DOI:10.1145/2018396.2018411

Article development led by acmqueue
queue.acm.org

Difficult technical problems and tough business challenges.

BY LI GONG

Java Security Architecture Revisited

THE JAVA PLATFORM JDK 1.0 was released in 1995 with a simplistic all-or-nothing "sandbox" security model. Li Gong joined the JavaSoft division of Sun Microsystems in 1996 and led the redesign of the security architecture that was first released in 1998 as JDK 1.2 and is now deployed on numerous systems and devices from the desktop to enterprise and mobile versions of Java.

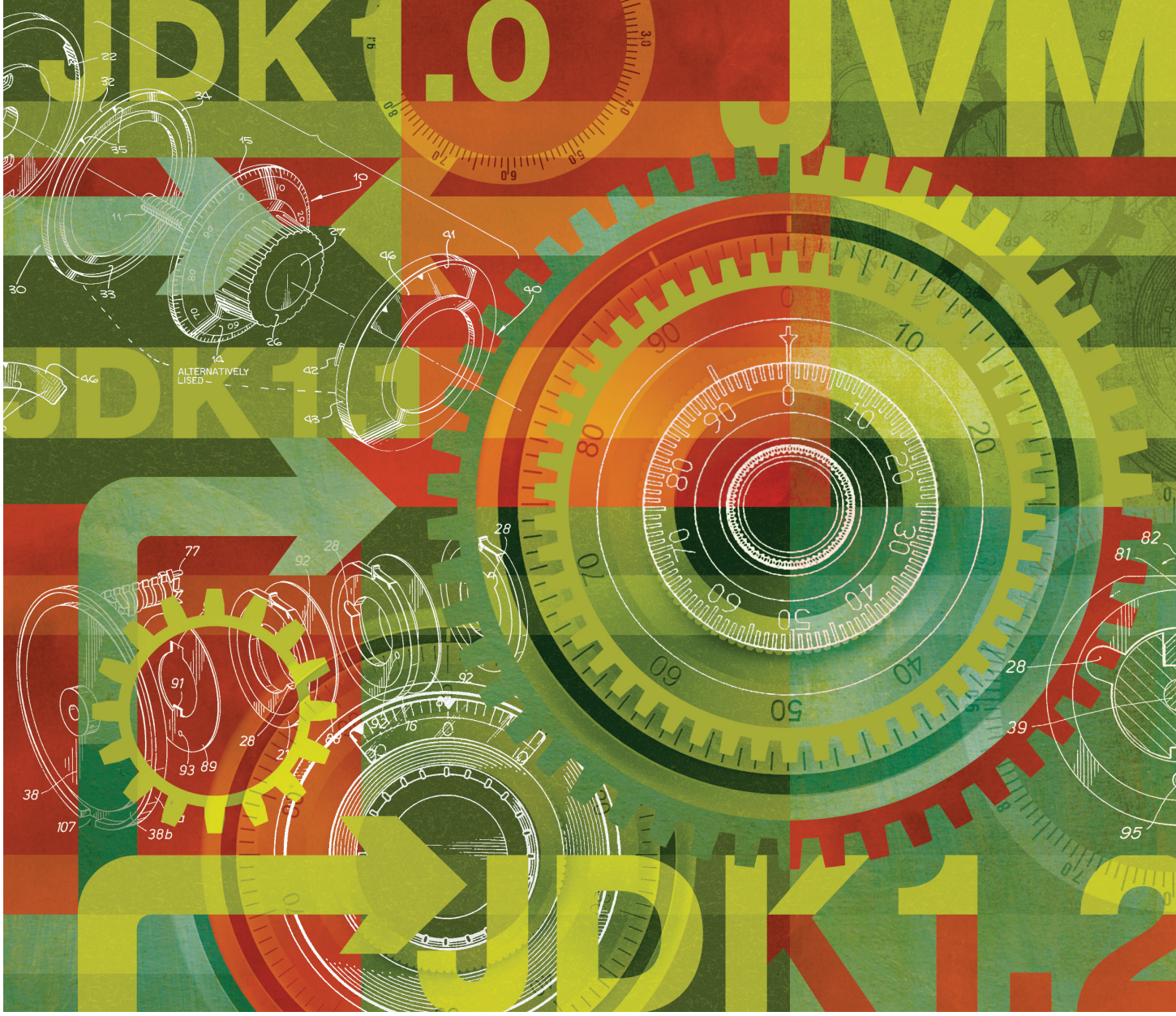
This article looks back at a few of the most difficult technical problems from a design and engineering perspective, as well as some tough business challenges for which research scientists are rarely trained. Li offers a retrospective here culled from four previous occasions when he had the opportunity to dig into old notes and refresh his memory: 2002 Workshop on the Economics of Information Security, Berkeley, CA; 2003 UW/MSR/CMU Summer Institute, Stevenson, Washington; ACM's 2009 Computer Security Applications Conference, Honolulu; and a seminar last May at the University of Cambridge Computer Laboratory, England.

