ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

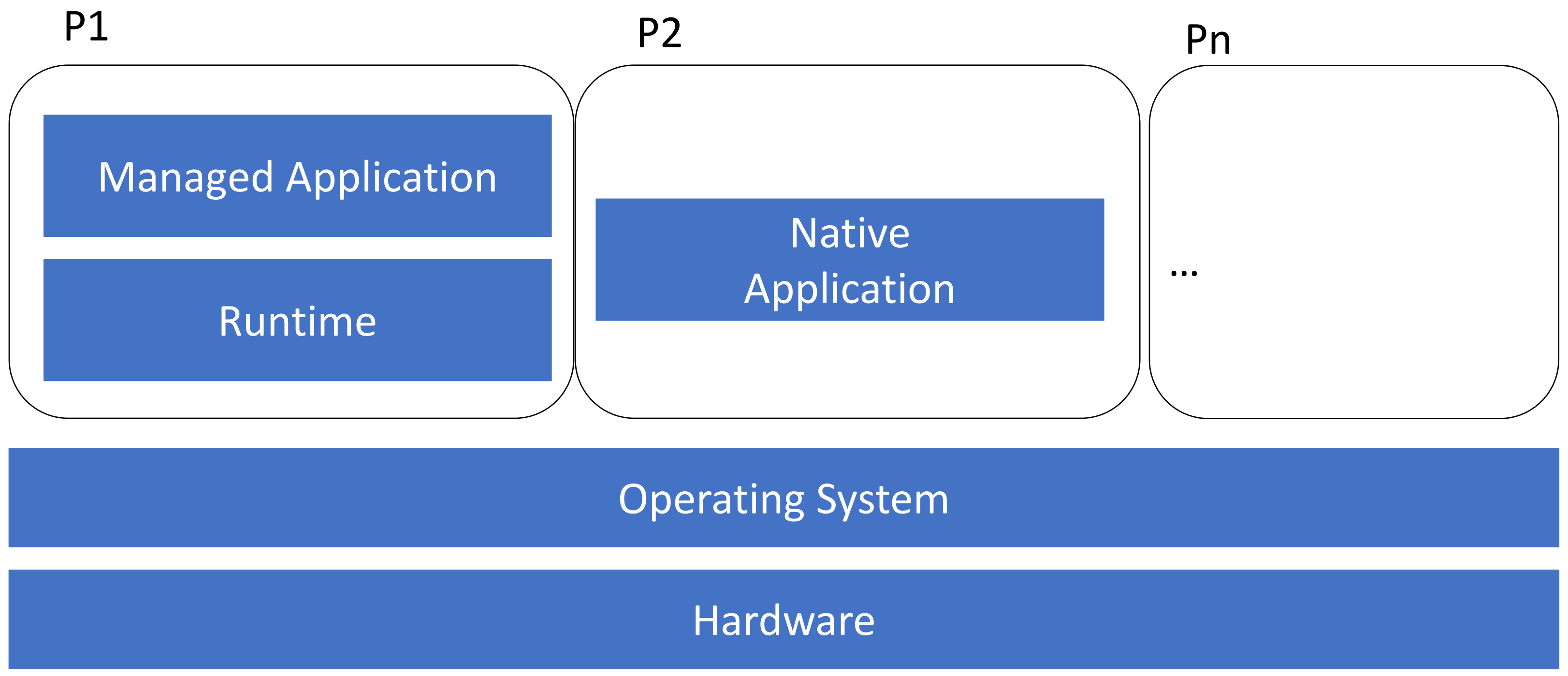
Segurança em linguagens, runtimes e sistemas operativos

ISEL – Instituto Superior de Engenharia de Lisboa
Rua Conselheiro Emídio Navarro, 1 | 1959-007 Lisboa

Agenda

- Elements of a computer system
- Security in runtime and native applications
- Security in operating systems
    - Separation vs. Mediation
    - Access control models
    - Examples

Sistema computacional



- Language Runtimes Ensure Type, Memory, and Execution Flow Safety
- Operating system ensures separation and mediation between processes and hardware

Native applications

(In)Security in native applications

- Applications written in C/C++ can perform address arithmetic and arbitrarily access memory (with limits only imposed by the OS)
  - unlike virtual machines for high-level languages (java, C#, python),
- Vulnerability
  - Write to a buffer beyond its limit

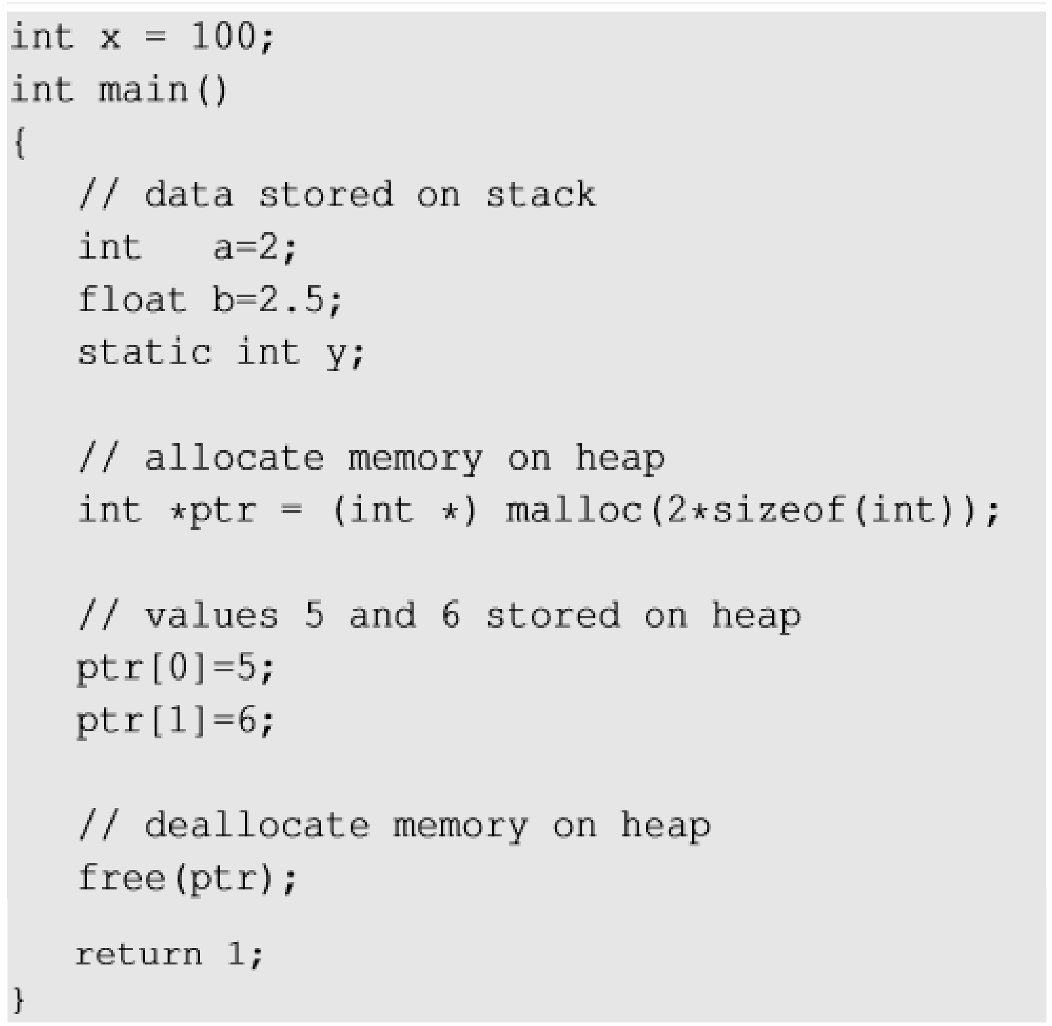
```
void TheProblem(char * in, int len){
    char buf[20];
    memcpy(buf, in, len);
}
```

