Lecture 04 – Control Flow I

Stephen Checkoway

Outline for today

A bit about the C programming language

x86-64 control flow example

Start looking at control-flow hijacking attacks: buffer overflows on the stack

The C programming language

Most software used today is written in the C or C++ programming languages

[These are pretty different languages, especially modern C++ but they tend to get lumped together as C/C++ as they have similar security characteristics (i.e., poor)]

We’re going to focus on code written in C in this course although there will be a tiny bit about C++ a little later

C, bad news and good news

C doesn't define a whole bunch of things, leaving a lot of choice up to "the implementation"; i.e., the compiler and operating system

- Appendix J of the C standard (ISO/IEC 9899:201x is a freely-available draft standard) list a whole bunch of unspecified and undefined behavior

Some things not defined:

- stack frame layout (C doesn't even require a stack at all!)
- sizes of basic data types like int
- the behavior of programs containing "undefined-behavior" or "implementation-defined behavior"

Good news:

- We don't care about any of that
- The hardware a program runs on has well-defined behavior (mostly) and we care what the hardware will actually do, not what the programming language says or doesn't say

Some examples of undefined behavior

If x and y are ints then x + y isn't allowed to overflow (remember integer overflow from CSCI 210/241)

- If it does, then C allows anything to happen

If x is an int, then x << 32 (left shift of x by 32) isn't defined if the size of an int is less than or equal to 32 bits!

If p is a pointer to an array of length 10, then accessing p[15] is undefined behavior

The hardware defines it

Let’s look at those examples from the lens of the x86-64 hardware

x + y

- add eax, edx 32-bit 2’s complement arithmetic, regardless of overflow

x << 32; if the operand size is 64, then the shift count is masked to 6 bits, otherwise it is masked to 5 bits

- shl eax, 32 Surprisingly, this performs eax << (32 & 31) which is eax << 0
- shl rax, 32 rax is 64 bits so this does rax << (32 & 63) which is rax << 32

int x = p[15]; if p is a pointer to an array of 32-bit ints, then

- mov eax, DWORD PTR [ebx + 60] Well defined, but might crash if ebx + 60 doesn’t belong to a page of memory allocated to the process by the kernel

We don’t care what C says

We’re working with Linux on x86-64 which uses the Sys V x86-64 ABI so we get

- `sizeof(char) = 1` (This is always true for C)
- `sizeof(short) = 2`
- `sizeof(int) = 4`
- `sizeof(long) = 8`
- `sizeof(char *) = sizeof(int *) = sizeof(long *) = 8`

Data model: Sys V x86-64 ABI uses the LP64 data model

- **LP64 means long and pointers are 64 bits, int is 32 bits**
- ILP32 is common for 32-bit hardware and means int, long, and pointers are 32 bits

Strings in C

Recall CSCI 241

A string in C is an array of nonzero bytes followed by a 0 byte

```
char greeting[] = “hello!”;
```

is an 7-element array of characters that’s exactly the same as

```
char greeting[7] = { 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x21, 0x00 };
```

It’s not possible to pass arrays around in C the way it is in Rust; instead, we use a pointer to the first character of the array

```
char *s = &greeting[0];   // Or equivalently
char *s = greeting;
```

I**t’s not possible to have a 0 byte in a C string**; we’ll have to deal with this next class when we talk about shellcode

Command line arguments

Command-line arguments are passed to the main function in a C program by the kernel as follows (this is a bit of a lie, there's some code that runs before main that sets some of this up in concert with the kernel)

- The command line arguments are placed on the bottom of the stack
- An array of pointers to each of these strings is constructed on the stack
- This array ends with a `NULL` pointers (value 0)
- The count of command line arguments as well as a pointer to the array of pointers is passed to the `main` function
  `int main(int argc, char *argv[])`
- `main` can access the nth command line argument by accessing `argv[n]`

Example

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("argc = %d\n", argc);
    printf("argv = %p\n", argv);
    for (int i = 0; i < argc; ++i) {
        printf("argv[%d] = %p \"%s\"\n", i, argv[i], argv[i]);
    }
    printf("argv[%d] = 0x%lx\n", argc, (long)argv[argc]);
    return 0;
}
```

Notice how there is a 1 byte gap between each string, that's the 0 byte terminating the string

```
[mcnulty:~] steve$ gcc args.c
[mcnulty:~] steve$ ./a.out one two three four
argc = 5
argv = 0x7ffd55953cb8
argv[0] = 0x7ffd5595549d "./a.out"
argv[1] = 0x7ffd559554a5 "one"
argv[2] = 0x7ffd559554a9 "two"
argv[3] = 0x7ffd559554ad "three"
argv[4] = 0x7ffd559554b3 "four"
argv[5] = 0x0
```

Let’s step through a simple program

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

int foo(int a, char *p) {
    int b = strtol(p, NULL, 10);
    return a + b;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s num\n", argv[0]);
        return 1;
    }
    int x = foo(100, argv[1]);
    printf("%d\n", x);
    return 0;
}
```

Things to notice:

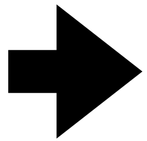
strtol(str, endptr, base) converts str to an integer in the given base

printf/fprintf take format strings containing conversion specifiers that say how to print the arguments

- %s prints a string
- %d prints a decimal integer
- %x prints a hexadecimal integer
- and many others

```c
int foo(int a, char *p) {
    int b = strtol(p, NULL, 10);
    return a + b;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s num\n", argv[0]);
        return 1;
    }
    int x = foo(100, argv[1]);
    printf("%d\n", x);
    return 0;
}
```

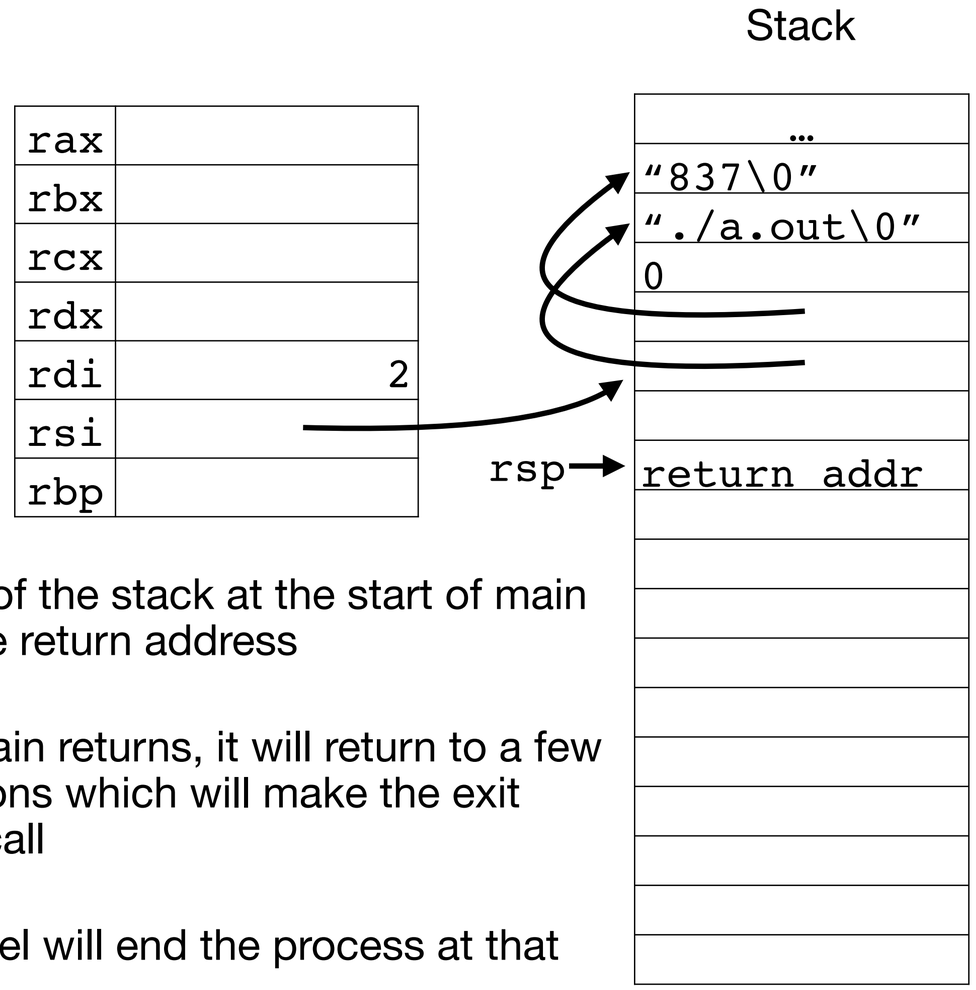


```asm
foo:
        push    rbx
        mov     ebx, edi
        mov     rdi, rsi
        mov     edx, 10
        mov     esi, 0
        call    strtol
        add     ebx, eax
        mov     eax, ebx
        pop     rbx
        ret
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
 rip→   sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

rip points to the next instruction

```
$ ./a.out 837
```



The top of the stack at the start of main holds the return address

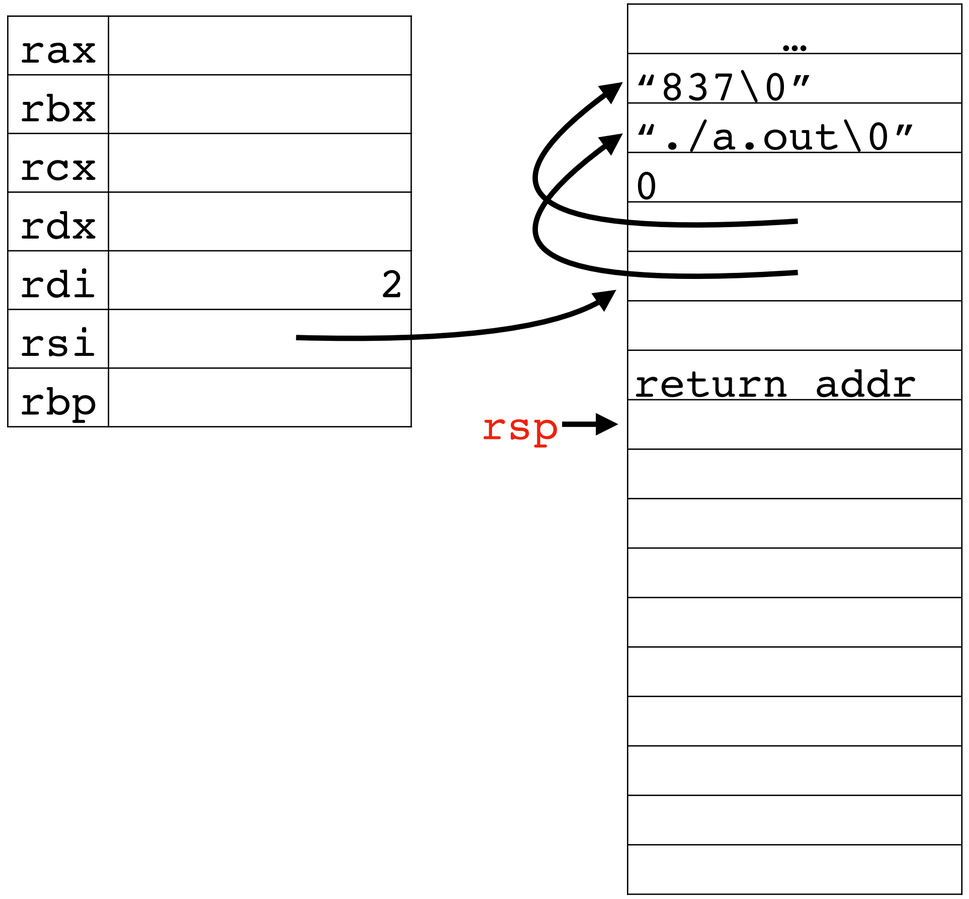
When main returns, it will return to a few instructions which will make the exit system call

The kernel will end the process at that point

