La rétro-ingénierie

CHAPITRE 2 – 03/11/2025


Merci de ne pas enregistrer ni diffuser le contenu du cours à l'extérieur de la classe.

Objectifs chapitre 2

- Rappels
- Rétro-ingéniérie statique
  - Définition
  - Outils
- Rétro-ingénierie dynamique
  - Définition
  - Outils
- Etude de mini-cas statique + dynamique
  - Le crackme
  - Le stack frame
  - Les appels systèmes

Rappels

- Notions abordées pendant le chapitre 1
  - Généralités sur l’écosystème de la rétro-ingénierie
  - L’ Application Binary Interface (ABI)
  - Jeux d’instructions ARM / x86
  - Saut conditionnels
  - Type de données (signed / unsigned)
  - Appels de fonctions
  - Les formats ELF / PE

Rappels – TP1

- ex1 : compilation d'un programme simple dans divers architectures / formats
  - concepts: compilation croisée / ABI / format / architecture
  - outils: hexdump / bvi / objdump / docker / aarch64-linux-gnu-gcc / gcc
- ex2: observation de binaires hétérogènes
  - concepts: idem que ex1 / code natif / code manage
  - outils: idem que ex1 / file
- ex3: Assembler / observer / patcher
  - concepts: compilation / désassemblage / modification des opcodes / boucle infinies
  - outils: jwasm / nasm / objdump / bvi
- ex4: crackme trivial
  - concepts: méthodologie / passage d'arguments
  - outils : objdump / bvi
- ex5: TP2

Le reverse statique

101

Le reverse statique

Définition

On parle de rétro-ingénierie statique lorsqu'on regarde un programme exécutable en tant que fichier brute, comme une image ou un fichier texte.

Des outils existent pour découvrir le format, parser les instructions, les sections, et décoder toutes les instructions : ce sont les désassembleurs

Des outils existent pour tenter de reconstituer le code source origine : ce sont des décompilateurs.

Le reverse statique - méthodo

Trouver les caractéristiques du fichier qu'on regarde:

- format

- architecture cible / jeux d'instructions

- langage lorsque c'est possible

- Est-ce du code natif ou managé ?

- En déduire les outils à utiliser.

Le reverse statique – élaborer une stratégie

Regarder les chaînes de caractères est toujours une bonne idée

Regarder les fonctions exportées

Regarder les fonctions importées

Ensuite ça depend de la surface à étudier

Ex1: reverser un protocole réseau

Ex2: reverser un lanceur de programme

Ex3: reverser un algo

Le reverse statique – les outils

- objdump / ndisasm
- IDA / ghidra (on utilisera la version idafree dans ce cours)
  - -> Il vous faudra installer IDAFREE sur linux pour le TP2 !
  - -> Les fichiers sont disponibles sur moodle
- jadx-gui
- dnspy
- dis / uncompyle6 / decompyle3

Niveaux de représentation

Un programme informatique peut se voir de différentes façons:

- une succession d’instructions linéaires
- des appels de fonctions successifs
- un flux d’exécution
- un flux de données

Niveaux de représentation

Un programme informatique peut se voir de différentes façons:

- une succession d'instructions linéaires

- des appels de fonctions successifs

- un flux d'exécution

- un flux de données

Niveaux de représentation : Le graphe d’appel

Un programme informatique peut se voir de différentes façons:

- une succession d’instructions linéaires

- des appels de fonctions successifs

- un flux d’exécution

- un flux de données

Niveaux de représentation: graphe d’appel

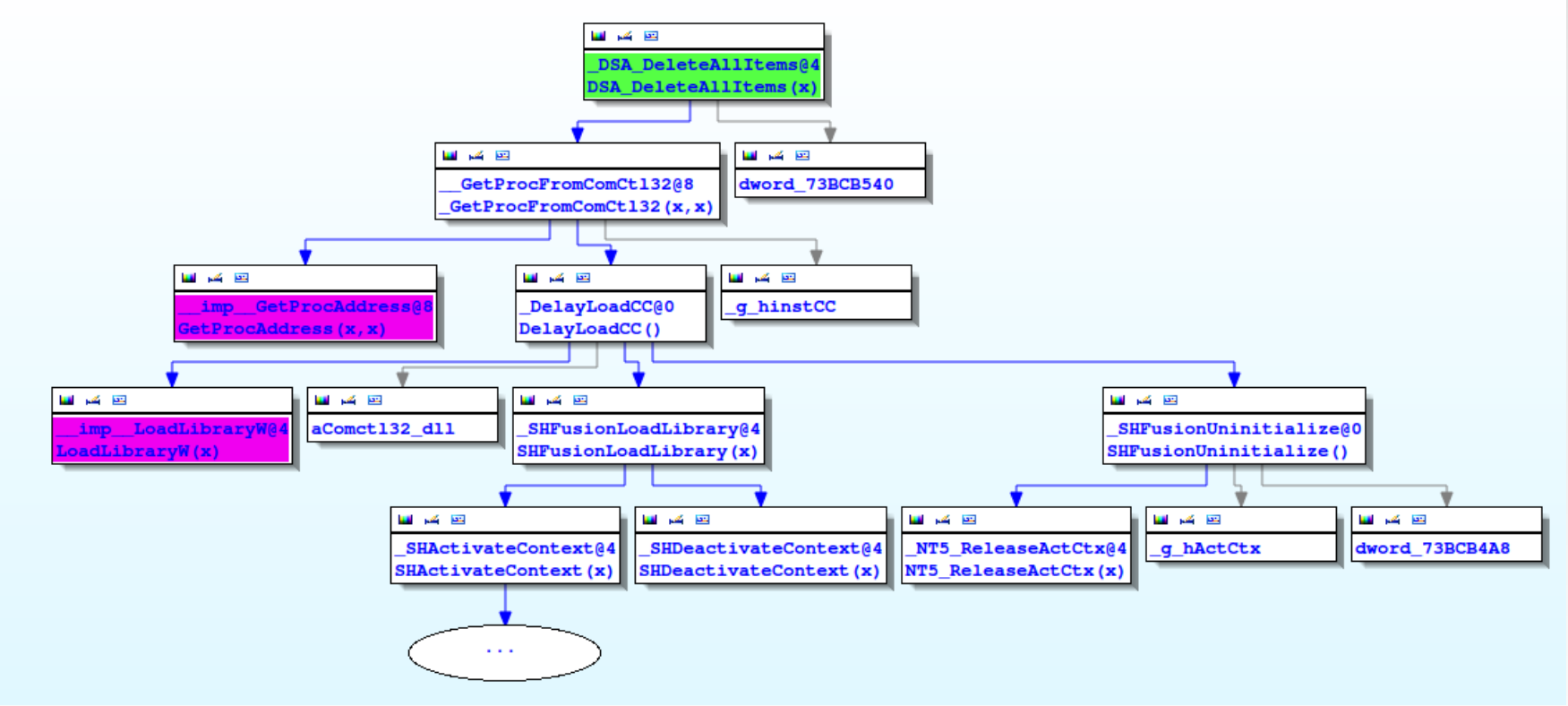
Il s’agit de la réprésentation de tous les appels des fonctions.

