Security Risks of Java Cards

Anup K. Ghosh
Reliable Software Technologies Corporation
21515 Ridgetop Circle, #250, Sterling, VA 20166
email: aghosh@rstcorp.com
http://www.rstcorp.com

1 Introduction

As early as the 1980s, France issued smart cards for their Public Telephone and Telegraph (PTT) system. Only recently have smart cards begun penetrating the commercial market in North America. With the introduction of Java Card 2.0 (hereafter referred to simply as Java Card), interest in smart cards for commercial applications in North America appears certain to grow. The key innovation that Java Card brings to smart cards is the ability now for programmers anywhere to write programs that will drive smart card applications. Previously, programming smart cards was the exclusive sanctuary of a small group of assembly language programmers that coded at the machine architecture level. With adoption of the Java Card subset of Java, smart card applications can be developed by a broad range of programmers using a variety of development environments.

Commensurate with innovation in smart card technology is the potential for new, serious violations of security in using these cards. For example, smart cards that hold stored value must not be susceptible to attacks that can add value arbitrarily to the card without deducting an identical amount from another account. Incorporating a sophisticated operating system on a smart card also introduces the possibility that errors in the implementation may be exploited by malicious programs to subvert the card security. The rocky history of Java security has shown that even though security control mechanisms were built into the Java Virtual Machine (JVM), the complexity of the machine made an error-free implementation impossible. Some of these flaws were exploited to break the security of the JVM [6].

Java Card is a stripped down version of the Java language that is designed for the smaller memory footprint of smart cards. For instance, Java Card does not support threading, and optionally may not support garbage collection and de-allocation of memory. The Java Card subset has its disadvantages as well as its advantages in relation to security. On the one hand, the security manager class is not included in the Java Card. In standard Java, the security manager is responsible for denying unsafe operations. On the other hand, dynamic class loading is not supported for Java Card. This means that an applet cannot dynamically download classes that are not already existent on the card. Dynamic class loading is a key source of insecurity in Java; omitting it from Java Card goes a long way to mitigate the type confusion attacks that have plagued Java security to date [6].

In this article, an overview of the security issues in smart cards is presented to identify potential risk areas that must be assessed prior to fielding Java-enabled smart cards in commercial applications. The different risk areas this paper covers are: secure protocols for electronic transactions, protocol interactions, risks of multi-application cards, the risks of an immature technology deployed in critical applications, and physical security considerations.

2 Secure Protocols

The transaction protocols implemented by smart cards must be secure from protocol-based attacks while fulfilling the desired functions of the transaction. Protocol-based attacks attempt to exploit weaknesses in the protocols such as errors of omission or errors of commission in the protocol. For example, a protocol that fails to include a dynamic nonce during authentication may be susceptible to capture-replay attacks.

Some smart card applications have stricter requirements than others. In electronic currency applications, for example, properties desired for transactions include: security, anonymity, scalability, general acceptability, off-line operation, transferability, and hardware independence [7].

For most commerce-related transactions, smart cards must provide the following security attributes: confidentiality, authentication, data integrity, and non-repudiation. Confidentiality ensures that the transaction remains a secret between the two parties to a transaction during the course of the transaction. Note that most confidentiality mechanisms do not ensure confidentiality of transaction data after the transaction is complete. Authentication is the process of verifying the identities of the parties to a transaction. Note that in some protocols, only one party is authenticated. In some protocols, anonymity of one or more parties is a desired attribute and can be enforced through blinding [3]. Data integrity is required for most commerce-related transactions to ensure that the data transferred between the parties is not altered in any way through either incidental noise or deliberate tampering. Non-repudiation is a property of transactions that removes deniability for the parties to a transaction. For example, when making an investment transaction over the phone with a broker, the phone conversations are often taped to provide proof that the investor did in fact authorize the transaction. These tapes are only recalled in case the investor attempts to deny authorization for a transaction. Non-repudiation can be enforced to some extent in electronic transaction by requiring the participant(s) to digitally sign the transaction. Whether these signatures are legally binding is still a matter for the courts to decide.

