Drawing: Function Call

- Take three minutes to draw something related to a “Function Call”

- Include the following
  - A diagram of the stack
  - A few of the relevant assembly instructions

Assembly

B :

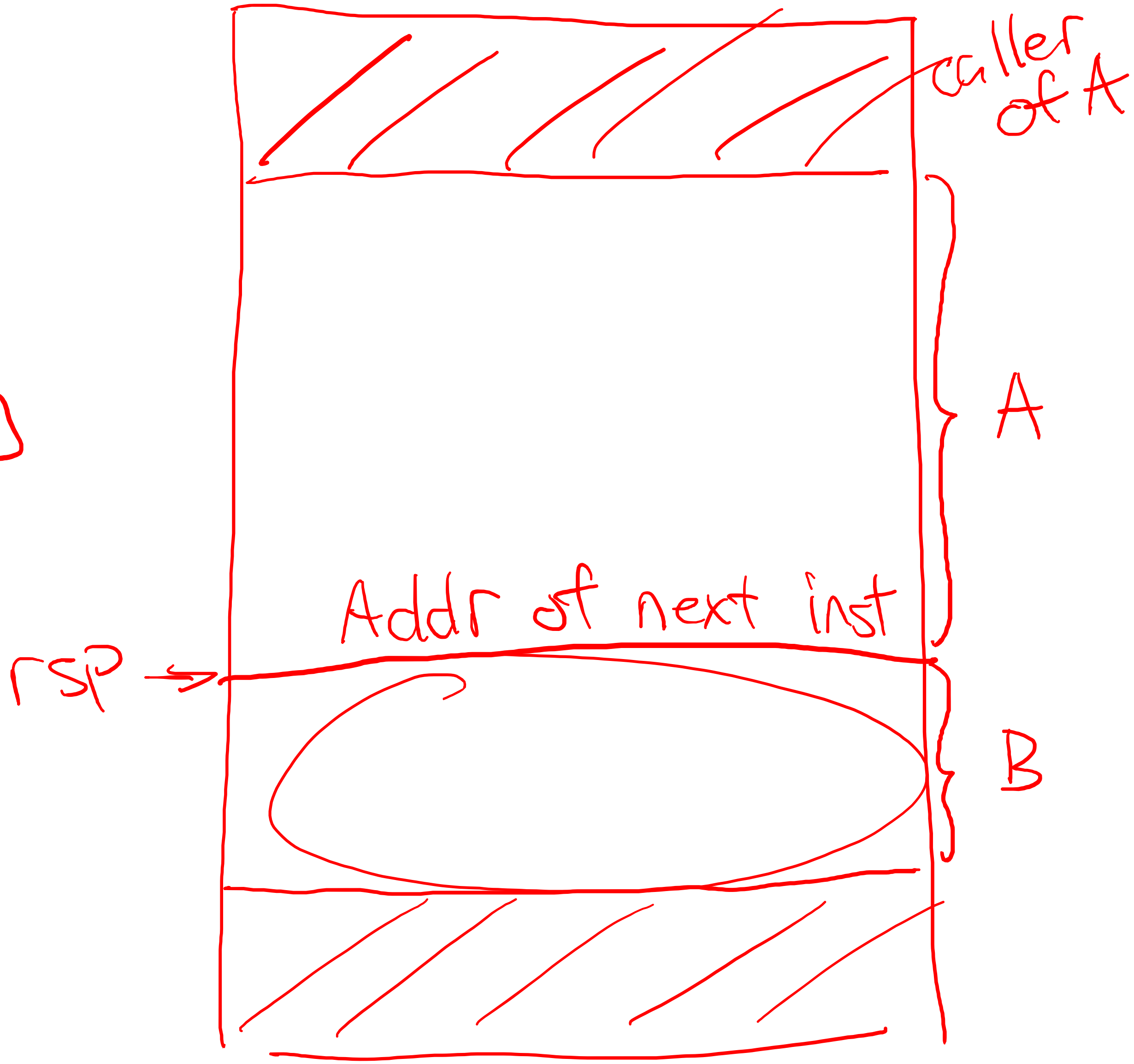
```
sub   rsp, 0x10
add
sub
mov   rax, [rsp + 0x8]
add   rsp, 0x10
ret
```

A :

```
mov   rdi, 6
mov   rsi, [rsp+ 8]
call  B
// do something w/ rax
```

rip →

Stack

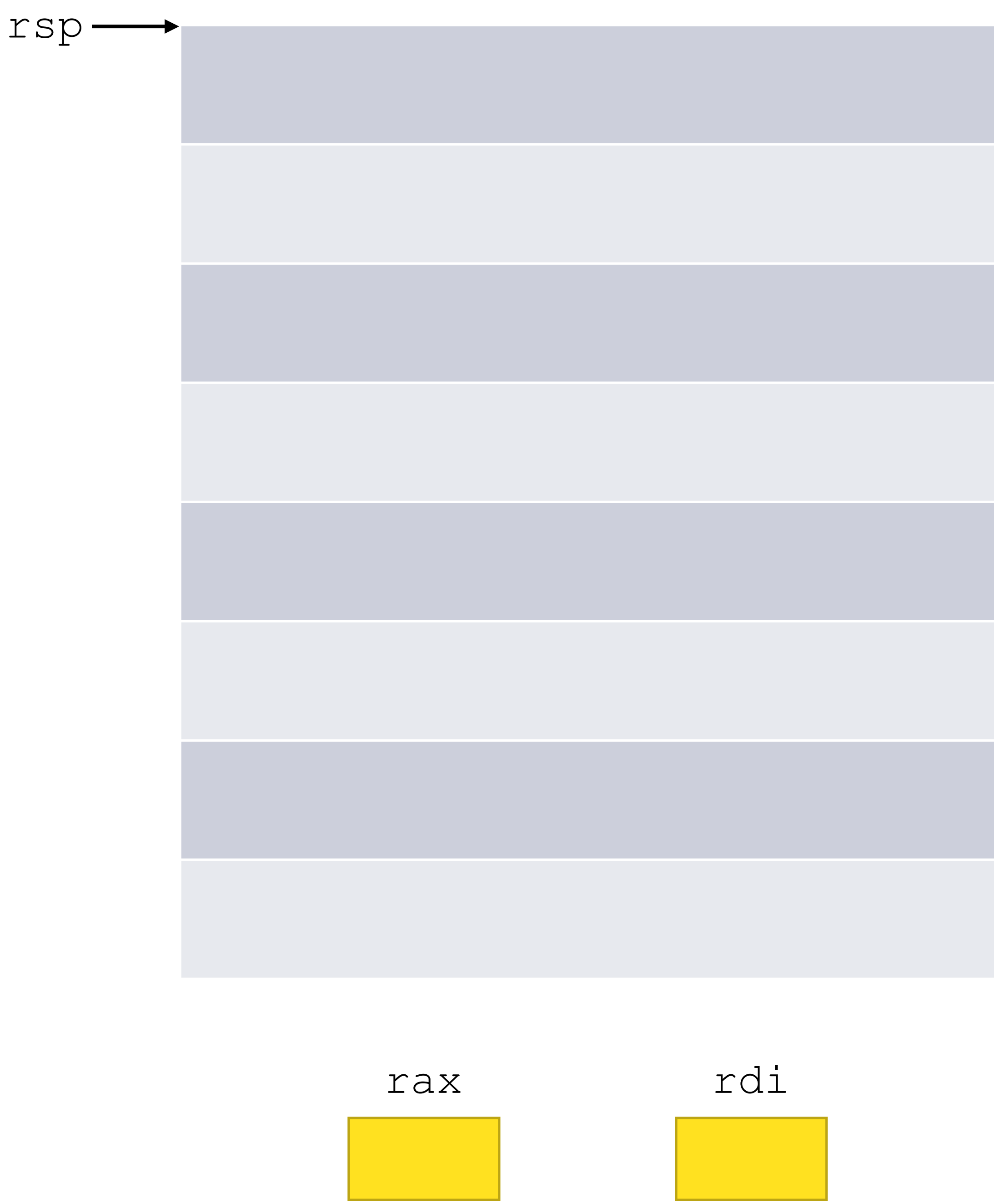


Security and Overflows

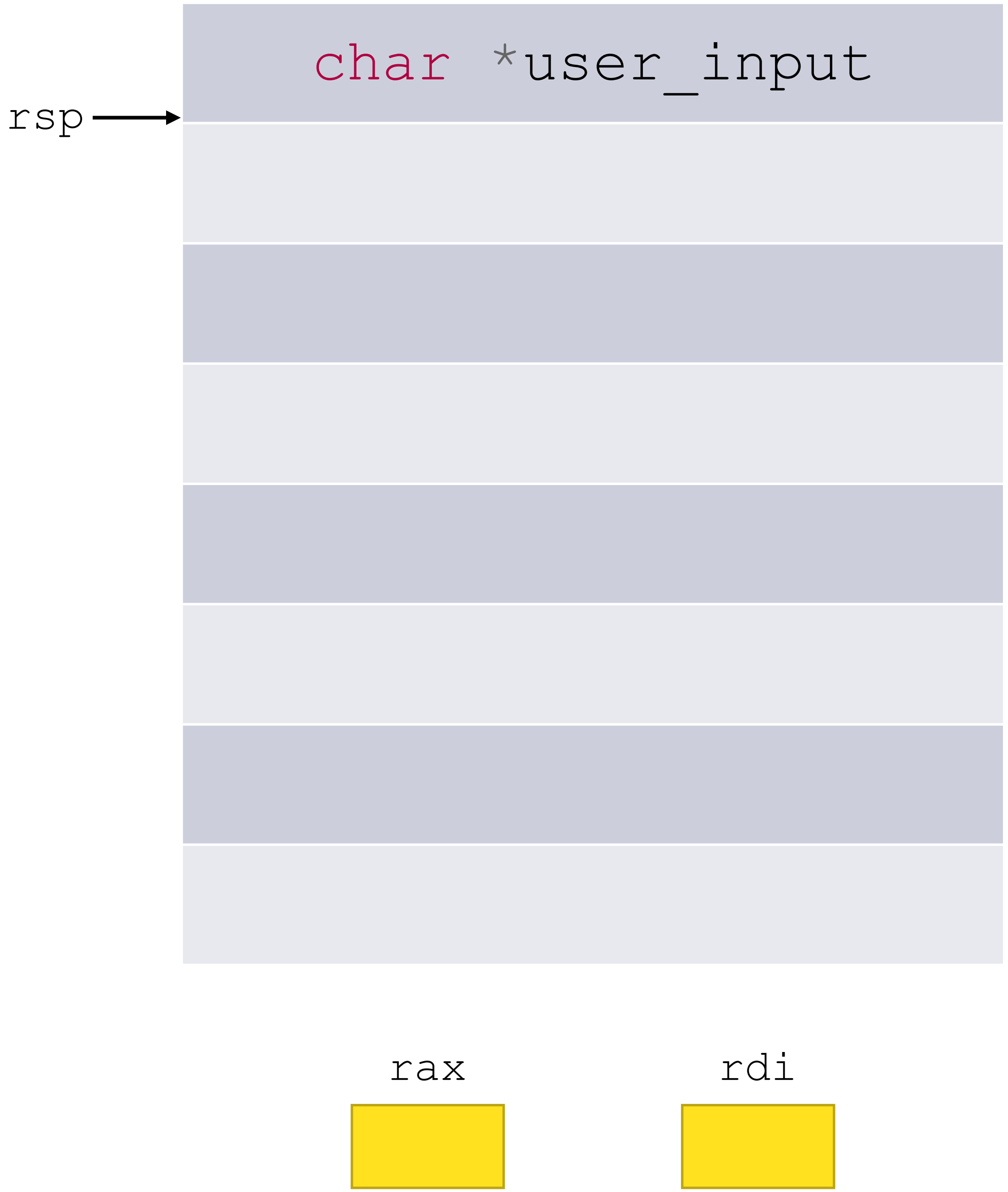
Stack Smashing and Return Oriented Programming

```c
#include <stdio.h>

char *read_string() {
    char char_buffer[10];
    gets(char_buffer);
    return char_buffer;
}

void print_string(char *str) {
    printf("%s\n", str);
}

int main() {
    char *user_input = read_string();
    print_string(user_input);
}
```

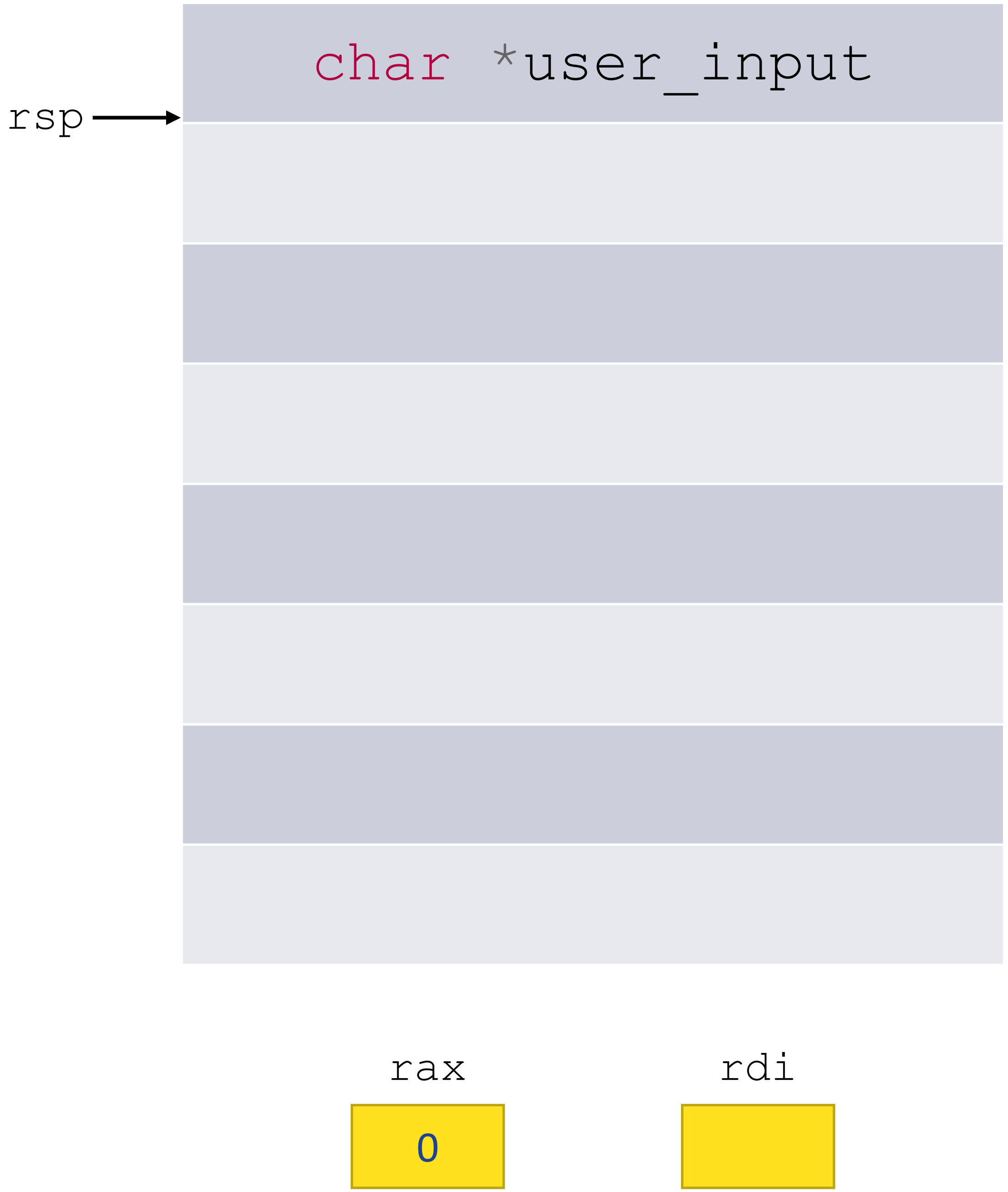
```asm
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```



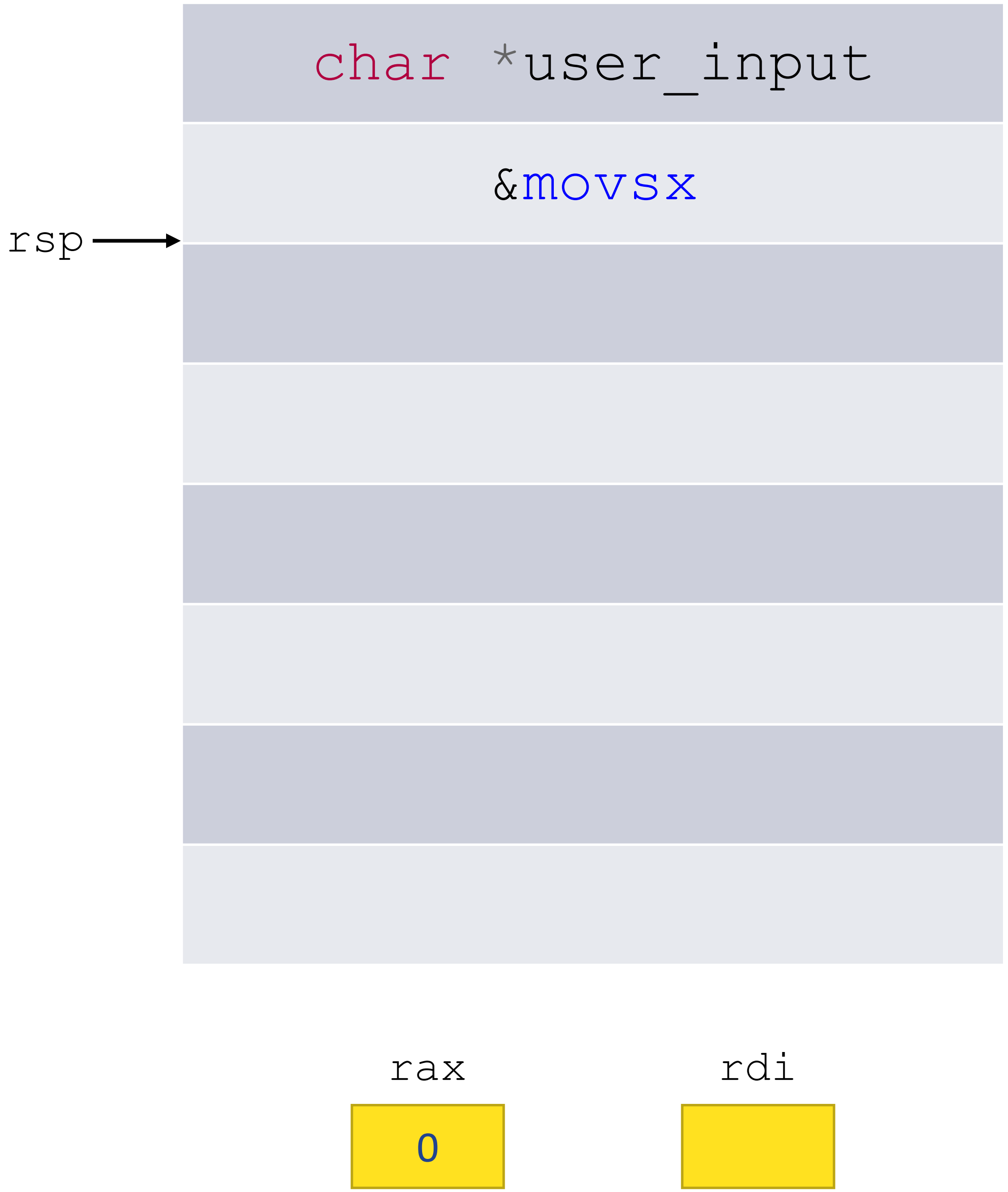

```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
  rip → sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```



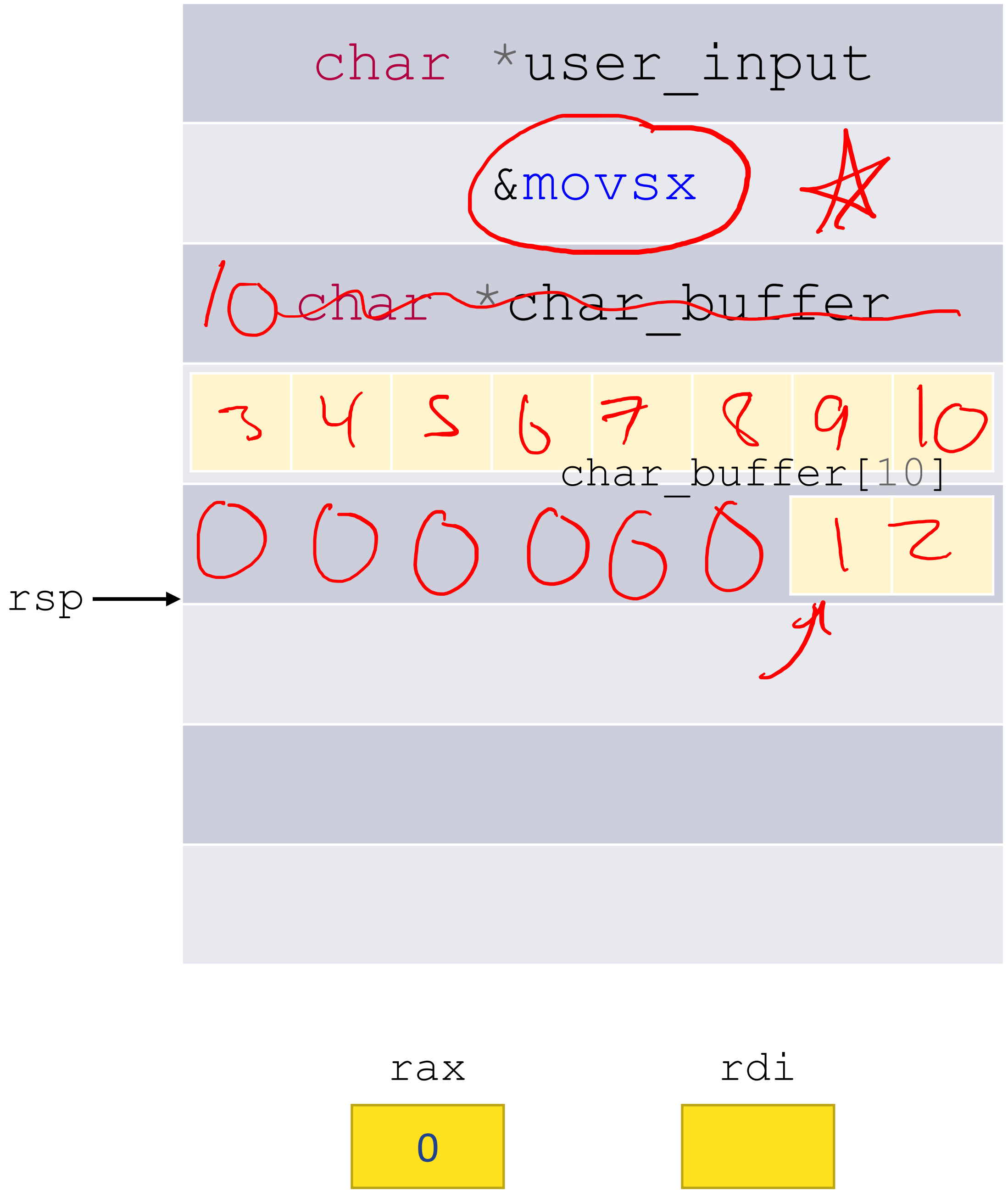

```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
rip --> mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```



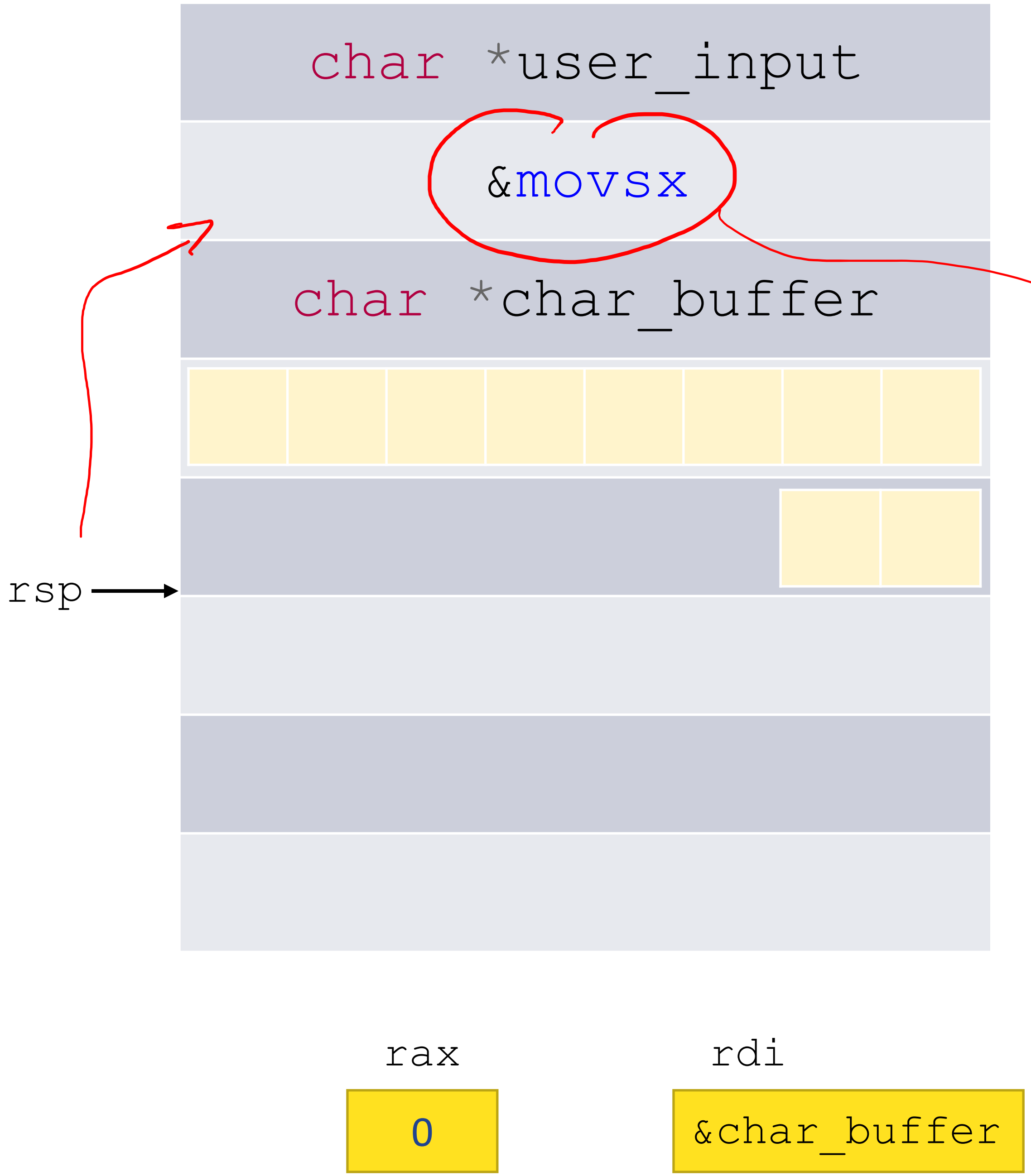

```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
rip --> call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```



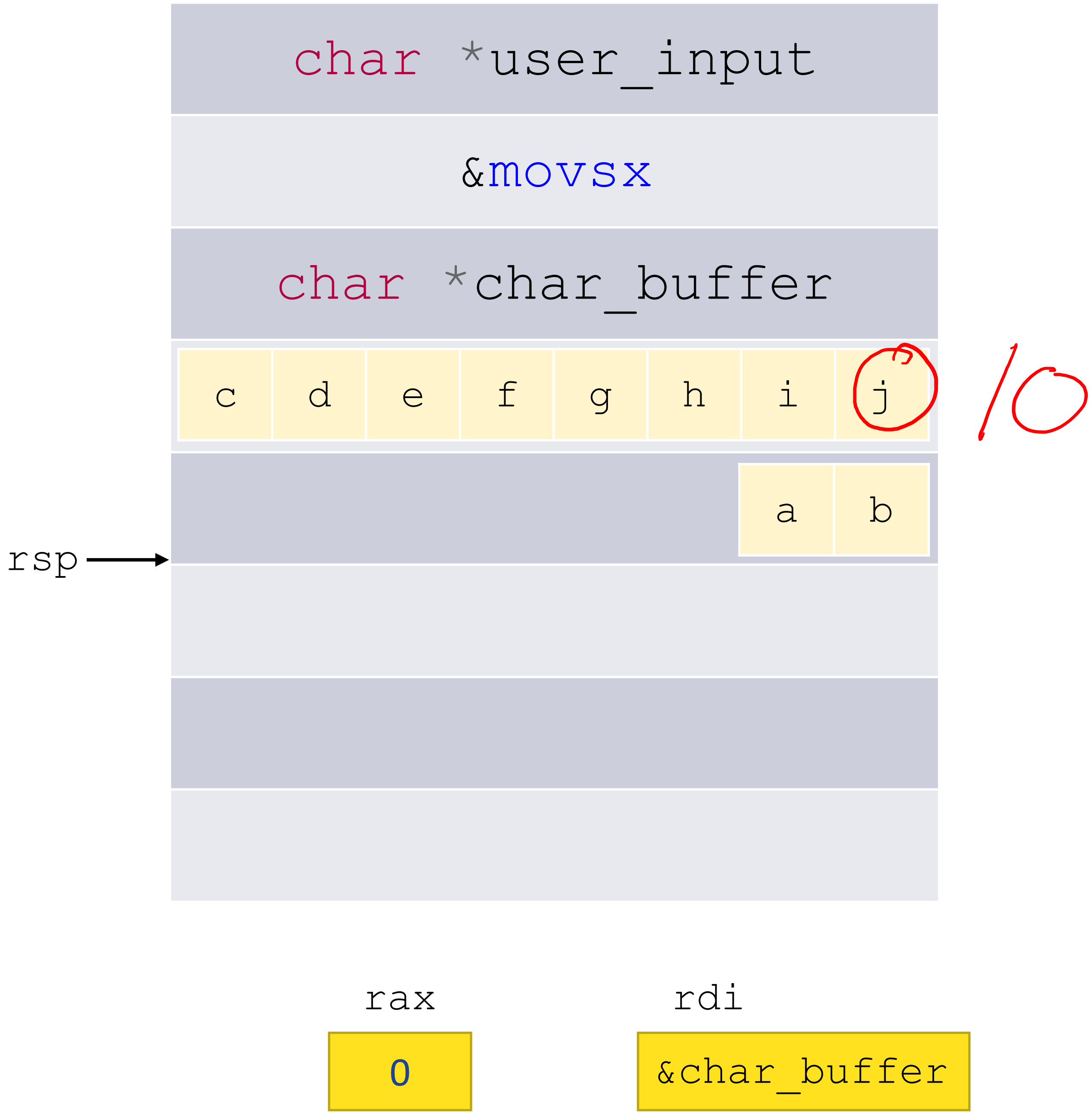

```
read_string:
 rip ──→ sub    rsp, 24
         lea    rdi, [rsp+6]
         mov    eax, 0
         call   gets
         mov    eax, 0
         add    rsp, 24
         ret

print_string:
         sub    rsp, 8
         call   puts
         add    rsp, 8
         ret

main:
         sub    rsp, 8
         mov    eax, 0
         call   read_string
         movsx  rdi, eax
         mov    eax, 0
         call   print_string
         mov    eax, 0
         add    rsp, 8
         ret
```



