ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

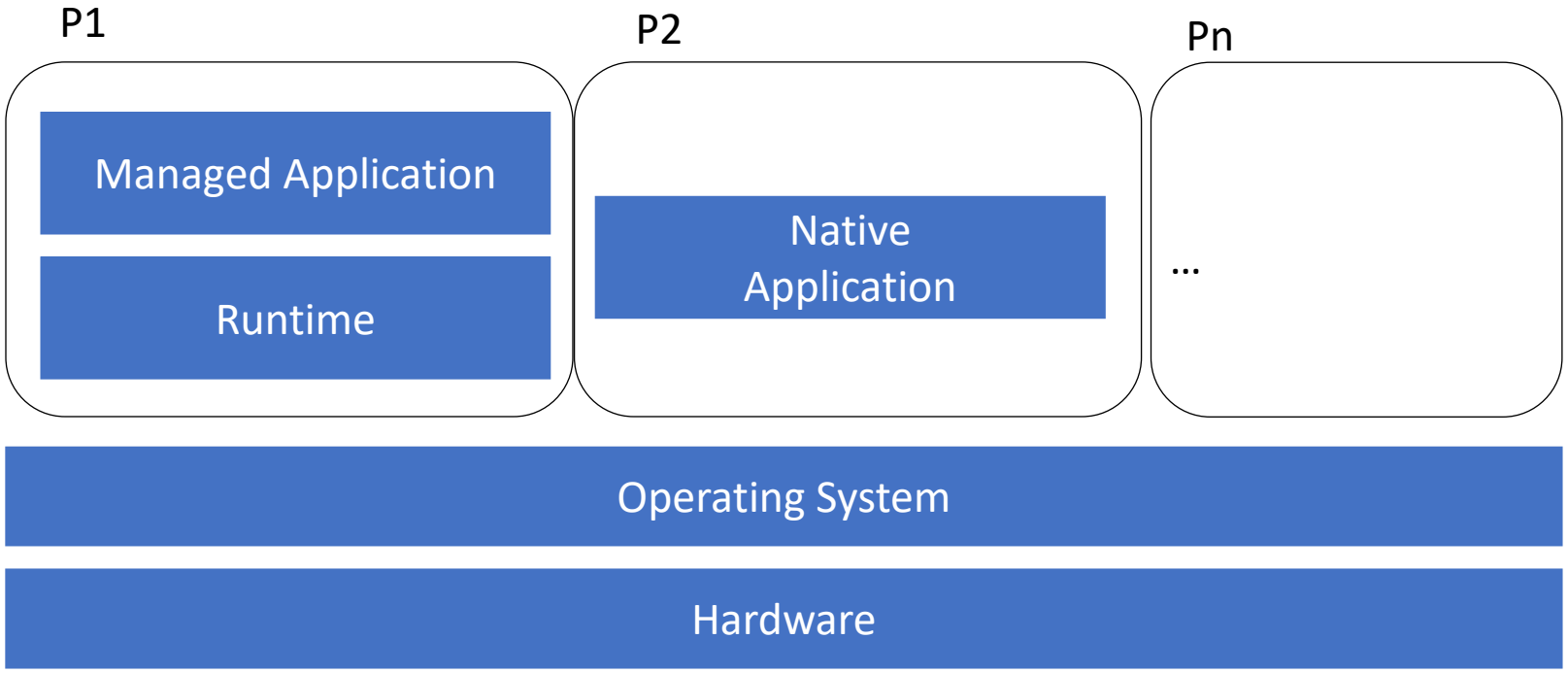
Segurança em linguagens, runtimes e sistemas operativos

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

Agenda

- Elementos de um sistema computacional
- Segurança nos runtime e aplicações nativas
- Segurança nos sistemas operativos
  - Separação vs. Mediação
  - Modelos de controlo de acessos
  - Exemplos

Sistema computacional



- Runtimes de linguagens garantem segurança de tipos, memória e fluxo de execução
- Sistema operativo garante sepação e mediação entre processos e hardware

Aplicações nativas

(In)Segurança em aplicações nativas

- As aplicações escritas em C/C++ podem realizar aritmética de endereços e aceder arbitrariamente à memória (com limites apenas impostos pelo SO)
  - ao contrário da máquinas virtuais para linguagens de alto nível (java, C#, python),
- Vulnerabilidade
  - Escrever num *buffer* para além do seu limite

