WILEY

Securing JAVA™

Getting Down to Business with Mobile Code

Gary McGraw
Edward W. Felten

Praise for *Securing Java...*

"This second edition is mandatory up-to-date reading for every user and developer of Webware. Its eye-opening analysis of the security risks provides timely realism amidst an otherwise mad dash to universal Net use."

—Peter G. Neumann
Principal Scientist
Computer Science Lab, SRI International
Moderator of the Risks Forum
Author of *Computer-Related Risks*

"This book cuts through the hype and clears up the confusion surrounding Java and security."

—Bill Venners
Author of *Inside the Java Virtual Machine*

"McGraw and Felten give Java's designers their due for trying to create a secure Internet programming language, but they cut through a lot of the hype—both positive and negative—about how secure it really is. Since Java pervades many aspects of Internet commerce, knowing the true risks is important for anyone who plans to do business on the Net."

—David Carr
Senior Editor
Internet World

"Security is a part of modern networking that often gets ignored until it's too late. . . . Gary and Ed's book is a clear explanation of the situation, chock full of details and useful advice. A must-read book for anyone who is considering mobile code as part of their mission-critical infrastructure."

—Marcus J. Ranum
CEO, Network Flight Recorder, Inc.

Praise for the First edition: *Java Security*...

"One of the best treatments of Java security issues I have ever had the pleasure to read. The advice offered in this book is sound and reasonable."

Thomas A. Longstaff
Manager of Research & Development
CERT Coordination Center

"A provocative and useful discussion of security issues around Java and the Internet to date"

Li Gong, Ph.D.
Java Security Architect
JavaSoft

"This book is mandatory reading for every user and developer of webware. Its eye-opening analysis of the security risks provides timely realism amidst an otherwise mad dash to universal net browsing.

Java is hot. Java is cool.
Its use is riddled with risks that fool.
Java Security takes us all back to school."

Peter G. Neumann, Principal Scientist, Computer Science Lab,
SRI International; Moderator of the Risks Forum; Author of
Computer-Related Risks

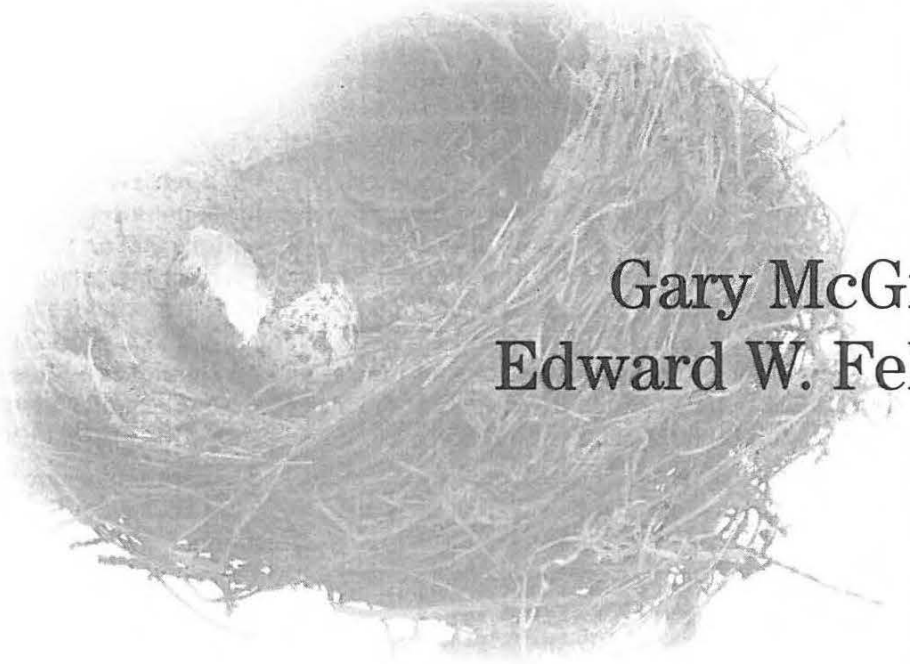
"... a tour de force.... clear and comprehensive discussions of the Java security model and various problems with its numerous implementations. It will make for enjoyable, and profitable, reading by all system administrators, webmasters, and programmers—particularly in the corporate world. If you surf, or if you maintain a website, this book is for you. This is an enormously useful book. Buy it!

Gregory J. E. Rawlins,
Associate professor, Indiana University;
author of *Moths to the Flame: The Seductions of Computer Technology*

"McGraw and Felten do a great job of presenting a thorough and understandable treatment of the complex security issues surrounding Java and other Web-related languages....This book is a must for anyone who uses Web browsers and related software, written by the experts who have practically defined the field of Java security."

Michael Shoffner, Java developer, Prominance.com

Securing Java: Getting Down to Business with Mobile Code



Gary McGraw
Edward W. Felten

WILEY COMPUTER PUBLISHING

John Wiley & Sons, Inc.
New York • Chichester • Weinheim • Brisbane • Singapore • Toronto

For our parents.

Publisher: Robert Ipsen
Editor: Marjorie Spencer
Assistant Editor: Margaret Hendrey
Managing Editor: Frank Grazioli
Composition: Benchmark Productions Inc., Boston

This book is printed on acid-free paper. ♾

Published by John Wiley & Sons, Inc.

Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:
McGraw, Gary, 1966–
Securing Java : getting down to business with mobile code
McGraw, Edward W. Felten, — 2nd ed.
p. cm.
Originally published under title: Java security. 1997.
Includes bibliographical references and index.
ISBN 0-471-31952-X (paper/online : alk. paper)
1. Java (Computer program language) 2. Computer security.
I. Felten, Edward, 1963– . II. McGraw, Gary, 1966– Java
security. III. Title.
QA76.73.J38M354 1999
005.8—dc21 98-49151
CIP

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

CONTENTS

PREFACE

Java has grown by leaps and bounds since its introduction in 1996, and is now among the most popular computing platforms on the planet. Java has evolved and changed so much that at a mere two-years old, our original work, *Java Security: Hostile Applets, Holes, and Antidotes*, found itself in serious need of revision and expansion. This book is the result of several years of thinking about mobile code and security, and includes many things we have discovered while working on real-world systems with businesses and government agencies. Our goal is to present enough information to help you separate fact from fiction when it comes to mobile code security.

Java has become much more complicated and multifaceted than it was when it was introduced. No longer simply a client-side language for applets, Java can now be found on everything from enterprise application servers to embedded devices like smart cards. We have tried to address security factors from throughout the entire Java range in this book.

We hope this book appeals to geeks and grandmothers alike (not that some grandmothers aren't geeks). Although it gets technical in places, we hope the messages are clear enough that even the casual Web user comes away with a broader understanding of the security issues surrounding mobile code. We kept four groups in mind as we wrote this book: Web users, developers, system administrators, and business decision-makers. Many of the issues of mobile code security cut across these groups. As Java integrates itself into the foundations of electronic commerce, Java security issues take on more urgency.

Java is only one kind of mobile code among many. Other systems immersed in the same security dilemma include ActiveX, JavaScript, and Word Macros. It is essential not to get the wrong message from this book. Our focus on Java is no accident. We believe Java is the most viable mobile code system created to date. Don't believe that through our work we imply that other systems are any more secure than Java. Just the opposite is true.

With the introduction of code signing to Java (in JDK 1.1) and its enhancement with access control (in Java 2), securing Java became much harder. Java's position along the security/functionality tradeoff has moved significantly toward functionality, to the detriment of security. This is good if you want more functionality, which most businesses and developers seem to need, but it is bad if you are charged with managing security risks. Forming an intelligent Java use policy is more important than ever, but doing so is more complicated than it used to be.

The computer field moves so fast that people have begun to refer to *Internet time* to grapple with its constantly accelerating speed. Three months is a year in Internet time. Java is directly involved in the speed of the field, and has done its share to make things move even more quickly. One tricky aspect of writing a topical book relating to the Web is figuring out when to stop the action. This process can be likened to freeze-framing a picture of a movie. In that sense, this book is a snapshot of Java security. We hope we have succeeded in making it a useful way to learn about Java security. For up-to-date information, see the book's companion Web site at *www.rstcorp.com/java-security.html*.

As we went to press, Sun Microsystems renamed JDK 1.2 and called it Java 2. We have attempted to use correct version numbers throughout and apologize for any confusion.

Chapter 1, "Mobile Code and Security: Why Java Security Is Important,"," sets the stage with a discussion of the four intended audiences. As Java matures, it is making important inroads into the enterprise world. That means Java security is now as important to business people and system administrators as it is to Web users and Java developers. For the uninitiated, Chapter 1 provides a quick and cursory introduction to Java. Pointers are provided to more through Java texts that cover the ins and outs of the entire Java language in more detail. This is, after all, not a book on Java per se, but is instead a book on Java security. We also spend some time discussing why the once-important distinction between applets and applications has been superceded by concerns about trust. It turns out that under the Java 2 architecture, applets can be completely trusted and applications can be completely untrusted. In fact, every kind of Java code can be doled out different amounts of trust, depending on what the user's policy says. Finally, we cover some other popular forms of mobile code and discuss how their security stacks up against Java. The main purpose of this chapter is to provide some context for the later discussion of Java's critical security implications and to introduce the central idea of the book: weighing the benefits of Java use against the risks.

Chapter 2, "The Base Java Security Model: The Original Applet Sandbox," examines the base Java security model in some detail. As a prelude to our discussion, we introduce four categories of attacks, ranging from the very serious to the merely annoying: system modification, invasion of privacy, denial of service, and antago-

nism. We then discuss Java's programming-languages approach to security and introduce the three parts of the original applet sandbox. These include the Verifier, the Class Loader Architecture, and the Security Manager. We also introduce the idea that Java security fundamentally relies on ensuring type safety. The base sandbox provides the foundation of Java's new trust-based security model. Starting with a restrictive sandbox for untrusted code, restrictions can be lifted little by little until code takes on complete trust and is awarded full run of the entire system.

Chapter 3, "Beyond the Sandbox: Signed Code and Java 2," examines Java's new trust-based security model. With the addition of code signing in JDK 1.1, Java's security architecture underwent a large shift. Java 2 completed the transformation with the addition of access control. It is now possible to create complex security policy for mobile code written in Java and have the Java system itself enforce the policy. The change certainly affords more power to mobile code than ever before, but it also introduces a major new risk to Java: a human-centered policy management risk. Setting up and managing a mobile code policy will be a complex and error-prone undertaking requiring security experience. JDK 1.1 and Java 2 rest on the notion of trust, which leverages the technological power of code signing. Understanding the new model requires understanding the way code signing and trust interact, and discounting some of the common myths associated with it. Chapter 3 ends with a discussion of stack inspection and the Java 2 code-signing API. (Appendix C, "How to Sign Java Code," is a code-signing tutorial covering Microsoft, Netscape, and Sun's three different code signing schemes.)

Chapter 4, "Malicious Applets: Avoiding a Common Nuisance," begins to discuss what happens when the Java security model is abused by hostile applets. Hostile applets come in two forms: very dangerous attack applets that involve security breaches, and merely annoying malicious applets that are more of a nuisance than anything else. Chapter 4 is all about malicious applets. Malicious applets are quite easy to create, and they are equally easy to find on the Web. Unfortunately, there are just as many unscrupulous individuals on the Net as there are in the rest of the world. Bad guys are more than happy to include Java in their list of offensive weapons. Our mission is to make Java users aware of common classes of attacks.

Chapter 5, "Attack Applets: Exploiting Holes in the Security Model," delves more deeply into the Java security model by focusing attention on some of the well-publicized security holes that have been discovered. This is where our discussion of hostile applets turns more serious. Securing Java is a difficult job, especially when it comes to implementing complicated models. Attack applets have been created in the lab that exploit the holes we discuss. Some of the holes are simple implementation bugs, while others indicate more serious design flaws. The good news is that Sun and other licensees take Java security very seriously and they respond quickly to fix any holes once they are discovered. We think discussing these holes is important since it emphasizes the true nature of computer security.

Chapter 6, "Securing Java: Improvements, Solutions, and Snake Oil," has two overall goals, both of which are meant to impact the Java security situation positively. The first is to suggest some high-level antidotes for Java security concerns that are not tied to particular attacks. Experts in computer security have pointed out several global deficiencies in the Java approach to security. Fixing some of

these would certainly improve the model. High-level concerns addressed in Chapter 6 include programming language issues, formal analysis of Java, applet logging, trust, decompilation, applet monitoring, and policy management. Hopefully, some of the high-level concerns we raise will eventually be addressed in the Java platform itself. In the meantime, a number of third-party vendors are eager to help. The second goal of Chapter 6 is to introduce the players briefly and to discuss what risks third-party vendors can and cannot address. The computer security field has its share of snake oil, and complex issues such as mobile code security tend to be easy to exploit. One of our goals is to bring some realism to the table and arm you with the right questions to ask.

If you only read one chapter of this book, read Chapter 7, "Java Security Guidelines: Developing and Using Java More Securely." This chapter presents two sets of guidelines: one for Java developers and one for Java users. Writing security-critical code is not easy, and developers need all the help they can get. We offer 12 rules for writing safer Java. Although the rules get a bit technical, it is worth spending some time to figure them out. By contrast, our guidelines for Java users are simple to understand and follow; in fact, most of them are simply common sense.

Chapter 8, "Java Card Security: How Smart Cards and Java Mix," is devoted to Java on smart cards. We decided to include this chapter since Java Cards are likely to show up in millions of consumer wallets in the next few years. Smart card security is really too big an issue to cover in a single chapter, so we focus primarily on the security impact of putting a Java Virtual Machine on a card. Chapter 8 covers six key questions, including: What is a smart card?, Why put Java on a smart card?, and How does the use of Java impact smart card security?

We conclude by covering some of the challenges to mobile code that remain to be conquered. Chapter 9, "The Future of Java Security: Challenges Facing Mobile Code," presents a concise set of six central lessons we have learned during our time in the Java security trenches. We go on to discuss several open research issues that you're likely to hear about again. Finally, we discuss the notion of security assurance, an important strategy in securing Java.

We hope that this book is both informative and useful. Making intelligent decisions regarding the use of Java (especially in business and other mission-critical systems) requires some knowledge of the current risks. Our goal is to disclose those risks—and countermeasures to curtail them—as clearly and objectively as possible. Armed with the knowledge that we present in this book, Java users, site managers, and business decision-makers can make better Java use policies.

Acknowledgments

This book is a collaborative effort in more ways than one. Not only did the authors work together closely, but we also sought input from many other people. We are grateful for the help we received.

Reliable Software Technologies (*www.rstcorp.com*) remains a great place to work. The intellectually stimulating environment makes going to work interesting and fun. Many people at RST read drafts of the book or helped in other ways. They include John Viega (intrepid proofreader and co-author of the code-signing tutorial

in Appendix C), Tom O'Connor (who also read the entire manuscript more than once and co-wrote the code-signing tutorial), Anup Ghosh (fellow security researcher), Peggy Wallace (travel, anyone?), Lora Kassab (one-time RST intern whose code from the first edition lives on), Jeff Payne (RST's forward-thinking CEO), Jon Beskin, Matt Schmidt, Brad Arkin, Andi Bruno (who herds the marketing cats and makes us be nice), and Jeff Voas (who continues to lead RST's excellent research group by example).

The members of Princeton University's Secure Internet Programming Team (*www.cs.princeton.edu/sip*) also provided valuable input. Besides wading through several drafts, the Princeton team was responsible for raising many of the key issues in Java security. Special thanks to Drew Dean and Dan Wallach (cofounders of the Princeton team) and Dirk Balfanz. Dan is now a professor at Rice University. Drew is a research scientist at Xerox PARC. Princeton's Computer Science department provides a wonderful environment for discovering and exploring new research topics.

We would also like to thank Tom Cargill, independent consultant and discoverer of two security flaws; David Hopwood, discoverer of several attack applets; Mark LaDue, creator of the Hostile Applets Home Page (keep 'em honest, Mark); Dennis Volpano of the Naval Postgraduate School; Tom Longstaff, research director at the CERT Coordination Center; Roland Schemers, JavaSoft security implementation wizard (who helped with code-signing tool questions); Marianne Mueller, Java developer, security expert, and long-suffering target of press inquiries at JavaSoft; Jim Roskind, Netscape's Java security expert; Andrew Herbert, APM's Chief Scientist in the real Cambridge; Ken Ayer, chip card security manager at Visa; Don Byrd, UMass research associate and careful proofreader of the first edition; Hideyuki Hayashi, who translated the first edition into Japanese (and did an excellent job according to friends at Sumitomo in New York); Kieran Murphy, editor at developer.com; Chuck Howell, now at Mitretek; and Mike Shoffner, Java developer at Prominence Dot Com. Li Gong, security architect at JavaSoft, has been a particularly valuable help, both as a research colleague and as a sane point-of-view at JavaSoft. More power to you, Li.

Wiley's staff did an excellent job shepherding this book through the editing and production process. Special thanks to Marjorie Spencer and Frank Grazioli, who went out of their way to make this project go smoothly. Thanks to Margaret Hendrey for playing fast and loose with extensions (don't tell anybody). Also thanks to the rest of the team at Wiley.

Finally, and most importantly, we're grateful to our families for putting up with us while we worked on the book, *again*. Amy Barley, Jack, and Eli seem to have adjusted to Gary's persistent book-writing. Laura Felten and Claire suspect that Ed's book-writing has become an addiction. Without the support of our families, this book would not have been possible.

CHAPTER 1

Mobile Code and Security: Why Java Security Is Important

Java security is more important than ever. Since its introduction in 1995, Java has become one of the most popular development platforms on the planet. In fact, Java has been widely adopted more quickly than any other computer language. It now easily tops the list of preferred platforms for Internet-savvy mobile code. There are tens of thousands of Java developers (some say hundreds of thousands), and demand for Java skills appears to be growing. Java is definitely here to stay.

Java holds great promise as a platform for component-based software, embedded systems, and smart cards. This means Java is poised to play an important enabling role in e-commerce as these systems move from etherware to reality. Java components (aka JavaBeans) are appearing at a rapid pace and encapsulate critical functionality for transaction-based systems. Java smart cards for e-commerce will debut soon.

But what of the hue and cry over security? Should people be so concerned about the security implications of Java that they disable Java in their browsers? Should developers avoid using Java in their systems in favor of other languages like C++? Should system administrators block Java content at the firewall (or better yet, can they)? Should business people avoid Java because of security problems? These are the some of the questions this book answers. The answers are nontrivial, and the issues are as complex as they are important.

Who Cares?

Java security is important to a number of distinct sets of people:

Web users, including one of the authors' 89-year-old grandmother, need to understand the risks of using a Java-enabled browser.

Developers of Java code that lives and works on the Internet need to keep security in mind when they are writing programs.

System administrators need to think carefully about how mobile code, including Java, impacts the security of the systems they run.

Business people need to understand what Java security risks are so they can make informed business decisions based on fact and not fiction.

As you can see, Java security issues are multifaceted. This book has useful information for all four groups, whose interests overlap in many ways.

Java security is a hot topic, but that does not make it an easy one. By itself, computer security is not well understood. Throw Java into the mix and things become even murkier. There is much confusion and misinformation floating around about Java and security. Beware of snake oil, impossible claims, and consultants who pretend to have all the answers. Also be aware that major vendors are just as capable of misinformation as fly-by-night companies. Skepticism, Rene Descartes' 300-year-old philosophical insight, is strangely relevant to computer security at the turn of the millennium. In fact, skepticism turns out to be an excellent strategy. Ask hard questions; you might be surprised by the answers.

Browser Beware

The most pressing security concerns surrounding Java impact millions of people—that is, anyone who browses the Web. Given that there are tens of millions of Netscape Navigator and Microsoft Internet Explorer users, the client security issue is no minor detail.[1] It turns out that a majority of the users of these browsers are also Java users, whether they know it or not. Java is built in to Netscape Navigator and Internet Explorer, so if you use either of these products, *you* are a Java user.

Just as all Internet users are taking security risks, all Java users are taking security risks. Because of the way Java works, computer security issues are a fundamental concern. Most Java code is automatically downloaded across

[1] Both the popular Netscape Navigator browser and the Microsoft Internet Explorer browser are capable of running Java applets.

the network and runs on your machine. This makes it very important to limit the sorts of things that Web-based Java programs can do. Simply put, a hostile Java program could trash your machine. Because Java is inherently Web-based, it provides crackers with an easy way to implement a Trojan Horse—a program that may seem innocent enough on the surface, but is actually filled with well-armed Greeks. Also of concern is the problem of computer virus propagation. Fortunately, the creators of Java have made a good effort to protect users from these hazards. For example, writing a Web-based Java virus as an applet would be very hard. (Writing a Microsoft Word macro virus like the *concept* virus is, by contrast, easy.) Because mobile code security is new, difficult, and complicated, Java's masters have not always been successful at protecting everyone all the time.

One goal of this book is to educate Java users about the risks that they incur by surfing the World Wide Web with Java-enabled browsers. This chapter provides a gentle introduction to Java and explains why Java is potentially dangerous.

Developer Concerns

Java security is essential to developers as well. As a platform, Java has much to offer in terms of security:

- Java has advanced cryptography Application Program Interfaces (APIs) and class libraries.
- Java includes language-level security mechanisms that can help make developing secure code easier.
- Some aspects of Java that make it more difficult to write insecure (unsafe) code.

This book explains how to use the security features built in to the Java environment inside your own programs.

That's not to say that developing secure programs with Java is trivial or automatic. Anyone who reads the newspapers or the trade press can see how often skilled programmers write code with security bugs. You can make almost as many gaffes developing security-critical code in Java as in any other language. Because of Java's security APIs and its position as a leading e-commerce platform, it is likely that Java will be used to carry out some very important activities. That means developers need to learn as much as they can about Java security. Know your enemy. Think about what might confront your code in terms of malicious attacks. Mitigate risks by designing, coding, and testing carefully.

A second goal of this book is to teach Java developers and project managers about the sorts of things that will confront their code in "the wild." If you're a seasoned Java developer (something that it was impossible to be a mere handful of years ago), this book will show you in great detail how the security model works. There are lessons to be learned from the Java attacks we cover. After all, like you, Java's designers and developers were serious about what they were doing. As we have seen, however, even the most subtle bug can be turned into a security disaster.

System Administration and Java

Today's system administrator is seriously overworked, and security is a big part of the problem. The days of the isolated Local Area Network (LAN) are behind us. Now, most networks are connected directly to the Internet, which means security is more important than ever. Some early adopters and sites with a lot to lose try to protect themselves with advanced security mechanisms such as firewalls, secure shells, and virtual private networks. Many sites, however, have a long way to go before they are "secure enough" (whatever that means). Mobile code systems, including Java, make administering site security trickier.

The problem is that users want Java content, but system administrators don't want to take on unnecessary risks. This is a classic example of the well-known tradeoff in computer security between functionality and security. Computer security boils down to managing risks, which in turn implies that the way to make better-informed decisions is to get a handle on the risks.

A third goal of this book is to present an informed discussion of the real risks of mobile code. Burying your head in the sand like an ostrich is not a good solution, because security problems are unlikely to miraculously disappear. However, the risks do not necessarily warrant throwing the Java baby out with the bath water. Such a move may leave your users high and dry.

Even if the risks turn out to be too much to bear (a decision that is very much context dependent), system administrators need to be wary of snake-oil "solutions" to the mobile code problem. There are a number of products on the market that purport to improve Java security. The question is, do they work? We will delve into these issues as well.

Java Gets Down to Business

Making informed business decisions at the edge of the technology curve has never been an easy task. In addition to the technological concerns discussed earlier, there are often intangible factors to consider. What impact will perceived security risks (whether justifiable or imagined) have on potential customers? Is Java the best platform to use when designing e-commerce systems? How will

the use of Java within an enterprise affect security risks? What are the security challenges in designing and deploying database-backed Web servers and three-tier applications?

It is surprising that some of the same companies that disallow the use of Java (often for silly reasons) expect their customers and business partners not to disallow Java. The information in this book can help business managers and leaders make better decisions about Java security. Good data are essential to decision-making, but sometimes good data are hard to find.

Mobile Code

The Java programming environment from Sun Microsystems is designed for developing programs that run on many different kinds of networked computers. Because of its multiplatform capabilities, Java shows great promise for relieving many of the headaches that developers encounter when they are forced to migrate code between different types of operating systems. Code that is written in Java *should* run on all of the most popular platforms—everything ranging from Macintosh and Windows/Intel machines to Linux and Solaris boxes.

Recently, the cross-platform capabilities of Java have been called into question. This has led Sun's marketing phrase "write once, run anywhere" to be reinterpreted by skeptics as "write once, test everywhere." Part of the problem is that not all implementations of Java are completely interoperable with Sun's version. Disagreement over what constitutes Java has generated at least one high-profile lawsuit. Most people, including a majority of Java developers, would like to see Java become a standard so that what happened to C (which was itself supposed to be a cross-platform language) doesn't happen to Java.

In any case, a nice side effect of Java's built-in portability is that one special kind of Java program (popularly known as an *applet*) can be attached to a Web page. More technically speaking, applets are embedded into a Web page's hypertext markup language (HTML) definition and executed by Java-savvy browsers.[2] Such Java-enabled browsers automatically download and begin running any Java applet they find embedded in a Web page. Java code's ability to run on many diverse platforms makes such "magic" possible.

The ability to dynamically download and run Java code over the Net has led some computer pundits to proclaim that the age of truly component-based software development may actually have arrived. The idea is that instead of buying huge monolithic word processing behemoths with hundreds of obscure features

[2] Java has some competition as an environment for creating executable content. Other languages with a similar bent are JavaScript, Safe-Tcl, Telescript, Word macros, Excel macros, ActiveX, and Postscript. Many of the security lessons in this book apply to those languages as well. Later in this chapter we will examine ActiveX security issues more closely.

that most users will never need, users can instead "create" a personal word processor on the fly out of Java building blocks. This modern sort of programming is akin to building a large toy ship out of Legos blocks. Or, more realistically, the process of creating a component-based software product could be likened to building a highway bridge out of standardized structural components.

Sun is advocating a Java component architecture called *JavaBeans*. A number of companies are creating sets of JavaBeans for various purposes. If these efforts are successful, developers will be able to create programs by putting together sets of prefabricated Beans as illustrated in Figure 1.1. Microsoft's Component Object Model (COM) is very much oriented this way, although it is not specifically designed to use Java. Component-based software has its own interesting security implications and open questions. For example, how can the developer of a system trust a component manufacturer not to have (purposefully or accidentally) introduced security holes into the system? How can a component manufacturer anticipate all uses to which a component will be put? And so on. These sorts of questions are the topic of current research, including some by the authors of this book.

Thinking even farther into the future, one can imagine a fundamentally new kind of computer document that contains the word processing, spreadsheet, and database software that was used to create it. Using a document's embedded components, a writer or editor could modify the document on any platform. The built-in components would allow different people using different machines to edit the document without worrying about the kind of computer they are using or file type compatibility issues. If Java is developed to its full potential, this future world may not be far off.

The new idea behind all of these exciting aspects of Java is simple: the ability to send data that can be automatically executed wherever it arrives, anywhere on the Net. Java is an implementation of *executable content*, or *mobile code*. This

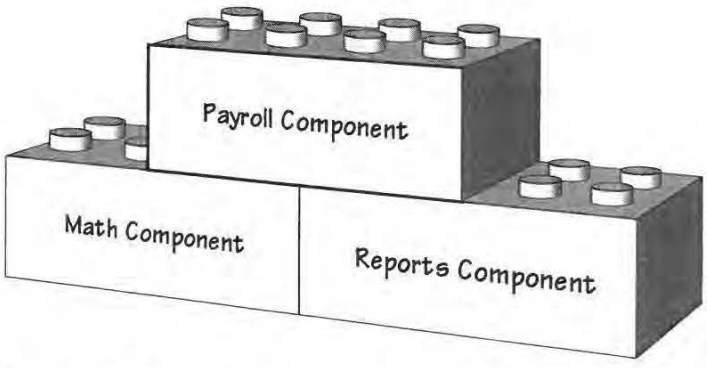


Figure 1.1 Component-based software allows a designer to create large applications from standardized building-blocks.

Components in Java are known as JavaBeans. The idea of using pre-fabricated components to build large-scale applications will likely do for software what the Industrial Revolution did for manufacturing.

powerful idea opens up many new possibilities on the World Wide Web. For the first time it is possible to have users download from the Web and locally run a program written in a truly common programming language.

These features of Java are certainly exciting; however, Java's fantastic potential is mitigated by serious security concerns. Security is always an issue when computers are networked. Realistically speaking, no computer system is 100-percent secure. Users of networked computers must weigh the benefits of being connected to the world against the risks that they incur simply by connecting. In practice, the goal of a security policy is to make such tradeoffs wisely.

One of the key selling points of Java is its use as a "cross-platform" language for creating executable content in the highly interconnected world of the Internet. Simply by using a Web browser, a Web surfer can take advantage of Java's cross-platform capability. Of course, the activity of locally running code created and compiled somewhere else has important security implications. These implications are one focus of this book.

The same risks and benefits that apply to connecting to the Internet itself directly apply to using the Java language. As you will see, these concerns become particularly critical when "surfing the Web." The same technology that allows Java applets to enliven once-static Web pages also allows unscrupulous applet designers to invade an unsuspecting Java user's machine. With Java applets showing up everywhere, and many millions of people using Java-enabled browsers, it pays to know where you are pointing your browser.

