ARQCP Course

Arquitetura de Computadores
Licenciatura em Engenharia Informática

2023/24
Paulo Baltarejo Sousa
pbs@isep.ipp.pt

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

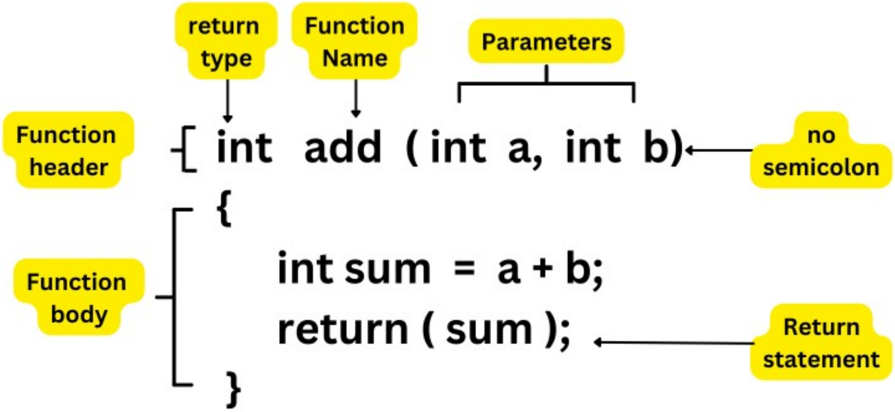
1. Functions
2. Stack
3. Calling conventions
4. Local storage
5. Stack Frames
6. Memory Errors Exploitation

Functions

What is?

- In computer programming, a **function** is a block of program instructions that performs a specific task, packaged as a **unit** (and identified by a **name**).
- Functions may be defined within **programs**, or separately in **libraries** that can be used by many programs.
  - A function may be called a **routine**, **subprogram**, **subroutine**, or **procedure**;
  - In object-oriented programming, it may be called a **method**.

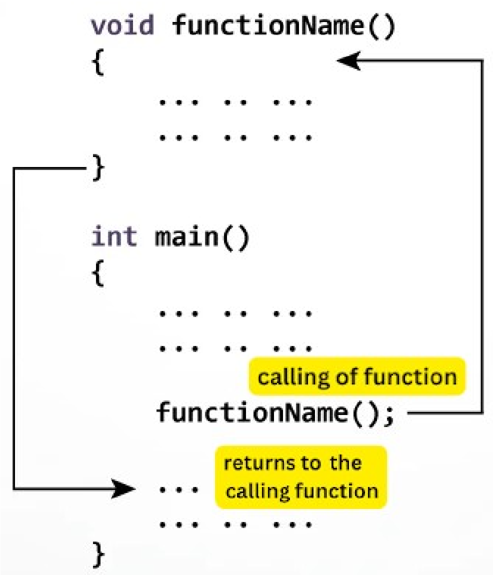
Function: Definition



```
int  add ( int  a,  int  b)
{
    int sum  =  a + b;
    return ( sum );
 }
```

Function: Invocation

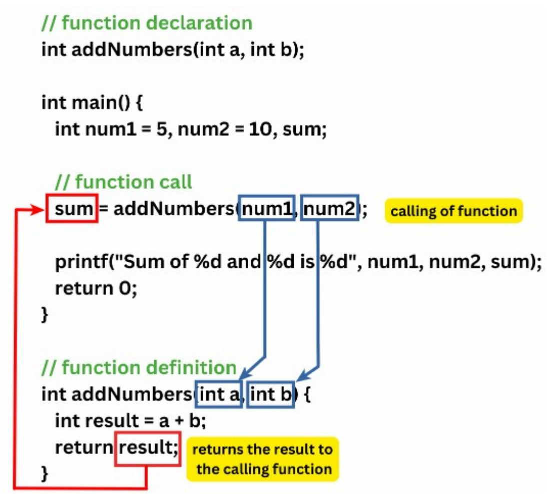
- A function is coded so that it can be **started/called several times and from several places** during one execution of the program and then **branch back (return) to the next instruction after the call**, once the function's **task is done**.
- A function's operation occurs only **when it is called**.



```
void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ...
    ... .. ...
}
```

Function: Input & Output

- Functions usually **take in** (parameters) data, **process** it, and **return** a result.

