until the context is established. While in this loop, the `initSecContext` and `acceptSecContext` methods produce tokens that the application—client or server—sends to the peer. The peer passes any such token as input to its `acceptSecContext` or `initSecContext`, as the case may be. More information about security context establishment is given in Section 11.1.2.

Delegation of Credentials

Java GSS-API allows a client to delegate its credentials securely to the server with which it has established a security context, such that the server can initiate other security contexts on behalf of the client. This feature is useful for single sign-on in a multitier environment. To enable this, the client requests credential delegation prior to making the first call to `initSecContext`:

```
// Enable delegation of credential
context.requestCredDeleg(true);
```

The server receives the delegated credential after context establishment by invoking `getDelegCred` on its instance of `GSSContext`. The server can then pass this `GSSCredential` to the `GSSManager.createContext` method, pretending to be the client.

In the case of the Kerberos V5 mechanism, the delegated credential is a forwardable Ticket Granting Ticket (TGT) that is encapsulated as part of the first token sent from the client to the server. Using this TGT, the server can obtain a service ticket on behalf of the client for any other service.

Mitigating Single Sign-On Risks

The convenience of single sign-on also introduces new risks. Single sign-on environments typically assume that the host operating environment has not been tampered with. Historically, single sign-on environments have not had to contend with untrusted mobile code. Without proper security checks to thwart untrusted code from impersonating the user, all sorts of mayhem may occur.

To illustrate, some details of the safeguards the permissions model must consider are given. For example, suppose that the browser has performed a JAAS login at start-up time and associated a `Subject` with all applets that run in it.

The `Subject` is protected from rogue applets by means of the `javax.security.auth.AuthPermission` class. A relevant `AuthPermission`, with target `"getSubject"` or `"getSubjectFromDomainCombiner"`, is checked whenever code tries to obtain a reference to the `Subject` associated with any access control context. Even if it were given access to a `Subject`, an applet needs a `javax.security.auth.PrivateCredentialPermission` to read the sensitive private credentials stored in it.

Other kinds of checks are to be done by Java GSS-API mechanism providers as they read credentials and establish security contexts on behalf of the credentials' owners. In order to support the Kerberos V5 mechanism, two new permission classes have been added within the `javax.security.auth.kerberos` package:

```
ServicePermission(String servicePrincipal, String action)
DelegationPermission(String principals)
```

As new GSS-API mechanisms are standardized for inclusion in J2SE, packages will be added that contain relevant permission classes for providers of those mechanisms.

Credential Acquisition

The `GSSManager.createCredential` method obtains mechanism-specific credential elements from the current `Subject`. The method then stores them in a `GSSCredential` container. Allowing applets to acquire `GSSCredential`s freely, even if they cannot use them to do much, is undesirable. Doing so would leak information about the existence of user and service principals. Thus, before an application can acquire a `GSSCredential` with any Kerberos credential elements within it, an authorization check is made to ensure that the caller has the necessary `ServicePermission`.

On the client side, a successful `GSSCredential` acquisition implies that a TGT has been accessed from a cache. A `ServicePermission` with action `"initiate"` and target name of the form `"krbtgt/realm@realm"` must be granted to the client code, where *`realm`* is a placeholder for the actual realm. Here is a sample instantiation of such a `ServicePermission`:

```
ServicePermission("krbtgt/EXAMPLE.COM@EXAMPLE.COM",
"initiate");
```

The service principal `krbtgt/EXAMPLE.COM@EXAMPLE.COM` represents the Ticket Granting Service (TGS) in the Kerberos realm `EXAMPLE.COM`, and the action `"initiate"`

suggests that a ticket to this service is being accessed. The TGS service principal will always be used in this permission check at the time of clientside credential acquisition.

On the server side, a successful `GSSCredential` acquisition implies that a secret key has been accessed from a cache. A `ServicePermission` with action `"accept"` and target name of the form *`"servicePrincipal@realm"`* must be granted to the client code, where *`servicePrincipal`* and *`realm`* are placeholders for the actual service principal and realm, respectively. Here is an example:

```
ServicePermission("nfs/foo.example.com@EXAMPLE.COM",
"accept");
```

Here, the service principal `nfs/foo.example.com` represents the Kerberos service principal, and the action `"accept"` suggests that the secret key for this service is being requested.

Context Establishment

An application that has permissions to contact a particular server—say, the LDAP server—must not instead contact a different server, such as the FTP (file transfer protocol) server. Of course, the application might be restricted from doing so with the help of a `java.net.SocketPermission`. Even if the network connection is permitted, it is still possible to use a `ServicePermission` to restrict the application from authenticating using an acquired identity.

When the Kerberos mechanism provider is about to initiate context establishment, it checks that a `ServicePermission` with action `"initiate"` and target name of the form *`"servicePrincipal@realm"`* has been granted to the client code. Here is a sample such `ServicePermission`:

```
ServicePermission("ftp@EXAMPLE.COM", "initiate");
```

This check prevents unauthorized code from obtaining and using a Kerberos service ticket for the principal `ftp@EXAMPLE.COM`.

Providing limited access to specific service principals using this permission is still dangerous. Downloaded code is allowed to communicate back with the host it originated from. A malicious applet could send back the initial GSS-API output token that contains a Kerberos Ticket encrypted in the target service principal's long-term secret key, thus exposing it to an off-line attack. For this reason, it is not advisable to grant any `"initiate" ServicePermission` to untrusted code.

On the server side, the permission to use the secret key to accept incoming security context establishment requests is already checked during credential acquisition. Hence, no checks are made in the context establishment stage.

Credential Delegation

An application that has permission to establish a security context with a server on behalf of a principal also has the ability to request that credentials be delegated to that server. But not all servers are trusted to the extent that all credentials can be delegated to them. Thus, before a Kerberos provider obtains a delegated credential to send to the peer, the provider checks that the context initiator—the client in the previous examples—has an appropriate `javax.security.auth.kerberos.DelegationPermission` to delegate

credentials on behalf of the principal to the server. A `DelegationPermission` has a target name that is a `String` composed of two `String`s designating service principals. When specifying such a `String`, each inner quote is escaped by a `"\"`. Here is a sample `DelegationPermission` instantiation:

```
DelegationPermission(
  "\"ftp@EXAMPLE.COM\"
   \"krbtgt/EXAMPLE.COM@EXAMPLE.COM\"");
```

This permission allows the Kerberos service principal `ftp@EXAMPLE.COM` to receive a forwarded TGT, represented by the Ticket Granting Service `krbtgt/EXAMPLE.COM@EXAMPLE.COM`. The use of two principal names in this permission allows for finer-grained delegation, such as proxy tickets for specific services, unlike a carte blanche forwarded TGT. Even though the GSS-API does not allow for proxy tickets, another API, such as JSSE, might support this idea at some point in the future.

11.1.2 Establishing a Security Context

Before two applications can use Java GSS-API to exchange messages securely between them, they must establish a joint security context using their credentials. As previously noted, the `org.ietf.jgss.GSSContext` interface encapsulates the GSS-API security context and provides the security services that are available. Both applications create and use a `GSSContext` object to establish and maintain the shared information that makes up the security context. If the `GSSContext` is instantiated using the default `GSSManager` instance, the Kerberos V5 GSS-API mechanism is guaranteed to be available for context establishment. This mechanism is identified by the `Oid` "`1.2.840.113554.1.2.2`" and is defined in RFC 1964 [76].

Before the context establishment phase is initiated, the context initiator may request specific characteristics desired of the established context:

```
// Instantiate and initialize a security context that will
be
// established with the server
GSSContext context = manager.createContext(serverName,
   krb5Mechanism, userCreds, GSSContext.DEFAULT_LIFETIME);
// set desired context options prior to context
// establishment
    context.requestConf(true);
    context.requestMutualAuth(true);
    context.requestReplayDet(true);
    context.requestSequenceDet(true);
```

Not all underlying mechanisms support all characteristics that a caller might desire. After the context is established, the caller can check the characteristics and services offered by that context, using various query methods. When using the Kerberos V5 GSS-API mechanism offered by the default `GSSManager` instance, all optional services will be available locally. They are mutual authentication, credential delegation, confidentiality and integrity protection, and per-message replay detection and sequencing. Note that in the GSS-API, message integrity is a prerequisite for message confidentiality.

Recall that the context establishment occurs in a loop, in which the initiator, such as a client, calls `initSecContext`, and the accepter, such as a server, calls `acceptSecContext` until

the context is established. During the context establishment phase, the `isProtReady` method may be called to determine whether the context can be used for the per-message operations of `wrap`, for confidentiality, and `getMIC`, for integrity controls. This allows applications to use per-message operations on contexts that aren't yet fully established.

After the context has been established or the `isProtReady` method returns `true`, the query routines can be invoked to determine the characteristics and services of the established context. The application can also start using the per-message methods of `wrap` and `getMIC` to obtain cryptographic operations on application-supplied data and then send the resulting tokens to the other application with which it has established a security context. The receiver calls the `unwrap` or `verifyMIC` method to interpret each token. When the context is no longer needed, the application should call `dispose` to release any system resources the context may be using.

A security context typically maintains sequencing and replay detection information about the tokens it processes. Therefore, the sequence in which any tokens are presented to this context for processing can be important. Also note that none of the methods in this interface are synchronized. Therefore, it is not advisable to share a `GSSContext` among several threads unless some application-level synchronization is in place.

11.1.3 Message Security

The `MessageProp` utility class is used within the per-message `GSSContext` methods to convey per-message properties. When used with the `GSSContext` interface's `wrap` and `getMIC` methods, an instance of this class is used to indicate the desired quality of protection (QOP) and to specify whether confidentiality services are to be applied to caller-supplied data (`wrap` only). To request the default QOP, the value of 0 should be used.

When used with the `unwrap` and `verifyMIC` methods of the `GSSContext` interface, an instance of the `MessageProp` class is used to indicate the QOP and confidentiality services to be applied over the supplied message. In the case of `verifyMIC`, the confidentiality state will always be `false`:

```
// perform wrap on an application-supplied message, appMsg,
// using QOP = 0, and requesting privacy service
byte [] appMsg ...
MessageProp mProp = new MessageProp(0, true);
byte [] tok = context.wrap(appMsg, 0, appMsg.length, mProp);
sendToken(tok);
// release the local end of the context
context.dispose();
```

11.2 JSSE

In Section 8.5.3, we described the mechanics of authentication supported by JSSE. This section provides API details and explains how to leverage the authentication and use it to provide confidentiality and integrity controls so that data passing between communicating peers is protected from unauthorized disclosure and is resistant to tampering. If the data is tampered with, the integrity controls will indicate that the data has been corrupted. The core JSSE classes are in the `javax.net` and `javax.net.ssl` packages.

11.2.1 Establishing an SSL Context

An `SSLSocket` is created either by an `SSLSocketFactory` or by an `SSLServerSocket` accepting an in-bound connection. (In turn, an `SSLServerSocket` is created by an `SSLServerSocketFactory`.) Both `SSLSocketFactory` and `SSLServerSocketFactory` objects are created by an `SSLContext`.

The simplest technique is to call the static `getDefault` method on either the `SSLSocketFactory` or `SSLServerSocketFactory` class. These methods create a default `SSLContext` with a default `KeyManager`, `TrustManager`, and secure random-number generator. For Sun's implementation of JSSE, the key material used is found in the default keystore/truststore.

The approach that gives the caller the most control over the behavior of the created context is to call the static method `getInstance` on the `SSLContext` class and then initialize the context by calling the instance's `init` method. The `init` method takes three arguments: an array of `KeyManager` objects, an array of `TrustManager` objects, and a `SecureRandom` random-number generator. The `KeyManager` and `TrustManager` objects are created by either implementing the appropriate interface(s) or using the `KeyManagerFactory` and `TrustManagerFactory` classes to generate implementations. The `KeyManagerFactory` and `TrustManagerFactory` can then each be initialized with key material contained in the `KeyStore` passed as an argument to the `TrustManagerFactory`/`KeyManagerFactory init` method. Finally, the `getTrustManagers` method (in `TrustManagerFactory`) and `getKeyManagers` method (in `KeyManagerFactory`) can be called to obtain the array of trust or key managers, one for each type of trust or key material.

Once an SSL connection is established, an `SSLSession` is created that contains various information, such as identities established, cipher suite used, and so on. The `SSLSession` is then used to describe an ongoing relationship and state information between two entities. Each SSL connection involves one session at a time, but that session may be used on many connections between those entities, simultaneously or sequentially. The following sections describe the core JSSE classes.

11.2.2 `SocketFactory` and `ServerSocketFactory` Classes

The abstract `javax.net.SocketFactory` class is used to create sockets. It must be subclassed by other factories, which create particular subclasses of sockets and thus provide a general framework for the addition of public socket-level functionality. (See, for example, `SSLSocketFactory` in Section 11.2.4.) The `javax.net.ServerSocketFactory` class is analogous to the `SocketFactory` class but is used specifically for creating server sockets.

Socket factories are a simple way to capture a variety of policies related to the sockets being constructed, producing such sockets in a way that does not require special configuration of the code that asks for the sockets.

- Due to polymorphism of both factories and sockets, various kinds of sockets can be used by the same application code just by passing various kinds of factories.
- Factories can themselves be customized with parameters used in socket construction. So, for example, factories could be customized to return sockets with various networking timeouts or security parameters already configured.

- The sockets returned to the application can be subclasses of `java.net.Socket` (or `javax.net.ssl.SSLSocket`) so that they can directly expose new APIs for such features as compression, security, record marking, statistics collection, or firewall tunneling.

11.2.3 `SSLSocket` and `SSLServerSocket` Classes

The `javax.net.ssl.SSLSocket` class, a subclass of the standard `java.net.Socket` class, supports all the standard socket methods and adds additional methods specific to secure sockets. Instances of this class encapsulate the `SSLContext` under which they were created. There are APIs to control the creation of secure socket sessions for a socket instance, but trust and key management are not directly exposed. The `javax.net.ssl.SSLServerSocket` class is analogous to the `SSLSocket` class but is used specifically for creating server sockets.

Instances of `SSLSocket` can be obtained in two ways. First, an `SSLSocket` can be created by an instance of `SSLSocketFactory` via one of the several `createSocket` methods on that class. The second way to obtain an `SSLSocket` is through the `accept` method on the `SSLServerSocket` class.

11.2.4 `SSLSocketFactory` and `SSLServerSocketFactory` Classes

A `javax.net.ssl.SSLSocketFactory` acts as a factory for creating secure sockets. This class is an abstract subclass of `javax.net.SocketFactory`.

Secure socket factories encapsulate the details of creating and initially configuring secure sockets. This includes authentication keys, peer certificate validation, enabled cipher suites, and the like. The `javax.net.ssl.SSLServerSocketFactory` class is analogous to the `SSLSocketFactory` class but is used specifically for creating server sockets.

The three primary ways of obtaining an `SSLSocketFactory` are

1. Getting the default factory by calling the `SSLSocketFactory.getDefault` static method.
2. Receiving a factory as an API parameter. That is, code that needs to create sockets but that doesn't care about the details of how the sockets are configured can include a method with an `SSLSocketFactory` parameter that can be called by clients to specify which `SSLSocketFactory` to use when creating sockets. (For example, `javax.net.ssl.HttpsURLConnection` has `setDefaultSSLSocketFactory` and `setSSLSocketFactory` methods that each has an `SSLSocketFactory` parameter.)
3. Constructing a new factory with specifically configured behavior.

The default factory is typically configured to support server authentication only so that sockets created by the default factory do not leak any more information about the client than a normal TCP socket would.

Many classes that create and use sockets do not need to know the details of socket creation behavior. Creating sockets through a socket factory passed in as a parameter is a good way of isolating the details of socket configuration and increases the reusability of classes that create and use sockets.

You can create new socket factory instances either by implementing your own socket factory subclass or by using another class that acts as a factory for socket factories. One example of such a class is `SSLContext` (Section 11.2.8), which is provided with the JSSE implementation as a provider-based configuration class.

11.2.5 `SSLSession` Interface

A `javax.net.ssl.SSLSession` represents a security context negotiated between the two peers of an `SSLSocket` connection. Assuming that the peers share compatible `SSLContext` data, once a session has been arranged, it can be shared by future `SSLSocket`s connected between the same two peers. The session contains the cipher suite that will be used for communications over a secure socket, as well as a nonauthoritative hint as to the network address of the remote peer and management information, such as the time of creation and last use. A session also contains a shared master secret negotiated between the peers that is used to create cryptographic keys for encrypting and guaranteeing the integrity of the communications over an `SSLSocket`. The value of this master secret is known only to the underlying secure socket implementation and is not exposed through the `SSLSession` API.

11.2.6 `HttpsURLConnection` Class

The HTTPS protocol is similar to HTTP, but HTTPS first establishes a secure channel via SSL/TLS sockets before requesting/receiving data. The class `javax.net.ssl.HttpsURLConnection` extends the `java.net.HttpsURLConnection` class, which itself extends `java.net.URLConnection`, and adds support for HTTPS-specific features. See the `java.net.URL, java.net.URLConnection`, and `java.net.HttpURLConnection` classes for more information about how HTTP URLs are constructed and used.

On obtaining an `HttpsURLConnection`, you can configure a number of HTTP/HTTPS parameters before initiating the network connection via the method `URLConnection.connect`. Of particular interest are

- Setting the assigned `SSLSocketFactory`
- Setting the assigned `HostnameVerifier`

Setting the Assigned `SSLSocketFactory`

In some situations, it is desirable to specify the `SSLSocketFactory` that an `HttpsURLConnection` instance uses. For example, you may wish to tunnel through a proxy type that isn't supported by the default implementation. The new `SSLSocketFactory` could return sockets that have already performed all necessary tunneling, thus allowing `HttpsURLConnection` to use additional proxies.

The `HttpsURLConnection` class has a default static `SSLSocketFactory` that is assigned when the class is loaded. Future instances of `HttpsURLConnection` will inherit the current default `SSLSocketFactory` until a new default `SSLSocketFactory` is assigned to the class via the static `HttpsURLConnection.setDefaultSSLSocketFactory` method. Once an instance of `HttpsURLConnection` has been created, the inherited `SSLSocketFactory` on this instance can be overridden with a call to the `setSSLSocketFactory` method.

Setting the Assigned `HostnameVerifier`

If the host name of the URL does not match that in the credentials received as part of the SSL/TLS handshake, it's possible that URL spoofing has occurred. If the implementation cannot determine a host name match with reasonable certainty, the SSL implementation will perform a callback to the instance's assigned `HostnameVerifier` for further checking. The host name verifier can perform whatever steps are necessary to make the determination, such as performing alternative host name pattern matching or perhaps popping up an interactive dialog box. An unsuccessful verification will close the connection. See [100] for more information about host name verification.

11.2.7 The SunJSSE Provider

Sun Microsystems' version of JSSE comes standard with a provider named Sun-JSSE. The SunJSSE provider package supplies the following services:

- A key factory implementation supporting the RSA algorithm
- A key-pair generator for generating a pair of public and private keys suitable for the RSA algorithm
- A keystore supporting PKCS12
- Digital signature algorithms supporting MD2withRSA, MD5withRSA, and SHA1withRSA.
- Key manager and trust manager factories that handle X.509 certificates.
- `SSLContext` implementations for SSL, SSLv3, TLS, and TLSv1 protocols.

11.2.8 `SSLContext` Class

The `javax.net.ssl.SSLContext` class is an engine class (Section 10.2) for an implementation of a secure socket protocol. An instance of this class acts as a factory for SSL socket factories. An `SSLContext` holds all the state information shared across all sockets created under that context. For example, session state is associated with the `SSLContext` when it is negotiated through the handshake protocol by sockets created by socket factories provided by the context. These cached sessions can be reused and shared by other sockets created under the same context.

Each instance is configured through its `init` method with the keys, certificate chains, and trusted root CA certificates that it needs to perform authentication. This configuration is provided in the form of key and trust managers. These managers provide support for the authentication and key-agreement aspects of the cipher suites supported by the context. Currently, only X.509-based managers are supported.

