Memory Corruption Attacks and Defenses: Background

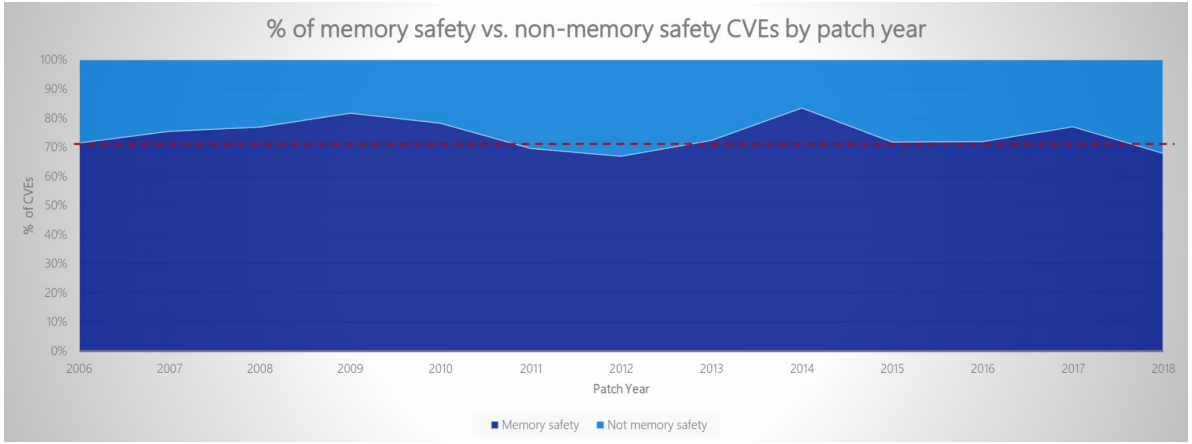
Introduction to Security (192.019)

Pedro **Bernardo**, Mauro **Tempesta**

Security & **Privacy** Research Unit (192-06)
https://**secpriv.wien**

Memory-Safety Issues are Still Dominant

Root cause trends of vulnerabilities

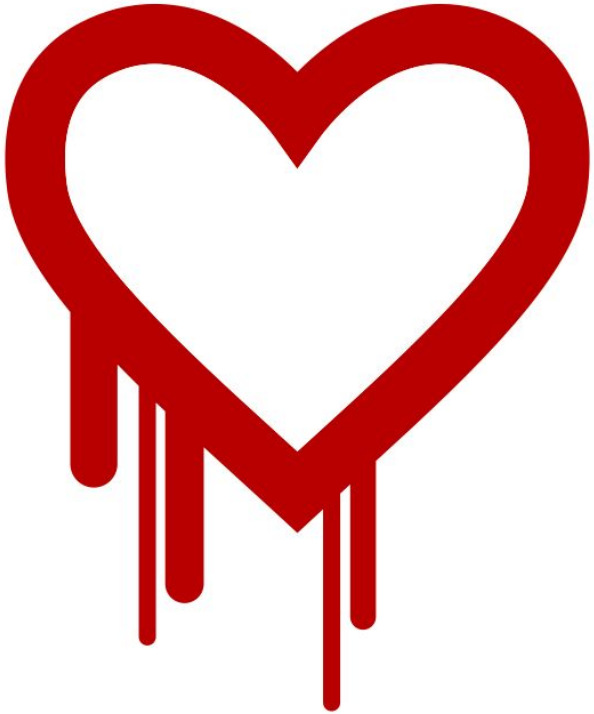


~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues

https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL

Heartbleed

- Buffer over-read vulnerability in the widely-used OpenSSL cryptographic library
- Cause: missing bounds check before a memory copy operation that uses non-sanitized user input as the length parameter
- This vulnerability allows anyone on the Internet to read the memory of the systems protected by vulnerable versions of OpenSSL, including:
    - Private keys, decrypted packets (in memory), etc.



TP-Link httpd vulnerability (2019)

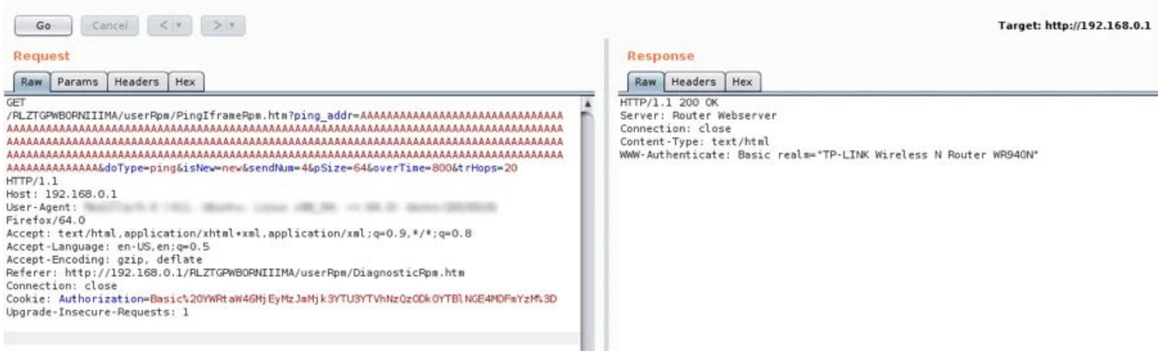
- Buffer overflow in TP-Link router firmware
- Character limit checked in the user interface
- But, an attacker can still inspect and modify network requests, adding more characters
- This vulnerability allows for remote takeover of the router

Security Intelligence News Series

Home / Topics / Network

Buffer Overflow Vulnerability in TP-Link Routers Can Allow Remote Attackers to Take Control

April 8, 2019 | By Grzegorz Wypych co-authored by Limor Kessem | 9 min read



https://securityintelligence.com/buffer-overflow-vulnerability-in-tp-link-routers-can-allow-remote-attackers-to-take-control/

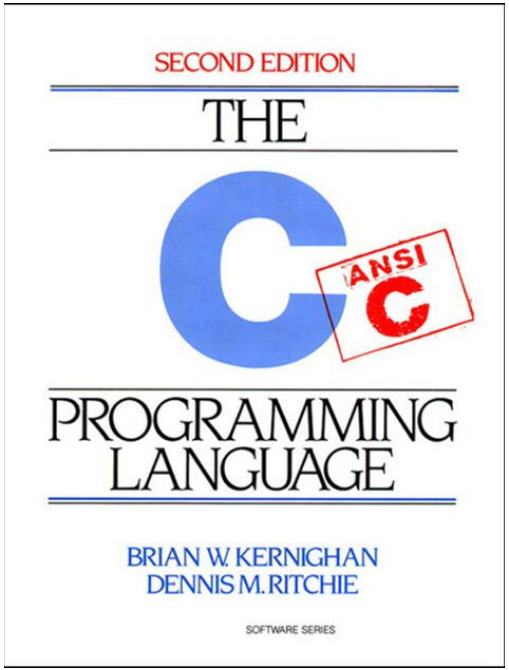
Background: C Programming

Communicating with Computers

- Computers do not understand human languages
- At the lowest level, computers only understand **sequences of numbers** that represent operational codes **(opcodes** for short)
- It would be very difficult for humans to write programs in terms of opcodes
- Therefore, **programming languages** were invented to make it easier for humans to write computer programs

C Programming Language

- Introduced by Dennis Ritchie between 1972 and 1973 at Bell Labs to write Unix utilities, and later the Unix kernel
- Compiled, low-level systems language
  - Provides fine-grained control over the machine
  - Fast: low-overhead compilation
- Statically typed and imperative
- Widely used in operating systems, device drivers and embedded programming



C Programming

stdio.h defines the **C standard library**'s related to I/O operations (e.g., printf)

```
#include <stdio.h>

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

     printf("Hello World!\n");

     return 0;

}
```

printf outputs its first argument to standard output (stdout)

Statements end with a semicolon

C Programming

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

int main(int argc, char *argv[]){
     char buffer[10] = {0};

     printf("What is your name?\n");

     read(0, buffer, 9);

     printf("Hello %s\n", buffer);

     return 0;
}
```

create a local array called **buffer** that is the size of 10 characters

read 9 characters from **stdin** (file descriptor 0) to the buffer

write the output of **buffer** to stdout using printf

C Programming

```
λ juno name → ./name
What is your name?
AAAABBBBB
Hello, AAAABBBBB
```

Just works!

C Programming

```c
#include <stdio.h>
int main(int argc, char *argv[]){
     char buffer[10] = {0};

     printf("What is your name?\n");

     read(0, buffer, 100);

     printf("Hello %s\n", buffer);

     return 0;
}
```

but... what happens if we read more than the size of **buffer**?

C Programming

Segmentation fault? The program tried to access an inaccessible memory location

```
λ juno name → ./name
What is your name?
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH
Hello, AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH
[1]    79310 segmentation fault (core dumped)  ./name
```

C is not memory-safe!

We will understand why the program crashes later.

C Programming - Scopes

- The scope of a variable is the part of the program where it is accessible
- There are two types of scope: Global Scope and Local Scope:

Local Scope

- Variables declared in the local scope are called **local variables**
- Local variables **are visible in the block** where they are declared, **and in nested blocks**
- Local variables **take precedence over global variables** with the same name

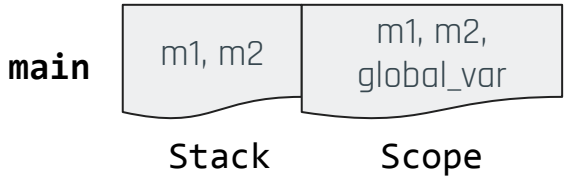
Global Scope

- Variables in the global scope are accessible everywhere
- Variables declared in the global scope are called **global variables**

C Programming - Scopes and Stack Frames

```c
const int global_var = 1;

void func_b(){
      int b1 = 1;
      printf("%d\n", b1);
      return;
}

void func_a(){
      int a1 = 1337;
      func_b();
      return;
}

int main(int argc, char* argv[]){
      int m1 = 20;
      int m2 = 10;
      func_a();
      return 0;
```

m1 and **m2** are local variables, so they are in main's stack frame.

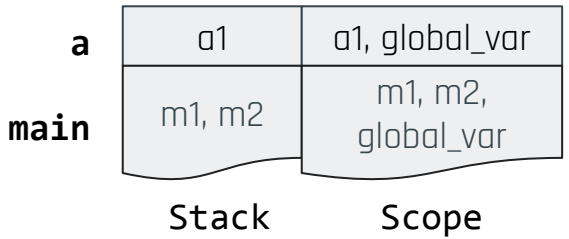


A function can access variables that are either global, or in its own stack frame. These variables are *"in scope"*

C Programming - Scopes and Stack Frames

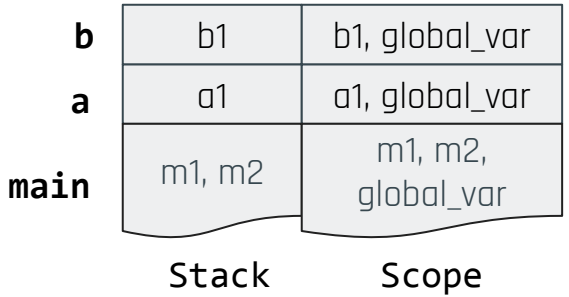
```c
const int global_var = 1;

void func_b(){
      int b1 = 1;
      printf("%d\n", b1);
      return;
}

void func_a(){
      int a1 = 1337;
      func_b();
      return;
}

int main(int argc, char* argv[]){
      int m1 = 20;
      int m2 = 10;
      func_a();
      return 0;
}
```

global_var stays in scope, but **m1** and **m2** are no longer accessible by name



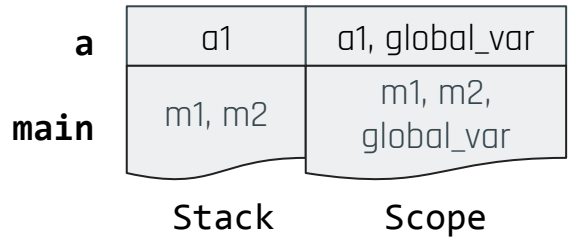
C Programming - Scopes and Stack Frames

