
Language Security (Java)

Internet Security [1] VU

Christian Platzer, **Gilbert Wondracek**,

Martin Mulazzani, Edgar Weippl

inetsec@iseclab.org

News and Announcements

Challenges

- Challenge 4 is current
 - >45 people have completed it
 - 1st place goes to Armin Egger
 - Congratulations!

Overview

- Introduction
- Java Security Concepts
- Common Attack Techniques
- Walkthrough of a Java vulnerability
 - Privilege escalation: from a Java applet to executing commands on the host

Introduction

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**
 - what does it mean that "java is secure"?

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 of unused objects
 - no memory leaks
 - no access to memory after it is freed
 - Memory bound violations are caught
 - (i.e., `ArrayIndexOutOfBoundsException`)

Java Security Concepts

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

Java Language Security Constructs

Memory is protected...

- Programs cannot access arbitrary memory locations
 - no pointers
- Cannot use a variable before initialization
 - could be used to snoop on private data of a freed class instance
- Array bounds must be checked on all array accesses
 - no buffer overflows!

Java Language Security Constructs

Classes are protected...

- Access methods are strictly adhered to
 - private,protected,default(package),...
- Entities declared *final* cannot be changed
 - cannot change value of *final* variable
 - cannot override *final* method
 - cannot subclass *final* class
- Objects cannot be arbitrarily cast into other objects

How is this enforced?

- Cannot trust compiler
 - a malicious compiler could break these rules
 - we want to enforce these rules on untrusted code (applets..)
- Bytecode verifier
 - when loading (linking) code in virtual machine, check it
 - check bytecode is well formed
 - check properties (simple theorem prover)
 - Do not forge pointers, Access restrictions, ...
- Some properties cannot be checked statically
 - is a type cast valid? `x=(MyClass) vector.get(i)`
 - array index out of bounds?
 - bytecode verifier inserts a runtime check

Java Application Security

- IF the Java VM successfully enforces the rules of the Java language...
- ...We can build a security model for applications/applets on top of it

- The Java APIs provide one such model

Code Source

- Classes are trusted (and given permissions) based on where they come from
 - url, certificate (digital signature)...
- System classes (loaded from classpath on disk) have all permissions
- Applets (loaded by browser from untrusted website) have almost no permissions
- Applications can have more complex use-cases
 - application composed of different classes from different sources
 - application-specific permissions

Java Security Architecture

Java Security Architecture Components

- Protection Domain
- Class Loader
- Security Manager
- Access Controller

Protection Domain

- ProtectionDomain = Code sources + Permissions
- Standard permissions:
 - FilePermission("/etc/-,"read")
 - SocketPermission("localhost:1024-", "accept,listen")
 - Create a ClassLoader:
RuntimePermission("createClassLoader")
 - AllPermissions
- Application-specific permissions:
 - example: read/write access to a certain table in my database

Class Loader

- Loads class definitions from files or from a remote host
 - URLClassLoader
 - custom class loaders inherit from ClassLoader
 - load a class from a DB? load it over FTP?
- Checks if definitions are valid (0xCAFEBAFE)
- Invokes the Bytecode Verifier
- Assigns a ProtectionDomain to classes it loads
 - can grant arbitrary privileges!

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

Security Manager

- Every security-relevant method starts with check...
- Example:

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

Security Manager

- Every security-relevant method starts with check...
- Example:

```
public boolean mkdir()  
{  
    SecurityManager securitymanager =  
        System.getSecurityManager();  
    if(securitymanager != null)  
        securitymanager.checkPermission(  
            new FilePermission(path,"write"));  
    return mkdir0();  
}
```

