CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

SPONSOR PLATINUM

accenture

aizoon AUSTRALIA EUROPE USA
TECHNOLOGY CONSULTING

B5

eni

exprivia | ITALTEL

IBM

KPMG

LEONARDO

NTT DATA
Trusted Global Innovator

NUMERA
SISTEMI E INFORMATICA S.p.A.

Telsy

SPONSOR GOLD

bip.

cisco

MONTE DEI PASCHI DI SIENA
BANCA DAL 1472

negg

NOVANEXT
connecting the future

pwc

SPONSOR SILVER

DGi ONE
the leading digital company

ICT CYBER CONSULTING

Software Security 3

Arbitrary Code Execution

Michele LORETI
Univ. di Camerino
michele.loreti@unicam.it

CYBER CHALLENGE
CyberChallenge.IT

cini Cybersecurity National Lab

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Outline

- Code injection
- Code reuse
  - Return to libc
  - Return Oriented Programming (ROP)
  - Jump Oriented Programming (JOP)
- Shellcode and pwntools

Arbitrary Code Execution (ACE)

- The attacker's ability to execute arbitrary commands or code on a target machine or in a target process
- Can be induced by:
  - Code injection
  - Code reuse

Code injection

- Attackers exploit software flaws to introduce malicious code into a vulnerable computer program:
  1. Some instructions are injected exploiting one of the input sections
  2. The program flow is redirected to them

Example: shellcode injection

- Aimed at executing a *shell command;*
- Very popular attack against remote servers;
- The injected payload is just a system call invoking the terminal (located at /bin/sh in Unix systems);
- Once obtained a shell, any command can be issued to the system, any rogue file can be created, any information can be easily stolen, etc.

Example: shellcode injection

- Let us consider the following code:

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

void welcome(char *name)
{
    char buf[10];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    welcome(argv[1]);
    return 0;
}
```


Example: shellcode injection

- This code contains a clear vulnerability:

```
char buf[10];
strcpy(buf, name);
printf("Welcome %s\n"
```

  we have not any guarantee that *buf* is large enough to contain *name*.

- This may cause a *buffer overflow.*

Example: shellcode injection

- To exploit this vulnerability an attacker can:
  - overwrite value of register *eip* to refer to a memory area that he/she controls;
  - fill that area with a *shellcode*, namely a set of *assembly* instructions that spawn a *shell.*
- Tools are available to generate these codes like, for instance, the Python library *pwntools*.
- Shellcode is machine dependent!
- To perform this attack, stack must be *executable.*

Pwntools…

- Pwntools is a python framework for CTF designed for rapid prototyping and development.
- Pwntools provides libraries for:
  - Interacting with local and remote processes;
  - Packing data in byte format;
  - Assembly and disassembly;
  - Reading and manipulating ELF files.
- Pwntools is available at:
  https://github.com/Gallopsled/pwntools

Code Reuse Attacks

- Code-reuse attacks are software exploits in which an attacker directs control flow through existing code with a malicious result.
- Examples of such attacks are:
  - Return-to-libc
  - Return Oriented Programming (ROP)
  - Jump Oriented Programming (JOP)

Return to libc…

- A *return-to-libc* attack is a technique that, by using a *buffer overflow*, replaces a return address with the one of another function in the process memory.
- The name is related to the fact that one of functions in the *C Standard Library (libc) is used.*
- This attack was spotted in 1997 by *Alexander Peslyak* (aka *Solar Designer*).

Return to libc

- Let us consider again the following vulnerable code:

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

void welcome(char *name)
{
    char buf[10];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    welcome(argv[1]);
    return 0;
}
```


Return to libc

- By performing a *return-to-libc* attack, we can execute a *shell* with the same access rights of the executed program.
- The basic steps of the attack are:
  - Find the address of *system() libc* function;
  - Find the address of string "*/bin/sh*";
  - Corrupt the stack and let *system()* be called with parameter */bin/sh*.
- We do not need an *executable* stack!

Return to libc…

- To find address of *system()* we can use *gdb*:

```
(gdb) file retlibc
Reading symbols from retlibc...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x104b4
(gdb) run
Breakpoint 1, 0x000104b4 in main ()
(gdb) p system()
Too few arguments in function call.
(gdb) p system
$1 = {int (const char *)} 0xb6e909c8 <__libc_system>
```

