CSIT 5740 Introduction to Software Security

Midterm revision

Dr. Alex LAM

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

The set of note is adopted and converted from a software security course at the Purdue University by Prof. Antonio Bianchi

Disclaimer

- This is a really a brief revision of some of the important concepts. It is by far complete! Please refer to the materials at the course web for all the details

Threat Model

- Asking “is this secure?” is a “bad” question.

- What can our attackers do? What we consider as possible?
  - Remote Attacker, Local Attacker, Attacker in Physical Proximity,Your Friend, Evil Maid, Hacker, Malware, Foreign Government, …

- What do we consider as “bad”?

The CIA triad and more

- CIA triad
  - Confidentiality
    - It refers to the need to protect information from unauthorized accesses, so that the data is kept safe and secret.
  - Integrity
    - It refers to the need to make sure data is trustworthy, complete and has not been modified intentionally or unintentionally by unauthorized users
  - Availability
    - It refers to the need to ensure the service/data is available when you need them

- Other properties
  - Authenticity
    - We know the origin of some data
  - Privacy

The Linux System

Users: Sudoers

- Normal users have an associated password
- Typically asked for “login”
  - System’s startup
  - Remote login (ssh)

- Users belonging to the group sudo can switch to another user (root) using the sudo command

- By default, it requires inserting the user’s password

Permission Checking

- In general, to perform system-level operations a process invokes kernel code, by using system calls

- The called kernel code checks
  - Which process called it
  - The owner of the process
  - Is the owner of the process authorized to perform the requested operation?
    - Some operations may be restricted only to:
    - Users belonging to specific groups
    - The root user
    - ...

File Permissions

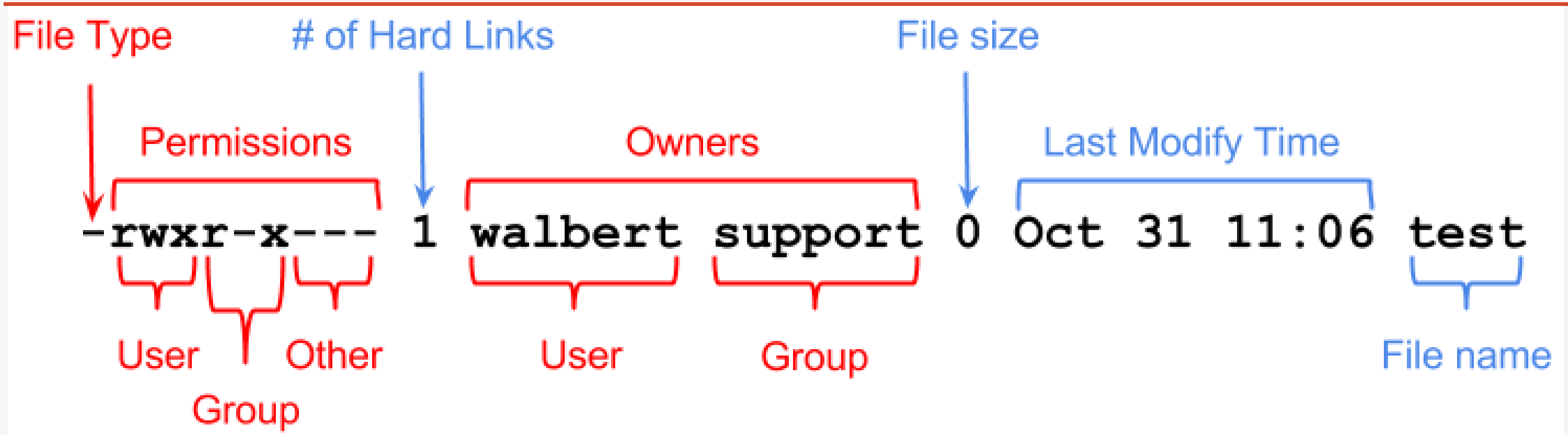
- Every file has an associated owner user and an associated owner group
  - Use: `ls -al`

    to list files and their owners

  - `chown`: change owner user/group

- Every file has permissions associated to it
  - `chmod`: change file permissions

File Permissions



Directory Permissions

Read permission

`-`: The directory's contents cannot be shown.

`r`: The directory's contents can be shown (listed).

Write permission

`-`: The directory's contents (the list of files) cannot be modified.

`w`: The directory's contents can be modified (create new files or folders, rename or delete existing files, ...). It requires the execute permission, otherwise it has no effect

Directory Permissions

Execute permission

`-`: The directory cannot be accessed (cannot cd inside the directory)

`x`: The directory can be accessed using cd

(this is the only permission bit that in practice can be considered to be "inherited" from the ancestor directories, in fact if *any* folder in the path does not have the x bit set, the final file or folder cannot be accessed either)

→ You can create directories from which a user can read files, but you cannot list them → remove the “`r`” permission, but keep the “`x`” permission

Directory Permissions

Sticky Bit

- files in the directory may only be deleted or renamed by
  - root
  - the directory owner
  - the file owner.

→ *Typically used for* $/tmp$

Every user can add files to $/tmp$ or list the content of $/tmp$, but a user cannot delete/rename other users' files in $/tmp$

```
drwxrwxrwt  13 root root        4096 Oct 29 03:34 tmp
```

Links

Hard links

- The file system interprets two paths as “the same file”
- When you delete a file that was hard linked to the data, the data is still on the harddrive as long as there are still hard-linked files existing. The data is deleted only when all the hard-linked files are deleted

Soft links

- More used, a link is a special file “pointing to” another file

```
ln -s <target> <link_name>

ls -la
```

- When the source file is deleted, the soft link is still there, though it will be pointing to an invalid file.

Links: Security Concerns

Symbolic Links can be used to make exploitations easier (**refer to note set 2, slides 21-31 for the details**).

For instance (will illustrate using a real example):

- An application runs as root and takes a `<filename>` as the argument, this `<filename>` could be in a world-readable and/or world-writeable folder (`/tmp` or /`var`/`log` for example)
- An attacker could create a link from `<filename>` to a strategic file (`/etc/shadow`, `/root/.rhosts` etc), tricking the app to make the file world-readable or even world-writable
- Even if the application checks if `<filename>` is not a link, the attacker can create a link just after the check, but before the application changes `<filename>` file permissions (more on that)

Links: Security Concerns

Consider the program fragment being run by the superuser `root` at the `/tmp` directory

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1],O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1],S_IRWXU|S_IRWXG|S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```

Links: Security Concerns

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1],O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1],S_IRWXU|S_IRWXG|S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```

Links: Security Concerns

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1],O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1],S_IRWXU|S_IRWXG|S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```

Links: Security Concerns

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1],O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1],S_IRWXU|S_IRWXG|S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```

Links: Security Concerns

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char ** argv) {
    // before we open the file, we do not check if
    // it is an existing file!

    // when we open the file with "O_CREAT" flag,
    // open will follow the link (if it is a symbolic link)
    // and create the file in the target directory!
    int file = open(argv[1],O_CREAT);

    // this is an innocent log file being opened, should allow everybody to do whatever thing on it
    chmod(argv[1],S_IRWXU|S_IRWXG|S_IRWXO);

    ::: // main part of the program not stated,
        // remove the above line if you want the program to run

    close(file);
}
```

This file is an innocent log file and in the /tmp directory should be accessible to everybody. S_IRWXU, S_IRWXG and S_IRWXO means we grant the **full set of RWX** rights on the file to the owner, group and other users!

Links: Security Concerns

- How we can make use of this buggy program run by the `root` to exploit the Linux system?
  - Scenario 1: a malicious user “alex” wants to read the password stored at `/etc/shadow` , but he does not have the permission
  - Scenario 2: a malicious user “alex” wants to remotely login the machine as the user “Kali”

Links: Security Concerns

- For scenario 1, "alex" just need to create a symbolic link to point to `/etc/shadow`, then wait quietly for the program to start and change the permission of `/etc/shadow` , then he can read and launch a dictionary attack on the password

- Alex creates a symbolic link from a file called "`log`" to "`/etc/shadow`":

```
┌──(alex㉿kali)-[/tmp]
└─$ ln -s /etc/shadow log
```

- Alex patiently waits for the root to start the program and run it on the "`log`" file, at that point is still unreadable for Alex:

```
└─# ls -al /etc/shadow
-rw-r----- 1 root root 1786 Sep  3 04:42 /etc/shadow
```

Links: Security Concerns

- For scenario 1, “alex” just need to create a symbolic link to point to `/etc/shadow`, then wait quietly for the program to start and change the permission of `/etc/shadow` , then he can read and launch a dictionary attack on the password

- The `root` executes the program from the `/tmp` directory, as a regular daily task

```
┌──(root💀kali)-[/tmp]
└─# ./symlink_vuln log
```

- Look at the permissions for “other users” (Alex included)!

```
└─# ls -al /etc/shadow
-rwxrwxrwx 1 root root 1786 Sep  3 04:42 /etc/shadow
```

Links: Security Concerns

- For scenario 2, “alex” just need to create a symbolic link to point to `/home/kali/.rhosts`. “`.rhosts`” is the file containing the list of remote hosts that are allowed to connect to the current host without the need of supplying a password! The symbolic link can be created even if the target is not existing!!

- Alex creates a symbolic link to “`.rhosts`”,

```
┌──(alex㉿kali)-[/tmp]
└─$ ln -s /home/kali/.rhosts log
```

- The file doesn’t even exist!

```
└─# ls -al /home/kali/.rhosts
ls: cannot access '/home/kali/.rhosts': No such file or directory
```

Links: Security Concerns

- For scenario 2, “alex” just need to create a symbolic link to point to `/home/kali/.rhosts`. “`.rhosts`” is the file containing the list of remote hosts that are allowed to connect to the current host without the need of supplying a password! The symbolic link can be created even if the target is not existing!!

- Again Alex waits patiently for the unalarmed `root` to run the buggy program

```
┌──(root💀kali)-[/tmp]
└─# ./symlink_vuln log
```

- The file is created, granting a full set of rights to everybody! Can you imagine how happy Alex is? He can add his host to login without the need of password.

```
┌──(root💀kali)-[/tmp]
└─# ls -al /home/kali/.rhosts
-rwxrwxrwx 1 root root 0 Sep 10 03:52 /home/kali/.rhosts
```

Links: Security Concerns

- What we can do to mitigate that? Do the following as the root:

```
┌──(root💀kali)-[/tmp]
└─# echo 1 >  /proc/sys/fs/protected_symlinks
```

- When `protected_symlinks` is 1, if the sticky bit is set on a world-writeable directory (`/tmp`), Linux will allow the program to follow a symbolic link to the target if:
  - the process `uid` is the same as the symlink `uid`
  - or the symlink and the `/tmp` directory have the same owner (i.e. usually the `root`)

Links: Solutions

- When `protected_symlinks` is 1, if the sticky bit is set on a world-writeable directory (`/tmp` , "world-writeable" so that Alex can put a symlink file there), Linux will allow the program to follow a symbolic link to the target if :
  - **either** the process `uid` is the same as the symlink `uid`
  - **Or** the symlink and the directory (i.e. `/tmp` here) have the same owner (i.e. usually the `root`)
- That will solve scenario 1 , and Alex will not be able to change the permissions of "`/etc/shadow`" with his symlink,
  - because the process `uid` is root, symlink uid is `alex`, mind that though `alex` can start a process himself to access the symlink, but the process may not have permission to do operations on the target
  - the symlink owner is `alex`, but the `/tmp` directory belongs to `root`
- That will also solve scenario 2 , and Alex will not be able to create the `.rhosts` file

The Linux setuid

setuid

- More precisely, every process has two properties:
    - **real user ID**
        - the user who started a process
    - **effective user ID**
        - the user used by the OS to determine what a process can/cannot do

setuid

- Normally, when a process creates a new process (i.e., a child process) the child process **keeps the same real user ID and effective user ID** of its parent process
  - New processes are normally created with the `fork/execv` system calls. Don’t worry too much about “system calls” if you don’t know about them, we will discuss them in 2-3 weeks.
  - This is what normally happens when you use a shell (e.g., `bash`) to run a command
  - `bash` runs with real/effective user ID of the logged in user
    - Processes created using the shell have the same real/effective user ID of the user using the shell

setuid

- However
  - if a program is stored with the **setuid bit set**
  - when it runs, the effective user ID of the executed program is equal to the owner of the file
  - the real user ID is unchanged

setuid

- This mechanism allows users to perform privileged operations by using setuid programs, since the code of a setuid program runs with:
  - Effective user ID = Owner of the file (typically root)
  - Instead of:

    Effective user ID = Parent process effective user ID (typically the current user)

The Attacks to Linux

Path Traversal Attack

- An application builds a path by concatenating a path prefix with values provided by the user (the attacker)

```
path = strncat("/var/log/app/", user_file, free_size);
file = open(path, O_RDWR);
```

- The user (attacker) provides a filename containing a number of: `"../"` that allow for escaping from the directory and access any file on the file system

- user_file == "`../../../etc/shadow`"
⇒ path = "`/var/log/app/../../../etc/shadow`" ⇒ opens "`/etc/shadow`"

Lessons Learned

- Input provided by the user should be sanitized before being used in creating a path

Some useful functions for sanitizing the path inputs			
Linux	*Windows*	*Java*	*Python*
`realpath()` `canonicalize_file_name()`	`PathCanonicalize()`	`getAbsolutePath()` `getCanonicalPath()`	`os.path.abspath()` `os.path.realpath()`

TOCTTOU Attacks

- Attacker may race against the application by exploiting the gap **between testing and accessing** an property
  - **T**ime-**O**f-**C**heck-**T**o-**T**ime-**O**f-**U**se
  - e.g., testing a file property and then accessing it

- Time-Of-Check (t1): validity of assumption A on entity E is checked
- Time-Of-Use (t2): E is used, assuming A is still valid
- Time-Of-Attack (t3): assumption A is invalidated by an attacker, but the attacker can still access E, as long as:
  - **t1 < t3 < t2**

- Data race condition

TOCTTOU Example

- A SETUID program may want to avoid accessing a file if its real user (real UID) is not allowed to access it
- The `access()` system call returns an estimation of the access rights of the user specified by the real UID
- The `open()` system call checks permissions using the effective UID