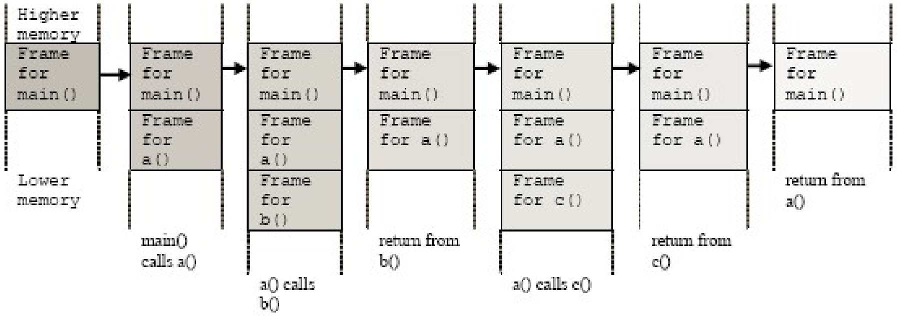


```
// function declaration
int addNumbers(int a, int b);

int main() {
  int num1 = 5, num2 = 10, sum;

  // function call
  sum = addNumbers(num1, num2);

  printf("Sum of %d and %d is %d", num1, num2, sum);
  return 0;
}

// function definition
int addNumbers(int a, int b) {
  int result = a + b;
  return result;
}
```

Function: Execution

```
int b(){
  return 0;
}

int main(){
  int x = a();
  return 0;
}
```

```
int c(){
  return 0;
}
int a(){
  int x = b();
  int y = c();
  return x+y;
}
```



Mechanisms in functions

- **Passing control**
  - Invoking (calling) a function
  - Return to the next instruction after the call
- **Passing data**
  - Function arguments
  - Return value
- **Memory management**
  - Allocate during function execution
  - Deallocate upon return
- These mechanisms are implemented with **register** and **stack** support

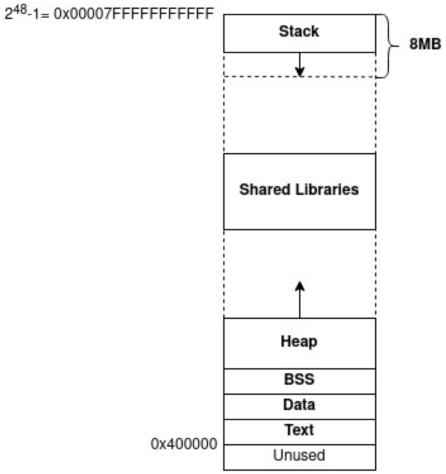
Application Binary Interface (ABI)

- An Application Binary Interface (ABI) is a **set of rules and conventions** that dictate how binary code or machine code communicates and interacts with other binary code, particularly in the context of software libraries, operating systems, or hardware.
- The ABI defines **data structures**, **calling conventions**, **register usage**, and other low-level details that ensure compatibility and interoperability between different software components.
- It acts as an **interface between high-level programming languages and the machine code**, allowing programs written in various languages to work together seamlessly.
- ABIs are crucial for binary compatibility and the proper functioning of software systems.

Stack

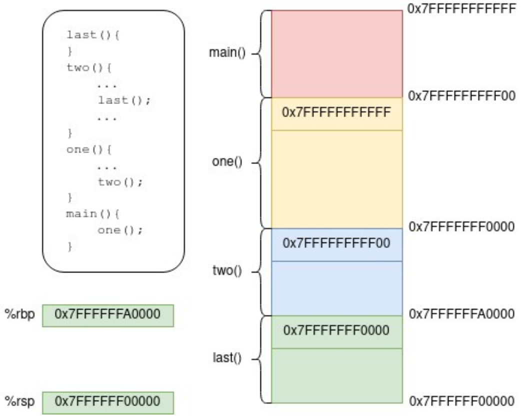
Virtual Memory Layout

- Memory viewed as **array of bytes** and different regions have different purposes



Stack (I)

- **Stack** it is used **to handle functions.**
  - As a program runs, calling one function after another, it continuously pushes data onto the stack and pops data off the stack, according **to last in, first out (LIFO) heuristic**.
- For each **function call** it creates a **stack frame**.

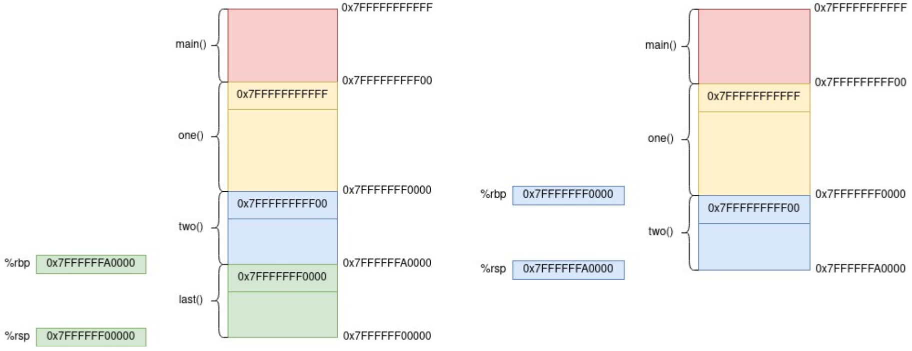


```
last(){
}
two(){
    ...
    last();
    ...
}
one(){
    ...
    two();
}
main(){
    one();
}
```

- **`%rbp` and `%rsp` registers hold bottom and top addresses of the current stack frame**, respectively.

