replace title

replace author

replace date

Contents

List of Figures

Chapter 1

Introduction

Chapter 2

Executive Summary

2.1 Planning

2.1.1 Approach

2.1.2 Scope

2.1.3 Objectives

2.2 Methodology

2.2.1 Information Gathering

2.2.2 System Attacks

2.3 Summary of Findings

Chapter 3

Key Findings

3.1 Auction Site

This section covers the vulnerabilities found with the user-facing auction site.

3.1.1 Path Traversal

Security Implications / Risk Level

Path traversal allows for arbitrary file read across the system, for any files readable by the apache user (www-data). This is dangerous as it could potentially leak sensitive company data, as well as user data. If combined with other vulnerabilities, such as incorrect permissions on log files, it is possible to achieve Remote Code Execution through malicious log read/write.
Overall, the execution of this exploit is trivial, and the repercussions are potentially serious but not disasterous. Due to this, the risk level of this exploit is evaluated to be **medium**.

Cause of Vulnerability

The vulnerability is caused by the method used to retrieve and display image files on the website. Instead of directly referencing the image file through the 'src' field on an 'img' HTML tag, a PHP script is instead used to include the file.
While using PHP include scripts may not normally be dangerous, the file name to be retrieved can be edited by the user, allowing them to easily select which file should be displayed. A lack of filter/extension whitelist makes this even more potent.

Steps to Replicate

- Firstly the inspect element tool in Mozilla Firefox was used to inspect an image, revealing the image URL (Fig. 3.1).

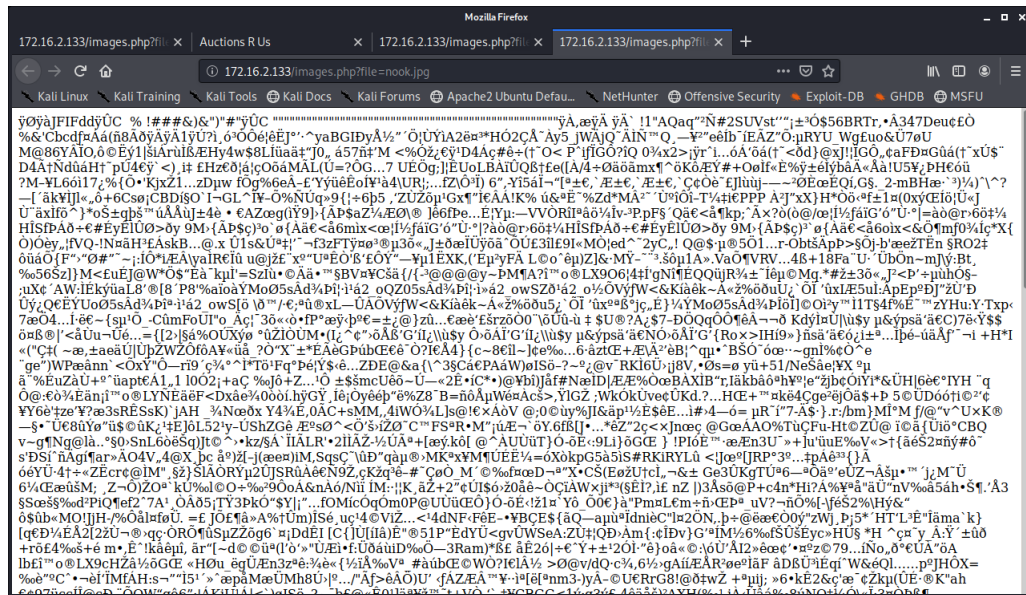- The image URL could then be opened, showing the ASCII representation of the binary content for the image file (Fig. 3.2).

- Finally, the URL parameter 'file' could be replaced with a file path, allowing for arbitrary file read. In this example, the ../ operator was used to go up directories until root, and then the /etc/passwd file was navigated to (Fig. 3.3).

