Chapter 5. ROP (Return-oriented Programming)

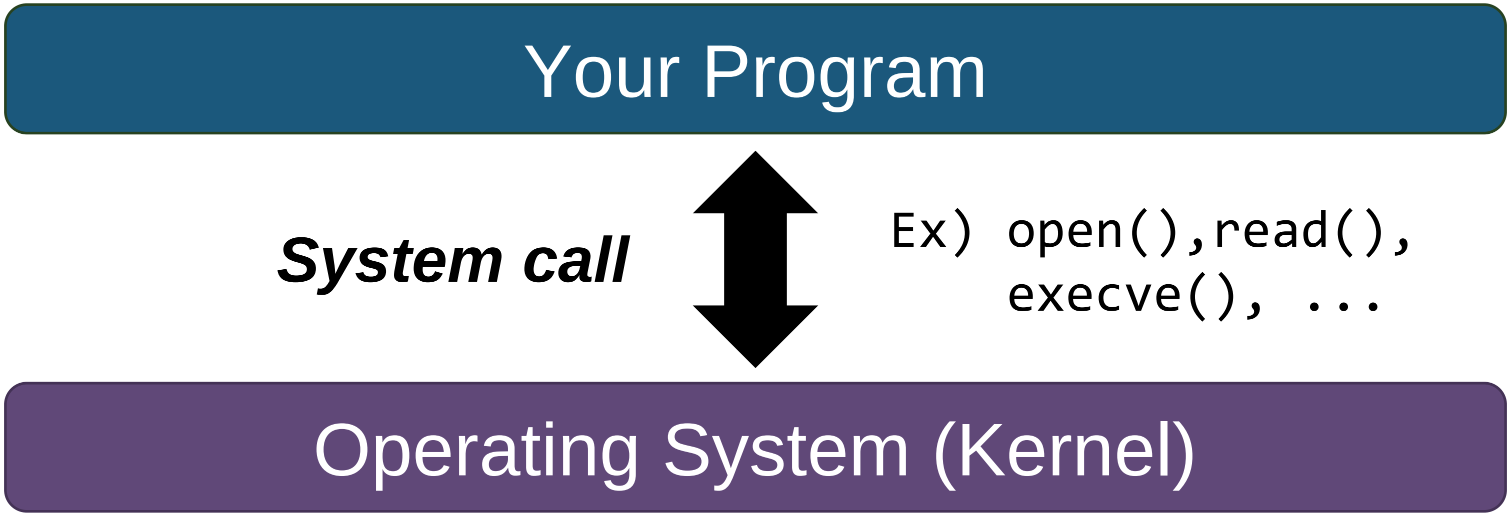
Prof. Jaeseung Choi
Dept. of Computer Science and Engineering
Sogang University

Topics

- **Background: system call**
- **The second round of war between attacker vs. defender**
  - Code reuse attack (**ROP**) to bypass mitigation with **ASLR**

System Call (Syscall)

- **Assume that you have written a C program that opens and reads a file**
  - What kind of x86-64 assembly instruction can we use?
  - No instruction is solely reserved for opening or reading a file
  - Instead, you must make some **request to OS (system call)**
    - The OS will do the task for you and return the result



Assembly Code For System Call

- **It's similar to function call, but the function is in kernel**
- **When you setup particular registers properly and execute `syscall` instruction, system call is invoked**
- **What does actually happen during the system call?**
  - Take ***System Programming*** or ***Operating System*** course

```
...
mov    $0, %rsi        # %rsi must contain flag (option)
mov    ..., %rdi       # %rdi must point to filename string
mov    $0x2, %rax      # System call ID of open() is 2
syscall
```

System Call Wrapper

- **You can also invoke system calls like `open()`, `read()` or `execve()` in your C source code**
  - You are actually calling a wrapper function around the `syscall`

Your Program

```
...
inf fd = open("a.txt")
...
```

```
...
call 0x2000 # <open()>
...
```

Library

```
<open syscall wrapper>
0x2000: ...
...
0x200a: mov $0x2, %rax
0x200f: syscall
...
```

High-Level Library Functions

- **You are probably more familiar with higher-level functions like `fopen()`, `fgets()`, `fread()`, etc.**
  - Such functions are implemented by using system calls internally

Your Program

```
...
File *f = fopen("a.txt")
...
```

```
...
call 0x1000 # <fopen()>
...
```

Library

```
<fopen() function>
0x1000: ...
...
0x1030: call 0x2000

<open syscall wrapper>
0x2000: ...
```

Topics

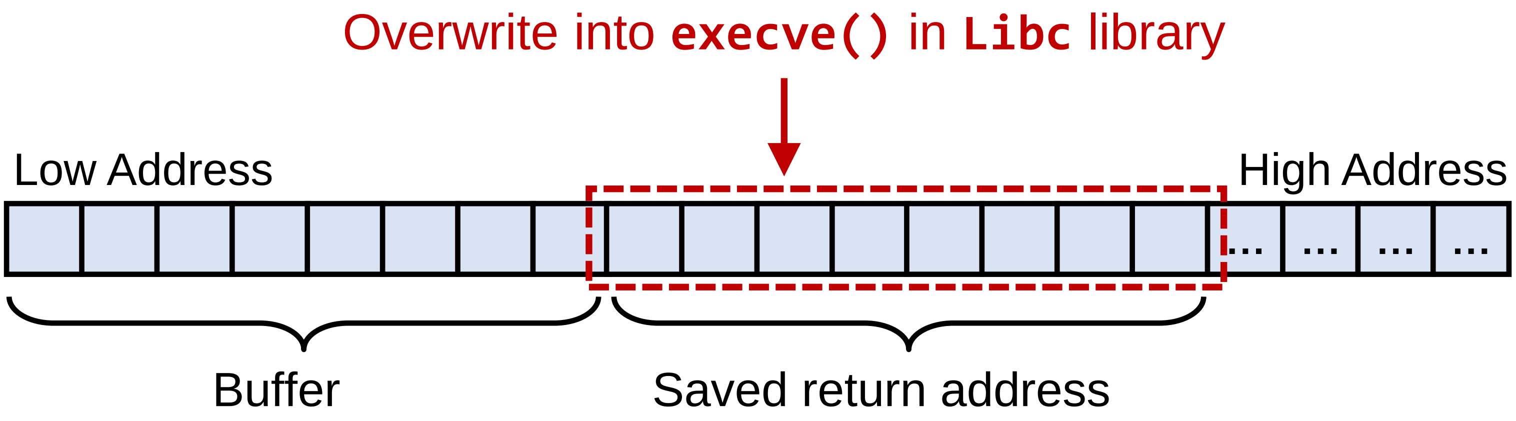
- **Background: system call**
- **The second round of war between attacker vs. defender**
  - Code reuse attack (**ROP**) to bypass mitigation with **ASLR**

Code Reuse Attack

- **Review: By introducing NX (non-executable memory), attack with shellcode was effectively mitigated**
  - Injected shellcode cannot be executed anymore
- **However, the attackers found a different way:**
  - **Code reuse attack**: use the existing code to achieve the goal
  - Ex) By executing **`execve()`** function in C library (`libc`) with **`"/bin/sh"`** as argument, the hacker can spawn a shell
  - Library code **must be executable**, so NX cannot prevent this

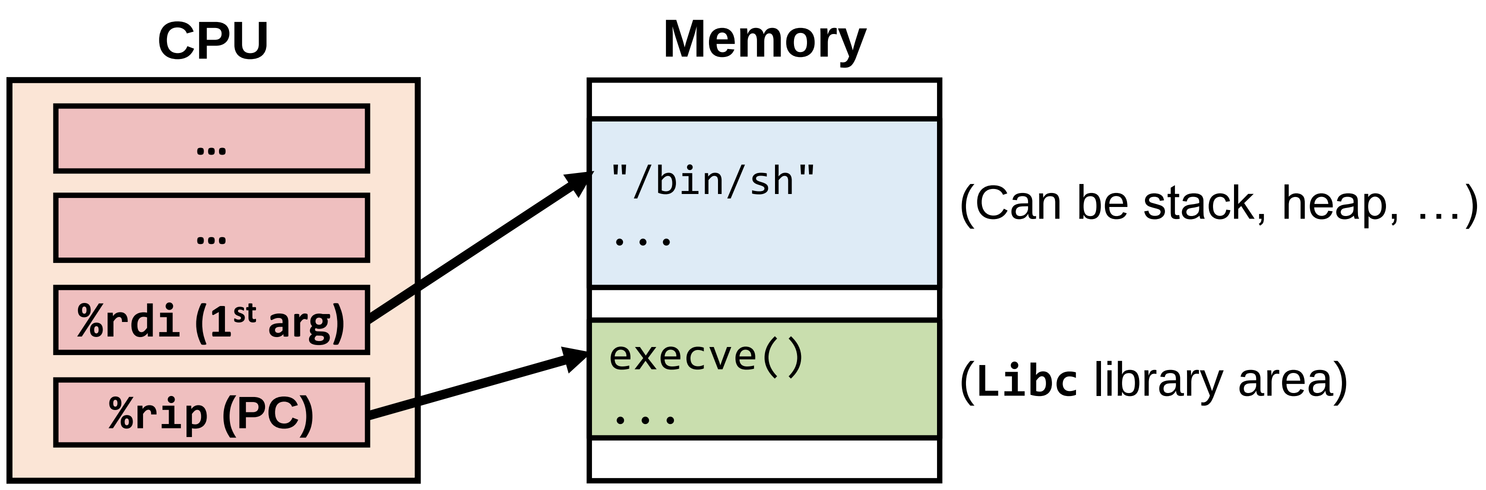
Return-to-Libc Attack

- **Before 2000s, code reuse attack was easy**
  - **32bit x86 system** was the mainstream: according to its calling convention, **controlling the argument was easy**
  - Also, library code was placed in easily **predictable address**
- **This primitive form of attack was called *return-to-libc***
  - But we will not cover this attack deeply **(outdated)**
  - Instead, we will focus on **x86-64 systems!**



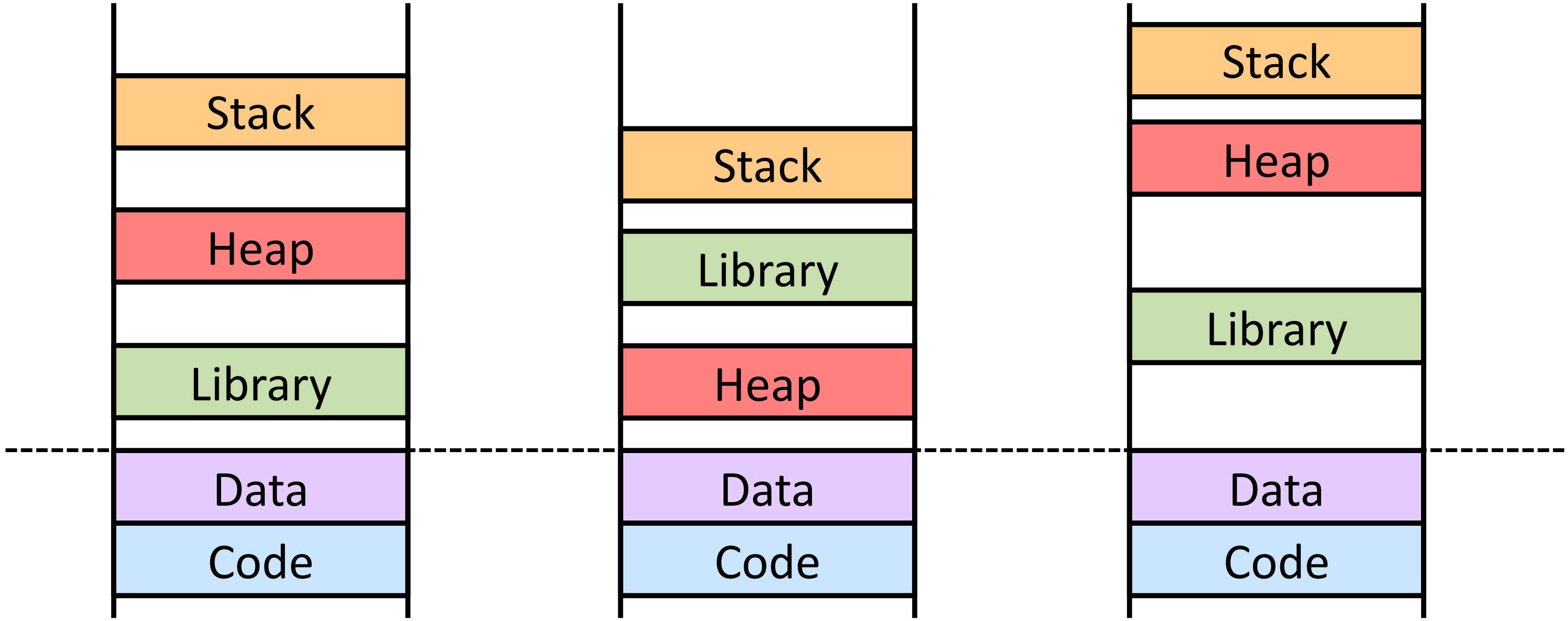
Challenge #1

- **In the calling convention of x86-64, manipulating the arguments of a function is difficult**
- **Even if we hijack the control to `execve()`, how can we pass `"/bin/sh"` argument to this function?**
  - We must manipulate **`%rdi`** register to point to string **`"/bin/sh"`**
  - We could corrupt **`%rip`** register, but what about other registers?



