18

Secure Coding Guidelines

Objectives

After completing this lesson, you should be able to:

- Describe the Java SE security overview
- Explain vulnerabilities
- Describe the secure coding guidelines and antipatterns

Java SE Security Overview

The Java language and virtual machine provide many features to mitigate common programming mistakes.

- The language is type-safe, and the runtime provides automatic memory management and bounds-checking on arrays.
- Java programs and libraries check for illegal state at the earliest opportunity.
- To minimize the likelihood of security vulnerabilities caused by programmer error, Java developers should adhere to recommended coding guidelines.

Java SE Security Overview

- Runtime security
  - Behavior enforced by Java runtime
  - Enables application to be run safely in a restricted environment
- Security APIs
  - Standard APIs for Cryptography, PKI, Authentication, Policy, and secure communication
  - Pluggable implementations
- Security tools
  - `keytool,jarsigner,policytool`

Secure Coding Guidelines

Secure coding guidelines provide a more complete set of security-specific coding guidelines targeted at the Java programming language.

- The guideline is organized into nine sections:

Guidelines	
0	Fundamentals
1	Denial of Service
2	Confidential Information
3	Injection and Inclusion
4	Accessibility and Extensibility
5	Input Validation
6	Mutability
7	Object Construction
8	Serialization and deserialization
9	Access control

Guidelines address the top-level concerns when writing secure code.

Note: This lesson doesn't cover the Access control guideline, and you can refer to the details from this tutorial.

Secure Coding Guidelines for Java SE ,

https://www.oracle.com/technetwork/java/seccodeguide-139067.html

While sections 0 through 3 are generally applicable across different types of software, most of the guidelines in sections 4 through 9 focus on applications that interact with untrusted code (though some guidelines in these sections are still relevant for other situations). Developers should analyze the interactions that occur across an application's trust boundaries and identify the types of data involved to determine which guidelines are relevant. Performing threat modeling and establishing trust boundaries can help to accomplish this.

These guidelines are intended to help developers build secure software, but they do not focus specifically on software that implements security features. Therefore, topics such as cryptography are not covered in this document (for information on using cryptography with Java). While adding features to software can solve some security-related problems, it should not be relied upon to eliminate security defects.

Vulnerabilities

- Definition:
  - A flaw or weakness that could be exploited to violate the systems security policy.
- Causes:
  - Design: Faulty assumptions in the application architecture
  - **Implementation: Insecure programming practices (antipatterns)**
  - Composition and setup: Errors in configuration

Secure Coding Antipatterns

An antipattern is a programming practice that you should avoid.

- May look beneficial in the first but may result in bad consequences
- For example: Implementing speed-optimized methods that don't validate input parameters

Antipatterns in Java

Common misconceptions lead to a larger attack surface.

- Neglecting to verify valid input formatting
- Granting unnecessary permissions to code
- Misusing mutable public static variables
- Ignoring changes to superclasses
- Assuming exceptions are harmless
- Assuming the value space of integers is unbounded
- Assuming that a constructor exception destroys the object

Fundamentals

Fundamentals	
0	Prefer to have obviously no flaws rather than no obvious flaws
1	Design APIs to avoid security concerns
2	Avoid duplication
3	Restrict privileges
4	Establish trust boundaries
5	Minimize the number of permission checks
6	Encapsulate methods, fields, and classes to coherent sets of behavior

Why should I care?

- Security vulnerabilities will never be eliminated.
- Well-designed and tested code is easier to secure.
- Least privilege by design makes the code more secure.

Fundamentals

0-1: Design APIs to avoid security concerns

- It is better to design APIs with security in mind.
- Trying to retrofit security into an existing API is more difficult and error prone.
- For example, making a class final prevents a malicious subclass from:
  - Adding finalizers
  - Cloning
  - Overriding random methods
  - Calling protected methods

0-2 : Avoid duplication

- Key characteristic of secure code is to maximize reuse.
  - Duplication of code and data causes many problems. Both code and data tend not to be treated consistently when duplicated; for example, changes may not be applied to all copies.

