Zeste de savoir

Introduction à la rétroingénierie de binaires

12 août 2019

Table des matières

Dans le domaine de l'informatique, et plus précisément celui de la **programmation**, nous développons et utilisons des **programmes**. Lorsque nous en écrivons le code source, nous avons généralement besoin de passer, parmi ces deux étapes, par l'une d'entre elles :

— Interpréter le code source à l'aide d'un programme appelé **interpréteur** ou
— Compiler le code source en **langage machine** pour que celui-ci soit compréhensible directement par notre processeur.

Le point commun de ces deux actions est que l'on exécute, au final, du code binaire, celui directement "compris" par votre processeur.

Les langages de programmation ont été créés pour se rapprocher davantage des dialectes de la langue humaine, parce que le binaire a une réputation d'être incompréhensible (la preuve, c'est une suite de 0 et de 1 !). Mais pour les plus curieux d'entre vous, quelques questions peuvent germer dans votre esprit : comment est-ce que cela fonctionne ? Que contient un fichier binaire qui résulte de la compilation de vos fichiers sources ?

On qualifie la rétroingénierie[1], dans le domaine informatique et plus précisément des binaires, par le processus de **désassembler** un programme afin d'en comprendre son véritable fonctionnement. C'est comme si vous disposiez d'un binaire et que vous voudriez en dresser le code source d'origine, celui qui a été écrit pour en parvenir à ce résultat (du moins les grandes lignes !). C'est une technique **puissante**, qui a permis à l'entreprise Américaine Compaq de copier la micro-puce d'IBM[2] !

Cet article, au trait le plus amical possible, se propose de vous présenter ce qu'est la rétro-ingénierie de binaires et quelles en sont les fins : faire du *reverse engineering* (ou *reverse* tout court pour les intimes) a beaucoup d'applications ! Enfin, nous illustrerons des exemples avec un binaire compilé sous l'architecture intel x86[3] (32 bits), car les programmes compilés pour ces architectures cibles sont encore répandus, pour des raisons de *rétrocompatibilité*. Pour ma part, j'utiliserai Debian. Pensez à installer le paquet `gcc-multilib` si vous êtes sous une architecture amd64, vous compilerez par défaut des binaires x64 (64 bits) et non x86.

Il va sans dire que j'estime que vous possédiez les prérequis suivant : des connaissances en langage C et donc un chouilla sur le fonctionnement de la mémoire, pas plus, pas moins.

0.1. Il était une fois un elfe

Soit le bête programme suivant :

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

int main(void) {
        printf("Hello, I'm a useless program!\n");
        return EXIT_SUCCESS;
}
```

Il a été écrit dans un langage lisible par l'être humain, afin d'être facilement compréhensible et maintenable. Compilons-le sans plus tarder :

```
ge0@kosmik:~/c$ cat test.c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
        printf("Hello, I'm a useless program!\n");
        return EXIT_SUCCESS;
}

ge0@kosmik:~/c$ gcc -Wall -m32 -o  test test.c # -m32 pour compiler
    un binaire au format 32-bit
ge0@kosmik:~/c$ ls
test  test.c
ge0@kosmik:~/c$ ll
total 12
-rwxr-xr-x 1 ge0 ge0 4884 Nov  3 20:27 test
-rw-r--r-- 1 ge0 ge0  126 Nov  3 20:25 test.c
```

Après la compilation, nous obtenons notre fichier binaire `test`. Le comportement qu'on a tous, vis-à-vis de ce fichier, c'est de l'exécuter. A la place, nous allons l'étudier. Première étape : récupérer des informations sur la structure interne de ce fichier ; l'utilitaire `file` est un excellent point de départ :

```
ge0@kosmik:~/c$ file test
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
    dynamically linked (uses shared libs), for GNU/Linux 2.6.26,
    BuildID[sha1]=0x3af49ad1f2359dba04499b6cef5d00561c263198, not
    stripped
```

Nous obtenons une flopée d'informations. Les plus intéressantes sont au début : "ELF 32-bit LSB executable, Intel 80386". Ces informations nous indiquent qu'il s'agit d'un binaire structuré au format **ELF**[4], pour la **Linux Standard Base** (LSB). L'acronyme ELF, signifiant **Executable and Linkable Format** est, comme son nom l'indique, un format de structuration de binaires exécutables. Ce format a notamment en charge d'indiquer la quantité de code et de données binaires à charger en mémoire, à quelle adresse, quelles sont les bibliothèques de code à charger à l'exécution du programme, etc.

Comme vous vous en doutez, il ne s'agit pas du seul format au monde qui soit responsable de faire ça. Sous Windows, on notera l'existence du format PE[5] (Portable Executable) qui structure les binaires exécutables (.exe), les bibliothèques de liens dynamiques (.dll) ou encore les pilotes (.sys). Sous Mac, nous avons même le format mach-O[6]. Ces trois formats sont différents, mais partagent un objectif commun : donner des informations sur le chargement du binaire en mémoire afin de créer un processus qui en sera son image.

Il existe une panoplie d'outils sous Linux pour "jouer" avec un ELF. Mais avant de vous les présenter, j'aimerais vous inviter à "dumper" le début du contenu de votre binaire. Comme si on ouvrait notre fichier `test` avec un éditeur de texte. Allez, faîtes un `cat test`, juste pour voir :

```
ge0@kosmik:~/c$ cat test
[...]
```

On obtient une série de caractères incompréhensibles, dits "non-imprimables". C'est normal, notre fichier est binaire. Mais vous aurez surement remarqué au début les caractères "ELF". Il s'agit d'une partie de la signature de notre fichier. En effet, la signature complète est [0x7f]ELF, et c'est cette signature qui permet à l'utilitaire file de déterminer que ce fichier est un ELF.

Une manipulation un peu plus intelligente : dumpons le début de notre fichier binaire à l'aide de l'outil `hd` (qui est en fait un alias de la commande `hexdump -C`) :

