CS 366
Intro to Computer Security

Dr. Stella Sun
EECS
University of Tennessee
Fall 2022

Today’s Class

- Buffer overflow attacks
- Defenses

Benign Program with Buffer ~~Overflow~~

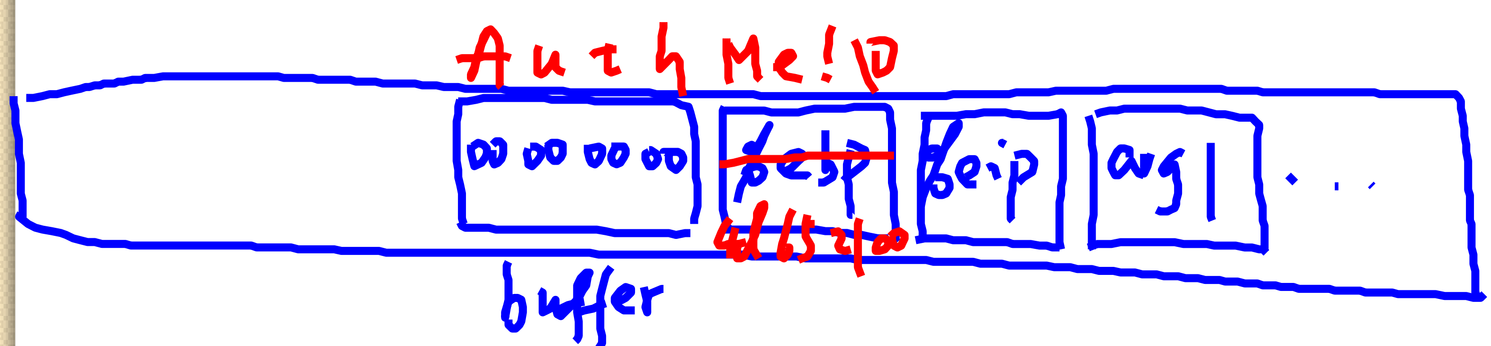
```
void func(char *arg1)
{
     char buffer[4];
     strcpy(buffer, arg1);
     ...
}

int main()
{
     char *mystr = "AuthMe!";
     func(mystr);
     ...
}
```

upon return, set %ebp to 0x0021654d

⇒ Segfault

⇒ Crash!

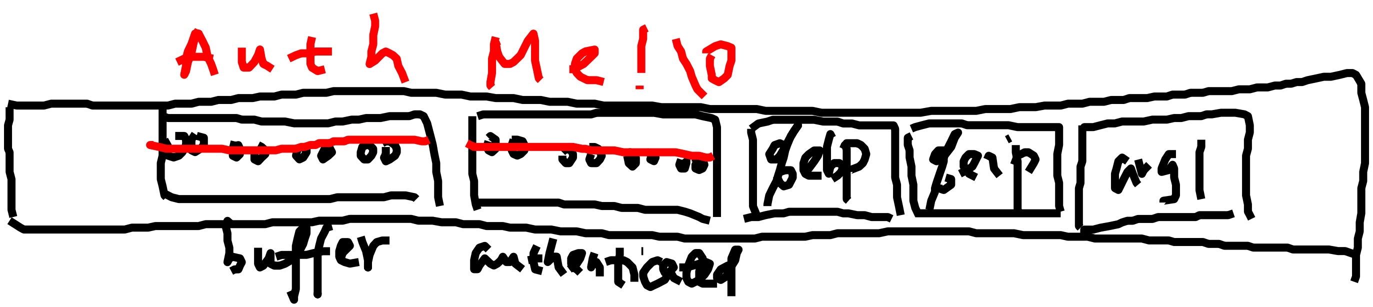


Malicious Program with Buffer Overflow

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

call stack:

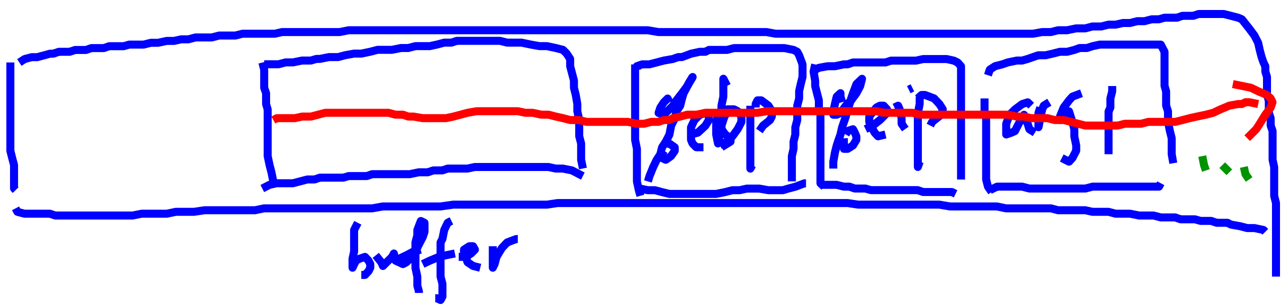


What Worse Can Be Done

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

culprit

Attacker's



Strcpy lets you write as much as you want until a "\0"

As an attacker, you want to write code (action!)

In reality, strings come from...

- Users, e.g.,
  - text input
  - file input
  - packets
  - environment variables
  - ...

- Validating assumptions about user input is extremely important!

Code Injection

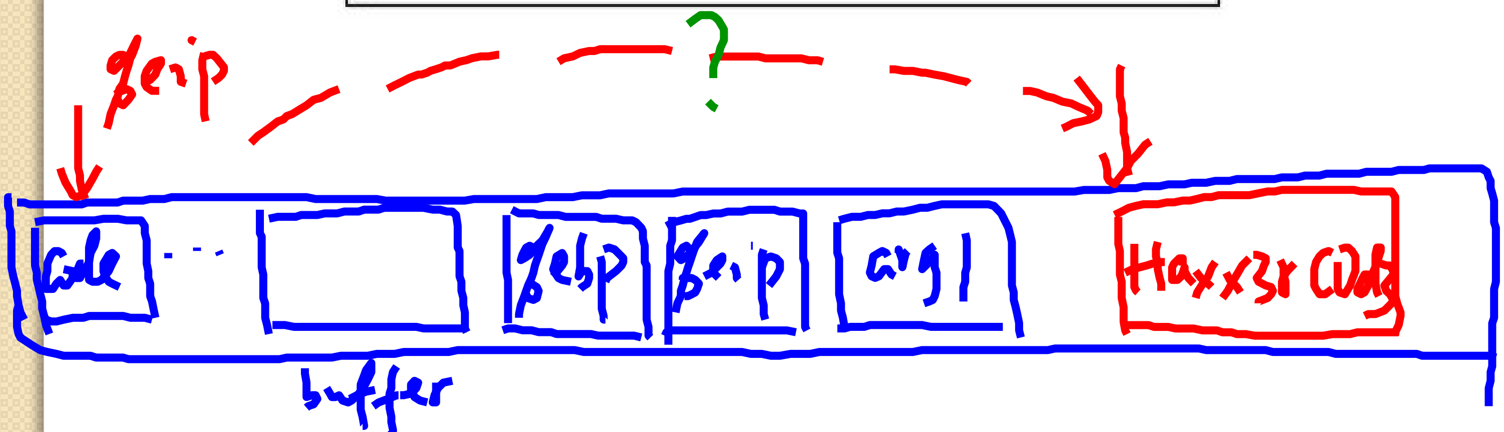
- Basic idea:

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

Code Injection

- Basic idea:

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



(1) Load my own code into memory

(2) Somehow get `%eip` to point to it

Code Injection

- How to load code into memory
- What code to load
- How to get the code to run

How to Load Code into Memory

- By exploiting a buffer overflow vulnerability in the program and overrun the buffer

