

3A Sécurité OS

Cours 3

Maxime Puys

IUT de Clermont-Ferrand — Université Clermont Auvergne

2026-10-01

Ce document est distribué sous licence Creative Commons BY-NC-SA 4.0. La copie, la distribution et la modification sont autorisées à des fins pédagogiques.

Plan

1. Cadrage
2. Modèle mémoire d'un processus
3. Pile et appels de fonction
4. Débordement de tampon sur la pile
5. Autres classes de débordement

6. ROP

7. Rétro-ingénierie

8. Protections et durcissement

9. Sécurité Linux

10. Authentification locale

11. Racine de confiance



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

1. Cadrage

Un système d'exploitation partage le matériel entre plusieurs programmes qui ne se font pas confiance. Il doit donc garantir :

- **Isolation des processus** — un processus ne peut pas lire ou modifier la mémoire d'un autre sans passer par un mécanisme prévu à cet effet
- **Séparation des privilèges** — espace noyau (accès complet au matériel) vs espace utilisateur (accès arbitré par des appels système)
- **Contrôle d'accès** — qui a le droit de lire, écrire, exécuter quelle ressource (fichier, socket, périphérique)
- **Surface d'attaque** — tout code qui traite une entrée non fiable (un programme setuid, un serveur réseau, un pilote) est un point d'entrée potentiel

⇒ Aujourd'hui : deux façons dont ces garanties se cassent — débordement de tampon, et contrôle d'accès mal configuré.

Un débordement de tampon n'est intéressant pour un attaquant que s'il **aboutit** à quelque chose : exécuter du code arbitraire, puis obtenir des privilèges qu'il n'avait pas.

⇒ C'est cette chaîne complète que nous allons suivre, pas seulement le bug initial.

1. Écraser une variable voisine sur la pile § Débordement
2. Écraser l'adresse de retour § Débordement
3. Contourner la protection NX avec des gadgets § ROP
4. Comprendre un binaire inconnu § Rétro-ingénierie
5. Transformer l'exécution de code en droits root § Sécurité Linux



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

2. Modèle mémoire d'un processus

Exemple

1	hello.c	(un seul printf)	73 octets de source
2	hello-dynamique		6 520 octets
3	hello-statique		371 168 octets

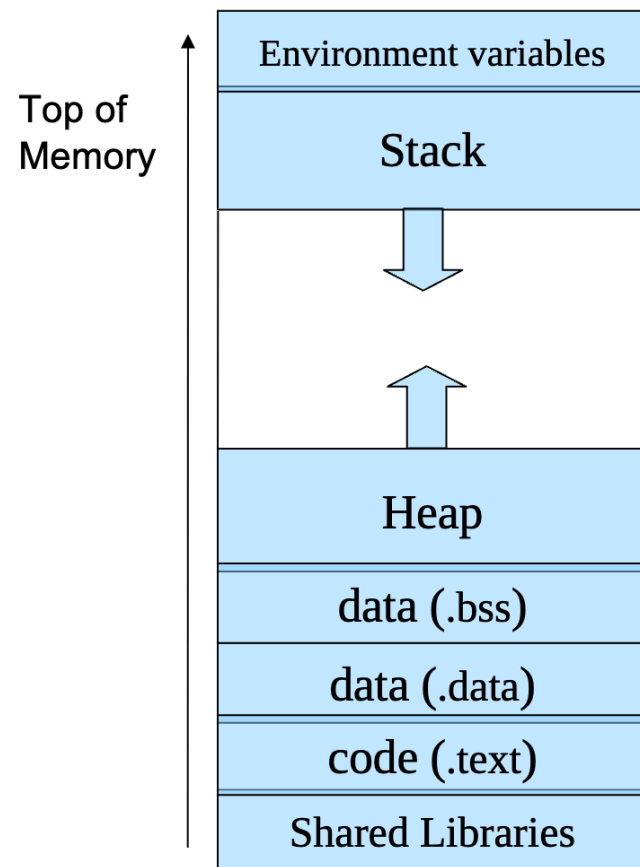
Inconvénients d'un exécutable statique

- Contient tout le code dont il dépend : chargement plus lent
- Bien plus volumineux sur disque et en mémoire
- Ne profite pas des correctifs de sécurité des bibliothèques partagées sans recompilation

Outils d'inspection : `readelf`, `objdump`, `nm`, `ldd`, `strings`, `gdb`

- **Pile (stack)**
 - ▶ variables locales
 - ▶ contexte d'appel de chaque fonction
(→ *section suivante*)
- **Données (data)**
 - ▶ `.bss` – globales non initialisées
 - ▶ `.data` – globales initialisées
 - ▶ `tas (heap)` – allocation dynamique
- **Code (text)**
 - ▶ instructions du programme
 - ▶ généralement en lecture seule

Sous Linux : `cat /proc/<pid>/maps`





**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

3. Pile et appels de fonction

Quand un programme appelle une fonction, deux moments distincts déposent des choses sur la pile :

1. **L'instruction `call`** — empile l'adresse à laquelle reprendre l'exécution une fois la fonction terminée (l'**adresse de retour**)
2. **Le prologue de la fonction appelée** — sauvegarde l'ancien pointeur de frame de l'appelant, puis réserve la place des variables locales

⇒ Ces éléments sont empilés dans cet ordre, et une fonction bien élevée les retire dans l'ordre inverse à sa sortie.