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

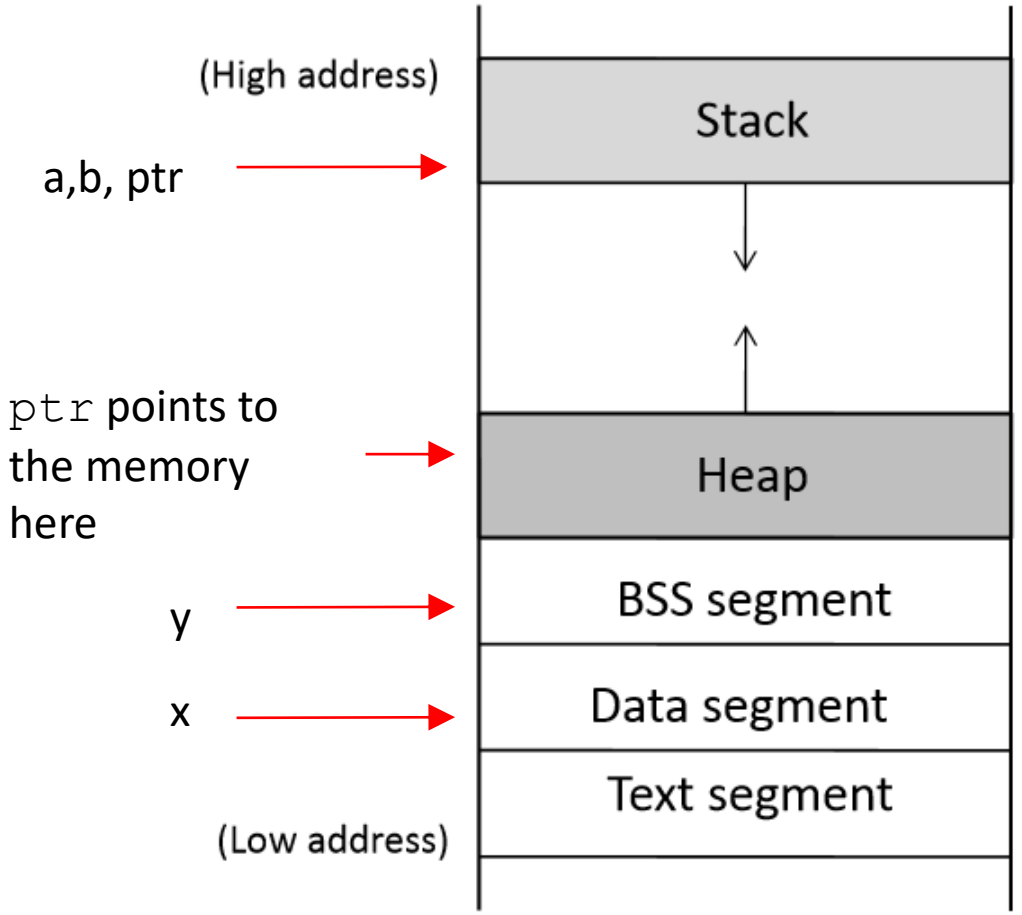
- Erro mais comum da lista Top25 CWE
  - https://cwe.mitre.org/data/definitions/787.html
- Erro "clássico" mas que nas suas variantes tem +10000 entradas na *Common Vulnerabilities and Exposures* (http://cve.mitre.org/cve/search_cve_list.html)
  - Ex: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3568

Consequências de um buffer overflow

- A sobreposição pode resultar em:
  - Ler informação privada do processo
    - http://heartbleed.com/
  - Instrução inválida
  - Endereço não existente
  - Violação de acesso
  - **Execução de código do atacante**

