

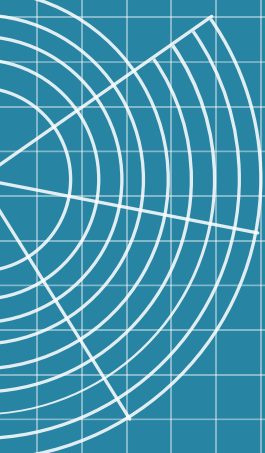
Intoduction au reverse engineering sous GNU/Linux avec radare2, cutter, gdb, ghida, ...



Association LinuX Maine
<https://linuxmaine.org/>
Session du 21/02/2026 a 14h
Thierry GAYET



<https://www.youtube.com/live/2uk3D4IjP4Y>



Titre : Utilisation de radar2, Ghidra, IDA, Wazuh, hachoir (python), exploitation de dump mémoire

Domaine : cyber sécu

Date: prévue le samedi 21/02/2026 à 14h

Niveau : avancé (forensic / reverse engineering)

Prérequis : bonne connaissance en programmation bas niveau

TP : disposer d'un ordinateur portable sous kali linux

Cette conférence-atelier propose une immersion avancée et pratique dans les techniques modernes de forensic numérique et de reverse engineering, à travers l'utilisation d'outils incontournables du domaine.

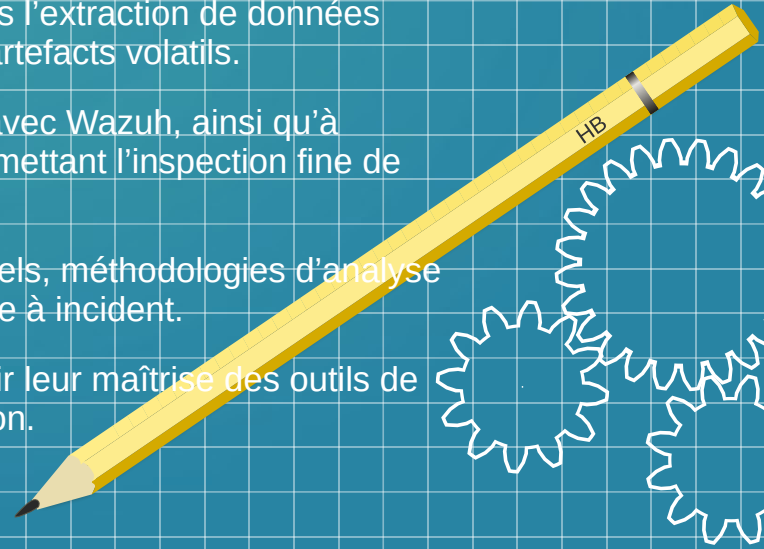
Les participants exploreront l'analyse de binaires et de malwares à l'aide de radare2, Ghidra et IDA, en mettant l'accent sur la compréhension du code bas niveau, la reconstruction de flux d'exécution et l'identification de comportements malveillants.

L'atelier abordera également l'exploitation de dumps mémoire, depuis l'extraction de données sensibles jusqu'à l'analyse de processus, structures en mémoire et artefacts volatils.

Un focus particulier sera consacré à l'outillage forensic et détection avec Wazuh, ainsi qu'à l'analyse de formats de fichiers complexes via Hachoir (Python), permettant l'inspection fine de données brutes, corrompues ou obfusquées.

L'approche sera concrète et orientée terrain : démonstrations, cas réels, méthodologies d'analyse et bonnes pratiques issues de scénarios d'investigation et de réponse à incident.

Cette session s'adresse aux profils techniques souhaitant approfondir leur maîtrise des outils de reverse engineering, forensic mémoire et analyse post-compromission.





- Cette presentation et a usage educatif ou documentaire uniquement. N'utilisez pas ces connaissances pour un usage pour des usages reprehensibles ou malveillants !



Messagerie sécurisée

Chiffré de bout-en-bout



Labellisé ANSSI



100% made in France

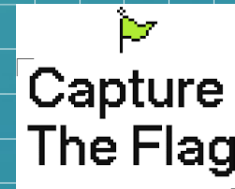


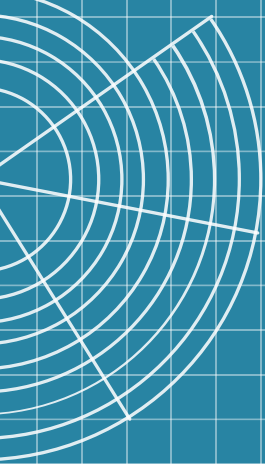
100% conforme RGPD

<https://www.tixeo.com>

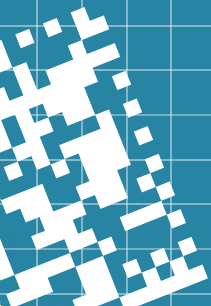
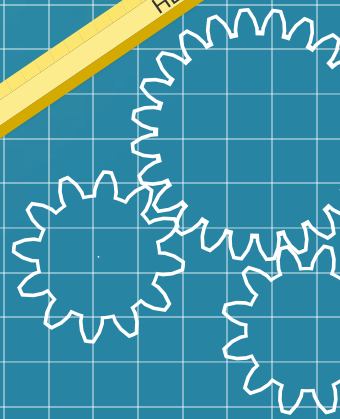
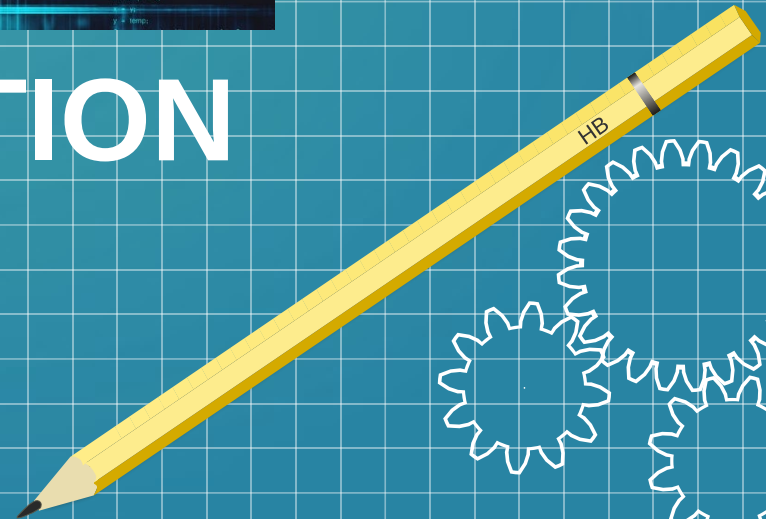
PLAN

- i. Introduction
- ii. Prérequis
- iii. Les outils
- iv. Le format ELF
- v. Reverse pas à pas
- vi. Annexes





I. INTRODUCTION



Introduction

Le reverse engineering de binaires sous Linux consiste à analyser un programme compilé (sans son code source) pour comprendre ce qu'il fait, comment il le fait, et éventuellement le modifier ou trouver des vulnérabilités.

Objectifs et cas d'usage

- **Comprendre le fonctionnement** d'un binaire inconnu (outil interne sans source, logiciel propriétaire, malware).
- **Auditer la sécurité** : chercher des vulnérabilités classiques (buffer overflow, format string, logique d'auth, crypto maison).
- **Déboguer ou instrumenter** du code quand les symboles ou les sources ne sont plus disponibles.
- **Interopérabilité** : documenter un protocole ou un format de fichier à partir du binaire qui le manipule.

Toujours garder en tête le cadre légal : selon la licence du logiciel et la législation locale, certaines formes de reverse sont restreintes (ex. contournement de DRM).

Binaire Linux : ELF, sections et symboles

Sous Linux, les exécutables et bibliothèques partagées sont en général au format ELF (Executable and Linkable Format).

Un ELF contient notamment :

- **Une en-tête** qui décrit l'architecture, le point d'entrée, le type (exécutable, librairie).
- **Des sections** comme .text (code), .data (données initialisées), .bss (données non initialisées), .rodata (constantes).
- **Des segments** utilisés pour le mapping en mémoire (code, data, heap, stack).
- *Éventuellement* **une table des symboles**, qui donne des noms aux fonctions et variables dans les binaires non « stripped ».

Comprendre cette structure (avec readelf, objdump, file, etc.) est la première étape avant d'attaquer l'assembleur ou la dynamique.

Deux grandes approches : statique et dynamique.

Approches statique

Analyse statique : tu ne lances pas le programme, tu inspectes directement le binaire.

- **Désassemblage/décompilation** avec Ghidra, IDA, Binary Ninja, radare2/Cutter, objdump.
- **Navigation dans le graphe de contrôle**, reconstruction de fonctions, renommage de variables, commentaires.
- **Recherche de patterns** (appel à fonctions sensibles, constantes, structures de données).

Approches dynamique

Analyse dynamique : tu exécutes le binaire dans un environnement contrôlé et tu observes son comportement.

- **Traces système** (strace, ltrace) pour voir appels système et fonctions de libs.
- **Débogage avec gdb** (breakpoints, watchpoints, inspection mémoire, patch à la volée).
- **Monitoring réseau** (tcpdump/Wireshark) ou instrumentation plus avancée (Frida, ptrace, QEMU, sandbox).

En pratique, on combine toujours les deux : la statique donne la vue d'ensemble, la dynamique permet de valider des hypothèses et de contourner l'obfuscation.

Méthodologie typique

Reconnaissance rapide :

- File
- checksec,
- Ldd
- strings pour savoir si le binaire est 32/64 bits dynamique/statique, protégé (NX, PIE, RELRO)
quelles libs il charge quelles chaînes de caractères il contient

Méthodologie typique

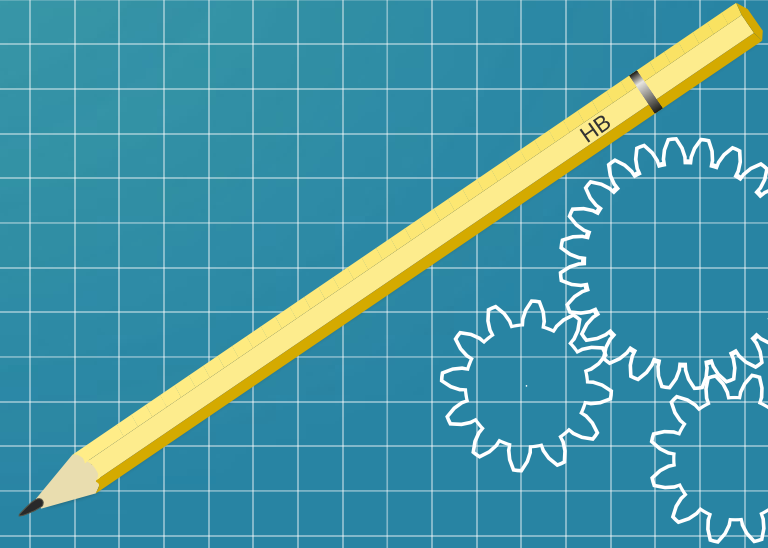
Cartographie du binaire :

- readelf -h -l -S pour voir sections et segments.
- Chargement dans ton IDE de reverse (Ghidra / r2 / Cutter) : lancement de l'analyse auto, repérage de main, des callbacks, des fonctions qui font des I/O, de la crypto, etc.

Méthodologie typique

Compréhension de la logique :

- Renommer les fonctions (par rôle), insérer des commentaires, tracer le flux d'exécution (graphes de fonctions, CFG).
- Identifier les structures (par exemple à partir des accès mémoire et offsets) pour reconstruire des struct.



Méthodologie typique

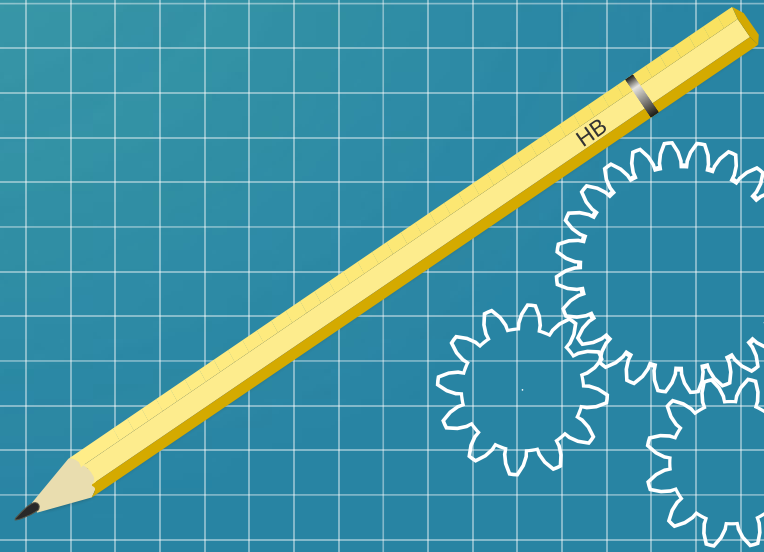
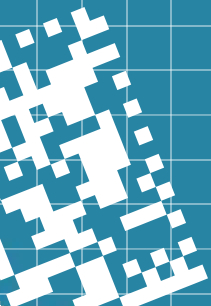
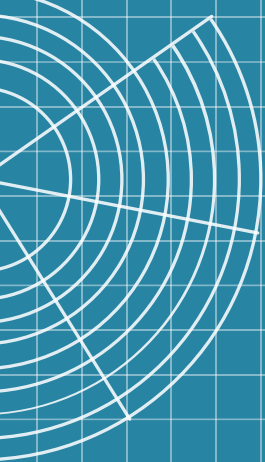
Validation par exécution :

- Lancer le programme avec des entrées de test, observer ce qui se passe en dynamique.
- Placer des breakpoints sur les fonctions critiques (authentification, chiffrement, traitement de paquets) et suivre l'état des registres/stack.

Méthodologie typique

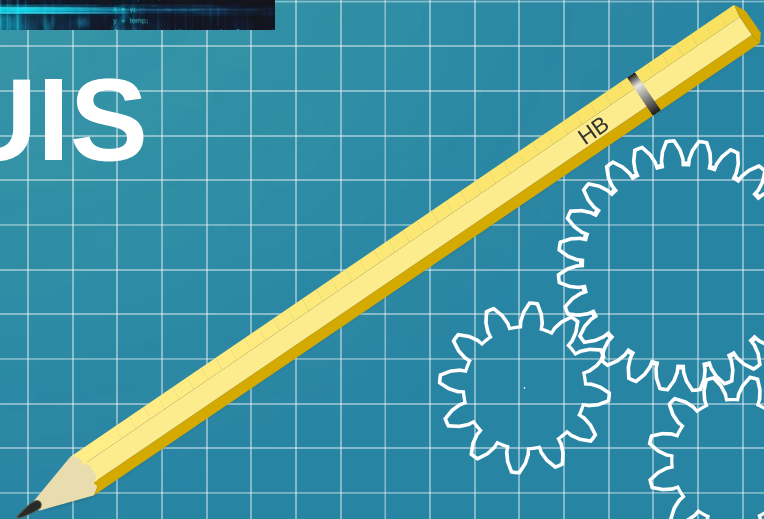
Documentation / exploitation :

- Écrire un petit rapport : ce que fait le binaire, protocole, formats, vulnérabilités éventuelles, patches possibles.

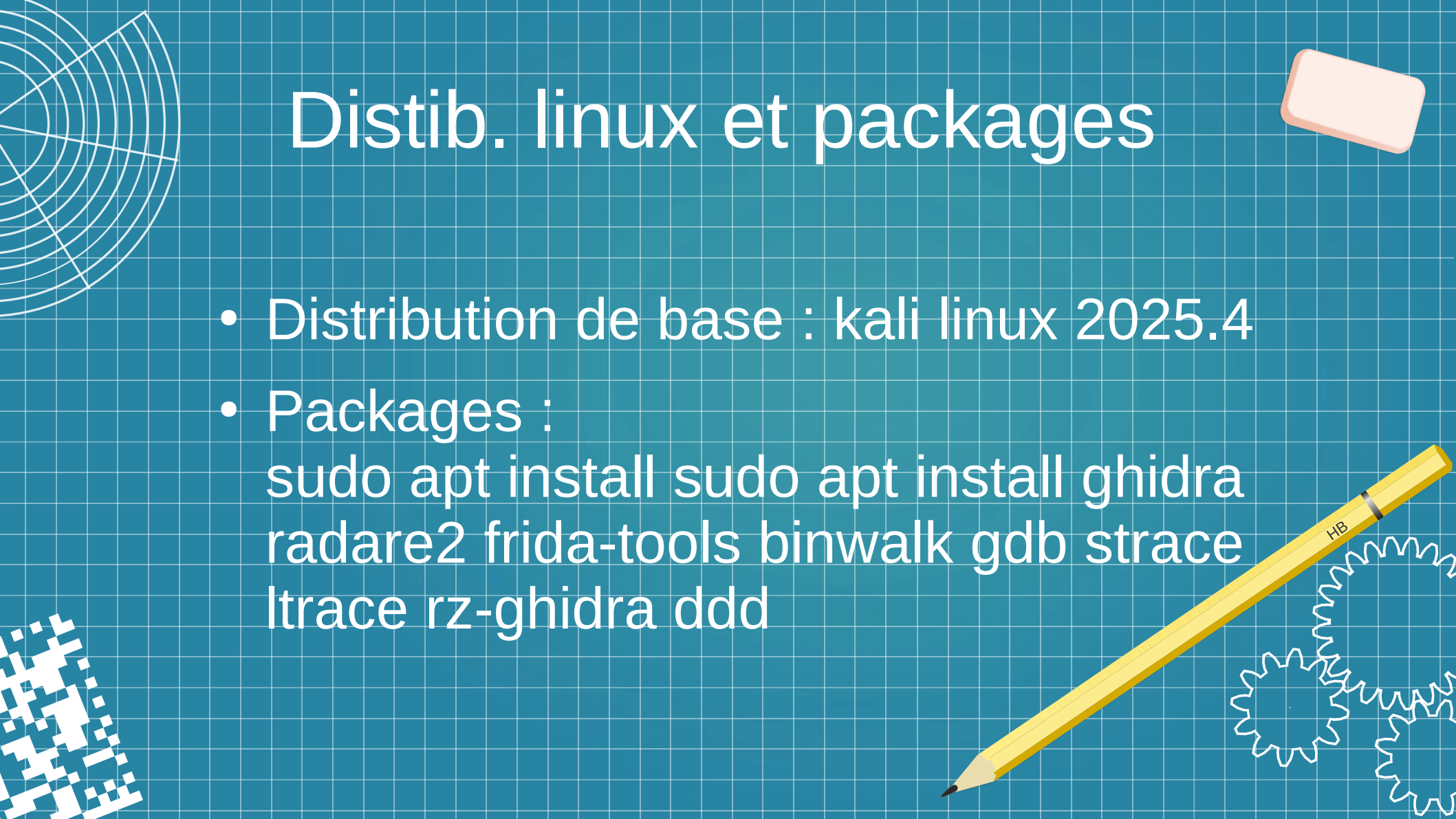




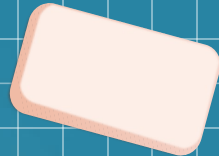
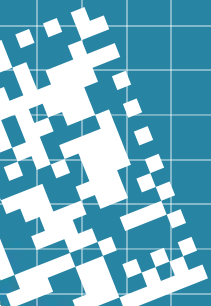
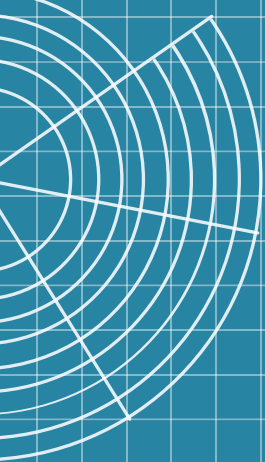
II. PREREQUISIS



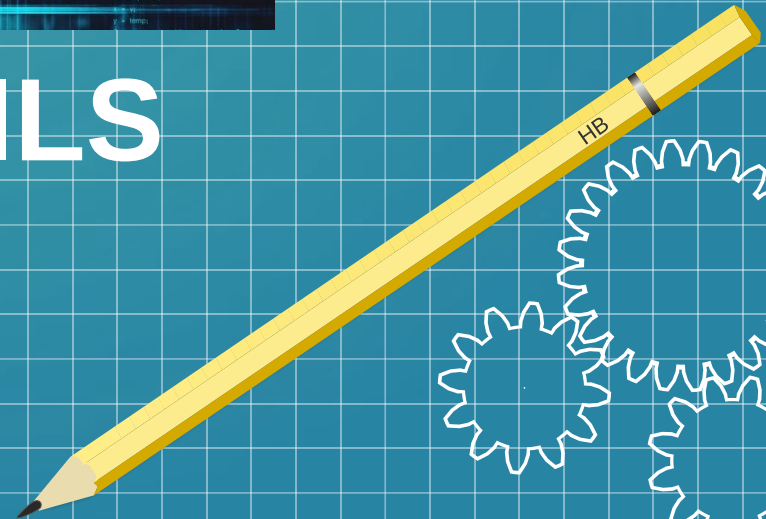
Distib. linux et packages



- Distribution de base : kali linux 2025.4
- Packages :
sudo apt install sudo apt install ghidra
radare2 frida-tools binwalk gdb strace
ltrace rz-ghidra ddd



III. LES OUTILS



Binary & Reverse Engineering

☒ Ghidra

☒ IDA Free

☒ Radare2

☒ Cutter

☒ Binary Ninja

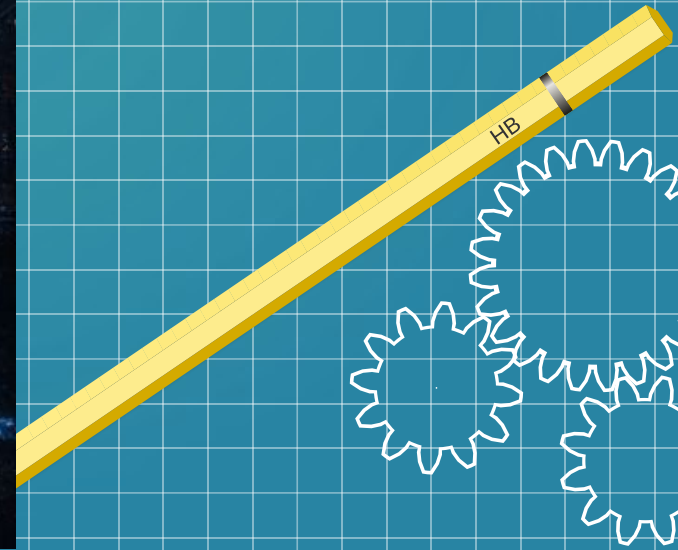
☒ x64dbg

☒ WinDbg

☒ OllyDbg

☒ Hopper

☒ RetDec

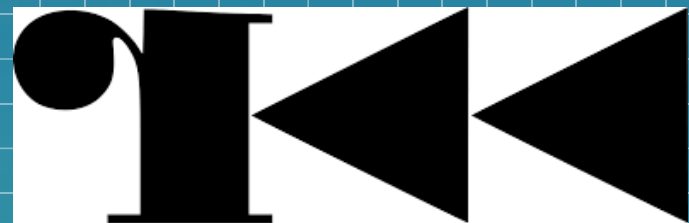


RADARE2



Radare2 Debugging
Part 1 - R2 Basics

- An advanced command-line reverse engineering framework.
- Supports debugging, disassembly, and binary patching.
- Preferred by advanced reverse engineers.
- Link: <https://github.com/radareorg/radare2>

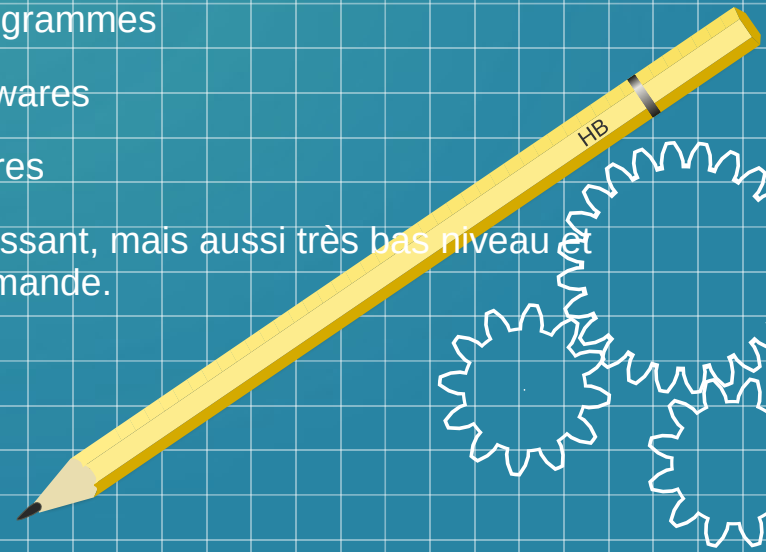


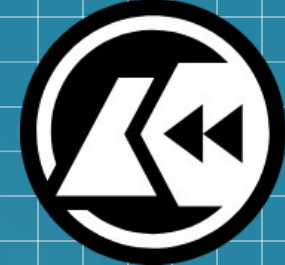
radare2 est une boîte à outils libre de reverse engineering et d'analyse binaire.

Il est utilisé pour :

-  analyser des exécutable (ELF, PE, Mach-O, WASM...)
-  désassembler et décompiler (partiellement)
-  debugger des programmes
-  analyser des malwares
-  patcher des binaires

C'est un outil très puissant, mais aussi très bas niveau et orienté ligne de commande.





Cutter est une interface graphique (GUI) moderne pour radare2.

Son objectif : rendre la puissance de radare2 beaucoup plus accessible et visuelle.

👉 Cutter = radare2 + GUI

👉 Souvent vu comme l'alternative libre à IDA / Ghidra, mais orientée radare2.

Cutter permet de :

🔍 analyser des binaires (ELF, PE, Mach-O, WASM...)

🧩 désassembler avec une interface graphique

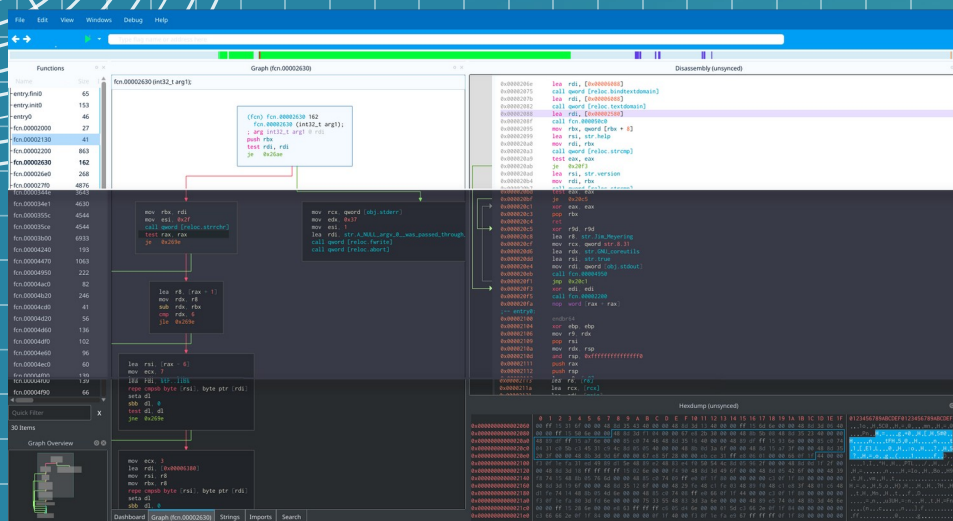
📊 visualiser des graphes de fonctions

🔍 explorer strings, sections, symboles

🔧 patcher des binaires

🐞 déboguer (via radare2)

👉 Idéal si tu aimes radare2 sans le côté 100 % CLI.



ANGR



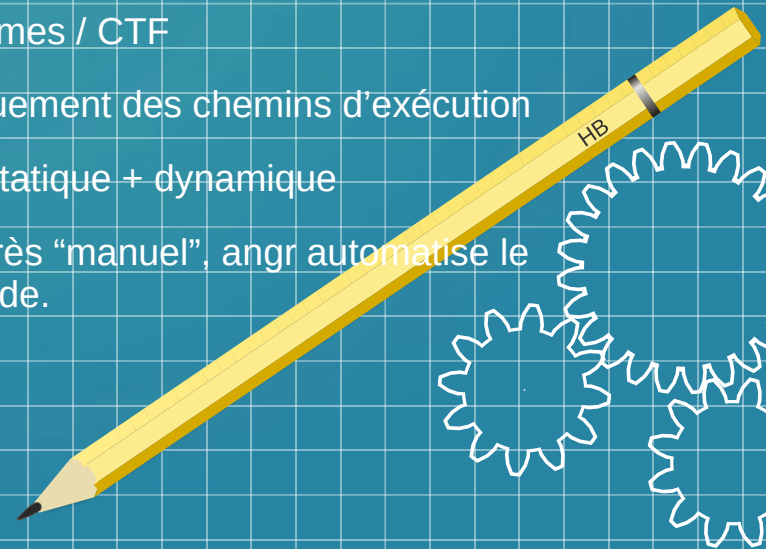
- A binary analysis framework using symbolic execution.
- Automates vulnerability discovery and program analysis.
- Used in advanced security research and automation.
- Link: <https://angr.io>



angr est un framework d'analyse binaire avancée en Python, orienté analyse statique, dynamique et surtout symbolique.

Il est utilisé pour :

-  analyser des binaires sans code source
-  faire de l'exécution symbolique
-  résoudre des crackmes / CTF
-  explorer automatiquement des chemins d'exécution
-  combiner analyse statique + dynamique
-  Là où radare2 est très "manuel", angr automatise le raisonnement sur le code.



GHIDRA



- A powerful reverse engineering suite developed by the NSA.
- Includes disassembler and decompiler for binary analysis.
- Ideal for malware analysis and exploit research.
- Link: <https://ghidra-sre.org>

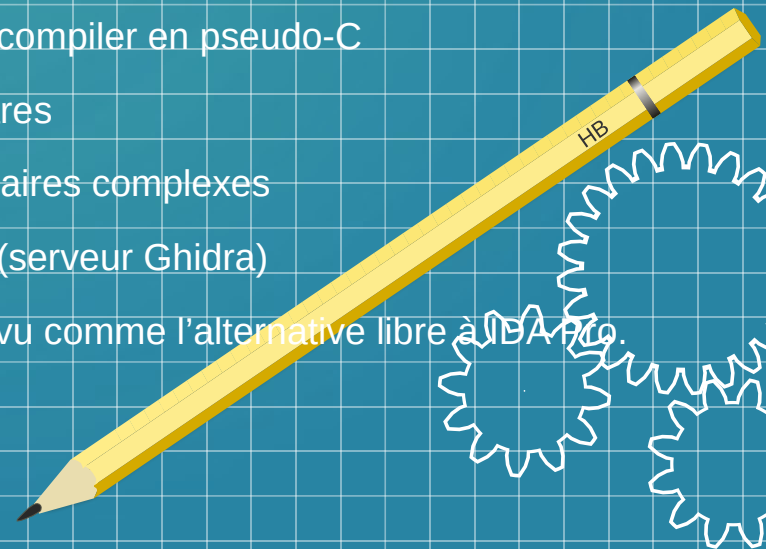


Ghidra est une suite de reverse engineering open-source développée par la NSA.

C'est un outil graphique, très complet, orienté analyse statique, avec un décompilateur puissant.

Il est utilisé pour :

- 🔍 analyser des binaires sans code source
- 🧩 désassembler et décompiler en pseudo-C
- 🐞 analyser des malwares
- 🔧 comprendre des binaires complexes
- 🤝 travailler en équipe (serveur Ghidra)
- 👉 Ghidra est souvent vu comme l'alternative libre à IDA Pro.





