La rétro-ingénierie

CHAPITRE 3 – ETUDE DE CAS & OBFUSCATION – 24/11/2024

Objectifs cours 3

- Rappels / Ressources pour le prochain TP :
  - Commandes GDB
  - Appels Systèmes
  - rand() / srand()
- Mécanismes de ralentissement de rétro-ingénierie : Obfuscation
  - Junk code
  - Graph flattening
  - Machine virtuelle -> étude de cas

Conclusion

GDB : commandes utiles (disponible sur moodle)

x/32i $rip # on affiche les 32 instructions du programme après $rip

set args # on ajoute un argument au programme

show args # on montre les arguments du programme

break *<addr> # on pose un breakpoint sur <addr>

maintenance info sections # on affiche une partie du layout mémoire

disass <function> # on disassemble la fonction <function>

disp /i $rip # on affiche à chaque commande l'instruction courante

x/s $rdi # on affiche la chaîne de caractère pointée par $rdi

x/s $rsi # on affiche la chaîne de caractère pointée par $rsi

ni # on exécute l'instruction courante et on passe à la suivante

si # pareil que ni mais l'exécution suit les appels

set disassembly-flavor intel # syntaxe INTEL

x/32xg $rsp # on affiche 32 valeurs à partir de $rsp

bt # on affiche la call stack

i r <reg> # on affiche la valeur du register (ex : $rax)

i r $eflags # on affiche le registre $eflags

i b # on affiche la liste des breakpoints posés

run # on execute le programme

continue # on continue l'exécution après un breakpoint par exemple

set $rax=0x1337 # on assigne 0x1337 au registre $rax

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					

Fonction rand() / srand()

Ces nombres sont des nombres pseudo-aléatoires

Ils sont déterminés par un entier qu'on appelle une *seed*.

On les utilise traditionnellement de la façon suivante:

```
int main(void)
{
   unsigned int i = 0;
   srand(time(NULL));

   printf("[+] Voici 10 nombres pseudo aléatoires:\n");
   for ( i = 0; i < 10; i ++)
      printf("%i", rand() % 100);

   return 0;

}
```

Fonction rand() / srand()

srand() permet de spécifier cette seed qui, elle, va determiner une suite unique de nombres.

```
int main(void)
{
   unsigned int i = 0;
   srand(time(NULL));

   printf("[+] Voici 10 nombres pseudo aléatoires:\n");
   for ( i = 0; i < 10; i ++)
      printf("%i", rand() % 100);

   return 0;

}
```

Fonction rand() / srand()

rand() vient ensuite calculer le prochain nombre d'après l'algorithme personalisé par la seed.

```
int main(void)
{
   unsigned int i = 0;
   srand(time(NULL));

   printf("[+] Voici 10 nombres pseudo aléatoires:\n");
   for ( i = 0; i < 10; i ++)
      printf("%i", rand() % 100);

   return 0;

}
```

Fonction rand() / srand()

Si la seed est controlée, toutes les valeurs retournées par rand() sont connues et déterminées.

Application -> TP

```
int main(void)
{
   unsigned int i = 0;
   srand(0x41414141);

   printf("[+] Voici 10 nombres pseudo aléatoires:\n");
   for ( i = 0; i < 10; i ++)
      printf("%i", rand() % 100);

   return 0;

}
```

Obfuscation

101

Obfuscation / Anti-rétro-ingénierie

Nous allons voir quelques techniques basiques qui permettent de ralentir la rétro-ingénierie de programme.

Ce sont des techniques qui cachent (obscurcissent) le véritable fonctionnement interne du programme souvent au détriment de quelque chose (performance / taille ou autre)

Obfuscation : Graph Flattening

La technique du graphe flattening est de casser la structure de graphe d'IDA par exemple pour avoir un graphe difficile à comprendre.

Tous les basicblocks seront au même niveau, et il existe un paramètre, comme un compteur qui va permettre de sauter sur les bons basicblocks.