```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
 rip→   cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
```

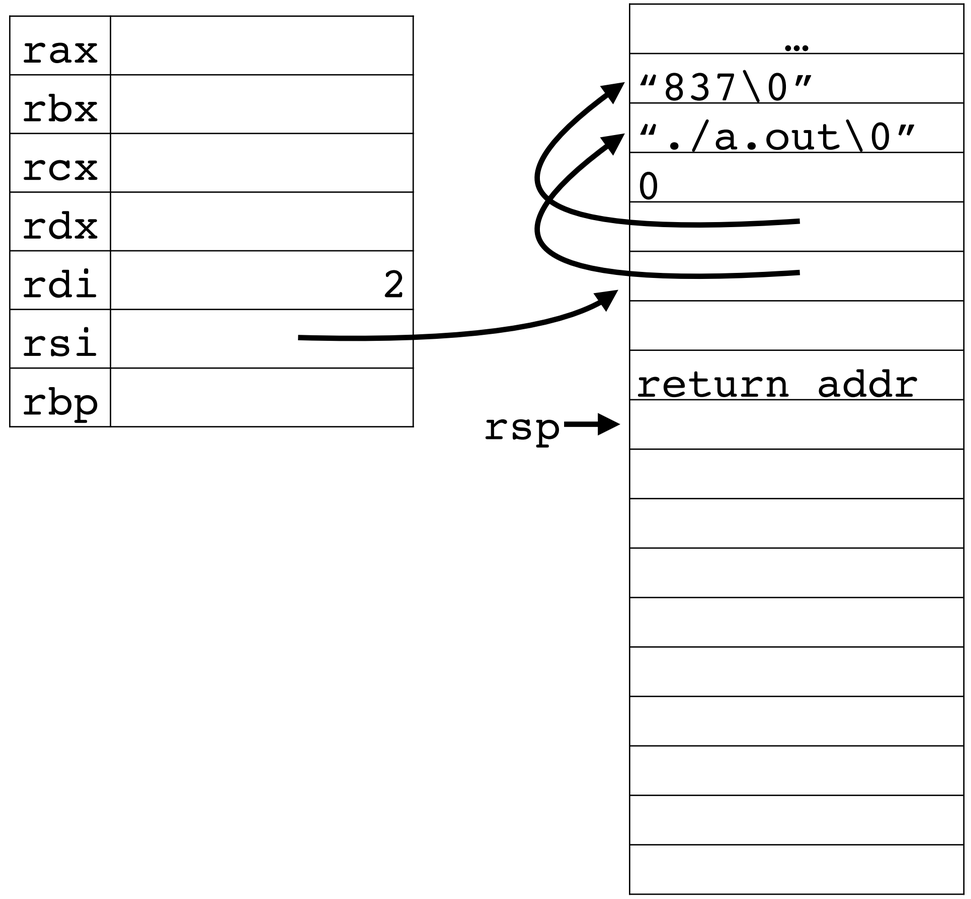
cmp will compare lower 32 bits of rdi to 2 and set rflags



```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
 rip→   je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

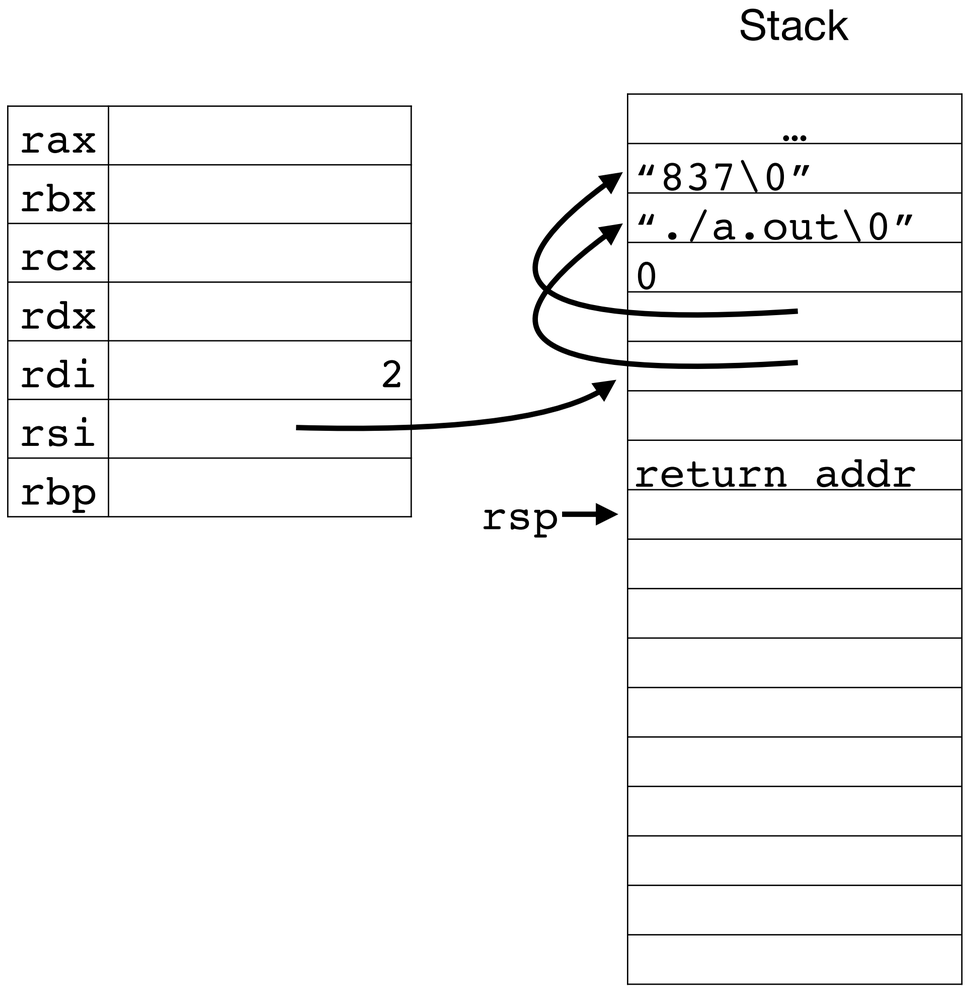
```
$ ./a.out 837
```

zero flag bit of `rflags` is set from the cmp
je/jz will jump to the label if the zero flag is set



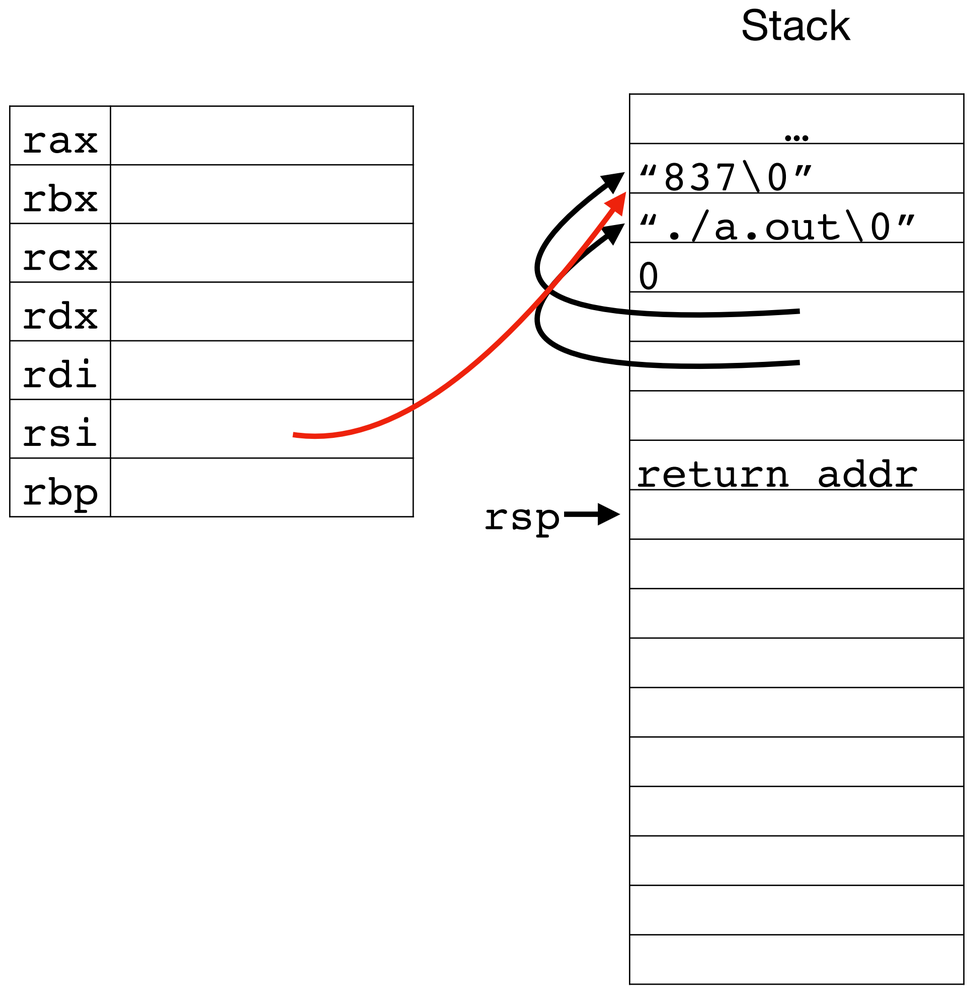
```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
 rip→   mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
```



