Hostile Java Applets

David Evans, *University of Virginia*

INTRODUCTION

Java was introduced in 1995 as both a high-level programming language and an intermediate language, Java Virtual Machine language (JVML, sometimes called *Java byte codes*), and execution platform, the Java Virtual Machine (Java VM), designed for secure execution of programs from untrusted sources in Web browsers (Gosling, 1995). These small programs that are intended to execute within larger applications are known as *applets*. Java runs on a wide range of platforms scaling from the Java Card smart card environment (Chen, 2000) to the Java 2 Enterprise Edition (J2EE) for large component-based enterprise applications (Singh, Stearns, Johnson, & the Enterprise Team, 2002). This chapter focuses on the Java 2 Platform, Standard Edition (J2SE), which is the most common platform for desktop applications and servers, including Web browsers. Most of the security issues are the same across all Java platforms, however. Because of the limited functionality of the Java Card environment, some of the security concerns with the standard edition do not apply; the added complexity of J2EE raises additional security issues (Gong, Ellison, & Dageforde, 2003).

The Java programming language adopted most of the syntax of C++ and semantics of Scheme. Because the Java programming language does not provide the type unsafe features of C++ (including pointer arithmetic and unchecked type casts), programs written in the Java programming language (and compiled correctly and executed in a correct virtual machine implementation) can guarantee certain security properties. However, because applets are transmitted as JVML there is no guarantee that Java applets were created using the Java programming language. JVML programs can be created using a compiler for a different programming language or edited directly. Hence, all security claims made for executing Java applets are based solely on the mechanisms provided by JVML and the Java VM execution platform.

The Java Virtual Machine attempts to provide security properties that enable code from untrusted sources to be safely executed. It confines executing applets to a *virtual playpen* (sometimes called a *sandbox*) that limits what they can do and mediates access to external resources according to a policy. Malicious applets can attempt to behave in ways that are detrimental to the host. The most serious malicious applets find a way to circumvent Java's security mechanisms and gain complete control of the host machine. These attack applets depend on exploiting a vulnerability in a Java implementation. Other classes of malicious applets may disturb the victim without circumventing Java's security mechanisms by behaving in an annoying or disruptive way that is within the behaviors permitted by the policy.

The next section of this chapter presents an overview of Java's security mechanisms. Next, we provide an overview of the Java security model. The next section describes Java's mechanisms for low-level code safety necessary to ensure that malicious applets cannot circumvent high-level security mechanisms. The third section describes Java's high-level code safety mechanisms that can impose a policy on an applet execution. The fourth section discusses hostile applets that behave maliciously without circumventing Java's security mechanisms, and the fifth section considers exploit applets that damage the victim by circumventing Java security mechanisms. The sixth section concludes.

JAVA SECURITY OVERVIEW

Security in Java is based on a model in which code executes in a virtual machine that mediates access to critical system resources. Instead of executing directly on the host machine and having access to all resources a user-level process can access on the machine, a Java applet executes inside the Java VM. The Java VM itself is typically a user-level process running on the host machine, so its access is limited by the underlying operating system according to the permissions assigned to its process owner. The Java VM, however, places additional constraints on

P1: jth
JWBS001C-142.tex WL041/Bidgoli WL041-Bidgoli.cls September 16, 2005 9:8 Char Count= 0

what the applets it executes may do. In particular, it mediates access to system resources that are considered security critical.

The Java VM executes programs written in JVML, a stack-based language with a relatively small and simple instruction set. Although they are often produced from a Java programming language program by a Java compiler, JVML programs can be produced manually or from some other language by a compiler that targets JVML.

Java security depends crucially on knowing that the only way an applet can manipulate controlled resources is through Java application program interface (API) method calls that perform security checks before permitting security-critical operations. Hence, we can divide Java security mechanisms into two categories:

- *Low-level* code safety mechanisms designed to ensure that all manipulations of critical resources are performed only through Java API calls
- *High-level* code safety mechanisms designed to enforce an access control policy on resource manipulations done through Java API calls.

Java's security model has evolved since the initial 1.0 release. Java 1.0 distinguished local trusted code from untrusted applets, but did not provide any mechanisms for signing code or applying different policies to different external code. Java 1.1 introduced code signing whereby cryptographically signed applets could execute as trusted code. The Java 2 platform provided richer mechanisms for access control and policy association. It allows different policies to be associated with different applets running in the same Java VM. Except when specifically noted, this chapter describes the security model provided by version 1.4.2 of the Java 2 platform.

The next sections describe what low-level code safety and policy-specific code safety involve, first describing Java's mechanisms for providing low-level code safety and second describing Java's mechanisms for providing policy-specific code safety.

Low-Level Code Safety

Low-level code safety ensures that all resource manipulation is done through standard method calls. It is necessary to prevent malicious applets from circumventing Java's security mechanisms. Low-level code safety requires *type safety*, *memory safety*, and *control flow safety*.

Type safety means that data values are always manipulated in a type-consistent way. All values in Java have a type, and different operations are possible on different types. For example, integers can be added and object references can be dereferenced but integers cannot be dereferenced. If it were possible to dereference integer values, or to perform arithmetic operations on object references, it would be possible to manipulate arbitrary locations in memory, thereby circumventing Java's security mechanisms. For example, if type safety is not enforced, an applet could create an integer constant that corresponds to the address where the security policy is stored and then use that integer as an object to change the security policy.

