Cozilet: Transparent Encapsulation to Prevent Abuse of Trusted Applets

Hisashi Kojima Ikuya Morikawa Yuko Nakayama Yuji Yamaoka

Secure Software Development Group
Fujitsu Laboratories Limited
{hisashi, ikuya, yamaoka}@labs.fujitsu.com, yuko.nakayama@jp.fujitsu.com

Abstract

We have developed a mechanism which prevents abuse of trusted Java applets, such as digitally signed applets. A signed applet is usually permitted by a user to perform certain functions. However, an attacker may improperly recompose the signed applet to include malicious components and harm the user by abusing such functions of a signed applet. In this paper, we call this a malicious recomposition attack and propose an innovative mechanism to solve the problem of such attacks. Before deployment, a target signed applet is encapsulated into a special signed applet, called a cozilet, in an indecomposable format. On its execution, the cozilet isolates the classes and resources of the encapsulated applet by assigning a special class loader unique to the applet. It also enforces applet-document binding so that it is never executed from untrusted HTML documents. The mechanism is easily applicable to target signed applets because it is transparent not only to target applets, but also to current Java VM implementations. Therefore, the mechanism can easily protect both applets developed in the future and the applets currently in use. We have implemented this mechanism for Sun Java VM. In this paper, we describe its basic architecture and implementation details.

1. Introduction

Web applications use Java applets to control client-side behavior. For security reasons, applets are not permitted to perform any behavior which may harm users. These restrictions sometimes become obstacles in Web application development. For example, various e-government systems ask users to digitally sign their application forms or registration forms using smart cards. This cannot be done with normal applets, because they cannot access client-side local devices, such as smart cards. To empower applets, applets whose origins are guaranteed are allowed to bypass the restrictions. A popular way to guarantee their origins is to digitally sign them. Signed applets can dynamically ask users for permission to bypass the restrictions, and provide an important benefit in that they do not oblige users to change their security settings in advance.

Unfortunately, attackers may reuse signed applets and harm users by abusing the functions of the applets. Signed applets consist of reusable components, such as signed JAR files [13]. Attackers can try to call methods or modify fields of classes in signed jar files by recomposing them as the attackers' own components for malicious purposes. In this paper, we call this a *malicious recomposition attack*. This issue applies not only to applets, but also to other signed reusable components, such as signed ActiveX controls. In contrast to server-side vulnerabilities, which can be mitigated by stopping vulnerable servers, it is difficult to stop the spread of damage caused by vulnerable client-side signed components because attackers may have already obtained them and can redistribute them for their own purposes. There is a pressing need to address this problem of malicious recomposition attacks.

We have analyzed the risks associated with signed applets and have considered various strategies that a malicious recomposition attack can follow. Although these attack strategies can be prevented through careful design and programming on the part of developers, most developers cannot easily understand and prevent all possible strategies. Developers need to ensure that some mechanism for signed applets lessens the possibility of a malicious recomposition attack. To easily apply such a mechanism to the signed applets now in use, however, the mechanism should not require changes to signed applets or the replacement of currently installed Java VM implementations. Therefore, we have developed the *Cozilet* mechanism which protects signed applets transparently by encapsulating them.

The rest of this paper is organized as follows. Section 2

describes the basic mechanism and typical examples of malicious recomposition attacks on signed applets, to show that current Java VM implementations are susceptible to such attacks. Section 3 describes our mechanism to protect signed apples from a malicious recomposition attack. This mechanism is transparent to both signed applets and current Java VM implementations. Section 4 describes the details of implementation. Section 5 discusses the issue in a more general manner, and Section 6 discusses related work. We conclude in Section 7.

2. Malicious recomposition attacks

To perform a malicious recomposition attack, an attacker needs to force a trusted applet to harm a user by recomposing the trusted applet with malicious components.

This section first describes an example of a signed applet. It then goes on to explain two typical attack strategies for malicious recomposition: exploiting an improperly implemented privileged code and class replacement.

2.1. Example of a signed applet

Suppose, for instance, there is a signed applet for e-commerce. It is digitally signed by a trustworthy company and allowed to perform potentially insecure behavior, such as accessing a smart card. The applet is composed of the signed JAR file `purchase.jar` and invoked by the applet tag below.

```
<applet code="foo.PurchaseApplet"
 archive="purchase.jar" ...
```

`purchase.jar` includes the applet main class `foo.PurchaseApplet`. Here, `purchase.jar` and an HTML document with the above applet tag would be deployed on a trusted Web site. If a user accesses the Web page (i.e., the HTML document) on the site, Java Runtime Environment (JRE) shows a security dialog to the user. The dialog displays the signer information of the applet and asks the user to grant the applet the permission required to perform the insecure behavior. In this example, since the signer is a trustworthy company, the user will grant it. The applet then runs and performs the insecure behavior.

2.2. Exploiting an improperly implemented privileged code

The first attack strategy is where an attacker uses classes of a signed applet as library classes. With this strategy, the attacking applet can access the classes of the target applet and execute their methods. This kind of access is usually prevented through stack inspection (see Section 2.2.2), but this is bypassed if there is a privileged code in the class and it is not implemented properly.

