

Programming Securely II

Computer Security Lecture 13

David Aspinall

School of Informatics
University of Edinburgh

5th March 2012

Outline

Web security issues

Java Security: Coding and Models

Trusting code

Language futures for security

Programming and Security

Programming Securely To develop code in a secure manner so that the code itself is not a vulnerability that can be exploited by an attacker.

Programming Security To develop code for security-specific functions such as encryption, digital signatures, firewalls, etc.

In this lecture, we look at both sides:

- ▶ continuing programming securely: some **web application** security issues and some **Java** guidelines.
- ▶ programming security: overview of **Java security APIs** and **current and future trust models**.

Web security: client-side threats

- ▶ Risky treatment of **MIME-types**: e.g., shell-escapes in troff. By design, downloaded **active content** (e.g., Java, ActiveX controls) should run in a restricted environment. Problems come when restrictions fail, or aren't tight enough.
- ▶ SSL issues: revoked certificates, spoofed site names, mixed encrypted/unencrypted pages.
- ▶ Browsers store **cookies** which have confidentiality implications. Even without cookies, web browsing is less anonymous than it feels: information is stored in browser's history and document cache, firewall and proxy logs, and the remote sites visited, even before any spyware is present. (All great for market researchers).
- ▶ Untrained users unwittingly make bad security decisions.
- ▶ Buggy browsers: buffer overflows, crypto bugs, etc.

Web security: server-side threats

- ▶ Access control: should prevent certain files being served.
- ▶ Complex or malicious URLs
- ▶ Denial of service attacks
- ▶ Remote authoring and administration tools
- ▶ Buggy servers, with attendant security risks
- ▶ Server-side **scripting languages**: **C or shell CGI, PHP, ASP, JSP, Python, Ruby**, all have serious security implications in configuration and execution. File systems and permissions have to be carefully designed. That's before any implemented *web application* is even considered. . .

Web programming: application security

Many issues (some of which are introduced in the practical).

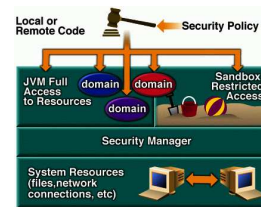
- ▶ **Input validation**: to prevent SQL injection, command injection, other confidentiality attacks. *Ajax: beware client-side validation!* Understand metacharacters at every point. Use labels/indexes for hidden values, not values themselves.
- ▶ **Output filtering**: **cross-site scripting (XSS)**, when attacker-generated HTML appears on site: used for **session hijacking, phishing attacks**. Beware passing informative error messages.
- ▶ **Careful cryptography**: encryption/hashing to *protect server state in client*, use of *appropriate authentication mechanisms* for web accounts (never Referer header).

Java Secure Coding Guidelines

- ▶ **Using modifiers.** Reduce scope of methods and fields; beware non-final public static (*global*) variables; avoid public fields, and add security checks to public accessors.
- ▶ **Protecting packages.** Stop insertion of untrusted classes in a package using java.security properties or “sealed” JAR file; avoid package-level access.
- ▶ **Beware mutable objects.** Returning or storing mutables may be risky, if caller then updates them; use immutable or *cloned* objects instead.
- ▶ **Serialization.** Once serialized, objects are outside JVM security. Designate **transient** fields and encrypt/sign persistent data. Beware overriding of serialization methods (among others).
- ▶ **Clear sensitive information.** Store sensitive data in mutable objects, then clear explicitly ASAP, to prevent heap-inspection attacks. Can't rely on Java's garbage collection to do this.

Access Control in Java

Java 1.0 had a **sandbox** security model, where downloaded Java applets ran in a restricted environment with no access to local files, etc: often too restrictive. Java 2 has a more flexible, fine-grained level of control:

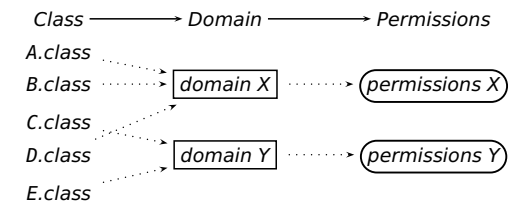


This picture is reproduced from the Java Security Tutorial (c) Sun

Applications and applets are subject to a **security policy** which specifies **protection domains** based on **location** of code, whether it is **signed** by a trusted entity, and the user **identity**. Each domain specifies a set of **permissions** for accessing resources.

Java security architecture

- ▶ A SecurityManager is installed by web browsers for Java applets; an application must either itself install the security manager, or be invoked with the option -Djava.security.manager. If the security manager's checks fail, a java.lang.SecurityException is raised.
- ▶ Access control in Java is based on **protection domains** which group together the set of objects which are currently accessible by a principal.



