Buffer Overflow

A simple function

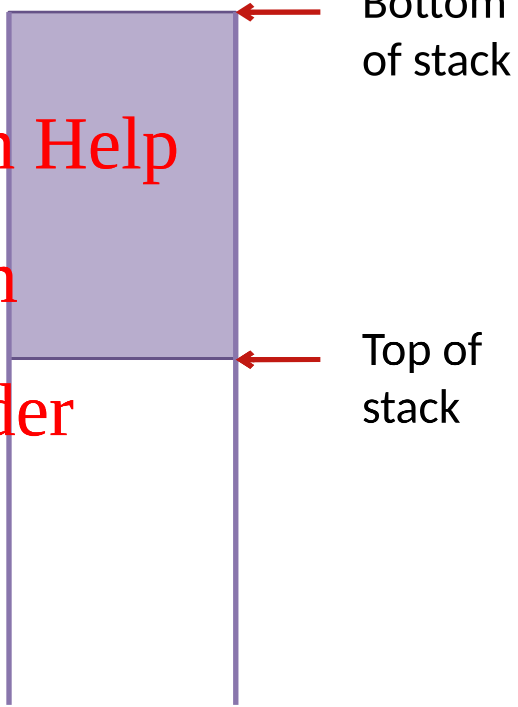
```
void f() {
  int i;
  int buf[9];

  for (i=0; i<5; i++)
    buf[4+i] = buf[4-i] = 0;
}
```

A simple function

```
void f() {
  int i;
  int buf[9];

  for (i=0; i < 10; i++)
    buf[4+i] = buf[4-i] = 0;
}
```

The call stack

- A data structure that stores information about function calls in a program
- In X86 the stack is bottom-up
  - The stack bottom is at a high address
  - The stack top is at a low address
- The stack grows towards lower addresses



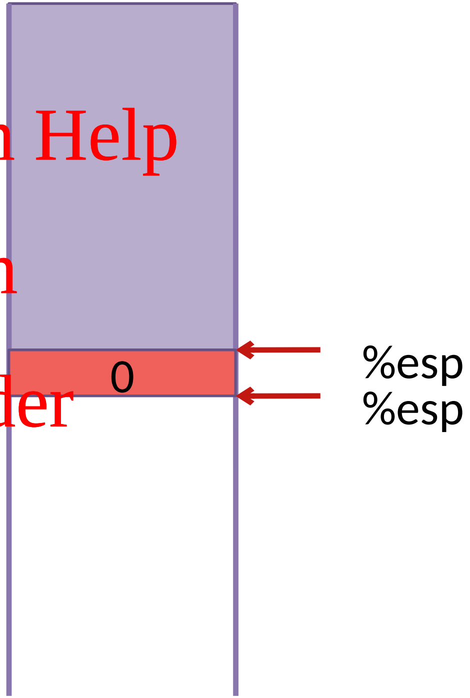
Implementation

- Register `%esp` points to the top of the stack
- The `push` instruction pushes a value onto the stack

```
xorl %eax,%eax
pushl %eax
```

