Secure Coding Guidelines, Continued: Preventing Attacks and Avoiding Antipatterns

Jeff Nisewanger

Senior Staff Engineer
Sun Microsystems
http://java.sun.com

TS-2594

Goal

Learn more about how to reduce vulnerabilities by avoiding insecure coding patterns

Source: http://en.wikipedia.org/wiki/Vulnerability_%28computer_science%29

What Is a Vulnerability?

A weakness in a system allowing an attacker to violate the integrity, confidentiality, access control, availability, consistency or audit mechanism of the system or the data and applications it hosts

Source: http://en.wikipedia.org/wiki/Vulnerability_%28computer_science%29

What Causes Vulnerabilities?

- Faulty assumptions in the application architecture
- Errors in configuration
- Incorrect logic
- Insecure programming practices (antipatterns)
- …

This session focuses on **antipatterns**

Secure Coding Antipatterns

- Programming practices you should **avoid**
  - Negative counterpart to a design pattern
  - E.g. implementing methods that don't validate input params
- Antipatterns not set in stone
  - Generally should avoid them, but there are exceptions
  - Make sure you understand the consequences
- Vulnerabilities may exist in various locations
  - Application code, shared libraries, Java™ platform core libraries

Antipatterns in C Versus the Java Programming Language

- C-based antipatterns often exploit buffer overflows
- Java runtime environment safely manages memory
  - Performs automatic bounds checks on arrays
  - No pointer arithmetic
- The Java runtime environment often executes untrusted code
  - Must protect against access to unauthorized resources
- Results in a different set of coding antipatterns than C

How This Presentation Is Organized

- List common coding antipatterns
- For each antipattern:
  - Show real example from an older JDK™ software release
  - Explain the problem and attack scenario
  - Describe the proper secure coding guidelines
- Summary
  - URL pointing to more comprehensive list of Java programming language secure coding guidelines

Common Java Platform Antipatterns

1. **Assuming objects are immutable**
2. Basing security checks on untrusted sources
3. Ignoring changes to superclasses
4. Neglecting to validate inputs
5. Misusing public static variables
6. Believing a constructor exception destroys the object

Antipattern 1: Assuming Objects Are Immutable

Example from JDK 1.1 software

```
package java.lang;

public class Class {

    private Object[] signers;

    public Object[] getSigners() {
       return signers;
    }
}
```

*Class.getSigners() is actually implemented as a native method, but the behavior is equivalent to the above. See http://java.sun.com/security/getSigners.html

Antipattern 1: Assuming Objects Are Immutable

Attacker can change signers of a class

```
package java.lang;

public class Class {

    private Object[] signers;

    public Object[] getSigners() {
       return signers;
    }
}

Object[] signers = this.getClass().getSigners();
signers[0] = <new signer>;
```

Antipattern 1: Assuming Objects Are Immutable

Problem

- Mutable input and output objects can be modified by the caller
- Modifications can cause applications to behave incorrectly
- Modifications to sensitive security state may result in elevated privileges for attacker
  - e.g. altering the signers of a class can give the class access to unauthorized resources

Antipattern 1: Assuming Objects Are Immutable

Secure coding guidelines

- Make a copy of mutable output parameters
- Make a copy of mutable input parameters

```
public Object[] getSigners() {
    // signers contains immutable type X509Certificate.
    // shallow copy of array is OK.
    return signers.clone();
}

public MyClass(Date start, boolean[] flags) {
    this.start = new Date(start.getTime());
    this.flags = flags.clone();
}
```

- Perform deep cloning on arrays if necessary

Common Java Platform Antipatterns

1. Assuming objects are immutable
2. **Basing security checks on untrusted sources**
3. Ignoring changes to superclasses
4. Neglecting to validate inputs
5. Misusing public static variables
6. Believing a constructor exception destroys the object

Antipattern 2: Basing Security Checks on Untrusted Sources

Example from JDK 5.0 software

```
public RandomAccessFile openFile(final java.io.File f) {
    askUserPermission(f.getPath());
    ...
    return (RandomAccessFile)AccessController.doPrivileged() {
        public Object run() {
            return new RandomAccessFile(f.getPath());
        }
    }
}
```