Obfuscation : Graph Flattening

L'outil TIGRESS développé par l'université de l'Arizona permet (entre autre) de modifier le code source afin de changer le CFG produit par la compilation.

On peut considérer le code ci-contre:

Que fait la fonction ?

```
#include <stdio.h>

int compute(int x) {
    int result = 0;
    if (x > 0) {
        result = x * 2;
    } else if (x == 0) {
        result = 42;
    }
    return result;
}
```

Obfuscation : Graph Flattening

```
#include <stdio.h>

int compute(int x) {
    int result = 0;
    if (x > 0) {
        result = x * 2;
    } else if (x == 0) {
        result = 42;
    }
    return result;
}
```

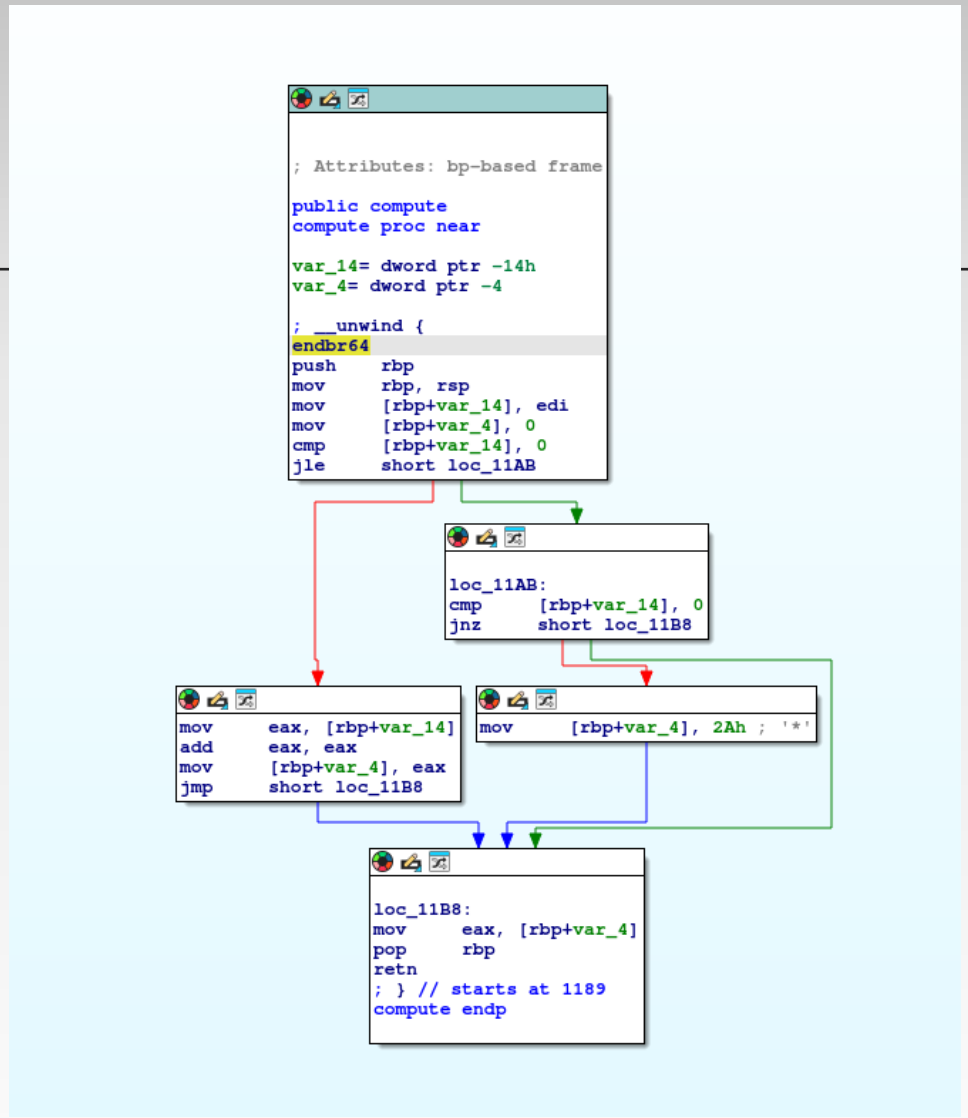
Compilation :

