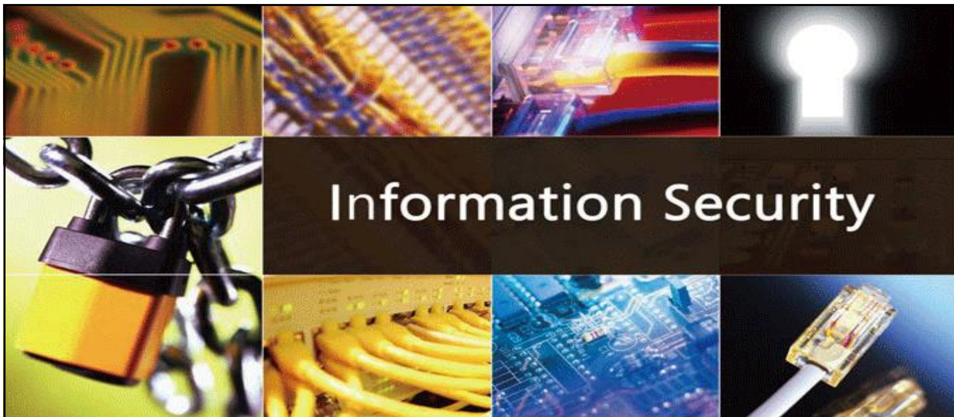


Software security

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

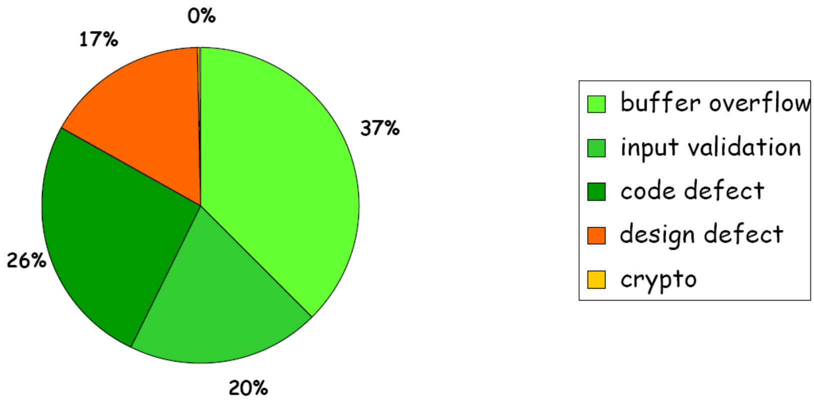
Software Security issues

- Insecure interaction between components
  - Ex, unvalidated 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
    - 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

Typical Software Security Vulnerabilities



Security bugs found in Microsoft bug fix month (2002)

11/09/2017 5

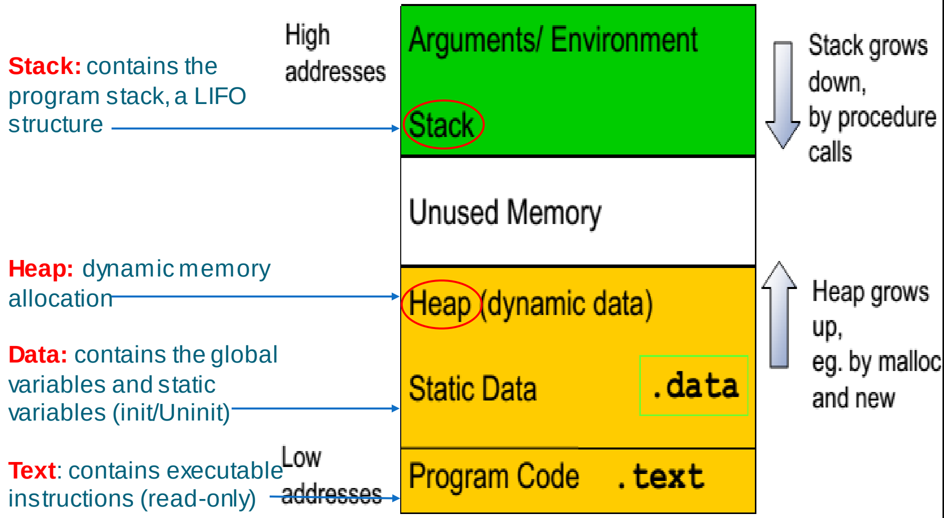
Software Vulnerabilities - Buffer overflows

