20th Annual Computer Security Applications Conference (ACSAC).
December 6-10, 2004, Tucson, Arizona.

.NET Security: Lessons Learned and Missed from Java

Nathanael Paul David Evans
University of Virginia
Department of Computer Science
[*nate, evans*]*@cs.virginia.edu*

Abstract

Many systems execute untrusted programs in virtual machines (VMs) to limit their access to system resources. Sun introduced the Java VM in 1995, primarily intended as a lightweight platform for execution of untrusted code inside web pages. More recently, Microsoft developed the .NET platform with similar goals. Both platforms share many design and implementation properties, but there are key differences between Java and .NET that have an impact on their security. This paper examines how .NET's design avoids vulnerabilities and limitations discovered in Java and discusses lessons learned (and missed) from Java's experience with security.

1. Introduction

Java and .NET are both platforms for executing untrusted programs with security restrictions. Each platform uses a virtual machine to enforce policies on executing programs as depicted in Figure 1.

The term Java is used to refer to both a high-level programming language and a platform. We use Java to refer to the platform consisting of everything used to execute the Java class containing Java virtual machine language code (JVML, also known as "Java bytecodes") in the left part of Figure 1 except the operating system and the protected resource. A Java archive file (JAR) encapsulates Java classes and may also contain other resources such as a digital signature or pictures. Java was designed primarily to provide a trusted environment for executing small programs embedded in web pages known as applets.

The .NET platform includes the .NET part of the figure involved in executing an assembly except for the operating system and the protected resource. A .NET assembly, analogous to Java's JAR file, is an executable or dynamically linked library containing Microsoft intermediate language instructions (MSIL), some metadata about the assembly, and some optional resources. .NET differentiates between managed (safe) and unmanaged (unsafe) code. Since a security policy cannot be enforced on unmanaged code, we only consider managed code.

Both Java and .NET have large trusted computing bases (TCBs) allowing many possible points of failure. The TCB includes everything in Figure 1 except for the external untrusted program (the Java class or .NET assembly). In Java, a flaw in the bytecode verifier, class loader, JVM or underlying operating system can be exploited to violate security properties. With .NET, a flaw in the policy manager, class loader, JIT verifier, CLR, or underlying operating system can be exploited to violate security properties. The size of the TCB makes it infeasible to make formal claims about the overall security of either platform; instead, we can analyze individual components using the assumption that other components (in particular, the underlying operating system) behave correctly.

The JVML or MSIL code may be generated by a compiler from source code written in a high-level program such as Java or C#, but these files can be created in other ways. Although high-level programming languages may provide certain security properties, there is no way to ensure that delivered JVML or MSIL code was generated from source code in a particular language with a trusted compiler. Hence, the only security provided against untrusted code is what the platform provides. This paper does not consider the relative merits of the Java and C#

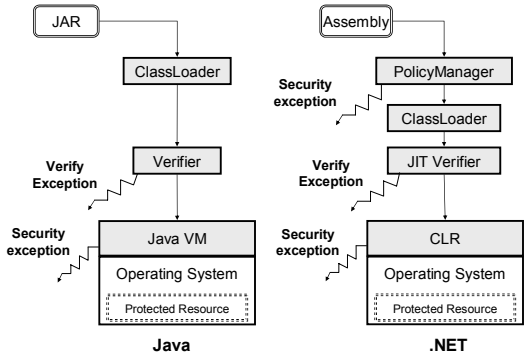

Figure 1. Architecture Overview

Table 1. Java Security Vulnerabilities.
Vulnerabilities reported in Java platform in CVE database [3] and Sun's web site [30, 33].
Instances of the form CVE-YEAR-NUMB are CVE entries; CAN-YEAR-NUMB are CVE candidates.

Category	Count	Instances
API bugs	10	CVE-2000-0676, CVE-2000-0711, CAN-2000-0563, CVE-2002-0865, CVE-2002-0866, CVE-2002-1260, CAN-2002-1293, CAN-2002-1290, CAN-2002-1288, CAN-2002-0979
Verification	9	http://java.sun.com/sfaq/chronology.html (4), CVE-1999-0141, CVE-1999-0440, CVE-2000-0327, CVE-2002-0076, CAN-2003-0111
Class loading	9	http://java.sun.com/sfaq/chronology.html (5), CAN-2000-1117, CVE-2002-1287, CAN-2003-0896, CAN-2004-0723
Other or unknown	5	CVE-1999-0766, CVE-2001-1008, CVE-2002-1257, CAN-2002-1286, CVE-2002-1325
Html tags	4	CAN-2001-0068, CAN-2002-1258, CAN-2002-1295, CAN-2002-1291
Missing policy checks	2	CVE-1999-0142, CVE-1999-1262
Configuration	2	CVE-1999-0162, CAN-2002-0058
DoS attacks (crash)	4	CVE-2002-0867, CAN-2002-1289, CAN-2003-0525, CAN-2004-0651
DoS attacks (consumption)	2	CAN-2002-1292, http://sunsolve.sun.com/pub-cgi/search.pl (alert 57555)

programming languages but only compares the security properties of the two execution platforms.