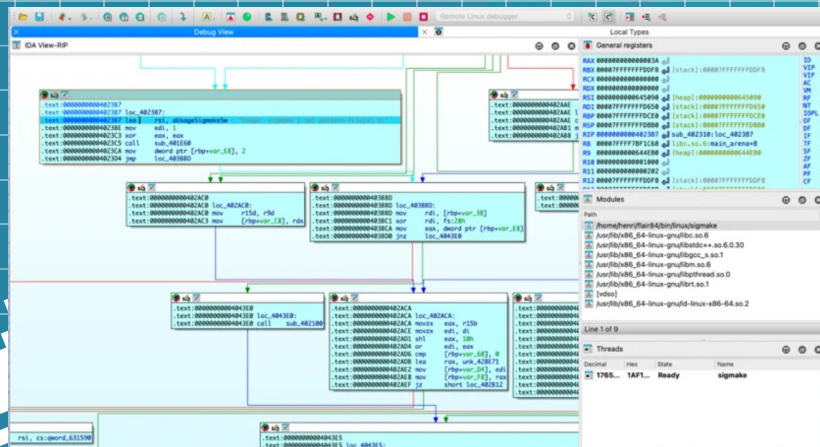
FREE IDA Pro Free Linux :

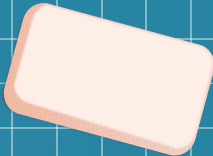
✓ Ce que permet IDA Free

- 🔍 Analyse statique
- 🧩 Désassemblage interactif
- 📊 Graphes de fonctions
- 🔍 Recherche de chaînes et symboles
- 🧠 Interface mature et stable

✗ Limitations importantes

- ✗ Pas de décompilateur Hex-Rays (pas de pseudo-C)
- ✗ Support limité des architectures
- ✗ Pas de debugging avancé
- ✗ Pas de plugins commerciaux
- 👉 IDA Free ≠ IDA Pro complet





GDB (GNU Debugger) est le débogueur standard sous Linux pour les programmes compilés (C, C++, Rust, Go, etc.).

Il permet de :

-  déboguer des programmes pas à pas
-  inspecter la mémoire, les registres, la pile
-  suivre les threads et processus
-  comprendre les crashes (segfault)
-  analyser des binaires avec ou sans symboles
-  C'est l'outil indispensable côté runtime.

DDD (Data Display Debugger) est une interface graphique pour GDB (et d'autres debuggers).

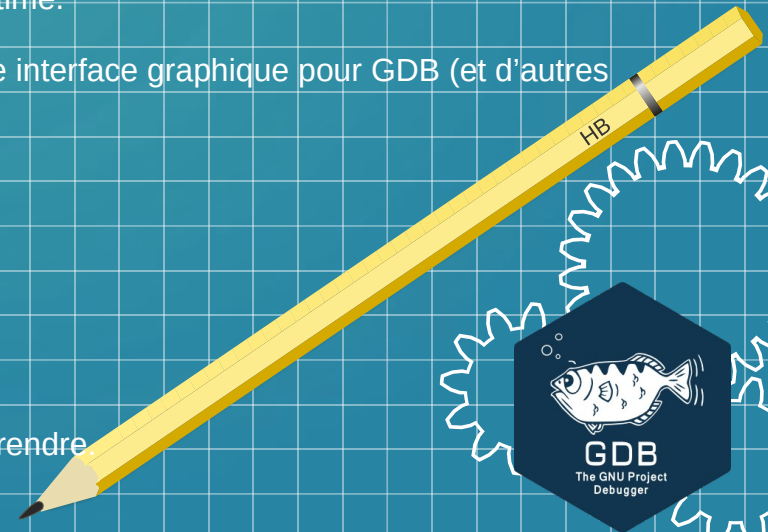
👉 DDD = GDB + GUI X11

vue du code

graphes d'appels

inspection visuelle des variables

! Ancien mais encore utile pour apprendre



✓ Reconnaissance

- Recon-ng
- theHarvester
- Nmap
- Zenmap
- DNSRecon
- Netdiscover
- Maltego
- UnicornsCan
- p0f
- Masscan
- ZMap

✓ Vulnerability Scanning

- OpenVAS
- w3af
- Nikto
- Vuls
- Nessus \$

✓ Network-based Attacks

- Wireshark
- Etttercap
- ArpSpooF
- NetCat
- dSniff
- Scapy
- hping3
- Yersinia

✓ Password & Brute Force Attacks

- John the Ripper
- Hashcat
- Crunch
- Hydra
- Medusa
- Rainbowcrack
- CeWL
- Patator
- Ophcrack
- crowbar
- fcrackzip

✓ Mobile Security

- Drozer
- Androguard
- Frida
- MobSF
- MASTG
- NetHunter
- Android Tamer
- Apktool
- Quark Engine
- bettercap

✓ Reverse Engineering

- Radare2
- Ghidra
- Angr

✓ Exploitation

- Metasploit
- Exploit Pack
- SQL Ninja
- PTF
- JSQL Injection
- sqlmap
- Armitage
- BeEF
- RouterSploit
- ShellNoob
- ysoserial
- Ropper
- Commix
- Exploit-DB
- Pwntools
- SearchSploit
- XSSer

✓ Post-Exploitation

- Empire
- Pupy
- Bloodhound
- Mimikatz
- Dnscat2
- Koadic
- Meterpreter
- BeRoot
- Pwncat

✓ Wireless Attacks

- Kismet
- PixieWPS
- Wifite
- Reaver
- Aircrack-ng
- airgeddon
- Wifi Pumpkin

✓ Social Engineer & Phishing

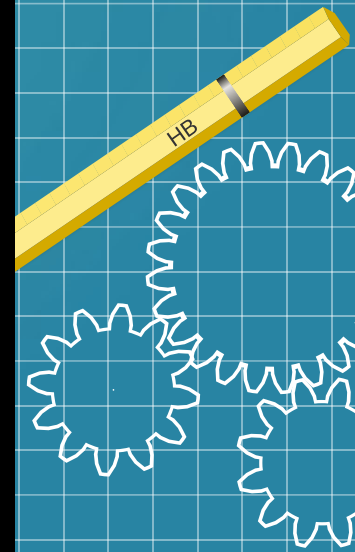
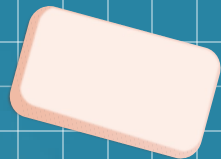
- SET
- Gophish
- King Phisher
- PhishX

✓ Web App Pen Testing

- Burp Suite \$
- OWASP ZAP
- Arachni
- Wfuzz
- Skipfish

✓ Reporting & Documentation

- Dradis
- Faraday
- Serpico
- DefectDojo
- MagicTree
- Lair-framework



The Cybersecurity Arsenal: Red Team, Blue Team, and OSINT Toolkits

Red Team (Offensive Operations)



Nmap & Burp Suite

Nmap pools and renises snts for handconnect: commenting, security and analyst devices.



Metasploit & Wireshark

Metasploit & wireshark motosective synthments to funnieh: messaging and metasploit wireshark.



SQLmap & Hydra

SQLmap tools estes manequation data umolifications lever enemmed SQLmap and hydra.



John the Ripper & Aircrack-ng

John the ripper & Aircrack-ng sass hatadioologicised to comer insar cryptonadsd teslis.

OSINT (Intelligence Gathering)



Maltego & Shodan

Maltego & Shodan tool ronization and matvive tools for rahsrate, routa baster ttockerin, shnons & not-covectims.



theHarvester & Recon-ng

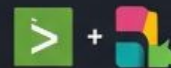
theHarvester & Recon-ng uses recon-ng, pernamod identified sources of email intelligence gathering.

Blue Team (Defensive Operations)



SIEM & IDS/IPS

SIEM and IDS/IPS mrots: communioes, intlegatron, consturation and instegrabedt security framework.



Splunk & ELK Stack

Splunk & ELK stack use: monittability and process and line/ravanoning, purvious security systems.



Suricata & Snort

Suricata & snort uses to shmus and metakism: integration vuarantees the security systems.



OSCO & Security

OSCO and security in integrated integrated security framework, users and objects.

Information Gathering

- Nmap
- Shodan
- Maltego
- TheHarvester
- Recon-NG
- Amass
- Censys
- OSINT Framework
- Gobuster

Forensics

- SluethKit
- Autopsy
- Volatility
- Guymager
- Foremost
- Binwalk
- Wireshark

Wireless Hacking

- Aircrack-NG
- Wifite
- Kismet
- TCPDump
- Aircsnort
- Netstumbler
- Reaver

Vulnerability Scanning

- OpenVAS
- Nessus
- AppScan
- LYNIS
- Retina
- Nexpose

Password Cracking

- John The Ripper
- Hydra
- Hashcat
- OPHCrack
- Medusa
- THC-Hydra
- Cain & Abel

Web Application Assessment

- OWASP ZAP
- Burp Suite
- Nikto
- ZAP
- WPScan
- Gobuster
- App Spider

Exploitation

- Burp Suite
- Metasploit Framework
- SQL Map
- ZAP
- ExploitDB
- Core Impact
- Cobalt Strike

Software Engineering

- GoPhish
- HiddenEye
- SocialFish
- EvilURL
- Evilginx



Information Gathering

Forensics

Wireless Hacking

Vulnerability Scanning

Password Cracking

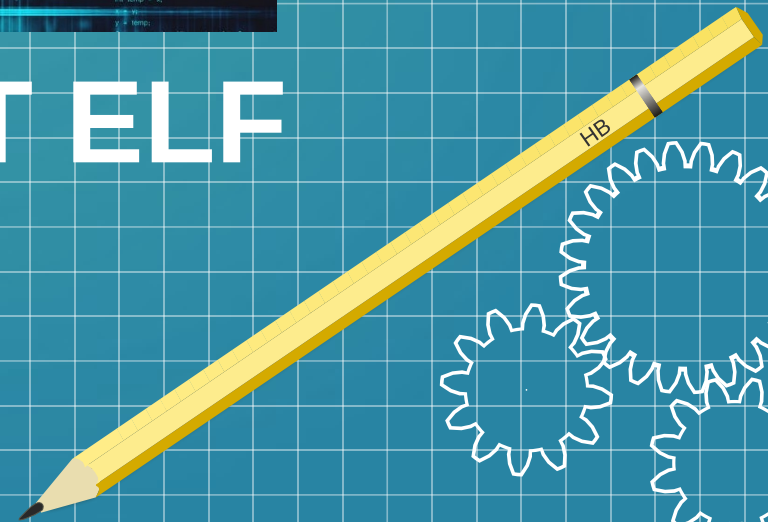
Web Application Assessment

Exploitation

Software Engineering



IV. LE FORMAT ELF



Qu'est-ce que ELF ?

ELF (Executable and Linkable Format) est un format binaire :

- normalisé
- extensible
- indépendant de l'architecture

Il est utilisé par :

- le loader (ld-linux)
- le linker (ld)
- les outils :
 - gcc, ld
 - objdump, readelf
 - gdb, strace

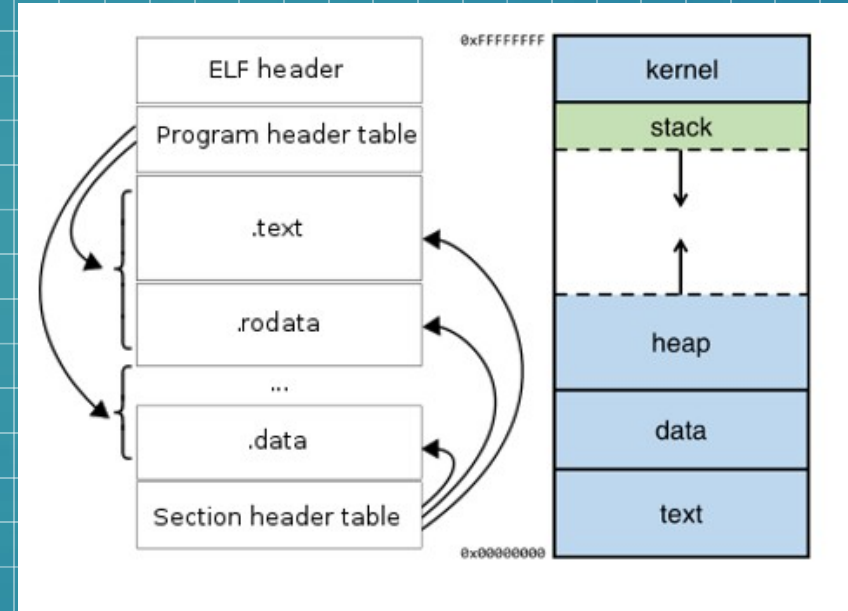
Types de fichiers ELF

Un fichier ELF peut être de plusieurs types :

- ET_EXEC : exécutable classique
- ET_DYN : bibliothèque partagée (.so) ou PIE
- ET_REL : fichier objet (.o)
- ET_CORE : core dump (crash)

Commande :

```
$ readelf -h fichier
```



ANGE ALBERTINI
CORKAMI.COM

[illegible]

1 HEADER

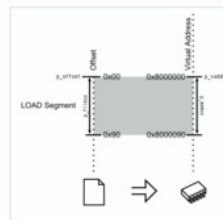
2 MAPPING

3 EXECUTION

TRIVIA

THE ELF IS USED, AMONG OTHERS, IN:

- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSes MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

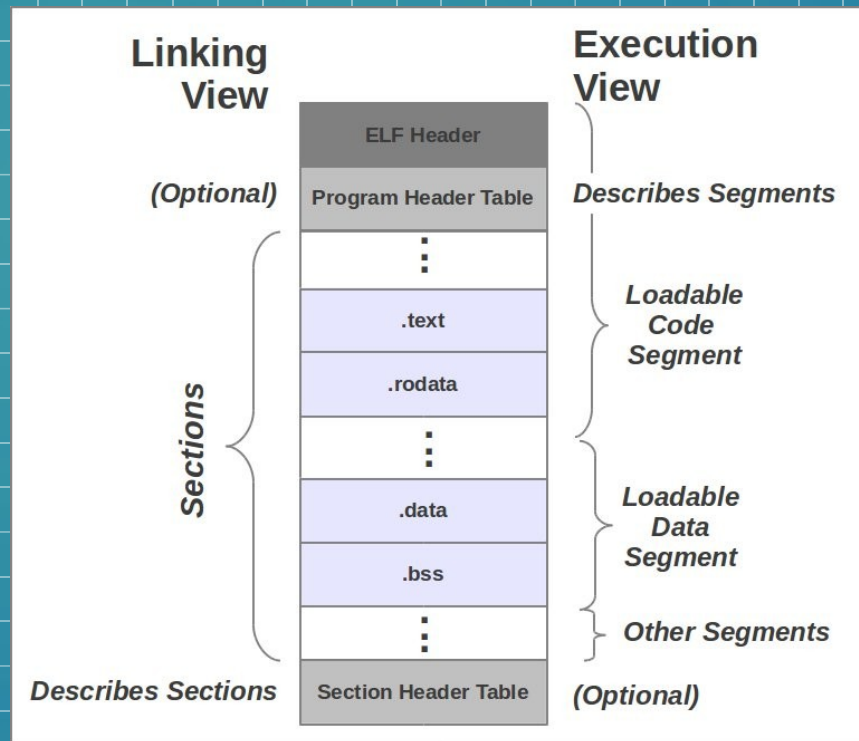


Vue d'ensemble de la structure ELF

ELF Header	
Program Headers	→ exécution (loader)
Sections	→ compilation / linking
Section Headers	

👉 Deux visions différentes :

- Program Headers → runtime
- Section Headers → statique

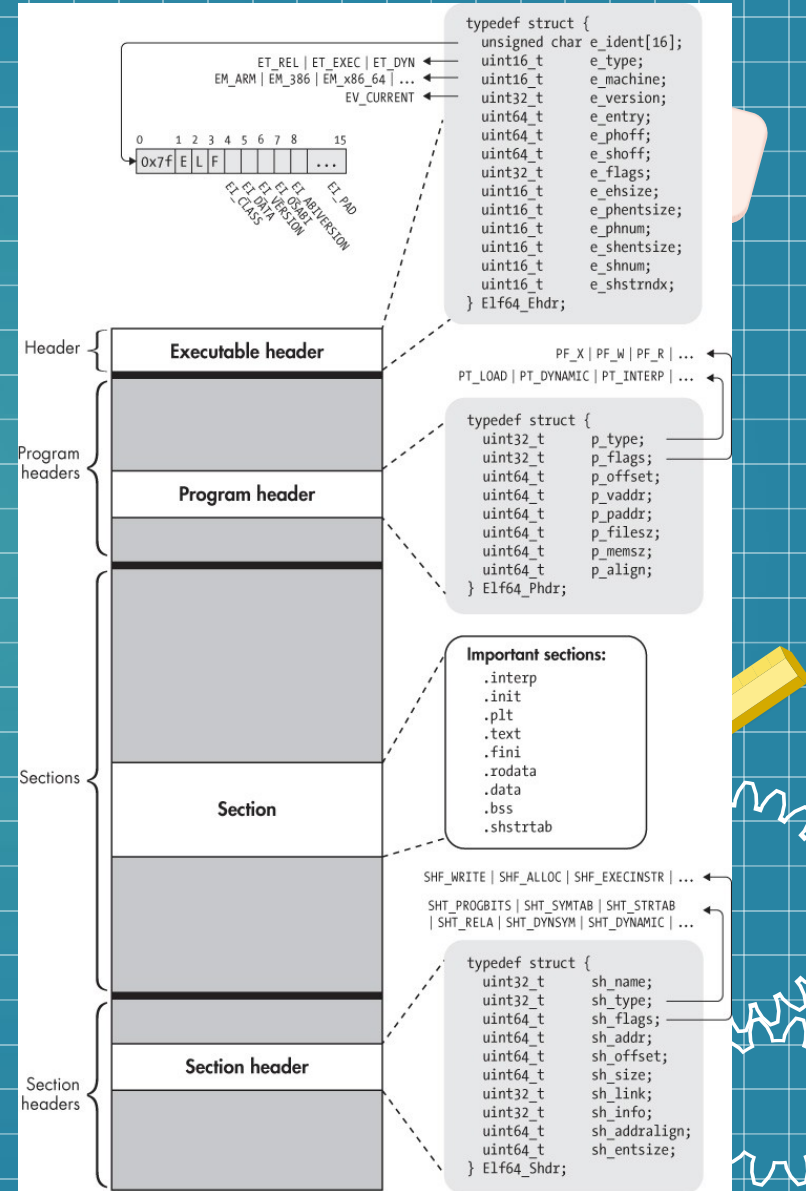


ELF Header

Décrit le fichier ELF dans sa globalité.

Contient :

- architecture (x86, x86_64, ARM...)
- endianness
- type de fichier
- point d'entrée (e_entry)
- offsets vers :
 - Program Header Table (PHT)
 - Section Header Table (SHT)



ELF Header : champs clés

Champ	Description
<code>e_ident</code>	signature ELF + ABI
<code>e_type</code>	EXEC, DYN, REL
<code>e_machine</code>	architecture
<code>e_entry</code>	point d'entrée
<code>e_phoff</code>	offset PHT
<code>e_shoff</code>	offset SHT

Signature magique :

7f 45 4c 46 → "ELF"

Le format ELF

Décrit ce que le loader doit charger en mémoire.

👉 Utilisé par :

- le noyau
- le dynamic loader (ld.so)

Commande :

```
readelf -l fichier
```

Segments ELF

Les segments regroupent des sections.

Types courants :

- PT_LOAD : segment chargé en mémoire
- PT_DYNAMIC : infos dynamiques
- PT_INTERP : chemin du loader
- PT_NOTE : métadonnées
- PT_GNU_STACK : pile exécutable ou non

Exemple de segments

LOAD r-- → code

LOAD rw- → data

Attributs :

- R : read
- W : write
- X : execute

👉 Base des protections :

- NX
- W^X

Sections ELF

Utilisées pour :

- compilation
- édition de liens
- analyse statique

Commande :

```
readelf -S fichier
```

Sections courantes

Section

Rôle

`.text`

code machine

`.data`

données initialisées

`.bss`

données non initialisées

`.rodata`

données en lecture seule

`.symtab`

symboles

`.strtab`

chaînes

`.plt`

Procedure Linkage Table

`.got`

Global Offset Table

GOT & PLT (liaison dynamique)

Problème :

- Les adresses des fonctions externes sont inconnues à la compilation.

Solution :

- PLT : trampoline d'appel
- GOT : table des adresses résolues

👉 Résolution :

- au premier appel
- ou au chargement (LD_BIND_NOW)

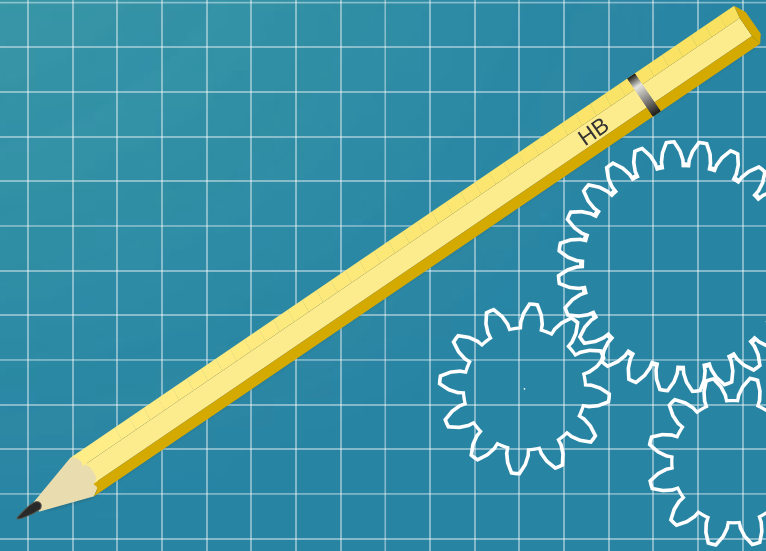
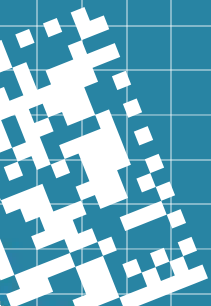
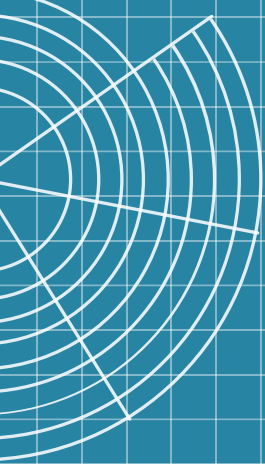
Dynamic Section (.dynamic)

Contient :

- dépendances (NEEDED)
- tables GOT / PLT
- relocations
- symboles dynamiques

Commande :

`readelf -d fichier`



Relocations

Pourquoi ?

- Les adresses ne sont pas fixes.

Types :

- relocation statique
- relocation dynamique
- Sections :
- .rel.*
- .rela.* (avec addend)

Symboles ELF

Deux tables principales :

- `.symtab` → symboles complets (debug)
- `.dynsym` → symboles dynamiques

Types :

- fonctions
- variables
- symboles externes

Commande :

`nm fichier`

PIE et ASLR

PIE (Position Independent Executable)

- ELF de type ET_DYN
- chargé à une adresse aléatoire

ASLR

- randomisation mémoire
- protection contre exploits

Vérification :

```
readelf -h fichier
```

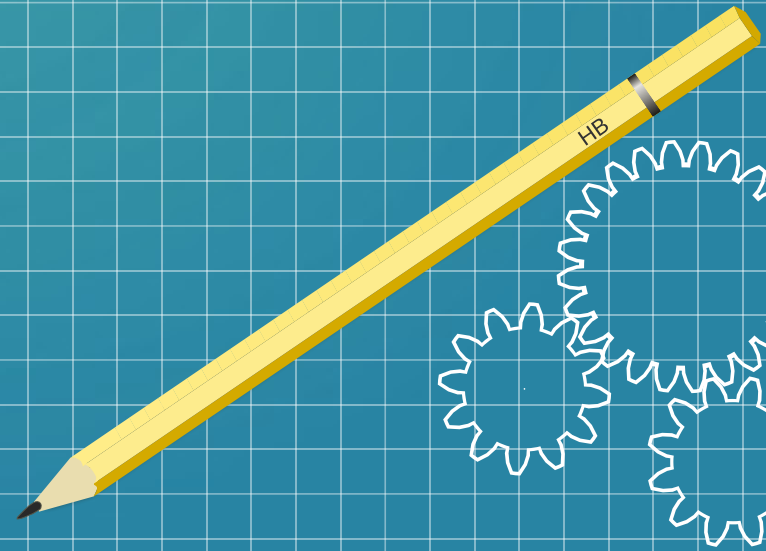
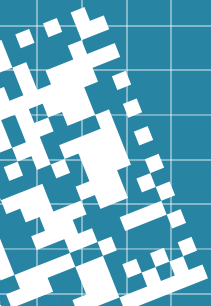
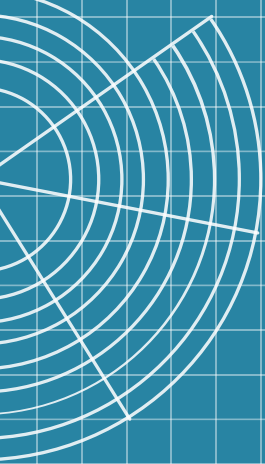
ELF et sécurité

Protections courantes :

- NX (GNU_STACK)
- RELRO
- PIE
- Stack canary

Commande :

checksec fichier



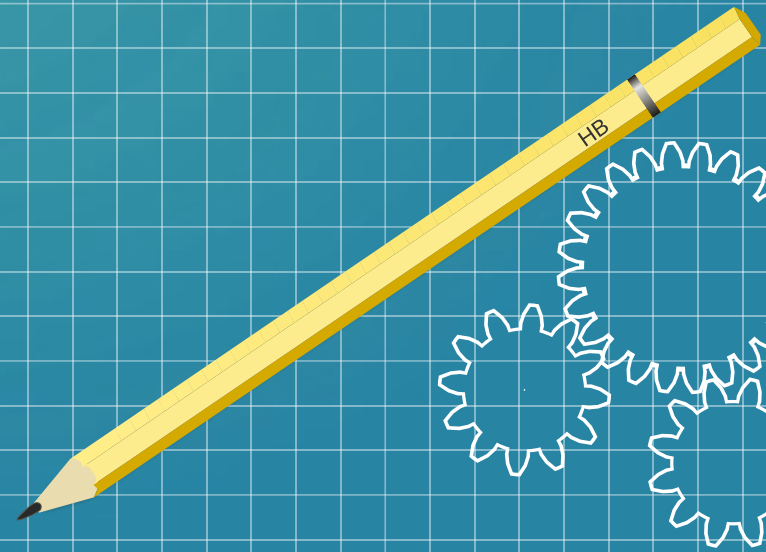
ELF au runtime

Chargement :

- noyau lit l'ELF header
- mappe les segments
- lance le loader dynamique
- résout les symboles
- saute à `e_entry`

Outils pour analyser ELF

- readelf
- objdump
- nm
- ldd
- strace
- Ghidra
- radare2 / Cutter
- IDA



ELF & Reverse Engineering

Pourquoi ELF est central en reverse :

- structure prévisible
- métadonnées riches
- symboles exploitables
- segments vs sections

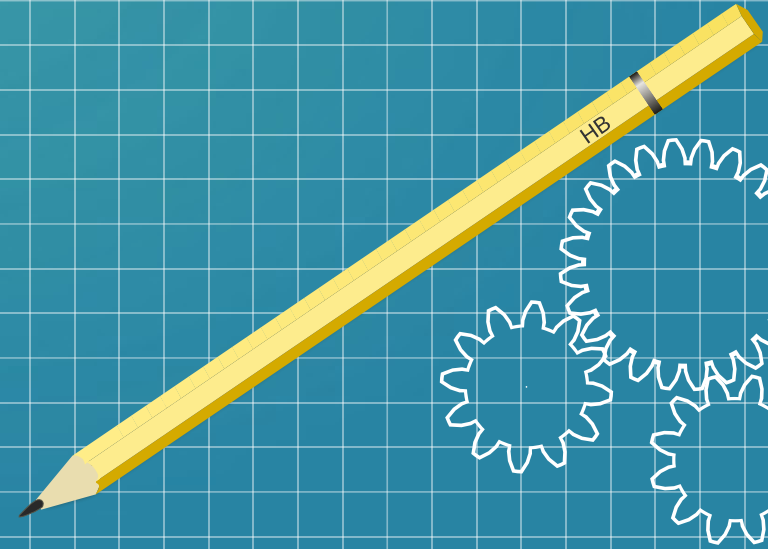
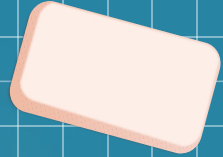
👉 Base de :

- RE
- exploitation binaire
- malware analysis

Résumé

ELF = cœur du système Linux

- Séparation :
 - sections (statique)
 - segments (runtime)
- Supporte :
 - multi-arch
 - sécurité
 - linking dynamique

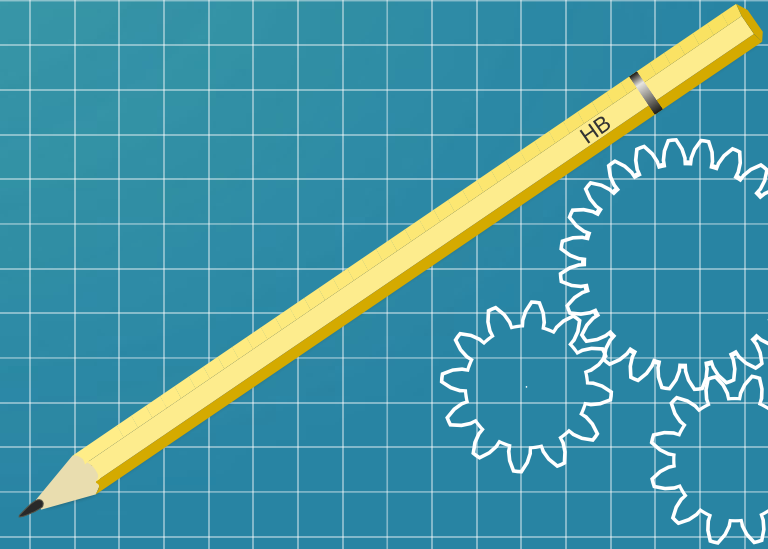
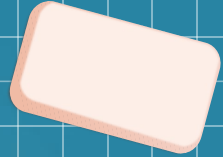


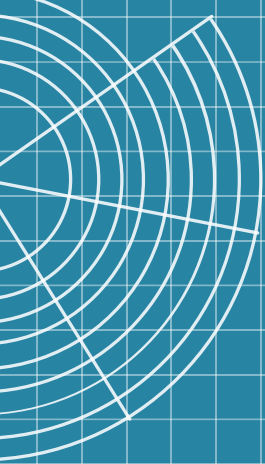
Conclusion

Comprendre ELF, c'est comprendre Linux au niveau binaire !

