CSE 127: Introduction to Security

Stack buffer overflows

Nadia Heninger

UCSD

Winter 2021 Lecture 3

Some slides from Kirill Levchenko, Stefan Savage, and Deian Stefan

Today: Stack buffer overflows

Lecture objectives:

- Understand how buffer overflow vulns can be exploited
- Identify buffer overflows and assess their impact
- Avoid introducing buffer oveflow vulnerabilities
- Correctly fix buffer overflow activities

Buffer overflows

- 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

Why are they interesting?

- Core concept $\rightarrow$ 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

How are they introduced?

How are they introduced?

- No automatic bounds checking in C/C++

How are they introduced?

- No automatic bounds checking in C/C++
- 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

How are they introduced?

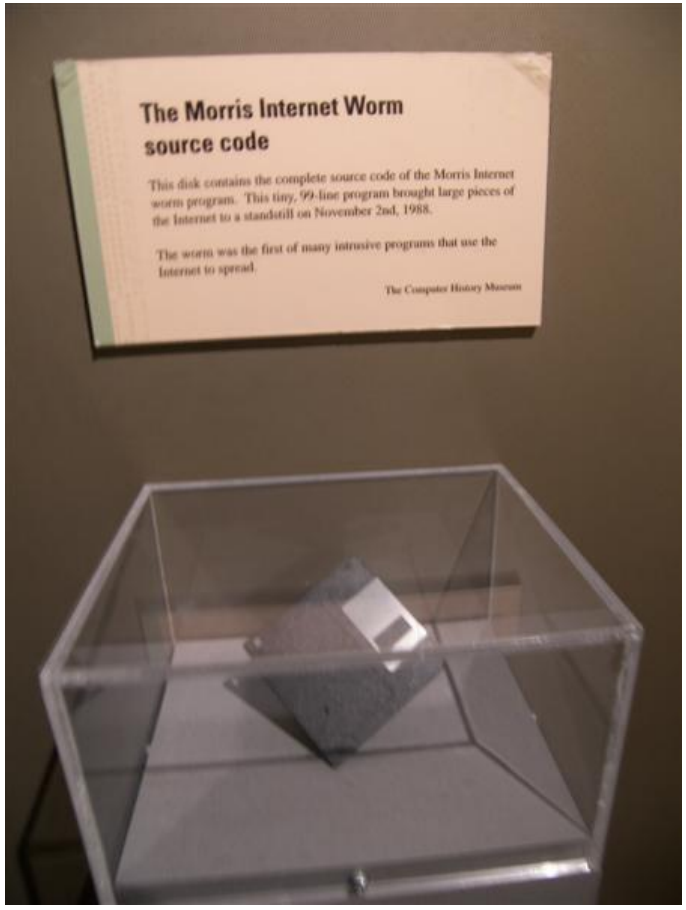
- No automatic bounds checking in C/C++
- The problem is made more acute by the fact that many C stdlib functions make it easy to go past bounds
- String manipulation functions like `gets()`, `strcpy()`, and `strcat()` all write to the destination buffer until they encounter a terminating `'\0'` byte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

Let's look at the finger daemon in BSD 4.3

```
/*
 * 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';
                }
        }
```

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



That was over 30 years ago!

Surely buffer overflows are no longer a problem...

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 em 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

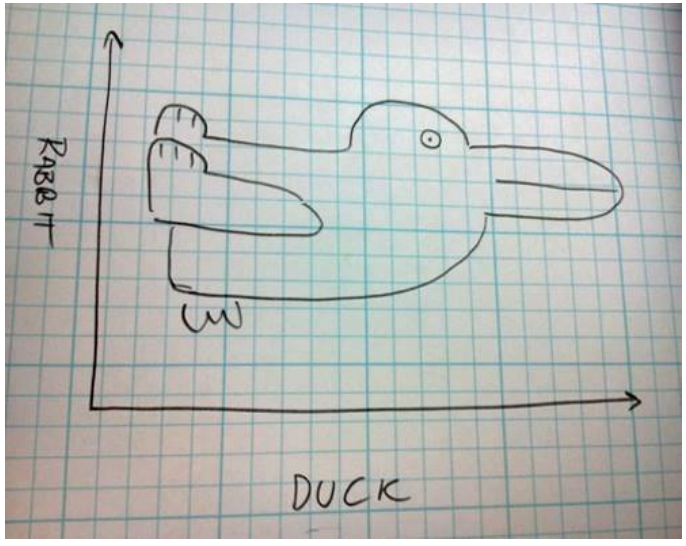
Vulnerability Discovery

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 overflow let you take over a machine?

