Smart cards are getting smarter

An analysis of the vulnerabilities introduced with Java Card 3 Connected Edition

by Andrew Calafato, MSc (RHUL 2012) and Konstantinos Markantonakis, supervisor, ISG, Royal Holloway
Smarter cards bring greater risks

An overview of the new features introduced in Java Card 3 Connected Edition and the associated security risks that developers need to be aware of

by Andrew Calafato, MSc (RHUL 2012) and Konstantinos Markantonakis, supervisor, ISG, Royal Holloway

Introduction

The use of smart cards has rocketed in a number of industries, reaching billions of SIM cards in the telecommunications sector, contact and contactless credit or debit cards in the finance industry, electronic ID cards and e-passports, pay-tv cards, transport contactless cards and other applications. These devices typically host a microcontroller and internal memory to store keys and other information securely in a tamper-resistant system, protecting the confidentiality and integrity of sensitive data. The microcontroller offers a secure computing environment for the execution of algorithms to carry out on-card functions, typically providing security services such as encryption, integrity and authentication.

Smart card technology has been progressing at a fast pace, and cards with a 32-bit processor, tens of Kbytes RAM and hundreds of Kbytes EEPROM/Flash memory are available on the market. Sun (acquired by Oracle in 2009-2010) has been in the smart card market for more than a decade with the Java Card technology, which is a secure, portable, platform-independent and multi-application platform based on the Java object oriented programming language for smart cards and other memory constrained devices. Version 3 of the Java Card specification was released in March 2008 and introduced a completely new architecture compared with the previous versions. The specification became available in two separate yet coherent editions, both of which are backwards compatible with existing applications and share important security features:

- The Classic Edition is an evolution of the predecessors with several incremental changes to ensure alignment with smart card and security standards.
- The Connected Edition introduced a significantly enhanced execution environment and a new virtual machine. It includes new network-oriented features, support for web applications with new Servlet APIs, multi-threading and support for applets with extended and advanced capabilities.

The new features of the Connected Edition increase the applicability of the smart card, and Oracle is marketing Java Card 3 Connected Edition as being for use in a networked environment, for example, for integrating a smart card as a network node into a local network providing security services. They also provide sample applications including features such as discovering services on other cards on the same local network, performing secure card-to-card transactions and administering cards through a web interface, taking advantage of the fast USB interface.

Such features add complexity to the smart card platform and the hosted applications, increasing the attack surface and introducing a multitude of vulnerabilities. The security models, testing and risk management programmes must cater for these susceptibilities. In this article we consider the new features of the Connected Edition and identify some of the security problems that developers need to be aware of.
Smart card operating systems

The firmware or software residing on smart cards has evolved over the years from monolithic firmware embedded in the chip by the semiconductor manufacturer for specific client requirements, into multilayered execution platforms with general-purpose operating systems capable of hosting multiple applications. In these open platforms, the manufacturer would typically embed the hardware management layer and a virtual machine such as Java Card, Multos or SmartCard.Net. The smart card manufacturers and issuers could thereafter add an application framework hosting a number of applications on the same card. The virtual machine (VM), such as Java Card VM, enables portability of applications, which could be loaded onto different hardware without modification.

Java Card

Java Card is one of the leading smart card platforms, providing the ability to host and manage multiple applications on smart cards and other memory constrained devices. Java Card applications, called applets, can be loaded into different Java Card devices without the need for recompilation. They can also be loaded, removed or dynamically updated after the card has been issued, without the need to issue a new or different card.

Java Card applications can be written in a subset of the Java language, inheriting all the benefits of object-oriented programming such as code reuse and design patterns, although development for such devices requires specific techniques and attention to the resource constraints. Standard Java development environments can be used during development, significantly facilitating application development.

The system architecture on the Java Card is made up of a number of layers, as illustrated in Figure 1. As can be seen in the top layer of the architecture, the Java Card specifications support three application models:

- The Classic applet application model – applet-based applications in the Classic Edition specifications, similar to ones in previous versions using the Application Protocol Data Unit (APDU) based scheme of communication. These applications comprise a single package and execute in a single-threaded environment.
- Extended applet application model – applets with more advanced capabilities using the APDU-based communication and the new features introduced in the Connected Edition specifications. They may compromise multiple packages.

