available at www.sciencedirect.com

SCIENCE DIRECT®

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Comparing Java and .NET security: Lessons learned and missed

*Nathanael Paul**, *David Evans**

University of Virginia, Department of Computer Science, VA, USA

ARTICLE INFO

Article history:
Received 24 February 2005
Revised 27 December 2005
Accepted 6 February 2006

Keywords:
Virtual machine security
Java security
.NET security
Security design principles
Bytecode verifier
Malicious code
Code safety

ABSTRACT

Many systems execute untrusted programs in virtual machines (VMs) to mediate their access to system resources. Sun introduced the Java VM in 1995, primarily intended as a lightweight platform for executing 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 experience with Java security.

1. Introduction

Java and .NET are both platforms for executing untrusted programs with security restrictions. Although they share similar goals and their designs are similar in most respects, there appear to be significant differences in the likelihood of security vulnerabilities in the two platforms.

Fig. 1 shows the number of major security vulnerabilities reported for each platform. As of December 2005, the Common Vulnerabilities and Exposures (CVE) database contains 150 entries concerning Java vulnerabilities (Mitre Corporation, Common Vulnerabilities), 38 of which we classify as major Java platform security vulnerabilities (we do not include application-specific bugs unrelated to the VM itself). The remaining vulnerabilities included in Fig. 1 but not in the CVE are from Sun (Sun Microsystems, 2002) (9 vulnerabilities) and McGraw and Felten (1999) (5 vulnerabilities). The contrast with the .NET platform, in which no major security vulnerabilities have yet been found, appears striking. This paper considers whether or not the difference in the number of security vulnerabilities found in the two platforms stems from fundamental differences in their designs.

Table 1 summarizes Java security vulnerabilities reported in the past 10 years. Hopwood (1996), Princeton's Secure Internet Programming team (Dean et al., 1996; Wallach et al., 1997; Wallach and Felten, 1998) and McGraw and Felten (1999) identified several vulnerabilities in early Java implementations. The rest are those documented in Sun's chronology (Sun Microsystems, 2002; Sun Microsystems Sun Alert) and the CVE database (Mitre Corporation, Common Vulnerabilities).

By contrast, no security vulnerabilities in the .NET virtual machine platform have been reported to date. The most widely publicized security issue in .NET was W32.Donut, a virus that took control of the executable before the .NET runtime had

* *Corresponding authors.*
E-mail addresses: nate@cs.virginia.edu (N. Paul), evans@cs.virginia.edu (D. Evans).
0167-4048/$ – see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2006.02.003

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Comparing Java and .NET security: Lessons learned and missed	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Virginia,Department of Computer Science,151 Engineer's Way,Charlottesville,VA,22904-4740**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**13**	

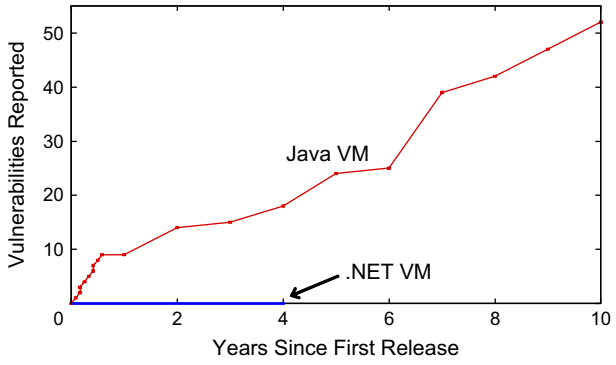

Fig. 1 – Major security vulnerabilities reported. The value plotted is the cumulative number of major security vulnerabilities reported in each platform since the first official release (Java 1.0 in January 1996 (Sun Microsystems, Java); .NET 1.0 in January 2002 (Microsoft Corporation, Technology Overview)).

control (Szor). Since the vulnerability occurs before the .NET runtime takes control, we consider this a problem with the way the operating system transfers control to .NET and not with the .NET platform. Eight other security issues that have been identified in the .NET are listed in Microsoft's Knowledge Base (Farkas) and the CVE database (Mitre Corporation, Common Vulnerabilities), but none of them are platform security vulnerabilities by the standard we use in this paper. Appendix A explains these issues and why we do not count them.

