CS 426 Lab1

Tyler Wykoff

Task

- Understand buffer overflow

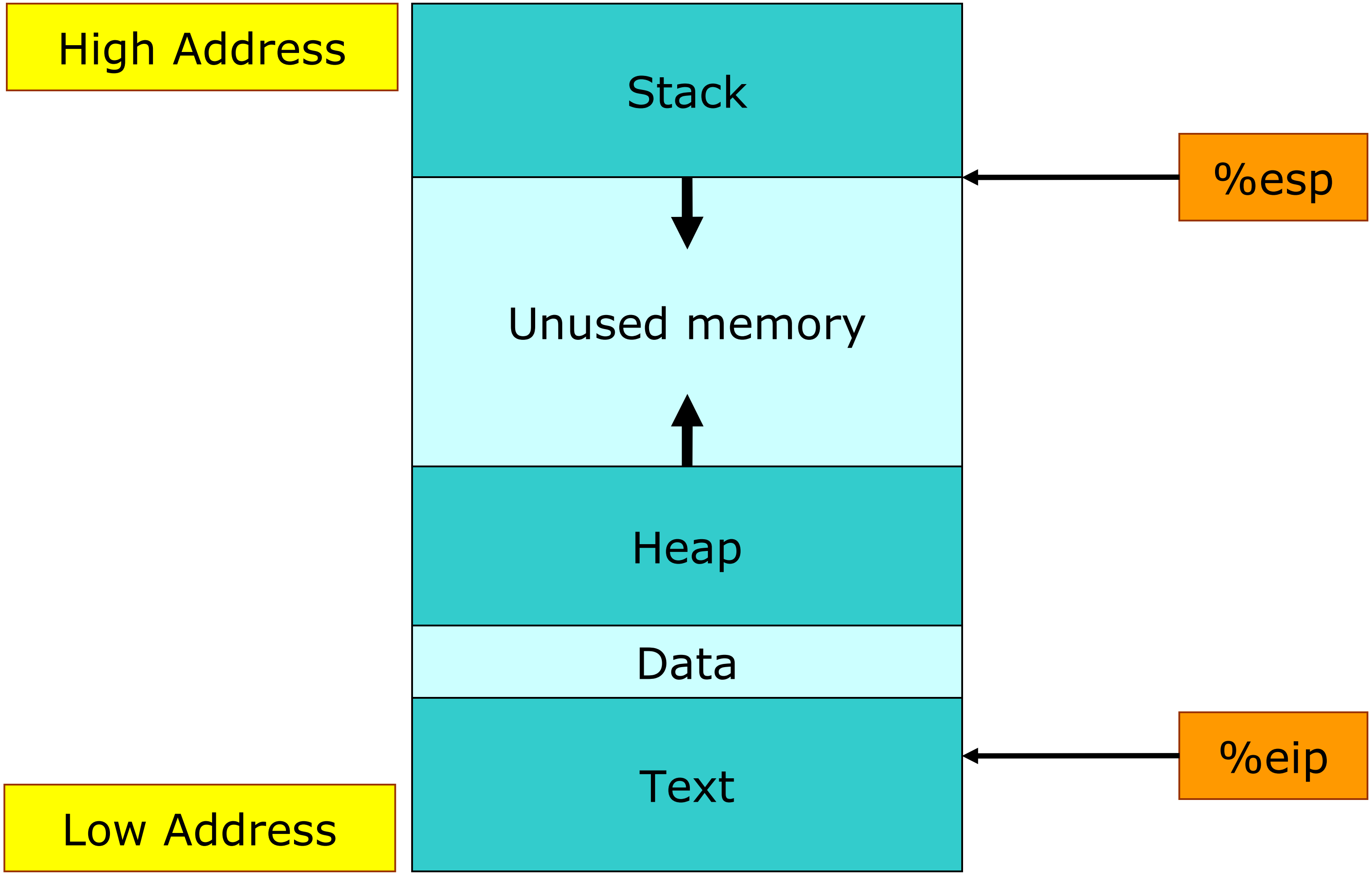
- Exploit some bugs

- Environment
  - Linux
  - Targets: C language
  - Exploits: in C or script

