Table of Contents

[1. Access Specifier Manipulation 2](#_Toc428366736)

[3. J2EE Bad Practices-Leftover Debug code. 5](#_Toc428366737)

[4. Password Management 5](#_Toc428366738)

[5. Path Manipulation 6](#_Toc428366739)

[6. Poor Error Handling-Overlay Broad Throws: 8](#_Toc428366740)

[7. System information Leak 9](#_Toc428366741)

[8. Unrealeased resource-streams 11](#_Toc428366742)

[9. Dead Code 13](#_Toc428366743)

[10. Poor Style –Non-final static field 13](#_Toc428366744)

[11. Poor Logging Practice : Use of a System Output Stream 14](#_Toc428366745)

[12. Cross site-Scripting: Reflected 15](#_Toc428366746)

[13. Password Management-HardCoded Password 19](#_Toc428366747)

[14. Code Correctness : Class Does not Implement Equals 21](#_Toc428366748)

[15. Denial of Service 22](#_Toc428366749)

[16. Header Manipulation 23](#_Toc428366750)

[17. Insecure Randomness: 27](#_Toc428366751)

[18. JSON Injection 28](#_Toc428366752)

[19. Poor Error Handling : Return inside Finally 30](#_Toc428366753)

[20. Poor style : Value Never Read 32](#_Toc428366754)

[21. Redundant Null check 32](#_Toc428366755)

[22. Server-side Request Forgery 33](#_Toc428366756)

[23. Unreleased Resource 34](#_Toc428366757)

# 1. Access Specifier Manipulation

**Explanation:**

The AccessibleObject API allows the programmer to get around the access control checks provided by Java access specifiers. In particular it enables the programmer to allow a reflected object to bypass Java access controls and in turn change the value of private fields or invoke private methods, behaviors that are normally disallowed.

**Recommendations:**

Access specifiers should only be changed by a privileged class using arguments that an attacker cannot set. All occurrences should be examined carefully.

**2.** Built ID Misconfiguration-External Maven Dependency Repository

**Abstract:**

This maven build script relies on external sources, which could allow an attacker to insert malicious code into the final product or to take control of the build machine.

**Explanation:**

Several tools exist within the Java development world to aid in dependency management: both Apache Ant and Apache Maven build systems include functionality specifically designed to help manage dependencies and Apache Ivy is developed explicitly as a dependency manager. Although there are differences in their behavior, these tools share the common functionality that they automatically download external dependencies specified in the build process at build time. This makes it much easier for developer B to build software in the same manner as developer A. Developers just store dependency information in the build file, which means that each developer and build engineer has a consistent way to obtain dependencies, compile the code, and deploy without the dependency management hassles involved in manual dependency management. The following examples illustrate how Ivy, Ant and Maven can be used to manage external dependencies as part of a build process.

Under Maven, instead of listing explicit URLs from which to retrieve the dependencies, developers specify the dependency names and versions and Maven relies on its underlying configuration to identify the server(s) from which to retrieve the dependencies. For commonly used components this saves the developer from having to researching dependency locations.

**Example 1:** The following except from a Maven pom.xml file shows how a developer can specify multiple external dependencies using their name and version:

<dependencies>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1</version>

</dependency>

<dependency>

<groupId>javax.jms</groupId>

<artifactId>jms</artifactId>

<version>1.1</version>

</dependency>

...

</dependencies>

Two distinct types of attack scenarios affect these systems: An attacker could either compromise the server hosting the dependency or compromise the DNS server the build machine uses to redirect requests for hostname of the server hosting the dependency to a machine controlled by the attacker. Both scenarios result in the attacker gaining the ability to inject a malicious version of a dependency into a build running on an otherwise uncompromised machine.

Regardless of the attack vector used to deliver the Trojan dependency, these scenarios share the common element that the build system blindly accepts the malicious binary and includes it in the build. Because the build system has no recourse for rejecting the malicious binary and existing security mechanisms, such as code review, typically focus on internally-developed code rather than external dependencies, this type of attack has a strong potential to go unnoticed as it spreads through the development environment and potentially into production.

Although there is some risk of a compromised dependency being introduced into a manual build process, by the tendency of automated build systems to retrieve the dependency from an external source each time the build system is run in a new environment greatly increases the window of opportunity for an attacker. An attacker need only compromise the dependency server or the DNS server during one of the many times the dependency is retrieved in order to compromise the machine on which the build is occurring.

**Recommendations:**

The simplest solution is to refrain from adopting automated dependency management systems altogether. Managing dependencies manually eliminates the potential for unexpected behavior caused by the build system. Obviously, the an attacker could still mount one of the attacks described above to coincide with the manual retrieval of a dependency, but limiting the frequency with which the dependency must be retrieved significantly reduces the window of opportunity for an attacker. Finally, this solution forces the development organization to rely on what is ostensibly an antiquated build system. A system based on manual dependency management is often more difficult to use and maintain, and may be unacceptable in some software development environments.

The second solution is a hybrid of the traditional manual dependency management approach and the fully automated solution that is popular today. The biggest advantage of the manual build process is the decreased window of attack, which can be achieved in a semi-automated system by replicating external dependency servers internally. Any build system that requires an external dependency can then point to the internal server using a hard-coded internal IP address to bypass the risk of DNS-based attacks. As new dependencies are added and new versions released, they can be downloaded once and included on the internal repository. This solution reduces the attack opportunities and allows the organization leverage existing internal network security infrastructure.