There are many possible explanations for the .NET platform's apparent lack of security vulnerabilities. One possibility is that .NET is a less desirable platform for attackers to compromise than Java so it has not received the scrutiny necessary to reveal vulnerabilities. This is unlikely, however, since the .NET framework is now provided as a Windows update. Since Windows has over 90% of the desktop market with a large number of machines using .NET, the .NET platform presents an attractive target.

Another possibility is that more vulnerabilities have been found in Java implementations because there are several different Java VM implementations whereas .NET's number is from Microsoft's sole implementation. From the available information, the one implementation that did have many of its own unique vulnerabilities was Microsoft's Java implementation, and this is largely due, in part, to 10 vulnerabilities reported in November 2002 by Pynnonen. As early as March 1996, both Microsoft and Netscape had licensed Java, two months after the 1.0 release date (Sun Microsystems, Java). As Java licensees, both Microsoft and Netscape implementations are based on the Sun implementation (Sun Microsystems, 2002) so much of the code and design are shared across the three implementations. The first 9 reported Java vulnerabilities did affect all three implementations, including the first 5 of 13 total verifier vulnerabilities. Although popular open source .NET platform implementations exist, such as Mono, and dotGNU, neither has fully implemented code access security to enable the execution of partially trusted code.

Another possibility is that .NET just avoided the specific security vulnerabilities that were already known because of previous experience with Java. This may be true in a few cases, but in general it is not the case. There are enough differences between the platforms that most security vulnerabilities would not have a direct analog. Further, vulnerabilities continue to be found in new versions of Java even after .NET's release.

In this paper we explore the more optimistic hypothesis that .NET's design is fundamentally more secure than Java's, and in particular, that it benefits from following general security principles that have been learned and reinforced from experience with Java. The general lessons to be learned from experience with Java are not new. Most of them go back at least to Saltzer and Schroeder's (1973) classic paper, 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. Other lists of security principles, including Viega and McGraw's (2001), include similar properties such as defense in depth and securing the weakest link. Viega and McGraw emphasize that security principles should be followed within an application's context and following these universal security principles allows a programmer to weigh different design

Table 1 – Java security vulnerabilities

Category	Count	Instances
API bugs	12	CVE-2000-0676, CVE-2000-0711, CVE-2000-0563, CVE-2002-0865, CVE-2002-0866, CVE-2002-1260, CVE-2002-1293, CVE-2002-1290, CVE-2002-1288, CVE-2002-0979, CVE-2005-3905, CVE-2005-3906
Verification	13	Sun Chronology (4), McGraw and Felten (2), CVE-1999-0766, CVE-1999-0141, CVE-1999-0440, CVE-2000-0327, CVE-2002-0076, CVE-2003-0111, CVE-2004-2627
Class loading	8	Sun Chronology (5), CVE-2002-1287, CVE-2003-0896,[a] CVE-2004-0723
Other or unknown	3	CVE-2001-1008, CVE-2002-1325, CVE-2005-3907
Missing policy checks	3	CVE-1999-0142, CVE-1999-1262, McGraw and Felten (1)
Configuration	5	CVE-2000-0162, CVE-2002-0058, CVE-2005-0471, McGraw and Felten (2)
DoS attacks (crash)	4	CVE-2002-0867, CVE-2002-1289, CVE-2003-0525, CVE-2004-0651
DoS attacks (consumption)	4	CVE-2002-1292, CVE-2004-2540, CVE-2005-3583, CVE-2004-1503

Vulnerabilities reported in Java platform in CVE database (Mitre Corporation, Common Vulnerabilities), Sun's web site (Sun Microsystems, 2002; Sun Microsystems, Sun Alert), and McGraw and Felten (1999).
Vulnerabilities reported in more than one source were counted once.
[a] Revealed under keyword search for JVM vulnerabilities instead of Java vulnerabilities.

trade-offs while preparing for unknown attacks that may not fit past attack patterns (Viega and McGraw, 2001). The concrete experience with Java shows how failure to apply these well known principles has lead to vulnerabilities in a particular, security-critical system.

Previous work, including Pilipchuk's article, has 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. The primary contributions of this paper are as follows: (1) an illustration of how the history of Java security vulnerabilities reveals 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.