```
if(access(file, W_OK) == 0) { //time of check t1

   //an attacker replaces file with a symlink to the file /etc/shadow t3
   // i.e.  symlink("etc/shadow",file);
   // the line below , it opens /etc/shadow using the root privilege

  if ((fd = open(filename, O_WRONLY)) < 0){ //time of use t2
      return -1;
  }

  write(fd, buf, count);
}
```

TOCTTOU Example: same code as previous slide

- The `access()` system call returns an estimation of the access rights of the user specified by the real UID
- The `open()` system call checks permissions using the effective UID

Victim

```
if(access(file, W_OK) == 0) {



 if ((fd = open(filename, O_WRONLY)) < 0){
    :    :
  write(fd, buf, count);
}
```

t1

t3

t2

time

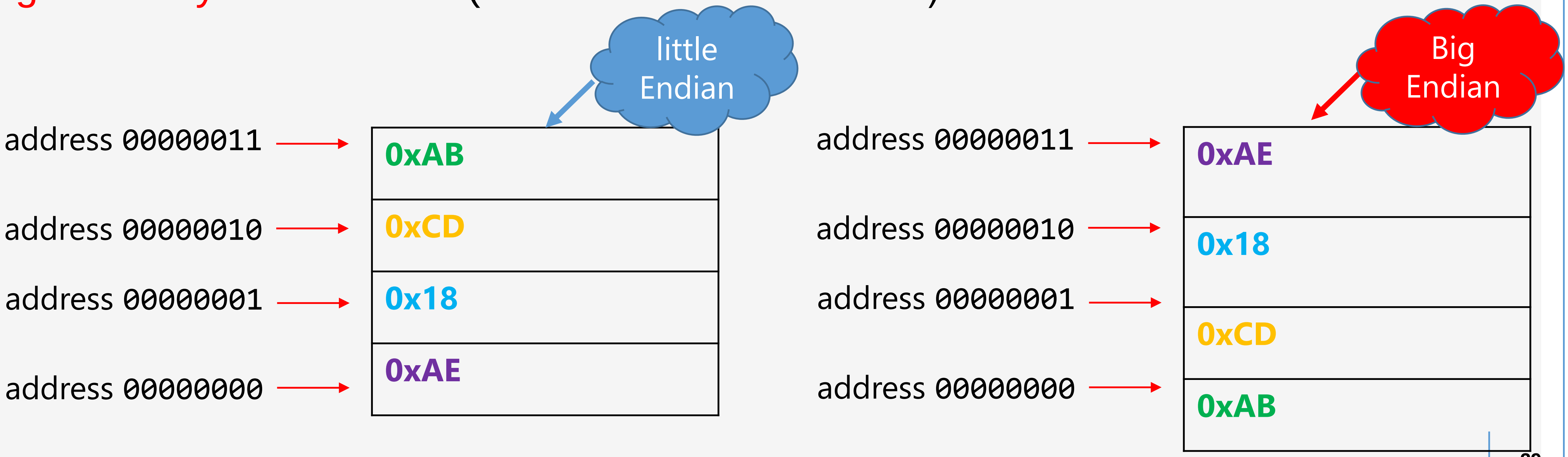
Attacker

