Cet article a été publié dans les actes du Symposium sur la Sécurité des Technologies de l'Information et des Communication 2003.

Reverse engineering des systèmes ELF/INTEL

Julien Vanegue `may@epita.fr`
Sébastien Roy `seb@nbs-system.com`

SSTIC 2003

Organisation

SSTIC03 a été organisé par l'École Supérieure d'Électricité, l'École Supérieure et d'Application des Transmissions, le Commissariat à l'Énergie Atomique, la société Cartel Sécurité et le magazine de la sécurité informatique MISC.

Comité d'organisation

Christophe Bidan	Supélec
Philippe Biondi	Cartel Sécurité
Eric Detoisien	
Thiébaut Devergranne	
Eric Filiol	École Sup. et d'Application des Transmissions
Thierry Martineau	École Sup. et d'Application des Transmissions
Laurent Oudot	Commissariat à l'Énergie Atomique
Frédéric Raynal (Président)	MISC Magazine

Comité de programme

Gildas Avoine	EPFL
Christophe Bidan	Supélec
Renaud Bidou	
Philippe Biondi	Cartel Sécurité
Cédric Blancher	Cartel Sécurité
Patrick Chambet	Edelweb
Yves Correc	CELAR
Eric Detoisien	
Thiébaut Devergranne	
Eric Filiol	ESAT
Nicolas Fischbach	Colt Telecom / Securite.org
Sylvain Gombault	ENST Bretagne
Pascal Lointier	CLUSIF
Robert Longeon	CNRS
Thierry Martineau	ESAT
Marc Mouffron	EADS Telecom
Laurent Oudot	CEA/DAM
Frédéric Raynal	MISC magazine
Ludovic Rousseau	Gemplus
Marc Rybowicz	Université de Limoges
Franck Veysset	France Télécom R&D

Reverse engineering des systèmes ELF/INTEL

SSTIC03

Julien Vanegue may@epita.fr
Sebastien Roy seb@nbs-system.com

12 juin 2003

Table des matières

Chapitre 1

Introduction

À travers le reverse engineering, nous abordons de nombreuses problématiques de la sécurité informatique, notamment dans les domaines de manipulation binaire, analyse de code, et approche statique des protections automatiques contre les failles de sécurité de type buffer overflow. Le shell ELF [sc03] est utilisé comme support logiciel pour l'exposé. Cette boite a outil de manipulation d'objets ELF se compose de plusieurs composants :

1. Le shell [interpreteur de commandes]
2. La librairie de manipulation ELF : libelfsh
3. La librairie de manipulation de code : libasm

Le shell est un interpreteur de commandes, il permet de faire plusieurs types d'opérations sur les objets. Au niveau de ce composant, chaque opération de manipulation est representée sous forme d'une commande. Chaque opération peut etre appelée depuis la ligne de commande du shell UNIX, depuis une session ELFsh interactive, ou depuis un script ELFsh. Ceci est un script ELFsh :

```
$ cat relinject.esh

#!/usr/bin/elfsh

# Call regular shell
exec cat func2.c
exec gcc -c func2.c
exec cat exec.c
exec gcc exec.c
exec a.out

# Load ELF objects
load a.out
load func2.o
```

```
# raise a SIGSTOP, sometimes used for debugging
#stop

# Insert ET_REL into ET_EXEC
reladd 1 2

# Switch on a.out and list SHT
switch 1
s

# Grep in the symbol table for inserted symbols
sym reloc|func2

# Disassemble injected section
D func2.o.text%134

# Dump in hexa injected data objects
X testreloc

# Hijack puts() fonction using GOT infection
got puts
set 1.got[puts] puts_troj

# Save and execute the output object
save a.out.injected
exec a.out.injected
quit

$
```

Les mécanismes sous-jascents seront bien sûr detaillés. La librairie de manipulation ELF (libelfsh) permet plus de 260 opérations sous forme de fonctions C. La librairie de manipulation d'opcodes IA32 permet la lecture et modification des instructions, elle met a disposition une trentaine de fonctions. Chacune de ces librairies permet la lecture et l'ecriture.

Chapitre 2

Présentation des besoins

Il fût nécessaire de coder les composants bas niveau de notre modèle, en délaissant les composants existants, comme libopcodes et libbfd [GNU03a] .

Libopcodes n'est malheureusement qu'une librairie de désassemblage qui ne dispose pas vraiment d'une interface d'analyse. Libasm permet une representation interne détaillée de chaque instruction, ce qui permet de simplifier et modulariser les algorithmes, notamment pour les besoins de reencodage (modification) du code, et de tracing de flux.

Libbfd est une bonne librairie de translation binaire, mais dont le backend ELF reste assez pauvre dès que l'on veut disposer de certaines capacités éxotiques, comme le détournement de fonctions (ou tout autre capacité dont l'implémentation est dépendante de l'architecture), la relocation d'object non relogeable, ou encore le remapping.

Globalement, les outils GNU ont été codés pour garantir une interface unique sur un grand nombre de formats et d'assembleurs, ce qui est parfait du point de vue de l'éditeur de liens ou d'un outil comme objdump [GNU03a], mais insuffisant pour un outil d'analyse inverse. Ces dernières années, le besoin de translation binaire se fait de moins en moins ressentir, puisque les principaux UNIX tendent a implementer ELF comme format principal.