Security Manager

- Provides sandboxing of Java applets



Access Controller

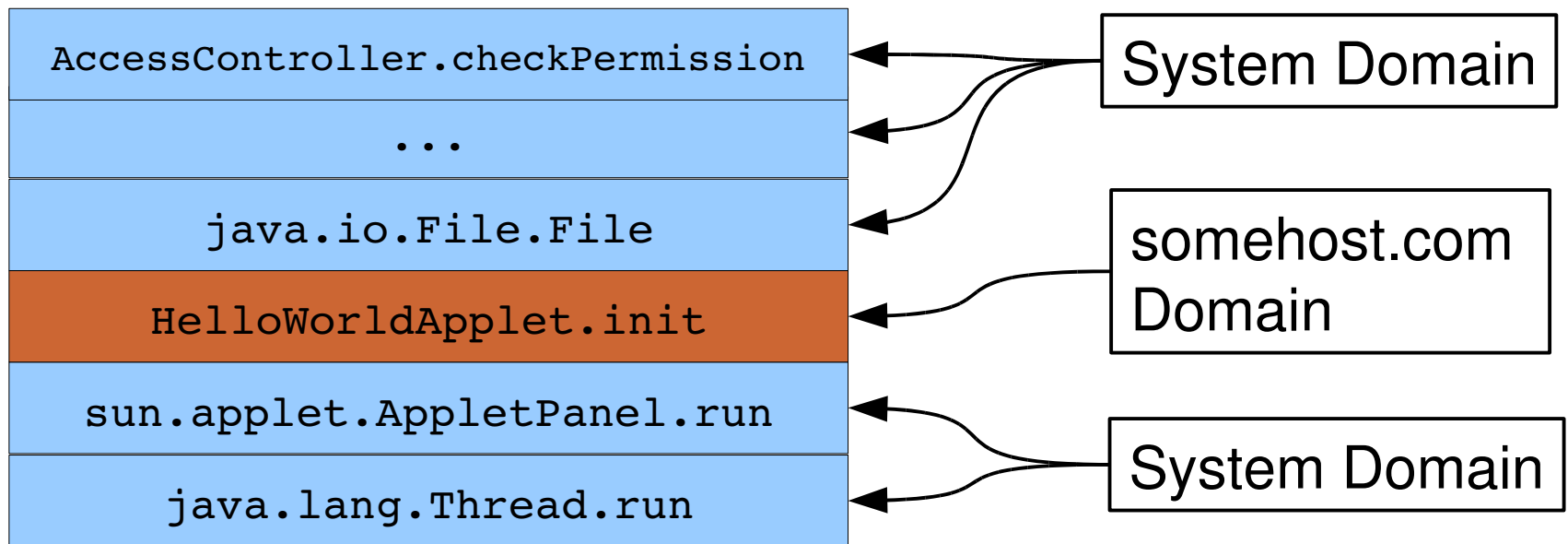
- Default Security Manager delegates decisions to the AccessController
- Policies for the access controller can be specified in a policy file
- How does the access controller decide which class is responsible for a certain action?
 - class -> ProtectionDomain -> Permissions

Example Policy File

```
keystore "file:///[/PATH]/keystore/ca", "xxx";
grant signedBy "[USER] " {
    permission java.io.FilePermission "<>", "write";
    permission java.net.SocketPermission
"1.2.3.4:8000",
    "connect, accept, listen, resolve";
};
```

AccessController

- All classes on stack need to have the required permission
 - _ Otherwise, not permitted!

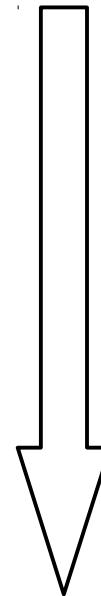
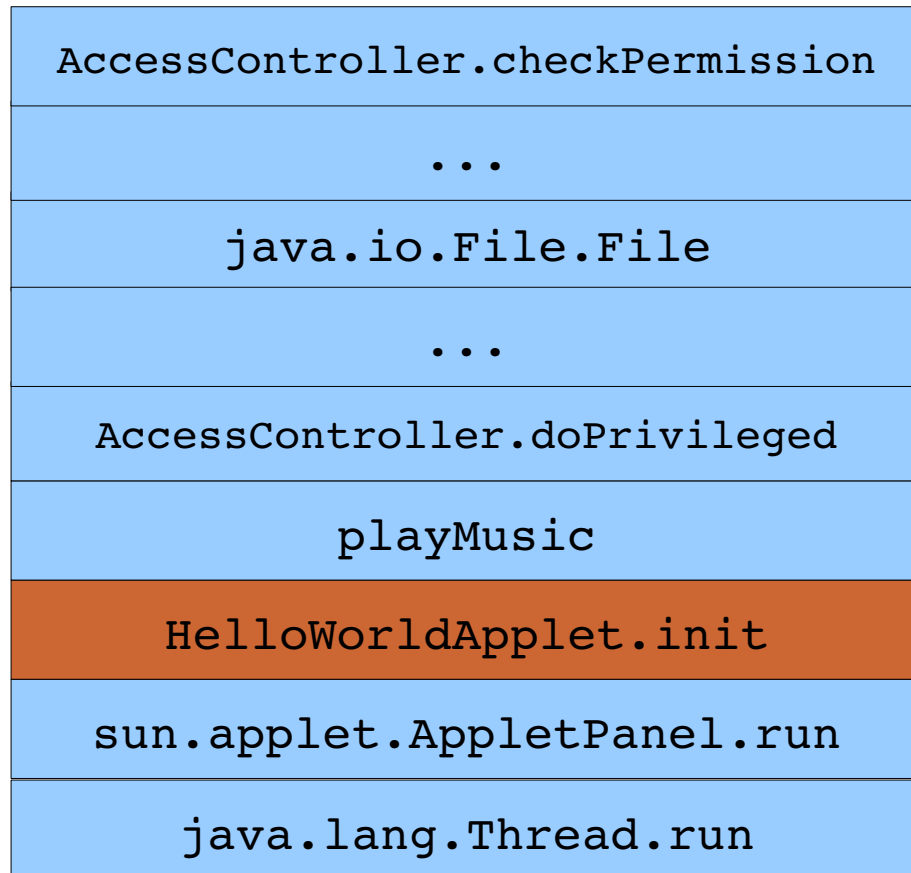


