Introduction: Java for Smart Cards

Java is only six years old. Yet in its short lifetime, much about Java has changed, especially from a security perspective. When Java was first introduced in 1995, security was high on its list of “features.” Since that time, the hype surrounding Java has lost its initial dazzle and the world has learned a great deal about Java security, both good and bad.

This white paper is not about Java or Java security in general, but about an emerging branch of the Java tree; Java for smart cards. Smart cards are an important enabling technology for secure e-commerce, and Java can help make smart cards more accessible to developers and business people by providing a well-understood, familiar environment that includes a certain amount of built-in security. Javacards, as they are called, are an integral building block in many systems currently in service as well as on the drawing board.

Cigital's Software Quality Management experts worked closely with Visa International to identify, understand and mitigate the security risks in the company's javacards.

Obstacles to Smart Cards

One obstacle hindering the widespread use of smart cards in the U.S. has been the large number of incompatible and often obscure development languages available for writing smart card applications. Historically, programming languages for smart cards have amounted to special-purpose assembly languages. Since only a few developers were familiar with card-application languages, only a handful of people could develop smart card code. Sun's Java Card specification addresses this obstacle head-on by creating a smart-card platform based on a standard programming platform well understood by tens of thousands of knowledgeable developers.

Another obstacle slowing U.S. smart card adoption is the historical “one-application-per-card” paradigm, in which a number of different smart cards must be used to carry out different functions. For example, one smart card might be used for authentication at a company workstation while another card altogether must be used in company vending machines. This involves obvious wallet scalability problems (who wants to carry around 20 smart cards, let alone have to fish out the right one for a particular function?).

As the leader among a handful of multi-application platforms, Java solves the one-application-per-card problem by allowing multiple applications – potentially written by different organizations – to exist on the same card. As a result, one card could be used for many different applications, including network access, online purchasing, electronic cash, and physical identification and authentication. From the perspective of a payment consortium such as Visa International, multi-application cards simplify cardholders’ lives. In a consumer environment, a single smart card could be used to record loyalty points for airline travel, hotel stays and car rentals. Likewise, a college student could use a smart card both as an electronic dorm room key and as a debit card at campus vending machines. In a business environment, smart cards could be used as authentication tokens for intranet access, control keys for physical access to buildings, and identifiers for corporate discounts when used on travel. Clearly, the possibilities are limitless.

In contrast to their European counterparts, U.S. businesses and consumers have offered a lukewarm reception to the concept of smart cards. A notable exception to this rule is smart card use in satellite TV systems. While single-purpose smart card applications have slowly grown in popularity, the concept of a multi-purpose smart card has yet to take hold. Java Card promises to reverse that trend and increase smart card adoption, for several reasons:

The information in this document was adapted from an article that appeared at pp. 42-52 of the March 2001 issue of Information Security magazine (www.infosecuritymag.com). It is reprinted here with their permission.

1 Throughout this document, the term “javacard” is used to denote Java-based smart cards. The term “Java Card” is used to denote Sun Microsystems’ Java Card specification.
1. **Mature technology.** Early versions (pre-2.0) of Java Card were released as proof-of-concept engineering systems. The current specification – version 2.1.1 – involves technology that is significantly more mature than that of early prototypes. As a result, businesses are beginning to converge on Java Card as a multi-application solution. Initiatives such as Global Platform’s Open Platform library (developed originally by Visa) exist to enhance and expand Java Card’s base functionality and security in particular vertical markets.

2. **Development.** A number of very big players – including Visa International, Sun Microsystems, Microsoft, MasterCard and American Express – have championed multi-application smart cards.

3. **Security.** In order for multi-application smart cards to take off, consumers must feel secure in placing “all their eggs in one basket.” Considering Java’s (somewhat unjustified) spotty reputation in security circles, this may be Java Card’s largest obstacle, particularly in tech-savvy environments. However, as Java Card’s underlying technology improves, so does its security.

### Java Security: An Oxymoron?

While this brief document does not dwell on the finer points of Java security, understanding the security implications of Java Card first requires a quick review of the Java security model. The Java tree is crowded with diverse and divergent virtual machines (VMs) that execute Java’s architecturally neutral byte code. Because Java byte code runs on the Java VM (JVM), it is possible to run Java code on any platform to which the JVM has been ported. JVMs can be found on most PCs (Web browsers such as Netscape Navigator and Microsoft Internet Explorer include a built-in version of the JVM), Java-based middleware servers, embedded systems such as mobile phones (using the kVM), and smart cards (which, as we will see, run a completely stripped-down JVM).