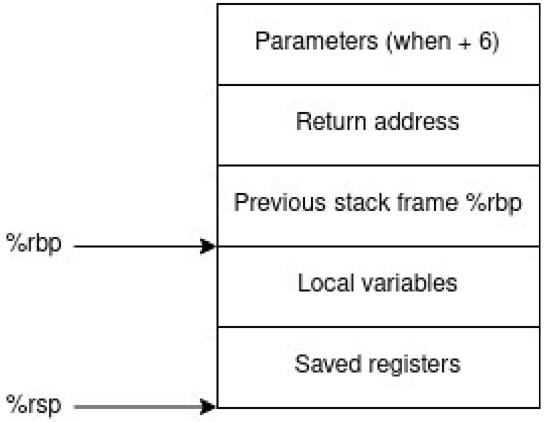
Stack (II)

- **Whenever a function finishes** (the last one, which stack frame is at the top of the stack) its execution, **the stack frame is destroyed** and `%rbp` **and** `%rsp` **are updated**.



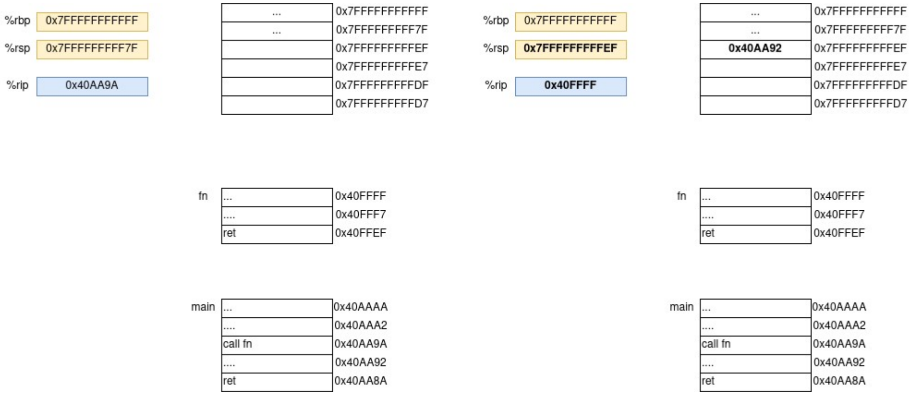
Stack Frame

- **Stack frames are data structures that store information about function calls, such as parameters, local variables, return addresses, and saved registers**.



Instruction: `call`

- `call label`
  - Push **Return address** on stack.
    - The **Return address is the address of the next instruction right after `call`**
    - `%rsp` **is decremented by 8.**
  - **Sets the `%rip` register with the address represented by `label`**



%rbp	0x7FFFFFFFFFFF
%rsp	0x7FFFFFFFFF7F
%rip	0x40AA9A

...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	....	0x40AAA2
	call fn	0x40AA9A
	....	0x40AA92
	ret	0x40AA8A

%rbp	0x7FFFFFFFFFFF
%rsp	**0x7FFFFFFFFFEF**
%rip	**0x40FFFF**

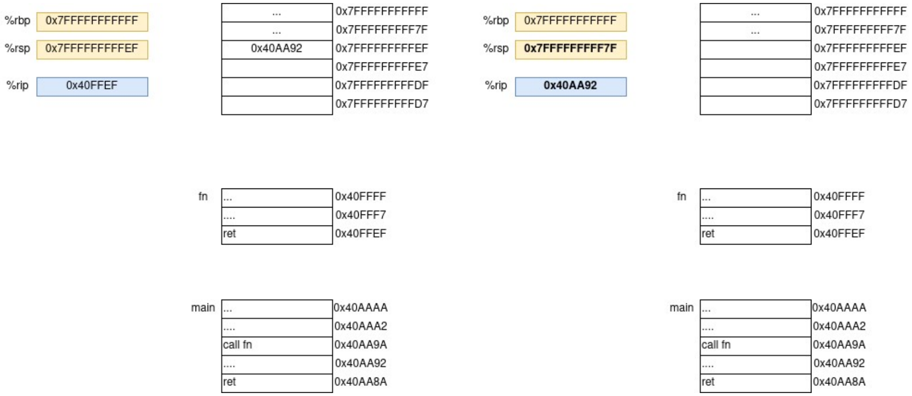
...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
0x40AA92	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	....	0x40AAA2
	call fn	0x40AA9A
	....	0x40AA92
	ret	0x40AA8A