- Top25 CWE List Most Common Mistake
  - https://cwe.mitre.org/data/definitions/787.html
- “Classic” error but in its variants it has +10000 entries in Common Vulnerabilities and Exposures (http://cve.mitre.org/cve/search_cve_list.html)
  - Ex: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3568

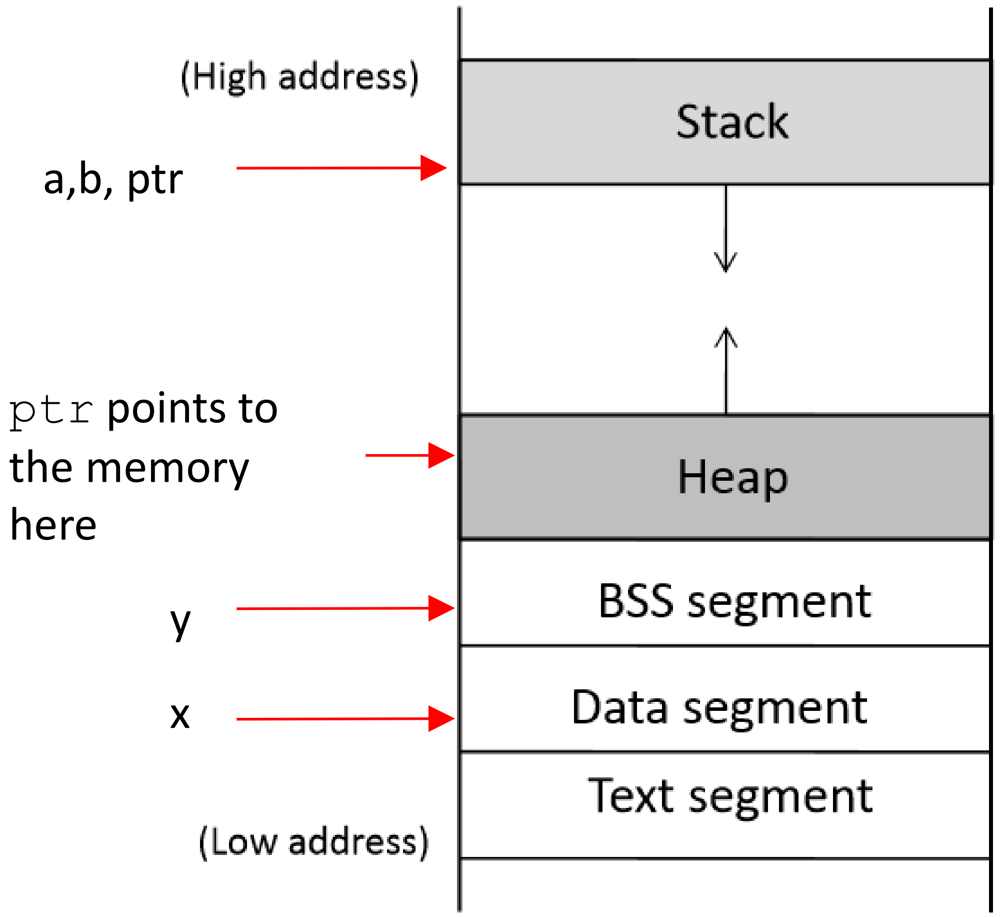
Consequences of a buffer overflow

- Overlapping can result in:
  - Read private process information
    - http://heartbleed.com/
  - Invalid instruction
  - Address does not exist
  - Access violation
  - Attacker Code Execution