```
ge0@kosmik:~/c$ hd test | head
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00
    |.ELF............|
00000010  02 00 03 00 01 00 00 00  20 83 04 08 34 00 00 00
    |........ ...4...|
00000020  bc 07 00 00 00 00 00 00  34 00 20 00 08 00 28 00
    |........4. ...(.|
00000030  1f 00 1c 00 06 00 00 00  34 00 00 00 34 80 04 08
    |........4...4...|
00000040  34 80 04 08 00 01 00 00  00 01 00 00 05 00 00 00
    |4...............|
00000050  04 00 00 00 03 00 00 00  34 01 00 00 34 81 04 08
    |........4...4...|
00000060  34 81 04 08 13 00 00 00  13 00 00 00 04 00 00 00
    |4...............|
```

```
00000070  01 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08
    |................|
00000080  00 80 04 08 5c 05 00 00  5c 05 00 00 05 00 00 00
    |....\...\.......|
00000090  00 10 00 00 01 00 00 00  5c 05 00 00 5c 95 04 08
    |........\...\...|
```

On remarque que les quatre premiers octets - 7f 45 4c 46 - correspondent bel et bien à la signature de notre fichier. Le reste, bien que lu par un éditeur hexadécimal, ne nous est pas plus compréhensible. Il s'agit ni plus ni moins que du contenu de l'en-tête de notre fichier ELF. Ce contenu peut-être "parsé" par un outil tel que `readelf`[7]. Un fichier ELF contient beaucoup d'informations. Nous aimerions déterminer où se trouve le code binaire de notre fonction main, en supposant que le binaire ait été écrit en C. Car peut-être a-t-il été écrit dans un autre langage, aussi faudrait-il chercher à ce moment-là le **point d'entrée**, c'est-à-dire l'adresse mémoire à laquelle débutera l'exécution du programme.

Il faut savoir qu'un ELF est constitué de **sections**. Une section est désignée par un libellé et caractérisée par une adresse mémoire de début, une taille et son contenu. Chacune d'entre elles voit son contenu offrir une et une seule utilité (contenir du code, des données, des symboles, des fonctions...). Regardons les sections de notre ELF :

```
ge0@kosmik:~/c$ readelf -S test
There are 29 section headers, starting at offset 0x7ac:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES
    Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00
    0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00
    A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00
    A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00
    A  0   0  4
  [ 4] .hash             HASH            0804818c 00018c 000028 04
    A  6   0  4
  [ 5] .gnu.hash         GNU_HASH        080481b4 0001b4 000020 04
    A  6   0  4
  [ 6] .dynsym           DYNSYM          080481d4 0001d4 000050 10
    A  7   1  4
  [ 7] .dynstr           STRTAB          08048224 000224 00004a 00
    A  0   0  1
  [ 8] .gnu.version      VERSYM          0804826e 00026e 00000a 02
    A  6   0  2
  [ 9] .gnu.version_r    VERNEED         08048278 000278 000020 00
    A  7   1  4
```

```
  [10] .rel.dyn          REL             08048298 000298 000008 08
    A  6   0  4
  [11] .rel.plt          REL             080482a0 0002a0 000018 08
    A  6  13  4
  [12] .init             PROGBITS        080482b8 0002b8 000026 00
    AX  0   0  4
  [13] .plt              PROGBITS        080482e0 0002e0 000040 04
    AX  0   0 16
  [14] .text             PROGBITS        08048320 000320 000180 00
    AX  0   0 16
  [15] .fini             PROGBITS        080484a0 0004a0 000017 00
    AX  0   0  4
  [16] .rodata           PROGBITS        080484b8 0004b8 000026 00
    A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        080484e0 0004e0 00001c 00
    A  0   0  4
  [18] .eh_frame         PROGBITS        080484fc 0004fc 000060 00
    A  0   0  4
  [19] .init_array       INIT_ARRAY      0804955c 00055c 000004 00
    WA  0   0  4
  [20] .fini_array       FINI_ARRAY      08049560 000560 000004 00
    WA  0   0  4
  [21] .jcr              PROGBITS        08049564 000564 000004 00
    WA  0   0  4
  [22] .dynamic          DYNAMIC         08049568 000568 0000f0 08
    WA  7   0  4
  [23] .got              PROGBITS        08049658 000658 000004 04
    WA  0   0  4
  [24] .got.plt          PROGBITS        0804965c 00065c 000018 04
    WA  0   0  4
  [25] .data             PROGBITS        08049674 000674 000008 00
    WA  0   0  4
  [26] .bss              NOBITS          0804967c 00067c 000004 00
    WA  0   0  4
  [27] .comment          PROGBITS        00000000 00067c 000038 01
    MS  0   0  1
  [28] .shstrtab         STRTAB          00000000 0006b4 0000f6 00
    0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x
    (unknown)
  O (extra OS processing required) o (OS specific), p (processor
    specific)
ge0@kosmik:~/c$
```

L'option -S permet de lister les sections présentes au sein d'un binaire ELF.

Le résultat peut, bien évidemment, changer d'un poste à un autre. J'attire votre attention sur la section .text - la numéro 14. Ce nom est généralement attribué aux sections qui vont contenir

le code exécutable de notre binaire. Par ailleurs, dans la colonne **Flg**, qui signifie "Flags", nous noterons la présence du X qui signifie eXecutable. C'est-à-dire que le code binaire qui s'y trouve peut-être exécuté.

Il serait intéressant d'y jeter un oeil et il y a plusieurs façons de le faire. Si vous possédez le code source d'origine du programme, vous pouvez appliquer l'option -S à gcc pour vous générer un fichier en langage d'assemblage, qui est le résultat de la **compilation** de votre fichier source.