Next, we provide an overview of both platforms. Section 3 describes low-level code safety highlighting the importance of simplicity. Section 4 examines policy definition and permissions emphasizing the principles of fail-safe defaults, least privilege, and complete mediation. Associating policies with code according to the code attributes is discussed in Section 5. Next, Section 6 describes how the JVM and CLR enforce policies on executions evaluating them in their application of the principles of least privilege, fail-safe defaults, and complete mediation. Section 7 discusses the shortcomings of both platforms with respect to psychological acceptability.

2. Platform overview

Both Java and .NET use a virtual machine to enforce policies on executing programs as depicted in Fig. 2. 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 Fig. 1 except the operating system and the protected resource. A Java archive (JAR) file 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.

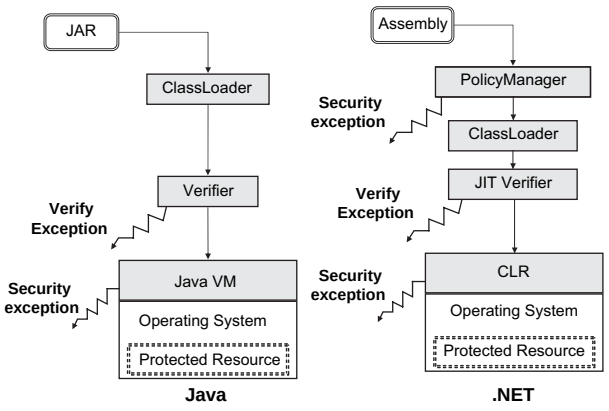

Fig. 2 – Architecture overview.

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) codes. 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 Fig. 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 the 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# 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.

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 3 describes the principle of simplicity in low-level code safety properties. Six of the 30 Java platform security vulnerabilities in the Common Vulnerabilities and Exposures database (Mitre Corporation, Common Vulnerabilities), and 6 of the earlier vulnerabilities (McGraw and Felten, 1999; Sun Microsystems, 2002) 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.

3. 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 (Yellin, 1995). The primary lesson learnt 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 (AlephOne, 1996). 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 runtime 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.

3.1. *Verification*

The first step in the verification process is the validation of the file format of the code (ECMA International, 2002; Lindholm and Yellin, 1999). The file is checked according to the Java class file or .NET PE/COFF file specifications (Lindholm and Yellin, 1999; Microsoft Corporation, Microsoft Portable Executable). 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 (Pierce, 1992), 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 (Cook, 1989)), 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 (Sun Microsystems, Java 2 SDK) 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 (Stutz). The JIT verifier in the production .NET release is either very similar or identical to the Rotor verifier (Lewin, 2004). Rotor's integrated verifier and JIT compiler total about 9400 lines, roughly 4300 of which are needed for verification.

3.2. *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 on 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 Core Security Technologies Advisory).

3.2.1. *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

Table 2 – Instruction sets comparison

Type	JVML		MSIL	
	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`
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`

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 (Sun Microsystems, Sun Security) affected many implementations of the JVM, and could be exploited to violate type safety (Last Stage of Delirium Research Group).

.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 (Meijer and Gough), 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 (ECMA International, 2002). 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.

3.2.2. *Object creation*

Sometimes a complex single instruction is 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 whether any new object is initialized before use (Leroy, 2001). Java's verifier has difficulty with two areas in object creation. In cases where the `new`, `dup` and `invokespecial` instructions are separated by instructions, this can pose problems for the verifier. The second problematic area is the complexity of the `invokespecial` instruction. Microsoft and Netscape's Java verifiers have both had vulnerabilities relating to improper object initialization. The Microsoft verifier bug involved calling a constructor within an exception handler inside a child class (Last Stage of Delirium Research Group). 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 the initialization was incorrectly assumed to have completed.

3.2.3. *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 (Lindholm and Yellin, 1999) 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 (Last Stage of Delirium Research Group). 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.

Recently, Sun has announced a radical redesign of its bytecode verifier. The new verifier that is part of the new Java SE Mustang release will have two very important simplifications: the separation of the type inferencing process and the removal of any code that generates a jsr or ret instruction (Sun Microsystems, JSR 202; Sun Microsystems, New Java SE). The verifier can now use type information embedded in the class file represented as a code attribute instead of having to infer the type information. To ease this transition to the new verifier, the verification process reverts to using the old verifier if a class file is not recognized as a newer version 50.0 class file (Sun Microsystems, New Java SE).

