How to unwittingly sign non-repudiable documents with Java applications*

D. Bruschi, D. Fabris, V. Glave, E. Rosti
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano, Italy

Abstract

Digital signatures allow us to produce documents whose integrity and authenticity, as we generated them, is verifiable by anybody who has access to our public key. Furthermore, we cannot repudiate those documents as something we never saw, let alone signed, since nobody else but us could access our private key.

In this paper we show how the previous statement can be proved wrong when carefully crafted malicious software is installed on a machine running a Java digital signature application. By using such a software, a user may unwittingly sign another document besides the one he/she intends to digitally sign or sign a different document altogether. Our attack exploits a known vulnerability of the security architecture of the Java run-time environment that allows non-Java malicious software to replace some Java system classes with malicious ones, which then alter the victim application behavior.

1. Introduction

In the cyber-world, as well as in the real world, non-repudiation, i.e., the impossibility to deny having done something such as sending an e-mail message, writing a document, or committing any sort of transaction, is a very important property. Non-repudiation is critical for services on-line such as e-commerce and home banking, as it contributes to the reciprocal trust in the system of the parties involved. The introduction of public key cryptography [3] made non-repudiation possible in the electronic world through digital signature. From a mathematical point of view, a digital signature is a digest encrypted using the private key of an asymmetric key pair in a public key cryptographic system. In a computer, they are performed as follows. A digest of the document to be signed is computed using a one-way hash function. The digest is then sent to the cryptographic engine, either a dedicated device such as a smart card or the local CPU itself, where it is encrypted with the private key of the user. The encrypted hash is attached to the original document. Anyone that knows the corresponding public key may verify the integrity of the document and the identity of the signer. Because the private key is only accessible to its owner, a digital signature is deemed non-repudiable.

A potential security flaw in the digital signature process described above was recently pointed out by some authors [4, 2]. It was hypothesized that in a standard non-trusted computing platform the entire digital signature process could be subverted. In such a case, the document digest could be intercepted before its encryption and covertly replaced with the hash of a different document unknown to the user. The user would then unintentionally sign a document he/she is not aware of. The covertly signed document would then be stored on the local computer or distributed to a remote one, resulting in a digitally signed document whose content is unknown to the author, i.e., a document with a "non-repudiable" ownership proof the author is unaware of.

In this paper we show that the potential security flaw can indeed be turned into a feasible attack. Bt leveraging well known weaknesses of the Java2 platform [7], we implemented an attack to a commercial digital signature application, which allows an attacker to obtain documents digitally signed by a user, without the latter knowing it.

The Java weaknesses exploited in our attack are related to the possibility of modifying classes of the Java Virtual Machine, in particular classes contained in the `rt.jar` archive. When such an archive is not adequately protected by the operating system, as it is the default case on a large number of systems, the classes in the `rt.jar` archive can be modified or simply substituted with malicious ones.

A purposefully written email virus or worm can be used to bring the malicious classes on the victim system and install them in the Java run-time environment in the infection phase by exploiting the weakness mentioned above. The malicious classes are responsible for altering the behavior of any Java application. In our case, they intercept the event triggered to confirm the start of a signature operation and

* This work was partially sponsored under the Italian Dept. of Education and Research F.I.R.S.T. project.

make it possible to perform the authentic signature operation on the legitimate document and on additional ones provided by the attacker. Note that the attack does not compromise the victim application, as it might be signed by the vendor. The `rt.jar` archive is the target of the only modifications introduced. This means that even an integrity check of the application will not be able to reveal the presence of the malicious code. Note that the attack could be easily avoided by forcing users to adopt security practices, such as protecting system directories from unauthorized write operations or adopting integrity checkers to verify the integrity of system files. Unfortunately, average users are not willing to, or not aware of, adopt such countermeasures, thus exposing their systems to the vulnerability described in this paper.

The result presented in this paper is a tangible evidence of the concept that, as an operational process, "digital signature is not secure" [5]. Although the security community has been aware of this and has pointed it out since the emergence of the technology, other communities, such as the legal one, seem to have overlooked the problem. Several countries have enacted legislations regulating digital signature, based on the assumption that digital signatures cannot be compromised. Our goal with the work presented in this paper is to contribute to raise awareness and understanding of the technical limitations of the digital signature technology. Hopefully, current legislations could be adjusted to take into account the inherent limitation of the technology.

This paper is organized as follows. Section 2 briefly recalls some aspects of the Java2 platform and the Java2 security architecture that can help readers not familiar with Java to understand how the exploit works. Section 3 describes the attack and Section 4 presents the implementation targeted at a digital signature commercial application. Section 5 discusses possible countermeasures. Section 6 summarize our findings and concludes the paper.