```
ge0@kosmik:~/c$ gcc -m32 -S test.c
ge0@kosmik:~/c$ cat test.s
        .file   "test.c"
        .section        .rodata
.LC0:
        .string "Hello, I'm a useless program!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
        .section        .note.GNU-stack,"",@progbits
```

Mais dans le cas où vous ne possédez pas le code source du programme et que vous n'êtes qu'en possession du binaire - c'est un peu le principe de cet article - l'outil `objdump` vous ravira. Il permet, entre autres, de désassembler votre fichier binaire. Désassemblons la section .text dont nous parlions :

```
ge0@kosmik:~/c$ objdump -d test --section=.text

```

```
 3 test:     file format elf32-i386
 4
 5
 6 Disassembly of section .text:
 7
 8 08048320 <.text>:
 9  8048320:        31 ed                           xor    %ebp,%ebp
10  8048322:        5e                              pop    %esi
11  8048323:        89 e1                           mov    %esp,%ecx
12  8048325:        83 e4 f0                        and
       $0xfffffff0,%esp
13  8048328:        50                              push   %eax
14  8048329:        54                              push   %esp
15  804832a:        52                              push   %edx
16  804832b:        68 30 84 04 08                  push   $0x8048430
17  8048330:        68 40 84 04 08                  push   $0x8048440
18  8048335:        51                              push   %ecx
19  8048336:        56                              push   %esi
20  8048337:        68 0c 84 04 08                  push   $0x804840c
21  804833c:        e8 cf ff ff ff                  call   8048310
       <__libc_start_main@plt>
22  8048341:        f4                              hlt
23  8048342:        90                              nop
24  8048343:        90                              nop
25  8048344:        90                              nop
26  8048345:        90                              nop
27  8048346:        90                              nop
28  8048347:        90                              nop
29  8048348:        90                              nop
30  8048349:        90                              nop
31  804834a:        90                              nop
32  804834b:        90                              nop
33  804834c:        90                              nop
34  804834d:        90                              nop
35  804834e:        90                              nop
36  804834f:        90                              nop
37  8048350:        b8 7f 96 04 08                  mov
       $0x804967f,%eax
38  8048355:        2d 7c 96 04 08                  sub
       $0x804967c,%eax
39  804835a:        83 f8 06                        cmp    $0x6,%eax
40  804835d:        77 02                           ja     8048361
       <__libc_start_main@plt+0x51>
41  804835f:        f3 c3                           repz ret
42  8048361:        b8 00 00 00 00                  mov    $0x0,%eax
43  8048366:        85 c0                           test   %eax,%eax
44  8048368:        74 f5                           je     804835f
       <__libc_start_main@plt+0x4f>
45  804836a:        55                              push   %ebp
46  804836b:        89 e5                           mov    %esp,%ebp
```

```
804836d:        83 ec 18                    sub    $0x18,%esp
8048370:        c7 04 24 7c 96 04 08        movl
   $0x804967c,(%esp)
8048377:        ff d0                       call   *%eax
8048379:        c9                          leave
804837a:        c3                          ret
804837b:        90                          nop
804837c:        8d 74 26 00                 lea
   0x0(%esi,%eiz,1),%esi
8048380:        b8 7c 96 04 08              mov
   $0x804967c,%eax
8048385:        2d 7c 96 04 08              sub
   $0x804967c,%eax
804838a:        c1 f8 02                    sar    $0x2,%eax
804838d:        89 c2                       mov    %eax,%edx
804838f:        c1 ea 1f                    shr    $0x1f,%edx
8048392:        01 d0                       add    %edx,%eax
8048394:        d1 f8                       sar    %eax
8048396:        75 02                       jne    804839a
   <__libc_start_main@plt+0x8a>
8048398:        f3 c3                       repz ret
804839a:        ba 00 00 00 00              mov    $0x0,%edx
804839f:        85 d2                       test   %edx,%edx
80483a1:        74 f5                       je     8048398
   <__libc_start_main@plt+0x88>
80483a3:        55                          push   %ebp
80483a4:        89 e5                       mov    %esp,%ebp
80483a6:        83 ec 18                    sub    $0x18,%esp
80483a9:        89 44 24 04                 mov    %eax,0x4(%esp)
80483ad:        c7 04 24 7c 96 04 08        movl
   $0x804967c,(%esp)
80483b4:        ff d2                       call   *%edx
80483b6:        c9                          leave
80483b7:        c3                          ret
80483b8:        90                          nop
80483b9:        8d b4 26 00 00 00 00        lea
   0x0(%esi,%eiz,1),%esi
80483c0:        80 3d 7c 96 04 08 00        cmpb   $0x0,0x804967c
80483c7:        75 13                       jne    80483dc
   <__libc_start_main@plt+0xcc>
80483c9:        55                          push   %ebp
80483ca:        89 e5                       mov    %esp,%ebp
80483cc:        83 ec 08                    sub    $0x8,%esp
80483cf:        e8 7c ff ff ff              call   8048350
   <__libc_start_main@plt+0x40>
80483d4:        c6 05 7c 96 04 08 01        movb   $0x1,0x804967c
80483db:        c9                          leave
80483dc:        f3 c3                       repz ret
80483de:        66 90                       xchg   %ax,%ax
80483e0:        a1 64 95 04 08              mov    0x8049564,%eax
```