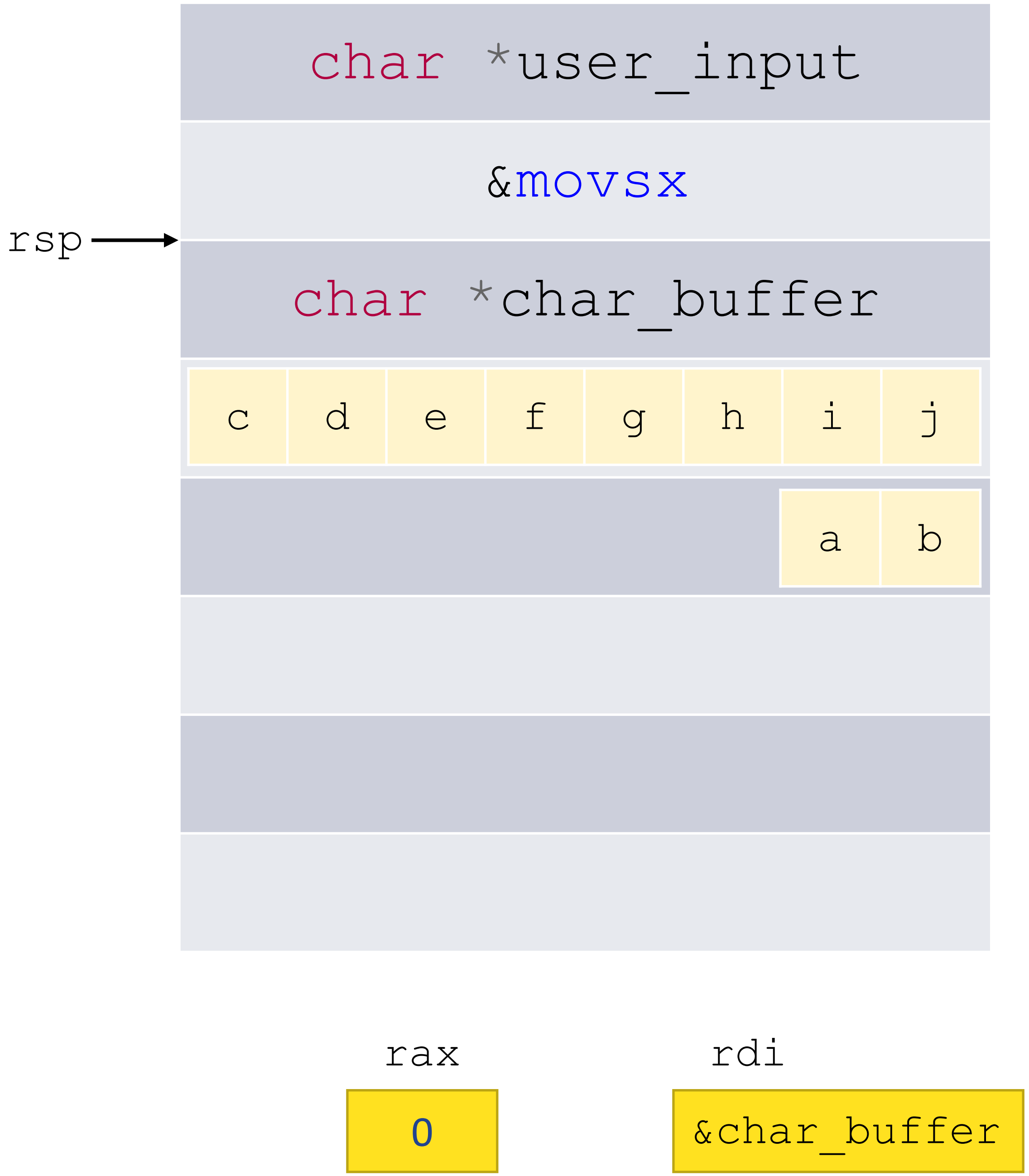

```
read_string:
        sub     rsp, 24
rip --> lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```



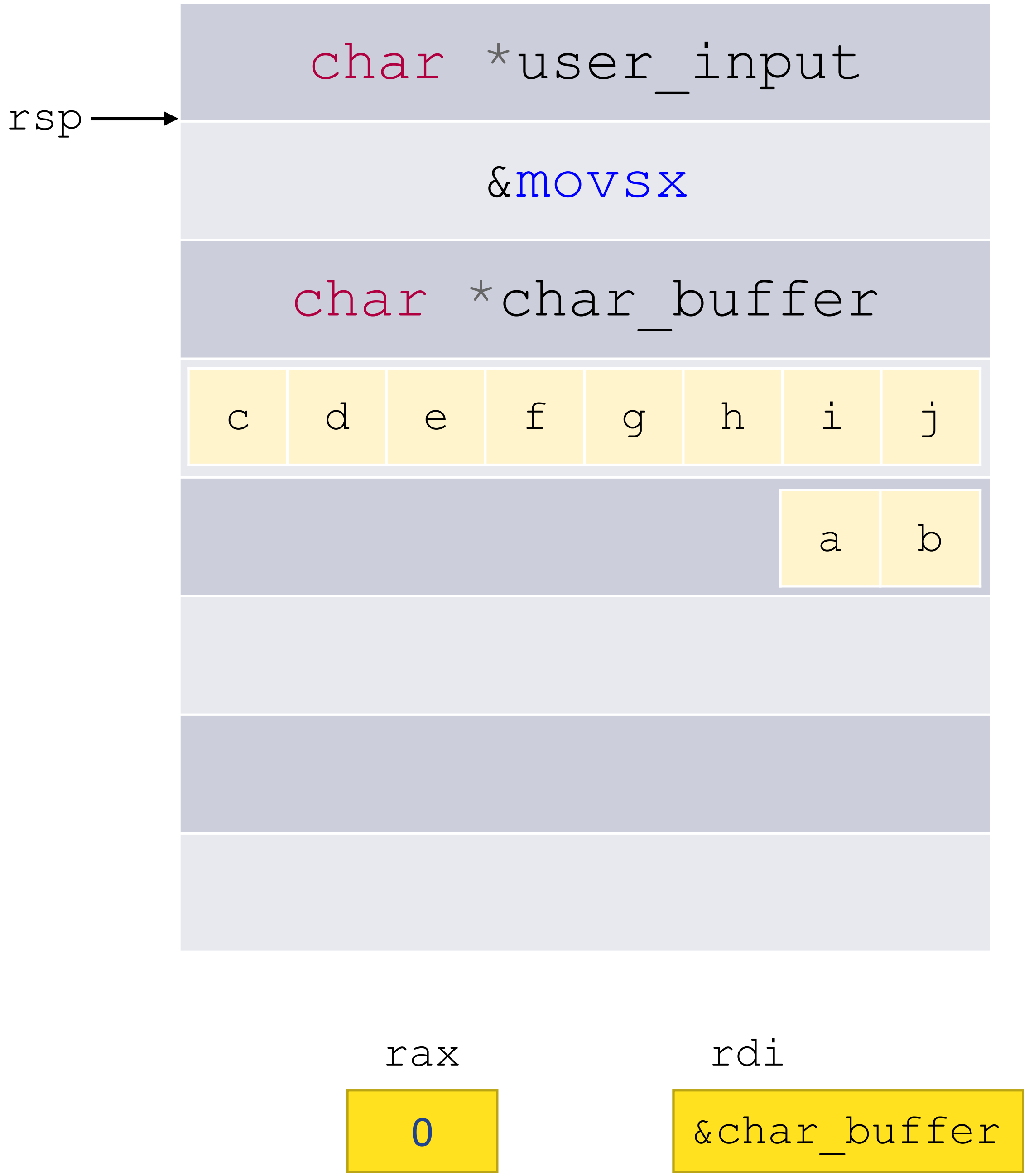

```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
rip --> mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```



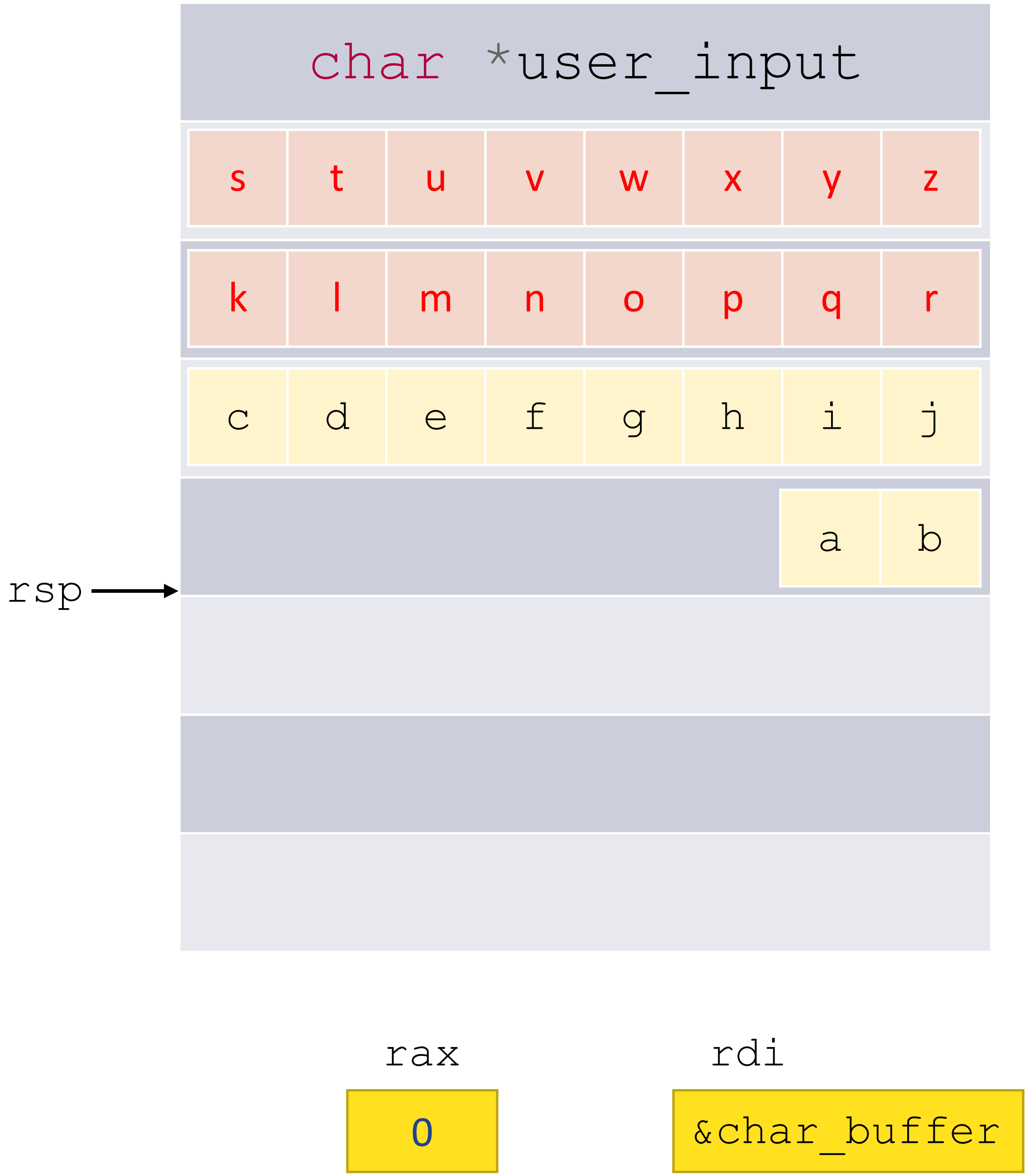

```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
rip --> add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```



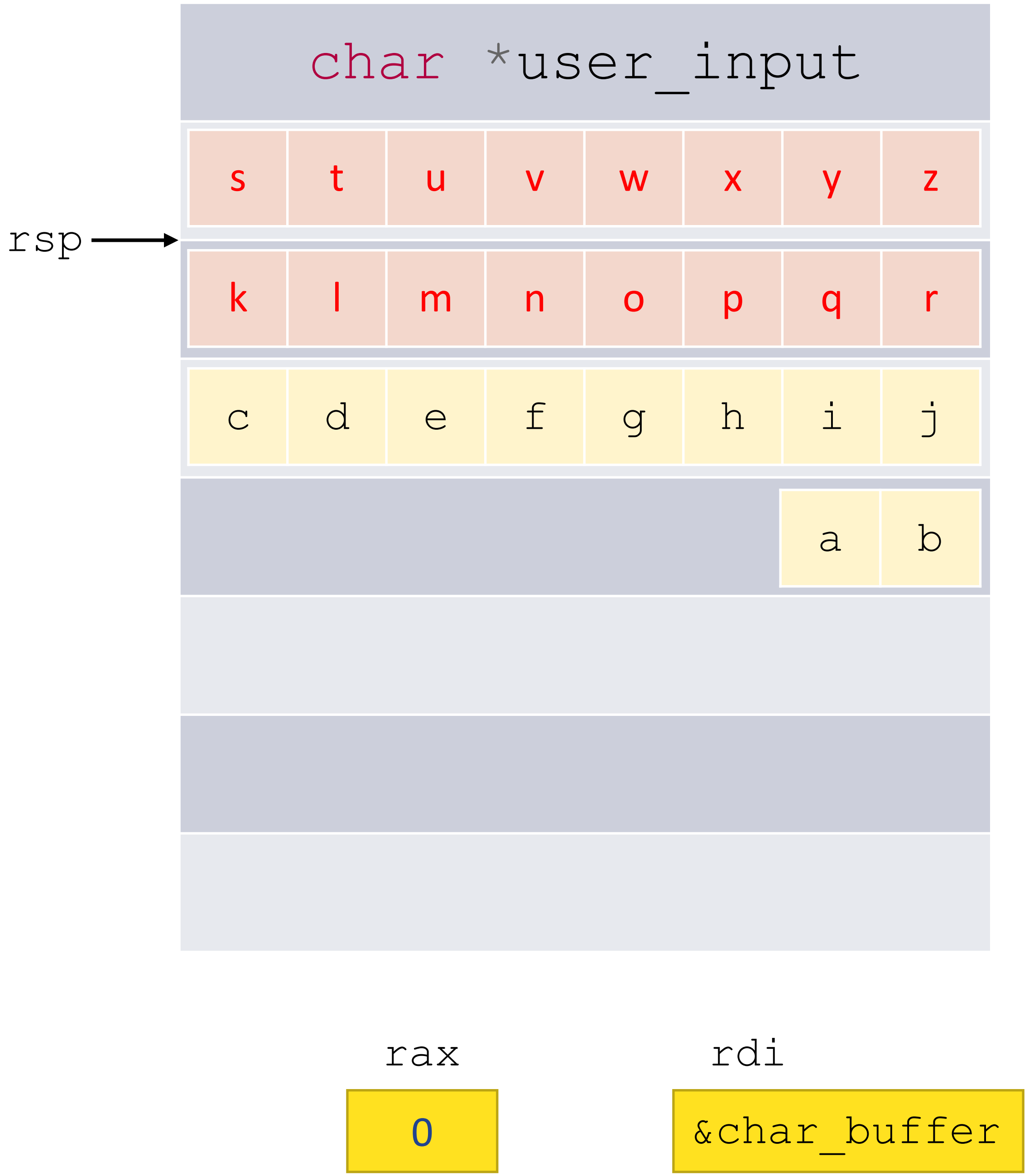

```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
rip --> ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```




```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
rip --> movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```




```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
rip --> add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```



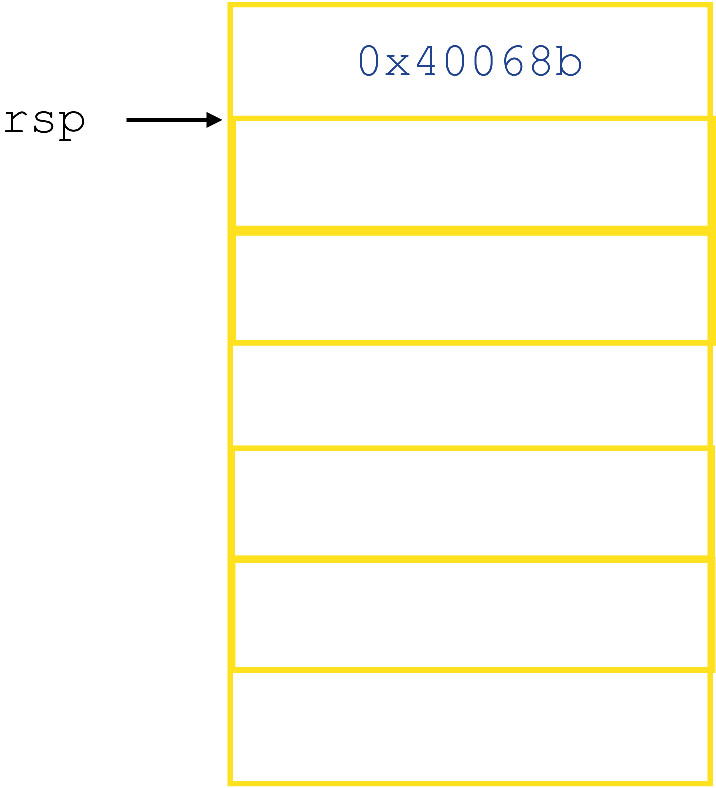

```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
rip --> add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```

Practice with Memory Bugs

- What is the state of the stack immediately before the program returns from f2?
- What will happen immediately after f2 returns?

```c
int f2(){
  int a1[4] = {1,2,3,4};
  a1[6] = 47;
}
```

```
f2:
rip →  sub    rsp, 0x18
       mov    [rsp]         , 0x1
       mov    [rsp + 0x4] , 0x2
       mov    [rsp + 0x8] , 0x3
       mov    [rsp + 0xC] , 0x4
       mov    [rsp + 0x18], 0x2F
       add    rsp, 0x18
       ret
```

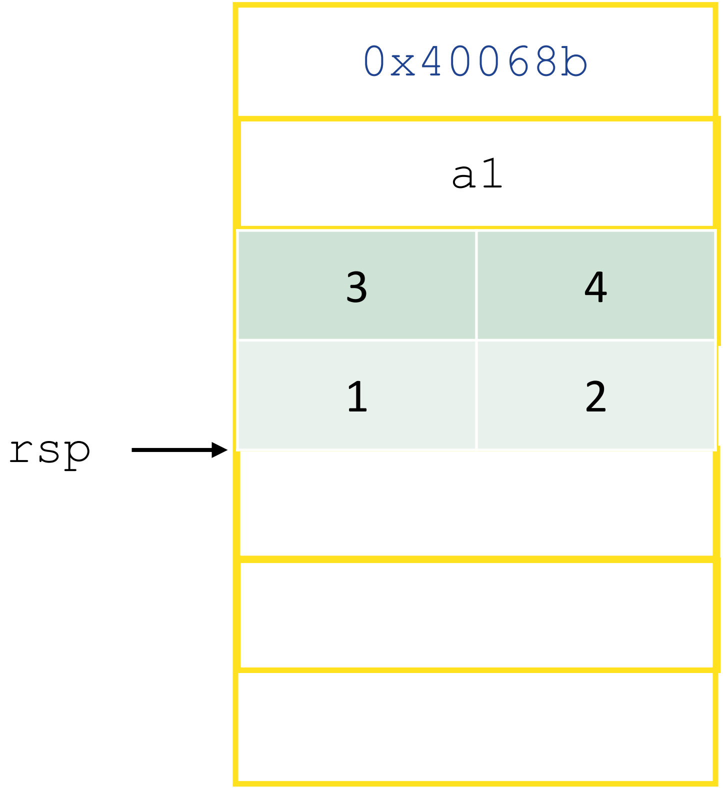


Practice with Memory Bugs

- What is the state of the stack immediately before the program returns from f2?
- What will happen immediately after f2 returns?

```c
int f2(){
   int a1[4] = {1,2,3,4};
   a1[6] = 47;
}
```

```
f2:
        sub     rsp, 0x18
        mov     [rsp]        , 0x1
        mov     [rsp + 0x4] , 0x2
        mov     [rsp + 0x8] , 0x3
        mov     [rsp + 0xC] , 0x4
rip →   mov     [rsp + 0x18], 0x2F
        add     rsp, 0x18
        ret
```

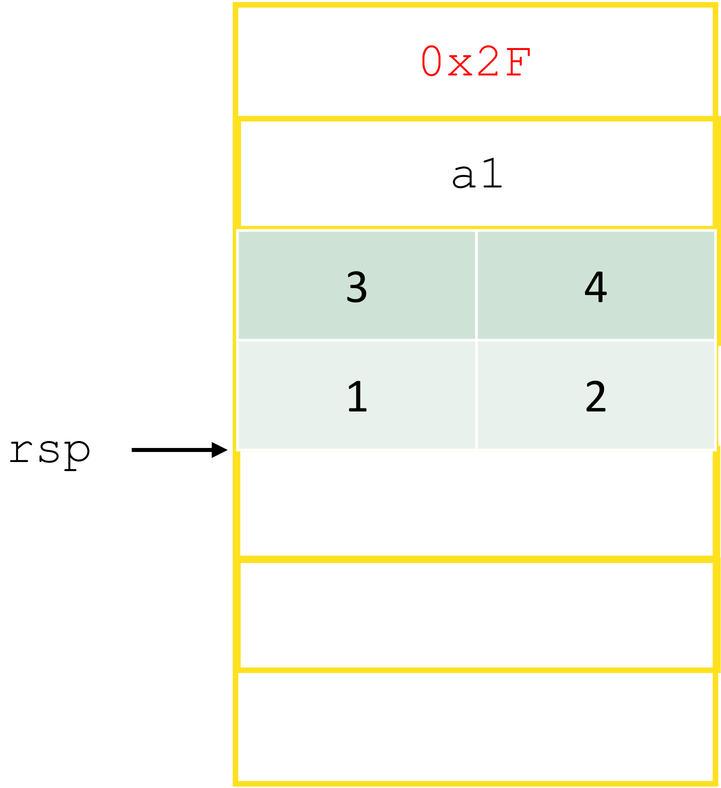


Practice with Memory Bugs

- What is the state of the stack immediately before the program returns from f2?
- What will happen immediately after f2 returns?

```c
int f2(){
  int a1[4] = {1,2,3,4};
  a1[6] = 47;
}
```

```
f2:
    sub     rsp, 0x18
    mov     [rsp]        , 0x1
    mov     [rsp + 0x4]  , 0x2
    mov     [rsp + 0x8]  , 0x3
    mov     [rsp + 0xC]  , 0x4
    mov     [rsp + 0x18], 0x2F
rip →  add     rsp, 0x18
    ret
```

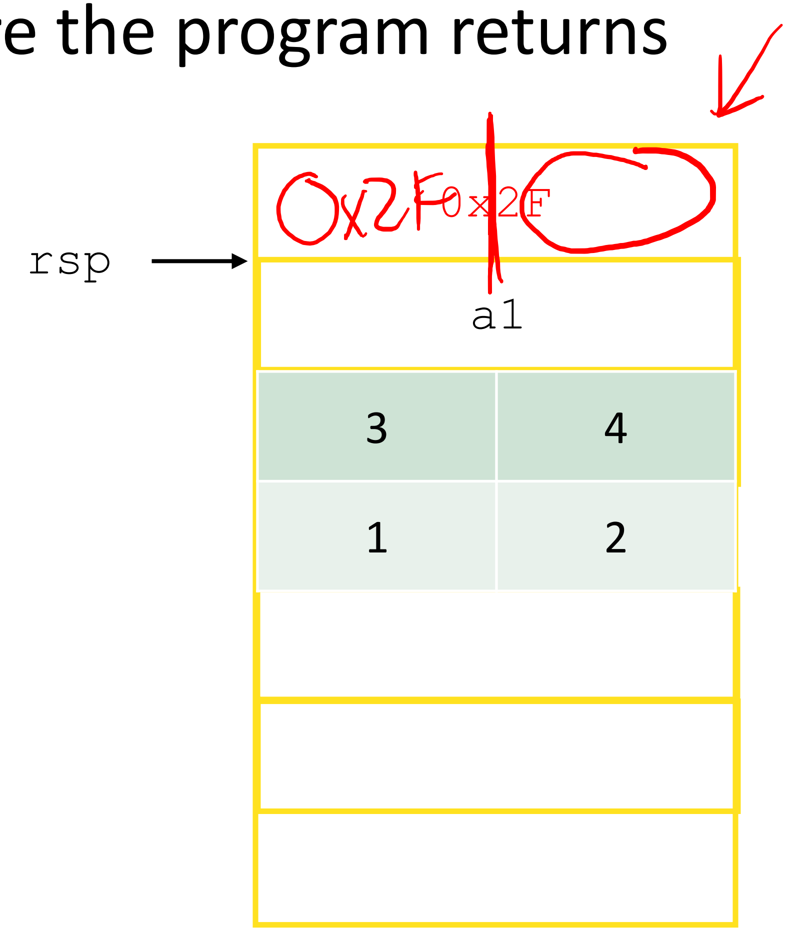


Practice with Memory Bugs

- What is the state of the stack immediately before the program returns from f2?
- What will happen immediately after f2 returns?

```c
int f2(){
  int a1[4] = {1,2,3,4};
  a1[6] = 47;
}
```

```
f2:
    sub   rsp, 0x18
    mov   [rsp]        , 0x1
    mov   [rsp + 0x4]  , 0x2
    mov   [rsp + 0x8]  , 0x3
    mov   [rsp + 0xC]  , 0x4
    mov   [rsp + 0x18], 0x2F
    add   rsp, 0x18
rip → ret
```



Practice with Buffer Overflow Exploits

Construct an exploit string that will successfully cause the program to print "You are now logged in" without knowing the correct password

1. How many bytes of padding are in this exploit string?
2. What value will you overwrite the return address with?

```
int authenticate(char *password){
  char buf[4];
  gets(buf);
  int correct = !strcmp(password, buf);
  return correct;
}


int main(int argc, char ** argv){
  char * pw = "123456";
  printf("Enter your password: ");
  while(!authenticate(pw)){
    printf("Incorrect. Try again: ");
  }
  printf("You are now logged in\n");
  return 0;
}
```

Practice ... rflow Exploits

Construct a[...] successfully [...] "You are no[...] knowing the[...]

1. How ma[...] this expl[...]

2. What va[...] return a[...]