2. Background

In this section we recall some basic notions regarding the execution of a Java application, which will be used to describe the strategy we adopted to subvert the behavior of our victim application.

Several Java applications are designed according to the event-driven paradigm. For example, every time a user interacts with a graphical component, the corresponding event is triggered and handled by a particular object called *event handler*. Any Java object may be used as an event handler. In order to do this, its class must implement an event listener interface[1], which must be associated with the components whose events are to be managed [1]. As an example, in order to listen to window events, an object must be registered with that window by calling the `addWindowListener()` method and its class must implement the `java.awt.WindowListener` interface. In the following we will refer to the component where an event occurs as the "event source" and to the object which listens to the event as the "event listener". In the Java run-time Environment, all the events triggered by an application are placed on a single system event queue. Such a queue is an instance of the system class `java.awt.EventQueue`, which provides a method that dispatches events in FIFO order. When an event reaches the head of the event queue, the dispatcher notifies the event to all the event listeners registered to receive such an event.

The security architecture of Java provides the necessary protection against possibly malicious mobile code, which is typical of the language thanks to its machine independence. Java applications are compiled in an intermediate code, known as bytecode, which is independent from the underlying system architecture. The bytecode is loaded and executed by the Java Virtual Machine (JVM). With the Java2 platform, code sandboxing and digital signature are combined to provide protection against malicious software. Before execution, any untrusted code is assigned by the JVM to a particular sandbox, i.e., a restricted environment that specifies the set of permissions granted to the code for accessing local resources. Every sandbox defines a security domain. At run-time, the security manager and the access controller guarantee that the application behaves properly and monitor its access to the resources. Figure 1 illustrates the Java 2 security architecture.

When a Java application is launched, the Java Run-time Environment (JRE), i.e., the Java Virtual Machine and its core classes, loads the class that contains the main method of the application and executes it. Then, all the classes instanced by the application are loaded into memory. During such a process the class loader builds in the computer memory an object for the class that needs to be loaded. Note that in the JRE, a hierarchy of class loaders is used. At the top of the hierarchy is the BootstrapClassloader, which is responsible for loading the JVM core classes. One level below is the ExtClassloader, which is responsible for loading the extension packages, i.e., classes that programmers may use to extend the functionality of the JVM. At the bottom of the hierarchy is the AppClassloader, which loads the classes defined by the applications. The loading operation can be customized by the user, according to a delegation model (see [6]).

The class loader assigns each untrusted class to a particular sandbox depending on the credentials of the class. Classes loaded by the BootstrapClassloader (i.e., JVM core classes such as those distributed in the `rt.jar` archive)

---

1 Different types of listener interfaces are available for different types of events.

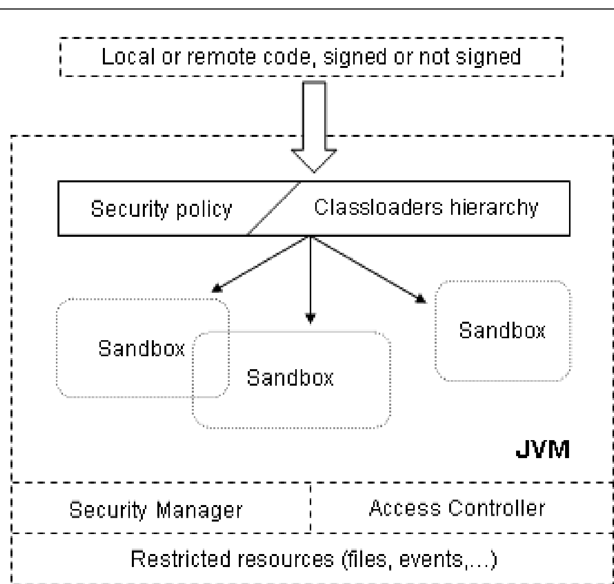

Figure 1. Java 2 security architecture.

are instead considered trusted by definition and therefore all rights are always granted to them. Java code can be digitally signed to guarantee its origin. Such a signature is a parameter used by the class loader to define the association between a sandbox and the Java code itself.

