# RESUMOS

# 1. Introdução

## 1.1. Razões para comprometer a máquina de um utilizador

- credenciais: roubar passwords bancárias, empresariais, jogos…
- *ransomware*: tipo de malware que restringe o acesso ao sistema infectado com uma espécie de bloqueio e cobra um resgate em criptomoedas para restabelecer o acesso, que torna praticamente impossível o rastreamento do criminoso que pode vir a receber o dinheiro.
- para utilizar o processador (por exemplo, para minerar *bitcoin*)
- para usurpar o endereço de rede e parecer um utilizador normal (e com isso, fazer *spam*, *denial of service* e geração de *clicks*)

## 1.2. Razões para comprometer servidores

- *data breaches* (ter acesso a várias credenciais de utilizadores, por exemplo números de cartão de crédito, de uma só vez)
- motivações políticas e geo-estratégicas
- para depois infetar as máquinas dos utilizadores
    - *supply-chain attacks*: infetar servidores que distribuem software e distribuir, por sua vez, malware
    - *web-server attacks*: infetar servidores web, que depois comprometem browsers

## 1.3. Segurança

- a propriedade  de um sistema que se comporta como esperado
- protection of computer systems and networks from information disclosure, theft of or damage to their hardware, software, or electronic data, as well as from disruption or misdirection of the services they provide
- é relativa, depende de quem a define
- muda consoante o contexto (até mesmo a terminologia usada muda)
- é defensiva, deve-se assumir que tudo pode acontecer

## 1.4. Atores ou participantes

- são entidades que intervêm no sistema
- pessoas, organizações, empresas, máquinas…
- muitas vezes deposita-se confiança em alguns atores/componentes intermediários que facilitam a interação entre dois atores, que ambos confiam no intermediário mas não um no outro (por exemplo, Trusted Third Party, Trusted Agent)
- considera-se "seguro" se pressuposto de confiança de verificar

## 1.5. Adversário ou atacante

- atores com intenção explícita de utilizar o sistema/recursos de forma indevida ou de impossibilitar a sua utilização
- nenhum sistema é seguro contra todos os adversários (ex: bomba nuclear, guerras, aliens)
- tipos de adversários:
    - script-kiddies
    - atacantes ocasionais que visam compreender o sistema
    - pessoas com intenção de causar danos/destruição
    - grupos organizados e tecnicamente sofisticados
    - competidores (espionagem industrial)
    - países/estados/governos

## 1.6. Notas

- Ao desenhar software, deve-se sempre assumir que vai haver atacantes. Isto obriga o desenvolvedor a pensar de forma diferente, a pensar como um atacante.
- Quem desenha o sistema não deve ser a pessoa responsável pela segurança desse sistema, pois já está demasiado focado nas funcionalidades.

# 2. “Segurança”

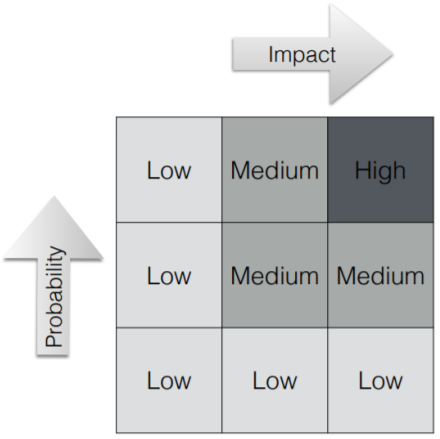
## 2.1. Dois modelos para pensar em segurança

### 2.1.1. Modelo binário

- Típico em criptografia e sistemas confiáveis
- Define formalmente capacidades e objetivos
- Limitações:
    - Não escala para sistemas complexos
    - Os modelos formais podem estar errados (por exemplo, side-channels)

### 2.1.2. Modelo de gestão de risco

- Típico em engenharia de software e segurança no mundo real
- Minimiza o risco em função das ameaças mais prováveis
- Optimizar o custo das medidas de segurança versus potenciais perdas
    - Faz-se uma análise de risco através de uma matriz que analisa dois fatores:
        - o potencial impacto
        - a probabilidade de materialização




    - A decisão de mitigar algum risco é um processo de gestão de custos que se baseia nessa matriz. Quanto custaria um possível ataque e quanto custa implementar uma defesa. Por vezes sai mais barato simplesmente aceitar o risco de que o ataque ocorra.

- **Ativo**
    - É um recurso que detém valor para um ator do sistema.
    - Exemplos:
        - informação
        - reputação/imagem
        - dinheiro/recurso com valor monetário intrínseco
        - infraestrutura
        - etc
- O objetivo em segurança é que os ativos não percam o seu valor. A perda desse valor pode acontecer por quebra de (CIA):
    - **Confidencialidade**: garantia de que a informação é acessível apenas aqueles que têm autorização para a ver.
    - **Integridade**: garantia de que a informação não foi alterada sem autorização.
    - **Disponibilidade**: garantia de que a informação está acessível e é alterável sempre que necessário.
- Limitações:
    - A análise de risco pode estar errada
    - Uma ameaça mal classificada pode deitar tudo por terra

## 2.2. Vulnerabilidades

- São falhas que estão acessíveis a um adversário que poderá ter a capacidade de as explorar.
- Têm geralmente origem em erros de concepção:
    - Software de má qualidade
    - Análise de requisitos desadequada
    - Configurações erradas
    - Utilização errada

- **Vulnerability reporting:** gestão de novas vulnerabilidades que aparecem todos os dias, onde toda a comunidade trabalha em conjunto para identificar, classificar, divulgar, detectar e eliminar vulnerabilidades
- **Zero-Day Vulnerabilities:** vulnerabilidades que nunca foram reportadas ou que estão latentes

## 2.3. Ataques

- Ocorrem quando alguém tenta explorar uma vulnerabilidade.
- Tipos de ataques:
    - **Passivos**: monitorização de comunicações, sistemas ou fluxo de informações, como por exemplo, eavesdropping, onde o atacante monitoriza sem autorização uma comunicação, podendo roubar dados e informações que poderão ser usadas posteriormente
    - **Activos**: interação com o sistema, onde o atacante tenta adivinhar passwords ou faz engenharia social
    - **Denial-of-Service**: perturbar a disponibilidade de um sistema, como por exemplo, enviar *spam* com o objetivo de encher a *queue* do email e, por sua vez, abrandar um servidor de email.
- Quando um ataque é bem sucedido diz-se que um sistema foi **comprometido**.
- **Método/exploit**: forma de explorar a vulnerabilidade.

## 2.4. Ameaças

- Causas possíveis para um incidente que possa trazer consequências negativas para um sistema, pessoa ou organização.
- Podem ser pessoas, eventos naturais, falhas acidentais ou causadas intencionalmente.
- Definem-se pelo seu tipo e origem
    - Tipo de ataque: dano físico, perda de serviços, quebra de protecção de informação, falhas técnicas, abuso de funcionalidades
    - Tipo de atacante: ação deliberada (implica definir a fonte), ação negligente (implica definir a fonte), acidente (implica definir o componente afetado), evento ambiental.
- Estão identificadas e classificadas quanto à relevância.

## 2.5. Mecanismo de segurança

- É um método, ferramenta ou procedimento que permite implantar uma (parte de uma) política de segurança.
- Exemplos:
    - Mecanismos de identificação/autenticação (e.g., biometria, one-time passwords)
    - Mecanismos de controlo de acessos (e.g., RBAC)
    - Criptografia (e.g., cifras, MACs, assinaturas)
    - Controlos físicos (e.g., cofres, torniquetes)
    - Auditorias (e.g., penetration testing)

## 2.6. Política de segurança

- Determina um conjunto de processos/mecanismos que devem ser seguidos para garantir segurança num determinado modelo de ameaças.
- Exemplos:
    - Segurança física
    - Controlo de acessos
    - Política de passwords
    - Política de email
    - Política de acesso remoto

## 2.7. Modelo de confiança

- Modelo que define o conjunto de atores em que confiamos e que temos como pressuposto garantido que não são possíveis adversários.
- Um sistema é confiável se o sistema faz exatamente (e apenas) aquilo que foi especificado.
- Exemplos de sistemas confiáveis:
    - Sistemas que não transmitem a nossa informação sensível para o exterior sem autorização.

- Sistemas que garantem que as nossas comunicações são estabelecidas com as entidades com quem queremos comunicar (e.g., servidores Google)
- Sistemas que cifram toda a informação em disco e limpam a memória quando fazemos shutdown.

## 2.8. Processo

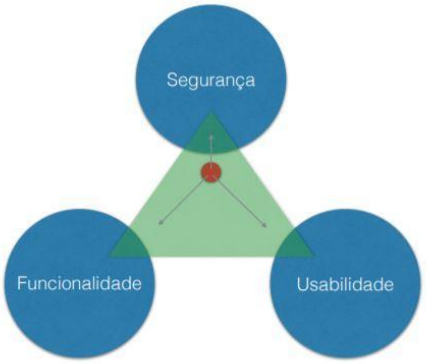
- **Confiabilidade**: o objetivo é garantir que a segurança não seja um "salto de fé", ou seja, definir o que é a segurança e garantir uma forma de a conseguir. Geralemente, este processo define-se da seguinte forma:

1. **Definir o problema**
    a. Análise de requisitos de segurança
    b. Definir o modelo de ameaças
        i. Quais são os ativos e objetivos de segurança?
        ii. O que queremos proteger e de quem?
        iii. Quem são os nossos potenciais adversários?
        iv. Quais as suas motivações, capacidades e tipo de acesso?
        v. Que tipos de ataques temos de precaver?
        vi. Que tipos de ataque podemos descartar/ignorar?
    c. Matriz de gestão de risco
2. **Definir o modelo de confiança**
    a. Em que componentes/atores confiamos?
    b. O que fazem eles?
    c. O que nos é dado como ponto de partida/âncora?
3. **Definir a solução**
    a. Conceber as políticas de segurança que se aplicam ao sistema
    b. Identificar os mecanismos de segurança necessários e suficientes
4. **Validar e justificar a solução**
    a. Validar a adequação dos modelos à aplicação concreta em análise
    b. Demonstrar (formal ou informalmente) que os mecanismos de segurança subjacentes são suficientes, em conjunto com os pressupostos de confiança assumidos, para implantar as políticas de segurança
5. **Auditoria de segurança**
    a. Testes de penetração. Ethical hacking.
    b. Simulação de um ataque de procura de vulnerabilidades que poderiam ser exploradas

i. Black box: semelhante a um ataque real
ii. White box: com conhecimento privilegiado

## 2.9. Notas

- A segurança é um processo contínuo e cíclico em que um adversário inventa um novo ataque e o defensor cria uma nova defesa.
- O melhor a que se pode ambicionar é um equilíbrio ao longo do tempo.




