JAVA SECURITY: PRESENT AND NEAR FUTURE

Li Gong

JavaSoft

The global system infrastructure is evolving rapidly. Java's security model must keep pace.

Since the inception of Java,[1,2] there has been strong and growing interest in Java's security as well as in new security issues raised by this technology's deployment. These concerns have reached the mainstream—the *New York Times*, the *Wall Street Journal*, and others have run stories on Java security. However, we must examine Java's trustworthiness comprehensively, from a broad dependability perspective,[3-6] not simply from the journalistic view.

A simplistic statement such as "Java is [or is not] secure" doesn't come close to capturing the complexity of the issues involved. To pass judgment on whether Java is a trustworthy language or platform, we must precisely specify requirements of the overall system—in this case, computer systems and the global Internet infrastructure. Only then can we hope to demonstrate vigorously that the platform architecture satisfies or contributes to the satisfaction of the overall requirements, and that the platform's actual implementation satisfies the architecture's design requirement.

Unfortunately, neither of these two crucial steps is feasible for security today. The shape and the characteristics of the national (or global) system infrastructure are evolving rapidly. Consequently, such widely used terms as "Internet security" cannot capture the essence of the trustworthiness required of such an infrastructure. (Admittedly, such terms were not very helpful in the past, either.) Moreover, though formal methods have been used successfully to verify some hardware devices, they are not sufficiently advanced to address correctness issues of large-scale software systems.

Given this state of affairs, we must use an alternative (albeit less scientific and idealistic) set of criteria to examine platform trustworthiness. From an industrial point of view, such criteria must include the following considerations:

- *Usability*—To be ubiquitous and accepted in the marketplace, the platform must be easy to use for building small- and large-scale systems and applications.
- *Simplicity*—To inspire confidence in its correctness, the platform cannot be too complex to analyze (by code inspection or testing, for example) for critical properties.
- *Adequacy*—The platform must contain all essential features and building blocks for supporting higher-level trustworthiness (security) requirements.
- *Adaptability*—The platform must evolve with ease, following demand and market reality.

Judged by these (perhaps biased) criteria, Java scores very favorably compared with other existing languages and programming systems. In particular, Java's platform independence greatly reduces the complexity—and thus the probability of design and implementation errors—of dealing with heterogeneous environments. The "write once, run everywhere" philosophy also helps improve availability.

Moreover, the encouragement to write code exclusively in Java (Sun's "100% pure Java" initiative) is highly desirable because it forces all security controls to go through the Java Virtual Machine. This raises the chances of putting the necessary protection mechanisms in place and getting them (almost) right. Other popular platform systems, such as MS-DOS/Windows, typically require native methods with direct access to raw system resources and devices. This makes retrofitting nonbypassable protection mechanisms with backward compatibility an almost insurmountable task.[7,8]

This article introduces Java's existing security features and the way they contribute to

0272-1732/97/$10.00 © 1997 IEEE

its overall usability, simplicity, adequacy, and adaptability in the global computing arena. It also discusses JavaSoft's plans to make new features available as the technology evolves.

The sandbox security model

The original security model that Java provides is known as the sandbox, a very restricted environment in which to run untrusted code (applets) obtained from the open network. The essence of the sandbox model is that a computer entrusts local code with full access to vital system resources (such as the file system). It does not, however, trust downloaded remote code, which can access only the limited resources provided inside the sandbox.

We have deployed the sandbox model through the Java Development Toolkit (JDK, the current version as of May 1997 is 1.1.2), and applications built with JDK generally adopt it.

JDK enforces overall security through a number of mechanisms. First, the language is designed to be type safe[1] and easy to use. The hope is that the burden on programmers is such that they are less likely to make subtle mistakes than they would using other programming languages such as C or C++.

Second, compilers and a bytecode verifier ensure that applications built with JDK execute only legitimate Java code.

Moreover, a class loader defines a local name space and ensures that an untrusted applet cannot interfere with the operation of other Java programs.

Finally, the Java Virtual Machine mediates access to crucial system resources, and a SecurityManager class checks and restricts untrusted code's actions to the bare minimum.

The actual implementation of the Java language and the runtime system is likely to contain bugs, at least for some time, just as any large-scale software system would. This is clearly manifested in the discovery of some well-known bugs, perhaps most prominently those found by the research team at Princeton University.[9]

Evolving the sandbox model

As Java gains popularity among users and developers, requests for various security features have been flowing in. Among these are support for enabling flexible and configurable security policies, support for cryptographic capabilities, security protocols for network-based authentication and secure communication, and easy-to-use administration tools.

