CSIT 5740 Introduction to Software Security

Note set 4A

Dr. Alex LAM

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

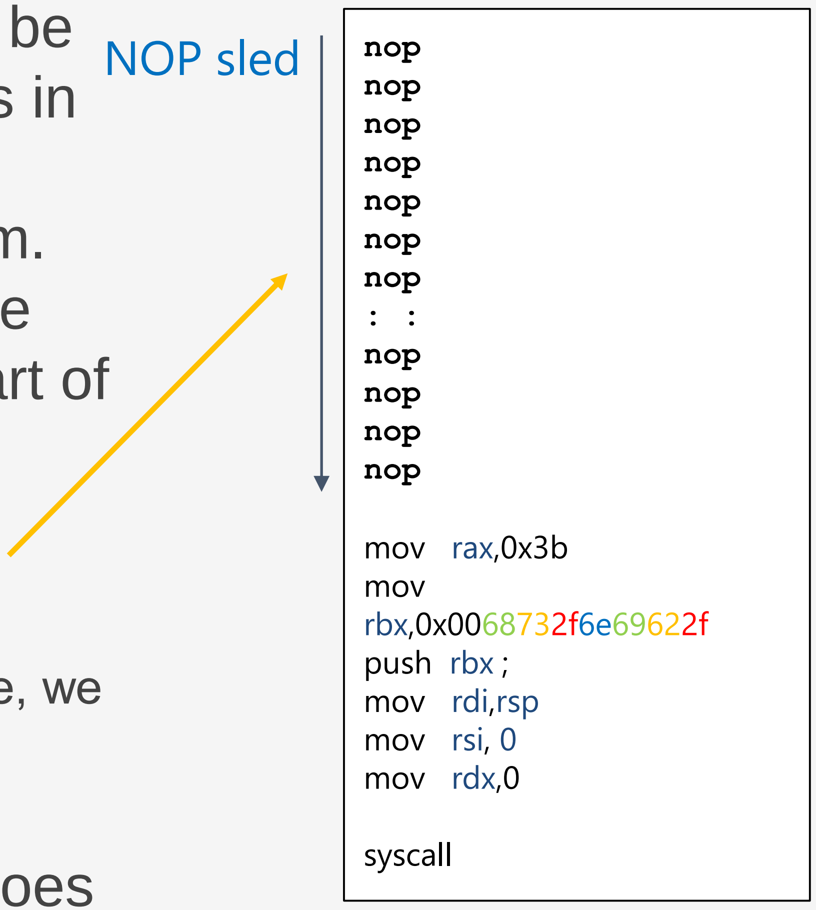
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

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

Writing more robust Shellcode with NOPs

NOP Sleds

- The start address of the array loaded with shellcode could be hard to get (the compiler may put the local arrays/variables in slightly different addresses). Addresses in GDB could be different than the real addresses when you run the program.
- Idea: Instead of jumping to the exact starting address of the shellcode, we would like to jump to a place close to the start of the shellcode to start it
  - We put many NOPs (no operation instructions) to the front of the shellcode
  - We don’t need to know the exact starting address of the shellcode, we just need to jump to any of the NOPs
- **NOP**: Short for no-operation or no-op, an instruction that does nothing (except advance the RIP/EIP)
  - It is a real instruction in x86 and x64



```
nop
nop
nop
nop
nop
nop
nop
: :
nop
nop
nop
nop

mov   rax,0x3b
mov
rbx,0x0068732f6e69622f
push  rbx ;
mov   rdi,rsp
mov   rsi, 0
mov   rdx,0

syscall
```

Memory Corruption Exploitation Examples

Non-terminated String Overflow

- Some functions, such as `strncpy`, limit the amount of data copied in the destination buffer but do not include a terminating `NULL` byte when the limit is reached
    - From `man strncpy`
        - “The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied. Warning: If there is no null byte among the first `n` bytes of **src**, the string placed in **dest** will not be null-terminated.”

- If adjacent buffers are not null-terminated it is possible to cause the overflow when length of data is determined by functions such as `strlen()`

The “safe” function strncpy(): is it really safe?

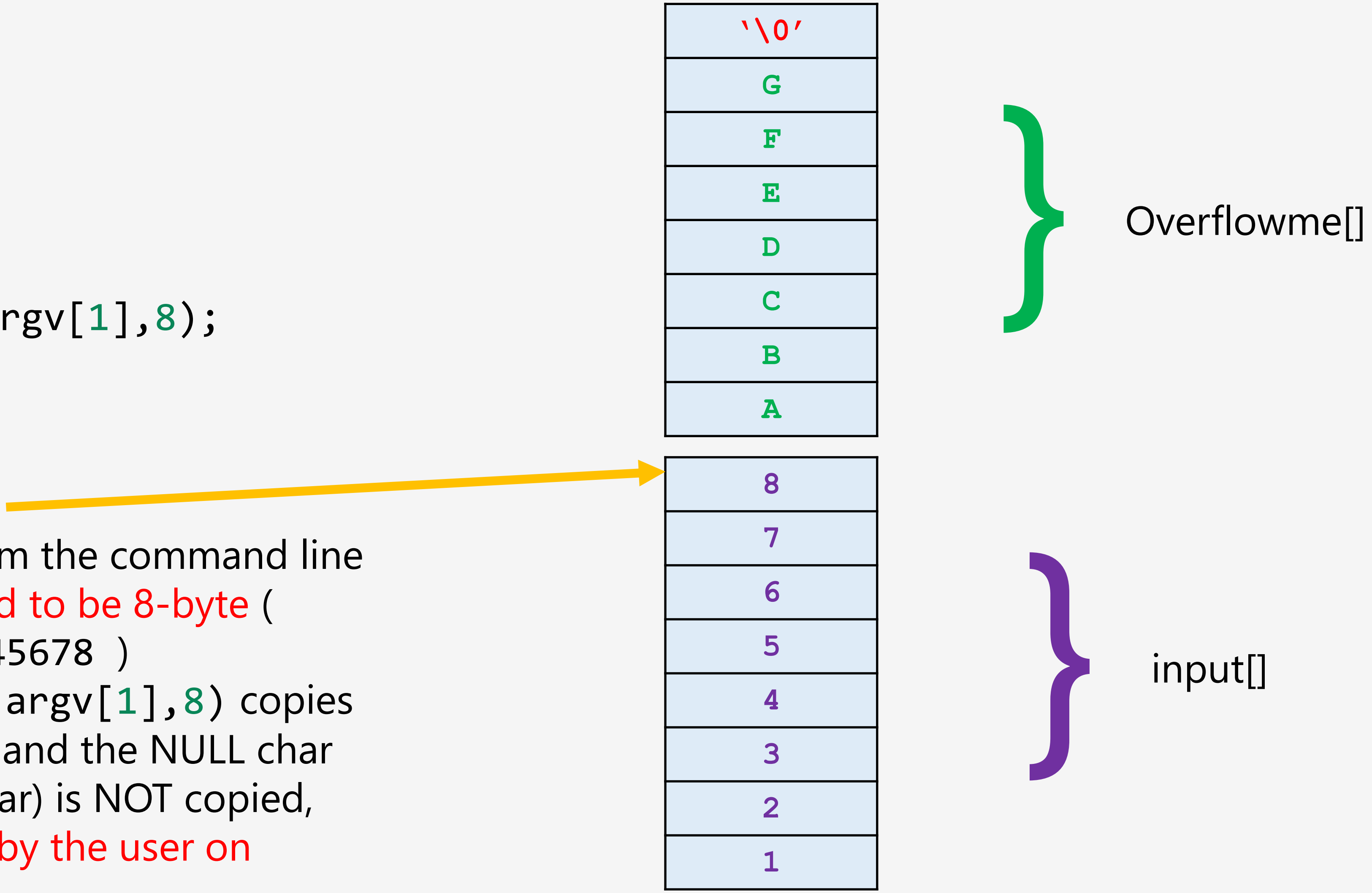
```
/* compiled using gcc strncpy.c -o strncpy */
#include <stdio.h>
#include <string.h>

void main (int argc, char**argv){

   char overflowme[8]="ABCDEFG";
   char input[8];

   strncpy(input,argv[1],8);

   strncpy(overflowme,input,strlen(input));

   printf("The size of overflowme[] is 8, the actual size of overflowme[] string is %d!“\
s,strlen(overflowme));

}
```

There is an overflow!

```
 ./strncpy 12345678
The size of overflowme[] is 8, the actual size of overflowme[] string is 15!
```

The “safe” function strncpy(): is it really safe?

```
strncpy(input,argv[1],8);
```

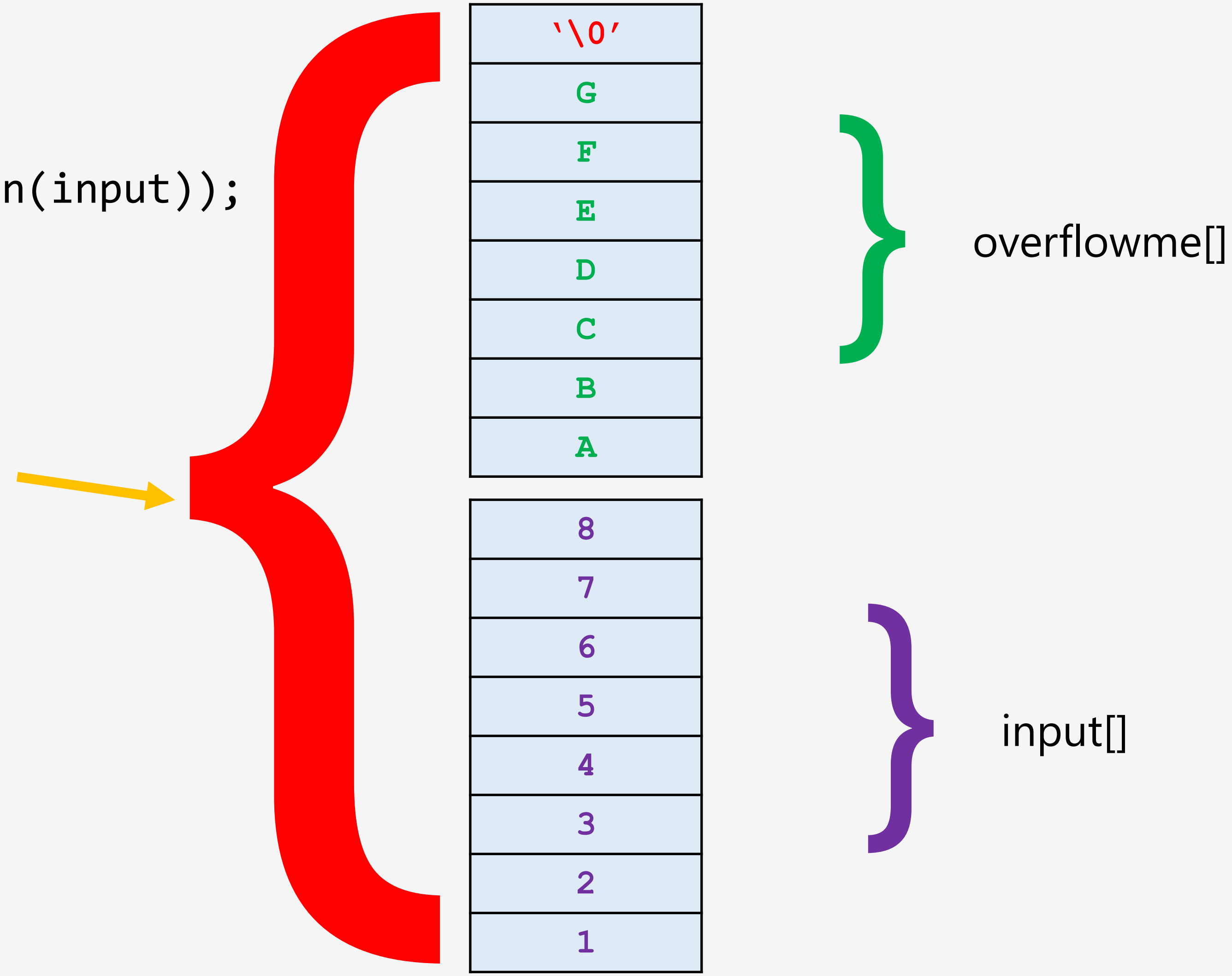


- the input size from the command line is carefully crafted to be 8-byte (`./strncpy 12345678`)
- `strncpy(input,argv[1],8)` copies 8 byte to input[], and the NULL char (end-of-string char) is NOT copied, and that is done by the user on purpose.

