

ATTACKING THE JAVA VIRTUAL MACHINE TO CAPTURE CRITICAL USER INFORMATION

Arthur Wongtschowski

Escola Politécnica
Universidade de São Paulo
BR 05508-900, São Paulo(SP), Brazil
arthur@larc.usp.br

Wilson V. Ruggiero

Escola Politécnica
Universidade de São Paulo
BR 05508-900, São Paulo(SP), Brazil
wilson@larc.usp.br

Paulo S. L. M. Barreto

Escola Politécnica
Universidade de São Paulo
BR 05508-900, São Paulo(SP), Brazil
pbarreto@larc.usp.br

RESUMO

Apresentamos nesse paper um ataque contra a Máquina Virtual Java. Esse ataque pode ser utilizado por cavalos de tróia para capturar informações críticas de usuários durante transações on-line. O ataque é possível pela falta de verificação da integridade na Máquina Virtual Java. Um programa malicioso pode alterar as bibliotecas do Java, modificando seu comportamento e desabilitando completamente sua segurança. Um trojan específico poderia explorar essa falha para capturar informações críticas ou desabilitar softwares de segurança executando na máquina da vítima.

ABSTRACT

In this paper, we present an attack against the Java Virtual Machine environment that could be used to capture critical user information during on-line banking transactions. The attack itself is made possible due to the lack of validation of the Virtual Machine's integrity. A malicious program can change the JRE jar files, modifying its behavior and disabling security completely. A specially crafted trojan-horse could exploit this flaw in order to capture critical information or disable security related software from the victim's computer.

1 Introduction

The number of financial applications on the Internet has been growing fast in the last few years. Most of the banks and stores have now websites where users can make online transactions. The access to these websites normally requires an authentication method of some sort in order to validate the user identity. These methods can vary from simple numeric passwords to complex digital signatures protocols.

As financial applications increase over the Internet, so does the amount of money involved in these transactions. This creates an appealing reason for attackers to create many specific attacks (BERTIN, 2001) to capture the user information during the authentication process (PUENTE S. GONZALEZ, 2000; SLEWE; HOOGENBOOM, 2004). Since web servers themselves are normally very secure (FREEDMAN, 2000), these attacks are now migrating to the client side. These attacks can occur in many forms: trojan-horses, keyloggers, phishing scams, DNS hijacking, and so on. In any case, the idea is to capture critical user information during the authentication process. This is called "personification" or "identity-theft" (TIMES, 2003).

In order to thwart this kind of attack, many banks hope that client side security tools could reduce these frauds. These tools go from an anti-virus program,

a firewall to an anti-keylogger¹ or anti-trojan. This means that the banks already know that the integrity of the client's machine cannot be trusted (SEGEV; ROLDAN, 1998; NILSSON ANNE ADAMS, 2005). Many users execute unsafe e-mail attachments or fail to apply all security patches (KOSKOSAS; PAUL, 2004), creating breaches for trojan horses. Even worse, if the user has privileged access to the operating system, the trojan horse can do anything that an administrator can. The UNIX "root" account, the Microsoft Windows NT "administrator" account, or any user on a single-user operating system (e.g., Windows 98 or MacOS) has such access to the operating system. Today, the trojan horse problem is a real and dangerous threat to Internet Banks.

In this paper, we describe a new form of attack that could be exploited by trojans to capture critical user information. The basic idea of this attack was proposed by (WHEELER; LUO, 2001; GOLDBERG D. WAGNER; BREWER, 1996; DEAN E. W. FELTEN; BALFANZ, 1997), but with a very different scope. In contrast to most of the capturing techniques described before, the new attack is multi-platform, transparent and very hard to detect. Like trojan horses and keyloggers attacks, it also requires local software installation on the user's machine, but only once. But today's Internet scenery shows us that this is

¹Anti-Keylogger is a program which prevents the real time capture of the keyboard strokes in a local machine

not as farfetched as one might expect.

We also present a discussion about possible solutions to the described problem. Since attackers are still not aware of this problem, understanding the attack and knowing how to thwart it is extremely important to prevent future frauds. The idea of this paper is not only to bring to the public a dangerous problem that could be exploited by trojan horses, but also to discuss possible solutions before the attack is used to commit real frauds.

