Software Security Aspects of Java-Based Mobile Phones [*]

Karsten Sohr
Center for Computing Technologies (TZI)
Universität Bremen
Bibliothekstr. 1
28359 Bremen, Germany
sohr@tzi.de

Tanveer Mustafa
Center for Computing Technologies (TZI)
Universität Bremen
Bibliothekstr. 1
28359 Bremen, Germany
tanveer@tzi.de

Adrian Nowak
Otaris Interactive Services GmbH
Fahrenheitstr. 1
28359 Bremen, Germany
nowak@otaris.de

ABSTRACT

More and more functionality is provided by mobile phones today; this trend will continue over the next years. However, with the increasing functionality new risks go along. This not only applies to security-critical mobile applications such as m-banking or m-commerce applications. The end user's privacy may also be in danger or the operator may be the target of an attack. In this paper, we discuss security risks introduced by mobile phones considering the perspectives of the different parties involved in telecommunications systems. Specifically, we demonstrate those risks by means of a security hole discovered in a large number of mobile phones. The security hole can be exploited to obtain manufacturer or even operator permissions. In particular, we implemented a Java-based Trojan horse. This way, the compromised mobile phone can be used as an eavesdropping device by an attacker. All in all, this demonstrates that the risks are not only theoretical, but also real. We also sketch a methodology for the security analysis of mobile phone software.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive Software

General Terms

Security

Keywords

Mobile Phone, Java Security, Static Security Analysis

1. INTRODUCTION

More and more functionality is incorporated into mobile phones. Current mobile phones not only provide the possibility to initiate and accept phone calls. SMS, MMS, and now even e-mails can be sent or received. In particular, new applications such as Java MIDlets [13], small Java programs for mobile phones, can be downloaded or installed via Infrared or Bluetooth. Furthermore, a broad number of Java Specification Requests (JSRs) are made available for mobile phones, again introducing new functionality which can be conveniently used, such as the Multimedia API [11]. Given the fact that most of the people in developed countries possess a mobile phone and use those phones either for private or for business purposes, security- and privacy-relevant data are stored on mobile phones such as lists with addresses of customers, tasks lists and lists of phone numbers.

As a consequence, one might assume that security and privacy seem to be at least important as in the PC world. Up to now, however, end users do not seem to be prepared for attacks on mobile phones, although some reports on hacking mobile phones already exist, e.g., [19, 25, 30]. In the future, it can be expected that the rate of security incidents will rise because of the increasing attractiveness of the attack target "mobile phone" [10].

In this paper, we show by means of a concrete security hole occurring in a large number of current mobile phones from Sony Ericsson which security risks can arise in the context of mobile applications. The security hole can be exploited by using simple Java API calls, and we have implemented a Trojan horse based upon the security hole. We also show that the end user can act as an attacker obtaining access rights originally not intended for him. For example, a user might circumvent digital rights management (DRM) mechanisms or obtain operator permissions. In the end, the security hole can be seen as a case study demonstrating security risks related to mobile phone platforms. This specifically applies to high-end mobile phone platforms such as BlackBerry and Android, which provide their own rich Java-based middleware.

We also briefly discuss a methodology for a static security analysis of the source code which Sony Ericsson could have employed for improving the security mechanisms of the Java Virtual Machine (JVM). The proposed analysis technique is based upon the Java Modeling Language (JML) [2], a specification language for Java, and employs related static analysis tools such as ESC/Java2 [2] as well as a reverse engineering tool-suite called Bauhaus [23]. The Bauhaus tool allows one to carry out an analysis on the software architecture, i.e., at a higher abstraction level. Thereafter, a more focused security analysis at the source-code level can be performed with the help of JML tools.

The rest of the paper is structured as follows. In Section 2,

[*]This work was supported by the German Federal Ministry of Education and Research (BMBF) under the grant 01IS09032 (VOGUE project).

we provide an overview of related concepts and technologies such as Java security mechanisms for mobile phones. Section 3 gives more details on the security hole discovered in mobile phones, discusses several attack vectors and specifically describes our Trojan horse. In Section 4, we discuss the consequences of the security hole for all parties involved, whereas we sketch solutions for avoiding such problems in Section 5. We outline our conclusions in Section 6.

2. BACKGROUND

In the following, we first give a short overview of Java for mobile devices (often called "Mobile Java") including the Java 2 Micro Edition (J2ME). As indicated above, we later employ Mobile Java as the main vehicle for exploiting the aforementioned security hole, and in particular, the additional functionality made available by the Java APIs can be conveniently used by the Trojan horse to exploit the security hole. To give a flavor of the functionality that can be used by an attacker, we describe the various security-relevant Java APIs for the J2ME in more detail. We also explain the basic security concepts of the J2ME, and at the end of this section, we sketch DRM mechanisms for mobile phones because these mechanisms can also be circumvented by means of the security hole.

