Secure Programming I

Engin Kirda
kirda@eurecom.fr

Administrative News

- Challenges
  - Good stuff: 23 people have successfully submitted
  - Challenge 4 will be announced on the 11th of May
  - Source code…
- Slides and News (please visit regularly!)
  - http://www.iseclab.org/secprog/

Language Security (Java)

Overview

- Introduction
- Java Security Concept
- Common Attack Techniques

Introduction

- Java
  - Developed by Sun Microsystems in the early 1990s
  - Designed to be platform-independent
  - Compiled to *bytecode* that runs on a Virtual Machine (VM)
  - **Designed with security in mind - especially for mobile code**

Java Security Concepts

- Strong data typing
  - Compiler enforces that every operation is only applied to values of a type compatible to that operation
  - Robin Milner provided the following slogan to describe type safety:
    "Well-typed programs never go wrong."

Java Security Concepts

- Automatic Memory Management
  - Garbage Collection takes care of disposing unused objects -> no memory leaks
  - Memory bound violations are caught (i.e., ArrayIndexOutOfBoundsException)

Java Security Concepts

- Secure Class Loading
  - Class definitions are checked for validity before loading
  - Bytecode is verified before execution to ensure
    - validity
    - type safety

Java Security Concepts

- Sandboxing
  - Untrusted code is not allowed to perform security critical tasks
  - Especially web applets have to be safeguarded

Java Security Architecture

- Java Security Architecture consists of three vital parts
  - Class Loader
  - Bytecode Verifier
  - Security Manager

Class Loader

- Loads class definitions from files or from a remote host
- Checks if definitions are valid (0xCAFEBABE)
- Invokes the Bytecode Verifier
- Distinguishes system and user classes

Bytecode Verifier

- Performs a four pass static data flow analysis on the bytecode
  - execution of bytecode instructions is modelled
  - every execution path is simulated
  - dynamic fourth pass to discover linking errors
- Identifies
  - Class inconsistencies
  - Type Safety violations
  - Illegal byte code sequences
    - Java semantics
    - comprehensible by the VM

Security Manager

- Responsible for enforcing a given security policy
- Implemented as a Java class
- Every “dangerous” operation first calls a check routine of the Security Manager
- Not all classes are checked the same way for performance reasons
  - System classes are inherently trusted – no security checks at all