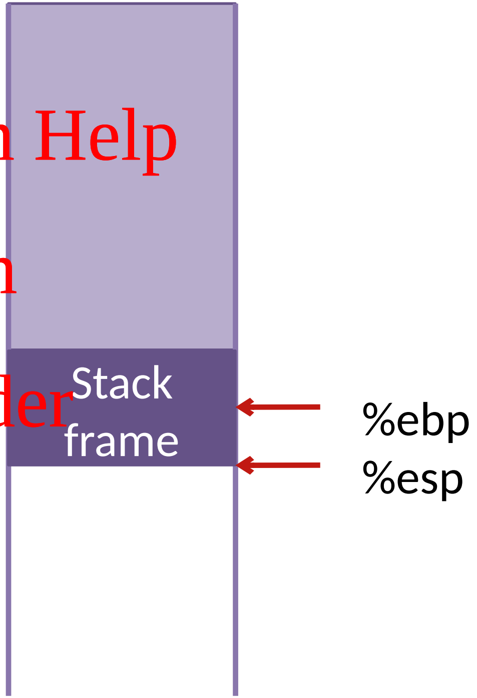
- `pop` pops a value

```
popl %eax
```



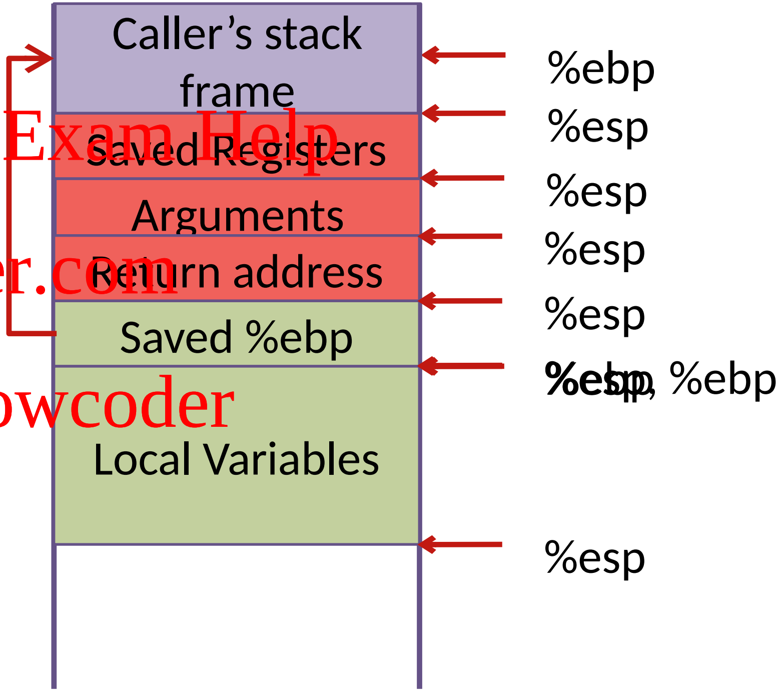
Calling a function

- Calling a function pushes a *stack frame* onto the stack
  - The *stack base pointer register* (`%ebp`) points to the frame of the current function
- Return pops the stack frame



Calling conventions

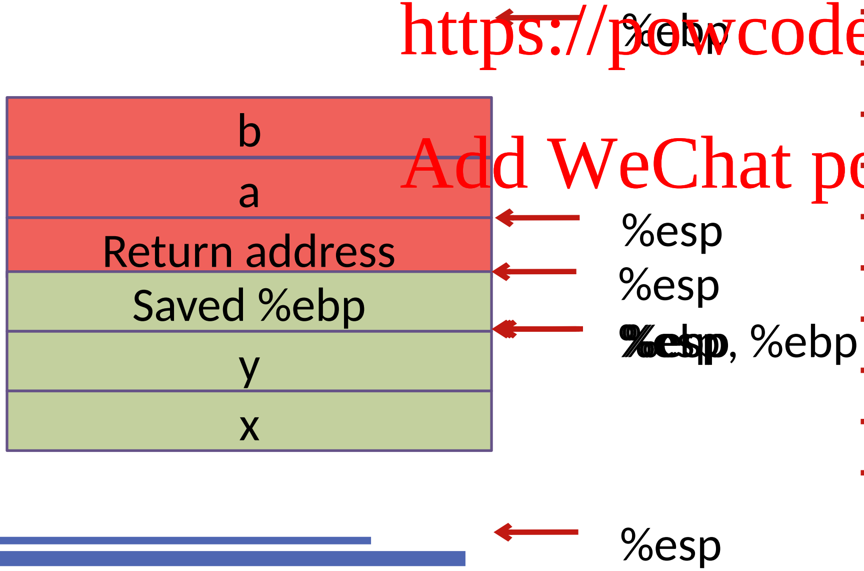
- Caller does:
  - Save registers
  - Push arguments
  - Call function
- Callee does
  - Save `%ebp`
  - Set new `%ebp`
  - Create space for local variables