Return to libc…

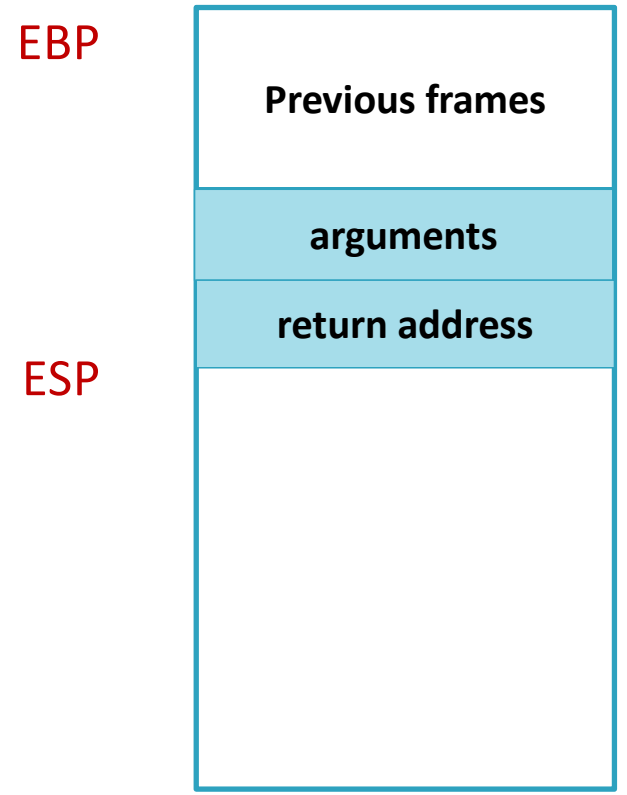
- A similar approach can be used to find address of */bin/sh:*

```
$1 = {int (const char *)} 0xb6e909c8 <__libc_system>
(gdb) find 0xb6e909c8, +999999999, "/bin/sh"
0xb6f83b6c
warning: Unable to access 16000 bytes of target memory at 0xb6f93574, halting se
arch.
1 pattern found.
(gdb)
```


Stack behaviour (recall)

- Each function starts with the following instructions (in assembly) with arguments and return address on top of the stack:

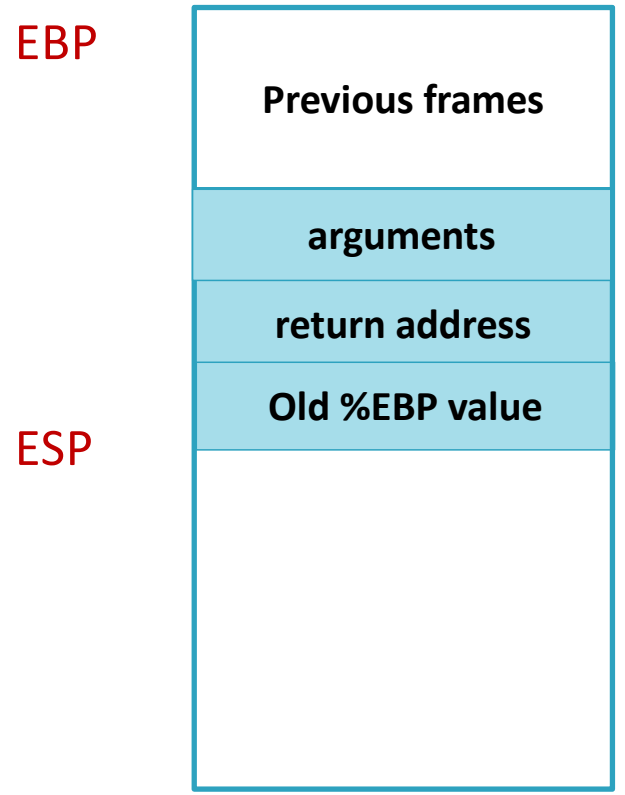
```
push %ebp
mov %esp,%ebp
sub $N,%esp
```



Stack behaviour (recall)

- First, previous value of %ebp is stored:

```
push %ebp
mov %esp,%ebp
sub $N,%esp
```

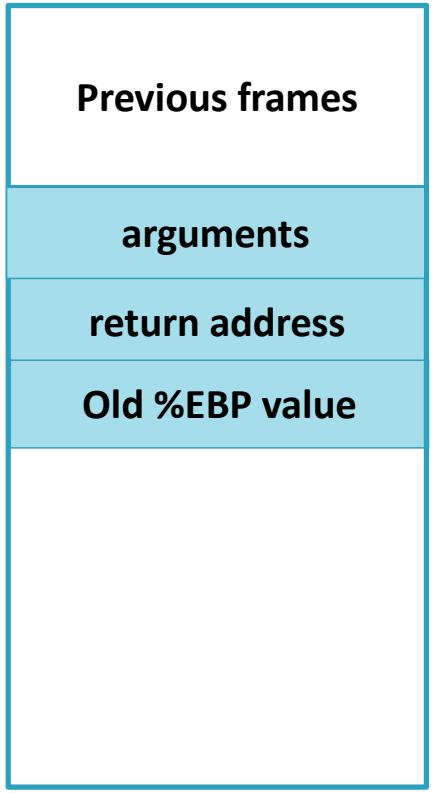


Stack behaviour (recall)