Every new version of the JVM is faced with the same classic security tradeoff: security versus functionality. The premise of this tradeoff is that the greater a system’s functionality, the less secure it is likely to be. Though negative press may lead one to believe otherwise, Java has done an admirable job balancing the two sides of this tradeoff.

As of this writing, Java is by far the most sophisticated language-based security model in widespread use. In general terms, Java keeps security in mind at all times – a claim that languages such as C and C++ cannot make – adding as much functionality as possible while managing the very real security risks of mobile code. This strategy remains constant even as Java changes from the black-and-white sandbox model of Java Development Kit (JDK) 1.0.2 and JDK 1.1 to the shades-of-gray policy-driven model of JDK 1.2 (a.k.a. Java 2) (see Figure 1).

---

2 Note that Microsoft has jumped into the language-based security fray with its .NET initiative, which promises to foment competition for Java. Similar to Java’s virtual machine, the .NET security model attempts to encapsulate dangerous OS functionality and enable managed mobile code.
Java security is all about managing trust. The applet/application distinction, much touted in the early JDK 1.0.2 security days, is actually misleading. A distinction is more correctly drawn between Untrusted and Trusted code. Java’s evolution can be shown in terms of this distinction as moving from the early black-and-white fully trusted vs. fully untrusted model to the shades-of-gray policy-driven model of Java 2.

At its most fundamental level, Java security is about protecting software objects and system resources from malicious activity by encapsulating potentially dangerous functionality with a layer of security. For example, in JDK 1.0.2, the sandbox disallows untrusted code in the form of an applet from opening files or replacing critical Java classes. This process is accomplished by monitoring those calls that might be abused by an attacker, and disallowing dangerous activities from occurring in certain situations. For example, a poorly constructed hostile applet might attempt to read the system password file and send it over the network to a listening attacker who could then run it through a standard password cracker. Java’s security model protects the file system from this sort of malicious mobile code by disallowing the necessary file I/O calls.

The JVM is the interface through which all Java code must execute. Calling directly into the base Operating System is not allowed, except by the JVM. The security model is built directly into the JVM, which must distinguish between calls coming from trusted and untrusted code.

The JDK 1.0.2 security model is comprised of three major parts:

1. **The Byte Code Verifier**, which helps enforce critical “type safety” rules (without which a language-based security model cannot exist). Type safety is a programming language construct that provides the cornerstone of language-based encapsulation.
2. **The Class-Loader System**, which loads classes from the network and disk, and manages name spaces to keep code from different places separated. The Class Loader is responsible for identifying the origin of mobile code.
3. **The Security Manager**, which keeps tabs on dangerous method calls (e.g., the I/O calls mentioned above). The Security Manager is, in some sense, a classic reference monitor.
JDK 1.1 added cryptographic primitives to Java, largely to implement a code-signing model. The idea was to enhance Java so that code that was originally untrusted (applets) could become fully trusted by virtue of carrying a trusted digital signature.

JDK 1.2 (Java 2) significantly enhances the security model by adding stack-based access control, which is fine-grained, expressive and extensible. The main problem with Java 2 – one that has significantly slowed the adoption of Java 2 – is that the model must be configured and managed by a complicated policy.

Java Card 2.1.1

Java Card is a stripped down, miniature version of Java. Although Java Card is relatively new, it has been around long enough to go through several changes. The majority of these changes have been brought about in the name of security.

One advantage of Java Card is that the Java programming language is popular among hundreds of thousands of developers worldwide. Though Java Card certainly differs in important ways from standard Java, there are enough similarities that developers can make the transition with ease. In fact, in its current form, Java Card makes use of a standard Java compiler (creating standard Java byte code) as one of the development steps. As we will later explain more fully, standard Java byte code is transformed into Java Card byte code as a final step prior to being loaded onto a card.

Fitting a complex and large system like Java and its supporting class files onto a small smart card is not possible. Though some smart cards are quite powerful, the majority of smart cards in use today operate with the same computational power and resources of an IBM XT platform from the early ‘80s. As a result, Java Card supports many but not all standard Java features (see Table 1).

