Fortify Security Report

Mar 18, 2023

Executive Summary

Issues Overview

On Mar 18, 2023, a source code review was performed over the juice-shop-31854 code base. 646 files, 32,553 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 216 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Critical	132
Low	60
High	21
Medium	3

Recommendations and Conclusions

The Issues Category section provides Fortify recommendations for addressing issues at a generic level. The recommendations for specific fixes can be extrapolated from those generic recommendations by the development group.

Project Summary

Code Base Summary

Code location: /home/devops/git/juice-shop
Number of Files: 646
Lines of Code: 32553
Build Label: <No Build Label>

Scan Information

Scan time: 06:22
SCA Engine version: 20.1.0.0158
Machine Name: 899380d3b2ac
Username running scan: devops

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface

Attack Surface:
Environment Variables:
null.null.null

File System:
null.null.open
null.file.read
null.file.readlines

Private Information:
null.null.null

System Information:
null.null.null
null.null.checkDiffs
null.null.lambda
null.null.lambda
null.null.setUserName
null.null.~file_function

Web:

null.null.null

Filter Set Summary

Current Enabled Filter Set:
Security Auditor View

Filter Set Details:

Folder Filters:
If [fortify priority order] contains critical Then set folder to Critical
If [fortify priority order] contains high Then set folder to High
If [fortify priority order] contains medium Then set folder to Medium
If [fortify priority order] contains low Then set folder to Low

Audit Guide Summary

Audit guide not enabled

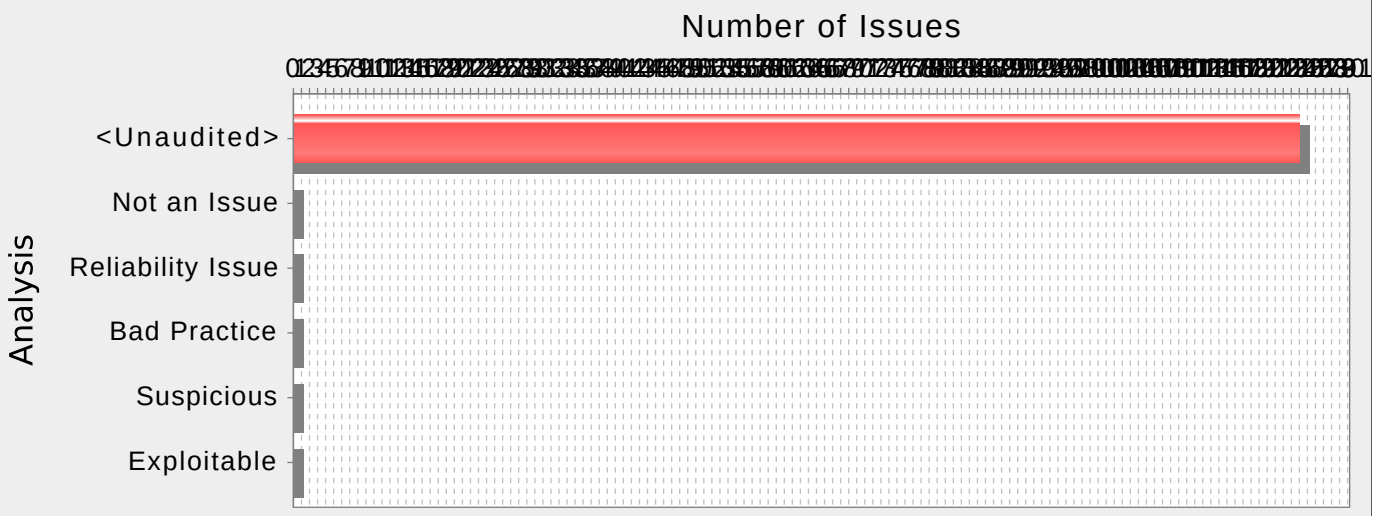
Results Outline

Overall number of results

The scan found 216 issues.

Vulnerability Examples by Category

Category: Password Management: Hardcoded Password (125 Issues)



Abstract:

Hardcoded passwords can compromise system security in a way that is not easy to remedy.

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. After 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 must choose between security and availability.

Example: The following code uses a hardcoded password to connect to an application and retrieve address book entries:

```
...
obj = new XMLHttpRequest();
obj.open('GET','/fetchusers.jsp?id='+form.id.value,'true','scott','tiger');
...
```

This code will run successfully, but anyone who accesses the containing web page can view the password.