Antipattern 2: Basing Security Checks on Untrusted Sources

Attacker can pass in subclass of Java.io.File that overrides getPath()

```
public RandomAccessFile openFile(final java.io.File f) {
    askUserPermission(f.getPath());
    ...
        return new RandomAccessFile(f.getPath());
    ...
}

public class BadFile extends java.io.File {
    private int count;
    public String getPath() {
        return (++count == 1) ? "/tmp/foo" : "/etc/passwd";
    }
}
```

Antipattern 2: Basing Security Checks on Untrusted Sources

Problem

- Security checks can be fooled if they are based on information that attackers can control
- It is easy to assume input types defined in the Java platform core libraries (like java.io.File) are secure and can be trusted
  - Non-final classes/methods can be subclassed
  - Mutable types can be modified

Antipattern 2: Basing Security Checks on Untrusted Sources

Secure coding guidelines

- Don't assume inputs are immutable
- Make defensive copies of non-final or mutable inputs and perform checks using copies

```
public RandomAccessFile openFile(File f) {
     final File copy = f.clone();
     askUserPermission(copy.getPath());
     ...
        return new RandomAccessFile(copy.getPath());
}
```

Antipattern 2: Basing Security Checks on Untrusted Sources

Secure coding guidelines

- **WRONG**: clone() copies attacker's subclass

```
public RandomAccessFile openFile(java.io.File f) {
    final java.io.File copy = f.clone();
    askUserPermission(copy.getPath());
    ...
}
```

- **RIGHT**

```
java.io.File copy = new java.io.File(f.getPath());
```

Common Java Platform Antipatterns

1. Assuming objects are immutable
2. **Basing security checks on untrusted sources**
3. **Ignoring changes to superclasses**
4. **Neglecting to validate inputs**
5. Misusing public static variables
6. Believing a constructor exception destroys the object

Antipattern 3: Ignoring Changes to Superclasses

Example from JDK 1.2 software

```
java.util.Hashtable              put(key, val)
                                 remove(key)
        ↑ extends
java.util.Properties

        ↑ extends
java.security.Provider           put(key, val) // security check
                                 remove(key)   // security check
```

Antipattern 3: Ignoring Changes to Superclasses

Example from JDK 1.2 software (Cont.)

```
java.util.Hashtable                put(key, val)
                                   remove(key)
        ↑ extends                  Set entrySet()

java.util.Properties

        ↑ extends

java.security.Provider             put(key, val) // security check
                                   remove(key)   // security check
```

Antipattern 3: Ignoring Changes to Superclasses

Attacker bypasses **remove** method and uses inherited **entrySet** method to delete properties

```
java.util.Hashtable              put(key, val)
                                 remove(key)
       ↑ extends                 Set entrySet() //supports removal

java.util.Properties

       ↑ extends

java.security.Provider           put(key, val) // security check
                                 remove(key)   // security check
```

Antipattern 3: Ignoring Changes to Superclasses

Problem

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

Antipattern 3: Ignoring Changes to Superclasses

Secure coding guidelines

- Avoid inappropriate subclassing
  - Subclass when the inheritance model is well specified and well understood
- Monitor changes to superclasses
  - Identify behavioral changes to existing inherited methods and override if necessary
  - Identify new methods and override if necessary

```
java.security.Provider  put(key, value)// security check
                        remove(key)    // security check
                        Set entrySet() // immutable set
```

Common Java Platform Antipatterns

1. Assuming objects are immutable
2. **Basing security checks on untrusted sources**
3. Ignoring changes to superclasses
4. **Neglecting to validate inputs**
5. Misusing public static variables
6. Believing a constructor exception destroys the object

Antipattern 4:
Neglecting to Validate Inputs

Example from JDK 1.4 software

```
package sun.net.www.protocol.http;

public class HttpURLConnection extends
                                   java.net.HttpURLConnection {
    /**
     * Set header on HTTP request
     */
    public void setRequestProperty(String key, String value) {
        // no input validation on key and value
    }
}
```