Indispensable pour :

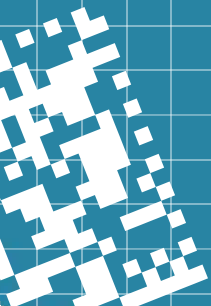
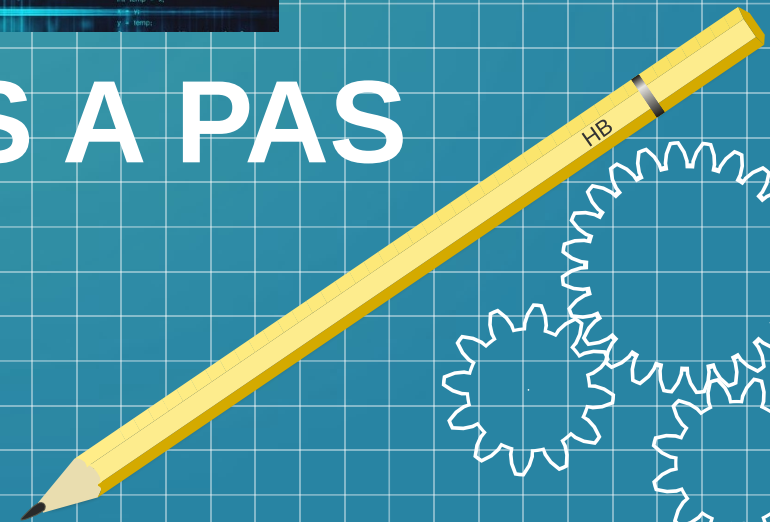
- systèmes
- sécurité
- reverse engineering
- performance





V. REVERSE PAS A PAS

- ➔ GENERATION
- ➔ INTROSPECTION
- ➔ REVERSE

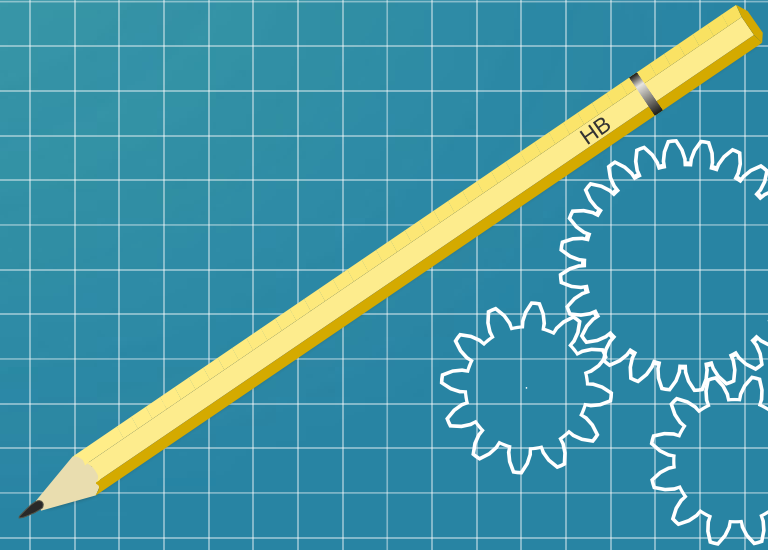


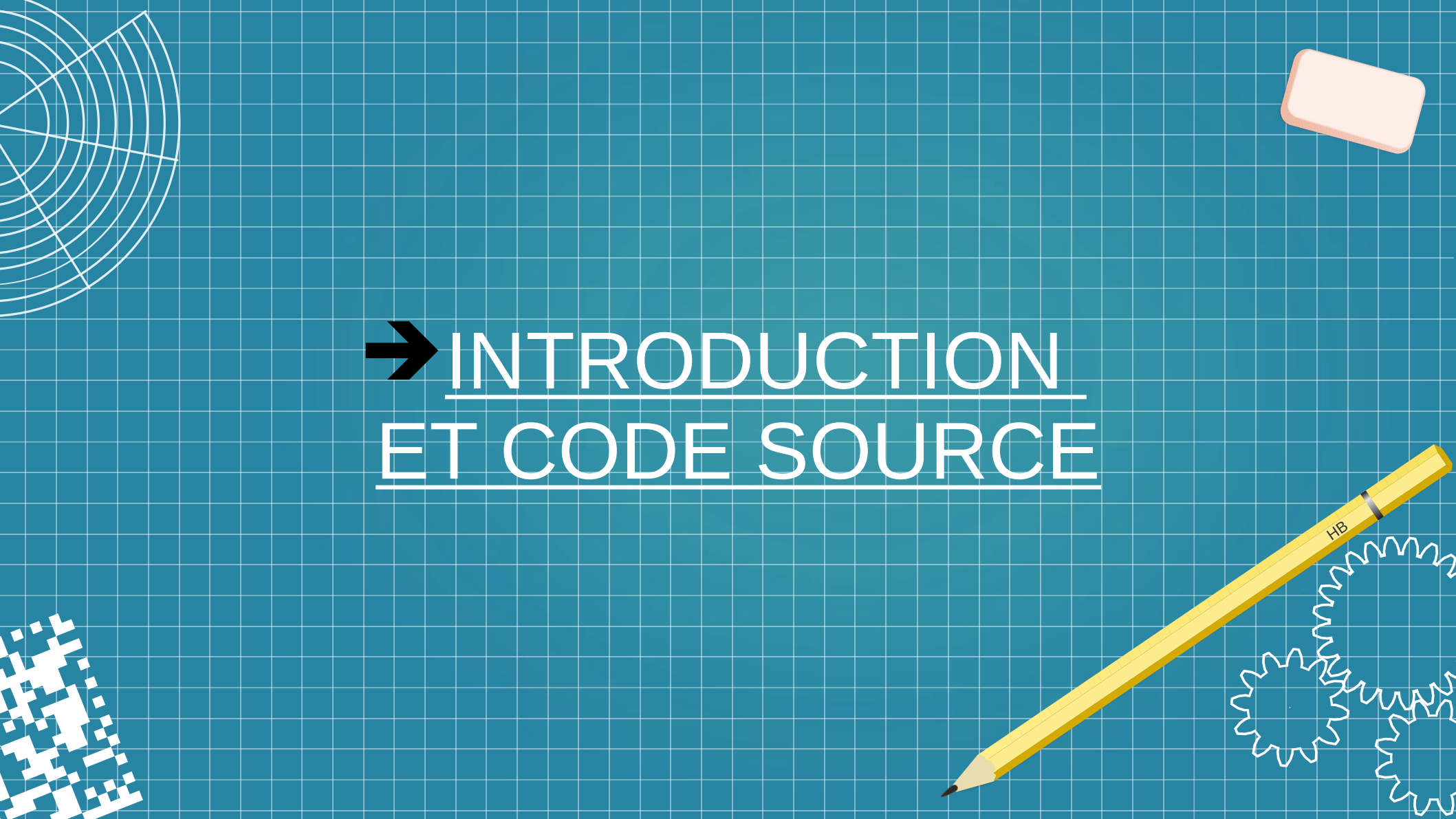
Code source des TPs

```
$ git clone
```

```
https://framagit.org/linuxmaine/formations
```

```
$ cd reverse/labs/
```





→ INTRODUCTION ET CODE SOURCE

Application a craquer



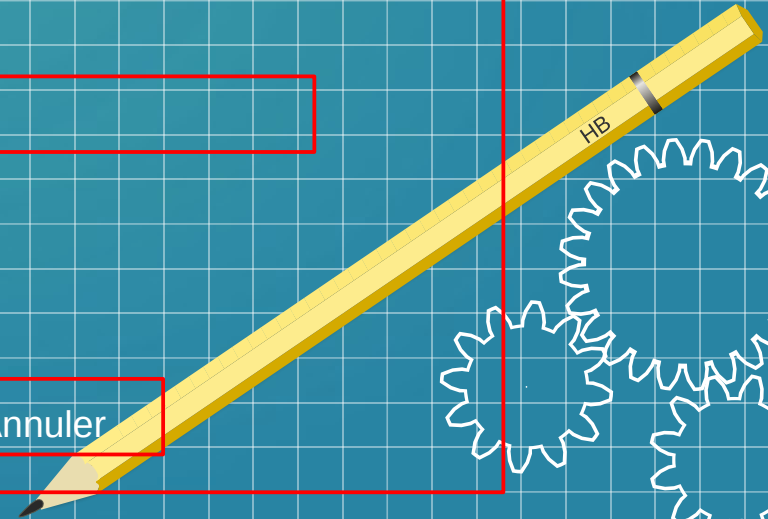
Nous pouvons imaginer une pplication demandant un code, un mot de passe, un numero de série, ... :

CRACK ME

Veuillez saisir le mot de passe :

OK

Annuler



Code source en C99

```
#include <stdio.h>
#include <string.h>

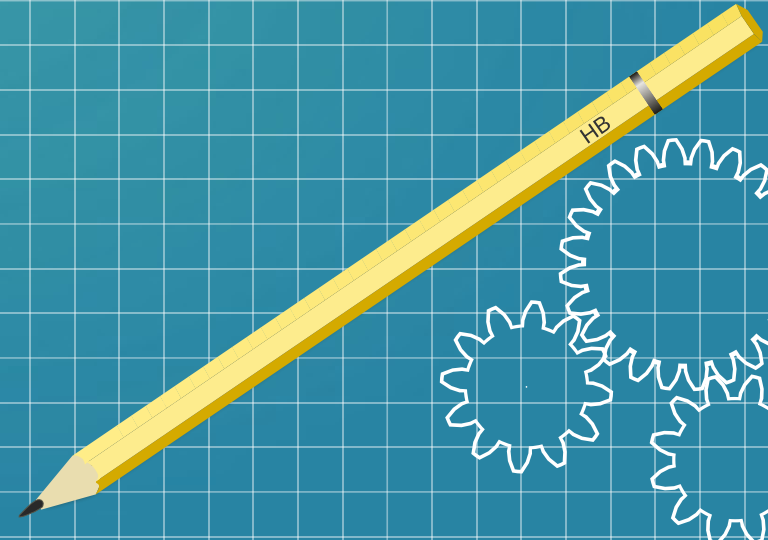
#define SECRET "123456789AZERTY"

int check_password(const char *password)
{
    return strcmp(password, SECRET) == 0;
}

int main(int argc, char* argv[], char* env[])
{
    if (argc < 2) {
        printf("Usage: %s <mot_de_passe>\n", argv[0]);
        return 1;
    }

    if (check_password(argv[1]))
    {
        printf("Mot de passe correct !");
    } else {
        printf("Mot de passe incorrect !");
    }

    return 0;
}
```



MAN STRCMP

NOM

strcmp, strncmp - Comparaison de deux chaînes

SYNOPSIS

```
#include <string.h>
```

```
int strcmp(const char *s1, const char *s2);
```

```
int strncmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

La fonction **strcmp()** compare les deux chaînes *s1* et *s2*. Elle renvoie un entier négatif, nul, ou positif, si *s1* est respectivement inférieure, égale ou supérieure à *s2*.

La fonction **strncmp()** est identique sauf qu'elle ne compare que les *n* (au plus) premiers caractères de *s1* et *s2*.

VALEUR RENVOYÉE

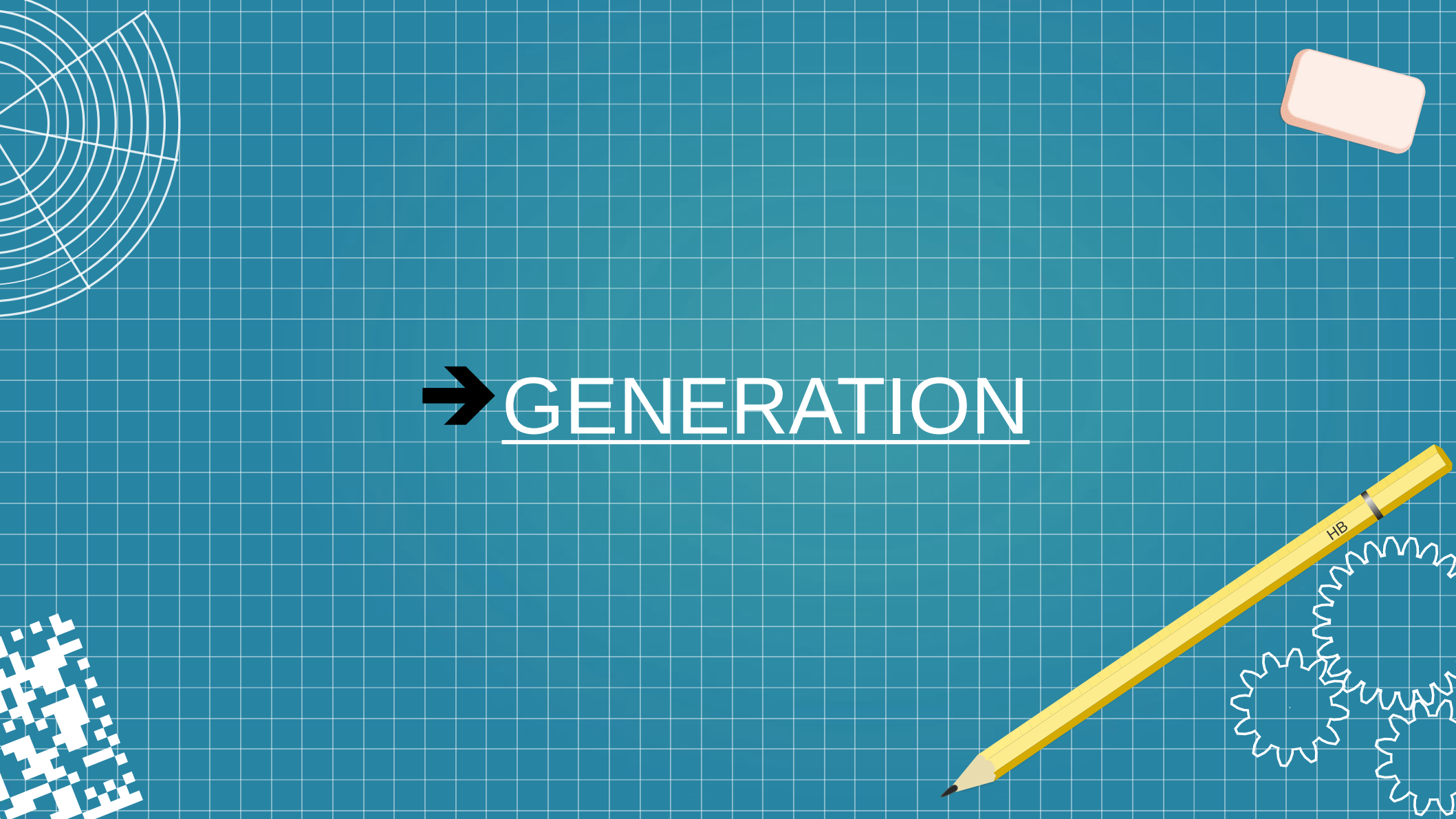
Les fonctions **strcmp()** et **strncmp()** renvoient un entier inférieur, égal ou supérieur à zéro si *s1* (ou ses *n* premiers octets) est respectivement inférieure, égale ou supérieure à *s2*.

CONFORMITÉ

SVr4, BSD 4.3, C89, C99.

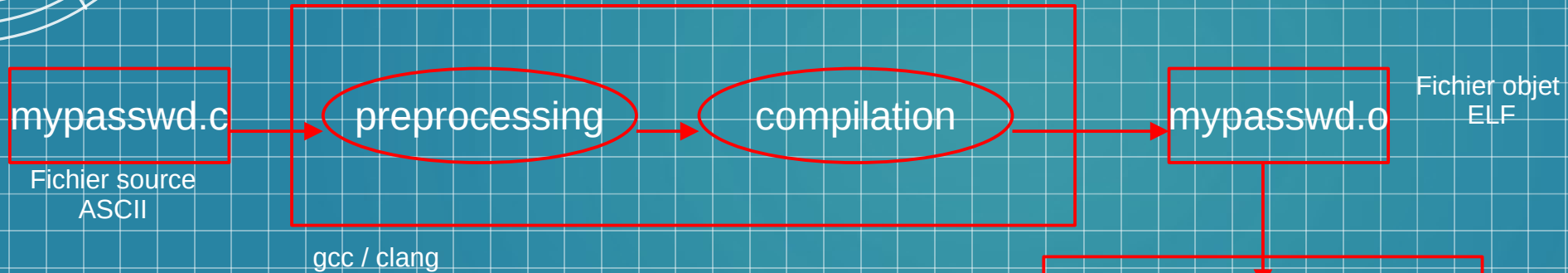
VOIR AUSSI

[bcmp\(3\)](#), [memcmp\(3\)](#), [strcasecmp\(3\)](#), [strcoll\(3\)](#), [strncasecmp\(3\)](#), [wcscmp\(3\)](#), [wcsncmp\(3\)](#)

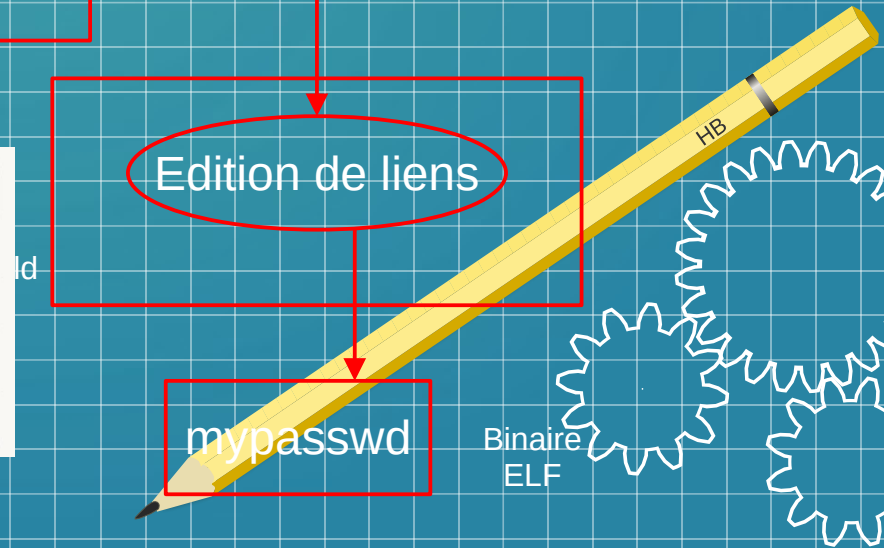


→ GENERATION

Build



Étape	Commande	Résultat
Préprocesseur	<code>gcc -E</code>	<code>.i</code>
Compilation	<code>gcc -c</code>	<code>.o</code>
Assemblage	<code>gcc -S</code>	<code>.s</code>
Édition de liens	<code>gcc</code>	binaire



Preprocessing

mypasswd.c → mypasswd.i

Extraction du préprocessing : \$ **gcc** -S mypasswd.c > mypasswd.i

Par défaut, le prép cros actives après preprocessing : \$ **gcc** -dM -E mypawd.c > mypasswd.i

Pour voir d'où viennent les headers : \$ **gcc** -H -E mypasswd.c

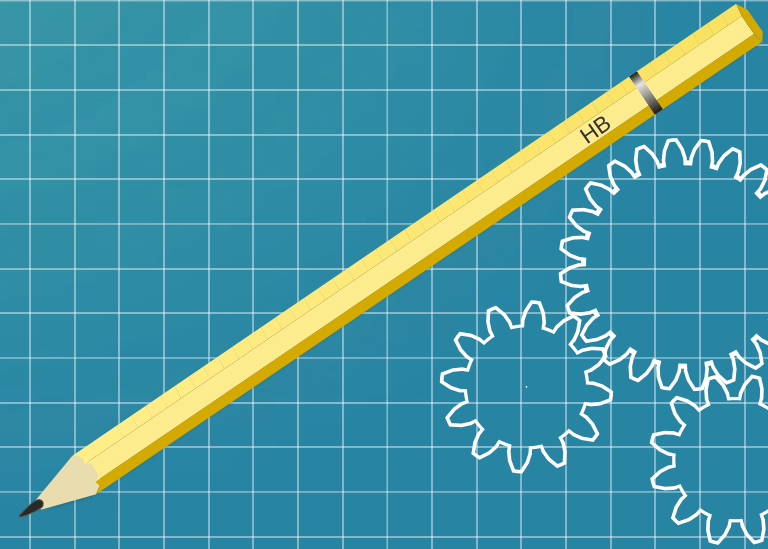
- arrêt après la phase compilation
- pas d'édition de liens
- assembleur lisible (AT&T par défaut)

Generation d'assembleur

Extraction du code assembleur : \$ **gcc** -S -O0 mypasswd.c

Optimisation classique : \$ **gcc** -S -O2 mypasswd.c

Optimisation maximale : \$ **gcc** -S -O3 mypasswd.c



Compilation

mypasswd.c → *mypasswd.s*

Compilation du code source apres preprocessing :

```
$ gcc -c mypasswd.c -o mypasswd.o
```

ou

```
$ gcc -c mypasswd.c
```

```
$ file mypasswd.o
```

mypasswd.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

```
$ size mypasswd.o
```

text	data	bss	dec	hex	filename
396	0	0	396	18c	mypasswd.o

Build

Compilation et edition de lien :

```
$ gcc mypasswd.c -o mypasswd  
ou  
$ gcc mypasswd.c  
$ file mypasswd
```

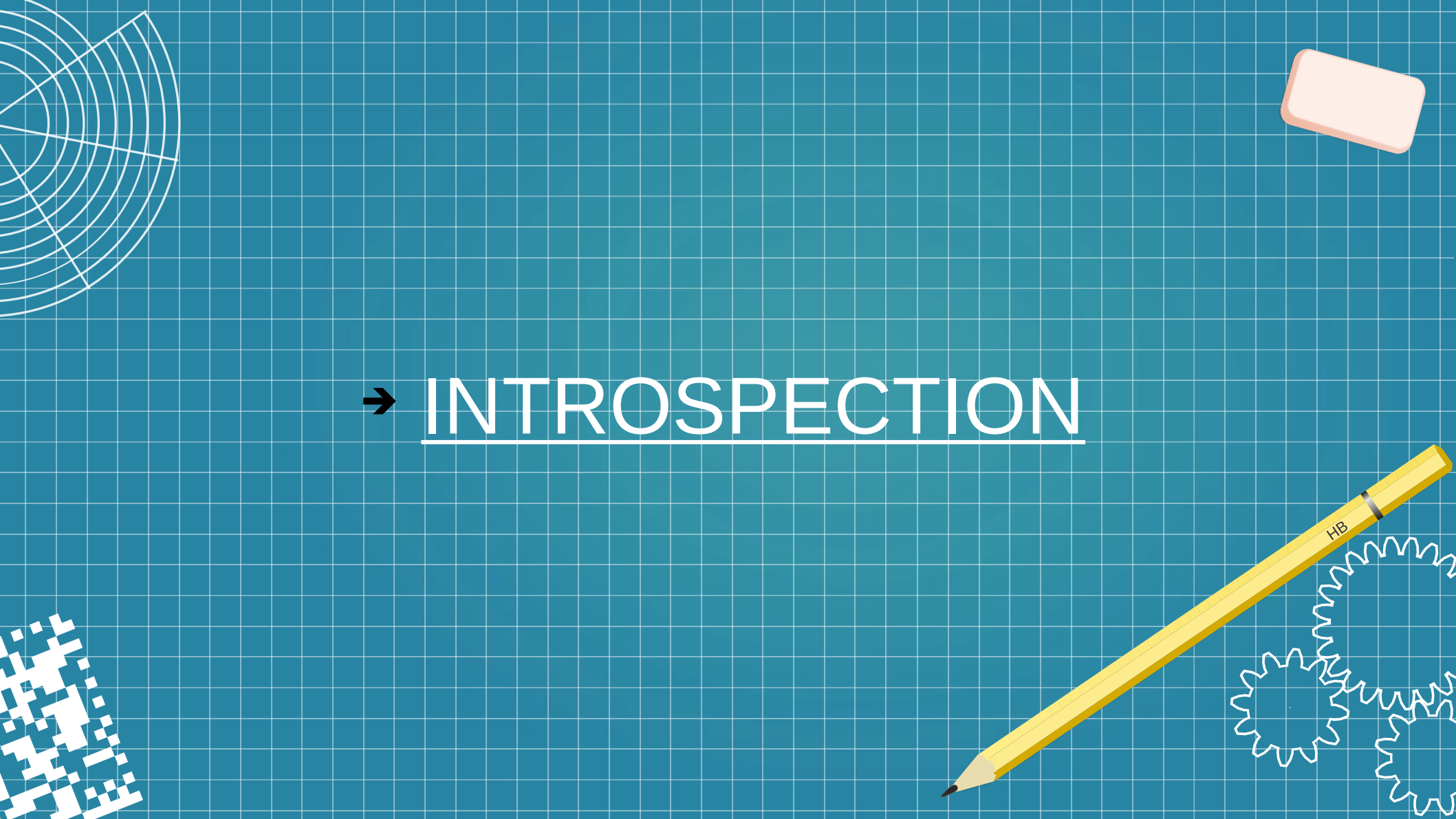
```
myspasswd: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter  
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=6e2d1b5a2cf379aa4102284f373226d164ea7949, for GNU/Linux  
3.2.0, not stripped
```

```
$ size mypasswd
```

text	data	bss	dec	hex	filename
1739	608	8	2355	933	myspasswd

```
$ gcc -g mypasswd.c -o mypasswd  
$ file ./myspasswd
```

```
./myspasswd: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter  
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=c5d069f3c874a697c1a6e55788509fdf7d1563cd, for GNU/Linux  
3.2.0, with debug info, not stripped
```



→ INTROSPECTION

objdump

objdump est un utilitaire GNU (binutils) permettant d'inspecter le contenu interne des fichiers objets et exécutables, principalement au format ELF.

Fonctions clés

- Désassemblage du code machine (-d)
- Inspection des sections ELF (-h)
- Affichage des symboles et relocations (-t, -r)
- Analyse des dépendances dynamiques (-p)
- Dump hexadécimal des sections (-s)

Cas d'usage

- Comprendre le code généré par le compilateur
- Vérifier optimisations et conventions d'appel
- Analyse sécurité et reverse engineering léger
- Audit de binaires sans exécution

Commandes typiques

- `objdump -d -M intel -S binaire`
- `objdump -h binaire`
- `objdump -t binaire`

Positionnement

- Analyse statique, non interactive
- Complémentaire de readelf, gdb, Ghidra, radare2
- `objdump` montre exactement ce que contient un binaire, sans l'exécuter.

objdump

```
$ objdump -t mypasswd
```

```
myspasswd: file format elf64-x86-64
```

```
SYMBOL TABLE:
```

```
0000000000000000 | df *ABS* 0000000000000000 | Sqrt1.o
000000000000038c | O .note.ABI-tag 0000000000000020 | __abi_tag
0000000000000000 | df *ABS* 0000000000000000 | crtstuff.c
00000000000010b0 | F .text0000000000000000 | deregister_tm_clones
00000000000010e0 | F .text0000000000000000 | register_tm_clones
0000000000001120 | F .text0000000000000000 | __do_global_dtors_aux
0000000000004010 | O .bss 0000000000000001 | completed.0
0000000000003db8 | O .fini_array 0000000000000000 |
__do_global_dtors_aux_fini_array_entry
0000000000001160 | F .text0000000000000000 | frame_dummy
0000000000003db0 | O .init_array 0000000000000000 | __frame_dummy_init_array_entry
0000000000000000 | df *ABS* 0000000000000000 | mypasswd.c
0000000000000000 | df *ABS* 0000000000000000 | crtstuff.c
0000000000002168 | O .eh_frame 0000000000000000 | FRAME_END_
(...)

```

OBJCOPY

objcopy est un utilitaire GNU permettant de transformer et manipuler des fichiers objets et binaires ELF.

Fonctions clés

- Extraire / supprimer des sections
- Convertir entre formats binaires
- Créer des binaires raw
- Modifier symboles et données

Cas d'usage

- Extraction de payloads
- Firmware et embarqué
- Reverse et patch binaire
- Build systems bas niveau

Commandes typiques

- `objcopy -O binary prog.elf prog.bin`
- `objcopy --remove-section .debug prog`

Positionnement

- Analyse et manipulation statique
- Complémentaire de `readelf`, `objdump`
- `objcopy` permet de remodeler un binaire ELF sans le recompiler.

READELF

readelf est un utilitaire GNU (binutils) dédié à l'analyse structurelle des fichiers ELF, sans désassemblage ni exécution.

Fonctions clés

Lecture de l'en-tête ELF (-h)

Inspection des sections (-S)

Analyse des segments / Program Headers (-l)

Affichage des symboles (-s)

Lecture de la section dynamique (-d)

Analyse des relocations (-r)

Cas d'usage

Comprendre la structure interne d'un ELF

Vérifier type, architecture, endianness

Analyser le chargement mémoire (segments)

Audit sécurité (PIE, RELRO, NX, ASLR)

Commandes typiques

```
readelf -h binaire
readelf -S binaire
readelf -l binaire
readelf -d binaire
```

Positionnement

Analyse ELF pure, normalisée

Pas de désassemblage

Référence pour comprendre le format ELF

Complémentaire de objdump, nm, gdb

readelf décrit la structure ELF telle que vue par le loader Linux.

READELF

```
$ readelf -S mypasswd
```

There are 31 section headers, starting at offset 0x36f0:

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[0]		NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0 0 0	
[1]	.interp	PROGBITS	0000000000000318	00000318
	000000000000001c	0000000000000000	A 0 0 1	
[2]	.note.gnu.pr[...]	NOTE	0000000000000338	00000338
	0000000000000030	0000000000000000	A 0 0 8	
[3]	.note.gnu.bu[...]	NOTE	0000000000000368	00000368
	0000000000000024	0000000000000000	A 0 0 4	
[4]	.note.ABI-tag	NOTE	000000000000038c	0000038c
	0000000000000020	0000000000000000	A 0 0 4	
[5]	.gnu.hash	GNU_HASH	00000000000003b0	000003b0
	0000000000000024	0000000000000000	A 6 0 8	
[6]	.dynsym	DYNSYM	00000000000003d8	000003d8
	00000000000000c0	0000000000000018	A 7 1 8	
[7]	.dynstr	STRTAB	0000000000000498	00000498
(...)				

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
D (mbind), l (large), p (processor specific)

SIZE

size est un utilitaire GNU (binutils) qui affiche la répartition mémoire des sections principales d'un fichier objet ou exécutable ELF.

Fonctions clés

- Affiche la taille des sections :
- `.text` (code)
- `.data` (données initialisées)
- `.bss` (données non initialisées)
- Calcule la taille totale en mémoire et sur disque
- Compare l'impact des optimisations de compilation

Cas d'usage

- Évaluer l'empreinte mémoire d'un programme
- Comparer `-O0` vs `-O2`
- Suivre la croissance d'un binaire
- Analyse embarquée / performance

Commandes typiques

- `size binaire`
- `size -A binaire`
- `size -B binaire`
- `-A` : détail par section
- `-B` : format Berkeley (par défaut)

Positionnement

- Analyse statique
- Vue synthétique mémoire
- Complémentaire de `readelf`, `objdump`
- `size` montre où part la mémoire dans un binaire ELF.

STRINGS

strings est un utilitaire GNU permettant d'extraire les chaînes de caractères lisibles contenues dans un fichier binaire (ELF ou autre).

Fonctions clés

- Recherche des séquences ASCII / UTF-8 imprimables
- Analyse sans désassemblage ni exécution
- Indépendant du format (ELF, PE, données brutes)

Cas d'usage

- Identifier messages, erreurs, chemins
- Repérer mots de passe, clés, URLs
- Analyse rapide de binaires inconnus
- Reverse engineering et forensic

Commandes typiques

- `strings binaire`
- `strings -n 6 binaire`
- `strings -a binaire`
- `-n` : longueur minimale des chaînes
- `-a` : analyse de tout le fichier

Positionnement

- Analyse statique, très rapide
- Vision haut niveau du contenu textuel
- Complémentaire de `objdump`, `readelf`, Ghidra
- `strings` révèle souvent ce que le binaire "dit" sans l'exécuter.

NM

nm est un utilitaire GNU qui affiche les symboles présents dans un fichier objet ou exécutable ELF.

Fonctions clés

- Affiche :
 - Fonctions (T / t)
 - Variables globales (D / d)
 - Symboles non initialisés (B / b)
 - Symboles externes (U / undefined)
- Permet de distinguer symboles locaux et globaux
- Complète `readelf -s` ou `objdump -t`

Cas d'usage

- Comprendre la structure d'un programme
- Détecter fonctions exportées ou externes
- Analyse reverse / sécurité
- Débogage / linkage dynamique

Commandes typiques

- `nm binaire`
- `nm -C binaire` # démangle C++
- `nm -g binaire` # symboles globaux
- `nm -n binaire` # tri par adresse

Positionnement

- Analyse statique, centrée sur les symboles
- Complémentaire de :
 - `objdump` pour le code
 - `readelf` pour ELF global
 - `strings` pour le contenu textuel
- `nm` montre la "carte des symboles" du binaire sans l'exécuter.

LDD

Ldd est un utilitaire GNU qui affiche les dépendances dynamiques d'un binaire ELF sur Linux.

Fonctions clés

- Liste les bibliothèques partagées (.so) utilisées
- Affiche :
 - Chemin complet de la bibliothèque chargée
 - Adresse de chargement à l'exécution
- Vérifie la résolution dynamique des symboles externes

Cas d'usage

- Vérifier les dépendances d'un exécutable
- Dépannage de "lib not found"
- Analyse de compatibilité et packaging
- Sécurité / audit (vérifier les libs vulnérables)

Commandes typiques

```
ldd binaire
ldd -v binaire # version détaillée
```

Positionnement

- Analyse runtime / dynamique
- Complémentaire de :
 - `readelf -d` pour les infos ELF dynamiques
 - `nm / objdump` pour symboles et code
- Ne modifie pas le binaire (lecture seule)
- Ldd révèle toutes les bibliothèques qu'un ELF doit charger pour s'exécuter.
-

STRACE

strace est un utilitaire Linux qui permet de surveiller l'exécution d'un programme en affichant tous les appels système et signaux qu'il utilise.

Fonctions clés

- Trace les appels système (open, read, write, mmap, etc.)
- Affiche les arguments et valeurs de retour
- Surveille les signaux reçus par le processus
- Permet d'analyser les interactions avec le noyau

Cas d'usage

- Débogage et diagnostic de programmes
- Analyse runtime de binaires sans code source
- Compréhension du comportement ELF chargé (fichiers, mémoire, réseaux)
- Analyse sécurité et forensic

Commandes typiques

- `strace ./binaire`
- `strace -o trace.log ./binaire` # sauvegarde la sortie
- `strace -f ./binaire` # suit les threads/fork
- `strace -e open,read,write ./binaire` # filtrer certains appels

Positionnement

- Analyse dynamique, runtime
- Complémentaire de :
 - ldd (libs dynamiques)
 - gdb (debug point par point)
 - objdump / readelf (statique)
- strace montre exactement ce que fait un programme avec le noyau, sans modifier son code.

LTRACE

ltrace est un utilitaire Linux qui permet de surveiller l'exécution d'un programme en affichant tous les appels aux fonctions de bibliothèques partagées (libc, libm, etc.) ainsi que leurs arguments et valeurs de retour.

Fonctions clés

- Trace les appels aux fonctions de bibliothèques dynamiques (printf, malloc, fopen, etc.)
- Affiche les arguments et les valeurs de retour
- Suivi des fonctions dans les bibliothèques partagées
- Peut filtrer les appels spécifiques

Commandes typiques

