LIEF : Développement d'une bibliothèque d'instrumentation des formats binaires

Rapport de stage

Quarkslab
71 – 73 avenue des Ternes
75017 Paris
France

Table des matières

1. Introduction

Mon stage s'est déroulé au sein de Quarkslab, une entreprise spécialisée dans la sécurité informatique et la protection d'informations.
Elle a été crée en 2011 par Frédéric Raynal et il y a actuellement une quarantaine de salariés.
J'ai précédemment effectué deux autres stages dans cette entreprise sur des sujets différents. Le premier sur le `JTAG` et le deuxième sur Epona, un logiciel de protection de programmes. Le stage que j'ai réalisé porte sur le développement d'une bibliothèque d'instrumentation des formats exécutables. Je suis encadré par M. Serge Guelton ingénieur R&D à Quarkslab et également responsable du projet Epona. Il était mon tuteur pour le stage sur Epona.
L'entreprise est divisée en différent groupes en rapport avec la sécurité informatique et je fais parti de l'équipe rétro ingénierie (*reverse engineering*).

Dans une première partie je présenterai certaines subtilités des formats binaires. Dans une deuxième partie j'expliquerai les différentes solutions techniques et les choix qui ont été pris pour le développement du projet et dans la dernière partie je détaillerai des aspects de l'architecture de LIEF.

1.1 Présentation du projet

Une des tâches d'un compilateur est de transformer le code source d'un programme en code assembleur compréhensible par la machine cible. Ça aboutit à un fichier exécutable, `foo.exe` par exemple.
Ce code assembleur n'est pas tel quel dans le fichier, il est enveloppé par un format.
Un format d'exécutable est en quelque sorte un conteneur qui contient en premier lieu le code assembleur à exécuter mais également des informations sur la façon d'exécuter ce code.
On peut faire une analogie avec une lettre postale, l'enveloppe qui contient l'adresse du destinataire va représenter le format et la lettre qui est à l'intérieur, le code de l'exécutable.
Le facteur va jouer le rôle du système d'exploitation.

L'objectif de cette bibliothèque est donc de pouvoir lire et modifier toutes ces méta-informations.
Pour ça, on va décomposer le format sous forme d'objets qui pourront être manipulés par l'utilisateur. Cette partie est implémentée dans une classe *parser*.
Une deuxième partie va consister à reconstruire un binaire valide à partir de ces objets. Elle est implémentée dans une classe *builder*.

Le nom donné à la bibliothèque est **LIEF** pour **L**ibrary **T**o **I**nstrument **E**xecutable **F**ormat.
C'est une bibliothèque et non un programme elle va donc servir de support pour le développement de nouveaux logiciels ou d'outils. Des exemples d'utilisation seront présentés à la fin du rapport.

Il existe trois principaux formats d'exécutables :
- **P**ortable **E**xecutable (**PE**) pour les systèmes Windows
- Mach-O pour les systèmes OS X
- **E**xecutable and **L**inkable **F**ormat (**ELF**) pour les systèmes Linux

Il y a des similitudes entre ces trois formats. Par exemple tous ont un point d'entrée qui est l'adresse de la première instruction assembleur à exécuter. Il est donc possible de factoriser certaines de ces parties.

Dans l'état actuel, il n'existe pas d'outils permettant d'analyser et de modifier les trois formats exécutables. Des outils existent pour analyser le format **PE** d'autre pour instrumenter le format **ELF**, mais ils sont bien souvent incomplets. Les objectifs de la bibliothèque sont donc d'être :

Portable

La bibliothèque doit être utilisable sur les trois principaux systèmes d'exploitations :
- Linux
- Windows
- OS X

Multi-formats

La bibliothèque doit pouvoir gérer les trois principaux formats :

— **ELF**
— **PE**
— **Mach-O**

Fournir une abstraction des formats

La bibliothèque doit pouvoir tirer partie des similarités qui existent entre les trois formats.

Fournir une API facile d'utilisation

Afin de développer des outils utilisant cette bibliothèque, l'API doit être simple et intuitive.

2. Formats executables

2.1 ELF

Le format **ELF** (**E**xecutable and **L**inkable **F**ormat) est le format utilisé pour les binaires et les bibliothèques sous Linux, Android, FreeBSD.
La figure ci-dessous représente le format de manière générale :

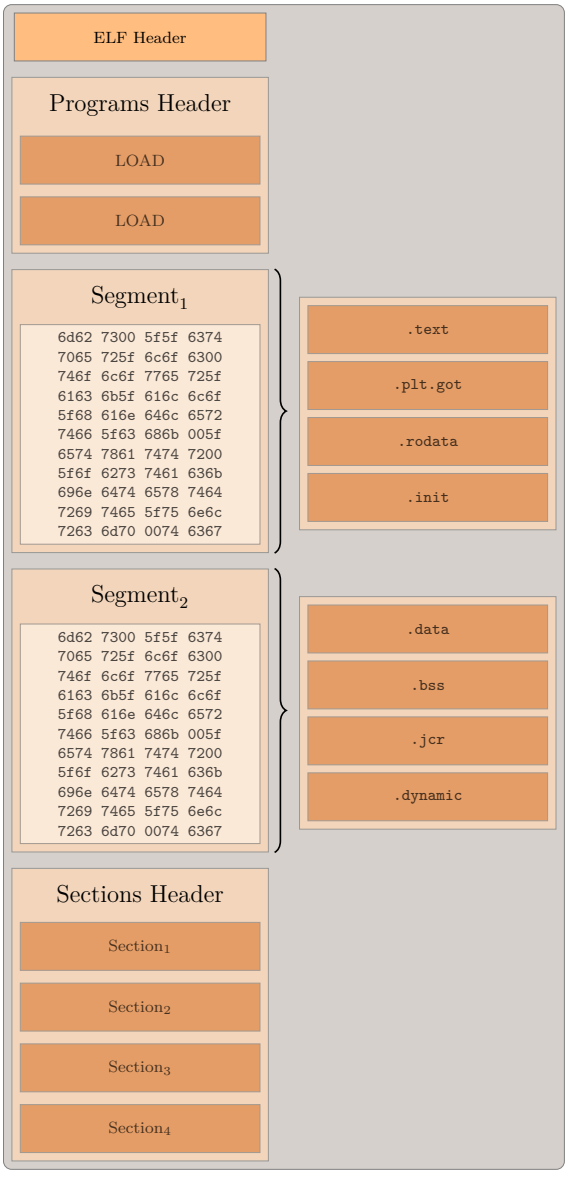


FIGURE 2.1 –

Au début du fichier il y a un en-tête **ELF** qui contient des informations comme :
- L'architecture sur laquelle le binaire va s'exécuter (`x86`, `ARM`, `PowerPC`...)
- Le type du binaire (exécutable, bibliothèque...)

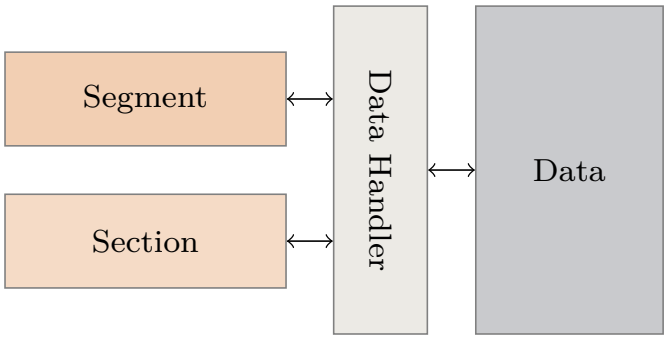
— Le point d'entrée du programme : l'adresse de la première instruction à exécuter.

Vient ensuite la table des segments (*Program header table*), cette table contient la liste des blocs de données à charger dans l'espace mémoire du processus. Généralement, à la fin du binaire se trouve la table des sections (*Section header table*).
Contrairement à la table des segments, la table des sections est utilisée pendant la liaison des fichiers objets (*linking*) pour avoir une plus grande précision sur la position des données. Nous y reviendrons par la suite.