Avant l'appel, la pile ne contient que la frame de l'appelant.

adresses hautes

call empile l'adresse de retour : où reprendre une fois la fonction terminée.

adresses hautes

Adresse de retour

Le prologue sauvegarde l'ancien pointeur de frame de l'appelant (**rbp sauvegardé**).

- `rbp` pointe vers la frame **actuellement active** : c'est le repère à partir duquel la fonction retrouve ses variables locales et ses paramètres, à des décalages fixes
- Il n'existe qu'**une seule copie** de `rbp` dans tout le programme — la fonction appelée s'apprête à la réutiliser pour sa propre frame
- Elle doit donc d'abord sauvegarder la valeur de l'appelant, sinon celui-ci serait perdu à son retour

adresses hautes

Adresse de retour

Ancien rbp sauvegardé

Le prologue réserve la place des variables locales — dont nos buffers.

adresses hautes

Adresse de retour

Ancien rbp sauvegardé

Variables locales (buffer, ...)

adresses basses — sommet de la pile

ret dépile l'adresse de retour : l'exécution reprend exactement là.

adresses hautes

Adresse de retour

Ancien rbp sauvegardé

Variables locales (buffer, ...)

adresses basses — sommet de la pile



reprise à l'adresse de retour

Suivons ce mécanisme sur un cas concret où **plusieurs frames coexistent** : une fonction récursive. Chaque appel imbriqué crée sa propre frame, empilée au-dessus de celle de son appelant.

```
1  int sum(int n) {  
2      if (n == 0) return 0;  
3      return sum(n - 1) + n;  
4  }  
5  
6  int main(void) {  
7      int b = sum(2);  
8  }
```

⇒ Trois appels de `sum` vont coexister sur la pile avant que le premier ne se termine :
`sum(2)`, `sum(1)`, `sum(0)`.

Avant l'appel, la pile ne contient que le contexte de `main` — c'est lui que pointent `rbp` et `rsp`.

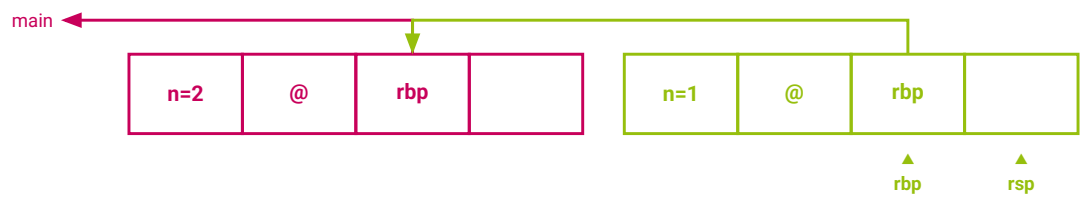
(pile vide)

`sum(2)` est appelée : sa case **rbp sauvegardé** reçoit une copie du `rbp` de `main` — le premier maillon de la **chaîne des frames**. `rbp` saute sur cette case, `rsp` avance jusqu'au bout de la frame (après la place réservée pour le résultat).



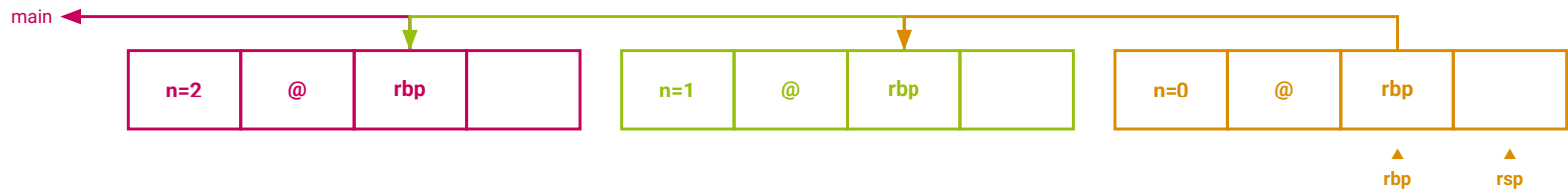
chaque frame = argument · adresse de retour · rbp sauvegardé (chaîné) · résultat

`sum(1)` est appelée depuis `sum(2)` : sa case **rbp sauvegardé** contient une copie du `rbp` de `sum(2)` — un maillon de plus dans la chaîne. `rbp` et `rsp` avancent tous deux dans cette nouvelle frame, plus loin qu'avant.



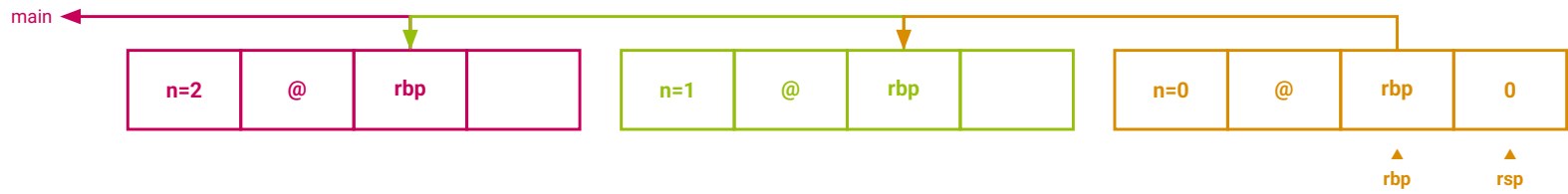
chaque frame = argument · adresse de retour · rbp sauvegardé (chaîné) · résultat

`sum(0)` atteint le cas de base (`n == 0`) : une troisième frame s'ajoute, chaînée à celle de `sum(1)`. `rbp` et `rsp` y avancent à leur tour.