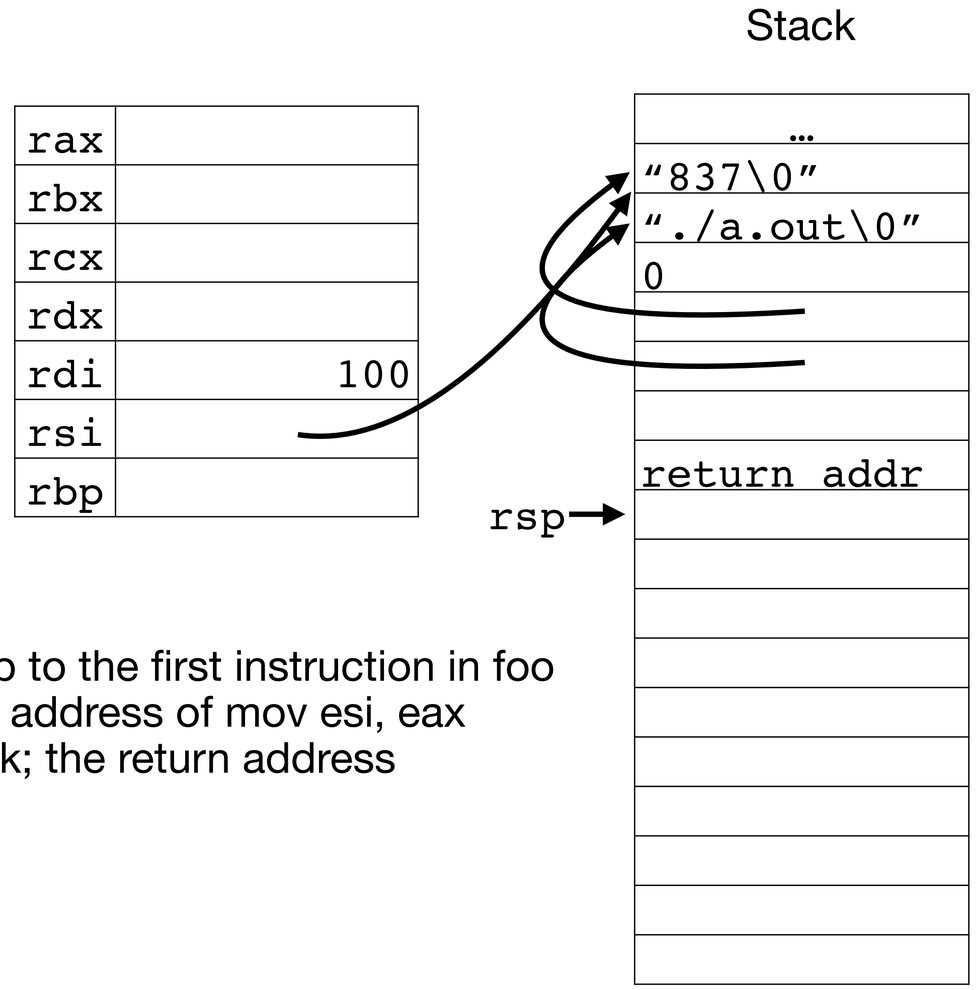
```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
  rip→  mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
```



```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
  rip→  call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

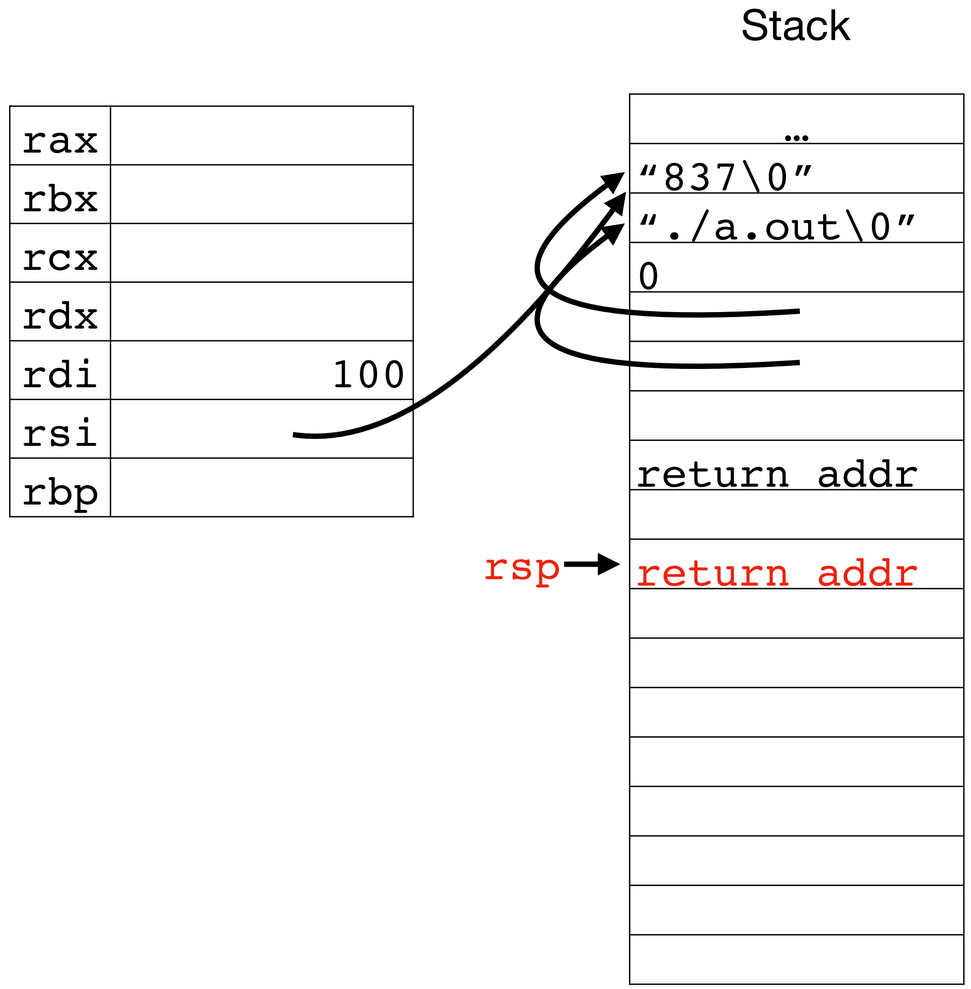
```
$ ./a.out 837
```



call will set rip to the first instruction in foo and push the address of mov esi, eax onto the stack; the return address

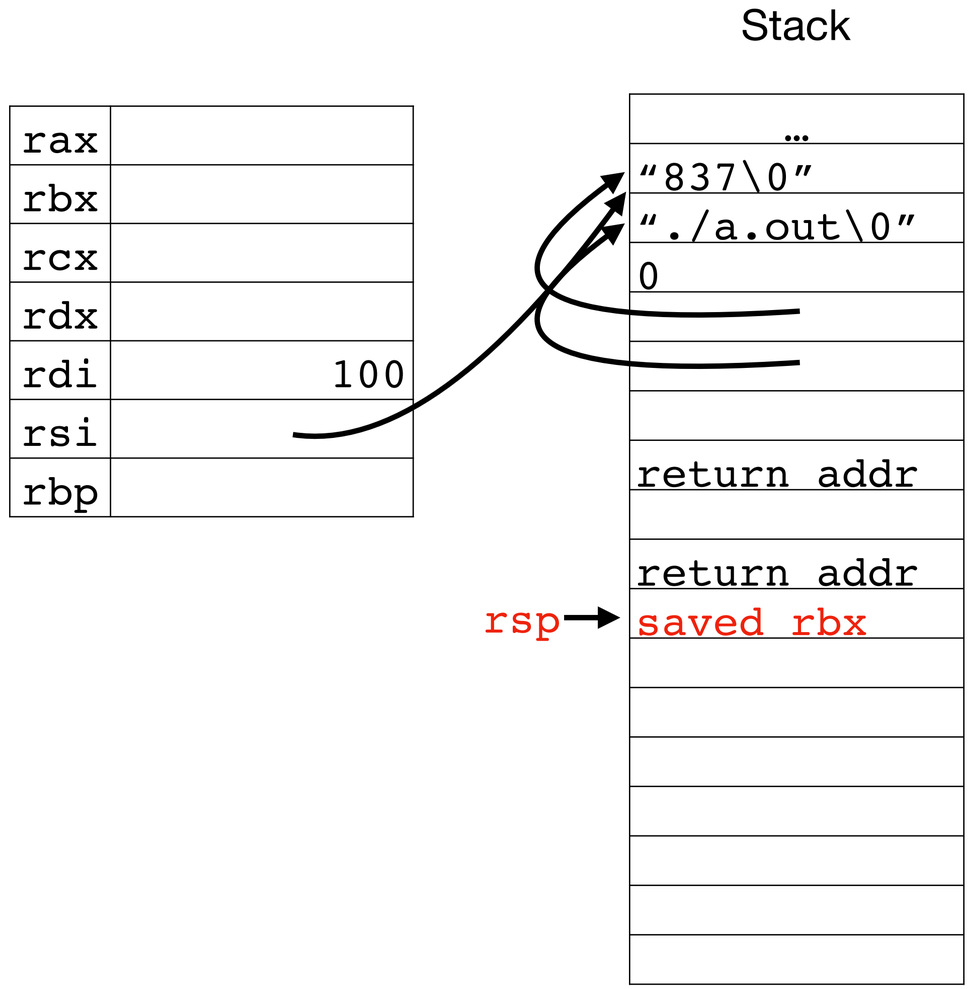
```
foo:
  rip→ push    rbx
       mov     ebx, edi
       mov     rdi, rsi
       mov     edx, 10
       mov     esi, 0
       call    strtol
       add     ebx, eax
       mov     eax, ebx
       pop     rbx
       ret
```

