

## SAE - ASSEMBLEUR

### **Note**

Ce projet est à réaliser dans le cadre de la SAÉ suivante :

- SAÉ.Cyber.03 - Administrer et Surveiller un système d'information sécurisé.

### Compétences cibles

- Surveiller un système sécurisé.
- Développer un système sécurisé.

### Compétences ciblées

- Comprendre l'assembleur
- Coder en assembleur
- Reverse engineering

### Livrables attendus

- Former son équipe et fournir un planning prévisionnel du projet, à rendre la première semaine, pour le reste la date de rendu vous sera communiquée ultérieurement.
- Un lien vers l'archive demandée.
- Une démo fonctionnelle à montrer lors de la soutenance.
- Un document PDF d'environ 20 à 30 pages, de préférence rédigé en LATEX(pas obligatoire), montrant votre démarche dans les deux parties.
- La démo (20 min max) sera présentée lors de la soutenance suivie d'une session de question/réponse (10 à 15 min). Pas de diapositives pour ce projet.



## Sommaire :

<b>Introduction</b>	<b>3</b>
<b>Gestion de Version avec Git dans le Projet</b>	<b>4</b>
<b>L'organisation :</b>	<b>5</b>
<b>Première partie : Super libc</b>	<b>6</b>
Introduction à la Première Partie - s_libc : Une Alternative Assembleur à la Libc	6
Bibliothèque s_string.s - Manipulation de Chaînes de Caractères en Assembleur	7
Bibliothèque s_maths.s - des fonctions mathématiques en assembleur.	8
Bibliothèque s_stdio.s - Opérations de messages.	8
Bibliothèque s_stdlib.s - fonctions utilitaires courantes et générales.	9
<b>Deuxième partie - Crackmes :</b>	<b>10</b>
Deviner le mot de passe (challenge 1) :	10
Deviner le mot de passe (challenge 2) :	13
Deviner le mot de passe (challenge 3) :	15
Deviner le mot de passe avec Cutter (challenge 4) :	16
Deviner le mot de passe avec Cutter (challenge 5) :	16
Deviner le mot de passe avec Cutter (challenge 6) :	21
Deviner le mot de passe avec Cutter (challenge 7) :	27
Deviner le mot de passe avec Cutter (challenge 8) :	31

## Introduction

L'assembleur est un langage de programmation de bas niveau qui permet de contrôler directement le processeur d'un ordinateur. Il est souvent utilisé pour optimiser les performances des programmes ou pour réaliser de l'ingénierie inverse sur des logiciels dont le code source n'est pas disponible.

Dans le cadre de la SAÉ du Hacker, nous avons réalisé deux projets en utilisant le langage assembleur : le premier consistait à réécrire des fonctions de la bibliothèque standard C et en faire une nouvelle librairie "**Super libC**", et le second à résoudre des challenges de **CrackMeS**, qui sont des programmes conçus en C pour tester nos compétences.

Ces projets nous ont permis de développer nos connaissances sur le fonctionnement interne des programmes informatiques, ainsi que sur les techniques de sécurité et de protection du code.

Ce rapport présente les objectifs, la méthodologie et les résultats de ces deux projets, ainsi que les difficultés rencontrées et les perspectives d'amélioration. Il est structuré en deux parties :

- **la première partie** décrit le projet de la **super libC**, qui consiste à implémenter des fonctions de la bibliothèque standard C en assembleur
- **la deuxième partie** décrit le projet de désassemblage, qui consiste à analyser et à cracker des programmes de type CrackMeS.

Chaque partie présente le contexte, le travail réalisé, les outils utilisés et les conclusions tirées.

# Gestion de Version avec Git dans le Projet



La gestion de version et la collaboration est une pratique essentielle dans le développement logiciel moderne, permettant de suivre les modifications apportées au code source tout au long du projet. Dans le cadre du projet "La Carotte Electronique" à l'IUT de Vélizy, nous avons utilisé Git comme système de gestion de versions distribuées.

Notre Github : <https://github.com/ceZarMax/I.U.T-3rd-Year-SAE-Carte-a-puce>

## **Qu'est-ce que Git ?**

Git est un système de gestion de versions décentralisé, créé par Linus Torvalds. Il permet de suivre les changements dans le code source, de collaborer efficacement avec d'autres développeurs, et de gérer les différentes versions du projet.

## **Utilisation de Git dans le Projet :**

L'utilisation de Git dans le projet a permis une collaboration fluide entre les membres de l'équipe. Chaque aspect du développement, des ajustements de code aux nouvelles fonctionnalités, a été suivi et commenté grâce à Git.

## **Avantages de l'utilisation de Git :**

- Historique des Modifications : Git a enregistré chaque modification apportée au code, facilitant le suivi des évolutions du projet.
- Collaboration Efficace : Les membres de l'équipe ont pu travailler simultanément sur différentes parties du projet sans conflits majeurs, grâce aux fonctionnalités de branches et fusion de Git.
- Sauvegardes Régulières : Les "commits" réguliers ont assuré des sauvegardes fréquentes du code, offrant une sécurité contre la perte de données.

## **Plateforme d'Hébergement :**

Le dépôt Git du projet a été hébergé sur une plateforme en ligne, offrant un accès centralisé au code source.

En conclusion, l'utilisation de Git dans le projet "La Carotte Electronique" a grandement contribué à la réussite du développement logiciel en assurant une collaboration harmonieuse, un suivi précis des modifications, et une gestion efficace des versions du code source.

## L'organisation :



Nous nous sommes réparties les librairies à faire et les crackmes.  
Chacun gérait son temps comme il l'entendait tant que le travail était fait.

## Première partie : Super libc

### Introduction à la Première Partie - s\_libc : Une Alternative Assembleur à la Libc

Dans le cadre de ce projet, la première partie, nommée s\_libc, se présente comme une bibliothèque C optimisée, développée en langage assembleur x86\_32bits. L'objectif de cette partie est d'offrir une alternative performante à la bibliothèque standard du langage C (libc).

#### **Caractéristiques Principales :**

- 1. Optimisation des Performances** : Les fonctions de la libc ont été réimplémentées en assembleur pour garantir une exécution rapide et efficace, en tirant parti des spécificités de l'architecture x86.
- 2. Consommation Minimale de Mémoire** : Une attention particulière a été accordée à la gestion efficace de la mémoire, visant à minimiser l'empreinte mémoire tout en fournissant des fonctionnalités robustes.

#### **Utilisation de la Bibliothèque s\_libc :**

- 1. Makefile** : Un fichier Makefile est fourni pour simplifier la compilation et l'intégration de la bibliothèque dans les projets. Le Makefile définit les règles de compilation, les dépendances et permet une gestion fluide du processus de build.
- 2. Page de Manuel (man)** : Chaque fonction de la bibliothèque est documentée dans une page de manuel (section 3). Il suffit d'utiliser la commande `man 3 nom_de_la_fonction` pour obtenir des informations détaillées sur le fonctionnement et les paramètres d'une fonction spécifique.
- 3. Architecture des Bibliothèques** : Les bibliothèques sont fournies sous forme d'archives partagées (.so) regroupant les fonctions par catégorie (s\_stdio, s\_stlib, etc.). Les fichiers d'en-tête (.h) sont également disponibles pour faciliter l'intégration.

## **Makefile - Guide Rapide :**

- `make all` : Compile la bibliothèque s\_libc.
- `make clean` : Supprime les fichiers temporaires générés lors de la compilation.
- `make test` : Compile et exécute les tests pour vérifier le bon fonctionnement des fonctions.

## **Bibliothèque s\_string.s - Manipulation de Chaînes de Caractères en Assembleur**

La bibliothèque s\_string.s, développée dans le cadre du projet s\_libc, se consacre à la manipulation efficace de chaînes de caractères en langage assembleur. Elle propose des fonctions essentielles pour le traitement des chaînes, visant à offrir des performances optimales tout en minimisant la consommation de mémoire.

## **Fonctions Principales :**

### **1. s\_strlen : Calcul de la Longueur d'une Chaîne**

- Cette fonction détermine la longueur d'une chaîne de caractères, offrant ainsi une alternative rapide à la fonction équivalente de la libc.

### **2. s strcpy : Copie de Chaînes**

- La fonction s strcpy assure la copie d'une chaîne source vers une destination, fournissant ainsi une opération de copie optimisée en assembleur.

### **3. s strncpy : Copie Sécurisée de Chaînes**

- Similaire à s strcpy, mais avec une gestion sécurisée de la copie en spécifiant une longueur maximale. Cela évite les débordements de mémoire potentiels.

### **4. s strcat : Concaténation de Chaînes**

- La fonction s strcat permet de concaténer deux chaînes de caractères, offrant une alternative rapide et optimisée.

### **5. s strncat : Concaténation Sécurisée de Chaînes**

- Similaire à s strcat, mais avec une gestion sécurisée de la concaténation en spécifiant une longueur maximale.

### **6. s strcmp : Comparaison de Chaînes**

- Cette fonction compare deux chaînes de caractères et retourne un résultat indiquant leur équivalence ou différence.

## 7. **s\_strncmp** : Comparaison Sécurisée de Chaînes

- Similaire à s\_strcmp, mais avec une gestion sécurisée de la comparaison en spécifiant une longueur maximale.

Utilisation :

- Les développeurs peuvent inclure la bibliothèque s\_string.s dans leurs projets en ajoutant le fichier source correspondant lors de la compilation.
- Les prototypes des fonctions sont disponibles dans le fichier d'en-tête s\_string.h pour une intégration aisée dans les programmes utilisateurs.

## **Bibliothèque s\_maths.s - des fonctions mathématiques en assembleur.**

1. **s\_abs()** : retourne l'absolue d'un nombre entier.
2. **s\_div()** : divise deux nombres entiers et renvoie le quotient.
3. **s\_pow()** : calcule la puissance d'un nombre entier (x par y donneras  $x^y$ ).

