Web Application Security Evaluation

A web application security evaluation of the Astley Skateboards web store

Darren Sherratt

CMP319: Ethical Hacking 2

BSc Ethical Hacking Year 3

2017/2018

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

Contents

Table of Figures

1. Introduction

This document is the second part of a two-part web application penetration test. Part one documented the penetration test itself along with a description of the vulnerabilities discovered and the risk they could pose to the company or the users of the website. This report will explain the vulnerabilities themselves, the potential implications, and how they can be mitigated by the web development team.

These mitigations should be immediately implemented prior to deployment of the web application.

2. Vulnerabilities and Countermeasures

2.1. INFORMATION DISCLOSURE

a) Robots.TXT

i. Vulnerability

The robots.txt file is used to prevent web spiders and search engines from cataloguing the items located inside. In this instance, the page info.php was included in the robots.txt file.

Contrary to the intention, using robots.txt incorrectly serves to point malicious attackers to pages that the web developers would rather remain undetected. The info.php page was publicly available enabling any user to access this page, which contained information which could give insight on the technologies used on the web server. This information could then be used to further refine attacks.

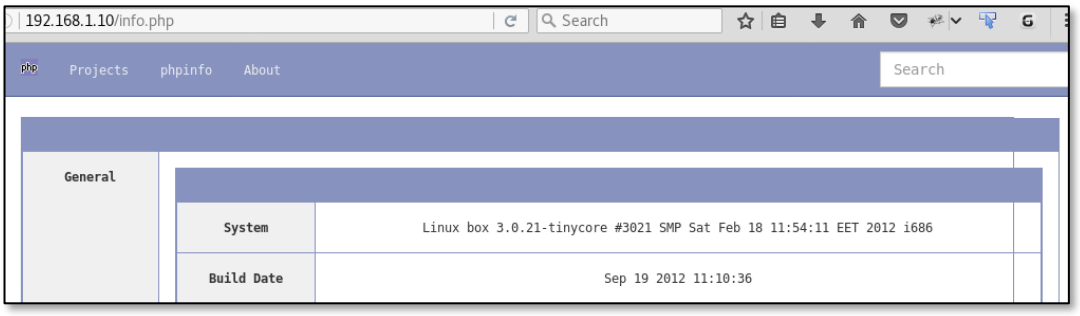

FIGURE 2.1 - INFO.PHP

ii. Mitigations

Documents containing sensitive data should not be placed in the robots.txt files – the robots.txt page is publicly available. If the info.php page is not required, it should be removed from the web server.

Rather than using robots.txt to blacklist pages, a more secure method would be to create a whitelist of pages or folders which search engines and spiders would be allowed to view – these pages should be public facing and not contain any sensitive information.-.

b) HIDDEN SOURCE CODE

i. Vulnerability

Contained within the customers/index.php page was a comment suggesting the existence of the info.php page. Comments of this nature can lead attackers to pre-existing pages which could increase the likelihood of a breach.

The same can be said for the page hidden.php. This page is blank, but the source code contains a door entry code. If this information was found by an attacker the physical security of the business could be put in jeopardy.

ii. Mitigations

Web pages should not contain superfluous information. Prior to launch, each page should be read through to identify whether comments in the html are needed for the live version of the website. Any information which is not necessary should be stripped from the source files.

c) REVERSIBLE COOKIES

i. Vulnerability

The SecretCookie cookie assigned to a user on login was easily decrypted and contains the username, password, and time of login of the user. If acquired by a third party, this information could be used by an attacker to grant access to a user's account.

In this case rot13 was used to encode the user data. Rot13 is a simple cipher which is commonly used to encrypt cookie data. This method of encryption on its own is not enough to prevent a user or attacker from decrypting the cookie to display the information within.

```php
<?php
$str=$username.':'.$password.':'.strtotime("now");
$str = str_rot13(base64_encode($str));
setcookie("SecretCookie", $str);
?>
```

FIGURE 2.2 - SECRETCOOKIE CREATION

ii. Mitigations

If this information is required to be stored in a cookie, the information should be stored securely with a strong algorithm. However, with a proper implementation of sessions, cookies should not be required to store user passwords.

