Rapport Technique
Coffre-fort Mémoire Sécurisé en Assembleur x86_64

Projet de Programmation Assembleur

MONCOMBLE Jules

4SI4

Master 1 - Sécurité Informatique
Année Universitaire 2024-2025

Table des matières

1 Introduction

Le projet que j'ai décidé de développer ce semestre est un coffre-fort mémoire sécurisé programmé entièrement en assembleur x86_64. Cette application, bien que conceptuellement simple, aborde des problématiques profondes liées à la sécurité mémoire, au chiffrement de données et à la programmation bas niveau. Ce rapport technique détaille l'ensemble des aspects du projet, des fondamentaux de l'assembleur aux choix d'implémentation, en passant par les défis rencontrés et leurs solutions.

J'ai choisi ce projet spécifique car il s'aligne parfaitement avec mon projet annuel de recherche sur le chiffrement de la RAM au niveau kernel que je compte développer en Rust. Bien que les approches et technologies soient différentes, ce travail en assembleur m'a fourni une compréhension fondamentale des mécanismes sous-jacents que j'exploiterai plus tard.

Le coffre-fort que j'ai implémenté permet à un utilisateur de stocker des chaînes de caractères en mémoire, protégées par un chiffrement XOR et accessibles uniquement via une interface en ligne de commande dédiée. Ce qui pourrait sembler trivial en langage de haut niveau devient un véritable défi technique lorsqu'on travaille au niveau le plus bas de la programmation.

2 Les fondamentaux de l'assembleur x86_64

2.1 Architecture des registres

L'un des premiers aspects à comprendre lorsqu'on programme en assembleur x86_64 est le rôle et l'utilisation des registres. Contrairement aux langages de haut niveau où l'on peut créer autant de variables que nécessaire, en assembleur, nous sommes limités à un ensemble fixe de registres :

- **Registres généraux** : RAX, RBX, RCX, RDX, etc.
- **Registres d'index** : RSI, RDI, etc.
- **Registres pointeurs** : RBP (base pointer), RSP (stack pointer), etc.
- **Registres de segments** : CS, DS, SS, etc.

Dans mon implémentation, j'ai dû planifier soigneusement l'utilisation de ces registres. Par exemple, pour les appels système Linux, RAX contient le numéro de l'appel système, RDI le premier argument, RSI le deuxième, etc. Pour l'opération de lecture depuis stdin, j'ai utilisé :

```
mov rax, 0   ; sys_read
mov rdi, 0   ; stdin
mov rsi, input_buffer
mov rdx, 1023 ; taille maximale
syscall
```

Cette utilisation précise des registres m'a obligé à adopter une discipline rigoureuse dans mon code, sachant que chaque opération peut affecter l'état global du programme.

2.2 Sections du programme

Mon programme est divisé en trois sections principales, chacune ayant un rôle spécifique :

— **.data** : Contient toutes les données initialisées (messages, chaînes de commande, etc.)

— **.bss** : Déclare l'espace pour les variables non initialisées (buffers, etc.)

— **.text** : Contient le code exécutable (instructions, fonctions, etc.)

Cette organisation n'est pas juste une convention, elle reflète la façon dont le programme est chargé en mémoire par le système d'exploitation, avec des permissions différentes selon les sections (lecture, écriture, exécution).

2.3 Modèle d'exécution linéaire

Contrairement aux langages structurés où le flux d'exécution est clairement défini par des structures de contrôle, en assembleur, l'exécution est linéaire avec des sauts conditionnels et inconditionnels. Cela m'a obligé à réfléchir différemment à la logique du programme :

```
main_loop:
  ; Afficher l'invite de commande
  mov rsi, prompt
  call print_string

  ; Lire l'entr e utilisateur
  call read_line

  ; Traiter la commande
  mov rdi, input_buffer
  call parse_command

  ; Continuer la boucle
  jmp main_loop
```

Cette boucle principale illustre bien le contrôle de flux en assembleur, avec l'instruction `jmp` qui crée une boucle infinie jusqu'à ce que la commande `quitter` soit traitée.