Memory safety means that reads and writes to memory must be within the storage allocated for an object. Without memory safety, a malicious applet could access memory beyond the range allocated for an object to change the value of some other value. The standard stack smashing buffer overflow attack (Aleph One, 1996) operates this way: by writing data beyond the space allocated for an object, the attacker can overwrite the return address on the stack and inject malicious code into memory. Memory safety prevents this class of attacks in Java, because it should not be possible for a malicious applet to write to memory beyond the allocated space for an object.

Control flow safety ensures that execution jumps are always to valid locations. In Java, this means that execution may not jump into the middle of a method but only to the beginning. Without control flow safety, a malicious applet could jump directly to security critical code inside an API method, thereby circumventing the security checks that must be done before performing the critical operation.

High-Level Code Safety

High-level code safety is concerned with controlling access to security-critical resources. Unlike low-level code safety, which is necessary to ensure that security mechanisms are not circumvented and essentially the same mechanisms are necessary to enforce *any* security policy, high-level code safety enforces a resource control policy that will vary by users, systems, and applications. Enforcement is done using a straightforward reference monitor: before an applet attempts a security critical operation, checks are performed to determine whether the intended operation is permitted by the policy. If the check fails, the security critical operation is not permitted and a security exception is raised. A policy is associated with code using a ClassLoader, which controls how new classes are loaded into a Java VM.

The two main challenges in high-level code safety are defining a policy that allows enough access to enable applets to do useful things and associating the appropriate policy with particular code. Later sections in this chapter describe the permissions that can be granted in Java to control access to system resources, describe how policies for different applets and executions are defined using those permissions and how a particular policy is associated with an executing class, and explain Java's mechanisms for enforcing those policies.

LOW-LEVEL CODE SAFETY MECHANISMS

Java enforces low-level code safety using a combination of static and dynamic checks. Static checks are performed by the Java bytecode verifier before a Java class file is loaded into the Java virtual machine; dynamic checks are performed by the Java virtual machine to check certain properties before an instruction that could violate low-level code safety is executed.

Bytecode Verification

The Java bytecode verifier statically checks certain properties of Java class files before they are loaded into the

P1: jth
JWBS001C-142.tex WL041/Bidgoli WL041-Bidgoli.cls September 16, 2005 9:8 Char Count= 0

virtual machine (Lindholm & Yellin, 1999). It checks that the class file is in the correct format and that it contains the data it should. Most important, it checks properties of the class file that are necessary for low-level code safety. The bytecode verifier by itself is not sufficient to completely guarantee low-level code safety, but it does establish properties that in combination with the run-time checks are enough to provide low-level code safety. These properties include ensuring that values of the appropriate types are on the stack before every instruction and that a value stored in a memory location is treated as the same type of value when it is loaded.

The bytecode verifier simulates execution of a JVML program. In general, checking low-level code safety is an undecidable problem that requires reasoning about all possible executions of a program. To enable efficient verification, Java's bytecode verifier makes some conservative assumptions and puts off checking certain instructions (such as type casts and array fetches) until run time. The conservative assumptions mean that there are some safe programs that the verifier will reject, but that there are no unsafe programs that are accepted by a correctly implemented verifier. One assumption made by the bytecode verifier is that the type stored in a particular local memory location is the same throughout a procedure's execution. This, along with a few other similar assumptions, means that the bytecode verifier can simulate execution without needing to follow any backward jumps (e.g., loop repetitions) and that each method call can be checked based only on the type signature, without any need to simulate the method body for each call site.

If the verifier finds a type violation or an instruction that would cause a stack overflow or underflow, it raises an exception and the code will not be loaded to execute in the Java VM. If the code is accepted by the verifier, it means that all operations except checked casts and array assignments are type safe, all jumps are to valid locations, and all memory loads and stores are either to valid locations or are done through instructions that will be checked for memory safety at run time.

Run-Time Checks

Because of the limits of static analysis, some type and memory safety properties cannot be checked by the bytecode verifier. These properties must be checked at run time by the Java VM. Run-time checking is typically simpler and hence less prone to vulnerability than static checking, but it imposes an execution time penalty because the checking is done during the program execution. The other disadvantage of run-time checking is that problems will not be detected until the program has already begun executing.

Array fetches and stores in Java use load and store instructions that expect an array and an integer index value that identifies the array element on the stack. Memory safety depends on the index being within the array bounds. Arbitrary computations can calculate the index value, so it is impossible for the bytecode verifier to determine whether the index is within range. Hence, the Java VM must check at run time that all array indexes are within bounds. If the index is not in bounds, an ArrayIndexOutOfBoundsException is raised and the attempted array access is prevented.

Type casts are necessary in Java to change the apparent type of an object to a subtype of its apparent type. The actual type of the object must be a subtype of the cast type, but it is not always possible to determine the actual type of an object at verification time. For type casts that are not known to be safe at compile time, the Java compiler produces a checkcast instruction that changes the apparent type of an object for bytecode verification but is checked at run time to ensure that the actual type of the object matches the cast type.

The final run-time low-level code safety check performed by the Java VM is for stores into elements of array parameters. Java's type system allows an array of type S elements to be passed as a parameter whose type is an array of T elements if S is a subtype of T. However, if the method body stores a value of type T into the array, it may not be of type S and would violate the array element type. The verifier checks the method body according to the apparent type of the array parameter, but the actual type is not known at verification time. Hence, the array store must be checked at run time. If the type of the value stored in the array is not a subtype of the actual element type of the array, the array store is prohibited and an ArrayStoreException is raised.