Organização do stack

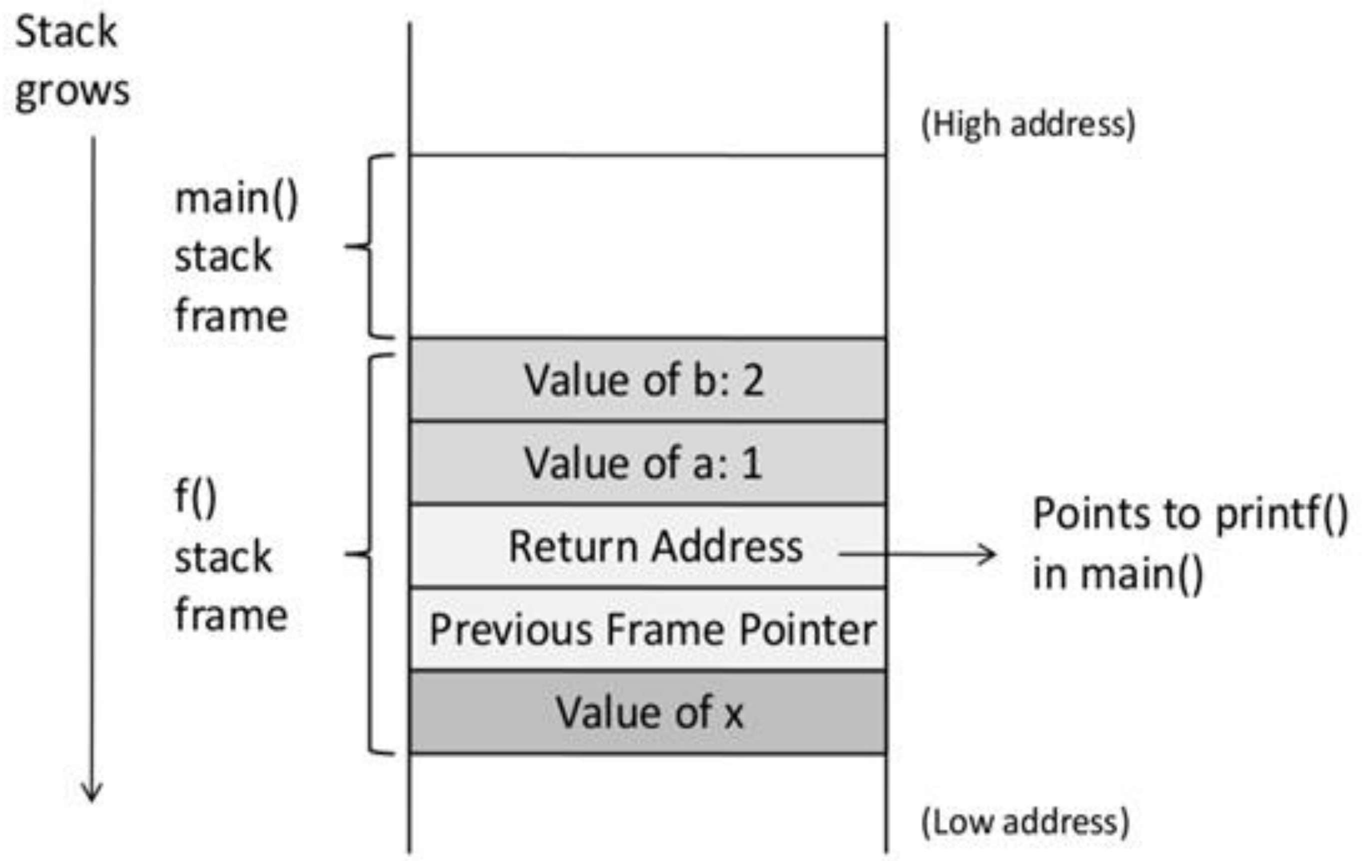


```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

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

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

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```

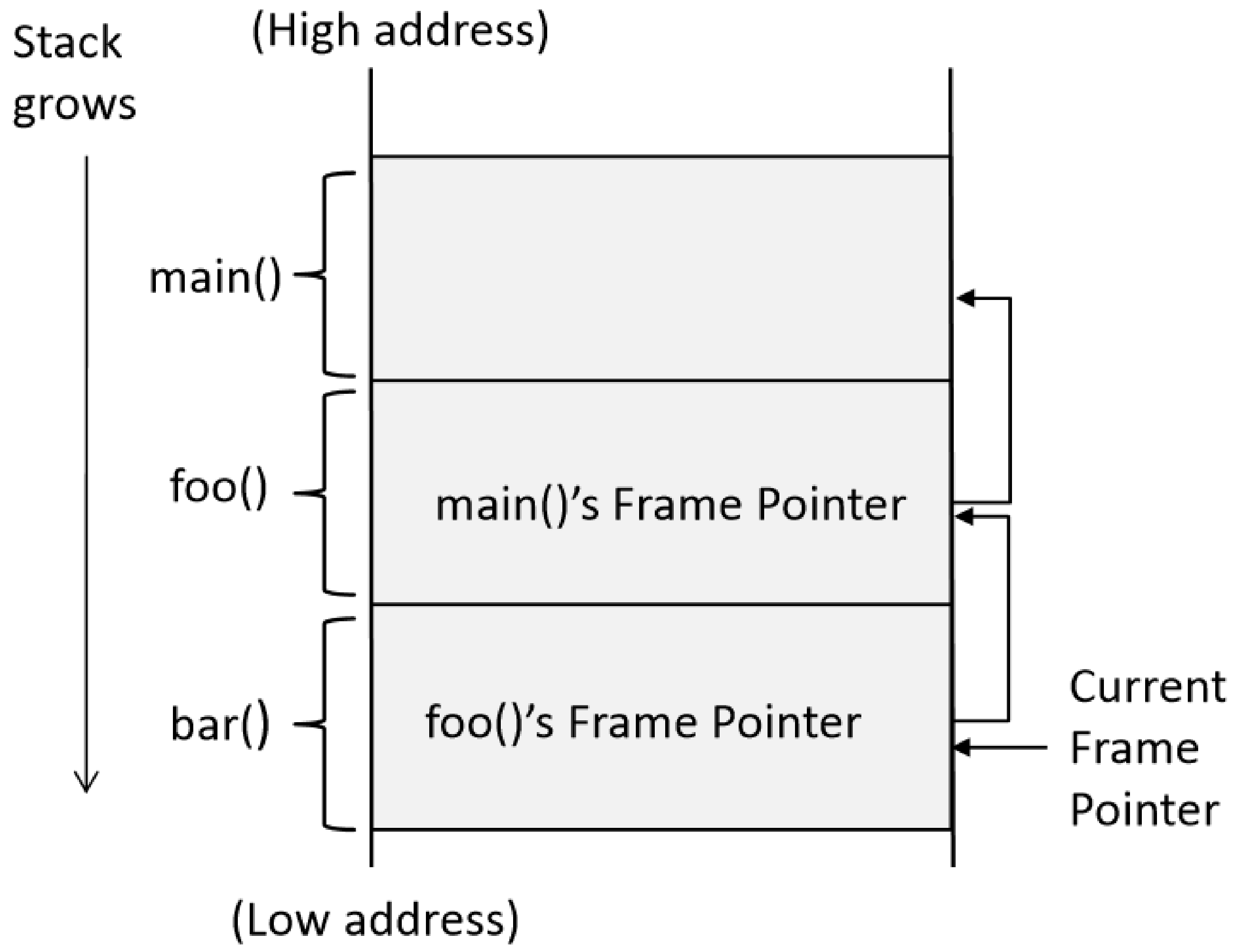


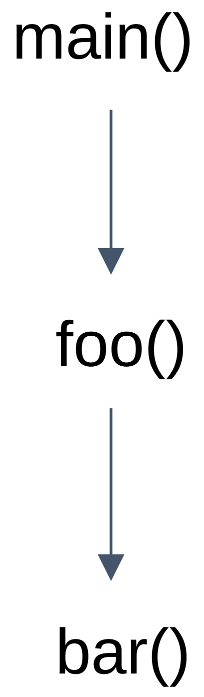
Calls to Functions

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



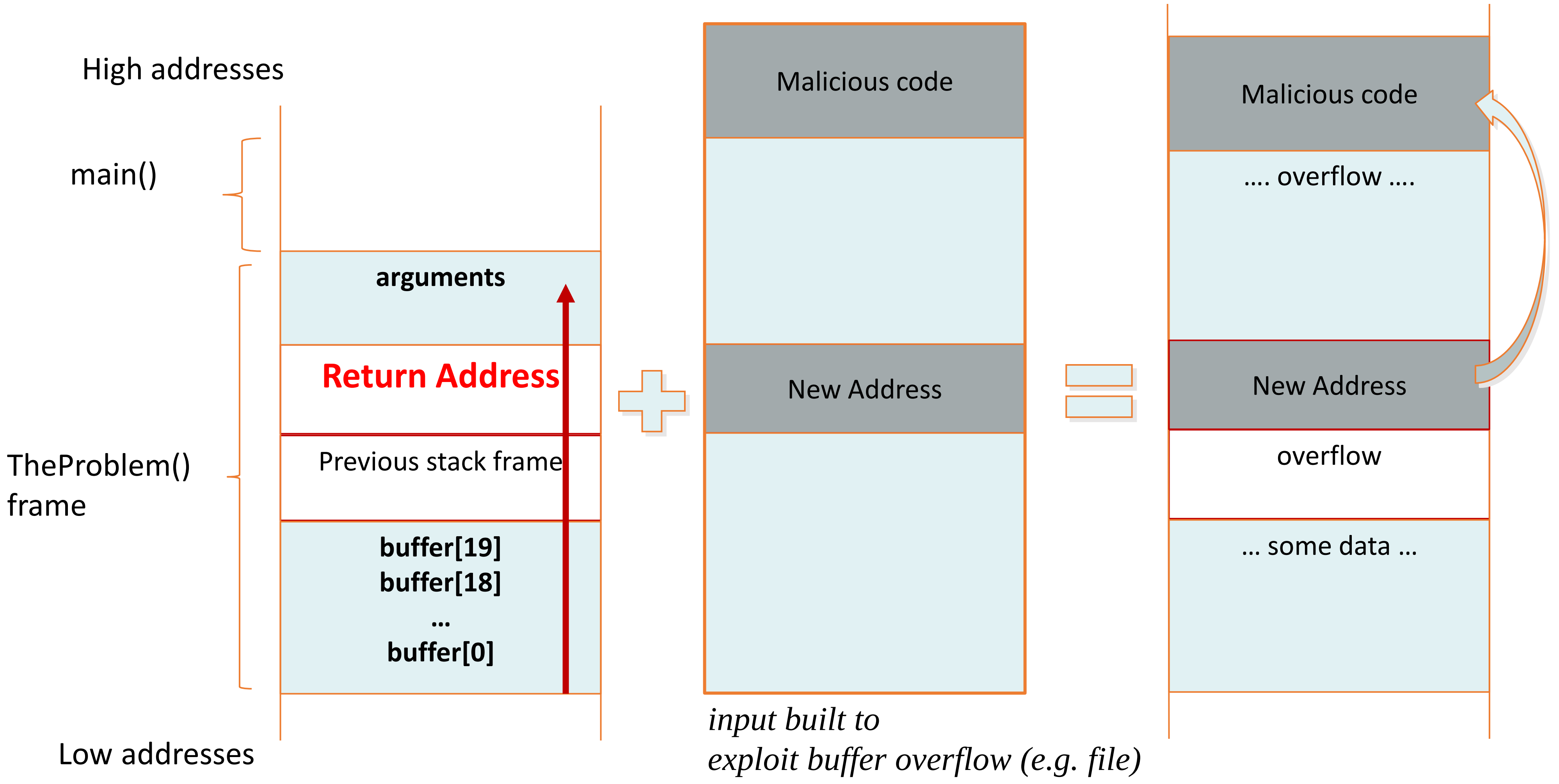
Stack organization between calls





Code injection