Ceci peut être particulierement utiles pour savoir les fonctions importantes qui seront appelées même si bien imbriquée

Exemple:

Un programme prend un paramètre un paquet réseau avec une syntaxe bien précise et un numéro d’opération et va effectuer des opérations unitaires (lire / écrire / liste processus / exécution de commande)

Niveaux de représentation: graphe d'appel



Niveaux de représentation: graphe d’appel

On peut utiliser l’outil Proximity Browser d’IDA pour avoir une idée du graphe. Attention, dans ce mode, certain noeuds ne sont pas forcément des fonctions.

Disponible depuis IDA 6.2

Niveaux de représentation : Le CFG

Un programme informatique peut se voir de différentes façons:

- une succession d’instructions linéaires

- des appels de fonctions successifs

- un flux d’exécution

- un flux de données

Niveaux de représentation : Le CFG

Le CFG est le Control Flow Graph

Autrement dit en français le graphe du flux d'exécution.

Définition:

Il s'agit de la représentation de l'ensemble des chemins possibles d'exécution d'un programme

Le CFG est une notion très importante en rétro-ingéniérie.

Niveaux de représentation : Le CFG

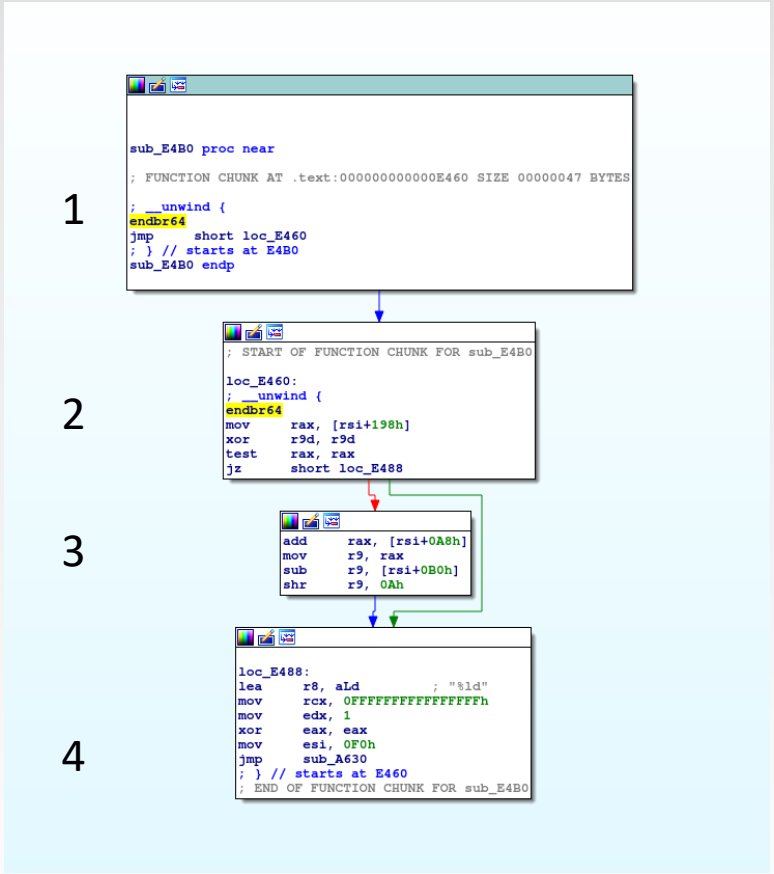
Un exemple de CFG:

Composé de 4 noeuds, numéroté de 1 à 4.

Il contient 4 arêtes.

On doit voir avec ce graphe, que 2 chemins d'exécution sont possibles.

Lesquels sont-ils ?



Niveaux de représentation : Le Basic Block

Un basicblock est un noeud du CFG, il est composé d'instructions qui seront toutes exécutées si la première l'est. Un chemin d'exécution du programme est composé d'un nombre entier de basicblock.

Il contient une entrée (première instruction du basicBlock)

Il contient une sortie (dernière instruction du basicBlock)

La sortie du basicBlock est une instruction qui modifie de le flux d'exécution (control flow).

Il peut s'agir:

- - D'un saut conditionnel : jg / ja / je / jns
- - D'un saut inconditionnel : jmp _label
- - Du retour à la fonction appelante : retn

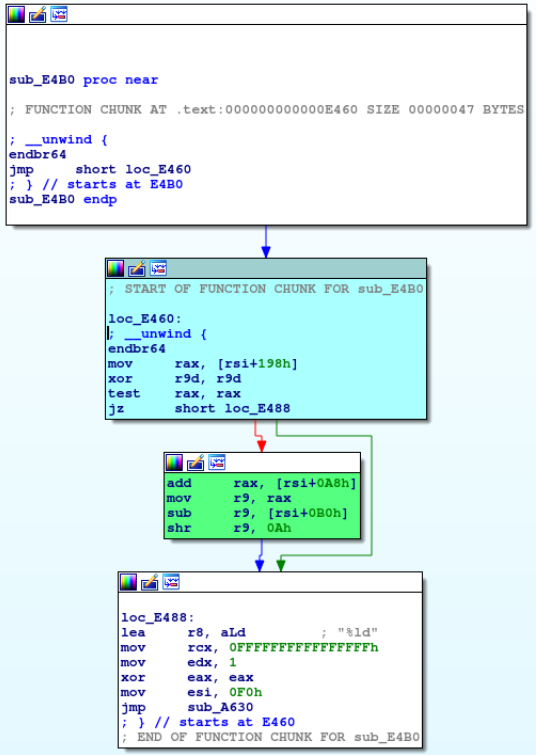
Niveaux de représentation : Le Basic Block

Le CFG ci-contre contient 4 Basicblock

En entrant dans le Basicblock bleu, les 5 instructions seront exécutées, il y aura ensuite deux entrées possibles selon le résultat du test.

