CONTRASTING MALICIOUS APPLETS BY MODIFYING THE JAVA VIRTUAL MACHINE

Vincenzo Ciaschini[1] and Roberto Gorrieri[2]

[1] *INFN CNAF*
Viale Berti Pichat 6/2, 40127 Bologna, Italy
vincenzo.ciaschini@cnaf.infn.it

[2] *Dipartimento di Scienze dell'Informazione*
Mura A. Zamboni 7, 40127 Bologna, Italy
gorrieri@cs.unibo.it

Abstract Java is the most popular language for web programming. However it suffers from some well-known denial-of-service attacks (e.g., obscuring the screen) due to the execution of malicious code that uses resources in an improper way. In this paper we present a new approach to alleviate these problems by patching the Java Virtual Machine, in order to force the needed checks on resources usage bounds directly at the level of the source code.

Keywords: Denial of Service, Mobility

1. INTRODUCTION

The Java[2] language, developed by Sun Microsystems, is nowadays an accepted standard in Web programming. Many applications are developed using it, and there are even more Java applets included in web pages from millions of sites. This is certainly due to its many desirable features like code mobility (supported by machine independancy at the bytecode level), simplicity of programming (due to similarities with C and C++), efficient execution, use of a virtual machine, and so on. Last but not least, Java is certainly a language for *secure* Web programming. As a matter of fact, Sun addressed from the very beginning and with great care the problem of stopping all potential system-breaking attacks due to code mobility, and it did so by adopting the *sandbox* [9, 4] approach. It consists of enclosing the applet in an isolated space

Y. Deswarte et al. (eds.), *Security and Protection in Information Processing Systems*

from which it cannot escape unless the user gave it some special permissions. In this way, many limits are imposed on it and, assuming there is no bug in the implementation of the sandbox, most system attacks are effectively made impossible.

Unfortunately, Java still suffers from some security problems. Of course, not everything can be prohibited if we want to have a useful system, and this leaves the way open for downloaded malicious applets to start attacks more subtle but no less important, such as denial-of-service, antagonism and invasion of privacy. It is worth noting that these kinds of attack are simple to implement, generally ranging from 10 to 30 lines of code, and with no conceptual complexity in them, thus becoming a tempting instrument for malicious users.

In the next section, we survey the many approaches that have appeared in the recent literature to contrast malicious applets. This survey will end with the motivations for our current proposal, i.e., a monitor, called *Limiter* and implemented as a patch [1] for the Java Virtual Machine (JVM, for short)[4], that performs many checks on the Java source code before actually executing it. Section 3 describes Limiter; in particular, the general criteria that have guided its definition and the many different sources of attack that have been contrasted. Section 4 reports about the effectiveness of the approach and discusses performance issues. Finally, some remarks on possible improvements for future work conclude the paper.

2. ATTACKS AND DEFENSES

It is widely accepted (see, e.g., [6]) that there are four major categories of attacks:

1 *Attack applets.* They try to modify the system (e.g., the local filesystem) to allow, or at least facilitate, a successive intrusion. This kind of applet is the most dangerous one, as the whole system can be compromised and possibly destroyed. Sun has focused its efforts mainly on this class, and so the available defenses are strong.

2 *Privacy Invasion.* There are applets that try to gather information from the user or to impersonate him (for example by sending e-mails in his place). Some attacks in this class are easily implementable. The consequences of these attacks are usually moderate, but occasionally they may be very nasty (e.g., password cracking attack); the countermeasures Java offers in this respect are usually sufficient.

3 *Denial of Service.* These are applets that try to deny the legitimate user to access some services or resources that he could normally obtain from his local machine. For instance, allocating all of a system's memory, obscuring the screen, locking the keyboard and mouse, deny access to the internet, and so on. Implementing such attacks is usually not difficult, but contrasting them is hard and Java does not offer adequate countermeasures.