2.1.1 Dualité section/segment

Dans le format **ELF** il y a deux types de structures qui peuvent contenir des données brutes (code assembleur, chaine de caractères...) : les sections et les segments. Généralement, le contenu des sections est également dans le segment associé (voir la figure 2.1). Le problème qui s'est posé était donc de savoir dans quelle structure enregistrer ces données.
On ne pouvait pas les enregistrer deux fois en effet si l'utilisateur modifie les données d'une section et que celle-ci fait partie d'un segment, alors il aurait également fallu mettre à jour les données du segment.

Pour résoudre ce problème nous avons crée une interface entre les données brutes du binaire et les sections/segments. L'interface se matérialise par une classe **`DataHandler`** qui contient les données brutes du binaire et les méthodes pour y accéder, en ajouter ou en supprimer.



Les classes **`Section`** et **`Segment`** de LIEF ont un attribut *dataHandler_* qui est un pointeur (plus exactement un **`std::shared_ptr`**) sur le **`DataHandler`** du binaire.
Le code du Listing 2.1 représente une partie de la définition de la classe **`Section`** et le Listing 2.2 une partie de son implémentation.

Listing 2.1 – **`Section.hpp`**

```
1 namespace ELF {
2   class Section : public LIEF::Section {
3    public:
4       Section(ELF::Structures::Elf64_Shdr* header);
5       Section(ELF::Structures::Elf32_Shdr* header);
```

```
6       ...
7       std::vector<uint8_t> content(void) const;
8       void content(const std::vector<uint8_t>& data);
9       ...
10    private:
11      ...
12      std::shared_ptr<ELF::DataHandler::Handler>  dataHandler_;
13      ...
14  };
15  }
```

Listing 2.2 – **Section.cpp**

```
1   #include "LIEF/ELF/Section.hpp"
2
3   using namespace ELF;
4   ...
5   std::vector<uint8_t> Section::content(void) const {
6     return this->dataHandler_->content(this->offset_, this->size_,␣
    ↪DataHandler::Node::SECTION);
7   }
8
9   void Section::content(const std::vector<uint8_t>& data) {
10    if (this->originalSize_ > 0 and data.size() > this->originalSize_) {
11      std::cout << "[WARNING] You insert data in the section "
12                << this->name() << " whose the size it bigger ("
13                << std::dec << data.size() << " > "
14                << this->originalSize_ << "). It may lead to overaly" << std::
    ↪endl;
15    }
16
17    if (this->type_ == ELF::Structures::SECTION_TYPES::SHT_NOBITS) {
18      std::cout << "[WARNING] You insert data in section "
19                << this->name() << " which has SHT_NOBITS type !" << std::
    ↪endl;
20    }
21    if (data.size() > 0) {
22      this->dataHandler_->content(this->offset_, data, DataHandler::Node::
    ↪SECTION);
23    }
24    this->size_ = data.size();
25  }
26  ...
```

Dans le Listing 2.2, la méthode **Section::content(void)** permet d'accéder aux données de la section en passant par le *DataHandler* et prend en paramètre :

- L'offset où se trouve les données dans le binaire : **this->offset_**
- Sa taille : **this->size_**
- Le type de la classe qui souhaite accéder aux données :

`DataHandler::Node::SECTION` ou `DataHandler::Node::SEGMENT`

La méthode `void Section::content(const std::vector<uint8_t>& data)` va permettre de modifier les données de la sections. Les lignes 10-20 effectuent des vérifications pour garder une structure valide de l'objet binaire. La ligne 22 va mettre a jour les données brutes en utilisant les mêmes paramètres que ci-dessus.

2.2 Utilisation de LIEF pour supprimer la table des sections

Comme nous l'avons expliqué précédemment, les sections sont utilisées pendant la phase de liaison par le *linker* et les segments par le Système d'exploitation lors de la phase d'exécution. Toute l'information utile pour l'exécution se trouve dans les segments. On peut donc supprimer la **table des sections** sans altérer le fonctionnement du programme. Nous allons voir comment le faire avec LIEF via l'API Python.

Pour cet exemple, nous allons supprimer la table des sections du binaire `ls` qui permet de lister les éléments (dossiers et fichiers) d'un répertoire.
Nous devons dans un premier temps ouvrir le binaire :

```
>>> from lief import ELF
>>> binary = ELF.parse("/usr/bin/ls")
```

Nous allons ensuite récupérer l'*header* du binaire car c'est dans cette structure que contient l'offset vers la table des sections ainsi que le nombre de sections. On va donc mettre ces valeurs à 0 et lors de la reconstruction du binaire, le *builder* ne construira pas la table.

```
>>> header = binary.header
>>> print header
Object file type:                EXECUTABLE
Machine type:                    x86_64
Object file veresion:            CURRENT
Entry Point:                     0x4049a0
Program header offset:           0x40
Section header offset:           1e718
Processor Flag                   0
Header size:                     40
Program header size:             38
Number of program header:        9
Size of section header:          40
Number of section headers:       1d
Section Name Table idx:          1c
>>> header.numberof_section_header = 0
>>> header.section_header_offset   = 0
>>> print header
Object file type:                EXECUTABLE
```

```
Machine type:                    x86_64
Object file veresion:            CURRENT
Entry Point:                     0x4049a0
Program header offset:           0x40
Section header offset:           0
Processor Flag                   0
Header size:                     40
Program header size:             38
Number of program header:        9
Size of section header:          40
Number of section headers:       0
Section Name Table idx:          1c
```

Nous devons ensuite reconstruire le binaire, nous pouvons alors utiliser la méthode **write**.

```
>>> binary.write("ls_updated")
```

Avec l'utilitaire **readelf** qui permet d'afficher les informations **ELF** d'un binaire, nous pouvons donc vérifier que la table des sections n'est plus présente :

Listing 2.3 – Liste des sections du binaire **ls**

```
$ readelf -S /usr/bin/ls

There are 29 section headers, starting at offset 0x1e718:
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
       0000000000000104  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000004003a0  000003a0
       0000000000000cc0  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000401060  00001060
       00000000000005da  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           000000000040163a  0000163a
       0000000000000110  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000401750  00001750
       0000000000000070  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             00000000004017c0  000017c0
       00000000000000a8  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000401868  00001868
```

```
      0000000000000a68  0000000000000018  AI       5    24     8
  [11] .init             PROGBITS         00000000004022d0  000022d0
```

Listing 2.4 – Liste des sections sur le binaire **ls_updated**

```
$ readelf -S ls_updated

There are no sections in this file.
```

Et le binaire **ls_updated** peut toujours s'exécuter :

```
$ ./ls_updated
bindings  build  build-debug  CMakeLists.txt  CMakeModules  doc  examples ␣
 ↪include  package  README.md        src  tests  third_party  tools
```

On peut noter qu'en supprimant la table des sections d'un binaire, **gdb** est incapable de debugger le programme :

```
$ gdb -q ls_updated
"/home/romain/dev/LIEF/ls_updated": not in executable format: File format␣
 ↪not recognized
(gdb)
```

Car **gdb** utilise les sections pour localiser les informations de debug.

2.2.1 Les Symboles

Une des autres parties du format **ELF**, également présente dans les autres formats, est la table des symboles. Elle permet de localiser et de repositionner les définitions et les références symboliques d'un programme. Pour l'illustrer prenons le code **C** suivant :

Listing 2.5 – exemple_symboles.c

```c
#include <stdio.h>
#include <stdlib.h>
void print_hello(void) {
  puts("Hello\n");
}
int main(int argc, char** argv) {
  print_hello();
  return EXIT_SUCCESS;
}
```

Dans ce code, on définit une fonction **print_hello** qui utilise une fonction **puts** de la *libc* (**libc.so**). On a également la fonction **main** qui est le *point d'entrée* dans le programme. Avec l'utilitaire **readelf** [1], on peut lister les symboles du binaire :

1. Toutes les commandes utilisant **readelf** peuvent être faites en utilisant *LIEF* (voir **examples/python/elf_reader.py**)

```
1  $ readelf -s exemple_symboles
2
3  Symbol table '.dynsym' contains 4 entries:
4     Num:  Value Size Type    Bind   Vis      Ndx Name
5       1: 000000 0    FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
6       ...
7
8  Symbol table '.symtab' contains 69 entries:
9     Num:  Value Size Type  Bind   Vis      Ndx Name
10     ...
11     38: 000000  0 FILE    LOCAL  DEFAULT  ABS exemple_symboles.c
12     51: 000000  0 FUNC    GLOBAL DEFAULT  UND puts@@GLIBC_2.2.5
13     ...
14     62: 4004e6 17 FUNC    GLOBAL DEFAULT   14 print_hello
15     ...
16     64: 4004f7 27 FUNC    GLOBAL DEFAULT   14 main
```

Ces symboles peuvent être stockés dans deux types de section :

- **`.dynsym`** : Section associée aux symboles dynamiques, elle ne peut pas être supprimée.
- **`.symtab`** : Section associée aux symboles statiques, elle peut être supprimée.