Recommendations:

Passwords should never be hardcoded and should generally be obfuscated and managed in an external source. Storing passwords in plain text anywhere on the web site allows anyone with sufficient permissions to read and potentially misuse the password. For JavaScript calls that require passwords, it is better to prompt the user for the password at connection time.

Tips:

1. Avoid hardcoding passwords in source code and avoid using default passwords. If a hardcoded password is the default, require that it be changed and remove it from the source code.

2. To identify null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the Fortify 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.

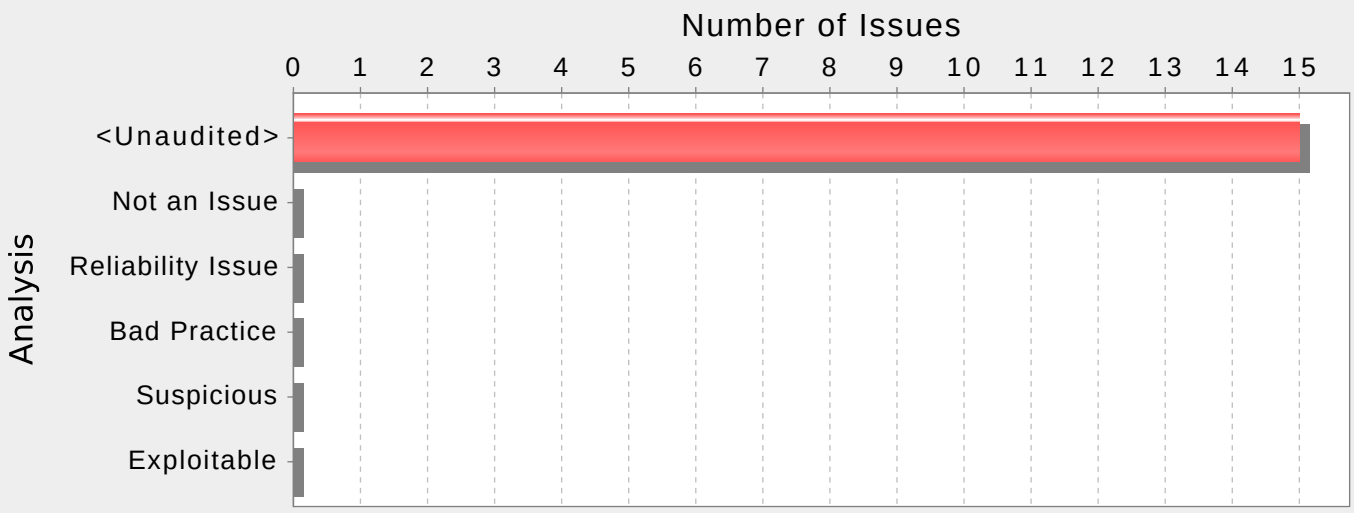
oauth.component.spec.ts, line 85 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords can compromise system security in a way that is not easy to remedy.		
Sink:	oauth.component.spec.ts:85 FieldAccess: passwordRepeat()		

```
83      userService.oauthLogin.and.returnValue(of({ email: 'test@test.com' }))
84      component.ngOnInit()
```

```
85                      expect(userService.save).toHaveBeenCalledWith({ email: 'test@test.com', password:
                      'bW9jLnRzZXRAdHNldA==', passwordRepeat: 'bW9jLnRzZXRAdHNldA==' })
86                    }))
```

Category: Insecure Randomness (15 Issues)



Abstract:

The random number generator implemented by random() cannot withstand a cryptographic attack.

Explanation:

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a 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 form 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 the generated random value 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.

```
function genReceiptURL (baseURL){
var randNum = Math.random();
var receiptURL = baseURL + randNum + ".html";
return receiptURL;
}
```

This code uses the Math.random() function to generate "unique" identifiers for the receipt pages it generates. Since Math.random() 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. (Do not use values such as the current time because it offers only negligible entropy.)

In JavaScript, the typical recommendation is to use the window.crypto.random() function in the Mozilla API. However, this method does not work in many browsers, including more recent versions of Mozilla Firefox. There is currently no cross-browser solution for a robust cryptographic PRNG. In the meantime, consider handling any PRNG functionality outside of JavaScript.