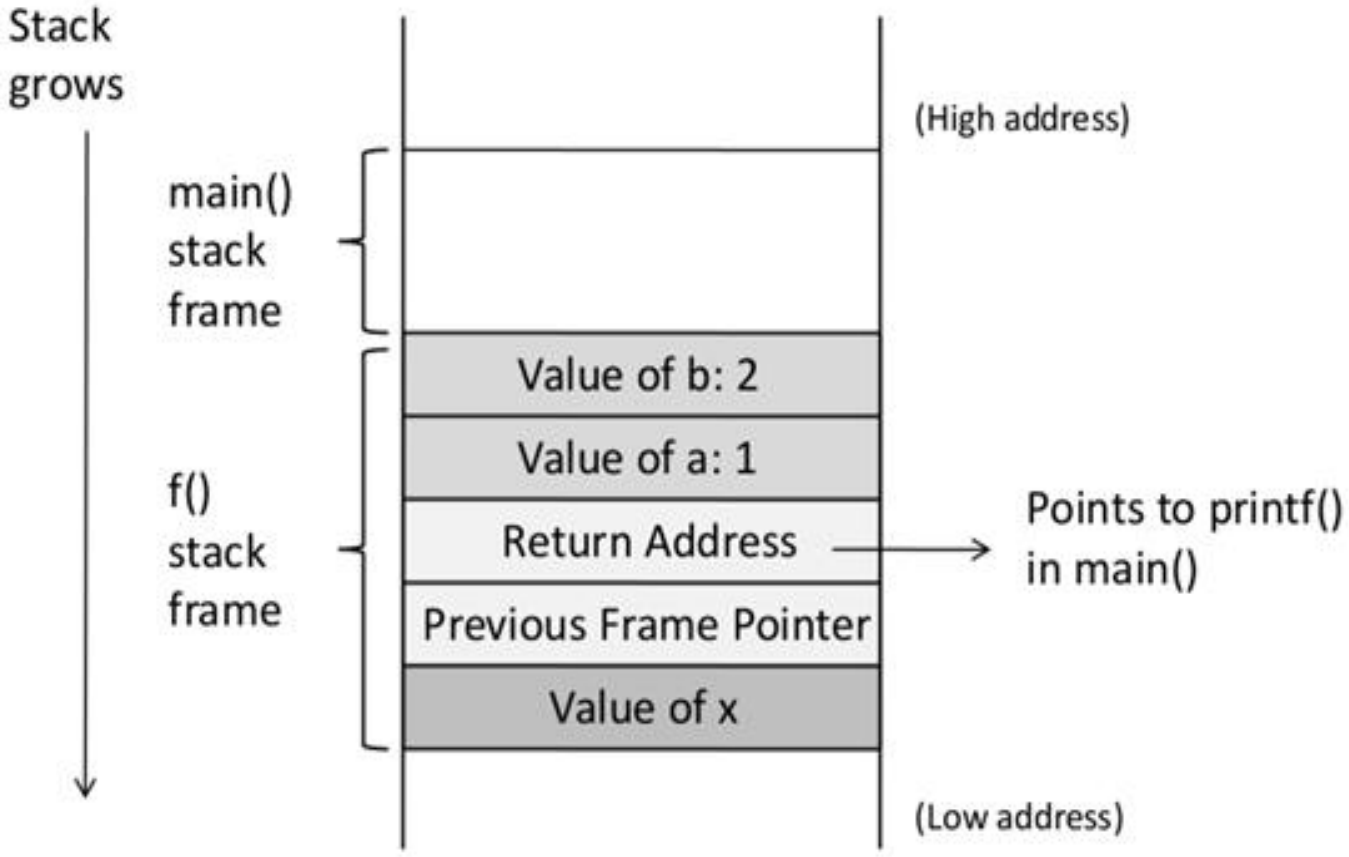
Organização da memória (e do *stack*)

```
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;
}
```