```
symlink("/etc/shadow","foo");
```

The x86/x64 assembly

Data representation

- The data we would like to store is typically bigger than 1 byte, and will take multiple memory slots to store (each memory slot can store 1 byte only)
- The data/information represented by 0xABCD18AE could be a number, 4 characters, an instruction, etc. It will be stored to the memory occupying 4 slots
- This multi-byte data could be stored in one of the following ways. Mind that the rightmost byte of the data (i.e. AE of 0xABCD18AE) is called the "end":



Data representation

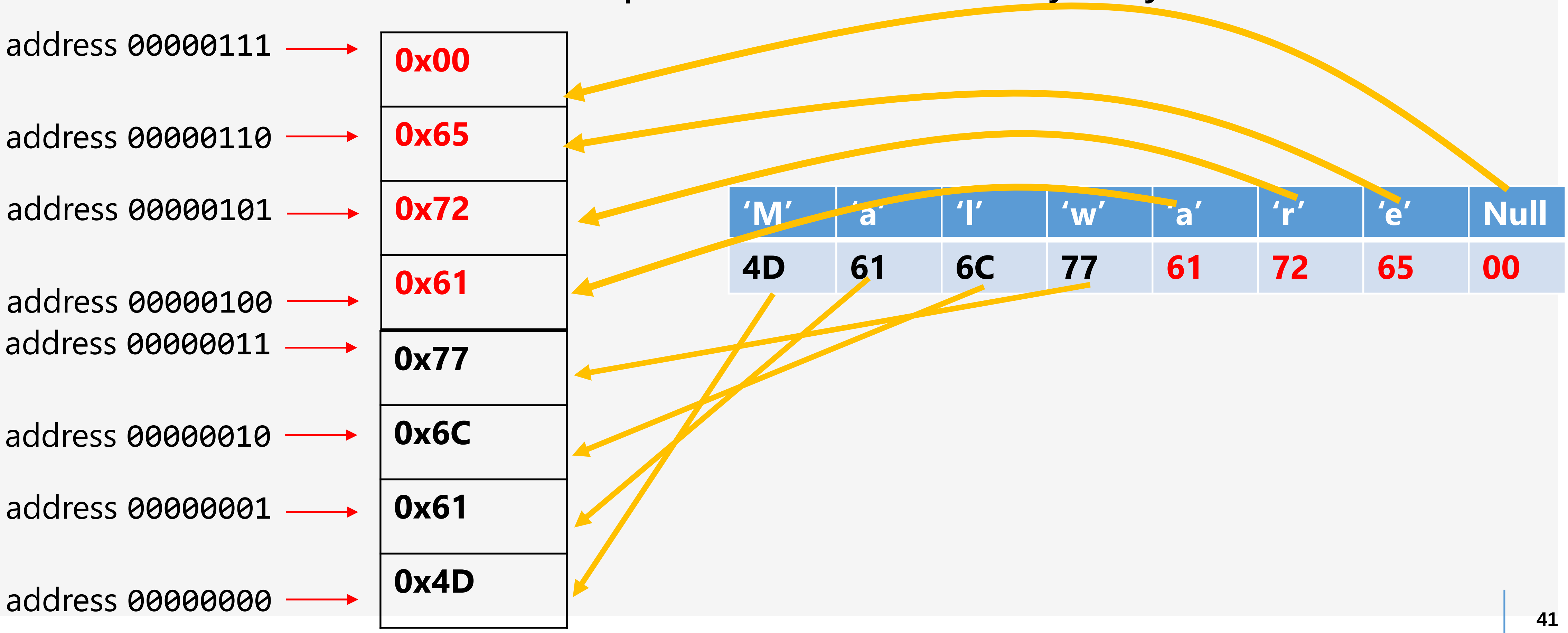
- **Endianness is needed only when each piece of data to be stored is bigger than 1 byte**.
- How about the string "Malware" ?
- The string "Malware" is considered to be **8 pieces of data**
- Each piece of data is a character. The character is encoded in the ASCII scheme and the encoded character is exactly 1 byte.
- The computer will put the characters one after the other, from character number 0 at the lowest memory address, to character number 7 at the highest memory address. There is **no need to consider endianness** here (more on this in 3 slides).

Character index	0	1	2	3	4	5	6	7
	M	a	l	w	a	r	e	\0

Mind this end of string special character

Data representation

- The computer will put the characters one after the other, from character number 0 to character number 7 to the memory. There is **no need to consider endianness** here. Because each piece of data is exactly 1 byte.



X86 Registers (General Purpose)

There are 16 general purpose registers in the x86 a

Size (in Bits)			
64	32	16	8
RAX	EAX	AX	AH/AL
RBX	EBX	BX	BH/BL
RCX	ECX	CX	CH/CL
RDX	EDX	DX	DH/DL
RDI	EDI	DI	DIL
RSI	ESI	SI	SIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8~R15	R8D~R15D	R8W~R15W	R8L~R15L

There following are two important registers in the x86 architecture, they are consider “general purpose” because you can use them in any instruction, but they serve important functions and can not be used for storing arbitrary data

- **rbp**: base/frame pointer
- **rsp**: stack pointer

- And then there is the **rip special purpose register** (which is equivalent to programmer counter) that points to the instruction being executed

X86 Registers (General Purpose)

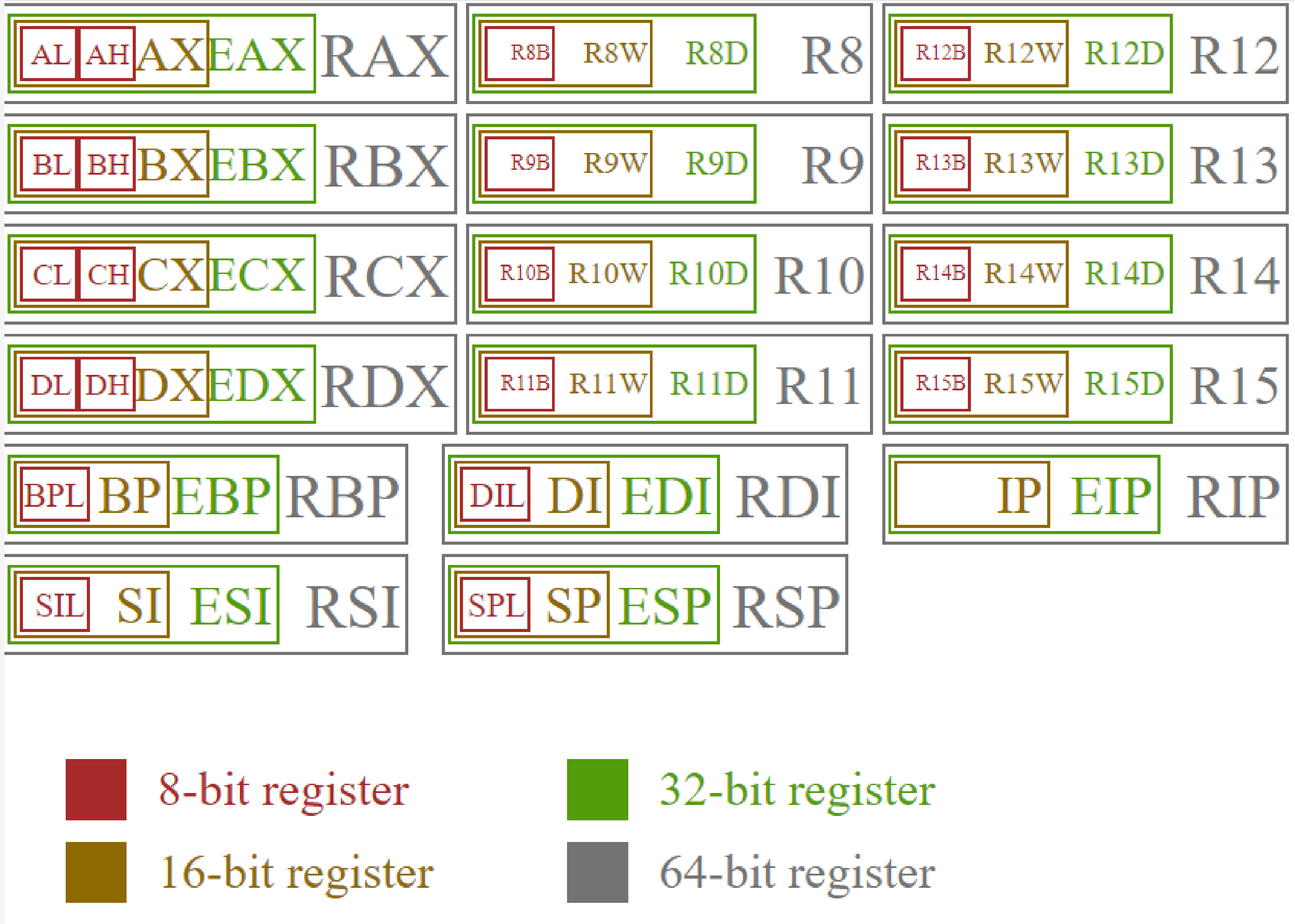
There are 16 general purpose registers in the x86 a

Size (in Bits)			
64	32	16	8
RAX	EAX	AX	AH/AL
RBX	EBX	BX	BH/BL
RCX	ECX	CX	CH/CL
RDX	EDX	DX	DH/DL
RDI	EDI	DI	DIL
RSI	ESI	SI	SIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8~R15	R8D~R15D	R8W~R15W	R8L~R15L

The registers nelow were originally envisioned for the following purposes, but now they could be used for any data

- **rax**: accumulator
- **rbx**: base index (for arrays)
- **rcx**: counter (for loops and strings)
- **rdx**: extend the precision of the accumulator
- **rdi**: destination index
- **rsi**: source index for string operations

X86 Registers (General Purpose)



CPU Registers (64-bit General Regs)

- The following extra registers are only in 64-bit mode
  - R8, R9, R10, R11, R12, R13, R14, R15

- The corresponding 32-bit registers are (lower 32 bits)
  - R8D,R9D,R10D,R11D, R12D, R13D, R14D, R15D

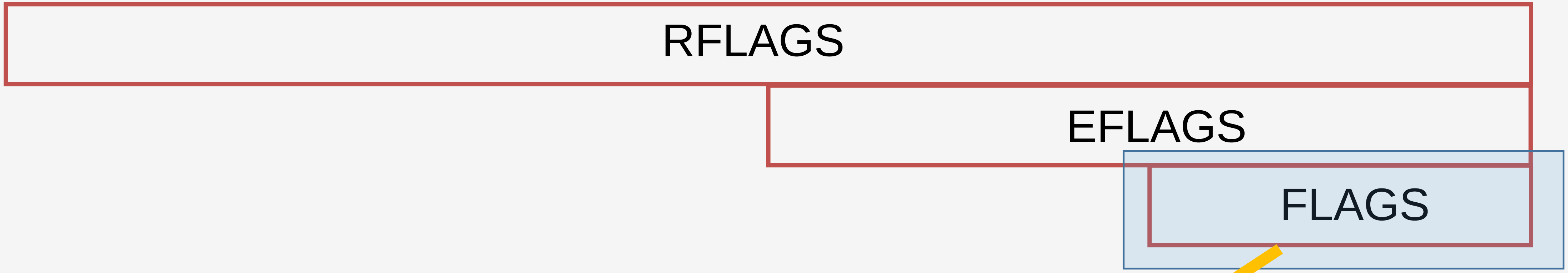
- The corresponding 16-bit registers are (lower 16 bits)
  - R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W

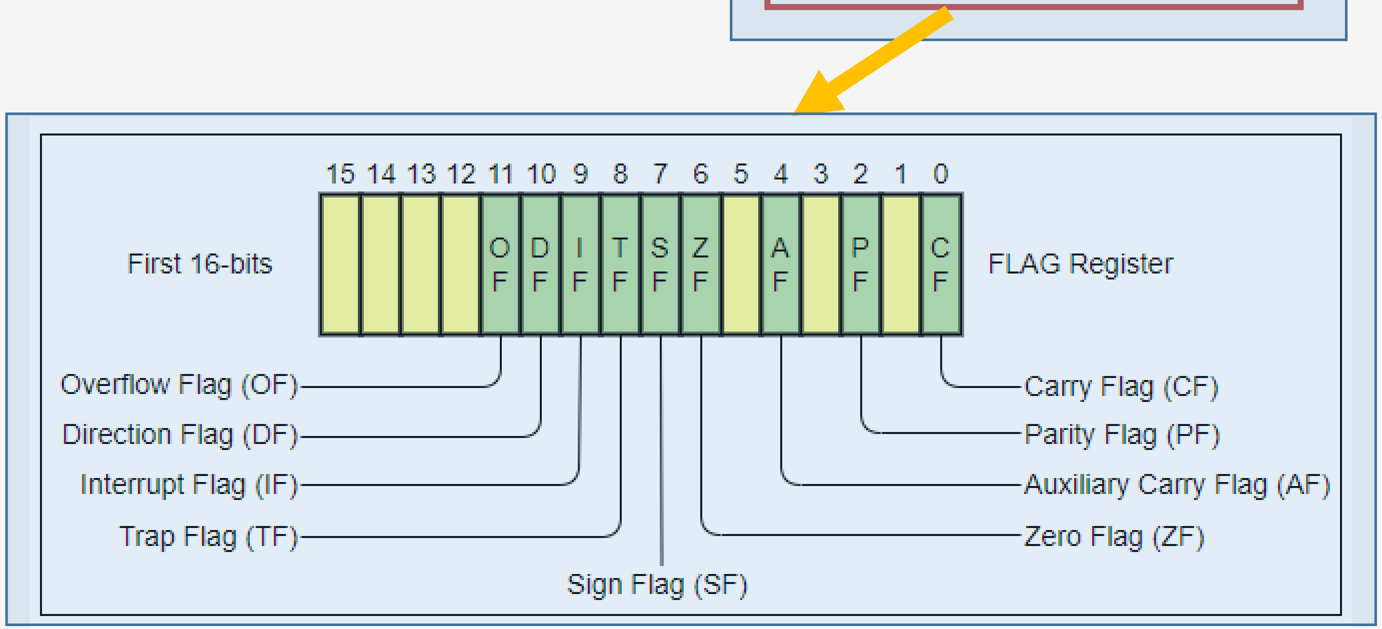
CPU Special Registers (review)

- SP/ESP/RSP: Stack pointer
  Decremented/incremented by push/pop instructions
- BP/EBP/RBP: Stack base pointer (frame pointer)
  Used to keep track of the stack pointer value when a function starts
- IP/EIP/RIP: Instruction pointer
  Points to the next instruction to be executed
- SI/ESI/RSI: Typically used as source index for string operations
  DI/EDI/RDI: Typically used as destination index for string operations

CPU Special Registers: Flags

- The Flags register is a special register to show the x86 CPU status





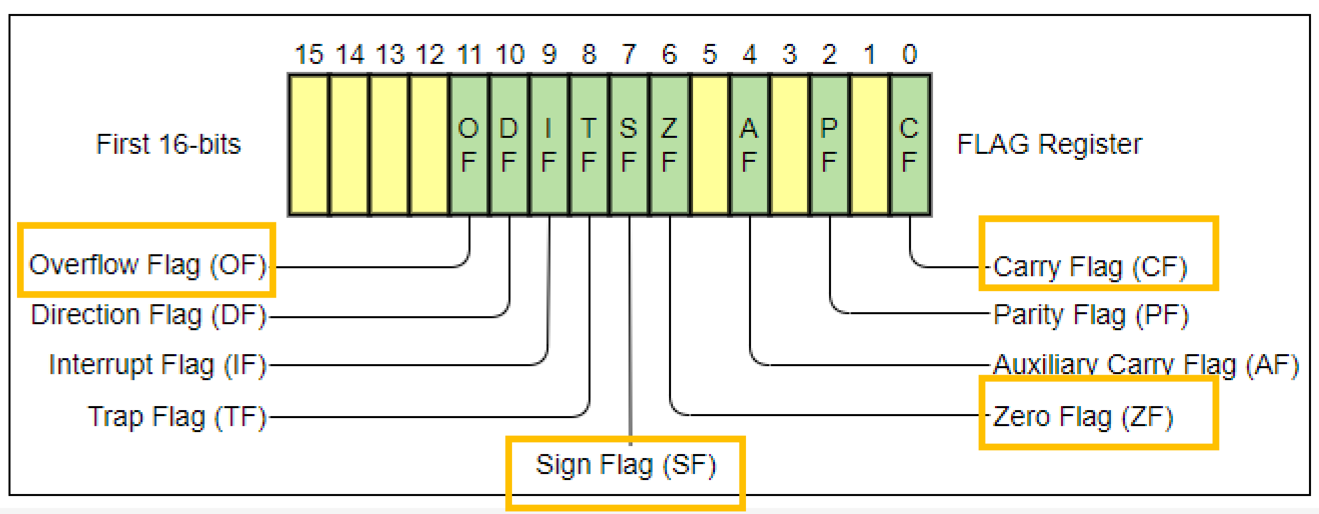
CPU Special Registers: Flags

- Typically we only need to know the FLAGS register which is 16-bit in size as shown below:

- These flags are of interest
  - CF: Carry flag

    Whether arithmetic carry or borrow has been generated out of the most significant bit. If a carry or borrow has been generated out of the most significant bit, this flag will be 1. Otherwise it will be 0.
  - ZF: Zero flag

    Whether an arithmetic result is zero



The flags are automatically set by mov, cmp, and other instructions, used to determine jumps

CPU Special Registers: Flags

- Typically we only need to know the FLAGS register which is 16-bit in size as shown below:

- These flags are of interest
  - SF: Sign flag

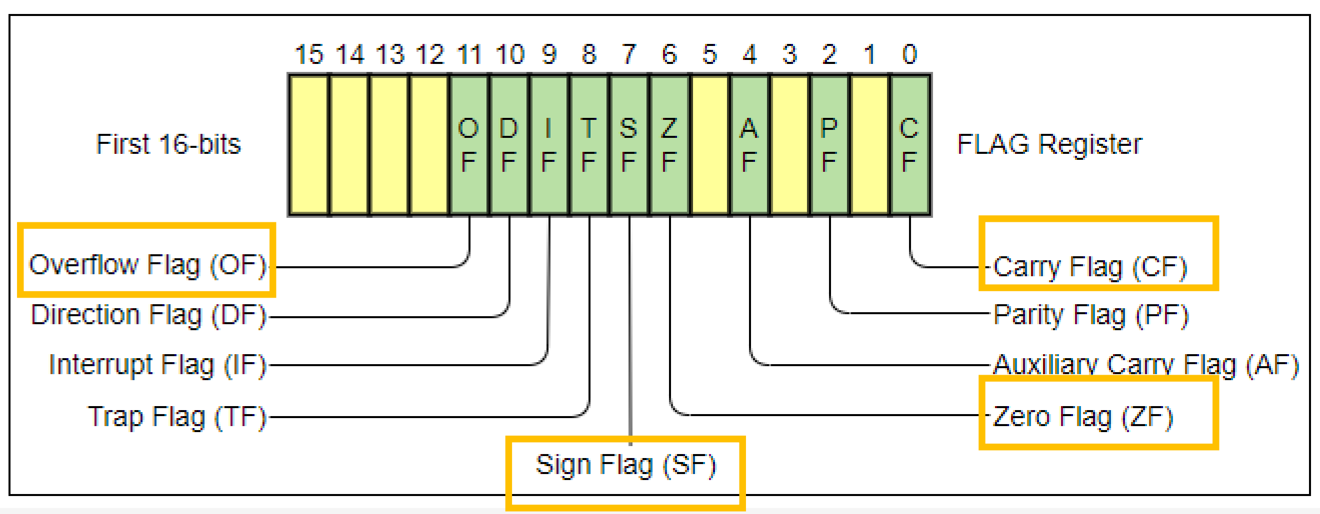
    Whether the result of the last mathematical operation produced a value in which the most significant bit is 1 (indicating the result is negative).

  - OF: Overflow flag

    Detects only signed overflow:

    1) positive + positive but result is negative

    2) negative + negative but result is positive



The flags are automatically set by mov, cmp, and other instructions, used to determine jumps

CPU Special Registers: Flags

- In general for the compare (`cmp`) instruction

cmp dest, src:

If dest < src, then the flags will be **ZF = 0**, **CF = 1**
If dest == src, then the flags will be **ZF = 1**, **CF = 0**
If dest > src, then the flags will be **ZF = 0**, **CF = 0**

Jump instructions can test ZF and CF to decide whether to jump to another place or to keep execute the instructions sequentially.

The "mov" instruction

`mov eax, 5` → copies 5 into eax

`mov eax, ebx` → **copies** value of ebx into eax

`mov eax, dword ptr [ebp - 8]` → copies the contents of the memory pointed by ebp - 8 into eax

the **`ptr`** keyword here indicates we are pointing to memory

`dword` indicates we are copying a dword (32-bit data size)

`mov eax, dword ptr [eax]` → copies the contents of the memory pointed by eax to eax

`mov dword ptr [edx + ecx * 2], eax` → moves the contents of eax into the memory at address edx + ecx * 2

`mov ebx, 804a0e4h` → copies the value 804a0e4h into ebx

`mov eax, dword ptr [804a0e4h]` → copies a dword from address 804a0e4h into eax

1. No 2 memory accesses in the same instruction! (i.e. you can not read a data from memory and then copy it also to the memory)
2. Width: size of reference (byte, word, dword, qword → 8, 16, 32, 64 bits)

The “lea” instruction

- **`lea`** is known as the “load effective address” instruction

- It loads/copies an address calculated into a register

- For example “**`lea rax, [rip+0x2ec6]`**”
  - It is equivalent to the operation `rax = value_of(rip)+0x2ec6`
  - `rip` is a 64-bit register here
  - Suppose rip is holding 0x0000787FFFFF0000, then the instructions puts 0x0000787FFFFF0000+0x2ec6=0x0000787FFFFF2EC6 into `rax`

The “push” instruction

- The **`push`** instruction “pushes/stores” a piece of data to the stack and decreases the stack pointer
- **`push <register>`** → decreases the stack pointer (esp/rsp) and saves the content of **`<register>`** in the newly pointed location
- For example “**`push rax`**”
    - Will decrease the stack pointer rsp by 8 to allocate 8 bytes of new space on the stack
    - Then it will put the 8-byte rax regiser value into the newly allocated 8-byte space on the stack
    - more on this with the help of a picture
    - “**`push eax`**” is the 32-bit version, it will decrease `esp` by 4 and put the 4-byte `eax` content to the stack

The “pop” instruction

- The **pop** instruction “pops/retrieves” a piece of data from the stack and increases the stack pointer
- **pop <register>** → retrieves the last piece of data (i.e. top) from the stack and stores it to **<register>**, then it increases the stack pointer (`esp`/`rsp`) to de-locate the data from the stack (i.e. the data will be out of the stack boundary)
- For example “**pop rax**”
    - Will copy 8-byte of data on the top of the stack to the 64-bit register
    - Then it will increase the stack pointer `rsp` by 8 to de-locate 8 bytes of space from the stack
    - more on this with the help of a picture
    - “**pop eax**” is the 32-bit version, it will copy the 4-byte data from the top of stack to `eax` and then increase `esp` by 4 to de-locate the data

The "call" instruction

- The `call` is a special instruction for making function calls
- `call funct` → stores the return address (address immediately after the `call` instruction itself) and jumps to `funct` to run it (i.e calls the '`funct`' function)
- For example "`call factorial`"
-
  - ❑ stores the address of "instruction 4" to the Stack and then the program jumps to **`factorial`**
  - ❑ the instruction pointer (rip/eip) will be copied the address of instruction x by the `call` instruction, the computer executes the instruction by referring to rip.