```
80483e5:        85 c0                           test   %eax,%eax
80483e7:        74 1e                           je     8048407
   <__libc_start_main@plt+0xf7>
80483e9:        b8 00 00 00 00                  mov    $0x0,%eax
80483ee:        85 c0                           test   %eax,%eax
80483f0:        74 15                           je     8048407
   <__libc_start_main@plt+0xf7>
80483f2:        55                              push   %ebp
80483f3:        89 e5                           mov    %esp,%ebp
80483f5:        83 ec 18                        sub    $0x18,%esp
80483f8:        c7 04 24 64 95 04 08            movl
   $0x8049564,(%esp)
80483ff:        ff d0                           call   *%eax
8048401:        c9                              leave
8048402:        e9 79 ff ff ff                  jmp    8048380
   <__libc_start_main@plt+0x70>
8048407:        e9 74 ff ff ff                  jmp    8048380
   <__libc_start_main@plt+0x70>
804840c:        55                              push   %ebp
804840d:        89 e5                           mov    %esp,%ebp
804840f:        83 e4 f0                        and
   $0xfffffff0,%esp
8048412:        83 ec 10                        sub    $0x10,%esp
8048415:        c7 04 24 c0 84 04 08            movl
   $0x80484c0,(%esp)
804841c:        e8 cf fe ff ff                  call   80482f0
   <puts@plt>
8048421:        b8 00 00 00 00                  mov    $0x0,%eax
8048426:        c9                              leave
8048427:        c3                              ret
8048428:        90                              nop
8048429:        90                              nop
804842a:        90                              nop
804842b:        90                              nop
804842c:        90                              nop
804842d:        90                              nop
804842e:        90                              nop
804842f:        90                              nop
8048430:        55                              push   %ebp
8048431:        89 e5                           mov    %esp,%ebp
8048433:        5d                              pop    %ebp
8048434:        c3                              ret
8048435:        8d 74 26 00                     lea
   0x0(%esi,%eiz,1),%esi
8048439:        8d bc 27 00 00 00 00            lea
   0x0(%edi,%eiz,1),%edi
8048440:        55                              push   %ebp
8048441:        89 e5                           mov    %esp,%ebp
8048443:        57                              push   %edi
8048444:        56                              push   %esi
```

