

Intro à la Rétro-ingénierie

Jeanne D'Hack CTF - 2025

Fenrisfulsur

Hello world !

Qu'est ce que la "Rétro-ingénierie" (ou "Ingénierie à rebours" en bon ANSSI) ?

Une méthode qui tente d'expliquer comment un mécanisme, un dispositif, un système ou un programme existant, accomplit une tâche sans connaissance précise de la manière dont il fonctionne.

Peut être effectué sur différentes cibles:

- Composants mécaniques
- Composants électroniques
- Softwares
- Matières chimiques
- Etc.

En informatique, le RE peut être fait pour:

- Retrouver des données ou un algorithme utilisés dans un exécutable
- Comprendre le format d'un fichier
- Documenter un protocole
- Etc.

Un peu de compilation

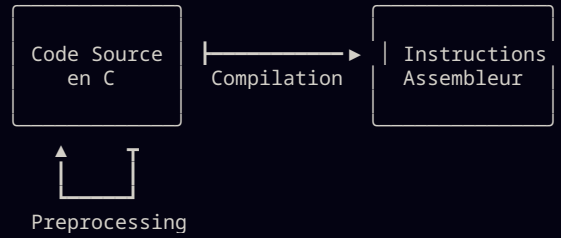
Code Source
en C

Un peu de compilation

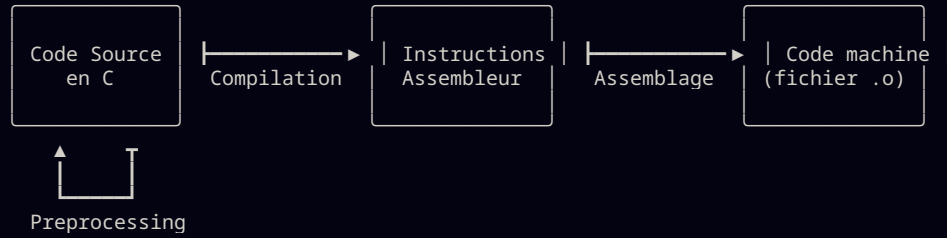
Code Source
en C



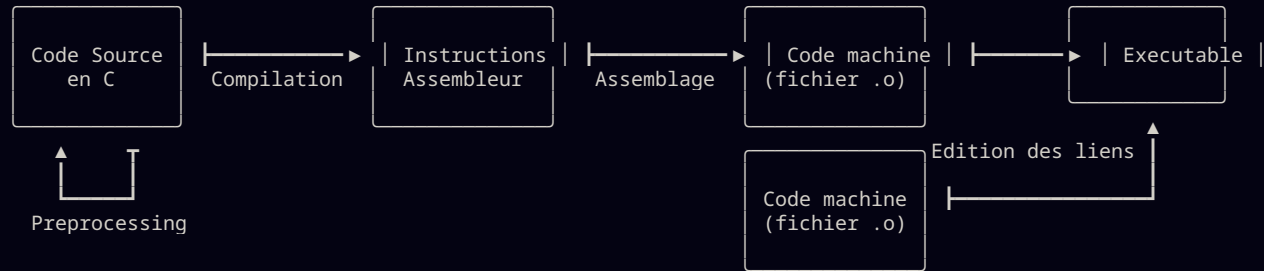
Un peu de compilation



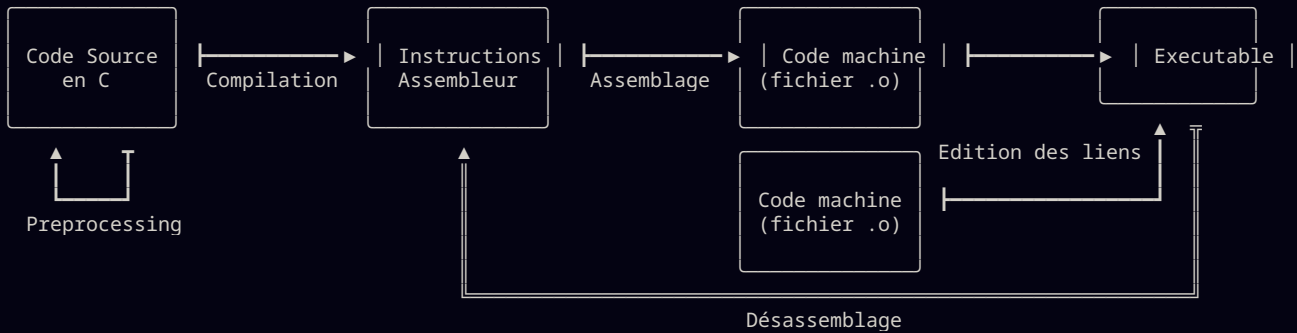
Un peu de compilation



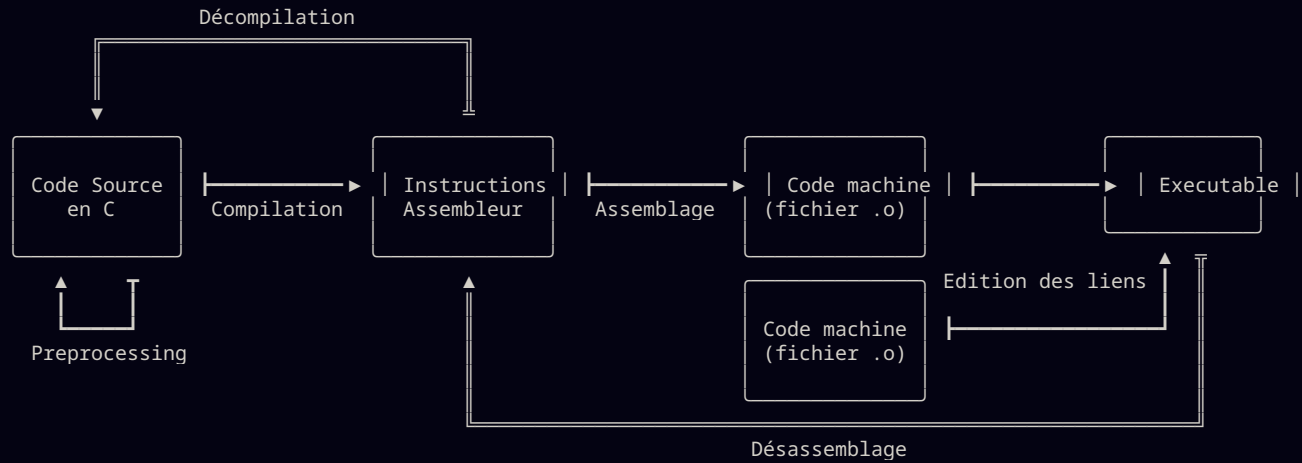
Un peu de compilation



Un peu de compilation



Un peu de compilation



Compilation: Demo

On commence par créer un petit programme en C

```
#define RETURN_VALUE    (42)

int foo(void) {
    return RETURN_VALUE;
}

int main(void) {
    return foo();
}
```

On utilise le préprocesseur avec la commande `gcc -E main.c`

```
int foo(void) {
    return (42);
}

int main(void) {
    return foo();
}
```

Compilation: Demo

On compile le code en assembleur avec la commande `gcc -S -fno-asynchronous-unwind-tables main.c`

```
.file    "main.c"
.text
foo:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $42, %eax
    popq    %rbp
    ret

main:
    pushq   %rbp
    movq    %rsp, %rbp
    call    foo
    popq    %rbp
    ret
```

L'option `-fno-asynchronous-unwind-tables` permet de désactiver la génération des CFI. <https://clang.llvm.org/docs/ControlFlowIntegrity.html>.

