Shellcoding, an introduction

Daniele Bellavista

University of Bologna, Scuola di Ingegneria ed Architettura
Cesena Security Network and Applications

December 11, 2012

Outline

Warm-Up

Materiale iniziale

- La teoria di questa presentazione vale per sistemi Unix, FreeBSD e Windows.
- Il materiale è tuttavia preparato per sistemi Linux 64 bit con kernel $>=$ 2.6 e processore $>$ Pentium 4.
- Per chi non ha 64 bit sono disponibili anche gli esempi a 32 bit.
- Per chi non ha installato un OS Linux o non vuole sporcare il suo, sono disponibili due Virtual Machine:

  32 bit: *sorry non ho fatto in tempo...*
  64 bit: *sorry non ho fatto in tempo...*

What is shellcoding?

Exploiting software vulnerabilities

- Vulnerabilities like *BOF*s permit the insertion and the execution of a custom payload.
- The historical function of the payload is to spawn a shell (hence the name *shell-code*).
- If the exploited program runs as a privileged user, the payload can assume the control of the system.

Prerequisites

Basic notion of:

- Assembly language.
- Memory structure.
- System Calls.

System Memory

Memory management

- Memory management is a vast topic, so let's discuss from an high-level viewpoint.
- Thanks to the *Virtual Memory Management*, each process sees a flat addressing space of 128 TiB (4 GiB in 32 bit processors).
- Users can create sections inside the memory with read, write and/or execute permissions.
- The basic operations for memory management are *push* and *pop*.

  push Insert a 64-bit value into the top of the stack (the bottom).

  pop Retrieve a 64-bit value from the top of the stack (the bottom).

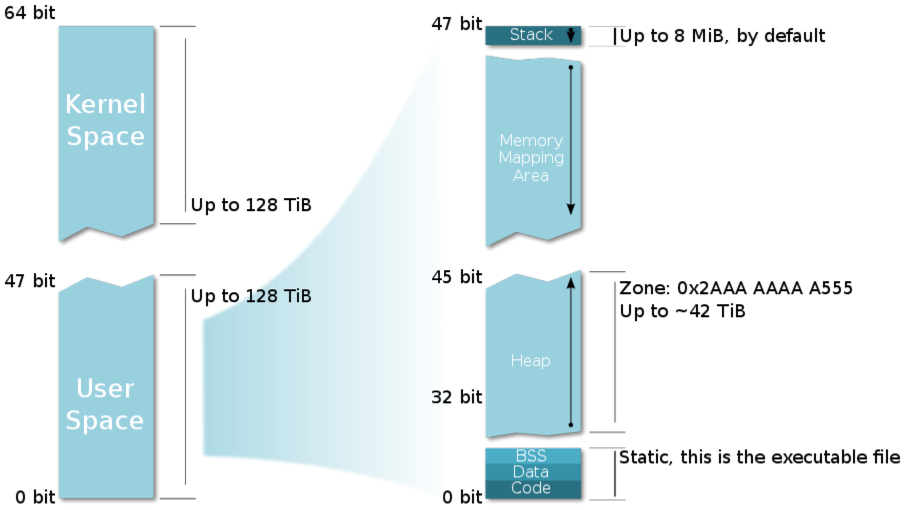


Figure : Linux 64 bit memory management

Program Code

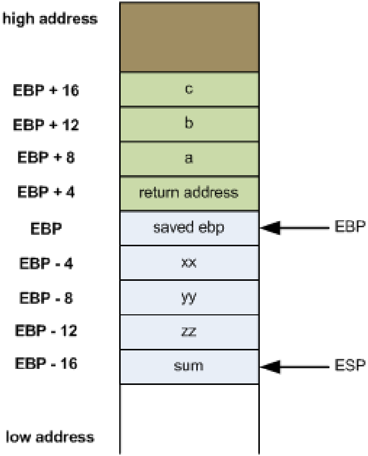
Text segment and instruction pointer

- The code is saved in a RO and executable memory called *Text segment*.
- The current instruction is pointed by the *Instruction Pointer*.
- The instruction pointer is a value stored into the CPU.