4 *Antagonism.* There are applets that antagonize or annoy the user in some way. Three examples are: continuously playing a sound, bringing up popup windows at unwanted moments, or even working in background to do something without the awareness of the user. At first sight, this last example of antagonism does not seem hostile at all: such an applet does not disturb the user or critically consume machine's resources, so that the user does not realize to be under attack; indeed, it is difficult to detect such applets.

Sun actively researched to stop only the first two classes of attacks, and with very good results. It is also important to cope with the other classes, because of the costs due to repair time and time offline. However, it must be said that the reason why Sun has not coped with the latter is that those classes of problems are not Java-specific, but on the contrary are common to all network applications.

While the problems presented by the latter two classes may seem trivial if they manifest themselves on a normal user workstation (a system reboot is generally enough to take care of them), they become of much greater gravity if the machine attacked is a server, because in this case a reboot may cause much greater problems: loss of data, network downtime, and so on. For more details on hostile applets, see[5–7].

2.1 Defensive Approaches

Many solutions have been proposed to deal with the problem, and here is a short review of the most representative ones.

1 *Disable Java.* There is no doubt that disabling Java would solve all these problems. However, it is a much too drastic decision. As a matter of fact, this way would also lock yourself out of all the advantages that the Java language has, such has a high portability of *compiled* code, and it can easily be said to do more harm than good. However, it should be noted these problems have in the past even pushed CERT to recommend disabling Java. An implementation of this solution, namely blocking Java directly at the firewall, has been tested in [10].

2 *Code Signing.* This is the solution implemented by the Java developers from Sun. Signing a piece of bytecode certifies its author, and on the base of this knowledge, extra rights and possibilities (like read or write access to the file system) may be given or denied to the bytecode itself. The problem with code signing is that it gives absolutely no guarantees on the behaviour of the signed code. It only offers a way to trace the code back to its author. Even this is not certain. In fact, if you need to sign some code, you need a certificate. There are essentially two ways to obtain one: you can make one yourself, or you can request one to a certification authority. In the first case it can be easily falsified, thus removing all the (limited) usefulness of code signing, while in the second case you have to pay a hefty sum, thus putting it out of reach for the non-profit programmers. Also there are certainly ways to fool the company by impersonating someone else. And there is always the possibility that the user's browser will not recognize your signature anyway.

3 *Code Inspecting.* This approach consists of examining the bytecode of every Java class before it is run, looking for suspicious contracts, and then deciding whether to allow execution or not. An example of this approach is Finjan's SurfinGate program [11], which attempts to discriminate code when it reaches the machine. The problem with this technique is that it permits only to screen for *known* attacks, and is generally useless against techniques like code obfuscation. Also, many successful attack techniques are very difficult to distinguish from normal code. For example, there are known attacks that consist simply of a sequence of type casting operations.

4 *Language Modification.* Many problems stem directly from some of the capabilities of the language, so it would seem natural to simply modify Java to remove or modify such capabilities. Examples of this are given in [6]. However, it is not a simple task. What is really executed is not the Java language, but its compiled version, called *bytecode.* The association between Java and bytecode is not particularly strong. For example it is possible to compile C code into bytecode. As a consequence, simply modifying Java would net you nothing, because the bytecode can be generated in many other ways. On the other hand, modifying the bytecode (for example by replacing it with abstract syntax trees, as proposed in [6] can be a much more rewarding task, but the price would be to remove compatibility with all the previous versions. Not a good thing

for a language that makes compatibility one of his most alluring features.

5 *Codebase filtering.* This strategy consists essentially of filtering out applets basing such a decision on the applet's codebase, and has been proposed and implemented in Princeton's JavaFilter[14]. The obvious problem is that it is rather easy to fool the controlling software into believing that the applet has a different origin than the real one.