```c
const int global_var = 1;

void func_b(){
      int b1 = 1;
      printf("%d\n", b1);
      return;
}

void func_a(){
      int a1 = 1337;
      func_b();
      return;
}

int main(int argc, char* argv[]){
      int m1 = 20;
      int m2 = 10;
      func_a();
      return 0;
```



C Programming - Scopes and Stack Frames

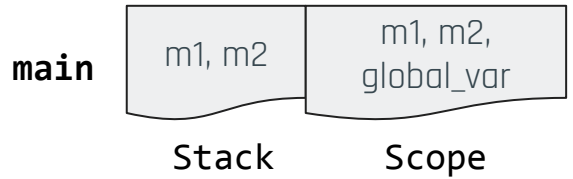
```c
const int global_var = 1;

void func_b(){
      int b1 = 1;
      printf("%d\n", b1);
      return;
}

void func_a(){
      int a1 = 1337;
      func_b();
      return;
}

int main(int argc, char* argv[]){
      int m1 = 20;
      int m2 = 10;
      func_a();
      return 0;
}
```



After returning from a function, its stack frame is destroyed and its local variables are discarded

C Programming - Scopes and Stack Frames

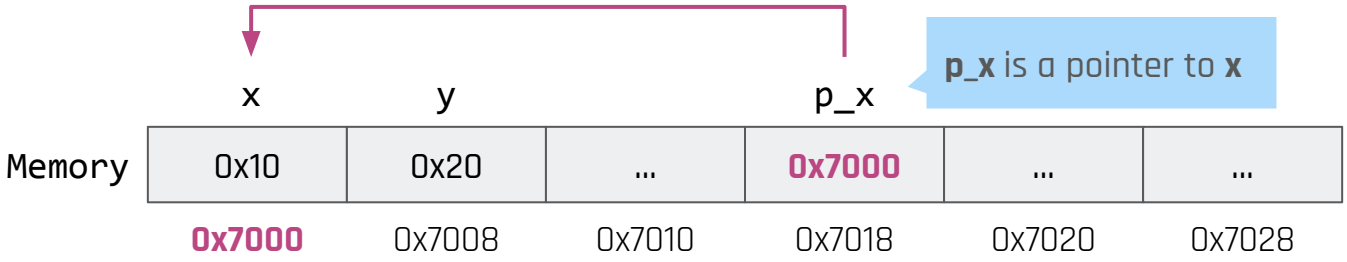
```c
const int global_var = 1;

void func_b(){
      int b1 = 1;
      printf("%d\n", b1);
      return;
}

void func_a(){
      int a1 = 1337;
      func_b();
      return;
}

int main(int argc, char* argv[]){
      int m1 = 20;
      int m2 = 10;
      func_a();
      return 0;
```



After returning from a function, its stack frame is destroyed and its local variables are discarded

C Programming - Pointers

- A pointer is a data type that stores (points to) a memory address
- Pointers can be used to manipulate the data at the address they point to



C Programming - Pointers

```
int main(int argc, char* argv[]){
    int a = 0x10; (1)
    int b = 0x20;

    int* p_a = &a;(2)

    int y = *p_a; (3)
}
```

<type>* declares a pointer of type **type**

& is the *address of* operator

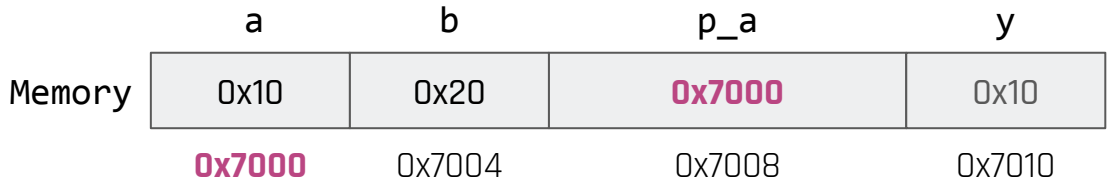
* is the *dereference* operator, and it is used to access the value at the address pointed to by a pointer (aka, *dereferencing a pointer*)

(1) Integer named **a** is set to 0x10

(2) Integer pointer named **p_a** is set to the address of **a**

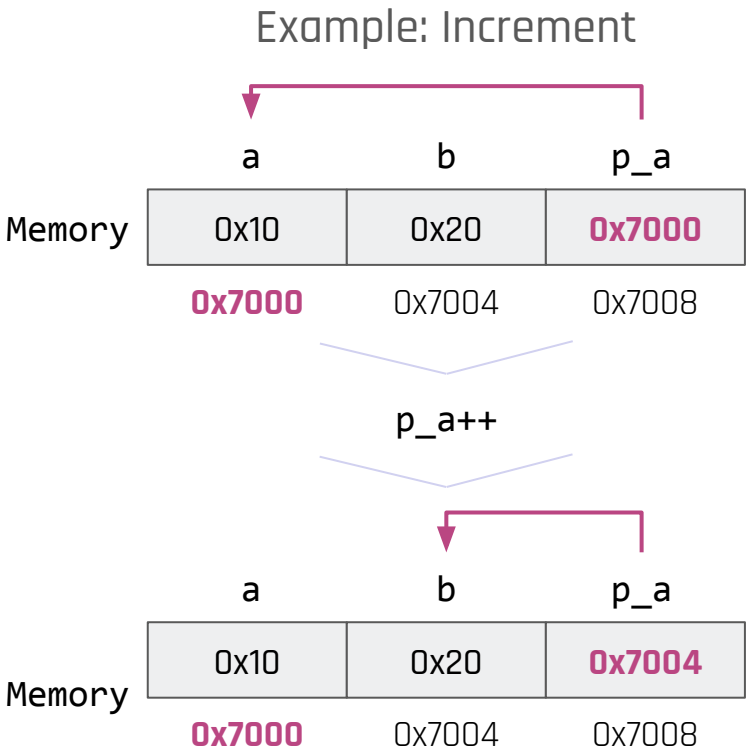
(3) Integer named **y** is set to the value at the address pointed to by **p_a** (y = 0x10)

Generally, in 64-bit architectures, **int** is 32 bits long, but all pointers are 64 bits. The specifics are dictated by the compiler.



C Programming - Pointer Arithmetic

- Arithmetic operations on pointers depend on the pointer type (size)
- Pointers can be:
  - incremented/decremented
  - added/subtracted with integers
  - subtracted/compared with other pointers of the same type



C Programming - Pointer Arithmetic

Intuition (using **int pointers**):

- **increment**: the address of the next integer in memory
- **addition w/ *x***: the address of the integer at distance *x*
- **subtraction by pointer *x***: the distance (in the size of integers) between the integer pointed to and the integer at address *x*

	Value	Subtraction (w/ pointer) -0x7000	Addition +4	Increment
(1 byte) char ptr	0x7100	0x100	0x7104	0x7101
(4 bytes) int ptr	0x7100	0x40	0x7110	0x7104
(8 bytes) long ptr	0x7100	0x20	0x7120	0x7108

C Programming - Pointers

- Pointers allow us to access variables that are not in scope by passing references around instead of copying the entire content (+ performance)
- Pointers give us fine-grained control over memory (byte-indexed)
- Pointers allow us to manage memory dynamically (more about this later…)

-> is the **arrow operator,** which is syntactic sugar for (*book).title