Pareil pour le Basicblock vert, si la première instruction est exécutée, les 4 instructions seront exécutées.

<u>Combien de Basicblock comportent les 2 chemins d'exécution du programme ?</u>



Niveaux de représentation: Le CFG

On peut utiliser l'outil de représentation en Graphe d'IDA

Il faut appuyer sur la touche <espace> pour basculer en mode graphe

2 situations dans lesquelles ce mode est inaccessible :

- Lorsque le code n'est pas analysé (IDA analyse le code par ses branchements et pas linéairement)

- lorsque la fonction contient trop de basicblock

Niveaux de représentation : Le DFG

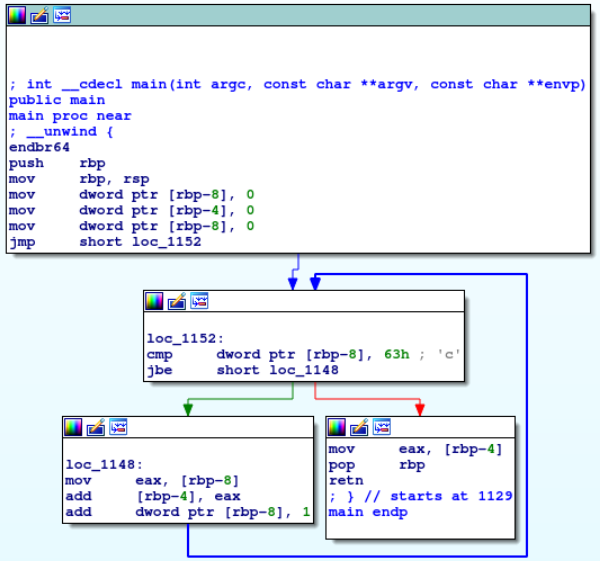
Un programme informatique peut se voir de différentes façons:

- une succession d'instructions linéaires
- des appels de fonctions successifs
- un flux d'exécution
- un flux de données