This paper is organized as follows: section 2 describes the scope of the proposed attack. In section 3 we briefly present the basics about Java security, focusing in the relevant aspects to better understand the work developed in this paper. In section 4 we explain the attack we will use. Section 5 presents new applications of the attack that could be used to capture critical user information. In section 6 we briefly explain what could be done to prevent this attack. We draw our conclusions in section 7 and specify what is still missing.

2 Scope of the attack

The scope of this paper is to describe an attack that could potentially be used as a tool to commit a bank fraud. An attacker that wishes to do an "identity-theft" attack needs to follow several steps in order to achieve such goal. In very short terms, a common trojan attack would be composed by the following:

1. Get access to the victim's machine.
2. Steal the victim's critical information, such as passwords.
3. Use this stolen information to personify the user

This paper describes an attack that could be used by a trojan to achieve step number 2. The first and last steps are not described in detail here.

3 The Security Model of the Java Language

The Java 2.0 Security Model is a key feature of the language's architecture that makes it appropriate for network environments (MATAYOSHI, 1998; VENNERS, 1996). Java security has become very important because its environment allows a potential attack from any computer in the network. These concerns are especially strong when remote programs (software) are brought across the network in order to be executed locally (namely, Java Applets). It is likely that a user will encounter non-reliable applets while surfing through the net. To face these potentially hostile programs, the

security mechanisms of the Java language must establish different privileges for codes coming from different sources.

The security model of the Java language should protect the user from hostile programs brought across the net from untrusted sources. To accomplish such protection, the Java environment uses what we call a *sandbox* inside of which the Java program is executed. The program can try to execute anything it wants, but all of its operations are bounded by the limits imposed by the sandbox. Java applet brought across the net cannot perform innumerable actions, such as:

- Read or write files from the file system
- Establish connections across the network (except with the server the applet was downloaded from)
- Create new processes
- Dynamically load new libraries and call native methods directly

By avoiding that remote loaded code executes certain operations, the Java security model protects the local user from hostile programs.

In addition to many safety-related characteristics of the Java language, Java security relies on a multipart defense. The default sandbox is made of three interrelated parts: the *Verifier* (VENNERS, 1997b), the *Class Loader* (VENNERS, 1997a), and the *Security Manager*. These parts are like links in a chain: If any of the three parts breaks, the entire security system breaks. We will focus here on last part: the Security Manager.

3.1 The Security Manager

The final part of the Java security model is the Security Manager. This part is responsible for restricting the way an applet calls Java APIs (MCGRAW; FELTEN, 1999).

The job of the Security Manager is to keep track of who is allowed to do what. A standard Security Manager will disallow most operations when they are requested by untrusted code, and will allow trusted code to do whatever it wants. The Security Manager is a single Java object that performs runtime checks on sensitive² methods. Code in the Java library consults the Security Manager whenever a potentially dangerous operation is attempted. The Security Manager can veto the operation by generating a *SecurityException*. Decisions made by the Security Manager take into account the origin of the requesting class.

The Java library's use of the Security Manager works as follows:

²Sensitive code is one that executes a potentially dangerous operation. A dangerous operation is one that may change the access privileges of the Java Virtual Machine

1. A Java program makes a call to a potentially dangerous operation in the Java API.
2. The Java API code asks the Security Manager whether the operation should be allowed.
3. The Security Manager throws a SecurityException back to the Java API if the operation is denied. This exception propagates back to the Java program.
4. If the operation is permitted, the Security Manager call returns without throwing an exception, and the Java API performs the requested dangerous operation and returns normally.

The Java runtime library is written so that all requests to perform dangerous operations are referred to the Security Manager. Access checks are used for thread access, OS access, network access, and Java component access.

```
import java.applet.Applet;
import java.awt.Graphics;
import java.io.*;

public class FileWrite extends Applet {
    public void paint(Graphics g) {
        // declare a file output object
        FileOutputStream out;
        // declare a print stream object
        PrintStream p;
        try {
            // Create a new file output stream
            // connected to "myfile.txt"
            out = new FileOutputStream("myfile.txt");
            // Connect print stream to
            // the output stream
            p = new PrintStream( out );
            p.println ("This is written to a file");
            p.close();
            g.drawString("FileWrite OK", 50, 25);
        } catch (Exception e) {
            g.drawString("FileWrite error!", 50, 25);
        }
    }
}
```

4 Description of the attack

We now present an example attack only for demonstration purposes. The attack explained in this section does not capture critical user information, but demonstrate how the Java APIs behavior can be arbitrarily modified by a trojan horse. The attack utilized by a real trojan horse would be somehow different, but the same basic principles apply to both.