The Power of Networking

Networking has changed the face of computing. We once thought of computers as calculating machines, but now most people rightly view them primarily as communication tools. An Internet connection is as essential a part of today's computer as a disk drive. The move toward a globally networked world has been significantly furthered by Java.

The Internet: A World of Connections

Since its birth in the early 1970s as a 12-node network called the ARPANET,[3] the Internet has exponentially exploded into a worldwide network that provides a central piece of the planet's information infrastructure. Figure 1.2 shows the growth pattern of the Internet from its humble 12-host beginning through today's some 30-million registered addresses.

[3] ARPA (now DARPA) is an acronym for the United States Department of Defense's Advanced Research Project Agency that sponsored initial research on networking computers. DARPA currently supports many research projects in computer security, including work by the authors.

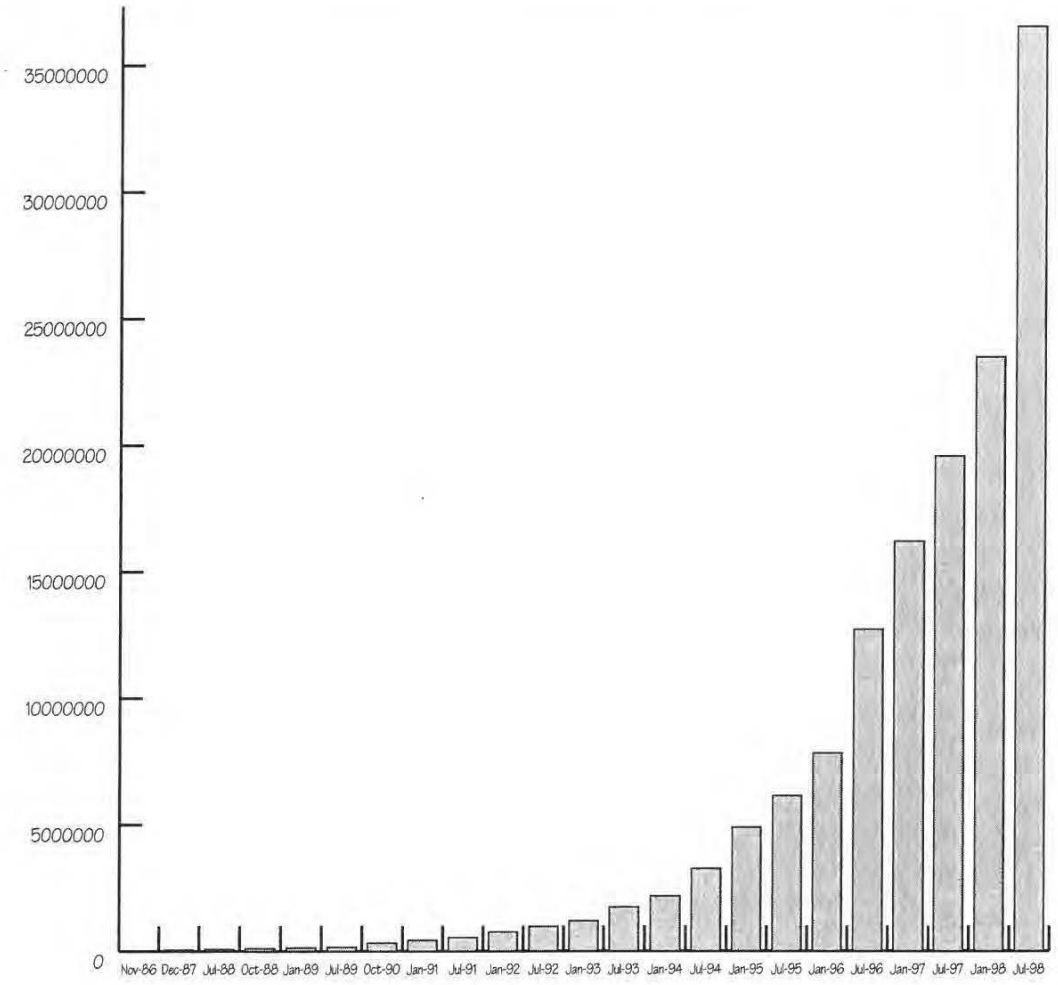


Figure 1.2 Growth of the Internet since its early days as the ARPANET.
Data is from Network Wizards (*www.nw.com*). The Internet continues to grow at an astounding rate.

Connecting computers together in a network allows computer users to share data, programs, and each others' computational resources. Once a computer is put on a network, it is possible to access a remote machine in order to retrieve data or to use its CPU cycles and other resources. Along with this ability comes concern about security. Computer security specialists worry about issues such as:

- Who is allowed to connect to a particular machine
- How to determine whether access credentials are being faked
- Who can access which resources on a shared machine
- How to protect data (especially in transit) using encryption
- How and where to collect and store audit trails

Whenever machines are networked, these concerns must be addressed.

The Internet, the world's largest network of machines, has encouraged research into these security issues. Mechanisms now in place go beyond simple password authentication, to firewalls and security checking tools such as SATAN, ISS, and Ballista. New ideas in computer security are constantly becoming available on the Net. Security approaches currently in preliminary use include encryption-based authentication, encrypted communications, and intrusion detection based on Artificial Intelligence (AI) [Hughes, 1995; Garfinkel and Spafford, 1996; Ghosh, 1998]. Computer security has recently matured into a substantial commercial enterprise as well. As in any new field, however, there is as much hype as there are barrels of snake oil. If it sounds too good to be true, it probably is. Buyer beware.

The Web: Making the Internet Enticing

One of the driving forces behind the exponential growth of the Internet in the last several years has been the introduction of the World Wide Web. In 1992, Tim Berners-Lee, a British researcher at the CERN physics facility in Europe, invented the Web, a new way to use the Internet. His invention introduced hypertext markup language (HTML) and Web browsing to the world. In 1993, Marc Andreessen helped to write the Mosaic Web browser while affiliated with the National Center for Supercomputer Applications (NCSA). He later cofounded the company now known as Netscape Communications. Though it may be hard to believe, the Web is only a few years old.

Before the invention of the Web, the Internet was almost exclusively text based. Researchers used it to transfer files to one another and to keep in touch via email. After the Web was invented, it suddenly became possible to see graphical pages sent across the Net by Web servers. These Web pages can include pictures, sound, video, and text, as well as hyperlinks to related pages. A Web browser provides an easy-to-use, intuitive interface for "surfing," or traveling around the Web, visiting other people's pages. Figure 1.3 shows how a typical Web page looks when viewed with the Netscape browser.

Ease of use is partially responsible for the astonishing numbers of Web users, and perhaps for the sense of safety that most Web users seem to enjoy. In addition, creating Web pages is a relatively simple process. HTML editors like Netscape Navigator Gold and Microsoft FrontPage make the job especially easy. Given one of these editors and a Web server, you have all the pieces you need to create your own Web site. An alternative to using an HTML editor is to write HTML code directly. Either way, this snazzy HTML facade makes the Internet more attractive than ever.

As shown in Figure 1.4, the Web has grown just as quickly as the Internet itself. The figure charts a conservative estimate of the number of Web servers on the Net. It is these servers that allow people to make Web pages available

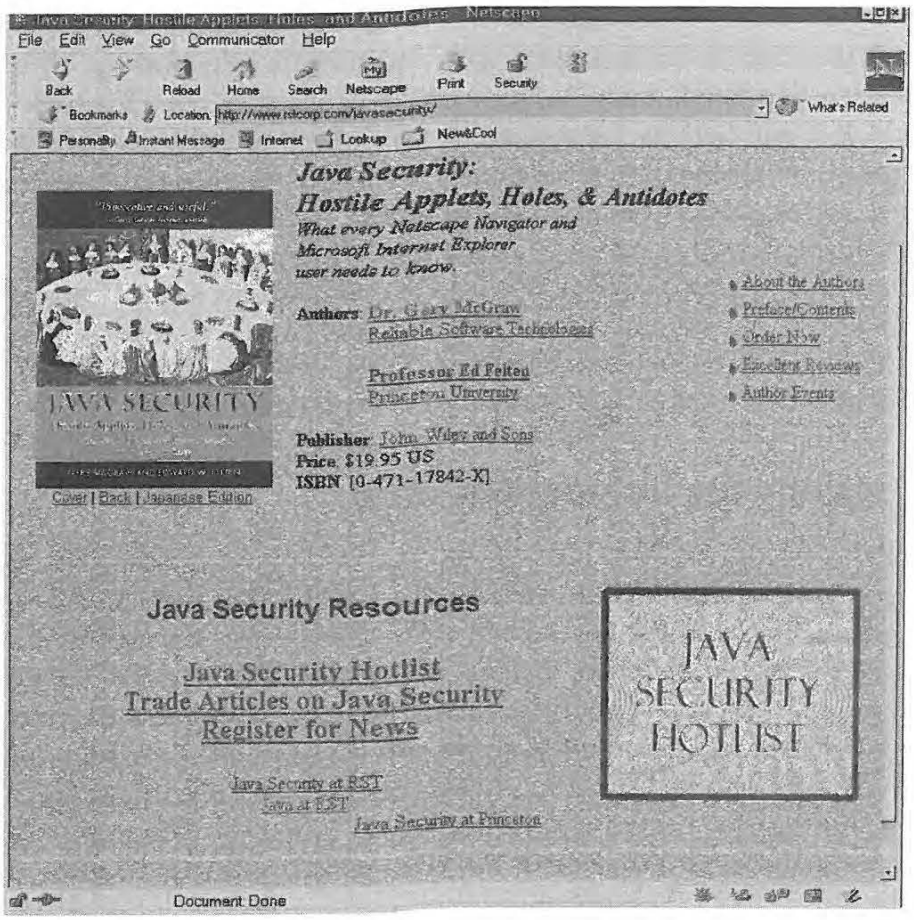


Figure 1.3 A view of this book's companion Web site (*www.securingjava.com*) as displayed by Netscape Communicator.
All current Web browsers include the capability of running mobile code automatically.

to everyone. The figure does not properly reflect the number of Web pages that are out there, which some people number in the hundreds of millions. Keep in mind that a server has the potential to serve hundreds or even thousands of pages for multiple users simultaneously.

Java: Spicing Up the Web

HTML-based Web pages are certainly a big step up from using the obscure, text-based Unix incantations of ftp, news, gopher, wais, and telnet to get around on the Net; however, they also have a major drawback. Much like the page that you are reading now, Web pages are static. Wouldn't it be better to have interactive Web pages that dynamically change themselves according to feedback from a user? Wouldn't it be better to program your Web pages to accept input, compute results, and then display them?

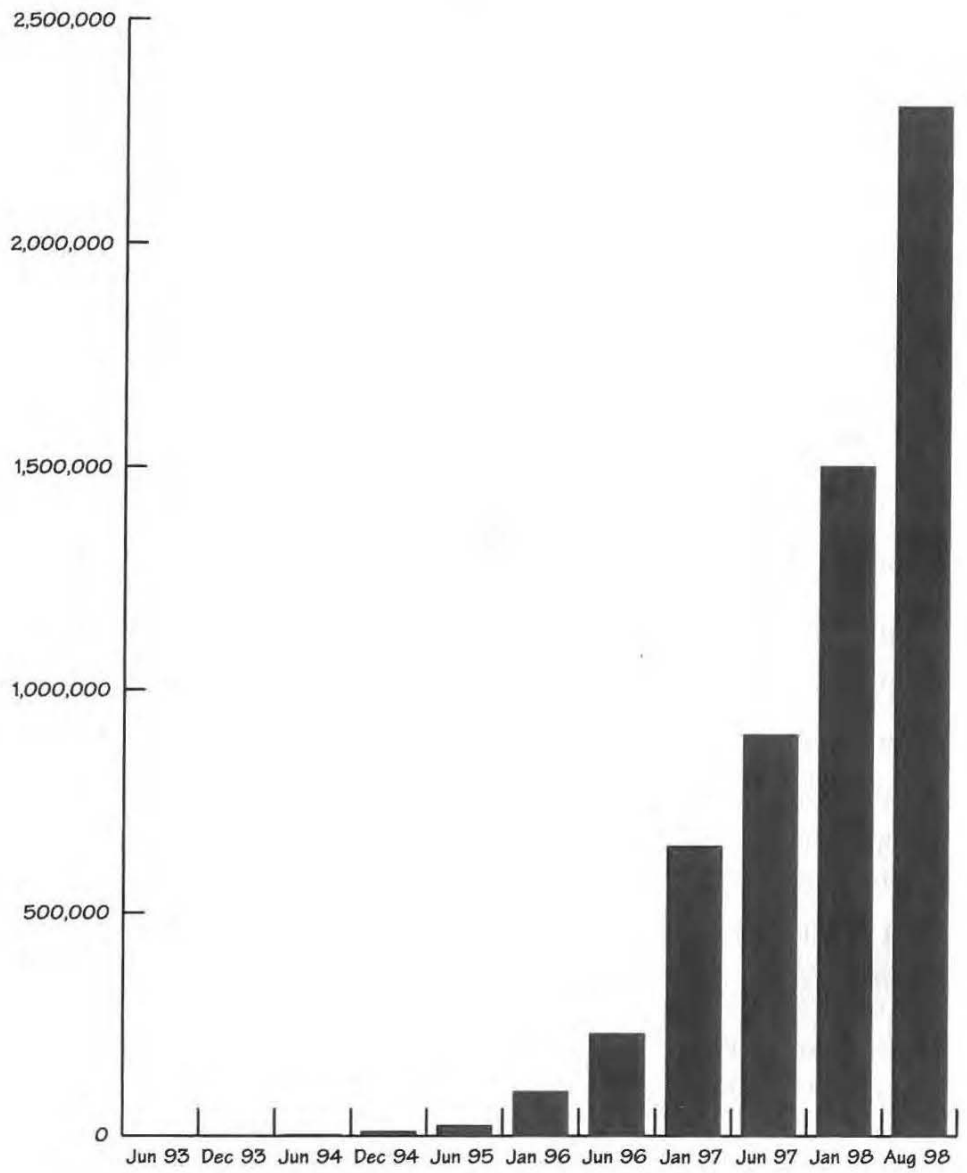


Figure 1.4 Growth of the World Wide Web, shown as the number of Web servers, since its introduction in 1993.
Data from the Internet Society (*www.isoc.org*).

This sort of dynamic activity should ring a bell. After all, programming languages allow people to program machines to do just these sorts of things. Why not make a programming language for the Web?

That is the essence of Java. Java is a full-featured programming language that allows programmers to compose executable content for the Web. The Java language is designed to be usable on all platforms so that code can move from one machine to another and still work, regardless of the kind of machine it ends up

on. Cross-platform compatibility has always been a stumbling block in previous attempts to create programming languages for executable content. Mobile code can only truly be mobile if it can be executed on all platforms without porting and recompiling!

In order to allow Java to run on a computer, the administrator must first install a Java Virtual Machine (JVM), or a browser that includes a Java VM. The JVM interprets Java instructions and translates them into machine-specific instructions. This allows Java to be run on many different types of machines.[4] For old timers, the whole idea is reminiscent of P-code from the 1970s.

Having a well-defined, platform-independent definition allows Java to get around problems that have plagued the C programming language, making C less platform independent than its designers intended. Unlike C programs, Java programs are not hampered by machine-dependent structures such as:

- Byte ordering (low or high endian)
- Pointer size (16 or 32 bit)
- Integer size (16 bit, 32 bit, or 64 bit)

Java's careful definition shields it from these platform-specific elements of programming. Each Java VM is written to a specific platform and translates the more generic Java instructions into platform-specific instructions.

Java has upped the ante on the Web. The best Web pages now include Java applets that do everything from displaying selectable news tickers to providing front-end graphical user interfaces (GUIs) for internal databases. There are even some Web-based videogames written in Java. Java applets have become commonplace.

The Promise of Java

Java is by far the most popular implementation of Web-based mobile code. Lesser-known competitors include JavaScript, Safe-Tcl, Telescript, Word macros, Excel macros, ActiveX, and Postscript. Each of these systems raises its own security issues. Any document-embedded scripting language that can be transferred around the Net and run on different machines falls under the classification of executable content.[5] Propelled by the marketing powers of Sun Microsystems and IBM, the Java wave is still building. Java avoids the interactive content limitations that were built in to forms and

[4]All Netscape Navigators since 2.0x and Microsoft Internet Explorers since 3.0 include a Java VM that can interpret the Java byte code making up a Java applet.

[5]Note that many of the lessons of this book apply directly to all of these varieties of mobile code since the crux of the security problem is the idea of running untrusted code safely.

CGI (Common Gateway Interface) scripts.[6] Java's power lies in the ability to program complete applications in a real programming language that can then be dynamically distributed and run by virtually any user over the Web.

Downloading Code: Not a New Problem

In the early days of the Internet, everyone agreed that downloading arbitrary binaries and executing them on your machine was a bad idea. Of course, most people did it anyway. By the mid 1980s, there was a lot of freeware and shareware out there to be downloaded. To find it, you could use `archie`, which provided a way to search a large index of anonymous ftp content. Once you dug up some leads (often several ASCII pages worth), you chose your target and `ftp`'ed what you needed. Then you installed and ran it.

The risks of running some random person's downloaded-from-the-Net code on your machine are clear. If the code has a virus attached, your machine can be infected. If the program is a Trojan Horse that appears to be doing something useful while it is actually doing something nefarious, your machine can become "owned" by someone else. This is especially dangerous for machines connected to the Net. How can we be sure that a program that someone says is useful hasn't been hijacked to do something nasty?

When it works flawlessly, the Java security model provides one possible answer to this question, as it was designed to allow untrusted programs to be run on a computer safely. As we will see, the base Java security model is meant to counter the threat of viruses and other forms of attacks. But in the early days of the Net, Java did not yet exist. (To be completely accurate, Java was evolving in the early 1990s from an embedded platform called Oak that was meant to be used for smart devices like that Internet-enabled toaster you've heard so much about.)

Back to our history . . . The question in the late 1980s was, how could a user be sure that a program had not been hijacked (or Trojan'ed)? Checksumming provided part of the answer. A checksum is a simple computation performed on a piece of code to provide a digest, or "thumbprint," of a program. (Combine this with digital signatures and you have a system that can provide both *data integrity* and *authentication*, which is most desirable, as we will discover in Chapter 3.)

Not many people were into checksums back then, but they existed for at least a few anonymously downloadable programs. Of course, who was to say that

[6] These limitations had mainly to do with the fact that CGI scripts run on the server side, whereas Java applets run on the client side. CGI scripts trade off client-side security risks for risks induced on the Web server on which they reside. They are a common target of cracker attacks. See [Rubin et al., 1997] for more.

a program's checksum hadn't been tampered with? In reality, most people either ignored the risks or chose to live with them.

Skipping the advent of `gopher`, which most people pretty much ignored anyway, the next big thing was the Web. As discussed in the last section, the Web got its start in 1992. At first, the Web was static. Java changed all that, making it possible for a Web server to provide programs as content. Java applets are these programs. The dangers of mobile code and systems for addressing these dangers are the focus of this book. But there's still a drawback, even with the power that Java adds to the Web—the only way to tell when new content has been added to a Web page is to surf back over and find out. That's where *push technology* comes in.

Push: Too Much of a Good Thing?

As if surfing the Web with a Java-enabled browser isn't bad enough security-wise, another new step in mobile code delivery appears to be "push" technology. Push provides a way to have information (including mobile code) automatically flow to your machine—without you even asking for it! (Well, you do have to set things up once in the beginning, but after that, no more clicking.) Now the inconvenience of clicking on a hyperlink is completely removed. Heck, you don't need to make any decisions at all. Just sit back and watch the content (which may include Java applets, ActiveX controls, and client-side scripts) come to you. With push it is possible to subscribe to "channels" that do things like provide constant stock information, news headlines, and (most dangerously of all) software updates.

There are many push systems out there. Two of the most popular are Castanet by startup Marimba, and PointCast by PointCast, Inc. The security systems of Castanet and PointCast are briefly covered in an article written by McGraw entitled "Don't Push Me: The Security Implications of Push," which is available at *www.developer.com/techfocus/123097_pushsec.html*. Push channels are now available in both Internet Explorer 4.0 and Netscape Communicator.

First off, push is not very well named. It should actually be called "timed pull." Most systems, including PointCast, work by having a tuner program, which functions like a fancy browser, issue HTTP requests for information from a push server. (This is the "pull" part.) Once requested, the information comes back across the Internet as HTML-based HTTP traffic and is eventually displayed in a special window. PointCast is set up to take over the screen when the computer is not in use, much like a screensaver program. Every once in a while, the program will wake up and check for new information, which is grabbed in chunks and sometimes cached. (This is the "timed" part.)

Let's get this straight: It is still a really bad idea to download and run arbitrary binaries off the Internet. Automating things so that this happens more easily, behind the scenes, doesn't serve to make it any less dangerous. We've gone from having to request binaries through the text-based ftp interface and install them, through clicking on a hyperlink (the Java model), all the way to having content come to you.

In the meantime, security issues have yet to be properly addressed. How do you know that the information a push server is sending you is secure? How do you know that the update that was just pushed onto your PC is really from the company that developed the software? These questions are familiar ones to people interested in security. What we need to make push systems safe is strong authentication, foolproof data integrity, and trust in the broadcasters. Current push systems are only beginning to address security concerns.

Java in a Demitasse

The security concerns raised in this book apply equally to both Java users and Java developers. Using Java is as easy as surfing the Web. The simple use of Netscape Navigator, Internet Explorer, or any other Java-enabled browser to run Java applets is a risky activity. In order to really understand these risks, it is important to gain a deeper understanding of how Java really works. Here is a short but thorough introduction to the Java language.

The Java development environment comprises three major components:

1. A programming language that compiles into an intermediate, architecturally neutral format called *byte code*
2. The Java Virtual Machine that executes the byte code
3. An execution environment that runs on the JVM and provides some base classes useful for building complete applications

Figure 1.5 shows how these three parts of the Java environment work together to provide executable content for the Web. The Java Developers' Kit (JDK) is provided free to all. It includes the three parts of the Java environment outlined here. To get your own copy, point your browser to URL *java.sun.com*.

Because Java byte code runs on the Java Virtual Machine, it is possible to run Java code on any platform to which the JVM has been ported. Some Web browsers, such as Netscape and Internet Explorer, include an encapsulated version of the JVM. Using their built-in VMs, such Java-ready browsers can automatically download and execute Java applets when a user accesses an HTML Web page including the <APPLET> tag.

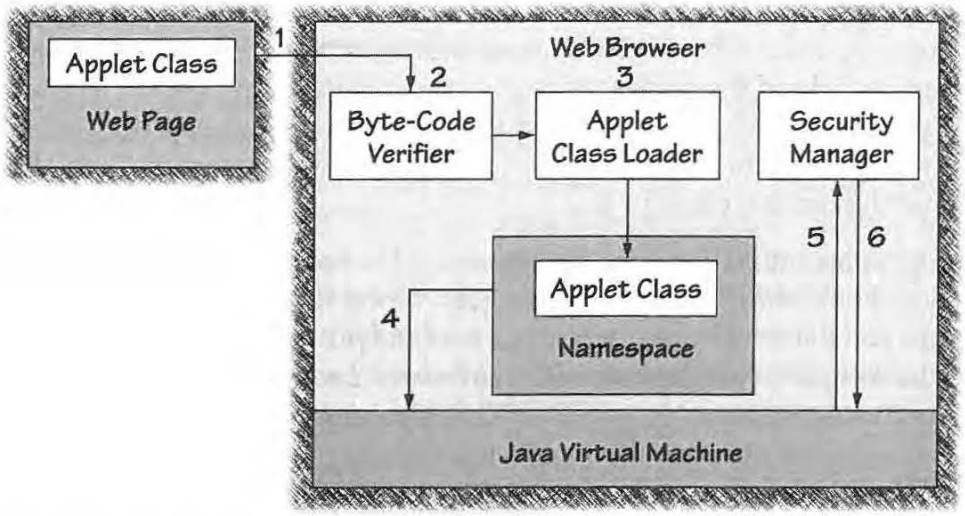


Figure 1.5 How Java implements the original sandbox approach to mobile code.

Java source code is compiled into Java byte code which is transferred across the Web to the browser that requested it. The HTML in a Web page specifies which code is to be fetched from the Web server. The requesting Web browser, prompted into action when a user clicks on a hyperlink, (1) fetches the code from the Web, (2) verifies it, (3) instantiates it as a class or set of classes in a namespace. The applet executes and (4) invokes a dangerous method (5) causing the Security Manager to be consulted before the method runs. The Security Manager (6) performs runtime checks base on the calling class's origin and may veto some activities.

The Java Language

One of the first public introductions to Java came in the form of a whitepaper released by Sun (and since updated many times) [Sun Microsystems, 1995]. An especially pithy sentence from that document attempts to describe the fundamental aspects of Java all at once. It reads:

> **Java**: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded, and dynamic language.

Quite a collection of buzzwords. In fact, some people joke that Java is "buzzword compliant." This book is concerned mostly with the security claim, of course, but in order to understand the implications of Java for computer security, you need to grasp the other important characteristics of the language first.

As the quote claims, Java has many interesting features. They will be briefly introduced here. Pointers to more information on Java can be found on page 31. The Java language is:

> **Object-oriented**: Unlike C++, which is an objectivized version of C, Java is intrinsically object-oriented. This changes the focus of programming from the old procedural way of thinking (as in C and Pascal) to a new data-centric model. In this new model, data objects possess associated methods. Methods perform actions on data objects. Every Java program is composed of one or more classes. Classes are collections of data objects and the methods that manipulate these data objects. Each class is one kind of object. Classes are arranged in a hierarchy such that a subclass inherits

behavior and structure from its superclass. Object-oriented languages were designed using the physical world as a metaphor. Classes communicate with each other in much the same way that real physical objects in the world interact.

Strongly typed: This means that a Java program cannot arbitrarily access the host computer's memory. Memory access by Java programs is limited to specific, controlled areas having particular representations. Type safety is verified when code is loaded into the JVM by the Byte Code Verifier *(see Chapter 2, "The Base Java Security Model: The Original Applet Sandbox")*. In addition, runtime checks on type safety (such as checks for array bound overflow, type incompatibility, and local-versus-remote code security policy) are all handled by the Java Virtual Machine. As we shall see, type safety is essential for Java security. In fact, a majority of serious Java security attacks target the type system.

Multi-threaded: Java programs can execute more than one task at the same time. For example, a multimedia Java applet may want to play a sound file, display a picture, and download a file all at once. Since Java is multi-threaded, it supports the concurrent execution of many lightweight processes. An obvious benefit of this capability is that it improves the performance of multimedia applications at the user end. Java's built-in support for threads makes designing such applications far easier than it is in C and C++. Primitives for synchronization are also provided in Java.

Java has other important characteristics adapted from modern programming languages such as Scheme (a popular dialect of Lisp) and ML. In particular, Java uses:

Garbage collection: Memory management is usually handled in one of two ways. The old-fashioned approach is to have a program allocate and deallocate memory itself. This approach allows all sorts of insidious errors and hard-to-squash bugs. C, for instance, uses this method. By contrast, Lisp introduced the modern concept of garbage collection in 1959! Garbage collection requires the system (rather than the programmer) to keep track of memory usage, providing a way to reference objects. When items are no longer needed, the memory where they live is automatically freed so it is available for other uses. Java provides a garbage collector that uses a low-priority thread to run silently in the background. Java's memory management approach has important implications for the security model since it prevents problems associated with dangling pointers.

No pointers: This is also a feature of Java's modern memory management scheme. Instead of allowing access to memory through pointers, memory is managed by reference. The crucial difference between references and pointers is that references cannot be manipulated through arithmetical means (as can pointers). This eliminates many potential bugs. Pointers are one of the most bug-prone aspects of C and C++. Eliminating pointers has the effect of making Java a much more reliable and safer language.

Exception handling: This defines how the program will manage an error condition. For example, if a Java program tries to open a file that it has no privilege to read, an exception will be thrown. Exception throwing and catching is a system for gracefully managing runtime errors that might otherwise crash a system. This is a good idea if you are concerned about security.

Dynamic linking: Software modules (classes in Java) are linked together as they are needed. The Java language knows where it should look for classes that need to be

linked while a Java program runs. By contrast, C has a linking phase during which all needed constructs are linked before the program is run. The linking phase in C is static since library functions are assembled together with other code into a complete executable at compile time. Dynamic linking makes it easier to keep Java programs up-to-date since the latest version of a class will always be used. This can turn out to be a problem for programs that expect a class to behave the way it has in the past and are surprised when a new version appears. Version control and software assurance become much more complicated with dynamic linking too. Java finds classes that it needs by searching for them in locations specified in the CLASSPATH environment variable (though the system is undergoing revision for Java 2). (As we will discuss in Chapter 2, it turns out to be very hard to ensure type safety when dynamic class loading is allowed.)

Though it has more than doubled in size since its original introduction, Java is still a relatively simple language. This is especially apparent when Java is compared with C and C++ [Daconta, 1996]. In C, there are often many possible ways in which to do the same thing. Java tries to provide only one language mechanism with which to perform a particular task. Also, Java provides no macro support. Although some programmers like using macros, macros often end up making programs much harder to read and debug.

The designers of Java made their language simple by removing a number of features that can be found in C and C++. Things that were removed include the `goto` statement, the use of header files, the `struct` and `union` constructs, operator overloading, and multiple inheritance. Together with the elimination of pointers, removal of these aspects of C and C++ makes Java easier to use. This should result in more reliable code.[7]

We will revisit the impact that Java's features as a language have on security in Chapter 2.