Since the Java platform was introduced in 1995, Java's security model has evolved to incorporate additional security mechanisms including code signing and increasingly flexible policies. When specific implementation issues are considered, we address the current standard implementations of each platform: the Java 2 Software Development Kit 1.4.2 and the .NET Framework 1.1.

Previous work, including Pilipchuk's article [24], have compared security mechanisms and features in Java and .NET from an operational perspective. In this paper, we consider how they differ from the perspective of what has and has not been learned from experience with Java security vulnerabilities. Table 1 summarizes security vulnerabilities reported in Java over the past 8 years. Hopwood [10], Princeton's Secure Internet Programming team [4, 35, 36] and McGraw and Felten [18] identified several vulnerabilities in early Java implementations.

The general lessons to be learned from experience with Java are not new. All of them go back at least to Saltzer and Schroeder's classic paper [25], and none should be surprising to security analysts. In particular: economy of mechanism, least privilege, and fail-safe defaults are design principles that enhance security, but can often conflict with other goals including usability and complexity. The concrete experience with Java shows how failure to apply these well known principles has lead to vulnerabilities in a specific, security-critical system. The primary contributions of this paper are: 1) an illustration of how the history of Java security vulnerabilities reveal failures to follow established security principles; 2) an identification of how .NET's security mechanisms have addressed the vulnerabilities and limitations of Java; and 3) a discussion on how differences in the design of .NET and Java are likely to impact their security properties.

Both Java and .NET use a combination of static analysis and dynamic checking to enforce policies on executing programs. The bytecode verifier in Java and the just-in-time (JIT) verifier in .NET statically verify some low-level code properties necessary (but not sufficient) for type safety, memory safety and control flow safety before allowing programs to execute. Other properties must be checked dynamically to ensure low-level code safety. Section 2 describes how Java and .NET guarantee low-level code safety properties. Five of the 42 Java platform security vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database [3], and four of the earlier vulnerabilities [30], are directly attributed to flaws in implementations of the Java bytecode verifier.

Programs that pass the verifier are executed in the Java virtual machine (JVM) or .NET Common Language Runtime (CLR). Both virtual machines use a reference monitor to mediate access to protected system resources. Section 3 describes how policies are defined in Java and .NET. Section 4 details how a particular policy is associated with a resource action. Section 5 describes how the JVM and CLR enforce policies on executions.

2. Low-Level Code Safety

Low-level code safety comprises the properties of code that make it type, memory, and control-flow safe. Without these properties, applications could circumvent nearly all high-level security mechanisms [37]. The primary lesson learned from Java's experience with low-level code safety goes back to one of the earliest security principles: keep things simple.

Type safety ensures that objects of a given type will be used in a way that is appropriate for that type. In particular, type safety prevents a non-pointer from being dereferenced to access memory. Without type safety, a program could construct an integer value that corresponds to a target address, and then use it as a pointer to reference an arbitrary location in memory. Memory safety ensures that a program cannot access memory outside of properly allocated objects. Buffer overflow attacks violate memory safety by overwriting other data by writing beyond the allocated storage [1]. Control safety ensures that all jumps are to valid addresses. Without control safety, a program could jump directly to system code fragments or injected code, thereby bypassing security checks.

Java and .NET achieve low-level code safety through static verification and run-time checks. In typical Java implementations, static verification is done by the Java bytecode verifier at load time. An entire class is verified before it is executed in the virtual machine. In .NET, parts of the verification are done as part of the JIT compilation. All code must pass the verifier, however, before it is permitted to execute.

2.1 Verification

The first step in the verification process is the validation of the file format of the code [5, 17]. The file is checked according to the Java class file or .NET PE/COFF file specifications [17, 20]. Following the verification of the file format, the verifier also checks some static rules to ensure that the objects, methods, and classes are well formed.

Next, the verifier simulates each instruction along each potential execution path to check for type and other violations. Since JVML and MSIL are stack-based languages, executions are simulated by modeling the state of the stack while tracking information about each instruction to help ensure low-level code safety. Verification fails if a type violation could occur, or a stack operation could cause underflow or overflow. In addition, control flow safety is ensured by checking that all branch instructions target valid locations.

The general problem of verifying type safety is undecidable [23], so certain assumptions must be made to make verification tractable. Both verifiers are conservative: if a program passes verification it is guaranteed to satisfy prescribed safety properties; however, programs exist that are type safe but fail verification. A more sophisticated verifier could accept more of the safe programs (still rejecting all unsafe programs), but increasing the complexity of the verifier is likely to introduce additional vulnerabilities.

Code passing the verifier is permitted to run in the virtual machine, but additional runtime checks are needed that could not be checked statically. Runtime checks are required to ensure that array stores and fetches are within the allocated bounds, elements stored into array have the correct type (because of covariant typing of arrays in both JVML and MSIL this cannot be checked statically [2]), and down cast objects are of the correct type.