What Code to Load

- Must be machine code: compiled and ready to run

- Can't contain all-zero bytes

  - otherwise, sprintf/scanf/gets…will stop copying

- Can't use the loader: must be self-contained

What Code to Load

- Best choice: general-purpose shell
  - a command line prompt that gives attacker general access to the system

- The code to launch a shell: shellcode

Shellcode Example

```c
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = “/bin/sh”;
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Assembly

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

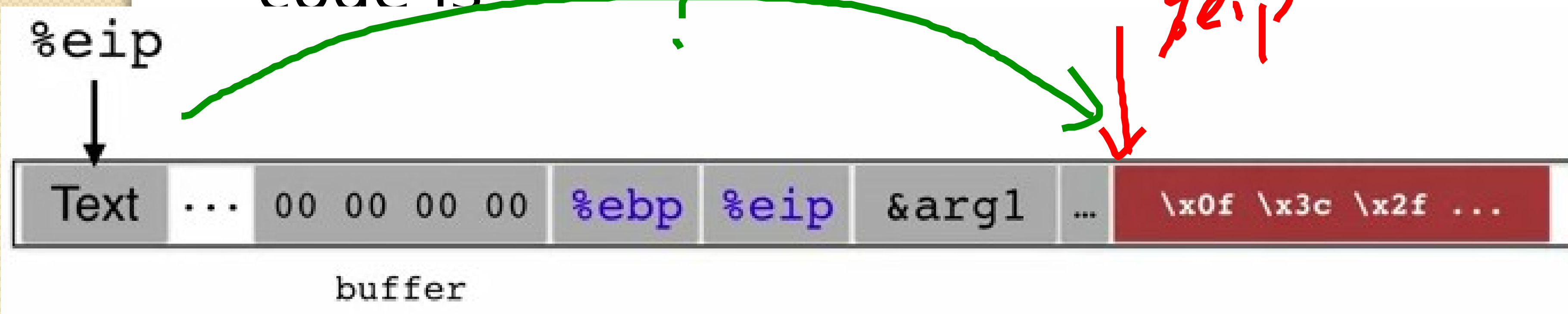
Machine code

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

How to Get The Code to Run

Attacker:

- Can’t insert a “jump to this code” instruction
- Doesn’t know precisely where the code is

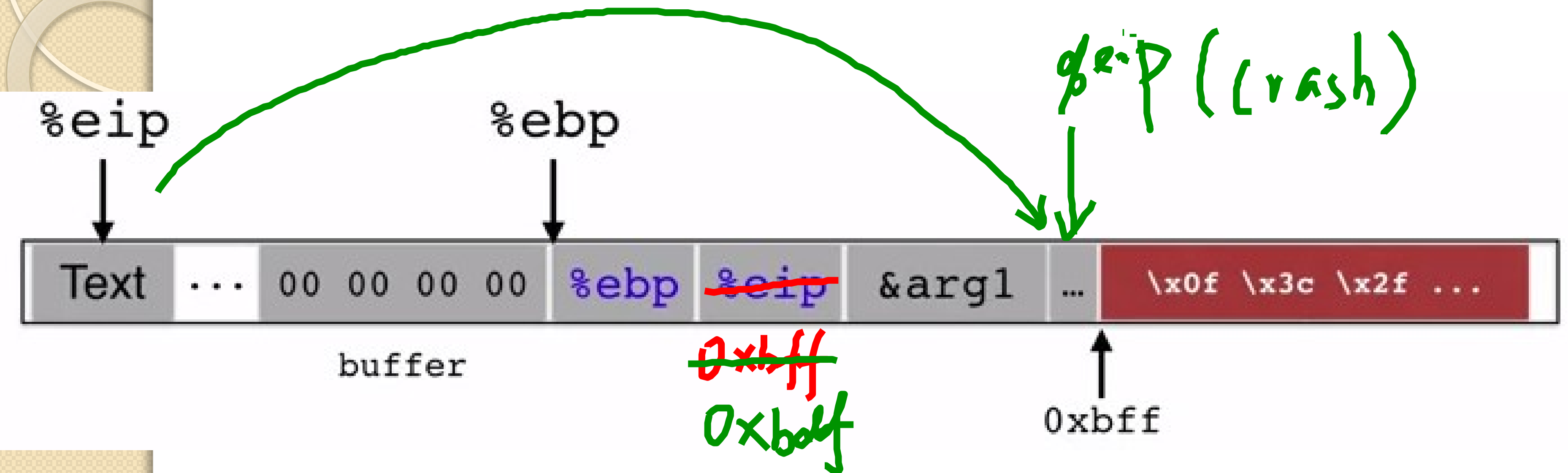


Recall: Memory Layout Summary

- Calling function
  - Push arguments onto stack in reverse
  - Push the return address (%eip): the address of instruction you want run after control returns to you