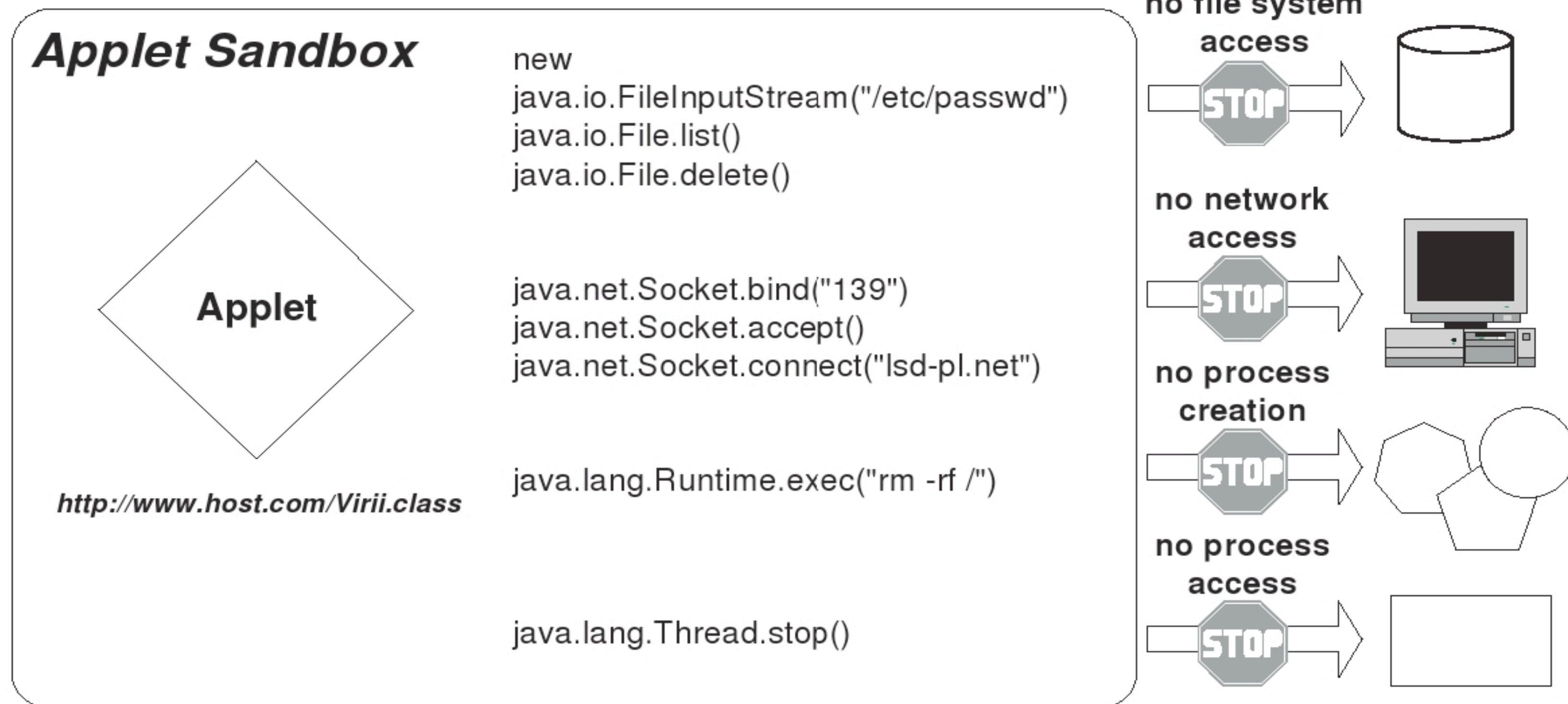
Security Manager

- Every Security Manager method starts with check...
- Example

```
public boolean mkdir()
{
	SecurityManager securitymanager = System.getSecurityManager();
	if(securitymanager != null)
		securitymanager.checkWrite(path);
	return mkdir0();
}
```

Security Manager

- Provides sandboxing of Java applets



Attack Techniques

- Various different kinds of attacks
  - attacks that exploit bugs in the VM
    - might compromise the whole platform when successful
    - implementation specific
  - attacks against the system classes
    - compromise the whole Java platform
    - can even attack the underlying system through trusted classes
  - attacks against user code
    - compromise the user application only
    - less risk for the underlying system but still problematic

Type Confusion Attack

- Java enforces that every operation is only applied to objects of a type the operation was defined on
- Additionally, every casting operation has to be done in an implicit way
- During the casting process, strict rules are enforced
  - casting is only allowed between compatible classes, interfaces or collections

  -> Java is type safe

- Why is Type Safety that important?

Type Confusion Attack

- Assume the following is possible

```
public class Original {
	private boolean initialized;
	private Security sec;
}
public class fakeOriginal {
	public boolean initialized;
	public Security sec;
}

fakeOriginal = cast2fakeOriginal(original);
fakeOriginal.initialized = true;
fakeOriginal.sec = new Security(MODE_UNRESTRICTED);
```

Type Confusion Attack

- The code line in red effectively breaks type safety and allows altering private fields of a class

- This attack, if successful, breaks the security of the whole Java system (Privilege Escalation Attack)

- Type Safety is checked mostly through the Bytecode Verifier
  -> small flaws in the verifier can break the system

Class Spoofing

- Attack against the class loader
- The Class Loader should
  - load every class definition at most once
  - make sure that there exists only one unique class file for a given class name
- There may exist more than one class loader in a given VM whose name spaces may overlap
  -> the name spaces have to be separated carefully

Class Spoofing

- Idea: Trick the VM into thinking a class is defined by another class loader than it really is

```
Class Loader CL1:
  public Spoofed {
      public Object var;
  }

  Spoofed sp = new Spoofed();
  sp.var = ??
```

```
Class Loader CL2:
  public Spoofed {
     public MyArbitraryClass var;
  }
```

- This attack can be used to cast any Object to MyArbitraryClass
  -> Type Confusion Attack

Attacking the Verifier

- Verifier is the most important check against type safety breaking byte code instructions and thus a rewarding target
- Breaking the data flow analysis allows inserting
  – arbitrary casts (type confusion attacks)
  – illegal instructions (breaks the VM)
- A possible attack against the verifier:
  – trick the data flow analysis into taking the wrong execution path