3 Gestion de la mémoire et sécurité

3.1 Allocation de mémoire statique

Pour ce projet, j'ai choisi une approche d'allocation mémoire statique en définissant des buffers dans la section .bss :

```
section .bss
  input_buffer  resb 1024 ; Tampon d'entr e
  secure_memory  resb 4096 ; Zone m moire s curis e
  data_buffer   resb 1024 ; Tampon pour les donn es
  key_buffer   resb 64  ; Tampon pour la cl
```

```
data_length    resq 1    ; Longueur des donn es stock es
key_length     resq 1    ; Longueur de la cl  de chiffrement
cmd_buffer     resb 32   ; Tampon pour la commande
```

Cette approche, bien que simple, présente des avantages et inconvénients :

Avantages et Inconvénients

Avantages :

- Allocation déterministe (pas de risque d'échec d'allocation)
- Simplicité d'implémentation
- Performances prévisibles

Inconvénients :

- Taille fixe (gaspillage potentiel d'espace)
- Pas de véritable isolation ou protection mémoire au niveau hardware

3.2 Tentative d'utilisation de mmap

Dans mes premières versions du projet, j'avais tenté d'utiliser l'appel système `mmap` pour créer une zone mémoire véritablement isolée avec des protections spécifiques :

```
allocate_secure_memory:
  ; mmap(NULL, size, PROT_READ|PROT_WRITE,
   MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
  mov rax, 9          ; sys_mmap
  xor rdi, rdi        ; addr = NULL (laisser le noyau choisir)
  mov rsi, secure_mem_size ; length = 4KB
  mov rdx, 0x3        ; prot = PROT_READ | PROT_WRITE
  mov r10, 0x22       ; flags = MAP_PRIVATE | MAP_ANONYMOUS
  mov r8, -1          ; fd = -1 (pas de fichier)
  xor r9, r9          ; offset = 0
  syscall
```

Cette approche aurait permis une véritable isolation mémoire, mais j'ai rencontré des problèmes de segmentation. Après analyse, j'ai découvert que ces problèmes étaient liés à des erreurs subtiles dans la gestion des pointeurs retournés par mmap et dans le passage de ces pointeurs entre fonctions.

Bien que j'aie finalement opté pour une approche plus simple avec des buffers statiques, cette expérience m'a beaucoup appris sur les défis de la gestion mémoire au niveau système.

3.3 Problèmes de segmentation et leur résolution

Les erreurs de segmentation (segfaults) ont été l'un des défis majeurs de ce projet. Voici les principales causes que j'ai identifiées et leurs solutions :

1. **Accès à des adresses mémoire non allouées** : J'ai résolu ce problème en vérifiant systématiquement les limites des buffers avant d'y accéder.

2. **Corruption de la pile** : Les appels de fonction en assembleur nécessitent une gestion manuelle de la pile. Des erreurs comme ne pas restaurer correctement la pile peuvent causer des corruptions. J'ai adopté une approche rigoureuse de sauvegarde/restauration des registres :

```
string_equals:
  push rdi
  push rsi

  ; ... corps de la fonction ...

  pop rsi
  pop rdi
  ret

```

3. **Déréférencement de pointeurs null** : J'ai ajouté des vérifications systématiques avant de déréférencer des pointeurs :

```
find_command_data:
  ; ... autre code ...

  .end_of_input:
    cmp rsi, 0
    je .error_handler ;  viter  de d r f rencer un pointeur null
    mov byte [rsi], 0 ; S curis  uniquement si rsi n'est pas null
    ret

```

4. **Débordements de buffer** : J'ai implémenté des limites strictes sur toutes les opérations de copie :

```
.copy_cmd_loop:
  mov al, [rsi + rcx]
  ; ... autre code ...
  mov [rdi + rcx], al
  inc rcx
  cmp rcx, 31      ; Limite stricte pour  viter  les d bordements
  jge .end_of_cmd

```

Ces problèmes m'ont rappelé que l'assembleur ne fournit aucune protection contre les erreurs courantes qui sont automatiquement gérées dans les langages de haut niveau. Cette expérience directe avec les segfaults sera précieuse pour mon travail au niveau kernel, où de telles erreurs peuvent avoir des conséquences catastrophiques.