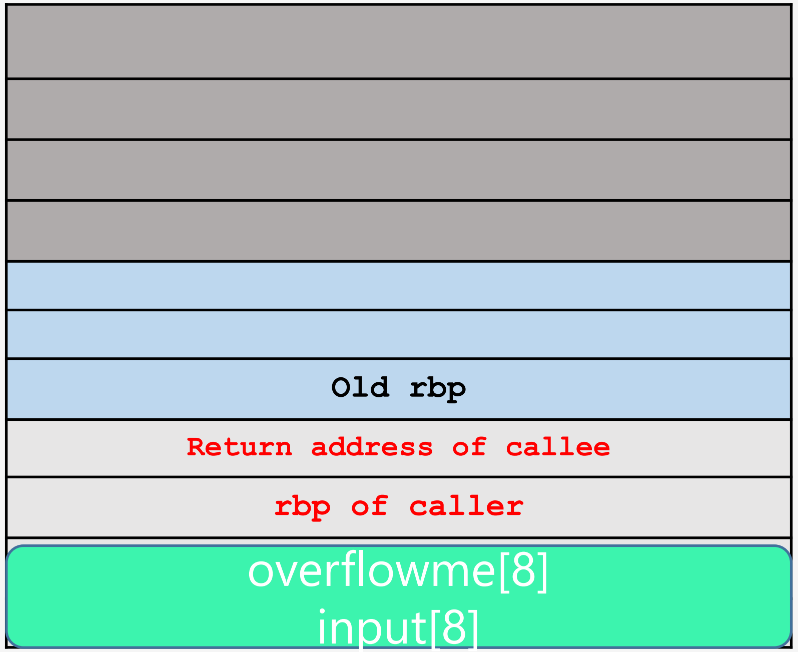
The “safe” function strncpy(): is it really safe?

```
strncpy(overflowme,input,strlen(input));
```

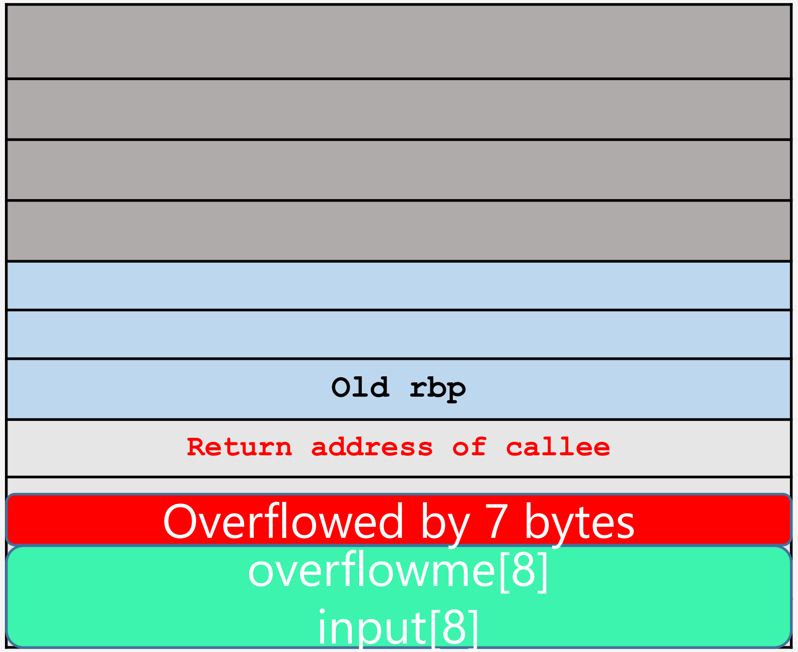
- `strlen()` check the length of the string by counting the number of characters before encountering the first NULL. So it sees input[] to be this big array consists of both input[] and overflowme[] (which is wrong),and it returns 15
- strncpy overflows by: 15 (data size copied)- 8 (actual array size) = 7 bytes!



What happens in the stack level



What happens in the stack level



Old rbp

Return address of callee

Overflowed by 7 bytes

overflowme[8]

input[8]

Non-terminated String Overflow

- Lessons Learned
  - Make sure strings are NULL-terminated
  - Reserve enough space for the terminator

Index Overflow

- This type of overflow exploits the lack of boundary checks in the value used to index an array

- They are particularly easy to exploit because they allow for the direct assignment of memory values

- a[**x**] = **y**

  if an attacker fully control **x** and **y**, the attacker decides both **what** to write and **where** to write it, x can be +ve or –ve !! (What's the implication?)

Index Overflow

```c
/* compiled using gcc indexOverflow.c -o indexOverflow*/
#include <stdio.h>
int main(int argc, char *argv[])
{
  long array[8];
  int index;
  int value;
  /*convert arg1 to long, assume base10*/
  index = (int) strtol(argv[1], NULL, 10);
  /*convert arg2 to unsigned long, assume base16*/
  value = (int) strtoul(argv[2], NULL, 16);
  array[index] = value;
  return 0;
}
```

This careless code allows a user to assign an arbitrary value to an arbitrary memory location!

```
./indexOverflow 11 AAAAAAAA
Segmentation fault (core dumped)
```

array[**x**] = **y**

x could be **+ve and –ve!**

Index Overflow

- Lesson Learned: always check that array indexes controllable by user input are within the array bounds

Loop Index Overflow (off-by-one)

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

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;
}
```

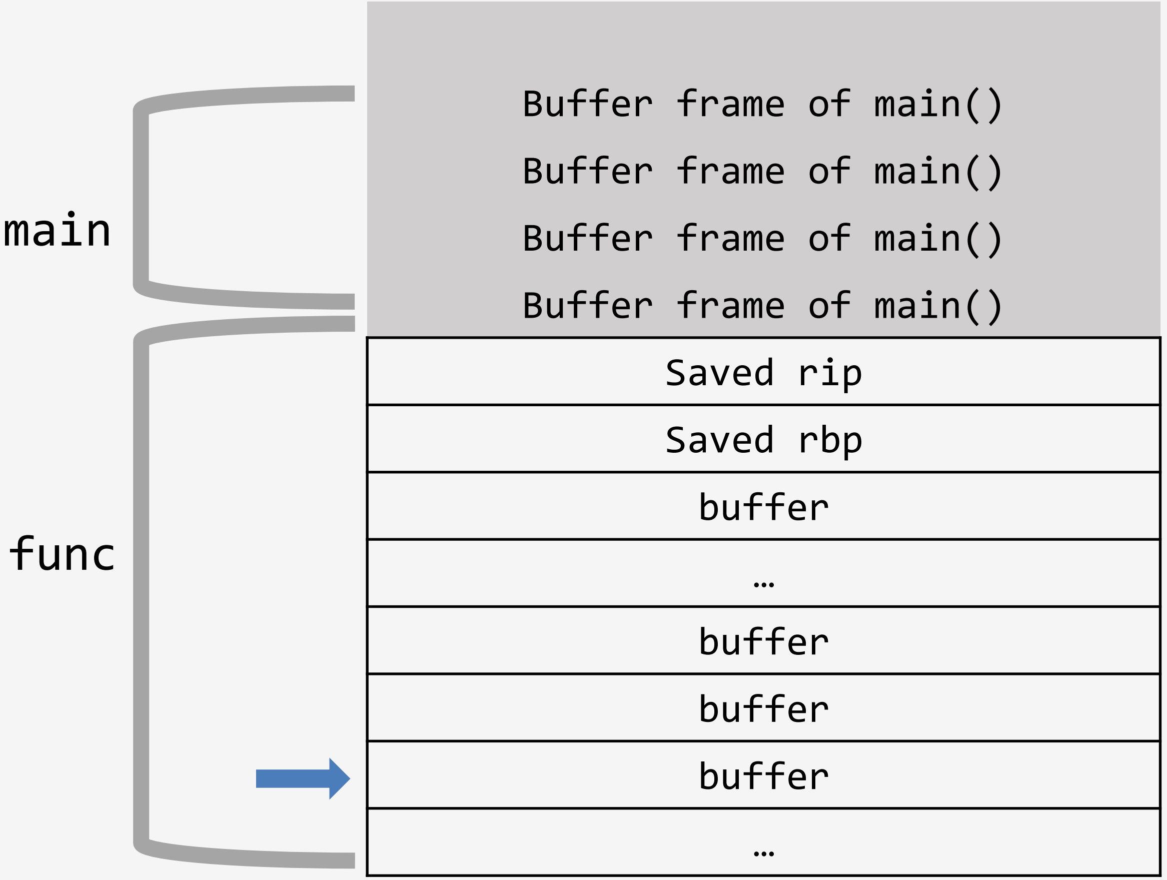
This loop repeats for 65 times (i=0 to 64), writing 65 bytes to the array with an allocated size of 64 bytes.

Overflows the buffer 1 byte!

Loop Index Overflow (off-by-one), the concept

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

```
PROLOGUE
push  rbp
mov   rbp, rsp

EPILOGUE
leave # mov rsp, rbp; pop rbp
ret
```



main

func

Buffer frame of main()

Buffer frame of main()

Buffer frame of main()

Buffer frame of main()

Saved rip

Saved rbp

buffer

…

buffer

buffer

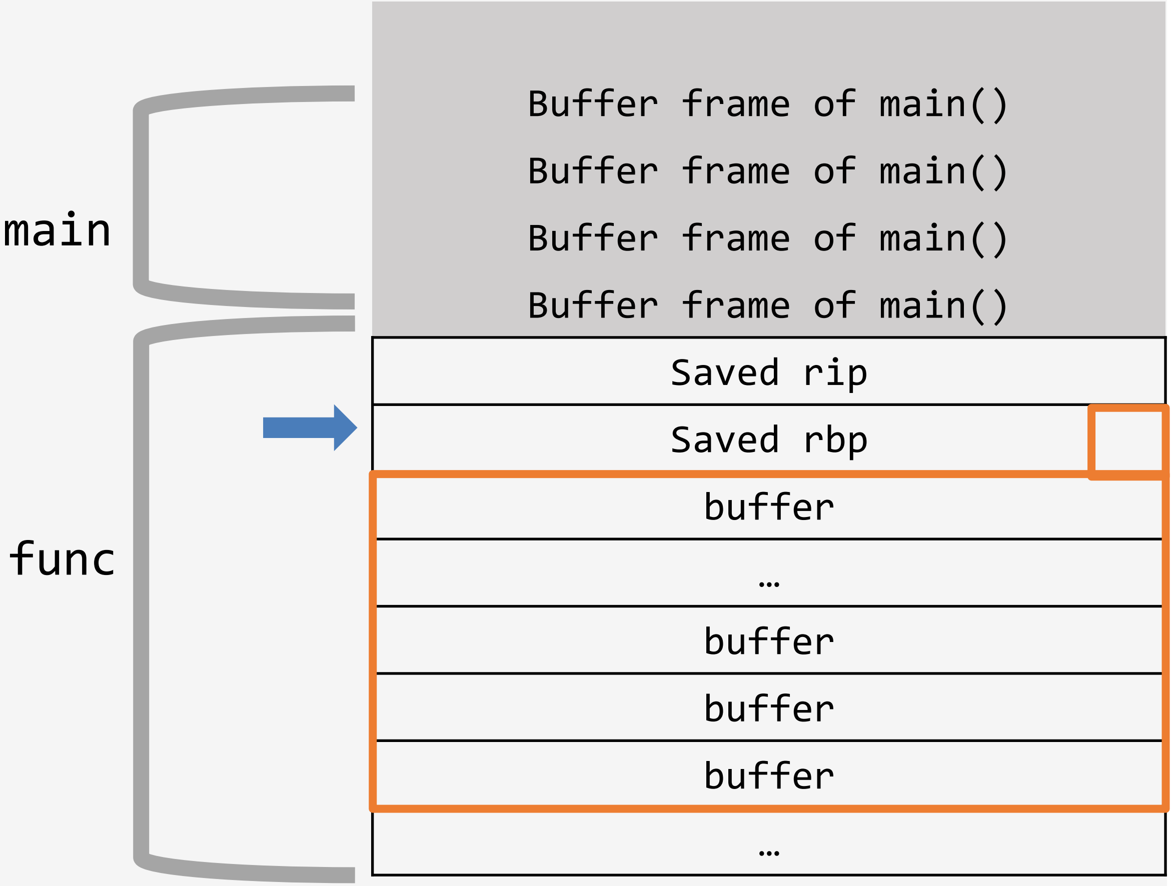
buffer

…

Loop Index Overflow (off-by-one), the concept

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