Prenons le cas simple du symbole **`print_hello`** (ligne 14). Ce symbole est défini comme étant une fonction (**`FUNC`**) à l'adresse **`0x4004e6`** sa taille est de 17 octets, la fonction se termine donc à l'adresse **`0x4004f7`**. On peut le vérifier avec l'outil **`objdump`** qui permet de désassembler le code d'un exécutable.

```
$ objdump -d --start-address=0x4004e6 --stop-address=0x4004f7 -M intel␣
 ↪exemple_symboles

00000000004004e6 <print_hello>:
4004e6:       55                      push   rbp
4004e7:       48 89 e5                mov    rbp,rsp
4004ea:       bf a4 05 40 00          mov    edi,0x4005a4
4004ef:       e8 cc fe ff ff          call   4003c0 <puts@plt>
4004f4:       90                      nop
4004f5:       5d                      pop    rbp
4004f6:       c3                      ret
```

On remarquera la référence à **`puts`** dans l'instruction **`call`** à l'adresse **`0x4004ef`**. Le symbole **`main`** se comporte comme **`print_hello`**. Le symbole **`exemple_symboles.c`** (ligne 11) définit notre fichier.

Le symbole **`puts@@GLIBC_2.2.5`** est diffèrent des autres. Il est défini dans la section **`.dynsym`** et **`.symtab`** il est donc **nécessaire** à l'exécution du programme. La colonne *Value*[2] de ce symbole est à 0 ce qui est normal puisque cette fonction n'est pas définie dans notre code mais dans **`/usr/lib/libc.so`** à l'adresse **`0x676b0`** comme le montre le

2. Généralement cette valeur contient l'adresse du symbole dans l'espace mémoire du processus.

Listing 2.6. Le *loader* Linux (**ld-linux.so**) va donc résoudre cette adresse au moment de l'exécution en utilisant les *relocations* et le système de PLT/GOT.

Comme les symboles statiques de la section **.symtab** ne sont pas requis pour exécuter correctement le programme, il est possible de les supprimer en utilisant l'utilitaire **strip** ou bien LIEF comme nous allons le voir.
Supprimer ces symboles permet de réduire la taille du binaire. En effet, sur de gros binaires cette taille peut être relativement grande.

Listing 2.6 – Symbole **puts** de **libc.so**

```
$ readelf -s /usr/lib/libc.so.6 | grep -E "\<puts$"
7152: 00000000000676b0   456 FUNC    WEAK   DEFAULT   13 puts
```

Modification des symboles avec LIEF

Nous allons à présent voir comment modifier ou lire les symboles d'un binaire en utilisant LIEF.
Toujours avec le binaire **exemple_symboles** nous pouvons lire les symboles comme ceci :

```
>>> from lief import ELF
>>> binary = ELF.parse("exemple_symbole")
>>> static_symbols  = binary.static_symbols
>>> dynamic_symbols = binary.dynamic_symbols
>>> for symbol in static_symbols:
...     print symbol

ex_symbole.c                  4         LOCAL     0         0
...
puts@@GLIBC_2.2.5             2         GLOBAL    0         0
...
print_hello                   2         GLOBAL    4004e6    11
...
main                          2         GLOBAL    4004f7    1b
>>> for symbol in dynamic_symbols:
...     print symbol

puts                          2         GLOBAL    0         0
...
```

Nous pouvons également changer le nom des symboles :

```
>>> print_hello_sym = [s for s in static_symbols if s.name == "print_hello
 ↪"][0]
>>> print_hello_sym.name = "print_toto"
>>> binary.write("exemple_symboles_1")
```

Pour vérifier que le programme s'exécute correctement et que notre changement a bien

été pris en compte :

```
$ exemple_symboles_1
Hello
$ readelf -s ./exemple_symboles_1|grep print_toto
62: 00000000004004e6    17 FUNC    GLOBAL DEFAULT   14 print_toto
```

Un des exemples d'utilisation réel de la modification des symboles est la protection des noms des fonctions d'une bibliothèque. En effet une bibliothèque dynamique (et statique) va exporter ses symboles pour pouvoir être liée à l'exécutable qui l'utilise. Ces exécutables vont eux-mêmes avoir le nom des symboles de la bibliothèque qu'ils importent. A titre d'exemple, nous allons compiler une bibliothèque **libadd.so** qui fournit la fonction **int add(int x, int y)** et qui sera utilisée par le binaire **binadd.bin**.

Le code de **libadd.so** est donné au Listing 2.7 et le code de **binadd.bin** au listing Listing 2.8

Listing 2.7 – **libadd.c**

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

int add(int a, int b) {
  printf("%d + %d = %d\n", a, b, a+b);
  return a + b;
}
```

Listing 2.8 – **binadd.c**

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

int main(int argc, char argv) {
  if (argc != 3) {
    printf("Usage: %s <a> <b>\n", argv[0]);
    exit(-1);
  }
  int res = add(atoi(argv[1]), atoi(argv[2]));
  printf("From myLIb, a + b = %d\n", res);
  return 0;
}
```

Le Makefile utilisé pour compiler ces sources est donné dans le listing suivant :

```
CC=gcc
CXX=g++

all: binadd.bin
```

```
libadd.so: libadd.c
	$(CC) -Wl,--hash-style=sysv -fPIC -shared -o $@ $^

binadd.bin: binadd.c libadd.so
	$(CC) $^ -Wl,--hash-style=sysv -L. -ladd -o $@
	chmod a+rx $@

run: libadd.so binadd.bin
	LD_LIBRARY_PATH=. ./binadd.bin 1 2
```

En listant les symboles du binaire et de la bibliothèque on peut voir la présence du symbole **add** dans la partie dynamique de la bibliothèque et de l'exécutable :

```
$ readelf -s binadd.bin | grep add
2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND add

$ readelf -s libadd.so | grep add
3: 0000000000000670    58 FUNC    GLOBAL DEFAULT   12 add
```

L'idée est donc de modifier le nom de ce symbole dans le binaire **et** la bibliothèque. Le script suivant montre comment faire avec LIEF :

```
import lief

libadd = lief.ELF.parse("libadd.so")
binadd = lief.ELF.parse("binadd.bin")

libadd_dynsym = libadd.dynamic_symbols
binadd_dynsym = binadd.dynamic_symbols

# Change add in the libary
for sym in libadd_dynsym:
    if sym.name == "add":
        sym.name = "zzz"

# Change "add" in the binary
for sym in binadd_dynsym:
    if sym.name == "add":
        sym.name = "zzz"

libadd.write("libadd.so");
binadd.write("binadd_obf.bin")
```

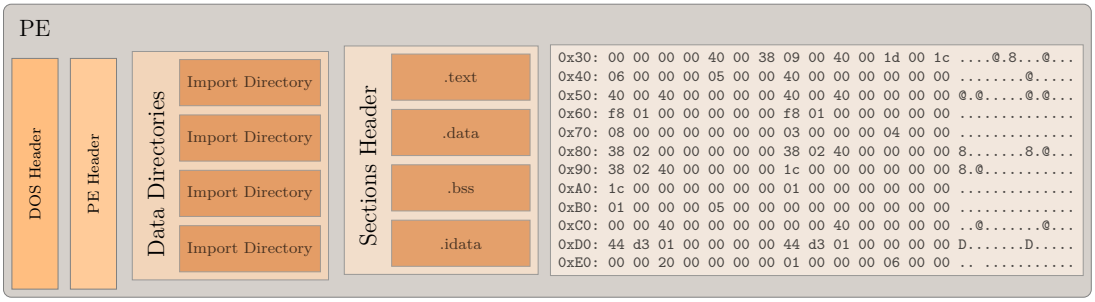
Le symbole de la fonction a été changé et le binaire **binadd_obf.bin** s'exécute toujours. La modification du noms des fonctions d'une bibliothèque peut être utile pour protéger certaines des fonctions sensibles de cette bibliothèque. Comme nous l'avons vu précédemment on retrouve le nom de la fonction dans le code désassemblé.