The idea behind the attack appears with a simple question: Does the Virtual Machine verify the integrity of the Java run time libraries before it executes it? If so, how?

The attack is very simple and somehow obvious. In order to achieve it, we took the following steps:

- Decompress the file rt.jar (Java Runtime Libraries) into a temporary directory
- Decompile a specific class or download the source code from the Internet
- Recompile this class, changing somehow its behavior
- Replace this class in the temporary directory
- Compress the rt.jar file, replacing it in the Class-Path directory

With these modifications, we were able to change the way the Java APIs behave.

In order to understand the seriousness of the problem, lets take a look at a simple example. Suppose there is an Applet called *FileWrite* with the following code:

This is a very simple applet that tries to open a file named *myfile.txt* and write the line *This is written to a file* into it. When executed on top of a well behaved Java Virtual Machine, it is clear that the Applet will be unable to open the file (due to the limits imposed by the SandBox).

As we seen before, the Java API asks the SecurityManager if the code executing may or may not perform a certain action. If the SecurityManager wants to disallow this call, it simply throws an exception. The code below is extracted from Sun's JVM SecurityManager.java:

```
...
public void checkWrite(String file) {
    checkPermission(new FilePermission(file,
        SecurityConstants.FILE_WRITE_ACTION));
}
...
```

This function verifies if the executing code may write to a file. It throws a SecurityException exception if the calling thread does not have permission to access the specified file. That is exactly what happened before when we tried to execute our FileWrite applet.

Now comes the part where we change the way the virtual machine behaves. Lets take the above code from the SecurityManager class, and just change the following line:

```
...
public void checkWrite(String file) {
    // checkPermission(new FilePermission(file,
    // SecurityConstants.FILE_WRITE_ACTION));
}
...
```

As one can see, we just removed the line that actually checked whether the running code could or could not

write to a file, transforming them in a comment. After recompiling this code and replacing the JRE jar file with this new one, we executed the FileWrite applet again. To our surprise, no errors were reported by the virtual machine, and the applet could successfully write to a file.

Although this is a very simple example, with a non-malicious applet, it is clear that this attack could be extended to any other class (and API) of the Virtual Machine. By doing so, an attacker who exploits this vulnerability could break the entire SandBox or make any Java program behave the way he wants.

This attack was successfully executed with the following Virtual Machines:

1. Sun's Java Virtual Machine 1.4.2 for Microsoft Windows®
2. Sun's Java Virtual Machine 1.5.0 for Microsoft Windows®
3. Microsoft's Java Virtual Machine 5.0.20.9 for Microsoft Windows®
4. JRE 1.3.1 Java HotSpot for MAC OS X® 10.2
5. IBM Java Virtual Machine 1.4.2 for Microsoft Windows®

The attack against the Microsoft and the IBM virtual machines was a little different: since there is no source code for the those classes, we had to decompile (VLIET, 1996) the SecurityManager class. Furthermore, with the Microsoft virtual machine, we had to change two different classes in order for the attack to work: The SecurityManager.class and StandardSecurityManager.class. Other than that, the attack worked the same way.

5 Real Scenarios

In order to show the seriousness of this attack, we present some real scenarios where changing the way the Java APIs works may be used to capture critical information from a victim and may compromise the entire security of the machine.

5.1 Breaking into the victim's file system

First of all, what an attacker needs to do in order to start the attack is obtain access to the user file system. This is not as farfetched as one might think. Below we present some statistics about virtual frauds in Brazil obtained from March to June of 2005 (IDGNOW, 2005; FOLHA, 2005):

- Virtual frauds have grown 1331% in the second trimester of 2005, when compared with the same period of 2004.
- E-mails scams have grown 226% in May when compared to the previous month. Three out of ten e-mails sent in May contained viruses.
- In March of 2005, the police arrested a Brazilian hacker that could have stolen more than US\$ 37 millions in virtual frauds.

Since a great part of the virtual frauds in Brazil are performed by trojan horses, it is reasonable to assume that attackers today can already obtain access to many users machine's.