This subsection describes the strategy. First, the basic strategy is described. After that, the Java security model is described and it is shown that an attacker cannot usually use a target applet. Finally, the privileged code for bypassing stack inspection is described and it is shown that improperly implemented privileged code causes a security breach.

2.2.1. Abuse of a signed applet as a library. Suppose that an attacker creates an attacking applet which maliciously uses the classes in `purchase.jar` from Section 2.1. The attacking applet is unsigned and composed of the unsigned JAR file `evil.jar`. The applet main class is `EvilApplet` and is included in `evil.jar`. The attacker deploys the HTML document including the applet tag shown below, with `evil.jar` and `purchase.jar` on the attacker's own Web site.

```
<applet code="EvilApplet"
 archive="evil.jar,purchase.jar" ...
```

If the attacker can lead a user to the above page, the attacking applet runs and may harm the user. Although the security dialog in Section 2.1 is shown, the user will grant permission because the dialog shows the information of the trusted signer who signed `purchase.jar`.

However, the attacker normally cannot use the target applet, because Java uses stack inspection to prevent an attacker from performing any insecure behavior [14].

2.2.2. Protection through stack inspection. Java uses the *Sandbox* security model which prevents untrusted programs from accessing system resources. Java defines interfaces to system resources as system methods. In this paper, we call them *insecure methods*. Java also defines the corresponding permissions required for execution of insecure methods. The stack inspection checks whether each caller of a methods has the required permission [14].

When an insecure method is called, JRE checks the trustworthiness of classes in its method call chain by inspecting the stack of the current thread. If all the classes in the method call chain are trusted (e.g., classes of a signed applet) as shown in Figure 1(A), the insecure method is executed. Or if there is any single untrusted class (e.g., the classes of attackers) as shown in Figure 1(B), a security exception is thrown instead of execution of the insecure method. Therefore, an attacker's class cannot execute any insecure method, either directly or indirectly, via any method of trusted classes in a signed applet.

Unfortunately, an attacker is sometimes able to bypass the stack inspection by invoking privileged code in a signed applet.

2.2.3. Privileged code. The privileged code mechanism [14] is provided to enable a trusted class to delegate its

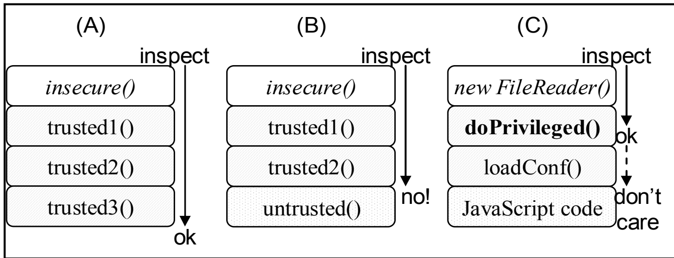

Figure 1. Stack inspection

execution of an insecure method to an untrusted class. In the case of signed applets, it is mainly used to accept accesses from script code running on a Web browser (which we will refer to as browser script code), because most Java VM implementations regard browser script code as untrusted code.

Suppose that the `foo.PurchaseApplet` from Section 2.1 declares the method `loadConf()` which is called by browser script code. A sample implementation of the method for Sun Java VM is shown below.

```
public String loadConf(final String name) {
 Reader r = // Begin privileged block
   (Reader)AccessController.doPrivileged(
    new PrivilegedAction() {
     public Object run() {
      try {
       // Open a text file
       return new FileReader(name);
      } catch (Exception e) {
       e.printStackTrace();
       return null;
 }}});      // End privileged block
 // Read the file and return string ...
  :
}
```

`loadConf()` executes the insecure method `new FileReader()` in the *privileged block*, which starts at `doPrivileged()`. JRE regards the privileged block and all the methods called within the block as privileged code, and simplifies the stack inspection as follows. When the browser script code calls `loadConf()`, JRE checks classes in the call chain of `new FileReader()`, but does not check any callers of `loadConf()` as shown in Figure 1(C). Therefore, untrusted browser script code can execute `new FileReader()`.

2.2.4. Improperly implemented privileged code. Privileged code should be carefully implemented because an attacker can bypass the stack inspection by executing this code. The example in Section 2.2.3 is vulnerable, and an attacker may steal an arbitrary file on a user's local disk by executing `loadConf()`. The problem here is that an attacker can specify an arbitrary file name as the argument `name`, and this is passed to `new FileReader()` without being properly validated.

The exploitation of improperly implemented privileged code is a widely known problem, and some guidelines recommend that developers pay particular attention to it [2][3][4]. Developers of signed applets for Microsoft or Netscape Java VM especially should use care because these Java VM implementations require the invocation of any insecure method in a signed applet to be implemented as privileged code, regardless of whether or not there is a call from untrusted code.

Fortunately, a developer can avoid use of privileged code through careful design. For example, privileged code is not needed for a signed applet which is only for Sun Java VM and is not called by browser script code.

2.3. Class replacement

The second attack strategy is for an attacker to force a signed applet to use classes of the attacker instead of classes of the signed applet. We call this *class replacement*. With this strategy, an attacker can trick a user into running a target signed applet whose classes have been partly replaced and force the target applet to harm the user by falsifying data on its dataflow. Unlike the first strategy, this one does not require privileged code, and it is the threat common to most signed applets.