To implement this solution using Maven, a project should have the IP address for an internal repository hard coded the pom.xml. Specifying the IP address in the pom.xml ensures the internal repository will be used by the corresponding build, but is tied to a specific project. Alternatively, the IP address can be specified in settings.xml, which makes the configuration easier to share across multiple projects.

**Example 2:** The following Maven pom.xml demonstrates the use of an explicit internal IP address (the entries can also be used in settings.xml):

<project>

...

<repositories>

<repository>

<releases>

<enabled>true</enabled>

<updatePolicy>always</updatePolicy>

<checksumPolicy>warn</checksumPolicy>

</releases>

<snapshots>

<enabled>true</enabled>

<updatePolicy>never</updatePolicy>

<checksumPolicy>fail</checksumPolicy>

</snapshots>

<id>central</id>

<name>Internal Repository</name>

<url>http://172.16.1.13/maven2</url>

<layout>default</layout>

</repository>

</repositories>

<pluginRepositories>

...

</pluginRepositories>

...

</project>

# 3. J2EE Bad Practices-Leftover Debug code.

**Explanation:**

A common development practice is to add "back door" code specifically designed for debugging or testing purposes that is not intended to be shipped or deployed with the application. When this sort of debug code is accidentally left in the application, the application is open to unintended modes of interaction. These back door entry points create security risks because they are not considered during design or testing and fall outside of the expected operating conditions of the application.

The most common example of forgotten debug code is a main () method appearing in a web application. Although this is an acceptable practice during product development, classes that are part of a production J2EE application should not define a main ().

**Recommendations:**

Remove debug code before deploying a production version of an application. Regardless of whether a direct security threat can be articulated, it is unlikely that there is a legitimate reason for such code to remain in the application after the early stages of development.

**Tips:**

1. The presence of a main () method may represent the tip of an iceberg. When you find a main (), look for other indications that developers were rushed or otherwise not able to conclude their efforts normally.

2. If you are auditing a non-J2EE Java application, the J2EE Bad Practices category might not apply to your environment. If this is the case, you can use Audit Guide to suppress these issues.

# 4. Password Management

**Abstract:**

Storing a plaintext password in a configuration file may result in a system compromise.

**Explanation:**

Storing a plaintext password in a configuration file allows anyone who can read the file access to the password-protected resource. Developers sometimes believe that they cannot defend the application from someone who has access to the configuration, but this attitude makes an attacker's job easier. Good password management guidelines require that a password never be stored in plaintext.

**Recommendations:**

A password should never be stored in plaintext. Instead, the password should be entered by an administrator when the system starts. If that approach is impractical, a less secure but often adequate solution is to obfuscate the password and scatter the de-obfuscation material around the system so that an attacker has to obtain and correctly combine multiple system resources to decipher the password.

Some third-party products claim the ability to manage passwords in a more secure way. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. For a secure solution the only viable option is a proprietary one.

**Tips:**

1. HP Fortify Static Code Analyzer searches configuration files for common names used for password properties. Audit these issues by verifying that the flagged entry is used as a password and that the password entry contains plaintext.

2. If the entry in the configuration file is a default password, require that it be changed in addition to requiring that it be obfuscated in the configuration file.

# 5. Path Manipulation

**Explanation:**

Path manipulation errors occur when the following two conditions are met:

1. An attacker can specify a path used in an operation on the file system.

2. By specifying the resource, the attacker gains a capability that would not otherwise be permitted.

For example, the program may give the attacker the ability to overwrite the specified file or run with a configuration controlled by the attacker.

**Example 1:** The following code uses input from an HTTP request to create a file name. The programmer has not considered the possibility that an attacker could provide a file name such as "../../tomcat/conf/server.xml", which causes the application to delete one of its own configuration files.

String rName = request.getParameter("reportName");

File rFile = new File ("/usr/local/apfr/reports/" + rName);

...

rFile.delete();

**Example 2:** The following code uses input from a configuration file to determine which file to open and echo back to the user. If the program runs with privileges and malicious users can change the configuration file, they can use the program to read any file on the system that ends with the extension .txt.

fis = new FileInputStream(cfg.getProperty("sub") +".txt");

amt = fis.read(arr);

out.println(arr);

Some think that in the mobile world, classic vulnerabilities, such as path manipulation, do not make sense -- why would the user attack themself? However, keep in mind that the essence of mobile platforms is applications that are downloaded from various sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which necessitates expanding the attack surface of mobile applications to include inter-process communication.

**Example 3:** The following code adapts Example 1 to the Android platform.

...

String rName = this.getIntent().getExtras().getString("reportName");

File rFile = getBaseContext().getFileStreamPath(rName);

...

rFile.delete();

...

**Recommendations:**

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to keep track of. Programmers often resort to blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a whitelist of characters that are allowed to appear in the resource name and accept input composed exclusively of characters in the approved set.

**Tips:**

1. If the program is performing input validation, satisfy yourself that the validation is correct, and use the Custom Rules Editor to create a cleanse rule for the validation routine.

2. Since implementing a blacklist that is effective on its own is notoriously difficult, if validation logic relies on blacklisting, one should be skeptical. Consider different types of input encoding and different sets of meta-characters that might have special meaning when interpreted by different operating systems, databases, or other resources. Determine whether or not the blacklist can be updated easily, correctly, and completely if these requirements ever change.

3. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Spring MVC are among them. To highlight the invalidated sources of input, the HP Fortify Secure Coding Rule packs dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the HP Fortify Software Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

# 6. Poor Error Handling-Overlay Broad Throws:

**Explanation:**

Declaring a method to throw Exception or Throw able makes it difficult for callers to do good error handling and error recovery. Java's exception mechanism is set up to make it easy for callers to anticipate what can go wrong and write code to handle each specific exceptional circumstance. Declaring that a method throws a generic form of exception defeats this system.