```
.LC0:   .string "Enter your password: "
.LC1:   .string "Incorrect. Try again: "
.LC2:   .string "123456"
.LC3:   .string "You are now logged in"
authenticate:
        push    rbx
        sub     rsp, 16
        mov     rbx, rdi
        lea     rdi, [rsp+12]
        mov     eax, 0
        call    gets
        lea     rsi, [rsp+12]
        mov     rdi, rbx
        call    strcmp
        test    eax, eax
        sete    al
        movzx   eax, al
        add     rsp, 16
        pop     rbx
        ret
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        jmp     .L4
.L5:
        mov     edi, OFFSET FLAT:.LC1
        mov     eax, 0
        call    printf
.L4:
        mov     edi, OFFSET FLAT:.LC2
        call    authenticate
        test    eax, eax
        je      .L5
        mov     edi, OFFSET FLAT:.LC3
        call    puts
        mov     eax, 0
        add     rsp, 8
        ret
```

```c
int authenticate(char *password){
  char buf[4];
  gets(buf);
  int correct = !strcmp(password, buf);
  return correct;
}


int main(int argc, char ** argv){
  char * pw = "123456";
  printf("Enter your password: ");
  while(!authenticate(pw)){
    printf("Incorrect. Try again: ");
  }
  printf("You are now logged in\n");
  return 0;
}
```

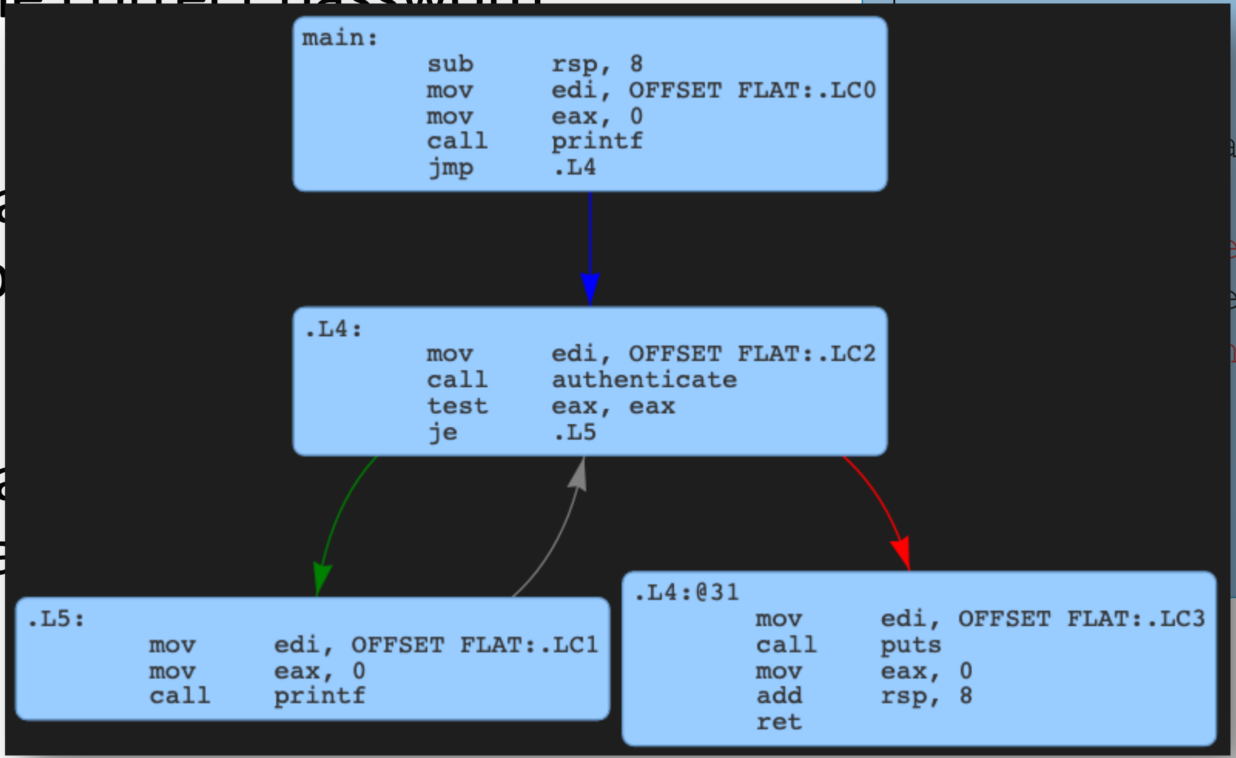
Practice with Buffer Overflow Exploits

Construct an exploit string that will successfully cause the program to print "You are now logged in" without knowing the correct password

1. How ma
   this exp

2. What va
   return a

```
int authenticat
  char buf[4];
  gets(buf);
  int correct =
  return correc
}
```




```
.LC0:    .string "Enter your password: "
.LC1:    .string "Incorrect. Try again: "
.LC2:    .string "123456"
.LC3:    .string "You are now logged in"
authenticate:
         push    rbx
         sub     rsp, 16
         mov     rbx, rdi
         lea     rdi, [rsp+12]
         mov     eax, 0
         call    gets
         lea     rsi, [rsp+12]
         mov     rdi, rbx
         call    strcmp
         test    eax, eax
         sete    al
         movzx   eax, al
         add     rsp, 16
         pop     rbx
         ret
main:
         sub     rsp, 8
         mov     edi, OFFSET FLAT:.LC0
         mov     eax, 0
         call    printf
         jmp     .L4
.L5:
         mov     edi, OFFSET FLAT:.LC1
         mov     eax, 0
         call    printf
.L4:
         mov     edi, OFFSET FLAT:.LC2
         call    authenticate
         test    eax, eax
         je      .L5
         mov     edi, OFFSET FLAT:.LC3
         call    puts
         mov     eax, 0
         add     rsp, 8
         ret
```

Practice with Buffer Overflow Exploits

Construct an exploit string that will successfully cause the program to print "You are now logged in" without knowing the correct password

1. How many bytes of padding are in this exploit string?

2. What value will you overwrite the return address with?

```
int authenticat
  char buf[4];
  gets(buf);
  int correct =
  return correc
}


int main(int ar
  char * pw = "
  printf("Enter
  while(!authen
    printf("Inc
  }
  printf("You a
  return 0;
}
```

```
.LC0:    .string "Enter your password: "
.LC1:    .string "Incorrect. Try again: "
.LC2:    .string "123456"
.LC3:    .string "You are now logged in"
authenticate:
         push    rbx
         sub     rsp, 16
         mov     rbx, rdi
         lea     rdi, [rsp+12]
         mov     eax, 0
         call    gets
         lea     rsi, [rsp+12]
         mov     rdi, rbx
         call    strcmp
         test    eax, eax
         sete    al
         movzx   eax, al
         add     rsp, 16
         pop     rbx
         ret
main:
401188   sub     rsp, 8
40118C   mov     edi, OFFSET FLAT:.LC0
401191   mov     eax, 0
401196   call    printf
40119b   jmp     .L4
.L5:
40119d   mov     edi, OFFSET FLAT:.LC1
4011a2   mov     eax, 0
4011a7   call    printf
.L4:
4011ac   mov     edi, OFFSET FLAT:.LC2
4011b1   call    authenticate
4011b6   test    eax, eax
4011b8   je      .L5
4011ba   mov     edi, OFFSET FLAT:.LC3
4011bf   call    puts
4011c4   mov     eax, 0
4011c9   add     rsp, 8
4011cd   ret
```

Practice with Buffer Overflow Exploits

Construct an exploit string that will successfully cause the program to print "You are now logged in" without knowing the correct password

1. How many bytes of padding are in this exploit string?
2. What value will you overwrite the return address with?

```
int authenticat
  char buf[4];
  gets(buf);
  int correct =
  return correc
}


int main(int ar
  char * pw = "
  printf("Enter
  while(!authen
    printf("Inc
  }
  printf("You a
  return 0;
}
```

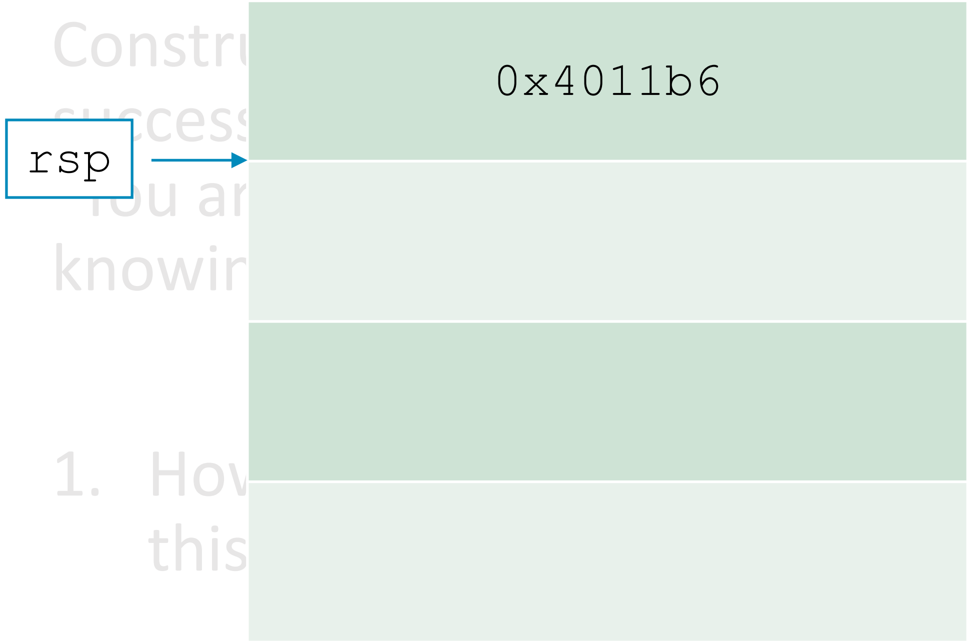
```
.LC0:    .string "Enter your password: "
.LC1:    .string "Incorrect. Try again: "
.LC2:    .string "123456"
.LC3:    .string "You are now logged in"
authenticate:
         push    rbx
         sub     rsp, 16
         mov     rbx, rdi
         lea     rdi, [rsp+12]
         mov     eax, 0
         call    gets
         lea     rsi, [rsp+12]
         mov     rdi, rbx
         call    strcmp
         test    eax, eax
         sete    al
         movzx   eax, al
         add     rsp, 16
         pop     rbx
         ret
main:
401188   sub     rsp, 8
40118C   mov     edi, OFFSET FLAT:.LC0
401191   mov     eax, 0
401196   call    printf
40119b   jmp     .L4
.L5:
40119d   mov     edi, OFFSET FLAT:.LC1
4011a2   mov     eax, 0
4011a7   call    printf
.L4:
4011ac   mov     edi, OFFSET FLAT:.LC2
4011b1   call    authenticate
4011b6   test    eax, eax
4011b8   je      .L5
4011ba   mov     edi, OFFSET FLAT:.LC3
4011bf   call    puts
4011c4   mov     eax, 0
4011c9   add     rsp, 8
4011cd   ret
```

Practice with Buffer Overflow Exploits

PUSH RBX

Constru… will successf… o print You ar… knowin…

1. How… are in this…
2. What value will you overwrite the return address with?

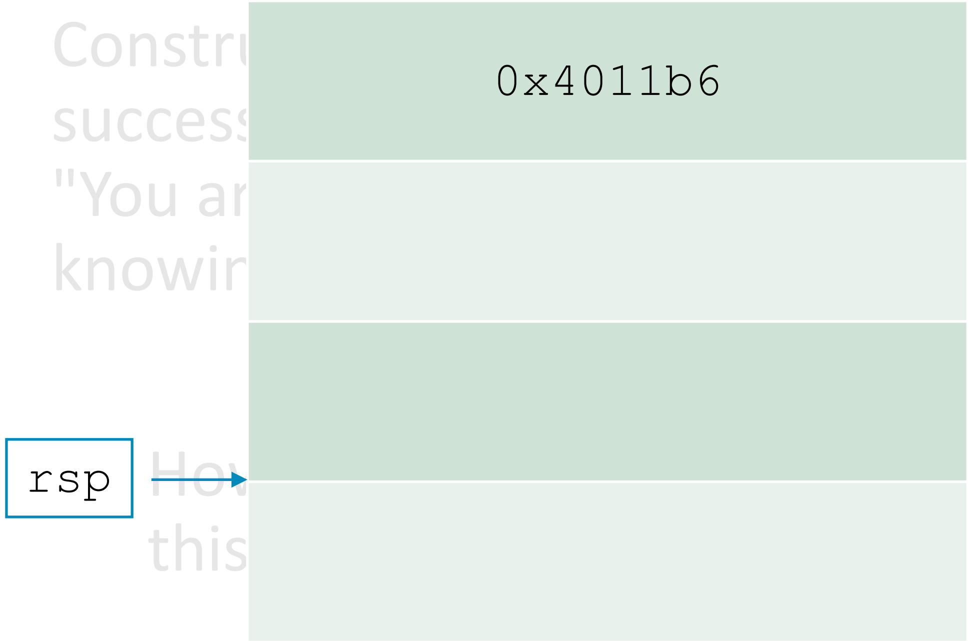



```
int authenticat
  char buf[4];
  gets(buf);
  int correct =
  return correc
}

int main(int ar
  char * pw = "
  printf("Enter
  while(!authen
    printf("Inc
  }
  printf("You a
  return 0;
}
```

```
.LC0:   .string "Enter your password: "
.LC1:   .string "Incorrect. Try again: "
.LC2:   .string "123456"
.LC3:   .string "You are now logged in"
authenticate:
        push    rbx
        sub     rsp, 16
        mov     rbx, rdi
        lea     rdi, [rsp+12]
        mov     eax, 0
        call    gets
        lea     rsi, [rsp+12]
        mov     rdi, rbx
        call    strcmp
        test    eax, eax
        sete    al
        movzx   eax, al
        add     rsp, 16
        pop     rbx
        ret
main:
401188  sub     rsp, 8
40118c  mov     edi, OFFSET FLAT:.LC0
401191  mov     eax, 0
401196  call    printf
40119b  jmp     .L4
.L5:
40119d  mov     edi, OFFSET FLAT:.LC1
4011a2  mov     eax, 0
4011a7  call    printf
.L4:
4011ac  mov     edi, OFFSET FLAT:.LC2
4011b1  call    authenticate
4011b6  test    eax, eax
4011b8  je      .L5
4011ba  mov     edi, OFFSET FLAT:.LC3
4011bf  call    puts
4011c4  mov     eax, 0
4011c9  add     rsp, 8
4011cd  ret
```

Practice with Buffer Overflow Exploits

Constr... will success... o print "You ar... knowi...



How... re in this...

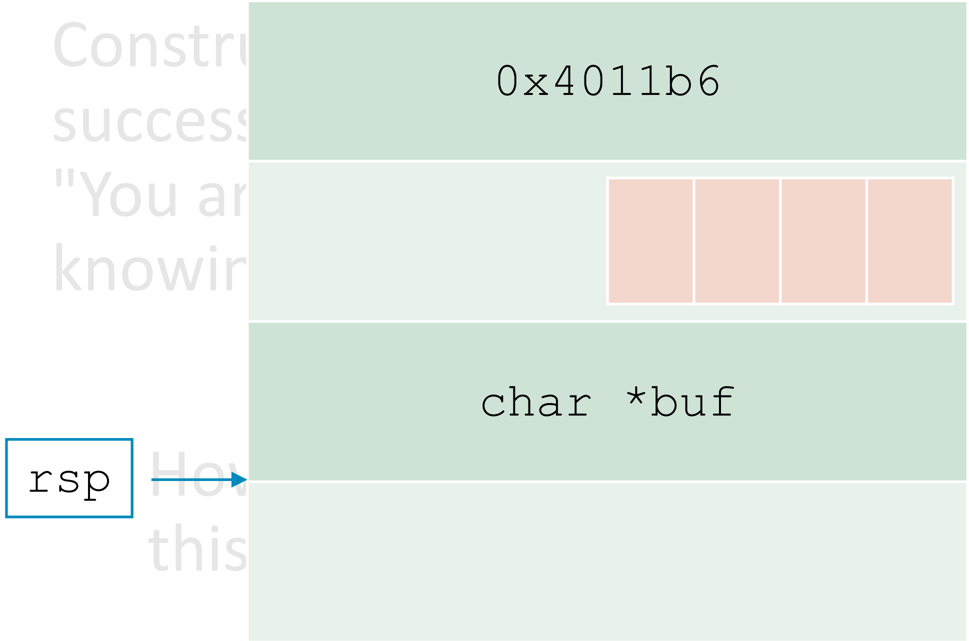
2. What value will you overwrite the return address with?

```
int authenticat
  char buf[4];
  gets(buf);
  int correct =
  return correc
}

int main(int ar
  char * pw = "
  printf("Enter
  while(!authen
    printf("Inc
  }
  printf("You a
  return 0;
}
```

```
.LC0:   .string "Enter your password: "
.LC1:   .string "Incorrect. Try again: "
.LC2:   .string "123456"
.LC3:   .string "You are now logged in"
authenticate:
        push    rbx
        sub     rsp, 16
        mov     rbx, rdi
        lea     rdi, [rsp+12]
        mov     eax, 0
        call    gets
        lea     rsi, [rsp+12]
        mov     rdi, rbx
        call    strcmp
        test    eax, eax
        sete    al
        movzx   eax, al
        add     rsp, 16
        pop     rbx
        ret
main:
401188  sub     rsp, 8
40118c  mov     edi, OFFSET FLAT:.LC0
401191  mov     eax, 0
401196  call    printf
40119b  jmp     .L4
.L5:
40119d  mov     edi, OFFSET FLAT:.LC1
4011a2  mov     eax, 0
4011a7  call    printf
.L4:
4011ac  mov     edi, OFFSET FLAT:.LC2
4011b1  call    authenticate
4011b6  test    eax, eax
4011b8  je      .L5
4011ba  mov     edi, OFFSET FLAT:.LC3
4011bf  call    puts
4011c4  mov     eax, 0
4011c9  add     rsp, 8
4011cd  ret
```

Practice with Buffer Overflow Exploits

Constr... will
success... o print
"You ar...
knowin...



How... re in
this...

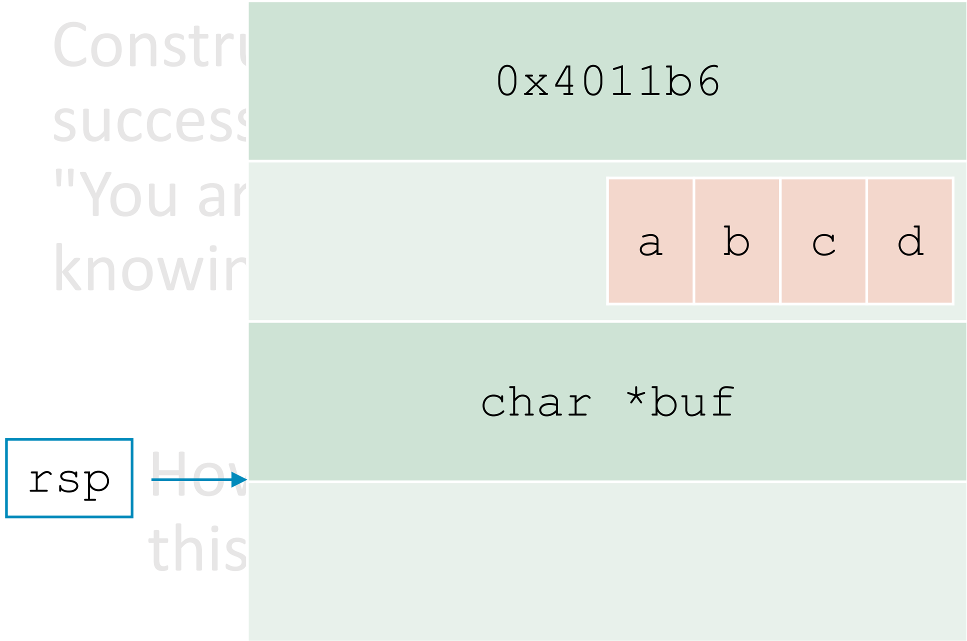
2. What value will you overwrite the return address with?

```
int authenticat
  char buf[4];
  gets(buf);
  int correct =
  return correc
}

int main(int ar
  char * pw = "
  printf("Enter
  while(!authen
    printf("Inc
  }
  printf("You a
  return 0;
}
```

```
.LC0:    .string "Enter your password: "
.LC1:    .string "Incorrect. Try again: "
.LC2:    .string "123456"
.LC3:    .string "You are now logged in"
authenticate:
         push    rbx
         sub     rsp, 16
         mov     rbx, rdi
         lea     rdi, [rsp+12]
         mov     eax, 0
         call    gets
         lea     rsi, [rsp+12]
         mov     rdi, rbx
         call    strcmp
         test    eax, eax
         sete    al
         movzx   eax, al
         add     rsp, 16
         pop     rbx
         ret
main:
401188   sub     rsp, 8
40118c   mov     edi, OFFSET FLAT:.LC0
401191   mov     eax, 0
401196   call    printf
40119b   jmp     .L4
.L5:
40119d   mov     edi, OFFSET FLAT:.LC1
4011a2   mov     eax, 0
4011a7   call    printf
.L4:
4011ac   mov     edi, OFFSET FLAT:.LC2
4011b1   call    authenticate
4011b6   test    eax, eax
4011b8   je      .L5
4011ba   mov     edi, OFFSET FLAT:.LC3
4011bf   call    puts
4011c4   mov     eax, 0
4011c9   add     rsp, 8
4011cd   ret
```

Practice with Buffer Overflow Exploits

Constr... will success... o print "You ar... knowir...



How ... re in this

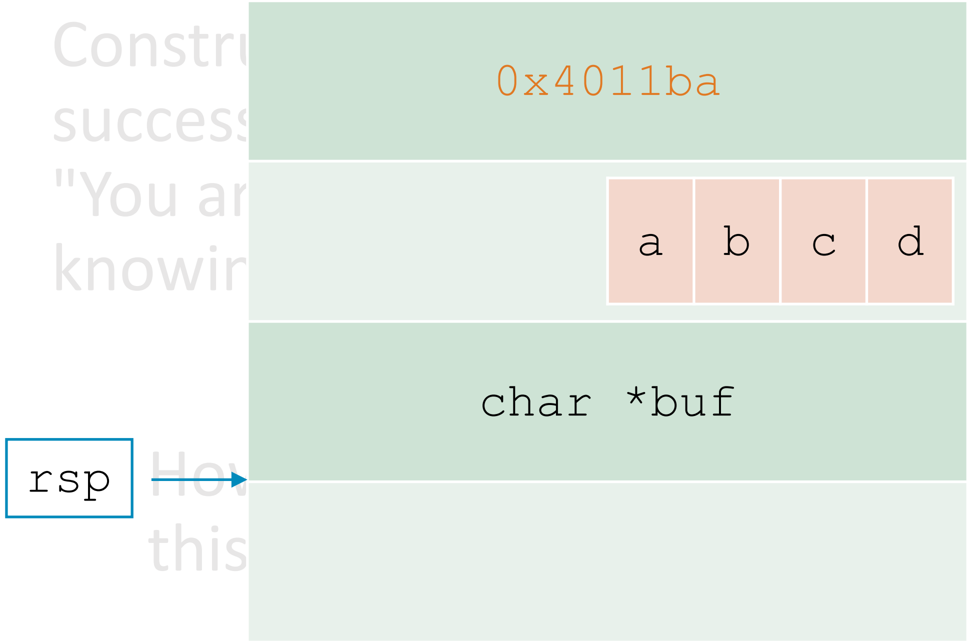
2. What value will you overwrite the return address with?

```
int authenticat
  char buf[4];
  gets(buf);
  int correct =
  return correc
}


int main(int ar
  char * pw = "
  printf("Enter
  while(!authen
    printf("Inc
  }
  printf("You a
  return 0;
}
```

```
.LC0:   .string "Enter your password: "
.LC1:   .string "Incorrect. Try again: "
.LC2:   .string "123456"
.LC3:   .string "You are now logged in"
authenticate:
        push    rbx
        sub     rsp, 16
        mov     rbx, rdi
        lea     rdi, [rsp+12]
        mov     eax, 0
        call    gets
        lea     rsi, [rsp+12]
        mov     rdi, rbx
        call    strcmp
        test    eax, eax
        sete    al
        movzx   eax, al
        add     rsp, 16
        pop     rbx
        ret
main:
401188  sub     rsp, 8
40118c  mov     edi, OFFSET FLAT:.LC0
401191  mov     eax, 0
401196  call    printf
40119b  jmp     .L4
.L5:
40119d  mov     edi, OFFSET FLAT:.LC1
4011a2  mov     eax, 0
4011a7  call    printf
.L4:
4011ac  mov     edi, OFFSET FLAT:.LC2
4011b1  call    authenticate
4011b6  test    eax, eax
4011b8  je      .L5
4011ba  mov     edi, OFFSET FLAT:.LC3
4011bf  call    puts
4011c4  mov     eax, 0
4011c9  add     rsp, 8
4011cd  ret
```

Practice with Buffer Overflow Exploits

Constru… will success… o print "You ar… knowir…



How… are in this…

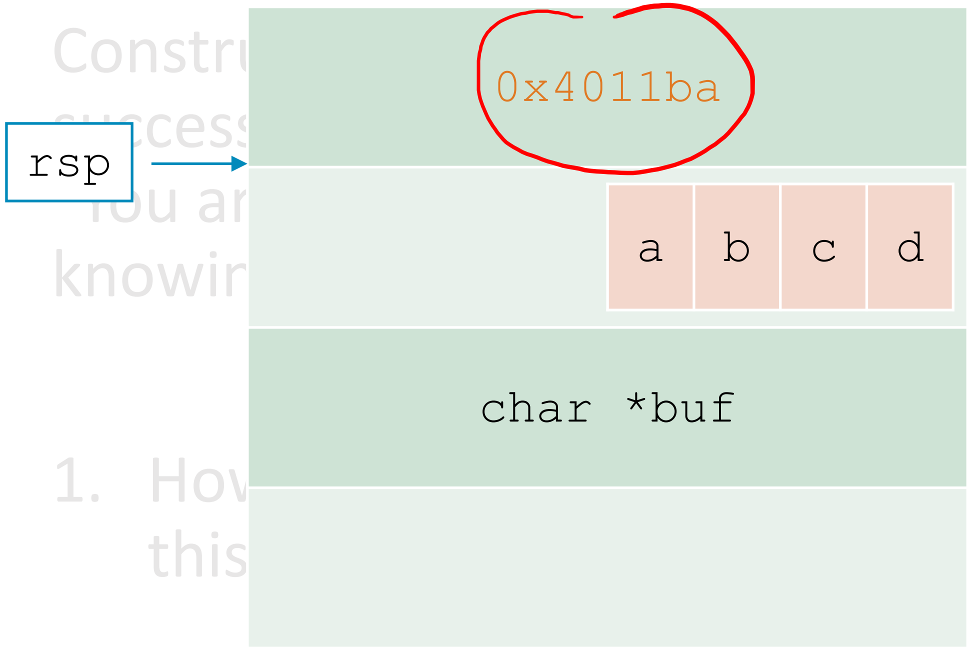
2. What value will you overwrite the return address with?

```
int authenticat
  char buf[4];
  gets(buf);
  int correct =
  return correc
}


int main(int ar
  char * pw = "
  printf("Enter
  while(!authen
    printf("Inc
  }
  printf("You a
  return 0;
}
```

```
.LC0:    .string "Enter your password: "
.LC1:    .string "Incorrect. Try again: "
.LC2:    .string "123456"
.LC3:    .string "You are now logged in"
authenticate:
         push    rbx
         sub     rsp, 16
         mov     rbx, rdi
         lea     rdi, [rsp+12]
         mov     eax, 0
         call    gets
         lea     rsi, [rsp+12]
         mov     rdi, rbx
         call    strcmp
         test    eax, eax
         sete    al
         movzx   eax, al
         add     rsp, 16
         pop     rbx
         ret
main:
401188   sub     rsp, 8
40118c   mov     edi, OFFSET FLAT:.LC0
401191   mov     eax, 0
401196   call    printf
40119b   jmp     .L4
.L5:
40119d   mov     edi, OFFSET FLAT:.LC1
4011a2   mov     eax, 0
4011a7   call    printf
.L4:
4011ac   mov     edi, OFFSET FLAT:.LC2
4011b1   call    authenticate
4011b6   test    eax, eax
4011b8   je      .L5
4011ba   mov     edi, OFFSET FLAT:.LC3
4011bf   call    puts
4011c4   mov     eax, 0
4011c9   add     rsp, 8
4011cd   ret
```

Practice with Buffer Overflow Exploits

Constru... will ... success... o print ... You ar... knowin...



1. How ... are in this ...

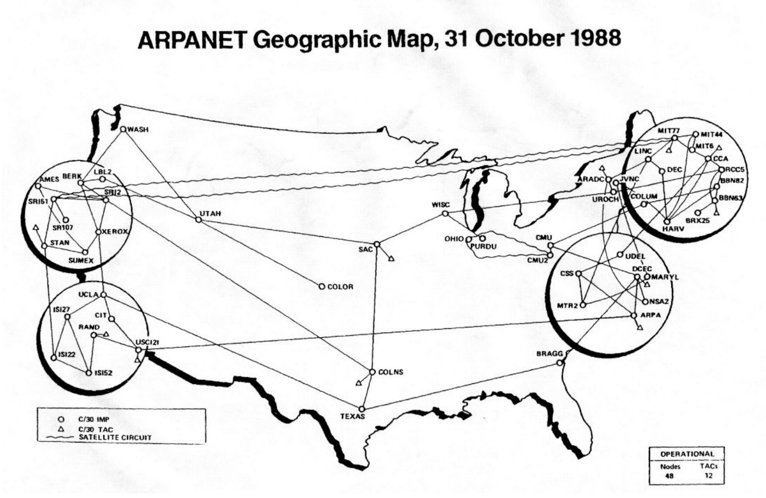
2. What value will you overwrite the return address with?