- Existem vários tipos de segurança e em sistemas complexos a segurança é tratada de forma estruturada
    - Segurança de redes
    - Segurança de sistemas/computadores
    - Segurança de programas
    - Segurança física
    - Privacidade

# 3. Usurpação de controlo

- Ocorre quando um input externo leva o programa a quebrar a convenção:
    - alterando a sequência de instruções que é executada
    - substituindo a sequência de instruções esperada por uma sequência de instruções controlada pelo atacante

## 3.1. Criação de um exploit

- Domínio de programação baixo nível (assembly) e debugging de código binário
- O que se faz?
    - **Buffer Overflow -** Injetar código malicioso na memória (shellcode) e alterar controlo de execução para saltar para essa zona.
        - Compreender as causas do overflow e como o desencadear de forma controlada
        - Prever endereços a alterar
        - Prever localização de shellcode
        - Evitar crash antes de tomada de controlo
    - **Bibliotecas** - Se não podemos injetar o nosso código, temos de usar código que já está em memória executável.

## 3.2. Descoberta de vulnerabilidades

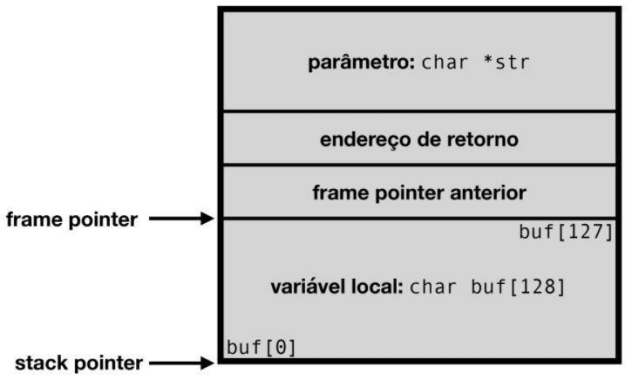
- Geralmente, as vulnerabilidades são erros na gestão de memória por parte dos programadores.
- O primeiro sinal é um crash do programa. Para procurar por vulnerabilidades utiliza-se um "fuzzer" para fornecer inputs cegamente a um programa. Se o programa crashar, investiga-se porquê.
- ***Fuzzing*** *or* ***fuzz*** *testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.*
- **Ethical hacking**: É crucial haver quem procure estas vulnerabilidades para termos sistemas mais seguros. É importante ter muito cuidado com a forma como se divulgam essas vulnerabilidades. Embora um hacker seja bem intencionado, o facto

de estar a divulgar a vulnerabilidade sem cuidados (como por exemplo, não avisar a empresa desse bug) pode criar problemas tanto ao hacker, que pode ser processado, como à empresa que poderá depois ser atacada com base nessa vulnerabilidade por hackers mal intencionados. Os sistemas de bug bounty e vulnerability reporting são desenhados para evitar que estas "zero-day vulnerabilities" detectadas causem danos, tanto ao hacker bem intencionado como às empresas.

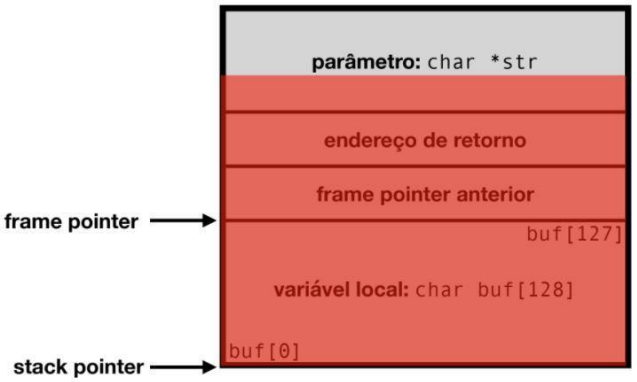
## 3.3. Buffer overflow na stack (stack smashing)

```
void func(char *str) {
   char buf[128];
   strcpy(buf, str);
   usar(buf);
}
```

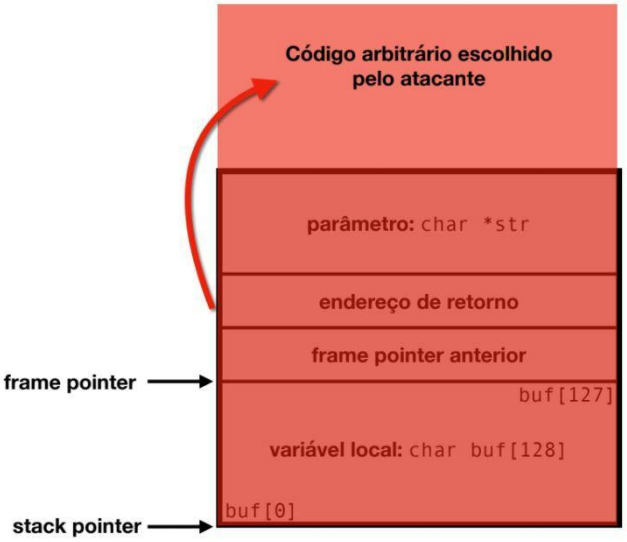
- Quando entramos nesta função (caixa azul), a stack frame tem o seguinte aspeto:




- Se **str** tiver tamanho maior que 128, a função escreve para posições crescentes na memória:



- O atacante pode preencher stack com código, substituir endereço de retorno e executar código arbitrário.




- O código arbitrário escolhido pelo atacante é geralmente um "Shell code".
- **Shell code**: sequência de instruções em código binário que executa comandos shell. Exemplo:

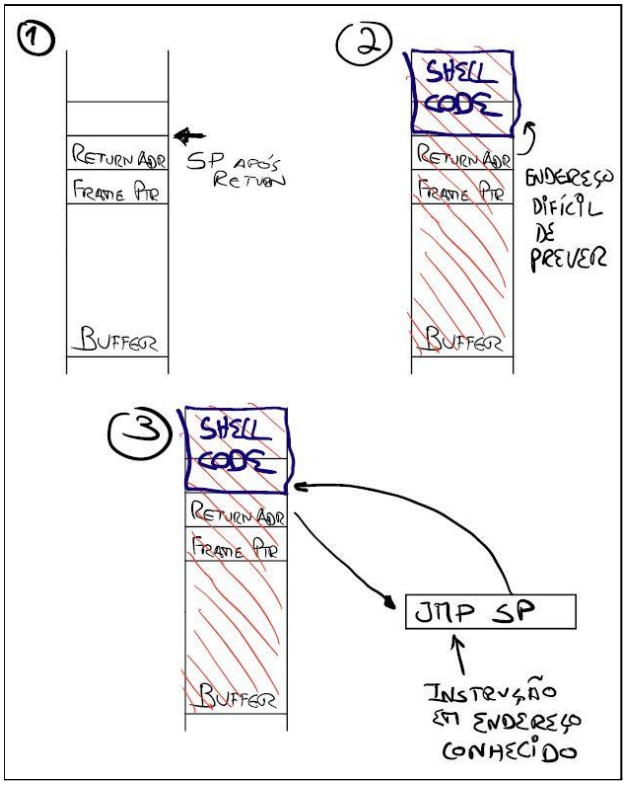
```
char shellcode[] = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70"
                   "\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61"
                   "\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52"
                   "\x51\x53\x89\xe1\xcd\x80";
```

Esta shellcode corresponde ao seguinte código-máquina:

```
8048054:      6a 0b                push   $0xb
8048056:      58                   pop    %eax
8048057:      99                   cltd
8048058:      52                   push   %edx
8048059:      66 68 2d 70          pushw  $0x702d
804805d:      89 e1                mov    %esp,%ecx
804805f:      52                   push   %edx
8048060:      6a 68                push   $0x68
8048062:      68 2f 62 61 73       push   $0x7361622f
8048067:      68 2f 62 69 6e       push   $0x6e69622f
804806c:      89 e3                mov    %esp,%ebx
804806e:      52                   push   %edx
804806f:      51                   push   %ecx
8048070:      53                   push   %ebx
8048071:      89 e1                mov    %esp,%ecx
8048073:      cd 80                int    $0x80
```

- **Nota**: shellcode não pode conter "0x00" uma vez que este caracter denota o fim de uma string e impede o atacante de copiar a shellcode para um buffer com sucesso. Portanto, os atacantes têm de encontrar shellcodes que não contenham "0x00".

- **Como evitar ter de adivinhar endereço específico onde reside shellcode?**




**1. Disposição na stack (buffer controlado por atacante)**. Após a execução do *return*, *sp* aponta para endereço mais acima como na imagem. Este endereço pode ser difícil de prever, mas a plataforma ajusta-o automaticamente.

**2. Buffer overflow normal.** É preciso explicitamente colocar no sítio do *return address* um endereço que aponta para a shellcode (este endereço pode ser difícil de prever e os atacantes tentam adivinhá-lo). Nota: apesar da shellcode ser código do atacante, ele não sabe onde é que este se encontra na memória.

**3. Truque que usa *jmp sp*.** Este truque pode ser usado pelos atacantes para evitar ter de adivinhar o endereço. Se o opcode *jmp sp* estiver num endereço conhecido X, reescrevemos o *return address* com X. Quando a função retornar salta para X, executa-se *jmp sp*, e como sp está a apontar para a shellcode que está na stack, obtemos o mesmo efeito.

- **Outros exemplos de stack smashing:**
  - **fingerid**

```
main(argc, argv)
    char *argv[];
{
    register char *sp;
    char line[512];
    struct sockaddr_in sin;
    int i, p[2], pid, status;
    FILE *fp;
    char *av[4];

    i = sizeof (sin);
    if (getpeername(0, &sin, &i) < 0)
        fatal(argv[0], "getpeername");
    line[0] = '\0';
    gets(line);
```

Finger Daemon Buffer Overflow

Vulnerability Description

Brief Description: The *finger*(1) daemon is vulnerable to a buffer overrun attack, which allows a network entity to connect to the *fingerd*(8) port and get a *root* shell.

Detailed Description: *Fingerd* is a daemon that responds to requests for a listing of current users, or specific information about a particular user. It reads its input from the network, and sends its output to the network. On many systems, it ran as the *superuser* or some other privileged user. The daemon, *fingerd* uses *gets*(3) to read the data from the client. As *gets* does no bounds checking on its argument, which is an array of 512 bytes and is allocated on the stack, a longer input message will overwrite the end of the stack, changing the return address. If the appropriate code is loaded into the buffer, that code can be executed with the privileges of the *fingerd* daemon.

