# CIS 447/544: Computer and Network Security

Anys Bacha

Slides from U. Shankar, M. Hicks, K. Du, D. Boneh, N. Zeldovich, A. Rahmati

# What is a Buffer Overflow?

- The Bugs Framework (government entity that classifies bugs into distinct classes) defines a buffer overflow as

**software accesses through an array of memory that is outside the boundary of the array**

# What is a Buffer Overflow?

- Buffer overflows (BOF) stem primarily from low level bugs written in C/C++

- In most cases buffer overflows cause crashes, but if maliciously crafted can result in:

  - Private data being stolen
  - Arbitrary code being executed
  - Critical information being corrupted

# How Relevant Are BOF

- Performance is always at the top of the feature list
  - We like technology to always be fast

- Low level languages such as C/C++ are still very popular

- Systems software often written in C/C++ (operating systems, file systems, databases, compilers, network servers, command shells, etc.)

# How Relevant Are BOF

- Many big companies still rely on C++ for their software including Google and Facebook (driven by performance)

- Internet of Things (IoT) software is primarily developed in C due to the limited hardware resources

- Compromises can result in significant damage
  - Arbitrary code execution

# How Relevant Are BOF

- Low level languages has the downside of exposing memory details
  - Exposes raw pointers to memory
  - Does not explicitly perform bounds-checking on arrays
  - Hardware doesn’t check this
  - We want to be as close to the hardware as possible

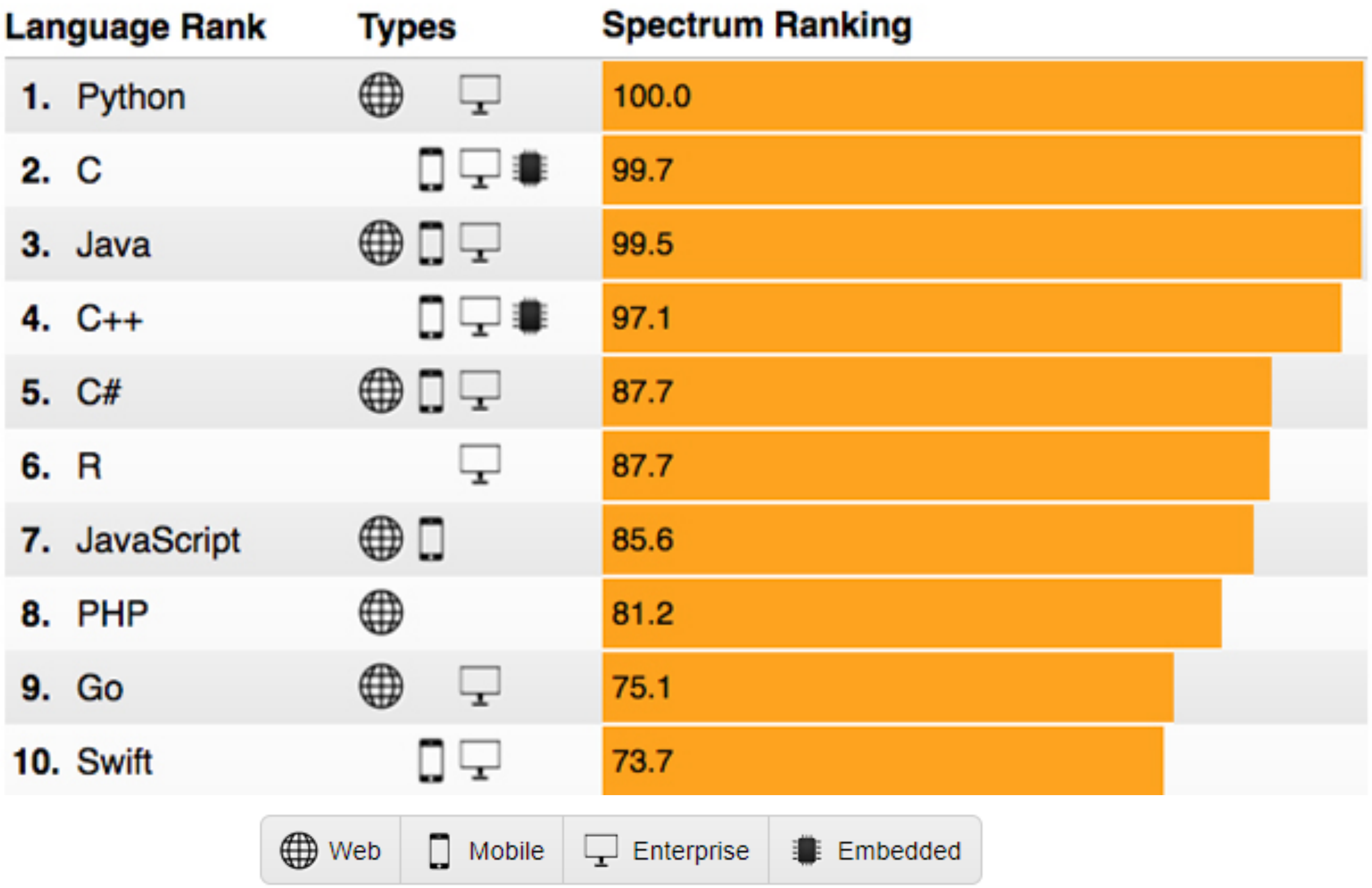
# C/C++ Still Popular

| Rank | Language | Type | Score |
|---|---|---|---|
| 1 | Python | Web, Enterprise, Embedded | 100.0 |
| 2 | Java | Web, Mobile, Enterprise | 96.3 |
| 3 | C | Mobile, Enterprise, Embedded | 94.4 |
| 4 | C++ | Mobile, Enterprise, Embedded | 87.5 |
| 5 | R | Enterprise | 81.5 |
| 6 | JavaScript | Web | 79.4 |
| 7 | C# | Web, Mobile, Enterprise, Embedded | 74.5 |
| 8 | Matlab | Enterprise | 70.6 |
| 9 | Swift | Mobile, Enterprise | 69.1 |
| 10 | Go | Web, Enterprise | 68.0 |

Web | Mobile | Enterprise | Embedded

https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019

# C/C++ Still Popular




https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

# Notable BOF Attacks

- Morris Worm (1988)
    - Worm intended to gauge the size of the ARPANET (precursor to the internet)
    - Exploited a vulnerability in fingerd
    - Sent a special string to the finger daemon that allowed it to replicate itself and execute on a new machine
    - The worm spread too aggressively (replicate itself multiple times on a given system)

# Notable BOF Attacks

- Morris Worm (1988)
  - The entire ARPANET literally came to a screeching halt
  - Over 6000  systems infected resulting in $10-100M in damages
  - Robert Morris is now a professor at MIT

# Notable BOF Attacks


- CodeRed (2001)
  - Exploited a buffer overflow in Microsoft’s IIS web server
    - Send a special request that causes and overflow and point to the worm loader
  - Worm involved different stages:
    - Days 1 – 19: Spread itself by scanning for more IIS servers on the internet
    - Days 20-27: Launch denial of service attacks on several fixed IP addresses (included White House web server)
    - Days 28-end of month: Sleep
  - Worm infected 300,000 machines in 14 hours

# Notable BOF Attacks

- CodeRed was discovered by UNIX admins seeing weird requests on their apache servers

08-02-2001, 08:53 AM

**bert**
Web Hosting Master

**"GET /default.ida?NNNNNNNNNNNNNNNNNNNN"**

This was all over one of our Apache logs today. The requests are coming from many different IPs from all over the world. We traced IPs to Italy, Brazil, Korea, USA, etc.

I was reading about it and found that this is an exploit for IIS. We run Apache so we are not too concerned, but I just wanted to know if you knew anything about this how problematic it might be.

Thanks.

# Notable BOF Attacks

- This is the payload that was used:
  - GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNN
    NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
    NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
    NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
    NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
    NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN NNNNNNNNNNNNNNNNNNN
    %u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801
    %u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3
    %u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.

- Once infected, the webserver would display:
  - HELLO! Welcome to http://www.worm.com! Hacked By Chinese!