In conjunction with the bytecode verifier, the run-time safety checks provided by the Java VM are designed to ensure that Java applets cannot violate type safety, memory safety, or control flow safety. As long as they are implemented correctly (see the Violating Low-Level Code Safety section for a discussion of hostile applets that exploit bugs in the bytecode verifier) and the assumptions they make about the computing environment are true (that section includes a description of an example attack that depends on violating those assumptions), they prevent hostile applets from being able to manipulate resources without going through the high-level code safety mechanisms.

HIGH-LEVEL CODE SAFETY MECHANISMS

Low-level code safety mechanisms prevent hostile applets from circumventing the high-level code safety mechanisms provided by the Java VM. Those high-level code safety mechanisms are designed to impose a policy on an executing applet that limits its access to system resources. Depending on the level of trustworthiness associated with the applet, a different policy may apply that enables and disables permissions appropriately. The next sections describe the types of policies Java can impose and how they are defined, how a particular policy is associated with code, and how the Java VM enforces a high-level code safety policy on an execution.

Permissions

A Java policy specifies what actions an applet may perform. Particular actions require specific permissions. If an applet attempts an action, but does not have the associated permission, the action will not be permitted and a security exception is raised.

P1: jth
JWBS001C-142.tex WL041/Bidgoli WL041-Bidgoli.cls September 16, 2005 9:8 Char Count= 0

Java supports 19 permission classes for specifying different actions; many of these permissions can be parameterized (Sun 2002a). For example, the java.io.FilePermission class represents permissions related to file input and output. An instance of the class is a pathname and a set of actions (selected from read, write, execute, and delete) permitted for that pathname.

Summarizes all the permissions that can be granted by a Java security policy. Many of the permissions are inherently dangerous: if they are granted, a hostile applet could use them to obtain other permissions. For example, granting the ReflectPermission permits an applet to use Java's reflection methods to access field and call methods without normal access checks being performed. A hostile applet could use this permission to manipulate other resources in ways that circumvent the security policy. For example, reflection could be used to invoke the private java.io.File.delete0 method to delete a file regardless of whether the applet has the required permission (normally, the java.io.File.delete method first checks whether the caller has permission to delete the file and then invokes delete0 to delete the file). The permission RuntimePermission.setSecurityManager allows an applet to replace the security manager with a custom security manager, thereby circumventing any checking done by the original security manager.

Policies

When loading a class in Java, a subclass of the abstract class, ClassLoader, is responsible for creating the association between the loaded class and its protection domain. These static permissions are associated with the class at run time through a protection domain (PD). Each Java class will be mapped to one PD, and each PD encapsulates a set of permissions. A PD is determined based on the person running the code, the code's signers, and the code's origin. If two classes share the same context (principal, signers, and origin), they will be assigned to the same PD, because their set of granted permissions will be the same.

Policies are sets of rules that determine whether a particular action is permitted. To assign permissions, the class loader checks the security policy defined, and the policy grants specific permissions to code based on certain code attributes and then associates the permissions with the class by a PD. Prior to J2SE 1.4, permissions were assigned statically at load time by default, but now dynamic security permissions are supported (Sun, 2003). This provides more flexibility, but increases complexity and makes reasoning about security policies difficult.

Java policies are defined by specifying the permissions granted in a policy file. The policy file specifies permissions to be granted based on properties of an execution: the origin of the code, the digital signers of the code (if any), and who is executing the code. A user can edit the policy files with a normal text editor or the PolicyTool. Java's policies are also affected by a system-wide properties file, java.security, that specifies paths to other policy files, a source of randomness for the random number generator, and other important properties. These security properties should not need to change often, but they are important in understanding how the policy is configured. Changes to this file could greatly influence the system's policy, since a user could change which files are used for the actual policy in this file.

The policy file contains a list of grant entries. Each grant entry identifies a context that determines when the grant applies and then lists a set of permissions that are granted in that context. The context may specify the signers of the code (a list of names, all of whom must have signed the code for the context to apply), the origin of the code (code base uniform resource locator [URL]), and one or more principals (on whose behalf the code is executing). If no principals are listed, the context applies to all possible principals.

The following is an example grant entry:

```
grant signedBy "John" {
      permission java.io.FilePermission
            "C:\\temp\\*", "read, write";
};
```

This grants all applets signed by "John" permission to read and write files in the C:\temp\ directory. The grant entry

```
grant codebase "http://www.cs.virginia.edu",
      principal javax.security.auth.x500.
X500Principal "cn=evans" {
      permission java.io.FilePermission
"/usr/evans/*", "read, write";
};
```

grants applets from URLs within the www.cs.virginia.edu domain permission to read and write files in the /usr/evans/ directory when they are running on behalf of principal "evans."

Java is installed with one system-wide policy file, but if a user adds his or her own policy file, that file will also be taken into account. The granted permission set is the union of the permissions granted in all the policy files, so the default permissions is the union of both of these policy files' granted permissions. This is dangerous, because a user may have a difficult time of actually determining what permissions are actually being granted. Further, it means a user can make the policy less restrictive than the system policy, but cannot make the policy more restrictive.

Because most users are unlikely to change the security policy themselves, hostile applets target the default system security policy. Sun's default Java security policy (J2SE 1.4.2) grants all permissions to code loaded from the lib/ext subdirectory of the Java installation on the local file system and grants several permissions to all applets (including untrusted applets):

- Listen on unprivileged ports (port numbers 1024–65535)
- Stop its own thread
- Read standard system properties including the Java version and vendor and operating system.

In addition, most containers permit applets to make network connections back to their originating host.

P1: jth
JWBS001C-142.tex WL041/Bidgoli WL041-Bidgoli.cls September 16, 2005 9:8 Char Count= 0

Everything else such as file system operations, network sockets (except to the originating host), and audio is forbidden.