- After that, %ebp updated to refer to the new frame:

```
push %ebp
mov %esp,%ebp
sub $N,%esp
```

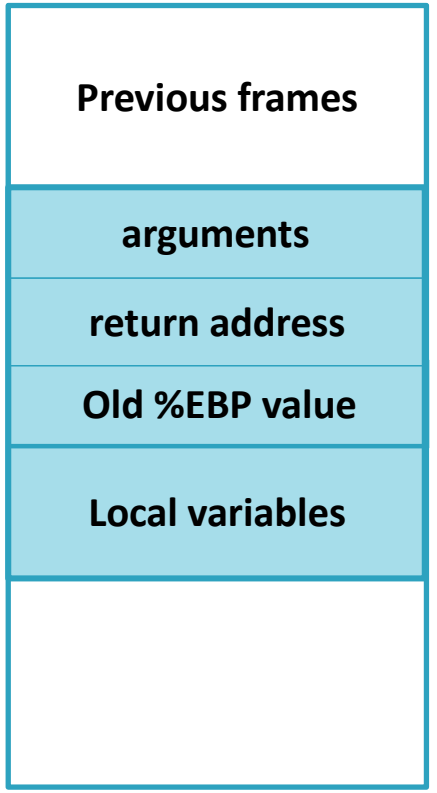
EBP ESP



Stack behaviour (recall)

- Finally, memory for local variables is allocated:

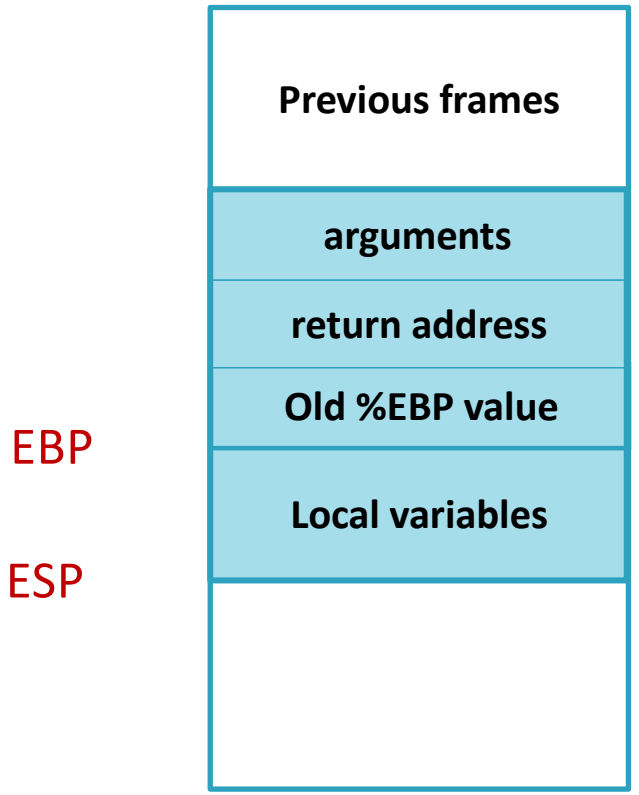
```
push %ebp
mov %esp,%ebp
sub $N,%esp
```



Stack behaviour (recall)

- When a function ends, the following instructions are executed:

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

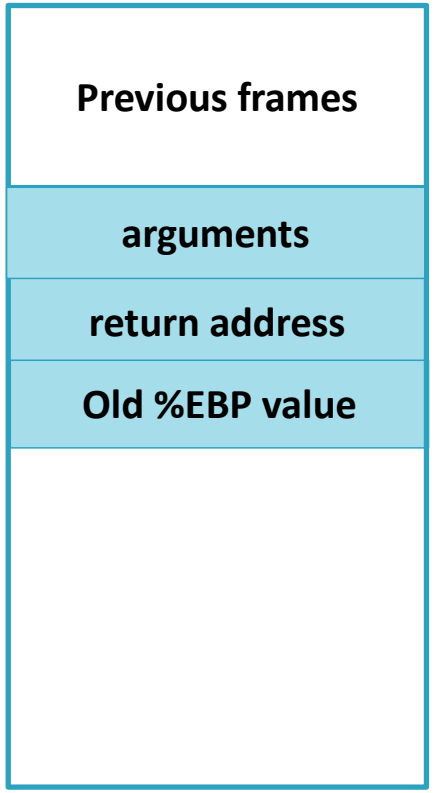


Stack behaviour (recall)

- Register %esp is set to %ebp (local variables are *lost*):

