Buffers

```
void say_hello() {
  char name[10];
  gets(name);
  printf("hello %s\n", name);
}
```

Buffers

A **buffer** receives data from a function call

```
void say_hello() {
  char name[10];
  gets(name);
  printf("hello %s\n", name);
}
```

Buffers

Usually a byte array

A **buffer** receives data from a function call

```
void say_hello() {
  char name[10];
  gets(name);
  printf("hello %s\n", name);
}
```

What happens if the given name is longer than 10 bytes?

Buffers

Usually a byte array

A **buffer** receives data from a function call

```
void say_hello() {
  char name[10];
  gets(name);
  printf("hello %s\n", name);
}
```

What happens if the given name is longer than ~~10~~ 9 bytes?

That's a **buffer overflow**

The `gets` Man Page

FGETS(3) Library Functions Manual FGETS(3)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

char *
fgets(char * restrict str, int size, FILE * restrict stream);

char *
gets(char *str);
```

NAME

fgets, **gets** – get a line from a stream

[....]

SECURITY CONSIDERATIONS

The gets() function cannot be used securely. [....]

The `gets` Man Page

```
FGETS(3)          Library Functions Manual       FGETS(3)

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdio.h>

     char *
     fgets(char * restrict str, int size, FILE * restrict stream);

     char *
     gets(char *str);

NAME
     fgets, gets - get a line from a stream

[....]

SECURITY CONSIDERATIONS
     The gets() function cannot be used securely. [....]
```

Using `fgets` Instead

Problem solved?

```
void say_hello() {
  char name[10];
  fgets(name, 64, stdin);
  printf("hello %s\n", name);
}
```

Using `fgets` Instead

Problem solved?

```
void say_hello() {
  char name[10];
  fgets(name, 64, stdin);
  printf("hello %s\n", name);
}
```

Bad buffer size

Using `fgets` Instead

```
void say_hello() {
  char name[10];
  fgets(name, 10, stdin);
  printf("hello %s\n", name);
}
```

Using `fgets` Instead

```
void say_hello() {
  char name[10];
  fgets(name, 10, stdin);
  printf("hello %s\n", name);
}
```

Beware of varying null-terminator conventions

Bugs and Security

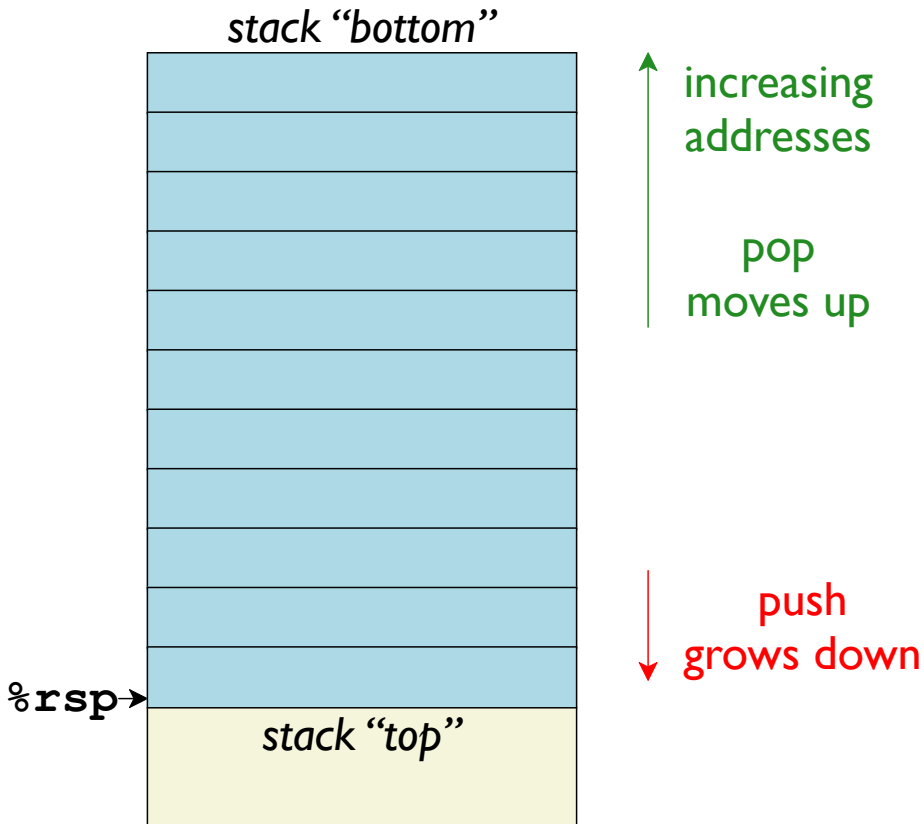
Using `gets` is bad for correctness, but why is it a *security* issue?

General reason: *undefined* behavior in C/C++ means *anything can happen*

- crash
- random file corruption
- password exposure

Specific reason: overflow of a buffer as local opens the door to a **buffer overflow** attack

C Stack


stack "bottom"
increasing addresses
pop moves up
push grows down
%rsp
stack "top"

Function Calls in C

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```

Function Calls in C

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

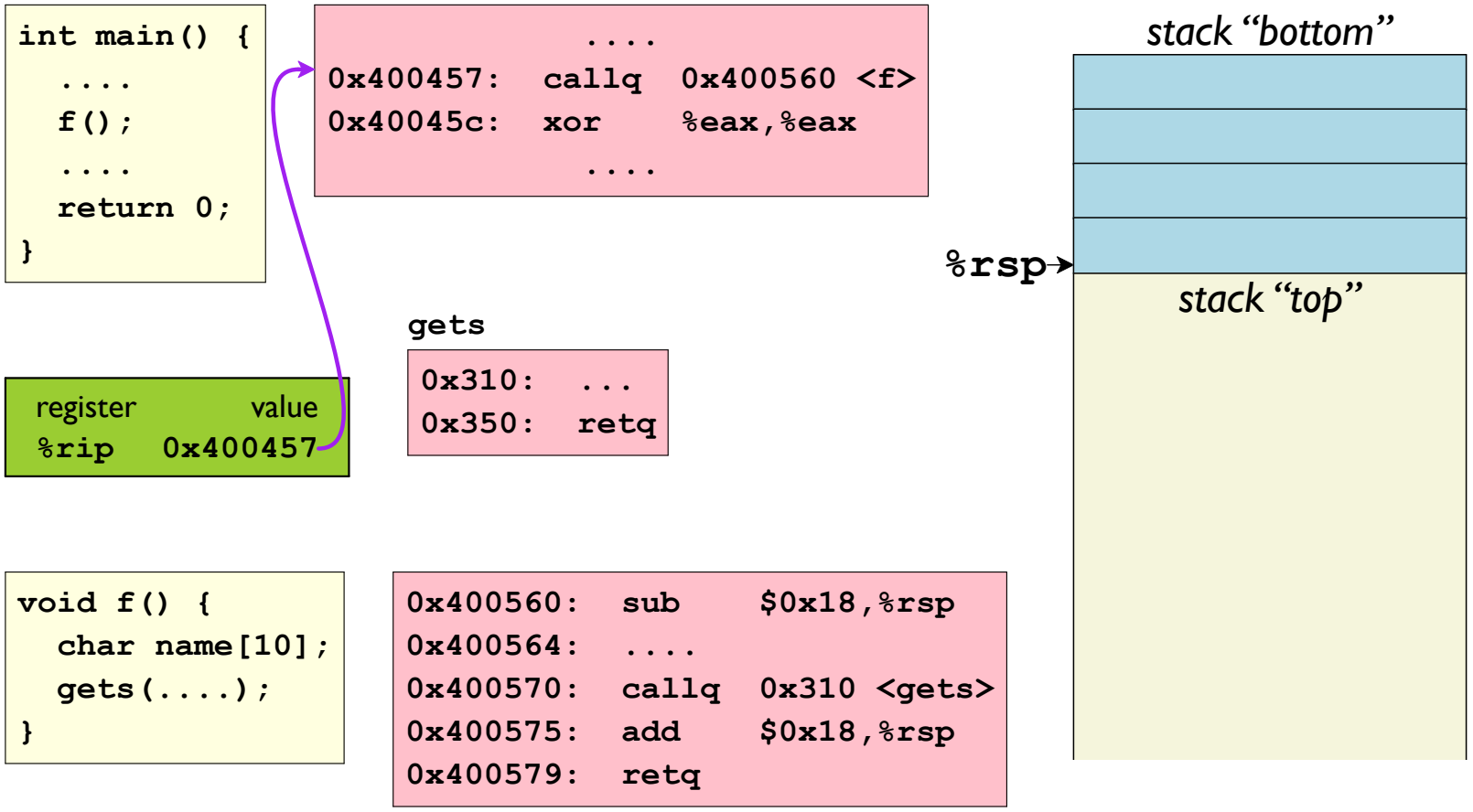
gets

