

CSE 127 Week 2

Discussion

Fall 2026

Agenda

- Overview the assignment
- Review memory layout and buffer overflow
- Go over a few targets and give hints

Overview

- Due: Thursday, October 22, 2026 by 11:59 PM PDT
- Total Points: 20 (+ 4 optional extra credit)
- 7 Targets to hack + 1 extra credit Target
 - Targets increase in difficulty
 - Defense mechanisms (next week)
 - Extra credit is hard. Only attempt after finishing the rest.
- Targets: short C programs each with some security vulnerabilities.
 - Exploit those vulnerabilities with malicious input
 - Requirements differ, but most require opening of root shell.

Setting up

- Windows, Linux, and Intel Macs: Download the [.ova](#) file and import it into [VirtualBox](#).
- M-series Mac users: Download the [.zip](#) to get a UTM file and use [UTM](#) to open it.
- Other ARM64 devices: Download the [.ova](#) file and extract to get a VirtualBox vmdk file.

Install [qemu](#) and run

```
qemu-system-x86_64 \  
-cpu qemu64 \  
-m 4096 \  
-device e1000,netdev=net0,mac=08:00:27:D3:23:52 \  
-netdev user,id=net0,hostfwd=tcp::2222-:22 \-drive  
file=/Path/to/pa2box/CSE127-PA2-disk001.vmdk
```

- SSH with: `ssh -p 2222 cse127@localhost`

- Download the starter code:

- `wget https://cseweb.ucsd.edu/classes/fa24/cse127-a/resources/pa2-starter.tar.gz`

- Unzip with: `tar -xvf pa2-starter.tar.gz`

Generating cookies

- Run `./build.sh`. It will prompt you for usernames.
- Use the usernames registered in your gradescope accounts. (local-part of your email)
 - *E.g. my email mortyz@ucsd.edu, so username is mortyz*
- If in a group, enter both usernames with a single space in between.
 - *E.g. npetrone example*
- Make sure you have correct cookie **before** doing the assignment (check on Gradescope)
 - Answers will change depending on the cookie.
 - If you messed up, regenerate:



```
cse127@cse127:~/targets$ ./build.sh
Using existing binary: /home/cse127/targets/bin/target0
Using existing binary: /home/cse127/targets/bin/target1
Using existing binary: /home/cse127/targets/bin/target2
Using existing binary: /home/cse127/targets/bin/target3
Using existing binary: /home/cse127/targets/bin/target4
Using existing binary: /home/cse127/targets/bin/target5
Using existing binary: /home/cse127/targets/bin/target6
Using existing binary: /home/cse127/targets/bin/target7
Enter usernames separated by spaces: npetrone example
Using cookie for [example,npetrone]
cse127@cse127:~/targets$ cat cookie
npetrone example
5604cse127@cse127:~/targets$
```

■ run `./build.sh clean` before you run `./build.sh` again.

Python 3 program

- You need to submit python programs that prints a line to be passed as input to the target.
- Work with byte literals:
 - `b"\xnn"`, where nn is a 2-digit hex value
 - E.g.: `sys.stdout.buffer.write(b"\x61\x62\x63")` Then the program receives `abc`
 - Can also do `b"\xnn" * n` to repeat n times.
- Pipe it to the target: `python3 sol.py | ./target`
- Use as command line argument: `./target "$(python3 sol.py)"`
- Endianness
 - Our machine uses little endian. 
 - to output 32-bit little-endian values:
 - `sys.stdout.buffer.write(0xDEADBEEF.to_bytes(4, "little"))`