Component(s): finger, fingerd

Version(s): Versions before Nov. 6, 1989.

Operating System(s): All flavors of the UNIX operating system.

- **libpng**

```
if (!(png_ptr->mode & PNG_HAVE_PLTE))
{
   /* Should be an error, but we can cope with it */
   png_warning(png_ptr, "Missing PLTE before tRNS");
}
else if (length > (png_uint_32)png_ptr->num_palette)
{
   png_warning(png_ptr, "Incorrect tRNS chunk length");
   png_crc_finish(png_ptr, length);
   return;
}
```

We can see, if the first warning condition is hit, the length check is missed due to the use of an "else if".

- **realpath**

```
/*
 * Join the two strings together, ensuring that the right thing
 * happens if the last component is empty, or the dirname is root.
 */
if (resolved[0] == '/' && resolved[1] == '\0')
        rootd = 1;
else
        rootd = 0;

if (*wbuf) {
        if (strlen(resolved) + strlen(wbuf) + rootd + 1 > MAXPATHLEN) {
                errno = ENAMETOOLONG;
                goto err1;
        }
        if (rootd == 0)
                (void)strcat(resolved, "/");
```

Details:
========

An off-by-one bug exists in fb_realpath() function. An overflow occurs when the length of a constructed path is equal to the MAXPATHLEN+1 characters while the size of the buffer is MAXPATHLEN characters only. The overflowed buffer lies on the stack.

The bug results from misuse of rootd variable in the calculation of length of a concatenated string:

- **Por que é que stack smashing acontece?**
  - Manipulação de strings/buffers usando bibliotecas tipo libc
  - Muitas funções são simplesmente *unsafe*, ou seja, não garantem escrita limitada região de memória pré-definida (por exemplo, strcpy, strcat, gets, scanf) ou escrevem em espaço pré-definido mas não garante que string está corretamente terminada (por exemplo, strncpy)
  - Ou simplesmente código implementado de raiz com os mesmos problemas: assume-se que o input vem de fonte confiável.

## 3.4. Buffer overflow na heap

### 3.4.1. Exception handlers

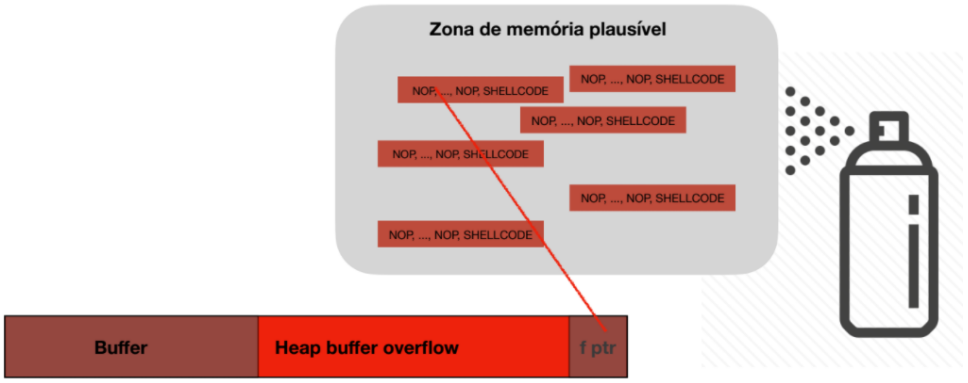
- Em linguagens com tratamento de exceções (e.g., C++), a stack frame de uma função inclui uma lista ligada/tabela de apontadores para as diferentes rotinas de tratamento de exceções.
- Se existe um buffer overflow na heap, pode escrever-se por cima de um desses endereços
- Se a exceção ocorrer, a execução prosseguirá para o endereço da nossa escolha.

### 3.4.2. Generalização "smashing the heap"

- As operações de **malloc/free** são utilizadas para alterar outras partes da memória: o **free** apaga um elemento de uma lista duplamente ligada reescrevendo apontadores, manipulado para escrever numa parte da memória completamente diferente, ou seja, reescreve um apontador para uma função com o endereço de código malicioso.

### 3.4.3. "Heap spraying"

- Servidor web malicioso pretende ganhar controlo da máquina de clientes para instalar malware, trojan, etc.
- Da parte do cliente, o comportamento do browser depende de muitos fatores, por exemplo, o posicionamento na memória de código injetado é difícil/impossível de prever.
- Os adversários fazem então "heap spraying", ou seja, executam código JavaScript na máquina do cliente que inunda a memória com cópias de shellcode. O buffer overflow coloca o apontador para uma função algures na zona alvo aumentando a probabilidade de obter controlo.
- Isto permite evitar ter de prever o endereço exato do código malicioso, criando muitas cópias em memória.




### 3.4.4. Use after free

- Em linguagens orientadas a objetos (C++), se libertarmos a memória destes objetos (através de um *free* ou destrutores), o atacante pode conseguir usar essa memória recentemente libertada. O adversário poderá conseguir preencher essa zona de memória com endereços à sua escolha, geralmente para execução de shellcode.

- Consideremos um programa que tem um erro de implementação: ocorre free(ptr) e depois um acesso a ptr.

- A sequência de eventos típica é:
    - ocorre o free(ptr) e a memória correspondente é devolvida ao sistema
    - o programa reutiliza essa memória para armazenar um input fornecido pelo atacante (por exemplo, um ficheiro)
    - o atacante definiu o ficheiro para reescrever &ptr com um endereço &malicioso à sua escolha
    - o acesso errado utiliza &malicioso permitindo a tomada de controlo

## 3.5. Outros tipos de overflow

### 3.5.1. Overflow de inteiros

- Overflow de inteiros acontece quando há perda de informação devido à representação de inteiros do processador.
- Formas de causar um overflow de inteiros:
    - truncatura por passagem para tipo mais pequeno

```
int i = 0x12345678;
short s = i;
char c = i;
```

    - aritmética

```
char sum = 0xFF + 0x1
```

Se estivermos a calcular o tamanho da região de memória podemos alocar 0 bytes em vez de 256 bytes!

    - comparação que esquece a representação com sinal

```
if (x < 100) // segue código que usa x
```

E se x for `0xffffffff`? Este valor representa -1 em palavras de 32 bits!

Mais uma vez podemos alocar uma região de memória demasiado pequena (ou vazia)

Também pode haver problemas quando comparamos valores sem sinal e com sinal:

```
if (size < sizeof(x))
  p = malloc(size); // size < 0
```

- O adversário tira partido do overflow por inteiros:
    - causando um crash (denial-of-service)
    - fazendo com que seja alocado 0 bytes para a variável do programa. Isto activa uma vulnerabilidade de buffer overflow clássico (tipicamente na heap) que pode ser explorado para tomada de controlo. Ou seja, como não é alocado bytes para o buffer mas este ainda existe, o atacante acaba por colocar ele a shellcode no buffer.

### 3.5.2. Strings de formatação

- O seguinte programa está vulnerável:

```
#include <stdio.h>
int main(int argc, char * argv[])
{
  printf("Your argument is:\n");
  // Does not specify a format string, allowing the user to supply one
  printf(argv[1]);
}
```

- O mesmo programa protegido seria incluir sempre *format strings* (%s) num *printf*:

```
#include <stdio.h>
int main(int argc, char * argv[])
{
  printf("Your argument is:\n");
  // Supplies a format string
  printf("%s",argv[1]);
}
```

- As *format strings* (%s) escrevem na memória. Quando o programador não coloca essa *format string*, o input é que controlará o formato do output. Assim, o atacante apenas precisa de construir um input que escreverá em localizações arbitrárias da memória (idealmente, escreverá o endereço de memória onde reside a *shellcode*). Este input inclui a *format string* "%n" que especifica que a função *printf* deve escrever o número de bytes do output para o endereço de memória indicado no primeiro argumento da função (no exemplo acima, argv[1]).

## 3.6. Bibliotecas

- A biblioteca ***libc*** é utilizada por (quase) todos os programas
- Contém funções úteis:

- ***system***: correr comando shell
- ***mprotect***: alterar permissões de memória

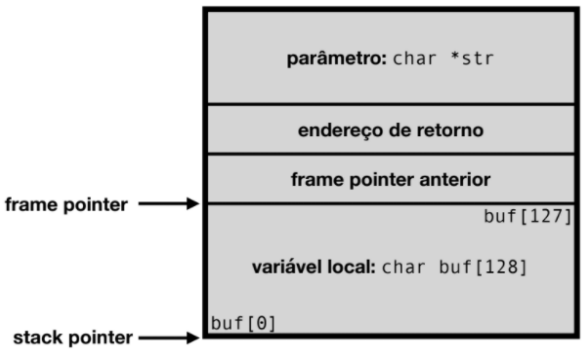
- Utilizam-se as técnicas de buffer overflow, mas o endereço de retorno será o de uma função da biblioteca. Por exemplo, se quisermos usar a função *system* ou *mprotect* temos de configurar a stack de acordo com o que essas funções necessitam.

- **Como funciona a stack?**

```
int main() {
    func("String \n");
    return 0;
}
```

```
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    usar(buf);
}
```

1. A função que faz a chamada (o *main*) armazena parâmetros e endereço de retorno na stack. Assim:




2. Quando se entra na função *func* é armazenado o frame pointer anterior e criado espaço para variáveis locais.

3. Cada função, neste caso a função *main*, limpa o espaço que cria na stack. O processador depois utiliza o endereço de retorno automaticamente, e a função que fez a chamada (*main*) limpa os parâmetros.

   - **É a função *main* (a função que chama *func*) que é responsável por colocar os parâmetros na stack e limpar depois a stack.**

### 3.6.1. Como chamar uma função da biblioteca (*system* ou *mprotect)*?

- Os dados injetados têm de:
  - substituir o endereço de retorno da função actual por &system ou &mprotect

    - configurar a stack para uma entrada correta na função *system* ou na função *mprotect*: parâmetros e endereço de retorno

- Com *system*, para evitar um crash
    - A string da shell deve ser colocada para cima na stack para dar espaço aos returns, incluindo o return exit(0).
    - Retornar para função que encerra programa sem erro, por exemplo, a instrução exit(0)
- Com *mprotect*, os parâmetros a passar geralmente contém '\0', o que reduz as possibilidades de exploit (não funciona com *strcpy*, mas talvez funcione com *memcpy*).