- Your program manipulates data
- Data manipulates your program



What we need to know

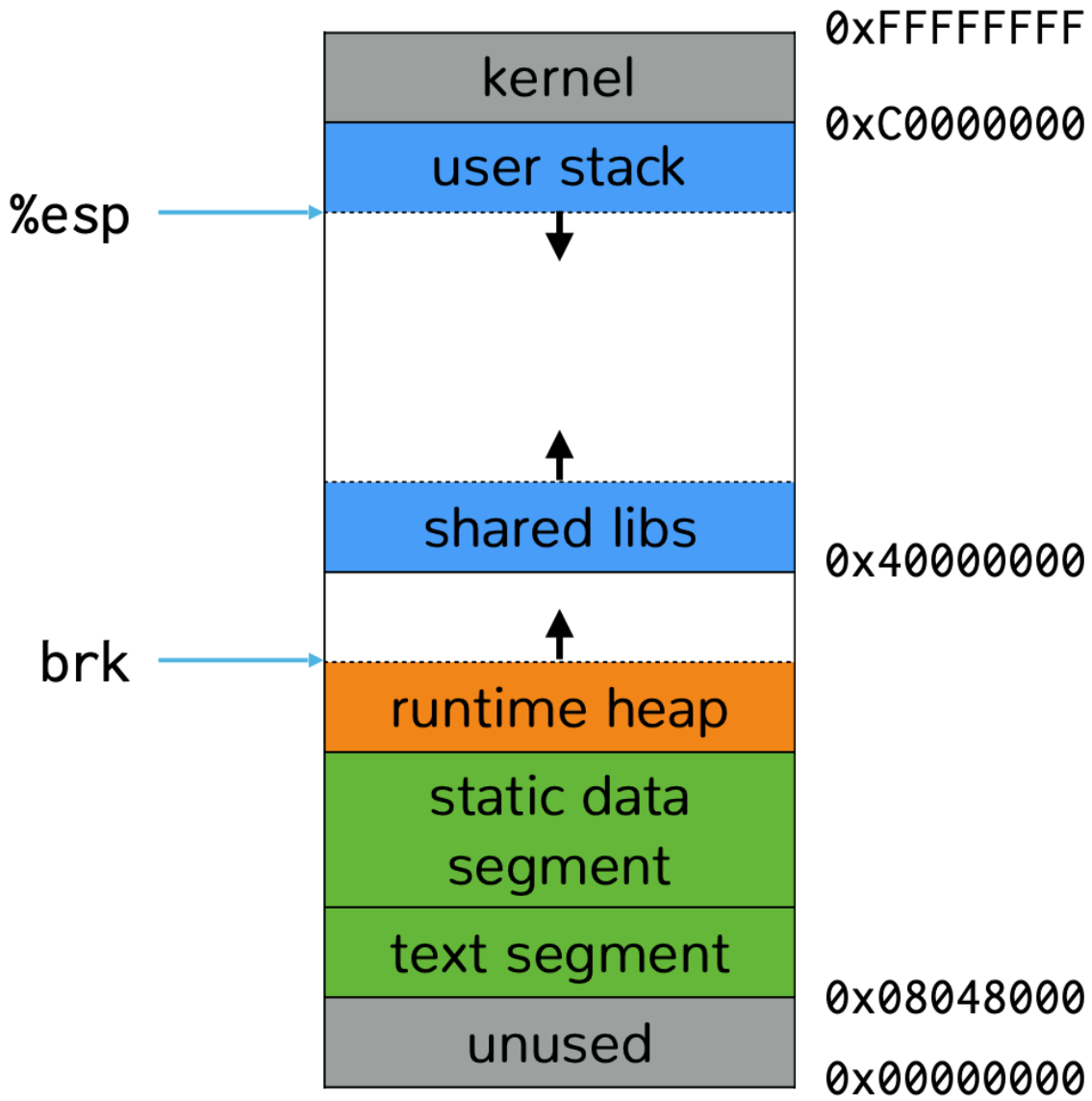
- How C arrays work
- How memory is laid out
- How the stack and function calls work
- How to turn an array overflow into an exploit

How do C arrays work?