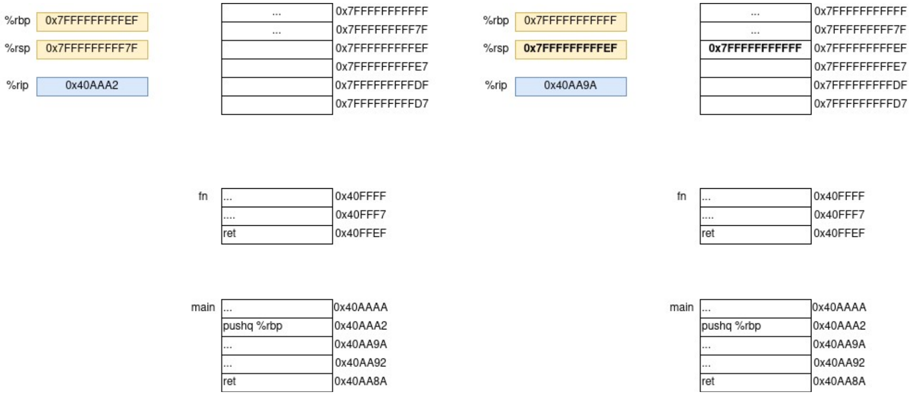
Instruction: `ret`

- `ret` instruction **pops the value off the stack and set `%rip` register with popped value**.
  - It should be the **Return address** previously pushed by `call` instruction.
  - `%rsp` **is incremented by 8**.



Instruction: `pushq`

- `pushq` *S*
  - Pushes *S* onto stack.
    - `%rsp` **is decremented by 8.**



%rbp	0x7FFFFFFFFFEF
%rsp	0x7FFFFFFFFF7F
%rip	0x40AAA2

...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	pushq %rbp	0x40AAA2
	...	0x40AA9A
	...	0x40AA92
	ret	0x40AA8A

%rbp	0x7FFFFFFFFFFF
%rsp	**0x7FFFFFFFFFEF**
%rip	0x40AA9A

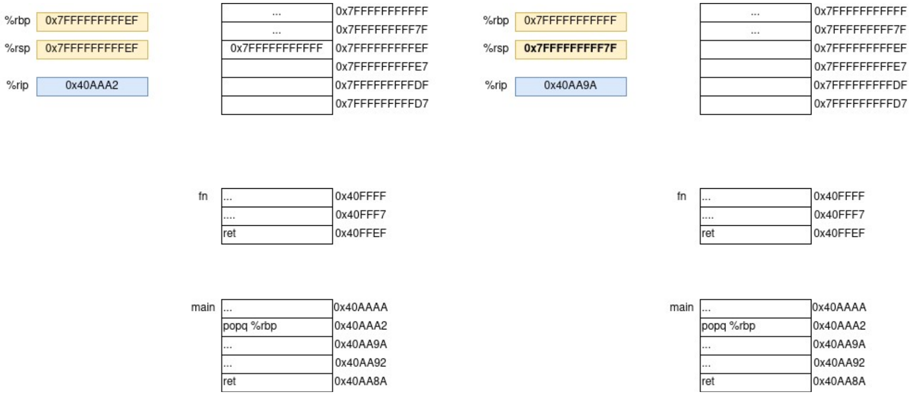
...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
0x7FFFFFFFFFFF	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	pushq %rbp	0x40AAA2
	...	0x40AA9A
	...	0x40AA92
	ret	0x40AA8A

Instruction: `popq`

- `popq` *D*
  - Pop top of stack into *D*
    - `%rsp` **is incremented by 8.**



%rbp	0x7FFFFFFFFFEF
%rsp	0x7FFFFFFFFFEF
%rip	0x40AAA2

...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
0x7FFFFFFFFFFF	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	popq %rbp	0x40AAA2
	...	0x40AA9A
	...	0x40AA92
	ret	0x40AA8A

%rbp	0x7FFFFFFFFFFF
%rsp	**0x7FFFFFFFFF7F**
%rip	0x40AA9A

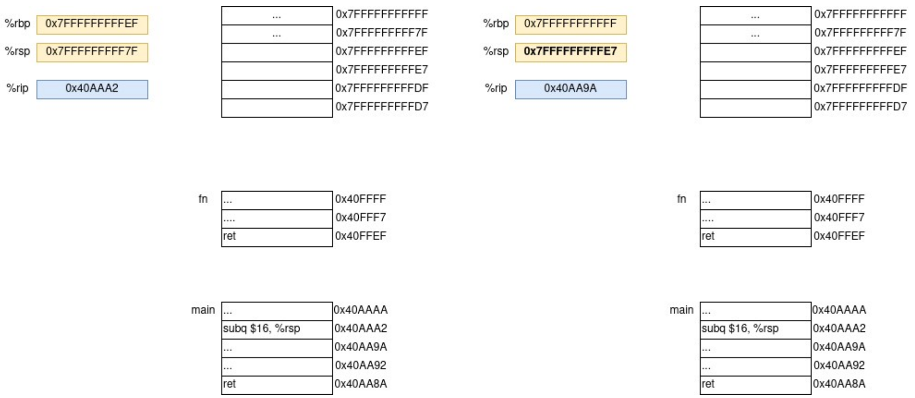
...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	popq %rbp	0x40AAA2
	...	0x40AA9A
	...	0x40AA92
	ret	0x40AA8A