6 *Sacrificial machine.* The basic idea of this approach is to setup a sacrificial machine with only minimum connections to the rest of the net, and execute all Java code on it [13]. While this approach can certainly be used to put a limit on the damage a malicious applet can wreak, it also foregoes the ability to use the Java language for distributed computing, and it feels like a step back. An implementation of this approach has been presented by Digitivity (now part of Citrix) [12].

7 *Monitors.* Finally, this approach consists of monitoring the applet behaviour to intercept and then to prohibit dangerous actions, obtaining the ability to stop even previously unknown attacks simply based on their effects. They can be implemented at various levels:

user level – At this level the monitor is nothing but another applet, obviously running *outside* of the sandbox, that runs all of the checks. The problem with this approach is that the applet can be interfered with by another misbehaving applet, and so you cannot even be sure that it is running. On the other hand, this approach is simple to implement and can work everywhere without any modification to existing systems. An example of this approach is [8].

browser level – At this level it is the browser itself that runs all the checks. The problem with this approach is that of a lack of uniformity, because every browser is independent of each other, and so this monitor should be reimplemented from scratch every time.

system level – At this level it is the JVM itself that runs the checks when the applet is running. An obvious advantage of this choice is that it is not possible to avoid the checks.

An obvious drawback to all these three approaches is that the added checks increase the execution time of each and every applet, whether they are malicious or not.

Other problems are shared by all these seven approaches. For example, it is possible to write applets capable of resurrecting themselves in case of death, or that simply refuse to die, and none of them can put a stop to it. We will approach again this issue in the conclusions.

2.2 Our Approach

The approach proposed in this paper is that of a patch to the JVM implementing a system-level monitor. The reason that led to this particular strategy is that by modifying the source code of the virtual machine, many weaknesses of all other defensive approaches can be circumvented.

For example, all the problems mentioned above can now be simply prevented by way of a few additional checks in the JVM, and the fact that they are implemented directly in the JVM itself serves to lessen their impact on performance.

There are other obvious advantages of this approach:

- It becomes impossible for applets to avoid the new checks the monitor implements, since they are done by the JVM on all running applets regardless of other factors, and this is a dramatic improvement on systems like [8], where the monitoring is done by an applet that can be shutdown or interfered with by other applets.
- This approach does not involve itself with an analysis of the code that it does execute, but instead simply takes step to limit or make impossible dangerous behaviour, with the result of catching also attacks that were unknown by the authors of the code.
- Finally, by directly modifying the JVM we sidestep the need to create patches for all existent browsers, since most of them allow the user to choose a specific JVM by using a plug-in system, like for example Mozilla does.

In light of all these factors, in the rest of this paper we describe the monitor, called Limiter, that we implemented.

3. THE MONITOR IMPLEMENTED IN THE JVM

3.1 General Criteria

Limiter was programmed with a few basic rules in mind:

1 *No changes to the class library.*

This rule (and partly the following one) is a direct consequence of the license under which the JVM source code is released. In

fact, this license prohibits all changes to standard class library. Furthermore, the modified JVM, as configured out-of-the-box must be indistinguishable from the standard one.

2 *Conservative extensions*

It should be possible to configure the new VM in such a way that no difference can be found between it and the standard one. As the previous one, this rule is a consequence of the JVM source license.

3 *Changes to the semantic of the language should be limited to the minimum necessary.*

This rule is a consequence of the need to keep compatibility as high as possible between the modified version of the JVM and the original one, to keep the existence of the monitor as transparent as possible to all existing (non malicious) applets, and to avoid breaking them unless absolutely necessary.

4 *Easily identifiable modifications*

Every change to the original source code should be clearly highlighted as such in the source itself. This last rule is here as a way to distinguish original code from the new one, and is also required from the JVM source code license.

These rules are evidently straight-forward and are derived primarily from the JVM source license. Of particular interest are rules 2 and 3. They together mean that, while it was necessary to slightly alter the semantics of the language to implement checks on some operations, if those checks are disabled the semantics of the operations revert to the original ones. As a special note, those changes consist of adding the possibility of failure to operations that could not normally do so. This means that existing applets execute in exactly the same way in both the the original and modified virtual machines, except for the case in which the operations they attempt are going against the limits imposed by the user of the modified VM, and are in this case stopped. Exactly what should happen.