A bug in the Java bytecode verifier or Microsoft's JIT verifier can be exploited by a hostile program to circumvent all security measures, so complexity in the verifier should be avoided whenever possible.

The JVML and MSIL verifiers are both relatively small, but complex, programs. Sun's 1.4.2 verifier [11] is 4077 lines of code (not including code for checking the file format). For .NET, we examined Rotor, the shared source code that is a beta version of Microsoft's implementation of the ECMA CLI standard [28]. The JIT verifier in the production .NET release is either very similar or identical to the Rotor verifier [16]. Rotor's integrated verifier and JIT compiler total about 9400 lines, roughly 4300 of which are needed for verification.

2.2 Instruction Sets

Since the verifier's complexity is directly tied to the instruction set of the virtual machine, examining the instruction sets provides some measure of the verifier's complexity. Each platform uses about 200 opcodes, but some important differences in their instruction sets impact the complexity of their verifiers. This section considers the differences between the JVML and MSIL instruction sets from the perspective of how complex it is to verify low-level code safety properties.

Table 2 summarizes the instruction sets for each platform. One obvious difference between the instruction sets is that JVML has separate versions of instructions for each type, whereas .NET uses a single instruction to perform the same operation on different types. For example, Java has four different add instructions depending on the type of the data (`iadd` adds integers, `fadd` adds floats, etc.) where .NET has one instruction that works on different types. Using generic instructions to perform an operation with

multiple types instead of just two types makes verification slightly more difficult, but means that .NET has more instruction opcodes available for other purposes. .NET uses some of these instructions to provide overflow and unsigned versions of the arithmetic operations. The overflow versions of arithmetic operations throw exceptions if the calculation overflows, enabling applications to better handle overflows and avoid security vulnerabilities related to arithmetic overflows (such as the Snort TCP Stream Reassembly Integer Overflow Vulnerability reported in [26]).

Function Calls. Complex, multi-purpose instructions further increase verification complexity. For example, the invokespecial instruction in JVML serves three purposes: calling a superclass method, invoking a private method, and invoking an initialization method. The multiple uses of this instruction make it difficult to verify correctly. Sun's verifier uses 260 lines to verify the invokespecial instruction (counting major methods used for verification). A 2001 verifier bug involving the invokespecial instruction [31] affected many implementations of the JVM, and could be exploited to violate type safety [14].

.NET has two main instructions for calling methods: call and callvirt (another MSIL calling instruction, calli, is used for calling functions indirectly through a pointer to native code). The call instruction is similar to Java's invokespecial and invokestatic instructions. The callvirt instruction is similar to Java's invokeinterface and invokevirtual instructions. The main difference between the call and callvirt instructions is how the target address is computed. The address of a call is known at link-time while callvirt determines the method to call based on the runtime type of the calling object. Combining Java's four different calling instructions into two instructions may make it easier for a compiler writer [19], but given Java's history of trouble it may have been better to have several single-purpose call instructions rather than a few instructions with multiple functions. The call, and callvirt instructions each have their own method for JIT compilation and verification totaling approximately 200 lines in the Rotor implementation.

To efficiently support tail recursion, the MSIL call instructions may also be preceded by a tail prefix which is treated as a special case by the verifier [5]. The tail prefix reuses the same activation record on the stack instead of creating a new record every time a call is made. About 250 extra lines are required for verification and compilation of the tail prefix including the extra lines needed to deal with call, calli, and callvirt. It is too soon to judge whether the performance advantages of supporting tail outweigh the additional security risks associated with the added complexity.

Object Creation. Sometimes complex instructions are better than using many separate instructions. For example, a Java program creates a new object by using new to allocate memory for the new object, dup to place an additional reference to the newly created object on the stack, and then invokespecial to call the object's initializing constructor. After returning from the constructor, a reference to the (now initialized) object is on top of the stack because of dup. In MSIL, the single newobj instruction calls a constructor, creating and initializing a new object in one step. This sacrifices flexibility, but verification of newobj is much easier than Java's sequence of instructions since the verifier knows that the object is initialized as soon the instruction is executed.

A Java verifier must check that any new object is initialized before use [15]. In cases where the new, dup and invokespecial instructions are separated by instructions, this can pose problems for the verifier. Microsoft and Netscape's Java verifiers have both had vulnerabilities related to improper object initialization. The Microsoft verifier bug involved calling a constructor within an exception handler inside a child

Table 2. Instruction Sets Comparison.