This subsection describes the strategy. First, class replacement is described using an example. After that, two weak points of the *same-package-same-signer* protection mechanism are described [15], and it is shown that the strategy is still effective under the protection mechanism.

2.3.1. Cutting into a signed applet. According to the Applet specification, if the same name is shared by classes in different JAR files within an archive attribute included in an HTML document, JRE loads the one in the leftmost file of the attribute. By using this specification, an attacker's class can replace a class of a signed applet. Suppose that `foo.PurchaseApplet` (from Section 2.1) is implemented as follows.

```
package foo;

public class PurchaseApplet extends Applet {

  public void start() {
    // Create the dialog
    foo.util.MyFileDialog dialog =
     new foo.util.MyFileDialog("Save");
    // Show the dialog to the user and
    // prompt the user to select
    // the save file
    String path = dialog.showAndReturn();
    // Write some data to the selected file
    FileWriter fw = new FileWriter(path);
      :
  }
 :
```

```
}
```

This applet shows a file dialog to a user and writes some data to the save file selected by the user. The applet main class `PurchaseApplet` is a member of the package named `foo`, and the file dialog class, `MyFileDialog`, is in the other package named `foo.util`. The structure of `purchase.jar` is as follows.

```
purchase.jar
  +- foo/PurchaseApplet.class
  +- foo/util/MyFileDialog.class
```

As shown in the above code, the file name of the save file is determined by `MyFileDialog`. Therefore, an attacker can force the applet to write an arbitrary file by replacing `MyFileDialog` with that of the attacker as follows.

First, the attacker creates the malicious `MyFileDialog` class and puts it into `evil.jar`.

```
evil.jar
  +- foo/util/MyFileDialog.class
```

Next, the attacker creates the HTML document including the applet tag.

```
<applet code="foo.PurchaseApplet"
 archive="evil.jar,purchase.jar" ...
```

In this case, the attacker's `MyFileDialog` takes precedence over that of the target applet. Finally, by leading a user to the attacking page (as explained in Section 2.2.1), the attacker can overwrite an arbitrary file on the local disk of the user.

2.3.2. Weaknesses of same-package-same-signer. The above example is basically for an unsigned applet. An attacker cannot usually replace classes of a signed applet, because of the same-package-same-signer protection mechanism in Sun Java VM 1.2.2 and later. This mechanism ensures that any class in the same package is signed by the same signer. Therefore, an attacker cannot replace any signed class in the same package with an unsigned class of the attacker. However, the mechanism has two weak points.

First, it ensures there is one signer per package, not per signed JAR file. In other words, different packages in a signed JAR file can have different signers, or no signer at all. The case given in Section 2.3.1 is an example of this. It is permissible that `foo.PurchaseApplet` be signed but `foo.util. MyFileDialog` be unsigned. Note that Java's package system has no inheritance mechanism between packages, and the package `foo` and the package `foo.util` are independent of each other.

Second, the mechanism does not protect resources. Resources often contain important applet data, but attackers can easily replace them.

Through this weakness, class replacement is still possible despite the same-package-same-signer mechanism.

These two attack strategies are serious threats to a signed applet. In particular, class replacement is an effective way to attack a signed applet that has no privileged code. The next section describes our approach for preventing these attacks.

3. Our approach

This section describes our approach for preventing malicious recomposition attacks. Our approach is *encapsulation* that prevents recomposition of the components of an applet by tightly coupling them as a single unit. In addition, encapsulation should be achieved transparently so that it can be applied without any changes to Java VM or target applets.

Section 3.1 describes how encapsulation can be realized, and Section 3.2 deals with encapsulation transparency.

3.1. Encapsulation

Encapsulation of a target signed applet effectively prevents a malicious recomposition attack. This subsection describes two encapsulation mechanisms:

- class-loader-based isolation
- applet-document binding

3.1.1. Class-loader-based isolation. Class-loader-based

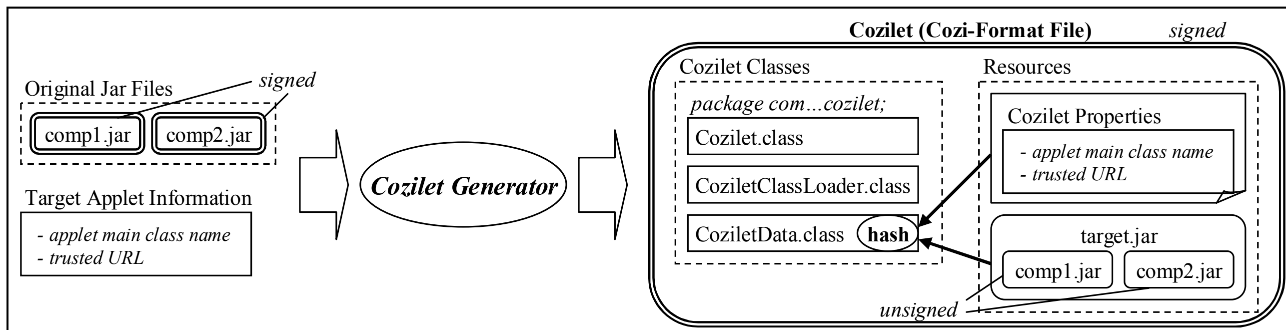

Figure 2. Generating cozilet

isolation is an effective encapsulation mechanism for classes and resources because a class loader is responsible for these. The basic idea is to isolate a target signed applet by assigning a class loader unique to the applet. This paper refers to such a class loader as a *cozi-loader*. We also propose a unique format that can contain the classes and resources of the target applet as a single unit. We refer to this format as a *cozi-format*.

