Fortify Standalone Report Generator

Developer Workbook

AgregacionNumeralesFront-Fortify

Table of Contents

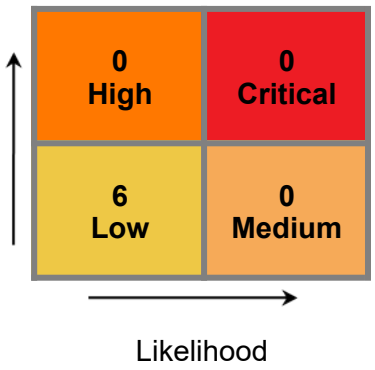
Executive Summary

This workbook is intended to provide all necessary details and information for a developer to understand and remediate the different issues discovered during the AgregacionNumeralesFront-Fortify project audit. The information contained in this workbook is targeted at project managers and developers.

This section provides an overview of the issues uncovered during analysis.

Project Name:	AgregacionNumeralesFr Fortify
Project Version:	
SCA:	Results Present
WebInspect:	Results Not Present
WebInspect Agent:	Results Not Present
Other:	Results Not Present

Issues by Priority



Top Ten Critical Categories

This project does not contain any critical issues

Project Description

This section provides an overview of the Fortify scan engines used for this project, as well as the project meta-information.

SCA

Date of Last Analysis:	Nov 6, 2020, 12:01 PM	**Engine Version:**	20.1.0.0158
Host Name:	WSCAC1D	**Certification:**	VALID
Number of Files:	133	**Lines of Code:**	3,293

Rulepack Name	Rulepack Version
Fortify Secure Coding Rules, Core, JavaScript	2020.3.0.0009
Fortify Secure Coding Rules, Extended, Configuration	2020.3.0.0009
Fortify Secure Coding Rules, Extended, Content	2020.3.0.0009
Fortify Secure Coding Rules, Extended, JavaScript	2020.3.0.0009

Issue Breakdown by Fortify Categories

The following table depicts a summary of all issues grouped vertically by Fortify Category. For each category, the total number of issues is shown by Fortify Priority Order, including information about the number of audited issues.

Category	Fortify Priority (audited/total)				Total Issues
	Critical	High	Medium	Low	
Cross-Site Request Forgery	0	0	0	0 / 4	0 / 4
System Information Leak: Internal	0	0	0	0 / 2	0 / 2

Results Outline

Cross-Site Request Forgery (4 issues)

Abstract

Form posts must contain a user-specific secret in order to prevent an attacker from making unauthorized requests.

Explanation

A cross-site request forgery (CSRF) vulnerability occurs when: 1. A Web application uses session cookies. 2. The application acts on an HTTP request without verifying that the request was made with the user's consent. A nonce is a cryptographic random value that is sent with a message to prevent replay attacks. If the request does not contain a nonce that proves its provenance, the code that handles the request is vulnerable to a CSRF attack (unless it does not change the state of the application). This means a Web application that uses session cookies has to take special precautions in order to ensure that an attacker can't trick users into submitting bogus requests. Imagine a Web application that allows administrators to create new accounts by submitting this form:

```
<form method="POST" action="/new_user" >
  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>
```

An attacker might set up a Web site with the following:

```
<form method="POST" action="http://www.example.com/new_user">
  <input type="hidden" name="username" value="hacker">
  <input type="hidden" name="user_passwd" value="hacked">
</form>
<script>
  document.usr_form.submit();
</script>
```

If an administrator for `example.com` visits the malicious page while she has an active session on the site, she will unwittingly create an account for the attacker. This is a CSRF attack. It is possible because the application does not have a way to determine the provenance of the request. Any request could be a legitimate action chosen by the user or a faked action set up by an attacker. The attacker does not get to see the Web page that the bogus request generates, so the attack technique is only useful for requests that alter the state of the application. Applications that pass the session identifier in the URL rather than as a cookie do not have CSRF problems because there is no way for the attacker to access the session identifier and include it as part of the bogus request. CSRF is entry number five on the 2007 OWASP Top 10 list.