Challenge #2

- **In the early 2000s, address space layout randomization (ASLR) mitigation was introduced**
  - Each memory section address is randomized **per execution**
  - However, for performance reason, the ***Code*** and ***Data*** sections were not randomized until recently



We can overcome the first challenge with a code-reuse attack called ROP (return-oriented programming)

ROP Gadget

- **ROP attack uses small code chunks called gadget**
  - Instruction sequence that ends with `ret` instruction
  - Your program often includes many unintended gadgets
- **Consider `mov $0xc35f, %rax` instruction below**
  - What if we jump to **`0x1003`**, which is middle of this instruction?
  - Bytes are interpreted as instruction sequence `pop %rdi; ret`

```
Addr      Bytes                  Instruction
0x1000    48 c7 c0 5f c3 00 00   mov     $0xc35f, %rax
0x1007    ...
```

Re-interpret

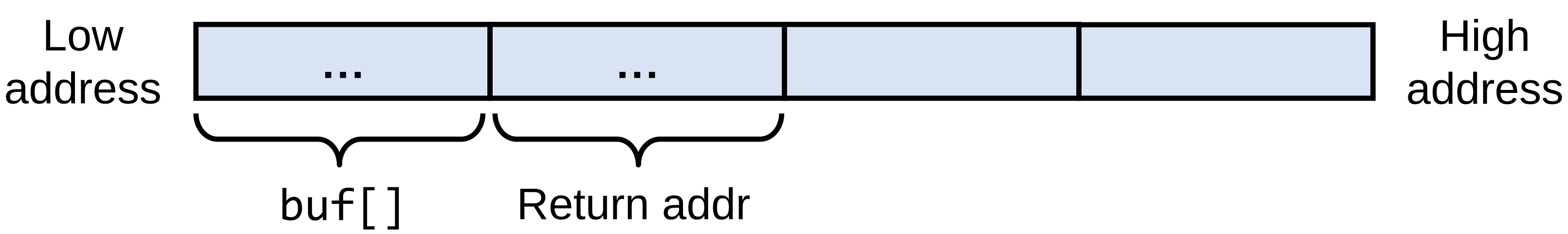
```
Addr      Bytes                  Instruction
0x1003    5f                     pop %rdi
0x1004    c3                     ret
```

What can we do with gadgets?

- **Assume function `vuln()` with a buffer overflow**
  - BOF allows us to overwrite the saved return address
  - Assume no unused space between **`buf[]`** and return address

```
void vuln(void) {
    char buf[8];
    gets(buf); // Buffer overflow
}
```

(Before the BOF. Assume each block is 8-byte)

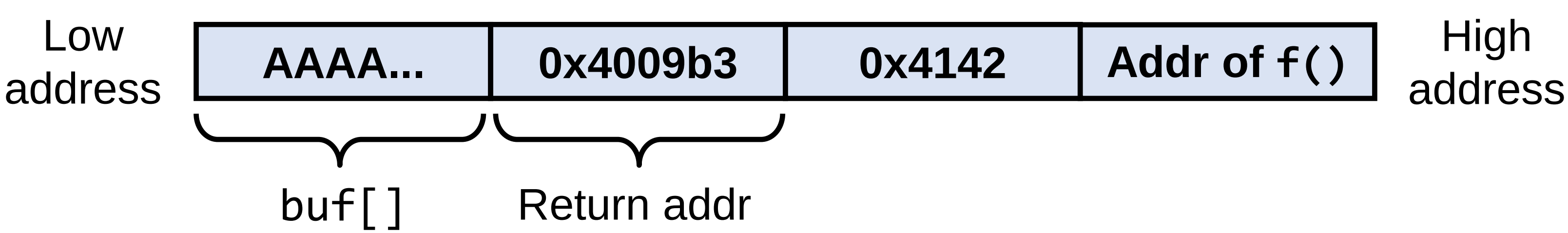


What can we do with gadgets?

- **We will overwrite the saved return address like below**
  - Note that we have corrupted **beyond** the saved return address

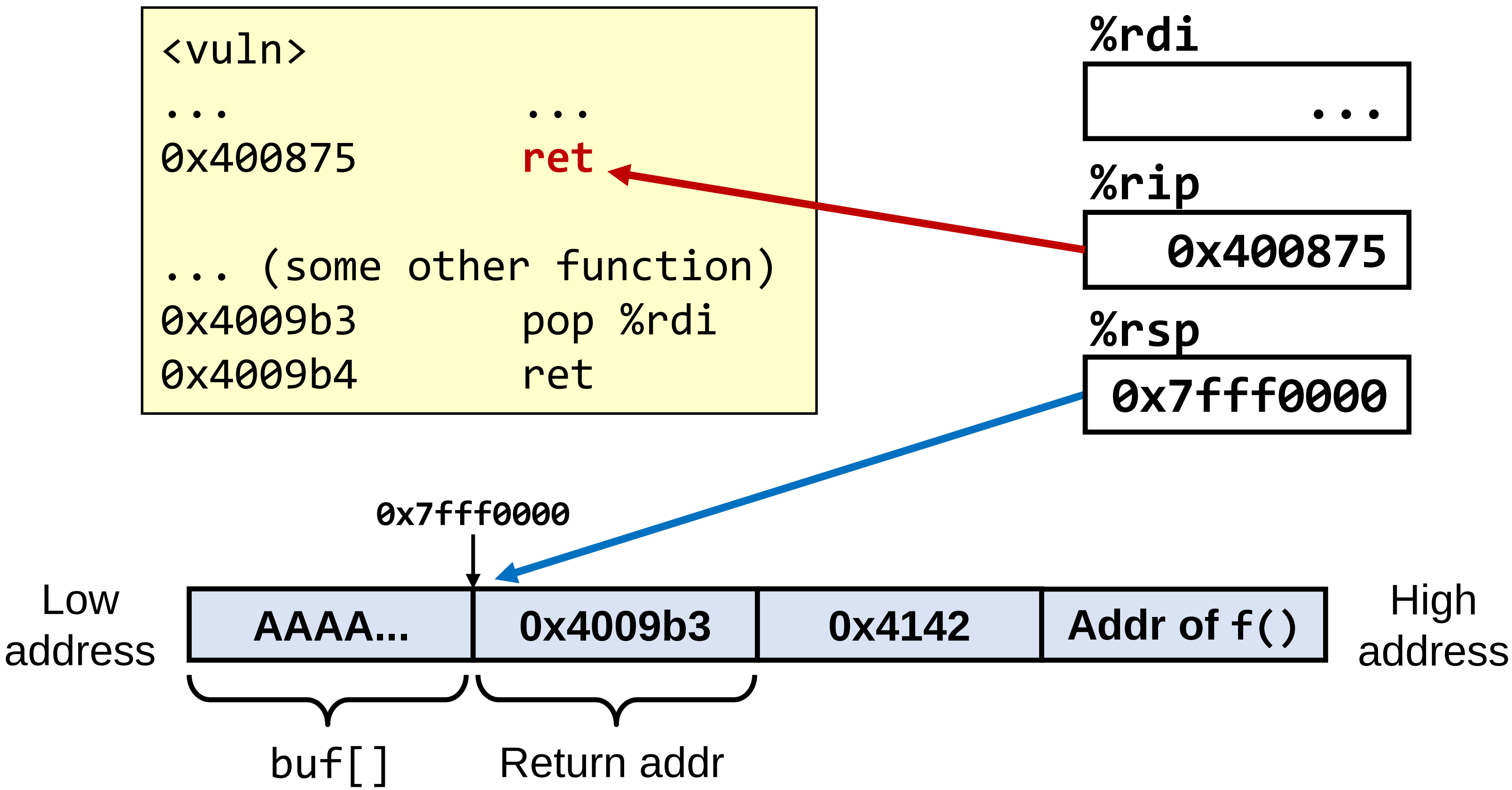
```
<vuln>
...             ...
0x400875        ret             # Last instruction

... (some other function)
0x4009b3        pop %rdi        # Gadget to use
0x4009b4        ret
```

(Corrupted with BOF. `f()` is arbitrary function we choose)



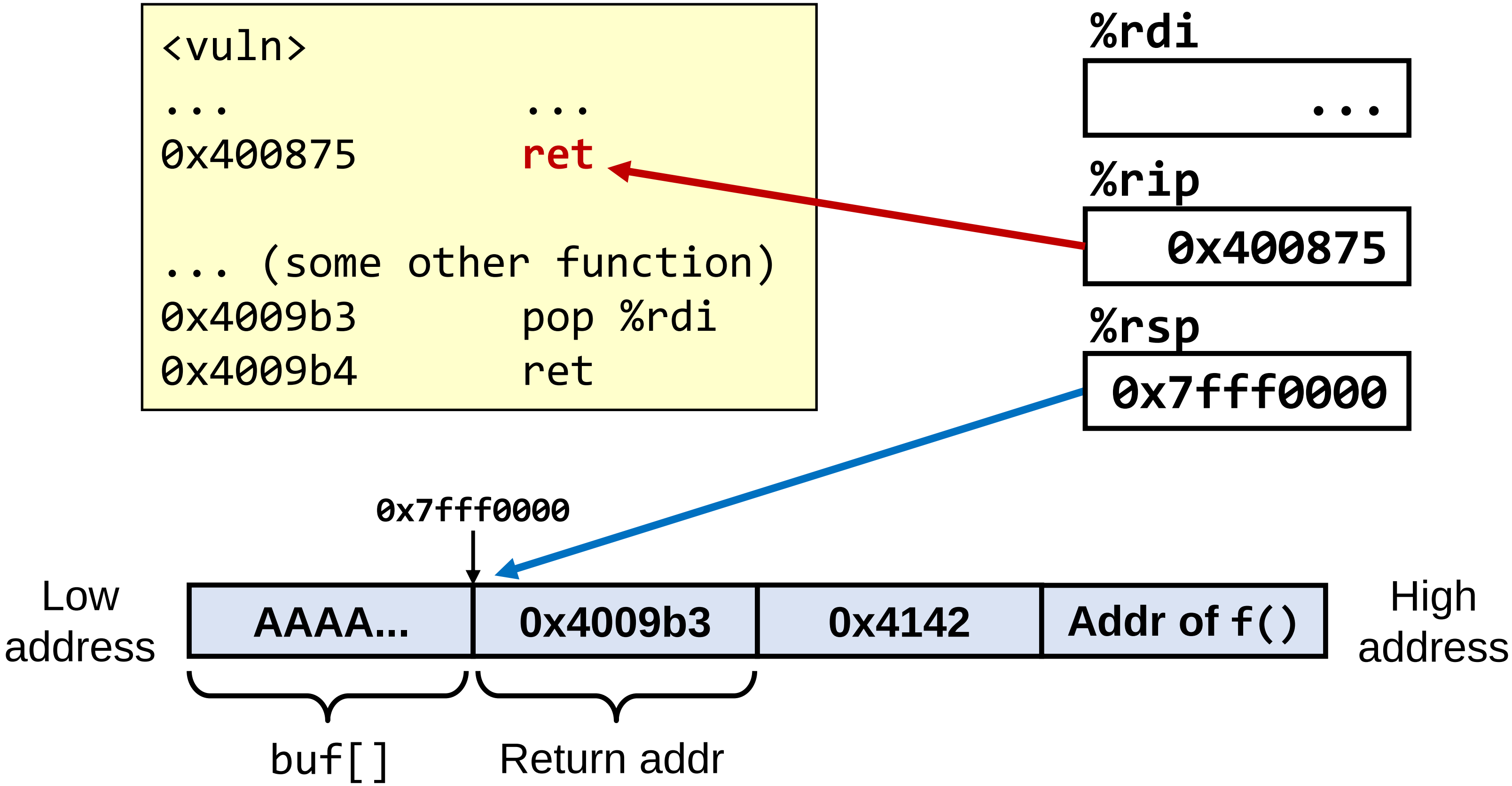
What can we do with gadgets?

