



Trusted Logic

## RESEARCH WHITE PAPER

---

# Proactive Security for Mobile Applications

---

By Eric Vétillard, October 2004.

Ref. PU-2004-RT-621-1.0

The present paper reviews several ways to enforce security measures on mobile Java applications. It opposes two ways to achieve security: reactive security, and proactive security. It also compares several techniques that can be used to enforce security policies, with a specific focus on static code analysis technologies. Finally, it includes a detailed study of a recent attack performed against J2ME/MIDP phones and how it can be addressed with the help of static code analysis.

## 1 Proactive Security vs. Reactive Security

### The Windows Heritage

When reasoning about application security, the Windows operating system immediately comes to mind. It prevails in the PC world, and it is riddled with security problems. As of today, the situation is concerning for many users: there is so much Windows malware that circulates on Internet that a new computer is likely to be infected while downloading the latest Windows updates that will protect it against this very malware.

The only solution available to Windows users is reactive security. Since there are many security holes and issues in the operating system itself, and since they cannot all be fixed, the best possible solution is to use a catalog of all existing malware and to ensure that none

is ever allowed on the computer. This catalog is the database provided by any antivirus software.

However, reactive security is not perfect, and it leaves many issues open:

- Catalogs must be kept up-to-date in order to be efficient, which is particularly difficult to individual end-users.
- When a new malware spreads, it takes a while before it is included in the catalog, and before the updated catalog is loaded on end-users' computers.
- The identification of malware is normally done through a signature (a pattern in the code), rather than on the vulnerability it relies on.

Naturally, the issues with the Windows operating system are more visible because this system is so widespread, but other systems are affected as well. On mobile devices, there are quite a few vulnerabilities in the current versions of the Symbian operating system, which allow malware to be developed for this platform. For instance, in June 2004, a virus was developed for Symbian devices, which spreads through the Bluetooth interface<sup>1</sup>.

In theory, this situation could be improved. Most of the vulnerabilities exploited by malware developers

---

<sup>1</sup> The Bluetooth interface is used because it is less controlled than the network connections, which all go through the operator's private network, making it difficult to use them to spread malware without being detected.

are discovered by “white-hat” hackers, who publish them without trying to exploit them illegally. Since the exploitation of a vulnerability can often be tied to a specific use of an operating system function, it is possible to determine whether or not a program relies on a vulnerability in a proactive manner. However, such proactive checks are very difficult to perform on native code such as this used on the Windows and Symbian platforms.

### Java on Closed Devices

Mobile Java applications are mostly developed with the J2ME/MIDP application model. These applications then run on a dedicated Java Virtual Machine, implemented on a mobile device. On many devices, Java is the only way to download new applications on the system, since the Java Virtual Machine is implemented on an otherwise closed operating system.

The original Java security model, known as the “sandbox model”, is a proactive security scheme, with the following characteristics:

- Applications are not allowed to perform any operation that could harm the system (for instance, they cannot access another application's persistent data).
- Applications are run on the Java Virtual Machine, which means that the application cannot directly access the device's operating system, and that all such accesses are mediated by the Java Virtual Machine.
- The Java Virtual Machine includes a bytecode verifier, which verifies before to run an application that its code is correct and does not perform any illegal operations.

This security model is very strong, and has proved very efficient over time: there has been very few malware written as Java applets, although applets are widely used in Web sites. However, this model is too limited, and it has been replaced over time with a more flexible model, which allows an application to access sensitive operating system functions under strict conditions:

- The application must explicitly request the right to perform a sensitive operation.

- All accesses to sensitive functions are checked by the Java Virtual Machine.
- This right may be granted by a trusted third party, which signs the application after certifying it.
- In some less sensitive cases, this right may be granted by the end-user, usually on a limited basis.

This security model introduces two major weaknesses: first, the trusted third-party who signs applications must have the ability to verify that the application is indeed “innocent”; then, the end-user, who can grant limited rights although he has no security skills, may be subjected to social engineering attacks from malware.

On the other hand, this model guarantees that every application that is granted access to highly sensitive APIs will go through some certification process before to be downloaded. In terms of certification, Java bytecode has two major advantages over native code: it is meant to be statically checked (by the bytecode verifier), and it is portable across all devices, so all static checks can be performed using the same technology.

Because of these advantages, it becomes possible to implement proactive security on Java applications: the bytecode can be analyzed in order to check that it does not exploit any known vulnerability. In addition, it also becomes possible for an operator to define a specific security policy regarding the use of specific system resources, and to enforce this policy on all applications.

