Java Security: A Ten Year Retrospective

Li Gong
Mozilla Online Ltd
Beijing, China
lgong@mozilla.com

Abstract— **The first edition of Java (both the language and the platform) was released in 1995, which contained the all-or-nothing security access model. A mid-1997 paper I published in IEEE Micro laid out a vision for the future of Java security, which notably included a model for fine-grained access control, a crypto architecture, and a number of other security mechanisms. The first implementation of these features was officially released in late 1998 as part of the JDK 1.2 platform. Ten years on, the original vision of Java security was largely realized and the overall architecture had in fact been carried over to both the enterprise Java and mobile Java platforms. This paper reflects on lessons -- technical and otherwise -- learned in the process of designing and implementing the Java security architecture and in the aftermath of its release into the real world**

Keywords-Java; security; least privilege; access control; safe programming; mobile code.

I. AN UNEXPECTED JOURNEY

In May of 1995, Sun Microsystems officially introduced Java, a language and a software platform that was to change the programming world completely. Initially designed for set-top boxes, it debuted in the form of the HotJava browser running on PCs and was soon integrated into the Netscape browser, making it accessible to millions of web creators and users. Java was described as objected-oriented, cross-platform, and secure. In fact it had so many marketing attributes attached that it was dubbed the most buzzword-complete technology.

However, labeling any system secure is at best imprecise and often plainly wrong, and is an unfailing public invitation for computer security researchers to attack it. Sure enough, security bugs and loopholes started flowing in, discovered by a wide range of people from individual "hackers" to specialized security research institutions, but most visibly a group of researchers at Princeton University led by Ed Felten. Given that Netscape was garnering such public attention and that the browser was the common conduit in which Java was deployed and utilized, stories of security breach in Netscape or Java (where sometimes it was not easy to differentiate between the two even to the experts) were headline news in the New York Times and the Wall Street Journal.

Later, when Microsoft licensed Java and incorporated it into the Internet Explorer browser, competition to deliver the most secure browser (factual or perceived) became an important battleground in the now well-documented David vs. Goliath browser war, with (seemingly weekly) security hole discoveries, mad scrambles to patch the problems, inevitable leaks to the press, and behind the scenes efforts to keep things quiet while solutions were being rolled out. Sun Microsystems was an eager participant in this fight, and I found myself at numerous industry conferences sitting on the same panel with Jim Roskind from Netscape and Barbara Fox from Microsoft, lobbing verbal missiles at and deflecting unflattering comments from the floor to each other. I quietly chuckled much later when I quoted George Bernard Shaw in my Java security book, who famously said "Security is the chief pretence of civilization." But I am getting ahead of myself here.

Soon after Java's first release, a more scholarly debate on runtime security (a term that was mostly focused on Java but also generalized to include any mobile code) raged on the ACM Risks Forum [1], the most popular discussion forum on such topics that was founded and moderated by Peter G. Neumann, who was an esteemed colleague of mine at the Computer Science Laboratory at SRI international. Sitting a few offices apart, one day we decided that it should be useful if we got all the major players into the same room to talk things out, and SRI was an ideal neutral ground. Invitations were sent out, and a one-day runtime security meeting was duly held at SRI on May 3, 1996.

The only surviving notes I am aware of (kept by Peter) listed participants including a Sun team led by James Gosling, a Netscape team headed by Taher ElGamal (then Chief Scientist for Security at Netscape), the Princeton gang (Felten, Drew Dean, and Dan Wallach), and highly-respected researchers Mike Schroeder (DEC SRC), Butler Lampson (Microsoft), and David Presotto (Bell Labs). Actual discussions are a hazy memory now, but I clearly remember Schroeder concluding at the end that if we were to put our heads together, we should be able to solve this problem.

After the meeting, Gosling and I talked a bit more, and on August 12 that year, I joined the JavaSoft division of Sun as its Java security architect with the mission to build out the security team there and to figure out what Java security was (and should be) all about. This was a rare chance to put one's research learning into practice, and Java seemed a pretty good place to jump in given it was the first commercial system in years that at least publicly and positively claimed security as a design goal. I took the plunge head first, although Java's future was anything but assured at the time.