Even so, below we present two alternatives an attacker could use to install a trojan horse in the victim's machines:

5.1.1 Exploit vulnerabilities in the operating system or applications

This form of attack is somehow complicated and limited. Exploiting vulnerabilities such as buffer overflow is technically demanding, and one will mostly attack known vulnerabilities, which will probably be fixed by that time. But, on the other hand, worms such as Blaster³ and Sasser⁴, although exploiting known and already repaired vulnerabilities, are still a major plague on the Internet. Users rarely install critical updates, leaving their computers susceptible to outsiders (MOORE VERN PAXSON; WEAVER, 2003).

5.1.2 Deceive the user

This is the simplest and most common way of getting access to a user's machine (CERT, 1999). Today, trojan-oriented SPAM is a major concern of all security analysts. Most users are not educated when it comes to secure browsing and e-mailing. The main idea is to deceive the user by getting him to install something in his machine that appears benign, but actually contains a trojan-horse. Below is a text extracted from (CERT, 1999), presenting CERT advisory on trojans:

Users can be tricked into installing Trojan horses by being enticed or frightened. For example, a Trojan horse might arrive in email described as a computer game. When the user receives the mail, they may be enticed by the description of the game to install it. Although it may in fact be a game, it may also be taking other action that is not readily apparent to the user, such as deleting files or mailing sensitive

³Blaster is a worm that exploits the DCOM RPC vulnerability in Windows 2000; Microsoft suggests that at least 8 million Windows computers have been infected by Blaster

⁴This worm spreads by exploiting a recent Microsoft vulnerability, spreading from machine to machine with no user intervention required. Microsoft has claimed that almost 1.5 million Windows customers downloaded a cleanup tool for the Sasser internet worm in the first two days after it began offering it

information to the attacker. As another example, an intruder may forge an advisory from a security organization, such as the CERT Coordination Center, that instructs system administrators to obtain and install a patch.

Other forms of "social engineering" can be used to trick users into installing or running Trojan horses. For example, an intruder might telephone a system administrator and pose as a legitimate user of the system who needs assistance of some kind. The system administrator might then be tricked into running a program of the intruder's design.

Software distribution sites can be compromised by intruders who replace legitimate versions of software with Trojan horse versions. If the distribution site is a central distribution site whose contents are mirrored by other distribution sites, the Trojan horse may be downloaded by many sites and spread quickly throughout the Internet community.

Because the Domain Name System (DNS) does not provide strong authentication, users may be tricked into connecting to sites different than the ones they intend to connect to. This could be exploited by an intruder to cause users to download a Trojan horse, or to cause users to expose confidential information. . .

. . . Finally, a Trojan horse may simply be placed on a web site to which the intruder entices victims. The Trojan horse may be in the form of a Java applet, JavaScript, ActiveX control, or other form of executable content.

The trojan could even come equipped with a compiled version (one for each Virtual Machine) of the Java class it is going to replace in the JRE files, so that it could apply the attack described in this paper more easily.

5.2 Authentication Applets

First, let's define what we mean by *Authentication Applets*. Any applet that may aid the user during its authentication process with any site or application may be called an authentication applet. Some examples would be: simple applets that the user must type their password in; virtual keyboards; applets that encrypt passwords; applets that ask for a private key in order to complete a challenge-response protocol; applets that provide biometrical authentication; etc. . .

The main purpose of these applets is to increase user security during the authentication process. Virtual Keyboards are intended to prevent keystroke capturing. Challenge-response applets try to implement a more complex validation method using digital certificates. Encryption applets are designed to increase the difficulty of networking sniffing. And so on.

Another main characteristic of authentication applets is that they run on many different machines of many different clients. It is impossible to vouch for the security and safety of all this machines. Some of them may have been compromised, some may still be intact. But in any case, the security applet must be able to work properly and safely.

5.2.1 Mounting the attack

With the attack described before, we can come up with a simple, multi-platform, almost detection-proof attack

that will affect a great family of these applets. The attack proceeds as follows: we locate the Java class that receives the password when it is entered by the victim. Then, all there is to do is replace this class adding a few lines at its start. These lines may store the victim's password to a file or even send it across the network (of course, if the applet is not signed, we will also need to replace the SecurityManager class in order to give the applet the permissions it needs). With this attack, the authentication applet would still work properly, giving no clues of the attack to the victims whatsoever.

With simple applets that take passwords as input, we just overwrite the class that implements the textbox field. With encryption applets, we could replace the class that encrypts the password and retrieve it before any encryption is done. With private-key-handling applets, we could replace the file access APIs in order to capture the private key itself.