2.1 Java for mobile phones

The J2ME introduced by Sun Microsystems in 1999 was meant for devices with low computational power and memory. In particular, this applies to PDAs and mobile phones, but also to other embedded systems such as VoIP phones or sensors. The central part of the J2ME is a *configuration*. For mobile phones, the Connected Limited Device Configuration (CLDC) [26] is used. The CLDC defines a lean JVM called "Kilo Virtual Machine". Based upon the CLDC, the Mobile Information Device Profile in version 2.1 (MIDP) has been specified and then been implemented or licensed by most of the manufacturers of mobile phones. The MIDP makes available certain APIs for developing Java MIDlets. For example, functionality for networking, access to serial ports, application auto invocation[1], and graphical user interfaces are provided. In addition to the MIDP, various JSRs are often implemented such as the Java Wireless Messaging API (supports sending and receiving of SMS); the Bluetooth API (for creating Bluetooth applications); the Personal Information Management API (for handling contact, calendar, and task lists); the FileConnection API (for accessing certain folders, subfolders, and files of the mobile phones); the Multimedia API (for access to audio and video capabilities of the mobile phone such as the built-in camera). In newer models, the Security and Trust Services API (SATSA) is implemented, which among other functionality lets MIDlets communicate with Subscriber Identity Module (SIM) cards by means of the Application Data Unit Protocol (APDU). For example, the `exchangeAPDU()` method of the class `APDUConnection` can be used to access smart card applications. Moreover, Sony Ericsson makes available additional APIs. One of them includes functionality to retrieve the current cell ID, and gives the local area code of the mobile user. This way, the current position of the user can be determined.

[1]With the help of this so-called PushRegistry, Java MIDlets can also be started remotely. Then, the activation of MIDlets can, for example, be triggered by an SMS sent to the mobile phone in question.

2.2 Security mechanisms for Mobile Java

Java for mobile devices provides various security-critical APIs. For this reason, a sandbox model similar to that of the Java 2 Standard Edition (J2SE) has been introduced [17]. In particular, a byte code verification mechanism serves as a low-level security mechanism guaranteeing certain language properties such as type safety, however, with a lower memory footprint and power consumption than its J2SE counterpart. The other important security concept of the J2ME are protection domains, once again similarly to the J2SE security model. A *protection domain* encompasses permissions on security-relevant Java APIs such as those mentioned in Section 2.1. Java MIDlets can now obtain access rights through the protection domains. In particular, one can configure a protection domain in a way that applications on mobile phones run in a sandbox with restricted access rights. Four domains are predefined according to the MIDP specification [13], namely, Unidentified Third Party Protection Domain, Identified Third Party Protection Domain, Operator Domain, and Manufacturer Domain. Java MIDlets are assigned to protection domains according to the X.509 code signing certificates used for signing the MIDlets.

The first two domains differ by the identification of the Third Party which created the application and different default and other user settings. The MIDlets from the Identified Third Party Protection Domain have been signed by a certificate from a trusted certification authority (CA) whereas the MIDlets of the Unidentified Third Party Protection Domain have been signed by an unknown CA or have not been signed at all. In both cases, however, the user shall be prompted (at least once) when a security-critical operation is performed such as sending an SMS.

The other two domains can be defined depending on the specific manufacturer and operator, respectively. Strictly speaking, the MIDlets in those domains should also ask for permissions on accessing sensitive APIs. This is, however, not mandatory.

2.3 Mobile digital rights management

Ring tones, music files, games, and Java applications are copy-protected in mobile phones by means of DRM mechanisms. There exist two different DRM mechanisms for Sony Ericsson mobile phones, one for the protection of Java applications and games, which is proprietary. The other is meant to protect ring tones and music files and is based upon the specifications of the Open Mobile Alliance (OMA) [20]. In both cases, the object to be protected is encrypted, e.g., in case of OMA DRM by means of AES keys. If an attacker has access to the decrypted versions of the objects under protection, then the DRM mechanism can be circumvented. For instance, this is the case when an attacker gets access to unencrypted files stored in the hidden storage of the mobile phone or when the AES keys can be accessed.

3. EXPLOITING THE SECURITY HOLE

In this section, we first explain the security hole and then describe several attack scenarios. Thereafter, we present a Trojan horse that has been developed to demonstrate the consequences of the security hole. We also sketch how a worm can be built by exploiting the vulnerability.

3.1 Accessing arbitrary internal files

Usually, Java MIDlets may only access the external memory of the mobile phones; the internal file system cannot be accessed by means of the methods of the FileConnection API (sandbox model). There is, however, a link concept incorporated into Sony Ericsson's mobile phones on the OS level. This means that files can be accessed by symbolic links similar to the Unix link concept [7]. The access rights of the files the links point to are not checked by the OS. This way, the well-known security principle "complete mediation" is violated. As a consequence, read and write access to arbitrary internal files (including system files) is possible if one can create link files pointing to internal files. In fact, the FileConnection API allows an attacker to create link files although this should have been prohibited. That means there is a second security hole—this time, in Sony Ericsson's version of Mobile Java. In the following, we discuss this Mobile Java hole in more detail. First, observe that link files on Sony Ericsson's operating system use the special character '@' as the file name suffix. Directly creating such link files is not possible yet, i.e., Sony Ericsson's JVM throws an exception in this case. However, there is an indirect way to achieve the aforementioned goal: The `rename()` method of the FileConnection API accepts file names with that special character. As a consequence, we can first create a file containing the link and thereafter rename the file in such a way that it is interpreted by the OS as a link file.