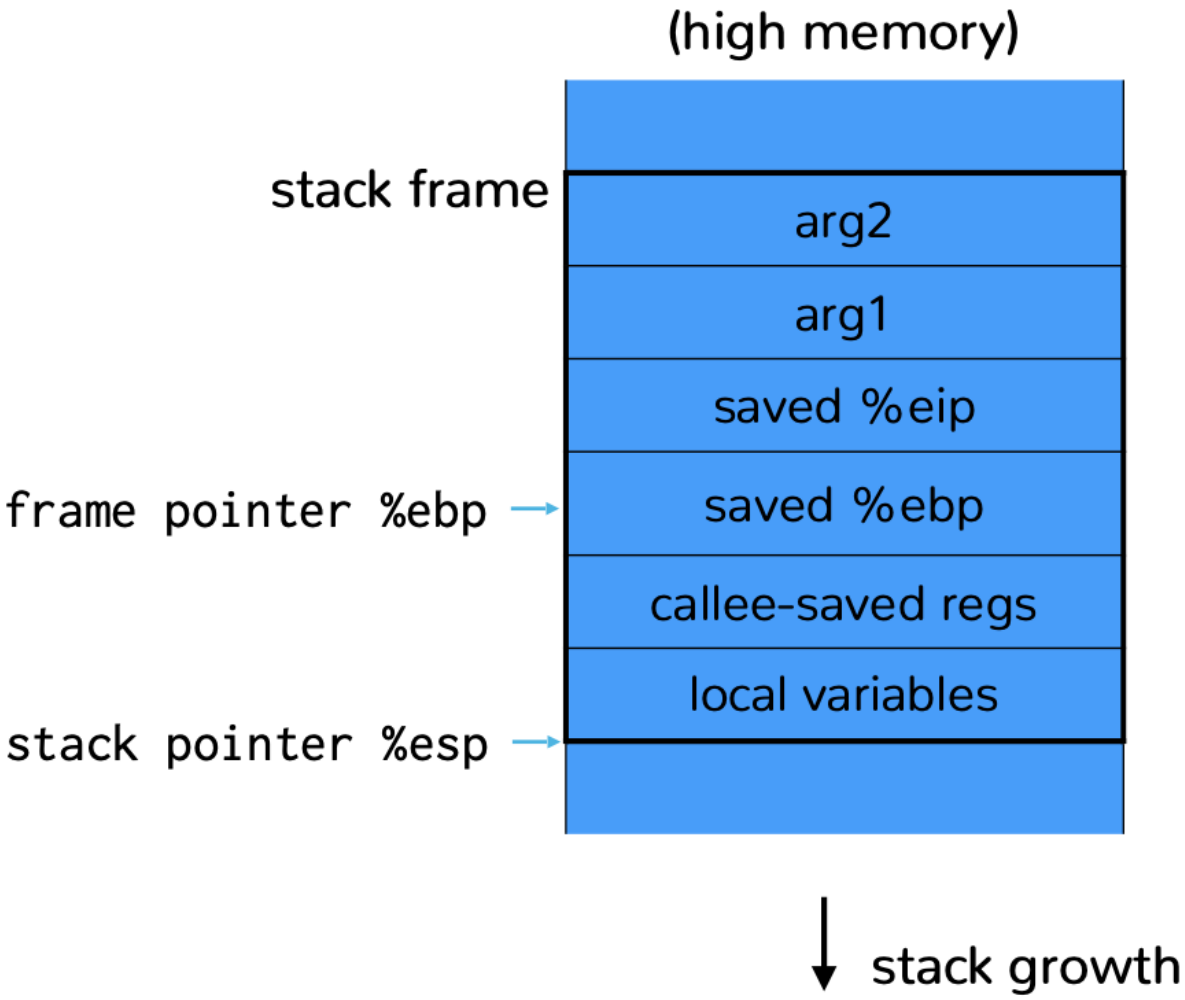
- What does `a[idx]` get compiled to?
  - *((a)+(idx))
- What does the spec say?
  - 6.5.2.1 Array subscripting in ISO/IEC 9899:2017
  - There is no concept of bounds!

