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;
     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
  - The stack bottom is at a high address
  - The stack top is at a low address
- The stack grows towards lower addresses

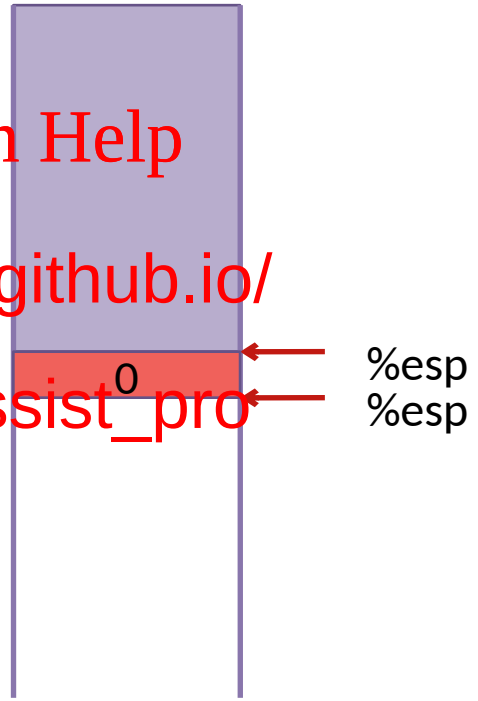
Implementation

- Register `%esp` points to the top of the stack
- The `push` instr value onto the