Enforcing Policies

The Java VM enforces policies on executions using the SecurityManager, which uses the AccessController class to check that code has the necessary permission before a controlled operation is executed. When a controlled operation is requested in Java, the call to the SecurityManager's checkPermission method simply calls on the AccessController and the AccessController grants or denies access according to the applicable security policy based on the code's protection domain, as determined from its associated ClassLoader. Security depends on the Java API calling the appropriate SecurityManager check method before every controlled operation and on the checking code associating the correct policy with the executing code.

When deciding to grant a permission to execute a requested action, the AccessController must determine which policy should apply to the request. Because code with different trust levels may be executing in the same Java VM, associating a policy with a request requires examining the stack to determine which code is responsible for the request. Every thread's stack consists of a set

Table 1 Java Permissions

Resource	Permission Class (Target)	Actions Controlled
General resources	AllPermission	All permissions. Granting AllPermission effectively turns off all access control security.
	SecurityPermission	Altering the security policy including setting the policy, setting security properties, and retrieving private keys.
	UnresolvedPermission	Used to represent permissions that are not resolved until run time (actual permission class does not exist when the policy is initialized).
	AuthPermission	Managing credentials and invoking code using a different identity
File System	FilePermission	Reading, writing, executing and deleting files.
Network	SocketPermission	Creating network sockets. Controls the ability to connect to specific hosts and ports, and to listen on local ports.
	NetPermission	Various network actions: control how authentication is done, stream handling, and requesting passwords.
	SSLPermission	Accessing SSL session contexts
Display	AWTPermission (showWindow-WithoutWarning, readDisplayPixels)	Creating pop-up windows that are not marked with a warning that indicates that they were created by an untrusted program; examine pixels on the display.
System Clipboard	AWTPermission (accessClipboard)	Reading from and writing to the system clipboard.
Keyboard, Mouse	AWTPermission (listenToAllAWTEvents, accessEventQueue, createRobot)	Examining, altering, and creating events in the system event queue.
System properties	PropertyPermission	Reading and writing system properties (environment variables)
Speaker, Microphone	AudioPermission	Playing and recording sounds
Printer	RuntimePermission (queuePrintJob)	Sending jobs to the printer
Application-Specific Resources	SQLPermission	Accessing the SQL log
	ServicePermission, Delegation Permission	Using and delegating Kerberos services
	PrivateCredentialPermission	Accessing private credentials associated with a specific subject
Java-Specific Resources	LoggingPermission	Altering system logging levels
	ReflectPermission	Using Java reflection to directly access fields and methods in a class. (Allows code to access private methods and fields.)
	RuntimePermission	Creating class loaders, setting class loader contexts, changing the security manager, altering threads, dynamic loading
	SerializablePermission	Alter the way objects are serialized by overriding the serialization methods.

P1: jth
JWBS001C-142.tex WL041/Bidgoli WL041-Bidgoli.cls September 16, 2005 9:8 Char Count= 0

of stack frames built by a sequence of method calls. The AccessController must not only verify that the current stack frame has the required permission, but also that the previously invoked stack frames have the permission. In this way, previously called methods cannot gain privileges by calling higher privileged code.

When the AccessController performs a security check, it examines the call stack to determine which protection domain should apply and then checks if it grants the appropriate permissions. This is known variously as *stack introspection* (Wallach, Balfanz, Dean, & Felten, 1997), *stack inspection* (McGraw & Felten, 1999) and *stack walking*. Because every Java method belongs to a class, and there is a protection domain associated with every class, each stack frame has an associated protection domain. A frame may also have dynamically granted permissions. If any stack frame has not been granted the permission for the requested access, then the request will be denied by throwing an exception. The AccessController checks permissions by calling a method to indirectly return an object encapsulating the current protection domains on the stack and then checking the associated permissions.

MALICIOUS BEHAVIOR

The simplest strategy for an attacker is to attempt to achieve the attack goals without violating the security policy. If the attacker's goal is just to obtain free computing resources or annoy the victim, then it is possible to do so without circumventing any security mechanisms. This section will consider hostile applets that have negative consequences without needing to violate typical security policies.

Exploiting Weak Policies

The challenge in defining a good security policy is to disallow all undesirable behavior while permitting all useful behavior. It is impossible to define a policy that does this exactly—all policies must either allow some undesirable behavior, disallow some useful behavior, or both. This is especially problematic when a generic, one-size-fits-all policy is applied to all applets, regardless of their intended purpose. For example, it would be appropriate (and necessary) for a chat room applet to send data entered by the user over the network but it may be unacceptable for a mortgage calculator applet to do so.

The default policy for typical Java implementations allows untrusted applets to perform many potentially detrimental actions. For example, standard security policies permit untrusted applets to make any number of network connections to the originating host and transmit any amount of data to the network. Because very few users are likely to customize their policy settings, a hostile applet that does these things without violating the default security policy is likely to succeed in victimizing most users who execute it.

Consuming Resources

Java's standard security manager supports only absolute permissions: an action is either permitted or not; if the action is permitted it is always permitted. This means that Java security policies provide no constraints on resource consumption, beyond what the underlying operating system provides. On typical consumer operating systems, such as Windows XP, there are no per-process resource constraints, so an untrusted Java applet could effectively consume all available system resources. Although we know of no cases in which a malicious resource consumption attack was discovered in the wild, one can imagine hostile applets designed to consume excessive amounts of almost any machine resource.

