Java Security

Dan Wallach, Rice University
dwallach@cs.rice.edu

What is Mobile Code?

- Code travels from server machine to your machine

Client

Applet

Browser

Server

Web Server

Code
Mobile Code is Your Friend

- Rich data display
- Efficient use of network
- Customize the experience

Mobile Code Is Scary

- Untrusted, possibly malicious code on your computer!
- Disclose or damage your private data?
- Spend your money?
- Crash your machine?
Mobile Code Can Be Safe

- *Sandbox policy*
  - no file system
  - limited networking
- All code prevented from doing dangerous things

How Mobile Code Works

```
Client   Server

Applet  | Verifier / Compiler  | Web Server
Java VM |               | Bytecode
Browser |               | Compiler
OS      |               | Code
```

```
Client Server

Applet | Verifier / Compiler | Web Server
Java VM |               | Bytecode
Browser |               | Compiler
OS     |               | Code
```
How Mobile Code Fails

- Verifier failures
- Secure service failures
- Name-space confusion
- Denial of service
- Trusted computing base issues

Verifier Failures

- Verifier failures allow malicious behavior
  - Violate protection rules
  - Forge pointers / make unchecked type casts
  - Read and write any address, execute arbitrary code

Original Microsoft JVM did not enforce access modifiers like `private` and `protected`
Verifier Internals

- Java VM has a stack-based architecture
  - Stack frames, local variables can be reused with new variable types
  - Verifier must validate types

```
if
    push(int)
    push(Object)
    pop(Object)
```

Verifier Internals

- Control / data-flow analysis
  - Track types on stack through all code
- Ugly bytecodes to handle
  - One instruction is equivalent to C switch statement
  - Internal subroutines within a method (jsr / ret)
  - Complex object initialization semantics
    - Memory allocation and initialization are not atomic
  - Exception tables introduce other control flows
Bugs in Sun / Microsoft Verifiers

- Many, many bugs found
- Most recent: Karsten Sohr (September 1999)
  - Microsoft did not properly flow type information through exception blocks
  - Result: Arbitrary type casting, system compromise

- `check_code.c` (from Sun’s source code):

```c
/*-
 * Verify that the code within a method block doesn't exploit any
 * security holes.
 *
 * This code is still a work in progress. All currently existing code
 * passes the test, but so does a lot of bad code.
 */
```

Building a Better Verifier?

- Kenny Zadeck (NaturalBridge) proposes:
  - Reduce bytecode to simpler format
    - Exceptions handled explicitly
    - Expand “subroutine” calls
    - Internal representation used by normal compilers
  - Data / control-flow analysis is now a standard problem

- Paul Martino (Ahpah) proposes:
  - Decompile bytecode to Java source, then recompile
  - Repeat until fixed-point or error
Defining “Correct” Bytecode?

- Simplest definition: bytecode has a corresponding “correct” Java source program
  - *Java Language Spec* is more precise than *JVM Spec*
  - Unnecessarily restrictive?
- Bytecode as its own formal language?
  - JVML – Stata and Abadi
  - Freund and Mitchell (OOPSLA ’98)
- Bytecode, version 2?
  - Abstract syntax trees, equivalent to Java source?

Abstract Syntax Trees vs. Bytecode

- ASTs easier to type check
  - No need for global dataflow analysis
- ASTs have same semantics as language
  - Bytecode has its own semantics
- Comparable compilation speed

- Bytecode was designed for an interpreter
- Modern Java systems use just-in-time compilers
Name Space Confusion

- Java “linking” happens dynamically at runtime
- ClassLoader: two functions
  - Map class name to bytecodes (fetch from network)
  - Map class name to internal representation (name space / linking)
    - Confusion allows for unchecked typecasting

Applet
Java VM
Browser
Verifier / Compiler

Applets are not allowed to create their own ClassLoaders

Name Space-based Attacks

- Name equality does not imply type equality
  - Attack by David Hopwood, 1996

```java
// Applet 1
class BadOutputStream extends OutputStream {
    public Object obj;
    ...
}

// Applet 2
class BadOutputStream extends OutputStream {
    public int obj;
    ...
}

// Shared system class, writable variable
class System {
    public InputStream in;
    public OutputStream out;
}
```
Fixing Name Spaces


- Rules that a ClassLoader must follow
- Rules for how dynamic type casting works

- Still possible to get in trouble with ClassLoader (ClassLoader still restricted)

Name Space Problems Again

- Balfanz, Dean, Felten, Wallach (August 1999)
- Race condition in Microsoft’s ClassLoader
- Two cooperating threads
  - Primary thread asks ClassLoader to map name to class
  - Helper tries to interrupt primary thread
  - `Thread.stop()` sends an asynchronous exception
- Results: same name resolves to more than one class
  - Access to “package scoped” variables anywhere
**Name Space: Deeper Problems**

- Tension between static linking and dynamic loading
  - Goal: running before loading / verification complete
  - Problem: incomplete type information when verifying
  - Solutions: rigid rules, dynamic type constraints
- ClassLoader hacks are dangerous
  - RMI (remote method invocation) will dynamically load classes for objects it has not seen
  - Complex ClassLoaders lead to security failures