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

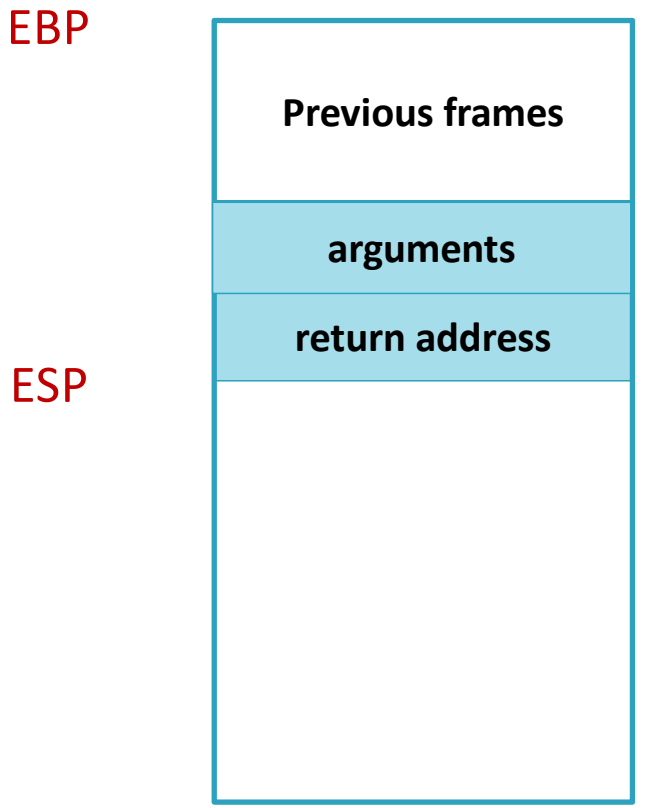
ESP EBP



Stack behaviour (recall)

- Previous value of %ebp is restored:

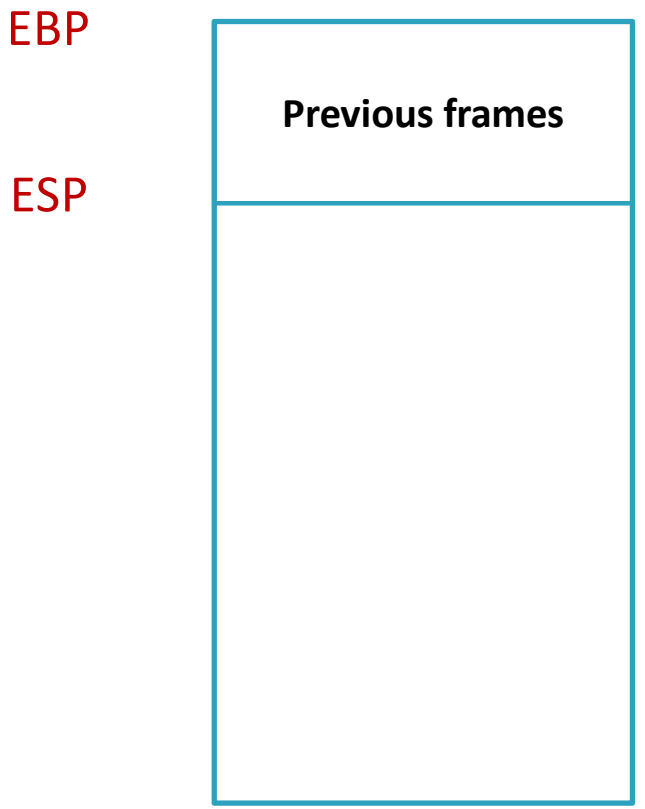
```
mov %ebp,%esp
pop %ebp
ret
```



Stack behaviour (recall)