Figure 3.1: Inspecting the image

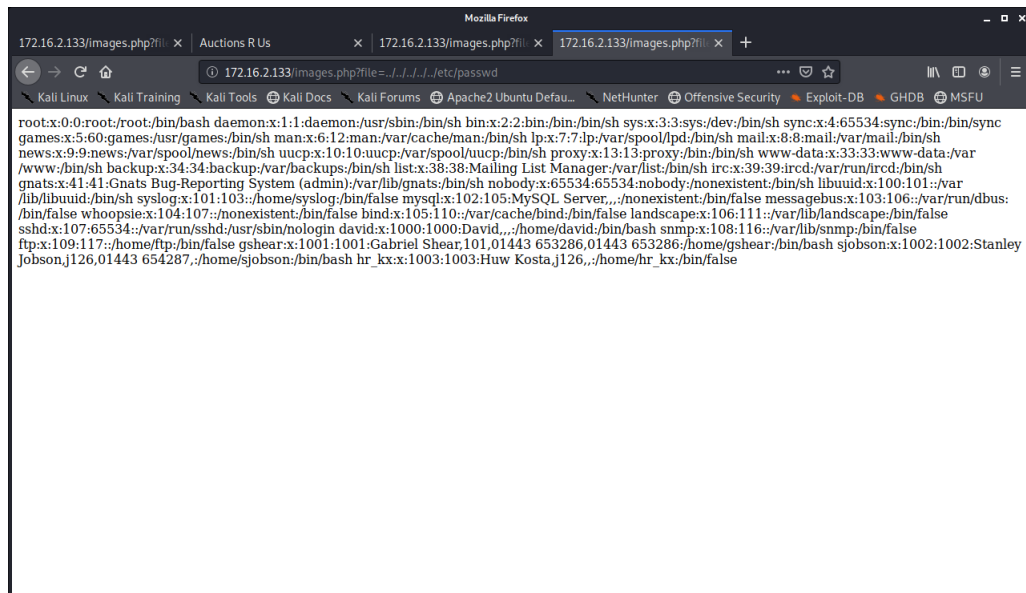Figure 3.2: Viewing the image contents

Figure 3.3: Exploiting the file read

3.1.2 SQL Injection

Security Implications / Risk Level

SQL injection is a blanket term covering any kind of unintended user control over the SQL queries interacting with a database. This can manifest in many

forms, such as:

- Authentication bypass, where SQL queries can be modified to bypass authentication checks such as login forms.

- Union injection, where the UNION keyword in SQL can be used to access data from other columns, tables, and databases.

- Error based injection, where SQL errors are intentionally created in order to gain information about the database.

- Blind injection, where queries that return TRUE or FALSE can be used to gain information about the database.

From the testing done, the website appears to be vulnerable to both authentication bypass, allowing attackers to log in to accounts, and blind injection, allowing for full read access across the database.

The execution of these exploits are relatively easy with the use of tools like SQLmap, and the repercussions can be very serious, allowing attackers to log in to administrator accounts as well as reading any user/auction data from the database. Due to this, the risk level of this exploit is evaluated to be **high**.

Cause of Vulnerability

SQL injection vulnerabilities are a result of allowing unsanitised user input in to SQL queries. Sanitising SQL queries involves removing any kind of dangerous character from the input, such as quotation marks (single and double) and comment tags. If this is not done, attackers are able to modify queries in specific ways to allow them to perform SQL injection.

One example of this would be performing an authentication bypass injection. A normal query may use a query like:

```
SELECT * FROM users WHERE username = '$inputname' AND password =
'$inputpass';
```

If an attacker enters something like ' OR 1=1# in to the username field, the query becomes:

```
SELECT * FROM users WHERE username = '' OR 1=1#;
```

Which will pick the first username from the table and sign the attacker in.

Steps to Replicate

- For initial testing of SQL injection, a basic authentication bypass was used. The payload ' OR 1=1– - was entered in to the username field (Fig. 3.4), which subsequently allowed access to the 'David' account, giving use of the admin panel as well (Fig. 3.5).

8

- For further testing, the tool SQLmap was used. This is a tool that automatically iterates through potential SQLi payloads, providing information such as DB names, table names, table data, SQL user names, and SQL version.

