Information Security

Chapter 02: Software & OS Security

Nguyễn Đăng Quang

Contents

Bad Software is Ubiquitous

- The **European Space Agency’s Ariane 5 Flight 501** was destroyed 40 seconds after takeoff (June 4, 1996).
- The **US$1 billion** prototype rocket **self-destructed due to a bug** in the on-board guidance software

Bad Software is Ubiquitous

A bug in the code controlling the Therac-25 radiation therapy machine was directly responsible for at least five patient deaths in the 1980s when it administered excessive quantities of X-rays

Bad Software is Ubiquitous

The **software error of a MIM-104 Patriot**, caused its system clock to drift by one third of a second over a period of one hundred hours — resulting in failure to locate and intercept an incoming missile. The Iraqi missile impacted in a military compound in Dhahran, Saudi Arabia (February 25, 1991), **killing 28 Americans**

Complexity – Lines Of Code

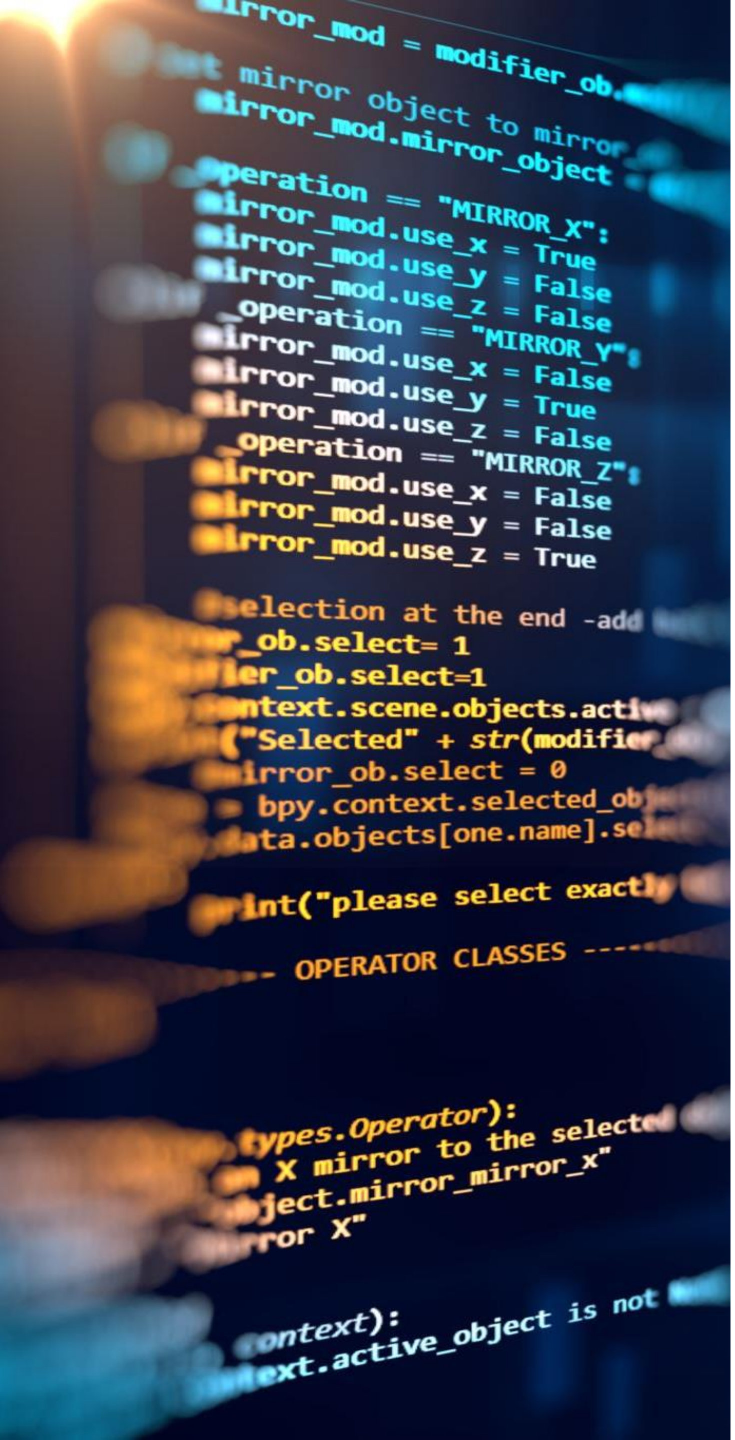
System	Lines of Code (LOC)
Netscape	17 million
Space Shuttle	10 million
Linux kernel 2.6.0	5 million
Windows XP	40 million
Mac OS X 10.4	86 million
Boeing 777	7 million

