CMPSC 497: Java Security

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Enforcement Mechanisms

• Static mechanisms
  ‣ Analyze programs before they are run; reject them if statically deemed harmful
  ‣ Static analysis; program verification

• Dynamic mechanisms
  ‣ AKA, reference monitors
  ‣ Analyze programs as they run; reject harmful runtime behavior
  ‣ Hardware-based fault isolation
  ‣ Software based
    • SFI

• A hybrid approach
  ‣ Java security
Java is Type Safe

- Array bounds checking
- Non-null checking
- Automatic memory management: GC
- Access control modifiers: public, private, protected, ...
- No bad cast: upcast and downcast
- String: always have a length stored
  - Compare this to null-terminated strings in C
- Initialization
  - A variable is always initialized before used
Java Security is Beyond Type Safety

- A design goal of Java is to support mobile code
  - Make interactive web pages
  - Java’s term for mobile code: applets
  - HTML has the `<APPLET>` tag
  - Applet code is downloaded and run automatically
    - A new JVM is started

- Concerns of mobile code
  - Platform independence – “Write once, run anywhere”
  - Security
## Security Threats of Applets

<table>
<thead>
<tr>
<th>Threats</th>
<th>Examples</th>
<th>Java’s defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>System modification</td>
<td>delete files; kill processes; make network connections</td>
<td>Strong</td>
</tr>
<tr>
<td>Invasion of privacy</td>
<td>Steal passwords, data</td>
<td>strong</td>
</tr>
<tr>
<td>Denial of service</td>
<td>Using up memory, CPU cycles; Completely filling up a file system;</td>
<td>weak</td>
</tr>
<tr>
<td>Antagonism</td>
<td>Playing annoying sound</td>
<td>weak</td>
</tr>
</tbody>
</table>
Java’s Rationale for Applet Security

- Stopping the worst kinds of attacks that hostile applets might carry out
  - Software-based protection; use language mechanisms
- Kind of successful
  - Ever heard of Java viruses?
Java’s Mobile Code

- Portable & Secure
- No need to trust the front end
Three Pillars of Java Security

• Class loaders
  ‣ Loading classes into the JVM

• Bytecode verifier
  ‣ Perform dataflow analysis to verify type safety of bytecode

• Security manager
  ‣ Monitors dangerous operations such as file accesses
Not Three Layers of Defense

*From "Securing Java" by McGraw and Felten
Bytecode Verification

• Ensure basic safety properties of the bytecode
  ‣ The file is not damaged
  ‣ The code is type safe
    • Which implies memory safety
  ‣ Does not prohibit the bytecode from reading your secret files
    • The job of the security manager

• Bytecode verifier
  ‣ a static verifier
  ‣ bytecode checked only once before running
  ‣ does not assume the bytecode is produced by a Java compiler; the bytecode could be written by an attacker
Bytecode Verifier: Multiple Passes

- **Pass 1:** file integrity check
  - Check the first four bytes is the magic hex number 0xCAFEBABE
  - Check version numbers of the class file
  - Check each structure in the file is of appropriate length

- **Pass 2:** structural checks
  - Final classes are not subclassed
  - Every class has a superclass (except for `java.lang.Object`)
  - Final methods are not overridden
  - ... 

- **Pass 3:** type checking the code
  - The most important and complicated pass
  - Performs data-flow analysis to verify type safety
The Bytecode Language

• A typed, stack-based intermediate language

• Types are encoded in the opcodes of instructions
  ‣ For example
    • iadd: integer add
    • ladd: long add
    • fadd: float add
    • dadd: double float add
  ‣ So that type safety can be verified
The JVM Types

- JVM Types and their prefixes
  - Byte b
  - Short s
  - Integer i (Java booleans are mapped to JVM ints!)
  - Long l
  - Character c
  - Single float f
  - Double float d
  - References a to Classes, Interfaces, Arrays

- These Prefixes used in opcodes (iadd, astore,...)
State

- State: operand stack, local variables (registers), heap (storing Java objects)
- Most instructions expect arguments on the operand stack
  - A stack-based abstract machine
- Operand stack: a stack of operands
  - pop
  - Various push operations: iconst_3, aconst_null, ...
  - iadd: pop two operands from the stack, add them, and push the result to the stack
    - E.g., iconst_3; iconst_4; iadd
  - iload d: the value of the local variable d is pushed onto the stack
  - istore d: the top of the stack is popped to variable d
The JVM Instruction Mnemonics