- The first test done with SQLmap was checking if it detected SQL injection. The command used for this was

```
sqlmap -u "http://172.16.2.133/login.php?username=qwe&
password=qwe&Search=" --batch
```

  The results provided some information on the DB system and potential attacks it was vulnerable to (Fig. 3.6).

- After this, the –passwords flag was used to test for DB users and retrieve any passwords if possible. SQLmap found a DB user named "auctuser", but had no access to the users table so could not retrieve a password (Fig. 3.7).

- With no password found, the next step was to search for databases. The command used to do this was

```
sqlmap -u "http://172.16.2.133/login.php?username=qwe&
password=qwe&Search=" --batch --databases
```

  This successfully found the databases, returning three in total (Fig. 3.8). The first was the information_schema DB, default in all installations of MySQL. The second was the auctionsrus DB, the one likely holding all info. The last was a test DB which also comes default with MySQL.

- With the new information of the auctionsrs db, the –tables flag could be used to dump the table names for said DB. The command used for this was:

```
sqlmap -u "http://172.16.2.133/login.php?username=qwe&
password=qwe&Search=" --batch -D auctionsrus --tables
```

  This returned two tables found in the auctionsrus DB (Fig. 3.9). One, auction_users, was likely to contain the user data for the site, while the other, auctions, was likely to contain the auction data.

- Finally, the last step was to attempt to dump the data from the tables. As a proof of concept, the data from the users table was dumped using the command:

```
sqlmap -u "http://172.16.2.133/login.php?username=qwe&
password=qwe&Search=" --batch -D auctionsrus -T auction_users --dump
```

This returned the full set of user data from the auction_users table, including usernames, MD5 hashed passwords (all cracked, with the one not displayed on the screenshot being '7331'), user IDs, and user levels (Fig. 3.10). With the dump done, the full extent of the SQL injection was explored.
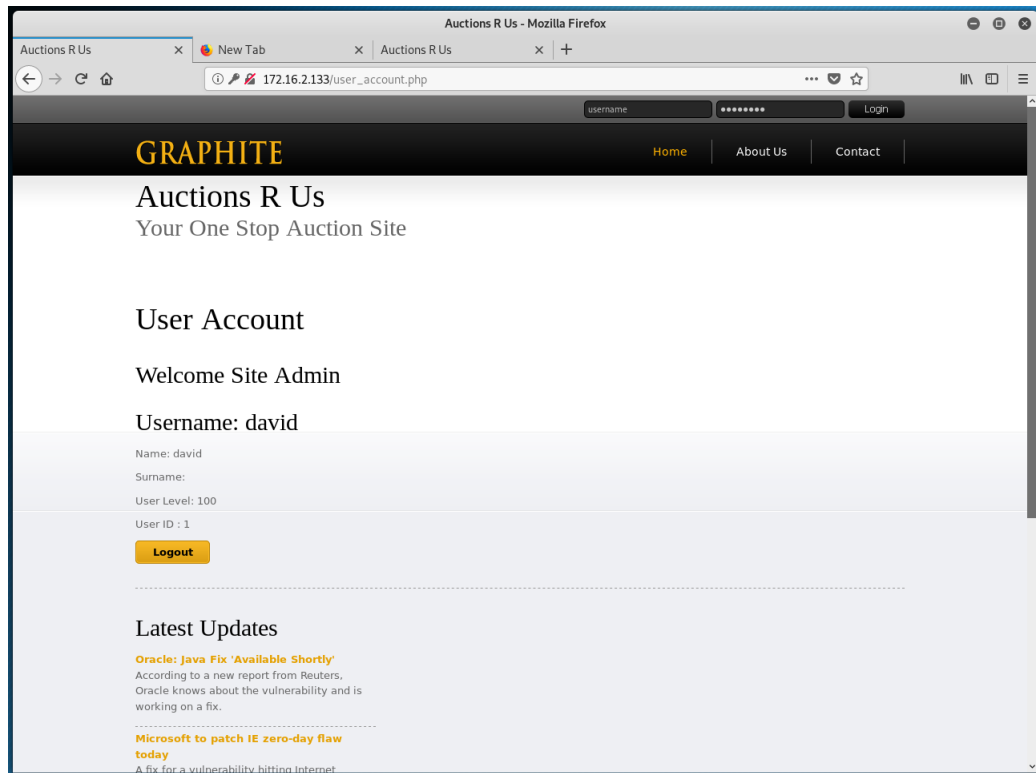
Figure 3.4: Authentication bypass SQLi