Starting with JDK 1.1, JavaSoft will provide general Java users and developers with a series of security features meeting customer requirements.

Applet signing. A prominent new security feature in JDK 1.1 is built-in support for applet signing. Using this feature, applet writers can digitally sign applets using their signature keys. The person viewing the applet can use a known public key to verify that the applet is correctly signed and then choose to accept the applet as trusted. This will enable the trusted applet to run as if it were local code. JDK 1.1 contains the necessary tools for code signing, including support for digital signatures, message digest, and certificate manipulation.

New architectural directions. The design of an overall Java security architecture is constrained in a number of ways. It must be backward compatible, but also flexible enough to accommodate rapidly emerging technologies based on Java. It must function across a wide range of platforms. These include JavaOS, which is a Java-based operating system that runs directly on naked hardware, as well as native host systems such as Microsoft Windows and MacOS.

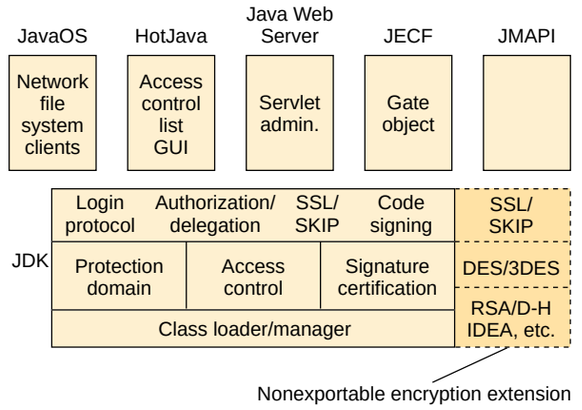

Figure 1. Overall architecture. JECF: Java Electronic Commerce Framework; JMAPI: Java Management API.

Figure 1 shows the relationships between various Java components.

It is natural that JDK should serve as the base platform for delivering security technologies to Java-based solutions. Future releases of JDK will deliver at least the following security features:

- Basic language safety features (type safety, bytecode verifier, class loader);
- Essential cryptographic primitives (MD5, SHA-1, DSA, X.509, DES, triple-DES);
- Fundamental system security features (protection domain, access control, authorization and delegation, security policy manager); and
- Necessary networking and Web-based security support (user authentication protocols, code signing, secure communication protocols such as SSL and SKIP).

Products built on top of JDK may need additional security features. For example, the Java Web Server (previously known as Jeeves) may provide an additional framework for securely administering servlets, while the HotJava browser may provide a browser-specific user interface for configuring security policies.

This technical architecture is potentially limited by business considerations such as licensing and exportability. The standard JDK will be fully exportable under US laws and thus cannot include strong encryption technology. The Java Cryptography Extensions will be the vehicle that makes unexportable technology available to US domestic users. JavaSoft will publish the API specification and documentation and make them available in accordance with the appropriate laws and regulations.

Not shown in Figure 1 is a policy and certification facility

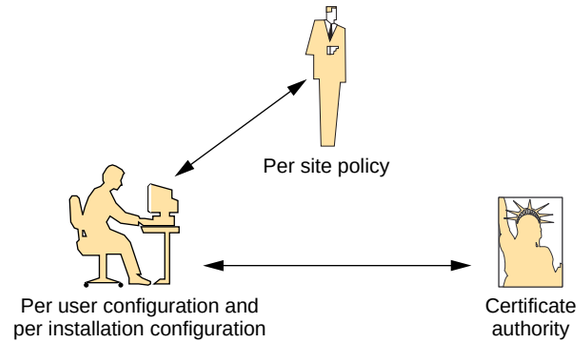

Figure 2. Policy configuration.

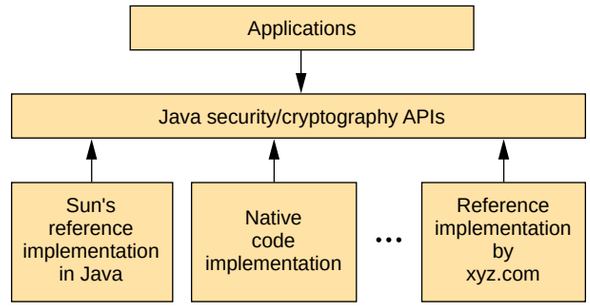

Figure 3. Cryptographic support allowing alternative providers.

that helps reduce the complexity of system administration. We envision that, as illustrated in Figure 2, a site security policy may override or be a supplement to the user-specific policy. A certification authority (CA) will assist in policy decisions.

Cryptographic library. We designed the cryptographic library architecture to allow different service providers to supply their own, compatible implementations of the same cryptographic functions. As illustrated in Figure 3, an application developer programs to a generic security/cryptography API obtained from JDK or its cryptographic extension package.