Compilation: Demo

On compile ensuite notre programme code machine avec la commande `gcc -o main.c`.

On obtient alors une fichier `main.o` au format ELF (fichier binaire).

```
$ xxd main.o | head
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF.....
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.....
00000020: 0000 0000 0000 0000 4802 0000 0000 0000  .....H.....
00000030: 0000 0000 4000 0000 0000 4000 0d00 0c00  ....@....@....
00000040: 5548 89e5 b82a 0000 005d c355 4889 e5e8  UH...*...].UH...
00000050: 0000 0000 5dc3 0047 4343 3a20 2847 4e55  ....]..GCC: (GNU
00000060: 2920 3134 2e31 2e31 2032 3032 3430 3532  ) 14.1.1 2024052
00000070: 3200 0000 0000 0000 0400 0000 2000 0000  2.....
00000080: 0500 0000 474e 5500 0200 01c0 0400 0000  ....GNU.....
00000090: 0000 0000 0000 0000 0100 01c0 0400 0000  .....
```

Compilation: Demo

Enfin on effectue l'édition des liens sur notre fichier objet afin d'obtenir un programme executable via la commande `gcc main.o -o main`.

On peut maintenant exécuter notre programme et afficher son code de retour:

```
$ ./main; echo $?  
42
```

Désassemblage

L'assemblage et le désassemblage sont des opérations de traduction d'un langage vers un autre, il est toujours possible de passer de l'un à l'autre; même si le résultat n'est pas toujours correct.

Pour cela, on peut utiliser la commande `objdump -d main` pour afficher le code assembleur de notre executable.

```
$ objdump -d main

main:      file format elf64-x86-64

Disassembly of section .init:
...

0000000000001119 <foo>:
   1119: 55                push    %rbp
   111a: 48 89 e5          mov     %rsp,%rbp
   111d: b8 2a 00 00 00    mov     $0x2a,%eax
   1122: 5d                pop     %rbp
   1123: c3                ret

0000000000001124 <main>:
   1124: 55                push    %rbp
   1125: 48 89 e5          mov     %rsp,%rbp
   1128: e8 ec ff ff ff    call    1119 <foo>
   112d: 5d                pop     %rbp
   112e: c3                ret
```

Comment fonctionne un ordinateur ?

Pour fonctionner un ordinateur utilise:

- un **CPU**: Composant électronique qui exécute les instructions;
- une **Mémoire**: Espace de stockage pour garder le résultat des calculs;
- un **Bus** de communication: Composant permettant d'interagir entre le CPU, la mémoire et les autres composants (écran, clavier, souris).

Un CPU possède plusieurs zones de stockage appelées **registres**. Il existe plusieurs tailles de registres (64, 32, 16 et 8):



Les registres

Registres Généraux

- **RAX**: Accumulateur;
- **RBX**: Base index (dans les tableaux par exemple);
- **RCX**: Counteur;
- **RDX**: Data/général;
- **RSI**: Source index pour les opérations sur des chaînes;
- **RDI**: Destination index pour les opérations sur des chaînes;
- **RSP**: Stack pointer (sommet de pile);
- **RBP**: Stack base pointer (adresse de la stack frame courante).

Registres Particuliers

- **RIP**: Pointeur d'instruction.
- **EFLAGS**: Registre d'état du processeur.

Le registre **EFLAGS** permet de connaître l'état du processeur et est implicitement mis à jour à chaque opération arithmétique.

Chaque bit du registre **EFLAGS** correspond à un flag:

- **ZF**: (zero flag) 1 lorsque le résultat est nul;
- **SF**: (sign flag) 1 lorsque le résultat est négatif;
- **CF**: (carry flag) la retenue lorsque le résultat ne tient pas dans le registre;
- **OF**: (overflow flag) 1 si le résultat signé ne tient pas dans le registre destination;
- **PF**: (parity flag) 1 si le nombre de bit d'un opérande est pair;
- **IF**: (interrupt flag) enlève la possibilité au processeur de contrôler les interruptions si sa valeur vaut 0;
- **DF**: (direction flag) détermine le sens de "lecture" des opérations de chaîne.

La pile

Les données volatiles d'un programme (variables, tableaux, adresses de retour de fonction, etc.) sont stockées dans un espace contigüe en mémoire appelé la **pile**.

- Principe LIFO (Last In First Out)
 - **push** -> ajoute une valeur au sommet de la pile
 - **pop** -> retire la dernière valeur ajoutée
- Par convention, la pile grandit vers les adresses basses.
 - le sommet de pile diminue lorsqu'une valeur est ajoutée
- Alignée sur 32 ou 64bits

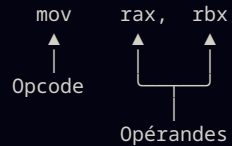


■ Structure d'un instruction

Une instruction ASM est constituée de plusieurs éléments:

- **Mnémonique** (ou **opcode**): spécifie l'opération à effectuer;
- **Opérande(s)** optionnel(s): paramètre(s) de l'opération.

Par exemple:



■ Mode d'adressage

Les opérandes d'une instruction peuvent être exprimées de plusieurs façons:

- **Registre**: valeur dans un registre. Ex: `mov rax, rbx`;
- **Directe**: valeur immédiate. Ex: `mov rax, 1234h`;
- **Indirecte**: valeur dans une case mémoire. Ex: `mov eax, dword ptr [0x401000]`;
- **Basée**: valeur indirecte via registre. Ex: `mov rdi, qword ptr [rsi]`;
- **Indexée**: Basée + Registre [+ Échelle]. Ex: `mov rax, qword ptr [rdx+rcx*8]`.

Jeu d'instructions x86

Affectations

- `mov destination, source`: Utilisée pour copier une valeur d'un endroit à un autre;
- `lea destination, [source]`: Calcule la valeur entre les crochets et la met dans destination.

Opérations sur la Pile

- `push opérande`: Décrémente `ESP/RSP` puis place la valeur de l'opérande en haut de la pile;
- `pop opérande`: Incrémente `ESP/RSP` puis Place la valeur du haut de la pile dans l'opérande.

Opérations arithmétiques

- `add destination, source`: Ajoute la valeur de source à la valeur de destination;
- `sub destination, source`: Soustrait la valeur de source à la valeur de destination;
- `inc opérande`: Ajoute 1 à la valeur du registre;
- `dec opérande`: Soustrait 1 à la valeur du registre;
- `mul source`: Multiplie RAX avec source;
- `mul destination, source`: Multiplie destination avec source;
- `div source`: Divise RAX par la valeur source.