<table>
<thead>
<tr>
<th>Applets, extended applets and servlets</th>
<th>Multiple Java Card applications can reside on a single card. The applet container manages classic and extended ISO 7816 APDU applet-base applications, while the servlet container manages the life cycle of HTTP servlet-based web applications.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Industry add-on classes</td>
<td>Add-on libraries providing services or refinements to the security and system models.</td>
</tr>
<tr>
<td>Java Card Framework</td>
<td>Defines a set of application programming interface (API) classes for developing Java Card applications and for providing system services to those applications, such as application management, transaction management, I/O communication and cryptographic functionality.</td>
</tr>
<tr>
<td>Java Card Virtual Machine (JCVM)</td>
<td>A virtual machine (a smaller and simpler Java Virtual Machine) which executes byte code, providing the functions accessible from outside, such as signature, log-in and applications. The JCVM controls access to all smart card resources, such as memory and I/O and allows applications to be securely loaded to the card post-issuance. It hides the manufacturer’s proprietary technology with a common language and system interface.</td>
</tr>
</tbody>
</table>

Figure 1: Java Card system architecture
and concurrently process APDU commands from different I/O interfaces.

- Web application model – applications based on servlets that use the HTTP protocol to support a web-based communication with the card, also using all the new features introduced in the Connected Edition.

**Java Card 3 – new features, new threats**

The innovative aspects of Java Card 3 were introduced in the Connected Edition specification, and can be used by extended applets and web applications. These features include the ability to develop multi-threaded applets and web-applications with advanced connectivity, thus introducing a number of vulnerabilities that should be managed in such a security-sensitive environment.

**Multi-threading**

The Java Card 3 Virtual Machine supports parallel execution of processes – multi-threading – in web application and extended applet environments and has a multi-threaded runtime environment. The runtime environment can service multiple incoming requests concurrently (for example, HTTP and APDU commands) and can perform multiple executions in parallel, such as application management tasks, using what are known as RE-managed threads. The web applications and extended applets can also spawn application-managed threads themselves to perform background or concurrent tasks.

An evident threat introduced by multi-threading comes from the inherent complexity that concurrency brings with it. Analysis of the Java Card Virtual Machine and multi-threaded application code becomes much more difficult from a security point of view in a multi-threaded environment because of the non-determinism introduced. Such applications are also more difficult to test and debug for the same reason.

A multi-threading environment introduces risks of race conditions whenever more than one thread accesses shared data at the same time. An example is a missed update, which occurs when a shared value is read by another thread before it is updated locally. The other thread might process the ‘original’ value and override the update in this thread. Standard solutions exist to avoid such situations, usually based on locking mechanisms which can be used in Java Card 3 applications.

Another form of race condition is time-of-check to time-of-use (TOCTTOU), whereby a change can occur between the time that a particular resource is checked and the time that it is used. An example of this was created on Oracle’s Java Card Netbeans-based simulator, whereby a debit card balance was illegitimately set to a negative value: two parallel calls for withdrawing $10 from the balance of $15 were successful, since both checks revealed there were sufficient funds. Once again, locks should come to the rescue here, but they actually have to be predicted and implemented.

Developers must be aware that two or more threads can end up in a deadlock while attempting to acquire each other’s locks, i.e. waiting for each other to release some lock. In general, deadlocks are difficult to detect, and can thus introduce unpredictable denial-of-service (DoS) vulnerabilities. A malicious thread might also lock a resource for a long period of time, thereby creating another form of denial of service. The fact that threads might be assigned different priorities might be abused, by having a number of high-priority threads causing thread starvation for others with lower priorities.

**Web applications**

The Java Card Connected Edition specifications support a subset of the Java Servlet Specification v2.4, allowing smart cards to host web-based applications.
Such applications interact with off-card clients via HTTP and HTTPS request/response protocols over TCP/IP, typically using high-speed interfaces such as USB.