Although security architects are not "in business," it is important that they are clear about who their customers are. They rarely build directly for individual end users, who do not directly use the operating system, although they are often the eventual beneficiaries.

Most of the work of a security architect is targeted at application programmers, and Java is no exception. Here the design goal is to help programmers get what is intended out of their code—more specifically, to make the most common cases the easiest to write and get right, and to reduce the risk of coding mistakes or bugs. As such, the four attributes of the Java security architecture[1] should generally apply:

- **Usability.** To be ubiquitous and accepted in the marketplace, the architecture must be easy to use and suitable for writing a wide variety of applications.
- **Simplicity.** To inspire confidence in the correctness of the architecture, it must be easy to understand the critical design properties and to analyze the implementation.
- **Adequacy.** The architecture must contain all essential features and building blocks for supporting higher-level security requirements.
- **Adaptability.** The design must evolve with ease, following demand and market reality. In particular, it should avoid overprescribing that restricts programmability.

In hindsight, having these guiding principles in place was crucial. In the original JDK 1.0, the security mechanism was all about special casing—code being inside versus outside the sandbox. That seemingly simple architecture paradoxically resulted in complicated design, fragile code, and numerous security bugs. In JDK 1.2, security was designed to be general, systematic, and simple-minded, and this resulted in a more robust and usable platform. We fought off a competing design from Netscape that was specialized for browser usage. Our design is not only broad in scope, covering desktops, servers, and embedded and mobile de-



vices, but also specific enough to enable programmers to build browser-centric applications. I will return to these topics later.

Guess Who's Coming to Dinner

The design aspiration is to ensure that Java code is executed as intended without undesirable side effects. This goal has three components. The first is to ensure that only valid Java code is accepted; this is the topic of the current section. The second is to ensure that intended behavior occurs as designed; this is usually taken care of via testing and is well understood, and therefore is not dealt with further here. The third is to prevent bad unintended behavior, such as access to critical data that should not have been allowed; this is dealt with later in the section on the principle of least privilege.

Yet another often-implicit requirement is that all the checks and balances must be done reasonably fast—meaning the system has performance characteristics comparable to that of a system with no security mechanism at all. The threat model here is focused primarily on untrusted code that might engage in malicious actions. The protection mechanism aims to stop those malicious behaviors; it also helps reduce risks of benign coding mistakes, although it cannot expect to protect against all faulty programming practices, such as not validating queries that might lead to SQL injection attacks.

Typically, an application is written in Java source code, which is compiled into platform-independent Java bytecode, which is then executed by the JVM (Java Virtual Machine). It is possible to compile Java source code directly into machine-specific native code (such as for x86 systems). This scenario is not discussed further here because compiled native code bypasses the Java mechanism and cannot be dealt with entirely within the Java platform except by way of allowing or disallowing native code access and execution. It is also possible to write Java bytecode directly, although most people choose not to practice this special art. In any event, even bytecode generated by compilers may not be trusted. In fact, Ken Thompson went much further in his well-known 1984 Turing Award lecture, "Reflections on Trusting Trust," by saying, "You can't trust code that you did not totally create yourself." Thus, the JVM must be able to decide if a piece of bytecode is valid and acceptable.

Each unit of Java bytecode (the opcode) is exactly one byte long and is well defined. A truthful compiler takes valid Java source code and produces a sequence of bytecode that accurately re-

ILLUSTRATION BY MATTHEW COOPER

flects the intentions of the source code and at the same time maintains the inherent properties of the Java language, such as type safety. A maliciously generated sequence of bytecode, on the other hand, may not correspond to any valid Java source code at all and can intentionally break language properties in order to enable security attacks.