To sum up, we have carried out the following steps to detect the aforementioned security hole(s):

1. Due to the fact that applications on Sony Ericsson mobile phones are distributed via Java the J2ME was selected as an attractive attack target.

2. Then operating system specifics have been analyzed. In the case of Sony Ericsson mobile phones, the operating system is Enea OSE RTOS [5] (and not Symbian as it is the case with Nokia smart phones). Since comparatively little documentation on this operating system is publicly available, we analyzed the file system structure, and found out that files with special characters in their file names existed.

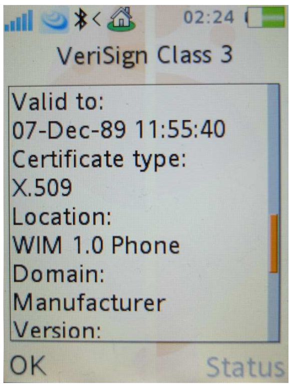
3. We took a closer look at those files and found out that some of them contained file names. Interestingly, we learnt that the mobile phone crashed occasionally on changing those file names. From that, we concluded that there was a kind of symbolic link concept.

4. Finally, we searched for a way of exploiting the link concept by means of Mobile Java and detected that the `rename()` method was not appropriately secured.

One consequence of the aforementioned security flaw is that the user could access sensitive files. For example, pre-installed certificates for m-banking, m-commerce or code-signing can be overwritten. This will be discussed in the following.

3.2 Attack scenarios

In this section, we describe several attack vectors, including the description of a concrete Trojan horse that we have implemented to give a real-world demonstration of the consequences. Given the fact that arbitrary system files are accessible, other attacks are also conceivable.



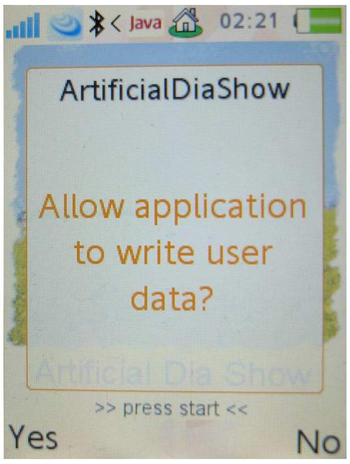
Valid to:
07-Dec-89 11:55:40
Certificate type:
X.509
Location:
WIM 1.0 Phone
Domain:
Manufacturer
Version:

Figure 1: Non-trustworthy certificates can be added to the Manufacturer Domain.

Circumventing DRM mechanisms. In the first scenario, the end user can be seen as the attacker in that he can break the DRM mechanism provided by mobile phones (see Section 2.3). In particular, the aforementioned security hole lets the user make a copy of Java games and programs with the help of a simple Java program. The Java games can then be copied to other mobile phones, even from a different manufacturer, due to the platform independence of Java. In addition, AES keys for protecting other digital content such as music files are available in the internal file system, which can lead to break the DRM mechanism for such content. We have not further investigated that point because our focus lies more on attacking the end user. However, we later discuss the general consequences when the DRM mechanisms are broken on mobile phones (see Section 4). Furthermore, we do not claim that there are not other ways to circumvent DRM on mobile phones. We only want to demonstrate that the security hole can be exploited in different ways.

Obtaining manufacturer permissions. The second scenario describes the possibilities from the perspective of an attacker who tries to gain unauthorized access to security-critical applications such as m-banking applications. As mentioned initially, one can manipulate files on the mobile phone through the aforementioned security hole. This also applies to security certificates in the protected storage area of the mobile phone, which are used to verify signed Java applications (see Section 2) or to confirm the trustworthiness of secured web pages (e.g., financial institutes, e-commerce shops). In Figure 1, we show a Verisign Class 3 certificate that has been overwritten and then added to the Manufacturer Domain. Note that we must overwrite an already existing certificate because we only have the right to read and write files, but not to create new ones.

As mentioned in Section 2.2, Java applications on mobile phones usually run in a sandbox with restricted access rights. However, if an application is signed with a manufacturer certificate, the application can even obtain the full functionality of the phone, without the need to ask the user. As indicated above, the MIDP 2.1 specification recommends that MIDlets in the protection domain of the manufacturer should prompt for confirmation in case security-critical APIs are called. Sony Ericsson seems to have implemented a less


02:21

ArtificialDiaShow

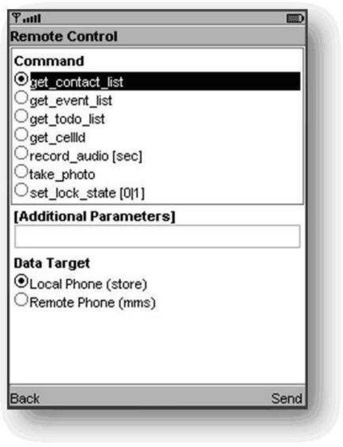
Allow application to write user data?