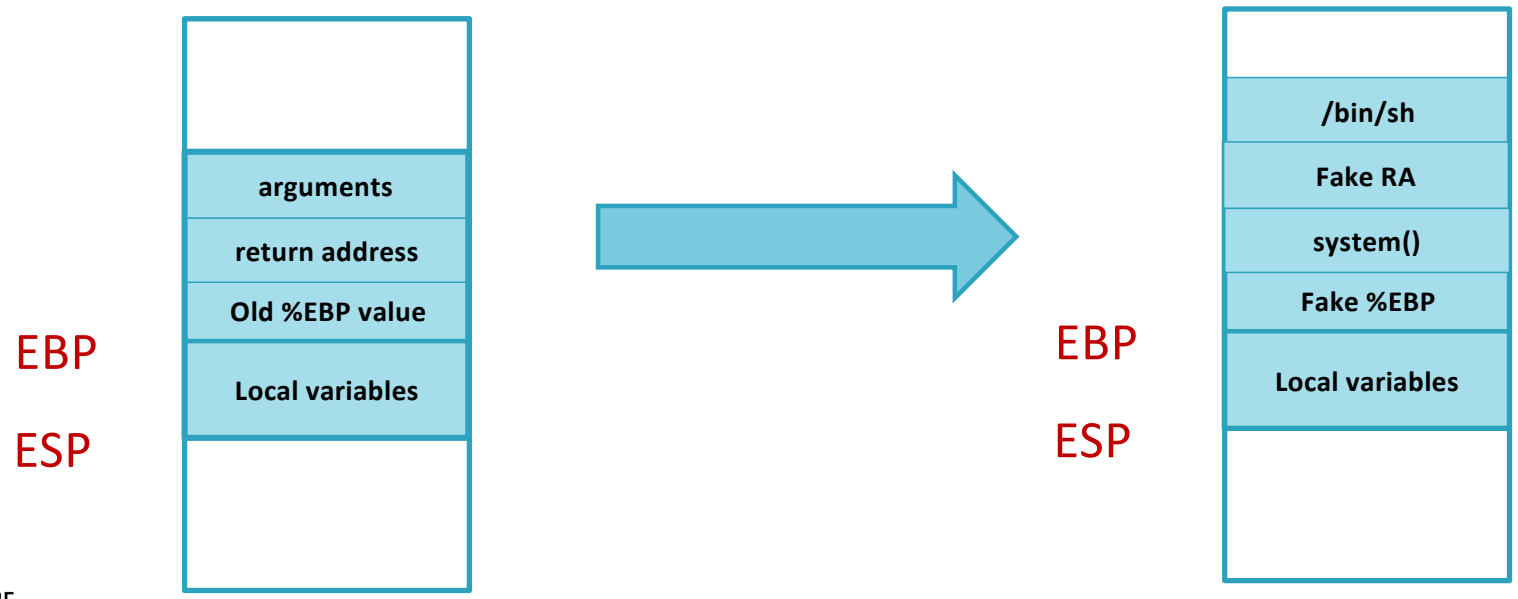
- Execution *returns* to the *return address*:

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



Return to libc in action…

- An attacker can corrupt the stack by inserting the addresses of *system()* and */bin/sh* in the *right place*:



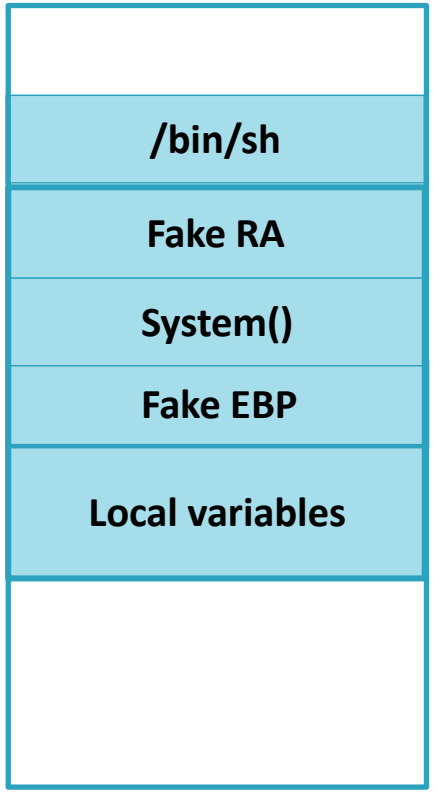
Return to libc in action…

- The executions of assembly instructions at the end of the vulnerable function will activate function *system()* with the *needed* argument!