Instruction: Increasing Stack

- `subq` *S* `%rsp`



%rbp	0x7FFFFFFFFFEF
%rsp	0x7FFFFFFFFF7F
%rip	0x40AAA2

...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	subq $16, %rsp	0x40AAA2
	...	0x40AA9A
	...	0x40AA92
	ret	0x40AA8A

%rbp	0x7FFFFFFFFFFF
%rsp	**0x7FFFFFFFFFE7**
%rip	0x40AA9A

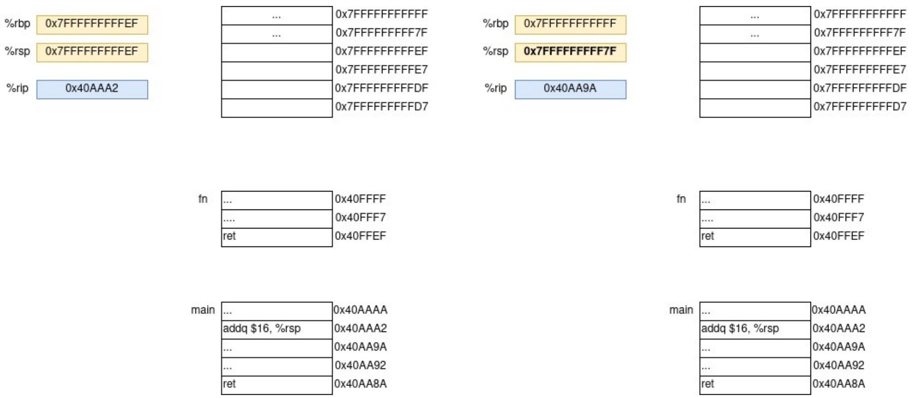
...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	subq $16, %rsp	0x40AAA2
	...	0x40AA9A
	...	0x40AA92
	ret	0x40AA8A

Instruction: Decreasing Stack

- `addq` *S* `%rsp`



%rbp	0x7FFFFFFFFFEF
%rsp	0x7FFFFFFFFFEF
%rip	0x40AAA2

...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	addq $16, %rsp	0x40AAA2
	...	0x40AA9A
	...	0x40AA92
	ret	0x40AA8A

%rbp	0x7FFFFFFFFFFF
%rsp	**0x7FFFFFFFFF7F**
%rip	0x40AA9A

...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

fn	...	0x40FFFF
	....	0x40FFF7
	ret	0x40FFEF

main	...	0x40AAAA
	addq $16, %rsp	0x40AAA2
	...	0x40AA9A
	...	0x40AA92
	ret	0x40AA8A

Calling conventions

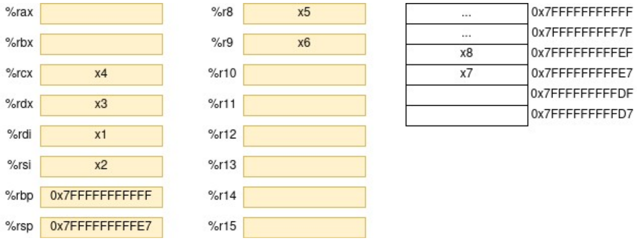
Calling Convention

- Calling conventions describe the **conventions or norms that functions use when it calls another function and when a function returns to its caller function**.
- The calling convention is based heavily on the use of the **stack** and **registers**.
- A **caller** is a function that calls another function;
- A **callee** is a function that was called.

Passing data to callee

- To **pass parameters to function**, we put **up to six of them into registers** (in order: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`).
  - If there are **more than six parameters** to the function, then **push the rest onto the stack in reverse order** (i.e. last parameter first)

```
void f2( long x1,long x2,long x3,long x4,
         long x5,long x6,long x7,long x8);
```



%rax		%r8	x5
%rbx		%r9	x6
%rcx	x4	%r10	
%rdx	x3	%r11	
%rdi	x1	%r12	
%rsi	x2	%r13	
%rbp	0x7FFFFFFFFFFF	%r14	
%rsp	0x7FFFFFFFFFE7	%r15	

...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
x8	0x7FFFFFFFFFEF
x7	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7

```
f1:
...
movq x1(%rip), %rdi
movq x2(%rip), %rsi
movq x3(%rip), %rdx
movq x4(%rip), %rcx
movq x5(%rip), %r8
movq x6(%rip), %r9
movq x8(%rip), %r10
pushq %r10
movq x7(%rip), %r10
pushq %r10
call f2
...
ret
```