Disabling the JVM from running older class files can be done by passing a flag to the JVM. This design can break backward compatibility for the benefit of the simplicity of verification, but users should have more confidence in the security of the Java verifier. This is an encouraging step towards simplicity, in contrast to nearly all of the modifications to the Java platform since 1995 that increased complexity.

3.2.4. *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 verifier, but it is encouraging that no verifier bugs 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.

4. 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.

4.1. *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 (Microsoft Corporation, Security Briefs; Sun Microsystems, Permissions). The permissions provided by each platform are summarized in Table 3.

The platforms differ in which resources are protected by permissions, and in the granularity of control over specific operations and resources the available permissions provide. In general, .NET and Java protect the same set of resources with permissions, except for platform-specific resources. For instance, Java must provide permissions that protect some resources that are not exposed in .NET including the `SecurityManager` and `AccessControlContext` (see Section 6.2). Although .NET has a registry permission to protect the Windows registry, Java does not expose this resource through their API by default. Similarly, Java has an `AudioPermission` for restricting access to audio system resources, but .NET's API provides no access to audio resource. Microsoft's DirectX 9.0 SDK provides access to audio resources, and adds the `SoundPermission` to control access to those resources.

In general, .NET provides permissions with finer granularity of control. For example, both platforms provide permissions to restrict access to the file system. But, whereas in Java the same permission controls deleting, writing to, and appending to files, in .NET it is possible to provide just append access to a file.

Table 3 reveals that both platforms suffer from the lack of a systematic design in their permissions. Many of these permissions are based on protecting methods provided by the platform API, rather than protecting security-critical resources. For example, Java's `SQLPermission` allows a program to set the logging stream that may contain private SQL data; it is checked before `setLogWriter` methods in several classes. Java's `AWTPermission.createRobot` permission protects `java.awt.Robot` objects that allow the creation of low-level mouse and keyboard events. .NET's `PerformanceCounterPermission` protects diagnostic information exposed by the API. These permissions do not correspond well to security properties a user could relate to, but rather correspond to dangerous API methods.

Designing permissions around API methods rather than security-critical resources is dangerous since it means granting a permission may provide unexpected capabilities. For example, Java's `ReflectPermission` indirectly allows a program to access private methods and fields; effectively, this allows a program to circumvent other security checks and is equivalent to granting most other permissions. .NET provides a similar `ReflectionPermission`, but allows a finer granularity of control over what can be accessed. Other permissions provided by Java that effectively grant code arbitrary access include `FilePermission.execute` (which allows a program to execute a system command) and `SecurityPermission.setPolicy` (which allows a program to change the security policy).

Knowing the resulting policy after granting certain permissions is an area of difficulty common to both architectures. A

Table 3 – Permissions summary (Meijer and Gough; .NET Framework Developer's Guide)

Resource	Restricted operations	Java permissions	.NET permissions
File system	Read/write/execute/delete files	`FilePermission`	`FileIOPermission, SecurityPermission`
	Append access information on path itself	No separate permissions (append = write)	`FileIOPermissionAccess (Append, PathDiscovery)`
	Access data in current directory from executing program	Can read any file in current directory or sub-directory of current directory	`IsolatedStoragePermission`
Network	Accept/connect/listen/resolve a host at an optional port range	`SocketPermission`	`SocketPermission`
Display	Show an applet-created window without warning, restrict access to event queue	`AWTPermission`	Events handled differently without special permissions
	Controlling different properties of a window (e.g., setting caption, hiding cursor)	Not provided	`UIPermission.Window`
Reflection	Use of reflection	`ReflectPermission`	`ReflectionPermission`
	Reflection on visible or invisible members of a type	Level of control not provided (all or nothing)	`ReflectionPermission`, different flag values control the level of use
System clipboard	Read/write clipboard (all or nothing)	`AWTPermission`	`UIPermission.Clipboard`
	Read clipboard (write unrestricted)	Level of control not provided (all or nothing)	`UIPermission.Clipboard (OwnClipboard)`
Threading	Control threads	`RuntimePermission`	`SecurityPermission`
	Control any thread, code's own threads, control a group of threads	`RuntimePermission`, different target values control level of privileges	Threadpool provides safety through implementation
Database	Set the logging stream of SQL actions	`SQLPermission`	Logging can be configured through registry or provided tools (Meier et al.)
	Allow blank passwords for a database user, access databases	Specific database API not included by default	`OdbcPermission, OleDbPermission, OraclePermission, SqlClientPermission`
Printer	Print	`RuntimePermission`	`PrintingPermission`
	Printing to any printer (default only) and through restricted print dialog boxes	All or nothing restriction by `RuntimePermission`	`PrintingPermission`
Platform specific	Read/write/delete registry keys/values	`Resource not exposed (no permission needed)`	`RegistryPermission`