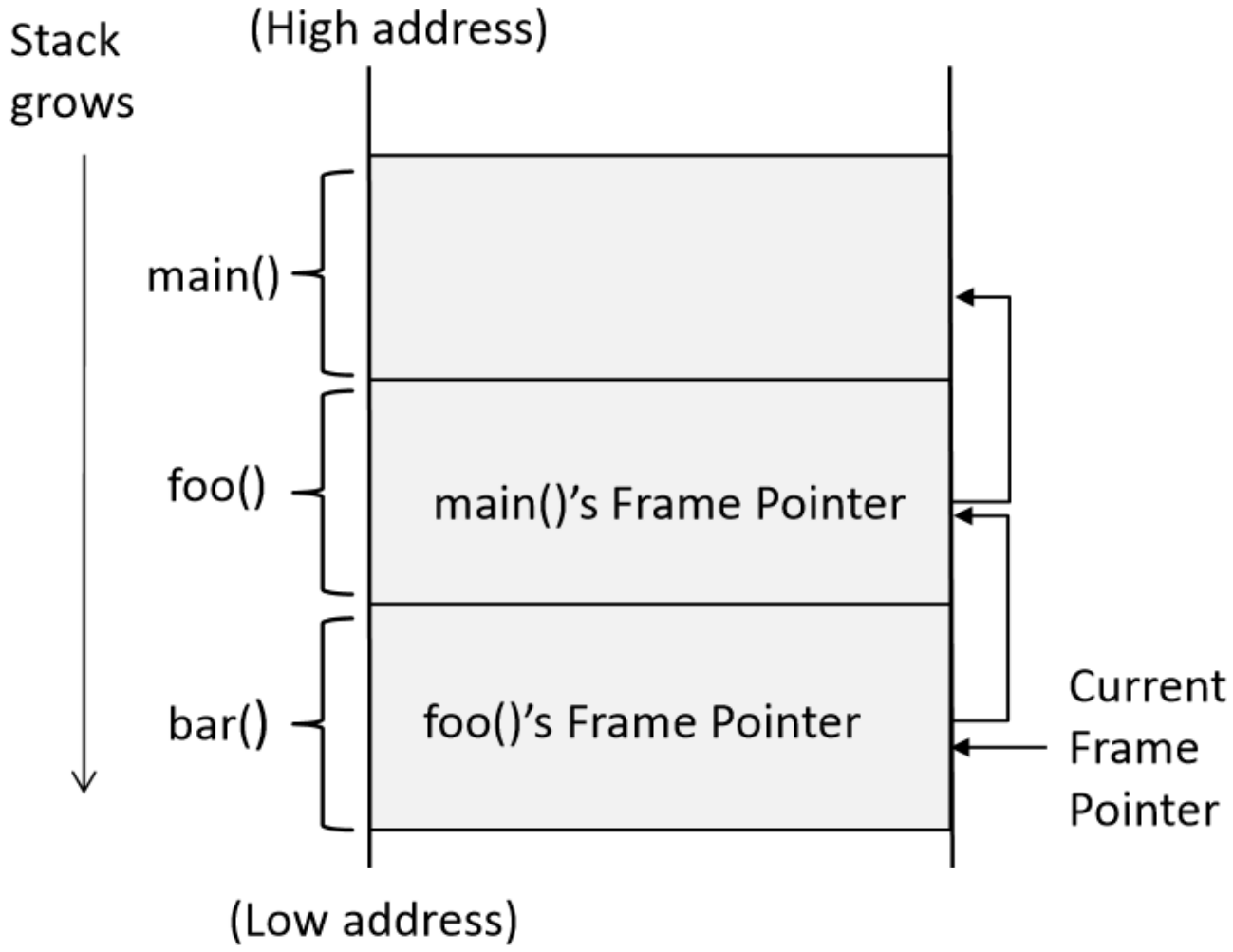


Chamadas a funções

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

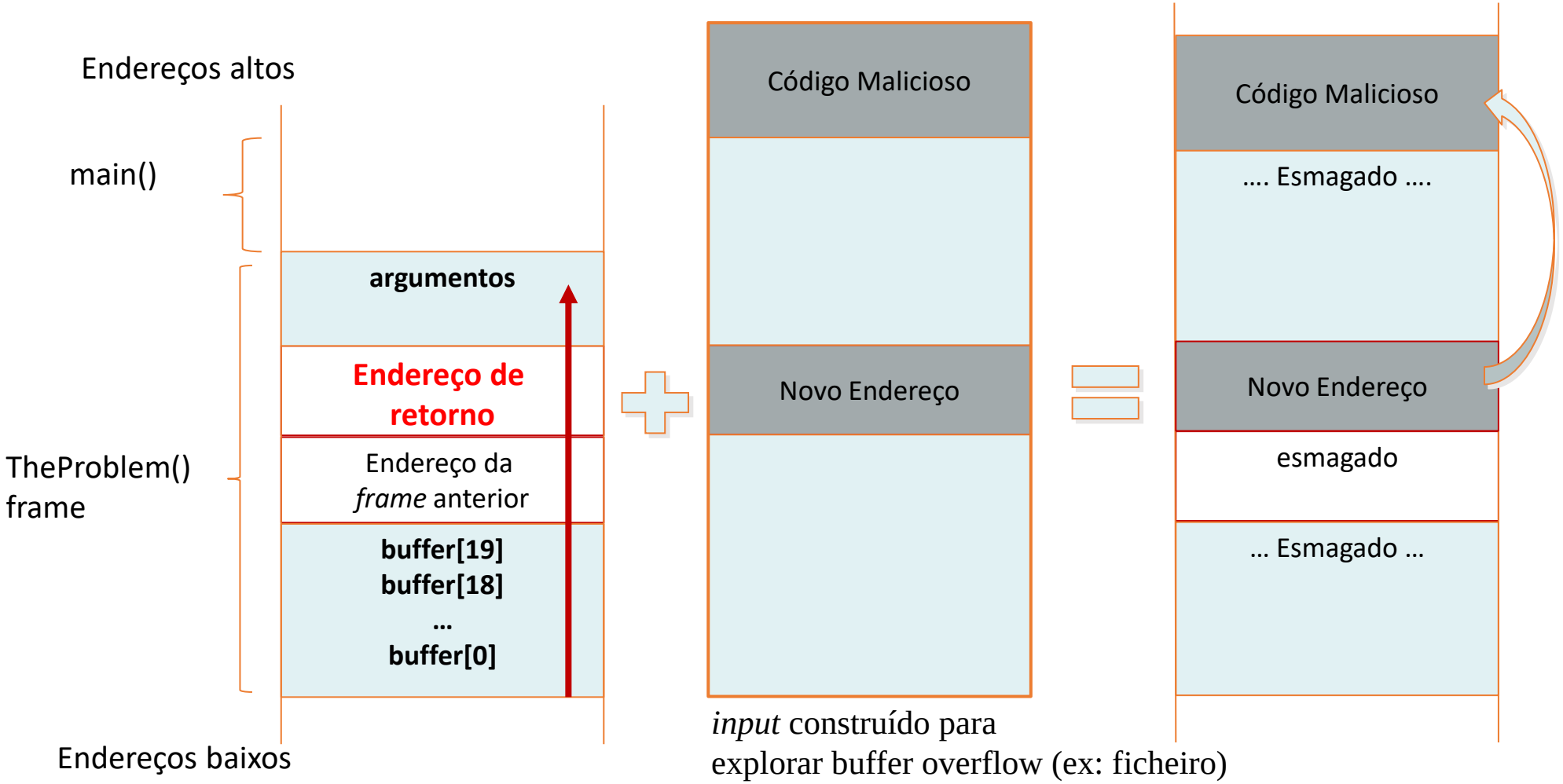


Organização do *stack* entre chamadas



Injeção de código

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



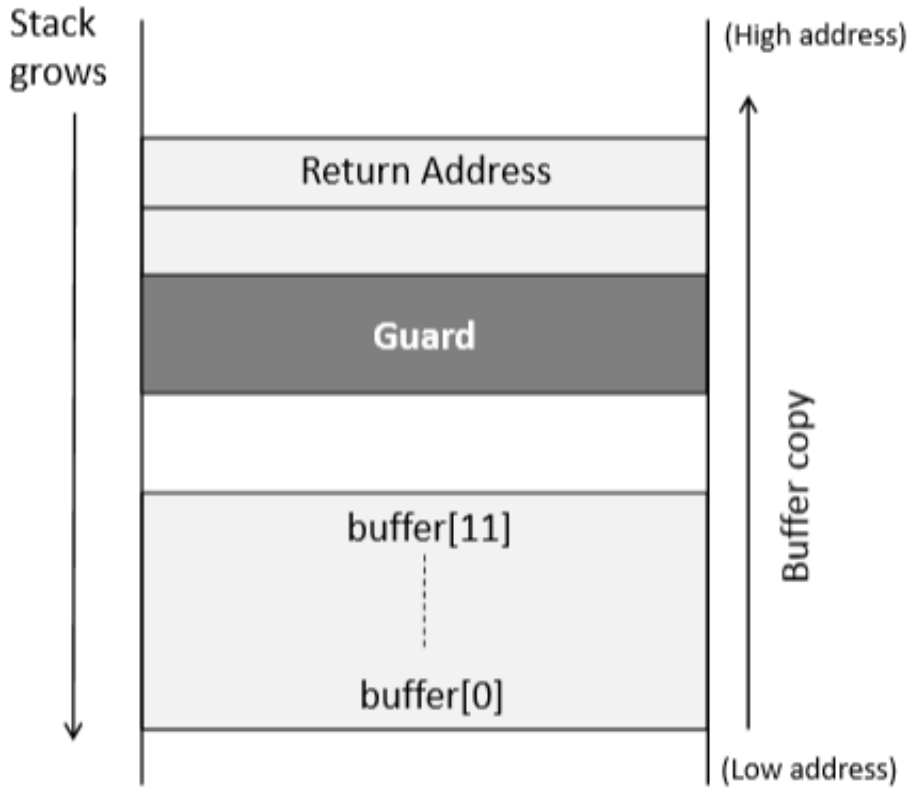
input construído para explorar buffer overflow (ex: ficheiro)

Mitigação da vulnerabilidade de buffer overflow

- Ao nível do código fonte
  - Usar funções mais seguras como strncpy ou strncat, que verificam a dimensão dos dados antes de efetuar a cópia
- Ao nível do compilador
  - “Canários” que verificam se o endereço de retorno foi sobreposto
- Ao nível do sistema operativo
  - Randomização do espaço de endereçamento (Address space layout randomization – ASLR)
- Ao nível do hardware
  - Stack não executável
  - Esta proteção pode ser contornada com uma variante conhecida como 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.