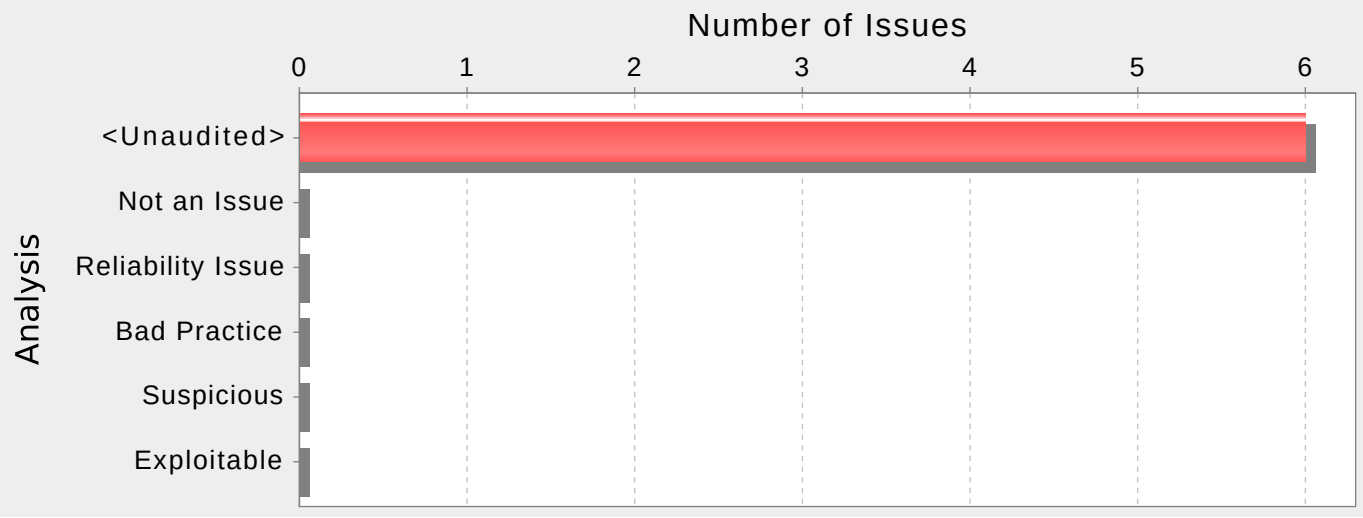
code-fixes.component.ts, line 49 (Insecure Randomness)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The random number generator implemented by random() cannot withstand a cryptographic attack.		
Sink:	code-fixes.component.ts:49 FunctionPointerCall: random()		

```
47          index++
```

```
48                        }
49                        let randomRotation = Math.random() * 100
50                        while (randomRotation > 0) {
51                          const end = this.randomFixes[this.randomFixes.length - 1]
```

Category: Key Management: Hardcoded Encryption Key (6 Issues)



Abstract:

Hardcoded encryption keys can compromise security in a way that is not easy to remedy.

Explanation:

It is never a good idea to hardcode an encryption key because it allows all of the project's developers to view the encryption key, and makes fixing the problem extremely difficult. After the code is in production, a software patch is required to change the encryption key. If the account that is protected by the encryption key is compromised, the owners of the system must choose between security and availability.

Example 1: The following code uses a hardcoded encryption key:

```
...
var crypto = require('crypto');
var encryptionKey = "lakdsljkalkjlksdfkl";
var algorithm = 'aes-256-ctr';
var cipher = crypto.createCipher(algorithm, encryptionKey);
...
```

Anyone with access to the code has access to the encryption key. After the application has shipped, there is no way to change the encryption key unless the program is patched. An employee with access to this information can use it to break into the system. If attackers had access to the executable for the application, they could extract the encryption key value.

Recommendations:

Encryption keys should never be hardcoded and should be obfuscated and managed in an external source. Storing encryption keys in plain text anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the encryption key.