2.1 Développement sans source

Dans un cadre de production, ou dans un environnement hostile comme celui d'un serveur compromis ou vulnérable, il est fréquent de ne pas disposer de l'intégrale source du systeme d'exploitation (et de ses packages) sur la machine locale. Il est aussi fréquent de ne pas pouvoir transferer ces sources intégralement dans un laps de temps raisonnable - elles ne sont parfois tout simplement

pas disponible - parfois meme la recompilation n'est pas desireée. Qu'il s'agisse de logiciels propriétaires, ou d'exploits binaires, le besoin de disposer d'une interface claire de manipulation se fait ressentir. Nous nous attarderons sur deux concepts :

1. La résidence : étude de l'injection
2. L'interception : étude du détournement

Une telle interface permet a la fois l'ajout de fonctionnalités et la modification de fonctionnalitée existantes, le tout sans recompilation.

2.2 Protection systeme : ASLR

L'ASLR (Address Space Layout Randomization) permet de bloquer l'exploitation des attaques basées sur la prédiction d'adresses, comme les buffer overflow par return into libc ou return into PLT [Ner01] .

Le projet PaX [4] propose une implementation dynamique de l'ASLR, qui consiste a modifier directement le processus au moment de l'execution. Cette implementation au niveau kernel fait souffrir la machine protegee INTEL d'une chute de performance de 200% (sur une compilation de noyau Linux avec l'option RANDEXEC activé sur tous les binaires), dûe aux techniques utilisées (notamment par le détournement du handler de page fault depuis le kernel).

ELFsh propose une implementation experimentale de l'ASLR en userland sur les machines INTEL, qui permet de conserver la performance d'origine, puisque le remapping est effectué statiquement directement dans le binaire, en modifiant l'adresse de chaque segment mappé, et en relogeant l'executable apres avoir reconstruit ses tables de relocation (qui sont strippées par defaut dans les objets ELF ET_EXEC).

Il existe une implementation statique fiable de l'ASLR [et_dyn.zip] , qui consiste a relinker les binaires d'une distribution en librairies dynamiques, il faut pour cela disposer de tous les objets relogeables qui composent l'executable (tout les .o), ainsi que les Makefile d'origine du package, ce qui n'est pas applicable aisément dans un environnement de production, ou dans un environnement sans source.

Nous verrons que le problème reste ouvert, puisque notre implementation ASLR admet toujours des faux positifs de relocation, qui ne sont pas toujours évidents à détecter de manière automatique.

2.3 Audit binaire

La recherche et l'analyse de vulnérabilités est menée sur deux fronts distincts :

L'audit des sources qui consiste à rechercher les erreurs d'implémentation, les erreurs de traitement dans les sources du logiciel. Ce travail est largement mené par la communauté des développeurs de logiciels opensource. Cependant, si les sources ne sont pas disponibles, ce travail de recherche est réalisé au moyen de désassembleurs, et de debugger avec pour objectif l'analyse logique du code, la compréhension des différentes manipulations de données et leur incidence sur le déroulement du processus.

Il existe un autre axe de recherche plus empirique : l'analyse de coredump. Lorsque une erreur surgit dans un logiciel, il est possible de récuperer un fichier dit *core* contenant l'état du processus lors de l'erreur, c'est à dire la valeur des registres et le contenu de la mémoire au moment du crash. Il est souvent possible à partir de ces informations de dépister les erreurs de developpement en procédant à une analyse inverse afin de retrouver le chemin parcouru lors de l'exécution, jusqu'à l'interruption du processus, et comprendre les causes de la défaillance.

Chapitre 3

Rappel des prérequis

3.1 Format ELF

La documentation de référence sur le format ELF [Con98] est certainement la premiere source d'information crédible que vous devez consulter pour combler les carences sévères ;)

Le format ELF (Executable and Linking Format) est le format binaire utilisé, parfois seulement supporté (pour le cas d'OpenBSD), par la plupart des systèmes d'exploitation de type UNIX, comme FreeBSD, NetBSD, Solaris, IRIX ou encore Linux. Il admet 2 variantes, que sont ELF32 et ELF64, respectivement pour les architectures 32 bits et 64 bits.

Certains OS (IRIX, Solaris) désignés pour des processeurs 64 bits fournis d'un mode d'emulation 32 bits, peuvent utiliser à la fois ELF32 et ELF64. La différence majeure entre ces formats est la taille du pointeur (qui correspond a la taille du registre de la machine).

Il va de soi qu'un programme manipulant des objets ELF32 pourra manipuler des objets ELF64 si l'on remplace (avec sed par exemple) le préfixe 'Elf32' des noms de structures 32 bits, par le préfixe 'Elf64' des structures 64 bits.

Une implémentation capable de manipuler les 2 formats en meme temps devra etre structurée en couches, mettant en place une interface de haut niveau pour manipuler à la fois les objets 32 et 64 bits de maniere transparente. Dans cet exposé, toutes les manipulations seront effectuées sur les objets ELF32.

La premiere chose à lire dans le fichier est l'en-tete (header) ELF, qui fait 52 octets pour les binaires ELF32 :

```
bash-2.05# elfsh -f /bin/ls -e
```

```
[*] Object /bin/ls has been loaded (O_RDONLY)

[ELF HEADER]
[Object /bin/ls]

Architecture       :        Intel 80386   ELF Version        :             1
Object type        :  Executable object   SHT strtab index   :            26
Data encoding      :      Little endian   SHT foffset        :         44893
PHT foffset        :                 52   SHT entries number :            29
PHT entries number :                  6   SHT entry size     :            40
PHT entry size     :                 32   ELF header size    :            52
Entry point        :          0x8049420   (.text)

[*] Object /bin/ls unloaded

bash-2.05#
```

Nous ne détaillerons pas chaque champ pour toutes des structures du format ELF, l'inspection du fichier d'en-tete ELF [/usr/include/elf.h] est laissée au lecteur. Toutefois, les abbréviations PHT (Program Header Table) et SHT (Section Header Table) seront utilisées par la suite.