4 Système de chiffrement

4.1 Fondements du chiffrement XOR

J'ai choisi d'implémenter un chiffrement XOR pour ce projet. L'opération XOR (ou exclusif) a des propriétés mathématiques intéressantes pour le chiffrement :

- $A \oplus 0 = A$
- $A \oplus A = 0$
- $A \oplus B = B \oplus A$ (commutativité)
- $(A \oplus B) \oplus C = A \oplus (B \oplus C)$ (associativité)
- $(A \oplus B) \oplus B = A$ (annulation)

La dernière propriété est particulièrement utile pour le chiffrement : si on chiffre un texte A avec une clé B pour obtenir C ($A \oplus B = C$), on peut retrouver A en appliquant à nouveau XOR avec B ($C \oplus B = A$).

4.2 Implémentation détaillée

Mon implémentation du chiffrement est basée sur l'application cyclique de la clé sur les données :

```
encrypt_data:
  xor rcx, rcx ; Compteur
  mov rax, [data_length]

.encrypt_loop:
  cmp rcx, rax
  jge .done

  ; Obtenir le caract re    chiffrer
  mov dl, [data_buffer+rcx]

  ; Calculer l'index de la cl  (cyclique)
  push rax
  push rdx

  mov rax, rcx
  xor rdx, rdx
  mov rsi, [key_length]
  div rsi    ; RDX = RCX % key_length

  ; Appliquer XOR avec la cl
  mov al, [key_buffer+rdx]
  pop rdx
  xor dl, al   ; XOR le caract re avec la cl

  ; Stocker le r sultat
  mov [secure_memory+rcx], dl

  pop rax
```

```
30    inc rcx
31    jmp .encrypt_loop
```

Quelques points techniques à noter :

1. L'utilisation de l'instruction `div` pour calculer le modulo et appliquer la clé cycliquement.
2. La sauvegarde/restauration des registres avec `push`/`pop` pour préserver les valeurs pendant les calculs intermédiaires.
3. L'opération XOR elle-même est réalisée avec l'instruction `xor dl, al`.

4.3 Limites de sécurité et améliorations possibles

Bien que fonctionnel, ce chiffrement XOR présente des limitations importantes en termes de sécurité :

1. **Vulnérabilité à l'analyse fréquentielle** : Si un attaquant connaît une partie du texte en clair, il peut facilement déduire la clé correspondante.
2. **Réutilisation de clé problématique** : Si la même clé est utilisée pour chiffrer plusieurs messages, des attaques statistiques deviennent possibles (attaque XOR connue sous le nom de "crib dragging").
3. **Absence de diffusion** : Contrairement aux algorithmes modernes comme AES, XOR ne propage pas les changements dans l'ensemble du texte chiffré.
4. **Pas d'authentification** : Le chiffrement XOR ne permet pas de vérifier l'intégrité des données.

Pour un système de production réel, j'aurais utilisé un algorithme comme AES-GCM qui offre à la fois confidentialité et authentification. Cependant, l'implémentation d'un tel algorithme en assembleur pur dépassait largement le cadre de ce projet.

5 Interface utilisateur et analyse des commandes

5.1 Boucle principale et traitement des commandes

La boucle principale de mon programme est relativement simple mais efficace :

```
1  main_loop:
2    ; Afficher l'invite de commande
3    mov rsi, prompt
4    call print_string
5
6    ; Lire l'entr e utilisateur
7    call read_line
8
9    ; Extraire la commande
10   call extract_command
11
12   ; V rifier la commande
13   call process_command
14
```

```
    ; Continuer la boucle
    jmp main_loop
```

Cette approche m'a permis de structurer le code de manière modulaire, chaque fonction ayant une responsabilité bien définie.