Telling whether a presented series opcode is "valid" is fundamentally a form of the input validation problem. Suppose the JVM takes any integer in the range of 1 to 9 as valid input; then input validation is trivial. In reality, the input space that contains arbitrary sequences of opcode is unlimited in size and sparsely populated with valid bytecode sequences. Because there is no simple validation test formula through which to run a target code, the Java runtime system does bytecode validation in multiple stages, with various techniques, at different places. The bytecode verifier statically checks incoming code. Once the code is inside the system, type-safety mechanisms are deployed throughout the JVM to spot and stop illegitimate code. All these maneuvers are complex, and, in the absence of formal verification of the total system, there is no way to know for sure that all possible invalid codes can be spotted.

Thus, one really hard problem for any runtime system that deals with executable code compiled from high-level languages is to ensure that code received from "foreign" sources is valid input. For most programming languages, this is simply not possible. Java's platform-independent bytecode makes this task possible, but it is still extremely difficult to get right. Brian Bershad and his (then) student Emin Gun Sirer at the University of Washington came up with the concept of a bytecode basher.[4] They set up an automated system to generate random sequences of opcode to throw at any particular Java runtime system; they watch to see if the system breaks and then analyze the results to figure out the flaws in the Java implementation. This randomized and automated approach is a surprisingly low-cost yet effective tool that the JavaSoft team quickly adopted.

Who Moved My Cheese?

Now comes the problem of preventing bad unintended behavior. For example, when a Java application or applet triggers an access request for a local file, should the request be granted? Well, it depends. If the request is to read the local file containing personal credit card information, then the request may or may not be granted, depending on if there is a security policy or user preference in place. If the request is to read the font file for 12-point Calibri type so that a text file can be displayed according to the word processor specification, then it almost always should be granted, because, implicitly, font files are structured to be harmless if used in this fashion. Note that applications never directly open files. They call the file APIs in the Java platform for these operations. These APIs have no built-in notion of security, except that they (or their designers) know that file operations are sensitive, so they better make a call to the SecurityManager for consultation.

Here the problem rears its ugly head—the SecurityManager is put on the spot and has no clothes on. For example, it is quite alright for the display code written in the system to access the font file, but usually it is a bad idea for the application to gain direct access to system font files, because these files could be arbitrarily changed and that might lead to future display problems. The SecurityManager can hardly differentiate between these two scenarios, however, let alone the indefinite number of variations.

In JDK 1.0/1.1, the all-or-nothing sandbox security model works more or less as follows. The SecurityManager looks up the call history (of method invocations). If all code is local (that is, no remote-loaded and therefore untrusted applets), then the access request is granted. If an applet is present in the call chain, then the request is denied, unless the immediate caller to access the file is system code; *except* when that code should not be accessing font files but not when the applet code is in fact in a call-back situation; except when the system code that makes the call back (to the applet) should not really access font files, and so on and so on. Moreover, what about threads, exceptions, and other constructs that mix up or disconnect the execution context? You get the picture.

Fundamentally, trying to guess a program's intention is impractical. You would be much better off requiring programmers to declare their intentions explicitly, as we shall see later.

To make matters worse, the SecurityManager implementation does not actually run through this logical deduction—it cannot. Instead, based on some rules of thumb and a particular instance of the Java system, the SecurityManager simply counts the *distance*—the number of method calls—between itself and the nearest applet code, and heuristically makes a decision on whether a request should be granted. It should be obvious now that this setup means that whenever a part of the system is changed, the heuristics can become wrong and thus must be adjusted, regardless of whether the heuristics were correct or complete in the first place, and regardless of the method used to extend the system to include a new concept such as users/principals in making access-request decisions. This fragile setup was the source of many security holes.

Security in JDK 1.2 was rearchitected completely, adopting the well-known but almost never practiced principle of least privilege.[3] All code is treated equally. Each piece of code is given a set of privileges (access rights), either explicitly (through policy, administration, or preferences) or implicitly (where system code has full privileges, while applets have the same level of privilege as in their sandbox days). At any point in execution, an access request is granted if each piece of code along the call chain has sufficient privilege for the requested access. In other words, the effective set of privileges is the intersection of all privilege sets for the code along the call history—the principle of least privilege. Moreover, context information pertaining to security can be encapsulated and passed along so that one cannot fool the system by spawning new threads or throwing exceptions. All of these assume, of course, that the code that implements the security mechanism is itself secure and correct.