### 3.6.2. Como chamar várias funções em sequência da biblioteca?

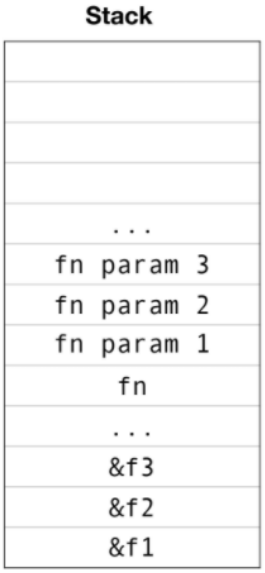
- É necessário construir toda uma estrutura na stack que esteja preparada para a sequência de funções. A função vulnerável retorna para a primeira função. Os parâmetros e endereço de retorno desta função conterão a segunda função a chamar. E assim sucessivamente.
- **Casos simples:**


Funções que não recebem parâmetros.

Assim, a função que as chama apenas armazena na stack o seu endereço de retorno.

Se quisermos que N funções sem parâmetros sejam executadas em sequência, basta colocar os seus endereços por ordem na stack.

A função vulnerável retorna para f1, que retorna para f2, etc.




Depois de uma sequência de qualquer tamanho de funções que não recebem parâmetros, podemos chamar **uma** função que recebe parâmetros. E terminar chamando uma função que não recebe parâmetros.

- **Casos complexos:**
    - É a função que chama uma outra função que é responsável por limpar a stack. Assim, quando a função vulnerável retorna para a primeira função e colocamos na stack os seus parâmetros, quando retornamos para a segunda função, esta encontrará "lixo" na stack da primeira função.
    - Se queremos chamar uma sequência arbitrária de funções (quer tenham ou não parâmetros) precisamos de criar outras funções:
        - cujo endereços são colocados na stack entre duas funções (por exemplo entre f1 e f2)
        - esses endereços são pedaços de código que retiram da stack o número de bytes necessário (por pop ou alteração direta do stack pointer), ou seja, que "limpam" a stack e que retornam para a função seguinte.
    - Como são feitas estas funções "intermédias"?
        - Usando a técnica de **return oriented programming**: consiste em identificar sequências de código úteis que já estão na memória (geralmente terminando em *ret*, *jmp* ou *call*). Chama-se a estas sequências "**gadgets**". Depois, esses gadgets são colocados juntos formando código malicioso. Assim, o atacante não precisa de injetar código, apenas precisa de encontrar várias instruções em memórias que juntas formam o código o que ele pretende .

## 3.7. Medidas de proteção

- Em geral, as medidas de proteção implicam um *overhead* significativo, afetando por vezes a performance de um programa.
- Apesar de todas estas medidas e mitigações, os ataques continuam a aparecer e são cada vez mais elaborados.
- **Medidas de proteção na própria plataforma**: impedir a execução de código malicioso (DEP, ASLR, KASLR, KARL, PIO)
- **Medidas de proteção no executável:** detectar tentativa de highjacking (Stack Canaries, Memory tagging, CFI)
- **Data Execution Prevention (DEP)** ou **Executable Space Protection (W^X)**
    - É um recurso de segurança incluído em sistemas modernos, como o Windows.
    - Uma página de memória executável não pode ser escrita (W)
    - Uma página de memória que pode ser escrita não pode ser executável (X)
    - Exceções a estas duas regras nos cenários de **Just-in-Time compilation**: forma de executar código compilando o programa durante a sua execução (run-time) em vez de compilar o programa antes da sua execução.
    - Pode ser implementada em hardware ou emulada em software
- **Address Space Layout Randomization (ASLR)**
    - A localização em memória de partes críticas da memória de um programa é "baralhada" aleatoriamente em cada execução, ou seja, a localização de código e dados em memória é gerada de forma aleatória em cada execução.
    - Esta medida de proteção impede que um atacante salte para endereços conhecidos tentando prever os endereços de código útil.
- **Kernel Address Space Layout Randomization (KASLR):**
    - Para além da técnica de ASLR, KASLR também randomiza a localização do código do kernel quando o sistema inicia (boot).
- **Kernel Address Randomized Link (KARL):**
    - O próprio código do kernel é baralhado (e não a localização em memória).