```
main:
    instruction 1
    instruction 2
    call factorial
    instruction 4
       :   :   :
factorial:
     instruction x
     instruction x +1
      ret
```

The “ret” instruction

- The `ret` is a special instruction for finishing a function call and returning back to the caller
- `ret` → pops the return address stored in stack and returns back to the caller

- For example “`ret`”

  for the sample program at the right:
  - Will pop the stored address (of instruction 4) from the stack
  - and then the program will jump back to resume executing the main function at instruction 4 (i.e. function call returned)

```
main:
    instruction 1
    instruction 2
    call factorial
    instruction 4
      :   :   :
factorial:
    instruction x
    instruction x +1
    ret
```

x86 Calling Convention

- We will be using 32-bit examples to explain, but the idea is the same for 64-bit program (usally we just need to change the registers to the 64-bit version)
- How to pass arguments
    - In the AMD64 convention the first 6 arguments are copied to the following registers in a non-syscall function call
        - rdi, rsi, rdx , rcx , r8 , r9
    - In the AMD64 convention, in a syscall function call, the first 6 arguments are copied to
        - rdi, rsi, rdx , r10, r8 , r9 (the only change is rcx->r10, because syscall will clobber rcx, destroying the argument passed)
        - Returned value of the syscall will be in rax
    - Further arguments (i.e. 7th argument and above) are pushed onto the stack in reverse order, so **`func(arg0, arg1,…,arg6,arg7,arg8)`** will place **`arg8`** at the highest memory address, then **`arg7`**, then **`arg6`**

x86 Calling Convention

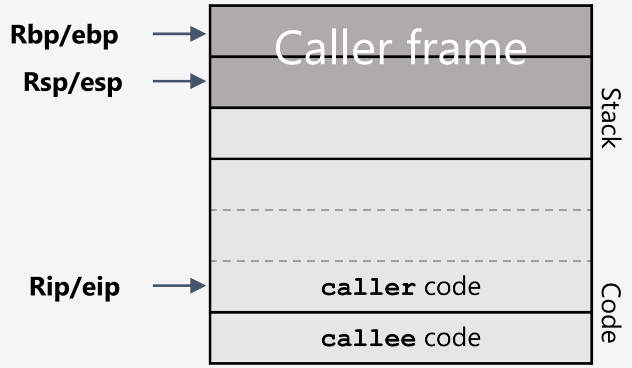
- Which registers are caller-saved or callee-saved
  - **Callee-saved**: The callee must not change the value of the register when it returns
  - **Caller-saved**: The callee may overwrite the register without saving or restoring it

Register	Usage	callee saved
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–%r14	callee-saved registers	Yes
%r15	callee-saved register; optionally used as GOT base pointer	Yes
%r16–%r31	temporary registers	No
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%xmm16–%xmm31	temporary registers	No
%tmm0–%tmm7	temporary registers	No
%mm0–%mm7	temporary registers	No
%k0–%k7	temporary registers	No
%st0,%st1	temporary registers, used to return `long double` arguments	No
%st2–%st7	temporary registers	No
%fs	thread pointer	Yes
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes
tilecfg	Tile control register	No

These are the **callee saved registers**

Calling a Function in x86

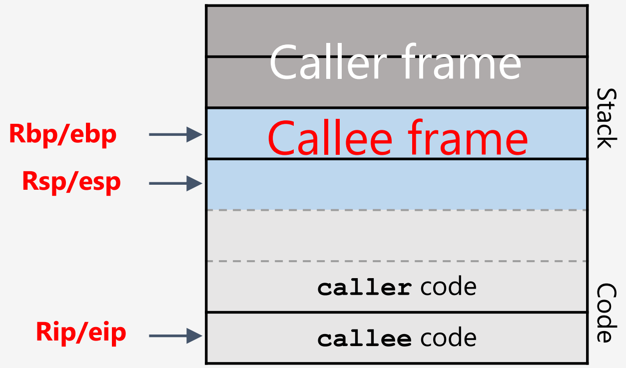
- When a function is called, the RSP/ESP and RBP/EBP registers need to be changed to create a new stack frame, and the RIP/EIP must move to the callee's code
- When returning from a function, the RSP/ESP, and RBP/EBP must return to their old values
- RIP/EIP should point to the return address in the caller



Before function call

Calling a Function in x86

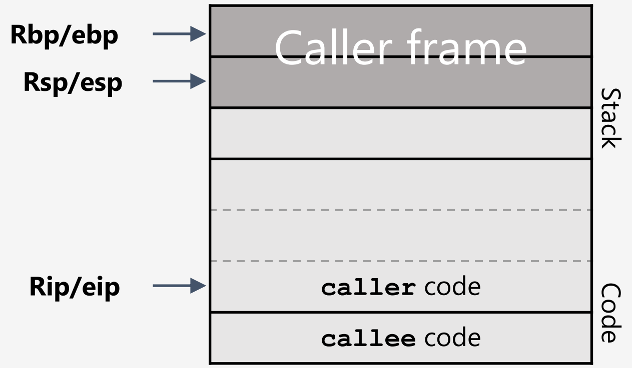
- When a function is called, the RSP/ESP and RBP/EBP registers need to be changed to create a new stack frame, and the RIP/EIP must move to the callee's code
- When returning from a function, the RSP/ESP, and RBP/EBP must return to their old values
- RIP/EIP should point to the return address in the caller



During function call

Calling a Function in x86

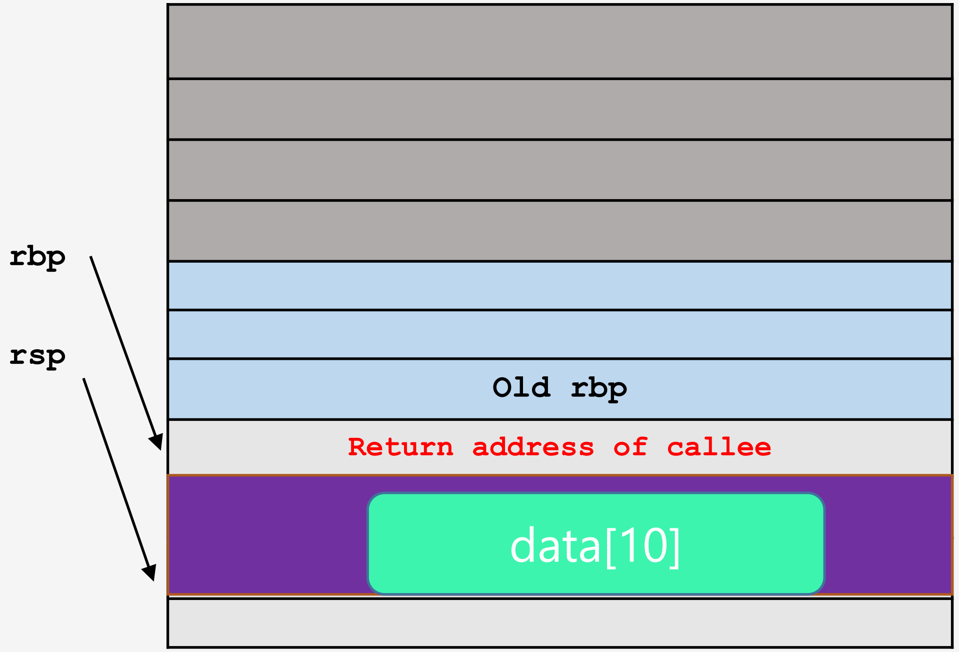
- When a function is called, the RSP/ESP and RBP/EBP registers need to be changed to create a new stack frame, and the RIP/EIP must move to the callee's code
- When returning from a function, the RSP/ESP, and RBP/EBP must return to their old values
- RIP/EIP should point to the return address in the caller



After function call

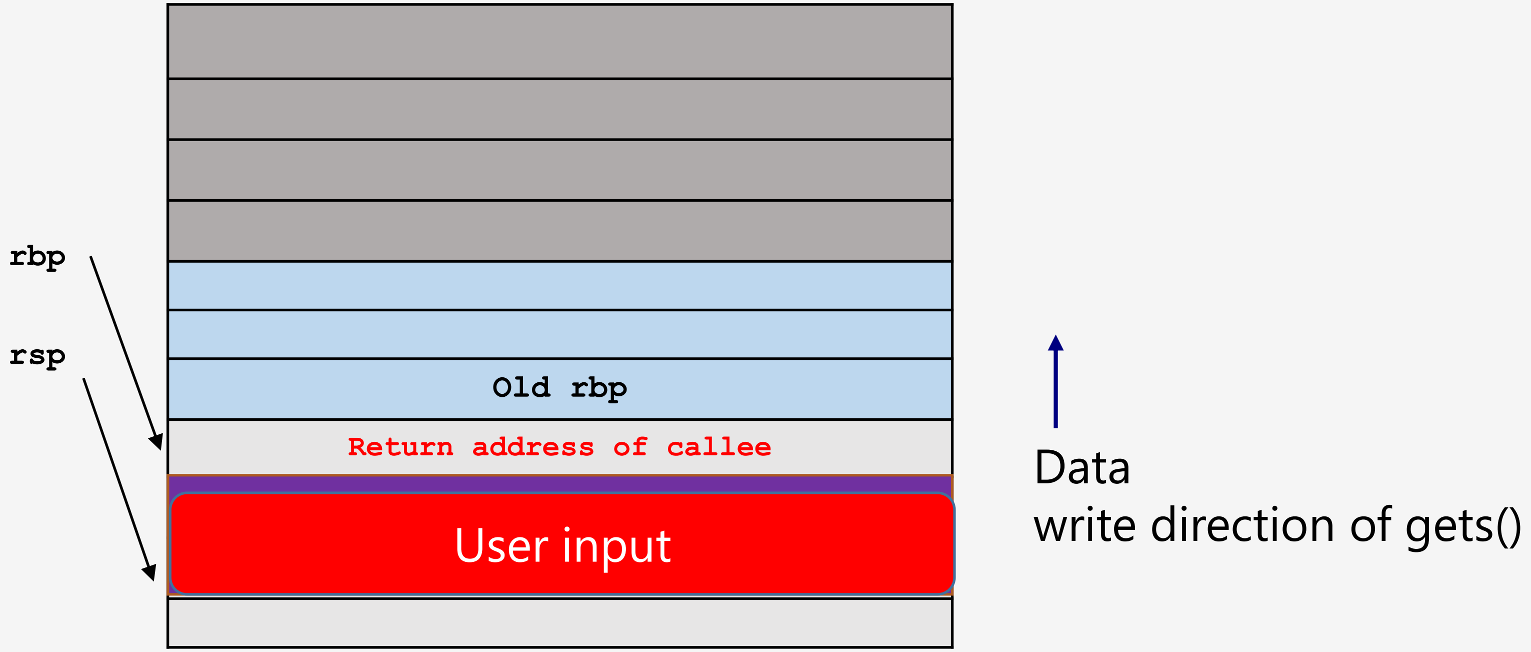
Buffer overflow

- Here is what would happen when the user inputs a piece of data bigger than the allocated 10 bytes



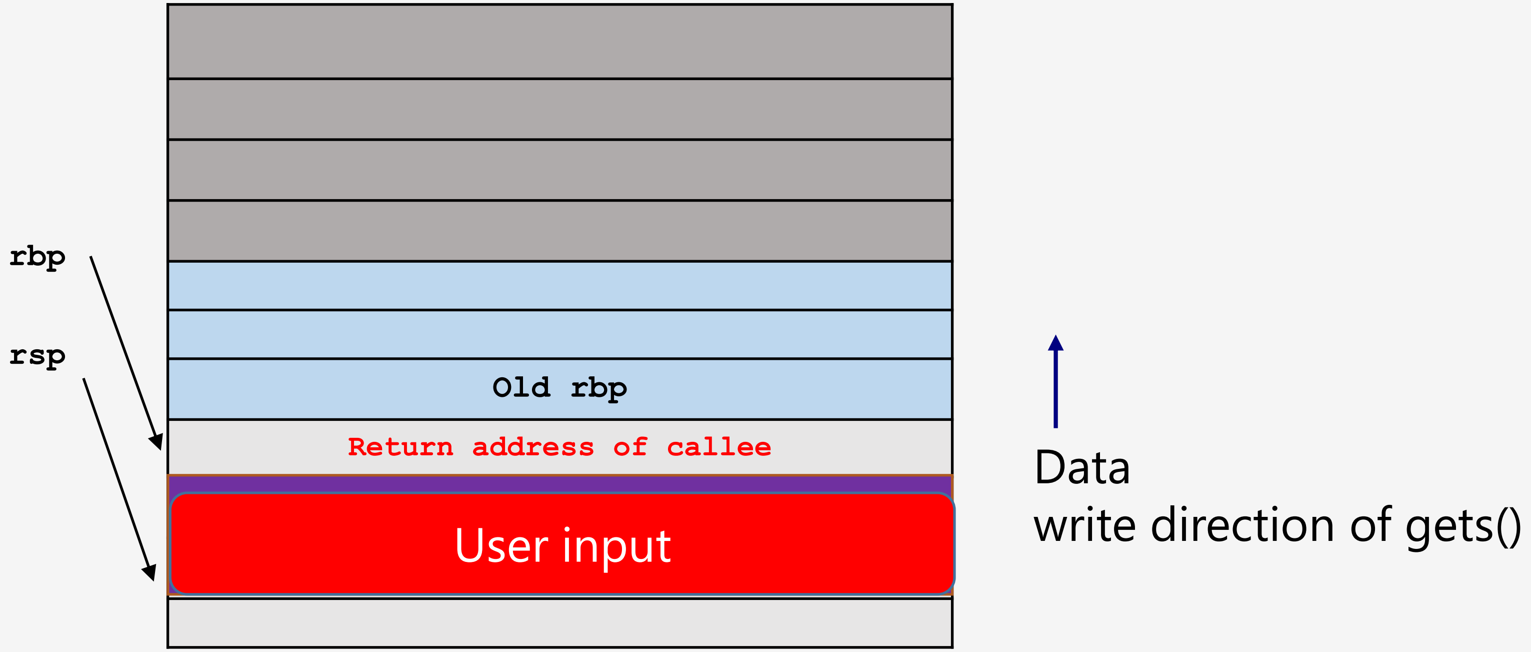
Buffer overflow

- Here is what would happen when the user inputs a piece of data bigger than the allocated 10 bytes



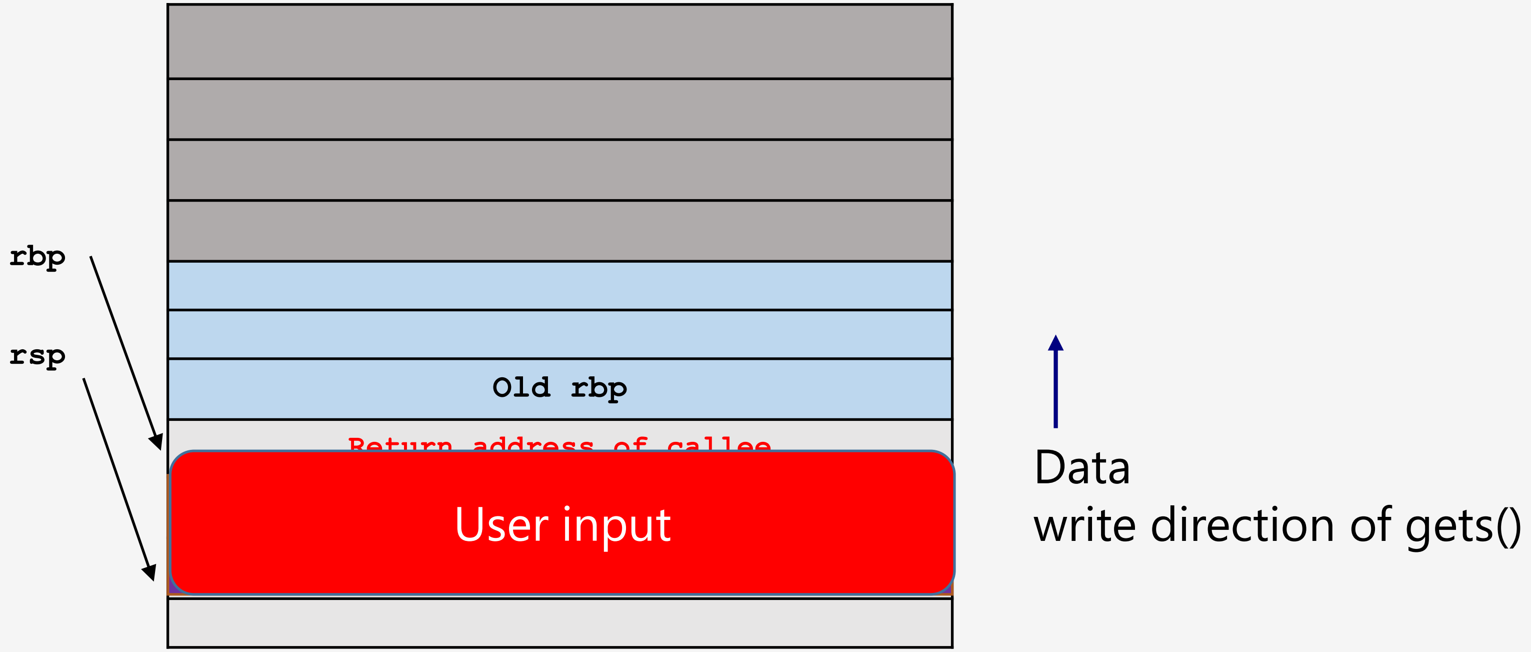
Buffer overflow

- Here is what would happen when the user inputs a piece of data bigger than the allocated 10 bytes



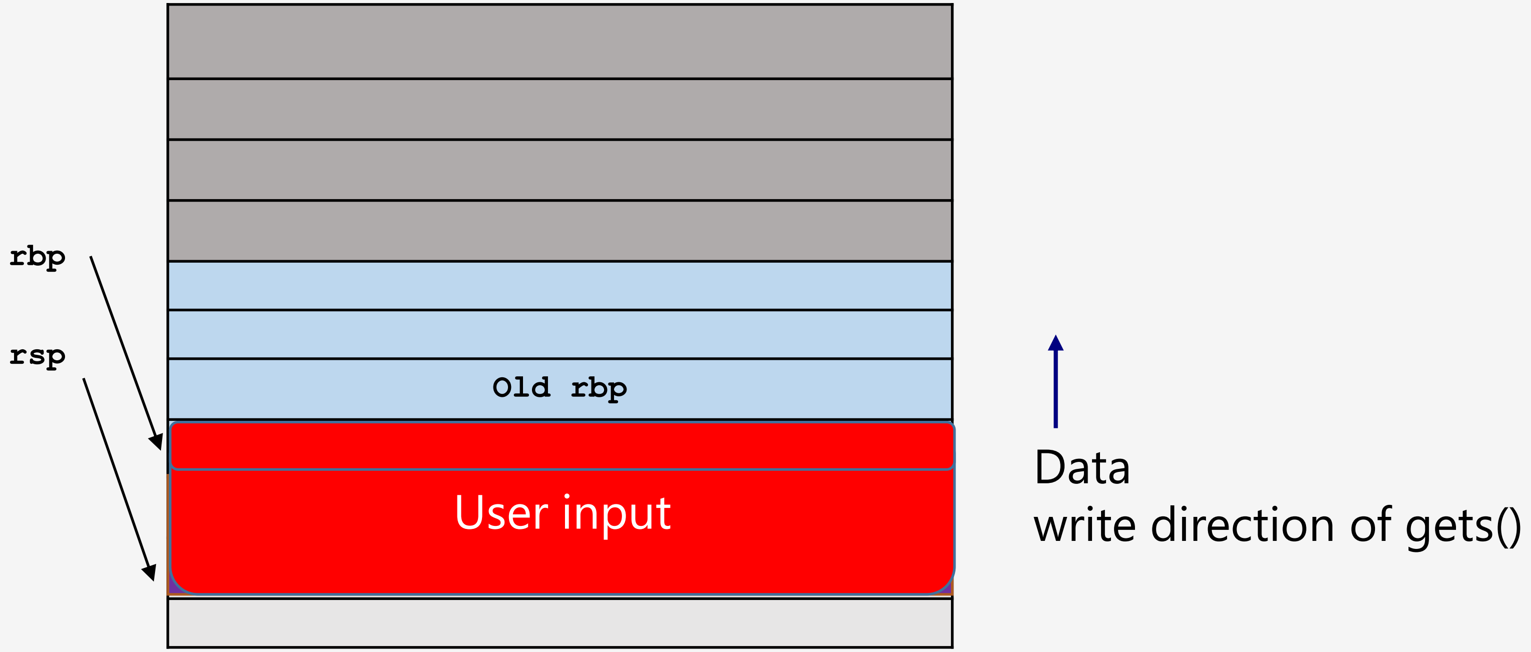
Buffer overflow

- Here is what would happen when the user inputs a piece of data bigger than the allocated 10 bytes



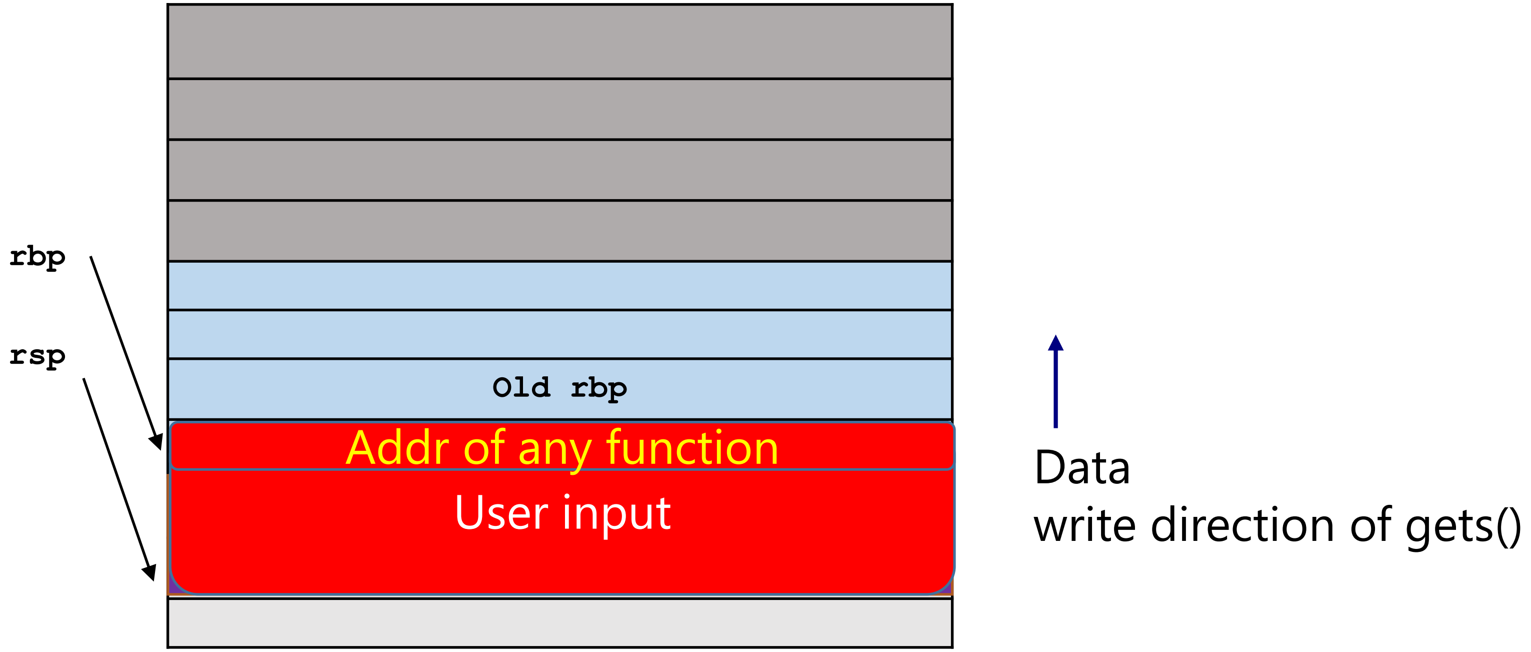
Buffer overflow

- Here is what would happen when the user inputs a piece of data bigger than the allocated 10 bytes



Buffer overflow

- Here is what would happen when the user inputs a piece of data bigger than the allocated 10 bytes



Buffer overflow

- The “ret” instruction at the end of the function will return to the caller according to this stored address on the stack.

C-version of a Shellcode