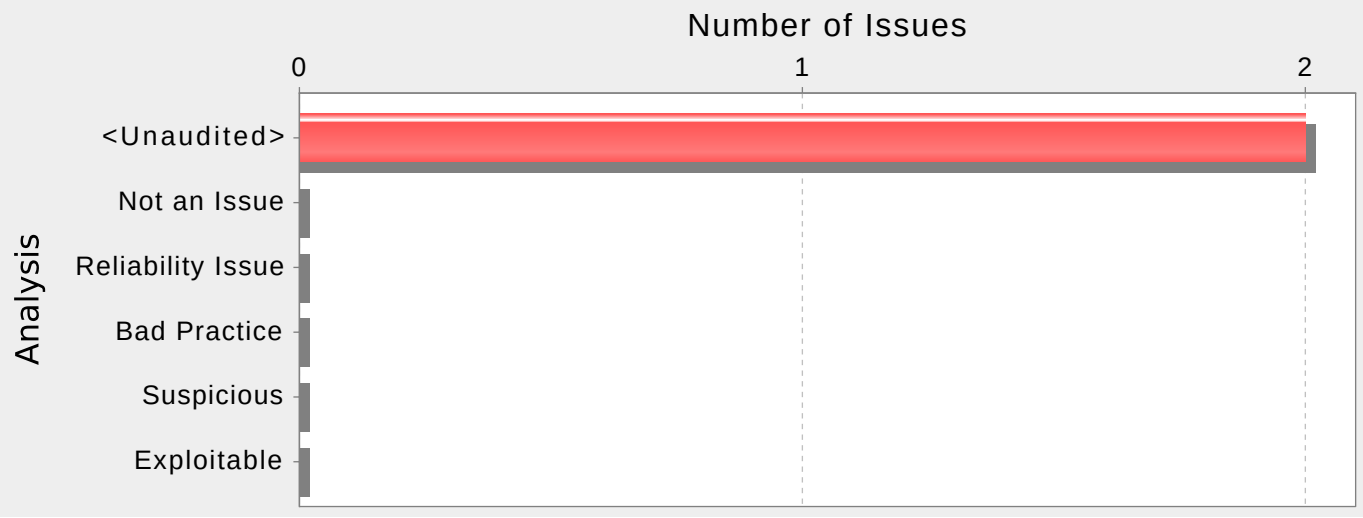
vulnCodeFixesSpec.ts, line 34 (Key Management: Hardcoded Encryption Key)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	Hardcoded encryption keys can compromise security in a way that is not easy to remedy.		

Sink: vulnCodeFixesSpec.ts:34 FieldAccess: key()

```
32      return frisby.post(URL + '/snippets/fixes', {
33        body: {
34          key: 'doesNotExistChallenge',
35          selectedFix: 1
36        }
```

Category: Insecure Transport (2 Issues)



Abstract:

The call to get() in productReviewApiSpec.ts on line 63 uses an unencrypted protocol instead of an encrypted protocol to communicate with the server.

Explanation:

All communication over HTTP, FTP, or gopher is unauthenticated and unencrypted. It is therefore subject to compromise, especially in the mobile environment where devices frequently connect to unsecured, public, wireless networks using WiFi connections.

Example 1: The following example reads data using the HTTP protocol (instead of using HTTPS).

```
var http = require('http');
...
http.request(options, function(res){
...
});
...
```

The incoming http.IncomingMessage object,res, may have been compromised as it is delivered over an unencrypted and unauthenticated channel.

Recommendations:

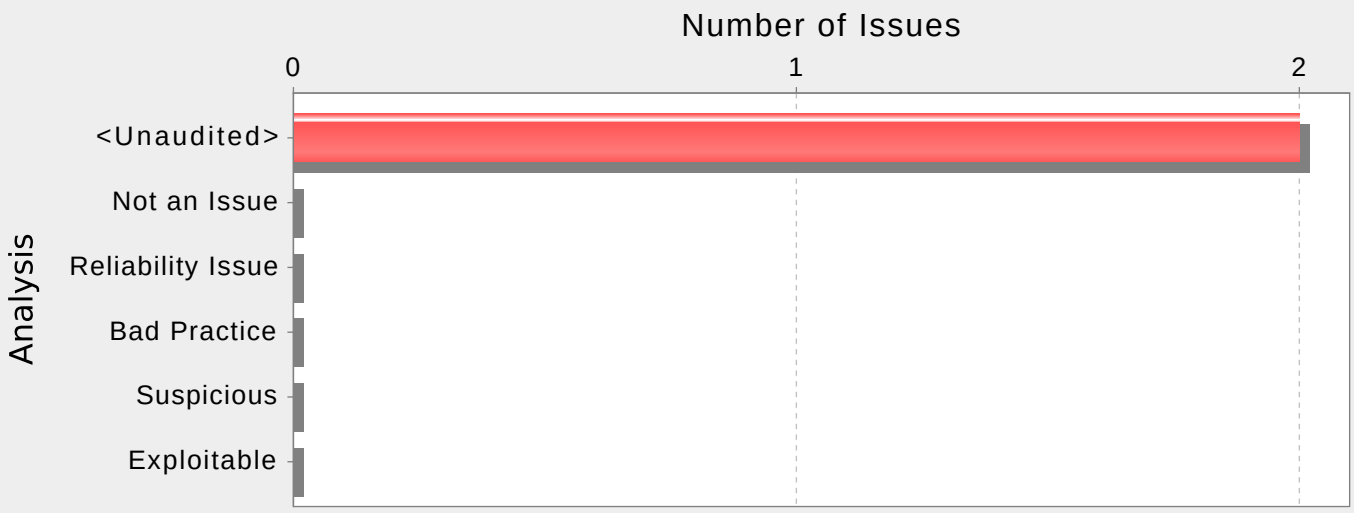
Use secure protocols such as HTTPS to exchange data with the server whenever possible.