Le résultat est stocké dans destination et les flags `OF`, `SF`, `ZF`, `AF`, `PF`, et `CF` sont modifiés en fonction du résultat de l'opération.

■ Opérations Logiques

Permet d'effectu  l'op ration logique bit   bit:

- `and destination, source`
- `or destination, source`
- `xor destination, source`
- `not destination`

OR	0	1	AND	0	1
0	0	1	0	0	0
1	1	1	1	0	1

XOR	0	1	NOT	/
0	0	1	0	1
1	1	1	1	0

Comparaisons

- `cmp opérande1, opérande2`: Soustrait la valeur de opérande2 à la valeur de opérande1 et ajuste les flags en fonction du résultat;
- `test opérande1, opérande2`: Fait un `&` logique entre opérande2 et opérande1 et ajuste les flags en fonction du résultat.

Branchements

- `jmp adresse`: Instruction de saut inconditionnel. Transfert l'exécution du programme à un autre endroit du code (TL;DR: Change la valeur de `RIP`).
- `jxx adresse`: Saut conditionnel
 - `je / jz`: jump if equal (`ZF==0`);
 - `jne / jnz`: jump if not zero (`ZF==1`);
 - `ja`: jump if above (`CF==0` et `ZF==0`);
 - `jb`: jump if below (`CF==1`);
 - `jg`: jump if greater (`ZF==0` et `SF==OF`);
 - `jl`: Jump if lower (`SF != OF`).

Appel à des fonctions

- `call adresse`:
 - Push l'adresse de retour sur la pile;
 - Met l'adresse dans `RIP`.
- `ret`:
 - Pop l'adresse en sommet de pile pour la mettre dans `RIP`.

Stack Frame

Qu'est ce qu'une "Stack Frame" ?

Il s'agit du contexte de pile dans laquelle s'exécute une fonction. Chaque fonction possède sa propre stack frame.

L'adresse de la base de la stack frame est dans `RBP`. La stack frame est la différence entre `RSP` et `RBP`.

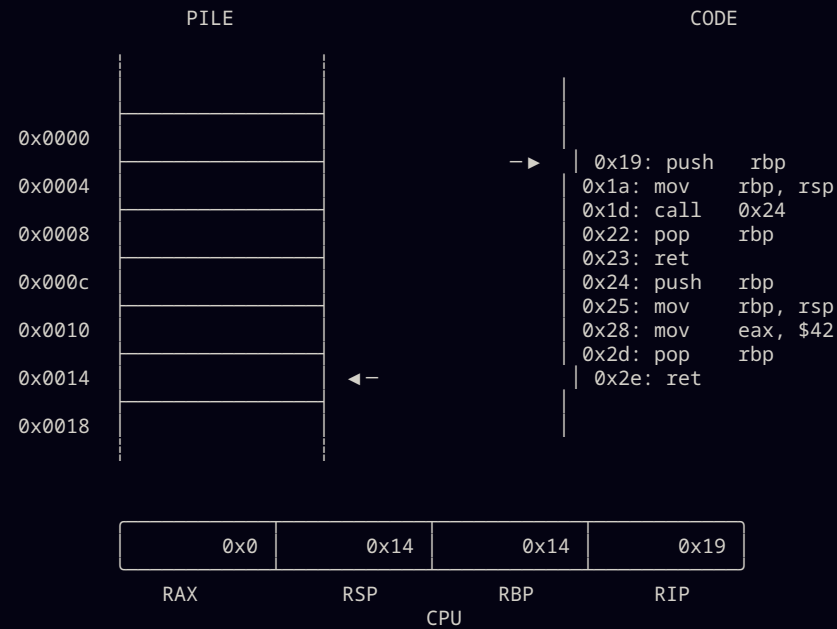
Une fonction commence **généralement** par un prologue:

```
push rbp
mov rbp, rsp
```

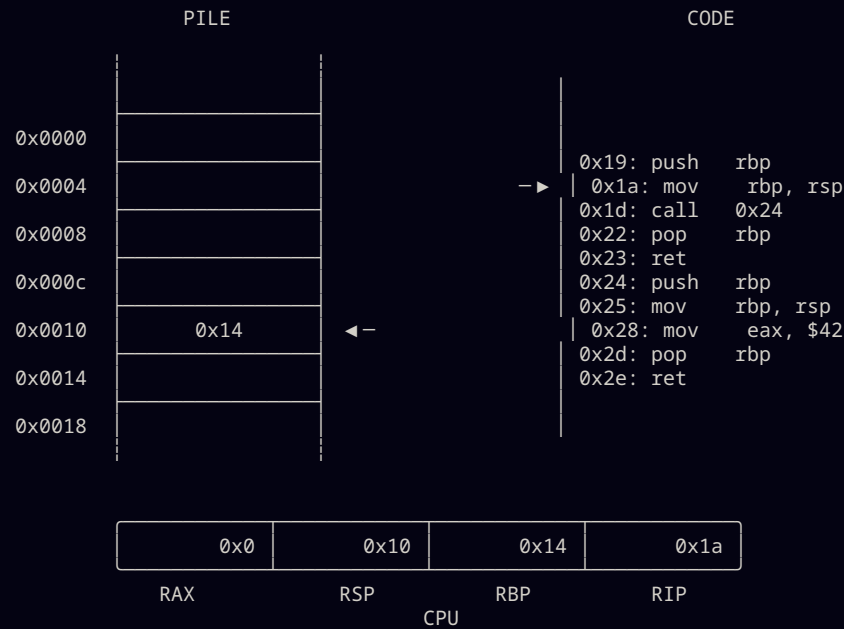
Et se termine par un épilogue (si prologue):

```
mov rsp, rbp
pop rbp
```

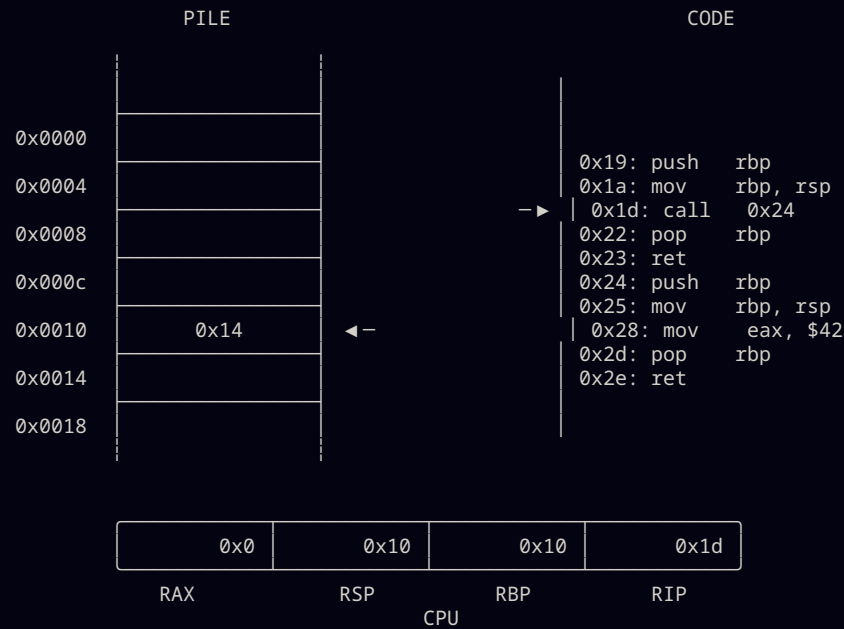
Que fait ce programme ?



