DIMACS Technical Report 97-64
September 1997

The Design and Implementation of a Java Playground

by

Dahlia Malkhi Michael Reiter Avi Rubin

AT&T Labs—Research, Florham Park, New Jersey, USA
{dalia,reiter,rubin}@research.att.com

ABSTRACT

Mobile code presents a number of threats to the machines that execute it. In this paper we introduce an approach for protecting machines and the resources they hold from mobile code, and describe a system based on our approach for protecting host machines from Java applets. In our approach, each Java applet downloaded to the protected domain is rerouted to a dedicated machine (or set of machines), the *playground*, at which it is executed. Prior to execution the applet is transformed to use the downloading user's web browser as a graphics terminal for its input and output, and so the user has the illusion that the applet is running on her own machine. In reality, however, mobile code runs only in the sanitized environment of the playground, where user files cannot be mounted and from which only limited network connections are accepted by machines in the protected domain. We describe the design, implementation, performance and limitations of our system, and directions for future work.

1 Introduction

Advances in mobile code technologies, particularly Java, have considerably increased the exposure of networked computers to attackers. Due to the "push" nature of these technologies, an attacker can download and execute programs on a victim's machine often without the victim's knowledge or consent. The attacker's code could conceivably delete, modify, or steal data on the victim's machine, or otherwise abuse other resources available from that machine. Moreover, mobile code "sandboxes" intended to constrain mobile code have in many cases proven unsatisfactory, in that errors in implementations enable mobile code to circument the sandbox's security mechanisms [DFW96, MF97].

One of the oldest ideas in security, computer or otherwise, is to physically separate the attacker from the resources of value. In this paper we present a novel approach for physically separating mobile code from those resources. The basic idea is to execute the mobile code somewhere other than the user's machine, where the resources of value to the user are not available, and to force the mobile code to interact with the user only from this sanitized environment. Of course, this could be achieved by running the mobile code at the server that serves it (thereby eliminating its mobility). However, the challenge is to achieve this physical separation without eliminating the benefits derived from code mobility, in particular reducing load on the code's server and increasing performance by co-locating the code and the user.

In order to achieve this protection at an organizational level, we propose the designation of a distinguished machine (or set of machines), a *playground*, on which all mobile code served to a protected domain is executed. That is, any mobile code pushed to a machine in this protected domain is automatically rerouted to and executed on the domain's playground. To enable the user to interact with the mobile code during its execution, the user's computer acts as a graphics terminal to which the mobile code displays its output and from which it receives its input. However, at no point is any mobile code executed on the user's machine. Provided that valuable resources are not available from the playground, the mobile code can entirely corrupt the playground with no risk to the domain's resources. Moreover, because the playground can be placed in close network proximity to the machines in the domain it serves, performance degradation experienced by users is minimal. There can even be many playgrounds serving a domain to balance load among them.

In this paper we report on the design and implementation of a playground for Java applets. As described above, our system reroutes all Java applets retrieved via the web to the domain's playground, where the applets are executed using the user's browser essentially as an I/O terminal. By disallowing the playground to mount protected filesystems or open arbitrary network connections to domain machines (in the limit, locate the playground just "outside" the domain's firewall), the domain's resources can be protected with high assurance even if the playground is completely corrupted. Our system is transparent to users and applet developers, and in some configurations requires no changes to web browsers in use today. While there is a class of applets that are not amenable to execution on our present playground prototype—e.g., due to performance requirements or code structure—in our experience this class is a small fraction of Java applets. We thus believe our prototype demonstrates that our approach to mobile code security is viable for Java applets.

As described above, the playground need not be trusted for our system to work securely. Indeed, the only trusted code that is common to all configurations of our system is the browser itself and a small "graphics server", itself a Java applet, that runs in the browser. The graphics server implements interfaces that the untrusted applet, running on the playground, calls to interact with the user. The graphics server is a simply structured piece of code, and thus should be amenable to analysis. In one configuration of our system, trust is limited to *only* the graphics server and the browser, but doing so requires a minor change to browsers available today. If we are constrained to using today's browsers off-the-shelf, then a web proxy component of our system, described in Section 3, must also be trusted.

The remainder of this paper is structured as follows. We begin in Section 2 by relating our work to previous efforts at protecting resources from hostile mobile code. We give an overview of our system in Section 3 and refine this description in Section 4, where we describe the implementation of the system in some detail. The security of the system is discussed in Section 5, and limitations are discussed in Section 6. We conclude in Section 7.

2 Related work

There are three general approaches that have been previously proposed for securing hosts from mobile code. The first to be deployed on a large scale for Java is the "sandbox" model. In this model, Java applets are executed in a restricted execution environment (the sandbox) within the user's browser; this sandbox attempts to prevent the applet from performing illegal actions. This approach has met with mixed success, in that even small implementation errors can enable applets to entirely bypass the security restrictions enforced by the sandbox [DFW96].

The second general approach is to execute only mobile code that is *trusted* based on some criteria. For example, Balfanz has proposed a *Java filter* that allows users to specify the servers from which to accept Java applets (see `http://www.cs.princeton.edu/sip/`). Here the criterion by which an applet is trusted is the server that serves it. A related approach is to determine whether to trust mobile code based on its author, which can be determined, e.g., if the code is digitally signed by the author. This is the approach adopted for securing Microsoft's ActiveX content. In addition, a combination of this approach and the sandbox model is implemented in Netscape Communicator, which enforces access controls on an applet based on the signatures it possesses [WBDF97]. A third variation on this theme is *proof-carrying code* [NL96], where the mobile code is accompanied by a proof that it satisfies certain properties. However, these techniques have not yet been applied to languages as rich as Java (or Java bytecodes).