Análise de um ataque - 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

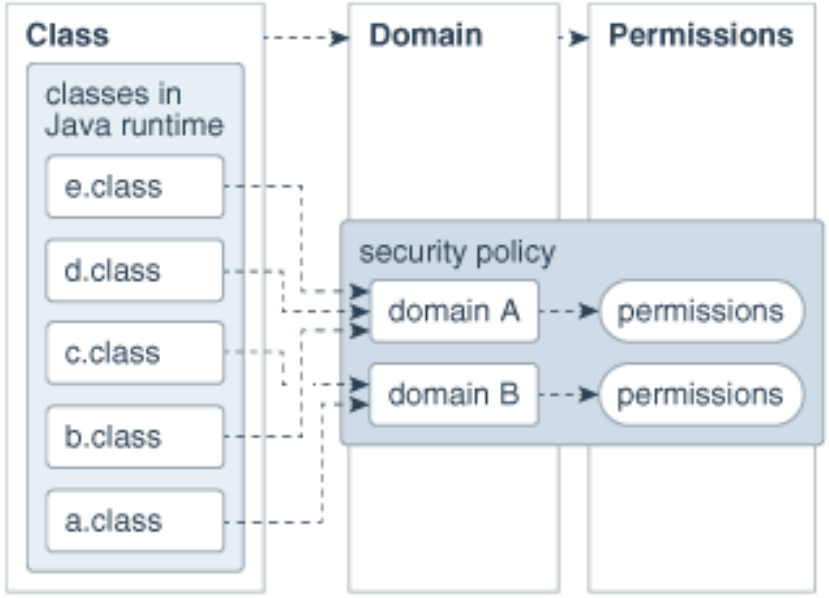
Runtimes e Sistemas operativos

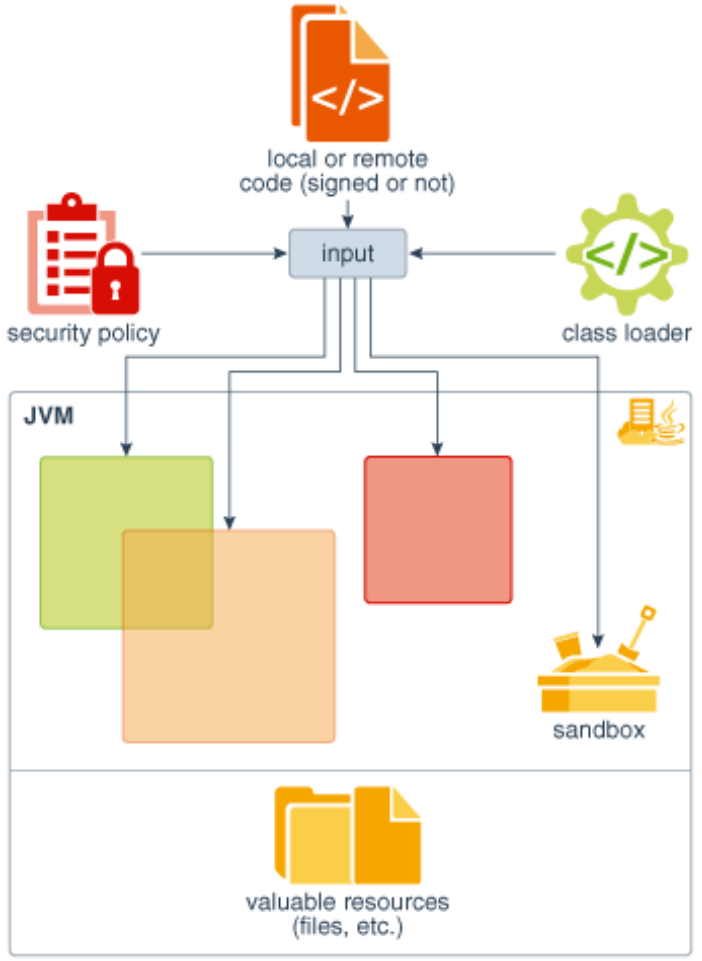
Segurança nas linguagens e ambientes de execução

- O impacto de vulnerabilidades em linguagens nativas como C e C++ está apenas limitado pelo sistema operativo
- Plataformas como Java e .NET têm um leque alargado de sistemas de segurança
  - Ao nível da linguagem: sistema de tipos, segurança de memória, segurança de fluxo
  - Verificador de byte codes
  - Permissões por domínio
  - Sandbox
  - Bibliotecas criptográficas

Permissões e domínios em Java

- Política de controlo de acesso determina permissões
- Permissões associadas a domínios
- Domínios