- **Now, assume `vuln()` is about to return with `ret`**
  - Let’s see what happens at this moment, step by step



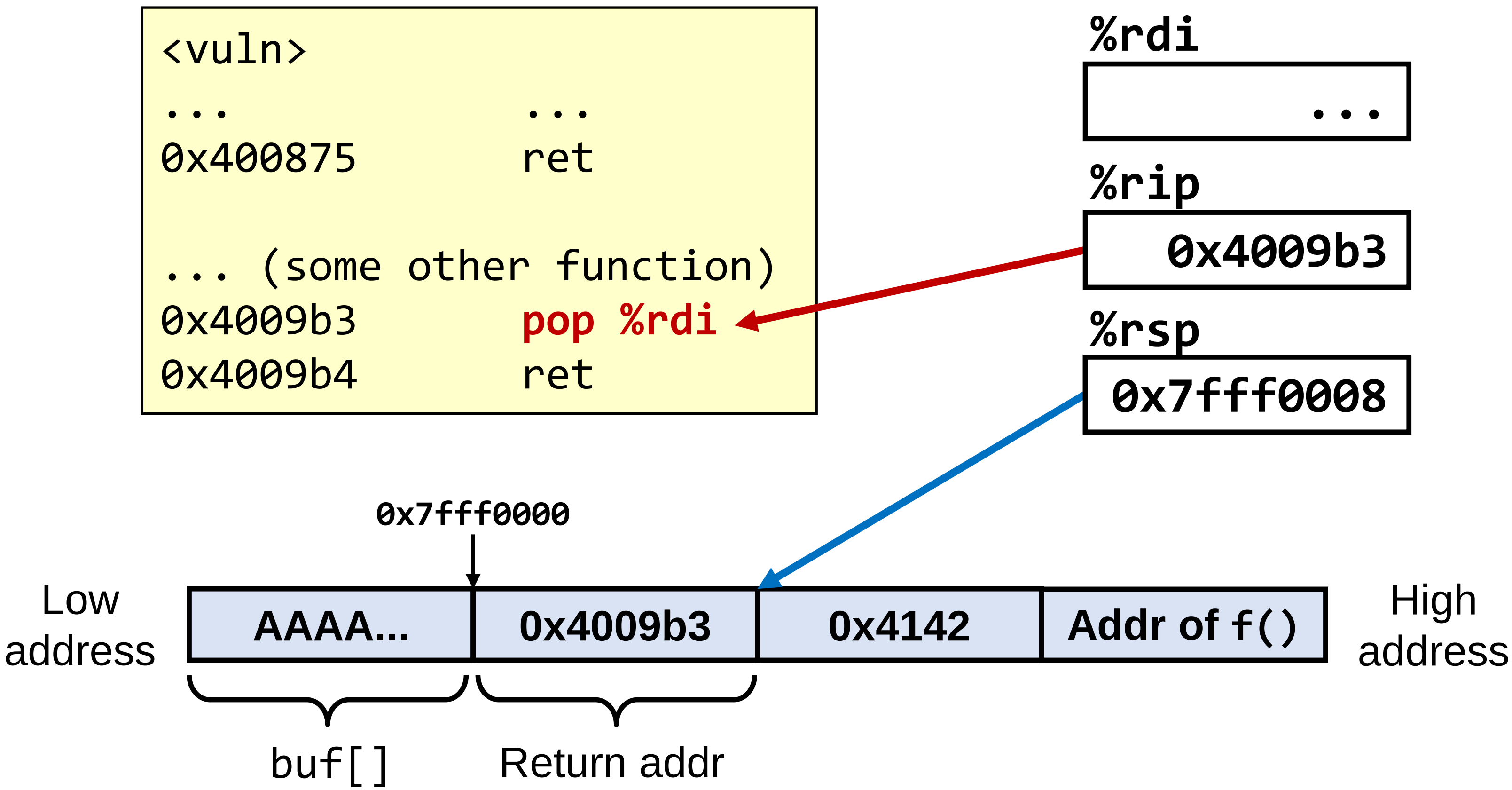
What can we do with gadgets?

- **First, `ret` of `vuln()` will pop `0x4009b3` into `%rip`**
  - At the same time, `%rsp` will be incremented by 8



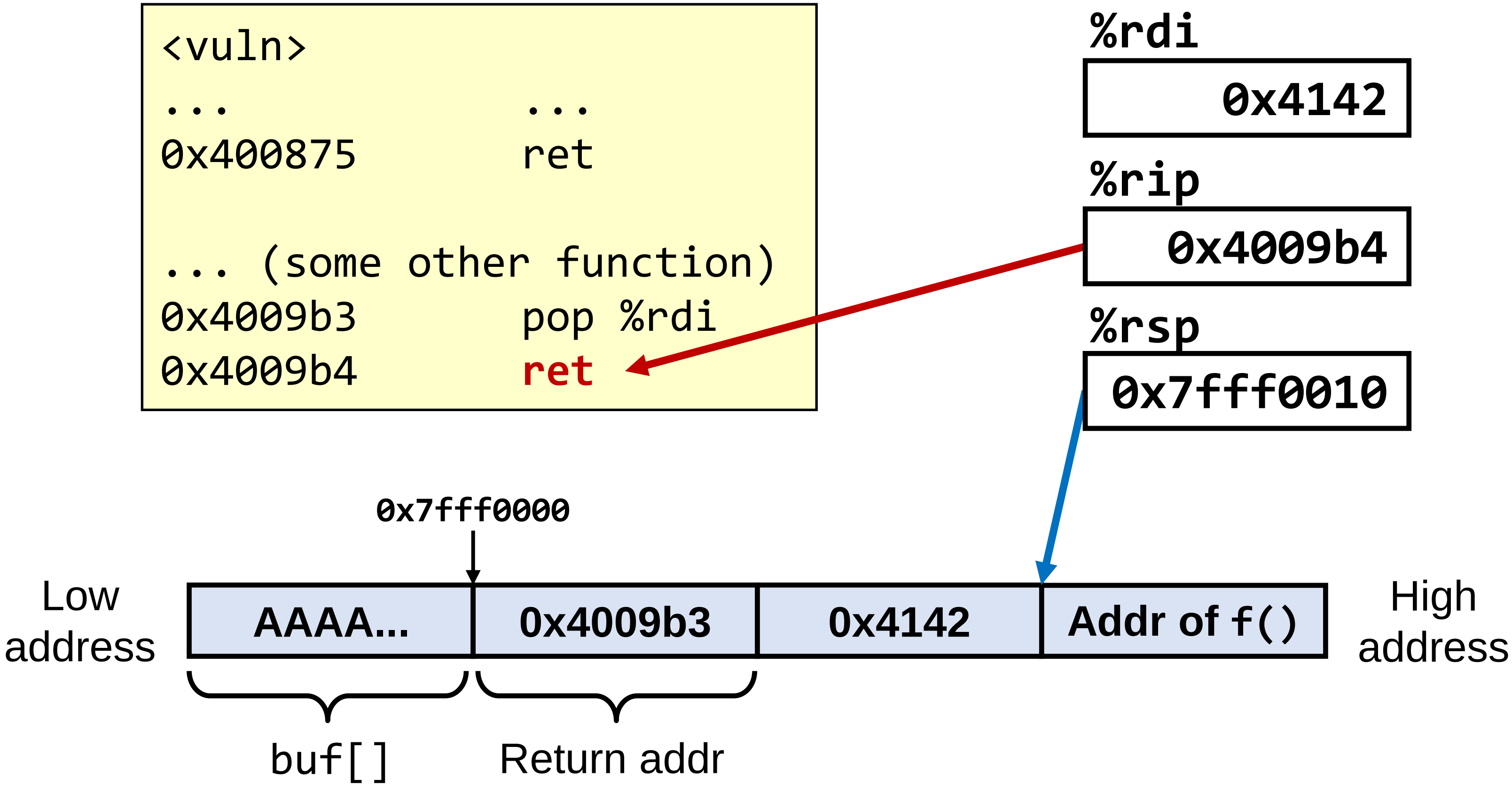
What can we do with gadgets?

- **Then, `%rip` points at `pop %rdi` instruction in gadget**
  - This instruction will pop **`0x4142`** into **`%rdi`** (= 1st argument)



What can we do with gadgets?

- **Now, `%rip` is pointing at `ret` instruction in gadget**
  - This will change **`%rip`** to point at the address of **`f()`**



We have just executed `f(0x4142)!`

ROP gadgets let us execute a chosen function with arbitrary argument

ROP Chain: Multiple Arguments

- **By chaining multiple ROP gadgets, it is also possible to call a function with multiple arguments**
- **Let's overwrite saved return address and use 2 gadgets**
  - **Caution:** Each block (even the small one) still represents 8-byte

```
... (Gadget 1)
0x4009b3        pop %rdi
0x4009b4        ret
```

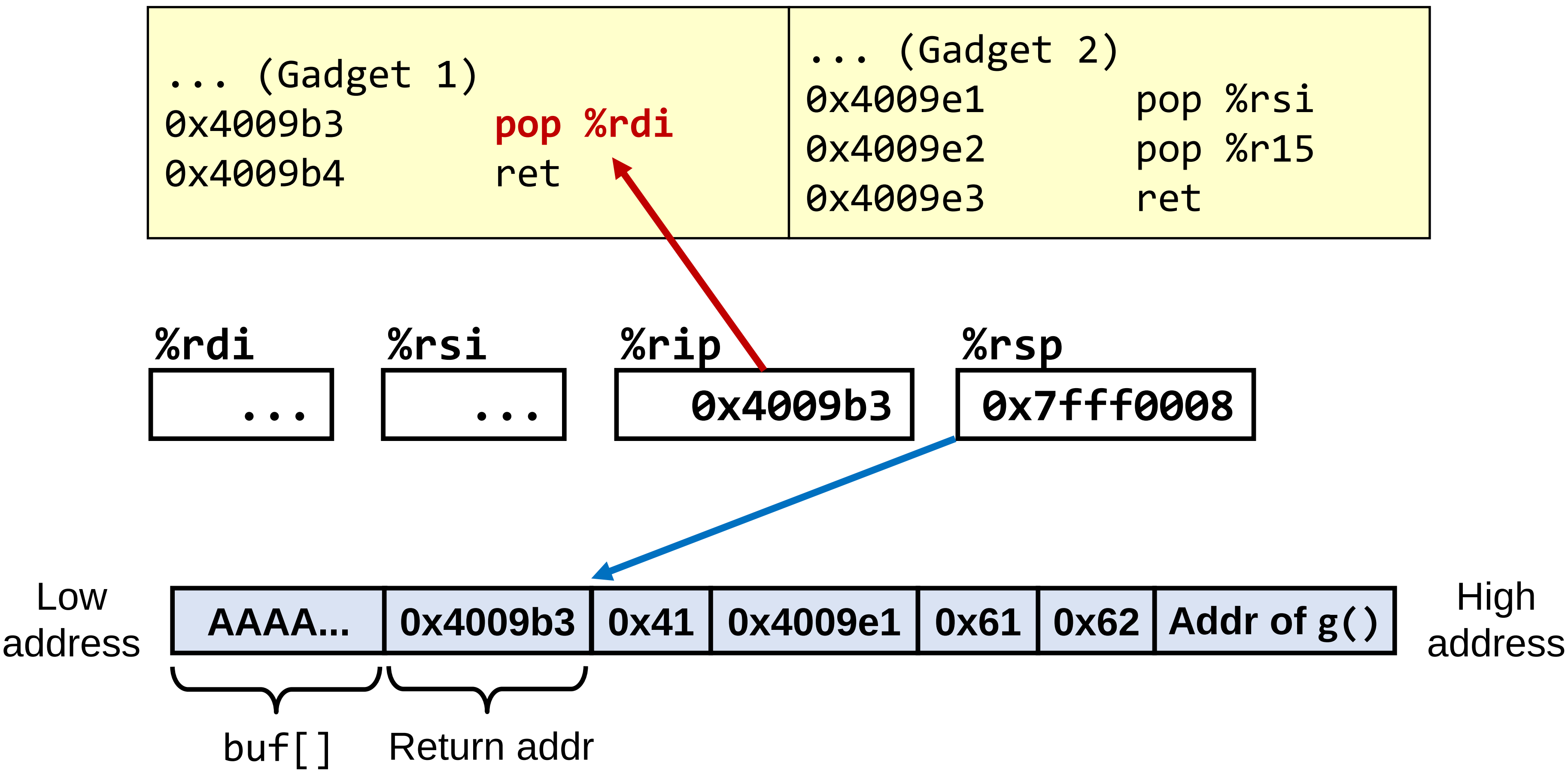
```
... (Gadget 2)
0x4009e1        pop %rsi
0x4009e2        pop %r15
0x4009e3        ret
```

Quiz: What will be passed to function `g()`?