An applet that enters an infinite loop will continue to consume processor cycles as long as it executes. How damaging this is depends on the operating system's scheduler. On an operating system without preemptive multitasking (such as the Macintosh before OS X), a process that enters an infinite loop can prevent any other process from executing. With more recent operating systems and virtual machines, the processing time allotted to a given process is controlled by the operating system. Hence, even if a thread enters an infinite loop it will be preempted by the operating system and another thread will be given a chance to execute. Nevertheless, hostile applets can still consume substantial central processing unit (CPU) cycles on modern operating systems. An attacker can use the stolen CPU cycles to perform a computation valuable to the attacker on the victim's machine such as attempting to do a brute-force key search.

A thread that is set to the highest priority (MAX_PRIORITY) will be given all available CPU cycles by most schedulers. Java applets can also create new threads to consume more host resources. Normally, when a browser leaves a Web page, all the applets running on that page will be terminated. This is done by the containing application invoking each thread's stop method. However, since an applet can override the stop method (for example, to do nothing), it is possible for a hostile applet to continue executing after the browser leaves its page. Recent implementations (such as Netscape Navigator 7.1) mitigate this attack by forcing the threads to terminate after the page is left even if their stop method does not.

Hostile applets could also consume other system resources such as memory (by allocating objects in a loop and holding live references to them to prevent the garbage collector from reclaiming the storage) and the display (by creating windows to fill the screen). If the security policy permits the applet to open a file for writing, the applet may write as much data as it wants to the file, filling up the victim's disk. If any network connections are permitted, the applet can send data at the maximum rate possible and consume much of the victim's network bandwidth. Mark LaDue's collection of hostile applets provides examples of many different resource consumption attacks (LaDue, 2004).

Countermeasures

Because there are no permissions associated with consuming CPU cycles, creating threads, consuming memory, and creating windows (except for creating windows without warning markings), none of these attacks can be mitigated using standard Java security policies. Attacks

P1: jth
JWBS001C-142.tex WL041/Bidgoli WL041-Bidgoli.cls September 16, 2005 9:8 Char Count= 0

that consume network or file system resources can be prevented by prohibiting all network or file system access, but there is no way to define a policy that allows an applet to open a network connection without also allowing it to send unlimited amounts of data over that connection.

One strategy to limit the damage resource consumption attacks can accomplish is to use the underlying operating system to place limits on the total resources consumed by the Java VM. An operating system with a preemptive scheduler, such as Windows XP, will be less vulnerable to a CPU consumption attack because even if the hostile applet is able to consume all available cycles of the Java VM, it will not receive more system CPU cycles than the Java VM is allocated. Operating systems can also place limits on the memory, network bandwidth, and other resources a process may consume. This can prevent the hostile applet from interfering substantially with non-Java applications running on the host, but does not prevent it from interfering with the execution of other Java applets.

Java VM implementations could also monitor resource consumption and terminate applets that are exceeding set resource limits. Accounting for resource consumption by individual applets is not necessarily straightforward, however, because applets may interact in ways that make it difficult to account for which applet is responsible for a particular resource use. JRes is a resource accounting mechanism that can be implemented on top of a Java VM (Czajkowski & von Eicken, 1998). It accounts for CPU, memory, and network use by individual threads and allows a user to enforce a policy that takes actions when a thread exceeds usage limits.

Accounting for memory consumption is difficult because the thread that allocates an object may not be the thread responsible for it remaining in memory. Code from one applet may hold references to objects created by other applets, and a memory consumption attack could exploit this by holding on to references to objects created by another applet. Price, Rudys, and Wallach (2003) developed a technique for accounting for memory consumption by applets by modifying a garbage collector to measure the amount of storage attributable to each applet.

A more general defense strategy is to extend Java's security mechanisms to support more expressive policies. Java policies are limited by the defined permissions, but more fundamentally they are limited by the need to insert calls to SecurityManager checks in the Java API before every controlled operation. This means extending Java to support permissions associated with allocating memory, consuming processor cycles, reading or writing bits to a file (as opposed to opening a file for reading or writing), or transmitting data over the network (on a socket that is already open) would involve a substantial performance penalty. The costs associated with checking the permissions would be necessary for every operation, regardless of whether the policy in effect places any constraints on consumption of that resource. One solution to this is to use an *inlined reference monitor* (Erlingsson, 2003). This approach involves inserting checking code directly into an untrusted applet (or a policy-specific Java API). The inserted checking code performs the necessary security checks to enforce a policy before each security-critical action. Because the checking code is inserted to enforce a specific policy, it is possible to express a large class of policies that provide arbitrary constraints on use and consumption of any resource visible through code instructions. Two systems that adopt this approach to provide fine-grained policies on Java applets are Naccio (Evans & Twyman, 1999) and SASI (Erlingsson & Schneider, 1999).

CIRCUMVENTING POLICIES

This section describes several strategies a hostile applet may use to circumvent Java's security mechanisms. In McGraw and Felten's (1999) terminology, these are known as *attack applets*. The attacker's goal is to be able to perform some action that is not permitted by the safety policy. An attacker may be able to circumvent those mechanisms by finding a way to violate low-level code safety properties, by exploiting a vulnerability in the Java VM to make it associate the wrong policy with the executing code, or by exploiting a flaw in the Java API implementation that allows a critical resource to be manipulated without checking the appropriate permission.