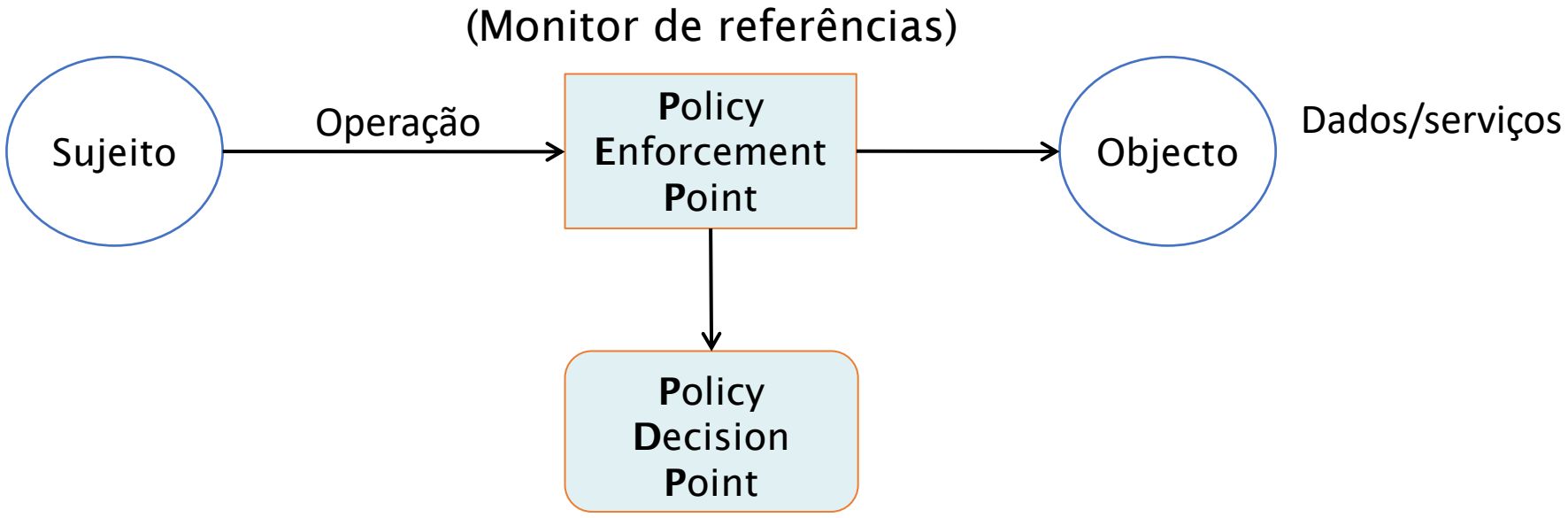


https://docs.oracle.com/en/java/javase/11/security/java-se-platform-security-architecture.html#GUID-D6C53B30-01F9-49F1-9F61-35815558422B

Segurança nos sistemas operativos

- A proteção de recursos é feita através de:
  - Separação
  - Mediação
- A separação evita que os processos acedam a zonas de memória de outros processos ou a instruções previligiadas
  - Memória virtual
  - Modo de execução de utilizador e de kernel
- A mediação permite o acesso controlado a recursos, tais como os objectos do sistema de ficheiros
  - Listas de controlo de acessos
  - Capacidades

Monitor de referência



- Propriedades do PEP:
  - Isolamento: não deve ser possível alterá-lo.
  - Completude: não deve ser possível contorná-lo.
  - Verificável: deve ser pequeno e estar confinado ao núcleo de segurança do sistema por forma a facilitar a verificação da sua correcção.

Elementos do sistema de controlo de acessos

- Política de segurança: define as regras do controlo de acessos
- Modelo de segurança: formalização da forma de aplicação das políticas de segurança
  - Permissões por grupo; lista de controlo de acessos; role-based access control
- Mecanismos de segurança: funções de baixo nível (software/hardware) que dão suporte à implementação de modelos e políticas de segurança

- PEP depende dos mecanismos de segurança
- PDP depende da política e modelo de segurança

Exemplo: sistemas Unix/Linux

- Utilizadores têm um identificador (*user id* – UID) e uma conta com esse ID
  - Effective user ID (EUID) – id com o qual um programa é executado
  - Real user ID (RUID) – id real do utilizador
  - Utilizador especial, *superuser (0)*, com direitos de administração
- Espaço de nomes do sistema de ficheiros é usado para aceder a pastas, ficheiros e dispositivos
- Controlo de acessos a ficheiros depende do EUID
- Cada objecto tem uma lista de acessos simples
  - UID do se dono e o GID do seu grupo
  - Permissões de acesso (r, w, x) a *owner*, *group* e *other*, ex: rwx r-- r--
- Sticky bits usados para controlo de acessos privilegiados
  - Programas designados *Set-uid* os quais correm com direitos de *superuser*

Conceito Set-UID

- **Permitir executar um programa com privilégios do *owner* do programa**
- Possibilita que utilizadores executem programas com privilégios temporariamente elevados
- Exemplo: programa `passwd` (precisa de acesso a ficheiro das *passwords*)
  - `$ ls -l /usr/bin/passwd`

```
-rwsr-xr-x 1 root root 41284 Sep 12  2012 /usr/bin/passwd
```

- Quando um programa normal executa, RUID = EUID
- Quando um programa Set-UID executa, RUID ≠ EUID. RUID é o id do utilizador, mas EUID corresponde ao ID do *owner*.
  - Se o dono de um programa é o *root*, o programa corre com privilégios de *root*.

