Report for DDSS A2

Klemen Janez Poličar - 2022165285

# Introduction

I have written the application in PHP. I've installed three libraries that I use within the project using Composer as a package manager:

1. **illuminate/support** is used for helper functions, such as encoding special HTML characters in a string
2. **illuminate/hashing** is used for hashing passwords using the bcyrpt algorithm
3. **symfony/var-dumper** is used for cleaner debug functions

# Part 1.1, 1.2 and 1.3

A03:2021-Injection:

Every input in the form takes the value from the URL query string ($\_GET ). After submitting the form, all the parameters that were sent are stored in the URL query string of the response. This is good practice, as it provides a better UX, because the user need not fill in the form every time he makes a request. The difference between the vulnerable and correct form is that the correct form escapes the value from the query string, whereas the vulnerable does not, which introduces a XSS vulnerability.

Example: Visit URL:

http://localhost:8081/part1.php?username=poof"/><script>alert('injected')</script "

A03:2021-Injection:

The vulnerable form request handler does not sanitize the input parameters when building the SQL query, which introduces an SQL injection vulnerability. The correct form uses the built-in PHP function *pg\_query\_params* to safely build a query.

A05:2021-Security Misconfiguration:

Neither the form request handler nor the page that displays the UI returns a Content-Security-Policy or X-Frame-Options HTTP header, which could be used to prevent XSS or clickjacking attacks by restricting the valid sources for external resources.

A05:2021-Security Misconfiguration:

Neither the form request handler nor the page that displays the UI returns a Strict-Transport-Security HTTP header which would enforce the use of the HTTPS protocol (encrypted). Since this is not the case, there is the vulnerability of a MITM attack with an attacker eavesdropping on the network traffic.

A05:2021-Security Misconfiguration:

The composer.lock file has not been assigned the proper permissions on the OS and is thus accessible via the web server. This file gives very sensitive information, such as dependency package versions. So if a certain version of a package is know to be vulnerable, the attacker can find out if the server is using the vulnerable version.

Example: Visit URL:

http://localhost:8081/composer.lock

# Part 1.1

A03:2021-Injection:

The vulnerable form request handler returns an error message without having escaped the username that is interpolated in the message. This introduces an XSS vulnerability.

Example: Visit URL:

http://localhost:8081/part1\_vulnerable.php?v\_username=<strong style="font-size:50px">unregistered</strong>&v\_password=password&v\_remember=0

A04:2021-Insecure Design:

The vulnerable form error request handler (incorrect username or password) redirects the user back to the login page with both the username and password sent in the query string, to allow for quick fixes by the user. This exposes the password as cleartext and is considered bad practice. If the page is loaded as an iframe for instance, it could be possible to uncover the password. The correct form only provides the username in the query string, so the user must enter his password again before submitting.

Example: Visit URL:

<http://localhost:8081/part1_vulnerable.php?v_username=kpolicar&v_password=password&v_remember=0>

A04:2021-Insecure Design:

The vulnerable form error request handler gives too much information about the failed login attempt – it tells the user if their exists an account with the provided email. This is considered bad practise. The correct form only replies with a failed attempt, without giving detail about the existance of an account with the provided email address.

Example: Visit URL:

<http://localhost:8081/part1_vulnerable.php?v_username=unregistered&v_password=&v_remember=0>

A03:2021-Injection:

The vulnerable form request handler does not encode the special HTML characters text in the page where the username is displayed (when the form is submitted and the username does not exist, an error message is displayed which contains the submitted username), so there is a second order XSS vulnerability. The user's username may contain HTML tags, which is later interpolated as valid HTML on the webpage.

Good Practice:

Once authenticated, the user's authentication is stored in his session data – meaning on the server. This mitigates impersonation attacks.

# Part 1.2

A01:2021-Broken Access Control:

The vulnerable form request handler checks authentication using post parameters rather than through the session. The post parameters can be manipulated by the end-user and since the authentication method of the app is simply to authenticate the user with the username that is provided, he can simply provide any registered user's email to achieve impersonation. In short, he is able to create comments in the database authored by any registered user. This differs in the correct form request handler, which uses the session to verify user authentication.