- **Position-independent executables (PIO)**
    - PIO é um conjunto de código máquina, que se colocado noutro lado da memória que não a original, executará bem independentemente do seu endereço absoluto. Ou seja, o código funcionará independentemente de onde se encontre na memória.
    - Esta medida de proteção torna muito mais difícil que se consiga fazer return oriented programming, pois o endereço dos vários pedaços de memória pode mudar consoante a máquina onde se encontra.
- **Stack Canaries**
    - O objetivo é prevenir a injeção de código detectando modificações à stack.
    - A ideia é introduzir "canários" gerados dinamicamente entre variáveis locais (os dados que o atacante poderia manipular) e os valores do frame pointer e do endereço de retorno guardados na stack. Se o atacante tentar algum buffer overflow, ele inevitavelmente escreverá por cima do canário (é a única forma de chegar ao endereço de retorno). Assim, o canário detecta esta tentativa e crasha o programa antes de utilizar o endereço de retorno.
    - Exemplo:

```
void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```


    - Tipos de canários:
        - **Canário aleatório**: No início da execução o programa escolhe um array de bytes aleatório. Esses bytes são colocados em todas as stack frames. Antes de retornar de uma função verifica-se a sua integridade. Se houver alguma alteração a esses bytes (o que acontece se se tentar escrever por cima deles) o programa termina.

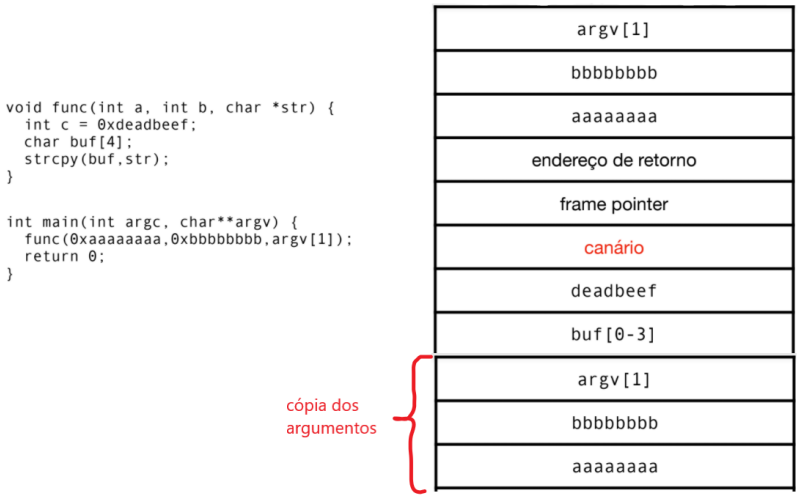
        - **Canário de terminação**: Usam '\0', '\n', EOF em vez de bytes aleatórios. As funções que manipulam strings (como o *strcpy*) irão sempre parar nestes valores, terminado o programa.

    - **StackGuard** é um canário implementado pelo GCC (-fstack-protector-strong). É simples e fácil de colocar em prática, mas tem algum overhead de espaço e performance.

- **Mitigações adicionais na stack**

    - Garantir que os buffer estão sempre junto ao canário. Qualquer tentativa mínima de overflow é logo detectada.

    - Copiar sempre os argumentos da função que estão no topo da stack para baixo (abaixo dos buffers locais). Assim, não é possível alterar estes argumentos com buffer overflow, pois temos uma cópia dos valores originais.




    - **Shadow control stack**: ter uma outra stack redundante apenas para controlo (com os endereços frame e retorno). Antes de retornar da função verifica a consistência entre a stack original e a shadow stack.

- **Memory tagging**

    - Um apontador tem uma tag associada como também cada zona de memória alocada.

    - Imagine-se que um apontador aponta para uma dada zona de memória. Ambas as tags têm de ser iguais.

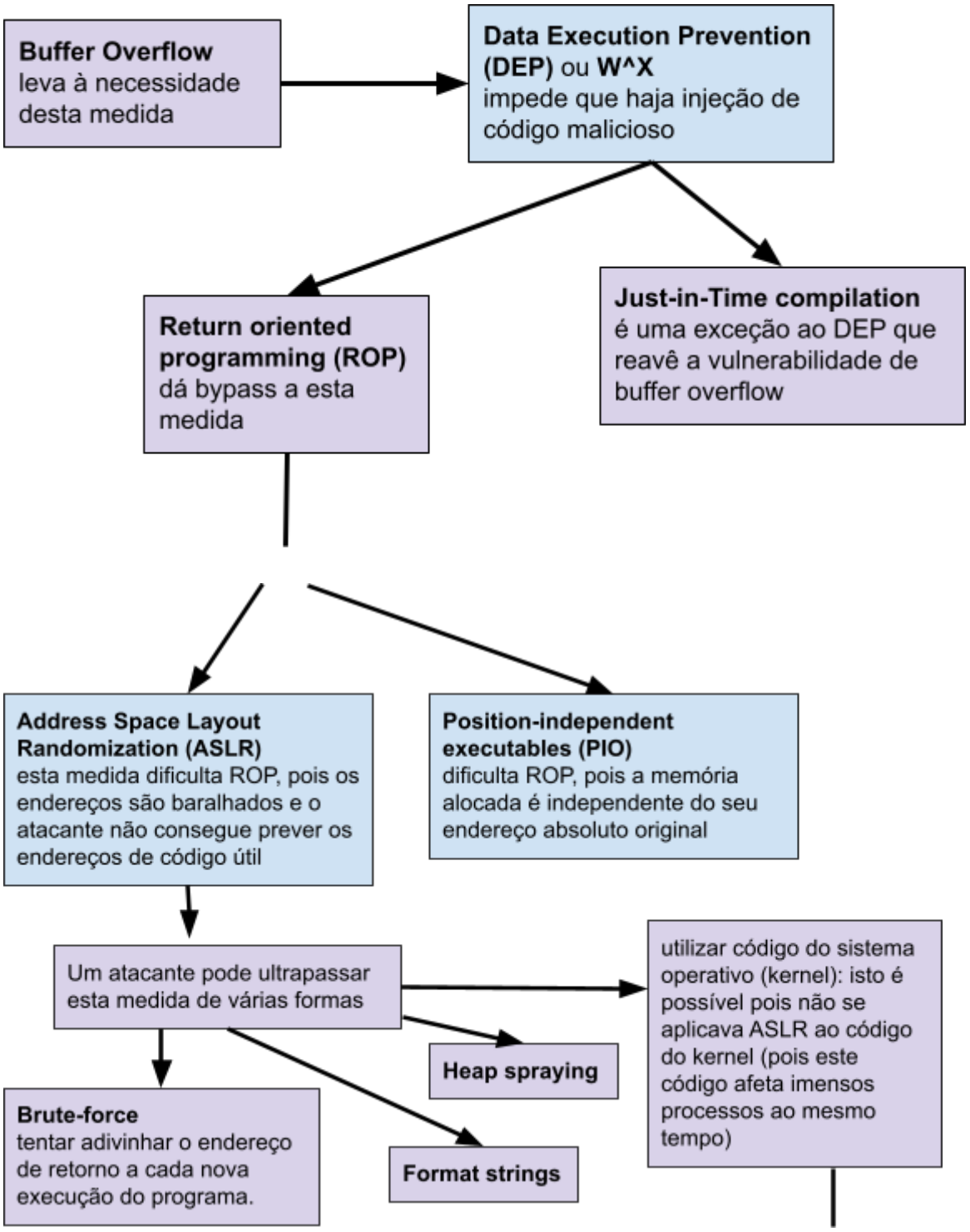
    - As tags são comparadas quando se tenta aceder à zona de memória. O programa lança uma exceção se detectar que a tag foi alterada por overflow.
    - Para alterar uma tag tem que se usar instruções específicas como o free. Se um atacante tentar reusar memória libertada através de um "use after free", o programa também lançará uma exceção.

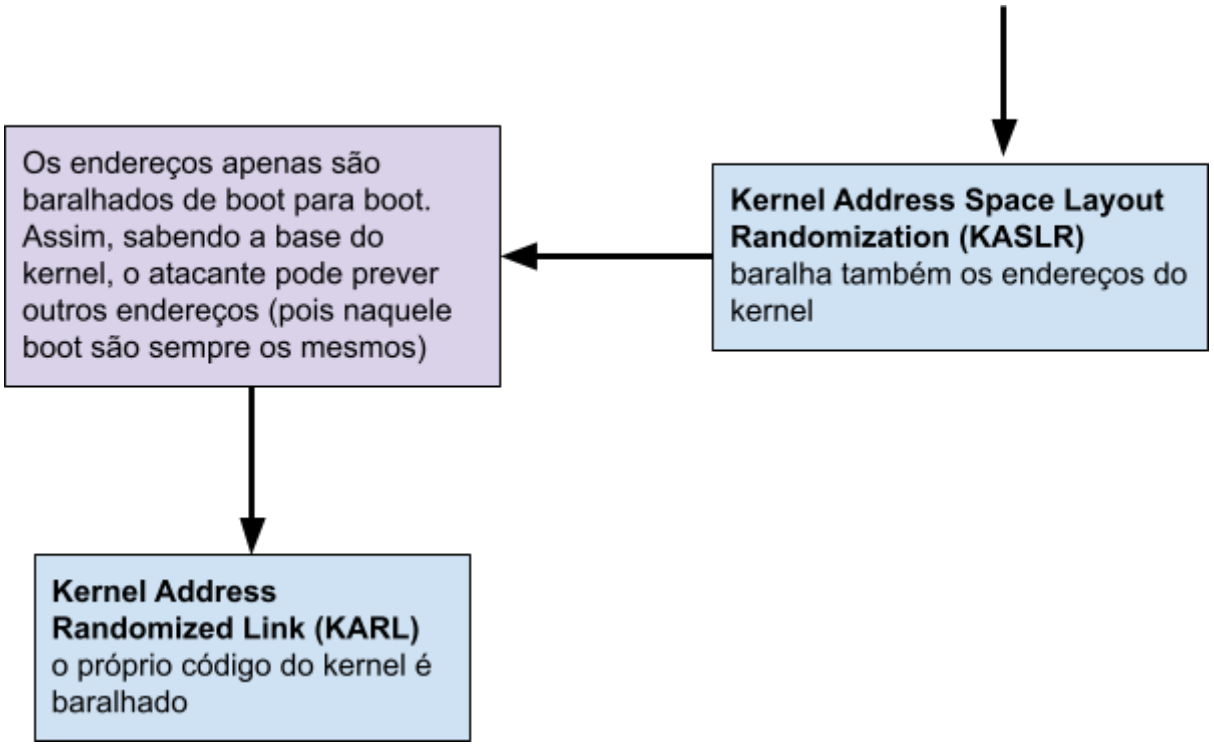
- **Control Flow Integrity (CFI)**
    - Versão simples: Confirmar que sempre que chegamos à entrada de uma função que esta é resultado de uma *call* (e não de endereços criados dinamicamente pelo atacante).
    - Versão elaborada: Computar um grafo que define todos os saltos válidos (usando criptografia)
        - sempre que um endereço é escrito algures na memória, guardar também um autenticador criptográfico
        - sempre que se usa o endereço verifica-se o autenticador
        - usa-se uma chave que não está em memória (um registo por exemplo)
    - A **shadow control stack** pode ser vista como parte do control-flow integrity.

## 3.8. Esquema "cronológico" de ataques e medidas de proteção

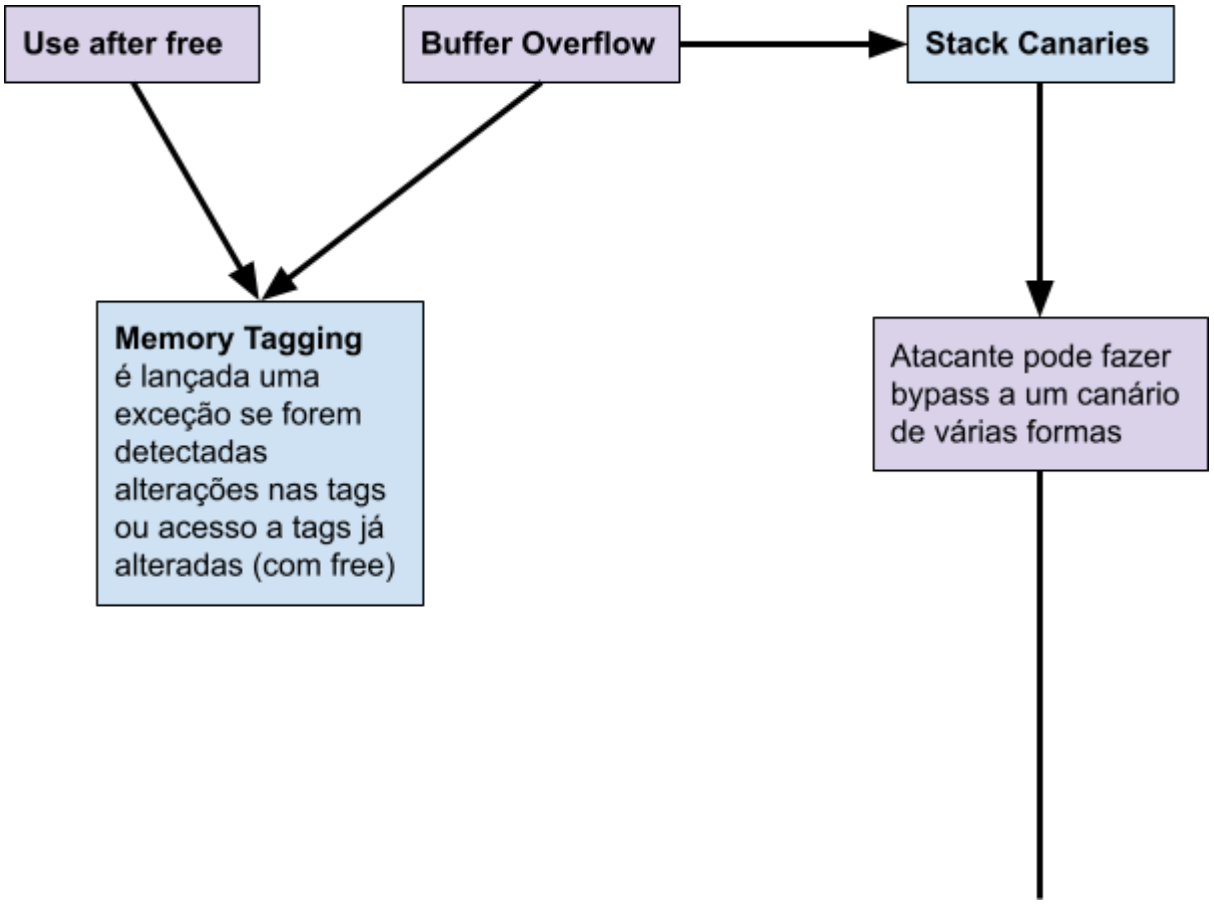
Para mais pormenores sobre cada medida de proteção ou ataque ver secções acima. Aqui pretende-se ter apenas um resumo ou um esquema "cronológico" da sequência de ataques e as medidas que os motivaram e vice-versa.

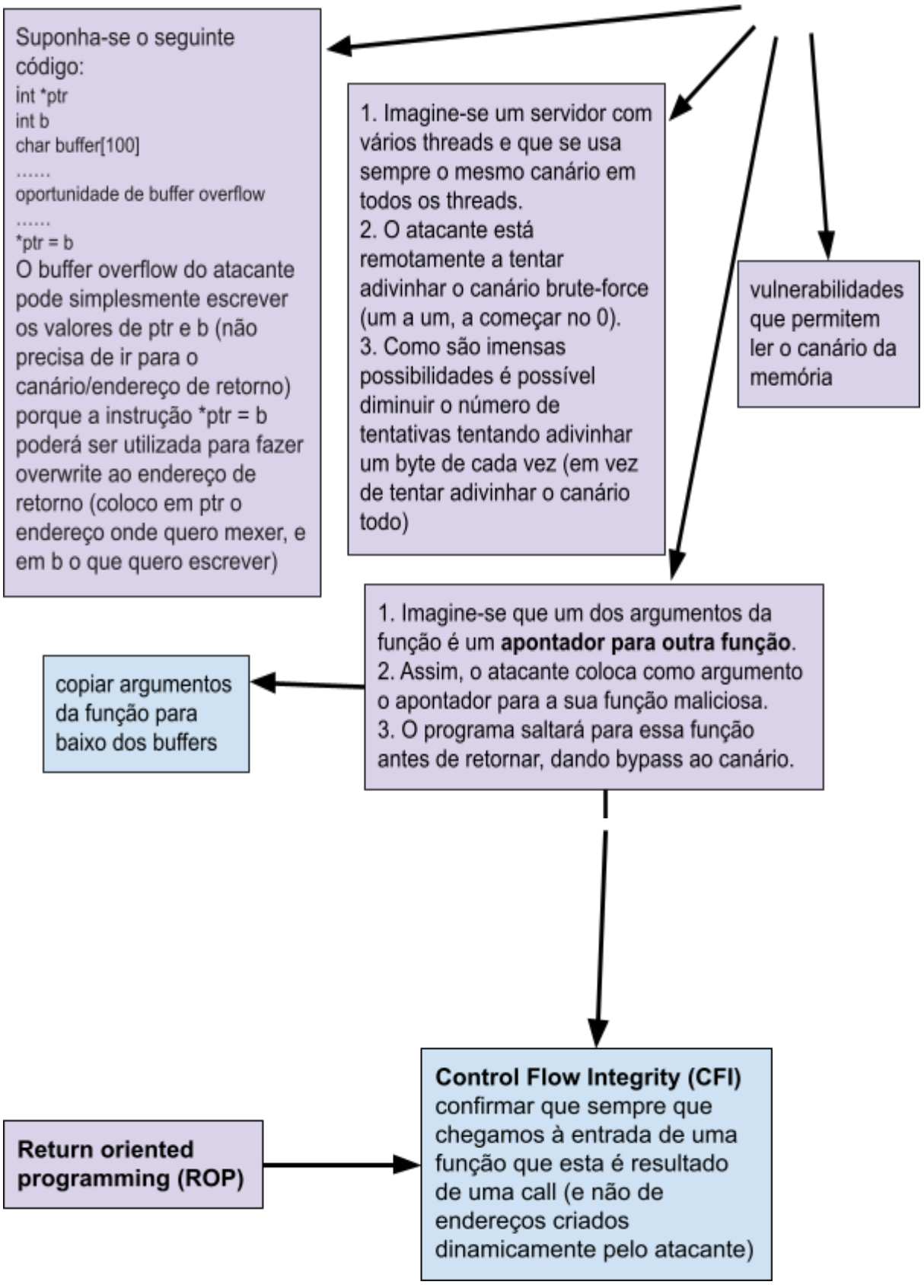
### 3.8.1. Medidas de proteção na própria plataforma






### 3.8.2. Medidas de proteção no executável




Suponha-se o seguinte código:
int *ptr
int b
char buffer[100]
.....
oportunidade de buffer overflow
.....
*ptr = b
O buffer overflow do atacante pode simplesmente escrever os valores de ptr e b (não precisa de ir para o canário/endereço de retorno) porque a instrução *ptr = b poderá ser utilizada para fazer overwrite ao endereço de retorno (coloco em ptr o endereço onde quero mexer, e em b o que quero escrever)
1. Imagine-se um servidor com vários threads e que se usa sempre o mesmo canário em todos os threads.
2. O atacante está remotamente a tentar adivinhar o canário brute-force (um a um, a começar no 0).
3. Como são imensas possibilidades é possível diminuir o número de tentativas tentando adivinhar um byte de cada vez (em vez de tentar adivinhar o canário todo)
vulnerabilidades que permitem ler o canário da memória
1. Imagine-se que um dos argumentos da função é um apontador para outra função.
2. Assim, o atacante coloca como argumento o apontador para a sua função maliciosa.
3. O programa saltará para essa função antes de retornar, dando bypass ao canário.
copiar argumentos da função para baixo dos buffers
Control Flow Integrity (CFI) confirmar que sempre que chegamos à entrada de uma função que esta é resultado de uma call (e não de endereços criados dinamicamente pelo atacante)
Return oriented programming (ROP)

## 3.9. Program safety

- Como se garante segurança ao nível da própria programação?
- Program safety é um conceito nas áreas da linguagem de programação que constata que cada linguagem de programação tem uma semântica, um significado. Ou seja, um programa é considerado seguro se todas as suas execuções têm significado. Quando ocorre um erro, a execução deixa de ter significado e deixa portanto de ser seguro. Assim, uma linguagem segura é aquela que garante que todas as execuções têm significado, ou seja, que o programa se comporta exatamente como esperado.
- Nas operações de memória existem diversos tipos de operações unsafe. Em geral todas elas aparecem como tendo resultados "undefined" na semântica de linguagens como o C. Por exemplo, em C, se usarmos uma variável que não foi inicializada “tudo pode acontecer”, a variável está “undefined”. Daí que C não seja considerada uma linguagem muito segura.
- As linguagens de programação não são desenhadas com o objetivo de garantir segurança. Ainda assim, umas linguagens são melhores que outras, e oferecem mais automação do que outras para garantir safety (tradeoff: para ser mais seguro, o programador tem menos controlo).
    - Linguagens **strongly typed** como Java ou Haskell verificam um conjunto muito alargado de condições em tempo de compilação (ou seja, nem sequer compilam se detectarem algum tipo de operação unsafe).
    - Outras linguagens como Python são **weakly typed**, mas verificam type safety na execução (ou seja, não verificam os tipos na compilação, mas verificam na execução).
    - Linguagens **interpretadas** como Java ou Python detectam problemas de safety na execução: acessos errados a memória/containers originam exceções
    - O Rust inclui um conjunto de checks estáticos e dinâmicos para garantir safety.
- Mesmo em linguagens como o C ou assembly é possível ter garantias de safety: existem ferramentas que nos permitem verificar essas propriedades (por exemplo, valgrind, fuzzing, Frama-C)

# 4. Segurança de Sistemas

## 4.1. Princípios fundamentais

- Quando pensamos sobre a segurança de sistemas complexos, os seguintes princípios devem estar sempre presentes:
    - Economy of mechanism
    - Fail-safe defaults
    - Complete mediation
    - Open design
    - Separation of privilege
    - Least privilege
    - Least common mechanism
    - Psychological acceptability
    - Work factor
    - Compromise recording

### 4.1.1. Economia nos mecanismos

- Um sistema deve ter apenas as funcionalidades (por exemplo, serviços a correr) necessários. De todos os mecanismos de segurança equivalentes devem ser adotados os mais simples.
