Secure Coding Antipatterns: Avoiding Vulnerabilities

Andreas Sterbenz
Charlie Lai

Sun Microsystems

TS-1238

Goal

Learn how to reduce vulnerabilities by avoiding insecure coding patterns (antipatterns)

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 Language

- C-based antipatterns often exploit buffer overflows
- Java runtime safely manages memory
  - Performs automatic bounds checks on arrays
  - No pointer arithmetic
- The Java runtime 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 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

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

- Make a copy of mutable input parameters

```
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 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

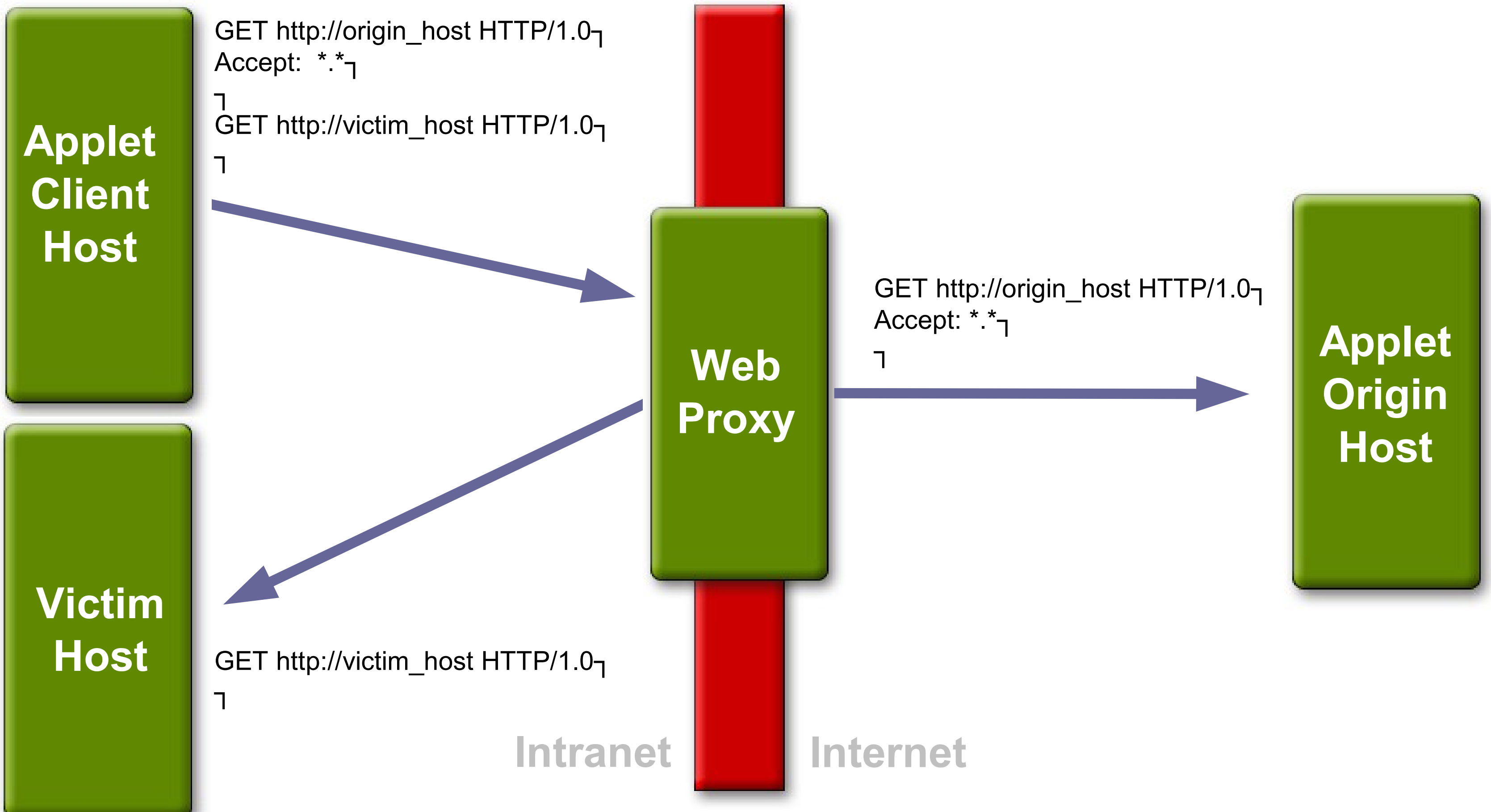
```
package sun.net.www.protocol.http;

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 subcomponents
  - Wrap native methods in Java 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 accessor 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 (Exception 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
    }
}
```

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

For More Information

- Contact the Java SE Security Team with comments
  - java-security@sun.com
- Meet the Java SE Security Team
  - 10:30pm, May 18, Gateway 102/103
- Secure coding guidelines for Java technology
  - http://java.sun.com/security/seccodeguide.html
    - Currently being updated, new version to be posted soon

Acknowledgements

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

Q&A

Secure Coding Antipatterns: Avoiding Vulnerabilities

Andreas Sterbenz

Charlie Lai

Sun Microsystems

TS-1238