Such applications are exposed to the same vulnerabilities that ‘standard’ web applications experience. These are listed in a number of websites such as the OWASP Top 10 most critical web application security risks, WASC reports and the CWE/SANS Top 25 Most Dangerous Software Errors. The limited resources of smart cards can sometimes make them easier targets, and countermeasures that consume a lot of resources are out of the question.

Java Card web applications will be listening for HTTP and HTTPS connections, and possibly also APDU connections for backward compatibility. This would entail new outward ports facing the network or the World Wide Web, new URIs available for genuine users as well as attackers, and new applications and services processing requests on the devices.

A common vulnerability in web applications is the failure to restrict URL access, whereby attackers easily forge URLs to access various services of a web system if the application does not perform the necessary authorisation checks. This might lead to information leakage and illegitimate alteration of sensitive data on the card.

Injection attacks occur when unexpected input data (or any untrusted data) is sent to some data processor (or interpreter) and ends up being executed unintentionally. Although injection attacks such as SQL and LDAP injection into database and LDAP queries might not be a significant threat in the restricted smart card environment, the risk of injecting HTML and OS commands does exist. For example, in HTML, any text between <!-- and --> tags is treated as a comment and not shown to the user. If a user inputs the text ‘<!--’ for some input such as a transaction description, whenever this text is displayed in the browser,
the rest of the report would be treated as a comment and therefore ignored. Such an attack is known as a comment injection attack and occurs since the untrusted input was not validated before being processed and was not encoded when displayed. Any sort of HTML command can be injected in the same way.

Similarly, if an application executes some OS command using unsanitised untrusted input, it may be vulnerable to OS injection attacks. The XPath query language, which is used to query entries from XML files, can also be vulnerable to such attacks.

Another form of injection attacks is cross-site scripting (XSS), which occurs when active content such as JavaScript is injected by an attacker. The motive behind XSS is to send malicious code, generally in the form of a browser side script, to a different unsuspecting user. The victim would then think the script is from a trusted source and may act upon the potentially altered content, such as following a malicious link. Some scripts might also execute without the user’s knowledge and might access or alter sensitive data on the page.

Whenever a file (or resource) is accessed on the basis of some input, if the right sanity checks are not performed, the application will be vulnerable to path manipulation attacks. The primary goal behind such attacks is for an attacker to gain file access and additional capabilities through a covert channel. Path manipulation attacks can also be the cause of integrity issues if the application code is altering, overriding or deleting files whose name is wholly or partly dependent on some untrusted user input. The attacker might change or forge the filename to access confidential areas of the file system.

Cross-site request forgery can also be applied to the smart card environment. The aim of this attack is to try to force an authenticated victim to send a forged HTTP request to a vulnerable application, without the knowledge and intent of the victim. For example, an attacker might create a web request that transfers some money to his account from an authenticated victim. Since the attacker is not authenticated, he sends the request in the form of a link in an email or web page. The authenticated victim is fooled into clicking the link, thereby executing the transfer without even knowing it. There are standard solutions to counter this attack, which have been implemented on the Oracle smart card simulator.

Web applications can also be vulnerable to attacks related to access control. Whenever a reference to a protected resource is exposed without any access control, an insecure direct object reference or failure to restrict URL access vulnerability is created. The attacker might alter the URL or its parameters to access privileged pages that are not protected by the necessary authorisation mechanisms.

DoS attacks on web applications and web servers are usually associated with vulnerabilities in lower layers of communication, but this susceptibility can be found in the application layer as well. Multi-threaded smart card applications can be vulnerable to asymmetric resource consumption, whereby resources such as file and connection handles can be left open, or closed after a predetermined amount of time, possibly causing DoS.

This vulnerability is amplified in the smart card environment because of the limited resource on such devices (when compared with servers) and because of the fact that objects on the heap are usually persistent in Java Card applications, thus retaining their state. A DoS attack, in particular on Java Cards, might be targeting an application which persists some state on every request. A large number of repeated requests would stress the rewrite limits of EEPROM.

While trying out these vulnerabilities on the Netbeans-based Java Card simulator, another form of DoS was simulated with an application that stored a
history of the past five transactions it effected. When a very long message was input, the simulator managed to store the value, but whenever the user tried to request the past five transactions, an exception was thrown by the web server and the page never appeared.