	JVML		MSIL	
Type	Number	Examples	Number	Examples
arithmetic	36	iadd, fadd, ladd, iand	21	add, add_ovf, xor
stack	11	pop, dup2, swap	2	pop, dup
compare	21	ifeq, ifnull, if_icmpeq	29	ceq, beq, brfalse
load	51	Ldc, iload, iaload	65	Ldarg, ldftn, ldstr
store	33	istore,lstore_1, castore	27	Starg, stloc_s, stelem_R8
type conversions	15	i2f, d2i, l2d	33	conv_i2, conv_ovf_u8, conv_u2
method calls	4	invokevirtual, invokestatic, invokespecial, invokeinterface	3	callvirt, call, calli
object creation	4	new, newarrary, anewarray, multianewarray	2	newobj, newarr
exceptions	3	athrow, jsr, ret	5	leave, leave_s, rethrow, endfilter, endfinally

class [14]. Once the code called the constructor from inside the child class, the parent class constructor would be called to create a ClassLoader object, but the child class had not been given permission to instantiate a class loader. The resulting exception was caught by the exception handler in the constructor of the child class, and initialization was incorrectly assumed to have completed.

Exception Handling. Java's exception handling instructions impose additional complexity compared to MSIL's simpler approach. The JVML instruction jsr is used to implement the Java programming language try-finally construct that transfers execution to a finally block [17] and is one of the most complex instructions to verify. To jump to a finally block, control transfers to an offset from the address of the jsr instruction, and the return address of the next instruction after the jsr instruction is pushed onto the stack. The main problem is the use of the operand stack to store the return address since this makes an attractive target for an attacker who may try to insert a different address while fooling the verifier. With the return address on the operand stack, more difficulty exists in a finally block's verification in the multiple ways one could execute a finally block: a jsr called after execution of the try clause, a jsr used upon a break/continue within the try clause, or a return executed within the try block.

Several vulnerabilities have been found in Java verifiers due to the complexity of the jsr instruction. One relating to subroutines in exception handling was found in 1999 in the Microsoft JVM [14]. To exploit this flaw, two return addresses are placed on top of the stack using different jsr instructions. Next, a swap instruction is executed. The verifier failed to account for the change of return addresses on the stack (ignoring the swap since the return addresses are of the same type). The switched return address is used by the ret instruction to return to the instruction that is now referenced by the address. The verifier continues to verify the method as if the swap had not executed, thus breaking type safety.

.NET avoids the complexity associated with Java's jsr instruction by providing a simpler instruction. The leave instruction used to exit a try or catch clause clears the operand stack and uses information stored in an exception handling clause for control flow.

Summary. We tested .NET to check that the verifier was behaving correctly according to the ECMA specification and attempted to carry out exploits that have previously worked on the Java verifier, but were unable to construct any successful exploits. Of course, this does not mean that there are no exploitable bugs in the .NET platform, but it is encouraging that none have been reported to date. .NET's designers avoided many of the pitfalls in early Java implementations benefiting from Java's history of problems with exception handling, creating objects, and calling methods. The MSIL instruction set design simplifies the verification process by avoiding instructions similar to the most complex instructions to verify in JVML.

3. Defining Policies

Low-level code safety mechanisms prevent hostile applets from circumventing the high-level code safety mechanisms, but security depends on high-level mechanisms to enforce a policy on program executions. A policy specifies what actions code may perform. If a program attempts an action contrary to the policy, a security exception is raised.

3.1 Permissions

The amount of control possible over system resources depends on the available permissions. Except for those permissions that are platform specific, Java and .NET provide similar permissions for controlling access to the file system, network, display, system properties and clipboard [21, 29]. For details on the differences, see [22]. Neither supports complete mediation: only actions associated with a predefined permission are checked. Further, there is no support to restrict the amount of a resource that is consumed, so many denial-of-service attacks are possible without circumventing the security policy. These limitations are serious [12], but more complete mediation is possible through the reference monitoring framework only by significantly reducing performance. Richer policy expression and efficient enforcement is an active research area [6, 7, 34].

3.2 Policies

Policies associate sets of permissions with executions. In Java, policies are defined by specifying the permissions granted in a policy file based on properties of an execution: the origin of the code, the digital code signers (if any), and the principal executing the code. Java's policies are also affected by a system-wide properties file, java.security, which specifies paths to other policy files, a source of randomness for the random number generator, and other important properties.

A Java policy file contains a list of grant entries. Each entry specifies a context that determines when the grant applies and lists a set of granted permissions in

that context. The context may specify the code signers (a list of names, all of whom signed the code for the context to apply), the code origin (code base URL), and one or more principals (on whose behalf the code is executing). If no principals are listed, the context applies to all principals.

Java is installed with one system-wide policy file, but a user can augment this policy with her own policy file. The granted permission set is the union of the permissions granted in all the policy files. This is dangerous since it means more permissions are granted than those that appear in the user's policy file. Further, it means a user can make the policy less restrictive than the system policy, but cannot make the policy more restrictive. Java users may not exclude permissions a system administrator allows unless they are able to edit java.security, the Policy implementation, or the policy file granting the unwanted permissions.