developer may not understand that granting file execute or reflection permissions to any code that asks for it is essentially the same as fully trusting the code. An attacker also has an additional method of attack to compromise a system if a permission can give higher privileges in a different way.

Neither platform supports complete mediation: only actions associated with an associated predefined permission are checked and many resources (for example, allocating memory) have no associated permission. 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 (LaDue), 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 (Erlingsson, 2003; Evans and Twyman, 1999; Walker, 2000).

4.2. *Policies*

Policies associate sets of permissions with executions. For security, policies should follow the principle of least privilege and fail-safe defaults, however, these principles often conflict with convenience and are not always followed.

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/his 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 5.2). The `AppDomain` policy is created at runtime, and there is no associated configuration file for this policy level. If no `AppDomain` exists at runtime, 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 principle of fail-safe defaults is followed by setting the final permission set to the intersection of all policy levels, whereas in Java it is the union.

Typical users will execute code found on untrusted web sites, so the Internet default policy is extremely important to protect users and resources. If the policy is too permissive, the granted privileges may be used to compromise the system. 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 the one that stops and kills all current and future applets and another one that consumes the CPU (LaDue; McGraw and Felten, 1999).

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 determine 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 6.2), 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 least privilege than Java's defaults. But their strictness may encourage users to assign too many code sources to more trusted zones.

5. 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 codes 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, but this added flexibility does cost additional complexity and decreased performance. Section 5.1 explains how granted permissions are associated with code. Section 5.2 describes how code properties determine which policy should be applied. There are important differences of 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.

5.1. *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 (Sun Microsystems, 2003). 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 (Sun Microsystems, 2003; Lindholm and Yellin, 1999). Several flaws have been reported in Java's class loading mechanisms, including 8 documented from Sun Microsystems, Sun Alert and Mitre Corporation, Common Vulnerabilities (see Table 1). It is important to note that these static permissions do not depend upon the dynamic permissions as specified in the Java policy files but rather depends on the class loader loading a class.

.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 (LaMacchia et al., 2002, p. 173–5). Then the CLR stores the permissions in a cached runtime object before passing the code onto the `ClassLoader` which loads the class.

5.2. *Code attributes*

Both Java and .NET grant permissions based on attributes of the executing code.

The Java VM 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 (Dean et al., 1996), and continue to plague Java today. In one recent classloader vulnerability (CVE-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 (Freeman and Jones, 2003).

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 6, LaMacchia et al., 2002, p. 534–44). Although .NET does not provide the same level of extensibility as Java in customizing security policy enforcement, a developer creating a new permission must still be careful to avoid errors.

5.3. *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 (Hopwood, 1996). 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 wise, 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 (LaMacchia et al., 2002, 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 (Microsoft Corporation, Security Briefs). If an attacker can successfully 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 is 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.

6. Enforcement

By allowing partially trusted code to execute, policy enforcement becomes more complicated. Policy enforcement is chiefly done at runtime by the virtual machine. Unlike Java, .NET can perform some policy enforcement statically by allowing 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 runtime. 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 runtime information is needed to evaluate a request (e.g., a filename), imperative permissions must be used.

Runtime enforcement mechanisms share many similarities across the two platforms. Both platforms implement a reference monitor designed to follow the principle of complete mediation by checking the necessary permissions before allowing any sensitive operation. 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.

6.1. *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 12 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,

CVE-2000-0563, used browser redirection to gain sensitive data in `java.net.URLConnection`. Two vulnerabilities, CVE-2002-0866 and CVE-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. CVE-2002-1290 and CVE-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. CVE-2002-0865, CVE-2002-0979 and CVE-2002-1288 exposed various resources including XML interfaces, logging, and directory information. The last API bugs, CVE-2005-3905 and CVE-2005-3906, are both related to errors in the Reflection API enabling an attacker to read and write local files or execute applications on the local machine.