$ gcc main.c -o avant_obfuscation

Obfuscation : Graph Flattening

```c
#include <stdio.h>

int compute(int x) {
    int result = 0;
    if (x > 0) {
        result = x * 2;
    } else if (x == 0) {
        result = 42;
    }
    return result;
}
```



Obfuscation : Graph Flattening

```
#include <stdio.h>

int compute(int x) {
    int result = 0;
    if (x > 0) {
        result = x * 2;
    } else if (x == 0) {
        result = 42;
    }
    return result;
}
```

Graphe Flattening transformation :

$ tigress –Transformation=Flatten –Functions=compute –out obfu_1.c original.c

Obfuscation : Graph Flattening

```
#include <stdio.h>

int compute(int x) {
    int result = 0;
    if (x > 0) {
        result = x * 2;
    } else if (x == 0) {
        result = 42;
    }
    return result;
}
```

Graphe Flattening transformation :

$ tigress –Transformation=Flatten –Functions=compute –out obfu_1.c main.c

$ ls -lah obfu1.c main.c

-rw-rw-r-- 1 h h 350 nov. 23 18:47 main.c

-rw-rw-r-- 1 h h 87K nov. 23 18:49 obfu1.c

$ gcc obfu1.c -o apres_obfuscation

ls -lah apres_obfuscation avant_obfuscation

-rwxrwxr-x 1 h h 16K nov. 23 19:54 apres_obfuscation

-rwxrwxr-x 1 h h 16K nov. 23 19:40 avant_obfuscation

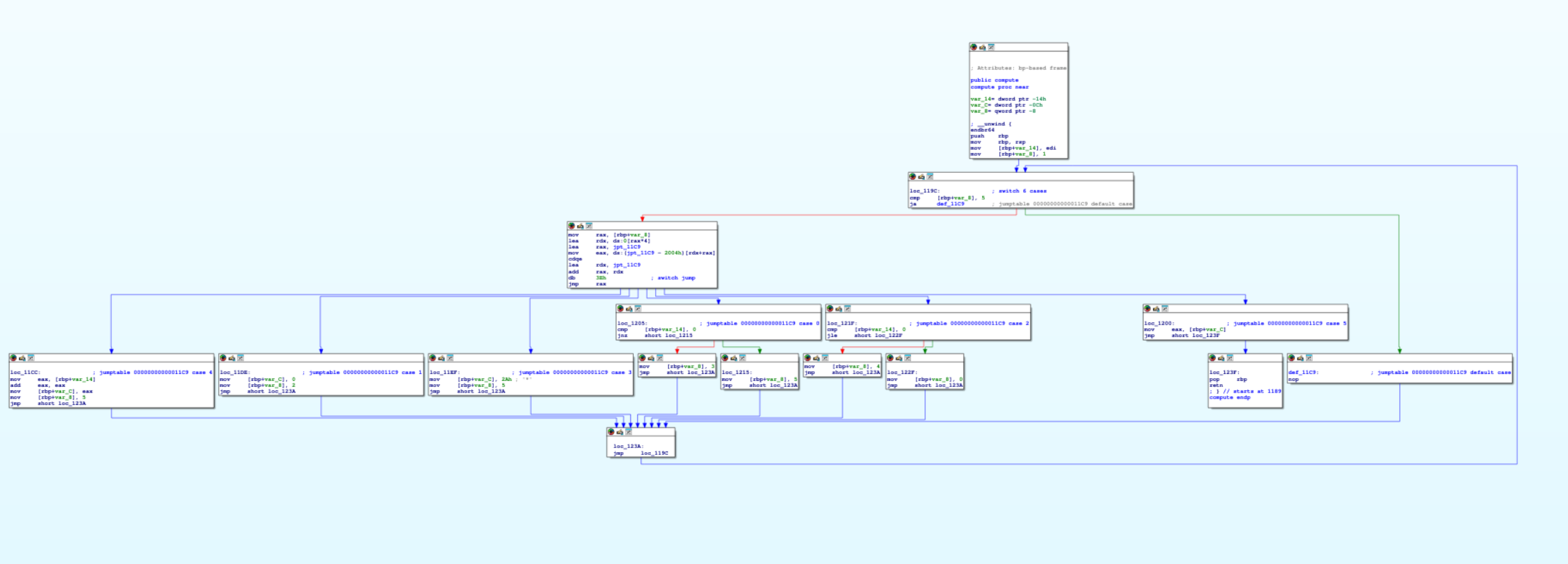
Obfuscation : Graph Flatt

```
#include <stdio.h>

int compute(int x) {
    int result = 0;
    if (x > 0) {
        result = x * 2;
    } else if (x == 0) {
        result = 42;
    }
    return result;
}
```

```
int compute(int x )
{
  int result ;
  unsigned long _TIG_FN_ghKb_1_compute_next ;
[...]
  while (1) {
    switch (_TIG_FN_ghKb_1_compute_next) {
    case 4UL:
#line 7 "main.c"
    result = x * 2;
    {
    _TIG_FN_ghKb_1_compute_next = 5UL;
    }
    break;
    case 1UL:
#line 4
    result = 0;
    {
    _TIG_FN_ghKb_1_compute_next = 2UL;
    }
    break;
  [...]
```