```
8048445:        53                              push   %ebx
8048446:        e8 4f 00 00 00                  call   804849a
   <__libc_start_main@plt+0x18a>
804844b:        81 c3 11 12 00 00               add    $0x1211,%ebx
8048451:        83 ec 1c                        sub    $0x1c,%esp
8048454:        e8 5f fe ff ff                  call   80482b8
   <puts@plt-0x38>
8048459:        8d bb 04 ff ff ff               lea
   -0xfc(%ebx),%edi
804845f:        8d 83 00 ff ff ff               lea
   -0x100(%ebx),%eax
8048465:        29 c7                           sub    %eax,%edi
8048467:        c1 ff 02                        sar    $0x2,%edi
804846a:        85 ff                           test   %edi,%edi
804846c:        74 24                           je     8048492
   <__libc_start_main@plt+0x182>
804846e:        31 f6                           xor    %esi,%esi
8048470:        8b 45 10                        mov
   0x10(%ebp),%eax
8048473:        89 44 24 08                     mov    %eax,0x8(%esp)
8048477:        8b 45 0c                        mov    0xc(%ebp),%eax
804847a:        89 44 24 04                     mov    %eax,0x4(%esp)
804847e:        8b 45 08                        mov    0x8(%ebp),%eax
8048481:        89 04 24                        mov    %eax,(%esp)
8048484:        ff 94 b3 00 ff ff ff            call
   *-0x100(%ebx,%esi,4)
804848b:        83 c6 01                        add    $0x1,%esi
804848e:        39 fe                           cmp    %edi,%esi
8048490:        72 de                           jb     8048470
   <__libc_start_main@plt+0x160>
8048492:        83 c4 1c                        add    $0x1c,%esp
8048495:        5b                              pop    %ebx
8048496:        5e                              pop    %esi
8048497:        5f                              pop    %edi
8048498:        5d                              pop    %ebp
8048499:        c3                              ret
804849a:        8b 1c 24                        mov    (%esp),%ebx
804849d:        c3                              ret
804849e:        90                              nop
804849f:        90                              nop
```

L'option -d indique à l'outil de désassembler les sections exécutables du binaire, et nous lui avons indiqué de désassembler seulement la section .text.

Le résultat est à peine aussi parlant que celui que nous avions obtenu à l'aide de gcc. Sur la gauche, nous avons les adresses mémoires auxquelles sont placés les codes binaires. Ensuite nous avons leur valeur et, enfin, sur la droite, nous avons... Du langage d'assemblage. Les codes binaires dont je viens de parler sont en fait des **instructions machine**.

Bien qu'il ait l'air un poil plus compréhensible que son équivalent binaire, il reste assez obscur pour qui n'a jamais pratiqué le langage d'assemblage.

0.2. L'architecture x86

On désigne en partie, par l'architecture x86, l'ensemble des **registres** et le **jeu d'instructions** exécutables par les processeurs de cette gamme. Ces termes peuvent vous paraitre nouveaux, définissons-les :

— registre : il s'agit d'une mémoire petite mais très rapide, directement présente à l'intérieur du processeur. Sa taille est de l'ordre de quelques octets seulement.
— jeu d'instructions : c'est l'ensemble des instructions exécutables par le processeur.

Une instruction binaire est généralement constituée de ce qu'on appelle un **opcode**, qui est la contraction de *operation code*. Un **opcode** est généralement codé sur un octet, mais il existe des opcodes à deux octets pour les jeux d'instruction étendus.

0.3. Les registres x86

Les registres, comme je le disais, sont des petites mémoires qui vont servir à contenir des valeurs de l'ordre de quelques octets. Ces valeurs peuvent représenter tout et n'importe quoi : le résultat d'une expression arithmétique / logique, la valeur de retour d'une fonction, une adresse mémoire... Vous devez juste retenir qu'elle est petite, contrairement à la RAM qui est de l'ordre de quelques Giga-octets ou le disque dur mécanique qui est de l'ordre de quelques Téra-octets.

L'architecture x86 recense huit registres à usages généraux : eax, ebx, ecx, edx, edi, esi, ebp, esp. Ils respectent cependant une convention d'utilisation au sein des programmes informatiques. En ce qui concerne les quatre premiers :

— **eax** : registre d'accumulation étendu : il est utilisé pour stocker le résultat d'opérations arithmétiques ou les valeurs de retour des routines appelées
— **ebx** : registre de base étendu : il est utilisé dans le cadre d'accès à la mémoire via l'adressage d'une donnée. Le registre de base contiendra une adresse mémoire de *base*, tandis que nous fournirons à côté un index, un offset, etc.
— **ecx** : registre de comptage étendu : il est utilisé par des opérations qui gèrent l'exécution de boucles.
— **edx** : registre de données étendu : il est utilisé pour stocker des données en tout genre.

Ces registres ont pour taille 32 bits, soit 4 octets, et peuvent-être déclinés en sous-registres de 16 bits AX, BX, CX et DX qui correspondront, respectivement, à la partie inférieure des registres EAX, EBX, ECX et EDX. Enfin, ces mêmes registres de AX à DX peuvent être déclinés en deux parties :

— la partie supérieure : AH, BH, CH et DH ;
— la partie inférieure : AL, BL, CL et DL.

Un schéma valant mieux qu'un long discours, la figure ci-dessous illustre nos propos. Un registre étendu offre l'accès des bits 0 à 31, la première déclinaison permet l'accès aux bits 0 à 15 (16 bits) et, enfin, les deux dernières déclinaisons permettent l'accès aux bits 0 à 7 (partie inférieure) et 8 à 15 (partie supérieure).

http://zestedesavoir.com/media/galleries/1435/

La modification d'une déclinaison impacte sur la valeur du registre général. Par exemple, si l'on modifie le contenu de AL, alors AX et EAX seront modifiés. Si EAX vaut 0xdeadbeef et que je mets le contenu de AL à 0x0d, EAX vaudra alors 0xdeadbe0d. L'utilité d'avoir des registres déclinés permet au processeur de faire des opérations sur des plus petites quantités de mémoire que 32 bits (16 bits et 8 bits, donc). A noter qu'il n'est pas possible d'accéder *directement* aux 16 bits supérieurs des registres généraux.

Les registres **esi** et **edi** sont souvent utilisés conjointement pour manipuler des données d'une source (esi) à une destination (edi), par exemple les copies.

Enfin, les registre **ebp** (extended base pointer) et **esp** (extended stack pointer) servent à délimiter ce qu'on appelle le **cadre de pile courant**[8], communément appelé **stack frame**. Nous y reviendrons.

Il existe aussi le registre **eip** (extended instruction pointer) qui contient l'adresse de la prochaine instruction à exécuter. Ce registre, non directement manipulable, est très important puisqu'il permet au processeur, entre autres, d'exécuter des instructions qui ne sont pas forcément séquentielles (appel d'une fonction distante, branchement (in)conditionnel avec des `if` et `else`, ...).

Vous vous doutez que l'architecture x86 est une myriade de fois plus complexe que ce que je viens d'expliquer. Je n'ai, par exemple, pas parlé des registres de segments ou encore de l'extension x87 pour les nombres flottants. Je souhaite en effet ne pas vous noyer d'informations inutiles dans le cadre de cet article, mais vous êtes invités à vous renseigner sur le sujet. ;-)

Voyons maintenant de plus près à quoi ressemble une instruction. Prenons-en une dans le listing de désassemblage plus haut, à savoir celle-ci :

```
1 8048451:        83 ec 1c                    sub    $0x1c,%esp
```

Cette instruction est présente à l'adresse 0x08048451. Elle mesure trois octets et, d'après le mnémonique, elle effectue une soustraction ("sub") de 0x1c au registre esp. Il existe également des instructions qui ne font rien, par exemple :