Portable Byte Code and the Java Virtual Machine

The second major component of the Java development environment is the Java Virtual Machine. The VM makes Java's cross-platform capabilities possible. In order to run Java byte code on a new platform, all that is required is a working VM. Once the VM has been ported to a platform, all Java byte code should run properly.

Making a byte code/VM pair that works well on many varied platforms involves setting a few things in stone. Java has variables that are of fixed size and of fixed format. An `integer` in Java is always 32 bits, no matter what the word size of the machine running Java. Making data formats machine independent and compiler independent is crucial to making Java truly

[7]Some experts' opinions about Java and reliability differ, however. For an interesting critique of Java, see [Lewis, 1996].

portable. The very different way in which variables are managed on different C platforms causes no end of portability problems for C programmers.

The VM also makes use of symbolic data stored inside of Java byte code files. Java byte code contains enough symbolic information to allow some analysis of the byte code before it is run. This is one way the Java environment ensures that Java's language rules have been followed by the compiler—something critical to security. Rules checked include, for example, type safety rules, and ensuring that certain things claiming to be of a certain type actually are of that type. Since the Java byte code Verifier is a critical part of the security model, it is discussed in detail in Chapter 2.

Using a Virtual Machine has obvious important repercussions for the Java approach. The VM makes portability possible, and it helps to ensure some of Java's security features. Since Java is often implemented using an interpreter, speed can be an issue. Interpreted languages are inherently slow because each command must be translated to native machine code before it can be run. With a compiler, this work is all done ahead of time, when an executable is created for some particular platform. Without just-in-time (JIT) and hotspot compilers, Java's interpreted code is about 20 times slower than native C code. When this new technology is used, Java speeds begin to approach native C.

Reusable Class Modules

The third part of the Java development environment is a set of predefined classes that implement basic functionality. The "personal" version of the JDK includes, for example, an Abstract Windowing Toolkit (AWT). These classes provide a set of graphical user interface (GUI) tools for creating windows, dialogue boxes, scrollbars, buttons, and so forth. Java also includes classes for full network support that provide application program interfaces (APIs) for sockets, streams, URLs, and datagrams. A POSIX-like I/O system with APIs for files, streams, and pipes makes the environment comfortable for experienced Unix programmers. Classes are grouped together into packages according to their functionality. Table 1.1 lists the packages included in the Java Developers' Kit (JDK) version 1.1. Note that Java's core classes have grown significantly in the last few years.

The predefined Java classes provide enough functionality to write full-fledged programs in Java. Using the predefined classes as primitives, it is possible to construct higher-level classes and packages. Many such home grown packages are available both commercially and for free on the Net.

Table 1.1 Packages Supplied by the JDK (version 1.1) Provide Multiplatform Primitives from Which Complete Applications Can Be Assembled

Java In A Nutshell [Flanagan, 1997] is an excellent reference describing these packages

PACKAGE	DESCRIPTION
java.applet	The applet class.
java.awt	Abstract Windowing Toolkit: The AWT provides graphics, GUI components, and layout managers. A new event model was introduced with JDK 1.1.
java.awt.datatransfer	Inter-application data transfer support, including clipboard cut-and-paste.
java.awt.event	Classes and interfaces for the new AWT event handler.
java.awt.image	Image processing classes.
java.awt.peer	Interface definitions for GUI components and platforms.
java.beans	The JavaBeans API for creating reusable software components.
java.io	Input/output classes: A relatively large number of classes for I/O.
java.lang	Central Java language classes: Defines `Object`, `Class`, and primitive types.
java.lang.reflect	Classes that allow a Java program to examine Java classes and to "reflect" on its own structure.
java.math	Two classes that support arithmetic on arbitrary-size integers and arbitrary-precision floating-point numbers (important for cryptography).
java.net	Networking classes.
java.rmi	Classes and interfaces for Remote Method Invocation.
java.rmi.dgc	Distributed garbage collection.
java.rmi.registry	Classes and interfaces for tracking, naming, and advertising remote objects.
java.rmi.server	The heart of the RMI system.
java.security	Classes and interfaces that define fundamental cryptographic services. (See Chapter 3.)
java.security.acl	Access control list interfaces.
java.security.interfaces	Interfaces required for the Java Security API's implementation-independent design.
java.sql	Java Database Connectivity (JDBC) API.
java.text	Classes and interfaces for internationalization.
java.util	Miscellaneous but critical classes. These classes are required for many others.
java.util.zip	Classes for manipulating zlib, ZIP, and GZIP file formats.

The World of Java Applications

In the early days of Java's popularity, most Java programs took the form of *applets*, small programs that were attached to Web pages and loaded and run in Web browsers. As Java developed, people began to write substantial applications in Java, using it simply as an improved version of traditional languages such as C.

Java has always been good for more than writing applets, and the world is now catching on to that fact. Java is really a good platform for any application that needs to be extended or customized, perhaps across the network, after it is deployed. A browser is only one example of such an application.

Another increasingly popular use of Java is in Web servers. Many servers have extension mechanisms, but the Java Servlet API provides a particularly flexible and compelling vehicle for extending a server with new application-specific or site-specific functions. Most major Web servers now support the Java Servlet API. Compared to browsers, servers present more difficult security challenges, since servers have more stringent reliability requirements and store more valuable data.

Java's features also make it a good platform for creating new server-type applications. With natural support for multithreading, database access, and networking, Java gives developers a natural leg up in designing such applications. For these reasons, Java is being used increasingly in enterprise computing.

One common structure for such systems uses a "three-tier" architecture. A traditional database server acts as the "back end" tier, storing and managing the data needed to support a business application. The middle tier is a Java-enabled specialized server that interacts with the database and implements the "business logic" needed to manage client interactions with the system. The "front end" tier is a Java applet that runs in the client's Web browser and provides a convenient user interface so that users can interact naturally with the system. Three-tier systems put together several uses of Java and, as a result, face a wide array of security issues.

In addition to all of these applications in traditional computers, Java is being deployed in embedded devices such as smart cards, key rings, and pagers. Embedded applications are often involved in electronic commerce systems, adding yet another series of twists to our security story.

The growing variety of applications is reflected in the subject matter of this book. While the first edition focused almost exclusively on applet security issues, this edition encompasses the full breadth of today's Java applications. We want to provide you with the information you need to know to maintain security while building, deploying, managing, and using up-to-date, Java-based systems. As Java has gotten down to business, so has this book.

Trust, Applets, and Applications

Java is much more than simply a language for creating applets. In the early days of Java (less than a handful of years ago), it was important to distinguish applet code (which was typically treated as untrusted and relegated to the sandbox) and application code (which was typically treated as fully trusted built-in code). This distinction is no longer a useful one.

An alternative way to carve up the Java program space is to think about code in terms of levels of trust. Programs that are more trusted can be allowed to carry out potentially dangerous acts (like writing files). Programs that are less trusted will have their powers and permissions curtailed.

If we think about Java programs this way, it is still possible to make sense of the old distinction between applets and applications. Java applets are usually, though not necessarily, small programs meant to be run in the context of a Web browser. Obviously, applets involve the most client-side (or user) security concerns of any Java programs. In fact, Java's security policies originally existed in order to make applets feasible. The Java runtime enforces severe limitations on the things that applet classes may do [McGraw and Felten, 1996]. See *www.javasoft.com/sfaq* and Chapter 2 for details. In terms of the new trust-based distinction, applets are clearly treated as untrusted. This makes sense, since the origin of an applet is often unknown or unfamiliar.

In the early days of Java, Java applications had no such restrictions. In terms of our trust distinction, applications in Java before Java 2 were treated as completely trusted code. That meant applications could use the complete power of Java, including potentially dangerous functionality.

The reason the old distinction between applets and applications no longer makes sense is that today, applets can be fully trusted and applications can be completely untrusted. (Note the use of the word *can* in the previous sentence; we don't mean to say that applets are always trusted or that applications are never trusted.) In fact, depending on the situation, each and every Java program can be trusted, partially trusted, or untrusted. Sound complicated? That's because it is.

With the introduction of Java 2, Java includes the ability to create and manage security policies that treat programs according to their trust level. The mechanisms making up the base sandbox are still under there somewhere, but they serve merely as a default situation to handle code that warrants no trust. The interesting thing is that code that is partially trusted can be placed in a specially constructed custom sandbox. That means a partially trusted applet can be allowed to, say, read and write a particular file or make a network connection to a particular server. This is good news for Java developers who were chafing under the constraints of the restrictive original sandbox.

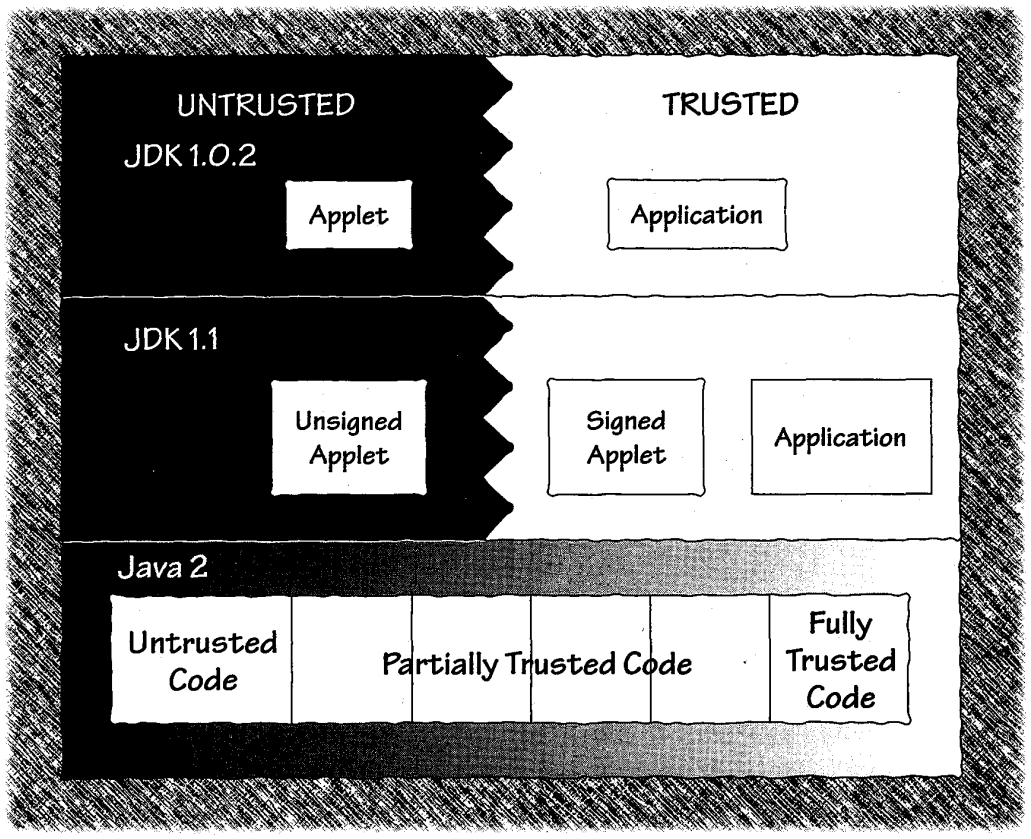


Figure 1.6 From black-and-white to shades-of-gray.

The distinction between applets and applications found to be useful during the JDK 1.0.2 days no longer applies to mobile Java code based on the Java 2 model. In fact, all along the real distinction behind the scenes was between fully trusted code and fully untrusted code. A black-and-white distinction between trusted code and untrusted code underlies both JDK 1.0.2 and JDK 1.1. By contrast, the Java 2 approach to trust management implements a policy-oriented shades-of-gray architecture. Under Java 2, code can be constrained or unconstrained regardless of whether it is applet or application code.

Figure 1.6 illustrates the way in which the old applet/application distinction can be recast in terms of black-and-white trust. It also shows the impact that Java 2 has on the black-and-white trust model, transforming it into a shades-of-gray trust model.

The Many Flavors of Java

Currently, a large and growing number of Java systems are running the gamut from Java gizmos (including Java rings), through smart cards with built-in Java interpreters, to complete Java Development Kits and IDEs. As with any platform meant to interact in a networked world, there are security concerns with each flavor of Java. This book discusses security risks that apply to all flavors of Java, but will focus on Java 2 and Card Java 2.0.

Counterintuitively, Java is both growing and shrinking at the same time. The JDK, now up to Java 2, is doubling in size with each major release. At the same time, embedded Java systems like Card Java 2.0 are stripping Java functionality down to bare bones. Both of these moves have important security implications. Java 2 involves fundamental changes to the Java security model as the Java sandbox is metamorphosing into a trust-based system built on code signing. Card Java 2.0 removes much of the sandbox, leaving smart card applets more room to misbehave.

All of Java's built-in security functionality, including the recently added authentication and encryption features (which began to appear with JDK 1.1), are available to Java application developers. This functionality makes it possible for an application to establish its own security policy. In fact, Java-enabled browsers do just that, determining the security policy by which all applets that run inside them must abide. For obvious reasons, an applet is not allowed to change the browser's (or for that matter, any application's) security model!

Securing Java

Security risks fall into four basic categories: system modification, invasion of privacy, denial of service, and antagonism. These four categories of risk are discussed in detail in Chapter 2. The first two of our risk categories are handled moderately well by Java; the second two are not. Risks are particularly egregious in Java since exploiting vulnerabilities is simply a matter of booby-trapping a Web page with a malicious applet or two. Chapter 4, "Malicious Applets: Avoiding a Common Nuisance," and Chapter 5, "Attack Applets: Exploiting Holes in the Security Model," discuss two distinct forms of hostile applets. Java applets with bad intentions—exploit scripts—are the equivalent of every security administrator's nightmare [Garfinkel and Spafford, 1996].

Java's designers are well aware of many of the risks associated with mobile code. To combat these risks, Java was specifically designed with security concerns in mind. The main goal was to address the security issue head-on so that naïve users (most of the millions of Netscape Navigator and Internet Explorer users) would not have to become security experts just to surf the Web.

In its default form, Java presents a multitiered approach to security. At a general level, the tiers include:

- Restricted access to file systems and the network
- Restricted access to browser internals

- A set of load time and runtime checks to verify that byte code is following the rules
- A system for signing code and assigning it some level of capability

The Java security model will be detailed in Chapter 2 and Chapter 3. Many claims have been made about the security of the Java language. We will try to separate reality from marketing hype in order to better understand the Java security model.

Java also provides a set of tools with which a developer can produce security-critical code (for both applets and applications). In addition to a number of advanced language features like array bounds checking and byte code validation, Java provides:

- A set of cryptographic APIs for standard algorithms
- Cryptography engines that provide the guts for a small subset of the APIs
- A strong, stack-based security system

Although this book is not just a guide to Java's security APIs, we will discuss Java's security functionality in detail. In particular, we will emphasize that no computer language as powerful as Java makes writing security-critical code automatic or easy.

How Does Java Security Stack Up?

As we have mentioned, Java is not the only game in town when it comes to mobile code. Other mobile code systems include JavaScript, Safe-Tcl, Telescript, Word macros, Excel macros, ActiveX, and Postscript. Of these systems, the one most often touted as a direct competitor to Java is Microsoft's ActiveX (sometimes called DNA depending on the whim of Microsoft marketeers). So what does ActiveX do for security, and how does it compare with Java's approach? Besides ActiveX, what other mobile code systems present security risks?

ActiveX Security Issues

The first thing to know about ActiveX is that it does not have an enforcement-related security model. It has a trust model that may be able to help you implement your own security policy. So the real question is: How does a trust model like ActiveX's compare with a sandbox like Java's?

Sandboxes and Signatures

There are two major approaches addressing the security concerns raised by mobile code systems: *sandboxing* and *code signing*. The first of these approaches, sandboxing, is an idea embraced by early implementations of Java (say, JDK 1.0.2). We extensively cover the Java sandbox in Chapter 2. The idea is simple: Make untrusted code run inside a box and limit its ability to do risky things. That is exactly what the Java security model aims to do.

The second approach, code signing, is how the ActiveX Authenticode system works. Binary files, such as ActiveX controls or Java class files, can be digitally signed by someone who "vouches" for the code. If you know and trust that person or organization, you may choose to trust the code that they vouch for. It is important to stress the fact that code signing is completely a matter of trust; there is no enforcement mechanism protecting you once you decide to trust a piece of code. The trust model implements authentication and authorization. What this means is that there is no such thing as ActiveX security enforcement! That's not to say signature-based trust models are not useful. They are. In fact, trust models will play an integral role in future security models for mobile code. Much more detail on code signing, especially as it relates to Java, is found in Chapter 3, "Beyond The Sandbox: Signed Code and Java 2."

Code Signing and ActiveX

ActiveX is a high-profile form of mobile code promoted by Microsoft. Note that in practice its "mobility" is completely constrained to one platform, however. As it is actually used today, ActiveX is language independent, but not platform independent, meaning that real ActiveX controls work only on Microsoft's Win32 platform (Windows 95, Windows 98, and Windows NT). Technically, these controls could be recompiled for other platforms, but virtually nobody currently produces controls for non-Win32 platforms.

One caveat: Comparing ActiveX and Java is somewhat like comparing apples and oranges, even though everyone does it. ActiveX is a component-based software model while Java is a language/platform. ActiveX should really be compared with Java components, JavaBeans. (In fact, some argue that the real religious Holy War between Java and ActiveX is destined to take place in the middleware arena and will be decided by the battle of component models [Lewis, 1998].)

ActiveX has been roundly criticized by computer security professionals since its approach to security is seen as lacking. Unlike the base Java security situation, in which an applet has to run in the sandbox and is limited in the sorts of things it can do, an ActiveX control has no limitations on its behavior once it is invoked. The upshot is that users of ActiveX must be very careful only to

run *completely trusted code*. On the other hand, Java users have the luxury of running untrusted code fairly safely.

The ActiveX approach relies on *digital signatures*, a kind of encryption technology in which arbitrary binary files can be "signed" by a developer, distributor, or certifier. Because a digital signature has special mathematical properties, it is very difficult to forge. That means a program like your browser can verify a signature, allowing you to be fairly certain who vouched for a piece of code (as long as people are carefully guarding and managing the private keys used to sign code). To make things easy, you can instruct your browser always to accept code signed by some party that you trust, or always to reject code signed by some party that you don't trust. The signature also supplies *data integrity*, meaning it can ensure that the code you received is the same as the code that was originally signed. Signed code cannot be easily hijacked and modified into a Trojan Horse.

The ActiveX system provides a black-and-white trust model: Either you trust the code completely and allow it to run unhampered on your machine, or you don't. That means trusting the wrong sort of code just once is all it takes. Once an *attack control* runs on your system, it can rewrite your security policy in such a way that all future attacks will work. Of course, it can do anything at all, so this is only one of zillions of attack scenarios. Serious attacks using ActiveX have been seen in the wild (although their use is not widespread). For an explanation of these attacks and more on ActiveX insecurity, see Anup Ghosh's book *E-Commerce Security: Weak Links, Best Defenses* [Ghosh, 1998].

Sandboxes versus Signatures

Do digital signatures make ActiveX more attractive security-wise than Java? No, especially in light of the fact that digital signature capability became available in Java's JDK 1.1 and, in combination with fine-grained access control, plays a major role in Java 2 security. That means in Java, you get everything that ActiveX is doing for security *plus* the ability to run untrusted code fairly safely.

Another significant factor is that the sandbox approach is more robust in the face of accidental bugs in mobile programs. Even if the sandbox isn't bullet-proof, it will most likely prevent a bug in a mobile program from trouncing important data or programs by mistake.

As we shall see in Chapter 3, when combined with access control, code signing allows applets to step outside the security sandbox gradually. In fact, the entire meaning of *sandbox* becomes a bit vague. As an example of how Java code-signing might work, an applet designed for use in an Intranet setting could be allowed to read and write to a *particular* company database as long as it was signed by the system administrator. Such a relaxation of the security model is important for developers who are chomping at the bit for their

applets to do more. Writing code that works within the tight restrictions of the sandbox is a pain, and the original sandbox is very restrictive.

Microsoft's Authenticode and Security Zones

When a signed ActiveX control is downloaded, the browser detaches the signature block (which is a signed one-way hash of the control packaged together with a standard X.509 certificate issued by a certificate authority) and performs checks on the identity of the signer using Authenticode. This is a two-step process. First the certificate is examined by checking the certificate authority's identity. Then the one-way hash is checked to ensure that the same code that was signed was the code that arrived. Note that these checks say nothing at all about whether a control will or will not behave maliciously. They only check the identity of the signer and that the code has not changed since signing.

Microsoft Internet Explorer 4.x implements a *security zone* concept meant to ease the management of security policies for signed content such as ActiveX controls and Java applets. The system organizes Web sites into four "zones of trust" (or more if you customize): Local intranet zone, Trusted sites zone, Internet zone, and Restricted sites zone. Each zone can be configured with security levels of:

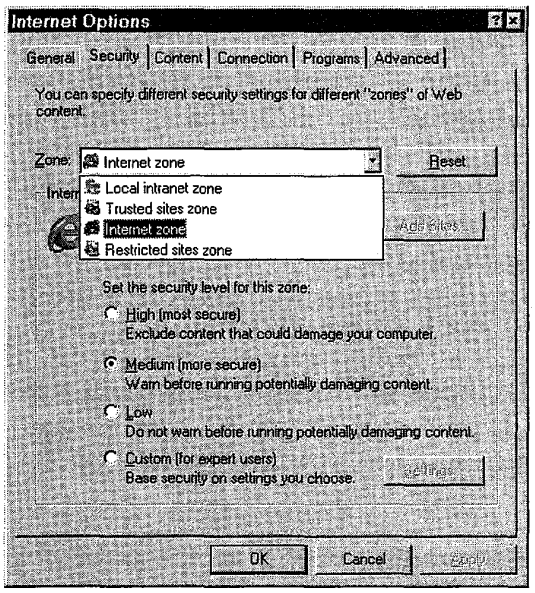


Figure 1.7 Authenticode's signature-based trust model implements the concept of security zones in order to aid in managing mobile code.

Microsoft Internet Explorer provides a dialog box that users can access to manage security zones. Though the importance of powerful policy management tools cannot be overstated, some security professionals complain that allowing a user to set security levels is not a good idea—especially if high security correlates with high level of annoyance (through implementing, for example, too many security queries).

High (most secure), Medium (more secure), Low, or Custom. The idea is to divide Web sites into these zones and assign the zones varying levels of trust.

Figure 1.7 shows a dialog box from Microsoft Internet Explorer (MSIE) that allows a user to manage Authenticode security zones.

Zones are a useful tool that can help make a security policy more coherent. The concept may be particularly useful in non black-and-white policy situations currently beyond the scope of ActiveX. We think security zones are a useful tool that Java security systems beyond Microsoft's should support as well.

In ActiveX with security zones, the security policy itself remains black and white: A mobile program is either fully privileged or completely banned from the system. Since most users are inclined to run cool-sounding code just to check it out regardless of the risk, popping a dialog box in front of a user and requiring an instant security decision is not a good idea. As one of the authors (Felten) is known to say, "Given a choice between dancing pigs and security, users will pick dancing pigs every time." The Princeton team correctly warns that relying completely on a human-judgment-based approach to security in not likely to be as successful as blending judgment with technology-based enforcement, as newer Java systems do. See Figure 1.8.

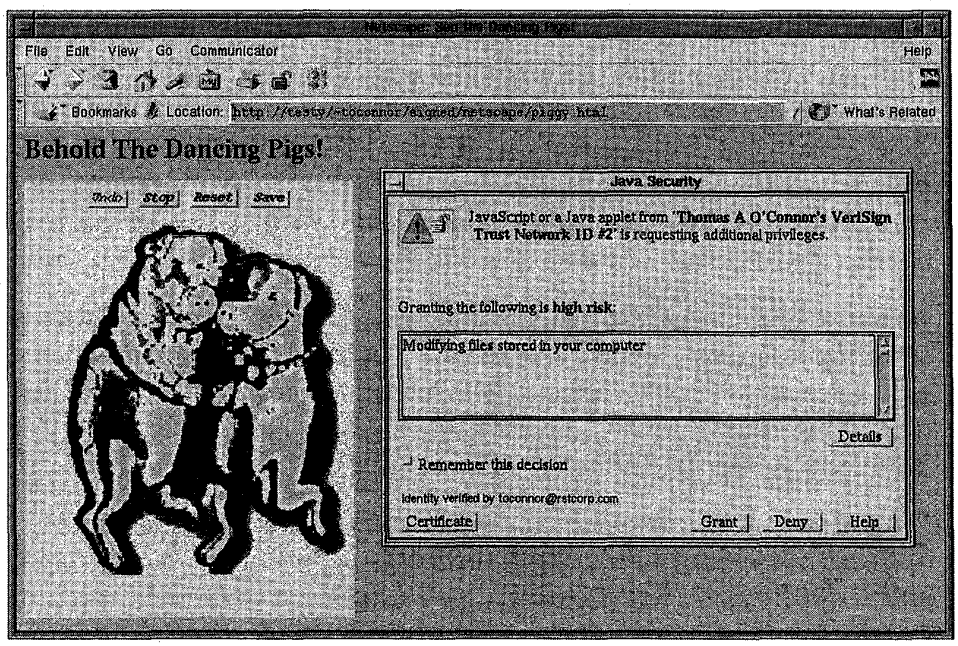


Figure 1.8 "Given the choice between dancing pigs and security, the world will pick dancing pigs every time."

The dancing pigs applet, available through the book's Web site (*www.securingjava.com*), demonstrates the use of digital signatures in Java. See Appendix C.

One way in which Authenticode addresses this problem is to put the security decisions in the hands of a system administrator. Using the MSIE Administration Kit (IEAK), an administrator can preinstall a list of permitted certificates and block the installation of others. This is a step toward centralizing security policy management (which is something most corporate users demand). However, in the end, the ActiveX model is still only a trust model. Just for the record, Netscape now includes a similar site-wide policy administration system.

We discuss these issues of trust, identity, and signatures again in more detail in Chapter 3, though the focus is on Java and not ActiveX.

More on ActiveX Security

The Princeton Team has written an FAQ, reprinted in Appendix A, called *Security Tradeoffs: Java versus ActiveX*, in which a number of common questions about Java and ActiveX are answered. On the Web, the FAQ can be found at *www.cs.princeton.edu/sip/java-vs-activex.html.*

Two other good places to look are in Chapter 2 of *E-Commerce Security* by Anup Ghosh [Ghosh, 1998] and page 18 of *Web Security Sourcebook* by Avi Rubin, Dan Geer, and Marcus Ranum [Rubin, Geer, and Ranum, 1997].

JavaScript

Another mobile code system is JavaScript (Microsoft's version is called JScript). Note that other than the four letters—J, A, V, and A—JavaScript has nothing in common with Java. In the early days, JavaScript was known as *LiveConnect*, but once the marketing folks at Netscape saw the Java wave building, they decided to ride along. JavaScript allows code to be directly contained in HTML documents themselves, code that can dynamically change the HTML that a Web user ultimately sees through a browser.

JavaScript has its own security headaches. Though it is not an ultra-powerful scripting language, JavaScript can easily be used to carry out denial of service and invasion of privacy attacks. Much more discussion about denial of service is found in Chapter 2. JavaScript was used extensively in the Princeton Team's Web Spoofing attack [Felten, et. al., 1997].

To find out more about JavaScript security, a good place to start is at John LoVerso's JavaScript security site: *www.oaf.org/~loverso/javascript/.* On his *JavaScript Problems I've Discovered* page, LoVerso describes JavaScript attacks that:

- *Track a surfer's history*, secretly keeping tabs on all sites visited by a user and reporting back to a collection site
- *Read directory listings*, learning about a Web surfer's file system and reporting back to a collection site
- *Steal files*, mailing the stolen goods back to an attacker
- *Construct Java tags*, circumventing systems that attempt to block Java applets by removing the <APPLET> tag. (For more on why this approach to stopping Java applets is silly, see Chapter 6.)

Make sure that your mobile code security policy (you have one, right?) addresses JavaScript as well as Java.

What Does All This Have to Do with Java?

The important take-home message of this section is that Java security concerns do not exist in a vacuum. If someone tells you that you should disable Java, but pays no attention to these other threats, he or she is not doing you much of a favor. The truth is, much scarier things than Java are out there. In fact, many of the attacks we have touched on here pale in comparison to security concerns raised by a Windows 95 PC connected to the Internet. Try to put all of the security concerns relevant to you on the same scale, and address the biggest risks first.

Where to Find More Information on Java

Java is growing rapidly, and keeping up with it requires as much energy as looking after a herd of two-year-olds (believe us, we know). Keeping up with the edge is just as important for security purposes as it is for any other. Here are some resources that can help make a time investment worth it.

Java on the Web

An excellent place to start learning about Java is the Web itself. The first URL to check is JavaSoft (*java.sun.com*). Also useful are developer.com (*www.developer.com*) and JavaWorld (*www.javaworld.com*). MindQ sells a set of excellent CD-ROMs that provides a multimedia introduction to programming Java applets and applications (among other issues). See its Web page for details at *www.mindq.com*. (MindQ produced the authors' Java Security CD-ROM as well.) To discover some of the many other Java resources on the Net, search for Java at Yahoo! (*www.yahoo.com*) or on AltaVista (*www.altavista.com*). Also see two collections of security-related Java

links put together by the authors at *www.securingjava.com* and at *www.cs.princeton.edu/sip*. The references section of this book includes a complete listing of all URLs cited throughout the book.