Memory Layout Refresher

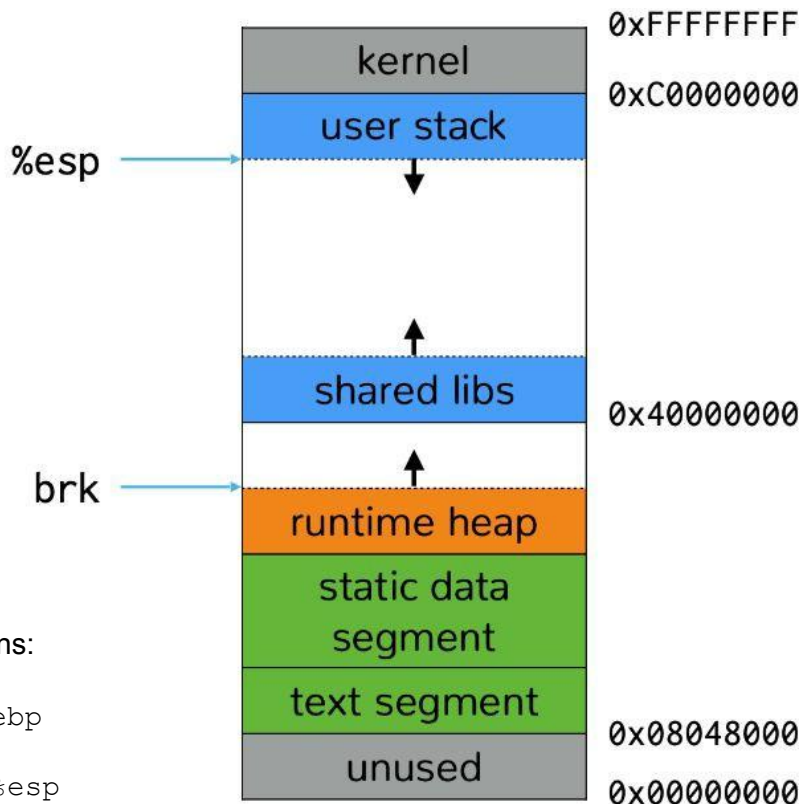
- **Stack** local variables, saved ebp
saved ret, args, saved registers

- **Heap** dynamically allocated memory,
ex: malloc, calloc, free

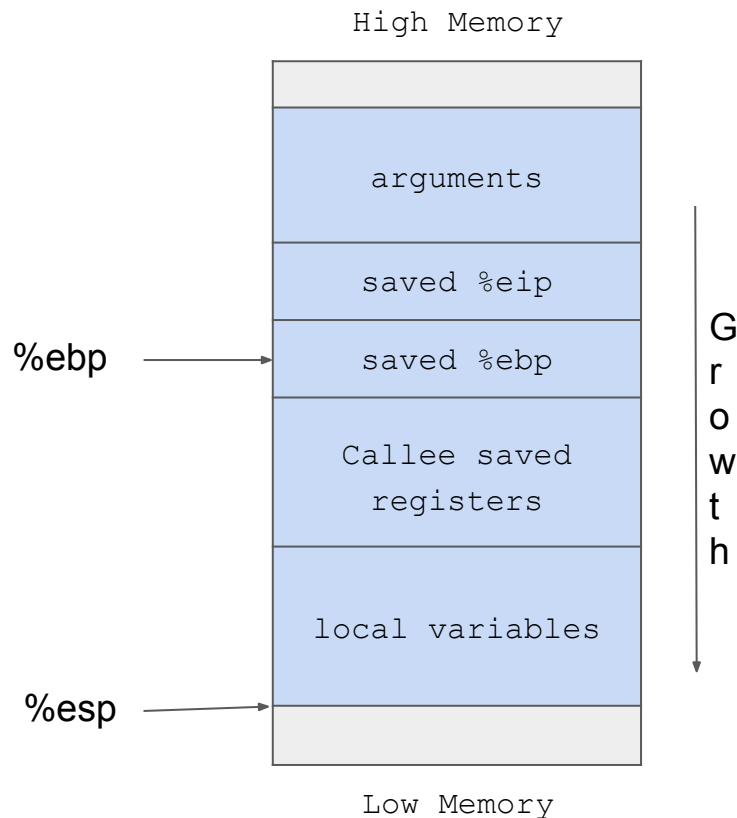
- **Data segment** static initialized data:
int i = 3;
char a[] = "Hello World";
static int b = 2023;

- **Text segment**
 - Binary instructions

```
program instructions:  
push    %ebp  
mov     %esp,%ebp  
push    %ebx  
sub     $0x14,%esp  
call    0x8049c40
```