```
0x310:  ...
0x350:  retq
```

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```

Function Calls in C



Function Calls in C

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

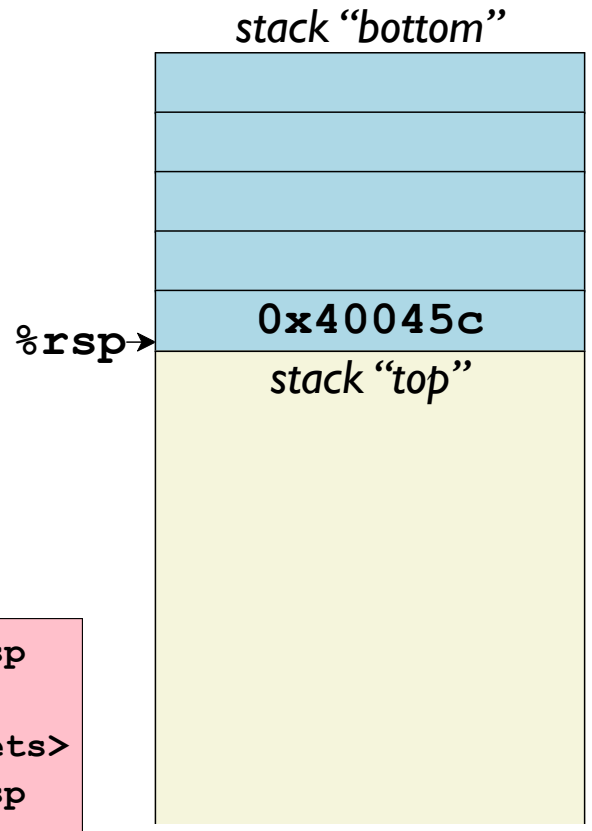
gets

```
0x310:  ...
0x350:  retq
```

register	value
%rip	0x400560

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



Function Calls in C

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

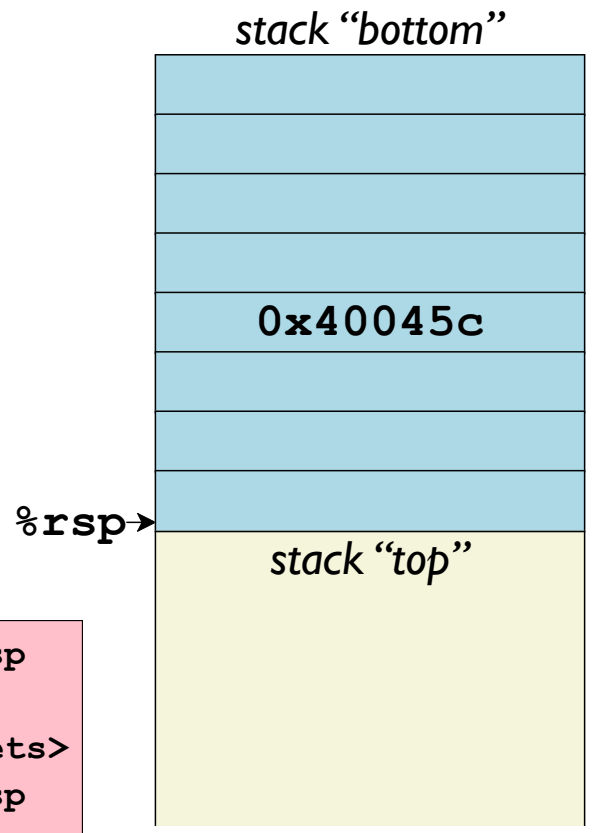
gets

```
0x310:  ...
0x350:  retq
```

register	value
%rip	0x400564

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



Function Calls in C

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

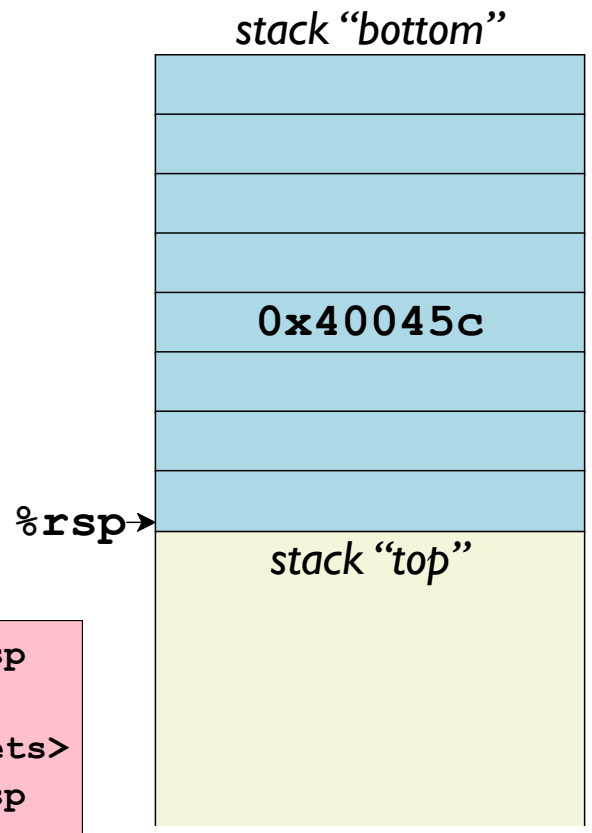
gets

```
0x310:  ...
0x350:  retq
```

register	value
%rip	0x400570

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



Function Calls in C

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

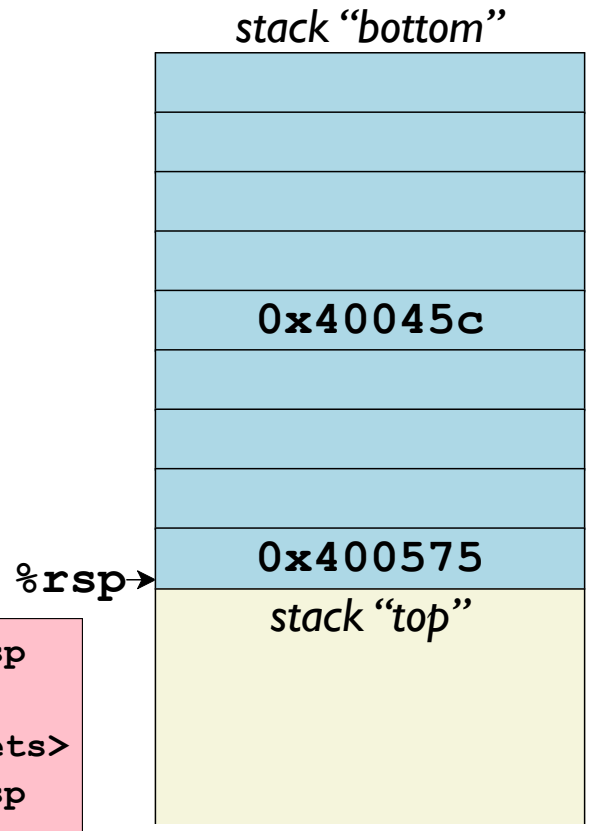
gets

```
0x310:  ...
0x350:  retq
```

register	value
%rip	0x310

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



Function Calls in C

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

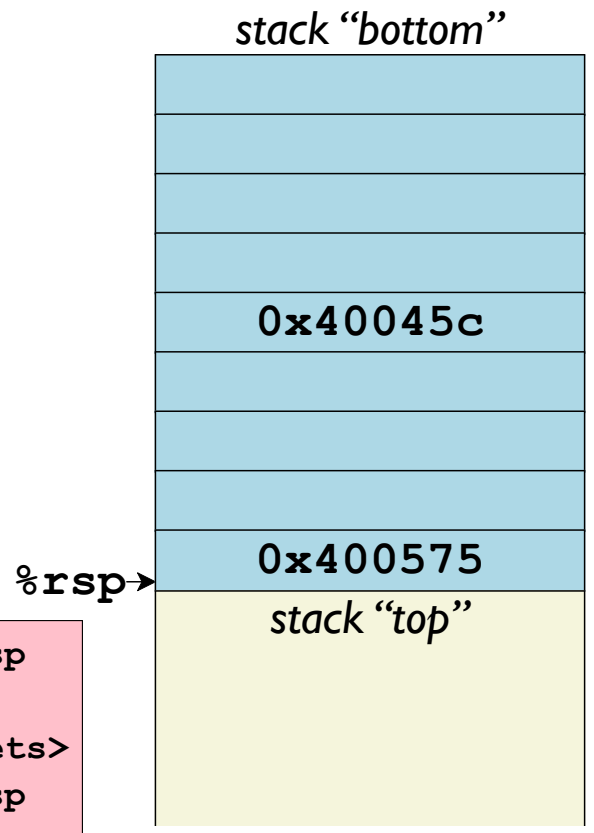
gets

