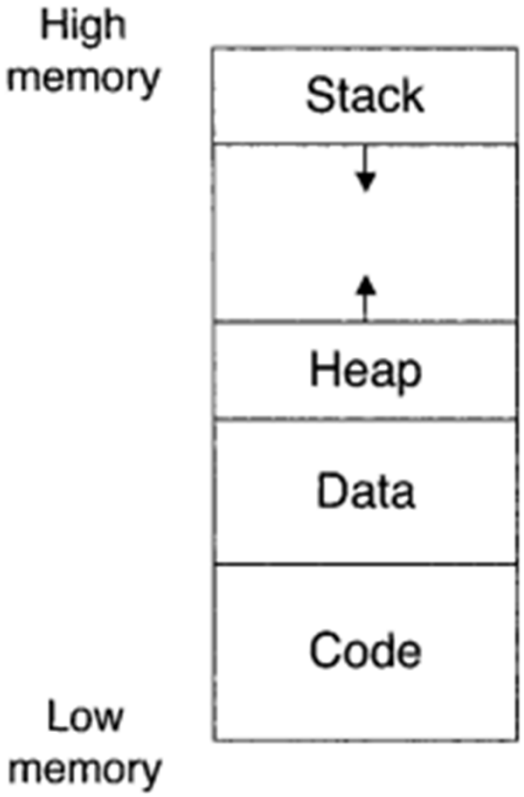
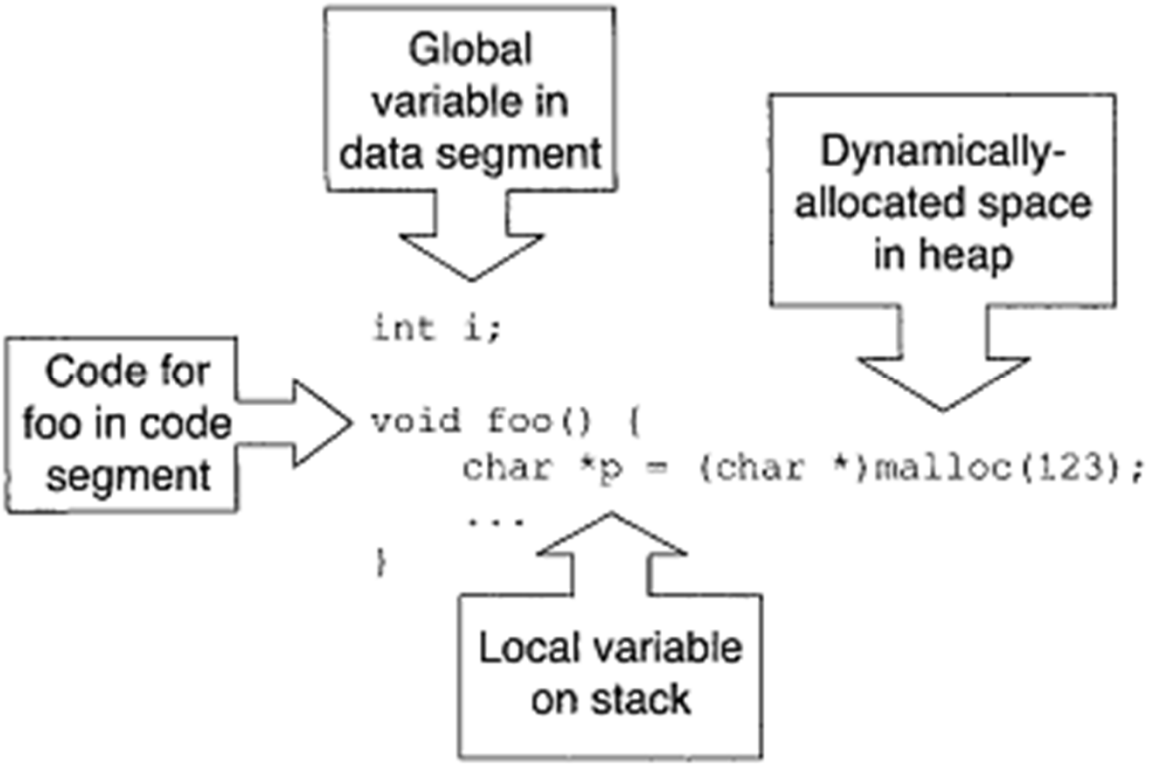
OS Vulnerabilities