Utilisation :

- Les développeurs peuvent inclure la bibliothèque s\_maths.s dans leurs projets en ajoutant le fichier source correspondant lors de la compilation.
- Les prototypes des fonctions sont disponibles dans le fichier d'en-tête s\_maths.h pour une intégration aisée dans les programmes utilisateurs.

## **Bibliothèque s\_studio.s - Opérations de messages.**

Cette bibliothèque a pour but d'être utilisée pour effectuer des opérations telles que l'affichage de messages, la création et la gestion de flux de lectures et écritures, la lecture et l'écriture de zones mémoires contenant des données provenant ou destinées à un fichier.

Cependant nous avons fait que la fonction suivante :

**s\_puts(\*)**: Impression d'une chaîne de caractères jusqu'au premier caractère null et ajoute d'un nouveau ligne

Utilisation :

- Les développeurs peuvent inclure la bibliothèque s\_studio.s dans leurs projets en ajoutant le fichier source correspondant lors de la compilation.
- Les prototypes des fonctions sont disponibles dans le fichier d'en-tête s\_studio.h pour une intégration aisée dans les programmes utilisateurs.

## **Bibliothèque s\_stdlib.s - fonctions utilitaires courantes et générales.**

**s\_abs()**: Retourne l'absolue d'un nombre entier. Cette fonction est semblable à la fonction abs() de la bibliothèque standard C. Elle ignore le signe du nombre passé en paramètre et renvoie sa valeur absolue.

**s\_atoi()**: Convertit une chaîne de caractères représentant un entier décimal en un entier. Cette fonction est semblable à la fonction atoi() de la bibliothèque standard C. Elle lit chaque caractère de la chaîne jusqu'à ce qu'elle trouve un caractère non numérique ou qu'elle a lu tous les caractères, puis convertit la séquence de chiffres lus en un entier.

Utilisation :

- Les développeurs peuvent inclure la bibliothèque s\_stdlib.s dans leurs projets en ajoutant le fichier source correspondant lors de la compilation.
- Les prototypes des fonctions sont disponibles dans le fichier d'en-tête s\_stdlib.h pour une intégration aisée dans les programmes utilisateurs.

## Deuxième partie - Crackmes :

### Introduction à la Partie des CrackMes : Challenges de Reverse Engineering

Cette partie du projet s'articule autour de l'univers fascinant du Reverse Engineering à travers des challenges appelés "CrackMes". Un CrackMe est un exercice qui nous invite à analyser, comprendre et contourner la protection d'un programme, souvent par le biais de techniques de désassemblage et de rétro-ingénierie.

#### Objectifs et Enjeux :

- 1. Découverte du Reverse Engineering :** Les CrackMes offrent une opportunité d'explorer le Reverse Engineering, une discipline cruciale dans le domaine de la sécurité informatique. Cela permet d'acquérir des compétences pour comprendre le fonctionnement interne d'un programme, identifier les mécanismes de protection, et contourner les obstacles mis en place.
- 2. Application des Connaissances Assembleur :** Nous aurons l'occasion de mettre en pratique nos connaissances en langage assembleur acquises dans la première partie du projet (s libc). Ils seront amenés à analyser le code assembleur d'un programme, à repérer des vulnérabilités potentielles, et à élaborer des solutions pour les exploiter.
- 3. Développement de Stratégies de Contournement :** Face à des mécanismes de protection tels que les checksums, les anti-debugging, ou les protections contre l'analyse dynamique, nous devrons concevoir des stratégies astucieuses pour contourner ces obstacles.

#### Déroulement des CrackMes :

- 1. Sélection des Challenges :** Plusieurs CrackMes de niveaux de difficulté variés seront proposés allant de 1 à 8. Chaque challenge représente une énigme à résoudre en utilisant des compétences spécifiques de Reverse Engineering.
- 2. Analyse du Code Assembleur :** Nous serons amenés à examiner attentivement le code assembleur du programme fourni, à identifier les fonctionnalités clés, et à déterminer le comportement attendu.
- 3. Contournement des Protections :** À travers des étapes de désassemblage, d'analyse statique et dynamique, nous serons encouragés à trouver des moyens créatifs de contourner les protections du programme, par exemple en modifiant le flow d'exécution, en évitant la détection, ou en manipulant des données critiques.

## Objectif Pédagogique :

Cette partie du projet vise à renforcer nos compétences en matière de Reverse Engineering, à stimuler notre esprit analytique, et à les familiariser avec des situations réalistes de contournement de protections logicielles. Les CrackMes constituent ainsi un terrain d'apprentissage interactif et stimulant pour explorer le Reverse Engineering.

## Librairies :

- Rd2dec
- R2pm
- Radare2

## Deviner le mot de passe (challenge 1) :

### Pour se faire nous allons utiliser l'outil radare 2

```
root@debian:/home/user# radare2
Usage: r2 [-ACdfLMnNqStuvwzX] [-P patch] [-p prj] [-a arch] [-b bits] [-i file]
          [-s addr] [-B baddr] [-m maddr] [-c cmd] [-e k=v] file|pid|-|--|=
```

### Nous allons exécuter notre crackmes :

```
root@debian:/home/user/Téléchargements# r2 ./crackmes_1
-- I am Pentium of Borg. Division is futile. You will be approximated.
```

### Puis utiliser l'option “iA” pour voir uniquement l'architecture de notre binaire (64 ou 32 bits) :

```
[0x00001090]> iA
0 0x00000000 15124 x86_32 machine=Intel 80386
```

C'est en 32 bits.

Si on suppose que le mdp et le username sont stockés en dur alors, ils doivent être dans la partie data de notre logiciel. Pour se faire, nous allons utiliser l'option “iz” permettant de lire l'ensemble des chaînes de caractères dans la section .data :

```
[0x00001090]> iz
[Strings]
nth paddr      vaddr      len size section type   string
-----
0 0x00002008 0x00002008 9 10 .rodata ascii superuser
1 0x00002012 0x00002012 9 10 .rodata ascii AlphaPass
2 0x0000201c 0x0000201c 22 23 .rodata ascii -----
3 0x00002033 0x00002033 22 23 .rodata ascii ----- CRACKME 1 -----
4 0x0000204a 0x0000204a 23 24 .rodata ascii -----
5 0x00002062 0x00002062 8 9 .rodata ascii Login :
6 0x0000206e 0x0000206e 11 12 .rodata ascii Password :
7 0x0000207a 0x0000207a 20 21 .rodata ascii ** ACCESS GRANTED **
8 0x0000208f 0x0000208f 19 20 .rodata ascii ** ACCESS DENIED **

[0x00001090]>
```

Et en effet, nous avons bien le username et le mot de passe.

Vérifions :

```
root@debian:/home/user/Téléchargements# ./crackmes_1
-----
----- CRACKME 1 -----
-----

Login : superuser
Password : AlphaPass
** ACCESS GRANTED **
```

C'est vérifié !

Enfin, nous allons désassembler le programme pour trouver son pseudo code SASM et le refaire en C :

### Première étape :

```
[0x00001070]> aaa
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze imports (af@@@i)
INFO: Analyze all functions arguments/locals (afva@@@F)
INFO: Analyze function calls (aac)
INFO: Analyze len bytes of instructions for references (aar)
INFO: Finding and parsing C++ vtables (avrr)
INFO: Analyzing methods
INFO: Recovering local variables (afva)
INFO: Type matching analysis for all functions (aaft)
INFO: Propagate noreturn information (aanr)
INFO: Use -AA or aaaa to perform additional experimental analysis
```

“aaa” permet d'effectuer diverses analyses automatiques sur le binaire, comme l'analyse des fonctions, la recherche de chaînes, etc. Elle est souvent utilisée pour préparer l'analyse.

### Deuxième étape :

```
[0x00001070]> afl
0x00001030    1      6 sym.imp.strcmp
0x00001040    1      6 sym.imp.__libc_start_main
0x00001050    1      6 sym.imp.puts
0x00001070    1      39 entry0
0x00001098    1      4 fcn.00001098
0x000010b0    4      52 sym.deregister_tm_clones
0x00001199    1      4 sym.__x86.get_pc_thunk.dx
0x000010f0    4      71 sym.register_tm_clones
0x00001140    5      75 sym.__do_global_dtors_aux
0x000010a0    1      4 sym.__x86.get_pc_thunk.bx
0x00001060    1      6 fcn.00001060
0x00001190    1      9 sym.frame_dummy
0x00001288    1      20 sym._fini
0x0000119d    8      232 main
0x00001000    3      32 sym._init
```

“afl” permet de lister toutes les fonctions trouvées dans le binaire. Cela nous permet de voir les différentes fonctions disponibles dans le programme.

### **Troisième étape :**

On aperçoit la fonction main dans le binaire, donc on en déduit que c'est cette fonction que nous allons devoir récupérer/inverser :

```
[0x00001070]> s main
[0x0000119d]> pdd
ERROR: Missing plugin. Run: r2pm -ci r2dec
```

Cette commande nous positionne le curseur sur la fonction main. On spécifie ainsi que nous souhaitons inspecter et analyser la fonction main du programme.

Puis on essaie d'exécuter pdd pour le pseudo code mais on a besoin de lancer r2pm et r2dec.