```
$ ./a.out 837
```



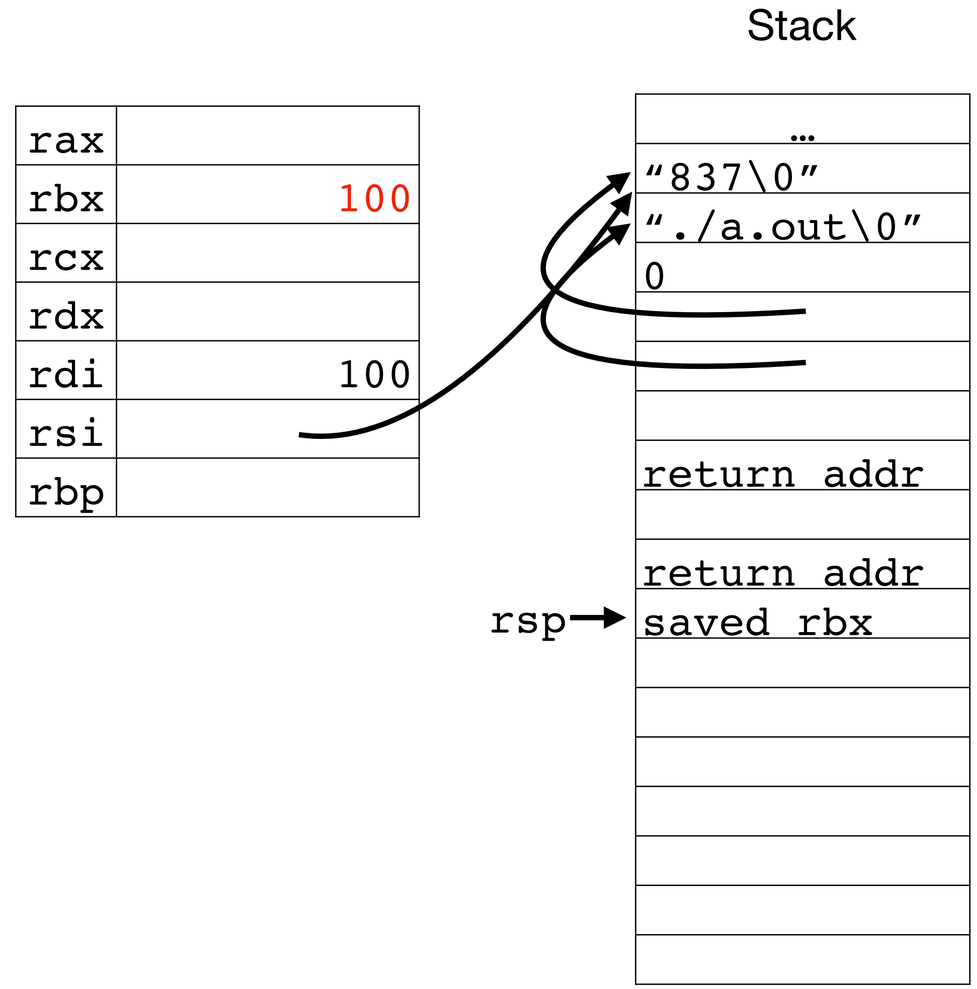
```
foo:
        push    rbx
  rip→  mov     ebx, edi
        mov     rdi, rsi
        mov     edx, 10
        mov     esi, 0
        call    strtol
        add     ebx, eax
        mov     eax, ebx
        pop     rbx
        ret
```

```
$ ./a.out 837
```



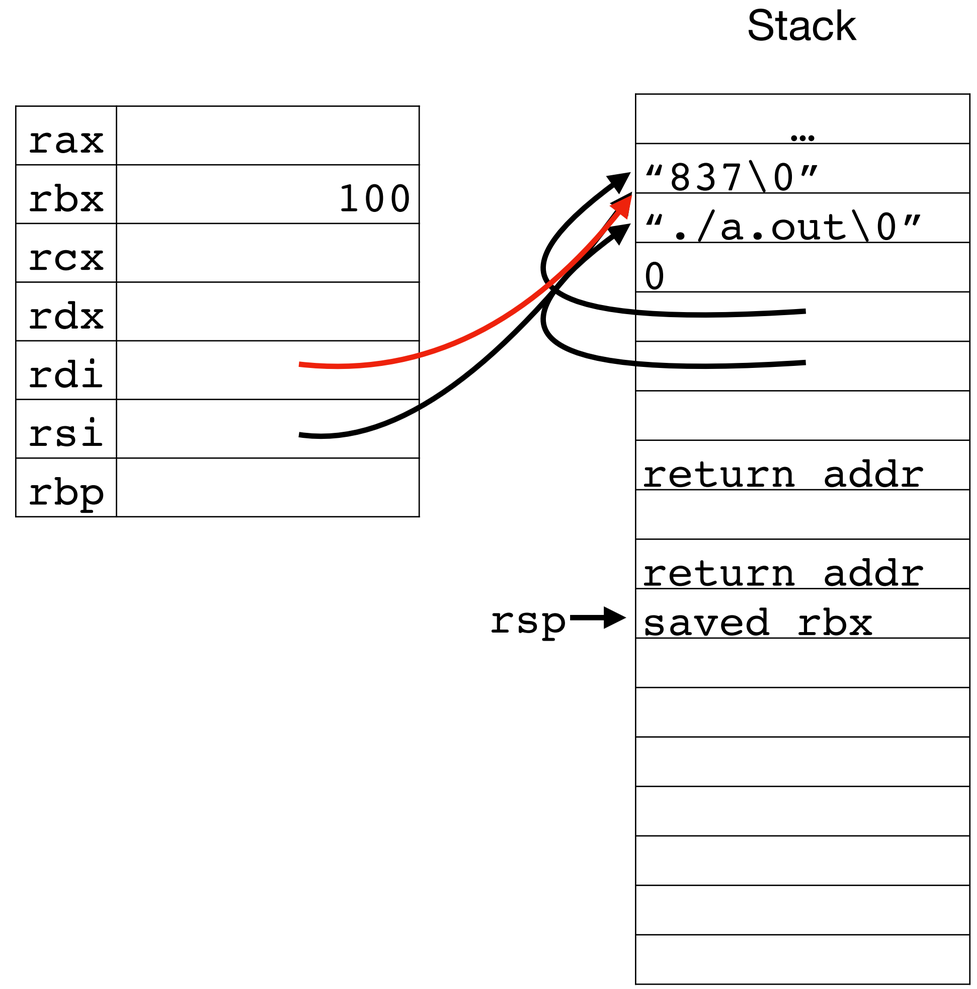
```
foo:
        push    rbx
        mov     ebx, edi
rip→    mov     rdi, rsi
        mov     edx, 10
        mov     esi, 0
        call    strtol
        add     ebx, eax
        mov     eax, ebx
        pop     rbx
        ret
```

```
$ ./a.out 837
```



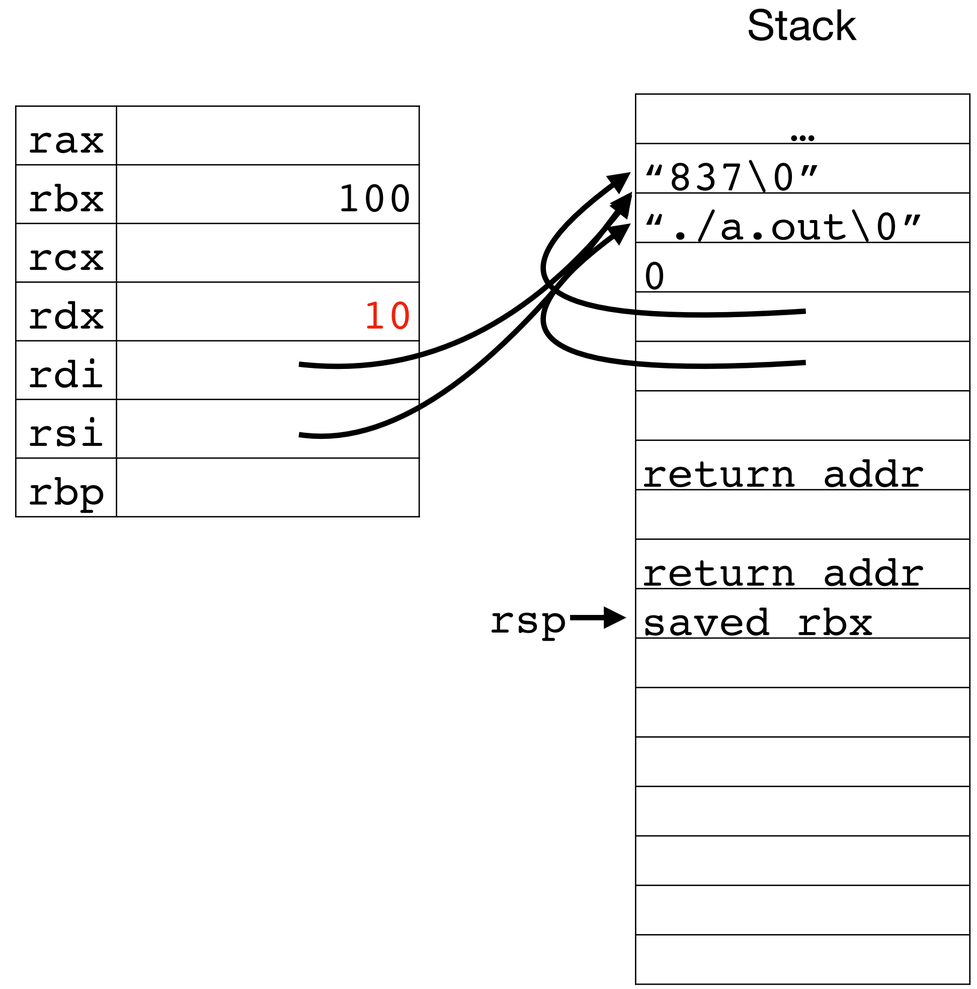
```
foo:
        push    rbx
        mov     ebx, edi
        mov     rdi, rsi
rip→    mov     edx, 10
        mov     esi, 0
        call    strtol
        add     ebx, eax
        mov     eax, ebx
        pop     rbx
        ret
```

```
$ ./a.out 837
```



```
foo:
        push    rbx
        mov     ebx, edi
        mov     rdi, rsi
        mov     edx, 10
rip→    mov     esi, 0
        call    strtol
        add     ebx, eax
        mov     eax, ebx
        pop     rbx
        ret
```

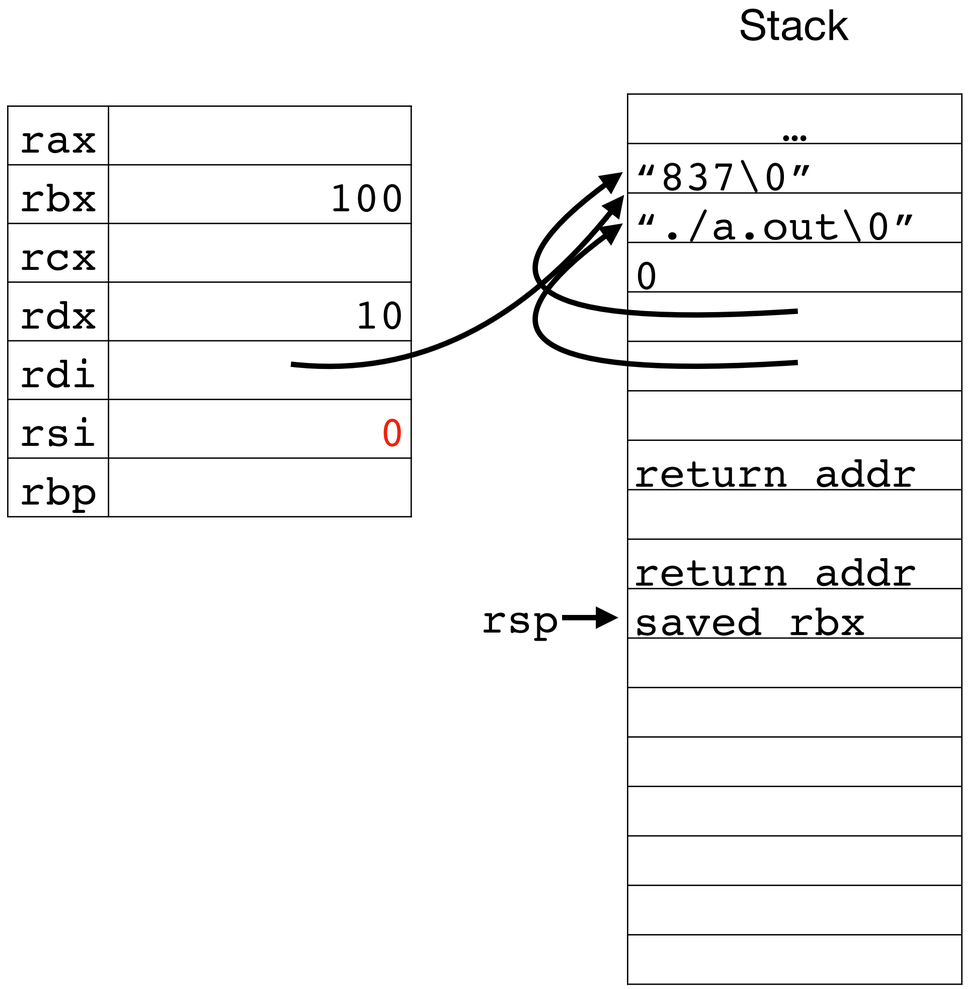
```
$ ./a.out 837
```