Figure 3.5: Admin panel access gained

Figure 3.6: Initial SQLmap testing

Figure 3.7: Finding DB users

Figure 3.8: Finding DB names

Figure 3.9: Finding table names

12

```
Database: auctionsrus
Table: auction_users
[4 entries]
+--------+---------------+---------+----------+-------------------------------------------+------------+------------------------------------------+
| userID | name          | surname | username | password                                  | user_level | user_level_md5                           |
+--------+---------------+---------+----------+-------------------------------------------+------------+------------------------------------------+
| 1      | david         | NULL    | david    | 509f8c419805dc16e7bd457e29155ef3 (newsletters) | 100   | f899139df5e1059396431415e770c6dd (100)   |
| 3      | Ginger Knowles| NULL    | gknowles | ac64504cc249b070772848642cffe6ff          | 1001       | b8c37e33defde51cf91e1e03e51657da (1001)  |
| 2      | john          | NULL    | john     | d00181ca30c1e884390fd694fb2a8137 (boffin) | 1000       | a9b7ba70783b617e9998dc4dd82eb3c5 (1000)  |
+--------+---------------+---------+----------+-------------------------------------------+------------+------------------------------------------+

[14:53:14] [INFO] table 'auctionsrus.auction_users' dumped to CSV file '/root/.sqlmap/output/172.16.2.133/dump/auctionsrus/auction_users.csv'
[14:53:14] [INFO] fetched data logged to text files under '/root/.sqlmap/output/172.16.2.133'
[14:53:14] [WARNING] you haven't updated sqlmap for more than 220 days!!!

[*] ending @ 14:53:14 /2020-03-10/

root@kali:~#
```

Figure 3.10: Dumping user data

3.1.3   Weak Authentication

Security Implications / Risk Level

Weak authentication is another blanket term for a multitude of security vulnerabilities surrounding the authentication systems on a website. Again, these exploits can manifest in a number of forms, and have varying levels of risk depending on the exploit.

During testing a number of vulnerabilities were found that could be classed under weak authentication. These were:

- Susceptibility to bruteforce attacks: There appeared to be no rate-limiting or IP blocking functionality on the website login form, allowing for bruteforce attacks to be performed easily. These attacks could potentially result in user accounts being accessed by attackers.

- Weakly hashed passwords: Using SQLmap, the users table was dumped, revealing that the passwords were hashed with unsalted MD5. This is a very weak algorithm, allowing it to be cracked quickly, as well as it having a plethora of rainbow tables already available online. This could result in hackers easily cracking passwords if they were able to retrieve the hashes.

- Weak session IDs: Instead of a secure session ID, the session IDs used within the website are simply an MD5 hash of the user level for that account. This is extremely dangerous, as an attacker can easily enumerate through the user levels, testing each one and gaining access to every account with ease.

Cause of Vulnerability

The exploits found each had varying causes, depending on which part of the website was being interacted with. These included:

- Susceptibility to bruteforce attacks: The lack of a rate-limit or IP block on each account for the login form allows attackers to attempt as many times as they want, which makes bruteforce attacks possible.

- Weakly hashed passwords: The use of the MD5 hashing algorithm results in weak hashing security, which stems from either legacy code (from

13

when MD5 was stronger) or poor security choices during the design of the system.

- Weak session IDs: Again, this is another issue stemming from poor security choices during the design stage. Session IDs should be chosen as a completely random string, so hackers cannot collate multiple hashes and find patterns between them. Using unsalted MD5 as the hashing algorithm for this makes it even worse, as it is very easy to reverse engineer the original text from the hashes due to the nature of the user ID being numeric.