II. EVOLUTION OF JAVA SECURITY

Java was released through the Java Development Kit (or JDK for short). JDK contains the Java Runtime Environment (JRE) that includes the Java Virtual Machine (JVM), as well as all the Java APIs and libraries. The initial release was JDK

1.0. Each subsequent major release, such as JDK 1.1 and JDK 1.2, included major new features, while minor releases such as JDK 1.1.2 and JDK 1.2.3 would include only bug fixes and minor upgrades.

As for security, JDK 1.0 has a clear-cut, binary model. Applications, defined as anything loaded from the local disk (or its equivalent), run as trusted code and can do anything the JVM can, which is everything. Applets, defined as code loaded over the network, run as untrusted and can do little. Applets were described as being contained in a safe sandbox, and thus the binary security model was often called the sandbox model.

JDK 1.1's main improvement over JDK 1.0 in terms of security support was the addition of a feature called code signing. An applet's code can be digitally signed. Upon loading, if the JRE recognizes the signatory and the signature checks out against the applet code, then the applet is treated as a trusted application and is given full access to the entire system. When I arrived on the scene in mid August of 1996, I was told that JDK 1.1 was going to be released at the end of that month and my immediate concern was to have everything related to JDK 1.1 wrapped up as quickly as possible. In reality, JDK 1.1 was not shipped until quite a few months later.

Design work for JDK 1.2 started in late 1996 and intensified once JDK 1.1 left the door. It was planned as a one-year project, more or less, but in practice it took a little more than two years of very hard work, finally shipping in December of 1998. At the engineering post mortem meeting on JDK 1.2, it was agreed that we made all those same mistakes we vowed to avoid at the JDK 1.1 post mortem meeting. JDK 1.2, which was renamed Java 2, saw the most significant changes to the Java platform in a single release, not just for security. JDK 1.2 introduced the fine-grained access control model, which continued essentially unchanged till this day. In order to implement the new security architecture, we needed to touch and tweak almost everything, because security is closely intertwined with the JVM and impacts every part of the JRE/JDK.

The new architecture designed for JDK 1.2 also introduced a full specification for cryptographic services, called the Java Cryptography Architecture (JCA). This set of services enables a crypto service provider -- anyone interested in providing its own cryptographic implementations for use by others – to plug in its custom implementations to a standard set of provider APIs. On the other hand, applications can call a standard set of crypto APIs for crypto services, and choose a specific provider if so desired. JCA was released as an optional component for JDK 1.2, called Java Cryptographic Extension (JCE), due to US export control restrictions at the time. Eventually JCA became part of the standard JDK release.

Separately, JSSE (Java Secure Socket Extension) was implemented to support SSL/TLS on the Java platform. Underlying all these was a full package for handling digital certificates. During that same period, a Java Authentication and Authorization Services (JAAS) framework was outlined. As its name suggests, JAAS introduces authentication and authorization – fundamental concepts for system security -- and enables a more sophisticated style of security policy management.

The first public musing about the directions of Java security was a talk I gave at IEEE COMPCON in San Jose in February of 1997 [2], and the content was expanded and later published in IEEE Micro in mid 1997 [3]. This is the designated "classic paper" for this conference. The first published paper on implementation of the JDK 1.2 architecture was presented at a USENIX symposium in Monterey at the end of that year [4]. The first edition of my book, "Inside Java 2 Platform Security" [5], which I wrote almost as a technical record to remind myself of what/how/why we did things in ways that often appeared peculiar to outside observers, was completed in time for release at JavaOne in June of 1999. Gary Ellison and Mary Dageforde later added new chapters and generally brought things up to date to produce the second edition in 2003 [6].

I left the JDK engineering group soon after the release of JDK 1.2, not unlike a good number of others after the super stressful two plus years on the project. I left Sun in 2005, right on the tenth anniversary of Java.

III. Criteria for a Good Foundation

One fundamental question facing us during JDK 1.2 was what sorts of security support are appropriate to build into a development platform. What are the critical rules that the system should require all programmers to abide by? Which set of features are often used such that they should be made available, even though their utilization is strictly optional? Which decisions should the system make, and which are best left to application developers?