5.2 Analyse lexicale des commandes

L'extraction et l'analyse des commandes ont été parmi les aspects les plus délicats à implémenter. J'ai divisé ce processus en deux étapes :

1. **Extraction de la commande** : Isoler le premier mot de l'entrée utilisateur.
2. **Traitement des arguments** : Pour les commandes qui nécessitent des arguments (comme `stocker` ou `cle`).

L'extraction de la commande est réalisée par cette fonction :

```
extract_command:
    mov rsi, input_buffer   ; Source
    mov rdi, cmd_buffer     ; Destination pour la commande

    ; Passer les espaces initiaux
    .skip_initial_spaces:
        cmp byte [rsi], ' '
        jne .extract_cmd
        inc rsi
        jmp .skip_initial_spaces

    ; Extraire la commande
    .extract_cmd:
        xor rcx, rcx        ; Compteur de caract res

    .copy_cmd_loop:
        mov al, [rsi + rcx] ; Prochain caract re

        ; V rifier la fin de la commande
        cmp al, 0           ; Fin de cha ne?
        je .end_of_cmd
        cmp al, ' '         ; Espace?
        je .end_of_cmd

        ; Copier le caract re dans le tampon de commande
        mov [rdi + rcx], al
        inc rcx
        cmp rcx, 31         ; Limite de taille de commande
        jge .end_of_cmd
        jmp .copy_cmd_loop

    .end_of_cmd:
        mov byte [rdi + rcx], 0 ; Terminer la commande par null
        ret
```

Pour les arguments, j'utilise une fonction dédiée :

```
find_command_data:
  ; Passer les espaces initiaux
  .skip_initial_spaces:
    cmp byte [rsi], ' '
    jne .find_first_space
    inc rsi
    jmp .skip_initial_spaces

  ; Trouver le premier espace apr s la commande
  .find_first_space:
    cmp byte [rsi], 0  ; Fin de cha ne?
    je .end_of_input
    cmp byte [rsi], ' ' ; Espace?
    je .found_space
    inc rsi
    jmp .find_first_space

  .found_space:
    ; Passer les espaces apr s la commande
    .skip_spaces_after_cmd:
      inc rsi
      cmp byte [rsi], ' '
      je .skip_spaces_after_cmd
      ret

  .end_of_input:
    mov byte [rsi], 0  ; Assurer la fin
    ret
```

Ces fonctions sont volontairement détaillées et méthodiques pour éviter les erreurs d'accès mémoire qui causaient des segfaults dans mes premières versions.

5.3 Comparaison de chaînes

La comparaison de chaînes, une opération triviale dans les langages de haut niveau, devient complexe en assembleur. Voici mon implémentation :

```
string_equals:
  push rdi
  push rsi

.compare_loop:
  mov al, [rdi]
  mov bl, [rsi]

  ; Si les caract res sont diff rents
  cmp al, bl
  jne .not_equal

  ; Si on a atteint la fin des deux cha nes
  test al, al
```

```
    jz .equal

    ; Avancer au caract re suivant
    inc rdi
    inc rsi
    jmp .compare_loop

.equal:
    mov rax, 1   ; 1 =  gal
    pop rsi
    pop rdi
    ret

.not_equal:
    xor rax, rax ; 0 = diff rent
    pop rsi
    pop rdi
    ret
```

Cette fonction compare les chaînes caractère par caractère et retourne 1 si elles sont identiques, 0 sinon. J'ai choisi de retourner 1 pour "vrai" par convention, bien que certains codes préfèrent 0 pour succès (comme les valeurs de retour de programme).

5.4 Gestion des erreurs utilisateur

Une partie importante de l'interface utilisateur est la gestion des erreurs. J'ai implémenté plusieurs vérifications :

1. **Commandes inconnues** : Si la commande n'est pas reconnue, un message d'erreur est affiché.
2. **Données manquantes** : Si l'utilisateur tente de lire des données alors qu'aucune n'est stockée.
3. **Arguments manquants** : Si l'utilisateur omet des arguments nécessaires pour les commandes comme `stocker` ou `cle`.