Linux process memory layout

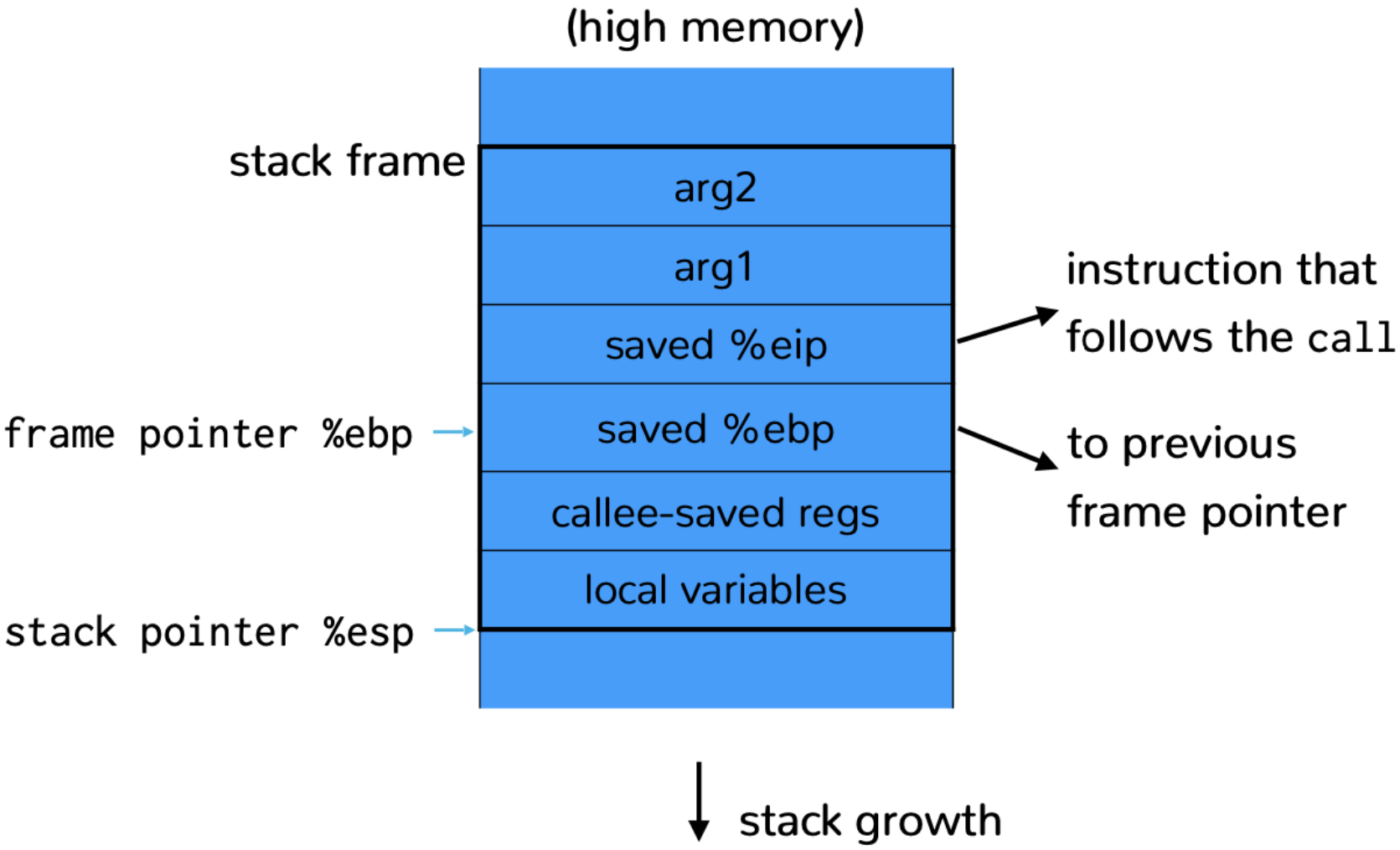
- Stack
- Heap
- Data segment
- Text segment
  - Binary instructions



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` $\rightarrow$

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` $\rightarrow$ `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` $\rightarrow$ `edx = eax`

`movl $0x123, %edx` $\rightarrow$

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` $\rightarrow$ `edx = eax`

`movl $0x123, %edx` $\rightarrow$ `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` $\rightarrow$ `edx = eax`

`movl $0x123, %edx` $\rightarrow$ `edx = 0x123`

`movl (%ebx), %edx` $\rightarrow$

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

- 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)
```

Brief review of stack instructions

```
pushl %eax      → subl $4, %esp
                  movl %eax, (%esp)

popl %eax       → movl (%esp), %eax
                  addl $4, %esp
```

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
```

Brief review of stack instructions

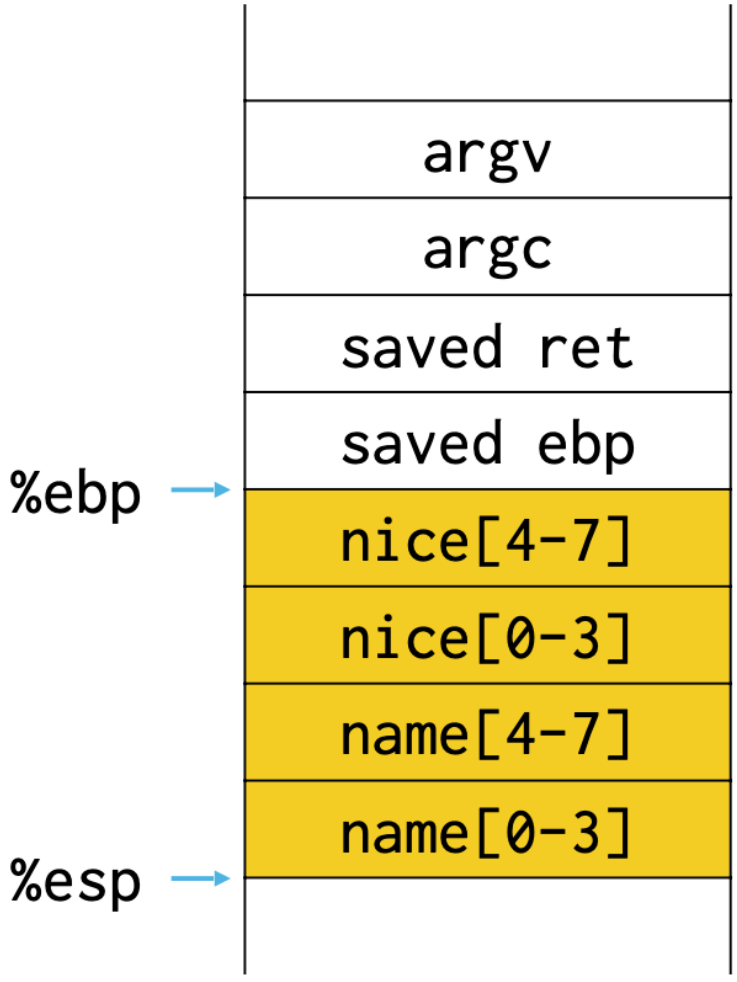
```
pushl %eax      → subl $4, %esp
                  movl %eax, (%esp)

popl %eax       → movl (%esp), %eax
                  addl $4, %esp

call $0x12345 → pushl %eip
                  movl $0x12345, %eip

ret             → popl %eip
```