# Notable BOF Attacks

- SQL Slammer (2003):
  - Exploited a buffer overflow in the MS-SQL server
  - Within 10 minutes infected 75,000 servers
  - Worm randomly generated IP addresses and send itself out to those addresses
  - New hosts rapidly infected over sessionless UDP protocol (fire and forget, many routers crashed as a result of high traffic)
  - The entire worm fit inside a single packet (376 Bytes)

# Notable BOF Attacks

- SQL Slammer (2003):
  - **This underscores the importance of patching**
  - **The patch was available 6 months prior to the worm's launch**

# Notable BOF Attacks

- Conficker Worm (2008/2009):
  - Exploited a buffer overflow in the Windows RPC
  - 10 million machined infected
  - Worm used Windows RPC to run shell code on the system
  - Shell code would then contact the source and download a malicious DLL
  - Other variants of the worm included dictionary attacks and removable media (autorun.inf in USB)

# Notable BOF Attacks

- Flame (2010-2012):
  - Exploited a buffer overflow in the Windows print spooler service and LNK shortcut display (similar to Stuxnet)
  - Primarily for cyber-espionage with lots of capabilities
  - Unusually large payload (20 MB)

# Notable BOF Attacks

- Flame (2010-2012):
  - Contained compression library (zlib), database (sqlite), virtual machine (for LUA)
  - Contained many encryption methods to obfuscate itself
  - Designed to steal information (record audio, take screenshots, log keystrokes, etc.)Very hard to analyze
  - Spreads itself over the network and removable media (USB autorun)

# 23-Year-Old X11 Server Security Vulnerability Discovered

213

Posted by Unknown Lamer on Wednesday January 08, 2014 @11:11AM from the stack-smashing-for-fun-and-profit

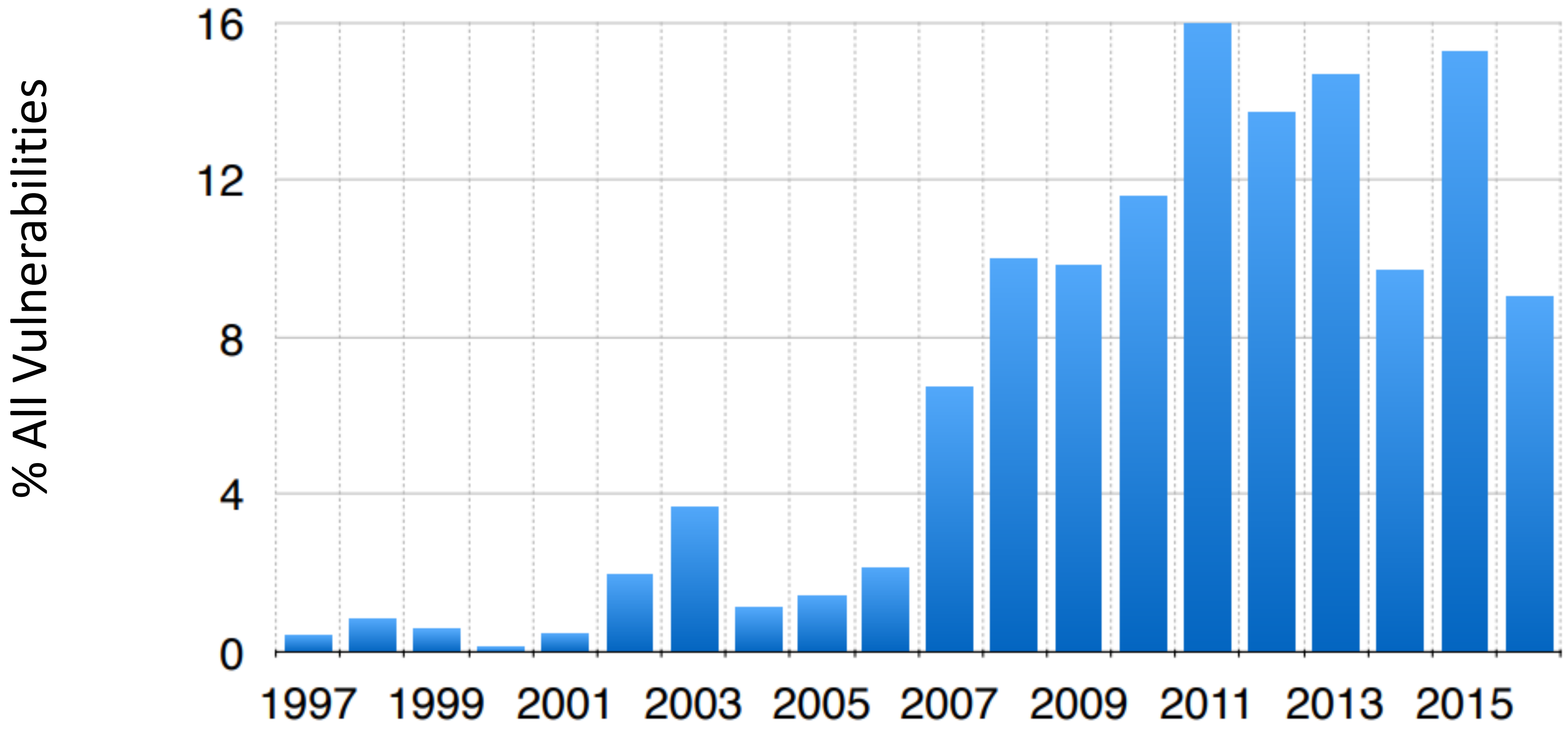
An anonymous reader writes

> "The recent report of X11/X.Org security in bad shape rings more truth today. The X.Org Foundation announced today that they've found a X11 security issue that dates back to 1991. The issue is a possible stack buffer overflow that could lead to privilege escalation to root and affects all versions of the X Server back to X11R5. After the vulnerability being in the code-base for 23 years, it was finally uncovered via the automated cppcheck static analysis utility."

There's a `scanf` used when loading BDF fonts that can overflow using a carefully crafted font. Watch out for those obsolete early-90s bitmap fonts.