A few general observation are valid for all the changes that have been done to the system:

1 Applets are classified respecting to their codebase (i.e. the location from which they have been downloaded) and not their actual code (or hash function). This because this second choice is too strong. It can easily distinguish applets when there is no real reason to do so; for example, the presence of an additional unused variable is reason enough to distinguish them.

2 Also, no distinction is made among applets with the same codebase. This has been done because there is no use in putting restrictions on an applet when those same restrictions can easily be circumvented by having two or more instances of the same applet work together. For example, if a limit of 2M is put on memory usage per applet, to use 20M it would be sufficient to use 10 different instances of the same applet and having them work together.

3.2 Which problems does Limiter Contrast?

Here is a list, categorized by type and gravity, of the areas that have shown troublesome and that were checked in some way.

- **Denial of Service, grave.** This category includes those denial of service attacks that can potentially crash the system.
  - **Thread Explosion** It is possible for an applet to create an enormous number of threads, filling the process table and crashing the system. This happens because Java threads relies on the system ones in many implementations, most notably the one considered here.

    The solution chosen was to put a limit on number of threads in execution at the same time for each applet. In case an applet attempts to create more threads than it is allowed to, the operation fails as if for lack of memory, since this was the only case in which this operation could originally fail.
  - **Frame Explosion** Also, an applet can create a great number of frames in a very short time, thus overloading the event manager and effectively locking the user out of the keyboard and the mouse.

    There is also a limit on the maximum number of frames opened. In case an applet tries to created a bigger number of frames than it is allowed to, the operation fails with a `NullPointerException`, the only way the function could normally fail. It is to be noted that this limit applies not to a single applet (like all the others), but to all the applets at the same time, e.g. it is a limit on the *total* number of frames. This because:

    1 It is very difficult to deduce from a frame which applet is trying to create it, and
    2 The cause of problems is the number of frames opened, it does not matter what applet opened them.

- **Memory Exhaustion** An applet can also make unlimited use of available memory, possibly filling it and crashing the system.

  There is now a limit on the total amount of memory that a single applet can use. It is said *total* memory, because the limit applies to the whole amount of memory that an applet uses during its lifetime, whether it has been released it or not. This is because aside from calling `System.gc()` (a thing rarely done by an applet) there is no way for the applet to effectively release memory, or even to know that some memory has been released. In case memory is refused, the exception handling is the same that would occur if the requested memory were truly unavailable.

- **Denial of Service, Weak.** This category includes those denial of service attacks that are not strong enough to crash the whole system.

  - **Priority Juggling** As an applet can modify its own priority level without any limit, this means that the applet can be put it to `MAX_PRIORITY`, with the result that all other applets, that run at normal priority, are effectively almost never executed, bringing about an unacceptable degradation of the system.

    The solution chosen was to put a limit, configurable by the user, on the maximum priority of the threads of an applet, preventing it from stealing CPU cycles from other applets. In case an applet attempts to set a higher priority than it is allowed to, then the priority is silently lowered to the maximum acceptable value.

  - **Frame Resizing** An applet can create frames of unlimited size. This could be abused by creating a frame larger than the screen, and positioning it in such a way to completely cover it, making impossible for the user to see what is happening under it. It should be noted that this fact can also be abused by moving it in such a way that the borders (and the notice that it was really an applet who created it) are off-screen, and this would lead to the possibility of having an applet frame masquerading itself as a system frame, a typical invasion of privacy attack.

    A limit has been put on the maximum size of a frame. In case an applet attempts to create a bigger frame than it is allowed

to or to resize it beyond acceptable values, the offending dimension is silently lowered to maximum acceptable value. It is to be noted that this limit is in effect only if the applet itself is trying to break it. The user can enlarge a frame without any limit.