A new car contains more LOC than was required to land the Apollo astronauts on the moon

Lines of code & Bugs

Conservative estimate: 5 bugs/10,000 LOC

- Typical: 3k EXE's of 100k LOC each.
- Conservative estimate: 50 bugs/EXE.
- Implies about 150k bugs per computer.
- A 30000-node network has 4.5 billion bugs
- Maybe only 10% of bugs security-critical and only 10% of those remotely exploitable.
- Then "only" 45 million critical security flaws!



IEEE Quality Terminology

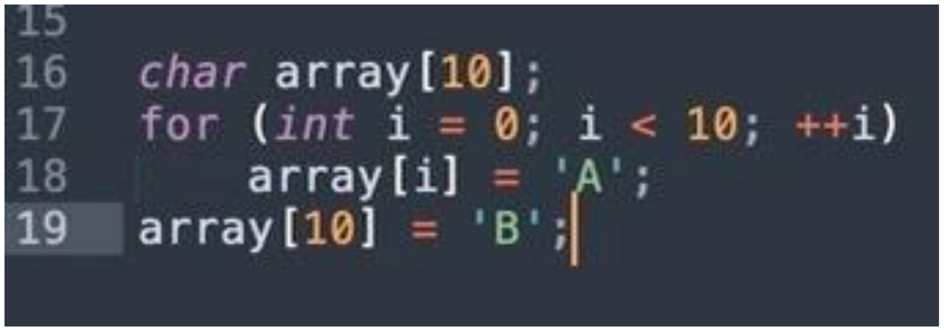
IEEE Standard 729 defines quality-related terms:

- Errors: A *human mistake* in performing some software-related activity, such as specification or coding.
- Fault: An incorrect step, command, process or data definition in a piece of software (*internal to program*)
- Failure: A departure from the system's desired behavior

$$\begin{tikzcd}\text{error} \arrow[r] & \text{fault} \arrow[r] & \text{failure}\end{tikzcd}$$

Example

- The code has an **error**
- This error might cause a **fault**
  - Incorrect internal state
- If a fault occurs, it might lead to a **failure**
  - Program behaves incorrectly (external)
- The term **flaw** is used for all of the above



```
char array[10];
for (int i = 0; i < 10; ++i)
    array[i] = 'A';
array[10] = 'B';
```

Program Error, Fault, Flaw examples

Error

A program error refers to an unintended and incorrect behavior that occurs during the execution of a program. It is a deviation from the expected or desired behavior. Program errors can manifest as crashes, incorrect outputs, unexpected termination, or other abnormal behaviors.

Example

Consider a program that is designed to calculate the average of a set of numbers. However, due to a logic error in the code, the program performs the calculation incorrectly, resulting in an inaccurate average being displayed.

Program Error, Fault, Flaw examples

Fault

A program fault, also known as a bug, is a flaw or defect in the program's code or design that causes it to behave in an unintended or incorrect manner. It is a mistake made by a developer that leads to a program error when the code is executed.

Example

In a video game, there might be a fault in the collision detection code, causing objects to pass through each other instead of colliding properly. This flaw in the code leads to an error where objects do not interact as intended.