productReviewApiSpec.ts, line 63 (Insecure Transport)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	The call to get() in productReviewApiSpec.ts on line 63 uses an unencrypted protocol instead of an encrypted protocol to communicate with the server.		
Sink:	productReviewApiSpec.ts:63 FunctionPointerCall: get()		

```
61
62                    beforeAll((done) => {
63                      http.get(`${REST_URL}/products/1/reviews`, (res: IncomingMessage) => {
64                        let body = ''
```

Category: Password Management: Insecure Submission (2 Issues)



Abstract:

The form in two-factor-auth.component.html submits a password as part of an HTTP GET request on line 27, which will cause the password to be displayed, logged, and stored in the browser cache.

Explanation:

By convention, the parameters associated with an HTTP GET request are not treated as sensitive data, so web servers log them, proxies cache them, and web browsers do not make an effort to conceal them. Sending a password or other sensitive data as part of an HTTP GET will likely cause the data to be mishandled and potentially revealed to an attacker.

Example 1: In the following example, new user password is submitted via an HTTP GET request.

```
<form method="get">
Name of new user: <input type="text" name="username">
Password for new user: <input type="password" name="user_passwd">
<input type="submit" name="action" value="Create User">
</form>
```

Also, note that the default value of the method attributed is GET, thus omitting the attribute results in the same outcome.

Recommendations:

Avoid sending sensitive data, such as passwords, via an HTTP GET request. Sensitive data should travel from the browser to the server using HTTP POST, not HTTP GET.

Example 2: In the following example, new user password is submitted via an HTTP POST request.

```
<form method="post">
Name of new user: <input type="text" name="username">
Password for new user: <input type="password" name="user_passwd">
<input type="submit" name="action" value="Create User">
</form>
```

HTML5 adds the ability to specify the formmethod attribute as part of the submit and image input tags, and the value of this attribute overrides the value of the method attribute of the corresponding form tag.

Example 3: In the following example, new user password is also submitted via an HTTP POST request, which is specified by the formmethod attribute of the submit input tag.

```
<form method="get">
Name of new user: <input type="text" name="username">
Password for new user: <input type="password" name="user_passwd">
<input type="submit" name="action" value="Create User" formmethod="post">
</form>
```

However, be aware that if the value of the formmethod attribute is set to get, the form will be submitted via an HTTP GET request no matter what the method attribute of the corresponding form tag specifies.

Avoid sending sensitive data via an HTTP redirect, because it causes the user's web browser to issue an HTTP GET request. The application should either store the sensitive data in a session object so that it does not need to be transmitted back to the browser or simply discard the data and ask the user to enter it again. Other options, such as embedding the sensitive data in a web page that automatically posts its data, are also problematic because the web page may be cached by a proxy or by the web browser. As is often the case, security may need to trump usability in this instance.

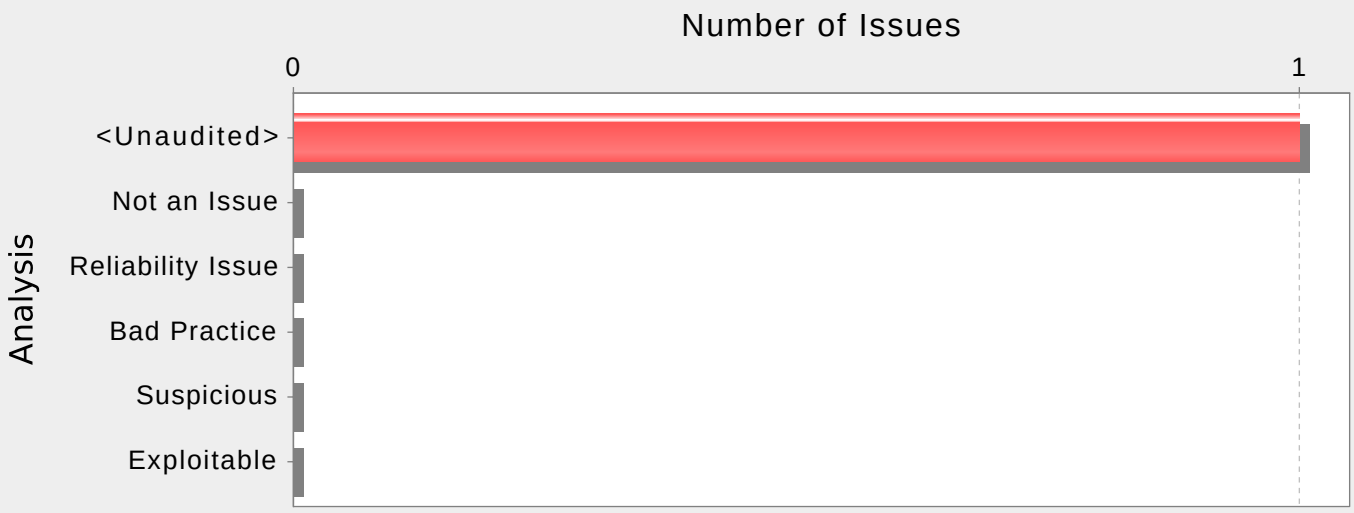
two-factor-auth.component.html, line 27 (Password Management: Insecure Submission)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	The form in two-factor-auth.component.html submits a password as part of an HTTP GET request on line 27, which will cause the password to be displayed, logged, and stored in the browser cache.		