```
1 804849e:        90                          nop
```

Le mnémonique "nop" signifie "no operation". On peut penser que cette instruction est inutile, mais la réalité est toute autre : elle a été introduite pour des raisons de bourrage, ou **padding**. Elle peut aussi servir dans d'autres situations que nous n'étudierons pas ici (je vois déjà les mordus d'exploitation de faille s'exciter !).

0.4. La pile d'exécution

Un dernier point théorique que j'aimerais aborder avant de vous montrer comment faire joujou avec avec votre binaire test : la pile d'exécution. Je la qualifiais de **stack frame** plus haut. En général, on l'appelle la **stack**.

Comme je l'ai dit plus haut, il s'agit d'une zone de mémoire à structure LIFO (Last In, First Out). On y empile et dépile des valeurs. Il est possible d'accéder à n'importe quelle valeur par lecture et écriture, certes, mais si vous voulez dépiler une valeur qui n'est pas au-dessus de la pile, il va falloir dépiler les autres ! La pile est délimitée par deux registres : ebp et esp. C'est-à-dire que ces registres, sur 32 bits, auront pour valeur une **adresse mémoire**. Et ces deux valeurs délimiteront ce qu'on appelle le **cadre de pile**.

La pile d'exécution, au sein d'un programme, est donc une zone de mémoire - c'est pour ça qu'elle est accessible en lecture et écriture comme je le disais - et est en constante évolution. Elle va contenir toutes les variables locales de votre programme ainsi que les sauvegardes des adresses de retour aux routines appelantes. Par exemple, dans notre programme, nous appelons la fonction `printf()`. Nous allons donc charger, dans le registre **eip**, l'adresse de la fonction `printf` et ainsi exécuter son code. Mais une fois que cette fonction a déroulé l'intégralité de son code, comment faire pour revenir à la fonction `main` ? Et bien il suffira de récupérer l'adresse mémoire que nous aurons déposée sur la stack et la remettre dans eip, ainsi nous retournerons à la fonction `main`. La pile est également utile pour le passage d'arguments à une fonction, mais ce n'est pas le seul moyen de procéder. Certaines conventions d'appel imposent l'utilisation de registres mais, dans notre exemple, nous utilisons exclusivement la stack pour passer nos arguments à une fonction.

Au fur et à mesure que l'on appelle des fonctions et alloue de la mémoire statiquement (à ne surtout pas confondre avec le mot-clé `static` en C, car ici je parle des variables locales, entendons-nous bien) au sein d'une fonction, la pile est amenée à grandir. Dans l'architecture x86, elle grandit vers les **adresses basses**, ce qui veut dire que plus nous allouons de la mémoire, **plus la valeur du registre esp décroit**. Ainsi, **esp** aura une valeur toujours inférieure au registre **ebp** (normalement...).

Afin que chaque fonction ait son propre cadre de pile, elle exécute ce qu'on appelle un **prologue** et un **épilogue**. Le prologue va sauvegarder le cadre de pile courant pour mettre en place le cadre de pile de la fonction courante, et l'épilogue va l'enlever pour restaurer l'ancien.

Supposons, dans notre exemple, que nous sommes à l'aurore de la fonction `main`. Nous ne savons pas comment la pile est conçue, ni ce qu'elle contient, si ce n'est la sauvegarde d'EIP qui permet de revenir à la fonction qui a appelé `main`. Le but est de ne pas perturber le fonctionnement de la fonction appelante (il ne faut surtout pas modifier son cadre de pile). Supposons l'état de la pile à ce moment-ci :

```
  Adresses basses
+------------------+ <-- esp
|  sauvegarde EIP  |
+------------------+
|     données      |
|  de la fonction  |
```

```
|    appelante     |
|                  |
+------------------+ <-- ebp
  Adresses hautes
```

La fonction appelée va sauvegarder la valeur du registre de base :

```
  Adresses basses
+------------------+ <-- esp
|  sauvegarde EBP  |
+------------------+
|  sauvegarde EIP  |
+------------------+
|     données      |
|  de la fonction  |
|    appelante     |
|                  |
+------------------+ <-- ebp
  Adresses hautes
```

Elle va ensuite placer dans ebp la valeur d'esp : on aura donc une stack "vide" :

```
  Adresses basses
+------------------+ <-- esp <-- ebp (la stack est "vide")
|  sauvegarde EBP  |
+------------------+
|  sauvegarde EIP  |
+------------------+
|     données      |
|  de la fonction  |
|    appelante     |
|                  |
+------------------+
  Adresses hautes
```

Enfin, elle va pouvoir soustraire la valeur d'esp pour allouer de la mémoire pour le nouveau cadre de pile :

```
  Adresses basses
+------------------+ <-- esp
|  données de la   |
| fonction appelée |
+------------------+ <-- ebp
|  sauvegarde EBP  |
```

```
+------------------+
|  sauvegarde EIP  |
+------------------+
|     données      |
|  de la fonction  |
|    appelante     |
|                  |
+------------------+
  Adresses hautes
```

Comment cela se passe une fois que la fonction appelée a terminé ? Eh bien elle fait le cheminement inverse : elle place cette fois-ci ebp dans la valeur d'esp :

```
  Adresses basses
+------------------+
|  données de la   |
| fonction appelée |
+------------------+ <-- ebp <-- esp
|  sauvegarde EBP  |
+------------------+
|  sauvegarde EIP  |
+------------------+
|     données      |
|  de la fonction  |
|    appelante     |
|                  |
+------------------+
  Adresses hautes
```

On dépile la sauvegarde d'EBP pour la placer dans le registre ebp (c'est une bête restauration) :

```
  Adresses basses
+------------------+ <-- esp
|  sauvegarde EIP  |
+------------------+
|     données      |
|  de la fonction  |
|    appelante     |
|                  |
+------------------+ <-- ebp
  Adresses hautes
```

Et nous en sommes revenus au point initial : la fonction appelante a ses données intactes et les données de la fonction appelée seront perdues et réécrasées par une autre fonction appelée. C'est pourquoi il ne faut jamais retourner de tableau en C : on retourne en fait un pointeur vers des données dans la pile qui seront tôt ou tard réutilisées sans scrupule !

0.5. Déboguage

Après cette longue explication théorique, je vous propose une petite partie pratique. La meilleure façon de se rendre compte du fonctionnement de la pile d'exécution est de déboguer notre programme pas à pas, de manière dynamique. Nous avions vu en effet l'utilisation d'objdump, qui est un outil primaire d'analyste statique, mais il est parfois plaisant d'utiliser un débogueur pour tracer le code dynamiquement. Nous utiliserons le célèbre débogueur GNU qu'est `gdb`[9].