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
```

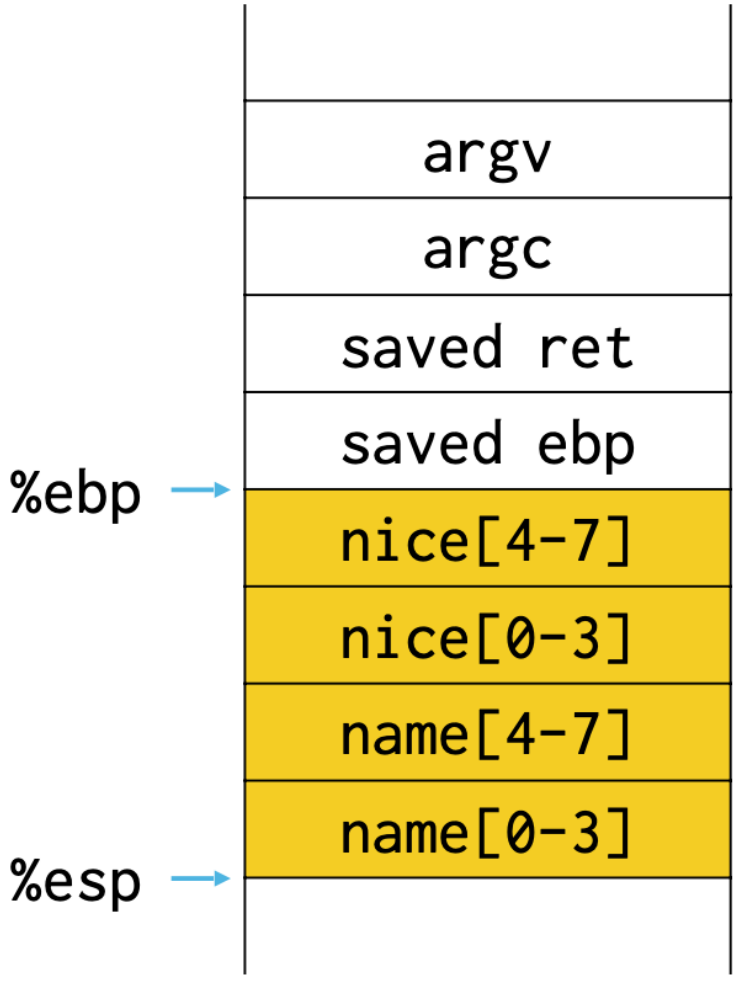
Example 1