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 call 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 (Stutz et al., 2003).

6.2. *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, but programmers must be careful to not allow more permissions when changing stack walk behavior.

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, CVE-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 introduce access modifier issues when implemented with an inner class (Gong et al., 2003; Sun Microsystems, Permissions).

.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.

7. Psychological acceptability

Saltzer and Shroeder (1973) identified ''Psychological Acceptability'' as their final security principle, and emphasized the importance of protection mechanisms fitting the user's protection goals. This principle is often overlooked (Clear, 2002), and challenging to follow even when it is considered, and Java and .NET are not exceptions.

Both VMs have extensible policies, but their policies are still difficult for typical users to configure and understand. Since the permissions do not clearly show what resources they may protect, the user may grant access to resources unintentionally. Even if a machine is properly configured, a user may be faced with a situation where the policy is violated, a security exception is raised, and the application terminates. In order to get the application to run, the user needs to understand what security violation happened, how to configure the machine to permit the security sensitive operation, and what security implications are there in granting the requested operation. If a certificate has been revoked or expired, exceptions will occur that a normal user will have trouble in understanding. Most likely, the user may grant full trust in both of these situations if the application is important enough. When a security exception or other similar exception occurs, more guidance is needed, so the user can take the correct action.

In Sun's Java Development Kit (JDK) 1.0 the security model treated all applets as untrusted and confined them to a limited, albeit inflexible, environment known as the Java *sandbox*. JDK 1.1 introduced signed applets, so the user could choose to execute the applet with full permissions based on the entity associated with the applet's signature. Just prior to the release of JDK 1.2, Microsoft and Netscape introduced a more flexible security model, the Java model in this paper, that allowed users to execute partially untrusted code with limited permissions. Unfortunately, Netscape's model (Netscape Capabilities Model) had drawbacks to its initial implementation. In the Netscape Capabilities Model, whenever an applet needs permission to access a protected resource, the user is presented

with a dialog box. Once the user grants the permission, the application can perform the requested operation until the user terminates Netscape Communicator (Netscape Communications Corporation). Although the user can click a button for further details, another dialog box is presented to help. If the user runs the applet again after restarting Netscape, then he/she again goes through the same process with the alternative options of denying permission, or he/she can permanently grant the requested permissions to the applet. Bombarding the user with dialog boxes that require a quick security decision to be made in order for execution to continue is a bad idea (McGraw and Felten, 1999). Luckily this behavior has changed in current Java security models, and security exceptions do not encourage the user to make a hasty decision.

Another problem area is the default permissions. The designers took steps to protect certain API functions, but it can be difficult to determine which permissions to grant by looking only at the permission (and not the resource). With a higher granularity of protection in some permissions, .NET helps the user to choose safe and usable permissions. For example, the clipboard permission is not a binary decision where the code has all or nothing access to a resource. Instead, code can have unlimited write access to the clipboard, but it cannot have read access. Since this permission model allows finer-grained protection, this allows the user to safely execute code while not having to grant full read/write access to the clipboard. Since a user only wants to protect his/her private data, this model conforms to the user's understanding. Another example is the window permissions that protect the user from fake dialog boxes and phishing attacks by enforcing restrictions on specific window components (e.g., Forms, DataGrids, and Cursors) (Microsoft Visual Studio). Users interact with GUI programs through these types of window components, so the user is better able to evaluate implications on security by granting access to these familiar resources.

8. 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 underlying 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`. However, Java is applying a learned lesson as it makes the verifier simpler (at a cost of compatibility). 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 Somesh Jha, Jane Prey, and Elizabeth Strunk for helpful comments on this paper.

Appendix A. .NET security issues

There have been 8 security issues identified in the .NET framework listed in Microsoft's Knowledge Base (Farkas), only one (KB327523) of which appears to be exploitable. Because this problem appears to be in an ASP.NET HTTP module's parsing of an HTML request (also included in the CVE database as CAN-2004-0847) and not in the .NET framework (Baier), we do not count this as a .NET platform security vulnerability. However, this is still a significant security vulnerability that could be exploited by an attacker to obtain arbitrary .aspx files from an ASP.NET web server. Notably, similar issues (CAN-2002-1258, CAN-2002-1295, CAN-2002-1291, CVE-2002-1257, CAN-2002-1286) appeared in parsing URLs and HTML content in Microsoft's Java implementation in the past (Clear, 2002) (these are not included in the count of Java security vulnerabilities either).