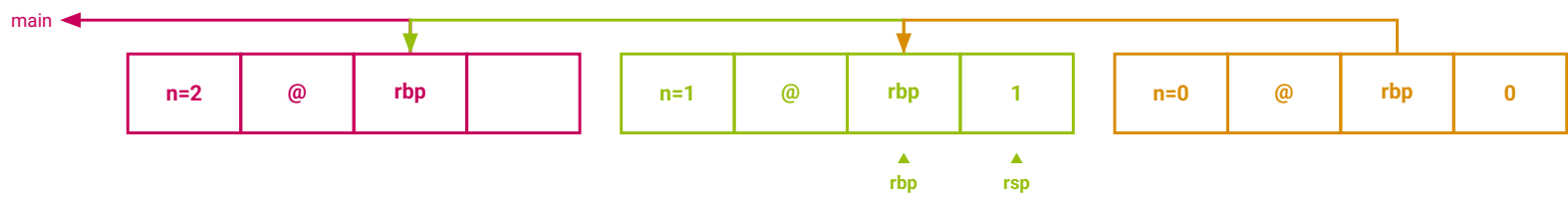
chaque frame = argument · adresse de retour · rbp sauvegardé (chaîné) · résultat

`sum(0)` retourne 0 : le résultat s'écrit dans **sa propre frame**, `rbp` et `rsp` n'ayant pas encore bougé.



chaque frame = argument · adresse de retour · rbp sauvegardé (chaîné) · résultat

`sum(1)` reprend la main : `rbp` est **restauré** à la valeur trouvée dans sa case `rbp` sauvegardé, `rsp` recule jusqu'au bout de la frame de `sum(1)` — **avant** celle de `sum(0)`, qui reste visible en mémoire mais n'est plus pointée. `sum(1)` calcule `0 + 1` et écrit `1`.



chaque frame = argument · adresse de retour · rbp sauvegardé (chaîné) · résultat

Même mécanisme pour `sum(2)` : `rbp` et `rsp` reculent jusqu'à sa frame, en suivant la chaîne en sens inverse. Elle calcule `1 + 2` — le résultat `3` remonte jusqu'à `main`.



chaque frame = argument · adresse de retour · rbp sauvegardé (chaîné) · résultat



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

4. Débordement de tampon sur la pile

```
1  int main(void) {  
2      int is_admin;  
3      char name[48];  
4  
5      is_admin = 0;  
6      scanf("%s", name);  
7  
8      if (is_admin != 0)  
9          printf("Bienvenue, administrateur\n");  
10     else  
11         printf("Acces refuse\n");  
12 }
```

`scanf("%s", name)` ne vérifie jamais la taille de l'entrée. Selon la disposition choisie par le compilateur, `is_admin` peut se trouver juste **au-dessus** de `name` en mémoire — une entrée assez longue l'écrase, sans lui avoir jamais été assignée directement.

⇒ Il suffit de dépasser la taille du buffer de quelques octets — sans connaître la valeur de `is_admin`, ni même qu'un lien existe entre les deux variables.

Rappel : voici la frame juste avant que la fonction n'écrive dans son buffer.

adresses hautes

Adresse de retour

Ancien rbp sauvegardé

Variables locales (buffer, ...)

adresses basses — sommet de la pile

Le programme écrit dans `buffer` sans vérifier sa taille. Si l'entrée est plus longue que prévu, l'écriture continue **au-delà** de la fin du buffer — vers les adresses plus hautes.

adresses hautes

Adresse de retour

Ancien rbp sauvegardé

Variables locales (buffer, ...)

adresses basses — sommet de la pile

L'écriture déborde dans la case suivante : le pointeur de frame sauvegardé est écrasé par des octets choisis par l'attaquant.

adresses hautes

Adresse de retour

??? (écrasé)

Variables locales (buffer, ...)

adresses basses — sommet de la pile

Puis l'**adresse de retour** elle-même est réécrite — avec une valeur choisie par l'attaquant.