```
foo:
        push    rbx
        mov     ebx, edi
        mov     rdi, rsi
        mov     edx, 10
        mov     esi, 0
rip→    call    strtol
        add     ebx, eax
        mov     eax, ebx
        pop     rbx
        ret
```

```
$ ./a.out 837
```

call will set rip to the first instruction in strtol and push the address of add ebx, eax onto the stack; the return address



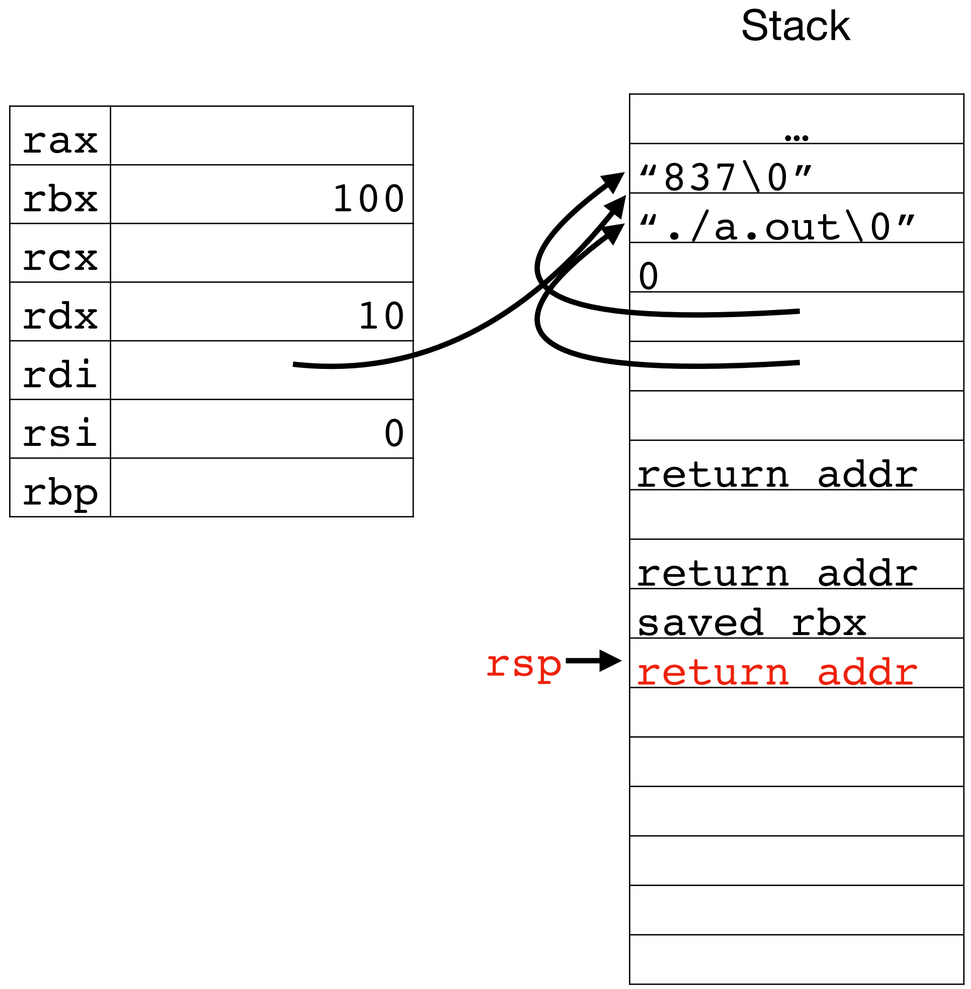
```
strtol:
  rip→ <body of strtol>
       ret
```

```
$ ./a.out 837
```

The strtol function will now run

It might change any register other than rbx, rsp, rbp, r12, r13, r14, and r15

Those 7 registers must be (and will be by correct code) restored to their values at the start of the function prior to ret



```
strtol:
        <body of strtol>
  rip→ ret
```

```
$ ./a.out 837
```

rbx will still hold 100

rsp will be pointing at the return address

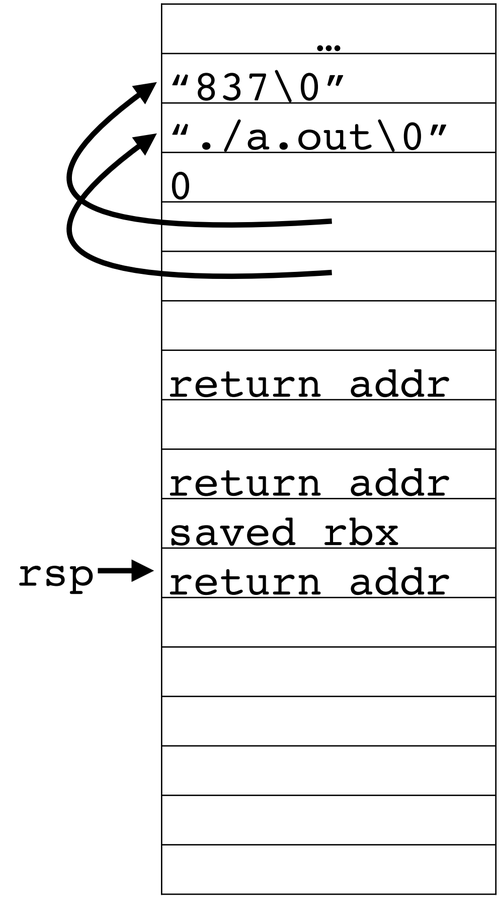
rax will hold the return value

ret will pop the top of the stack (the return address) into rip and will thus return to foo

Note that the stack doesn't change on a ret

rax	837
rbx	100
rcx	
rdx	
rdi	
rsi	
rbp	

Stack

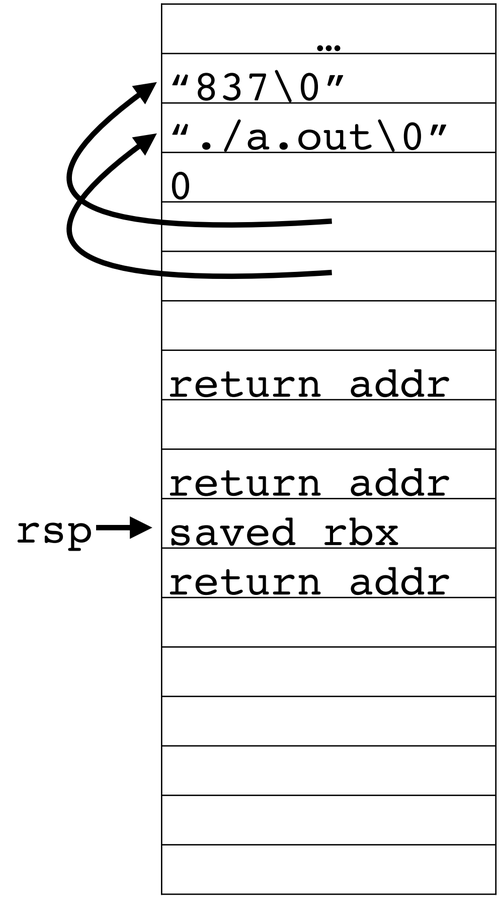


```
foo:
        push    rbx
        mov     ebx, edi
        mov     rdi, rsi
        mov     edx, 10
        mov     esi, 0
        call    strtol
rip→    add     ebx, eax
        mov     eax, ebx
        pop     rbx
        ret
```

```
$ ./a.out 837
```

rax	837
rbx	100
rcx	
rdx	
rdi	
rsi	
rbp	

Stack

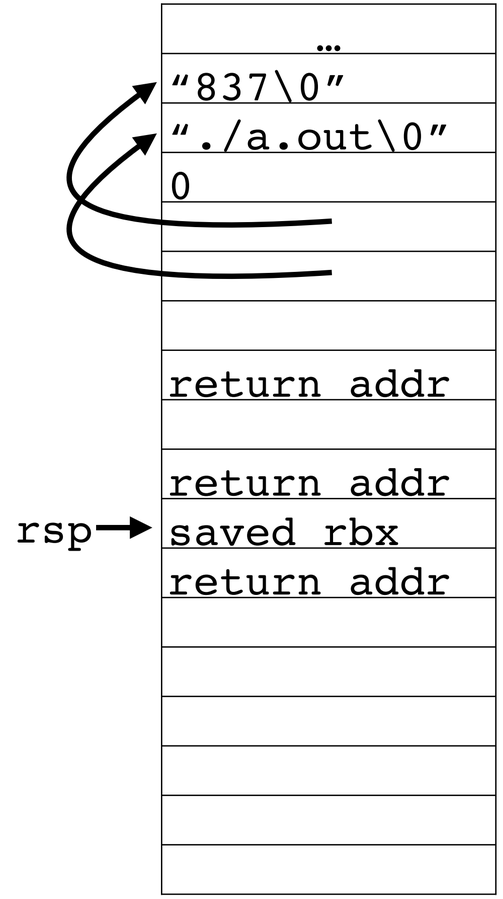


```
foo:
        push    rbx
        mov     ebx, edi
        mov     rdi, rsi
        mov     edx, 10
        mov     esi, 0
        call    strtol
        add     ebx, eax
rip→    mov     eax, ebx
        pop     rbx
        ret
```