```
0x310:  ...
0x350:  retq
```

register	value
%rip	0x350

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



Function Calls in C

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

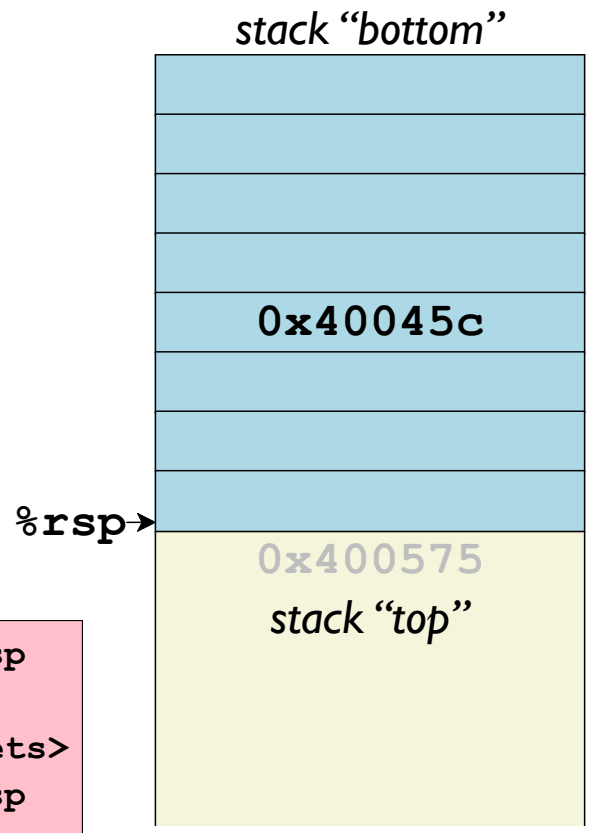
gets

```
0x310:  ...
0x350:  retq
```

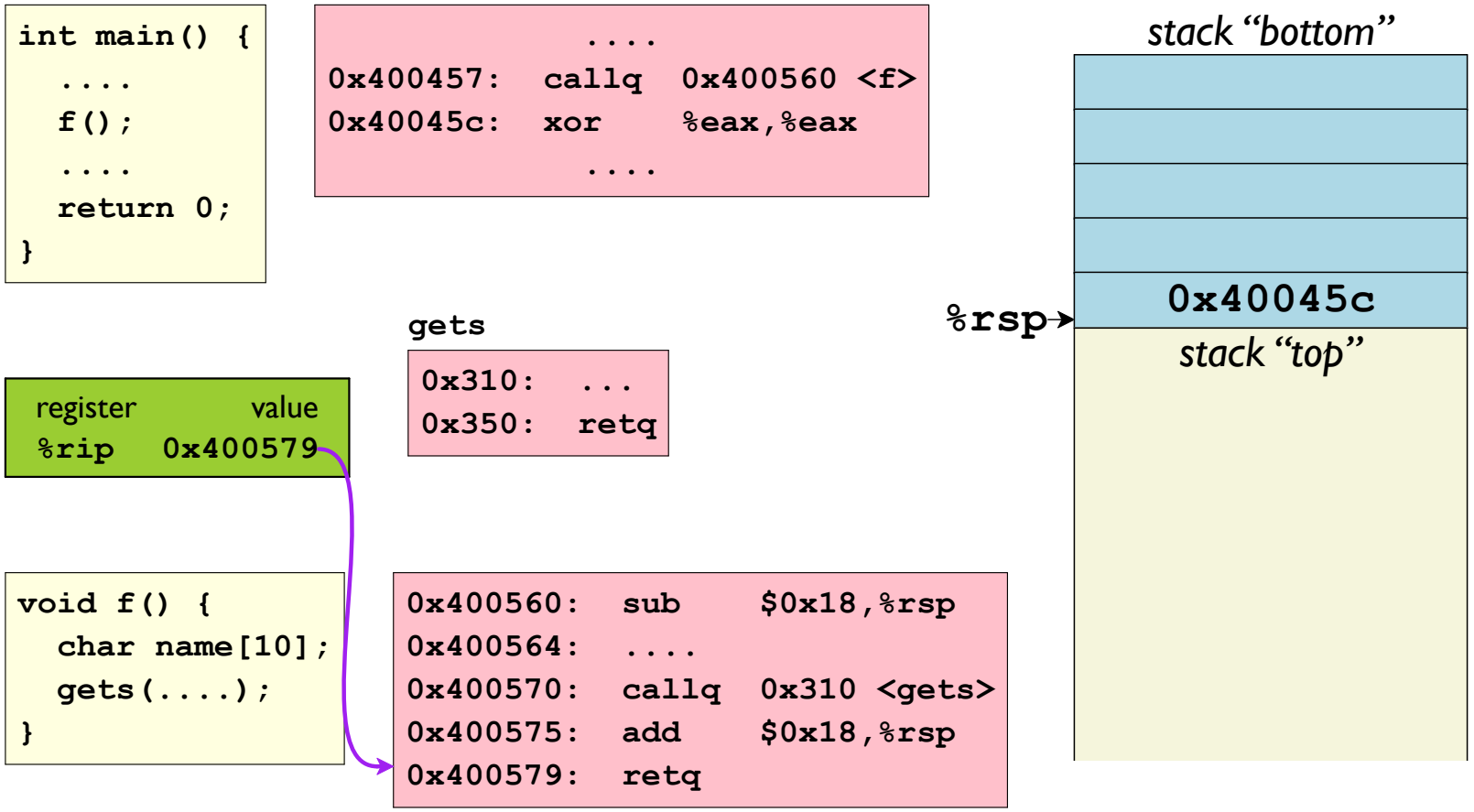
register	value
%rip	0x400575

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



Function Calls in C



Function Calls in C

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

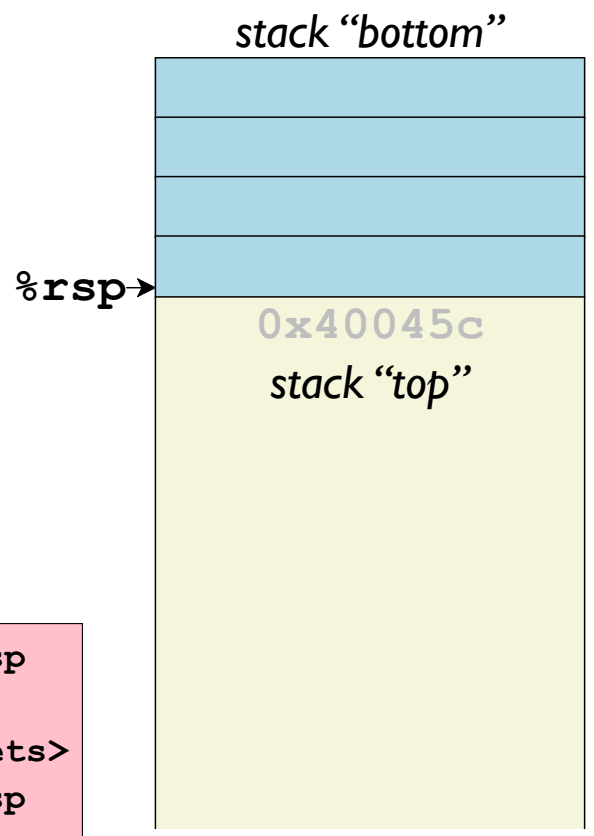
register value
%rip 0x40045c

gets

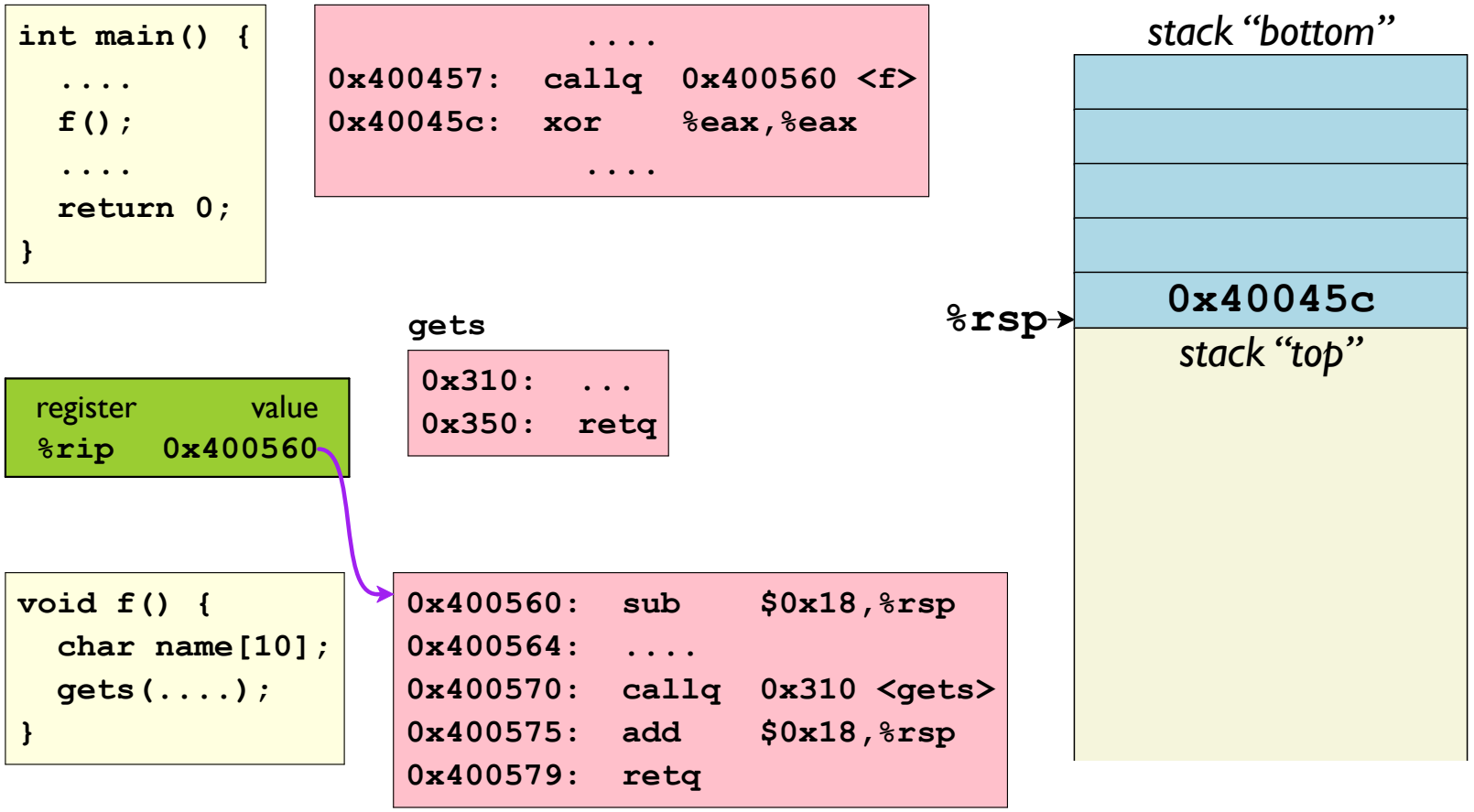
```
0x310:  ...
0x350:  retq
```

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



Local Buffers and Overflow



Local Buffers and Overflow

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

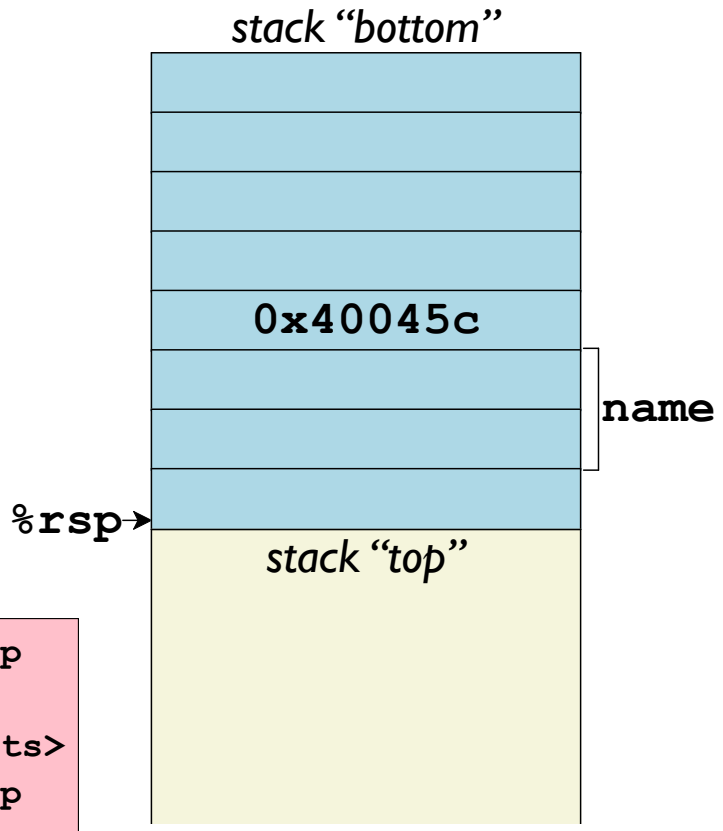
gets