Low address	AAAA...	0x4009b3	0x41	0x4009e1	0x61	0x62	Addr of `g()`	High address
	`buf[]`	Return addr						

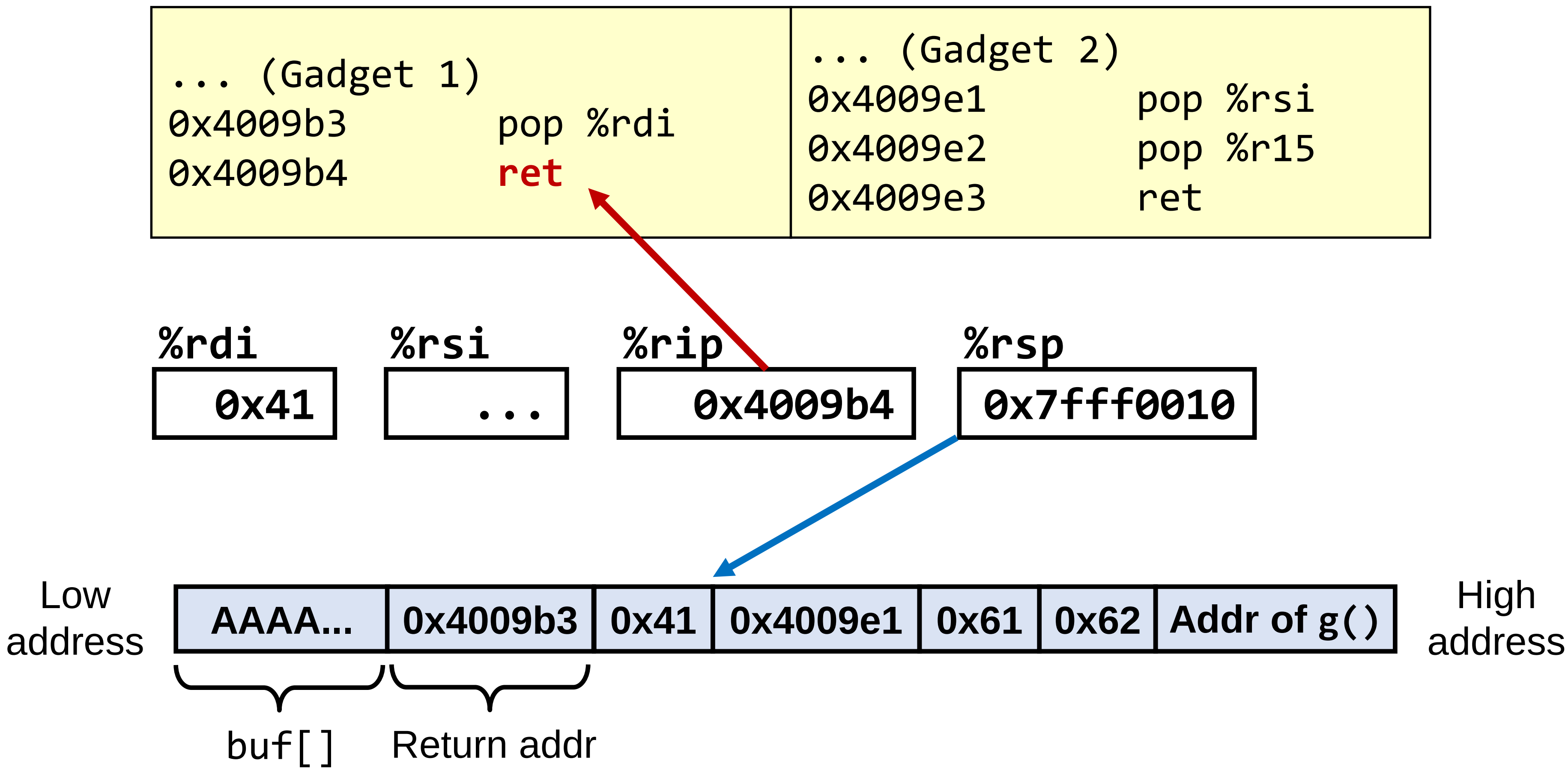
ROP Chain: Multiple Arguments

- **The following pages explain this process step-by-step**



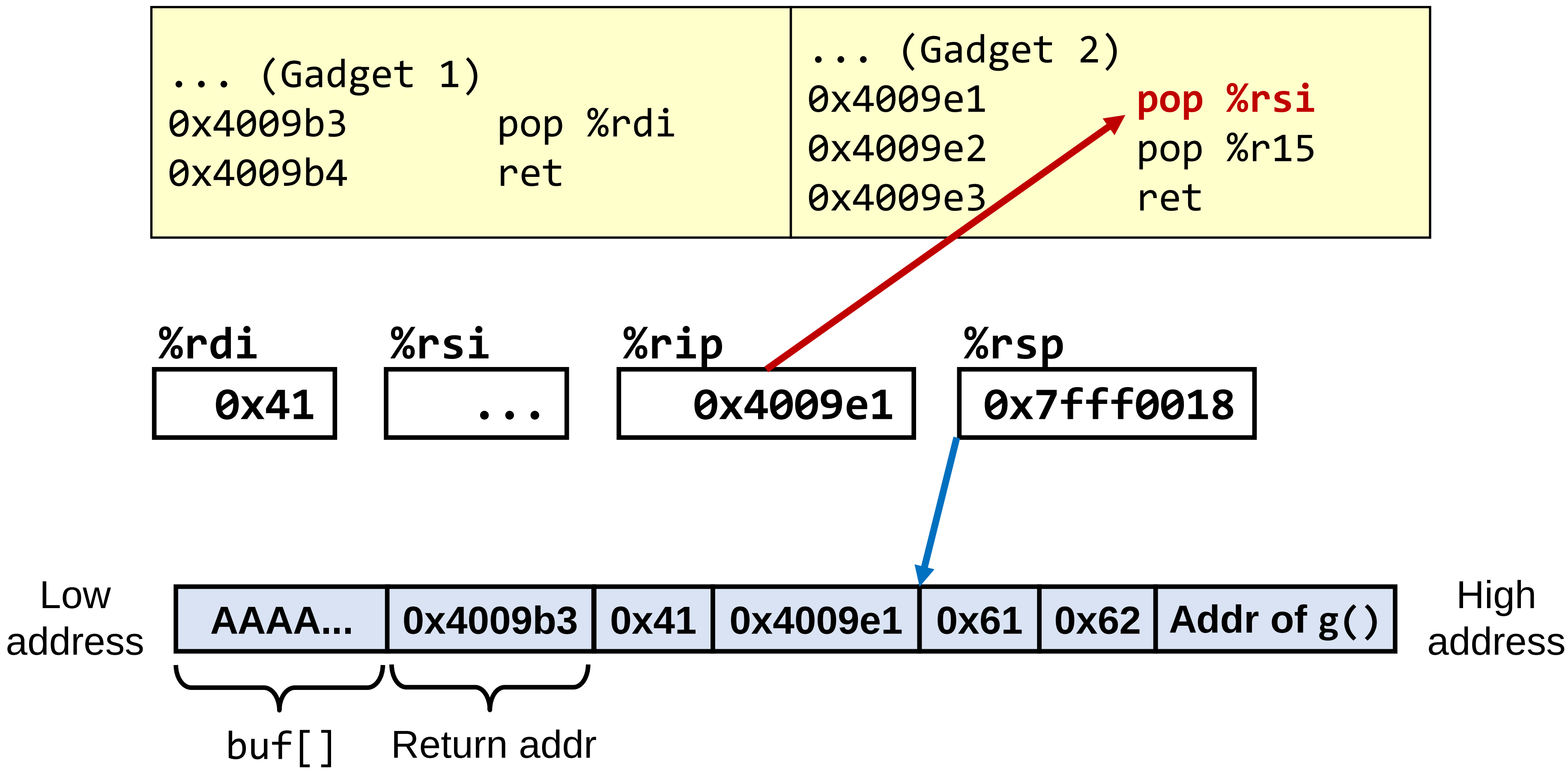
ROP Chain: Multiple Arguments

- The following pages explain this process step-by-step



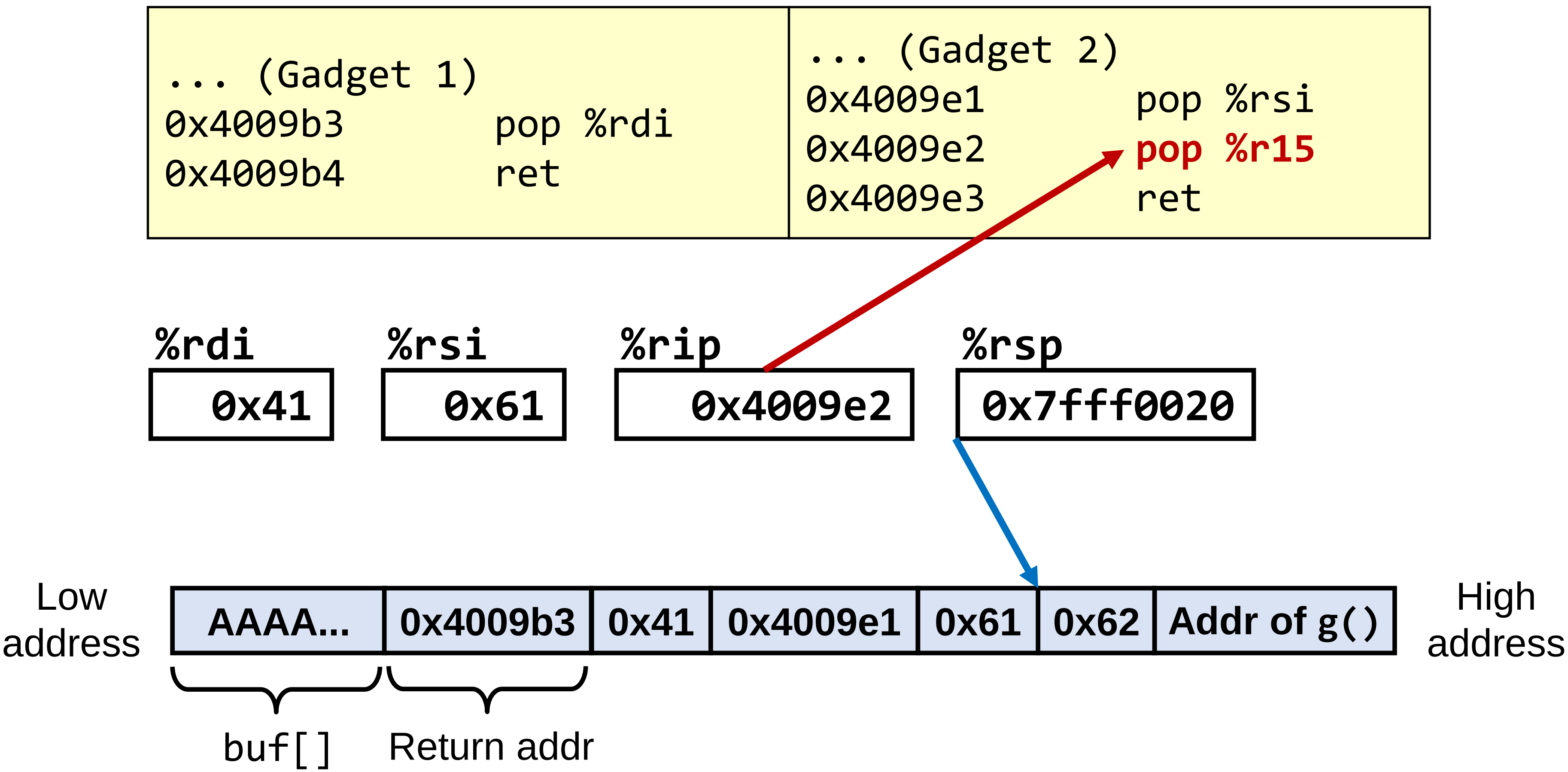
ROP Chain: Multiple Arguments

- **The following pages explain this process step-by-step**



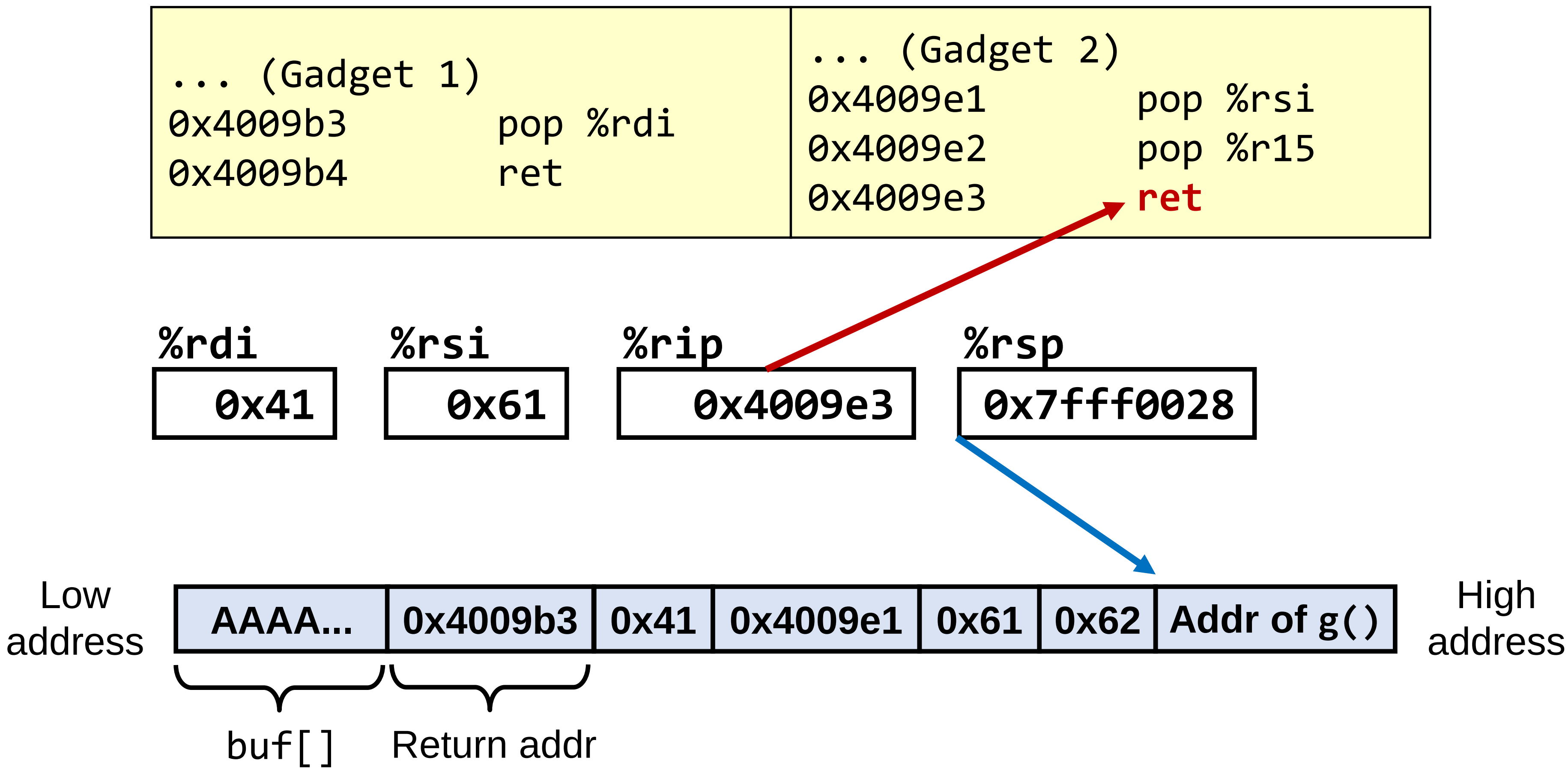
ROP Chain: Multiple Arguments

- **The following pages explain this process step-by-step**