- **Thread Brokering** At last, it is possible for a thread to interfere with the execution of other threads, modifying their priority, calling `suspend()` or `resume()`, etc...

  As a consequence, now an applet can only execute these operations on its own threads. Trying to call these functions on threads belonging to another applet has no results.

- **Antagonism.** This category includes all those attacks that are meant to simply annoy the user.

  - **Juke-box** The ability to play sounds can be misused. In fact it is sufficient to continuously play an annoying sound to create problems to users.

    The solution chosen, configurable by the user, was to prohibit an applet from using sounds. This is done by fooling the system into believing that the sound device is perpetually occupied. This does have the unfortunate side effect of forbidding the use of sounds for the whole applet.

- **Invasion of Privacy.** This category includes those attacks that are meant to gather information from a user, or to masquerade as one, or to have him working from oneself.

  - **Mail Forging.** A dangerous characteristic of applets is that they can make internet connections. It is true that these connections can only be made to the originating host, but this is not sufficient to prevent attacks like mail forging, and is also a necessary part of work-for-me attacks.

    As a consequence, it is now possible to prevent *all* socket access for an applet, *except* for the accesses made by a classloader. This makes impossible mail forging attacks and prevents work-for-me applets from returning the computed results. In case access is refused, the JVM is fooled into thinking that the `socket()` call returned an `EACCES` error, and the situation is treated accordingly to the already present code. The exception to this ban is necessary because an applet must in any case be allowed to completely load itself.

- **Miscellaneous.** This category include all those problems that, while not usually attacks by themselves, are commonly used as tools, aids or building blocks in creating them.
  - **Class loading** While applets cannot normally load external libraries, this ability is present in the JVM. Since external libraries cannot obviously be subjected to any check from the JVM, and the only thing that prevents an applet from using them is the security manager (that has been notoriously subject to bugs and attacks) this is an extremely dangerous features.

    The chosen solution chosen was to make it possible to prevent the loading of external libraries. In theory, this is the default behaviour and is enforced by the security manager, however in practice the security manager has often been circumvented, and this is the reason of the ban. In case the permission to load a library is refused, it acts as if it was not found. However, it is necessary for some libraries to be loaded for Java to work properly, and so libraries from the system paths can always be loaded regardless of this setting.
  - **Immortal Applets, Part 1.** One of the most consistently misused features of the Java Virtual Machine is the ability to catch the `ThreadDeath` exceptions. Because this is the exception raised when an applet is about to be terminated, catching it means preventing the death of the applet itself. Immortal applets are thus reborn.

    To prevent this, it is now possible to prohibit the catching of `ThreadDeath`. If the permission to catch this exception is refused, the system simply does not scan the class to look for `catch` or `finally` clauses, simply ignoring them. It is to be noted that system classes sometimes need to catch this exception, and so this check does not apply if the catching class is a system one.
  - **Immortal Applets, Part 2.** A similar problem can be presented by putting appropriate code in the `finalize()` method of the class. Since this method is called by the finalizer after the thread has been stopped but before memory is released, it could be used to resurrect a new thread at a certain moment after its end, when the user has already forgotten it.

    To solve this problem the finalizer thread, responsible of the resurrecting applets, has been subjected to the same limits

as the applets, and it can so be configured to prevent the creation of new threads or the use of many resources, just as for applets, effectively putting an end to resurrecting applets.

It should be noted that we have been as conservative as possible: whenever an operation exceeds one of the configured limits, it is made to fail according to one of the already handled possibilities. For instance, when an applet tries to allocate memory above the configured limit, the operation fails as if there was an out of memory situation. Hence the "programmed" failures of Limiter occur in such a way that the class libraries are already prepared to handle. This has been done to limit the changes on the semantics of the JVM itself.