```
PROLOGUE
push  rbp
mov   rbp, rsp

EPILOGUE
leave # mov rsp, rbp; pop rbp
ret
```



Loop Overflow

- Loop overflows happen when the attacker can control the loop iterations and checks are missing

- A special case: off-by-one loop overflows
  - “Humans” are prone to “1 element” mistakes
    (how many numbers between 1 and 10?)
  - These attacks are similar to array overflows, with the difference that only one element above the array capacity is overwritten
  - Can be used to modify the least significant byte of pointers

- Lesson learned
  - User-supplied input should not lead to arbitrary loop iterations
  - Off-by-one vulnerabilities are common

Controlling 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)

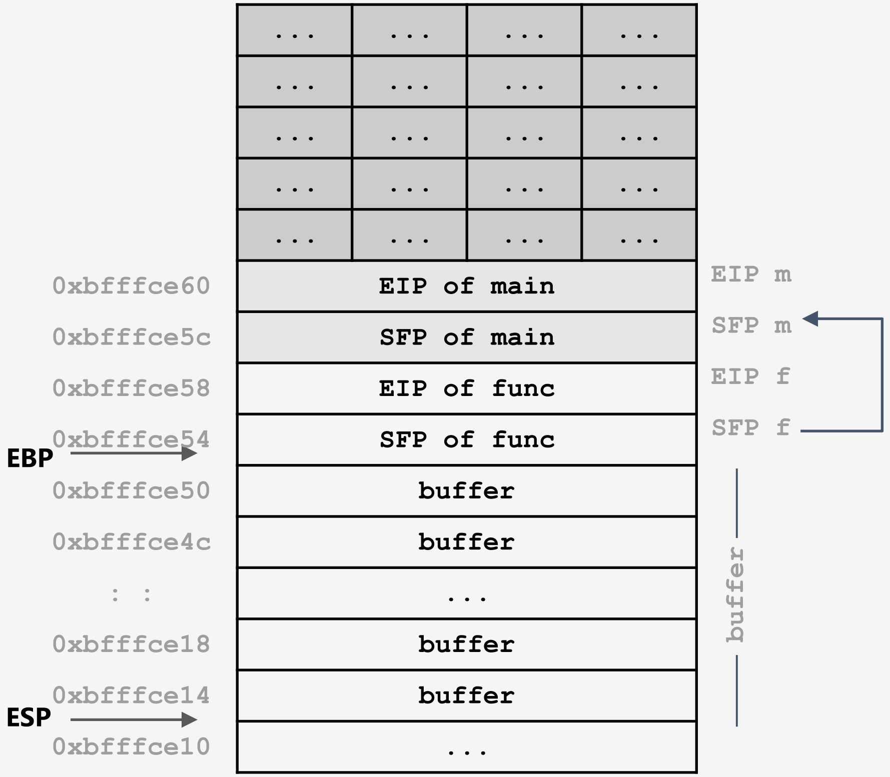
- This is an example of stack pivoting
  - We modify rsp, to change what the memory region the program uses as stack

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;
}
```

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

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



		...	...	...	...	
		...	...	...	...	
		...	...	...	...	
		...	...	...	...	
		...	...	...	...	
	0xbfffce60	EIP of main				EIP m
	0xbfffce5c	SFP of main				SFP m
	0xbfffce58	EIP of func				EIP f
EBP →	0xbfffce54	SFP of func				SFP f
	0xbfffce50	buffer				
	0xbfffce4c	buffer				
	: :	...				
	0xbfffce18	buffer				
ESP →	0xbfffce14	buffer				
	0xbfffce10	...				

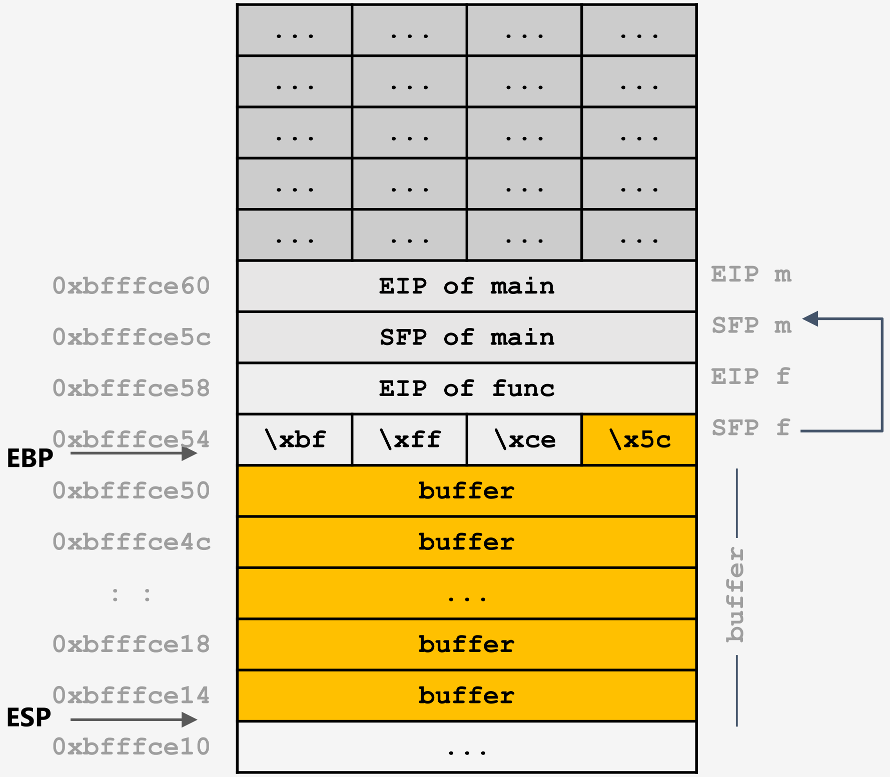
A 32-bit example

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;
}
```

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

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



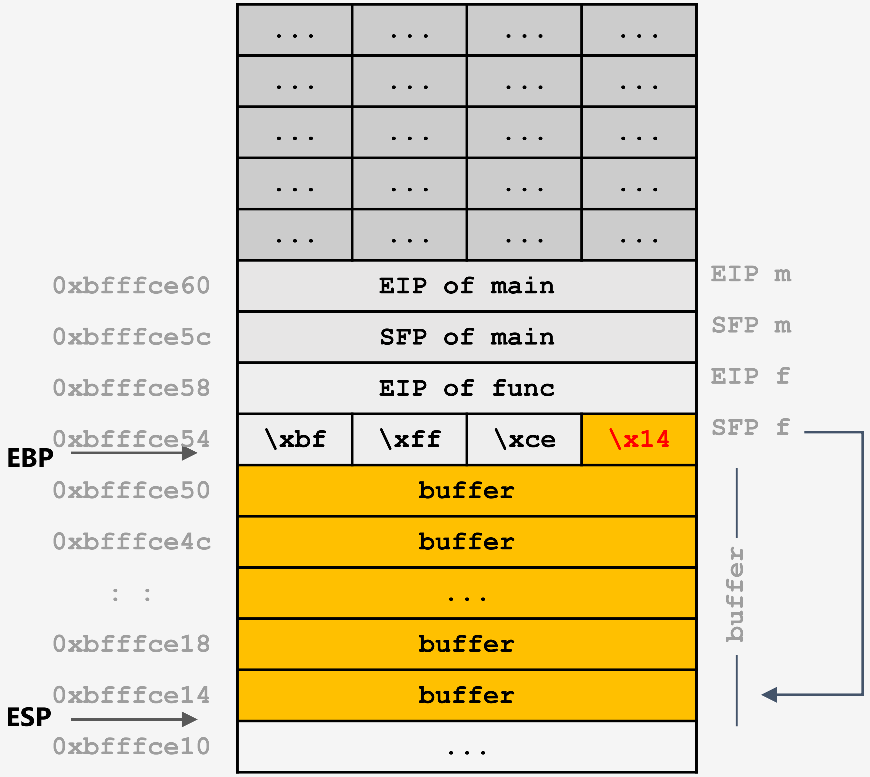
A 32-bit example

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;
}
```

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

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



	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffce60	EIP of main				EIP m
0xbfffce5c	SFP of main				SFP m
0xbfffce58	EIP of func				EIP f
EBP → 0xbfffce54	\xbf	\xff	\xce	\x14	SFP f
0xbfffce50	buffer				
0xbfffce4c	buffer				
: :	...				
0xbfffce18	buffer				
ESP → 0xbfffce14	buffer				
0xbfffce10	...				

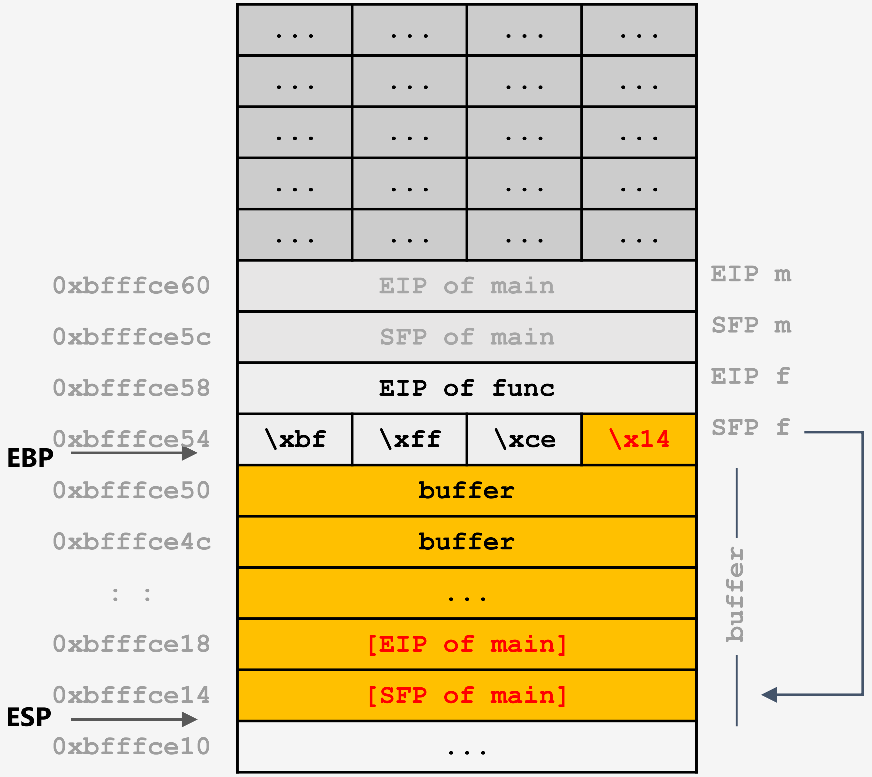
A 32-bit example

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;
}
```

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

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



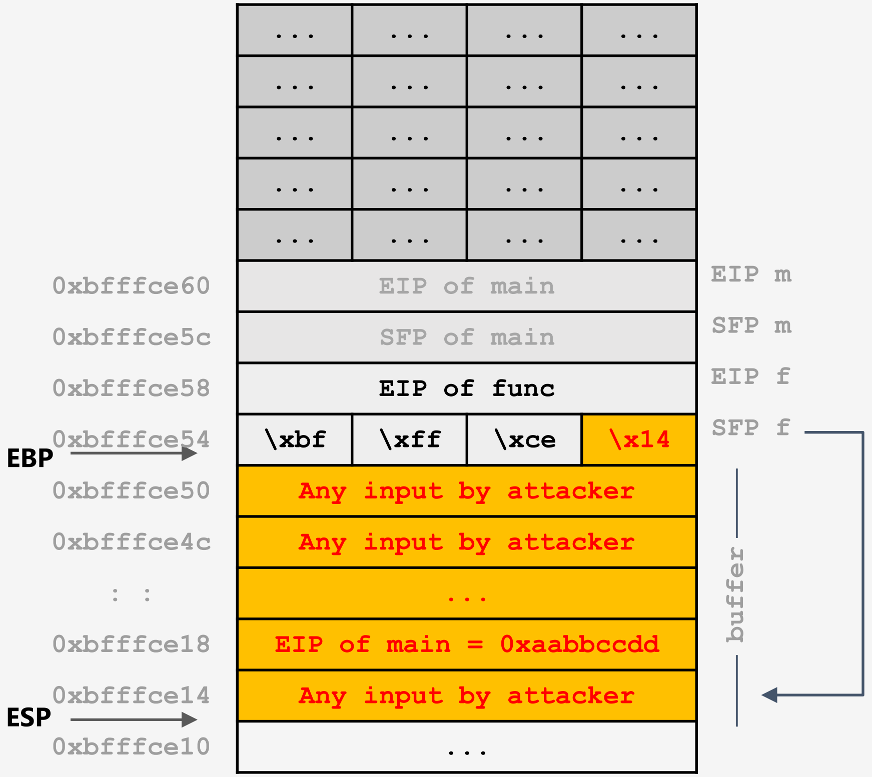
		...	...	...	...	
		...	...	...	...	
		...	...	...	...	
		...	...	...	...	
		...	...	...	...	
	0xbfffce60	EIP of main				EIP m
	0xbfffce5c	SFP of main				SFP m
	0xbfffce58	EIP of func				EIP f
EBP	0xbfffce54	\xbf	\xff	\xce	\x14	SFP f
	0xbfffce50	buffer				buffer
	0xbfffce4c	buffer				
	: :	...				
	0xbfffce18	[EIP of main]				
ESP	0xbfffce14	[SFP of main]				
	0xbfffce10	...				

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;
}
```

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

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



... ...
... ...
... ...
... ...
... ...
0xbfffce60 EIP of main — EIP m
0xbfffce5c SFP of main — SFP m
0xbfffce58 EIP of func — EIP f
EBP → 0xbfffce54 \xbf \xff \xce \x14 — SFP f
0xbfffce50 Any input by attacker
0xbfffce4c Any input by attacker
: : ...
0xbfffce18 EIP of main = 0xaabbccdd
ESP → 0xbfffce14 Any input by attacker
0xbfffce10 ...