Java access control permissions

- ▶ Domains are associated with sets of **permissions**

java.security.AllPermission	every resource
java.io.FilePermission	file system access
java.net.SocketPermission	accept/connect based on host/IP
java.awt.AWTPermission	window-system permissions
java.lang.RuntimePermission	JVM config; threads; printing
java.security.SecurityPermission	accessing security policy, key store

- ▶ Some are associated with **target** and **actions**:

```
import java.io.FilePermission;
FilePermission p1 = new
FilePermission("/tmp/myfile", "read");
FilePermission p2 = new
FilePermission("/tmp/*", "read");
```

Permissions implement an **implies** method for access control decisions. Here p2.implies(p1).

Java security policies

- ▶ The system security policy for a Java application environment specifies permissions available for code from various sources, represented by a Policy object. Only one in effect at a time.
- ▶ A Policy object evaluates the global policy using the ProtectionDomain for a class, and returns an appropriate Permissions object.
- ▶ Java supplies a GUI **policytool** utility for editing ASCII format policy files, with entries like this, specifying a key store and zero or more “grant” entries:

```
keystore "keystore", "JKS";
grant principal com.sun.security.auth.UnixPrincipal "da" {
    permission java.util.PropertyPermission "java.home", "read";
    permission java.io.FilePermission "/tmp/foo", "read,write";
};
```

Default, system policy is in **javahome/lib/security/java.policy**. User policy is in **userhome.java.policy**.

Java security extensions

- ▶ The Java security extensions add additional APIs for programming security features.
- ▶ **Java Cryptography Extension (JCE)**
A Java framework for cryptographic functionality, including message digests, encryption, signing, and X.509 certificates.
- ▶ **Java Secure Socket Extension (JSSE).**
- ▶ **Java Authentication and Authorization Service (JAAS).** Used for “reliable and secure” authentication of users, to determine who is currently executing Java code; and for authorization of users to ensure they have the permissions necessary for desired actions.
- ▶ **Java GSS-API.** Bindings for Generic Security Service API (RFC2853). Used for securely exchanging messages between communicating applications, using various underlying mechanisms (e.g., Kerberos).

Java Cryptography Extension (JCE)

- ▶ Crypto framework. A *provider* plug-in architecture allows multiple simultaneous implementations. Inclusion restricted because of import/export restrictions.
- ▶ Has *algorithm independence*, clients don't need to understand algorithms; abstract "engine" classes provide different services.
- ▶ *Service provider interfaces* (SPIs) added statically or dynamically; clients query installed providers to find out supported services. JVM and clients specify preference orders.
- ▶ Key management is through a "keystore" database. Different providers may have different formats.
- ▶ SUN provider implements common formats and proprietary keystore type JKS.
- ▶ See: `javax.crypto`, `javax.crypto.interfaces`, `javax.crypto.spec`.

JCE cryptography services

- ▶ A cryptography service is associated with a particular algorithm or type, and manipulates or generates data, keys, algorithm parameters, keystores, or certificates.
- ▶ Engine classes include:

<code>MessageDigest</code>	generate message digests (MDCs)
<code>Signature</code>	sign data and verify digital signatures.
<code>KeyPairGenerator</code>	generate public-private key-pair.
<code>CertificateFactory</code>	create certificates and CRLs.
<code>KeyStore</code>	create and manage key databases.
<code>AlgorithmParameters</code>	manage parameters for an algorithm.
<code>SecureRandom</code>	random or pseudo-random numbers.
- ▶ Factory methods in engine classes are used to return instances of the class, e.g.
`Signature.getInstance("SHA1withDSA").`

Java Secure Socket Extension (JSSE)

- ▶ The JSSE is also based on a provider plug-in architecture.
- ▶ Has a simple structure. Main use is with SSL client sockets, SSL server sockets, and SSL session handles. Sample classes:

<code>SSLSocket</code>	socket for SSL/TLS/WTLS protocols
<code>SSLSocketFactory</code>	factory for <code>SSLSocket</code> objects
<code>SSLServerSocket</code>	server socket for SSL/TLS/WTLS
<code>... Factory</code>	factory for <code>SSLServerSockets</code>
<code>SSLSession</code>	encapsulation of SSL session (an in
- ▶ Creating SSL client or server sockets is as easy as creating ordinary Java TCP/IP sockets: each SSL class extends the corresponding ordinary TCP socket class, and provides a few extra hooks for setting security parameters.
- ▶ See `javax.net.ssl`, also `javax.net` and `javax.security.cert`.