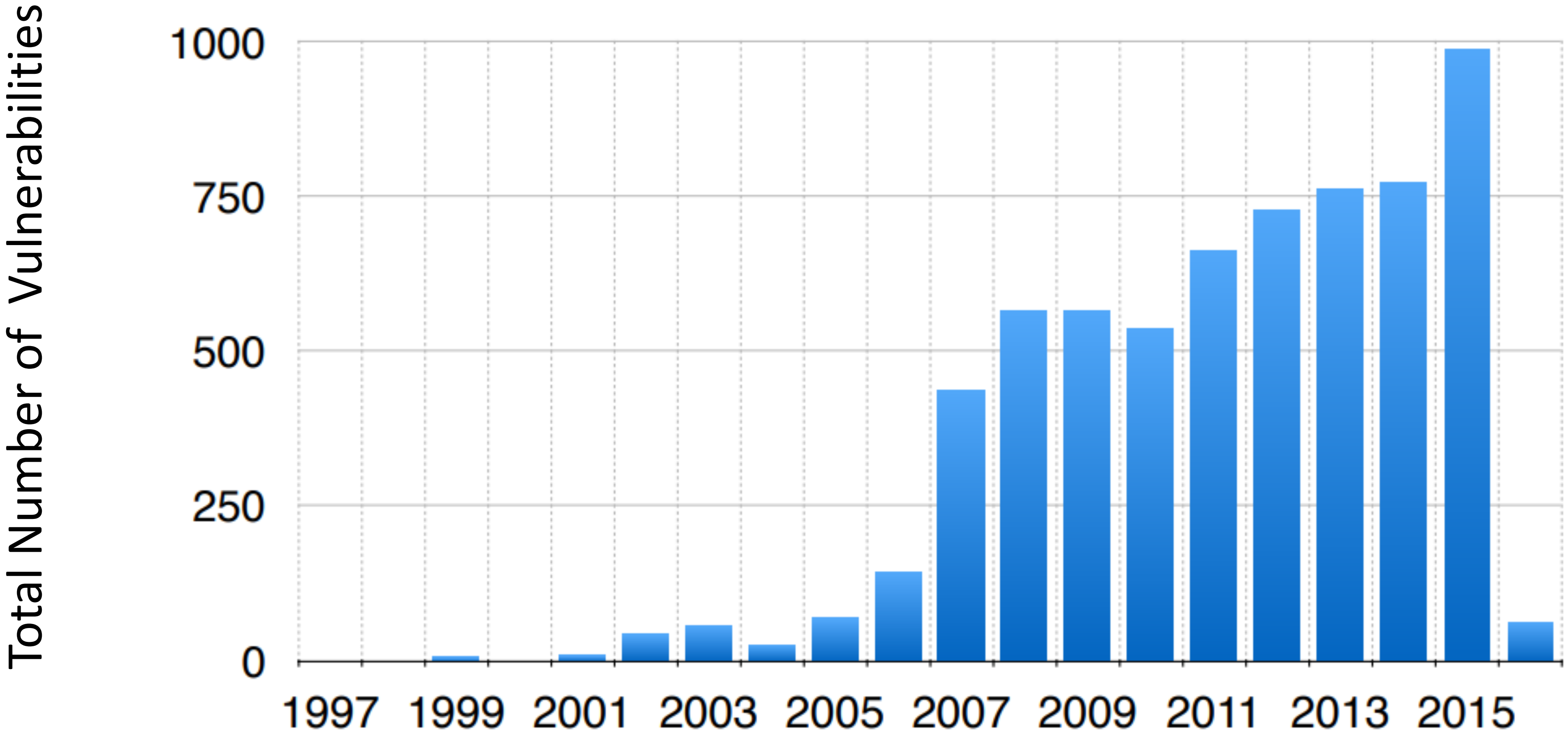
bug security xwindows

# The Prevalence of BOF




https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-119

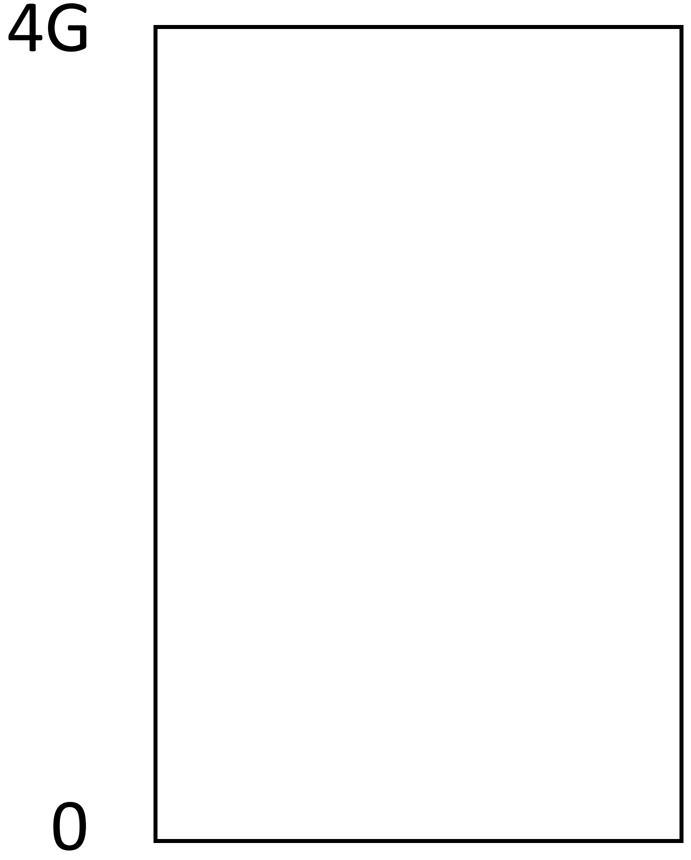
# The Prevalence of BOF




https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-119

# Memory Layout

# Program Layout in Memory



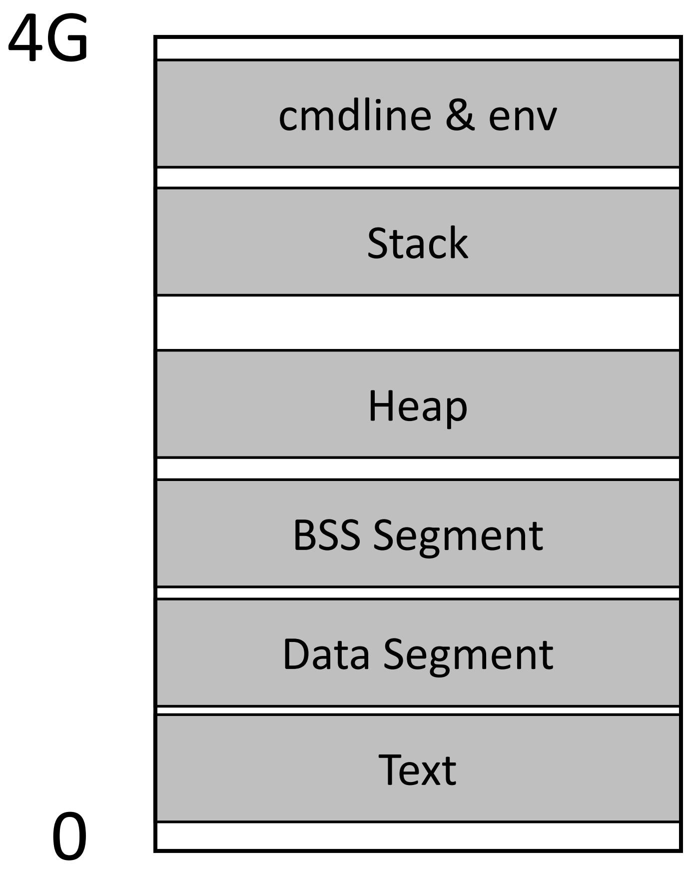

0xFFFFFFFF

Process thinks it owns the entire range

Virtual addresses that the OS maps to physical memory addresses

0x00000000

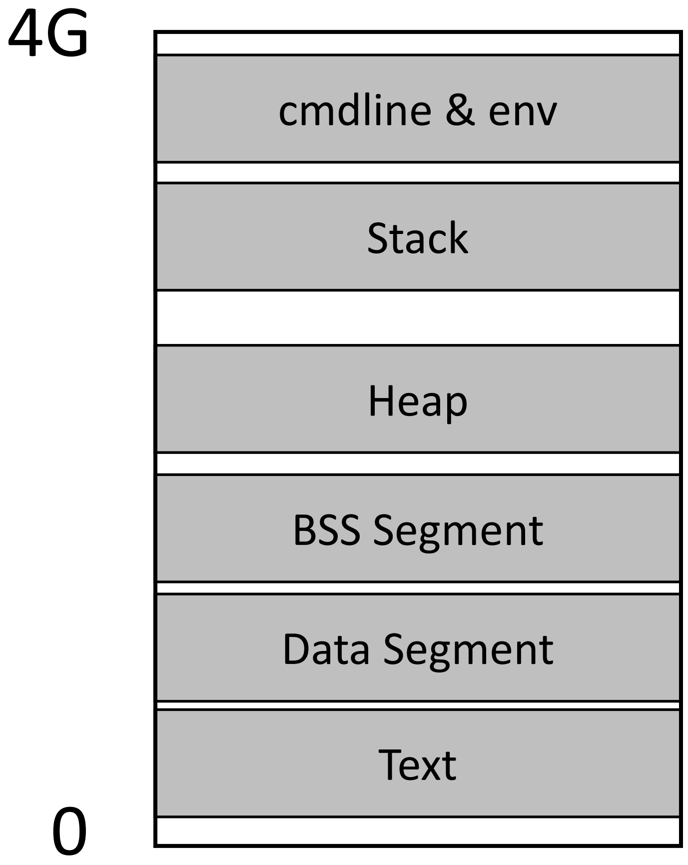
# Program Layout in Memory




```c
int x = 100;
int main()
{

  int  a=2;
  float b=2.5;
  static y;


  int *ptr = (int *) malloc(2*sizeof(int));


  ptr[1]=5;
  ptr[2]=6;


  free(ptr)

  return 1;
}
```

**Where would variables be located?**

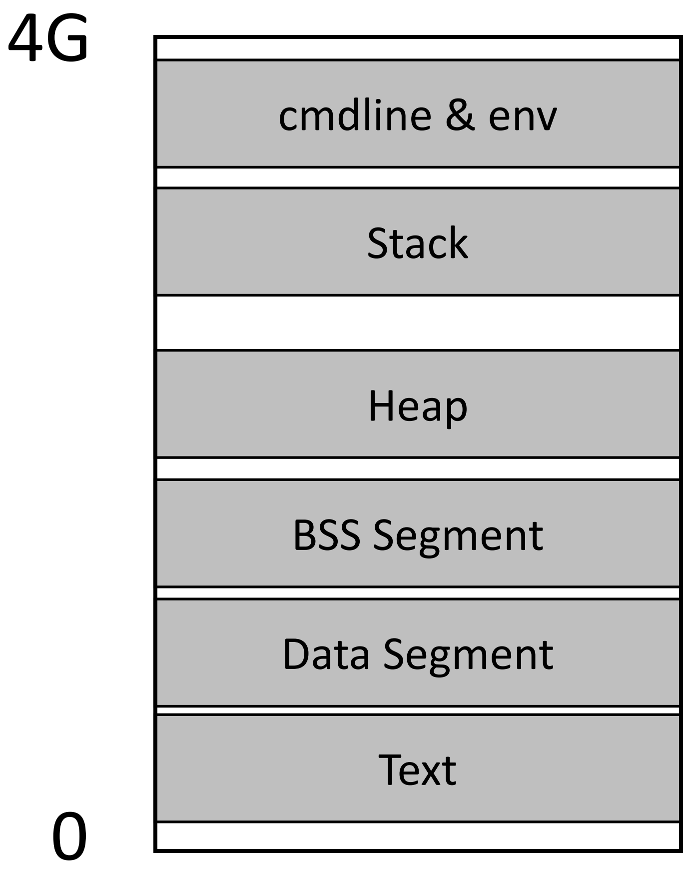
# Program Layout in Memory




```
int x = 100;
int main()
{

   int  a=2;
   float b=2.5;
   static y;

   int *ptr = (int *) malloc(2*sizeof(int));

   ptr[1]=5;
   ptr[2]=6;

   free(ptr)

   return 1;
}
```