Le binaire peut etre examiné sous deux aspects :

1. Perspective du Systeme - Le noyau et le linker dynamique -
2. Perspective du Linker - GNULD, ELFsh -

3.1.1 Perspective de l'éditeur de liens

Une section est une partie finie de l'objet. Chaque section a son en-tête dans la SHT, chaque entrée est codée sur sizeof(Elf32_Shdr). Les noms de sections sont stockés dans une section spéciale appelée Section String Table (.shstrtab) que l'on peut trouver depuis la SHT, grace a l'index qui nous est indique dans le header ELF. Ainsi, le champ sh_name du header de section est un offset entier (en octets) depuis le début de la section .shstrtab, qui dévoile alors l'emplacement de la chaine qu'est le nom de la section.

```
bash-2.05# elfsh -f /bin/ls -s

[*] Object /bin/ls has been loaded (O_RDONLY)

[SECTION HEADER TABLE .::. SHT is not stripped]
[Object /bin/ls]

[00] (nil)      -------                   foffset(00000000) size(00000000)
```

```
[01] 0x80480f4  a------ .interp          foffset(00000244) size(00000019)
[02] 0x8048108  a------ .note.ABI-tag    foffset(00000264) size(00000032)
[03] 0x8048128  a------ .hash            foffset(00000296) size(00000600)
[04] 0x8048380  a------ .dynsym          foffset(00000896) size(00001296)
[05] 0x8048890  a------ .dynstr          foffset(00002192) size(00000875)
[06] 0x8048bfc  a------ .gnu.version     foffset(00003068) size(00000162)
[07] 0x8048ca0  a------ .gnu.version_r   foffset(00003232) size(00000128)
[08] 0x8048d20  a------ .rel.got         foffset(00003360) size(00000016)
[09] 0x8048d30  a------ .rel.bss         foffset(00003376) size(00000040)
[10] 0x8048d58  a------ .rel.plt         foffset(00003416) size(00000560)
[11] 0x8048f88  a-x---- .init            foffset(00003976) size(00000037)
[12] 0x8048fb0  a-x---- .plt             foffset(00004016) size(00001136)
[13] 0x8049420  a-x---- .text            foffset(00005152) size(00024636)
[14] 0x804f45c  a-x---- .fini            foffset(00029788) size(00000028)
[15] 0x804f480  a------ .rodata          foffset(00029824) size(00012092)
[16] 0x80533bc  aw----- .data            foffset(00041916) size(00000188)
[17] 0x8053478  aw----- .eh_frame        foffset(00042104) size(00000004)
[18] 0x805347c  aw----- .ctors           foffset(00042108) size(00000008)
[19] 0x8053484  aw----- .dtors           foffset(00042116) size(00000008)
[20] 0x805348c  aw----- .got             foffset(00042124) size(00000300)
[21] 0x80535b8  aw----- .dynamic         foffset(00042424) size(00000168)
[22] 0x8053660  -w----- .sbss            foffset(00042592) size(00000000)
[23] 0x8053660  aw----- .bss             foffset(00042592) size(00000680)
[24] (nil)      ------- .comment         foffset(00042592) size(00000988)
[25] (nil)      ------- .note            foffset(00043580) size(00000520)
[26] (nil)      ------- .shstrtab        foffset(00044100) size(00000227)

[*] Object /bin/ls unloaded

bash-2.05#
```

On remarque que ce binaire est strippé (il ne contient pas de table des symboles, une section de type SHT_SYMTAB, généralement appelée .symtab).

Il contient cependant d'autres sections :

1. Les sections de code (.text, .plt, .init, .fini)
2. Les sections de données (.data, .rodata, .bss)
3. La table des symboles importés et exportés (.dynsym)
4. La table de noms (.dynstr) et de hash (.hash) des symboles dynamiques
5. Les tables de relocation (.rel.*)
6. Les tableaux des constructeurs (.ctors) et destructeurs (.dtors)
7. Les sections reservées aux binaires dynamiques (.dynamic, .got, .plt)

L'espace d'adressage des sections est le fichier, on utilise des offsets depuis le debut du fichier (foffset, ou file offset) pour les réferencer. Au cours des différentes étapes de l'édition de liens, le linker sera amené à déplacer, et meme fusionner certaines sections. Par exemple, les bss sont fusionnées, et de nouvelles instances de .bss, .symtab et .rel* sont crées à partir de tous les .rel*, .symtab et .bss des objets relogeables (.o) lors de la creation d'un objet de type ET_EXEC (executable).

3.1.2 Perspective systeme

Sous une autre perspective, le système d'exploitation, plus particulièrement le noyau (il ouvre le fichier et mappe les premieres informations du processus en mémoire) et l'editeur de liens dynamique (qui mappe entierement le binaire, ainsi que tout les objets dont il dépend, comme les librairies *.so) interprètent le binaire comme une liste de segments.