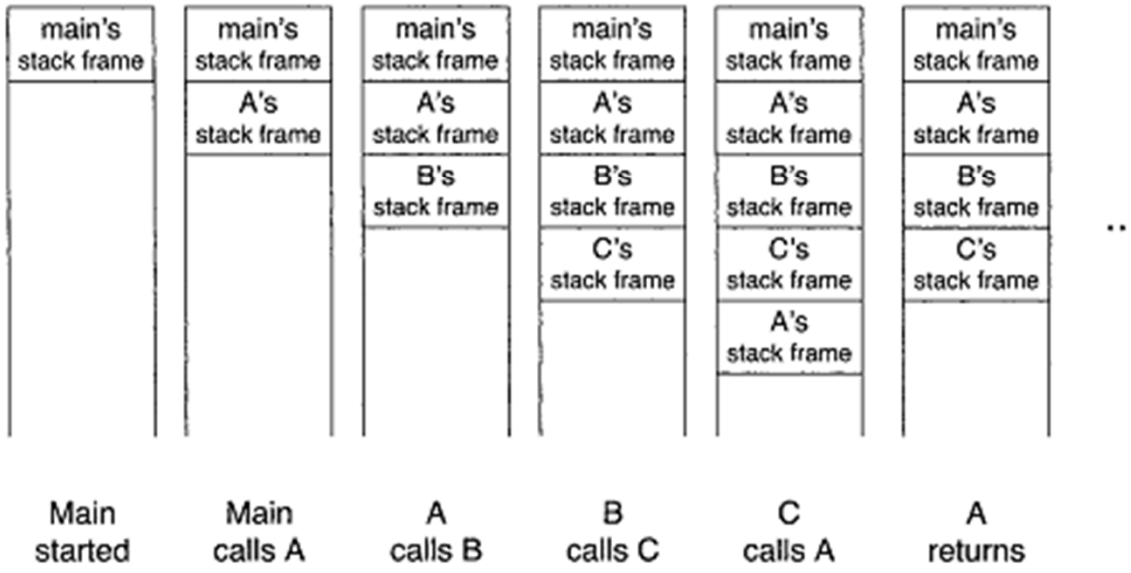
Background

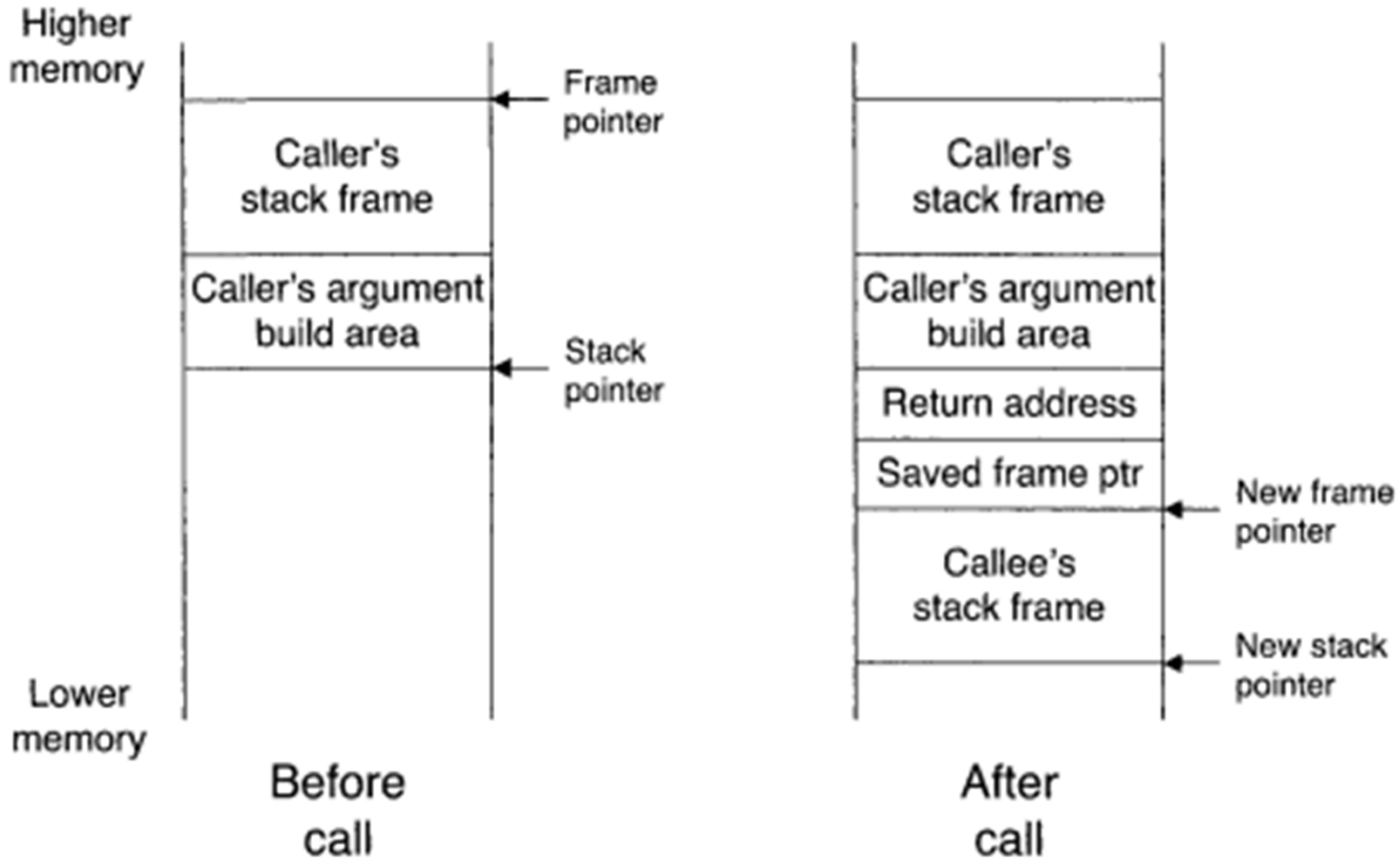
- A process' address space is divided into four "segments"




Global variable in data segment
Dynamically-allocated space in heap
Code for foo in code segment
int i;
void foo() {
char *p = (char *)malloc(123);
...
}
Local variable on stack

```
def main(): ... A() ...
def A(): ... B() ...
def B(): ... C() ...
def C(): ... A() ...
```



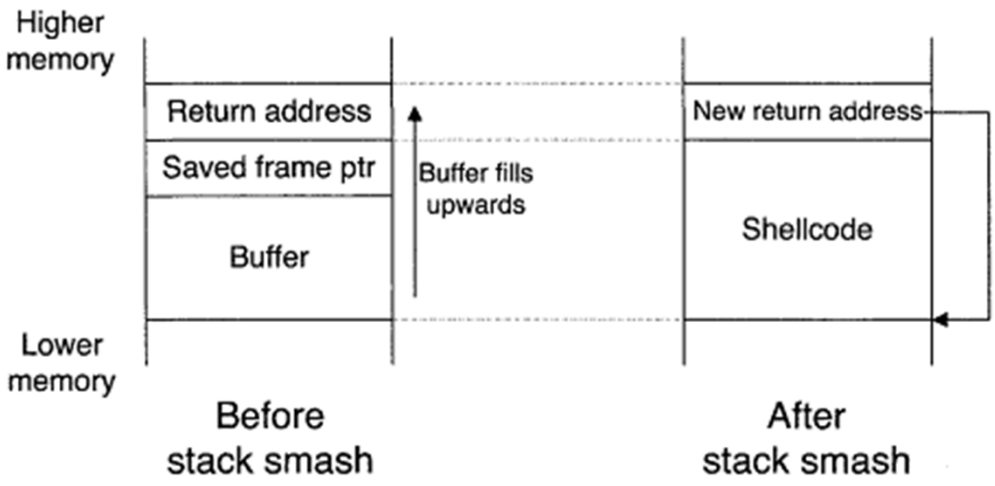

Higher memory
Frame pointer
Caller's stack frame
Caller's argument build area
Stack pointer
Lower memory
Before call
Caller's stack frame
Caller's argument build area
Return address
Saved frame ptr
New frame pointer
Callee's stack frame
New stack pointer
After call

Buffer Overflows

- A buffer overflow is a weakness in code where the bounds of an array (often a buffer) can be exceeded.
- An attacker will be able to write over other data in memory and cause the code to do something it wasn't supposed to
- Generally, this means that an attacker could coerce a program into executing arbitrary code of the attacker's choice.