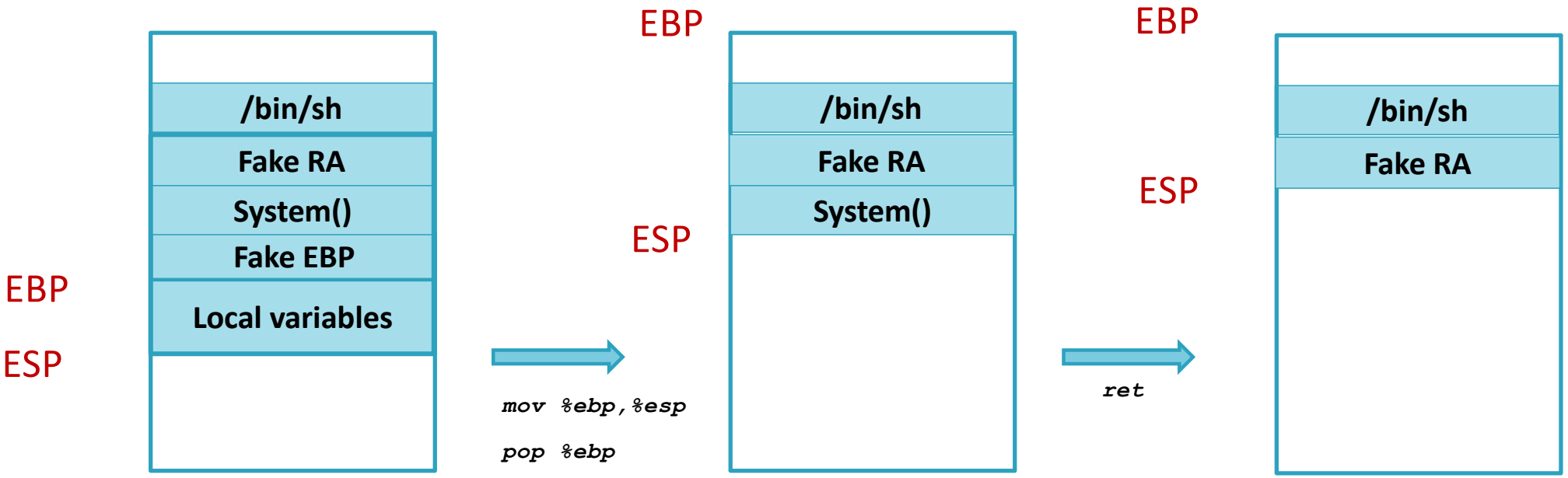
```
mov %ebp,%esp
pop %ebp
ret
```

EBP

ESP



Return to libc in action…



System(/bin/sh) is executed!

Return to libc in action…

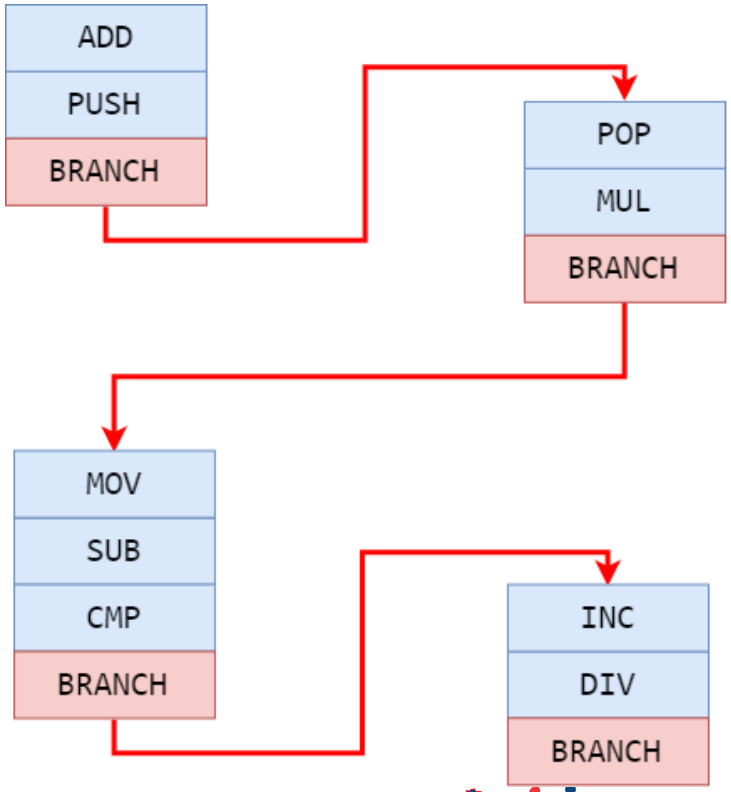
- To perform an attack we have to pass to a *vulnerable* program the following inputs:
  - A sequence of bytes long enough to trigger the buffer overflow (in our case 16)
  - The address of function *system()*
  - A sequence of 4 bytes for a return address
    - The address of function *exit() can be used.*
  - *The address of /bin/sh*

Return Oriented Programming (ROP)

- Return Oriented Programming (or ROP) is based on the idea of chaining together small snippets (or gadgets) of assembly with stack control to lead the program to do more complex things.
- The goal is to obtain a stack that leads to the execution of unwanted behavior.

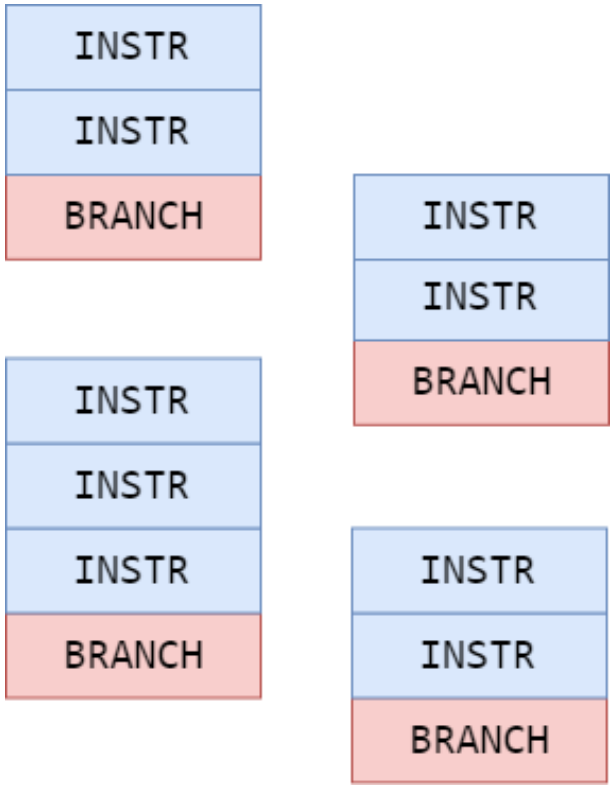
Return Oriented Programming (ROP)