Depending on system installation (including the possibility of dynamic library downloading), the developer may have a choice among several providers. One of these is Sun's reference implementation in Java, which is typically shipped together with the high-level APIs. Developers can use this default or choose (by naming) a specific provider.

Such provider-conscious design is rare in software systems but is somewhat fashionable in cryptographic library designs. Allowing different providers has a few advantages. For example, the developer can plug in a native-code implementation to make use of an already resident cryptographic package or a hardware-based solution. Also, there are any number of reasons that a particular user may prefer one provider over another.

We emphasize that our design is open so that any cryptography provider can choose to implement to our APIs and make the implementation available commercially. We do not follow a controlled-provider approach and do not seek to certify or restrict providers on the basis of export control or other reasons.

Secure communication. Applications built with JDK can provide secure network-based communications at link level, as in Sun Microsystems' SKIP technology, or at session level, as in Netscape's SSL protocol (see Glossary). JavaSoft supports open standards that solve our customers' technical needs, but we can only operate within our business restrictions.

Glossary of cross-platform security

API	Application program interface, the calling conventions an application uses to access the operating system and other services
AWT	Abstract Windowing Toolkit, a package for building GUIs in Java
bytecode verifier	Mechanism to verify that Java language constraints are satisfied by the class bytecode
class loader	Mechanism to dynamically locate and resolve Java classes
code signing	Mechanism to ensure code authenticity by signing classes with digital signatures
DES	Data Encryption Standard, an encryption algorithm, currently a US federal information-processing standard
Diffie-Hellman	A key-agreement mechanism based on exponentiation modulo a prime number
DSA	Digital Signature Algorithm, produces digital signatures based on arithmetic operations; a federal and NIST standard
JDK	Java Development Kit
JVM	Java Virtual Machine
MD5	A message digest algorithm; for digital signature applications where a large message must be compressed securely before being signed; an Internet standard
SHA-1	Secure hash algorithm (revised 1994), developed by NIST
SKIP	Simple key management for Internet protocols
SSL	secure sockets layer, TCP/IP security protocol
TCP/IP	Transmission Control Protocol/Internet Protocol, communications protocols that connect Internet hosts
type safety	Mechanism to ensure that an object of one type cannot be mistakenly viewed as an incompatible type

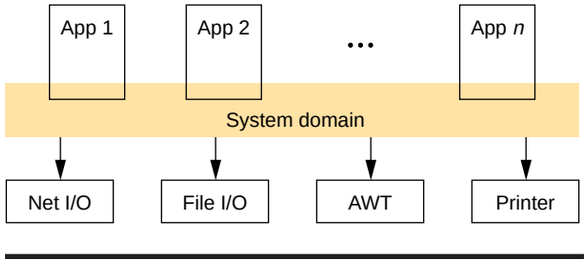

Figure 4. Java runtime domain architecture.

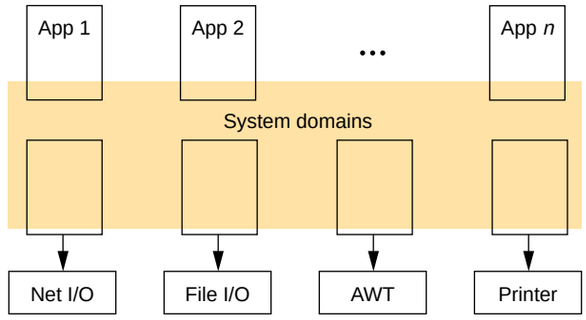

Figure 5. Separating system domains.

For example, SSL and SKIP support different "flavors," in that the user has a choice of which algorithm to use for key exchange and which to use for encrypting traffic. One flavor would be to use Diffie-Hellman for key exchange and triple-DES for traffic encryption.

However, some of these algorithms are proprietary, so it is practically impossible to include them in a toolkit such as JDK. This means that, as part of JDK, Java-based implementations of SSL and SKIP may be available in a limited number of flavors, while end-user products may bundle SSL and SKIP in fuller flavors.

Flexible security policy. The sandbox as it stands now is fixed and hard to change (and it is not meant to be changed lightly). To allow for flexible security policies, we must have mechanisms for supporting fine-grained authentication and authorization.

Java will be deployed on a wide range of platforms, where user authentication takes place in various forms—from simple password-based login to public-key technology with hardware support. Therefore, we must allow multiple ways to establish identity, but provide a uniform way to access such identity information. One benefit of this approach is that the Java runtime can support features such as single sign-on and interface with legacy systems.