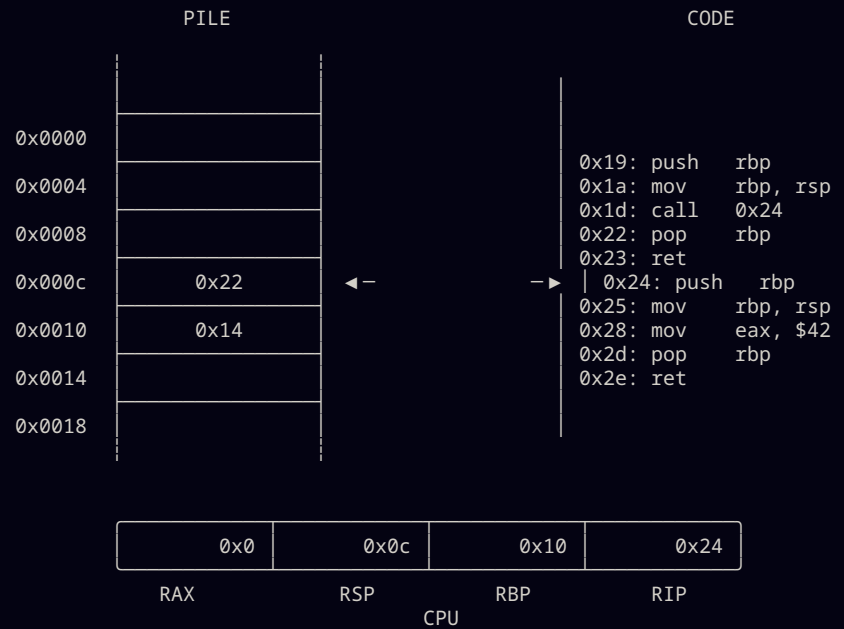
Que fait ce programme ?



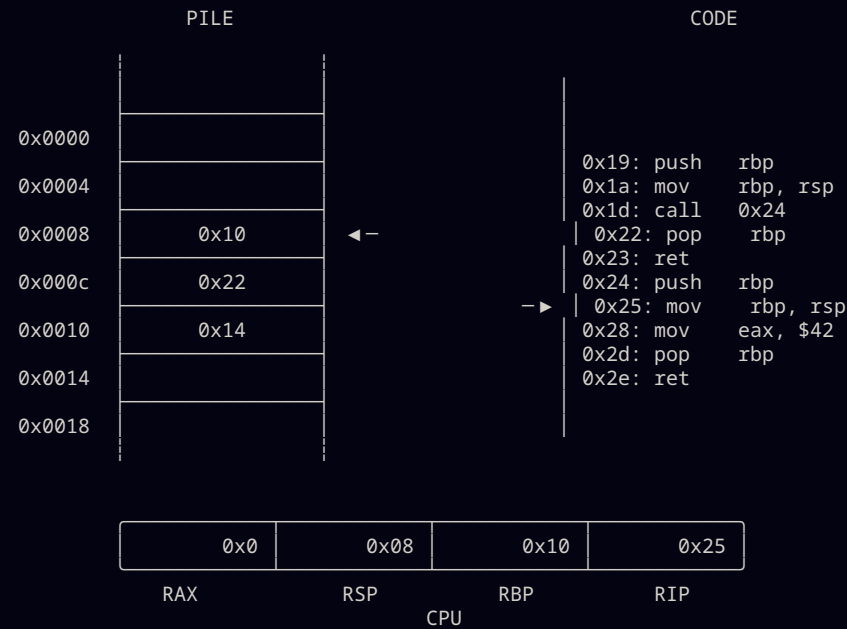
Que fait ce programme ?



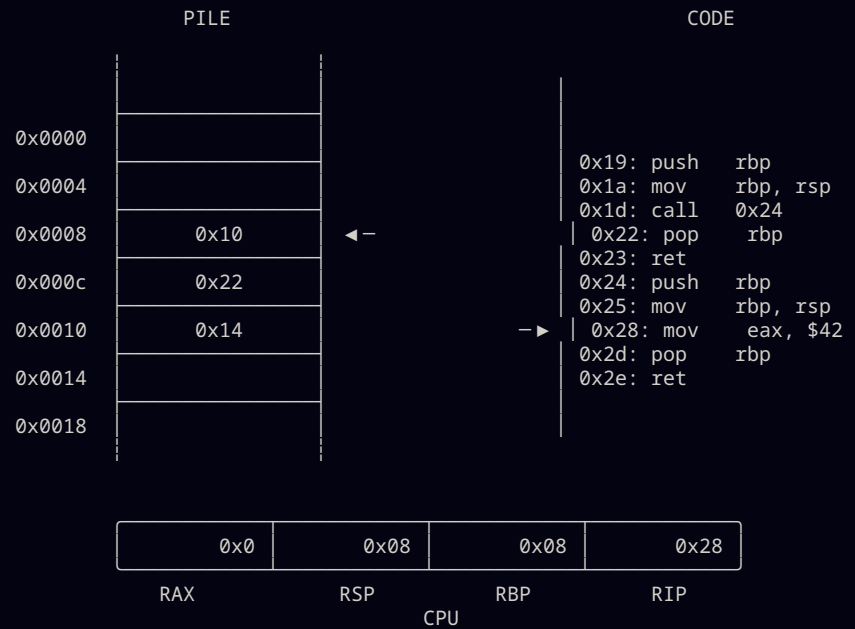
Que fait ce programme ?



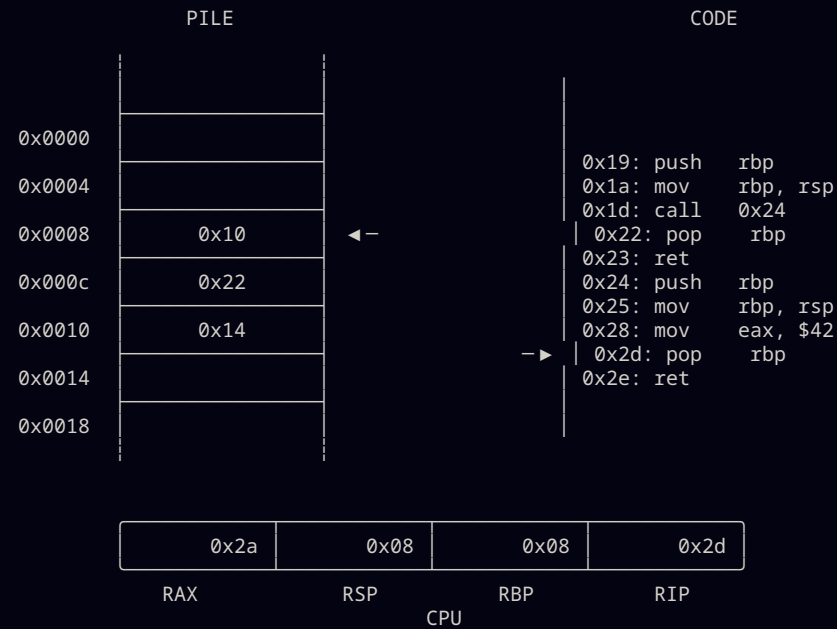
Que fait ce programme ?