3.3 The Structure of the Original JVM

The version of the JVM we have modified is 1.2.2 for Linux. The source code itself is divided into three subdirectories, `src/`, `ext/` and `build/`. `build/` contains the makefiles and all the other files necessary to build the code, `ext/` contains the source for extensions like the support for internationalization; the actual source is inside the `src/` directory. This directory is also divided into two subdirectories, `share/` and `linux/`. The former includes all machine-independent code, while the latter has all linux-specific code. In fact, every component of the JVM is divided into two parts. The portable one into the `share/` directory and its subdirectories, the non portable one into `linux/` and its subdirectories. This also means that the two directories have (roughly) the same subdirectory structure.

The source of the JVM itself is written in C, C++ and Java. Java is used primarily in the system class library, while the JVM itself is divided into many subsystems (audio, graphics, threads, core, etc...) programmed in C and C++. Each one of these subsytems has its own subdirectory, either in the directory itself or in `native/`. The core of the JVM itself is included in the `javavm/` subdirectory. It is further divided into `export/`, `include/` and `runtime/`. `export/` includes the files available to all the JVM, `include/` includes files available only to this particular subsystem, and `runtime/` contains the source code itself. In these three directories we have placed the bulk of the modified (and all the newly added) files. A few other files have been modified into the `src/share/native/sun/audio/`, `src/share/native/sun/awt`, `src/linux/hpi/green_threads/src`, `src/linux/hpi/native_threads/src` and a few other subdirectories.

3.4 Implementation

Let us see, in short detail, exactly how Limiter has been implemented.

The first thing needed is a structure capable of holding the limits for a specific applet and the resources currently in use. This is the duty of the following structure, taken from `src/share/javavm/include/limiter.h`.

```
struct AppletInfo
{
    long int  memLimit;
    long int  memCurr;

    long int  thrLimit;
    long int  thrCurr;

    long int  priLimit;

    long int  frmX,frmY;

    long int  Flags;

    /* A single AppletInfo structure can be referenced in more than
     * one place. This field takes account of the various
     * references.
     */
    volatile int uses;

    int code;

    /* Thread specific info starts here... */
    int *ignorememm;
    int *memused;
};
```

The meaning of its fields should be easy to understand, with the possible exception of the fields `Flags`, and the last three. The first one is in effect a flip-flop set for limits that do not have an associated parameter, but are in effect a simple on/off decision, while the last three are used for internal safe-keeping and are not really relevant now.

This structure is normally memorized into an hash table in pairs (thread, structure). The algorithms used are defined into `src/share/javavm/runtime/hash.c`, and are pretty standard ones. The only thing that should be noted for further reference is that the access function is called `FindHash()` and its parameter is the thread of which we want to find the respective structure.

This structure can be accessed concurrently by many threads, and so a way is needed to assure its consistency. This is done by defining a monitor and requiring the code to enter it before accessing the structure. This is its implementation.

From `src/share/javavm/include/limiter.h`:

```
extern sys_mon_t *_ai_lock;
#define AI_LOCK_INIT() monitorRegister(_ai_lock, "AppletInfo_lock");
```

```
#define AILOCK(self)    sysMonitorEnter(self, _ai_lock)
#define AILOCKED(self)  sysMonitorEntered(self, _ai_lock)
#define AIUNLOCK(self)  sysMonitorExit(self, _ai_lock)
```

and from **src/share/javavm/runtime/interpreter.c**:

```
/* CV */
    _ai_lock=(sys_mon_t *) sysMalloc(sysMonitorSizeof());
    if (_ai_lock == NULL) return FALSE;
    AI_LOCK_INIT();
/* CV */
```

The **/* CV */** comments are used to delimit the changes we have made in the original code.

These two fragments, together, define and initialize a monitor whose job is to regulate access to the AppletInfo structures, and to do so use the standard JVM mechanism.