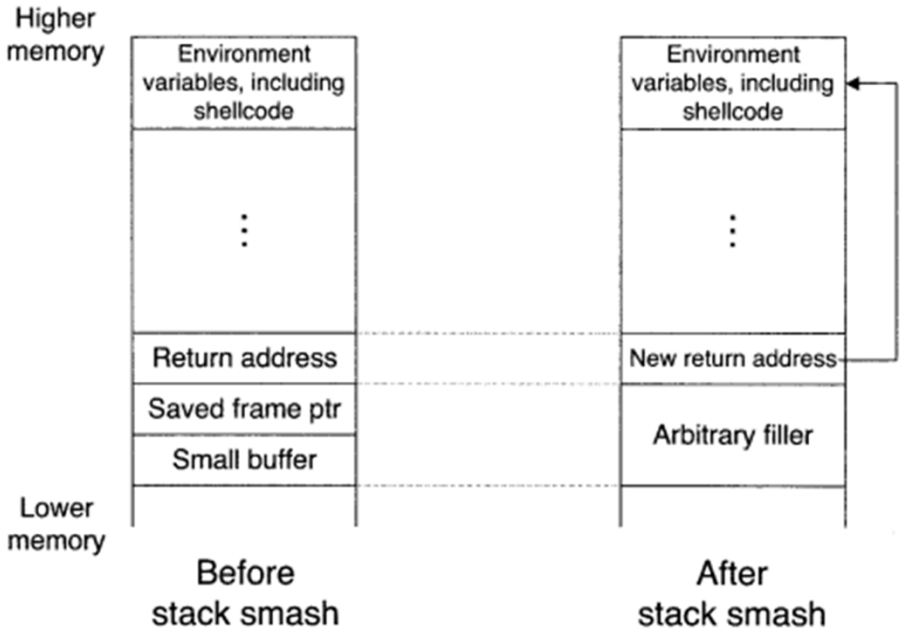
Buffer Overflow: Stack Smashing

- The buffer being overflowed is located in the stack. In other words, the buffer is a local variable in the code.
- As the stack-allocated buffer is filled from low to high memory, an attacker can continue writing, right over top of the return address on the stack



. . .

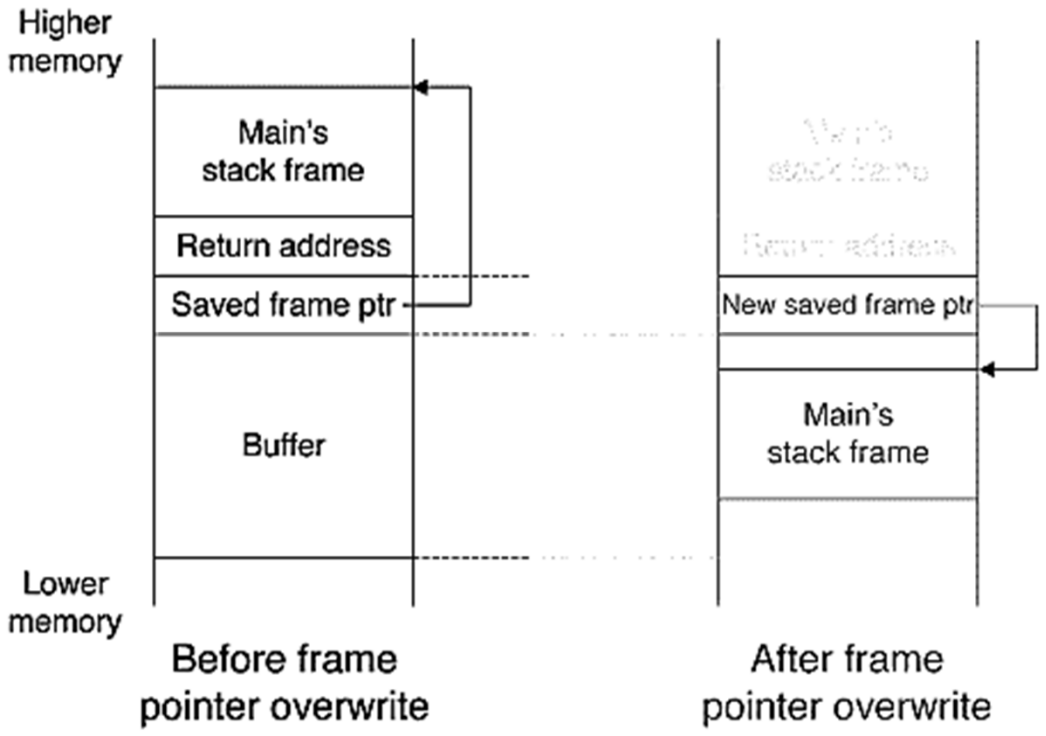
- If an attacker controls the exploited program's environment, they can put their shellcode into an environment variable.
- Instead of making the new return address point to the overwritten buffer, the attacker points the new return address to the environment variable's memory location



Buffer Overflow: Frame Pointer Overwriting

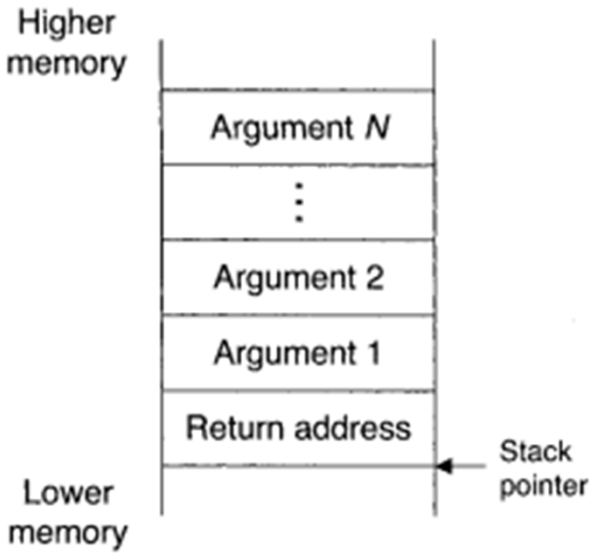
- The attack overwrites a byte of the saved frame pointer
- When the called subroutine returns, it restores the saved frame pointer from the stack; the caller's code will then use that frame pointer value.
- After a frame pointer attack, the caller will have a distorted view of where its stack frame is.