**Example:** The following method throws three types of exceptions.

public void doExchange()

throws IOException, InvocationTargetException,

SQLException {

...

}

**Recommendations:**

Do not declare methods to throw Exception or Throw able. If the exceptions thrown by a method are not recoverable or should not generally be caught by the caller, consider throwing unchecked exceptions rather than checked exceptions. This can be accomplished by implementing exception classes that extend Runtime Exception or Error instead of Exception, or add a try/catch wrapper in your method to convert checked exceptions to unchecked exceptions.

# 7. System Information Leak

**Explanation:**

An information leak occurs when system data or debugging information leaves the program through an output stream or logging function.

**Example 1:** The following code prints an exception to the standard error stream:

try {

...

} catch (Exception e) {

e.printStackTrace();

}

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. For example, with scripting mechanisms it is trivial to redirect output information from "Standard error" or "Standard output" into a file or another program. Alternatively the system that the program runs on could have a remote logging mechanism such as a "syslog" server that will send the logs to a remote device. During development you will have no way of knowing where this information may end up being displayed.

In some cases the error message tells the attacker precisely what sort of an attack the system is vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the leaked information could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Here is another scenario, specific to the mobile world. Most mobile devices now implement a Near-Field Communication (NFC) protocol for quickly sharing information between devices using radio communication. It works by bringing devices to close proximity or simply having them touch each other. Even though the communication range of NFC is limited to just a few centimeters, eavesdropping, data modification and various other types of attacks are possible, since NFC alone does not ensure secure communication.

**Example 2:** The Android platform provides support for NFC. The following code creates a message that gets pushed to the other device within the range.

public static final String TAG = "NfcActivity";

private static final String DATA\_SPLITTER = "\_\_:DATA:\_\_";

private static final String MIME\_TYPE = "application/my.applications.mimetype";

...

public NdefMessage createNdefMessage(NfcEvent event) {

TelephonyManager tm = (TelephonyManager)Context.getSystemService(Context.TELEPHONY\_SERVICE);

String VERSION = tm.getDeviceSoftwareVersion();

String text = TAG + DATA\_SPLITTER + VERSION;

NdefRecord record = new NdefRecord(NdefRecord.TNF\_MIME\_MEDIA,

MIME\_TYPE.getBytes(), new byte[0], text.getBytes());

NdefRecord[] records = { record };

NdefMessage msg = new NdefMessage(records);

return msg;

}

...

**Recommendations:**

Write error messages with security in mind. In production environments, turn off detailed error information in favor of brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Be careful, debugging traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example).

Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system.

If you are concerned about leaking system data via NFC on an Android device, you could do one of the following three things. Either do not include system data in the messages pushed to other devices in range, or encrypt the payload of the message, or establish secure communication channel at a higher layer.

**Tips:**

1. Do not rely on wrapper scripts, corporate IT policy, or quick-thinking system administrators to prevent system information leaks. Write software that is secure on its own.

2. This category of vulnerability does not apply to all types of programs. For example, if your application executes on a client machine where system information is already available to an attacker, or if you print system information only to a trusted log file, you can use AuditGuide to filter out this category.

3. Fortify RTA adds protection against this category.

# 8. Unrealeased resource-streams

**Explanation:**

The program can potentially fail to release a system resource.

In this case, there are program paths on which the resource allocated in CreateEntry.java at line 87 is not released.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker may be able to launch a denial of service attack by depleting the resource pool.

**Example:** The following method never closes the file handle it opens. The finalize() method for FileInputStream eventually calls close(), but there is no guarantee as to how long it will take before the finalize() method will be invoked. In a busy environment, this can result in the JVM using up all of its file handles.

private void processFile(String fName) throws FileNotFoundException, IOException {

FileInputStream fis = new FileInputStream(fName);

int sz;

byte[] byteArray = new byte[BLOCK\_SIZE];

while ((sz = fis.read(byteArray)) != -1) {

processBytes(byteArray, sz);

}

}

**Recommendations:**

1. Never rely on finalize() to reclaim resources. In order for an object's finalize() method to be invoked, the garbage collector must determine that the object is eligible for garbage collection. Because the garbage collector is not required to run unless the JVM is low on memory, there is no guarantee that an object's finalize() method will be invoked in an expedient fashion. When the garbage collector finally does run, it may cause a large number of resources to be reclaimed in a short period of time, which can lead to "bursty" performance and lower overall system throughput. This effect becomes more pronounced as the load on the system increases.

Finally, if it is possible for a resource reclamation operation to hang (if it requires communicating over a network to a database, for example), then the thread that is executing the finalize() method will hang.

2. Release resources in a finally block. The code for the Example should be rewritten as follows:

public void processFile(String fName) throws FileNotFoundException, IOException {

FileInputStream fis;

try {

fis = new FileInputStream(fName);

int sz;

byte[] byteArray = new byte[BLOCK\_SIZE];

while ((sz = fis.read(byteArray)) != -1) {

processBytes(byteArray, sz);

}

}

finally {

if (fis != null) {

safeClose(fis);

}

}

}

public static void safeClose(FileInputStream fis) {

if (fis != null) {

try {

fis.close();

} catch (IOException e) {

log(e);

}

}

}

This solution uses a helper function to log the exceptions that might occur when trying to close the stream. Presumably this helper function will be reused whenever a stream needs to be closed.