Si cette fonction se nomme **AES_cbc_encrypt()** il est plus facile de *deviner* ce que fait le code assembleur associé que si elle se nommait **aaabbbcccdddeee()**

2.3 PE

Le format **PE** pour **P**ortable **E**xecuatble est le format utilisé par les systèmes Windows. En réalité c'est une version modifiée du format COFF qui a été introduit dans `Unix System V` mais comme nous l'avons vu c'est maintenant le format **ELF** qui est utilisé par Linux. Windows avec son format **PE** est le seul a avoir gardé une implémentation du format COFF.

La figure suivante représente la structure (simplifiée) du format **PE**



Au début du fichier se trouve deux en-têtes :

- `DOS Header` : Présent pour reconnaitre un exécutable valide pouvant s'exécuter sous MS-DOS.
- `PE header` : En-tête plus complet contenant par exemple le point d'entrée du programme. Cet en-tête se rapproche plus de l'en-tête **ELF** que l'en-tête DOS-Header.

Vient ensuite les *data directories*. C'est un tableau de métadonnées sur l'exécutable. Ce tableau contient des *pointeurs* vers les sections qui contiennent l'information. Par exemple nous avons l'entrée *Table des imports* qui contient un pointeur (*offset*) vers les informations sur les bibliothèques importées (`kernel32.dll`, `msvcrt.dll` ...) une autre sur la signature de l'exécutable. Un exécutable **PE** peut être signé pour garantir l'intégrité et l'authenticité de celui-ci. Ces metadonnées sont au nombre de dix en voici les plus importantes :

- Table des exports : utilisée par les bibliothèques pour exporter des fonctions.
- Table des imports
- Table des ressources : contient les images, les icônes...
- Table des exceptions : utilisée pour les exceptions `C++`
- Table de signature
- Table des relocations : informations pour la relocation de l'exécutable.
- Table de debug : informations pour le debuggage.

Après les *data directories* se trouve la table des en-têtes de sections. Comme dans le format **ELF**, les sections contiennent les données de l'exécutable mais contrairement au format **ELF** celles-ci sont utilisées à la fois pendant la phase de liaison et pendant la phase d'exécution. Il n'y a donc pas de segment ce qui permet d'ajouter ou de supprimer des sections beaucoup plus facilement.

Comme pour le format **ELF**, la partie **PE** de LIEF intègre un *parser* qui décompose l'exécutable en un objet manipulable et un *builder* qui reconstruit un exécutable à partir de sa représentation objet.

Je vais ensuite détailler une des partie non trivialedans la reconstruction du binaire : la table des imports.

La table des imports indique que'elles sont les bibliothèques et les fonctions externes utilisées par l'exécutable. Par exemple :

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

int main(int argc, char** argv) {
  printf("Hello World\n");
  Sleep(3);
  return 0;
}
```

La fonction **printf** n'est pas implémentée dans le binaire, elle est localisée dans la bibliothèque **msvcrt.dll** et la fonction **Sleep** est implémentée dans **Kernel32.dll**. On va donc avoir une entrée dans la table des imports du type :

```
msvcrt.dll: printf
  ...
kernel32.dll: Sleep, GetTickCount
```

La table des import est généralement située dans la section **.idata** dont la structure (simplifiée) est donnée à la figure 2.2

La première structure donne les informations sur la bibliothèque à importer :

- Son nom
- L'*offset* de la *lookup table*
- L'*offset* de l'*address table*
- Le *timestamp* de la bibliothèque.

La *lookup table* est un tableau d'*offset* des noms des fonctions qui doivent être importées. Par exemple si nous voulons importer les fonctions **Sleep** et **GetTickCount** on aura une *lookup table* de la forme :

- **lookup_table[0] = offset_0**
- **lookup_table[1] = offset_1**

offset_0 et **offset_1** représentent les *offsets* des noms de ces fonctions dans la *Hint-Name Table* (voir figure 2.2) l'*address table* est **identique** à la *lookup table* mais lors de l'exécution du programme, le chargeur Windows va remplacer l'*offset* du nom par l'adresse de la fonction dans la bibliothèque. Le code assembleur associé à un appel de fonction est généralement (en **x86**) :

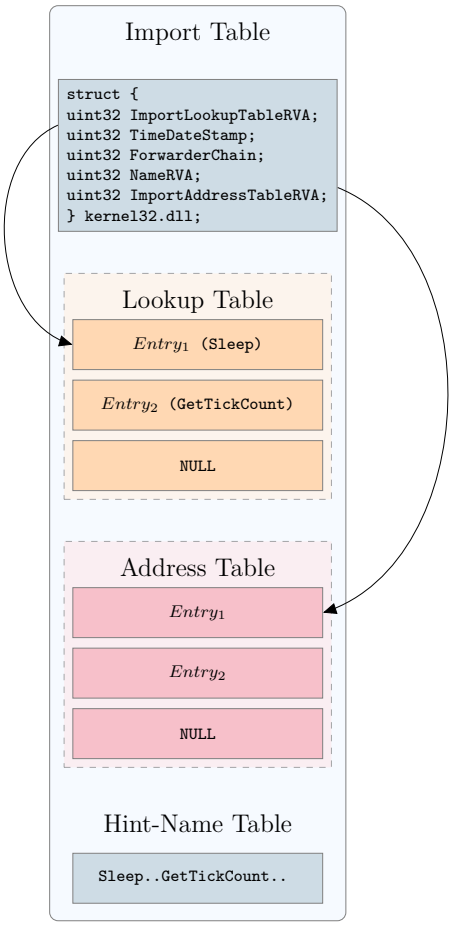


FIGURE 2.2 – Structure de la table des imports

```
Sleep(3)
```

```
push 3
call address_table[0] // Call vers l'adresse de sleep dans l'adresse table
```

Maintenant lorsque qu'on veut reconstruire la table des imports à un autre endroit dans le binaire, le **`call address_table[1]`** ne sera plus valide puisque la table des imports sera située ailleurs et donc l'*address table* aussi. Pour pallier à ce problème la solution trouvée est d'utiliser un système de *trampoline*.

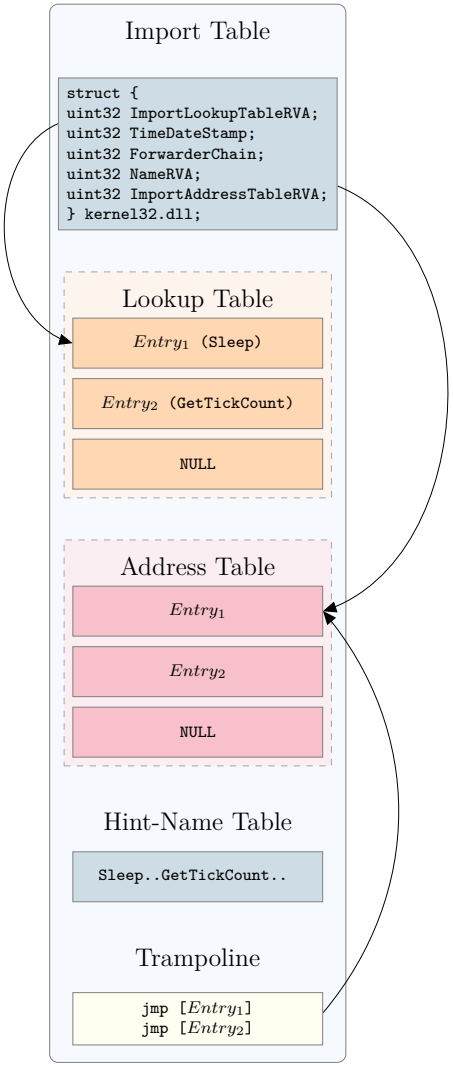
Au lieu d'avoir l'adresse de la fonction **`Sleep`** dans **`address_table[0]`** on va mettre un *jump* vers la nouvelle entrée dans l'*address table*. **`address_table[0] = jmp new_address_table[0]`**. De cette manière on va avoir cette suite d'instructions lors de l'appel à la fonction **`Sleep(3)`** :