In the "classic paper", four considerations are raised to determine the right platform security. I quote verbatim here:

> *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.

These four attributes highlight what were my primary concerns back then. On the one hand, fresh from academia, I was very much aware that many outstanding research results never see their light of day in practice because the work is too difficult to use or too hard to understand. On the other hand, given the visibility of Java and its security work, and knowing that it would take at least a year or more to add something in a later edition of the Java platform, I was constantly worried that we would miss a key ingredient that would make implementing a security feature (in applications) very difficult or impossible.

For example, I included SignedObject and SealedObject classes, to mimic authenticated and encrypted data packets, because these constructs are useful for security in distributed computing systems. The GuardedObject class

allows a programmer to encapsulate access conditions in a Guard object, and this is a more elegant way to express access control than directly calling the SecurityManager, which was/is the standard coding style inherited from JDK 1.0. The GuardedObject construct has a lot of potential but is perhaps one of the most neglected features introduced in JDK 1.2.

At the same time, I equally feared over-prescription. Once the JDK includes an API, it would be hard to remove, due to backward compatibility requirements. So I often told my team to think about forward compatibility and only design things that we were sure that we absolutely needed. In the same vein, I tried to leave everything as extensible and adaptable, to the maximum extent possible.

IV. IT'S NOT ALL TECHNICAL

The most fundamental question during the design phase of the JDK 1.2 security architecture was where Java was headed. Although Java had all the ingredients as a general programming platform, Java's early success was most visible through the integration into the Netscape browser. Should we design Java security so that it supports general applications? Or should we view Java as primarily a component of the browser?

It was not surprising that the Netscape team was pushing us towards features that they felt were important to a browser. In fact, the Netscape team already modified JRE's security implementation and designed their own security features (as Java extensions), blaming JavaSoft for not moving fast enough. This was partially justified, since the browser was being updated much more frequently than Java was.

Moreover, the Netscape team wanted us to adopt their extensions wholesale and make their security model the default for everyone. This request was met with strong resistance, since the browser specific features as proposed would not work if Java was going to be a general programming platform that might support applications for servers. For example, central to the Netscape model was a call-back feature where, when uncertainty arises, the user (who is presumably looking at the browser window on the PC screen) would be prompted to click a button to allow or disallow an action. This design is similar to what Windows Vista has recently implemented to alert users for critical actions, but it clearly would not work in an unmanned server environment.

This divergence of opinions escalated as time went on. Java security features inside Netscape's browser releases were becoming more and more different from where JDK 1.2 was heading. The longer the dispute dragged on, the harder it was to resolve the differences. This debate with Netscape was not a question of whose work is right or whose is wrong. No doubt the Netscape team knew the browser better and wanted to do the best for that. Rather, the key question is what to build into the platform, what should be done as (optional) extensions, and what should be left to application developers. To answer this question requires betting the future direction of Java. We obviously thought that the browser, albeit very important, was just one of the many types of applications Java should support, and browser-specific security support should not override or replace general systems security support.

In order to build better relationships, Jim Roskind and I were told (by our respective management) to visit each other's office on alternate weeks, which we did. But we could not patch up the work difference. Then one day, very late into the JDK 1.2 project, news came that Jim Barksdale (CEO of Netscape) and Alan Baratz (President of JavaSoft) made a deal in a New York hotel room that we would submit this dispute to arbitration, by IBM. IBM quickly assembled a task force led by Bob Blakley and consisted of security experts. They asked for and received design documents, code, test cases -- everything that would help them evaluate the two different systems. After a few months of high suspense, the Sun and Netscape teams were called into a conference room at IBM's Java building in Cupertino, a block away from JavaSoft. Bob led off their presentation and rated the systems à la Consumer Reports amid a ton of charts and stars. Then he simply announced that the JavaSoft's design won. And that was the abrupt end of this exhausting conflict.

V. SAFE CODING

David Wagner at UC Berkeley recently asked me how extensively the fine-grained access control mechanism, which is central to the JDK 1.2 security architecture, is used today. I have not done a survey and thus have no answer. However, I should emphasize that in addition to providing an implementation of the principle of least privilege, another important goal of this architecture was to make safe code easier to write.