```c
#include <unistd.h>
#include <stdlib.h>
void main() {
   char* args[2];

   args[0] = "/bin/sh";
   args[1] = NULL;
   execve(args[0], args, NULL);
}
```

Shellcode in assembly (position independent)

To run the shellcode, we need the registers to be in the following state:

(see https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit)

NR	syscall name	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)
59	execve	0x3b	const char *filename	const char *const *argv	const char *const *envp

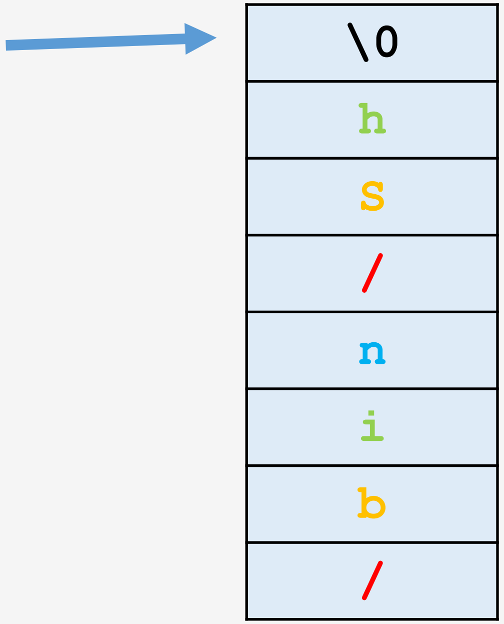
1. Value 59 (0x3b) in rax (execve index in syscall table)
2. rdi = address of the string “bin/sh”
3. rsi = NULL 0x0
4. rdx = NULL (0x0)

Shellcode in assembly (position independent)

int execve(char* filename, char* argv[], char* envp[])
execve(args[0], args, NULL);

```
BITS 64

mov   rax,0x3b
                      h s / n i b /
mov   rbx,0x0068732f6e69622f
push  rbx ;rsp now points to "/bin/sh"
mov   rdi,rsp

mov   rsi, 0
mov   rdx,0

syscall
```

- Value 59 (0x3b) in rax (execve index in syscall table)
- We use the stack to store the string "/bin/sh", remember intel is little endian
- rdi = rsp → "`/bin/sh`"
- rsi = `NULL` (0x0)
- rdx = `NULL` (0x0)
- Execute the syscall



Off-by-one-byte to Control Saved rbp

- As shown in the example, we may control the saved rbp on the stack
- When the function returns the base pointer of the caller (main, in the examples) will be changed
- Therefore, when the caller returns, the saved rip on the stack will be different (and hopefully controlled by us)
- This is an example of stack pivoting
  - We modify rsp, to change what the memory region the program uses as stack

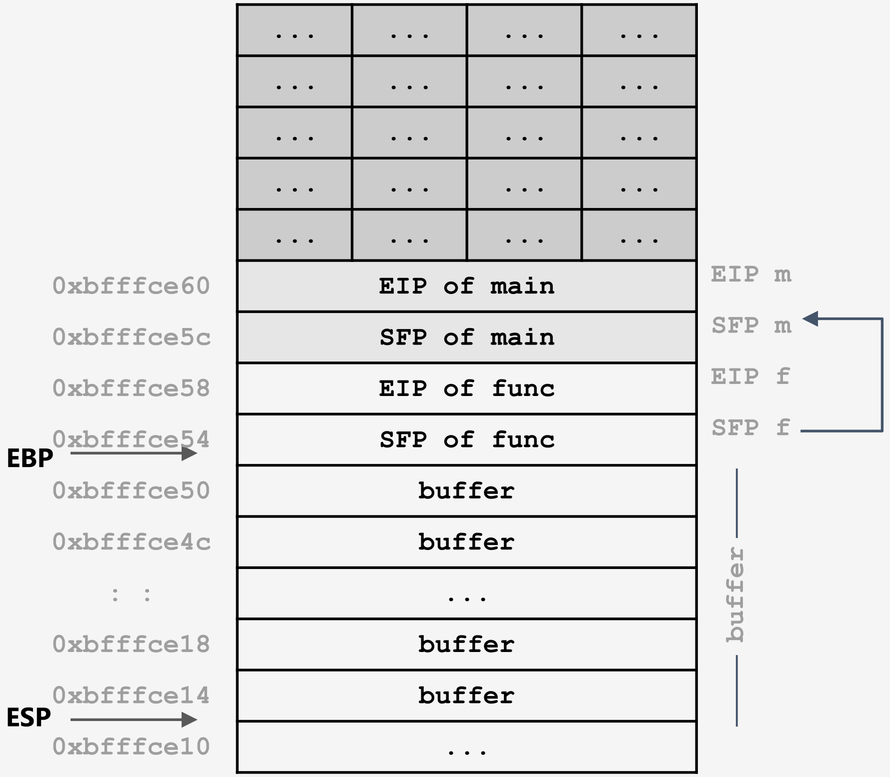
A 32-bit example (same idea for 64-bit machines)

Goal: execute the shellcode located at **0xaabbccdd**