```
struct Book {
     int fontsize;
     char title[64];
     char content[1024];
};
void update_font(struct Book *book, int size){
     book->fontsize = size;
}
void print_book(struct Book *book){
     printf("Title: %s\n%s",
               book->title, book->content);
}
int main(int argc, char* argv[]){
     struct Book LotR = { /* init book */ };
     printf("Fontsize: %d\n", LotR.fontsize);
     update_font(&LotR, 12);
}
```

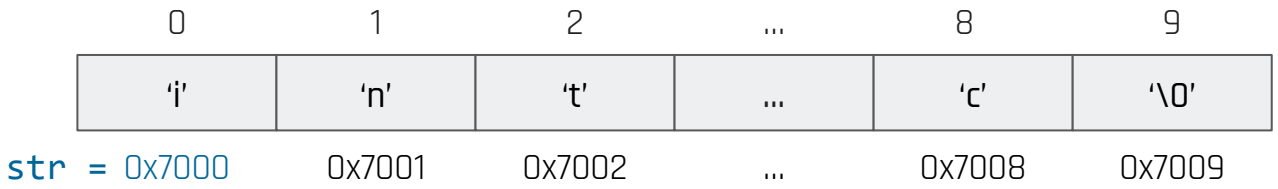
C Programming - Arrays

- Fixed size collection of values of the same type
- Declared and initialized as: `int array[6] = {1, 2, 5, 7, 11, 13};`
- Array elements elements can be accessed by index:
  - array[0] is 1
  - array[3] is 7
- The value of array is the address of the first element (e.g., 0x7000)
  - Can be thought of as a pointer to the first element

	0	1	2	3	4	5
	1	2	5	7	11	13
`array = 0x7000`	0x7000	0x7004	0x7008	0x700c	0x7010	0x7014

C Programming - Strings

- Strings are arrays of characters terminated by a null character, aka, the string terminator ('\0')
- **char** str[9] = "introsec"; or optionally **char** str[] = "introsec";
- Like with arrays, characters can by accessed and modifed by index:
  - str[0] is 'i'
  - str[3] is 'r'



C Programming - String Functions

The C Standard Library (libc) provides a number of utility functions to interact with strings

- **char*** strcat(**char*** dest, **const char*** src)
  - concatenates the string *src* to the string *dest* and returns a pointer to *dest*
- **int** strlen(**const char*** str)
  - returns the length of the string *str* (number of chars up the first null byte excluded)
- **int** strcmp(**const char*** fst, **const char*** snd)
  - returns 0 if the strings are equal, a number <0 if fst smaller than snd, >0 otherwise
- **char*** strcpy(**char*** dest, **const char*** src)
  - copies the string *src* into *dest* and returns a pointer to *dest*

Functions like **strcpy** and **strcat** do not perform length checks on the strings, making them unsafe. Instead, use the **n** alternatives of these functions, which take an extra argument (n) that control how many bytes are written (or read): **strncpy, strncat,** etc.

C Programming - I/O Functions

- **int** getc(**FILE*** stream) :
  - get a character from *stream* (*stdin* for standard input)
- **char*** gets(**char*** str)
  - reads characters into *str* from *stdin* until a newline ('\n') is found
  - gets does not check the length of the string, making it unsafe. **NEVER USE GETS!**
- **char*** fgets(**char*** str, **int** n, **FILE*** stream)
  - reads *n-1* characters into *str* from *stream* or until a newline ('\n') is found
- **int** printf(**const char*** format, ...)
  - builds the *format* string based on the format specifiers and the remaining arguments, and prints it to *stdout*. Returns the number of characters written
  - %d int, %x hex, %s string, %f float, %c character, %p pointer, etc.
- many more...

C Programming - Dynamic Memory

Arrays are fixed size, but what if I don't know how much memory I need at compile time? The C standard library (libc) provides functions for dynamic memory management:

- **void*** malloc(**size_t** size) : allocates memory (*size*) and returns a pointer to it
- **void** free(**void*** ptr) : deallocates memory previously allocated by malloc/calloc/realloc
- **void*** calloc(**size_t** nitems, **size_t** size)
  - allocates memory (*size* nitems*) and returns a pointer to it
  - **tip:** initializes the requested memory with 0s
- **void*** realloc(**void*** ptr, **size_t** size) : attempts to resize (*size*) the memory chunk pointed to by *ptr*

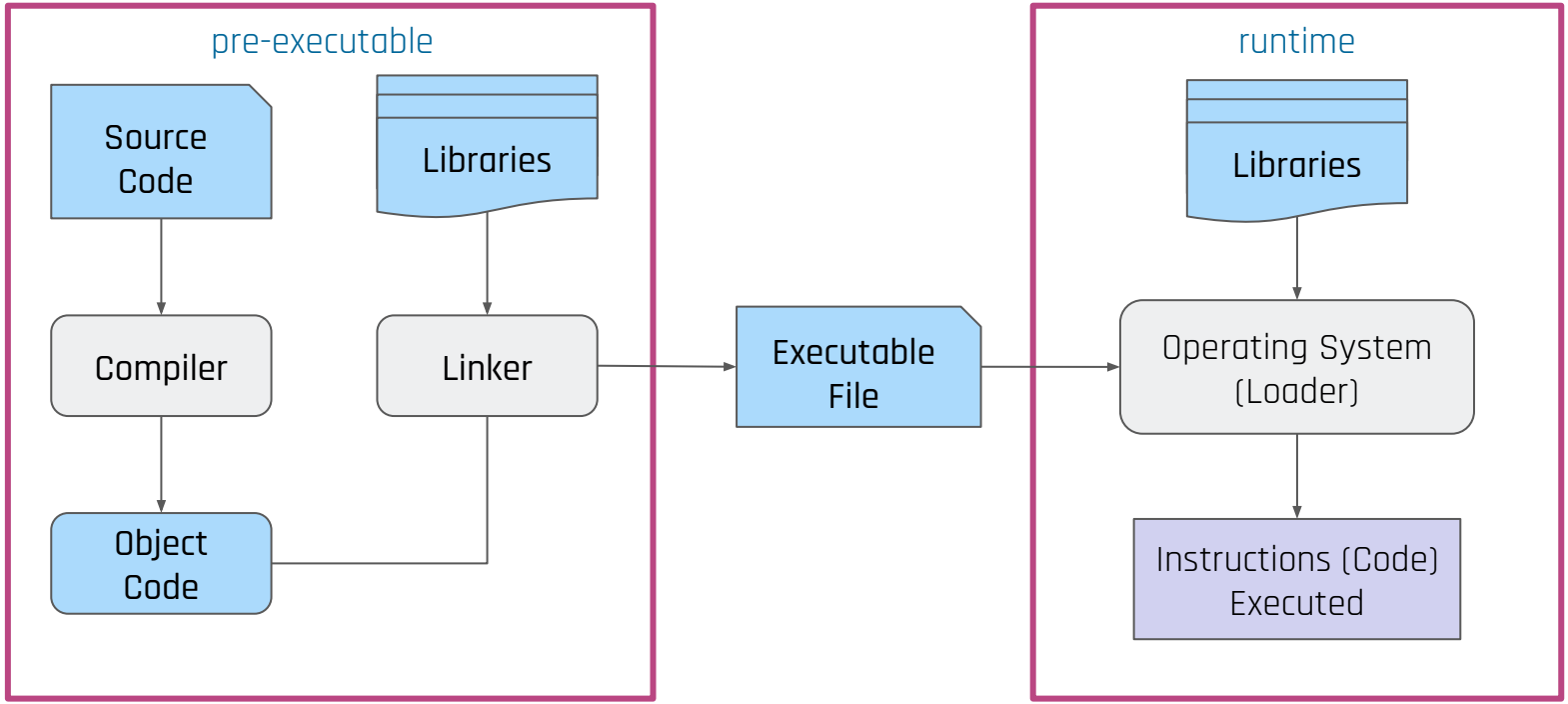
C Programming - Dynamic Memory

Arrays are fixed size, but what if I don't know how much memory I need at compile time?
The C standard library (libc) provides functions for dynamic memory management:

- **void*** malloc(**size_t** size) : allocates memory (*size*) and returns a pointer to it
- **void** free(**v**
  malloc/callo
- **void*** calloc(
  - allocates memory (*size* nitems*) and returns a pointer to it
  - **tip:** initializes the requested memory with 0s
- **void*** realloc(**void*** ptr, **size_t** size) : attempts to resize (*size*) the memory chunk pointed to by *ptr*

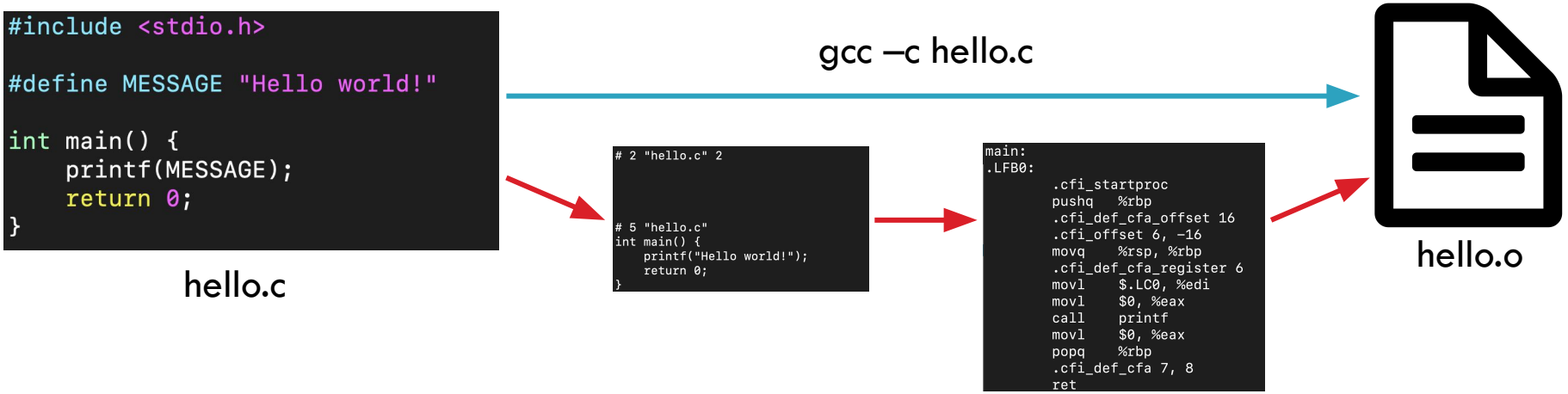
This dynamic memory is stored on the **Heap**, a memory segment **managed by the libc** specifically for memory **requested at runtime.**

Lifecycle of a compiled language (C/C++)



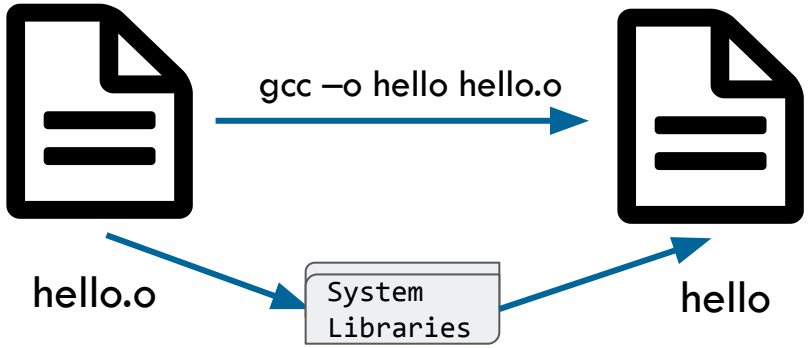
Compilation

A preprocessed source code is translated into **machine or object code**



Linking

- In the last phase (multiple) object files are combined in a single executable
- In the generated file, references (links) to the used library are added



Two approaches can be used in the linking phase

Static Link

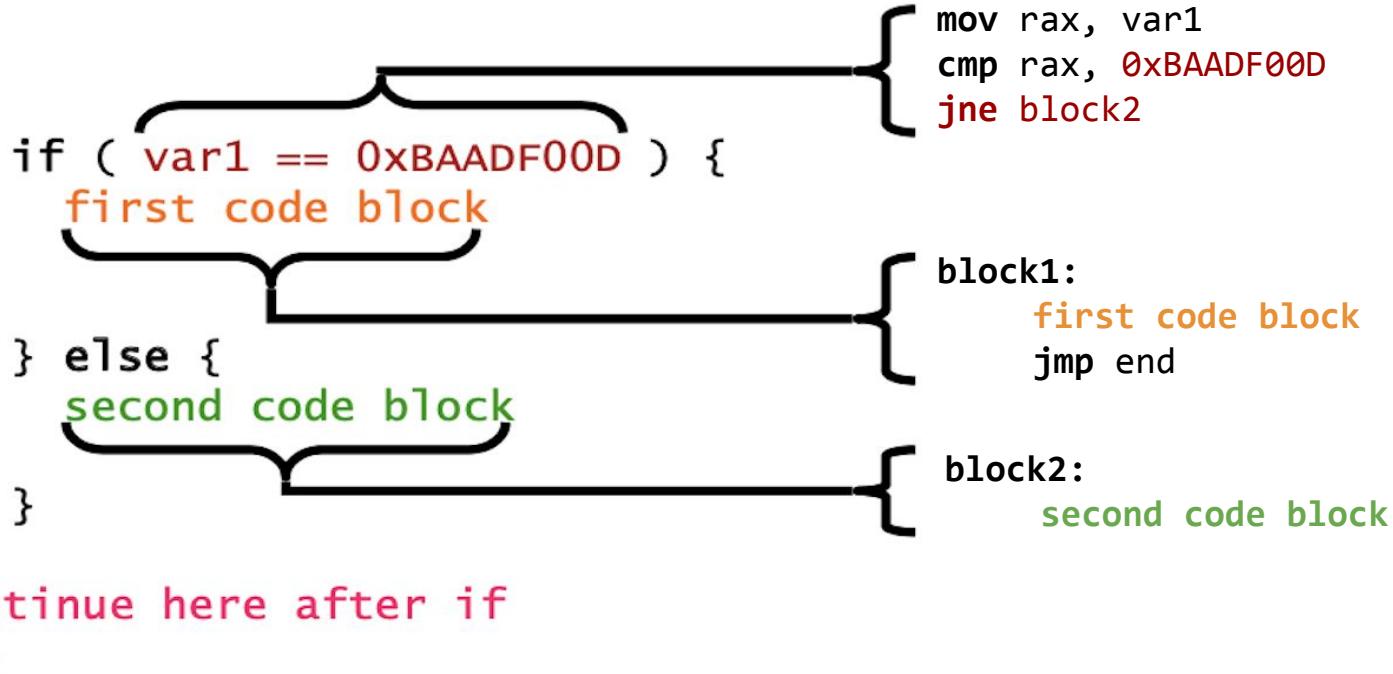
- Binaries are self-contained and do not depend on any external libraries

Dynamic Link

- Binaries rely on system libraries that are loaded when needed

x86-64 Assembly Crash Course

Translation of C/C++ to Assembly



**Assembly language is an abstraction of machine code that is more readable*

Assembly Language Instructions

- An assembly instruction has two components: **operation** and **operand**
- An instruction can have 0-3 operands
- Operands can be:
  - A register
  - A memory location
  - An immediate value
- For example: **mov rax, 0x6754**
  - Operation is mov
  - Operands are register RAX (register) and 0x6754 (immediate value)

Computer Architecture and Assembly Language

- Assembly instruction depends on computer architecture
  - Intel vs. AMD
  - 64-bit vs 32-bit

Intel Syntax:	**AT&T Syntax:**
`command <destination>, <source>`	`command <source>, <destination>`
Example	**Example**
`mov rax, 5`	`mov $5, %rax`
more readable and explicit	default of GNU tools

see http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Types of Assembly Instruction (Operation)

Instructions typically fall into one of three categories:

Data manipulation: Instructions in this category include arithmetic (ADD, SUB), boolean (AND, OR, XOR), bit manipulation (SHR,SHL) commands

Data transfer: Instructions in this category include PUSH/POP, MOV and XCHG

Branching and conditionals: The third category consists of branching and conditional instructions, JMP, CALL, CMP, TEST

```
mov   rsi,r13
mov   edi,r12d
call  QWORD PTR [r15+rbx*8]
add   rbx,0x1
cmp   rbp,rbx
jne   401200 <__libc_csu_init+0x40>
```

Intel Instruction Manual

`mov <dst>, <src>`
moves the <src> value to <dst>

`add <dst>, <src>`
adds the value in <src> to <dst>

`sub <dst>, <src>`
subtracts the value in <src> from <dst>

`and <dst>, <src>`
performs a **logical AND** between <src> and <dst>, placing the result in <dst>

`push <target>`
pushes the value in <target> to the stack

`pop <target>`
pops a value from the stack into <target>

`cmp <dst>, <src>`
compares <src> with <dst>. This is done by subtracting <src> from <dst> and updating flags that can be checked by subsequent conditional operations

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Intel Instruction Manual

`call <address>`

calls the function at <address>. Before jumping to the function, the address of the next instruction is pushed to the stack in order to be able to return

`ret`

pops the return address and **returns** control to it

`leave`

restores the stack frame (rsp←rbp and old rbp is popped)

`jle <target>`

jumps to the address in <target> if the previously compared <src> was **less than or equal** to <dst>. The test is done on the flags set by `cmp`

`jge <target>`

jumps to the address in <target> if the previously compared <src> was **greater than or equal** to <dst>. The test is done on the flags set by `cmp`

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Intel Instruction Manual

`jmp <target>`

jumps to the address in <target>. Copies target address into the RIP/EIP register

`lea <dst>, <src>`

stands for “load effective address”: loads the address of <src> into <dst>

`int <value>`

generates software interrupt<value>. This is commonly used to invoke system calls

`nop`

no-operation, does nothing

NOTE multiple nops directly after each other are called a nop-slide or a nop-sled

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

General Purpose Registers

x86-64 architecture uses the following general-purpose registers to hold code and data

- **RAX**: Used for addition, multiplication, and return values
- **RBX/RDX:** Used for various operations
- **RCX**: Used as a counter
- **RBP**: Used to reference arguments and local variables
- **RSP**: Points to the last item on a stack
- **RSI/RDI**: Used by memory transfer instructions

x86 32-bit equivalent of these registers are EAX, EBX/EDX, ECX, EBP, ESP and ESI/EDI

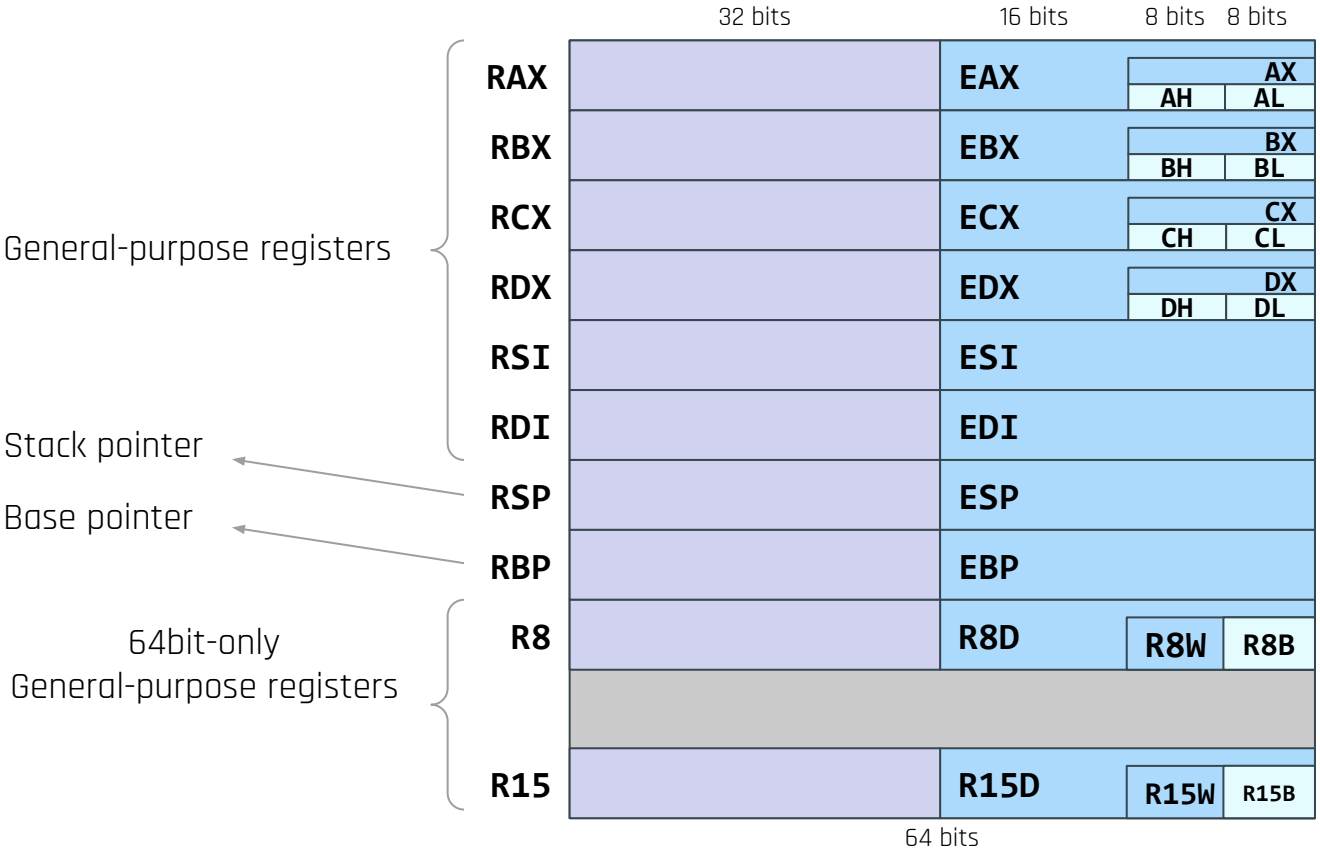
Registers are on chip memory locations used to track the state of computation

Special Purpose Registers

Special-use registers hold flags and track program execution

- **RIP** points to the next instruction to execute
- **EFLAG** bits represent the outcome of computations, and they control certain CPU operation (conditional jumps, e.g., jge)
  - **ZF** - zero flag, when result is zero
  - **CF** - carry flag, result too large
  - **SF** - sign flag, result is negative
- Segment registers include
  - **CS**: Code segment
  - **DS**: Data segment
  - **SS**: Stack segment

Registers: In a nutshell



RIP: Instruction pointer, points to the next instruction

(R/E)FLAGS: status flags
ZF - zero flag, when result is zero
CF - carry flag, result too large
SF - sign flag, result is negative

Addressing modes

- Computer instructions need to be told where to read data from and write data to
- These instructions are communicated by specifying different types of operands
- The different ways of specifying where to read and write data are collectively called **operand addressing modes**

Register direct