```
push 3
call address_table[0]

address_table[0]:
  jmp [new_address_table[0]]
```

```
new_address_table[0]: 0xFFFFC0FFEE
```

La figure qui suit résume la transformation :

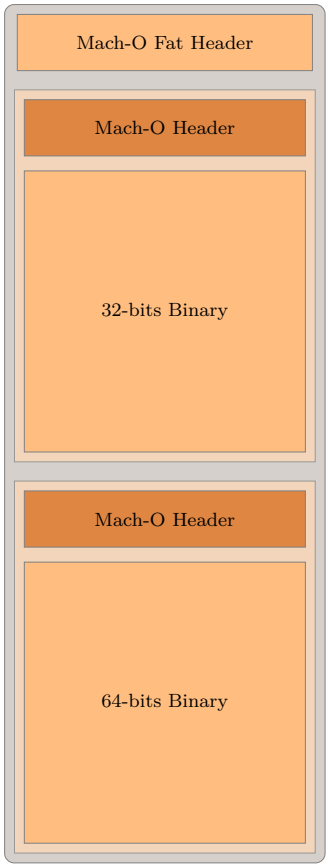


Le fait de pouvoir déplacer la table des imports ailleurs dans l'exécutable va permettre d'ajouter, de supprimer ou de modifier les bibliothèques et/ou les fonctions qui y sont importées.

2.4 MachO

Le format Mach-O est le dernier que j'ai implémenté dans LIEF.
La particularité de ce format est qu'il peut contenir plusieurs architectures dans un même binaire (Fat-binary). Par exemple, l'exécutable **`ls`** peut embarquer la version 32-bits et 64-bits dans le même binaire. Sur un système 32-bits, c'est la partie 32-bits

qui s'exécute et sur un système 64-bits, l'autre partie.
La figure suivante montre la structure d'un tel binaire :



Pour traiter ce genre de binaires avec LIEF, on va parser séparément le premier binaire puis le second pour les transformer en objet **`MachO::Binary`** puis on va retourner ces objets sous forme de liste (**`std::list`**).

En Python on obtient :

```
from lief import MachO

binaries = MachO.parse("/bin/ls")
binary_32bits = binaries[0]
binary_64bits = binaries[1]
```

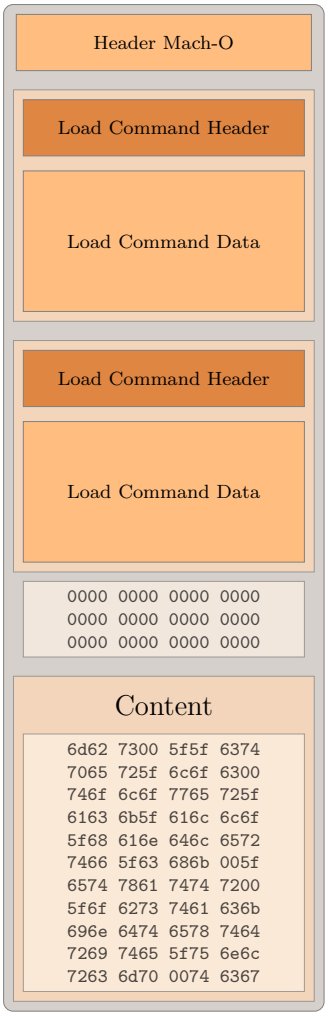
Nous allons à présent regarder plus en détail la structure d'un exécutable Mach-O.

Comme pour les deux précédents formats on trouve au début de l'exécutable un en-tête qui contient des informations telles que l'architecture cible de l'exécutable ou des drapeaux sur la façon d'exécuter le binaire. Après cet en-tête vient une suite de *load commands*. Celles-ci sont la structure de base du format Mach-O et permettent au chargeur OSX d'avoir plus d'informations sur le binaire. Nous avons par exemple les *load command* de type :

— **`LC_MAIN`** qui donne le point d'entrée du programme.

— **`LC_SEGMENT`** qui est l'équivalent du segment pour le format **ELF** et de la section pour le format **PE**.
— **`LC_SYMTAB`** qui contient les symboles de l'exécutable.

La figure suivante montre cet agencement.



Après la liste de *load commands* se trouve généralement une suite de zéros (*padding*) qui sont, comme nous le verront, très utiles. Après ce *padding* retrouve des données brutes : les chaines de caractères, les noms des symboles, le code assembleur du binaire etc.

Une des modifications essentielles du format est l'ajout de *load command*. La méthode naïve serait de rajouter la commande au début de l'exécutable, main en faisant cela, tout les *offset* de l'exécutable seraient corrompus. On va donc insérer la commande dans la zone de *padding* du format et ce qui évite de modifier les *offset* des autres structures.

3. Solutions techniques

3.1 Langage de développement

LIEF est développé en **C++11** et on intègre également une API Python. En utilisant cette combinaison **C++**/Python on peut intégrer LIEF dans une grande partie de projets projets tout en gardant la simplicité et la rapidité de développement en Python.

3.2 Système de construction

Comme nous voulons que la bibliothèque soit portable sur les différentes OS sans avoir à gérer leur chaine de compilation, le choix s'est porté sur CMake.
CMake va automatiquement générer les fichiers nécessaires à la compilation du projet. Par défaut, un Makefile sera généré pour Linux, un projet Visual Studio sous Windows et un projet XCode sous OSX.
En utilisant CMake, on facilite également l'intégration de LIEF dans d'autre projets. Par exemple si le projet **Foo** souhaite utiliser LIEF, il lui suffit de rajouter la ligne :

```
add_subdirectory(LIEF_PATH)
```

et le projet sera automatique compilé avec **LIEF**

CMake repose sur un fichier **CMakeLists.txt** qui décrit les dépendances du projet, les drapeaux de compilation nécessaires, le chemin d'installation...

Voici un aperçu du **CMakeLists.txt** de **LIEF** :

Listing 3.1 – **CMakeLists.txt** du projet LIEF

```
1  file(
2    GLOB_RECURSE
3    LIBLIEF_SOURCE_FILES
4    ${CMAKE_CURRENT_SOURCE_DIR}/src/*
5  )
6
7  add_library(LIB_LIEF_STATIC STATIC ${LIBLIEF_SOURCE_FILES})
8
9  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
10
11 add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/examples/cpp)
```

La commande **file** va définir les sources du projet. Dans le cas de LIEF tous les fichiers du répertoire **src**.

La ligne 7 indique qu'on veut compiler une bibliothèque en statique. La ligne 9 vas ajouter le drapeau **-std=c++11** au compilateur **C++** (i.e. **g++**, **clang++**...) pour lui indiquer que le code est en **C++11**.

La ligne 11 va indiquer que le projet contient également un sous dossier qu'il faut prendre en compte dans la compilation. Ce sous dossier contient également un fichier **CMakeLists.txt** (**examples/cpp/CMakeLists.txt**) qui decrit comment compiler les exemples.

Pour compiler et installer le projet sous Linux :

```
$ cmake        # Génère le Makefile
$ make         # Compile
$ make install # Installe
```

Sous OSX :

```
$ cmake        # Génère un projet XCode
```

Pour Windows nous devons indiquer que nous voulons utiliser le compilateur LLVM car le compilateur de Microsoft de supporte pas complètement le **C++11**.