Artificial Dia Show

>> press start <<

Yes No

Figure 2: A security warning asking for write access.


Remote Control

Command
- (●) get_contact_list
- ( ) get_event_list
- ( ) get_todo_list
- ( ) get_cellId
- ( ) record_audio [sec]
- ( ) take_photo
- ( ) set_lock_state [0|1]

[Additional Parameters]

Data Target
- (●) Local Phone (store)
- ( ) Remote Phone (mms)

Back Send

Figure 3: Controlling a mobile phone remotely.

restrictive policy, which violates the principle of least privilege. This also contributed to the exploitation of the security hole by the Trojan horse described in the subsequent section.

The Trojan Horse. Exploiting the aforementioned security hole, it is possible for the attacker to implement a Trojan horse which can violate the privacy of the end user. This Trojan horse is realized with the help of a simple, unsigned Java application, which has been installed on the user's mobile phone. In principle, this unsigned Java application could install a further Java MIDlet now signed by our (modified) manufacturer certificate. But we decided to go another more simple way: The Application Management System (AMS) of the JVM has an internal table (represented as a file) which contains the assignments between protection domains and Java MIDlets. Then we only had to change the protection domain entry of our Java MIDlet in this table. Note that we can carry out this attack because we have access to arbitrary files in the file system including that one containing the table. This way, we have an unsigned Java MIDlet, which now belongs to the Manufacturer Domain.

To hide the true functionality of the Trojan horse, we implemented a slide-show showing some pets. The user must now only confirm security warning "Allow application to write user data?" (see Figure 2). This warning looks harmless because the end user might guess that user data are stored. He can be confused by showing a simple message right before the actual security warning appears as a social engineering attack. For example, this message could state that the application wants to store the pictures of the pets and that this step must be confirmed by the user. The end user should be familiar with such a kind of security warning because often Java MIDlets from trustworthy vendors like Google Maps Mobile are unsigned and request for permissions with the help of similar security warnings.

With the confirmation of the warning, the Trojan horse breaks out of the sandbox, gathers the manufacturer rights, and uses all further services of the mobile phone, without any consent of the user or manufacturer. The security warning of the Java system looks harmless such that users will ignore it—this way, we can speak of a user interface error, which also contributes to a successful social engineering attack.

Having prepared the attack, we implemented a hidden functionality that reads the contact list and sends that list per SMS to the attacker. Moreover, the Trojan horse has a built-in eavesdropping functionality, i.e., the Trojan horse can record discussions of an internal meeting (as audio and video files) and then send that information back to the attacker. In addition, the current cell ID of the user can be recorded and then later be sent to the attacker. This way, a tracking attack is possible. With the help of the auto invocation functionality (see Section 2.1), the Trojan horse can be started remotely by an SMS. The attacker can then decide when to eavesdrop on the victim. To sum up, the Trojan horse eavesdrops on the user violating the user's privacy and obtains sensitive information. The user does not have any suspicion for that privacy violation.

We also implemented a program for a mobile phone or PC, which remotely controls mobile phones infected by our Trojan horse (see Figure 3). The Trojan horse on the hacked mobile phone then acts as a server, whereas the control program is the client. The Trojan horse can be started by the auto invocation functionality from the client software.

Beyond the spyware functionality, we also carried out other attacks such as corrupting system files. This way, the mobile phone was made unusable such that the firmware had to be reinstalled. Moreover, we changed an entry in the aforementioned internal table of the AMS in a way that the Trojan horse cannot be uninstalled by the end user. If the end user wants to delete the Trojan horse, he again must flash the firmware onto the phone.

Infection with the Trojan horse. There are several ways mobile phones can be infected by the Trojan horse. One such vehicle is Bluetooth [31]. For example, so-called Bluetooth hot spots are often installed on airports, fairs, or cinemas to place advertisements in the form of mobile content such as Java MIDlets, sound or video files on the end users' mobile phones. An attacker can install faked hot spots luring the victim into a trap; the user might assume that it is information about a new movie, to give an example. Employing the Java Bluetooth API, the attacker can search for

Bluetooth-enabled mobile phones and send the Trojan horse to those phones. Infecting mobile phones through Bluetooth has been well-demonstrated [31], but there are other ways of infection which do not demand the proximity of attacker and victim. This will be discussed now.

One further possibility is spreading via the MMS functionality. There are examples of malicious code using that way of infection [19, 25]. Interestingly, some operators such as Vodafone and T-Mobile allow Java MIDlets to be installed from an MMS. The Trojan horse might also be distributed on the Web or sent via e-mail, and then be installed on the mobile phone similarly to the PC world. At least, the e-mail client implemented on the mobile phones we investigated supports Java applications as attachments.

If an attacker is interested in attacking a specific person or organization, then an infection by means of storage cards is conceivable. Storage cards are used to exchange pictures, music, and video files. Then the attacker can automatically install the Trojan horse on the victim's storage card.