  - Jump to the function's address
- Callee function
  - Push the old frame pointer (%ebp) onto stack
  - Set frame point (%ebp) to the top of the stack right now (%esp)
  - Push local variables onto stack
- Returning function
  - Reset the previous stack frame: %esp=%ebp, %ebp=(%ebp)
  - Jump back to return address: %eip=4(%esp)

Attacker modifies %eip to point to his code

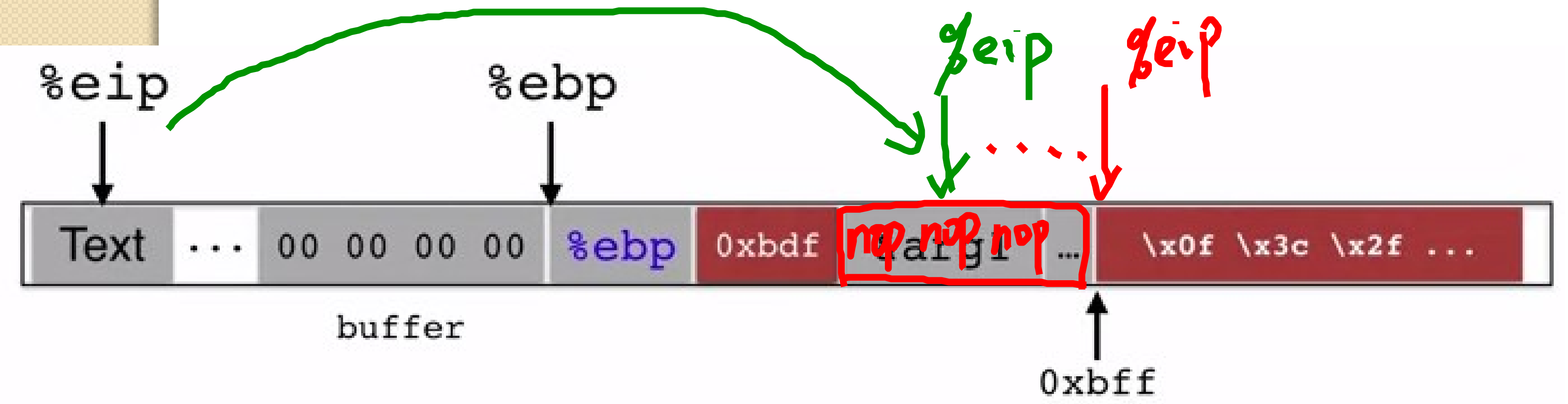
Manipulating The Saved %eip



1. Put 0xbff into %eip
2. But how does attack know 0xbff?
   - Most likely he doesn't know and needs to guess

Increasing Attacker's Chance

- Classic way: nop sled
- Single-byte instruction that does nothing and just moves on



But a bunch of nops look ~~suspicious~~

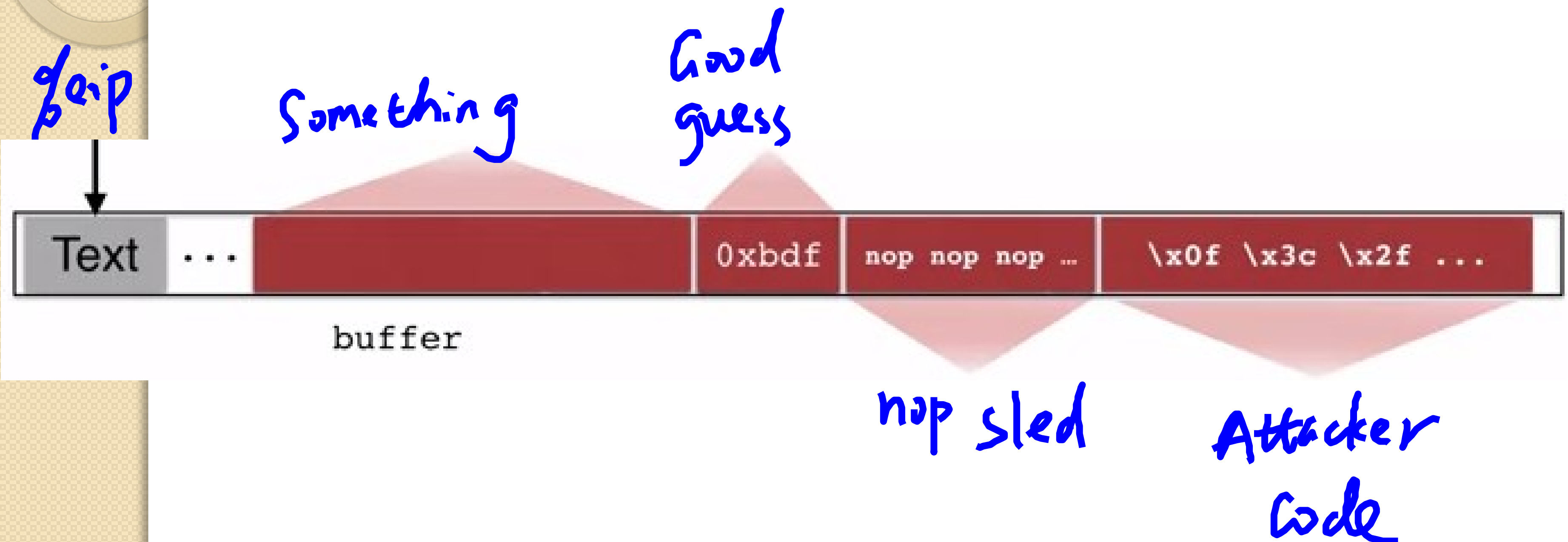
- More nops enlarge the attack surface

- But this pattern is easily detectable

- Various evasion techniques, e.g.,
  - move the register to itself
  - add zero to the register
  - add one to the register and then subtract one
  - … to make the instructions look normal

Code Injection Summary

- The injection technique we covered:



- This is the most commonly referred-to buffer overflow exploit, called stack smashing

One More Thing…

- If the attacker doesn't have access to the target code

  - he doesn't know how far buffer is from %ebp

  - thus he doesn't know where %eip is to overwrite

- He can try guessing

  - worse case: trying all 2^32 memory locations

- But he doesn't need to

  - the stack usually starts at a fixed address (unless address randomization is used)

  - the stack will grow but usually not very far (unless code has heavy recursion)

Other Buffer Overflow Attacks

- What we just saw is stack buffer overflow (write)

- There are other attacks that exploit bugs in buffer

  - heap overflow, integer overflow, read overflow, format string vulnerability

- Which one(s) of CIA does buffer overflow violate?

  - Confidentiality
  - Integrity
  - Availability

Heap Overflow

