Coping with Denial of Service due to Malicious Java Applets *

Maria Felicia Florio   Gianluigi Marchetti   Roberto Gorrieri

Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy, e-mail: gorrieri@cs.unibo.it

Abstract

A monitoring application, called (Signed) *Applet Watch-Dog*, is proposed to control the execution of malicious Java applets that users can (download and) execute during Web surfing. Typical security attacks that the monitor can stop are related to denial-of-service and antagonism (e.g., obscuring the screen), at the price of a modest degradation of the performance of the Web browser. The Applet Watch-Dog described here is the second release, improving over the original version presented in [GM98]. The application is a signed Java applet, to be executed outside of the sandbox, simple to use and easily configurable by the user, because it works like a user interface. Signed Applet Watch-Dog seems also a necessary tool for software development environments for Java applets.

1 Introduction

Java is the most widely used programming language by Internet programmers. One of the reasons for its success relies upon portability of compiled code (called bytecode) as the basis for mobility of Java code: a program written by any programmer in the world can in principle be loaded and executed locally by any user on his machine. Of course, this poses also serious security problems, as the imported programs could damage, even inadvertently, the resources of user's machine. The main countermeasure adopted by Java is to run the applets in an isolated space, called the *sandbox*, in order to prevent unwanted side-effects, e.g., reading or writing user's files. However, this is not enough. Indeed, one of the most disturbing phenomena that can happen when surfing the Web is to bump into a malicious applet that forces the user to kill the browser or, even worst, to reboot the system. (See, e.g., [MF97] for a panorama of hostile applets.)

In spite of all the efforts made by Sun Microsystems to improve Java security, bad programmers, hackers and the like can still use Java to build dangerous applications. While most security bugs have been fixed as soon as they appeared, the bugs related to denial-of-service and antagonism have been often

*Extended and revised version of [GM98]. Research partially supported by CNR and MURST.

ignored. This is somehow due to the fact that these forms of attack are not, strictly speaking, specific of Java, rather of any system accepting blindly any service request. [1] Hence, one may be induced to think that this problem is not of competence of Java engineers. And this explains why the problem has been underestimated for some time (cf. [G99], page 31).

The literature reports already some possible approaches to reduce security risks in general, and denial-of-service in particular, for Java applets (see, e.g., the survey paper [RG99]). Some of them are *language dependent*, usually obtained by constraining Java syntax and execution capabilities (as many researchers have proposed in the last two years, e.g. [Rie97]); on the contrary, other proposals are *language independent*, taking inspiration from operating systems concepts. Indeed, also our proposal we will present later, falls in this class, as we want to propose something that can be used in future releases of Java, or even adapted for any kind of language for mobile code. We can roughly classify these approaches as follows:

- **Turning off Java**. Of course, this is the most radical "solution"! By disabling applet execution, security is ensured.

- **Code signing**. The idea is that one accepts only signed applets, i.e., applets that come equipped with a signature of a person in a list of trusted persons. The signature links the code to the programmer that created or packaged it. Since this person is trusted, the applet can even be executed outside the sandbox, hence having all of the user's privileges. This solution has one major drawback. Even if we assume that (*i*) the signature is unforgeable, that (*ii*) one signs only code that he has actually produced and that (*iii*) the code is correct (many attacks are due to unintentional mistakes in the code), this approach assumes that a trusted person is, and will always be, so. It is rather easy to produce applets that cause an attack with a lot of delay, hence making hard to tie the attack to its real origin. This approach is also not yet practical, because it poses a lot questions that are to be solved in a widely accepted manner: Who can sign your applet? Is this signing free? Can you sign yourself your applet? Does the browser recognize your signature? And so on. Some of these questions have today possible answers, but still not in a unique way. [2]

- **Code-based filtering**. The code of a downloaded applet is inspected (static analysis) to check the absence of potentially dangerous operationsa (e.g., overriding of the *stop()* method, that can cause innatural survival of applets). In principle, this is nothing but what the bytecode verifier does, and indeed it shares with it the same deficiency: many valid bytecode programs are not verifiable and hence discarded. As, for denial-of-service,

[1] Like the web browser downloads and executes an applet even if it is malicious, in the same way for instance the mail system accepts all messages, even a long sequence of huge messages sent intentionally to saturate a user's mailbox.

[2] For instance, there are some entities, such as VeriSign, that supply a certificate to sign the applet code; most of these certificates are not free; besides, every browser accepts certificates issued only by some entities and so, if someone wants to sign code, (s)he is forced to ask a certicate to those entities recognized by the web browser.

almost all the usually safe operations are potentially harmful, then either almost all applets are discarded or, intentionally, some restictions are abandoned, hence allowing many malicious applets to pass the filter.