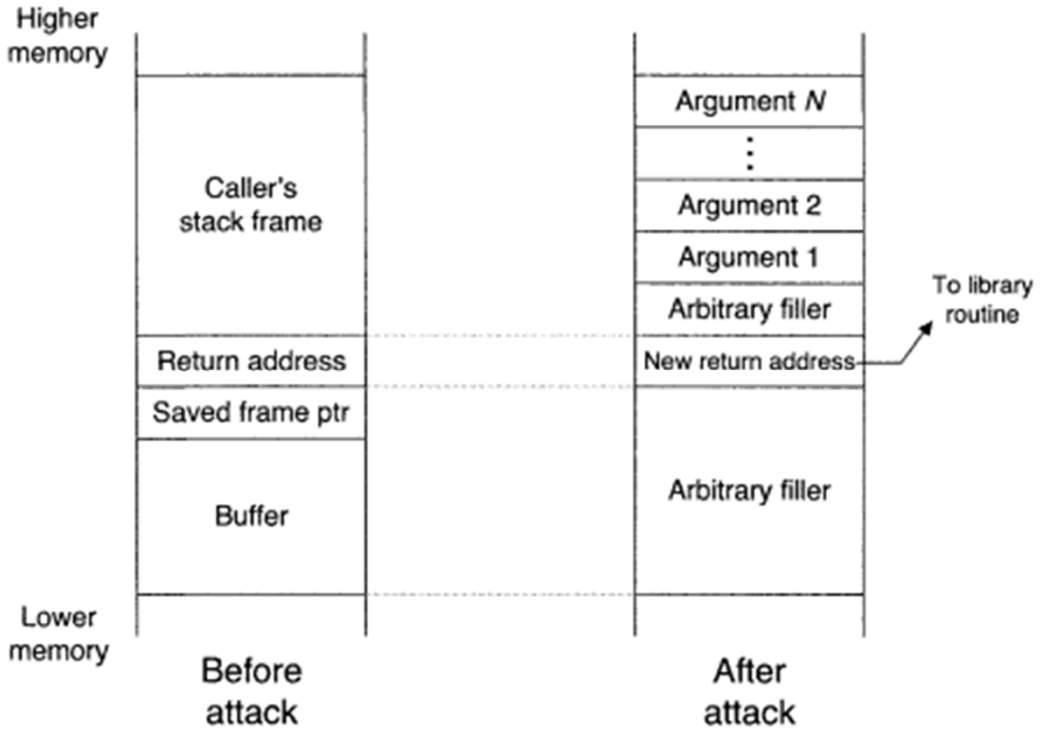
```
def main():
    fill_buffer()

def fill_buffer():
    character buffer[100]
    i = 0
    ch = input()
    while i <= 100 and ch ≠ NEWLINE:
        buffer_i = ch
        ch = input()
        i = i + 1
```



Buffer Overflow: Returns into Libraries

- If an attacker can't run arbitrary code, they can still run other code
- Shared library code
- An attacker can overwrite a return address on the stack to point to a shared library routine to execute
- For example, an attacker may call the system library routine, which runs an arbitrary command.
- Arguments may be passed to library routines by the attacker by writing beyond the return address in the stack


Higher memory
Argument N
Argument 2
Argument 1
Return address
Stack pointer
Lower memory


Higher memory
Caller's stack frame
Return address
Saved frame ptr
Buffer
Lower memory
Before attack
Argument N
Argument 2
Argument 1
Arbitrary filler
New return address
Arbitrary filler
To library routine
After attack

Buffer Overflow: Heap Overflows

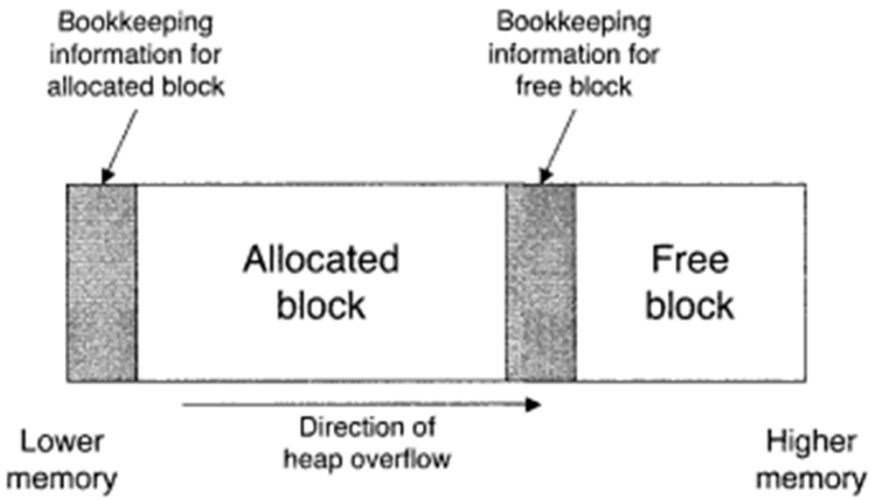
- A heap overflow is a buffer overflow, where the buffer is located in the heap or the data segment
- The idea is not to overwrite the return address or the saved frame pointer, but to overwrite other variables that are adjacent to the buffer.