Java Books

The number of books on Java is growing almost as fast as the Web itself, and the Java shelf is groaning under their combined weight. For a comprehensive list, see *lightyear.ncsa.uiuc.edu/~srp/java/javabooks.html*. We have had a chance to use a few of them as Java coders. Here are four, with a brief review for each:

Core Java Volume 1—Fundamentals [Horstmann and Cornell, 1997]. This is a good book; big, but definitely useful. In fact, Core Java got so big that it split into two volumes for the JDK 1.1 edition. It is full of comparisons to C++ and Visual Basic, including useful pictures. The authors provide implementations for other classes that are not in the Java libraries, but are commonly used.

Inside the Java Virtual Machine Specification [Venners, 1997]. For anyone interested in the inner workings of Java's Virtual Machine, this is the book to get. As we'll see, Java applet security boils down to what byte code is allowed to do and how its behavior can be constrained. That means that learning about how the VM does its thing is a useful exercise for those people concerned about security.

Java in a Nutshell, second edition [Flanagan, 1997]. This book remains everyone's favorite (well, every developer anyway), probably because it is so useful. O'Reilly is famous for its API books and, true to form, this book provides an extensive API for the packages provided by Java. This makes it excellent for a quick desk-side reference. There are some examples, but if you learn best by examples, you should consider *Java Examples in a Nutshell* [Flanagan, 1997]. Both books are equally useful for beginners and more advanced Java programmers.

Java Network Programming [Hughes, Shoffner, and Winslow, 1997]. One of the best reasons to use Java as a development platform is to take advantage of its built-in networking ability. This excellent book is filled with hands-on examples that are included on a CD-ROM. Of particular interest to security buffs, a number of cryptography algorithms are presented. Note that there is an O'Reilly book of the same title (this one is from Manning); however, this is the one to get.

Java Security Resources

We're glad to say the amount of information available on Java security is also growing. There are both a number of books available and a large number of Web sites. On the Web, we provide the most comprehensive and up-to-date hotlist—the *Java Security Hotlist*—at *www.securingjava.com*. The hotlist, which has over 100 links divided into 9 categories, has been reproduced as Appendix B, "The Java Security Hotlist." Of course URLs are notoriously dynamic, and Java security is a fast-moving field. For the latest version of the hotlist, see the Web site.

The Secure Internet Programming Lab at Princeton also maintains a site with information on security alerts and ongoing Java Security research at *www.cs.princeton.edu/sip/*.

Java Security Books

For at least a year, the first edition of this book, *Java Security: Hostile Applets, Holes, & Antidotes*, was the only available book on Java security. Since that time, a number of other books have come out that address the topic. Of course, we are biased about which one is best, but we thought you might appreciate our opinions about the others anyway:

Java Security: Hostile Applets, Holes, & Antidotes [McGraw and Felten, 1996]. The first book on Java security. This book was intended to educate Web users about the risks of Java security. It includes a discussion of the base Java security model and the original Java security holes. We're glad we wrote it.

Java Network Security [Macgregor, et al., 1998]. This book appeared in 1998 and includes information on JDK 1.1, but nothing on Java 2. The book has a number of technical errors and unintentionally misleading claims about security as well. If you want a copy of everything ever written on Java security, get a copy; otherwise this one is skippable.

Java Security [Oaks, 1998]. O'Reilly is well known for its developer-oriented books. This book fits the bill, as it provides both an API reference guide and a number of code samples. It is almost up to date (the switch to the `doPrivileged()` API discussed in Chapter 3 is not covered by Oaks) and carefully details Java 2 functionality. One caveat: Oaks is an employee of Sun Microsystems and certainly toes the party line. The discussion of security risks and implications reflects this fact. Also missing is any treatment of Java security holes. Nevertheless, if you are a developer who wants to learn about the APIs and you don't care too much about the bigger picture, this book is for you.

The Web Security Sourcebook [Rubin, Geer, and Ranum, 1997]. Although this is not a Java security book *per se*, many of the lessons this book teaches are entirely relevant to people interested in Java security. This is a practical, hands-on book that covers Web server security, mobile code, CGI, and more, written by security experts of the highest caliber.

E-Commerce Security [Ghosh, 1998]. Java is often put to use in e-commerce systems, and of course, e-commerce systems must take security very seriously. This book provides essential data for securing your e-commerce system. It discusses common errors, real attack targets, and solutions.

Mobile Code Has Its Price

Having programs embedded in Web pages that can run on any platform is an excellent idea. But in order to get this power, users take a great deal of risk.

A Web surfer can click over to a Web page with an embedded applet that immediately and automatically begins executing. Often, the user doesn't even know this is happening. This situation might not be so bad if the Java environment being used were 100-percent secure. However, to make Java really secure would require making it completely impotent.[8]

There is a price that must be paid for the power of executable content. This price is very similar to the price that must be paid in order to connect to the Internet in the first place. (In fact, if you decide Java security risks are too much to bear, you should ask yourself what you are doing connected to the Internet at all!) The bill is payable in terms of risk and exposure to attack. The question is, how much risk are you willing to take? How critical is the information on your machine? Our goal in writing this book is to arm the reader with the data that are needed to make an informed, intelligent decision about Java, both as a system for mobile code and as a development platform.

Downloading Mystery Code

How often do you download executable code from various unknown sites on the Net? Do you think about where the code is coming from and who wrote it? Do you know what it will do before you run it?

Even if you are particularly cautious about downloading binaries from the Net, the answers to the questions raised will undoubtedly soon change. Applets are cropping up everywhere. At the moment, surfing the Web with a Java-enabled browser is tantamount to downloading and running arbitrary binaries, albeit with some level of security provided by Java. Deciding whether this is a good idea is an important decision that is as personal as a financial investment strategy.

It is worth repeating that there is no such thing as perfect security. This is true for any system on the Internet, not just systems using Java. Someone will always be probing Java security, trying to find new ways around or through the existing system. In the real world, all you can expect is reasonable security. The solution to this conundrum is finding an acceptable tradeoff between functionality and security.

[8] Keep in mind that the most secure machine is a machine that is kept "off" at all times, has its hard disk wiped, and is buried in a hole filled with concrete. Of course a machine this secure is also useless.

Playing the Cost/Benefit Game

The Internet can be a dangerous playground. Java offers an intriguing approach to the problem of security by neither ignoring it entirely (as most languages do) nor being completely paralyzed by it. Deciding what level of risk to incur is really a matter of weighing the potential costs of using Java against the clear benefits of using Java. Making an informed and intelligent decision requires understanding both aspects of the situation. Business people are always weighing costs and benefits when making complicated decisions. The same sort of careful consideration that goes into forming a business plan should also go into the formulation of a Java use policy.

The Java hype machine has been exceptionally good at broadcasting the benefits of Java. It has been successful largely because Java really does have vast potential. On the other hand, the advertising has been slightly less straightforward about the risks. (To this day we hear claims that Java is 100-percent secure, or that there is no need to worry about Java security.) This may be because the risks are complicated and sometimes difficult to understand. Computer security is a new field to many users, and few people are aware of all the issues. As Java applets become ubiquitous, it behooves us to become more aware of security issues. Ignorance is *not* bliss.

Assessing the Risks

Now that the basics of the Java environment have been covered, you are ready to examine Java security in earnest. It is only after understanding what the security model is, how it works, and how it doesn't, that you can truly begin to assess the security situation.

People should think carefully about using Java even casually with a Java-enabled browser. This book will present some of the facts associated with Java security so that you may decide when, where, and how to use Java. Unfortunately, there is no black-and-white answer to the question: How and when should I use Java?

CHAPTER 3

Beyond the Sandbox: Signed Code and Java 2

Java has outgrown the original restrictive sandbox. The anticipated future of mobile code security, a complex mix of sandboxing and code signing, is now upon us with Java 2. In essence, the three parts of the sandbox explained in the previous chapter implement a language-based security enforcer. This enforcement model has been hybridized and expanded to include fine-grained notions of trust and permission built on digital signatures. That means major changes to Java security. This chapter centers on those changes.

Chapter 1, "Mobile Code and Security: Why Java Security Is Important," briefly introduced the notion of code signing and mobile code policy through the discussion of ActiveX. The ActiveX trust model is suited only to run completely trusted code. At the core of that kind of trust model is a black-and-white decision either to trust the code or not. Such a decision can be influenced by determining who vouches for the code. Digital signatures are used for the vouching.

Java's approach to trust is also based on digital signatures. However, instead of allowing only black-and-white trust decisions à la ActiveX, Java 2 allows fine-grained access control decisions to be made. With the introduction of code signing in JDK 1.1, Java's sandbox model underwent a state transition from a required model applied equally to all Java applets to a malleable system that could be expanded and personalized on an applet-by-applet basis. Java 2 further complicates the picture with the addition of access control.

When combined with access control, code signing allows applets to step outside the security sandbox gradually. In fact, the entire meaning of *sandbox* becomes a bit vague. As an example of how Java code signing might work, an applet designed for use in an Intranet setting could be allowed to read and write to a particular company database as long as it was signed by the system administrator. Such a relaxation of the security model is important for developers who have complained about Java's restrictive sandbox. Writing code that works within the tight restrictions of the sandbox is a pain, and the original sandbox is very restrictive.

The addition of code signing to Java complicates things. As it now stands, the Java sandbox has been reduced to a default. The whole game has changed. Tracing the history of this change as we do in this chapter can lend some important perspective.

Before we dig into the complex issues of code signing and trust models, it does us good to review what it is we're trying to achieve in the first place. After all, the point of all this highfalutin' architecture is not to make the world's most complicated system. The real objective is securing mobile code.

After we remind ourselves of the main goal of the new security model, we are ready to trace its evolution. We will begin by explaining the enhancements added to Java with the release of JDK 1.1, and go on to discuss the Java 2 model in detail.

What's the Main Goal?

Everyone agrees that code signing makes the Java security model a lot more complicated, not to mention actually using the new system. Where security is concerned, complexity is bad since it increases the odds of an error in the system's design or implementation. If we're going to add all of this complexity, what exactly is it that we are gaining? What's the main goal?

The main goal is to gain better control over the security of mobile code. We can achieve this goal by winning the battle on three fronts. By adding code signing and expanding beyond a black-and-white trust model, we hope to gain:

1. The ability to grant privileges when they're needed.
2. The ability to have code operate with the minimum necessary privileges.
3. The ability to closely manage the system's security configuration.

We can judge the JDK 1.1 and Java 2 security models by how well they meet these objectives.

The first objective is simple: We want to give trusted code the privileges it needs to get its job done. A word-processing applet needs the ability to read and write files, so we want to grant this privilege if we have enough faith that the applet won't misbehave. In general, users want to be able to grant any privileges at all to any code they choose, as long as the benefits of doing so outweigh the risks.

The second objective is to have code that operates with the minimum necessary privileges at all times. Security experts call this the "principle of least privilege." This is a common-sense idea—why use a chain saw when a butter knife is sharp enough for the job—but it has profound implications if we carry it to its logical conclusion. One simple implication is that we want programmers to have a way to renounce their privileges when they aren't needed and reenable the privileges when they are needed. The principle of least privilege can be applied in many places throughout the system:

- We want to grant each applet or application the minimum privileges it needs.
- Rather than assigning a given applet's entire collection of privileges to all of its classes, we want each class to get just what it needs.
- We want a class's privileges to be "turned off" except for brief periods of time.
- We even want to reduce the privileges of some of the built-in system classes.

The third objective is manageability. This is a tricky one. Some might think that the ultimate in management power is when all possible options are presented to the user. (Power users, developers, and other gurus tend to think along these lines.) But in reality, users are overwhelmed and irritated when they are confronted with too many big complicated dialog boxes. Somehow the choices must be boiled down so that users get just the control they need without being asked any unnecessary questions.

As we see in the rest of this chapter, the Java security model is still a work in progress. Nobody knows yet how to achieve all of these goals, or even how they trade off against each other. Today's models are pretty good, but they are a far cry from perfect.

Security Enhancements in JDK 1.1

JDK 1.1 appeared in the early Spring of 1997 and included a number of improvements and changes to the base Java security model of JDK 1.0.2. Fortunately, none

of the material about the base Java security model covered in the last chapter (or for that matter, things you learned from the previous edition of this book) was outdated or replaced; rather, the Java security architecture was changed through a process of enhancement and addition.

From a security perspective, the most important changes introduced in JDK 1.1 were the addition of authentication and simple access-control mechanisms that rely on the use of cryptography. Remember, security is much more than just cryptography. Think of cryptography as a means to an end—an important part of the puzzle, but only a part. A side effect of the need for cryptographic functionality inside the model itself was the creation of a crypto API. The crypto API, also introduced with JDK 1.1, provides a basic toolkit of cryptography algorithms that developers can use in their programs.

The Crypto API

Today, Java includes a cryptography toolkit that includes both an API and some packages implementing a portion of the functionality behind the API. Classes in the `java.security` package, the package implementing the cryptographic functionality, have a dual purpose. One purpose is to provide the cryptographic methods that Java's designers used to implement the JDK 1.1 and Java 2 security models. The second purpose is to provide cryptography functionality to Java developers charged with creating secure applications.

Parts of a crypto API were released with JDK 1.1. The parts included both one-way hash functions and digital signature capability. DES encryption tools were released only as an extension to North American users.

Encryption tools and their mathematically related cousins (such as digital signing) change the way Java use policies are managed. Digital signatures, which are discussed next, make it possible to authenticate who has vouched for a piece of code, and potentially check it for tampering. If you decide to trust a particular person, you can set things up so that you automatically trust programs that person signs. (Note that with the right tools, anyone can sign any piece of code. Whether or not a piece of code is written, released, or supported by the person who signed it is not something digital signatures can tell you.) Because the signature is a mechanism for vouching and spreading trust around, if you trust some experts in the field who agree to approve Java programs based on their analysis, you can trust any code that they sign as well. Digital signing paves the way for a true community of trust to develop. We think digital signing is important enough to warrant an entire section itself. See page 88.

Beyond digital signatures, the crypto API released with JDK 1.1 includes a couple of other capabilities. One-way hash functions provide a way to finger-

print a program or data so that you can verify that it has not been changed since being created. Fingerprinting hash functions such as MD5 and SHA make distribution over the Net easier to swallow. If you are certain that a program you are downloading from the Net is the original program (and not a Trojan Horse masquerading as the original), you will probably be more likely to use it. Many archives on the Web today make use of MD5.

Fingerprinting, also called *message digesting*, works by performing a one-way hash over a series of bytes. Given a program (which is really just a bunch of ones and zeros), it is possible to compute a hash that ends up being many times smaller than the original program, but (hopefully) represents only that program. The main trick is to avoid collisions, whereby the same fingerprint is computed for different programs, and to come up with a hash function that can't be run in the opposite direction. MD5 and SHA are systems for computing one-way hashes over a binary file. The crypto API provides a way for Java programs to include this functionality.

MD5 and SHA are useful when it comes to signing code because the act of signing is actually a complicated function of a secret crypto key and the data to be signed. The math is hairy enough that it is a much better idea to compute it using a program's hash instead of the program itself. Remember, the hash is many times smaller than the program it represents. Figure 3.1 shows the important role that one-way hash functions play in code signing.

Another function that appeared as part of the crypto API (at least in the package available only in the United States, and known as the Java Cryptography Extension, or JCE) was DES encryption. DES, an acronym for Digital Encryption Standard, is a venerable old encryption algorithm that can in some cases be deciphered (given enough effort and a small enough key). DES is certainly much more secure than plain text, but does not provide the best available security. In 1998, the EFF created a special-purpose machine to crack DES messages. The purpose of the machine was to emphasize just how vulnerable DES really is. (For more on the DES cracker, see www.eff.org/descracker/.)

Most Unix machines use a variant of DES to encrypt user passwords stored in the `/etc/passwd` file. If 56-bit (or smaller) keys are used for DES, then the U.S. government will allow its export and use outside the United States. There is also a variant called *triple DES* that effectively has a 112-bit key, which will be safe against brute-force searching for a long time. The ease of "breaking" DES is directly related to the length of its key.

Certificates

Another feature that appeared in JDK 1.1 is certificate technology based on the X.509v3 open standard. Certificates provide an authentication mechanism

by which one site can securely recognize another. Sites that recognize each other have an opportunity to trust each other as well. When a secure socket layer (SSL) connection initializes between two machines, they handshake by exchanging certificates. SSL is discussed in the next section.

A certificate is a piece of identification (credential) much like a driver's license. Information stored inside a typical certificate file includes the subject's name, the subject's public key, the certificate's issuer, the issuer's digital signature, an expiration date, and a serial number.

So the question is, who gives out these certificates? Someone (or some thing) called a certification authority (CA). There are a handful of companies that have set themselves up as CAs in the world. These include Netscape, GTE, Verisign, and a few others. But why should you trust them? Good question. (See page 92.)

Secure Communication

Java 2 now includes a package for secure socket layer (SSL) communication. Similar to Netscape's SSL, the Java SSL provides a secure communications channel by using encryption. SSL works by providing a mechanism for encrypting packets on the sending end, sending them over an untrusted channel, and decrypting them at the receiving end. SSL is useful for many business applications, including the transmission of proprietary information and electronic currency.

Most Web servers and browsers now support SSL, allowing a browser to communicate with a Web server without anyone else overhearing the conversation. (Well, an outsider might overhear a conversation, but he or she certainly won't understand it.) Though SSL is commonly used over the Web, it can actually be used to protect virtually any sort of network transaction.

Most browsers support SSL by providing a "Secure HTTP Connection" service that looks to the user just like a normal Web connection, but uses SSL underneath. This allows you to reap the benefits of SSL without having to learn anything except how the browser tells you whether a connection is secure.

The encryption technology underlying SSL is generally believed to be secure, but there are two potential problems. First, the U.S. government restricts the export of strong cryptography software. If your browser version includes dumbed-down exportable cryptography software, your communications might not be as secure as you think. Second, SSL is good at providing secure communications, but it is not as good at establishing who you are communicating with. This leads into all the problems of authentication and key distribution discussed on page 90.

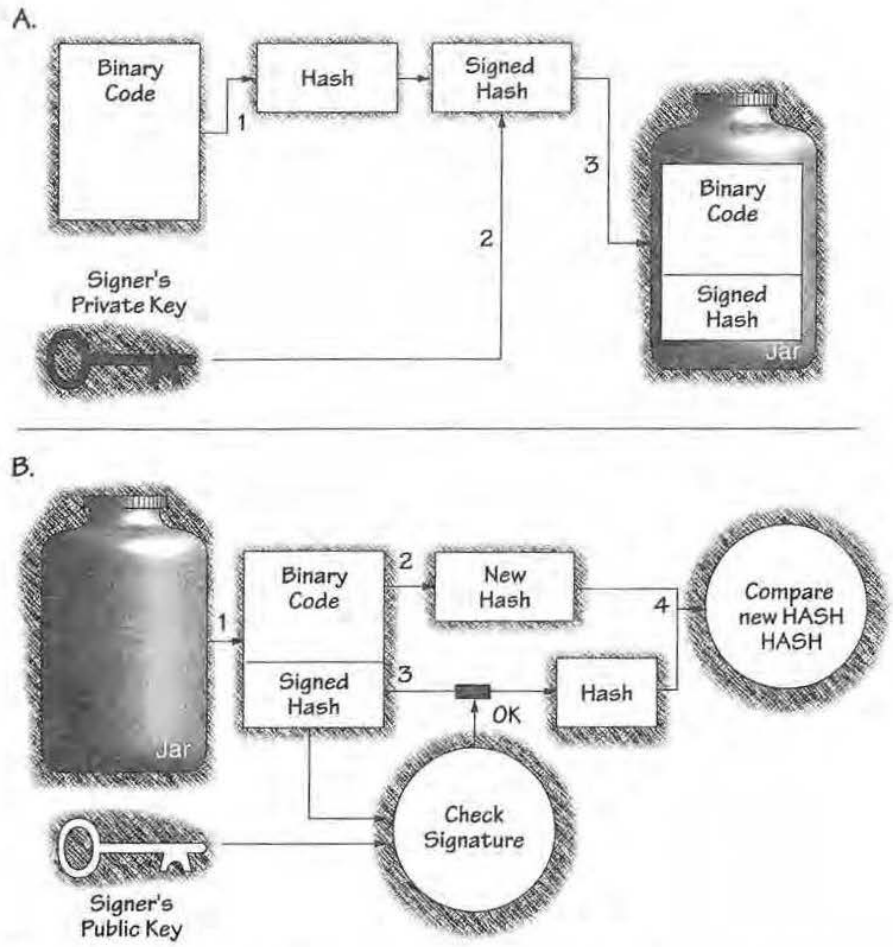


Figure 3.1 How code is digitally signed (A) and digital signatures are verified (B).

(A) Signing code takes several distinct operations: (1) a one-way hash calculation is run on a piece of binary code, resulting in a small "thumbprint" of the code; (2) the hash is signed using the signer's private key; (3) the signed hash and the original binary code are placed together (potentially along with other signed and unsigned code) in an archive JAR. Now the JAR can be shipped around as mobile code.

(B) Validating signed code also takes several steps: (1) a piece of binary code and its associated signed hash are removed from the JAR; (2) a new hash is calculated using the same one-way hash algorithm that the signer used to create the signed hash; (3) the signature carried by the signed hash is cryptographically validated with the signer's public key (possibly with reference to certificate authorities and trust chains); (4) if the signature checks out, the now decrypted original hash is available for comparison with the new hash. Though all three Java code signing schemes (Sun, Microsoft, and Netscape) share these two processes, there are enough differences that the systems do not inter-operate. See Appendix C for examples of how to sign Java code under each implementation.

Signed Code

The capability to digitally sign Java byte code (at least byte code files placed in a Java archive, called a JAR file) was introduced with JDK 1.1 and greatly expanded with Java 2. Digital signing capability is an important part of the new Java security regimen. This is exciting because digital signing radically alters the amount of trust you can place in a piece of code. A Tutorial on signing Java code with the current tools from Microsoft, Netscape, and Sun can be found in Appendix C.

One particular kind of cryptography tool allows a chunk of digital information (including, of course, Java byte code) to be signed by a person or organization. See Figure 3.1. Because a digital signature has special mathematical properties, it is difficult to forge. Your browser can verify a signature, allowing you to be fairly certain that a particular person or organization vouches for the code. That means you can instruct your browser always to accept applets signed by some party that you trust, or always to reject applets signed by some party that you don't trust. Same thing goes for a non-browser–based VM, which can be instructed (through policy) how to treat application code signed by particular entities.

It is important to recognize that even if you know exactly which Web pages you are visiting and who created them, you probably don't know who wrote each applet that appears on the pages you visit. Applets are shuffled around on the Net like Beanie Babies in a fifth-grade classroom.

Contrary to popular belief, you don't always know where information is coming from on the Internet. A nasty attack called *IP spoofing* allows a bad guy to send you network traffic that claims to come from someplace else. For instance, you might think the traffic is coming from "whitehouse.gov", when it's really coming from "cracker.org". IP spoofing used to be considered just a theoretical possibility, but it has actually happened in recent years. The best-known example is an attack by the infamous cracker Kevin Mitnick on a machine managed by computer security worker Tsutomu Shimomura. Mitnick's attack led to his eventual capture and conviction [Shimomura and Markoff, 1996].

An attack known as *Web spoofing* shows that even in the absence of IP spoofing, it is not always clear that you are visiting the site you may think you're visiting [Felten, et al., 1997]. An attacker can lure you into a "false Web" that looks just like the real one, except that the attacker can see everything you do, including anything you type into a form, and the attacker can modify the traffic between you and any Web server. All of this is possible even if your browser tells you that you have a "secure" connection. See Figure 3.2.

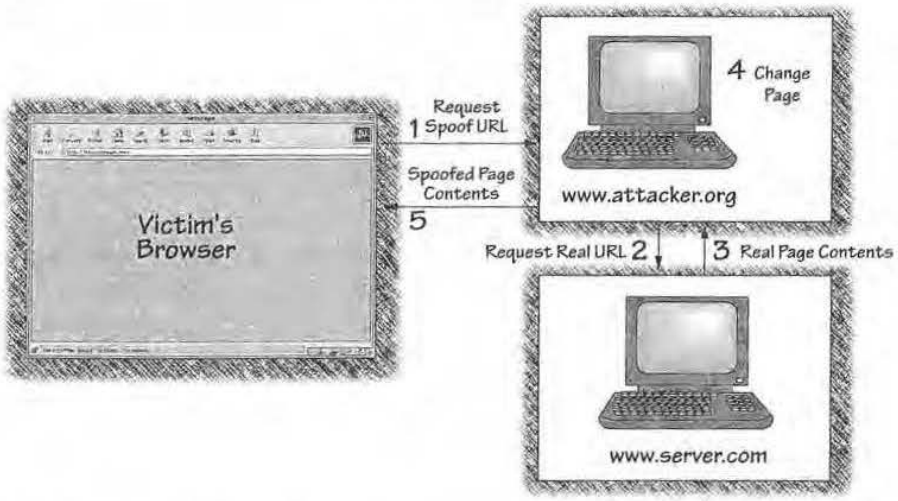


Figure 3.2 A Web Spoofing attack can be carried out with extensive use of a browser's mobile code capability.

The Princeton Team has implemented a demonstration of Web Spoofing that makes extensive use of JavaScript. Once an attacker has lured the victim to the attack server (shown as *www.attacker.org*), the attacker can control the victim's view of the Web by acting as a rewriting proxy. Clever use of JavaScript makes all changes invisible to the victim and can even appear to offer encrypted traffic.

Even if you ignore the possibility of spoofing, using the return address of an applet (that is, knowing the Web site where you got the applet code) still isn't good enough to base a trust decision on. A digital signature holds much more information. For example, such a signature could tell you that although the applet is being redistributed by a site you don't trust, it was originally signed by someone you do trust. Or it can tell you that although the applet was written and distributed by someone you don't know, your friend has signed the applet, attesting that it is safe. Or perhaps it can simply tell you which of the thousands of users at aol.com signed the applet.

Digital Signatures

So how do you sign a piece of code? The key to certification and authentication is the use of digital signatures. The idea is simple: to provide a way for people to sign electronic documents so that these signatures can be used in the same way we use signatures on paper documents. In order to be useful, a digital signature should satisfy five properties [Schneier, 1995]. It should be:

1. **Verifiable:** Anyone should be able to validate a signature.
2. **Unforgeable:** It should be impossible for anyone but you to attach your signature to a document.

3. **Nonreusable:** It should be impossible to "lift" a signature off one document and attach it to another.
4. **Unalterable:** It should be impossible for anyone to change the document after it has been signed, without making the signature invalid.
5. **Nondeniable:** It should be impossible for the signer to disavow the signature once it is created.

Mathematicians and computer scientists have devised several digital signature schemes that appear to work quite well. The full details are very technical. If you're interested in learning more about such schemes, Bruce Schneier's excellent book, *Applied Cryptography*, is a good place to start [Schneier, 1995].

The digital signatures used for Java code are based on public-key cryptography. If Alice wants to be able to sign documents, she must first use a special mathematical technique to generate two large numbers: her own private key, and her public key. As the names suggest, Alice keeps her private key to herself. Keeping it secret is essential. Her public key, however, is announced to the world.

Alice's private key is used for signing electronic documents. Her public key is used to verify those signatures. See Figure 3.1. Anyone who knows the private key (hopefully only Alice!) can run a special computation involving the document and Alice's private key. The result of this process is a digitally signed version of the document.

Anyone who knows Alice's public key can verify her signature by running a special computation involving the signed document and Alice's public key. Since only Alice knows the private key, she is the only one who can put her signature on documents. Since everyone knows her public key, anyone can verify that the signature is hers.

Everything sounds great. You tell your browser to trust applets signed by Alice by registering Alice's public key. Whenever applets claim to come from Alice, the browser can verify that claim by comparing the registered public key to the signed applet. If the applet is not from Alice, it can be rejected.

Key Distribution

But how do you know what Alice's public key is?

If you know Alice, she can call you on the phone and tell you her public key. In this case, you will know the key is valid because you recognize Alice's voice. This doesn't work if you don't know Alice. How do you know the person on the other end of the phone is Alice? Maybe it's Alice's evil twin Zelda, trying to pass off Zelda's public key as Alice's so she can forge Alice's signature.

One way around this problem is to ask Alice's twin brother Allan to help. Alice can create a document containing her public key and have Allan sign that document. If you trust Allan and you know Allan's public key, then the document tells you reliably what Alice's public key is.

But how do you know Allan's public key? You can't ask Alice and Allan to vouch for each other's public keys, because Zelda could create a false Alice key and a false Allan key and use them to sign documents vouching for each other! This leaves us stuck with a chicken-and-egg problem.

The usual solution is to use a certification authority (CA). The CA, Claire in our example, is in the business of certifying keys. Alice goes to the CA's office with her birth certificate, passport, driver's license, and DNA sample. Once she has convinced Claire that she really is Alice, she tells Claire her public key, and Claire signs an electronic document that contains Alice's public key. That document serves as an electronic credential for Alice.