While many protocols have been developed for smart card transactions, few have been formally verified. Most attempts at verifying protocols have been by the academic community for protocols designed by the academic community [9]. Commercial protocols are often designed by commercial vendors with little public domain scrutiny. This approach is commonly called security through obscurity and will often fail through careful protocol analysis.

Protocols that interact with principals over public networks such as the Internet face additional risks. For example, eavesdropping by a passive sensor is trivially performed. Encryption protocols can reduce the risk of passive eavesdroppers, however, covert timing channels can be exploited to discern secret key information. Another real threat to Internet-based protocols is from an active intruder that is positioned between both transaction parties to intercept the communications. Such an intruder can not only block traffic from one principal to another, but also impersonate one principal to be given the rights and privileges of this principal. Again, encryption protocols can be used effectively to mitigate risks from these types of attacks and must be employed in any public network where unknown elements lurk.

Encryption is used to provide the security properties defined above for most smart card protocols. Encryption, however, is not a panacea to the problems of providing secure transactions on a large scale. For example, many authentication systems used in practice are symmetric [9]; that is, the same key used for encryption is used for decryption. Employing symmetric keys is generally cheaper to implement than using asymmetric key systems such as public/private key cryptography. However, symmetric encryption systems suffer from scalability problems. Because each set of communicating principals must agree in advance to a shared secret, a large number of keys must be generated through a trusted mechanism. The number of keys necessary to support symmetric encryption between a community of principals grows with the square of the size of the community [2]. Generating symmetric keys for protocols implemented over distributed networks is a nontrivial problem that has resulted in a large body of research [9].

The design of secure protocols for secure applications is a very difficult process rife with errors. For example, the Needham and Schroeder authentication protocol [8] that revolutionized security over distributed systems, employed in protocols such as Kerberos and Andrew File System, was later found to be flawed. The difficulty in designing secure protocols resulted in a body of research focused on formally analyzing protocols. Despite the amount of formality and rigor used in analyzing protocols for security, it is clear that it is not possible to prove that any given protocol is secure [9]. Part of the problem lies in defining what secure means. It is possible, for instance, to prove certain assertions about protocols such as whether it is susceptible to capture-replay attacks; however, no general purpose method exists for determining how secure a given protocol is in general.

As difficult as the design of a secure protocol is, the correct implementation of the protocol is an even harder problem. Even given a secure protocol, it is often possible to "break" such a protocol by exploiting weaknesses in its hardware or software implementation. To this end, software testing processes must not only perform oracle-based testing, but also stress testing with the purpose of attempting to break the security

policy of the application.

Java Cards, more than any other innovation in smart card technology, will provide the ability to develop and implement several different protocols for a wide range of commercial applications. The risks of secure protocol design and implementation must be weighed more heavily in light of the capabilities Java Card brings to smart card applications.

3 Protocol Interactions

In today's smart card market, multi-application cards are becoming the rule rather than the exception. That is, rather then carrying several different cards for different purposes, the applications are integrated on a single card. For example, a smart card may provide your driver's license or social security identification, a credit line, a debit account, e-cash, health insurance information, calling card applications, digital certificates for authentication, and different loyalty programs such as frequent flyer mileage. Now, in practice, it would be difficult to get so many different commercial vendors and government entities to agree to integrate their applications on a single card. One stumbling block may be the branding of the card. But, technologically, it is not only possible, but happening today on a limited scale. The Java Card platform makes application integration relatively simple. Developers can write their applications to a particular standard interface to seamlessly (and independently) integrate their applications with others. Some of the practical risks of multi-application integration are covered in the next section.

Insofar as multiple applications implement different protocols (many of which may require authentication), unintended protocol interactions may compromise the security of the card and its multiple applications. Because of small memory constraints in smart cards, it is currently impractical to store unique cryptographic public/private key pairs for each different application requiring cryptographic functions. As a result, multiple applications often share the same key material.