Also, the processFile method does not initialize the fis object to null. Instead, it checks to ensure that fis is not null before calling safeClose(). Without the null check, the Java compiler reports that fis might not be initialized. This choice takes advantage of Java's ability to detect uninitialized variables. If fis is initialized to null in a more complex method, cases in which fis is used without being initialized will not be detected by the compiler.

# 9. Dead Code

**Explanation:**

This expression (or part of it) will always evaluate to false; the program could be rewritten in a simpler form. The nearby code may be present for debugging purposes, or it may not have been maintained along with the rest of the program. The expression may also be indicative of a bug earlier in the method.

**Example 1:** The following method never sets the variable secondCall after initializing it to false. (The variable firstCall is mistakenly used twice.) The result is that the expression firstCall && secondCall will always evaluate to false, so setUpDualCall() will never be invoked.

public void setUpCalls() {

boolean firstCall = false;

boolean secondCall = false;

if (fCall > 0) {

setUpFCall();

firstCall = true;

}

if (sCall > 0) {

setUpSCall();

firstCall = true;

}

if (firstCall && secondCall) {

setUpDualCall();

}

}

**Recommendations:**

In general, you should repair or remove unused code. It causes additional complexity and maintenance burden without contributing to the functionality of the program.

# 10. Poor Style –Non-final static field

**Explanation:**

Typically, you do not want to provide external classes direct access to your object's member fields since a public field can be changed by any external class. Good object oriented designed uses encapsulation to prevent implementation details, such as member fields, from being exposed to other classes. Further, if the system assumes that this field cannot be changed, then malicious code might be able to adversely change the behavior of the system.

'

**Example 1:** In the following code, the field ERROR\_CODE is declared as public and static, but not final:

public class MyClass

{

public static int ERROR\_CODE = 100;

//...

}

**Recommendations:**

If you intend to expose a field as a constant value, the field should be declared as public static final, otherwise declare the field private.

**Example 2:**

public class MyClass

{

public static final int ERROR\_CODE = 123;

//...

}

# 11. Poor Logging Practice: Use of a System Output Stream

**Explanation:**

**Example 1:** The first Java program that a developer learns to write often looks like this:

public class MyClass

public static void main(String[] args) {

System.out.println("hello world");

}

}

While most programmers go on to learn many nuances and subtleties about Java, a surprising number hang on to this first lesson and never give up on writing messages to standard output using System.out.println().

The problem is that writing directly to standard output or standard error is often used as an unstructured form of logging. Structured logging facilities provide features like logging levels, uniform formatting, a logger identifier, timestamps, and, perhaps most critically, the ability to direct the log messages to the right place. When the use of system output streams is jumbled together with the code that uses loggers properly, the result is often a well-kept log that is missing critical information.

Developers widely accept the need for structured logging, but many continue to use system output streams in their "pre-production" development. If the code you are reviewing is past the initial phases of development, use of System.out or System.err may indicate an oversight in the move to a structured logging system.

**Recommendations:**

Use a Java logging facility rather than System.out or System.err.

**Example 2:** For example, the "hello world" program above can be re-written using log4j like this:

import org.apache.log4j.Logger;

import org.apache.log4j.BasicConfigurator;

public class MyClass {

private final static Logger logger =

Logger.getLogger(MyClass.class);

public static void main(String[] args) {

BasicConfigurator.configure();

logger.info("hello world");

}

}

# 12. Cross site-Scripting: Reflected

**Explanation:**

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of Reflected XSS, the untrusted source is typically a web request, while in the case of Persisted (also known as Stored) XSS it is typically a database or other back-end datastore.

In this case the data enters at doPost() in CreateProject4TW.java at line 42.

2. The data is included in dynamic content that is sent to a web user without being validated.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

**Example 1:** The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

<% String eid = request.getParameter("eid"); %>

...

Employee ID: <%= eid %>

The code in this example operates correctly if eid contains only standard alphanumeric text. If eid has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

**Recommendations:**

The solution to XSS is to ensure that validation occurs in the correct places and checks for the correct properties.

Since XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent XSS vulnerabilities is to validate everything that enters the application and leaves the application destined for the user.

The most secure approach to validation for XSS is to create a whitelist of safe characters that are allowed to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alpha-numeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser should still be considered valid input once they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines what characters have special meaning, many web browsers try to correct common mistakes in HTML and may treat other characters as special in certain contexts, which is why we do not encourage the use of blacklists as a means to prevent XSS. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.

Within the body of a <SCRIPT> </SCRIPT>:

- Semicolons, parentheses, curly braces, and new line characters should be filtered out in situations where text could be inserted directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and may bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

Once you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option in this situation is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and may be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

**Tips:**

1. The HP Fortify Secure Coding Rule packs warn about SQL Injection and Access Control: Database issues when untrusted data is written to a database and also treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources. Nonetheless, it is often still a good idea to validate everything read from the database.

2. Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the rule packs no longer treat URL encoding routines as sufficient to protect against cross-site scripting. Data values that are URL encoded and subsequently output will cause Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

3. Fortify RTA adds protection against this category.

# 13. Password Management-HardCoded Password

**Explanation:**

It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.

**Example 1:** The following code uses a hardcoded password to connect to a database:

DriverManager.getConnection(url, "scott", "tiger");

This code will run successfully, but anyone who has access to it will have access to the password. Once the program has shipped, there is no going back from the database user "scott" with a password of "tiger" unless the program is patched. A devious employee with access to this information can use it to break into the system. Even worse, if attackers have access to the bytecode for the application they can use the javap -c command to access the disassembled code, which will contain the values of the passwords used. The result of this operation might look something like the following for the example above:

**Recommendations:**

Passwords should never be hardcoded and should generally be obfuscated and managed in an external source. Storing passwords in plaintext anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password. At the very least, passwords should be hashed before being stored.