Program Stack

- The stack is where function context, called frame, resides.
- The *Base pointer* points to the begin of the frame.
- The *Stack pointer* points to the first available memory location.

```
void f() {
  int xx = 12; int yy = 23;
  int zz = 24;
  int sum = xx + yy + zz;
}
```



Assembly Language

Low, low level programming

- Assembly is the final stage of the programming process.
- Each instruction is directly converted into an number.
- Writing the number on the CPU Command BUS cause the instruction to be executed.
- You can access to a set of 64 bit registers: *rax*, *rbx*, *rcx*, *rdx*, *rsi*, *rdi*, *r8*, *r9*, *r10*, *r11*, *r12*, *r13*, *r14*, *r15*
- As instruction pointer, base pointer and stack pointer the CPU uses respectively *rip*, *rbp* and *rsp*.

Assembly Language: function call

The call/ret instructions

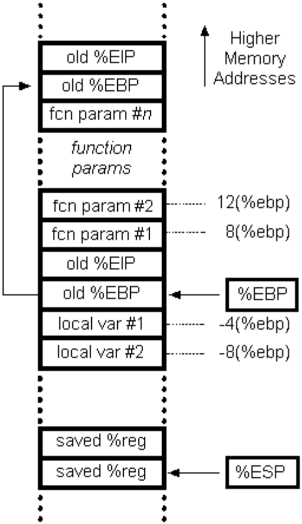
call address: push the RIP into the stack.

ret: pop the RIP from the stack.

Creating a frame for the function

- The callee may need a stack for his own variables.
- *RBP* is saved (pushed) into the stack.
- The new *RBP* become the old *RSP*.
- Before doing ret, the opposite operations must be performed.

Assembly Language: function call and stack



Assembly Language: function call and parameters

Where should I put the parameters?

- If you are coding in pure ASM, you can do what you want.
- Else, you have to follow the conventions (see `http://en.wikipedia.org/wiki/X86_calling_conventions#List_of_x86_calling_conventions`).

Linux calling convention, parameter order

First parameters:

x86-64: rdi, rsi, rdx, rcx, r8, and r9. For system calls, r10 is used instead of rcx.

IA-32: ecx, edx. For system calls ebx, ecx, edx, esi, edi, ebp.

Other parameters are pushed into the stack.

System calls

Invoking the OS

- User's programs run in the exterior ring with *PL* 3, while kernel runs in the inner ring with *PL* 0: Kernel can access to the hardware (files, devices, ...), user not.
- Syscalls form the interface between user and kernel.
- E.g.: *open()*, *read()*, *write()*, *chmod()*, *exec()*, ...

Try to use Syscalls

Using assembly

- In order to ease the programmer duty, syscalls are identified by a number.
- In modern x86-64 processor, a syscall is invoked using the operation *syscall*, putting the id in the register rax.
- In modern x86 processor, a syscall is invoked using the operation *sysenter* but a lot of operation must be performed before, so in the following exercise we will use the old-fashion-way *int 80h*, which rises a software interrupt.

```
mov rdi, 0
mov rax, 60
syscall
```

```
mov ebx, 0
mov eax, 1
int 80h
```

Linux Syscalls

Linux

- Syscalls signatures: */…kernel-sources…/include/linux/syscall.h*
- Syscalls numbers: */usr/include/asm/unistd_64.h*

```
unistd_64.h
  #define __NR_exit 60
syscall.h
  asmlinkage long sys_exit(int error_code);
```

The final preparation

First steps...

- Testing platform: `http://dl.dropbox.com/u/16169203/data.zip`
- Use nasm to compile your assembly code
- Feed the tester with the output.

Exercise 0

- Create a shellcode that does nothing!
- Test it!

Using a syscall

Exercise 1: The exit syscall

- Use the exit syscall to terminate the program
- Change your code setting the return code to $13h$

Data reference I

The reference problem

- When you write a program you can use global data because, at compile time, a static address is associated.
- But your shellcode is not compiled with the program it's intended to run.
- You must use relative addressing, but before IA-64 it was not possible.

