Secure OOP with Java

Lecture Unit - 12

Claudia Maderthaner <claudia.maderthaner@fh-hagenberg.at>

Security Concepts

Security means different things to different people.

- Non-intrusive
- Authenticated
- Encrypted
- Audited
- Well-defined
- Verified
- Well-behaved
- Safe from malevolent programs

Past

- Java Browser Plugin
- Applets
- Webstart

Present

- Web Applications
- Mobile Applications
- Enterprise Applications

Future?

- Container/Cloud
- Microservices
- IoT

Java Language Security

Access Modifiers

`public`

member can be accessed from everywhere

`protected`

member can only be accessed in the package where the class was declared or by any subclass of its class inside or outside its own package

none (default/package private)

member can only be accessed from within its own package

`private`

member can only be accessed in the class where it is declared

⇒ Encapsulation, Information Hiding

Memory Integrity

- Programs cannot access arbitrary memory locations
- Entities declared as `final` must not be changed
- Variables my not be used before the initialized
- Array bounds must be checked on all accesses
- Objects cannot be arbitrarily cast into other objects
- Access methods are strictly adhered to (exemptions: reflection, serialization)

Enforcement of Language Rules

- Compiler enforcement
- Bytecode verifier
- Runtime Enforcement

JVM

- Secure runtime environment
- Bytecode verification
- Bounds checking
- Type safety at runtime
- No unchecked type casts
- No pointer arithmetic
- Automatic memory management

Classloader

- Loads the bytecode of classes during runtime
- All classloaders are derived from `java.lang.ClassLoader`
- Custom classloaders may be implemented
- Sources - some form of byte stream
  - `.class` files
  - Java archives (`.jar`)
  - Data streams over the network (`InputStream`)

Class

Fundamental unit of program code that the Java platform will understand, accept and execute.

— The Well-Grounded Java Developer

Classloader and Namespaces

- Classes loaded through different class loaders
  - are kept in different namespaces
  - may not interact with each other

Example

- Web Applications (Tomcat, JBoss)

Classloader Hierarchy

- Bootstrap Classloader
  - Loads the Java API classes from the bootstrap classpath
  - The bootstrap classloader is no Java class
  - It loads the first classes during startup
- Platform Classloader
- App Classloader

Loading and Linking

- Loading
- Linking
  - Verification
  - Preparation
  - Resolution
  - Initialization

Loading

- Find the byte stream
- Check for valid class file structure (**format checking**)
- Basic checks

Verification

- Confirm, that the class has the correct format (Java specification)
  - check symbolic information in the constant pool
  - ensure that the bytecode is well-behaved
  - make sure that the bytecode doesn't try to manipulate the stack
  - check that every branch instruction has a proper destination instruction
  - make sure every class (except `java.lang.Object`) has a single superclass

Verification (cont.)

- check that final classes are not subclassed
- check methods are called with the right number and type parameters
- check local variables are only assigned suitable types
- check every exception that can be thrown has a legal catch handler
- and others…

Preparation

- Allocating memory
- Getting static variables in the class ready to be initialized

Resolution

- Check that the supertype and implemented interfaces are already linked
- If not then link them before continuing

Initialization

- Initialize static variables
- Run static initialization blocks

⇒ Now the class is fully loaded and ready for use.

Class objects

- Result of the loading and linking process
- Stored in the Java heap
- Represent the newly loaded and linked type at runtime
- May be used with the Java Reflection API

Memory Management

Runtime Areas

- Stack area
- Heap area
- Method area
- PC registers
- Native method stack area

Stack Area

- Stores primitives and references
- Every method call creates a frame on a stack
- Every thread has its on stack

Heap Area

- Dynamic memory allocation
- Stores runtime data like class instances
- De-allocation of objects is handled by the garbage collector

Method Area (Metaspace)

- Runtime code
- Static variables
- Constants pools
- Constructor code

Program Counter (PC) Registers

- Knows sequence of statements to be executed
- Knows which statement is currently executed
- Holding address of the instruction being executed next in its thread
- Every thread has its own PC register

Native Method Stack

aka C stack

- For the native code that is executed

Garbage Collection

- Reduces memory leaks (but does not completely prevent it)
- Allocated memory may not be used by other objects

Java Sandbox

Elements of the Java Sandbox

- Permissions
- Code sources
- Protection domains
- Policy files
- Keystores

Policy Files

```
grant signedBy "signer_names", codeBase "URL",
    principal principal_class_name "principal_name",
    principal principal_class_name "principal_name",
    ... {

  permission permission_class_name "target_name", "action", signedBy "signer_names";
  permission permission_class_name "target_name", "action", signedBy "signer_names";
  ...
};
```