```
void TheProblem(char * in, int len){
    char buf[20];
    memcpy(buf, in, len);
}
```

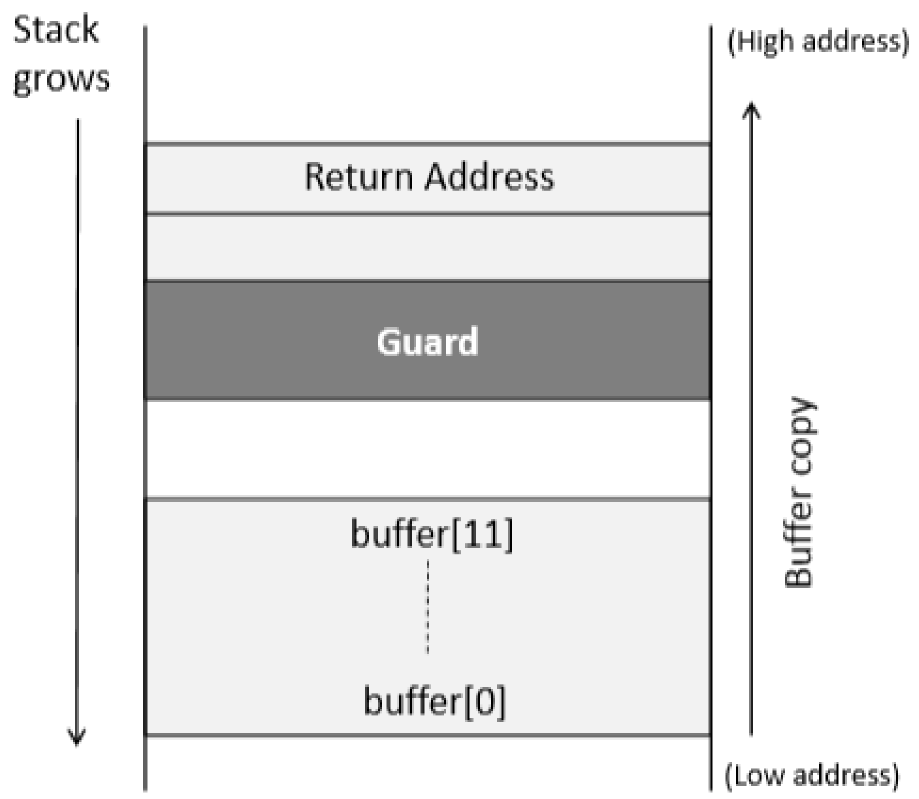


Buffer overflow vulnerability mitigation

- At source code level
  - Use more secure functions like strncpy or strncat, which check the data size before copying
- At compiler level
  - "Canaries" that check if the return address has been superimposed
- At the operating system level
  - Address space layout randomization (ASLR)
- At the hardware level
  - Unexecutable Stack
  - This protection can be bypassed with a variant known as Return-to-Libc

Stack guard

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

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

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



Análise de um ataque

- **Adobe Acrobat Buffer Overflow Vulnerability (CVE-2009-0658)**
- Adobe Acrobat and Reader version 9.0 and earlier are vulnerable to a buffer overflow, caused by improper bounds checking when parsing a malformed JBIG2 image stream embedded within a PDF document. By persuading a victim to open a malicious PDF file, a remote attacker could overflow a buffer and execute arbitrary code on the system with the privileges of the victim or cause the application to crash.
- The vulnerability is exploited by convincing a victim to open a malicious document on a system that uses a vulnerable version of Adobe Acrobat or Reader. An attacker must deliver a malicious document to the victim and relies upon the user to open it. Then the code execution achieved by the attacker depends on the privilege level of the user on the system and could potentially result in High impacts to Confidentiality, Integrity, and Availability.

Ataque analysis with CVSS - CVE-2009-0658

Metric	Value	Comments
Attack Vector	Local	A flaw in the local document software that is triggered by opening a malformed document.
Attack Complexity	Low	
Privileges Required	None	
User Interaction	Required	The victim needs to open the malformed document.
Scope	Unchanged	
Confidentiality	High	Assuming a worst-case impact of the victim having High privileges on the affected system.
Integrity	High	Assuming a worst-case impact of the victim having High privileges on the affected system.
Availability	High	Assuming a worst-case impact of the victim having High privileges on the affected system.

https://www.first.org/cvss/examples

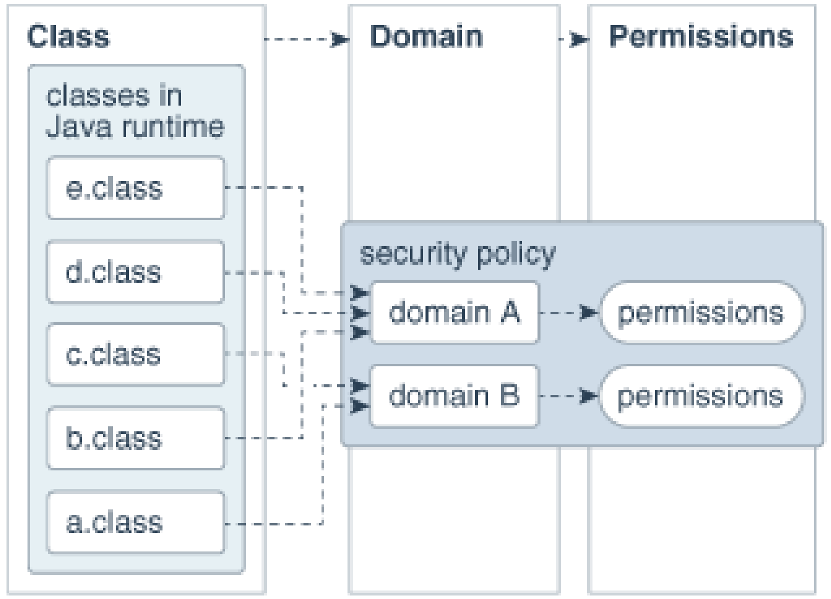
Runtimes and Operating Systems

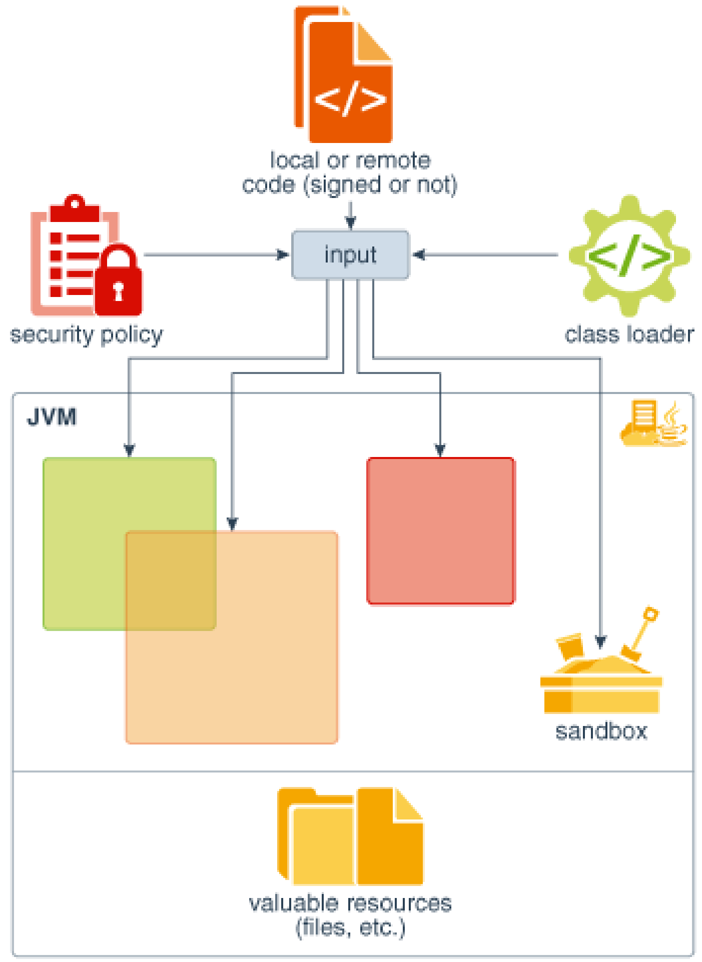
Security in languages and execution environments

- The impact of vulnerabilities in native languages like C and C++ is only limited by the operating system
- Platforms like Java and .NET have a wide range of security systems
  - At the language level: type system, memory security, flow security
  - Byte Code Checker
  - Permissions by domain