adresses hautes

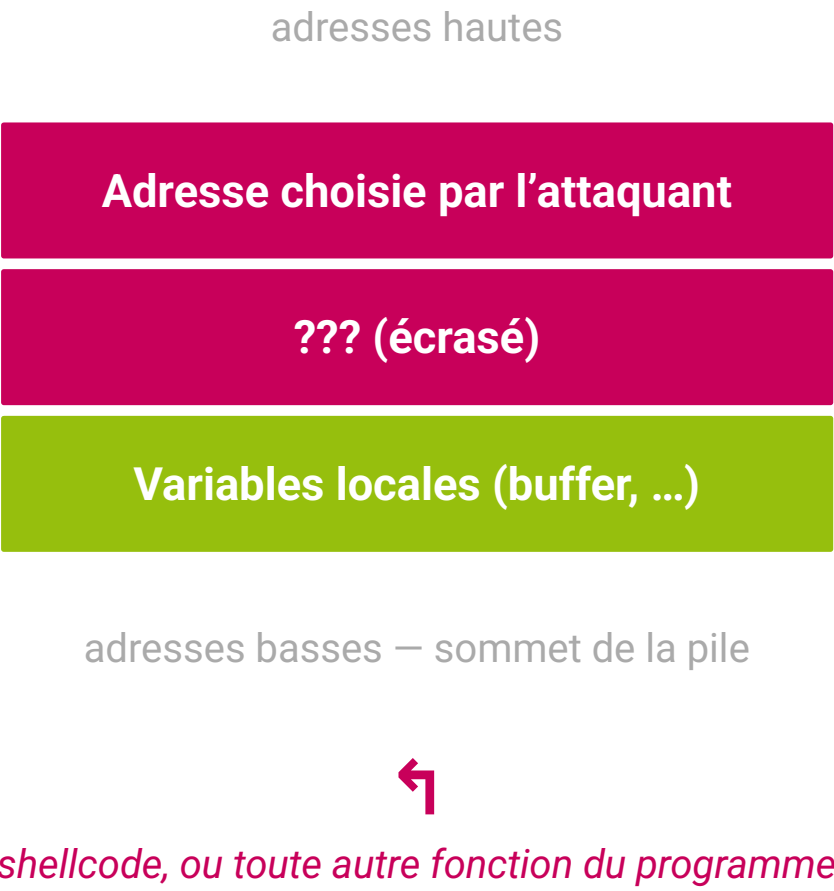
Adresse choisie par l'attaquant

??? (écrasé)

Variables locales (buffer, ...)

adresses basses — sommet de la pile

`ret` dépile cette fausse adresse :
l'exécution reprend **où l'attaquant l'a décidé**, pas chez l'appelant.



Une fois l'adresse de retour sous son contrôle, l'attaquant a deux cibles classiques :

Shellcode injecté

Le code exécutable est fourni par l'attaquant **dans le débordement lui-même** — l'adresse de retour pointe alors vers le buffer qui vient d'être rempli.

ret2libc

L'adresse de retour pointe vers une fonction **déjà présente** dans le programme ou ses bibliothèques (ex. `system("/bin/sh")`) : pas besoin d'injecter du code exécutable, seulement une adresse et des arguments.

⇒ Le shellcode injecté suppose que la pile est exécutable — la contre-mesure NX le rend inopérant. `ret2libc`, elle, n'a jamais besoin d'exécuter la pile : c'est le point de départ du ROP.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

5. Autres classes de débordement

Les types de base ont une plage de valeurs limitée :

- `unsigned int` (32 bits) : de 0 à $2^{32} - 1 = 4\,294\,967\,295$
- `int` signé (32 bits) : de -2^{31} à $2^{31} - 1$

Dépassement

$4\,294\,967\,295 + 1 = 0$ pour un `unsigned int` — le calcul « boucle » silencieusement en C, il ne lève aucune exception.

Un attaquant qui contrôle une valeur utilisée dans :

- le calcul de la taille d'un buffer alloué,
- une vérification de borne avant une copie,

peut choisir cette valeur pour que le calcul déborde et **contourne la vérification elle-même**.

```
1  struct frame {  
2      char head[150];  
3      unsigned char mark;  
4      char body[150];  
5  };  
6  
7  int merge(char *a, char *b, unsigned char lenA, unsigned char lenB) {  
8      if ((unsigned char)(lenA + lenB) > 150)  
9          return -1; // [ERREUR] Taille trop grande !  
10 }
```

La vérification caste `lenA + lenB` en `unsigned char`, le type de ses propres opérandes, avant de comparer.

Le contournement

Avec `lenA = 160` et `lenB = 100` : `lenA + lenB = 260`, mais (unsigned char) `260 = 4` — la vérification `4 > 150` est fausse, **elle passe**.

```
1 memcpy(f.head, a, lenA);           // 160 octets
2 memcpy(f.head + lenA, b, lenB);    // 100 octets de plus
```

Les deux appels utilisent pourtant les **vraies** valeurs (160 puis 100, pas leur somme tronquée) : 260 octets sont copiés dans un espace prévu pour 150 — bien au-delà de `f.head`, jusque dans `f.mark` et `f.body`.

⇒ Corriger : comparer dans un type assez large pour ne **jamais** déborder — ici
`(unsigned) lenA + (unsigned) lenB > 150`, en `int` plutôt qu'en `unsigned char`.

`printf` va chercher ses arguments sur la pile, dans l'ordre où le format l'indique :

```
1 printf("%x", 1);    // affiche 1 : l'argument est fourni
2 printf("%x");       // affiche... une valeur qui traîne sur la pile
```

Si le **format lui-même** vient d'une entrée utilisateur (`printf(buf)` au lieu de `printf("%s", buf)`), l'attaquant choisit les formats :

%n : le format dangereux

Écrit le nombre de caractères déjà affichés à l'adresse pointée par l'argument correspondant — une **écriture arbitraire** pilotée par la chaîne de format.

CVE-2012-0809 (format string dans `sudo`) · CVE-2022-26941 (radios police TETRA)

Le principe est le même que sur la pile — une écriture qui dépasse la taille d'un buffer — mais l'espace voisin n'est plus une adresse de retour : ce sont les métadonnées de l'allocateur, ou un autre objet alloué par `malloc`.

Conséquences typiques

- Corruption des métadonnées de chunk → écriture arbitraire lors d'un futur `free()`
- Écrasement d'un pointeur de fonction ou d'une table virtuelle (C++) logé dans l'objet voisin

⇒ Une classe de bug voisine : le **use-after-free** — utiliser un pointeur après avoir libéré la mémoire qu'il désigne.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

6. ROP

Le shellcode injecté suppose une pile exécutable : NX le rend inopérant. `ret2libc` contourne NX, mais reste limité à **un seul** appel de fonction existante.

Return-Oriented Programming

Au lieu d'appeler une seule fonction, l'attaquant enchaîne des dizaines de courts fragments de code **déjà présents** dans le binaire ou `libc`, pour reconstruire un comportement arbitraire — sans jamais injecter la moindre instruction.