Obfuscation : Graph Flattening



Obfuscation : Junk Code

La technique du junk code consiste à ajouter du code aléatoirement dans le binaire dans des endroits non utilisés.

Cette technique est efficace pour perturber les désassembleurs en cas de code non-aligné.

Obfuscation : Junk Code

```
0000000000001000 <_start>:
   1000:   cc                      int3
   1001:   e8 0d 00 00 00          call   1013 <_start+0x13>
   1006:   09 50 dd                or     DWORD PTR [rax-0x23],edx
   1009:   ee                      out    dx,al
   100a:   cc                      int3
   100b:   cc                      int3
   100c:   48 b8 42 42 42 42 41            movabs rax,0xc0ff484142424242
   1013:   48 ff c0
   1016:   c3                      ret
   1017:   eb fe                   jmp    1017 <_start+0x17>
```

Obfuscation : Junk Code

```
0000000000001000 <_start>:
    1000:   cc                      int3
    1001:   e8 0d 00 00 00          call   1013 <_start+0x13>
    1006:   09 50 dd                or     DWORD PTR [rax-0x23],edx
    1009:   ee                      out    dx,al
    100a:   cc                      int3
    100b:   cc                      int3
    100c:   48 b8 42 42 42 42 41            movabs rax,0xc0ff484142424242
    1013:   48 ff c0
    1016:   c3                      ret
    1017:   eb fe                   jmp    1017 <_start+0x17>
```