C'est une commande qui installe le plugin r2dec pour radare2. Le plugin r2dec est utilisé pour générer du pseudo-code à partir du code assembleur. L'option -ci installe le plugin depuis le dépôt :

```
[0x0000119d]> r2pm -ci r2dec
INFO: Cleaning r2dec
INFO: Using r2-5.8.9 and r2pm-5.8.9
INFO: Cleaning r2dec
INFO: git clone --depth=10 --recursive https://github.com/warg
.local/share/radare2/r2pm/git//r2dec
Clonage dans '/root/.local/share/radare2/r2pm/git//r2dec'...
remote: Enumerating objects: 154, done.
remote: Counting objects: 100% (154/154), done.
remote: Compressing objects: 100% (126/126), done.
remote: Total 154 (delta 32), reused 87 (delta 21), pack-reuse
Réception d'objets: 100% (154/154), 473.96 Kio | 824.00 Kio/s,
Résolution des deltas: 100% (32/32), fait.
INFO: Starting install for r2dec
INFO: R2PM_NEEDS: Found gcc in PATH
INFO: R2PM_NEEDS: Found make in PATH
INFO: R2PM_NEEDS: Found ninja in PATH
```

### **Quatrième étape :**

On relance pdd et on a bien le pseudo code :

```
0x00000119d]> pdd
* r2dec pseudo code output */
* ./crackmes_2 @ 0x119d */
#include <stdint.h>

nt32_t main (char ** argv) {
    int32_t var_ch;
    ecx = esp + 4;
    x86_get_pc_thunk_bx (ecx, ebx, esi, ebp);
    ebx += 0x2e3f;
    esi = ecx;
    puts (ebx - 0x1fd);
    puts (ebx - 0x1fc3);
    puts (ebx - 0x1fac);
    if (*(esi) != 3) {
        puts (ebx - 0x1f94);
        eax = 1;
    } else {
        eax = *(esi + 4));
    }
}
```

Nous allons devoir réécrire le code originel en C.

#### Cinquième étape - Réécriture du code en C :

#### Deviner le mot de passe (challenge 2) :

Ce programme n'est pas interactif, on en présume qu'il faudra mettre des arguments :

```
root@debian:/home/user/Téléchargements# ./crackmes_2
-----
----- CRACKME 2 -----
-----
Missing login and password !
```

#### Test avec arguments :

```
root@debian:/home/user/Téléchargements# ./crackmes_2 admin password
-----
----- CRACKME 2 -----
-----
** ACCESS DENIED **
```

C'est bien ça. Il utilise des arguments.

#### Refaisons la même chose que dans le challenge 1 en regardant la section .data

:

```

root@debian:/home/user/Téléchargements# r2 ./crackmes_2
-- Analyze socket connections with the socket plugin: 'radare2 socket://www.foo
.com:80'. Use 'w' to send data
[0x00001070]> iz
[Strings]
nth paddr      vaddr      len size section type  string
-----
0 0x00002008 0x00002008 8   9    .rodata ascii du6Thoo7
1 0x00002011 0x00002011 8   9    .rodata ascii EW8ingoh
2 0x0000201a 0x0000201a 22  23   .rodata ascii -----
3 0x00002031 0x00002031 22  23   .rodata ascii ----- CRACKME 2 -----
4 0x00002048 0x00002048 23  24   .rodata ascii -----
5 0x00002060 0x00002060 28  29   .rodata ascii Missing login and password !
6 0x0000207d 0x0000207d 20  21   .rodata ascii ** ACCESS GRANTED **
7 0x00002092 0x00002092 19  20   .rodata ascii ** ACCESS DENIED **
[0x00001070]> quit

```

Et désormais, rentrons le username et le mot de passe en argument :

```

root@debian:/home/user/Téléchargements# ./crackmes_2 du6Thoo7 EW8ingoh
-----
----- CRACKME 2 -----
-----
** ACCESS GRANTED **

```

Ça fonctionne bien.

Enfin, nous allons désassembler le programme pour trouver son pseudo code SASM et le refaire en C.

**Toutes les étapes sont équivalentes au premier challenge, pas la peine de remettre la même chose.**

Cinquième étape - Réécriture du code en C :

## Deviner le mot de passe (challenge 3) :

Pour ce challenge, le mot de passe n'est pas directement dans la section .data :

	lth	paddr	vaddr	len	size	section	type	string
0		0x00002008	0x00002008	22	23	.rodata	ascii	-----
1		0x0000201f	0x0000201f	22	23	.rodata	ascii	---- CRACKME 3 -----
2		0x00002036	0x00002036	23	24	.rodata	ascii	-----\n
3		0x0000204e	0x0000204e	28	29	.rodata	ascii	Missing login and password !
4		0x0000206b	0x0000206b	20	21	.rodata	ascii	** ACCESS GRANTED **
5		0x00002080	0x00002080	19	20	.rodata	ascii	** ACCESS DENIED **

Nous allons devoir chercher autre part, nous allons commencer par lister toutes les chaînes de caractères lisibles avec la commande izz :

	lth	paddr	vaddr	len	size	section	type	string
0		0x00000028	0x00000028	4	10		utf16le	4 \v(
1		0x00000038	0x00000038	5	24		utf32le	444S blocks=Basic Latin,Latin Extended-A
2		0x00000194	0x00000194	18	19	.interp	ascii	/lib/ld-linux.so.2
3		0x0000029d	0x0000029d	14	15	.dynstr	ascii	_IO_stdin_used
4		0x000002ac	0x000002ac	17	18	.dynstr	ascii	__libc_start_main
5		0x000002be	0x000002be	6	7	.dynstr	ascii	strcmp
6		0x000002c5	0x000002c5	4	5	.dynstr	ascii	puts
7		0x000002ca	0x000002ca	14	15	.dynstr	ascii	__cxa_finalize
8		0x000002d9	0x000002d9	9	10	.dynstr	ascii	libc.so.6
9		0x000002e3	0x000002e3	11	12	.dynstr	ascii	GLIBC_2.1.3
10		0x000002ef	0x000002ef	10	11	.dynstr	ascii	GLIBC_2.34
11		0x000002fa	0x000002fa	9	10	.dynstr	ascii	GLIBC_2.0
12		0x00000304	0x00000304	27	28	.dynstr	ascii	_ITM_deregisterTMCloneTable
13		0x00000320	0x00000320	14	15	.dynstr	ascii	__gmon_start__
14		0x0000032f	0x0000032f	25	26	.dynstr	ascii	_ITM_registerTMCloneTable
15		0x000011a0	0x000011a0	8	9	.text	ascii	XahHaf1J
16		0x000011ac	0x000011ac	5	6	.text	ascii	Fae9v
17		0x00001292	0x00001292	4	5	.text	ascii	Y[^]
18		0x00002008	0x00002008	22	23	.rodata	ascii	-----

Après de longues recherches en passant dans la fonction main, les registres et même tenter les boucles, on s'est aperçu que c'était tout bête.

Le mot de passe et le login étaient juste au dessus dans la section .text :

```
14 0x0000032f 0x0000032f 25 26 .dynstr ascii _ITM_registerTMCloneTable
15 0x000011a0 0x000011a0 8 9 .text ascii XahHaf1J
16 0x000011ac 0x000011ac 5 6 .text ascii Fae9v
17 0x00001292 0x00001292 4 5 .text ascii V[^]
18 0x00002008 0x00002008 22 23 .rodata ascii -----
19 0x0000201f 0x0000201f 22 23 .rodata ascii ----- CRACKME 3 -----
20 0x00002036 0x00002036 23 24 .rodata ascii -----
21 0x0000204e 0x0000204e 28 29 .rodata ascii Missing login and password !
22 0x0000206b 0x0000206b 20 21 .rodata ascii ** ACCESS GRANTED **
23 0x00002080 0x00002080 19 20 .rodata ascii ** ACCESS DENIED **
24 0x00002123 0x00002123 5 6 .eh_frame ascii ;*2$"
```

Testons :

```
[0x000035a3]> quit
root@debian:/home/user/Téléchargements# ./crackmes_3 XahHaf1J Fae9v
-----
----- CRACKME 3 -----
-----
** ACCESS GRANTED **
```

### Dernière étape - Réécriture du code en C :

### Deviner le mot de passe avec Cutter (challenge 4) :

Étant donné que la correction est déjà sur le PDF, nous allons passer l'étape de démonstration. Cependant, cette étape nous aura été crucial pour nous initier à Cutter.

Nous avons pu en apprendre un peu plus.

### Deviner le mot de passe avec Cutter (challenge 5) :

Pour ce challenge, les valeurs ne sont pas données en arguments, mais saisies directement en ligne de commande. Donc nous allons devoir insérer les valeurs du login et du mot de passe directement dans la console et pas en argument lors du lancement du débogage.

#### Etape 1 :

**Mettre la syntaxe de l'assembleur en at&t et mettre Cutter en anglais pour éviter les bugs lors des insertions de valeurs dans la console de débug :**

```
[0x000011bd]> e asm.syntax=att
```

```
[0x000011bd]> e asm.syntax  
att
```

## Etape 2 :

Repérer les insertions de valeurs et les comparaisons puis insérer des breakpoints :

The screenshot shows the Immunity Debugger interface. On the left, the symbol table lists various functions like main, entry0, etc. The central area displays assembly code for the main function. Below the assembly is a decompiled C code representation. Two specific lines in the C code are highlighted with red arrows:

```
scanf(unaff_EBX + 0xe83, &var_18h);  
scanf(unaff_EBX + 0xe83, &var_1ch);
```

These lines correspond to the scanf statements in the assembly code, which read from memory locations `unaff_EBX + 0xe83` and `unaff_EBX + 0xe83` respectively. The assembly code also includes instructions for printing to the screen and flushing buffers.

Nous voyons ici dans le décompilateur que nous avons bien 2 scanf ce qui est cohérent avec nos deux insertions de valeurs : login et password.

Puis, la boucle qui va comparer nos valeurs reçus avec celles enregistrés dans le logiciel.