- **Address-based filtering**. Instead of inspecting the code, in this approach the origin of the applet is considered. A database is maintained with the URLs that are known to be untrusted. The applets downloaded from such URLs are not executed. Such a solution has been adopted, e.g., in Finjan software [FIN]. Of course, this is a rather poor defense: first, there is no protection against applets from unkown sites; second, it is rather easy to misauthenticate the origin. Another similar approach is based on a firewall in order to assure that applets do not penetrate the internal network, or if they do so, to ensure that they are not executed at the local machine [MRR97].
- **Playground architecture**. This clever technique, preventing most security problems, is based on a change in the architecture: besides the browser and the web, a proxy and a sacrificial playground machine are added [MRR98]. The playground architecture distinguishes Java classes that prescribe graphics actions from those prescribing all other actions; the former are loaded on the client machine, while the latter are loaded on the sacrificial playground machine that uses the graphics server in the browser as an I/O terminal. This solution has, however, some problems. First, if the sacrificial machine is blocked by a denial-of-service attack, this has to be rebooted anyway if the client wants to continue web surfing. Second, it may be unable to prevent graphic operations on the client machine that are related to denial-of-service (e.g., windows opening). Third, the approach is based on bytecode modification (made by the proxy), that is actually not suited in conjunction with the usual approach of code signing.

The approach we are proposing in this paper is different. Instead of inspecting statically the applet code (that has the problems mentioned above), we prefer to monitor its execution at run-time. One may wonder if a dynamic analysis approach can be feasible at all, since it is well-known that in general the problem of determining if an arbitrary program will behave in a certain way is undecidable. However, in our case we are not looking for an exact solution (i.e., determining all and only the applets with the required property), rather for a safe – even if good – approximation (i.e., all, but not only, the applets with that property). Hence, the kind of guarantee we are providing is similar to what is usually given by a static approach (safe approximation only), but we claim that our approximation is quite good and useful, as we will explain later on.

The control over resource consumption can be made, at least, in two ways:

- At the level of the browser.
  This good solution ought to be realized by the browser's suppliers, who have to enrich the functionalities of the browser with this monitoring activity.

- At the level of the user.
  This solution is as valid as the previous one, and can be realized by an application running inside the browser, without modifying its code.

The solution that we propose in this paper works at the user level; in this way the application is quite indipendent from the architecture, quite simple to install and to use.

The basic idea is to monitor Java applets during Web surfing. Applet Watch-Dog (sometimes called Signed AWD, or SAWD for short) is an application that displays some windows containing information about the current status of the applets under execution and of the resources (e.g., memory allocated to the browser) they are using. The user can decide (typically at installation time) the upper-bounds of resources usage (e.g., the percentage of memory allocated to any applet). If the applet under execution exceeds one of these limits, then SAWD immediately kills it. Of course, the monitor is not completely fair w.r.t. applets: there are cases in which non-malicious applets are stopped (e.g., when an applet uses too much of the available memory) and cases when malicious applets are executed unnoticed (e.g., an applet forging new e-mail messages usually does not make any critical use of resources). The former case is not a real problem because the user can always raise the upper bound of the critically used resource to allow the completed execution of the applet. The latter case is more intriguing: even if malicious applets may pass the filter, they are not causing denial-of-service which, after all, is the only security property we intend to guarantee with this application.

For simplicity, the first prototypal implementation of the Applet Watch-Dog, presented in [GM98], was itself a Java applet to be executed in the same execution space of the other applets (the sandbox). Obviously, this made it vulnerable to the attacks of other applets: it is not so difficult to build an applet killer. The second release we are presenting here is signed, so that it runs outside the sandbox with full privileges. This has opened the way to some improvements. First, it is now possible to use system calls tha are strictly necessary to perform some checks on some resources (e.g., CPU time). Second, it cannot be killed by any other unsigned applet. Moreover, the policy of thread's monitoring is changed. The new policy can also stop an applet that creates a copy af the main thread every time the *stop()* method is called on it (see Section 3.4).

The price to pay for monitoring all the applet activities is a modest degradation of the performance of the Web browser (less than 10% of the normal execution time). This is an encouraging result: we can safely use SAWD without fear of performance loss.