.NET provides policy definition mechanisms that overcome these limitations by providing flexible, multi-level policies, but at the cost of greater complexity. A .NET policy is specified by a group of policy levels: Enterprise (intended for the system administrator), Machine (machine administrator), User, and Application Domain (AppDomain). The permissions granted to an assembly are the intersection of the permissions granted at the four policy levels. .NET's policies grant permissions based on *evidences* within an assembly (see Section 4.2). The AppDomain policy is created at run-time, and there is no associated configuration file for this policy level. If no AppDomain exists at run-time, then the policy is the intersection of the Enterprise, Machine, and User policy levels. .NET's policy levels are similar to Java having a system-wide policy file and a user policy file, however they are much more flexible. Importantly, in .NET the final permission set granted is the intersection of all policy levels, whereas in Java it is the union [9].

Typical users will execute code found on untrusted web sites, so the Internet default policy is extremely important to protect users and resources. Java's default policy allows an untrusted process to read some environment properties (e.g., JVM version, Java vendor), stop its own threads, listen to unprivileged ports, and connect to the originating host. All other controlled actions, such as file I/O, opening sockets (except to the originating host), and audio operations are forbidden. The default Java policy disallows the most security critical operations, but does not prevent untrusted applets from annoying the user. Many examples of disruptive applets exist, such as one that stops and kills all current and future applets and another one that consumes the CPU [12, 18].

The .NET default permissions are given by the intersection of the four policy levels expressed in three separate files (AppDomains exist only at runtime). At runtime, the CLR looks for the three XML policy files representing the Enterprise, Machine, and User policy levels. By default, .NET allows all code to have all the permissions in the Enterprise and User policy levels, and the Machine policy level's granted permissions determines the resulting permission set. The default policy grants permissions based on the zone evidence. Local code is given full trust along with any strong-named Microsoft or ECMA assemblies. Code from the local intranet is granted many permissions including printing, code execution, asserting granted permissions (see Section 5.2), and reading the username. Internet assemblies are given the Internet permission set which includes the ability to connect to the originating host, execute (itself), open file dialogs, print through a restricted dialog box, and use its own clipboard. The trusted zone will receive the Internet permission set. No permissions are granted to the restricted zone. These defaults are more consistent with the principle of fail-safe defaults than Java's defaults. But their strictness may encourage users to assign too many code sources to more trusted zones.

4. Associating Policies with Code

Since programs with different trust levels may run in the same VM, VMs need secure mechanisms for determining which policy should be enforced for each access to a controlled resource. The ability to assign different policies to different code within the same VM follows the principle of least privilege: every module (class or assembly) can be assigned the minimum permissions needed to do its job. Section 4.1 explains how granted permissions are associated with code. Section 4.2 describes how code properties determine which policy should be applied. There are important differences in how Java and .NET accomplish this. Java's initial design was a simple model where code was either completely trusted or untrusted, and all untrusted code ran with the same permissions. Later versions of Java extended this model, but were constrained by the need to maintain backwards compatibility with aspects of the original design. .NET was designed with a richer security model in mind from the start, so it incorporates an extensible policy mechanism in a consistent way.

4.1 Code Permissions

Both Java and .NET support two types of permissions: static and dynamic. Static permissions

are known and granted at load time. Dynamic permissions are unknown until runtime.

When Java loads a class, an instance 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 runtime 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 principal 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, since their set of permissions will be the same. Prior to J2SE 1.4, permissions were assigned statically at load time by default, but dynamic security permissions have been supported since J2SE 1.4 [32]. This provides more flexibility, but increases complexity and makes reasoning about security policies difficult.

To assign static permissions at load time in Java, a class loader will assign permissions to a PD based on properties of the code and its source, and the loaded class will be associated with that single PD for the duration of the class' lifetime [32, 17]. Several flaws have been reported in Java's class loading mechanisms, including eight documented from [33] and [3] (see Table 1).

.NET uses a similar approach to associate permission sets with assemblies. The role of the ClassLoader in Java is divided between the PolicyManager and ClassLoader in .NET. The PolicyManager first resolves the granted permission set [13, p. 173-175]. Then the CLR stores the permissions in a cached runtime object before passing the code on to the ClassLoader which loads the class.

4.2 Code Attributes

Both Java and .NET grant permissions based on attributes of the executing code. The expressiveness of policies is limited by the code attributes used to determine which permissions to grant.

The JavaVM examines the CodeSource and Principal and grants permissions based on the values found in these objects. The CodeSource is used to determine the location or origin of the code and signing certificates (if used), and the Principal represents the entity executing the code. The associated PD of a class encapsulates these objects along with the ClassLoader and static permissions granted at load time. To extend the default policy implementation, the Policy class may need to be rewritten, or a different SecurityManager may need to be implemented. It is questionable if this level of extensibility is actually a good idea—it introduces significant security risks, but the benefits in practice are unclear. Problems with class loading were found in early Java implementations [4], and continue to plague Java today. In one recent classloader vulnerability (CAN-2003-0896 in Table 1), arbitrary code could be executed by skipping a call to a SecurityManager method. The corresponding code characteristics in .NET are known as *evidences*. .NET's PolicyManager uses two types of evidences, host evidences and assembly evidences, to determine the permissions granted to an assembly. Assembly evidences are ignored by default. Evidences include the site of origin, zone (corresponding to Internet Explorer zones), publisher (X.509 certificate) and strong name (a cryptographic code signature). .NET's design incorporates the ability to extend not only the permissions that may be granted, but also to add new evidences as well. Any serializable class can be used as evidence [8].