Some third-party products claim the ability to manage passwords in a more secure way. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. For a secure generic solution, the best option today appears to be a proprietary mechanism that you create.

For Android, as well as any other platform that uses SQLite database, a good option is SQLCipher -- an extension to SQLite database that provides transparent 256-bit AES encryption of database files. Thus, credentials can be stored in an encrypted database.

**Example 3:** The code below demonstrates how to integrate SQLCipher into an Android application after downloading the necessary binaries, and store credentials into the database file.

import net.sqlcipher.database.SQLiteDatabase;

...

SQLiteDatabase.loadLibs(this);

File dbFile = getDatabasePath("credentials.db");

dbFile.mkdirs();

dbFile.delete();

SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase(dbFile, "credentials", null);

db.execSQL("create table credentials(u, p)");

db.execSQL("insert into credentials(u, p) values(?, ?)", new Object[]{username, password});

...

Note that references to android.database.sqlite.SQLiteDatabase are substituted with those of net.sqlcipher.database.SQLiteDatabase.

To enable encryption on the WebView store, WebKit has to be re-compiled with the sqlcipher.so library.

**Tips:**

1. The Fortify Java Annotations FortifyPassword and FortifyNotPassword can be used to indicate which fields and variables represent passwords.

2. When identifying null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named fields and variables.

# 14. Code Correctness: Class does not Implement Equals

**Explanation:**

When comparing objects, developers usually want to compare properties of objects. However, calling equals() on a class (or any super class/interface) that does not explicitly implement equals() results in a call to the equals() method inherited from java.lang.Object. Instead of comparing object member fields or other properties, Object.equals() compares two object instances to see if they are the same. Although there are legitimate uses of Object.equals(), it is often an indication of buggy code.

**Example 1:**

public class AccountGroup

{

private int gid;

public int getGid()

{

return gid;

}

public void setGid(int newGid)

{

gid = newGid;

}

}

...

public class CompareGroup

{

public boolean compareGroups(AccountGroup group1, AccountGroup group2)

{

return group1.equals(group2); //equals() is not implemented in AccountGroup

}

}

# 15. Denial of Service

**Explanation:**

Attackers may be able to deny service to legitimate users by flooding the application with requests, but flooding attacks can often be defused at the network layer. More problematic are bugs that allow an attacker to overload the application using a small number of requests. Such bugs allow the attacker to specify the quantity of system resources their requests will consume or the duration for which they will use them.

**Example 1:** The following code allows a user to specify the amount of time for which a thread will sleep. By specifying a large number, an attacker can tie up the thread indefinitely. With a small number of requests, the attacker can deplete the application's thread pool.

int usrSleepTime = Integer.parseInt(usrInput);

Thread.sleep(usrSleepTime);

**Example 2:** The following code reads a String from a zip file. Because it uses the readLine() method, it will read an unbounded amount of input. An attacker can take advantage of this code to cause an OutOfMemoryException or to consume a large amount of memory so that the program spends more time performing garbage collection or runs out of memory during some subsequent operation.

InputStream zipInput = zipFile.getInputStream(zipEntry);

Reader zipReader = new InputStreamReader(zipInput);

BufferedReader br = new BufferedReader(zipReader);

String line = br.readLine();

**Recommendations:**

Validate user input to ensure that it will not cause inappropriate resource utilization.

**Example 3:** The following code allows a user to specify the amount of time for which a thread will sleep just as in Example 1, but only if the value is within reasonable bounds.

int usrSleepTime = Integer.parseInt(usrInput);

if (usrSleepTime >= SLEEP\_MIN &&

usrSleepTime <= SLEEP\_MAX) {

Thread.sleep(usrSleepTime);

} else {

throw new Exception("Invalid sleep duration");

}

}

**Tips:**

1. Denial of service can happen even if the quantity of system resources that will be consumed or the duration for which they will be used is not controlled by an attacker, or at least not directly. Instead, a programmer might choose unsafe constant values for specifying these parameters. The HP Fortify Secure Coding Rulepacks will report such cases as potential Denial of Services vulnerabilities.

# 16. Header Manipulation

**Explanation:**

Header Manipulation vulnerabilities occur when:

1. Data enters a web application through an untrusted source, most frequently an HTTP request.

2. The data is included in an HTTP response header sent to a web user without being validated.

As with many software security vulnerabilities, Header Manipulation is a means to an end, not an end in itself. At its root, the vulnerability is straightforward: an attacker passes malicious data to a vulnerable application, and the application includes the data in an HTTP response header.

One of the most common Header Manipulation attacks is HTTP Response Splitting. To mount a successful HTTP Response Splitting exploit, the application must allow input that contains CR (carriage return, also given by %0d or \r) and LF (line feed, also given by %0a or \n)characters into the header. These characters not only give attackers control of the remaining headers and body of the response the application intends to send, but also allows them to create additional responses entirely under their control.

Many of today's modern application servers will prevent the injection of malicious characters into HTTP headers. For example, recent versions of Apache Tomcat will throw an IllegalArgumentException if you attempt to set a header with prohibited characters. If your application server prevents setting headers with new line characters, then your application is not vulnerable to HTTP Response Splitting. However, solely filtering for new line characters can leave an application vulnerable to Cookie Manipulation or Open Redirects, so care must still be taken when setting HTTP headers with user input.