There are many other attacks related to web applications that may be ranked as less critical in publicly available statistics, but are still applicable to the smart card environment. A number of these were also simulated on the Java Card simulator.

**Inter-application communications – services and events**

Java Cards can host multiple applications and the VM guarantees their integrity and confidentiality with the use of an application firewall. The Java Card Runtime Environment defines access restrictions to components and objects owned by different applications, but provides a number of ways where applications can interact with each other: inter-application communication.

The Java Card inter-application communication facility was extended in the Java Card 3 Connected Edition to allow interaction between classic applet applications and extended applets as well as web applications. It is based on shareable interface objects (SIO), which are objects implementing shareable interfaces. The Connected Edition specifications introduced a global service registry whereby web applications and extended applet applications can register and look up SIO-based services.

Another inter-application communication feature introduced is SIO-based asynchronous communication, which is a notification mechanism based on the classic observer design pattern, with the implementation of events and event listeners. Applications can register as SIO-based event listeners with the Java Card Platform or other applications for certain conditions. When these conditions occur, the application fires SIO-based events to the registered listeners, which can handle (act upon) the event.

Such communication can become an attack vector, especially when considering physical and combined attacks in smart cards. For example, the attacker might take advantage of the expected context switching when SIO method calls are performed, or might use power and timing information in side-channel attacks. Sensitive information might be leaked to other applications through the offered services, perhaps unintentionally. Apart from confidentiality issues, if the service exposes references to the internal system state such as a balance, the client application could access and alter the information, causing integrity issues.

**Network communications: the generic connection framework**

Connected Edition smart cards typically have one or more high-speed interfaces such as a universal serial bus (USB), and the Java Card platform provides logical network interfaces typically, but not necessarily, over IP.

Communication with external systems can be performed intrinsically through the web application container and through the Generic Connection Framework (GCF) API, a service offered to applications residing on the device. Applications can open server communication endpoints using the GCF APIs to receive requests, and can also initiate client communications with off-card entities, thus acting as clients to off-card services across a network.

Being connected to a network for a prolonged period of time and making use of communication stacks such as TCP/IP and layers above, introduces new vulnerabilities. In APDU communication, information would be typically flowing between the card and card reader, but now sensitive information might be travelling over public networks to or from the smart card. All the passive attacks
(eavesdropping and traffic analysis) and active attacks (message alteration, replay, masquerading to get privileged access, traffic floods causing DoS, and so on) related to communication should be catered for and the new risks should be managed in this ‘connected’ environment. Besides, other attacks such as brute-force vulnerabilities, dictionary attacks and cryptanalysis would be significantly easier to perform.

The risks related to the different lower network layers would all apply to Connected Java Cards. The prevalent ones include:

**Physical layer:**
- Information leakage;
- Integrity violation through tapping or sniffing;
- Denial of service (DoS) through accidental or deliberate damage and any sort of illegitimate use of the network.

**Datalink layer (link layer in TCP/IP):**
- ARP spoofing or ARP poisoning, which route frames to the attacker for eavesdropping, MAC flooding to exhaust switch memory causing a DoS;
- VLAN vulnerabilities;
- Weak protocols and key management.

**Network layer (internet layer in TCP/IP):**
- IP spoofing to launch man-in-the-middle attacks and DoS;
- Smurf attack, teardrop and land flooding network with traffic such as ICMP.

**Transport layer (TCP, UDP and others in ICP/IP):**
- Open SYN attack or SYN flood consuming precious smart card resources causing DoS;
- Injecting forged packets.
Dynamic class loading

With Java Card 3, Java classes can be loaded dynamically at runtime as opposed to being explicitly imported from another class. This complicates the type safety enforcement process significantly, and type confusion (illegal type cast) can lead to serious security threats such as reference forgery. Researchers have shown examples of application firewall circumvention using reference forgery to alter the class bytecode of a neighbouring application.

When loading classes dynamically, there is always a chance that the class being loaded includes malicious code, thus introducing harmful code after the application has been deployed.

To limit the possibility of harmful code injection, the Java Card VM specification is quite restrictive about the classes that can be loaded dynamically, and allows loading only of pre-declared classes from the same library (Jar file).