```
#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

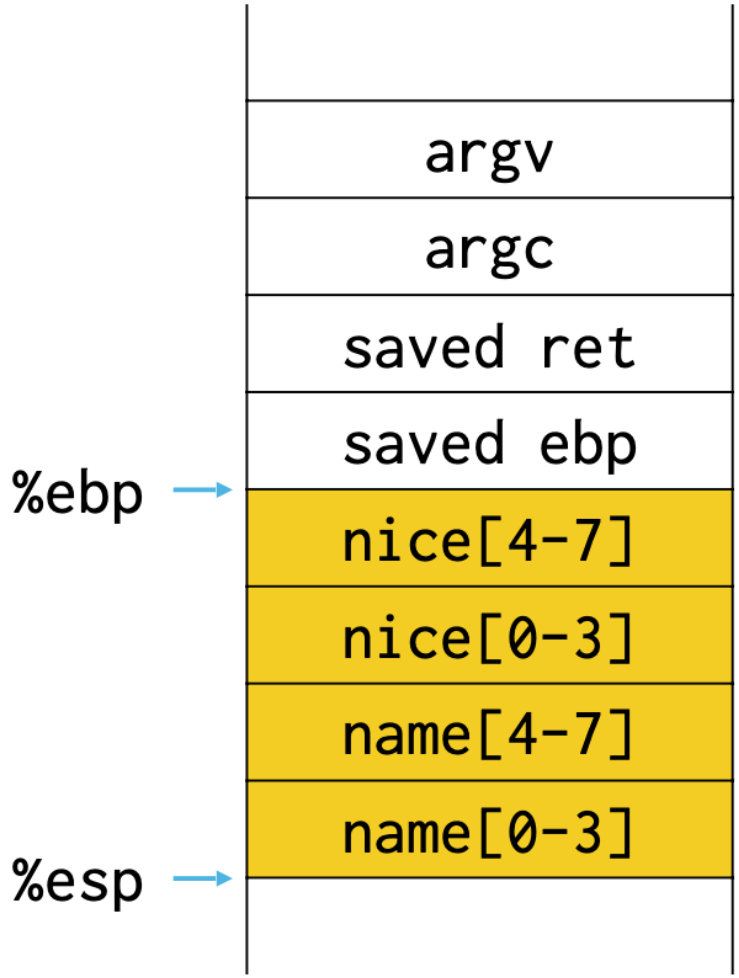
```
#include <stdio.h>
#include <string.h>

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

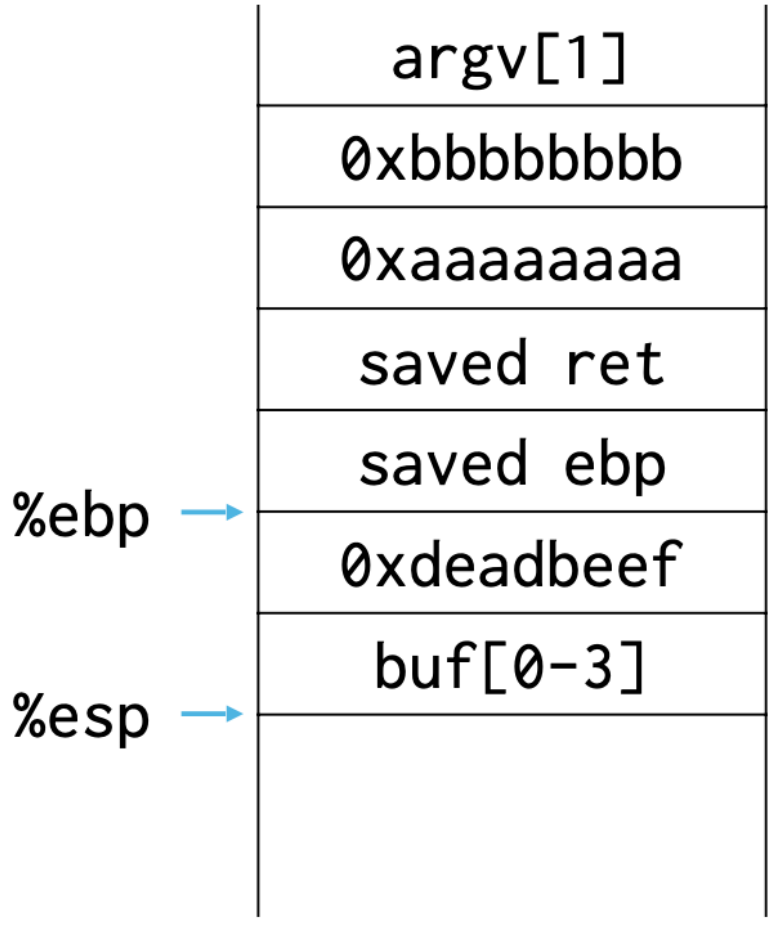
```
#include <stdio.h>
#include <string.h>

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...

If program argument is long...

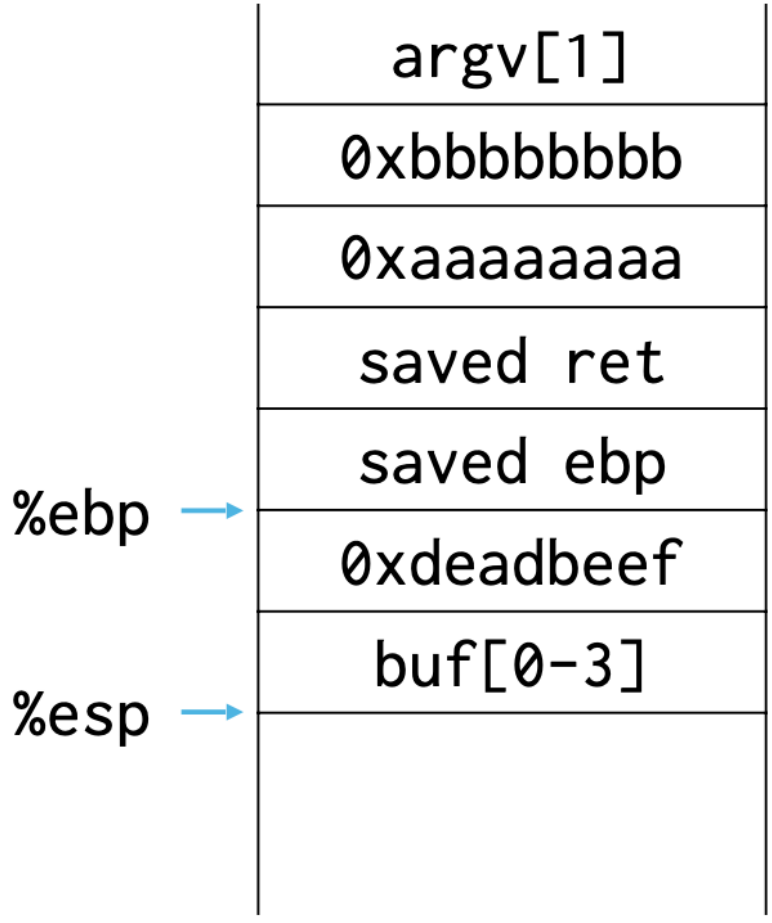
Stack buffer overflow

- 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?

