Cyberthon: Operating Systems

Reverse(Faze) [1000]

The hacker left a binary executable behind in ShoppingBaba's server as a way to connect back to their C2 server's ssh service at p7ju6oidw6ayykt9zeglwyxired60yct.ctf.sg:16382

Please help to find the credentials left behind inside the binary executable and find the flag on the hacker's server!

Files: reverse_faze

Taking Off

```
USAGE:
./reverse_faze <ip_addr>

ip_addr:        C2 server's IP address [\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}], e.g. 192.168.0.1

DESCRIPTION:
        Report alive to C2 server.
```

Regex is a dour to the eyes, sometimes.

We're going to skip the description and dive right into Hex-Rays.

```
__int64 __fastcall main(int argc, char **argv, char **a3)
{
  __int64 rdx1; // rdx
  __int64 rdx2; // rdx
  __int64 rdx3; // rdx
  unsigned int exit_code; // eax
  __int64 ip_copy; // [rsp+0h] [rbp-88h]
  unsigned __int64 cookie; // [rsp+68h] [rbp-20h]

  cookie = __readfsqword(0x28u);
  if ( argc != 2 || (unsigned int)ip_regex_check((__int64)argv) )
  {
    print_usage(argv);
  }
  else if ( (unsigned int)faze1((__int64)argv, (__int64)argv, rdx1)
         && (unsigned int)faze2((__int64)argv, (__int64)argv, rdx2) == 42
         && (unsigned int)faze3((__int64)argv, (__int64)argv, rdx3) == 1 )
  {
    __isoc99_sscanf(argv[1], "%s", &ip_copy);
    __sprintf_chk(
      0LL,
      1LL,
      -1LL,
      "echo %s | ssh -o StrictHostKeyChecking=no %s@%s \"TIME=`date` echo alive at $TIME
      password);
    puts("\n");
    exit_code = system(0LL);
    __printf_chk(1LL, "exit code %d\n", exit_code);
  }
  else
  {
    puts("[-] Complete all 3 fazes to successfully report back alive.");
  }
  return 0LL;
```

I've taken the liberty of renaming a couple of variables to make the code a little bit more readable.

The binary is a simple crackme with 3+1 distinct `faze`s. First, we pass a valid IP through `argv` to actually get to the first phase:

```
$ ./reverse_faze 0.0.0.0
Enter key for faze 1: ?????
[-] Complete all 3 fazes to successfully report back alive.
```

Which runs on this code:

```
14  cookie = __readfsqword(0x28u);
15  __printf_chk(1LL, "Enter key for faze 1: ", a3);
16  fgets(s, 100, stdin);
17  __isoc99_sscanf(s, "%s", &input);
18  *(&input_pminus1 + strlen(&input) + 1) = 0;
19  rtr = 0;
20  if ( input == hex_FF - 177 && input_p4 == '%' && !input_p5 && input_p3 == hex_FF - 181 && input_p1 == '6' )
21    rtr = input_p2 == hex_FF - 220;
22  for ( i = 0LL; strlen(&input) > i; ++i )
23    password[i] = *(&input + i);
24  return (unsigned int)rtr;
25}
```

Normally, you'd use something like `angr` to escape the tedium of manual RE[1].

My team was rather stuffed for time during the competition, and the binary was short enough to eyeball within the hour:

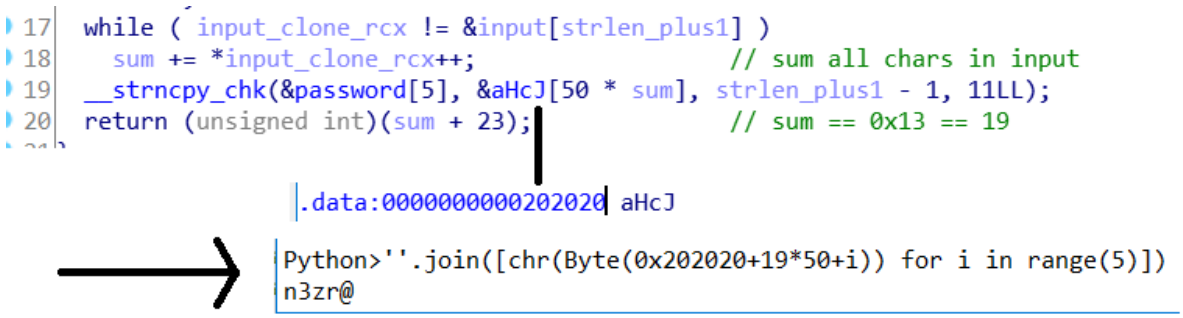
```
>>> r = process(['stdbuf', '-i0', '-o0', './reverse_faze', '0.0.0.0'])
[x] Starting local process '/usr/bin/stdbuf'
[+] Starting local process '/usr/bin/stdbuf': pid 77
>>> r.sendlineafter(': ', 'N6#J%')
'Enter key for faze 1'
>>> r.sendlineafter(': ', '\x13')
'Enter key for faze 2'
>>> r.sendlineafter(': ', '1 3 3 7')
'Enter key for faze 3'
```

AN: *Non-linebuffered challenges are really, really irksome*

After passing the three functions, you'll be able to find the ssh credentials (`hacker:N6#J%...`) in `gdb` by crudely spamming `n` enough times:

```
gdb-peda$ n
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7fa289e01130 ("echo %s | ssh -o StrictHostKeyChecking=no %s@%s \"TIME=`da
te` echo alive at $TIME >> /home/%s/victim.log\" && history -c")
RDX: 0xffffffffffffffff
RSI: 0x1
RDI: 0x0
RBP: 0x7fffd5dc8640 --> 0x302e302e302e30 ('0.0.0.0')
RSP: 0x7fffd5dc8630 --> 0x7fffd5dc8640 --> 0x302e302e302e30 ('0.0.0.0')
RIP: 0x7fa289e00f4d
R8 : 0x7fa28a0029f0 ("N6#J%nrd12MZjHU")
R9 : 0x7fa289e0108f --> 0x72656b636168 ('hacker')
R10: 0xffffffff
R11: 0xfffffffffffff98
R12: 0x7fa289e009c0
R13: 0x7fffd5dc87a0 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
```

Now, that password in the box **isn't actually correct**. I never figured out why PEDA got the password wrong, but you can correct it manually by inspecting `faze2()` & querying `IDAPython` for what the correct value *should* be:



I'm compressing a more convoluted explanation here, but essentially this changes the characters `nrd12` in the password found from `gdb` into `n3zr@`, making the final password `N6#J%n3zr@MZjHU`.

Confused? So I am, but what works will work:

```
$ ssh -p 16382 hacker@p7ju6oidw6ayykt9zeglwyxired60yct.ctf.sg cat flag.txt
hacker@p7ju6oidw6ayykt9zeglwyxired60yct.ctf.sg's password:
Cyberthon{isnt_this_really_ezaf?}
```

Flag

`Cyberthon{isnt_this_really_ezaf?}`

Footnotes

1. That's not to say I didn't try; I did, and it got complicated fast.