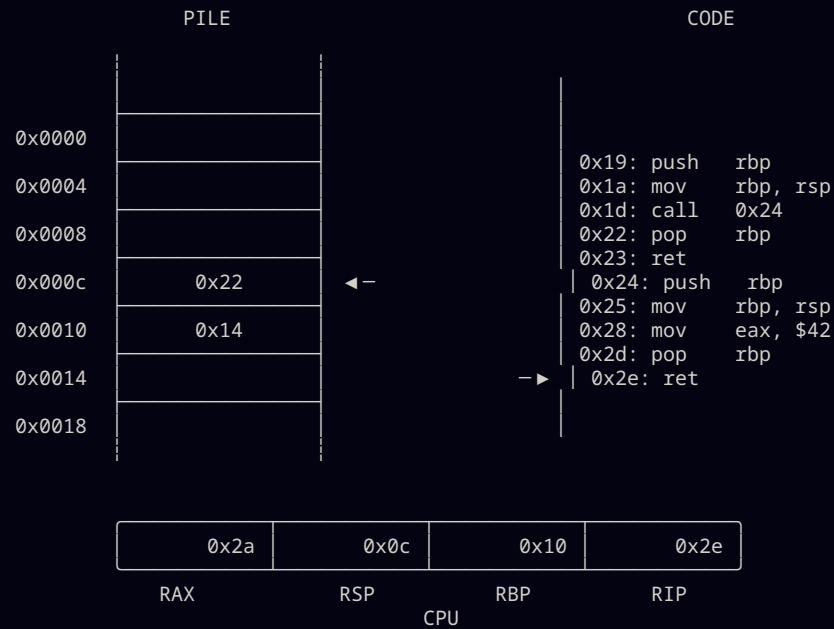
Que fait ce programme ?



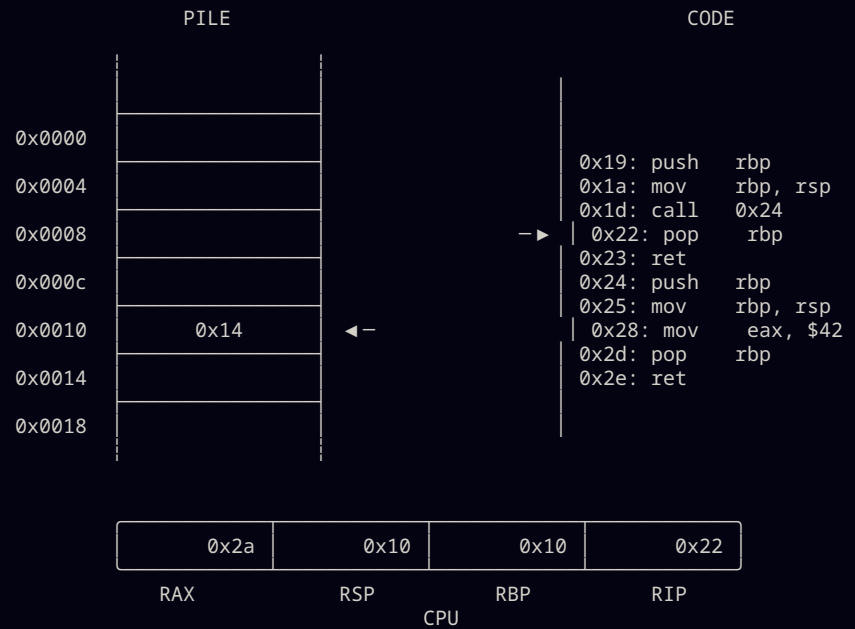
Que fait ce programme ?



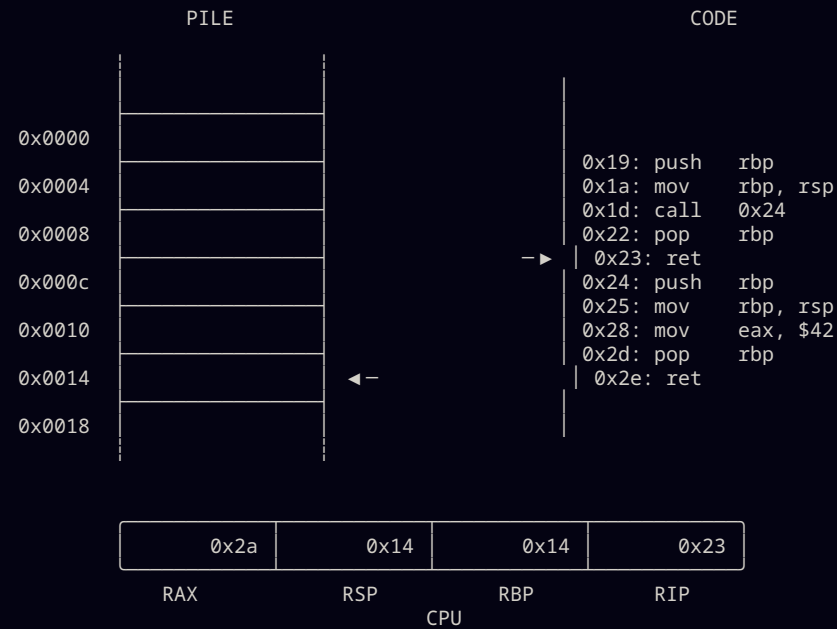
Que fait ce programme ?



Que fait ce programme ?



Que fait ce programme ?



Comment on fait dans la vraie vie ?

Problème:

Comment reverser un programme avec des milliers de fonctions ?

Solution:

De nombreux outils existent pour vous aider dans votre tâche. Il en existe 2 types:

- Outils d'analyse statique vont analyser le code sans l'exécuter, rechercher des patterns connus, des artefacts de compilation, etc.
- Outils d'analyse dynamique vont exécuter le code et permettre de voir les instructions exécuter une à une (tracing, hooking, etc).

Les outils les plus connus:

Nom	Auteur(s)	Prix	Decompilateur	API
angr	UCSB	Gratuit	Non	Python
Ghidra	NSA	Gratuit	Oui	Java/Python
IDA	HexRays	~3000\$	Oui	C++/Python
radare2	pancake	Gratuit	Non	Python
Binary Ninja	Vector 35	300*/1500\$	Oui	Python

Dans ce cours, nous utiliserons Ghidra.

Ghidra: Here be dragons

Ghidra est un logiciel gratuit et open-source développé par la NSA.

- Certains commentaires dans son code source indiquent qu'il existait déjà en 1999;
- Initialement rendu publique via WikiLeaks en mars 2017;
- Première sortie publique en 2019.