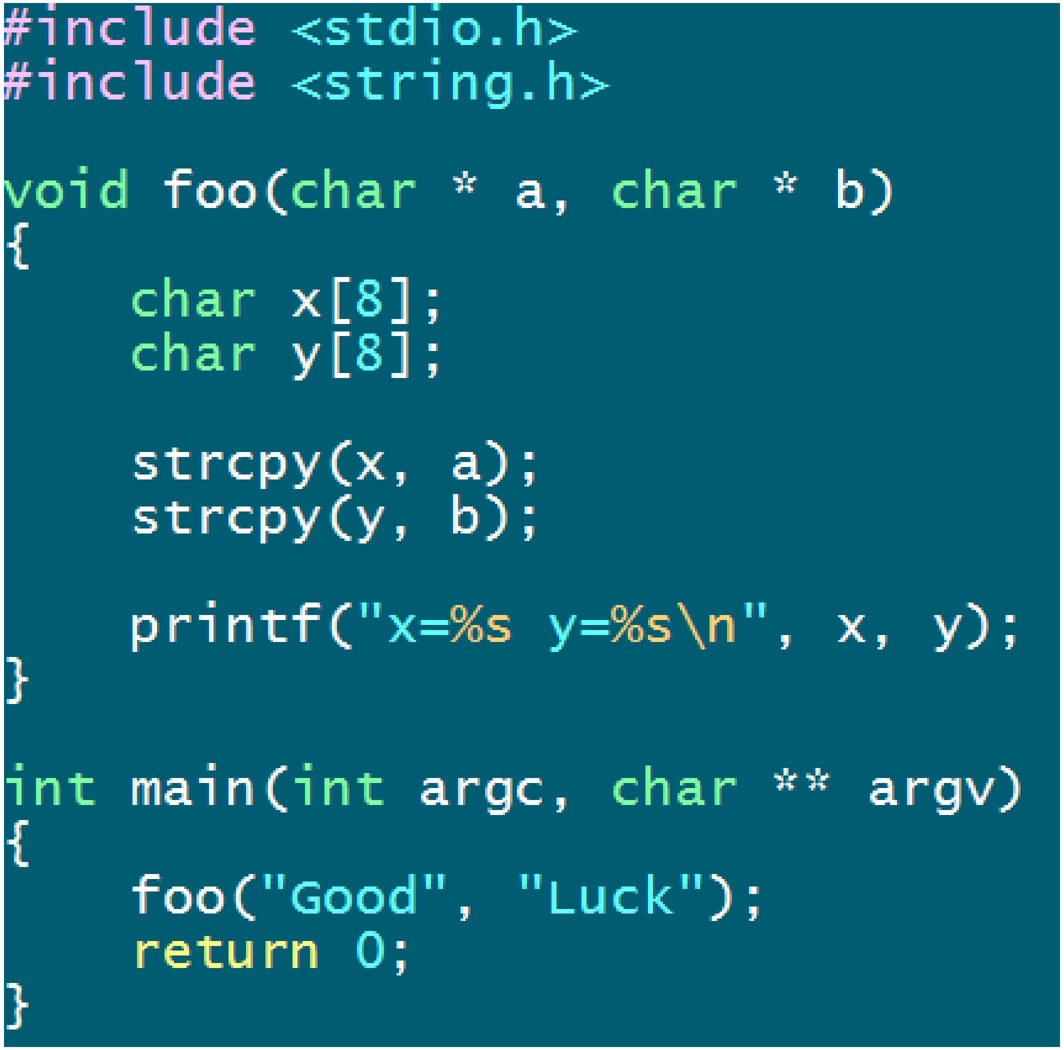
Outline

- Function call
- Examples
- Targets
- Useful tools
- Environment setup

Memory Layout Overview

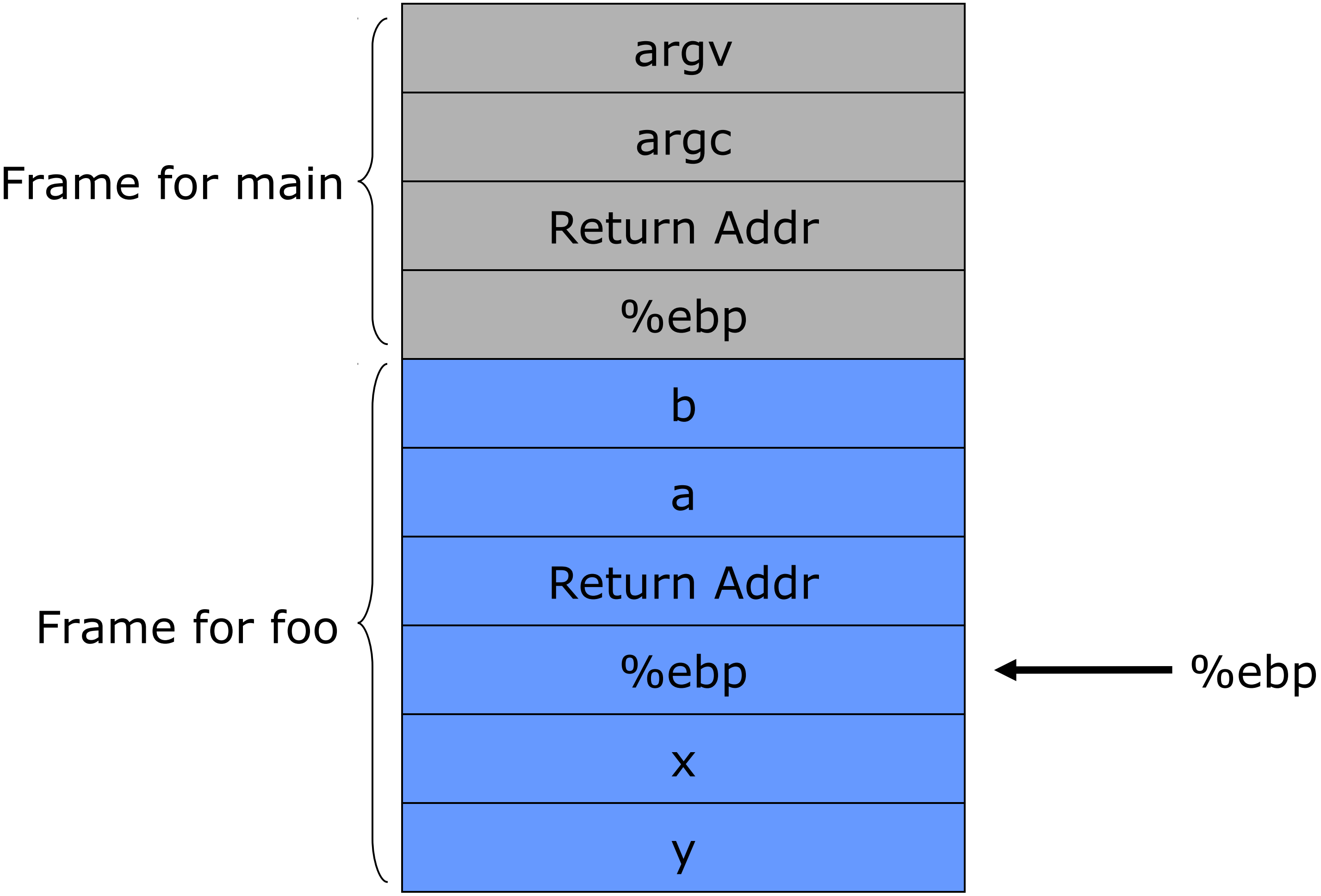


Function Call(1) example1.c



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

void foo(char * a, char * b)
{
    char x[8];
    char y[8];

    strcpy(x, a);
    strcpy(y, b);