Example: Visit URL:

[http://localhost:8081/part2\_vulnerable.php?v\_text=injected<script>alert("injected")</script>&auth=jpolicar](http://localhost:8081/part2_vulnerable.php?v_text=injected%3cscript%3ealert(%22injected%22)%3c/script%3e&auth=jpolicar)

A03:2021-Injection:

The vulnerable form request handler does not sanitize the inputed text, nor does it encode the special HTML characters text in the page where the text is displayed, so there is a second order XSS vulnerability. The user may send HTML tags as content of the comment's text, which is later interpolated as valid HTML on the webpage.

# Part 1.3

A04:2021-Insecure Design:

The vulnerable form request handler takes the database connection credentials from the request parameters (GET/POST). This is insecure, since it means the client must provide the database credentials, which is information he should not have.

# Part 2

I have selected OWASP ZAP as my active security scanner. I have chosen this tool because it is often remarked as the flagship web application security scanners. It's open source, so it's free to use. It has a lot of documentation and quick-start videos. It didn't take me very long to learn how to perform a scan. This is the summary of the active scan:

Graphical user interface, text, application

Description automatically generated

It has detected quite a few vulnerabilities, some of which I have already mentioned myself in the introductory chapters, such as XSS and SQL injection.

It has also detected that CSRF tokens are not present in the forms, which introduces the possibility of attack. An attacker might send an authenticated user a link which would create a comment (part1.2) in his name.

It has also detected application error disclosures in the vulnerable forms. This is a sign that there is no proper exception handling and warning message supression. This is not the case in the correct form handler, where I have supressed such messages using the PHP function *error\_reporting(0)*. I have also added a global exception handler to prevent exception messages from leaking into the output of a bad request. Such incidences disclose information about the web application.

It is also warning of some issues with cookies and server information leaks and missing security headers.

I should mention that many of the reported issues were detected merely because I mostly followed general practices when developing the web application. For instance, it was able to detect that sensitive information is leaked because the password field was named »password«. If it was named something else that didn't match the pattern it was searching for, the tool would not classify it as »sensitive information«. I could also have implemented CSRF token protection using an input field which would not be named by the pattern the tool is searching for, in which case it would report the false-positive vulnerability.

I have selected SonarQube as my static code analysis tool. I found the tool on this list of static analysis tools for PHP: <https://github.com/exakat/php-static-analysis-tools>

I have selected it because it seems like a modern solution with a clean UI, so it's easy to quick-start. It also has a plugin for PHPStorm, my IDE of choice. It works for PHP, which is the language I wrote the web app in.

This is the summary of the analysis:

Graphical user interface, application, Teams

Description automatically generated

After reviewing the security hotspots and vulnerabilities detected, there have been no false positives. All are true positives. The following images show the issues.

Graphical user interface, text, application, email

Description automatically generated

Slika 1: It has detected the SQL injection vulnerabilities:

Graphical user interface, text, application

Description automatically generated

Slika 2: Precise location of SQL injection hotspot

It has also detected a security hotspot. The password for the DB connection is hardcoded, which means it will be present in the version control history. This is considered bad practice and can lead to credential leaks. Such information should be stored in an .env file which is ignored by version control systems.

A screenshot of a computer

Description automatically generated

Slika 3: DB credentials are hardcoded

Neither of the tools were able to detect the first vulnerability mentioned in Part 1.2: A01:2021-Broken Access Control. See that part for a ready to use exploit.

# Comparison

In my case, I have found OWASP ZAP, the active security scanner tool to be much more useful in detecting vulnerabilities. It has detected many more issues, even more low risk ones, such as security headers and information disclosure. SonarQube didn’t report many security issues at all in comparison. It didn’t find any XSS vulnerabilities, even though there are many such vulnerabilities in the web app. It’s strength is that it can easily be implemented with CI, whereas OWASP ZAP was very slow to produce results – even for such a small web app. I presume SonarQube was not able to detect many vulnerabilities simply because it does not know the entry point of the script, or rather the lifecycle of the web app. Which script can be called as an entry point (accessible via a web browser)? If it doesn’t know this, it cannot know which headers are sent and similar. This is where OWASP ZAP shines, because it uses a spider to locate all accessible web pages, so it’s able to analyze the entire result rather than what the static analysis tool can only presume is a fraction of the web request.