```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

EIP

```
func:
    ...
    ...
--> last instruction of for-loop
    mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
    mov $esp, $ebp
    pop $ebp
    ret
```



A 32-bit example

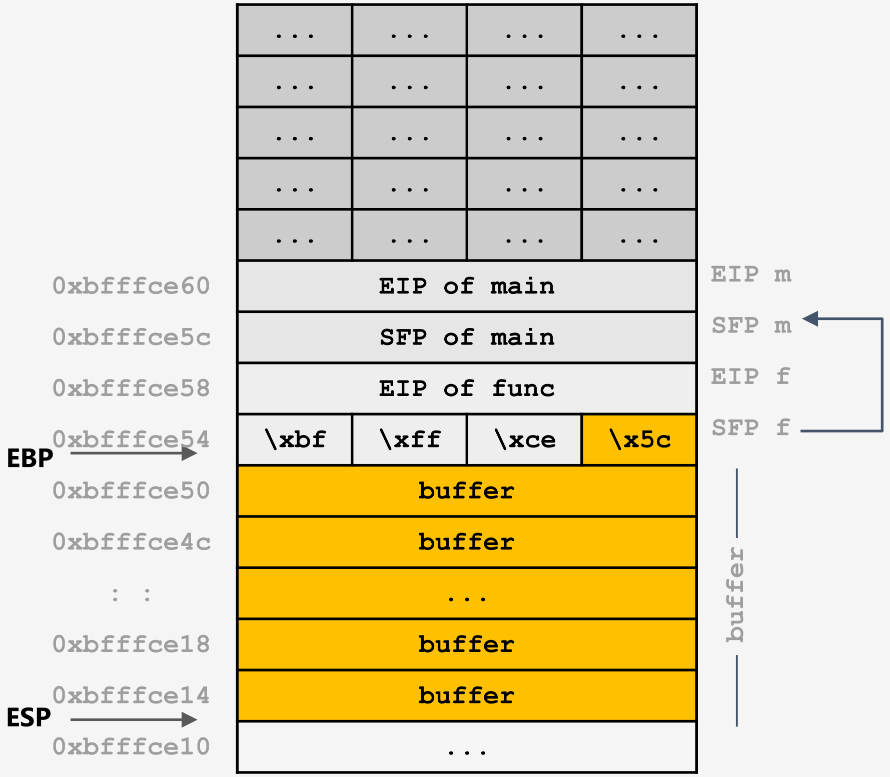
The attacker is able to overwrite all of **`buffer`** and the least-significant byte of the stack frame pointer (SFP) of **`func()`**.

If the attacker can change where **`func()`** points, how can they use this to execute shellcode?

```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

EIP

```
func:
    ...
    ...
--> last instruction of for-loop
    mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
    mov $esp, $ebp
    pop $ebp
    ret
```



A 32-bit example

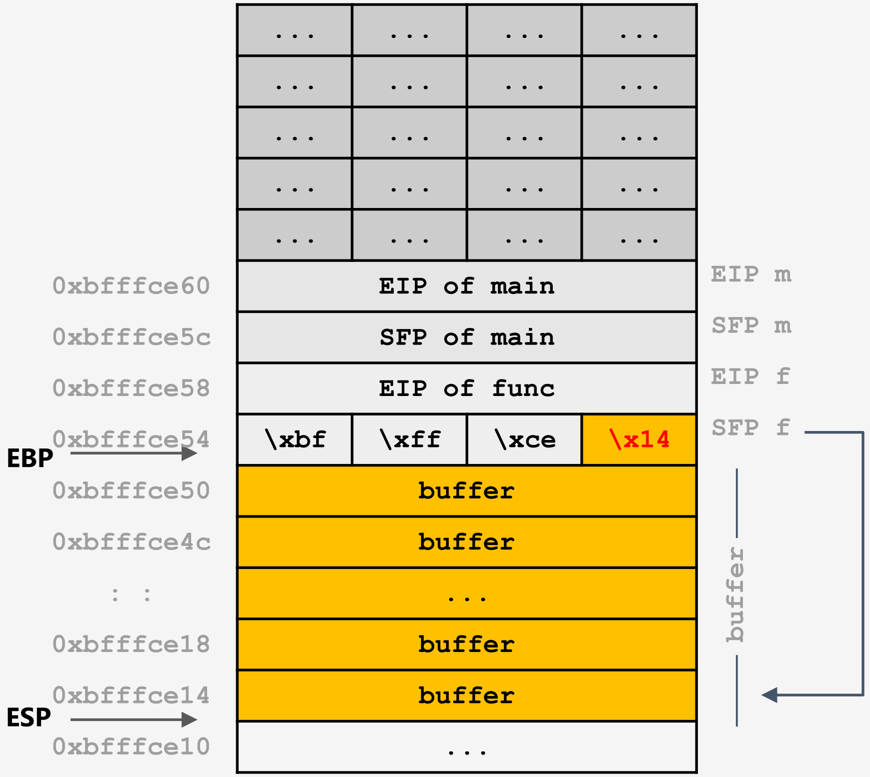
The SFP **`func()`** nows points inside **`buffer`**, this **`buffer`** stores user input (under control of the attacker)

SFP usually points to the base stack framer of the caller function (i.e. **`main()`**)

```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

EIP

```
func:
    ...
    ...
--> last instruction of for-loop
    mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
    mov $esp, $ebp
    pop $ebp
    ret
```



A 32-bit example

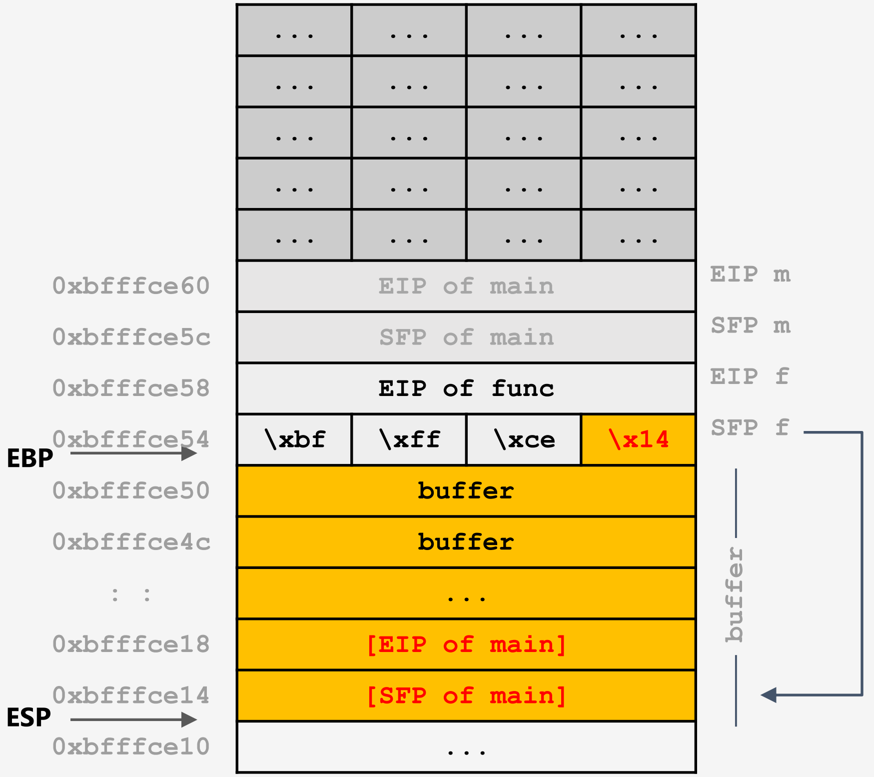
The program now thinks that the SFP and EIP of **`main()`** are inside **`buffer`**, this **`buffer`** stores user input (under control of the attacker).

The attacker knows that when he makes that 1 byte change to the SFP of **`func()`**, so he can overwrite the data at the address where the program consider the **`main()`** "EIP" – this will be the address where **`main()`** will return.

```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

EIP →

```
func:
    ...
    ...
    last instruction of for-loop
    mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
    mov $esp, $ebp
    pop $ebp
    ret
```



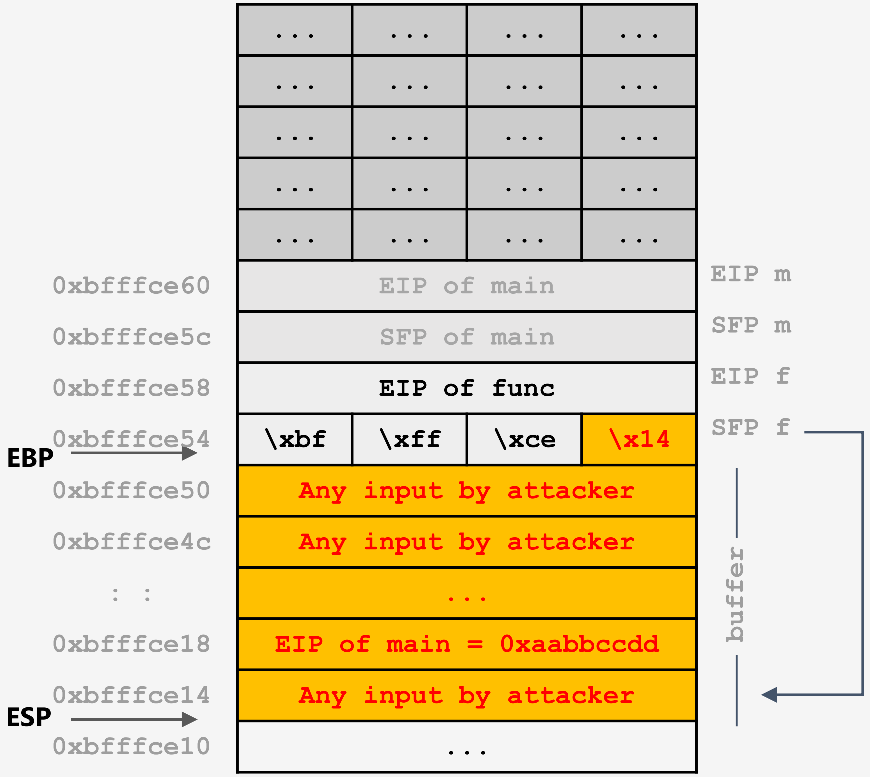
A 32-bit example

Let's run the program and see what happens
Mind that the target shellcode is located at **0xaabbccdd**

```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

EIP

```
func:
    ...
    ...
    last instruction of for-loop
--> mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
    mov $esp, $ebp
    pop $ebp
    ret
```



A 32-bit example

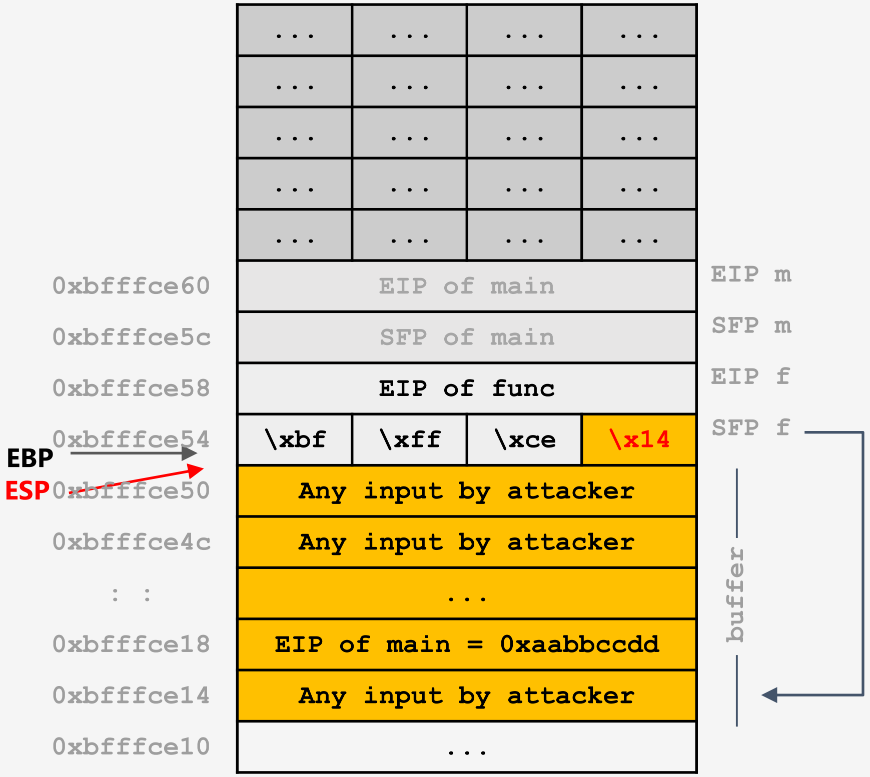
Let's run the program and see what happens

Epilogue step 1: pointing **ESP** back to **EBP** (points **ESP** to the beginning of the stack frame). This is to "clear" the allocated spaces of the stack for **`main()`**

```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

EIP