Our approach is compatible with both of the approaches described above. That is, our playground can (and does) execute applets in sandboxes (hence the name "playground"), and could easily be adapted to execute only "trusted" applets based on any of the criteria above. Our approach provides an orthogonal defense against hostile applets, and in particular, in our system a hostile applet is still physically separated from valuable resources even after circumventing these other types of defenses.

The third approach to securing hosts from mobile code is simply to not run mobile code. A course-grained approach for Java is to simply disable Java in the browser. Another approach is to filter out all applets at a firewall [MRR97] (see also [MF97, Chapter 5]), which has the advantage of allowing applets served from behind the firewall to be executed.

It has recently come to our attention that Digitivity, Inc., a California-based company, has independently proposed and is commercially developing a system similar to ours for defending against hostile Java applets. We are unaware of descriptions of their system in the scientific literature, but information obtained from a white paper [Her97], their web site (`http://www.digitivity.com`), and discussions with company representatives[1] have allowed us to identify several differences between our systems. First, elements of the Digitivity system—notably the protocol for communication between the user's browser and (their analog of) the playground, and the Java Virtual Machine (JVM) running on their playground—are proprietary, whereas our system is built using only public, widely-used protocols and JVMs. This may enable Digitivity to more easily tune its system's performance, but our approach promotes greater confidence in the security of our system by exposing it to maximum public scrutiny and understanding. Second, our system does not require trust in certain elements of the system that, according to [Her97], are trusted in their architecture. We discuss the trusted elements of our architecture in Section 5.1.1.

3 Architecture

As discussed in the previous two sections, the core idea in this paper is to establish a dedicated machine called a *playground* at which mobile code is transparently executed, using users' browsers as I/O terminals. In this section we give an overview of the playground and supporting architecture that we implemented for Java, deferring many details to Section 4.

[1]These discussions took place on August 26, 1997 at the Java Internet Business Expo held in the Jacob Javitz Center, New York City, August 25-28, 1997.

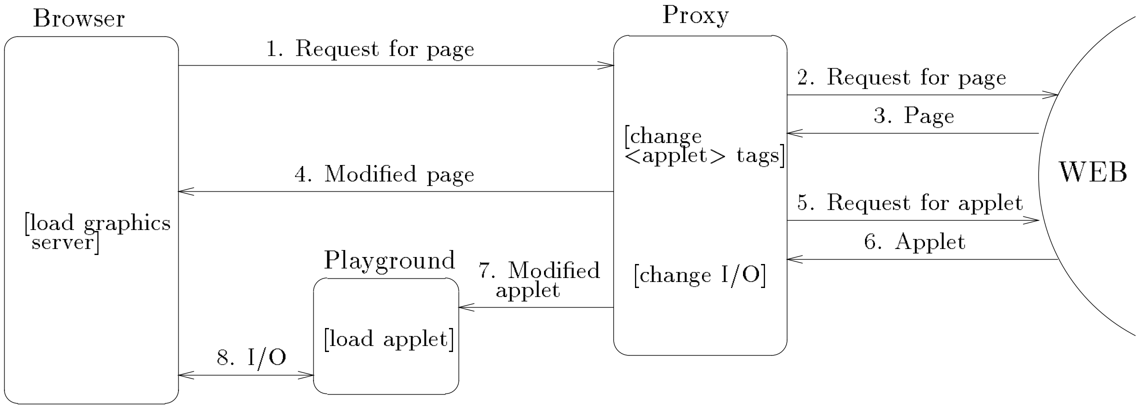

Figure 1: Playground architecture

To understand how our system works, it is first necessary to understand a bit about how browsers retrieve, load, and run Java applets. When a browser retrieves a web page written in Hypertext Markup Language (HTML), it takes actions based on the HTML *tags* in that page. One such tag is the `<applet>` tag, which might appear as follows:

```
<applet code=hostile.class ...>
```

This tag instructs the browser to retrieve and run the applet named `hostile.class` from the server that served this page to the browser. The applet that returns is in a format called *Java bytecode*, suitable for running in any JVM. This bytecode is subjected to a bytecode verification process, loaded into the browser's JVM, and executed. Further details can be found, e.g., in [LY97].

In our system, when a browser requests a web page, the request is sent to a *proxy* (Figure 1, step 1). The proxy forwards the request to the end server (step 2) and receives the requested page (step 3). As the page is received, the proxy parses it to identify all `<applet>` tags on the returning page, and for each `<applet>` tag so identified, the proxy replaces the named applet with the name of a trusted *graphics server* applet stored locally to the browser.[2] The proxy then sends this modified page back to the browser (step 4), which loads the graphics server applet upon receiving the page. For each `<applet>` tag the proxy identified, the proxy retrieves the named applet (steps 5–6) and modifies its bytecode to use the graphics server in the requesting browser for all input and output. The proxy forwards the modified applet to the playground (step 7), where it is executed using the graphics server in the browser as an I/O terminal (step 8).

To summarize, there are three important components in our architecture: the graphics server applet that is loaded into the user's browser, the proxy, and the playground. None of these need be executed on the same machine, and indeed there are benefits from executing them on different machines (this is discussed further in Section 5). The same proxy can be used for multiple browsers and multiple playgrounds. In the case of multiple playgrounds, the proxy can distribute load among the playgrounds for improved performance. In the following subsections, we describe each of these components in additional detail, focusing on its functional role in the system. Security issues are discussed in Section 5.