Seule la Program Header Table (Table des en-têtes de segments, ou PHT) est utilisée dans cette perspective, la SHT n'est donc pas requise dans un binaire. Les entrées particulierement intéressantes de la PHT sont celles de type PT_LOAD (Loadable segment), qui correspondent aux segments mappés.

```
$ elfsh -q -f /bin/ls -p

[Program header table .:::. PHT]
[Object /bin/ls]

[00] 0x08048034 r-x memsz:000192 foff:000052 filesz:00192 alig:0004 PT_PHDR
[01] 0x080480F4 r-- memsz:000019 foff:000244 filesz:00019 alig:0001 PT_INTERP
[02] 0x08048000 r-x memsz:041916 foff:000000 filesz:41916 alig:4096 PT_LOAD
[03] 0x080533BC rw- memsz:001356 foff:041916 filesz:00676 alig:4096 PT_LOAD
[04] 0x080535B8 rw- memsz:000168 foff:042424 filesz:00168 alig:0004 PT_DYNAMIC
[05] 0x08048108 r-- memsz:000032 foff:000264 filesz:00032 alig:0004 PT_NOTE

[...]

$
```

Chaque entrée est codée sur sizeof(Elf32_Phdr). L'espace d'adressage des segments est le processus, on utilise des adresses virtuelles (l'adresse basse n'est pas standardisée, mais elle est identique pour tous les binaires pour un OS donné. Par convention, elle est de 0x08048000 sous Linux. Il est important de comprendre que les données indéxées par la SHT et la PHT sont les *mêmes*, on notera notamment les correspondances entre :

1. La section .interp et le segment de type PT_INTERP
2. La section .dynamic et le segment de type PT_DYNAMIC

3.1.3 La relocation

La relocation consiste à modifier l'image memoire des segments mappés, dans le but de les rendre executables. Chaque objet du processus contient des tables .rel*, qui sont en fait des tableaux d'entrées de relocation, indiquant ou patcher la memoire, et de quelle facon. En fait, le besoin de l'étape de relocation existe du fait qu'une librairie peut etre mappée à n'importe quel endroit de la memoire, et qu'il n'est pas possible de prédire les adresses relatives et absolues nécessaire à la relocation durant la compilation.

Voici une entrée de relocation :

```
typedef struct
{
  Elf32_Addr    r_offset;    /* Adresse */
  Elf32_Word    r_info;      /* reloc type et index du symbole dynamique */
} Elf32_Rel;
```

Cette version est utilisée sur les machines SPARC :

```
typedef struct
{
  Elf32_Addr    r_offset;     /* Adresse */
  Elf32_Word    r_info;       /* reloc type et index du symbole dynamique */
  Elf32_Sword   r_addend;     /* Increment final */
} Elf32_Rela;
```

On peut associer les types de relocation à des types de références, qui doivent etre pour la plupart mise à jour avant l'execution.

Les types de relocation sont differents selon l'architecture ; alors que les binaires ELF i386 disposent de 12 types, les binaires SPARC (32 et 64 bits confondus) en disposent d'une cinquantaine. Ils ne sont pas tous utilisés, et beaucoup sont dediés á la compatibilité avec les anciennes versions de l'architecture.

```
$ grep R_ /usr/include/elf.h | grep 386 | wc -l
    12
$ grep R_ /usr/include/elf.h | grep SPARC | wc -l
    56
$
```

En fait, il existe deux mecanismes de base : la relocation au moment du mapping mémoire, et la relocation au moment de l'execution. Ces deux mécanismes seront explicités par la suite, nous nous contenterons de citer quelques types de relocation pour le moment :

```
#define R_386_PC32      2   : Patch relatif par rapport au PC
#define R_386_COPY      5   : Copie du symbole au moment de l'execution
#define R_386_GLOB_DAT  6   : R\'esolution par GOT
#define R_386_JMP_SLOT  7   : R\'esolution par PLT
```

Ces deux derniers types sont respectivement utilisés pour les références aux variables externes (pour lesquelles une mecanisme utilisant la .got est nécessaire) et aux fonctions externes (pour lesquelles un mecanisme utilisant la .plt et la .got est necessaire).

3.2 L'assembleur IA32

3.2.1 Le processeur INTEL

L'assembleur est le langage du microprocesseur ; il a pour rôle la manipulation des données entre la mémoire et les registres du processeur. Les registres sont des accumulateurs à accès rapide sur lesquels il est possible de réaliser différentes opérations : addition, soustraction, décalage de bits, comparaison, etc.

Dans le but d'analyser automatiquement (ou de manière semi-automatique) le code binaire, il nous faut garder trace de toutes les transactions entre les variables. Ce travail est rendu assez complexe étant donné les nombreuses instructions (plus de 300 sur Pentium 4) et constructions d'instructions du compilateur, ainsi que par la complexité des programmes soumis à l'analyse.

Dans un processus, les variables peuvent être déclarées statiquement (elles sont alors directement présentées dans le binaire) ou dynamiquement (elles sont alors stockées sur la pile ou dans le tas, pour une durée de vie limitée).

Nous allons dans cette introduction, expliquer la base du langage assembleur, dans le but de familiariser le lecteur aux spécificités du langage, pour pouvoir par la suite expliciter les algorithmes d'analyse, qui sont basés sur une analyse linéaire du code dans le but de modéliser, sous forme de graphes, les flux d'exécution et de données.

Voici deux instructions assembleur du processeur *INTEL* :

`mov $42 , %eax`	$\longrightarrow$	stocke la valeur 42 dans l'accumulateur `EAX`
`add $100, 8(%ebp)`	$\longrightarrow$	additionne 0x100 au mot située à l'adresse (`EBP` + 8)

La première partie est le mnémonique de l'instruction. Il se compose de quelques lettres, souvent extraites de l'opération réalisée.

Les paramètres d'instructions sont appelés les opérandes. Elles peuvent indiquer notamment des valeurs absolues, des registres, ou des emplacements mémoire. Voici les registres incontournables de l'architecture *INTEL* :

On compte 8 registres généraux sur l'architecture IA32 : `EAX`, `ECX`, `EDX`, `EBX`, `ESP`, `EBP`, `ESI` et `EDI`, chacun d'une taille de 32 bits.