Different protocols that share the same key material can introduce potential security problems that would not otherwise exist with each protocol considered in isolation. That is, the security of two different protocols do not necessarily compose, especially given protocol interactions. In a paper on this topic, Kelsey et al. postulate that it is always possible in principle to defeat the security of one protocol if another protocol can be chosen by the same principals in parallel [5].

There are many instances in which protocols may unknowingly have interactions. For example, simply observing the messages from one protocol $\mathcal{P}$ may provide an observer the ability to attack a second protocol $\mathcal{Q}$ executing on the same card. Consider that both protocols rely on a digitally signed time stamp from one party for capture-replay resistance. An external observer can use the response to a challenge from $\mathcal{P}$ to form a response to a challenge from protocol $\mathcal{Q}$. Multi-application cards that share secret key material may be compromised by protocol interactions in unanticipated ways. Several instances of known protocols are presented in [5] where a chosen protocol may potentially compromise the security of the known, target protocol.

Today, there are several motivating reasons for smart card vendors and application developers to share cryptographic keys. A small memory footprint makes storing several public/private key pairs infeasible. The adoption of digital certificates makes maintaining several digital certificates per person for each different application unlikely. Finally, with standard cryptographic APIs supported in Java Card, it becomes increasingly likely that multiple applications will use the APIs to access the user's certified public key for authentication [5]. To reduce the likelihood of protocol interactions that compromise card security, application and card designers should not reuse keys in multiple applications.

4 Multi-application Risks

Aside from the risks of shared secret key material, the introduction of multi-application cards poses new risks to card security. On Java Card, multiple applets can be installed to execute different applications. While Java Card is a subset of the standard Java language, it is important to realize that the security models implemented by each are different. For example, Java Card does not have the security manager class that standard Java relies on to prevent applets for gaining unauthorized access to system resources. On the other hand, Java Card does not allow dynamic class loading. This means that Java classes may not be downloaded

"on-the-fly" during execution — a security risk assumed by standard Java. Instead, Java applets are usually burned in during masking when the card is initialized and personalized prior to being fielded.

Although dynamic class loading is prohibited by Java Card, post-issuance loading of Java applets is not. That is, even after cards are issued and deployed in the field, Java applets may still be installed on the cards, albeit not on-the-fly during applet execution. This feature is one of the significant benefits of placing a true operating system on a card. Existing software applications (applets) can be upgraded to the latest release versions and new applications can be installed without trashing the card that has been issued. For example, if a banking client wishes to add a debit account to her credit card, this customization can be performed by a local branch of the bank by installing a debit card applet that draws off of an account of deposit with the bank. Similarly, when joining loyalty programs with hotels or airlines, a loyalty program applet can be installed that keeps track of all benefits accrued via various transactions. While promising to simplify our wallets, this technical innovation also has its risks. Post issuance of applets provides the avenue for an unscrupulous merchant to install an applet that can compromise the security of the card. Because Java Card does not implement a "sandbox" model of security, the applets that are installed are trusted to not behave maliciously. To address these threats, one model adopted for installing applets post issuance is to require that a secret cryptographic key is presented in order to install an applet. This access control mechanism ensures that only an "approved" vendor can install applets. However, requiring a cryptographic secret says nothing about the secure behavior of the applet once it is installed. This point is touched on later in this section.

Three features of Java Card are responsible for providing a secure environment executing card applets: dynamic class loading is not supported, multi-threading of applets is not supported, and applet firewalls prevent unintended applet interactions. Eliminating dynamic class loading prevents a number of potential security attacks that have compromised the security of standard Java. Eliminating multi-threading prevents the possibility of one applet observing or interfering with the execution of another applet. Finally, the use of applet firewalls to prevent one applet from referencing another applet's name space can prevent colluding applets from compromising card security or prevent one applet from illegally referencing another applet's objects to escalate its own privileges.