Exploiting the Vulnerability to Build a Worm. Due to the fact that the vulnerability allows an attacker to open arbitrary network connections without further confirmation (beyond that shown in Figure 2) the malware can propagate per e-mail, i.e., the code can be sent through an SMTP connection. In this case, the attacker can exploit a feature that is offered by some operators, namely, that e-mail addresses are built from phone numbers. Then, the attacker can read the phone list and try to send e-mails to the respective e-mail addresses. As mentioned above, using mail on mobile phones is becoming more and more convenient and the latest models of mobile phones let a user easily install Java MIDlets. Due to the platform-independence of Java the malware can run on various (vulnerable) models and hence propagate. In the next step, it is also conceivable that an attacker takes over thousands of mobile phones and builds a "mobile" botnet. This way, a DDoS attack on a mobile phone network can be carried out by flooding the network with SMS and MMS messages [29].

4. DISCUSSION

The aforementioned security hole has several implications, which will be discussed in the following sections.

Impact of the security bug. We analyzed a mobile phone platform which has not been the main target of an attacker before. The vulnerable phones employ the real-time operating system Enea OSE [5], and not the supposedly ubiquitous Symbian OS. However, it is mostly unknown that Enea OSE is also used in many models of other mobile phone vendors such as Nokia and Samsung and even in base station equipment. Specifically, the company Enea AB stated in March 2008: "Enea OSE is a feature rich commercial RTOS deployed by half of the world's 3G mobile phones. The software powers 350 million of the new mobiles delivered during last year, a number equivalent to 30 percent of the 1,2 billion mobiles which were sold that year..." [6]. As a consequence, security flaws in such a widely used mobile phone OS might have a larger impact than expected. Specifically, it is conceivable that the unmediated link also exists in mobile phones of other vendors. However, the J2ME implementations made available by these vendors, were not vulnerable because they did not support the FileConnection API at that time.

The Mobile Java vulnerability has been detected in various Sony Ericsson mobile phone models such as K750i and K 880i. It is expected that most of Sony Ericsson's mobile phones produced between 2005 and end of 2007 are affected by this security hole, all in all more than 100 million devices. As a reaction to our vulnerability report, the manufacturer fixed (at least) the Java problem in the latest models not yet shipped to customers at that time. This way, those models are not affected by the security hole.

Different attackers. Another point is the fact that the security hole can be exploited by different attackers. We now discuss the consequences from the point of view of the different attackers. As demonstrated above, the end user can be attacked by Trojan horses. The aforementioned Trojan horse gives only a small flavor what is possible. In principle, also newer security-relevant Java APIs such as the SATSA API (see Section 2.1) can be employed by the Trojan horse. Specifically, those APIs are implemented in the latest generation of Sony Ericsson mobile phones. This way, security-critical transactions can be manipulated. With the help of the SATSA API, commands can be sent to applications of SIM cards, and functions such as `changePin()` or `enterPin()` can be called. However, we checked that the new generation of mobile phones is not vulnerable. The manufacturer was able to fix the security hole before the release of the latest phone models.

The user can also act as an attacker. As indicated above, the DRM copy protection of Java games and programs can be circumvented. Beyond copying Java applications for mobile phones, it is conceivable that also the DRM copy protection for music files can be broken. This would be the case if AES keys, used for the encryption of the music files, are stored on the internal file systems.

Not only can DRM mechanisms be circumvented, but also excessive operator permissions can be gained. This may have an impact on billing for value-added services, provided by the operator. For example, if an operator makes available specific Java Card [27] applications on a SIM card, an attacker can install a MIDlet on the mobile phone and then add that MIDlet to the Operator Domain. Thereafter, he tries to access the Java Card application on the SIM card by calling the method `exchangeAPDU()` (see Section 2.1). If the Java Card application is responsible for billing services, the attacker can committ fraud. In the specification for the SATSA API several scenarios for using that API are discussed [12]. For example, a user can download a Java MIDlet from the operator making available various services. Whenever the user accesses those services, a loyalty counter on the SIM card is incremented by means of the corresponding Java Card application. When the loyalty points of the user reach a predefined level, the user earns free minutes. If an attacker now gets access to that Java Card application, he can abuse that loyalty application.

Another attack on an operator is a "debranding" (unlocking) attack, say, a mobile phone formerly tied to Vodafone can now be used in the O2 network. In fact, the link files are used by operators to make their ring tones and logos readable from the external file system and at the same time to protect those files from being overwritten. Technically, links exist in the external file system which point to the ring-tones and logos stored in the internal file system.

Problems of patching mobile phones. Some remarks are to be made on possible reactions of the manufacturer, and on the process of patching the phones. Certainly, the security hole can be fixed in the firmware of the mobile phone, implementing an appropriate access control mechanism or at least fixing the Mobile Java vulnerability. However, since the JVM is part of the firmware as discussed with the manufacturer, the firmware and not only an application must be patched in this case.