Returning data to caller

- To return a value, the callee **stores it into `%rax`**
  - This is the reason for **a function returning only one value**

```
int x = 10;
long f2(){
 return 1;
}
void f1(){
  x += f2();
  ...
}
```

```
x: .int 10

f2:
  ...
  movq $1, %rax
  ...
  ret
f1:
  ...
  movq x(%rip), %r10
  call f2
  addq %rax, %r10
  ret
```

Calling function: Registers usage (I)

```
f1:
  ...
  movq $50, %rbx
  movq $200, %rdx
  call f2
  addq %rbx, %rdx
  ...
  ret
```

```
f2:
  ...
  movq $10, %rdx
  ...
  addq $20, %rdx
  ...
  ret
```

- Contents of register `%rdx` overwritten by `f2`
- This is an issue, that requires some **coordination mechanism**

Calling function: Registers usage (II)

```
f1:
  ...
  movq $50, %rbx
  movq $200, %rdx
  pushq %rdx
  call f2
  popq %rdx
  addq %rbx, %rdx
  ...
  ret
```

```
f2:
  ...
  movq $10, %rdx
  ...
  addq $20, %rdx
  ...
  ret
```

- **Caller saves `%rdx` before call** and **restores it after return**
  - Caller-saved registers **can be modified by any function**
  - Since the **called function is free to alter these registers**, it is incumbent upon the **caller to first save the data before it makes the call**

Register Saving Convention (I)

- **Caller save**
  - Caller saves (on Stack) temporary values before the call
  - Caller restores (from Stack) them after returning from the callee
- **Callee save**
  - Callee saves (on Stack) temporary values before using
  - Callee restores (from Stack) them before returning to caller
- Which registers are **caller-save** or **callee-save**?

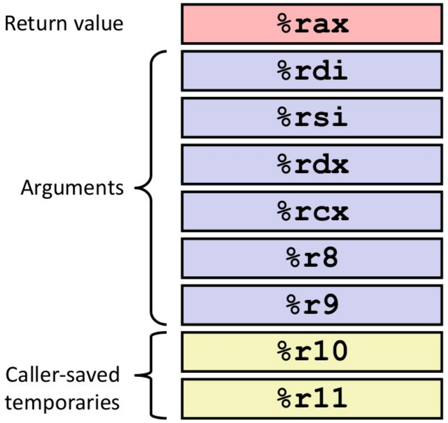
Register Saving Convention (II)

%rax	Return value - Caller saved	%r8	Argument #5 - Caller saved
%rbx	Callee saved	%r9	Argument #6 - Caller saved
%rcx	Argument #4 - Caller saved	%r10	Caller saved
%rdx	Argument #3 - Caller saved	%r11	Caller Saved
%rsi	Argument #2 - Caller saved	%r12	Callee saved
%rdi	Argument #1 - Caller saved	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

Caller-saved registers

- These registers can be modified by **callee function**.

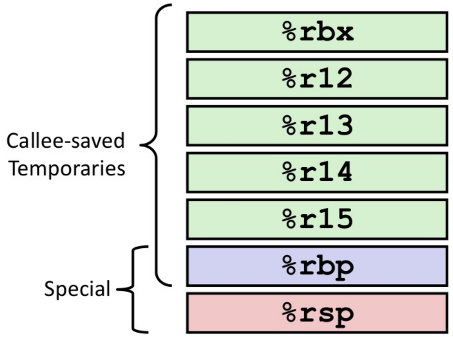
```
f1:
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  call f2
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  ...
  ret
```



Callee-saved registers

- Callee **must save and restore** (could be being used by caller)

```
f2:
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  ...
  popq %r15
  popq %r14
  popq %r13
  popq %r12
  ...
  ret
```



- %rbp
  - May be used as **stack frame pointer**
- %rsp
  - Special form of **callee save**
  - **Restored to original value** upon exit from function.

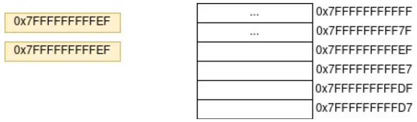
Local storage

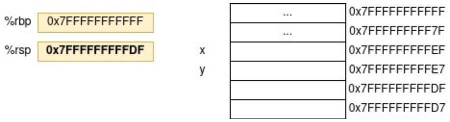
Function variables (I)

- **Local variables are allocated onto Stack**
- It simply **subtracting the number of bytes required by each variable from the `%rsp`**.
- This **does not store any data in the variables**, it simply sets aside memory that we can use.
- There are **no labels in this area of memory**

```
void fn(){
  long x;
  long y;
  ...
}
```

```
fn:
  ...
  subq $16, %rsp
  ...
  ret
```