Of the remaining seven issues, two (KB836989, KB828295) are not security vulnerabilities, but false positives in which a security exception prevents a safe operation from proceeding. Two more bugs (KB324488, KB321562), also false positives, do not throw a security exception, but still prevent a normally safe operation. Two of the remaining bugs are in system classes that were implemented incorrectly. The first (KB327132) ignores a parameter for Passport authentication in ASP.NET incorrectly authenticating users without requiring a PIN. The other (KB839289) is a GC heap corruption exhibited in a cryptography provider class when the class constructor is called during garbage collection. The last

Microsoft knowledge base bug (KB323683) is an optimization fix for NLTM authentication that does not require re-authentication on multiple calls over the same connection. Although these are legitimate security issues, none of them are at the level of the .NET platform itself.

REFERENCES

AlephOne. Smashing the stack for fun and profit. Phrack November 1996;7(49).

Baier Dominick. Security Bug in .NET forms authentication, <http://sourceforge.net/mailarchive/forum.php?thread_id=5671607&forum_id=24754>; November 1996.

Clear Tony. Design and usability in security systems – daily life as a context of use? ACM SIGCSE Bulletin December 2002;4(34).

Cook WR. A proposal for making Eiffel type-safe. In: Third European conference on object-oriented programming (ECOOP); July 1989.

Core Security Technologies Advisory. Snort TCP stream reassembly integer overflow vulnerability, <http://www.securityfocus.com/advisories/5294>; December 2002.

Dean Drew, Felten Edward W, Wallach Dan S. Java security: from HotJava to Netscape and beyond. IEEE Symposium on Security and Privacy May 1996.

Directx 9.0 SDK Update. Sound permission, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_m/directx/ref/ns/microsoft.directx.security/c/soundpermission/soundpermission.asp>; May 1996.

ECMA International. Standard ECMA-335: common language infrastructure. 2nd ed. <http://www.ecma-international.org/publications/standards/Ecma-335.htm>; December 2002.

Erlingsson Úlfar. The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University Department of Computer Science (Technical Report 2003–1916); 2003.

Evans David, Twyman Andrew. Policy-directed code safety. IEEE Symposium on Security and Privacy May 1999.

Farkas Shawn. List of bugs that are fixed in the .NET Framework 1.1 Service Pack 1 (SP1), <http://blogs.msdn.com/shawnfa/archive/2004/09/02/224918.aspx>; May 1999.

Freeman Adam, Jones Alan. Programming .NET security. O'Reilly; June 2003.

Gong Li, Ellison Gary, Dageforde Mary. Inside Java 2 platform security. 2nd ed. Sun Microsystems; June 2003.

Hopwood David. Java security bug (applets can load native methods). Risks Forum March 1996.

LaDue Mark. A collection of increasingly hostile applets, <http://www.cigital.com/hostile-applets/>; March 1996.

LaMacchia Brian A, Lange Sebastian, Lyons Matthew, Martin Rudi, Price Kevin T. NET framework security. Addison-Wesley; April 2002.

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>; April 2002.

Leroy Xavier. Java bytecode verification: an overview. In: Computer aided verification, vol. 2101. Springer Verlag; 2001. p. 265–85.

Lewin Mark. Email communication; January 2004.

Lindholm Tim, Yellin Frank. The Java virtual machine specification. 2nd ed. Addison-Wesley; April 1999.

McGraw Gary, Felten Edward W. Securing Java. John Wiley and Sons; January 1999.

Meier JD, Mackman Alex, Dunner Michael, Vasireddy Srinath. Building secure ASP .NET applications: authentication, authorization, and secure communication, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/SecNetch13.asp>; January 1999.

Meijer Erik, Gough John. Technical overview of the common language runtime, <http://research.microsoft.com/~emeijer/Papers/CLR.pdf>; January 1999.

Microsoft Corporation. Microsoft portable executable and common object file format specification, <http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx>; January 1999.

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>; January 1999.

Microsoft Corporation. Technology overview, <http://msdn.microsoft.com/netframework/previous/v1.0/overview/default.aspx>; January 1999.

Microsoft Visual Studio. Additional security considerations in Windows forms, <http://msdn.microsoft.com/library/en-us/vbcon/html/vbconadditionalsecurityconsiderationsinwindowsforms.asp>; January 1999.