ROP Chain: Multiple Arguments

- **The following pages explain this process step-by-step**



ROP Chain: Multiple Functions

- **Moreover, we can call a series of functions as well**
- **Simply put the next gadget right after the first function**
  - In the example below, we place the **second `0x4009b3`** immediately after the **address of `f()`**

```
...
0x4009b3        pop %rdi
0x4009b4        ret
```

Quiz: Which functions will called? What are their arguments?

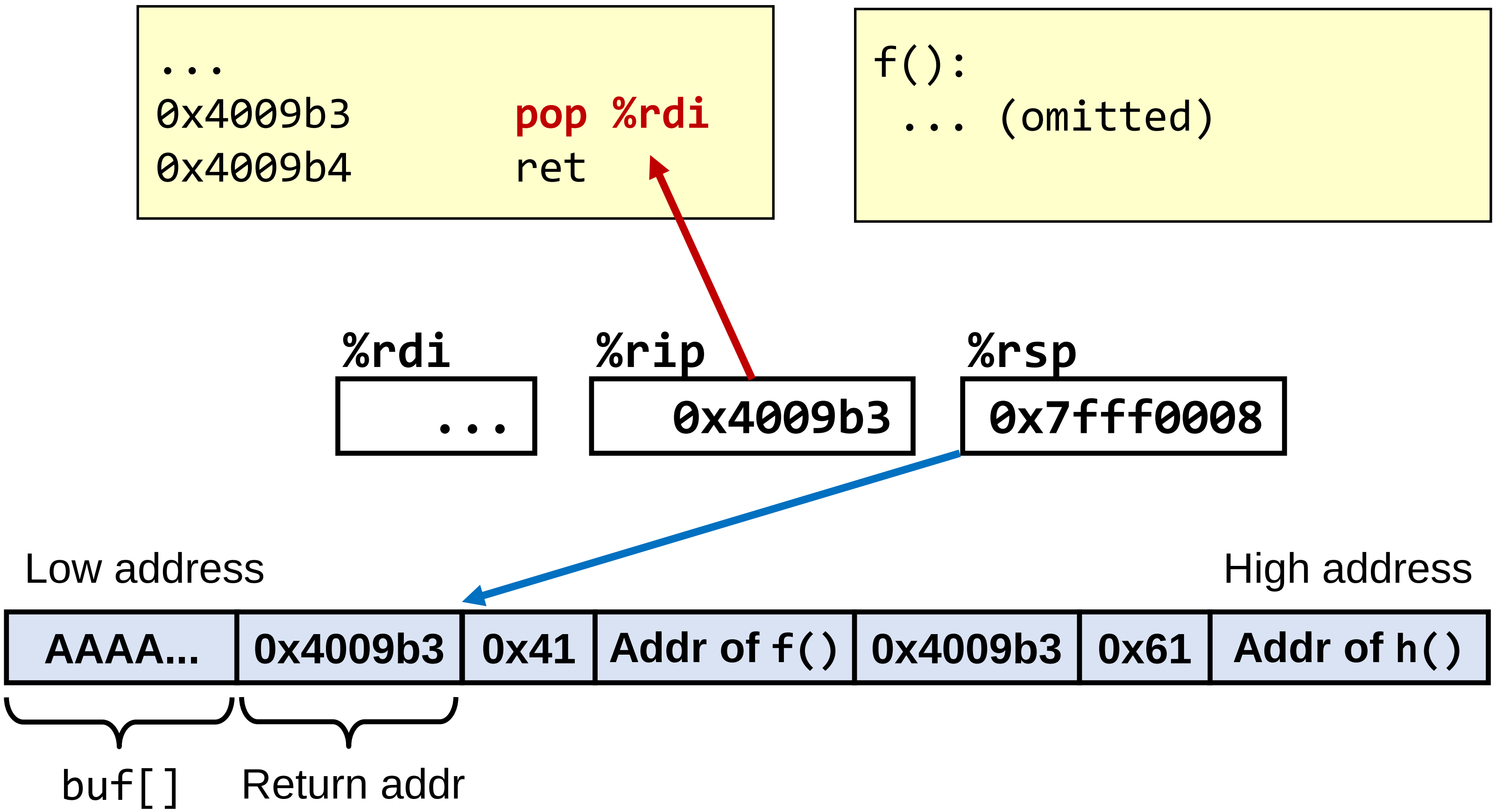
Low address   High address

AAAA...	**0x4009b3**	**0x41**	**Addr of `f()`**	**0x4009b3**	**0x61**	**Addr of `h()`**

