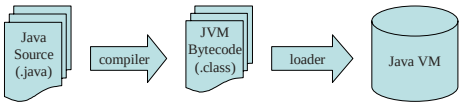
Java Security

Alexander V. Konstantinou
Columbia University
akonstan@cs.columbia.edu

Fall 2002

The Java Platform (Review)

- Java Programming Language
- Java Libraries
- Java Virtual Machine (JVM)



The Java Language

- Object-oriented
  - Single inheritance, interfaces
- Strong typing
  - No pointer arithmetic/conversion
  - Array bounds checking
- Garbage collection
- Exceptions
- Threads

Java Libraries

- I/O
- Utilities & collections
- Network programming
  - Sockets, RMI, CORBA
- Security: access control, crypto, authentication
- Graphics (GUI, 2D, 3D)
- SQL, XML

The Java Virtual Machine

- Abstract computing machine
  - Stack-based
- Knows nothing about Java language
- Specifies binary class file format
  - Class file contains VM instructions (byte-code)
- Emulated on different platforms
- Compilers exist for other languages
  - Ada, Smalltalk, Eiffel, COBOL, etc

Java Security Features

- Strong typing
- No pointer conversion/arithmetic
- Array bounds checks
- Multiple package name scopes
- Security model & instrumentation
- Security libraries
  - Encryption, signature, SSL

Java Security Evolution

- Java 1.0
  - Applets operate in sandbox
  - All other applications trusted
- Java 1.1
  - Signed applets treated as trusted applications
- Java 1.2 (Java 2)
  - New policy-based security architecture

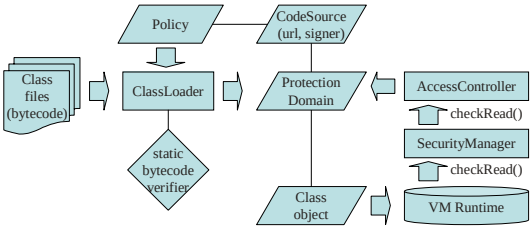
Applet Sandbox Security

- No file access
- No system property access
- Restricted network access
  - Can only connect to server host
  - No local host, or other network connections
- Windows opened have warning tag
- Cannot access other applet threads

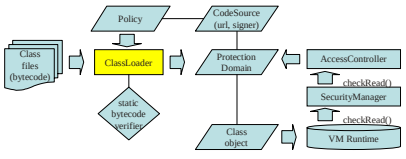
What's Special About Java Security?

- Security-conscious design
- Implemented in Java !?!
  - Security components are regular Java classes
- Need to secure the Virtual Machine
  - Compiler provides “advisory” access control
- Design supports extensibility
  - Interdependent components
  - Complex dependencies (bad news)

Java Security Components



Class Loader



Class Loader

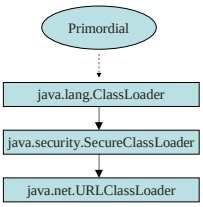
- Class loaders are regular Java objects
  - Chicken & egg problem
- Primordial class-loader
  - Written in C
  - Loads system classes
- Lazy class loading
- Dynamic class loading

Class Loader (2)

- Forms Class object out of byte-array
  - File, network, dynamic compilation
- Defines namespace
- Type defined as < class, loader >
- System classes have null class-loader

Class Loader Delegation

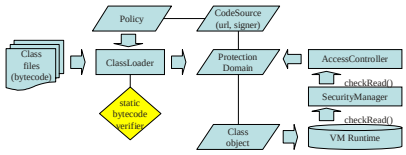
- Class loader delegation
  - Parent-child relationship
- Control access to delegation
- SecureClassLoader
- URLClassLoader
  - Loads across network



Customized Class Loader Example

```
public class MyClassLoader extends ClassLoader {

  public MyClassLoader(ClassLoader parent) {
    super(parent);
  }

  public Class loadClass(String name) {
    // Delegate to parent first
    try {
      return(super.loadClass(name));
    } catch (Throwable e) { }

    byte[] bytecode = new byte[0]; // XXX (read class file)
    return(defineClass(name, bytecode, 0, bytecode.length));
  }
}
```


Bytecode Verifier



Enforcing Type Safety

- Cornerstone of Java security
- Static type checking
  - Optimization step to reduce run-time checking
- Dynamic type checking

Bytecode Verifier

- Uses theorem prover
- Most complex Java security component
- Sun implementation is two-phase & complex
  - Difficult to formally verify
- Alternative research verifiers
  - Partially formally verified

Bytecode Theorem Prover Checks

- Pointer forging
- Class access violation
  - Private/protected fields and methods
- Object casting
- Method invocation
  - Correct number and type of arguments
  - No stack overflows
- No illegal data conversions
  - Integer → pointer