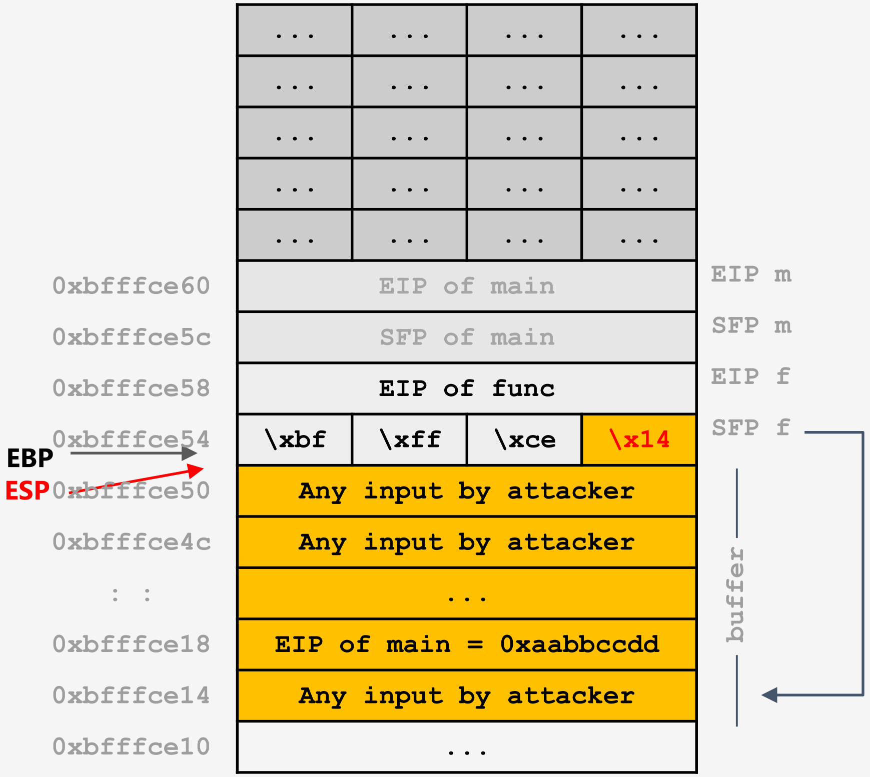
A 32-bit example

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;
}
```

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

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



	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffce60	EIP of main				EIP m
0xbfffce5c	SFP of main				SFP m
0xbfffce58	EIP of func				EIP f
EBP → 0xbfffce54	\xbf	\xff	\xce	\x14	SFP f
ESP → 0xbfffce50	Any input by attacker				buffer
0xbfffce4c	Any input by attacker				
: :	...				
0xbfffce18	EIP of main = 0xaabbccdd				
0xbfffce14	Any input by attacker				
0xbfffce10	...				

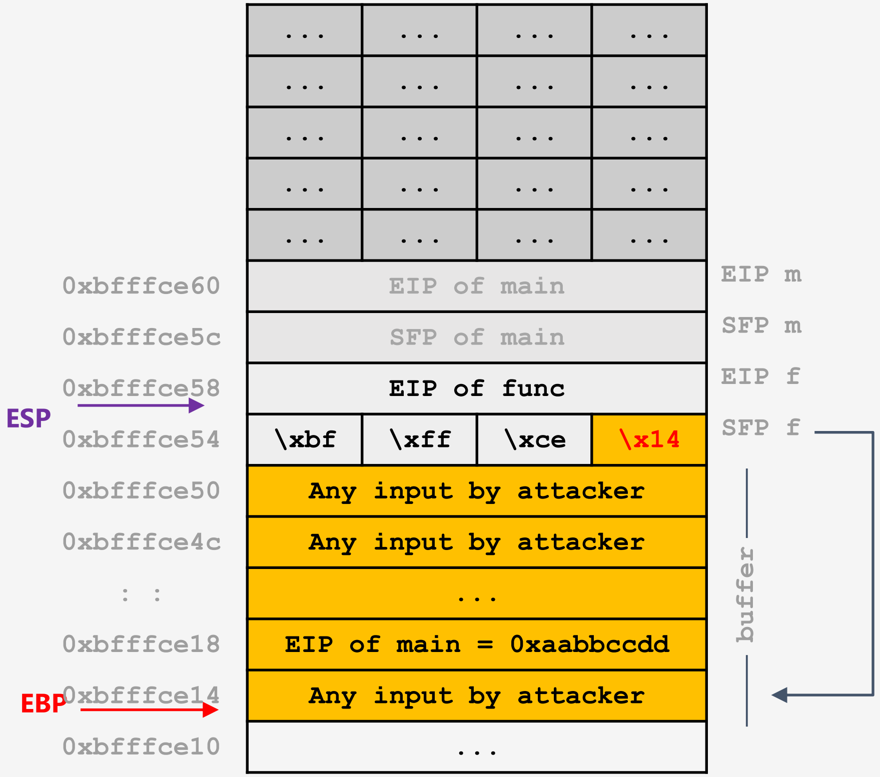
A 32-bit example

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;
}
```

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

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



... ...

... ...

... ...

... ...

... ...

0xbfffce60 EIP of main — EIP m

0xbfffce5c SFP of main — SFP m

ESP 0xbfffce58 EIP of func — EIP f

0xbfffce54 \xbf \xff \xce \x14 — SFP f

0xbfffce50 Any input by attacker

0xbfffce4c Any input by attacker

: : ...

0xbfffce18 EIP of main = 0xaabbccdd

EBP 0xbfffce14 Any input by attacker

0xbfffce10 ...

buffer

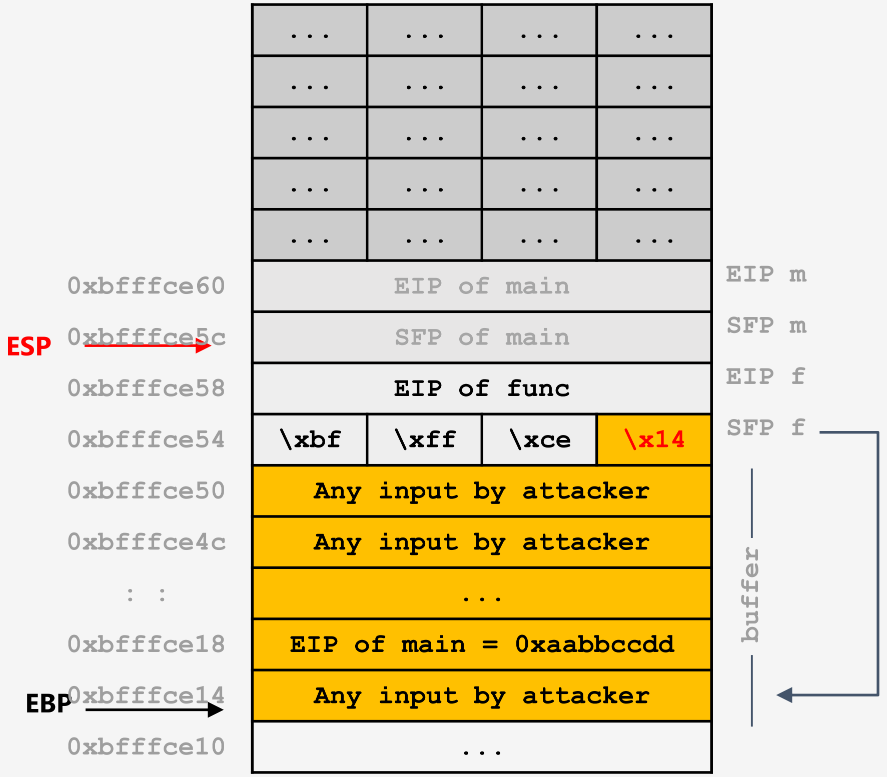
A 32-bit example

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()`**

```
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
```



		...	...	...	...	
		...	...	...	...	
		...	...	...	...	
		...	...	...	...	
		...	...	...	...	
	0xbfffce60	EIP of main				EIP m
ESP	0xbfffce5c	SFP of main				SFP m
	0xbfffce58	EIP of func				EIP f
	0xbfffce54	\xbf	\xff	\xce	\x14	SFP f
	0xbfffce50	Any input by attacker				buffer
	0xbfffce4c	Any input by attacker				
	: :	...				
	0xbfffce18	EIP of main = 0xaabbccdd				
EBP	0xbfffce14	Any input by attacker				
	0xbfffce10	...				

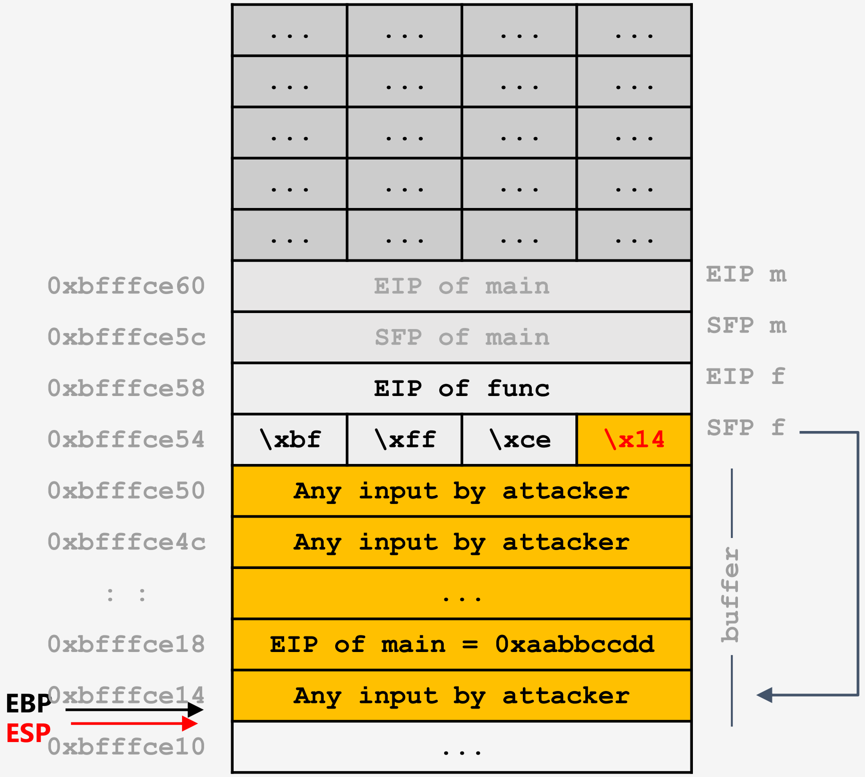
A 32-bit example

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
EIP  --> pop $ebp
         ret
```



	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffce60	EIP of main				EIP m
0xbfffce5c	SFP of main				SFP m
0xbfffce58	EIP of func				EIP f
0xbfffce54	\xbf	\xff	\xce	\x14	SFP f
0xbfffce50	Any input by attacker				buffer
0xbfffce4c	Any input by attacker				
: :	...				
0xbfffce18	EIP of main = 0xaabbccdd				
EBP → 0xbfffce14	Any input by attacker				
ESP → 0xbfffce10	...				

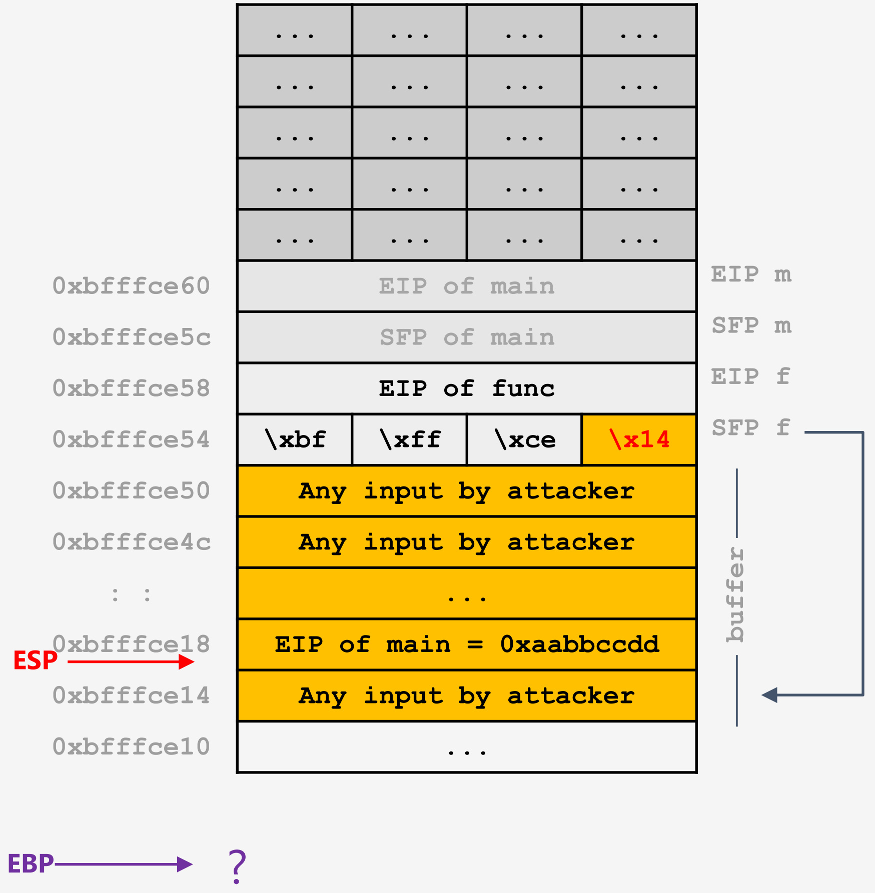
A 32-bit example

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.