Como funciona

- Um programa Set-UID é como outro qualquer, com *owner* root e o bit set-uid ativo na lista de permissões na lista de permissões

```
$ 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) ...
```

Exemplo de um programa set-uid

```
$ 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:::
...
```

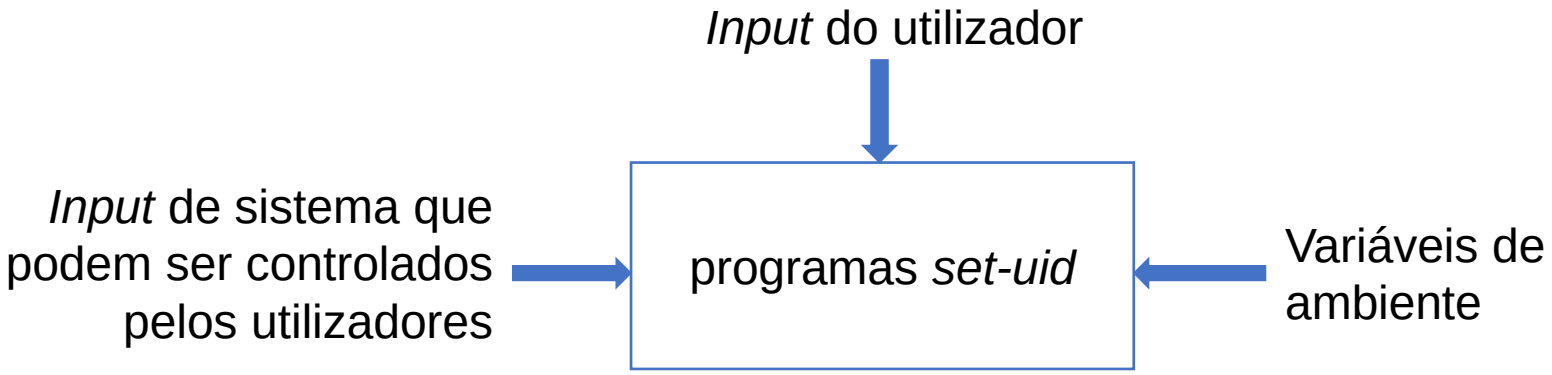
- Tornar-se um programa priveligiado

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

- É priveligiado, mas não priveligiado para root

Vulnerabilidades nos programas set-uid

- O conceito do *set-uid* é seguro, mas os programas podem terem falhas, expondo assim diferentes superfícies de ataque



- *User input* – por exemplo, *buffer overflow* é um problema bem conhecido, que se explorado num programa *set-uid* pode ter grande impacto
- *System input* – Escrita em zonas do sistema como /tmp pode ser controlado pelo utilizador com links simbólicos
- Variáveis de ambiente – o controlo de varáveis de ambiente, como o PATH, pode colocar em execução programas maliciosos

Estudo de caso – Injeção de comandos

Progamas que chamam comandos de sistema

- Invocar comandos externos dentro de um programa
  - O comando externo é escolhido pelo programa Set-UID
  - Os utilizadores não devem fornecer o comando (não é seguro)

Ataque:

- Muitas vezes os utilizadores são solicitados a fornecer dados de entrada para o comando.
- Se o comando não for invocado corretamente, os dados de entrada do utilizador podem ser transformados num nome de comando, introduzindo assim uma vulnerabilidade.

Invocação de programas: Versão insegura

```
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` é um program set-uid com *owner* root

Problema: Uma parte dos dados é interpretado como código (nome do comando)

Invocar commandos de forma segura - execve()

```
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)

Porque motivo é seguro?

Os dados e o código seguem "canais" diferentes; não existe forma dos dados dos utilizadores passarem a ser código

Invocar comandos de forma segura - 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!
```

Os dados são tratados como dados, não código

Caso prático: *Shellshock*

Vulnerabilidade na linha de comandos Linux

Contexto: Shell Functions

- O *Shell* é um interpretador de linha de comando em sistemas operativos
  - Fornece uma interface entre o utilizador e o sistema operativo
  - Diferentes tipos de shell: sh, bash, csh, zsh, windows powershell etc.
- Em particular o *bash shell* é um dos programas de *shell* mais populares em Linux
  - A vulnerabilidade do shellshock está relacionada com operações no shell.

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

Vulnerabilidades shellshock

- A vulnerabilidade chamada Shellshock ou bashdoor foi publicada em setembro de 2014 (CVE-2014-6271)
- Esta vulnerabilidade explora um erro no programa bash ao converter variáveis de ambiente para definição de função
- O bug encontrado existia no código-fonte do GNU bash desde 5 de agosto de 1989
- Após a identificação deste *bug*, vários outros *bugs* foram encontrados no shell bash amplamente utilizado
- Shellshock refere-se à família dos bugs de segurança encontrados no GNU bash

Vulnerabilidade Shellshock

- Processo pai pode passar uma função a um processo filho na forma de variável de ambiente
- Devido a um erro na lógica de *parsing*, o `bash` executa commandos contidos na variável

```
$ 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