`buf[]`   Return addr

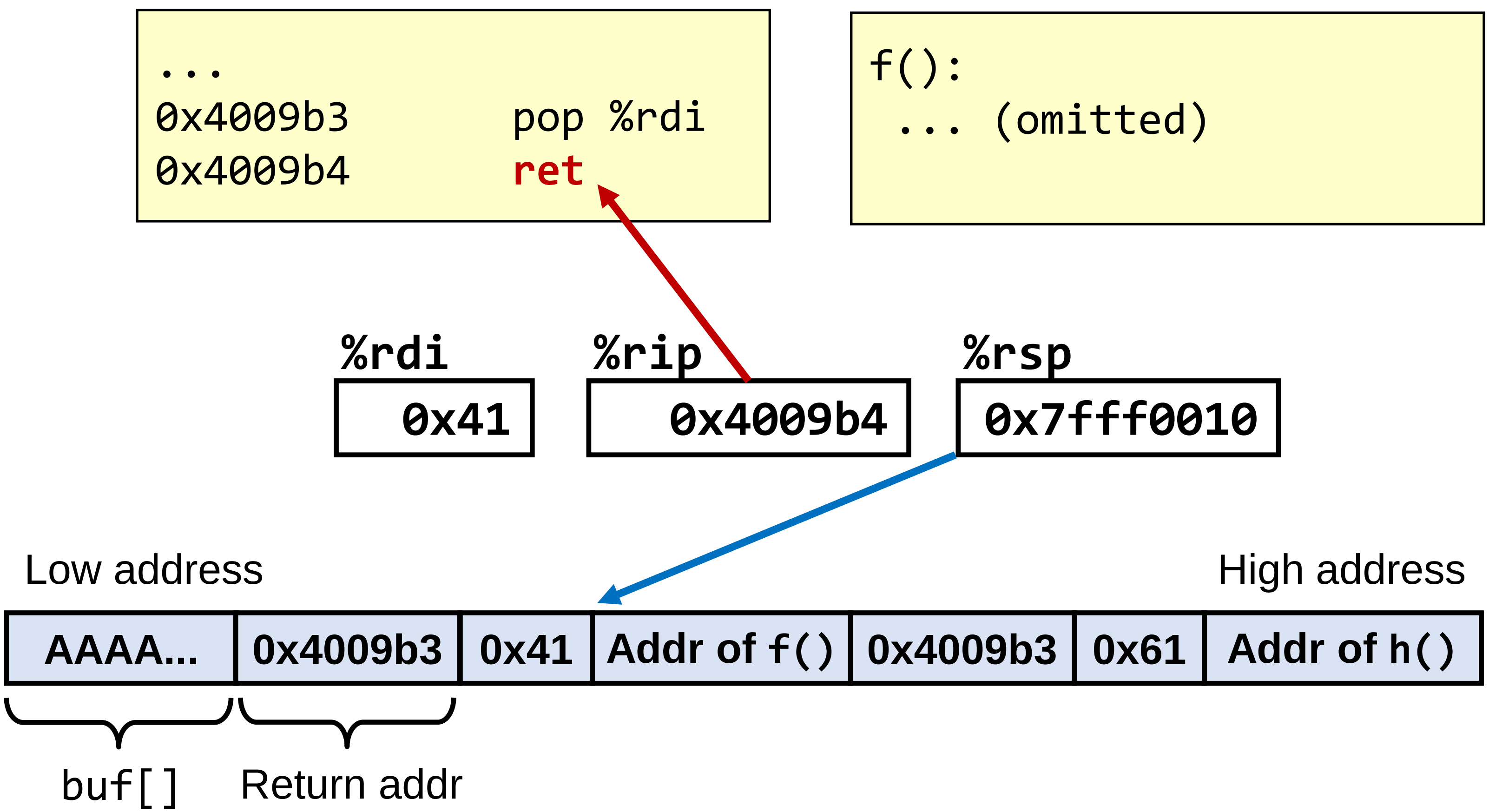
ROP Chain: Multiple Functions

- The following pages explain this process step-by-step



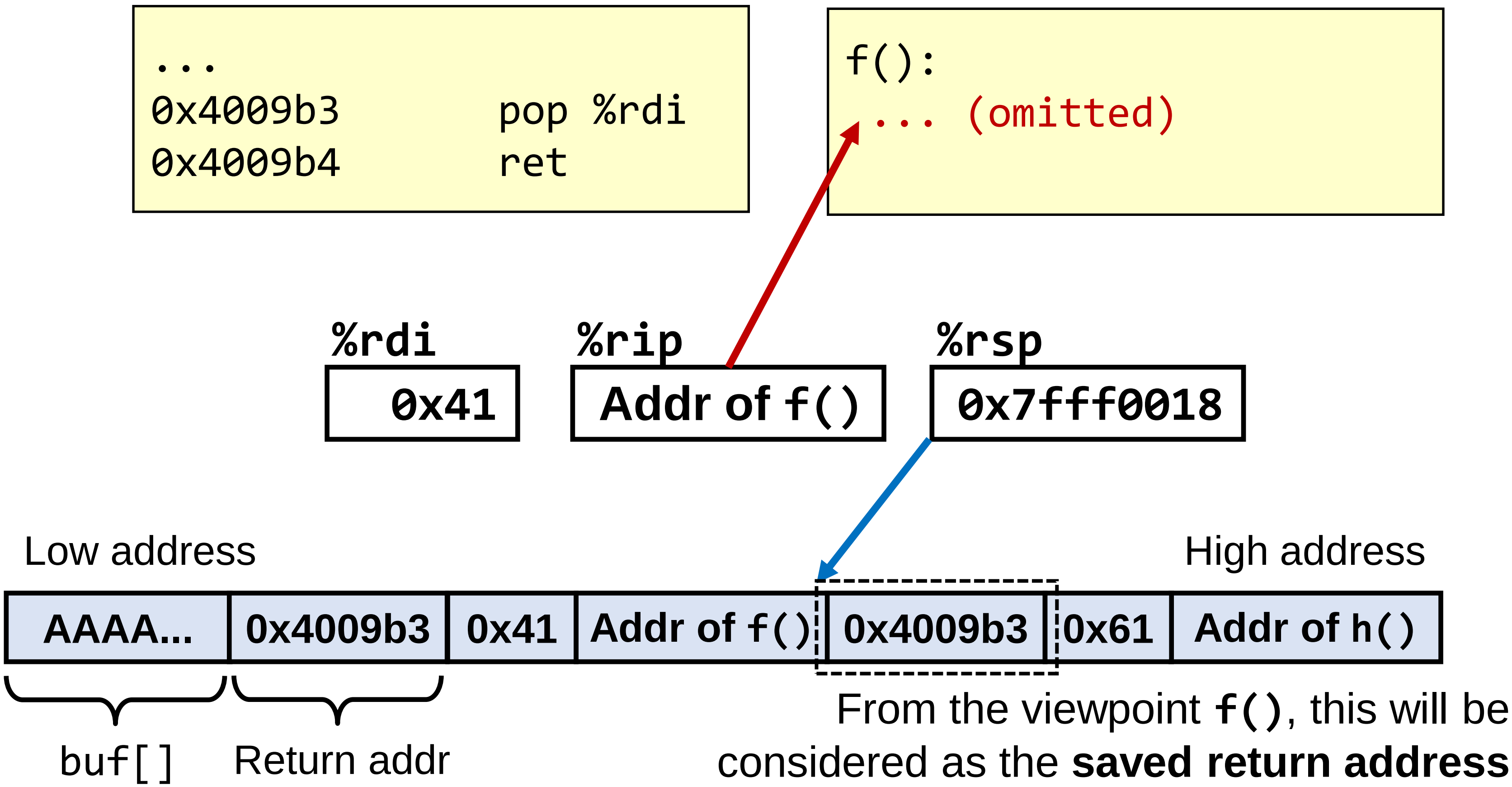
ROP Chain: Multiple Functions

- The following pages explain this process step-by-step



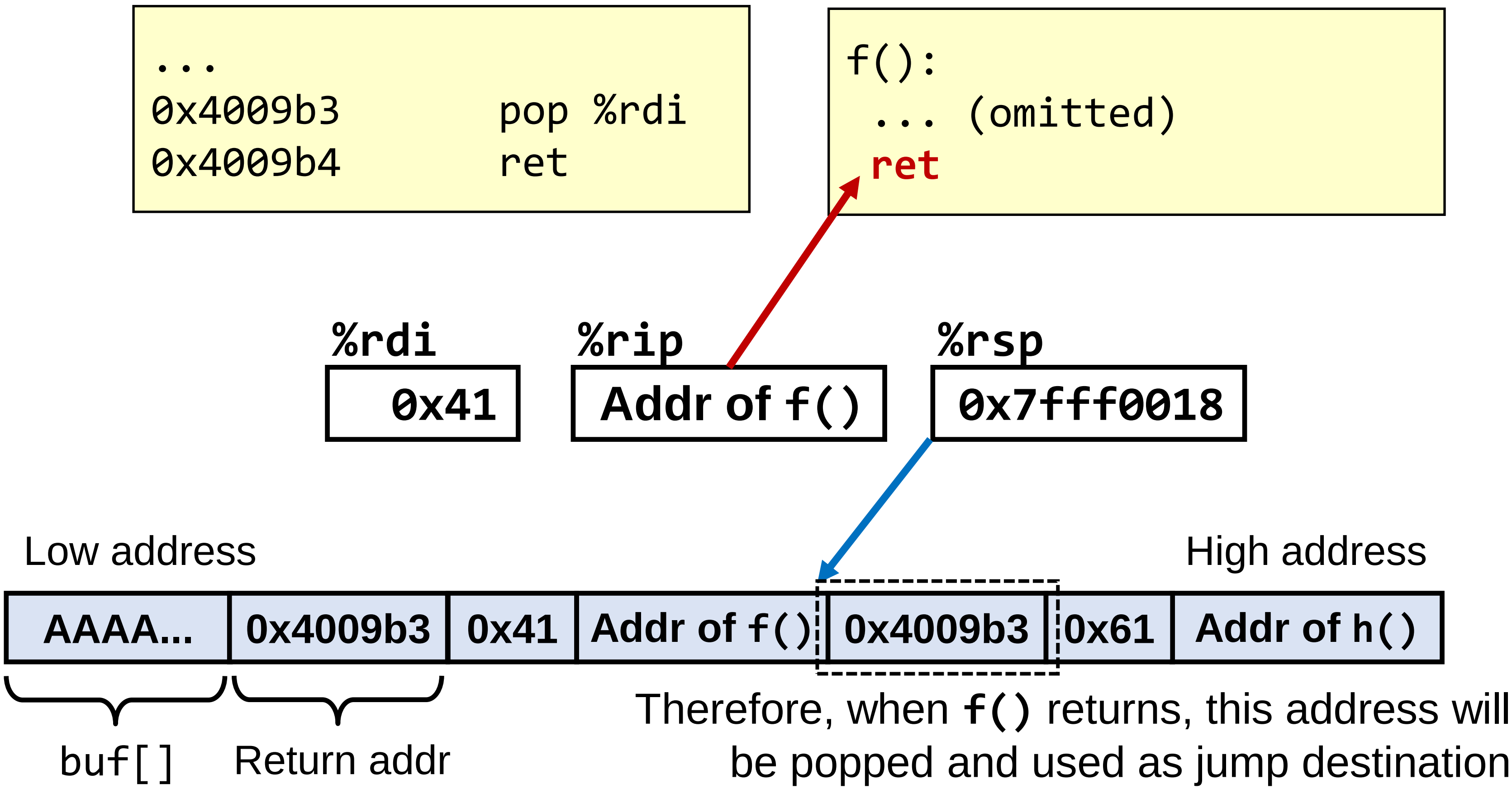
ROP Chain: Multiple Functions

- **The following pages explain this process step-by-step**



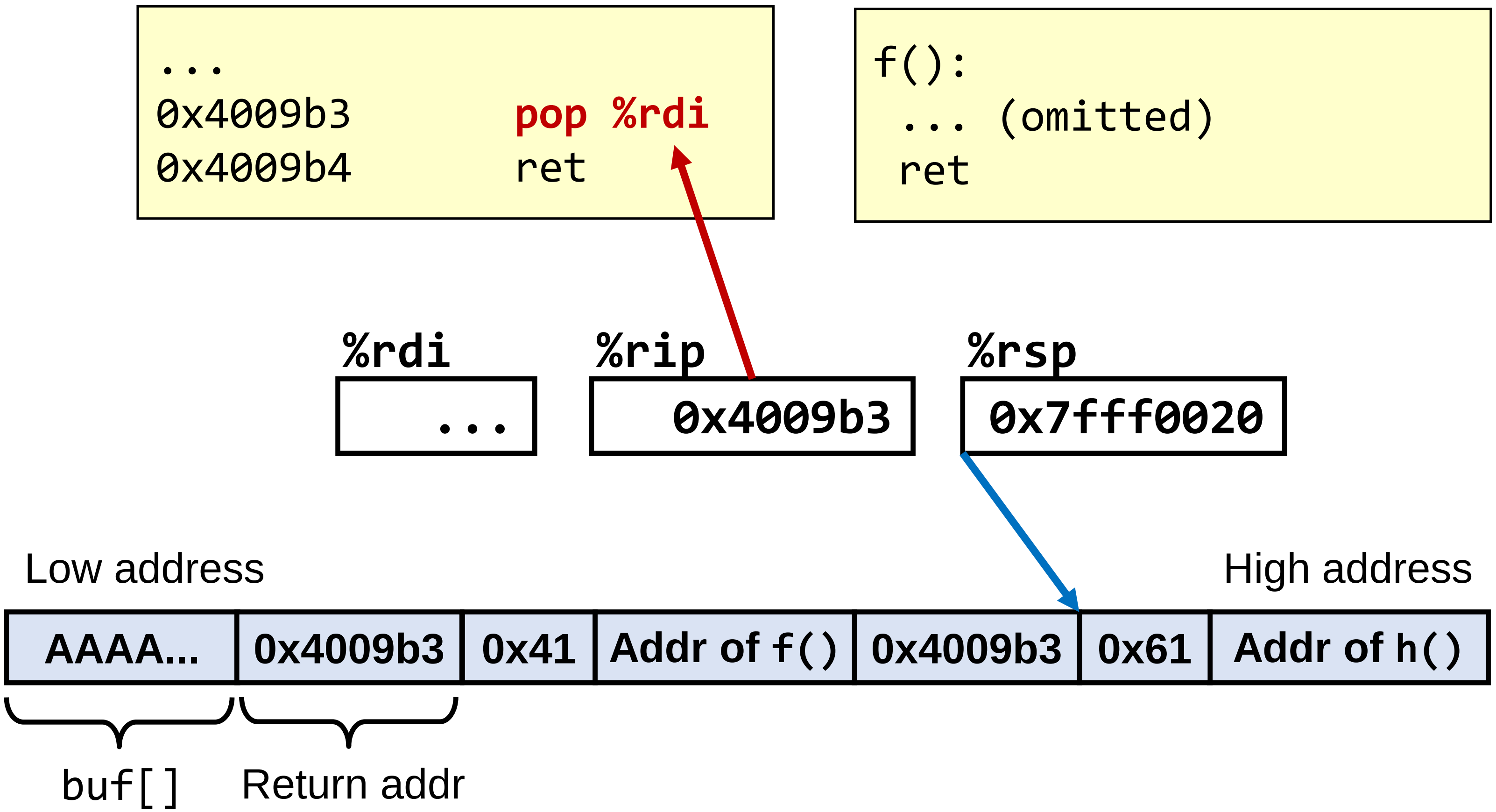
ROP Chain: Multiple Functions

- **The following pages explain this process step-by-step**



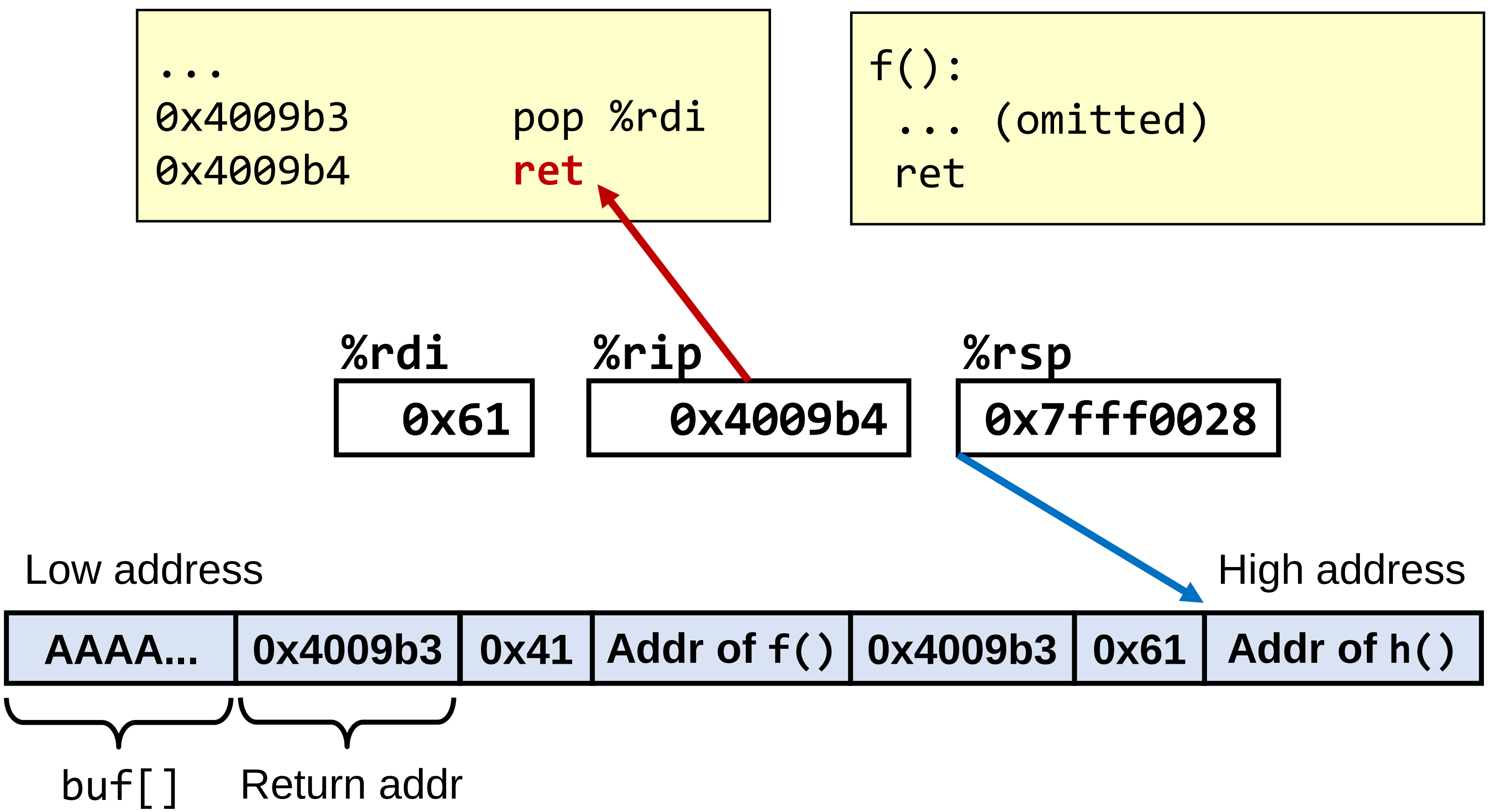
ROP Chain: Multiple Functions

- The following pages explain this process step-by-step



ROP Chain: Multiple Functions

- The following pages explain this process step-by-step



Various Types of ROP Gadgets

- **So far, we have discussed `"pop; … ret;"` style gadgets**
- **But there can be other types of ROP gadgets, too**
  - **Ex) `"add $1000, %rcx; ret"`**
  - **Ex) `"xchg %rbx, %rdx; ret"` # Exchange two registers**
  - **Ex) `"mov %rax, (%rbx); ret"` # Write to memory**