Program Error, Fault, Flaw examples

Flaw

A flaw refers to a weakness or vulnerability in the design or implementation of a program that can be exploited to compromise its security, stability, or integrity. It indicates a fundamental defect or oversight in the software.

Example

An application that fails to properly validate user input and allows unfiltered SQL queries to be executed is said to have a flaw. This flaw opens the door to SQL injection attacks, where a malicious user can manipulate the input and execute unauthorized database operations.

How to prevent program flaws and vulnerabilities

1. Input Validation and Sanitization
2. Secure Coding Practices
3. Secure Configuration
4. Secure Development Lifecycle (SDL)
5. Secure Third-Party Libraries
6. Data Secure Authentication and Authorization
7. Encryption
8. Error Handling and Logging
9. Regular Security Testing
10. Security Awareness and Training

Defensive Programming

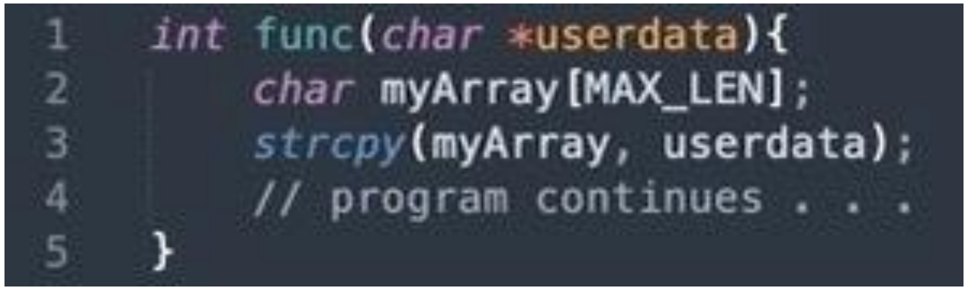
- ❑ Defensive programming is a coding approach that aims to develop programs that are capable of detecting potential security abnormalities and make predetermined responses. It ensures the continuing function of a piece of software under unforeseen circumstances.
- ❑ Defensive programming practices are often used where high availability, safety, or security is needed.
- ❑ It is intended to improve software and source code in terms of general quality, making the source code comprehensible, and making the software behave in a predictable manner despite unexpected inputs or user actions

Related
what is the difference between error handling and defensive programming
how can defensive programming help prevent software failures
what are some common mistakes that defensive programming can help avoid

Defensive Programming Techniques

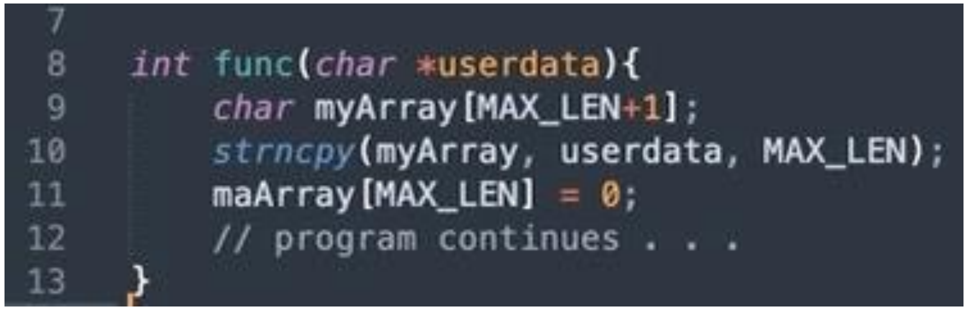
1. **Input validation**: This technique involves checking the input data for errors or invalid values before processing it.
2. **Threat modeling**: This technique involves identifying potential threats to your program and designing it to be resistant to those threats.
3. **Modularization and clear commenting**: This technique involves breaking your program down into smaller, more manageable modules and adding clear comments to make the code more understandable.
4. **Error handling and logging**: This technique involves handling errors gracefully and logging them for later analysis.