0-3: Restrict privileges

- Not all coding flaws will be eliminated even in well-reviewed code.
- Whenever possible use the principle of least privileges.
- Reduced privileges means reduced potential impact of exploits
- Apply the Java security mechanisms:
  - Statically by restricting permissions through policy files.
  - Dynamically with `java.security.AccessController.doPrivileged`

0-4: Establish trust boundaries

- Having simple APIs to distinguish clearly between inside and outside of trust boundaries
  - In order to ensure that a system is protected, it is necessary to establish trust boundaries.
  - Data that crosses these boundaries should be sanitized and validated before use.
  - Trust boundaries are also necessary to allow security audits to be performed efficiently.

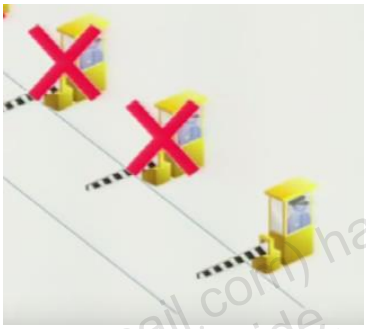
Code that ensures integrity of trust boundaries must itself be loaded in such a way that its own integrity is assured.

For instance, a web browser is outside of the system for a web server. Equally, a web server is outside of the system for a web browser. Therefore, web browser and server software should not rely upon the behavior of the other for security.

When auditing trust boundaries, there are some questions that should be kept in mind. Are the code and data used sufficiently trusted? Could a library be replaced with a malicious implementation? Is untrusted configuration data being used? Is code calling with lower privileges adequately protected against?

0-5: Minimize the number of permission checks

- Prefer a point of access
- After initial check, provide clients with (immutable) capability objects.



Fundamentals

0-6: Encapsulate

- Group coherent functionality
- Do not expose implementation details.
- Have a simple and stable public API

Allocate behaviors and provide succinct interfaces. Fields of objects should be private and accessors avoided. The interface of a method, class, package, and module should form a coherent set of behaviors, and no more.

Fundamentals: Why Should I Care?

0-7: Document security-related information

- API documentation should cover security-related information such as required permissions, security-related exceptions, and any preconditions or postconditions that are relevant to security.
- Documenting this information in comments for a tool such as Javadoc can also help to ensure that it is kept up to date.

Allocate behaviors and provide succinct interfaces. Fields of objects should be private and accessors avoided. The interface of a method, class, package, and module should form a coherent set of behaviors, and no more.

Denial of Service

Input into a system should be checked so that it will not cause excessive resource consumption disproportionate to that used to request the service. Common affected resources are CPU cycles, memory, disk space, and file descriptors.

1- Denial of Service	
1-1	Beware of activities that may use disproportionate resources
1-2	Release resources in all cases
1-3	Resource limit checks should not suffer from integer overflow

Why should I care?

- Plan against denial-of-service during design phase.

Denial of Service

1-1: Beware of activities that may use disproportionate resources

- Examples of attacks include:
  - Large image processing
  - Integer overflows
  - Complex object graphs errors can cause sanity checking of sizes to fail
  - Careless decompression, i.e. “Zip bombs," whereby a short file is very highly compressed
  - "Billion laughs attack“ such as that caused by XXE, XML external entity
  - Parsing and processing complex grammars(XPATH,RegEx)
  - Deserialization processing anomalies
  - Logging with inappropriate detail level
  - Parsing corner cases that cause infinite loops

Requesting a large image size for vector graphics. For instance, SVG and font files.

Integer overflow errors can cause sanity checking of sizes to fail.

An object graph constructed by parsing a text or binary stream may have memory requirements many times that of the original data.

"Zip bombs" whereby a short file is very highly compressed. For instance, ZIPs, GIFs, and gzip-encoded HTTP contents. When decompressing files, it is better to set limits on the decompressed data size rather than relying upon compressed size or meta-data.