Sink: two-factor-auth.component.html:27 null()

```
25              <mat-form-field appearance="outline" color="accent">
26                <mat-label translate>LABEL_CURRENT_PASSWORD</mat-label>
27                <input id="currentPasswordDisable" formControlName="passwordControl"
type="password" matInput
28                       aria-label="Field to enter the current password"
29                       placeholder="{{'MANDATORY_CURRENT_PASSWORD' | translate }}"
```

Category: Password Management: Empty Password (1 Issues)



Abstract:

Empty passwords may compromise system security in a way that cannot be easily remedied.

Explanation:

It is never a good idea to have an empty 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 empty password is compromised, the owners of the system must choose between security and availability.

Example: The following code has an empty password to connect to an application and retrieve address book entries:

```
...
obj = new XMLHttpRequest();
obj.open('GET','/fetchusers.jsp?id='+form.id.value,'true','scott','');
...
```

This code will run successfully, but anyone can access when they know the username.

Recommendations:

Passwords should never be empty and should generally be obfuscated and managed in an external source. Storing passwords in plain text anywhere on the web site allows anyone with sufficient permissions to read and potentially misuse the password. For JavaScript calls that require passwords, it is better to prompt the user for the password at connection time.

Tips:

1. Avoid empty passwords in source code and avoid using default passwords. If an empty password is the default, require that it be changed and remove it from the source code.

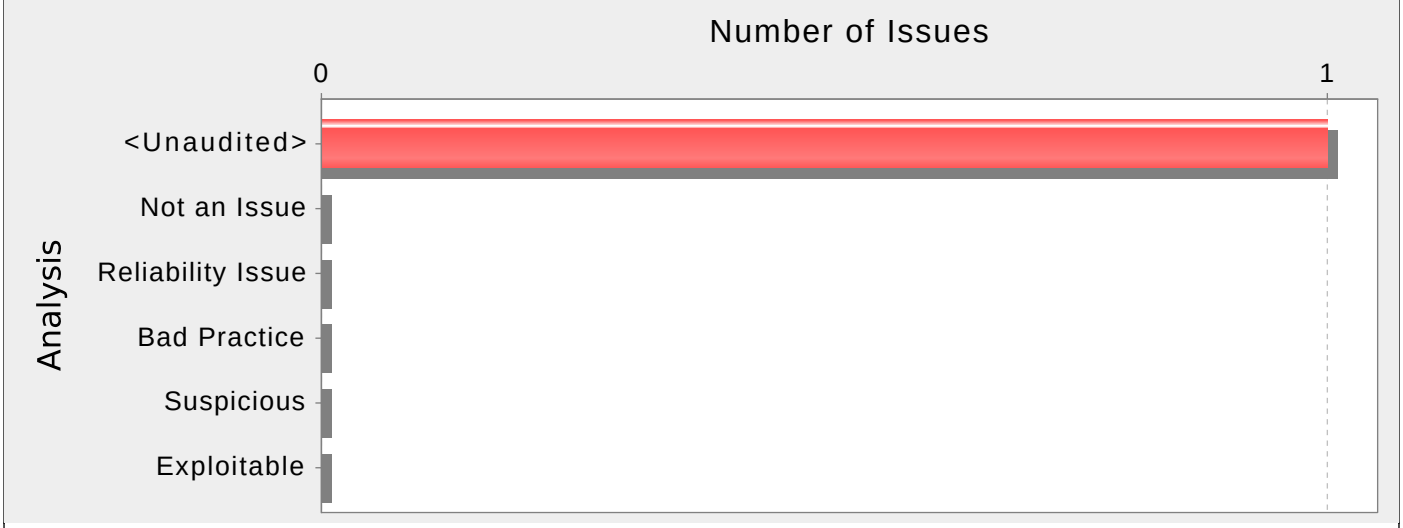
2. To identify null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the Fortify Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named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