Example



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

Buffer overflow

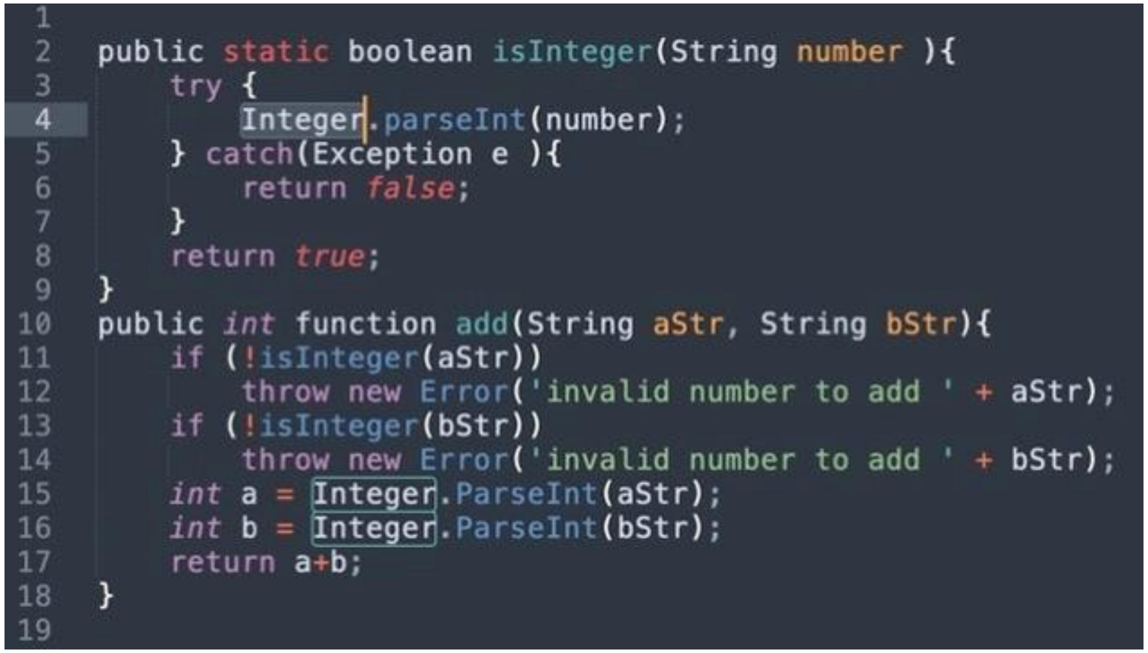


```

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

Defensive programming

Defensive coding example