Java Assembly Example

```
public class Simple {
  public static void main(String[] args) {
    java.util.Date date = new java.util.Date();
    int i = 2002;
    i++;
  }
}
```

```
.source Simple.java
.class  public synchronized Simple
.super  java/lang/Object
; >> METHOD 1 <<
.method public <init>()V
    .limit stack 1
    .limit locals 1
.line 3
    aload_0
    invokenonvirtual java/lang/Object/<init>()V
    return
.end method
```

```
; >> METHOD 2 <<
.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 3
.line 5
    new java/util/Date
    dup
    invokenonvirtual java/util/Date/<init>()V
    astore_1
.line 6
    sipush 2002
    istore_2
.line 7
    iinc 2 1
.line 8
    return
.end method
```

```
javac Simple.java
D-Java -o jasmin Simple.class
```


Java Assembly Example (2)

```
; >> METHOD 2 <<
.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 3
.line 5
    new java/util/Date
    dup
    invokenonvirtual java/util/Date/<init>()V
    astore_2   ; was astore_1
```

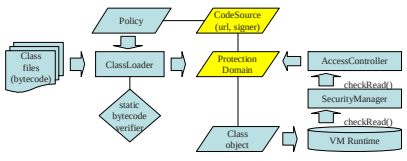
```
; .line 6
;    sipush 2002
;    istore_2
.line 7
    iinc 2 1
.line 8
    return
.end method
```

```
jasmin Simple.jasmine
java Simple

java Simple
java.lang.VerifyError: (class: Simple, method: main signature:
([Ljava/lang/String;)V) Register 2 contains wrong type
Exception in thread "main"
```


Classloader & Verifier Threats

- Class loader reach-over
  - Bypass intended class loader
- Type-confusion
  - Use classes with the same name loaded from different class loaders interchangeably
- Exploit theorem-proving bugs
  - Multiple exploits: interface casts, etc

Protection Domains



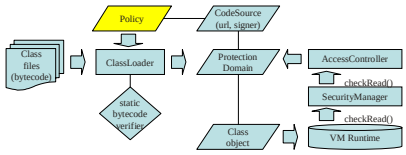
Code Source & Protection Domains

- Permissions granted based on:
  - Code source
  - Code signer
- Policies cover sets of classes with the same source and signer
  - Set forms a “protection domain”
  - Note that this term is overloaded

CodeSource Threats

- Exploiting policies trusting code source
  - Example: browsers trust classes loaded from the file system
  - Attackers introduced class file in browser cache
  - Guessed location of cached file
  - Exploited class loader reach-over to load class from file
  - Attack class had full privileges

Permissions & Policies



Permissions

- Positive permissions only
- Permissions imply other permissions
  - Example: FilePermission(“<<ALL_FILES>>”, “read”) implies FilePermission(“/tmp/foo.txt”, “read”)
- User defined permissions supported

Sample Permissions

- File access
  - java.io.FilePermission “/tmp/*”, “read,write”
  - java.io.FilePermission “${user.home}${/}*”, “read”
- System permissions
  - java.lang.RuntimePermission “getClassLoader”, “”;
- AWT permissions
  - java.awt.AWTPermission “accessEventQueue”, “”;
- Network access
  - java.io.SocketPermission “*:1024-”, “connect”
  - java.io.SocketPermission “*:8080”, “accept,listen”

Policy

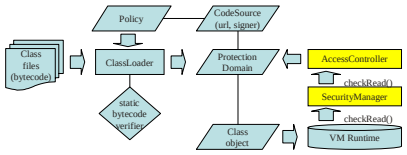
- Grant a set of permissions to classes based on
  - Source (URL)
  - Signer(s)

```
grant { // all classes
  permission java.io.FilePermission “<<ALL_FILES>>”, “read”;
};

grant codeBase “http://www.cs.columbia.edu/~akonstan/java”{ … };

keystore “/appdir/keystore.jks”;
grant signedBy “Alexander, Columbia” { … };
grant signedBy “Alexander”, codeBase “http://www...” { … };
```


Policy Threats

- Difficult to manage
- Sun JVM reads policy at class-load time
- No signature revocation protocol

Security Manager & Access Controller

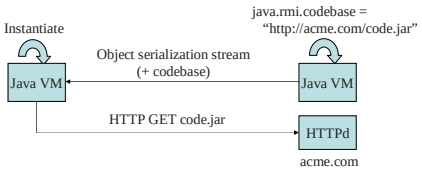


Security Manager

- Focal point of access-control
- Java 2 delegates to AccessController
- Extensible
  - Users can add their own permission classes
- checkPermission(Permission perm)
- checkPermission(Permission perm, Object context)

RMI Security Manager

- Mobile code
  - Serialized objects include codebase URL
  - Client downloads class bytecode from URL
  - Objects instantiated



Access Controller

- Static singleton instance
- Checks access to system resources
  - Based on current security policy
  - Implements stack inspection algorithm
- Marks code as privileged
  - Similar to UNIX set-uid concept
- Obtains "snapshot" of calling context
  - Used to perform out-of-context security checks

Context Access Control Algorithm

- Principle of least privilege
- Grant access iff every protection domain in the current execution context (stack) has that permission