```
mov rax, rbx
```

moves the content of rbx into rax

Immediate

```
mov rax, 3
```

move the value 3 into rax

Memory Direct

```
mov rax, [0x1234]
```

move value at address 0x1234 into rax

Memory indirect

```
mov [rax], rbx
```

moves the content of rbx into the memory location in rax.

[rax] : refers to the content rax that stores the address of the destination

Byte Ordering

- A byte consists of 8 bits
- A collection of bytes form a word
- Convention of ordering a byte in a word from left to right or right to left is called **byte ordering**
- Example: Variable *x* has a 4-byte value **`0x0123a675`**

Big Endian: When machines store bytes ordered from most significant byte to least significant

Little Endian: Some machines store bytes ordered from least significant byte to the most significant

Example fragment : Little-endian representation

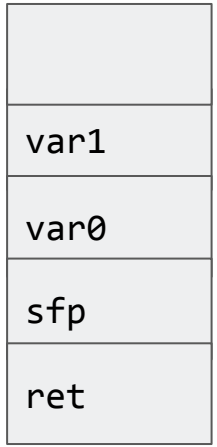
Address	Machine Code	Assembly Rendition
8058345:	5b	pop rbx
8058346:	48 81 c3 fe ca 00 00	add rbx, 0xcafe
805834c:	48 83 f8 28 00 00 00	cmp rax, 0x28

`0xcafe` -> Pad to 32 bits ->`0x0000cafe` -> Split into bytes -> `00 00 ca fe` -> Reverse it -> `fe ca 00 00`

Stack and Calling Convention

Understanding the Stack

- The stack is a section in memory used to store saved registers, local variables and function arguments
- Grows towards lower memory addresses → pushed values have lower addresses
- **PUSH** adds an element, and **POP** removes one
- When a function is called, a **stack frame** is set up
  - RBP/EBP contains the address of the base of the current stack frame
  - RSP/ESP contains the address of the top element of the stack
- **RBP/EBP** (a.k.a. "base pointer (bp) or stack frame pointer (sfp)") serves as a constant reference
- **RSP/ESP** changes with instructions like **PUSH, POP, CALL, LEAVE, RET**



Stack follows Last In, First Out (LIFO)

Understanding Function Calls

- A function is a group of instructions that performs a specific task (for example, read/write files, send network data, log keystrokes)
- A function has three basic components:
  - Input (values passed in)
  - Body (code to perform the task)
  - Return (value passed back)

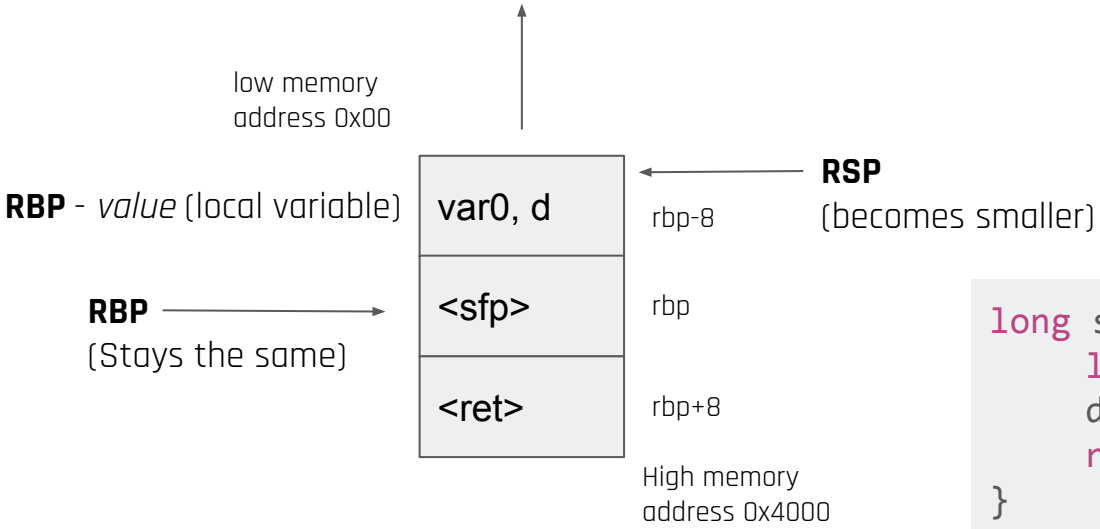
```
int somefunc(int a, int b, int c){
     int d;
     d = a + b + c;
     return d;
}
```

- Calling a function involves a **jump** to another memory location
- After the function is done, execution continues at the instruction after the original function call

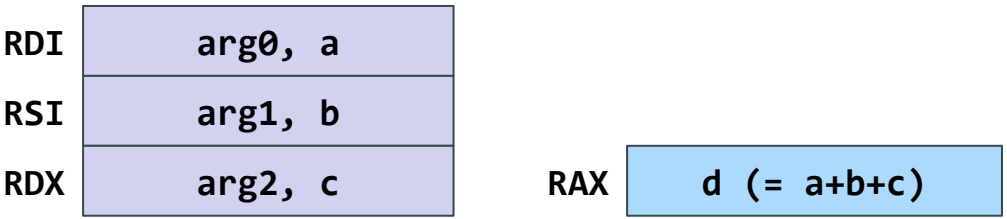
Understanding Call Stacks

- Function format: **`return = function(arg0, arg1,..)`**
- Before calling a function
  - Store function arguments on the registers (RDI, RSI, RDX, RCX, R8, R9), up to 6 arguments
  - The remaining arguments are stored on the stack (first argument in the lower address)
  - Save the **return pointer** on stack (The address of the next instruction after the function call)
  - Transfer control to the function
- When returning from a function:
  - Set up a return value (**typically RAX**)
  - Clean up the stack and restore registers
  - Transfer control to the saved **return pointer**

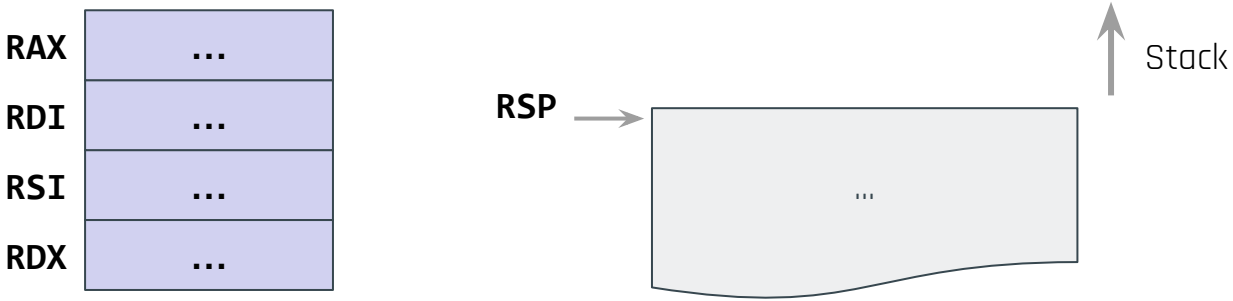
Call Stacks: In a nutshell




```
long somefunc(long a, long b, long c) {
    long d;
    d = a + b + c;
    return d;
}
```

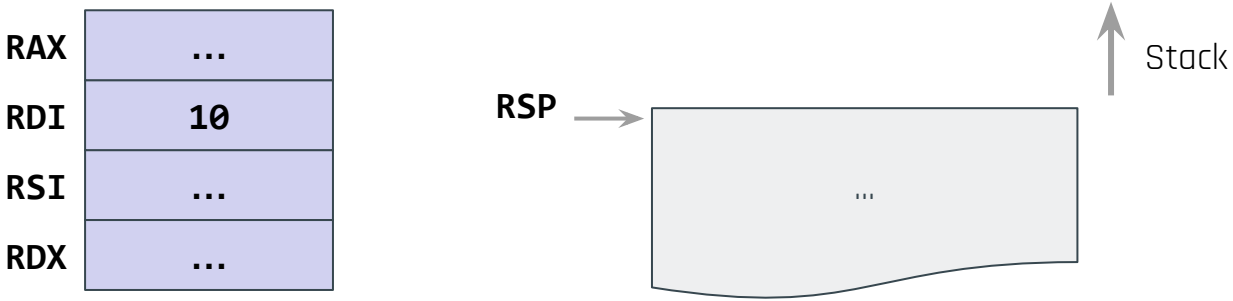


```
func(10);          mov rdi, 10
                   call func    /* push next inst. addr */
                                /* jmp func */
```



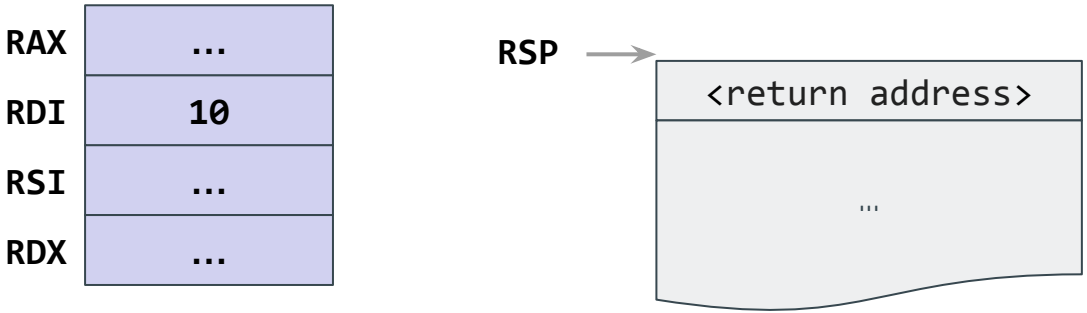
Function calls (64bit)