    printf("x=%s y=%s\n", x, y);
}

int main(int argc, char ** argv)
{
    foo("Good", "Luck");
    return 0;
}
```

Function Call(2) function frame



Frame for main

argv

argc

Return Addr

%ebp

Frame for foo

b

a

Return Addr

%ebp

x

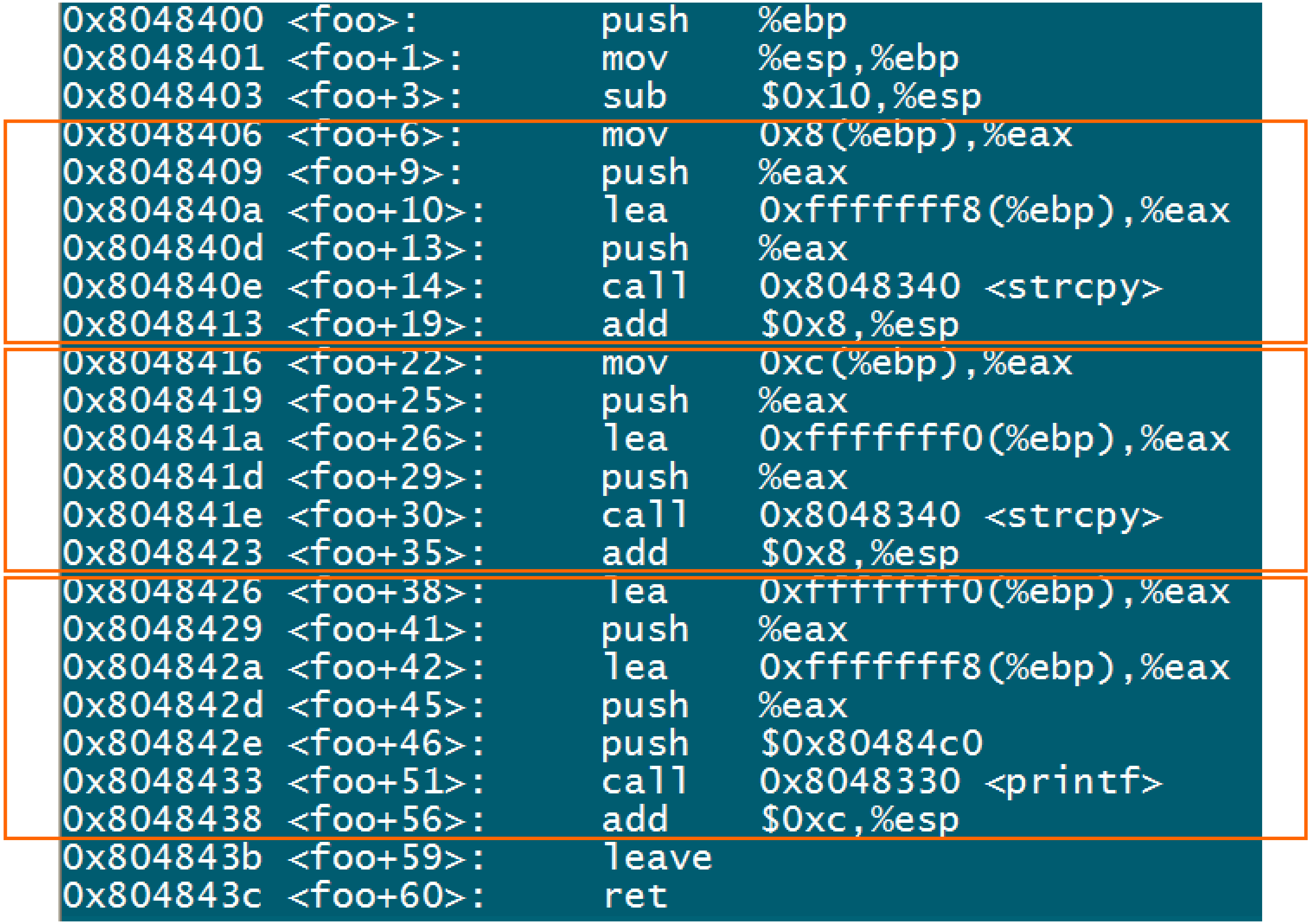
y

%ebp

Function Call(3)   what happens?