Antipattern 4: Neglecting to Validate Inputs

Attacker crafts HTTP headers with embedded requests that bypass security

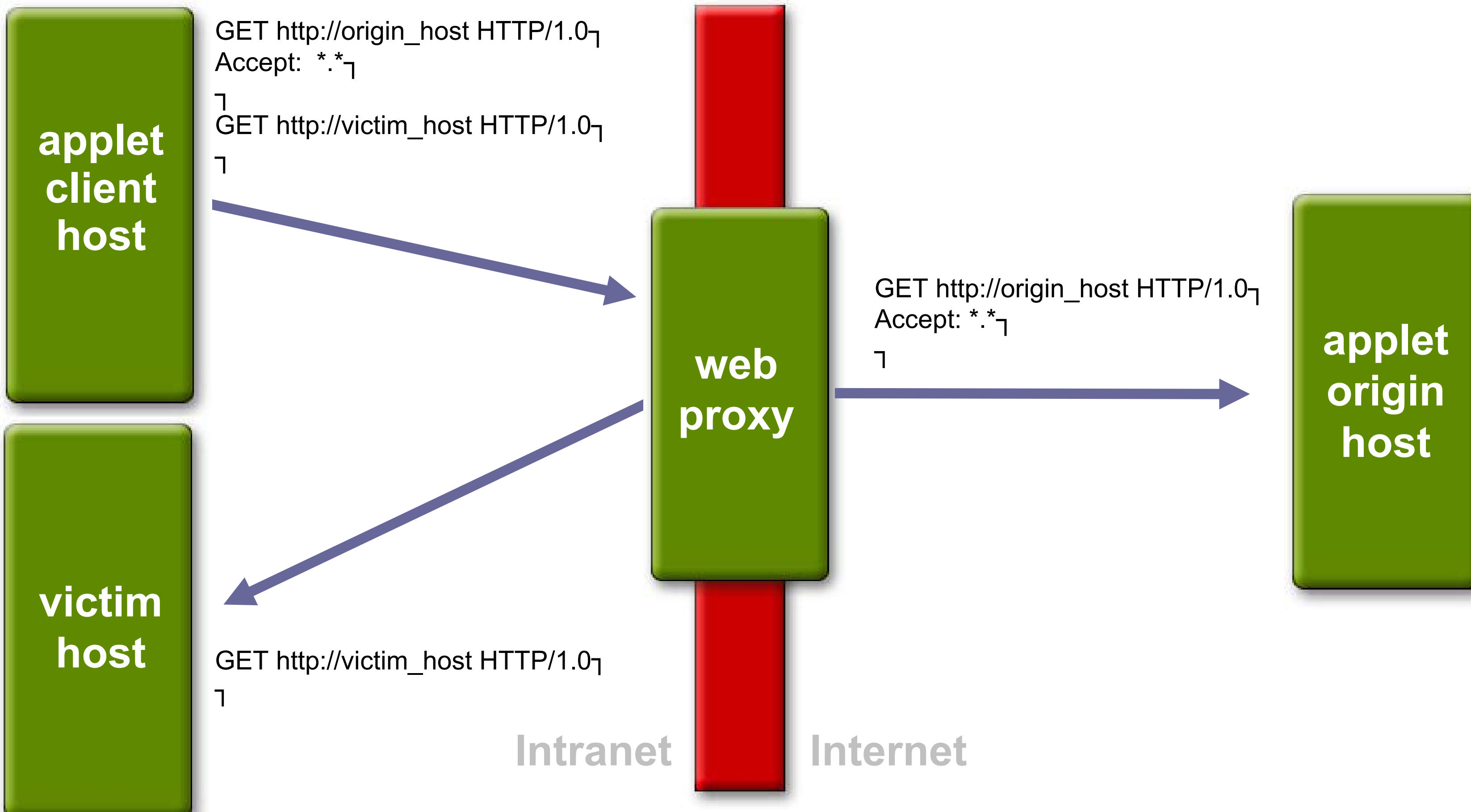
```
package sun.net.www.protocol.http;

public class HttpURLConnection extends java.net.URLConnection {
    public void setRequestProperty(String key, String value) {
        // no input validation on key and value
    }
}

urlConn.setRequestProperty
        ("Accept",
        "*.*\r\n\r\nGET http://victim_host HTTP/1.0\r\n\r\n");
```

Antipattern 4: Neglecting to Validate Inputs

Embedded request bypasses security check



Antipattern 4: Neglecting to Validate Inputs

Problem

- Creative inputs with out-of-bounds values or escape characters can be crafted
- Affects code that processes requests or delegates to subcomponents
  - Implements network protocols
  - Constructs SQL requests
  - Calls shell scripts
- Additional issues when calling native methods
  - No automatic array bounds checks

Antipattern 4: Neglecting to Validate Inputs

Secure coding guidelines

- Validate inputs
  - Check for escape characters
  - Check for out-of-bounds values
  - Check for malformed requests
  - Regular expression API can help validate String inputs
- Pass validated inputs to sub-components
  - Wrap native methods in Java programming language wrapper to validate inputs
  - Make native methods private