Privilege Escalation

- Common attack scheme used previously by applets to escape the sandbox (IE, Netscape)
- Idea: Use a type confusion attack to change critical fields of a security relevant class
  - Security Manager: grant more permissions to the malicious class
  - Class Loader: load and run another class with extensive rights
- Alternative: Use a bug in a system class that runs with privileged rights

Privilege Escalation

```
ClassLoader cl = getClass().getClassLoader();

VerifierBug bug = new VerifierBug();
MyURLClassLoader mucl = bug.cast2MyURLClassLoader(cl);

PermissionDataSet pds = new PermissionDataSet();
pds.setFullyTrusted(true);
PermissionSet ps = new PermissionSet(pds);
mucl.defaultPermissions = ps;

Class c = cl.loadClass(“Beyond”);
c.newInstance();
```

JIT Bugs

- When Java programs are compiled to native code, the execution speed greatly improves
- Downside: security checks are reduced too
  - Example: buffer overflows
- Example: Microsoft JVM had a long history of applets crashing when JIT enabled

Attacks Against Java Classes

- Attacks we discussed until now targeted VM implementation bugs, the following attacks exploit Java programming mistakes
- In the Java Security Model, system classes are inherently trusted
- Even small mistakes in those classes pose a huge threat to the security model

  -> they are obvious targets of security related attacks

- Of course, user classes should be written as secure as possible too!

Integer Overflow

- Integer overflows are a common problem in C:
  - size checks can be bypassed and buffers overflown which can lead to the execution of injected code

- Java suffers from the same problem but with much less impact
  - size checks can still be bypassed but memory access is forbidden
  - Nevertheless, out of bounds exceptions can be raised
    -> denial of service is possible

Integer Overflow

- The Integer data type in Java has a range from -2^{31} to $+2^{31}-1$
- In Java the following is true:
  - $+2^{31}-1 + 1 == -2^{31}$
  - Integer.MAX_VALUE + 1 == Integer.MIN_VALUE
- Problem: Java integer overflows are silent
  - unlike on x86 machines, there is no way for the program to notice an overflow

Integer Overflow

- Example:

```
private string [] names;
private void output(int from, int len)
{
	if (names == null   ||   from < 0   ||   from + len > names.length)
		return;
	for (int i=0; i<len; i++)
		System.out.println( names[i+from] );
}
```

What happens if from+len > Integer.MAX_VALUE??

Integer Overflow

DEMO

Integer Overflow

- Fixing the problem:
  change

```
if (names == null   ||   from < 0   ||   from + len > names.length)
```

To

```
if (names == null   ||   from < 0   ||   from > names.length - len)
```

- Integer overflow in java.util.zip.CRC32
  - CRC32 used native functions
  - overflow could crash the VM

Inappropriate Scope

- Java inheritance model is very powerful
  - provides high flexibility
  - downsize: complex security model needed
- Problem: Many classes do not properly limit access to their classes, methods and variables
- For user classes this could be problematic, for system classes this an almost always critical issue
  - Method overwriting attacks
  - Protected fields can be overwritten

Inappropriate Scope

- Java scope modifiers:

Specifier	Class	Package	Subclass	World
private	Y	N	N	N
no specifier	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

- Pitfall: “Protected” is less protected than no specifier !!

Inappropriate Scope

- Simple attacks

```
class A {
	protected int secret;
}

class Evil extends A {
	public int getSecret() { return secret; };
}
```

Inappropriate Scope

- Simple attacks

```
class SecureWrite {
    protected void realWrite() { ... };
    public void write() {
        if (checkPermission())
            realWrite();
    }
}

class EvilWrite extends SecureWrite {
    public void write() { realWrite(); };
}
```

Inappropriate Scope

DEMO

Inappropriate Scope

- Despite their simplicity, those attacks actually worked (and still might...)
  - even on highly security relevant classes/methods
- Known issues were:
  - java.lang.object: hashCode, equals, clone
  - java.lang.ClassLoader: loadClass, defineClass, resolveClass
  - java.lang.SecurityManager: implementation specific methods

Conclusion

- In this lecture, we looked at language security (with focus on Java):
  - Java security model
  - Class Loader
  - Byte Verifier
  - Security Manager
- .NET concepts are similar