Stack

AccessController

- This may not be the required semantics
- Example:
 - an applet may be allowed to play music
 - music playing is implemented in trusted classes
 - to play music, need to write to a device (file)
- Perform an action on behalf of untrusted classes
- `AccessController.doPrivileged`
 - perform an action with the privilege of the calling class

AccessController



Only verify permissions up to the class that called doPrivileged

`playMusic`
has temporarily granted these classes permission

Serialization API

- Allows saving objects to file/network/etc, and loading them back
- Is an exception to the Java Language rules:
 - the serialization API can read private fields of a class (to serialize it)
 - the serialization API can write to private fields of a class (to deserialize it)

Serialization API

- Only allowed if class implements `java.io.Serializable` interface
- Making a class serializable has security implications
- Can read an object's private data.
 - just serialize it to somewhere (file, network, string)
- Can create objects with private attributes set to arbitrary values
 - just craft the serialized representation of an object with the required fields
 - then deserialize it

Reflection API

- The reflection API can be used to access private fields of any class
 - totally breaks the security model
- Requires `java.lang.reflect.ReflectPermission("suppressAccessChecks")`
- Never grant this permission to classes that are not 100% trusted
 - completely breaks Java class security model

Attack Techniques

Attack Techniques

- Several different classes 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

Attack Techniques

- The Java VM is a complex (C++?) application
- Takes as input Java programs
 - (and their input)
- Does complex processing of this input
 - byte verification, emulation,...
- Susceptible to the usual security problems!
 - buffer overflows, etc...

- Not our concern in this lecture

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

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

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

Attacking the Verifier

- The Bytecode 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)

Privilege Escalation

- Common attack scheme that has been used by applets to escape the sandbox (IE, Firefox, Safari,..)
- 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

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!

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 attributes
- 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

- 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

Walkthrough of a Recent Java Exploit (+DEMO)

Walkthrough of a Java Exploit

from CVE-2008-5353:

- The Java Runtime Environment (JRE) for Sun JDK and JRE 6 Update 10 and earlier; JDK and JRE 5.0 Update 16 and earlier; and SDK and JRE 1.4.2_18 and earlier does not properly enforce context of ZoneInfo objects during **deserialization** which allows remote attackers to **run untrusted applets** and applications in a privileged context, as demonstrated by "deserializing Calendar objects".

Walkthrough of a Java Exploit

- Was unpatched for many months on Mac OS X
- This is a pure Java exploit (no native code)
 - portable
- An explanation available at:
<http://blog.cr0.org/2009/05/write-once-own-everyone.html>
- Proof of concept available at:
<http://landonf.bikemonkey.org/static/moab-tests/CVE-2008-5353/hello.html>
- Decompiled it using **Jode**

java.util.Calendar.readObject

```
/** Reconstitutes this object from a stream (i.e.,
deserialize it)*/
private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException {
    final ObjectInputStream input = stream; (...)
    // If there's a ZoneInfo object, use it for zone.
    ZoneInfo zi = (ZoneInfo) AccessController.doPrivileged(
        new PrivilegedExceptionAction() {
            public Object run(){
                return input.readObject();
            }
        }
    );
```

java.io.ObjectInputStream.readObject

Called in privileged context

- If an instance of a class X is found in the stream, it is deserialized in a privileged context

```
ZoneInfo zi = (ZoneInfo) AccessController.doPrivileged(  
    new PrivilegedExceptionAction() {  
        public Object run(){  
            return input.readObject();  
        }  
    }  
);
```

java.io.ObjectInputStream.readObject

Called in privileged context

- If an instance of a class X is found in the stream, it is deserialized in a privileged context

```
ZoneInfo zi = (ZoneInfo) AccessController.doPrivileged(  
    new PrivilegedExceptionAction() {  
        public Object run(){  
            return input.readObject();  
        }  
    }  
);
```

Type check happens
after readObject
(too late!)