For example, with a seemingly trivial binary security model, the implementation in JDK 1.0/1.1 was very fragile. Consider the case of a file system API being called to open a file. According to the model, if this request originated from an applet, it should be denied. Otherwise, it is allowed. One problem was how to differentiate the two situations.

Without a generalized security architecture in place, the JRE was reduced to counting how many call frames there are on a particular stretch of the system execution stack at that moment, trying to determine if there is an applet on the call stack. It was figured out that, in most cases, if the stack count is 2 (or maybe 3, depending on a number of factors), then the call must have come from an applet and therefore must be disallowed. There were exceptions to this rule, and the whole system was patched together through trial and error. Any change to the system, such as adding a layer of indirection by adding a new class, may change the system behavior in an undesirable way. Adding a new package to the system would require a fresh look at all the existing assumptions. Thus it is not difficult to see that basing security decisions on semi-blind stack counting is not the best idea and why it was the source of a number of security bugs.

JDK 1.2 introduced a uniform way to make access control decisions, based on the classic concept of protection domains and the well-known but rarely implemented principle of least privilege. It completely removes

guesswork in security enforcement. A critical action is allowed if and only if all individual classes in the calling chain have privileges for the action.

In conjunction with the principle of least privilege, JDK 1.2 introduced a "trusted" mode, where a programmer can specifically invoke all the privileges the code has, rather like the setuid feature in Unix. This is very useful for writing system code. However, unlike UNIX where setuid is turned on for an entire program, in Java the special privilege lasts only during that one part of the code. Moreover, programmers who have not used this special feature in their code can rest assured that the code will not accidentally cause undue harm, since it is prevented from violating the least privilege principle.

For whatever reason, my Java security book did not go into any of these details on why the JDK system security was so fragile in earlier versions. I guess it was considered sensitive information at the time.

VI. Archimedes's Fulcrum

An often repeated adage is that security for a system must be designed in; otherwise it would be almost impossible to retrofit. We were lucky with Java. Thanks to the foresight of the initial Java designers, they defined the SecurityManager class and made it clear that classes controlling access to critical resources should check with the SecurityManager for a decision to either grant to deny access. It does not matter that in JDK 1.2 we completely rewrote the SecurityManager class. It also does not matter that we practically changed everything related to security from what was originally released. What matters is that the early designers made it acceptable, and indeed necessary, to query the SecurityManager at critical junctures.

I believe that this acceptance singled-handedly enabled us to retrofit a much more stylized security architecture without causing too many troubles at the code level and the culture level. The continued use of SecurityManager minimized backward compatibility problems. The request that all classes in JDK must consider its security implications was easier to accept because security was there in the first place. This is why I chose to quote Archimedes in the preface of my Java security book – "Give me a lever and a fulcrum, and I can move the globe." SecurityManager was that fulcrum.

Nevertheless, retrofitting was not easy. For example, one technique used by JIT (just-in-time) compilers to speed up code execution was to try to combine a consecutive number of execution instructions into a single instruction. Under the JDK 1.2 security model, each instruction might come from a different piece of code, associated with a unique set of privileges. The JVM folks had to do things differently so that security properties are not broken during code optimizations.

Here is one more story, on the lighter side. Remember that the principle of least privilege effectively lowers the privilege of systems code so that, when they need to exercise their special power, the programmer needs to explicitly enable the special power for that part of the program. This principle broke a bunch of existing code – code that was perhaps dangerously exercising too much privilege. Instead of dealing with the new security architecture, an engineer at JavaSoft duplicated system classes but modified them to bypass security checks, for the sole purpose to let his existing code continue to function as before. Fortunately, this misdeed was caught during our internal security code audit.

VII. Second Thoughts and Afterthought

I am sometimes asked if I have any regrets or second thoughts. Would I have done things differently with hindsight? Did we do too little, too much, or just about right?

In retrospective, it is truly amazing how far we managed to go. My final job interview at JavaSoft (with a very senior figure) was centered on the question of whether security for the Java platform was merely about finding bugs and providing fixes; and whether the whole thing would "blow over" in a few months. However, once inside JavaSoft, my team was given much green light and priority, and was granted a lot of freedom (e.g., to require changes to various parts of the system, to delay releases until all security tests were passed) that would be rare in other places.