  - Sandbox
  - Cryptographic libraries

Permissions and domains in Java

- Access control policy determines permissions
- Permissions associated with domains
- Domains



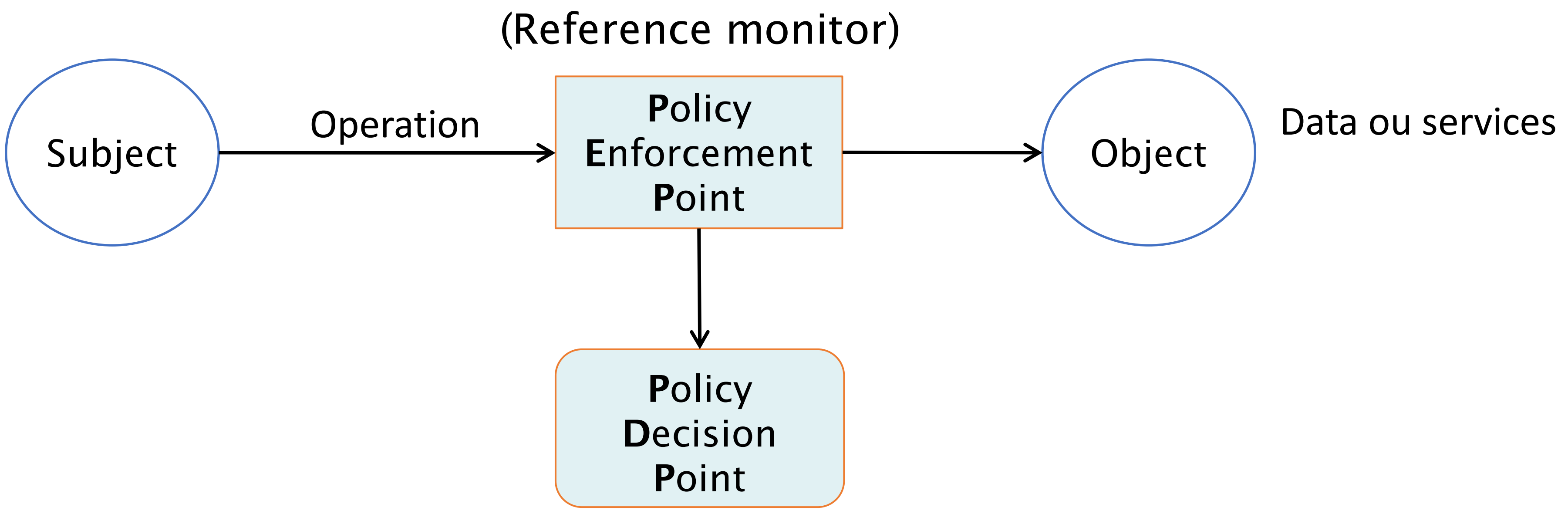

local or remote code (signed or not)
security policy
input
class loader
JVM
sandbox
valuable resources (files, etc.)

https://docs.oracle.com/en/java/javase/11/security/java-se-platform-security-architecture.html#GUID-D6C53B30-01F9-49F1-9F61-35815558422B

Segurança nos sistemas operativos

- Resource protection is done through:
  - Separation
  - Mediation
- Separation prevents processes from accessing memory zones of other processes or privileged instructions
  - virtual memory
  - User and kernel execution mode
- Mediation allows controlled access to resources such as file system objects
  - Access Control Lists
  - Capabilities

Reference monitor



- Properties of PEP:
    - Isolation: it should not be possible to change it.
    - Completeness: it should not be possible to bypass it.
    - Verifiable: it should be small and confined to the system's security core in order to facilitate verification of its correctness.

Elements of the access control system

- Security policy: defines access control rules

- Security model: formalization of how security policies are applied
  - Permissions by group; access control list; role-based access control

- Security Mechanisms: Low-level functions (software/hardware) that support the implementation of security models and policies

- PEP depends on security mechanisms
- PDP depends on security policy and model

Example: Unix/Linux

- Users have an identifier (user id – UID) and an account with that ID
  - Effective user ID (EUID) – id with which a program runs
  - Real user ID (RUID) - real user id
  - Special user, superuser (0), with administration rights
- File system namespace is used to access folders, files and devices
- File access control depends on EUID
- Each object has a simple access list
  - Owner's UID and your group's GID
  - Access permissions (r, w, x) to owner, group and other, eg rwx r-- r--
- Sticky bits used for privileged access control
  - Programs named Set-uid which run with superuser rights

Conceito Set-UID

- **Allow running a program with program owner privileges**
- Enables users to run programs with temporarily elevated privileges
- Example: passwd program (need access to the *password* file)
  - $ls -l /usr/bin/passwd

  -rwsr-xr-x 1 root root 41284 Sep 12 2012 /usr/bin/passwd
- When a normal program runs, RUID = EUID
- When a Set-UID program runs, RUID ≠ EUID. RUID is the user id, but EUID is the owner id.
  - If the owner of a program is root, the program runs with root privileges.

How it works

- A Set-UID program is like any other program, with owner root and the set-uid bit enabled in the allow list in the allow list