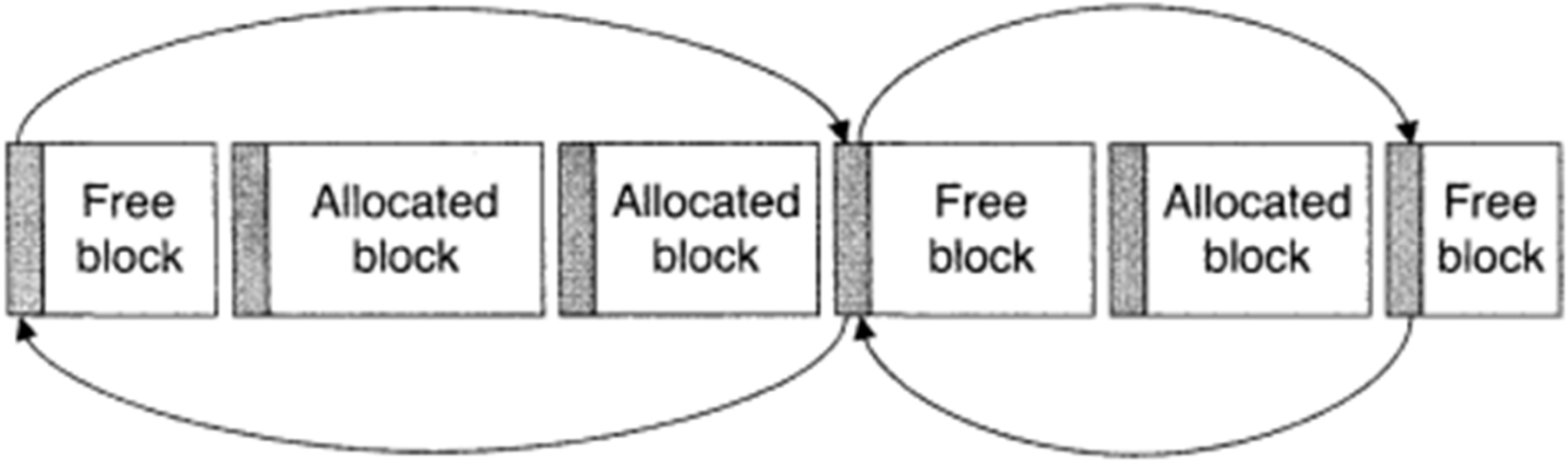
```
character buffer[123]
function pointer p
```

- Overflowing the buffer allows an attacker to change the value of the function pointer p, which is the address of a function to call
- If the program performs a function call using p later, then it jumps to the address the attacker specified; again, this allows an attacker to run arbitrary code.

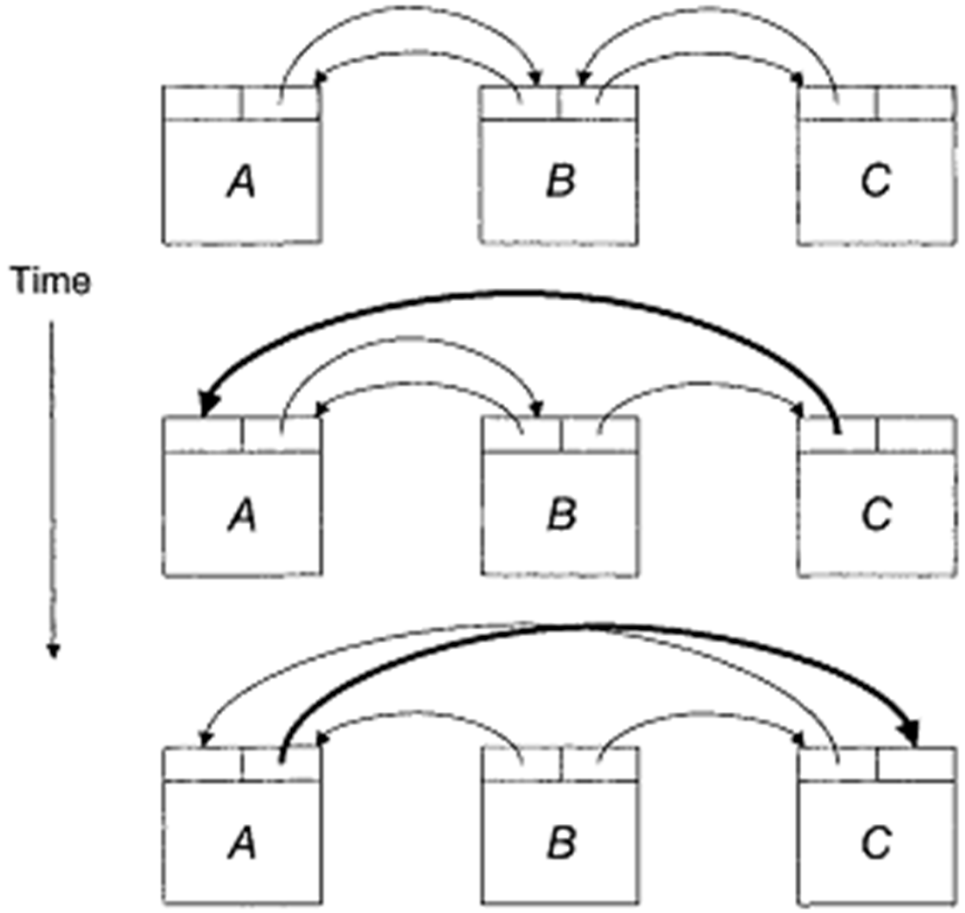
Buffer Overflow: Memory Allocator Attacks

- One way heap overflows can be used is to attack the dynamic memory allocator
- The allocator needs to maintain bookkeeping information for each block of memory that it oversees in the heap
- When a program requests an X-byte block of memory, the allocator reserves extra space:
  - Before the block, room for bookkeeping information.
  - After the block, space may be needed to round the block size up
- Exploiting a heap overflow in one block allows the bookkeeping information for the following block to be overwritten