- Ter funcionalidades "nice-to-have" só irá acrescentar complexidade tanto ao sistema como à segurança desse sistema, pelo que devem ser evitados.
- Este princípio facilita a implementação, usabilidade, validação, etc.

### 4.1.2. Proteção por omissão

- A configuração de qualquer sistema deve, por omissão (default) impor um nível de proteção conservador (por exemplo, um novo utilizador deve, por omissão, ter o mínimo de permissões).

- ***Fail closed***: caso um sistema falhe, reduz-se as permissões/privilégios. Por exemplo, quando um sistema se recusa a arrancar se algum componente crítico de segurança não está disponível.

#### 4.1.3. Desenho aberto

- A arquitetura de segurança e os detalhes de funcionamento de um sistema devem ser públicos. O sistema não deve basear-se em "security through obscurity". Mesmo que o código seja "closed-source", devemos assumir que um atacante pode ter acesso ao código, pode saber as versões do software que estou a usar, etc.
- Os segredos são parâmetros do sistema, que podem ser alterados: chaves criptográficas, passwords, tokens, etc.
- Vantagens:
    - Posso saber de bugs e vulnerabilidade muito mais cedo do que se escondesse estes detalhes todos, pois terei uma comunidade muito mais alargada a procurar por essas vulnerabilidades no código.
    - Podemos mudar os segredos sem ter que redesenhar todo o sistema de novo. (Se o desenho do sistema se baseia em linhas de código escritas e versões de software, se alguém as decobre tenho de alterar isso tudo).

#### 4.1.4. Defesa em profundidade

- Todos os mecanismos de segurança podem falhar. Por isso não podemos depositar toda a nossa confiança num só mecanismo. Devemos ter um vasto conjunto de medidas de segurança.
- É uma estratégia que procura adiar, em vez de fazer frente, o avanço do atacante. Ou seja, "prevenir em vez de remediar".
- Minimizar a probabilidade de erros na gestão de memória
- Usar linguagens de programação que garantem *memory safety* (Java, Go, Rust, etc.)
- Verificar que os programas estão corretos

#### 4.1.5. Privilégio mínimo

- Cada utilizador, compartimento, programa, etc deve ter apenas os privilégios/permissões essenciais para desempenhar a sua função.
- Contrariar este princípio amplifica sem necessidade o impacto de uma brecha local na segurança.
- Um exemplo de violação: correr todos os serviços como root.

#### 4.1.6. Separação de privilégios

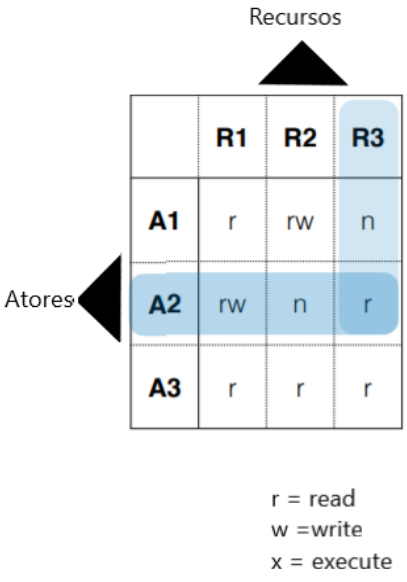
- As funcionalidades/utilização de recursos devem ser compartimentadas:
    - cada compartimento deve estar isolado dos outros
    - todos os compartimentos devem encarar os restantes como estando num outro domínio de confiança

#### 4.1.7. Mediação completa

- Um sistema gere, invariavelmente, recursos: ficheiros, dispositivos de hardware, memória, etc.
- Mediação completa implica:
    - Para todos os recursos definir uma política de proteção (para cada recurso deve-se definir que operações é possível fazer nesse recurso, quem deve ter permissões para realizar essas operações, etc)
    - Todos os acessos a todos os recursos são validados relativamente a essa política (sempre que alguém quer aceder a um recurso, o sistema deve verificar se essa pessoa tem as permissões para aceder ao recurso)
    - Exemplo: o sistema operativo é o mediador de todos os acessos à memória, um processo entre a memória virtual e a memória física. Todos os processos acedem a um espaço de memória virtual. E todos os acessos à memória são mediados pelo sistema operativo. Ou seja, um processo não pode aceder À memória física diretamente, tem primeiro de passar pelo sistema operativo que permitirá ou recusará esse acesso.

## 4.2. Controlo de acessos

- Refere-se a uma família de mecanismos de segurança que visam aplicar alguns dos princípios anteriores: Privilégio mínimo e Mediação total.
- **Ator**: entidade que realiza uma acção
- **Recurso**: sobre o qual se realiza a acção
- **Operação**: o tipo de acção que é realizada
- **Permissão**: A combinação de recurso/operação
- **Matriz de acessos:** descreve todos os tipos de acesso. Permite clareza e eficiência, mas ocupa imenso espaço.



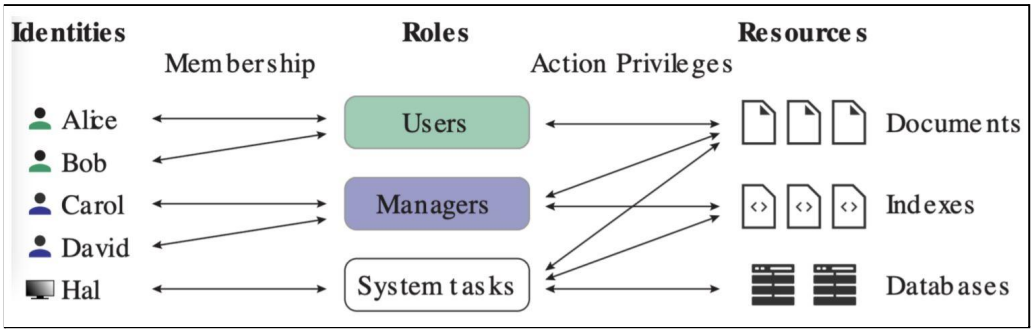
- **Listas de controlo de acessos (ACL):** para cada recurso, as permissões dos atores sobre esses recursos.
    - Vantagens:
        - Faz-nos pensar sobre como proteger cada recurso individualmente
        - Permite garantir facilmente que um recurso apenas pode ser acedido por um número limitado de atores
    - Desvantagens:
        - Não é possível determinar as permissões que um ator tem sem ver todos os recursos
        - Não há agregação de perfis de permissões

| | R1 | R2 | R3 |
|---|---|---|---|
| A1 | r | rw | n |
| A2 | rw | n | r |
| A3 | r | r | r |

- **Listas de permissões (Capabilities)**: para cada ator, as operações que pode realizar sobre cada recurso.
    - Vantagens:
        - Faz-nos pensar sobre como lidar com cada ator individualmente
        - Permite garantir facilmente que um ator apenas acede a um número limitado de recursos
    - Desvantagens:
        - Não é possível determinar todos os atores que podem aceder a um recurso sem percorrer todos os atores
        - Não há agregação de perfis de permissões.

| | R1 | R2 | R3 |
|---|---|---|---|
| A1 | r | rw | n |
| A2 | rw | n | r |
| A3 | r | r | r |

- **Role Based Access Control (RBAC):** modelo com dois tipos de relações que permitem separar a gestão de recursos da gestão de atores:
    - Ligação de perfis (roles) a conjuntos de permissões
    - Ligação de atores a perfis




    - **Desvantagem**: o facto de termos que encaixar os utilizadores/recursos num perfil, faz com que percamos expressividade (não temos muitas opções).

- **Attribute-based Access Control (ABAC):** alternativa a RBAC, para sistemas mais complexos que precisem de mais expressividade.
    - Atores e recursos têm atributos associados
    - Matriz de acessos descreve permissões com base nos atributos: para aceder a recurso com atributo A o ator deve possuir atributo B
    - Permite políticas mais expressivas, como por exemplo ter em conta o contexto geográfico e temporal, ou sistemas hierárquicos.

## 4.3. Segurança sistemas operativos

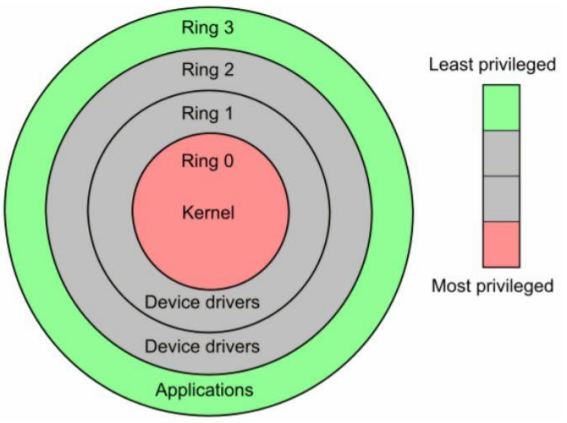
- **Sistema operativo**: Interface entre os utilizadores de um computador e o hardware, com o intuito de fazer a gestão do acesso a recursos por parte de aplicações (armazenamento, processador, memória, I/O, rede, etc; partilha destes recursos por vários utilizadores e aplicações; software que trata operações de baixo nível e oferecem abstrações convenientes para o desenvolvimento de aplicações).
- **NOTA**: aspectos concretos sobre o funcionamento de sistemas operativos (como gestão de memória, processos, comunicação entre processos, etc) já foram dadas noutra unidade curricular (Sistemas Operativos), pelo que se omite aqui esses pormenores. Aqui mencionamos aspectos de segurança e aspectos de sistemas operativos quando estes se relacionam com a segurança.

### 4.3.1. Aspetos de segurança a considerar

- Os sistemas operativos têm múltiplos utilizadores com diferentes níveis de acesso, diferentes necessidades e privilégios relativamente a recursos a utilizar.
- Os utilizadores são sempre potenciais ameaças e os recursos são sempre ativos a proteger.
- As aplicações são também potenciais ameaças e devem estar protegidas da interferência de uma das outras (separação de privilégios). Esta proteção aplica-se mesmo quando não estão a executar (afinal a informação que as aplicações armazenam está geralmente em recursos partilhados, como por exemplo, o disco).

### 4.3.2. Kernel

- O **Kernel** é a parte do sistema operativo que desempenha as operações mais críticas.
- Os dispositivos podem estar num dos seguintes modos:
    - **Kernel mode**: todas as operações são permitidas. O processador garante que apenas código que corre em modo kernel pode executar um conjunto de instruções privilegiadas