In spite of these secure features, it is important to realize, however, that much of the card security is based on two assumptions: the Java Card Runtime Environment (JCRE) is correctly implemented and that applets loaded on the card do not behave maliciously. The JCRE must be correctly implemented to prevent one applet from illegally referencing another applet's objects. That is, the applet firewalls must be correctly implemented or else applets may be able to exploit holes in the JCRE to access other applet's objects (such as another applet's private keys). The latter point also underscores the vulnerability of Java Cards to malicious applet developers. To a large extent, the Java Card security model is based on trust — trust that applets that are burned in during masking are safe and will not behave maliciously. Without a sandbox model and an associated security manager, there is no mechanism in place to prevent a malicious applet from exploiting its own privileges for someone else's malicious gain. For example, an applet developer can code in additional logic that allows a colluding terminal application to surreptitiously swipe money off the card or to obtain private cryptographic keys.

The trust-based model assumes that applet developers will not write Trojan horses that generally behave benignly except given certain inputs known only to conspiring parties. The trust-based model may be acceptable under certain circumstances where tight control over development and installation of applets is maintained. However, because Java applets can be installed post issuance by developers from any number of vendors, the trust-based model may not be feasible. Two factors should give pause to smart card stake holders that depend on the trust-based model: first, when the financial stakes rise, it may no longer be sensible to blindly trust one's developers; second, errors in omission or in commission of applets may make them vulnerable to concerted attacks that attempt to subvert an applet against itself and its hosting card. With the adoption of Java for smart cards, the range of qualified programmers for smart card applications increases significantly over the range of qualified assembly language programmers traditionally employed for smart card development. The implications are that future smart card developers will not necessarily be an elite corp of highly skilled developers. With a larger pool of developers the chances for a programmer to write malicious applets for personal gain increases. Given the current Java Card security model, the best approach for smart card application vendors is "trust but verify". That is, the card applets must be tested for secure behavior in addition to functional testing. Given the potential for Trojan horses and back

doors, white-box software testing techniques should be employed in addition to standard black-box testing. Ultimately, when adopting a model that supports the installation of executable code, all such code must be viewed with suspicion and proved beyond a reasonable doubt to be secure.

Finally, the use of native methods from Java applets should be carefully controlled. Native method calls bypass any security model implemented within the JCRE. By calling native code within the card OS directly, a Java applet can essentially execute functions without any restriction imposed by the JCRE. Unfortunately, native method calls are sometimes preferred by developers to eliminate the extra performance overhead in using Java APIs. However, the security risks for bypassing the JCRE controls are tantamount to opening new holes in a firewall for convenience sake.

5 Risks of Immature Technology

As in any new technology, there are always significant risks associated with deploying immature technology in critical applications such as financial transactions. Ambiguities in the Java Card specification may result in different Java Card implementations that execute identical applets with different behaviors. Currently, the JCRE and Java Card Virtual Machine (JCVM) provide developers lots of latitude to choose among different options for execution of applets. For example, garbage collection may be implemented on some cards but not on others. In cards without garbage collection, memory leaks resulting from programming flaws may exhaust the card's limited resources quickly. Cards that do perform garbage collection will not suffer from this problem. By studying the differences in the JCRE/JCVM implementations, it may be possible to find and exploit vulnerabilities. The Kimera research group out of the University of Washington did precisely such a study on commercial JVM releases in products such as Netscape Communicator and Microsoft Internet Explorer.[1] Their study found several differences in the behavior of the JVMs given identical Java byte codes. Ultimately, some of these differences were exploited by the Kimera group to violate the security of the JVM.

Another potential risk of immature technology and ambiguous specifications is the implementation of the byte code converter. The development process of Java starts from writing Java source code than compiling it to Java byte codes. Java byte codes are portable to any Java-compliant virtual machine. Because Java Card is a subset of standard Java, the byte code produced for Java Cards must also be restricted to the subset of acceptable instructions and types for Java Card. To this end a byte code converter is responsible for post-processing Java byte codes to ensure that the Java Card constraints are met. If the set of byte codes (class files) accepted by the byte code converter is greater than the subset of Java specified for Java Card, than the potential to compromise card security exists by executing instructions outside of the Java Card specification.