3.1 The graphics server

In this section we give an overview of the graphics server that is loaded into a user's browser in place of an applet provided by a web server. Because the graphics server is a Java applet, we must introduce some Java

[2]Here, "stored locally" means stored in a directory named by the `CLASSPATH` environment variable.

terminology to describe it. In Java, a *class* is a collection of data fields and functions (called *methods*) that operate on those fields. An *object* is an instance of a class; at any point in time it has a state—i.e., values assigned to its data fields—that can be manipulated by invoking the methods of that object (defined by the object's class). Classes are arranged in a hierarchy, so that a *subclass* can inherit fields and methods from its *superclass*. A running Java applet consists of a collection of objects whose methods are invoked by a runtime system, and that in turn invoke one another's methods. For more information on Java see, e.g., [Fla97].

3.1.1 Remote AWT classes

The Abstract Window Toolkit (AWT) is the standard API for implementing graphical user interfaces (GUI) in Java programs. The AWT contains classes for user input and output devices, including buttons, choice boxes, text fields, images, and a variety of types of windows, to name a few. Virtually every Java applet interacts with the user by instantiating AWT classes and invoking the methods of the objects so created.

The intuitive goal of the graphics server is to provide versions of the AWT classes whose instances can be created and manipulated from the playground. For example, the graphics server should enable a program running on the playground to create a dialog window in the user's browser, display it to the user, and be informed when the user clicks the "ok" button. In the parlance of distributed object technology, such an object—i.e., one that can be invoked from outside the virtual machine in which it resides—is called a *remote object*, and the class that defines it is called a *remote class*. So, the graphics server, running in the user's browser, should allow other machines (the playground) to create and use "remote AWT objects" in the user's browser for interacting with the user.

Accordingly, the graphics server is implemented as a collection of remote classes, where each remote class (with one exception that is described in Section 3.1.2) is a remote version of a corresponding AWT class. The (modified) Java applet running on the playground creates a collection of graphical objects in the graphics server to implement its GUI remotely, and uses *stubs* to interact with them (see Section 3.3). To minimize the amount of code in the graphics server, each remote class is a subclass of its corresponding AWT class, which enables it to inherit many method implementations from the original AWT class. Other methods must be overwritten, for example those involving event monitoring: in the remote class, methods involving the remote object's events (e.g., mouse clicks on a remote button object) must be adapted to pass back the event to the stubs residing in the playground JVM. In our present implementation, the remote classes employ Remote Method Invocation (RMI) to communicate with the playground. RMI is available on all Java 1.1 compliant platforms, including Netscape Communicator, Internet Explorer 4.0, JDK 1.1, and HotJava 1.0.

3.1.2 The remote Applet class

As described in Section 3.1.1, most classes that comprise the graphics server are remote versions of AWT classes. The main exception to this is a remote version of the `java.applet.Applet` (or just `Applet`) class, which is the class that all Java applets must subclass. The main purpose of the `Applet` class is to provide a standard interface between applets and their environment. Thus, the remote version of this class serves to provide this interface between the applet running on the playground and the environment with which it must interact, namely the user's browser.

More specifically, this class implements two types of methods. First, it provides remote interfaces to the methods of the `Applet` class, so that applets on the playground can invoke them to interact with the user's environment. Second, this class defines a new "constructor" method for each remote AWT class (see Section 3.1.1). For example, there is a `constructButton` method for constructing a remote button in the user's browser. This constructor returns a reference to the newly-created button, so that the remote methods of the button can be invoked directly from the playground. Similarly, there is a constructor method for each of the other remote AWT classes.

When initially started, the graphics server consists of only one object, whose class is the remote Applet class, called `BrowserServer.class`. The applet on the playground can then invoke the methods of this object (and objects so created) to create the graphical user interface that it desires for the user.

3.2 The proxy

The proxy serves as the browser's and playground's interface to the web. It retrieves HTML pages for the browser and Java bytecodes for the playground. In addition to retrieving them, however, it transforms them to formats suitable for the browser or playground to use.

When retrieving an HTML page for the browser, the proxy is responsible for parsing the returning HTML, identifying all `<applet>` tags in the page, and replacing them with references to the remote Applet class of the graphics server (see Section 3.1.2). Thus, when the browser receives the returned HTML page it loads this remote Applet class (stored locally), instead of the applet originally referenced in the page.

When retrieving a Java bytecode file for the playground, the proxy is responsible for transforming it into bytecode that interacts with the user on the browser machine while running at the playground. It does so by replacing all invocations of AWT methods with invocations of the corresponding remote AWT methods at the browser, or more precisely with invocations of playground-side stubs for those remote AWT methods (which in turn call the remote AWT methods). This task involves parsing the incoming bytecode and making automatic textual substitutions to change the names of AWT classes to the names of the representative stubs of the corresponding remote AWT classes.

3.3 The playground

The playground is a machine that loads modified applets from the proxy and executes them. As described above, the proxy modifies the applet's bytecodes so that playground-resident stubs for remote AWT methods are called instead of the (non-remote) AWT methods themselves. So, when a modified applet runs on the playground, a skeleton of its GUI containing stubs for corresponding remote graphics objects is built on the playground. The stubs contain code for remotely invoking the remote objects' methods at the user's browser and for handling events passed back from the browser. For example, in the case of a dialog window with an "ok" button, stubs for the window and for button objects are instantiated at the playground. Calls to methods having to do with displaying the window and button are passed to the remote objects at the user's browser, and methods for handling the "button press" events are provided by the button's stub to handle such events passed back from the browser. These stubs are stored locally on the playground, but aside from this, the playground is configured as a standard JVM.

A playground machine is a centralized resource that can be carefully administered, and moreover investments in the playground (e.g., upgrading to the latest high-performance hardware and software, or performing enhanced monitoring of the JVM) can improve observed Java performance and security for all users in the protected domain. There can even be multiple playgrounds in operation for performance purposes. In this case, the proxy would use a load-balancing algorithm to choose the playground to which to assign each applet.