```
$ cp /bin/id ./myid
$ sudo chown root myid
$ ./myid
uid=1000(seed) gid=1000(seed) groups=1000(seed), ...
```

```
$ sudo chmod 4755 myid
$ ./myid
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

Example of a set-uid program

```
$ cp /bin/cat ./mycat
$ sudo chown root mycat
$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Feb 22 10:04 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

↰ Não é um programa priveligiado

```
$ sudo chmod 4755 mycat
$ ./mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8c...
daemon:*:15749:0:99999:7:::
...
```

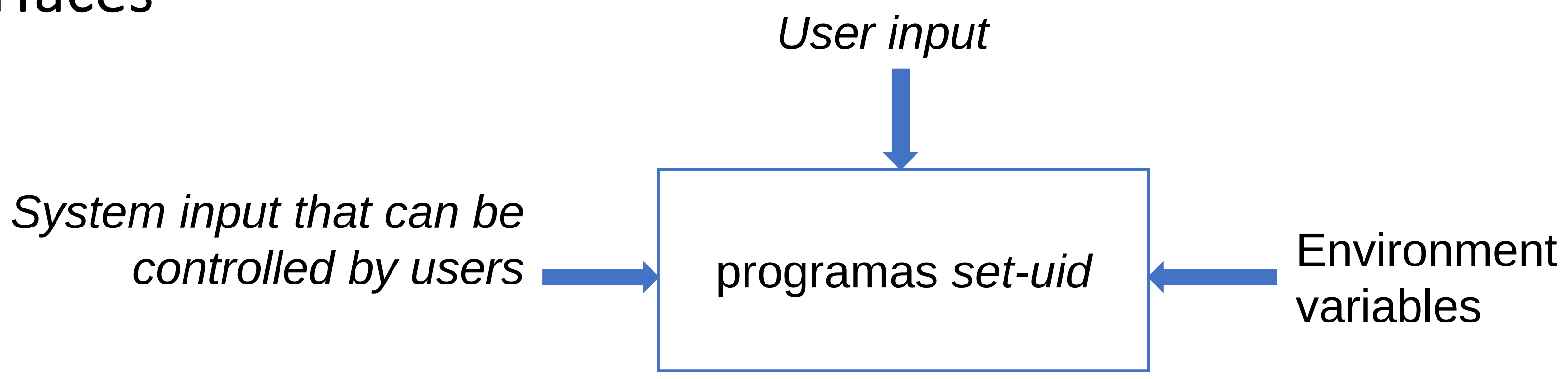
↰ Tornar-se um programa priveligiado