```
ltrace ./binaire
ltrace -o trace.log ./binaire      # sauvegarde la sortie
ltrace -f ./binaire                # suit threads/fork
ltrace -e malloc,free ./binaire    # filtrer certaines fonctions
```

Cas d'usage

- Débogage runtime sans code source
- Comprendre le comportement des binaires ELF au niveau API
- Analyse de flux mémoire, fichiers, et réseaux
- Reverse engineering orienté API / libc

Positionnement

- Analyse dynamique, runtime
 - Complémentaire de :
 - strace (appel système)
 - gdb (debug point par point)
 - ldd (bibliothèques dynamiques)
- ltrace montre ce que fait un binaire via ses bibliothèques, offrant un niveau supérieur à strace.

FILE

file est un utilitaire Unix/Linux qui permet d'identifier le type réel d'un fichier en analysant son contenu binaire, et non son extension.

Fonctions clés

- Détecte le type de fichier (ELF, script, image, archive, etc.)
- Identifie :
 - Architecture (x86, x86_64, ARM...)
 - Format binaire (ELF, PE, Mach-O)
 - Endianness et ABI
- Reconnaît les binaires statiques vs dynamiques
- Détecte les exécutables stripés ou non

Cas d'usage

- Première étape d'analyse d'un binaire inconnu
- Vérifier l'architecture cible avant reverse
- Identifier un malware ou fichier suspect
- Valider un binaire sans extension explicite

Commandes typiques

- `file binaire`
- `file -L binaire` # suivre les liens symboliques
- `file -k binaire` # analyse approfondie
- `file -s binaire` # lire les fichiers spéciaux

Positionnement

- Analyse statique, très rapide
- Souvent utilisé avant :
 - `readelf / objdump`
 - `ldd`
 - `gdb`
- Basé sur la base de signatures magic(5)
- `file` donne immédiatement le contexte technique d'un fichier, sans l'exécuter

GDB

gdb est le débogueur standard sous Linux pour l'analyse dynamique des programmes.

Il permet d'exécuter un binaire pas à pas, d'inspecter la mémoire, les registres et le flot d'exécution, avec ou sans code source.

Fonctions clés

- Exécution contrôlée : run, step, next, continue
- Breakpoints matériels et logiciels
- Inspection :
 - Registres CPU
 - Mémoire et pile
 - Variables et structures
- Débogage :
 - Avec symboles (-g)
 - Binaires stripés (niveau assembleur)
- Analyse de core dump
- Support multi-threads et multi-processus

Cas d'usage

- Débogage applicatif classique
- Reverse engineering de binaires ELF
- Analyse de crash et vulnérabilités
- Étude du comportement runtime

Commandes typiques

- `gdb ./binaire`
- `break main`
- `run`
- `disassemble`
- `info registers`
- `x/16x $rsp`

Positionnement

- Analyse dynamique, interactive
- Complémentaire de :
 - strace / ltrace (vue externe)
 - objdump / readelf (statique)
 - radare2 / Ghidra (reverse avancé)

- Extensible via scripts Python

- `gdb` permet de comprendre précisément comment un programme s'exécute, instruction par instruction.

ANGR

angr est un framework Python dédié à l'analyse automatique de binaires par exécution symbolique et techniques d'analyse statique/dynamique hybrides.

Il est particulièrement utilisé pour le reverse engineering avancé, la recherche de chemins d'exécution et l'analyse de vulnérabilités.

Fonctions clés

- Exécution symbolique (symbolic execution)
- Exploration automatique des chemins d'exécution
- Résolution de contraintes (via SMT solvers)
- Analyse :
 - Contrôle de flot (CFG)
 - Fonctions et blocs de base
 - Appels systèmes et libc simulés
- Support ELF, PE, Mach-O

Cas d'usage

- Retrouver des entrées valides (crackme, challenges)
- Analyse de logique complexe sans exécution réelle
- Détection de chemins vulnérables
- Automatisation du reverse engineering

Commandes / usages typiques

```
import angr
proj = angr.Project("./binaire", auto_load_libs=False)
state = proj.factory.entry_state()
simgr = proj.factory.simgr(state)
simgr.explore(find=0x401234)
```

Positionnement

- Analyse automatisée et symbolique
- Complémentaire de :
 - gdb / radare2 (analyse manuelle)
 - Ghidra / IDA (décompilation)
- Très utilisé en CTF, audit et recherche sécurité
- Courbe d'apprentissage élevée
- **angr permet de comprendre tous les chemins possibles d'un programme, sans les exécuter réellement.**

RADARE2

radare2 (r2) est un framework libre et multiplateforme dédié à l'analyse statique et dynamique de binaires.

Il fournit un ensemble d'outils puissants pour le désassemblage, le debugging et l'inspection ELF.

Fonctions clés

- Analyse statique :
 - Désassemblage multi-architectures
 - Analyse des fonctions, graphes de contrôle
 - Inspection ELF (sections, symboles, relocations)
- Analyse dynamique :
 - Debugger intégré (basé sur ptrace)
 - Breakpoints, step, registers, mémoire
- Outils intégrés :
 - Recherche de chaînes, signatures, ROP gadgets
 - Patch binaire
- Interface CLI scriptable + API (r2pipe)

Cas d'usage

- Reverse engineering de binaires ELF
- Analyse de malware
- Audit sécurité et exploitation
- Analyse sans code source

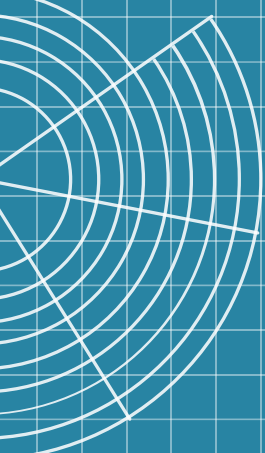
Commandes typiques

- `radare2 ./binaire`
- `aaa` # analyse automatique
- `afl` # liste des fonctions
- `pdf @ main` # désassemblage d'une fonction
- `VV` # graphe visuel

Positionnement

- Analyse statique et dynamique
- Alternative libre à IDA Pro
- Complémentaire de :
 - `objdump / readelf` (bas niveau)
 - `gdb` (debugging pur)
 - Ghidra / Cutter (GUI)
- Interface graphique via Cutter
- radare2 est une boîte à outils complète pour comprendre, modifier et exploiter des binaires.

GHIDRA



Ghidra est une suite complète de reverse engineering développée par la NSA, spécialisée dans l'analyse statique et la décompilation de binaires.

Fonctions clés

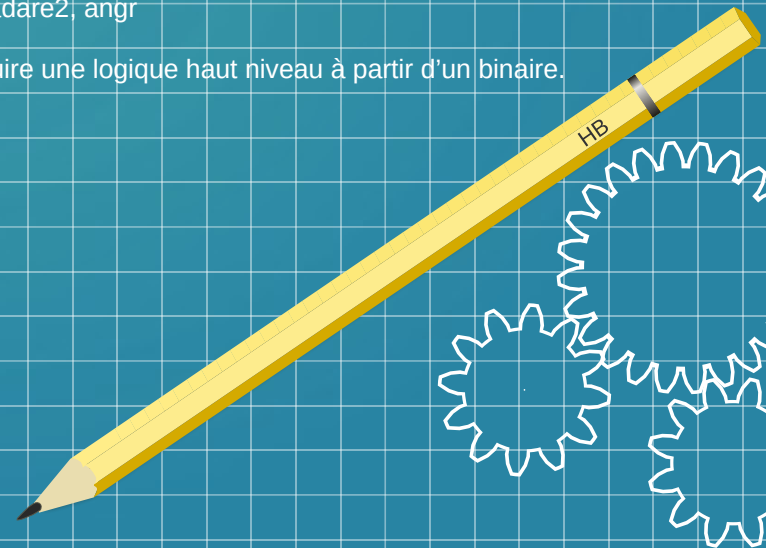
- Désassemblage multi-architectures
- Décompilation en pseudo-C
- Analyse automatique du flot de contrôle
- Analyse des symboles, fonctions, structures
- Scripting Java et Python (Jython)

Cas d'usage

- Reverse de binaires complexes
- Audit de firmware et malware
- Analyse sans code source
- Recherche sécurité

Positionnement

- Analyse statique avancée
- Alternative libre à IDA Pro
- Complémentaire de gdb, radare2, angr
- Ghidra permet de reconstruire une logique haut niveau à partir d'un binaire.



IDA FREE

IDA Free est la version gratuite d'IDA, outil historique du reverse engineering.

Fonctions clés

- Désassemblage interactif
- Graphes de contrôle
- Analyse des fonctions et appels
- Navigation avancée dans le code ASM

Limitations

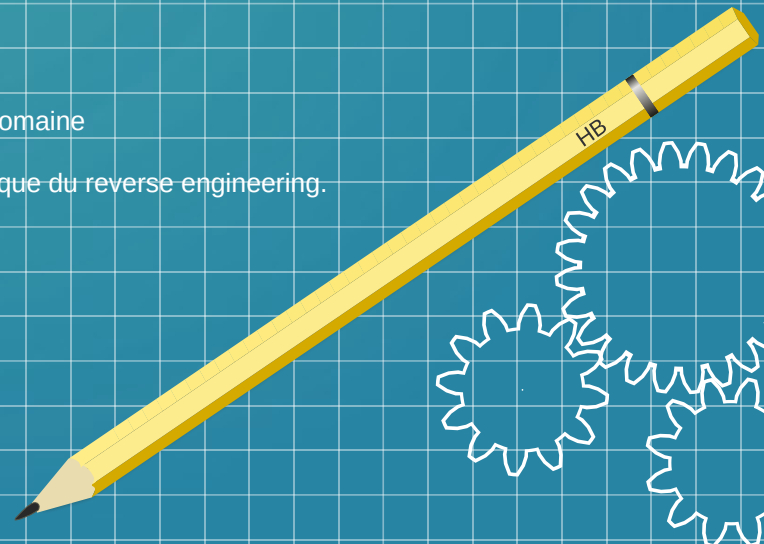
- Pas de décompilateur (Hex-Rays)
- Support architectures limité
- Usage non commercial

Cas d'usage

- Analyse rapide de binaires
- Étude de code assembleur
- Apprentissage du reverse

Positionnement

- Analyse statique
- Référence historique du domaine
- IDA est le standard historique du reverse engineering.



BINARY NINJA

Binary Ninja est un outil de reverse engineering moderne, axé sur la lisibilité et les représentations intermédiaires.

Fonctions clés

- Désassemblage et analyse statique
- Représentation IR (Low / Medium / High Level)
- Pseudo-code lisible
- API Python puissante

Cas d'usage

- Reverse structuré et lisible
- Automatisation et plugins
- Recherche sécurité

Positionnement

- Analyse statique avancée
- Concurrent direct de Ghidra et IDA
- Payant (version gratuite limitée)
- Binary Ninja privilégie la clarté et l'analyse intermédiaire.

KEYSTONE

Keystone est une bibliothèque d'assemblage multi-architectures utilisée pour générer du code machine.

Fonctions clés

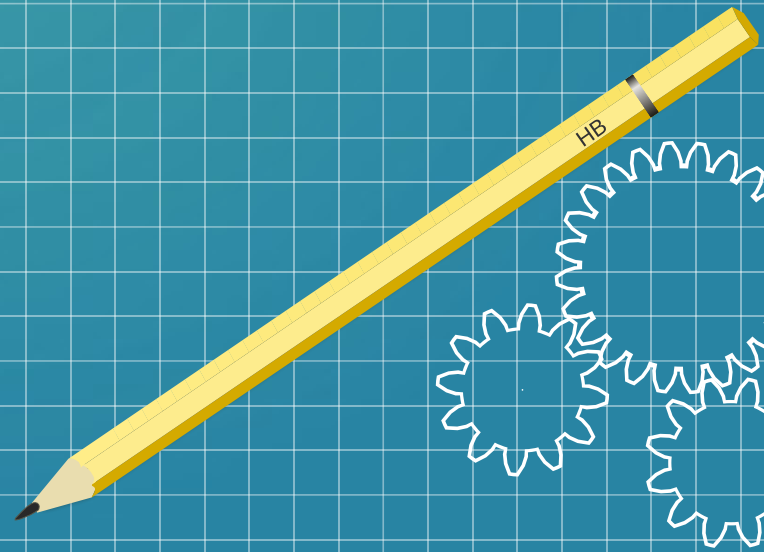
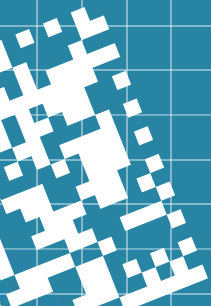
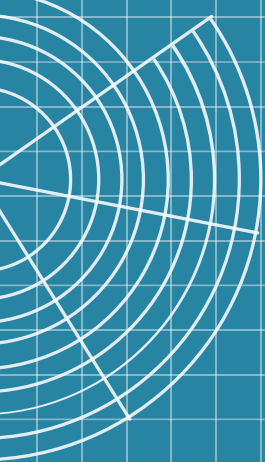
- Assemblage ASM → opcodes
- Support x86, ARM, MIPS, PPC, RISC-V...
- API C, Python, Rust, Go

Cas d'usage

- Génération de shellcode
- Patch binaire
- Exploitation et fuzzing
- Outils de reverse personnalisés

Positionnement

- Génération de code machine
- Complémentaire de Capstone
- Keystone transforme de l'assembleur en code exécutable.



CAPSTONE

Capstone est une bibliothèque de désassemblage légère et rapide.

Fonctions clés

- Désassemblage code machine → ASM
- Support multi-architectures
- API C, Python, Rust, Go...

Cas d'usage

- Analyse automatisée de binaires
- Moteurs de reverse custom
- Outils de sécurité et fuzzing

Positionnement

- Analyse bas niveau
- Utilisé par :
 - angr
 - radare2
 - frameworks de sécurité
- Capstone est le moteur de désassemblage programmable par excellence

CHECKSEC

checksec est un utilitaire de sécurité permettant d'analyser les mécanismes de protection activés sur un binaire ELF sous Linux.

Il est couramment utilisé en :

- audit sécurité
- exploitation binaire
- reverse engineering
- analyse de malware

Que vérifie checksec ?

checksec inspecte les protections au niveau ELF et runtime :

- NX (Non-eXecutable stack)
- PIE (Position Independent Executable)
- RELRO (Relocation Read-Only)
- Stack Canary
- Fortify Source
- RPATH / RUNPATH

👉 Sans exécuter le programme.



Protections analysées :



NX (DEP)

- Empêche l'exécution de code sur la pile
- Basé sur le segment GNU_STACK



PIE

- Permet l'ASLR sur les exécutables
- ELF de type ET_DYN



RELRO

- Protège la GOT
- Types :
 - Partial RELRO
 - Full RELRO



Stack Canary

- Détecte les débordements de pile
- Inséré par le compilateur



→ REVERSE avec gdb

<https://polytech-prog.gricad-pages.univ-grenoble-alpes.fr/polytech-microc/documents/gdb-refcard.pdf>

Utilisation de gdb

```
$ gdb --args ./mypasswd test
```

```
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
```

```
Copyright (C) 2024 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

```
This is free software: you are free to change and redistribute it.
```

```
There is NO WARRANTY, to the extent permitted by law.
```

```
Type "show copying" and "show warranty" for details.
```

```
This GDB was configured as "x86_64-linux-gnu".
```

```
Type "show configuration" for configuration details.
```

```
For bug reporting instructions, please see:
```

```
<https://www.gnu.org/software/gdb/bugs/>.
```

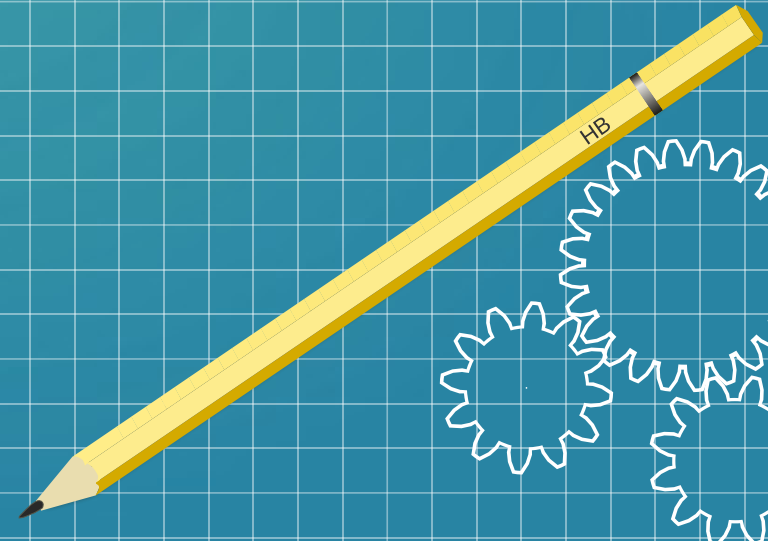
```
Find the GDB manual and other documentation resources online at:
```

```
<http://www.gnu.org/software/gdb/documentation/>.
```

```
For help, type "help".
```

```
Type "apropos word" to search for commands related to "word"...
```

```
Reading symbols from ./mypasswd...
```



GDB

execution jusqu'à son point d'entrée, le programme s'arrête à la fonction `_start()`

(gdb) starti

Starting program: `/home/tgayet/Workspace/linuxmaine/formations/reverse/mypasswd test`

This GDB supports auto-downloading debuginfo from the following URLs:

`<https://debuginfod.ubuntu.com>`

Enable debuginfod for this session? (y or [n]) `y`

Debuginfod has been enabled.

To make this setting permanent, add 'set debuginfod enabled on' to `.gdbinit`.

Downloading separate debug info for system-supplied DSO at `0x7ffff7fc3000`

Program stopped.

`0x00007ffff7fe4540 in _start () from /lib64/ld-linux-x86-64.so.2`

GDB

Lister les fonctions

(gdb) info functions

All defined functions:

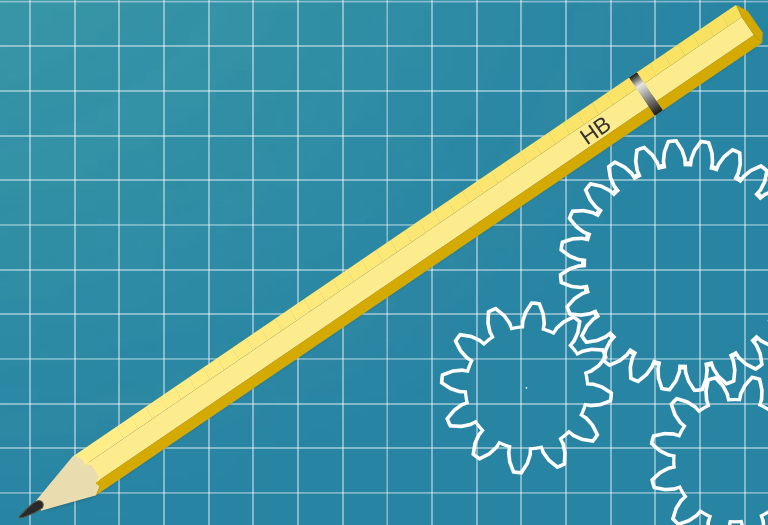
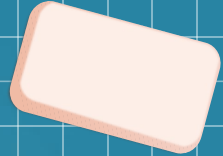
File ../elf/dl-tls.c:

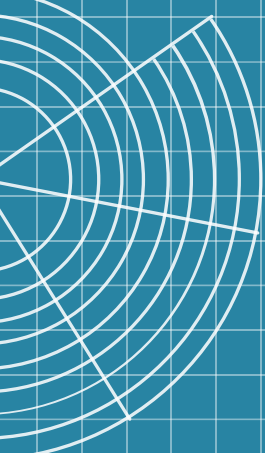
```
625: void * __GI__dl_allocate_tls(void *);
522: void * __GI__dl_allocate_tls_init(void *, _Bool);
635: void __GI__dl_deallocate_tls(void *, _Bool);
941: void * __tls_get_addr(tls_index *);
1016: void __dl_add_to_slotinfo(struct link_map *, _Bool);
416: void * __dl_allocate_tls_storage(void);
124: void __dl_assign_tls_modid(struct link_map *);
191: size_t __dl_count_modids(void);
215: void __dl_determine_tsoffset(void);
392: void __dl_get_tls_static_info(size_t *, size_t *);
1092: void __dl_init_static_tls(struct link_map *);
971: void * __dl_tls_get_addr_soft(struct link_map *);
96: void __dl_tls_static_surplus_init(size_t);
719: struct link_map * __dl_update_slotinfo(unsigned long, size_t);
479: static dtv_t * __dl_resize_dtv(dtv_t *, size_t);
853: static void * __tls_get_addr_tail(tls_index *, dtv_t *, struct link_map *);
914: static struct link_map * __update_get_addr(tls_index *, size_t);
```

File ../misc/sbrk.c:

```
36: void * __sbrk(intptr_t);
```

(...)





(...)

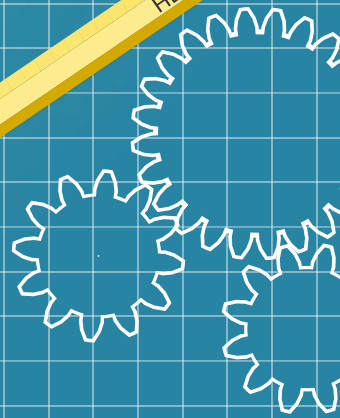
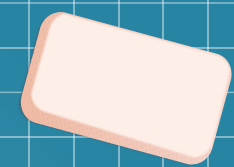
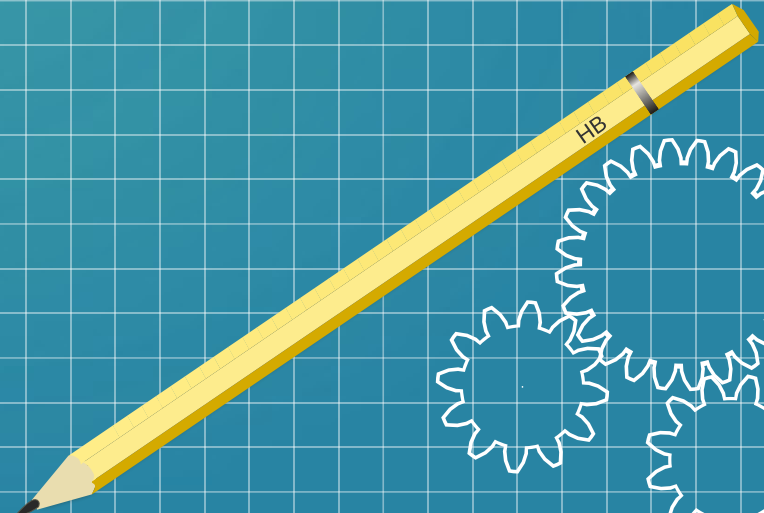
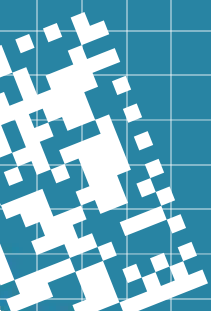
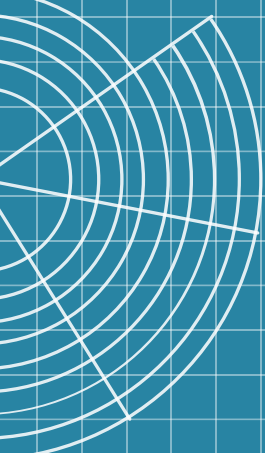
File mypasswd.c:

6: **int check_password(const char *);**

11: int main(int, char **, char **);

Non-debugging symbols:

0x0000555555555000 _init
0x0000555555555050 _cxa_finalize@plt
0x0000555555555060 printf@plt
0x0000555555555070 strcmp@plt
0x0000555555555080 _start
0x00005555555550b0 deregister_tm_clones
0x00005555555550e0 register_tm_clones
0x0000555555555120 _do_global_dtors_aux
0x0000555555555160 frame_dummy
0x0000555555555224 _fini
0x00007ffff7fc3fb0 _vdso_gettimeofday
0x00007ffff7fc3fb0 gettimeofday
0x00007ffff7fc3fc0 __vdso_time
0x00007ffff7fc3fc0 time
0x00007ffff7fc3ff0 _vdso_clock_gettime
0x00007ffff7fc3ff0 clock_gettime
0x00007ffff7fc4000 _vdso_clock_getres
0x00007ffff7fc4000 clock_getres
0x00007ffff7fc4080 __vdso_getcpu
0x00007ffff7fc4080 getcpu
0x00007ffff7fc40b0 __vdso_getrandom
0x00007ffff7fc40b0 getrandom
0x00007ffff7fc45f0 _vdso_sgx_enter_enclave



GDB

positionnement d'un point d'arrêt

```
(gdb) break check_password
```

```
Breakpoint 1 at 0x55555555179: file mypasswd.c, line 8.
```

Ensuite il existe plusieurs commandes pour poursuivre l'exécution :

- **continue** ou **c** : pour laisser le binaire s'exécuter jusqu'au prochain point d'arrêt ou jusqu'à sa terminaison
- **step** ou **n (step into)** : pour exécuter la prochaine instruction et rentrer dans l'appel de fonction si un appel de ce type à lieu
- **next** ou **n :(step over)** : pour exécuter la prochaine instruction sans rentrer dans l'appel de fonction si un appel de ce type à lieu

GDB

Continuer le debug

(gdb) c

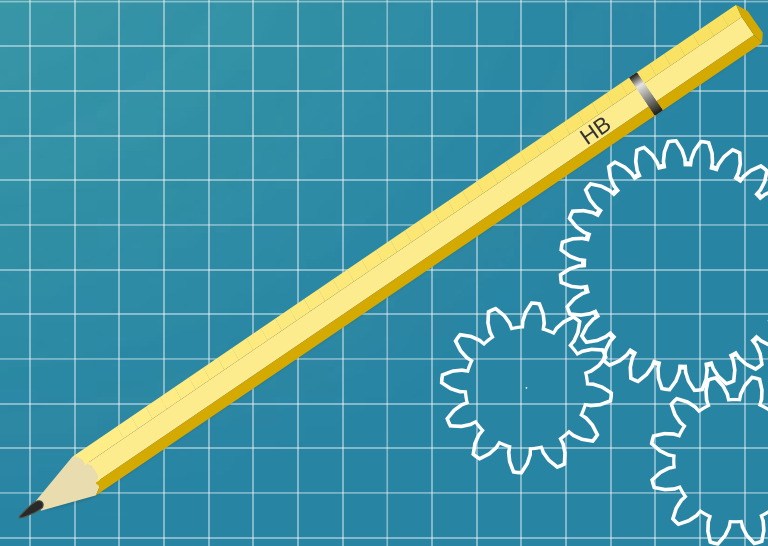
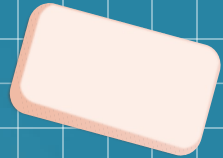
Continuing.

[Thread debugging using libthread_db enabled]

Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, check_password (password=0x7ffffffd93c "test") at mypasswd.c:8

```
8      return strcmp(password, SECRET) == 0;
```



GDB

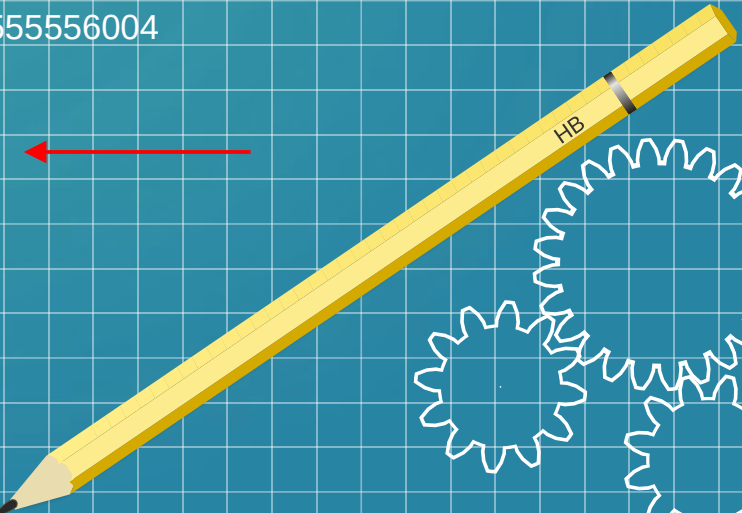
obtention de l'assembleur de la fonction `check_password`
et nous voyons l'appel à la fonction `strcmp`

(gdb) disassemble

Dump of assembler code for function `check_password`:

```
0x000055555555169 <+0>: endbr64
0x00005555555516d <+4>: push  %rbp
0x00005555555516e <+5>: mov  %rsp,%rbp
0x000055555555171 <+8>: sub  $0x10,%rsp
0x000055555555175 <+12>: mov  %rdi,-0x8(%rbp)
=> 0x000055555555179 <+16>: mov  -0x8(%rbp),%rax
0x00005555555517d <+20>: lea  0xe80(%rip),%rdx    # 0x555555556004
0x000055555555184 <+27>: mov  %rdx,%rsi
0x000055555555187 <+30>: mov  %rax,%rdi
0x00005555555518a <+33>: call 0x555555555070 <strcmp@plt>
0x00005555555518f <+38>: test %eax,%eax
0x000055555555191 <+40>: sete %al
0x000055555555194 <+43>: movzbl %al,%eax
0x000055555555197 <+46>: leave
0x000055555555198 <+47>: ret
```

End of assembler dump.



GDB

execution jusqu'à l'appel à strcmp

(gdb) break *check_password+33

Breakpoint 2 at 0x5555555518a: file mypasswd.c, line 8.

(gdb) c

Continuing.

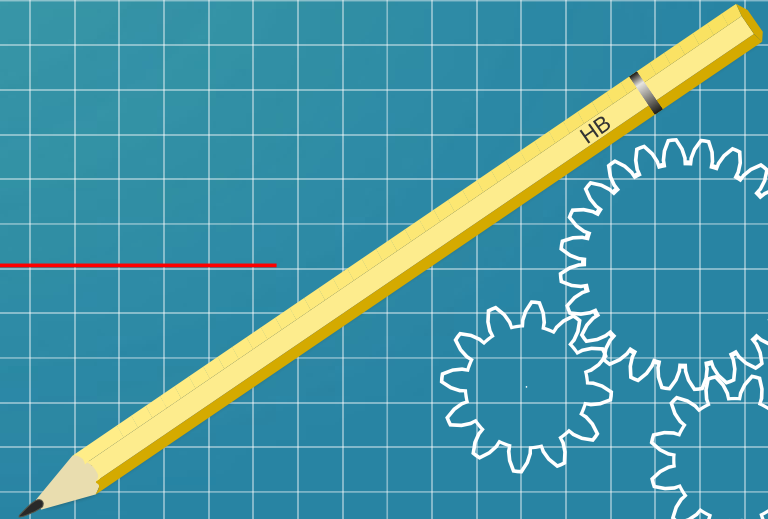
Breakpoint 2, 0x00005555555518a in check_password (password=0x7ffffffd93c "test") at mypasswd.c:8
8 return strcmp(password, SECRET) == 0;

(gdb) disassemble

Dump of assembler code for function check_password:

```
0x000055555555169 <+0>: endbr64
0x00005555555516d <+4>: push %rbp
0x00005555555516e <+5>: mov %rsp,%rbp
0x000055555555171 <+8>: sub $0x10,%rsp
0x000055555555175 <+12>: mov %rdi,-0x8(%rbp)
0x000055555555179 <+16>: mov -0x8(%rbp),%rax
0x00005555555517d <+20>: lea 0xe80(%rip),%rdx # 0x555555556004
0x000055555555184 <+27>: mov %rdx,%rsi
0x000055555555187 <+30>: mov %rax,%rdi
=> 0x00005555555518a <+33>: call 0x55555555070 <strcmp@plt>
0x00005555555518f <+38>: test %eax,%eax
0x000055555555191 <+40>: sete %al
0x000055555555194 <+43>: movzbl %al,%eax
0x000055555555197 <+46>: leave
0x000055555555198 <+47>: ret
```

End of assembler dump.