Recommendation

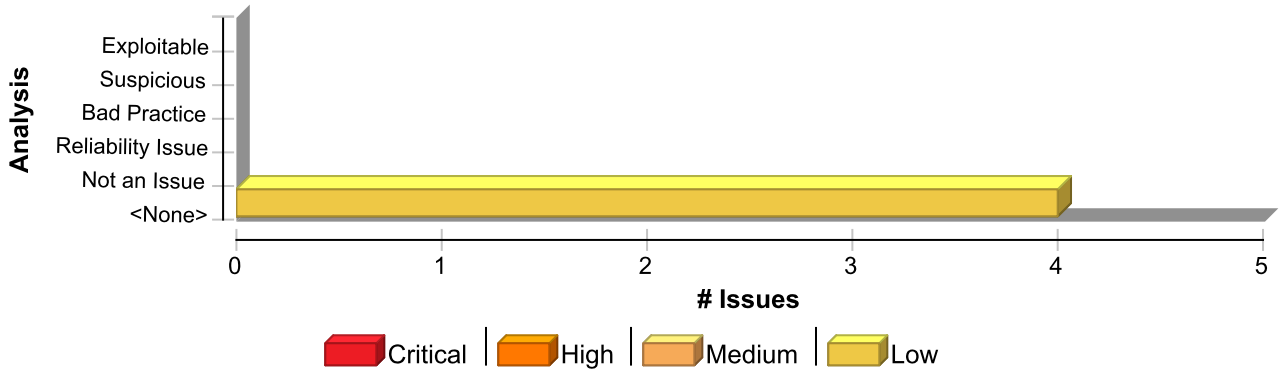
Applications that use session cookies must include some piece of information in every form post that the back-end code can use to validate the provenance of the request. One way to do that is to include a random request identifier or nonce, as follows:

```
  RequestBuilder rb = new RequestBuilder(RequestBuilder.POST, "/new_user");
  body = addToPost(body, new_username);
  body = addToPost(body, new_passwd);
  body = addToPost(body, request_id);
  rb.sendRequest(body, new NewAccountCallback(callback));
```

Then the back-end logic can validate the request identifier before processing the rest of the form data. When possible, the request identifier should be unique to each server request rather than shared across every request for a particular session. As with session identifiers, the harder it is for an attacker to guess

the request identifier, the harder it is to conduct a successful CSRF attack. The token should not be easily guessed and it should be protected in the same way that session tokens are protected, such as using SSLv3. Additional mitigation techniques include: **Framework protection:** Most modern web application frameworks embed CSRF protection and they will automatically include and verify CSRF tokens. **Use a Challenge-Response control:** Forcing the customer to respond to a challenge sent by the server is a strong defense against CSRF. Some of the challenges that can be used for this purpose are: CAPTCHAs, password re-authentication and one-time tokens. **Check HTTP Referer/Origin headers:** An attacker won't be able to spoof these headers while performing a CSRF attack. This makes these headers a useful method to prevent CSRF attacks. **Double-submit Session Cookie:** Sending the session ID Cookie as a hidden form value in addition to the actual session ID Cookie is a good protection against CSRF attacks. The server will check both values and make sure they are identical before processing the rest of the form data. If an attacker submits a form in behalf of a user, he won't be able to modify the session ID cookie value as per the same-origin-policy. **Limit Session Lifetime:** When accessing protected resources using a CSRF attack, the attack will only be valid as long as the session ID sent as part of the attack is still valid on the server. Limiting the Session lifetime will reduce the probability of a successful attack. The techniques described here can be defeated with XSS attacks. Effective CSRF mitigation includes XSS mitigation techniques.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
Cross-Site Request Forgery	4	0	0	4
Total	**4**	**0**	**0**	**4**

Cross-Site Request Forgery	**Low**
Package: .src.app.estructura.form	
src/src/app/estructura/form/form.component.html, line 11 (Cross-Site Request Forgery)	**Low**

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Content)

Sink Details

File: src/src/app/estructura/form/form.component.html:11
Taint Flags:

```
8  </div>
```

Cross-Site Request Forgery — Low

Package: .src.app.estructura.form

src/src/app/estructura/form/form.component.html, line 11 (Cross-Site Request Forgery) — Low

```
9  <mat-card class="push-bottom-xxl" tdMediaToggle="gt-xs" [mediaClasses]="['push']">
10  <ng-template tdLoading="estructura.form">
11  <form #estructuraForm="ngForm">
12  <td-steps mode="horizontal" linear="true">
13  <td-step #step1 label="Estructura" sublabel="Datos
Básicos" [state]="stateStep1" [active]="true">
14  <mat-card flex-gt-sm>
```

src/src/app/estructura/form/form.component.html, line 100 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Content)

Sink Details

File: src/src/app/estructura/form/form.component.html:100
Taint Flags:

```
97  <mat-card-title>Agregar Nodos</mat-card-title>
98  <mat-card-subtitle>{{Desc_Estructura}}</mat-card-subtitle>
99  <mat-card-content class="push-bottom-none">
100  <form #addForm="ngForm">
101  <div layout="row" layout-align="start center" class="pad-sm">
102
103  <div layout="row" class="example-list" flex>
```

Package: src.app.estructura.form

src/app/estructura/form/form.component.html, line 11 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Content)