⇒ Ce que l'attaquant place sur la pile n'est que des **données** (des adresses) — NX ne s'y oppose pas, il n'interdit que l'exécution de la pile elle-même.

Gadget

Une courte séquence d'instructions, terminée par `ret`, qui existe déjà quelque part dans le code du programme ou d'une bibliothèque chargée.

```
1  pop rdi    ; retire une valeur de la pile, la place dans rdi
2  ret        ; puis saute vers l'adresse suivante sur la pile
```

Isolé, ce gadget ne fait presque rien. Mais `rdi` est le registre du **premier argument** en convention d'appel System V amd64 — et son `ret` final ne rend pas la main à l'appelant : il saute vers **la prochaine adresse** que l'attaquant a placée sur la pile.

⇒ Chaque gadget se termine par `ret` : c'est ce `ret`, et non plus `call`, qui enchaîne un gadget vers le suivant.

Le débordement ne place plus une seule adresse de retour, mais une **suite** d'adresses et de valeurs choisies par l'attaquant.

sommet de la pile

gadget 1 : pop rdi ; ret

adresse de "/bin/sh"

gadget 2 : system

Le premier `ret` de la fonction vulnérable dépile la première case : l'exécution saute sur le premier gadget.

sommet de la pile

gadget 1 : `pop rdi ; ret`

adresse de `"/bin/sh"`

gadget 2 : `system`

↩ premier saut

`pop rdi` consomme la case **juste en dessous** — ici, l'adresse de la chaîne `"/bin/sh"`. Puis son propre `ret` dépile la case suivante.

sommet de la pile

gadget 1 : pop rdi ; ret

adresse de `"/bin/sh"`

gadget 2 : system

← **chargée dans rdi**

Ce `ret` saute à son tour — ici, directement vers `system`, avec `rdi` déjà prêt.

sommet de la pile

gadget 1 : `pop rdi ; ret`

adresse de `"/bin/sh"`

gadget 2 : `system`

↩ `system("/bin/sh")`

Sur ce principe, TP3 construit une chaîne complète avec les gadgets déjà présents dans le binaire fourni : un `pop rdi ; ret` pour l'argument, un appel à `system` (ou un `execve` monté gadget par gadget) pour l'exécution.

Outillage

`ROPgadget --binary <prog>` liste automatiquement tous les gadgets disponibles dans un binaire ; il reste à choisir ceux qui posent les bons registres, dans le bon ordre, puis à écrire leurs adresses dans l'ordre d'exécution voulu.

⇒ Le ROP ne dispense pas de comprendre la pile et les conventions d'appel — il les exploite directement.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

7. Rétro-ingénierie

```
1 source.c → préprocesseur → compilateur → assembleur → éditeur de liens → binaire
```

À chaque étape, de l'information est perdue : noms de variables, commentaires, structure des boucles et des fonctions. Le binaire final n'est qu'une suite d'instructions machine — la rétro-ingénierie consiste à remonter ce chemin **sans le code source**.

Désassembler ou décompiler

Le **désassembleur** (`objdump -d`) traduit les octets en instructions assembleur, fidèlement. Le **décompilateur** (Ghidra) va plus loin : il reconstruit un pseudo-code C plausible — plus lisible, mais parfois approximatif (types, noms de variables inventés).

⇒ Aucune des deux étapes n'est réversible à l'identique : le code retrouvé n'est jamais le code source original.

Analyse statique — sans exécuter le programme

- `strings` : liste les chaînes de caractères imprimables du binaire
- `objdump -d` / Ghidra : désassemblage et décompilation
- `nm`, `readelf` : symboles et structure du fichier

Analyse dynamique — en exécutant le programme

- `ltrace` : trace les appels aux bibliothèques (`strcmp`, `malloc`, ...)
- `strace` : trace les appels système (`open`, `read`, `execve`, ...)
- `gdb` : exécution pas à pas, points d'arrêt, inspection de la mémoire

⇒ Statique pour comprendre la structure, dynamique pour observer le comportement réel — les deux se combinent.

Le binaire `login_check.out` demande un mot de passe. `strings` ne le révèle pas : il n'est jamais écrit en clair dans le binaire.

```
1 // dans hide_password(), en assembleur inline :  
2 movb $0x67, 0(%rdi) // écrit l'octet 0x67 ('g') à buffer[0]  
3 movb $0x61, 1(%rdi) // écrit l'octet 0x61 ('a') à buffer[1]  
4 // ... un movb par octet du mot de passe
```

Chaque octet est déposé en mémoire un par un, par une instruction distincte — aucune chaîne de caractères n'existe dans le binaire pour `strings`.

⇒ Le mot de passe **stocké** échappe ainsi à `strings` — mais la logique qui le construit reste, elle, visible dans le code décompilé.

En décompilant avec Ghidra, `hide_password` apparaît clairement : une suite d'instructions `MOV`, dans l'ordre, chacune plaçant un octet à la position suivante du buffer.

```
1  octets trouvés, dans l'ordre : 67 61 72 64 69 65 6E
```

```
1  0x67 = 'g'    0x61 = 'a'    0x72 = 'r'    0x64 = 'd'
```

```
2  0x69 = 'i'    0x65 = 'e'    0x6E = 'n'    -> "gardien"
```

⇒ Ce que `strings` ne voit pas comme du texte, le décompilateur le montre comme une suite d'instructions parfaitement lisible.