The data stored within a cookie should be considered public as with enough time, any algorithm can be broken, and cookies are stored in view of the user. Important information should not be stored

client-side, but on server-side to reduce the impact should an attacker gain access to another user's client-side information.

d) COOKIE ATTRIBUTES

i. Vulnerability

Cookies used in the site are not set with the HTTPonly flag set. This flag tells the browser that the cookie should only be accessed by the server and any attempt to access the cookies from the client side is forbidden. (Atwood, 2008) .

If an attacker was able to obtain a user's cookie through stored cross-site scripting they would be granted access to the user's account and the functionality therein. Shown in Figure 2.3 is a simple example of accessing the user's cookie using stored cross-site scripting.

FIGURE 2.3 - XSS COOKIE ALERT

ii. Mitigations

Set the HTTPonly flag when assigning cookies. This will prevent cookies from being accessed via client-side methods such as JavaScript, which will prevent attackers from using stored cross-site scripting.

e) DIRECTORY BROWSING

i. Vulnerability

Currently, it is possible for a non-authorised user to access directories on the web server which should not be accessible publicly. Depending on what information is located on the server outwith the user accessible areas, pages accessed could reveal sensitive data including server information, usernames, or passwords.

ii. Mitigations

Using sessions, users at all levels can be prevented from accessing webpages other than the pages approved by the web developers. By using PHP and Apache .htaccess files, users can be restricted to viewing preapproved content. When a restricted page is requested, Apache redirects to a PHP file which authenticates the use and determines whether the user should be permitted access to the content; if the user is successfully authenticated the content is delivered otherwise the user is redirected (Smistad, 2009). The authentication level of the user should be tied to the PHPSESSID and stored on the server while the session ID is active.

f) PHP INFORMATION DISCLOSURE

i. Vulnerability

The file phpinfo.php is in the root directory – this means all users have access. This file contains information relating to the technologies used on the server and can be used to narrow down potential attack vectors and strategies by providing file paths and service versions.

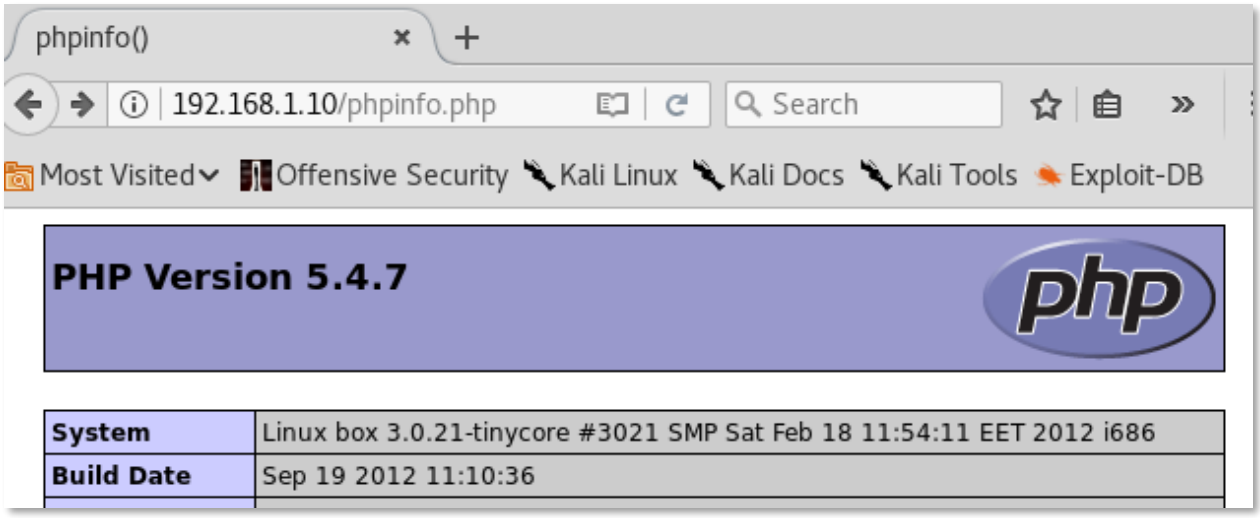

FIGURE 2.4 - PHPINFO.PHP

ii. Mitigations

This page should be removed from the root directory. If there is a requirement for the file to remain on the web server, it should be relocated to a directory inaccessible to unauthorised users.