GDB

affichage des registres

Le premier argument est placé dans le registre RAX et le second dans le registre RSI :

(gdb) info registers

rax	0x7fffffff93c	140737488345404		
rbx	0x7fffffff418	140737488344088		
rcx	0x55555557db8	93824992247224	rip	0x5555555518a 0x5555555518a <check_password+33>
rdx	0x55555556004	93824992239620	eflags	0x202 [IF]
rsi	0x55555556004	93824992239620	cs	0x33 51
rdi	0x7fffffff93c	140737488345404	ss	0x2b 43
rbp	0x7fffffff2c0	0x7fffffff2c0	ds	0x0 0
rsp	0x7fffffff2b0	0x7fffffff2b0	es	0x0 0
r8	0x0	0	fs	0x0 0
r9	0x7fff7fca380	140737353917312	gs	0x0 0
r10	0x7fffffff010	140737488343056	k0	0x8000000 134217728
r11	0x203	515	k1	0x40400 263168
r12	0x2	2	k2	0x0 0
r13	0x0	0	k3	0x0 0
r14	0x55555557db8	93824992247224	k4	0x0 0
r15	0x7fffffd000	140737354125312	k5	0x0 0
			k6	0x0 0
			k7	0x0 0
			fs_base	0x7fff7f97740 140737353709376
			gs_base	0x0 0

CLAUDE (IA)

dans /Documents/PERSO/reverse j'ai un fichier c compile le avec gcc puis récupère l'assembleur avec objdump et essaye de reformer le code C depuis l'assembleur en mode balckblox (sans connaissance aucune du code C d'origine)

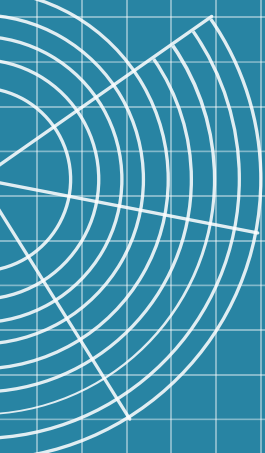
∴ Thinking...

The user wants me to:

1. Find a C file in /home/tgayet/Documents/PERSO/reverse
2. Compile it with gcc
3. Use objdump to get the assembly
4. Then try to reconstruct the C code from the assembly alone (blackbox - without looking at the original C code)

Let me first find the C file(s) in that directory.

- Je vais d'abord trouver le fichier C, le compiler, récupérer l'assembleur, puis tenter de reconstituer le code C uniquement depuis l'assembleur.
- Search(pattern: "**/*.c", path: "~/Documents/PERSO/reverse")
 - └ Found 1 file (ctrl+o to expand)
- Trouvé : mypasswd.c. Je vais le compiler sans le lire (mode blackbox).
- Bash(gcc -o /home/tgayet/Documents/PERSO/reverse/mypasswd /home/tgayet/Documents/PERSO/reverse/mypasswd.c -no-pie 2>&1)
 - └ (No output)
- Compilation réussie. Maintenant, récupérons l'assembleur avec objdump.
- Bash(objdump -d -M intel /home/tgayet/Documents/PERSO/reverse/mypasswd 2>&1)



L /home/tgayet/Documents/PERSO/reverse/mypasswd: file format elf64-x86-64

Disassembly of section .init:

000000000401000 <_init>:

401000: f3 0f 1e fa endbr64

401004: 48 83 ec 08 sub rsp,0x8

401008: 48 8b 05 e9 2f 00 00 mov rax,QWORD PTR [rip+0x2fe9] # 403ff8 <__gmon_start__@Base>

40100f: 48 85 c0 test rax,rax

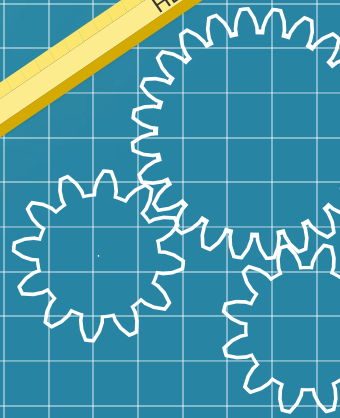
401012: 74 02 je 401016 <_init+0x16>

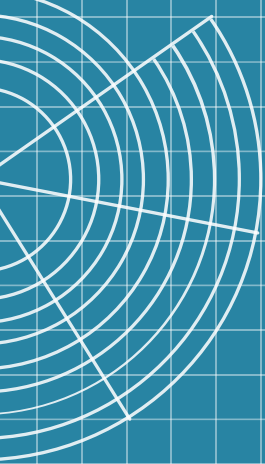
401014: ff d0 call rax

401016: 48 83 c4 08 add rsp,0x8

40101a: c3 ret

Disassembly of section .plt:





∴ Thinking...

Now let me extract the strings from the binary to understand what string literals are used, since I can see references to data at addresses like 0x402004, 0x402014, 0x40202e, 0x402045.

- Bien. Maintenant récupérons les chaînes de caractères référencées dans le code.
- Bash(objdump -s -j .rodata /home/tgayet/Documents/PERSO/reverse/mypasswd 2>&1)

```
L /home/tgayet/Documents/PERSO/reverse/mypasswd:  file format elf64-x86-64
```

Contents of section .rodata:

```
402000 01000200 31323334 35363738 39415a45  ....123456789AZE
```

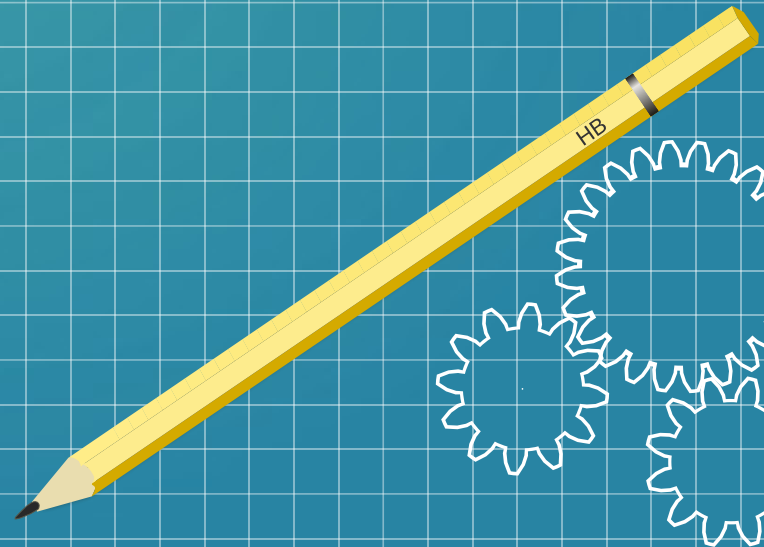
```
402010 52545900 55736167 653a2025 73203c6d  RTY.Usage: %s <m
```

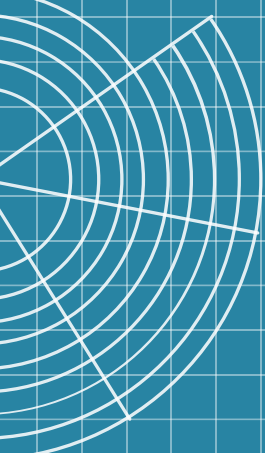
```
402020 6f745f64 655f7061 7373653e 0a004d6f  ot_de_passe>..Mo
```

```
402030 74206465 20706173 73652063 6f727265  t de passe corre
```

```
402040 63742021 004d6f74 20646520 70617373  ct!.Mot de pass
```

```
402050 6520696e 636f7272 65637420 2100      e incorrect!.
```





∴ Thinking...

Now I have all the information I need to reconstruct the C code from the assembly. Let me analyze:

Strings at known addresses:

- 0x402004: "123456789AZERTY" (the password)

- 0x402014: "Usage: %s <mot_de_passe>\n"

- 0x40202e: "Mot de passe correct !" (actually 0x40202e - let me check... 0x402014 = "Usage: %s \n\0" that's 26 bytes, so 0x402014 + 26 = 0x40202e. Yes: "Mot de passe correct !")

- 0x402045: "Mot de passe incorrect !"

Function check_password (0x401156):

- Takes one argument (rdi = pointer to string)

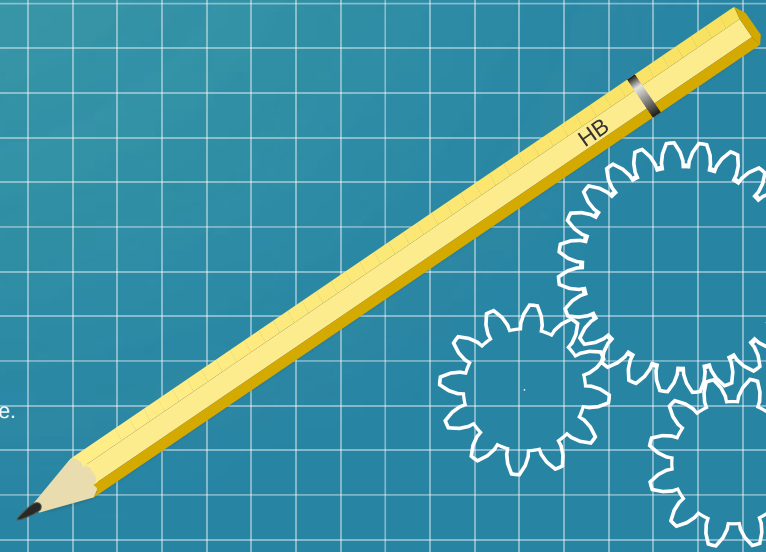
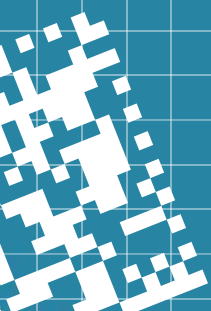
- Saves it to [rbp-0x8]

- Loads the string at 0x402004 ("123456789AZERTY") into rdx

- Calls strcmp(rdi, "123456789AZERTY")

- Returns (eax == 0) ? 1 : 0 (sete al, then movzx eax, al)

So the function compares the input password against the hardcoded string and returns 1 if they match, 0 otherwise.



Function main (0x401186):

The main function checks if a password argument was provided, and if so, validates it using check_password. If the password matches "123456789AZERTY", it prints the success message; otherwise it prints the failure message.

- Parfait. J'ai maintenant tout ce qu'il faut. Voici mon analyse complète de l'assembleur, sans avoir lu le fichier source.

Analyse de l'assembleur

Chaînes trouvées dans .rodata

Adresse	Chaîne
0x402004	"123456789AZERTY"
0x402014	"Usage: %s <mot_de_passe>\n"
0x40202e	"Mot de passe correct !"
0x402045	"Mot de passe incorrect !"

Fonction check_password (0x401156)

- Reçoit un argument rdi (un char*)
- Compare cet argument avec la chaîne "123456789AZERTY" via strcmp
- sete al + movzx eax, al : retourne 1 si les chaînes sont égales, 0 sinon

Fonction main (0x401186)

- Reçoit argc (edi), argv (rsi), envp (rdx)
- Si argc <= 1 : affiche l'usage avec argv[0] et retourne 1
- Sinon : appelle check_password(argv[1])
 - Si retour non nul → affiche "Mot de passe correct !"
 - Sinon → affiche "Mot de passe incorrect !"
- Retourne 0

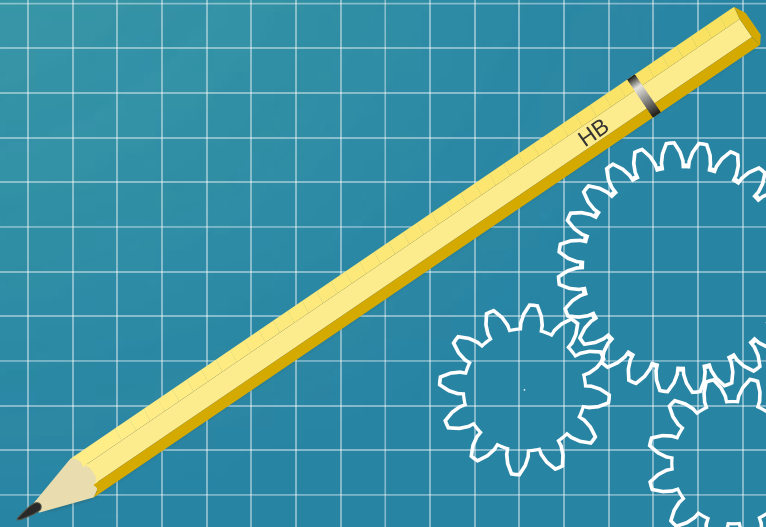
Code C reconstitué

- Write(/home/tgayet/Documents/PERSO/reverse/mypasswd_reversed.c)

L Wrote 22 lines to /home/tgayet/Documents/PERSO/reverse/mypasswd_reversed.c

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int check_password(const char *password) {
5     return strcmp(password, "123456789AZERTY") == 0;
6 }
7
8 int main(int argc, char *argv[], char *envp[]) {
9     if (argc <= 1) {
10         printf("Usage: %s <mot_de_passe>\n", argv[0]);
11         return 1;
12     }
13
14     if (check_password(argv[1])) {
15         printf("Mot de passe correct !");
16     } else {
17         printf("Mot de passe incorrect !");
18     }
19
20     return 0;
21 }
```

- Vérifions que le code reconstitué compile et se comporte identiquement.



- Bash(gcc -o /home/tgayet/Documents/PERSO/reverse/mypasswd_reversed /home/tgayet/Documents/PERSO/reverse/mypasswd_reversed.c -no-pie 2>&1)

└ (No output)

- Bash(# Test sans argument, avec bon mot de passe, avec mauvais mot de passe - original vs reversed)

```
echo "=== ORIGINAL ===" && /home/tgayet/Documents/PERSO/reverse/mypasswd 2>&1; echo "(exit: $?)" &&  
/home/tgayet/Documents/PERSO/reverse/mypasswd 123456789AZERTY 2>&1; echo "(exit: $?)" && /home/tgayet/Documents/PERSO/reverse/mypasswd  
wrong 2>&1; echo "(exit: $?)" && echo "" && echo "=== REVERSED ==="
```

```
&& /home/tgayet/Documents/PERSO/reverse/mypasswd_reversed 2>&1; echo "(exit: $?)" &&  
/home/tgayet/Documents/PERSO/reverse/mypasswd_reversed 123456789AZERTY 2>&1; echo "(exit: $?)" &&  
/home/tgayet/Documents/PERSO/reverse/mypasswd_reversed wrong 2>&1; echo "(exit: $?)"
```

└ === ORIGINAL ===

Usage: /home/tgayet/Documents/PERSO/reverse/mypasswd <mot_de_passe>

(exit: 1)

Mot de passe correct !(exit: 0)

Mot de passe incorrect !(exit: 0)

=== REVERSED ===

Usage: /home/tgayet/Documents/PERSO/reverse/mypasswd_reversed <mot_de_passe>

(exit: 1)

Mot de passe correct !(exit: 0)

Mot de passe incorrect !(exit: 0)

- Les deux binaires ont un comportement strictement identique dans les 3 cas de test :

Test	Original	Reversed
Sans argument	Usage: ... <mot_de_passe> (exit 1)	Identique
Bon mot de passe (123456789AZERTY)	Mot de passe correct ! (exit 0)	Identique
Mauvais mot de passe	Mot de passe incorrect ! (exit 0)	Identique

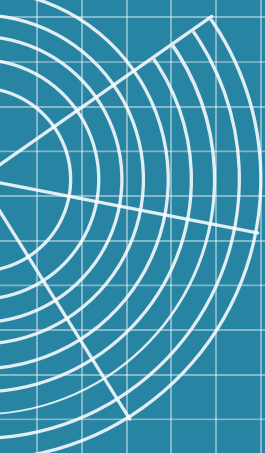
Résumé

Le programme est un simple vérificateur de mot de passe en ligne de commande :

- Mot de passe hardcodé : 123456789AZERTY (visible en clair dans .rodata - faille de sécurité classique)
- La fonction `check_password` compare avec `strcmp` et retourne un booléen
- `main` gère les arguments et affiche le résultat

Le code reconstitué est dans `mypasswd_reversed.c` et produit un comportement identique à l'original.

* Sautéed for 1m 2s



Code source reconstitué à partir de l'assembleur :

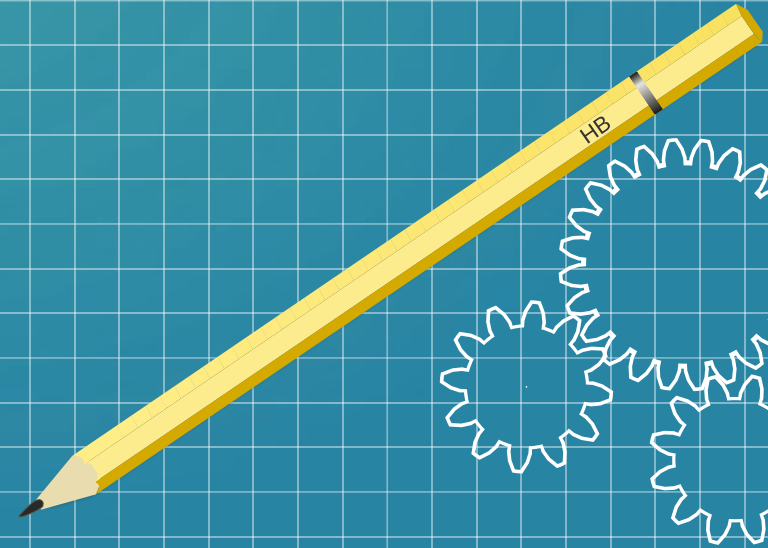
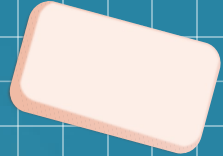
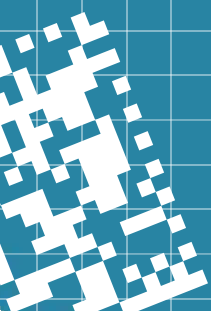
```
#include <stdio.h>
#include <string.h>
```

```
int check_password(const char *password) {
    return strcmp(password, "123456789AZERTY") == 0;
}
```

```
int main(int argc, char *argv[], char *envp[]) {
    if (argc <= 1) {
        printf("Usage: %s <mot_de_passe>\n", argv[0]);
        return 1;
    }

    if (check_password(argv[1])) {
        printf("Mot de passe correct !");
    } else {
        printf("Mot de passe incorrect !");
    }

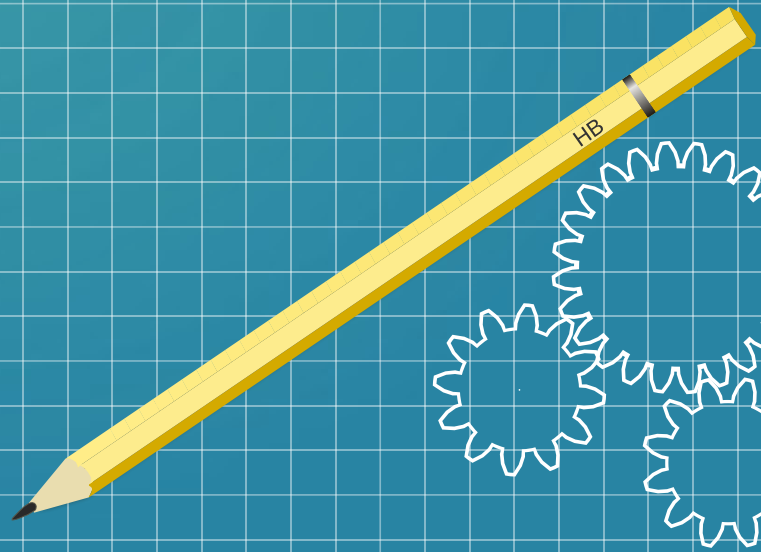
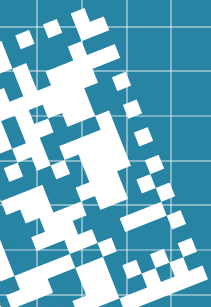
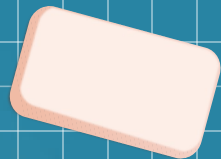
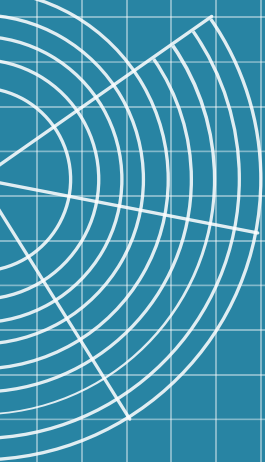
    return 0;
}
```





→ REVERSE avec gdb

<https://polytech-prog.gricad-pages.univ-grenoble-alpes.fr/polytech-microc/documents/gdb-refcard.pdf>





→ REVERSE avec claudia

Pré-analyse du binaire (avant angr)

Toujours commencer hors angr

- `file binaire`
- `checksec binaire`
- `strings binaire`
- `readelf -h binaire`

🎯 Objectifs :

- Architecture (x86_64, ARM...)
- PIE / NX / RELRO ?
- Dynamique ou statique ?
- Présence de chaînes intéressantes (password, error, success)

⚡ angr ne remplace pas l'analyse statique classique, il la complète.

ANGR

Installation et environnement angr

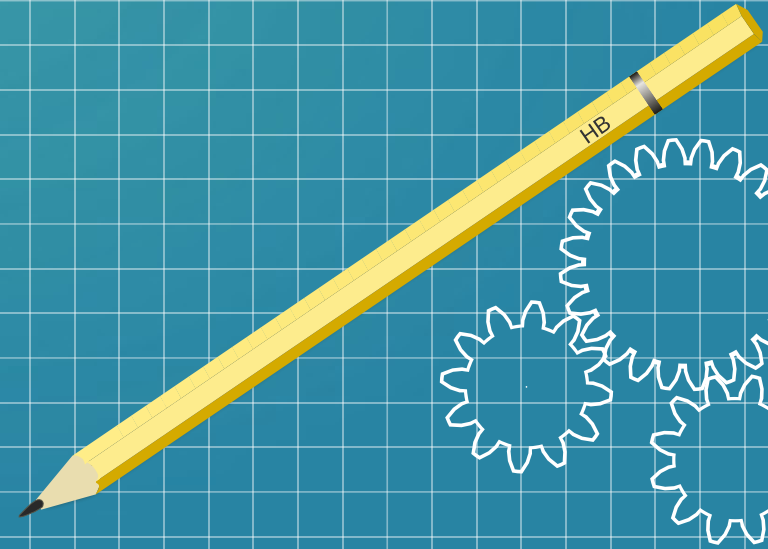
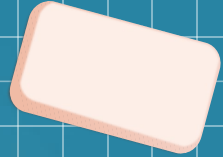
```
P$ python3 -m venv angr-env
```

```
$ source angr-env/bin/activate
```

```
$ pip install angr claripy
```

✓ angr fonctionne exclusivement en Python

✓ Utilise un solveur SMT (claripy + z3)



ANGR

Création du projet angr

```
import angr  
  
proj = angr.Project(  
    "./mypasswd",  
    auto_load_libs=False  
)
```

Bonnes pratiques

- `auto_load_libs=False` ← évite l'explosion des chemins
- Toujours travailler sans libc sauf besoin précis

ANGR

Identifier le point d'entrée logique

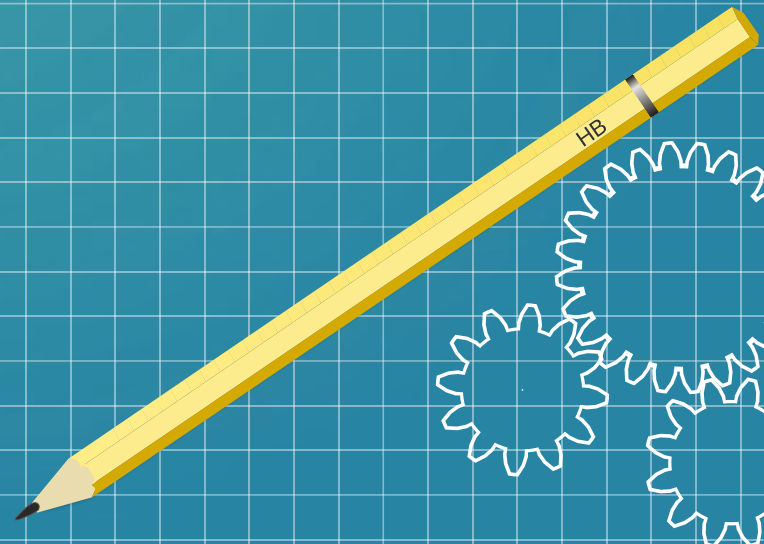
Options possibles :

- `main`
- `_start`

Une fonction trouvée via `objdump`, `radare2`, `ghidra`

Exemple :

```
$ Objdump -d binaire | grep main
```



ANGR

Créer un état initial symbolique

Cas courant : entrée utilisateur (stdin)

```
import claripy

input_size = 16

user_input = claripy.BVS("input", input_size * 8)

state = proj.factory.full_init_state(
    args=["./mypasswd"],
    stdin=user_input
)
```

🎯 Ici :

```
user_input = entrée symbolique

angr testera toutes les valeurs possibles
```

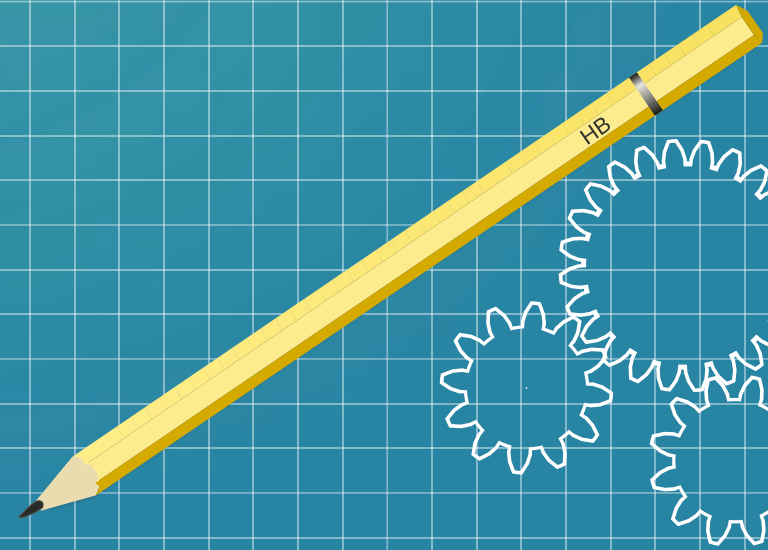
ANGR

Créer le gestionnaire d'exploration

```
simgr = proj.factory.simgr(state)
```

simgr gère :

- chemins actifs
- chemins trouvés
- chemins morts
- erreurs



ANGR

Définir les objectifs (find / avoid)

Méthodes possibles

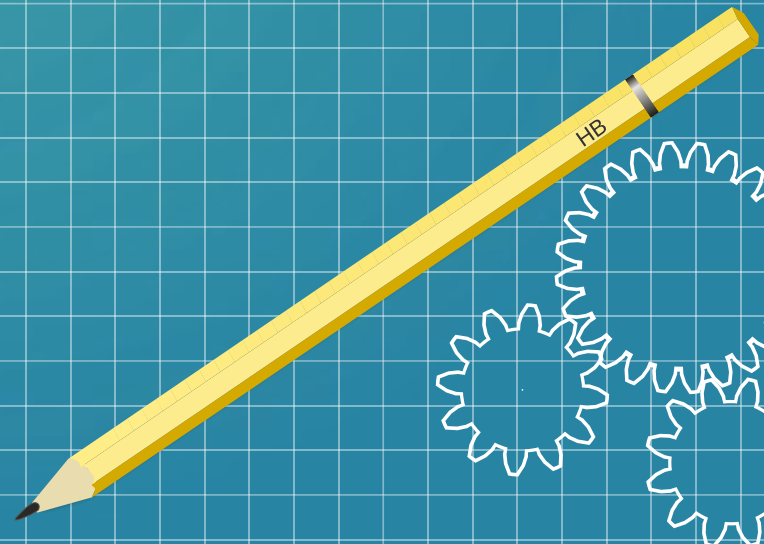
- Adresse mémoire
- Message affiché
- Code de retour
- Appel système

Exemple avec chaînes repérées via strings

```
strings binaire | grep -i success
```

Puis :

```
def is_success(state):  
    return b"Access granted" in state.posix.dumps(1)  
  
def is_failure(state):  
    return b"Access denied" in state.posix.dumps(1)
```



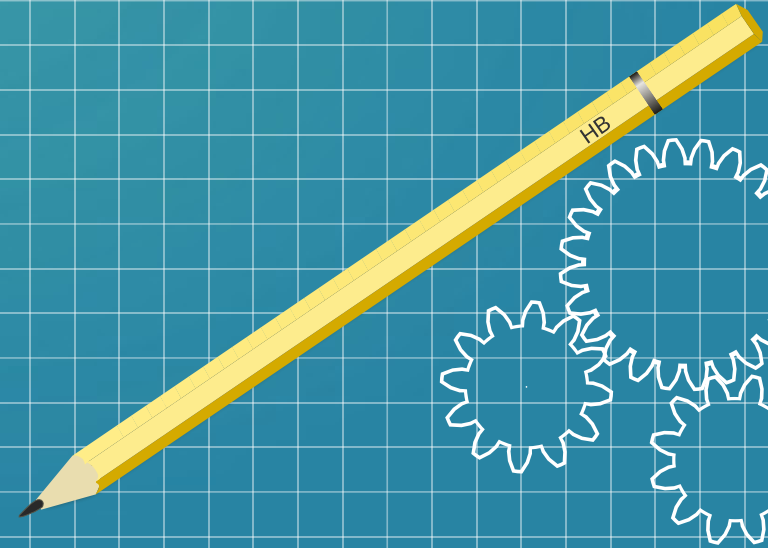
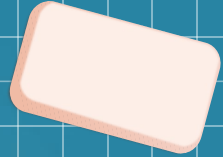
ANGR

Lancer l'exploration symbolique

```
simgr.explore(  
    find=is_success,  
    avoid=is_failure  
)
```

🎯 angr explore automatiquement tous les chemins possibles

⚠️ Peut être coûteux si trop de branches → filtrer tôt

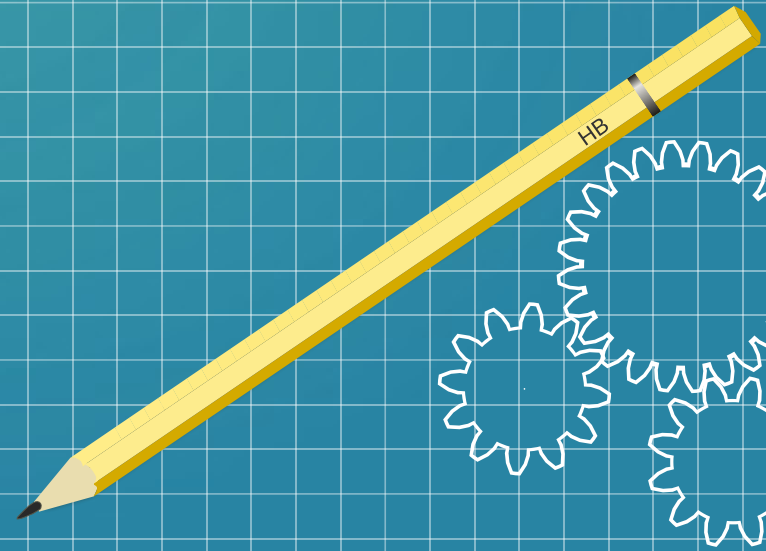


ANGR

Extraire la solution

```
if simgr.found:  
    found = simgr.found[0]  
    solution = found.solver.eval(user_input, cast_to=bytes)  
    print("Input valide :", solution)
```

👉 Tu récupères l'entrée exacte qui mène au succès 🏆

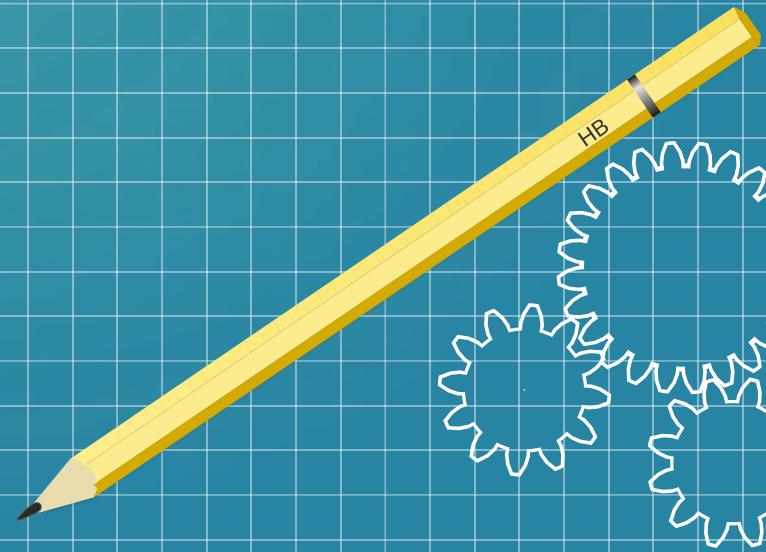


ANGR

Vérification réelle