Naturally, this technology is not exempt from bugs and vulnerabilities, and it may come under attack. The particular attack described below is a typical exploit of an implementation bug. It weakens the basic sandbox model, but it does not really affect the extended sandbox model, since this attack can be thwarted through the organizational measures that lead an issuer to sign an application. There are several ways to do so:

- The first one is to use the standard Java verifier to verify all classes. Its design is completely different, so its bugs are likely to be different. In addition, the standard Java verifier has been widely in use for a long time. In the case of Gowdiak's attack, the classes would be rejected by this verifier.
- If the issuer defines a security policy (which restricts the use of the API further than what is al-

### Attacking J2ME

A successful attack of a closed phone has been published recently [1]. This attack is based on the weakest point of the security model: the bytecode verifier. Since the verifier is quite complex, Sun's reference version is used on almost all phones. Gowdiak has discovered a flaw in this verifier, which allows some incorrect programs to be loaded on the devices. By exploiting this flaw, Gowdiak has proved that he was able to take complete control of the device. More details about this attack are provided in appendix.

Like any attack that exploits a verifier flaw, this attack uses illegal code, which can be countered by any kind of static analysis during the certification process. Other kinds of attacks can be avoided by proactive security in the same way. For instance, limiting the size of the buffers passed to some sensitive APIs can be used against buffer overflow attacks. See Appendix for detailed analysis of the attack and proposed counter-measures.

lowed by MIDP), then many attacks can be avoided by enforcing this policy.

The key idea is here that the Java principles are not only useful at the level of the execution platform; they also make certification easier, which can be successfully used to thwart attacks that rely on implementation bugs at the platform level.

## 2 Certification Technologies

Most mobile Java applications deployed today under the responsibility of telecom operators have to go through some kind of certification program, either before it is made available for download, or just before downloading it.

### Testing

Testing is performed today by most operators, but the main focus is not security. Operators typically need to verify that their look-and-feel guidelines are strictly followed, that all required legal mentions are included, and that the application is functional on all devices on which it is supposed to run.

Application testing is an important issue for operators, in particular because it is costly. The importance of the costs comes from two factors: the devices are not fully compatible with each other, so many different versions of each application must be tested, and testing is labor-intensive, since it must be performed manually.

For this reason, several manufacturers have teamed with Sun Microsystems in order to define a set of unified testing criteria [2], and to select a number of testing houses to test the applications. This “Java Verified” program has been endorsed by a few operators, but it does not yet address security concerns.

Testing for security is difficult in the sense that the amount of effort spent in testing is necessarily limited, and some categories of malware, such as Trojan horses, can simply not be detected using testing. On the other hand, testing can be efficient in order to detect social engineering attacks, in which the application attempts to misinform the end-user.

Rule	How to check it
The application requests the privilege to open HTTP connections	Format and consistency checks
The application does not use RMS to store data	API usage scanning
The application does not use proprietary APIs	API usage scanning
The application only accesses the domain “ <a href="http://www.anything.com">www.anything.com</a> ”	API usage analysis
The application does not send a SMS to a premium number	API usage analysis
No user private data is sent to the outside	Control and data flow analysis
The application checks the certificate data for every secure connection	Control and data flow analysis

Table 1: How to check rules statically

### File Scanning

File scanning is the technology used by antivirus software. The idea is to identify malware through specific signatures, and to check all running software against these signatures. This technology is necessarily reactive: its goal is to locate applications that are known dangerous, and to allow the other applications to be loaded and run. It can only detect malware that has been previously identified, and it is not able to verify that an operator-specific security policy is applied.

File scanning nevertheless has some advantages, in particular its performance, which allows operators to run it transparently on all downloads. However, on a technology such as mobile Java, there are a number of reasons that reduce its ability to detect actual problems:

- Mobile phone networks are private, and phones only have limited means to communicate directly with one another, which drastically limits the spreading of malware such as worms and viruses.
- Since the technology is young, there is very little known malware, and most issues are expected to be related to API misuse (intentional or not).
- The operators' security policies are likely to evolve rapidly, which means that checking the security policies is an important factor.

### Static Program Analysis

Static program analysis is the technology used by the Java bytecode verifier. The idea is to analyze all possible behaviors of an application in order to show that the application cannot in any way violate a given policy. A wide variety of rules can be checked, and some examples are provided in Table 1.