"Billion laughs attack" whereby XML entity expansion causes an XML document to grow dramatically during parsing. Set the XMLConstants.FEATURE_SECURE_PROCESSING feature to enforce reasonable limits.

Causing many keys to be inserted into a hash table with the same hash code, turning an algorithm of around O(n) into $O(n^2)$.

Regular expressions may exhibit catastrophic backtracking.

XPath expressions may consume arbitrary amounts of processor time.

Java deserialization and Java Beans XML deserialization of malicious data may result in unbounded memory or CPU usage.

Detailed logging of unusual behavior may result in excessive output to log files.

Infinite loops can be caused by parsing some corner case data. Ensure that each iteration of a loop makes some progress.

Denial of Service

1-3: Resource limit checks should not suffer from integer overflow

Antipattern: Believing the value space of integers is unbounded

- Java language provides bound checking on all arrays.
  - Mitigates the vast majority of integer overflow attacks
- However, all the primitive integer types silently overflow.
  - Potential bypass of Java-level validity checks of native code
  - Causing memory corruption (out of bounds writes)
- Replace suspicious checks

```
private void checkGrowBy(long extra) {
    if (extra < 0 || current > max - extra) {
                    throw new IllegalArgumentException();
    }
}
```

If(current+extra >max) //bad

Some checking can be rearranged so as to avoid overflow. With large values, current + max could overflow to a negative value, which would always be less than max.

A peculiarity of two's complement integer arithmetic is that the minimum negative value does not have a matching positive value of the same magnitude. So, `Integer.MIN_VALUE == -Integer.MIN_VALUE`, `Integer.MIN_VALUE == Math.abs(Integer.MIN_VALUE)` and, for integer a, a < 0 does not imply -a > 0. The same edge case occurs for `Long.MIN_VALUE`.

As for Java SE 8, the `java.lang.Math` class also contains methods for various operations (addExact, multiplyExact, decrementExact, etc.) that throw an `ArithmeticException` if the result overflows the given type.

Confidential Information

- Confidential data should be readable only within a limited context.
- Data that is to be trusted should not be exposed to tampering.
- Privileged code should not be executable through intended interfaces.

2- Confidential Information	
2-1	Purge sensitive information from exceptions
2-2	Do not log highly sensitive information
2-3	Consider purging highly sensitive information from memory after use

Why should I care?

- Prevent malicious information gathering of your configuration settings and passwords.

Confidential Information

2-1: Purge sensitive information from exceptions

- Exception objects may convey sensitive information. For example:
  - `java.io.FileInputStream`, to read an underlying configuration file
  - `java.io.FileNotFoundException`, to probe the file system
- Verbose debugging is good but not in production.
  - Consider separating output channels
  - Reuse a good logging framework

Confidential Information

2-2: Don’t log highly sensitive information

- Have a security policy in place.
- Encrypt passwords with standard APIs.

Some information, such as Social Security numbers (SSNs) and passwords, is highly sensitive. This information should not be kept for longer than necessary nor where it may be seen, even by administrators. For instance, it should not be sent to log files, and its presence should not be detectable through searches. Some transient data may be kept in mutable data structures, such as char arrays, and cleared immediately after use. Clearing data structures has reduced effectiveness on typical Java runtime systems as objects are moved in memory transparently to the programmer.

This guideline also has implications for implementation and use of lower-level libraries that do not have semantic knowledge of the data they are dealing with. As an example, a low-level string parsing library may log the text it works on. An application may parse an SSN with the library. This creates a situation where the SSNs are available to administrators with access to the log files.

Confidential Information

2-3:Consider purging highly sensitive from memory after use

- Limit exposure time in memory:
  - Delete as soon as possible.
  - Don't depend on garbage collection.
  - Use `char[]` arrays to clear the traces.
  - Keep the information local.

To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism.

