

awsum-quotes

The program is a collection of quotations, each composed of two strings: the actual text of the quotation and the author.

The collection is implemented as a structure which contains:

- number of stored quotations, as an integer
- array of quotations (length 8)

Each quotation is also a structure:

- text (length 80)
- author (length 30)

The main menu allows the following operations:

1. Show all the quotations
2. Show a specific quotation
3. Add a new quotation to the collection

During the initial setup, the program fills 5 out of the 8 elements available in the quotations array, so the user can add only 3 more quotations. If it tries to add more, the program will terminate.

Some of the default quotations may give small hints on the solution of the challenge.

Build

Compiled with gcc 7.5.0 (Ubuntu 18.04)

```
gcc -o awsum_quotes awsum_quotes.c
```

Writeup

An unexpected gadget

`checksec` shows that the binary has all the security options in place, so some kind of ROP will probably be necessary. We run `ropper` to see if there are some useful gadgets:

```

0x00000000000010dc: pop r12; pop r13; pop r14; pop r15; ret;
0x00000000000010de: pop r13; pop r14; pop r15; ret;
0x00000000000010e0: pop r14; pop r15; ret;
0x00000000000010e2: pop r15; ret;
0x000000000000983: pop rbp; jmp rax;
0x00000000000010db: pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
0x00000000000010df: pop rbp; pop r14; pop r15; ret;
0x000000000000990: pop rbp; ret;
0x000000000000994: pop rdi; xor rax, rax; mov al, 0x3b; syscall;
0x00000000000010e3: pop rdi; ret;
0x00000000000010e1: pop rsi; pop r15; ret;
0x00000000000010dd: pop rsp; pop r13; pop r14; pop r15; ret;
0x000000000000aff: push rax; add dword ptr [rax - 0x75], ecx; call 0xffffffffc9901b92; ret;
0x000000000000902: ret 0x2016;
0x000000000000a9a: ret 0xfffd;

```

Well, that's unusual: at 0x994 there is a gadget that seems to be preparing the registers for the syscall `SYS_EXECVE`. Why should the program execute those instructions? Checking with Ghidra, we see that the instructions are not recognized by the disassembler and they are not reached by any flow of execution:

		LAB_00100990			
00100990	5d	POP	RBP		
00100991	c3	RET			
00100992	0f	??	0Fh		
00100993	1f	??	1Fh		
00100994	5f	??	5Fh	-	
00100995	48	??	48h	H	
00100996	31	??	31h	1	
00100997	c0	??	C0h		
00100998	b0	??	B0h		
00100999	3b	??	3Bh	;	
0010099a	0f	??	0Fh		
0010099b	05	??	05h		
0010099c	90	??	90h		
0010099d	00	??	00h		
0010099e	00	??	00h		
0010099f	00	??	00h		


```

*****
*
*                               FUNCTION
*****
undefined register_tm_clones()
AL:1 <RETURN>
register_tm_clones
001009a0 48 8d 3d LEA RDI, [__TMC_END__]
        69 16 20 00

```

The area of code in which the gadget resides is actually *padding code* that the compiler adds so that the beginning of some functions are aligned at nibble level (4 bits). Normally in x64 this padding code is a [multi-byte NOP](#) (or a sequence of them):

bytes	sequence	encoding
1	90H	NOP
2	66 90H	66 NOP
3	0F 1F 00H	NOP DWORD ptr [EAX]
4	0F 1F 40 00H	NOP DWORD ptr [EAX + 00H]
5	0F 1F 44 00 00H	NOP DWORD ptr [EAX + EAX*1 + 00H]
6	66 0F 1F 44 00 00H	NOP DWORD ptr [AX + AX*1 + 00H]
7	0F 1F 80 00 00 00 00H	NOP DWORD ptr [EAX + 00000000H]
8	0F 1F 84 00 00 00 00 00H	NOP DWORD ptr [AX + AX*1 + 00000000H]
9	66 0F 1F 84 00 00 00 00 00H	NOP DWORD ptr [AX + AX*1 + 00000000H]

In this case, the multi-byte NOP has been partially overwritten with the instructions of the gadget. That's interesting.

Let's go back to the list of gadgets and see if there is everything we need to prepare the registers for the *SYS_EXECVE*: we need to set RAX, RDI, RSI and RDX. The gadget mentioned takes care of RDI and RAX, there is another gadget that pops RSI but there is nothing for RDX. RDX will point to the environment of the execve so it can't be a random value (the easiest way is to make it a pointer to NULL), but there is no gadget that gives us control over it.

But wait, let's think about what we saw until now: somebody put instructions that seem like a backdoor inside some padding code. Maybe there is other padding code that has been altered in the same fashion?

00100983	5d	POP	RBP
00100984	ff e0	JMP	RAX=>_ITM_deregisterTMCloneTable
-- Flow Override: CALL_RETURN (COMPUTED_CALL_TERMINA			
00100986	66	??	66h f
00100987	5a	??	5Ah Z
00100988	48	??	48h H
00100989	89	??	89h
0010098a	d6	??	D6h
0010098b	eb	??	EBh
0010098c	07	??	07h
0010098d	00	??	00h
0010098e	00	??	00h
0010098f	00	??	00h

Ha-ha! Just before the bytes of the backdoor gadget there is another strange padding sequence. Disassembling those bytes we find:

```

pop rdx
mov rsi, rdx
jmp <+0x7>

```

This small block jumps to the backdoor gadget, so they are actually a single magic gadget that pops everything that is needed to call *SYS_EXECVE*. With this gadget at disposal we only need to somehow prepare the stack with a pointer to NULL (8 null bytes) and a pointer to the string *"/bin/sh"*. The latter can be found in *.rodata* because it is used as the author string for one of the default quotations.