Java and .NET both provide complex policy resolution mechanisms and a bug in the policy resolution could open a significant security hole. There are difficult issues to consider in introducing new permissions including XML serialization, and declarative/imperative testing of a new permission (see Section 5) [13, p. 534-544]. Although .NET does not provide the same level of extensibility as Java in the policy implementation, a developer creating a new permission must still be careful to avoid errors.

4.3 Bootstrapping

Both platforms need some way of bootstrapping to install the initial classes and loading mechanisms. Java 1.0 used a trusted file path that gave full trust to any class stored on the path. Code on the system CLASSPATH was fully trusted, so problems occurred when untrusted code could be installed on the CLASSPATH [10]. Java 2 treats code found on the CLASSPATH as any other code, but maintains backwards compatibility by using the bootclasspath to identify completely trusted code necessary to bootstrap the class loader. Hence, the same risks identified with installing untrustworthy code on the CLASSPATH now apply to the bootclasspath. Having exceptions based on the location of code is not a good idea, since an attacker who can modify the trusted path or trick a web browser into storing code in a location on the trusted path will be able to execute a program with full permissions.

.NET uses full-trust assemblies to break the recursive loading of policies since all referenced assemblies must also be loaded [13, p. 112]. .NET did not completely abandon the notion of a trusted path, but it has added some security. .NET uses a global

assembly cache (GAC) where assemblies in this cache are signed and then shared among different assemblies. The GAC acts as a trusted repository, similar to the bootclasspath in that an assembly within the GAC will be fully trusted [21]. To speed up loading, a GAC assembly's strong name (or signature) is checked when the assembly is added to the GAC, not when the assembly is loaded. If an attacker can modify an assembly in the GAC, then the attacker may have full control of the machine. Sometimes fully trusted assemblies across all policy levels are needed; for example, the default assemblies used for policy resolution that are fully trusted by default.

As an illustration, the .NET default policy trusts all signed Microsoft assemblies, and this is checked by examining the strong name evidence of each assembly. If all four policy levels fully trust signed Microsoft assemblies, then any assembly from Microsoft is fully trusted on that machine.

5. Enforcement

Policy enforcement is chiefly done at run-time by the virtual machine. Unlike Java, .NET can perform some policy enforcement statically. It allows the programmer to specify static or dynamic policy enforcement. *Declarative* security permissions are statically known and contained within the assembly manifest. *Imperative* security permissions are compiled to MSIL and evaluated at run-time. The declarative permissions can be class-wide or method-wide and can be used for some actions that cannot be expressed using imperative permissions. When run-time information is needed to evaluate a request (e.g., a filename), imperative permissions must be used.

Run-time enforcement mechanisms share many similarities across the two platforms. In Java, the SecurityManager checks code permissions. Programmers can implement SecurityManager subtypes to customize security checking, and programs with sufficient permission can change the security manager. This makes it especially easy to exploit a type safety break in Java, since the security manager can be set to null to turn off all access control. .NET's design does not allow programmers to implement their own SecurityManager class, but the reduced flexibility provides stronger security.

5.1 Checking Permissions

When a Java program attempts a restricted operation, the called Java API method first calls the SecurityManager's appropriate checkPermission method which calls the AccessController to determine if the necessary permission is granted. When deciding to grant a permission to execute a requested action, the AccessController checks that the current executing thread has the needed permission.

The 10 API bugs in Table 1 illustrate the difficulty in implementing permission checks correctly. Many of these vulnerabilities involve an API method that allows access to a protected resource without the necessary security checks. CVE-2000-0676 and CVE-2000-0711 both bypass calls using SecurityManager by exploiting the java.net.ServerSocket and netscape.net.URLInputStream classes. Another flaw, CAN-2000-0563, used browser redirection to gain sensitive data in java.net.URLConnection. Two vulnerabilities, CAN-2002-0866 and CAN-2002-1260, involve bugs in the Java Database Connectivity (JDBC) classes with the former allowing an attacker to execute any local Dynamic Link Library (DLL) through a JDBC constructor and the latter allowing access to a database through a JDBC API call. CAN-2002-1290 and CAN-2002-1293 were bugs in Microsoft's JVM that exposed interfaces to the INativeServices and CabCracker classes allowing access to the clipboard or local file system respectively. CAN-2002-0865, CAN-2002-0979 and CAN-2002-1288 exposed various resources including XML interfaces, logging, and directory information.