Because the Java bytecode is well-structured, there are several levels of static analysis that can be used:

1. Format and consistency checks.  
The checks simply consist in verifying that the binary file is properly formatted, and that the static parts of the file (for instance the manifest) contain the proper information. For instance, it is possible

to check the list of privileges requested by the applications.

2. API usage scanning.

The analysis consists in scanning the file in order to identify all the API classes and methods used by the application. This provides interesting information, but not enough to certify the application, because the context in which the classes and methods are used is not known.

3. API usage analysis.

The analysis consists in an actual analysis of the application's behavior. It can determine which APIs are used, as well as the context in which they are used, in particular the values of the arguments used to invoke a method.

4. Code and data flow analysis.

The analysis is here more complex, since it is able to figure out how the code is sequenced, and how the data flows in the application. This allows the definition of very fine checks, yielding very flexible security policies.

Static analysis is a powerful tool, because it can actually prove that an application behaves properly with respect to a security policy. This means that it can be used to implement a proactive security model, which only accepts the applications that follow the security model and rejects all others. In such a system, an application is rejected unless it is proved correct.

Static analysis of levels 1 and 2 is quite common. Checks on the APIs used are commonly integrated in provisioning systems, where they are used to verify that an application can run on a given device. Consistency checks are used in the "Java Verified" program as a preliminary step before interactive testing.

In the higher levels, static analysis faces a few general issues that need to be tackled properly in the definition of a static analysis system:

- Performance.

Since static analysis is able to prove that an application abides to a given policy, it sometimes requires complex computations, which are time-consuming. However, the performance issues can be addressed

by designing the static analysis adequately for each use.

- Precision.

One issue with static analysis are "false positives", i.e., programs that are correct but rejected because they cannot be proved correct. This issue is related to the precision of the analysis and also to the security policy itself, because the rules must be stated in a precise way, readily usable by developers.

These issues are very hard to address for general programs, but it is possible to design efficient and precise systems on limited programming models. The J2ME/MIDP model is one of these, because it is not as complex as Java's model, and the size of the applications to analyze usually remains small.

### Wrap-up

Each technology mentioned above has advantages and drawbacks, which are summarize in Table 2.

This table shows that testing hardly provides any help in the context of a security certification, that scanning is an effective way to apply a reactive security model (for instance to filter download Symbian applications), and that static analysis is the most effective way to apply a proactive security model (for instance to sign Java applications).

In the context of a Java application filtering, a complete solution can be used by using static analysis and scanning:

- All signed applications are evaluated using a full static analysis during their certification.
- All unsigned applications are evaluated using a simple static analysis and a file scanning during the download process.

With such a process, the operator uses proactive security when its responsibility may be at stake (for instance, when it certifies the applications provided through its own portal). On the other hand, the operator simply uses reactive security when the user assumes the responsibility of downloading applications from unknown servers.

Issue	Testing	Scanning	Static Analysis
Ability to be performed automatically	---	+++	+++
Ability to be performed at download time	---	+++	-
Ability to detect known malware	--	+++	+++
Ability to detect social engineering attacks	+	--	++
Ability to detect known vulnerability exploits	--	+	+++
Ability to enforce a security policy	--	--	++
Ability to re-process certified applications	--	+++	+++

Table 2: Comparison of Certification Techniques

### 3 Trusted Logic's Approach

Trusted Logic has a strong background in automated program processing, ranging from simple static analysis techniques to automated code generation. In particular, Trusted Logic has developed the first Java bytecode verifier embedded on a smart card, and has developed static analysis technologies that are currently used in the security evaluation of Java Card applications.

Trusted Logic is combining this technological expertise with its significant experience in terms of security analysis, in order to offer to operators the tools and methodology that they need in order to define a security policy and to enforce it through a certification program.

#### A Powerful Analysis Tool

The heart of Trusted Logic's offer is a powerful static analysis tool, which performs significant checks on mobile Java applications. This static analysis tool includes a full API usage analysis, and it also performs some code and data flow analysis. In particular, Trusted Logic's tool considers the possible values of variables and method arguments in its analysis, making it possible to verify that a sensitive method is only used with proper arguments.

For instance, a method that often comes under close scrutiny in the MIDP API is the `Connector.open` method, which opens a method according to the URL passed as parameter. This URL is at the center of many security policy rules:

- The application only uses some protocols.
- The application only opens connections with a given domain.
- The application only processes SMS's on given ports.
- The application does not send SMS's to premium numbers.
- ...