`AccessController.checkPermission()`	system
`SecurityManager.checkPermission()`	system
`SecurityManager.checkRead()`	system
`java.io.FileInputStream(File)`	system
`com.acme.Editor.openFile(String)`	application
`com.acme.Editor.actionPerformed(ActionEvent)`	application
`java.awt.EventDispatchThread`	system

Privileged Operations

- Export restricted services to unauthorized clients
- UNIX setuid concept
- Prevents further stack inspection

```
Object value =
  AccessControlor.doPrivileged(new PrivilegedAction() {
    public Object run() {
      // do some privileged action
      return(value);
    }
  };
```

Thread Context

- New threads inherit parent thread context
- Context snapshot taken at creation time
- Context checking algorithm

Access Control Risks

- Giving code permission to install its own security manager
- Neglecting to invoke the security check
- Writing privileged objects that depend on externally modifiable state

Policy example

Policy Example

- Read protected system property

```
public class PolicyTest {
  public static void main(String[] args) throws Exception {
    System.out.println(System.getProperty("user.name"));
  }
}
```

```
java -Djava.security.manager PolicyTest
java.security.AccessControlException: access denied (java.util.PropertyPermission user.name read)
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:270)
        at java.security.AccessController.checkPermission(AccessController.java:401)
        at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
        at java.lang.SecurityManager.checkPropertyAccess(SecurityManager.java:1291)
        at java.lang.System.getProperty(System.java:572)
        at PolicyTest.main(PolicyTest.java:3)
Exception in thread "main"
```


Policy Example (2)

- Policy file allows locally loaded classes to read all properties starting with “user.”

```
grant codebase "file:" {
  permission java.util.PropertyPermission "user.*", "read";
};
```

```
java -Djava.security.manager -Djava.security.policy=property-read.policy PolicyTest
Alexander
```


Writing Secure Java code

Object Security

- Class security
  - Use private fields, avoid protected, never public
  - Use final classes
    - Avoid subclassing attacks (tradeoff with extensibility)
- Do not return references to mutable objects
  - Examples: arrays, collections
- Keep privileged code short
- Validate de-serialized data
  - Use SignedObject/SealedObject

History of Java Security Bugs

- DNS attack
  - Applet would be served by host whose DNS entry pointed to another address
- Denial of service attacks
  - Threads/Windows/Memory
  - Locking critical objects (e.g. classloader)
- Bytecode verifier/class-loader bugs
  - Create type confusion
  - Combine with other bug to obtain full control

Java security API

Cryptography

- Java Cryptography Architecture (JCA)
  - Interface API
  - Supports different “provider” implementations
- Encryption
  - Symmetric/Asymmetric
- Authentication
  - Message digests, digital signatures

SSL

```
final ServerSocket server =
        SSLServerSocketFactory.getDefault().createServerSocket(8888);

Thread thread = new Thread() {
    public void run() {
        try {
          System.out.println("Waiting for an SSL connection ...");
          Socket socket = server.accept();
          System.out.println("Connection from " + socket.getInetAddress());
        } catch (Throwable e) { e.printStackTrace(); }
        // XXX - no error handling or socket closing!
    }
};
thread.start();

System.out.println("Connecting to local host ...");
Socket socket =
  SSLSocketFactory.getDefault().createSocket("localhost", 8888);
```


SSL (2)

```
keytool -genkey -keyalg RSA -keystore test.jks -dname "CN=Test User"
Enter keystore password:  test123
Enter key password for <mykey>
        (RETURN if same as keystore password):

java -Djavax.net.ssl.trustStore=test.jks
     -Djavax.net.ssl.keyStore=test.jks
     -Djavax.net.ssl.keyStorePassword=test123
     Secure
Connecting to local host ...
Waiting for an SSL connection ...
Connection from /127.0.0.1
```

References

- Java 2 Security Architecture
  - http://java.sun.com/j2se/1.4/docs/guide/security/
- Book References
  - Li Gong, *Inside Java 2 Platform Security*, Addison-Wesley, 1999: Security architecture and rationale.
  - Jess Garms Daniel Somerfield, *Professional Java Security*, Wrox Press, 2001: focus on security APIs and practical security examples
  - Gary McGraw, Edward W. Felten. *Securing Java*, Wiley 1999: general security principles as relating to Java, history of security breaches
  - Alexander V. Konstantinou, et al. *Beginning Java Networking*, Wrox Press, 2001: general Java networking information

References (2)

- Java Jasmin assembler/D-Java disassembler
  - http://mrl.nyu.edu/~meyer/jasmin/
  - http://www.cat.nyu.edu/~meyer/jvm/djava/
- Alternative language Java-VM compilers
  - http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html
- Pieter H. Hartel, Luc Moreau, Formalizing the safety of Java, the Java virtual machine, and Java Card. *ACM Computing Surveys*, v.33, n.4, December 2001
- Java SSL over RMI
  - http://www.cs.columbia.edu/~akonstan/rmi-ssl/