Dead code insertion

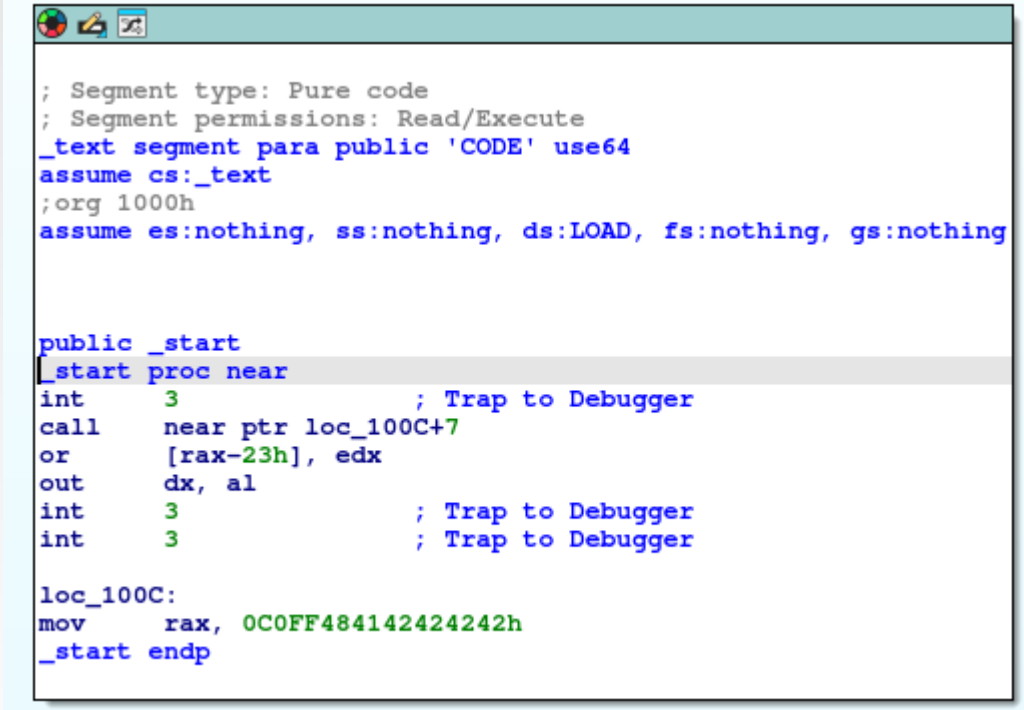
Obfuscation : Junk Code

```
0000000000001000 <_start>:
   1000:   cc                      int3
   1001:   e8 0d 00 00 00          call   1013 <_start+0x13>
   1006:   09 50 dd                or     DWORD PTR [rax-0x23],edx
   1009:   ee                      out    dx,al
   100a:   cc                      int3
   100b:   cc                      int3
   100c:   48 b8 42 42 42 42 41            movabs rax,0xc0ff484142424242
   1013:   48 ff c0
   1016:   c3                      ret
   1017:   eb fe                   jmp    1017 <_start+0x17>
```

Le call arrive au milieu d'une instruction, de sorte que le désassembleur semble perdu.

Pourquoi ?

Obfuscation : Junk Code



```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
;org 1000h
assume es:nothing, ss:nothing, ds:LOAD, fs:nothing, gs:nothing



public _start
_start proc near
int     3               ; Trap to Debugger
call    near ptr loc_100C+7
or      [rax-23h], edx
out     dx, al
int     3               ; Trap to Debugger
int     3               ; Trap to Debugger

loc_100C:
mov     rax, 0C0FF484142424242h
_start endp
```

IDA ne parvient pas à trouver

l'instruction cachée.

Pourtant elle existe bel et bien.

Obfuscation : Junk Code

```
gef➤  x/10i _start
  0x555555555000 <_start>:  int3
=> 0x555555555001 <_start+1>:          call   0x555555555013 <_start+19>
  0x555555555006 <_start+6>:           or     DWORD PTR [rax-0x23],edx
  0x555555555009 <_start+9>:           out    dx,al
  0x55555555500a <_start+10>:          int3
  0x55555555500b <_start+11>:          int3
  0x55555555500c <_start+12>:          movabs rax,0xc0ff484142424242
  0x555555555016 <_start+22>:          ret
  0x555555555017 <_start+23>:          jmp    0x555555555017 <_start+23>
  0x555555555019:add    BYTE PTR [rax],al
gef➤  x/4i _start+19
  0x555555555013 <_start+19>:          inc    rax
  0x555555555016 <_start+22>:          ret
  0x555555555017 <_start+23>:          jmp    0x555555555017 <_start+23>
```