```
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
```



	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
	...	...	...	...	
0xbfffce60	EIP of main				EIP m
0xbfffce5c	SFP of main				SFP m
0xbfffce58	EIP of func				EIP f
0xbfffce54	\xbf	\xff	\xce	\x14	SFP f
0xbfffce50	Any input by attacker				buffer
0xbfffce4c	Any input by attacker				
: :	...				
0xbfffce18	EIP of main = 0xaabbccdd				
0xbfffce14	Any input by attacker				
0xbfffce10	...				

A 32-bit example

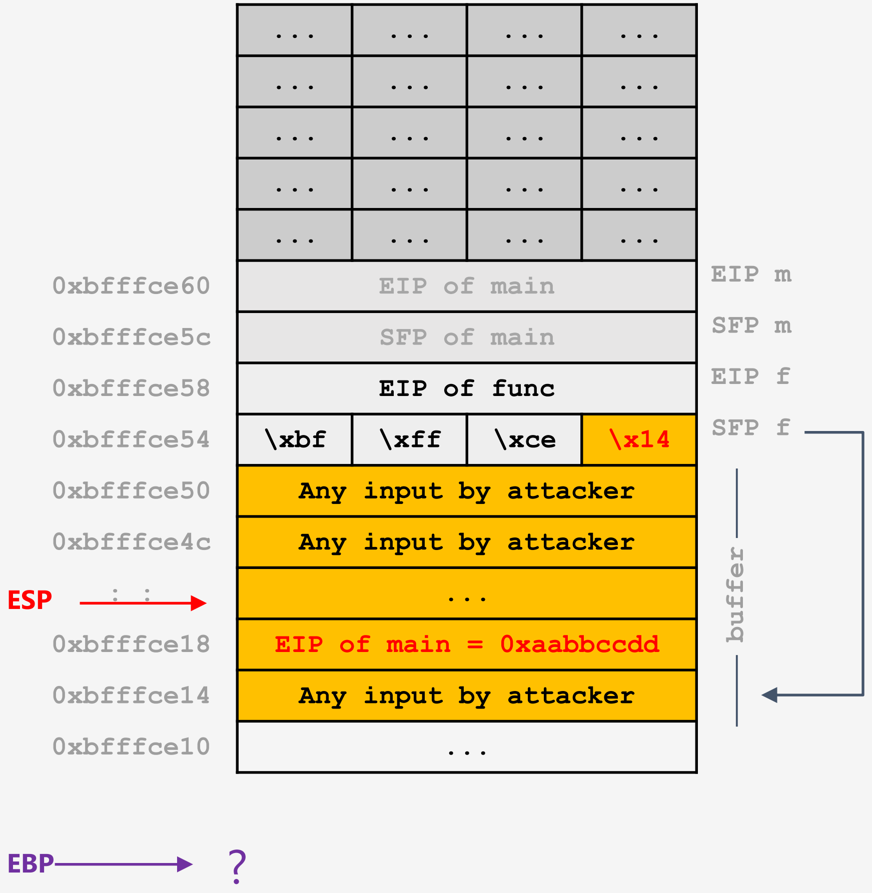
After returning to **main()**, the bad thing starts to happen

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 →



	...	...	...	...
	...	...	...	...
	...	...	...	...
	...	...	...	...
	...	...	...	...
0xbfffce60	EIP of main			
0xbfffce5c	SFP of main			
0xbfffce58	EIP of func			
0xbfffce54	\xbf	\xff	\xce	\x14
0xbfffce50	Any input by attacker			
0xbfffce4c	Any input by attacker			
ESP →	...			
0xbfffce18	EIP of main = 0xaabbccdd			
0xbfffce14	Any input by attacker			
0xbfffce10	...			

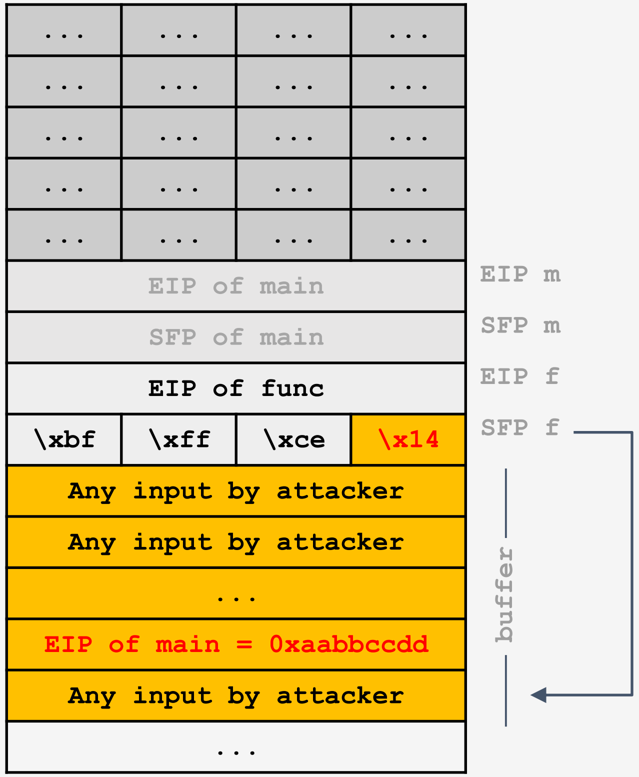
A 32-bit example

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				
...				

Loop Overflow

- Lesson learned
  - User-supplied input should not lead to arbitrary loop iterations
  - Off-by-one vulnerabilities are common

Non-executable Memory and Return Oriented Programming (ROP)

Non-executable stack

- NX (Non-executable stack)
  - Up to now, we have always assumed that it exists a memory region (the stack), which is writable and executable
  - Therefore we can write code into it and execute the code
  - This is not true in modern scenarios!

Non-executable memory

- By default, in modern systems no memory page is
  - Both writable and executable
    - Including the stack
  - In Intel CPUs, this is implemented by setting the `NX bit` in the page table → the CPU enforces the data in those pages cannot be executed
  - The kernel makes sure that when a program is loaded all executable pages are set as non writable
    - GCC option to create a program with executable stack: -z execstack
  - The general security principle is sometimes called
    - W^X (W xor X)
    - Pages cannot be both writable and executable
  - In Windows it is called Data Execution Prevention (DEP)

Non-executable memory

- In Linux, running on Intel CPUs, writable and executable pages can exist but they need to be explicitly created by the application
  - mmap/mprotect syscalls
  - Certain applications, like JIT-ing interpreters, might require this feature
    - For performance reasons, they “create” some code and then execute it
    - Typical example: Javascript interpreters in browsers

Non-executable memory

- Non-executable memory makes exploitation significantly harder
  - Even if we are able to
    - Place shellcode in a memory location we know
    - Detour the normal execution to the shellcode
  - The CPU will refuse to execute it → Segmentation Fault

- How to bypass this?
  - Code Reuse!
  - We can reuse “code” stored in already executable pages

Code Reuse

- We have seen already a case of code reuse
  - Jump to a function

- A more realistic case is **Return-into-libc**
  - We could jump to any function linked by our program

Code Reuse

- A more realistic case is **Return-into-libc**
  - We could jump to any function linked by our program
    - Assuming that we control the stack’s content
      (e.g., stack buffer overflow) we can chain together multiple function calls

  - How do we control function arguments?
    - In Intel 32 bit, arguments are on the stack
      - If we control the stack, we can place them on the stack

Code Reuse

- How do we control arguments?
  - In Intel 64 bit, we can re-use snippets of code copying the stack such content to appropriate registers, such as
    - **pop rdi; ret** (encoded as: 0x5F 0xC3)
  - For instance, suppose we want to call system("/bin/sh"), we can set the stack like this:

Address of pop rdi; ret
Pointer to the string "/bin/sh"
Address of system

Return Oriented Programming (ROP)

- We can generalize the idea of code re-use
    - We re-use gadgets (such as **pop rdi; ret**):
        - Snippets of existing code, terminated by a ret instructions
        - If we control the stack, we can setup the stack so that the execution “jumps” from one gadget to the next one
        - A chain of multiple gadgets (ROP chain) can potentially executes arbitrary operations
            - In theory, gadgets may offer a Turing-complete set of operations
            - In practice, Turing-completeness is not really required
                - An attacker, typically just need to call “a few” syscalls (e.g., execve, open, read, write, ...) to achieve the desired malicious goal

Return Oriented Programming (ROP)

From: “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”, by Hovav Shacham

“Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.”

Return Oriented Programming (ROP)

Return Oriented Programming (ROP)

- As far as they are in executable memory, gadgets may not even be an “original” instruction
- For instance:

  ```
  mov eax, 0xc35f
  ```

  is encoded as:

  ```
  0xB8, 0x5F, 0xC3, 0x00, 0x00
  ```

- If we jump to its second byte we execute: 0x5F, 0xC3

  Which is:

  ```
  pop rdi (0x5F)
  ret     (0xC3)
  ```

Return Oriented Programming (ROP)

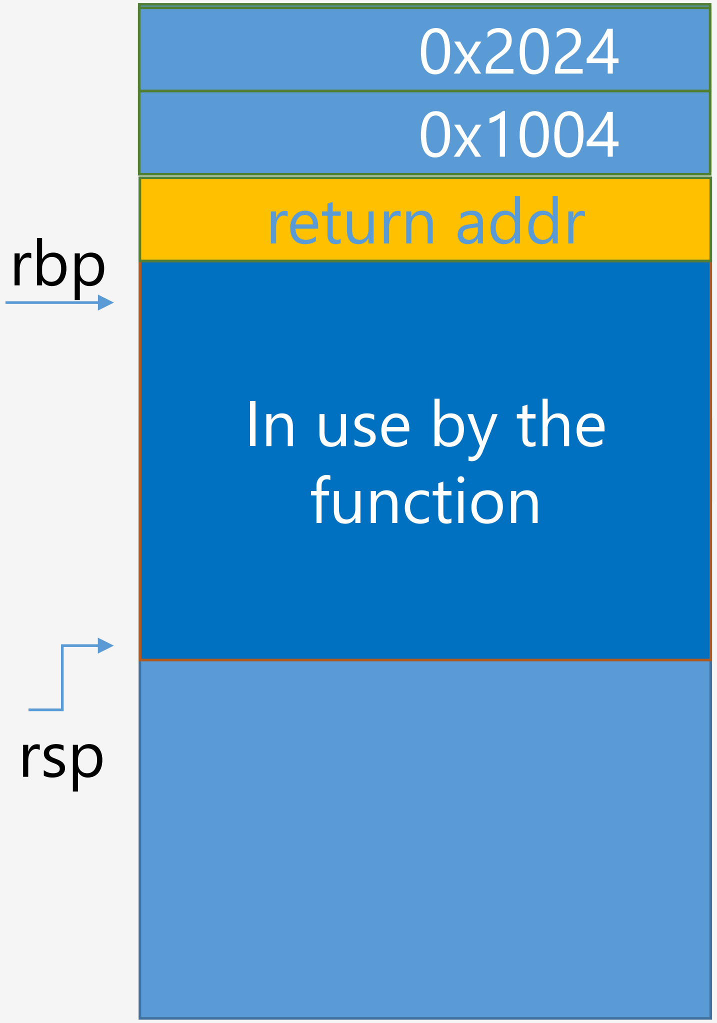
- There are many automated tools to find gadgets
  - Pwntools
  - ROPgadget
  - Ropper
  - …

- Also, automated tools to build complete ROP chains
  - ROPgadget
  - …

ROP: illustration using a simple example of function calls (instead of gadgets)

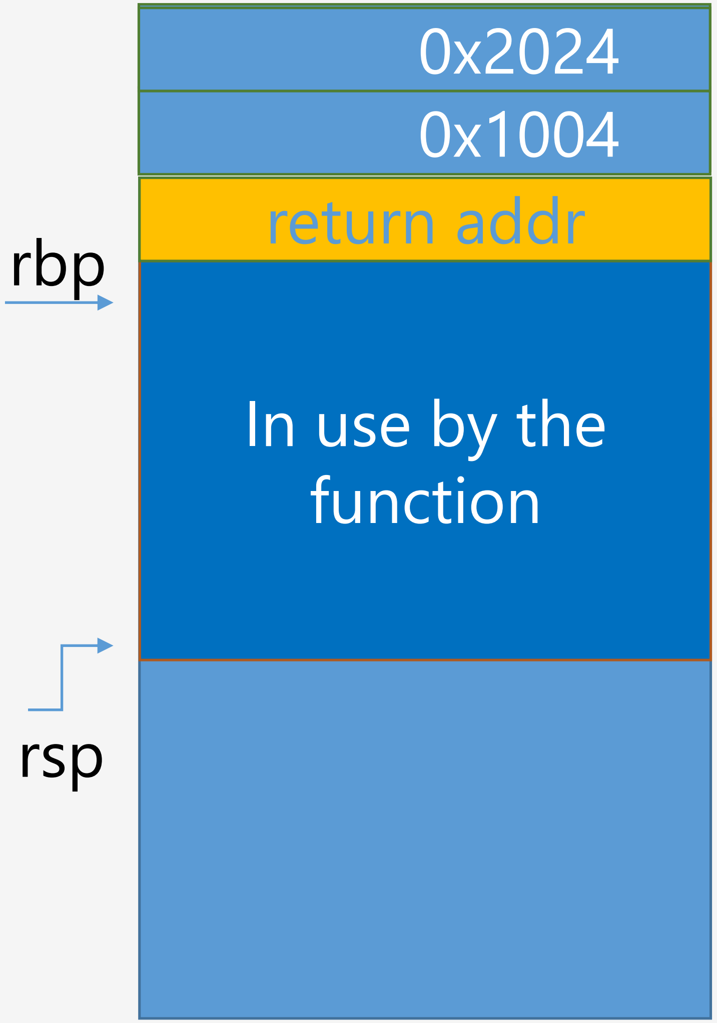
What happens when a function is not called using “call”, but instead using “ret”

- In general “ret” instruction is added by gcc at the end of every function (could be absent if you call __builtin_unreachable())
- Without the “call” instruction, the return address of the function call will not be pushed to the stack, this return address is supposed to be picked up by ret at the end of function call (i.e. ret => pop rip)



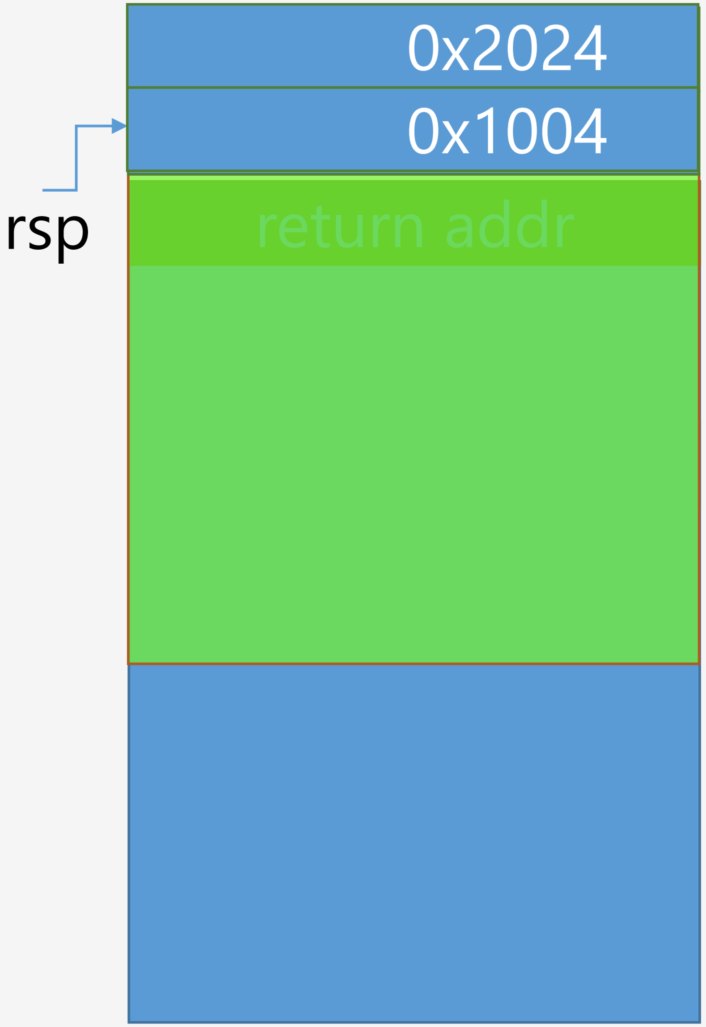
What happens when a function is not called using “call”, but instead using “ret”

- Each function “called” with “ret” will eat up one space at the bottom of the stack



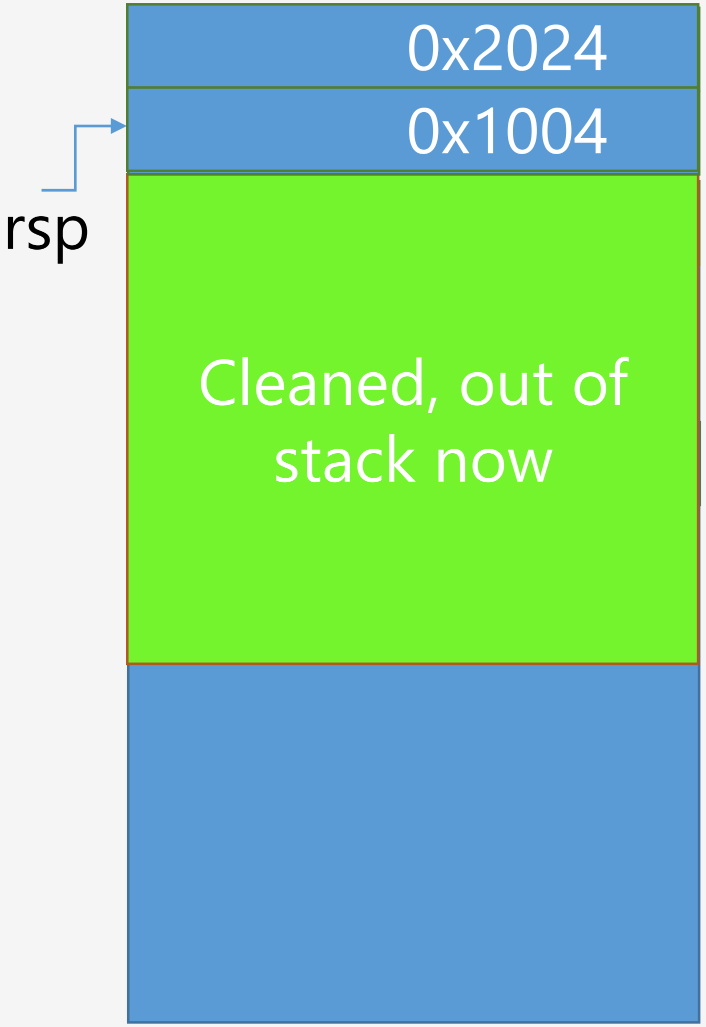