- **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

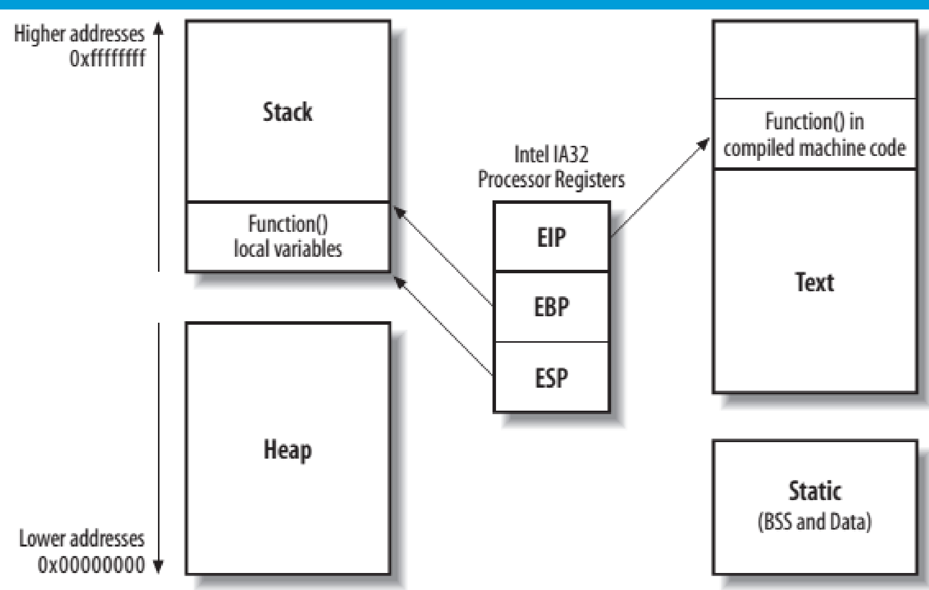
Process memory layout



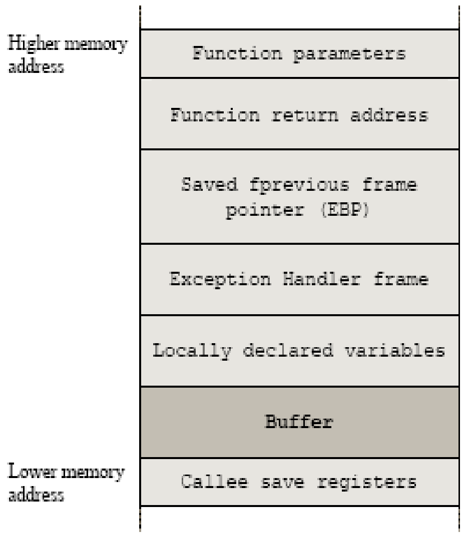
Stack Layout: Terminologies

- **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.
- **Stack Pointer (esp) Register:** Stores the memory address to which the stack pointer (the current top of the stack: pointing towards the low memory end) is pointing to.
  • 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 (the reference pointer for a stack frame with respect to which the different memory locations can be accessed using relative addressing) is pointing to.
  • 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.

The processor registers and memory



Stack layout




```
void func(char *a1, int a2, int a3)
{
char b1[12];
int b2;
int b3;
…
}
void main()
{
        func("hello", 5, 6);
}
```

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

11/09/2017 13

Stack smash

- places data into a fixed sized buffer

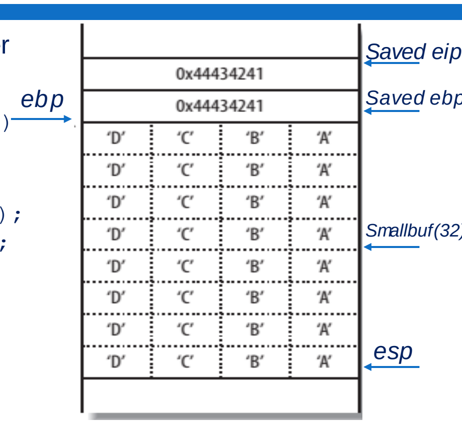
Code:

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

Run:

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

Segmentation fault (core dumped)