g) HIDDEN GUESSABLE FOLDERS

i. Vulnerability

The web server hosts files containing important information within folders which can be accessed by unauthorised users. These directories have names which can be brute forced by programs such as Dirbuster, or guessed. In this instance files found within guessable folders contained passwords for the database.

ii. Mitigations

If these files are not in used on the web server, they should be removed – otherwise, the files should be moved to directories accessible only to users with advanced privileges.

Alternatively, any information relating to the technologies on the server or application logic should be omitted from these files.

2.2. AUTHORISATION

a) WEAK PASSWORD REQUIREMENTS

i. Vulnerability

The requirements for a user password are very weak – requiring only a single letter to be valid as a password. Additionally, upper and lowercase letters are interchangeable, vastly reducing the password strength. When guessing or bruteforcing passwords, weak password requirements will reduce the amount of time and effort it will take to guess user passwords.

ii. Mitigations

User passwords should consist of a minimum length of eight letters containing a combination of uppercase letters, lowercase letters, numbers, and special characters. This will greatly increase the complexity of all user passwords, resulting in passwords which will hold up against brute force attacks.

b) NO HTTPS

i. Vulnerability

The web application currently uses HTTP rather than the more secure HTTPS. This results in all information sent to and from the server to be unencrypted; if this information was intercepted then all parameters would be presented in plain text. This could result in a user's credentials being stolen during transit.

ii. Mitigations

HTTPS will encrypt all traffic to and from the web server, preventing information from being intercepted and preventing values from being edited by malicious users.

Incorporate HTTPS site wide using OpenSSL on the server. It is imperative that the private key is securely stored out with the reach of the public.

c) USER ENUMERATION

i. Vulnerability

The error message for an attempted log in with an incorrect username alerts the user to the username not existing. This can be used to enumerate users by sending a large number of requests and analysing the responses.

Shown below, the response on the left is seen when the username is valid, and the response on the right is seen when the username is invalid. As shown, these responses are different enough to make a distinction between a valid and invalid username.

```
<script>alert('Email or password is incorrect!')</script><script>window.open('index.php','_self')</script>
```

```
<script language="javascript">alert("Username not found");window.history.back();</script>
```

FIGURE 2.5 - VALID & INVALID USERNAME RESPONSE

ii. Mitigations

The error message for a wrong password or wrong username should be the same to prevent an attacker from being told which parameter was incorrect.

d) UNLIMITED LOGIN ATTEMPTS

i. Vulnerability

A user is unlimited in the number of attempts they take to log in. This results in the capability for an attacker to brute force a log in by sending a large number of log in requests with varying passwords, continually trying until a successful password is found.

ii. Mitigations

A captcha system or similar should be implemented to require user input after a set number of failed log in attempts to prevent automated brute forcing.

e) BRUTE-FORCEABLE ADMIN PASSWORD

i. Vulnerability

Similar to the user log in, the admin log in does not have a limit on the number of requests than can be made; this allows for a brute force attack using a number of usernames and passwords. Given enough time, this attack will reveal the admin credentials.

ii. Mitigations

Implement a Captcha system to prevent automated attacks. The system will require user input after a number of failed log in attempts; this value will be chosen by the web application administrators but should be in the range of three to seven.

f) SESSION USAGE

i. Vulnerability

The PHPSESSID is not correctly implemented in the application. When a user logs out, another user logging in on the same device will be given the same PHPSESSID. As more than one session per user can be open at any one time, this could allow an attacker to gain access to a user's account by using the same session ID in a public setting when a new user has logged in to the web application.

Sessions are also not required for requests. The password change, add to cart, and checkout functions can all be executed with only a user ID.