Stack Layout: eip



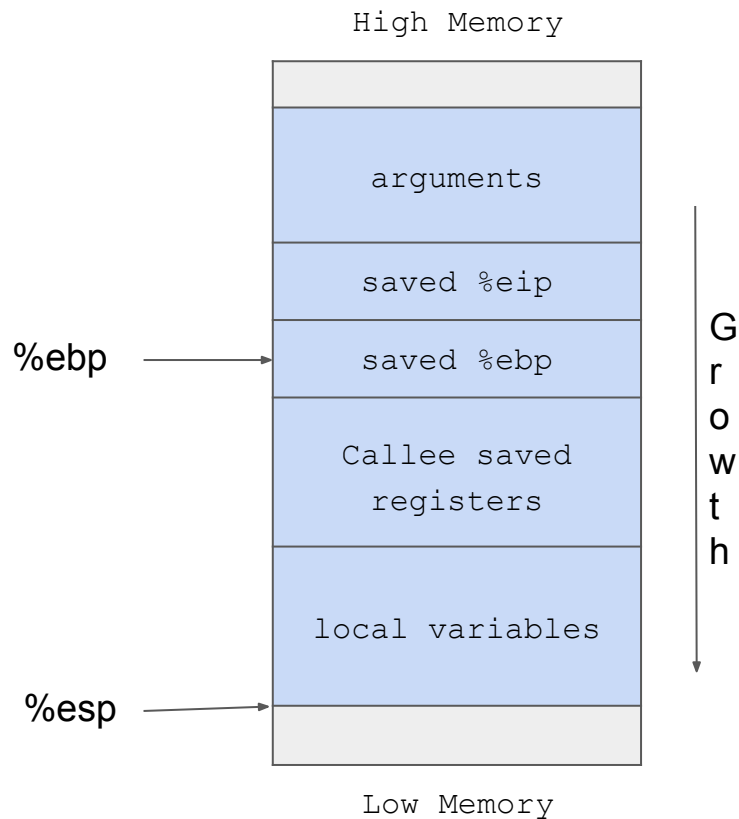
`%eip` is a register pointing to the next instruction

saved `%eip` (aka saved return) is stored on stack when a function call is made. It has the address of where to resume execution in the caller function.

When a function returns, the saved `%eip` value will be popped into the register `%eip`
→ resume execution at saved `%eip`

return address == `%ebp+4`

Stack Layout: esp



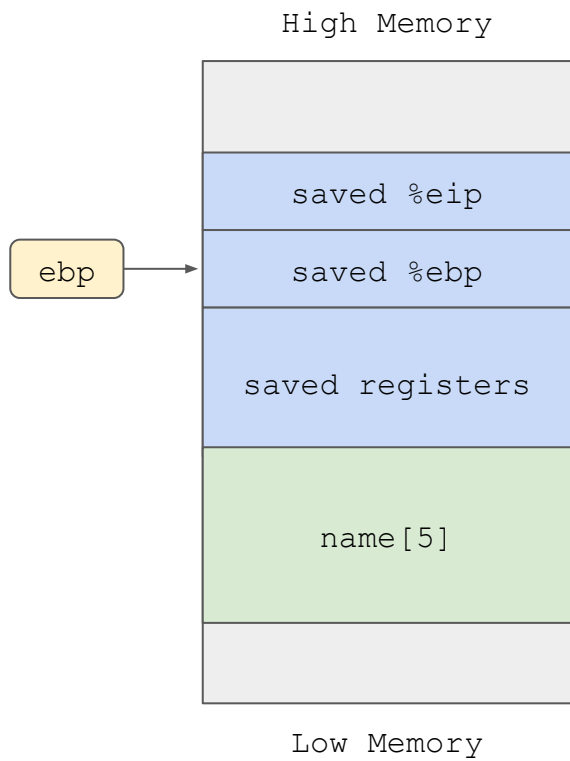
`%esp` aka stack pointer points to top of stack.

Stack grows down (from high to low addresses), so `%esp` points to the bottom of our picture.

examine gdb commands (`x /10x address`) examines memory low to high.

So to get the current stack dump, examine the `%esp` register!