As a final remark, we want to stress that the Applet Watch-Dog seems also a necessary tool for any software development environment for Java applets. When executing an applet, it is often the case that the program shows unexpected (and unintended) effects (e.g., it loops forever). If the monitor is integrated in the Java Development Kit [Fla97], then the developer has the capability of stopping/killing the applet without exiting the environment.[3]

[3]The Java Development Kit already contains *appletviewer* to test applets, but it does not

The paper is organised as follows. Section 2 introduces a brief introduction to some security problems raised by malicious and dangerous applets. Sections 3 reports on the application: architecture, implementation, functionality, performance and usage. Finally, in Section 4 we discuss some further enhancements.

2 Building Dangerous Applets

Malicious applets are often present on Internet and everybody can easily find one surfing the Web. Some examples are reported, e.g., on Mark LaDue's Web page (`http://www.math.gatech.edu/~mladue/HostileApplets.html`). In this paper we talk especially about denial-of-service and antagonism attacks: Signed Applet Watch-Dog was intentionally built to prevent them.

- **Denial of Service**
  This attack, making a critical use of resources, can bring a machine to a standstill. The attack can lead the user to kill the browser or reboot the machine (sometimes this is the only way out!). Hackers use some tricks to build such applets. Typically, the applet's priority thread is set at MAX_PRIORITY and the *stop()* method is redefined empty so as to prevent the user from stopping the applet. In order to inspire user's trust, some pretty animation often appears on the screen; in a few cases, the malicious thread sleeps for a while, so the user is led to think that bad actions are not due to the applet.
  The following are the most frequent malicious applets causing denial-of-service that a web-surfer can find:

  - **Filling a file system:** Silently, the applet occupies more and more memory, so that at the end the system quits for lack of memory.
  - **Using all CPU cycles:** The applet prevents any other process to be executed by disabling any other process or by keeping its priority at the highest level.
  - **Windows and windows...:** This denial-of-service attack consists of opening a lot of windows, so to prevent the user from accessing the screen. Indeed, opening thousands of windows in a short time puts a lot of events in the window manager's queue. Practically, this disables mouse and keyboard since they talk to the machine through window events. The only way out for the user is to reboot the system.
  - **Screen obscuration:** The screen becomes suddenly black (or white) and there is no way to access desktop using mouse or keyboard. One example is at `http://www.finjan.com/applet_alert.cfm`.

- **Antagonism**
  This is the most frequent threat a user can find surfing the Web and consists of a non-critical use of resources. It is not dangerous but it is very annoying and may require to restart the browser.

work in presence of communicating applets because it can load only one applet at a time.

- **Sounds:** An applet may play unwanted sound files through a loudspeaker.
- **Images:** An applet may display objectable (or even obscene) pictures on a monitor. In the same class we can include publicity applets, opening windows with commercial animation, that sometimes users prefer not to see.
- **Work for me!** This malicious applet works "in the shadow", i. e. it uses machine's resources to attain its aims (for example, integer numbers factoring, ciphertext decryption, etc.) and transmits the results to his creator. One example is in La Due's web page.
  At first sight, this kind of applet does not seem hostile at all: it does not try to disturb the user or to critically consume machine's resources, so that the user does not realize to be under attack. Nonetheless, they could originate some illegal situations: e.g., imagine a state employee that imprudently loads an applet that uses public resources for private purposes. Unfortunately, it is difficult to detect such a kind of applet because all of its actions are normally permitted by the Java security policy. If resources consumption are too high compared to applet actions, (but it is not easy to valuate this), maybe a malicious applet that works for someone else is running. It would be helpful to see which applets are running: if one of this has not visible effect, the user can reasonably guess that this is an applet of this kind.

Other security threats

People surfing the Web with a Java enabled browser are subject to other classes of security threats too, as listed below. These attacks are, however, outside the scope of SAWD.

- **System Modification** It consists of modifying data on the runtime system so as to allow intrusion into the system itself (for example inducing type confusion). It is a very dangerous attack and it can be implemented by *attack applets* (see [MF97]).
- **Invasion of Privacy** These attacks include detection and disclosure of private information about a user or host machine (for example the file */etc/passwd* in the Unix System). Mail forging can be added to this class too. The standard approach consists of making a telnet operation on port 25. Using Java, the user loads an applet that shows something innocuous on the screen, while, in background, quietly telnets and forges mail. The surprising fact is that the attack is completely within the security bounds of what SMTP and Java are allowed to do.

One may think that, compared to these forms of attack, denial-of-service and atagonism attacks are not really dangerous. Unfortunately, this is not true. It has been estimated [1] that the indirect costs (due, e.g., to the inability of users to access the network) and the direct costs (due, e.g., to working hours of system administrators to reboot the system) caused by internet worms in general (and malicious Java applets are a subset of these) are of several dozens of millions of dollars.