Violating Low-Level Code Safety

A hostile applet can violate low-level code safety by exploiting a bug in the bytecode verifier that allows type-unsafe code to pass verification. At least 4 of the 31 known Java security bugs (13%) from February 1996 to December 2003 were due to bugs in the bytecode verifier that allows code that violates type safety to execute in a Java VM (Sun 2002b, 2004). These bugs often involve mistakes in implementing the verification of complex instructions. For example, a bug in the Java bytecode verifier found in 2001 involving the invokespecial instruction (Sun, 2002c) affected many implementations of the Java VM and could be exploited to violate type safety ("Last Stage of Delirium," 2002). Exploiting a bytecode verifier bug to achieve a hostile goal is generally possible if the attacker can obtain two references to the same object with different apparent types. The hostile applet can then proceed to access fields of the object through either reference. If the types of the fields of the first reference type and the second reference type are different, the attack can access arbitrary locations in memory by following references in the actual type that correspond to integers in the apparent.

A hostile applet may also circumvent Java's security mechanisms by violating control flow safety. Multiple bugs have been found within the implementation of the exception-handling mechanism of Java. One flaw in exception-handling subroutines was discovered in the Microsoft Java VM in 1999 ("Last Stage of Delirium," 2002). The jsr and ret instructions are used to implement finally clauses in Java. Control flow safety depends on the correct return addresses being on the stack when the ret instruction executes. A flaw in the bytecode verifier allowed an applet to use two jsr instructions to put two addresses on the stack and then to use a swap instruction to exchange the addresses (a correct verifier implementation would disallow this). Then, the swapped return address is used by the ret instruction to return to the instruction that is now referenced by the address. The verifier verifies the method as if the correct return was done. In this case,

P1: jth
JWBS001C-142.tex WL041/Bidgoli WL041-Bidgoli.cls September 16, 2005 9:8 Char Count= 0

violating control flow safety leads to the ability to violate type safety because after the switched return the stack can contain values of different types than is expected by the verifier's analysis.

Another strategy for violating low-level code safety is to break one of the assumptions the verifier relies on. For example, bytecode verification for type safety assumes that values in memory cannot be altered except through Java instructions. This seems like a reasonable assumption because all memory allocated to the applet is controlled by the Java VM process, so as long as the operating system provides virtual memory it should be impossible for another process to alter values in the memory accessible to an executing applet. However, it is possible for this assumption to be violated if bits in memory flip due to faulty hardware. Govindavajhala and Appel (2003) invented an attack based on violating this assumption. By filling memory with objects of a particular type, they were able to create a situation in which a random bit flip had a high probability (about 70%) of being exploitable by a hostile applet to violate type safety. After type safety is violated in this way, the hostile applet can change the value stored in arbitrary locations in memory. For example, an attacker could replace the reference to the current security manager with null, thereby circumventing all successive policy checking. Random bit flips can be induced in typical memory hardware simply by heating the memory chips (for example, with a light bulb or hair dryer). This requires physical access to the host machine, which may not be likely for remote desktop attacks, but is a serious concern for Java VMs that attempt to execute isolated applets on smart cards to which a potential attacker readily has physical access.

Policy Association

A hostile applet can obtain permissions beyond its trust level if it can confuse the Java VM into assigning it to the wrong protection domain. When a class is loaded, the protection domain is assigned based on the apparent origin of the code, its signers, and the principal executing the code. If a hostile applet can either alter the ClassLoader associated with a particular class or forge the origin or signers of a class, it can prevent the Java VM from associating the appropriate policy with the code.

Vulnerabilities in Java's class loading mechanism have been fairly common. Four of the 31 known Java security bugs are directly attributable to ClassLoader issues (Sun 2002b, 2004). Java 1.0 assumed that all code loaded from a trusted path (set by the CLASSPATH environment variable) was fully trusted and obtained all permissions. The Java 2 platform treats code loaded on the CLASSPATH as any other application, but uses the bootclasspath to identify fully trusted code. This is necessary to bootstrap the class loader, but poses the risk that an attacker who can store a file on the bootclasspath can circumvent all access controls.

The ClassLoader is also responsible for ensuring that there are never two different classes loaded with the same name. If this property is violated, low-level code safety properties can be violated because the two classes will type check according to their matching names, but may be implemented differently.

Security Checking

Security checking happens when Java API methods call SecurityManager check methods before performing critical operations. A hostile applet may be able to exploit mistakes in the way the Java API calls those methods or in the way the checks are implemented to circumvent and intended security policy. Types of possible flaws in Java security implementations include allowing access to a protected resource indirectly without the necessary security checking (for example, an applet can read a protected file by instead loading a font with a peculiarly constructed name), race conditions that allow system changes to occur between the time a check is made and the protected resource is used (for example, a file is checked and then replaced with a symbolic link before it is opened), or checks that make incorrect assumptions about resources.

One example of an exploitable security checking flaw was the Java domain name sytem (DNS) bug discovered by Drew Dean, Ed Felten, and Dan Wallach (1996). The security policy permitted an applet to open network connections to its originating host. However, connections were checked based on the DNS name, not the Internet protocol (IP) address. Netscape Navigator 2.0's Java implementation would use DNS to lookup a list of IP addresses corresponding to the originating host and a list of IP addresses corresponding to the host the applet is attempting to connect to and would allow the connection if there are any common IP addresses between the two lists. An attacker who can create bogus DNS mappings can then exploit this flaw to connect to arbitrary network hosts.

Defenses

The best defense against hostile applets that exploit vulnerabilities in the Java VM implementation is to obtain a Java VM implementation with no bugs. Of course, producing bug-free code is beyond our current capabilities, so it is worth considering techniques that can mitigate the damage a hostile applet can produce if it successfully circumvents Java's security mechanisms. Below, we describe five approaches.