Nous allons donc placer nos breakpoints ici :

```

Disassembly
Address: 0x00000787
Name
entry.init0
entry0
fcn.00001080
fcn.00001088
main
sym._do_global_dtors_au
sym._x86.get_pc_thunk.t
sym._x86.get_pc_thunk.c
sym._fini
sym._init
sym.deregister_tm_clones
sym.imp._isoc99_scanf
sym.imp._libc_start_main
sym.imp.flush
sym.imp.printf
sym.imp.puts
sym.register_tm_clones

Functions
Name
entry.init0
entry0
fcn.00001080
fcn.00001088
main
sym._do_global_dtors_au
sym._x86.get_pc_thunk.t
sym._x86.get_pc_thunk.c
sym._fini
sym._init
sym.deregister_tm_clones
sym.imp._isoc99_scanf
sym.imp._libc_start_main
sym.imp.flush
sym.imp.printf
sym.imp.puts
sym.register_tm_clones

Disassembly
Address: 0x00000787
0x000001297 leal -0x1fd(%ebx), %eax
0x00000129d pushl %eax ; const char *format
0x00000129e calll __isoc99_scanf ; sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x0000012a3 addl $0x10, %esp
0x0000012a6 movl %eax, var_14h
0x0000012a9 cmpl $1, var_14h
0x0000012ad je 0x1c8
0x0000012af subl $0xc, %esp
0x0000012b2 leal -0xf79(%ebx), %eax
0x0000012b8 pushl %eax ; const char *
0x0000012b9 calll puts ; sym.imp.puts ; int puts(const char *)
0x0000012bc addl $0x10, %esp
0x0000012c1 movl $2, %eax
0x0000012c6 jmp 0x13d
0x0000012c8 movl 0x28(%ebx), %edx
0x0000012ce movl var_10h, %eax

Dashboard Strings Imports Search Disassembly Graph(main) Hexdump
Decompiler (main)
var_14h = 0;
printf(unaff_EBX + 0xe7a);
fflush();
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_18h);
if (var_14h == 1) {
    printf(unaff_EBX + 0xe9b);
    fflush();
}
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_1ch);
if (var_14h == 1) {
    if ((*(int32_t *) (unaff_EBX + 0x2e40) == var_18h) && (*(int32_t *) (unaff_EBX + 0x2e4c) == var_1ch)) {
        puts(unaff_EBX + 0xebf);
    } else {
        puts(unaff_EBX + 0xed4);
    }
    uVar1 = 0;
} else {
}

Console

```

### Etape 3 :

Lançons le débogage et insérons des valeurs théoriques pour le login :

```

Disassembly
Address: 0x00000787
Name
entry.init0
entry0
fcn.00001080
fcn.00001088
main
sym._do_global_dtors_au
sym._x86.get_pc_thunk.t
sym._x86.get_pc_thunk.c
sym._fini
sym._init
sym.deregister_tm_clones
sym.imp._isoc99_scanf
sym.imp._libc_start_main
sym.imp.flush
sym.imp.printf
sym.imp.puts
sym.register_tm_clones

Functions
Name
entry.init0
entry0
fcn.00001080
fcn.00001088
main
sym._do_global_dtors_au
sym._x86.get_pc_thunk.t
sym._x86.get_pc_thunk.c
sym._fini
sym._init
sym.deregister_tm_clones
sym.imp._isoc99_scanf
sym.imp._libc_start_main
sym.imp.flush
sym.imp.printf
sym.imp.puts
sym.register_tm_clones

Disassembly
Address: 0x00000787
0x5663d239 leal var_18h, %eax
0x5663d23c pushl %eax
0x5663d23d leal -0x1fd(%ebx), %eax
0x5663d243 pushl %eax ; const char *format
0x5663d244 calll __isoc99_scanf ; sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x5663d249 addl $0x10, %esp
0x5663d24c movl %eax, var_14h
0x5663d24f cmpl $1, var_14h
0x5663d253 leal -0x5663d271(%ebx), %eax
0x5663d255 subl $0xc, %esp
0x5663d258 pushl %eax ; const char *
0x5663d25e calll puts ; sym.imp.puts ; int puts(const char *)
0x5663d25f addl $0x10, %esp
0x5663d264 movl $1, %eax
0x5663d267 jmp 0x5663d30d

Strings Search Memory Map Breakpoints Disassembly Graph(main) Hexdump
Decompiler (main)
var_14h = 0;
printf(unaff_EBX + 0xe7a);
fflush();
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_18h); ↗
if (var_14h == 1) {
    printf(unaff_EBX + 0xe9b);
    fflush();
}
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_1ch);
if (var_14h == 1) {
    if ((*(int32_t *) (unaff_EBX + 0x2e40) == var_18h) && (*(int32_t *) (unaff_EBX + 0x2e4c) == var_1ch)) {
        puts(unaff_EBX + 0xebf);
    } else {
        puts(unaff_EBX + 0xed4);
    }
    uVar1 = 0;
} else {
}

Console
----- CRACKME 5 -----
Login : hit breakpoint at: 0x5663d239
Sent input: '54'

Debugger Input | Enter input for the debugger

```

Continuons pour aller dans la prochaine fonction qui contiendra le password :

Disassembly

```

0x565d9293: leal var_1ch, %eax
0x565d9296: pushl %eax
0x565d9297: leal -0x1f9d(%ebx), %eax
0x565d929d: pushl %eax
0x565d929e: calll _isoc99_scnaf ; sym.imp._isoc99_scnaf ; int scanf(const char *format)
0x565d92a3: addl $0x10, %esp
0x565d92a6: movl %eax, var_14h
0x565d92a9: cmpl $1, var_14h
0x565d92ad: je 0x565d92c8
0x565d92a7: subl $0xc, %esp
0x565d92b2: leal -0x1f79(%ebx), %eax
0x565d92b6: pushl %eax ; const char *
0x565d92b9: calll puts ; sym.imp.puts ; int puts(const char *)
0x565d92be: addl $0x10, %esp
0x565d92c1: movl $2, %eax
0x565d92c6: jmp 0x565d930d

```

Decompiler (main)

```

undefined4 main(char **argv)
{
    undefined4 uVar1;
    int32_t unaff_EBX;
    int32_t var_1ch;
    int32_t var_18h;
    unsigned long var_14h;
    int32_t var_10h;

    var_10h = (int32_t)&argv;
    _x86.get_pc_thunk.bx();
    puts(unaff_EBX + 0xe34);
    puts(unaff_EBX + 0xe40);
    puts(unaff_EBX + 0xe62);

    Console
    Login : hit breakpoint at: 0x565d9293
    Sent input: '54'
    Continue until 0x565d91bd
    Password : hit breakpoint at: 0x565d9293
    Sent input: '108'
}

```

Quick Filter x

Debugger Input Enter input for the debugger

Puis continuons d'avancer jusqu'à notre dernier breakpoint qui concerne la comparaison de nos deux valeurs :

Disassembly

```

0x565d92c8: movl 0x28(%ebx), %edx
0x565d92c9: movl var_18h, %eax
0x565d92d1: cmpl %eax, %edx
0x565d92d3: jne 0x565d92f6
0x565d92d5: movl 0x2c(%ebx), %eax
0x565d92d8: movl %eax, %eax
0x565d92d9: cmpl %eax, %edx
0x565d92e0: jne 0x565d92f6
0x565d92e2: subl $0xc, %esp
0x565d92e5: leal -0x1f61(%ebx), %eax
0x565d92e8: puts(var_14h); const char *
0x565d92ec: calll puts ; sym.imp.puts ; int puts(const char *)
0x565d92f1: addl $0x10, %esp
0x565d92f4: jmp 0x565d9308
0x565d92f9: subl $0xc, %esp
0x565d92f9: leal -0x1f4c(%ebx), %eax

```

Decompiler (main)

```

var_14h = 0;
printf(unaff_EBX + 0xe7a);
fflush(0);
var_14h = _isoc99_scnaf(unaff_EBX + 0xe83, &var_18h);
if (var_14h == 1) {
    printf(unaff_EBX + 0xeb9);
    fflush(0);
    var_14h = _isoc99_scnaf(unaff_EBX + 0xe83, &var_1ch);
    if (var_14h == 1) {
        if (((int32_t *)unaff_EBX + 0xe48) == var_18h && ((int32_t *)unaff_EBX + 0xe4c) == var_1ch)) {
            puts(unaff_EBX + 0xebf);
        } else {
            puts(unaff_EBX + 0xed4);
        }
    }
    uVar1 = 0;
}

Console
Continue until 0x565d91bd
Password : hit breakpoint at: 0x565d9293
Sent input: '108'
Continue until 0x565d91bd
hit breakpoint at: 0x565d92c8

```

Registers

Register	Value
eip	0x565d92c8
ebx	0x565dbff4
ecx	0x0
esi	0x91
eax	0x36
edx	0x565dbeec
edi	0xf7f45b80
esp	0xfffff99a0
ebp	0xfffff99b8

Stack

Offset	Value
0xfffff99a0	0xfffff99a0 -- stack rw 0xf7e1c4
0xfffff99a4	0x0000000c -- ascii 0xf7e1cff4
0xfffff99a8	0x00000036 -- %eax ascii 0xf7e1c4
0xfffff99ac	0x00000001 --
0xfffff99b0	0xfffff99b0 -- stack rw 0x000000
0xfffff99b4	0x7e1cf1f4 -- (/usr/lib32/libc.so.6)
0xfffff99b8	0x00000000 --
0xfffff99bc	0x7c232c5 -- (/usr/lib32/libc.so.6)
0xfffff99c0	0x00000000 --
0xfffff99c4	0x00000070 -- ascii 0x8310c483
0xfffff99c8	0x77f45f4 -- (/usr/lib32/libc.so.6)
0xfffff99cc	0x7c232c5 -- (/usr/lib32/libc.so.6)
0xfffff99d0	0x00000001 --

Debugger Input Enter input for the debugger

Nous voyons qu'il compare le registre %eax qui contient 54 (ce que nous avons entré précédemment) en hexadécimal, donc 0x36 et le registre %edx qui contient 0x91 en hexa ce qui donne 145 en décimal. Nous allons en déduire que la bonne valeur que le logiciel garde dans sa mémoire est celle-ci.