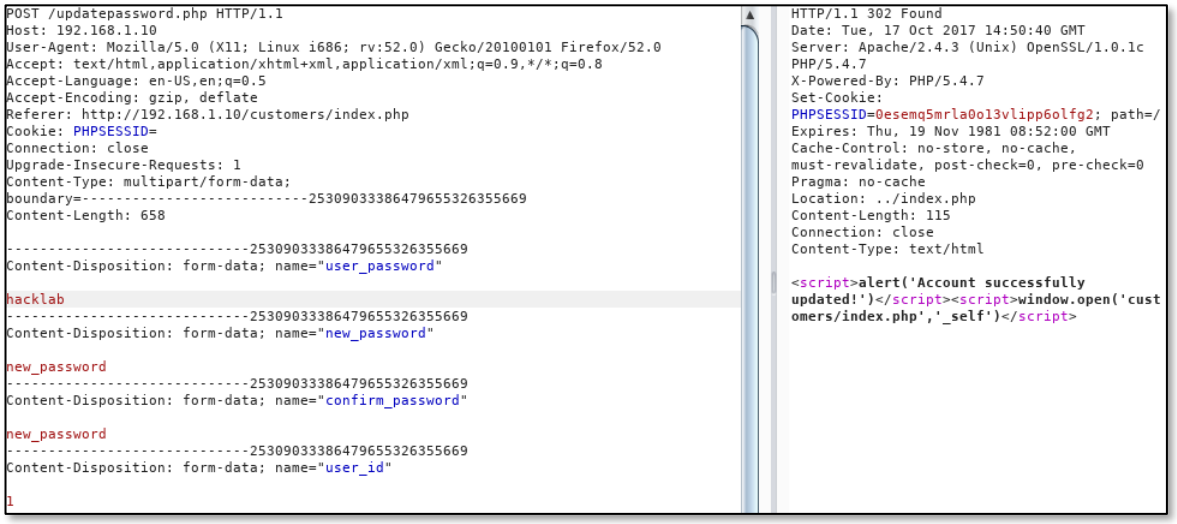

FIGURE 2.6 - PASSWORD CHANGE W/ NO CREDENTIALS

As shown in Figure 2.6, the password for the user with the ID of 1 was changed without the presence of a PHPSESSID or SecretCookie. By sending the POST request with the relevant criteria, any user can be targeted; using a program like the Burp Repeater, a large number of user IDs can be run through, changing all user's passwords. As the web application does not have a recovery feature this would grant full access to all user account to the attacker.

Similarly, an attacker can add any quantity of any valid item at any price to any cart using the save_order.php function, then check out with cart_items.php?update_id=x – where x is the user ID. This functionality could disrupt the operation of the service and cause financial and logistical disrupt for users.