```
$ sudo chown seed mycat
$ chmod 4755 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

↰ É priveligiado, mas não priveligiado para root

Vulnerabilities in set-uid programs

- The set-uid concept is safe, but programs can be flawed, thus exposing different attack surfaces



- User input - for example, buffer overflow is a well known problem, which if exploited in a set-uid program can have a big impact
- System input - Writing to system zones like /tmp can be controlled by the user with symbolic links
- Environment Variables - Controlling environment variables such as PATH may run malicious programs

Case study – Command injection

Programs that call system commands

- Invoking external commands within a program
  - The external command is chosen by the Set-UID program
  - Users must not provide the command (not secure)

- Attack:
  - Users are often asked to provide input data for the command.
  - If the command is not invoked correctly, user input data can be turned into a command name, thus introducing a vulnerability.

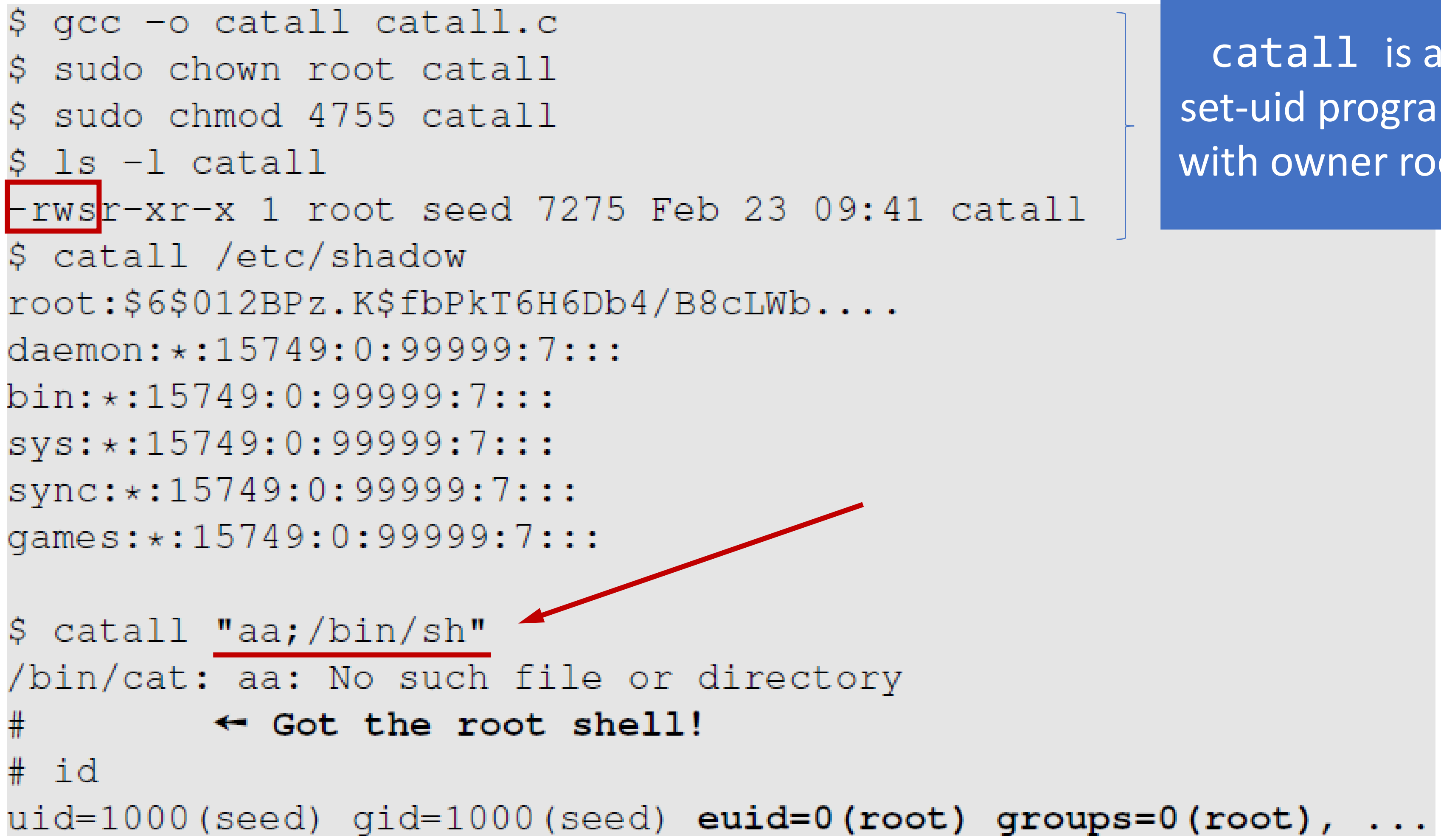
Program invocation: Insecure version

```c
int main(int argc, char *argv[])
{
  char *cat="/bin/cat";

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  char *command = malloc(strlen(cat) + strlen(argv[1]) + 2);
  sprintf(command, "%s %s", cat, argv[1]);
  system(command);
  return 0 ;
}
```

- A maneira mais fácil de invocar um comando externo é a função `system()`.
- Suponhamos um programa `Set-UID`, capaz de visualizar todos os arquivos, mas não pode alterar nenhum deles.
- Este programa deve executar o programa `/bin/cat`.

Como pode este programa correr outros comandos, com privilégios root?