Common Java Platform Antipatterns

1. Assuming objects are immutable
2. Basing security checks on untrusted sources
3. Ignoring changes to superclasses
4. Neglecting to validate inputs
5. Misusing public static variables
6. Believing a constructor exception destroys the object

Antipattern 5: Misusing Public Static Variables

Example from JDK 1.4.2 software

```
package org.apache.xpath.compiler;

public class FunctionTable {
    public static FuncLoader m_functions;
}
```

Antipattern 5: Misusing Public Static Variables

Attacker can replace function table

```
package org.apache.xpath.compiler;

public class FunctionTable {
    public static FuncLoader m_functions;
}
```

```
FunctionTable.m_functions = <new_table>;
```

Antipattern 5: Misusing Public Static Variables

Problem

- Sensitive static state can be modified by untrusted code
  - Replacing the function table gives attackers access to the XPathContext used to evaluate XPath expressions
- Static variables are global across a Java runtime environment
  - Can be used as a communication channel between different application domains (e.g. by code loaded into different class loaders)

Antipattern 5: Misusing Public Static Variables

Secure coding guidelines

- Reduce the scope of static fields

```
private static FuncLoader m_functions;
```

- Treat public statics primarily as constants
  - Consider using enum types
  - Make public static fields final

```
public class MyClass {
    public static final int LEFT = 1;
    public static final int RIGHT = 2;
}
```

Antipattern 5: Misusing Public Static Variables

Secure coding guidelines

- Define assessor methods for mutable static state
  - Add appropriate security checks

```
public class MyClass {
    private static byte[] data;

    public static byte[] getData() {
       return data.clone();
    }
    public static void setData(byte[] b) {
       securityCheck();
       data = b.clone();
    }
}
```

Common Java Platform Antipatterns

1. Assuming objects are immutable
2. **Basing security checks on untrusted sources**
3. **Ignoring changes to superclasses**
4. **Neglecting to validate inputs**
5. **Misusing public static variables**
6. **Believing a constructor exception destroys the object**

Antipattern 6: Believing a Constructor Exception Destroys the Object

Example from JDK 1.0.2 software

```
package java.lang;

public class ClassLoader {
    public ClassLoader() {
        // permission needed to create class loader
        securityCheck();
        init();
    }
}
```

Antipattern 6: Believing a Constructor Exception Destroys the Object

Attacker overrides finalize to get partially initialized ClassLoader instance

```
package java.lang;

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

```
public class MyCL extends ClassLoader {
    static ClassLoader cl;

    protected void finalize() {
        cl = this;
    }

    public static void main(String[] s) {
        try {
            new MyCL();
        } catch (SecurityException e) { }

        System.gc();
        System.runFinalization();
        System.out.println(cl);
    }
}
```

Antipattern 6: Believing a Constructor Exception Destroys the Object

Problem

- Throwing an exception from a constructor does not prevent a partially initialized instance from being acquired
  - Attacker can override ***finalize*** method to obtain the object
- Constructors that call into outside code often naively propagate exceptions
  - Enables the same attack as if the constructor directly threw the exception

Antipattern 6: Believing a Constructor Exception Destroys the Object

Secure coding guidelines

- Make class final if possible
- If ***finalize*** method can be overridden, ensure partially initialized instances are unusable
  - Do not set fields until all checks have completed
  - Use an ***initialized*** flag

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

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

Common Java Platform Antipatterns

7. Assuming exceptions are harmless
8. Believing deserialization is unrelated to construction
9. Believing deserialization field values are unshared

Antipattern 7: Assuming Exceptions Are Harmless

Problem

- Exceptions may contain sensitive data such as directory paths that imply user identity

Antipattern 7: Assuming Exceptions Are Harmless

Attacker can learn sensitive data

```
public class PersonalData {
    public load() throws IOException {
        String homedir = System.getProperty("user.dir");
        File f = new File(homedir, "personal.dat");
        FileInputStream s = new FileInputStream(f);
    }
}

try { personal.load(); } catch (IOException e) {
    String homedir = parsePath(e.message());
    String username = parseUser(homedir);
}
```

Antipattern 7: Assuming Exceptions Are Harmless

Secure coding guidelines

- Sanitize or mask exceptions