Example

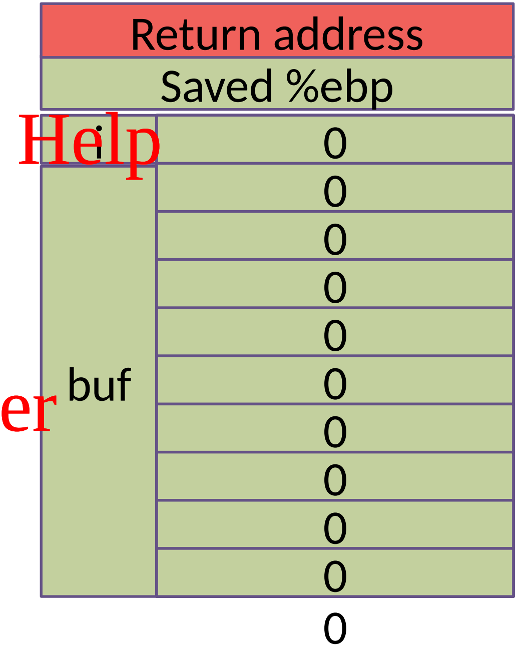
```
int g(int a, int b) {
  int x = a + 1;
  int y = b + 2;

  return x*y;
}
```

```
g:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    8(%ebp), %eax
    addl    $1, %eax
    movl    %eax, -8(%ebp)
    movl    12(%ebp), %eax
    addl    $2, %eax
    movl    %eax, -4(%ebp)
    movl    -8(%ebp), %eax
    imull   -4(%ebp), %eax
    leave
    ret
```



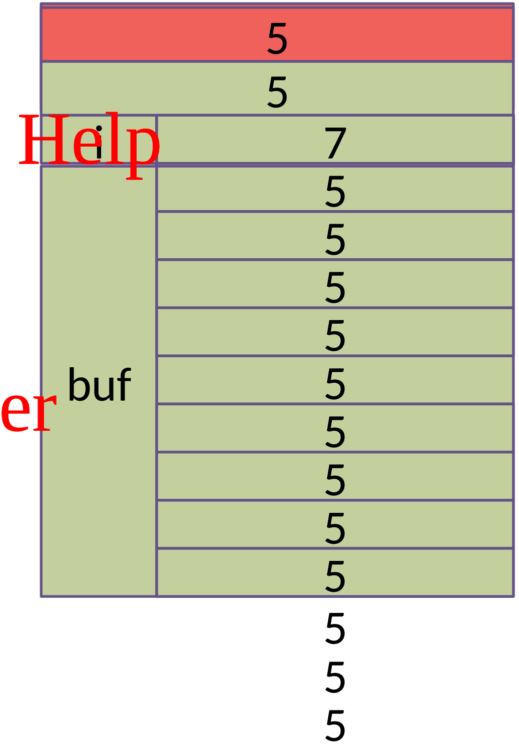
Back to a simple function

```
void f() {
  int i;
  char buf[9];

  for (i=0; i < 10; i++)
    buf[4+i] = buf[4-i] = 0;
}
```



With a minor change

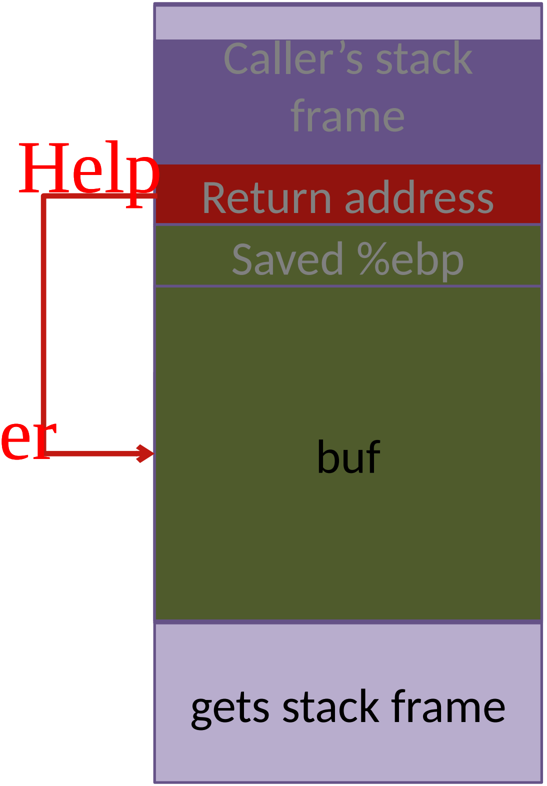
```
void f() {
  int i;
  char buf[9];

  for (i=0; i < 10; i++)
    buf[4+i] = buf[4-i] = 5;
}
```



Stack smashing

```
void f() {
   char buf[512];
   gets(buf);
   doSomething(buf);
}
```

- The attacker diverts execution to data it injected
- How does the attacker know where to jump to?