**Where would variables be located?**

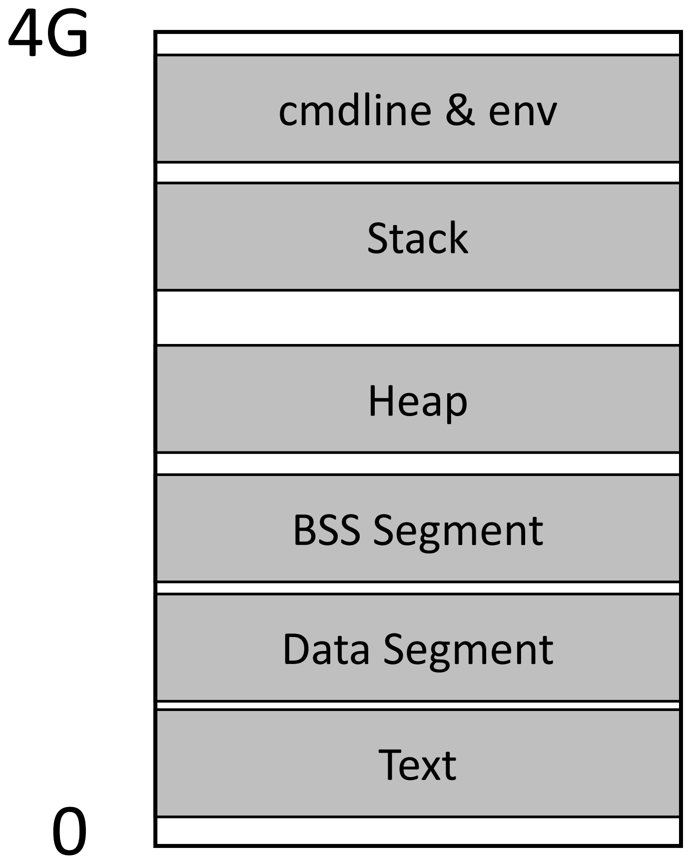
# Program Layout in Memory




```
int x = 100;
int main()
{
   int  a=2;
   float b=2.5;
   static y;

   int *ptr = (int *) malloc(2*sizeof(int));

   ptr[1]=5;
   ptr[2]=6;

   free(ptr)

   return 1;
}
```

**Where would variables be located?**

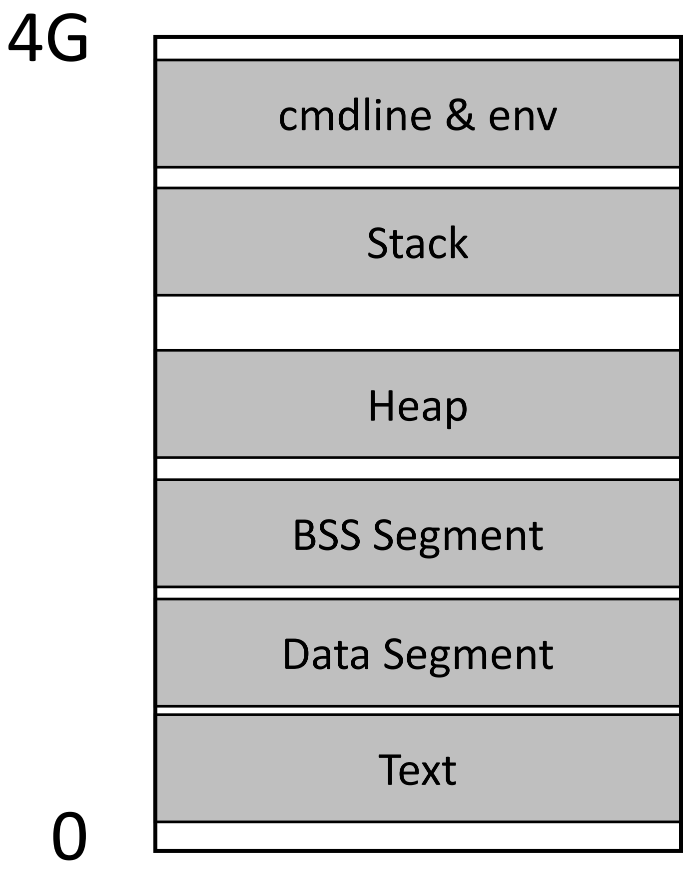
# Program Layout in Memory




```
int x = 100;
int main()
{

  int  a=2;
  float b=2.5;
  static y;

  int *ptr = (int *) malloc(2*sizeof(int));

  ptr[1]=5;
  ptr[2]=6;

  free(ptr)

  return 1;
}
```

**Where would variables be located?**

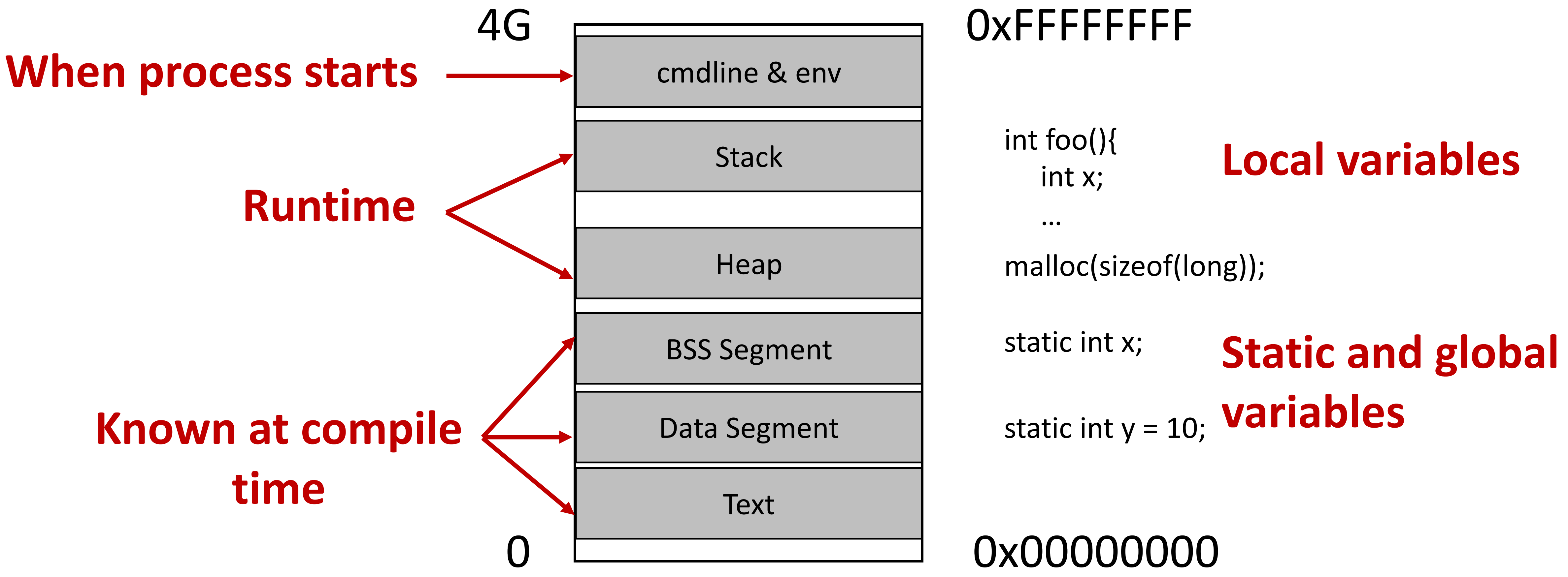
# Program Layout in Memory




```
int x = 100;
int main()
{

   int  a=2;
   float b=2.5;
   static y;

   int *ptr = (int *) malloc(2*sizeof(int));

   ptr[1]=5;
   ptr[2]=6;

   free(ptr)

   return 1;
}
```