11.2.9 Creating an `SSLContext` Object

Like other JCA provider-based engine classes, `SSLContext` objects are created using the `getInstance` factory methods of the `SSLContext` class. These static methods each return an instance that implements at least the requested secure socket protocol. The returned instance may implement other protocols too. For example, `getInstance("SSLv3")` may return an instance that implements SSLv3 and TLSv1. The `getSupportedProtocols` method returns a list of supported protocols when an `SSLSocket` or `SSLServerSocket` is created from a socket factory obtained from this context. You can control which protocols are enabled for an SSL connection by using the method `setEnabledProtocols(String[] protocols)`. Refer to the API documentation for this method in the `SSLSocket` and `SSLServerSocket` classes for more information.

Note: An `SSLContext` object is automatically created, initialized, and statically assigned to the `SSLSocketFactory` class when you call `SSLSocketFactory.getDefault`. Therefore, you don't have to create and initialize an `SSLContext` object directly, unless you want to override the default behavior.

To create an `SSLContext` object by calling a `getInstance` factory method, you must specify the protocol name. You may also specify which provider you want to supply the implementation of the requested protocol:

```
public static SSLContext getInstance(String protocol);
public static SSLContext getInstance(String protocol, String
   provider);
public static SSLContext getInstance(String protocol,
Provider
    provider);
```

If just a protocol name is specified, the system will determine whether an implementation of the requested protocol is available in the environment and, if there is more than one, if one is preferred. If both a protocol name and a provider are specified, the system will determine whether an implementation of the requested protocol is in the provider requested and throw an exception if there is not.

A protocol is a string, such as `"SSL"`, that describes the secure socket protocol desired. Here is an example of obtaining an `SSLContext`:

```
SSLContext sc = SSLContext.getInstance("SSL");
```

A newly created `SSLContext` should be initialized by calling the `init` method:

```
public void init(KeyManager[] km, TrustManager[] tm,
                 SecureRandom random);
```

If the `KeyManager[]` parameter is `null`, an empty `KeyManager` will be defined for this context. If the `TrustManager[]` parameter is `null`, the installed security providers will be searched for the highest-priority implementation of the `TrustManagerFactory`, from which an appropriate `TrustManager` will be obtained. Likewise, the `SecureRandom` parameter may be `null`, in which case a default implementation will be used.

If the internal default context is used—for example, when an `SSLContext` is created in the internals of JSSE—a default `KeyManager` and `TrustManager` are created. The default `SecureRandom` implementation is also chosen.

11.2.10 `TrustManager` Interface

The primary responsibility of the `javax.net.ssl.TrustManager` is to determine whether the presented authentication credentials should be trusted. If the credentials are not trusted, the connection will be terminated. To authenticate the remote identity of a secure socket peer, you need to initialize an `SSLContext` object with one or more `TrustManager`s. You need to pass one `TrustManager` for each authentication mechanism that is supported. If `null` is passed into the `SSLContext` initialization, a trust manager will be created for you. Typically, an `SSLContext` has a single trust manager. It is also common for the trust manager to support authentication based on X.509 public-key certificates. Some secure socket

implementations may also support authentication based on shared secret keys, Kerberos, or other mechanisms.

11.2.11 `TrustManagerFactory` Class

The `javax.net.ssl.TrustManagerFactory` is an engine class for a providerbased service that acts as a factory for one or more types of `TrustManager` objects. The SunJSSE provider from Sun Microsystems implements a factory that can return a basic X.509 trust manager. As of J2SE 1.4.2, Sun Microsystems also supplies a CertPath-based PKIX trust manager, SunPKIX, in addition to the simple legacy SunX509 trust manager. Because `TrustManagerFactory` is provider based, additional factories can be implemented and configured to provide additional or alternative trust managers that provide more sophisticated services or that implement installation-specific authentication policies.

To create an instance of a `TrustManagerFactory`, the static `getInstance` method is invoked, passing in an algorithm name string and an optional provider specification:

```
public static TrustManagerFactory
                  getInstance(String algorithm);
public static TrustManagerFactory
                  getInstance(String algorithm,
                              String provider);
public static TrustManagerFactory
                  getInstance(String algorithm,
                              Provider provider);
```

A sample algorithm name string is

```
"SunX509"
```

A sample call is the following:

```
TrustManagerFactory tmf =
    TrustManagerFactory.getInstance("SunX509", "SunJSSE");
```

The preceding call creates an instance of the SunJSSE provider's default trust manager factory, which provides basic X.509-based certification path validity checking.

A newly created factory should be initialized by calling one of the `init` methods:

```
public void init(KeyStore ks);
public void init(ManagerFactoryParameters spec);
```

Which `init` method should be invoked depends on what is appropriate for the `TrustManagerFactory` being used. (Ask the provider vendor.)

For many factories, such as the default SunX509 `TrustManagerFactory` from the SunJSSE provider, the `java.security.KeyStore` is the only information required in order to initialize the `TrustManagerFactory`, and thus the first `init` method is the appropriate one to call. The `TrustManagerFactory` will query the `KeyStore` for information on which remote certificates should be trusted during authorization checks.

In some cases, initialization parameters other than a `KeyStore` may be needed by a provider. Users of that particular provider are expected to pass an implementation of the appropriate `ManagerFactoryParameters` as defined by the provider. The provider can then call methods in the `ManagerFactoryParameters` implementation to obtain the needed information.

For example, suppose that the `TrustManagerFactory` provider requires initialization parameters B, R, and S from any application that wishes to use that provider. Like all providers that require initialization parameters other than a `KeyStore`, the provider will require that the application provide an instance of a class that implements a particular `ManagerFactoryParameters` subinterface. In our example, suppose that the provider requires that the calling application implement and create an instance of `MyTrustManagerFactoryParams` and pass it to the second `init`. Here is what `MyTrustManagerFactoryParams` may look like:

```
public interface MyTrustManagerFactoryParams extends
  ManagerFactoryParameters {
    public boolean getBValue();
    public float getRValue();
    public String getSValue():
}
```

Some trust managers are capable of making trust decisions without having to be explicitly initialized with a `KeyStore` object or any other parameters. For example, they may access trust material from a local directory service via LDAP, may use a remote on-line certificate status-checking server, or may access default trust material from a standard local location.

11.2.12 `KeyManager` Interface

The primary responsibility of the `KeyManager` is to select the authentication credentials that will eventually be sent to the remote host. To authenticate yourself—a local secure socket peer—to a remote secure socket peer, you need to initialize an `SSLContext` object with one or more `KeyManager`s. You need to pass one `KeyManager` for each authentication mechanism that will be supported. If `null` is passed into the `SSLContext` initialization, no `KeyManager` will be available. If the internal default context is used, a default `KeyManager` is created. Typically, an `SSLContext` has a single key manager. It is also common for the key manager to support authentication based on X.509 public-key certificates. Some secure socket implementations may also support authentication based on shared secret keys, Kerberos, or other mechanisms.

11.2.13 `KeyManagerFactory` Class

The `javax.net.ssl.KeyManagerFactory` class is an engine class for a providerbased service that acts as a factory for one or more types of `KeyManager` objects. The SunJSSE provider implements a factory that can return a basic X.509 key manager. As of J2SE 1.4.2, Sun Microsystems also supplies a Cert-Path-based PKIX key manager, SunPKIX, in addition to the simple legacy SunX509 key manager. Because `KeyManagerFactory` is provider based, additional factories can be implemented and configured to provide additional or alternative key managers.

An instance of this class is constructed in a similar manner to `SSLContext`, except for passing an algorithm name string instead of a protocol name to the `getInstance` method:

```
public static KeyManagerFactory
                  getInstance(String algorithm);
public static KeyManagerFactory
                  getInstance(String algorithm,
                              String provider);
public static KeyManagerFactory
                  getInstance(String algorithm,
                              Provider provider);
```

A sample algorithm name string is

```
"SunX509"
```

A sample call is the following:

```
KeyManagerFactory kmf =
    KeyManagerFactory.getInstance("SunX509", "SunJSSE");
```

The preceding call creates an instance of the SunJSSE provider's default key manager factory, which provides basic X.509-based authentication keys.

A newly created factory should be initialized by calling one of the `init` methods:

```
public void init(KeyStore ks, char[] password);
public void init(ManagerFactoryParameters spec);
```

Invoke whichever `init` method is appropriate for the `KeyManagerFactory` being used.

For many factories, such as the default SunX509 `KeyManagerFactory` from the SunJSSE provider, only the `KeyStore` and password are required in order to initialize the `KeyManagerFactory`, and thus the first `init` method is the appropriate one to call. The `KeyManagerFactory` will query the `KeyStore` for information on which private key and matching public-key certificates should be used for authenticating to a remote socket peer. The password parameter specifies the password that will be used with the methods for accessing keys from the `KeyStore`. All keys in the `KeyStore` must be protected by the same password.

In some cases, initialization parameters other than a `KeyStore` and password may be needed by a provider. Users of that particular provider are expected to pass an implementation of the appropriate `ManagerFactoryParameters` as defined by the provider. The provider can then call methods in the `ManagerFactoryParameters` implementation to obtain the needed information.

Some factories are capable of providing access to authentication material without having to be initialized with a `KeyStore` object or any other parameters. For example, they may access key material as part of a login mechanism, such as one based on JAAS. As indicated earlier, the SunJSSE provider supports a SunX509 factory that must be initialized with a `KeyStore` parameter.

The `javax.net.ssl.KeyManagerFactory` class is an engine class for a providerbased service that acts as a factory for one or more types of `KeyManager` objects. The SunJSSE provider implements a factory that can return a basic X.509 key manager. Because it is provider based, additional factories can be implemented and configured to provide additional or alternative key managers.

11.3 Remote Method Invocation

Remote Method Invocation (RMI) enables objects in one Java virtual machine to seamlessly invoke methods on objects in a remote Java virtual machine. To accomplish the remote invocation, an RMI stub marshals the arguments to the method, using object serialization, and sends the marshaled invocation across the wire to the server. On the server side, the call is received by the RMI system and connected to a skeleton, which is responsible for unmarshaling the arguments and invoking the server's implementation of the method. When the server's implementation completes, either by returning a value or by throwing an exception, the skeleton marshals the result and sends a reply to the client's stub. The stub unmarshals the reply and either returns the value or throws the exception, as appropriate.

11.3.1 RMI Security Basics

This distributed programming model has obvious security implications. To mitigate the risks, all programs using RMI must install a security manager, or RMI will not download classes—other than from the local class path—for objects received as parameters, return values, or exceptions in remote method calls. This restriction ensures that the operations performed by downloaded code go through a set of security checks.

RMI imposes on clients access control restrictions that are not drastically different from that of an applet. RMI requires the same `SocketPermission` as the equivalent `java.net.Socket` network communication: `"connect"` to make a remote call to an object exported at a given host and port. On the server side, RMI requires the same `SocketPermission`s as the equivalent `java.net.Server-Socket` network communication: `"listen"` to export a remote object on a given port; `"accept"` to receive a remote call made from a given host and port. Also, when an RMI call is dispatched to the exported object on the server, the `AccessControlContext` that was in effect when that remote object was exported will be restored.

11.3.2 RMI Activation

The class `java.rmi.activation.Activatable` and the RMI daemon, `rmid`, were introduced with the 1.2 release of the J2SDK. Now programs can be written to register information about remote object implementations that should be created and executed "on demand." The RMI daemon provides a Java virtual machine from which other JVM instances may be launched.

When it launches a JVM for an activation group, `rmid` uses the information in the group's registered activation group descriptor, `java.rmi.activation.ActivationGroupDesc`. The group descriptor specifies an optional `ActivationGroupDesc.CommandEnvironment` that includes the command to execute to start the activation group, as well as any command line options to be added to the command line. This feature is very powerful, and thus its use should be guarded. Therefore, Sun's implementation of the RMI activation daemon relies on security policy so that `rmid` can verify whether the information in each `ActivationGroupDesc` is allowed to be used to launch a JVM for an activation group.

The permission `com.sun.rmi.rmid.ExecPermission` is used to grant `rmid` permission to execute a command to launch an activation group. The permission `com.sun.rmi.rmid.ExecOptionPermission` is used to allow `rmid` to use command line options, specified as property overrides in the group descriptor or as options in the `CommandEnvironmen`t, when launching the activation group. Refer to the `rmid` javadocs for the normative descriptions of these permission classes.

11.3.3 Securing RMI Communications

RMI fully uses serialization to effect remote method invocations. In Section 9.6, we described the security aspects of object serialization, highlighting the fact that serialized objects may be vulnerable when they exist outside the JVM. When using RMI, objects are essentially shuttled back and forth between two Java virtual machines. While in transit, the serialized data may be exposed to interlopers; as a result, the integrity or confidentiality of the data may be compromised.

In J2SE, RMI added support for custom socket factories for RMI-based communication. Thus, instead of using standard sockets, an RMI application can export a remote object to use an RMI socket factory that creates an SSL socket. Following is a simple example of a `java.rmi.server.RMIClientSocketFactory` implementation:

```
public class RMISSLClientSocketFactory
    implements RMIClientSocketFactory, Serializable {

    public Socket createSocket(String host, int port)
        throws IOException {
        SSLSocketFactory factory =
            (SSLSocketFactory)SSLSocketFactory.getDefault();
         SSLSocket socket =
(SSLSocket)factory.createSocket(host,
                                port);
         return socket;
    }
}
```

Implementing the equivalent server socket is not that much more complex. An implementation of `java.rmi.server.RMIServerSocketFactory` follows:

```
public class RMISSLServerSocketFactory
    implements RMIServerSocketFactory, Serializable {

    public ServerSocket createServerSocket(int port)
        throws IOException {
        SSLServerSocketFactory ssf = null;
        try {
            // set up key manager to do server
authentication
            SSLContext ctx;
            KeyManagerFactory kmf;
            KeyStore ks;
            char[] passphrase = "open sesame".toCharArray();
            ctx = SSLContext.getInstance("TLS");
            kmf = KeyManagerFactory.getInstance("SunX509");
            ks = KeyStore.getInstance("JKS");

            ks.load(new FileInputStream("testkeys"),
passphrase);
            kmf.init(ks, passphrase);
            ctx.init(kmf.getKeyManagers(), null, null);

            ssf = ctx.getServerSocketFactory();
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
        return ssf.createServerSocket(port);
    }
}
```

To tie it together, the exported remote object must simply specify the two RMI socket factory implementation classes when the `java.rmi.server.UnicastRemoteObject` default constructor is invoked:

```
super(0, new RMISSLClientSocketFactory(),
      new RMISSLServerSocketFactory());
```

Chapter 12. Deploying the Security Architecture

Security is the chief pretence of civilization.

—George Bernard Shaw

Chapter 3 provided an overview of the Java security architecture, and Chapters 4 through 6 supplied details. Chapter 7 showed how to customize the architecture: create new `Permission` types, extend the `SecurityManager`, and so on. This chapter provides information about deploying the security architecture.

Section 12.1 provides a link to Web pages that document how to install the Java 2 platform software. Sections 12.2 and 12.3 supply information that is relevant to various configuration options and that is thus referred to by other sections of this chapter and others in the book. Section 12.2 specifies what the placeholder `<java.home>`, used throughout the book, refers to. Section 12.3 tells how to set system and security property values and indicates the location of the *security properties file*, as well as the default value for the `user.home` system property. Section 12.4 explains how to customize a deployment, including how to override or append security properties and how to specify application-specific policies when you run an application. Section 12.5 provides instructions for installing provider packages that supply concrete implementations of a subset of the cryptography or other security services defined as part of the Java 2 Security API or one of its extensions.

Section 12.6 discusses how to specify the security policy indicating which security-sensitive resource accesses are allowed for code from specified sources. The section defines the format to be used to specify the policy in one or more files that can be processed by the default `Policy` implementation and provides examples. This section also tells how you can configure a different `Policy` implementation, if you want to use one other than the default.

Section 12.7 explains how to make and use login configuration files for use by the Java Authentication and Authorization Service. JAAS can be used for user authentication and authorization and is described in Section 8.4.

Section 12.8 then describes two Java security command line tools: `keytool` and `jarsigner`. The `keytool` utility is used to manage a keystore, or database, of private keys and their associated certificates authenticating the corresponding public keys. This utility also manages certificates from trusted entities. The `jarsigner` tool is used to generate digital signatures for Java Archive (JAR) files and to verify the authenticity of signatures of signed JAR files.

Section 12.9 tells what X.500 Distinguished Names are and provides examples of their use. Section 12.10 provides a little advice on managing security policies for nonexpert computer users.

12.1 Installing the Latest Java 2 Platform Software

Java 2 Platform, Standard Edition (J2SE), software for Windows, Solaris, and Linux is available from Sun Microsystems' Java Products and APIs Web site at

http://java.sun.com/products/

Generally, new versions of operating systems or Internet browsers—for example, upgrades that you install or software that comes with a new computer system—already support the latest Java 2 Runtime Environment (JRE). However, to upgrade the JVM yourself, you can download and install any of the following: (1) the JRE, for running Java applications and applets, (2) the SDK

itself for development use, which also includes the JRE, or (3) the Java Plug-in, which upgrades the JVM inside Microsoft Internet Explorer (IE), Netscape Navigator (Navigator), and Mozilla.

12.2 The Installation Directory `<java.home>`

The term `<java.home>` is used throughout this book to refer to the value of the `java.home` system property, which specifies the directory where the JRE is installed. This is the top-level directory of the JRE, or the `jre` directory in the Java 2 SDK (J2SDK) software. For example, if you have J2SDK 1.4 installed on Solaris in a directory named `/home/user1/j2sdk1.4.0` or on Win32 in a directory named `C:\j2sdk1.4.0`, `<java.home>` refers to the following directory:

```
/home/user1/j2sdk1.4.0/jre   [Solaris]
C:\j2sdk1.4.0\jre            [Win32]
```

Similarly, if you have JRE 1.4 installed on Solaris in a directory named `/home/user1/j2re1.4.0` or on Win32 in a directory named `C:\j2re1.4.0`,`<java.home>` refers to the following directory:

```
/home/user1/j2re1.4.0        [Solaris]
C:\j2re1.4.0                 [Win32]
```

12.3 Setting System and Security Properties

Some items are customized by setting system properties, whereas others are customized by setting security properties. The following sections explain how to set values for both types of properties.

12.3.1 Setting System Properties

You can set a system property either statically or dynamically. To set a system property *statically*, use the `-D` option of the `java` command. For example, to run an application named `MyApp` and set the `user.home` system property to specify the user home directory `/home/marys`, enter the following:

```
java -Duser.home=/home/marys MyApp
```

To set a system property *dynamically*, call the `java.lang.System.setProperty` method in your code, substituting the appropriate property name and value:

```
System.setProperty("propertyName",
    "propertyValue");
```

For example, a `setProperty` call corresponding to the previous example for setting the `user.home` system property to specify the user home directory `/home/marys` would be:

```
System.setProperty("user.home",
    "/home/marys");
```

Some system properties have default values. For example, the `java.home` property defaults to the directory into which the Java 2 runtime environment was installed. Another example already mentioned is the `user.home` system property, whose default value is described next.

12.3.2 The Default `user.home` System Property Value

On a Solaris system, the `user.home` system property value defaults to the user's home directory. On a Windows system, the `user.home` property value defaults as follows, given user name `uName`:

`C:\Documents and Settings\uName`	Windows 2000 and XP
`C:\Winnt\Profiles\uName`	Multiuser Windows NT
`C:\Windows\Profiles\uName`	Multiuser Windows 95/98/ME
`C:\Windows`	Single-user Windows 95/98/ME

Thus, if the user name is `cathy`,`user.home` defaults to

`C:\Winnt\Profiles\cathy`	Multiuser Windows NT
`C:\Windows\Profiles\cathy`	Multiuser Windows 95/98/ME

However, it's always possible that the `user.home` value has been modified: for example, by a system administrator when creating a user account. The following program demonstrates how to determine what the `user.home` value is:

```
public class getUserHome {

    public static void main(String[] args) {
        System.out.println("user.home is " +
                    System.getProperty("user.home"));
    }
}
```

12.3.3 Setting Security Properties

Some aspects of Java security may be customized by setting security properties. As is the case for system properties, a security property may be set statically or dynamically. To set a security property *statically*, add a line to the *security properties file*. By default, the security properties file shipped with the Java 2 release is installed at

```
<java.home>/lib/security/java.security     [Solaris,Linux]
<java.home>\lib\security\java.security     [Win32]
```

where `<java.home>` refers to the directory where the JRE software is installed. However, the location of the security properties file to be used can be specified by setting the `java.security.properties` system property. This feature first appeared in Sun's J2SDK 1.4 implementation. Section 12.4.1 describes this in greater detail.