Walkthrough of a Java Exploit

- We can have an object of an arbitrary class X deserialized (in privileged context)
- Construct an **evil** serialized calendar object with an instance of class X instead of the ZoneInfo field
- Have your applet deserialize that object
- An X object is deserialized
 - X.readObject is called (this is our code!)

Do we win already?

<code>AccessController.checkPermission</code>
<code>...</code>
<code>doSomethingEvil</code>
<code>X.readObject</code>
<code>...</code>
<code>AccessController.doPrivileged</code>
<code>java.util.Calendar.readObject(<i>evilObject</i>)</code>
<code>...</code>
<code>HelloWorldApplet.init</code>
<code>sun.applet.AppletPanel.run</code>
<code>java.lang.Thread.run</code>

`Calendar.readObject`
has temporarily granted
these classes permission

Do we win already?

<code>AccessController.checkPermission</code>
<code>...</code>
<code>doSomethingEvil</code>
<code>X.readObject</code>
<code>...</code>
<code>AccessController.doPrivileged</code>
<code>java.util.Calendar.readObject(<i>evilObject</i>)</code>
<code>...</code>
<code>HelloWorldApplet.init</code>
<code>sun.applet.AppletPanel.run</code>
<code>java.lang.Thread.run</code>



Permission rejected!

`Calendar.readObject`
has temporarily granted
these classes permission

Walkthrough of a Java Exploit

- What types of objects am I normally forbidden from instanciating?
- A ClassLoader!
 - building an object that inherits from ClassLoader, requires `RunTimePermission("createClassLoader")`
 - a subclass of ClassLoader can call protected method `ClassLoader::defineClass`
 - `java.util.Calendar` has this permission
- Store it in a static variable
- Then you can load any classes you want in privileged context

Walkthrough of a Java Exploit

...
<code>AccessController.checkPermission(RunTimePermission("createClassLoader"))</code>
<code>java.lang.ClassLoader.ClassLoader</code>
...
<code>AccessController.doPrivileged</code>
<code>java.util.Calendar.readObject(evilObject)</code>
...
<code>HelloWorldApplet.init</code>
<code>sun.applet.AppletPanel.run</code>
<code>java.lang.Thread.run</code>

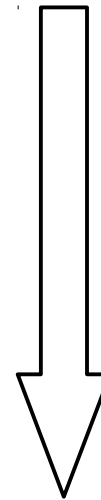
my class is:

Funloader extends ClassLoader

To construct it, `ClassLoader` constructor is called first

Walkthrough of a Java Exploit

...
<code>AccessController.checkPermission(RunTimePermission("createClassLoader"))</code>
<code>java.lang.ClassLoader.ClassLoader</code>
...
<code>AccessController.doPrivileged</code>
<code>java.util.Calendar.readObject(evilObject)</code>
...
<code>HelloWorldApplet.init</code>
<code>sun.applet.AppletPanel.run</code>
<code>java.lang.Thread.run</code>



Permission Granted!

`Calendar.readObject`
has temporarily granted
these classes permission

fun.FunLoader.bootstrapClass

```
(...)  
Permissions permissions = new Permissions();  
permissions.add(new AllPermission());  
Class var_class = defineClass(  
    string, is_0_, 0, is_0_.length,  
    new ProtectionDomain(  
        new CodeSource(url,certificates), permissions));
```

Command Execution

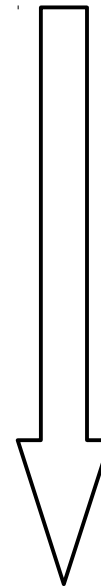
Privileged class can run arbitrary commands with permissions of Java VM

```
final String[] strings = { "/usr/bin/say",  
                           "YOU ARE OWNED"};
```

```
AccessController.doPrivileged(  
    new PrivilegedExceptionAction() {  
        public Object run() throws Exception {  
            Runtime.getRuntime().exec(strings);  
            return null;  
        }  
    }  
);
```

Command Execution

<code>java.lang.Runtime.exec(evilcommand)</code>
<code>AccessController.doPrivileged</code>
<code>Exec.Exec()</code>
<code>HelloWorldApplet.init</code>
<code>sun.applet.AppletPanel.run</code>
<code>java.lang.Thread.run</code>



Only verify permissions
up to the class that
called doPrivileged

`Exec.Exec()`
has temporarily granted
these classes permission

Summary

- We looked at a different kind of security issues
- While the topics presented were Java specific, the same underlying principles are similar in other languages
- Language security often overlooked even by “experts”

Thanks!