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

Injection and Inclusion

A very common form of attack involves causing a particular program to interpret data crafted in such a way as to cause an unanticipated change of control. Typically, but not always, this involves text formats.

3- Injection and Inclusion	
3-1	Generate valid formatting
3-2	Avoid dynamic SQL
3-3	XML and HTML generation requires care
3-4	Avoid any untrusted data on the command line
3-5	Restrict XML inclusion
3-6	Care with BMP files
3-7	Disable HTML display in Swing components
3-8	Take care in interpreting untrusted code

Why should I care?

- Protect data integrity:

  Validate data from untrusted sources.

Injection and Inclusion

3-1: Generate valid formatting

Antipattern: Neglecting to verify valid input formatting.

- Validate input:
  - Check for out-of-bounds values and escape characters.
  - Regular expressions can help validate String inputs.
  - Pass only validated inputs to subcomponents.
  - Re-use well-tested libraries instead of ad-hoc code.

Guideline 3-2 / INJECT-2: Avoid dynamic SQL

It is well known that dynamically created SQL statements including untrusted input are subject to command injection. This often takes the form of supplying an input containing a quote character (') followed by SQL. Avoid dynamic SQL.

For parameterized SQL statements using Java Database Connectivity (JDBC), use java.sql.PreparedStatement or java.sql.CallableStatement instead of java.sql.Statement.

Guideline 3-3 / INJECT-3: XML and HTML generation requires care

Untrusted data should be properly sanitized before being included in HTML or XML output. Failure to properly sanitize the data can lead to many different security problems, such as Cross-Site Scripting (XSS) and XML Injection vulnerabilities. There are many different ways to sanitize data before including it in output. Characters that are problematic for the specific type of output can be filtered, escaped, or encoded.

Guideline 3-4 / INJECT-4: Avoid any untrusted data on the command line

When creating new processes, do not place any untrusted data on the command line. Behavior is platform-specific, poorly documented, and frequently surprising. Malicious data may, for instance, cause a single argument to be interpreted as an option (typically a leading—on Unix or / on Windows) or as two separate arguments. Any data that needs to be passed to the new process should be passed either as encoded arguments (e.g., Base64), in a temporary file, or through a inherited channel.

Accessibility and Extensibility

The task of securing a system is made easier by reducing the "attack surface" of the code.

Accessibility and Extensibility	
4-1	Limit the accessibility of classes, interfaces, methods, and fields
4-2	Limit the accessibility of packages
4-3	Isolate unrelated code
4-4	Limit exposure of ClassLoader instances
4-5	Limit the extensibility of classes and methods
4-6	Understand how a superclass can affect subclass behavior

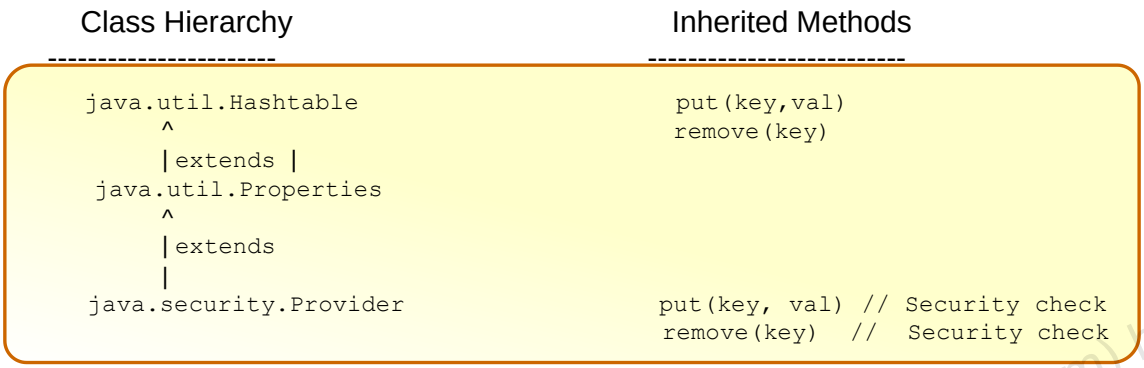
Why should I care?