4 Implementation

4.1 An example

In order to understand how the components described in Section 3 work together, in this section we illustrate the execution of a simple applet. This example describes how the applet is automatically transformed to interact with the browser remotely, and how the graphics server and its playground-side stubs interact during applet execution.

The applet we use for illustration is shown in Figure 2. This is a very simple (but complete) applet that prints the word "Click!" wherever the user clicks a mouse button. For the purposes of this discussion, it implements two methods that we care about. The first is an `init()` method that is invoked once and registers `this` (i.e., the applet object) as one that should receive mouse click events. This registration is achieved via a call to its own `addMouseListener()` method, which it inherits from its `Applet` superclass. The second method it implements is a `mouseClicked()` method that is invoked whenever a mouse click occurs. This method calls `getGraphics()`, again inherited from `Applet`, to obtain a `Graphics` object whose methods can be called to display graphics. In this case, the `mouseClicked()` method invokes the `drawString()` method of the `Graphics` object to draw the string "Click!" where the mouse was clicked.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Click extends Applet
  implements MouseListener {

  public void init() {
    // Tell this applet what MouseListener objects to notify when
    // mouse events occur.  Since we implement the MouseListener
    // interface ourselves, our own methods are called.
    this.addMouseListener(this);
  }

  // A method from the MouseListener interface.  Invoked when the
  // user clicks a mouse button.
  public void mouseClicked(MouseEvent e) {
    Graphics g = this.getGraphics();
    g.drawString("Click!", e.getX(), e.getY());
  }

  // The other, unused methods of the MouseListener interface.
  public void mousePressed(MouseEvent e) {;}
  public void mouseReleased(MouseEvent e) {;}
  public void mouseEntered(MouseEvent e) {;}
  public void mouseExited(MouseEvent e) {;}
}
```

Figure 2: An applet that draws "Click!" wherever the user clicks

The `Click` applet is a standalone applet that is not intended to be executed using a remote graphics display for its input and output. Thus, when our proxy retrieves (the bytecode for) such an applet, the applet must be altered before being run on the playground. For one thing, the `addMouseListener()` method invocation must somehow be passed to the browser to indicate that this playground applet wants to receive mouse events, and the mouse click events must be passed back to the playground so that `mouseClicked()` is invoked.

In our present implementation, passing this information is achieved using Java Remote Method Invocation (RMI). Associated with each remote class is a stub for calling it that executes in the calling JVM. The stub is invoked exactly as any other object is, and once invoked, it marshals its parameters and passes them across the network to the remote object that services the request. In the discussion that follows, the stubs are described by *interfaces* with the suffix `Xface`. For example, `BrowserXface` is the interface that is used to call the `BrowserServer` object of the graphics server.

RMI provides the mechanism to invoke methods remotely, but how do we get the `Click` applet to use RMI? To achieve this, we exploit the subclass inheritance features of Java to interpose our own versions of the methods it invokes. More precisely, we alter the `Click` applet to subclass our own `PlaygroundApplet`, rather than the standard `Applet` class. This is a straightforward modification of the bytecode for the `Click` applet. By changing what `Click` subclasses in this way, the `addMouseListener()` method called in Figure 2 is now the one in Figure 3. `PlaygroundApplet`, shown in Figure 3,[3] passes calls, when necessary, to the

[3]For readability, in Figures 3–6 we omit `import` statements, error checking, exception handling (`try/catch`

remote applet object of the graphics server described in Section 3.1.2. The `addMouseListener()` method simply adds its argument (the `Click` applet) to an array of mouse listeners and, if this is the first to register, registers itself as a mouse listener at the graphics server.

This registration at the graphics server is handled by the `addPlaygroundMouseListener()` remote method of `BrowserServer`, the class of the remote applet object running on the browser (see Section 3.1.2). The relevant code of `BrowserServer` is shown in Figure 5. Recall that `BrowserServer` implements the `BrowserXface` *interface* that specifies the remote methods that can be called from the playground. The `addPlaygroundMouseListener()` method, which is one of those remote methods, simply records the fact that the playground applet wants to be informed of mouse events and then registers its own object as a mouse listener, so that its object's `mouseClicked()` method is invoked when the mouse is clicked. Such an invocation passes the mouse-click event—or more precisely, a reference to a `BrowserEvent` remote object that holds a reference to the actual mouse-click event object—back to the `playgroundMouseClicked()` remote method of the `PlaygroundApplet` class. The `playgroundMouseClicked()` method invokes `mouseClicked()` with a `PlaygroundMouseEvent` object, which holds the reference to the `BrowserEvent` object. That is, the `PlaygroundMouseEvent` object translates invocations of its own methods (e.g., `getX()` and `getY()` in Figure 2) into invocations of the corresponding `BrowserEvent` remote methods, which in turn translates them into invocations of the actual `Event` object in the browser. For the sake of brevity, the `BrowserEvent` and `PlaygroundMouseEvent` classes are not shown.

The call to `getGraphics()` in `Click` is also replaced with the `PlaygroundApplet` version. As shown in Figure 3, the `getGraphics()` method of `PlaygroundApplet` retrieves a reference to a remote `BrowserGraphics` object, via the `getBrowserGraphics()` method of Figure 5. The `getGraphics()` method of `PlaygroundApplet` then returns this `BrowserGraphics` reference encapsulated within a `PlaygroundGraphics` object for calling it. So, when `Click` invokes `drawString()`, the arguments are passed to the browser and executed (Figure 4).

4.2 Nonserializable objects and passing by reference

In the example of the previous section, we were fortunate in that all parameters that needed to be passed across the network were *serializable*. *Object serialization* refers to the ability to write the complete state of an object to an output stream, and then recreate that object at some later time by reading its serialized state from an input stream [OSS96, Fla97]. Object serialization is central to remote method invocation—and thus to communication between the graphics server and the playground stubs—because it allows for method parameters to be passed to a remote method and the return value to be passed back. In the example of Section 4.1, the remote method invocation `bg.drawString(str, x, y)` in the `PlaygroundGraphics.drawString()` method of Figure 4 causes no difficulty because each of `str` (a string) and `x` and `y` (integers) are serializable.

However, not all classes are serializable. One case in point is the `Image` class, which represents a displayable image in a platform-dependent way. So, while the previous invocation of `bg.drawString(str, x, y)` succeeds, a similar invocation `bg.drawImage(img, x, y, ...)` fails because `img` (an instance of `Image`) cannot be serialized and sent to the graphics server. For such reasons, an object that can be passed as a parameter to a remote method of the graphics server is generally constructed *in the graphics server* originally (with a corresponding stub on the playground). Then, a reference to this object in the browser is passed to graphics server routines in place of the object itself. In this way, only the object reference, which is serializable, is ever passed over the network. This also has the benefit of avoiding the copying and marshaling costs associated with serializing complex objects.

To illustrate this manner of passing objects by reference, we continue with the example of an `Image`. When the downloaded applet calls for the creation of an `Image` object, e.g., via the `Applet.getImage()` method, our interposed `PlaygroundApplet.getImage()` passes the arguments (a URL and a string, both of which are serializable) to a remote image creation method in the graphics server. This remote method constructs the image, places it in an array of objects, and returns the array index it occupies. Playground

statements), etc.