Authentication and Authorization (JAAS)

- ▶ JAAS has a pluggable architecture; applications independent of underlying authentication methods. Implementation is decided at runtime, in a **login configuration file**.
- ▶ A Subject may have multiple identities; each is a `Principal` (name). Subjects own public and private **credentials** (e.g., key material).
- ▶ To authenticate, a `LoginContext` object is created, which then consults a configuration to load the required `LoginModules`. To authenticate a subject the `login` method is invoked for each module.
- ▶ **Authorization** happens when a subject is associated with a thread's `AccessControlContext` using the **doAs** methods for performing actions (`java.security.PrivilegedAction.run`). Then principal-based entries in the current security policy are used.

Flaws in the Java infrastructure

- ▶ Java was touted from the start as a secure mechanism for mobile code. But it has suffered from flaws in both design and implementation, surveyed in 1999 by McGraw and Felten in *Securing Java*, see <http://www.securingjava.com>.
- ▶ Most fundamental are any problems in the **Byte Code Verifier**, which checks proper use of JVMIL (protecting against "malicious" or merely buggy compilers):
 - ▶ no operand stack overflow/underflow
 - ▶ correct types and conversions
 - ▶ field accesses obey visibility modifiers
- ▶ **Type safety** relies on byte code verification being correct. Unfortunately getting this right is complicated...

Flaws in Java itself – continued

- ▶ The Java Language Specification is written in English. It suffers from usual problems of large language specifications: missing details, ambiguity, and other inaccuracies.
 - ▶ Sun BUG ID 6360463 (Dec 05): "offset item of the stack map frame" not defined in specification ... "renders most of discussion on type checking moot"
- ▶ Sun's implementations are usually taken as the reference behaviour. But these have had a series of type safety and access control failings (from 1.x SDKs to J2ME in mobile phone KVMs).
 - ▶ 8th Feb 2006, CVE-2006-0614,0615,0617: Sun fixes seven vulnerabilities in current JREs which allowed remote code to bypass sandbox using reflection.
- ▶ Shows *defence in depth* is important; even with a careful Java security policy restricting what downloaded code can do, you should still beware *untrusted* code.

The Trusted Computing Base

Trusted Computing Base (TCB)

The set of all components (hardware, software, human, ...) whose correct functioning is sufficient to ensure that the security policy is enforced.

- ▶ Equivalently: failure of the TCB causes failure of security.
Misplaced trust can hurt you!
- ▶ This motivates design principles for the TCB:
 - ▶ make it as small as possible
 - ▶ do not change it often
 - ▶ verify it carefully: so it is as secure as possible
- ▶ In access control systems, the TCB is the **Reference Monitor** implementation.

Palladium/TCPA/NGSCB/Trustworthy Computing

- ▶ PCs now contain a **Trusted Platform Module** (TPM) security chip with embedded master keys.
- ▶ Security model idea: PC boots, hashing BIOS, OS and application code. Builds a chain of trust.
- ▶ Protection domains in OS extended into hardware (**secure keyboard reading, sound channels**). Desire: close down an open system (cf XBox).
- ▶ Allows certificates, e.g. *"this document created with v 1751 of MS Word, on Windows Vista Trusted, 27th August 2008, on Dell Megaplex ZZ5 S/N 5091237896"*. Files stored encrypted, cannot be decrypted on other machines.
- ▶ Many uses. Strong anti-privacy measures. Business clients: financial services, government, and healthcare. Home PC users: reduction in spyware, digital rights management (DRM). New uses: renting, lending, time-limited, etc. Considerable controversy (Stallman: "Treacherous Computing").

Language-based security

An active research area: applying programming language theory, designing new constructs and mechanisms.

Most work applies verification technology including static analysis, extended type systems and theorem proving.

- ▶ Proof-carrying code (PCC), which equips code with independently checkable safety certificates.
- ▶ Cyclone, Vault and others.
Add richer, safer and more expressive typing and annotations to existing languages.
- ▶ Other security specialised typing includes:
 - ▶ detecting and preventing illegal **information flows**
 - ▶ **ensuring authentication** before authorisation
 - ▶ fixing patterns of **access control**, e.g. close file after opening.

References

- 📖 Mark G. Graff and Kenneth R. van Wyk.
Secure Coding: Principles & Practices.
O'Reilly, 2003.
- 📖 Sverre H. Huseby.
Innocent Code: a security wake-up call for web programmers.
Wiley.
- 📖 Gary McGraw.
Securing Java.
John Wiley & Sons, 1999.

Recommended Reading

For web programming: Huseby's book, or the more recent information at OWASP, <https://www.owasp.org>.
For Java security: the Oracle/CERT guidelines at <https://www.securecoding.cert.org>