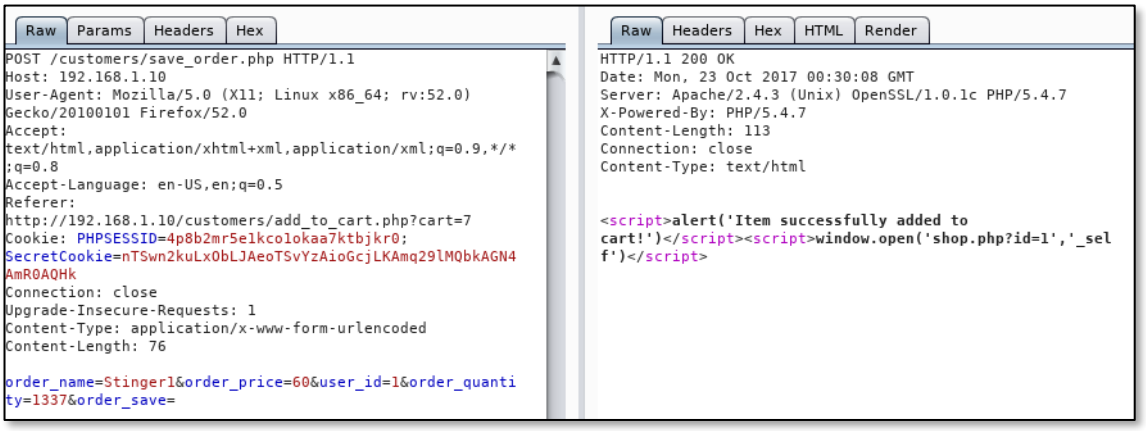

FIGURE 2.7 - ADD ITEMS TO CART

ii. Mitigations

Requests should not be accepted unless they originate from an account with a legitimate PHPSESSID. These IDs are randomly generated and are complex enough to not be guessable. When a request is made, both the PHPSESSID and the SecretCookie should be checked to determine if the request is authentic – otherwise the request is dropped.

2.3. COMMAND INJECTION

a) LOCAL PAGE INCLUSION

i. Vulnerability

The file extras.php presents a vulnerability on the web server. When this page is loaded a file is used as a parameter as extras.php?type=xxx.php. The included file can be replaced with the location of another file on the server and the contents of the file will be displayed on the loaded page. This can be used for example to navigate to extras.php?type=../../../../etc/passwd and the password hashed of the web server will be displayed.

ii. Mitigations

Remove the additional functionality which this web page requires and display the page in the same manner as all other pages on the web application. The reasoning behind this page being displayed in this way is unclear but, if it is a requirement, functionality should be put in place to prevent other pages and directories from being accessed.

b) SQL INJECTION VULNERABILITY

i. Vulnerability

The login function used for the web application is vulnerable to SQL injection. With the correct commands, it is possible to access all usernames and passwords for all users – including the administrators. This form if vulnerable due to the syntax of the SQL query used to log users in.

The query used to log users in is shown below in Figure 2.8; by supplying the password 'OR'1'='1, the password field is completed and submitted empty, and the application logs the user in because 1=1 equates to true.

```
$sql=mysql_query("select * from users WHERE user_email=('$username') AND user_password='$password'");
$rows=mysql_fetch_array($sql);
```

FIGURE 2.8 - USERLOGIN.PHP LOG IN QUERY

ii. Mitigations

Use prepared statements to negate this possibility. Prepared statements take the whole query as a parameter which prevents an attacker from forming a query which will allow for arbitrary requests. An example of a prepared statement user login is shown below in Figure 2.9. In this example only strings are allowed to be taken as parameters. The string will consist of the complete field value and as there is no query to break out of, SQL injection cannot occur.

```
$stmt = $con->prepare("SELECT user_id, username, password, status FROM users WHERE username=? AND password=? LIMIT 1");
$stmt->bind_param('ss', $username, $password);
```

FIGURE 2.9 - PREPARED STATEMENTS LOG IN

c) CROSS-SITE SCRIPTING

i. Vulnerability

Certain pages on the website are vulnerable to XSS. The only way to add text to the document is through the admin portal with modifying, or adding items. By including JavaScript, code can be executed client-side.

The admin portal will execute this code as soon as the malicious item is viewed but the customer side website will display the item with the script tags visible. This code will then not execute until the item has been added to the cart and the checkout portion of the site has been accessed.

With the functionality to modify the contents of other user's carts however, malicious items can be added to the user's cart. This way, an attacker could inject JavaScript code into the name while adding an item to the cart.

ii. Mitigations

Input sanitisation for the admin portal and POST/GET requests should be used to strip out malicious code prior to the page being received on the client side.

Furthermore, the HTTPOnly flag should be set as discussed in the section above, and the X-XSS-Protection header should be set to '**1; mode=block'**. This header will be discussed later in the report.

2.4. FILE INJECTION

a) PAGE UPLOAD VULNERABILITY

i. Vulnerability

By changing the parameters of the uploaded file, a user can upload any file to the server. This permits the upload of exploits or trojans which can be used to gain a foothold on the server or obtain access to the server systems.

ii. Mitigations

The uploaded file should be checked server-side as opposed to client side. The extension of the uploaded file should be examined to restrict any other files other than those set in a whitelist of approved file types. Input validation should be used to check for evasion techniques such as appending a second file type, using spaces or dots, or using null characters. The uploaded file should have a maximum file name, and maximum file size.