```
func(10);   →   mov rdi, 10
                call func   /* push next inst. addr */
                            /* jmp func */
```



Function calls (64bit)

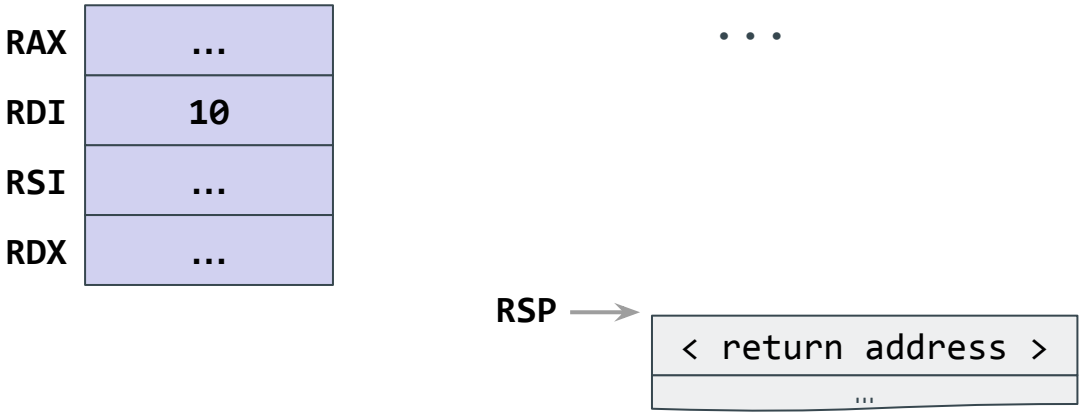
```
func(10);          mov rdi, 10
                   call func    /* push next inst. addr */
                                /* jmp func */
```



Function calls (64bit)

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

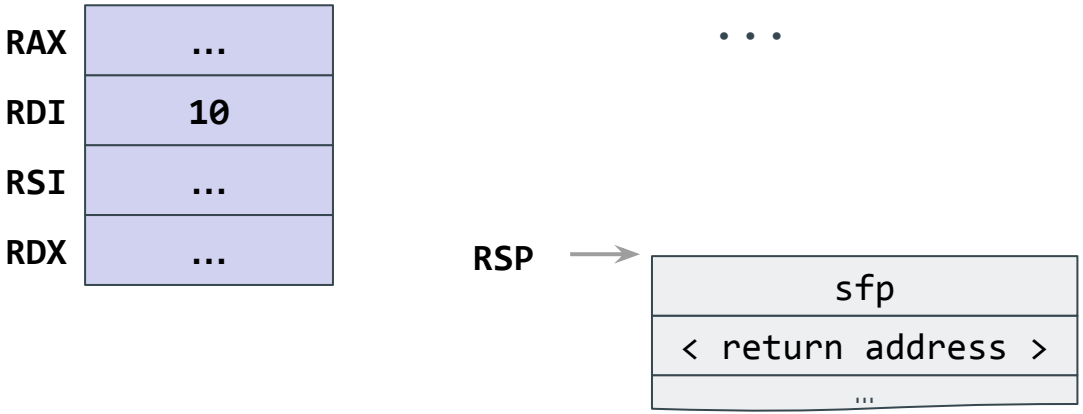
```
push rbp
mov rbp, rsp
sub rsp, 16
mov QWORD PTR [rbp - 8], 0
mov rax, rdi
mov QWORD PTR [rbp - 16], rax
...
```



Function calls (64bit) - Prologue

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

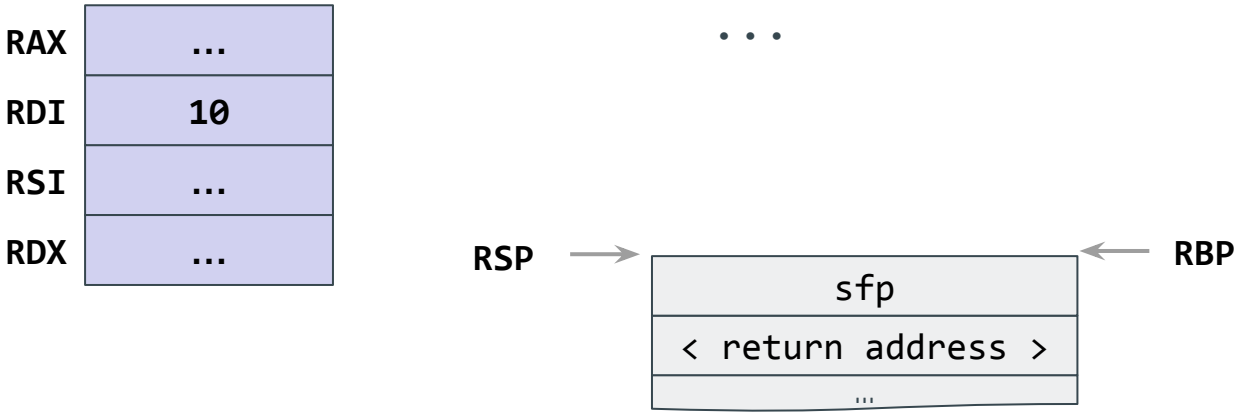
```
push rbp
mov rbp, rsp
sub rsp, 16
mov QWORD PTR [rbp - 8], 0
mov rax, rdi
mov QWORD PTR [rbp - 16], rax
...
```



Function calls (64bit) - Prologue

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

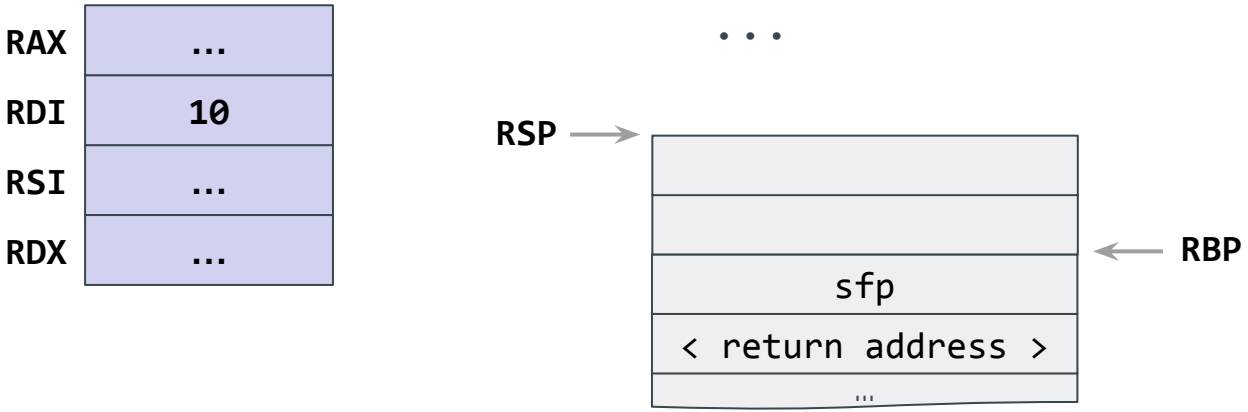
```
push rbp
mov rbp, rsp
sub rsp, 16
mov QWORD PTR [rbp - 8], 0
mov rax, rdi
mov QWORD PTR [rbp - 16], rax
...
```



Function calls (64bit) - Prologue

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

```
push rbp
mov rbp, rsp
sub rsp, 16
mov QWORD PTR [rbp - 8], 0
mov rax, rdi
mov QWORD PTR [rbp - 16], rax
...
```

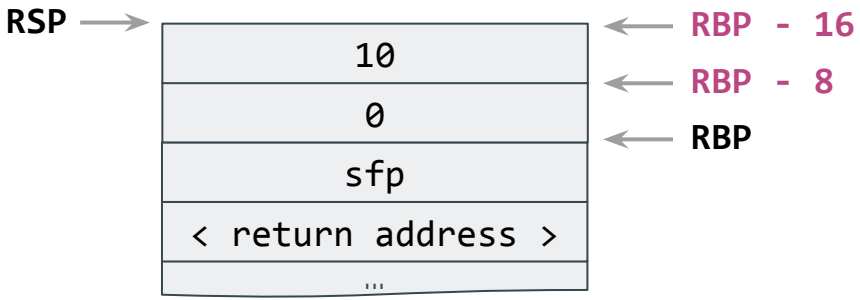


Function calls (64bit) - Prologue

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

```
push rbp
mov rbp, rsp
sub rsp, 16
mov QWORD PTR [rbp - 8], 0
mov rax, rdi
mov QWORD PTR [rbp - 16], rax
...
```

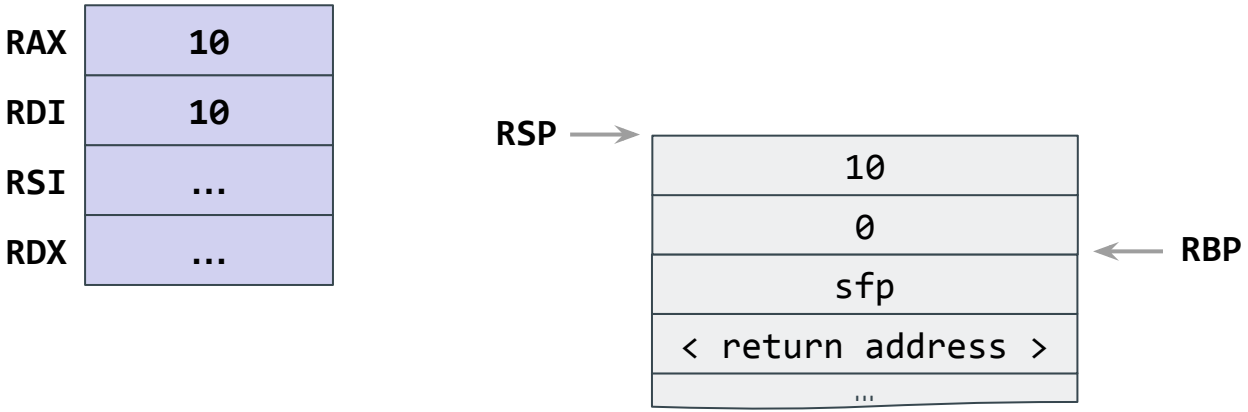
RAX	10
RDI	10
RSI	...
RDX	...



Function calls (64bit)

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

}
```

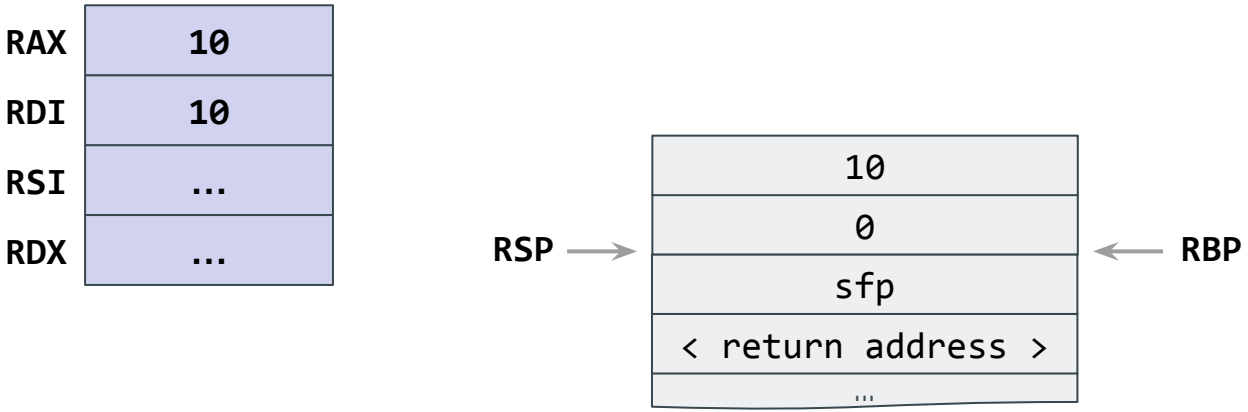
```
...
mov rsp, rbp
pop rbp
ret
```



Function calls (64bit) - Epilogue

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

}
```

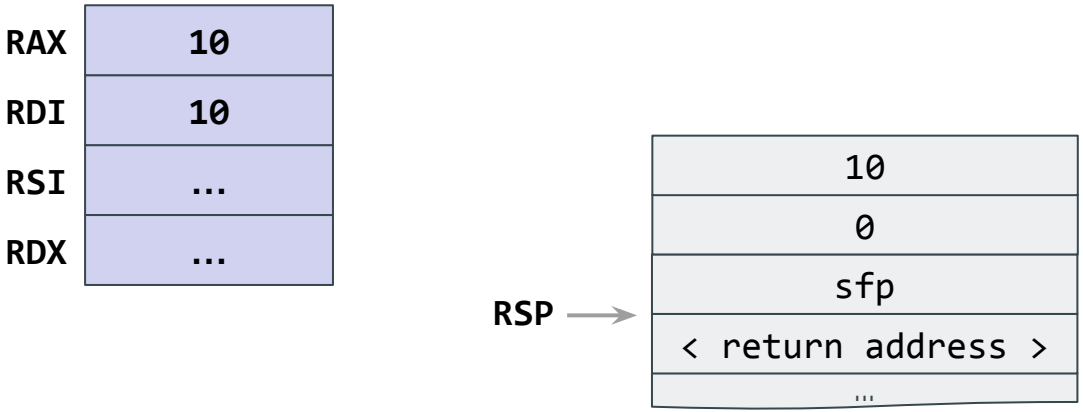
```
...
mov rsp, rbp
pop rbp
ret
```



Function calls (64bit) - Epilogue

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

}
```