```
0x310:  ...
0x350:  retq
```

register	value
%rip	0x400564

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



Local Buffers and Overflow

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

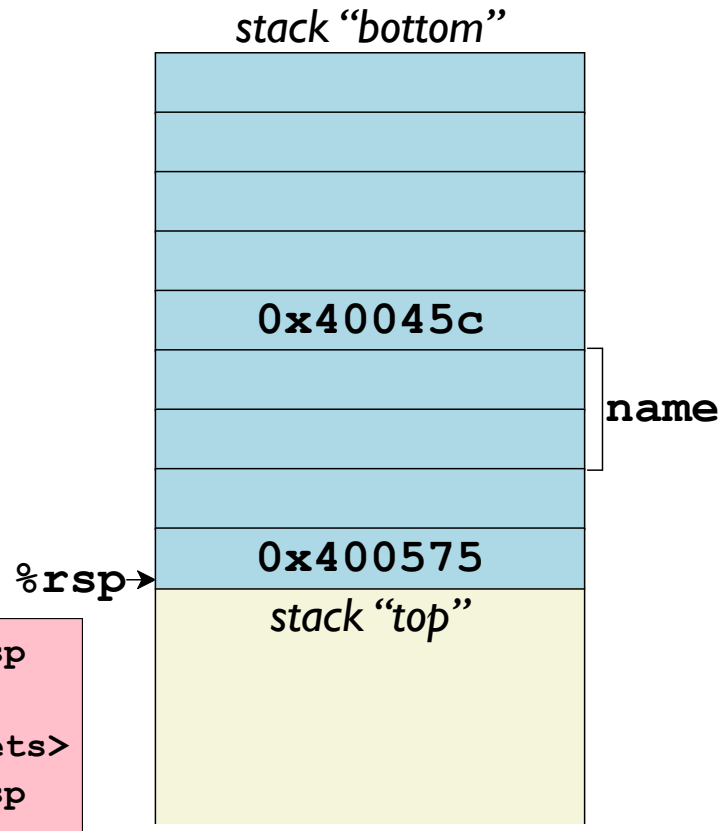
gets

```
0x310:  ...
0x350:  retq
```

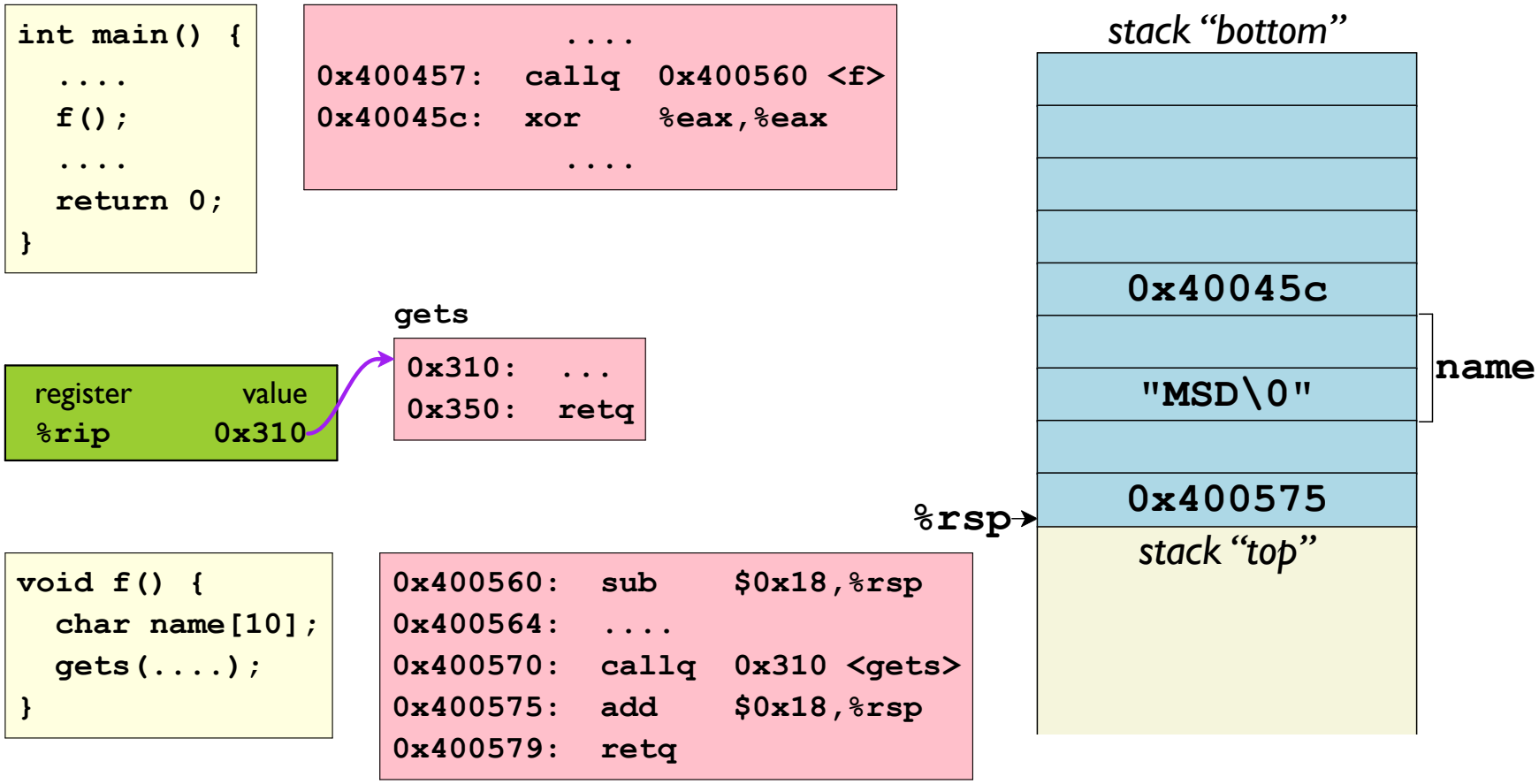
register	value
%rip	0x310