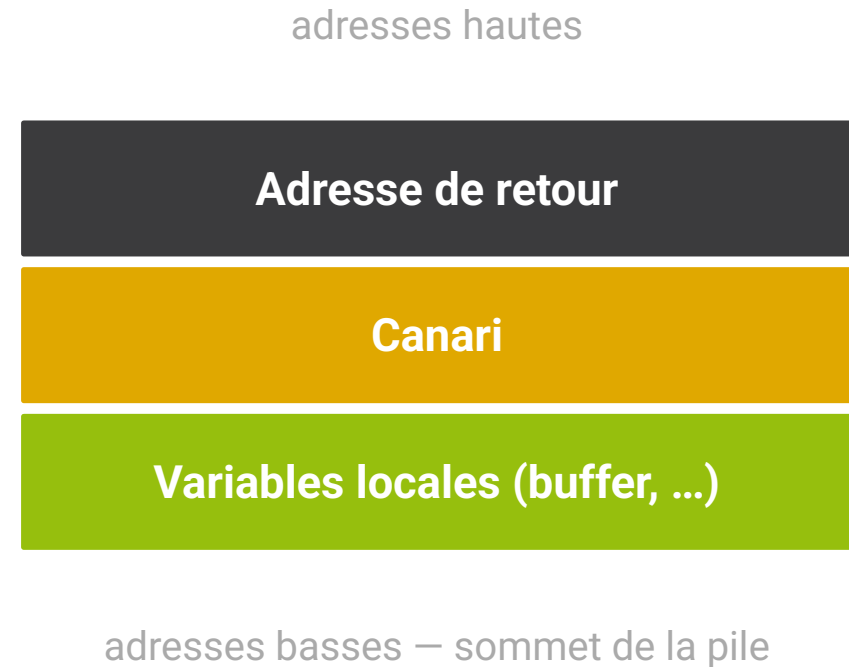


**IUT CLERMONT
AUVERGNE**

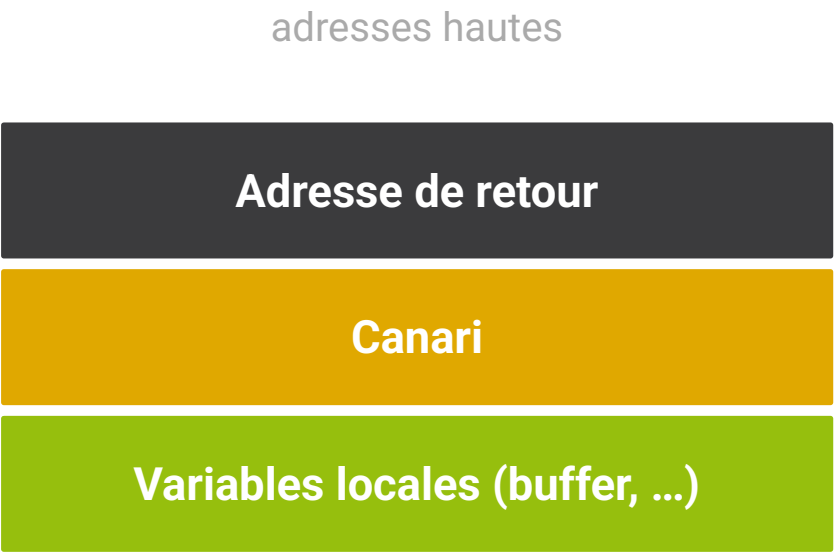
Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

8. Protections et durcissement

Le compilateur insère une valeur **aléatoire** entre les variables locales et l'adresse de retour, tirée au démarrage du programme.



Un débordement qui vise l'adresse de retour doit forcément **traverser** cette case en premier.



↑ **premier obstacle sur le chemin de l'adresse de retour**

Juste avant `ret`, l'épilogue relit le canari et le compare à sa valeur d'origine.

adresses hautes

Adresse de retour

Canari (écrasé)

Variables locales (buffer, ...)

adresses basses — sommet de la pile

X valeur différente de l'originale

Si les deux valeurs diffèrent, le programme s'arrête immédiatement (`__stack_chk_fail`) — **avant** d'exécuter `ret`.

⇒ Le canari protège l'adresse de retour, pas les variables voisines : les écraser reste possible.



adresses basses — sommet de la pile

 **arrêt du programme**

Non-eXecutable / Data Execution Prevention

Marque les pages de pile et de tas comme **non exécutables** au niveau du processeur (bit NX). Le processeur refuse d'exécuter la moindre instruction qui s'y trouve.

Conséquence directe : un shellcode injecté dans le buffer ne peut plus être exécuté, même si l'adresse de retour pointe dessus — d'où le détournement par ROP et ret2libc.

⇒ NX ne protège rien de l'écroulement de l'adresse de retour lui-même — il n'interdit que l'exécution des données écrites.

Address Space Layout Randomization

À chaque exécution, la pile, le tas et les bibliothèques partagées sont chargés à des adresses **aléatoires** — plus d'adresse fixe à viser pour un gadget ou une fonction `libc`.

Position-Independent Executable

Sans PIE, seul le reste (pile, tas, bibliothèques) est randomisé : le code du programme lui-même reste à une adresse fixe. PIE rend le binaire **lui-même** relogeable, pour que l'ASLR s'applique aussi à son propre code.

⇒ `cat /proc/sys/kernel/randomize_va_space` (0 = désactivé, 2 = activé) — un ASLR actif mais un binaire compilé sans PIE laisse encore le code du programme à une adresse prévisible.

RELRO — RELocation Read-Only

La table des symboles externes (GOT) est réécrite dynamiquement au premier appel de chaque fonction — une cible classique d'écrasement. RELRO la rend en lecture seule après le chargement : **partielle** la protège en partie, **complète** (`-z now`) dès le démarrage.