- **By chaining such gadgets, hacker can execute various logics and operations**
  - This explains why it is called **return-oriented programming**

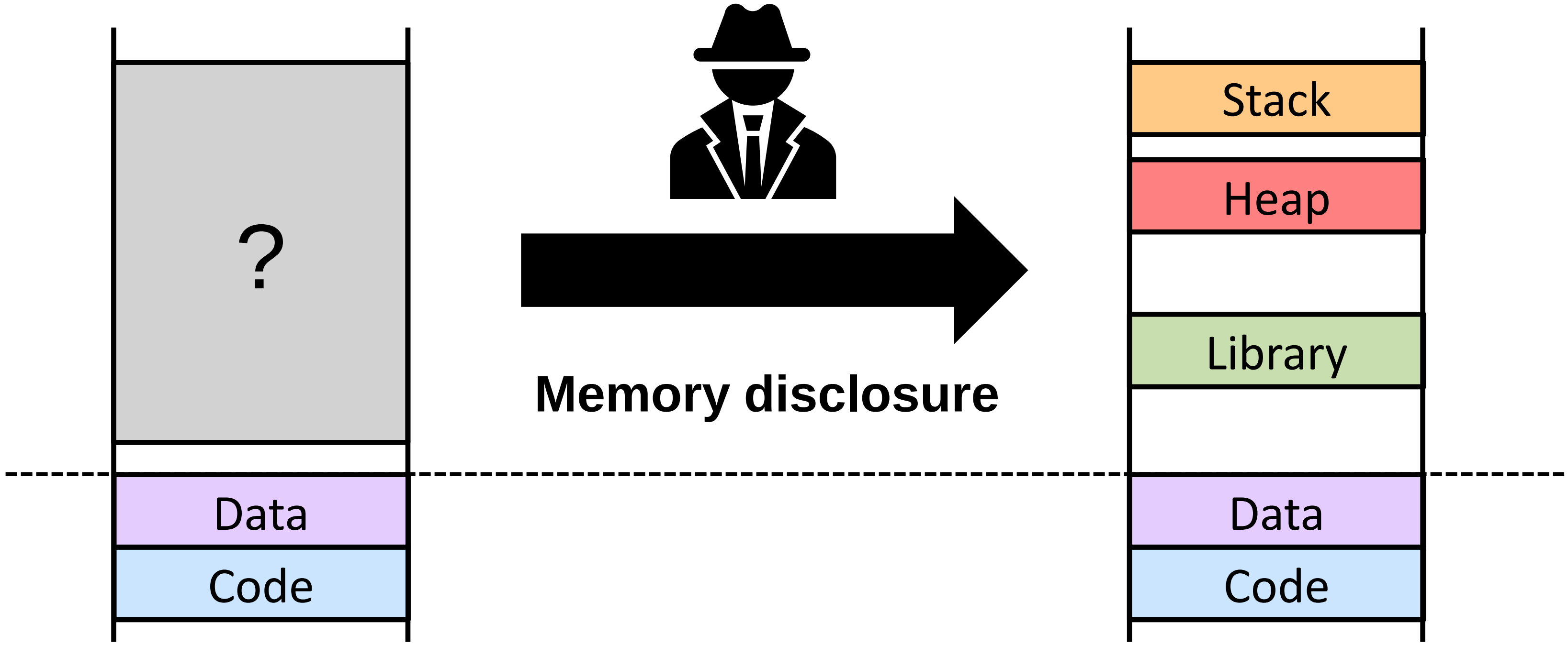
Using *ROP*, we have addressed *Challenge #1* (controlling arguments)

…

but what about *Challenge #2* (ASLR)?

Memory Disclosure Revisited

- **Your program will contain many pointer variables that store addresses values**
  - By disclosing (printing out) those values, hackers can get some clues about the memory layout



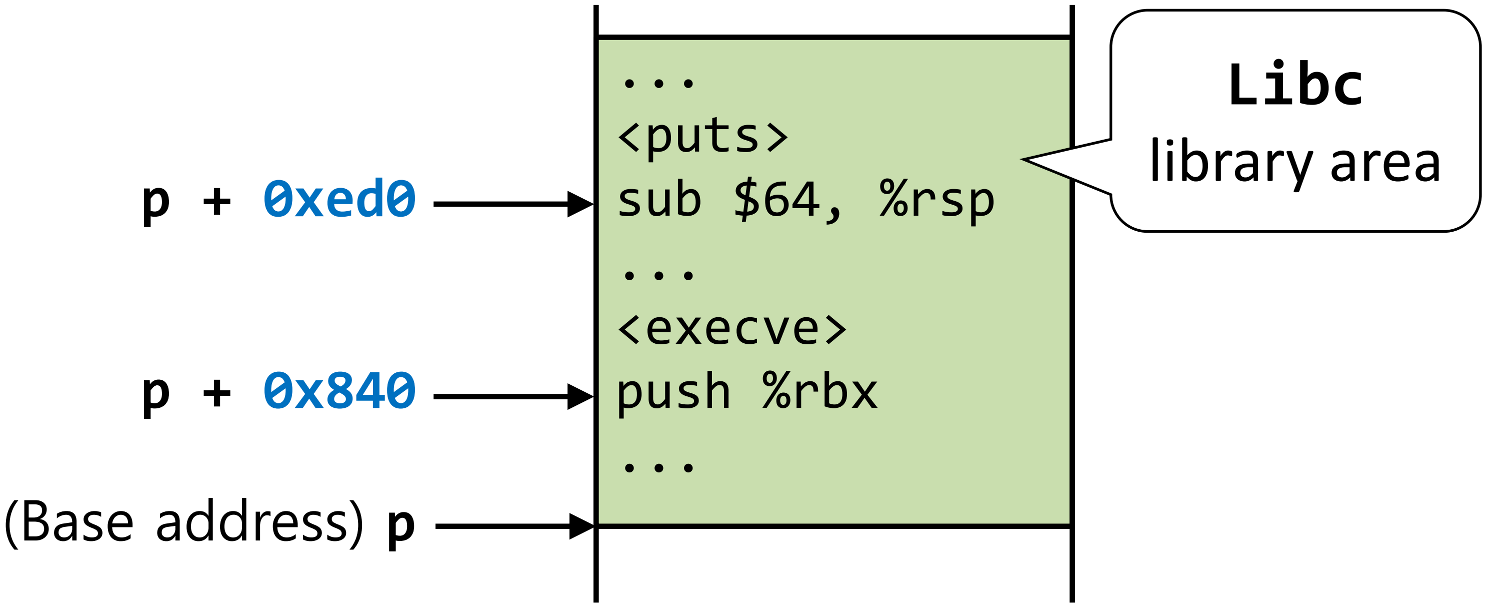
Disclosing Library Address

- **Let's consider the following code for example**
  - **`dlopen()`** + **`dlsym()`** are used to get the address of **`puts()`**
    - Don't have to know the details of these functions
  - The obtained address will be stored in pointer **`puts_fptr`**
  - Of course, this value will be different per execution **(ASLR)**

```
int main(void) {
    int (*puts_fptr)(const char *);
    void * handle = dlopen("libc.so.6", RTLD_LAZY);
    puts_fptr = dlsym(handle, "puts");
    puts_fptr("Hello world");
    return 0;
}
```