```

public static boolean isInteger(String number ){
    try {
        Integer.parseInt(number);
    } catch(Exception e ){
        return false;
    }
    return true;
}
public int function add(String aStr, String bStr){
    if (!isInteger(aStr))
        throw new Error('invalid number to add ' + aStr);
    if (!isInteger(bStr))
        throw new Error('invalid number to add ' + bStr);
    int a = Integer.ParseInt(aStr);
    int b = Integer.ParseInt(bStr);
    return a+b;
}

```

Secure Programming

- Secure programming is a category of defensive programming that emphasizes the development of highly secure programs. It focuses on developing software that is resistant to bugs and vulnerabilities.
- Secure programming practices include security by design, password management, access control, error handling and logging, system configuration, threat modeling, cryptographic practices, input validation, and output encoding.

Best practices

- ❑ Data input validation
- ❑ Authentication and password management
- ❑ Cryptographic Practices
- ❑ Error Handling and Logging
- ❑ Data Protection
- ❑ Communication Security
- ❑ Adopt a secure coding standard

Common unsecure functions should be avoided

1. **Insecure Input Functions:** gets() in C/C without specifying field width or format specifiers
2. **Insecure String Manipulation Functions:** without bounds checking: strcpy(), strcat(), and sprintf.
3. **Insecure Memory Functions:** malloc(), calloc() can lead to memory leaks.
4. **Insecure Cryptographic Functions:** using weak or outdated cryptographic functions MD5, SHA-1