```
public class PersonalData {
    public load() throws IOException {
        try {
            ...
        } catch (Exception e) {
            throw new IOException("Could not load data");
        }
    }
}
```

Common Java Platform Antipatterns

7. **Assuming exceptions are harmless**
8. **Believing deserialization is unrelated to construction**
9. **Believing deserialization field values are unshared**

Antipattern 8: Believing Deserialization Is Unrelated to Constructors

Example from JDK 1.1 software

```
package java.math;

public class BigInteger extends Number {
    private int signum;
    public BigInteger(int signum, byte[] magnitude){
        if (signum < -1 || signum > 1) {
            throw new NumberFormatException()
          ...
    }
}
```

Antipattern 8: Believing Deserialization Is Unrelated to Constructors

Attacker can deserialize a stream with invalid field data

```
package java.math;

public class BigInteger extends Number {
    private int signum;
    public BigInteger(int signum, byte[] magnitude){
       if (signum < -1 || signum > 1) {
          throw new NumberFormatException()
         ...
    }
}
ObjectInputStream is = new FileInputStream("bad.ser");
BigInteger bigInt = is.readObject();
```

Antipattern 8: Believing Deserialization Is Unrelated to Constructors

Problem

- The default deserialization mechanism cannot automatically apply the same invariant and parameter checking present in the constructor
  - Attacker can create a malicious serialization stream with invalid field values

Antipattern 8: Believing Deserialization Is Unrelated to Constructors

Secure coding guidelines

- Create a custom readObject() method that shares the same validation checking as the class constructors

```
private void readObject(ObjectInputStream s) {
    s.defaultReadObject();
    // Validate signum
    if (signum < -1 || signum > 1)
       throw new StreamCorruptedException();
}
```

Common Java Platform Antipatterns

7. Assuming exceptions are harmless
8. Believing deserialization is unrelated to construction
9. Believing deserialization field values are unshared

Antipattern 9: Believing Deserialized Field Values Are Unshared

Example from JDK 1.1 software

```
package java.math;

public class BigInteger extends Number {
    private byte[] magnitude;
    public BigInteger(int signum, byte[] magnitude){
        this.magnitude = stripLeadingZeroBytes(magnitude);
          ...
    }
}
```

Antipattern 9: Believing Deserialized Field Values Are Unshared

Attacker can deserialize a stream with malicious 'extra' references to mutable fields

```
package java.math;

public class BigInteger extends Number {
    private byte[] magnitude;
    public BigInteger(int signum, byte[] magnitude){
        this.magnitude = stripLeadingZeroBytes(magnitude);
         ...
    }
}
ObjectInputStream is = new FileInputStream("bad.ser");
BigInteger bigInt = is.readObject();
byte[] magnitudeCopy = is.readObject();
```

Antipattern 9: Believing Deserialized Field Values Are Unshared

Problem

- The default deserialization mechanism assumes object references in a stream might have multiple legitimate references
  - Attacker can create a malicious serialization stream with unintended extra references to a mutable field object instance

Antipattern 9: Believing Deserialized Field Values Are Unshared

Secure coding guidelines

- Create a custom readObject() method that creates an unshared private copy of mutable field instances

```
private void readObject(ObjectInputStream s) {
    s.defaultReadObject();
    magnitude = (byte [])magnitude.clone();
}
```

Summary

- Vulnerabilities are a concern for all developers
  - Can have severe impacts on security and privacy
- Follow secure coding guidelines to reduce vulnerabilities
  - Encourages secure programming from the outset
  - Helps limit bad assumptions that might be made
  - Avoids common antipatterns

Acknowledgements

- Secure Internet Programming group at Princeton University
  - Dirk Balfanz, Drew Dean, Edward W. Felten, and Dan Wallach
- Marc Schönefeld
- Harmen van der Wal
- Sun Microsystems
  - Andreas Sterbenz, Charlie Lai

For More Information

- Contact the Java Platform, Standard Edition (Java SE Platform) Security Team with comments
  - java-security@sun.com
- Meet the Java SE Platform Security Team
  - BOF-2516 8:55pm–9:45pm, Thurs. May 10
- Secure coding guidelines for Java technology
  - http://java.sun.com/security/seccodeguide.html

Q&A

Secure Coding Guidelines, Continued: Preventing Attacks and Avoiding Antipatterns

Jeff Nisewanger

Senior Staff Engineer
Sun Microsystems
http://java.sun.com

TS-2594