---

**Denial of Service**

- Consume infinite memory or CPU resources
- Impossible to terminate safely
  - Applet can catch exceptions from `Thread.stop()`
  - `Thread.destroy()` is dangerous

```java
class BadApplet {
  public void start() {
    while(true) {}  
  }
}
```

- `Thread.stop()` is “deprecated” but still supported in Java 2.
Safe Termination

- Threads are not the same as processes
  - Unix process encapsulates all resources in use
  - Unix kernel tracks all resources in use
- Java threads can cross from “user” to “kernel” code
  - Memory is shared
  - Resources in use are not tracked
- Separate JVM per applet (Digitivity, AT&T, others)
- Process-style solutions (U. of Utah)
  - Restrictions on memory sharing

Class vs. Thread Termination

- Our goal: terminate “applets”
  - Applet is a set of classes loaded by one ClassLoader
- Rewrite applet bytecode while loading
  - Add code to check “termination” flag
    - Once per basic block of code
  - Overhead will vary (worst cast: code with tight loops)
    - No overhead on system classes
- Applet threads will now terminate in finite time
- System code will not be disturbed by applet termination

joint work with Algis Rudys
Secure Services

- System classes enforce sandbox policy
  - Bugs in system classes lead to security failures

- Name space attack relied on a race condition in a system class

- Sun JVM 1.0 had bugs in network connect logic. Applets could connect anywhere.

Netscape 2.0 Insecurity

- Java trusts DNS
  - Internet hosts can have multiple IP addresses
  - Java host equality test is *too lenient*

- With a hacked DNS server
  - Two-way channel to any machine on the Internet
  - Applets can connect to machines *behind* a firewall
    - Exploit numerous Unix and Windows bugs
    - Talk to internal Web and NetNews servers
Netscape DNS Attack

The DNS attack allows connections to any machine behind the firewall.  
*Joint work with Dean and Felten (1996)*

Another Secure Services Problem

- *Some* parts of Java still need the file system!
  - URL file cache
  - Class dynamic loader
- Secure services
  - Use dangerous primitives
  - Export safe interfaces
  - How to decide if an operation should be allowed?
Handling the “Maybe” Cases

- Dangerous actions should be forbidden unless explicitly allowed
  - principle of least privilege
  - fail-safe

Solution: Stack Inspection

- Code must explicitly authorize a dangerous action
  - A method enables its privileges
    - Privileges revert when the method returns
- Used in Netscape 4, Microsoft IE 4, Sun JDK 1.2
  - Invented at Netscape
How Stack Inspection Works

What if the URL code wants to use a file cache?

```
Applet Applet()
System Privilege.enable(FileRead, "cache/...")
System URL.open("http://foo.com")
System File.open("cache/XQ45Z9")
```

How Stack Inspection Works

First, enable the file reading privilege...

```
Applet Applet()
System URL.open("http://foo.com")
System Privilege.enable(FileRead, "cache/...")
```
How Stack Inspection Works

... which places an annotation on the stack

System
Privilege.check(FileRead, "cache/...")

System
File.open("cache/XQ45Z9")

System
URL.open("http://foo.com") +FileRead, "cache/..."

Applet
Applet()

How Stack Inspection Works

... then search for the enabled privilege

System
Privilege.check(FileRead, "cache/...")

System
File.open("cache/XQ45Z9")

System
URL.open("http://foo.com") +FileRead, "cache/..."

Applet
Applet()
**How Stack Inspection Works**

**What if the privilege was never enabled?**

- Privileged?
  - System
    - File.open("cache/XQ45Z9")
- Privileged?
  - System
    - URL.open("http://foo.com")
- Not
  - Applet
    - Applet()

---

**Netscape 4.0 Privileges**

- File system, network
  - UniversalFileRead, UniversalFileDelete, UniversalAccept, UniversalConnect
- Browser features
  - UniversalPrintJobAccess, UniversalSendMail
- Parameterized variants of universal privileges
  - FileRead, FileWrite
- Macros
  - TerminalEmulatorAccess, GamesAccess
Why Stack Inspection is Cool

- Software engineering experience
  - Security audits
  - Porting code
- Formal basis
  - Modeled with a belief logic
- Fast implementation
  - Based on the formal model
  - Portable, compiler-friendly
- Extends naturally to remote procedure calls

My Contributions

Trusted Computing Base

- TCB – the subset that must be correct for the system to be secure
  - TCB minimization = secure software engineering
  - Stack inspection helps reduce the TCB
**Browser / External Interaction**

- Some “safe” modules are dangerous
  - ActiveX problems: Richard Smith, Phar Lap
- Trap users with infinite popup windows

```
Applet
  └── Verifier / Compiler
    ├── Java VM
    └── Browser
```

**Conclusions**

- Java has had serious problems
  - Security issues at all levels of the design
- Great research problems come from security holes
  - Dean’s PhD research: understanding class loading
  - My PhD research: understanding stack inspection
  - My current research: how to build a “secure” Java OS
- Java is a great source of research problems
  - Combine hacking, theorem proving, software engineering and press releases