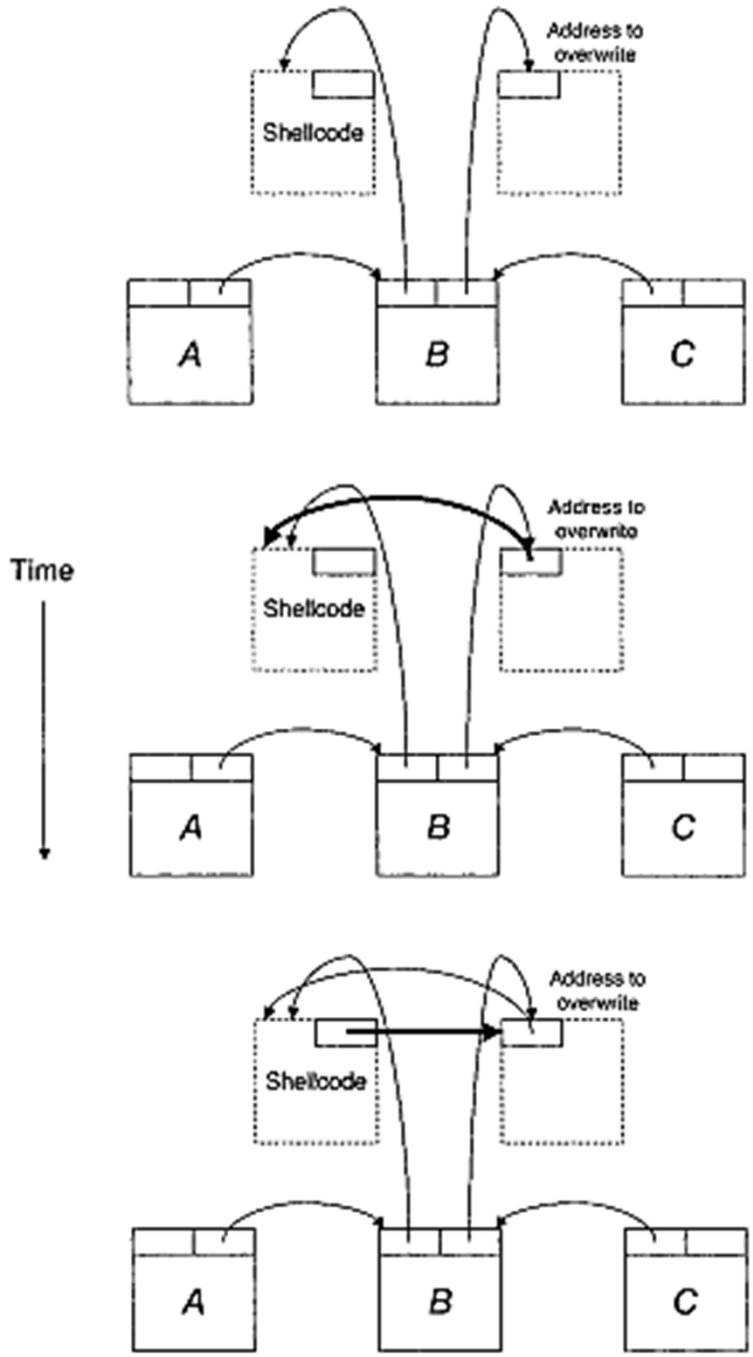

Bookkeeping information for allocated block
Bookkeeping information for free block
Allocated block
Free block
Direction of heap overflow
Lower memory
Higher memory


Free block
Allocated block
Allocated block
Free block
Allocated block
Free block

There are two steps to unlink a node B



- An attacker exploits a heap overflow in the allocated block immediately before B, and overwrites B's list pointers
- B's previous pointer is set to the address of the attacker's shellcode, and B's next pointer is assigned the address of a code pointer that already exists in the program. For example, this code pointer may be a return address on the stack, or a function pointer in the data segment.
- The attacker then waits for the program to free the memory block it overflowed


Address to overwrite
Shellcode
A
B
C
Time
Address to overwrite
Shellcode
A
B
C
Address to overwrite
Shellcode
A
B
C

Integer Overflows

- In most programming languages, numbers do not have infinite precision.
- For instance, the range of integers may be limited to what can be encoded in 16 bits
- Integer overflows, where a value "wraps around." For example, 30000 + 30000 = -5536.
- Sign errors. Mixing signed and unsigned numbers can lead to unexpected results. The unsigned value 65432 is -104 when stored in a signed variable, for instance
- Truncation errors, when a higher-precision value is stored in a variable with lower precision. For example, the 32-bit value 8675309 becomes 24557 in 16 bits

- These effects can be exploited by an attacker - they are collectively called integer overflow attacks
- Usually the attack isn't direct, but uses an integer overflow to cause other types of weaknesses, like buffer overflows

```
n    = input_number()
size = input_number()
totalsize = n * size
buffer = allocate_memory(totalsize)
i = 0
buffer_pointer = buffer
while i < n:
buffer_pointer 0...size-1 = input_N_bytes(size)
buffer_pointer = buffer_pointer + size
i - i + 1
```

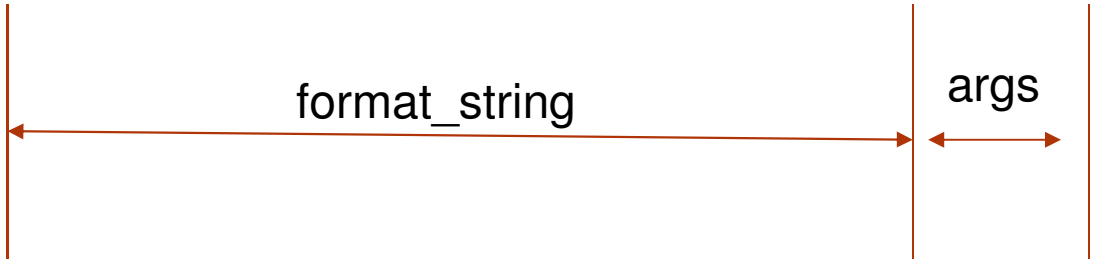
....

- If an attacker's input results in n being 1234 and size being 56, their product is 69104, which doesn't fit in 16 bits - totalsize is set to 3568 instead.
- As a result of the integer overflow, only 3568 bytes of dynamic memory are allocated, yet the attacker can feed in 69104 bytes of input in the loop that follows, giving a heap overflow