- In ROP, an attacker chains pieces of existing code redirecting the program flow by changing the target address of branch instructions
- Concatenation of such opportunely chosen pieces leads to the execution of a malware
- Traditional code-injection defenses are bypassed



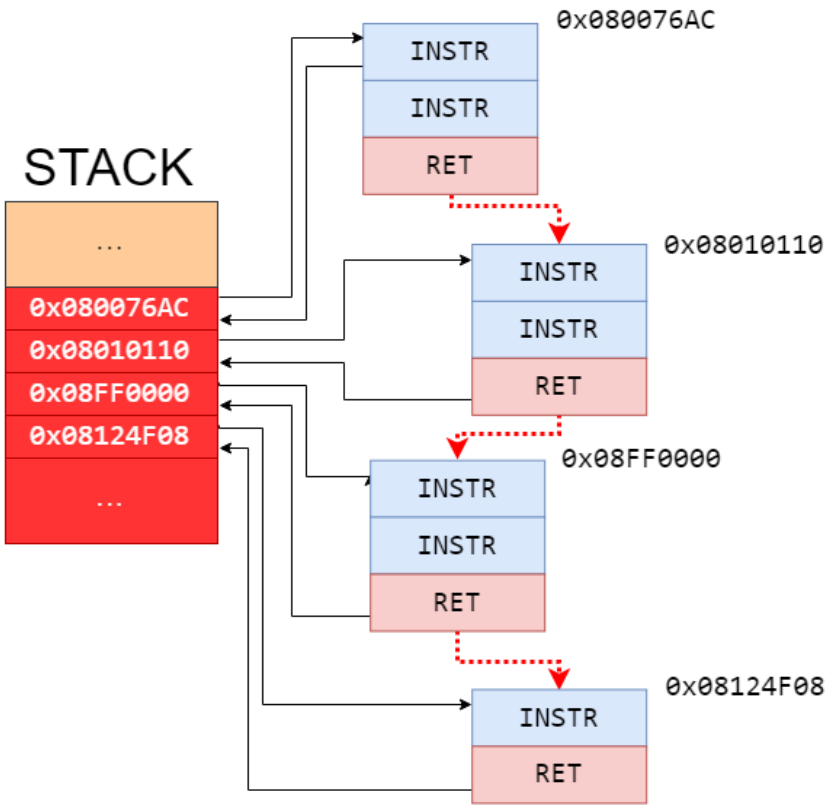
Gadgets

- In principle, any block of instructions that ends with a control-flow transfer
- Especially available in standard libraries of common programming languages (e.g., *libc*), which are compiled for every application
- Dimensions: typically 2 to 5 instructions to reach complete expressiveness



Return-Oriented Programming (ROP)

- Based on gadgets ending with a routine return instruction (RET)
- RET pops the return location from the stack and jumps there
- If the stack data are overflowed, a series of “ fake” return addresses can be stacked
- Every time a RET is executed, control is passed to the next gadget



ROP in action…

- To perform an attack based on ROP one has to:
  - Find the chain of gadgets that induces the expected behavior;
  - Store the gadgets on the stack
    - The value of EIP register must be overwritten with the address of the first gadget.

ROP in action…

- ***Question:*** how can we find such gadgets?
  - By hand, e.g., by inspecting *objdump* (this is the *old school approach*)
  - By using one of the available tools*:*
    - *Ropper (https://github.com/sashs/Ropper)*
    - *ROPGadget (https://github.com/JonathanSalwan/ROPgadget)*