Files should be uploaded to a directory outside of the website root to minimise the damage inflicted should a malicious file be uploaded. If possible, uploaded files should be scanned with an

(Shapland, 2012)

2.5. INSECURE STORAGE

a) PASSWORD STORAGE

i. Vulnerability

The passwords are stored in plaintext in the database. If the database is breached all usernames and passwords will be accessible to the attacker – granting access to all accounts.

user_id	thumbnail	user_email	user_address	user_lastname	user_password	user_firstname
1	rick.jpg	hacklab@hacklab.com	1 Bell Street	Bloggs	hacklab	Joe
3	<blank>	StevePlumber@hacklab.com	2 Brown Street	Plumber	gebbz03	Steve
4	<blank>	RedAdiaire@hacklab.com	3 Red Street	Adaire	mik	Red
5	<blank>	user@user.com	user	use	new	user

FIGURE 2.10 - PLAINTEXT USER PASSWORDS

ii. Mitigations

Passwords should be hashed before saving the values in the database, and hashed upon entry to compare with the database stored value. A strong algorithm should be used to prevent the algorithm from being easily cracked. It is recommended that bcrypt be used; this algorithm is suitably strong and is deliberately slow so if the hashes are acquired by an attacker, they will take a long time to crack.

b) ADMIN CREDENTIALS

i. Vulnerability

The admin credentials, like the user credentials, are stored in plaintext. These credentials are stored in the same database – meaning if an attacker is able to access the database looking for user credentials, they will also have access to the table containing the admin credentials.

admin_id	admin_password	admin_username
1	comrades	admin

FIGURE 2.11 – PLAINTEXT ADMIN CREDENTIALS

ii. Mitigations

The admin passwords, like the user passwords, should be hashed with bcrypt to prevent access to the database revealing the plaintext passwords.

2.6. CLIENT-SIDE ATTACK

a) CROSS SITE REQUEST FORGERY (CSRF)

i. Vulnerability

Cross site request forgery is an attack that forces a user to execute unwanted actions on a web application in which they're currently authenticated. In this web application, a user is vulnerable to a CSRF attack which would cause their password to change to one set by an attacker. By clicking a specially crafted link, the password change request is sent to the web server and, as the user is already authenticated due to being currently logged in, the request will be honoured – changing the user's password to the password. (OWASP, 2017)

This will result in the user being locked out of their account – with full access granted to the attacker.

ii. Mitigations

User functions which will send requests to the web server should use header checking to determine if the request originated from the webpage or outside the website. If the referrer is not valid, the request did not come from within the web application and should not be honoured.

Alternatively, the inclusion of CSRF tokens for form submissions would negate the threat of CSRF. These tokens consist of a very large random number and will be generated by the server and sent to the user when the user signs in. Whenever a form is submitted the token is sent along with the form and verified by the server. If the token is not valid, the request does not execute. This token should be protected from access by JavaScript and other methods to prevent an attacker obtaining the value. With a proper implementation these tokens will be large and random enough to be impossible to guess. If a user clicks on a malicious CSRF link, the request will be sent without the token and the user will be protected. (Prechelt, 2015)

2.7. OTHER ISSUES

a) X-POWERED-BY HEADER REPORTS PHP/5.4.7

i. Vulnerability

The header reveals superfluous information by displaying the language used on the web server with the current version in use. This allows attackers to target attacks which could include zero-day vulnerabilities.

ii. Mitigations

Remove the header from the website. The x-powered-by header does not offer any additional functionality so disabling it will not restrict use of the web application. (dimokaragiannis, 2015)

b) ANTI-CLICKJACKING X-FRAME-OPTIONS HEADER IS NOT PRESENT

i. Vulnerability

Clickjacking consists of an attacker adding opaque layers on top of the rendered website to trick a user into clicking on buttons or performing actions when they are trying to perform actions on the website. (OWASP, 2017).