```
ge0@kosmik:~/c$ gdb ./test
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
    <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
    copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/ge0/c/test...(no debugging symbols
    found)...done.
(gdb)
```

L'outil possède également une fonctionnalité de désassemblage. Désassemblons la fonction `main` à l'aide de la commande `disass` :

```
(gdb) disass main
Dump of assembler code for function main:
   0x0804840c <+0>:     push   %ebp
   0x0804840d <+1>:     mov    %esp,%ebp
   0x0804840f <+3>:     and    $0xfffffff0,%esp
   0x08048412 <+6>:     sub    $0x10,%esp
   0x08048415 <+9>:     movl   $0x80484c0,(%esp)
   0x0804841c <+16>:    call   0x80482f0 <puts@plt>
   0x08048421 <+21>:    mov    $0x0,%eax
   0x08048426 <+26>:    leave
   0x08048427 <+27>:    ret
End of assembler dump.
```

Nous avons 9 instructions dans la fonction main. En guise de pratique pour cet article sur l'initiation à la rétro-ingénierie, nous nous contenterons d'étudier cette fonction seulement. C'est un excellent départ !

Avant de continuer, j'aimerais attirer votre attention sur un point particulier : les mnémoniques. Sachez qu'il existe plusieurs syntaxes de mnémoniques pour représenter un opcode. Celle

par défaut, illustrée dans cet article, est la syntaxe **AT&T**. La source est spécifiée avant la destination. Exemple :

```
1 mov $4, %eax ; place la valeur 4 dans le registre eax
```

Il existe aussi une autre syntaxe (que je préfère de loin) qui est la syntaxe d'Intel. Ici, la destination est spécifiée avant la source :

```
1 mov [eax], 0xbaadf00d ; place la valeur 0xbaadfood à l'adresse
      mémoire contenue dans eax
```

Cette parenthèse de faite, revenons-en à notre fonction main. Les deux premières instructions, `push %ebp` et `mov %esp, %ebp`, constituent le prologue dont je vous parlais plus haut : ce sont ces instructions qui vont mettre en place le cadre de pile.

L'instruction `push %ebp` a pour fonction de "pousser" sur la pile la valeur du registre ebp. Elle va donc prendre la valeur présente dans le registre ebp et la déposer sur la pile (rappelez-vous, c'est une structure de type LIFO). Nous allons donc exécuter cette instruction et vérifier son impact sur le programme. Nous allons poser un point d'arrêt sur l'instruction et lancer le programme :

```
1  (gdb) b *main+0
2  Breakpoint 1 at 0x804840c: file test.c, line 4.
3  (gdb) r
4  Starting program: /home/ge0/c/test
5
6  Breakpoint 1, 0x0804840c in main ()
7
8  (gdb) x/i $eip
9  => 0x804840c <main>:        push   %ebp
10 (gdb)
```

L'instruction x permet d'afficher une valeur au format choisi. Le format "i" est le format "instruction". Affichons la valeur d'ebp, puis le sommet de la pile (2 valeurs suffiront) :

```
1 (gdb) p $ebp
2 $1 = (void *) 0xffffd538
3 (gdb) x/2x $esp
4 0xffffd4bc:        0xf7e77e46        0x00000001
```

Exécutons ensuite l'instruction et affichons de nouveau le sommet de la pile :

```
(gdb) nexti
0x0804840d in main ()
(gdb) x/i $eip
=> 0x804840d <main+1>:        mov    %esp,%ebp
(gdb) x/2x $esp
0xffffd4b8:        0xffffd538        0xf7e77e46
(gdb)
```

Nous constatons deux choses :

— La première, qui est la plus évidente, c'est que la valeur d'ebp (0xffffd538) a bien été positionnée au sommet de la pile, l'autre valeur (0xf7e77e46) se retrouvant en dessous.
— La seconde, c'est que la valeur d'esp a changé ! Elle est passée de 0xffffd4bc à 0xffffd4b8, soit de 4 octets de moins. Rappelez-vous, la pile croit vers les adresses basses, c'est donc normal.

L'instruction suivante, `mov %esp,%ebp`, met à même niveau les registres esp et ebp pour constituer une pile vide. Elle va placer, dans ebp, la valeur d'esp (rappelez-vous, dans la syntaxe AT&T, c'est la source qui est spécifiée avant la destination). Viennent ensuite ces deux instructions :

```
and    $0xfffffff0,%esp
sub    $0x10,%esp
```

La première instruction, à l'aide d'un ET logique et d'un masque, force l'alignement du registre esp sur un multiple de 16 octets. La seconde permet d'ajouter à notre pile une taille de 16 octets. Cette mémoire a été choisie par notre compilateur et ne servira pas nécessairement à être utilisée intégralement.

Avançons à ces deux instructions qui nous intéressent davantage :

```
0x08048415 <+9>:        movl   $0x80484c0,(%esp)
0x0804841c <+16>:        call   0x80482f0 <puts@plt>
```

Pour cela, posons un point d'arrêt sur *main+9 et continuons l'exécution du programme :

```
(gdb) b *main+9
Breakpoint 2, 0x08048415 in main ()
(gdb) c
Continuing.