register.component.spec.ts, line 115 (Password Management: Empty Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Empty passwords may compromise system security in a way that cannot be easily remedied.		
Sink:	register.component.spec.ts:115 VariableAccess: password()		

```
113
114   it('password should not be more than 20 characters', () => {
115     let password: string = ''
116     for (let i = 0; i < 41; i++) {
117       password += 'a'
```

Category: Privacy Violation (1 Issues)



Abstract:

The file authenticatedUsers.ts mishandles confidential information on line 24, which can compromise user privacy and is often illegal.

Explanation:

Privacy violations occur when:

1. Private user information enters the program.

2. The data is written to an external location, such as the console, file system, or network.

Example: The following code stores user's plain text password to the local storage.

localStorage.setItem('password', password);

Although many developers treat the local storage as a safe location for data, it should not be trusted implicitly, particularly when privacy is a concern.

Private data can enter a program in a variety of ways:

- Directly from the user in the form of a password or personal information

- Accessed from a database or other data store by the application

- Indirectly from a partner or other third party

Sometimes data that is not labeled as private can have a privacy implication in a different context. For example, student identification numbers are usually not considered private because there is no explicit and publicly-available mapping to an individual student's personal information. However, if a school generates identification numbers based on student social security numbers, then the identification numbers should be considered private.

Security and privacy concerns often seem to compete with each other. From a security perspective, you should record all important operations so that any anomalous activity can later be identified. However, when private data is involved, this practice can create risk.

Although there are many ways in which private data can be handled unsafely, a common risk stems from misplaced trust. Programmers often trust the operating environment in which a program runs, and therefore believe that it is acceptable to store private information on the file system, in the registry, or in other locally-controlled resources. However, even if access to certain resources is restricted, this does not guarantee that the individuals who do have access can be trusted. For example, in 2004, an unscrupulous employee at AOL sold approximately 92 million private customer email addresses to a spammer marketing an offshore gambling web site [1].

In response to such high-profile exploits, the collection and management of private data is becoming increasingly regulated. Depending on its location, the type of business it conducts, and the nature of any private data it handles, an organization may be required to comply with one or more of the following federal and state regulations:

- Safe Harbor Privacy Framework [3]

- Gramm-Leach Bliley Act (GLBA) [4]

- Health Insurance Portability and Accountability Act (HIPAA) [5]

- California SB-1386 [6]

Despite these regulations, privacy violations continue to occur with alarming frequency.

Recommendations:

When security and privacy demands clash, privacy should usually be given the higher priority. To accomplish this and still maintain required security information, cleanse any private information before it exits the program.

To enforce good privacy management, develop and strictly adhere to internal privacy guidelines. The guidelines should specifically describe how an application should handle private data. If your organization is regulated by federal or state law, ensure that your privacy guidelines are sufficiently strenuous to meet the legal requirements. Even if your organization is not regulated, you must protect private information or risk losing customer confidence.

The best policy with respect to private data is to minimize its exposure. Applications, processes, and employees should not be granted access to any private data unless the access is required for the tasks that they are to perform. Just as the principle of least privilege dictates that no operation should be performed with more than the necessary privileges, access to private data should be restricted to the smallest possible group.

Tips:

1. As part of any thorough audit for privacy violations, ensure that custom rules are written to identify all sources of private or otherwise sensitive information entering the program. Most sources of private data cannot be identified automatically. Without custom rules, your check for privacy violations is likely to be substantially incomplete.

authenticatedUsers.ts, line 24 (Privacy Violation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	The file authenticatedUsers.ts mishandles confidential information on line 24, which can compromise user privacy and is often illegal.		