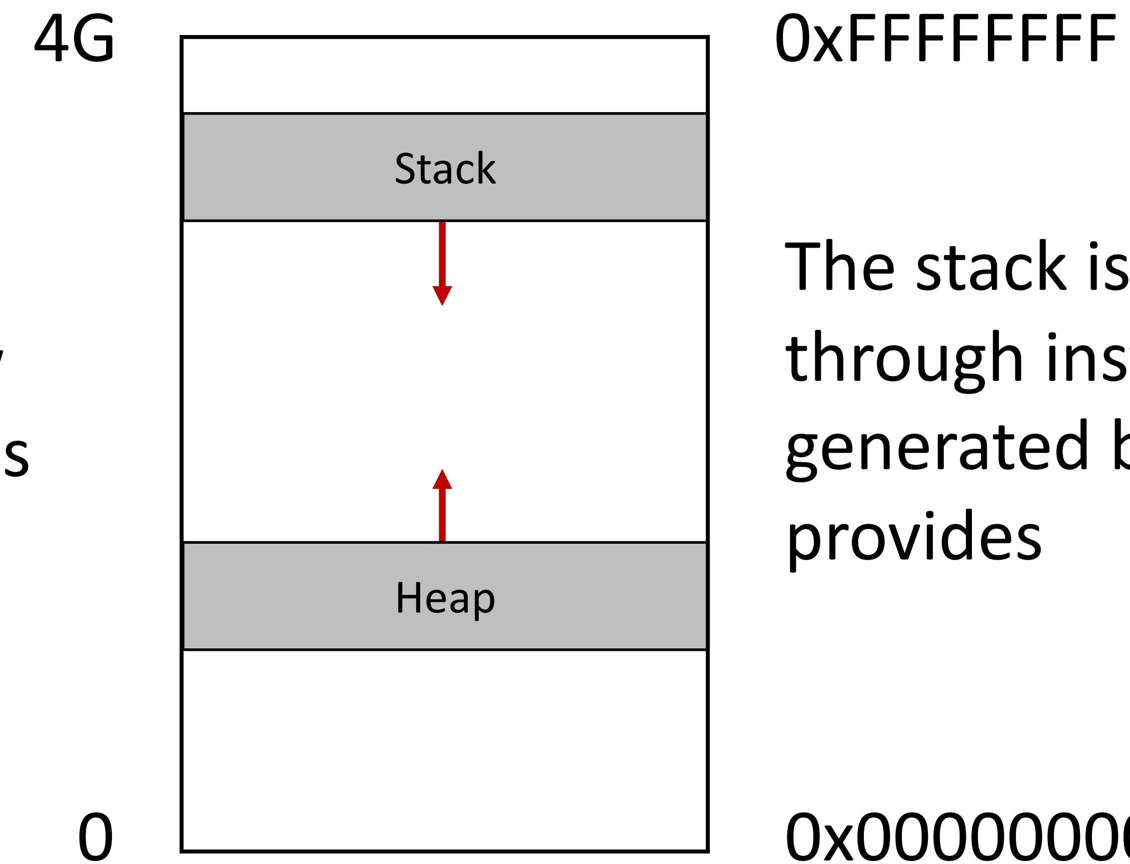
**Where would variables be located?**

# Program Layout in Memory



# Focus on Stack-based Attacks

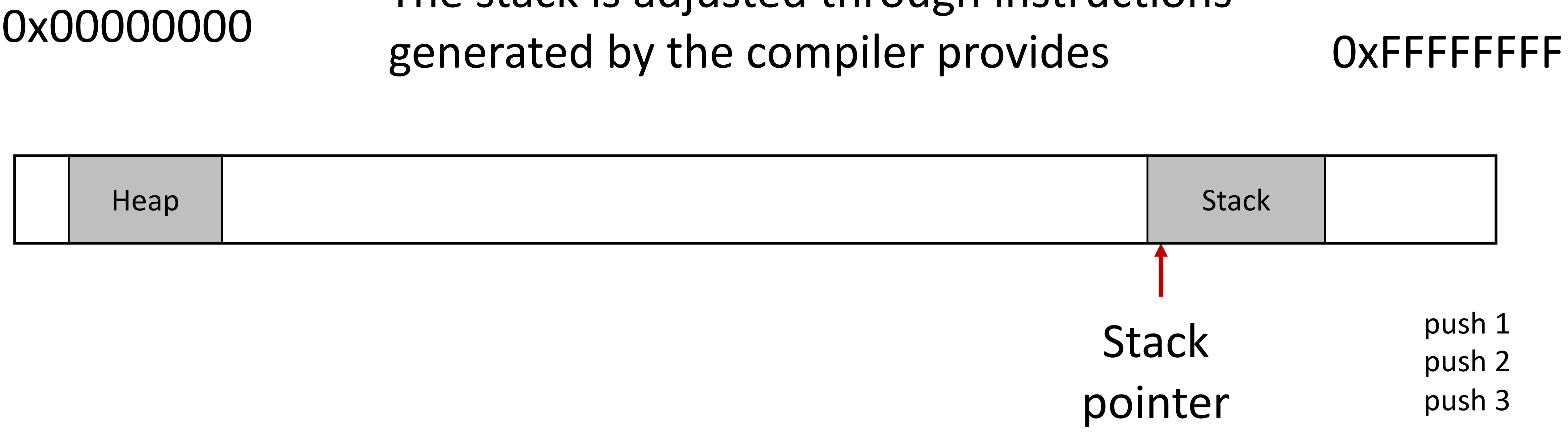
Stack and heap grow in opposite directions




The stack is adjusted through instructions generated by the compiler provides

# Focus on Stack-based Attacks

The stack is adjusted through instructions generated by the compiler provides



# Focus on Stack-based Attacks

0x00000000

The stack is adjusted through instructions generated by the compiler provides

0xFFFFFFFF


The heap is allocated by the OS and managed by the process through malloc()

Stack pointer

push 1
push 2
push 3

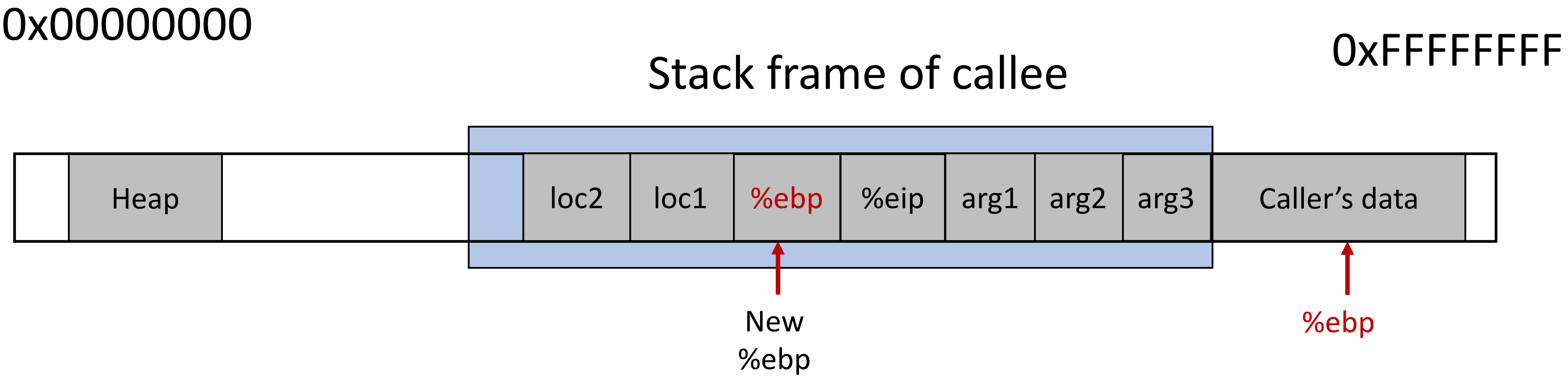
# Function Calls

```
int main() {
  …
  foo(1, 2, 3);
  …
}
```

```
void foo(int arg1, int arg2, int arg3) {
  char loc1[4];
  int loc2;
  …
}
```

- Caller:
  - Push arguments onto stack in reverse order
  - Push return address
    - %eip + sizeof( curr inst.)
  - Branch to function address
  - Restore stack by popping arguments
- Callee:
  - Push old frame pointer (%ebp)
  - Set %ebp to top of stack (where old %ebp stored)
  - Push local variables
  - …
  - Restore old stack frame
    - %esp = %ebp; pop %ebp
  - Branch to return address: pop %eip

# Function Calls



# Summary of Function Calls

- Calling function:
  - Push arguments onto the stack in reverse order
  - Push the return address of the next instruction to be run in the calling function
    - %eip + sizeof(current instruction)
  - Branch to the function’s address

- Called function:
  - Push the old frame pointer onto the stack (%ebp)
  - Set the new frame pointer %ebp to where the old %ebp was pushed
  - Push local variables onto the stack

# Summary of Function Calls

- Returning to calling function:
  - Reset the previous stack frame
    - %ebp = (%ebp)
    - Need to copy %ebp into another register first
  - Jump back to the return address
    - %eip = 4(%ebp)
    - Need to use copied value of ebp (current stack frame)

# Stack Layout Example

- Stack Frame

```
void foo(int a, int b) {
  int x, y;
  x = a+b;
  y = a – b;
}
```

foo(5, 6);


What does the stack frame look like?