```
$ cmake -TLLVM-vs2014 # Génère un projet Visual Studio avec LLVM
```

3.3 API Python

La bibliothèque est évidement utilisable avec d'autre projets **C++**, comme un des objectifs est de pouvoir développer des petits programmes facilement et rapidement, nous avons donc décidé d'ajouter une API Python. Pour se faire plusieurs choix étaient possibles :

- Boost Python [1]
- C Python [2]
- PyBind11 [3]

L'avantage de Boost Python est d'être relativement simple d'utilisation mais son installation sur les différents système d'exploitation n'est pas toujours évidente. L'API C de Python travail au niveau **C** et le code produit n'est pas toujours facilement lisible et demande une bonne gestion (manuelle) des références sur les objets Python.
PyBind11 est développé en **C++11** et se présente sous forme de fichiers **.h** on a donc pas de dépendance supplémentaire et son utilisation est très intuitive comme nous allons le voir.

A titre d'exemple, nous allons prendre la classe **ELF::Header** dont la définition est dans **include/LIEF/ELF/Header.hpp**. Une version (très) simplifiée est donnée dans le listing qui suit.

1. http://www.boost.org/doc/libs/1_61_0/libs/python/doc/html/index.html
2. https://docs.python.org/2/extending/extending.html
3. https://pybind11.readthedocs.io/en/latest/

Listing 3.2 – **ELF/Header.hpp**

```
class Header {
    public:
      Header(void);

      ELF::Structures::E_TYPE  file_type(void) const;
      ELF::Structures::ARCH    machine_type(void) const;

      void file_type(EdddLF::Structures::E_TYPE type);
      void machine_type(ELF::Structures::ARCH machineType);

    private:
      ELF::Structures::E_TYPE  fileType_;
      ELF::Structures::ARCH    machineType_;
  };
```

Pour transformer cette classe en Python nous devons d'abord déclarer le module dans lequel elle va se trouver :

Listing 3.3 – **PythonAPI.cpp**

```
...

py::module LIEF_module("lief");
py::module LIEF_ELF_module = LIEF_module.def_submodule("ELF");

py::class_<ELF::Header>(LIEF_ELF_module, "Header")

  ...

return LIEF_module.ptr();
```

Avec ce code, la classe **Header** sera accessible en Python avec le code suivant :

```
from lief.ELF import Header
...
```

Nous devons implémenter à proprement parler la classe **Header**. PyBind11 fournit, entre autre, la méthode **def_property(<name>, <getter>, <setter>, <doc>)** qui permet de modéliser un attribut de la classe python tout en utilisant des *getters* et des *setters*. Cela donne :

Listing 3.4 – **PythonAPI.cpp**

```
...

py::class_<ELF::Header>(LIEF_ELF_module, "Header")
  .def_property("file_type",
      static_cast<getter_t<ELF::Structures::E_TYPE>>(&ELF::Header::file_
 ↪type),
```

```
      static_cast<setter_t<ELF::Structures::E_TYPE>>(&ELF::Header::file_
↪type),
      "Return binary's type It determine if the binary \
      is a executable, a library..."
      )

  .def_property("machine_type",
      static_cast<getter_t<ELF::Structures::ARCH>>(&ELF::Header::machine_
↪type),
      static_cast<setter_t<ELF::Structures::ARCH>>(&ELF::Header::machine_
↪type),
      "Return target architecture");

return LIEF_module.ptr();
```

L'ensemble des sources de l'API Python est dans le répertoire **bindings/python**. L'API se présente sous la forme d'une bibliothèque dynamique (**lief.so** pour Unix et **lief.pyd** pour Windows). Lors de l'installation elle sera placée dans le répertoire python contenant les modules (ex : **/usr/lib/python2.7/{dist,site}-packages** pour Linux).

Depuis l'interpréteur, une fois l'API installée, il est alors possible de faire :

```
>>> from lief.ELF import Header
>>> header = Header()
>>> header.file_type = 1
>>> print header.file_type
1
```

3.4 Tests

Une des parties de ce projet a consisté à mettre en place les tests de LIEF. Pour tester la partie **C++**, je me suis orienté vers Google Test [4] qui est un framework de tests unitaires. Il y a principalement deux choses à tester sur la bibliothèque :

— Le *parser* : Modélisation du binaire sous forme d'objets.
— La modification : Modification du binaire et reconstruction d'un binaire valide.

Le développement de petits outils a également permis de tester l'API.

4. https://github.com/google/googletest/

3.5 Documentation

Une documentation développeur et utilisateur a également été mise en place en utilisant les outils Sphinx [5] et Doxygen [6].
Cette documentation explique l'API mais également les concepts utilisés.

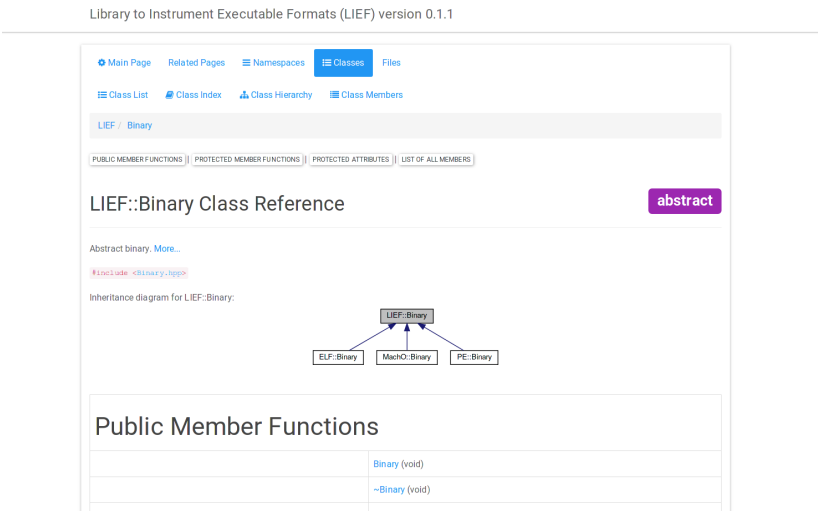


FIGURE 3.1 – Documentation Doxygen

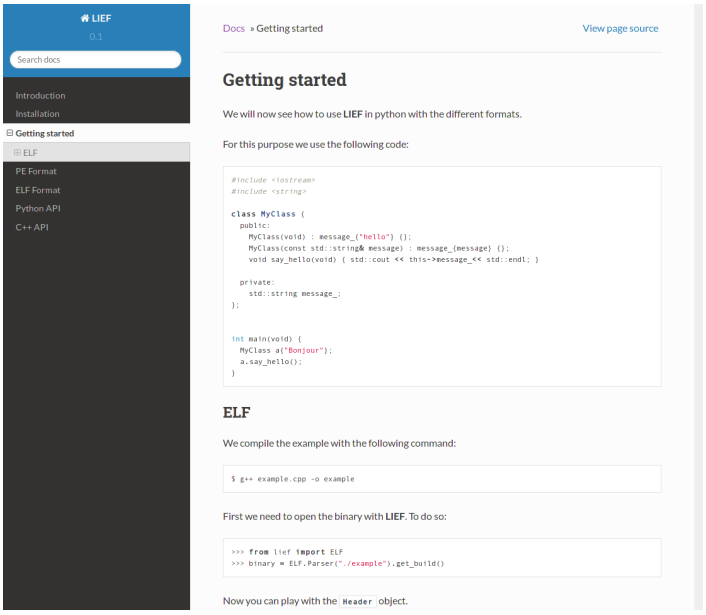


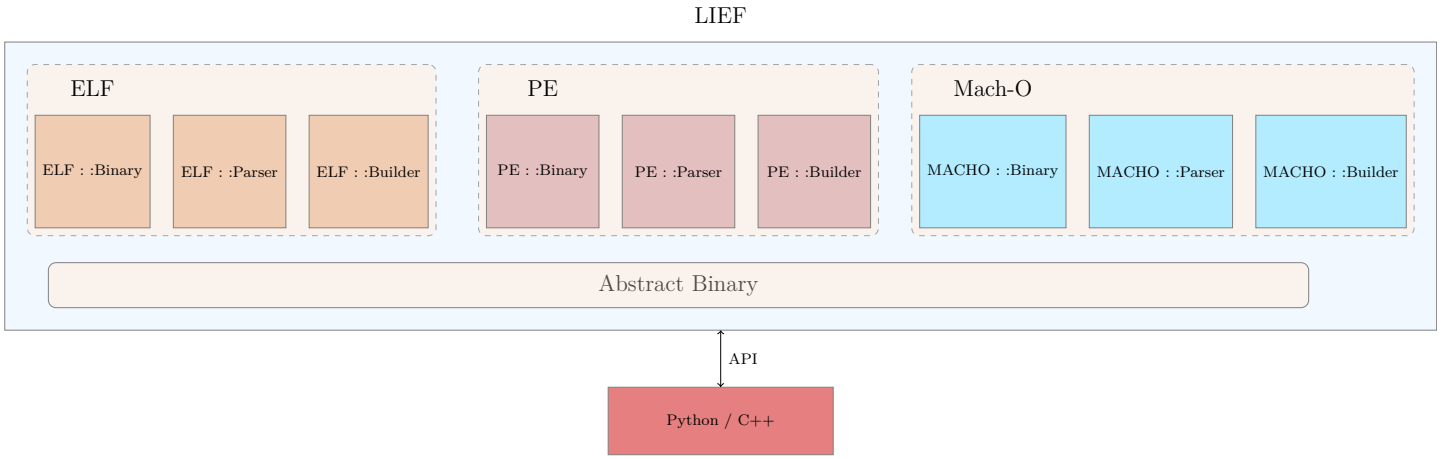
FIGURE 3.2 – Documentation Sphinx

5. http://www.sphinx-doc.org
6. http://www.doxygen.org

4. Architecture

La figure ci-dessous montre l'architecture globale de LIEF. Chaque formats a son propre *namespace* et contient :

— Une classe **`Binary`** pour modéliser le binaire.
— Une classe **`Parser`** pour transformer un binaire en objet **`Binary`**.
— Une classe **`Builder`** pour transformer un objet **`Binary`** en binaire.

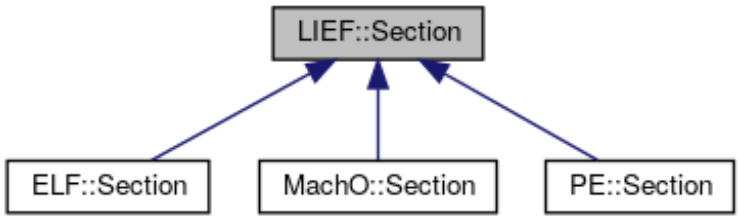


La classe **`Binary`** des différents formats hérite de la classe **`LIEF::Binary`** pour factoriser tous les éléments communs aux trois formats. Ces éléments communs sont :

— Le point d'entrée
— Les sections
— Les symboles

Les classes **`Section`** des trois format vont donc hériter de la classe **`LIEF::Section`** et aurons les attributs suivant :

— **`name_`** : le nom de la section.
— **`virtualAddress_`** : l'adresse virtuelle où la section sera mappée.
— **`offset_`** : l'*offset* des données de la section dans le binaire.



Dans la classe symbole, nous avons actuellement factorisé uniquement le nom associé au symbole.

Le fait d'avoir factoriser ces éléments permet de développer des outils qui seront indépendants du format. Nous avons par exemple développé un outil qui permet de calculer l'entropie d'un binaire.

4.1 Calcul de l'entropie d'un binaire

L'entropie est une notion mathématique pour évaluer le degré de *désodre* d'un système. Par exemple si la température d'une pièce est élevée, son entropie le sera aussi car les électrons auront une agitation thermique importante. Autre exemple : la suite de lettres `aaaaaaaaaaaaaaaaa` a un entropie beaucoup plus faible que `eRTMNXkNFxOmYIsNO@LM3y6YUnWkVmZykc3vHGa3s`.

L'entropie au sens de Shannon se calcul selon la formule :

$$-\sum_i P_i \log_2(P_i)$$

Avec P_i la probabilité d'apparition du symbole i.

La détermination de l'entropie d'un binaire permet de localiser les endroits où il y a potentiellement du code chiffré ou compressé. Le chiffrement du code d'un binaire est une technique particulièrement utilisée par les *Malwares* ou les virus pour compliquer le travail de l'analyste.

Le logiciel UPX qui est gratuit et open source permet de compresser le code assembleur d'un exécutable. La Figure 4.1 est une partie du code assembleur de l'exécutable `ls` et la Figure 4.2 le même code mais compressé avec UPX.