In the default security properties file, the defaults for various security properties are initially set. You can modify or add to the contents of the file in order to change security property values. To specify a security property value in the security properties file, add a line of the following form:

```
propertyName=propertyValue
```

For example, as described in Section 12.6.5, the value of the security property named `policy.provider` specifies the canonical name of the `Policy` implementation class to be installed. The default implementation from Sun Microsystems is specified in the security properties file as follows:

```
policy.provider=sun.security.provider.PolicyFile
```

In order to specify that a different `Policy` implementation be used, change the value to indicate a different class that extends the `Policy` class, as in the following:

```
policy.provider=com.example.MyPolicy
```

To set a security property *dynamically*, call the `java.security.Security.setProperty` method, substituting the appropriate property name and value:

```
Security.setProperty("propertyName",
    "propertyValue");
```

For example, a `setProperty` call corresponding to the previous example for specifying the `com.example.MyPolicy` implementation would be

```
Security.setProperty("policy.provider",
      "com.example.MyPolicy");
```

12.4 Securing the Deployment

The previous sections supplied background information about installation and the various ways to set system and security property values. This section provides concrete details on how to customize a deployment.

Typically, the default installation, such as the preinstalled JRE on Solaris or the JRE bundled with Netscape Navigator, does not customize the security properties but rather relies on the default settings provided by the J2SE platform vendor. The J2SE platform vendor will generally select security properties that suffice for general-purpose deployments. However, the default security properties may not be appropriate when the deployment environment requires more stringent controls.

In order to secure a deployment, the first challenge is to ensure that the installation of the JRE is appropriately secure. A common best practice is to install the JRE on a file system or in a directory that only an administrator can modify. This could be a local file system or one accessible over the network. Once the installation is under administrative control, the next step is to customize the security properties file such that it conforms to the deployment objectives. The following sections describe critically important aspects of securing a deployment.

12.4.1 Restricting Property Override Mechanisms

J2SE 1.4 introduced the system property `java.security.properties`, which when specified enables an application to override or append security properties. The option to override versus append is determined by whether a double or single equals sign is supplied, respectively. We use this syntactic convention whenever a system property is used to override or augment a security property.

For instance, the following augments the settings in the security properties file:

```
java -Djava.security.properties=someURL someApp
```

In this case, where a single equals sign precedes the security properties file URL, the specified file will be used in addition to the installed security properties file. That is, the properties specified in the alternative file will be merged with and take precedence over the properties in the default security properties file. Here, *someURL* is a URL specifying the location of the additional security properties file. The URL can be any regular URL or simply the name of a security properties file in the current directory

This mechanism is typically used when multiple instances of the Java runtime share a common installation yet need to have instance-specific settings in the security properties file. Of course, not all deployments need or desire this flexibility. Thus, the security property `security.overridePropertiesFile` can be set to `false` to inhibit the override mechanism. By default, this property is set to `true`.

12.4.2 Configuring Application-Specific Policies

The policy files specified in the security properties file, as described in Section 12.6.1, are systemwide in that this same set of policy files will be used when running any applet or application. When invoking an application, you may specify an additional or a different policy file to be used. This can be done via the `-Djava.security.policy` command line argument, which sets the value of the `java.security.policy` system property, as in the following example:

```
java -Djava.security.manager -Djava.security.policy=someURL
someApp
```

Here, *someURL* is a URL specifying the location of another policy file. In this case, where a single equals sign precedes the policy file URL, the specified policy file will be used in addition to all the policy files specified in the security properties file. The URL can be an absolute URL or simply the name of a policy file in the current directory.

The `-Djava.security.manager` argument ensures that the default security manager is installed so that the application is run with a security policy in effect. This option is not required if the application *someApp* itself installs a security manager (Section 6.1.2).

Note the double equals sign in the following command:

```
java -Djava.security.manager -Djava.security.policy==someURL
someApp
```

When a double equals sign precedes the policy file URL, *only* the specified policy file will be used; all others will be ignored.

When running applets using `appletviewer`, you may specify the policy by using the `-Djava.security.policy` argument, with one or two equals signs, as appropriate. Here is an example:

```
appletviewer -J-Djava.security.policy=someURL someApplet
```

Note: The policy file value given in the `-Djava.security.policy` option will be ignored if the `policy.allowSystemProperty` security property value is set to `false`. This property is by default set to `true`.

12.5 Installing Provider Packages

The term *provider* refers to a package or set of packages that supplies concrete implementations of a subset of the cryptography or other security services defined as part of the Java 2 SDK Security API or one of its extensions (optional packages). For more information on providers, see Sections 10.2 and 10.3

Several types of services can be implemented by provider packages: digital signature algorithms, encryption algorithms, message digest algorithms, key generation and conversion services, and so on. Some of these services are described throughout this book, especially in Chapter 10.

The Java 2 runtime comes with several providers from Sun Microsystems already installed. One of these is the SUN provider, which includes implementations of the Digital Signature Algorithm (DSA), DSA key pair and algorithm parameter generators, a `KeyStore` for handling keystores of the proprietary JKS keystore type, and so on. Another is the SunJCE provider, which supplies further cryptography services, including implementations of various encryption and related algorithms.

An application may request a particular type of object implementing a particular service and get an implementation from one of the installed providers. For example, as described in Section 10.4.4, the way to request a `Signature` object for a particular type of signature algorithm is to call a `getInstance` static factory method on the `Signature` class and specify the algorithm. An example is

```
Signature dsa = Signature.getInstance("SHA1withDSA")
```

When this call is made, the installed providers are searched until one with a `SHA1withDSA` algorithm is found, and that provider's implementation is used. If desired, a program may request an implementation from a *specific* provider: for example, if you want to use a provider whose implementation has received government certification. To request a specific provider's implementation, call the `getInstance` method that includes a provider argument. An example is the following, which requests a `SHA1withDSA` signature algorithm implementation from the `"SUN"` provider:

```
Signature dsa = Signature.getInstance("SHA1withDSA", "SUN")
```

End users are free to install whatever provider implementations they wish. The installed providers may be updated at any time, transparently to applications: for example, when faster or more secure versions are available.

This section explains how to install providers. Installing a provider consists of two parts: installing the provider package classes and configuring the provider to include it in the list of providers that are searched when services are requested.

12.5.1 Installing the Provider Classes

A provider must be installed before it can be used. If a requested provider is not installed, a `java.security.NoSuchProviderException` is thrown, even if a different

installed provider implements the requested algorithm. There are two ways to install provider classes:

1. Place a zip or JAR file containing the classes anywhere on your `CLASSPATH`.
2. Supply your provider JAR file as an installed or bundled extension (optional package). See the Java 2 SDK documentation for information on extensions.

12.5.2 Configuring the Provider

The next step is to configure the provider, that is, to add the provider to a list of providers that will be searched when services are requested. You can do this statically or dynamically. To install a provider *statically*, edit the Java security properties file (Section 12.3.3). One type of property that can be set is the property defining the provider master class:

```
security.provider.n=masterClassName
```

This declares a provider and specifies its preference order, *n*. The preference order is the order in which providers are searched for requested algorithms or other services when no specific provider is requested. The order is 1-based; 1 is the most preferred, followed by 2, and so on.

The *masterClassName* specifies the canonical name of the provider's *master class*. The provider's documentation will specify what you should put as the `masterClassName`. This class must be a subclass of the `java.security.Provider` class, and its constructor sets the values of various properties that are required for the Java 2 Security APIs to look up the algorithms or other facilities implemented by the provider. Suppose that the master class is `COM.abcd.provider.Abcd`. To configure `Abcd` as the third provider in the preference order, add the following line to the security properties file:

```
security.provider.3=COM.abcd.provider.Abcd
```

Alternatively, a provider can be registered *dynamically* by calling either the `addProvider` or the `insertProviderAt` method in the `java.security.Security` class. This type of registration is not persistent and can be done only by programs that are granted sufficient permissions. In order to call either of these methods, a `java.security.SecurityPermission` with target name `"insertProvider.providerName"` is required, where `providerName` is replaced by the provider name assigned by the provider vendor. Alternatively, an `*` can be used instead of the provider name to indicate that any provider can be added.

For example, the following entry in Sun's policy syntax specifies that code loaded from a signed JAR file relative to the `/home/sysadmin/` directory on the local file system may call methods in the `Security` class to add or remove providers. The JAR file's signature must be verified using the public key referenced by the alias `sysadmin` in the relevant keystore. Policy files and the default policy file format are described next:

```
grant signedBy "sysadmin", codeBase "file:/home/sysadmin/*"
{
   permission java.security.SecurityPermission
                  "insertProvider.*";
   permission java.security.SecurityPermission
                  "removeProvider.*";
};
```

12.6 Policy Configuration

If a security manager is installed but no security policy is specified for running applets or applications, the JRE will default to a sandbox security model. To utilize fully the Java 2 security model, described in Chapters 3 through 7, a security policy should be crafted indicating which security-sensitive resource accesses are permitted. The security policy to be enforced must also be specified to the JRE.

The design of the `Policy` API does not mandate how a security policy is expressed externally to the Java runtime system. Thus, a `Policy` class implementation is free to specialize where and how policy information is stored: for example, in a database, a directory service, a file system, or other location. The default J2SE `Policy` implementation supports the specification of a security policy in one or more plain ASCII text files, in a particular syntax. Obviously, policy files should be well protected from unauthorized access and disclosures.

Sections 12.4.2 and 12.6.1 explain how to indicate the locations of policy files that together make up the security policy to be used by the default `Policy` implementation. Section 12.6.2 describes the syntax used to express policy information in such policy files, and Section 12.6.3 provides examples. Section 12.6.4 explains how property expansion can be used to simplify the expression of security policy and security properties. Finally, Section 12.6.5 relates how to replace the `Policy` class implementation.

12.6.1 Configuring Systemwide and User-Specific Policies

The source location for the policy information used by the `Policy` object is up to the `Policy` class implementation. The default implementation, from Sun Microsystems, obtains its information from static policy configuration files, usually referred to as *policy files*. A policy file can be composed using a simple text editor.

The location of the default system policy file is at

```
<java.home>/lib/security/java.policy    [Solaris,Linux]
<java.home>\lib\security\java.policy    [Win32]
```

where `<java.home>` refers to the value of the `java.home` system property (Section 12.2). For example, if the `java.home` value is `/home/cathy` on a Solaris system or `C:\cathy` on a Windows system, the default system policy file is

```
/home/cathy/lib/security/java.policy    [Solaris, Linux]
C:\cathy\lib\security\java.policy       [Win32]
```

The default user policy file is located at

```
<user.home>/.java.policy                [Solaris, Linux]
<user.home>\.java.policy                [Win32]
```

where `<user.home>` refers to the value of the `user.home` system property (Sections 12.3.1 and 12.3.2).

The default `Policy` object is initialized the first time its `getPermissions` method is called or whenever its `refresh` method is called. Initialization involves parsing the policy files and then populating the `Policy` object with policy information. When the `Policy` object is

initialized, the system policy is loaded, followed by the user policy. If neither policy is present, a built-in sandbox policy is used.

Policy file locations are by default given in the security properties file, whose name and location are given in Section 12.3.3. The policy file locations are specified as the values of properties whose names are of the form `policy.url.n`, where *n* is a number. The values are URLs and thus always use forward slashes, even on Windows systems. For example, the default system and user policy files are defined in the security properties file as follows:

```
policy.url.1=${java.home}/lib/security/java.policy
policy.url.2=${user.home}/.java.policy
```

Here, `${java.home}` and `${user.home}` indicate the values of the `java.home` and `user.home` system properties, respectively. The special `${propName}` syntax for specifying property values is described in Section 12.6.4.

As an example of customization, to ignore the default user policy, delete or comment out the second line. To specify multiple policy files to form a composite security policy, provide multiple `policy.url.n` lines, each indicating a different URL. The content of all the designated policy files will be used to populate the `Policy` object.

Note that *n* in `policy.url.n` must start with 1 and be consecutive integers. The first policy file must be specified by `policy.url.1`, the second by `policy.url.2`, and so on, until there are no more policy files. If you specify, for example, `policy.url.1` and `policy.url.3` but not `policy.url.2`, `policy.url.3` is ignored.

It is possible to specify that another policy file should be used in addition to or instead of the policy files specified in the security properties file. That case is described in Section 12.4.2.

12.6.2 Default Policy File Format

The policy configuration files for a JRE installation specify the permissions—which types of system resource accesses—allowed for code from specified code sources and principals. In order for a program to be permitted to perform a secured action, such as reading or writing a file, it must be granted permission for that particular action. When the default `Policy` implementation is used, the permission must be granted by a grant entry in a policy file.[1]

[1] One exception is that code always automatically has permission to read files from its own `CodeSource` (see Section 5.2.1) and the subdirectories of that `CodeSource`. It does not need explicit permission to do so.

The syntax of a policy configuration file to be read by the default `Policy` implementation includes a list of entries. It may contain a single *keystore entry* and zero or more *grant entries*.

Keystore Entry

A keystore is a protected database of private keys and their associated digital certificates, such as X.509 certificate chains (Sections 8.1 and 8.2). The default keystore implementation in the J2SE implements the keystore as a file (Section 12.8.1). The `keytool` utility may be used to create and administer keystores. The keystore specified in a policy configuration file is used to look up the public keys of the signers specified in the grant entries of the file. A keystore entry must appear in a policy configuration file if any grant or permission entries specify signer aliases.

Only one keystore entry is allowed in the policy file; others after the first one are ignored. The entry may appear anywhere outside the file's grant entries and has the following syntax:

```
keystore "some-keystore-url", "keystore-type";
```

Here, "*some-keystore-url*" specifies the URL where the keystore is located. The "*keystore-type*" is optional and if present specifies the keystore type. The URL is typically relative to the policy file location. Thus, if the policy file being processed is the one specified in the security properties file (Sections 12.3.3 and 12.6.1 as

```
policy.url.1=http://foo.bar.com/fum/some.policy
```

and that policy file has an entry

```
keystore ".keystore";
```

the keystore will be loaded from

```
http://foo.bar.com/fum/.keystore
```

The keystore URL can also be given as absolute, such as

```
keystore "http://foo.bar.com/fum/.keystore";
```

A keystore type defines the storage and data format of the keystore information and the algorithms used to protect private keys in the keystore and the integrity of the keystore itself. The default type supported in the J2SE is a proprietary keystore type named JKS.

Grant Entries

A policy configuration file contains 0 or more grant entries. Code being executed is always considered to come from a particular code source, represented at runtime by an object of type `CodeSource`. The code source includes not only the location (URL) from which the code originated but also a reference to the certificate(s) containing the public key(s) corresponding to the private key(s) used to sign the code. Certificates in a code source are referenced by symbolic alias names from the specified keystore.

Each grant entry may specify optional `codeBase`, `signedBy`, and `principal` qualifiers. The `codeBase` and `signedBy` name/value pairs qualify which code is granted the specified permissions. To represent the set of certificates that may be part of a `CodeSource`, a policy file simply includes a list of signer names, after the `signedBy` keyword, that are aliases that map to the actual certificates via a keystore. The alias design is useful because certificates can be large and can contain binary data and unprintable characters, whereas a policy file should be easy to view and to edit. Each `grant` entry may also specify one or more `principal` fields indicating the list of principals that must be executing the code in order for the permission(s) to be granted.

Following is the basic format of a `grant` entry:

```
grant signedBy "signer-names", codeBase "URL",
      principal principal-class-name "principal-name",
      principal principal-class-name "principal-name",
      . . . {
```

```
   permission permission-class-name "target-name", "action",
      signedBy "signer-names";
   ...
   permission permission-class-name "target-name", "action",
      signedBy "signer-names";
};
```

A grant entry must begin with the keyword `grant`. In the preceding, italicized items represent variable values. The `signedBy`, `codeBase`, and `principal` fields are optional. All nonitalicized items must appear as is, although the case and order of the keywords are insignificant.

The absence of the `signedBy` field signifies "any signer." That is, whether the code is signed and by whom does not matter. Its value, when specified, is a comma-separated list of one or more aliases that are mapped, using the keystore, to certificates. When the `signedBy` value is a comma-separated string containing names of multiple signers, for example `"Adam,Eve,Charles"`, the relationship is AND, not OR. That is, the specified list means "signed by Adam and Eve and Charles."

A `codeBase` value indicates the code source location (URL); you grant the permission(s) to code from that location. The absence of a `codeBase` entry signifies "any code"; that is, where the code originates from does not matter.

The meaning of a `codeBase` value depends on the characters at the end. A `codeBase` with a trailing `/` matches all class files, not JAR files, in the specified directory. A `codeBase` with a trailing `/*` matches all files, both class and JAR files, contained in that directory. A `codeBase` with a trailing `/-` matches all files, both class and JAR files, in the directory and recursively all files and subdirectories relative to the specified directory.

Note that a `codeBase` value is always specified as a URL; thus, a forward slash `/` must always be used as the directory separator. For example, if the source location for code on a Windows system is `C:\somepath\api\`, the policy `codeBase` entry would look like this:

```
grant codeBase "file:/C:/somepath/api/" {
...
}
```

Each `principal` value specifies a principal-class-name/principal-name pair indicating a named principal of the specified class. The code must be considered to be executed by the specified principal(s) in order to be granted the permissions. The principal field is optional in that, if it is omitted, it signifies "any principals"; that is, it doesn't matter whether the code is considered to be executed by any principals or by which principals.

An informal BNF (Backus Naur Form) grammar for the policy file format is given next. Terms that are not capitalized are terminals.

```
PolicyFile --> PolicyEntry | PolicyEntry; PolicyFile
PolicyEntry -->    grant {PermissionEntry}; |
   grant SignerEntry {PermissionEntry} |
   grant CodebaseEntry {PermissionEntry} |
   grant PrincEntry {PermissionEntry} |
   grant SignerEntry, CodebaseEntry {PermissionEntry} |
   grant SignerEntry, PrincEntry {PermissionEntry} |
```

```
   grant CodebaseEntry, SignerEntry {PermissionEntry} |
   grant CodebaseEntry, PrincEntry {PermissionEntry} |
   grant PrincEntry, SignerEntry {PermissionEntry} |
   grant PrincEntry, CodebaseEntry {PermissionEntry} |
   grant SignerEntry, CodebaseEntry, PrincEntry
{PermissionEntry} |
   grant SignerEntry, PrincEntry, CodebaseEntry
{PermissionEntry} |
   grant CodebaseEntry, SignerEntry, PrincEntry
{PermissionEntry} |
   grant CodebaseEntry, PrincEntry, SignerEntry
{PermissionEntry} |
   grant PrincEntry, SignerEntry, CodebaseEntry
{PermissionEntry} |
   grant PrincEntry, CodebaseEntry, SignerEntry
{PermissionEntry} |
   keystore "url"
SignerEntry -->   signedBy (a comma-separated list of
strings)
CodebaseEntry --> codeBase (a string representation of a
URL)
PrincEntry --> OnePrincipal | OnePrincipal, PrincEntry
OnePrincipal --> principal [ principal-class-name ]
"principal-name"
PermissionEntry --> OnePermission | OnePermission
PermissionEntry
OnePermission --> permission permission-class-name
            [ "target-name" ] [, "action"]
            [, SignerEntry];
```

A permission entry must begin with the keyword `permission`. The *`permission-class-name`* specified after the word `permission` in the previous grammar would be a specific permission type, such as `java.io.FilePermission` or `java.lang.RuntimePermission`.

The *"target-name"* is required for all permission types. The *"action"* is optional for some permission types and required for others. For example, the `java.io.FilePermission` requires the target to specify the file and the action that specifies the permitted type of file access (`"read"` or `"write"` or both `"read"` and `"write"`, separated by a comma). An action is not required for permissions such as `java.lang.RuntimePermission` that restrict access just to a given target: You either do or do not have the permission specified by the *`"target-name"`*.