Mitre Corporation. Common vulnerabilities and exposures (version 20040901), <http://www.cve.mitre.org/>; January 1999.

Netscape Communications Corporation. Netscape object signing, <http://web.archive.org/web/20040221120620/http://developer.netscape.com/docs/manuals/signedobj/trust/owp.htm>; January 1999.

Pierce Benjamin C. Bounded quantification is undecidable. In: ACM SIGPLAN symposium on principles of programming languages (POPL); January 1992.

Pilipchuk Denis. Java vs. .NET security, <http://www.onjava.com/pub/a/onjava/2003/11/26/javavsdotnet.html>; January 1999.

Pynnonen Jouko. Vulnerabilities in Microsoft's Java implementation, <http://www.securityfocus.com/archive/1/290966>; January 1999.

Saltzer Jerome, Schroeder Michael. The protection of information in computer systems. In: Fourth ACM symposium on operating system principles; October 1973 [revised version in Communications of the ACM, July 1974].

Stutz Daivd, Neward Ted, Shilling Geoff. Shared source CLI essentials. O'Reilly; March 2003.

Stutz David. The Microsoft shared source CLI implementation, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Dndotnet/html/mssharsourcecli.asp>; March 2003.

Sun Microsystems. JSR 202: Java Class File Specification Update, <http://www.jcp.org/en/jsr/detail?id=202>.

Sun Microsystems. New Java SE Mustang Feature: Type Checking Verifier, <https://jdk.dev.java.net/verifier.html>.

Sun Microsystems. Chronology of security-related bugs and issues, <http://java.sun.com/sfaq/chronology.html>; November 2002.

Sun Microsystems. Java 2 platform, standard edition: 1.4.2 API specification, <http://java.sun.com/j2se/1.4.2/docs/api/>; November 2002.

Sun Microsystems. Java 2 SDK 1.4.2 SCSL source, <http://www.sun.com/software/communitysource/j2se/java2/download.html>; November 2002.

Sun Microsystems. Java: the first 800 days, <http://web.archive.org/web/20000815090553/http://java.sun.com/events/jibe/timeline.html>; November 2002.

Sun Microsystems. Permissions in the Java 2 SDK, <http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html>; November 2002.

Sun Microsystems. Sun alert notifications, <http://sunsolve.sun.com/pub-cgi/search.pl,category: security java>; November 2002.

Sun Microsystems. Sun security bulletins article 218, <http://sunsolve.com/pub-cgi/retrieve.pl?doctype=coll&doc=secbull/218&type=0&nav=sec.sba>; November 2002.

Szor Peter. Tasting Donut, <http://www.peterszor.com/donut.pdf>; November 2002.

The DotGNU Project. DotGNU, <http://www.dotgnu.org/pnet.html>; November 2002.
The Mono Project. What is Mono?, <http://mono-project.com/about/index.html>; November 2002.
.NET framework developer's guide. Permissions, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconpermissions.asp>; November 2002.
Viega John, McGraw Gary. Building secure software. Addison-Wesley Pub. Co.; September 2001.
Walker David. A type system for expressive security policies. In: ACM SIGPLAN symposium on principles of programming languages (POPL); January 2000.
Wallach Dan, Felten Edward. Understanding Java stack inspection. IEEE Symposium on Security and Privacy May 1998.
Wallach Dan, Balfanz Dirk, Dean Drew, Felten Edward. Extensible security architectures for Java. In: Symposium on operating systems principles; October 1997.
Yellin Frank. Low level security in Java. In: Fourth international WWW conference; December 1995.

David Evans is an Assistant Professor at the University of Virginia and Chair of the Computer Science BA committee. He has SB, SM and PhD degrees in Computer Science from MIT. His research interests include program analysis, exploiting properties of the physical world for security, and applications of cryptography. He teaches courses on computer science, software engineering, security, and cryptography. For more, see http://www.cs.virginia.edu/evans/.

Nathanael Paul is a doctoral candidate at the University of Virginia in Computer Science. In 2000, he received a B.S. in Computer Science from Bob Jones University and a M.S. in Computer Science from Clemson University in 2002. His primary research interests in security includes electronic voting, virtual machines and malware. He is a member of ACM and USENIX.