```
$ echo "<solution>" | ./mypasswd
```

- ✓ Confirme que le reverse est correct
- ✓ angr a trouvé un chemin valide, pas une supposition



ANGR

Cas avancés

Limiter les chemins

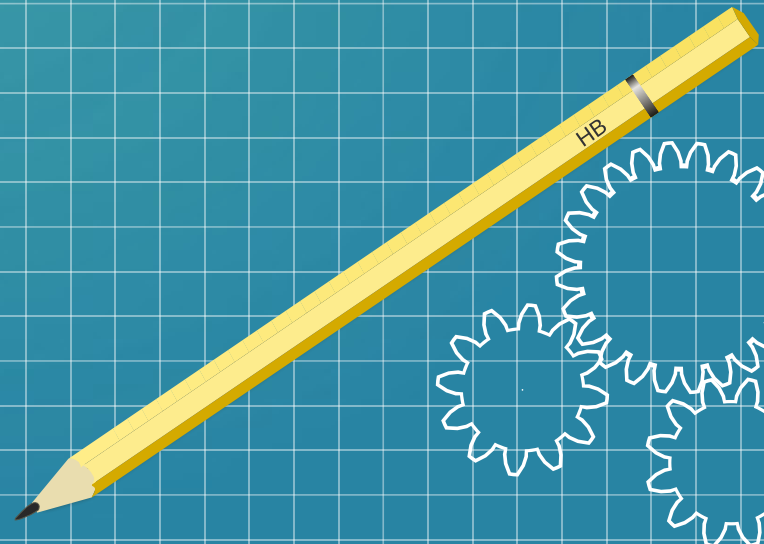
```
simgr.use_technique(angr.exploration_techniques.DFS())
```

Ignorer certaines fonctions

```
proj.hook(addr, angr.SIM_PROCEDURES['libc']['puts']())
```

Travailler sur une fonction spécifique

```
state = proj.factory.call_state(func_addr, user_input)
```



ANGR

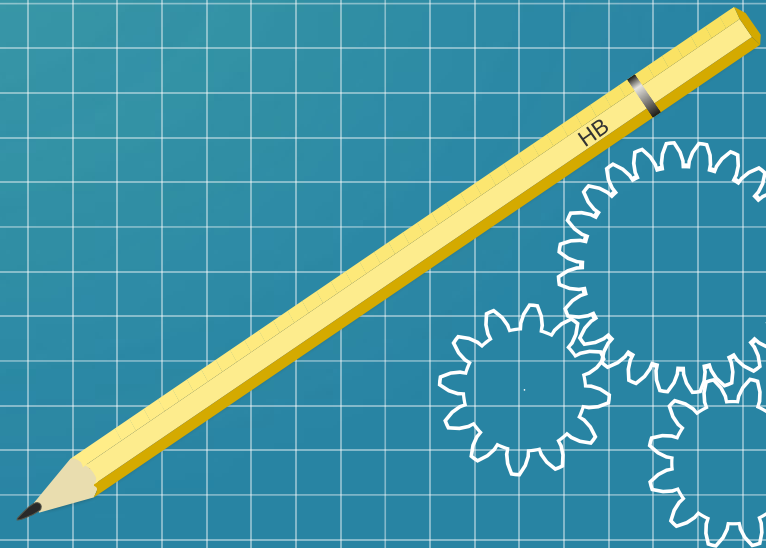
Bonnes pratiques angr

✓ Toujours :

- Réduire la taille des entrées symboliques
- Éviter les bibliothèques dynamiques
- Utiliser avoid agressivement
- Pré-analyser avec Ghidra / radare2

✗ Éviter :

- Symboliser tout le programme
- Explorer sans objectif clair
- Lancer angr sans analyse statique préalable



ANGR

Positionnement dans un workflow reverse

file → strings → readelf → ghidra



angr



gdb (validation)

angr permet d'explorer automatiquement tous les chemins d'un binaire ELF et de calculer les entrées menant à un état donné, sans exécution réelle.

ANGR

ETAPE 1



Pour un binaire sans symboles de debug, tu peux quand même trouver tout ce qu'il faut avec objdump/strings/gdb.

Trouver les adresses des messages

Lister les chaînes lisibles :

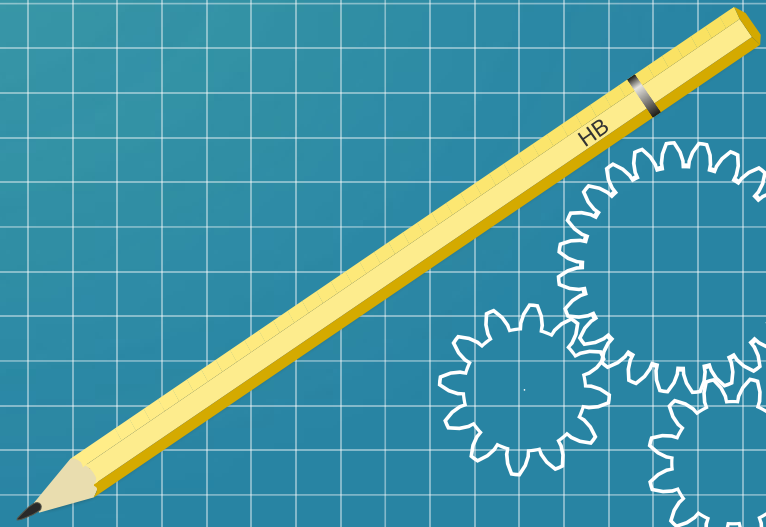
```
$ strings -tx mypasswd | grep "Mot de passe"
```

Tu obtiens quelque chose comme :

```
00404030 Mot de passe correct !
```

```
00404050 Mot de passe incorrect !
```

Ce sont les adresses dans la section `.rodata`.



ANGR

ETAPE 2

Retrouver les appels qui utilisent ces chaînes :

```
$ objdump -d mypasswd | less
```

Puis tu cherches 404030 (adresse de la chaîne sans les zéros de poids fort) :

```
text0000000000401190 <main>:
```

```
...
```

```
4011a3: 48 8d 3d 86 2e 00 00    lea    0x2e86(%rip),%rdi # 404030 <"Mot de passe correct !">
```

```
4011aa: e8 61 fe ff ff         call   401010 <puts@plt>
```

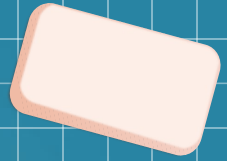
Ici :

L'adresse de l'instruction call à puts avec "Mot de passe correct !" est ton `success_addr` pour angr (ici 0x4011aa).

Tu fais la même chose pour "Mot de passe incorrect !" et tu prends l'adresse de ce call comme `failure_addr`.

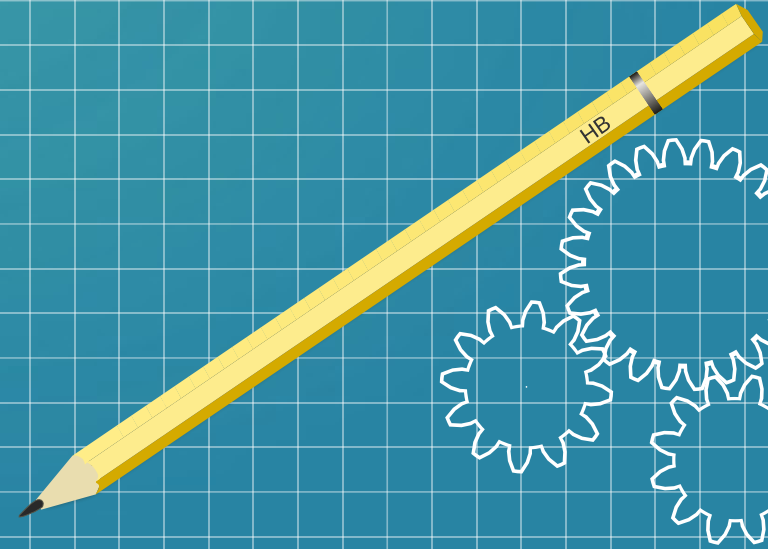
ANGR

ETAPE 3



Même sans connaître la constante interne, on connaît :

- Le point d'entrée (main ou l'entrypoint ELF).
- Le fait que le binaire lit un mot de passe (argv ou stdin).
- Le fait qu'il affiche "Mot de passe correct !" en cas de succès et "Mot de passe incorrect !" en cas d'échec.
- Ces deux messages sont visibles dans strings et/ou au runtime, donc ils sont légitimes pour définir la condition de succès dans angr.



ANGR

ETAPE 4

Deux approches possibles :

- Par adresse (statique) : tu regardes seulement où le binaire affiche "Mot de passe correct !".

`strings -tx mybin | grep "Mot de passe correct"` → te donne l'adresse de la chaîne dans `.rodata`, mais ça ne révèle pas le mot de passe.

Avec `objdump -d mypasswd`, tu trouves le call `puts/printf` qui utilise cette chaîne (tu ne touches pas à `SECRET`, tu ne le cherches pas).

L'adresse de ce call (ou du basic block qui le contient) devient ton `success_addr`.

L'adresse du call qui affiche "Mot de passe incorrect !" devient ton `failure_addr`.

Tu n'as toujours pas regardé la constante du mot de passe, juste les chaînes de sortie.

- Par sortie (dynamique, "clean") : tu ne regardes même pas les adresses dans le binaire.

Tu laisses `angr` exécuter symboliquement, et tu définis une fonction `is_successful(state)` qui retourne `True` si `stdout` contient b"Mot de passe correct" et `False` sinon.

Optionnel : `should_abort(state)` qui retourne `True` si `stdout` contient b"Mot de passe incorrect".

Tu appelles `simgr.explore(find=is_successful, avoid=should_abort)` et tu ne manipules jamais les constantes internes.

La deuxième approche est la plus "CTF-friendly" et ne nécessite ni debug info ni désassemblage profond.

ANGR

Script angr sans connaissance de la constante

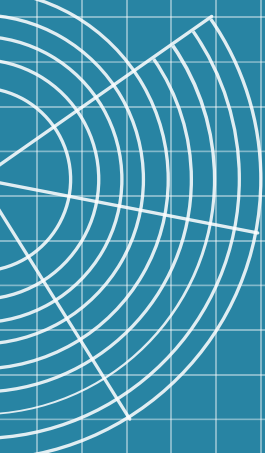
Exemple complet, qui ne suppose pas la valeur du mot de passe, uniquement la présence des messages de succès/échec :

```
import angr
import claripy
import sys

proj = angr.Project("./mypasswd", auto_load_libs=False)

# On suppose que le mot de passe fait au plus 32 chars (à ajuster)
passwd_len = 32
passwd_chars = [claripy.BVS(f'c{i}', 8) for i in range(passwd_len)]
passwd = claripy.Concat(*passwd_chars)

# Contraintes de base : ASCII imprimable
for c in passwd_chars:
    proj.solver.add(c >= 0x20)
    proj.solver.add(c <= 0x7e)
```



```
# argv[1] = mot de passe symbolique
argv = ["/mypasswd", passwd]
state = proj.factory.full_init_state(args=argv)
simgr = proj.factory.simulation_manager(state)

def is_successful(s):
    out = s.posix.dumps(sys.stdout.fileno())
    return b"Mot de passe correct" in out

def should_abort(s):
    out = s.posix.dumps(sys.stdout.fileno())
    return b"Mot de passe incorrect" in out

simgr.explore(find=is_successful, avoid=should_abort)

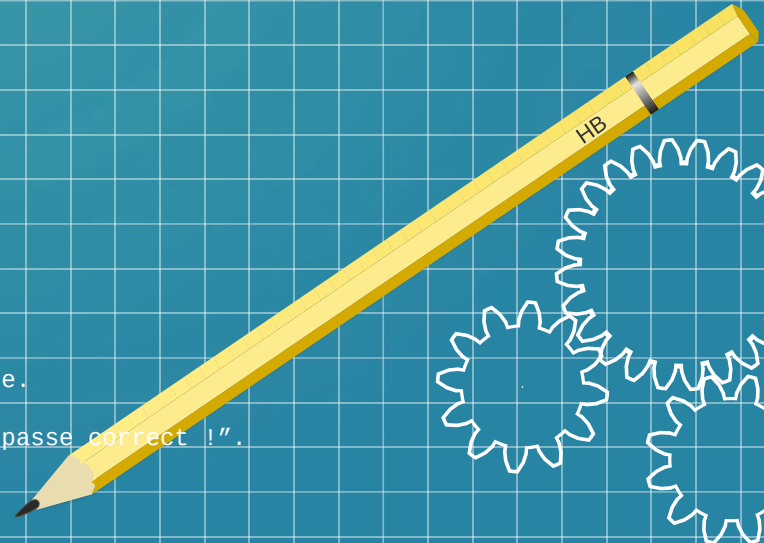
if simgr.found:
    found = simgr.found[0]
    solution = found.solver.eval(passwd, cast_to=bytes)
    print(solution)
else:
    print("No solution found")
```

Ici :

Tu ne lis jamais la constante SECRET en clair.

Tu ne cherches pas l'adresse de SECRET, tu ne regardes que la sortie texte.

angr "devine" l'input qui déclenche le chemin où stdout contient "Mot de passe correct !".



ANGR

SCRIPT FINAL

```
import angr
import claripy
import sys

proj = angr.Project("./mypasswd", auto_load_libs=False)

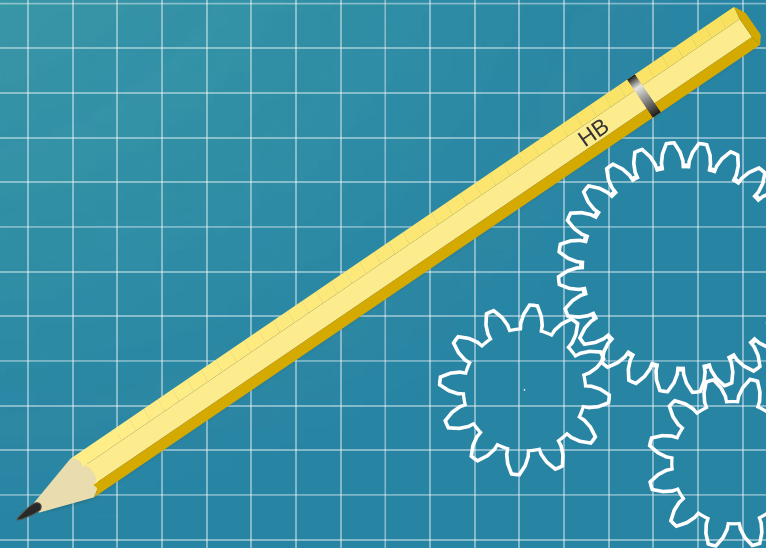
# On suppose que le mot de passe est court (<= 32 chars)
PW_LEN = 32

# argv[1] symbolique
pw_chars = [claripy.BVS(f'c{i}', 8) for i in range(PW_LEN)]
pw = claripy.Concat(*pw_chars)

# Contraintes: ASCII imprimable
for c in pw_chars:
    proj.solver.add(c >= 0x20)
    proj.solver.add(c <= 0x7e)

argv = ["./mypasswd", pw]

state = proj.factory.full_init_state(args=argv)
simgr = proj.factory.simulation_manager(state)
```



ANGR

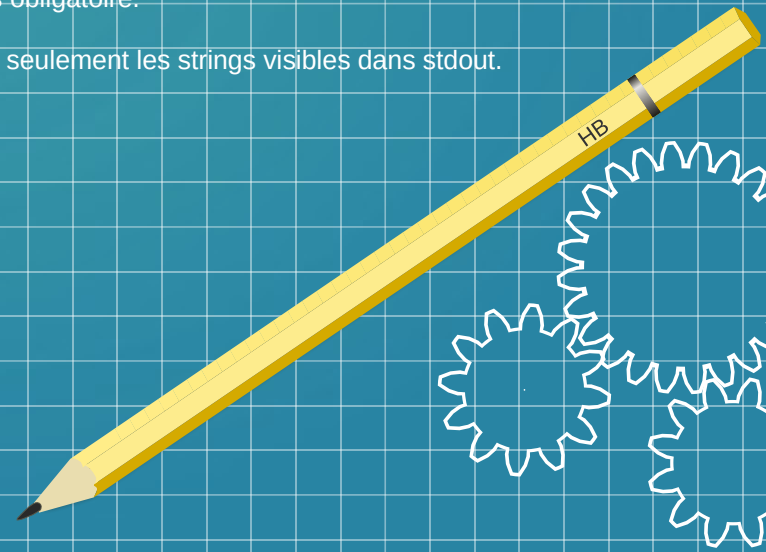
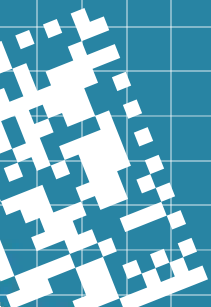
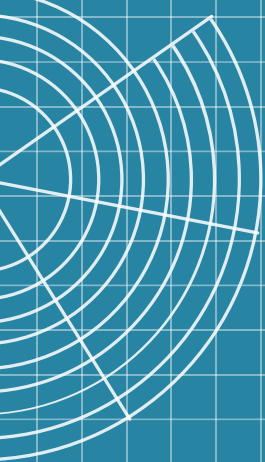
CONCLUSION

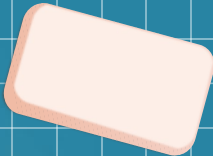
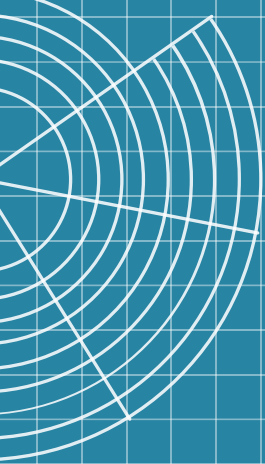
Points importants :

Le script ne lit jamais SECRET dans le binaire, il ne s'intéresse qu'à la sortie.

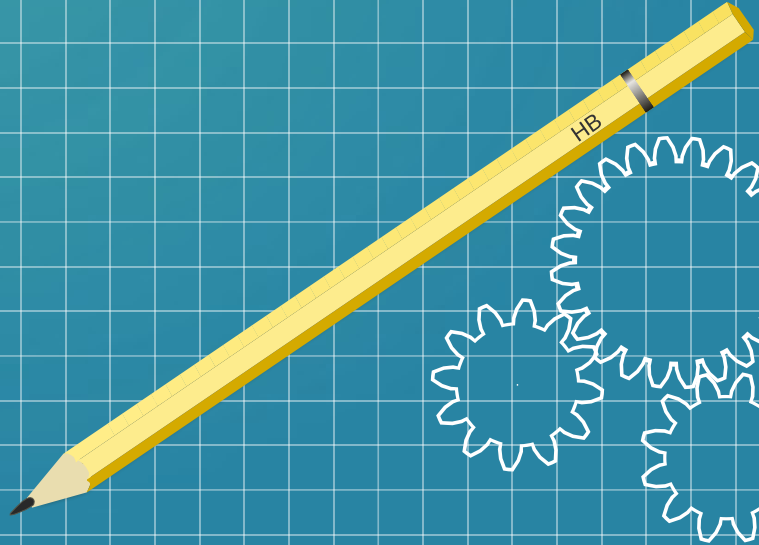
Tu peux ajuster PW_LEN si tu sais que le pass ne fait que 15 chars, mais ce n'est pas obligatoire.

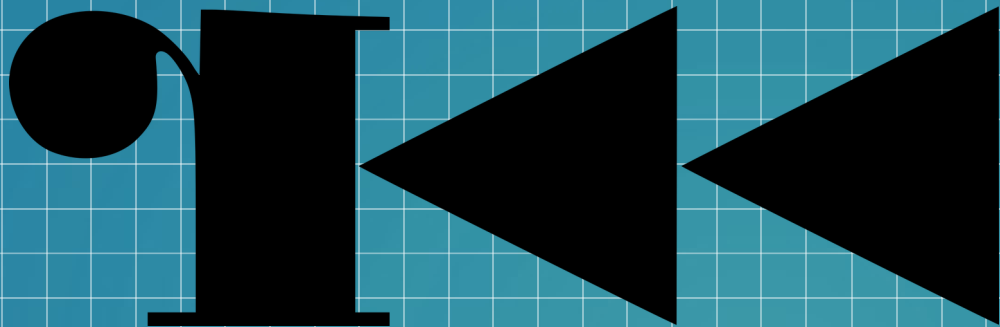
Même si le binaire est strip, on n'utilise pas d'adresses de fonctions ou de constantes, seulement les strings visibles dans stdout.





```
def is_successful(s):  
    out = s.posix.dumps(sys.stdout.fileno())  
    return b"Mot de passe correct" in out  
  
def should_abort(s):  
    out = s.posix.dumps(sys.stdout.fileno())  
    return b"Mot de passe incorrect" in out  
  
simgr.explore(find=is_successful, avoid=should_abort)  
  
if simgr.found:  
    found = simgr.found[0]  
    solution = found.solver.eval(pw, cast_to=bytes)  
    # Optionnel: enlever les \x00 de padding  
    solution = solution.split(b"\x00")[0]  
    print(solution)  
  
else:  
    print("No solution found")
```





→ REVERSE avec radare2 / cutter

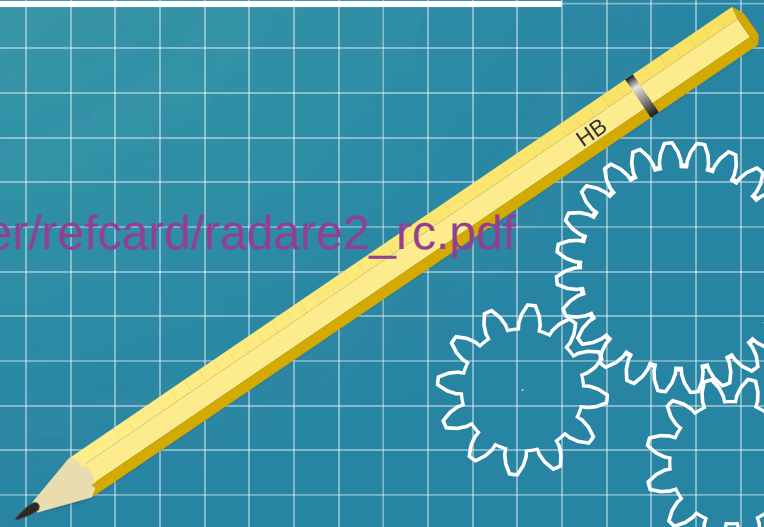
<https://book.rada.re/refcard/intro.htm>

|

https://github.com/radareorg/radare2-book/blob/master/refcard/radare2_rc.pdf

https://scoding.de/uploads/r2_cs.pdf

<https://rada.re/get/radare.pdf>



RADARE2

INSTALLATION

Sur Linux, tu as trois façons principales d'installer radare2 : sur une Debian/Ubuntu classique je te conseille soit le paquet, soit la version git.

1. Via le gestionnaire de paquets (simple)

Sur Debian/Ubuntu (et dérivées, y compris Kali) :

```
$ sudo apt update
$ sudo apt install radare2
```

C'est suffisant pour faire ce que tu fais déjà (r2 -A ./mypasswd).

2. Version la plus récente depuis le dépôt git (recommandé par le projet)

Pour avoir la dernière version (souvent mieux pour la doc et les features récentes) :

```
$ git clone https://github.com/radareorg/radare2
$ cd radare2
$ ./sys/install.sh
$ ./sys/install.sh va compiler et installer radare2 système-wide.
```

Pour mettre à jour plus tard, tu fais simplement :

```
$ cd radare2
$ git pull
$ ./sys/install.sh
```

3. Via Snap (si tu as snapd)

Si tu es sur Ubuntu avec snap activé :

```
$ sudo snap install radare2 --classic
```

Vu ton profil, je partirais plutôt sur la méthode git (2) pour avoir un r2 à jour et compatible avec les tutos / plugins récents (r2pm etc.).

RADARE2

CHARGEMENT DU BINAIRE

```
$ r2 -A ./mypasswd
```

Warning: run r2 with -e bin.cache=true to fix relocations in disassembly

[x] Analyze all flags starting with sym. and entry0 (aa)

[x] Analyze function calls (aac)

[x] Analyze len bytes of instructions for references (aar)

[x] Finding and parsing C++ vtables (avrr)

[x] Type matching analysis for all functions (aافت)

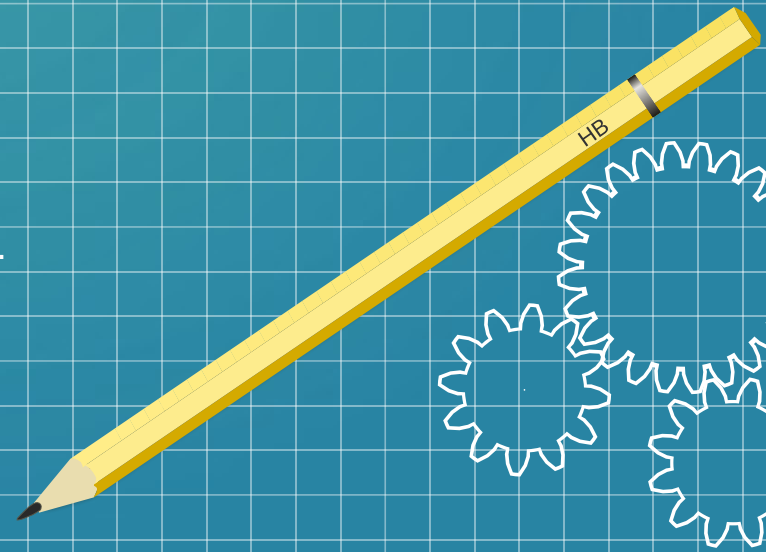
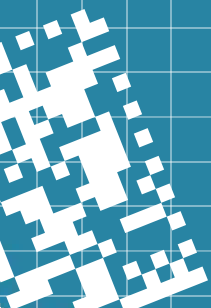
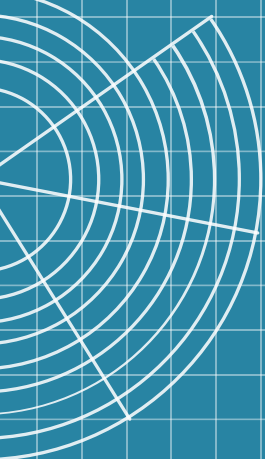
[x] Propagate noreturn information (aanr)

[x] Integrate dwarf function information.

[x] Use -AA or aaaa to perform additional experimental analysis.

```
[0x00001080]>
```

-A lance l'analyse auto (équivalent aaa).

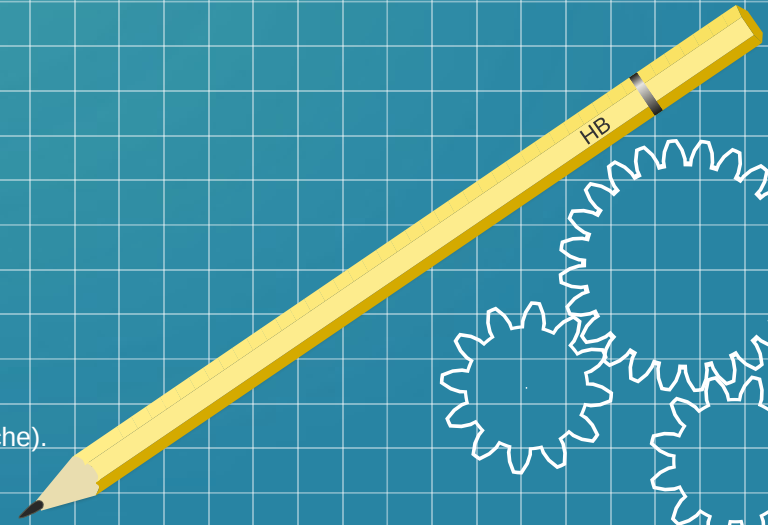
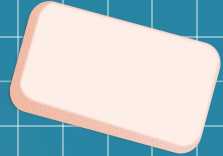


RADARE2

LISTER LES FONCTIONS

```
[0x00001080]> afl
0x00001080  1 38      entry0
0x000010b0  4 41  -> 34  sym.deregister_tm_clones
0x000010e0  4 57  -> 51  sym.register_tm_clones
0x00001120  5 57  -> 54  sym.__do_global_dtors_aux
0x00001050  1 10      fcn.00001050
0x00001160  1 9       entry.init0
0x00001224  1 13      sym._fini
0x00001199  7 138     dbg.main
0x00001169  1 48      dbg.check_password
0x00001070  1 10      sym.imp.strcmp
0x00001000  3 27      sym._init
0x00001060  1 10      sym.imp.printf
```

Cherche une fonction qui ressemble à main (souvent sym.main ou quelque chose de proche).



RADARE2

DUMP DU CODE L'ASSEMBLEUR

[0x00001080]> s dbg.main
[0x00001199]> pdf

Tu devrais voir quelque chose dans ce style (en assembleur, mais logique équivalente) :

test de argc < 2, sinon affichage de "Usage: %s\n",

appel à dbg.check_password(argv[1]),

selon le retour, printf("Mot de passe correct !") ou "Mot de passe incorrect !".

Note l'appel à dbg.check_password et l'argument passé (normalement argv[1]).