- Shuffling (pop, swap, dup, ...)
- Calculating (iadd, isub, imul, idiv, ineg,...)
- Conversion (d2i, i2b, d2f, i2z,...)
- Local storage operation (iload, istore,...)
- Array Operation (arraylength, newarray,...)
- Object management (get/putfield, invokevirtual, new)
- Push operation (aconst_null, icost_m1,....)
- Control flow (nop, goto, jsr, ret, tableswitch,...)
- Threading (monitorenter, monitorexit,...)
Compilation Examples

```java
public static int add (int x, int y) {
    return x + y;
}

Get compiled to (dumped by javap)

0:  iload_0
1:  iload_1
2:  iadd
3:  ireturn
```
Compilation Example Two

```c
static int factorial (int n) {
    int res;
    for (res=1; n>0; n--)
        res = res * n;
    return res;
}
```
Result of Compilation

0:  iconst_1  //push the integer 1
1:  istore_1  //store it in register 1 (the res variable)
2:  iload_0  //push register 0 (the n parameter)
3:  ifle 16  //if negative or 0, goto PC 16
6:  iload_1  //push register 1 (the res variable)
7:  iload_0  //push register 0 (the n parameter)
8:  imul  //perform multiplication
9:  istore_1  //store it in register 1
10:  iinc 0, -1  //decrement register 0 by 1
13:  goto 2  //go to PC 2
16:  iload_1  //load register 1 (res)
17:  ireturn  //return the value to caller
Verifying Type Safety

• **Phase 1**: disassembling; locate the start of each instruction
  ‣ Check all control transfers target valid start addresses
  ‣ Check all opcodes have the correct number of arguments
  ‣ ...

Verifying Type Safety

- **Phase 2**: Use static analysis to calculate types of local variables and stacks before each instruction
  - Stack: number and types of items
  - Local variables: types of local variables
    - Their values are not tracked
- **Perform type checking along the way**
  - If adding two integers, check there are enough arguments on the stack and the operands are of type int
  - When a function is invoked, check if enough arguments with correct types are on the stack
  - ...

...
Some Examples of Verification

• At each program point, track
  ‣ types of local variables; sizes and types of the operand stack

{0:int, 1:int}, stack:[]

0: iload_0
  {0:int, 1:int}, stack: [int]

1: iload_1
  {0:int, 1:int}, stack: [int, int]

2: iadd
  {0:int, 1:int}, stack: [int]

3: ireturn

• What happens if
  ‣ “iload_1” is omitted?
  ‣ “iadd” changed to “fadd”?
Example Verification: Loops

0:  iconst_1     //push the integer 1
1:  istore_1     //store it in register 1 (the res variable)
2:  iload_0      //push register 0 (the n parameter)
3:  ifle 16      //if negative or 0, goto PC 16
6:  iload_1      //push register 1 (the res variable)
7:  iload_0      //push register 0 (the n parameter)
8:  imul         //perform multiplication
9:  istore_1     //store it in register 1
10: iinc 0, -1    //decrement register 0 by 1
13: goto 2       //go to PC 2
16: iload_1      //load register 1 (res)
17: ireturn      //return the value to caller

- What happens
  - if “goto 2” is changed to “goto 5”?
  - if “istore_1” is omitted?
Threats Eliminated

- Contain illegal bytecode instructions
- Contain illegal parameters to bytecode instructions
  - Too many; too few; of wrong types
- Overflow or underflow the operand stack
- Illegal type cast
  - Cast an integer to a reference (type confusion attack)
- Illegal access to classes, fields, or methods
  - Nonexistent class, field, method; wrong types; with no access rights
Complications

• Verifying OO types
  ‣ A hierarchy of OO types
  ‣ Merge the OO types of two paths
    • Need to find the first common supertype

• Dynamic loading
  ‣ Classes are loaded before used for the first time
  ‣ When doing verification, doesn’t have all code

• References
  ‣ “Enterprise Java Security” by Pistoia et al.
  ‣ “Java bytecode verification, an overview” by Leroy
Next, the Security Manager

• Bytecode verifier checks type safety
• But what about other dangerous operations such as accessing files, or sending network packets?
• Java’s solution
  ‣ Let administrators specify a security policy
  ‣ Security manager enforces the policy during runtime
• This design went through several iterations
The Applet Sandbox In Java 1.0

- **Idea**: limit the resources that can be accessed by applets
  - Local code: no restriction
  - Untrusted code cannot read/write files; make arbitrary network connections; …
  - It can: access to the CPU, access to memory to create objects; connect to the site where the applet is downloaded
Java 1.1: Signed Applets