14

- 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

13/09/2017

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)
```


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)

Stack smash - challenges

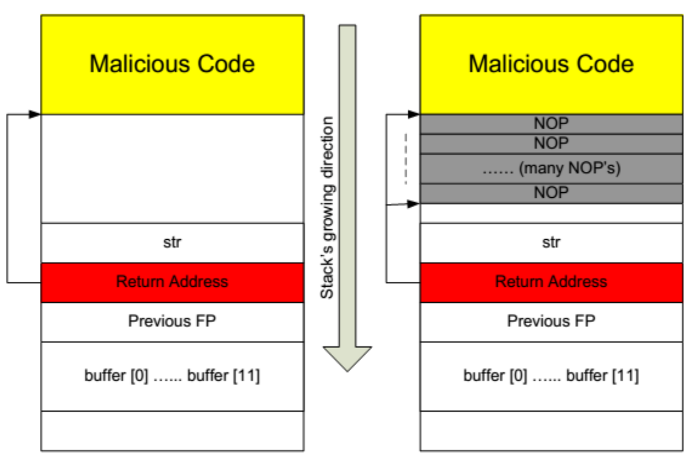
Attacker need to overcome to make the successful attack

- 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]
- A shellcode is the code to launch a shell

13/09/2017

Stack smash - challenges

Finding the starting point of the malicious code



13/09/201

(a) Jump to the malicious code

(b) Improve the chance

18

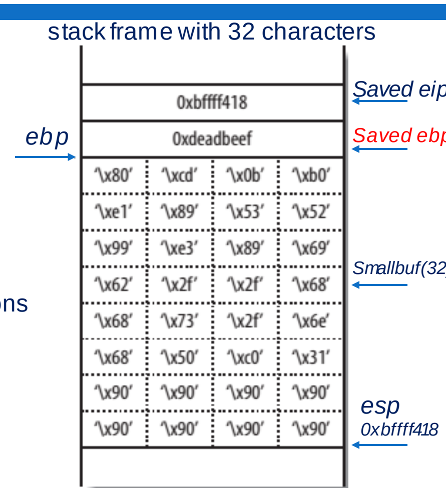
Shellcode

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

```
"\x31\xc0\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
```



13/09/2017 19

Stack off-by-one

- 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:
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;
}
```

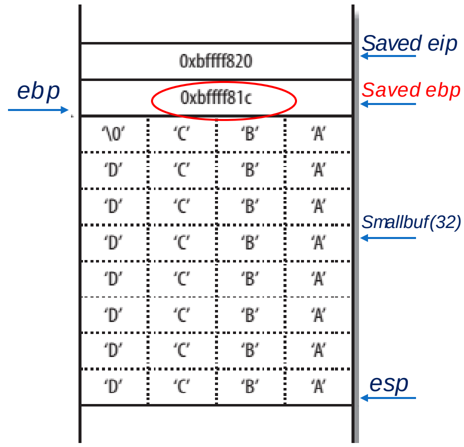
Input:

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

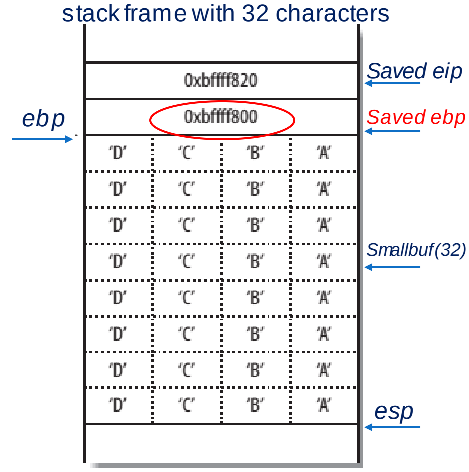
11/09/2017 20

Stack off-by-one

stack frame with 31 characters



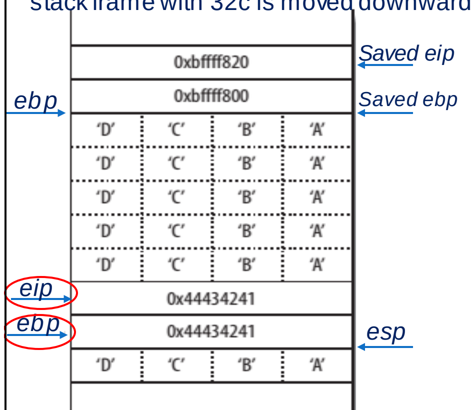
stack frame with 32 characters



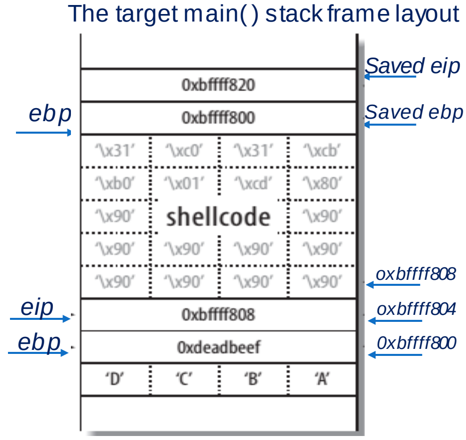
the least significant byte of the saved ebp is overwritten, changing it from 0xbfff81c to 0xbffff800

Stack off-by-one - challenge

stack frame with 32c is moved downward



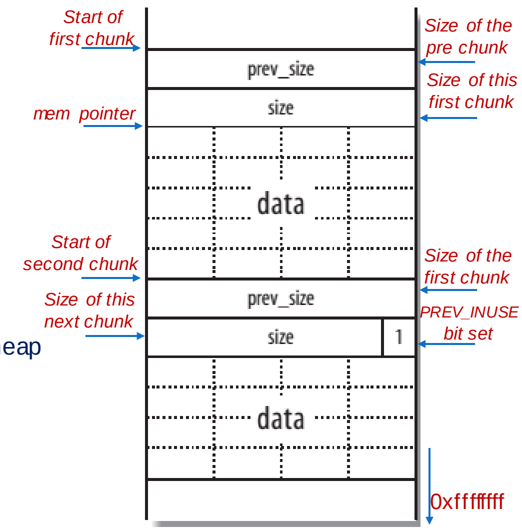
The target main() stack frame layout



First, the saved ebp is popped and changed to 0xbfff800,
The ebp has been slid down to a lower address.
Popping the new saved eip (ebp+4, 0x44434241)

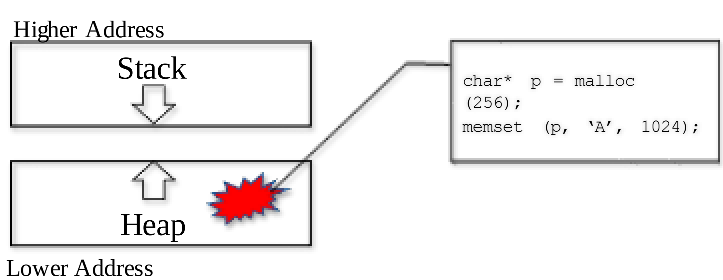
Heap overflows

- heap overflows
  - dynamically allocate buffers of varying sizes
  - are reliant on the way certain operating systems 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).