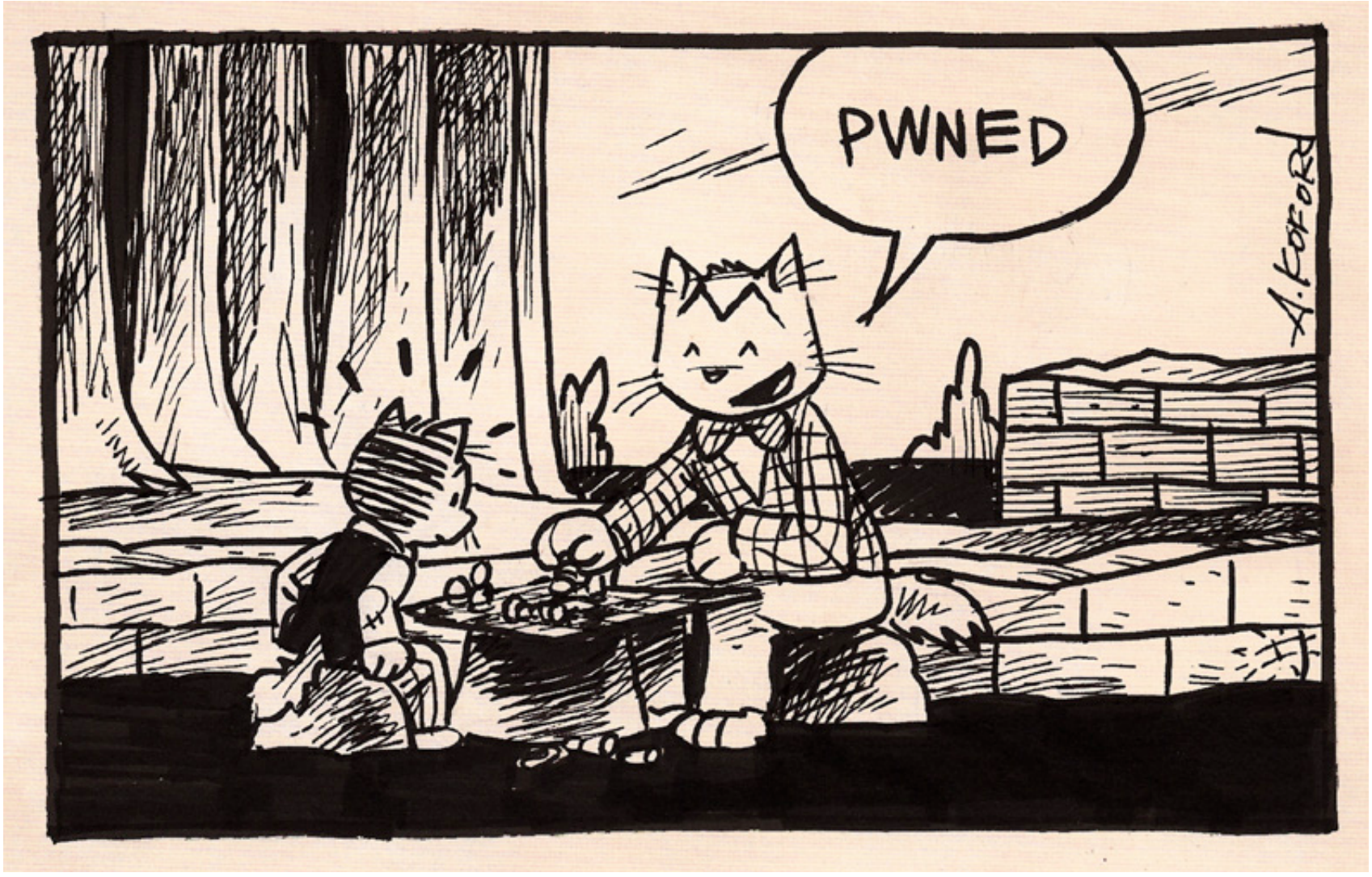
Can jump to existing functions

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;
}
```

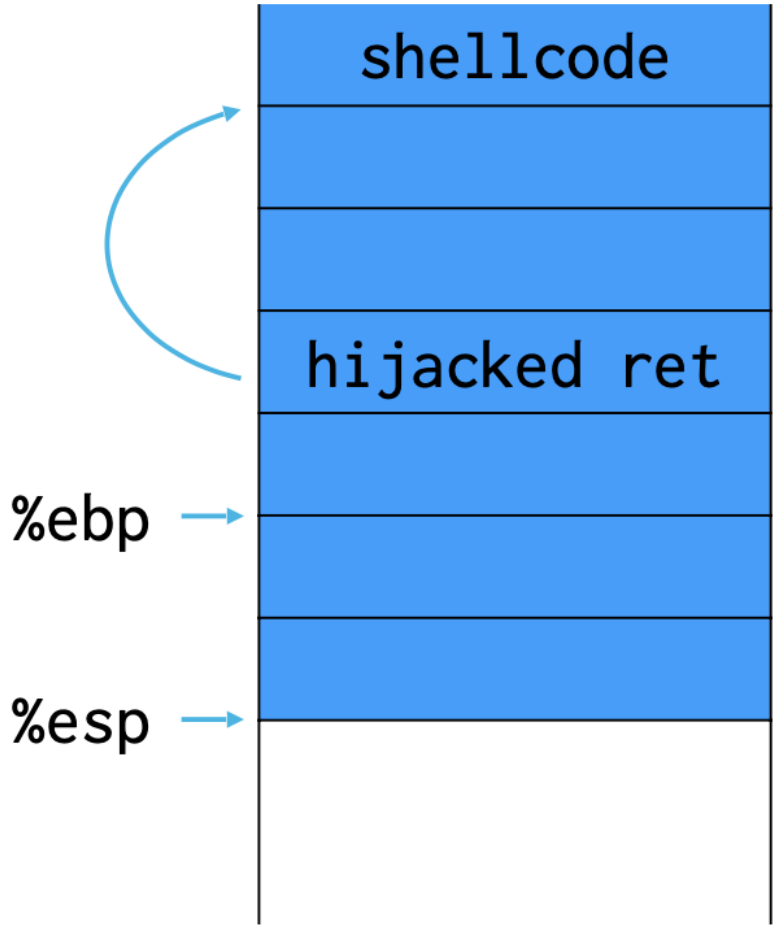


Jump to existing functions



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?
- If payload is via `gets()` must also avoid line breaks
  - Why?