Ces vérifications améliorent considérablement l'expérience utilisateur en fournissant des feedbacks clairs et en évitant les comportements inattendus.

6 Manipulations de chaînes et entrées/sorties

6.1 Lecture depuis stdin

La lecture des entrées utilisateur est réalisée avec l'appel système `read` :

```
read_line:
    mov rax, 0   ; sys_read
    mov rdi, 0   ; stdin
    mov rsi, input_buffer
    mov rdx, 1023
    syscall

```

```
  ; Ajouter terminateur null
  mov byte [input_buffer+rax], 0

  ; Remplacer newline par null si pr sent
  cmp rax, 0
  je .done    ; Si rien n'a   t   lu

  dec rax
  cmp byte [input_buffer+rax], 10 ; Newline?
  jne .done
  mov byte [input_buffer+rax], 0

.done:
  ret
```

Notez comment je gère explicitement le terminateur null et le caractère de nouvelle ligne, des détails que les langages de haut niveau gèrent automatiquement.

6.2 Écriture vers stdout

Pour l'affichage, j'utilise l'appel système `write` :

```
print_string:
  push rsi
  xor rdx, rdx ; Compteur pour la longueur

.length_loop:
  cmp byte [rsi+rdx], 0
  je .print
  inc rdx
  jmp .length_loop

.print:
  mov rax, 1  ; sys_write
  mov rdi, 1  ; stdout
  ; RSI et RDX contiennent d j  la cha ne et la longueur
  syscall

  pop rsi
  ret
```

Je dois d'abord calculer la longueur de la chaîne, car contrairement aux fonctions comme `printf` en C, l'appel système `write` nécessite que l'on spécifie explicitement la longueur.

6.3 Calcul de longueur de chaîne

Calculer la longueur d'une chaîne terminée par null est une opération fondamentale en manipulation de chaînes :

```
; Calcul de longueur int gr  dans print_string
.length_loop:
```

```
    cmp byte [rsi+rdx], 0
    je .print
    inc rdx
    jmp .length_loop
```

Cette fonction incrémente un compteur jusqu'à ce qu'elle rencontre un octet nul, signalant la fin de la chaîne.

6.4 Gestion des terminateurs null

La gestion correcte des terminateurs null est cruciale pour éviter les débordements de mémoire. Dans mon code, j'ai été particulièrement vigilant :

```
; Ajout de terminateur null apr s lecture
mov byte [input_buffer+rax], 0

; Terminaison de la commande extraite
mov byte [rdi + rcx], 0 ; Terminer la commande par null
```

Cette gestion explicite des terminateurs null est une autre illustration de la différence entre la programmation assembleur et les langages de haut niveau, où les chaînes sont généralement traitées comme des types de données abstraits avec une gestion automatique des terminateurs.

7 Défis techniques spécifiques et solutions

7.1 Problème : Segfaults lors de l'extraction des commandes

Problème et Solution

Symptôme : Le programme se terminait abruptement avec un segfault lorsque j'entrais certaines commandes.
Cause : Dans ma première implémentation, j'utilisais une approche de modification in-place du buffer d'entrée pour isoler la commande, en remplaçant le premier espace par un null. Cette approche fonctionnait dans certains cas mais échouait dans d'autres, notamment quand l'entrée ne contenait pas d'espace ou contenait uniquement des espaces.
Solution : J'ai complètement révisé l'approche en créant un buffer dédié pour la commande et en copiant explicitement les caractères jusqu'à un espace ou la fin de la chaîne :

```
extract_command:
    mov rsi, input_buffer   ; Source
    mov rdi, cmd_buffer     ; Destination d di  e

    ; ... reste de la fonction ...
```

Cette approche plus robuste évite les modifications in-place risquées et gère correctement tous les cas limites.