What happens when a function is not called using “call”, but instead using “ret”

- Function call done, clean up the stack
- ret to the “return addr”



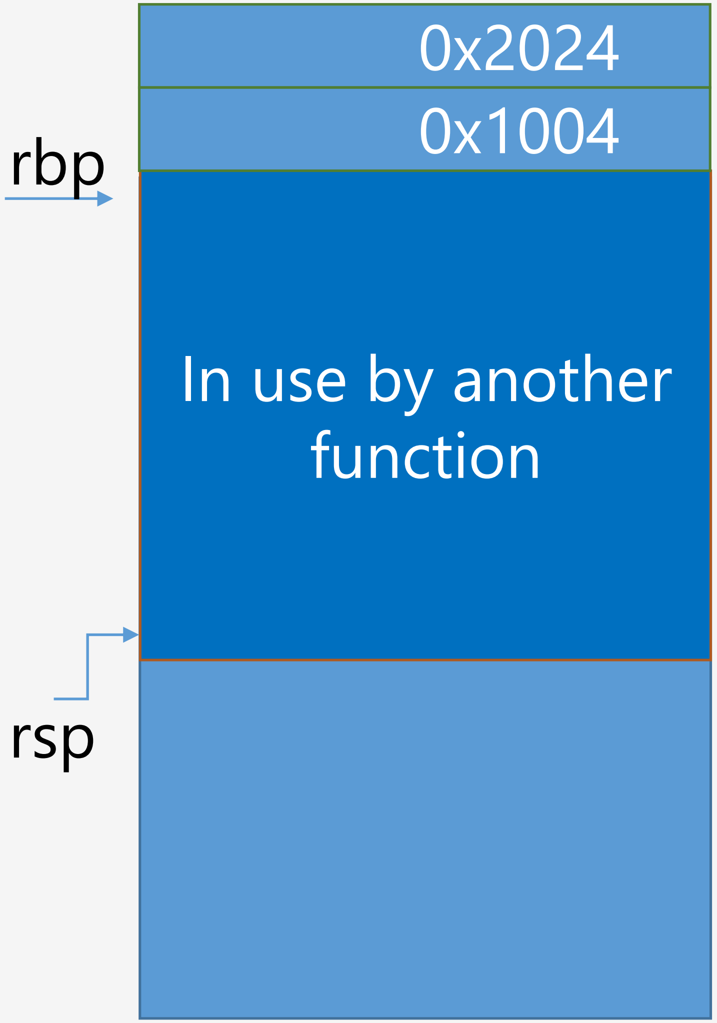
What happens when a function is not called using “call”, but instead using “ret”

- Function call done, clean up the stack
- ret to the “return addr”



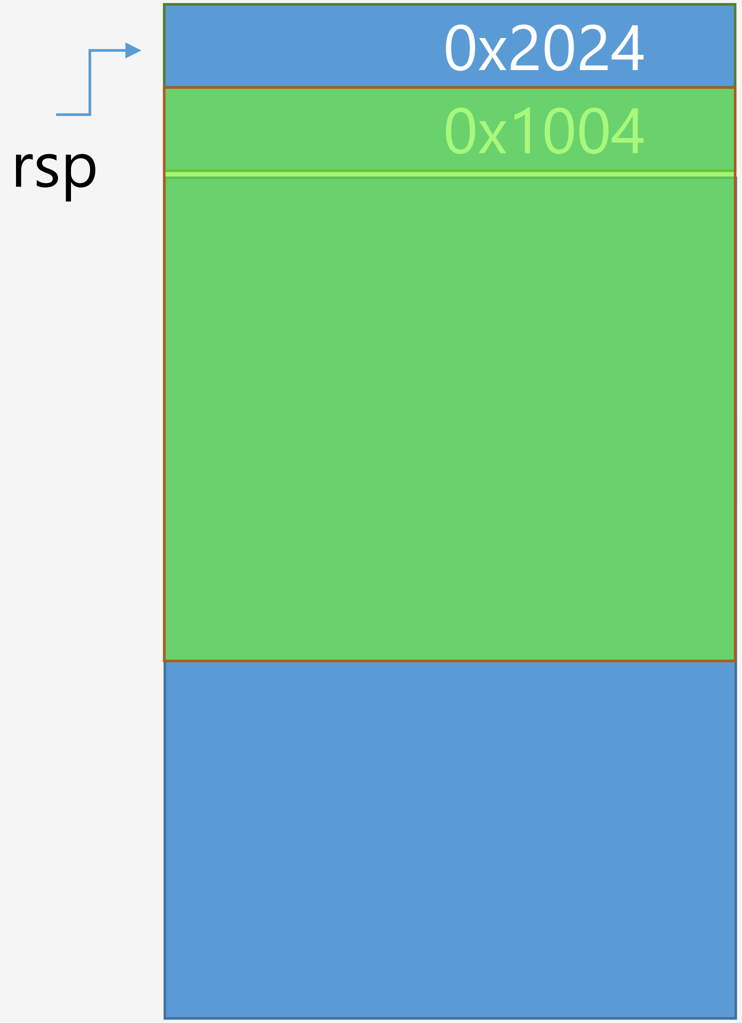
What happens when a function is not called using “call”, but instead using “ret”

Function “called” with “ret”



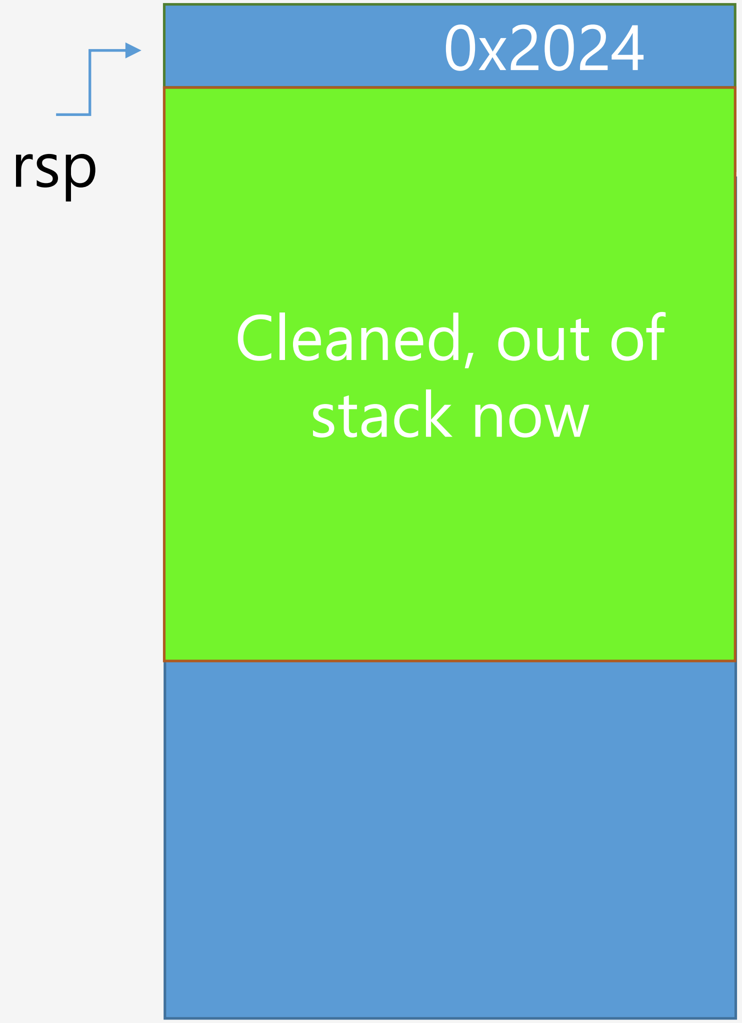
What happens when a function is not called using "call", but instead using "ret"

Function call done, clean up the stack, ret to the "0x1004"



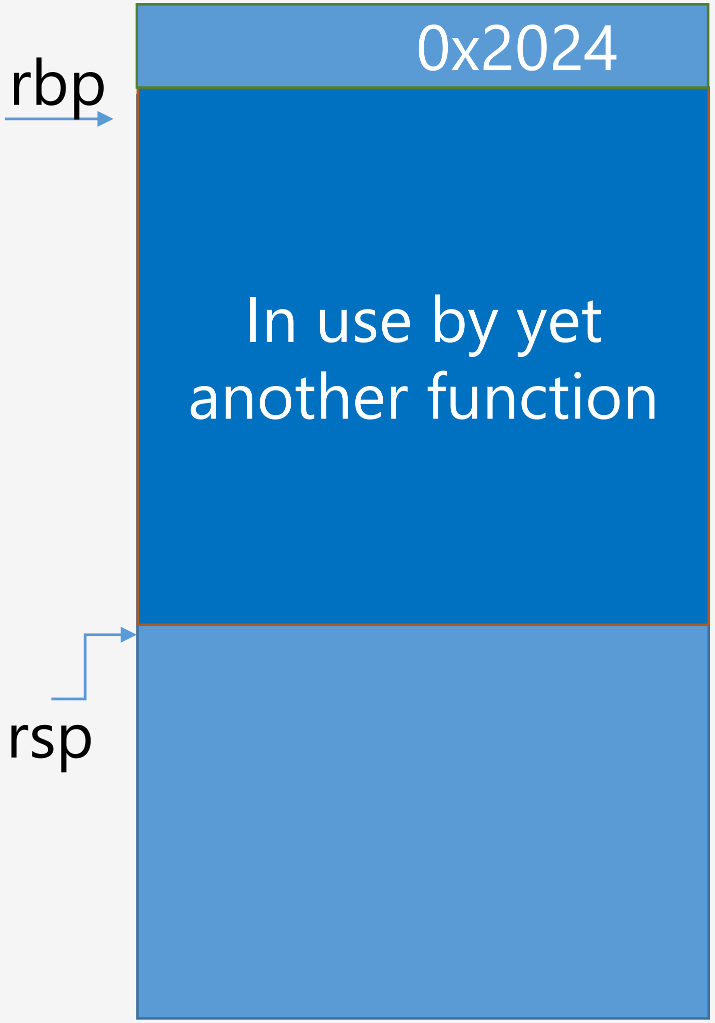
What happens when a function is not called using "call", but instead using "ret"

Function call done, clean up the stack, ret to the address "0x1004"



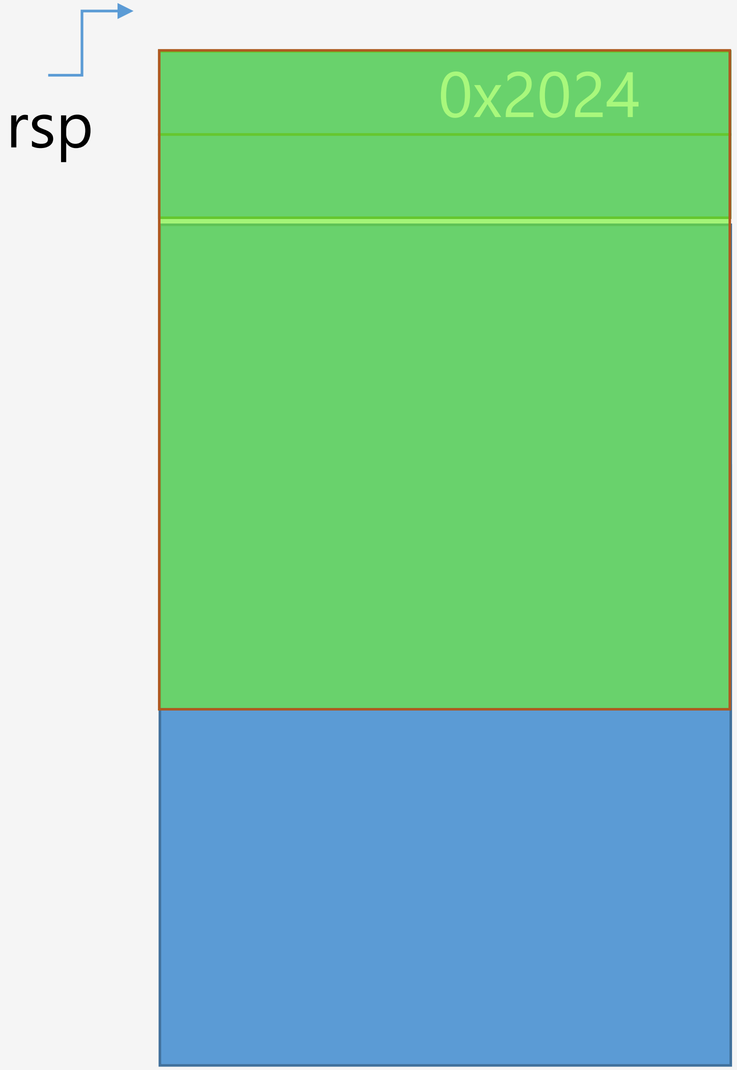
What happens when a function is not called using “call”, but instead using “ret”

Another Function “called” with “ret”



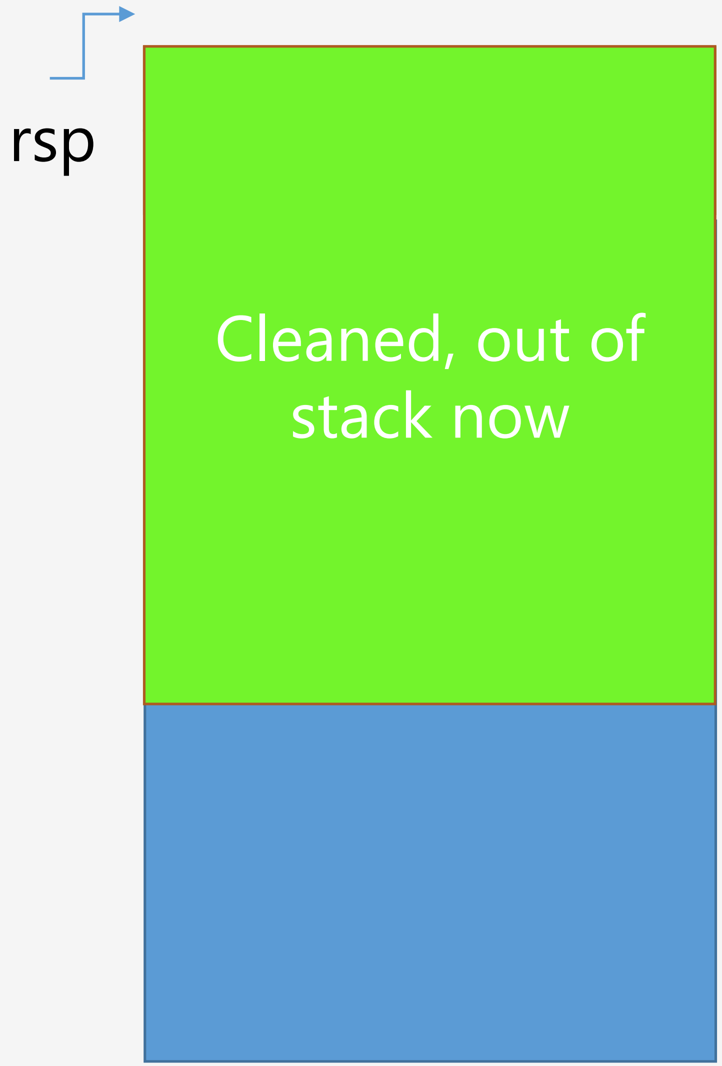