- Caller
  - Push parameter(s) on stack
  - Push return addr
  - Jump to start addr of calee
- Callee
  - Push %ebp, %ebp ← %esp
  - Allocate space for local variables
  - …
  - %esp ← %ebp, Pop %ebp
- Return
  - Pop return addr, jump to the addr
  - Restore %esp

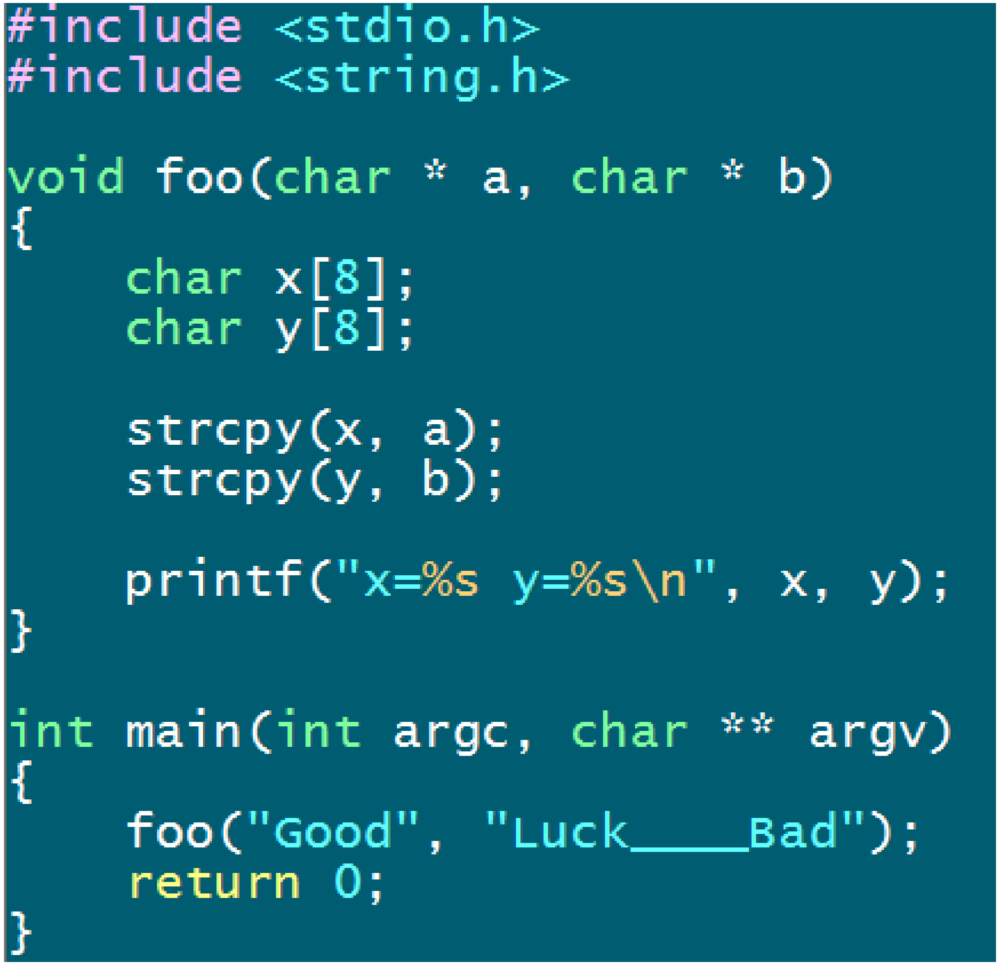
Function Call(4) assembly