- Reduce the attack surface.
- Assign the least accessibility required for your code.
- Prevent unwanted modifications of your code.

Accessibility and Extensibility

4-6:Understand how a superclass can affect subclass behavior

Consider the following example that occurred in JDK 1.2:

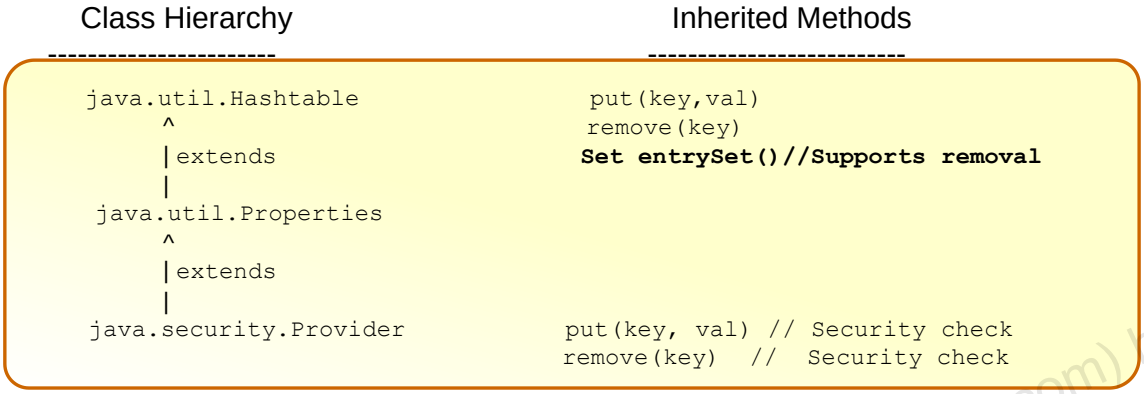


To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism.

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

4-6: Antipattern-Ignoring changes to superclasses

Attacker bypasses `remove` method and uses inherited `entrySet` method to delete properties.



To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism.

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

Accessibility and Extensibility

4-6:Understand how superclass effects the behavior of a subclass

- Subclasses don't guarantee encapsulation.
  - Subclasses can add new methods.
  - Subclasses may modify behavior of methods that have not been overridden.
- Security checks enforced in subclasses can be bypassed.
  - `Provider.remove` security check is bypassed if attacker calls newly inherited `entrySet` method to perform removal.

To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism.

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

Accessibility and Extensibility

4-6:Understand how superclass effects the behavior of a subclass

- Avoid inappropriate subclassing
  - Subclass only when the inheritance model is well specified and understood
  - When in doubt, use composition instead of inheritance
- Monitor changes to superclasses
  - Identify behavorial changes to existing inherited methods and override if necessary
  - Identify new methods and override if necessary

```
java.security.Provider           put(key, val) // Security check
                                 remove(key)  //  Security check
                /*override*/     Set entrySet() //immutable set
```

To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism.

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

Input Validation

Validating external inputs is an important part of security.

5- Input Validation	
5-1	Validate inputs
5-2	Validate output from untrusted objects as input
5-3	Define wrappers around native methods

Why should I care?

- Invalid input may trick harmless code into malicious behavior.
- Enables to prevent attackers from modifying the control flow

Guideline 5-1 / INPUT-1: Validate inputs

Input from untrusted sources must be validated before use. Maliciously crafted inputs may cause problems, whether coming through method arguments or external streams. Examples include overflow of integer values and directory traversal attacks by including "../" sequences in filenames. Ease-of-use features should be separated from programmatic interfaces.

Guideline 5-2 / INPUT-2: Validate output from untrusted objects as input

In general method, arguments should be validated but not return values. However, in the case of an upcall (invoking a method of higher level code) the returned value should be validated. Likewise, an object only reachable as an implementation of an upcall need not validate its inputs.