Principales caractéristiques:

- Multi-plateforme: Ghidra fonctionne sous Linux, MAC et Windows.
- Multi-architecture: Ghidra supporte de nombreuses architectures (x86, ARM, MIPS) et offre un désassembleur ainsi qu'un décompilateur.
- Ghidra dispose d'un gestionnaire de scripts pour automatiser des tâches répétitives. Les scripts peuvent être écrits en Python.



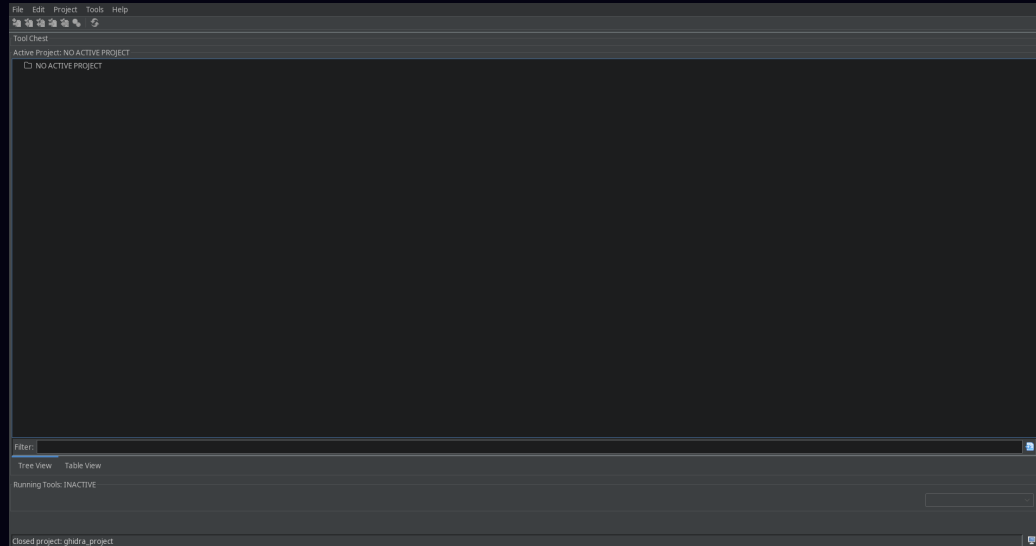
Ghidra: Installation

Comment installer Ghidra:

1. Télécharger l'archive zip depuis les releases github.
2. (Optionnel) Installer `openjdk 17` avec la commande suivante:

```
sudo apt install openjdk-17-jdk
$ java --version
openjdk 17.0.5 2022-10-18
```

1. Décompresser l'archive avec `unzip ghidra_XXXX.zip`.
2. Lancer `./ghidra_XXXX/ghidraRun`.

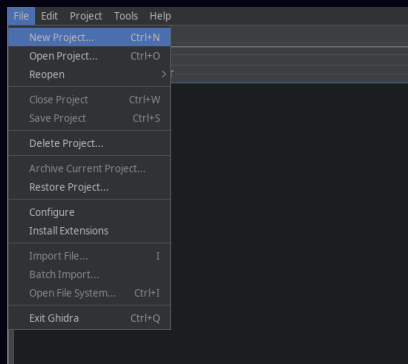


<https://github.com/NationalSecurityAgency/ghidra/releases>

Ghidra: Créer un nouveau projet

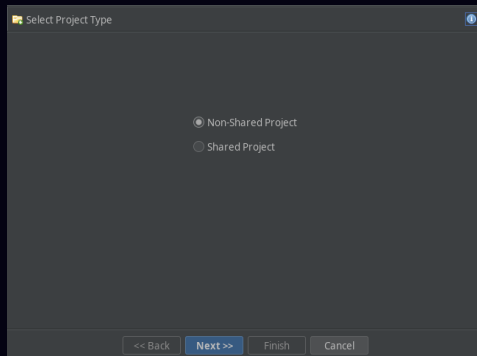
Les projets sous Ghidra sont des conteneurs générique permettant de sauvegarder votre travail (individuel ou collaboratif).

Pour commencer, cliquez sur **New Project**.



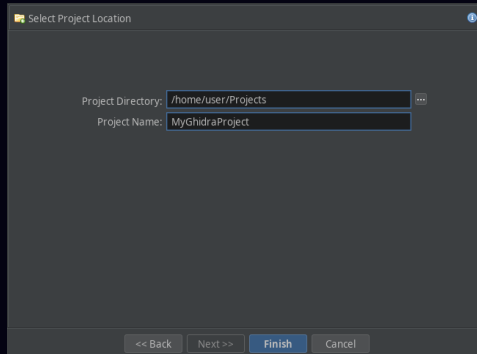
Ghidra: Créer un nouveau projet

Choisir un projet individuel **Non-Shared Project**.



Ghidra: Créer un nouveau projet

Choisissez le nom et l'emplacement de votre projet.



The image shows a 'Select Project Location' dialog box from the Ghidra application. It has a title bar with a folder icon and an information icon. The main area contains two text input fields: 'Project Directory:' with the value '/home/user/Projects' and a browse button '...', and 'Project Name:' with the value 'MyGhidraProject'. At the bottom, there are four buttons: '<< Back', 'Next >>', 'Finish' (highlighted in blue), and 'Cancel'.

Select Project Location

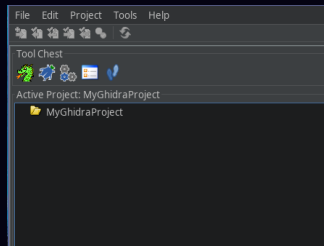
Project Directory: /home/user/Projects ...

Project Name: MyGhidraProject

<< Back Next >> Finish Cancel

Ghidra: Créer un nouveau projet

Votre notre nouveau projet est créé \o/.

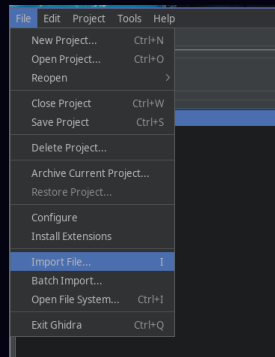


Ghidra: Importer un programme

On peut maintenant importer un programme à analyser. Ghidra reconnaît automatiquement les formats de fichiers connus :

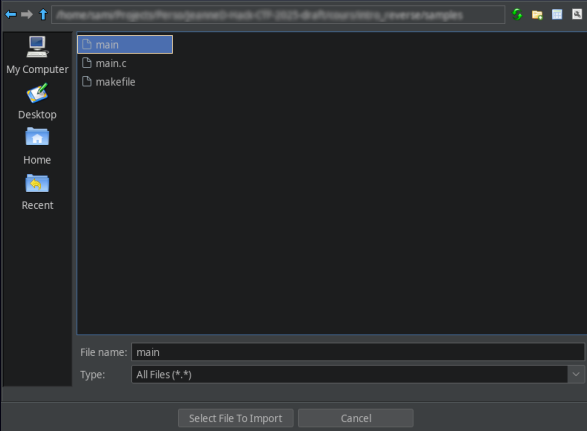
- ELF
- PE
- MachO
- Etc.

Pour importer un programme, cliquez sur **Import File**.