```
0x8048400 <foo>:        push   %ebp
0x8048401 <foo+1>:      mov    %esp,%ebp
0x8048403 <foo+3>:      sub    $0x10,%esp
0x8048406 <foo+6>:      mov    0x8(%ebp),%eax
0x8048409 <foo+9>:      push   %eax
0x804840a <foo+10>:     lea    0xfffffff8(%ebp),%eax
0x804840d <foo+13>:     push   %eax
0x804840e <foo+14>:     call   0x8048340 <strcpy>
0x8048413 <foo+19>:     add    $0x8,%esp
0x8048416 <foo+22>:     mov    0xc(%ebp),%eax
0x8048419 <foo+25>:     push   %eax
0x804841a <foo+26>:     lea    0xfffffff0(%ebp),%eax
0x804841d <foo+29>:     push   %eax
0x804841e <foo+30>:     call   0x8048340 <strcpy>
0x8048423 <foo+35>:     add    $0x8,%esp
0x8048426 <foo+38>:     lea    0xfffffff0(%ebp),%eax
0x8048429 <foo+41>:     push   %eax
0x804842a <foo+42>:     lea    0xfffffff8(%ebp),%eax
0x804842d <foo+45>:     push   %eax
0x804842e <foo+46>:     push   $0x80484c0
0x8048433 <foo+51>:     call   0x8048330 <printf>
0x8048438 <foo+56>:     add    $0xc,%esp
0x804843b <foo+59>:     leave
0x804843c <foo+60>:     ret
```

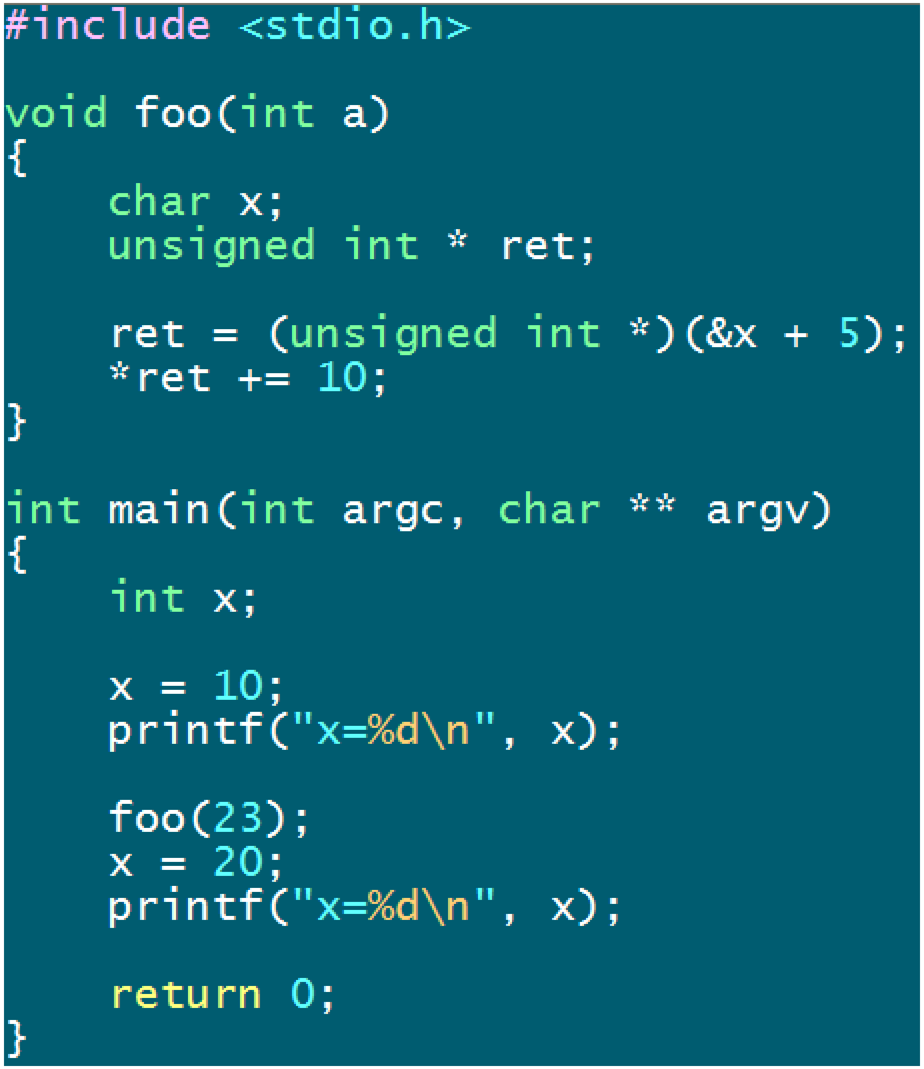
Buffer Overflow example1b.c

- C doesn’t check boundaries!



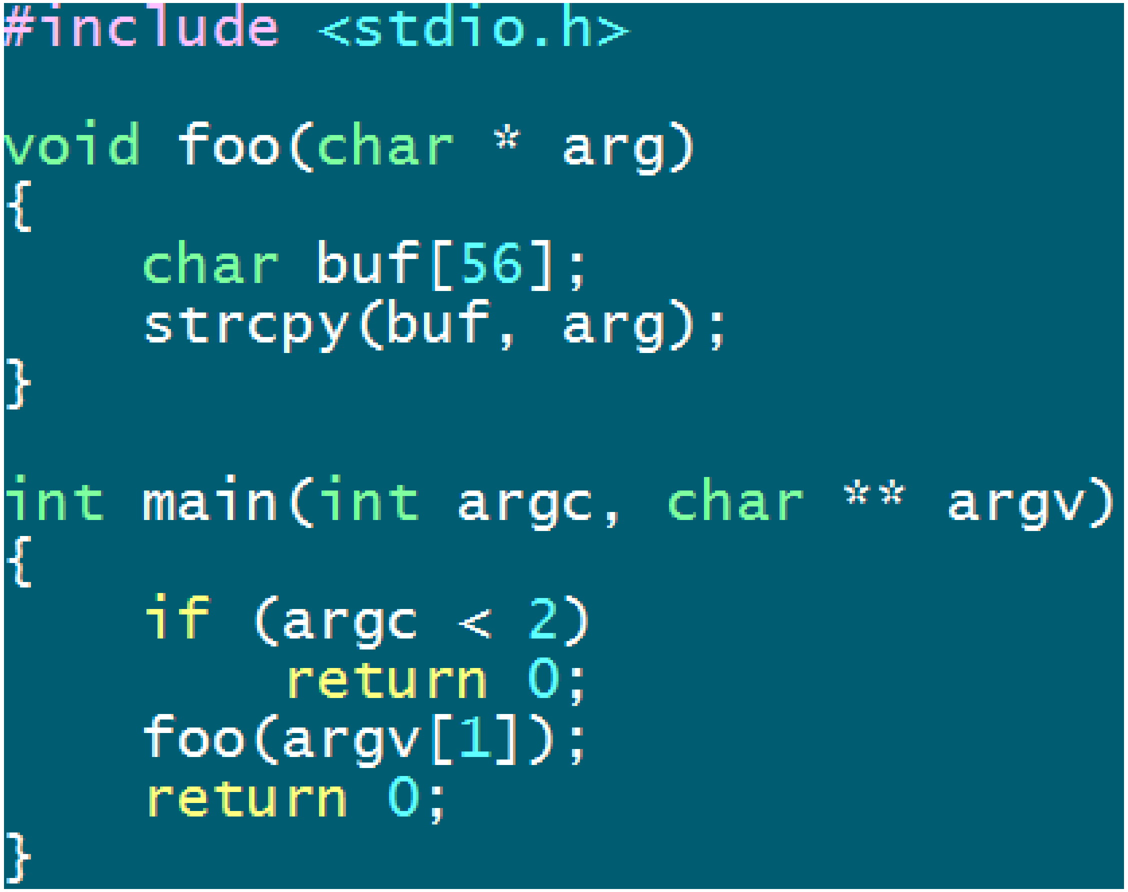
```c
#include <stdio.h>
#include <string.h>