```
void f() {
  char name[10];
  gets(....);
}
```

```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



Local Buffers and Overflow



Local Buffers and Overflow

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

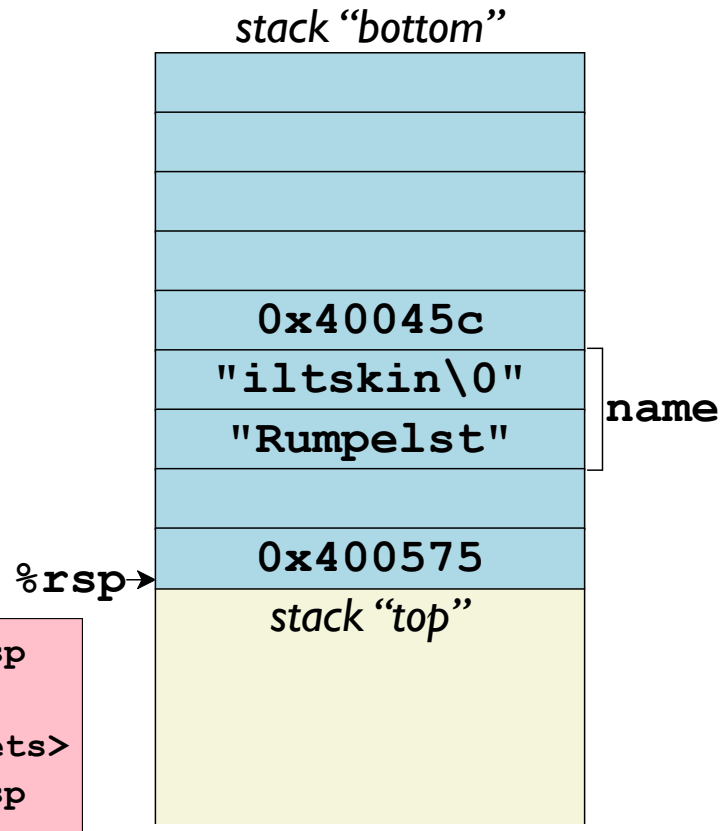
gets

```
0x310:  ...
0x350:  retq
```

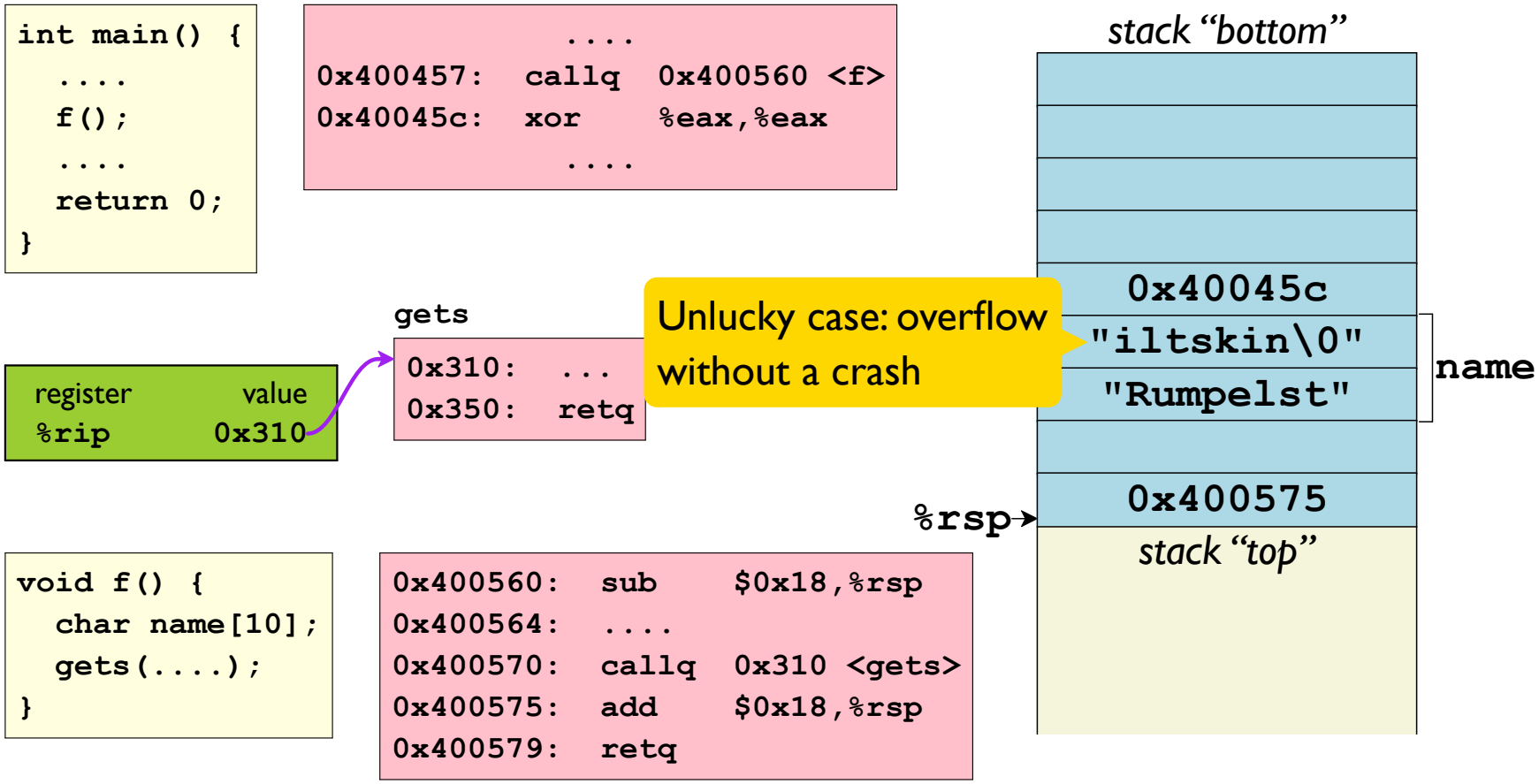
register	value
%rip	0x310

```
void f() {
  char name[10];
  gets(....);
}
```