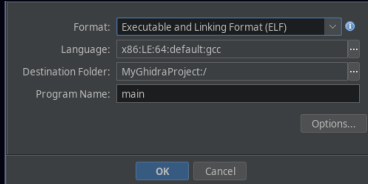
Ghidra: Importer un programme

Puis sélectionner votre fichier dans l'explorateur de fichier.



Ghidra: Importer un programme

Comme **Ghidra** à détecter que le format utilisé était un **ELF**, les paramètres d'analyse on déjà fixés, on peut cliquer sur **OK**



Format: Executable and Linking Format (ELF) [v] ⓘ

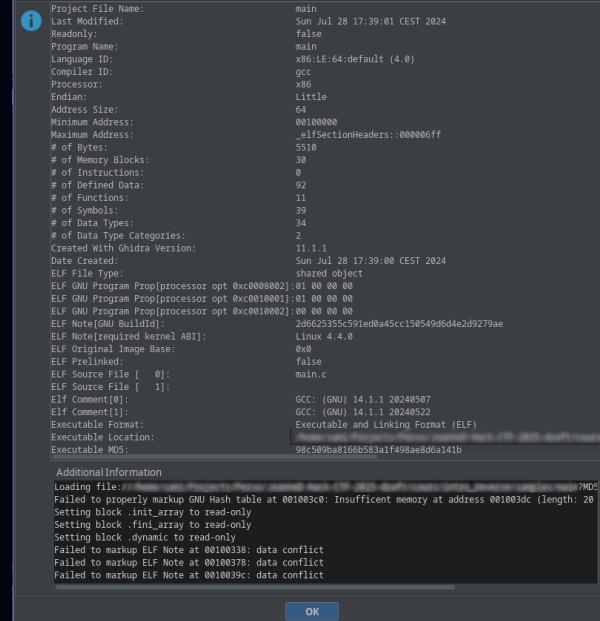
Language: x86_64:default:gcc [...]

Destination Folder: MyGhidraProject/ [...]

Program Name: main

Options...

OK Cancel



Project File Name: main

Last Modified: Sun Jul 28 17:39:01 CEST 2024

ReadOnly: false

Program Name: main

Language ID: x86_64:default (4.0)

Compiler ID: gcc

Processor: x86

Endian: Little

Address Size: 64

Minimum Address: 00100000

Maximum Address: _elfSectionHeaders::000006ff

of Bytes: 5510

of Memory Blocks: 30

of Instructions: 0

of Defined Data: 92

of Functions: 11

of Symbols: 39

of Data Types: 34

of Data Type Categories: 2

Created With Ghidra Version: 11.1.1

Date Created: Sun Jul 28 17:39:00 CEST 2024

ELF File Type: sharded object

ELF GNU Program Prop[processor opt 0xc0000002]: 01 00 00 00

ELF GNU Program Prop[processor opt 0xc0010001]: 01 00 00 00

ELF GNU Program Prop[processor opt 0xc0010002]: 00 00 00 00

ELF Note[GNU BuildId]: 2d625355c591ed0a45cc158549d6d4e2d9279ae

ELF Note[required kernel ABI]: Linux 4.4.0

ELF Original Image Base: 0x0

ELF PreLinked: false

ELF Source File [0]: main.c

ELF Source File [1]:

Elf Comment[0]: GCC: (GNU) 14.1.1 20240507

Elf Comment[1]: GCC: (GNU) 14.1.1 20240522

Executable Format: Executable and Linking Format (ELF)

Executable Location:

Executable MD5: 98c509ba816db583a1f498ae8d6a141b

Additional Information

Loading file: //

Failed to properly markup GNU Hash table at 001003c0: Insufficient memory at address 001003dc (length: 20)

Setting block .init_array to read-only

Setting block .fini_array to read-only

Setting block .dynamic to read-only

Failed to markup ELF Note at 00100338: data conflict

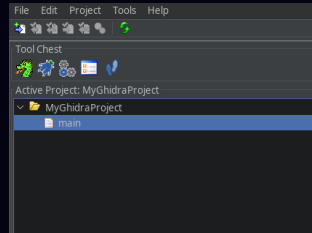
Failed to markup ELF Note at 00100378: data conflict

Failed to markup ELF Note at 0010039c: data conflict

OK

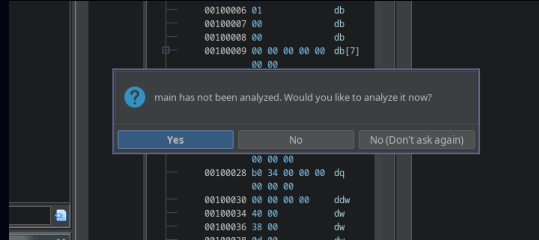
Ghidra: Importer un programme

Votre programme est importé et doit apparaître dans l'explorateur.



Ghidra: Analyser un programme

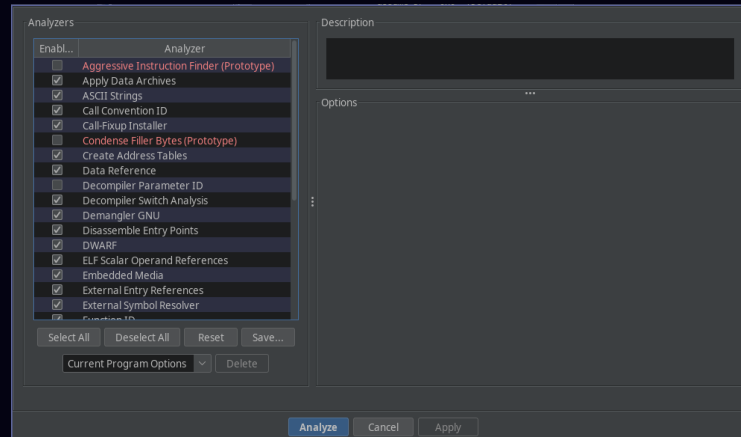
Lors de la première ouverture du programme, Ghidra va vous proposer d'analyser le programme. Cliquez sur **Ok**.



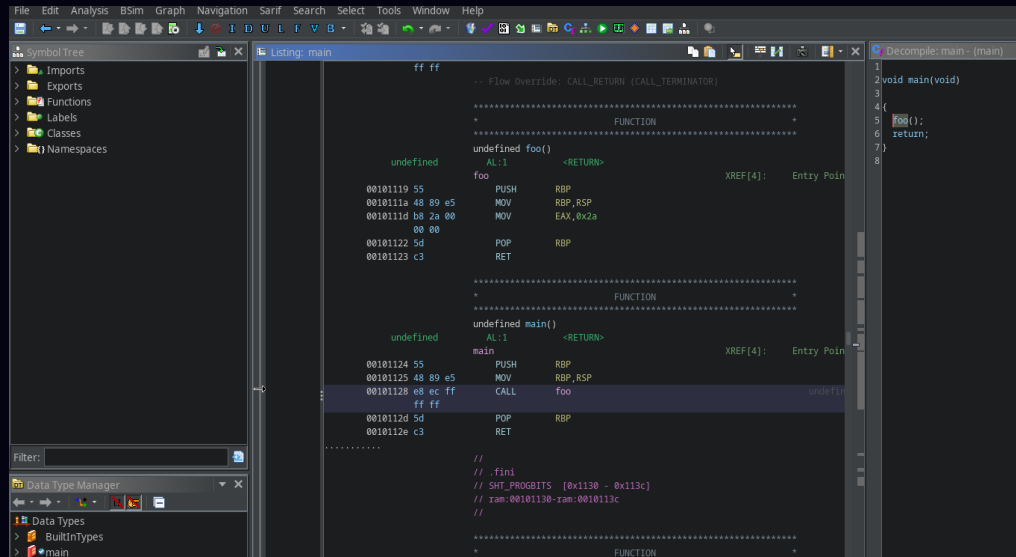
Ghidra: Analyser un programme

Il existe de nombreuses analyses disponibles permettant à Ghidra d'ajouter de la connaissance à votre analyse. Cependant, certaines analyses peuvent vous induire en erreur ou prendre beaucoup de temps.