```
-- main:
; DATA XREF from entry0 @ 0x1098
138: int dbg.main (signed int64_t argc, char **argv, char **envp);
; var char **env @ rbp-0x18
; var char **var_10h @ rbp-0x10
; var signed int64_t var_4h @ rbp-0x4
; arg signed int64_t argc @ rdi
; arg char **argv @ rsi
; arg char **envp @ rdx
0x00001199  f30f1efa  endbr64                ; int main(int argc,char ** argv,char ** env);
0x0000119d  55        push rbp
0x0000119e  4889e5    mov rbp, rsp
0x000011a1  4883ec20  sub rsp, 0x20
0x000011a5  897dfc    mov dword [var_4h], edi ; argc
0x000011a8  488975f0  mov qword [var_10h], rsi ; argv
0x000011ac  48895e08  mov qword [env], rdx    ; envp
0x000011b0  837dfc01  cmp dword [var_4h], 1
0x000011b4  7f25     jg 0x11db
0x000011b6  488b45f0  mov rax, qword [var_10h]
0x000011ba  488b00    mov rax, qword [rax]
0x000011bd  4889c6    mov rsi, rax
0x000011c0  488d54d0e00. lea rax, str.Usage:~s__mot_de_passe_n ; 0x2014 ; "Usage: %s <mot_de_passe>\n"
0x000011c7  4889c7    mov rdi, rax             ; const char *format
0x000011ca  b800000000 mov eax, 0
0x000011cf  e88cfeffff call syn.imp.printf      ; int printf(const char *format)
0x000011d4  b801000000 mov eax, 1
0x000011d9  eb46     jmp 0x1221
; CODE XREF from dbg.main @ 0x11b4
0x000011db  488b45f0  mov rax, qword [var_10h]
0x000011df  4883c008  add rax, 8
0x000011e3  488b00    mov rax, qword [rax]
0x000011e6  4889c7    mov rdi, rax             ; char *arg1
0x000011e9  e87bffff call dbg.check_password
0x000011ee  85c0     test eax, eax
0x000011f0  7416     je 0x1208
0x000011f2  488d5350e00. lea rax, str.Mot_de_passe_correct__ ; 0x202e ; "Mot de passe correct !"
0x000011f9  4889c7    mov rdi, rax             ; const char *format
0x000011fc  b800000000 mov eax, 0
0x00001201  e85afeffff call syn.imp.printf      ; int printf(const char *format)
0x00001206  eb14     jmp 0x121c
; CODE XREF from dbg.main @ 0x11f0
0x00001208  488d5360e00. lea rax, str.Mot_de_passe_incorrect__ ; 0x2045 ; "Mot de passe incorrect !"
0x0000120f  4889c7    mov rdi, rax             ; const char *format
0x00001212  b800000000 mov eax, 0
0x00001217  e844feffff call syn.imp.printf      ; int printf(const char *format)
; CODE XREF from dbg.main @ 0x1206
0x0000121c  b800000000 mov eax, 0
; CODE XREF from dbg.main @ 0x11d9
0x00001221  c9       leave
0x00001222  c3       ret
```

RADARE2

INSPECTION DE CHECK_PASSWORD

[0x00001199]> s dbg.check_password

[0x00001169]> pdf

[0x00001199]> s dbg.check_password

[0x00001169]> pdf

```
    ;-- check_password:
    ; CALL XREF from dbg.main @ 0x11e9
48: dbg.check_password(char *arg1);
    ; var char const *password @ rbp-0x8
    ; arg char *arg1 @ rdi
0x00001169  f30f1efa      endbr64
0x0000116d  55            push rbp
0x0000116e  4889e5        mov rbp, rsp
0x00001171  4883ec10      sub rsp, 0x10
0x00001175  48897df8      mov qword [password], rdi ; arg1
0x00001179  488b45f8      mov rax, qword [password]
0x0000117d  488d15800e00. lea rdx, str.123456789AZERTY ; 0x2004 ; "123456789AZERTY"
0x00001184  4889d6        mov rsi, rdx ; const char *s2
0x00001187  4889c7        mov rdi, rax ; const char *s1
0x0000118a  e8e1feffff    call sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
0x0000118f  85c0          test eax, eax
0x00001191  0f94c0        sete al
0x00001194  0fb6c0        movzx eax, al
0x00001197  c9            leave
0x00001198  c3            ret
```

[0x00001169]> █

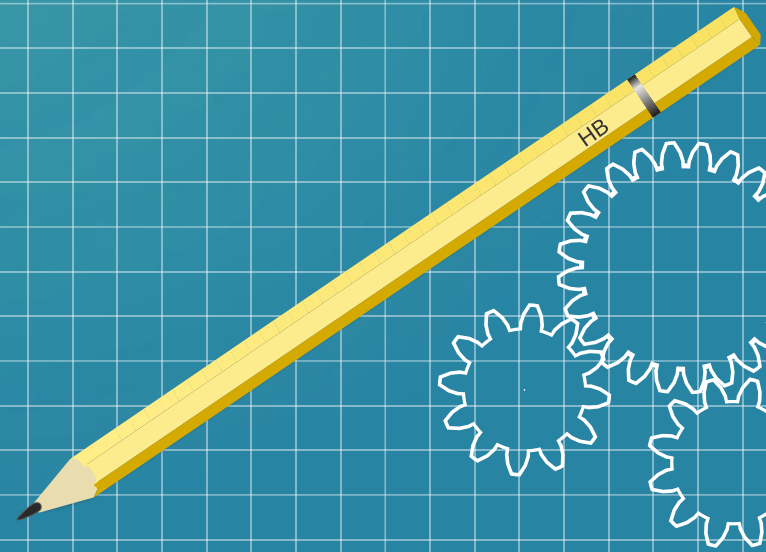
RADARE2

CHALLENGE #2



On va faire un “god-mode patch” sur `check_password`: l’idée est de remplacer son corps par “retourne toujours 0” (succès en C). En pratique, on va forcer le registre de retour (`eax` en 32 bits, `rax` en 64 bits) à 0, puis faire `ret`.

On va partir sur un binaire x86-64 ELF.



RADARE2

Ouvrir le binaire en écriture et trouver check_password

➤ `r2 -Aw ./mypasswd`

Warning: run r2 with `-e bin.cache=true` to fix relocations in disassembly

[x] Analyze all flags starting with sym. and entry0 (aa)

[x] Analyze function calls (aac)

[x] Analyze len.bytes of instructions for references (aar)

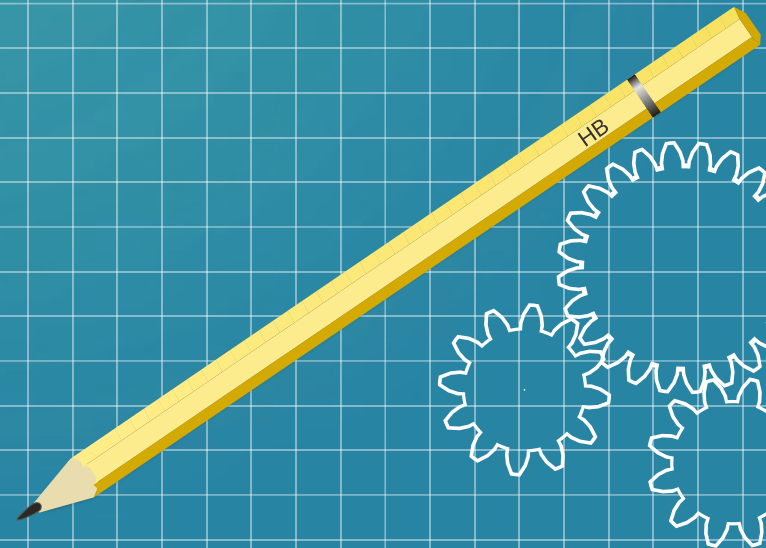
[x] Finding and parsing C++ vtables (avrr)

[x] Type matching analysis for all functions (aaft)

[x] Propagate noreturn information (aanr)

[x] Integrate dwarf function information.

[x] Use `-AA` or `aaaa` to perform additional experimental analysis.



RADARE2

Trouve et affiche check_password

```
[0x00001169]> afl | grep check_password
0x00001169      1 48      sym.check_password
```

```
# suppose que ça te donne, par ex.: 0x00001169      1 48      sym.check_password
[0x00001169]> s sym.check_password
```

```
[0x00001169]> pdf
; CALL XREF from main @ 0x11e9
- 48: sym.check_password(char *arg1);
; var char *s1 @ rbp-0x8
; arg char *arg1 @ rdi
0x00001169      f30f1efa      endbr64
0x0000116d      55              push rbp
0x0000116e      4889e5          mov rbp, rsp
0x00001171      4883ec10        sub rsp, 0x10
0x00001175      48897df8        mov qword [s1], rdi      ; arg1
0x00001179      488b45f8        mov rax, qword [s1]
0x0000117d      488d1580e00.    lea rdx, str.123456789AZERTY ; 0x2004 ; "123456789AZERTY"
0x00001184      4889d6          mov rsi, rdx             ; const char *s2
0x00001187      4889c7          mov rdi, rax             ; const char *s1
0x0000118a      e8e1feffff      call sym.imp.strcmp      ; int strcmp(const char *s1,
const char *s2)
0x0000118f      85c0            test eax, eax
0x00001191      0f94c0          sete al
0x00001194      0fb6c0          movzx eax, al
0x00001197      c9              leave
0x00001198      c3              ret
```

Tu devrais voir un prologue standard, puis un strcmp, puis un test eax,eax et un je/jne, etc.

RADARE2

Remplace complètement la fonction par :

```
xor eax, eax
inc eax
ret
```

Cela doit mettre 1 dans le registre eax et retourner la valeur

se placer au début de la fonction

[0x00001169]> s sym.check_password

écraser proprement par 3 instructions

[0x00001169]> wa xor eax, eax

[0x00001169]> wa inc eax

[0x00001169]> wa ret

vérifier

[0x00001169]> pdf

Sauvegarder le binaire

[0x00001169]> q

```
[0x00001169]> pdf
; CALL XREF from main @ 0x11e9
48: sym.check_password (char *arg1);
; var char *s1 @ rbp-0x8
; arg char *arg1 @ rdi
0x00001169 c3 ret

0x0000116d 55 push rbp
0x0000116e 4889e5 mov rbp, rsp
0x00001171 4883ec10 sub rsp, 0x10
0x00001175 48897df8 mov qword [s1], rdi ; arg1
0x00001179 488b45f8 mov rax, qword [s1]
0x0000117d 488d1580e000 lea rdx, str.123456789AZERTY ; 0x2004 ; "123456789AZERTY"
0x00001184 4889d6 mov rsi, rdx ; const char *s2
0x00001187 4889c7 mov rdi, rax ; const char *s1
0x0000118a e8e1feffff call sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
0x0000118f 85c0 test eax, eax
0x00001191 0f94c0 sete al
0x00001194 0fb6c0 movzx eax, al
0x00001197 c9 leave
0x00001198 c3 ret
```

RADARE2

test

Maintenant, quelque soit le mot de passe, il sera affiché qu'il est correct :

```
./mypasswd toto
```

```
./mypasswd 123
```

```
./mypasswd 123456789AZERTY
```

Tout doit afficher "Mot de passe correct !".

La nouvelle check_password en ASM donne :

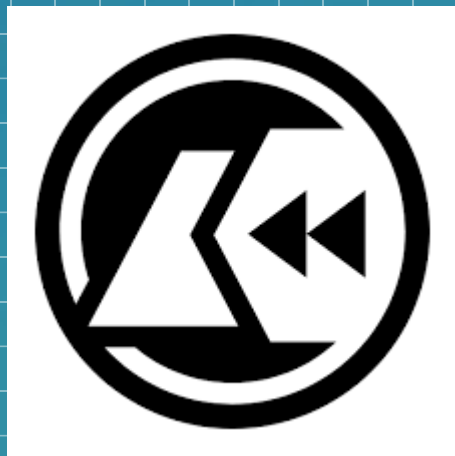
```
xor eax, eax    ; eax = 0
inc eax         ; eax = 1
ret             ; retourne 1
```

En C, ça correspond grosso modo àau code suivant:

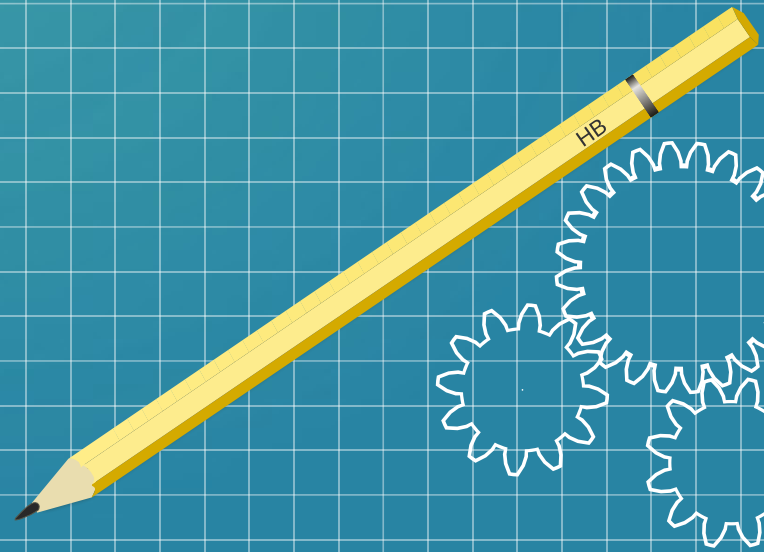
```
int check_password(const char *password) {
    return 1;
}
```

Quel que soit password, eax vaut 1 au moment du ret.

Pour main, ça simule le cas "mot de passe correct" tout le temps, puisque main s'attend à 1 pour succès



RIZIN CUTTER



RIZIN CUTTER

INSTALLATION

Installer Cutter sous Linux

Le plus simple pour tester rapidement :

Option 1 : AppImage officielle

Va sur la page Cutter (Rizin Cutter maintenant) : <https://cutter.re/>

Télécharge l'AppImage pour Linux (x86_64).

Dans ton terminal :

```
$ chmod a+x Cutter-v2.4.1-Linux-x86_64.AppImage  
$ ./Cutter-v2.4.1-Linux-x86_64.AppImage
```

C'est tout, pas d'installation système nécessaire.

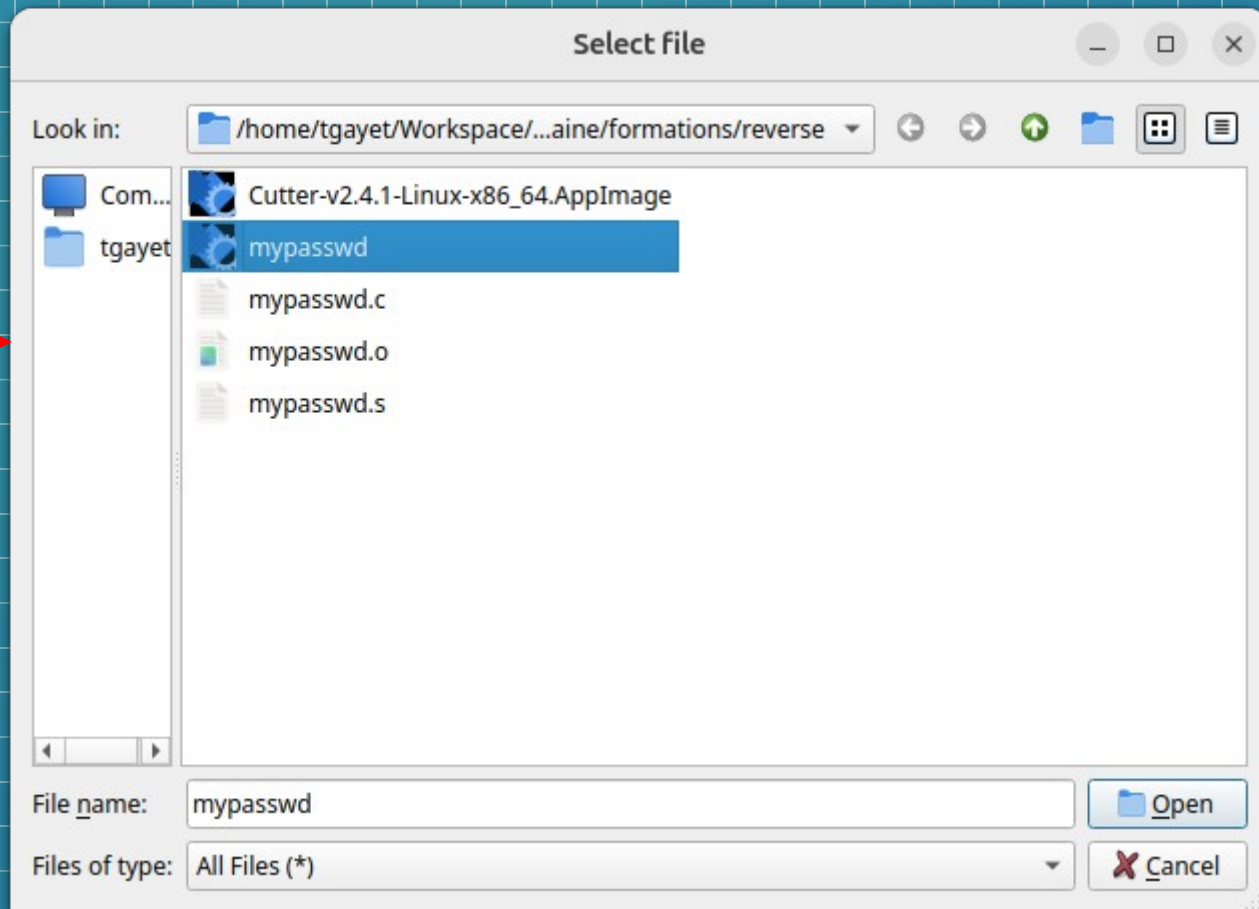
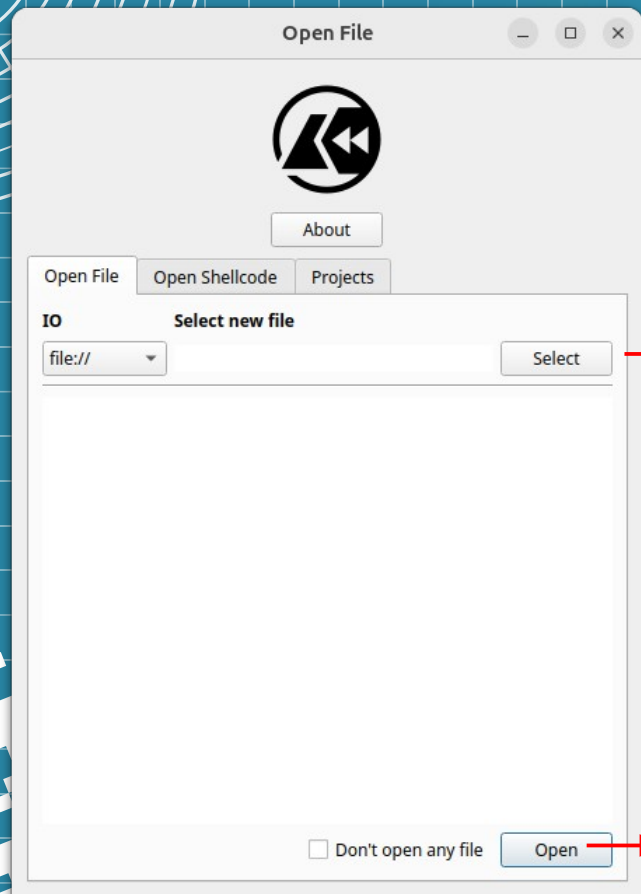
Option 2 : paquet distro

Sur Kali / Debian dérivées récentes il existe souvent rizin-cutter ou cutter :

```
$ sudo apt install rizin-cutter  
Sur Arch : $ paquet radare2-cutter ou rz-cutter.
```

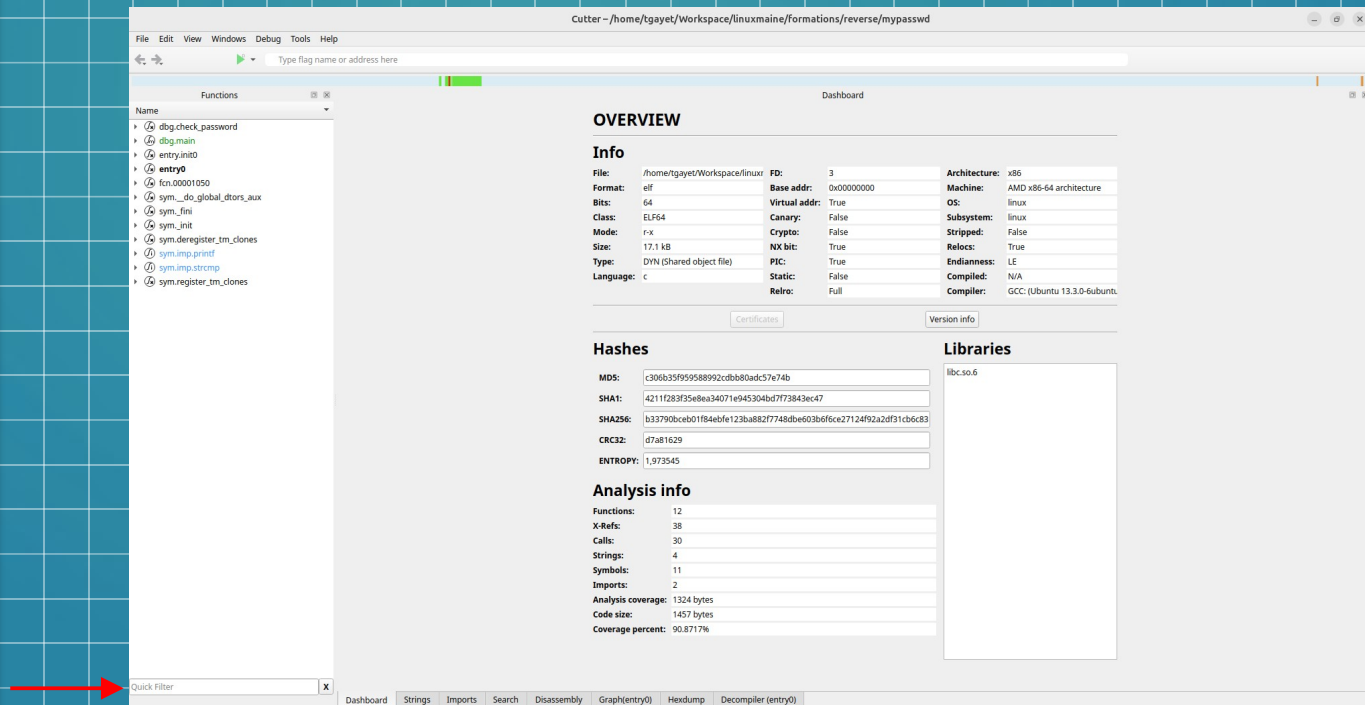
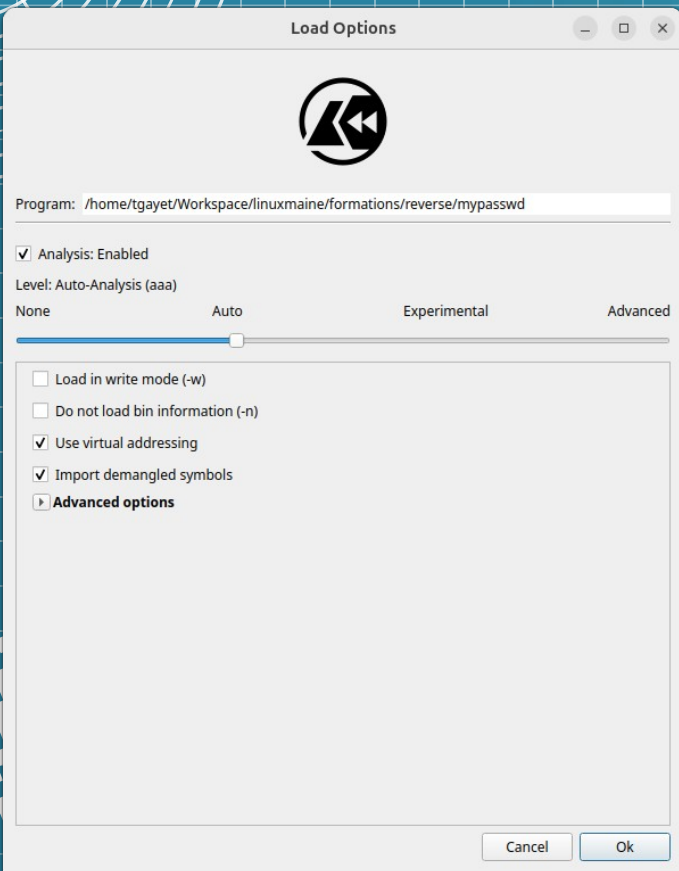
RIZIN CUTTER

OPEN BINARY



RIZIN CUTTER

OPEN BINARY





Type flag name or address here

Functions

Disassembly

Name

- dbg.check_password
- dbg.main
- entry.init0
- entry0
- fcn.00001050
- sym.__do_global_dtors_aux
- sym.fini
- sym.init
- sym.deregister_tm_clones
- sym.imp.printf
- sym.imp.strcmp
- sym.register_tm_clones

```

;-- main:
int main(int argc, char **argv, char **env)
; arg int argc @ stack - 0xc
; arg char **argv @ stack - 0x18
; arg char **env @ stack - 0x20
0x00001199      endbr64      ; mypasswd.c:12 {
0x0000119d      push     rbp
0x0000119e      mov     rbp, rsp
0x000011a1      sub     rsp, 0x20
0x000011a5      mov     dword [argc], edi ; argc
0x000011a8      mov     qword [argv], rsi ; argv
0x000011ac      mov     qword [env], rdx ; env
0x000011b0      cmp     dword [argc], 1 ; mypasswd.c:13 if (argc < 2) {
0x000011b4      jg      0x11db
0x000011b6      mov     rax, qword [argv] ; mypasswd.c:14 printf("Usage: %s <mot_de_passe>\n", argv[0]);
0x000011ba      mov     rax, qword [rax]
0x000011bd      mov     rsi, rax
0x000011c0      lea     rax, [str.Usage: __s__mot_de_passe] ; 0x2014
0x000011c7      mov     rdi, rax ; const char *format
0x000011ca      mov     eax, 0
0x000011cf      call    section..plt.sec ; sym.imp.printf ; int printf(const char *format)
0x000011d4      mov     eax, 1 ; mypasswd.c:15 return 1;
0x000011d9      jmp     0x1221
0x000011db      mov     rax, qword [argv] ; mypasswd.c:18 if (check_password(argv[1]))
0x000011df      add     rax, 8
0x000011e3      mov     rax, qword [rax]
0x000011e6      mov     rdi, rax ; const char *password
0x000011e9      call    dbg.check_password ; dbg.check_password ; int check_password(const char *password)
0x000011ee      test    eax, eax
0x000011f0      je      0x1208
0x000011f2      lea     rax, [str.Mot_de_passe_correct] ; mypasswd.c:20 printf("Mot de passe correct !"); ; 0x202e
0x000011f9      mov     rdi, rax ; const char *format
0x000011fc      mov     eax, 0
0x00001201      call    section..plt.sec ; sym.imp.printf ; int printf(const char *format)
0x00001206      jmp     0x121c
0x00001208      lea     rax, [str.Mot_de_passe_incorrect] ; mypasswd.c:22 printf("Mot de passe incorrect !"); ; 0x2045
0x0000120f      mov     rdi, rax ; const char *format
0x00001212      mov     eax, 0
0x00001217      call    section..plt.sec ; sym.imp.printf ; int printf(const char *format)
0x0000121c      mov     eax, 0 ; mypasswd.c:25 return 0;
0x00001221      leave   ; mypasswd.c:26 }
0x00001222      ret
0x00001223      add     bl, dh
;-- section..fini:
_fini():
0x00001224      endbr64      ; [16] -r-x section size 13 named .fini
0x00001228      sub     rsp, 8
0x0000122c      add     rsp, 8
0x00001230      ret
0x00001231      invalid
0x00001232      invalid
0x00001233      invalid
0x00001234      invalid

```

Quick Filter

X

Dashboard

Strings

Imports

Search

Disassembly

Graph(dbg.main)

Hexdump

Decompiler (entry0)



Type flag name or address here

Functions

Name

- dbg.check_password
- dbg.main
- entry.init0
- entry0
- fcn.00001050
- sym._do_global_dtors_aux
- sym._fini
- sym._init
- sym.deregister_tm_clones
- sym.imp.printf
- sym.imp.strcmp
- sym.register_tm_clones

Disassembly

```

int check_password(const char *password)
; arg const char *password @ stack - 0x10
0x00001169      endbr64      ; mypasswd.c:7 {
0x0000116d      push     rbp
0x0000116e      mov     rbp, rsp
0x00001171      sub     rsp, 0x10
0x00001175      mov     qword [password], rdi ; password
0x00001179      rax, qword [password] ; mypasswd.c:8 return strcmp(password, SECRET) == 0;
0x0000117d      lea     rdi, [str.123456789AZERTY] ; 0x2004
0x00001184      mov     rsi, rdx ; const char *s2
0x00001187      mov     rdi, rax ; const char *s1
0x0000118a      call    strcmp ; sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
0x0000118f      test    eax, eax
0x00001191      sete    al
0x00001194      movzx   eax, al
0x00001197      leave   ; mypasswd.c:9 }
0x00001198      ret

;-- main:
int main(int argc, char **argv, char **env)
; arg int argc @ stack - 0xc
; arg char **argv @ stack - 0x18
; arg char **env @ stack - 0x20
0x00001199      endbr64      ; mypasswd.c:12 {
0x0000119d      push     rbp
0x0000119e      mov     rbp, rsp
0x000011a1      sub     rsp, 0x20
0x000011a5      mov     dword [argc], edi ; argc
0x000011a8      mov     qword [argv], rsi ; argv
0x000011ac      mov     qword [env], rdx ; env
0x000011b0      cmp     dword [argc], 1 ; mypasswd.c:13 if (argc < 2) {
0x000011b4      jg      0x11db
0x000011b6      mov     rax, qword [argv] ; mypasswd.c:14 printf("Usage: %s <mot_de_passe>\n", argv[0]);
0x000011ba      mov     rax, qword [rax]
0x000011bd      mov     rsi, rax
0x000011c0      lea     rax, [str.Usage: __s__mot_de_passe] ; 0x2014
0x000011c7      mov     rdi, rax ; const char *format
0x000011ca      mov     eax, 0
0x000011cf      call    section..plt.sec ; sym.imp.printf ; int printf(const char *format)
0x000011d4      mov     eax, 1 ; mypasswd.c:15 return 1;
0x000011d9      jmp     0x1221
0x000011db      mov     rax, qword [argv] ; mypasswd.c:18 if (check_password(argv[1]))
0x000011df      add     rax, 8
0x000011e3      mov     rax, qword [rax]
0x000011e6      mov     rdi, rax ; const char *password
0x000011e9      call    dbg.check_password ; dbg.check_password ; int check_password(const char *password)
0x000011ee      test    eax, eax
0x000011f0      je      0x1208
0x000011f2      lea     rax, [str.Mot_de_passe_correct] ; mypasswd.c:20 printf("Mot de passe correct !"); ; 0x202e
0x000011f9      mov     rdi, rax ; const char *format
0x000011fc      mov     eax, 0
0x00001201      call    section..plt.sec ; sym.imp.printf ; int printf(const char *format)
0x00001206      jmp     0x121c
0x00001208      lea     rax, [str.Mot_de_passe_incorrect] ; mypasswd.c:22 printf("Mot de passe incorrect !"); ; 0x2045

```

Quick Filter

X



→ REVERSE avec gildra

<https://www.scribd.com/document/475260023/Ghidra-Cheat-Sheet>

https://ghidra.re/ghidra_docs/GhidraClass/Advanced/improvingDisassemblyAndDecompilation.pdf

<https://defkey.com/ghidra-10-2-3-shortcuts?Orientation=portrait&Font=arial&cellAlternateColor=%23f7e0a1&ShowPageNumber=true&pdf=True&ShowPageNumber=false>

GILDRA

INSTALLATION

```
$ sudo apt install openjdk-21-jdk
```

```
$ wget
```

```
https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra\_12.0.2\_build/ghidra\_12.0.2\_PUBLIC\_20260129.zip
```