After Alice has a credential, key distribution is much easier. Alice can plaster copies of her credential everywhere: on bulletin boards, on her homepage, and at the end of every email message she sends. Better yet, whenever Alice signs a document, she can attach a copy of her credential to the signed document. On receiving the document, you can first check the credential by verifying Claire's signature, and then verify Alice's signature using the public key included with the document. Zelda can't trick you into accepting a bogus public key for Alice, because she can't forge Claire's signature. Figure 3.3 shows the process by which a signature on a piece of signed code can be validated.

The beauty of this approach is that if everyone can visit Claire and get a credential, then no one has to remember any keys except for his or her own private key

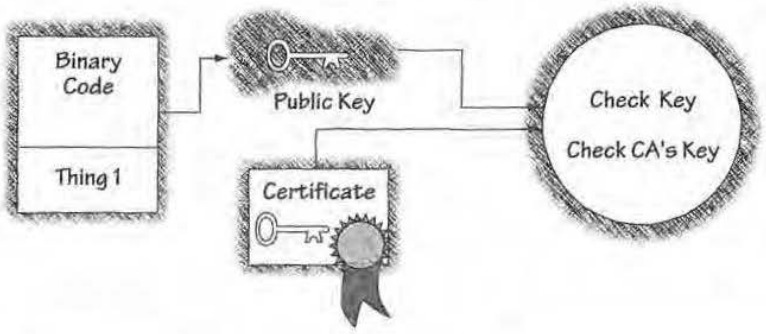


Figure 3.3 Validating a signature on signed code.

In this example, a piece of code is signed by the private key of thing1. The corresponding public key, available on thing1's certificate can be used to validate the signature carried by the code. For added security and to make key management more reasonable, browsers typically validate the CA signature carried on the certificate.

(to sign documents), and Claire's public key (to verify credentials). There are still two problems, though. Everyone must trust Claire. As the authority, she can impersonate anyone. And you still need a reliable way to get Claire's public key. It doesn't help to have Claire get a credential from Claire's mom, Elena. You would have no more reliable way of knowing who Elena is.

There is no technological solution to this. Claire's key will probably be hard-wired into your browser software, or entered by support staff at install time. As long as you get a valid copy of the browser, and no one has messed with your hard disk, everything will be okay. How do you know you have a valid copy of the browser? It will be signed by the browser vendor. How do you know the browser vendor's signature is valid? Don't ask—there lies madness.

What Signing Can't Do

Even if the signing and signature-checking mechanisms work perfectly and are able to reveal who signed each applet, a huge unsolved problem still remains. Technology can tell you who signed an applet, but it can't tell you whether that person is trustworthy. That's a decision you have to make based on human judgment. And you'd better make the right decision.

Trust

Once a code signing infrastructure is in place, you will be able to know reliably who vouches for each Java program. The next link in the chain is figuring out what to do with that knowledge.

One thing you can certainly do is to relax Java's security rules for applets that you trust. For example, with the default sandbox Java normally prohibits any access to files in order to prevent an applet from corrupting your hard drive or reading your private data. If you trust applets from particular sources, though, you might want to allow them to read files. Introducing permissions granted according to trust level opens up vast new application areas, including things like spreadsheet applets, games with stored high scores, Web sites that recall your preferences, a host of different remote management possibilities, and so on.

Besides access to files, there are many other capabilities you might want to grant a trusted applet or application: access to your machine's microphone and camera, freedom to make network connections, and maybe even freedom to label other code as trusted. It all depends on your decision to trust and how much to trust a signed program. There are several ways you can make these decisions.

Who Do You Trust?

The first decision is whether to use a black-and-white or a shades-of-gray policy. A black-and-white policy is one that divides all programs into two groups: trusted and untrusted. This was the only sort of trust policy that was easy to implement using JDK 1.1. Java 2, however, changed all that. Java 2 makes it possible to create a shades-of-gray policy, allowing you to assign any degree of partial trust to a Java program. (Recall Figure 1.6 in Chapter 1.)

Before Java came along, most Internet software worked on a black-and-white model. If someone offered to let you download a program, you had two choices: either you downloaded the program or you didn't. If you did, you were trusting the program completely since there was nothing to stop it from running wild on your machine. If you didn't download the program, you were treating it as completely untrusted. Java, with its security policies as implemented in the base sandbox, changed the rules a bit by making it easier to decide what to download in the first place. If an untrusted applet can't bite you, you might as well check it out.

The black-and-white model is sometimes called the *shrink-wrap model* because it's similar to software you purchase. If you buy a software package from a reputable software store, you can reasonably assume that the software is safe to load on to your machine. People who use the term *shrink-wrap model* tend to assume that no one would ever want to run software that wasn't written by a large software company. We don't agree with that implication, so we'll stick with the term *black-and-white*.

It might seem that the shades-of-gray model is better than the black-and-white model, because black-and-white only allows you to label programs as completely trusted or completely untrusted. On the other hand, shades-of-gray gives you more choices. You may still label an applet as completely trusted or completely untrusted if you wish.

Choices are not always good, as anyone who has encountered the cereal aisle of a large supermarket can attest. Making choices takes up time that you would probably rather spend doing something else. Frequent decision-making saps your attention span, so you are more likely to make a mistake, thus opening yourself up to attack. Finally, having more options saddles your browser with more complicated record-keeping duties to keep track of all of your decisions. This extra complexity might lead to bugs in the browser, possibly jeopardizing security yet again.

Which model is better, black-and-white or shades-of-gray? It depends on how people react to the two systems, which is hard to predict. Mostly likely, competing browsers will offer different models, and the models will fight it out in the marketplace. The decision is ultimately one of user preference.

Free the Trusted Code!

Once you've decided who to trust, the next issue is what you allow trusted programs to do. If you're using the black-and-white model, then you have to decide whether to allow untrusted programs, like applets off unknown Web sites, to run at all. You also have to decide what extra capabilities, if any, you want to give to trusted programs. You might decide to let trusted programs do whatever they want, with no restrictions at all. Or you might decide to run trusted programs under the restrictive Java security rules of JDK 1.0.2. The choices depend on your taste for risk, and what kinds of programs you want to run. With black-and-white security, however, all the programs you trust receive the same level of trust.

If you're using a shades-of-gray model, you face more choices. You may decide on a program-by-program (or signer-by-signer) basis exactly which capabilities to grant. Rather than presenting you with a huge laundry list of possible capabilities for each program and forcing you to tick items off the list, a good browser will probably provide a way for you to grant certain prepackaged sets of capabilities. For example, there might be a set of permissions for videoconferencing applets, which would include things like permission to use the camera, the microphone, the speaker, the display, and networking access. Perhaps there would be another set of document-editing applet permissions, which would include file-creation, file-reading, and file-modification capabilities.

There are two basic ways to group the mapping of program (or programs) to permission (or permissions). Microsoft's Authenticode system, introduced in Chapter 1, defines *security zones*, which are ways of grouping programs together. For example, all programs from a company intranet signed by the system administrator's key might comprise a zone. (These zones might well involve multiple keys and origins.) Policies can then be defined on a per-zone basis. Netscape defines *macro targets*, which are groups of permissions (as sketched in the previous paragraph). For example, a macro target might be called "typical game privileges" and define the permissions typically needed by a network-enabled game.

Sun has a system of implication in which permission for code to use one resource can imply permission to use another resource. In their model, each resource is required to define an `implies()` method that can be used to ask a resource whether it implies a particular other permission. (More detail is provided later in this chapter.)

All of these are examples of grouping signers or privileges together and treating the group as a unit. Grouping is generally a good idea in security management because it reduces the number of decisions that the user (or other policy-maker) faces. Fewer decisions means more attention paid to each decision and hence, better decisions.

JDK 1.1, which introduced the concept of a signed applet, provides a black-and-white model. A digitally signed applet can be treated as trusted local code as long as the signature key is recognized as trusted by the system finally running the code. Java 2 provides a shades-of-gray model.

An Introduction to Java 2 Security

Signatures alone don't provide the infrastructure needed to allow Java code out of the sandbox gradually. Access control mechanisms are required as well. In JDK 1.1, for example, applet code signed by a trusted party can be treated as trusted local code, but not as partially trusted code (without an inordinate amount of extra programming). There is no notion of access control beyond the one-and-only trust decision made per class. That means in practice, JDK 1.1 offers a black-and-white trust model much like ActiveX (although with the clear advantage that untrusted code must stay in the sandbox).

The new security architecture in Java 2 has four central capabilities [Gong and Schemers, 1998]:

Fine-grained access control: The ability to specify that code with proper permissions is allowed to step outside the sandbox constraints gradually (for example, an applet signed by a trusted key might be allowed to open arbitrary network connections).

Configurable security policy: The ability for application builders and Java users to configure and manage complex security policies.

Extensible access control structure: The ability to allow typed permissions and to group such permissions in logical, policy-oriented constructs.

Security checks for all Java programs: A departure from the concept that built-in code should be completely trusted. (It is this capability that serves to erase the once-important distinction between applets and applications.)

It is important to note that the first three of these four capabilities are not really new to Java. Java is a powerful programming language, and it has always been possible to implement complex, configurable, extensible security policies based on fine-grained access control. It was just exceptionally tricky. Java 2 serves to make this task possible for mere mortals.

A View from 50,000 Feet

At its heart, the Java 2 security model has a simple idea: Make all code run under a security policy that grants different amounts of privilege to different programs. While the idea may be simple, in practice, creating a coherent policy

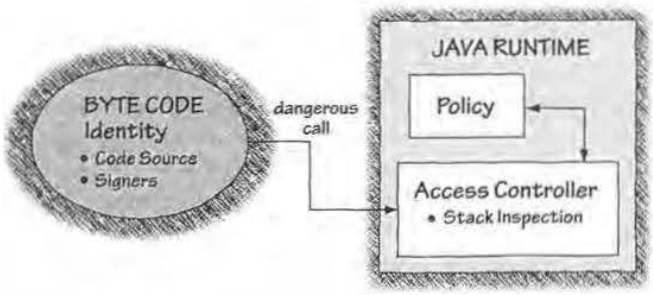


Figure 3.4 Mobile code in Java 2 interacts with user defined policy through the `AccessController`.

Byte code may make calls to potentially-dangerous functionality. When such calls are made, the `AccessController` (new to Java 2) consults policy and uses stack inspection to decide whether to allow or disallow a call. Decisions are based on the identity of the code.

is quite difficult. Figure 3.4 shows the role that mobile code identity and policy play in Java 2.

Java 2 code running on the new Java VMs can be granted special permissions and have its access checked against policy as it runs. The cornerstone of the system is *policy* (something that will not surprise security practitioners in the least). Policy can be set by the user (usually a bad idea) or by the system administrator, and is represented in the class `java.security.Policy`. Herein rests the Achilles' Heel of Java 2 security. Setting up a coherent policy at a fine-grained level takes experience and security expertise. Today's harried system administrators are not likely to enjoy this added responsibility. On the other hand, if policy management is left up to users, mistakes are bound to be made. Users have a tendency to prefer "cool" to "secure." (Recall the dancing pigs of Chapter 1.)

Executable code is categorized based on its URL of origin and the private keys used to sign the code. The security policy maps a set of access permissions to code characterized by particular origin/signature information. Protection domains can be created on demand and are tied to code with particular `CodeBase` and `SignedBy` properties. If this paragraph confuses you, imagine trying to create and manage a coherent mobile code security policy!

Code can be signed with multiple keys and can potentially match multiple policy entries. In this case, permissions are granted in an additive fashion.

A Simple Example

An easy example of how this works in practice is helpful. First, imagine a policy representing the statement "code from "www.rstcorp.com/" applet signed by

'self' is given permission to read and write files in the directory `/applet/tmp` and connect to any host in the `rstcorp.com` domain." Next, a class that is signed by "self" and that originates from "www.rstcorp.com/" applet arrives. As the code runs, access control decisions are made based on the permissions defined in the policy. The permissions are stored in permission objects tracked by the Java runtime system. Technically, access control decisions are made with reference to the runtime call stack associated with a thread of computation (more on this later).

Access Control and Stack Inspection

The idea of access control is not a new one in computer security. For decades, researchers have built on the fundamental concept of grouping and permissions. The idea is to define a logical system in which entities known as *principals* (often corresponding one to one with code owned by users or groups of users) are authorized to access a number of particular protected objects (often system resources such as files). To make this less esoteric, consider that the familiar JDK 1.0.2 Java sandbox is a primitive kind of access control. In the default case, applets (which serve as principals in our example) are allowed to access all objects inside the sandbox, but none outside the sandbox.

So what we're talking about here is a way of setting up logical groupings. Then we can start talking about separating groups from each other and granting groups particular permissions. Security is all about separation. Readers familiar with the Unix or NT file system will see clear similarities to the notion of user IDs and file permissions.

Sometimes a Java application (like, say, a Web browser) needs to run untrusted code within itself. In this case, Java system libraries need some way of distinguishing between calls originating in untrusted code and calls originating from the trusted application itself. Clearly, the calls originating in untrusted code need to be restricted to prevent hostile activities. By contrast, calls originating in the application itself should be allowed to proceed (as long as they follow any security rules that the operating system mandates). The question is, how can we implement a system that does this?

Java implements such a system by allowing security-checking code to examine the runtime stack for frames executing untrusted code. Each thread of execution has its own runtime stack (see Figure 3.5). Security decisions can be made with reference to this check. This is called *stack inspection* [Wallach, et al., 1997]. All the major vendors have adopted stack inspection to meet the demand for more flexible security policies than those originally allowed under the old sandbox model. Stack inspection is used by Netscape Navigator 4.0, Microsoft Internet Explorer 4.0, and Sun Microsystems' Java 2.

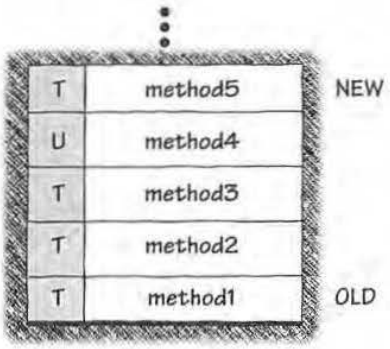


Figure 3.5 Each Java program thread includes a runtime stack that tracks method calls.

The purpose of the stack is to keep track of which method calls which other method in order to be able to return to the appropriate program location when an invoked method has finished its work. The stack grows and shrinks during typical program operation. Java 2 inspects the stack in order to make access control decisions. In this example, each stack frame includes both a method call and a trust label (T for trusted, U for untrusted).

(Interestingly, Java is thus the most widespread use of stack inspection for security ever. You can think of it as a very big security-critical experiment.)

Simple Stack Inspection

Netscape 3.0's stack-inspection–based model (and every other black-and-white security model) is a simple access control system with two principals: *system* and *untrusted*. Just to keep things simple, the only privilege available is *full*.

In this model, every stack frame is labeled with a principal (*system* if the frame is executing code that is part of the VM or the built-in libraries and *untrusted* otherwise). Each stack frame also includes a flag that specifies whether privilege is *full*. A system class can set this flag, thus enabling its privilege. This need only be done when something dangerous must occur—something that not every piece of code should be allowed to do. Untrusted code is not allowed to set the flag. Whenever a stack frame completes its work, its flag (if it has one) disappears.

Every method about to do something potentially dangerous is forced to submit to a stack inspection. The stack inspection is used to decide whether the dangerous activity should be allowed. The stack inspection algorithm searches the frames on the caller's stack in sequence from the newest to the oldest. If the search encounters an untrusted stack frame (which as we know can never get a privilege flag) the search terminates, access is forbidden, and an exception is thrown. The search also terminates if a system stack frame with a privilege flag is encountered. In this case, access is allowed (see Figure 3.6).

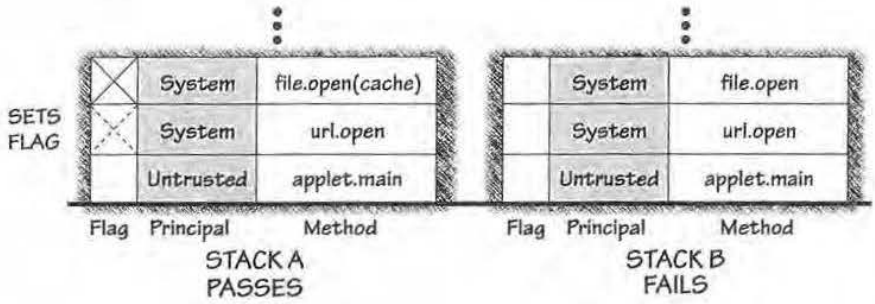


Figure 3.6 Two examples of simple stack inspection.

Each stack is made of frames with three parts: a privilege flag (where full privilege is denoted by an X), a principal entry (untrusted or system), and a method. In STACK A, an untrusted applet is attempting to use the `url.open()` method to access a file in the browser's cache. The VM makes a decision regarding whether to set the privilege flag (which it does) by looking at the parameters in the actual method invocation. Since the file in this case is a cache file, access is allowed. In short, a system-level method is doing something potentially-dangerous on the behalf of untrusted code. In STACK B, an untrusted applet is also attempting to use the `url.open()` method, however in this case, the file argument is not a browser cache file but a normal file in the filesystem. Untrusted code is not allowed to do this, so the privilege flag is not set by the VM and access is denied.

Real Stack Inspection

The simple example of stack inspection just given is only powerful enough to implement black-and-white trust models. Code is either fully trusted (and granted full permission at the same level as the application) or untrusted (and allowed no permission to carry out dangerous operations). However, what we want is the ability to create a shades-of-gray trust model. How can we do that?

It turns out that if we generalize the simple model we get what we need. The first step is to add the ability to have multiple principals. Then we need to have many more specific permissions than *full*. These two capabilities allow us to have a complex system in which different principals can have different degrees of permission in (and hence, access to) the system.

Research into stack inspection shows that four basic primitives are all that are required to implement a real stack inspection system. In particular, see Dan Wallach's Ph.D. thesis at Princeton and the paper *Understanding Java Stack Inspection* [Wallach and Felten, 1998]. Each of the major vendors uses different names for these primitives, but they all boil down to the same four essential operations (all explained more fully in the following discussions):

enablePrivilege()

disablePrivilege()

checkPrivilege()

revertPrivilege()

Some resources such as the file system or network sockets need to be protected from use (and possible abuse) by untrusted code. These resources are protected by permissions. Before code (trusted or otherwise) is allowed access to one of these resources, say, `R`, the system must make sure to call `checkPrivilege(R)`.

If you recall our discussion of the Security Manager from the previous chapter, you'll remember that the Java libraries are set up in such a way that dangerous operations must go through a Security Manager check before they can occur. As we said, the Java API provides all calls necessary to implement a virtual OS, thus making isolation of all required security checks possible within the API. When a dangerous call is made to the Java API, the Security Manager is queried by the code defining the base classes. The `checkPrivilege()` method is used to help make behind-the-scenes access control decisions in a very similar fashion. To achieve backwards compatibility, the Security Manager can be implemented using the four stack inspection primitives.

When code wants to make use of some resource `R`, it must first call `enablePrivilege(R)`. When this method is invoked, a check of local policy occurs that determines whether the caller is permitted to use `R`. If the use is permitted, the current stack frame is annotated with an enabled-privilege(R) mark. This allows the code to use the resource normally.

Permission to use the resource does not last forever; if it did, the system would not work. There are two ways in which the privilege annotation is discarded. One way is for the call to return. In this case, the annotation is discarded along with the stack frame. The second way is for the code to make an explicit call to `revertPrivilege(R)` or `disablePrivilege(R)`. The latter call creates a stack annotation that can hide an earlier enabled privilege. The former simply removes annotations from the current stack frame.

All three major Java vendors implement a very similar (and simple) stack inspection algorithm. A generalization of this algorithm, after Wallach, is shown in Listing 3.1 [Wallach and Felten, 1998].

The algorithm searches stack frames on the caller's stack in order from newest to oldest. If the search finds a stack frame with the appropriate enabled-privilege annotation, it terminates, allowing access. If the search finds a stack frame that is forbidden from accessing the target by local policy, or has explicitly disabled its privileges, the search terminates, forbidding access.

It may seem strange that the vendors take different actions when the search reaches the end of the stack without meeting any of the conditions (sometimes called *falling off* the end of the stack). Netscape denies permission, while both Microsoft and Sun allow permission. This difference has to do with backward compatibility. The Netscape choice causes legacy code to be treated like an old-fashioned applet, and confined to the sandbox. The Microsoft/Sun choice

Listing 3.1 An algorithm for stack inspection.

```
checkPrivilege(target){
     // loop, newest to oldest stack frame
    foreach stackFrame{
          if (local policy forbids access to target
             by class executing in stackFrame)
                throw ForbiddenException;
          if (stackFrame has enabled privilege for target)
                return; // allow access
          if (stackFrame has disabled privilege for target)
                throw ForbiddenException;
     }
// if we reach here, we have fallen off the end of the stack
if (Netscape 4.0)
          throw ForbiddenException;
if (Microsoft IE 4.0 || Sun JDK 1.2)
          return; // allow access)
```

allows a signed Java application to use its privileges even without explicitly marking its stack frames, thus making it easy to migrate existing applications. Since Netscape did not support applications, they felt no need to follow the Microsoft/Sun approach and instead chose the more conservative course of denying permission. For more implementation detail on the three vendors' different code signing schemes, see Appendix C.

Formalizing Stack Inspection

Members of Princeton's Secure Internet Programming team (in particular, Dan Wallach and Edward Felten) have created a formal model of Java's stack inspection system in a belief logic known as ABPL (designed by Abadi, Burrows, Lampson, and Plotkin) [Abadi, et al., 1993]. Using the model, the Princeton team demonstrates how Java's access control decisions correspond to proving statements in ABPL. Besides putting Java's stack inspection system on solid theoretical footing, the work demonstrates a very efficient way to implement stack inspection systems as pushdown automata using security-passing style. Interested readers should see [Wallach and Felten, 1998], which is available through the Princeton Web site at *cs.princeton.edu/sip/pub/oakland98.html*. A more recent paper on how to implement stack inspection more efficiently is also available on the Princeton site.

New Security Mechanisms in Sun's Java 2

Now that we have covered the basic concepts and the underlying mechanisms of Java 2 security, we can delve into the details of the system. Essential

mechanisms include many of the things we have already discussed: identity, permissions, `implies`, policy, protection domains, access control, and privilege. Sources for the information presented here include [Gong, et. al., 1997; Gong and Schemers, 1998].

This section describes Sun's version of stack inspection. Netscape and Microsoft each have their own version, but we decided to forgo a lengthy discussion of all three systems. Though the vendors claim they are very different, we think the three systems are really quite similar. Perhaps one day they will all converge, making developers' and managers' lives much easier.

Identity

Every piece of code needs a specific identity that serves as a basis for security decisions. In Java 2, each piece of code has two identity-defining characteristics: *origin* and *signature*. These two characteristics are represented in the class `java.security.CodeSource`, which allows the use of wildcard entries to denote "anywhere" for origin and "unsigned" for signature.

Origin boils down to the location the code came from specified as a URL. This is the same sort of identity used in separation of applets in the JDK 1.0.2 class loading scheme. In fact, Java 2 identity is really an extension of that idea.

Signature is a bit more complicated. Remember, public/private keys come in pairs. As we know, code can be digitally signed by a person or organization who vouches for it. The key used to actually sign the code is the signer's private key. The key used to check the signature for validity is the signer's public key. So, the public key corresponding to the private key used to sign a piece of code is the second identity characteristic. (In practice, implementations actually use an alias for the public key corresponding to the private key used to sign the code.)

Many people say that a signature on code tells you "who wrote the code" or "where the code came from" (we've been guilty of this faux pas ourselves in days gone by), but this is not true. All a signature tells you is who signed the code. The author, distributor, and signer of the code may all be different parties. All you know for sure is that the signer vouches for the code. And since it makes perfect sense for several people to vouch for the same piece of code, a good signature scheme ought to allow a piece of code to carry several signatures; then each recipient can decide which of the signers (if any) should be trusted.

Permissions

Requests to perform a particular operation (most notably a dangerous one) can be encapsulated as a permission. A *policy* says which permissions are granted to

which principals. The abstract class `java.security.Permission` types and parameterizes a set of access permissions granted to classes. Permissions can be subclassed from this class (and its subclasses). Good practice dictates that a permission class should belong to the package in which it is used. Java 2 defines access methods and parameters for many of the resources controlled by the VM.

Permissions include:

`java.io.FilePermission` for file system access

`java.net.SocketPermission` for network access

`java.lang.PropertyPermission` for Java properties

`java.lang.RuntimePermission` for access to runtime system resources

`java.security.NetPermission` for authentication

`java.awt.AWTPermission` for access to graphical resources such as windows

Permissions usually include a *target* and an *action*. For file access, a target can be a file or a directory specified as `file`, `directory`, `directory/file`, `directory/*`, or `directory/-`. The `*` denotes all files in the specified directory. The `-` denotes all files under the associated file system subtree (meaning all by itself, `-` denotes all files in the entire system). Actions for file access include `read`, `write`, `execute`, and `delete`.

An example of a file permission is:

```
p = new FilePermission("/applets/tmp/scratch", "read");
```

For network access, a target can be an IP address, hostname, or generalized set of hostnames and a range of port numbers. The target argument takes the form "hostname:port-range". Actions for network access include: `connect`, `listen`, and `accept`. An example of a socket permission is:

```
p = new SocketPermission("bigbrother.rstcorp.com:-1023",
"connect");
```

For getting and setting properties, a target is the property (where `*` denotes all properties). Actions are `get` and `set`. Runtime system resource targets include `createClassLoader`, `exit`, `setFactory`, `thread`, `multicast`, `fileDescriptor.read`, `fileDescriptor.write`, and so on. AWT permission targets include `topLevelWindow`, `systemClipboard`, and `eventQueue`.

Fully trusted Java applications can add new categories of permissions.

Implies

Each `Permission` class must include the abstract method `implies`. The idea is straightforward: having permission `x` automatically implies having permission `y`. We denote this `x.implies(y) == true` in code. A permission `x` implies another permission `y` if and only if both the target of `x` implies the target of `y` and the action of `x` implies the action of `y`.

Consider the permission "read file /applets/tmp/scratch," which can be written as:

```
p = new FilePermission("/applets/tmp/scratch", "read");
```

A permission allowing a read on any file in `/applets/tmp`; that is, a permission denoted by the pair (`/applets/tmp/*`, `read`) implies our example permission `p`, but not vice versa. Similarly, a given socket permission `s` implies another socket permission `t` if and only if `t` covers the same IP address and port numbers for the same set of actions.

Alert readers might have noticed something funny about the implies method: Each permission class says which other permissions it implies. This is a bit like Johnny writing himself a note saying he can drive Dad's car. It seems safer to require Dad's signature on the note. Similarly, it would be safer if permission for `A` to imply `B` had to be granted by `B`.

Policy

Security policy in Java 2 can be set by a user (which is a bad idea since, as we know, users like dancing pigs) or a system administrator (which in a Catch-22–like situation is also a bad idea since system administrators are severely overworked). The policy is represented by a policy object as instantiated from the class `java.security.Policy`.

The policy is a mapping from identity (as defined earlier) to a set of access permissions granted to the code. The policy object is a runtime representation of policy usually set up by the VM at startup time (much like the Security Manager).

An example policy object (in plaintext form) is shown here:

```
grant CodeBase "https://www.rstcorp.com/users/gem", SignedBy "*" {
     permission java.io.FilePermission "read,write", "/applets/tmp/*";
     permission java.net.SocketPermission "connect", "*.rstcorp.com";
};
```

This policy states that any applet that arrives from the Web URL "www.rstcorp.com/users/gem", whether signed or unsigned, can read and write any file in

the directory `/applets/tmp/*` as well as make a socket connection to any host in the domain `rstcorp.com`. Policies are usually made of many grant clauses.

In practice, policy is set in a plaintext configuration file and is loaded into the VM at startup. In these policies, a public key (usually a very long string of bits) is signified by an alias. The alias is the name of a signer represented as a string. For example, a popular alias is the string `"self"`, meaning your own private key. Primitive mechanisms are included to create and import public keys and certificates into the Java 2 system. (See Appendix C for the details.)

By default, Sun's VM expects to find a system policy in the file `<java.home>/lib/security/java.policy` (where `<java.home>` is a configurable Java property). This policy can be extended on a per-user basis. User policy files can be found in a user's home directory in the file `.java.policy`. The VM loads the system policy at startup and then loads any relevant user's policy. If neither policy can be found, a built-in default is used. The built-in default policy implements the base Java sandbox model.

It is possible to specify a particular policy to use when invoking an application. This is carried out by using the Java-property–defining `-D` flag as follows (for the example, our application is the appletviewer):

```
appletviewer -Djava.policy=/home/users/gem/policy <applet>
```

Note that when application policy is defined in this way, neither the system policy nor any user policy is enforced.

Mapping Policy

Code's identity is checked against the entries of a policy object to determine what permission(s) a piece of code should be given. At the most basic level of understanding, a match is made when both the origin and the signature match. In terms of origin, this means the URL defining the origin for a piece of code is a prefix of a policy entry's `CodeBase` pair. In terms of signature, this means one public key corresponding to the signature carried by the code matches the key of a signer in the policy. Verification of signatures makes use of functionality in the `java.security.cert` package, which is a Java implementation of X.509v3 certificates.

Code can be signed with multiple signatures. In case the signatures a piece of code carries have different policy entries, all entries apply in an additive fashion. That means code is given the union of all permissions in every match (see Figure 3.7).