5. **Insecure Dynamic SQL Queries:** SQL queries dynamically without proper input validation and sanitization.
6. **Insecure Command Execution Functions:** system(), exec() can lead to command injection vulnerabilities

The Art

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

Secure Software

- In software engineering, try to ensure that a program does what is intended
- ***Secure*** software engineering requires that software **does what is intended…**

…and nothing more

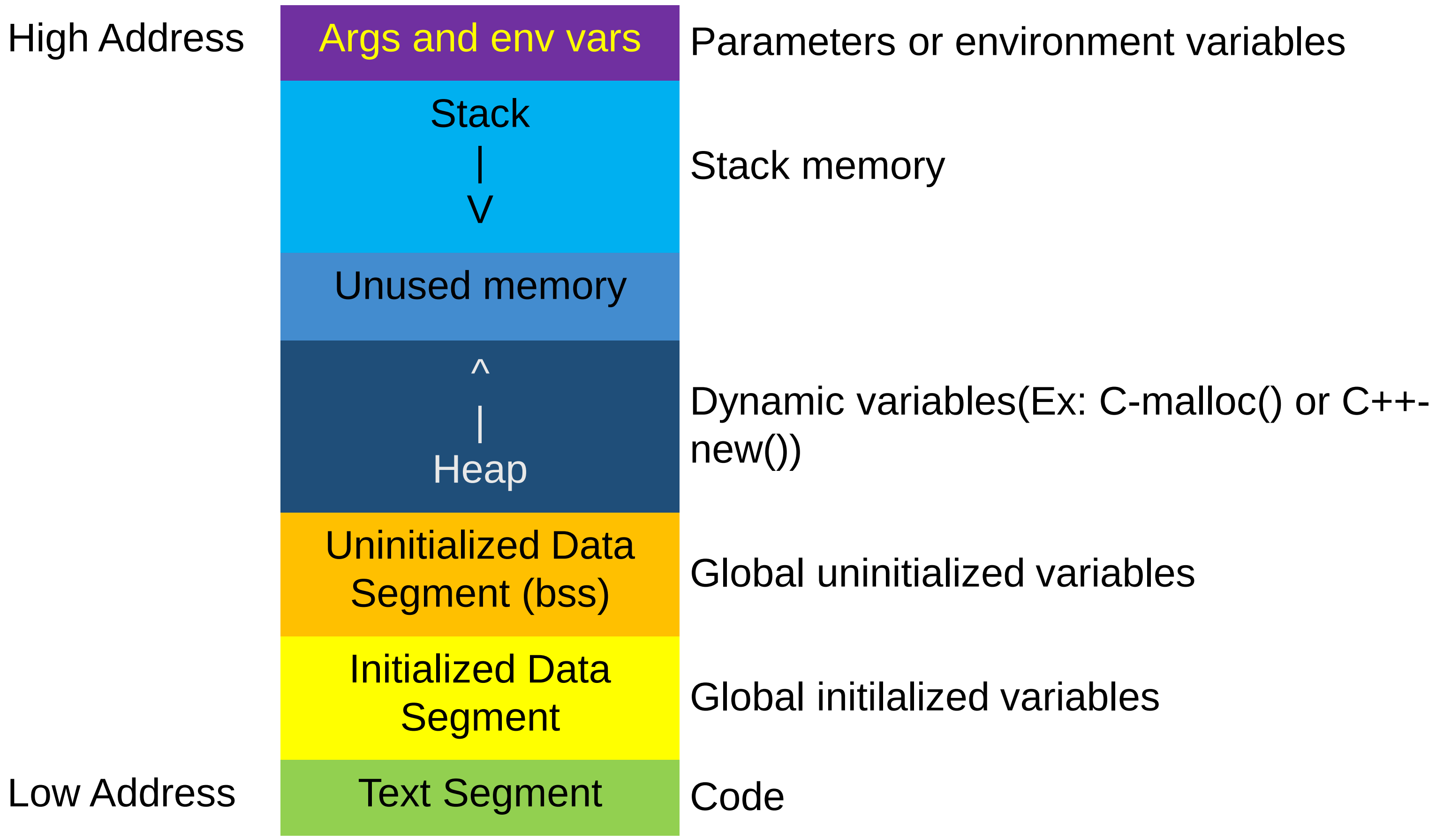
- Absolutely secure software? Dream on…

  Absolute security ***anywhere*** is impossible

- Manage software risks

Buffer Overflow

Program memory layout


High Address
Args and env vars
Parameters or environment variables
Stack
|
V
Stack memory
Unused memory
^
|
Heap
Dynamic variables(Ex: C-malloc() or C++-new())
Uninitialized Data Segment (bss)
Global uninitialized variables
Initialized Data Segment
Global initilalized variables
Low Address
Text Segment
Code

Secure Software

- In software engineering, try to ensure that a program does what is intended
- ***Secure*** software engineering requires that software **does what is intended…**

…and nothing more

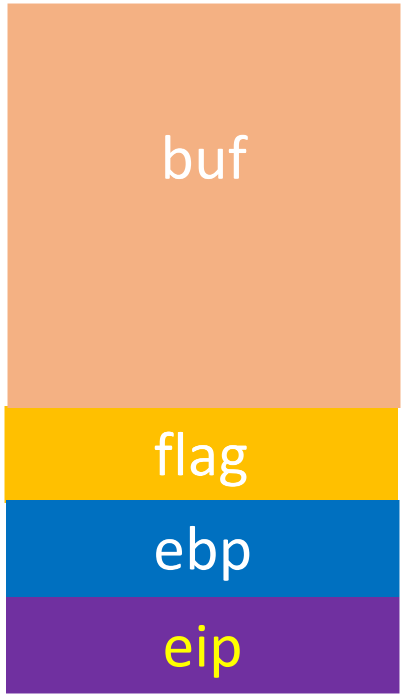
- Absolutely secure software? Dream on…

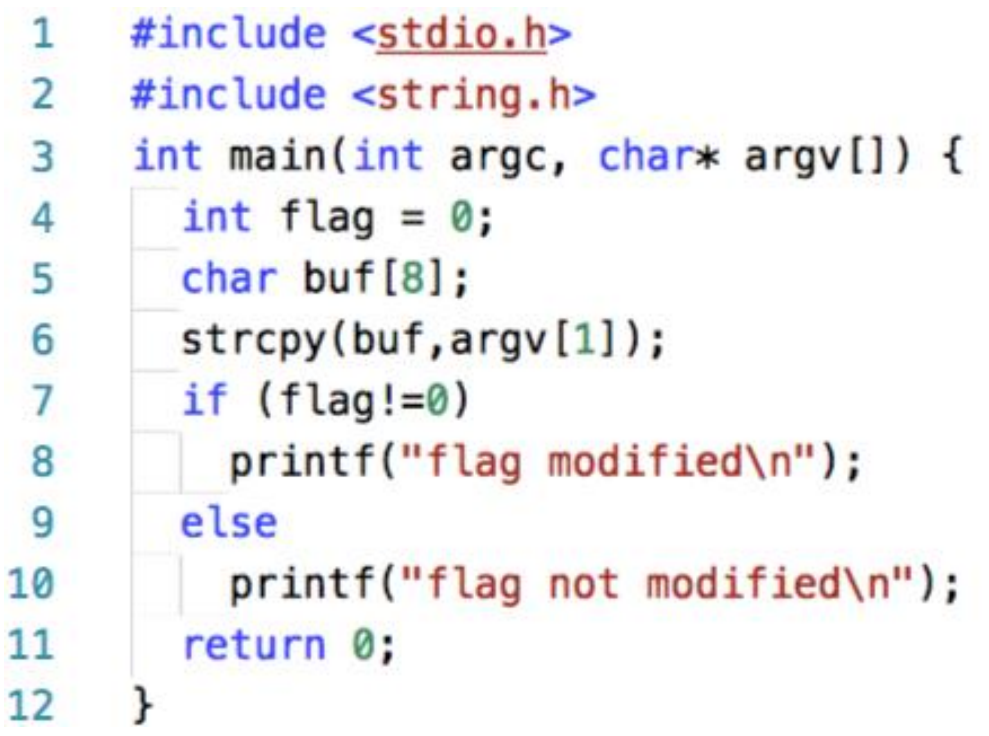
  Absolute security ***anywhere*** is impossible

- Manage software risks

Simple Buffer Overflow

- Consider boolean flag for authentication
- Buffer overflow could overwrite flag allowing anyone to authenticate

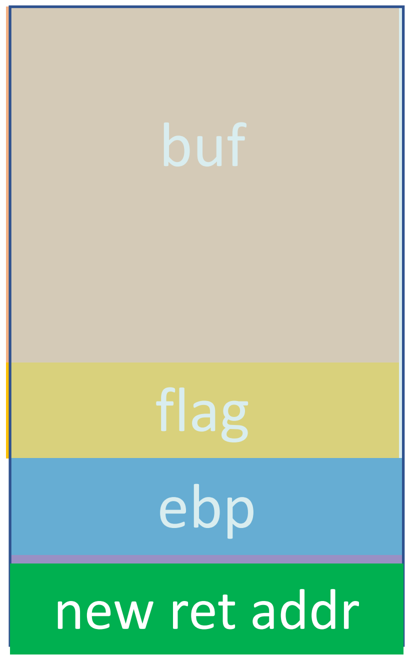