Broadly speaking, our first major achievement was that we largely "settled" the Java security debate, in the sense that we mapped out the landscape (of security for such a complex language plus development platform), developed a general security model, and produced an implementation of the least-privilege principle. Of course, security work in general and Java security in particular continue to this day; but the discussion has matured.

The second major achievement is that we raised the bar for everybody else. Security for any new system is now required, not optional.

The third major achievement is that we made a strong impact, with regard to secure programming, on not only those engineers (at Sun and elsewhere) who were involved in the development of the Java platform itself, but also the very large number of Java programmers. These developers in turn bring their security awareness and influence into their next projects.

Looking back, we probably did all we could during that relatively short period of time, under all the constraints we had. Nonetheless, there were a few ideas that I would have loved to investigate a bit more. For example, it would have been much cleaner for the programmer if the privileged operation was specified via a new method modifier "privileged"; however, such a language extension would have required major changes to the system and compilers, involving not only Sun but also many other parties.

As another example, GuardedObject potentially can be used to express security requirements more elegantly and accurately, especially when objects are accessed in a context (or system) that is different from the one in which the objects were created. However, we simply had to continue with the SecurityManager centric design because it was both backward compatible with JDK 1.1 and a better understood solution.

In late 1990s, researchers came up with many interesting ideas to approach security from fresh angles. For example, Fred Schneider, a key member of the Java Security Advisory Council, together with his PhD student at Cornell, Ulfar Erlingsson, proposed Inline Reference Monitors, which promised not only a mechanism to completely separate security policy from enforcement (via bytecode rewriting) but also a theoretical proof that the solution was extremely expressive – it is able to encode all enforceable policies. Despite the very attractive attributes of those ideas, we simply could not afford to try going down those paths, timing being just one of the many obstacles.

JDK 1.2 turned out to be the single shot we got to overhaul security for the Java platform.

VIII. Never Forget Class Struggle

I was very fortunate to have had the opportunity to participate in the shaping of the Java platform, which has succeeded beyond expectations to become a major force in the software world. In this effort, I have had tremendous help and advice from friends and colleagues too numerous to name them all in this paper. Any and all omissions of names here are not intentional. I have chosen to recall a few stories along this journey, which I hope are interesting to the readers.

As time flies by, memories start to fade. And I did not have the time to double check events with all the participants. I stand corrected, and apologize in advance, for any inaccuracies or errors. A more comprehensive bibliographical reference to prior work that influenced the Java security architecture can be found in my Java security book.

So what is Java security, after all? Well, the answers are in the book, but probably not where you would look for them. The book cover, which I designed to depict the Duke wielding a sword standing on the Great Wall of China, was meant to be ironic (especially if you can read the Chinese slogans on the two flags) – the Wall was built to defend against barbarians but the invaders simply broke through elsewhere. My favorite quote in the book is Mao Ze-Dong's well-known utterance, "Never Forget Class Struggle," which is basically what Java security is all about.

IX. Acknowledgements

Jeremy Epstein invited me to write this retrospect and helped me with editorial matters. Peter Neumann tracked down old meeting notes and provided detailed comments on a draft of this paper. Drew Dean reminded me of some of the early Java security bugs. I am very grateful for their time and effort.

References

[1] The ACM RISKS Forum. Moderated by P.G. Neumann. Online at www.risks.org.

[2] L. Gong, "New Security Architectural Directions for Java (Extended Abstract)". In Proceedings of IEEE COMPCON, San Jose, California, February, 1997, pp.97--102.

[3] L. Gong. Java Security: Present and Near Future., IEEE Micro, 17(3):14--19, May/June 1997.

[4] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers., "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2". In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, December, 1997, pp.103--112.

[5] L. Gong. Inside Java 2 Platform Security, First Edition, Addison Wesley, 1999.

[6] L. Gong, G. Ellison, and M. Dageforde. Inside Java 2 Platform Security, Second Edition, Addison Wesley, 2003.