1. Le registre `EIP` contenant l'adresse de l'instruction à exécuter. Il ne peut être accédé ou modifié directement, mais au moyen de certaines instructions : `CALL`, `RET`, `JMP`, `JZ`, ...

   (a) `JMP` et les instructions de branchement conditionnel ne permettent pas de conserver l'état du flux de contrôle (le chemin d'exécution du programme), dans la mesure où elles modifient immédiatement le contenu du registre `EIP`.

   (b) `CALL` permet de modifier le flux de contrôle en appelant une procédure. L'adresse contenue dans `EIP` est empilée. Il sera alors possible de reprendre le cours du flux de contrôle là où il a été interrompu en utilisant l'instruction `RET`.

   (c) `RET` permet de rétablir le flux de contrôle là où il fût interrompu par un `CALL`. `RET` dépile la valeur courante de la pile et la stoque dans `EIP`.

2. Le registre `EFLAGS` est un registre de 32 bits dont les 16 premiers bits sont utilisés par différentes catégories d'instructions, notamment par les instrutions de manipulation de chaîne (*direction flag*) et lors des instructions de branchement conditionnel (*overflow*, *zero*, ...)

Direction Flag	$\longrightarrow$	Influence le sens de copie des chaînes.
Zero Flag	$\longrightarrow$	1 si le resultat de la derniere instruction est 0
Overflow Flag	$\longrightarrow$	1 si la derniere instruction a provoqué un dépassement de capacité du registre
Carry Flag	$\longrightarrow$	Contient le bit de retenue suite à certaines instructions de décallage de bits

   ...

   Nous reviendrons sur l'utilisation de ces flags au cours de l'article.

Certains accumulateurs généraux ont un rôle particulier, comme le registre `ESP`, qui est le pointeur de pile, et indique la prochaine adresse d'empilement.

La pile est un espace temporaire réalloué dynamiquement par le système à la demande. Cette mémoire est utilisée pour stocker les variables locales aux fonctions. Elle est par ailleurs utilisée pour sauvegarder les adresses de retour lors d'appels aux procédures (au moyen de l'instruction `CALL`). Ce registre est modifié à chaque instruction qui empile ou dépile une valeur (c'est-à-dire qui stocke et déstocke sur la pile).

Pour l'instruction `PUSH` (qui sauve une variable sur la pile), `ESP` est décrémenté de 4 et la valeur empilée est stockée à l'adresse contenue dans `ESP`. L'instruction `CALL` procède de la même manière pour sauvegarder l'adresse ou reprend le flux de contrôle. Pour l'instruction `POP` (qui restaure une valeur sauvée sur la pile et la place dans le registre indiqué par l'opérande), la valeur à dépiler est lue à l'adresse contenue dans ESP, puis ESP est incrémenté de 4.

Voici la représentation d'une pile au fur et à mesure de l'exécution de plusieurs instructions successives.

```
[1] 0x804b3a : mov $0x12345678, %ebx

 esp: 0xbfbff3b8 |0x????????| esp: 0xbfbff3b8
      0xbfbff3b4 |0x????????|
      0xbfbff3b0 |0x????????|

[2] 0x804b3ff : push %ebx

      0xbfbff3b8 |0x????????| esp: 0xbfbff3b4
 esp: 0xbfbff3b4 |0x12345678|
      0xbfbff3b0 |0x????????|

[3] 0x804b404 : call 0x804b469

      0xbfbff3b0 |0x????????| esp: 0xbfbff3b0
      0xbfbff3b4 |0x12345678|
[4]  esp: 0xbfbff3b0 |0x0804b409|

[5] 0x804b469 : ret

      0xbfbff3b0 |0x????????| esp: 0xbfbff3b4
 esp: 0xbfbff3b4 |0x12345678|
      0xbfbff3b0 |0x0804b409|

[6] 0x804b409 : add $4, %esp

 esp: 0xbfbff3b0 |0x????????|   esp: 0xbfbff3b8
      0xbfbff3b4 |0x12345678|
      0xbfbff3b0 |0x0804b409|
```

Historiquement, chaque registre possède une fonctionnalité précise, cela dit ils peuvent tous être utilisés par le programmeur pour un usage général.

`EBP` : registre contenant le pointeur de cadre de pile (*frame pointer*).

`EAX` : registre contenant la valeur de retour d'une fonction.

`ECX` : registre servant de compteur lors de certaines itérations.

`ESI` : registre contenant un pointeur vers une source (manipulation de données).

`EDI` : registre contenant un pointeur vers une destination (manipulation de données).

3.2.2 Format des instructions

Certaines instructions comme `RET`, `NOP`, `CLD`, `AAM` sont ne nécessitent pas d'opérandes.

Cependant, la majorité des instructions disposent de une et trois opérandes. Chaque opérande peut être composée d'un accumulateur, de références composées du contenu d'un registre de base et d'une valeur immédiate, éventuellement d'un registre d'index.