3. The attack

Our strategy for compromising the behavior of a Java application is based on the consideration that in general Java applications are event driven. Therefore, in order to modify their behavior it is sufficient to modify the application event handlers or, if this is not possible, modify the system event dispatcher. The idea is that of intercepting events when they happen, and force the dispatcher to notify them first to a malicious event handler, before -or instead of- passing them to the application handlers. In order to implement such a strategy, the dispatcher has to be modified. Since it is a method of the system event queue class, a new class must replace the original one. This is possible since Java allows programmers to create a new event queue and push it on top of the current system event queue. Hence, by redefining its methods, in particular the `dispatchEvent()` method, the new system event queue will be able to handle all events before they are notified to the registered event listeners. This way it is possible to modify the application response to each action of the user. The malicious code that modifies the system event queue must have sufficient permissions, otherwise a security exception is thrown. By exploiting the weakness of the Java environment mentioned in Section 1, such a code can be written and made executable on platforms with default configuration. We now outline how this can be done.

In the vast majority of the hosts, the Java system archive `[java_home]/lib/rt.jar`, which contains most of the JVM core classes, is installed in a writable directory. Furthermore, such an archive is not signed, thus it can be easily tampered with. An external non-Java agent is used to replace some core classes of the JVM with malicious ones. Since these malicious classes will be loaded by the BootstrapClassloader, they are granted *any* permission regardless of the security policy in use, thus they may perform any action on any other class.

A general scheme to compromise any (event driven) Java application is as follows.

1. Replace, via a non-Java application, the AppClassloader contained in the `[java_home]/lib/rt.jar` with a malicious one, which installs the malicious system event queue before loading the main class of the victim application. The malicious queue class is inserted in the `[java_home]/lib/rt.jar` archive too and therefore is loaded by the bootstrap classloader, enjoying all permissions including those for event management. The malicious AppClassLoader redefines the `loadClass()` method, as depicted in Figure 2, to substitute the event queue for the application with the new event queue.

```
protected Class loadClass(String s, boolean flag)
throws ClassNotFoundException{
  ...
  if(s.equals(Name of the main class)){
    Toolkit.getDefaultToolkit().getSystemEventQueue().
    push(New_event_queue);
  }
  ...
  return super.loadClass(s, flag);
}
```

Figure 2. Source code of the malicious `loadClass` method.

2. Identify the events of the applications to be intercepted and redefine the dispatch method of the queue class accordingly. Such a modification can be easily performed using a code like the one reported in Figure 3.

3. Define the malicious event handlers that perform the attack.

```
protected void dispatchEvent(AWTEvent awtevent){
  ...
  if(awtevent = event_to_intercept){
    Action_to_perform
    ...
    }
  ...
  }
```

Figure 3. Source code of the `dispatchEvent` method.

4. The case study

We now describe how we applied the general scheme presented in the previous section to a digital signature commercial application. We selected a commercial product that uses (E3 ITSEC certified) smart cards . The test was performed on a Windows2000 platform running service pack3 installed with defaults values, thus no "write" restriction was applied to any directory.

In order to instantiate the attack scheme, some details of the victim application must be known. In our case, we need to find the event that is generated when a user confirms the signature operation on a document, as such an event has to be intercepted by the dispatcher in order to start the malicious activity. We also need to find the name and the prototype of the method called by the application to generate the digital signature, as the malicious software will have to call it to have the other documents signed by the unaware user. Such information can be obtained by reverse engineering the application behavior during its execution. The order in which its classes are loaded, methods are called and events are fired at run-time must be traced. Alternatively, decompilers can be used to produce source code starting from bytecode.

From the analysis performed on the target application we recovered the following information:

- The event fired just before the digital signature operation starts is the event associated with the mouse click on a "confirm" button.
- All the information about the current smartcard session is stored in a `SmartcardRepository` object.
- In order to digitally sign the document, the application calls a public method, `modularclient.signencrypt.CtrlSignEncrypt.sign()`, which is accessible from any other class. Such a method references two objects: a `java.io.InputStream` related to the document to be sign, and a `SmartcardRepository` object.
  The method returns a `crypto.server.SignedDocument`, which is then used to save the signed document.

We also have to consider the constraint of using the `SmartcardRepository` object that is declared inside a method, so it cannot be directly referenced. Since such an object is required by the signature method, we have to find a way around this problem. We slightly modify the first step of the attack adding the following functionalities. Besides the application class loader, we also replace in the `[java_home]/lib/rt.jar` archive the class `java.lang.Object`, the ancestor of all system and application classes. The skeleton of the modified class is reported in Figure 4. As it can be noticed, such a class redefines its constructor in order to store `SmartcardRepository` instances in a public static variable that can be referenced by any other class. With such modifications, when the digital signature application is executed, the malicious code will work as follows.

```
public class Object{
    ...
    public static Object repository = null;
    public Object(){
        ...
        if(this instanceof SmartcardRepository){
              repository = this;
        }
    }
    ...
```

Figure 4. Source code of the malicious class `Object`.