```
int authenticat
   char buf[4];
   gets(buf);
   int correct =
   return correc
}


int main(int ar
   char * pw = "
   printf("Enter
   while(!authen
      printf("Inc
   }
   printf("You a
   return 0;
}
```

```
.LC0:    .string "Enter your password: "
.LC1:    .string "Incorrect. Try again: "
.LC2:    .string "123456"
.LC3:    .string "You are now logged in"
authenticate:
         push    rbx
         sub     rsp, 16
         mov     rbx, rdi
         lea     rdi, [rsp+12]
         mov     eax, 0
         call    gets
         lea     rsi, [rsp+12]
         mov     rdi, rbx
         call    strcmp
         test    eax, eax
         sete    al
         movzx   eax, al
         add     rsp, 16
         pop     rbx
         ret
main:
401188   sub     rsp, 8
40118c   mov     edi, OFFSET FLAT:.LC0
401191   mov     eax, 0
401196   call    printf
40119b   jmp     .L4
.L5:
40119d   mov     edi, OFFSET FLAT:.LC1
4011a2   mov     eax, 0
4011a7   call    printf
.L4:
4011ac   mov     edi, OFFSET FLAT:.LC2
4011b1   call    authenticate
4011b6   test    eax, eax
4011b8   je      .L5
4011ba   mov     edi, OFFSET FLAT:.LC3
4011bf   call    puts
4011c4   mov     eax, 0
4011c9   add     rsp, 8
4011cd   ret
```

Heartbleed (2014): bug in openssl library hearbeat

Buffer Overflow Examples

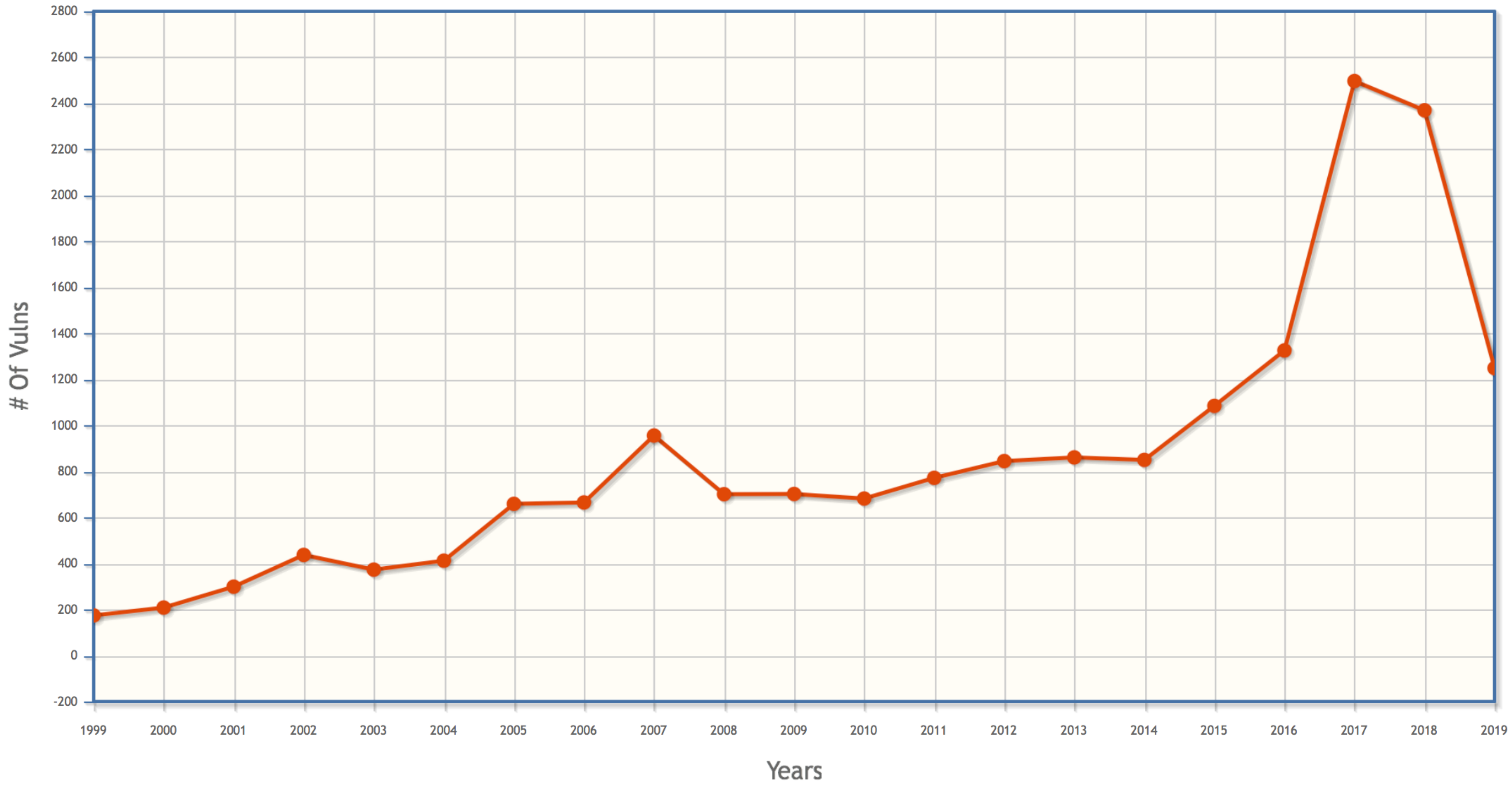


Morris Worm (November 2, 1988), $100,000-$10mil, bugs in finger and sendmail

WhatsApp (2019)

Stuxnet (discovered 2010): bug in functions that process files to display icons when USB connected to PC

Buffer Overflow Vulnerabilities



https://www.cvedetails.com/vulnerabilities-by-types.php

Linus Torvalds: Rust will go into Linux 6.1

Sept. 19, 2022

Defense #1: Avoid Overflow Vulnerabilities

```c
// Echo Line
void echo() {
    char buf[4];  // Way too small!
    fgets(buf, 4, stdin);
    puts(buf);
}
```

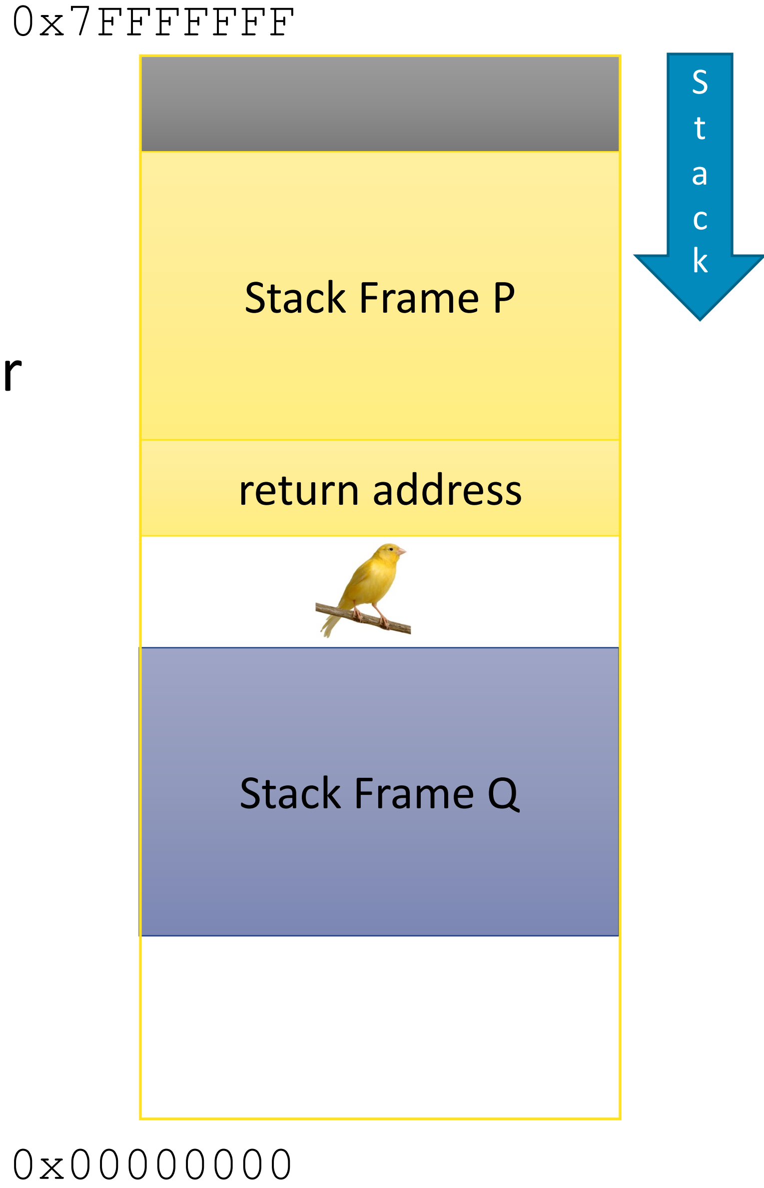
- For example, use library routines that limit string lengths
  - **`fgets`** instead of **`gets`**
  - Don't use **`scanf`** with **`%s`** conversion specification (use **`fgets`** to read the string or use **`%ns`** where **`n`** is a suitable integer)

https://github.com/leafsr/gcc-poison

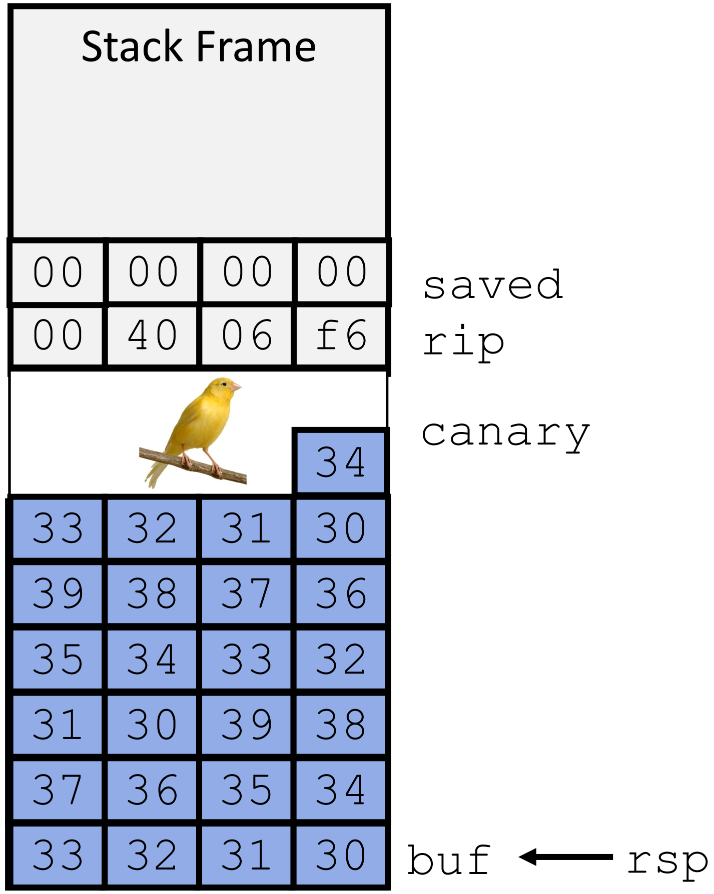
- Or use a better high-level language

Defense #2: Compiler checks

- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - -fstack-protector
  - Now the default (disabled earlier)



Stack Canaries

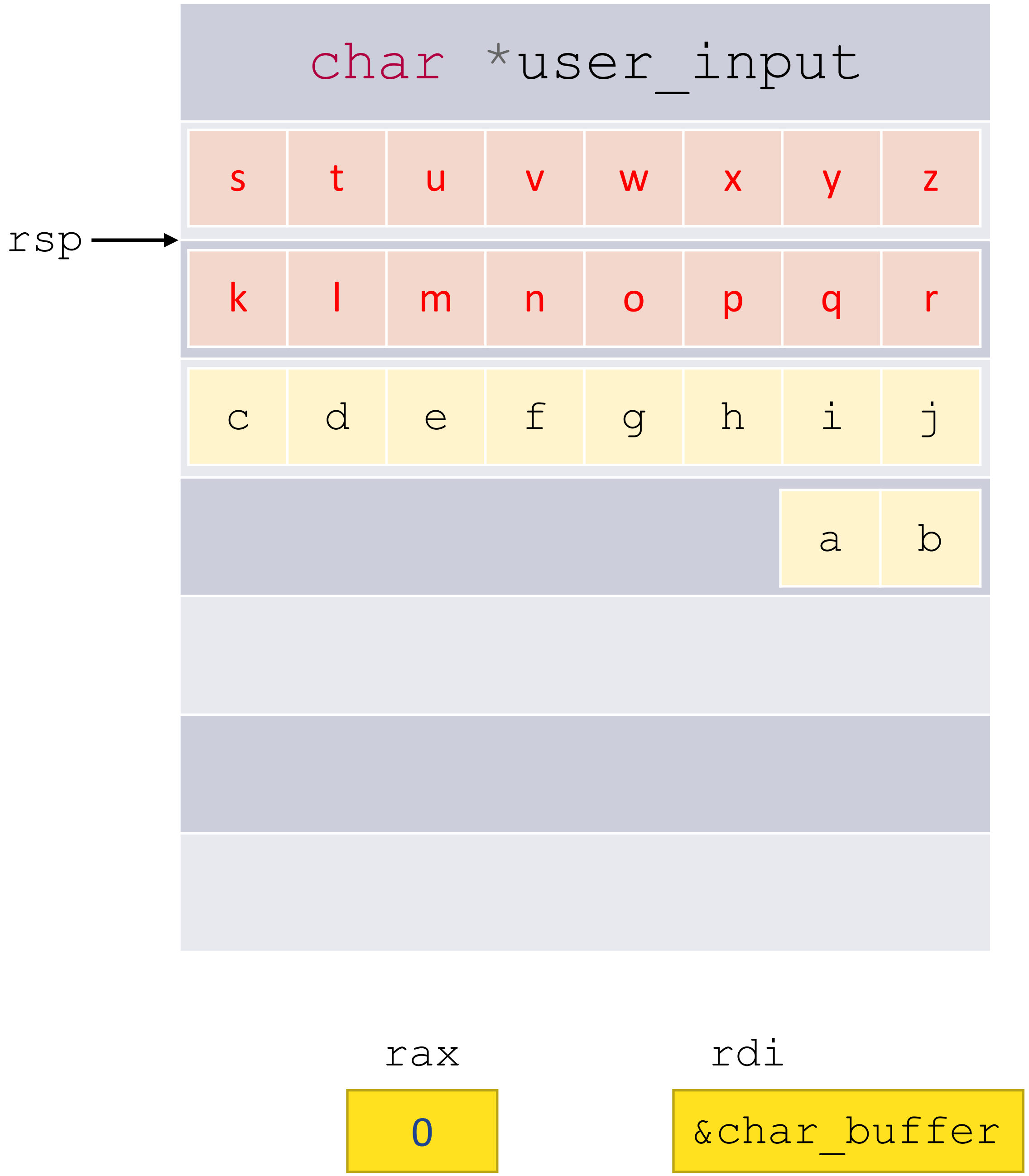



```
authenticate:
  push    rbx
  sub     rsp, 16
  mov     rbx, rdi
  mov     rax, fs:40
  mov     [rsp+8], rax
  xor     eax, eax
  mov     rdi, rsp
  call    gets
  mov     rsi, rsp
  mov     rdi, rbx
  call    strcmp
  test    eax, eax
  sete    al
  mov     rdx, [rsp+8]
  xor     rdx, fs:40
  je      .L2
  call    __stack_chk_fail
.L2:
  movzb   eax, al
  add     rsp, 16
  pop     rbx
  ret
```

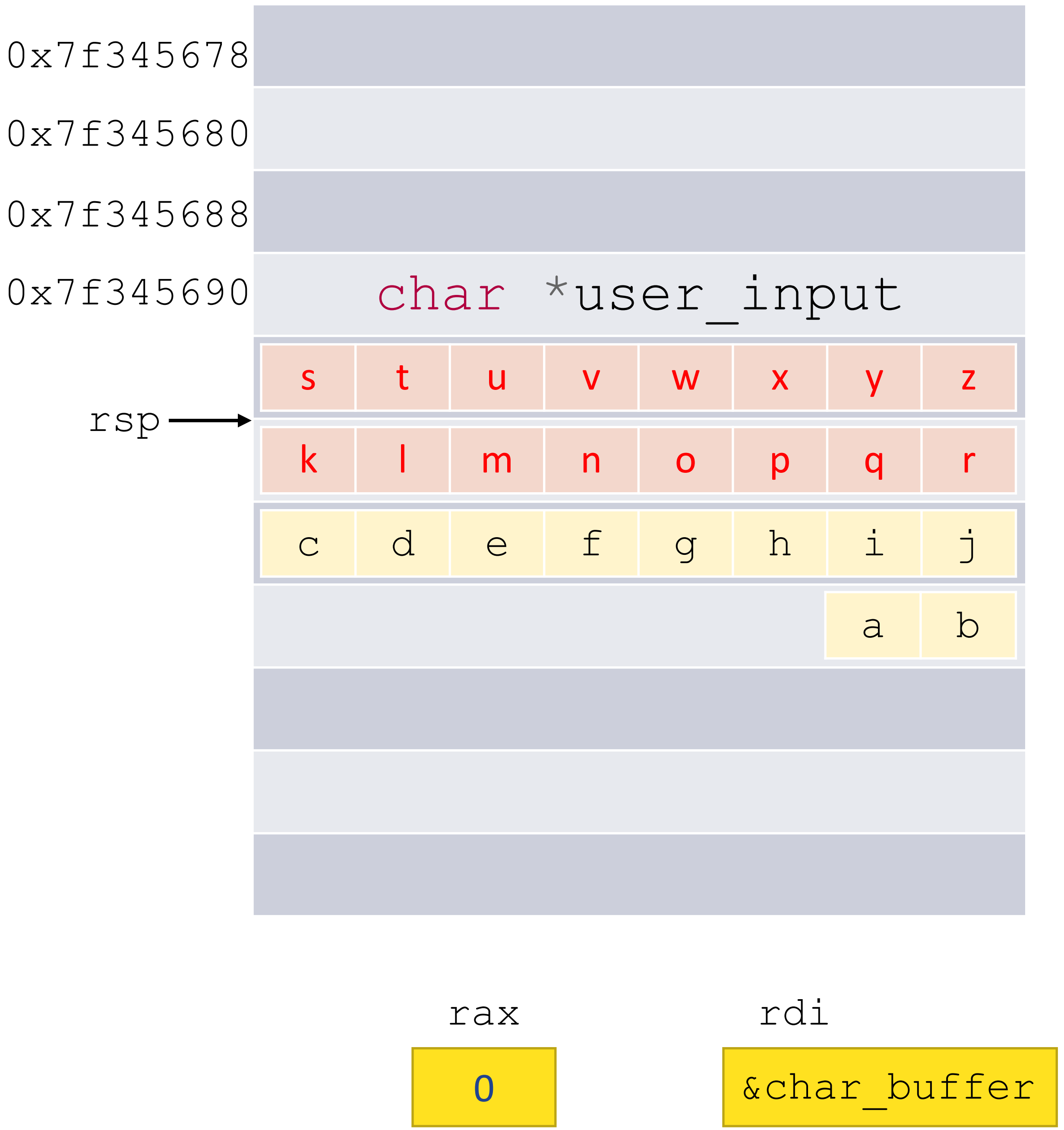
Stack Canaries

Which of the following would make a good stack canary?

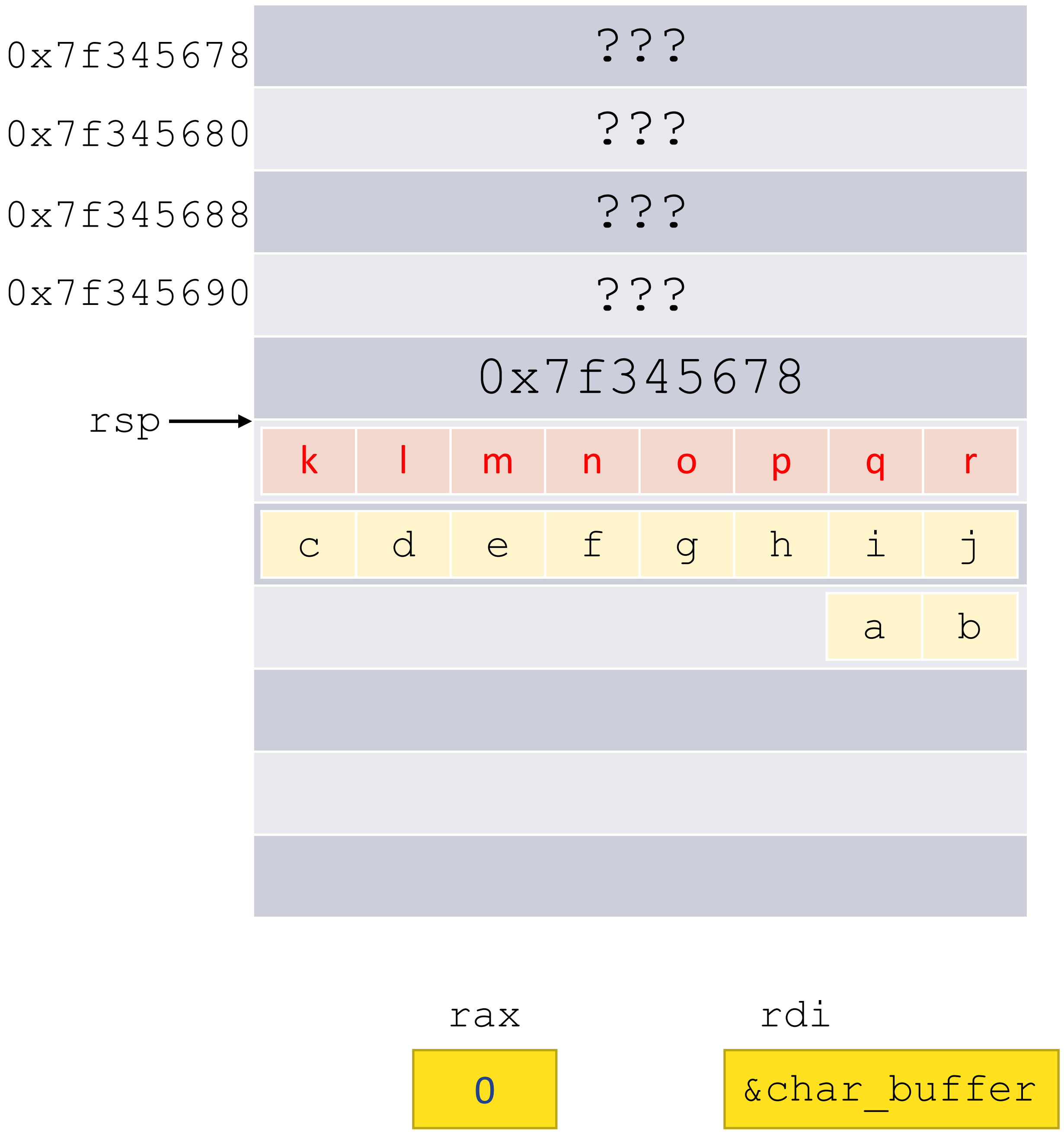
1. A secret, constant value
2. A fixed sequence of common terminators (\0, EOF, etc.)
3. A random number chosen each time the program is run



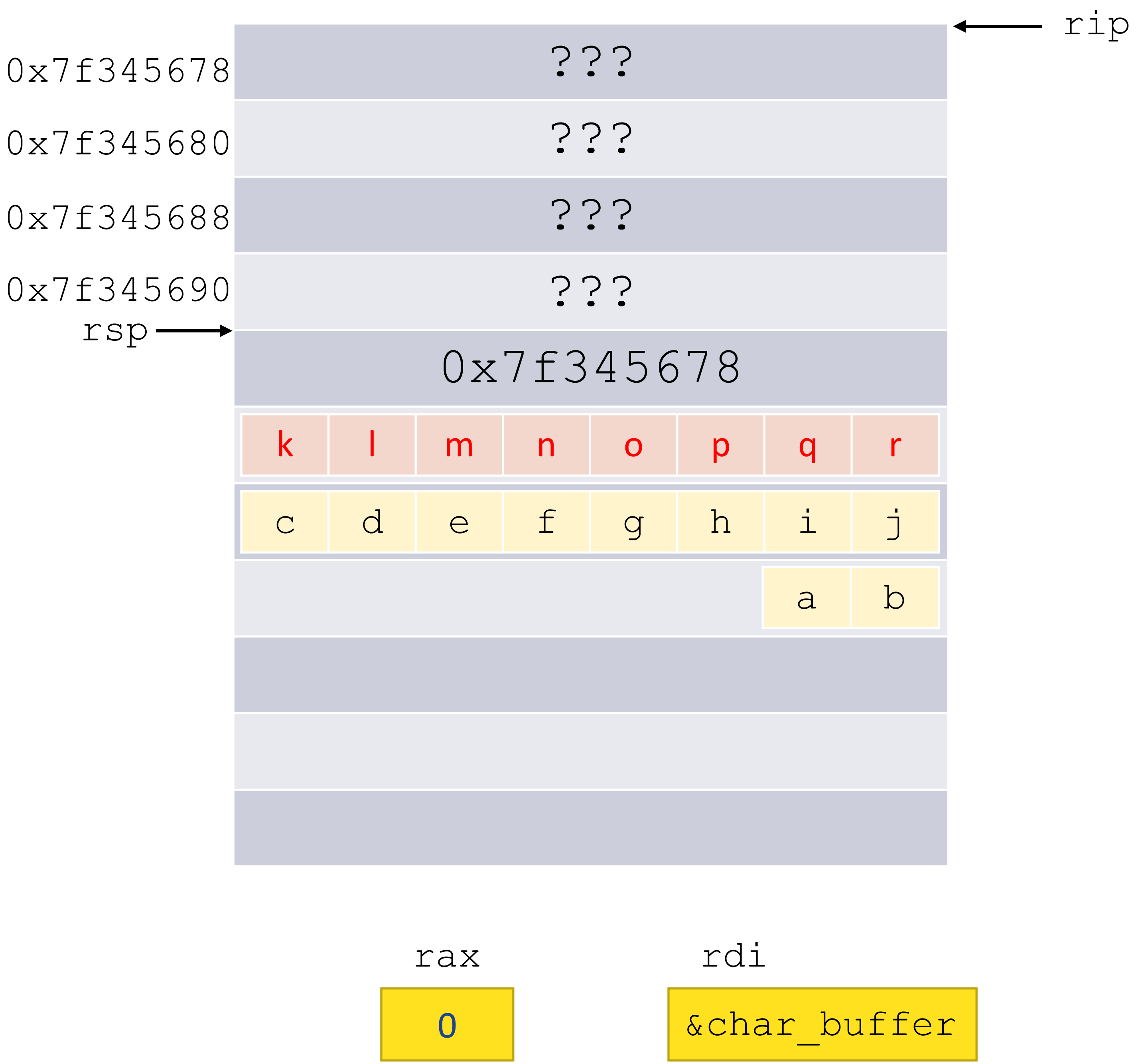

```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
rip --> ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```



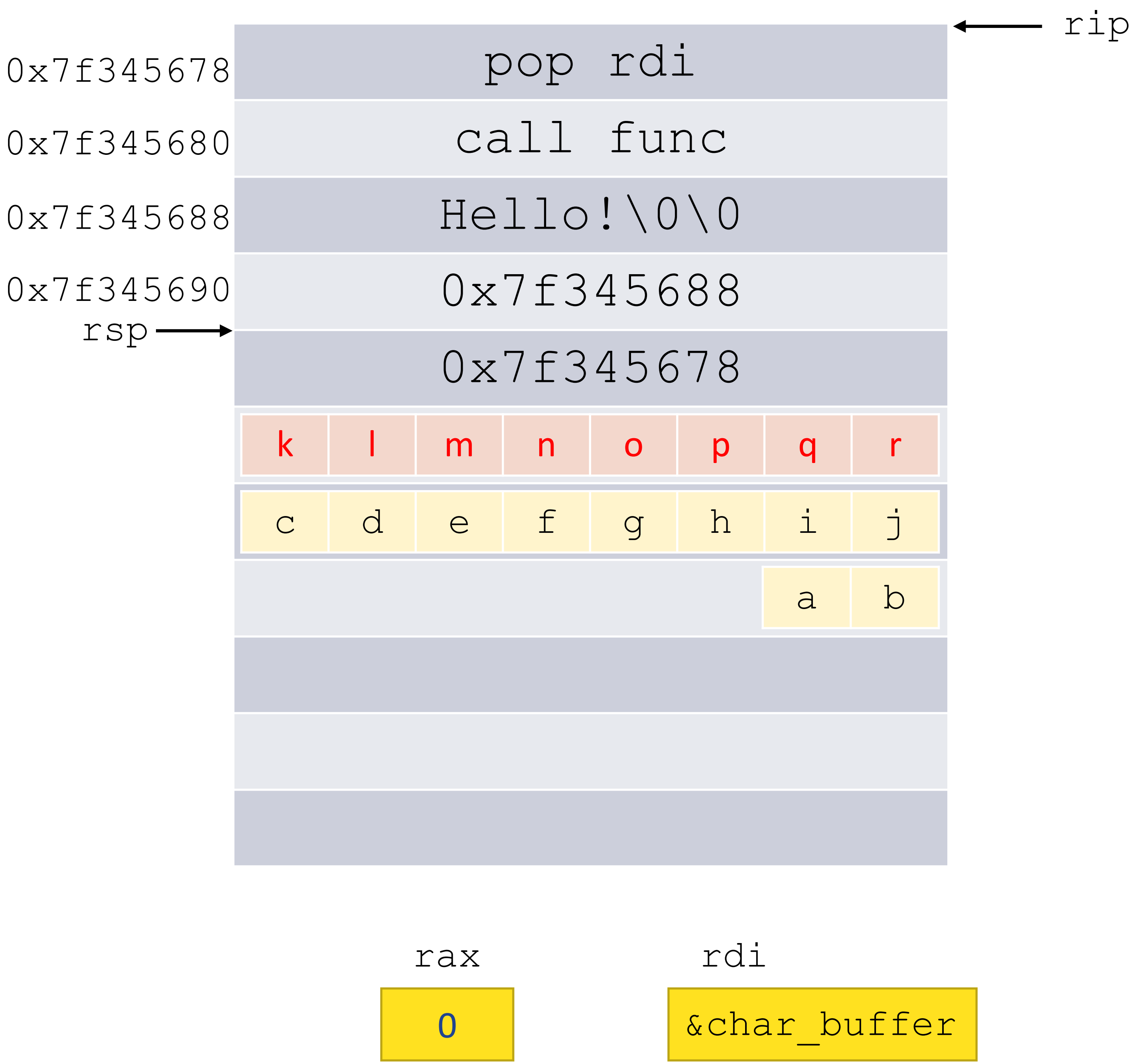

```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
rip --> ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```




```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
rip --> ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```