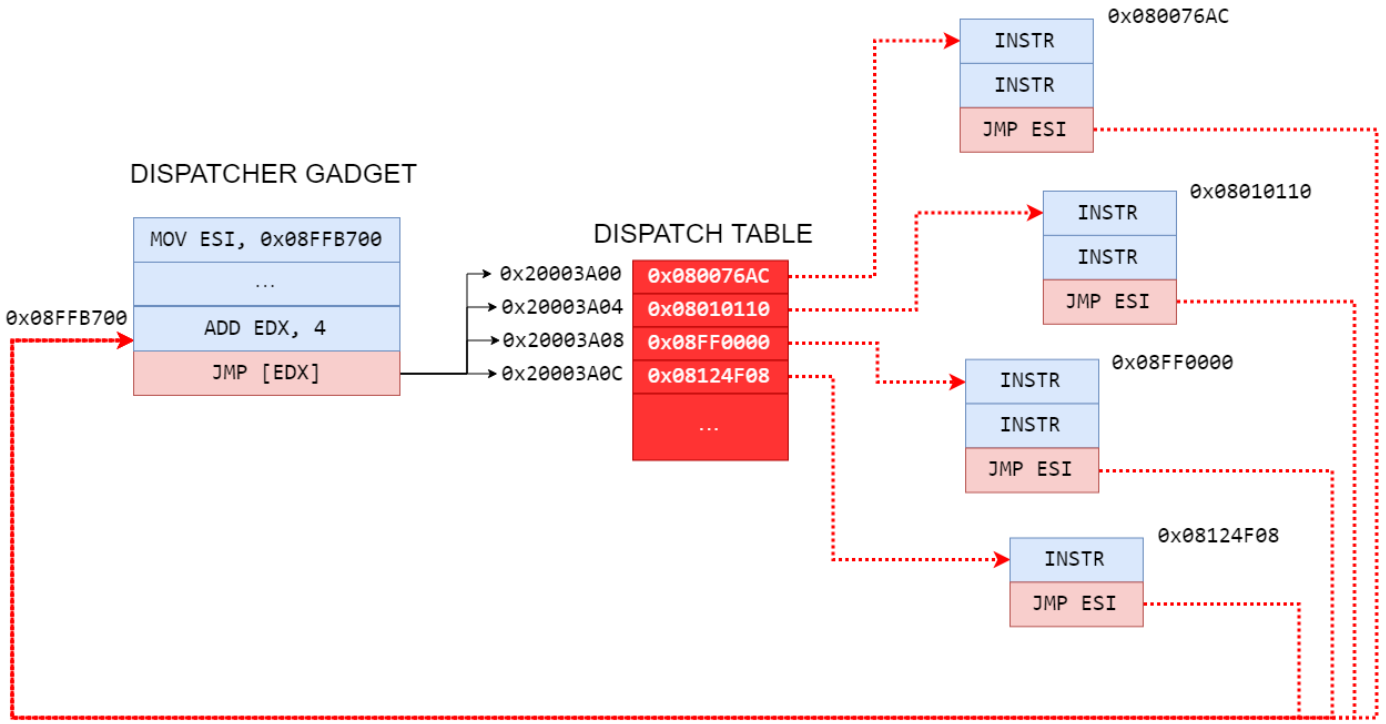
Jump-Oriented Programming (JOP)

- Jump-Oriented Programming (JOP) is a technique that triggers the execution of a given functions via a sequence of *indirect jump instructions.*
- Similarly to ROP, JOP is based on a sequence of small *gadgets…*
  - in ROP each gadget ends with a return instruction (ret);
  - in JOP each gadget ends with an unconditional jump instruction (jmp).

Jump-Oriented Programming (JOP)

- Gadget addresses are collected in a *dispatch table* located in any section of data memory containing a vulnerability (stack, but also heap, global, etc.)
- The final JMP of the gadget is redirected to the *dispatcher gadget*, which advances a pointer to the dispatcher table
- The dispatcher ends with an indirect jump to the address pointed by the pointer, in order to pass the control to the next gadget listed in the table

Jump-Oriented Programming (JOP)



Jump-oriented programming: a new class of code-reuse attack
Tyler Bletsch, Xuxian Jiang, Vincent W. Freeh, Zhenkai Liang
ACM ASIACCS'11

Conclusions

- We have discussed *Arbitrary Code Execution*
  - Code injection
  - Code reuse
    - Return to libc
    - Return Oriented Programming (ROP)
    - Jump Oriented Programming (JOP)

Software Security 3

Code Reuse Attacks

Michele LORETI
Univ. di Camerino
michele.loreti@unicam.it

CYBER CHALLENGE
CyberChallenge.IT

cini Cybersecurity National Lab

https://cybersecnatlab.it

CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

SPONSOR PLATINUM

accenture

aizoon AUSTRALIA EUROPE USA
TECHNOLOGY CONSULTING

B5

eni

exprivia | ITALTEL

IBM

KPMG

LEONARDO

NTT DATA
Trusted Global Innovator

NUMERA
SISTEMI E INFORMATICA S.p.A.

Telsy

SPONSOR GOLD

bip.

cisco

MONTE DEI PASCHI DI SIENA
BANCA DAL 1472

negg

NOVANEXT
connecting the future

pwc

SPONSOR SILVER

DGi ONE
the leading digital company

ICT CYBER CONSULTING