Authorization at runtime depends on many factors, including the code base (whether it is local or remote, and which hosts), whether the code is signed and by whom, and whether we recognize and trust the signers. We envision specifying a policy in terms of sets of permissions, where any piece of code runs with a permissions set in effect.

The permissions format is typed in the sense that it allows for extension and customization. For example, the following are three permissions JavaSoft has specified to permit reading a file, connecting to a specified host at a particular port number, and exiting the Java runtime.

```
COM.JavaSoft File read path
COM.JavaSoft Net connect remoteIP:port
COM.JavaSoft Runtime allow system.exit
```

We expect that this format will evolve in the near future and that we will publish a full specification for consultation.

Protection domains. A fundamental concept and important building block of system security is called the protection domain.[8] A domain can be scoped by the set of objects that are currently directly accessible by a principal. A principal is an entity in the computer system to which the system grants authorizations (and, as a result, accountability). The Java sandbox is an example of a protection domain with a fixed boundary.

By default, the system must keep protection domains from interacting with each other. Any permitted interaction must be either through trusted system code or explicitly allowed by the domains concerned. Often a domain supports inheritance: a subdomain automatically inherits the parent domain's security attributes. In certain cases, however, the parent further restricts the subdomain explicitly. Relaxing a subdomain is also a possibility with the notion of trusted code.

Protection domains generally fall into two distinct categories: system and application. It is important that all protected external resources, such as the file system, the networking facility, and the screen and keyboard, are only accessible via system domains. Figure 4 shows the domain composition of a Java runtime.

Each domain (system or application) may also implement additional protection of its internal resources within its own domain boundary. Because it is unlikely that the JDK system can predict or enforce the semantics of such protection, the protection system at this level is best left to the application developers.

One example is a banking application, where the banking software can support and protect internal concepts such as a checking account, deposits, withdrawals, and so on. The developer for the banking application can use the protection domain primitive in JDK to run the application in a domain, thus insulating the application from other applications. The developer can also construct the system so that the handling of each separate account runs inside a distinct subdomain, thus ensuring that accounts cannot be accidentally mixed up.

For convenience, we can think of the system domain as a single, big collection of all system code. For better protection, though, system code can run in multiple system domains, where each domain protects a particular type of resource and has a special set of rights. For example, as shown in Figure 5, file system code and network system code can run in two separate domains, where neither has rights to the other's resources. With this arrangement, the risks and consequences of an error or security flaw in one system domain are more likely to be confined within its boundary.

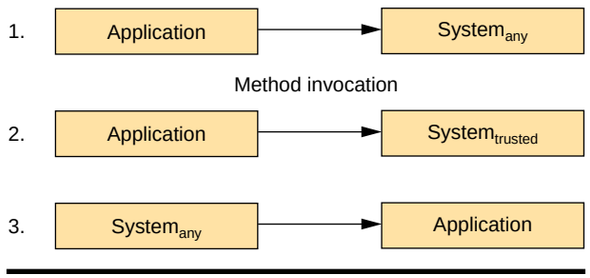

Figure 6. Execution involving two domains.

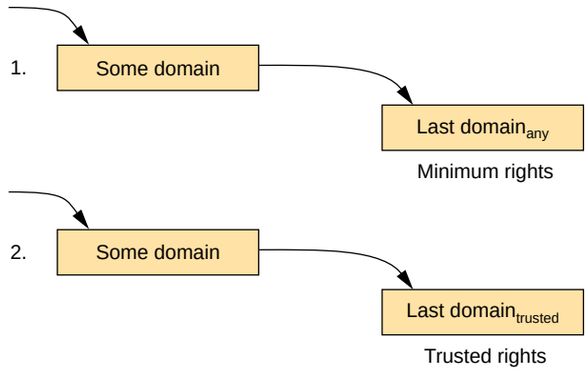

Figure 7. Execution involving multiple domains.

Multidomain computation. A thread of execution (not necessarily tied to a single thread) may occur completely within a single protection domain. Alternatively, it may involve an application domain and also the system domain, as shown in Figure 6.

For example, an application that prints a message must interact with the system domain, which is the only access point to an output stream. In this case, it is crucial that the application domain does not gain additional permissions at any time by calling the system domain. If it does, there can be security implications.

In the reverse situation, a system domain invokes a method from an application domain, such as when the AWT system domain calls an applet's paint method to display the applet. In these situations, it is again crucial that the effective access rights are the same as current rights enabled in the application domain at all times. The generalization is an execution thread that can also traverse multiple protection domains, as shown in Figure 7.