7.2 Problème : Débordements de buffer lors de la copie de chaînes

Problème et Solution

Symptôme : Comportements instables et crashes aléatoires après certaines opérations.
Cause : Absence de vérification des limites lors de la copie de chaînes, permettant potentiellement d'écrire au-delà des buffers alloués.
Solution : Ajout de vérifications strictes des limites dans toutes les opérations de copie :

```
.copy_loop:
  test rdx, rdx
  jz .done
  dec rdx
  mov al, [rsi+rdx]
  mov [rdi+rdx], al
  jmp .copy_loop
```

Cette approche garantit que je ne copie jamais plus de caractères que la taille du buffer destination.

7.3 Problème : Corruptions de pile lors des appels de fonction

Problème et Solution

Symptôme : Comportements inexplicables après le retour de certaines fonctions.
Cause : Utilisation incorrecte de la pile, notamment absence de préservation de certains registres modifiés par les fonctions.
Solution : Adoption d'une convention d'appel rigoureuse avec sauvegarde/restauration systématique des registres modifiés :

```
my_function:
  push rbx
  push rcx
  push rdx
  ; ... corps de la fonction ...
  pop rdx
  pop rcx
  pop rbx
  ret
```

Cette discipline est essentielle en assembleur car il n'y a pas de mécanisme automatique pour préserver l'état des registres comme dans les langages compilés.

7.4 Problème : Difficultés avec les opérations arithmétiques sur les chaînes

Problème et Solution

Symptôme : Résultats incorrects lors de l'application cyclique de la clé de chiffrement.
Cause : Erreurs dans la gestion de la division et du modulo pour calculer l'index cyclique de la clé.
Solution : Utilisation correcte de l'instruction `div` qui divise RDX :RAX par l'opérande, en s'assurant que RDX est initialisé à 0 avant l'opération :

```
mov rax, rcx
xor rdx, rdx  ; Crucial pour  viter  une exception de division
mov rsi, [key_length]
div rsi       ; RDX = RCX % key_length
```

8 Lien avec mon projet annuel de chiffrement RAM au niveau kernel

8.1 Concepts fondamentaux transposables

Mon projet actuel, bien que simplifié, partage plusieurs concepts fondamentaux avec mon projet annuel de chiffrement RAM au niveau kernel :

1. **Isolation mémoire** : Dans mon projet actuel, j'ai simulé une isolation en utilisant des buffers dédiés. Dans le projet kernel, j'utiliserai les mécanismes de protection mémoire du processeur via la MMU.
2. **Chiffrement/déchiffrement à la volée** : Le concept d'application de chiffrement XOR sur des données en mémoire est similaire à ce que je ferai dans le projet kernel, bien que j'utiliserai des algorithmes plus sophistiqués.
3. **Gestion des clés** : La gestion d'une clé de chiffrement, sa validation et son utilisation cyclique sont des concepts que je réutiliserai.
4. **Manipulation directe de la mémoire** : Travailler au niveau byte, comprendre les alignements et les accès mémoire sont des compétences directement transférables.

8.2 Différences majeures

Plusieurs différences significatives existent entre ce projet d'assembleur et mon projet annuel kernel :

1. **Contexte d'exécution** : Mon projet actuel s'exécute en espace utilisateur, tandis que le projet kernel s'exécutera en ring 0 avec des privilèges élevés, ce qui permet un contrôle total du matériel mais impose aussi des contraintes strictes.
2. **Échelle et portée** : Mon projet actuel protège une petite zone mémoire dans une seule application, alors que le projet kernel visera à chiffrer toute la mémoire RAM du système.

3. **Complexité** : Le projet kernel devra gérer les interactions avec le MMU, les changements de contexte, les accès DMA, et bien d'autres aspects complexes absents de mon projet actuel.
4. **Langage et outils** : L'utilisation de Rust pour le projet kernel offrira des garanties de sécurité mémoire statiques, contrairement à l'assembleur où tout doit être vérifié manuellement.