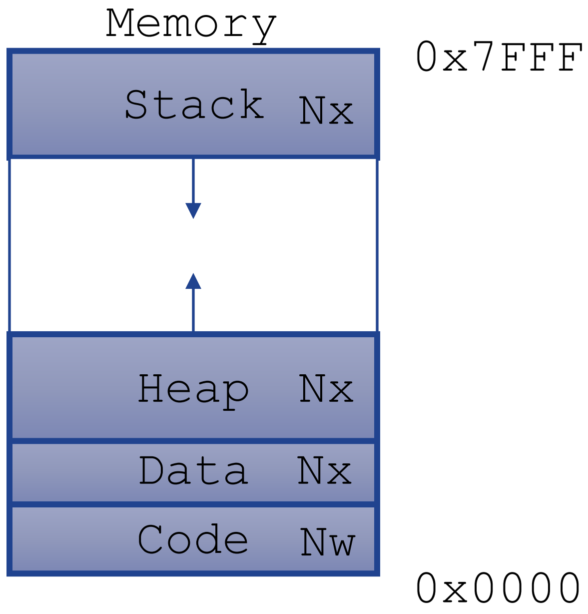



```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```




```
read_string:
        sub     rsp, 24
        lea     rdi, [rsp+6]
        mov     eax, 0
        call    gets
        mov     eax, 0
        add     rsp, 24
        ret

print_string:
        sub     rsp, 8
        call    puts
        add     rsp, 8
        ret

main:
        sub     rsp, 8
        mov     eax, 0
        call    read_string
        movsx   rdi, eax
        mov     eax, 0
        call    print_string
        mov     eax, 0
        add     rsp, 8
        ret
```

Defense #3: Memory Tagging



Code Reuse Attacks

- Key idea: execute instructions that already exist
- Defeats memory tagging defenses
- Examples:
  1. return to a function or line in the current program
  2. return to a library function (e.g., return-into-`libc`)
  3. return to some other instruction (return-oriented programming)

Return-into-libc

Sr.No.	Function & Description
1	**double atof(const char *str)** Converts the string pointed to, by the argument *str* to a floating-point number (type double).
2	**int atoi(const char *str)** Converts the string pointed to, by the argument *str* to an integer (type int).
3	**long int atol(const char *str)** Converts the string pointed to, by the argument *str* to a long integer (type long int).
8	**void free(void *ptr** Deallocates the memory previously allocated by a call to *calloc, malloc,* or *realloc*.
9	**void *malloc(size_t size)** Allocates the requested memory and returns a pointer to it.
10	**void *realloc(void *ptr, size_t size)** Attempts to resize the memory block pointed to by ptr that was previously allocated with a call to *malloc* or *calloc*.
15	**int system(const char *string)** The command specified by string is passed to the host environment to be executed by the command processor.
16	**void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *))** Performs a binary search.
17	**void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))** Sorts an array.
18	**int abs(int x)** Returns the absolute value of x.
22	**int rand(void)** Returns a pseudo-random number in the range of 0 to *RAND_MAX*.
23	**void srand(unsigned int seed)** This function seeds the random number generator used by the function **rand**.

Defense #4: ASCII Armoring

- Make sure all system library addresses contain a null byte (0x00).

- Can be done by placing this code in the first 0x01010101 bytes of memory

Properties of x86-64 Assembly

- Lots of instructions
- Variable length instructions
- Not word aligned
- Dense instruction set → most bytes encode an actual instruction

Gadgets

```
void setval(unsigned *p)
{
  *p = 3347663060u; // 0xC78948D4
}
```

```
<setval>:
4004d9: c7 07 d4 48 89 c7   mov [rdi], 0xC78948D4
4004df: c3                  ret
```

<u>gadget address</u>: `0x4004dc`

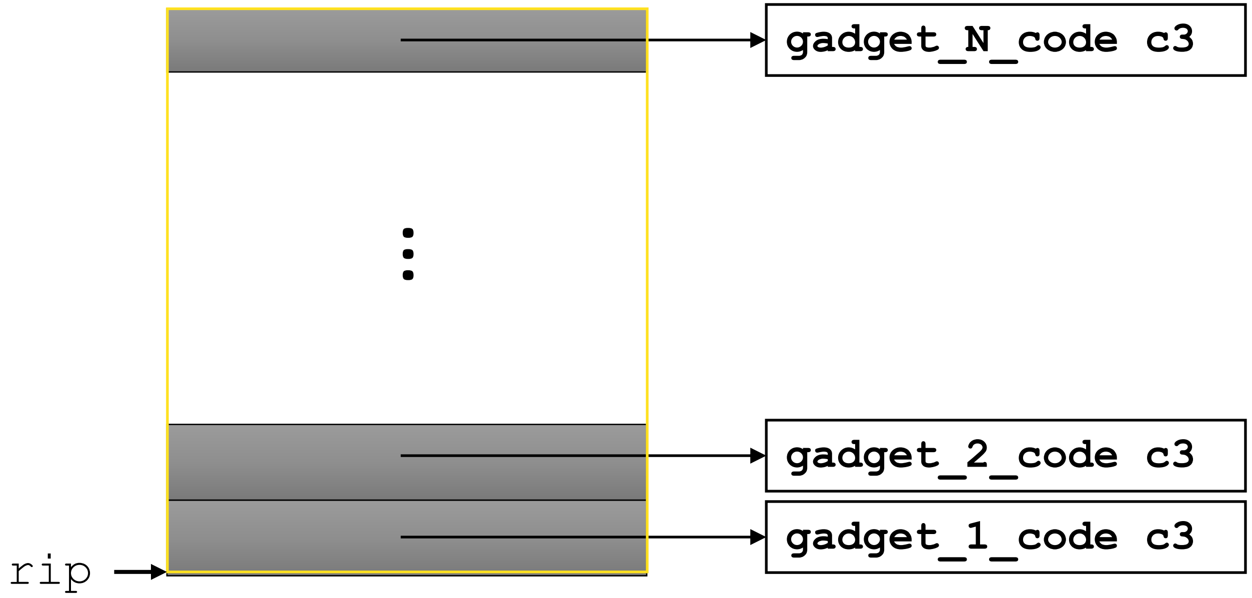
<u>encodes</u>:

```
mov rdi, rax
ret
```

Return-oriented Programming

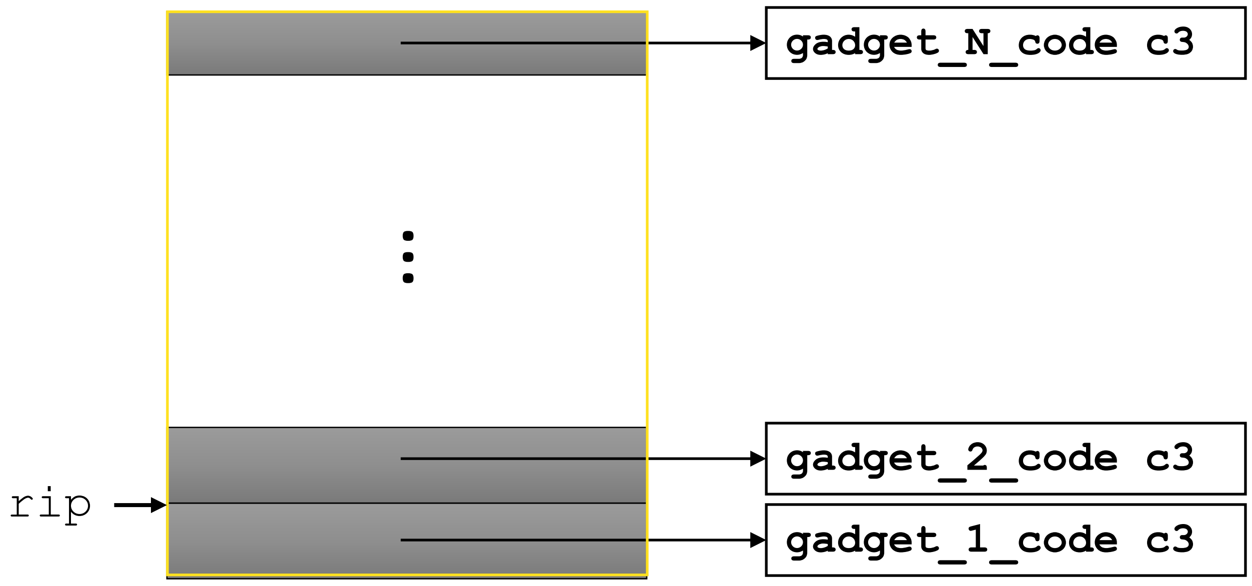
Image By: Dino Dai Zovi

Return-oriented Programming



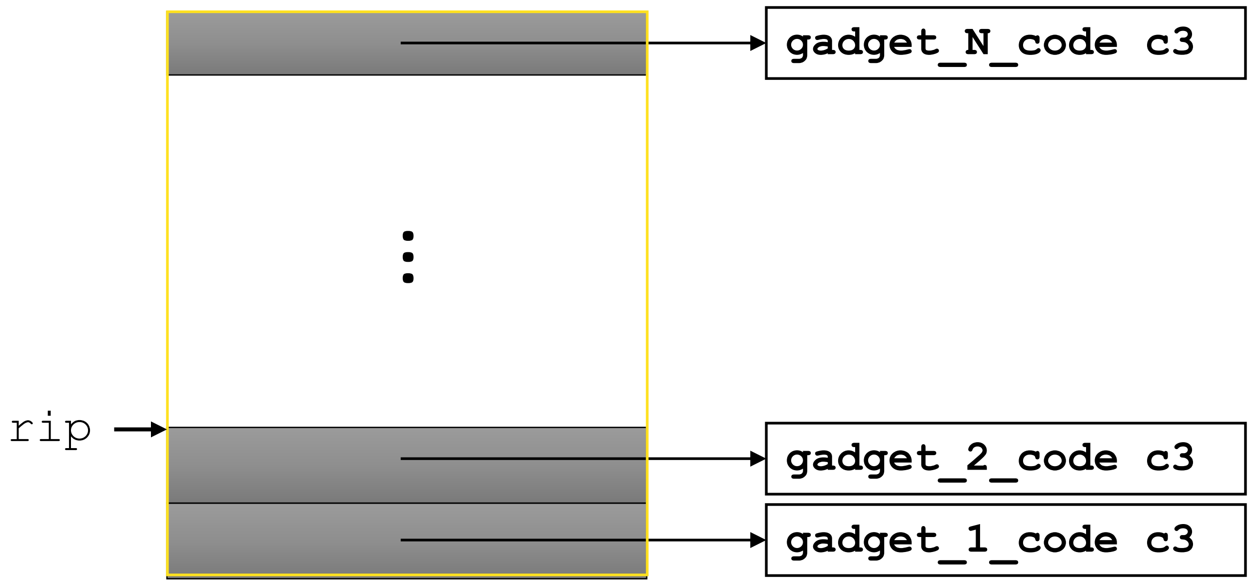
Final ret in each gadget sets pc (`rip`) to beginning of next gadget code

Return-oriented Programming



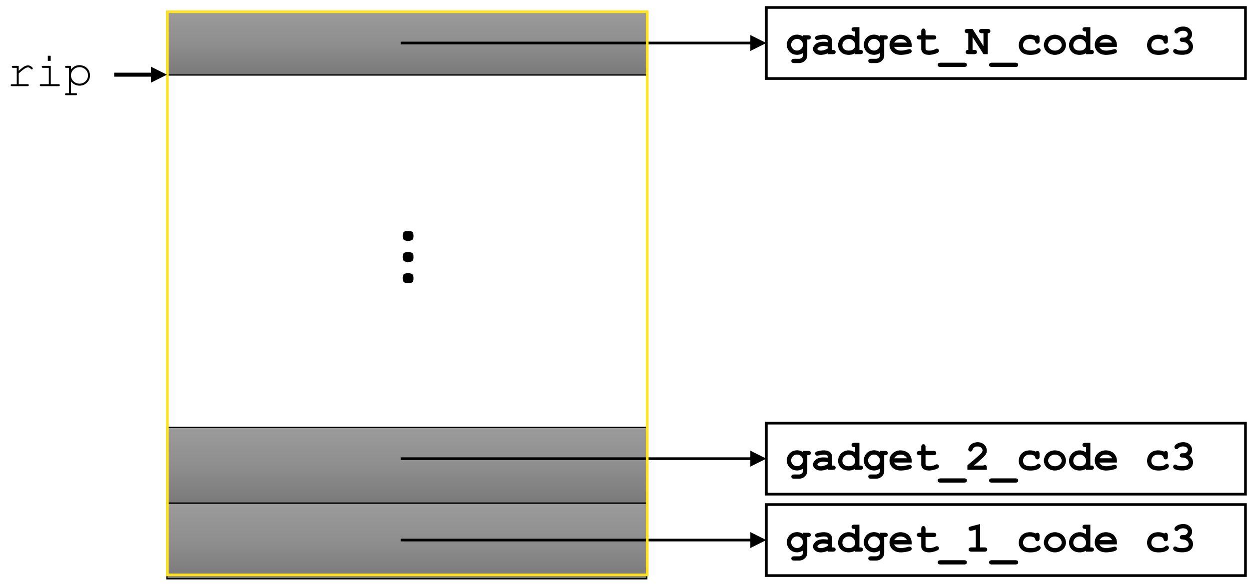
Final ret in each gadget sets pc (`rip`) to beginning of next gadget code

Return-oriented Programming

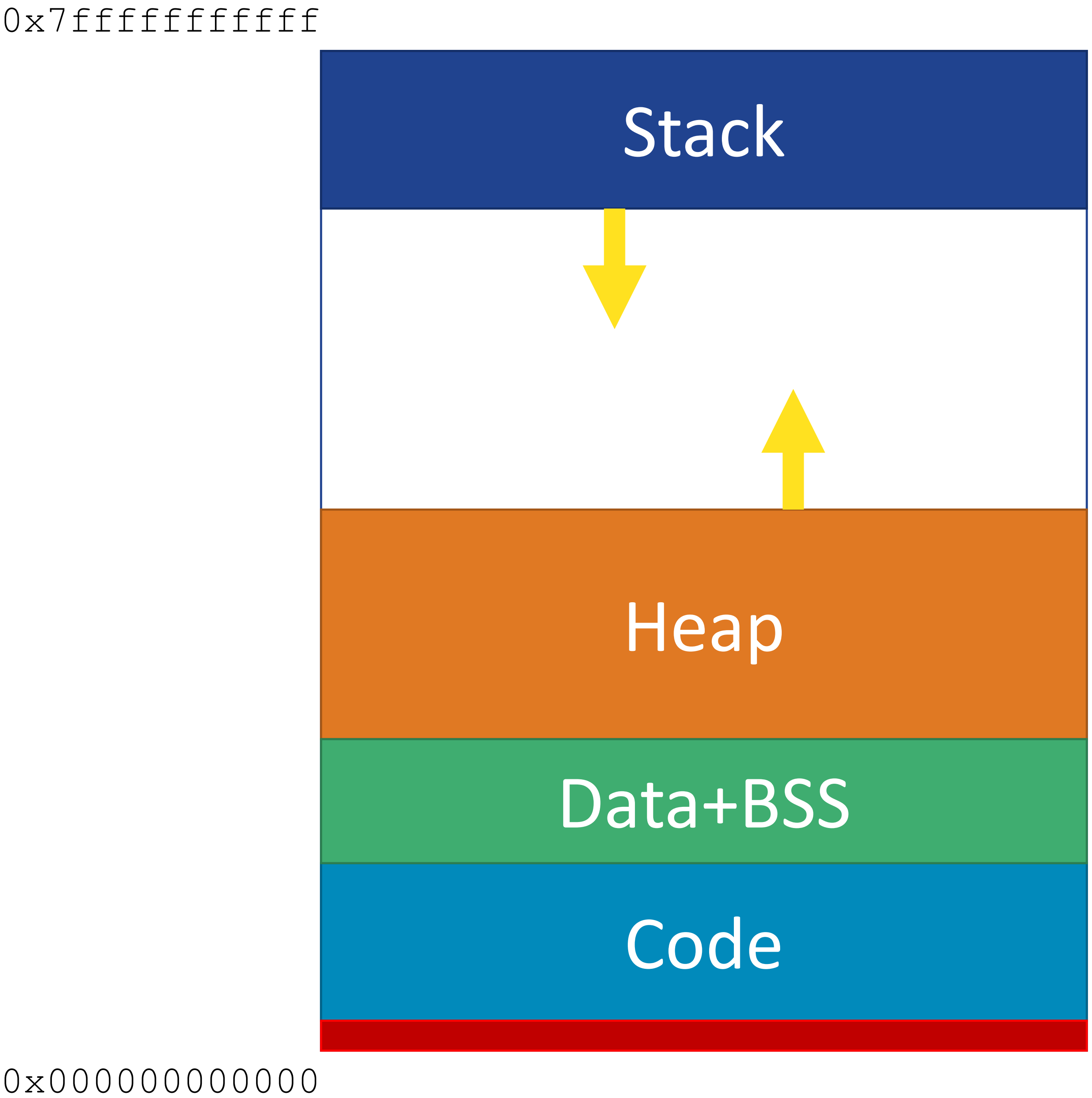


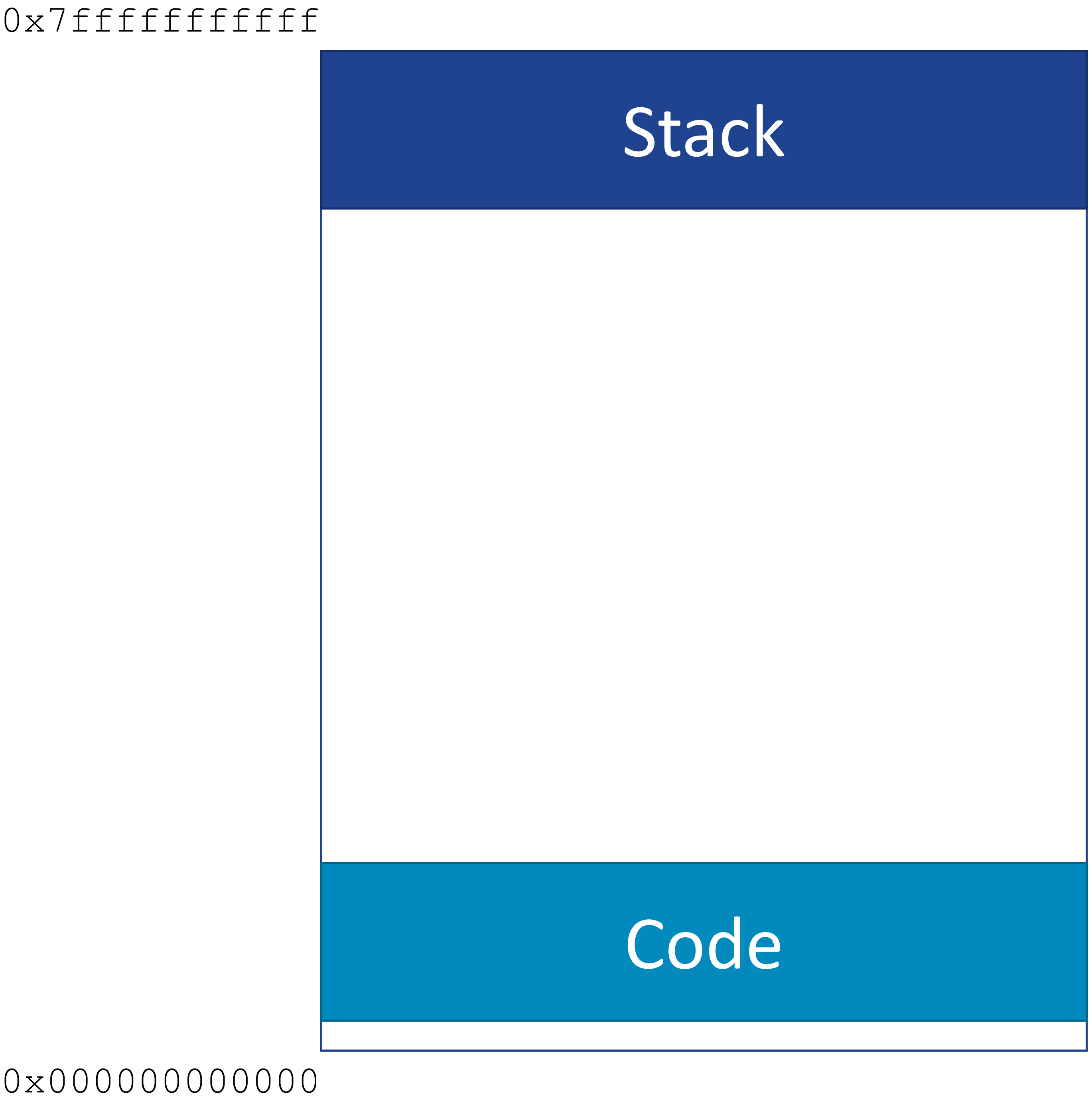
Final ret in each gadget sets pc (`rip`) to beginning of next gadget code

Return-oriented Programming



Final ret in each gadget sets pc (`rip`) to beginning of next gadget code


0x7fffffffffff
Stack
Heap
Data+BSS
Code
0x000000000000


0x7fffffffffff
Stack
Code
0x000000000000

Stack

Code

Stack

7fffffffea88	…
7fffffffea80	"\bin\sh\0"
7fffffffea78	0x1122334455667788
7fffffffea70	0x7fffffffea80
7fffffffea68	0x40042a
7fffffffea60	0x7fffffffea80
7fffffffea58	0x7fffffffea70
7fffffffea50	0x4004b8
7fffffffea48	0x4002a3
7fffffffea40	0x400660
7fffffffea38	0x7fffffffea78
7fffffffea30	0x3b3b3b3b3b3b3b3b
7fffffffea28	0x400420
7fffffffea20	0x40090b
7fffffffea18	…

Code

Take a guess at the significance of each value on the stack.

Suppose that I manually crafted this smashed stack.

Stack

7fffffffea88	…
7fffffffea80	"\bin\sh\0"
7fffffffea78	0x1122334455667788
7fffffffea70	0x7fffffffea80
7fffffffea68	0x40042a
7fffffffea60	0x7fffffffea80
7fffffffea58	0x7fffffffea70
7fffffffea50	0x4004b8
7fffffffea48	0x4002a3
7fffffffea40	0x400660
7fffffffea38	0x7fffffffea78
7fffffffea30	0x3b3b3b3b3b3b3b3b
7fffffffea28	0x400420
7fffffffea20	0x40090b
7fffffffea18	…

Code

Take a guess at the significance of each value on the stack.

Stack

7fffffffea88	…
7fffffffea80	"\bin\sh\0"
7fffffffea78	0x1122334455667788
7fffffffea70	0x7fffffffea80
7fffffffea68	0x40042a
7fffffffea60	0x7fffffffea80
7fffffffea58	0x7fffffffea70
7fffffffea50	[illegible]
7fffffffea48	[illegible]
7fffffffea40	0x400660
7fffffffea38	0x7fffffffea78
7fffffffea30	0x3b3b3b3b3b3b3b3b
7fffffffea28	0x400420
7fffffffea20	0x40090b
7fffffffea18	…

Code

0x4002a3	40 00 F8 C3
…	
0x400420	5F 5E C3
…	
0x40042a	0F 05 C3
…	
0x4004b8	5E 5F C3
…	
…	
0x400660	48 89 06 48 89 C2 C3
…	
…	
…	
0x40090b	48 31 C0 C3
…	

What do you notice about the code section?

Stack

8-Byte Alignment

7fffffffea88	…
7fffffffea80	"\bin\sh\0"
7fffffffea78	0x1122334455667788
7fffffffea70	0x7fffffffea80
7fffffffea68	0x40042a
7fffffffea60	0x7fffffffea80
7fffffffea58	0x7fffffffea70
7fffffffea50	0x4004b8
7fffffffea48	0x4002a3
7fffffffea40	0x400660
7fffffffea38	0x7fffffffea78
7fffffffea30	0x3b3b3b3b3b3b3b3b
7fffffffea28	0x400420
7fffffffea20	0x40090b
7fffffffea18	…

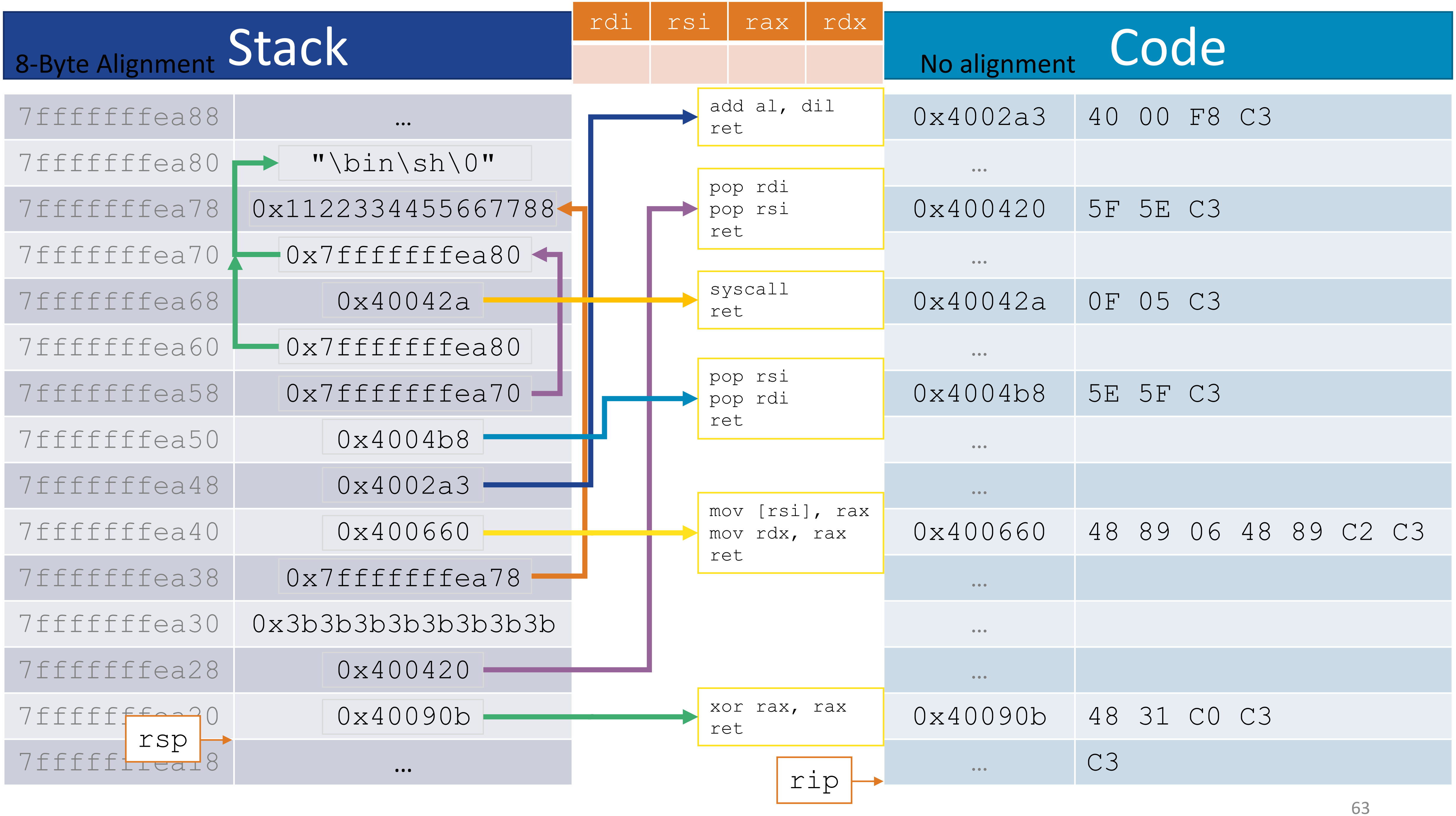
Code

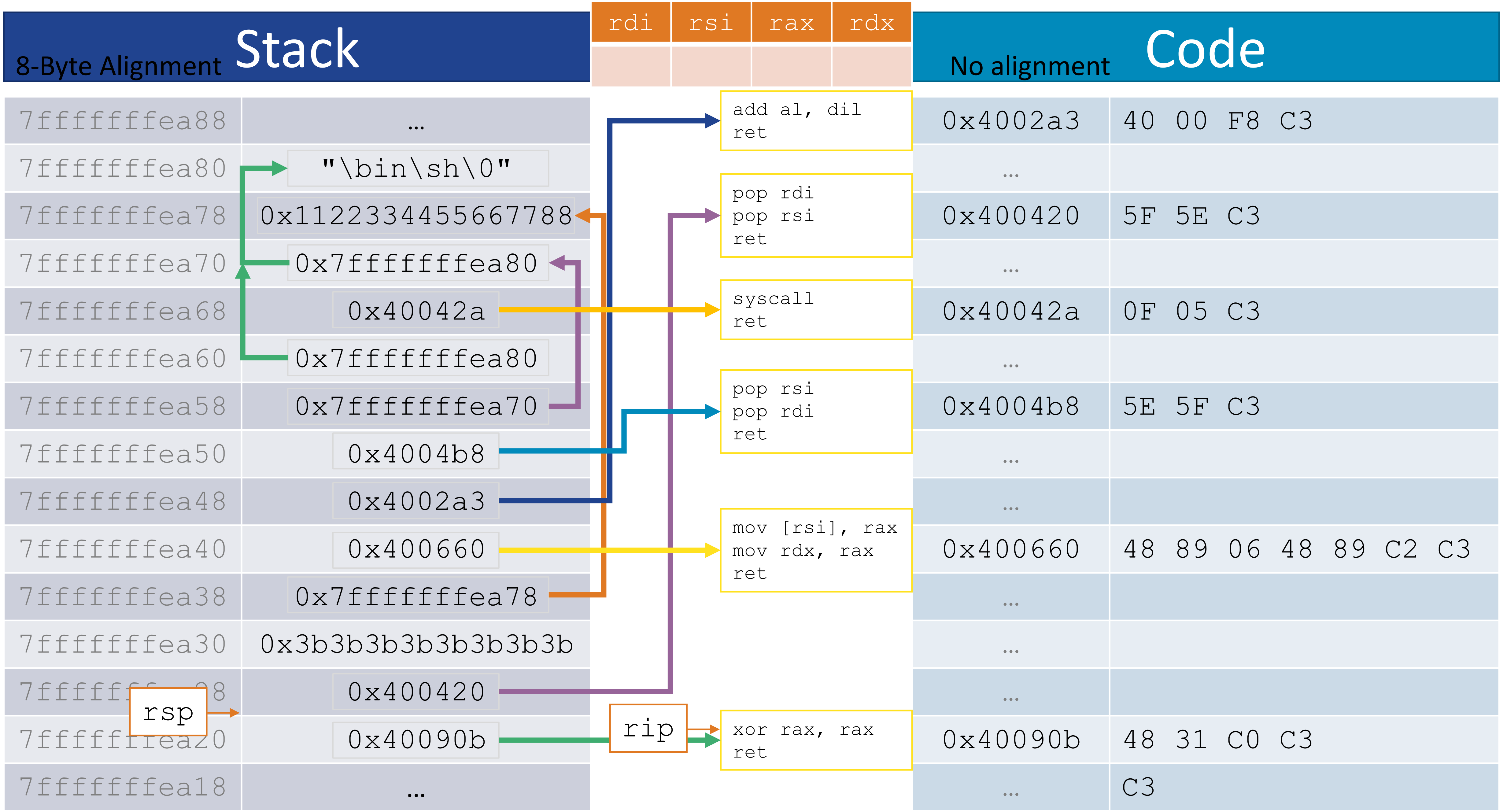
No alignment

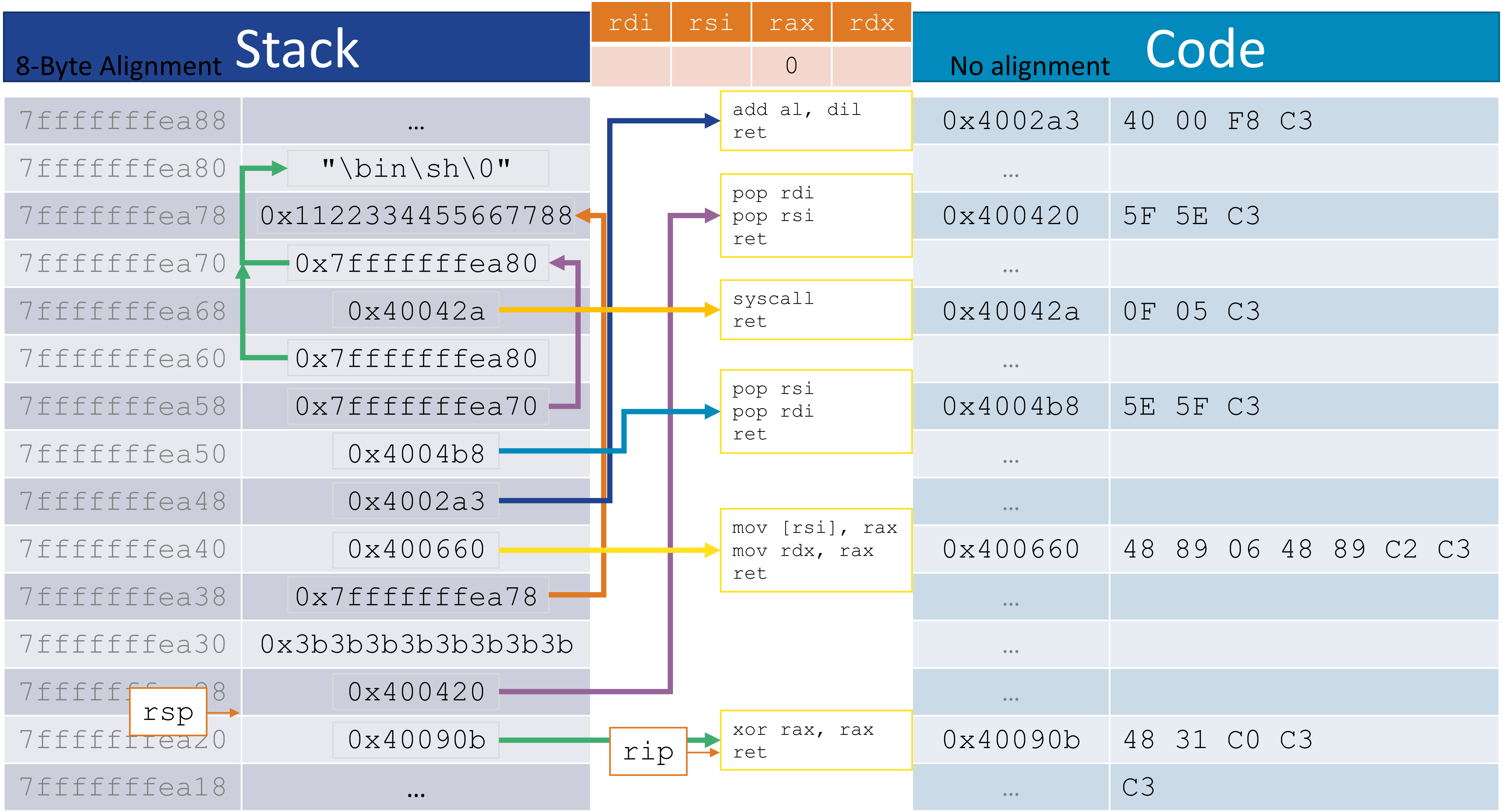
Gadgets		
add al, dil ret	0x4002a3	40 00 F8 C3
	…	
pop rdi pop rsi ret	0x400420	5F 5E C3
	…	
syscall ret	0x40042a	0F 05 C3
	…	
pop rsi pop rdi ret	0x4004b8	5E 5F C3
	…	
	…	

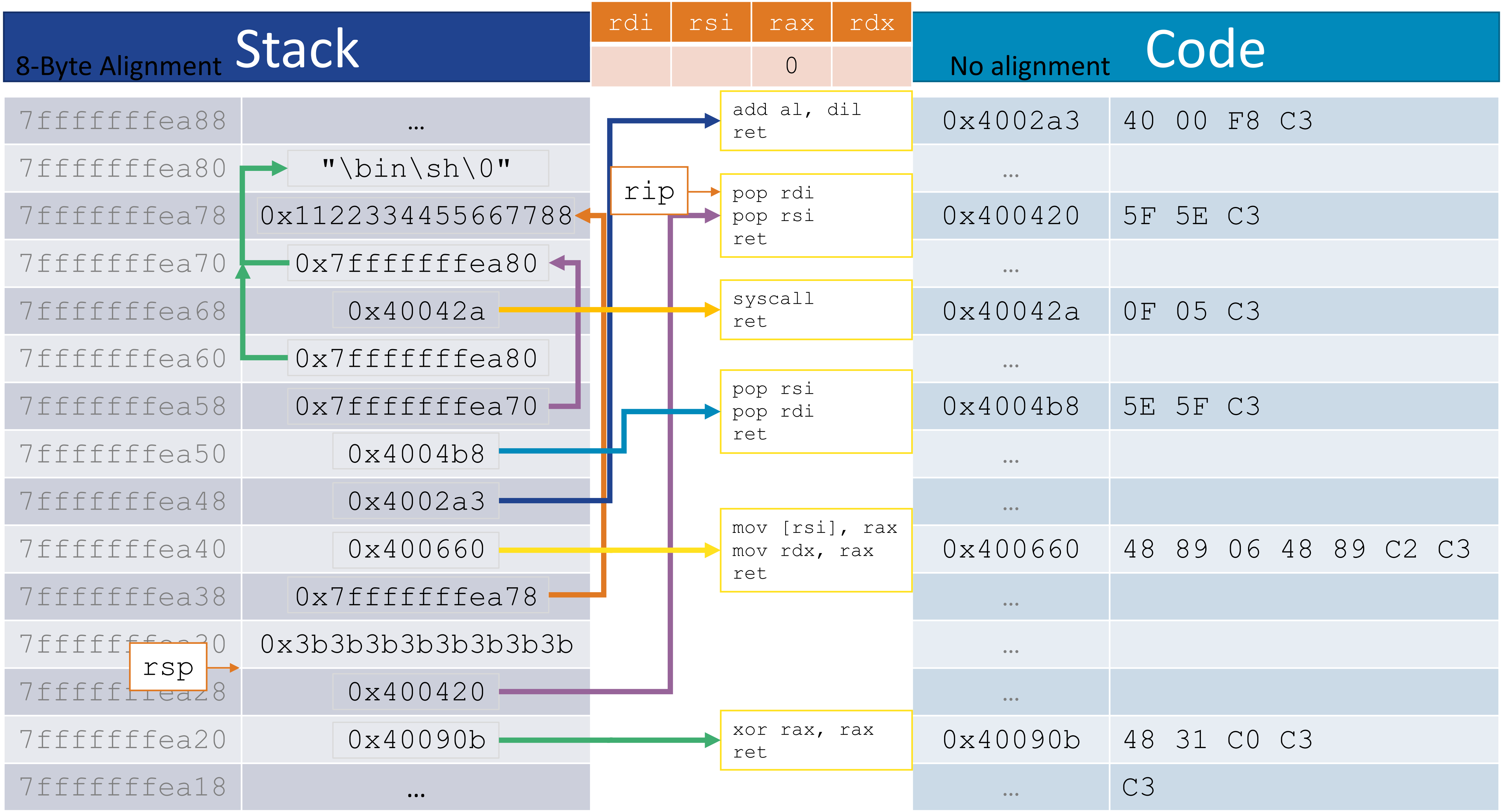
mov [rsi], rax
mov rdx, rax
ret
	0x400660	48 89 06 48 89 C2 C3

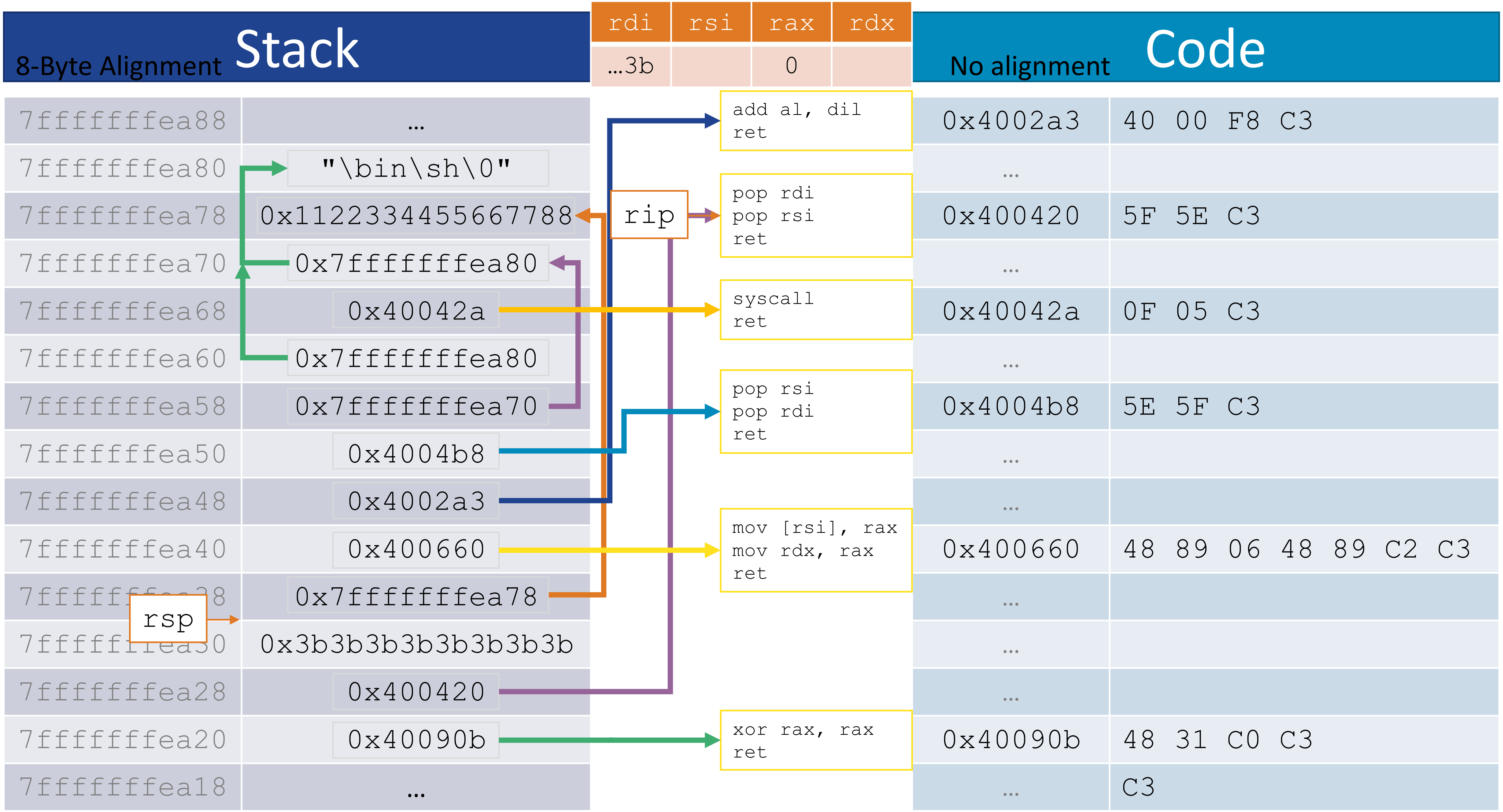
	…	
	…	
	…	
xor rax, rax ret	0x40090b	48 31 C0 C3
	…	

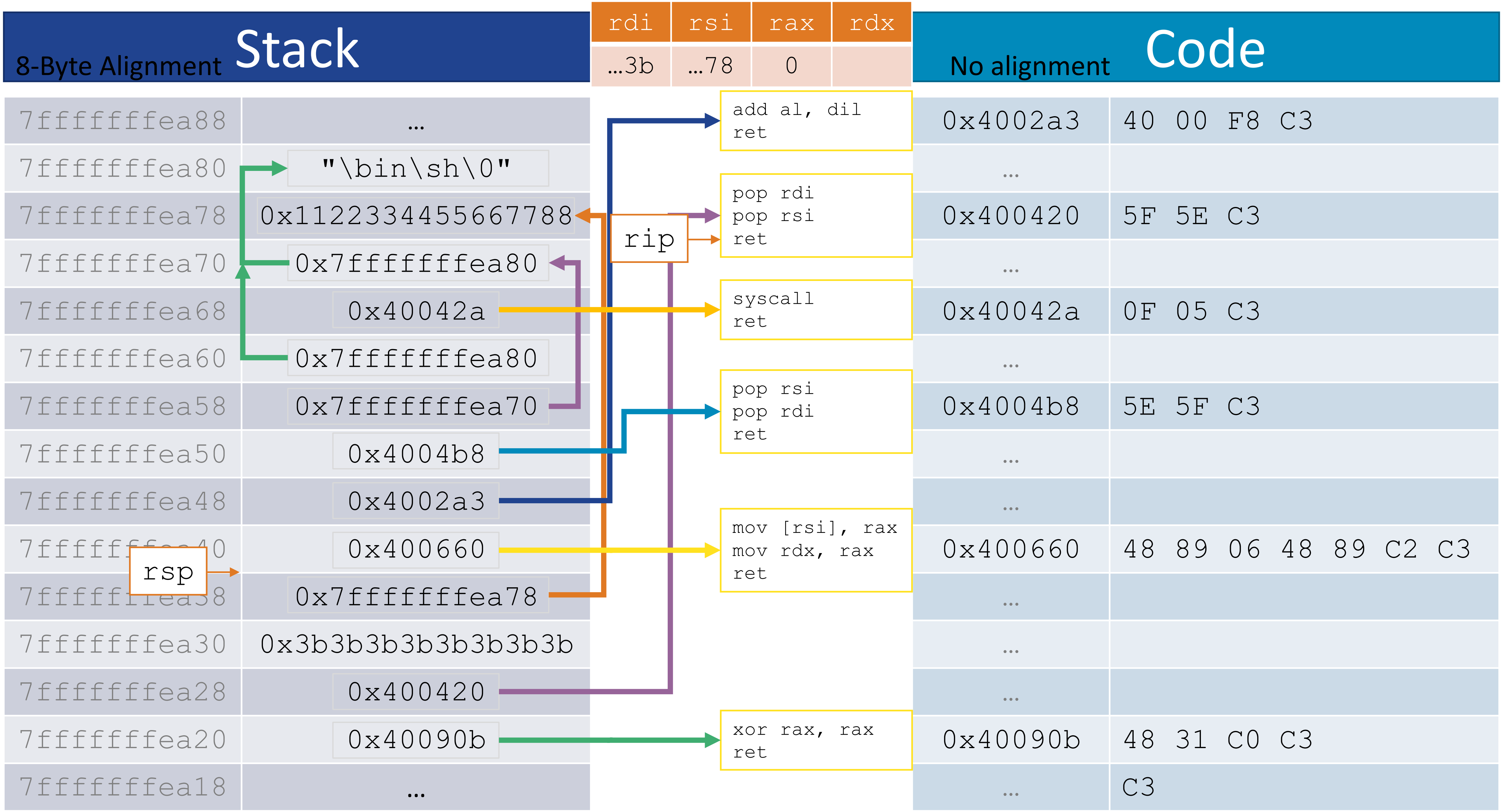

Stack
8-Byte Alignment
rdi
rsi
rax
rdx
Code
No alignment
7fffffffea88
...
7fffffffea80
"\bin\sh\0"
7fffffffea78
0x1122334455667788
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
0x40090b
rsp
...
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
rip
0x4002a3
40 00 F8 C3
0x400420
5F 5E C3
0x40042a
0F 05 C3
0x4004b8
5E 5F C3
0x400660
48 89 06 48 89 C2 C3
0x40090b
48 31 C0 C3
C3

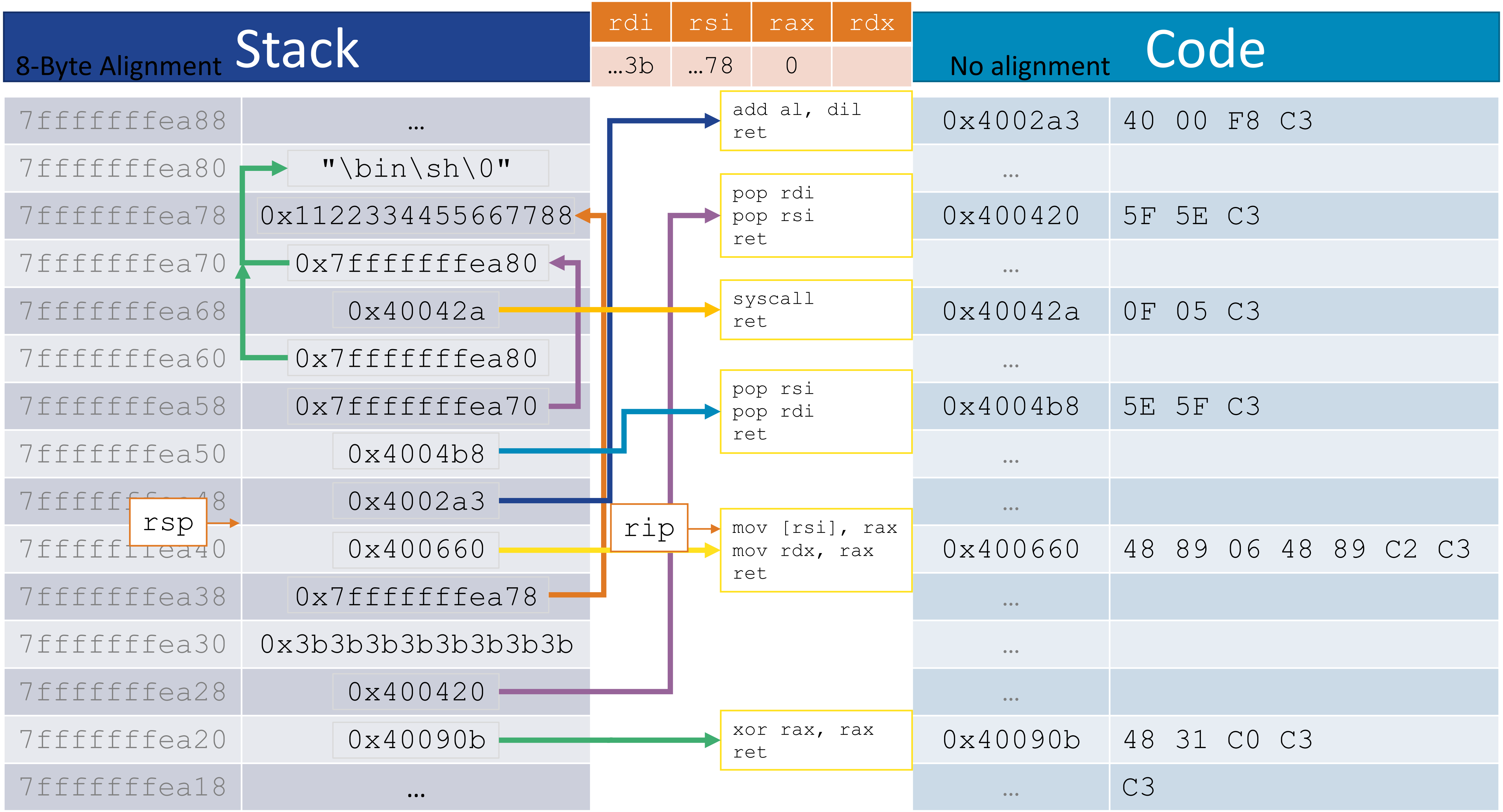

Stack
8-Byte Alignment
rdi
rsi
rax
rdx
Code
No alignment
7fffffffea88
...
7fffffffea80
"\bin\sh\0"
7fffffffea78
0x1122334455667788
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
0x400420
rsp
0x40090b
7fffffffea18
...
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
rip
xor rax, rax
ret
0x4002a3
40 00 F8 C3
...
0x400420
5F 5E C3
...
0x40042a
0F 05 C3
...
0x4004b8
5E 5F C3
...
...
0x400660
48 89 06 48 89 C2 C3
...
...
...
0x40090b
48 31 C0 C3
...
C3

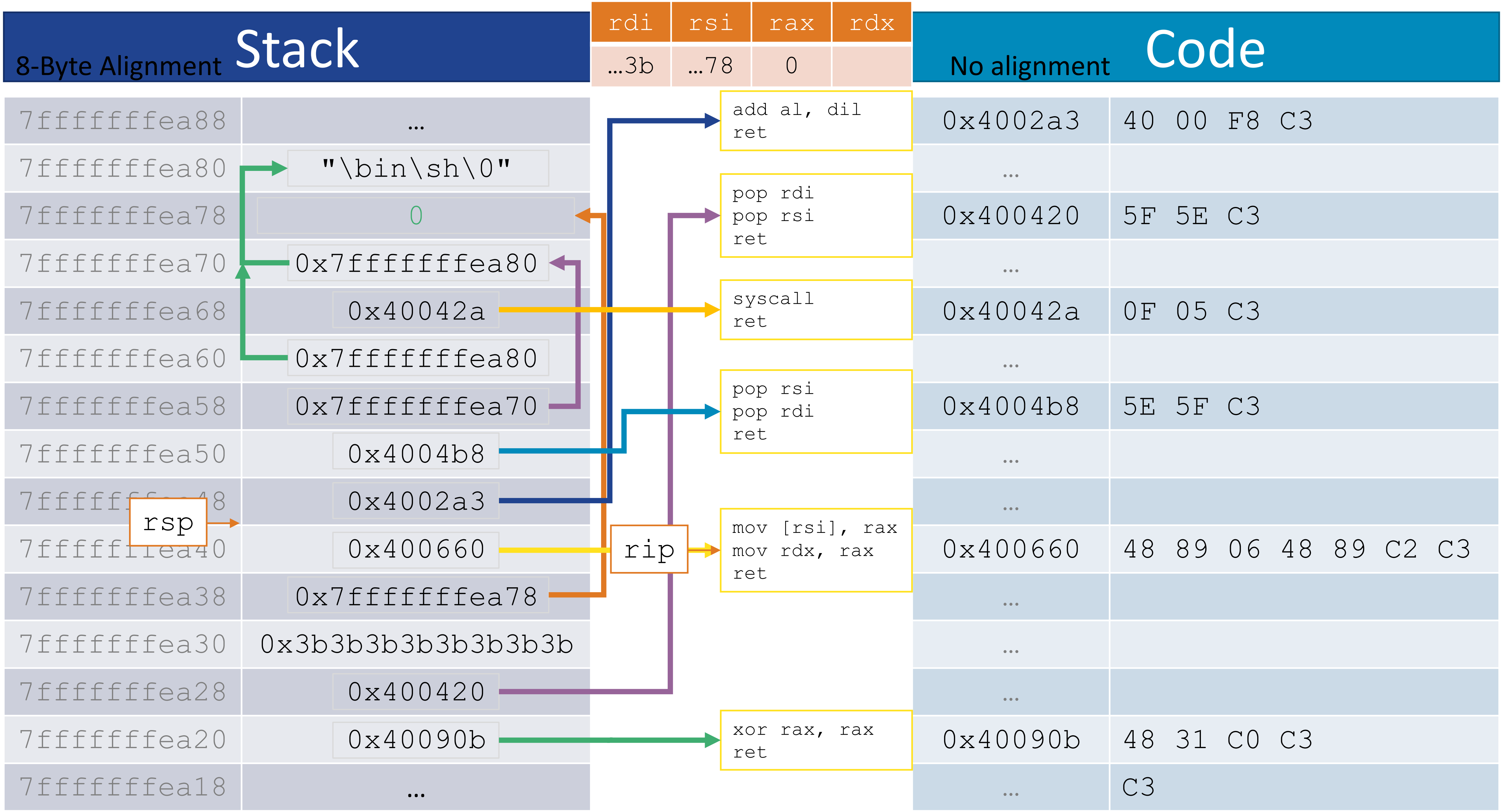

Stack
8-Byte Alignment
rdi
rsi
rax
rdx
0
Code
No alignment
7fffffffea88
...
7fffffffea80
"\bin\sh\0"
7fffffffea78
0x1122334455667788
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
0x400420
rsp
0x40090b
7fffffffea18
...
rip
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
...
0x400420
5F 5E C3
...
0x40042a
0F 05 C3
...
0x4004b8
5E 5F C3
...
...
0x400660
48 89 06 48 89 C2 C3
...
...
...
0x40090b
48 31 C0 C3
...
C3

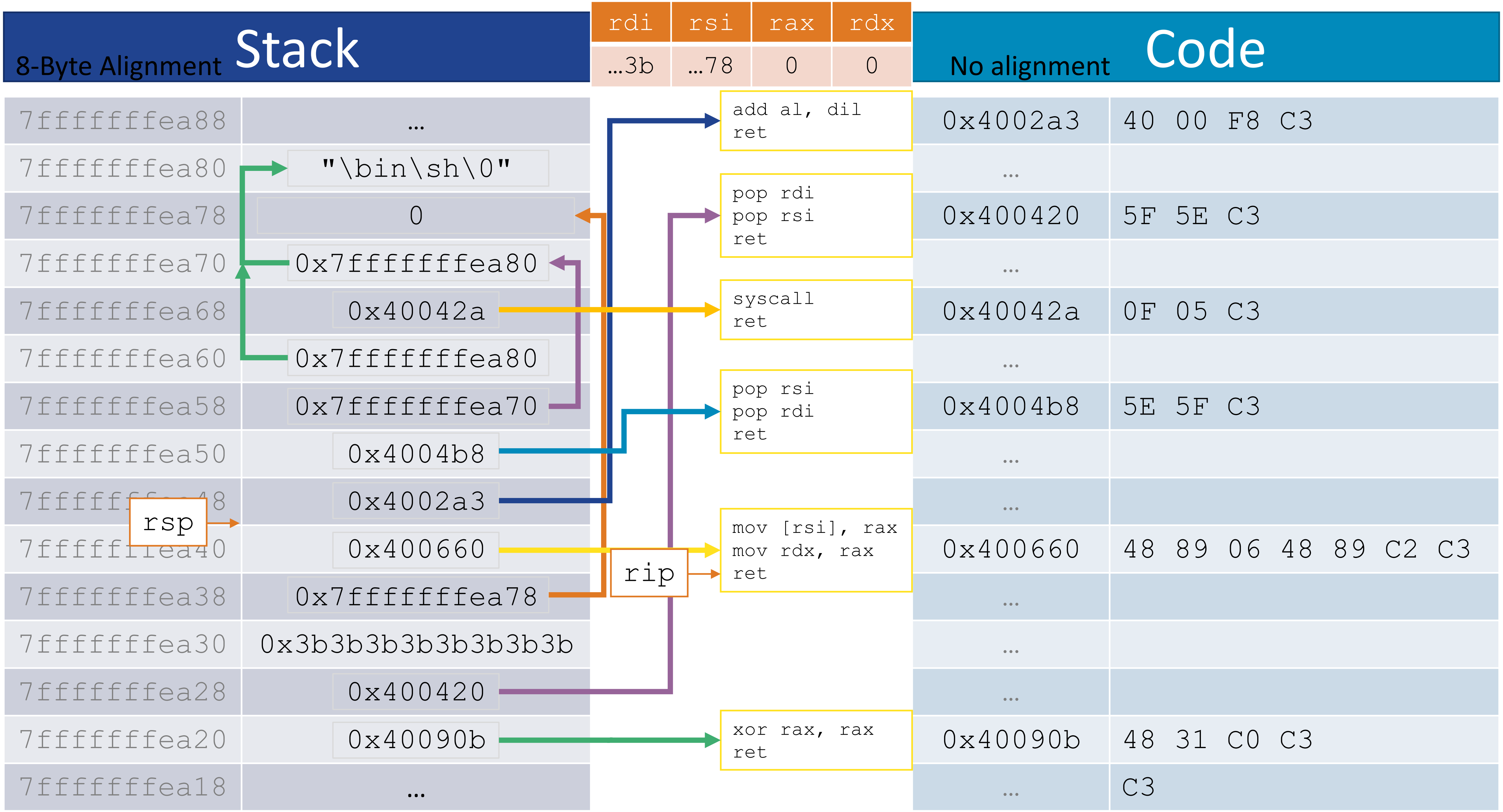

8-Byte Alignment
Stack
rdi
rsi
rax
rdx
0
No alignment
Code
7fffffffea88
...
7fffffffea80
"\bin\sh\0"
7fffffffea78
0x1122334455667788
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
...
rip
rsp
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
0x400420
5F 5E C3
0x40042a
0F 05 C3
0x4004b8
5E 5F C3
0x400660
48 89 06 48 89 C2 C3
0x40090b
48 31 C0 C3
C3


Stack
8-Byte Alignment
rdi
rsi
rax
rdx
...3b
0
Code
No alignment
7fffffffea88
...
7fffffffea80
"\bin\sh\0"
7fffffffea78
0x1122334455667788
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
0x7fffffffea78
rsp
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
...
rip
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
0x400420
5F 5E C3
0x40042a
0F 05 C3