The manufacturer was able to fix the Mobile Java problem in the latest models which have not been delivered to the customers at that time. However, patching all or most of the already deployed mobile phones is hard because most of the end-users will never notice that their phones are vulnerable and do not take care of patching their phones. Furthermore, this would also mean that the process of installing the firmware should become more comfortable than today. Recalling all vulnerable phones would also not be an adequate solution for the manufacturer because this would be by far too costly. Given the situation that the operator has often remote access to their customers' mobile phones, one solution would be that the operators install the new firmware on their customers' mobile phones (with the consent of the end users). Even if there is an adequate patch distribution mechanism for mobile phones (as for the iPhone), still the question remains "Who does pay the costs for downloading the patch?"— the end user might obviously not be willing to do this. Moreover, in some countries legal issues would prohibit a patch distribution solution via an operator.

Awareness aspects. The security hole demonstrates the dangers of the connection of telephone and PC functionality. At this time, the risks of mobile phones are not completely understood by the end users. In order to assess the full functionality of mobile phones, the end user should be adequately trained or informed about the risks related to mobile phones. Furthermore, the security warnings should not be misleading as in the Sony Ericsson case. Other mobile phone platforms, however, have similar problems. For example, Android phones present the end user a list of permissions for confirmation on installing an application. Strictly speaking, for each listed permission, the end user should know the consequences of granting the permission in question, specifically, if there are listed both relatively harmless permissions such as activating the sleep mode and very sensitive permissions such as unrestricted Internet access.[2]

In general, the aforementioned attacks demonstrate the specific risks of mobile phones: Mobile phones can be used to conveniently eavesdrop on the victim because we tend to always take mobile phones with us and often have them switched on. In contrast, PCs or even laptops are not so ubiquitous and not switched on for such long periods of time. Nevertheless, it should be stressed that we do not claim to abandon the additional functionality and give up all benefits of such new technologies. However, an appropriate strategy to assess the risks of mobile phone platforms for all the different stakeholders (such as end users, enterprises, operators) is necessary.

[2]This leads to another Social Engineering attack, requesting some harmless-looking self-defined permissions and hiding this way a security-critical permission such as Internet access.

Software quality of mobile phones. Another question is how such a kind of security hole can arise. On the one hand, we have an internal mechanism, the link file concept, having been introduced in early mobile phone generations. On the other hand, new functionality is added such as the FileConnection API. The combination of the (hidden and undocumented) link file concept and the newly available FileConnection API leaves the door open for a Trojan horse that breaks out of the sandbox of Mobile Java.

The security flaw described in this paper helps quality testers of mobile phones in finding similar security problems. Given the unsecure link concept of the underlying operating system (or possibly the unsecure access control mechanisms of the file system), special care must be taken on those Java APIs allowing file access. If those APIs are not checked appropriately, they can be conveniently exploited by an attacker to access security-critical system files, leading to a complete take-over of the mobile phone.

In order to prevent a Trojan horse from overwriting system files such as manufacturer certificates, these files can be stored in a protected area. ARM processors, which are usually incorporated into mobile phones, support the ARM TrustZone technology [15]. This way, access to security-critical files is only possible through secured API calls (TrustZone API). The implementation of the protection domain mechanism must then be adjusted, i.e., the module responsible for checking the certificates would reside in the TrustZone, which can only be accessed via the TrustZone API. In the end, however, this approach only cures the symptoms. The main problem still lies in faulty software. The Java hole (and more generally, similar vulnerabilities in Java APIs for mobile phones) can be avoided by integrating security into the software development process [16]. Discussions with the manufacturer revealed that a methodology for the security analysis of Mobile Java platforms was desirable.

5. STATIC SECURITY ANALYSIS

We now sketch an approach to statically analyzing security mechanisms of Mobile Java plattforms. This approach comprises JML-related analysis tools in conjunction with a reverse engineering tool, which helps one in extracting security-relevant components from the source code.

JML-based analysis of Mobile Java. The basis of our methodology is JML which in general is useful for specifying detailed designs of Java classes and interfaces [2]. JML can detail the pre- and postconditions for methods as well as class invariants in the Design by Contract (DBC) style [18]. JML has the advantage that its syntax is tailored towards Java, i.e., the JML constraints can be specified as Java comments that are ignored by a conventional Java compiler. In addition, there is a variety of tools available that allow one to check the JML constraints at run-time or even (in part) statically [2]. These tools check that the code corresponds to the JML specifications. In the following, we use ESC/Java2 which statically can detect inconsistencies between the code and the specification using a built-in theorem prover [2]. ESC/Java2 employs modular reasoning, analyzing one method at a time and considering the specifications (pre- and postconditions) rather than the whole code of the methods being called.

We can employ JML to specify security requirements for J2ME APIs, and check at compile-time that the implementation satisfies the requirements. Specifically, defining a class invariant which forbids that the special characters occur in a file name can reveal the Java security hole described in this paper. Note that invariants must hold at the end of each Java constructor's execution and at the beginning and end of *all methods*. Hence, we even do not need to define a pre- or postcondition for the `rename()` method because the class invariant already encompasses that condition. This would have helped the developers at Sony Ericsson to avoid the Java hole even if they missed considering `rename()`. Since Sony Ericsson secured other APIs such as `write()` appropriately, they knew in principle that Java APIs were not allowed to create links—they only failed to secure `rename()`.