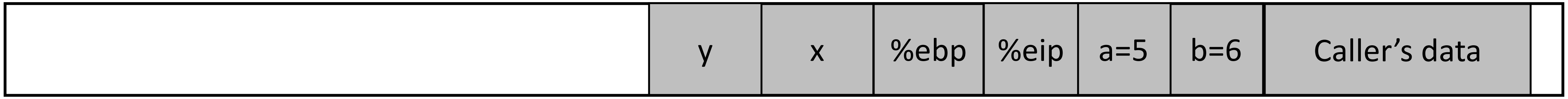
# Stack Layout Example

- Stack Frame

```
void foo(int a, int b) {
  int x, y;

  x = a+b;
  y = a – b;
}
```

foo(5, 6);

How do we reference a, b, x, y?




Binary code is generated during compilation stage!

# Stack Layout Example

- Stack Frame

```
void foo(int a, int b) {
  int x, y;

  x = a+b;
  y = a – b;
}
```

foo(5, 6);

How do we reference a, b, x, y?

- Frame Pointer

```
movl 12(%ebp), %eax
movl 8(%ebp), %edx
addl %edx, %eax
movl %eax, -4(%ebp)
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | y | x | %ebp | %eip | a=5 | b=6 | Caller’s data | |

Binary code is generated during compilation stage!

Compiler uses offsets relative to ebp

# Copying Data to a Buffer

```
int main() {
  …
  char src[40] = “Hello world \0 Extra string”;
  char dest[40];

   strcpy(dest, src);

   return 0;
}
```

A buffer overflow involves copying data to a buffer

# Copying Data to a Buffer

```
int main() {
  ...
  char src[40] = “Hello world \0 Extra string”;
  char dest[40];

   strcpy(dest, src);

   return 0;
}
```

What is this?

A buffer overflow involves copying data to a buffer

# Copying Data to a Buffer

```
int main() {
  …
  char src[40] = "Hello world \0 Extra string";
  char dest[40];

   strcpy(dest, src);

   return 0;
}
```

What is this?

Tells compiler to insert 0x0 in binary

A buffer overflow involves copying data to a buffer

# Copying Data to a Buffer

```
int main() {
  …
  char src[40] = “Hello world \0 Extra string”;
  char dest[40];

   strcpy(dest, src);

   return 0;
}
```

What is this?

Tells compiler to insert 0x0 in binary

Different ways to copy data

strcpy()

How does strcpy do the copy?

memcpy()

Needs size

A buffer overflow involves copying data to a buffer

# Buffer Overflow

```c
void foo (char *str) {
  char buffer[12];
  strcpy(buffer, str);
}
int main() {
  char *str = “This is definitely longer than 12”;
  foo(str);
  return 0;
}
```

What will happen after this?

| | | | | | | |
|---|---|---|---|---|---|---|
| | buffer[0]…buffer[11] | %ebp | %eip | str | Caller’s data | |

Buffer copy

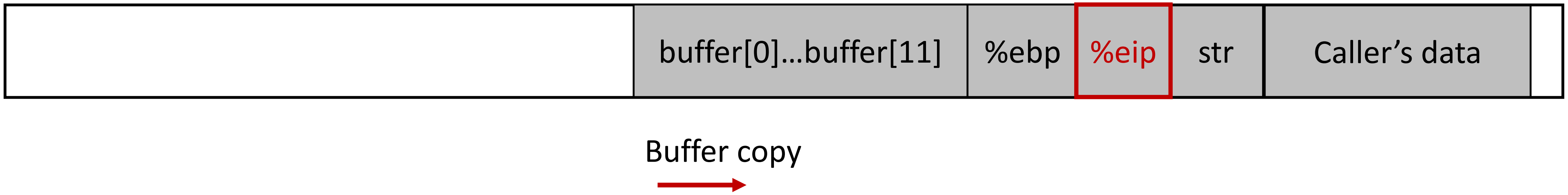
# Buffer Overflow

```
void foo (char *str) {
  char buffer[12];
  strcpy(buffer, str);
}
int main() {
  char *str = “This is definitely longer than 12”;
  foo(str);
  return 0;
}
```

Execute unmapped address

Jump to protected place

Invalid instruction

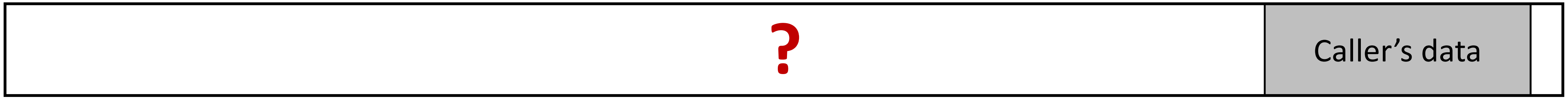


# Buffer Overflow Example 1

```
void foo (char *arg1) {
  char buffer[4];
  strcpy(buffer, arg1);
  …
}
int main() {
  char *str = “AuthMe!”;
  foo(str);
  …
}
```

What will this code do?

Describe the stack layout after foo() is called?



# Buffer Overflow Example 1

```
void foo (char *arg1) {
   char buffer[4];
   strcpy(buffer, arg1);
   …
}
int main() {
   char *str = “AuthMe!”;
   foo(str);
   …
}
```

What will happen to the program?

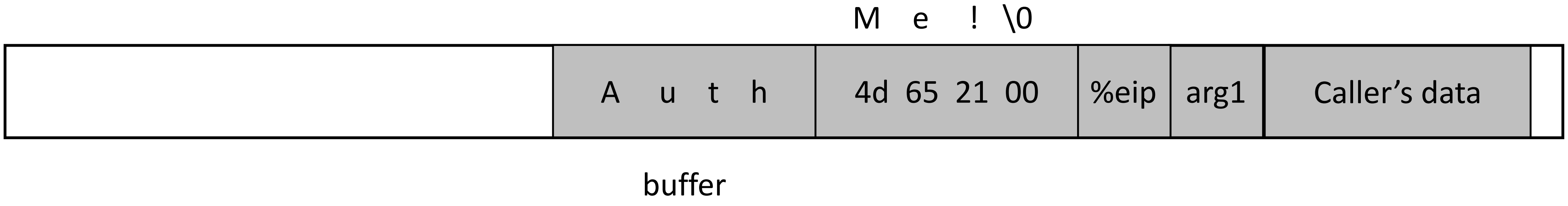
| | | M e ! \0 | | | | |
|---|---|---|---|---|---|---|
| | A u t h | 4d 65 21 00 | %eip | arg1 | Caller’s data | |
| | buffer | | | | | |

# Buffer Overflow Example 1

```
void foo (char *arg1) {
  char buffer[4];
  strcpy(buffer, arg1);
  …
}
int main() {
  char *str = “AuthMe!”;
  foo(str);
  …
}
```

What will happen to the program?

**Crash with SEGFAULT due to bad %ebp**

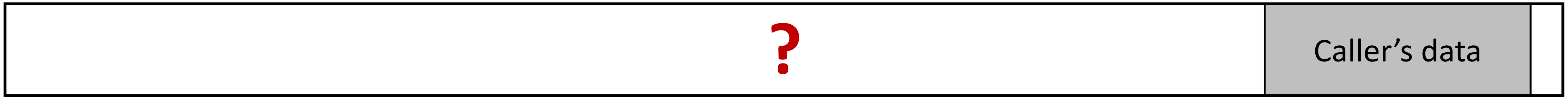


# Buffer Overflow Example 2