A piece of code—say, the display library that may need access to font files from time to time—can explicitly declare to exercise its own privilege unilaterally, telling the security system to ignore codes that have come before it. This design lets programmers, especially those who write system and library

code, to explicitly declare their intentions when performing sensitive operations. This is akin to the setuid feature in Unix except that, instead of enabling system-high privilege for the entire program, in Java the privileged mode lasts only as long as the duration of the privileged method call. Note that if this privileged code later calls less privileged code, the effective set of privileges will still be curtailed because of the principle of least privilege.

The need for explicit declaration may appear cumbersome at first, but programmers can rest assured that their code will not unintentionally weaken security. Furthermore, the majority of programs do not need to invoke their privileges specifically, so we have given the most common programming cases the best of both worlds—ease of coding and peace of mind. Note that it may not be easy for programmers to figure out exactly which of their privileges they need to declare in order to make their programs work properly in all possible scenarios. The JDK 1.2 design actually does not support fine-grained privilege declarations. A declaration enables all the privileges associated with the code.

The major lesson here is that being systematic is easier and more robust than being ad hoc, though not everyone understands this. Toward the end of JDK 1.2 development, during a security audit of the entire code base (which got enacted after much begging, plus sticks and carrots), we discovered that a Sun engineer working on JDK had deliberately duplicated and then modified system code such that his own library code would not have to bother with an explicit declaration of privileges—a move that may have made his job slightly easier but would have led to serious security breaches for all users of the Java platform if his misdeed had gone undetected.

A number of hard problems remain in this area. Top among them is whether least privilege calculations can be done efficiently, especially with very complex security parameters—for example, complicated access-control policies, many different types of access rights, and an intricate execution environment. Another problem is that assigning different privileges to different code created complexities for other parts of the system. For example, optimizations done by JIT (just-in-time) compilers must now conform to additional security requirements.

A piece of code can explicitly declare to exercise its own privilege unilaterally, telling the security system to ignore codes that have come before it. This design lets programmers to explicitly declare their intentions when performing sensitive operations.

Yet another perennial problem is the practical side of security policy management and deployment. A more theoretical question, but one nonetheless worth pondering, is the scope of security policies that can (or cannot) be enforced with the least-privileged model, using the rather conventional categories of access-control permission types defined in JDK 1.2. Fred Schneider of Cornell University developed an intriguing concept called Inline Reference Monitor and proved that it can express and enforce any and all policies enforceable by monitoring system execution.[5]

Despite these difficult issues, one comforting thought may be that, after more than 12 years in the field, the principle of least privilege as architected in JDK 1.2 has stood the test of time and probably saved untold numbers of coding mistakes from turning into security blunders.

The Importance of Being Earnest

Many other technical lessons are worth repeating periodically. For example, you should be very judicious about the use of NULL, because you cannot change the behavior of nothing. In JDK 1.0/1.1, in certain circumstances, the `ClassLoader` or `SecurityManager` could be `NULL`, which made it difficult to retrofit a more fine-grained design.

As another example, during runtime Java turns static code into live objects. This process actually contains two separate steps: locate the code description; define it into a live object. The first step should be open and extensible in nature, because both the runtime system and applications should be able to specify desired locations for obtaining code. The second step, on the other hand, must be strictly controlled so that only trusted system code can handle the job of creating objects. Unfortunately, these two steps were overloaded into a single method, which worked well in the all-or-nothing model but caused much difficulty when JDK 1.2 changed into a more nuanced world.

Another issue may be surprising to many people: strictly speaking, Java cannot guarantee sequential execution of consecutive instructions. One

simple reason is that exceptions can be thrown, causing the execution thread to detour (and may never return). One remedy is to use clauses such as `Try/Finally` to force a return. In more extreme cases—for example, when the actual physical machine runs out of memory—the behavior of the Java runtime system is undefined and certainly nowhere near being failsafe. These situations are further complicated by the fact that many key JVM functionalities, including some for security, are written in Java, so problems in one part of the system could easily impact the correctness of another part of the system. For all these design challenges and alternatives, please refer to the Java security book[2] and latest Java documentation.

The remainder of this article addresses the challenges that were entirely unexpected for someone whose previous work experience was confined to the world of academia. Scientists and engineers are trained to tackle technical problems, but real-world projects—especially one with industry-wide impact such as Java—are equally social and political. In roughly 30 months of working on JDK, I attended around 1,000 meetings and took 300-plus pages of notes. One can easily forget the war-zone atmosphere back then, especially the Friday fire drills. Too often, (outside) security researchers would inform us of newly discovered security holes on Friday and give us until Monday at noon to figure out a patch and response, when they would inform *The New York Times*, *Wall Street Journal*, and other media. Sometimes leaks to journalists occurred right after we rolled out patches to Java licensees (including IBM, Microsoft, Netscape, and many others), and we could only guess which of them had the motivation to publicize the security holes before patches were put in place.