Obfuscation : Junk Code

```
gef➤  x/10i _start
   0x555555555000 <_start>:  int3
=> 0x555555555001 <_start+1>:        call   0x555555555013 <_start+19>
   0x555555555006 <_start+6>:        or     DWORD PTR [rax-0x23],edx
   0x555555555009 <_start+9>:        out    dx,al
   0x55555555500a <_start+10>:       int3
   0x55555555500b <_start+11>:       int3
   0x55555555500c <_start+12>:       movabs rax,0xc0ff484142424242
   0x555555555016 <_start+22>:       ret
   0x555555555017 <_start+23>:       jmp    0x555555555017 <_start+23>
   0x555555555019:add    BYTE PTR [rax],al
gef➤  x/4i _start+19
   0x555555555013 <_start+19>:       inc    rax
   0x555555555016 <_start+22>:       ret
   0x555555555017 <_start+23>:       jmp    0x555555555017 <_start+23>
```

Obfuscation : VM

Pour cacher l'exécution d'un programme, il est possible de virtualiser ses instructions

Comment ça marche:

Un programme est composé d'instructions assembleurs. Or ces instructions pourraient être codées par une fonction dans n'importe quelle langage:

Virtualiser signifie dans ce contexte que le programme écrit avec des instructions bas-niveau (ex push / pop / mov / xor / ret) sera transformé en instructions haut-niveau.

Ces instructions haut-niveau seront codées dans un langage de programmation.

Obfuscation : VM

Exemple 1:

J’écris une fonction en C qui ajoute deux nombres:

```
int func_origin( int a, int b) {
    return a + b
}
```

La compilation de cette fonction donnera le code suivant:

Obfuscation : VM - Exemple

```
0000000000001129 <func_add>:
1129:   f3 0f 1e fa             endbr64
112d:   55                      push   rbp
112e:   48 89 e5                mov    rbp,rsp
1131:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
1134:   89 75 f8                mov    DWORD PTR [rbp-0x8],esi
1137:   8b 55 fc                mov    edx,DWORD PTR [rbp-0x4]
113a:   8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
113d:   01 d0                   add    eax,edx
113f:   5d                      pop    rbp
1140:   c3                      ret
```

Obfuscation : VM - Exemple

```
0000000000001129 <func_add>:
1129:   f3 0f 1e fa             endbr64
112d:   55                      push   rbp
112e:   48 89 e5                mov    rbp,rsp
1131:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
1134:   89 75 f8                mov    DWORD PTR [rbp-0x8],esi
1137:   8b 55 fc                mov    edx,DWORD PTR [rbp-0x4]
113a:   8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
113d:   01 d0                   add    eax,edx
113f:   5d                      pop    rbp
1140:   c3                      ret
```

ASSIGN P1

Obfuscation : VM - Exemple

```
0000000000001129 <func_add>:
1129:   f3 0f 1e fa             endbr64
112d:   55                      push   rbp
112e:   48 89 e5                mov    rbp,rsp
1131:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
1134:   89 75 f8                mov    DWORD PTR [rbp-0x8],esi
1137:   8b 55 fc                mov    edx,DWORD PTR [rbp-0x4]
113a:   8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
113d:   01 d0                   add    eax,edx
113f:   5d                      pop    rbp
1140:   c3                      ret
```

ASSIGN P2

Obfuscation : VM - Exemple

```
0000000000001129 <func_add>:
1129:   f3 0f 1e fa             endbr64
112d:   55                      push   rbp
112e:   48 89 e5                mov    rbp,rsp
1131:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
1134:   89 75 f8                mov    DWORD PTR [rbp-0x8],esi
1137:   8b 55 fc                mov    edx,DWORD PTR [rbp-0x4]
113a:   8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
113d:   01 d0                   add    eax,edx
113f:   5d                      pop    rbp
1140:   c3                      ret
```

ADD

Obfuscation : VM - Exemple

```
0000000000001129 <func_add>:
1129:   f3 0f 1e fa             endbr64
112d:   55                      push   rbp
112e:   48 89 e5                mov    rbp,rsp
1131:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
1134:   89 75 f8                mov    DWORD PTR [rbp-0x8],esi
1137:   8b 55 fc                mov    edx,DWORD PTR [rbp-0x4]
113a:   8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
113d:   01 d0                   add    eax,edx
113f:   5d                      pop    rbp
1140:   c3                      ret
```