```
.text:0000000000404BD0                mov     rax, [rdi]
.text:0000000000404BD3                mov     r9, [rsi]
.text:0000000000404BD6                mov     r11, rcx
.text:0000000000404BD9                push    r15
.text:0000000000404BDB                mov     r10d, 1
.text:0000000000404BE1                push    r14
.text:0000000000404BE3                push    r13
.text:0000000000404BE5                push    r12
.text:0000000000404BE7                lea     r8, [rax+1]
.text:0000000000404BEB                push    rbp
.text:0000000000404BEC                mov     eax, 1
.text:0000000000404BF1                push    rbx
.text:0000000000404BF2                mov     ebp, edx
.text:0000000000404BF4                mov     rbx, 7E0000000000000h
.text:0000000000404BFE
.text:0000000000404BFE loc_404BFE:                            ; CODE XREF: sub_404BD0+B8↓j
.text:0000000000404BFE                movzx   ecx, byte ptr [r9]
.text:0000000000404C02                lea     r13, [r8-1]
.text:0000000000404C06                lea     r12, [r10-1]
.text:0000000000404C0A                cmp     cl, 3Dh
.text:0000000000404C0D                jz      loc_404E40
.text:0000000000404C13                jle     short loc_404C58
.text:0000000000404C15                cmp     cl, 5Ch
.text:0000000000404C18                jz      short loc_404C90
.text:0000000000404C1A                cmp     cl, 5Eh
.text:0000000000404C1D                jnz     short loc_404C70
.text:0000000000404C1F                movzx   ecx, byte ptr [r9+1]
.text:0000000000404C24                lea     r14, [r9+1]
.text:0000000000404C28                lea     r15d, [rcx-40h]
.text:0000000000404C2C                cmp     r15b, 3Eh
.text:0000000000404C30                jbe     loc_404CC8
.text:0000000000404C36                cmp     cl, 3Fh
.text:0000000000404C39                jz      loc_404CD8
```

FIGURE 4.1 – Code assembleur de l'exécutable `ls`

Le calcul de l'entropie se fait sur l'objet qui contient les données de l'exécutable. C'est à dire dans la classe `LIEF::Section` :

FIGURE 4.2 – Exécutable **ls** compressé avec UPX

Listing 4.1 – **src/Abstract/Section.hpp**

```
namespace LIEF {
  class Section {
    public:
      Section(void);
      ~Section(void);
      Section& operator=(const Section& copy);
      Section(const Section& copy);
      //! @brief section's name
      const std::string& name(void) const;
      //! @brief section's content
      virtual std::vector<uint8_t> content(void) const;
       ...
      //! @brief Set section content
      virtual void content(const std::vector<uint8_t>& data);
      //! @brief Section's entropy
      double entropy(void) const;
    protected:
      std::string   name_;
      uint64_t      virtualAddress_;
      uint32_t      size_;
      uint64_t      offset_;
  };
}
```

Son implémentation est relativement simple :

Listing 4.2 – Implémentation de la fonction **entropy()**