A cozi-format file is shown in Figure 2. It is generated by the cozilet generator (described in Section 3.2.2) from JAR files and other information regarding the target signed applet. The cozi-format file is externally a standard JAR format file, but it contains the target classes and resources in a non-standard way. It includes JAR files formerly specified in an archive attribute (which we refer to as *remote jar files*) as its resource file `target.jar`. Note that applets may use classes or resources which are not contained in any JAR file (e.g., those in classpath or codebase), but we do not target these.

The cozi-format file can be signed by a standard signing tool `jarsigner` [16], and only the cozi-loader can recognize target classes and resources included in the cozi-format file as signed data. Attackers can extract and use these classes and resources as unsigned data, but cannot use them as signed because standard class loaders cannot recognize them as signed. Attackers can create their own class loaders similar to the cozi-loader, but these will be of no use because JRE prevents instantiation of any untrusted class loader through the stack inspection. The key here is that remote JAR files in the cozi-format file must not be signed. This prevents attackers from extracting and abusing them as signed data.

3.1.2. Applet-document binding. An HTML document also plays an important role in association with an applet. It determines how an applet is executed by specifying attributes such as codebase and archive or by giving initial parameters to the applet. In addition, it may contain script code which will interact with the applet using an interface such as Java-to-JavaScript communication (or vice versa). Creating a malicious HTML document is therefore a common technique used in malicious recomposition attacks. If we can bind an applet only to trusted HTML documents, it will provide an effective defense against such attacks.

To realize applet-document binding, we chose to use the URL as the origin of an HTML document when determining the trustworthiness of the document. `java.applet.Applet` has `getDocumentBase()` which returns the URL of an HTML document embedding a corresponding applet [15]. For applet-document binding, right after the applet execution starts we compare the URL returned by `getDocumentBase()` with the URL of the trusted Web site. A sample of the source code is shown below.

```
public final class PurchaseApplet
 extends Applet {

  // Hard-coded trusted server URL
  private static final String TRUSTED_URL
    = "https://example.com/";

  public void init() {
    // Obtain URL of the corresponding
    // HTML document
    URL docBaseURL = getDocumentBase();
    String docBase;
    if (docBaseURL != null) {
      docBase = docBaseURL.toString();
    } else {
      docBase = null;
    }
    // Check applet-document binding
    if ((docBase == null) ||
        !docBase.startsWith(TRUSTED_URL)) {
      throw new SecurityException(
         "applet-document binding error");
    }
     :
  }
}
```

In the above sample, the trusted URL is hard-coded. In our approach, the trusted URL is included in a resource file *cozilet properties* of the cozi-format file in Figure 2.

Microsoft recommends that ActiveX control programmers use a similar mechanism *SiteLock* [1]. Provided that the Web server, data communication channel, and the domain name of the Web site are trustworthy, this check can bind applets to HTML documents.

3.2. Transparency

This subsection describes how encapsulation is transparently applied to target signed applets. Our approach should be transparent both to current Java VM implementations and to target signed applets. There are two mechanisms to meet such requirements:

- applet switching
- cozilet generator

3.2.1. Applet switching. To transparently apply the protection mechanisms (described in Section 3.1) to current Java VM implementations, we propose realizing them in a unique signed applet. The unique applet provides class-loader-based isolation and applet-document binding to protect the encapsulated applet. In this paper, we refer to the unique applet as a *cozilet*.

However, a problem arises regarding Java VM, where the cozilet rather than the encapsulated applet seems to run as an applet. To cope with this, we provide an *applet switching* mechanism in the cozilet.

An applet is a type of GUI component that is simply added to a parent panel (i.e., `java.awt.Panel`). The cozilet can switch to the encapsulated applet by removing itself from the parent panel and adding the encapsulated applet to the panel. After doing this, GUI-related method calls are directed to the encapsulated applet.

For common non-GUI methods applied as applets (such as `start()` or `stop()`), JRE maintains an internal list of all running applets, and calls such methods of each applet by referring to the list. The cozilet is registered to the list, not the encapsulated applet, and the cozilet cannot update the list. Therefore, we have designed the cozilet so that it delegates common method calls from JRE to the encapsulated applet.

3.2.2. Cozilet generator. To realize the mechanism described in the previous subsection, a target signed applet must be converted to a cozilet. For transparency, however, such conversion should not require changes to the target applet. We therefore propose a tool for such conversion which we refer to as the cozilet generator.