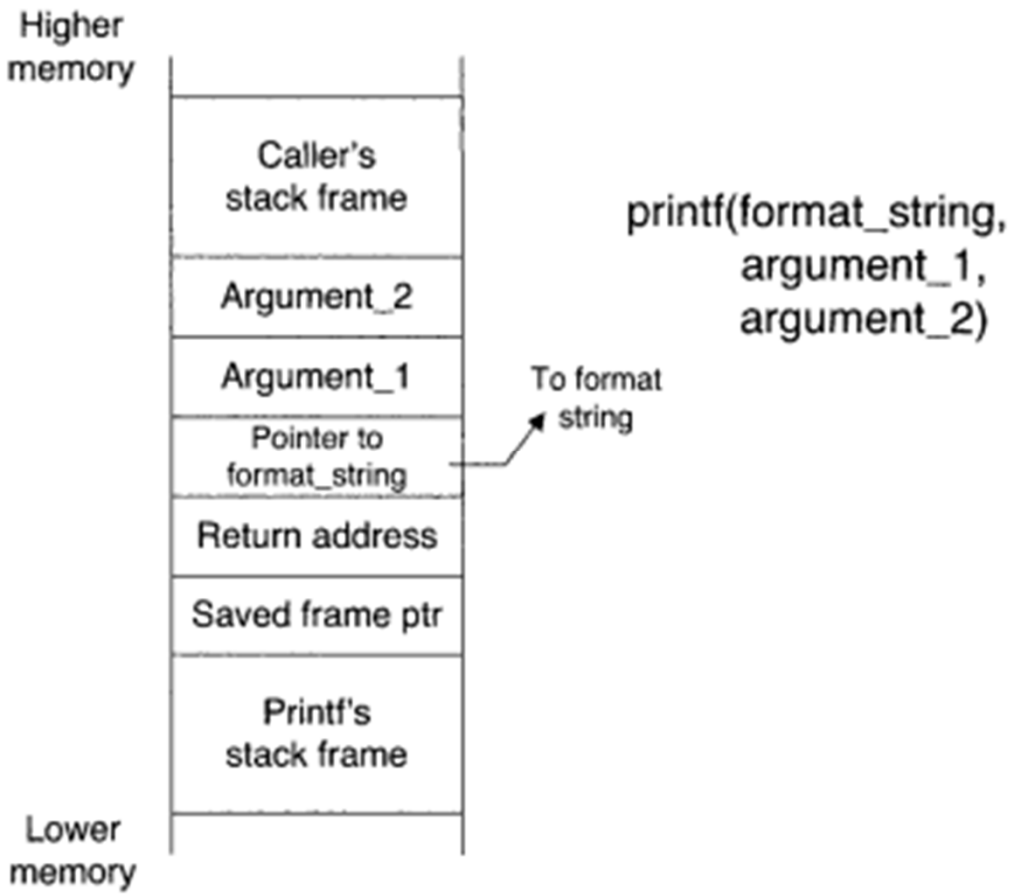
Format String Vulnerabilities

- The format string tells the function how to compose the arguments into an output string
- Depending on the format function, the output string may be written to a file, the standard output location, or a buffer in memory

```
char *s = "is page";
int n = 125;
printf ( "Hello, world!");
printf ( "This %s %d.", s, n ) ;
```

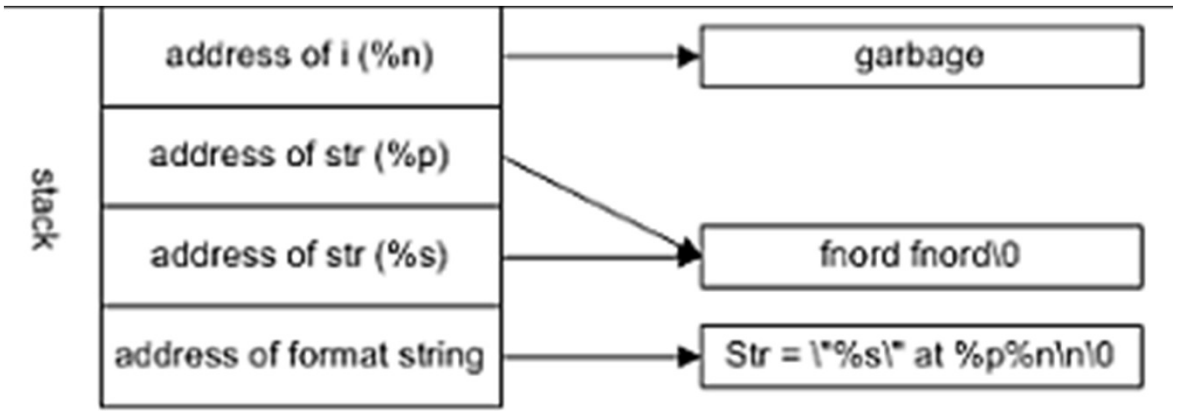


Printf (“so nguyen = %d, chuoi = %s\n”, i, str);

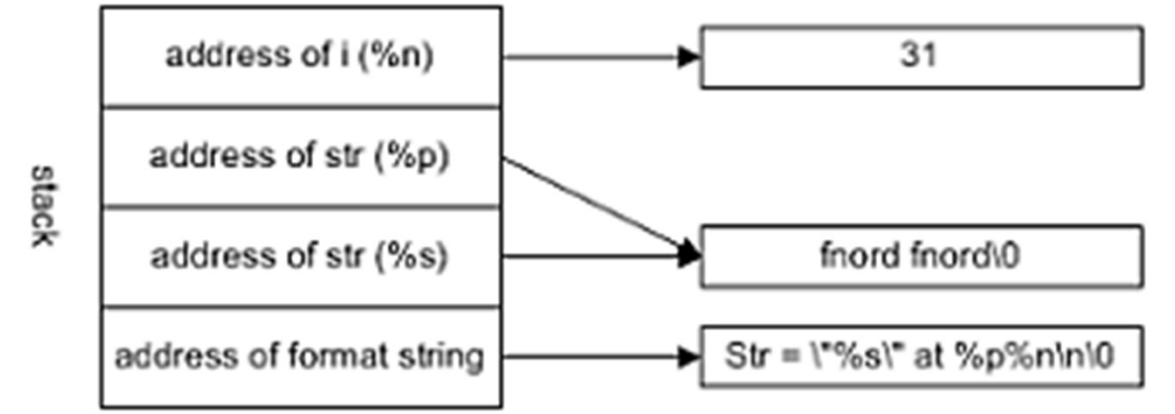

Higher memory
Caller's stack frame
Argument_2
Argument_1
Pointer to format_string
Return address
Saved frame ptr
Printf's stack frame
Lower memory
To format string
printf(format_string,
argument_1,
argument_2)

Example %n

```
1.  /*format3.c – various format tokens*/
2.  #include "stdio.h"
3.  #include "stdarg.h"
4.  void main (void)
5.  {
6.  char * str;
7.  int  i;
8.  str = "fnord fnord";
9.  printf("Str = \"%s\" at %p%n\n ", str, str, &i);
10. printf("The number of bytes in previous line is %d", i);
11.   }
```



before printf()



after printf()