```
void foo (char *arg1) {
  int authenticated = 0;
  char buffer[4];
  strcpy(buffer, arg1);
  if(authenticated) {…}
}
int main() {
  char *str = “AuthMe!”;
  foo(str);
  return 0;
}
```

What will this code do?

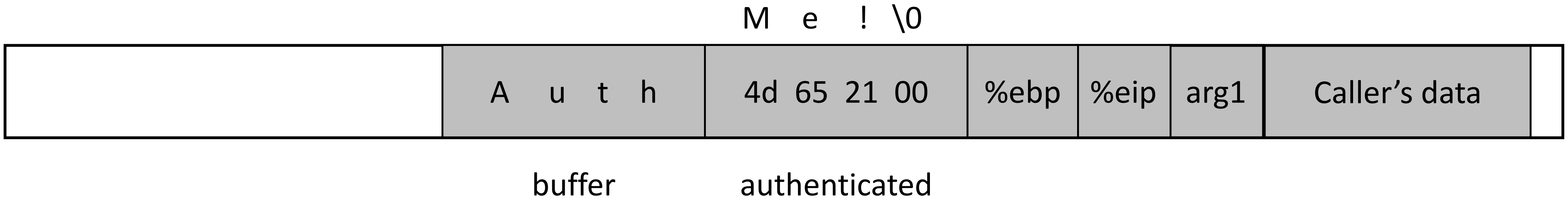
Describe the stack layout after foo() is called?



# Buffer Overflow Example 2

```
void foo (char *arg1) {
  int authenticated = 0;
  char buffer[4];
  strcpy(buffer, arg1);
  if(authenticated) {…}
}
int main() {
  char *str = "AuthMe!";
  foo(str);
  return 0;
}
```

**The user is now authenticated without any crashes**



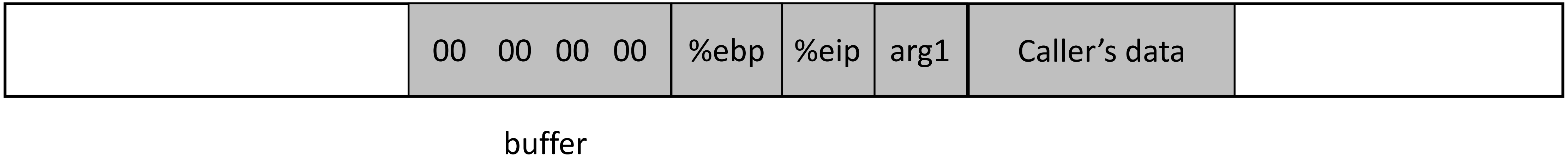
# Most Programs Process User Input

- Previous examples used hardcoded strings
- Most useful programs require some level of interaction with the user
- Users can supply input through a multitude of mechanisms including text input, packets over the networks, environment variables, and file input

# What Can We Do with User Input?

```
void foo (char *arg1) {
  char buffer[4];
  strcpy(buffer, arg1);
  …
}
```

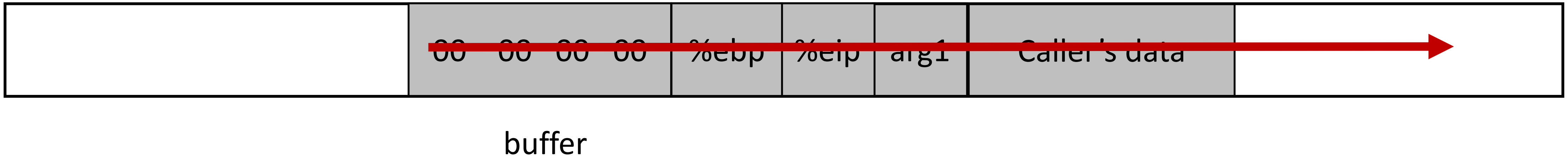
What can we do with user input to make this more interesting?



# What Can We Do with User Input?

```
void foo (char *arg1) {
   char buffer[4];
   strcpy(buffer, arg1);
   ...
}
```

What can we do with user input to make this more interesting?




strcpy() allows you to overwrite memory until \0 is encountered

What can you do with this knowledge?

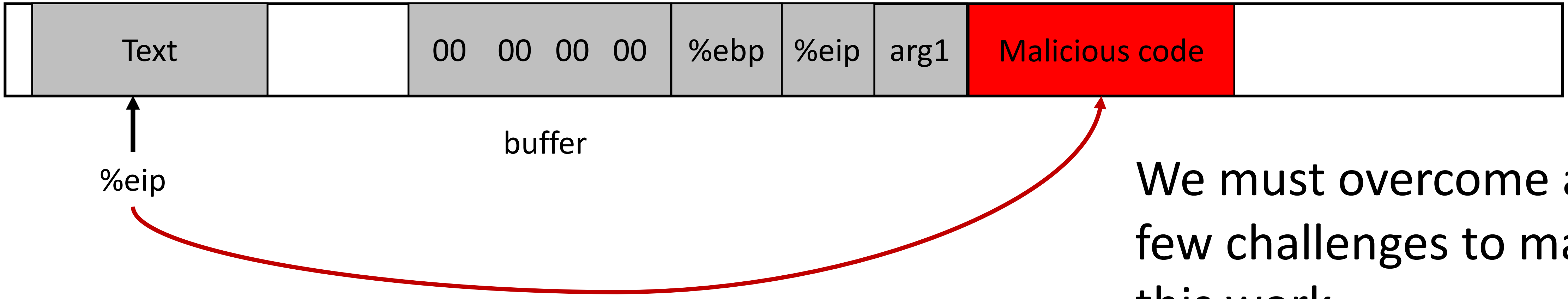
# Code Injection

# Overview

```
void foo (char *arg1) {
  char buffer[4];
  sprintf(buffer, arg1);
  …
}
```

Goal:

- Use input as attack surface
- Insert user supplied code into memory
- Set %eip to point to user code




We must overcome a few challenges to make this work

# Challenge 1

- Must directly load machine code into memory (instructions we want to see executed)
- The machine code must not contain any zeros
  - Zeros would cause sprintf(), gets(), scanf() to stop copying
- Need to run a general purpose shell that provides attacker with easy access to system resources

# Shellcode

```
int main() {
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Shellcode is code that spawns a shell

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...
```

Write code in assembly

Assembler

```
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
...
```

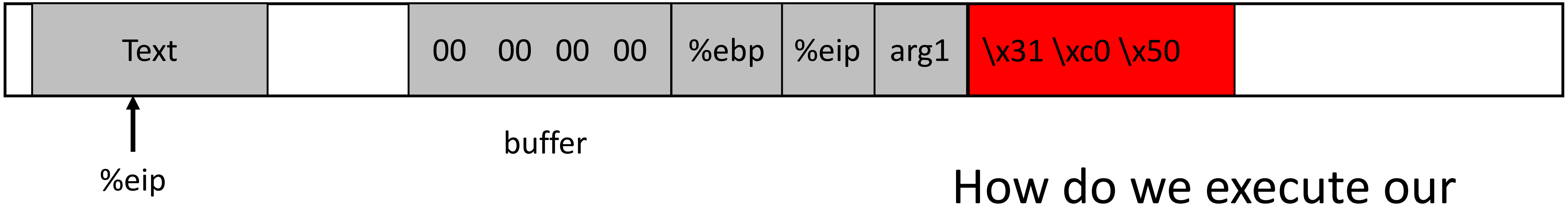
Machine code

# Shellcode Example