3 Applet Watch-Dog

A possible way to make applets execution safer is to create a monitor to control them. This monitor must be included in the browser because a stand-alone application cannot absolutely control what happens inside a browser; furthermore the application must be configurable by the user.
The first prototype was developed under Windows95 O.S. and built using the Java Development Kit (JDK) version 1.1 and KAWA editor (available at the URL `www.tek-tools.com/kawa`). The prototype was tested on HotJava browser version 1.0β because, at the time it was built (summer'97 [Mar97]), neither Netscape Navigator nor Microsoft Explorer supported Java class files version 1.1.
The new release, Signed AWD, was developed under Solaris O.S. and using the Java Develpment Kit (JDK) version 1.1.6. The applet was tested on HotJava browser version 1.2, hence allowing us to use the *javakey* tool to sign the applet and create a certificate (autumn '98 [Flo98]).

The main purposes of this application are the following:

1. to control all the *threads* running inside the browser (in particular *applet threads*);
2. to change their status, e.g., by changing their priority level or by killing them;
3. to be sure that the memory consumption and CPU consumption does not exceed pre-defined levels and the screen cannot be obscured by malicious applets.

SAWD was not built for security purposes only: it can be also used as a development tool, to help programmers in testing and controlling their Java applets.

3.1 Architecture

Figure 1 shows the structure of the Signed Applet Watch-Dog. The source code for SAWD is divided into the seven files, all contained in a package called *WatchDog*.

The new release has the same structure, but have seven files. The seven files are listed below:

- **AppWatchDog:** This is the principal class, i.e. the application kernel. It contains the *run()* method that continues its cycle until the user exits the applet. The AppWatchDog class implements the *Runnable* interface, allowing threads management and other interfaces such as *ActionListener* and *ItemListener* for new events management.

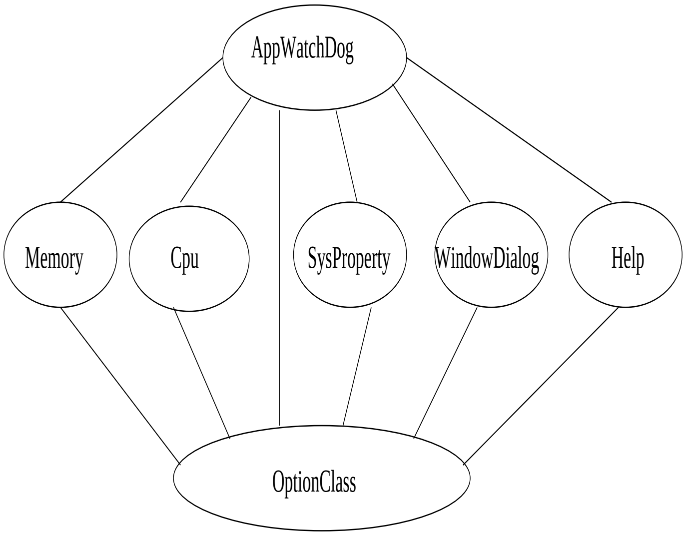

Figure 1: Structure of the Applet Watch-Dog

System Properties	**Meaning**
file.separator	File separator (ex.: "\")
java.file.version	File class version
java.vendor	String that specifies the vendor
java.vendor.url	Vendor URL
java.version	Java version
java.classpath	Java Classpath
line.separator	Line separator
os.arch	Operating System Architecture
os.name	Operating System Name
path.separator	Path separator (ex.: ".")
browser	Browser used
browser.vendor	Browser vendor
browser.version	Browser version

Table 1: System Properties read by SAWD

Moreover, the AppWatchDog class contains some methods used to build the menus and the principal frame; they are called through the *init()* method when the applet starts. Inside this class there are other three classes: BoolAction (managing some user events), NewWindow (building the principal frame), Refresh (used to refresh the principal frame). AppWatchDog class calls methods inside other classes when it needs them (e.g. setting refresh time or thread's priority value). The thread list is updated every 200 ms.[4] The *stop()* method is defined empty so to continue the execution even when the user changes web page.