Invoking Programs : Unsafe Approach (Continued)



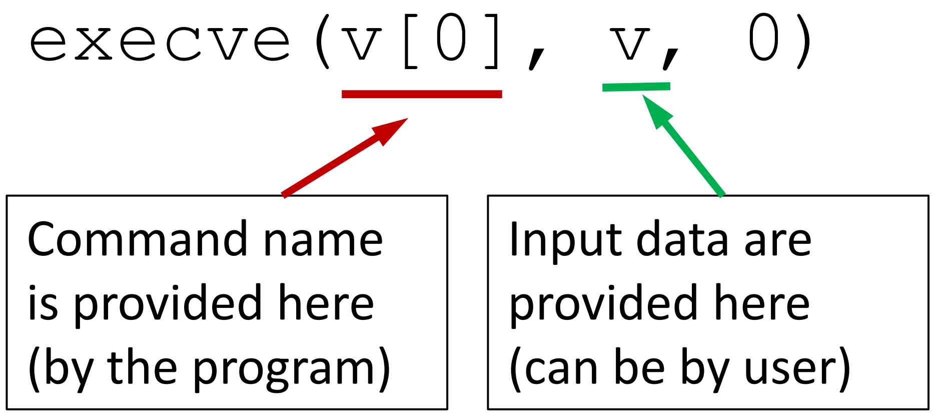
```
$ gcc -o catall catall.c
$ sudo chown root catall
$ sudo chmod 4755 catall
$ ls -l catall
-rwsr-xr-x 1 root seed 7275 Feb 23 09:41 catall
$ catall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::

$ catall "aa;/bin/sh"
/bin/cat: aa: No such file or directory
#         ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

`catall` is a set-uid program with owner root

Problem: A piece of data is interpreted as code (command name)

Safely invoke commands - execve()

```c
int main(int argc, char *argv[])
{
  char *v[3];

  if(argc < 2) {
    printf("Please type a file name.\n");
    return 1;
  }

  v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = 0;
  execve(v[0], v, 0);

  return 0 ;
}
```



`execve(v[0], v, 0)`

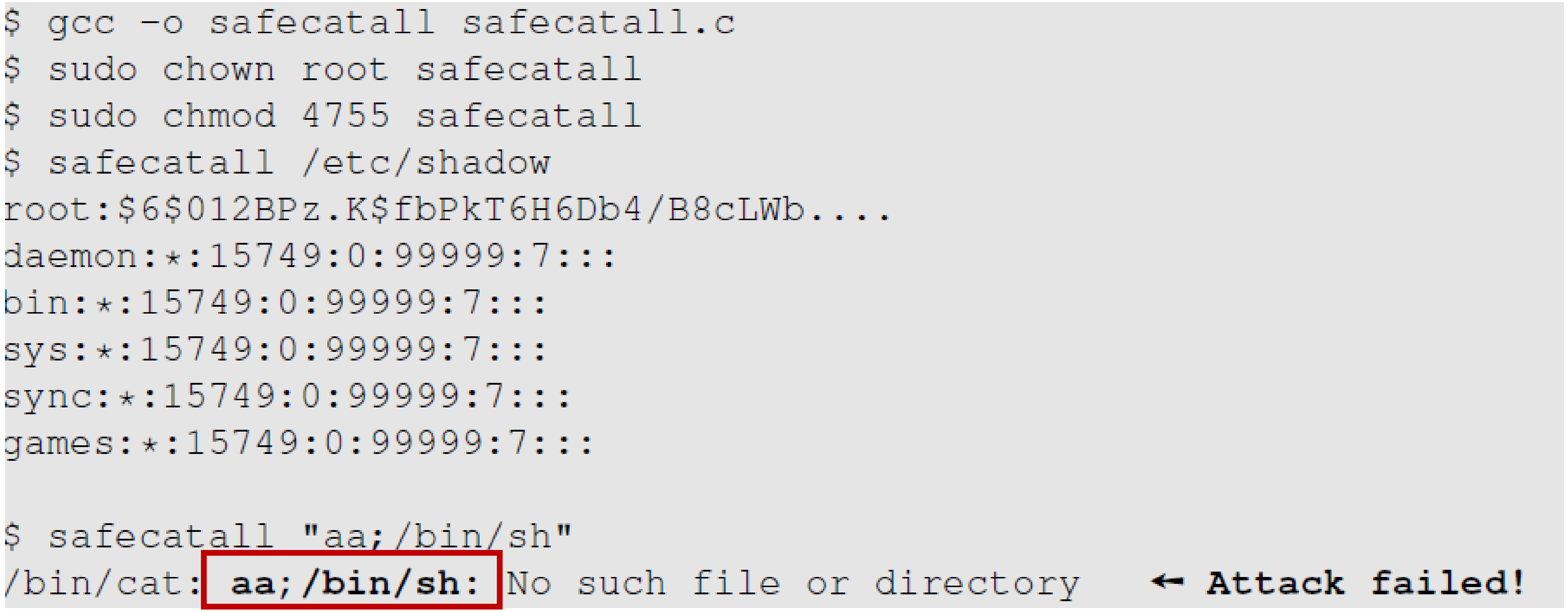
Command name is provided here (by the program)

Input data are provided here (can be by user)

Why is it safe?

Data and code follow different “channels”; there is no way for user data to become code

Safely invoke commands - execve()



```
$ gcc -o safecatall safecatall.c
$ sudo chown root safecatall
$ sudo chmod 4755 safecatall
$ safecatall /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWb....
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::

$ safecatall "aa;/bin/sh"
/bin/cat: aa;/bin/sh: No such file or directory    ← Attack failed!
```

Data is treated as dada, not as code

Caso prático: *Shellshock*

Vulnerabilidade na linha de comandos Linux

Contexto: Shell Functions

- Shell is a command line interpreter in operating systems
    - Provides an interface between the user and the operating system
    - Different shell types: sh, bash, csh, zsh, windows powershell etc.
- In particular bash shell is one of the most popular shell programs on Linux
    - The shellshock vulnerability is related to shell operations.

```
$ foo() { echo "Inside function"; }
$ declare -f foo
foo ()
{
    echo "Inside function"
}
$ foo
Inside function
$ unset -f foo
$ declare -f foo
```

Vulnerabilidades shellshock

- Vulnerability called Shellshock or bashdoor was published in September 2014 (CVE-2014-6271)
- This vulnerability exploits a bug in the bash program when converting environment variables to function definition
- The bug found exists in the GNU bash source code since August 5, 1989
- After identifying this bug, several other bugs were found in the widely used bash shell.
- Shellshock refers to the family of security bugs found in GNU bash

Vulnerabilidade Shellshock

- Parent process can pass a function to a child process as an environment variable
- Due to an error in the parsing logic, bash executes commands contained in the variable

```
$ foo='() { echo "hello world"; }; echo "extra";'
$ echo $foo
() { echo "hello world"; }; echo "extra";
$ export foo
$ bash_shellshock    ← Run bash (vulnerable version)
extra                ← The extra command gets executed!
seed@ubuntu(child):$ echo $foo

seed@ubuntu(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
```

Commando extra