```
Line 1: xorl %eax,%eax
Line 2: pushl %eax            # push 0 into stack (end of string)
Line 3: pushl $0x68732f2f     # push "//sh" into stack
Line 4: pushl $0x6e69622f     # push "/bin" into stack
Line 5: movl %esp,%ebx        # %ebx = name[0]
Line 6: pushl %eax            # name[1]
Line 7: pushl %ebx            # name[0]
Line 8: movl %esp,%ecx        # %ecx = name
Line 9: cdq                   # %edx = 0
Line 10: movb $0x0b,%al
Line 11: int $0x80            # invoke execve(name[0], name, 0)
```

# Challenge 2

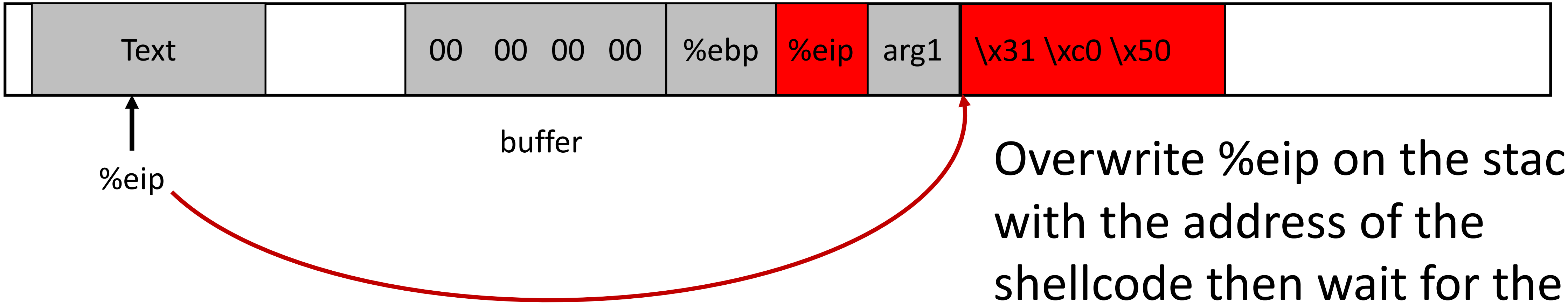
- We can only write to memory sequentially
- We need to have a way to execute code from code that's already executing




How do we execute our shellcode?

# Challenge 2

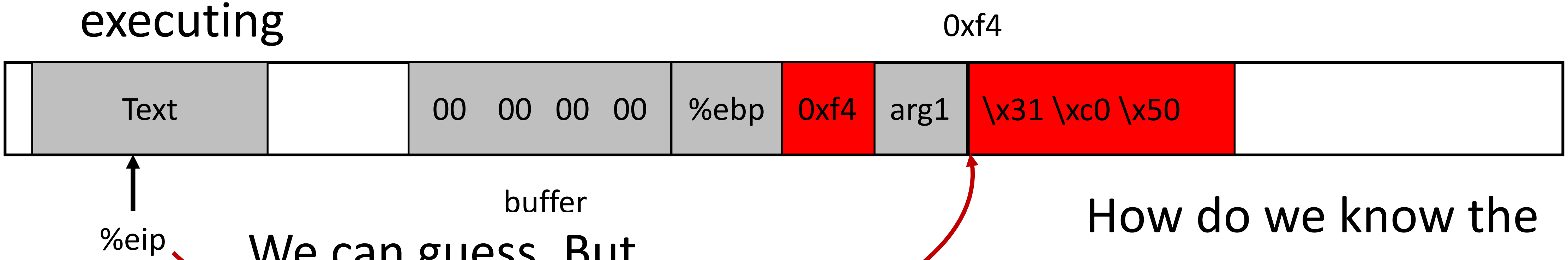
- We can only write to memory sequentially
- We need to have a way to execute code from code that's already executing




Overwrite %eip on the stack with the address of the shellcode then wait for the function call to return

# Challenge 2

- We can only write to memory sequentially (cannot skip specific regions)
- We need to have a way to execute code from code that's already executing




We can guess. But what if we're wrong?

How do we know the address to use?

Possibly panic if invalid instruction (i.e. data)

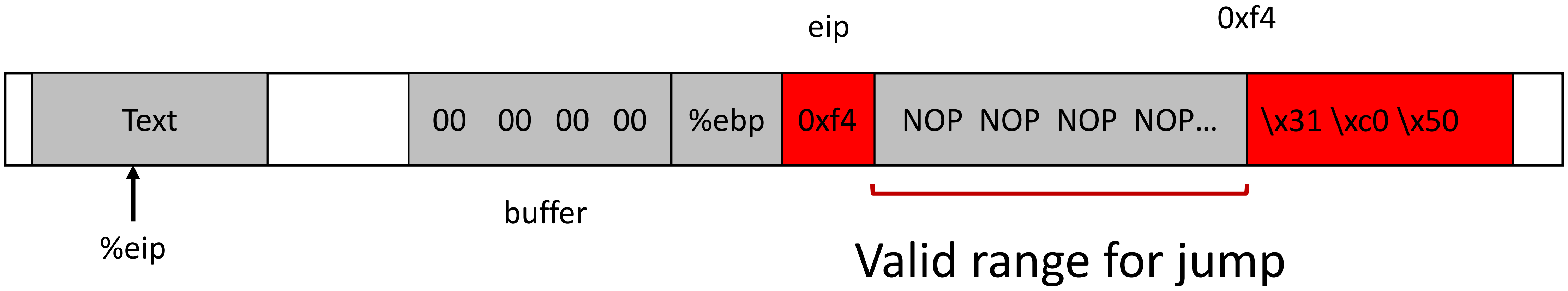
# Challenge 3

- We need to determine the location of the return address on the stack
  - Where %eip is saved
  - We don't know how far %ebp is from the buffer

- We could brute force the address space and try all 2^32 addresses on a 32-bit machine
- Can be done more efficiently if address space layout randomization (ASLR) is disabled
  - The stack will always start from a fixed location
  - Most programs don't have a deep stack

# NOP Sleds

- Inserting NOPs in the malicious code can improve our chances
- A NOP will just increment the value of the %eip and move to the next instruction
- Chance of succeeding improves according to the number of inserted NOPs



# Users and Groups

# Users and Groups

- Two primary users on Unix/Linux systems: root vs. non-root
- Each user is assigned a unique ID (uid)
  - uid = 0 is reserved for root (super user)
- Users need to login with their password
  - User information is stored in /etc/passwd
    - /etc/passwd used to contain the password, but has now been moved to a different file
    - Example: john:x:30000:40000:John Doe:/home/john:/bin/bash

username Password uid gid gecos Home dir shell

# Users and Groups

- The encrypted password is stored in /etc/shadow
  - `john:$6$Etg2ExUZ$F9NTP7omafhKIlqaBMqng1:15651:0:99999:7: : :`

    1 2 3 4 5 6 7 8 9

- The fields are as follow:
  - 1: username
  - 2: encoded password
  - 3: days since the UNIX time that the password was changed
  - 4: minimum number of days before password can be changed (0 means allow password changes anytime)
  - 5: maximum number of days the password is valid (99999 means user can keep their password unchanged forever)
  - 6: number of days before user is warned about password expiration
  - 7: number of days after password expires that the account is disabled (inactive)
  - 8: days since the UNIX time the account is disabled (expiration)
  - 9: reserved field

# Users and Groups

- The password field is further broken down into the subfields (notice $ in :$6$Etg2ExUZ$F9NTP7omafhKIlqaBMqng1:)
  - 6: is the ID of the algorithm, in this case SHA512 hashing algorithm
  - Etg2ExUZ: is a salt
  - F9NTP7omafhKIlqaBMqng1: is the hash(salt + password) **Why do we do this?**

# Users and Groups

- Sometimes it is convenient to assign permissions to a group of users for accessing common resources
- A user can be a member of multiple groups
- Group member information is stored in /etc/group
  - *# groups uid* (will display the groups a given uid belongs to)

# File Permissions

- Permissions on files:
  - 3 attributes (bits) are used to describe permissions
    - Owner(u), Group(g), and Others(o)
    - Readable(r), Writable(w), and Executable(x)
    - Example: -rwxrwxrwx which is equivalent to 777
- Permissions on directories:
  - r: the directory can be listed
  - w: can create/delete a file or directory within the directory
  - x: the directory can be entered
  - *chmod* is used to change permissions
- Default file permission:
  - The default file permission assigned to a user is controlled through the *umask* environment variable
  - *umask* contains bits set for the permissions you don’t want to provide
    - Example: *umask 077* will set the permission for newly created files to rwx---r-- (non-execultable)

**Why does 644 mean?**

# Security Related Commands

- Change your user ID to xyz with *su* (substitute user)
  *su xyz*

- To change your user to root you run the command below. Once root, you get # as a prompt
  *su -*

- Running a command using superuser privilege without logging in as root is useful. We can use *sudo* for that
  - Example: to view the shadow file as a superuser
    *sudo more /etc/shadow*
  - To be able to use sudo, the superuser (root) must grant permission to the user by adding them to the list of sudoers (/etc/sudoers)
  - To change ownership of a file, use *chown*
    *chown john filename*

# Privilege Escalation

# Set-uid

- How can a user run *passwd* the command to change their password, but can’t access the /etc/shadow file?

# Set-uid

- How can a user run *passwd* the command to change their password, but can’t access the /etc/shadow file?
- Each process has a real uid (ruid) and an effective uid (euid)
  - When a user logs in, the effective uid is the same as the real uid
  - The effective uid can change temporarily to allow privileged access to resources
    - Without this ability most programs would be useless
- In addition to rwx attributes, each executable file has a set-uid bit
  - If the set-uid bit is set on a program, the euid will be set to the owner id when entering the executable
  - euid is set back to the ruid after returning from the executable

# END