At this point we have a clear idea of what to do: find some way to hijack the flow of the program and execute the magic gadget. Let's see what vulns we can find.

Buffer overflow

There is an evident buffer overflow in the `get_new_quote()` function, both in the text of the quotation and in the author buffers. The program calls two `read()` of 0x90 bytes for buffers of length 80 (text) and 30 (author).

Also, the text buffer is printed to stdout before reading the the author, so... what about overflowing the buffer until we reach the canary, so that the canary itself is printed as part of the text buffer?

Leaking stuff

Let's look at the stack we get before sending our input:

0x7fff13087cc0:	esp	0x00007f0bf3e14a00	0x00005624b3dfe260	
0x7fff13087cd0:		0x000000000000000a	0x0000000000000000	
0x7fff13087ce0:		0x00007f0bf3e14a00	0x00007f0bf3ab73f2	
0x7fff13087cf0:	author	0x0000000000000000	0x0000000000000000	
0x7fff13087d00:		0x0000000000000000	0x0000000000000000	
0x7fff13087d10:	text	0x0000000000000000	0x0000000000000000	
0x7fff13087d20:		0x0000000000000000	0x0000000000000000	
0x7fff13087d30:		0x0000000000000000	0x0000000000000000	
0x7fff13087d40:		0x0000000000000000	0x0000000000000000	
0x7fff13087d50:		0x0000000000000000	0x0000000000000000	
0x7fff13087d60:		0x0000000000000000	0x07b8f36c3cac6e00	canary
0x7fff13087d70:	ebp	0x00007fff13087da0	0x00005624b2a1203d	saved IP
0x7fff13087d80:		0x00000003b2a12080	0x00005624b3dfe260	
0x7fff13087d90:		0x00000a3313087e80	0x07b8f36c3cac6e00	

Highlighted in blue and green are the beginning of the text and author buffers, respectively.

Our end goal is to overwrite the **saved IP** with the address of the magic gadget. Since the binary is PIE, a leak of the `.text` section is needed in order to calculate that address. Can we leak the saved IP?

Yes, but it's a bit convoluted. We can't leak it directly because the **canary** would be modified in the process and the program would exit before returning from `get_new_quote()`. What we can do is to leak the canary first, together with the **saved BP**, and *only then*, leak the saved IP.

It's time to add some quotes.

Quote #1

leak canary + saved BP

The idea is to write enough bytes in the text buffer to reach the canary. We need 0x58 bytes (writing into memory goes towards the higher addresses). Since `printf` prints until the character terminator (null byte `\x00` for C programs), we need to overwrite also the least significant byte of the canary - notice that the canary is implemented so that it always has a null byte at the end.

Since the program performs a substitution of the first `\n` (newline) character in the buffer with a null byte, we must make sure that also the newline byte `\x0a` is not in our buffer.

Can we? Well, the we control the payload of $0x58 + 1$ bytes, so no problem for that, but don't forget that also the canary itself will be part of the printed stuff and we have no control on it whatsoever: it is a random number generated at the beginning of the execution. From now on we assume that the canary does not contain any newline or null characters (besides the least significant byte that is always `\x00`). We'll discuss the other case later. This reasoning applies also to the other values that we will leak during this exploit.

Together with the canary, the `printf` will also print the saved BP. It won't print further than that because the 2 most significant bytes of the saved BP are always `\x00`.

After the leak, we can restore the least significant byte of the canary with an overflow in the author buffer. This time the payload will be $0x78 + 1$ bytes long, with the last byte set to `\x00`. This way the program won't detect any change to the canary.

Quote #2

leak saved IP

At this point we know the canary and the saved BP. We can leak the saved IP following the same process described for the first quote:

1. write enough bytes to reach the value to leak (text buffer overflow)
2. restore values on the stack to let the execution go on (author buffer overflow)

For the first part $0x68$ bytes are needed. For the second one, we must restore both the canary and the saved BP.

Quote #3

final trigger

Now we know where to jump. After finding the addresses of `"/bin/sh"` and of 8 `\x00` bytes in the executable, let's send everything in the text buffer (*from low to high addresses*):

- padding bytes
- canary
- saved BP
- address of the gadget
- address of null bytes
- address of `"/bin/sh"`

After sending a short author string, we get the shell.

```
> cat flag
```

Issues

Throughout the exploit we assumed that the leaked values didn't have any `\x00` or `\x0a` bytes. These values are all random: the **canary** is random by implementation, the addresses of the stack (**saved BP**) and the .text segment (**saved IP**) are randomized by ASLR. There is nothing that can be done about that, that's why this exploit is not always succesful.

Testing 30 runs gave a success rate of ~75%.

Hosting

The challenge can be hosted on a system with the latest libc-2.31. The exploit does not make use of libc addresses.