The `signedBy` name/value pair for a `permission` entry is optional. If present, it indicates a signed permission. That is, the `Permission` class itself must be in a JAR file that was signed by the entity referred to by the given alias(es) in order for the permission to be granted. For example, suppose that you have the following grant entry:

```
grant {
   permission Foo "foobar", signedBy "FooSoft";
}
```

This permission of type `Foo` is granted if the `Foo.class` permission is in a signed JAR file that was signed by the private key corresponding to the public-key certificate specified by the `"FooSoft"` alias.

This per permission signer field is included to prevent spoofing when a permission class does not reside with the Java runtime installation. For example, a copy of the `com.mypvr.PVRPermission` class can be downloaded as part of a remote JAR file, and the user policy might include an entry that refers to it. Because the archive is not long-lived, the second time that the `com.mypvr.PVRPermission` class is downloaded, possibly from a different Web site, the second copy absolutely must be authentic. The reason is that the presence of the permission entry in the user policy might reflect the user's confidence or belief in the first copy of the class bytecode.

We chose to use digital signatures to ensure authenticity rather than storing a hash value of the first copy of the bytecode and using it to compare with the second copy. We did this because the author of the `Permission` class can legitimately update the class file to reflect a new design or implementation.

Items in a permission entry must appear in the following order:

```
permission
permission-class-name
"target-name"
"action"
signedBy "signer-names"
```

A permission entry is terminated with a semicolon. Case is unimportant for the identifiers (`permission` and `signedBy`) but is significant for *`permission-class-name`*. The case sensitivity of the *`"target-name"`* and *`"action"`* fields is implementation dependent.

In the specification of a `java.io.FilePermission`, *`"target-name"`* is a file path. On a Windows system, whenever directly specifying a file path in a string—but not in a `codeBase` URL—you need to include two backslashes `\\` for each single backslash in the path, as in this example:

```
grant {
   permission java.io.FilePermission
     "C:\\users\\cathy\\foo.bat", "read";
};
```

This is necessary because the strings are processed by a tokenizer (`java.io.StreamTokenizer`), which allows `\` to be used as an escape character. For example, `\n` indicates a new line. The tokenizer requires two backslashes to be used to indicate a single backslash. After the tokenizer has processed the preceding file path string, in the process converting double backslashes to single backslashes, the result is `"C:\users\cathy\foo.bat"`.

12.6.3 Policy File Examples

This section offers several examples of policy expressions. The first example shows two grant entries in a policy file. As with Java programs, lines preceded with `//` are comments and are not interpreted:

```
// If the code is signed by "Duke", grant it read/write
access
// to all files in /tmp:
grant signedBy "Duke" {
   permission java.io.FilePermission "/tmp/*", "read,write";
};

// Grant everyone permission to read the java.vendor
property value:
grant {
   permission java.util.PropertyPermission "java.vendor",
"read";
};
```

Here are the contents of another sample policy file:

```
grant signedBy "sysadmin", codeBase "file:/home/sysadmin/*"
{
   permission java.security.SecurityPermission
      "insertProvider.*";
   permission java.security.SecurityPermission
      "removeProvider.*";
   permission java.security.SecurityPermission
      "putProviderProperty.*";
};
```

This example specifies that only code that satisfies the following two conditions will be allowed to add or remove providers or to set provider properties:

1. The code was loaded from a signed JAR file from within the `/home/sysadmin/` directory on the local file system.
2. The signature can be verified by using the public key referenced by the alias name `"sysadmin"` in the keystore.

Because the code source contains two components, `codeBase` and `signedBy`, and either or both components may be omitted in a policy file, the following policy also would be valid:

```
grant signedBy "sysadmin" {
   permission java.security.SecurityPermission
      "insertProvider.*";
   permission java.security.SecurityPermission
      "removeProvider.*";
};
```

This policy says that code that is signed by `"sysadmin"` can add or remove providers, regardless of the code location. Here is an example without a signer:

```
grant codeBase "file:/home/sysadmin/-" {
   permission java.security.SecurityPermission
      "insertProvider.*";
   permission java.security.SecurityPermission
      "removeProvider.*";
};
```

In this case, code that comes from the `/home/sysadmin/` directory or any child subdirectory on the local file system can add or remove providers. The code does not need to be signed.

Following is an example that does not mention `codeBase` or `signedBy`:

```
grant {
   permission java.security.SecurityPermission
      "insertProvider.*";
   permission java.security.SecurityPermission
      "removeProvider.*";
};
```

Under this security policy, any code, regardless of the code source or who is running the code, may add or remove providers. Obviously, this policy is too liberal for most situations.

The following includes a principal-based entry:

```
grant principal javax.security.auth.x500.X500Principal
"cn=Alice" {
   permission java.io.FilePermission "/home/Alice", "read,
write";
   };
```

This example permits any code executing as the `X500Principal` whose common name (`cn`) is `"Alice"` permission to read and write to `/home/Alice`. *Note:* `X500Principal`s are represented by X.500 Distinguished Names (Section 12.9).

The following example shows a grant statement with both code source and principal information:

```
grant codebase "http://www.games.com",
   signedBy "Duke",
   principal javax.security.auth.x500.X500Principal
"cn=Alice" {
      permission java.io.FilePermission "/tmp/games",
            "read, write";
   };
```

This example allows code downloaded from "www.games.com", signed by `"Duke"`, and executed by `"Alice"` permission to read from and write to the `/tmp/games` directory.

12.6.4 Property Expansion in Policy Files

To make policy configuration and specification easier, the J2SE allows property expansion both in policy files and in the security properties file. Property expansion is similar to expanding environment variables in a UNIX shell. When a string of the following form appears in a policy file or in the security properties file, it will be expanded to the value of the specified system property:

```
${some.property}
```

Suppose that you have

```
permission java.io.FilePermission "${user.home}", "read";
```

This entry, when processed, expands `${user.home}` to the value of the `user.home` system property. If that property's value is `/home/cathy`, the previous permission entry is equivalent to

```
permission java.io.FilePermission "/home/cathy", "read";
```

To assist in the creation of platform-independent policy files, the special notation `${/}` is recognized as a shortcut for `${file.separator}`. Thus, you can write such permission entries as

```
permission java.io.FilePermission "${user.home}${/}*",
"read";
```

If you are using a Solaris system and the value of the `user.home` system property is `/home/cathy`, the previous entry gets expanded to

```
permission java.io.FilePermission "/home/cathy/*", "read";
```

If you are using a Windows system and the `user.home` system value is `C:\users\cathy`, the expansion result is

```
permission java.io.FilePermission "C:\users\cathy\*",
"read";
```

As a special case, if a property in a `codeBase` string, such as `grant codeBase "file:${java.home}/lib/ext/"` is expanded, any file-separator characters in the `codeBase` value will automatically be expanded, or converted, to forward slash (`/`) characters. The reason is that `codeBase` values are URLs and should always have forward slashes. Thus, if the sample grant entry is used on a Windows system, the expansion result is as follows, even if `java.home` is set to `C:\j2sdk1.4.0\jre`:

```
grant codeBase "file:C:/j2sdk1.4.0/jre/lib/ext/"
```

Thus, you don't need to use the notation `${/}` in `codeBase` strings, and you shouldn't.

Because property expansion can take place anywhere that a double-quoted string is allowed in the policy file, the fields *`"signer-names"`*, *`"URL"`*, *`"target-name"`*, and *`"action"`* can all be expanded. You can disable property expansion by setting to `false` the value of the `policy.expandProperties` security property. The default value of this property is `true`. (See Section 12.3.3 for information on setting the values of security properties.)

Nested properties are not supported and do not expand properly. For example, `"${user.${foo}}"` does not result in `"${user.home}"`, even if the `foo` property is set to `home`. The reason is that the property parser does not recognize nested properties. Rather, it simply looks for the first `${` and then keeps looking until it finds the first `}`. It tries to interpret the result, in this case, `${user.$foo}`, as a property but fails when there is no such property.

If a property expansion is given in a grant entry and property expansion fails, the entry is ignored. For example, suppose that the system property `foo` is not defined and you have the following:

```
grant codeBase "${foo}" {
   permission ...;
```

```
   permission ...;
};
```

All the permissions in this grant entry are ignored. On the other hand, suppose that you have the following:

```
grant {
   permission Foo "${foo}";
   permission Bar "barTarget";
};
```

Only the `permission Foo` entry is ignored, and `permission Bar` is granted.

If the property file has a keystore entry `keystore "${foo}"` and the system property `foo` is not defined, the keystore entry is ignored.

Expansion of a property in a string takes place after the tokenizer has processed the string. Thus, for string `"${user.home}\\foo.bat"`, the tokenizer first processes the string, converting the double backslashes to a single backslash, and the result is `"${user.home}\foo.bat"`. Then `${user.home}` is expanded, and the end result is `"C:\users\cathy\foo.bat"`, assuming that the `user.home` value is `C:\users\cathy`. To achieve platform independence in this example, the string should initially be specified without any explicit slashes, that is, by using the `${/}` property instead, as in `"${user.home}${/}foo.bat"`.

12.6.5 Configuring an Alternative `Policy` Class Implementation

An alternative `Policy` class implementation can be specified to replace the default `Policy` class. The security property `policy.provider` is used to specify the `Policy` class. For example, the `policy.provider` property is set in the security properties file (see Section 12.3.3) as follows:

```
policy.provider=PolicyClassName
```

where *`PolicyClassName`* indicates the canonical name of the desired `Policy` implementation class. The default security properties file entry for this property is

```
policy.provider=sun.security.provider.PolicyFile
```

By changing the property value to specify another class, you substitute a new `Policy` class, as in

```
policy.provider=com.example.MyPolicy
```

When the `Policy` object is to be initialized, this class is used rather than the default implementation class, `PolicyFile`.

12.7 JAAS Login Configuration Files

As noted in Chapter 7, the Java Authentication and Authorization Service, initially an optional package and subsequently integrated into J2SDK 1.4, can be used for user authentication and authorization. JAAS authentication is performed in a pluggable fashion, so applications can

remain independent from underlying authentication technologies. A system administrator determines the authentication technologies, or `LoginModule`s, to be used for each application and configures them in a login configuration. The source of the configuration information, such as a file or a database, depends on the implementation of the `javax.security.auth.login.Configuration` class. The default implementation from Sun Microsystems reads configuration information from configuration files. This section describes such files.

12.7.1 Login Configuration File Structure and Contents

A login configuration file consists of one or more entries, each specifying which underlying authentication technology should be used for a particular application or applications. The structure of each entry is the following:

```
<name used by application to refer to this entry> {
   <LoginModule> <flag> <LoginModule options>;
   <optional additional LoginModules, flags and options>;
   };
```

Thus, each login configuration file entry consists of a name followed by one or more `LoginModule`-specific entries, where each `LoginModule`-specific entry is terminated by a semicolon, and the entire group of `LoginModule`-specific entries is enclosed in braces. Each configuration file entry is terminated by a semicolon. Following is an example:

```
Login {
    com.sun.security.auth.module.UnixLoginModule REQUIRED;
    com.abc.AbcLoginModule REQUIRED;
};
```

Here, the entry is named `Login`, and that is the name that an application would use to refer to this entry when instantiating a `LoginContext`. The name can be whatever name the application and configuration file developer wishes to use. (Here, the term "application" refers to whatever code does the JAAS login to authenticate the user.)

Each login configuration entry contains a list of login modules. Authentication proceeds down the list in the exact order listed, with the flag values controlling the overall behavior. The sample entry specifies that the `LoginModule`s to be used to do the user authentication are the `UnixLoginModule` in the `com.sun.security.auth.module` package and the `AbcLoginModule` in the `com.abc` package and that these `LoginModule`s are both required to succeed in order for authentication to be considered successful.

Each `LoginModule`-specific entry has the following subparts:

- `LoginModule`, a class implementing the desired authentication technology. Specifically, the class must be a subclass of the `LoginModule` class, which is in the `javax.security.auth.spi` package. A typical `LoginModule` may prompt for and verify a user name and password. Any vendor can provide a `LoginModule` implementation that you can use. Some implementations, such as `UnixLoginModule`, `KeyStoreLoginModule`, and `Krb5LoginModule`, are supplied with the JRE from Sun Microsystems, in the `com.sun.security.auth.module` package.

- `Flag`, a value indicating whether success of the preceding `LoginModule` is `REQUIRED`, `REQUISITE`, `SUFFICIENT`, or `OPTIONAL`. If there is just one `LoginModule`-specific entry, the flag for it should be `REQUIRED`.
- `LoginModule` options, values for any desired options in the specified `LoginModule` implementation. This space-separated list of values is passed directly to the underlying `LoginModule`. Options are defined by the `LoginModule` itself and control the behavior within it. For example, a `LoginModule` may define options to support debugging/testing capabilities. The correct way to specify options in the configuration file is by using a name/value pairing—for example, `debug=true`—where the option name, here `debug`, and value, `true`, should be separated by an equals sign.

Following is a description of the valid flag values:

- `REQUIRED.` The login module is required to succeed. Regardless of whether it succeeds or fails, however, authentication still proceeds down the login module list. It must continue, even when faced with failure, because aborting at that point would give potential attackers useful information, such as which module failed and why.
- `REQUISITE.` The login module is required to succeed. If it succeeds, authentication continues down the login module list. If it fails, control immediately returns to the application; authentication does not proceed down the login module list.
- `SUFFICIENT.` The login module is not required to succeed. If it does succeed, control immediately returns to the application; authentication does not proceed down the login module list. If it fails, authentication continues down the login module list.
- `OPTIONAL.` The login module is not required to succeed. If it succeeds or fails, authentication still proceeds down the login module list.

The overall authentication succeeds only if all `REQUIRED` and `REQUISITE` login modules succeed. If no `REQUIRED` or `REQUISITE` login modules are configured for an application, at least one `SUFFICIENT` or `OPTIONAL` login module must succeed.

12.7.2 Login Configuration File Location

The configuration file to be used can be specified in one of two ways:

1. **On the command line.** You can use a `-Djava.security.auth.login.config` interpreter command line argument to specify the login configuration file that should be used. For example, the following specifies that the configuration file is the `myjaas.config` file in the current directory:
2. `java -Djava.security.auth.login.config==myjaas.config MyApp`
3. **In the security properties file.** An alternative approach for specifying the location of the login configuration file is to indicate its URL as the value of a `login.config.url.`*n* property in the security properties file. (The security properties file is described in Section 12.3.3.) Here, *n* indicates a consecutively numbered integer starting with 1. Thus, if desired, you can specify more than one login configuration file by indicating one file's URL for the `login.config.url.1` property, a second file's URL for the `login.config.url.2` property, and so on. If more than one login configuration file is specified—that is, if n > 1—the files are read and concatenated into a single configuration.

Here is an example of what would need to be added to the security properties file in order to indicate the `samplejaas.config` login configuration file. This example assumes that the file is in the `C:\AcnTest` directory on a Win32 system:

```
login.config.url.1=file:/C:/AcnTest/samplejaas.config
```

(Note that URLs always use forward slashes, regardless of what operating system the user is running.)

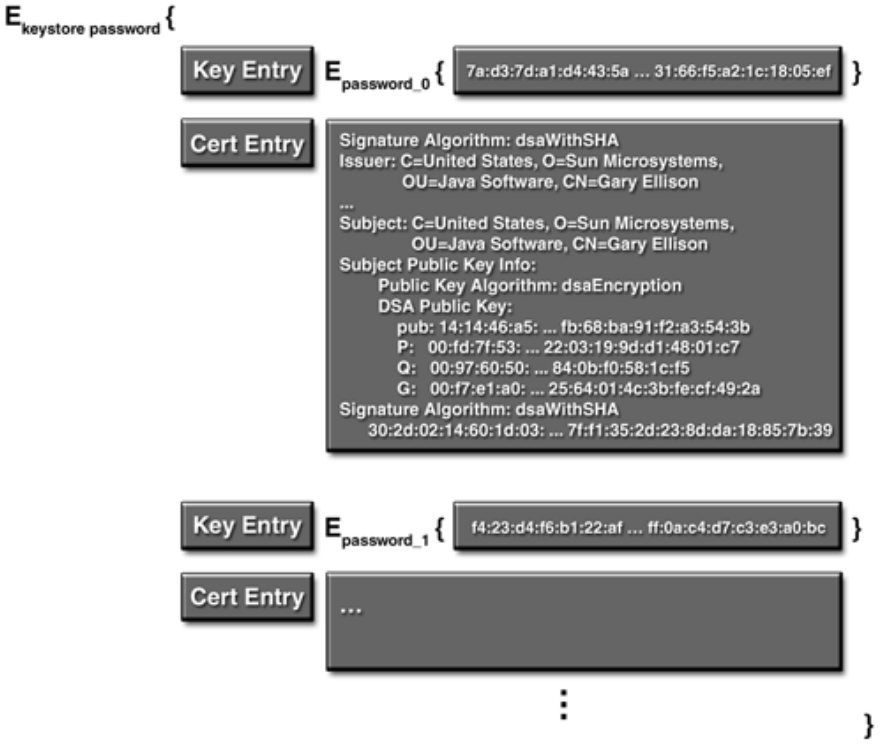
12.8 Security Tools

To assist developers, the Java 2 SDK is delivered with these security tools: the `keytool` and `jarsigner` command line tools and the `Policy Tool` utility, which is a graphical tool invoked via the `policytool` command. We first describe keystore databases that may be used by the tools and then the `keytool` and `jarsigner` security tools. The use of the `Policy Tool` is not covered in this book.

12.8.1 Keystore Databases

Recall from Section 12.6.2 that a keystore is a protected database that holds certificates and private keys. The default `Keystore` implementation implements the keystore as a file, as depicted in Figure 12.1. Access to a keystore is guarded by a password, which is chosen at the time the keystore is created. A keystore so protected can be changed only by someone who can provide the correct password. In addition, each private key in a keystore can be guarded, for extra security, by its own password. In the figure, E depicts an encryption function, the data in braces represents the data protected, and the E subscript indicates the password required to access the data.

Figure 12.1. Keystore



Information from a keystore may be used by the security tools. For example, the `jarsigner` tool may use a keystore to generate or verify digital signatures for JAR files. A JAR file packages class files, images, sounds, and/or other digital data in a single file. The `jarsigner` tool verifies the digital signature of a JAR file, using the certificate that comes with it, included in the signature block file of the JAR file. The tool then checks whether the public key of that certificate is trusted, that is, whether it is contained in the specified keystore.

A keystore contains two types of entries: key entries and trusted certificate entries. A *key entry* holds sensitive cryptographic key information and is stored in a protected format to prevent unauthorized access. Typically, a key stored in this type of entry is either a secret key or a private key accompanied by the certificate chain for the corresponding public key. *Note:* The `keytool` and `jarsigner` tools do not currently handle secret keys.

A *trusted certificate entry* contains a single public-key certificate belonging to an entity. It is called a *trusted certificate* because the keystore owner, by accepting this entry into the keystore, indicates trust that the public key in the certificate indeed belongs to the entity identified by the subject—that is, the owner—of the certificate. The issuer of the certificate vouches for this by signing the certificate.

All keystore entries—key and trusted certificate entries—are accessed via unique aliases. Aliases are case insensitive; for example, the aliases `Hugo` and `hugo` refer to the same keystore entry. You specify an alias when you add an entity to the keystore using the `keytool -genkey` command to generate a key pair—public and private key—or the `-import` command to add a certificate or certificate chain to the list of trusted certificates. Subsequent `keytool` commands must use this same alias to refer to the entry. For example, suppose that you use the alias `duke` in the following command:

```
keytool -genkey -alias duke -keypass dukekeypasswd
```

This command generates a new key pair, creates an X.509 self-signed, or signed by the private key, certificate containing the public key, and stores the private key and associated certificate in a new keystore entry identified by the alias `duke`. The command specifies an initial password of `dukekeypasswd`, which will be required by subsequent commands to access the private key associated with the alias `duke`. To change the private-key password of `duke`, you use a command like the following, which changes the password from `dukekeypasswd` to `newpass`:

```
keytool -keypasswd -alias duke -keypass dukekeypasswd -new
newpass
```

For better security, a password should not be specified on a command line or in a script unless for testing purposes or if you are on a secure system. If you do not specify a required password option on a command line, you will be prompted for one.