Policy Files

```
keystore "http://foo.bar.com/blah/.keystore";

grant signedBy "Tony" {
  permission java.io.FilePermission "/tmp/*", "read,write";
};

// Grant everyone the following permission:
grant {
  permission java.util.PropertyPermission "java.vendor", "read";
};

grant codeBase "file:/home/sysadmin/-" {
  permission java.security.SecurityPermission "Security.insertProvider.*";
  permission java.security.SecurityPermission "Security.removeProvider.*";
};
```

Default Policy Files

Policy configuration

`${java.home}/conf/security/java.security`

- `${java.home}/conf/security/java.policy`
- `${user}/.java.policy`

Permissions

- AllPermisson
- AudioPermission
- AWTPermission
- FilePermission
- NetPermission
- PropertyPermission
- ReflectPermission
- RuntimePermission
- SecurityPermission
- SerializablePermission
- SocketPermission
- SQLPermission

Security Manager

Enable Security Manager with default configuration

```
java -Djava.security.manager SomeApp
```

Enable Security Manager with additional configuration file

```
java -Djava.security.manager -Djava.security.policy=<Path|URL> SomeApp
```

Enable Security Manager **only** with configuration file

```
java -Djava.security.manager -Djava.security.policy==<Path|URL> SomeApp
```

⚠ The Security Manager is **not** activated by default!

Keys and Certificates

Keystore

- File holding a set of keys and certificates

`alias`	short key-specific name
`DN`	distinguished name, subset of full X.500 name `CN=John Doe, OU=Doe, O=Doe Inc, L=Hagenberg, S=Upper Austria, C=AT`
Key entry	asymmetric key pair or single secret key
Certificate entry	only public key

`cacerts`

- Trusted certificate authorities
- `${java.home}/lib/security/cacerts`

Signing

keytool genkeypair

```
keytool -genkeypair -alias signFiles -keyalg RSA -keysize 2048 \
-keystore susanstore -storepass ab987c
```

`-alias <alias>`	alias name of the entry to process
`-keyalg <alg>`	key algorithm name
`-keysize <size>`	key bit size
`-keystore <keystore>`	keystore name
`-storepass <arg>`	keystore password

keytool export

```
keytool -export -keystore susanstore -alias signFiles -file SusanJones.cer
```

`-alias <alias>`	alias name of the entry to process
`-file <file>`	output file name
`-keystore <keystore>`	keystore name

keytool import

```
keytool -import -alias susan -file SusanJones.cer -keystore raystore
```

`-alias <alias>`	alias name of the entry to process
`-file <file>`	input file name
`-keystore <keystore>`	keystore name

keytool printcert

```
keytool -printcert -file SusanJones.cer
```

`-file <file>`	input file name

keytool list

```
keytool -list -keystore raystore
```

-keystore <keystore>	keystore name

jarsigner

```
jarsigner -keystore susanstore -signedjar sCount.jar Count.jar signFiles
```

-keystore <keystore>	keystore name
-signedJar <jar>	name of signed jar file

Java API

Java Security Libraries

JCA	encryption/decryption
	• Java Cryptography Architecture
JSSE	network connections/SSL protocols
	• Java Secure Socket Extension
JAAS	authentication/authorization
	• Java Authentication and Authorization Service

Java Cryptography Architecture (JCA)

- Provider architecture
- A set of APIs for
  - digital signatures,
  - hashes,
  - certificates and certificate validation,
  - encryption (symmetric/asymmetric block/stream ciphers),
  - key generation and management, and
  - secure random number generation

JCA Features

- Implementation independence
- Implementation interoperability
- Algorithm extensibility

Provider Architecture

- Cryptographic Service Providers

```
MessageDigest.getInstance("SHA-256");
MessageDigest.getInstance("SHA-256", "SpecificProvider");
```

Java Authentication and Authorization Service (JAAS)

- Authentication
- Authorization

JAAS entities

Subject

the source of the request

Principal

represents subject identities

Authentication

1. Instantiate `LoginContext`
2. Load `LoginModules` configured
3. Call login in `LoginContext`
4. Invoke `LoginModules`
5. Return authentication status
6. Retrieve subject from `LoginContext`

LoginModule

- `login()`
- `commit()`
- `abort()`
- `logout()`

Challenges

Dynamic Class Loading

Late Binding

Serialization/Deserialization

- Object serialization allows an object to be written as a series of bytes
- Those bytes can be deserialized in another running application creating an object with the same state
- For serialization and deserialization to work, it also needs access to private variables
- Only objects from classes implementing `java.io.Serializable` can be serialized

Contact

Moodle Discussion Board

claudia.maderthaner@fh-hagenberg.at