Java's AccessController must not only verify that the current stack frame has the required permission, but also that the calling stack frames do. In this way, previously called methods cannot gain privileges by calling higher privileged code. Since every method belongs to a class and a class to a PD, each stack frame's permissions are checked through the associated PD in addition to any dynamic permissions granted by the policy. 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 accomplishes permission checks by calling a method to indirectly return an object encapsulating the current PDs on the stack (i.e., *current context*) and then checking those PDs' permissions. The act of gathering the current permissions from each stack frame is called a *stack walk*.

.NET performs a similar stack walk with Frame objects representing the frames on the stack. To support multiple languages (including type unsafe languages like C++), the stack has frames that are *managed* and *unmanaged*. The managed frames are frames that are verified for type safety while the unmanaged frames have no safety guarantees. As the stack is traversed, the managed code's permissions are checked with a security object contained in each JIT-compiled method on the stack [27].

5.2 Modifying the Stack Walk

In both platforms, programmers can modify the stack walk. This should be done to enforce the principle of least privilege by explicitly denying permissions to called methods.

A Java program can modify the stack walk to deny certain permissions past a specific stack frame or to simply stop checking permissions at a specific point. If a method invokes doPrivileged (PrivilegedAction), the stack walk will not look at any frames further up the call stack. Attacks have occurred where the caller gains access to some protected resource by calling code that has higher privileges which indirectly provides access to that resource (for example, CAN-2002-1288). To deny permissions to a method in Java, a method can invoke doPrivileged (PrivilegedAction, AccessControlContext). This creates a new context that is the same as the stack's current execution context without the denied permissions. The stack walk will then use this context to check permissions. However, using doPrivileged can cause problems when null is passed as the AccessControlContext object. This removes the stack frame from any more security decisions and introduces scoping problems when implemented with an inner class [18, 29].

.NET has extended Java's stack walk design with the Permission methods PermitOnly(), Assert(), and Deny(). A stack walk is done when a demand() call is made, similar to Java's checkPermission(). .NET provides slightly better interfaces for the programmer to alter the stack walk since many of the mechanisms involve only one method call after constructing the specified permissions. Calling the PermitOnly() method means a stack walk will continue only if the permission is granted. After a Deny() call, if any of the specified permissions are requested an exception is thrown to terminate the stack walk. Assert() terminates the stack walk successfully if the current stack frame has the asserted permission.

Although stack inspection is complex in both models, .NET's added flexibility using these new Permission methods can be used to help programmers improve security by writing code that does not expose protected resources unnecessarily.

6. Conclusion

Java and .NET have similar security goals and mechanisms. .NET's design benefited from past experience with Java. Examples of this cleaner design include the MSIL instruction set, code access security evidences, and the policy configuration. .NET has been able to shield the developer from some of the complexity through their new architecture.

Where Java evolved from an initial platform with limited security capabilities, .NET incorporated more security capability into its original design. With age and new features, much of the legacy code of Java still remains for backwards compatibility including the possibility of a null SecurityManager, and the absolute trust of classes on the bootclasspath. Hence, in several areas .NET has security advantages over Java because of its simpler and cleaner design.

Most of the lessons to learn from Java's vulnerabilities echo Saltzer and Schroeder's classic principles, especially economy of mechanism, least privilege and fail-safe defaults. Of course, Java's designers were aware of these principles, even though in hindsight it seems clear there were occasions where they could (and should) have been followed more closely than they were. Some areas of design present conflicts between security and other design goals including fail-safe defaults vs. usability and least privilege vs. usability and complexity. For example, the initial stack walk introduced in Java has evolved to a more complex stack walk in both architectures to enable developers limit privileges. In addition, both platforms default policies could be more restrictive to improve security, but restrictive policies hinder the execution of programs. .NET's use of multi-level policies with multiple principals provides another example of showing the principles of least privilege and fail-safe defaults in contention with usability and complexity.

Several of the specific complexities that proved to be problematic in Java have been avoided in the .NET design, although .NET introduced new complexities of its own. Despite .NET's design certainly not being perfect, it does provide encouraging evidence that system designers can learn from past security vulnerabilities and develop more secure systems. We have no doubts, however, that system designers will continue to relearn these principles for many years to come.

Acknowledgements

This work was funded in part by the National Science Foundation (through grants NSF CAREER CCR-0092945 and NSF ITR EIA-0205327) and DARPA (SRS FA8750-04-2-0246). The authors thank Jane Prey, Elizabeth Strunk for help with the title, and the anonymous reviewers for their helpful comments.

References

[1] AlephOne. *Smashing the stack for fun and Profit*. Phrack, 7(49), Nov. 1996.

[2] W. R. Cook. *A Proposal for Making Eiffel Type-safe*. Third European Conference on Object-Oriented Programming (ECOOP). July 1989.