Recall that a keystore is, by default, implemented as a file. Each `keytool` command has an option for specifying the name and location of this persistent keystore file. During keystore creation (described in Section 12.8.2), if you do not specify a `-keystore` option, the keystore is, by default, stored in a file named `.keystore` in the user's home directory, as determined by the `user.home` system property. (Section 12.3.1 tells how to set system property values, and Section 12.3.2 tells what the default `user.home` property value is.)

The `KeyStore` class provided in the `java.security` package and described in Section 10.4.9 supplies interfaces for accessing and modifying the information in a keystore. Applications

can choose various types of keystore implementations from various providers, using the `getInstance` method supplied in the `KeyStore` class.

A keystore *type* defines the storage and data format of the keystore information, as well as the algorithms used to protect private keys in the keystore and the integrity of the keystore itself. Keystore implementations of different types need not be compatible in implementation details, such as format. The SDK default implementation of the keystore uses a proprietary keystore type named JKS. `String`s specifying types are not case sensitive; thus, `"jks"` would be considered the same as `"JKS"`.

The `keytool` tool works on any file-based keystore implementation, treating the keystore location passed to it at the command line as a file name and converting it to a `FileInputStream`, from which it loads the keystore information. The `jarsigner` and `Policy Tool` tools, on the other hand, can read a keystore from any location that can be specified using a URL.

For `keytool` and `jarsigner`, you can specify a keystore type at the command line, via the `-storetype` option. If you don't explicitly specify a keystore type, the tools choose a keystore implementation based simply on the value of the `keystore.type` security property. (Setting security property values is described in Section 12.3.3.) For example, if you have a provider package that supplies a `KeyStore` implementation for a keystore type called pkcs12, you can include the following line in the security properties file to indicate that keystore type:

```
keystore.type=pkcs12
```

Each tool gets the `keystore.type` value and then examines all the currently installed providers until it finds one that implements keystores of that type. The tool then uses the `KeyStore` implementation from that provider.

The `KeyStore` class defines a static method, `getDefaultType`, that lets programs retrieve the value of the `keystore.type` property. The following line of code creates a `KeyStore` instance that handles keystores of the default keystore type:

```
KeyStore keyStore =
KeyStore.getInstance(KeyStore.getDefaultType());
```

12.8.2 `keytool`

The `keytool` utility is used to create and manage a keystore of private keys and their associated X.509 certificate chains authenticating the corresponding public keys. The utility can also manage certificates from trusted entities. It can display, import, and export X.509 v1, v2, and v3 certificates stored as files and can generate new, self-signed v1 certificates.[2]

[2] Even though the underlying certificate package supports X.509 v3 format, `keytool` generates only X.509 v1–formatted certificates due to command line complexity in dealing with various extensions and options. One can easily imagine extended or customized `keytool`s that take advantage of the v3 format.

The `keytool` utility allows users to specify any key-pair generation and signature algorithm supplied by any of the cryptographic service providers that are registered, or configured, with the Java runtime environment. (See Section 10.3 for information about providers and Section 12.5 for information on installing and configuring providers.) The default key-pair generation algorithm is Digital Signature Algorithm (DSA). The size of a DSA key must be in the range 512 to 1,024 bits and must be a multiple of 64. The default key size for any algorithm is 1,024 bits. The signature

algorithm is derived from the algorithm of the underlying private key. For example, if the underlying private key is of type DSA, the default signature algorithm is SHA1withDSA; if the underlying private key is of type RSA, the default signature algorithm is MD5withRSA.

All `keytool` command and option names are preceded by a minus sign (`-`). The options for each command may be provided in any order. Refer to the on-line SDK documentation for a detailed explanation of all commands and command options.

Generating a Key Pair

Following is a sample X.500 Distinguished Name (DN) string (see Section 12.9):

```
"CN=Gary Ellison, OU=Java Software, O=Sun, L=Santa Clara,
S=CA, C=US"
```

You can use the following command to generate a key pair—public key and associated private key—for the entity specified by this DN and to assign the resulting key entry the alias `mark`:

```
keytool -genkey -dname "CN=Gary Ellison, OU=Java Software,
O=Sun,
L=Santa Clara, S=CA, C=US" -alias mark
```

Note: The command must be typed on a single line. Multiple lines are used here for legibility.

When it generates a key pair, `keytool` creates a key entry in the keystore and populates it with the private key and a certificate containing the public key associated with the private key. The certificate is self-signed. That is, a signature is generated for the certificate, using the private key corresponding to the public key in the certificate.

Certificate Signing Requests and Certificate Chains

When a public/private key pair is first generated by `keytool`, the resulting key entry starts off containing the private key and a single certificate, a self-signed certificate. A certificate is more likely to be trusted by others if it is signed by a Certification Authority (CA). To get such a signature, you first generate a *Certificate Signing Request* (CSR), via a `-certreq` command, such as the following:

```
keytool -certreq -alias joe -file JoeJ.csr
```

This command creates a CSR for the entity identified by the alias `joe` and puts the request in the file named `JoeJ.csr`. Submit this file to a CA, such as VeriSign, Inc. The CA will authenticate the information in the certificate describing the entity, usually offline, and then will return a *certificate reply* containing one or more certificates. Once you receive a certificate reply from a CA, you should import the reply, and the self-signed certificate will be replaced by a chain of certificates.

If the reply contains multiple certificates—a *certificate chain*—at one end of the chain is the end-entity certificate that is essentially the same as the certificate sent to the CA except that the returned certificate has been signed by the CA, thereby binding the subject of the certificate with the public key being certified. The next certificate in the chain binds the CA's public key. Often, this is a self-signed certificate and also the last certificate in the chain. In other cases, the CA might return a longer chain of certificates. (Sections 8.1 and 8.2 provide comprehensive coverage of certificates and certification paths, respectively.)

Many CAs return only the issued certificate, with no supporting chain, especially when the hierarchy is flat, that is, without intermediate CAs. In this case, the certificate chain to replace the original self-signed certificate must be established—by `keytool` when the reply is imported—from trusted certificate information already stored in the keystore. A single-certificate reply, as well as the alternative reply format, which is defined by the PKCS #7 standard and includes the supporting certificate chain in addition to the issued certificate, can both be handled by `keytool`.

Importing Certificates

You can import certificates by using the `-import` command, specifying the name of the file containing the certificate(s) to be imported. A sample import command is the following:

```
keytool -import -alias joe -file jcertfile.cer
```

This command imports the certificate(s) in the file `jcertfile.cer` and stores them in the keystore entry identified by the alias `joe`. You can import either a single certificate to be considered a trusted certificate or a certificate reply received from a CA as the result of submitting a CSR to that CA. Which is imported is indicated by the value of the `-alias` option. If the alias exists in the database and identifies an entry with a private key, it is assumed that you are importing a certificate reply. In that case, the `keytool` utility checks whether the public key in the certificate reply matches the public key stored with the alias. If, on the other hand, the alias identifies an existing trusted certificate entry, the new certificate will not be imported. Otherwise, a new trusted certificate entry with the specified alias will be created and associated with the newly imported trusted certificate.

Before you import a certificate to your keystore as a trusted certificate entry, you should ensure its authenticity. For example, suppose that a certificate is in a file named `/tmp/cert`. Before you consider adding the certificate to your list of trusted certificates, you could execute a `-printcert` command to view its fingerprints. Here is an example of such a command and the resulting printout:

```
keytool -printcert -file /tmp/cert
Owner: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
Issuer: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
Serial Number: 59092b34
Valid from: Thu Sep 25 18:01:13 PDT 2002 until: Wed Dec 24
17:01:13
   PST 2002
Certificate Fingerprints:
MD5: 11:81:AD:92:C8:E5:0E:A2:01:2E:D4:7A:D7:5F:07:6F
SHA1:
20:B6:17:FA:EF:E5:55:8A:D0:71:1F:E8:D6:9D:C0:37:13:0E:5E:FE
```

In fact, before importing a certificate to the list of trusted certificates in the keystore, `keytool` prints out the certificate information and prompts you to verify it. You should do so by, for example, comparing the displayed certificate fingerprints with those obtained from another trusted source of information, which might be the most trusted CA itself. You then have the option of accepting the certificate as trusted or aborting the import operation.

When importing a certificate reply, the certificate reply is validated by using trusted certificates from the keystore and, optionally, using the certificates configured in the `cacerts` keystore file if the `-trustcacerts` option was specified for the `-import` command. (The `cacerts` file is described later.) If the reply is a single X.509 certificate, `keytool` attempts to establish a

trust chain, starting at the certificate reply and ending at a self-signed certificate belonging to a trusted CA. The certificate reply and the hierarchy of certificates used to authenticate the certificate reply form a new certificate chain that replaces the initial self-signed certificate in the key entry referred to by the alias.

If the reply is a certificate chain, the root (final) CA certificate in the reply is self-signed. However, the trust aspect of the root's public key does not come from the trust anchor itself, because anybody could generate a self-signed certificate with the appropriate DN. When you import such a certificate chain, `keytool` attempts to match the most trusted CA certificate provided in the reply with any of the trusted certificates in the keystore or in the `cacerts` keystore file, if you used the `-trustcacerts` option. If no match can be found, `keytool` prints out the information of the most trusted CA certificate, and you are prompted to verify it. As when verifying a trusted certificate entry you are importing, you then have the option of accepting the certificate or aborting the import operation.

Note: Certificates read by the `-import` and `-printcert` commands can be either in Base64 or binary encoded. Such encodings are briefly described in Section 8.1.3.

The `cacerts` Certificates File

A certificates file named `cacerts` resides in the security properties directory

```
<java.home>/lib/security          [Solaris, Linux]
<java.home>\lib\security          [Win32]
```

where `<java.home>` refers to the directory where the Java 2 runtime environment is installed, as described in Section 12.2.

The `cacerts` file represents a systemwide keystore with CA certificates. System administrators can configure and manage that file by using `keytool`, specifying `jks` as the keystore type. The `cacerts` keystore file ships with several most trusted CA certificates, including ones for VeriSign and Thawte.

The initial password of the `cacerts` keystore file is `changeit`. System administrators should change that password and the default access permission of that file on installing the SDK.

When importing certificates via the `keytool -import` command, you can include a `-trustcacerts` option to indicate that certificates in the `cacerts` file may be considered when establishing a chain of trust, as described in the `-import` command description. Here is a sample import command that includes such an option:

```
keytool -import -trustcacerts -alias joe -file jcertfile.cer
```

Exporting a Certificate

To export a certificate to a file, use the `-export` command, as in

```
keytool -export -alias jane -file janecertfile.cer
```

This command exports `jane`'s certificate to the file janecertfile.cer. By default, the command outputs a binary-encoded certificate, but it will instead output a Base64 certificate if you use the `-rfc` option.

Printing Keystore Contents

To print the contents of a keystore entry, use the `-list` command, as in

```
keytool -list -alias joe
```

If an alias is not specified, the contents of the entire keystore are printed. The command by default prints the MD5 fingerprint of a certificate. If the `-verbose` option is specified, it prints the certificate in human-readable format.

Help Commands

The following commands output help information:

```
keytool
keytool -help
```

Default Values for keytool Options

The `keytool` utility has the following built-in default values for the following options:

`-alias`	`mykey`
`-keyalg`	`DSA`
`-keysize`	`1024`
`-validity`	`90`
`-keystore`	the file named .keystore in the user's home directory
`-file`	`stdin` if reading; `stdout` if writing

`keytool` Usage Example

Following is an example of creating and managing a keystore that has your public/private key pair and certificates from entities you trust. First, you need to create a keystore and generate the key pair. You can use a command like the following, typed on a single line:

```
keytool -genkey -dname "cn=Mark Smith, ou=Accounting, o=Sun,
c=US"
   -alias business -keypass kpi135 -keystore
/working/mykeystore
   -storepass ab987c -validity 180
```

This command creates the keystore `mykeystore` in the /`working` directory, if it does not already exist, and assigns it the password (`storepass`) `ab987c`. It generates a public/private key pair for the entity whose DN (`dname`) has a common name `Mark Smith`, organizational unit `Accounting`, organization `Sun`, and two-letter country code `US`. It uses the default DSA key generation algorithm to create the keys, both 1,024 bits long, the default key size.

The command creates a self-signed certificate, using the default SHA1withDSA signature algorithm that is used when the key algorithm is DSA. The certificate includes the public key and the DN information. This certificate will be valid for 180 days and is associated with the private

key in a keystore entry referred to by the alias `business`. The private key is assigned the password `kpi135`.

The command can be significantly shorter if more option defaults are accepted, as you are prompted for any required values that are not specified and have no defaults. Thus, you could simply type the following:

```
keytool -genkey
```

In this case, a keystore with name `.keystore` in the user's home directory (Section 12.3.2) is used or created, if it doesn't exist yet, and an entry with alias `mykey` is created, with a newly generated key pair and a certificate that is valid for 90 days.

The rest of the examples in this section assume that you executed the `-genkey` command without options specified and that you responded to the prompts with values equal to those given in the `-genkey` command used at the beginning of this section. So far, all you have is a self-signed certificate. A certificate is more likely to be trusted by others if it is signed by a CA. To get such a signature, you first generate a CSR, using a command like the following:

```
keytool -certreq -file MarkS.csr
```

This command creates a CSR for the entity identified by the default alias `mykey` and puts the request in the file named `MarkS.csr`. You then submit this file to a CA. The CA will authenticate you as the requester, usually offline, and return a certificate, signed by it, authenticating your public key. In some cases, it will return a chain of certificates.

You need to replace your self-signed certificate with a certificate chain, where each certificate in the chain authenticates the public key of the signer of the previous certificate in the chain, up to the most trusted CA. Before you import the certificate reply from a CA, you need one or more trusted certificates in either your keystore or the `cacerts` keystore. You determine which certificates are needed as follows:

- If the certificate reply is a certificate chain, you need only the top certificate of the chain, that is, the most trusted CA certificate authenticating that CA's public key.
- If the certificate reply is a single certificate, you need a certificate for the issuing CA, the one that signed your certificate; if the issuing certificate is not self-signed, you need a certificate for its signer, and so on, up to a self-signed most-trusted CA certificate.

The default `cacerts` file in the JRE ships with a number of VeriSign and Thawte root CA certificates, so you probably will not need to import a VeriSign or Thawte certificate as a trusted certificate in your keystore. But if you request a signed certificate from a different CA and your keystore does not contain a certificate authenticating that CA's public key, you will need to import a trusted certificate from the CA.

A certificate from a CA is usually either self-signed or signed by another CA, in which case you also need a certificate authenticating that CA's public key. Suppose that company ABC, Inc., is a CA and you obtain a file named `ABCCA.cer` that contains a purportedly self-signed certificate from ABC, authenticating that CA's public key. Be very careful to ensure that the certificate is valid prior to importing it as a trusted certificate. For example, you could execute a `-printcert` command to view its fingerprints, as in the following:

```
keytool -printcert -file ABCCA.cer
```

You can then verify the certificate by, for example, comparing the displayed certificate fingerprints with those obtained from another trusted source of information, perhaps the CA itself. If you trust that the certificate is valid, add it to your keystore, using a `-import` command:

```
keytool -import -alias abc -file ABCCA.cer
```

This command creates a trusted certificate entry in the keystore, with the data from the file `ABCCA.cer`, and assigns the alias `abc` to the entry.

Once you have imported a certificate authenticating the public key of the CA to which you submitted your CSR or if such a certificate is already in your keystore or in the `cacerts` file, you can import the certificate reply, thereby replacing your self-signed certificate with a certificate chain. This is the chain returned by the CA in response to your CSR, if the CA reply is a chain, or one constructed, if the CA reply is a single certificate, using the certificate reply and trusted certificates that are already available in the keystore in which you imported the reply or in the `cacerts` keystore file.

For example, suppose that you sent your CSR to VeriSign. You can then import the reply by using the following command, assuming that the returned certificate is named `VSMarkS.cer`:

```
keytool -import -trustcacerts -file VSMarkS.cer
```

The `-trustcacerts` option indicates that the `import` command may search the `cacerts` file when searching for a certificate verifying VeriSign's public key.

Suppose that you have used the `jarsigner` tool to sign a JAR file (see Section 12.8.3). Clients who want to use the file will want to authenticate your signature.

One way they can do this is by first importing your public-key certificate into their keystore as a "trusted" entry. You can export the certificate and supply it to your clients. As an example, you can copy your certificate to a file named `MS.cer` via the following:

```
keytool -export -alias mykey -file MS.cer
```

Given that certificate and the signed JAR file, a client can use the `jarsigner` tool to authenticate your signature.

Suppose that your DN changes, perhaps because you have changed departments or moved to another city. You may still use the same public/private key while updating your DN. For example, suppose that your name is Susan Miller and that you created your initial key entry with the alias `sMiller` and this DN:

```
"cn=Susan Miller, ou=Finance Department, o=BlueSoft, c=us"
```

If you later change from the Finance Department to the Accounting Department, you can still use the previously generated public/private key pair but update your DN by following these steps:

1. Copy, or clone, your key entry:
2. `keytool -keyclone -alias sMiller -dest sMillerNew`

   This command will prompt for the `storepass` password and for the initial and destination private-key passwords, as they are not provided at the command line.

3. Change the certificate chain associated with the copy so that the first certificate in the chain uses your new DN. Start by generating a self-signed certificate with the appropriate department name:

```
4.  keytool -selfcert -alias sMillerNew -dname "cn=Susan
   Miller,
5.      ou=Accounting Department, o=BlueSoft, c=us"
```

6. Generate a Certificate Signing Request (CSR) by using the information in the new certificate:

```
7.  keytool -certreq -alias sMillerNew
```

8. Submit the CSR to a CA and import the CA certificate reply:

```
9.  keytool -import -alias sMillerNew -file
   VSSMillerNew.cer
```

10. You might want to remove the initial key entry that used your old DN:

```
11. keytool -delete -alias sMiller
```

12.8.3 `jarsigner`

The Java Archive (JAR) feature enables the packaging of class files, images, sounds, and other data in a single file for faster and easier distribution. A tool named `jar` enables developers to produce JAR files. The `jarsigner` tool can sign JAR files and verify the signatures and integrity of signed JAR files. Attaching a digital signature to a JAR file helps to ensure that its authenticity can be verified, by recomputing the signature based on the current JAR content and comparing it to the stored signature. If the two do not match, either the content or the signature in the JAR file was modified. Thus, as long as the private key is kept secret, someone without the private key cannot forge a signed JAR file.

Basic usage of the `jarsigner` tool follows, along with examples. Refer to the SDK on-line documentation for more information on `jarsigner`'s options.

Signing a JAR File

The `jarsigner` tool uses private key and certificate information from a keystore to generate the digital signatures for JAR files. Thus, when using `jarsigner` to sign a JAR file, you first must specify the keystore location as a URL, as well as the alias for the keystore entry containing the private key needed to generate the signature. For example, the following, typed on one line, will sign the JAR file named `MyJARFile.jar`, using the private key associated with the alias `duke` in the keystore named `mystore` in the `/working` directory:

```
jarsigner -keystore /working/mystore -storepass myspass
-keypass dukekeypasswd MyJARFile.jar duke
```

As no output file is specified, `jarsigner` overwrites `MyJARFile.jar` with the signed JAR file.

You can use a `-signedjar` option to specify the name to be used for the signed JAR file if you don't want the source file to be overwritten. For example, the following does not change `MyJARFile.jar` and instead creates the signed JAR file `MySignedJF.jar`:

```
jarsigner -keystore /working/mystore -storepass myspass
-keypass dukekeypasswd -signedjar MySignedJF.jar
MyJARFile.jar duke
```

Because keystores may be of different types, the `jarsigner` tool has an option

```
-storetype storetype
```

This option lets you indicate the type by replacing `storetype` in the preceding template with the type. If you do not explicitly specify a keystore type, `jarsigner` chooses a keystore implementation, based on the value of the `keystore.type` security property. This property, as with all security properties, can be set via the methods described in Section 12.3.3. The value can be obtained by calling either of the following:

```
java.security.KeyStore.getDefaultType()
java.security.Security.getProperty("keystore.type")
```

Currently, the SDK's default implementation of `jarsigner` can sign only zip files or JAR files created by the SDK `jar` tool.[3] If the signer's public and private keys are DSA keys, `jarsigner` signs the JAR file by using the SHA1withDSA algorithm. If the signer's keys are RSA keys, `jarsigner` will attempt to sign the JAR file by using the MD5withRSA algorithm. Providers must be statically installed with supporting implementations of these signing algorithms. The default SUN provider makes both SHA1withDSA and MD5withRSA signing algorithms available.