Table 1: Java Card – To Have and Have Not

<table>
<thead>
<tr>
<th>Supported Java Features</th>
<th>Unsupported Java Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
<td>Dynamic class loading</td>
</tr>
<tr>
<td>Dynamic object creation</td>
<td>Security manager</td>
</tr>
<tr>
<td>Virtual methods</td>
<td>Threading</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Object cloning</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Garbage collection/Large data types</td>
</tr>
</tbody>
</table>

Taken together, the differences between standard Java and Java Card have a large and direct impact on security. Certain differences enhance security, while others detract from it. The differences that simplify Java – and thus remove risk – include the following:

- **No dynamic class loading.** Type safety is critical for (though not equivalent to) a language-based security model such as Java’s. Java’s class-loading system seriously complicates type safety because code that has yet to be loaded cannot be checked using static verification. Name space management issues are also much easier to handle without dynamic class loading.
- **Only one active applet.** Inter-applet security issues can be tricky, especially when a multi-threaded system is involved. Java Card avoids this complication.

- **No threading.** Protecting threads from each other and worrying about conditions such as lockout or deadlock are not a large risk factor in Java Card.

- **Objects include rudimentary access control.** With a simple model comes less risk. The complex stack-inspection-based access control model of Java 2 is difficult to achieve properly. By contrast, Java Card’s simple model is much easier to implement correctly.

On the other hand, certain differences between standard Java and Java Card negatively impact security:

- **No applet sandbox.** One of the central security concepts of the original Java security model is that possibly dangerous (untrusted) code can safely run on a VM, which ensures security with the traditional Java sandbox security model. However, since Java Card has no sandbox, any Java Card application **must** behave itself from a security perspective. Malicious code can and will wreak havoc on a card. This is a critical point to understand, because many people assume that Java Card (since it is Java) has a sandbox. In the final analysis, Java Card has a very simple access-control model based on a standard unique identifier assigned to each applet (the AID).

- **Applets added to a card post-issuance.** This problem is exacerbated by the fact that applets may be added to a Java-based smart card after the card has been issued (for example, by a bank). The idea is that a smart-card user might download a new applet onto a Java Card over the Web using an ATM or a PC-based smart-card reader as an interface. This is highly problematic from a security perspective. The problem is that to ensure the security of the rest of the card, the new code must be benign and well behaved.

- **Requirement for trusted code.** The lack of a real applet sandbox in Java Card means that its code must be trusted. Put another way, Java Card applets **must** behave. Malicious Java Card applets present a real risk that must be countered with policies and procedures controlling the applet loading process. Careful design and testing of Java Card applets (including security issues) should be a mandatory part of the development process.

- **Java Card’s Verifier is an “out-of-band” process.** In standard Java, byte code is verified as it is loaded into the Java Runtime Environment. This is the main method by which type safety principles are enforced. By contrast, the Java Card model includes a “CAP file converter,” which is applied after (standard) byte code is verified but **not necessarily on the card itself.** The upshot is that the CAP file conversion process must preserve type safety. Furthermore, all verified Java Card byte code should be cryptographically protected as it awaits loading onto a card.

- **Availability of native method calls.** Over the years, smart card chip vendors have designed cryptographic co-processors for use on their smart cards. Because the vendors want to make use of this and other related special functionality, they argue that Java Card applets should have access to the functionality through “native methods.” Simply put, native methods completely break language-based encapsulation models and should not be present in Java implementations of any sort. By avoiding the use of native methods, smart card vendors can avoid this problem.

- **No garbage collection.** Standard Java makes use of a modern memory management technology (introduced by LISP in 1959) that includes garbage collection. The idea is to collect and re-use areas of memory where objects that are no longer referenced have been stored. Java’s garbage collection scheme is complex and too large to fit onto a standard smart card. As a result, Java Card has no garbage collection. This means that Java Card applets are susceptible to memory leaks and memory-based denial-of-service problem.

- **Object-sharing complexity.** This is possibly the most re-designed and reworked area in Java Card. Object sharing allows different applets to access the same structures in the memory of a card. This enables the memory to act as a communication band and a way for one applet to directly affect another. Early versions of Java Card introduced large security risks through a somewhat broken object-sharing scheme. Transitivity, in particular, was a major problem. That is, class A could share a method with class B, which could then go on to share the method with class C. The problem is, class A may not have wanted to share with class C. This problem was addressed in a redesign (and is no longer present in Java Card 2.1.1), but the resulting system is very complicated and difficult to use. Vendor implementations of the object-sharing systems are at high risk of error.
Risk Management for Smart Cards
As with any technology used in security-critical situations, smart cards require serious attention to risk management issues. There is no such thing as 100 percent security, even when smart cards are involved. Advanced security architectures such as the security architecture of Java 2 and Microsoft’s .NET Framework provide a number of useful tools for future security designs. However, such approaches must be seriously scaled-down in order to fit onto a smart card.