Then, a particular set of files is modified for each check we added to the JVM. For example, the checks to prohibit an applet to interfere with the execution of another are implemented by modifying the **JVM_SuspendThread()**, **JVM_ResumeThread()**, **JVM_SetThreadPriority()** and **JVM_StopThread()** functions in **src/share/javavm/runtime/jvm.c** by having them call a new **limallowthread()** function in the new file **/src/share/javavm/runtime/limutil.c**. For more details, look directly in [1].

Let us now see the **JVM_SuspendThread()** function, the one called to cause a thread to be suspended.

```
JNIEXPORT void JNICALL
JVM_SuspendThread(JNIEnv *env, jobject this)
{
    Hjava_lang_Thread *p = (Hjava_lang_Thread *)DeRef(env, this);
/* CV */
    struct Classjava_lang_Thread *tid;
    ExecEnv *ee;

    tid = THREAD(p);

    ee = (ExecEnv *)ll2ptr(tid->eetop);
    if (limallowthread(NULL, ee)) {
/* CV */
        if (ll_nez(THREAD(p)->eetop)) {
            (void) threadSuspend(p);
        }
/* CV */
    }
/* CV */
}
```

As you can see, the modified function determines the thread that we want to interfere with, and then calls the **limallowthread()** function (detailed below) to see if the current thread (**NULL**) has permission to interfere with the **ee** thread. In case the permission is not granted, execution continues as if it had been granted, but nothing really occurs.

This is done in respect to the principles stated in subsection 3.1 to limit incompatibilities.

Similar changes are done to the three functions `JVM_ResumeThread()`, `JVM_SetThreadPriority()`, `JVM_StopThread()`.

Now, from `src/share/javavm/runtime/limutil.c`, a file which contains all functions directly regarding the new security checks, here is `limallowthread()`.

```
/*
 * This function is called to verify that thread 'this' is
 * allowed to mess with thread 'thread'. If 'this' is NULL,
 * it is intended to represent the current thread.
 *
 * return values are as usual.
 */
int limallowthread(ExecEnv *this, ExecEnv *thread)
{
    struct AppletInfo *aithis, *aithread;
    int res = 0;

    if (!this) this = SysThread2EE(sysThreadSelf());

    if (this == thread) return 1;

    ALLOCK(sysThreadSelf());
    aithis = FindHash(this);
    aithread = FindHash(thread);

    if ((aithis == aithread) || (aithis == NULL) ||
        (aithis->Flags & F_ALLOW_THREAD))
      res = 1;
    ALUNLOCK(sysThreadSelf());

    return res;
}
```

As can be seen, this function first determines which thread is calling it. This is done by looking at the `this` parameter, and in the case it is `NULL`, it is taken to mean the current thread.

Then, if the two threads (the one determined above and the one on which to work) are the same, permission is always granted.

Otherwise, inside a block of code protected by the monitor, the algorithm discovers the applets to which the threads belong, and if it discovers that the two belong to the same applet or that one belongs to a system thread, then permission is granted, otherwise the calling applet is first checked for the permission to interfere with other applets.

3.5 Configuration

A JVM can be configured to apply the changes shown above to applet with the use of two configuration files: `/etc/applist` and `$HOME/applist`. The first one, `/etc/applist`, contains settings that are system-wide

and apply to all users, whether they wish to make use of the new features or not, while `$HOME/applist` is a user specific one and can be used to modify the system-wide settings. Please do note, however, that the user settings cannot relax the system-wide ones, but can only made them even more stringent.

A useful feature of this files is that it is not necessary to specify the complete codebase of an applet, but it is possible to specify just part of the path or even of the hostname, thus permitting to specify settings that could be applied to whole sites and not just single pages.

4. EFFECTIVENESS AND PERFORMANCE

The proposed approach has shown that, if properly configured, our approach can stop many of the applets from [7]. In the cases of privacy invasion or work-for-me applets, Limiter does not stop them, but prevents them from sending back their results to the originator of the malicious applet.