[3] JAR files are the same as zip files but also have a `META-INF/MANIFEST.MF` file. Such a file will automatically be created when `jarsigner` signs a zip file.

The Signed JAR File

As described in Section 8.3.2, when `jarsigner` is used to sign a JAR file, the output signed JAR file is exactly the same as the input JAR file, but it has two additional files placed in the `META-INF` directory: a signature instructions file, or signature file, with an `SF` extension and a signature block file whose extension varies, depending on the type of digital signature algorithm used. For example, the extension would be `DSA` if the DSA algorithm was used. The base file names for these two files come from the value of the `-sigFile` option. For example, suppose that the option appears as follows and that the DSA algorithm was used:

```
-sigFile MKSIGN
```

The files are named `MKSIGN.SF` and `MKSIGN.DSA`. If no `-sigfile` option appears, the base file name for the `SF` and `DSA` files is the first eight characters of the alias name specified on the command line—or the alias name itself, if it has fewer than eight characters—all converted to uppercase.

A signature instructions file (an `SF` file) looks similar to the manifest file that is always included in a JAR file when `jarsigner` is used to sign the file. That is, for each source file included in the JAR file, the signature file contains at least the file name, the name of the digest algorithm used, such as SHA, and a digest value. In the manifest file, the digest value for each source file is the digest (hash) of the binary data in the source file. In the `SF` file, however, the digest value for a given source file is the hash of the lines in the manifest file for the source file. The `SF` file also, by default, includes a header containing a hash of the whole manifest file. The presence of the header enables verification optimization, as described later in the chapter.[4]

[4] The signed header can also be used to assist in sealing a Java software package stored inside a JAR such that no other class can belong to the same package unless the other class is signed by the same signature key.

A signature is generated for the `SF` file and is placed in the signature block file: the `DSA` file in our example. The `DSA` file also contains, encoded within, the certificate or certificate chain from

the keystore that authenticates the public key corresponding to the private key used for signing. The `jarsigner` tool can use the certificate (chain) to verify the signature.

Multiple Signatures for a JAR File

A JAR file can be signed by multiple people simply by running `jarsigner` on the file multiple times, specifying the alias for a different person each time, as in the following command sequence:

```
jarsigner myBundle.jar susan
jarsigner myBundle.jar kevin
```

When a JAR file is signed multiple times, the resulting JAR file will contain multiple `SF` and signature block files, one pair of files for each signature. In the previous example, the output JAR file includes files with the following names if the DSA algorithm was used:

```
SUSAN.SF
SUSAN.DSA
KEVIN.SF
KEVIN.DSA
```

Verifying a JAR File

A sample command to verify the signature of the signed JAR file `MySignedJF.jar` is the following:

```
jarsigner -verify MySignedJF.jar
```

Successful JAR file verification occurs if the signature(s) are valid and none of the files in the JAR file when the signatures were generated have been changed since then. JAR file verification involves the following steps:

1. Verify the signature of the `SF` file itself. The verification ensures that the signature stored in each signature block file, such as one with extension `DSA`, was in fact generated using the private key corresponding to the public key whose certificate or certificate chain also appears in the `DSA` file. It also ensures that the signature is a valid signature of the corresponding signature (`SF`) file and thus that the `SF` file is tamper free.
2. Verify the digest listed in each entry in the `SF` file with each corresponding section in the manifest. The `SF` file may include a header containing a hash of the entire manifest file. When the header is present, the verification can simply check to see whether the hash in the header indeed matches the hash of the manifest file. If that is the case, verification proceeds to the next step. Otherwise, the hash in each source file information section in the `SF` file must be checked to determine whether it equals the hash of its corresponding section in the manifest file. The hash of the manifest file stored in the `SF` file header might not equal the hash of the current manifest file: for example, when one or more files are added to the JAR file, using the `jar` tool, after the signature and thus the `SF` file were generated. When the `jar` tool is used to add files, the manifest file is changed—sections are added to it for the new files—but the `SF` file is not. Given that the interest here is in only those signed files, a verification is still considered successful if signatures and hashes of these files verify.
3. Verify each file mentioned in the `SF` file. The `jarsigner` utility reads each file in the JAR file that has an entry in the `SF` file. While reading, it computes the file's digest and then compares the result with the digest for the file in the manifest section. The digests should be the same, or else verification fails. If any security-sensitive verification failures

occur during the verification process, the process is stopped, and a security exception that is caught and displayed by `jarsigner` is thrown.

Code Signing and Verification Example

Here is an example of signing and verifying a JAR file. Suppose that you have a JAR file, `bundle.jar`, that you want to sign by using the private key of the user whose keystore alias is `jane` in the keystore named `mystore` in the `/working` directory. Suppose that the keystore password is `myspass` and that the password for `jane`'s private key is `j638klm`. You can use the following command, on a single line, to sign the JAR file and name the signed JAR file `sbundle.jar`:

```
jarsigner -keystore /working/mystore -storepass myspass
-keypass j638klm -signedjar sbundle.jar bundle.jar JANE
```

The resulting `SF` and `DSA` files are `JANE.SF` and `JANE.DSA`.

To verify the signed JAR file `sbundle.jar`, you could use the following command:

```
jarsigner -verify sbundle.jar
```

If the verification is successful, the message `jar verified` displays. Otherwise, an error message appears. You can get more information about the verification process by using the `-verbose` option, as follows:

```
jarsigner -verify -verbose sbundle.jar


      198 Fri Sep 26 16:14:06 PDT 2001 META-INF/MANIFEST.MF
      199 Fri Sep 26 16:22:10 PDT 2001 META-INF/JANE.SF
     1013 Fri Sep 26 16:22:10 PDT 2001 META-INF/JANE.DSA
smk  2752 Fri Sep 26 16:12:30 PDT 2001 AclEx.class
smk   849 Fri Sep 26 16:12:46 PDT 2001 test.class


s = signature was verified
m = entry is listed in manifest
k = at least one certificate was found in keystore


jar verified.
```

If, when verifying, you specify the `-certs` option along with the `-verify` and `-verbose` options, the output includes

- Certificate information, including the certificate type, for each signer of the JAR file
- The signer's DN (see Section 12.9) information if, and only if, the certificate is an X.509 certificate
- The keystore alias for the signer, in parentheses, if the public-key certificate in the JAR file matches that in a keystore entry

Here is an example:

```
jarsigner -keystore mystore -verify -verbose -certs
myTest.jar


        198 Fri Sep 26 16:14:06 PDT 2001 META-
INF/MANIFEST.MF
        199 Fri Sep 26 16:22:10 PDT 2001 META-INF/JANE.SF
       1013 Fri Sep 26 16:22:10 PDT 2001 META-INF/JANE.DSA
        208 Fri Sep 26 16:23:30 PDT 2001 META-
INF/JAVATEST.SF
       1087 Fri Sep 26 16:23:30 PDT 2001 META-
INF/JAVATEST.DSA
smk    2752 Fri Sep 26 16:12:30 PDT 2001 Tst.class


X.509, CN=Test Group, OU=Java Software, O=Sun Microsystems,
L=CUP,
    S=CA, C=US (javatest)
X.509, CN=Jane Smith, OU=Java Software, O=Sun, L=cup, S=ca,
C=us
    (JANE)


s = signature was verified
m = entry is listed in manifest
k = at least one certificate was found in keystore


jar verified.
```

If the certificate for a signer is not an X.509 certificate, no DN information is available. In this case, just the certificate type and the alias are shown. For example, if the certificate was a PGP certificate and the alias was `bob`, you would get this as output:

```
PGP, (bob)
```

12.9 X.500 Distinguished Names

In some cases, values for various properties or options are expected to contain X.500 Distinguished Name (DN) strings. For example, the `keytool` command for generating a key pair requires a DN as the value of the `-dname` option indicating the DN of the entity for whom keys should be generated. Another example is that of a policy file principal entry that indicates a principal of type `javax.security.auth.x500.X500Principal`. The value for that principal must be specified as a DN. A final example is when X.500 Distinguished Names are used to identify entities for the subject and issuer (signer) fields of X.509 certificates.

A sample X.500 Distinguished Name (DN) string is the following:

```
"CN=Duke, OU=Java Software, O=Sun, L=Santa Clara, S=CA,
C=US"
```

The keywords `CN`, `OU`, and so on, are abbreviations for the following:

- `CN` = `commonName`
- `OU` = `organizationUnit`
- `O` = `organizationName`
- `L` = `localityName`, indicating a city
- `S` = `stateName`, indicating a state or province
- `C` = `country`, expected to be a two-letter country code

DN keyword abbreviations are case insensitive; for example, the following are all treated the same: `CN`, `cn`, and `Cn`. However, the order of the keywords does matter in that each subcomponent must appear in the designated order `CN`, `OU`, `O`, `L`, `S`, `C`. However, not all subcomponents need be present; subsets are allowed, for example,

```
"CN=Duke, OU=Java Software, O=Sun, C=US"
```

See [130] for more information about X.500 Distinguished Names.

12.10 Managing Security Policies for Nonexperts

This chapter has discussed the technical details of deploying the Java 2 security architecture, as well as how to configure security policies, create keys and certificates, and sign JAR files containing classes. The overall complexity might appear overwhelming to the nonexpert computer user. This complexity is the natural result of having a feature-rich security architecture that must cater to a wide range of needs, such as those that arise in programming secure enterprise applications.

Two approaches are useful for the nonexpert when dealing with this complexity. One is to call in professional care and management. In the case of an enterprise environment, system administrators and information resource departments can be made responsible for establishing and deploying security policies on behalf of other corporate employees. Technical details in this and preceding chapters have shown that the Java security architecture design has taken this into account and has introduced a number of ways for the user to defer or delegate security policy decisions to another party. For example, employees can be instructed to configure their browsers to point to a centrally controlled Web page to obtain the current security policy. Or the company might want to customize a version of the browser, which it then distributes to employees.

Developers of enterprise applications can also incorporate security policy management in such a way that the typical user does not have to deal with, or even be aware of, the underlying security management features. In the case of the individual outside the corporate environment, Internet service providers (ISPs) are also a good source for security advice and management. For example, many ISPs already offer limited security mechanisms, such as firewalls and junk-mail filtering. Thus, it is quite reasonable for them to offer security policy management help for executable content and mobile code.

The second approach to security management for the nonexpert is to focus on the human interface. Field experience and controlled studies have shown that it is extremely difficult for the vast majority of computer and Internet users to understand security issues, which range from terminology to solutions to consequences. Moreover, various users interpret things so differently that it is very difficult to describe security in the same way to a diverse group of people. Thus, the Java 2 platform has not attempted to provide a uniform human-computer interface to deal with security policy and management. Instead, it expects that software vendors will integrate such functionalities into their own system environments and customize the contents and presentations to suit the particular set of users of their systems.

Moreover, application developers can choose to embed security solutions in such a way that they are invisible to users. For example, imagine a Java application that provides AOL-style Internet access and user experience. Such an application might use many Java platform features, such as dynamic component upgrading, and provide such services as secure access to e-mail messages. Thus, the application will depend on extensive security technology, which calls for security management. In this case, the application can "lock in" the particular security policies that are needed to make it work and not provide any customization capability in this respect. As a result, apart from the initial login process, users do not have to deal with any further security issues and indeed might not even be aware that complicated security decisions are being made throughout the application.

Security management and user interface remains an understudied subject, partly because the Internet brought security into the mainstream for the first time, making it an everyday concern, and partly because older technologies have generally not had security as a design goal. As time goes by and extensive security solutions are deployed ubiquitously, developers will gain valuable insight into this important aspect of security technology.

Chapter 13. Other Platforms and Future Directions

All progress is precarious, and the solution of one problem brings us face to face with another problem.

—Martin Luther King, Jr.

In the first edition of this book, we concluded by describing industry trends that were both changing and advancing the application of Java technology. Today, Java technology, in one form or another, is woven into just about every conceivable device. No longer relegated to the desktop or the browser, Java technology is wired into smart cards, cell phones, pagers, digital communicators, personal digital assistants, set-top boxes, game consoles, desktop computers, network appliances, routers, switches, blade servers, mainframes, and supercomputers. We described a trend whereby multiple server applications would be executing concurrently within a JVM. This prediction has come to pass and is embodied in the specifications for Java 2 Platform, Enterprise Edition (J2EE). This in turn generated performance, scalability, and manageability requirements on Java virtual machine implementations. Today, high-performing JVMs, such as Java HotSpot, are widely deployed in mission-critical application environments.

We spoke of Java technology being embedded in smaller, less computationally capable devices. At the time, two early profiles of Java were beginning to take shape: PersonalJava Application Environment and EmbeddedJava Application Environment. These early adaptations of Java have since been replaced with more general adaptations that build on a common foundation and that have since been formalized and specified as profiles within Java 2, Micro Edition (J2ME). Even smaller Java application execution environments have gained widespread adoption with the success of Java Card technology.

In this chapter, we provide some general background on the various Java platforms, their advancement, and descriptions of their security models and capabilities. Where possible, we detail security technologies that are available or that are anticipated to emerge in the near term.

13.1 Introduction to Java Card

Java Card technology enables programs written in the Java programming language to run in smart cards and other memory-constrained devices. Java Card technology is widely deployed in many industry sectors, including mobile cellular handsets, health care identity cards, and financial services. Java Card technology consists of three components:

1. The Java Card Virtual Machine (JCVM) specification [126] defines a subset of the Java programming language and virtual machine specifications that is suitable for smart card applications.
2. The Java Card Runtime Environment (JCRE) specification [125] describes Java Card runtime environment behavior: memory management, application management, security enforcement, and other runtime features.
3. The Java Card Application Programming Interface (API) specification [124] describes the set of core and extension Java Card packages and classes for programming smart card applications.

These three components provide a secure Java platform for smart cards. The Java Card platform separates applications, called applets, from the proprietary technologies of smart card vendors and provides standard system and API interfaces for applets.

The Java Card specifications enable Java technology to run on smart cards and other devices with limited memory. The Java Card technology has several unique benefits, such as

- **Interoperability.** Applets developed with Java Card technology will run on any Java Card technology-based smart card, independently of the card vendor and underlying hardware.
- **Security.** Java Card technology relies on the inherent security of the Java programming language to provide a secure execution environment.
- **Multiapplication coexistence.** Java Card technology enables multiple applications to coexist securely on a single smart card. New applications can be installed securely after a card has been issued.
- **Compatibility with existing standards.** The Java Card API is compatible with international standards, such as ISO 7816 [55], for smart cards.

13.1.1 Virtual Machine Lifetime

In Java Card technology, the execution lifetime of the virtual machine is the lifetime of the card. Most of the information stored on a card is preserved even when power is removed from the card. Persistent memory technology enables a smart card to store information when power is removed. Because the VM and the objects created on the card are used to represent application information that is persistent, the JCVM appears to run forever. When power is removed, the VM stops only temporarily. When the card is next reset, the VM starts up again and recovers its previous object heap from persistent storage. Aside from its persistent nature, the Java Card virtual machine is just like the Java virtual machine.

In Java Card technology, the term *applet* refers to an applet written for the Java Card platform. An applet instance's lifetime begins when it is successfully registered with the Java Card Runtime Environment (JCRE) via the `javacard.framework.Applet.register` method. Applets registered with the `Applet.register` method exist until deleted by the applet deletion manager. The JCRE initiates interactions with the applet via the applet's public methods `install`, `select`, `deselect`, and `process`.

An applet should implement the static `install(byte[], short, byte)` method. If the `install` method is not implemented, the applet's objects cannot be created or initialized. When the applet is installed on the smart card, the static `install` method is called once by the JCRE for each applet instance created. The JCRE should not call the applet's constructor directly. The main task of the `install` method within the applet is to create an instance of the `Applet` subclass, using its constructor, and to register the instance.

Applets remain in a suspended state until they are explicitly selected. Selection occurs when the JCRE receives a `SELECT FILE` Application Protocol Data Unit (APDU) [55] command in which the name data matches the Application Identifier (AID) of the applet. Applet selection can also occur on a `MANAGE CHANNEL OPEN` command. Selection causes an applet to become the currently selected applet.

All APDUs are received by the JCRE and preprocessed. At any time during processing, the applet may throw a `javacard.framework.ISOException` with an appropriate status word, in which case the JCRE catches the exception and returns the status word to the card-accepting device (CAD), or card reader.

When the JCRE receives a `SELECT FILE` APDU command in which the name matches the AID of an applet, the JCRE calls the `deselect` method of the currently selected applet or, if it is multiply selected on more than one logical channel, its `Multiselectable.deselect` method. Applet deselection may also be requested by the `MANAGE CHANNEL CLOSE` command. The `deselect` method allows the applet to perform any clean-up operations that may be required in order to allow another applet to execute.

Power loss occurs when the card is withdrawn from the CAD or if there is some other mechanical or electrical failure. When power is reapplied to the card, and on card reset, the JCRE ensures that (1) transient data is reset to the default value, (2) the transaction in progress, if any, when power was lost or reset occurred, is aborted, and (3) the applet instance that was selected when power was lost, or reset occurred, becomes implicitly deselected. (*Note:* In this case, the `deselect` method is not called.) If the JCRE implements default applet selection, the JCRE will also ensure, when power is reapplied, that the default applet is selected as the active applet instance for the basic logical channel and that the default applet `select` method is called. Otherwise, the JCRE sets its state to indicate that no applet is active on the basic logical channel.

13.1.2 Java Card Remote Method Invocation

Java Card Remote Method Invocation (JCRMI), a subset of the Java Remote Method Invocation (RMI) system, provides a mechanism for a client application running on the CAD platform to invoke a method on a remote object on the card. The on-card transport layer for JCRMI is provided in the package `javacard.framework.service` by the class `RMIService` and is designed as a service requested by the JCRMI-based applet when it is the currently selected applet. The JCRMI message is encapsulated within the APDU object passed into the `RMIService` methods.

A transaction is a logical set of updates of persistent data. For example, transferring money from one account to another is a banking transaction. It is important for transactions to be atomic: Either all the data fields are updated, or none is. The JCRE provides robust support for atomic transactions, so that card data is restored to its original pretransaction state if the transaction does not complete normally. This mechanism protects against such events as power loss in the middle of a transaction and program errors that might cause data corruption should all steps of a transaction not complete normally.

13.1.3 Java Card's Applet Isolation and Object-Sharing Model

Isolation means that one applet cannot access the fields or methods of an applet in another context unless the other applet explicitly provides an interface for access. The Java Card firewall provides protection against the most frequently anticipated security concern: developer mistakes and design oversights that might allow sensitive data to be leaked to another applet. To obtain an object reference from a publicly accessible location, an applet must satisfy certain access rules before it can use the reference to access the object. The firewall also provides protection against incorrect code. If incorrect code is loaded onto a card, the firewall still protects objects from being accessed by this code.

The JCRE maintains its own JCRE context. This context is much like the context of an applet but has special system privileges so that it can perform operations denied to contexts of applets. At any time, only one JCRE or applet context is active within the VM. A context switch occurs when certain well-defined conditions are met during the execution of invoke-type bytecodes. During a context switch, the previous context and object owner information are pushed on an internal VM stack, a new context becomes the currently active context, and the invoked method executes in this new context. On exit from that method, the VM performs a restoring context switch. The original context of the caller of the method is popped from the stack and is restored as the currently active context. Context switches can be nested. The maximum depth depends on the amount of VM stack space available.

Any given object in the Java Card object space has a context and an owner associated with it. When a new object is created, it is associated with the currently active context and is owned by the applet instance that belongs to that currently active context. An object can be owned by an applet instance or by the JCRE.

The combined rules of context and object ownership within the firewall follow:

- Every applet instance belongs to a context.
- All applet instances from the same package belong to the same context.
- Every object is owned by an applet instance or the JCRE.
- An applet instance is identified by its AID.
- When executing an instance method of an object or a static class method called from within, the object's owner must be in the active context.