Voici un bref récapitulatif des instructions les plus couramment utilisées :

Instructions de		manipulations de données
`MOV SRC,DST`	$\longrightarrow$	charge le contenu de `SRC` dans `DST`
`LEA SRC,DST`	$\longrightarrow$	charge dans `DST` l'adresse de `SRC`
`INC OPERAND`	$\longrightarrow$	incrémente l'opérande `OPERAND`
`DEC OPERAND`	$\longrightarrow$	décrémente l'opérande `OPERAND`
`ADD SRC,DST`	$\longrightarrow$	`DST = DST + SRC`
`SUB SRC,DST`	$\longrightarrow$	`DST = DST - SRC`
`AND SRC,DST`	$\longrightarrow$	`DST = DST & SRC`
`OR SRC,DST`	$\longrightarrow$	`DST = DST | SRC`
`XOR SRC,DST`	$\longrightarrow$	`DST = DST ^ SRC`
Instructions de		manipulation de la pile.
`PUSH OPERAND`	$\longrightarrow$	empile le contenu de `OPERAND`
`POP OPERAND`	$\longrightarrow$	dépile et stocke la valeur dépilée dans `OPERAND`
Instructions		de comparaison.
`CMP SRC,DST`	$\longrightarrow$	soustrait `SRC` de `DST`
`TEST SRC,DST`	$\longrightarrow$	test le bit `SRC` de l'operande `DST`
Instructions de		de branchement.
`CALL OPERAND`	$\longrightarrow$	Appel de procédure a l'adresse `OPERAND`
`RET`	$\longrightarrow$	instruction de fin de procédure.
`JMP OPERAND`	$\longrightarrow$	Branchement a l'adresse `OPERAND`
`J* OPERAND`	$\longrightarrow$	Branchement conditionnel : ie `JE`, `JNL`

Voici un exemple de code en assembleur commenté.

`mov $0x8049450, %esi`	⟶	la valeur `0x8049450` est stockée dans `ESI`
`mov $0x1234567, (%esi)`	⟶	la valeur `0x1234567` est stockée à l'adresse contenue dans `ESI`
`mov 0x8049450, %eax`	⟶	stocke dans `EAX` la valeur contenue à l'adresse `0x8049450`

Cette séquence d'instructions stocke la valeur `0x1234567` à l'adresse `0x8049450` puis la lit la valeur à cette adresse et la stocke dans `EAX`.

Il est possible de référencer les variables par rapport à un registre donné par exemple lors d'accès indexés :

`mov 0x8(%ebp), %edx`	⟶	Assigne la variable située à `EBP`+8 dans `EDX`
`lea 0x0(,%edx,8),%ebx`	⟶	Stocke l'adresse de `ESI * 8` dans `EBX`

Ce type d'adressage est utilisé dans les fonctions où le référencement des variables locales et des arguments est effectué par rapport à `EBP` :

`0x8(%ebp)`	⟶	désigne le premier argument de la procédure
`0xc(%ebp)`	⟶	désigne le second argument de la procédure
`0xfffffffc(%ebp)`	⟶	Premier entier de 32 bits local
`0xfffffff8(%ebp)`	⟶	Second entier de 32 bits local
`0xfffffff4(%ebp)`	⟶	Troisième entier de 32 bits local

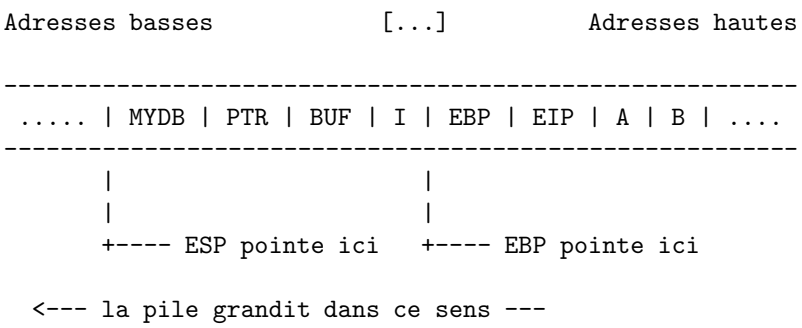
Note : les variables sont évoquées en terme d'entiers mais elles peuvent être d'une toute autre nature : structures, pointeurs, tableaux, ...

```
struct    db
{
  char    *name;
  int     age;
};

int         func(int a, int b)
{
  int      i;
  char      buf[16];
  char      *ptr;
  struct db mydb;

[CODE DE LA PROCEDURE]

  return (0);
}
```

Dans cet exemple, nous pouvons constater que :

`I`	est référencée par `0xfffffffc(%ebp)`	$\longrightarrow$	-4
`BUF`	est référencée par `0xffffffec(%ebp)`	$\longrightarrow$	-20
`PTR`	est référencée par `0xfffffff8(%ebp)`	$\longrightarrow$	-24
`MYDB`	est référencée par `0xfffffff0(%ebp)`	$\longrightarrow$	-32

Voici l'état de la pile pour la frame de cette fonction :



Nous allons voir comment garder trace de ces entrées/sorties, en regroupant les instructions selon leur encodage et leur fonction, puis en tracant symboliquement les accès aux variables du programme.

Chapitre 4

Le flux de contrôle sous UNIX

Le flux de contrôle est le chemin d'execution du programme. Il est intéressant de l'etudier pour plusieurs raisons. Dans le cadre de la manipulation binaire, nous sommes particulierement interessés par la mise en place de *hooks*, c'est à dire de points de détournement. Les utilisations sont multiples : audit, logging, déclenchement de fonctionalités supplémentaires, etc.

Certains modèles de hooks [Van01b] sont intéressants et peuvent etre appliqués en partie sur des binaires ELF. L'inconvénient de ce modèle est qu'il utilise un systeme dynamique de modification/restauration d'octets au début de la fonction, et qu'il est impossible de restaurer des octets originaux au moment de l'execution, puisque les fonctions d'un objet ELF sont contenues dans les segments executables en lecture seule (c'est a dire un PT_LOAD r-x). Bien qu'il soit possible de rendre une zone accessible en écriture pour un laps de temps fini par l'appel système *mprotect*, nous eviterons de l'utiliser puisqu'il est en général filtrer par les protections non-executable comme PaX.