A strong point about these attacks is that they are very hard to detect. Even experienced user won't notice any difference during the authentication process. Also, there is no clear way to check a virtual machine's integrity: the applet cannot validate itself, since the virtual machine may be compromised and even the validation functions used by the applet may now be untrusted; and up to date, the most common virtual machines do not perform any kind of integrity check themselves.

5.3 Security Related Software

Another family of applications that may suffer greatly with this attack are *Security Related Software* applications. Let's take a look at some security related software, like anti-virus, personal firewalls and anti-keylogger programs.

One of the main features of these products is that they have to function in an unsafe or compromised environment. An anti-virus should work in a virus-infected machine. A firewall should prevent malicious programs installed on the current machine from communicating with the outside. An anti-keylogger needs to prevent installed keyloggers from capturing keystrokes.

What happens if the trojan, virus, or keylogger decides to disable the protection software? A well designed security software should be able to prevent this kind of action. Such programs should have validations of their own integrity or at least they may alert the user in case something fails to load or is tampered with. We are not saying that it is easy for a program to check its own integrity. On the contrary. But nevertheless, it is an important and necessary feature that security related software should have.

5.3.1 Mounting the attack

Let us examine the following issue: What do the attack has to do with security software validation? We believe

that here is where this attack can produce serious damage.

Suppose there exists a perfect security-related software which cannot be disabled or tampered with in any way. Suppose this software is written in Java. With the attack, we identified a way for the attacker to easily disable this security software. Now, all the attacker needs to do is change the Java APIs in a way that the security program will still believe that all is safe, but actually the whole security environment is compromised.

Against an anti-virus program, an attacker could change the file APIs, sending to the anti-virus only files with safe content no matter what its scans. Against firewall programs, an attacker could change the TCP APIs in order for them to always return that there is no network traffic at all. A keylogger program could change the keyboard handling APIs, so that an anti-keylogger software would always believe that there is no key being pressed.

It should now become clear the seriousness and effectiveness of the attack.

6 Preventing the attack

Finding a way to thwart this attack is not an easy task. It is clear that no Java program can verify the integrity of the virtual machine: since the machine could have been tampered with, there is no guarantee that even the functions we use to validate the virtual machine are safe.

The main problem here is that the integrity of the Virtual Machine has been compromised, and there is no easy way for us to validate it. So basically, what needs to be done is find a way to safely verify this integrity.

One idea that might appear simple enough is to create an outside program, written in platform dependent code, which would validate the Virtual Machine. This idea, although at first very appealing, has many pitfalls: by writing machine-dependent code, we lose the Java portability; there are many different virtual machines out there, with different versions and releases. It is almost impractical to maintain an updated list of them all; it is also very difficult to guarantee that this outside program wouldn't be deleted or tampered with.

The job of validating the integrity could be passed to the virtual machine itself. Even so, an attacker might still replace the virtual machine executables. But, since these executables are platform dependent and reasonably large, the attack would become much more complex. So, an immediate (but not definite) solution to this problem would be for the virtual machines manufacturers to introduce a verification mechanism inside the virtual machine itself, preferentially using a digital signature scheme of some sort.

One different approach proposed by (MITTELSDORF, 2004) is to use a trusted computing platform built on

common computers, based on virtual machine monitors, static and dynamic attestation. That way, even if the Java Virtual Machine is highly compromised, we should be able to run security applications in an isolated form. The main idea here is to do a chain validation; for example, the hardware processor could validate the correctness of the BIOS; the BIOS then could validate the Operating System, which could then validate the applications, and so on. We refer to (MITTELSDORF, 2004) for more information.

7 Conclusions

We have presented a new variant of a attack that could potentially make any Java software unsafe and untrusted. Although we know that the premises for this attack are very strong, we believe to have shown real examples where the attack could be extremely effective and dangerous.

Generally speaking, the same idea of this attack could be performed in any platform that executes software. But normally, these platforms (such as Operating Systems) are loaded at system startup, making it difficult for an attacker to change its core components. Some are even digitally signed. We are not saying that it is impossible to replace the Linux kernel or to change a system DLL in Windows. All we are saying is that it is much easier to switch the Java runtime libraries than any of the above.

As described in the section 6, there are ways to prevent this kind of attack. None of them appears very simple and are not easily implemented, at least for now. But nevertheless, there is a path we should follow in order to improve software integrity validation.