Steps to Replicate

Bruteforcing the login form

- To bruteforce the login form, a custom python script was used (Appendix A). This script read in usernames from a 'usernames.txt' file, and passwords from a provided wordlist (in this case english top 10000 wordlist). For each username, the script iterated through the passwords, sending a GET request using the python 'requests' library. If the text 'User Account' was found in the response text, this would indicate a successful sign in and print the found credentials before moving on to the next username. Using the script successfully found the credentials for the admin user 'david' (Fig. 3.11), and could likely find the rest if a more extensive wordlist was used.

Cracking the password hashes from SQLmap

- The password hashes gained from SQLmap were unsalted MD5, so one of the first things to check would be an online rainbow table (a database of hashes and their corresponding plaintexts). In this case, the website 'Crackstation' was used to reverse all three hashes (Fig. 3.12).

Enumerating the session IDs

- To enumerate the session IDs, another custom python script was created (Appendix B). This script enumerated through the numbers 0-1500, hashing each number of iteration. With the hash, the requests library was again used to send a GET request to the user_account.php page, with the hash set as a cookie. The response text of the request could then be analysed - first to check if any sign in was detected, which would then print the hash used, user level, and username. It would then check if the admin panel text was present, and if so, print that the user was an admin. This method was used to gain access to all 3 auction accounts (Fig. 3.13).

Figure 3.11: Bruteforcing the David user

Figure 3.12: Cracking the hashes from SQLmap

Figure 3.13: Enumerating the session IDs

3.2 SRV1

3.3 SRV2

3.4 Ubuntu Client

Chapter 4

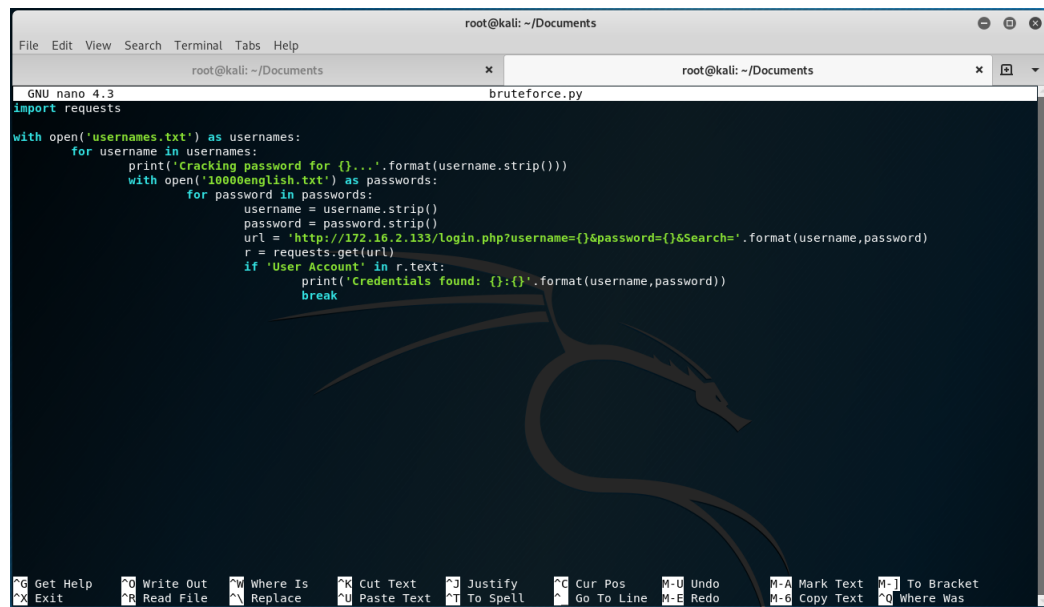# Recommendations

4.1 Auction Site

4.2 SRV1

4.3 SRV2

4.4 Ubuntu Client

Chapter 5

Conclusion

Chapter 6

Appendices

6.1 Appendix A: Python bruteforce script

Figure 6.1: Login form bruteforce script

6.2 Appendix B: Session enumeration script

Figure 6.2: Enumerating the session cookies

Chapter 7

References