# A Java Filter

Dirk Balfanz
Princeton University
balfanz@cs.princeton.edu

Edward W. Felten
Princeton University
felten@cs.princeton.edu

## Abstract

Rogue Java applets are currently a major concern for big companies and private users alike. While the best protection against them is to turn off Java support in the WWW browser, this "solution" is unsatisfying: it deprives users of many of the advantages of the Java platform. Other mechanisms such as firewalls and code signing have been proposed to "enhance" security. In this paper we argue that these mechanisms cannot deliver the security they promise. As an alternative, we describe a simple yet effective way to prevent untrusted applets from entering the user's computer. At the same time, we allow trusted applets to execute in whatever sandbox the browser provides for them. Our technique works by modifying Java class loaders and can be extended to provide fine-grained access control for Java applets.

## 1 Introduction

One especially interesting topic in the realm of Internet security is that of mobile code. While users browse the World Wide Web, programs — known variously as applets, controls, or scripts — are downloaded onto their computers as part of Web pages they are viewing. The result is that untrusted programs are run on the user's machine. The system must protect the user against programs that have hostile intentions. For example, the mobile programs must be prevented from reading personal information or altering the system's state.

This is why mobile code technologies such as ActiveX [1] provide a line of defense: before downloading the mobile code, the user is asked whether or not she really wants to install the software in question. The code can be digitally signed to give the user some evidence as to who wrote (or endorsed) it. Security "breaches" of ActiveX take two forms: either the user is tricked into accepting hostile code, or an ActiveX program can (by exploiting some sort of implementation error or oversight) transport itself onto the user's system without the user's consent. Once an ActiveX control is installed on a machine, it can do whatever it wants.

Java has taken a different approach to mobile code security. The Java platform is designed in such a way that all system calls made by a Java program (Java applet) must be routed through a *security manager*, which can decide whether or not certain sensitive operations should be allowed. In the past, the security manager would deny applets almost any operations that would give them access to the local system or the network, except to the network host they were loaded from. This had the effect that Java applets would be executed in a "sandbox" which they could not leave. Recently, the security manager has been augmented with fine-grained access control mechanisms that allow it to make decisions based on who signed the applet and/or where it was loaded from [5, 9]. However, it is still the case that every security-relevant system call is routed through the security manager, and that the security manager may deny invocation of the respective system call. Breaking Java security, then, means being able to access the system (by bypassing the security manager or some other means) when the security policy would have forbidden that access.

Again, due to implementation errors, oversights, or design flaws, ways have been found to break Java security. The seriousness of possible attacks ranged from unauthorized network access to the ability to call any system call, thus completely exposing the system [3, 6]. All in all, there have been at least six flaws in various Java versions that would have allowed an attacker to completely defeat the security of the Java Virtual Machine. While all of the reported bugs were fixed promptly, often within a few days, the threat of malicious Java applets that might exploit undiscovered security holes remains.

Java also suffers from "denial-of-service" attacks, in which an applet does not penetrate the system but merely crashes it or makes it unusable. This is often done via allowed operations (ones permitted by the security manager), for example by opening 50 windows per second until the window manager software dies. Denial-of-service attacks do not constitute a breach of security, but they are a concern for users. So far, no commercial Java system has provisions against denial-of-service attacks.

### 1.1 Approaches to Java Security

There are two basic strategies for protecting Java users against malicious applets: strengthen the sandbox, or prevent potentially hostile applets from running.

Strengthening the sandbox means improving the design and implementation of the Java system, including formal work to increase assurance by proving the foundations of the system to be sound. While much valuable work (e.g. [4, 2]) follows this approach, it does not solve the problem for everybody. There are two reasons for this. First, while most users are comfortable with the current level of assurance, some are not. Second, preventing the full range of possible denial of service attacks is a very difficult problem that is probably beyond the current state of the art.

The second approach is to stop potentially hostile applets from running in the first place. It is easy to do this by turning off Java entirely, but this prevents even the safest locally-developed applets from running. A midway point is to attempt to "filter" applets based on their origin or on who has digitally signed them. This is the approach we follow.

Many applet filtering mechanisms have been proposed, but, as we will explain below, many of them are overly complex or are effective only against the most obvious attacks. In this paper we will present a simple and effective applet filtering mechanism which we have implemented for Netscape Navigator 3.x and Microsoft Internet Explorer 3.x.

The paper is structured as follows: in Section 2 we will describe current methods for protecting systems from Java applets, and discuss their shortcomings. Since our method uses class loaders, in Section 3 we will give a brief overview of how a Java class loader works. In Section 4 we will present our system, the *Java Filter*.