```
typedef struct _vulnerable_struct {
  char buff[MAX_LEN];
  int (*cmp)(char*,char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two)
{
 strcpy( s->buff, one );
 strcat( s->buff, two );
 return s->cmp( s->buff, "file://foobar" );
}
```

← copy one into buffer

← concatenate two into buffer

Must have: strlen(one) + strlen(two) < MAX_LEN

Otherwise: overwrite s->cmp

Integer Overflow

```
void vulnerable()                    OpenSSH 3.3
{
            Huge
  char *response;
  int nresp = packet_get_int();                   wrap-around
  if (nresp > 0) {
   response = malloc(nresp*sizeof(char*));
   for (i = 0; i < nresp; i++)        4bytes
    response[i] = packet_get_string(NULL);
  }
    overflow
```

- If attacker sets nresp to a large number, say, ~1 billion
- Then nresp*sizeof(char*) wraps around to 0
- Any writes to allocated response will overflow it

Data Corruption

- The attacks we have seen so far corrupt *code*
  - return address
  - function pointer
  - ...

- They can also corrupt data
  - modify the secret key to use the value the attacker knows
  - modify state variables to bypass checks (e.g., the authenticated flag we saw)
  - modify interpreted strings used as part of commands (e.g., SQL injection)

Read Overflow

- So far we have focused on write overflow
- In read overflow, a bug can permit reading past the end of a buffer rather than writing past the end
  - can result in information leakage

Read Overflow

```
int main() {
  char buf[100], *p;
  int i, len;
  while (1) {
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   len = atoi(p);
   p = fgets(buf,sizeof(buf),stdin);
   if (p == NULL) return 0;
   for (i=0; i<len; i++)
    if (!iscntrl(buf[i])) putchar(buf[i]);
    else putchar('.');
   printf("\n");
}}
```

Read integer

Read message

Echo-back the message

Sample Output

```
% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
10
hello there
ECHO: |hello ther|
25
hello
ECHO: |hello..here..y does fine.|
```

read overflow

Format String Attack

- Formatted I/O: C's printf family

```
void print_record(int age, char *name)
{
  printf("Name: %s\tAge: %d\n",name,age);
}
```

- Format specifiers
  - position in string indicates stack arguments to print
  - type of specifier indicates type of arguments

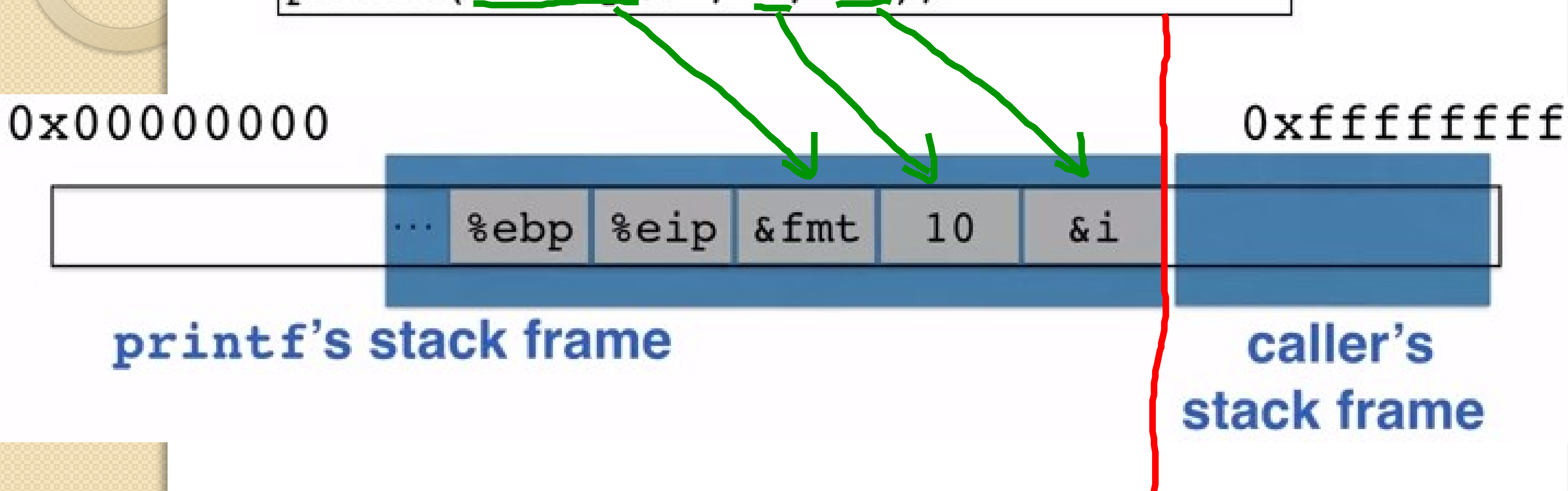
Difference between The Two?