No runtime context check is performed when a class static field is accessed, and no context switch occurs when a static method is invoked. To enable applets to interact with each other and with the JCRE, some well-defined yet secure mechanisms are provided so one context can access an object belonging to another context:

- JCRE entry point objects
- Global arrays
- JCRE privileges
- Shareable interfaces

13.1.4 Java Card Security APIs

The Java Card API defines a security API, `javacard.security`, and a cryptography API, `javacardx.crypto`, which supply functionality similar to their J2SE counterparts, `java.security` and `javax.crypto`, respectively. Specifically, the APIs provide classes and interfaces that contain publicly available functionality for implementing a security and cryptography framework on Java Card. Classes in the `javacard.security` package provide the definitions of algorithms that perform the security and cryptography functions.

The cryptographic capabilities of the Java Card security API include support for symmetric and asymmetric signature algorithms, as well as key-agreement algorithms. The cryptography API supports key encryption and cipher abstractions similar to those found in J2SE.

13.2 Introduction to Java 2 Micro Edition

J2ME is a subset of J2SE and is further categorized by two configurations: Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC). Within a device configuration, a profile that specifies the available Java APIs is defined. For example, CLDC defines Mobile Information Device Profile (MIDP), which is targeted for mobile phones. Due to the limited computational resources available to these small devices, the granularity accorded to the security model and support mechanisms varies. CLDC 1.0 specifies a rather coarse security model similar to the sandbox of yore.

MIDP 2.0 defines a more advanced security model that incorporates the notion of trusted applications. In MIDP, applications are referred to as *MIDlet suites*. Once a MIDlet suite is determined to be trusted, access is allowed, based on the permissions accorded by the domain policy. The protection domain owner defines how the device authenticates and verifies a MIDlet suite so as to bind it to the protection domain. One such binding specified by MIDP 2.0 is an X.509 PKI-based mechanism for determining whether a MIDlet is trusted. Under this mechanism, which is similar to J2SE code signing, the authenticity and integrity of the MIDlet suite are verified, and, if successful, the MIDlet suite is placed into the corresponding protection domain.

The configurations and profiles of J2ME are not static, and the future holds much richer security functionality. Within the Java community, work is under way to define APIs for security and trust services. The API defines an abstraction layer to access and use a security element integrated into

the device. A typical security element would be a smart card, but other security devices are specifically not precluded.

J2ME-enabled devices are evolving to become trusted computing elements in a distributed environment. Advances in cryptography have made it possible for devices constrained by resource limitations, such as power, bandwidth, memory size, and computational horsepower, to perform cryptographic functions locally. Elliptic curve cryptography (ECC) has reached a level of maturity such that we can expect to see it implemented and deployed in small devices.

ECC offers cryptographic strength equivalent to the more prevalent cryptosystems in use today but with considerably smaller key sizes. This reduction in key size makes strong cryptography possible in resource-constrained environments. Specifically, we will soon be seeing J2ME devices with implementations of ECC. This will likely include the Elliptic Curve Digital Signature Algorithm (ECDSA) [2]. ECDSA can be used to provide applications with authentication and integrity protection mechanisms. More important, ECC-capable TLS cipher suites are in the process of being standardized by the IETF. Of course, for ECC cipher suites to be negotiated and used between communicating peers, the TLS implementation of the peer must also support the cipher suite. Thus, it is fair to expect such support to be available in J2SE shortly after the standards are codified.

As is the case with any conforming Java platform, the JVM specification requires that type safety be maintained and that invalid class files be rejected. This verification is traditionally performed by the Java class file verifier. However, implementing class file and bytecode verification is expensive in both time and space. The conventional verifier requires tens of kilobytes of code and memory to implement an iterative dataflow algorithm.

This is not an option in a resource-constrained device. To remedy the situation, CLDC specifies an alternative approach to verification, whereby the verification is split into two phases: off-device preverification and runtime verification. By offloading the computationally intense preverification step, the core runtime verification algorithm can be implemented in a few kilobytes of code and consume only a few hundred bytes of memory, and it requires only a linear scan of the bytecode. This saves both time and space at the expense of a slightly larger class file. To achieve this, the preverification step annotates the class file with special attributes referred to as stack maps. Stack maps describe the types of local variables and operand stack items. The information in the stack map facilitates the runtime verification step.

13.3 Security Enhancements on the Horizon for J2SE

The primary focus of this book has been J2SE platform security. Throughout this text, we have described the security features and the design underpinnings of J2SE. It would not be too great of a stretch to suggest that J2SE is possibly the most secure and feature-rich development and deployment environment ever created. However, that is no reason to sit idle and let the world pass by. We now detail possible enhancements to the Java platform.

13.3.1 Virtual Machine Enhancements

They say that good things come in small packages. The partitioning of preverification and runtime verification as specified by CLDC is a good example. The performance gain and the simplification this provides certainly warrant consideration for future inclusion into J2SE. To better integrate this into the development process,it is envisioned that the Java compiler (`javac`) will support the generation of the stack map and that the class file specification will be revised to support the necessary attributes to carry the information in a backward-compatible way.

13.3.2 Language Enhancements

J2SE provides a conservative and robust access control algorithm that can prevent some programming mistakes from turning into security holes. Additional techniques that further this effort are available. For example, recall that the `doPrivileged` primitive in a sense "enables" all permissions granted to a piece of code. This mechanism of asserting privilege is necessary and useful, but the common idiom of using anonymous classes for the `PrivilegedAction` argument adds complexity to the readability of the code. A more suitable language construct, such as a block construct with guaranteed prologue and epilogue execution semantics, would be simpler to understand. Additionally, closures could simplify implementing privileged code by relaxing the language requirements about variable usage within the scope of inner classes.

13.3.3 Trusted Computing Base Enhancements

In some cases, an application might want to enable only some of its granted permissions. This selective enabling further reduces the security impact of making a programming mistake. This reduction is possible today by supplying an `AccessControlContext` object to the `doPrivileged` method. However, this mechanism is cumbersome to use. We contemplate enriching the primitive so that it takes an additional parameter, possibly of type `Permission`, `PermissionCollection`, or `Permissions`, that specifies the permissions to be enabled. Alternatively, we envision simplifying the mechanics of reducing privileges by exploiting advances in declarative paradigms and policy management.

Another way to reduce security liabilities is to subdivide the system domain. For convenience, the system domain can be thought of as a single large collection of all system code. For better protection, system code should be run in multiple system domains, where each domain protects a particular type of resource and is given a special set of rights. For example, if file system code and network system code run in separate domains, with the former having no rights to the networking resources and the latter having no rights to the file system resources, the risks and consequences of an error or security flaw in one system domain are more likely to be confined within its boundary.

Moreover, protection domains currently are created transparently as a result of class loading. Providing explicit primitives to create a new domain might be useful. Often, a domain supports inheritance in that a subdomain automatically inherits the parent domain's security attributes, except when the parent further restricts or expands the subdomain explicitly.

Class loaders are very delicate in terms of their security implications, and the way the `ClassLoader` classes are specified can be improved. Applets and applications can create class loaders only if the system security policy is configured to allow this to happen, with the only exception being `URLClassLoader`. Such a severe restriction might impede the development of certain applications.

A way to handle nonclass content consistently is needed. When applets or applications are run with signed content—classes and other resources—the `JAR` and `Manifest` specifications on code signing allow a very flexible format. The classes within the same archive can be unsigned, signed with one key, or signed with multiple keys. Other resources within the archive, such as audio clips and graphic images, can also be signed or unsigned. However, it is unclear whether images and audio clips should be required to be signed with the same key if any class in the archive is signed. If images and audio files are signed with different keys, can they be placed in the same `appletviewer`, or browser page, or should they be sent to different viewers? Such questions are not easy to answer. Any response requires consistency across platforms and products in order to be most effective. The current approach is to process all images and audio clips whether or not they are signed. This temporary solution should be improved once a consensus is reached.

13.3.4 Evolution of Pluggable Security Providers

When we designed the Java security APIs, we envisioned an architecture that could withstand evolutionary changes and technological advances. We also realized that we could not implement everything; nor could we foresee the needs of everybody. Our solution to this realization was the Service Provider Interface, which has proved to be very versatile. The Service Provider Interface has been a boon to the Java security team, but even more important, it provides flexibility that third-party developers require.

Recall that J2SE has integral support for TLS, which is exposed to applications by the Java Secure Socket Extension (JSSE) API. JSSE relies primarily on the cryptographic algorithms plugged into Java Cryptography Architecture (JCA) to implement the various cipher suites. With the advent of standardized ECC cipher suites, we expect to see JCA cryptographic service providers implementing the underlying ECC signature and encryption algorithms. However, the implementation of JSSE must also provide implementations of the ECC cipher suites. Currently, JSSE does not expose a Service Provider Interface such that alternative implementations of the TLS/SSL protocol can be plugged in. Certainly this is a desirable capability, as it provides a degree of flexibility that is more amenable to integrating support for evolving standards. Thus, we anticipate an exposed Service Provider Interface so that alternative implementations can plug in under the JSSE API.

The Java GSS-API was developed and standardized in the security area of the IETF. Subsequent to its being standardized, Java GSS-API was incorporated into J2SE. Unlike the majority of the security APIs in J2SE, Java GSS-API does not expose a Service Provider Interface. Originally, we expected to standardize the Service Provider Interface within the IETF. As a standards body, however, the IETF is no longer interested in standardizing programming interfaces. Rather, the IETF prefers to expend its energy on Internet protocols. That said, the need for an SPI still exists for a variety of reasons. GSS mechanisms, such as SPKM [1], SPNEGO [4], and LIPKEY [33], have been standardized and are actively being deployed. Without an open programming interface, the Java GSS binding would eventually become obsolete. Given that, we expect a Java GSS-API Service Provider Interface to be standardized via the Java community process.

13.3.5 Security Expressions

At approximately the same time as Java 2 was released, another significant innovation began to mature: eXtensible Markup Language (XML) [17]. XML is essentially a language for describing data. The success of XML is the result of many factors, but possibly the most obvious is the one it shares with the Java programming language: ease of use. XML is a generalized markup language, and considerable standardization work is either complete or nearing completion for specifying XML syntax to describe and represent data that has been cryptographically secured. The XML Signature [32] and the XML Encryption [31] specifications are now stable. Java APIs are being created within the Java community for inclusion as core platform security APIs. Finally, the XML Key Management Specification (XKMS) [49] is also stabilizing. XKMS specifies an XML syntax and protocols for distributing and registering public keys. The design center for XKMS is to create simple abstractions and mechanisms to obscure the complexity of PKI. Around the notion of simplifying PKI, Java APIs are planned to support the protocol interaction with an XKMS trust service for retrieving public keys and for processing of XML Signature key information.

Once the basic XML security plumbing is in place, it can be applied to higher-level infrastructure, such as Web services. Industry research suggests that Web services will not be widely adopted until the security substrate is standardized. Many near-term standards efforts are active in this space. Java APIs are in the early standardization stages within the Java community process.

13.3.6 Security Management

Most will agree that the more difficult it is to do or use something, the more likely it will not be used. Two of the design tenets of J2SE are ease of use and ease of deployment. At times, these heuristics are at odds with security requirements. Obtaining platform, network, and/or information security is a complex undertaking. Many factors must be considered in order to deploy a system securely. The security posture of most distributions of J2SE defaults to a restrictive sandbox security model. This default posture is generally the safest stance a distribution can assume. Although safe, it is very limiting. To accommodate the deployment requirements of a system, we have crafted a variety of means by which to configure a system. However, the primitives are very low level. To manage the security of the environment better, specialized deployment paradigms have become prevalent. The three most common Java deployment paradigms are J2EE, Java Web Start, and the Java Plug-in.

13.4 Brief Introduction to Jini Network Technology

The design center for Jini Network Technology, which was introduced in 1999, is that of a dynamic adaptable network of available services [131]. Jini is based on the notion of a *federation*: a collection of users and the resources they rely on. The goals of the Jini distributed system are

- Enabling users to share services and resources
- Ease of access to the resources
- Location independence and mobility
- Simplification of the administration of users, network available services, and the devices that make up the network

One view of Jini is by its functional decomposition. Jini comprises an underlying infrastructure, a programming model, and the clients and services making up the federation.

The infrastructure component of Jini can be further subdivided into (1) the *discovery protocol*, which allows an entity wishing to join a Jini network to find a lookup service, and (2) the *lookup service*, which acts as a place where services advertise themselves and clients go to find a service. Jini infrastructure [123] is built on the Java RMI programming model.

Jini builds on the notion of a *proxy*, a local object that stands in for the remote object. While presenting the same programmatic interface to the local code, the proxy handles any network-related functions, transmitting any parameters to the remote service and receiving any return values from that service.

Obviously, there are many threats to distributed systems, and Jini is no different. In the following sections, we present an overview of the security architecture and components that make up the Davis project. The Davis project is an effort by the Jini technology project team at Sun Microsystems to address the need for a Jini technology security architecture, as well as to contribute other improvements to Jini technology.

13.4.1 Overview of Jini Technology Security Architecture

Jini fully leverages code mobility to enable the adaptive and dynamic behavior of a federated distributed system. The Jini technology security architecture ensures that proxies can be trusted before using potentially threatening code. Jini technology also supports configurable security. This enables a Jini client, service, or application to be deployed into environments that have varying or evolving security requirements.

The Jini technology security architecture supplies a programming model that supports these requirements. The programming model generally solves this problem by defining a constraints-

based system that is used to express the network security requirements. The client's network invocation constraints are conveyed by way of the proxy interface. The server's invocation constraints are generally conveyed to the proxy when a remote object is exported.

13.4.2 Constraints Model

The types of invocation constraints that can be specified are extensible. For example, existing invocation constraints include authentication, message integrity, and confidentiality. Both clients and servers can specify such constraints, and it is the responsibility of the proxy implementation to satisfy both parties' security constraints.

Jini technology relies on JAAS for the principals and credentials that will be used to effect network authentication of a local entity and its remote peer. Jini technology does not specify JAAS login modules to perform network authentication handshakes. The client's `Subject` is the current `Subject` associated with the thread making the remote call. This is retrieved from a thread by calling `javax.security.auth.Subject.getSubject` with the thread's current access control context. The server's `Subject` is normally taken from the thread that was current when the remote object was exported.

An entity may authenticate itself in a remote call as a subset of the `Principal`s in its `Subject` and only if that `Subject` contains the necessary public and/or private credentials required for the authentication mechanism used by the proxy and server implementations. However, additional principals and credentials might be derived, based on that authentication. The principals and credentials used to authenticate the server normally are not provided directly to the client making the remote call; instead, the client uses security constraints to restrict what they must be. In the server, the result of authenticating the client is represented by a `Subject` containing the subset of authenticated client `Principal`s and any derived `Principal`s, as well as the public credentials used during authentication and any derived public credentials. This `Subject` normally does not contain any private credentials and so cannot be used for authentication in further remote calls, unless delegation is used. However, the `Subject` can be used by the server for local authorization: for example, by binding the `Subject` to the running thread with `javax.security.auth.Subject.doAs`. Thus, the server could rely on the installed J2SE security policy to take into consideration the `Principal`s contained in the `Subject` when making policy decisions.

13.4.3 Establishing Proxy Trust

Clearly, the process as outlined requires that the proxy be trusted. Rather than having the client directly determine whether it trusts the proxy, the approach taken is first to determine that the client can trust the server, by requiring the server to authenticate itself, and then to determine that the server trusts the proxy, by asking the server for a verifier and passing the proxy to that verifier. If the client trusts the server and the server trusts the proxy, transitively, the client is inferred to trust the proxy. However, there are questions to answer for both of these steps.

First is the question of how the client can reliably verify that the server authenticates itself, when the client does not yet trust the downloaded proxy. To accomplish this, the proxy is required to support some bootstrap remote communication mechanism that the client does trust. Specifically, the proxy must be able to provide a bootstrap proxy that is trusted by the client. In the common case of a smart proxy wrapped around a standard dynamic proxy—for example, a dynamic proxy for a remote object exported to use Jini extensible remote invocation (Jini ERI)—the dynamic proxy usually can serve as the bootstrap proxy if its invocation handler does not depend on any downloaded code, on the assumption that the client will have configured in a trust verifier for such dynamic proxies and their invocation handlers. However, there is no requirement that normal

communication with the server use the same protocol used by the bootstrap proxy; the bootstrap proxy could be used strictly for trust verification.

Second is the question of how to reliably obtain a verifier from the server. (The proxy is not sent to the server, because an untrusted proxy could use a `writeReplace` serialization method to replace itself on the wire with a trusted proxy; the proxy needs to be checked in the client.) Authentication of the server and object integrity should be required during the remote call through the bootstrap proxy to obtain the verifier from the server, so that the remote call and the verifier object received from it can be trusted. To accomplish this, the client is responsible for specifying the server principals it trusts and for mandating constraints that ensure integrity protection and the server authentication.

13.4.4 Dynamic Policy

Once a proxy has been determined to be trustworthy, the client can dynamically grant the proxy permissions so that the proxy can interact with the remote peer and use the `Principal`s and credentials contained in the client's `Subject`. It is important to delay granting such permissions until after the trust decision, so that an untrusted proxy cannot abuse the grants in a way that might cause harm. Jini technology specifies a dynamic policy interface that a security policy provider can implement. Additionally, Sun's implementation of Jini includes a generic wrapper policy provider that implements the interface, and it can be wrapped around an arbitrary `Policy` instance. The implementation supports dynamic grants at runtime to protection domains with a given class loader and set of `Principal`s. Thus, the intent is that you receive a proxy, verify that it can be trusted, then grant it permissions, and then make calls through it.

One final note: For deployments having known trusted `CodeSource`s, an implementation of an `RMIClassLoader` provider can be configured such that code is downloaded only from `CodeSource`s that have been granted sufficient permission. This approach effectively eliminates denial-of-service attacks from untrusted code.

The preceding description of the Jini technology security architecture only skims the surface. The reader is encouraged to visit the Jini community Web site, www.jini.org, for additional information.

13.5 Brief Introduction to J2EE

J2EE, a superset of J2SE, manages the security of container-resident applications. J2EE provides a component-based development and deployment model. Three container types are prevalent today: JavaServer Pages (JSP), Java servlets, and Enterprise JavaBeans (EJB). The J2EE platform manages the underlying infrastructure for the components so that the application developer can focus on the application functionality and not be concerned with managing the security and computing infrastructure. When a component is installed into the container, the security requirements of the component are declared within a deployment descriptor. Once deployed, the container ensures that the security requirements of the component are met prior to dispatching into the component and throughout the life cycle of component invocations and interactions.

This model has proved very successful, and with success come new challenges. Success in the enterprise has generated new platform requirements. For instance, enterprises have begun to deploy access management systems that enable a cohesive and unified security posture. To integrate J2EE within such an environment, future versions of the J2EE platform specification will incorporate the Java Authorization Contract for Containers, which fully integrates the security policy capabilities of J2SE with the container-managed notions of J2EE. The realization of this particular specification manifests itself as implementations of `Permission`, `Policy`, and

management interfaces that a provider must make available to allow container deployment tools to create and manage permission collections corresponding to roles.

The J2EE platform manages peer authentication for the hosted components. Presently, the authentication mechanisms are fixed by the platform specification to the most common mechanisms. We anticipate that new authentication technologies will need to be supported in the near term. To accommodate advances in authentication technologies, a Java authentication Service Provider Interface is being developed within the Java community process.

13.6 Client Containers

Client containers have been a part of the application delivery model since the inception of Java. The `appletviewer` utility program was the original client container. As the utility and popularity of Java began to take hold, browser vendors added integral support for applets by integrating Java runtime environments into their Web browsers. As Java matured, it became even more popular within corporate intranets; thus, Sun developed the Java Plug-in (JPI) applet container to meet the needs of enterprises. Enterprises generally have stringent application delivery, manageability, and compatibility requirements that the JPI specifically addresses. For example, the Java Plug-in has provisions for specifying the version information so that a compatible Java runtime can be selected and even installed. The Java Plug-in has become the de facto Java runtime environment for the most popular browsers in use today. The JPI can be installed as the default VM such that it is invoked when the browser encounters an `<APPLET>` tag.

Applets are a wonderful delivery and deployment mechanism, but in many scenarios, a stand-alone application is a better solution. However, applications have the same delivery, manageability, and compatibility requirements as applets. To address this problem, an application container, Java Web Start (JWS), was developed by the Java community.