## 2 Protecting Your System (Not!)

### 2.1 Switching off Java

Whenever people find a new security hole in the Java system, they usually recommend to switch off Java support in Web browsers until the bug is fixed. This is in fact an effective way to protect the user's system: Switching off Java support has the effect that Java applets will not be fetched from across the network and installed on the local machine. Malicious applets can therefore cause no harm whatsoever.

Alas, switching off Java support of course has an important side-effect: It switches off Java support! If a company or government agency decides that Java applets are potentially harmful and that an effective measure against them should be deployed, this would unfortunately also mean that the company or agency decides against employing useful Java technology altogether.

While switching off Java support is an effective way to protect oneself from malicious applets, it is no solution for someone who wants to use Java technology in their intranets.

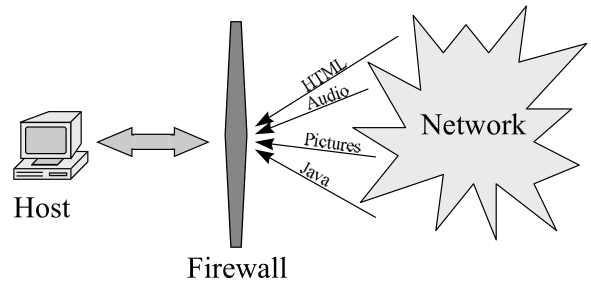


Figure 1: A firewall has to block all Java code, while permitting other data.

### 2.2 Firewalls

Usually, companies find Java applet technology very useful for building WWW frontends to legacy systems, group ware, and other things that are used within the intranet. These applets are written, and used, *within* the company. They are *trusted* not to be vicious. On the other hand, there is a great number of *untrusted* applets out there on the Internet, written by people and with intentions that we don't know. It would be desirable to configure browsers so that they accept applets from the company's intranet, but refuse to load applets from the Internet.

One way to achieve this goal is to filter applets out at the firewall (see Figure 1). Several firewall vendors ship Java filtering technology (e.g [8]) as part of their firewalls. Unfortunately, these products cannot guarantee full protection from unwanted applets as it is very hard to detect every applet that tries to sneak its way through the firewall. Here are a few techniques that try to detect Java applets at the firewall and ways to circumvent them:

- The first idea is to look for Java class files. They can be recognized by a magic byte sequence that is required at the beginning of every class file.

  **Pitfall:** Java class files may come as part of an compressed archive (e.g. Jar or Cab files). Due to the nature of compression, nothing in the archive (not necessarily even its name) exposes the fact that it contains Java class files. Class files that are part of an archive cannot be detected by this technique. In addition, class files may be passed via an encrypted (SSL) connection, which will make them indistinguishable from ordinary files to the firewall.

- Java class files can be recognized by their name, which will end in "`.class`".

  **Pitfall:** Depending on the browser, this may either not be the case or, again, be circumvented by sending applets as part of an archive.

- HTML pages can be rewritten at the firewall so that no "`Applet`" tags are left in the HTML file. This will have the effect that the browser will never ask for an applet to be fetched across the firewall.

  **Pitfall:** Javascript can be used to build `Applet` tags on the fly. Although there is no `Applet` tag in the HTML file, the browser's executing of Javascript will cause it to be inserted at the time the page is viewed.

### 2.3 Code Signing

When Sun Microsystems and the major browser companies switched from the simple sandbox security model to fine-grained access control based on signed code, they sometimes tried to give the impression that this would "enhance" security. Code signing cryptographically binds a certain principal, such as an individual or company, to a piece of code. It is not feasible to change a signed piece of code without being detected, nor is it possible to impersonate someone else by augmenting code with their signature.

With this technology in place, it is always possible to find out who signed the code. The argument goes that now users can decide that code signed by untrusted or unknown parties should be denied privileges.

However, the browser will still *load* "untrusted" code, the difference is only that the untrusted code will run with fewer privileges. Untrusted code is usually run within the old sandbox (i.e. no file system access, no network access except back to the originating host). But we have already seen that code can exploit implementation bugs and break out of that sandbox. Code signing provides no enhanced security at this point.

Even worse, code signing introduces new system code which can have, and in fact has been found to have[1], new bugs that weren't there before. This opens even more doors for malicious applets. On the trade-off continuum between security and functionality, code signing belongs on the functionality side, as it can be used to provide fine grained access control and provides the user with information about who endorsed a specific piece of code. Code signing does not "enhance" security.