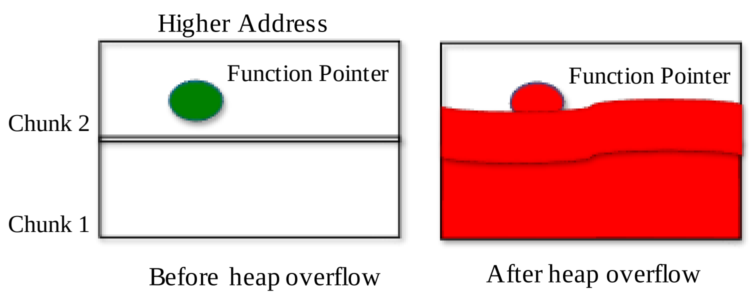
Heap Overflow

- 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

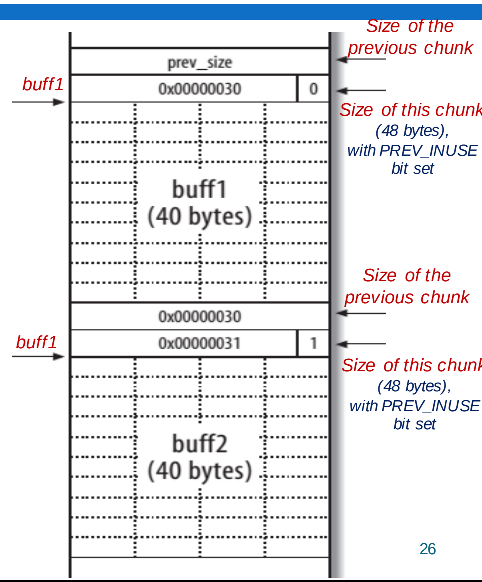


Heap Overflow – Example

```
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.



Attacks: code injection & code reuse

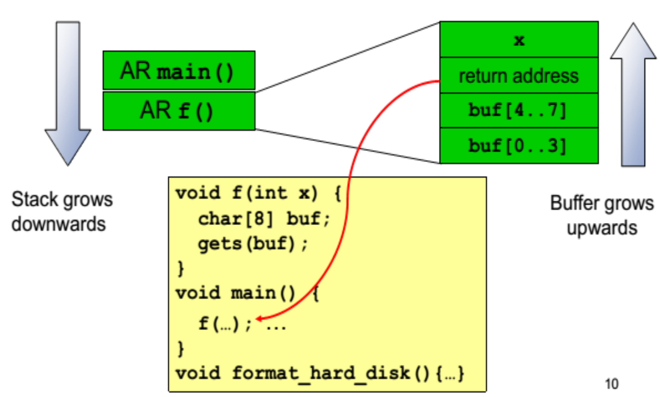
- Code *injection* attack
  attacker inserts his own shell code in a buffer and corrupts the return addresss to point to this code
  Ex, **exec (/bin/sh)**
  This is the "classic" buffer overflow attack
  [Smashing the stack for fun and profit, Aleph One, 1996]

- Code *reuse* attack
  attacker corrupts the return address to point to existing code,
  Ex , **format_hard_disk**

11/09/2017 27

format_hard_disk

- The stack consists of Activation Records:



11/09/2017 28

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.

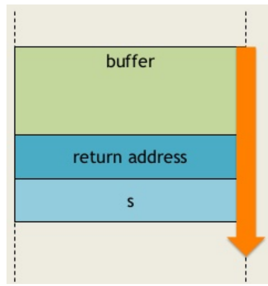
Return-to-libc

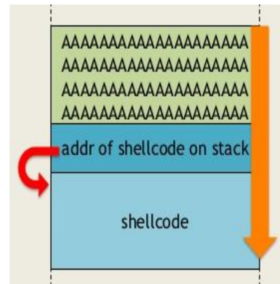
- Create 1 fake frame on the stack
- After an overflowed function returns…
- …set the eip return address to the new function
- Append the fake frame
- New function executes
  - Parameters consumed from the fake frame
- System("/bin/sh")

Return-to-libc

- Function is vulnerable to a stack overflow