```
$ unzip ghidra_12.0.2_PUBLIC_20260129.zip
```

```
$ cd ghidra_12.0.2_PUBLIC/
```

```
$ ./ghidraRun
```

Ghidra User Agreement

Licensed under the Apache License, Version 2.0 (the "License"); Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

As a software reverse engineering (SRE) framework, Ghidra is designed solely to facilitate lawful SRE activities. You should always ensure that any SRE activities in which you engage are permissible as computer software may be protected under governing law (e.g., copyright) or under an applicable licensing agreement. In making Ghidra available for public use, the National Security Agency does not condone or encourage any improper usage of Ghidra. Consistent with the Apache 2.0 license under which Ghidra has been made available, you are solely responsible for determining the appropriateness of using or redistributing Ghidra.

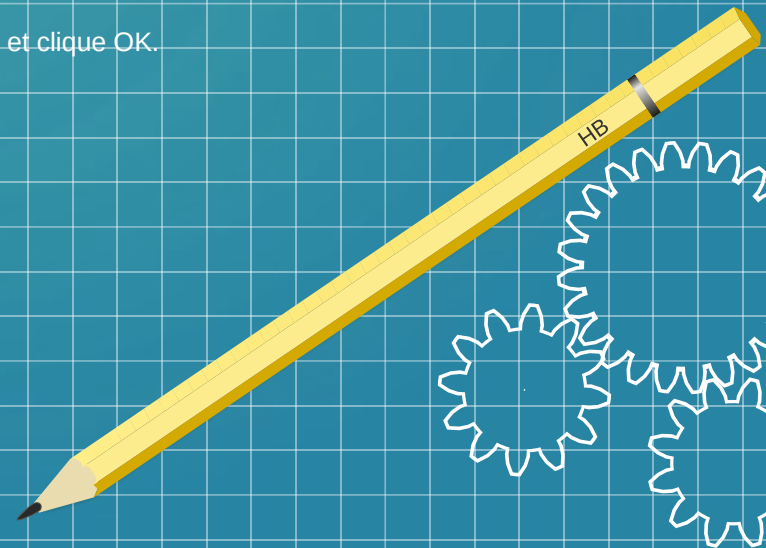
GILDRA

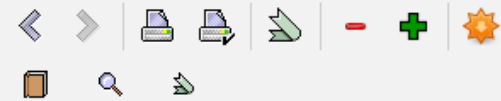
CHARGEMENT



Charger ton binaire mypasswd dans Ghidra

- Démarre Ghidra → crée un nouveau projet non partagé.
- Menu File → Import File... → sélectionne ./mypasswd.
- Ghidra va détecter automatiquement ELF x86-64 ou 32-bit (selon ta compilation).
- Valide l'import, puis double-clique sur le binaire dans la fenêtre de projet pour l'ouvrir.
- Au premier open, Ghidra propose Analyze → laisse coché l'analyse par défaut et clique OK.





- Welcome to Help
 - Introduction
- > Getting Started
- > Ghidra Projects
- > Programs
- > Tools
- > Ghidra Functionality
- > Support
 - Keyboard Navigation
 - Undo/Redo
 - Glossary
 - What's New
 - Tips of the Day
- > Theming
- > Appendix

Ghidra: NSA Reverse Engineering Software

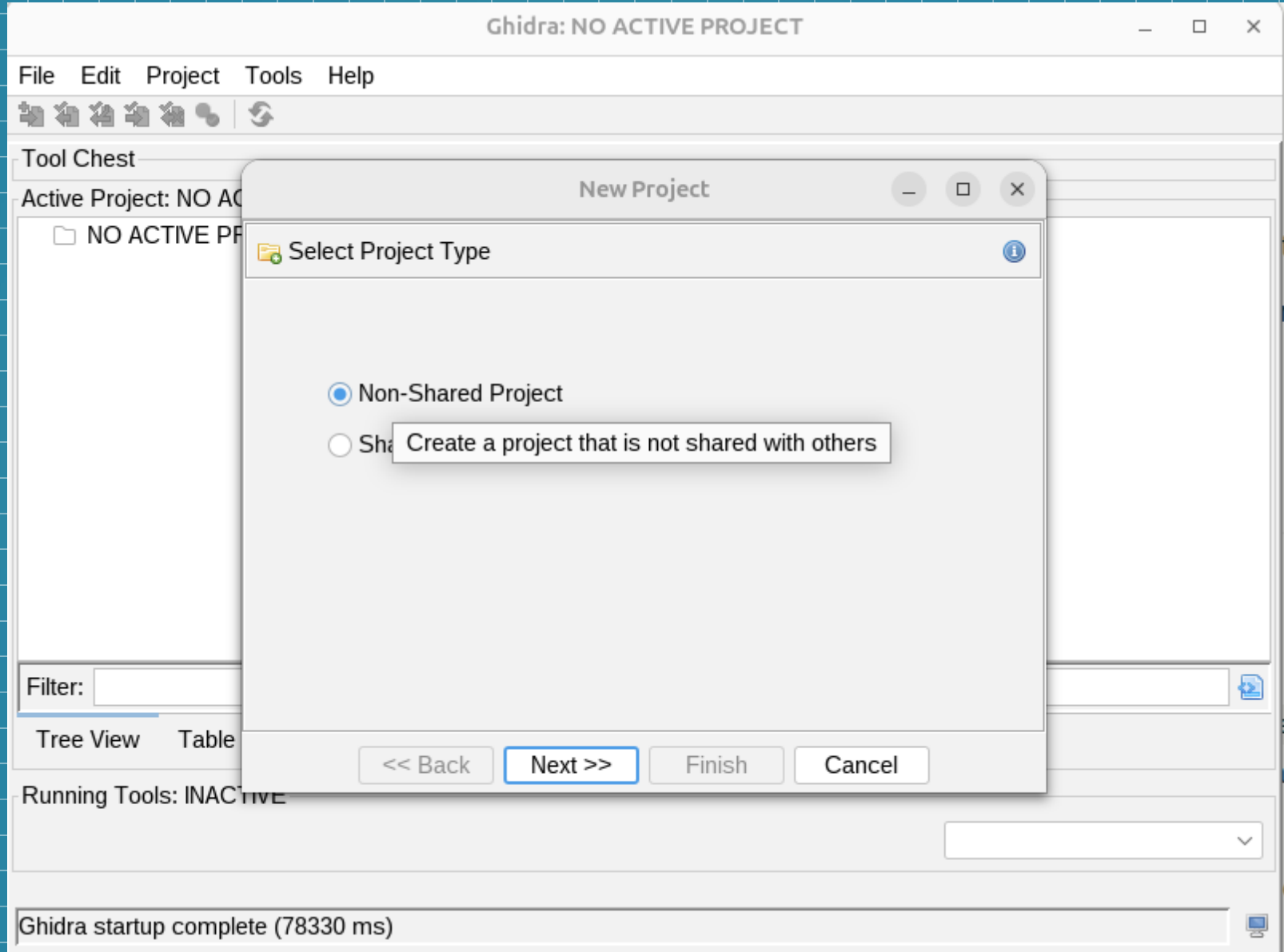
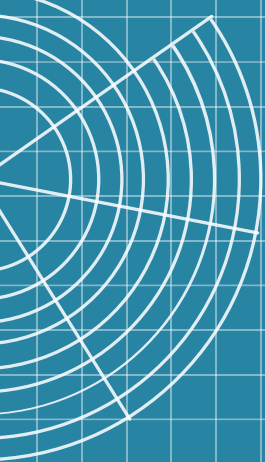
Ghidra is a software reverse engineering (SRE) framework developed by NSA's Research Directorate. This framework includes a suite of full-featured, high-end software analysis tools that enable users to analyze compiled code on a variety of platforms including Windows, MacOS, and Linux. Capabilities include disassembly, assembly, decompilation, debugging, emulation, graphing, and scripting, along with hundreds of other features. Ghidra supports a wide variety of processor instruction sets and executable formats and can be run in both user-interactive and automated modes. Users may also develop their own Ghidra plug-in components and/or scripts using the exposed API. In addition, there are numerous ways to extend Ghidra such as new processors, loaders/exporters, automated analyzers, and new visualizations.

In support of NSA's Cybersecurity mission, Ghidra was built to solve scaling and teaming problems on complex SRE efforts and to provide a customizable and extensible SRE research platform. NSA has applied Ghidra SRE capabilities to a variety of problems that involve analyzing malicious code and generating deep insights for NSA analysts who seek a better understanding of potential vulnerabilities in networks and systems.

What's New in Ghidra 12.0

This release includes new features, enhancements, performance improvements, quite a few bug fixes, and many pull-request contributions. Thanks to all those who have contributed their time, thoughts, and code. The Ghidra user community thanks you too!

The not-so-fine print: [Please Read!](#)



Import Results Summary

i

```

Project File Name:      mypasswd
Last Modified:         Fri Feb 06 20:15:38 CET 2026
ReadOnly:              false
Program Name:          mypasswd
Language ID:           x86:LE:64:default (4.6)
Compiler ID:           gcc
Processor:             x86
Endian:               Little
Address Size:          64
Minimum Address:       00100000
Maximum Address:       _elfSectionHeaders::0000093f
# of Bytes:            8448
# of Memory Blocks:    39
# of Instructions:     12
# of Defined Data:     120
# of Functions:        17
# of Symbols:          51
# of Data Types:       38
# of Data Type Categories: 2
Created With Ghidra Version: 12.0.2
Date Created:          Fri Feb 06 20:15:37 CET 2026
ELF File Type:         shared object
ELF GNU Program Prop[processor opt 0xc0000002]: 03 00 00 00
ELF GNU Program Prop[processor opt 0xc0008002]: 01 00 00 00
ELF Note[GNU BuildId]: c5d069f3c874a697c1a6e55788509fdf7d1563cd
ELF Note[required kernel ABI]: Linux 3.2.0
ELF Original Image Base: 0x0
ELF Prelinked:         false
ELF Source File [ 0]:  Scrt1.o
ELF Source File [ 1]:  crtstuff.c
ELF Source File [ 2]:  mypasswd.c
ELF Source File [ 3]:  crtstuff.c
ELF Source File [ 4]:
Elf Comment[0]:        GCC: (Ubuntu 13.3.0-6ubuntu2-24.04) 13.3.0
Executable Format:     Executable and Linking Format (ELF)
Executable Location:   /home/tgayet/Workspace/linuxmaine/formations/reverse/mypassw
Executable MD5:        c306b35f959588992cddb80adc57e74b
Executable SHA256:     b33790bceb01f84ebfe123ba882f7748dbe603b6f6ce27124f92a2df31c
FSRL:                  file:///home/tgayet/Workspace/linuxmaine/formations/reverse
Preferred Root Namespace Category:
Relocatable:          true
  
```

Additional Information

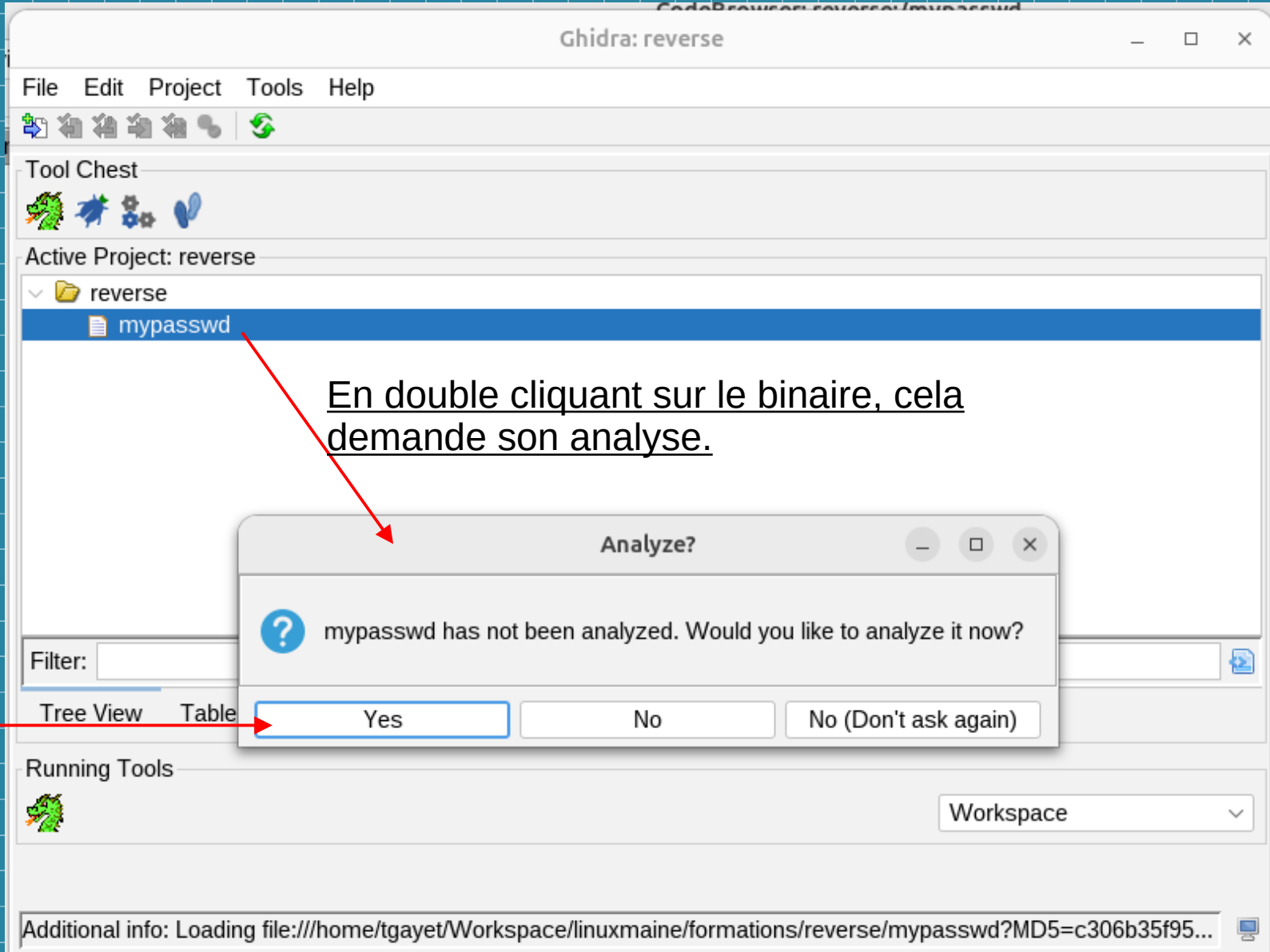
```

Loading file:///home/tgayet/Workspace/linuxmaine/formations/reverse/mypasswd?MD5=c306b35f959588992cddb80adc57e74b
Setting block .init_array to read-only
Setting block .fini_array to read-only
Setting block .dynamic to read-only
  
```

```

Linking the External Programs of 'mypasswd' to imported libraries...
[libc.so.6] -> not found in project
  
```

OK



Analysis Options

Analyzers

Enab...	Analyzer
☐	Aggressive Instruction Finder (Prototype)
☒	Apply Data Archives
☒	ASCII Strings
☒	Call Convention ID
☒	Call-Fixup Installer
☐	Condense Filler Bytes (Prototype)
☒	Create Address Tables
☒	Data Reference
☐	Decompiler Parameter ID
☒	Decompiler Switch Analysis
☒	Demangler GNU
☒	Disassemble Entry Points
☒	DWARF
☒	ELF Scalar Operand References
☒	Embedded Media
☒	External Entry References
☒	External Symbol Resolver

Select All

Deselect All

Reset

Save...

Standard Defaults



Delete

Description

Options

Analyze

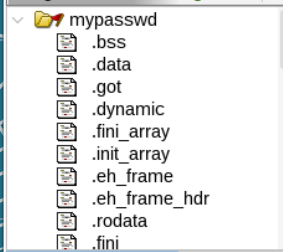
Cancel

Apply

File Edit Analysis BSim Graph Navigation Search Select Tools Window Help

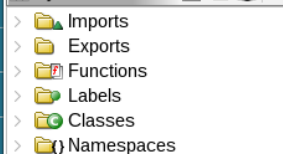


Program Trees



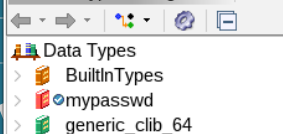
Program Tree x DWA > ^

Symbol Tree



Filter:

Data Type Manager



Filter:

Listing: mypasswd

```
00101179 48 8b 45 f8 MOV RAX,qword ptr [RBP + password_local]
0010117d 48 8d 15 LEA RDX,[s_123456789AZERTY_00102004] = "123456789AZERTY"
00101184 48 89 d6 MOV RSI=>s_123456789AZERTY_00102004,RDX = "123456789AZERTY"
00101187 48 89 c7 MOV password,RAX
0010118a e8 e1 fe CALL <EXTERNAL>::strcmp int strcmp(char * __s1, c
ff ff

mypasswd.c:8 (8)
0010118f 85 c0 TEST EAX,EAX
00101191 0f 94 c0 SETZ AL
00101194 0f b6 c0 MOVZX EAX,AL

mypasswd.c:9 (2)
00101197 c9 LEAVE
00101198 c3 RET

*****
* FUNCTION
*****
int __stdcall main(int argc, char * * argv, char * * env)
EAX:4 <RETURN>
EDI:4 argc
RSI:8 argv
RDX:8 env
int Stack[-0xc]:4 argc_local XREF[2]: 001011a5(W),
001011b0(R)
char * * Stack[-0x18]:8 argv_local XREF[3]: 001011a8(W),
001011b6(R),
001011db(R)
char * * Stack[-0x20]:8 env_local XREF[1]: 001011ac(W)
main XREF[4]: Entry Point(*),
_start:00101098(*), 00102094
00102150(*)

mypasswd.c:12 (23)
00101199 f3 0f 1e fa ENDBR64
0010119d 55 PUSH RBP
0010119e 48 89 e5 MOV RBP,RSP
001011a1 48 83 ec 70 SUB RSP,RBP
```

Decompile: main - (mypass...

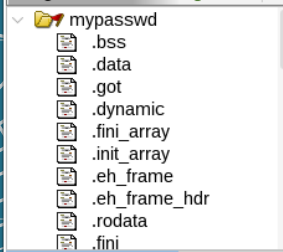
```
1
2 int main(int argc, char **argv, char **env)
3
4 {
5     int iVar1;
6     char **env_local;
7     char **argv_local;
8     int argc_local;
9
10    if (argc < 2) {
11        printf("Usage: %s <mot_de_passe>\n",*argv);
12        iVar1 = 1;
13    }
14    else {
15        iVar1 = check_password(argv[1]);
16        if (iVar1 == 0) {
17            printf("Mot de passe incorrect !");
18        }
19        else {
20            printf("Mot de passe correct !");
21        }
22        iVar1 = 0;
23    }
24    return iVar1;
25 }
26
```

Console - Scripting

File Edit Analysis BSim Graph Navigation Search Select Tools Window Help

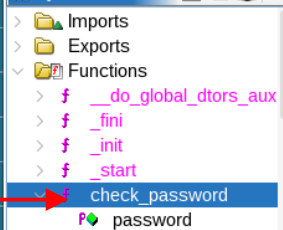


Program Trees



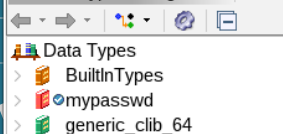
Program Tree x DWA > ^

Symbol Tree



Filter:

Data Type Manager



Filter:

Listing: mypasswd

```
00101160 f3 0f 1e fa ENDBR64
00101164 e9 77 ff JMP register_tm_clones
ff ff

-- Flow Override: CALL_RETURN (CALL_TERMINATOR)

*****
* FUNCTION
*****

int __stdcall check_password(char * password)
EAX:4 <RETURN>
RDI:8 password
Stack[-0x10]:8 password_local

check_password
XREF[2]: Entry Point(*), 00103db0(*)

XREF[4]: Entry Point(*), main:001011e
0010208c, 00102130(*)

mypasswd.c:7 (16)
00101169 f3 0f 1e fa ENDBR64
0010116d 55 PUSH RBP
0010116e 48 89 e5 MOV RBP,RSP
00101171 48 83 ec 10 SUB RSP,0x10
00101175 48 89 7d f8 MOV qword ptr [RBP + password_local],password

mypasswd.c:8 (22)
00101179 48 8b 45 f8 MOV RAX,qword ptr [RBP + password_local]
0010117d 48 8d 15 LEA RDX,[s_123456789AZERTY_00102004] = "123456789AZERTY"
80 0e 00 00
00101184 48 89 d6 MOV RSI=>s_123456789AZERTY_00102004,RDX = "123456789AZERTY"
00101187 48 89 c7 MOV password,RAX
0010118a e8 e1 fe CALL <EXTERNAL>::strcmp int strcmp(char * __s1, c
ff ff

mypasswd.c:8 (8)
0010118f 85 c0 TEST EAX,EAX
00101191 0f 94 c0 SETZ AL
00101194 0f b6 c0 MOVZX EAX,AL

mypasswd.c:9 (2)
00101197 c9 LEAVE
00101198 c3 RET
```

Decompile: check_password ...

```
1
2 int check_password(char *password)
3
4 {
5     int iVar1;
6     char *password_local;
7
8     iVar1 = strcmp(password,"123456789AZERTY");
9     return (int)(iVar1 == 0);
10 }
11
```

Console - Scripting

00101169

check_password

ENDBR64



→ REVERSE avec boomerang

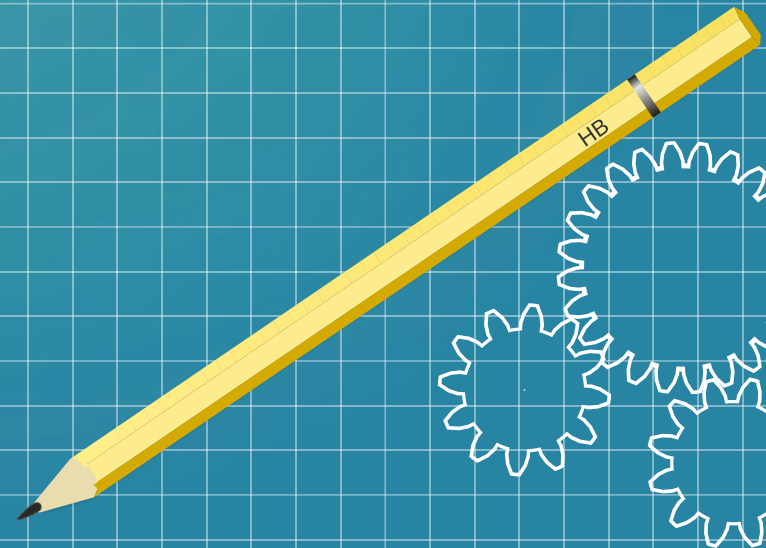
(Decompilateur C)

BOOMERANG



This is a fork of **Boomerang**, a general, open source (BSD licensed) machine code decompiler. Boomerang currently supports:

- architectures: x86 (IA-32 only), PPC, ST20
- file formats: ELF, PE, DOS MZ, DOS/4GW LE, Mach-O
- high-level language output: C
- Site officiel : <https://github.com/BoomerangDecompiler/boomerang/>
- <https://boomerang.sourceforge.net/>



BOOMERANG

```
$ wget https://github.com/BoomerangDecompiler/boomerang/releases/download/v0.5.2/boomerang-v0.5.2-linux64.tar.gz
$ tar zxvf boomerang-v0.5.2-linux64.tar.gz
$ cd boomerang-v0.5.2-linux64/bin/
$ export LD_LIBRARY_PATH=$PWD/../../lib/boomerang/
$ ./boomerang-gui
```

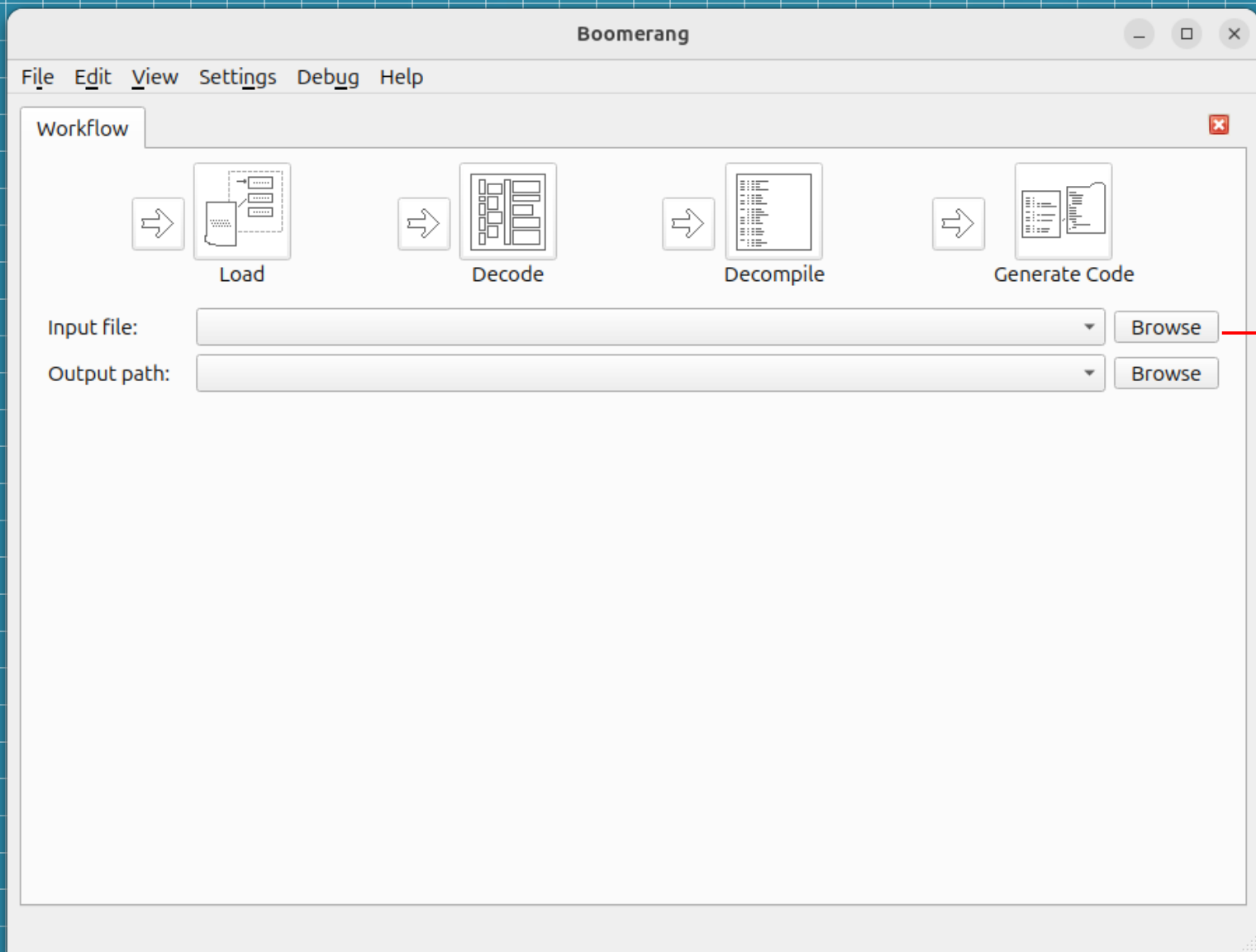
ou

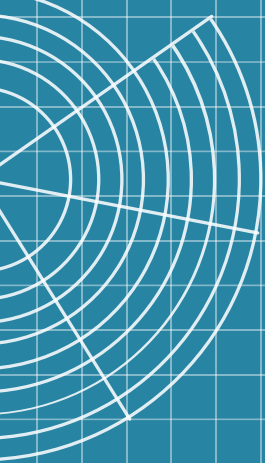
```
$ sudo apt-get install git build-essential cmake qt5-default libcapstone-dev flex bison qtbase5-dev qt5-qmake qtbase5-dev-tools gcc-multilib
$ cd ~/src # ou le répertoire de dev que tu veux
git clone https://github.com/BoomerangDecompiler/boomerang.git
cd boomerang
mkdir build && cd build
cmake ..
make -j"$(nproc)"
sudo make install
```

\$

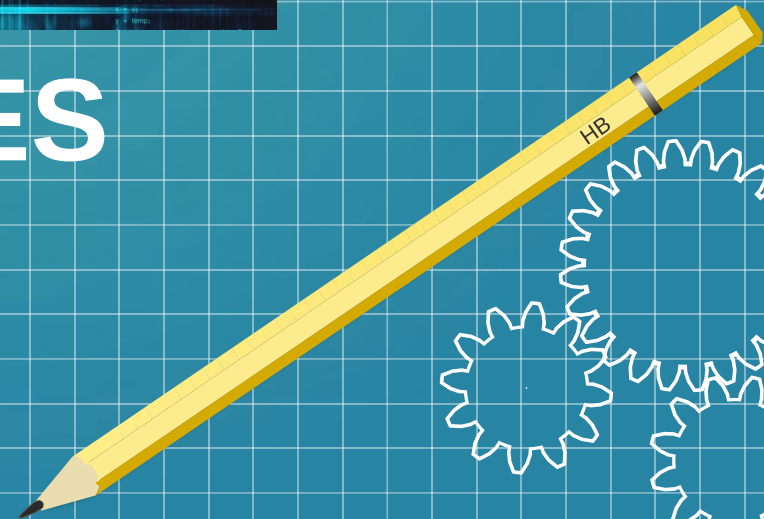
Ou

```
$ sudo apt-get install git build-essential cmake qt5-default libcapstone-dev flex bison
$ cd ~/src # ou le répertoire de dev que tu veux
$ git clone https://github.com/BoomerangDecompiler/boomerang.git
$ cd boomerang
$ mkdir build && cd build
$ cmake ..
$ make -j"$(nproc)"
$ sudo make install
```





VI. ANNEXES



GDB / DDD



GDB / DDD



Sites officiels

GDB Documentation (GNU) — le manuel officiel : <https://sourceware.org/gdb/current/onlinedocs/gdb/>

DDD (Data Display Debugger, GUI pour GDB) : info et docs sur les pages des distributions Linux (pas de site central dédié)



Cheatsheets & PDF

GDB Cheat Sheet (PDF) — référence rapide commandes GDB :
<https://cs.brown.edu/courses/cs033/docs/guides/gdb.pdf>

Autre cheat sheet GDB en PDF : <https://www.cristal.univ-lille.fr/~marquet/ens/cu/doc/gdb-refcard-a4.ps>



Tutoriels & références

Manuel GDB complet (parties interactives & scripting) — <https://sourceware.org/gdb/current/onlinedocs/gdb/>

Tutoriel GDB (PDF) pour débiter — https://perso.ens-lyon.fr/daniel.hirschkoft/C_Cam1/docs/doc_gdb.pdf

ANGR



Sites officiels

Documentation officielle d'angr : <https://docs.angr.io/>

Cheatsheets & PDF

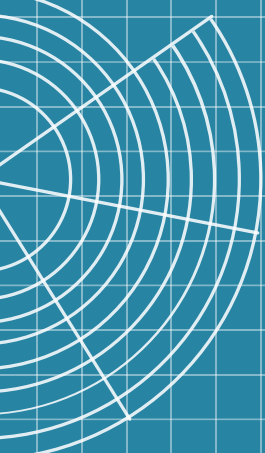
angr Cheatsheet (commande / API) dans la doc officielle :
<https://docs.angr.io/en/latest/appendix/cheatsheet.html>

Tutoriels & références

Documentation complète avec exemples Python : <https://docs.angr.io/>

Beaucoup de tutoriels pratiques sont disponibles sur GitHub et blogs (recherche "angr tutorial")

RADARE2



⚙️ radare2

📌 **Sites officiels**

Site officiel radare2 (infos & liens) : <https://radare.org/n/radare2.html>

GitHub radare2 : <https://github.com/radareorg/radare2>

📄 **Cheatsheets & PDF**

Radare2 Cheatsheet (PDF) — résumé commandes de base : https://scoding.de/uploads/r2_cs.pdf

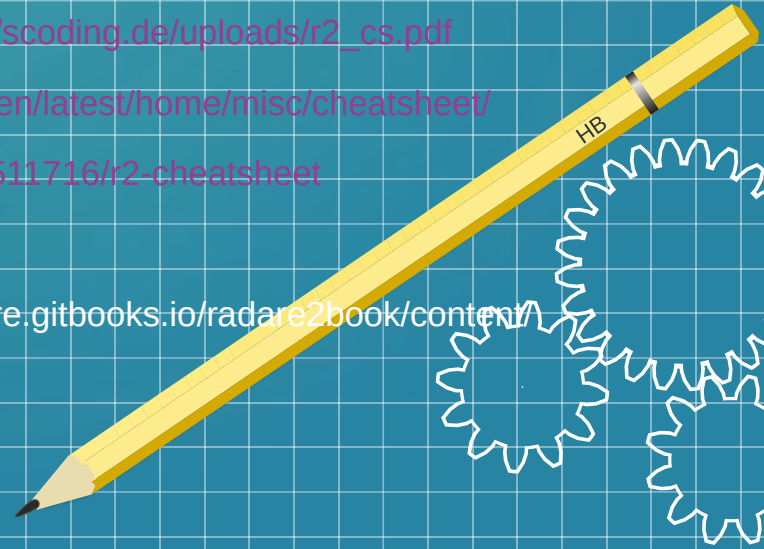
R2 Wiki avec liste commandes (texte) : <https://r2wiki.readthedocs.io/en/latest/home/misc/cheatsheet/>

r2 Cheatsheet Scribd (PDF) : <https://www.scribd.com/document/371511716/r2-cheatsheet>

📖 **Tutoriels & références**

The Official Radare2 Book (documentation complète) — <https://radare.gitbooks.io/radare2book/content/>

GitHub tutoriel radare2 : <https://github.com/0xd1ng/radare2-tutorial>



CUTTER



Cutter



Sites officiels

Site officiel Cutter (page projet) : <https://cutter.re/>

GitHub Cutter (code & releases) : <https://github.com/radareorg/cutter>



Cheatsheets & PDF

Pas de refcard officielle PDF unique, la documentation inclut souvent les commandes clés dans les tutoriels du site.



Tutoriels & références

Beaucoup de guides disponibles via GitHub, blogs et vidéos en ligne (recherche “Cutter tutorial”)

GILDRA

Ghidra

Sites officiels

Site officiel Ghidra (téléchargement & docs) : <https://ghidra-sre.org/>

GitHub Ghidra : <https://github.com/NationalSecurityAgency/ghidra>

Cheatsheets & PDF

Ghidra Cheat Sheet (PDF) (clavier & navigation) téléchargeable sur des sites de partage de docs : <https://fr.scribd.com/document/475260023/Ghidra-Cheat-Sheet>

Tutoriels universitaires Ghidra (PDF slides d'exercices) : <https://ligerlabs.org/lectures.html> (PDF guidés)

Tutoriels & références

Ghidra Tutorial (PDF) de cours universitaires : <https://kapravelos.com/teaching/csc405-s22/lectures/ghidra-tutorial.pdf>

Documentation et guides intégrés avec exemples et scripts (Java / Python) — <https://ghidra-sre.org/>

The Ghidra Book (guide complet PDF non officiel mais très utile) — <https://themalevolentone1.github.io/My-Notes-Collection/.../The%20Ghidra%20Book%20The%20Definitive%20Guide.pdf>

Résumé des liens officiels par outil

Outil	Site officiel	Refcard / Cheatsheet	Docs / Tutoriels
GDB	https://sourceware.org/gdb/current/onlinedocs/gdb/ ↗ 0x1ris.pages.dev	PDF cheat sheet (Brown CS) cs.brown.edu	GDB docs + PDF tutoriels 0x1ris.pages.dev
DDD	(via GNU / Linux packages)	Généralement intégré aux tutoriels GDB	docs Linux / man ddd
angr	https://docs.angr.io/ ↗ docs.angr.io	angr cheatsheet (docs) docs.angr.io	Docs officiels & exemples docs.angr.io
radare2	https://radare.org/n/radare2.html ↗ radare.org	radare2 PDF cheatsheet scoding.de	Official Radare2 Book 0x1ris.pages.dev
Cutter	https://cutter.re/ ↗ IHA089	Documentation en ligne	Tutoriels GitHub/vidéo IHA089
Ghidra	https://ghidra-sre.org/ ↗ Wikipedia	Ghidra cheat sheet PDF	Ghidra tutorial PDFs





DES QUESTIONS ?