**Example:** The following code segment reads the name of the author of a weblog entry, author, from an HTTP request and sets it in a cookie header of an HTTP response.

String author = request.getParameter(AUTHOR\_PARAM);

...

Cookie cookie = new Cookie("author", author);

cookie.setMaxAge(cookieExpiration);

response.addCookie(cookie);

Assuming a string consisting of standard alpha-numeric characters, such as "Jane Smith", is submitted in the request the HTTP response including this cookie might take the following form:

HTTP/1.1 200 OK

...

Set-Cookie: author=Jane Smith

...

However, because the value of the cookie is formed of unvalidated user input the response will only maintain this form if the value submitted for AUTHOR\_PARAM does not contain any CR and LF characters. If an attacker submits a malicious string, such as "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n...", then the HTTP response would be split into two responses of the following form:

HTTP/1.1 200 OK

...

Set-Cookie: author=Wiley Hacker

HTTP/1.1 200 OK

...

Clearly, the second response is completely controlled by the attacker and can be constructed with any header and body content desired. The ability of attacker to construct arbitrary HTTP responses permits a variety of resulting attacks, including: cross-user defacement, web and browser cache poisoning, cross-site scripting and page hijacking.

**Cross-User Defacement:** An attacker can make a single request to a vulnerable server that will cause the server to create two responses, the second of which may be misinterpreted as a response to a different request, possibly one made by another user sharing the same TCP connection with the server. This can be accomplished by convincing the user to submit the malicious request themselves, or remotely in situations where the attacker and the user share a common TCP connection to the server, such as a shared proxy server. In the best case, an attacker can leverage this ability to convince users that the application has been hacked, causing users to lose confidence in the security of the application. In the worst case, an attacker may provide specially crafted content designed to mimic the behavior of the application but redirect private information, such as account numbers and passwords, back to the attacker.

**Cache Poisoning:** The impact of a maliciously constructed response can be magnified if it is cached either by a web cache used by multiple users or even the browser cache of a single user. If a response is cached in a shared web cache, such as those commonly found in proxy servers, then all users of that cache will continue receive the malicious content until the cache entry is purged. Similarly, if the response is cached in the browser of an individual user, then that user will continue to receive the malicious content until the cache entry is purged, although only the user of the local browser instance will be affected.

**Cross-Site Scripting:** Once attackers have control of the responses sent by an application, they have a choice of a variety of malicious content to provide users. Cross-site scripting is common form of attack where malicious JavaScript or other code included in a response is executed in the user's browser. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site. The most common and dangerous attack vector against users of a vulnerable application uses JavaScript to transmit session and authentication information back to the attacker who can then take complete control of the victim's account.

**Page Hijacking:** In addition to using a vulnerable application to send malicious content to a user, the same root vulnerability can also be leveraged to redirect sensitive content generated by the server and intended for the user to the attacker instead. By submitting a request that results in two responses, the intended response from the server and the response generated by the attacker, an attacker can cause an intermediate node, such as a shared proxy server, to misdirect a response generated by the server for the user to the attacker. Because the request made by the attacker generates two responses, the first is interpreted as a response to the attacker's request, while the second remains in limbo. When the user makes a legitimate request through the same TCP connection, the attacker's request is already waiting and is interpreted as a response to the victim's request. The attacker then sends a second request to the server, to which the proxy server responds with the server generated request intended for the victim, thereby compromising any sensitive information in the headers or body of the response intended for the victim.

**Cookie Manipulation:** When combined with attacks like Cross-Site Request Forgery, attackers can change, add to, or even overwrite a legitimate user's cookies.

**Open Redirect:** Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

**Recommendations:**

The solution to Header Manipulation is to ensure that input validation occurs in the correct places and checks for the correct properties.

Since Header Manipulation vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating responses dynamically, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for Header Manipulation.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for Header Manipulation is generally relatively easy. Despite its value, input validation for Header Manipulation does not take theplace of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent Header Manipulation vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for Header Manipulation is to create a whitelist of safe characters that are allowed to appear in HTTP response headers and accept input composed exclusively of characters in the approved set. For example, a valid name might only include alpha-numeric characters or an account number might only include digits 0-9.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning in HTTP response headers. Although the CR and LF characters are at the heart of an HTTP response splitting attack, other characters, such as ':' (colon) and '=' (equal), have special meaning in response headers as well.

Once you identify the correct points in an application to perform validation for Header Manipulation attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. The application should reject any input destined to be included in HTTP response headers that contains special characters, particularly CR and LF, as invalid.

Many application servers attempt to limit an application's exposure to HTTP response splitting vulnerabilities by providing implementations for the functions responsible for setting HTTP headers and cookies that perform validation for the characters essential to an HTTP response splitting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

**Tips:**

1. Many HttpServletRequest implementations return a URL-encoded string from getHeader(), will not cause a HTTP response splitting issue unless it is decoded first because the CR and LF characters will not carry a meta-meaning in their encoded form. However, this behavior is not specified in the J2EE standard and varies by implementation. Furthermore, even encoded user input returned from getHeader() can lead to other vulnerabilities, including open redirects and other HTTP header tampering.

2. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Spring MVC are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the HP Fortify Software Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

# 17. Insecure Randomness:

**Explanation:**

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in security-sensitive context.

Computers are deterministic machines, and as such are unable to produce true randomness. Pseudorandom Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated.