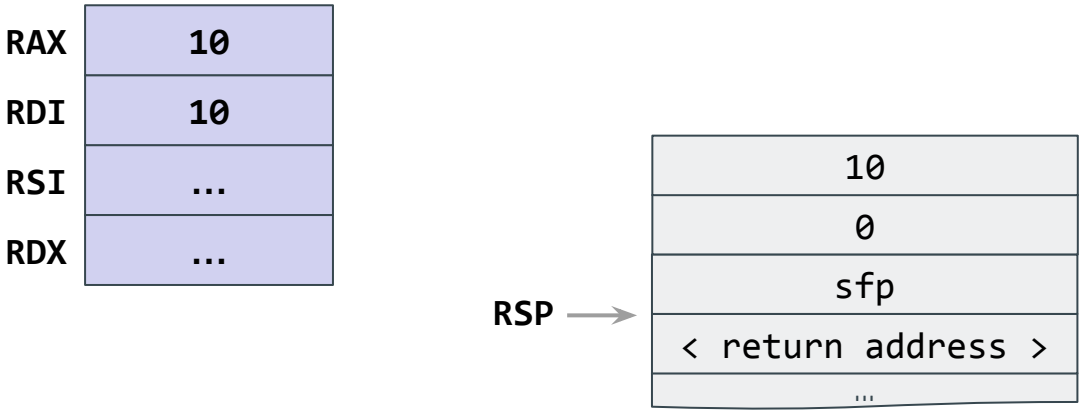
```
...
mov rsp, rbp
pop rbp
ret
```



Function calls (64bit) - Epilogue

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

}
```

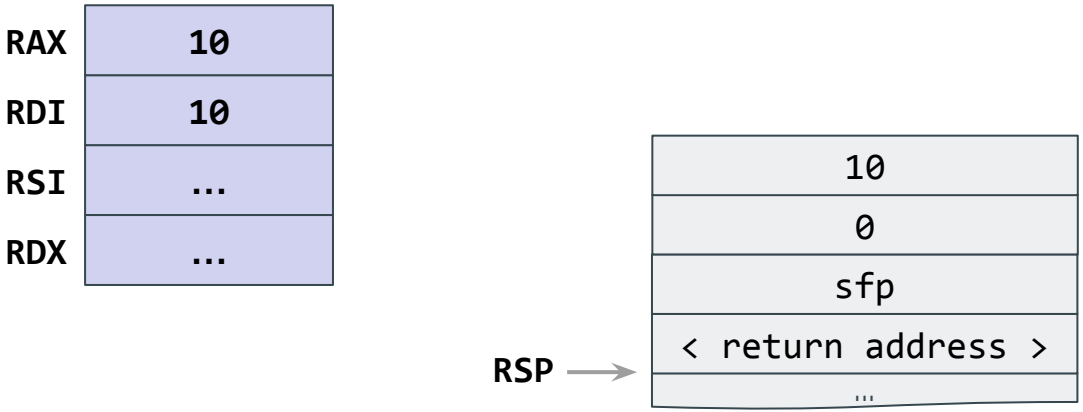
```
...
mov rsp, rbp
pop rbp        /* or leave */
ret
```



Function calls (64bit) - Epilogue

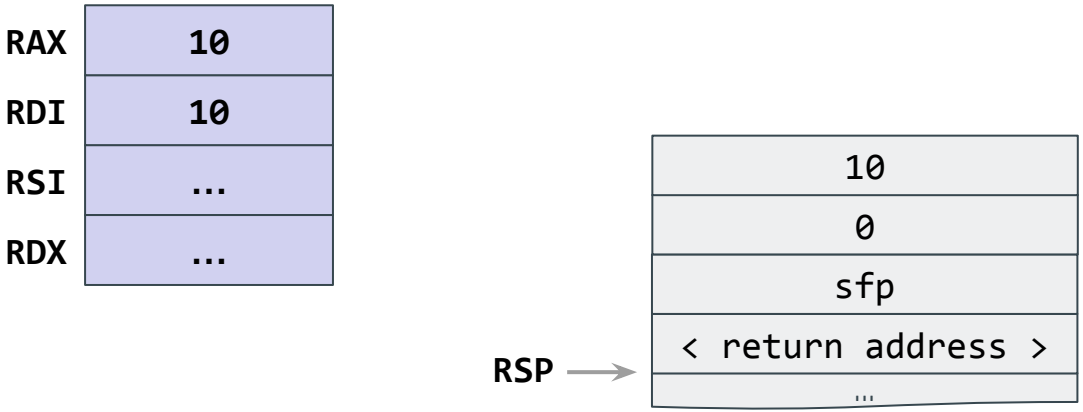
```
long func (long x) {
  long a = 0;
  long b = x;
  ...

}
```

```
...
mov rsp, rbp
pop rbp        /* or leave */
ret
```

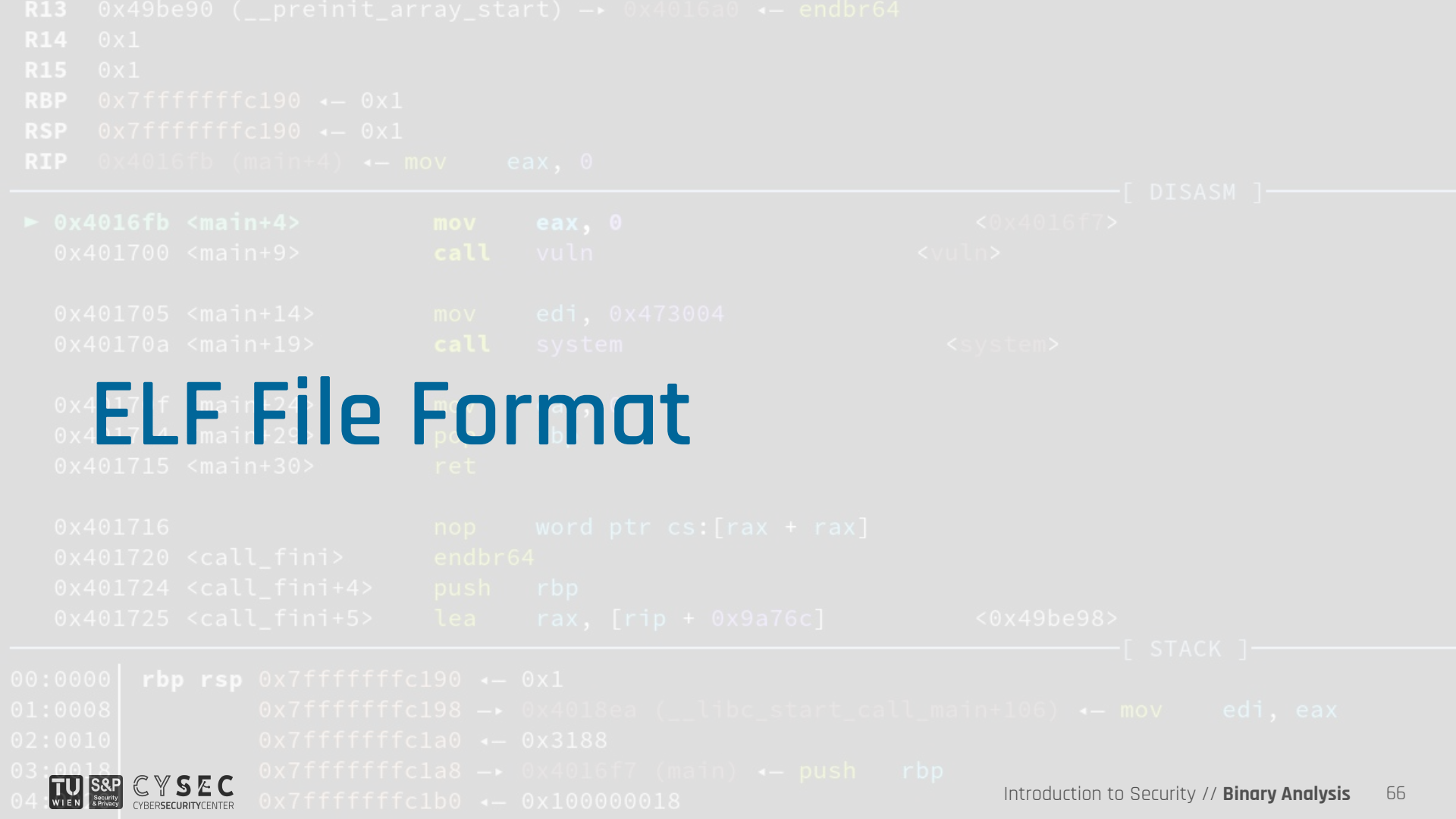


Function calls (64bit) - Epilogue

```
func(10);          mov rdi, 10
                   call func    /* push next inst. addr */
                →  ...          /* jmp func */
```



Function calls (64bit)



ELF File Format

ELF object file format

The **Executable and Linkable Format** (ELF) is a common file format for object files, originally developed and published by UNIX System Laboratories.

There are three types of object files

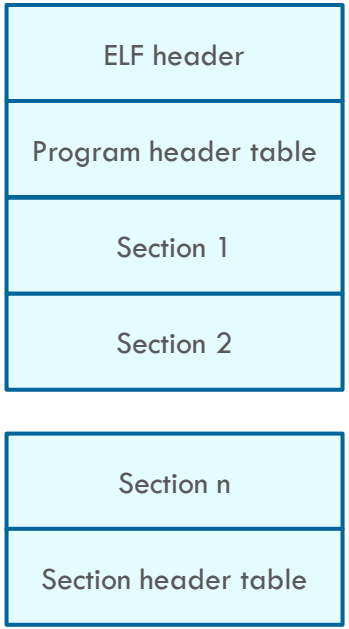
- **Relocatable file** containing code and data that can be linked with other object files to create an executable or a shared object file
- **Executable files** holding a program suitable for execution
- **Shared object files** that can be
  - linked with other relocatable and shared object files to obtain another object file
  - used by a **dynamic linker** together with other executable files and object files to create a **process image**

Object files are binary representation of files intended to execute directly on a processor.

ELF: File Structure

Any ELF file is composed of

- **ELF header** describing the file content
- **Program header table** providing informations on how to create a process image
- sequence of **Sections** containing what is needed for linking (instructions, data, symbol table, relocation information, ...)
- **Section header table** with a description of previous sections



ELF: Relevant Sections

.text contains the executable instructions of a program

.bss contains uninitialised data that contribute to the program's memory image

.data contain initialized data that contribute to the program's memory image

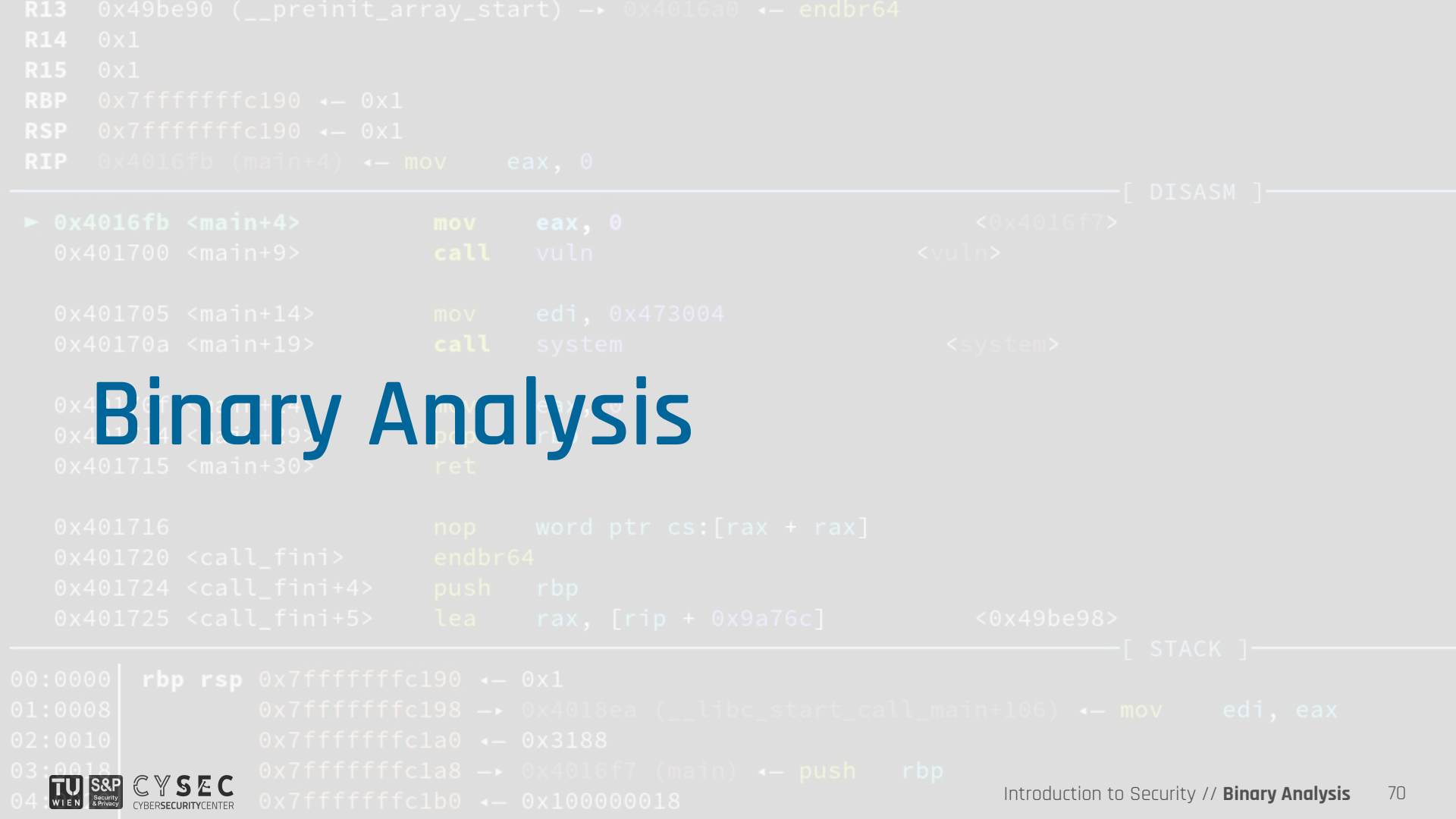
.rodata is similar to .data, but refers to read only data

.symtab contains the program's symbol table

.plt Procedure Linkage Table (PLT) with information necessary for calling functions from shared libraries

.got Global Offset Table (GOT) holds information used by the PLT to resolve the addresses of shared library functions dynamically (at runtime)

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000020  0000000000000338  0000000000000338  00000338  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000358  0000000000000358  00000358  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .note.ABI-tag 00000020  000000000000037c  000000000000037c  0000037c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.hash     00000024  00000000000003a0  00000000000003a0  000003a0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynsym       000000f0  00000000000003c8  00000000000003c8  000003c8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .dynstr       000000ab  00000000000004b8  00000000000004b8  000004b8  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version  00000014  0000000000000564  0000000000000564  00000564  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .gnu.version_r 00000030  0000000000000578  0000000000000578  00000578  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rela.dyn     000000c0  00000000000005a8  00000000000005a8  000005a8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .rela.plt     00000060  0000000000000668  0000000000000668  00000668  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 11 .init         0000001b  0000000000001000  0000000000001000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .plt          00000050  0000000000001020  0000000000001020  00001020  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .plt.got      00000010  0000000000001070  0000000000001070  00001070  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .plt.sec      00000040  0000000000001080  0000000000001080  00001080  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .text         00000255  00000000000010c0  00000000000010c0  000010c0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 16 .fini         0000000d  0000000000001318  0000000000001318  00001318  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 17 .rodata       0000004b  0000000000002000  0000000000002000  00002000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 18 .eh_frame_hdr 0000004c  000000000000204c  000000000000204c  0000204c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 19 .eh_frame     00000128  0000000000002098  0000000000002098  00002098  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 20 .init_array   00000008  0000000000003da0  0000000000003da0  00002da0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 21 .fini_array   00000008  0000000000003da8  0000000000003da8  00002da8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 22 .dynamic      000001f0  0000000000003db0  0000000000003db0  00002db0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 23 .got          00000060  0000000000003fa0  0000000000003fa0  00002fa0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 24 .data         00000010  0000000000004000  0000000000004000  00003000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 25 .bss          00000008  0000000000004010  0000000000004010  00003010  2**0
                  ALLOC
```