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

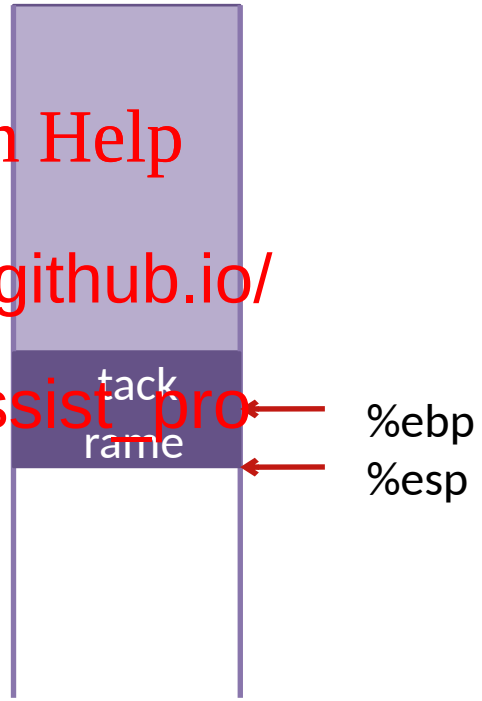
- `pop` pops a value

```
popl %eax
```

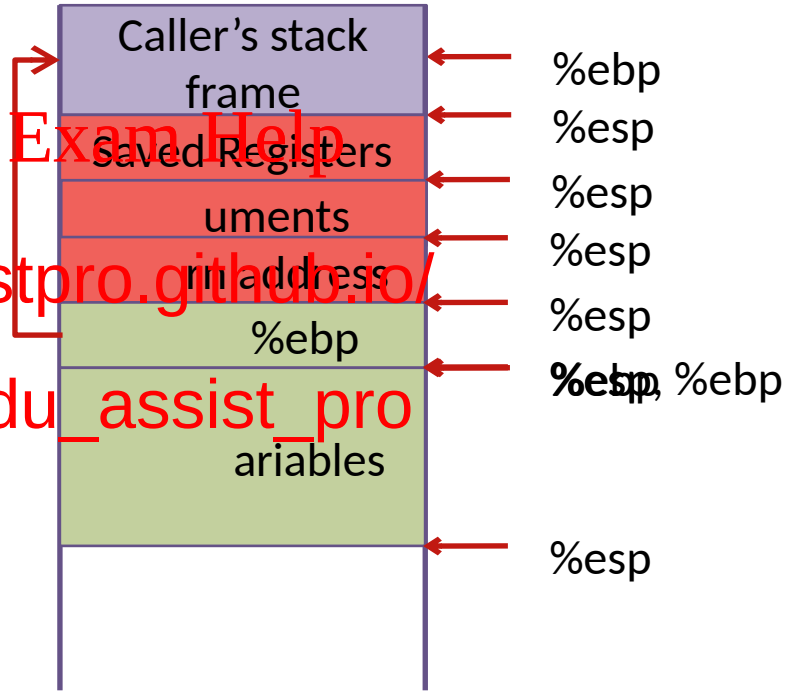
Calling a function

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



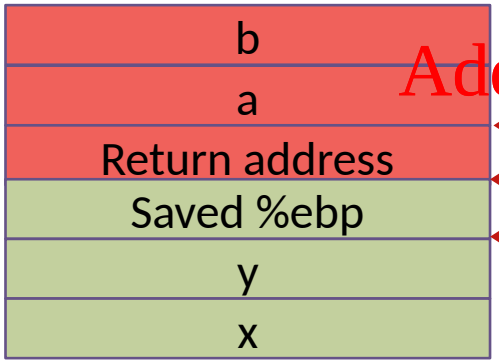
Calling conventions

- Caller does:
  - Save registers
  - Push argument
  - 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
```

%esp

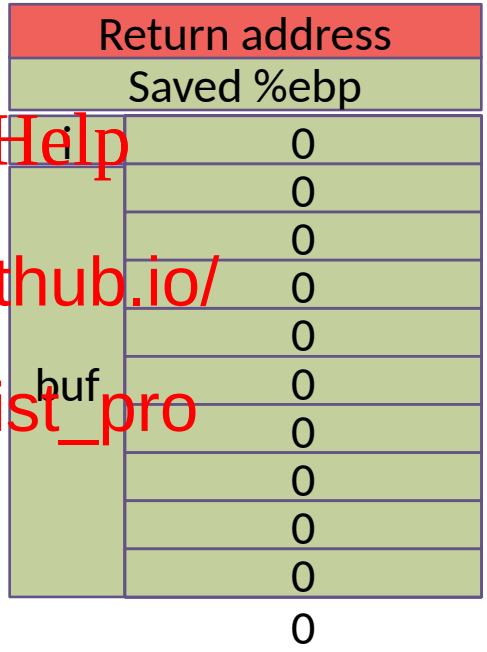
%esp

%esp, %ebp

%esp

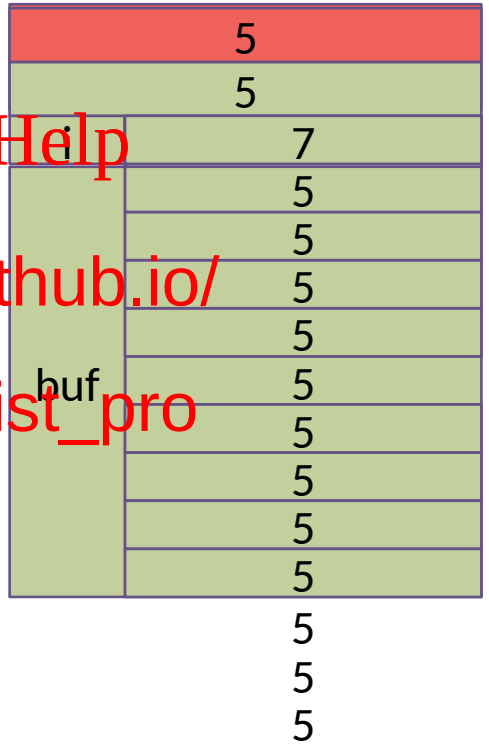
Back to a simple function

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

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

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
}
```


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

NOP Sled

- A sequence of NOP instructions l to the attack

```
NOP
NOP
NOP
```


```
Code
```

Problem patterns

- Any use of `gets`
- `strcpy`, `sprintf`, `strcat`, etc.

```
sprintf(buf, "https://%s/index.html", argv[1])
buf=new char[
strcpy(buf, a
wchar_t buf[MAXLEN];
swprintf(buf, sizeof(buf)          gv[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:

```
ar *src) {char *str
cpy\n");     fprintf
             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 a
- Use static code analysis tool
- Switch to Java, C#, etc.

Non-executable stacks

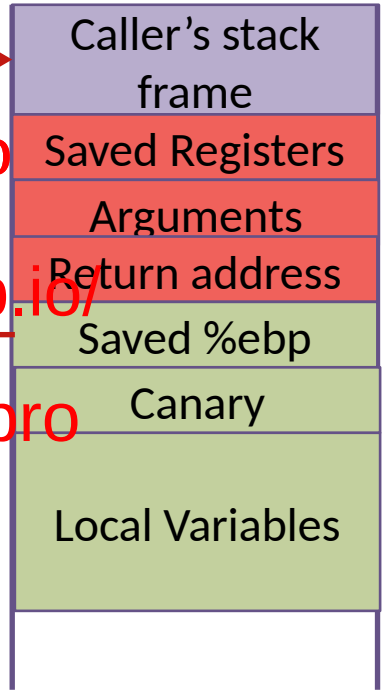
- The stack is only used for data. There's no need to run code from the stack
- The memory management unit can prevent code exe address
- Only protects against smashing the stack
- Does not prevent:
  - Heap overflow
  - Return Oriented Programming

ROP Illustrated

StackGuard

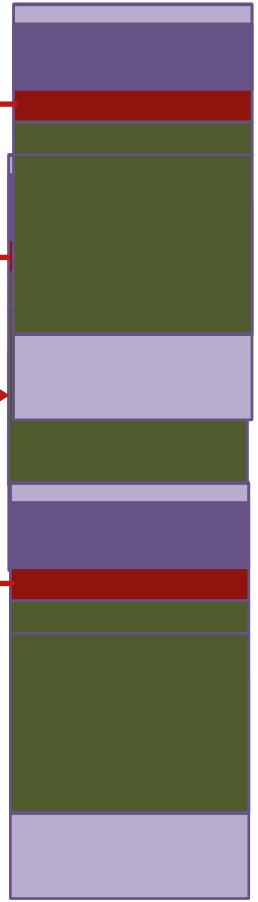
- On function entry, callee
  - Saves `%ebp`
  - Sets new `%ebp`
  - Pushes the *canary*
  - Creates space for local
- Verify the canary on

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



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 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 som echanisms
- **There's no silv**