Since Java Card 2.1.1 removes a majority of the Java sandbox, smart-card applets are left with plenty of room to misbehave. That makes securing a particular smart-card system a non-trivial exercise. The most striking area of risk involves the fact that smart-card applets must behave in order for the system to remain secure. Regardless of the rigor of a Java Card implementation, applet testing is an essential piece of the security puzzle.

Open Platform Security Enhancements
Over the last several years, Visa International has influenced the development of Java Card with the goal of creating the next generation e-commerce platform. The Open Platform library and card-management system – now owned and administered by GlobalPlatform – was created to provide convenience and peace of mind to smart-card users. GlobalPlatform (www.globalplatform.org) is a cross-industry enterprise that develops technical standards to advance smart-card growth. More than 35 companies are GlobalPlatform members, including those in the financial services, healthcare, telecommunications, retail, government, travel and entertainment industries. Open Platform enhances Java Card with essential vertical-market features (including compliance to international smart-card standards, such as ISO 7816), while at the same time addressing the major security risks found in Java Card.

Each of the major risks identified above is managed in some way by the Open Platform system: The first three risks – no applet sandbox, applets added to a card post-issuance, and requirement for trusted code – are addressed by placing strict controls on who may add code to a card, and under what conditions. By allowing only those applets that have been carefully scrutinized and approved onto an Open Platform card, the very real risks that come with allowing additional mobile code to be added to a card can be appropriately managed. Trusted applets have no need for an elaborate applet sandbox.

Open Platform enforces its strict controls over applet loading with cryptographic controls. Different vendors can be provided with different “security domains” on a single card, thus clearly separating security policy and its enforcement among different concerns. Applet-loading controls help manage the fourth security risk – Java Card’s Verifier is an out-of-band process – by cryptographically protecting a Java Card applet after CAP-file conversion and prior to loading. (Type-safety issues, however, remain problematic.)

The remaining risks – availability of native method calls, no garbage collection, and object-sharing complexity – must be addressed by extensive applet testing. Such testing must address security requirements in addition to more standard functional requirements.

Java Card Security
Cigital and Visa have worked hard to ensure that the security risks of Java Card are well understood and properly managed. The approach begins with the identification and rank ordering of security risk inherent in the system’s design. Only after risks have been identified can they be properly managed. A large emphasis is placed on testing particular vendor-card implementations to ensure adherence to security design. Risk-based security testing (linked directly to risk-analysis findings) of both Java Card and Open Platform helps to ensure that cards adhere to critical design features and will thus be able to perform in a secure manner in the field.
Together, Cigital and Visa continue to develop an automated security test framework for Open Platform cards built on Java Card 2.1.1. The end result of the effort is a sophisticated test framework that runs with minimal human intervention and results in a qualitative security analysis of a sample smart card platform. The current security test framework tests the Open Platform and Java Card security functionality. Additional security testing must be performed on each of the applets to be loaded onto the card in Visa's and its members' system, ensuring that no applet introduces unacceptable security risk.

With the advent of multi-application smart cards, including the javacard, an entirely new set of software-related security concerns has been raised. As we have discussed, allowing post-personalization modification of a fielded card can be very dangerous. Malicious code placed onto a loadable card can, in some cases, completely compromise card security and lead directly to automated attacks that work against large sets of related cards (especially those created by the same vendor). One of the primary goals of the Cigital security test framework is to probe Open Platform cards for such weaknesses and determine a basic risk factor for a given implementation.

Conclusion

Java Card and related smart-card technologies that allow for multiple applications will open new markets and are likely to play a central role in many modern systems. Multi-application smart cards such as javacards provide a well-understood, common platform on which to build advanced systems. As such, they can provide consumers with both ease of use and peace of mind. As with any new technology, however, using advanced functionality involves taking on security risk. Just as Java is not secure by virtue of its name alone, smart cards do not offer a security panacea. Smart card security risks must be properly managed just like any other security risk.

Java Card 2.1.1 and Open Platform do an admirable job of managing security risks, bringing Java's advanced language-based security approach to an entirely new market. When risks are properly managed, Java Card provides an excellent platform for the future of e-business.

The software security issues faced by the architects and adopters of Java Card offer a peek at the future of security writ large. New mechanisms, including language-based access control and type-safe languages are destined to play a very large role in future security architecture and design. Such mechanisms will become prevalent in the coming age of mobile, distributed and wireless systems, even outside the smart-card arena.

*Gary Mcgraw, Ph.D., is the Chief Technology Officer at Cigital, Inc. He can be reached at gem@cigital.com.*

*Mark McGovern is a former Director of Technologies at Cigital.*

Original article © 2001 Information Security magazine.

Copyright © 2002 by Cigital, Inc. All rights reserved.