Java is still a very important technology. It provides portability and easy code maintenance, along with many other features. While used on server side, such as in JSP pages or Servlets, the Java security remains untouched. The same can be said about common Java applets, such that implements games or graphic interfaces. But care should be taken while writing security related software or authentication applets that might run on many different platforms of many different users. There is no guarantee that this software will be running on a safe Virtual Machine.

References

- BERTIN, M. *The new security threats*. 2001. Available from: <http://www.zdnetindia.com/biztech/enterprise/features/stories/11609.htm%1>.
- CERT. *CERT Advisory CA-1999-02 Trojan Horses*. 1999. Available from: <http://www.cert.org/advisories/CA-1999-02.html>.

- DEAN E. W. FELTEN, D. S. W. D.; BALFANZ, D. Java security: Web browsers and beyond. *Internet Beseiged: Countering Cyberspace Scofflaws*, Oct. 1997.
- FOLHA. *Folha Online*. 2005. Available from: <http://www.folha.uol.com.br/>.
- FREEDMAN, D. H. *How To Hack A Bank*. 2000. Available from: <http://www.forbes.com/asap/2000/0403/056.html>.
- GOLDBERG D. WAGNER, R. T. I.; BREWER, E. A. A secure environment for untrusted helper applications: Confining the wily hacker. *Proceedings of the 6th Usenix Security Symposium*, 1996.
- IDGNOW. *International Data Group (Brasil)*. 2005. Available from: <http://www.idgnow.com.br/>.
- KOSKOSAS, I. V.; PAUL, R. J. The interrelationship and effect of culture and risk communication in setting internet banking security goals. *Proceedings of the 6th international conference on Electronic commerce*, v. 60, p. 341 – 350, 2004.
- MATAYOSHI, C. M. Modelo de segurança da linguagem java. *LARC - Laboratório de Arquitetura e Redes de Computadores*, 1998.
- MCGRAW, G.; FELTEN, E. *Securing Java*. [S.l.]: John Wiley & Sons, Inc., 1999. Available from: <http://www.securingjava.com/>.
- MITTELSDORF, A. W. Uma plataforma para computação com confiança baseada em monitor de máquinas virtuais e atestamento dinâmico. *Escola Politécnica da Universidade de São Paulo*, 2004.
- MOORE VERN PAXSON, S. S. C. S. S. S. D.; WEAVER, N. *The Spread of the Sapphire/Slammer Worm*. 2003. Available from: <http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html>.
- NILSSON ANNE ADAMS, S. H. M. Building security and trust in online banking. *Conference on Human Factors in Computing Systems*, p. 1701 – 1704, 2005.
- PUENTE S. GONZALEZ, J. S. P. H. F. de la. Viral attack to internet banking applications. *Aerospace and Electronic Systems Magazine, IEEE*, June 2000.
- SEGEV, J. P. A.; ROLDAN, M. Internet security and the case of bank of america. *Communications of the ACM*, v. 41, n. 10, p. 81 – 87, October 1998.
- SLEWE, T.; HOOGENBOOM, M. Who will rob you on the digital highway? *Communications of the ACM*, v. 47, n. 5, p. 56 – 60, May 2004.
- TIMES eCommerce. *Identity Theft Is Fastest-Growing Online Crime*. 2003. Available from: <http://www.epaynews.com/index.cgi?survey=&keywords=ID%20theft&optional%=&subject=&location=&ref=keyword&f=view&id=1056453397622215212&block=1>.
- VENNERS, B. Java's security architecture - an overview of the jvm's security model and a look at its built-in safety features. *Java World*, Aug. 1996. Available from: <http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood.html>.
- VENNERS, B. Security and the class loader architecture - a look at the role played by class loader in the jvm's overall security model. *Java World*, Sep. 1997. Available from: <http://www.javaworld.com/javaworld/jw-09-1997/jw-09-hood.html>.
- VENNERS, B. Security and the class verifier - a look at the role played by class verifier in the jvm's overall security model. *Java World*, Oct. 1997. Available from: <http://www.javaworld.com/javaworld/jw-10-1997/jw-10-hood.html>.
- VLIET, H. van. *Mocha, the Java Decompiler*. 1996. Available from: <http://www.brouhaha.com/~eric/software/mocha/>.
- WHEELER, A. C. D.; LUO, A. X. J. Java security extensions for a java server in a hostile environment. *17th Annual Computer Security Applications Conference (ACSAC'01)*, Dec. 2001.