- **Memory:** This class monitors the memory consumption, showing it in a colored graphic pie (total memory, free memory and used memory). The class implements the *Runnable* interface, so it manages the thread of control in an independent way. To clean up unused memory, we have inserted a call to the *gc()* method that starts the Garbage Collector.
- **CPU:** This class monitors the CPU consumption, showing it in a colored graphic pie (used CPU). The class implements the *Runneable* interface, so it manages the thread of control in an independent way. To get the browser CPU consumption, we call the *top()* program, using a native method.
- **WindowDialog:** This class manages dialog windows for user inputs.
- **SysProperties:** It shows the system properties, such as those reported in Table 1. For security reasons not all system properties can be read (it could help some programmers to build attack applets) [CW96].
- **Help**: It contains the Help Menu showing explanations for all items in the MenuBar.
- **OptionClass:** Since the principal application thread continues its execution while moving from a menu item to another one, the problem about how to communicate between classes is solved by OptionClass, that contains user choices that are available from every point inside the program.

3.2 Usage

The Signed Applet Watch-Dog can be loaded including the following HTML instruction in a Web page:

```
<applet code=AppWatchDog.class archive="WatchDog.jar"
width=... height=...>
</applet>
```

or it can be loaded at browser start-up, setting in a suitable way the *start-up page*. When loaded, SAWD begins to monitor all threads present in the browser, showing them in a column list where the user can see the name, the priority and the type of each thread. Each element of this list can be selected by simple click (for changing its priority or for killing it) or double click (for showing the

[4] We consider this value a good compromise between CPU consumption and data refresh.

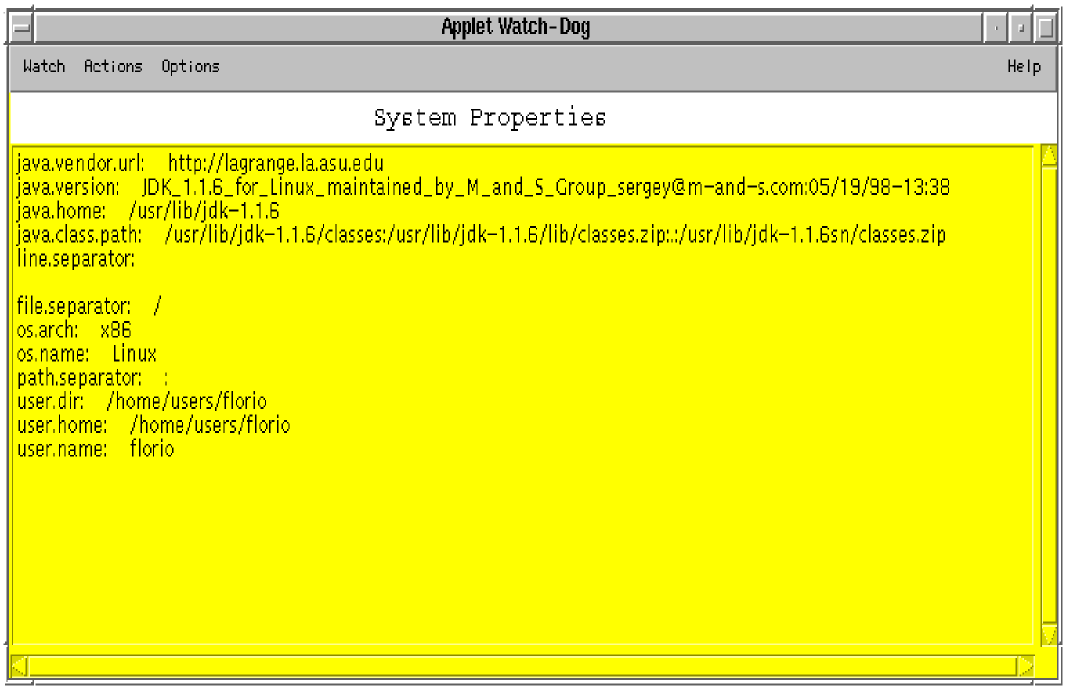

Figure 2: Properties Window of the Applet Watch-Dog

elements of a group). The application is always active, even if the user changes the page or continues web surfing.
The Applet Watch-Dog appears in a separated frame, so the user has a "console" from which he can directly see and control all the other applets running in the browser. It is something like the "Java console" under Netscape Navigator, but with different purposes: indeed the "Java console" only shows the Java interpreter execution and has no way to control applets execution.

Menus

There are four items in the MenuBar: **Watch, Actions, Options** and **Help**. Let us see them in detail.

- **Watch:** It is the principal menu and it is used to observe applets execution. It contains six submenus as follows:
  - **Threads:** Shows all the threads in the frame. It is the starting item.
  - **Applet threads only:** Shows only the applet threads.
  - **Memory Resources:** Shows the available memory and the used memory in a colored graphic pie, w.r.t. the total memory allocated to the browser. When the user chooses this item, he can then activate the Memory Usage Options (see below) to control applets consuming a quantity of memory exceeding a predetermined value.