The tool generates a single cozi-format file containing the following files (Figure 2): target remote JAR files, cozilet properties, and cozilet classes. Cozilet properties contain a trusted URL for applet-document binding and the applet main class name of the target applet. Cozilet classes are needed for cozilet execution, including `Cozilet` which is the applet main class of the cozilet and `CoziletClassLoader` which is the class of the cozi-loader. Because the cozilet is a standard signed applet, classes or resources in the cozi-format file may be replaced in a replacement attack as described in Section 2.3. To protect cozilet classes, all of these are in the same package, `com...cozilet`, so that the same-package-same-signer mechanism prevents replacement of them. Signing of the generated cozi-format JAR file is mandatory. Because the tool does not sign it, it should be signed by other tools, such as `jarsigner`.

To protect resources in the cozi-format file, the tool also generates a special class `CoziletData` and adds it to the cozi-format file. `CoziletData` contains hash values of `target.jar` and the cozilet properties as constant fields (Figure 2). The cozi-loader checks the integrity of `target.jar` and the cozilet properties in the cozi-format file based on hash values in `CoziletData`. The cozilet has other protection mechanisms to prevent malicious recomposition attacks on the cozilet itself, and these are described in Section 4.4.

4. Implementation

We have implemented the Cozilet mechanism for Sun Java VM (versions 1.3, 1.4, and 1.5) and this section describes the details of this implementation.

4.1. Architecture

Figure 3 shows the cozilet architecture.

A target signed applet is converted into a cozilet consisting of a cozi-format file by the cozilet generator. (Section 4.2 describes the cozilet generation.)

Class-loader-based isolation prevents an attacker's applets from accessing the encapsulated applet and applet-document binding prevents attackers from recomposing the cozilet and encapsulated applet with malicious HTML documents. In addition, applet switching enables transparent execution of the encapsulated applet. (Section 4.3 describes the cozilet execution.)

The cozilet should be able to protect itself from attackers because it is a standard signed applet. (Section 4.4 describes the cozilet protection.)

4.2. Cozilet generation

As described in Section 3.2.2, the cozilet generator converts a target signed applet into a cozilet. An example is

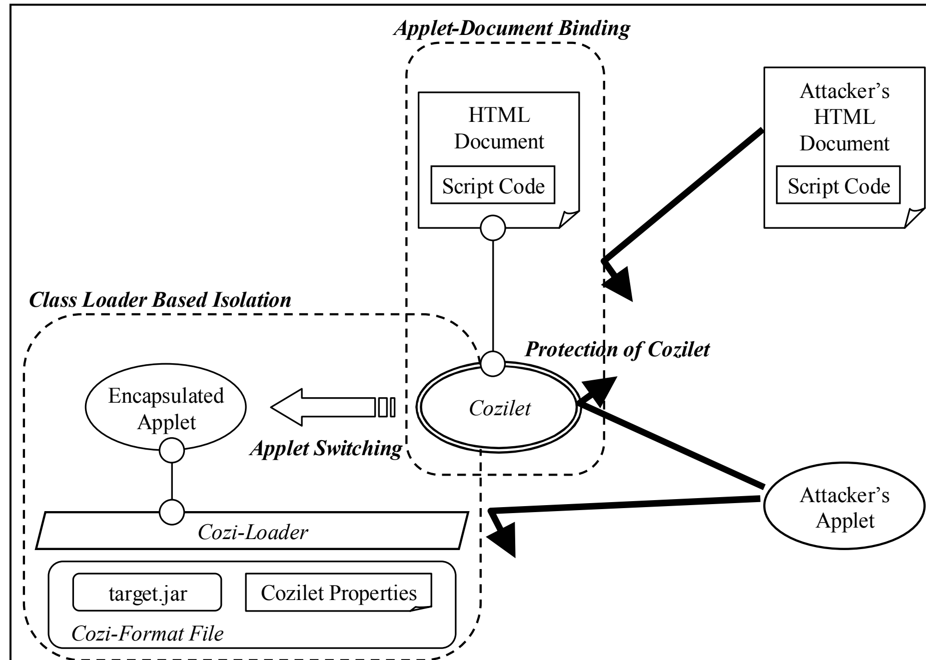

Figure 3. Cozilet architecture

shown below.

```
% java -jar cozilet-tool.jar –applet ¥
     foo.MyApplet cozilet.jar sample.jar
```

This operation generates the cozi-format file `cozilet.jar` to encapsulate a target applet whose remote JAR file is `sample.jar` and whose applet main class is `foo.MyApplet`. The developer of the target applet has to sign `cozilet.jar` through other tools, such as `jarsigner`.

The cozilet is always invoked by the following applet tag.

```
<applet code="com...cozilet.Cozilet"
 archive="cozilet.jar" ...
```

4.3. Cozilet execution

This subsection describes the behavior of cozilets, and shows that they can securely and transparently execute encapsulated applets.

4.3.1. Cozilet startup. Cozilets are standard signed applets which are executed by JRE in the following way.

(1) First, JRE loads and instantiates the cozilet main class `Cozilet`. `Cozilet` loads the cozilet properties that contain the information regarding the encapsulated applet at its static initializer. Its integrity can be verified by using the hash value included in `CoziletData`.