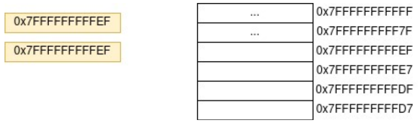


Function variables (II)

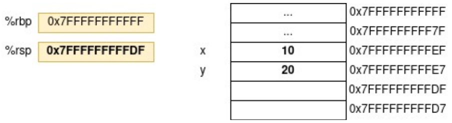
- Accessing to **local variable could be done using `%rsp` as anchor**.
- Recall, there are no labels in this area of memory

```
void fn(){
  long x;
  long y;
  ...
  x = 10;
  y = 20;
}
```

```
fn:
...
subq $16, %rsp
...
movq $20, (%rsp)
movq $10, 8(%rsp)

ret
```



...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7



	...	0x7FFFFFFFFFFF
	...	0x7FFFFFFFFF7F
x	10	0x7FFFFFFFFFEF
y	20	0x7FFFFFFFFFE7
		0x7FFFFFFFFFDF
		0x7FFFFFFFFFD7

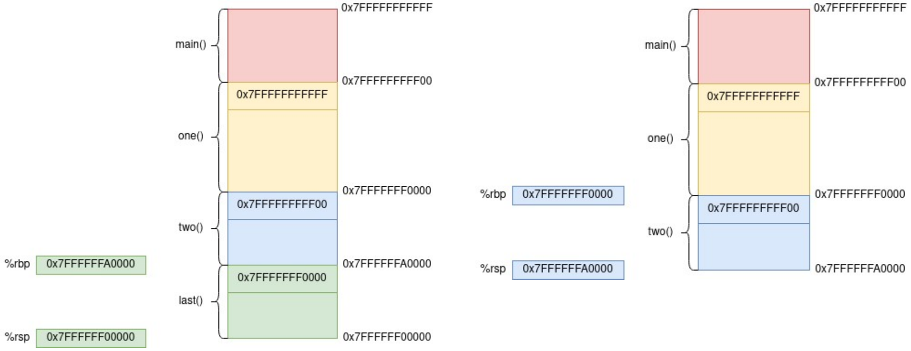
Stack Frames

Stack frames

- Stack frames **only exist at run-time**.
- They are used **to handle the function calls**.
- Contents:
  - Local variables
  - Return information
  - Temporary space
- Management
  - Space allocated when enter function
    - “Set-up” code (prologue)
  - Deallocated when return
    - “Finish” code (epilogue)

Stack frames: Prologue and Epilogue (I)

```
function:
  #Prologue
 pushq %rbp      # Save old %rbp
 movq %rsp,%rbp # Set %rbp as frame pointer
  ...
  #Epilogue
 movq %rbp,%rsp # Set %rsp to beginning of frame
 popq %rbp      # Restore saved %rbp
  ret
```



Stack frames: Prologue and Epilogue (II)

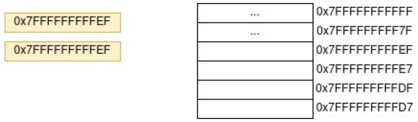
- The **function prologue is the process of creating a stack frame to hold callee function information**.
  - It is done by the callee function—the code to create the frame is located at the start of the callee function.
  - There are three steps to the function prologue:
    1. The current value of `%rbp` is pushed onto the stack. This will allow the calling function's stack frame to be rebuilt after the callee function finish;
    2. The current value of `%rsp` is moved into `%rbp`;
    3. Space is allocated for any local variables. This is done by subtracting their collective size (in hexadecimal form) from `%rsp`.
- Function epilogue **reverses the actions of the function prologue and returns control to the calling function by resetting its stack frame**.
  - The function epilogue also has three steps:
    1. `%rbp` is moved into `%rsp`;
    2. `%rbp` is popped from the stack;
    3. The return address is read from the top of the stack (where `%rbp` is pointing) and the instruction pointer jumps to that address.

Managing Local data

- Accessing to **stack frame data using `%rsp` as anchor could be dangerous**, because **there are instructions that change `%rsp`**.
- Using **Prologue** and **Epilogue** approach to manage stack frames, **provides a more stable option is to be used as acnchor: `%rbp`**

```
void fn(){
  long x;
  long y;
  ...
  x = 10;
  y = 20;
}
```

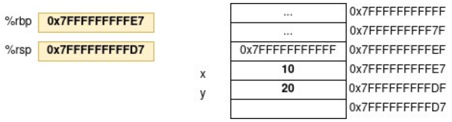
```
fn:
  pushq %rbp
  movq %rsp, %rbp
  ...
  subq $16, %rsp
  ...
  movq $20, -16(%rbp)
  movq $10, -8(%rbp)

  movq %rbp, %rsp
  popq %rbp
ret
```