A key point to remember is that the security model(s) of standard Java do not apply to Java Card. For example, the security model defined by Java Developers Kit (JDK) 1.1 provides a sandbox for constraining the behavior of applets such that they cannot read or write to the file system, nor make network connections to hosts other than the one from which they downloaded. As mentioned earlier, this sandbox model is not adopted by Java Card. The recent changes in JDK 1.2 have changed the Java security model from a sandbox model to a trusted security model. That is, Java applets are given more latitude to move beyond the sandbox to access system resources provided they are given the correct cryptographic signatures. The risks of this trust-based model are described in detail in [4]. Because the security model in Java Card is not well-defined, it is not clear if Java Card will adopt a firmer security sand box model or the cryptographic signature model for assigning trust. In its current specification, Java Card appears to adopt neither. What is known, however, is that with the types of financial applications supported by Java Cards, the security stakes are much higher for Java Card applets than for Java applets downloaded from a Web page.

6 Physical Security Considerations

As in any security-critical system, the design and analysis of such systems must take a holistic approach that includes hardware as well as software considerations. This paper highlights the risks of software- and protocol-based attacks against smart cards especially in light of the new capabilities that Java Cards support. These considerations, however, do not preclude consideration for the physical security of the card. Physical

[1] See http://kimera.cs.washington.edu/flaws/ for an overview of this research.

attacks against smart cards historically have been the preferred method for breaching smart card security by card hackers. Fortunately, physical attacks against smart cards have been well-documented [1] and smart card vendors are well aware of physical techniques used for attacking smart cards.

One property of smart cards that has been proven time and again is that no card is tamper proof. The goal of smart card security is to make the cards tamper resistant enough to make the cost of breaching card security high enough (in terms of time and resources) to outweigh the rewards of successfully hacking the card.

7 Conclusions

This article has highlighted several risks associated with the adoption of a high level programming language support on a smart card. The introduction of Java Cards may well be a boon to the smart card industry and electronic commerce as a whole. As in most technical innovations that provide greater functionality in a more accessible form, there is often a trade-off in dependability. In the case of Java Cards, the ability to support Java applets will result in a wider variety of applications being supported on a single card. However, the security risks for supporting executable content on smart cards must be carefully evaluated in light of the types of applications the cards will support. The risks of Java Cards include an untested security model, a broader range of application developers, a trust-based model for applets, the potential for protocol interactions via shared keys, and the ability to bypass the JCRE with native method calls. To provide a level of security assurance commensurate with the high-risk financial applications, rigorous testing and certification technologies must be applied to Java applets developed for Java Cards.

Acknowledgment

The author wishes to thank and acknowledge Chuck Howell for reviewing the manuscript and providing useful feedback.

References

[1] R. Anderson and M. Kuhn. Tamper resistance—a cautionary note. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, pages 1–11, November 1996.

[2] R.W. Baldwin and C.V. Chang. Locking the e-safe. *IEEE Spectrum*, 14(2):40–46, February 1997.

[3] D. Chaum, A. Fiat, and N. Naor. Untraceable electronic cash. In *Proceedings of Crypto '88*, 1988.

[4] A.K. Ghosh. *E-Commerce Security: Weak Links, Best Defenses.* John Wiley & Sons, New York, NY, 1998. ISBN 0-471-19223-6.

[5] J. Kelsey, B. Schneier, and D. Wagner. Protocol interactions and the chosen protocol attack. In *Security Protocols, International Workshop April 1996 Proceedings.* Springer-Verlag, 1997.

[6] G. McGraw and E. Felten. *Java Security: Hostile Applets, Holes, and Antidotes.* John Wiley and Sons, New York, 1996.

[7] G. Medvinsky and B.C. Neuman. Netcash: A design for practical electronic currency on the internet. In *Proceedings of the First ACM Conference on Computer and Communications Security*, November 1993.

[8] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[9] A. Rubin and P. Honeyman. Formal methods for the analysis of authentication protocols. Submitted for publication, 1995.