Then there was a whole assortment of other equally time- and energy-consuming distractions, such as U.S. export control regulations on basic cryptography (since relaxed), patents on RSA and public-key technologies (since expired), and issues such as code obfuscation, Java for e-commerce and smart cards, and JavaOS.

To make sure we were on the right path, we invited a small number of academic and industry experts (including Jerome Saltzer of MIT and Michael Schroeder from DEC Systems Research Center, authors of the original principle of least privilege paper) and convened a formal Java Security Advisory Council, which provided regular reviews and valuable feedback as the rearchitecting progressed. We also received great advice from many sources, mainly academic researchers and industry partners—not all of which was solicited or friendly. A few strong-headed researchers wanted their alternative designs incorporated into the Java platform and threw various threats at us.

Netscape was a unique story. It was the most popular browser to include Java and therefore was a valued partner; it also had its own ideas about where Java should be headed, and those ambitions made the relationship difficult. On a technical level, the main dispute in the area of security was between Netscape's notion that Java was basically just a browser component so security mechanisms should be geared toward browser users, and our vision that Java was a general programming platform that should cater to all kinds of uses, including browsers and server-side applications.

On an engineering level, Netscape was innovating and shipping a new version every three months, while Sun/JavaSoft would take a year or two to ship a major release with new features (such as those requested by Netscape) that would become available through the official JDK platform. As the divergence between Java code in JDK and that in the Netscape browser was becoming unmanageably large, the presidents of Netscape and JavaSoft invited IBM to perform a confidential and binding arbitration. After months of intensive fact finding, code collecting, and *Consumer Reports*-style scoring, IBM called a resolution meeting at IBM's Java building, a block away from JavaSoft, on Oct. 15, 1997, and announced that JavaSoft's design had won.

Looking back after so many years, I can see at least three lasting effects of the Java security work. The most obvious is that the new security architecture provided better support for Java programmers to make their applications more secure and to reduce risks when the code was faulty. Second, we raised the bar for everyone else in the sense that any new language or platform must consider type safety, systems security, and the principle of least privilege, because we have demonstrated that these are achievable even in a large-scale commercial setting. Finally, the security constructs in Java have increased security awareness for thousands of developers who can then transfer this knowledge to other programming languages and development platforms.

Acknowledgments

I'd like to thank Jeannette Wing of Carnegie Mellon University, Jeremy Epstein and Peter Neumann of SRI International, and Ross Anderson and Robert Watson at the University of Cambridge for inviting me to give those retrospective talks on Java security. I am grateful to Robert Watson and Jim Maurer at ACM for encouraging me to write up the Cambridge talk for *Communications*, and to the thoughtful anonymous reviewers. I am, of course, deeply in debt to all the people who have cared for, helped with, and supported the Java security project.

Related articles on queue.acm.org

An Open Web Services Architecture
Stan Kleijnen, Srikanth Raju
http://queue.acm.org/detail.cfm?id=637961

How OSGi Changed My Life
Peter Kriens
http://queue.acm.org/detail.cfm?id=1348594

Untangling Enterprise Java
Chris Richardson
http://queue.acm.org/detail.cfm?id=1142045

References

1. Gong, L. Java security: present and near future. *IEEE Micro* (May 1997), 14–19.
2. Gong, L., Ellison, G. and Dageforde, M. *Inside Java 2 Platform Security: Architecture, API Design and Implementation*, second ed. Addison-Wesley, Reading, PA, 2003.
3. Saltzer, J. H. and Schroeder, M.D. The protection of information in computer systems. *Commun. ACM 17*, 7 (July 1974).
4. Sirer, E. and Bershad, B. Testing Java Virtual Machines. In *Proceedings of the International Conference on Software Testing and Review* (Nov. 1999).
5. Schneider, F.B. Enforceable security policies. *ACM Trans. Information and System Security* (Feb. 2000), 30–50.

Li Gong is chairman and CEO of Mozilla Online Ltd., the Beijing-based Mozilla subsidiary. He was formerly a Distinguished Engineer and the chief Java security architect at the JavaSoft division of Sun Microsystems. Two of his patents on Java security are among the seven that were the focus of a lawsuit between Oracle and Google over Android in 2010.