```
double Section::entropy(void) const {
  std::array<uint64_t, 256> frequencies = {0};
  const auto& content = this->content();
  for (const auto& x : content) {
    frequencies[x]++;
  }
  double entropy = 0.0;
  for (const auto& p : frequencies) {
    if (p > 0) {
      double freq = static_cast<double>(p) / static_cast<double>(content.
 ↪size());
      entropy += freq * std::log2(freq) ;
    }
  }
  return (-entropy);
}
```

A l'aide de **pyqtgraph** on peut visualiser cette entropie. La Figure 4.3 représente l'entropie des sections d'un exécutable.
L'histogramme orange représente l'entropie de la section **.text** qui contient le code assembleur.
Cette entropie est proche de 6 bits. L'histogramme rouge représente l'entropie de la section **.rsrc** qui contient des images et des icônes. On observe des fluctuations importantes avec une entropie relativement basse en moyenne. Car dans une image il y a souvent beaucoup de pixels noirs et de pixels blancs.

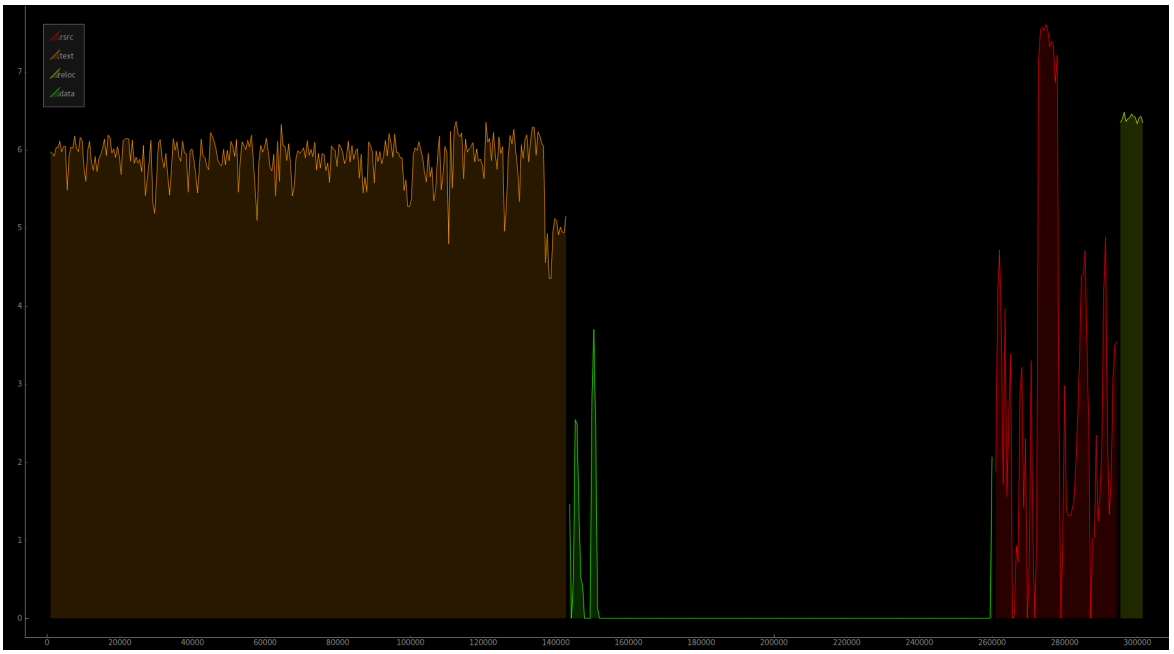


FIGURE 4.3 – Entropie des sections d'un exécutable.

La Figure 4.4 représente l'entropie d'un exécutable compressé avec UPX. L'histogramme jaune représente l'entropie de la section **`.text`**. Contrairement à la figure précédente, l'entropie est proche de 8 bits ce qui est caractéristique d'un binaire chiffré ou compressé.

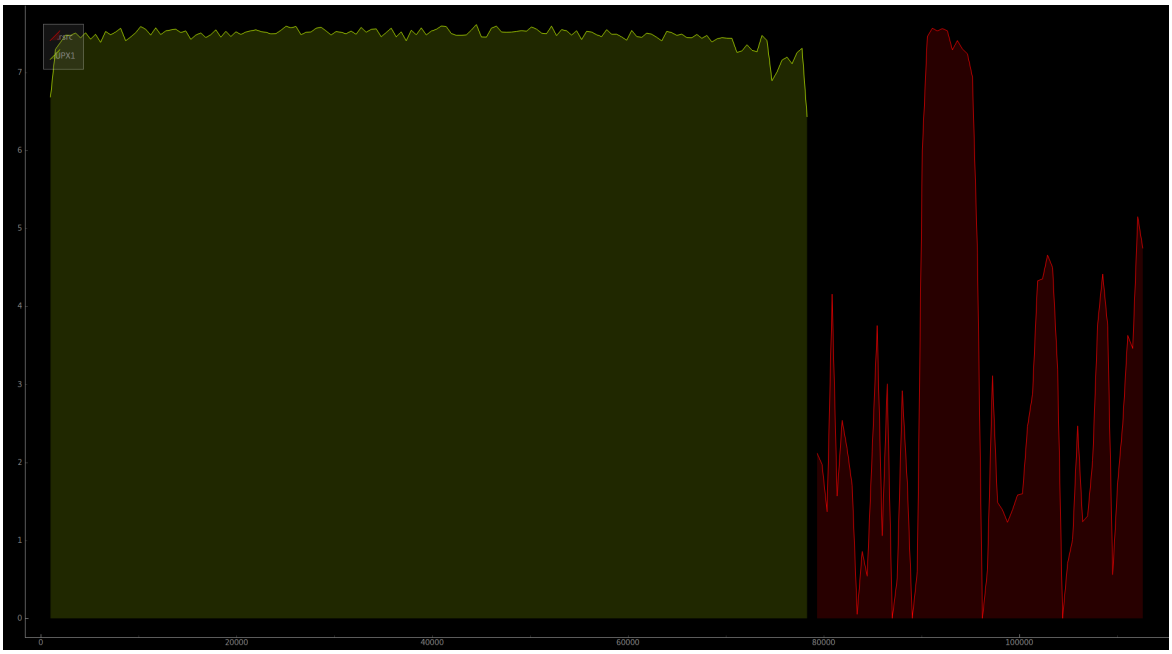


FIGURE 4.4 – Entropie des sections d'un exécutable compressé.

5. Bilan des objectifs et bilan personnel

5.1 Bilan des objectifs

Une grande partie des objectifs techniques ont été atteints au bout de ces six mois de stage

LIEF a été utilisé par un autre stagiaire de Quarkslab pour développer une probe d'analyse pour le produit **IRMA**[1]. C'est un outil qui permet d'analyser un exécutable par diffèrents anti-virus ou par des *probes* d'analyses. La bibliothèque a également servi à développer un logiciel de protection contre le *reverse engineering*. La documentation utilisateur n'est pas encore très complète et dans les prochains mois je vais avancer sur cette partie.
Il reste à automatiser les tests pour la partie Mach-O de LIEF.

5.2 Bilan personnel

Au cours de ce stage j'ai eu une très grande autonomie ce qui m'a permi de progresser sur plusieurs aspects du développement d'un projet : les tests, la documentation, le système de gestion de version Git ...
Je pense également avoir progressé sur le langage **C++11** et sur les idioms de ce langage. Il y a un an j'aurais écrit quelque chose comme :

```
std::vector<Section> sections = binary.get_section();
Section section_text;
for (uint32_t i = 0; i < sections.size(); ++i) {
  if (section[i].name() == ".text") {
    section_text = section[i];
    break;
  }
}
```

Maintenant j'ai plus le réflexe d'écrire :

```
const std::vector<Section>& sections = binary.get_section();
Section result;
auto itSectionText = std::find_if(
  std::begin(sections),
  std::end(sections),
  [] (const Section& section) { return section.name() == ".text" });

if (itSectionText != std::end(section)) {
  result = *itSectionText;
}
```

1. http://irma.quarkslab.com

J'ai également presenté LIEF lors de la journée des projets à l'ESIEE où j'ai eu de bons retours.

Suite aux travaux que j'avais menés l'année dernière, j'ai donné en avril dernier une conférence avec Jonathan Salwan à Bordeaux sur la *deobfuscation* de programmes avec Triton [2], un outils d'analyse dynamique développé par Jonathan. Les slides de notre présentation sont disponibles ici : http://triton.quarkslab.com/files/sthack2016-rthomas-jsalwan.pdf.

2. http://triton.quarkslab.com/

6. Conclusion

Ces six mois de stage au sein de Quarkslab m'ont permis d'enrichir mes connaissances sur plusieurs aspects de la sécurité informatique et plus particulièrement sur le developpement d'un projet. C'était l'occasion de mettre en application des concepts théoriques vu à l'ESIEE. Je remercie à cette occasion Serge Guelton pour ses nombreuses relectures de code et ses conseils. Cela m'a permis d'appréhender de nouveaux outils qui me seront utiles par la suite.

Enfin je tiens à remercier toute l'équipe de Quarkslab pour m'avoir accueilli une nouvelle fois, pour la confiance qu'ils m'ont accordé en tant que stagiaire et pour les nombreux échanges et conseils fort utiles qui m'ont été donnés.