- **CPU Usage:** When the user chooses this item, he can activate the CPU Usage Option (see below) to control applets consuming a quantity of CPU exceeding a predetermined value. These quantities are computed w.r.t. the CPU time allocated to the browser. This functionality, already planned in the first release, has been implemented in the new release by including native code.
- **System Properties:** Shows the system properties. An example in Figure 2.
- **Exit:** Exits the program.

- **Actions:** The Actions menu allows the user to kill a thread or change its priority. Using the **Kill Applet Threads** submenu, a user can kill the selected thread. Using the **Change Priority Settings** submenu, a user changes the thread's priority. (Recall that the new value cannot be higher than that of the owner ThreadGroup, see [CW96].)

- **Options:** This menu allows the user to configure the Applet Watch-Dog. These options are stored in a file outside the sandbox.[5]
  - **Memory Usage Options** This option causes the halting of all applets belonging to the ThreadGroup of the applet that causes the overflow of the fixed limit. [6]
  - **CPU usage Options** Shows a window dialog with four *checkboxes* for a limit on the CPU usage. This option window is available and the corresponding functionality is now implemented. Similarly to the above, all applets in the TreadGroup of the applet that exceeds the fixed limit are halted.
  - **Priority Options** Choosing this item, the user can see a window dialog that allows him to stop every applet having a determined prioriy. For example, stop all applets having MAX_PRIORITY set.
  - **Refreshing Time** To combat the applets that obscure the screen, it is possible to refresh the main window. As a default value, the refreshing time is set at 0 minutes (no refresh), but the user can change this value as (s)he wants. In this way the user can then manually kill the applet by selecting it from the Actions menu.

- **Help:** This menu helps the user in using the application.
  - **Contents:** Shows a description of the commands.
  - **About:** Informations about authors and version.

[5] In the first release, instead, as no file could be persistent, the options were valid for one session only.

[6] Instead of halting all the applets in the ThreadGroup, one could think of suspending them, then resuming those that (the user thinks) are not dangerous. This improvement is left for a future release.

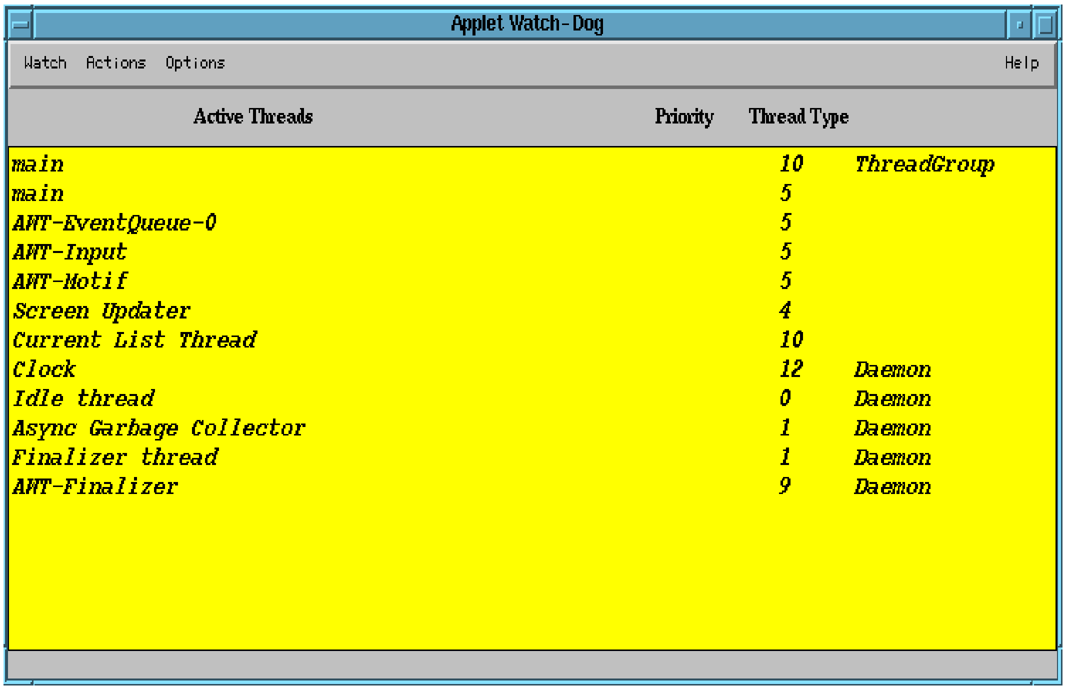

Figure 3: Applet Thread Window

3.3 What you can do and what you cannot

Applet Watch-Dog allows users to contrast malicious applets running in their browsers. In Table 2 we can see which kinds of bad applets it can stop.