An exceptional case is when a piece of trusted system code can temporarily exercise more permissions than are available directly to the application. This is necessary in some cases. For example, a user cannot have direct access to the password file, but can call a trusted system utility that changes the password on the user's behalf. In this situation, the system utility must enter a special privileged mode.

The following are simple and prudent rules of thumb for calculating permissions:

- The permission of an execution thread is the intersection of the permissions of all protection domains traversed by the execution thread.
- When a piece of code in the execution thread has entered its privileged mode, the intersection of permission is applied only to protection domains between the privileged code and the end of the thread.

THIS ARTICLE HAS GIVEN A VERY BRIEF OVERVIEW of Java's basic security mechanisms—those existing, and some planned for the future. We have chosen to describe in some detail the design of the cryptographic library and the concept of protection domain,[10-12] but have not had room to elaborate on other topics such as password-protected authentication protocols,[13] access control and delegation,[14-16] and secure remote method invocation. We expect to report our design and implementation experiences in future articles.

Acknowledgments

Ideas from decades of research in computer security influenced the ideas presented here. It is unfortunately beyond our ability here to give a complete listing of original publications or to thank by name everyone from whom we have benefited in recent discussions.

This document is a purely technical discussion for the wider Java community, and does not necessarily commit JavaSoft to any particular features or implementations.

References

1. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification,* Addison-Wesley, Menlo Park, Calif., 1996.
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.
3. J.C. Laprie, ed., *Dependability: Basic Concepts and Terminology*, Dependable Computing and Fault-Tolerant Systems, Vol. 5, Springer-Verlag, New York, 1992.
4. R.M. Needham, "Denial of Service: An Example," *Comm. ACM*, Vol. 37, No. 11, Nov. 1994, pp. 42–46.
5. P.G. Neumann, *Computer-Related Risks,* Addison-Wesley, 1995.
6. "Information Security: Computer Attacks at Department of Defense Pose Increasing Risks," US General Accounting Office, Tech. Report GAO/AIMD-96-84, Washington, D.C., 1996.
7. M. Gasser, *Building a Secure Computer System,* Van Nostrand Reinhold Co., New York, 1988.
8. J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, Vol. 63, No. 9, Sept. 1975, pp. 1278–1308.
9. D. Dean, E.W. Felton, and D.S. Wallach, "Java Security: From HotJava to Netscape and Beyond," *Proc. IEEE Symp. Research in Security and Privacy*, IEEE, Piscataway, N.J., 1996, pp. 190–200.
10. M.D. Schroeder, *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*, doctoral dissertation, Dept. Electrical Eng., Massachusetts Institute of Technology, Cambridge, Mass., 1972.

11. A.K. Jones, *Protection in Programmed Systems,* doctoral dissertation, Dept. Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., 1973.
12. W.A. Wulf, R. Levin, and S.P. Harbison, *HYDRA/C.mmp–An Experimental Computer System*, McGraw-Hill, New York, 1981.
13. L. Gong et al., "Protecting Poorly Chosen Secrets from Guessing Attacks," *IEEE J. Selected Areas in Comm.*, Vol. 11, No. 5, Jun. 1993, pp. 648–656.
14. B.W. Lampson, "Protection," *Proc. Fifth Princeton Symp. Information Sciences and Systems*, Princeton University, Princeton, N.J., 1971. Reprinted in *ACM Operating Systems Review,* Vol. 8, No. 1, Jan. 1974, pp. 18–24.
15. J.H. Saltzer, "Protection and the Control of Information Sharing in Multics," *Comm. ACM*, Vol. 17, No. 7, Jul. 1974, pp. 388–402.
16. B. Lampson et al., "Authentication in Distributed Systems: Theory and Practice," *ACM Trans. Computer Systems*, Vol. 10, No. 4, Nov. 1992, pp. 265–310.

Li Gong joined JavaSoft from Stanford Research Institute as Java security architect. His research interests include distributed systems and networking, especially security and fault tolerance issues.

Gong received the PhD degree from the University of Cambridge, England, and the 1994 IEEE Communications Society Leonard G. Abraham award. He served as program cochair of the third and fourth ACM Conferences on Computer and Communications Security and as program chair of the seventh and eighth IEEE Computer Security Foundations Workshops. He is currently program cochair of the 1998 IEEE Symposium on Security and Privacy. He is an associate editor of *ACM Transactions on Information and System Security* and is on the editorial board of the *Journal of Computer Security*. He is a member of the IEEE and the ACM.

Direct questions concerning this article to Li Gong, JavaSoft, MS UCUP02-102, 2550 Garcia Ave., Mountain View, CA 94043-1100; gong@eng.sun.com.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 153 Medium 154 High 155