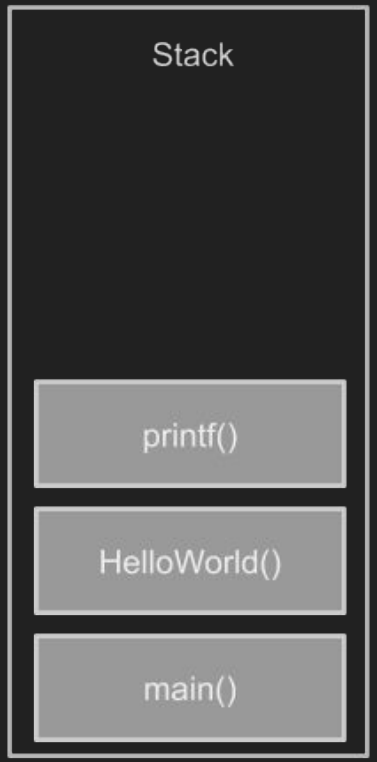
CENG331
HW2 - Recitation

Stack - Buffer Overflow - ROP

What is the stack?

- Obviously a stack data structure
- Used to keep the state information about the active function and the previous functions
- Stores function arguments, local variables, saved registers, return address
- Stack space allocated for a single function is called a stack frame
- Each function call gets its own stack frame



Function Calls

- Different architectures use different calling conventions. [We use x86-64 for Linux or officially “System V AMD64 ABI”]
- Calling conventions define rules about how function arguments are placed on stack, where the return address is stored etc…
- We will look at:
  - Argument passing
  - Return address
  - Callee/Caller-saved registers
  - %rbp

Argument Passing

- On x84-64, the first 6 arguments are passed to the function in registers:
  - `%rdi, %rsi, %rcx, %r8, %r9`
  - Rest of the arguments are put on stack
- Return value of a function is stored and passed to the caller in `%rax` register

```
int main(){
  int a=1, b=2;
  int res1;
  res1 = normal(a,b);
  return 0;
}
```

```
movl   $0x1,-0xc(%rbp)
movl   $0x2,-0x8(%rbp)
mov    -0x8(%rbp),%edx
mov    -0xc(%rbp),%eax
mov    %edx,%esi
mov    %eax,%edi
callq  1119 <normal>
mov    %eax,-0x4(%rbp)
```

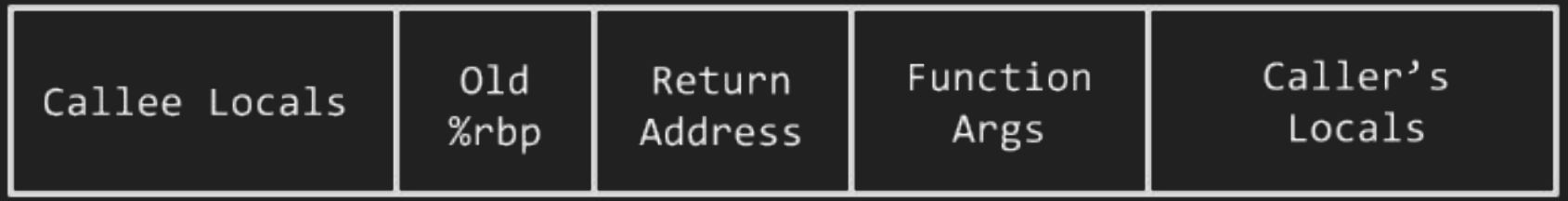
More than 6 arguments

```
int main(){
  int a=1, b=2, c=3, d=4, e=5, f=6, g=7, h=8;
  int res1;
  res1 = another(a,b,c,d,e,f,g,h);
  return 0;
}
```

```
mov    -0x14(%rbp),%r9d
mov    -0x18(%rbp),%r8d
mov    -0x1c(%rbp),%ecx
mov    -0x20(%rbp),%edx
mov    -0x24(%rbp),%esi
mov    -0x28(%rbp),%eax
mov    -0xc(%rbp),%edi
push   %rdi
mov    -0x10(%rbp),%edi
push   %rdi
mov    %eax,%edi
callq  115b <another>
```

Return address

- `callq <func>` ➥ `push [next instruction]; jmp <func>`



Lower Addresses
"Stack Top"