```
$ ./a.out 837
```

rax	837
rbx	937
rcx	
rdx	
rdi	
rsi	
rbp	

Stack

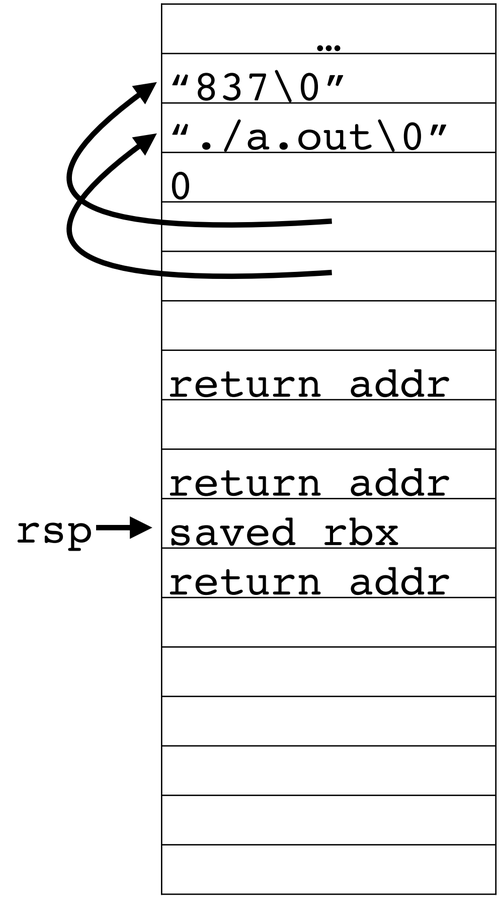


```
foo:
        push    rbx
        mov     ebx, edi
        mov     rdi, rsi
        mov     edx, 10
        mov     esi, 0
        call    strtol
        add     ebx, eax
        mov     eax, ebx
rip →   pop     rbx
        ret
```

```
$ ./a.out 837
```

rax	937
rbx	937
rcx	
rdx	
rdi	
rsi	
rbp	

Stack

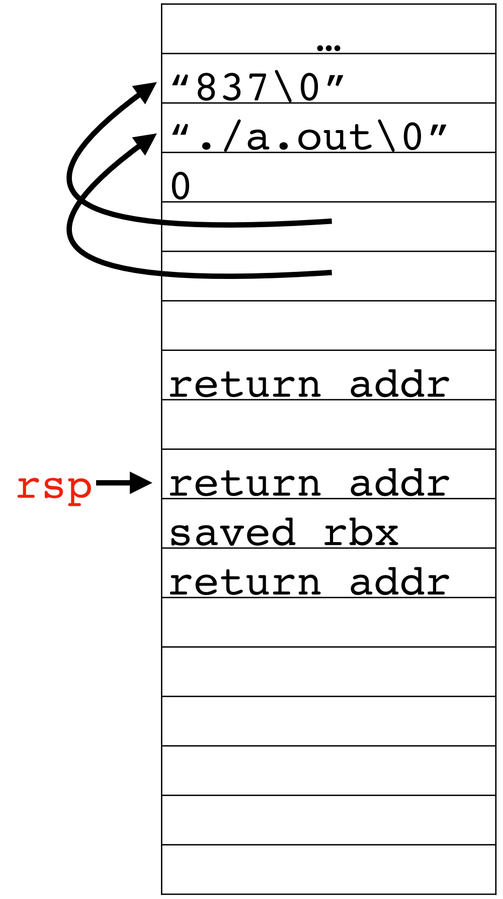


```
foo:
        push    rbx
        mov     ebx, edi
        mov     rdi, rsi
        mov     edx, 10
        mov     esi, 0
        call    strtol
        add     ebx, eax
        mov     eax, ebx
        pop     rbx
rip→    ret
```

```
$ ./a.out 837
```

rax	937
rbx	<original val>
rcx	
rdx	
rdi	
rsi	
rbp	

Stack



rbx has been restored to its original value

rsp will be pointing at the return address

rax will hold the return value

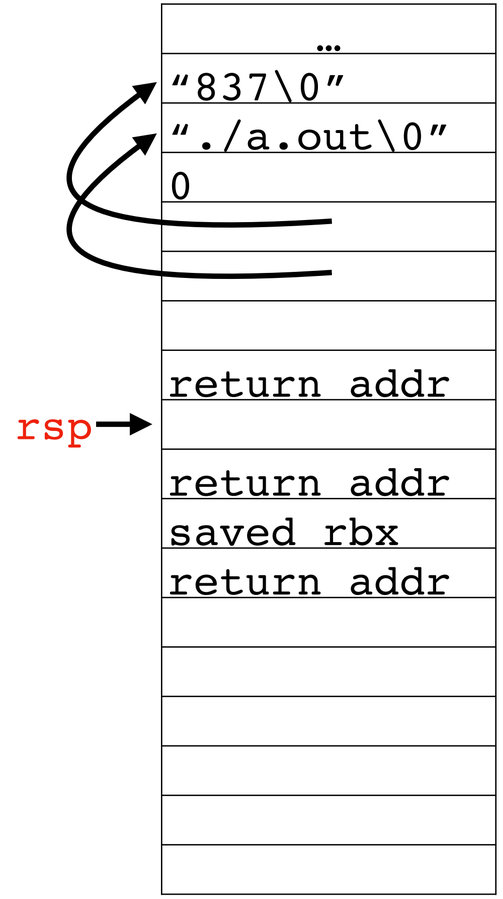
ret will pop the top of the stack (the return address) into rip and will thus return to main

```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
rip→    mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
```

rax	937
rbx	
rcx	
rdx	
rdi	
rsi	
rbp	

Stack



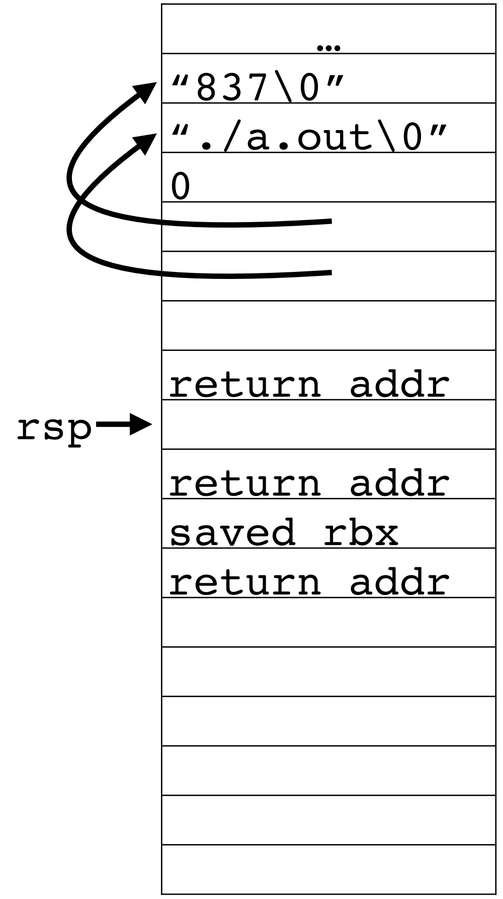
```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
rip→    mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
```

rax	937
rbx	
rcx	
rdx	
rdi	
rsi	937
rbp	

This will set rdi to point
to the first character
of the “%d\n” format string

Stack

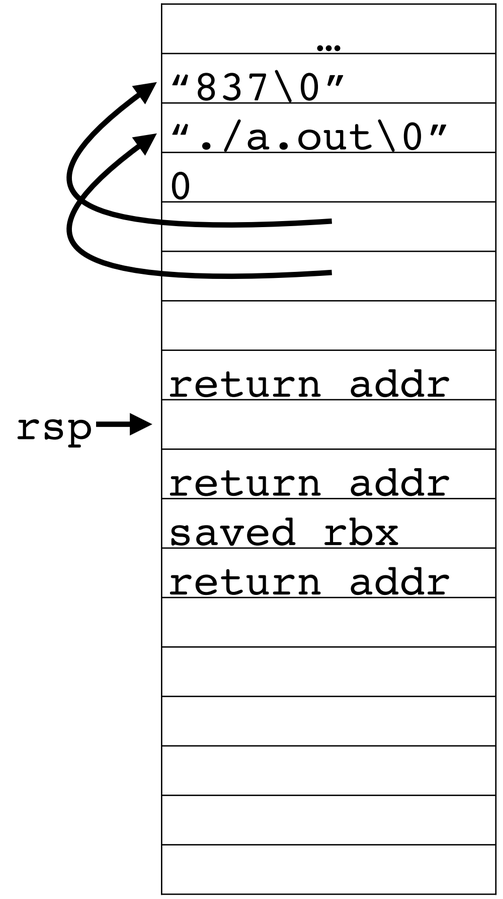


```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
  rip→  mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
```

rax	937
rbx	
rcx	
rdx	
rdi	.LC1
rsi	937
rbp	

Stack

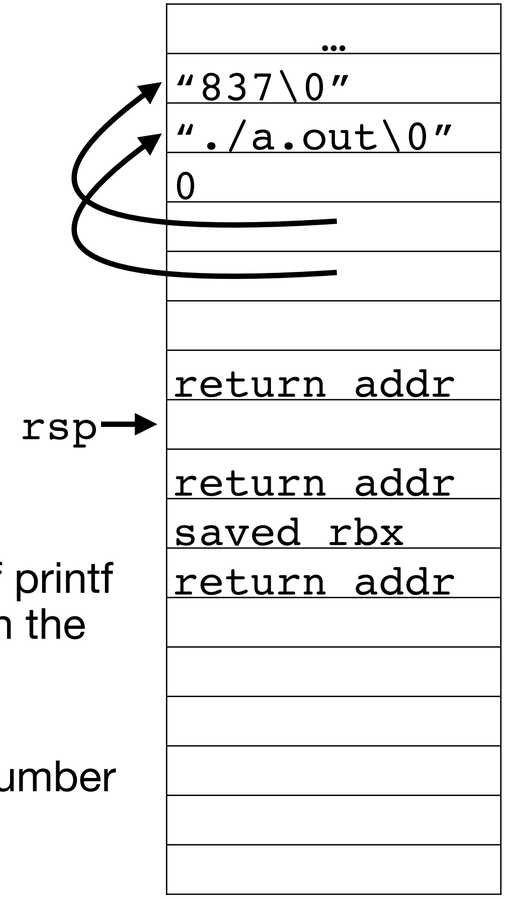