An applet may be automatically killed by SAWD when, e.g., it uses of a resource more than the predetermined (by the user) amount. In this class, we can put the first applet in the table (memory allocation). Many of the other applets listed above are automatically killed if their priority level is MAX_PRIORITY, as in the case of windows creation. However, there are other cases in which the user has to intervene manually to kill the applet from the suitable menu. This is the case, for instance, of screen obscuration or of a work-for-me applet. Es-

Malicious Applet Type	**Stopped by Applet Watch-Dog**
Memory allocation	YES
Windows creation	YES
Play sounds	YES
Display obscene pictures	YES
Work for me!	YES
Screen obscuration	YES
Mail forging	NO

Table 2: Malicious applet stopped by SAWD

pecially delicate is the latter case, as it is not always immediate to realise when a work-for-me applet is active.
Summing up, Applet Watch-Dog stops all the applets causing denial-of-service on the monitored resources and most of those causing antagonism. Unfortunately, it does not stop applets causing privacy attacks. One can think of introducing specific checks for coping with specific problem. E.g. for mail forging, we thought to monitor port 25 waiting for incoming connections, to find if some applet tried to illegally send mail.[7] Nonetheless, in general we cannot reasonably expect to be able to prevent all of these problems.

3.4 Immortal Applets

In this section we focus on a class of very dangerous applets, that we have called *immortal* applets. Such applets have all the same structure: there is an occurrence of a block *try/catch*, followed by the clause *finally*, where a copy af the main thread is created. Before the availability of SAWD, when a user loads an applet of this kind, the only way to stop it is to kill the browser.

Indeed, in order to block a dangerous applet, SAWD calls the *stop()* method on the applet's ThreadGroup. The Java runtime, after the call of the method, raises a ThreadDeath Error, that causes the death of the thread. This solution effectively halts all applets, except the immortal applets. Such an applet catches the TreadDeath Error in the *finally*, raises an *IllegalThreadStateException*, creates a copy of it and runs the new thread. In this way, the applet can perform its malicious action.

Our solution to stop immortal applets works as folows. Every time SAWD detects an IllegalThreadStateException, it calls the *suspend()* method on the main thread. This solution effectively halts the immortal applet, even if it does not destroy the thread. Finally, we present an example of immortal applet, that SAWD has halted.

```
import java.applet.*;
import java.awt.*;
import java.io.*;
public class SimpleApplet extends java.applet.Applet implements Runnable
{
 Thread riborn; boolean ok;
 public void init()
 {
  riborn=null;
 }
 public void start()
 {
```

[7] This intention was dropped because of new security policy introduced by Sun ([Fla97]): an untrusted applet cannot make a telnet in a *well-known* port. If all other companies will follow this policy, the mail forging problem will be no longer present (or, at least, it will be very difficult, if not impossible, to build such applets). Moreover, this problem is only present if not using POP Mail Clients.

```
  if (riborn==null)
  {
   riborn= new Thread(this, "riborn");
   riborn.setPriority(Thread.MAX_PRIORITY);
   riborn.start();
  }
 }
 public void paint(Graphics Gr)
 {
  if (true)
  {
   Gr.setFont(new Font("Helvetica", Font.BOLD, 24));
   Gr.drawString("I'm alive", 30,50);
   ok=false;
  }
  else
  {
   Gr.setFont(new Font("Helvetica", Font.BOLD,24));
   Gr.drawString("You can't kill me");
   ok=true;
  }
 }
 public void stop() {}                  % do nothing
 public void run()
 {
  try
  {
   while (true)
   {
   try
   {
    repaint();
   }
   catch (Throwable e) {}
   }
  }
  catch (ThreadDeath td){}             % catch ThreadDeath Exception
  finally
  {
   SimpleApplet sa=new SimpleApplet();   % create a copy of main thread
   Thread reborn=new Thread(sa,"reborn");
  }
 }
}
```

3.5 Performance

Using the top program under Solaris OS, we have tested the CPU consumption when the Applet Watch-Dog runs. During normal background execution, the CPU consumption increases of 3-4 %. Of course, this value is different when SAWD is used by the user to kill or change priority or set options, that is when it does not run in background.
This application, however, introduces a small delay in loading other applets, a delay that can be evaluated in less than 10% of total loading time. Considering the number of applets that a user normally loads during his Web surfing, Signed Applet Watch-Dog can be used without relevant performance loss.