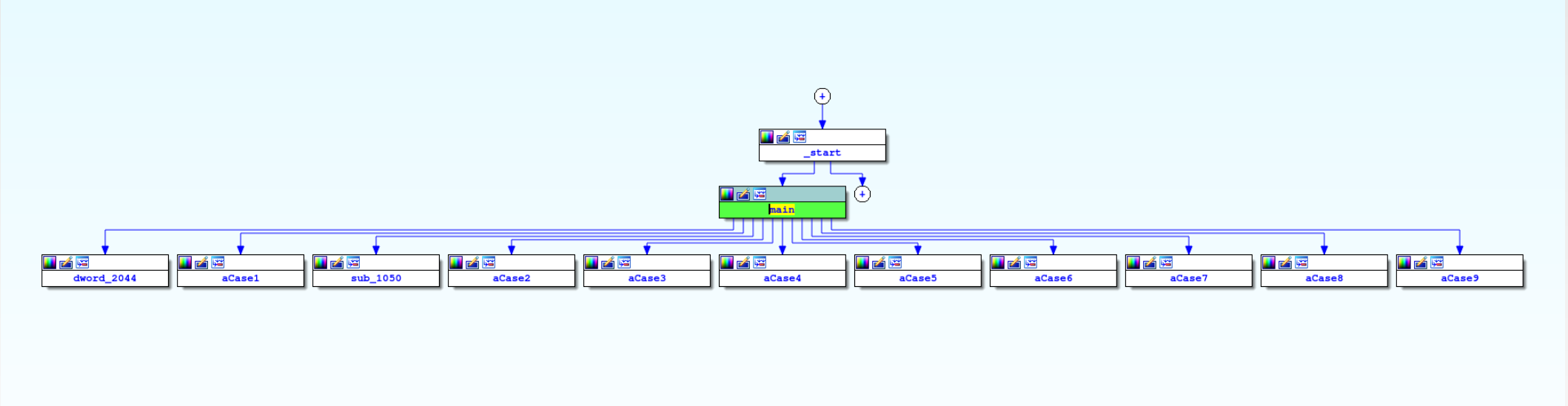
Les boucles

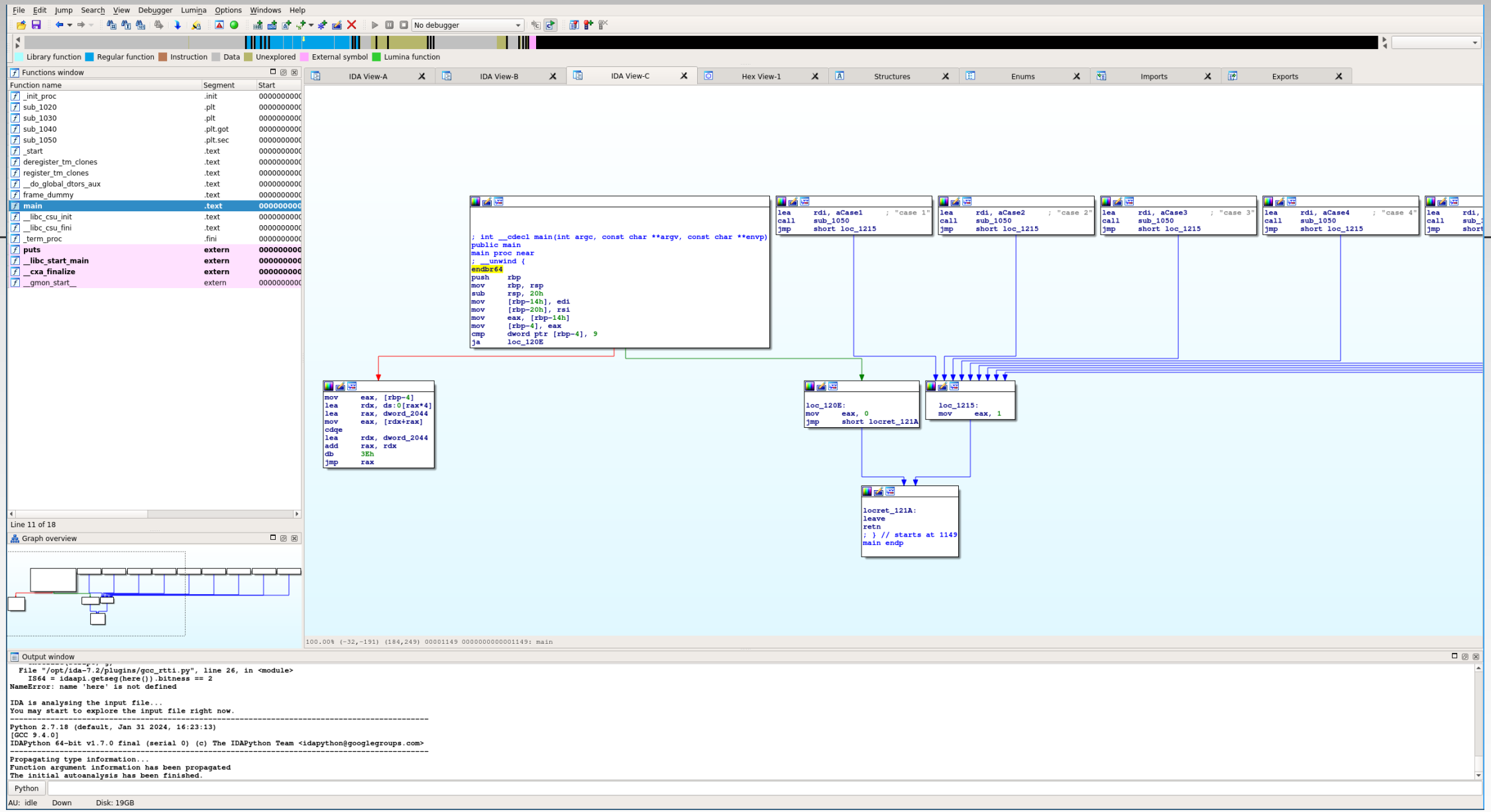
En général, les registres $RCX ou $RAX sont utilisés en tant que compteur. Mais cela peut être également une variable locale comme dans le dessin ci-contre:

Quelle variable locale est utilisée comme compteur de boucle dans l'exemple ci-contre ?



switch case


_start
main
dword_2044
aCase1
sub_1050
aCase2
aCase3
aCase4
aCase5
aCase6
aCase7
aCase8
aCase9


File Edit Jump Search View Debugger Lumina Options Windows Help
No debugger
Library function Regular function Instruction Data Unexplored External symbol Lumina function
Functions window
Function name Segment Start
_init_proc .init
sub_1020 .plt
sub_1030 .plt
sub_1040 .plt.got
sub_1050 .plt.sec
_start .text
deregister_tm_clones .text
register_tm_clones .text
__do_global_dtors_aux .text
frame_dummy .text
main .text
__libc_csu_init .text
__libc_csu_fini .text
_term_proc .fini
puts extern
__libc_start_main extern
__cxa_finalize extern
__gmon_start__ extern
Line 11 of 18
Graph overview
IDA View-A
IDA View-B
IDA View-C
Hex View-1
Structures
Enums
Imports
Exports
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near
; __unwind {
endbr64
push rbp
mov rbp, rsp
sub rsp, 20h
mov [rbp-14h], edi
mov [rbp-20h], rsi
mov eax, [rbp-14h]
mov [rbp-4], eax
cmp dword ptr [rbp-4], 9
ja loc_120E
lea rdi, aCase1 ; "case 1"
call sub_1050
jmp short loc_1215
lea rdi, aCase2 ; "case 2"
call sub_1050
jmp short loc_1215
lea rdi, aCase3 ; "case 3"
call sub_1050
jmp short loc_1215
lea rdi, aCase4 ; "case 4"
call sub_1050
jmp short loc_1215
mov eax, [rbp-4]
lea rdx, ds:0[rax*4]
lea rax, dword_2044
mov eax, [rdx+rax]
cdqe
lea rdx, dword_2044
add rax, rdx
db 3Eh
jmp rax
loc_120E:
mov eax, 0
jmp short locret_121A
loc_1215:
mov eax, 1
locret_121A:
leave
retn
; } // starts at 1149
main endp
100.00% (-32,-191) (184,249) 00001149 0000000000001149: main
Output window
File "/opt/ida-7.2/plugins/gcc_rtti.py", line 26, in <module>
IS64 = idaapi.getseg(here()).bitness == 2
NameError: name 'here' is not defined
IDA is analysing the input file...
You may start to explore the input file right now.
Python 2.7.18 (default, Jan 31 2024, 16:23:13)
[GCC 9.4.0]
IDAPython 64-bit v1.7.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
Python
AU: idle Down Disk: 19GB

Les registres volatiles / non-volatiles

Derrière ce nom compliqué, le concept est très simple:

-> Il s'agit des registres qui peuvent être modifiés de façon irreversible par une sous-fonction.

En pratique, lorsqu'on revient d'une sous-fonction, il y a des registres qui doivent être restaurés par la sous-fonction et d'autres non.

Ceux qui doivent être restaurés sont appelés "non-volatiles".

Les registres volatiles / non-volatiles

Liste des registres non-volatiles en x86_64 (ABI Système V):

Ils sont appelés “callee saved” dans la littérature. En effet, leur valeur est sauvegardée puis restaurée si le registre est utilisé dans une sous-fonction.

%RBX

%RSP %RBP

%R12 %R13 %R14 %R15

A retenir : RBX RBP RSP sont non volatiles

Le reverse dynamique

101

Le reverse dynamique

Définition

On parle de rétro-ingénierie dynamique lorsqu'on regarde un programme s'exécuter dans son environnement.