RETURN

Obfuscation : VM - Exemple

On peut donc voir cette fonction comme la fonction virtuelle suivante:

def func_obfu_(p1, p2):

assign p1 à mon registre R0

assign p2 à mon registre R1

fait l'addition entre mon registre R0 et mon registre R1

quitte la fonction

opcode: 0x13
syntaxe: 0x13 <numREG> <valeur>
code:
- def assign(reg, val):
    self.registre[reg] = val

Obfuscation : VM - Exemple

On peut donc voir cette fonction comme la fonction virtuelle suivante:

def func_obfu_(p1, p2):

assign p1 à mon registre R0

assign p2 à mon registre R1

fait l'addition entre mon registre R0 et mon registre R1

quitte la fonction

opcode: 0x13
syntaxe: 0x13 <numREG> <valeur>
code:
- def assign(reg, val):
    self.registre[reg] = val

Obfuscation : VM - Exemple

On peut donc voir cette fonction comme la fonction virtuelle suivante:

```
def func_obfu_( p1, p2):
    assign p1 à mon registre R0
    assign p2 à mon registre R1
    fait l'addition entre mon registre R0 et mon registre R1
    quitte la fonction
```

```
opcode: 0x15
syntaxe: 0x15 <numREG> <numREG>
code:
-    def assign( reg1, reg2 ):
         self.registre[reg1] += self.registre[reg2]
```

Obfuscation : VM - Exemple

Une fois l'ensemble des instructions virtuelles implémentées, chacune d'entres elles aura un identifiant (comme un opcode).

On pourra alors écrire cette function avec un nouveau bytecode :

Exemple:

0x13 0x00 0x1234 -> assigne le registre interne R0 avec la valeur 0x1234

0x13 0x01 0x2345 -> assigne le registre interne R1 avec la valeur 0x2345

0x15 0x00 0x01 -> additionne les registres internes R0 et R1 et met le résultat dans R0

Obfuscation : VM - Exemple

La VM consiste à implementer le mécanisme de lecture des bytecodes et à dispatcher les bonnes instructions aux bons traitement.

Le graphe d'une VM ressemble souvent à un switch case qui va justement associer les opérations élémentaires en fonction du bytecode lu.

-> TP 302

Vulnérabilités

101

Use-After-Free : Vulnérabilité très répandue

Vulnérabilité très répandue pour le moment

Pour comprendre ce type de vulnérabilité il faut comprendre des mécanismes d'allocation dynamique:

Allocation sur la stack:

char name[10];

Allocation sur le tas:

char *name = malloc(10);

Use-After-Free : Vulnérabilité très répandue

Les allocateurs sont en general très complexes. Voici leurs objectifs:

- réussir à fournir un bloc de mémoire à la taille souhaitée.
- réussir à fournir un bloc de mémoire à la taille souhaitée rapidement.
- réussir à réutiliser facilement un bloc de mémoire libéré.
- réussir à recycler la mémoire.

Use-After-Free : Vulnérabilité très répandue

Prenons l'exemple suivant:

```
void *buffer = malloc(0x100);

int res = traitement( (void *) buffer );

free( buffer );
```

Use-After-Free : Vulnérabilité très répandue

Prenons l'exemple suivant:

```
void *buffer = malloc(0x100);                < - - - demande de la mémoire au système
int res = traitement( (void *) buffer );     < - - - utilise la mémoire pour faire des choses
free( buffer );                              < - - - rend la mémoire au système
```

Use-After-Free : Vulnérabilité très répandue

Prenons l'exemple suivant:

```
void *buffer = malloc(0x100);                  < - - - demande de la mémoire au système
int res = traitement( (void *) buffer );       < - - - utilise la mémoire pour faire des choses
free( buffer);                                 < - - - rend la mémoire au système
char * UaF = buffer[0];                        < - - - donnée controlee via UaF
```

Use-After-Free : Vulnérabilité très répandue

Video de démontration de Use-After-free

Questions ?

MERCI