0x7FFFFFFFFFEF

0x7FFFFFFFFFEF

...	0x7FFFFFFFFFFF
...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFEF
	0x7FFFFFFFFFE7
	0x7FFFFFFFFFDF
	0x7FFFFFFFFFD7



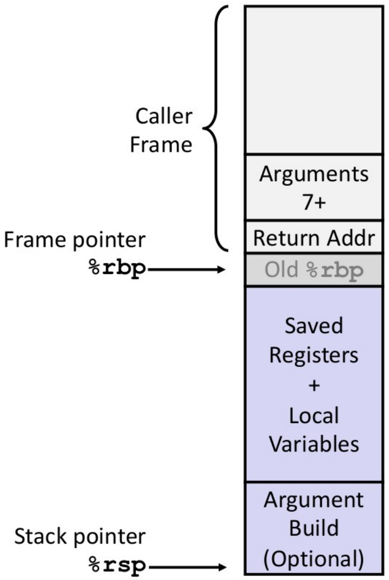
%rbp 0x7FFFFFFFFFE7

%rsp 0x7FFFFFFFFFD7

	...	0x7FFFFFFFFFFF
	...	0x7FFFFFFFFF7F
	0x7FFFFFFFFFFF	0x7FFFFFFFFFEF
x	10	0x7FFFFFFFFFE7
y	20	0x7FFFFFFFFFDF
		0x7FFFFFFFFFD7

Stack frame structure

- **Current stack frame** (Top to Bottom)
  - Argument build: 7+ parameters for function about to call
  - Local variables, if can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller stack frame
  - Return address (pushed by `call`)
  - 7+ arguments for this call



Memory Errors Exploitation

Buffer overflow (I)

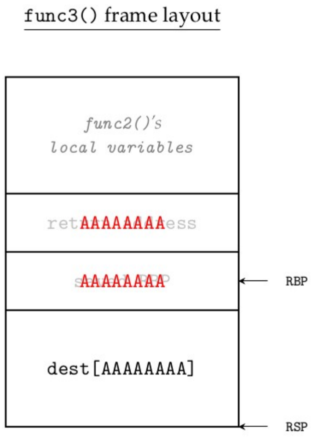
- A **buffer overflow is the result of stuffing more data into a buffer than it can handle**
- `strcpy`, is the most **infamous for being the cause of buffer overflows**.
- `strncpy` operates in the same way as `strcpy`, except that it copies a specified amount of bytes, `n`, from `src` to `dest`.
- Although the `strcpy` copy could be stopped before if the source strings ends, under an attack the input length is controlled by attackers and will never happen.

Buffer overflow (II)

```
char dest[8];
```

```
strcpy(dest, "AAAAAAAAAAAAAAAAAAAA");
```

```
strncpy(dest,"AAAAAAAAAAAAAAAAAAAAAAAA", 24);
```



func3() frame layout

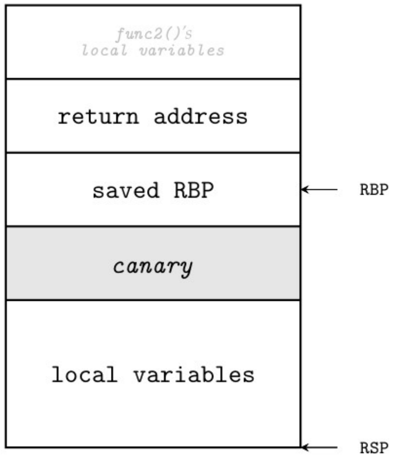
Attack Approach: Code injection

- One of the first goals for attackers **when probing for buffer overflow vulnerabilities is gaining the ability to overwrite the stack frame return address**.
- When it is possible to **overwrite the return address of a stack frame**, and an attacker does so, the **CPU will jump to whatever address is stored in the return address when the function attempts to return to its caller**
- An example of `shellcode` that can be inserted into a vulnerable process to reboot a Linux x86-64 machine

```
char shellcode_reboot[] =
"\xBA\xDC\xFE\x21\x43"
"\xBE\x69\x19\x12\x28"
"\xBF\xAD\xDE\xE1\xFE"
"\xB0\xA9"
"\x0F\x05";
```

Memory Protection Techniques: Stack Smashing Protector (SSP)

- To accomplish return address overwritten mitigation, **a canary value** was inserted next to the return address of the current stack frame to prevent an attacker from overwriting the return address.
- The **canary value is checked before the instruction pointer loads the return address of the stack frame**.
  - If the canary value is altered, the processor knows that an attack has been attempted and execution is aborted.



Memory Protection Techniques: Address Space Layout Randomisation (ASLR)

- ASLR is a protection technique that attempts **to render exploits that depend on predetermined memory addresses useless**
- It is a protection technique that which the memory address layout to prevent attacks that relies on knowing the location of an application's memory map