```
public class PlaygroundApplet extends Applet implements PlaygroundAppletXface {

  BrowserXface bx;
  MouseListener ml_array[] = new MouseListener[...];
  int ml_index = 0;

  // Adds a MouseListener.  If this is the first, then register
  // this object at the graphics server as a MouseListener.
  public synchronized void addMouseListener(MouseListener l) {
    ml_array[ml_index++] = l;
    if (ml_index == 1) {
      bx.addPlaygroundMouseListener(this);
    }
  }

  // Part of the PlaygroundAppletXface remote interface.  Invoked
  // from the browser graphics server when the mouse is clicked.
  public void playgroundMouseClicked(BrowserEventXface e) {
    int i;
    PlaygroundMouseEvent pme = new PlaygroundMouseEvent(e);
    for (i = 0; i < ml_index; i++)
      ml_array[i].mouseClicked(pme);
  }

  // Returns an object that encapsulates the remote graphics
  // object of the browser applet.
  public Graphics getGraphics () {
    return new PlaygroundGraphics(bx.getBrowserGraphics());
  }
  ...
}
```

Figure 3: Part of the PlaygroundApplet class

```
public class PlaygroundGraphics extends Graphics {

  // Reference to the BrowserGraphics remote object in the graphics server.
  private BrowserGraphicsXface bg;

  // The constructor for this object.  Calls its superclass constructor
  // and saves the reference to the BrowserGraphics remote object.
  public PlaygroundGraphics(BrowserGraphicsXface b) {
    super();
    bg = b;
  }

  // Invokes drawString in the graphics server.
  public void drawString(String str, int x, int y) {
    bg.drawString(str, x, y);
  }
  ...
}
```

Figure 4: Part of the PlaygroundGraphics class

```
public class BrowserServer extends Applet implements BrowserXface, MouseListener {

  // Reference to the MouseListener object on the playground
  PlaygroundMouseListenerXface ml;

  // Part of the BrowserXface remote interface.  Invoked from
  // the playground to add a remote MouseListener.
  public void addPlaygroundMouseListener(PlaygroundMouseListenerXface pml) {
    ml = pml;
    addMouseListener(this);
  }

  // Invoked whenever the mouse is clicked.  Passes the event
  // to the MouseListener on the playground.
  public void mouseClicked(MouseEvent event) {
    ml.playgroundMouseClicked(new BrowserEvent(event));
  }

  // Returns a remote object that encapsulates the graphics
  // context of this applet.
  public BrowserGraphicsXface getBrowserGraphics() {
    Graphics g = getGraphics();
    return new BrowserGraphics(g);
  }
  ...
}
```

Figure 5: Part of the BrowserServer class

```
public class BrowserGraphics extends Graphics implements BrowserGraphicsXface {

  private Graphics g;

  // The constructor for this class.  Calls the superclass constructor,
  // saves the pointer to the "real" Graphics object (passed in), and
  // exports its interface to be callable from the playground.
  public BrowserGraphics(Graphics gx) {
    super();
    g = gx;
    UnicastRemoteObject.exportObject(this);
  }