```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
rip→    call    printf
        mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
```

rax	0
rbx	
rcx	
rdx	
rdi	.LC1
rsi	937
rbp	

Stack



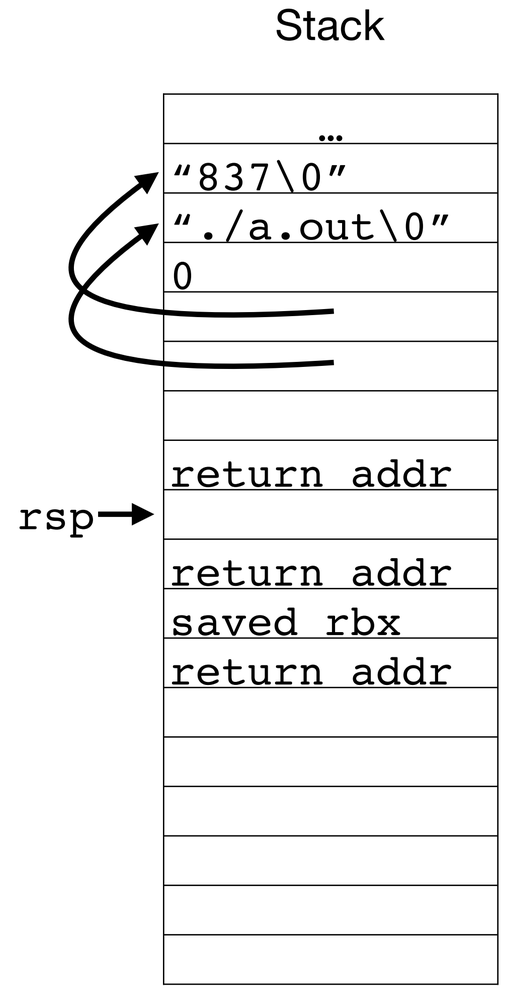
call will set rip to the first instruction of printf and push the address of mov eax, 0 on the stack as the return address

When printf returns, rax will hold the number of characters printed

```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
rip→    mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
937
```

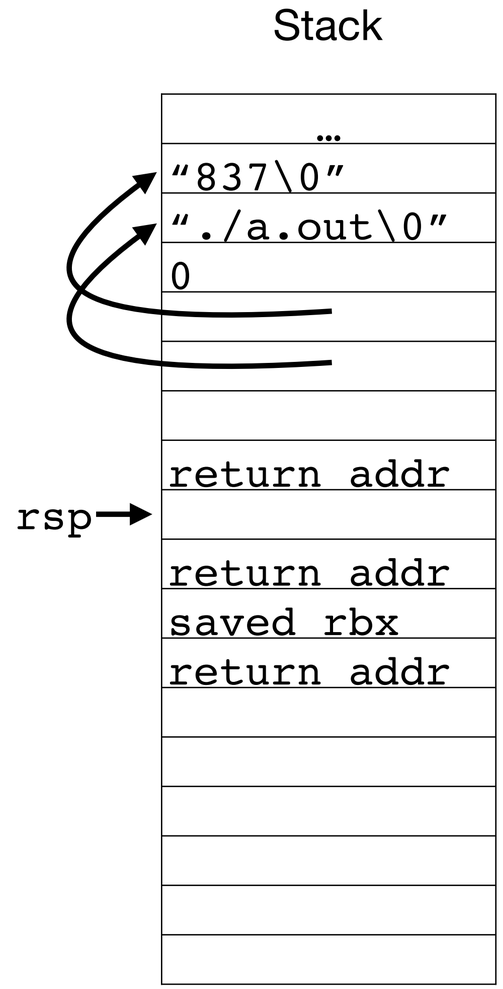
rax	4
rbx	
rcx	
rdx	
rdi	
rsi	
rbp	



```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
rip→    jmp     .L3
```

```
$ ./a.out 837
937
```

rax	0
rbx	
rcx	
rdx	
rdi	
rsi	
rbp	

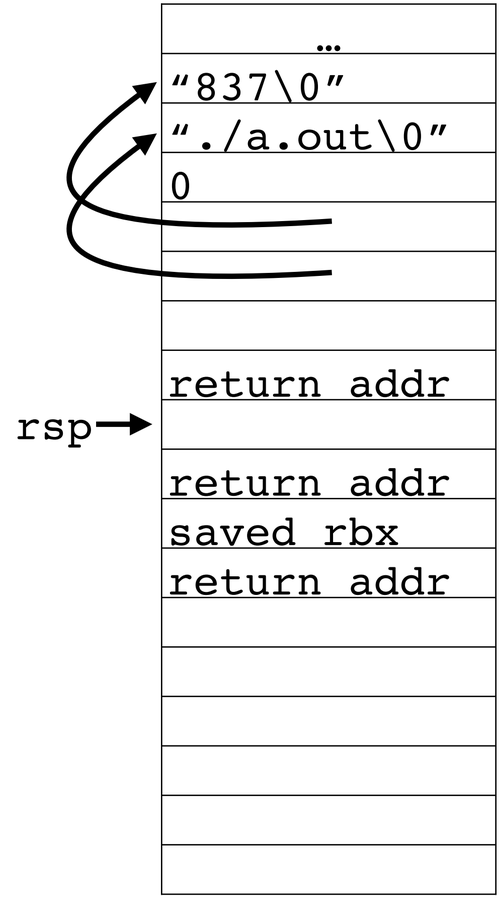


```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
  rip→  add     rsp, 8
        ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
937
```

rax	0
rbx	
rcx	
rdx	
rdi	
rsi	
rbp	

Stack

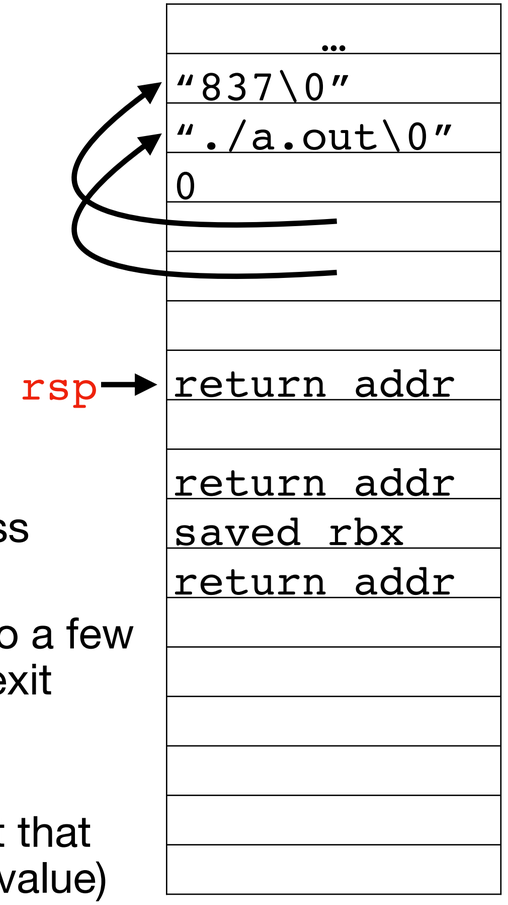


```
.LC0:
        .string "Usage: %s num\n"
.LC1:
        .string "%d\n"
main:
        sub     rsp, 8
        cmp     edi, 2
        je      .L4
        mov     rdx, QWORD PTR [rsi]
        mov     esi, OFFSET FLAT:.LC0
        mov     rdi, QWORD PTR stderr[rip]
        mov     eax, 0
        call    fprintf
        mov     eax, 1
.L3:
        add     rsp, 8
  rip→  ret
.L4:
        mov     rsi, QWORD PTR [rsi+8]
        mov     edi, 100
        call    foo
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
        mov     eax, 0
        jmp     .L3
```

```
$ ./a.out 837
937
```

rax	0
rbx	
rcx	
rdx	
rdi	
rsi	
rbp	

Stack



rsp points to main’s return address

When main returns, it will return to a few instructions which will make the exit system call

The kernel will end the process at that point with exit code 0 (the return value)

Key take aways from that

The machine language program (which corresponds 1-1 to the assembly language program) is the real program that is executed, regardless of what high-level language it comes from

The hardware doesn’t know anything about the programmer’s intent other than the machine language it is executing

We can understand exactly what is happening by looking at and stepping through assembly

The function call stack is ***super*** interesting in that it holds both control data (i.e., data which tells the processor what instruction to execute next on a return) as well as normal program data

Control-flow

Control flow is the order in which program instructions are executed

Most instructions execute and then control flows to the next instruction in memory

Some instructions (jmp, jz/jc/jnz/…, and call) directly encode the address of the next instruction to execute which is not the following instruction

- Aside: these use rip-relative addressing meaning the address is encoded as an offset from rip rather than the full 8-byte address

A few instructions use **data** to decide what to execute next

- ret — Pops the top of the **stack** into rip
- call rax — Calls the function whose address is in **rax**
- jmp rax — Jumps to the instruction whose address is in **rax**

Control-flow hijacking

Control-flow hijacking is a generic name for attacks on software that cause the control to flow somewhere other than intended by the programmer

Basic exploit idea

1. Inject some malicious code into the virtual address space of a process
2. Cause the address of that code to be loaded into rip

We’re going to learn multiple techniques for doing this as well as defenses against these techniques

What problems could happen with this code? Take a minute to think and select A when you have an answer.

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str); // copies the string str into the array buf
  do-something(buf);
}
```

Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

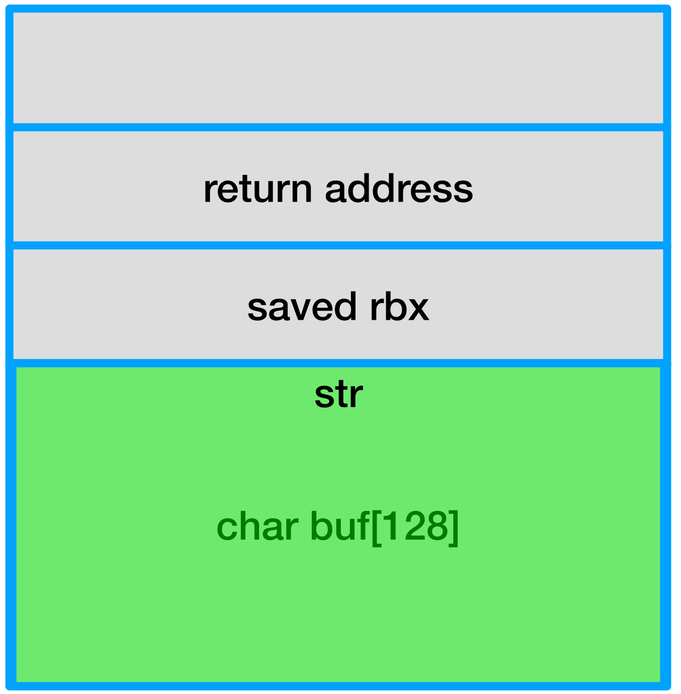
We don’t check the size of str!!!