On va redémarrer le programme pour que l'on puisse ajouter 145 au login et qu'il passe à la comparaison du mot de passe, sans ça, il va directement passer à une autre boucle car il a une ligne juste après : jne ... qui veut dire : n'est pas égal à.

## Le password :

The screenshot shows the Ghidra debugger interface. The assembly window displays the following code snippet:

```

0x566282c8 movl $0x28(%ebx), %edx
0x566282c9 movl %eax, %eax
0x566282d0 cmpl %eax, %edx
0x566282d1 jne 0x566282f2
0x566282d2 movl $0xc(%ebx), %edx
0x566282d3 movl var_1ch, %eax
0x566282d4 cmpi %eax, %edx
0x566282d5 jne 0x566282f2
0x566282d6 movl $0xc(%ebx), %edx
0x566282d7 movl var_1ch, %eax
0x566282d8 cmpi %eax, %edx
0x566282d9 jne 0x566282f2
0x566282dA movl $0xc(%ebx), %edx
0x566282dB pushl %eax
0x566282dC calll puts ; const char *
0x566282dD addl $0x10, %esp
0x566282dE subl $0x10, %esp
0x566282dF jmp 0x56628308
0x566282e0 subl $0x10, %esp
0x566282e1 leal -0x1f4(%ebx), %eax

```

A red arrow points to the instruction `jne 0x566282f2`. The registers window shows:

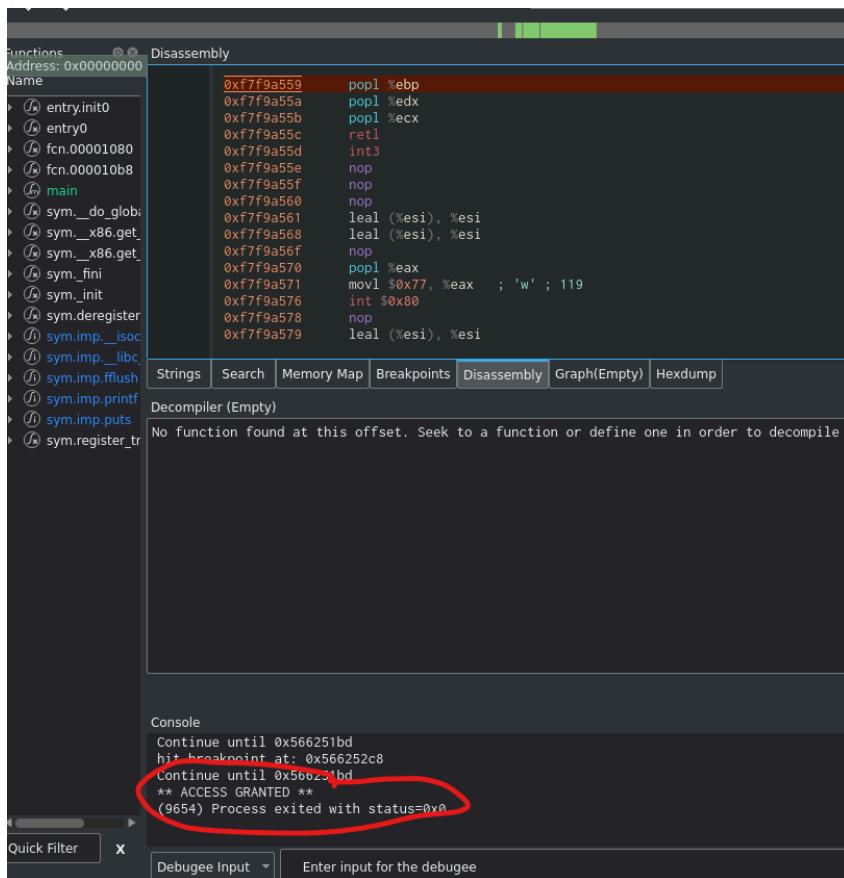
Register	Value
eax	0x91
ebx	0x5662aff4
ecx	0x0
edx	0x3ba
esi	0x5662aeecc
edi	0xf7f21bb0
esp	0xffffc9ca10
ebp	0xffffc9ca28

The stack window shows memory dump data starting at `0xffffc9ca10`.

Une fois que l'on a mis le login 145, nous avons pu passer la comparaison et l'instruction de saut conditionnel "n'est pas égal à". Cela veut dire que nous avons réussi à prendre le bon login.

Lorsqu'on continue dans la boucle, l'application va faire une deuxième vérification, le mot de passe, j'ai entré ultérieurement 145 donc c'est pour cela que %eax affiche 0x91. Il compare ça à 0x3ba qui est 954.

## Refaisons la même manipulation avec le bon login et password :



Quand on termine le débogage nous avons bien le **ACCES GRANTED** ce qui signifie que nous avons réussi à trouvé le bon login et mot passe.

### Version graphique :

```
user@debian:~/Téléchargements$ ./crackmes_5
-----
----- CRACKME 5 -----
-----

Login : 145
Password : 954
** ACCESS GRANTED **
user@debian:~/Téléchargements$
```

### Deviner le mot de passe avec Cutter (challenge 6) :

Le CrackMe 6 propose un challenge dont plusieurs mots de passe peuvent correspondre. Les mots de passe suivent une loi, à nous de découvrir laquelle. Tout d'abord, comme à mon habitude je test des mdp :

```
-----  
---- CRACKME 6 ----  
-----
```

```
Login : 55  
Password : 55  
** ACCESS GRANTED **
```

Je m'aperçois que cela fonctionne. Je n'ai même pas eu besoin d'utiliser cutter.  
Donc testons d'autres mots de passe :

```
-----  
---- CRACKME 6 ----  
-----
```

```
Login : 66  
Password : 66  
** ACCESS GRANTED **
```

```
-----  
---- CRACKME 6 ----  
-----
```

```
Login : 15  
Password : 15  
** ACCESS GRANTED **
```

Tous les entiers identiques fonctionnent, si en login je met 150 et en mot de passe 150 cela fonctionnera. La loi permettant ça est une condition qui dit que si le mot de passe est égal au login, alors c'est bon.

Si je mets 33 en login et 88 en mot de passe, on me refuse l'accès et je ne passe pas la boucle. C'est comme ça que j'en est déduit que l'application te laisse passer uniquement si on a le même login et mot de passe. Ce qui n'est pas très bien sécurisé.

Nous allons quand même vérifier ça en détail dans l'application :

ym.imp.fflush  
ym.imp.printf  
ym.imp.puts  
ym.register\_tm\_clones

Dashboard | Strings | Imports | Search | Disassembly | Graph(main) | Hexdump

Decompiler (main)

```

puts(unaff_EBX + 0xe34);
puts(unaff_EBX + 0xe4b);
puts(unaff_EBX + 0xe62);
var_14h = 0;
printf(unaff_EBX + 0xe7a);
fflush(0);
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_18h);
if (var_14h == 1) {
    printf(unaff_EBX + 0xe9b);
    fflush(0);
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_1ch);
if (var_14h == 1) {
    if (var_18h == var_1ch) {
        puts(unaff_EBX + 0xebf);
    } else {
        puts(unaff_EBX + 0xed4);
    }
}
}

```

Je place 3 breakpoints : 2 sur le scanf, 1 sur la comparaison :

Disassembly

```

0x000012ce    cmpl %eax, %edx
0x000012d0    jne 0x12e6
0x000012d2    subl $0xc, %esp
0x000012d5    leal -0x1f61(%ebx), %eax
0x000012db    pushl %eax           ; const char *
0x000012dc    calll puts          ; sym.imp.puts ; int puts(const c
0x000012e1    addl $0x10, %esp
0x000012e4    jmp 0x12f8
0x000012e6    subl $0xc, %esp
0x000012e9    leal -0x1f4c(%ebx), %eax
0x000012ef    pushl %eax           ; const char *
0x000012f0    calll puts          ; sym.imp.puts ; int puts(const c
0x000012f5    addl $0x10, %esp
0x000012f8    movl $0, %eax
0x000012fd    leal var_10h, %esp
0x00001300    popl %ecx

```

Dashboard | Strings | Imports | Search | Disassembly | Graph(main) | Hexdump

Decompiler (main)

```

puts(unaff_EBX + 0xe34);
puts(unaff_EBX + 0xe4b);
puts(unaff_EBX + 0xe62);
var_14h = 0;
printf(unaff_EBX + 0xe7a);
fflush(0);
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_18h);
if (var_14h == 1) {
    printf(unaff_EBX + 0xe9b);
    fflush(0);
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_1ch);
if (var_14h == 1) {
    if (var_18h == var_1ch) {
        puts(unaff_EBX + 0xebf);
    }
}
}

```

Analysons le code avec le débogage :

Disassembly

```

0x5663d239 leal    var_18h, %eax
0x5663d23c pushl    %eax
0x5663d23d leal    -0x1f9d(%ebx), %eax
0x5663d243 pushl    %eax ; const char *format
0x5663d244 calll   __isoc99_scanf ; sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x5663d249 addl    $0x10, %esp
0x5663d24c movl    %eax, var_14h
0x5663d24f cmpl    $1, var_14h
0x5663d253 je     0x5663d271
0x5663d255 subl    $0xc, %esp
0x5663d258 leal    -0x1f9a(%ebx), %eax
0x5663d25e pushl    %eax ; const char *
0x5663d25f calll   puts ; sym.imp.puts ; int puts(const char *)
0x5663d264 addl    $0x10, %esp
0x5663d267 movl    $1, %eax
0x5663d26c jmp    0x5663d2fd

```

Strings Search Memory Map Breakpoints Disassembly Graph(main) Hexdump

Decompiler (main)

```

var_14h = 0;
printf(unaff_EBX + 0xe7a);
fflush(0);
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_18h);
if (var_14h == 1) {
    printf(unaff_EBX + 0xe9b);
    fflush(0);
    var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_1ch);
    if (var_14h == 1) {
        if (var_18h == var_1ch) {
            puts(unaff_EBX + 0xebf);
        } else {
            puts(unaff_EBX + 0xed4);
        }
    }
}

```

Console

```

-----
Login : hit breakpoint at: 0x5663d239
Sent input: '55' 

```

On passe bien le scanf en insérant 55 (au hasard).