**Idea:** applets from a trusted source can be trusted

- local code
- applets signed by trusted principals
- unsigned applets

[Diagram showing JVM, sandbox, resources, and arrows connecting local code, signed applets, and unsigned applets.]
Java 2: User-Defined Policy

**Idea:** Every code (remote or local) has access to the system resources based on what is defined in a *policy file*.

- **Example**

  ```
  grant CodeBase "http://java.sun.com", SignedBy "Sun" {
    permission java.io.FilePermission "${user.home}${/}/*", "read, write";
    permission java.net.SocketPermission "localhost:1024-", "listen";
  }
  ```
Security Manager: Enforcer

- A reference monitor
- The security manager are consulted before dangerous operations are performed
Consulting the Security Manager

From java.io.File:

```java
public boolean delete() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkDelete(path);
    }
    if (isDirectory()) return rmdir0();
    else return delete0();
}
```

checkDelete throws a SecurityException if the delete would violate the policy
Stack Inspection

• Dynamic authorization mechanism
  ‣ close (in spirit) to Unix effective UID
  ‣ attenuation and amplification of capabilities

• Richer notion of context
  ‣ An operation can be good in one context and bad in another
  ‣ E.g., all local file access goes through java.io.File
    • Applet 1: no file-access permissions; call java.io.File
    • Applet 2: with file-access permissions; call java.io.File
Call Stack is the Context

• Each method on the stack has an associated **protection domain**
  ‣ A method belongs to a class
  ‣ Each class is assigned a protection domain depending on its origin

• **Protection domain**: code with permissions
  ‣ System domain: system classes (e.g., java.io.File) + AllPermissions
  ‣ Multiple application domains
    • E.g.,
      grant CodeBase “http://java.sun.com”, SignedBy “Sun” {
        permission java.io.FilePermission “${user.home}/[*]”, “read, write”;
        permission java.net.SocketPermission “localhost:1024-”, “listen”;
      }
    • This domain specifies all code from java.sun.com has certain file permissions and socket permissions
System Domain vs. App Domains

- System domain can access protected resources
- App domains calls into the system domain for access to resources

Fig 8.9 “Enterprise Java Security”
**checkPermission**

- `security.checkDelete(path)`
  - Turn into `security.checkPermission(P)`
    - P stands for the permission to remove file with path

- `checkPermission (P)` is implemented as
  ```
  for (f := pop(S); !empty(Stack) ; f := pop(S)) {
    if domain(f) does not have perm. P then error;
  }
  ```
  - Ignoring privileged code (see later slides)
Examples

- Applet 1: no file-access permissions; call java.io.File
  - access denied

- Applet 2: with file-access permissions; call java.io.File
  - access allowed
Motivation for Privileged Code

• Suppose
  ‣ A class in the system domain needs to connect to the internet and write to a log file
  ‣ An untrusted app invokes the class to make a network connection

• With stack inspection
  ‣ The untrusted app needs both SocketPermission and FilePermission
  ‣ But is requiring FilePermission reasonable?

Fig 8.2 “Enterprise Java Security”
• doPrivileged(P){S}:
  ‣ fails if method's domain does not have perm. P.
  ‣ exempt the caller from having perm. P
  ‣ checkPermission (P) is implemented as
    for (f := pop(S); !empty(Stack) ; f := pop(S)) {
      if domain(f) does not have perm. P then error;
      if f is a doPrivileged frame then break;
    }
• Q: Why can't any code use “doPrivileged(P){…}” to grab the perm P?
• If the log() operation is wrapped in
  ‣ doPrivileged(FilePermission){log();}
  ‣ Then the untrusted code needs the SocketPermission to call the trusted library code
Class Loader

• Loading Java classes into the JVM

• Dynamic loading
  ‣ loading classes in a lazy way
  ‣ when a class is referenced, it is the class loader’s job to find that class
    • reference a field/method, create a new object
Class Loader Responsibilities

- Maintaining separation of namespaces
  - Maintain separation between classes of different trust levels (loaded from different sources)

- Permission assignment
  - Associating permissions to protection domains

- Search-order enforcement
  - Prevent user classes from spoofing security-critical classes, such as the security manager
Caveats of Java Security

- Its TCB is quite big
  - Many security holes were identified and fixed in the past
- Native code
- Serialization
- ...

...
Further Readings

- Secure Coding Guidelines for the Java Programming Language