It is therefore very important to determine statically the possible values of these URL's, in order to verify that these rules are not violated. As we have seen, precision is a key issue for static analysis tools. Trusted Logic's tool fares quite well in this area, as shown in Figure 1, which shows the results obtained on a sample of applets obtained from a free download site.

These statistics show that over 80% of URL's were correctly analyzed by the program, and that around 16% of URLs could not be determined sufficiently in order to apply the rules they satisfied. This result is very good, because the applets developed are completely unaware of the security policy to be applied.

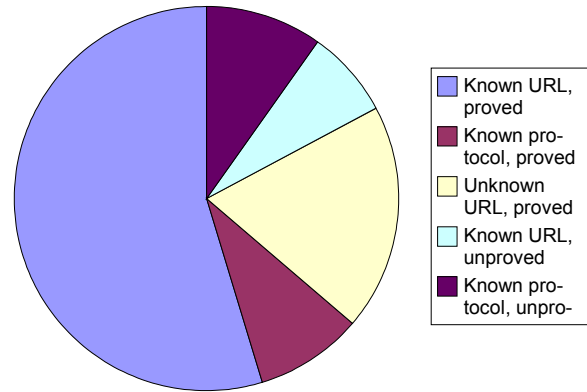


Figure 1: Precision Statistics

In addition, the precision would be improved further by applying a real security policy, and with applets going really through certification.

In terms of performance, the analysis runs in a few seconds on all applications, even when performing a complete static analysis. In this configuration, the tool can be used automatically on a developer site, and the developers get a rapid feedback. A complete analysis is too complex to run on all downloaded applications, but it is possible to only perform a subset of the analysis, and to perform a verification in 1 or 2 seconds.

The static analysis technology used in this tool has been developed over several years by Trusted Logic. Tools based on the same technology are currently used on a daily basis for the security evaluation of smart card applications, especially in the banking industry. A similar tool is distributed by the SIM Alliance, and used to check the portability of SIM Toolkit applications.

#### Security Policy Definition

One of the important differences between proactive security and reactive security is that operators have the possibility to define a specific proactive security policy which suits their needs, and their customer's needs.

For instance, an operator may choose to allow or not the use of a given protocol, or to allow or not the use of a given proprietary API. In addition, a security policy can evolve over time, as some features become more common and more standardized. Finally, a security policy does not have to be monolithic, and different rules may be applied to different kinds of applications, to different providers, or even to different target users.

Trusted Logic has acquired a significant experience in defining security policies that can suit an operator's needs, and that can be readily applied by developers. The process defined by Trusted Logic includes several steps:

- Risk analysis.

During this phase, Trusted Logic gathers the re-

quirements from the issuer, and analyzed the risks related to these requirements. For instance, the APIs to be made accessible and the mobile phones to be used are to be considered in the analysis.

- **Policy Definition.**  
The second step consists in defining the policy itself, by translating the requirements expressed as a result of the risk analysis into rules that can be verified on programs.
- **Guidelines Definition.**  
The final step consists in translating the rules of the security policy into rules guidelines for developers. This step is crucial, since it provides to developers the information required to write applications that respect the security policy.

This process has already been performed many times by Trusted Logic in the wireless market, as well as in other markets.

In addition, Trusted Logic has defined a few generic security policies that can be used as a base from which to derive an operator's own security policy.

## 4 Conclusion

We have compared two approaches for controlling the security of the Java applications downloaded on mobile phones.

Reactive security, and in particular antivirus software, is an efficient approach on platforms on which the security is already compromised, such as the Windows and Symbian platforms. However, it requires a heavy maintenance, and it is not appropriate for emerging platforms, on which the attacks remain widely unknown.

Proactive security is a very appropriate solution for mobile Java applications: it can verify that applications do not use any known vulnerability, and it can also check that applications verify an operator's security policy. However, proactive security requires the ability to statically analyze the applications before downloading them on a device and as such needs to be carried out as part of an operator driven certification process.

Trusted Logic offers consulting services to operators who want to define a security policy, as well as a static analysis tool that enforces the security policies once they have been defined.

## References

- [1] Adam Gowdiak. *Java 2 Micro Edition Security Vulnerabilities*. Hack In The Box Security Conference 2004. Kuala Lumpur, Malaysia.
- [2] The Java Verified Program. *Unified Testing Criteria for Java Technology-based Applications for Mobile Devices*. Version 1.4. 28 September 2004. Available on [www.javaverified.com](http://www.javaverified.com).

## About the author

Eric Vétillard is the Chief Architect of Trusted Logic. His main expertise is on the security of embedded Java technologies, and in particular about tools and methodology for the evaluation of embedded Java applications and platforms.