void foo(char * a, char * b)
{
    char x[8];
    char y[8];

    strcpy(x, a);
    strcpy(y, b);

    printf("x=%s y=%s\n", x, y);
}

int main(int argc, char ** argv)
{
    foo("Good", "Luck____Bad");
    return 0;
}
```

Example 2 example2.c



```c
#include <stdio.h>

void foo(int a)
{
    char x;
    unsigned int * ret;

    ret = (unsigned int *)(&x + 5);
    *ret += 10;
}

int main(int argc, char ** argv)
{
    int x;

    x = 10;
    printf("x=%d\n", x);

    foo(23);
    x = 20;
    printf("x=%d\n", x);

    return 0;
}
```

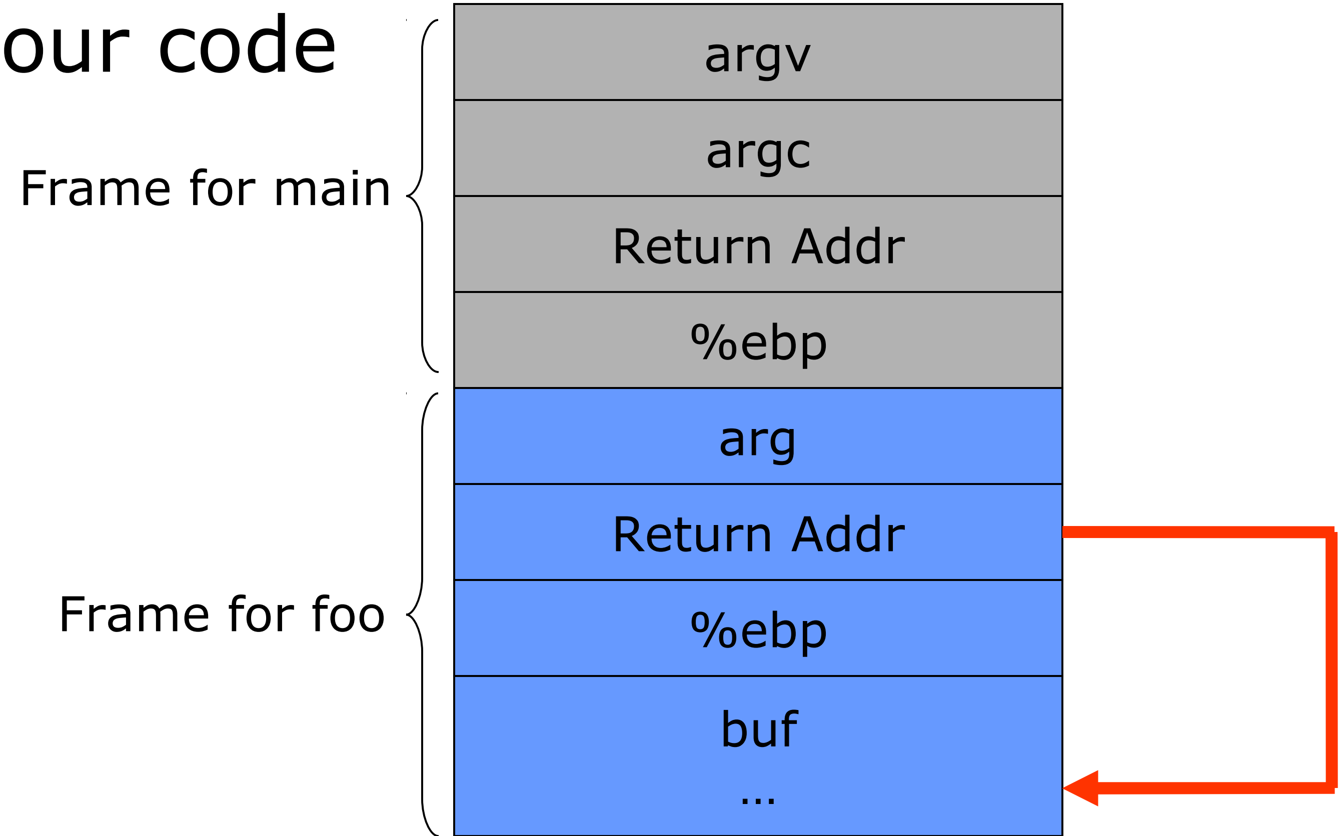
Example 3 example3.c



```c
#include <stdio.h>

void foo(char * arg)
{
    char buf[56];
    strcpy(buf, arg);
}

int main(int argc, char ** argv)
{
    if (argc < 2)
        return 0;
    foo(argv[1]);
    return 0;
}
```

Example 3 goal

- Load some code to the buffer
- Modify the return addr to execute our code



Example3 preparation

- Need to know
  - Address of the buf
  - Address of the return addr
  - Distance between buf and return addr
  - Length of the buffer
- Insert code in the buffer
  - The code to launch a shell
  - Reading: Smashing The Stack For Fun And Profit by Aleph One
  - Provided in exploits/shellcode.h

Example3 exploitation

- Insert shellcode at the beginning of the buffer
- Put the addr of buf somewhere in the buffer
- Excecute the target program

Return to lib-c attack

- Defense against buffer overflow
  - Stack data are not executable
  - Attack cannot provide code in the stack

- Attacker can still modify the return address
  - Return to some system library
  - For example, system(const char * string)

Target1

- A program to check the correctness of the password
- Goal: Make the program accept your ‘password’
- Exploit1.sh: a shell script
- Credit: 20%

Target2

- A program to print a coupon
- Goal: to print a lot coupons!
- Exploit2.c: c program
- Credit: 10% will be given if you can print two coupons(only launching the target program once) 20% will be given if you can print more than twenty coupons

Target3

- A program to check if a password is strong or weak
- Goal: to start a shell, by using a buffer overflow and shellcode
- Exploit3.c: c program
- Credit: 30%

Target4

- A program to check if a password is strong or weak
- Goal: to start a shell, using a return-to-libc attack
- Exploit4.c: c program
- Credit: 30%

Useful tools

- GDB
  - Start: gdb ./example1
  - Source: list linenum
  - Assembly: disassemble func
  - Step: step/stepi
  - Memory: x addr
  - Variables/registers: print var/reg
- Will give GDB tutorial in PSO this week!

Warming up

- Understand what is going on
  - The assembly code
  - The memory(stack)
  - The registers
  - The variables
  - What does LEAVE/RET do
  - …

Environment Setup

- The OS is running in a virtual machine
- Login
  - Connect to the VM
    - ssh cs426vm1.cs.purdue.edu
- Tools available
  - gcc, make, gdb, vim, emacs

Submission

- Deadline is 11:59pm Oct 8th (two weeks from Friday)
- Just leave your solution files (including answers to questions) in ./exploits of your home directory
- .c files should be compiled and ready to run without any arguments

Team Details

- Email me (twykoff@purdue.edu):
  - Who you are working with (both of your names)
  - What your requested login name is
- If you don't yet have a partner
  - Email me and I'll pair you up

Other stuff (1)

- Exploits codes are short
- Several ways to exploit
- Start early
- Codes from others may *not* work
- Backup files often (outside the virtual machine)
- Make your exploits stable

Other stuff (2)

- Don’t use the machines for other purposes
- Updates may be available through mailing list
- Have fun☺

Questions?