An attacker could host a website and overlay an invisible page from the web application. This could be used to then force the user to unwillingly perform actions such as purchasing items.

ii. Mitigations

Setting the X-Frame-Options header to SAMEORIGIN restricts the browser to allow framing only from the same domain – preventing attackers from loading their frames. If the web application does not intend on using iframes, the header should be set to DENY, restricting the browser from loading any frames. (Law, 2010). This will prevent the website from being rendered in any iframes, negating clickjacking.

c) X-XSS-PROTECTION HEADER IS NOT DEFINED

i. Vulnerability

The XSS filter/XSS auditor is a feature of Internet Explorer, Chrome, and Safari which can prevent pages from loading when they detect reflected cross-site scripting attacks. This feature is controlled by the X-XSS-Protection header. Currently, this header is not set.

The header can be set to one of three settings:

0 - Disables XSS filtering.

1 - Enables XSS filtering (usually default in browsers). If a cross-site scripting attack is detected, the browser will sanitize the page (remove the unsafe parts).

1; mode=block - Enables XSS filtering. Rather than sanitizing the page, the browser will prevent rendering of the page if an attack is detected. (ccsplit, 2017)

Currently, the header is not set; this informs the XSS auditor to default setting 1 – which will remove unsafe parts of a page. However, this can further extend the attack surface.

By abusing false-positives, attackers can disable innocent scripts within the page. The auditor cannot distinguish whether a script is injected by attackers or intended for the page. This means potential security libraries can be disabled by faking an injection and having the library removed.

Additionally, as with any filter, it is inevitable that it will be bypassed one day; even at the time of writing it is stated that it is not guaranteed to catch all variants of XSS. (@filedescriptor, 2016)

ii. Mitigations

Enabling the header will enable control over the XSS auditor which will increase the security of the site. The header should in this case should be set to '**1; mode=block**'. This will stop the page from loading if XSS is detected by the auditor.

This setting keeps users safe from XSS while not allowing attackers to use the auditor to further exploit the web application.

d) X-CONTENT-TYPE-OPTIONS HEADER IS NOT SET

i. Vulnerability

Files served to the host from the server are handled differently depending on the file type. When internet explorer downloads a file, it uses a process known as MIME sniffing to read the first 256 bytes of a file to compare against a database to determine the file type.

This allows for an attacker to upload a file containing HTML and JavaScript code which, when uploaded in a format which leaves the file type ambiguous to the web server, enables the code to be run. However, when Internet Explorer is used to access that file, it will use MIME to identify the file type, and will then execute the code.

This can be used to store a persistent XSS vulnerability. (Bijl, 2012)

ii. Mitigations

Setting the X-Content-Type-Options header to 'nosniff' disables MIME sniffing and forces the browser to use the file type sent by the server. This will prevent any arbitrary code from being run by an uploaded file.

e) GET APACHE MOD_NEGOTIATION HEADER IS ENABLED WITH MULTIVIEWS

i. Vulnerability

Mod negotiation is an apache module which allows for content negotiation. Content negotiation is the selection of the document which best matches the client's capabilities from one of several available documents.

The MultiViews option, which is enabled on the web application, will conduct an implicit filename pattern match and will choose from the results. If the server received a request for /some/dir/foo and the directory /some/dir/foo does not exist, the server will read the directory looking for all files called foo.*. It will then return the best match to the client's requirements. (Auburn University, n.d.)

This aids an attacker in bruteforcing directory names by returning files with the same name if the directory doesn't exist.

The response will return with all variations of the file name in a response, making it easier for an attacker to map out the application.

ii. Mitigations

Disable MultiViews by creating a .htaccess file with the line 'Options -Multiviews'. (Acunetix.com, n.d.)

f) SHELLSHOCK

i. Vulnerability

Shellshock is an exploit which targets a vulnerable bash shell which exists on the Linux web server. A CGI script is targeted with a specially crafted packet which, when the CGI script makes a call to a script, runs the payload.