    - **User mode**: não tem acesso direto aos recursos do sistema. Os processos neste modo tem um espaço de memória gerido de forma independente (pelo próprio kernel). Qualquer processo em user mode (incluindo device drivers) tem de aceder aos recursos do sistema usando system calls. Por sua vez, parte do código das system calls executa em kernel mode!
- Sistema de **rings**: em Intel, os processadores permitem geralmente definir vários níveis de privilégio.



- Qualquer troca de informação entre os dois níveis faz parte de uma **superfície de ataque**.
- **Superfície de ataque:** define a fronteira entre contextos de confiança e, portanto, as interações com entidades potencialmente hostis. Uma superfície de ataque está bem definida quando tem um número limitado de pontos de entrada: isto é um ponto positivo para a segurança pois simplifica a análise de requisitos de segurança e a construção de políticas de segurança para os garantir.

#### 4.3.3. System calls

- Em segurança, os pontos críticos / pontos de entrada são as system calls. Ou seja, para causar danos, um processo em modo utilizador tem de o fazer através de uma system call!
- Por essa razão, existe um número limitado deste tipo de pontos de entrada: registos específicos para parâmetros, que são tipicamente apontadores para memória de processos em user mode. O processamento dessa informação é da total responsabilidade do kernel.

- Isto implica implementar sistemas de monitorização e controlo de chamadas ao sistema, por exemplo o sistema de **Reference monitor**:
    - deve estar sempre presente (se terminar, têm de ser terminados os processos monitorizados)
    - tem de ser simples para poder ser analisado e validado mais facilmente que o sistema todo
- O que podem fazer as system calls?
    - Controlo de processos (e.g., fork, load, execute, wait, alloc, free)
    - Acesso a ficheiros (criar, ler, escrever, etc.)
    - Gestão de dispositivos (obter acesso, escrever/ler, etc.)
    - Configuração do sistema (hora, data, características, estado, etc.)
    - Comunicações (estabelecer ligação, enviar/receber mensagens, etc.)
    - Proteção (alterar/obter permissões de acesso a recursos)
- O que pode fazer um atacante com system calls?
    - Para entrar num modo de funcionamento com mais privilégios, o código *user mode* deve preparar argumentos, e identificar um ponto permitido para acesso

a *kernel mode* e executar uma instrução especial que passa o controlo para o kernel.

#### 4.3.4. Processos

- **Modelo de confiança nos processos.** Confiança que depositamos nos processos:
    - O código armazenado no computador (nomeadamente a BIOS e o kernel) após uma instalação é "confiável". O processo de boot utiliza este código para colocar o kernel em memória e passar-lhe o controlo, criando um estado "confiável"
    - O kernel lança processos com permissões que garantem que nenhum novo processo pode alterar o estado de confiança.
    - Os processos de hibernação preservam o estado de confiança
    - Os administradores podem alterar o software instalado no sistema e o sistema de permissões, mas garantem que qualquer actualização preserva o estado de confiança
- **Modelo de ameaças nos processos.** Possíveis vulnerabilidades/ataques:
    - A maioria dos problemas de segurança surgem através de erros de administração.
    - Ataques em todos os níveis do boot (arranque do sistema). Vulnerabilidades que afetam a implementação dos mecanismos de arranque fazem um bypass completo ao modelo de confiança!
        - BIOS corrompida
        - ficheiros de hibernação corrompidos/roubados
        - bootloader (processo inicial que carrega tudo) corrompido
        - cold boot attacks (ler a memória do computador antes de o arrancar)
- **Medidas de mitigação nos processos:**
    - Sistemas de monitorização:
        - logs de eventos permitem detectar comportamentos suspeitos, como o crash repetido de um processo que está a tentar explorar uma vulnerabilidade
        - visualizar os processos que estão a executar, os recursos que utilizam, e os ficheiros de código que os originam

- Mediação de instalação/execução de código com base em assinaturas digitais (um entrave à introdução de código malicioso num sistema)
- Boa gestão de memória: um processo não pode aceder ao espaço de memória de outro processo!
    - Em run-time os acessos são mediados por um conjunto de mecanismos de hardware e software geridos pelo kernel.
    - Cifrar o disco para que um adversário não aceda a partes da memória virtual que estão em disco
- Nem o próprio kernel deve ter todas as permissões sobre a memória de outros processos. Impedir o kernel de escrever em partes da memória do utilizador é uma forma de impedir fugas de informação/código malicioso no caso de kernel corrompido.
- Virtualização da memória garante isolamento entre processos. A comunicação com o exterior (ficheiros, rede, ou o próprio kernel) é feita através de system calls.
    - TLB (Translation Lookaside Buffer): cache de páginas traduzidas recentemente. É uma optimização da implementação da memória virtual que visa acelerar a tradução de endereços e a implementação dos mecanismos de memória virtual.
    - Kernel mapping é uma otimização do mecanismo de memória virtual para acelerar a operação de system calls, entre outras coisas. Permite que não seja necessária uma mudança total de contexto quando se avalia uma system call.

### 4.3.5. Sistemas de ficheiros

- **Medidas de mitigação nos sistemas de ficheiros:**
    - Não abusar do uso do superuser (root), que tem permissões totais.
    - Discretionary Acess Control: apenas quem é dono de um recurso é que pode alterar as permissões sobre esse recurso.
    - Mandatory Acess Control: variante mais restritiva. Apenas o administrador (superuser) é que pode alterar as permissões.
    - Utilizar a mesma interface construída para o sistema de ficheiros para outros recursos (como por exemplo, sockets, pipes, dispositivos de I/O, objetos do

kernel). Isto permite minimizar o número de system calls/superfície de ataque, pois o sistema de controlo de acessos é sempre o mesmo.

## 4.4. Confinamento

- A ideia é que podemos precisamos de executar código não confiável. E, portanto, precisamos de confinar esse código para que não afete o resto do sistema.
- Exemplos de códigos não confiáveis que podemos precisar de executar:
    - código proveniente de fontes externas, nomeadamente sites Internet (Javascript, extensões de browsers…)
    - código legacy (original) que pode ter falhas (mesmo que venha de uma fonte confiável)
    - honeypots: código vulnerável criado de propósito para atrair atacantes e perceber que estamos a ser atacados.
    - análise forense de malware: precisamos de executar código malicioso para poder analisá-lo.
- Medidas que permitem implementar esta ideia de confinamento, ou seja, medidas para garantir confinamento aquando a execução propositada de código malicioso:
    - **Air gap**: medida que garante a segurança isolando fisicamente o sistema de redes inseguras, como a Internet pública ou uma rede local. É uma medida difícil de gerir, pois depende muito da localização física.
    - **Máquinas virtuais** ou **hypervisors**: permitem partilhar hardware, oferecendo visão virtual de hardware a cada sistema operativo. Isto garante que as ações em SO1 não afetem o contexto de SO2 e vice-versa (isto inclui vulnerabilidades: vulnerabilidades do SO1 não afetam o SO2 e vice-versa).
    - **Software Fault Isolation (SFI)**: nome genérico para isolamento de processos que partilham o mesmo espaço de endereçamento. (Por exemplo, sistemas *nix garante-se isolamento de memória: virtualizar o espaço de endereçamento e monitorizar os acessos nos mecanismos de tradução de endereços)
        - O objetivo é limitar a zona de memória acessível a uma aplicação.
        - Atribui-se segmento de memória e usam-se operações bitwise para verificar que acesso na gama correta.
        - Há certas operações perigosas, as quais devemos proteger com guardas. Exemplos de operações perigosas:

            - load/store de memória (antes de executar acesso adicionar guarda que verifica segmento ou endereço dentro do segmento)
