Shellcode

Georgios (George) Portokalidis

How to Write Shellcode

- **Code in assembly**

```
# comments start with #
# Your program needs to eventually call exit(0), unless it
# is running in an infinite loop

   .global _start        # Define symbol _start as global

   .text                 # Whatever follows goes into .text
 _start:                 # The program entry point symbol
   mov     $1, %rax
   mov     $1, %rdi
```

How to Write Shellcode

- Code in assembly
- **Compile with gcc**

```
$ gcc -m64 -nostdlib -no-pie -o hello64.bin hello64.s
```

How to Write Shellcode

- Code in assembly
- Compile with gcc
  - **If your code is expected to run on its own, you can execute it to test it**

```
$ gcc -m64 -nostdlib -no-pie -o hello64.bin hello64.s
```

How to Write Shellcode

- Code in assembly
- Compile with gcc
  - If your code is expected to run on its own, you can execute it to test it
- **View generated code**

```
$ objdump -d hello64.bin
hello64.bin:  file format ELF64-x86-64
Disassembly of section .text:
00000000004000d4 _start:
  4000d4: 48 c7 c0 01 00 00 00          movq   $1, %rax
  4000db: 48 c7 c7 01 00 00 00          movq   $1, %rdi
```

How to Write Shellcode

- Code in assembly
- Compile with gcc
  - If your code is expected to run on its own, you can execute it to test it
- View generated code
- **Extract machine code only**

```
$ objcopy -O binary --only-section=.text hello64.bin hello64.sc
```

How to Write Shellcode

- Code in assembly
- Compile with gcc
  - If your code is expected to run on its own, you can execute it to test it
- View generated code
- Extract machine code only
- **View Bytes in hex**

```
$ hexdump –C hello64.sc
```

How to Write Shellcode

- Code in assembly
- Compile with gcc
  - If your code is expected to run on its own, you can execute it to test it
- View generated code
- Extract machine code only
- View Bytes in hex
  - **Encode them in a string**

```
$ hexdump -v hello64.sc  -e '"\\""x" 1/1 "%02x" ""'
```

Getting Things Done

- For anything interesting you need to execute system calls
- Linux: system call API is powerful, easy to use, and well documented
  - Can be used in a straightforward manner from assembly
- Windows: system call API is harder to use and not well documented
  - Using API functions calls is preferable
- Calling functions is easy, if you know their offset from the call instruction
  - Requires additional work

Calling System Calls on 32-bit Linux

- Use interrupt 0x80 using the `int` instruction
- The number of the syscall has to be passed in register %eax
- The kernel interface uses %ebx, %ecx, %edx, %esi, %edi and %ebp for passing arguments
  - All registers are preserved
- Register %eax contains the result of the system call.

Calling System Calls on 64-bit Linux

- Use the `syscall` instruction
- The number of the syscall has to be passed in register %rax
- The kernel interface uses %rdi, %rsi, %rdx, %r10, %r8 and %r9 for passing arguments
  - The kernel destroys registers %rcx and %r11
  - System-calls are limited to six arguments, no argument is passed directly on the stack
- Returning from the syscall, register %rax contains the result of the system call. A value in the range between -4095 and -1 indicates an error, it is -errno
- Note: 64-bit kernels can also execute 32-bit binaries, so 32b Linux system calls can also be executed using `int 0x80`

Linux System Call Table

- https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

x86_64 (64-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-
2	open	man/ cs/	0x02	const char *filename	int flags	umode_t mode	-	-	-
3	close	man/ cs/	0x03	unsigned int fd	-	-	-	-	-
4	stat	man/ cs/	0x04	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
5	fstat	man/ cs/	0x05	unsigned int fd	struct __old_kernel_stat *statbuf	-	-	-	-
6	lstat	man/ cs/	0x06	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
7	poll	man/ cs/	0x07	struct pollfd *ufds	unsigned int nfds	int timeout	-	-	-
8	lseek	man/ cs/	0x08	unsigned int fd	off_t offset	unsigned int whence	-	-	-
9	mmap	man/ cs/	0x09	?	?	?	?	?	?
10	mprotect	man/ cs/	0x0a	unsigned long start	size_t len	unsigned long prot	-	-	-
11	munmap	man/ cs/	0x0b	unsigned long addr	size_t len	-	-	-	-

Example: Hello World Shellcode

- Write “Hello World” to standard output
- Gracefully terminate program

Calling write()

- Find the API for sys_write()

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			

- write(1, “Hello World”, 11);
  - 1 → file descriptor corresponding to **stdout**
  - “Hello World” → Pointer to data to be written
  - 11 → Number of bytes to be written

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout

        mov     $11, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write
```

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout

        mov     $11, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

message:
        .ascii  "Hello world\n"
```

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                    # system call 1 is write
        mov     $1, %rdi                    # file handle 1 is stdout
        mov     $message, %rsi
        mov     $11, %rdx                   # number of bytes
        syscall                             # invoke operating system to do the write

message:
        .ascii  "Hello world\n"
```

Calling exit()

- Find the API for sys_exit()

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
60	sys_exit	int error_code					
61	sys_wait4	pid_t upid	int *stat_addr	int options	struct rusage *ru		

- exit(0);
  - 0 → return value for correct termination

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                        # system call 1 is write
        mov     $1, %rdi                        # file handle 1 is stdout
        mov     $message, %rsi
        mov     $11, %rdx                       # number of bytes
        syscall                                 # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax
        xor     %rdi, %rdi                      # we want return code 0
        syscall                                 # invoke operating system to exit
message:
        .ascii  "Hello world\n"
```

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi
        mov     $11, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello world\n"
```

xor reg, reg
sub reg, reg

Common idiom on x86 for zeroing a register

Avoiding Absolute Addresses

- Use of absolute addresses implies the code needs to be loaded at a specific address → hard to inject in another process
- 64-bit processors allow RIP relative addressing → position independent code (PIC)

```
mov     $message, %rsi       → mov     0x4000fe, %rsi
mov     message(%rip), %rsi  → lea     0x15(%rip),%rsi
```

- Alternatively, can use call-pop combination to load current PC or other shellcode address on register

```
        call   GETPC
        add    data_offset, %eax

        rest of shellcode

GETPC:  pop    %eax
        jmp    *%eax

data:   ...
```

```
        jmp    A
GETA:   pop    %eax

        rest of shellcode

A:      call   GETA
data:   ...
```

“Special” Bytes Limitations

- Certain characters may not be allowed
  - strcpy() stops copying at null byte
  - gets() reads one line at a time (stops at ‘\n’)
  - Input may need to be alphanumeric
- Bypasses
  - Rewrite shellcode to avoid characters
    - Alternate instructions can achieve a similar result
    - Use multiple instructions and ALU operations to construct constants and addresses at run time
  - Encode shellcode
    - A 1st stage shellcode decodes the 2nd stage and then executes it