0x4004b8
5E 5F C3
0x400660
48 89 06 48 89 C2 C3
0x40090b
48 31 C0 C3
C3

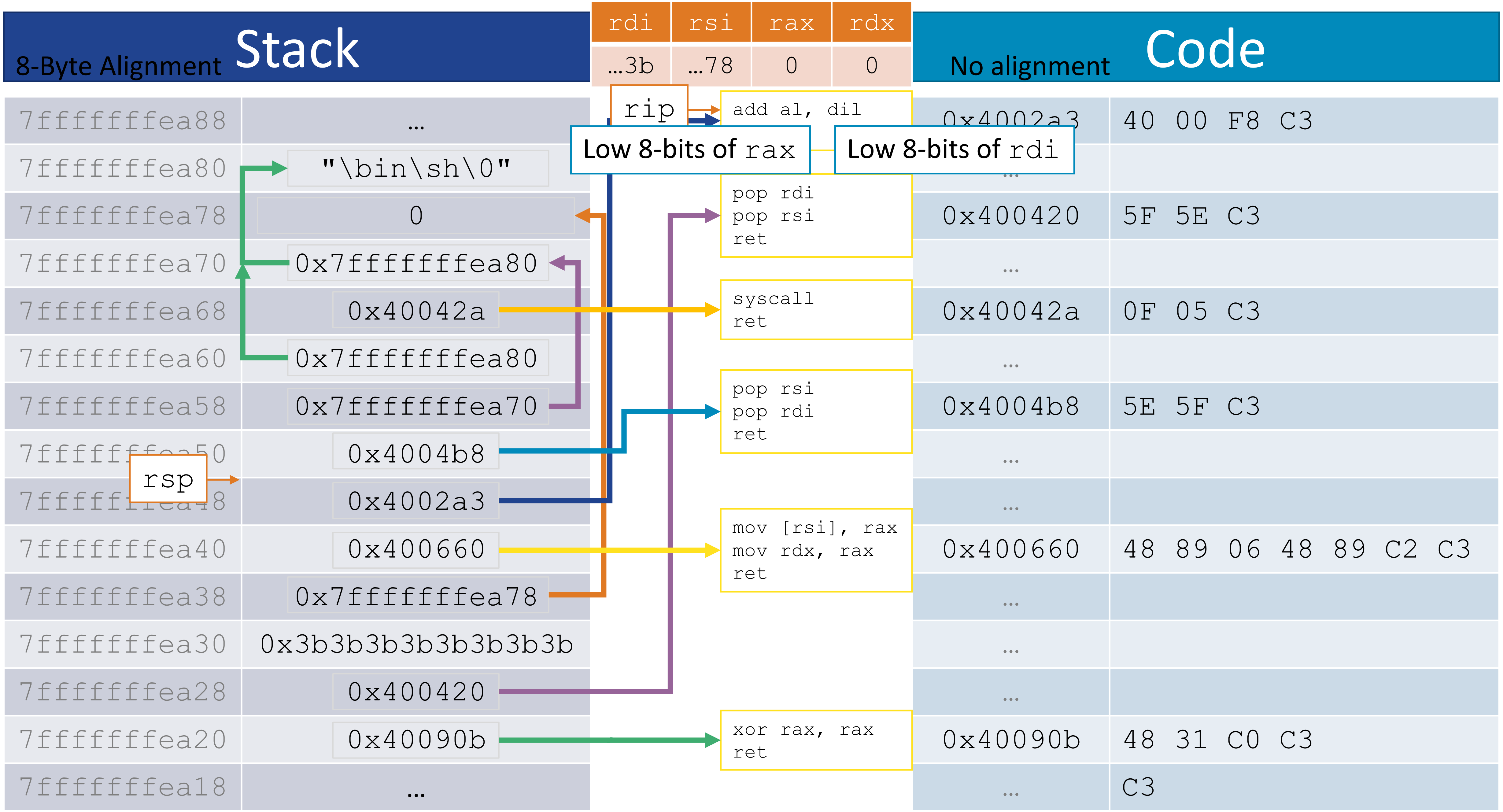

Stack
8-Byte Alignment
rdi
rsi
rax
rdx
…3b
…78
0
Code
No alignment
7fffffffea88
…
7fffffffea80
"\bin\sh\0"
7fffffffea78
0x1122334455667788
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
…
rsp
rip
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
…
0x400420
5F 5E C3
…
0x40042a
0F 05 C3
…
0x4004b8
5E 5F C3
…
…
0x400660
48 89 06 48 89 C2 C3
…
…
…
0x40090b
48 31 C0 C3
…
C3

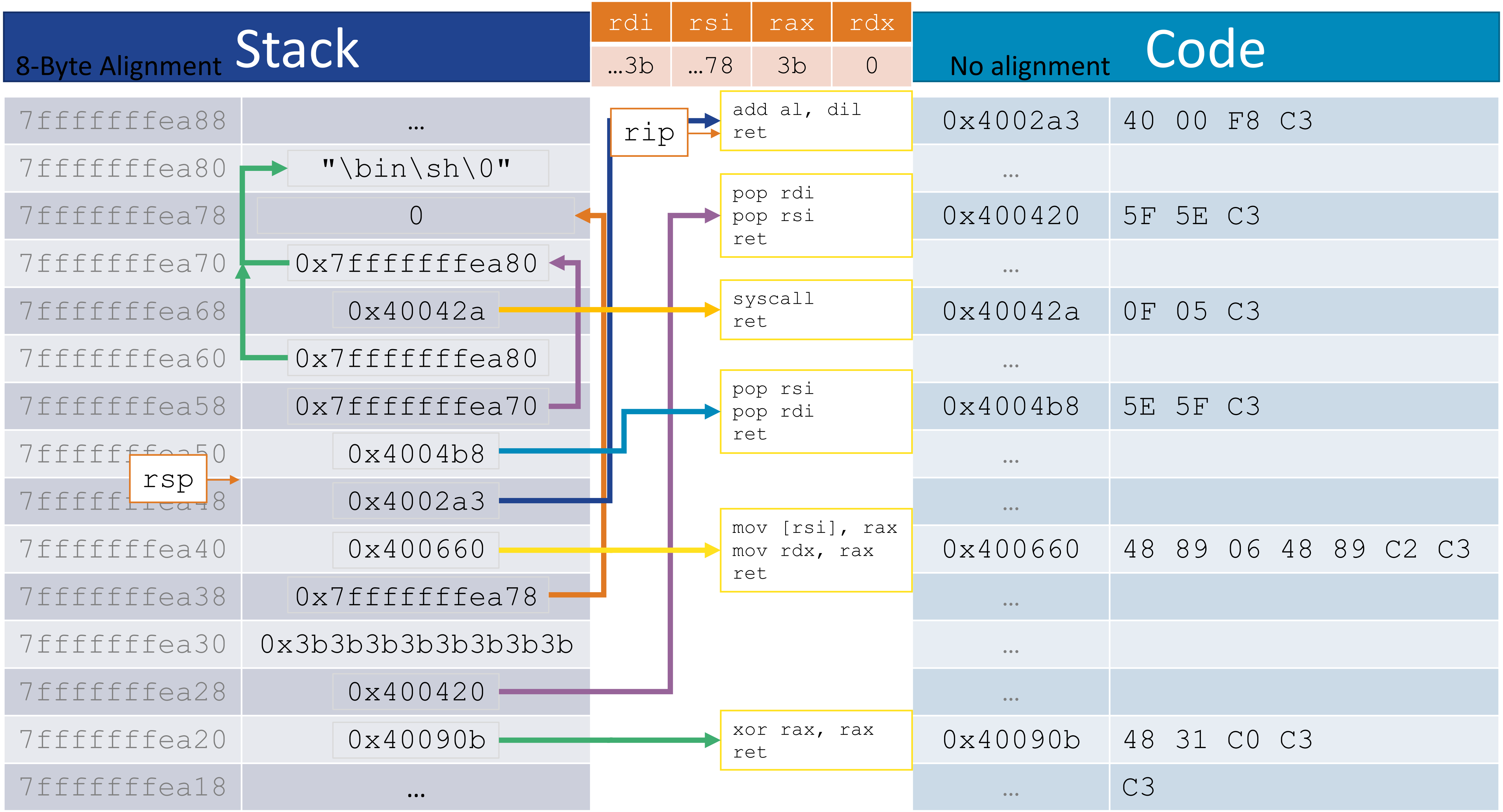

Stack
8-Byte Alignment
rdi
rsi
rax
rdx
…3b
…78
0
Code
No alignment
7fffffffea88
…
7fffffffea80
"\bin\sh\0"
7fffffffea78
0x1122334455667788
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
0x4002a3
rsp
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
…
rip
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
…
0x400420
5F 5E C3
…
0x40042a
0F 05 C3
…
0x4004b8
5E 5F C3
…
…
0x400660
48 89 06 48 89 C2 C3
…
…
…
0x40090b
48 31 C0 C3
…
C3

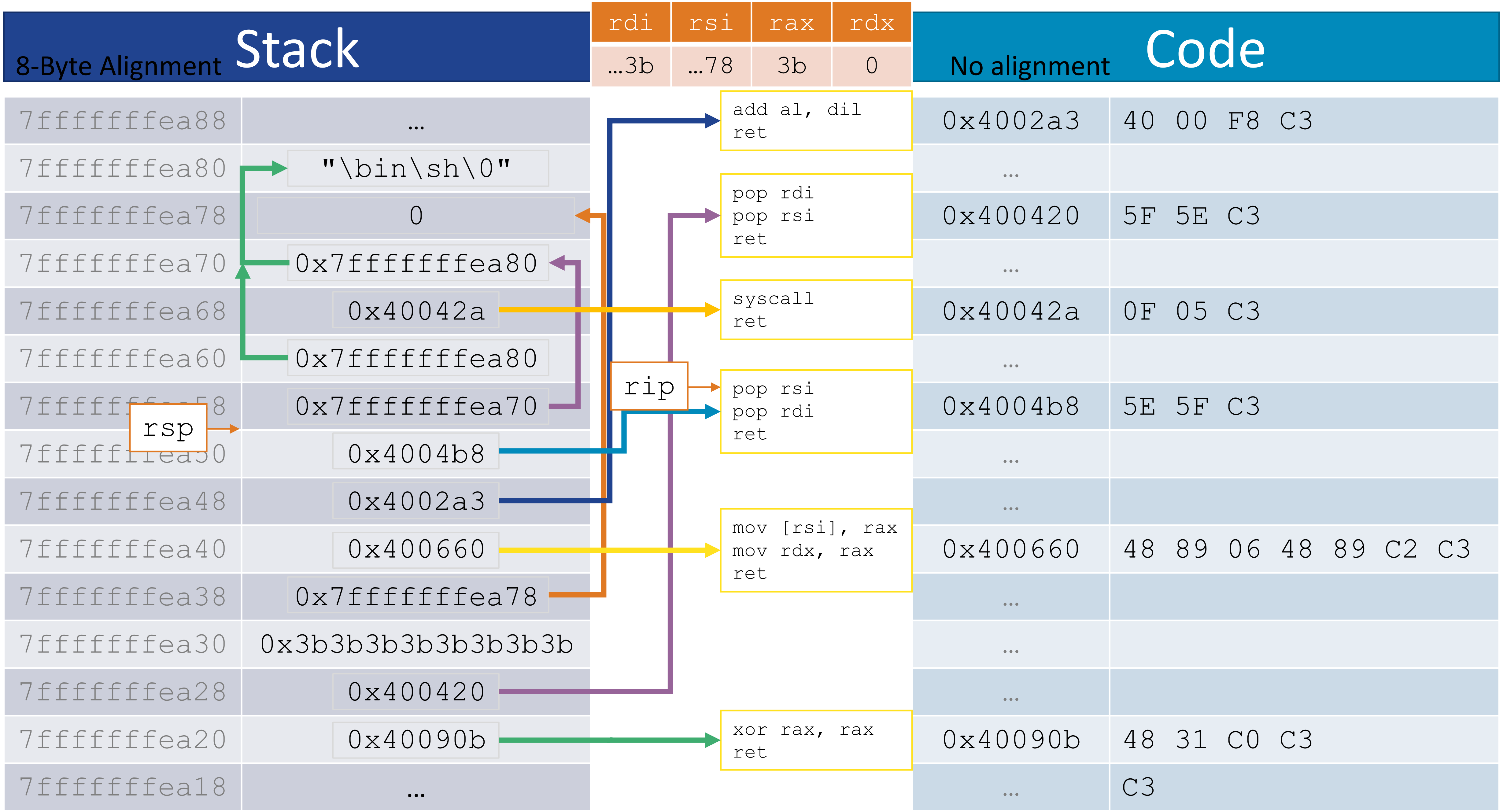

8-Byte Alignment
Stack
rdi
rsi
rax
rdx
...3b
...78
0
No alignment
Code
7fffffffea88
...
7fffffffea80
"\bin\sh\0"
7fffffffea78
0
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
0x4002a3
rsp
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
...
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
rip
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
0x400420
5F 5E C3
0x40042a
0F 05 C3
0x4004b8
5E 5F C3
0x400660
48 89 06 48 89 C2 C3
0x40090b
48 31 C0 C3
C3

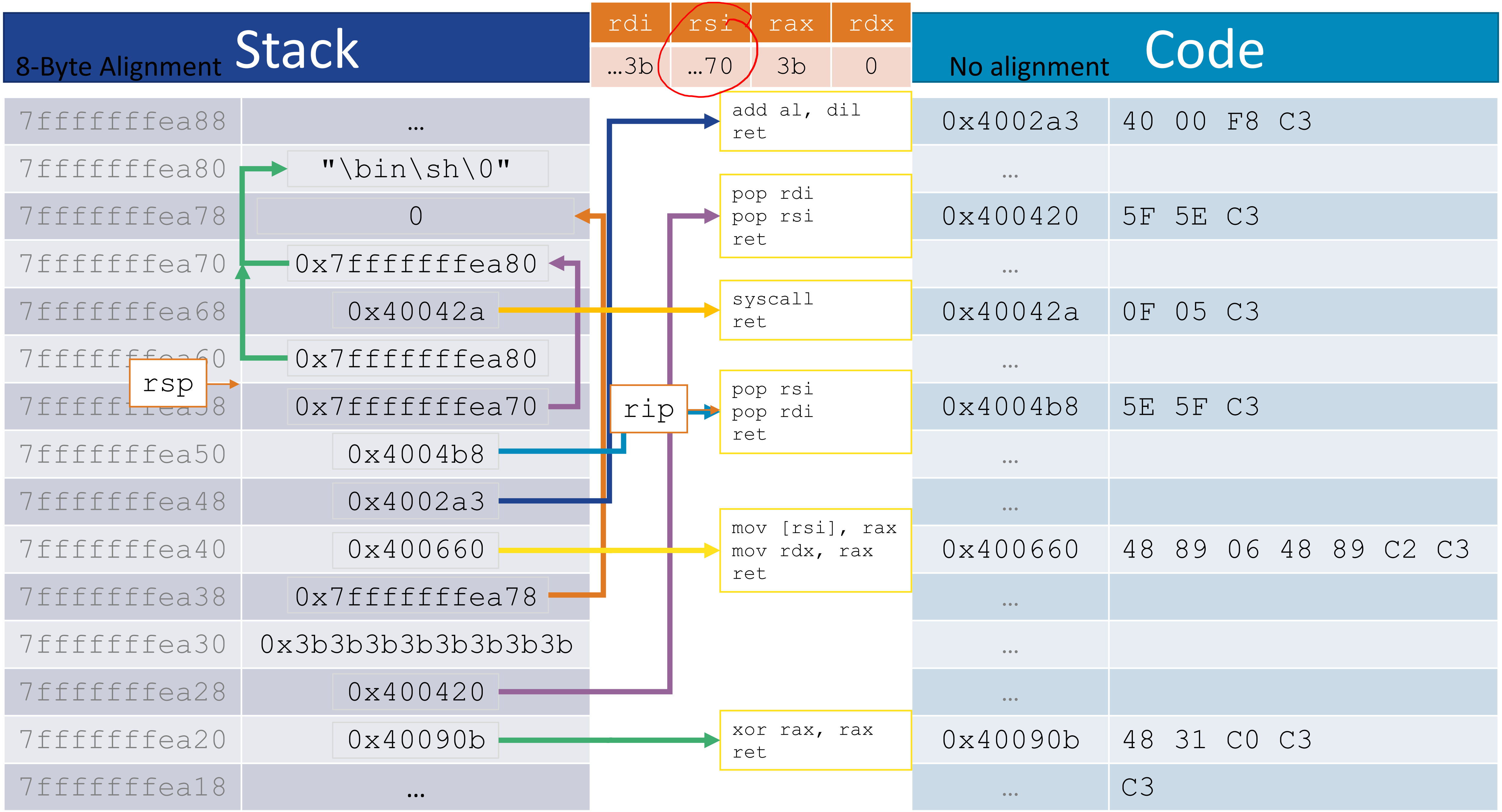


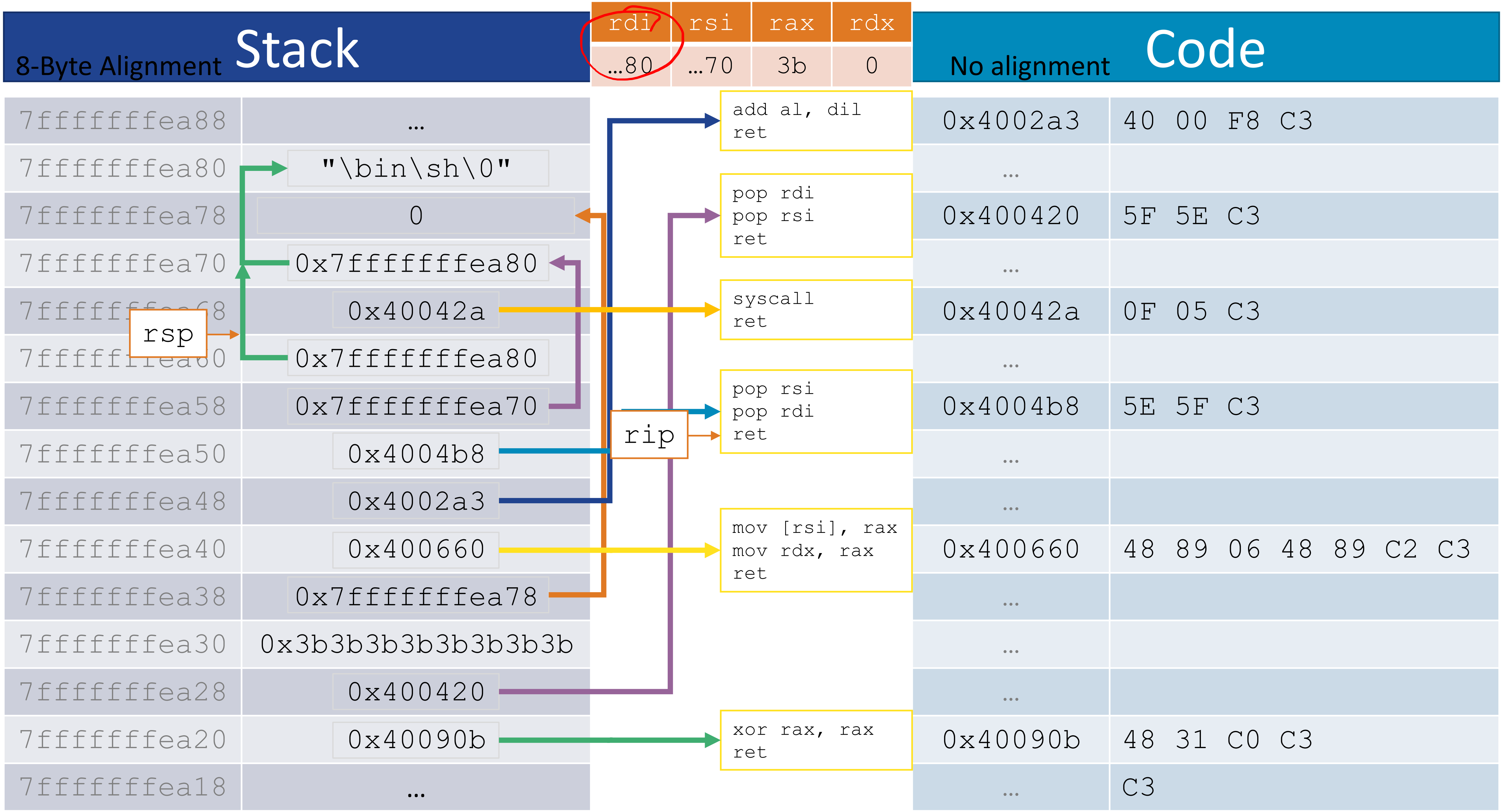
Stack
8-Byte Alignment
rdi
rsi
rax
rdx
…3b
…78
0
0
Code
No alignment
7fffffffea88
…
7fffffffea80
"\bin\sh\0"
7fffffffea78
0
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
0x4002a3
rsp
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
…
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
rip
xor rax, rax
ret
0x4002a3
40 00 F8 C3
…
0x400420
5F 5E C3
…
0x40042a
0F 05 C3
…
0x4004b8
5E 5F C3
…
…
0x400660
48 89 06 48 89 C2 C3
…
…
…
0x40090b
48 31 C0 C3
…
C3

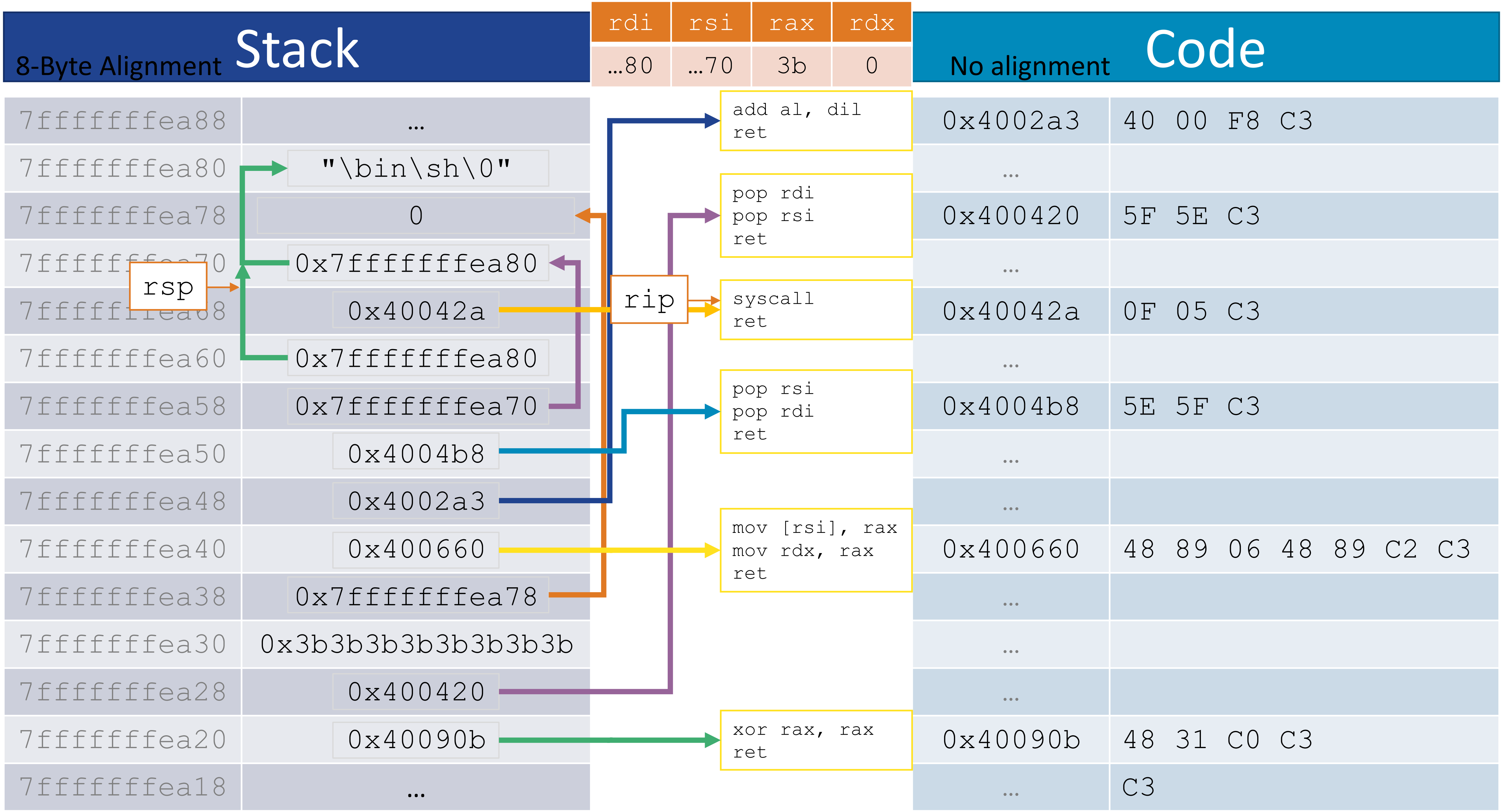

Stack
8-Byte Alignment
rdi
rsi
rax
rdx
...3b
...78
0
0
rip
rsp
Code
No alignment
Low 8-bits of rax
Low 8-bits of rdi
7fffffffea88
...
7fffffffea80
"\bin\sh\0"
7fffffffea78
0
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
...
add al, dil
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
...
0x400420
5F 5E C3
0x40042a
0F 05 C3
0x4004b8
5E 5F C3
0x400660
48 89 06 48 89 C2 C3
0x40090b
48 31 C0 C3
C3

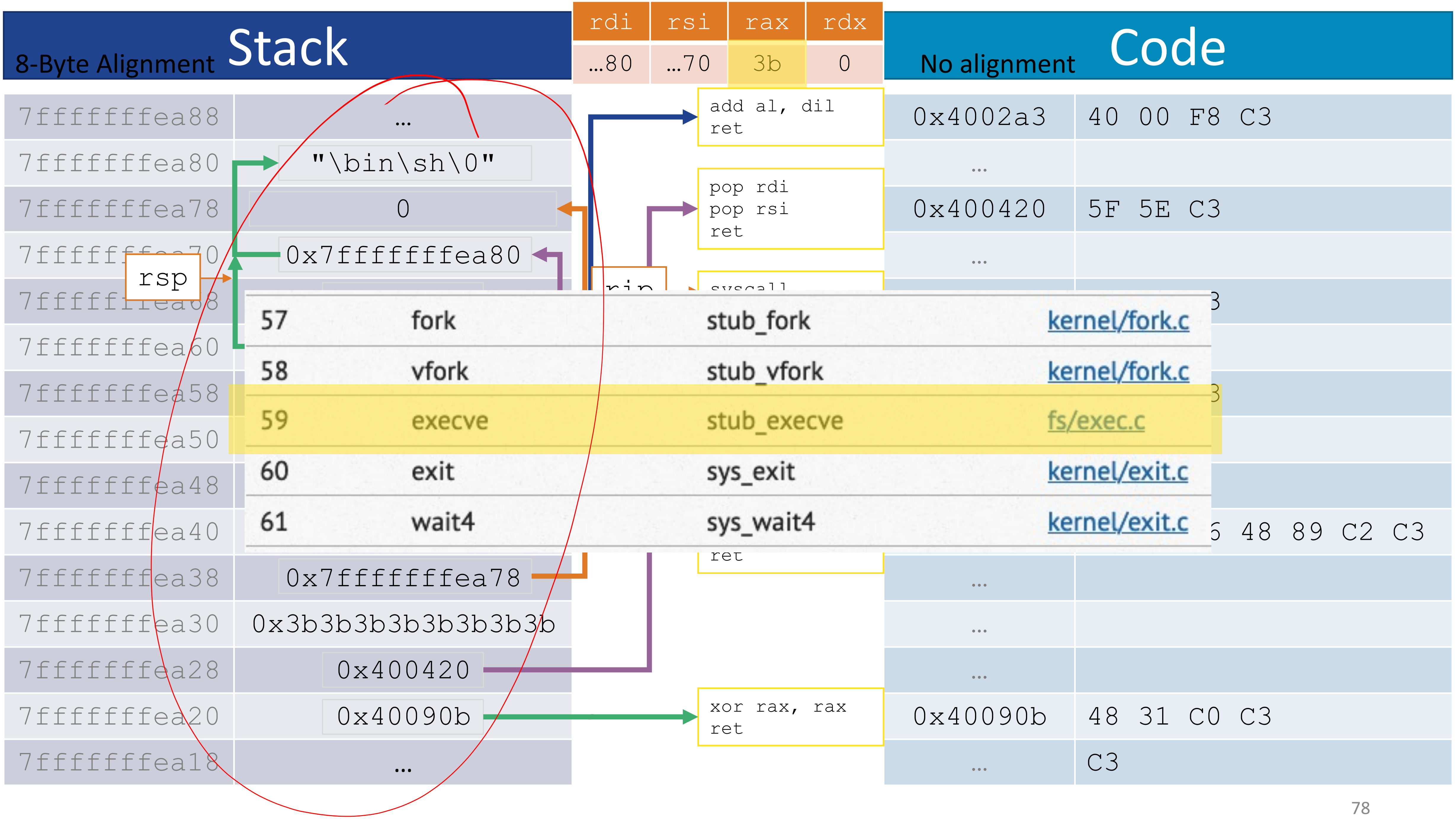

Stack
8-Byte Alignment
rdi
rsi
rax
rdx
…3b
…78
3b
0
Code
No alignment
7fffffffea88
…
7fffffffea80
"\bin\sh\0"
7fffffffea78
0
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
0x4004b8
rsp
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
…
rip
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
0x400420
5F 5E C3
0x40042a
0F 05 C3
0x4004b8
5E 5F C3
0x400660
48 89 06 48 89 C2 C3
0x40090b
48 31 C0 C3
C3


8-Byte Alignment
Stack
rdi
rsi
rax
rdx
...3b
...78
3b
0
No alignment
Code
7fffffffea88
...
7fffffffea80
"\bin\sh\0"
7fffffffea78
0
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
rsp
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
...
rip
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
...
0x400420
5F 5E C3
...
0x40042a
0F 05 C3
...
0x4004b8
5E 5F C3
...
...
0x400660
48 89 06 48 89 C2 C3
...
...
...
0x40090b
48 31 C0 C3
...
C3


Stack
8-Byte Alignment
rdi
rsi
rax
rdx
…3b
…70
3b
0
Code
No alignment
7fffffffea88
…
7fffffffea80
"\bin\sh\0"
7fffffffea78
0
7fffffffea70
0x7fffffffea80
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
rsp
7fffffffea58
0x7fffffffea70
rip
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
…
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
…
0x400420
5F 5E C3
…
0x40042a
0F 05 C3
…
0x4004b8
5E 5F C3
…
…
0x400660
48 89 06 48 89 C2 C3
…
…
…
0x40090b
48 31 C0 C3
…
C3


Stack
8-Byte Alignment
rdi
rsi
rax
rdx
…80
…70
3b
0
Code
No alignment
7fffffffea88
…
7fffffffea80
"\bin\sh\0"
7fffffffea78
0
7fffffffea70
0x7fffffffea80
0x40042a
rsp
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
…
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
rip
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
…
0x400420
5F 5E C3
…
0x40042a
0F 05 C3
…
0x4004b8
5E 5F C3
…
…
0x400660
48 89 06 48 89 C2 C3
…
…
…
0x40090b
48 31 C0 C3
…
C3


Stack
8-Byte Alignment
rdi
rsi
rax
rdx
…80
…70
3b
0
Code
No alignment
7fffffffea88
…
7fffffffea80
"\bin\sh\0"
7fffffffea78
0
7fffffffea70
0x7fffffffea80
rsp
7fffffffea68
0x40042a
7fffffffea60
0x7fffffffea80
7fffffffea58
0x7fffffffea70
7fffffffea50
0x4004b8
7fffffffea48
0x4002a3
7fffffffea40
0x400660
7fffffffea38
0x7fffffffea78
7fffffffea30
0x3b3b3b3b3b3b3b3b
7fffffffea28
0x400420
7fffffffea20
0x40090b
7fffffffea18
…
rip
add al, dil
ret
pop rdi
pop rsi
ret
syscall
ret
pop rsi
pop rdi
ret
mov [rsi], rax
mov rdx, rax
ret
xor rax, rax
ret
0x4002a3
40 00 F8 C3
…
0x400420
5F 5E C3
…
0x40042a
0F 05 C3
…
0x4004b8
5E 5F C3
…
…
0x400660
48 89 06 48 89 C2 C3
…
…
…
0x40090b
48 31 C0 C3
…
C3



57	fork	stub_fork	kernel/fork.c
58	vfork	stub_vfork	kernel/fork.c
59	execve	stub_execve	fs/exec.c
60	exit	sys_exit	kernel/exit.c
61	wait4	sys_wait4	kernel/exit.c

8-Byte Alignment **Stack**

rdi	rsi	rax	rdx
…80	…70	3b	0

No alignment **Code**

7fffffffea88	…
7fffffffea80	"\bin\sh\0"
7fffffffea78	0
7fff	
7fff	
7fff	
7fff	
7fff	
7fff	
7fff	
7fff	
7fff	
7fff	
7fff	
7fff	