De plus, le contexte utilisateur veut que les interfaces de résolution d'adresses des fonctions soient différentes pour les fonctions propres au binaire, et les fonctions des librairies dont dépend le binaire. Cette dernière utilise un mécanisme de patch au besoin, au moment de l'execution, en utilisant les sections .got et .plt [Van01a] [Ces00] : nous l'expliciterons dans la partie consacrée à l'interception dynamique.

4.1 Interception statique

Dans un premier temps, voyons les points d'entrée statiques du binaire, ils existent quelquesoit le contenu de l'executable, et peuvent être modifiés avec les commandes 'write' et 'set' dans ELFsh [sc03] . On distingue le point d'entrée

du programme (champ e_entry dans le header ELF, qui pointe sur la fonction _start, généralement située au debut de la section .text), la fonction _init (dans la section .init), la fonction _fini (dans la section .fini), les constructeurs (section .ctors), et les destructeurs (section .dtors), et enfin la fonction main.

```
$ elfsh -f ./a.out -D _start

08048310 [foff: 784] _start + 0    xor        %ebp,%ebp
08048312 [foff: 786] _start + 2    pop        %esi
08048313 [foff: 787] _start + 3    mov        %esp,%ecx
08048315 [foff: 789] _start + 5    and        $FFFFFFF0,%esp
08048318 [foff: 792] _start + 8    push       %eax
08048319 [foff: 793] _start + 9    push       %esp
0804831A [foff: 794] _start + 10   push       %edx
0804831B [foff: 795] _start + 11   push       $<.fini>
08048320 [foff: 800] _start + 16   push       $<.init>
08048325 [foff: 805] _start + 21   push       %ecx
08048326 [foff: 806] _start + 22   push       %esi
08048327 [foff: 807] _start + 23   push       $<main>
0804832C [foff: 812] _start + 28   call       <__libc_start_main@@GLIBC_2.0>
08048331 [foff: 817] _start + 33   hlt
08048332 [foff: 818] _start + 34   nop
08048333 [foff: 819] _start + 35   nop

$ elfsh -f ./a.out -D '^\.init:0%37' -D '^\.fini:0%28'

08048298 [foff: 664] .init + 0     push     %ebp
08048299 [foff: 665] .init + 1     mov      %esp,%ebp
0804829B [foff: 667] .init + 3     sub      $14,%esp
0804829E [foff: 670] .init + 6     push     %ebx
0804829F [foff: 671] .init + 7     call     <_init + 12>
080482A4 [foff: 676] .init + 12    pop      %ebx
080482A5 [foff: 677] .init + 13    add      $129C,%ebx
080482AB [foff: 683] .init + 19    call     <Letext>
080482B0 [foff: 688] .init + 24    call     <frame_dummy>
080482B5 [foff: 693] .init + 29    call     <__do_global_ctors_aux>
080482BA [foff: 698] .init + 34    pop      %ebx
080482BB [foff: 699] .init + 35    leave
080482BC [foff: 700] .init + 36    ret

0804842C [foff: 1068] .fini + 0    push     %ebp
0804842D [foff: 1069] .fini + 1    mov      %esp,%ebp
0804842F [foff: 1071] .fini + 3    sub      $14,%esp
08048432 [foff: 1074] .fini + 6    push     %ebx
08048433 [foff: 1075] .fini + 7    call     <_fini + 12>
```

```
08048438 [foff: 1080] .fini + 12   pop      %ebx
08048439 [foff: 1081] .fini + 13   add      $1108,%ebx
0804843F [foff: 1087] .fini + 19   nop
08048440 [foff: 1088] .fini + 20   call     <__do_global_dtors_aux>
08048445 [foff: 1093] .fini + 25   pop      %ebx
08048446 [foff: 1094] .fini + 26   leave
08048447 [foff: 1095] .fini + 27   ret

$ elfsh -f ./a.out -D __do_global_ctors_aux%36

08048400 [foff: 1024] + 0    push      %ebp
08048401 [foff: 1025] + 1    mov       %esp,%ebp
08048403 [foff: 1027] + 3    sub       $14,%esp
08048406 [foff: 1030] + 6    push      %ebx
08048407 [foff: 1031] + 7    mov       $<.ctors>,%ebx
0804840C [foff: 1036] + 12   cmp       $FFFFFFFF,<.ctors>
08048413 [foff: 1043] + 19   je        <__do_global_ctors_aux + 33>
08048415 [foff: 1045] + 21   mov       (%ebx),%eax
08048417 [foff: 1047] + 23   call      *%eax
08048419 [foff: 1049] + 25   add       $FFFFFFFC,%ebx
0804841C [foff: 1052] + 28   cmp       $FFFFFFFF,(%ebx)
0804841F [foff: 1055] + 31   jne       <__do_global_ctors_aux + 21>
08048421 [foff: 1057] + 33   pop       %ebx
08048422 [foff: 1058] + 34   leave
08048423 [foff: 1059] + 35   ret

$ elfsh -D __do_global_dtors_aux%79

08048360 [foff: 864] + 0     push      %ebp
08048361 [foff: 865] + 1     mov       %esp,%ebp
08048363 [foff: 867] + 3     sub       $8,%esp
08048366 [foff: 870] + 6     cmp       $0,<completed.4>
0804836D [foff: 877] + 13    jne       <__do_global_dtors_aux + 77>
0804836F [foff: 879] + 15    jmp       <__do_global_dtors_aux + 35>
08048371 [foff: 881] + 17    mov       <p.3>,%eax
08048376 [foff: 886] + 22    lea       4(%eax),%edx
08048379 [foff: 889] + 25    mov       %edx,<p.3>
0804837F [foff: 895] + 31    mov       (%eax),%eax
08048381 [foff: 897] + 33    call      *%eax
08048383 [foff: 899] + 35    mov       <p.3>,%eax
08048388 [foff: 904] + 40    cmp       $0,(%eax)
0804838B [foff: 907] + 43    jne       <__do_global_dtors_aux + 17>
0804838D [foff: 909] + 45    mov  $<__deregister_frame_info@@GLIBC_2.0>, %eax
08048392 [foff: 914] + 50    test      %eax,%eax
08048394 [foff: 916] + 52    je        <__do_global_dtors_aux + 67>
08048396 [foff: 918] + 54    add       $FFFFFFF4,%esp
```

```
08048399 [foff: 921] + 57    push      $<.eh_frame>
0804839E [foff: 926] + 62    call      <__deregister_frame_info@@GLIBC_2.0>
080483A3 [foff: 931] + 67    mov       $1,<completed.4>
080483AD [foff: 941] + 77    leave
080483AE [foff: 942] + 78    ret

$ elfsh -f /bin/telnet -ctors -dtors

[Constructors array .::. CTORS]
[Object /bin/telnet]

[00]  0xffffffff        <?>
[01]   0x804e0b0        <.text + 15088>
[02]   0x804e970        <.text + 17328>
[03]   0x805291c        <.text + 33628>
[04]   0x80547d0        <.text + 41488>
[05]   0x8054bec        <.text + 42540>
[06]       (nil)        <?>

[Destructors array .::. DTORS]
[Object /bin/telnet]

[00]  0xffffffff        <?>
[01]   0x804e0c8        <.text + 15112>
[02]   0x804e988        <.text + 17352>
[03]   0x8052934        <.text + 33652>
[04]   0x80547e8        <.text + 41512>
[05]   0x8054c04        <.text + 42564>
[06]       (nil)        <?>

$
```