When a signature event happens, the malicious dispatcher calls the method `modularclient.signencrypt.CtrlSignEncrypt.sign()` used by the application to generate the digital signature of the document. The parameters passed to the method are the covert document to be signed by the user unwittingly and an instance of the `SmartcardRepository` object, saved by the malicious `java.lang.Object` class previously installed in the environment. The result is the covert document with a legitimate signature, which is saved in a hidden directory. The malicious code then proceeds with the execution of the intended digital signature operation, as chosen by the user. Since the two signature operations are strictly consecutive, no additional logging session is re-

quired and the user does not notice anything[2].

The malicious code that we have just described is intended to be installed in the environment by another malicious module that may spread itself, for example, as an e-mail attachment or as a worm. Several variations of the exploit are possible, while keeping its core structure, depending upon the behaviour of the victim application. For instance, it is possible to substitute the document originally selected by the user with a different one, instead of producing a second signed document.

5. Countermeasures

It is important to notice that our attack may fail under particular conditions, since countermeasures may be in place to prevent it. First, it could be possible to prevent the `rt.jar` modification. As we previously noted, this kind of attack needs to modify such a system archive on the target computer and to intercept all graphical events triggered by the victim application. The following scenario can prevent the attack from succeeding. If the "write" privilege on `rt.jar` (or on the directory that contains it) is denied, the infection process cannot substitute any Java system classes and so the attack fails. This can be done only on operating systems that allow to protect files and/or directories, such as Windows2000 or all Unix/Linux flavors.

An alternative countermeasure is to check whether the Java system archive has been tampered with. Integrity checking should rely on cryptographic properties, such as digital signatures or hash functions. Moreover, to be effective it should be automatic, for example before application start-up and it should not be Java-based. Although the use of integrity checkers is a well known security practice, few users systematically adopt it. Since we cannot expect users to care for their security, the most reasonable approach to checking the `rt.jar` integrity is to automate this operation.

6. Conclusions and Future Work

In this paper we presented an attack against Java applications, which exploits a weakness of the Java2 platform that allows to modify classes of the JVM, if they are not adequately protected by the operating system. By combining such a weakness together with the event driven programming paradigm that characterizes most Java applications, an adversary may change the behavior of any Java application by changing the system classes that dispatch the events to the application. In particular, we have shown the applicability of the attack to a commercial application for digital signature. The malicious software we developed allows an adversary to have a victim user digitally sign a document without realizing it, whenever the users legitimately sign a document of their choice.

This kind of attack could be performed in any environment where the use of digital signature is appropriate, such as Public Administration offices, home transactions, remote banking, trading on-line ad other web centered activities. Therefore all of these activities are vulnerable to the unwitting digital signing process. Luckily, such an attack cannot be easily launched on a mass scale, thus limiting its power, since the document to be unwittingly signed by the user may be related to the user. This requires customizing the attack to each user, thus hindering its mass deployment.

We are currently investigating the possibility to export our attack to applications written in other languages, such as C, that compile into native machine code. If the application relies on dynamically linked libraries, it seems possible to replace them with malicious ones.

We wish the results presented in this paper contribute to stimulate the discussion on the security of digital signature. Digital signature laws have been enacted in many countries all over the world based on a complete trust in the impossibility to counterfeit or manipulate a digital signature, except with the willing, negligent or coerced help of the user. Here we have shown that even honest users can be deceived and unconsciously sign documents they have never even seen. The signature is correct and the user cannot deny having signed such documents. If a digital signature has the same legal value as an autograph, provision should be present in the law to address cases such as the one just described.

2 A careful user may notice a slightly longer delay for the signature operation but only accurate timing comparisons may show a difference.

References

[1] Java2 platform standard v1.4: API specification. Technical report, Sun Microsystem, 2002.

[2] B. Balacheff, L. Chen, D. Plaquin, and G. Proudler. A trusted process to digitally sign a document. In *Proc. of New Security Paradigms Workshop*, pages 79–86, 2001.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[4] M. Freundenthal, S. Heiberg, and J. Willemson. Personal security environment on palm pda. In *Proc. of 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 366–371, 2000.

[5] C. Kaner. The insecurity of the digital signature, September 1997. http://www.badsowftare.com/digsig.htm.

[6] S. Oaks. *Java Security*. O'Reilly, 2001.

[7] D. M. Wheeler, A. Conyers, and A. Xiong. Java security extension for a Java server in a hostile environment. In *Proc. of Annual Computer Security Applications Conference*, 2001.