Virus Scanners. Traditional virus scanners analyze untrusted programs to see if they contain strings that match a database of known hostile programs. Although commercial virus scanners focus on Windows platform exploits, most do include some hostile Java applets in their virus database (McAfee, 2004; Symantec, 2004). The string-matching approach works against known threats, but provides little protection against new attacks.

Malicious Behavior Detectors. To detect attacks from unknown threats, the system must observe the behavior and identify behavior that is likely to be malicious. Because several Java applets may be running concurrently in one Java VM, it is awkward to apply standard intrusion detection techniques to detect anomalous behavior of Java applets. The actions caused by a particular applet are not clear, because applets may interact through shared data structures and multiple threads. Soman, Krintz, and Vigna (2003) proposed a thread-level auditing facility for a Java VM that enables precise accounting for actions and detection of malicious behavior from Java applets. Note,

P1: jth
JWBS001C-142.tex WL041/Bidgoli WL041-Bidgoli.cls September 16, 2005 9:8 Char Count= 0

however, that this assumes the hostile applet is not able to violate type safety, because an attack applet that could do so could also interfere with the auditing mechanisms.

Firewalls. Firewalls monitor network traffic and can prevent harmful incoming packets from reaching system applications and harmful outgoing packets from reaching the network. Firewalls can prevent hostile applets from behaving harmfully in the same way they would prevent other applications from doing so. A firewall can also be designed to prevent potentially hostile Java applets from executing by blocking Java applets when they arrive on the network (Martin, Rajagopalan, & Rubin, 1997).

Isolation. Another approach is to execute possibly hostile applets in an isolated environment. Malkhi, Reiter, and Rubin (1998) propose running untrusted applets on a dedicated machine. The interface to the applet will appear in the users' browser in a way that provides users with the illusion that it is running on their own machine. Because the applet is executing on a separate machine and has only limited access to the outside world through the network, it cannot carry out any hostile actions on the user's machine. If a dedicated machine is not available, similar security properties can be achieved by executing untrusted applets in a way that isolates their effects. Liang, Venkatakrishnan, and Sekar (2003) describe a system that allows untrusted programs to execute in an environment in which all changes they make to the system are recorded. When the execution completes, the user can inspect the changes and decide whether to approve them.

Proof-Carrying Code. The size and complexity of the Java VM make implementing a correct Java VM difficult, so one approach to improving security is to reduce the complexity of the trusted computing base. Proof-carrying code attempts to do that by using a small and simple verifier to check a proof that is included with an untrusted program (Necula, 1997; Necula & Lee, 1996). Because it is easier to check a proof than to create one, the trusted computing base can be reduced by requiring programs to provide a proof that they satisfy required security properties. Automatically producing proofs of complex security policies and representing proofs in a condensed way remain challenging research problems, however.

CONCLUSION

Although numerous hostile applets have been proposed by security researchers, hostile applets are very rare in the wild. Reports of intentionally hostile Java applets are rare and only minor incidents have been reported. Compared to the damage caused by e-mail worms, viruses that exploit buffer overflows in Windows and common server applications, and cross-site scripting attacks, the actual damage caused by hostile Java applets is miniscule. Symantec's security response site reports 44 threats discovered in January 2004, none of which involved Java. Their entire database includes only two Java applet attacks that have been found in the wild: Java.Nocheat and Trojan.ByteVerify. Both are exploit applets that exploited a vulnerability in the Java VM included with Microsoft Internet Explorer (Microsoft, 2003). No significant damage was caused by either attack, and fewer than 50 infections were known.

The lack of instances of actual Java applet attacks is not terribly surprising given the motivations and technical capabilities of most malicious attackers. E-mail worms are comparatively very easy to write and far more effective in causing damage; buffer overflow attacks require a bit more sophistication, but can be created by nonexperts using widely available tools and can readily give the attacker complete control over the victim's machine. By contrast, most Java exploits depend on subtle flaws in the bytecode verifier or class-loading mechanisms, which are both harder to identify and often difficult to exploit even after the flaw is identified. As a result, most of the work on finding vulnerabilities in Java has been done by nonmalicious researchers interested in improving the security of the platform, not by malicious attackers interested in causing harm.

Java's security mechanisms are certainly not perfect, and there are many ways a malicious applet can cause harm. Some of these exploit vulnerabilities in Java implementations to violate low-level code safety and enable the attacker to circumvent the security mechanisms; others work within the security mechanisms, but exploit weak policies that provide insufficient limits on resource consumption and access. Ongoing work in industry and academic research labs is developing techniques for efficiently enforcing precise policies that can control resource consumption, accurately account for resources consumed by applets, and execute untrusted programs in protected environments. As with most security issues, understanding new attacks leads to new work on defensive countermeasures, and new defensive countermeasures lead to new approaches to attacks.

GLOSSARY

Applet Small program intended for execution inside a container (such as a Web browser).

Control Flow Safety Property of a platform or programming language that ensures that attempts to jump to instruction addresses always jump to valid locations.

Denial-of-Service Attack Attack intended to prevent legitimate users from accessing a resource.

Dynamic Checking Analysis done on program executions.

Java Platform Platform that includes the Java virtual machine intended for executing programs written in JVML.

Java virtual machine langauge (JVML) Stack-based intermediate language.

Low-Level Code Safety Properties necessary to prevent circumvention of high-level security mechanisms. Comprises type safety, memory safety, and control flow safety.

Malicious Code Code created with the intention of causing harm to someone who executes it.

Memory Safety Property of a platform or programming language that ensures that attempts to read and write to memory are to valid locations.