Ensuite on va au prochain breakpoint pour passer le password :

Disassembly

```

0x5663d293 leal    var_1ch, %eax
0x5663d296 pushl    %eax
0x5663d297 leal    -0x1f9d(%ebx), %eax
0x5663d29d pushl    %eax ; const char *format
0x5663d29e calll   __isoc99_scanf ; sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x5663d2a3 addl    $0x10, %esp
0x5663d2a6 movl    %eax, var_14h
0x5663d2a9 cmpl    $1, var_14h
0x5663d2ad je     0x5663d2c8
0x5663d2af subl    $0xc, %esp
0x5663d2b2 leal    -0x1f79(%ebx), %eax
0x5663d2b8 pushl    %eax ; const char *
0x5663d2b9 calll   puts ; sym.imp.puts ; int puts(const char *)
0x5663d2be addl    $0x10, %esp
0x5663d2c1 movl    $2, %eax
0x5663d2c6 jmp    0x5663d2fd

```

Strings Search Memory Map Breakpoints Disassembly Graph(main) Hexdump

Decompiler (main)

```

var_14h = 0;
printf(unaff_EBX + 0xe7a);
fflush(0);
var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_18h);
if (var_14h == 1) {
    printf(unaff_EBX + 0xe9b);
    fflush(0);
    var_14h = __isoc99_scanf(unaff_EBX + 0xe83, &var_1ch);
    if (var_14h == 1) {
        if (var_18h == var_1ch) {
            puts(unaff_EBX + 0xebf);
        } else {
            puts(unaff_EBX + 0xed4);
        }
    }
}

```

Console

```

Sent input: '55'
Continue until 0x5663d1bd
Password : hit breakpoint at: 0x5663d293
Sent input: '50' 

```

On met un mot de passe différent pour tester la boucle et voir comment elle fonctionne en détail.

Passons au dernier breakpoint :

Disassembly

```

0x5663d2ce cmpl    eax, edx
0x5663d2d0 jne     0x566412e6
0x5663d2d2 subl    $0xc, %esp
0x5663d2d5 leal    -0x1f61(%ebx), %eax
0x5663d2d8 pushl    %eax ; const char *
0x5663d2dc calll    puts ; sym.imp.puts ; int puts(const char *)
0x5663d2e1 addl    $0x10, %esp
0x5663d2e4 jmp     0x5663d2f8
0x5663d2e6 subl    $0xc, %esp
0x5663d2e9 leal    -0x1f4c(%ebx), %eax
0x5663d2ef pushl    %eax ; const char *
0x5663d2f0 calll    puts ; sym.imp.puts ; int puts(const char *)
0x5663d2f5 addl    $0x10, %esp
0x5663d2f8 movl    $0, %eax
0x5663d2f9 leal    var_10h, %esp
0x5663d300 popl    %ecx

```

Registers

Register	Value
eax	0x32
ebx	0x5663fff4
ecx	0x0
edx	0x37
esi	0x5663fec
edi	0xf7f04b80
esp	0xfffb25fb0
ebp	0xfffb25fc8

On voit qu'il va comparer les deux valeurs edx et eax.

Ensuite, si les deux valeurs ne correspondent pas, alors la fonction "jne" (jump not equal) va nous envoyer à l'adresse 0x566412e6.

Et lorsqu'on continue d'avancer dans le programme, on a le message access denied

Disassembly

```

0x7f91559 invalid
0x7f9155a invalid
0x7f9155b invalid
0x7f9155c invalid
0x7f9155d invalid
0x7f9155e invalid
0x7f9155f invalid
0x7f91560 invalid
0x7f91561 invalid
0x7f91562 invalid
0x7f91563 invalid
0x7f91564 invalid
0x7f91565 invalid
0x7f91566 invalid
0x7f91567 invalid
0x7f91568 invalid

```

Registers

Register	Value
eax	0xfffffffda
ebx	0x0
ecx	0xfc
edx	0x1
esi	0xfffffb4
edi	0x1
esp	0xfffe495c4
ebp	0x7e1e500

Sauf que si nous mettons le même mot de passe et login, par exemple : 2024 2024

Disassembly

```

0x565df2ce cmpl    edx, eax
0x565df2d0 jne     0x565df2e6
0x565df2d2 addl    $0x10, %esp
0x565df2d5 lea     eax, [esp+0x1f61]
0x565df2d8 pushl    %eax ; const char *
0x565df2db calll    puts ; sym.imp.puts ; int puts(const char *)
0x565df2e1 addl    $0x10, %esp
0x565df2e4 jmp     0x565df2f8
0x565df2e6 addl    $0x10, %esp
0x565df2e9 lea     eax, [esp+0x1f4c]
0x565df2ef pushl    %eax ; const char *
0x565df2f0 calll    puts ; sym.imp.puts ; int puts(const char *)
0x565df2f5 addl    $0x10, %esp
0x565df2f8 movl    $0, %eax
0x565df2f9 lea     esp, [var_10h]
0x565df300 popl    %ecx

```

Registers

Register	Value
eax	0x7e8
ebx	0x565e1ff4
ecx	0x0
edx	0x7e8
esi	0x565e1eec
edi	0xf7ccb88
esp	0xfffe11a0
ebp	0xfffe11b8

Là, nous avons les mêmes valeurs dans les deux registres.  
Avançons dans la boucle et passons le jne.

```

0x565df2e6 cmp    eax, eax
0x565df2e9 jne    0x565df2e6
0x565df2eb sub    eax, 0x0
0x565df2f5 lea    eax, [ebx - 0x1f61]
0x565df2f8 push   eax
0x565df2f9 call   puts, : const char *
0x565df2fa add    esp, 0x10
0x565df2e4 jmp    0x565df2f8
0x565df2e9 sub    eax, 0x0
0x565df2e9 lea    eax, [ebx - 0x1f4c]
0x565df2f9 push   eax
0x565df2f9 call   puts, : const char *
0x565df2f9 mov    eax, 0
0x565df2f8 lea    eax, [var_10h]
0x565df2f0 pop    ecx

```

Offset	Value
0xffee1194	0xffee11a4
0xffee1198	0x7c1ca2f_d_audit_preinit
0xffee119c	0x565df1d4
0xffee11a0	0x7c1ca2f_d_audit_preinit
0xffee11a4	0x000007e8
0xffee11a8	0x000007e8
0xffee11ac	0x00000001
0xffee11b0	0x7c1ca2f_d_audit_preinit
0xffee11b4	0x565df1d4
0xffee11b8	0x00000001

Continuons le débogage jusqu'à la fin du processus et nous pouvons apercevoir que nous avons passé l'accès :

```

0x7f97559 pop    ebp
0x7f9755a pop    edx
0x7f9755b pop    ecx
0x7f9755c ret
0x7f9755d int3
0x7f9755e nop
0x7f9755f nop
0x7f97560 nop
0x7f97561 lea    esi, [esi]
0x7f97562 lea    esi, [esi]
0x7f97563 nop
0x7f97570 pop    eax
0x7f97571 mov    eax, 0x77 ; 'w' ; 119
0x7f97576 inc    eax, 0x80
0x7f97578 nop
0x7f97579 lea    esi, [esi]

```

Offset	Value
0xffee1154	0xf7e1e500
0xffee1158	0x00000001
0xffee115c	0x0000000f
0xffee1160	0xf7cdff862
0xffee1164	0xf7e1e330
0xffee1168	0xf7e1e330
0xffee116c	0xf7cbb5d
0xffee1170	0x00000000
0xffee1174	0x00000001

## Deviner le mot de passe avec Cutter (challenge 7) :

Le crackme attend de nous un mot de passe avec des caractères en argument, décompilons le dans Cutter et regardons en détail dans le désassemblleur :

The screenshot shows the Cutter debugger interface. The left pane displays a list of functions, with 'main' selected. The right pane shows the assembly code for 'main' at address 0x000000da8. A yellow bracket highlights the assembly code from 0x5663025e to 0x5663028d. Below the assembly, the decompiler output is shown:

```
int32_t unaff_EBX;
int32_t var_14h;

__x86.get_pc_thunk.bx();
puts(unaff_EBX + 0xde8);
puts(unaff_EBX + 0xdff);
puts(unaff_EBX + 0xe16);
if (*extraout(ECX) == 2) {
    iVar2 = check_pal(*char **)(extraout(ECX[1] + 4));
    if (iVar2 == 1) {
        puts(unaff_EBX + 0xe42);
    } else {
        puts(unaff_EBX + 0xe57);
    }
    uVar1 = 0;
}
```

A red arrow points from the highlighted assembly code to the corresponding decompiled line.

Cette va nous intéresser car elle utilise une fonction et s'en suit des vérifications avec des if. Nous allons mettre un breakpoint ici et débugger ensemble pour voir ce que fait l'application.