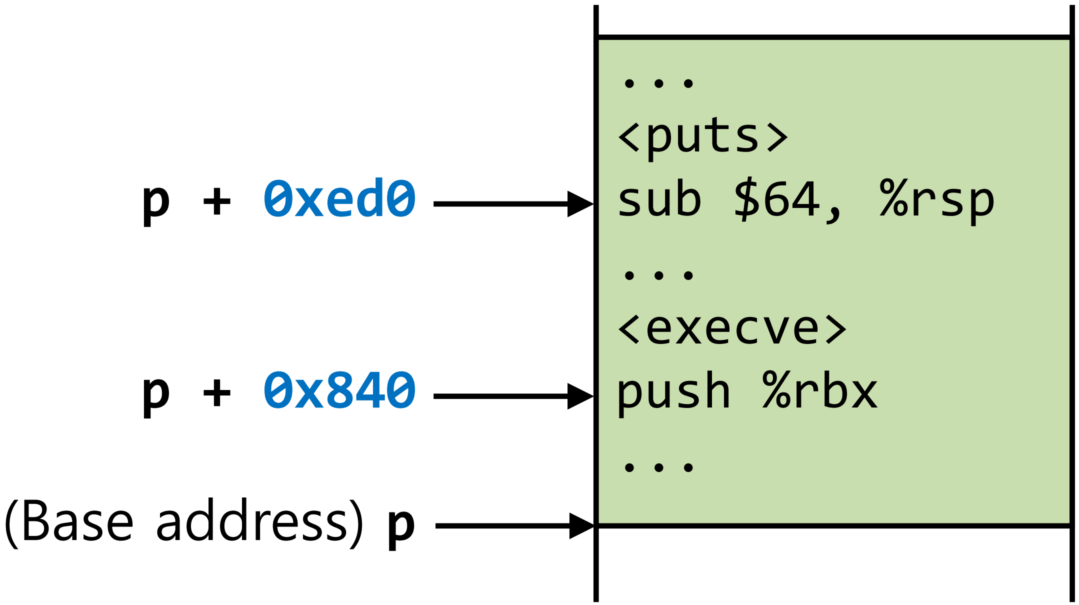
Disclosing Library Address

- **Now, assume that hacker can disclose (print out) the `puts_fptr` variable in the previous example**
  - For example, by using some buffer overflow (over-read)
- **Then, the hacker figure out the address of library code**
  - Why? The **offset** of a function within the library is **fixed**



Disclosing Library Address

- **ASLR only randomizes the base address (the starting address) of the library in memory**
  - Each function has unique and **fixed** offset (like `0xed0` for **`puts`**)
- **Thus, by disclosing `fptr_puts`, hacker can also learn the base address (p) and the address of `execve()`**



Ex) If leaked **`fptr_puts`** value is **`0x7ffd3ed0`**:

Base address **p** = ?

Address of `execve()` = ?

Does it really work?

- **You may wonder if such scenario works in practice**
- **For example, let's consider a simple program below**
  - The program has **no function pointer** to be leaked
  - Also, the BOF only allows memory corruption **(no disclosure)**
- **Is this program still exploitable? (assuming ASLR)**
  - **Yes, we can disclose the memory and bypass ASLR**

```
int main(void) {
  char buf[8];
  write(1, "Hello", 5);
  read(0, buf, 160);
  return 0;
}
```

Background: Library Function Call

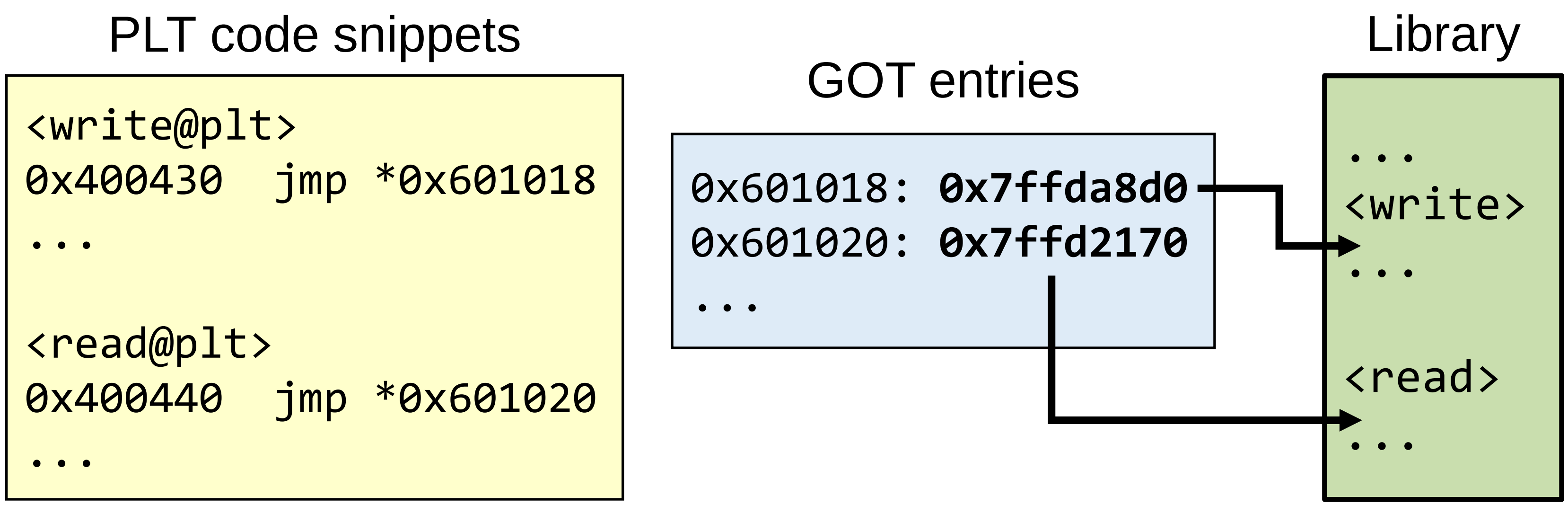
- **To understand how it is possible, you should know what really happens during the library function call**
- **When a program calls a library function like `write()`, its `%rip` does not directly transfer to the library**
  - Instead, it first moves to a small code snippet called **PLT**
  - This code snippet uses a function pointer in a table called **GOT**

```
int main(void) {
   ...
   write(1, "Hello", 5);
   ...
}
```

```
<main>
...
0x400579:  call 0x400430 <write@plt>
...

<write@plt>
0x400430  jmp *0x601018  # GOT entry
...
```

Background: PLT and GOT

- **In other words, you can think that compiler and linker implicitly generate some function pointer table and fill it**
  - To enable your program to call a function in library
  - Each library function called by your program has its PLT+GOT
  - GOT is filled at runtime (cannot be determined during compile)



So we can bypass ASLR by disclosing this GOT entry (function pointer)

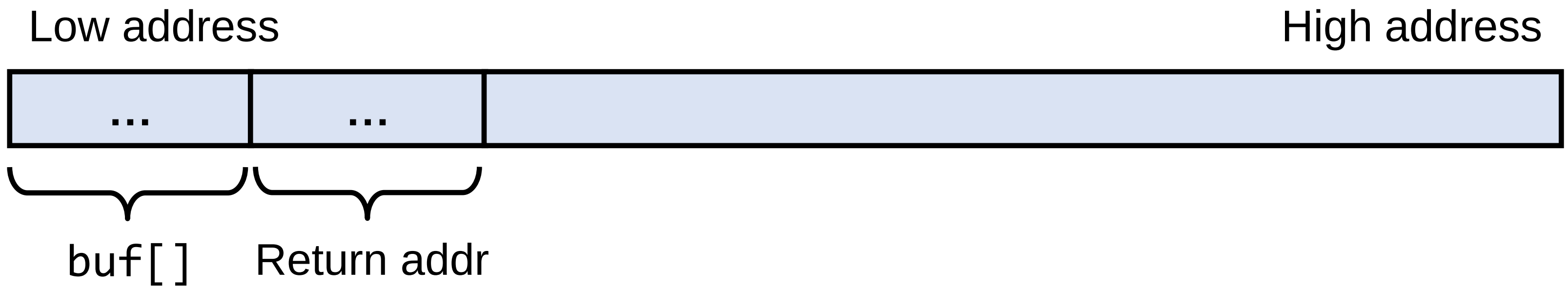
How? By using ROP again

ROP for Memory Disclosure

- **As before, we will overwrite the saved return address by using the buffer overflow in the example program**

```c
int main(void) {
  char buf[8];
  write(1, "Hello", 5);
  read(0, buf, 160); // BOF
  return 0;
}
```

(Before the BOF. Assume each block is 8-byte)



ROP for Memory Disclosure

- **By using ROP, we can call `write(1, &GOT, ...)`**
  - And chain more gadgets to call more functions and **get a shell**

```
<main>
...
0x400579:  call 0x400430 <write@plt>
...

<write@plt>
0x400430  jmp *0x601018  # GOT entry
...
```

```
(Gadgets)
...
0x4009b3        pop %rdi
0x4009b4        ret
...
0x4009c5        pop %rsi
0x4009c6        ret
```

Low address | High address

AAAA...	**0x4009b3**	**0x1**	**0x4009c5**	**0x601018**	**...**	**0x400430**	**...**

`buf[]` Return addr

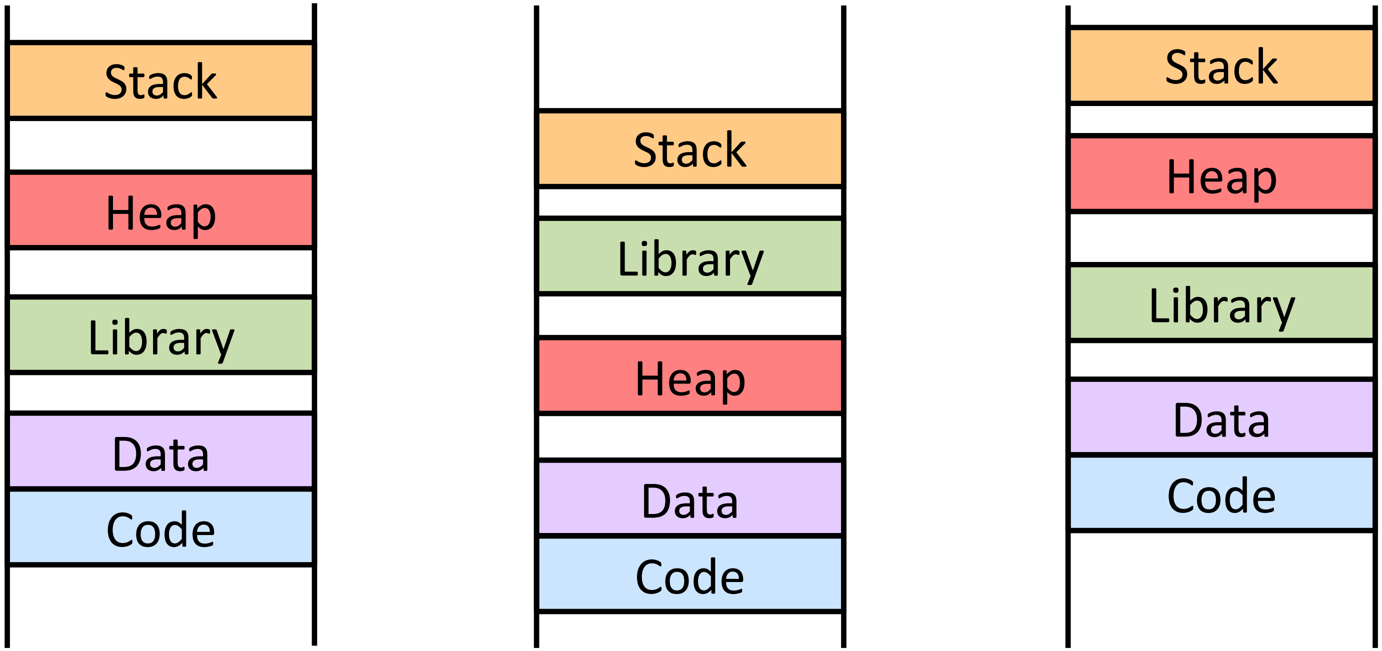
More gadgets may come

Notes on Memory Disclosure

- **Is library function offset always predictable?**
  - Depending on your Linux version, `Libc` library version will vary and the offset of each function will change, too
  - Even without an access to the `Libc` file, attackers can still infer the version of `Libc` (we will not discuss this deeply)
- **The idea of figuring out library address can be applied to other memory areas, too**
  - For example, the address of stack or heap can be also inferred

Position-independent Executable

- **Nowadays, ASLR is applied to Code & Data sections too**
  - This is called position-independent executable (PIE)
- **For this, compiler must generate complex assembly code**
  - So we will not cover it in this course



Lessons

- **Hackers are more persistent than you think**
  - They often come up with creative methods to bypass mitigation
- **So the impact of software vulnerabilities should not be underestimated or overlooked**
- **To precisely understand the outcome of a bug, it is important to know the internals of computer systems**
  - Ex) If you didn’t know the existence of PLT/GOT, it would be hard to imagine how the exploit is possible