Garbage Collector

The Java Card 3 specifications support automatic Garbage Collection (GC), which is a process that reclaims any unreferenced (inaccessible) dynamically allocated storage during the execution of a program.

The GC can be considered as a security feature giving some protection against memory leaks (objects left dangling in memory without being freed), but it is also an additional process running on the device that can be exploited by attackers. One such attack explained in a research paper exploited the GC to launch a form of application replay attack, with the possibility of application firewall circumvention.

What needs to be done

There are a number of countermeasures that should be considered in Java Card Connected Edition application development to strengthen the security models used in ‘Classic’ smart card environments.

The Java Card platform itself provides a security model suiting devices with limited resources used in security-critical environments. The model can be split into three levels:

- Low-level security or virtual machine security, including features such as bytecode verification, ensuring that applications do not harm the target device;
- Application-level security controlling application access to libraries, file system, network and other resources based on the security policy defined for the application. This is based on a sandbox model, whereby applications run in a closed environment with strict access controls to resources;
- End-to-end security: an infrastructure for the implementation of end-to-end security services such as authentication, authorisation, encryption and integrity between the device and other parties involved in different forms of communication.

The runtime environment and virtual machine offer a number of security features, which in some cases need to be correctly configured and implemented. These include:

- Controlled loading of applications onto the device;
- Bytecode verification for checking the validity of application bytecode loaded on a device to guarantee memory safety;
- Controlled dynamic class loading;
System class protection from modification, overriding and introduction of new classes;
Applications and applet firewalling, isolating applications loaded onto the card;
Thread-related safety measures, including access control, thread-safe system code and system libraries;
Ability to group a number of operations as transactions treating the whole operation as a single unit of work. The transaction mechanism is used to guarantee the integrity of data being processed, and recoverability from crashes, card tear (sudden loss of power) and transaction abortion to provide application stability;
Authentication and authorisation – a permission-based model used to provide fine-grained access control to sensitive resources such as network access, files, context switching shareable objects, and so on. A declarative role-based security model is also available. These models provide a lot of flexibility, but are complex to master, implement and configure flawlessly.

Multi-threaded applications should be developed in a thread-safe way, protecting against race conditions and deadlocks, typically using locking strategies. The testing of multi-threaded applications should address the aspect of non-deterministic executions, whereby different runs of the program can potentially produce different results. It should also be noted that the underlying physical layers and operating system can play a role in the timing, scheduling and context switching of threads. Thus multi-threaded applications should be tested on different smart card architectures.

In web application development, a number of controls and tests should be implemented to protect from web-related vulnerabilities. These include:

- Input validation and output encoding to protect against injection attacks and DoS;
- The implementation of the synchroniser token pattern to protect against cross-site request forgery;
- Access control of URLs and other resources.
The necessary configuration and controls should also be implemented to protect interactions of the device with the outside world. This can be achieved using end-to-end communication security to protect the integrity and confidentiality of data. Necessary access control should also be put in place to control intra-application communication.

Having a Java-based environment and familiar application development tool might make one think that developing Java Card applications is the same as developing standard Java applications. The physical layer and environment on which the applications will run differ significantly from server boxes locked inside rooms and behind firewalls. A Java developer cannot create good quality Java Card code without understanding these differences, which include:

- Non-volatile memory is stressed every time it is used and becomes unreliable after a certain number of writes;
- Very limited RAM space;
- Many objects need to exist across card restarts or need to be persistent;
- Card tears (sudden loss of power) need to be handled.
- Smart card applications require security features beyond the scope of the standard Java platform. For instance, a legitimate cardholder should not be able to tamper with his balance amount.

Conclusion

With the Java Card 3 Connected Edition features, smart cards are evolving into multi-threaded web servers integrated to IP networks hosting complex applications offering a number of services.

The increasingly sophisticated applications, the more services and application integration offered and the greater connectivity entail a larger application attack surface and a number of new vulnerabilities that need to be addressed in the evolving smart card environment.

These new attack possibilities, described above, need to be considered in the security management frameworks of smart card services and the applications and environment of which they are part. Choosing and implementing countermeasures on the resource-limited devices can offer a challenge and complicates the risk management processes.