```
Code:
void func1(char *s)
{
   char buffer[80];
   strcpy(buffer, s);
   printf("%s\n", buffer);
   return 0;
}
```





13/09/2017 31

Defense Against Buffer Overflow Attacks

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

Defense Against Buffer Overflow Attacks

Programming language choice is crucial.

The language...

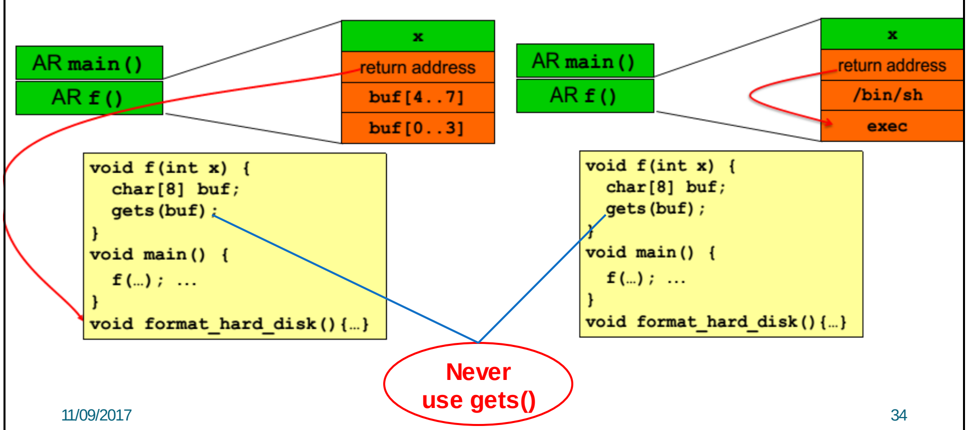
- Should be **strongly typed**
- Should do **automatic bounds checks**
- Should do **automatic memory management**

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

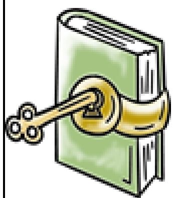
Ex

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

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



Defense Against Buffer Overflow Attacks



Why are some languages safe?

- Buffer overflow becomes impossible due to runtime system checks

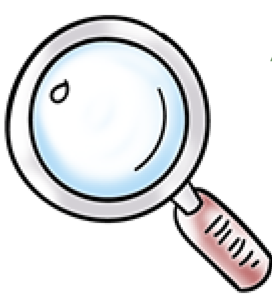
The drawback of secure languages

- 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…

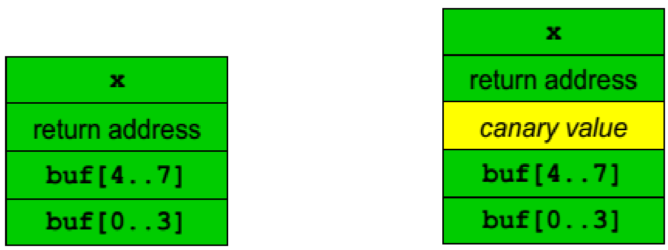
- Can **flag** potentially unsafe functions/constructs
- Can **help mitigate security lapses**, but it is really hard to eliminate all buffer overflows.

Examples of analysis tools can be found at:
https://www.owasp.org/index.php/Source_Code_Analysis_Tools

Thwarting Buffer Overflow Attacks: Stack Canaries

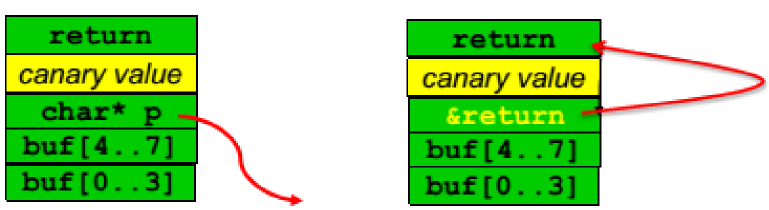
Stack Canaries:

- 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

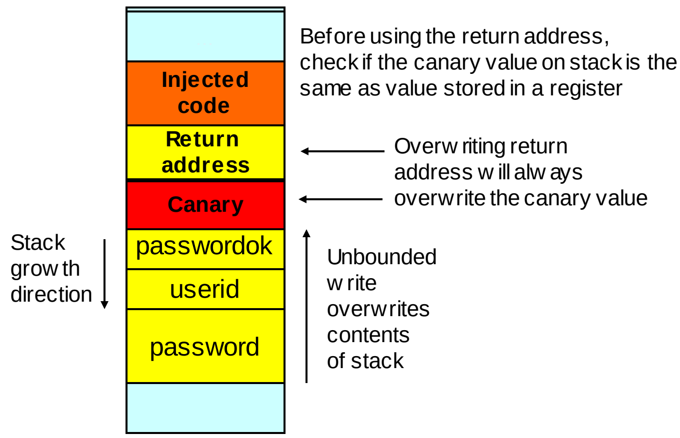
- A careful attacker can defeat this protection, by
  - overwriting the canary with the correct value
  - corrupting a pointer to point to the return address



- 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,

Countermeasure – Stack Protection

- Canary for tamper detection



- No code execution on stack

Thwarting Buffer Overflow Attacks

- **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`

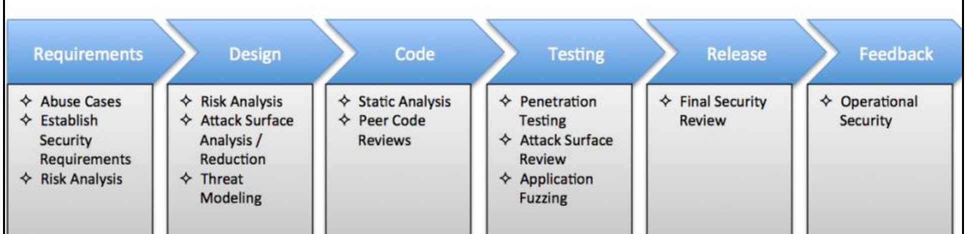
- **The StackGuard Protection Scheme**

$ 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

Q & A