```
func:
    ...
    ...
    last instruction of for-loop
    mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
    mov $esp, $ebp
    pop $ebp
    ret
```



A 32-bit example

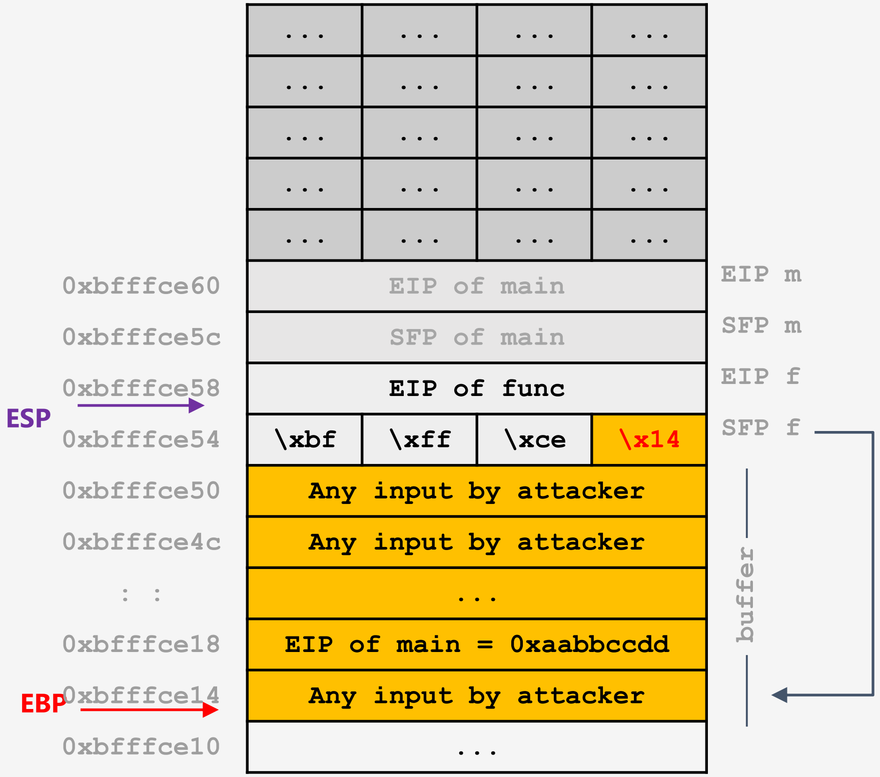
Let's run the program and see what happens

Epilogue step 2: point the **EBP** back to the starting frame for the caller, the **`main()`**, this is to restore the stack frame allocated to **`main()`** but because of the 1 byte change, the **EBP** of **`main()`** points to the inside of **buffer**

```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

EIP

```
func:
    ...
    ...
    last instruction of for-loop
    mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
    mov $esp, $ebp
    pop $ebp
    ret
```



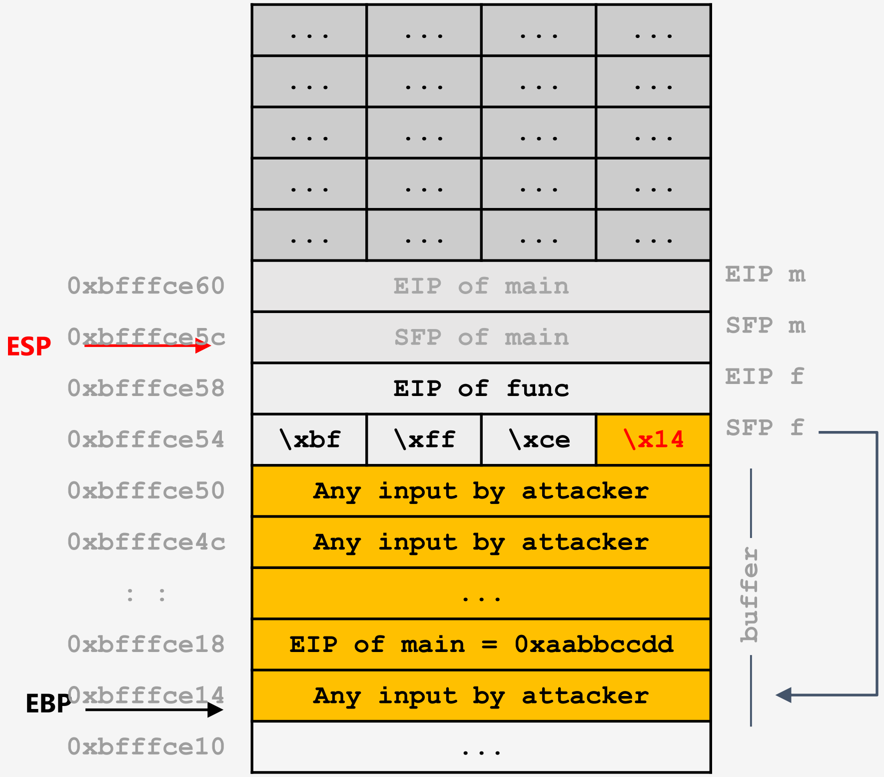
A 32-bit example

Let's run the program and see what happens

Epilogue step 3: retrieve **EIP** from the stack, and return to **`main()`**, up till now everything seems normal because we haven't used the modified return address the **`main()`** would return to, we are just returning to **`main()`**

```c
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

```
func:
    ...
    ...
    last instruction of for-loop
    mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
EIP mov $esp, $ebp
    pop $ebp
    ret
```



A 32-bit example

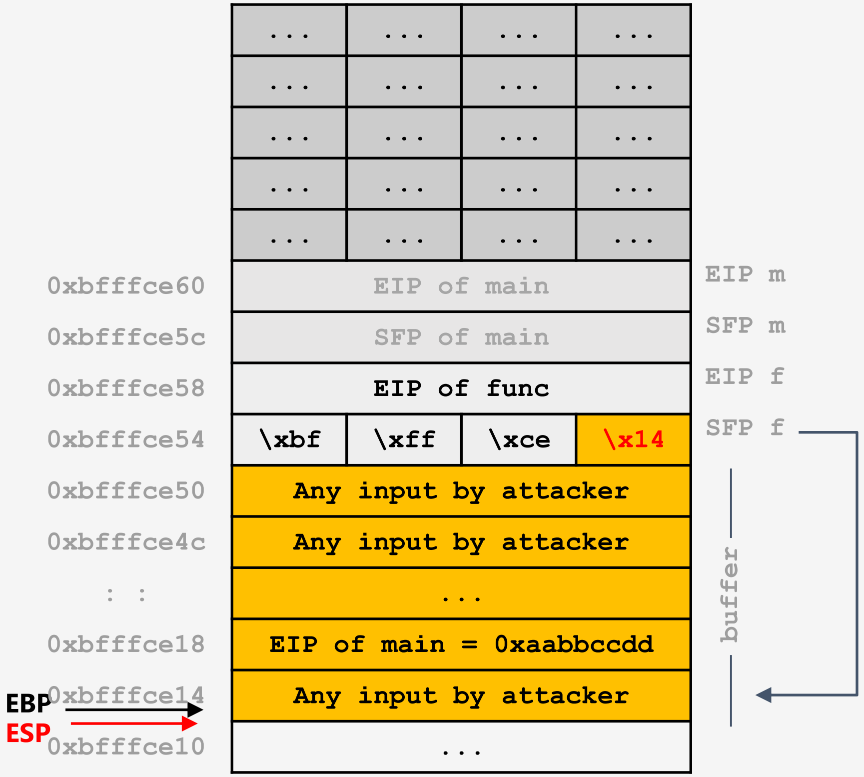
After returning to **main()**, the bad thing starts to happen

Epilogue step 1: pointing **ESP** back to **EBP**, because the program thought this **EBP** is the start of the stack frame for the caller of **main()**, but this **EBP** is a value supplied by the attacker!

```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

```
func:
    ...
    ...
    last instruction of for-loop
    mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
    mov $esp, $ebp
    pop $ebp
    ret
```

EIP



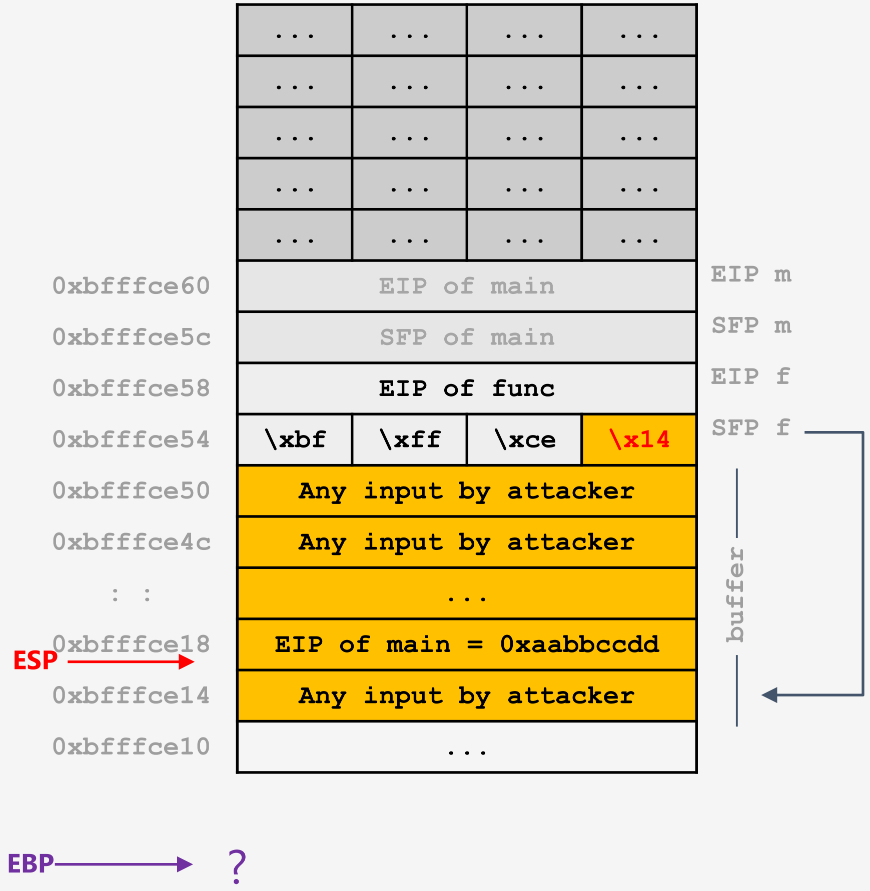
A 32-bit example

After returning to **main()**, the bad thing starts to happen

Epilogue step 2: point the **EBP** back to the starting frame for the caller, which is a garbage address supplied by attacker.

```c
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

```
func:
        ...
        ...
        last instruction of for-loop
        mov $esp,$ebp
        pop $ebp
        ret

main:
        ...
        call func
        mov $esp, $ebp
        pop $ebp
EIP  →  ret
```



A 32-bit example

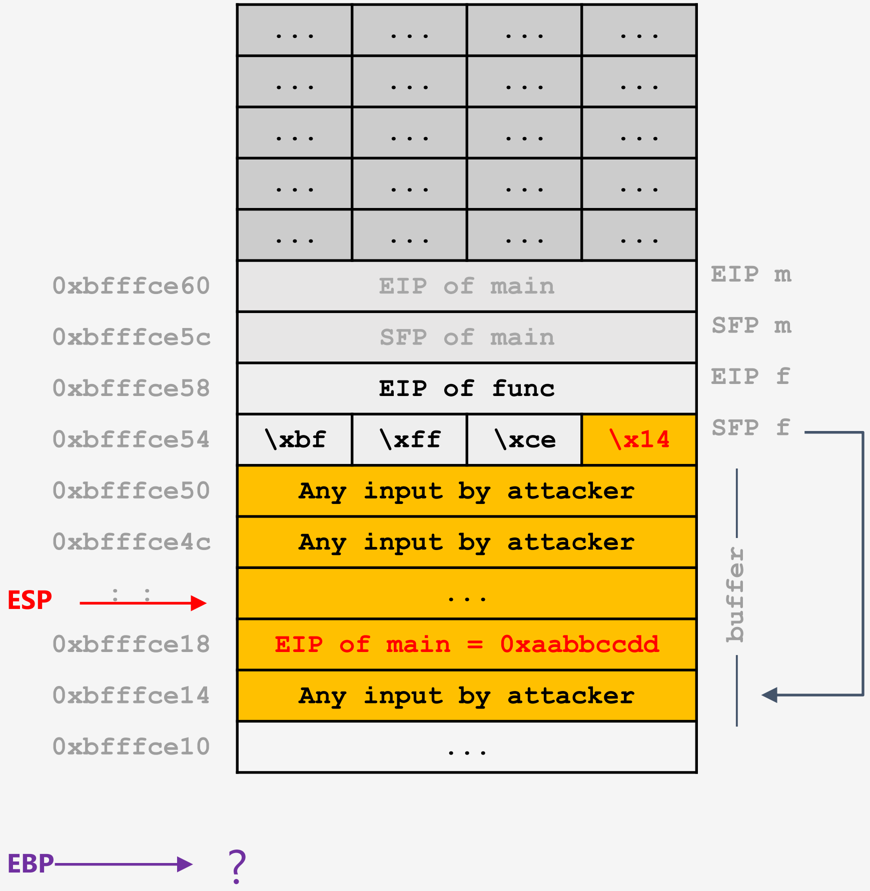
After returning to **main()**, the bad thing starts to happen

Epilogue step 3: retrieve **EIP** from the stack, and return to the caller, and the address to return to is **0xaabbccdd**, which is the address of our shellcode

```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

```
func:
    ...
    ...
    last instruction of for-loop
    mov $esp,$ebp
    pop $ebp
    ret

main:
    ...
    call func
    mov $esp, $ebp
    pop $ebp
    ret
```

EIP →



A 32-bit example

After returning to **main()**, the bad thing starts to happen

Epilogue step 3: retrieve **EIP** from the stack, and return to the caller, and the address to return to is **0xaabbccdd**, which is the address of our shellcode

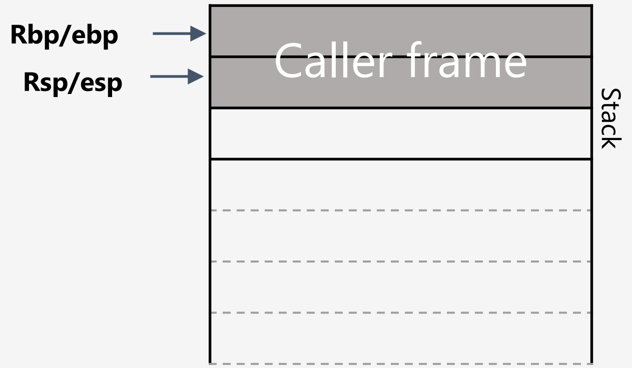
```
void func(char *offByOne, int i) {
  char buffer[64];
  for(i = 0; i <= 64; i++)
  {
    buffer[i]=offByOne[i];
  }
}

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

  func(argv[1], 0);
  return 0;
}
```

?