NOP Sled

- A sequence of NOP instructions leading to the attack code

```
NOP
NOP
NOP
.
.
.
NOP
NOP
Attack
Code
```

Problem patterns

- Any use of `gets`

- `strcpy`, `sprintf`, `strcat`, etc.

```
sprintf(buf, "https://%s/index.html", argv[1])

buf=new char[strlen(argv[1])]
strcpy(buf, argv[1])

wchar_t buf[MAXLEN];
swprintf(buf, sizeof(buf), "%s", argv[1]);
```

- Any low-level implementation of similar code

```
while (*src != ';')
   *dst++ = *src++;
*dst = '\0';
```

Avoiding buffer overflows

- Do not use `gets`.
- Replace unsafe C string functions with safe version
  - Redefine unsafe functions to catch use, for example:

```
char *strcpy(char *dst, const char *src) {
  fprintf(stderr, "Don't use strcpy\n");
  abort();
}
```

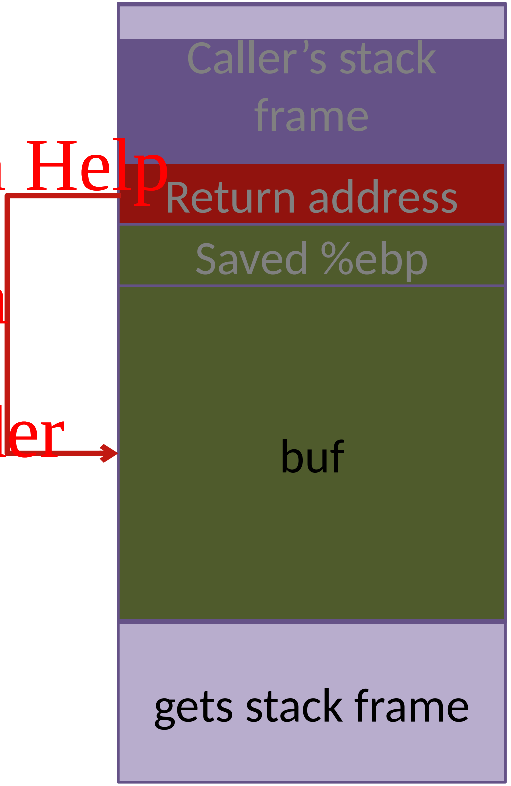
  - May fail if library functions use `strcpy`
- Replace C strings with safe(r) C++ strings

Avoiding buffer overflows - 2

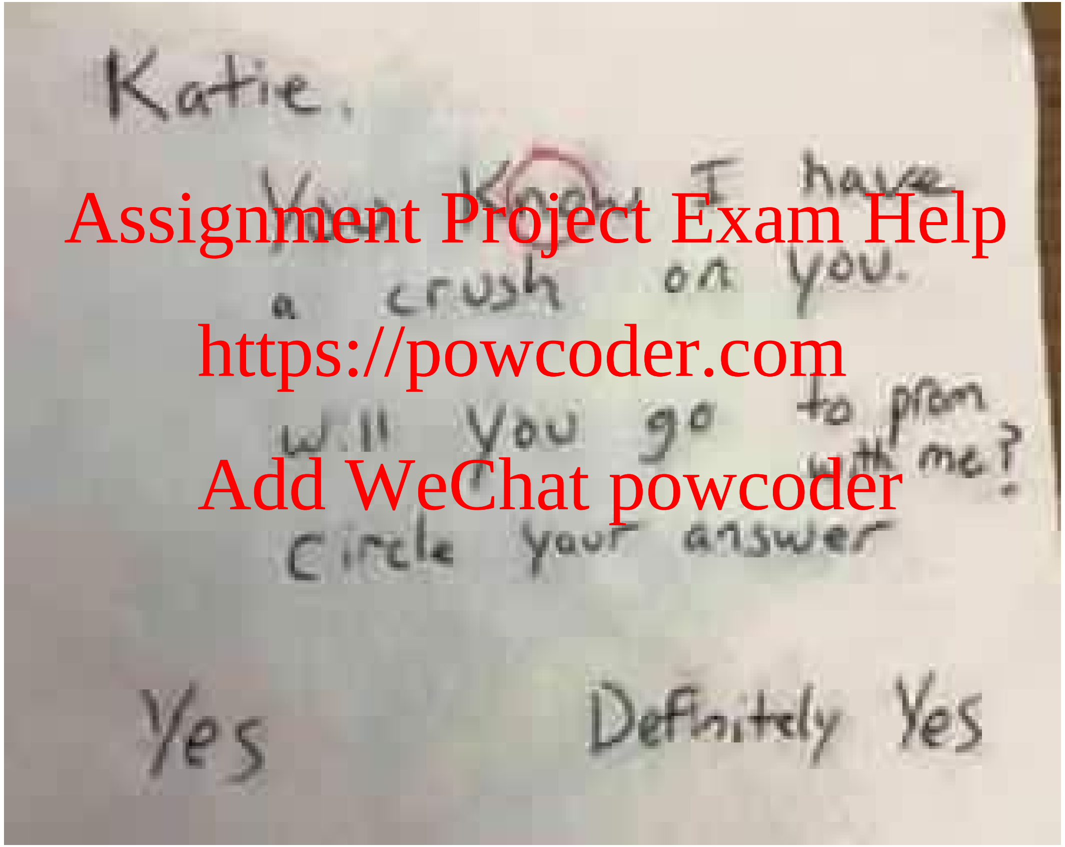
- Abstract over array access to include bounds checking
  - For example, use the C++ vector `.at()` method
  - What about performance?
- Code reviews and audits.
- Use static code analysis tools
- Switch to Java, C#, etc.