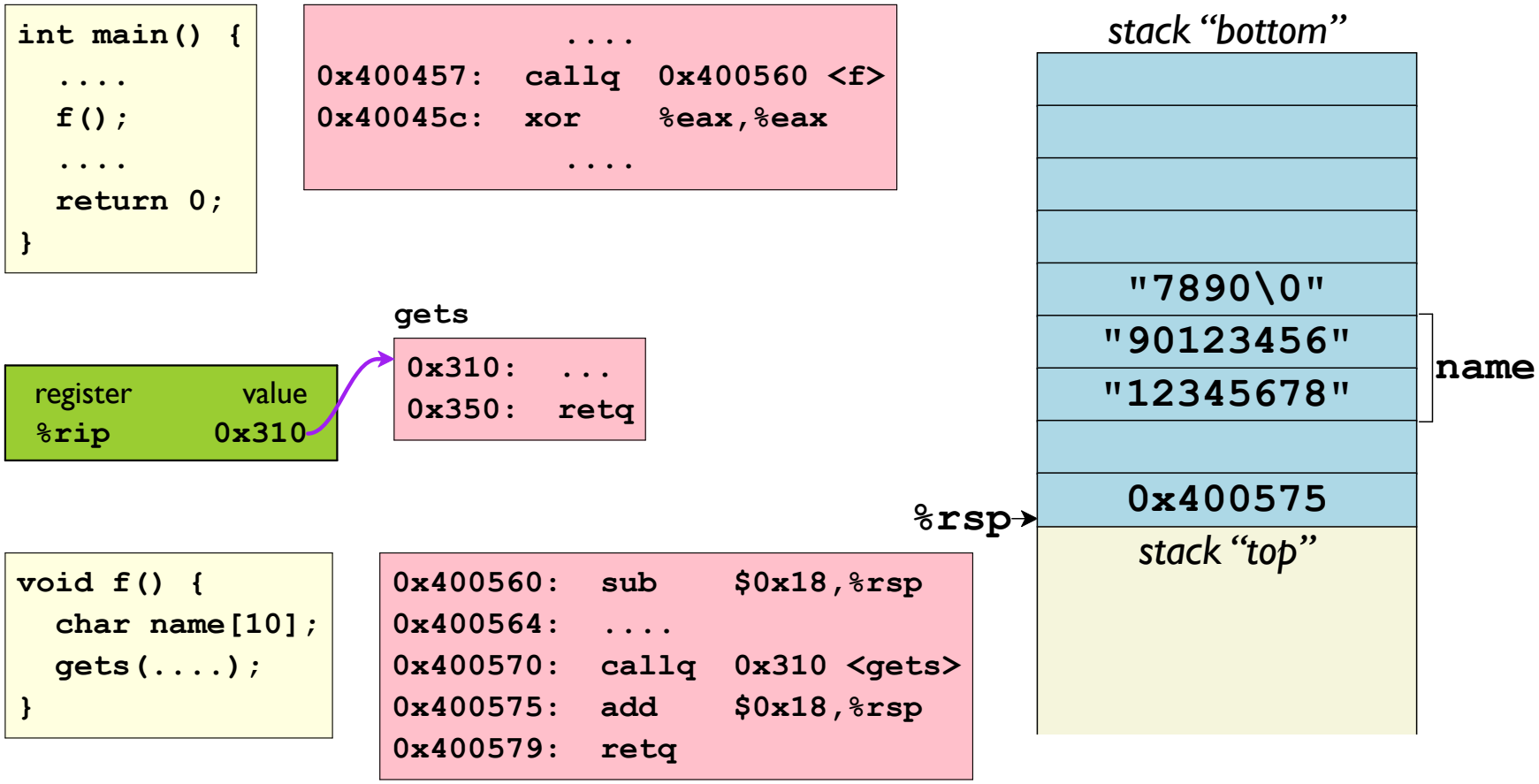
```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



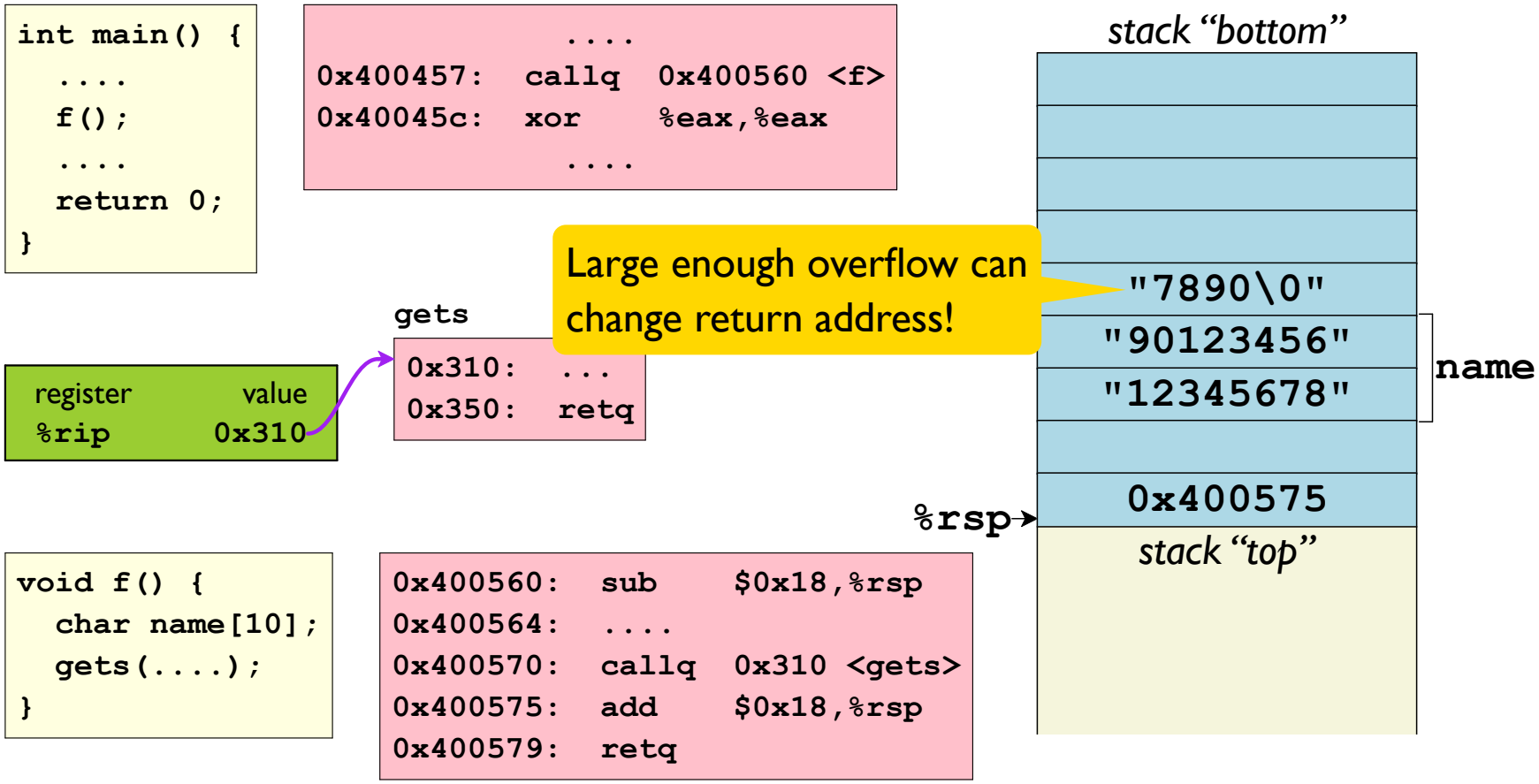
Local Buffers and Overflow



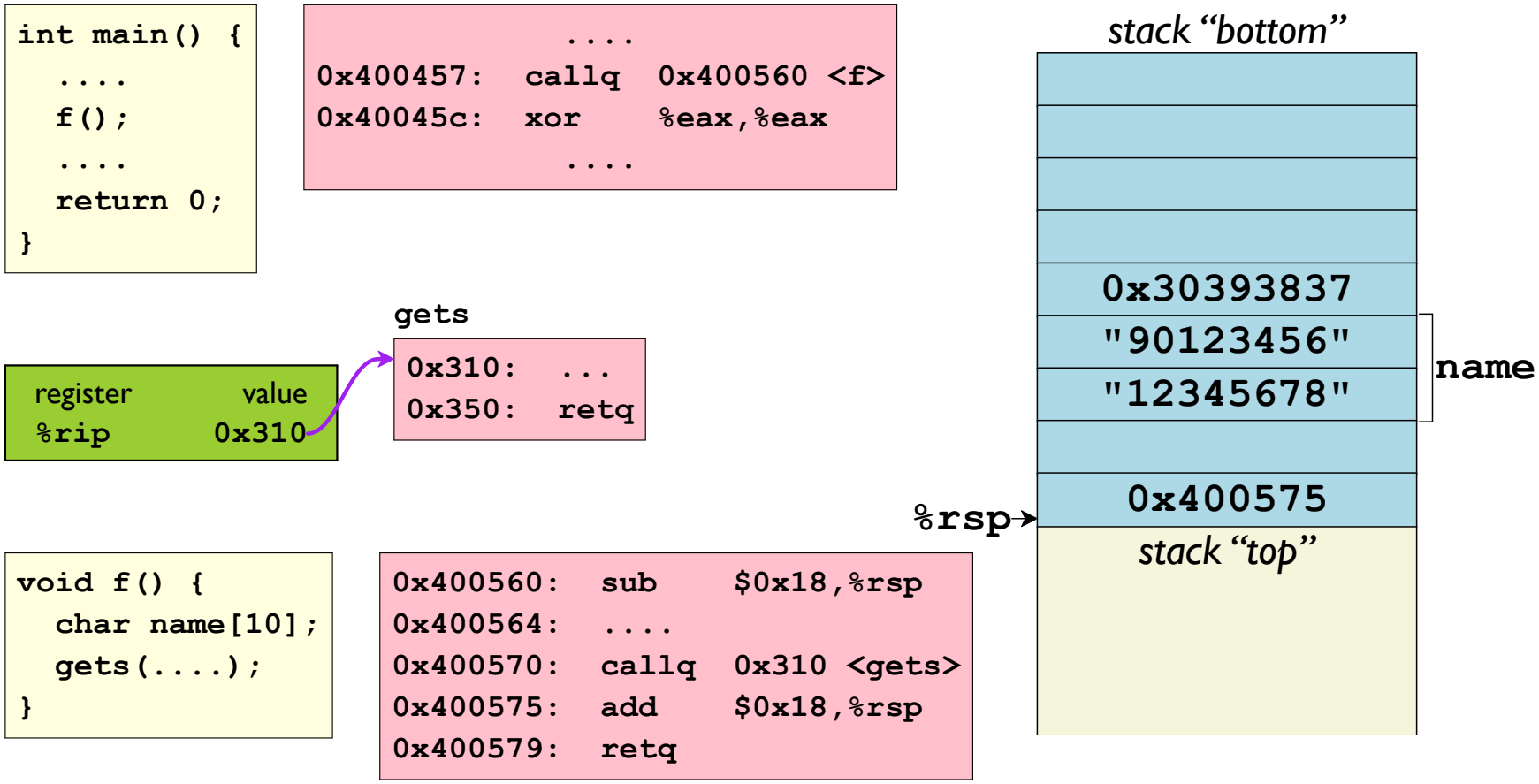
Local Buffers and Overflow



Local Buffers and Overflow



Local Buffers and Overflow



Local Buffers and Overflow