```c
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]) {
  int flag = 0;
  char buf[8];
  strcpy(buf,argv[1]);
  if (flag!=0)
    printf("flag modified\n");
  else
    printf("flag not modified\n");
  return 0;
}
```

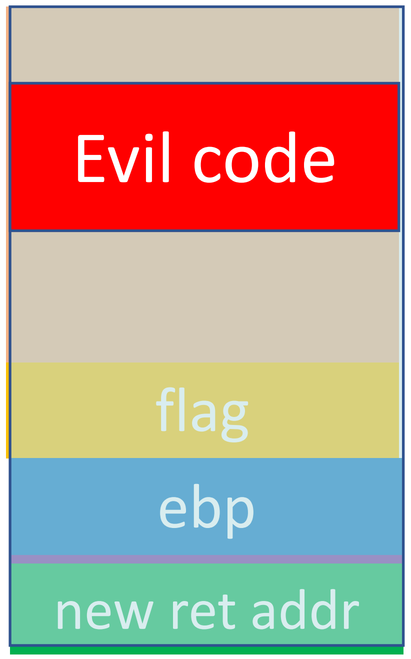
Smashing the stack

- Return address (eip) is overwritten.
- Crash is likely



Smashing the stack – Better idea

- Evil code is injected into the buf
- Return address (eip) is overwritten with address of evil code
- The evil code then activated with the new return address



Practice

```
//vuln.c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
        /* [1] */ char buf[256];
        /* [2] */ strcpy(buf,argv[1]);
        /* [3] */ printf("Input:%s\n",buf);
        return 0;
}
```

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

or