The screenshot shows the Cutter debugger interface. The left pane displays a list of functions, with 'main' selected. The right pane shows the assembly code for 'main' at address 0x000000da8. A yellow bracket highlights the assembly code from 0x5663025e to 0x5663028d. Below the assembly, the decompiler output is shown:

```
int32_t unaff_EBX;
int32_t var_14h;

__x86.get_pc_thunk.bx();
puts(unaff_EBX + 0xde8);
puts(unaff_EBX + 0xdff);
puts(unaff_EBX + 0xe16);
if (*extraout(ECX) == 2) {
    iVar2 = check_pal(*char **)(extraout(ECX[1] + 4)); 1
    if (iVar2 == 1) {
        puts(unaff_EBX + 0xe42);
    } else {
        puts(unaff_EBX + 0xe57);
    }
    uVar1 = 0;
}
```

A red circle labeled '1' is placed over the 'iVar2 = check\_pal...' line. A red circle labeled '2' is placed over the 'testmax' entry in the 'Native debugging console' window.

Lorsqu'on avance dans le débogage, l'application va appeler la fonction check\_pal et nous pouvons voir notre mot rentré en argument dans le registre eax :

Disassembly:

```

0x565c827f addl    $4    %eax
0x565c8282 movl    (%eax), %eax
0x565c8284 subl    $0xc, %esp
0x565c8287 pushl    %eax ; const char **
0x565c8288 calll   check_pal ; sym.check_pal
0x565c828d addl    $0x10, %esp
0x565c8290 cmpl    $1, %eax ; edx
0x565c8293 jne     $0x565c82a9
0x565c8295 subl    $0xc, %esp
0x565c8298 leal    -0x1f92(%ebx), %eax
0x565c829e pushl    %eax ; const char *
0x565c82a1 puts    $0x10(%esp) ; int puts(const char *)
0x565c82a4 addl    $0x10, %esp
0x565c82a7 jmp     $0x565c82bb
0x565c82a9 subl    $0xc, %esp
0x565c82ac leal    -0x1f7d(%ebx), %eax

```

Registers:

Register	Value
eax	0xffffe0fc8
eax	0x565c8288
ecx	0x565c82a4
edx	0x1
esi	0xffffe7d80
edi	0xf7f02b80
esp	0xffffe7d40

Stack:

Offset	Value
0xffffe7d40	0xffffe0fc8 testmax
0xffffe7d44	0x7edf8cb -> (/usr/lib32/ld-linux.so.2) lib
0xffffe7d48	0x7f7c1a2f_d1_audit_preinit
0xffffe7d4c	0x565c8220 -> (/home/user/Téléchargements/cr)
0xffffe7d50	0xffffe7d90 -> stack rw 0xf7e1cff4 -> (/usr/l
0xffffe7d54	0x7fec7d78
0xffffe7d58	0x7fec7b50
0xffffe7d5c	0xffffe7d80 -> @esi stack rw 0x00000002
0xffffe7d60	0x7e1cff4 -> (/usr/lib32/libc.so.6) library
0xffffe7d64	0x565c8ec -> (/home/user/Téléchargements/cr)
0xffffe7d68	0x00000000
0xffffe7d6c	0xf7c232c5 -> (/usr/lib32/libc.so.6) library
0xffffe7d70	0x00000000

Ensuite des qu'on continue on rentre dans la fonction check pal et on arrive à un point où il va comparer le registre %eax avec le registre %edx :

Disassembly:

```

0x565c81e3 cmpb    %al, %dl
0x565c81e5 je     0x565c81ee
0x565c81e7 movl    $0, %eax
0x565c81ec jmp     0x565c8203
0x565c81ee addl    $1, var_10h
0x565c81f2 subl    $1, var_-14h
0x565c81f6 movl    var_-10h, %eax
0x565c81f9 cmpi    var_-14h, %eax
0x565c81fc jl     $0x565c81cd
0x565c81fe movl    $1, %eax
0x565c8203 movl    var_-8h, %ebx
0x565c8206 leave
0x565c8207 retl
int main(int argc, char **argv, char **envp);
var int32_t var_14h @ stack - 0x14
arg char **argv @ stack + 0x4

```

Registers:

Register	Value
eax	0x78
eax	0x565c8288
ecx	0x565c81f4
edx	0x7e1e9b8
esi	0x565c82a4
edi	0x565c8220
esp	0xffffe7d40

Stack:

Offset	Value
0xffffe7d20	0xffffe7d68 -> stack rw
0xffffe7d24	0x7ee18d0 -> (/usr/l
0xffffe7d28	0x00000006
0xffffe7d2c	0x7ec7d2c -> stack rw
0xffffe7d30	0xffffe7d80 -> @esi stack
0xffffe7d34	0x565caff4 -> @ebx (/h
0xffffe7d38	0xffffe7d68 -> stack rw
0xffffe7d3c	0x565c828d -> (/home/u
0xffffe7d40	0xffffe0fc8 testmax
0xffffe7d44	0x7edf8cb -> (/usr/l
0xffffe7d48	0x7f7c1a2f_d1_audit_pr
0xffffe7d4c	0x565c8220 -> (/home/u
0xffffe7d50	0xffffe7d90 -> stack rw

0x78 est "x" et 0x74 est "t", on dirait qu'il compare notre premier et dernier caractère, bizarre.

Disassembly:

```

0x7ecd559 popl    %ebp
0x7ecd55a popl    %edx
0x7ecd55b popl    %ecx
0x7ecd55c retl
0x7ecd55d int3
0x7ecd55e nop
0x7ecd55f nop
0x7ecd560 nop
0x7ecd561 leal    (%esi), %esi
0x7ecd562 leal    (%esi), %esi
0x7ecd563 leal    (%esi), %esi
0x7ecd564 leal    (%esi), %esi
0x7ecd565 leal    (%esi), %esi
0x7ecd566 leal    (%esi), %esi
0x7ecd567 leal    (%esi), %esi
0x7ecd568 leal    (%esi), %esi
0x7ecd569 leal    (%esi), %esi
0x7ecd570 leal    (%esi), %esi
0x7ecd571 leal    (%esi), %esi
0x7ecd572 int    $0x80
0x7ecd573 leal    (%esi), %esi
0x7ecd574 leal    (%esi), %esi
0x7ecd575 leal    (%esi), %esi
0x7ecd576 leal    (%esi), %esi
0x7ecd577 leal    (%esi), %esi
0x7ecd578 leal    (%esi), %esi
0x7ecd579 leal    (%esi), %esi

```

Registers:

Register	Value
ebp	0x78
edx	0x565caff4
ecx	0x6
esi	0x74
esi	0xffffe7d80

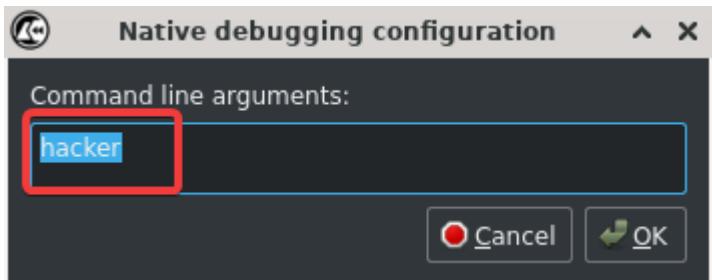
Console:

```

Continue until 0x565c8208
-----
----- CRACKME 7 -----
-----
** ACCESS DENIED **
(2722) Process exited with status=0x0

```

Et lorsqu'on continue d'avancer on a un access denied. Recommençons avec un autre mot.



Et il va encore comparé nos deux dernières lettres et encore une fois j'ai pas d'accès.

Registers	
eax	0x72
ebx	0x56611ff4
ecx	0x5
edx	0x68
esi	0xffff0d720

C'est la que je me suis dis que c'était peut être un palindrome, car deux fois il va vérifier nos deux dernières lettres, de plus la fonction s'appelle check\_pal ce qui laisse un indice. Donc j'ai voulu tester kayak dans l'application :

```
user@debian:~/Téléchargements$ ./crackmes_7 kayak
-----
----- CRACKME 7 -----
-----
** ACCESS GRANTED **
```

Ça fonctionne, testons d'autres palindrome :

```
user@debian:~/Téléchargements$ ./crackmes_7 sus
-----
----- CRACKME 7 -----
-----
** ACCESS GRANTED **
user@debian:~/Téléchargements$ ./crackmes_7 radar
-----
----- CRACKME 7 -----
-----
** ACCESS GRANTED **
```

Nous avons bien la confirmation. Vérifions une dernière fois comment l'application procède dans la fonction check\_pal :

The screenshot shows two windows from the Immunity Debugger. The top window is titled "Native debugging configuration" and contains a "Command line arguments:" field with "kayak" (highlighted with a red box). Below it are "Cancel" and "OK" buttons. The bottom window is titled "Registers" and lists CPU registers with their values: eax (0x6b), ebx (0x565f9ff4), ecx (0x4), edx (0x6b), esi (0xffe2dc0d), edi (0xf7f66b80), esp (0xffe2dc70), and ebp (0xffe2dc88). The eax and edx entries are highlighted with green boxes.

Il compare k et k.

The screenshot shows three windows from the Immunity Debugger. The top window is the "Disassembly" view, showing assembly code starting at address 0xf7f31559. The bottom window is the "Console" window, which displays the message "\*\* ACCESS GRANTED \*\* (2860) Process exited with status=0x0". A red box highlights the "Console" output.

Et une fois le débogage terminé nous avons l'accès autorisé. L'application va comparer dans une boucle while si le mot qu'on a rentré a ses caractères égaux de gauche à droite, alors il considère que c'est un palindrome.

## Deviner le mot de passe avec Cutter (challenge 8) :

**Première théorie :** Je pense qu'il vérifie le nombre d'octets. Sachant que 1 caractère = 1 octet et 1 entier = 8 octets, soit 32 bits .