Basic Buffer Overflow



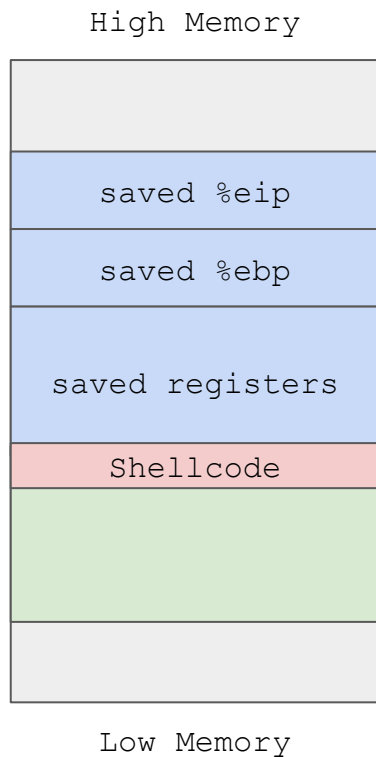
```
int main(int argc, char *argv[])
{
    char name[5];

    gets(name);

    printf("Hi %s!", name);

    exit(0);
}
```

Executing Shell Code



- Where do I place the shellcode?
- How to ensure the shellcode is always executed?

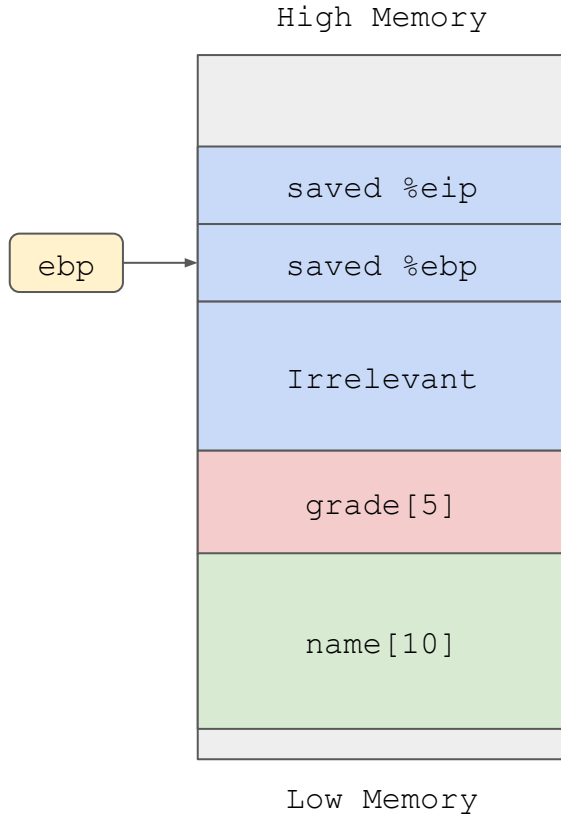
```
int main(int argc, char *argv[])
{
    char name[100];

    gets(name);

    printf("Hi %s!", name);

    exit(0);
}
```

Target 0: Overwriting a variable on the stack



- How does `gets` copy `stdin` into `name`?
- When does `gets` stop copying?
- Will `gets` check the boundary between `name` and `grade`?

```
int _main(int argc, char *argv[])
{
    char grade[5];
    char name[10];

    strcpy(grade, "nil");

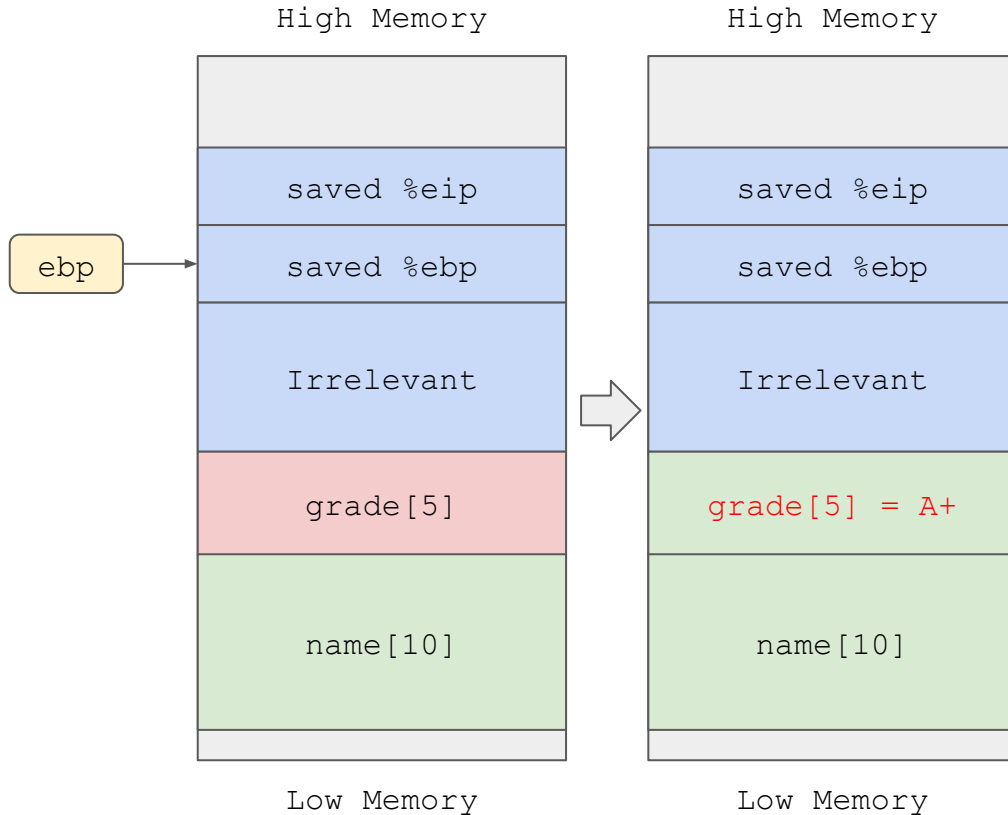
    gets(name);

    printf("Hi %s! Your grade is %s.\n", name,
grade);

    exit(0);
}
```