(gdb) x/i $eip
=> 0x8048415 <main+9>:        movl   $0x80484c0,(%esp)
```

L'instruction `movl $0x80484c0,(%esp)` va placer, à l'adresse pointée par %esp (car le registre est entre parenthèses), la valeur $0x80484c0. Ici, au lieu de déposer une valeur sur la pile, on va directement écrire à l'emplacement pointé par le sommet de la pile. C'est comme si, dans une pile d'assiettes, on prenait l'assiette du dessus, qu'on réécrivait son étiquette (ou que sais-je) et qu'on la remettait en place, sans vraiment la déplacer...

On constate que cette valeur - $0x80484c0 - ressemble fortement à une adresse mémoire si l'on relit le listing d'objdump. Il s'agit en fait de l'emplacement notre chaîne de caractères, celle que nous avons donnée à la fonction `printf` dans notre programme. Vous vous souvenez quand je vous disais que la pile d'exécution servait à passer des arguments à la routine appelée ? Ici, c'est exactement ce qu'il se passe : on va écrire, sur le sommet de la pile (*confer* mon explication foireuse plus haut), la valeur qui correspond au pointeur sur notre chaîne de caractères. Pour preuve :

```
1 (gdb) printf "%s\n",0x80484c0
2 Hello, I'm a useless program!
```

Il s'agit bien de notre chaîne de caractères.

La prochaine instruction, comme le montre le listing de gdb, appellera la fonction `puts()`. Le fait qu'elle n'appelle pas `printf()` à la place est que notre chaîne ne contient aucune directive de formatage, de fait le compilateur a remplacé l'appel à `printf()` par `puts()` pour des raisons de performance.

L'instruction `call 0x80482f0` va permettre d'exécuter le code présent à l'adresse fournie en opérande. Pour cela, elle va d'abord déposer, sur la pile d'exécution, l'adresse de l'instruction qui suit, à savoir `0x08048421 <+21>: mov $0x0,%eax` afin de pouvoir y revenir une fois la fonction appelée intégralement déroulée. Après avoir empilé cette adresse, l'instruction chargera dans eip l'adresse $0x80484c0 et le code qui s'y trouve (à savoir, le code de `puts()`) sera exécuté.

Il nous reste trois instructions à analyser :

```
1 0x08048421 <+21>:        mov    $0x0,%eax
2 0x08048426 <+26>:        leave
3 0x08048427 <+27>:        ret
```

Elles sont très simples. La première, `mov $0x0,%eax`, place dans le registre eax la valeur 0. Comme je le disais plus haut, le registre eax sert à contenir le résultat d'opérations mathématiques ou, entre autres, la valeur de retour d'une fonction. Il ne s'agit ni plus ni moins que le code de retour de notre fonction main, la traduction de l'instruction C `return 0;`. "Tout s'est bien passé".

Pour finir, les instructions `leave` et `ret` constituent l'épilogue de notre programme.

L'instruction `leave` va remettre, d'une part, la valeur d'esp à celle d'ebp, réinitialisant ainsi le cadre de pile et, d'autre part, dépiler la sauvegarde du pointeur vers la base de pile pour la restaurer dans le registre adéquat. En résumé, elle détruit le cadre de pile actuel et restaure l'ancien, tout ça en une instruction. C'est l'équivalent de cette série de deux instructions :

```
1 mov %ebp, %esp ; place dans esp la valeur d'ebp, ainsi esp = ebp
2 pop %ebp ; dépile la valeur pointée par esp dans ebp, et en théorie
      cette valeur est la sauvegarde de la base du cadre de pile
      précédent
```

Les explications peuvent paraître confuses, n'hésitez pas à exécuter du code binaire pas à pas pour vraiment vous rendre compte de comment cela fonctionne.

L'instruction `ret`, quant à elle, dépile la sauvegarde du registre eip qui a été empilée par la fonction appelante (qui a fait un call) et ainsi la fonction main() se termine.

Je reconnais que cela fait beaucoup de choses d'un coup et que ce n'est qu'un ridicule début, mais le mieux est de s'exercer sur des programmes compilés.

0.6. Conclusion

La rétroingénierie est une technique puissante. Elle permet, pour quelqu'un qui ne dispose pas des sources d'un programme, d'en connaître son fonctionnement interne. Il s'agit ni plus ni moins d'une analyse en boîte noire : on ne connait pas les spécifications du logiciel et on utilise les outils à disposition pour l'analyser. Ces techniques sont utilisées par les "crackers" qui produisent des cracks ou des keygens pour des sharewares (comment pensez-vous que vous pouvez obtenir Adobe CS6 gratuitement ?!), ou encore par des hackers pour analyser la sécurité des logiciels dont ils ne disposent pas du code source (analyse en "boîte noire") ou, encore, l'analyse des *malwares* ou *rootkits* pour produire les correctifs qui vont bien. C'est un domaine passionnant.

Il existe beaucoup de contremesures qui consistent à ralentir le travail d'un *reverse engineer*, à savoir l'obfuscation de code binaire pour le rendre moins compréhensible ou, encore, l'utilisation de techniques d'anti-débogage si l'analyse est dynamique. Mais tout ça est hors de la portée de cette modeste introduction.

Pour progresser en toute légalité dans le domaine de la rétro-ingénierie, il est mis à disposition des challenges nommés *crackmes*. Un site connu, http ://crackmes.one/ [10], est une référence dans le domaine et en met plusieurs à disposition.

Dans tous les cas, armez-vous de patience. Bonne chance !

0.7. Références

1. Rétroingénierie : Article sur wikipédia (fr)
2. Compaq : Article sur wikipédia (en)
3. Langage d'assemblage x86 : Article sur wikipédia (en)
4. Executable and Linkable Format : Article sur wikipédia (en)
5. Portable Executable : Article sur wikipédia (fr)
6. mach-O : Article sur wikipédia (en)
7. readelf : http ://sourceware.org/binutils/docs/binutils/readelf.html
8. Pile (informatique) : Article sur wikipédia (fr)
9. gdb : GDB : The GNU Project Debugger
10. crackme.one : Challenges de crackmes