Binary Analysis

Gathering Information from Binary Files

Several tools are available to extract information from an **ELF** file

- `objdump` Displays information about object files
- `readelf` Displays information about ELF files
- `strings` Displays strings and printable characters in a file
- `file` Determines file type and displays some general info
- `ldd` Displays shared object dependencies

These tools can be used to gather information from binaries **without executing them**: they **statically** inspect the structure of the file

Static vs. Dynamic Analysis

Programs can be analysed in two ways

- **Static analysis**
  by inspecting the assembly we try to understand the program logic (tools can infer the program control flow effectively)

- **Dynamic analysis**
  the program is run with **debuggers** (on virtual or real processors) to observe its dynamic behaviour (for example, malware executed in sandboxes)

Usually the two techniques complement each other

Several **dynamic analysis tools** are available

- **gdb** The GNU project debugger
- **strace** Trace system calls and signals
- **ltrace** Trace library calls

see https://wizardzines.com/zines/strace/

Disassembly

```
$ objdump -M intel -d {{path_to_your_binary}}
0804843b <main>:
```

Addresses
(may be relative / relocatable addresses)

The actual **machine code** as bytes. Note that commands may have **different lengths**

Assembly in Intel Syntax

see https://tldr.ostera.io/objdump

Example Time!

Example binaries are located at on the testbed server under **'/exercises/password_manager'**

With everything you know so far, you can answer these questions *without executing the program itself:*

- Is it a 32 bit or 64 bit executable?
- What libraries are linked?
- What is the address of the main function?
- What are the most likely messages it will print when you execute the program?

Static Analysis

file

```
λ juno password_manager → file password_manager
password_manager: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
 interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d4bd820641be7bb8adf408108e4d626ebf
ee2281, for GNU/Linux 3.2.0, stripped
```

64-bit binary

ldd

```
λ juno password_manager → ldd password_manager
        linux-vdso.so.1 (0x00007ffc499db000)
        libc.so.6 => /usr/lib/libc.so.6 (0x000076994e24e000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x000076994e469000)
```

Static Analysis

readelf

```
λ juno password_manager → readelf -h password_manager
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401050
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13552 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

```
λ juno password_manager → readelf -s password_manager

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _[...]@GLIBC_2.34 (2)
     2: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     6: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]

Symbol table '.symtab' contains 25 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS password_manager.c
     2: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
     3: 0000000000403df8     0 OBJECT  LOCAL  DEFAULT   21 _DYNAMIC
     4: 000000000040205c     0 NOTYPE  LOCAL  DEFAULT   17 __GNU_EH_FRAME_HDR
     5: 0000000000403fe8     0 OBJECT  LOCAL  DEFAULT   23 _GLOBAL_OFFSET_TABLE_
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_mai[...]
     7: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     8: 0000000000404010     0 NOTYPE  WEAK   DEFAULT   24 data_start
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5
    10: 0000000000404020     0 NOTYPE  GLOBAL DEFAULT   24 _edata
    11: 00000000004011c8     0 FUNC    GLOBAL HIDDEN    15 _fini
    12: 0000000000404010     0 NOTYPE  GLOBAL DEFAULT   24 __data_start
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND strcmp@GLIBC_2.2.5
    14: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    15: 0000000000404018     0 OBJECT  GLOBAL HIDDEN    24 __dso_handle
    16: 0000000000404060     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    17: 0000000000404028     0 NOTYPE  GLOBAL DEFAULT   25 _end
    18: 0000000000401080     5 FUNC    GLOBAL HIDDEN    14 _dl_relocate_sta[...]
    19: 0000000000401050    38 FUNC    GLOBAL DEFAULT   14 _start
    20: 0000000000404020     0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
    21: 0000000000401136   143 FUNC    GLOBAL DEFAULT   14 main
    22: 0000000000404020     0 OBJECT  GLOBAL HIDDEN    24 __TMC_END__
    23: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
    24: 0000000000401000     0 FUNC    GLOBAL HIDDEN    12 _init
```

address of **main**

Static Analysis

strings

From the result of **strings**, the program likely asks for a password and checks if it is correct.

You can try different parameters, for example, with `-n 2`, we can set the minimum string length to two

```
λ juno password_manager → strings password_manager
/lib64/ld-linux-x86-64.so.2
puts
__libc_start_main
strcmp
ibc.so.6
LIBC_2.2.5
LIBC_2.34
ITM_deregisterTMCloneTable
_gmon_start__
TM_registerTMCloneTable
1
= @@
passwordH
Enter the password as argument
Correct password!
Wrong password!
Usage: <password>
;*3$"
GCC: (GNU) 13.2.1 20230801
password_manager.c
_DYNAMIC
__GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
__libc_start_main@GLIBC_2.34
_ITM_deregisterTMCloneTable
puts@GLIBC_2.2.5
_edata
_fini
__data_start
```

Disassembly

Given a binary file, we can use a disassembler to extract info about the executed code

- This can be done with `objdump`:

  ```
  objdump -M intel -ds ./password_manager > pm.s
  ```

  Here we ask `objdump` to produce the assembly code (`-d`) and display sections (`-s`) in Intel syntax (`-M intel`) and put the result in the file `pm.s`

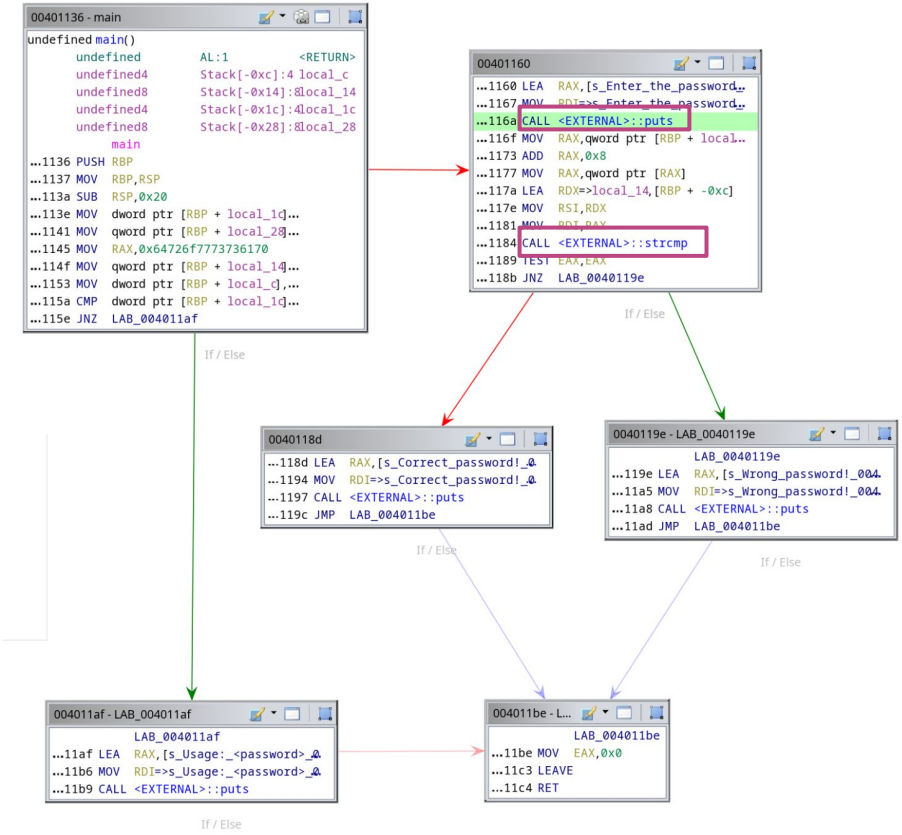
Disassembly

Scrolling through the main function disassembly in we see calls to `<strcmp@plt>`, and `<puts@plt>`

We also find four jump instructions:

- `jne  40119e <main+0x68>`
- `jnz  4011af <main+0x79>`
- `2 x jmp  4011be <main+0x88>`

```
0000000000401136 <main>:
  401136:       55                      push   rbp
                                        ...
  40117e:       48 89 c7                mov    rdi,rax
  401181:       e8 ba fe ff ff          call   401040 <strcmp@plt>
  401186:       85 c0                   test   eax,eax
  401188:       75 0e                   jne    401198 <main+0x62>
  40118a:       48 8d 3d 96 0e 00 00    lea    rdi,[rip+0xe96]        # 40202
  401191:       e8 9a fe ff ff          call   401030 <puts@plt>
  401196:       eb 1a                   jmp    4011b2 <main+0x7c>
  401198:       48 8d 3d 9a 0e 00 00    lea    rdi,[rip+0xe9a]        # 40203
```



Strace

Traces system calls

`strace ./password_manager`

In this case, it is not that useful.

But, now we know that we need to supply the password as an argument

```
λ juno password_manager → strace ./password_manager
execve("./password_manager", ["./password_manager"], 0x7ffc9a952a90 /* 66 vars */) = 0
brk(NULL)                               = 0x22a3000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=223347, ...}) = 0
mmap(NULL, 223347, PROT_READ, MAP_PRIVATE, 3, 0) = 0x756e0dba3000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220^\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0\0@\0\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
fstat(3, {st_mode=S_IFREG|0755, st_size=1948952, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x756e0dba1000
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0\0@\0\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 1973104, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x756e0d9bf000
mmap(0x756e0d9e3000, 1421312, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x24000) = 0x756e0d9e3000
mmap(0x756e0db3e000, 348160, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x17f000) = 0x756e0db3e000
mmap(0x756e0db93000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1d3000) = 0x756e0db93000
mmap(0x756e0db99000, 31600, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x756e0db99000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x756e0d9bc000
arch_prctl(ARCH_SET_FS, 0x756e0d9bc740) = 0
set_tid_address(0x756e0d9bca10)         = 135825
set_robust_list(0x756e0d9bca20, 24)     = 0
rseq(0x756e0d9bd060, 0x20, 0, 0x53053053) = 0
mprotect(0x756e0db93000, 16384, PROT_READ) = 0
mprotect(0x403000, 4096, PROT_READ)     = 0
mprotect(0x756e0dc0d000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x756e0dba3000, 223347)          = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
getrandom("\xd8\x70\xc7\x92\xae\x49\x42\x24", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x22a3000
write(1, "Usage: <password>\n", 18Usage: <password>
)       = 18
+++ exited with 0 +++
```

Ltrace

Traces library calls

```
ltrace ./password_manager asdf
```

```
λ juno password_manager → ltrace ./password_manager asdf
puts("Enter the password as argument"Enter the password as argument
)                                    = 31
strcmp("asdf", "password_1!")                                    = -15
puts("Wrong password!"Wrong pas ord!
)                                                                = 16
+++ exited (status 0) +++
```

There is the right password :)

Debugging - GDB Cheat Sheet

Gdb is the **G**NU **D**e**B**ugger. Gdb executes commands, like a terminal:

- **help** - displays the build-in help if you want to know more
- **quit** - exits gdb. the most important, best when you were successful!
- **run** - runs the loaded program as if you did from the command line
- **break** <where> - sets a breakpoint at an address.
  - When the execution reaches the address where the breakpoint was set, execution is frozen and you can interact with gdb again
- **continue** - resumes execution
- **info** <what> - displays information, most commonly about registers or the stack frame
  - info register
  - info frame
- **x**/<nuf> <what> - print memory. n=how many, u=unit, f=format character
  - x/1gx $rax -> print the 1 double word as a hexadecimal from the address pointed to by rax

Debugging - GDB Cheat Sheet

- **disassemble** <func> - disassembles the argument (<func>) function
- **print** (p) - prints the value of an expression
- **set** - to change settings of gdb or set values inside the program memory
  - set disassembly-flavor intel - output the disassembly code in Intel syntax
- **nexti** (ni) - step over the next instruction
- **stepi** (si) - step into the next instruction (follows function calls)

Debugging - GDB Demo

- We will use **pwndbg**[1]**,** a GDB plugin designed to help in reverse engineering and exploit development
- All the GDB commands work with **pwndbg**, but it offers extra useful commands and a nicer user experience
- Features:
  - **context** - summarizes the current execution context (registers, stack, code)
  - **disassembly** - displays extract memory targets and condition codes
  - **telescope** - displays memory dumps and recursively dereferences pointers
  - **search** - searches for bytes, strings, integers values or pointers in the memory space

(1) https://github.com/pwndbg/pwndbg

Debugging - GDB Demo

Let us continue with debugging and the `password_manager_v2` binary

- Load the binary into the debugger with:

  `gdb ./password_manager_v2`

  After a header you are presented with a prompt, like the terminal

Note:

1. On testbed, you might need to set a local variable with the locale for pwndbg to work. Run it like this:
   `$ C_CTYPE=C.UTF-8 gdb </path/to/binary>`
2. To enable pwndbg, add this line to the .gdbinit file in your home directory:
   `"source /opt/pwndbg/gdbinit.py"`

Debugging - GDB Demo

GDB can also disassemble.
Let's disassemble **main**!

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x0000000000401156 <+0>:     push   rbp
   0x0000000000401157 <+1>:     mov    rbp,rsp
   0x000000000040115a <+4>:     push   rbx
   0x000000000040115b <+5>:     sub    rsp,0x38
   0x000000000040115f <+9>:     movabs rax,0x796763636b7a6977
   0x0000000000401169 <+19>:    mov    QWORD PTR [rbp-0x23],rax
   0x000000000040116d <+23>:    mov    WORD PTR [rbp-0x1b],0x6e62
   0x0000000000401173 <+29>:    mov    BYTE PTR [rbp-0x19],0x0
   0x0000000000401177 <+33>:    lea    rdi,[rip+0xe86]        # 0x402004
   0x000000000040117e <+40>:    call   0x401030 <puts@plt>
   0x0000000000401183 <+45>:    lea    rax,[rbp-0x40]
   0x0000000000401187 <+49>:    mov    rsi,rax
   0x000000000040118a <+52>:    lea    rdi,[rip+0xe86]        # 0x402017
   0x0000000000401191 <+59>:    mov    eax,0x0
                                call   0x401060 <__isoc99_scanf@plt>
                                mov    DWORD PTR [rbp-0x18],0x0
```