[1] see `http://www.cs.princeton.edu/sip/news/april29.html`

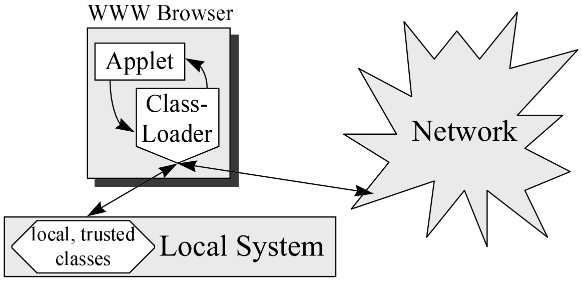


Figure 2: The normal workings of a class loader.

## 3 Class Loaders

In this paper we will present a system that allows users who surf the World Wide Web to selectively "switch off" Java support in their browsers. For untrusted applets, this will be as effective as the solution discussed in Section 2.1, but trusted applets can still be used. Since our system uses the Java class loader, we will here give a brief overview about what a class loader is and how it works.

In Java, linking is done at runtime. Whenever execution of Java bytecode requires the presence of a new class (e.g. because one of its methods is invoked), then a *class loader* is invoked to fetch the class material (the class file). It is up to the class loader how to get the class material. Once fetched, the class is added to the runtime system and can then be used by the Java program.

WWW browsers have special class loaders, so-called *applet class loaders*. When an `Applet` tag is encountered on a Web page, a new applet class loader is created for that applet.[2] Whenever a new class needs to be linked into the system, the applet class loader will first try and load the class from the local `CLASSPATH`. Only if it doesn't find the class locally, it will go to the origin of the applet and fetch the class material across the network. The first class that it fetches is the class mentioned in the `Applet` tag of the HTML page.

Figure 2 summarizes how a class loader works: When an applet is detected as part of a Web page, an applet class loader is created that subsequently has to load all the classes for that applet, either from the local system or across the network.

## 4 Guarding the Door

### 4.1 How it Works

Our system is very simple: We prevent the creation of

[2] This is not quite correct. If there already was an applet on the same page that came from the same place as the new applet, its class loader can be used.

| Action | URL |
|---|---|
| allow | `http://www.javasoft.com:80/applets/WordMatch/` |
| deny | `http://www.javasoft.com/` |
| allow | `http://www.packet.com/java/hotwired/` |
| deny | `http://www.earthweb.com/java/Thingy/` |
| deny | `http://www.cruzio.com/˜sabweb/arcade/` |

Figure 4: A sample list of URLs.

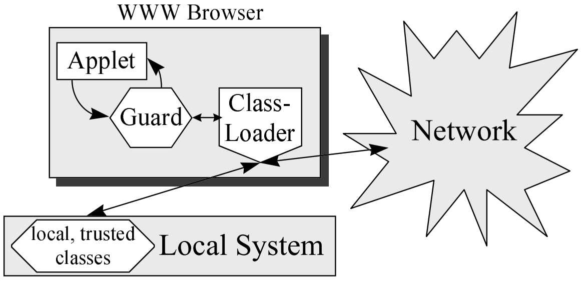


Figure 3: A guard object decides whether or not to create a class loader.

a class loader for untrusted applets. Assume that we come across an `Applet` tag on a Web page. The runtime system will try to create a class loader for that applet by creating a new instance of class `AppletClassLoader`. One of the arguments passed to that call is the URL of the applet. If the applet class loader decides that this URL is untrusted, it can then throw an exception. We note that this exception is thrown during the creation of the class loader. This has the effect that the class loader will not be created at all. When there is no class loader, no-one can fetch classes across the network. We see that for untrusted applets this is as effective as switching off Java support altogether: The remote classes don't even enter the local system. For trusted applets, our class loader works as before.

Figure 3 shows how the system works: We introduce a "guard" object that has the task of deciding whether or not a given URL is trusted. During creation of the class loader, the guard object is consulted, which may or may not cause an exception to be thrown. If no exception is thrown, the class loader just works like before. If an exception is thrown, then the creation of the class loader fails. This corresponds to there being no class loader at all in the picture.

We only distinguish between "trusted" and "untrusted" URLs since our goal is binary as well: We either want to load the applet (and subject it to whatever security policies the browser provides) or we don't want to load it at all.