  // Part of the BrowserGraphicsXface remote interface.  Invoked
  // from the playground to draw a string.
  public void drawString(String str, int x, int y) {
    g.drawString(str, x, y);
  }
  ...
}
```

Figure 6: Part of the BrowserGraphics class

objects then pass this index to remote graphics server methods in place of the image itself. For example, the `BrowserGraphics` class in the graphics server implements versions of the `drawImage()` method that accept image indices and display the corresponding `Image`.

Conversely, there are circumstances in which objects that need to be passed as parameters to remote methods *cannot* first be created in the browser. This can be due to security reasons—e.g., the object's class is a user-defined class that overwrites methods of an AWT class—or because the class of the parameter object is unknown (e.g., it is only known to implement some interface). In these circumstances, a reference is passed in the parameter's place, and method invocations intended for the object are passed back to the playground object for processing. Continuing with our `Image()` example, such "callbacks" can occur when the downloaded applet applies certain *image filters* to an image before displaying it. One such filter is an `RGBImageFilter`: A subclass of `RGBImageFilter` defines a per-pixel transformation to apply to an image by overwriting the `filterRGB()` method. To avoid loading untrusted code in the browser, such a filter must be executed on the playground with callbacks to its `filterRGB()` method.

In some circumstances, the need to pass objects by reference can considerably hurt performance. Continuing with the `RGBImageFilter` example above, filtering an image may require that *every image pixel* be passed from the browser to the playground, transformed by the `filterRGB()` method, and passed back. This can result in considerable delay in rendering the image, though our experience is that this delay is reasonable for images whose pixel values are indices into a colormap array (i.e., for images that employ an `IndexColorModel`).

4.3 Addressing

The previous sections described how an applet running on the playground is coerced into using the user's browser as its I/O terminal. Before any I/O can be performed at the browser, however, the applet running on the playground and the graphics server running in the user's browser must be able to find each other to communicate. This is complicated by the fact that an HTML page can contain any number of `<applet>` tags that, when modified by the proxy, result in multiple instances of the graphics server running in the browser. To retain the intended function of the page, it is necessary to correctly match each applet running on the playground with its corresponding instance of the graphics server in the browser.

The addressing scheme that we use requires that the proxy make additional changes to the HTML page containing applet references prior to forwarding it to the browser. Specifically, if the page contains an `<applet>` tag of the form

```
<applet code=hostile.class ...>
```

then the proxy not only replaces `hostile.class` with `BrowserServer.class` (as described in Section 3), but also adds a *parameter tag* to the HTML page, like this:

```
<applet code=BrowserServer.class ...>
<param name=ContactAddress value=address>
```

Parameter tags are tags that contain name/value pairs. This one assigns an *address* value, which the proxy generates to be unique, to be the value of `ContactAddress`. The `<param>` tags that appear between an `<applet>` tag and its terminator (`</applet>`, not shown) are used to specify parameters to the applet when it is run. In this case, the `BrowserServer.class` object (i.e., the remote Applet object of the graphics server; see Section 3.1.2) looks for the `ContactAddress` field in its parameters and obtains the address assigned by the proxy. Once the `BrowserServer` object is initialized and prepared to service requests from the playground, it binds a remote reference to itself to the address assigned by the proxy; this binding is stored in an RMI name server [RMIS97].

The proxy remembers what address it assigned to each `<applet>` tag and provides this address to the playground in a similar fashion. That is, the proxy loads applets into the playground by sending to the playground an HTML page with identical `ContactAddress` `<param>` tags to what it forwarded to the browser (for simplicity, this step is not shown in Figure 3). A JVM on the playground loads the referenced applets and uses the corresponding `<param>` tags provided with each to look up the corresponding graphics servers in the RMI name server.

5 Security

5.1 Requirements

The security of our approach relies on at least the following two distinct requirements.

1. Prevent the JVM in the user's browser from loading any classes from the network. If this is achieved, then untrusted code can never be loaded into the browser (unless the browser machine's local class files are maliciously altered, a possibility that we do not consider here). Later in this section we describe how by disabling network class loading, many classes of attacks that have been successfully mounted on JVMs are prevented by our system.

2. Prevent untrusted applets running on the playground from accessing valuable resources. Because we assume that untrusted applets might circumvent the sandbox protections on the playground, this requirement can be met only by relying on underlying operating system protections on the playground, or preferably by isolating the playground from valuable user resources.

In the next two subsections, we describe alternatives for meeting these requirements in a playground system.

5.1.1 Preventing network class loading

Preventing network class loading by the browser can be achieved in one of two ways in our system.

Trusted proxy One approach is to depend on the proxy to rewrite all incoming `<applet>` tags on HTML pages to point to the graphics server class (stored locally to the browser), and to further intercept and deny entry from any class files destined for protected machines (as in [MRR97]). Rewriting all `<applet>` tags ensures that the browser is never directed by an incoming page to load anything but the trusted graphics server. Moreover, if the playground passes an unknown class to the graphics server (e.g., as a parameter to a remote method call) the graphics server is unable to load the class because the class' passage is denied by the proxy. The advantage of this approach is that it works with any browser "off-the-shelf": it requires no changes to the browser beyond specifying the proxy as the browser's HTTP proxy, which can typically be done using a simple preferences menu in the browser. A disadvantage, however, is that the proxy becomes part of the trusted computing base of the system. For this reason, the proxy must be written and protected carefully, and we refer to this approach as the "trusted proxy" approach.

Untrusted proxy A second approach to preventing the browser from loading classes over the network is to directly disable network class loading in the browser. The main disadvantage of this approach is that it requires either configuration or source-code changes to all browsers in the protected domain. In particular, for most popular browsers today (including Netscape Communicator and Internet Explorer 4.0), a source-code change seems to be required to achieve this, but we expect such changes to become easier as browser's security policies become more configurable. A strong advantage of this approach is that it excludes the proxy from the trusted computing base of the system, and hence we refer to this as the "untrusted proxy" approach. In fact, in this approach only the browser and the graphics server classes are in the trusted computing base.

To more precisely understand how the above approaches prevent network class loading, below we describe the circumstances that can cause classes to be loaded from the network, and how the approaches described above prevent this from happening.

1. Section 3 briefly described the process by which a browser loads an applet specified in an `<applet>` tag in an HTML page. As described in Sections 3 and 4.3, the proxy modifies the `code` attribute of each `<applet>` tag to reference the trusted graphics server applet that is stored locally to the browser. Thus, the browser is coerced into always loading the graphics server applet as its applet. In the untrusted proxy approach, this coercion serves merely a functional purpose—i.e., running the graphics server to which the playground applet can connect—but security is not threatened if the proxy fails to rewrite the `<applet>` tag. In the trusted proxy approach, this coercion is central to both security and function.

2. Once an applet is loaded and started, the *applet class loader* in the browser further loads any classes referenced by the applet. In the case that a class is not in the core Java library or stored locally, the applet class loader would normally retrieve the class over the network. However, in our approach, the applet class loader needs to load only local classes, because the graphics server applet refers only to other local classes.

3. As described in Section 3, the modified applet on the playground invokes remote methods in the graphics server. Because our present implementation uses RMI to carry out these invocations, the *RMI class loader* can load additional classes to pass parameters and return values. As above, the RMI class loader first looks in local directories to find these classes before going to the network to retrieve them. It is possible that the playground applet passes an object whose class is not stored locally on the browser machine. In the trusted proxy approach, the RMI class loader goes to the network, via the proxy, to retrieve the class, but the class is detected and denied by the proxy. In the untrusted proxy approach, the RMI class loader returns an exception immediately upon determining that the class is not available locally.

5.1.2 Isolating untrusted applets from valuable resources

Once untrusted applets are diverted to the playground, security relies on preventing those applets from accessing protected resources. A first step is for the playground's JVMs (and thus the applets) to execute under an account different from actual users' accounts and that has few permissions associated with it. For some resources, e.g., user files available to the playground, this achieves security equivalent to that provided by the access control mechanisms of the playground's operating system. (Similarly, different JVMs on the playground execute under different accounts, to reduce the threat of inter-JVM attacks; see Section 5.3.)

If the complete compromise of the playground is feared, then further configuration of the network may provide additional defenses. For example, if network file servers are configured to refuse requests from the playground (and if machines' requests are authenticated), then even the total corruption of the playground does not immediately lead to the compromise of user files. Similarly, if all machines in the protected domain are configured to refuse network connections from the playground, except to designated ports reserved for browsers' graphics servers to listen, then the compromise of the playground should gain little for the attacker.

In the limit, an organization's playground can be placed outside its firewall, thereby giving applets no greater access than if they were run on the servers that served them. However, because most firewalls disallow connections from outside the firewall to inside, additional steps may be necessary so that communication can proceed uninhibited between the graphics server in the browser and the applet on the playground. In particular, RMI in Java 1.1 (used in our prototype) opens network connections between the browser and the playground in both directions, i.e., from the browser to the playground and vice versa. One approach to enable these connections across a firewall is to multiplex them over a single connection from the graphics server to the playground (i.e., from inside to outside). This can be achieved relatively easily if both the graphics server and the playground applet interpose a customized connection implementation (specifically by changing the `SocketImplFactory`), but for technical reasons this does not appear to be possible with all off-the-shelf browsers (e.g., it appears to work with HotJava 1.0 but not Netscape 3.0). Another alternative is to establish reserved ports on which graphics servers listen for connections from playground applets, and then configure the firewall to let connections to those ports through.

5.2 RMI security

Though the requirements discussed in Section 5.1 are necessary for security in our system, they may not be sufficient. In particular, Java's Remote Method Invocation mechanism, which forms the heart of playground-to-browser communication in our present implementation, is a relatively new technology that could conceivably present new vulnerabilities. A first step toward securing RMI is to support authenticated and encrypted transport, so that a network attacker cannot alter or eavesdrop on communication between the browser and the playground. This can be achieved by interposing an encrypting connection implementation or similar protections at the object serialization layer (see [OSS96]).