Dans notre cas, on utilise les analyses par défaut et on clique sur **Ok**.




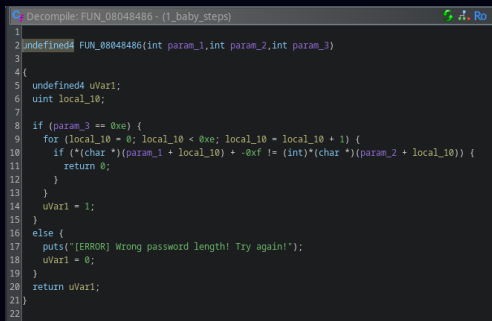
Ghidra: Les différentes fenêtres



Ghidra: Renommer, Retyper, Recommencer

Le travail d'un reverser consiste souvent à renommer/retyper des variables ou fonctions afin de:

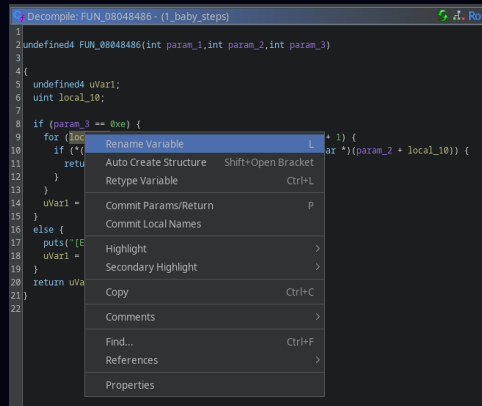
- Corriger les éventuelles erreurs du décompilateur (Oui oui ça arrive souvent ).
- Améliorer la connaissance que l'on a sur le programme.
- ~~Faire semblant de travailler.~~



```
Decompile: FUN_08048486 - (1 baby_steps)
1
2 undefined4 FUN_08048486(int param_1,int param_2,int param_3)
3
4 {
5     undefined4 uVar1;
6     uint local_10;
7
8     if (param_3 == 0xee) {
9         for (local_10 = 0; local_10 < 0xee; local_10 = local_10 + 1) {
10             if (*(char *)(param_1 + local_10) + -0xf != (int)*(char *)(param_2 + local_10)) {
11                 return 0;
12             }
13         }
14         uVar1 = 1;
15     }
16     else {
17         puts("[ERROR] Wrong password length! Try again!");
18         uVar1 = 0;
19     }
20     return uVar1;
21 }
22
```

Ghidra: Renommer, Retyper, Recommencer

Pour renommer une variable ou une fonction, on utilise **Clic droit** puis **Rename Variable**:



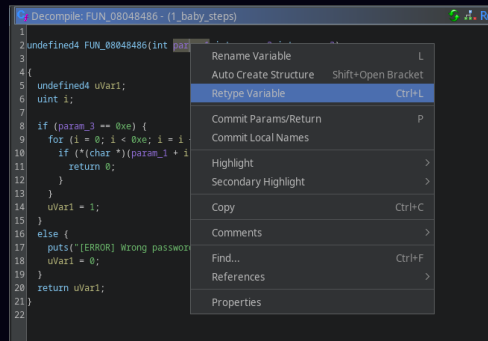
Ghidra: Renommer, Retyper, Recommencer

On obtient alors:

```
Decompile: FUN_08048486 - (1_baby_steps)
1
2 undefined4 FUN_08048486(int param_1,int param_2,int param_3)
3
4 {
5     undefined4 uVar1;
6     uint i;
7
8     if (param_3 == 0x0) {
9         for (i = 0; i < 0x0e; i = i + 1) {
10             if (*(char *)(param_1 + i) < -0xf != (int)(char *)(param_2 + i)) {
11                 return 0;
12             }
13         }
14         uVar1 = 1;
15     }
16     else {
17         puts("[ERROR] Wrong password length! Try again!");
18         uVar1 = 0;
19     }
20     return uVar1;
21 }
22
```

Ghidra: Renommer, Retyper, Recommencer

Pour retyper une variable ou une fonction, on utilise **Clic droit** puis **Retype Variable**:



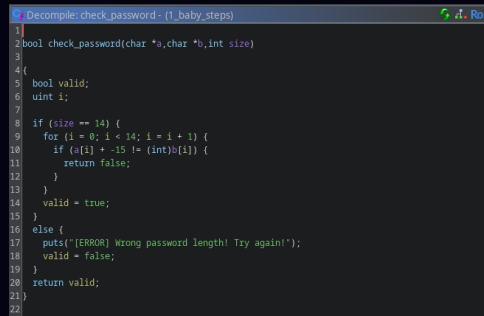
Ghidra: Renommer, Retyper, Recommencer

On obtient alors:

```
Decompile: FUN_08048486 - (1_baby_steps)
1
2 undefined4 FUN_08048486(char *param_1,char *param_2,int param_3)
3
4 {
5     undefined4 uVar1;
6     uint i;
7
8     if (param_3 == 0xe) {
9         for (i = 0; i < 0xe; i = i + 1) {
10             if (param_1[i] + -0xf != (int)param_2[i]) {
11                 return 0;
12             }
13         }
14         uVar1 = 1;
15     }
16     else {
17         puts("[ERROR] Wrong password length! Try again!");
18         uVar1 = 0;
19     }
20     return uVar1;
21 }
22
```

Ghidra: Renommer, Retyper, Recommencer

En répétant le processus plusieurs fois, on finit par obtenir:



```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
Decompile: check_password - (1_baby_steps)
1 bool check_password(char *a,char *b,int size)
2
3
4 {
5     bool valid;
6     uint i;
7
8     if (size == 14) {
9         for (i = 0; i < 14; i = i + 1) {
10             if (a[i] + -15 != (int)b[i]) {
11                 return false;
12             }
13         }
14         valid = true;
15     }
16     else {
17         puts("[ERROR] Wrong password length! Try again!");
18         valid = false;
19     }
20     return valid;
21 }
22
```

-> On a réussi à comprendre ce que faisait la fonction! Victoire \o/

A vous de jouer!

Objectif: Récupérer le programme sur Discord et retrouver le flag!



Questions?