...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
...	...	...	...	
EIP of main				EIP m
SFP of main				SFP m
EIP of func				EIP f
\xbf	\xff	\xce	\x14	SFP f
Any input by attacker				buffer
Any input by attacker				
...				
EIP of main = 0xaabbccdd				
Any input by attacker				
...				

ROP: what happens when we call a function the normal way

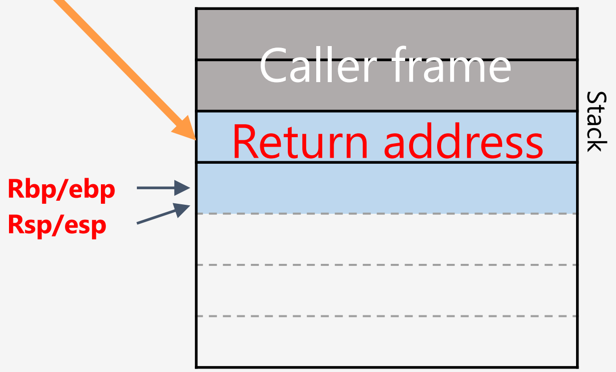
- The is the original stack before the function call



Before function call

ROP: what happens when we call a function the normal way

- When a function is called using the `call` instruction, the call instruction will put the return address to the stack
- Mind that the return address is not within the stack boundary between rbp and rsp, but for simplicity it is still considered to be in the callee's stack frame



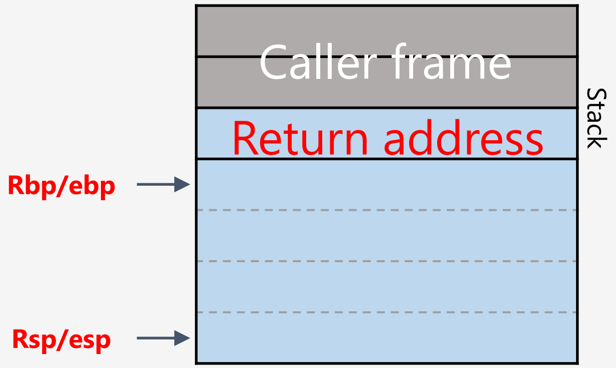
During function call

```
callee:
 push rbp
 mov rbp, rsp
```

Function prologue

ROP: what happens when we call a function the normal way

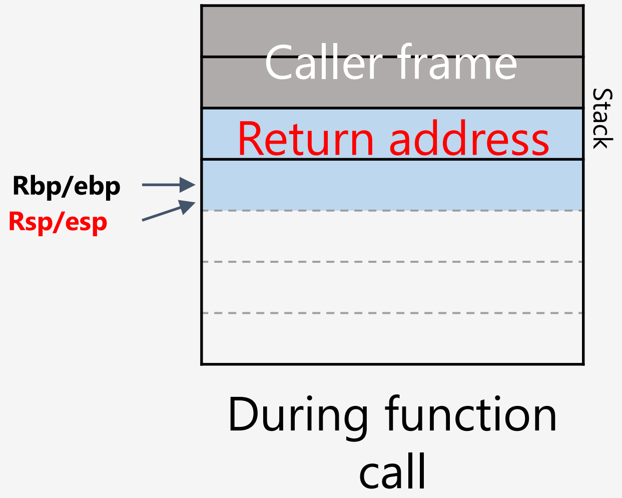
- The stack will grow towards the bottom as needed



During function call

ROP: what happens when we call a function the normal way

- At the end of the function call stack will shrink back to its start size

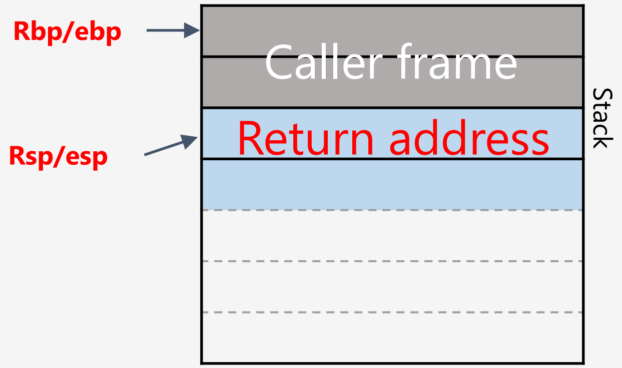



```
callee:
    ...
    mov rsp,rbp
    pop rbp
    ret
```

Function epilogue

ROP: what happens when we call a function the normal way

- At the end of the function call stack will shrink back to its start size



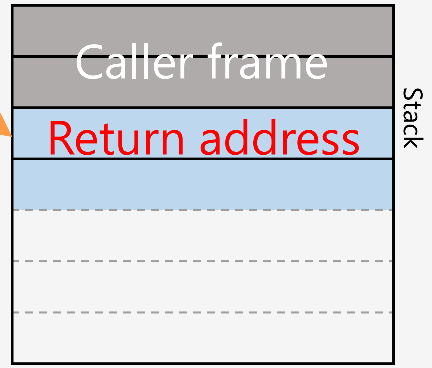
During function call

```
callee:
    ...
    mov rsp,rbp
    pop rbp
    ret
```

Function epilogue

ROP: what happens when we call a function the normal way

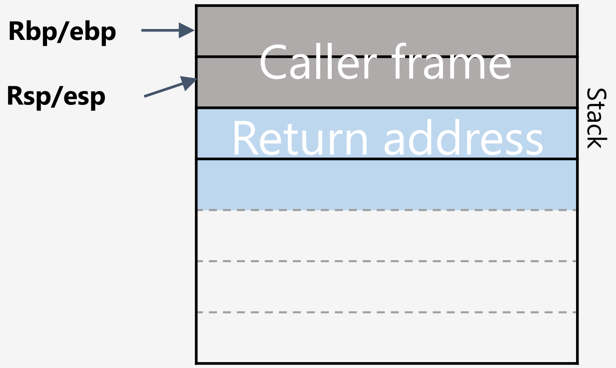
- Thr **`ret`** instruction at the end will pop the Return address and return to it.



During function call

ROP: what happens when we call a function the normal way

- Thr **`ret`** instruction at the end will pop the Return address and return to it.



During function call

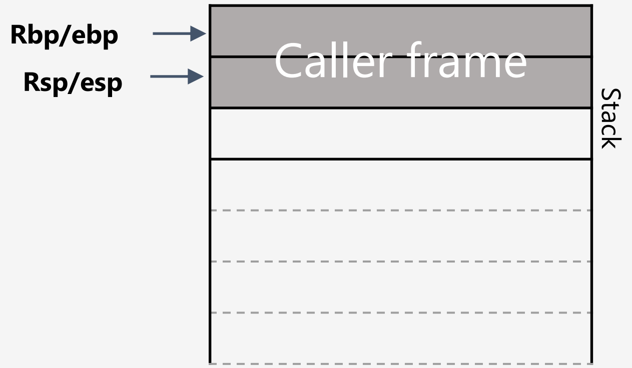
```
callee:
    ...
    mov rsp,rbp
    pop rbp
    ret
```

Function epilogue

Rip/eip = Return address

ROP: what happens when we call a function using ret

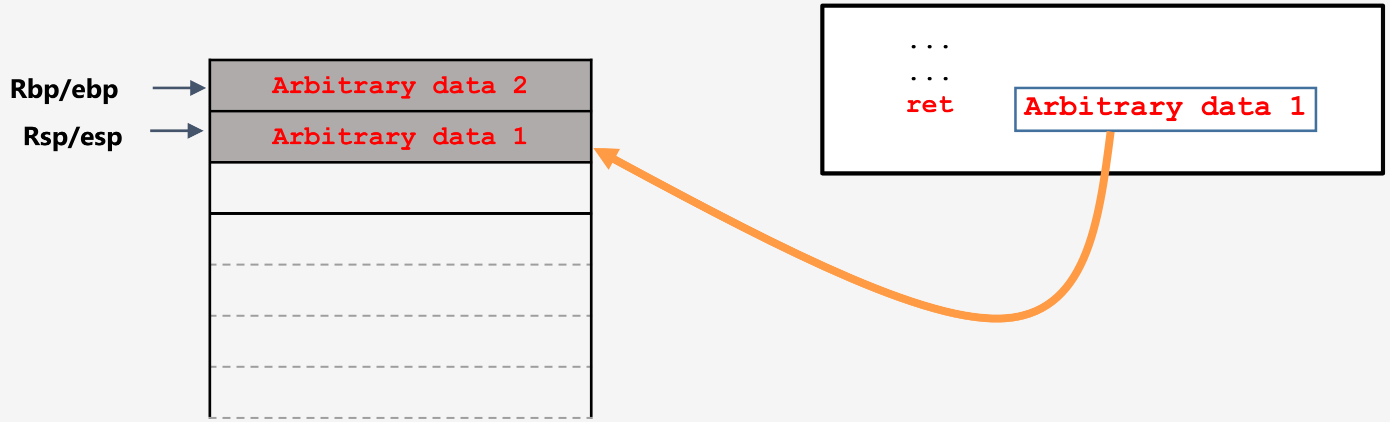
- There is no `call` instruction to put the return address to the top of the caller stack frame
- The `ret` instruction just pops without pushing anything



Before function call

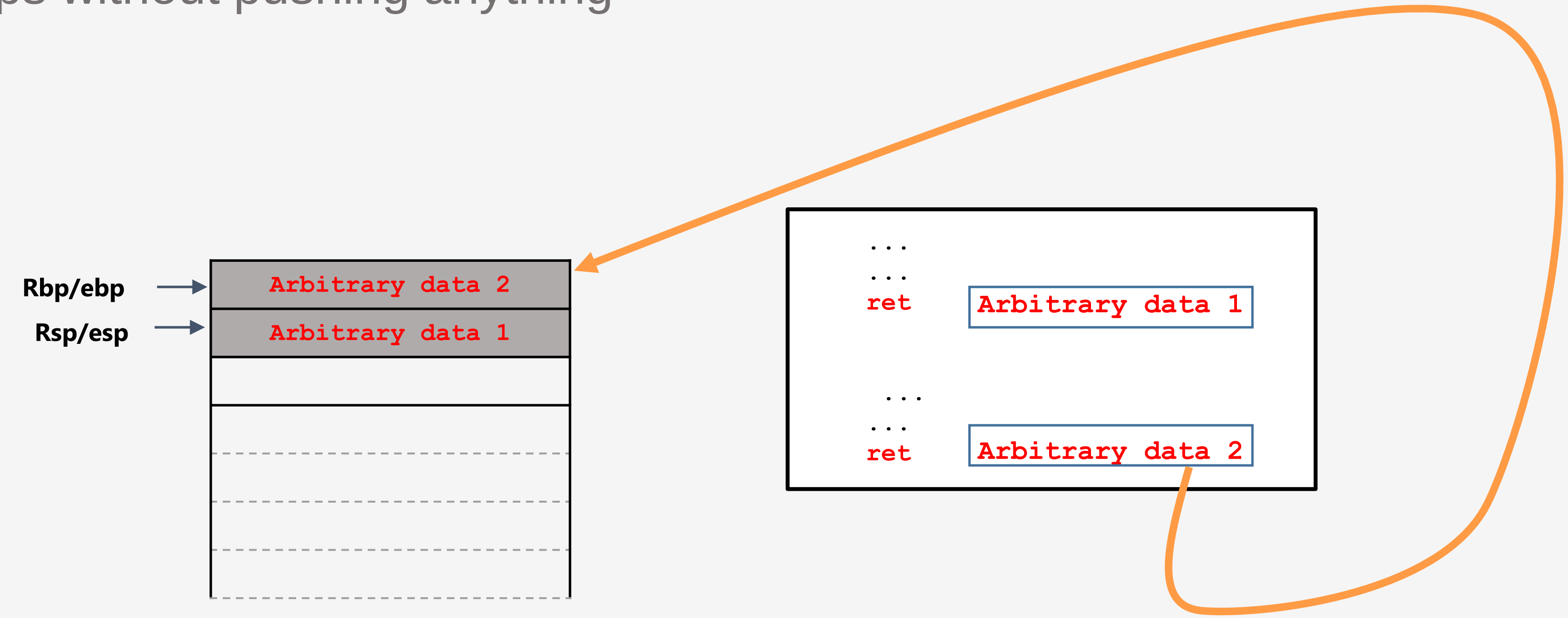
ROP: what happens when we call a function using ret

- There is no **`call`** instruction to put the return address to the top of the caller stack frame
- The **`ret`** instruction just pops without pushing anything



ROP: what happens when we call a function using ret

- There is no **`call`** instruction to put the return address to the top of the caller stack frame
- The **`ret`** instruction just pops without pushing anything



ROP: what happens when we call a function using ret

- There is no **`call`** instruction to put the return address to the top of the caller stack frame
- The **`ret`** instruction just pops without pushing anything