- Fix: Use different instructions and NOPs.

Payload is not always robust

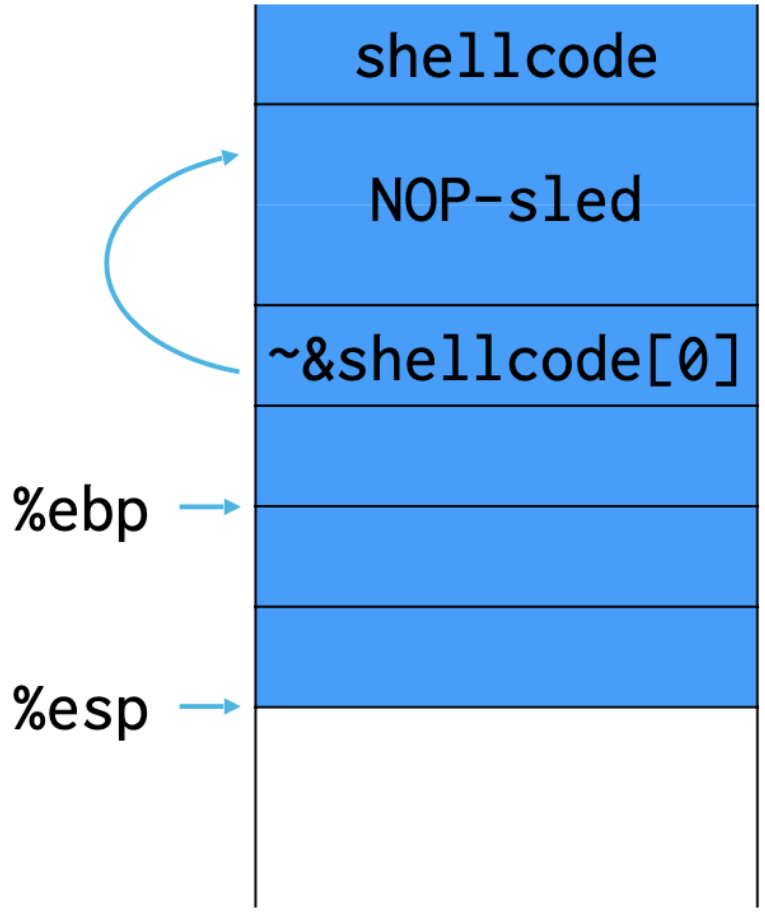
- Exact address of shellcode start not always easy to guess.

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.
  - Fix: NOP sled. Fill space with NOP instructions to allow error in stack locations.



Next up: Defenses and more advanced attacks.