There is a constant being saved on the stack
[rbp-0x23] = "wizkccgybn\0"

string input read by **scanf** is being saved at [rbp-0x40]

Debugging - GDB Demo

- [rbp-0x40] - stores our input
- [rbp-0x23] - stores the comparison target. Is this the password we want?

```
04011e3 <+141>:   lea    rdx,[rbp-0x40]
04011e7 <+145>:   lea    rax,[rbp-0x23]
04011eb <+149>:   mov    rsi,rdx
04011ee <+152>:   mov    rdi,rax
04011f1 <+155>:   call   0x401050 <strcmp@plt>
04011f6 <+160>:   test   eax,eax
04011f8 <+162>:   jne    0x401208 <main+178>
04011fa <+164>:   lea    rdi,[rip+0xe1b]        # 0x40201c
0401201 <+171>:   call   0x401030 <puts@plt>
0401206 <+176>:   jmp    0x401214 <main+190>
0401208 <+178>:   lea    rdi,[rip+0xe1f]        # 0x40202e
040120f <+185>:   call   0x401030 <puts@plt>
0401214 <+190>:   mov    eax,0x0
0401219 <+195>:   add    rsp,0x38
040121d <+199>:   pop    rbx
040121e <+200>:   pop    rbp
040121f <+201>:   ret
er dump.
```

Later, [rbp-0x40] is compared to [rbp-0x23] with **strcmp**

Debugging - GDB Demo

Let's set some breakpoints!

after **scanf**

```
pwndbg> b *main+69
Breakpoint 1 at 0x40119b
pwndbg> b *main+155
Breakpoint 2 at 0x4011f1
pwndbg> run
Starting program: /home/student5/pm/password_manager_v2
Enter the password
aaaaaaaa
```

at **strcmp**

run and provide a password

1st breakpoint hit (after **strcmp**)

```
Breakpoint 1, 0x000000000040119b in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────[ REGISTERS / show-flags off / show-com
*RBP  0x7fffffffe4d0 ◂— 0x0
*RSP  0x7fffffffe490 ◂— 'aaaaaaaa'
*RIP  0x40119b (main+69) ◂— mov dword ptr [rbp - 0x18], 0
──────────────────────────────[ DISASM / x86-64 / set emulat
 ► 0x40119b <main+69>     mov    dword ptr [rbp - 0x18], 0
   0x4011a2 <main+76>     jmp    main+121                    <main+121>
    ↓
   0x4011cf <main+121>    mov    ebx, dword ptr [rbp - 0x18]
   0x4011d2 <main+124>    lea    rax, [rbp - 0x40]
   0x4011d6 <main+128>    mov    rdi, rax
   0x4011d9 <main+131>    call   strlen@plt                  <strlen@plt>

   0x4011de <main+136>    cmp    rbx, rax
   0x4011e1 <main+139>    jb     main+78                     <main+78>

   0x4011e3 <main+141>    lea    rdx, [rbp - 0x40]
   0x4011e7 <main+145>    lea    rax, [rbp - 0x23]
   0x4011eb <main+149>    mov    rsi, rdx
──────────────────────────────[ STACK ]
00:0000| rsp 0x7fffffffe490 ◂— 'aaaaaaaa'
                  0x7fffffffe498 —▸ 0x401200 (main+170) ◂— add al, ch
                  0x7fffffffe4a0 —▸ 0x7ffff7fc22e8 (__exit_funcs_lock) ◂— 0x0
                  0x7fffffffe4a8 ◂— 0x7a69770000401220
                  0x7fffffffe4b0 ◂— 0x6e62796763636b /* 'kccgybn' */
05:0028|          0x7fffffffe4b8 —▸ 0x401070 (_start) ◂— endbr64
06:0030|          0x7fffffffe4c0 —▸ 0x7fffffffe5c0 ◂— 0x1
07:0038|          0x7fffffffe4c8 —▸ 0x401220 (__libc_csu_init) ◂— endbr64
──────────────────────────────[ BACKTRACE ]
 ► f 0         0x40119b main+69
   f 1   0x7ffff7df5083 __libc_start_main+243
```

our input at [rbp-0x40]

Debugging - GDB Demo

After stepping through a few instructions (with `ni`) we reach the following code:

1. call to **strlen** on [rbp-0x40] which is our input
2. comparison with [rbp-0x18], currently set to 0
3. `jb` instruction, that jumps back to **main+78** if rbp[0x18] < strlen([rbp-0x40])

Looks like a **for loop**

```
0x4011a2 <main+76>     jmp    main+121                        <ma
 ↓
0x4011cf <main+121>    mov    ebx, dword ptr [rbp - 0x18]
0x4011d2 <main+124>    lea    rax, [rbp - 0x40]
0x4011d6 <main+128>    mov    rdi, rax
0x4011d9 <main+131>  1 call   strlen@plt                       <

0x4011de <main+136>  2 cmp    rbx, rax
0x4011e1 <main+139>    jb     main+78  3                      <mai
 ↓
0x4011a4 <main+78>     mov    eax, dword ptr [rbp - 0x18]
0x4011a7 <main+81>     movzx  eax, byte ptr [rbp + rax - 0x40]
0x4011ac <main+86>     movsx  eax, al
0x4011af <main+89>     add    eax, 0xa
```

Debugging - GDB Demo

[rbp-0x18] is also used to index our input string. Must be the iterator! Let's call it `i`

```
   0x4011a4 <main+78>     mov    eax, dword ptr [rbp - 0x18]
   0x4011a7 <main+81>     movzx  eax, byte ptr [rbp + rax - 0x40]
   0x4011ac <main+86>     movsx  eax, al
   0x4011af <main+89>     add    eax, 0xa
   0x4011b2 <main+92>     mov    dword ptr [rbp - 0x14], eax
►  0x4011b5 <main+95>     cmp    dword ptr [rbp - 0x14], 0x7a
   0x4011b9 <main+99>     jle    main+105                    <m

                ain+105>  mov    eax, dword ptr [rbp - 0x14]
               <main+108>  mov    edx, eax
               <main+110>  mov    eax, dword ptr [rbp - 0x18]
               <main+113>  mov    byte ptr [rbp + rax - 0x40], dl
```

0xa is added to `input[i]` in a local variable [rbp-0x14] (let's call it `current`) and then compared with 0x7a ('z' character)

if `current` <= 'z', then we go to main+105 where the `input[i]` is set to `current`

Debugging - GDB Demo

However, if `current` is bigger than 0x7a, it will be subtracted by 0x1a instead (at <main+101>). The for loop appears to be implementing a **rotate right** operation by 10.

`i` is incremented and the `if guard` is executed again. The loop body is repeated until we reach the end of our input.

```
04011a7 <+81>:    movzx  eax,BYTE PTR [rbp+rax*1-0x40]
04011ac <+86>:    movsx  eax,al
011af <+89>:      add    eax,0xa
011b2 <+92>:      mov    DWORD PTR [rbp-0x14],eax
011b5 <+95>:      cmp    DWORD PTR [rbp-0x14],0x7a
11b9 <+99>:       jle    0x4011bf <main+105>
bb <+101>:        sub    DWORD PTR [rbp-0x14],0x1a
011bf <+105>:     mov    eax,DWORD PTR [rbp-0x14]
04011c2 <+108>:   mov    edx,eax
04011c4 <+110>:   mov    eax,DWORD PTR [rbp-0x18]
04011c7 <+113>:   mov    BYTE PTR [rbp+rax*1-0x40],dl
                  add    DWORD PTR [rbp-0x18],0x1
                  mov    ebx,DWORD PTR [rbp-0x18]
                  lea    rax,[rbp-0x40]
                  mov    rdi,rax
                  call   0x401040 <strlen@plt>
                  cmp    rbx,rax
04011e1 <+139>:   jb     0x4011a4 <main+78>
```

Debugging - GDB Demo

If we continue until the last breakpoint (at strcmp), we can see our input shifted by 10 is being compared to "wizkccbgybn"

```
► 0x4011f1 <main+155>    call   strcmp@plt
      s1: 0x7fffffffe4ad ◂— 'wizkccgybn'
      s2: 0x7fffffffe490 ◂— 'kkkkkkkk'

  0x4011f6 <main+160>    test   eax, eax
  0x4011f8 <main+162>    jne    main+178

  0x4011fa <main+164>    lea    rdi, [rip + 0xe1b]
  0x401201 <main+171>    call   puts@plt

  0x401206 <main+176>    jmp    main+190

  0x401208 <main+178>    lea    rdi, [rip + 0xe1f]
  0x40120f <main+185>    call   puts@plt
```

If we perform the reverse operation (rotate left), on the target string, we should obtain the correct password!

Debugging - GDB Demo

Rotating “wizkccbgybn” 10 charcaters to the left, we obtain the string “mypassword”.

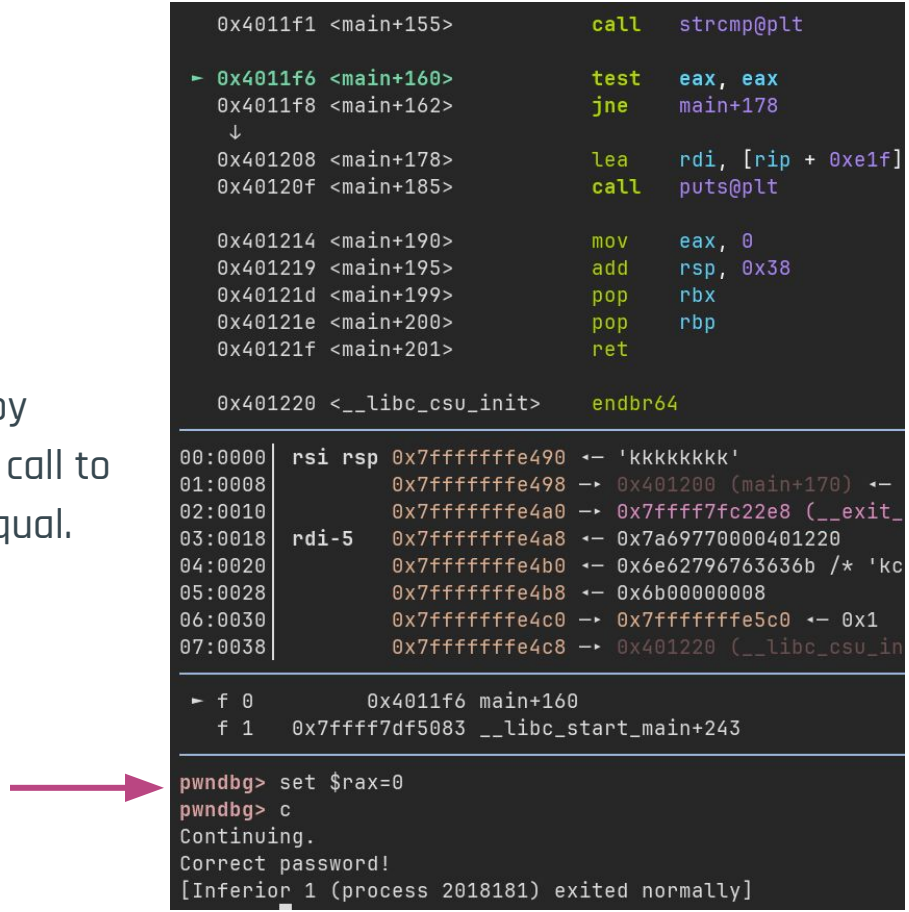
Let’s try it!

IT WORKS!

```
pwndbg> r
Starting program: /home/student5/pm/password_manager_v2
Enter the password
mypassword
Correct password!
[Inferior 1 (process 2012188) exited normally]
pwndbg>
```

Debugging - GDB Demo

It is also possible to bypass the check using gdb, by setting the value of the rax register to 0 after the call to strcmp, since strcmp returns 0 if the strings are equal.



Other resources

- LiveOverflow great YouTube videos about different security topics
- pwntools exploit development library
- Ghidra reverse engineering tool
- CyberChef encoding, encryption, etc.
- checksec check properties like PIE, or canaries
- Vagrant easy to use VM
- Online Assembler quick and easy reference for assembly
- And many more, try out what suits you best!