Be careful, when `name` is printed, no extra garbage content should be printed!
Think about how `printf` knows to stop printing.

Target 0: Overwriting a variable on the stack



```
int _main(int argc, char *argv[])
{
    char grade[5];
    char name[10];

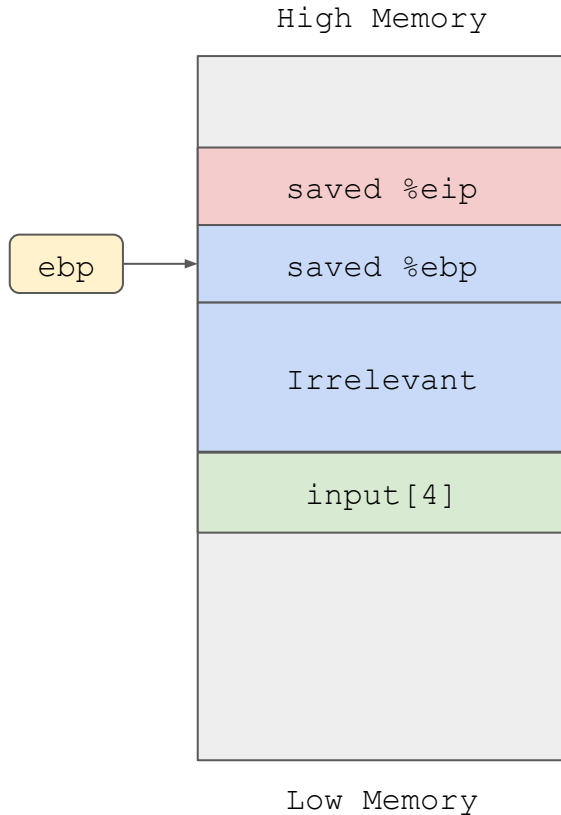
    strcpy(grade, "nil");

    gets(name);

    printf("Hi %s! Your grade is %s.\n", name,
grade);

    exit(0);
}
```

Target 1: Overwriting the return address



- What is the distance between `input` and `saved %eip`?
- How do you fill the gap in between?
- Where do you want to redirect control to?