            - saltos que podem ser utilizados para executar código externo sem guardas (é necessário validar os endereços de destino com mecanismos semelhantes)
    - **System Call Interposition (SCI)**: nome genérico para mediação de todas as system calls, concentrando os pontos de acesso a operações privilegiadas num número pequeno de pontos que podem ser monitorizados. (Por exemplo, sistemas *nix fornecem um número limitado de system calls, e mapeiam subsistemas nos mecanismos de manipulação de ficheiros)
        - A ideia é monitorizar system calls e bloquear as que não são autorizadas.
        - Há opções de implementação dentro do kernel, fora do kernel e esquemas híbridos.
        - Exemplos:
            - ptrace
            - seccomp+bpf
    - **Sandboxing**: fornece virtualização (e, portanto, proteção contra código externo) dentro de uma aplicação. (Por exemplo, os browsers são aplicações, onde internamente criam um ambiente de execução isolado para código proveniente de fontes externas com monitorização incorporada).
    - **Reference monitor**: faz mediação de todos os pedidos de acesso a recursos e implementa uma política de proteção de recursos/isolamento.
        - Tem de ser sempre invocado, este sistema está presente em todos os sistemas vistos anteriormente (air gap, hypervisors, SFI, SCI, sandboxing).
        - Todas as aplicações são mediadas
        - Tem de ser omnipresente: quando morre o reference monitor, morrem todos os processos
        - Tem de ser simples o suficiente para poder ser analisado, senão mais valia estarmos a analisar o próprio código.

### 4.4.1. System Call Interposition - Exemplo do ptrace

- Em linux, pode-se fazer ***process tracing*** através da *system call* **ptrace.**
    - permite a um processo monitor ligar-se a processo alvo (descendente)
    - é notificado quanto o processo alvo faz uma system call
    - o monitor pode terminar o alvo se a chamada não for autorizada
- Limitações:
    - É ineficiente porque obriga a interceptar todas as chamadas.
    - Se se abortar uma system call, aborta-se também o processo.
    - Ocorrem situações de race-conditions, e ataques **TOCTOU** (ataques em que tudo está OK em **Time Of Check**, após este check o atacante pode aproveitar para fazer o seu ataque, ficando o sistema operativo vulnerável em **Time Of Use**).
    - A monitorização e a execução ocorrem de forma não atómica.

### 4.4.2. System Call Interposition - Exemplo do seccomp+bpf

- **seccomp (Secure Computing Mode)**: faz um processo entrar em secure mode (confinamento) chamando a função prctl(). Esse processo, dentro desse modo, só pode terminar/retornar ou utilizar ficheiros já abertos. Uma violação leva o kernel a terminar o processo.
- Muito utilizado em sistemas como o Chromium, Docker (para isolar conatiners), etc.

### 4.4.3. Máquinas virtuais

- **Exemplos:**
    - NSA NetTop
    - Providers cloud
    - Qubes
- **Quebras de isolamento**. Como é possível quebrar o confinamento nas máquinas virtuais?
    - Existem muitos mecanismos no hardware partilhado que permitem fugas de informação e quebra de isolamento. (Por exemplo, canais subliminares em que observamos os efeitos colaterais que a virtualização tem no hardware e especulamos sobre o que se passa tendo em conta esses efeitos.)
    - Um malware pode alterar o seu comportamento se souber que está a ser executado numa máquina virtual para análise forense (isto porque geralmente os malwares são analisados em máquinas virtuais). Formas de detetar que se está em ambiente virtualizado:
        - Instruções disponíveis
        - Features de HW específicas
        - Latência no acesso a memória (profiling)
        - Uma máquina virtual utiliza sempre uma parte dos recursos do hardware. Um guest OS malicioso pode detectar esta limitação de recursos.

# 5. Malware

## 5.1. Terminologia

- **Malware:** software desenhado especificamente para causar danos através da exploração de uma vulnerabilidade ou execução de código malicioso, comprometendo o estado de confiabilidade de um sistema.
- Tipos de malware:
    - **Vírus**: malware que não se propaga sozinho, fica latente até o utilizador fazer alguma determinada ação que dará trigger a esse malware.
    - **Worm**: malware que se propaga autonomamente e consegue criar sozinho as condições para se executar e propagar. (exemplos: Morris (1998), Blaster (2003), Wannacry (2017)).
    - **Rootkit**: malware desenhado para permitir a um atacante acesso com privilégios elevados a um sistema, escondendo também a sua presença.
    - **Trojan**: código que aparenta ser legítimo, mas tem como objetivo (tipicamente) transmitir informação para um atacante.

## 5.2. Vírus

- Sequência de passos:
    1. Utilizador executa um programa infectado
    2. O código do vírus fica armazenado no programa
    3. O vírus é executado quando o programa é executado.
    4. Por sua vez, o vírus localiza outro programa para infectar.
    5. O vírus replica-se para o código desse outro programa (dissimulação cada vez mais elaborada)
    6. Se determinada lógica for ativada: executa tarefa maliciosa
    7. No final pode desaparecer (apagar-se).
- Este tipo de malware executa com os privilégios do utilizador ativo no momento da tomada de controlo: pode fazer o que o utilizador faz. Pode também explorar outras vulnerabilidades (como por exemplo escalar privilégios, com o programa setuid)

- Objetivos:
    - brincadeira ou causar danos: pop-ups, apagar ficheiros, danificar hardware
    - vigilância/espionagem: key logging, captura de ecrã, audio, câmara

## 5.3. Botnets

- **Botnet**: é uma rede de computadores com um sistema de comando e controlo comum (C2).
    - C2 é o sistema de comando e controlo.
    - Cada computador é chamado um bot
    - O controlador envia comandos através da rede: spam, phishing, DoS, roubo de informação local, etc.
    - As botnets têm geralmente a capacidade de se actualizar automaticamente.
- Dois tipos de arquitectura:
    - **centralizada**: com múltiplos servidores centrais para robustez
    - **peer-to-peer**: auto organizada e hierárquica
- Dois tipos de fluxo:
    - **push**: servidor envia comandos
    - **pull**: periodicamente, bot pergunta se há comandos
- Estratégias de detecção:
    - Detectar o malware na máquina comprometida
    - Detectar tráfego de rede para comunicação com C2
    - Expor máquinas (honeypots) para serem comprometidas e monitorizadas
- Estratégias de combate:
    - Limpar máquinas comprometidas e/ou isola-las da rede
    - Isolar/desligar o C2
    - Tomar o controlo do C2 e usá-lo para desativar botnet.

## 5.4. Outros exemplos de tomada de controlo

- Muitos exemplos de tomada de controlo já foram vistas em secções anteriores.
- **Malvertising**: utilizar sistemas de placement de anúncios para chegar até browsers vulneráveis (exemplo: Cryptowall)
- **Engenharia social**: convencer o utilizador a instalar/executar software que oferece serviços, mas que depois toma conta da máquina. (exemplos: Fake Antivirus, USB autorun
- Deturpar equipamento no fabrico.
- Deturpar equipamento em trânsito (no caso de encomendas).
- Atacar fornecedor de software, injetar updates que contêm backdoors.

## 5.5. Combater malware

- **Intrusion Detection System (IDS)**: detecção ocorre depois do ataque ter sido concretizado
- **Intrusion Prevention System (IPS)**: intervenção rápida para evitar ataque
- **Blacklisting**: tendo em conta os ataques que já se conhece, impedir logo à partida que esses ataques aconteçam. Por omissão, considera-se que todos os outros são bons.
- **Whitelisting**: definimos quem pode entrar e por omissão, tudo o resto não pode entrar.

### 5.5.1. Erros de detecção

- **Falso positivo**: caso em que há alertas de detecção quando não houve intrusão
- **Falso negativo**: caso em que não há alerta de detecção e houve de facto uma intrusão
- O objetivo é ter um bom equilíbrio entre a quantidade de falsos positivos e falsos negativos. (Nem mesmo 0% de falsos negativos é desejável, pois estaremos sempre a receber alertas por tudo e por nada.)