We therefore maintain a simple ordered list of URLs and whether or not applets from that URL should be downloaded. Figure 4 shows an example. The guard object will go down the list and compare the URL of the applet that is about to be created with the URLs in the list. If it finds a match, then it applies the specified action. Note that the URLs in the list cover all their subdirectories. In our example, an applet from `http://www.packet.com/java/hotwired/news/` would be allowed because of the third entry in the list. Also note that in our example, all applets from `http://www.javasoft.com/` would be denied except those from `http://www.javasoft.com:80/applets/WordMatch/`, because the list is processed from top to bottom.

The list of URLs is saved on the local file system and thus preserved across browsing sessions. The file used for storing this list is human-readable and can be edited. For example, we could specify that all applets from within the company are allowed, but no others, by saying something like this:

```
allow   http://www.mycompany.com/
deny    http://
deny    ftp://
deny    https://
```

Since the last three lines match any possible applet[3] only applets from `http://www.mycompany.com/` will be allowed on the machine.

### 4.2 How it looks

What if none of the URLs in the list match the given applet? Then some user intervention is necessary. As with all security relevant software, the design of the graphical user interface is crucial. We don't want users to be annoyed by windows popping up too often to warn them about potential security risks. Also, we want to make is easy for the non-expert to make security-relevant decisions, while giving the expert full control over the system.

When we first come across an applet that doesn't match any URL in the list, the dialog shown in Figure 5 will pop

[3]Depending on what browser is used, more protocols like `gopher:` etc. may have to be included.


Figure 5: This dialog that pops up when an unknown applet is discovered.

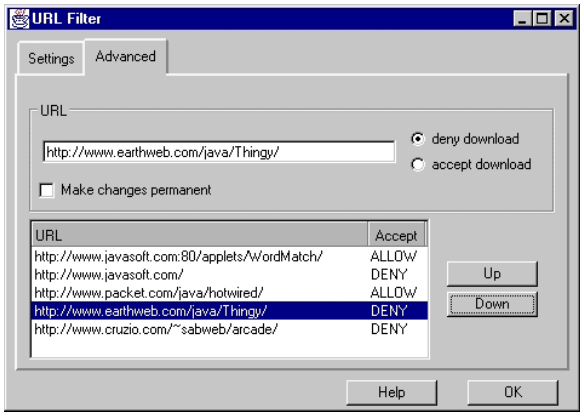


Figure 6: Advanced version of the dialog.

up. It displays the URL the applet is about to be loaded from, and asks to answer two questions: First, we have to decide whether or not we want this applet to run on our machine. The default is not to run the applet. Second, we should decide whether that decision is to be "remembered" for the future, i.e. whether the URL should be included in the file that is saved for future use. The default is not to remember that decision, which will cause the browser to ask the same question again if and when we visit the same applet during a future session. During a browser session, we will usually not be asked again for a specific applet. If we choose to make the settings permanent, then the applet in question will always be allowed on our system (or be denied entrance - depending on what we chose) - unless or until we manually change the file that holds the list of URLs.

We can have more advanced control over which applets should be allowed into our system, and which applets should be blocked. When we click on the "Advanced" tab bar, the dialog changes to what we see in Figure 6.

The upper half of the dialog lets us answer the same questions that we saw in the "easy" version. The field displaying the URL is now editable. We can change this URL to whatever we like: If we come across an applet from `http://java.sun.com/applets/WordMatch/` we can decide that we trust all applets from JavaSoft and edit the URL to show `http://java.sun.com/`. This edited URL, and not the original one, will be included in the list of URLs against which the guard object checks applet URLs.

The lower half of the dialog shows the URLs known so far to the system, and whether or not we allowed access for them. Since the list of URLs is searched from top to bottom, the order in which the URLs occur matters. The user can change the priority of the current URL within that list.

### 4.3 Implementation

We implemented the Java Filter for Netscape Navigator 3.0 and Microsoft Internet Explorer 3.0 on the Windows platform. Most of the code could be shared, although access to the Windows registry and file system (for saving the list of URLs) turned out to be slightly different on each platform. The solution is not "100% pure", since we needed native methods to access file system and Windows registry in Navigator and used Microsoft-specific classes for Internet Explorer. The installation, which can be downloaded from our Web site,[4] will change the applet class loader of the installed browsers and add necessary classes to the class library.

#### 4.3.1 Changing the Class Library

How did we change the browser's class library, in particular the class loader so that it employs our guard object? Remember that we do not have source code of the class loader. There are several ways to do this:

- One obvious way is to extract the class loader from the class library, decompile, change the Java code, compile it, and inject it back into the class library (which is a ZIP file). This has some legal issues (which we will not address in this paper) and is also difficult: A number of decompilers that we tested would either fail on the task of decompiling the class loader or produce obviously incorrect code. However, if a decompiler is available that does the job, this is a valid measure.
- Since our changes to the class loader are minimal (we basically add one line of code in the constructor) we

[4] `http://www.cs.princeton/sip/`

could theoretically change the class file directly. Although we only add a few bytecode instructions, we need a program that can parse and write class files: Injecting only a few bytecode instructions can cause complex changes in the class file (e.g. constant pool, attributes).