Consider the program X shown here. In one case, X is signed only by thing1. In another, code is signed by both thing1 and thing2. In the second case, the

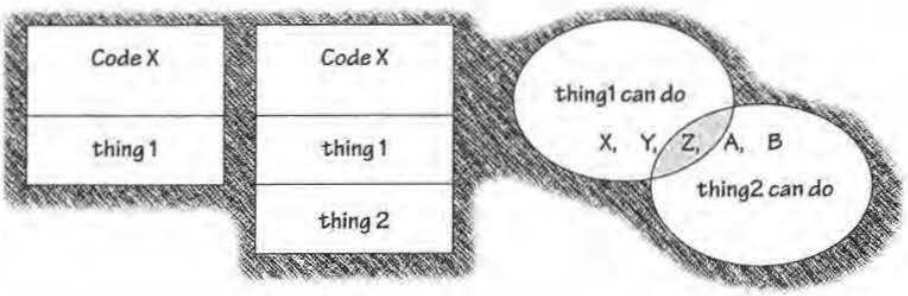


Figure 3.7 The danger of additive policy.

Consider the program X shown here. In one case, X is signed by thing1. In another, code is signed by both thing1 and thing2. In the second case, the policies of both thing1 and thing2 apply to the code (meaning in this case that it has more permission to do dangerous activities). A policy administrator may forget to anticipate what happens when code is signed by multiple keys.

policies of both thing1 and thing2 apply to the code (meaning in this case that it has more permission to do dangerous activities). A policy administrator may forget to anticipate what happens when code is signed by multiple keys.

Protection Domains

Sun says that classes and objects in Java 2 Java belong to *protection domains*. In fact, *protection domain* is just a fancy name for a bunch of classes that should be treated alike because they came from the same place and were signed by the same people. (The fact that *protection domain* means something completely different to people familiar with the security literature is reason enough to avoid the term.) An object or class belongs to one and only one protection domain. This should ring a bell, since classes can have one and only one class loader (the one that loaded them). So really this is a new way of describing a somewhat familiar logical construct for grouping classes together. A class belongs to the protection domain associated with the class loader that loaded the class.

Permissions are granted to protection domains and not directly to classes and objects, as Figure 3.8 reflects. The class `java.security.ProtectionDomain` is private in its package and is used internally to implement protection domains. As we discussed earlier, a domain is made up of a set of objects belonging to a principal. In Java 2, protection domains are based on identity and can be created "on demand." The Java runtime maintains a mapping from code to protection domains to permissions (see Figure 3.8).

System security policy specifies which protection domains should be created and which protection domains should be granted what permissions.

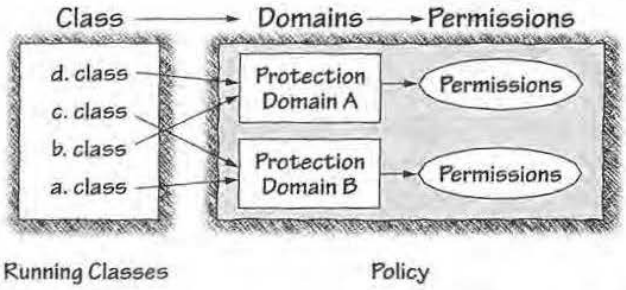


Figure 3.8 Grouping classes together to map them to policy.
Classes map into what Sun calls protection domains which in turn map to permissions. Policy is defined in terms of protection domains.

There is one protection domain that is special: the system domain. The system domain includes all system code loaded with the Primordial Class Loader. This includes classes in the CLASSPATH. The system domain is given special privileges.

Access Control

The `java.security.AccessController` class implements a stack inspection algorithm similar to the one we described earlier. Any code is allowed to query this class, which performs a dynamic inspection of the relevant thread's runtime stack. The method used to implement the check is `checkPermission()`, which takes as its argument a `Permission` object. If the call returns silently, permission is granted and the potentially dangerous computation can proceed. If the call fails, an `AccessControlException` is thrown.

Using the Access Controller

Access control under JDKs previous to Java 2 typically used the Class Loader and Security Manager to make access control decisions. For example, the following code snippet checks whether a file `/tmp/junk` can be read in the old-fashioned way:

```
ClassLoader loader = this.getClass().getClassLoader();
if (loader != null) {
     SecurityManager sm = System.getSecurityManager();
     if (sm != null) {
          sm.checkread("/tmp/junk");
     }
}
```

Here's how to do the same thing in Java 2 fashion (using the Access Controller):

```
FilePermission p = new FilePermission("/tmp/junk", "read");
AccessController.checkPermission(p);
```

The Access Controller call performs the appropriate stack inspection.

Privilege

Up through JDK 1.2beta3, Sun's JDK used the primitives `beginPrivileged` and `endPrivileged` as versions of the stack inspection primitives `enablePrivilege` and `disablePrivilege` we described in our discussion of stack inspection. These are the calls that a piece of privileged system code (that is allowed to do things like perform file access) was supposed to use to grant temporary permission to less-trusted code. These calls were featured in a number of technical publications from Sun [Gong, et. al., 1997; Gong and Schemers, 1998].

The idea is to encapsulate potentially dangerous operations that require extra privilege into the smallest possible self-contained code blocks. The Java libraries make extensive use of these calls internally, but partially trusted application code written using the Java 2 model will be required to make use of them, too.

Correct use of the JDK primitives required using a standard `try/finally` block is as follows:

```
<normal code>
try {
     AccessController.beginPrivileged();
     <insert dangerous code here>
} finally {
     AccessController.endPrivileged();
}
<more normal code>
```

This usage was required to address the problem of asynchronous exceptions (though there was still some possibility of an asynchronous exception being thrown in the `finally` clause sometime before the `endPrivileged()` call).

Wallach and Felten first explained a particularly efficient way to implement stack inspection algorithms in [Wallach and Felten, 1998]. Unfortunately, Sun decided to abandon the multiprimitive approach to stack inspection (which could benefit from Princeton's security-passing style implementation). In fact, JDK 1.2beta4 introduced a completely new API for privileged blocks. The new API removes the need for a developer to: 1) make sure to use `try/finally` properly, and 2) remember to call `endPrivileged()`. The `try/finally`

usage was symptomatic of a problem that could only really be fixed with some changes to the VM specification and its resulting implementations.

In order to properly implement the early API, VMs would have been forced to keep track of the `beginPrivileged()` call (unless they adopted a security-passing style approach). This requires tracking a stack frame (the one where the `beginPrivilege` is mentioned) and matching the beginning of a privileged block to its corresponding end—every time a privileged block is used. Doing all this bookkeeping is inefficient and thwarts optimization tricks that compilers like to play. For example, just in time (JIT) compilation approaches are hard to adapt to this model. Plus it turns out that security boundaries are crossed many thousands of times a second, so even a slight delay gets magnified quickly.

Bookkeeping would slow the VM down, which is about the last thing Java VMs need now as they near native C speeds. A Sun document explaining the change (from which some of the material here was drawn) is on the Web at *www.javasoft.com/products/jdk/1.2/docs/guide/security/doprivileged.html*.

The new API interface wraps the complete enable-disable cycle in a single interface accessed through a new `AccessController` method called `doPrivileged()`. That means the VM can efficiently guarantee that privileges are revoked once the method has completed, even in the face of asynchronous exceptions.

Here's what the new usage looks like. Note the use of Java's new inner classes capability:

```
somemethod() {
    <normal code>
    AccessController.doPrivileged(new PrivilegedAction() {
          public Object run() {
                <insert dangerous code here>
                return null;
          }
    });
    <more normal code>
}
```

Ironically, one of our developer rules for writing more secure Java code is to avoid using inner classes (see Chapter 7, "Java Security Guidelines: Developing and Using Java More Securely")! But if you want to include privileged blocks in your Java 2 code, you are encouraged to use them. In addition to the inner-class problem, verbosity is also a problem with the new API.

It turns out that using the new API is not always straightforward. That's because anonymous inner classes require any local variables that are accessed to be `final`. A small diversion can help explain why this is.

Closures

The new API is doing its best to simulate what programming languages researchers call *closures*. The problem is, Java doesn't have closures. So what are they anyway? And why are they useful?

Functions in most programming language use variables. For example, the function `f(x)=x+y` adds the value of the formal parameter `x` to the value of variable `y`. The function `f` has one free variable, `y`. That means `f` may be evaluated (run) in different environments in which the variable `y` takes on different values. In one environment, `E1`, `y` could be *bound* to 2. In another environment, `E2`, `y` could be bound to 40. If we evaluate `f(2)` in `E1`, we get the answer 4. If we evaluate `f(2)` in `E2`, we get the answer 42.

Sometimes we want a function to retain certain bindings that its free variables had when it was created. That way we can always get the same answer from the expression. In terms of our example, we need to make sure `y` always takes on a certain value. What we want is a *closed* package that can be used independent of the environment in which it is eventually used. That is what a *closure* is. In order to be self-contained, a closure must contain a function body, a list of variables, and the bindings of its variables. A closure for our second example might look like this:

```
[{y=40;} f(2)=2+y]
```

Closure is particularly useful in languages with first-class functions (like Scheme and ML). In these and other related languages, functions can be passed to and returned from other functions, as well as stored in data structures. Closure makes it possible to evaluate a function in a location and external environment that may differ from where it was created. For more on this issue, see [Friedman, et al., 1992; Fellisen and Friedman, 1998].

As we said before, Java does not have closures. Java's anonymous inner classes come as close to being closures as Java gets. A real closure might include bindings for the variables that can be evaluated sometime in the future. In an anonymous inner class, however, all state must be made `final` (frozen) before it is passed in. That is, the final state is the *only* visible state inside the inner class. This is one reason anonymous inner classes are not true closures. The problem of making everything `final` turns out to have strong implications for the use of the new privileged block API.

Local Variables

For example, in the following code, the variable `lib` must be declared final if it is to be used inside the privileged block:

```
randommethod() {
    <normal code>
    final String lib = "awt";
    AccessController.doPrivileged(new PrivilegedAction() {
          public Object run() {
               System.loadLibrary(lib);
               return null;
          }
    });
    <more normal code>
}
```

Making all local variables that are to be accessed in the block `final` is a pain, especially if an existing variable can't be made `final`. In the latter case, the trick is to create a new `final` variable and set it to the non-`final` variable just before the call to `doPrivileged`. We predict this will be a source of both headaches and errors. Errors may lead to security problems.

What Comes Out

Another problematic issue with the new interface is the fact that the inner class always returns an `Object` (a return type seen throughout the Java language). That means if a call to a piece of privileged code (for example, a call to `System.getProperty()`) usually returns something other than an `Object` (for example a `String`), it will have to be dynamically cast to the usual type. Using a `final` variable to pass types out is possible, too. Unfortunately, both of these operations will incur a runtime performance hit (especially casting). The returns-only-`Object` problem is another source of potential errors.

Whence the Change

It is good that VM vendors want their machines to be fast and efficient; however, purely in terms of security, it is unclear whether the decision to change the API was a good one. Not that the previous API was perfect, but the new one seems to introduce several places in which errors are bound to be made by developers charged with actually *using* VMs. The real answer to the problem is introducing closures to Java. Closures are something to look for in future JDK versions.

It turns out that using the new API is not always straightforward. That's because anonymous inner classes require any local variables that are accessed to be `final`. A small diversion can help explain why this is.

Closures

The new API is doing its best to simulate what programming languages researchers call *closures*. The problem is, Java doesn't have closures. So what are they anyway? And why are they useful?

Functions in most programming language use variables. For example, the function `f(x)=x+y` adds the value of the formal parameter `x` to the value of variable `y`. The function `f` has one free variable, `y`. That means `f` may be evaluated (run) in different environments in which the variable `y` takes on different values. In one environment, `E1`, `y` could be *bound* to 2. In another environment, `E2`, `y` could be bound to 40. If we evaluate `f(2)` in `E1`, we get the answer 4. If we evaluate `f(2)` in `E2`, we get the answer 42.

Sometimes we want a function to retain certain bindings that its free variables had when it was created. That way we can always get the same answer from the expression. In terms of our example, we need to make sure `y` always takes on a certain value. What we want is a *closed* package that can be used independent of the environment in which it is eventually used. That is what a *closure* is. In order to be self-contained, a closure must contain a function body, a list of variables, and the bindings of its variables. A closure for our second example might look like this:

```
[(y=40;) f(2)=2+y]
```

Closure is particularly useful in languages with first-class functions (like Scheme and ML). In these and other related languages, functions can be passed to and returned from other functions, as well as stored in data structures. Closure makes it possible to evaluate a function in a location and external environment that may differ from where it was created. For more on this issue, see [Friedman, et al., 1992; Fellisen and Friedman, 1998].

As we said before, Java does not have closures. Java's anonymous inner classes come as close to being closures as Java gets. A real closure might include bindings for the variables that can be evaluated sometime in the future. In an anonymous inner class, however, all state must be made `final` (frozen) before it is passed in. That is, the final state is the *only* visible state inside the inner class. This is one reason anonymous inner classes are not true closures. The problem of making everything `final` turns out to have strong implications for the use of the new privileged block API.

Local Variables

For example, in the following code, the variable `lib` must be declared final if it is to be used inside the privileged block:

```
randommethod() {
    <normal code>
    final String lib = "awt";
    AccessController.doPrivileged(new PrivilegedAction() {
          public Object run() {
               System.loadLibrary(lib);
               return null;
          }
    });
    <more normal code>
}
```

Making all local variables that are to be accessed in the block `final` is a pain, especially if an existing variable can't be made `final`. In the latter case, the trick is to create a new `final` variable and set it to the non-`final` variable just before the call to `doPrivileged`. We predict this will be a source of both headaches and errors. Errors may lead to security problems.

What Comes Out

Another problematic issue with the new interface is the fact that the inner class always returns an `Object` (a return type seen throughout the Java language). That means if a call to a piece of privileged code (for example, a call to `System.getProperty()`) usually returns something other than an `Object` (for example a `String`), it will have to be dynamically cast to the usual type. Using a `final` variable to pass types out is possible, too. Unfortunately, both of these operations will incur a runtime performance hit (especially casting). The returns-only-`Object` problem is another source of potential errors.

Whence the Change

It is good that VM vendors want their machines to be fast and efficient; however, purely in terms of security, it is unclear whether the decision to change the API was a good one. Not that the previous API was perfect, but the new one seems to introduce several places in which errors are bound to be made by developers charged with actually *using* VMs. The real answer to the problem is introducing closures to Java. Closures are something to look for in future JDK versions.

The Security Manager Revisited

As we described in Chapter 2, "The Base Java Security Model: The Original Applet Sandbox," the Security Manager up until JDK 1.1 invoked a direct `check()` method for dangerous resource access control. This method was responsible for evaluating the request and denying or granting access. The new Security Manager in Java 2 still supports the use of `check()` methods, but now many of these calls are actually implemented to make use of the Access Controller and Permission objects (whenever possible).

It would be best to dispense entirely with the Security Manager, but history dictates that it remain available for reasons of backwards compatibility. Breaking all existing JDK 1.1 code in order to introduce a new security design is not an economically viable approach for Java.

The Secure Class Loader

Java 2 introduces the class `java.security.SecureClassLoader`, which is a concrete implementation of the abstract `ClassLoader` class. It tracks the code source and signatures of each class, and hence assigns classes to protection domains. All Java code is loaded by a Secure Class Loader (except for code loaded by the Primordial Class Loader) either directly or indirectly (that is, by another class loader that was itself loaded by the Secure Class Loader). For more on class loading, refer to Chapter 2.

Sandboxing Java Applications

Now that the security enforcement mechanisms are more complex and do not rely on the distinction between applet code and built-in code (as in the early days), it is possible (and desirable!) to force Java applications, in addition to applets, to run within the (highly mutable) sandbox. This means application code can be made to cohere with locally defined security policy. Java 2 provides a mechanism for doing this with the class `java.security.Main`. The implementation ensures that local applications stored in the `java.app.class.path` are loaded with the Secure Class Loader. It is a good idea to have applications run from this location as opposed to placing them in the CLASSPATH where they will be treated as built-in code.

Adding Permissions

It is possible to add new permissions to Java that are tailored to your specific needs. This is done by subclassing and extending the `java.security.Permission` class we detailed earlier. The new

permission classes that you create should be stored in the application package where they apply.

Next, a representation of the permission (that is, a string representing a policy entry) needs to be added to the policy file. This ensures that the permission is "automatically" configured for each domain.

Finally, the application code itself may include a section that manages resources. This section of code should make use of the `checkPermission()` method of the `AccessController` class (explained earlier). Use of this method obviates the need to think about Class Loaders and Security Managers.

Outside the Sandbox

Java 2 clearly introduces significant changes to the Java security landscape. It is likely that the days of black-and-white security policy for mobile code are numbered. With the major changes to Java's security architecture come a number of important responsibilities, the most important of which is mobile code policy creation and management. The tools are still primitive, but the policy itself is essential.

Also essential to any mobile code system that makes use of code signing is solid key management capability. Although the subject of public key infrastructure (PKI) is really beyond the scope of this book, we at least invoke some important concepts. Managers responsible for setting and maintaining policies based on signed code will encounter issues including choice of certificate authority, who to issue keys to, how to ensure that private keys are kept private, whether to get a corporate key and how to protect it, how to disable keys of employees who leave an organization, where to store keys, and so on. These are nontrivial issues that have yet to be worked out in the real world.

Hopefully, widespread support for code-signing systems will quickly appear on consumer desktops worldwide. Truthfully, the PKI is much less mature than many security researchers and pundits predicted it would be by now. This is partly because deploying an effective PKI is much more difficult than it sounds. But it is also at least partially due to the greed of certificate authorities who chose to charge developers for identities (public/private key pairs) instead of issuing them for free and charging elsewhere for their use. Without a solid PKI, systems like Java 2 Java may take a while to catch on. We predict that signed mobile code will find its most pervasive use among early adopters as an intranet technology (as opposed to an Internet technology). Of course, we're very well prepared to be wrong about that.

For a long time, Java developers have wanted some way in which less restriction could be placed on their applets. At the same time, managers in many

enterprises have been searching for ways to manage code (not just mobile code, but any code) more securely. In its Java 2 guise, Java offers a powerful answer to these needs.

We would be irresponsible not to note that with code signing comes a host of new risks to manage. Most notable among the risks are two: first, that the implementation will have holes (JDK 1.1 code signing has already fallen prey to this risk); and second, that security policies will get too complicated to understand and manage.

APPENDIX C

How to Sign Java Code

This tutorial was put together by John Viega and Tom O'Connor, both research associates at Reliable Software Technologies. The four major sections each describe a separate vendor's code-signing tools, including:

Netscape's Object Signing

Microsoft's Authenticode

Sun's JDK 1.1 Code Signing

Sun's Java 2 Code Signing

Some of the tools are tricky to figure out and use. This tutorial should help.

Before you dig into this tutorial, you should read Chapter 3, "Beyond the Sandbox: Signed Code and Java 2," which discusses the major impact that signed code has on the Java security architecture. Of special interest are the sections entitled *Signed Code* (see page 88) and *Trust* (see page 92). The material there discusses the notions of trust, digital signatures, and certificate authorities.

Signing Classes with the Netscape Object Signing Tool

First in our tutorial, we'll take on Netscape's Object Signing Tool that can be used to sign Java code (among other things). As in all of these systems, step one is obtaining an identity.

Getting a Certificate

Most digital signature schemes (PGP being a notable exception) involve the use of a Certificate Authority (CA)—an organization that can vouch for someone's signature. After all, why trust code just because it carries a signature? We need an objective third party to make sure people are who they say they are. That means the first task in code signing is to obtain the proper credentials from a CA. There are many CAs that can sell you certificates for signing Java code.

Netscape has links to CAs that support their Netscape Object Signing Tool at *https://certs.netscape.com*. You can visit that page and pick a CA. Make sure that the CA you choose provides a certificate that can be used to sign objects (some certificates can't).

VeriSign (*www.verisign.com*) offers many flavors of Digital IDs. It heads up Netscape's list of CAs. We'll use VeriSign as an example for obtaining a certificate; however, note that the process will differ depending on the CA that you choose. To get a VeriSign certificate for Netscape Object Signing, select Software Developer ID from the popup list at the top of VeriSign's homepage. Choose Netscape Object Signing from the page that follows. There are two kinds of Software Developer IDs: a Class 2 Individual ID, and a Class 3 Commercial ID. The Class 2 ID costs $20 annually, while the Class 3 ID is a whopping $400 annually. For our purposes, we'll focus on Class 2 certificates. After selecting Class 2, fill out the information form that asks who will be identified by the certificate (making sure to include the all-important billing information).

VeriSign will do a limited background check on you before it will issue a certificate. For example, it checks the data you enter against information publicly available on you through a credit check. If your request for a certificate is accepted, VeriSign will email you a PIN and a URL that you can use to retrieve the certificate. For an individual Class 2 certificate, the verification process is usually close to instantaneous.

Once you receive that information, open the URL with Netscape Communicator and then enter your PIN. Communicator will install the certificate in itself automatically. If you are using a shared version of Communicator, someone may have already entered a password for the certificate database that is stored in your browser. You will need this password before you can download and install your certificate; otherwise, you will be prompted to enter a password for the certificate database. Although this password is optional, it does prevent people from starting up your version of Netscape and stealing your certificate by exporting it. You definitely don't want your certificate stolen, because then other people can sign applets as you. Password information can be found in the Security Info box of the Communicator menu, under Passwords.

If everything is successful, your certificate will appear in the Security Info box; check by going to Yours under Certificates.

There are several things to know as you sign up for a certificate:

1. You do not want a Class 1 certificate, as it cannot be used to sign objects.
2. Use Netscape Communicator (4.x), even though you are allowed to request and download a certificate using Netscape Navigator 3.x, because the support for certificates in 3.x is not as good as it is in later browsers. For example, object signing tools may not be able to locate your certificate inside Netscape 3.x. Also, you may not be able to export your certificate, which is useful if you want to sign code from a machine other than the one from which you originally downloaded the certificate.
3. Use the same browser on the same computer both to request and to retrieve the certificate. If the browser is set up with multiple user profiles, make sure you use the same user profile as well; otherwise, you will likely be unable to retrieve your certificate.
4. Finally, note that many versions of Netscape Communicator will be unable to verify your certificate (assuming you got a VeriSign certificate). Unfortunately, information concerning this problem seems to have disappeared from the Netscape Web site.

Exporting and Importing Certificates

It is a good idea to export your certificate to a file, just in case you install a new version of Communicator over your old one. Doing so also allows your certificate to follow you to other machines.

To export a certificate, bring up Communicator's Security Info dialog box. Select the certificate you wish to export by clicking on its name. Then, click on the Export button. At this point, you may be asked to enter the password that protects your local browser's certificate database. Next, you will be asked to enter a password to protect the certificate data. This password is used to make sure that no one can steal your certificate if he or she sees an exported copy of it somewhere (unless that person is able to crack your password, so choose wisely!). To make sure you typed the password in correctly, you will be asked to enter it again. Assuming you've entered the same password both times, Netscape will prompt you for a filename, which it will use to store the certificate. Once you enter the filename, you're finished exporting the certificate. You can copy that file to another machine so you can sign code from there as well.

To import a certificate into a new browser, bring up Communicator's Security Info box. Click on Yours, which is a subitem of Certificates. Press the button, Import a Certificate. If you have not previously entered the password protecting the certificate database of the local copy of Netscape, you will now be prompted to enter it. After you enter the correct password, a file dialog box will come up; use it to select the file containing your certificate.

Once you have selected the file, you will be prompted for the password used to protect the certificate, which is the password that you entered when you exported the certificate. At this point, assuming all has gone well, you should get a dialog box indicating success.

Netscape Object Signing Tool 1.1

The Netscape Object Signing Tool is a command line program that creates digital signatures for files. These signatures aren't stored in the files themselves; they're stored in the JAR file in which you bundle your applet. Note that since digital signature information is transmitted in JAR files, you must package your applets in a JAR file in order to sign them, even if they consist only of a single class. The important syntax for using a JAR file with the HTML APPLET tag is:

```
<APPLET CODE="somefile.class" ARCHIVE="jarfile.jar">
```

where somefile.class is the class in the JAR file where execution should begin, and jarfile.jar is the URL of the JAR file.

The Netscape Object Signing Tool may be downloaded from *developer.netscape.com/software/signedobj/jarpack.html*.

The tool is available for most operating systems. While, as of this writing, version 1.0 is still available for download, we recommend that you use version 1.1.

After the download is complete, unpack the archive file in a directory. Included are three files: `readme.txt`, `license.txt`, and `signtool`. To make signing objects easier, put the directory that contains signtool in your PATH environment variable, as per your operating system. For example, a Windows 95 user who unpacked the tool to `C:\nos` would run the following line (and then add it to the `autoexec.bat` file):

```
PATH=%PATH%;C:\nos
```

Before attempting to sign anything, check to see if `signtool` is able to locate the certificate that will be used to sign objects. Unix flavors of signtool look for certificates in the `$HOME/.netscape`. If your local Netscape files are kept somewhere else, or if you are using the Win32 version, signtool must be explicitly told the path to the certificates. This is done with the -d flag. On Win32, this path is commonly `c:\Program Files\Netscape\Users\name`, where `name` is the name of your Netscape Profile. To verify that your signing certificate was installed properly, run `signtool -l` or, if your certificate cannot be found,

```
signtool -d"<path to certificates>" -l
```

For example, if your certificates were stored in `C:\nos`, you would type:

```
signtool -d"C:\nos" -l
```

If your certificate still does not appear in the listing, verify that the certificate is installed in Netscape properly. (See the instructions given earlier). Also check that the path to the Netscape `.db` files was properly specified. If all else fails, check with Netscape and the issuing Certificate Authority. Make note of the full name of your certificate as it appears in the listing, you will need these data when it comes time to sign.

Create a directory in which to put all the class files for the applet you wish to sign. Once all the class files that make up the applet are in the right place, the signtool program can create a signed JAR file in one step.

Navigate into the directory containing the soon-to-be signed classes. To sign the classes and create a JAR file in one step, issue the command:

```
signtool -d"<path to certificate>" -k"<name of certificate>"
-e ".class" -Z myjar.jar .
```

If your Communicator Certificate Database is password protected, signtool will prompt for the password before signing the classes. The "." at the end of

the command should be the last thing to appear. It specifies that the signing should begin in the current working directory. The `signtool` command recursively signs files by default. To keep the tool from recursing through directories, add `--norecurse` to the command line.

Here's a brief explanation of the flags used in the previous example, as well as some of the other more useful flags for signing applets:

-k "certificate name": Specifies the certificate with which you would like to sign. This flag is necessary when signing an applet. The certificate name should be the entire name of the certificate as it appeared as the output of `signtool -l`. Since the certificate name is likely to have spaces in it, make sure you place it in quotes; otherwise, the signing will fail.

-e".extension": Specifies the file extensions to sign. If you don't include this flag, the tool will sign all files, as opposed to the preceding example, which uses this flag to sign `.class` files only.

-x"name": Allows you to sign all files except a particular file or directory. An example where this might be useful is when you are using an untrusted library in your applet. You probably will not want to vouch for code you did not write!

-Z"jarfile": Specifies the name of the JAR file to create. If you omit this option, you will have to JAR everything up yourself.

When the JAR file is created, `signtool` can be used to test the validity of the signatures. This is done by issuing the command:

```
signtool -d"<path to certificate>" -v myjar.jar
```

`signtool` will list the contents of the JAR and verify that they have been signed, and that they have not been tampered with since the signature was created.

You may also check to see who signed the JAR file:

```
signtool -d"<path to certificate>" -w myjar.jar
```

`signtool` can be used to sign anything, not just Java files. In fact, it can extract JavaScript from HTML files, and sign just the JavaScript; however, that functionality is outside the scope of this tutorial. Documentation on signtool is available from Netscape at *developer.netscape.com/docs/manuals/signedobj/signtool/*.

Adding Capabilities to Your Classes

As Chapter 3 describes, signing a Java applet does much more than just allow people to verify that you signed it. It can also give your applets the chance to step outside the Java sandbox. If your applet has a digital signature vouching for it, then the applet may request special privileges, such as accessing the file system. However, the user of the applet doesn't have to let your applet do what you request just because you sign it.

The special privileges an applet can request are called capabilities by Netscape.[1] Predictably, no two browsers support flexible privileges in quite the same way, so privilege-management code will only work with one browser. (So much for "write once, run anywhere"!) As a result, while Netscape keeps its own internal version of these classes, in order to actually compile and test an applet that can request them, you must download the library from *developer.netscape.com/docs/manuals/signedobj/capsapi_classes.zip*

Put the zip file in your CLASSPATH (or otherwise edit the CLASSPATH). Now you will be able to develop code that requests extra privileges in Netscape. Note that you should not include these classes with your applet; the Netscape browser running on the remote machine will use its internal version of the classes.

The Capabilities library provides a class called the Privilege Manager that handles requests from the program to turn on and off privileges. When the first request to enable a certain privilege is made, the Privilege Manager prompts the browser's user, showing the certificate used to sign the code requesting the privilege, and asking whether the privilege should be granted. See Figure C.1. If the user agrees to grant the privilege, the privilege is granted for the lifetime of the applet. However, once the applet has obtained a privilege, it can turn the privilege off and on at its discretion.