_FORTIFY_SOURCE

Remplace à la compilation certains appels (`memcpy`, `strcpy`, `sprintf`, ...) par des variantes qui vérifient, quand la taille du buffer est connue à la compilation, qu'elles ne débordent pas.

⇒ Deux protections orthogonales aux précédentes : elles ne portent ni sur la pile ni sur l'adressage, mais sur des cibles et des appels précis du binaire.

```

1  $ checksec --file=./prog
2  RELRO                STACK CANARY          NX                PIE
3  Full RELRO           Canary found          NX enabled        PIE enabled

```

Sans l'outil, les mêmes informations se lisent directement :

- canari : présence d'un appel à `__stack_chk_fail` (`objdump -d`)
- NX : `readelf -l prog` — segment `GNU_STACK` sans le drapeau `E`
- PIE : `readelf -h prog` — type `DYN` plutôt que `EXEC`

⇒ Avant d'attaquer un binaire, `checksec` — ce que ces protections autorisent ou interdisent détermine la technique utilisable.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

9. Sécurité Linux

Chaque fichier appartient à un utilisateur et à un groupe, et porte trois jeux de droits **lecture / écriture / exécution** — pour le propriétaire (**User**), le groupe (**Group**), et les autres (**Other**) :

```
1 $ ls -l rapport.txt
2 -rw-r----- 1 alice profs 4096 rapport.txt
```

`rw-` pour alice, `r--` pour le groupe `profs`, rien pour les autres.

Discrétionnaire

C'est le **propriétaire** du fichier qui décide qui y accède (`chmod`, `chown`) — pas une politique centrale. Le nouveau fichier hérite de droits par défaut, ajustés par l'`umask` du processus qui le crée.

⇒ Trois catégories seulement : impossible de donner un droit à **un** utilisateur précis sans le mettre dans le bon groupe.

Les listes de contrôle d'accès étendent UGO à des utilisateurs et groupes **arbitraires**, sur un même fichier :

```
1 $ setfacl -m u:bob:rw rapport.txt
2 $ getfacl rapport.txt
3 user::rw-
4 user:bob:rw-
5 group::r--
6 other::---
```

`bob` obtient un accès en lecture-écriture **sans** appartenir au groupe `profs` et sans devenir propriétaire.

⇒ Les ACL restent indexées par uid et gid — ce sont les mêmes identifiants que UGO, seulement plus nombreux et plus précis.

```
1 $ ls -l /usr/bin/passwd
2 -rwsr-xr-x 1 root root 68208 /usr/bin/passwd
```

Le **s** remplace le **x** du propriétaire : **setuid**. Le programme s'exécute avec les droits de son **propriétaire** (ici `root`), pas ceux de l'utilisateur qui le lance — indispensable pour que `passwd` puisse écrire dans `/etc/shadow`, inaccessible à un utilisateur normal.

⇒ Trois bits analogues : **setuid** (droits du propriétaire), **setgid** (droits du groupe), **sticky bit** (sur `/tmp`, seul le propriétaire d'un fichier peut le supprimer).

Élévation de privilèges

Un débordement de tampon dans un binaire **setuid root** ne corrompt plus seulement la mémoire d'un processus ordinaire : le code qui prend le contrôle du programme hérite des droits de **root**.

```
1 $ find / -perm -4000 2>/dev/null
```

Recense tous les binaires setuid du système — la première commande d'un audit de surface d'attaque locale.

⇒ Ce n'est pas la vulnérabilité qui change de nature, c'est ce qu'elle permet d'obtenir une fois exploitée.

`sudo` élève temporairement les privilèges d'un utilisateur **authentifié**, selon une politique écrite dans `/etc/sudoers` — à l'opposé du `setuid`, qui élève systématiquement, pour n'importe qui exécute le binaire.

```
1 # /etc/sudoers
2 alice ALL=(ALL) /usr/bin/systemctl restart nginx
```

alice peut relancer `nginx` en root, et uniquement cette commande.

Erreurs de configuration fréquentes

NOPASSWD: sur une commande sensible, un chemin avec un joker (*), ou une commande qui permet elle-même d'exécuter du code arbitraire (`sudo vim puis :!sh`) — GTF0Bins recense ces contournements binaire par binaire.

⇒ `sudo -l` liste ce qu'un utilisateur peut faire en élévation — le premier réflexe pour auditer une machine.

DAC repose entièrement sur la décision du propriétaire — un binaire compromis hérite de tous ses droits, sans exception. Le contrôle d'accès **obligatoire** (MAC) ajoute une politique centrale que même root ne contourne pas.

Deux implémentations Linux

- **SELinux** : chaque processus et fichier porte une étiquette (contexte) ; une politique explicite autorise ou refuse chaque interaction entre étiquettes — refus par défaut.
- **AppArmor** : un profil par exécutable, listant les fichiers et permissions autorisés — plus simple à écrire, moins granulaire.

⇒ Même un processus root confiné par une politique MAC reste limité aux actions que la politique autorise explicitement.

Un conteneur n'est pas une machine virtuelle : c'est un processus Linux ordinaire, auquel on restreint la **vue** et les **ressources**.

Namespaces — ce que le processus voit

PID (ses propres numéros de processus), `net` (sa propre pile réseau), `mnt` (son propre système de fichiers), `uts`, `ipc`, `user` — chacun isole une ressource du noyau, indépendamment des autres.

cgroups — ce que le processus consomme

Limite CPU, mémoire, E/S — empêche un processus (ou un groupe) de tout monopoliser.