```
add al, dil
ret
```

```
pop rdi
pop rsi
```

0x4002a3	40 00 F8 C3
…	
0x400420	5F 5E C3
	C3

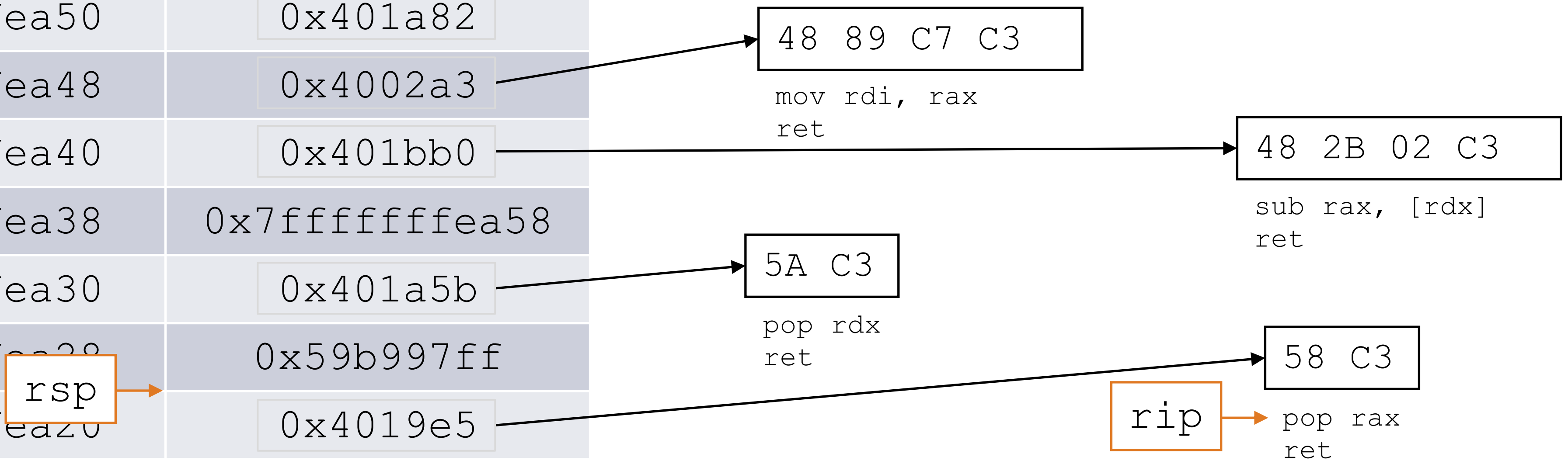
```
EXECVE(2)                         System Calls Manual                         EXECVE(2)

NAME
       execve - execute a file

SYNOPSIS
       #include <unistd.h>

       int
       execve(const char *path, char *const argv[], char *const envp[]);

DESCRIPTION
       execve() transforms the calling process into a new process.  The new process is constructed from an ordinary file, whose name is
       pointed to by path, called the new process file.  This file is either an executable object file, or a file of data for an
       interpreter.  An executable object file consists of an identifying header, followed by pages of data representing the initial
       program (text) and initialized data pages.  Additional pages may be specified by the header to be initialized with zero data;
       see a.out(5).

       An interpreter file begins with a line of the form:

             #! interpreter [arg ...]

       When an interpreter file is execve()'d, the system runs the specified interpreter.  If any optional args are specified, they
       become the first (second, ...) argument to the interpreter. The name of the originally execve()'d file becomes the subsequent
       argument; otherwise, the name of the originally execve()'d file is the first argument.  The original arguments to the invocation
       of the interpreter are shifted over to become the final arguments.  The zeroth argument, normally the name of the execve()'d
       file, is left unchanged.

       The argument argv is a pointer to a null-terminated array of character pointers to null-terminated character strings.  These
       strings construct the argument list to be made available to the new process.  At least one argument must be present in the
       array; by custom, the first element should be the name of the executed program (for example, the last component of path).

man://execve(2) [RO]                                                          1,1           Top
```

Practice with Return-Oriented Programming (ROP)

What are the values in the registers when the function at address `0x401a82` gets called?

rdi	rsi	rax	rdx

7fffffffea58	0x59b997d0
7fffffffea50	0x401a82
7fffffffea48	0x4002a3
7fffffffea40	0x401bb0
7fffffffea38	0x7fffffffea58
7fffffffea30	0x401a5b
7fffffffea28	0x59b997ff
7fffffffea20	0x4019e5

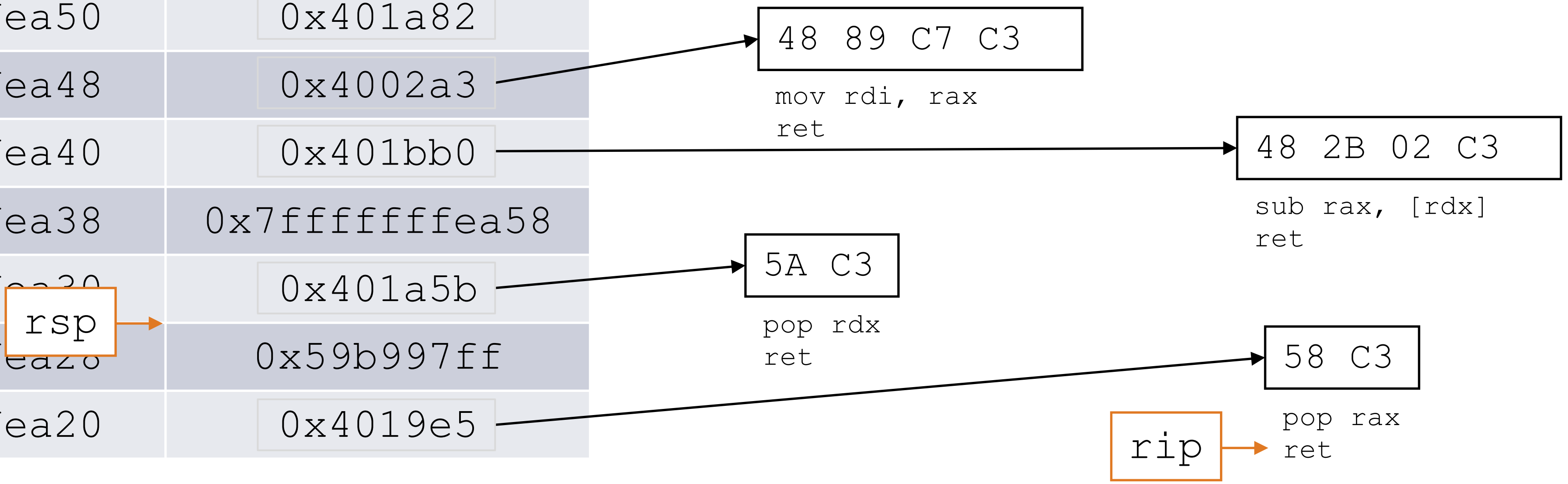


Practice with Return-Oriented Programming (ROP)

What are the values in the registers when the function at address `0x401a82` gets called?

rdi	rsi	rax	rdx
		…97ff	

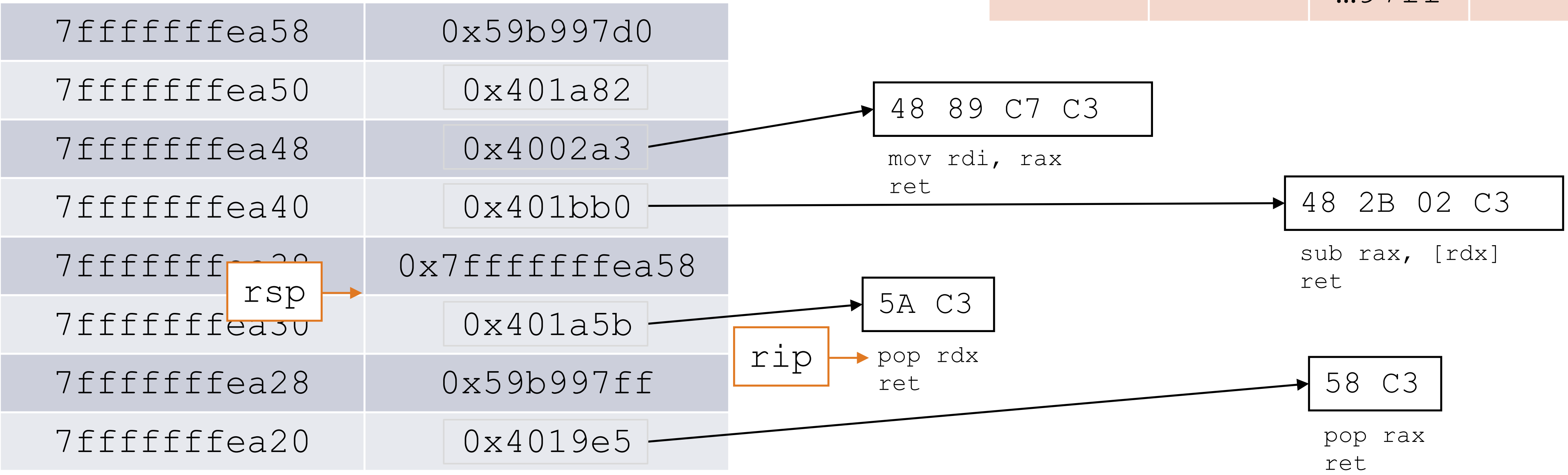
7fffffffea58	0x59b997d0
7fffffffea50	0x401a82
7fffffffea48	0x4002a3
7fffffffea40	0x401bb0
7fffffffea38	0x7fffffffea58
7fffffffea30	0x401a5b
7fffffffea28	0x59b997ff
7fffffffea20	0x4019e5



Practice with Return-Oriented Programming (ROP)

What are the values in the registers when the function at address `0x401a82` gets called?

rdi	rsi	rax	rdx
		…97ff	

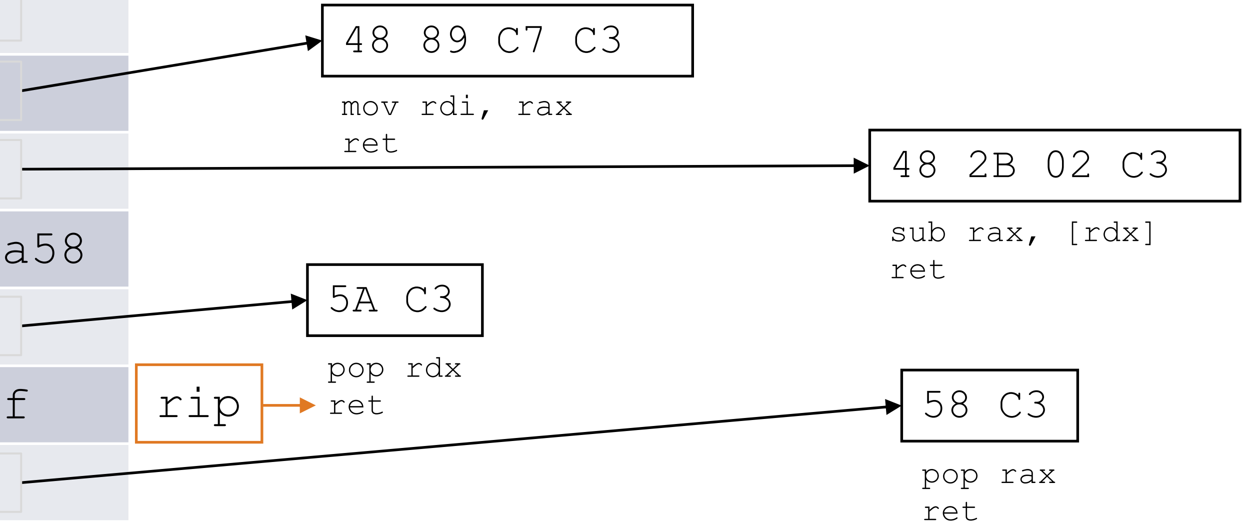


Practice with Return-Oriented Programming (ROP)

What are the values in the registers when the function at address `0x401a82` gets called?

rdi	rsi	rax	rdx
		…97ff	…ea58

7fffffffea58	0x59b997d0
7fffffffea50	0x401a82
7fffffffea48	0x4002a3
7fffffffea40	0x401bb0
7fffffffea38	0x7fffffffea58
7fffffffea30	0x401a5b
7fffffffea28	0x59b997ff
7fffffffea20	0x4019e5



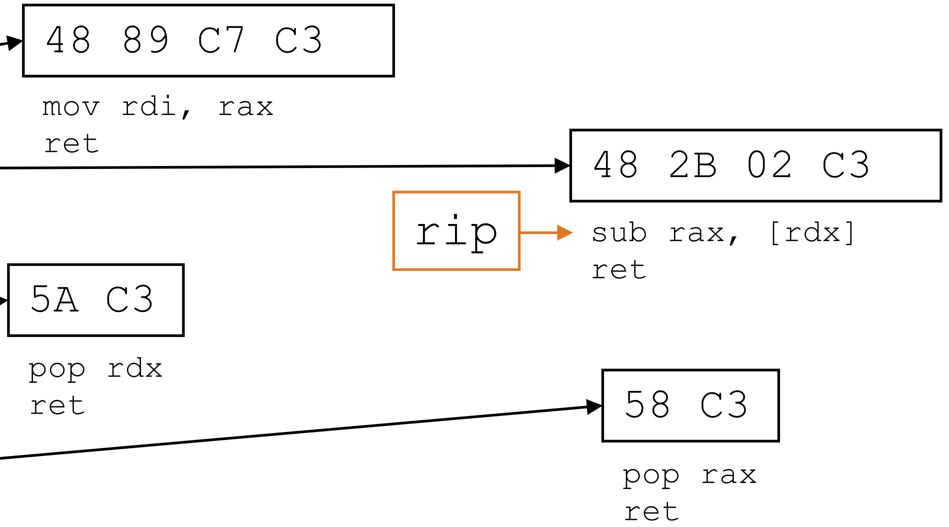
Practice with Return-Oriented Programming (ROP)

What are the values in the registers when the function at address `0x401a82` gets called?

rdi	rsi	rax	rdx
		…97ff	…ea58

7fffffffea58	0x59b997d0
7fffffffea50	0x401a82
7fffffffea48	0x4002a3
7fffffffea40	0x401bb0
7fffffffea38	0x7fffffffea58
7fffffffea30	0x401a5b
7fffffffea28	0x59b997ff
7fffffffea20	0x4019e5

rsp

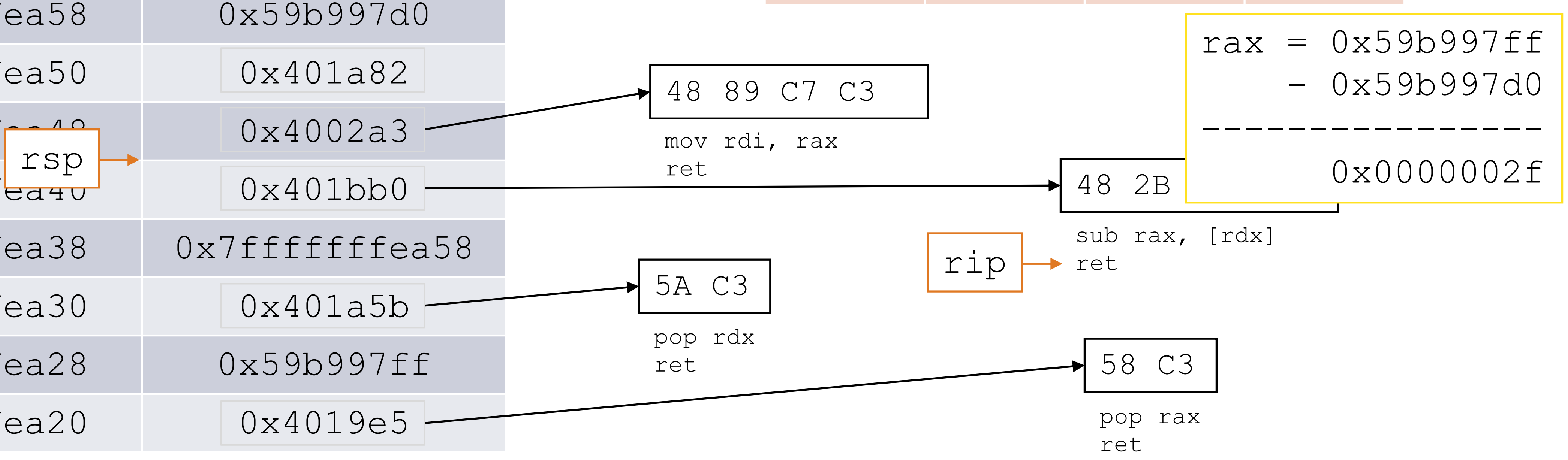


Practice with Return-Oriented Programming (ROP)

What are the values in the registers when the function at address `0x401a82` gets called?

rdi	rsi	rax	rdx
		2f	…ea58

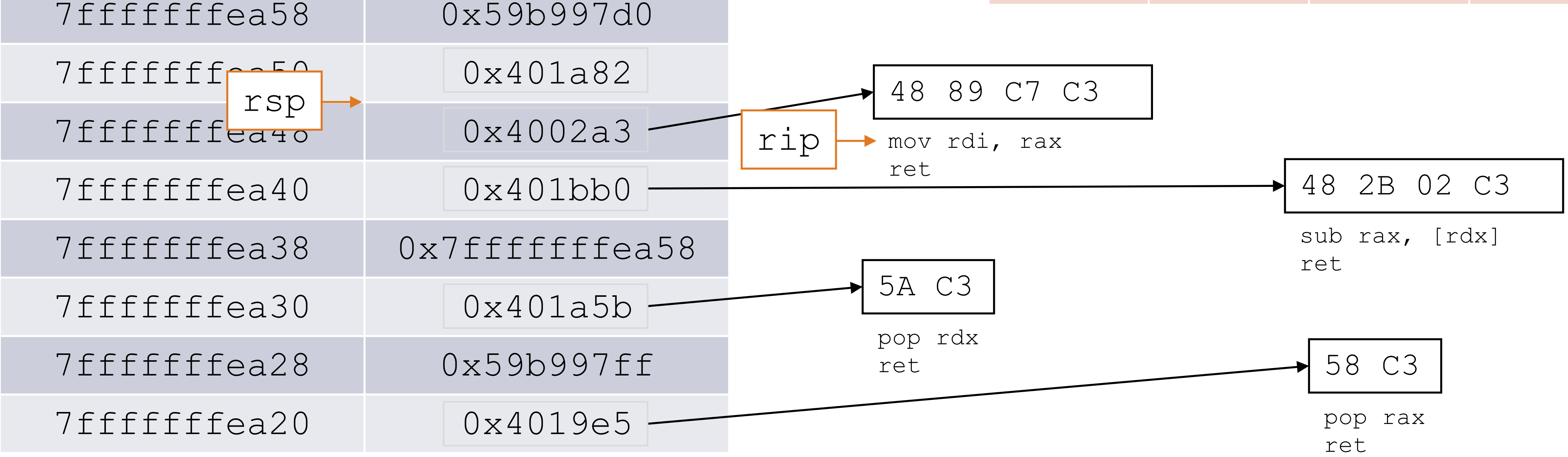
7fffffffea58	0x59b997d0
7fffffffea50	0x401a82
7fffffffea48	0x4002a3
7fffffffea40	0x401bb0
7fffffffea38	0x7fffffffea58
7fffffffea30	0x401a5b
7fffffffea28	0x59b997ff
7fffffffea20	0x4019e5



Practice with Return-Oriented Programming (ROP)

What are the values in the registers when the function at address `0x401a82` gets called?

rdi	rsi	rax	rdx
		2f	…ea58

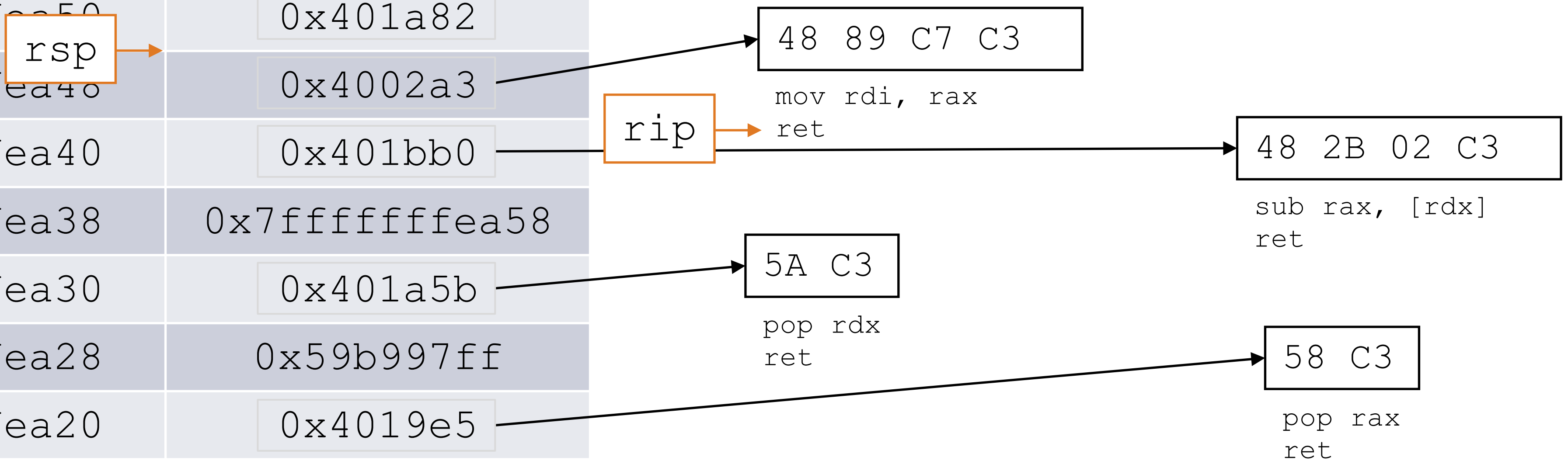


Practice with Return-Oriented Programming (ROP)

What are the values in the registers when the function at address `0x401a82` gets called?

rdi	rsi	rax	rdx
2f		2f	…ea58

7fffffffea58	0x59b997d0
7fffffffea50	0x401a82
7fffffffea48	0x4002a3
7fffffffea40	0x401bb0
7fffffffea38	0x7fffffffea58
7fffffffea30	0x401a5b
7fffffffea28	0x59b997ff
7fffffffea20	0x4019e5

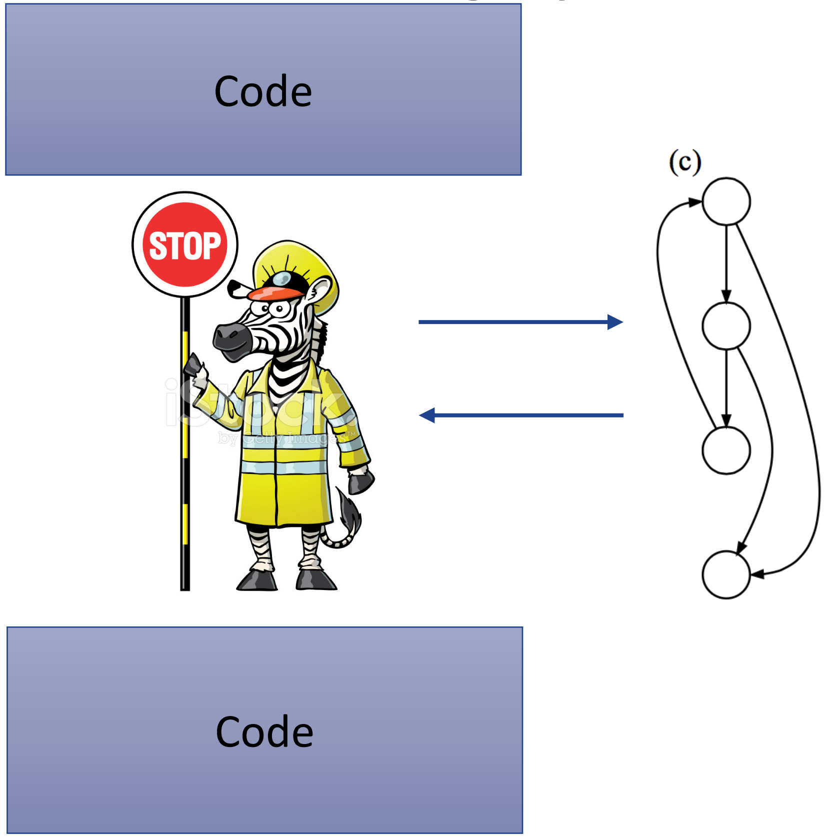


Defense #5: Address Space Layout Randomization

Other defenses

Gadget Elimination

Control Flow Integrity



The state of the world

Defenses:

- Use high-level languages
- Stack Canaries
- Memory tagging
- Address Space Layout Randomization
- Continuing research and development.

But they aren't perfect!