The same is valid for the applets described in chapter 4 of [6], with the exception of the ones based on the Ackerman functions or analogues, against which this patch offers no protection.

In fact there are two features conspicuous by their absence. The first one is that Limiter has no reasonable way to limit the CPU usage of an applet, and this because we are at too low a level to obtain a meaningful value of CPU time. Indeed, at this level CPU usage can be measured with respect either to the whole system or to the JVM. In the latter case we are bound to obtain high values regardless of real usage, while in the former almost always the opposite holds true.

The second feature is that no attempt has been made to prevent the redefinition of the `stop()` method in an applet. This because there seem to be only two ways to do this: either by substituting the definition the user may have provided with a new one or by hiding the `stop()` method from the JVM. Both are unsatisfactory, because they will result in broken compatibility and changing code behind the programmer's back.

On a different matter it should be noted that the inclusion of this new tests has caused an inevitable hit on performance. However the approach of including them directly in the JVM has paid off again, and as it will be shown these losses are perfectly acceptable.

We executed some tests on a computer with 32MB of ram, and an AMD K6 200 processor and running the applets found in the `demo` directory of the JVM source.

The applets were run using the standalone applet runner distributed with JVM SDK. Execution times were measured by having the applet

itself print the time when it started and when it ended and repeating the process several times in order to reduce measure errors. The functions of the applets tested ranged from sound execution, to drawing, to numer crunching. We measured the execution times of the applets on two different virtual machines: on a standard JVM, and on the patched JVM with all the checks turned on. We found that there was a mean degrade of 0.1%, which we judged inconsequential for normal use.

5. CONCLUSIONS

Our approach has proven to be rather effective in practice, stopping most mischievious applets right before they are allowed to cause damages, and doing so in such a way as to maintain maximum compatibility with all pre-existing code.

As it has been said in the previous section, however, there are two problems our approach does not solve. As a future work, it would be interesting to find a way to integrate these two functionalities in the monitor in a sensible way or, as an alternative, to use it in conjunction with another, external, monitor which could at least provide CPU usage monitoring. An example of such a monitor could be the Applet WatchDog, described in [8].

Another interesting idea, given the effectiveness of the approach, would be to try to implement it also on other architectures, like for example .NET[15] or similar.

References

[1] Vincenzo Ciaschini. The patch itself, http://www.cnaf.infn.it/~marotta/patch.html.

[2] J. Goslin, K. Arnold, The Java Programming Language, Addison-Wesley, Reading, MA 1996

[3] Mary Campione, Kathy Walrath, The Java tutorial, ftp://ftp.javasoft.com/docs/tutorial.tar.gz, May 2001

[4] T. Lindholm, F. Yellyn, The Java Virtual Machine Specification, ftp://ftp.javasoft.com/docs/specs/vmspec.html.tar.gz

[5] G. McGraw, E.W. Felten, Java Security. Hostile Applets, Holes and Antidotes, Wiley, 1997.

[6] G. McGraw, E.W. Felten, Securing Java, Wiley, 1999.

[7] M. LaDue, Hostile Applets Home Page, http://www.cigital.com/hostile-applets/index.html.

[8] M.F. Florio, R. Gorrieri, G. Marchetti, Coping with denial of service due to malicious Java applets, Computer Communications 23 (2000) 1645–1654.

[9] Li Gong, Inside Java 2 Platform Security, Addison-Wesley, Reading, MA, 1999.

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

[11] Finjan Software, www.finjan.com

[12] Citrix, www.citrix.com

[13] D. Malki, M. K. Reiter, A. D. Rubin, Secure Execution of Java Applets Using a Remote Playground, procs. IEEE Computer Society, Symposium on Security and Privacy, pages 40–51, 1998

[14] The Princeton Java Filter, http://www.cs.princeton.edu/sip/JavaFilter/

[15] David S. Platt, Introducing the Microsoft .NET Platform, Microsoft Press International, 2001