There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and forms an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between it and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts, where its use can lead to serious vulnerabilities such as easy-to-guess temporary passwords, predictable cryptographic keys, session hijacking, and DNS spoofing.

**Example:** The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

String GenerateReceiptURL(String baseUrl) {

Random ranGen = new Random();

ranGen.setSeed((new Date()).getTime());

return (baseUrl + ranGen.nextInt(400000000) + ".html");

}

This code uses the Random.nextInt() function to generate "unique" identifiers for the receipt pages it generates. Because Random.nextInt() is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

**Recommendations:**

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Values such as the current time offer only negligible entropy and should not be used.)

The Java language provides a cryptographic PRNG in java.security.SecureRandom. As is the case with other algorithm-based classes in java.security, SecureRandom provides an implementation-independent wrapper around a particular set of algorithms. When you request an instance of a SecureRandom object using SecureRandom.getInstance(), you can request a specific implementation of the algorithm. If the algorithm is available, then it is given as a SecureRandom object. If it is unavailable or if you do not specify a particular implementation, then you are given a SecureRandom implementation selected by the system.

Sun provides a single SecureRandom implementation with the Java distribution named SHA1PRNG, which Sun describes as computing:

"The SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used [1]."

However, the specifics of the Sun implementation of the SHA1PRNG algorithm are poorly documented, and it is unclear what sources of entropy the implementation uses and therefore what amount of true randomness exists in its output. Although there is speculation on the Web about the Sun implementation, there is no evidence to contradict the claim that the algorithm is cryptographically strong and can be used safely in security-sensitive contexts.

# 18. JSON Injection

**Explanation:**

JSON injection occurs when:

1. Data enters a program from an untrusted source.

2. The data is written to a JSON stream.

Applications typically use JSON to store data or send messages. When used to store data, JSON is often treated like cached data and may potentially contain sensitive information. When used to send messages, JSON is often used in conjunction with a RESTful service and can be used to transmit sensitive information such as authentication credentials.

The semantics of JSON documents and messages can be altered if an application constructs JSON from unvalidated input. In a relatively benign case, an attacker may be able to insert extraneous elements that cause an application to throw an exception while parsing a JSON document or request. In a more serious case, such as that involving JSON injection, an attacker may be able to insert extraneous elements that allow for the predictable manipulation of business critical values within a JSON document or request. In some cases, JSON injection can lead to cross-site scripting or dynamic code evaluation.

**Example 1:** The following Java code uses Jackson to serialize user account authentication information for non-privileged users (those with a role of "default" as opposed to privileged users with a role of "admin") from user-controlled input variables username and password to the JSON file located at ~/user\_info.json:

...

JsonFactory jfactory = new JsonFactory();

JsonGenerator jGenerator = jfactory.createJsonGenerator(new File("~/user\_info.json"), JsonEncoding.UTF8);

jGenerator.writeStartObject();

jGenerator.writeFieldName("username");

jGenerator.writeRawValue("\"" + username + "\"");

jGenerator.writeFieldName("password");

jGenerator.writeRawValue("\"" + password + "\"");

jGenerator.writeFieldName("role");

jGenerator.writeRawValue("\"default\"");

jGenerator.writeEndObject();

jGenerator.close();

**Recommendations:**

When writing user supplied data to JSON some guidelines should be followed:

1. Don't create JSON attributes whose names are derived from user input.

2. Ensure that all serialization to JSON is performed using a safe serialization function that delimits untrusted data within single or double quotes and escapes any special characters.

**Example 2:** The following Java code implements the same functionality as that in Example 1, but instead uses JsonGenerator.writeString() rather than JsonGenerator.writeRawValue() to serialize the data, therefore ensuring that any untrusted data is properly delimited and escaped:

...

JsonFactory jfactory = new JsonFactory();

JsonGenerator jGenerator = jfactory.createJsonGenerator(new File("~/user\_info.json"), JsonEncoding.UTF8);

jGenerator.writeStartObject();

jGenerator.writeFieldName("username");

jGenerator.writeString(username);

jGenerator.writeFieldName("password");

jGenerator.writeString(password);

jGenerator.writeFieldName("role");

jGenerator.writeString("default");

jGenerator.writeEndObject();

jGenerator.close();

# 19. Poor Error Handling: Return inside Finally

**Explanation:**

A return statement inside a finally block will cause any exception that might be thrown in the try block to be discarded.

**Example 1:** In the following code excerpt, the MagicException thrown by the second call to doMagic with true passed to it will never be delivered to the caller. The return statement inside the finally block will cause the exception to be discarded.

public class MagicTrick {

public static class MagicException extends Exception { }

public static void main(String[] args) {

System.out.println("Watch as this magical code makes an " +

"exception disappear before your very eyes!");

System.out.println("First, the kind of exception handling " +

"you're used to:");

try {

doMagic(false);

} catch (MagicException e) {

// An exception will be caught here

e.printStackTrace();

}

System.out.println("Now, the magic:");

try {

doMagic(true);

} catch (MagicException e) {

// No exception caught here, the finally block ate it

e.printStackTrace();

}

System.out.println("tada!");

}

public static void doMagic(boolean returnFromFinally)

throws MagicException {

try {

throw new MagicException();

}

finally {

if (returnFromFinally) {

return;

}

}

}

}

**Recommendations:**

Move the return statement outside the finally block. If a value from inside the finally block must be returned, simply assign it to a local variable and return this value after the finally block executes.

**Tips:**

1. HP Fortify Static Code Analyzer will detect explicit throws as well as calls to methods that throw uncaught exceptions in finally blocks.