8.3 Architecture cible pour le projet kernel

Mon projet annuel vise à développer un module kernel qui intercepte les accès mémoire pour chiffrer/déchiffrer les données à la volée. L'architecture envisagée comprend :

1. **Hooks du gestionnaire de mémoire** : Interception des opérations de pagination et de swap.
2. **Cryptographie efficace** : Implémentation d'AES-NI ou ChaCha20 pour des performances optimales.
3. **Gestion sécurisée des clés** : Utilisation potentielle d'un TPM ou d'enclaves sécurisées.
4. **Compatibilité** : Support des fonctionnalités Linux standard comme fork, mmap, etc.
5. **Performance** : Minimisation de l'impact sur les performances système via des techniques comme le caching intelligent et le chiffrement sélectif.

8.4 Avantages du Rust pour l'implémentation kernel

Pour mon projet annuel, j'ai choisi Rust plutôt que l'assembleur ou le C pour plusieurs raisons :

1. **Sécurité mémoire** : Le système de propriété et d'emprunt de Rust élimine les classes entières de bugs mémoire à la compilation.
2. **Abstractions zéro-coût** : Rust permet d'écrire du code de haut niveau qui se compile vers un code machine aussi efficace que du C ou de l'assembleur optimisé.
3. **Concurrence sûre** : Les garanties de Rust concernant la concurrence sont précieuses dans un environnement kernel multitâche.
4. **FFI sans coût** : Rust s'intègre parfaitement avec le C, permettant d'interagir facilement avec le code kernel existant.
5. **Communauté et écosystème** : L'écosystème Rust pour le développement kernel est en pleine croissance, avec des projets comme Redox OS démontrant sa viabilité.

8.5 Enseignements spécifiques transférables

Mon travail en assembleur m'a fourni plusieurs enseignements directement applicables à mon projet kernel :

1. **Compréhension approfondie des registres et instructions x86_64** : Utile pour déboguer le comportement du kernel et optimiser les performances.
2. **Familiarité avec les appels système Linux** : Base pour comprendre l'implémentation des services kernel.

3. **Expérience avec les erreurs de segmentation** : Précieuse pour anticiper et éviter les panics kernel.
4. **Manipulation bit-à-bit** : Essentielle pour les opérations cryptographiques efficaces.
5. **Discipline de programmation défensive** : Cruciale dans un environnement kernel où les erreurs peuvent compromettre tout le système.

9 Améliorations futures

Si je devais poursuivre le développement de ce projet, voici les améliorations que j'envisagerais :

9.1 Améliorations cryptographiques

1. **Implémentation d'AES** : Remplacer XOR par un algorithme cryptographiquement sûr comme AES-128 en mode CBC ou GCM.
2. **Dérivation de clé sécurisée** : Utiliser une fonction comme PBKDF2 ou Argon2 pour dériver une clé cryptographiquement forte à partir du mot de passe.
3. **Vecteurs d'initialisation (IV)** : Ajouter un IV aléatoire pour chaque opération de chiffrement afin d'éviter les attaques par motif.
4. **HMAC** : Ajouter une vérification d'intégrité via HMAC pour détecter toute modification non autorisée des données chiffrées.

9.2 Améliorations de la gestion mémoire

1. **Véritable isolation mémoire** : Utiliser correctement mmap avec les protections PROT_READ|PROT_WRITE et les flags MAP_PRIVATE|MAP_ANONYMOUS.
2. **Effacement sécurisé** : Implémenter un effacement sécurisé (zeroization) de la mémoire sensible après utilisation.
3. **Allocation dynamique** : Permettre des buffers de taille variable selon les besoins, évitant le gaspillage d'espace.
4. **Protection contre les accès concurrents** : Bien que limité en assembleur pur, ajouter des verrouillages basiques pour les accès concurrents.

9.3 Extensions fonctionnelles

1. **Persistance** : Ajouter la possibilité de sauvegarder/charger des données chiffrées depuis un fichier.
2. **Interface enrichie** : Commandes supplémentaires comme `info` pour afficher des informations sur les données stockées.
3. **Sessions multiples** : Permettre la gestion de plusieurs "coffres" avec des clés différentes.
4. **Historique de commandes** : Ajouter une fonctionnalité pour rappeler les commandes précédentes.