The reference implementation for the FileConnection API, for example, has a private field variable called `fullPath` of type `String` which contains the file name. This variable can be used to specify the aforementioned class invariant:
`//@ invariant !fullPath.contains(".@")`.
Instead of the black-listing approach, which filters out forbidden characters, white-listing specifications should be used. Then we can specify rules (as invariants) which file names must satisfy.

In general, the security mechanisms for Mobile Java (e.g., the implementation of the protection domain concept) can also be specified in JML and then analyzed by JML tools, i.e., the approach is not restricted to the specific Java problem discussed in this paper. This even more applies when security-critical Java Card applets are accessed from MIDlets such as in the case of the SATSA API. For instance, Gemalto, one of the world-leading smart card producers, makes available the SIM Application Toolkit. MIDlets can then call SIM Application Toolkit functionality via the SATSA API [8]. In this case, JML tools can be employed to detect API-level attacks on Java-based SIM cards [1].

The aforementioned approach can also be applied to other Java-based mobile phone platforms. Two interesting candidates are the BlackBerry platform [24] and Android [9], both supporting Java. Due to the fact that large parts of Android are made available as open source software, an analysis of the source code is possible (in contrast to the proprietary system of Sony Ericsson) and hence Android is an interesting target for an external security analysis. Specifically, the Java-based Android middleware implements a reference monitor to mediate access to application components based on permission labels. Security holes in this reference monitor could be exploited to circumvent the whole compartmentalization mechanism of Android [4].

Architectural-level analysis. JML tools help in analyzing Mobile Java's security mechanisms as discussed above. However, one problem with static analysis tools like ESC/Java2 is that they do not scale. In particular, this applies to invariant checking, which must be carried out for *all* methods and constructors. Hence, the analysis of a complete Java-based platform is unrealistic, specifically, if dataflow conditions such as the aforementioned invariant are to be considered [14]. In order to make a JML-based analysis feasible, it is hence necessary to extract security relevant functionality from the source code to simplify the dataflow analysis. In general, a more abstract view on the code is desirable, e.g., to carry out analyses directly at the architectural level. For these purposes, we propose to employ the functionality of a reverse engineering tool such as Bauhaus [23].

In particular, Bauhaus lets one deduce the resource flow graph (RFG) from the source code [23]. The RFG works at a higher abstraction level than the source code and represents architecturally relevant information of the software, i.e., it refers to the implemented software architecture. The nodes of the RFG represent types (e.g., classes and interfaces), constants, member variables, and methods. Edges represent various relationships between the nodes such as the call graph relation, the types of a method's signature, and *Use* and *Set* relations, which indicate whether a member variable is used or set within a method. Data and control flow analyses can be conducted on the RFG.

With the help of graph search algorithms on the RFG, one can create subgraphs of the RFG (so-called "views"), i.e., we conduct slicing at the architectural level. For example, we can select data types, member variables, and methods, which are used to implement the security functionality under investigation (e.g., enforcement mechanisms). The resulting "security views" are then used as input for the source code analysis step employing ESC/Java2. This makes the source code analysis more tractable.

6. RELATED WORK

There exist works concerning vulnerabilities of mobile phones [19, 25, 30, 31]. Some vulnerabilities are caused by the interaction between mobile devices, the Internet, and telecommunications networks [31]. Other mobile phone security problems are related to earlier versions of Symbian when there was not much security incorporated before the advent of the Symbian Signed initiative [28]. Moreover, well-known Symbian worms such as CommWarrior [19] mostly spread only via unsecured Bluetooth connections or MMS. Other ways of infection such as e-mail were not exploited at that time because e-mail was not widely used on mobile phones. In contrast, we attacked a platform which seems to be supposedly secure due to Java's sandbox model. Thus, the level of security should be similar to the Symbian Signed system. In fact, not many attacks are known on Mobile Java with the exception of Gowdiak's attack on the byte code verifier of Nokia mobile phones [25] although Java security problems in general are well-demonstrated [17]. Recently, security holes in Android's Java-based middleware are reported [21].

There are also relations to Unix security, in particular, Unix systems protect their symbolic links for a long time [7]. This way, the security hole demonstrates that old problems solved in the desktop PC and workstation world crop up again in new systems such as mobile phone platforms.

Our work can also be compared with works on code review tools, which, for example, are discussed by Chess and West [3]. Available (commercial) code review tools mostly deal with common classes of security bugs such as buffer overflows, race conditions, and SQL injection vulnerabilities. In contrast, we focus more on domain-specific security problems (e.g., for Java-based mobile phone platforms). Another difference is that we additionally carry out our analysis at the level of the software architecture. This helps to pinpoint security-critical components of the software which thereafter can be analyzed in more detail to make an analysis with JML feasible. In addition, our analysis approach also resembles the work by Poll et al. in which the Java Card API was specified in JML. Thereafter, a component of a smart card operation system was verified by means of JML's run-time assertion checker [22].