Sink Details

File: src/app/estructura/form/form.component.html:11
Taint Flags:

```
8  </div>
9  <mat-card class="push-bottom-xxl" tdMediaToggle="gt-xs" [mediaClasses]="['push']">
10  <ng-template tdLoading="estructura.form">
11  <form #estructuraForm="ngForm">
12  <td-steps mode="horizontal" linear="true">
13  <td-step #step1 label="Estructura" sublabel="Datos
Básicos" [state]="stateStep1" [active]="true">
```

Cross-Site Request Forgery — Low

Package: src.app.estructura.form

src/app/estructura/form/form.component.html, line 11 (Cross-Site Request Forgery) — Low

```
14  <mat-card flex-gt-sm>
```

src/app/estructura/form/form.component.html, line 82 (Cross-Site Request Forgery) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Content)

Sink Details

File: src/app/estructura/form/form.component.html:82
Taint Flags:

```
79  <mat-card-title>Agregar Nodos</mat-card-title>
80  <mat-card-subtitle>{{Desc_Estructura}}</mat-card-subtitle>
81  <mat-card-content class="push-bottom-none">
82  <form #addForm="ngForm">
83  <div layout="row" layout-align="start center" class="pad-sm">
84
85  <div layout="row" class="example-list" flex>
```

System Information Leak: Internal (2 issues)

Abstract

Revealing system data or debugging information could enable an adversary to use system information to plan an attack.

Explanation

An internal information leak occurs when system data or debug information is sent to a local file, console, or screen via printing or logging. **Example 1:** The following code writes an exception to the standard error stream:

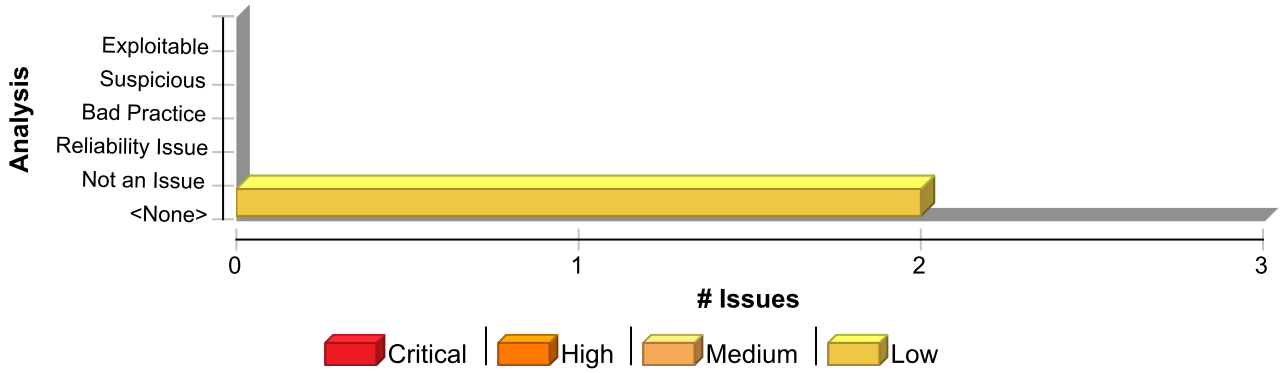
```
var http = require('http');
...

http.request(options, function(res){
  ...
}).on('error', function(e){
  console.log('There was a problem with the request: ' + e);
});
...
```

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a user. In some cases, the error message provides the attacker with the precise type of attack to which the system is vulnerable. 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 `Example 1`, 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.

Recommendation

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. Debug 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.

Issue Summary



Engine Breakdown

	SCA	WebInspect	SecurityScope	Total
System Information Leak: Internal	2	0	0	2
Total	**2**	**0**	**0**	**2**

System Information Leak: Internal — Low

Package: src

src/main.ts, line 12 (System Information Leak: Internal) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Data Flow)

Source Details

Source: lambda(0)
From: lambda
File: src/main.ts:12

```
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule)
12   .catch(err => console.error(err));
13
14 undefined
15 undefined
```

Sink Details

Sink: ~JS_Generic.error()
Enclosing Method: lambda()
File: src/main.ts:12
Taint Flags: SYSTEMINFO

```
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule)
12   .catch(err => console.error(err));
13
14 undefined
15 undefined
```

Package: src.src

src/src/main.ts, line 12 (System Information Leak: Internal) — Low

Issue Details

Kingdom: Encapsulation
Scan Engine: SCA (Data Flow)

Source Details

System Information Leak: Internal — Low

Package: src.src

src/src/main.ts, line 12 (System Information Leak: Internal) — Low

Source: lambda(0)
From: lambda
File: src/src/main.ts:12

```
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule)
12   .catch(err => console.error(err));
13
14 undefined
15 undefined
```

Sink Details

Sink: ~JS_Generic.error()
Enclosing Method: lambda()
File: src/src/main.ts:12
Taint Flags: SYSTEMINFO

```
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule)
12   .catch(err => console.error(err));
13
14 undefined
15 undefined
```