```
int main() {
  ....
  f();
  ....
  return 0;
}
```

```
                ....
0x400457:  callq  0x400560 <f>
0x40045c:  xor    %eax,%eax
                ....
```

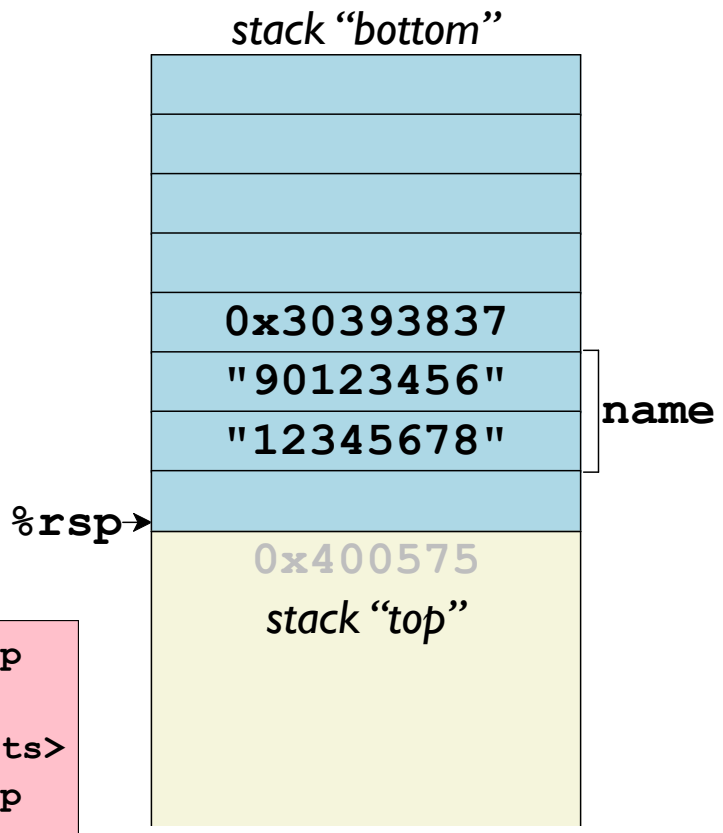
gets

```
0x310:  ...
0x350:  retq
```

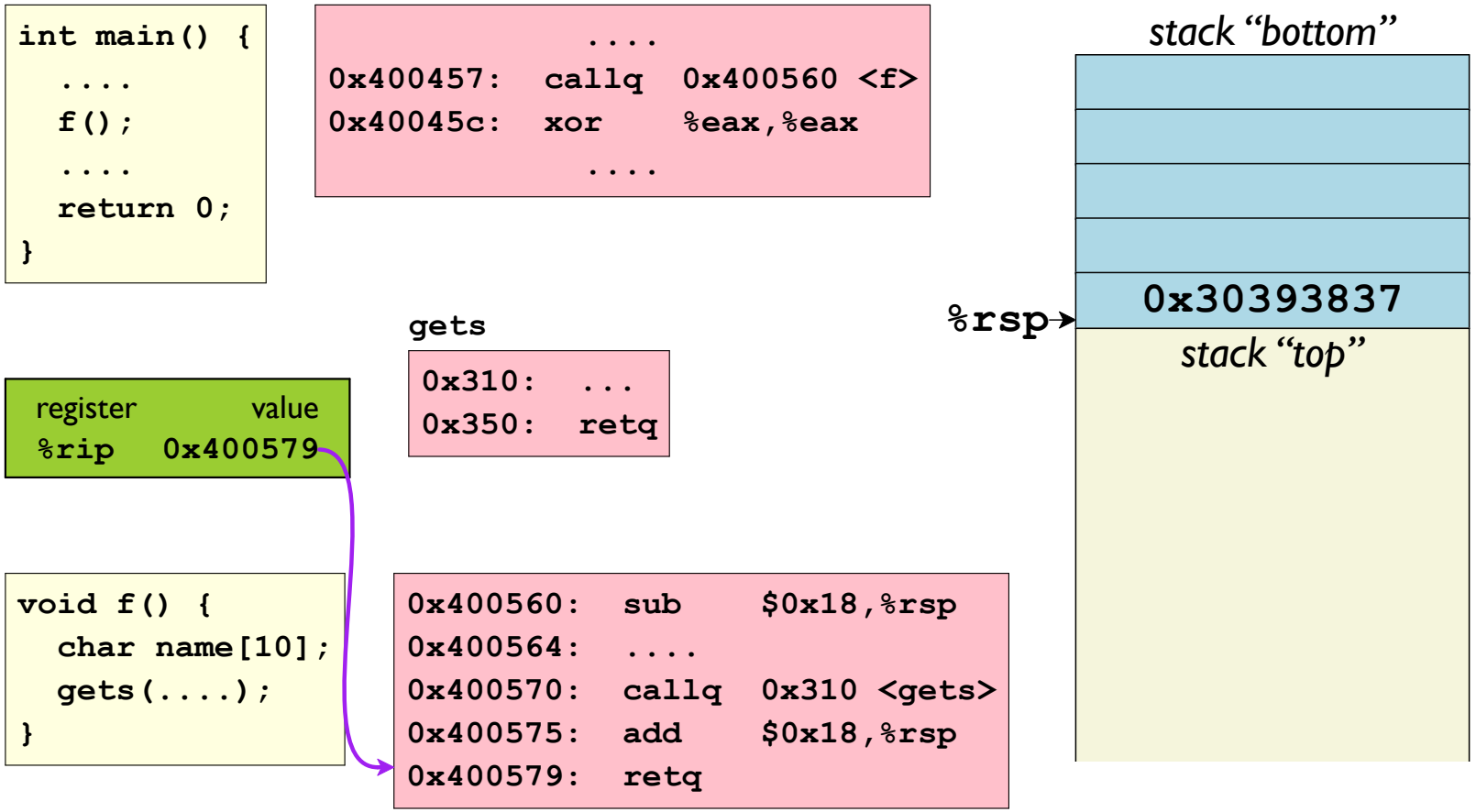
register	value
%rip	0x400575