To request a particular privilege to be enabled, you use the static method `enablePrivilege()` of class `netscape.security.PrivilegeManager`. The method takes a single String argument, which is the name of the privilege to enable. Some useful privileges include:

UniversalFileAccess: This privilege gives the applet the ability to access any file available to the user. It will enable the applet to call most things in the `java.io` package related to file manipulation. This privilege is a superset of other file manipulation privileges that may be requested individually, such as UniversalFileRead, UniversalFileWrite, and UniversalFileDelete.

UniversalSendMail: This privilege allows the applet to send email on behalf of the user.

[1] Although Netscape and some other vendors use the term *capabilities* to refer to this system, this is not really a correct use of the term *capability*, which is a technical term in the security literature.

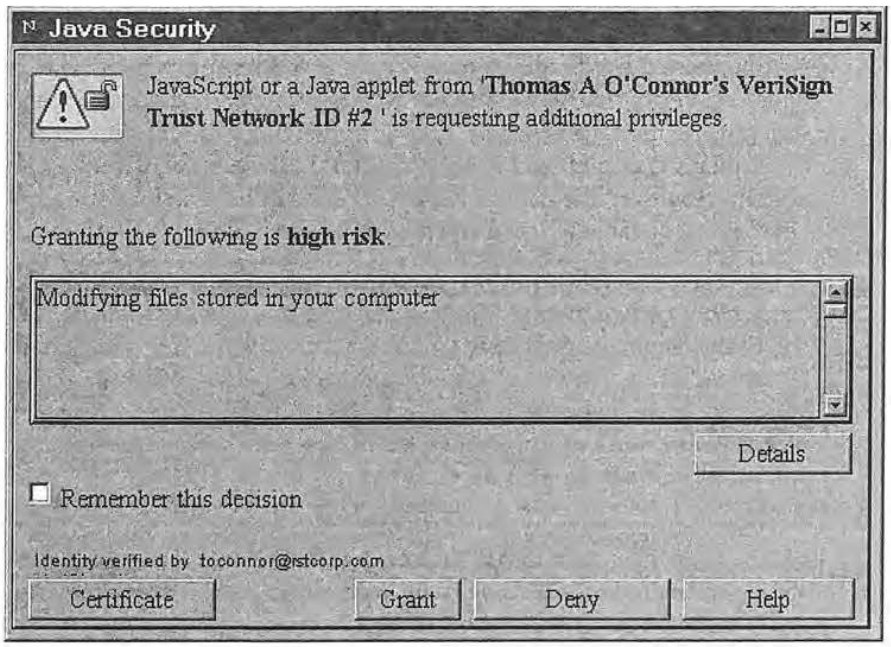


Figure C.1 Netscape Navigator's Privilege Manager alerts a browser user with this window.

The dialog box explains which dangerous privileges have been requested and who is vouching for the applet (through a digital signature). Clicking the "Remember this decision" is probably a bad idea.

UniversalExitAccess: Allows the Java applet to shut down the Netscape browser.

UniversalExecAccess: Enables the applet to run programs already stored on the user's local computer.

PrivateRegistryAccess: Grants access to application-specific portions of the computer's registry (Win32 only).

There are many more privileges that an applet can request. For a full list, see the documentation for the Capabilities API at *developer.netscape.com/docs/manuals/signedobj/capabilities/01cap.htm*.

A call to `enablePrivilege` will throw an exception that the applet must catch if the user decides not to grant the privilege specified in the call. Thus, the applet must be prepared to catch instances of `netscape.security.ForbiddenTargetException`.

Here's a sample applet called `FirstTry.java` that uses `enablePrivilege`

```
import java.applet.*;
import java.awt.*;
import netscape.security.PrivilegeManager;
import netscape.security.ForbiddenTargetException;

public class FirstTry extends Applet {

 private TextArea ta = new TextArea(10,100);

 public void init() {
 this.add(ta);
 this.show();
 try {
 PrivilegeManager.enablePrivilege("UniversalFileRead");
 ta.appendText("Read enabled!\n");
 }
 catch (ForbiddenTargetException fte) {
 ta.appendText("Read not enabled.\n");
 ta.appendText(fte.toString());
 }
 catch (Exception e) {
 ta.appendText("Unexpected exception!\n");
 }

 public void paint(Graphics g) {

 }
}
```

The `FirstTry` applet doesn't do anything with the privilege it asks for, even if it is granted. However, it *would* be able to read any file, including the system password file on a Unix system, if it tried. That could be considered an abuse of privilege. Another potentially bad thing this applet could do would be to put the `enablePrivilege` call inside the `paint` method. Doing this will cause the browser to continually prompt the user for permission every time `paint` is called, which will happen until permission is granted, or until the browser is killed. Actually, the Netscape Grant/Deny window has a checkbox that says "Remember this decision." Checking the deny box will make this pop-up never appear again. The take home message is that signed applets can be hostile too.

When you enable a privilege, it does not have to stay enabled for the entire execution of the applet. There are a couple of ways to turn privileges off (which is always a good idea). First, when the method that calls `enablePrivilege` returns, the privilege will automatically be disabled. As

a result, you should not use a helper method to enable a privilege, because once execution returns from that method, the privilege will no longer be enabled. Second, you can call `revertPrivilege`, which also takes the name of a privilege as an argument. Finally, you can call `disablePrivilege`, which turns off a particular privilege. In no case will the granting of the privilege be revoked; the applet can turn the privilege back on by simply calling `enablePrivilege` again. To see an example of a signed applet, surf to *www.rstcorp.com/javasecurity/applets/dpig/netscape.html*.

Signing Java Applets with Microsoft's Authenticode

Next in our tutorial, we'll take on Microsoft's code-signing system for Java. It's a bit peculiar since it does not interact with the JDK 1.1 or Java 2 security models in an intuitive fashion. As usual, step one is securing an identity.

Getting an Authenticode Certificate

There are several ways to get a certificate for Microsoft Authenticode. One of the things you can do is generate "test certificates," which allows you to try things out. We'll tell you how to do that in a bit, if you just want to play around. However, if you plan on distributing any code, you're going to want to get a real digital ID. This costs money. A number of vendors distribute certificates, one of them being VeriSign, which we'll use in our examples.

To obtain a VeriSign certificate for Authenticode, point Internet Explorer to *digitalid.verisign.com/developer/ms_pick.htm*.

Select a flavor of ID. For personal use, select a Class 2 ID. For business use, select Class 3.

You'll be given a form to fill out. Note that the personal Class 2 ID is $20, and you'll have to pay by charge card. Once you submit the form, VeriSign will try to verify you are who you say you are, mainly by running a credit check. Sometimes the credit check won't have up-to-date information, so if you get rejected and you can remember the address for the last place you lived (which is the most common problem), you might want to try it again using old information, pretending you never moved. (Not that we condone this strategy, mind you.)

Once your data are approved, VeriSign will send you an email with instructions on picking up the certificate. When downloading your certificate, two files will need to be saved: your private key file, and your certificate file. You should probably save these files to a floppy disk instead of your hard drive,

so that someone can't just snag your certificate off your computer (although, without knowing the password you use to protect your private key, snagging the files alone may not do a bad guy much good). Remember the password used to protect the private key; it will be needed when it comes time to sign code. For the sake of simplicity, we'll assume you saved your certificate as `a:\Cert.spc` and your private key as `a:\Key.pvc`.

Getting the Signing Software

Before signing anything with the new certificates, download and install the Microsoft Java SDK. It's located at *www.microsoft.com/msdownload/java/sdk/31f/SDK-JAVA.asp*.

We'll assume you installed the Java SDK in the directory `C:\SDK-Java.31`.

All of the programs we're going to need for signing Java code live in `C:\SDK-Java.31\Bin\PackSign`, so you should probably add that directory to your PATH. Under Windows 95/98, running the following command at the DOS prompt will fix up your PATH for the current session:

```
PATH=%PATH%;C:\SDK-Java.31\Bin\PackSign
```

You can add that command to your `autoexec.bat` file to make the change persist through a reboot.

Cabinet Files

Unlike Netscape's Object Signing and Sun's signing tools (which work on JAR files), Authenticode signing will only work on cabinet (CAB) files. There's nothing special about the CAB format; it's just another way of archiving many files into one. However, it's the only archive format IE supports for signing Java code.

Say we have an applet that consists of two files: `file1.class` and `file2.class`. We can create a CAB file in the same directory by typing the following at the DOS prompt:

```
cabarc N test.cab file1.class file2.class
```

If there are no other class files in the directory, we can also type:

```
cabarc N test.cab *.class
```

Security Zones

In order to understand what we're doing when we sign a CAB file, we need to know a little something about what an IE "security zone" is. By default, a security zone is a group of Web sites. Each zone is assigned a security level, which may be Low, Medium, High, or Custom. We won't cover Custom zones, except to say that they can implement arbitrary security policies. For more on security zones, see Chapter 1, "Mobile Code and Security: Why Java Security Is Important."

There's a default zone called Trusted Sites, into which a user can put any server. All code from that zone will be completely trusted (i.e., the zone has a Low security level). Similarly, there's a Restricted Sites zone. Any sites the user puts in this zone will need explicit permission before they can run anything "outside the sandbox." By default, most everything else falls into the Internet zone, which is assigned a Medium security level. Code can run outside the Java sandbox in a very limited manner. For example, code can use up to 1 megabyte of data on your hard drive by using the API `com.ms.io.clientstorage`, which is included with Microsoft VMs only. (So much for "write once, run anywhere"!) Unlike fully trusted applets, applets restricted to the Medium security level should not otherwise be able to use your file system.

We're going to sign our cabinet file, requesting to run either with Medium or High privileges (we can also request Low privileges, but since we'll always be allowed to run in the sandbox, doing so is mainly useful only to show you vouch for the CAB file). If our code ends up in a Low security zone, our code will always run without prompting the user for permission. If our code ends up in a Medium security zone, then before code that requests Medium level privileges can run, the user will be prompted as to whether to let the code run. If our code ends up in a High security zone, all code that wants to run outside the sandbox will need to be approved through a dialog with the user. See Figure C.2.

Signing CAB Files

To sign test.cab, we're going to use the `signcode` command, which is included in the Java SDK. Here's a typical command line:

```
signcode -j JavaSign.dll -jp High -spc a:\Cert.spc -v a:\Key.pvk
-n "My Applet" -i http://www.mywebpage.com/ test.cab
```

The flags here are a bit arcane. If you want your CAB file to request permissions, the `-j` flag should always be there, and take `JavaSign.dll` as a parameter, unless you're signing something other than Java code (the same

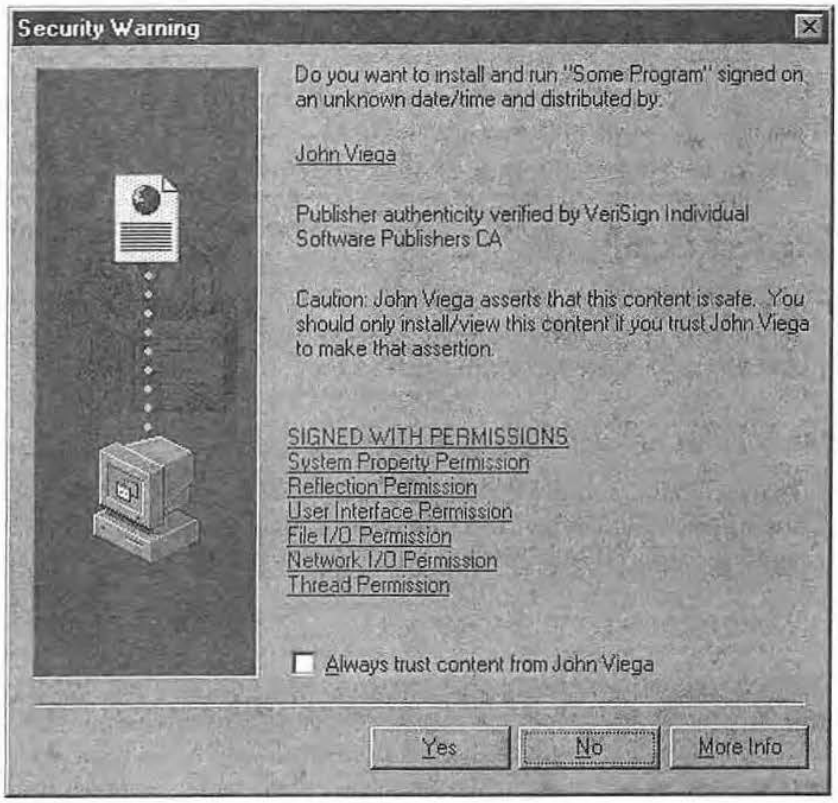


Figure C.2 The security warning dialog used by Microsoft Internet Explorer's Authenticode system.

This dialog explains who has vouched for the code (by signing it) and what permissions are being requested. Clicking "Always trust content from <user>" is probably a bad idea.

command can be used to sign ActiveX controls and other mobile code, too). The `-jp` flag passes a parameter to the DLL. For Java signing, that's how we request High privileges. The `-spc` flag and `-v` flag are used to specify the location of your certificate and private key, respectively. The `-n` option needs to be present, and it specifies the name of the software, which is displayed to the user before the user decides whether to run your code. The `-i` option specifies where to go for more information about the product, which also gets displayed when the user is prompted to give your code permission to run.

You can also "timestamp" your signature, so that after your certificate expires, your applet will still work. However, doing so requires a time-stamp server, which isn't covered here. For more information on Authenti-code for Java, visit *www.microsoft.com/java/security.*

To confirm that everything has worked properly so far, run the command:

```
chkjava test.cab
```

A window should appear similar to the one an end user will see when IE asks if the application should be allowed to run.

Making Test Certificates

To avoid putting down some cash for a real certificate from a CA and still be able to play around with Authenticode, you can make a test certificate. The first step is to create the certificate with the command:

```
makecert -sk Key.pvk -n "CN=Your Name" Cert.cer
```

That command makes a certificate and a private key you can use in other applications, but it won't work for code signing. To get it to work with code signing, convert it to a Software Publisher Certificate (SPC) by typing:

```
cert2spc Cert.cer Cert.spc
```

When you're finished with that, you can use `Key.pvk` and `Cert.spc` for testing purposes in the same way as if they came from a CA.

Special HTML Tag

When deploying a signed CAB file in an HTML page, a slight variation on the <APPLET> tag is necessary. As with all applets, the name of the class that extends `java.applet.Applet` goes in the CODE attribute. However, instead of putting the name of the CAB file in the ARCHIVE attribute as is done with JAR files, CAB files signed with Authenticode are passed using the PARAM tag. As an illustration, the tag to embed into a web page the signed applet `"MyApplet"` stored in `myapp.cab` would look like:

```
<APPLET CODE="MyApplet.class"> <PARAM name="cabbase"
VALUE="myapp.cab"></APPLET>
```

The named prameter "cabbase" is how Internet Explorer finds the CAB file containing the class specified in the CODE attribute.

Comparing Authenticode to Netscape Object Signing

Microsoft's Authenticode model is somewhat simpler than the Communicator model for the end user. Assuming the user doesn't know anything about zones, lots of stuff runs without asking the user for permission; the user is prompted only to approve code generally when the code requests full access, and doesn't already have permission. Less interaction generally means less hassle for the user. You can make more dialog boxes disappear if you check boxes like, "always trust code from this person," and "always trust code from this site," which appear in the window that announces that code is trying to gain permissions. However, spreading trust around so easily just to avoid dialog boxes can have bad consequences.

Authenticode is also simpler for the developer. There's no need for calls to a Capabilities library, meaning you can simply request an access level, as opposed to requesting a set of privileges. However, Netscape is capable of finer-grained access control, which allows the applet to secure only the resources it needs to run without a user feeling the need to give a program complete access to the computer.

Another convenience of Authenticode over Object Signing is that the user only gets prompted at most once per applet. Netscape prompts the user whenever new privileges are requested (which is usually during execution). While the Netscape model is more intrusive, it does afford the user a bit more control over what privilege is granted to an applet.

Signing Code with Sun's JDK 1.1.x

Sun makes its own set of signing tools. The tools have evolved along with the JDK. We'll briefly cover both the JDK 1.1 tools and the Java 2 tools.

The JDK ships with a command-line tool called `javakey`. Its job is to manage a database of entities (people, companies, etc.) and their public/private keys and certificates. It is also supposed to generate and verify signatures for archive files; however, verification is not implemented as of JDK 1.1.7.

As Chapter 3 describes, an applet contained within a digitally signed JAR file is allowed to leave the bounds of the Java sandbox under certain circumstances. In JDK 1.1, if a JAR is signed and the user who has browsed to the Web site containing the applet has a policy stating that he or she trusts the person who signed the JAR, the applet can do anything at all that Java code is capable of. For example, it can read and write from the file system, start another process

running on the computer (outside of the browser), open a network connection to an arbitrary machine, or myriad other tasks that applets are not normally allowed to do. In other words, trusted signed code under JDK 1.1 is as powerful as Java application code from the JDK 1.0.2 days. Remember, under JDK 1.1, we're operating under a black-and-white security model.

To get going with code signing in JDK 1.1, there are few things to gather. On the development side, an applet that tries to perform actions that aren't normally allowed by the Java sandbox is needed (or at least one that can be augmented to attempt such an action). The most rudimentary operation that a signed applet can do that an unsigned applet can't do is read the `user.name` System property. An example applet follows:

```
public class UserApplet extends java.applet.Applet {
 public void init() {
 String username = "user: ";
 try {
 username += System.getProperty("user.name");
 } catch (SecurityException se) {
 username += "cannot read";
 }
 showStatus(username);
 }
}
```

A signed applet containing the preceding code (running in a browser of a user who trusts the entity that signed the applet) will be able retrieve the name of the user running the applet and display it in the status bar of the browser.

Once the applet to be signed and its containing Web page have been created, the class files that contain the applet must be put into a JAR file. Even if the applet in question is only one class, it must be placed inside a JAR file. It is not possible to sign standalone class files.

In order to sign Java code with `javakey`, a signing certificate needs to be created. Once this certificate is created, it can be used to sign the JAR file and distributed to users who wish to allow the signer's applet full access to their system.

Creating a Signing Certificate

A file called `identitydb.obj` stores all certificate information and lives in the directory specified by the Java System Property value `user.home`. For Unix Java users, this value evaluates to `$HOME`. For Win32 users, `user.home` can take a number of values. Different VMs assign the value of `user.home` either to the `USERPROFILE` directory, to the `HOMEDRIVE\HOMEPATH` direc-

tory, or when all else fails, to the value of the `java.home` System Property. To clear up any ambiguity, write and run a simple Java containing the line:

```
System.out.println("user.home= " +
System.getProperty("user.home"));
```

Regardless of `user.home`, the location of `identitydb.obj` can be set explicitly by adding an `identity.database` entry to the `java.security` file that lives in the `lib` subdirectory of the java installation, wherever that may be on the system.

First, the signer's identity must be created in the database. To create an identity, `signername`, that will be able to sign objects, run the following command on the command line:

```
javakey cs signername true
```

Now that a signer has been created, that signer's public and private keys must be initialized. Keys can be between 512- and 1024-bits long. To initialize public and private keys for a signer, run the following command (where # is a number between 512 and 1024):

```
javakey gk signername DSA #
```

Higher numbers mean more security. We recommend always using 1024-bit keys. The parameter DSA signifies the algorithm used to generate the keys. The JDK only comes with the DSA algorithm by default.

To verify that all has gone well so far, run:

```
javakey ld
```

This command will list all the information in the current identity database. The entry for `signername` should identify it as a trusted signer as well as noting that the public and private keys have been initialized. The next step generates a certificate that will be valid for signing JAR files. This is different from Netscape Object Signing in that there is no Certificate Authority involved.

First, a directives file must be created. The directives file is a Java Properties file that provides values used during certificate generation. Here are the contents of an example directives file:

```
issuer.name=signername
subject.name=signername
subject.real.name=Sol S. Signer
subject.country=US
```

```
subject.org=Signing Corp
subject.org.unit=Development
start.date=22 Jul 1998
end.date=16 Aug 1999
serial.number=41
```

To generate the certificate once the directives file exists, run:

```
javakey gc directivesfile
```

To verify that the certificate was generated properly, run:

```
javakey ld
```

Look for the signername entry to have the `subject.*` information from the directives file listed.

Everything is now finally set for signing. The command actually used to sign a JAR file also requires a directives file. This is a different directives file than the one used to generate the signing certificate. The first directives file used to generate the certificate will no longer be needed, unless a different certificate needs to be generated. The second kind of directives file is used whenever a JAR gets signed, and should be kept handy. An example directives file for signing a JAR looks like this:

```
signer=signername
# look at javakey ld for certificate numbers, should be 1
cert=1
# chain unsupported, must have as value=0
chain=0
# must be 8 characters or less and not conflict with any other
# digital signatures that could be inserted into the JAR
signature.file=anything
out.file=Signed.jar
```

Once the signingdirective file has been created, run the command:

```
javakey gs signingdirective UnsignedApplet.jar
```

Running this command will generate `Signed.jar`, which will be a signed version of `UnsignedApplet.jar`. Putting `Signed.jar` in the ARCHIVE field of the APPLET tag in an HTML page will cause a Java-enabled browser to bring the JAR file over the network and execute the signed code contained within. For more information on `javakey`, the official Sun documentation for Solaris can be found at *java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/javakey.html*.

The Win32 specific version can be found at *java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javakey.html*.

Testing the Signed Applet

Now that a signed applet exists, and it is embedded within a Web page, it's ready for testing before release. Testing requires either `appletviewer` or a Web browser that knows how to validate JARs signed by `javakey` and allows signed JARs to leave the sandbox. Unfortunately, neither of the two major browsers (Netscape Communicator and Microsoft Internet Explorer) support javakey-signed JARs.

HotJava and the `appletviewer` program that comes with the JDK can validate JARs signed by `javakey`. They will allow signed applets out of the sandbox if the signature is valid and the policy states that the user whose signature appears is trusted. Both of these programs search for the `identitydb.obj` in the same manner that javakey does.

The problem is that no one should be surfing the Net with HotJava (too dangerous), and the `appletviewer` cannot be used to browse the Internet. Since the VMs in Communicator and Internet Explorer do not support `javakey` signing, in order run javakey-signed applets with those browsers, users must download and install Sun's Java Plug-In.

Java Plug-In for Communicator and Internet Explorer

Java Plug-In can be used to run applets instead of the browser's default VM. The Java Plug-In can be configured to use the most recent version of the Java Runtime Environment available from Sun. When an applet is run through the Java Plug-In instead of the browser's default VM, javakey-signed JARs can be verified and can step outside of the sandbox (if policy allows).

Users must download the Plug-In from Sun and install it on their system. The download page for the Plug-In is *java.sun.com/products/plugin/index.html*.

Applet developers also need to modify the HTML pages that contain their applets and modify the <APPLET> tag. Applets that are in Web pages using the standard <APPLET> tag will still be run by the browser's default page. The Plug-In will run applets only when it detects a different set of HTML tags that specify an applet. Sun provides an application called HTMLConverter, which can convert pages with the <APPLET> tag into pages containing tags that will launch applets using the Plug-in. The HTMLConverter homepage is *java.sun.com/products/plugin/converter.html*.

Two things to note about using the Plug-In. On Solaris, JavaScript *must* be enabled for the Plug-In to work properly. With JavaScript disabled, applets did not load or run when we tested the Plug-In with Communicator 4.02 and 4.06.

On Win32, the Java Plug-In did not find the `identitydb.obj` file in the same place that `javakey` did. This has to do with different versions of the VM setting different values of the `user.home` Property. If you run into trouble, try moving the `identitydb.obj` file to a different location. Places to try are mentioned in the section, *Creating A Signing Certificate* (see page 298).

Distributing Public Keys and Certificates

In order for someone to verify who signed a signed JAR, he or she needs the public key of the entity who signed the JAR in the first place. Until the public key is distributed to people other than its owner, no one but the owner can verify that an applet is signed and by whom. Once the signed applet has been tested and has proven to be functional, it can be placed on a Web site for use by others.

In order for the applet to escape the sandbox imposed by other people's browsers, users must have the public key or certificate of the entity that signed the applet. Also, the user must tell the identity database that he or she trusts the entity that signed the applet. Trusting the entity that signed the applet allows the applet *complete access* to the host. Here's how to create a trusted entry:

```
javakey c signername true
```

To import the signer's public key contained in keyfile, run:

```
javakey ik signername keyfile
```

To import the signer's certificate (which contains the signer's public key) from certfile, run:

```
javakey ic signername certfile
```

The identity must be created in the database before trying to import either the public key or certificate. In order to verify the signature on a signed JAR, you need only the public key of the signer. Certificates include the public key.

The signer of the applet must make his or her public key (or certificate) available to users of the applet in some way. It could be linked from a Web site, phoned in, or delivered through email. Whichever way it is done, the identity must first be extracted from the identity database. To extract a public key from the database to a file keyfile, use the command:

```
javakey ek signername keyfile
```

To extract signername's certificate number 1 to a file certfile, use the command:

```
javakey ec signername 1 certfile
```

The information in the `keyfile` or `certfile` should be given to those who want to create a policy that allows applets signed with the identity to leave the sandbox.

Differences Between Netscape Object Signing and JDK 1.1.x `javakey`

There are five major differences between Netscape and Sun's approach to code signing:

1. Netscape Object Signing *only* works within Communicator. JDK 1.1 signed applets can work in any browser, although Netscape Navigator and Microsoft Internet Explorer both require the installation of the Java Plug-In for the applet to leave the sandbox.
2. Netscape Object Signing requires getting a certificate from a certificate authority such as VeriSign. JDK 1.1 users can generate their own certificates.
3. Netscape Object Signing requires no modifications to HTML tags. If the Plug-In is needed for JDK 1.1 (in case you want to use IE or Netscape), the <APPLET> tag must be changed by HTMLConverter.
4. Netscape Object Signing uses Netscape's own classes to step outside of the sandbox. A Netscape-specific exception is thrown when permission to leave the sandbox is denied. JDK 1.1 javakey-signed applets do not need to include calls to any other non-java.* classes to leave the sandbox, and `java.lang.SecurityException` is thrown when permission is denied.
5. Netscape Object Signing prompts the user when an applet attempts to leave the sandbox, asking the user for permission to carry out the dangerous act. Actions are grouped, so the user can allow some actions (file reads) but not others (file writes). JDK 1.1 javakey-signed applets that are trusted get complete access to the host.

Signing Code with Sun's Java 2

The `javakey` tool from JDK 1.1 has been replaced by two tools in Java 2. One tool manages keys and certificates in a database. The other is responsible for signing and verifying JAR files. Both tools require access to a keystore

that contains certificate and key information to operate. The keystore replaces the `identitydb.obj` from JDK 1.1. New to Java 2 is the notion of policy, which controls what resources applets are granted access to outside of the sandbox (see Chapter 3).

The `javakey` replacement tools are both command-line driven, and neither requires the use of the awkward directive files required in JDK 1.1.x. Management of keystores, and the generation of keys and certificates, is carried out by `keytool`. `jarsigner` uses certificates to sign JAR files and to verify the signatures found on signed JAR files.

Getting Started with `Keytool`

The first step in working with Java 2 is getting the latest beta version from Sun. Members of the Java Developers Connection (JDC) can download Early Access releases of Java 2 software. Membership in the JDC is free with registration. Once registered, point your browser to *developer.java.sun.com/developer/earlyAccess/jdk12/index.html.*

The Win32 version of JDK 1.2beta4 comes with the latest version of the Java Plug-In, which supports Java 2. During the Install, answer *yes* when it wants to know if the JRE and the Plug-In should be installed as well—it will be needed later.

The keytool command operates on a keystore file. The name of the keystore file is `.keystore` by default, and it is located in the directory named by the `user.home` Java System Property. It is possible to have multiple keystores. Changing the keystore on which the current `keytool` command will operate is done through the `-keystore <path to keystore>` option.

Documentation from the Sun Java Web pages states that Java 2 VMs will run and properly authenticate JARs signed with JDK 1.1's `keytool`. It also states that the last beta release of JDK 1.2 does not yet support 1.1-signed JARs. The `keytool` utility also supposedly allows porting keys and certificates from 1.1 `identitydb.obj` files into a Java 2 keystore. According to the documentation, the command to perform the translation is:

```
keytool -identittydb -file <path to identitydb.obj file>
```

Unfortunately, an `identitydb.obj` file created with JDK 1.1.6 did not successfully import into the keystore when we tested the `keytool` from JDK 1.2 beta4—we tried on both Win32 and Unix platforms. The error message returned from `keytool` mentions an `InvalidClassError` and states that a class used in key management became obsolete, resulting in a serialization error. Until this problem works itself out in later beta and production

versions, the certificates and keys used under 1.1 cannot be used in Java 2. New certificates and keys will need to be generated for use with `jarsigner` and `keytool`.

Generating a public and private key pair and self-signed certificate can be performed from the command line in one shot without the need to create any directives files. All keys and certificates stored in the keystore are accessible through an *alias*. An alias is a name associated with a certificate entry that keytool uses to uniquely identify each certificate under its control. To generate a certificate keyed by the alias `keyname`, run the command:

```
keytool -alias keyname -genkey
```

`keytool` will begin prompting for information. The first prompt is for a keystore password, which will be needed for all further `keytool` and `jarsigner` operations on this keystore. It must be at least six characters long and is unfortunately echoed to the screen as it is typed. This means that the keystore password can be leaked to casual observers whenever `keytool` or `jarsigner` is used.