Safety Policy Set of rules that specify behavior that is permitted. If the safety policy is enforced correctly,

P1: jth
JWBS001C-142.tex WL041/Bidgoli WL041-Bidgoli.cls September 16, 2005 9:8 Char Count= 0

programs are prevented from actions that are not permitted by the policy.

Static Checking Analysis done by examining programs directly without executing them.

Type Safety Property of a platform or programming language that ensures that values of a particular type can only be used with operations that expect values of that type. In particular, type safety prevents forging pointers.

Virtual Machine A program that provides an abstract platform for executing programs to enable portability, simplicity, and security. The Java virtual machine interprets programs written in JVML.

CROSS REFERENCES

See *Computer Viruses and Worms; Hackers, Crackers and Computer Criminals; Hoax Viruses and Virus Alerts; Mobile Code and Security; Spyware; Trojan Horse Programs*.

REFERENCES

Aleph One. (1996). Smashing the stack for fun and profit. *Phrack Magazine, 7*(49). Retrieved from http://www.insecure.org/stf/smashstack.txt

Chen, Z. (2000). *Java card technology for smart cards: Architecture and programmer's guide*. Reading, MA: Addison-Wesley.

Czajkowski, G., & von Eicken, T. (1998, October). *JRes: A resource accounting interface for Java*. Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications.

Dean, D., Felten, E., & Wallach, D. (1996, May). *Java security: From HotJava to Netscape and beyond*. IEEE Symposium on Security and Privacy, Oakland, CA.

Erlingsson, Ú. (2003). *The inlined reference monitor approach to security policy enforcement* (Technical Report 2003-1916). Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY.

Erlingsson, Ú., & Schneider, F. B. (1999, September). *SASI enforcement of security policies: A retrospective*. Proceedings of the New Security Paradigms Workshop.

Evans, D., & Twyman, A. (1999, May). *Policy-directed code safety*. Proceedings of the IEEE Symposium on Security and Privacy.

Gong, L., Ellison, G., & Dageforde, M. (2003). *Inside Java 2 platform security: Architecture, API design, and implementation* (2nd ed.) Reading, MA: Addison-Wesley.

Gosling, J. (1995, February). *Java: An overview* (Sun Microsystems White Paper). Santa Clara, CA: Sun Microsystems.

Govindavajhala, S., & Appel, A. (2003, May). Using memory errors to attack a virtual machine. *IEEE Symposium on Security and Privacy*.

LaDue, M. (2004). *A collection of increasingly hostile applets*. Retrieved from http://www.cigital.com/hostile-applets/

Last Stage of Delirium Research Group. (2002, October). *Java and virtual machine security: Vulnerabilities and their exploitation techniques*. Retrieved from http://www.lsd-pl.net/documents/javasecurity-1.0.0.pdf

Liang, Z., Venkatakrishnan, V. N. & Sekar, R. (2003, December). *Isolated program execution: An application transparent approach for executing untrusted programs*. 19th Annual Computer Security Applications Conference.

Lindholm, T., & Yellin, F. (1999). *The Java virtual machine specification* (2nd ed.). Reading, MA: Addison-Wesley.

Malkhi, D., Reiter, M., & Rubin, A. (1998, May). *Secure execution of Java applets using a remote playground*. Proceedings of the 1998 IEEE Symposium on Security and Privacy.

Martin, D. M., Jr., Rajagopalan, S., & Rubin, A. (1997). *Blocking Java applets at the firewall*. Internet Society Symposium on Network and Distributed Systems Security.

McAfee Security. (2004, January). *Virus information library*. Retrieved from http://us.mcafee.com/virusInfo/

McGraw, G., & Felten, E. (1999). *Securing Java: Getting down to business with mobile code*. New York: Wiley.

Microsoft. (2003, June 27). *Flaw in Microsoft VM could enable system compromise* (Microsoft Security Bulletin MS03-011). Redmond, WA: Author.

Necula, G. (1997). *Proof-carrying code*. Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

Necula, G., & Lee, P. (1996). *Safe kernel extensions without run-time checking*. Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation.

Price, D. W., & Rudys, A., & Wallach, D. S. (2003). *Garbage collector memory accounting in language-based systems*. 2003 IEEE Symposium on Security and Privacy, Oakland, CA.

Singh, I., Stearns, B., Johnson, M., & the Enterprise Team. (2002). *Designing enterprise applications with the J2EE platform*. Reading, MA: Addison-Wesley.

Soman, S., Krintz, C., & Vigna, G. (2003). *Detecting malicious Java code using virtual machine auditing*. Proceedings of the 12th USENIX Security Symposium.

Sun Microsystems. (2002a). *Permissions in the Java 2 SDK*. Retrieved from http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html

Sun Microsystems. (2002b, November 19). *Chronology of security-related bugs and issues*. Retrieved from http://java.sun.com/sfaq/chronology.html

Sun Microsystems. (2002c). *Sun security bulletins article 218*. Retrieved from http://sunsolve.sun.com/pub-cgi/retrive.pl?doc=secbull/218

Sun Microsystems. (2003). *Java 2 platform, standard edition: 1.4.2 API specification*. Retrieved from http://java.sun.com/j2se/1.4.2/docs/api/

Sun Microsystems. (2004, January). *Sun alert notifications*. Retrieved from http://sunsolve.sun.com/pub-cgi/search.pl, category:security java

Symantec Corporation. (2004, January). *Virus threats page*. Retrieved from http://www.symantec.com/avcenter/vinfodb.html

Wallach, D. S., Balfanz, D., Dean, D., & Felten, E. W. (1997, October). *Extensible security architectures for Java*. Proceedings of the 16th Symposium on Operating Systems Principles.