(2) Next, JRE executes `init()` of the cozilet. The method checks the applet-document binding as described in Section 3.1.2. The trusted URL is obtained from the cozilet properties. If the check succeeds, it goes to the next step.

(3) The method instantiates `CoziletClassLoader`, which works as a cozi-loader, and requests the cozi-loader to load the main class of the encapsulated applet whose name is included in the cozilet properties. The cozi-loader verifies the integrity of the class by using the hash value included in `CoziletData`. If it is verified, the cozi-loader loads the class and the cozilet instantiates it. After this, all the classes and resources of the encapsulated applet are loaded only by the cozi-loader.

(4) Finally, the cozilet switches itself for the encapsulated applet and executes its `init()`.

4.3.2. Switching to an encapsulated applet. As described in Section 3.2.1, applet switching can be achieved by replacing a reference of an applet's parent panel. In addition, an applet needs an appropriate applet context and an applet stub. A stub can be set to an applet by calling its `setStub()` [15], and a context can be obtained from the stub. Therefore, it is sufficient that only an applet stub be passed to an applet.

A cozilet must pass its own applet stub to the encapsulated applet. It is not easy to get the stub, though, because there is no method like `getStub()` and because `setStub()` cannot be overridden since it is declared as final.

Fortunately, in Sun Java VM implementations, a parent panel of an applet also serves as an applet stub, and the applet can get its stub by simply converting the type of (or casting) its parent panel. In this way, cozilets can hand over their applet stubs to encapsulated applets. After this, cozilets finish the applet switching by calling `init()` of the encapsulated applets.

Also, as described in Section 3.2.1, cozilets delegate control calls issued by JRE to encapsulated applets, by overriding corresponding methods, such as `start()` or `stop()`. This delegation enables encapsulated applets to be normally controlled by JRE after switching.

4.4. Cozilet protection

A cozilet may be attacked in various ways because it is a standard signed applet. It protects itself through several measures. The most effective measure is the applet-document binding explained in Section 3.1.2, which can block most attacks provided that the Web server, data communication channel, and the domain name of the Web site are trustworthy. And as described in Section 3.2.2, a cozilet prevents class and resource replacement by declaring its classes as a single package and verifying the hash values of its resource files. A cozilet has additional protection mechanisms, and this subsection describes two of these which are particularly important.

First, the public methods of cozilet are protected through stack inspection. The cozilet inherits many public methods from `Applet`. An attacker may be able to harm a user by calling these methods. To prevent this, the cozilet overrides such public methods and calls `AccessController.checkPermission()` at the beginning of the methods. `checkPermission()` causes the stack inspection to check if all the callers in the current call chain have the permission specified by a caller of `checkPermission()` [15]. The cozilet specifies `AllPermission` to the method so that only system classes can execute its public methods.

Second, serialization of the cozilet is disabled. If serialization is enabled, an attacker can take advantage of the serialization and deserialization interface to read a sensitive value of a private field and write an illegal value to the field [2][3][4]. Unfortunately, the cozilet is serializable because its superclass `Applet` is serializable. The cozilet forcibly disables its serialization by throwing

an exception unconditionally for every method related to serialization in the following way. This measure is introduced in Securing Java [2].

```
public final class Cozilet extends Applet {

   :

  private void
     writeObject(ObjectOutputStream o)
                       throws IOException {
    throw new NotSerializableException();
  }

  private void
     readObject(ObjectInputStream o)
          throws IOException,
                 ClassNotFoundException {
    throw new NotSerializableException();
  }

  private Object writeReplace()
          throws ObjectStreamException {
    throw new NotSerializableException();
  }

  private Object readResolve()
          throws ObjectStreamException {
    throw new NotSerializableException();
  }

}
```

4.5. Optional features

Up to now, this paper has described the basic architecture of the Cozilet mechanism. This section describes some additional features. Although they are optional, they can make cozilets and encapsulated applets more secure or useful.

4.5.1. Interaction with browser script code. Current Java VM implementations enable applets to interact with browser script code. For transparency, the cozilet has to allow the encapsulated applet to do the same.

Fortunately, access from encapsulated applets to browser script code is possible without any particular mechanism. In the Sun Java VM implementations, an applet needs to get a mediate object to access the browser script code by calling `getWindow()` of `JSObject` or `getService()` of `DOMService` with a reference to the applet itself passed as the parameters of these methods [15]. These methods obtain the applet context and return the mediate object corresponding to the applet context. Because the encapsulated applet has the same applet context as the cozilet, the encapsulated applet can obtain the same mediate object as the cozilet.

However, access from the browser script code to encapsulated applets is not possible without a trick, because JRE refers to its internal list (described in Section 3.2.1) when the browser script code requires access to applets. Cozilet's trick to enable such access is to delegate limited kinds of calls from the browser script code to encapsulated applets. If the developer of a target applet specifies that the browser script code should call the method signatures at conversion by the cozilet generator, the tool adds delegation methods having the same signatures to the cozilet. Delegation also has the advantage of limiting methods which can be called by the script code of attackers to the minimum required. This is because attackers can call all the public methods of the applet if the applet is not encapsulated.