Once the password has been entered, `keytool` prompts for some personal information, such as name, company name, city, state, and country. All this information is stored in the generated self-signed certificate, which is saved in the default keystore location. All the personal information is displayed for verification before `keytool` generates the keys and certificate. After the certificate and keys are generated, `keytool` prompts for another password. Each certificate has its own password, separate from the keystore password. Entering nothing does *not* give the key an empty password. It gives the certificate the same password as the keystore. `jarsigner` will not prompt for the passwords of certificates that have the same password as the keystore, so it may appear that a certificate has no password. However, if the password of the keystore changes, the passwords of the certificates do not change, so `jarsigner` will start prompting for not only the password of the keystore, but for the certificate as well. The command to change the password of a keystore is:

```
keytool -storepasswd
```

`keytool` will prompt for the old password, and the new password twice, all in cleartext. This command does not affect the passwords of certificates in the keystore, including those that happen to have the same password as the keystore.

An apparent weakness of the keytool certificate generation system is that a user can accept all the default values for the personal information prompted for before certificate generation. The default value for all the questions is

"Unknown." So keytool will generate a valid certificate that can be used to sign JAR files, but is filled with bogus information. No data validation is performed by `keytool`, so it is possible to, say, create a certificate for Elvis.

Certificates generated by the system will be valid for just under one year by default. To change the length of validity for a certificate to n days, add the flag `-validity n` to the keytool `-genkey` command.

To view the fingerprints of certificates in the keystore, use the command:

```
keytool -list
```

To view the personal information about the issuer and owner of the certificate, run:

```
keytool -list -v
```

Signing a JAR

Once a private key has been generated, `jarsigner` can be used to mark a JAR file with the public key of the signer. The command to sign a JAR file called `SignMe.jar` with the keyname private key generated previously is:

```
jarsigner SignMe.jar keyname
```

`jarsigner` will prompt for the keystore password and the private key password if different than the keystore password before signing the JAR file. To monitor the progress of the signing process, run:

```
jarsigner -verbose SignMe.jar keyname
```

`jarsigner` can also be used to verify that a JAR has or has not been signed, and by whom. For a simple signed/not signed answer for a JAR file `Unknown.jar`, run:

```
jarsigner -verify Unknown.jar
```

To get more information from the verification process, such as the signing status of each file in the JAR file, the personal information from the certificates used to sign each file in the JAR, and whether or not the certificate is known in the default keystore, run:

```
jarsigner -verify -verbose -certs Unknown.jar
```

After each signed file in the listing will be the personal information encoded in the certificate for the entity that signed the file. If that certificate is known in the keystore, the name it is known by will appear in parentheses after the certificate's personal information.

Enter the CA

So far, the only changes from JDK 1.1 are the syntax and the names of the commands. Certificates can be generated by `keytool` with any personal information at all. There is nothing to stop anyone from creating a certificate that claims that it is owned by someone else and signing a JAR with it. What a Certificate Authority can provide is a level of assurance that a certificate truly represents the individual that it claims to represent. That is, of course, if you trust that a Certificate Authority isn't being spoofed, and is properly checking the certificates it vouches for. (Recall, this way madness lies.)

Certificates generated by `keytool` can be exported in a form suitable for submission to a Certificate Authority such as VeriSign. This can be accomplished by running:

```
keytool -certreq -alias keyname -file requestfile
```

That command puts a Certificate Signing Request into requestfile for the certificate know by the `keyname` alias. However, there is no information as to how to submit this data to a CA for validation. According to the `keytool` documentation, the CA will validate the certificate and return something that must be imported into the keystore. Although we haven't tested it, the command to import the response from the CA into the keystore is supposed to be:

```
keytool -import -alias newalias -trustcacerts -file response
```

That command imports the response from the CA stored in a file called response into the keystore under the name `newalias`, which must not already exist in the keystore. The `-trustcacerts` flag tells `keytool` to check the response certificate against the five VeriSign certificates that come shipped with Java 2 (at least there were five in JDK 1.2beta4).

Turning Over the Keys

Until the certificate used to sign the JAR is made public, no one can grant any permissions to the enclosed applet. To retrieve a copy of the keyname certificate from the keystore into a file `mycert`, use:

```
keytool -export -alias keyname -file mycert
```

As usual, `keytool` will prompt for the appropriate passwords. When the command finishes, the file `mycert` can be distributed to users who wish to grant additional privileges to applets signed by that certificate.

As in JDK 1.1, there is currently limited support for a JAR signed with the JDK tools. Again, Sun provides support through the Java Plug-In. Plug-In version 1.1.1 does not necessarily support Java 2. Although the Java Plug-In can be configured to use different VMs installed on the local system, the Plug-In hangs the browser when pointed to a Java 2 VM on Solaris. Documentation for the Win32 version of the Plug-In mentions running a program off the Start menu to configure the Plug-In. The installation script does not create a program group for a Plug-In Control Panel as advertised under Windows NT unless the user performing the installation has permission to create program groups.

An Early Access version of Plug-In version Java 2 for Solaris is available to members of the Java Developer's Connection. The latest version of the Plug-In for Win32 ships with JDK 1.2beta4.

As with JDK 1.1, any HTML pages that contain Java 2-signed JAR files must be converted using the same HTMLConverter used in JDK 1.1. Converting the HTML ensures that the applet will run in the Plug-In and not in the browser's default VM. See the section on JDK 1.1 JAR signing for information on where to get the HTMLConverter.

Running a Signed Applet

The first step upon encountering a signed applet is to locate the certificate of the entity that signed the JAR file and import it into the local keystore. Assuming that the certificate can be located and placed into a file called `acert`, run:

```
keytool -import -alias analias -file acert
```

An entry in the keystore is created keyed by the name `analias` for the certificate stored in `acert`. This is now a trusted entity. Whereas in JDK 1.1, aliases could either be trusted or untrusted, all aliases in Java 2 keystores are trusted. However, in JDK 1.1, trusted aliases could do anything they wanted; aliases in Java 2 cannot do anything unless granted permission. Permissions are granted to aliases through the use of policy files (see Chapter 3).

Creating a Simple Policy for Signed Applets

Java 2 introduces the notion of *policy*. Creating, understanding, and managing security policy for signed mobile code is a difficult and complex problem. Since this discussion is about signing code and not about constructing policy,

an extremely simple example of how to construct policy is presented. Creating good policy is beyond the scope of this tutorial. The example policy is strong enough to allow an applet limited file access to the host machine.

Java policy files can be created with the new `policytool`. This application has a GUI to guide users though the many twists and turns encountered when creating policy files. It's a very simplistic GUI with no online help. In its current form as of beta4, it is only useful if one does not know the syntax of a policy file.

Policy files are plaintext files that follow a format outlined at *java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html*.

The default security policy system first reads a system-level policy file from the `lib/security/` subdirectory under the Java installation directory. It then tries to read a `.java.policy` file from the current user's `user.home` directory. In this file, users specify their personal security policy, which merges with the system security policy. Permissions that can be granted in a Java policy file are outlined at *java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html*, as well as in Chapter 3.

If the policy file is to make reference to a certificate stored in a keystore, a keystore entry must appear in the policy file. The keystore entry specifies the path relative to the policy file itself and the name of the keystore file. To keep things simple and use the default keystore file, add the following line to the `.java.policy` file in the `user.home` directory:

```
keystore ".keystore";
```

To grant an applet permission to write or create any file in the `c:\tmp` directory, assuming the applet comes from *www.friendly.com/~mybuddy/applets/* and is signed by a certificate known in the default keystore as friend, add to the `.java.policy` file:

```
keystore ".keystore";
grant signedBy "friend",
 codeBase "http://www.friendly.com/~mybuddy/applets/" {
  permission java.io.FilePermission "c:\\tmp\\*", "write";
};
```

Note the double backslashes. All Win32 pathnames must use double backslashes to indicate directories. Unix pathnames use regular singleton forward slashes. `CodeBase` follows URL syntax.

Sign Only Privileged Code

Applets that request permission to leave the sandbox are usually built for greater purposes than saving a high-score list on the local drive. Applets that do serious business and hence require access to the local system are most likely some of the larger applets in existence. It is unlikely that these applets will be built completely by one developer or one software company. Chances are some of the components of an applet will be bits of utility code found on the Internet or purchased from a tool vendor. A smart organization wants to sign only code that it produces; third-party utility code cannot be safely vouched for.

If all the code is signed, then any code can leave the sandbox based on the policy. However, if some code in an applet is from a third party, it should not be signed unless the individual signing the code is willing to vouch that the third-party code won't try to do anything malicious (or introduce a security hole that others can exploit). To say the least, we don't recommend signing code you don't completely understand.

Java 2 presents an API for privileged blocks. Privileged blocks are meant to be small sections of code that have a higher privilege than the code that invoked them. JDK 1.2beta4 introduced a new API for privileged blocks. Using this API, the only code that needs to be signed is the code that invokes the `AccessController` class, and the code that performs the privileged action.

All other code can remain unsigned, preventing it from leaving the sandbox on its own (or tempting others to attack it). Documentation on the new API can be found at *java.sun.com/products/jdk/1.2/docs/guide/security/doprivileged.html*.

There are two things to consider when writing signed code that will be integrated with unsigned code. First, make the code in the privileged block as small as possible. The less code that is privileged, the less chance that granting it higher privilege will result in nasty and unwanted side effects. Second, to prevent mix-and-match attacks, all the code for the applet should live in one JAR file, even if the third-party libraries that are used by the applet live in their own JAR. (See *Guidelines for Java Developers* in Chapter 7, "Java Security Guidelines: Developing and Using Java More Securely.")

To sign some portions of a JAR file and leave others unsigned takes a number of steps we'll cover now. First, create a JAR file containing all classes that need to be signed.

```
jar cvf MyApp.jar Signme1.class Signme2.class
```

List all the classes that need to be signed in the previous command. Once the JAR containing classes that need to be signed is created, sign the JAR with `jarsigner`.

```
jarsigner MyApp.jar mykey
```

Now, add the remainder of the classes in the application to `MyApp.jar`. The Java 2 version of `jar` added the `v` flag, which allows JAR files to be updated with new files.

```
jar uvf MyApp.jar Other1.class Others.class
```

List the remaining classes in the application in this step. If parts of the application are already in a JAR or ZIP file, they will need to be unarchived before being JARed into the new partially signed JAR file. To verify that all went correctly, use `jarsigner` to verify the contents.

```
jarsigner -verify -verbose MyApp.jar
```

Only the classes that were added before `jarsigner` was invoked the first time to create the signature will be marked as signed. All the other classes will be listed, but no certificate or signature will be associated with their listing.

If `jarsigner` fails to verify the entire JAR, or classes that are supposed to be signed appear not to be, use the `jar` command to list the contents of the JAR.

```
jar tvf MyApp.jar
```

The first entry in the JAR *must* be `META-INF/MANIFEST.MF`. If the manifest file is missing or not in the first position in the file, the JAR will not verify properly. Following the `MANIFEST.MF` file should be a `.SF` and `.DSA` (or `.RSA`) file. If either of those files is missing, then the signature is missing from the JAR. Remove the JAR file and start over. If the commands listed earlier still move the `META-INF/MANIFEST.MF` file out of the first position in the file, it may not be possible to create a JAR containing signed and unsigned code. (The JAR command with JDK 1.2beta4 did not move the `META-INF/MANIFEST.MF` file around in the JARs we created.)

Differences between JDK 1.1 Code Signing and Java 2 Code Signing

There are a number of major differences between Sun's approach to code signing in JDK 1.1 and Java 2:

1. JDK 1.1 trusts code completely or does not trust it at all; Java 2 allows policy to define what code can and cannot do. This reflects the change from black-and-white trust to shades-of-gray.

2. JDK 1.1 has one tool, `javakey`, for all code-signing related functions; Java 2 has `keytool` for certificate management and `jarsigner` for signing and verifying JARs.
3. JDK 1.1 does not support certificates from Certificate Authorities; Java 2 does allow Certificate Authorities to sign generated certificates, however it is unclear if any CAs currently offer this service.

In Conclusion

Both Netscape and Microsoft have provided browser-specific methods for leaving their sandboxes. Both rely on external Certificate Authorities to manage identities, but the same certificate used for Netscape cannot be used for Microsoft. Netscape requires applets to use special classes to take advantage of code signing. Microsoft also provides a vendor-specific API for certain capabilities. Both take a similar approach when it comes to prompting the browser's user when certain applets attempt to leave the sandbox.

Sun has moved from a black-and-white security policy that allowed trusted code to do anything it wants to a shades-of-gray security policy by which only certain code from certain people can do certain things, depending upon configuration. However, in Java 2, unsigned code can be granted free reign of the system as well if the policy is configured as such. Having unsigned code play outside the sandbox is something that none of the other schemes allow.

Each of the four Java code-signing techniques discussed in this tutorial vary in their complexity level, have their own special tools for signing and key management, have different levels of support from VM to VM, and take different approaches to the user's interface to security controls. Considering that Java is meant to be a portable, mobile code system, the large number of compatibility issues surrounding code signing is worrisome. Developers want their applets to do more than the original JDK 1.0.2 sandbox model allowed, but with each vendor providing different ways for code to leave the sandbox, the goal of "sign once, leave the sandbox anywhere" seems highly unlikely.

References

Abadi, M., Burrows, M., Lampson, B., and Plotkin, G. (1993) A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

Anderson, R. and Kuhn, M. (1996) Tamper Resistance—A Cautionary Note. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, pp. 1–11. Also available on the Web at www.cl.cam.ac.uk/users/cm213/Publications/tamper.html.

Badger, L. and Kohli, M. (1995) Java: Holds Great Potential—But Also Security Concerns. *Data Security Letter*, 3:12–15. The Data Security Letter (DSL) is published by Trusted Information Systems (TIS).

Boneh, D., DeMillo, A., and Lipton, R. (1997) On the Importance of Checking Cryptographic Protocols for Faults. In W. Funny (ed) *Advances in Cryptology—Eurocrypt'97*, Volume 1233 of *Lecture Notes in Computer Science*, pp. 37–51, Springer-Verlag. Also available on the Web at theory.stanford.edu/~dabo/papers/faults.ps.gz.

CERT (1996a) CA-96.05: Java Applet Security Manager. See URL www.cert.org/advisories/index.html.

CERT (1996b) CA-96.07: Java Security Bytecode Verifier. See URL www.cert.org/advisories/index.html.

Daconta, M. (1996) *Java for C++ Programmers*. John Wiley & Sons, New York, NY.

Dean, D., Felten, E., and Wallach D. (1996) Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 190–200, Oakland, CA.

Dean, D. (1998) *Formal Aspects of Mobile Code Security*. Ph.D. dissertation, Department of Computer Science, Princeton University.

Drossopoulou, S. and Eisenbach, S. (1998) Towards an Operations Semantics and Proof of Type Soundness for Java. A technical paper to be included in an as yet unnamed book. Available on the Web at outoften.doc.ic.ac.uk/projects/slurp/papers.html.

Erdos, M., Hartman, B., and Mueller, M. (1996) Security Reference Model for the Java Developer's Kit 1.0.2. Available from Sun Microsystems and also as a Web document at www.javasoft.com/security/SRM.html.

Fellisen, M. and Friedman, D. (1998) *A Little Java, A Few Patterns*. MIT Press, Cambridge, MA.

Felten, E., Balfanz, D., Dean, D., and Wallach, D. (1997) Web Spoofing: An Internet Con Game. In *Proceedings of the 20th National Information Systems Security Conference*, Baltimore, MD. An early version appeared as technical report 540-96 (revised), Department of Computer Science, Princeton University.

Flanagan, D. (1997) *Java in a Nutshell, Second Edition*. O'Reilly & Associates, Sebastopol, CA.

Flanagan, D. (1997) *Java Examples in a Nutshell*. O'Reilly & Associates, Sebastopol, CA.

Friedman, D., Wand, M., and Haynes, C. (1992) *Essentials of Programming Languages*. MIT Press/McGraw-Hill, Cambridge, MA.

Garfinkel, S. and Spafford, G. (1996) *Practical Unix & Internet Security, Second Edition.* O'Reilly & Associates, Sebastopol, CA.

Ghosh, A. (1998) *E-Commerce Security: Weak Links, Best Defenses.* John Wiley & Sons, New York, NY.

Gong, L. (1998) Secure Java Class Loading. *IEEE Internet Computing*, 2(6):56–61, November/December 1998.

Gong, L., Mueller, M., Prafullchandra, H., and Schemers, R. (1997) Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems.* Monterey, CA.

Gong, L. and Schemers, R. (1998) Implementing Protection Domains in the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, CA.

Hastings, R. and Joyce, B. (1992) Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, ACM Press.

Horstmann, C. and Cornell, G. (1997) *Core Java Volume IæFundamentals.* SunSoft Press, Mountain View, CA.

Hughes, L. J. (1995) *Actually Useful Internet Security Techniques.* New Riders, Indianapolis, IN.

Hughes, M., Shoffner, M., and Winslow, M. (1997) *Java Network Programming.* Manning, Greenwich, CT.

ISO7816 (1987) International Standards Organization, International Standard ISO 7816-1 through 7816-6 "Identification Cards—Integrated Circuit(s) Cards with Contacts." Available through ISO, New York, NY.

LaDue, M. (1996) Java Security: Whose Business Is It? Published by Online Business Consultants and available as a Web document at www.rstcorp.com/hostile-applets/OBCArticle/Article.html.

Lewis, T. (1996) What's Wrong with Java? *IEEE Software*, 29(6):8. Lewis' letter to the editor was in response to Java criticism originally printed by him in "The NC Phenomena: Scenes From Your Living Room," *IEEE Software*, 29(2):8–10.

Lewis, T. (1998) Java Holy War '98. *IEEE Computer*, 31(3):126–128.

Macgregor, R., Durbin, D., Owlett, J., and Yeomans, A. (1998) *Java Network Security.* Prentice Hall, Saddle River, NJ.

Martin, D., Rajagopalan, S., and Rubin, A. (1997) Blocking Java Applets at the Firewall. *Proceedings of the 1997 Network and Distributed System Security Symposium.* San Diego, CA. March 1997. Also available on the Web at www.cs.bu.edu/techreports/96-026-java-firewalls.ps.Z.

McGraw, G. and Felten, E. (1996) *Java Security: Hostile Applets, Holes, and Antidotes.* John Wiley & Sons, New York, NY. (The first edition of this book.)

McGraw, G. (1998) Testing for Security During Development: Why We Should Scrap Penetrate and Patch. *IEEE Aerospace and Electronic Systems*, 13(4):13–15, April 1998.

Neumann, P. (1995) *Computer Related Risks.* Addison-Wesley, Reading, MA.

Oaks, S. (1998) *Java Security.* O'Reilly & Associates, Sebastopol, CA.

Rubin, A., Geer, D., and Ranum, M. (1997) *The Web Security Sourcebook.* John Wiley & Sons, New York, NY.

Schneier, B. (1995) *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. John Wiley & Sons, New York, NY.

Shimomura, T. and Markoff, J. (1996) *Takedown*: The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaw—By the Man Who Did It. Hyperion, New York, NY.

Spafford, E. (1989) The Internet Worm Program: An Analysis. *Computer Communications Review*, 19(1):17–57.

Stata, R. and Abadi, M. (1998) A Type System for Java Bytecode Subroutines. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pp. 149–160, January 1998.

Sun Microsystems (1995) The Java Language: An Overview. Available from Sun and also as a Web document at java.sun.com/docs/overviews/java/java-overview-1.html.

Sun Microsystems (1996b) The Java Virtual Machine Specification. Web document at www.javasoft.com/docs/books/vmspec/html/VMSpecTOC.doc.html. Available as a book by Lindholm and Yellin from Addison-Wesley.

Sun Microsystems (1996c) Low-level Security in Java. Web document at URL www.javasoft.com/sfaq/verifier.html/ by Frank Yellin.

Sun Microsystems (1997) Java Card 2.0 Programming Concepts, Revision 1.0 Final. Web document at URL www.javasoft.com/products/javacard/index.html.

Venners, B. (1998) *Inside the Java Virtual Machine*. McGraw-Hill. New York, NY.

Voas, J. and McGraw, G. (1998) *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons. New York, NY. See the Web site at www.rstcorp.com/fault-injection.html.

Wallach, D., Balfanz, D., Dean, D., and Felten, E. (1997) Extensible Security Architectures for Java. In *Proceedings of the 16th Symposium on Operating Systems Principles* (Saint-Malo, France), October 1997.

Wallach, D. and Felten, E. (1998) Understanding Java Stack Inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pp. 52–63, Oakland, CA.

Wallach, D. (1998) *A New Approach to Mobile Code Security*. Ph.D. dissertation, Department of Computer Science, Princeton University.

Young, Boebert, and Kain (1985) Article in an IEEE Tutorial on Computer Network Security. IEEE Press.

Web Sites Referenced in the Text

All of the following links can be found on a page of the companion Web site for this book at www.rstcorp.com/java-security.html.

Chapter 1

www.developer.com/news/techfocus/123097_pushsec.html
"Don't Push Me: The Security Implications of Push." *developer.com* TechFocus article by Gary McGraw.

java.sun.com
Java Developer's Kit (JDK) available free from JavaSoft. Also other *official* Java information.

www.javasoft.com/sfaq
JavaSoft's Frequently Asked Questions: Applet Security.

www.cs.princeton.edu/sip/java-vs-activex.html
Security Tradeoffs: Java versus ActiveX. Princeton Safe Internet Programming FAQ. Also see Appendix A.

www.oaf.org/~loverso/javascript/
JavaScript Problems I've Discovered. John LoVerso's JavaScript Security site.

www.developer.com
developer.com, an online publication for Java developers.

www.javaworld.com
JavaWorld, an online publication for Java enthusiasts and developers.

www.minq.com
MindQ, an online training company specializing in Java.

www.yahoo.com
Yahoo! An excellent starting point for Web surfing. A large Web index.

www.altavista.com
AltaVista. One of the top search engines on the Web.

www.rstcorp.com/javasecurity/links.html
Java Security Hotlist. Also see Appendix B.

www.cs.princeton.edu/sip
Princeton's Secure Internet Programming Team. Includes the *Java Security FAQ*.

lightyear.ncsa.uiuc.edu/~srp/java/javabooks.html
The Java Books list. An extensive list of all books published about Java (*way* too many).

www.rstcorp.com/java-security.html
The Java Security Web Site. This book's companion Web site. Includes the *Java Security Hotlist*.

Chapter 2

www.rstcorp.com/hostile-applets/
The Hostile Applets Home Page. A collection of hostile applets written by Mark LaDue.

Chapter 3

www.cs.princeton.edu/sip/pub/oakland98.html
"Understanding Java Stack Inspection," by Wallach and Felten.

www.javasoft.com/products/jdk/1.2/docs/guide/security/dopriviledged.html
Sun's document explaining the security API change.

Chapter 4

www.rstcorp.com/hostile-applets
The Hostile Applets Home Page.

www.digicrime.com
DigiCrime (disable Java and JavaScript before you surf this site).

www.rstcorp.com/javasecurity/applets.html
The Java Security Hotlist: Hostile Applets and Other Toys.

www.digicrime.com/exploits/javawin
Digicrime's Blue Screen of Death page.

www.digicrime.com/surprise/bluescreen.class
The actual byte code of the `bluescreen` applet.

www.ahpah.com
Ahpah software makes the SourceAgain decompiler.

www.gamelan.com
Earthweb's Java applet database.

java.sun.com/sfaq
Sun Microsystems' Frequently Asked Questions—Java Security.

www.cs.princeton.edu/sip/java-faq.html
Princeton's Java Security: Frequently Asked Questions (included as Appendix A).

www.cs.princeton.edu/sip/java-vs-activex.html
Princeton's Security Tradeoffs: Java vs. ActiveX (included as Appendix A).

www.rstcorp.com/java-security.html
The Java Security Web Site, companion Web site for this book.

Chapter 5

geek-girl.com/bugtraq
An archive of the security-related bugtraq archive.

java.javasoft.com/sfaq
JavaSoft's Frequently Asked Questions: Java Security.

www.cs.princeton.edu/sip/java-faq.html
Princeton Secure Internet Programming Team's Java Security FAQ. (Also see Appendix A).

www.alcrypto.co.uk/java/
Major Malfunction and Ben Laurie explain the security holes they discovered.

www.cs.princeton.edu/sip
Princeton's Secure Internet Programming Team.

kimera.cs.washington.edu
University of Washington's Kimera Project.

kimera.cs.washington.edu/flaws/sunflaws0423.html
Type safety problems discovered in Sun's Verifier by the Kimera Project.

kimera.cs.washington.edu/flaws/msflaws0423.html
Flaws discovered in Microsoft's Verifier by the Kimera Project.

neurosis.hungry.com/~ben/msie_bug
Ben Mesander's applet `WhereDoYouWantToGoToday`.

Chapter 6

www.cs.princeton.edu/sip/pub/secure96.html
Princeton's seminal paper, *Java Security: From HotJava to Netscape and Beyond*.

www.cli.com/software/djvm/index.html
Formalizing the JVM at Computational Logic, Inc.

www.javasoft.com/security/SRM.html
JavaSoft's *Security Reference Model for JDK 1.0.2.*

www.isbe.ch/~wwwinfo/sc/cb/tex/jasmin/guide.html
The Jasmin bytecode assembler.

www.ahpah.com
Ahpah Software sells the SourceAgain Java Decompiler.

www.finjan.com
Finjan Software, Ltd.

www.rstcorp.com/hostile-applets/rube.html
Mark LaDue takes on Finjan.

www.rstcorp.com/hostile-applets/drowning.html
Mark LaDue takes on Finjan again.

www.digitivity.com
Digitivity.

www.security7.com
Security7.

www.withinreach.co.il
WithinReach.

www.cultdeadcow.com
Cult of the Dead Cow produces the Back Orifice exploit

www.esafe.com.
eSafe

www.cs.princeton.edu/sip/JavaFilter
Princeton Secure Internet Programming Team's Java Filter Class Loader.

www.icsa.net
International Computer Security Association.

www.clark.net/pub/mjr/pubs/fwtest/index.html
Marcus Ranum discusses firewall certification.

www.rstcorp.com/hostile-applets/Rube/HAMGen.java
Mark LaDue's Hostile Applet Mutation Generator.

Chapter 7

www.rstcorp.com/java-security.html
The Java Security Web Site, companion site for this book.

www.rstcorp.com/javasecurity/links.html
The Java Security Hotlist.

www.javasoft.com/sfaq
Sun's Java Security FAQ.

www.cs.bu.edu/techreports/96-026-java-firewalls.ps.Z
Martin et al.'s paper *Blocking Java Applets at the Firewall.*

Chapter 8

www.gemplus.com/javacard/index.htm
Gemplus: JavaCard, and GemXpresso.

www.cyberflex.slb.com
Schlumberger: Cyberflex.

www.javasoft.com/products/javacard/index.html
JavaSoft: Java Card Technology, specifications for Card Java can be found here.

theory.stanford.edu/~dabo/papers/faults.ps.gz
Boneh, DeMillo, and Lipton's *On the Importance of Checking Cryptographic Protocols for Faults.*

www.cl.cam.ac.uk/users/cm213/Publications/tamper.html
Anderson and Kuhn's Tamper Resistance—A Cautionary Note.

www.cryptography.com/dpa/technical/index.html
Cryptography Research, Inc. information on Differential Power Analysis.

$34.99 USA/$54.50 CAN

"This book is mandatory reading for every user and developer of Webware."
—Peter G. Neumann, Moderator of the Risks Forum, from his review of the first edition

Securing Java

Java security is more important now than ever before. As Java matures and moves into the enterprise, security takes a more prominent role. But as Java evolves, its security issues and architectures get more complicated. Written by the world's leading experts on mobile code security, this updated and expanded edition of the groundbreaking guide to Java security includes lessons for Web users, developers, system administrators, and business decision-makers alike. This book navigates the uncharted waters of mobile code security and arms the reader with the knowledge required for securing Java. It provides in-depth coverage of:

- The base Java security sandbox, made up of the Verifier, Class Loaders, and the Security Manager
- Code signing, stack inspection, and the new Java 2 security architecture
- The pros and cons of language-based enforcement models and trust models
- All known Java security holes and the attack applets that exploit them
- Techniques commonly used in malicious applets
- Twelve rules for developing more secure Java code, with explicit examples
- Hard questions to ask third-party Java security tools vendors
- Analysis of competing systems for mobile code, including ActiveX and JavaScript
- Card Java security, smart card risks, and their impact on e-commerce security

On the companion Web site www.securingjava.com you'll find:

- The Java Security Hotlist: Over 100 categorized and annotated Java security–related Web links
- An e-mail list to keep subscribers abreast of breaking Java security news
- A complete electronic edition of this book

GARY McGRAW is Vice President and Senior Research Scientist with Reliable Software Technologies and an international authority on Java security. Dr. McGraw is the author of over 50 peer-reviewed technical publications, consults with major e-commerce vendors including Visa, and is the principal investigator on several U.S. government research grants.

EDWARD W. FELTEN is Professor of Computer Science at Princeton University where he leads the world-renowned Secure Internet Programming team. Professor Felten discovered many of Java's security holes and is actively involved in designing more secure approaches to mobile code.

Wiley Computer Publishing **Timely. Practical. Reliable.**

Visit our Web site at **www.wiley.com/compbooks/**

7 23812 31952 7

Series Design: Howard Grossman

John Wiley & Sons, Inc.
Professional/Trade Division
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Weinheim
Brisbane • Singapore • Toronto

ISBN 0-471-31952-X
53499
9 780471 319528