4 Conclusion and further enhancements

The Applet Watch-Dog is a useful tool to control every applet a user can download. Depending on the degree of resource consumption, some applets are automatically killed by SAWD, others can be killed by human intervention.

The monitor that we have presented to stop the malicious applets can still show the following problems:

- Possible sensitivity to the browser: every browser has a different security policy implemented through the Security Manager.
- Inability to control signed applets running outside the sandbox.

Signed Applet Watch-Dog could behave differently when executed in different browsers. Since every browser has a different security policy implemented through the Security Manager, it is not a surprise to realize that SAWD is sentitive to the browser. The solution for this problem can be to integrate the monitor within the Web browser (i.e., working at the browser level). In this way, the browser's functionality would be enhanced with the features of the Signed Applet Watch-Dog, quite an appealing improvement for next generation browsers!

Even if the scope of Signed Applet Watch-Dog is restricted to unsigned applets only, we think that it can be considered a useful tool anyway, at least for the following two reasons:

1. It is unreasonable to think that all applets are (or will be) signed; at least because it is still difficult and expensive to sign code.
2. It is unrealistic to load only signed applets, as most applets in the Web are (and will probably remain) unsigned; furthermore, when a browser does not recognize the signature of an applet, it will be executed as an untrusted applet.

Finally, future work will be devoted to adapt SAWD to the new JDK 1.2 [G99] which allows for greater flexibility in handling Java applets.

Acknowledgements

The authors would like to thank Mark LaDue, Li Gong and Andreas Sterbenz for their help and suggestions.

References

[CW96] Mary Campione and Keith Walrath. The java tutorial, September 1996. Http://ftp.javasoft.com/docs/tutorial.ps.tar.Z.

[FIN] *http://www.finjan.com*

[Fla97] David Flanagan. *Java in a Nutshell.* O'Reilly & Associates Inc., second edition, 1997.

[Flo98] M. Florio. Signed Applet Watch-Dog Master thesis, University of Bologna, December 1998.

[GM98] R.Gorrieri and G.Marchetti. Applet Watch-Dog: a monitor controlling the execution of Java applets. in Procs. of *Fourteenth IFIP International Information Security Conference* (SEC'98), (G. Papp and R.Posch, Eds.), Chapman& Hall, September 1998.

[G99] Li Gong. *Inside Java 2 Platform Security.* Addison-Wesley, 1999.

[GA96] James Gosling and Keith Arnold. *The Java Programming Language.* Addison-Wesley, 1996.

[Mar97] G.Marchetti. Applet Watch-Dog: un monitor per controllare l'esecuzione di applet in Java. Master thesis, University of Bologna, October 1997.

[MRR97] D.Martin, S.Rajagopalan, A.D.Rubin. Blocking Java Applets at the Firewall, *Procs. Internet Society Symp. on Network and Distributed System Security*, pages 123-133, 1997.

[MRR98] D.Malkhi, M.K.Reiter, A.D.Rubin. Secure Execution of Java Applets Using a Remote Playground, *Procs. IEEE Computer Society Symp. on Security and Privacy*, pages 40-51, 1998.

[MF97] Gary McGraw and Edward W.Felten. *Java Security. Hostile Applets, Holes, and Antidotes.* Wiley Computer Publishing, 1997.

[Nau96] Patrick Naughton. *The Java Handbook.* Mac Graw-Hill, 1996.

[1] [NPS] Naval Postgraduate School. *Introduction to Computer Security.* Notes for the Course CISR CS3600.

[Rie97] Thomas Riechmann. Secure java: A security model based on transitive capabilities for distribueted java. Technical report, University of Erlangen-Nurnberg, February 1997.

[RG99] A.D.Rubin, D.E.Geer. Mobile Code Security. IEEE Internet Computing, 2(6), pages 30-34, 1998.

[Ste96] Andreas Sterbenz. An evaluation of the java security model. Technical report, Graz University Of Technology, Austria, March 1996. Http://www.iaik.tu-graz.ac.at/PEOPLE/sterbenz.html.

[Sun1] Sun Microsystems Inc. Javasoft forum 1.1, September 1996. Http://java.sun.com/forum/securityForum.html.

[Sun2] Sun Microsystems Inc. Frequently asked questions - applet security. Technical report, Sun Microsystems, February 1997. Http://www.javasoft.com/sfaq.