⇒ `chroot` change uniquement la racine visible du système de fichiers — bien plus faible que les namespaces, et contournable sans droits root correctement retirés.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

10. Authentification locale

```
1 $ getent passwd alice
2 alice:x:1001:1001:Alice Martin:/home/alice:/bin/bash
```

Utilisateur : identifiant, UID, GID principal, nom complet, dossier personnel, interpréteur de commandes. Le `x` signifie que le hash du mot de passe n'est **plus** ici — il est déplacé dans `/etc/shadow`, lisible uniquement par `root`.

`/etc/passwd` doit rester lisible par tous (`ls`, `id`, ... en dépendent) ; séparer le hash dans un fichier à part évite de l'exposer par la même occasion.

⇒ Le choix de l'algorithme de hachage (bcrypt, Argon2, ...) a déjà été vu en C1 — ce qui change ici, c'est **où** et **comment** cette valeur est protégée sur le système.

Linux-PAM ne code aucune logique d'authentification lui-même : chaque application (`login`, `sudo`, `sshd`, ...) délègue à une **pile de modules** configurée dans `/etc/pam.d/`.

Quelques modules courants

- `pam_unix` : vérifie le mot de passe contre `/etc/shadow`
- `pam_faillock` : verrouille le compte après plusieurs échecs (remplace `pam_tally2`, retiré depuis Linux-PAM 1.5)
- `pam_nologin` : bloque les connexions non-root si `/etc/nologin` existe — **pas** lié à un mot de passe vide

⇒ Changer la politique d'authentification d'un service ne demande jamais de modifier son code : seulement sa configuration PAM.

Un mot de passe seul est un **unique** secret, copiable et rejouable en cas de fuite. Le MFA combine des facteurs de nature différente :

- ce que l'utilisateur **sait** — mot de passe, code PIN
- ce que l'utilisateur **possède** — téléphone, clé physique (FIDO2/U2F)
- ce que l'utilisateur **est** — biométrie

TOTP

Un secret partagé au moment de l'inscription, combiné à l'heure courante, produit un code à 6 chiffres qui change toutes les 30 secondes — c'est le principe derrière les applications d'authentification.

⇒ Voler un mot de passe ne suffit plus si un second facteur, d'une nature différente, est exigé.



**IUT CLERMONT
AUVERGNE**

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

11. Racine de confiance

Chaque étape du démarrage vérifie la **signature** de la suivante avant de lui céder l'exécution : firmware → bootloader → noyau.

Ce que ça empêche

Un rootkit installé au niveau du bootloader ou du firmware s'exécute **avant** le système d'exploitation — avant tout antivirus, avant toute protection logicielle. Secure Boot refuse de démarrer un maillon dont la signature ne correspond à aucune clé de confiance.

⇒ La confiance part d'une racine matérielle (immuable) et se propage par signatures — casser un seul maillon compromet tout ce qui suit.

Un firmware se met à jour — c'est aussi une porte d'entrée : accepter une mise à jour non authentifiée revient à laisser n'importe qui remplacer le code qui s'exécute avant l'OS.

SBSFU — Secure Boot and Secure Firmware Update

Le nouveau firmware est signé par le fabricant ; l'appareil vérifie la signature **avant** d'installer la mise à jour, et refuse toute image non conforme — y compris un firmware plus ancien et vulnérable (anti-rollback).

⇒ Même une mise à jour légitime doit être authentifiée : c'est un canal d'écriture privilégié vers le composant le plus bas niveau du système.

TEE — Trusted Execution Environment

Une zone d'exécution isolée du reste du système par le processeur lui-même (ex. ARM TrustZone) : même un système d'exploitation compromis ne peut pas lire ce qui s'y exécute.

TPM — Trusted Platform Module

Une puce séparée qui stocke des clés et calcule des empreintes de l'état de démarrage (**measured boot**) — utilisée pour attester qu'un système a démarré sans modification, ou pour chiffrer un disque (BitLocker, LUKS).

⇒ Point commun : déplacer la confiance hors du système d'exploitation, vers un composant plus petit et plus difficile à attaquer.

Les cellules mémoire DRAM sont si proches que lire **frénétiquement** une rangée peut perturber électriquement la rangée voisine, jusqu'à inverser un bit qu'aucune instruction n'a jamais ciblé directement.

Pourquoi c'est grave

Cette inversion de bit peut tomber dans une table de pages ou une structure de permissions — un attaquant sans le moindre droit d'écriture logique obtient une corruption mémoire **physique**.

⇒ Une attaque purement matérielle : aucune faille logicielle n'est exploitée, seule une propriété physique de la mémoire.

Une implémentation correcte peut fuiter de l'information par un canal que son modèle de sécurité ignore : le temps de calcul, la consommation électrique, le cache du processeur.

Exemple : comparaison de mot de passe

`strcmp` s'arrête au premier octet différent : comparer avec le bon mot de passe prend, en moyenne, un peu plus longtemps qu'avec un mot de passe complètement faux. Mesurer ce délai, octet par octet, permet de reconstruire le secret sans jamais le lire.

Meltdown et Spectre (2018) exploitent le même principe au niveau du processeur : l'exécution spéculative laisse une trace mesurable dans le cache, même quand le résultat spéculé est annulé.

⇒ Correct sur le papier n'est pas correct en pratique — l'implémentation elle-même peut trahir le secret qu'elle est censée protéger.

Questions ?