Reach him at [eric.vetillard@trusted-logic.com](mailto:eric.vetillard@trusted-logic.com).

## About Trusted Logic

Since 1999, Trusted Logic has brought secure open technology based solutions to the embedded systems industry. Corner stone of its product offering, Trusted Logic provides Trusted Foundations™, complete solutions with first class security for smart cards, card-accepting devices and mobile phones including open technology platforms (jTOP™), accompanying tools and security services. Thanks to its innovation strategy, Trusted Logic has been providing several “firsts” in the industry (first on-card verifier, first on-card debugger, etc.).

For more information about Trusted Logic, visit [www.trusted-logic.com](http://www.trusted-logic.com).

## Appendix: Gowdiak's Attack

This attack, as mentioned in the paper is a perfect example of how a simple vulnerability can lead to a complete attack. Below is a summary of the attack followed by an analysis of possible countermeasures.

### The Attack

The starting point of the attack is here a bug in the implementation of the bytecode verifier embedded in the phones. One very simple verification has been omitted: the verifier does not check that all jumps are performed within the same method. However, by correctly exploiting this simple bug, the bytecode verifier can be defeated entirely, and Gowdiak has been able to make it accept applications that are not correctly typed. It is therefore possible to load and run applications that perform computations on Java references.

So far, the attack is really devastating. Concerning bytecode verification, the Java security model is brittle. Since the implementation of a verifier is quite complex, the reference implementation is used on almost all phones. The bug exploited by Gowdiak is therefore likely to be present on many MIDP phones.

The typical exploit of such a vulnerability is to write an application that forges references and then gets read and/or write access to the entire memory of the device. This reduces the portability of the attack, since this part of the exploit is necessarily specific to each device, or at least to each underlying software platform.

For his demonstration, Gowdiak has targeted a specific closed device, the Nokia 6310i phone. On this device, he has been able to use "type confusion" to access any part of the memory using a Java applet.

At this point, designing a really powerful attack is just a matter of patience. Since the device has been designed as a closed device, its level of protection against attacks is quite low. By reverse engineering the code of the Java Virtual Machine and of the operating system itself, it is possible to do many things, including:

- Change the domain of a Java application.  
If an application is associated to the operator domain, it is automatically granted all permissions, which for instance allows an application to send an unlimited number of premium SMS without any user interaction.
- Interfere with other applications.  
Since the interprocess communication has been discovered, it is possible to misuse other applications, for instance to send SMS messages or to initiate phone calls.

Basically, it is possible to do anything if somebody has enough knowledge and time to continue the reverse engineering of the device.

### Possible Countermeasures

There are several possible countermeasures to this attack, which can be implemented at various levels. The simplest one consists in running the standard J2SE verifier on any application's class files before to allow the download of this application on a phone.

This countermeasure is efficient because the J2SE bytecode verifier is compatible with the J2ME bytecode verifier, and it is based on a completely different design. It will therefore catch this particular attack, as well as most other exploits of the J2ME verifier's implementation bugs. It is simple to implement, since the J2SE verifier is available on most platforms and it is very fast. However, it may be difficult to make this verification compulsory, since users have many ways to download applications on their phones.

Another possible countermeasure is to use this verifier or another static analysis tool during an application's certification procedure, *i.e.*, before to sign it. For instance, Trusted Logic's application validation tool would catch this particular bug, and most likely most other verification-related bug. This tool does not include a verifier *per se*, but it performs very similar checks during its static analysis phase.

In addition, Trusted Logic's static analysis tool would also catch most illegal API uses, and would therefore limit greatly the scope of the attack.

However, this approach has the same drawback as other static analysis approaches: it can only be applied to applications that are signed. For an operator, this means that the approach can be applied to all applications available from its portal, but not on the applications that the user downloads from Internet, which in most cases are untrusted applications.

The two ways to counter this attack are either to somehow fix the initial vulnerability in the bytecode verifier, or to deny end-users the right to download uncertified applications. These approaches are not entirely feasible today, but they could become accessible in the near future:

- If the phone includes a security layer (both hardware and software, for instance based on ARM's TrustZone architecture, combined with Trusted Logic's Security Module), it will then become possible to update the device's security model, including the bytecode verification model.
- By catching all downloads on a device, an operator could propose a service that certifies untrusted applications using a very basic security policy, and then use a static analysis tool such as Trusted Logic's on every file downloaded into a device.