Through method delegation, however, cozilets cannot delegate their fields to encapsulated applets. We think that field delegation is unnecessary because most applets do not need it.

4.5.2. Exclusive mode. Cozilets can run in *exclusive mode* to prevent any untrusted applets from starting during the execution of an encapsulated applet. Although this may not prevent the execution of untrusted applets running before cozilets start, it can usually make cozilets secure.

The exclusive mode takes advantage of the class loading and defining restriction mechanisms in the Sun Java VM implementations [4]. The class loading restriction mechanism prevents untrusted classes from loading any class contained by the particular packages specified in the security property `"package.access"` (whose default value is `"sun."`). This mechanism makes use of the stack inspection at class loading. The class defining restriction mechanism prevents untrusted classes from defining any class contained by the particular packages specified in the security property `"package.definition"` (whose default value is `""`). This mechanism also makes use of the stack inspection at class definition.

Cozilets set the value `"java.,javax.,sun.,com.,org.,netscape.,sunw."` to both of the above security properties right after they start. This is very simple, but effectively prevents attacking applets from starting because they cannot load applet fundamental classes, even `java.applet.Applet`. Of course, attacking applets cannot set the security properties because the stack inspection mechanism restricts that operation.

The exclusive mode may not prevent the execution of attacking applets running before cozilets start, because the classes required by the attacking applets may have already been loaded. Also, the exclusive mode may prevent browser script code from accessing encapsulated applets, because JRE usually regards script code as untrusted code. With these exceptions, however, we recommend use of the exclusive mode.

4.5.3. Encapsulation of local applet components. An

applet usually consists of remote JAR files which are deployed on a Web site and specified in an archive attribute, but some applets require users to install local components on users' local disk in advance to reduce network traffic or for use as common system components. The cozilet supports encapsulation of such local components to some degree.

Local components of applets are local JAR files and native libraries. They are usually unsigned, but installed in directories from which the common *system class loader* can load (which we will refer to as a system path) and regarded as system classes having all permissions or system native libraries.

The cozi-format file can include their installed paths in the cozilet properties. The cozi-loader tries to find them by using the paths in the cozilet properties. Moreover, the cozi-format file can also include their hash values as constant fields of `CozilетData` (Section 3.2.2). The cozi-loader checks the integrity of local applet components by using these hash values. This encapsulation is done by the cozilet generator.

Note that local JAR files should be unsigned. Signed JAR files may be obtained and abused by attackers. As mentioned, the integrity of local JAR files (also native libraries) can be verified in the unique way of the Cozilet mechanism. Also note that local JAR files and native libraries should be moved to any path other than the system paths. If they are installed on the system paths, they may be abused by attackers. Moving them may affect the transparency of encapsulation (e.g., if they are re-installed), but it is strongly recommended.

5. Discussion

In this section, we discuss the issue in a more general manner, rather than one based on the detailed mechanisms of the Java applet environment described in the previous sections. Let us rephrase the issue as follows: to prevent malicious recomposition attacks, all components of an applet (or any mobile code program) must be *securely deployed* (e.g., identified and composed) on the client-side, and then the components must be *isolated* from other components.

For secure deployment of applets in current Java VM implementations, an APPLET tag in an HTML document plays a critical role, determining which JAR file is loaded and executed. The current VMs, however, do not consider the tag's trustworthiness. Our approach rectifies this weakness through two mechanisms: applet-document binding ensures the origin of the HTML document containing the APPLET tag, and digitally signed cozilet properties securely identify additional JAR files to be deployed.

Secure deployment is important not only for an applet, but also for other componentized mobile code technologies. Sun's Java Web Start [18] is a runtime environment for network-launchable Java applications. Although it is not a perfect substitute for applets because of its very limited interaction with Web browsers, it is more secure than applets by the same reason. In addition, malicious deployment can be prevented by digitally signing a JNLP file, which corresponds to an APPLET tag for an applet. Furthermore, a Web Start application is regarded as "signed" if and only if all the JAR files specified in its JNLP file are signed by a single signer. These features make malicious recomposition attacks hard. Microsoft recommends ActiveX control programmers use *SiteLock* [1], a mechanism similar to applet-document binding. Also, the Microsoft .NET Framework provides a mechanism called *LinkDemand* [17], which prevents trusted assemblies from being accessed by malicious assemblies. It is preferable that Java VM itself support such a secure deployment scheme for applets in the future. Until then, however, our approach provides an alternative way to protect signed applets without modifying VM.

Isolation is also important. Our approach uses *class-loader-based isolation*, which can be easily implemented on existing Java VMs. It isolates the namespace of classes, with the exception of system classes shared among the JRE. Therefore, in some cases, an attacking applet can obtain a reference to an instance of a class unique to a victim applet as a type of its superclass which is a system class, such as `java.applet.Applet` or `java.lang.Object`, and it may cause a security breach. While the cozilet takes some simple measures to prevent leakage of the object references of classes unique to the encapsulated applet, it is difficult to completely prevent such leakage.