**Deuxième théorie :** Il vérifie directement en **décimal** sur la table **ASCII** à quoi correspond ce que l'on a rentré, et il applique des conditions. Pour cela, nous devons nous aider de la table ASCII. Il effectue une vérification séquentielle des arguments entrés.

- Si c'est inférieur ou égal à 64 en décimal (caractère '@'), il saute la boucle et continue normalement. Voici ce que j'en ai déduit à la fin après avoir décortiqué check\_mi et check\_ma.

## ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

- Si c'est supérieur à 64, il refait une vérification et compare si le caractère est strictement supérieur à 90 en décimal ('Z'). S'il ne l'est pas, il incrémente la variable count de 1 (et sera comparé au prochain registre edi dans la fonction main) si le premier caractère du mot de passe est strictement inférieur à 'Z' (caractère ASCII 90)

Par exemple, si on prend le caractère ']' (91 en décimal), il entre dans la boucle car il est plus grand que 64, puis il refait une vérification du caractère pour voir s'il est strictement supérieur à 90 (décimal 'Z').

```

Disassembly
0x5662e1b3 addl    %edx, %eax
0x5662e1b5 movzbl  (%eax), %eax
0x5662e1b8 cmpb    $0x40, %al ; 64
0x5662e1ba jle     0x5662e1cf
0x5662e1bc movl    var_ch, %edx
0x5662e1bf movl    arg_4h, %eax
0x5662e1c2 addl    %edx, %eax
0x5662e1c4 movzbl  (%eax), %eax
0x5662e1c7 cmpb    $0x5a, %al ; 90
0x5662e1c9 je     0x5662e1cf
0x5662e1cb addl    $1, var_8h
0x5662e1cf addl    $1, var_ch
0x5662e1d3 movl    var_ch, %edx
0x5662e1d6 movl    arg_4h, %eax
0x5662e1d9 addl    %edx, %eax
0x5662e1db movzbl  (%eax), %eax

Stack
Offset Value
0xff873898 0xff873910 -> stack r
0xff87389c 0x56630ff4 -> eebx (/)
0xff8738a0 0x00000000
0xff8738a4 0x00000000
0xff8738a8 0xf8738d8 -> stack r
0xff8738ac 0xf8738d8 -> (/home/
0xff8738b0 0xf874fce [
0xff8738b4 0xf7f978cb -> (/usr/)
0xff8738b8 0xf7c1ca2f _dl_audit_p
0xff8738bc 0x5662e25a -> (/home/
0xff8738c0 0xf873900 -> stack r
0xff8738c4 0xf7f7f678
0xff8738c8 0xf8738f0 -> eesi st
0x6f073000 0x6f711f61 -> (/home/)

Registers
eax 0x56
ebx 0x56630ff4 91 eax ascii (']')

```

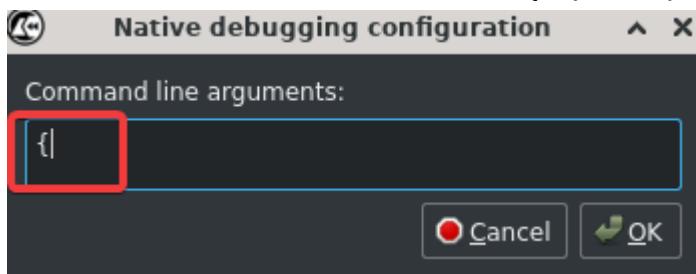
Ensuite, il va dans la deuxième fonction "check\_mi". Il vérifie si le caractère est supérieur à 96. Si ce n'est pas le cas, cela signifie que le caractère est compris entre 91 et 96 inclus (par exemple, '[' correspond à 91), donc il saute le jle.

```

Disassembly
0x5662e20d addl    %edx, %eax
0x5662e20f movzbl  (%eax), %eax
0x5662e212 cmpb    $0x60, %al ; 96
0x5662e214 jle     eip
0x5662e216 movl    var_ch, %edx
0x5662e219 movl    arg_4h, %eax
0x5662e21c addl    %edx, %eax
0x5662e21e movzbl  (%eax), %eax
0x5662e221 cmpb    $0x7a, %al ; 122
0x5662e223 jg     eip
0x5662e225 addl    $1, var_8h
0x5662e229 addl    $1, var_ch
0x5662e22d movl    var_cn, %eax
0x5662e230 movl    arg_4h, %eax
0x5662e233 addl    %edx, %eax
0x5662e235 movzbl  (%eax), %eax

```

Ensuite, il continue, et le logiciel autorise l'accès. Si maintenant on choisit le numéro 123 en décimal de la table ASCII, le '{', que se passera-t-il ?



Il passe le jle car c'est plus grand que 96 et il est de nouveau comparé à 122, soit 'z'.  
S'il est strictement supérieur, il saute la ligne : `0x565d9225 addl $1, var_8h`.  
var\_8h est une variable compteur, s'il elle est incrémentée, le programme comprendra qu'il y a un problème par la suite. Il vérifiera dans le main.

Disassembly

0x565c720d	addl	%edx, %eax
0x565c720f	movzbl	(%eax), %eax
0x565c7212	cmpb	\$0x60, %al ; 96
0x565c7214	jle	0x565c7229
0x565c7216	movl	var_ch, %edx
0x565c7219	movl	arg_4h, %eax
0x565c721c	addl	%edx, %eax
0x565c721e	movzbl	(%eax), %eax
0x565c7221	cmpb	\$0x7a, %al ; 122
0x565c7223	jg	0x565c7229
0x565c7225	addl	\$1, var_8h
0x565c7229	addl	\$1, var_ch
0x565c722d	movl	var_ch, %edx
0x565c7230	movl	arg_4h, %eax
0x565c7233	addl	%edx, %eax
0x565c7235	movzbl	(%eax), %eax

Disassembly

0x565c720d	addl	%edx, %eax
0x565c720f	movzbl	(%eax), %eax
0x565c7212	cmpb	\$0x60, %al ; 96
0x565c7214	jle	0x565c7229
0x565c7216	movl	var_ch, %edx
0x565c7219	movl	arg_4h, %eax
0x565c721c	addl	%edx, %eax
0x565c721e	movzbl	(%eax), %eax
0x565c7221	cmpb	\$0x7a, %al ; 122
0x565c7223	jg	0x565c7229
0x565c7225	addl	\$1, var_8h
0x565c7229	addl	\$1, var_ch
0x565c722d	movl	var_ch, %edx
0x565c7230	movl	arg_4h, %eax
0x565c7233	addl	%edx, %eax
0x565c7235	movzbl	(%eax), %eax

C'est ce que l'on peut voir dans le main lors de la prochaine comparaison entre eax et edi. S'ils ne sont pas égaux, le programme saute un certain nombre de lignes.

```

Disassembly
0x5663c2dd addl    $0x10, %esp
0x5663c2e0 cmpl    %eax, %edi
0x5663c2e2 jne     0x5663c2f8
0x5663c2e4 subl    $0xc, %esp
0x5663c2e7 leal    -0x1f92(%ebx), %eax
0x5663c2ed pushl    %eax      ; const char *
0x5663c2f3 calll    puts      ; sym.imp.puts ; int puts(const char *)
0x5663c2f5 addl    $0x10, %esp
0x5663c2f7 subl    $0xc, %esp
0x5663c2f8 leal    -0x1f7d(%ebx), %eax
0x5663c301 pushl    %eax      ; const char *
0x5663c302 calll    puts      ; sym.imp.puts ; int puts(const char *)
0x5663c307 addl    $0x10, %esp
0x5663c30a movl    $0, %eax
0x5663c30f leal    var_18h, %esp

Strings Search Memory Map Breakpoints Disassembly Graph(main) Hexdump
Decompiler (main)
undefined4 main(char **argv)
{
    undefined4 iVar1;
    int32_t iVar2;
    int32_t iVar3;
    int32_t iVar4;
    int32_t iVar5;
    int32_t iVar6;
    int32_t iVar7;
    int32_t iVar8;
    int32_t iVar9;
    int32_t iVar10;
    int32_t iVar11;
    int32_t iVar12;
    int32_t iVar13;
    int32_t iVar14;
    int32_t iVar15;
    int32_t iVar16;
    int32_t iVar17;
    int32_t iVar18;
    _x86.get_pc_thunk.bx();
    puts(unaff_EBX + 0xdae);
    puts(unaff_EBX + 0xdc5);
    puts(unaff_EBX + 0xddc);
    if (externaut_ECX == 0)
        return 0;
}

Registers
eax 0x1
ebx 0x5663eff4
ecx 0xf7e1e9b8
edx 0x1
esi 0xffffd8cb0
edi 0x0
esp 0xffffd8c80
ebp 0xffffd8c98

```

Et on voit qu'ils ne sont pas égaux, eax est à 1 et edi à 0, c'est l'incrémentation de tout à l'heure dans la fonction check\_mi. Il faut que eax et edi soient tous les deux à 0 pour que cela passe.

Donc je test :

```

Terminal - user@debian: ~/Téléchargements
Fichier Édition Affichage Terminal Onglets Aide
user@debian:~$ ./crackmes_8 {
bash: ./crackmes_8: Aucun fichier ou dossier de ce type
user@debian:~$ cd Téléchargements/
user@debian:~/Téléchargements$ ./crackmes_8 {
-----
----- CRACKME 8 -----
-----

** ACCESS GRANTED **

user@debian:~/Téléchargements$ 

----- ACCESS DENIED -----
user@debian:~/Téléchargements$ ./crackmes_8 a
-----
----- CRACKME 8 -----
-----

** ACCESS DENIED **

user@debian:~/Téléchargements$ 

```

Ma théorie est bonne !