[3] Common Vulnerabilities and Exposures. *Java Vulnerability Search Results (version 20040901)*. 1 September 2004. http://www.cve.mitre.org/

[4] Drew Dean, Edward W. Felten, and Dan S. Wallach. *Java security: From HotJava to Netscape and Beyond*. IEEE Symposium on Security and Privacy. May 1996.

[5] ECMA International. *Standard ECMA-335: Common Language Infrastructure* (Second Edition) December 2002. http://www.ecma-international.org/publications/standards/Ecma-335.htm

[6] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement***.** Ph.D. thesis, Cornell University Department of Computer Science. (Technical Report 2003-1916). 2003.

[7] David Evans and Andrew Twyman. *Policy-Directed Code Safety*. IEEE Symposium on Security and Privacy. May 1999.

[8] Adam Freeman and Alan Jones. *Programming .NET Security*. O'Reilly, June 2003.

[9] Li Gong, Gary Ellison and Mary Dageforde. *Inside Java 2 Platform Security (Second Edition)*. Sun Microsystems, June 2003.

[10] David Hopwood. *Java Security Bug (applets can load native methods)*. Risks Forum, March 1996.

[11] IBM Corporation. *Jikes Research Virtual Machine*. http://www-124.ibm.com/developerworks/ oss/jikesrvm/

[12] Mark LaDue. *A Collection of Increasingly Hostile Applets*. http://www.cigital.com/hostile-applets/

[13] Brian A. LaMacchia, Sebastian Lange, Matthew Lyons, Rudi Martin, Kevin T. Price. *.NET Framework Security*. Addison-Wesley, April 2002.

[14] Last Stage of Delirium Research Group. *Java and Virtual Machine Security Vulnerabilities and their Exploitation Techniques*. http://www.lsd-pl.net/documents/javasecurity-1.0.0.pdf

[15] Xavier Leroy. *Java Bytecode Verification: An Overview*. Springer Verlag Computer Aided Verification: 2101, pp. 265-285, 2001.

[16] Mark Lewin. Email communication, Jan. 2004.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, 2nd edition*. Addison-Wesley, April 1999.

[18] Gary McGraw and Edward W. Felten. *Securing Java*. John Wiley and Sons, January 1999.

[19] Erik Meijer and John Gough. *Technical Overview of the Common Language Runtime*. http://research.microsoft.com/~emeijer/Papers/CLR.pdf

[20] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification*. http://www.microsoft.com/whdc/hwdev/download/hardware/pecoff.pdf

[21] Microsoft Corporation. *Security Briefs: Strong Names and Security in the .NET Framework*. http://msdn.microsoft.com/netframework/?pull=/library/en-us/dnnetsec/html/strongNames.asp

[22] Nathanael Paul and David Evans. *.NET Security: Lessons Learned and Missed from Java* (extended version of this paper). UVA Computer Science Technical Report, September 2004.

[23] Benjamin C. Pierce. *Bounded quantification is undecidable*. ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), January 1992.

[24] Denis Pilipchuk. *Java vs. .NET Security*. http://www.onjava.com/pub/a/onjava/2003/11/26/javavsdotnet.html

[25] Jerome Saltzer and Michael Schroeder. *The Protection of Information in Computer Systems.* Fourth ACM Symposium on Operating System Principles, October 1973. (Revised version in Communications of the ACM, July 1974.

[26] *Snort TCP Stream Reassembly Integer Overflow Vulnerability*. http://www.securityfocus.com/advisories/5294

[27] Daivd Stutz, Ted Neward and Geoff Shilling. *Shared Source CLI Essentials*. O'Reilly, March 2003.

[28] David Stutz. *The Microsoft Shared Source CLI Implementation*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Dndotnet/html/mssharsourcecli.asp

[29] Sun Microsystems. *Permissions in the Java 2 SDK*. http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html

[30] Sun Microsystems. *Chronology of Security-Related Bugs and Issues*. November 2002. http://java.sun.com/sfaq/chronology.html

[31] Sun Microsystems. *Sun Security Bulletins Article 218*. http://sunsolve.com/pub-cgi/retrieve.pl?doctype=coll&doc= secbull/218&type=0&nav=sec.sba

[32] Sun Microsystems. *Java 2 Platform, Standard Edition: 1.4.2 API Specification*. 2003. http://java.sun.com/j2se/1.4.2/docs/api/

[33] Sun Microsystems. *Sun Alert Notifications*. http://sunsolve.sun.com/pub-cgi/search.pl,category: security java

[34] David Walker. *A Type System for Expressive Security Policies*. ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), January 2000.

[35] Dan Wallach, Dirk Balfanz, Drew Dean, Edward Felten. *Extensible Security Architectures for Java*. Symposium on Operating Systems Principles, October 1997.

[36] Dan Wallach and Edward Felten. *Understanding Java Stack Inspection*. IEEE Symposium on Security and Privacy, May 1998.

[37] Frank Yellin. *Low Level Security in Java*. 4th International WWW Conference, December 1995.