What happens when a function is not called using “call”, but instead using “ret”

Function call done, clean up the stack, ret to the address “0x2024”



What happens when a function is not called using “call”, but instead using “ret”

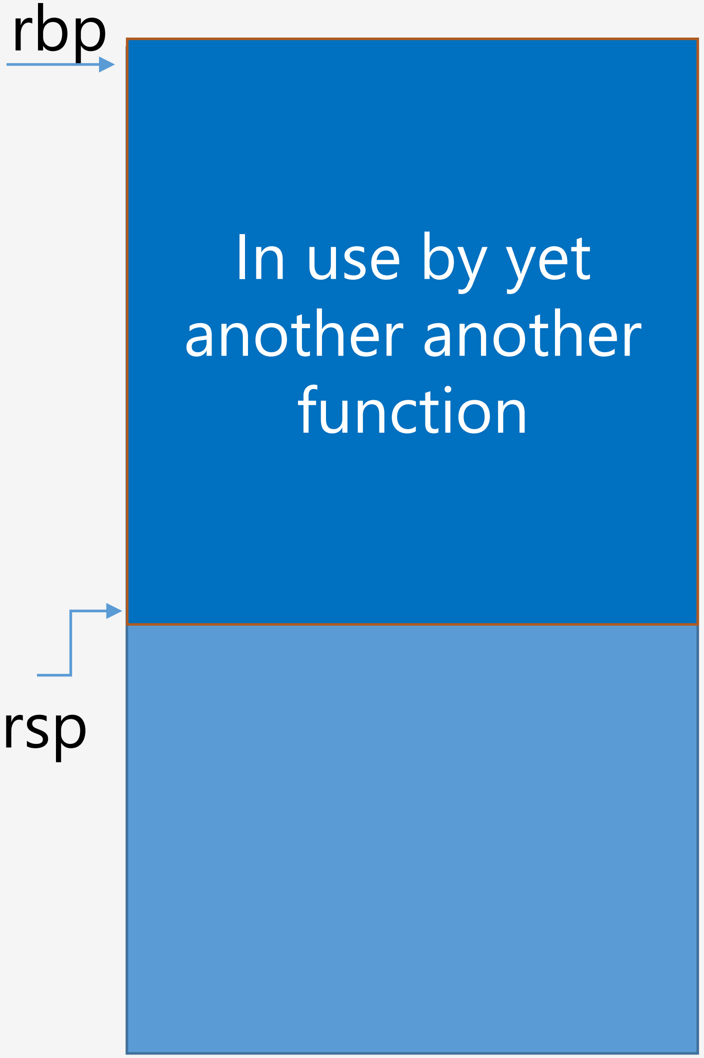
Function call done, clean up the stack, ret to the address “0x2024”



rsp

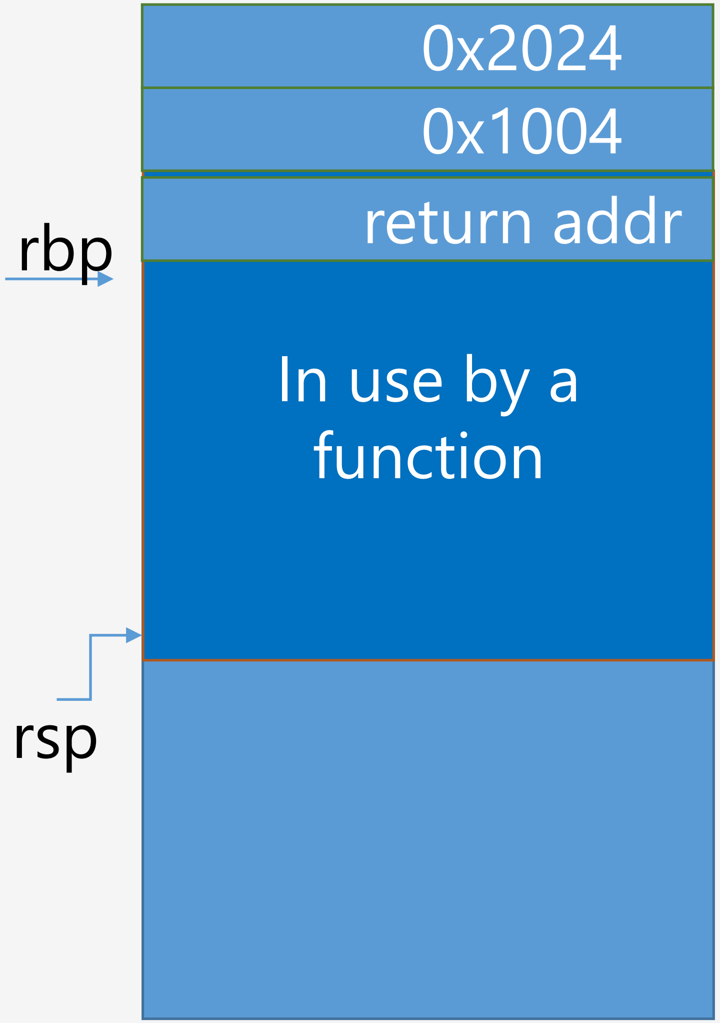
Cleaned, out of stack now

What happens when a function is not called using “call”, but instead using “ret”

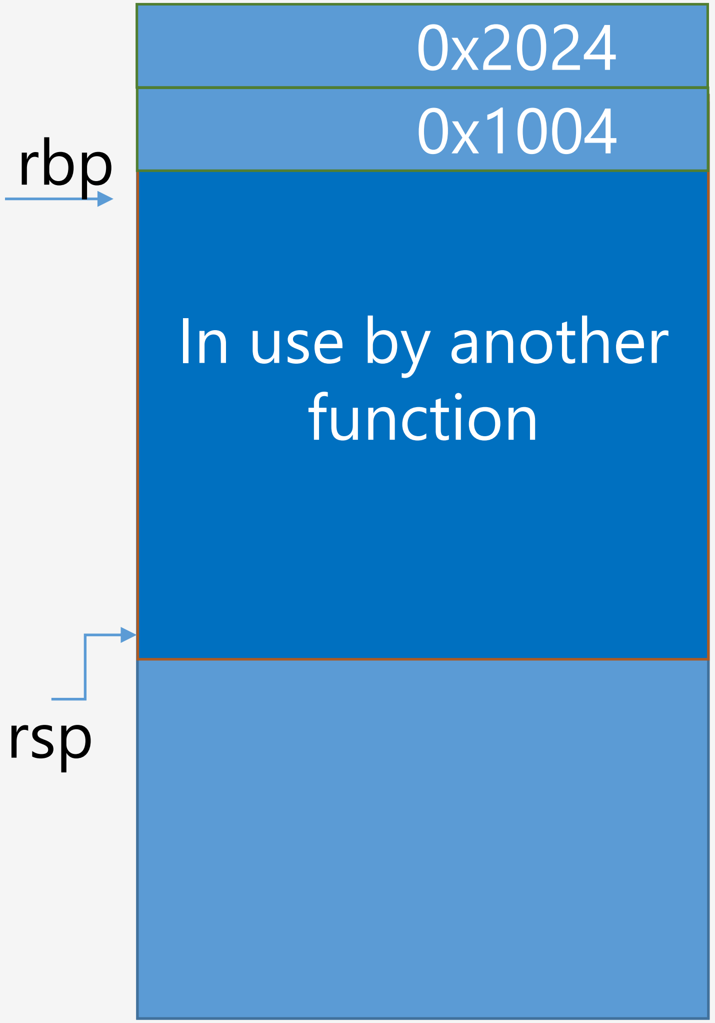


A flashback of what has happened

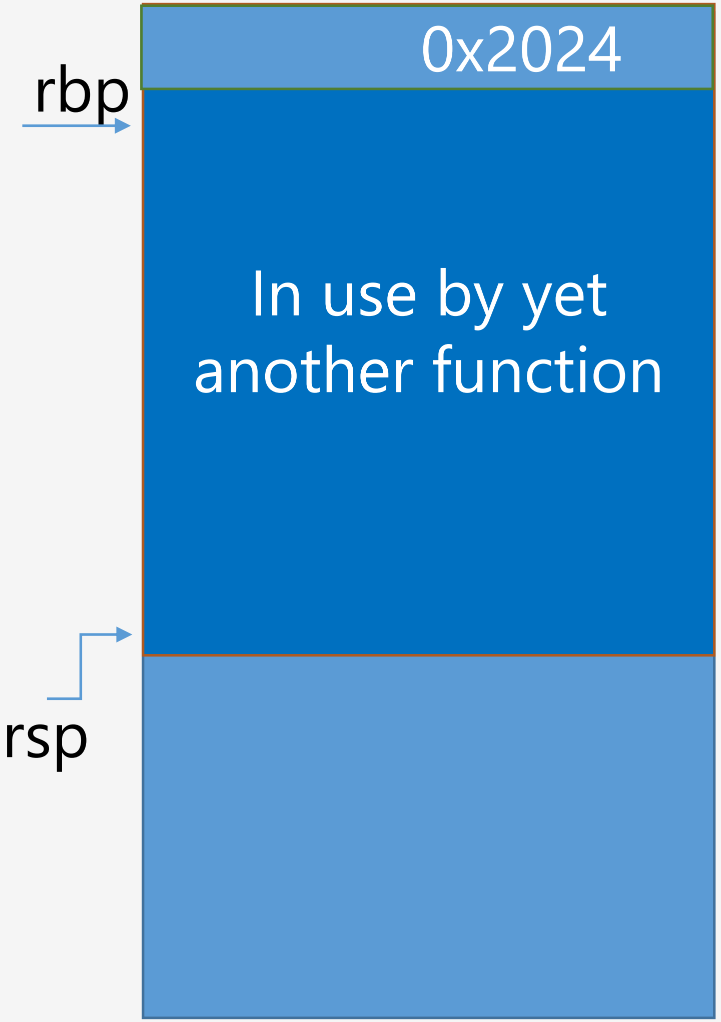
What happens when a function is not called using “call”, but instead using “ret”



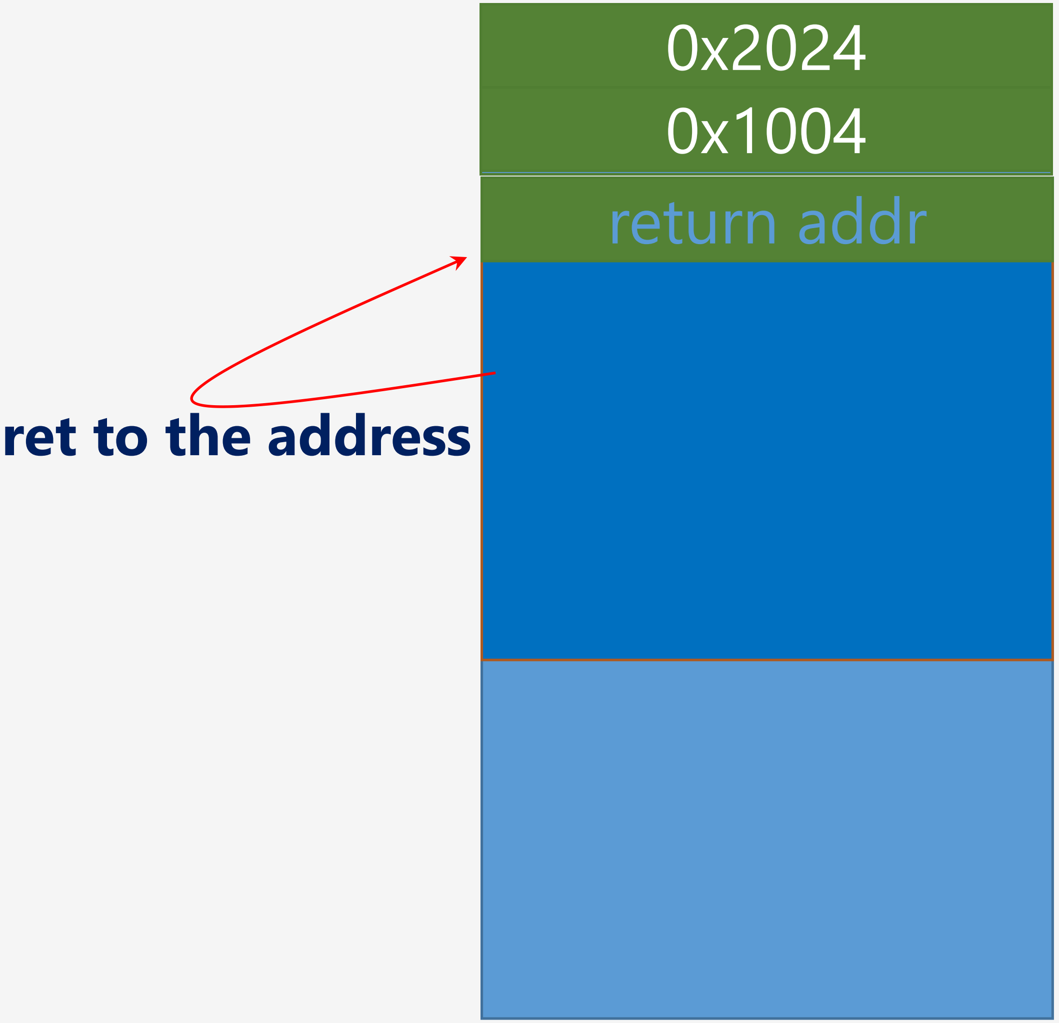
What happens when a function is not called using "call", but instead using "ret"



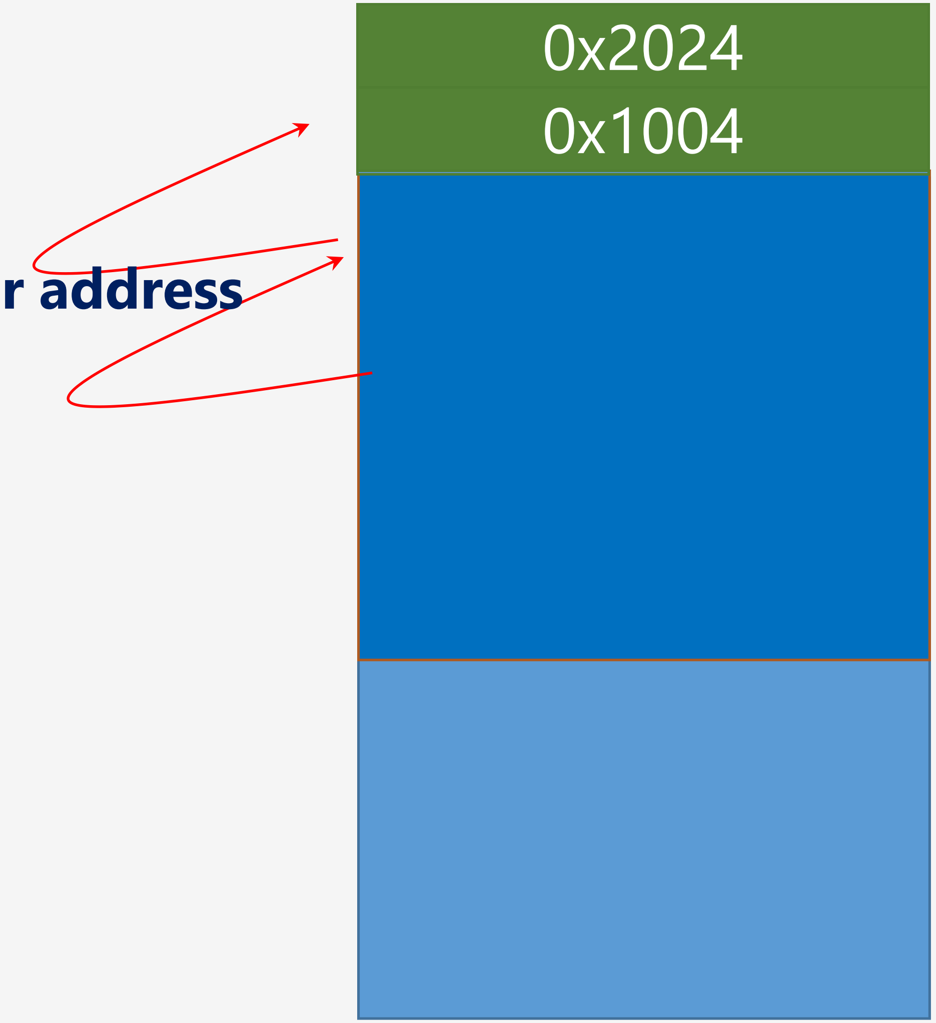
What happens when a function is not called using “call”, but instead using “ret”



Multiple function calls



Multiple function calls



ret to another address

Multiple function calls

ret to yet another address