Il est recommandé d'utiliser un environnement virtualisé dans tous les cas où le code est inconnu et potentiellement dangereux.

Des outils existent pour observer l'exécution du code: ce sont les débuggeurs.

Le reverse dynamique - méthodo

Il faut trouver un endroit dans le code et arrêter le programme pour inspecter l'état interne du processeur. On arrête le programme avec des breakpoints (points d'arrêt)

Il faut comprendre d'abord si :

- le code est strippé / compilé en statique / contient des symboles de debug

- le code est obfusqué

On peut alors en dynamique savoir:

- Quelles sont les valeurs des paramètres des fonctions

- Connaître le flux réel d'exécution et de données

- Il est très facile de se retrouver à debugger du code non pertinent en dynamique

Le reverse dynamique – élaborer une stratégie

- Ex1: reverser un protocole:
- On isole les API qui manipulent le protocole
- On isole les fonctions qui lisent et écrivent par ce protocole
- On inspecte la mémoire
- On peut coder un parseur (dans le language de notre choix)

Le reverse dynamique – Les outils

GDB (GNU debuggeur)

peda / gef / pwntools : wrappers basés sur gdb

windbg

radare2

IDA (peut être utilisé en mode debuggeur)

ollydbg / x64dbg (sous Windows)

Dans ce cours, nous allons utiliser GDB pour expliquer le reverse dynamique.

-> Merci d'installer gef pour le prochain TP (instruction sur moodle)

Le reverse dynamique – Les outils

GDB (Gnu debuggeur)

C'est l'outil incontournable et historique à connaître sous linux pour du dynamique

Il a eu plusieurs "wrappers" ces dernières années : peda / gef (recommandé)

Les commandes les plus utiles:

x/10xg $rax : inspecter la mémoire -> 10 qwadword(8bytes) depuis registre $rax

Disass op_assign : Désassembler la fonction **op_assign**

b *0x41414141 : Poser un breakpoint sur l'addresse 0x41414141

i b : info breakpoints: Lister les breakpoints enregistrés

set args p1 p2 : passe des arguments p1 p2 au programme débuggé par gdb.

Le reverse dynamique – Les outils - GDB

A quoi sert GDB ?

-> Il permet de s'attacher à un processus du système (ou bien de le démarrer) pour l'inspecter.

Une fois attaché au processus, il est possible de voir:

- l'ensemble de sa mémoire virtuelle

- les instructions du programme

- toutes les différentes parties du binaire ayant donné naissance au processus. On appelle ces parties des sections (ex: .text .data .rodata)

Le reverse dynamique – Rappels- Mémoire virtuelle

Un ordinateur est composé entre autre d'un processeur, de mémoire vive et d'un disque dur. Cette mémoire vive est appellee RAM (Random Access Memory).

Le processeur a bien accès à cette mémoire vive (physique), il a un accès en lecture / écriture. Néanmoins, il existe une couche d'abstraction (virtuelle) entre lui et cette mémoire vive.

Tout se passe comme si le processeur avait un espace d'adressage virtuel (presque) infini. Un mécanisme sous-jacent (Memory Management Unit MMU) s'occupe de faire la translation d'adresse pour que cet espace corresponde aux pages physique en mémoire vive.

Le reverse dynamique – Mémoire virtuelle

Intérêt pour le reverse: Cet espace d'adressage virtuel est le "terrain de jeu" pour la rétro-ingénierie dynamique. Lorsque GDB est attaché avec les droits adequats, tout est modifiable, tout est observable.

Il contient:

- le programme binaire (dispersé en mémoire)

- la pile du processus (unite d'exécution principale) et des autres threads

- le (ou les) tas

- les bibliothèques partagées du programme

Le reverse dynamique – Les outils - GDB

Comment s'attacher à un processus ? gdb –p <pid>

Comment démarrer un processus avec des paramètres ?

gdb <programme>

(gdb) set args <arg1> <arg2> <arg3>

PROTIP:

Le repertoire /proc/<pid> contient beaucoup d'information sur les processus <pid> en cours:

- ligne de commande
- informations sur l'espace d'adressage
- fichiers ouverts
- Les variables d'environnement

Le reverse dynamique – Les outils - GDB

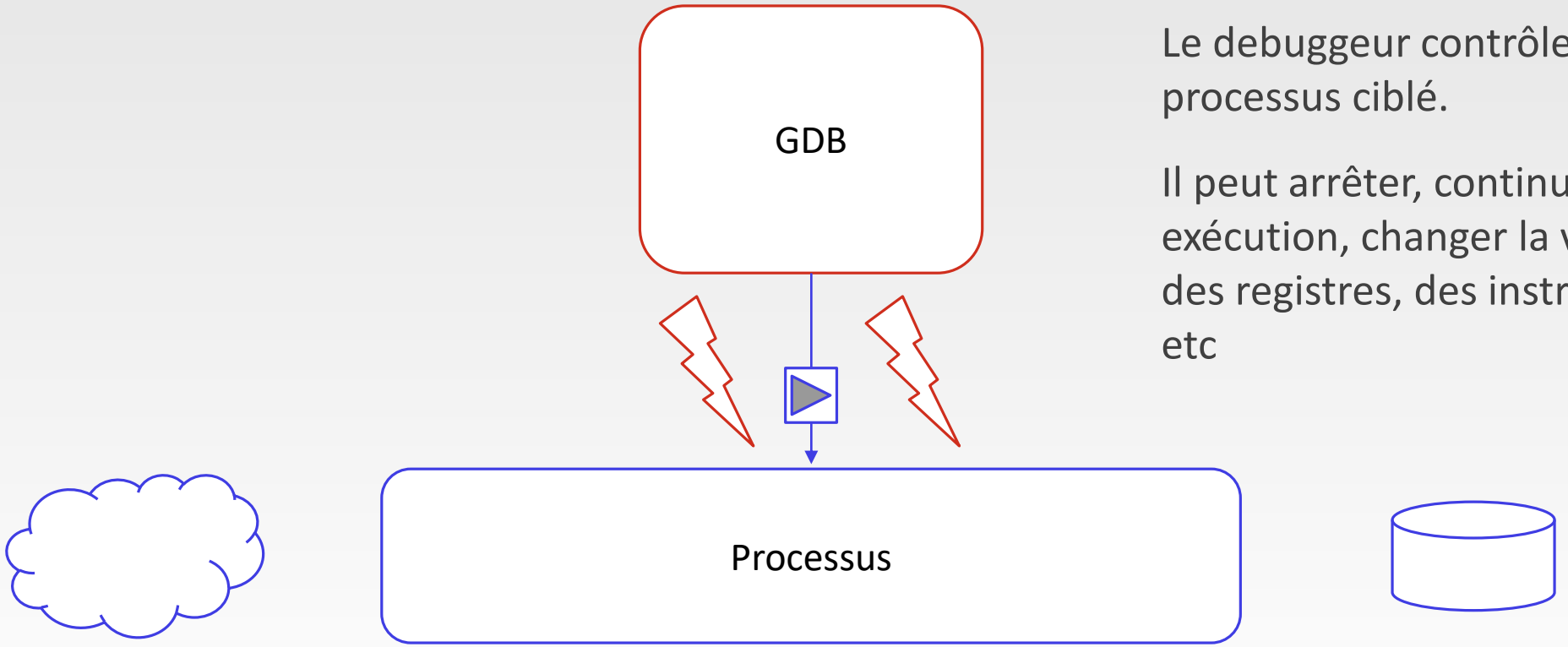
Après avoir réussi à s'attacher au processus cible, on peut observer l'espace d'adressage complet:

Il est possible de suivre l'exécution du programme pas à pas. Pour chaque instruction exécutée, il sera possible d'inspecter la mémoire (espace d'adressage) ainsi que les registres du processeur.

On utilise pour cela: les breakpoints (notion fondamentale en rétro-ingénierie dynamique)

Un point d'arrêt positionné à l'adresse A, rendra la main au débuggeur lorsque le registre RIP va rencontrer cette adresse.

Le reverse dynamique – Les outils - GDB



Le debuggeur contrôle le processus ciblé.

Il peut arrêter, continuer son exécution, changer la valeur des registres, des instructions etc

Le reverse dynamique – Les outils - GDB

Le breakpoint : Comment ça marche ?

Deux types de breakpoint, deux fonctionnements différents

- le breakpoint software

- le breakpoint hardware

Il s'agit d'un mécanisme pour arrêter l'exécution du processus attaché

Lorsqu'on parle de hardware dans ce contexte là, il s'agit de l'utilisation d'une fonctionnalité implémentée dans le processeur lui-même.

Lorsqu'on parle de software dans ce contexte là, il s'agit d'utiliser un mécanisme plus haut-niveau (c'est le programme qui l'implémente – ici gdb)

Le reverse dynamique – Les outils - GDB

Le breakpoint software : Comment ça marche ?

Lorsqu'on positionne un breakpoint software sur une instruction, gdb va remplacer l'instruction par une autre instruction qui permet d'émettre un signal. Une fois ce signal émis, gdb restaure l'instruction d'origine.

Attention: gdb fait en sorte que l'opération soit transparente pour l'utilisateur.

Avantage: Nombre illimité de breakpoint

Inconvénient: Modifie le code binaire

TP !

Le reverse dynamique – Les outils - GDB

Le breakpoint hardware : Comment ça marche ?

Le processeur contient des registres spéciaux dédiés à cet effet. Lorsque le registre RIP correspond à l'un de ces registres spéciaux, alors un signal est émis.

Avantage:

- Utilise uniquement ces registres et ne touche pas à la mémoire

Inconvénient:

- Nombre limité de registre

Mise en pratique

101

Démonstration

Installation d’IDA

Application sur le crackme

-> structure du graphe / basicblock

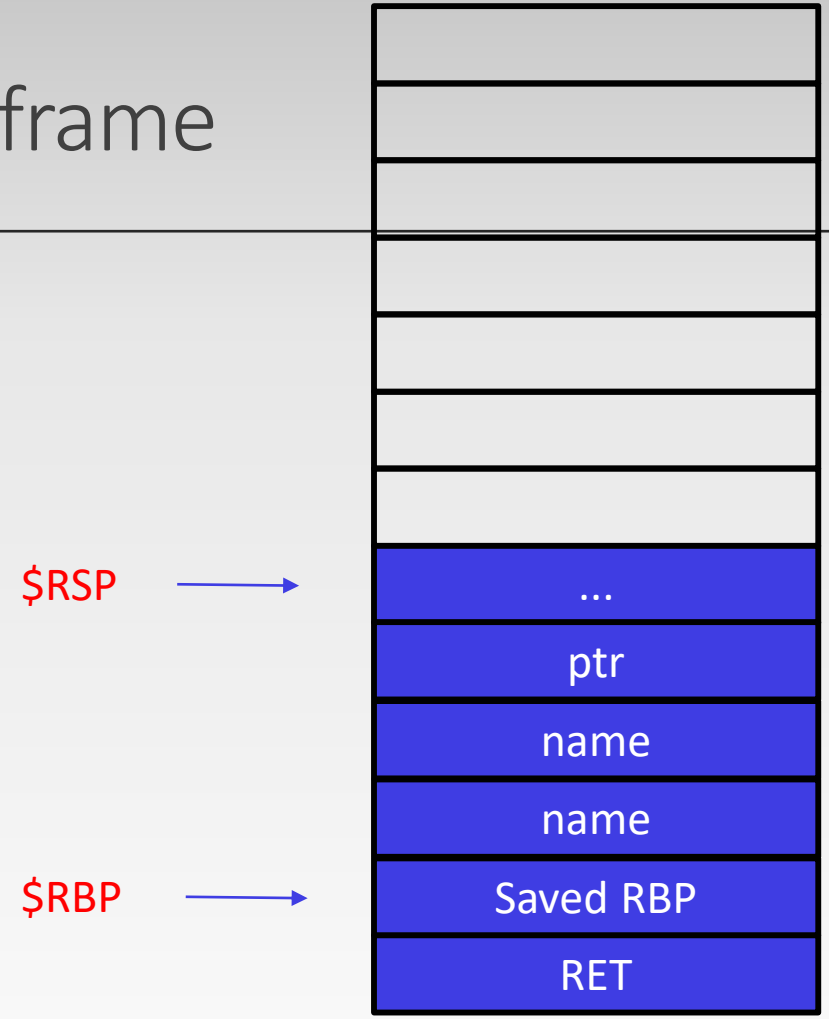
Le reverse dynamique – le stack frame

Un stackframe est lié à une fonction.