Non-executable stacks

- The stack is only used for data. There’s no need to run code from the stack
- The memory management unit can prevent code execution based on the address
- Only protects against branching back to the stack
- Does not prevent:
  - Heap overflow
  - Return Oriented Programming



ROP Illustrated



Katie,

You know I have a crush on you.

will you go to prom with me?

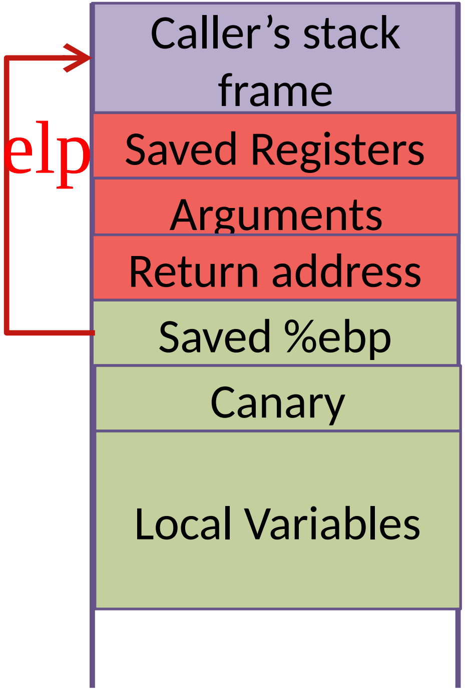
circle your answer

Yes Definitely Yes

StackGuard

- On function entry, callee
  - Saves `%ebp`
  - Sets new `%ebp`
  - Pushes the *canary*
  - Creates space for local variables
- Verify the canary on function exit

- The attacker has to overwrite the canary before changing the return address
- There are ways around the canary
- Does not protect from heap overflows, changing function pointers, etc.



Caller's stack frame

Saved Registers

Arguments

Return address

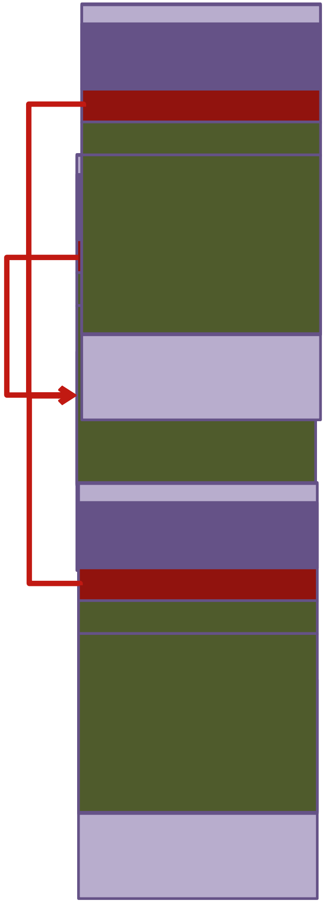
Saved %ebp

Canary

Local Variables

STack Overwrite Protection

- Push a large buffer to the stack at process initialisation
- The attacker does not know how to set the return address
- A large enough NOP-sled has a non-negligible probability of a success
- Only protects the stack
    - ASLR (Address Space Layout Randomization) extends protection to the heap and to libraries



Summary

- Buffer overflow is a common vulnerability
- Good coding practices often prevent overflows
- There are some systematic defence mechanisms
- **There's no silver bullet**