7. CONCLUSION AND OUTLOOK

By means of a security hole, we discussed the dangers that may arise from mobile phones, specifically, if phone functionality is combined with PC functionality. Due to the fact that the functionality and the computational power of mobile phones are still growing the risks will also increase, and software security of mobile phones becomes an important issue. In particular, a broken link concept together with an error in the FileConnection API led to a security hole giving access to the entire internal file system. The hole can be conveniently exploited using various security-critical Java APIs. We demonstrated this with the help of a Java-based Trojan horse. In this scenario, the privacy of the end user was attacked without the victim's knowledge.

Due to Java's platform independence, most of Sony Ericsson's mobile phone models are vulnerable such that a large number of mobile phones can be attacked. Due to the fact that other vendors also employ Enea OSE, mobile phones of those vendors might at least also suffer from the unmediated link vulnerability. Last but not least, the hole discussed in this paper should be regarded as a case study; most of the remarks would apply to future security holes and vulnerabilities detected in mobile phones or smart phones in general.

There is room for further research. To tackle at least the Java security issues of mobile phone platforms we intend to employ JML and related tools in a follow-up research project. Specifically, we can investigate how MIDlets and Java Card applications for value-added services interact in order to avoid security holes introduced at the API level. In this context, domain-specific security rules can be defined as JML annotations, which can be checked by supporting tools such as Bauhaus and ESC/Java2. More generally, the interplay between the security and reliability of telecommunications infrastructures and mobile phone platform security is also worth investigating.

8. REFERENCES

[1] M. Bond and R. Anderson. API-level attacks on embedded systems. *Computer*, 34(10):67–75, 2001.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.-T. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. In *Proc. 8th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, pages 73–89, 2003.

[3] B. Chess and J. West. *Secure Programming with Static Analysis*. Addison-Wesley, 2007.

[4] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[5] Enea AB. OSE 5.0 Architecture, 2004. http://www.enea.com.

[6] Enea AB. Enea Wins New Wireless Deal Worth MSEK 30, 2008. http://www.enea.com/Templates/NewsPage____24486.aspx.

[7] S. Garfinkel and G. Spafford. *Practical Unix and Internet Security*. O'Reilly, 2nd edition, 1996.

[8] Gemalto S.A. Developer Suite V 3.0, 2007.

[9] Google Inc. Android—An Open Handset Alliance Project, 2008. http://code.google.com/android/documentation.html.

[10] M. Hypponen. Malware Goes Mobile. *Scientific American*, 295(5):46–53, 2006.

[11] Java Community Process. List of all JSRs, 2007. http://jcp.org/en/jsr/all.

[12] Java Community Process. Security and Trust API for J2ME, 2007. http://jcp.org/aboutJava/communityprocess/jsr177.

[13] JSR 118 Expert Group. Mobile Information Device Profile for the Java 2 Micro Edition Version 2.1, 2007.

[14] J. Kiniry. Personal communication, 2009.

[15] ARM Limited. ARM Security Technology Building a Secure System using TrustZone Technology. *White Paper*, 2009.

[16] G. McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006.

[17] G. McGraw and E. W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. Wiley, 2nd edition, 1999.

[18] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[19] Nokia. Malware CommWarrior, 2005.

[20] Open Mobile Alliance. DRM Content Format Approved Version 2.0, 2006.

[21] Open Source Cert Advisory. #2009-006—Android improper package verification when using shared UIDs, 2009. http://www.ocert.org/advisories/ocert-2009-006.html.

[22] E. Poll, J. van den Berg, and B. Jacobs. Specification of the Javacard API in JML. In *Proceedings of the Fourth Working Conference on Smart Card Research and Advanced Applications*, pages 135–154, 2001.

[23] A. Raza, G. Vogel, and E. Plödereder. Bauhaus—A tool suite for program analysis and reverse engineering. In *Ada-Europe*, volume 4006 of *LNCS*, pages 71–82. Springer, 2006.

[24] Research in Motion. BlackBerry Enterprise Solution – Security Technical Overview, 2008. http://www.blackberry.net/products/software/-server/exchange/security.shtml.

[25] S. Shankland. Mobile Java Hit with Security Scare, October 2004. CNET News.

[26] Sun Microsystems. Connected Limited Device Configuration Specification Version 1.1, 2003.

[27] Sun Microsystems. Java Card 2.2.2 Platform, 2006. http://java.sun.com/products/javacard/specs.html.

[28] Symbian Ltd. Symbian Signed, 2006. https://www.symbiansigned.com.

[29] P. Traynor, P. McDaniel, and T. La Porta. On Attack Causality in Internet-Connected Cellular Networks. In *Proceedings of the USENIX Security Symposium (Sec'07)*, August 2007.

[30] P. Traynor, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta. From mobile phones to responsible devices. Technical report, Pennsylvania State University, Network and Security Research Center, January 2007.

[31] trifinite.org group. Homepage, 2006. http://trifinite.org.