The stack:

Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

The stack:



Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

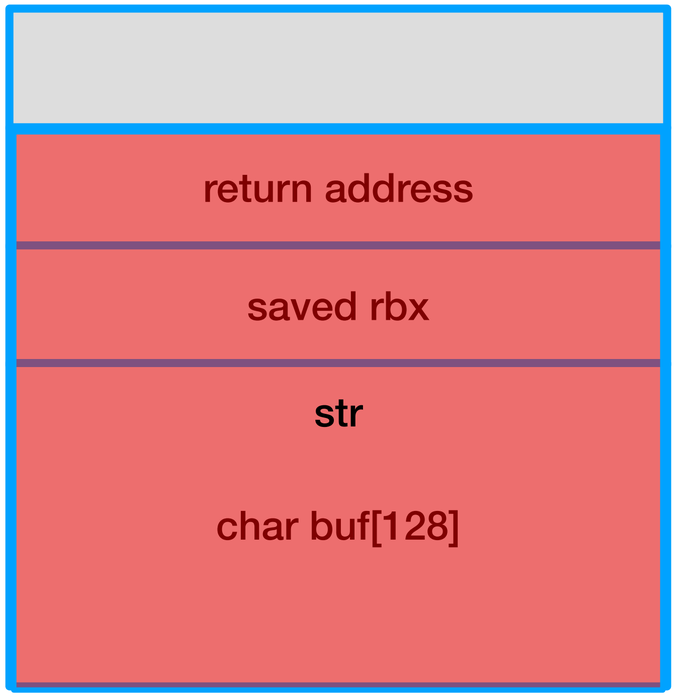
The stack:

What if str is >= 128 + 16 bytes long?

Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

The stack:

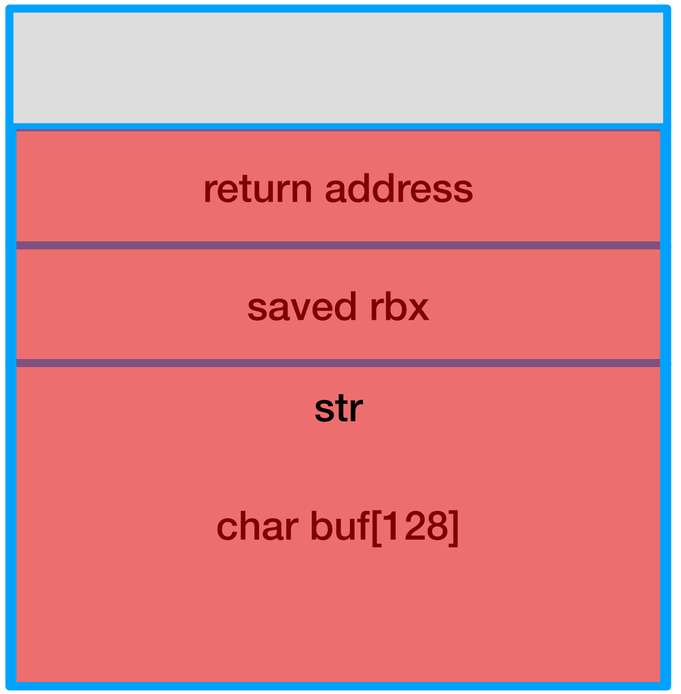


What if str is >= 128 + 16 bytes long?

Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

The stack:



We've overwritten the return address!

What happens when we overwrite the return address?

A. Nothing, the program would continue to execute normally

B. The program would crash immediately

C. The program would crash when the function returns

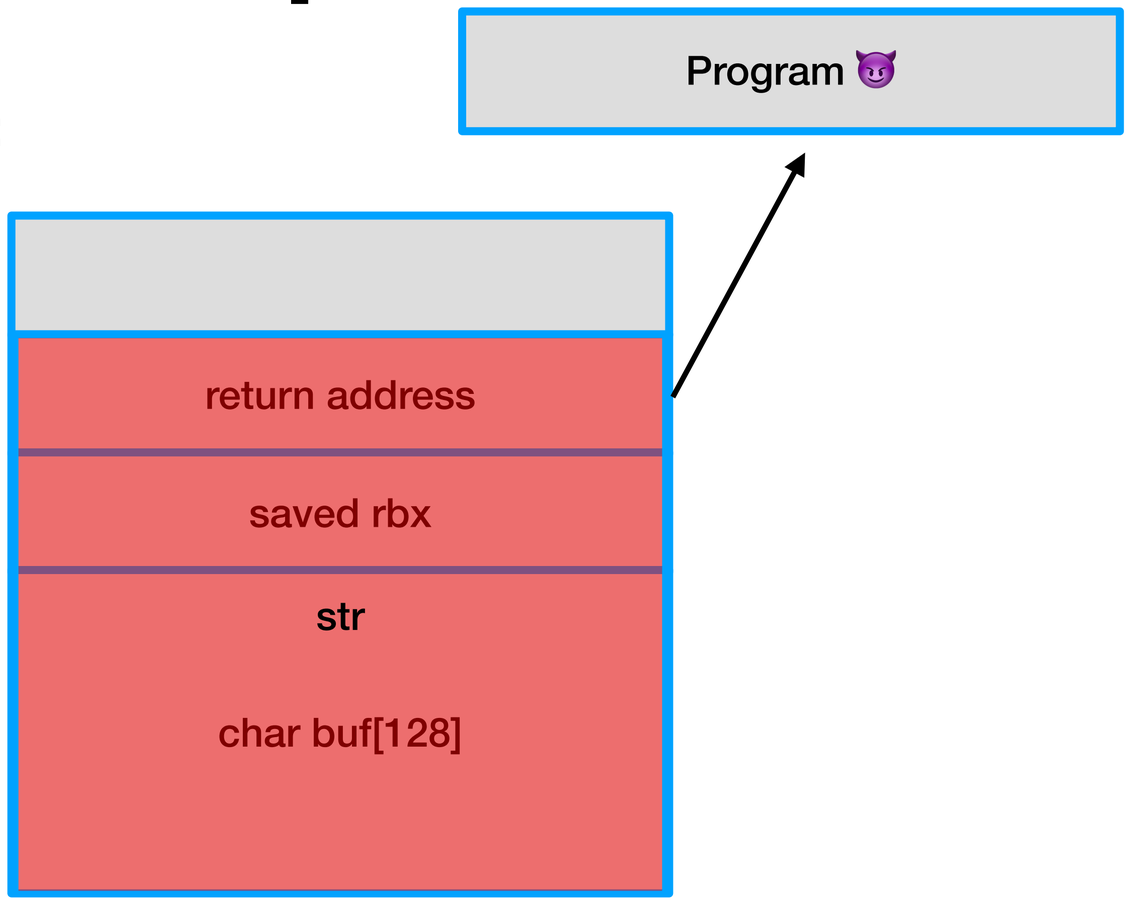
D. We would start executing a different program

E. Not sure/it depends

Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

The stack:



An attacker could construct a str such that the return address gets overwritten and now contains the memory address to a malicious program

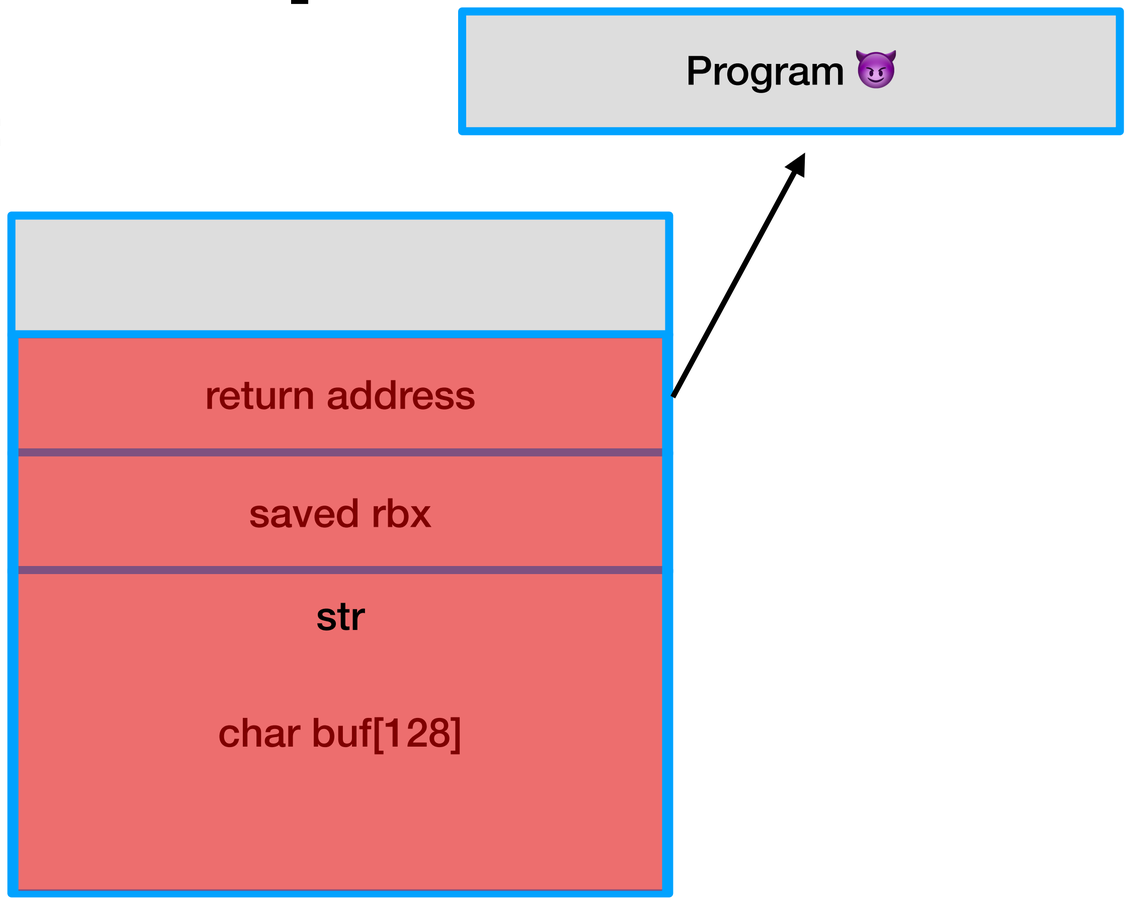
What strategies would you use to avoid this problem? Take a minute to think and select A when you have an answer.

A. Select A when you have an answer

Buffer Overflow Example

```
void func(char *str) {
  char buf[128];
  strcpy(buf, str);
  do-something(buf);
}
```

The stack:



An attacker could construct a str such that the return address gets overwritten and now contains the memory address to a malicious program

Where should the attacker place the malicious program in this case?

Next time

Next time, we'll construct a short, malicious, assembly program called "shellcode"

We'll see how to use it to exploit buffer overflows on the stack