Source of Format String Vulnerability

- Format functions exhibit a touching faith in the correctness of the format string.
- A format function has no way of knowing how many arguments were really passed by its caller
- An attacker is able to supply any part of a format string
- Example: `printf (error);` with error is set to %d%d%d%d, in the above example would be enough to print the stack contents. (This is one possible way that addresses can be discovered for a later stack smashing attack)

Example Exploiting the F_S Vulnerability

- Even more is possible if the attacker can control a format string located in the stack

```
void   print _error(char   *s)
{
char buffer[123];
snprintf(buffer, sizeof(buffer), "Error: %s",s) ;
printf (buffer);
}
```

Next item in stack is pointer to a integer

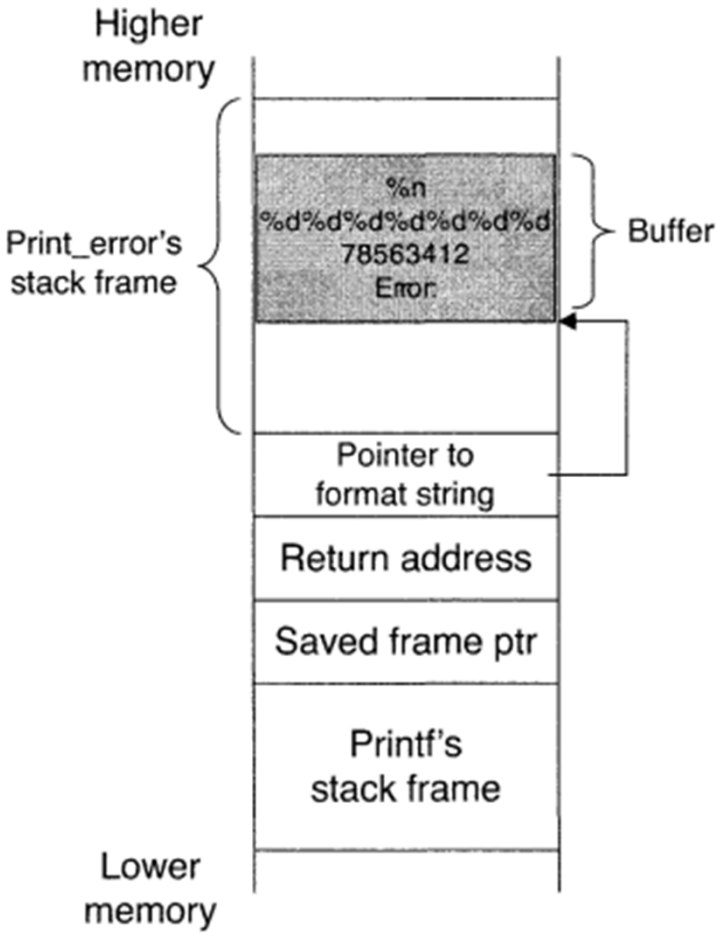
- The attacker has supplied in part:

"Error: \x78\x56\x34\x12 %d%d%d%d%d%d%d %n"

Address 12345678

Ghi số byte trong chuỗi định dạng vào biến integer

- The attacker's format string causes printf to walk up the stack, printing integers, until the next unread argument is the address the attacker encoded in the format string.
- The %n takes the attacker's address and writes a number at that address.
- Through this mechanism, the attacker has a way to have a value written to an arbitrary memory location



RACE CONDITIONS

- Race conditions result when an electronic device or process attempts to perform two or more operations at the same time, producing an illegal operation
- Race conditions are a type of vulnerability that an attacker can use to influence shared data, causing a program to use arbitrary data and allowing attackers to bypass access restrictions
- Such conditions may cause data corruption, privilege escalation, or code execution when in the appropriate context. Race conditions are also known as time-of-check and time-of-use (TOC/TOU) vulnerabilities because they involve changing a shared value immediately after the check phase.

Race Condition Vulnerability

- Example: a snippet of program in Linux

```
1: if (!access("/tmp/X", W_OK)) {
2:    /* the real user ID has access right */
3:    f = open("/tmp/X", O_WRITE);
4:    write_to_file(f);
5: }
6: else {
7:    /* the real user ID does not have access right */
8:    fprintf(stderr, "Permission denied\n");
9: }
```

Can we use this program to overwrite another file?

- Assume that the above program somehow runs very very slowly. It takes one minute to run each line of the statement in this program
- We cannot modify the program, but you can take advantage of that one minute between every two statements of the program.
- The /tmp directory has permission rwxrwxrwx, which allows any user to create files/links under this directory.

Attack Ideas

- Focus on the time window between Line 1 and Line 3. Within this time window, we can delete /tmp/X and create a symbolic link used the same name, and let it point to /etc/passwd.
- What will happen between Line 1 and Line 3?
  - The program will use open() to open /etc/passwd by following the symbolic link.
  - The open() system call only checks whether the effective user (or group) ID can access the file. Since this is a Set-UID root program, the effective user ID is root, which can of course read and write /etc/passwd.
  - Therefore, Line 4 will actually write to the file /etc/passwd.
  - If the contents written to this file is also controlled by the user, the attacker can basically modify the password file, and eventually gain the root privilege.
  - If the contents are not controlled by the user, the attacker can still corrupt the password file, and thus prevent other users from logging into the system.

The program runs very fast, have not that one-minute time window. What can we do?

- There is a short time window between access() and open().
- The window between the checking and using: Time-of-Check, Time-of-Use (TOCTOU).
- CPU might conduct context switch after access(), and run another process.
- If the attack process gets the chance to execute the above attacking steps during this context switch window, the attack may succeed
- If running once does not work, we can run the attack and the target program for many times.

Improving Success Rate

- The most critical step of a race-condition attack must occur within TOCTOU window.
- Since we cannot modify the vulnerable program, the only thing that we can do is to run our attacking program in parallel with the target program, hoping that the change of the link does occur within that critical window.
- The success of attack is probabilistic.The probability of successful attack might be quite low if the window is small.
- How do we increase the probability?
  - Slow down the computer by running many CPU-intensive programs.
  - Create many attacking processes.

The End