Higher Addresses
"Stack Bottom"

```
        ⋮
push   %rdi
mov    -0x10(%rbp),%edi
push   %rdi
mov    %eax,%edi
callq  115b <another>
mov    %rax, -0x8(%rbp)
        ⋮
```

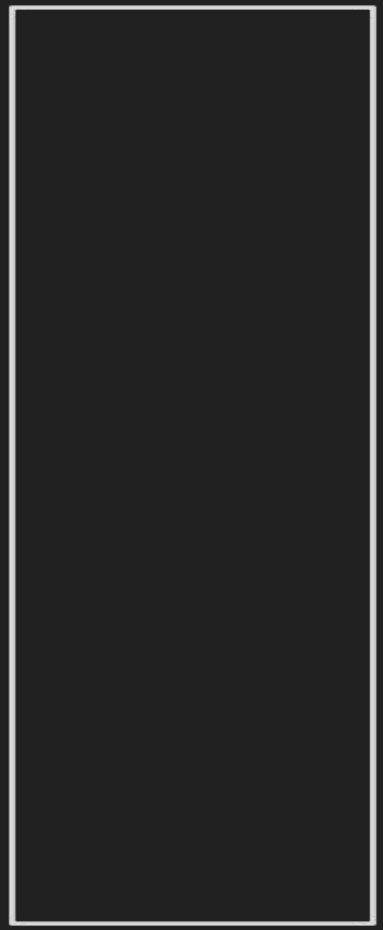
Function call example

Caller

1. Store arguments
2. Save caller-saved registers
3. Execute `callq` (store return addr)

Callee

1. Put return value in `%rax`
2. Restore stack top `%rsp`
3. Execute `retq` (`pop %rip`)



Caller / Callee Saved Registers

- Calling convention gives certain guarantees about values of registers across function calls
- Certain register values must be preserved across different functions
- Callee may use these registers, but must restore them before returning (These are called callee-saved registers)
- On x86-64 Linux, `%rbp`, `%rbx`, `%r12`, `%r13`, `%r14`, and `%r15` are callee-saved [`%rsp` and `%rip` also if you think about it]. Other registers are caller saved

%rbp

- %rpb is called base pointer [frame pointer]
- Points to the start of the stack frame
- Compiler may optimize it away or may keep it

Buffer Overflow

- Buffer overflows happen when you put a lot of data in a small space
- This can cause undefined behavior of the program
- Can be used to hijack the running program and execute arbitrary code

In call stack, if we can overwrite “ret”, we can execute any code we want

Very Highly Recommended Reading:
Smashing The Stack For Fun And Profit by Aleph One [Phrack Magazine Issue 49]

```c
void holy_grail() {
  char b[1024];
  char a[256];
  fputs("What is the airspeed velocity of an unladen swallow?\n", stdout);
  strcpy(a, "echo \"I am Arthur, King of the Britons\n\"");
  fgets(b, 1024, stdin);
  strcat(a, b);
  for (int i = 0; i < strlen(b); i++) {
    if(b[i] == '\n') continue;
    if (b[i] < '0' || b[i] > '9') {
      fputs("You can't expect to wield supreme "
            "executive power just because some "
            "watery tart threw a sword at you.\n",
            stdout);
      exit(0);
    }
  }
  fputs("Who are you, who are so wise in the "
        "ways of science?\n", stdout);
  system(a);
  exit(0);
}
```

Return Oriented Programming

- Stack memory can be marked as non-executable
  - No shellcode injection, so no arbitrary code
- Only meaningful thing to change is return address

Return Oriented Programming to the rescue. We want to:

i. Set registers (i.e. function args)
ii. Call functions

Return Oriented Programming

For example assume we want to execute this arbitrary code:

```
movq   $0x50, %rdi
movq   $0xff, %rsi
callq  target
```

Or since we control the stack

```
popq   %rdi
popq   %rsi
callq  target
```

We just need the “gadgets” for `popq %rdi` and `popq %rsi` and the address of `target`.

A gadget is an instruction(s) in the executable part of the program which ends with a return instruction (c3). And a ROP-Chain is the series of gadget addresses and values we intentionally put on the stack.

```
0xff1102: popq %rsi;
0xff1103: retq;
   …
0xff4312: popq %rdi;
0xff4313: retq;
```

We put following on stack overwriting the original return address

```
0xff4312 <- overwrite
0x50       original
0xff1102   return
0xff
0xff6699 <- addr
            of
            target
```