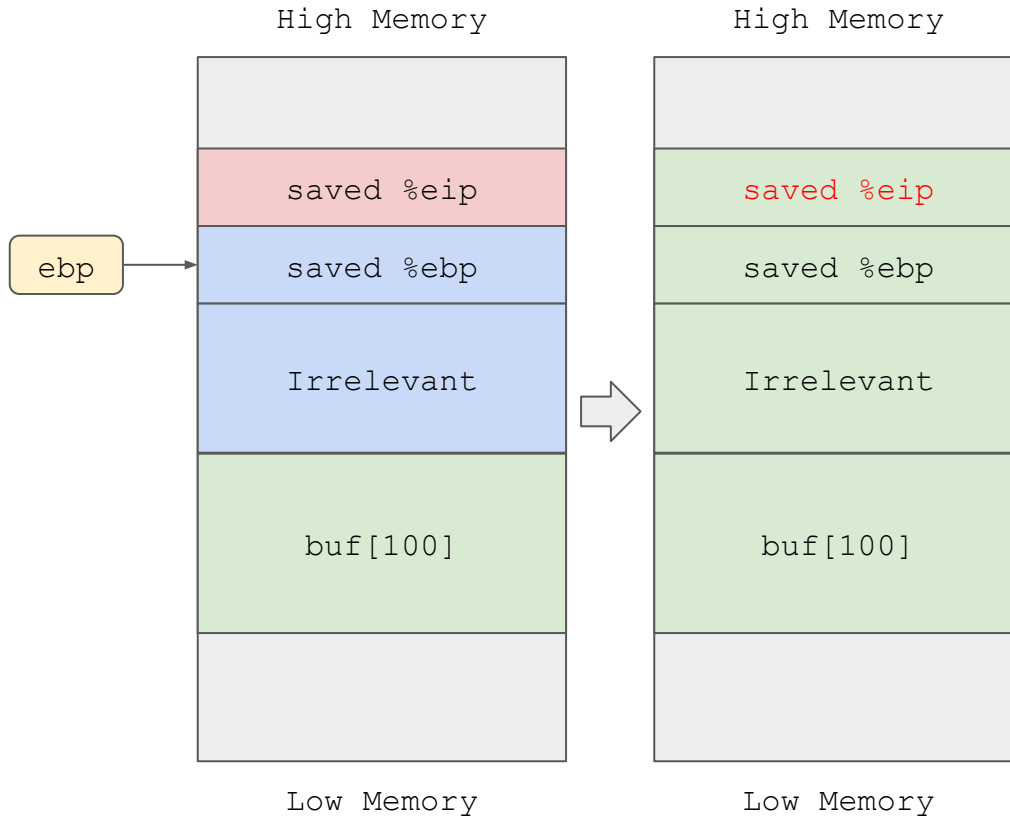
```
void print_bad_grade(void)
{
    puts("Your grade is nil.");
    exit(0);
}

void print_good_grade(void)
{
    puts("Your grade is perfect.");
    exit(1);
}

void vulnerable()
{
    char input[4];
    gets(input);
}

int _main()
{
    vulnerable();
    print_bad_grade();
    return 0;
}
```

Target 2: Redirecting control to shellcode

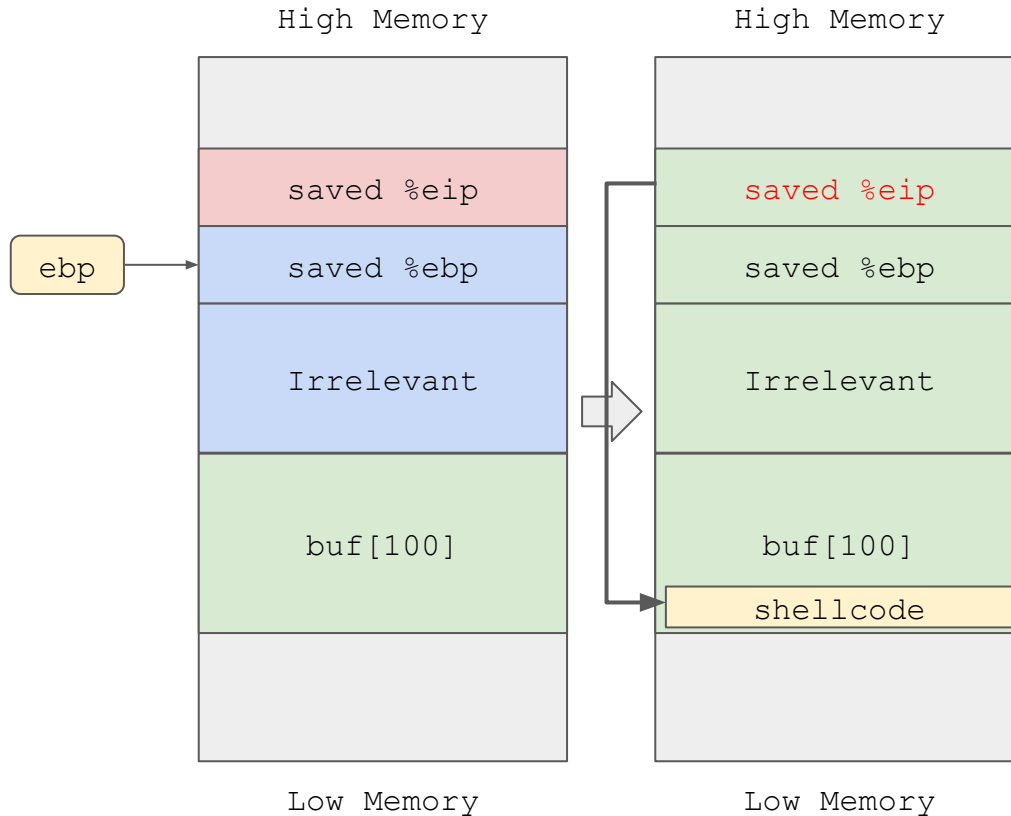


```
void vulnerable(char *arg)
{
    char buf[100];
    strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line
argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}
```