```
$gcc  vuln.c –o vuln -fno-stack-protector -m32 -g
```

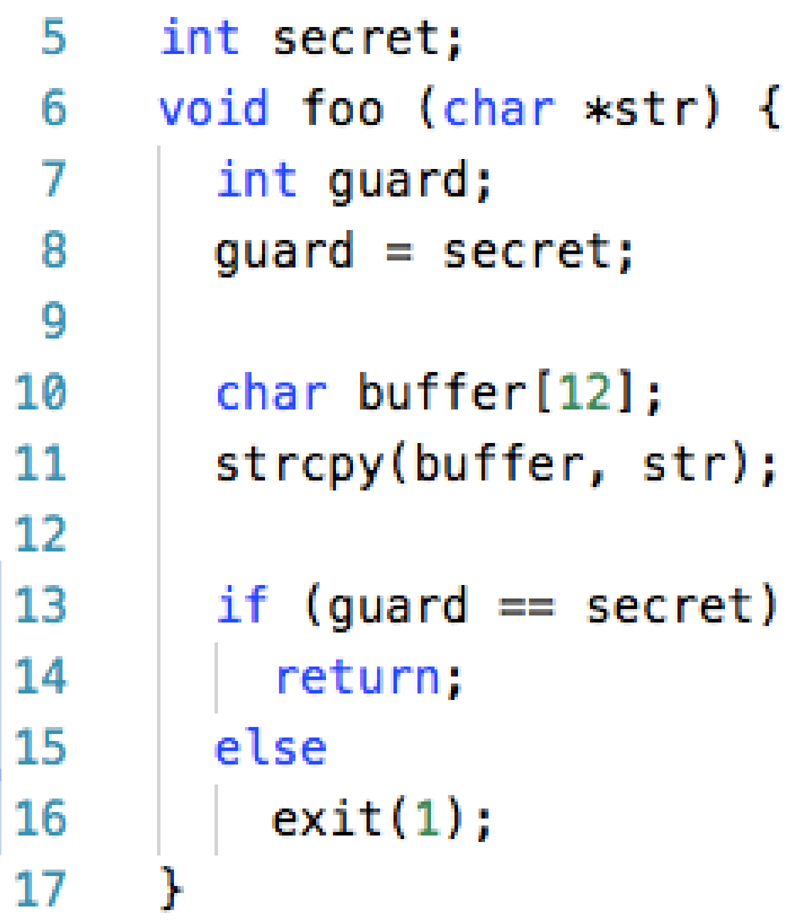
Buffer Overflow Countermeasures

1. **Safer Functions**: Use strncpy, snprintf, strncat, fgets instead of strcpy, sprintf, strcat, gets.
2. **Program Static Analyzer**: this type of solution warns developers of the patterns in code that may potentially lead to buffer overflow vulnerabilities.
3. **Programming Language**: Choose the language itself can do some check against buffer overflow like Java or Python (safer for development) [OSWAP, 2014]

Buffer Overflow Countermeasures

4. Compiler:

- StackGuard: Save a copy of the return address at a safer place



```
int secret;
void foo (char *str) {
  int guard;
  guard = secret;

  char buffer[12];
  strcpy(buffer, str);

  if (guard == secret)
    return;
  else
    exit(1);
}
```

implemented in compiler (gcc) automatically (canary)

canary is stored at gs:20 which pointed to a memory region isolated from the stack

-fno-stack-protector disable this feature

Buffer Overflow Countermeasures

5. **Operating System:**
   implemented in OS loader program: ASLR (Address Space Layout Randomization)

Set ASLR

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

or

```
sudo sysctl –w kernel.randomize_va_space = 0 (1,2)
```

Buffer Overflow Countermeasures

6. **Hardware Architecture:**

   Modern CPUs support NX-bit (No-eXecute) feature: memory areas when marked as non executable, the CPU will refuse to execute any code residing in these areas of memory.

   gcc -z execstack option turning off this feature.

Buffer Overflow

- A major security threat yesterday, today, and tomorrow
- The good news?
  - It <u>is</u> possible to reduce overflow attacks (safe languages, NX bit, ASLR, education, etc.)
- The bad news?
  - Buffer overflows will exist for a long time
  - Why? Legacy code, bad development practices, clever attacks, etc.