This can be used to open a connection to an attacker machine and grant access to the attacker. (Hunt, 2014)

ii. Mitigations

Update bash shell to a version which contains mitigations to Shellshock. Otherwise, remove all CGI scripts present on the web server.

g) TRACE HTTP METHOD IS ACTIVE

i. Vulnerability

Cross-Site Tracing (XST) involves the use of XSS and the TRACE HTTP method. TRACE allows the user to see what is being received at the other end of the request chain and use that data for testing or diagnostic information. XST can be used to get round the HTTPOnly cookie flag to steal user's cookies by using the TRACE method to read cookies that are otherwise blocked from JavaScript access.

XST can be used to expose sensitive header data which can be used in further attacks. (Schema, 2010)

ii. Mitigations

Disable the TRACE method by editing the httpd.conf file on the web server adding the following lines:

```
LoadModule  rewrite_module  "/usr/local/apache/modules/mod_rewrite.so"
RewriteEngine On
RewriteCond %{REQUEST_METHOD} ^(TRACE|TRACK)
RewriteRule .* - [F]
```

FIGURE 2.12 - HTTPD.CONF TRACE DISABLE

h) /PHPMYADMIN IS VISIBLE TO THE OUTSIDE WORLD

i. Vulnerability

The portal for the management system Phpmyadmin is accessible from the outside world. If an attacker could guess the correct credentials they would have access to the database which contains the unencrypted credentials for all users.

ii. Mitigations

Restrict access to the database to a logged in admin user and use a strong username and password to secure the database. These credentials should be unique to the database and be long and complex enough to prevent brute force attacks.

3. References

@filedescriptor, 2016. *The misunderstood X-XSS-Protection.* [Online]
Available at: https://blog.innerht.ml/the-misunderstood-x-xss-protection/

Acunetix.com, n.d. *Apache mod_negotiation filename bruteforcing.* [Online]
Available at: https://www.acunetix.com/vulnerabilities/web/apache-mod_negotiation-filename-bruteforcing

Atwood, J., 2008. *Protecting Your Cookies: HttpOnly.* [Online]
Available at: https://blog.codinghorror.com/protecting-your-cookies-httponly/

Auburn University, n.d. *Apache module mod_negotiation.* [Online]
Available at: http://www.auburn.edu/docs/apache/mod/mod_negotiation.html

Bijl, J., 2012. *MIME Sniffing: feature or vulnerability.* [Online]
Available at: https://blog.fox-it.com/2012/05/08/mime-sniffing-feature-or-vulnerability/

ccsplit, 2017. *X-XSS-Protection.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection

dimokaragiannis, 2015. *Unset/Remove apache's X-Powered-By header in ubuntu.* [Online]
Available at: https://dimokaragiannis.wordpress.com/2015/12/07/unsetremove-apaches-x-powered-by-header-in-ubuntu/

Hunt, T., 2014. *Everything you need to know about the Shellshock Bash bug.* [Online]
Available at: https://www.troyhunt.com/everything-you-need-to-know-about2/

Law, E., 2010. *Combating ClickJacking With X-Frame-Options.* [Online]
Available at: https://blogs.msdn.microsoft.com/ieinternals/2010/03/30/combating-clickjacking-with-x-frame-options/

OWASP, 2017. *Clickjacking.* [Online]
Available at: https://www.owasp.org/index.php/Clickjacking

OWASP, 2017. *Cross-Site Request Forgery (CSRF).* [Online]
Available at: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

Prechelt, L., 2015. *What is a CSRF token ? What is its importance and how does it work?.* [Online]
Available at: https://stackoverflow.com/questions/5207160/what-is-a-csrf-token-what-is-its-importance-and-how-does-it-work

Schema, M., 2010. *Cross-Site Tracing (XST): The misunderstood vulnerability.* [Online]
Available at: https://deadliestwebattacks.com/2010/05/18/cross-site-tracing-xst-the-misunderstood-vulnerability/

Shapland, R., 2012. *File upload security best practices: Block a malicious file upload.* [Online]
Available at: http://www.computerweekly.com/answer/File-upload-security-best-practices-Block-a-malicious-file-upload

Smistad, E., 2009. *How to restrict user access to content in folders using PHP and Apache .htaccess files.* [Online]
Available at: https://www.eriksmistad.no/how-to-restrict-user-access-to-content-in-folders-using-php-and-apache-htaccess-files/