Where do you want to transfer control to?

Target 2: Redirecting control to shellcode

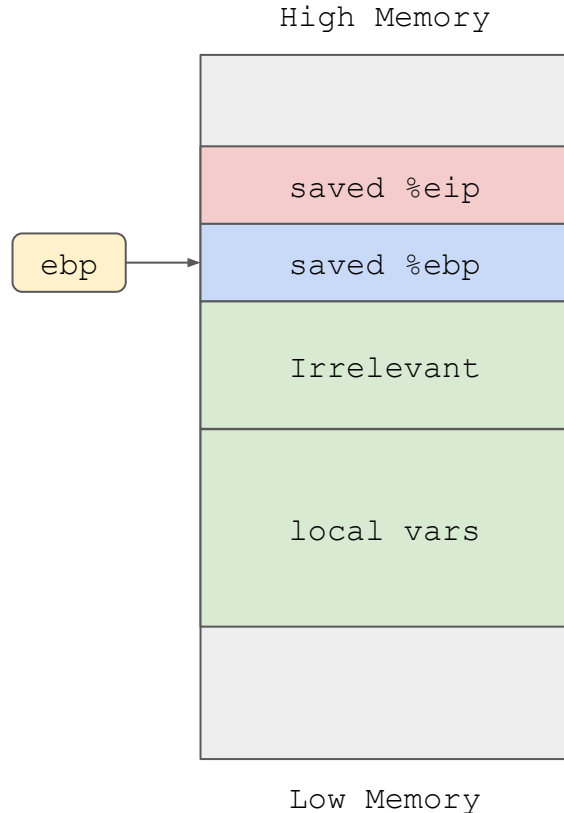


```
void vulnerable(char *arg)
{
    char buf[100];
    strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line
argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}
```

**Where do you want to
transfer control to?
What's the address?**

Target 3 (Hint): Overwriting the return address indirectly



```
void vulnerable(char *arg)
{
    int *p;
    int a;
    char buf[2048];

    strncpy(buf, arg, sizeof(buf) + 8);

    *p = a;
}
```

Oh no! **strncpy** only copies `sizeof(buf) + 8` bytes!

Well...

- `*p = a` is basically dereferencing 'a'
- Think about how dereferencing can be taken advantage of

Target 4 (Hint): Beyond strings

- fread reads bytes until EOF
→ count doesn't affect the number of bytes read
- count: 32-bit unsigned integer
- What happens when I overflow an unsigned int?

```
int main() {  
    unsigned int a = pow(2, 30) + 127;  
    unsigned int b = pow(2, 31) + 100;  
  
    printf("%u\n", a * 4);  
    printf("%u\n", b * 2);  
}
```

```
void read_elements(FILE *f, int *buf, unsigned int count)  
{  
    unsigned int i;  
    for (i=0; i < count; i++) {  
        if (fread(&buf[i], sizeof(unsigned int), 1, f) < 1) {  
            break;  
        }  
    }  
}  
  
void read_file(char *name)  
{  
    FILE *f = fopen(name, "rb");  
    ....  
  
    unsigned int count;  
    fread(&count, sizeof(unsigned int), 1, f);  
  
    unsigned int *buf = alloca(count * sizeof(unsigned int));  
    ....  
  
    read_elements(f, buf, count);  
}
```