Guideline 5-3 / INPUT-3: Define wrappers around native methods

Java code is subject to runtime checks for type, array bounds, and library usage. Native code, on the other hand, is generally not. While pure Java code is effectively immune to traditional buffer overflow attacks, native methods are not. To offer some of these protections during the invocation of native code, do not declare a native method public. Instead, declare it private and expose the functionality through a public Java-based wrapper method. A wrapper can safely perform any necessary input validation prior to the invocation of the native method.

Mutability

Mutability	
6-1	Prefer immutability for value types
6-2	Create copies of mutable output values
6-3	Create safe copies of mutable and subclassable input values
6-4	Support copy functionality for a mutable class
6-5	Do not trust identity equality when overridable on input reference objects
6-6	Treat passing input to untrusted object as output
6-7	Treat output from untrusted object as input
6-8	Define wrapper methods around modifiable internal state
6-9	Make public static fields final
6-10	Ensure public static final field values are constants
6-11	Do not expose mutable statics
6-12	Do not expose modifiable collections

Why should I care?

- Runtime security relies on trustworthy objects used by privileged code.
- Don't give chance to attackers to modify those on their behalf.

Mutability

6-8: Define wrapper methods around modifiable internal state

If a state that is **internal** to a class must be publicly accessible and modifiable, declare a private field and enable access to it via public wrapper methods.

If the state is only intended to be accessed by **subclasses**, declare a private field and enable access via protected wrapper methods. Wrapper methods allow input validation to occur prior to the setting of a new value.

```
public final class WrappedState {
   // private immutable object
      private String state;

     // wrapper method
     public String getState() {
     return state;
    }
    // wrapper method
    public void setState(final String newState) {
      this.state = requireValidation(newState);
    }

private static String requireValidation(final String state) {
    if (...) {
     throw new IllegalArgumentException("...");
     }
      return state;
     }
  }
```

To narrow the window when highly sensitive information may appear in core dumps, debugging, and confidentiality attacks, it may be appropriate to zero memory containing the data immediately after use rather than waiting for the garbage collection mechanism.

However, doing so does have negative consequences. Code quality will be compromised with extra complications and mutable data structures. Libraries may make copies, leaving the data in memory anyway. The operation of the virtual machine and operating system may leave copies of the data in memory or even on disk.

Mutability

6-9: Make public static fields as final

Antipattern: Misusing mutable public static variables

- Always declare public static fields as final.
  - Callers can trivially access and modify public nonfinal static fields
  - Neither accesses nor modifications can be guarded against, and newly set values cannot be validated

```
public class Files {
            public static final String separator = "/";
            public static final String pathSeparator = ":";
}
```

- Treat public statics primarily as constants.
  - Consider using enums(type-safe and implicitly static final).

Sensitive static code can be modified by untrusted code.

Static variables are global across a Java runtime environment.

Treat public statics primarily as constants-

Consider using enums(type-safe and implicitly static final).

Object Construction

7- Object Construction	
7-1	Avoid exposing constructors of sensitive classes
7-2	Prevent the unauthorized construction of sensitive classes
7-3	Defend against partially initialized instances of nonfinal classes
7-4	Prevent constructors from calling methods that can be overridden

Why should I care?

- Stay in charge of creation of critical object instances.
- Don't let attackers control critical classes.

Guideline 7-1 / OBJECT-1: Avoid exposing constructors of sensitive classes

Construction of classes can be more carefully controlled if constructors are not exposed. Define static factory methods instead of public constructors.

Guideline 7-2 / OBJECT-2: Prevent the unauthorized construction of sensitive classes

Where an existing API exposes a security-sensitive constructor, limit the ability to create instances.