Data reference II

Old IA-32 way

- You use a trick: jmp just before the data location, then do a call. Et voilà! On the top of the stack there is the data address.

```
jmp message
run:
  pop ebx ; now ebx contains the string reference
  ; ... shellcode
message:
  call run
  db 'CeSeNA',0x0a,0
```

Data reference III

New IA-64 way

- IA-64 introduces the RIP relative addressing.

```
lea rdi, [rel message] ; now ebx contains
                       ; the string reference
; ... shellcode

message db 'CeSeNA',0x0a,0
```

Data reference IV

Generic Way

- You can pop the string in hex format over the stack.
- The stack pointer is then the string reference.

```
push 0x000a414e ; 0x00, 0x0a, 'AN'
push 0x65536543 ; 'eSeC'
mov ebx, esp ; now ebx contains the string reference
; ...
```

Data reference V

Exercise 2: print on stdout

- Get the exercise skeleton.
- Understand the code
- Write your own message

Execute a program

```
unistd_64.h
  #define __NR_execve 59
syscall.h
  int kernel_execve( const char *filename,
                     const char *const argv[],
                     const char *const envp[]);
```

Exercise 3: A real shellcode, exec a shell!

4 HaXoRs

4 g33k

4 n00bz

Reverse shell I

One shellcode to rule them all

Step to execute:

1. Open a socket to the attacker server.
2. Duplicate the socket file descriptor into 0 and 1 (and optionally 2).
3. Exec a shell.

Step1: Create a socket

- The standard procedure involves socket creation and connection.
- *socket()* and *connect()* syscalls.
- The sockaddr_in structure requires port in network byte order (*htons()*) and ip address in numeric form (*inet_pton()*).
- The usual sockaddr_in size is 16.

Reverse shell II

```
long sys_socket(int domain,
                int type,
                   int protocol);
long sys_connect(int fd,
                 struct sockaddr __user *,
                 int addrlen);
```

```
struct sockaddr_in {
short              sin_family; // AF_INET
unsigned short     sin_port; // network order (htons())
struct in_addr     sin_addr; // As 32 bit
char               sin_zero[8];
};
```

Reverse shell III

Step2: Duplicate the file descriptor

- The return value of socket() is the file descriptor.
- The syscall dup2() copy the file descriptor and close the destination.
- dup2(fd, 0); dup2(fd, 1), dup2(fd, 2);

```
long sys_dup2(unsigned int oldfd,
              unsigned int newfd)
```

Reverse shell IV

Step3: Exec a shell

Already seen ;)

Exercise 4: Reverse shelling

Remember to open a server listening into a known address! E.g.: *nc -l -p 1234*, preparing reverse shell for 127.0.0.1:1234

4 HaXoRs

4 g33k

4 n00bz

Shellcode optimization

Why optimization?

- When writing a shellcode, you must consider various factors. The most important are:
  1. How the vulnerable program receives the payload.
  2. How long the payload can be.
- The previous exercise solutions, for instance, are not suitable in most cases.
- The next slides will discuss about the *NULL* byte presence.
- About the payload length, it's all about the exploiter expertise.

String payload

zero-bytes problem

- If the payload is red as a string, C interprets a zero byte as string terminator, cutting the shellcode.
- Zero-bytes presence is caused by data and addresses:
  - *mov rax, 11h* is equivalent to *mov rax, 0000000000000011h*.
  - *lea rax, [rel message]* is equivalent to *lea rax, [rip + 0000...xxh]*.
  - *execve* for instance, requires a null terminated string and some null parameters.
- Solutions are quite straightforward:
  - Use *xor* operation to zero a register.
  - Use when possible smaller part of registers (e.g.: rax → eax → ax → [ah,al])
  - Use *add* operation: immediate operators are not expanded.
  - Place not-null marker in strings and substitute them inside the code.
  - When using relative addressing, place the message above: offset will be negative [3].

Zero-byte removal example

```
; Set rax = 60h
xor rax, rax
mov al, 60
```

```
; Set to 0 a mem area
null db 'xxxx'
xor rbx,rbx
mov [rel null], ebx
```

```
; Set rdi = 12h
xor rdi, rdi
add rdi, 12h
```

```
; terminate string with 0
message db 'CeSeNA','x'
xor rbx,rbx
lea rdi, [rel message]
mov [rdi+7], bl
```

```
; Negative reference
message db 'CeSeNA','x'
lea rdi, [rel message]
```

References

Linux Foundation.
Linux documentation.
http://linux.die.net/.

Intel.
Intel® 64 and IA-32 Architectures Software Developer's Manuale Combined Volumes:1, 2A, 2B, 3C, 3A, 3B and 3C.
Intel, August 2012.

Mark Loiseau.
64 bit linux shellcode.
http://blog.markloiseau.com/2012/06/64-bit-linux-shellcode/.

G. Adam Stanislav.
ttp://www.int80h.org/.