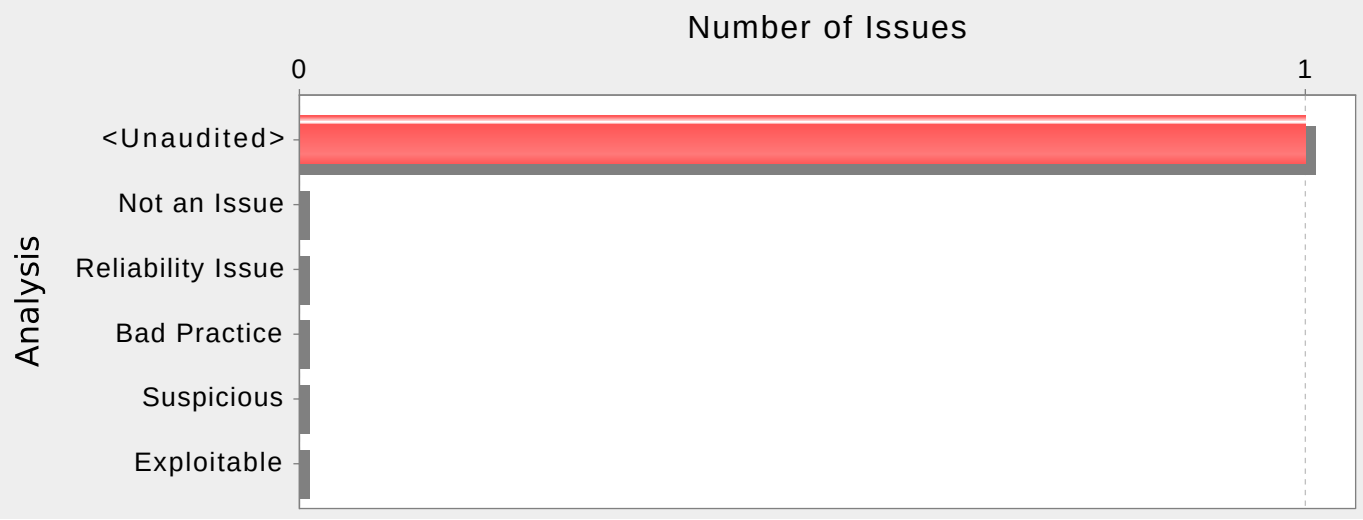
Source: authenticatedUsers.ts:18 Read user.password()

```
16          user.token = security.authenticatedUsers.tokenOf(user)
17          if (user.password) {
18            user.password =****** '*')
19          }
20          if (user.totpSecret) {
```

Sink: authenticatedUsers.ts:24 ~JS_Generic.json()

```
22          }
23        })
24        res.json(usersWithLoginStatus)
25      }).catch((error: Error) => {
26        next(error)
```

Category: Privacy Violation: Shoulder Surfing (1 Issues)



Abstract:

The form in login.component.html writes the password in plain text to the screen on line 27.

Explanation:

A password need not be viewable to its owner, and must not be viewable to others. If a password is displayed in plain text, anyone in the vicinity could see and use it to compromise the system. In computer security, shoulder surfing refers to using direct observation techniques, such as looking over someone's shoulder, to get information. Shoulder surfing is particularly effective in crowded, public environments. This threat particularly applies to mobile devices, which are generally intended for use in all environments, both private and public.

Recommendations:

Never show a password in plain text. Obscure the characters of the field as dots or stars instead of legible characters.

Example: In HTML forms, set the type attribute to password for sensitive inputs.

<input name="password" type="password" />

Note that the default value of the type attributed is text, not password. Thus do not omit the attribute when dealing with sensitive inputs.

login.component.html, line 27 (Privacy Violation: Shoulder Surfing)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	The form in login.component.html writes the password in plain text to the screen on line 27.		
Sink:	login.component.html:27 null()		

```
25          <mat-form-field color="accent" appearance="outline">
26            <mat-label translate>LABEL_PASSWORD</mat-label>
27            <input id="password" #password name="password"
28                   [formControl]="passwordControl"
29                   (focus)="this.error=null"
```

Issue Count by Category

Issues by Category	
Password Management: Hardcoded Password	125
Cross-Site Request Forgery	36
Insecure Randomness	15
Poor Logging Practice: Use of a System Output Stream	10
Key Management: Hardcoded Encryption Key	6
Password Management: Password in Comment	6
System Information Leak: Internal	5
Often Misused: File Upload	3
Insecure Transport	2
Password Management: Insecure Submission	2
Denial of Service	1
Password Management: Empty Password	1
Password Management: Null Password	1
Privacy Violation	1
Privacy Violation: Shoulder Surfing	1
Weak Cryptographic Hash	1

Issue Breakdown by Analysis

Issues by Analysis