A more troubling threat is possible vulnerabilities in the object serialization routines that are used to marshal parameters to and return values from remote method invocations. In the worst case, a corrupted playground could conceivably send a stream of bytes that, when unserialized at the browser, corrupt the type system of the JVM in the browser. Here our decision to generally pass only primitive data types (e.g., integers, strings) as parameters to remote method invocations (see Section 4.2) would seem to be fortuitous, as it greatly limits the number structurally interesting classes that the attacker has at its disposal for attempting such an attack. However, the possibility of a vulnerability here cannot yet be ruled out, and we and several other research efforts are presently examining RMI in an effort to identify and correct such problems. We view this process of public scrutiny as one of the main advantages to building our system from public and widely used components.

5.3 Resistance to known attacks

One way to assess the security of our approach is to examine it in the face of known attack types. In this section we review several classes of attack that applets can mount, and describe the extent to which our system defends against them.

Accessing and modifying protected resources Several bugs in the type safety mechanisms of Java have provided ways for applets to bypass Java sandboxes, including some in popular browsers [DFW96, MF97]. These penetrations typically enable the applet to perform any operation that the operating system allows, including reading and writing the user's files and opening network connections to other machines to attack them. We anticipate that in the foreseeable future, type safety errors will continue to exist, and therefore we must presume that applets running on our playground may run unconstrained by the sandbox. However, they are still confined by the playground operating system's protections and, ultimately, to attack only those resources available to the playground. In Section 5.1.2 we described several approaches to limit what resources are available to applets on the playground. We expect that through proper network and operating system configuration, hostile applets can be effectively isolated from protected resources.

Denial of service In a denial of service attack, a hostile applet might disable or significantly degrade access to system resources such as the CPU, disk, network and interactive devices. This form of attack is typically considered very difficult to protect against, since it can be mounted simply by over-utilizing resources to which the applet has legitimate access. Ladue [Lad96] presents several such applets, e.g., that consume CPU even after the user clicks away from the applet origin page, that monopolize system locks, or that pop up windows on the user's screen endlessly. Using our Java playground, most of these applets have no effect on the protected domain, and only affect the playground machine. However, an applet that pops up windows endlessly causes the graphics server running in the user's browser to create an infinite stream of windows. Uncontrolled, this may prevent access to the user terminal altogether and require that the user reboot her machine or otherwise shut off her browser. One approach to defend against this is to configure the graphics server and/or the playground to limit the number or rate of window creations.

In another type of denial of service, an applet may deny service to other applets within the JVM, e.g., by killing off others' threads [Lad96]. Although the sandbox mechanisms of most browsers are intended to separate applets in different web pages from one another, several ways of circumventing this separation have been shown [DFW96, Lad96]. In our system, applets from each downloaded page run in a separate JVM on the playground under a separate user account, and hence cannot directly affect applets from another page (except by attacking the playground itself).

Violating privacy The Java security policy in browsers, e.g., Netscape and Internet Explorer, is geared towards maintaining user privacy by disallowing loaded applets access to any local information. In practice, however, a Java applet can reveal a lot about a user whose browser executes it. For example, the applet can connect back to the server that served it, thus revealing the IP address of the user's machine. Note that such a connection is direct, thus bypassing any anonymity-providing proxies that the user might employ, such as the Anonymizer (see `http://www.anonymizer.com`). In [Lad96], Ladue presents an applet that uses a *sendmail* trick to send mail on the user's behalf to a sendmail daemon running on the applet's server. When this applet is downloaded onto a Unix host (running the standard *ident* service), this mail identifies not only the user's IP address, but also the user's account name. In our system, the applet runs on the playground machine under an account other than the user's, and the information that it can reveal is limited only to what is available on the playground.

6 Limitations

As mentioned in Section 1, in our experience, our system is transparent to users for most applets. There are, however, classes of applets for which our playground architecture is not transparent, and indeed our system may be unable to execute certain applets at all. In particular, the remote interface supported by the graphics server supports the passage of certain classes as parameters and return values of its remote methods. If the code running on the playground attempts to pass an object parameter whose class is an unknown subclass of the expected parameter class, then the browser is required to load that subclass to unserialize that parameter. However, because class loading from the network is prevented in our system (see Section 5.1.1), the load does not complete and an exception is generated. At the time of this writing, the number of applets that cannot be supported due to this limitation does not seem significant; indeed, we have yet to encounter an applet for which this is a problem.

A second limitation of our approach is that by moving the applet away from the machine on which the user's browser executes, the applet's I/O incurs the overhead of communicating over the network. Our experience indicates that for many applets, this cost is barely noticeable over typical local area network links such as a 10-Mbit/s Ethernet. However, for applets whose output involves intensive I/O operations, e.g., low-level image filtering (see Section 4.2), this overhead can be considerable.

Finally, our playground approach is limiting for applets intended to perform massively parallel computations on a vast array of machines, because our system centralizes applet executions on one or a few playgrounds. Examples of such intensive computations include large matrix computations (see, e.g., [BKKW96]) or cryptanalysis of ciphers.

The emergence of more functional applets will undoubtedly raise new challenges in transparently executing them on a playground. For example, one can envision a text editor applet that can be downloaded from the network and then used to compose a document (and save it to disk). Such an applet is inconsistent with the sandbox policies implemented in Netscape 3.X, Internet Explorer 3.X, and JDK 1.1, because network-loaded applets are not allowed to write files. However, it may well be possible with the more flexible policies implemented in recently (or soon-to-be) released versions of these products (e.g., [WBDF97]). Our system, as presently designed, would presumably constrain the applet to save the document on the playground. The user would then have to retrieve the file from the playground by other means.

7 Conclusion

This paper presented a novel approach to protecting hosts from mobile code and an implementation of this approach for Java applets. The central idea behind our approach is to execute mobile code in the sanitized environment of an isolated machine (a "playground") while using the user's browser as an I/O terminal. We gave a detailed account of the technology to allow transparent execution of Java applets separately from their graphical interface at the user's browser. Using our system, users can enjoy applets downloaded from the network, while exposing only the isolated environment of the playground machine to untrusted code.

Although we presented the playground approach and technology in the context of Java and applets, other mobile-code platforms may also utilize it. In the future, we plan to explore how to employ our ideas in other such platforms.

Acknowledgements

We are grateful to Ed Felten for helpful discussions regarding this work.

References

[BKKW96] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. *In Proc. 9th Intl. Conf. on Parallel and Distributed Computing Systems*, 1996.

[DFW96] D. Dean, E. W. Felten, and D. S. Wallach. Java security: From Hotjava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[Fla97] D. Flanagan. *Java in a Nutshell*, Second edition, O'Reilly & Associates, 1997.

[Her97] A. Herbert. *Secure mobile code management: Enabling Java for the enterprise.* Manuscript, May 1997.

[Lad96] M. Ladue. Pushing the limits of Java security. In *Tricks of the Java Programming Gurus*, G. Vanderburg, ed., Sams.net Publishing, 1996.

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, Addison-Wesley, 1997.

[MF97] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*, John Wiley & Sons, 1997.

[MRR97] D. Martin, S. Rajagopalan, and A. D. Rubin. Blocking Java applets at the firewall. In *Proceedings of the 1997 Internet Society Symposium on Network and Distributed System Security*, pages 16–26, February 1997.

[NL96] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 229–243, October 1996.

[OSS96] Sun Microsystems, Inc. *Java Object Serialization Specification*, Revision 1.2, December 1996.

[RMIS97] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*, 1997.

[WBDF97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.