```
void f() {
  char name[10];
  gets(....);
}
```

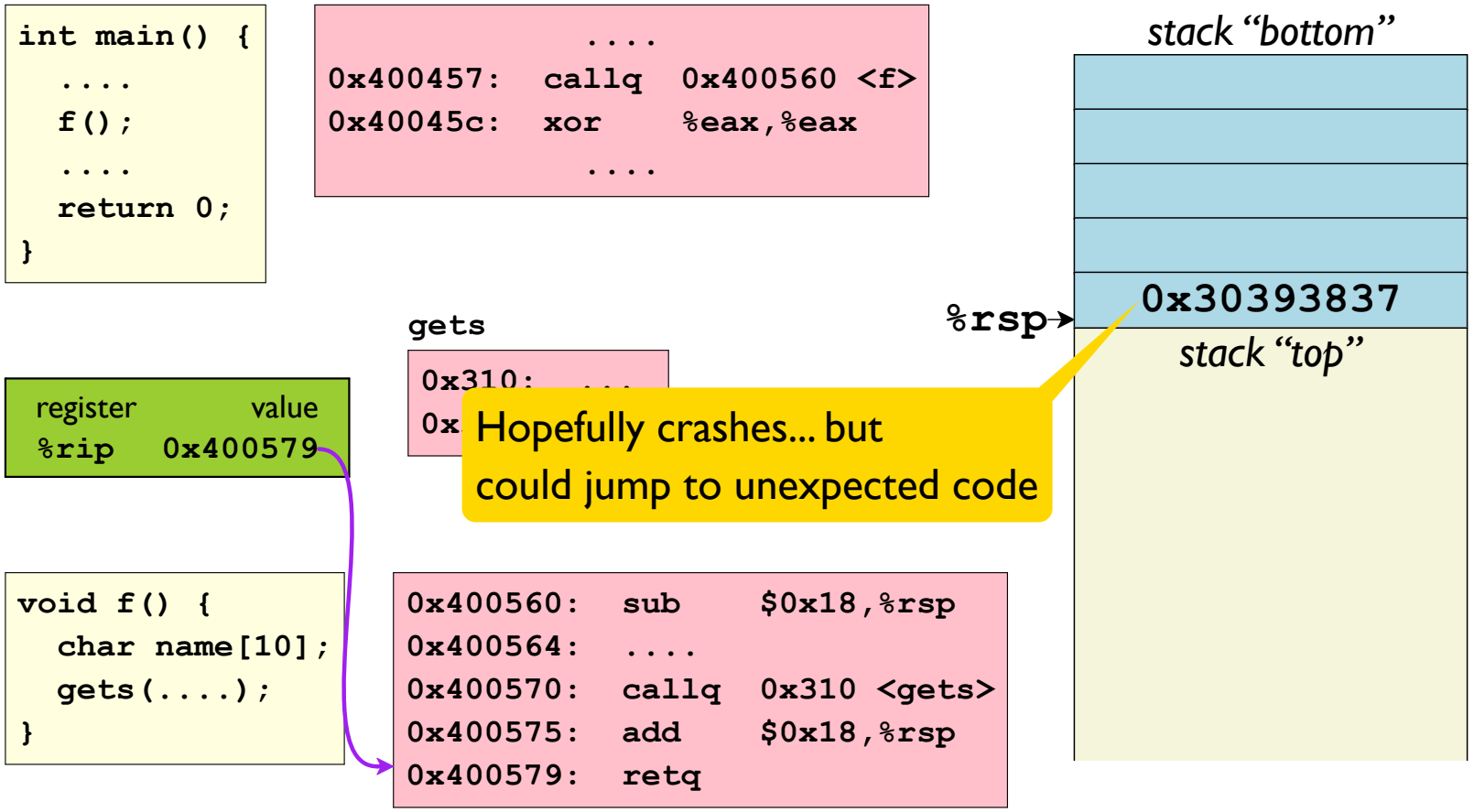
```
0x400560:  sub    $0x18,%rsp
0x400564:  ....
0x400570:  callq  0x310 <gets>
0x400575:  add    $0x18,%rsp
0x400579:  retq
```



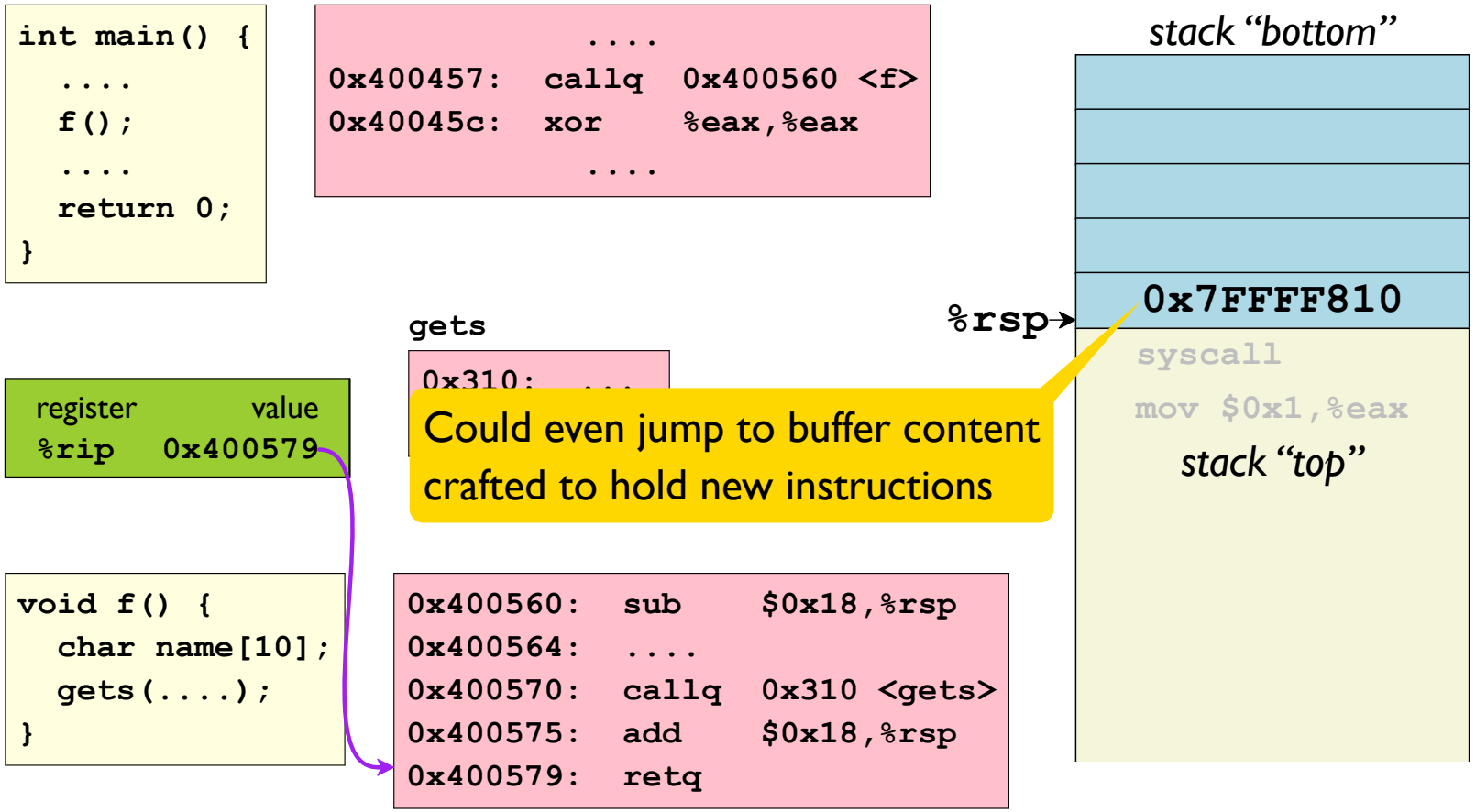
Local Buffers and Overflow



Local Buffers and Overflow



Local Buffers and Overflow



Buffer Overflow Insecurity

Crafting an input string to exploit a buffer overflow is a **buffer overflow attack**

Possible strategies:

- jumping to known code
- running externally supplied machine code
- stringing together existing code fragments as "gadgets"

Possible consequences:

- circumventing a check by jumping past it
- running arbitrary machine code in the host process
- using the `exec` system call to start `/bin/sh`

Buffer Overflow Insecurity

Crafting an input string to exploit a buffer overflow is a **buffer overflow attack**

Possible strategies:

- jumping to known code
- running externally supplied machine code
- stringing together existing code fragments as "gadgets"

Possible consequences:

- circumventing a check by jumping past it
- running arbitrary machine code in the host proc
- using the `exec` system call to start `/bin/sh`

Even better if the compromised process runs a `setuid` program

Buffer Overflow Defenses

Strategy 1: write bug-free code

Strategy 2: use a safe language, or at least a safer API

Java, C#, Rust, Go, etc.: overflow attack is not possible

C++: `std::string` at least avoids many pitfalls

Strategy 3: adjust the compiler or OS to make attacks more difficult

process limits, stack guards, ASLR, page protection and W^X

Process Limits

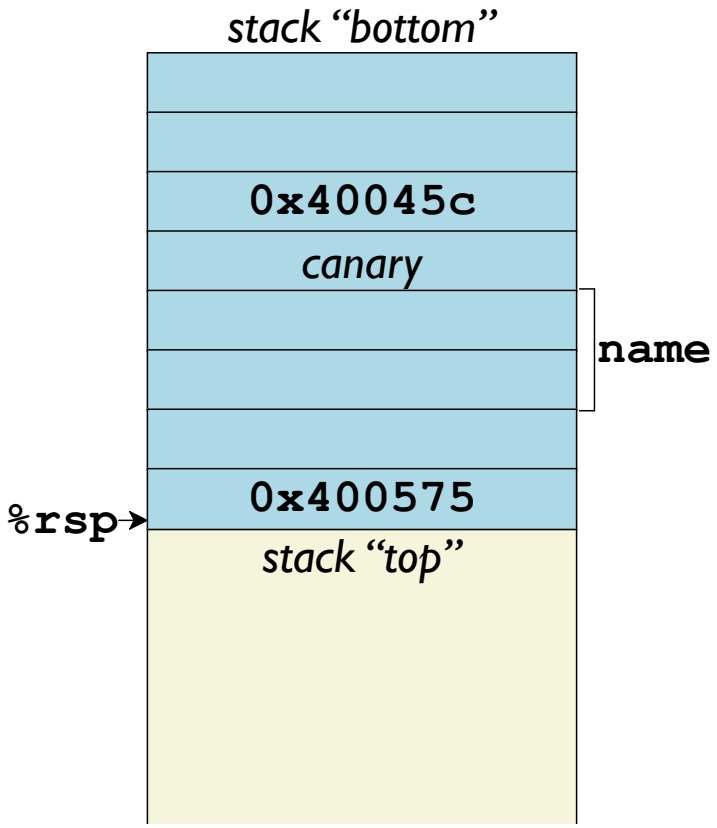
Every process runs as a particular user

e.g., file access granted to specific users and groups

- Set up service-specific accounts with limited permissions
- Avoid `sudo` and other `setuid` programs

OpenBSD's `pledge` system call can restrict capabilities further

Compiling with a Stack Guard



Compiling with a Stack Guard

Compiling `say_hello` with `-fstack-protector` (default on macOS):

```
say_hello:
    subq    $40, %rsp
    movq    %fs:40, %rax        # load canary
    movq    %rax, 32(%rsp)      # store canary
    leaq    22(%rsp), %rdi
    movq    %rdi, 8(%rsp)
    callq   gets@PLT
    movq    8(%rsp), %rsi
    leaq    .L.str(%rip), %rdi
    xorl    %eax, %eax
    callq   printf@PLT
    movq    %fs:40, %rax        # get canary again
    movq    32(%rsp), %rcx      # get stored value
    cmpq    %rcx, %rax          # check canary intact
    jne     .LBB0_2
    addq    $40, %rsp
    retq
.LBB0_2:
    callq   __stack_chk_fail@PLT
.L.str:
    .asciz  "hello %s\n"
```

Address Randomization

Address space layout randomization (ASLR) causes code and the stack to have a different address on every run

- Prevents easy construction of overflow strings
- Doesn't require recompiling existing programs
- Enabled by default on all modern OSes

```
#include <stdio.h>
int main() {
  printf("%p\n", main);
  return 0;
}
```

Page Protection

Modern hardware supports file-like permissions on each memory page: `read`, `write`, and `exec`

Stack pages lack `exec` permission, which blocks attempts to put machine code input buffers

W^X: a memory page can have either `write` or `exec`, but not both at the same time

Summary

Unsafe programming languages allow **buffer overflow** errors,
and those often turn into exploits

Mitigation strategies include compiler-inserted stack checks, ASLR, and page protection

Rest of this week: buffer-overflow exploit as an in-class lab