Isolation is a general challenge not only to prevent the abuse of trusted applets, but also to ensure the reliability of general Java programs. For example, a Java-based application server needs to isolate each application running on it to prevent applications affecting each other or the server. Therefore, much work has aimed at realizing not only class-loader-based isolation but also the isolation of heap memory, native code memory, and other JRE resources [7][8][9]. In particular, JSR-121 Application Isolation API [7] is now being standardized through the Java Community Process. (Unfortunately, Sun seems to have put off adopting it in J2SE 1.5.) To prevent the abuse of trusted applets more reliably, these strong isolation mechanisms are preferable.

6. Related work

Our approach is transparent to both target signed applet and current Java VM implementations. We know of no other groups taking a similar approach at the moment.

Isolation, as we explained in Section 5, is an effective way to protect trusted applets. However, current

approaches [7][8][9] are not transparent to users because they require users to replace installed Java VM implementations. Other approaches for non-Java applications have been reported. Alcatraz [6] is a logical application isolation approach that forces untrusted applications' insecure execution in Linux to be committed by users. SoftwarePot [5] is an application encapsulation approach that prevents malicious behavior by encapsulated untrusted applications based on security policies specified by users in Solaris and Linux. However, these approaches do not aim at preventing the abuse of trusted programs.

There are some guidelines available which describe problems and rule-of-thumb countermeasures. Securing Java [2] suggests programmers to care for twelve anti-patterns in developing secure applets. Sun has published the Security Code Guidelines [4] and Inside Java2 Platform Security [3] which describe various security guidelines for Java. Auditing tools based on these guidelines have been released or announced [10][11][12]. However, it is difficult for most developers to understand and prevent all attacks because these attacks are skillfully designed by attackers.

7. Summary

We have developed the *Cozilet* mechanism to prevent *malicious recomposition attacks*. Such attacks are a serious threat to trusted applets, such as signed applets. In our mechanism, before deployment, the *cozilet generator* encapsulates a target signed applet into a special signed applet, a *cozilet*, in the indecomposable *cozi-format*. The cozilet has two protection mechanisms: *class-loader-based isolation* and *applet-document binding*. Upon its execution, the cozilet isolates the classes and resources of the encapsulated applet by assigning a special class loader, the *cozi-loader*, unique to the applet. It also checks if the HTML document was downloaded from a trusted Web site, to ensure that no attackers have deployed it on their sites.

The cozilet can be executed instead of the target applet. When it is invoked by JRE, it applies the above protection mechanisms to the encapsulated applet, and switches itself with the applet. After switching, the encapsulated applet can run normally. The mechanism can easily protect both applets developed in the future and applets currently in use, because it is transparent not only to the target applets, but also to current Java VM implementations. We will apply this mechanism to significant applets now being used in critical systems, such as e-commerce or e-government systems.

8. References

[1] "SiteLock Template 1.04 for ActiveX Controls", http://msdn.microsoft.com/archive/en-us/samples/internet/components/sitelock/default.asp, Microsoft Developer Network.

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

[3] L. Gong, G. Ellison, and M. Dageforde, "Inside Java 2 Platform Security: Architecture, API Design, and Implementation, 2/E", Addison-Wesley, 2003.

[4] "Security Code Guidelines", http://java.sun.com/security/seccodeguide.html, Sun Microsystems, Inc., 2000.

[5] K. Kato and Y. Oyama, "SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation", Software Security - Theories and Systems, Volume 2609 of Lecture Notes in Computer Science, Springer-Verlag, February 2003.

[6] Z. Liang, V.N. Venkatakrishnan and R. Sekar, "Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs", ACSAC'03, Las Vegas, December 2003.

[7] "JSR 121: Application Isolation API Specification", http://jcp.org/en/jsr/detail?id=121, Java Community Process.

[8] "The Barcelona Project", http://research.sun.com/projects/barcelona/, Sun Microsystems, Inc.

[9] "The Janos Project", http://www.cs.utah.edu/flux/janos/, The Flux Research Group.

[10] J. Viega, et al, "Statically Scanning Java Code: Finding Security Vulnerabilities," IEEE Software 17(5), 2000.

[11] M. Curphey, "codespy", http://www.securityfocus.com/archive/107/349071/2004-01-02/2004-01-08/0, January 7th 2004.

[12] "SIMPLIA/JF Kiyacker", http://www.securityfocus.com/archive/98/341866/2003-10-17/2003-10-23/0, Fujitsu Limited, October 20th 2003.

[13] "Java Archive (JAR) Features", http://java.sun.com/j2se/1.4.2/docs/guide/jar/, Sun Microsystems, Inc.

[14] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers., "Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2", In Proceedings of the USENIX Symposium on Internet Technologies and Systems, 1997.

[15] "J2SE 1.4.2 API Documentation", http://java.sun.com/j2se/1.4.2/docs/api/, Sun Microsystems, Inc.

[16] "jarsigner - JAR Signing and Verification Tool", http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/jarsigner.html, Sun Microsystems, Inc.

[17] ".NET Framework Developer's Guide - Link Demands", http://msdn.microsoft.com/library/en-us/cpguide/html/cpconlinkdemands.asp, Microsoft Developer Network.

[18] "Java Network Launching Protocol & API Specification (JSR-56) Version 1.0.1", http://java.sun.com/products/javawebstart/, Sun Microsystems, Inc.