10 Réflexion sur le processus d'apprentissage

10.1 Valeur pédagogique de l'assembleur

Travailler en assembleur a été une expérience incroyablement formatrice. Cette programmation au plus bas niveau m'a forcé à comprendre des mécanismes fondamentaux souvent abstraits dans les langages de haut niveau :

1. **Gestion explicite de la mémoire** : Chaque allocation, accès, et libération doit être gérée manuellement.
2. **Contrôle de flux nu** : Sans structures comme les boucles `for` ou `while`, j'ai dû implémenter explicitement toutes les vérifications et sauts.
3. **Absence d'abstraction** : Pas de structures de données prédéfinies, de gestion d'exceptions, ou d'autres commodités des langages modernes.
4. **Optimisation manuelle** : Chaque instruction compte, ce qui m'a poussé à réfléchir constamment à l'efficacité.
5. **Débogage de bas niveau** : Sans outils sophistiqués, j'ai dû souvent examiner les registres et la mémoire brute pour comprendre les bugs.

Ces contraintes ont été initialement frustrantes mais finalement révélatrices, me permettant de toucher du doigt ce qui se passe réellement dans l'ordinateur.

10.2 Apprentissages inattendus

Outre les compétences techniques en assembleur, ce projet m'a apporté plusieurs apprentissages inattendus :

1. **L'importance de la planification** : En assembleur, une conception initiale rigoureuse est essentielle pour éviter des refactorisations complexes plus tard.
2. **La valeur de la documentation** : Sans types ou interfaces déclaratives, les commentaires et la documentation du code deviennent cruciaux.
3. **L'équilibre entre simplicité et robustesse** : J'ai appris à trouver le compromis entre un code minimal et une gestion complète des cas limites.
4. **L'humilité face à la complexité** : Ce qui semble trivial dans un langage de haut niveau peut être étonnamment complexe en assembleur, une leçon d'humilité pour tout développeur.

11 Conclusion

Ce projet de coffre-fort mémoire sécurisé en assembleur a été bien plus qu'un simple exercice académique. Il représente une étape fondamentale dans mon parcours vers la compréhension profonde des mécanismes de sécurité au niveau système.

En partant d'une idée conceptuellement simple – stocker des données chiffrées en mémoire – j'ai été confronté à une multitude de défis techniques qui m'ont forcé à approfondir ma compréhension de l'architecture x86_64, de la gestion mémoire, et des principes fondamentaux de la sécurité informatique.

Les erreurs de segmentation, les débordements de buffer, et les corruptions de pile que j'ai rencontrés et surmontés ne sont pas simplement des obstacles techniques, mais

des leçons précieuses sur la fragilité des systèmes informatiques et l'importance d'une programmation défensive rigoureuse.

Ce travail en assembleur m'a fourni une fondation solide pour mon projet annuel de chiffrement RAM au niveau kernel en Rust. Bien que les technologies et approches diffèrent, les principes fondamentaux restent les mêmes : isoler, protéger, et vérifier. Ma compréhension approfondie des mécanismes bas niveau me permettra de concevoir une solution plus robuste et efficiente.

En définitive, ce projet illustre parfaitement pourquoi la compréhension des couches basses d'un système est essentielle pour quiconque souhaite travailler sur la sécurité des systèmes. Comme on construit une maison en commençant par les fondations, j'ai commencé par l'assembleur pour construire ma compréhension de la sécurité mémoire.

La route vers un système de chiffrement RAM complet au niveau kernel est encore longue, mais grâce à ce projet, j'ai maintenant une carte plus précise du territoire à explorer, des défis à surmonter, et des opportunités à saisir. Cette pierre posée sur mon chemin est modeste, mais indispensable pour l'édifice que je souhaite construire.

Références

[1] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2021.

[2] The Rust Team, *The Rust Programming Language*, https ://doc.rust-lang.org/book/, 2023.

[3] Ferguson, N., Schneier, B., & Kohno, T., *Cryptography Engineering : Design Principles and Practical Applications*, Wiley, 2010.

[4] Love, R., *Linux Kernel Development*, Addison-Wesley Professional, 2010.

[5] Carter, P., *PC Assembly Language*, 2020.