The Java Plug-in has some rather unique and difficult integration requirements. For example, the JPI must ensure that applets adhere to the Java security model as well as enable applets to interact with the Document Object Model [120]. This requirement places restrictions on what an applet can access within the containing document and also on access to the applet by other document elements, such as scripts. Similarly, JWS has unique security requirements. For example, JWS must ensure that the permissions accorded an application are valid according to the policy and that the user accepts the signing certificate.

To that end, both JPI and JWS implement custom `ClassLoader` and `SecurityManager`s to address the unique interaction and security requirements of a client container. For example, the JPI must ensure that cached JAR files are accessed only by classes in accordance with `CodeSource` implication rules. Also, at times, the JPI may encounter an applet that requires more permission than is accorded by the current policy, and thus a dialogue may be needed to give the user the opportunity to override security policy. The JPI can capture the override such that it can persist so that the user will not be prompted again in the future.

User management of security policy is not always the best solution to these sorts of problems. We should see alternative solutions forthcoming. Advances in policy expression languages and certificate status and revocation checking should greatly simplify the user's burden.

Security management is not just for end users. Application developers often do not want to become experts in security simply to write secure applications. They would benefit, for example, from interfaces that they can call to obtain security services, such as encryption algorithms, key sizes, and protocol types, without having to know the specifics. Such simple security APIs should greatly increase productivity, as well as the security quality of the resulting applications.

13.7 Final Remarks

Throughout the chapters of this book, we have delved into the architectural underpinnings of the Java security architecture, presented overviews of the relevant APIs, and provided insight into our design philosophy. We have described the platform security from the lowest substrate the Java programming language and Java virtual machine provide, through to the application programming interfaces and up to the security policy expressions. We have touched on security mechanisms, including cryptography, object security, network authentication, and security protocols. We have highlighted the fundamental and the practical application of these technologies.

Over the last few decades, the importance of information, computer, and network security has come to the forefront of our attention. Networks and computers are integral components of our daily lives. Our financial, medical, transportation, and even our health care systems all rely on secure network computing technology. Compromise of these systems could be catastrophic, and so we are always searching for better ways to secure our systems. What this suggests is that security is an ongoing process that must constantly be evaluated, corrected, adapted, and refined. The future holds great promise along with equally great challenges.

Bibliography

[1] C. Adams. The Simple Public-Key GSS-API Mechanism (SPKM). Request for Comments (RFC) 2025, Internet Engineering Task Force, October 1996.

[2] ANSI. Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm, ANSI X9.62, 1998.

[3] K. Arnold and J. Gosling. *The Java Programming Language*, Second Edition. Reading, Mass.: Addison-Wesley, 1998.

[4] E. Baize and D. Pinkas. The Simple and Protected GSS-API Negotiation Mechanism. Request for Comments (RFC) 2478, Internet Engineering Task Force, December 1998.

[5] D. E. Bell and L. J. LaPadula. Secure Computer Systems: A Mathematical Model. *Journal of Computer Security*, 4(2–3): 239–263, 1996. *A modern reprint of the same-titled technical report*, ESD-TR-73-278, Vol. 2, Bedford, Mass.: The MITRE Corporation, 1973.

[6] S. Bellovin. Security Problems in the TCP/IP Protocol Suite. *Computer Communication Review*, 19(2): 32–48, April 1989.

[7] S. Bellovin. Using the Domain Name System for System Break-ins. *Proceedings of the Fifth Usenix UNIX Security Symposium*, June 1995.

[8] S. M. Bellovin and W. R. Cheswick. Network Firewalls. *IEEE Communications*, 50–57, September 1994.

[9] S. Bellovin and M. Merritt. Limitations of the Kerberos Authentication System. *Proceedings of the Winter 1991 Usenix Conference*, 253–267, Dallas, January 1991.

[10] S. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 72–84, Oakland, Calif., May 1992.

[11] S. Bellovin and M. Merritt. Augmented Encrypted Key Exchange: A Password-Based Protocol Secure Against Dictionary Attacks and Password File Compromise. *Proceedings of the 1st ACM Conference on Computer and Communications Security*, 244–250, Fairfax, Va., November 1993.

[12] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuchynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance in the SPIN Operating System. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 251–266, Copper Mountain Resort, Colo., December 1995. Published as *ACM Operating System Review* 29(5): 251–266, December 1995.

[13] K. J. Biba. Integrity Considerations for Secure Computer Systems. *U.S. Air Force Electronic Systems Division Technical Report 760372*, Bedford, Mass.: Hanscom Air Force Base, April 1977.

[14] M. Blaze, J. Feigenbaum, and A. Keromytis. The KeyNote Trust-Management System Version 2. Request for Comments (RFC) 2704, Internet Engineering Task Force, September 1999.

[15] J. Bloch. *Effective Java*. Boston, Mass.: Addison-Wesley, 2001.

[16] S. Boeyen, T. Howes, and P. Richard. Internet X.509 Public Key Infrastructure LDAPv2 Schema. Request for Comments (RFC) 2587, Internet Engineering Task Force, June 1999.

[17] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0*, Second Edition. W3C Recommendation, October 2000.

[18] D. F. C. Brewer and M. J. Nash. The Chinese Wall Security Policy. *Proceedings of the IEEE Symposium on Security and Privacy*, 206–214, Oakland, Calif., April 1989.

[19] M. Burrows, M. Abadi, and R. M. Needham. A Logic for Authentication. *ACM Transactions on Computer Systems*, 8(1): 18–36, February 1990.

[20] CCITT Recommendation X.509. The Directory Authentication Framework, 1988.

[21] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4): 271–307, November 1994.

[22] S. Chokhani and W. Ford. Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework. Request for Comments (RFC) 2527, Internet Engineering Task Force, March 1999.

[23] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. *Proceedings of the IEEE Symposium on Security and Privacy*, 184–194, Oakland, Calif., April 1987.

[24] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2): 57–78, February 1991.

[25] D. Crocker. Standard for the Format of ARPA Internet Text Messages. Request for Comments (RFC) 822, Internet Engineering Task Force, August 1982.

[26] I. B. Damgård. Design Principles for Hash Functions. *Advances in Cryptology: Proceedings of Crypto '89*, Vol. 435 of Lecture Notes in Computer Science, 416–427. New York: Springer-Verlag, October 1989.

[27] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. Request for Comments (RFC) 1883, Internet Engineering Task Force, December 1995.

[28] T. Dierks and C. Allen. The TLS Protocol Version 1.0. Request for Comments (RFC) 2246, Internet Engineering Task Force, January 1999.

[29] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT–22(6): 644–665, November 1976.

[30] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, IT–29(2): 198–208, March 1983.

[31] D. Eastlake and J. Reagle. XML Encryption Syntax and Processing. W3C Proposed Recommendation, October 2002.

[32] D. Eastlake, J. Reagle, and D. Solo. XML-Signature Syntax and Processing. Request for Comments (RFC) 3075, Internet Engineering Task Force, March 2002.

[33] M. Eisler. LIPKEY—A Low Infrastructure Public Key Mechanism Using SPKM. Request for Comments (RFC) 2847, Internet Engineering Task Force, June 2000.

[34] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. Request for Comments (RFC) 2693, Internet Engineering Task Force, September 1999.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Reading, Mass.: Addison-Wesley, 1995.

[36] M. Gasser. *Building a Secure Computer System*. New York: Van Nostrand Reinhold, 1988.

[37] J. A. Goguen and J. Meseguer. Security Policies and Security Models. *Proceedings of the IEEE Symposium on Security and Privacy*, 11–20, Oakland, Calif., April 1982.

[38] J. A. Goguen and J. Meseguer. Unwinding and Inference Control. *Proceedings of the IEEE Symposium on Security and Privacy*, 75–86, Oakland, Calif., April 1984.

[39] L. Gong. Collisionful Keyed Hash Functions with Selectable Collisions. *Information Processing Letters*, 55(3): 167–170, August 1995.

[40] L. Gong. New Security Architectural Directions for Java (Extended Abstract). *Proceedings of IEEE COMPCON*, 97–102, San Jose, Calif., February 1997.

[41] L. Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3): 14–19, May/June 1997.

[42] L. Gong, P. Lincoln, and J. Rushby. Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid Faults. *Proceedings of the 5th IFIP Working Conference on Dependable Computing for Critical Applications*, 79–90, Urbana-Champaign, Ill., September 1995.

[43] L. Gong, T. M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting Poorly Chosen Secrets from Guessing Attacks. *IEEE Journal on Selected Areas in Communications*, 11(5): 648–656, June 1993.

[44] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 103–112, Monterey, Calif., December 1997.

[45] L. Gong, R. Needham, and R. Yahalom. Reasoning about Belief in Cryptographic Protocols. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 234–248, Oakland, Calif., May 1990.

[46] L. Gong and X. Qian. Computational Issues of Secure Interoperation. *IEEE Transactions on Software Engineering*, 22(1): 43–52, January 1996.

[47] L. Gong and R. Schemers. Implementing Protection Domains in the Java Development Kit 1.2. *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 125–134, San Diego, Calif., March 1998.

[48] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Boston, Mass.: Addison-Wesley, June 2000.

[49] Phillip Hallam-Baker. XML Key Management Specification (XKMS 2.0). W3C Working Draft, March 2002.

[50] N. Haller, C. Metz, P. Nesser, and M. Straw. A One-Time Password System. Request for Comments (RFC) 2289, Internet Engineering Task Force, February 1998.

[51] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8): 461–471, August 1976.

[52] C. Hawblitzel, C-C Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. *Proceedings of the USENIX Annual Technical Conference*, 259–270, New Orleans, La., June 1998.

[53] M. P. Herlihy and J. D. Tygar. How to Make Replicated Data Secure. Advances in Cryptology. *Proceedings of Crypto '87*, Vol. 293 of Lecture Notes in Computer Science, 379–391. New York: Springer-Verlag, 1987.

[54] R. Housley, W. Ford, T. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Request for Comments (RFC) 3280, Internet Engineering Task Force, April 2002.

[55] International Standards Organization. *ISO 7816 Parts 1–6*. July 1987.

[56] International Telecommunication Union. ITU-T Recommendation X.509: The Directory: Public-Key and Attribute Certificate Frameworks, 2000.

[57] A. K. Jones. *Protection in Programmed Systems*. Ph.D. dissertation, Pittsburgh, Penn.: Carnegie-Mellon University, June 1973.

[58] J. Kabat and M. Upadhyay. Generic Security Service API Version 2: Java Bindings. Request for Comments (RFC) 2853, Internet Engineering Task Force, June 2000.

[59] B. Kaliski. The MD2 Message-Digest Algorithm. Request for Comments (RFC) 1319, Internet Engineering Task Force, April 1992.

[60] S. Kent. Privacy Enhancement for Internet Electronic Mail: Part II—Certificate-Based Key Management. Request for Comments (RFC) 1422, Internet Engineering Task Force, November 1993.

[61] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. Request for Comments (RFC) 2401, Internet Engineering Task Force, November 1998.

[62] S. Kent and J. Linn. Privacy Enhancement for Internet Electronic Mail: Part II—Certificate-Based Key Management. Request for Comments (RFC) 1114, Internet Engineering Task Force, November 1989.

[63] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Revised Edition*. Reading, Mass.: Addison-Wesley, 1969.

[64] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Searching and Sorting*. Reading, Mass.: Addison-Wesley, 1973.

[65] J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5). Request for Comments (RFC) 1510, Internet Engineering Task Force, September 1993.

[66] A. G. Konheim. *Cryptography: A Primer*. New York: John Wiley, 1981.

[67] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. Request for Comments (RFC) 2104, Internet Engineering Task Force, February 1997.

[68] L. Lamport. Password Authentication with Insecure Communication. *Communications of the ACM*, 24(11): 770–772, November 1981.

[69] B. W. Lampson. Protection. *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971. Reprinted in *ACM Operating Systems Review*, 8(1): 18–24, January 1974.

[70] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10): 613–615, October 1973.

[71] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4): 265–310, November 1992.

[72] C. E. Landwehr. Formal Models for Computer Security. *ACM Computing Survey*, 13(3): 247–278, September 1981.

[73] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, 36–44, Vancouver, British Columbia, October 1998.

[74] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Reading, Mass.: Addison-Wesley, 1999.

[75] J. Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. Request for Comments (RFC) 1421, Internet Engineering Task Force, February 1993.

[76] J. Linn. The Kerberos Version 5 GSS-API Mechanism. Request for Comments (RFC) 1964, Internet Engineering Task Force, June 1996.

[77] J. Linn. Generic Security Service Application Program Interface Version 2. Request for Comments (RFC) 2743, Internet Engineering Task Force, January 2000.

[78] T. M. A. Lomas, L. Gong, J. H. Saltzer, and R. M. Needham. Reducing Risks from Poorly Chosen Keys. *Proceedings of the 12th ACM Symposium on Operating System Principles*, Litchfield Park, Ariz. Published in *ACM Operating Systems Review*, 23(5): 14–18, December 1989.

[79] D. McCullough. A Hookup Theorem for Multilevel Security. *IEEE Transactions on Software Engineering*, 16(6): 563–568, June 1990.

[80] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes.* New York: John Wiley, 1997.

[81] C. Meadows. Using Narrowing in the Analysis of Key Management Protocols. *Proceedings of the IEEE Symposium on Security and Privacy*, 138–147, Oakland, Calif., May 1989.

[82] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. New York: CRC Press, 1997.

[83] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems.* Ann Arbor, Mich.: UMI Research Press, 1982. Stanford University. Revised from 1979 Ph.D. thesis.

[84] R. C. Merkle. A Fast Software One-Way Hash Function. *Journal of Cryptology*, 3(1): 43–58, 1990.

[85] C. H. Meyer and M. Schilling. Secure Program Load with Modification Detection Code. *Proceedings of the 5th Worldwide Congress on Computer and Communication Security and Protection–SECURICOM 88*, 111–130, Paris, 1988.

[86] J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol Security Analysis. *IEEE Transactions on Software Engineering*, SE-13(2): 274–288, February 1987.

[87] S. P. Miller, C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos Authentication and Authorization System. Project Athena Technical Plan Section E.2.1. Cambridge, Mass.: Massachusetts Institute of Technology, October 1988.

[88] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure Software Architectures. *Proceedings of the IEEE Symposium on Security and Privacy*, 84–93, Oakland, Calif., May 1997.

[89] J. Myers. Simple Authentication and Security Layer (SASL). Request for Comments (RFC) 2222, Internet Engineering Task Force, October 1997.

[90] M. Naor and M. Yung. Universal One-Way Hash Functions and Their Cryptographic Applications. *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, 33–43, Seattle, May 1989.

[91] National Institute of Standards and Technology. *Digital Signature Standard*, January 2000. U.S. Federal Information Processing Standards Publication, FIPS PUB 186-2.

[92] National Institute of Standards and Technology. *Advanced Encryption Standard*, November 2001. U.S. Federal Information Processing Standards Publication, FIPS PUB 197.

[93] National Institute of Standards and Technology. *Secure Hash Standard*, August 2002. U.S. Federal Information Processing Standards Publication, FIPS PUB 180-2.

[94] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12): 993–999, December 1978.

[95] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9): 33–38, September 1994.

[96] P. G. Neumann. *Computer-Related Risks*. Reading, Mass.: Addison-Wesley, 1995.

[97] J. Postel. Internet Protocol. Request for Comments (RFC) 791, Internet Engineering Task Force, September 1981.

[98] M. O. Rabin. *Fingerprinting by Random Polynomials*. Technical Report TR-15-81, Center for Research in Computing Technology, Cambridge, Mass.: Harvard University, 1981.

[99] M. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 68–80, Fairfax, Va., November 1994.

[100] E. Rescorla. HTTP over TLS. Request for Comments (RFC) 2818, Internet Engineering Task Force, May 2000.

[101] S. Ritchie. Systems Programming in Java. *IEEE Micro*, 17(3): 30–35, May/June 1997.

[102] R. L. Rivest. The MD5 Message-Digest Algorithm. Request for Comments (RFC) 1321, Internet Engineering Task Force, April 1992.

[103] R. L. Rivest and B. Lampson. *SDSI—A Simple Distributed Security Infrastructure*. Technical report, Cambridge, Mass.: Massachusetts Institute of Technology, October 1996.

[104] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2): 120–126, February 1978.

[105] RSA Laboratories. PKCS #1: RSA Encryption Standard, Version 1.5., November 1993.

[106] RSA Laboratories. PKCS #5: Password-Based Encryption Standard, Version 1.5., November 1993.

[107] RSA Laboratories. PKCS #7: Cryptographic Message Syntax Standard, Version 1.5., November 1993.

[108] RSA Laboratories. PKCS #8: Private-Key Information Syntax Standard, Version 1.2., November 1993.

[109] RSA Laboratories. PKCS #10: Certification Request Syntax Standard, Version 1.7., May 2000.

[110] J. H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7): 388–402, July 1974.

[111] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9): 1278–1308, September 1975.

[112] V. Samar and C. Lai. Making Login Services Independent from Authentication Technologies. *Proceedings of the SunSoft Developer's Conferenc*e, San Jose, Calif., March 1996.

[113] R. S. Sandhu. The Typed Access Matrix Model. *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 122–136, Oakland, Calif., May 1992.

[114] F. B. Schneider. Implementing Fault-Tolerant Services Using the State-Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4): 299–319, December 1990.

[115] B. Schneier. *Applied Cryptography*. New York: John Wiley, 1994.

[116] M. D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. Ph.D. dissertation. Cambridge, Mass.: Massachusetts Institute of Technology, September 1972.

[117] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, 213–227, Seattle, Wash., October 1996. Published as *ACM Operating Systems Review*, 30, special winter issue, 1996.

[118] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11): 612–613, November 1979.

[119] R. Shirey. Internet Security Glossary. Request for Comments (RFC) 2828, Internet Engineering Task Force, May 2000.

[120] Johnny Stenback, Philippe Le Hégaret, Arnaud Le Hors. Document Object Model Level 2 HTML Specification Version 1.0. W3C Proposed Recommendation, November 2002.

[121] Sun Microsystems. *JAR File Specification*. J2SE documentation, 1999. http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html.

[122] Sun Microsystems. *The Java Extension Mechanism Architecture*. J2SE documentation, 1999. http://java.sun.com/j2se/1.4/docs/guide/extensions/spec.html.

[123] Sun Microsystems. *Jini Architecture Specification Version 1.2*, December 2001. http://wwws.sun.com/software/jini/specs/jini1.2html/jini-title.html.

[124] Sun Microsystems. *Java Card 2.2 Application Programming Interface*, June 2002. http://java.sun.com/products/javacard/specs.html.

[125] Sun Microsystems. *Java Card 2.2 Runtime Environment Specification*, June 2002. http://java.sun.com/products/javacard/specs.html.

[126] Sun Microsystems. *Java Card 2.2 Virtual Machine Specification*, June 2002. http://java.sun.com/products/javacard/specs.html.

[127] U.S. General Accounting Office. Information Security: Computer Attacks at Department of Defense Pose Increasing Risks. Technical Report GAO/AIMD-96-84, Washington, D.C., May 1996.

[128] U.S. National Bureau of Standards. *Data Encryption Standard*, January 1977. U.S. Federal Information Processing Standards Publication, FIPS PUB 46.

[129] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure Network Objects. *Proceedings of the IEEE Symposium in Security and Privacy*, 211–221, Oakland, Calif., May 1996.

[130] M. Wahl, S. Kille, and T. Howes. Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names. Request for Comments (RFC) 2253, Internet Engineering Task Force, December 1997.

[131] J. Waldo. The Jini Architecture for Network-Centric Computing, *Communications of the ACM*, 42(7): 76-82, July 1999.

[132] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 116–128, Saint-Malo, France, October 1997.

[133] M. V. Wilkes. *Time-Sharing Computer Systems*. London: MacDonald, 1968.

[134] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp—An Experimental Computer System*, New York: McGraw-Hill, 1981.

[135] F. Yellin. Low Level Security in Java. *Proceedings of the 4th International World Wide Web Conference*, 369–379, Boston, Mass., December 1995.

[136] P. R. Zimmerman. The Official PGP User's Guide. Cambridge, Mass.: MIT Press, 1995.