# 20. Poor style: Value Never Read

**Explanation:**

This variable's value is not used. After the assignment, the variable is either assigned another value or goes out of scope.

**Example:** The following code excerpt assigns to the variable r and then overwrites the value without using it.

r = getName();

r = getNewBuffer(buf);

**Recommendations:**

Remove unnecessary assignments in order to make the code easier to understand and maintain.

# 21. Redundant Null check

**Explanation:**

Null pointer exceptions usually occur when one or more of the programmer's assumptions is violated. Specifically, dereference-after-check errors occur when a program makes an explicit check for null, but proceeds to dereference the object when it is known to be null. Errors of this type are often the result of a typo or programmer oversight.

Most null pointer issues result in general software reliability problems, but if attackers can intentionally cause the program to dereference a null pointer, they can use the resulting exception to mount a denial of service attack or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks.

**Example 1:** In the following code, the programmer confirms that the variable foo is null and subsequently dereferences it erroneously. If foo is null when it is checked in the if statement, then a null dereference will occur, thereby causing a null pointer exception.

if (foo == null) {

foo.setBar(val);

...

}

**Recommendations:**

Implement careful checks before dereferencing objects that might be null. When possible, abstract null checks into wrappers around code that manipulates resources to ensure that they are applied in all cases and to minimize the places where mistakes can occur.

# 22. Server-side Request Forgery

**Explanation:**

A Server-Side Request Forgery occurs when an attacker can influence a network connection made by the application server. The network connection will originate from the application server internal IP and an attacker will be able to use this connection to bypass network controls and scan or attack internal resources that are not otherwise exposed.

**Example:** In the following example, an attacker will be able to control the URL the server is connecting to.

String url = request.getParameter("url");

CloseableHttpClient httpclient = HttpClients.createDefault();

HttpGet httpGet = new HttpGet(url);

CloseableHttpResponse response1 = httpclient.execute(httpGet);

The ability of the attacker to hijack the network connection will depend on the specific part of the URI that he can control and on libraries used to stablish the connection. For example, controlling the URI scheme will let the attacker use protocols different from http or https like:

- up://

- ldap://

- jar://

- gopher://

- mailto://

- ssh2://

- telnet://

- expect://

An attacker will be able to leverage this hijacked network connection to perform the following attacks:

- Port Scanning of intranet resources.

- Bypass firewalls.

- Attack vulnerable programs running on the application server or on the Intranet.

- Attack internal/external web applications using Injection attacks or CSRF.

- Access local files using file:// scheme.

- On Windows systems, file:// scheme and UNC paths can allow an attacker to scan and access internal shares.

- Perform a DNS cache poisoning attack.

**Recommendations:**

Do not establish network connections based on user-controlled data and ensure that the request is being sent to the expected destination. If user data is necessary to build the destination URI, use a level of indirection: create a list of legitimate resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to keep track of. Programmers often resort to blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a whitelist of characters that are allowed to appear in the resource name and accept input composed exclusively of characters in the approved set.

Also, if required, make sure that the user input is only used to specify a resource on the target system but that the URI scheme, host, and port is controlled by the application. This way the damage that an attacker can do will be significantly reduced.

# 23. Unreleased Resource

**Explanation:**

The program can potentially fail to release a system resource.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker may be able to launch a denial of service attack by depleting the resource pool.

**Example:** The following method never closes the file handle it opens. The finalize() method for FileInputStream eventually calls close(), but there is no guarantee as to how long it will take before the finalize() method will be invoked. In a busy environment, this can result in the JVM using up all of its file handles.

private void processFile(String fName) throws FileNotFoundException, IOException {

FileInputStream fis = new FileInputStream(fName);

int sz;

byte[] byteArray = new byte[BLOCK\_SIZE];

while ((sz = fis.read(byteArray)) != -1) {

processBytes(byteArray, sz);

}

**Recommendations:**

1. Never rely on finalize() to reclaim resources. In order for an object's finalize() method to be invoked, the garbage collector must determine that the object is eligible for garbage collection. Because the garbage collector is not required to run unless the JVM is low on memory, there is no guarantee that an object's finalize() method will be invoked in an expedient fashion. When the garbage collector finally does run, it may cause a large number of resources to be reclaimed in a short period of time, which can lead to "bursty" performance and lower overall system throughput. This effect becomes more pronounced as the load on the system increases.

Finally, if it is possible for a resource reclamation operation to hang (if it requires communicating over a network to a database, for example), then the thread that is executing the finalize() method will hang.

2. Release resources in a finally block. The code for the Example should be rewritten as follows:

public void processFile(String fName) throws FileNotFoundException, IOException {

FileInputStream fis;

try {

fis = new FileInputStream(fName);

int sz;

byte[] byteArray = new byte[BLOCK\_SIZE];

while ((sz = fis.read(byteArray)) != -1) {

processBytes(byteArray, sz);

}

}

finally {

if (fis != null) {

safeClose(fis);

}

}

}

public static void safeClose(FileInputStream fis) {

if (fis != null) {

try {

fis.close();

} catch (IOException e) {

log(e);

}

}

}

This solution uses a helper function to log the exceptions that might occur when trying to close the stream. Presumably this helper function will be reused whenever a stream needs to be closed.

Also, the processFile method does not initialize the fis object to null. Instead, it checks to ensure that fis is not null before calling safeClose(). Without the null check, the Java compiler reports that fis might not be initialized. This choice takes advantage of Java's ability to detect uninitialized variables. If fis is initialized to null in a more complex method, cases in which fis is used without being initialized will not be detected by the compiler.