- One technique that we used is to make trivial changes to the class file of the `AppletClassLoader` to the effect that the class file then represents a class of a different Name (for example `XppletClassLoader`). Then we create a new Java class, called `AppletClassLoader` that *subclasses* the original `AppletClassLoader` (now `XppletClassLoader`). In its constructor it calls the original constructor and the guard object (which may or may not throw an exception). Since we inherit all non-private methods, to the runtime system the new class is a perfectly valid applet class loader.[5]

The other classes that are needed for the Java Filter can just be added to the class library.

We note that these techniques can be used to enable Java filtering not only for mainstream Web browsers, but also for other types of (Java) applications that execute Java applets but want to be protected from attacks exploiting weaknesses in the Java implementation.

## 5 Future Work

Since the Java Filter is not a pure Java implementation, ports to other platforms like the Macintosh or UNIX require additional work. Feedback we got from users who downloaded the Java Filter suggested that we should write versions of the Java Filter for those platforms.

Also, the current URL pattern matching is not very sophisticated. Sometimes companies have many Web servers and would like to have a line like this in their URL list:

allow `http://*.mycompany.com/`

We are looking into providing support for regular expressions for that purpose.

In recent versions of their browsers, Netscape and Microsoft have included code signing and features that are similar to what the Java Filter offers. We still believe that not downloading classes in the first place is the best protection against malicious Java applets. However, we are not certain whether a version of the Java Filter for the latest versions of Navigator and Internet Explorer is desirable (or feasible, for that matter: the class libraries are now signed and can no longer be easily altered).

[5] This turned out to be valid only for Internet Explorer. In Netscape this technique did not work, presumably because the runtime system relies on certain specific internals of the class loader like private fields etc.

## 6 Related Work

In Section 2 we saw which measures are sometimes taken to "enhance" security. However, we also saw that these measures are either inconvenient or not effective. Microsoft has recently included the notion of *security zones* in their Internet Explorer [7]. Depending on what security zone an applet comes from, it can run with more or less privileges. Unlike the systems that Netscape and Sun developed, the settings for a security zone can actually specify that Java code should not be downloaded at all.

This makes Microsoft's system very similar to our Java Filter, except that Microsoft's system is bigger and includes more features. On the other hand, the concepts and tools presented in this paper are simple and general enough to be applied almost without changes to any type of application that executes remote Java code.

We also note that the concept of using class loaders to control Java security can be extended to provide fine-grained access control. Instead of blocking the applet, we could (again, by using class loaders) block certain system classes from being linked against the applet, thus denying access to certain parts of the system. See [9] for details on this.

## 7 Conclusion

We presented a simple, yet effective protection against untrusted Java applets that doesn't have many of the shortcomings of other possible approaches. We implemented the design and have made available a copy for public download.


## Acknowledgements

Our work is supported by donations from Sun Microsystems, Bellcore, Microsoft, and Merrill Lynch. Edward Felten is supported in part by an NSF National Young Investigator award.


## References


[1] Microsoft Corporation. Proposal for authenticating code via the internet. `http://www.microsoft.com/security/tech/authcode/authcode-f.htm`, April 1996.

[2] Drew Dean. The security of static typing with dynamic linking. In *Fourth ACM Conference on Computer and*

*Communications Security*, Zurich, Switzerland, April 1997.

[3] Drew Dean, Edward Felten, and Dan Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of 1996 IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.

[4] S. Drossopoulou and S. Eisenbach. Java is type safe – probably. In *Proceedings of the Eleventh European Conference on Object-Oriented Programming*, June 1997.

[5] Li Gong and Roland Schemers. Implementing protection domains in the Java Development Kit 1.2. In *The Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998. Internet Society.

[6] Gary McGraw and Edward Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley and Sons, 1996.

[7] Microsoft Corporation, Redmond, Washington. *Microsoft Security Management Architecture White Paper*, May 1997. `http://www.microsoft.com/ie/security/ie4security.htm`.

[8] Finjan Software. SurfinGate. `http://www.finjan.com/products/html/surfingate.html`.

[9] 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*, Saint-Malo, France, October 1997.