Il s'agit de l'espace alloué sur la pile dédié à la fonction.

Exemple:

```
Int main(void)
{
Char name[0x10] = {0};
int *ptr = NULL;
printf("hello world\n");
Return 4;
}
```



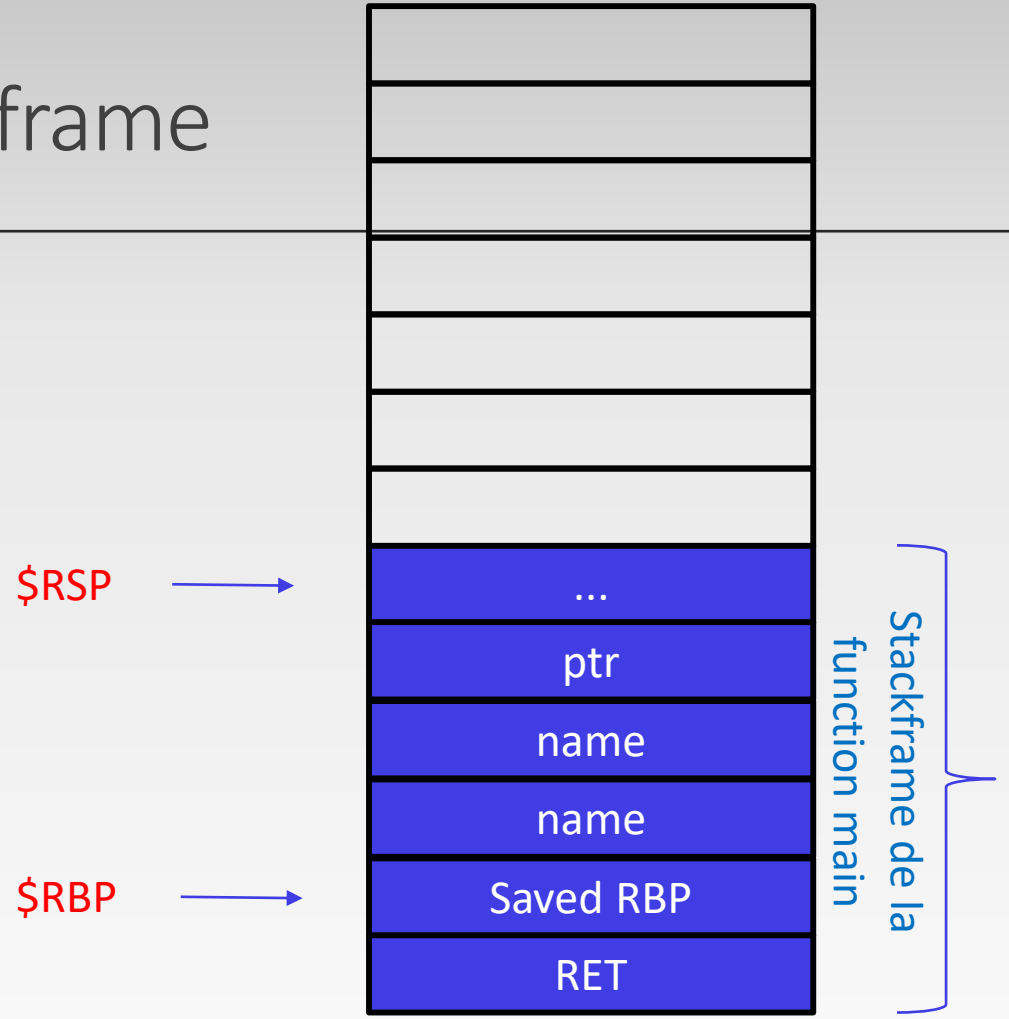
Le reverse dynamique – le stack frame

Un stackframe est lié à une fonction.

Il s'agit de l'espace alloué sur la pile dédié à la fonction.

Exemple:

```
Int main(void)
{
Char name[0x10] = {0};
int *ptr = NULL;
printf("hello world\n");
Return 4;
}
```



Le reverse dynamique – le stack frame

Un stackframe est lié à une fonction.

Il s'agit de l'espace alloué sur la pile dédié à la fonction.

Exemple:

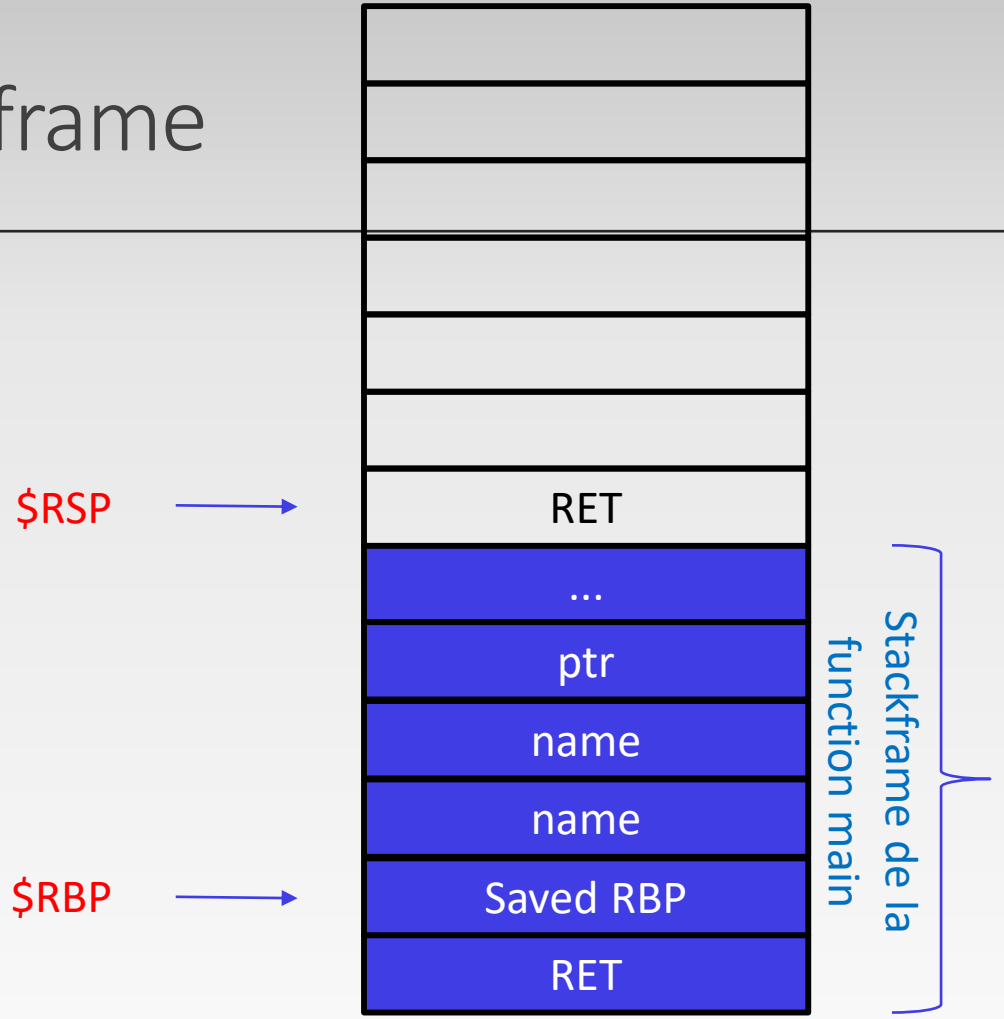
Call <printf>

<printf>:

<prologue>

<corps de function>

RETN



Le reverse dynamique – le stack frame

Un stackframe est lié à une fonction.

Il s'agit de l'espace alloué sur la pile dédié à la fonction.

Exemple:

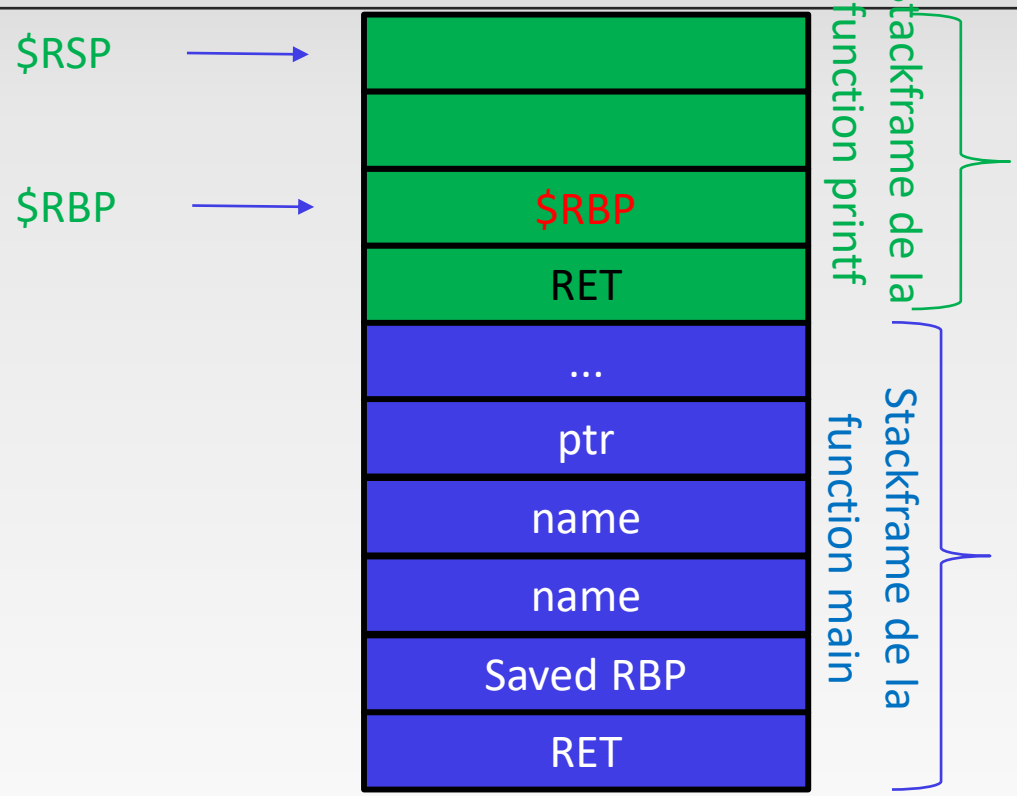
Call <printf>

<printf>:

<prologue>

<corps de function>

RETN



Appels systèmes

Les appels système suivent une ABI particulière, car il existe une instruction pour les déclencher, et le registre RAX contiendra le numéro de l'appel système.

L'instruction est *syscall*.

Ensuite, les paramètres sont traités par les registres.

Appels systèmes

Linux x86_64 System Call Reference Table

This document serves as a reference to the system calls within the x86_64 Linux Kernel.

x86_64 Linux Syscall Structure

Instruction	Syscall #	Return Value	arg0	arg1	arg2	arg3	arg4	arg5
SYSCALL	rax	rax	rdi	rsi	rdx	r	r	r

x86_64 Linux Syscall Table

rax	System Call	rdi	rsi	rdx	r10	r8	r9
0	sys_read	unsigned int fd	char* buf	size_t count			
1	sys_write	unsigned int fd	const char* buf	size_t count			
2	sys_open	const char* filename	int flags	int mode			
3	sys_close	unsigned int fd					

Appels systèmes

Linux x86_64 System Call Reference Table

This document serves as a reference to the system calls within the x86_64 Linux Kernel.

x86_64 Linux Syscall Structure

Instruction	Syscall #	Return Value	arg0	arg1	arg2	arg3	arg4	arg5
SYSCALL	rax	rax	rdi	rsi	rdx	r	r	r

x86_64 Linux Syscall Table

rax	System Call	rdi	rsi	rdx	r10	r8	r9
0	sys_read	unsigned int fd	char* buf	size_t count			
1	sys_write	unsigned int fd	const char* buf	size_t count			
2	sys_open	const char* filename	int flags	int mode			
3	sys_close	unsigned int fd					

Questions ?

MERCI