Object Construction

7-3: Defend against partially initialized instances of nonfinal classes

- A constructor exception doesn't always destroy the object.
  - Attackers override finalize method to get partially initialized `ClassLoader` instance.

```
public class ClassLoader {
  public ClassLoader() {
    SecurityCheck();
    init();
}
```

- Treat public statics primarily as constants-
  - Consider using enums(type-safe and implicitly static final).

Sensitive static code can be modified by untrusted code.

Static variables are global across a Java runtime environment.

Treat public statics primarily as constants-

Consider using enums(type-safe and implicitly static final).

Object Construction

7-3: Defend against partially initialized instances of nonfinal classes

- Declare class as final if possible.
  - If **finalize** method can be overridden, ensure partially uninitialized instances are usable.
- Don't set fields until all checks have completed.
  - Use an **initialized** flag.

```
public class ClassLoader {
 private boolean initialized=false;

  public ClassLoader() {
    SecurityCheck();
    init();
    initialized=true;//check flag in all relevant methods
}
```

Sensitive static code can be modified by untrusted code.

Static variables are global across a Java runtime environment.

Treat public statics primarily as constants-

Consider using enums(type-safe and implicitly static final).

Guideline 7-4 / OBJECT-4: Prevent constructors from calling methods that can be overridden

Constructors that call overridable methods give attackers a reference to this (the object being constructed) before the object has been fully initialized.

Guideline 7-5 / OBJECT-5: Defend against cloning of nonfinal classes

A nonfinal class may be subclassed by a class that also implements java.lang.Cloneable. The result is that the base class can be unexpectedly cloned, although only for instances created by an adversary. The clone will be a shallow copy. The twins will share referenced objects but have different fields and separate intrinsic locks.

Serialization and Deserialization

8-Serialization and Deserialization	
8-1	Avoid serialization for security-sensitive classes
8-2	Guard sensitive data during serialization
8-3	View deserialization the same as object construction
8-4	Duplicate the SecurityManager checks enforced in a class during serialization and deserialization
8-5	Filter untrusted serial data

Why should I care?

- Careless deserialization from untrusted sources allows attackers to create unwanted instances of critical classes.
- Expect side effects with serialization.
- Wherever possible use XML/DTD.

Serialization and Deserialization

8-3: View deserialization the same as object construction

Antipattern: Believing deserialization is unrelated to constructors

Deserialization creates a new instance of a class without invoking any constructor on that class. Therefore, deserialization should be designed to behave like normal construction.

```
public final class ByteString implements java.io.Serializable {

 private static final long serialVersionUID = 1L;
   private byte[] data;
   public ByteString(byte[] data) {
   this.data = data.clone(); // Make copy before assignment.
   }

 private void readObject(java.io.ObjectInputStream in)
       throws java.io.IOException, ClassNotFoundException
{
 java.io.ObjectInputStreadm.GetField fields = in.readFields();
 this.data = ((byte[])fields.get("data")).clone();
 }
  ...
}
```

- Default deserialization and `ObjectInputStream.defaultReadObject` can assign arbitrary objects to nontransient fields and does not necessarily return.
- Use `ObjectInputStream.readFields` instead to insert copying before assignment to fields.
- Or, if possible, don't make sensitive classes serializable

Summary

In this lesson, you should have learned how to:

- Describe the Java SE security overview
- Explain vulnerabilities
- Describe the secure coding guidelines and antipatterns

Resources

First follow the guidelines from the source:
Secure Coding Guidelines for Java SE

Additionally:
securecoding.cert.org

Practice 18: Overview

- There are no practices for this lesson.

Quiz

You can reduce the risk by running vulnerable native code by doing the following:

A. Defining wrappers around native methods
B. Using strict input validation
C. Using a stricter Java security policy

Quiz

Deserialization of untrusted data should be avoided whenever possible because:

A. It is unrelated to constructors

B. It is the same as object construction

C. It doesn't matter

Quiz

Which phase should you plan against denial-of-service?

A. Design
B. Development
C. Testing
D. Deployment

Quiz

Always declare public static fields as final.

A. True

B. False