```
void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf("%s",buf);
}
```

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

Let’s recall the printf ~~implementation~~

```
int i = 10;
printf("%d %p\n", i, &i);
```

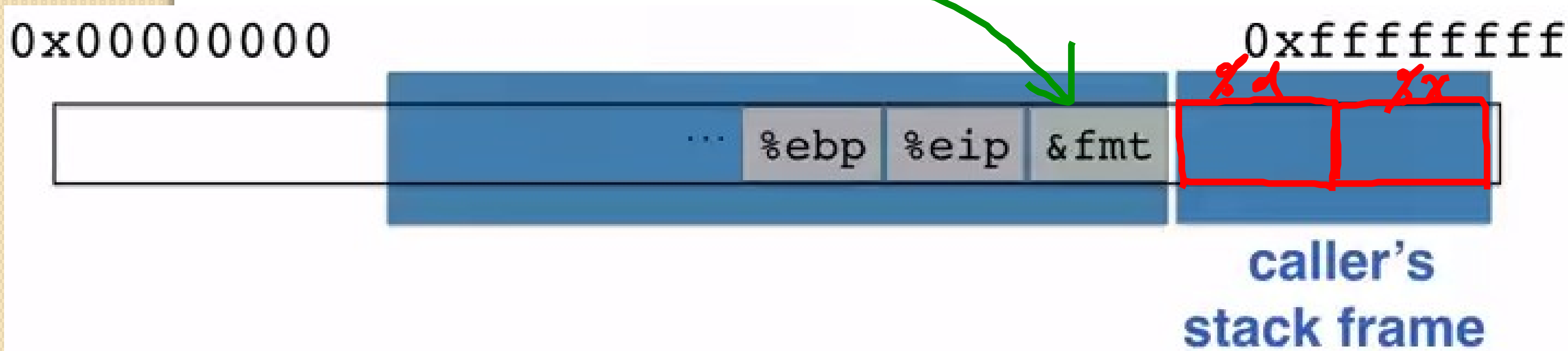


- Printf takes variable number of arguments
- It doesn’t care where its stack frame ends
- It assumes that it’s called with as many arguments as format specifiers

This Can Be Exploited

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

"%d %x"



Example Vulnerabilities

- printf(“%s”);
  - prints stack entry as a string
- printf(“%d %d %d …”);
  - prints a series of stack entries as integers
- printf(“%08x %08x %08x…”);
  - prints a series of stack entries as 8-digit hex
- printf(“100% decent!”);
  - prints stack entry 4 bytes above the saved %ei
- printf(“100% new”);
  - writes 3 (100) to location pointed to by stack entry

Why Is Format String Attack A Type of Buffer Overflow?

- In the sense that

    - the printf stack itself can be considered a buffer

    - the number and size of arguments passed to the function determines the buffer size

    - providing a malicious format string causes program to overflow this buffer → read or write overflow