Grâce à ces dumps, nous déduisons le chemin d'execution du programme :

```
DEBUT DU PROCESS

_start() =>
          __libc_start_main() =>
               [...] *
               _init() =>
                __do_global_ctors_aux() =>
                                        ctors[01]()
                                        ctors[02]()
                                        ctors[..]()
                                        ctors[NN]()
```

```
                main()  =>
                                          [...]
                exit()  =>
                Tables __atexit/__on_exit =>
                          __do_global_dtors_aux() =>
                                          dtors[01]()
                                          dtors[02]()
                                          dtors[..]()
                                          dtors[NN]()
FIN DU PROCESS
```

La fonction __libc_start_main() se trouve dans la libc, ou elle est chargée d'initialiser l'espace mémoire du processus. Cette étape, marquée d'une étoile, est hautement dépendante du système d'exploitation ; elle est en général tres peu [Van01c] documentée : l'éditeur de liens dynamique y prend la main pour la première fois, il mappe alors les dépendences et effectue leurs relocations, puis appelle les constructeurs par la fonction _init(), enfin appelle la fonction main(). Au retour du main(), exit() est directement appelée avec la valeur de retour, qui à son tour se charge de lancer tous les destructeurs, le processus est alors terminé.

4.2 Interception dynamique

Le rôle de l'éditeur de liens dynamique ne se résume pas à la construction du processus avant l'appel du main(). Il est aussi chargé d'assurer la résolution d'adresses entre les objets, ce qui leur permet d'assurer un transfert de contrôle au besoin, au moment de l'execution. En d'autres termes, il permet de résoudre les adresses des fonctions et variables externes à l'objet courant, de manière a pouvoir s'en servir.

Voyons de plus près les caractèristiques d'une librairie dynamique :

```
$ elfsh -f /lib/ld-linux.so.2 -s | grep -v nil

[01] 0x94       a------ .hash               foff:000148 sz:000836
[02] 0x3d8      a------ .dynsym             foff:000984 sz:001760
[03] 0xab8      a------ .dynstr             foff:002744 sz:003468
[04] 0x1844     a------ .gnu.version        foff:006212 sz:000220
[05] 0x1920     a------ .gnu.version_d      foff:006432 sz:000200
[06] 0x19e8     a------ .rel.data           foff:006632 sz:000096
[07] 0x1a48     a------ .rel.got            foff:006728 sz:000536
[08] 0x1c60     a------ .rel.plt            foff:007264 sz:000224
[09] 0x1d40     a-x---- .plt                foff:007488 sz:000464
[10] 0x1f10     a-x---- .text               foff:007952 sz:065161
```

```
[11] 0x11da0    a------ .rodata             foff:073120 sz:010648
[12] 0x15740    aw----- .data               foff:083776 sz:000180
[13] 0x157f4    aw----- .got                foff:083956 sz:000392
[14] 0x1597c    aw----- .dynamic            foff:084348 sz:000136
[15] 0x15a04    -w----- .sbss               foff:084512 sz:000000
[16] 0x15a20    aw----- .bss                foff:084512 sz:001004

$ elfsh -f /lib/ld-linux.so.2 -p

[Program header table .::. PHT]
[Object /lib/ld-linux.so.2]

[00] 0x00000000 -> 0x00014738 r-x memsz:83768 foff:00000 => Loadable segment
[01] 0x00015740 -> 0x00015E0C rw- memsz:01740 foff:83776 => Loadable segment
[02] 0x0001597C -> 0x00015A04 rw- memsz:00136 foff:84348 => Dynamic info

[Program header table .::. SHT correlation perspective]
[Object /lib/ld-linux.so.2]

[*] SHT is not stripped

[00] {PT_LOAD}          .hash .dynsym .dynstr .gnu.version .gnu.version_d
.rel.data .rel.got .rel.plt .plt .text .rodata
[01] {PT_LOAD}         .data .got .dynamic
[02] {PT_DYNAMIC}      .dynamic

$
```

[NOTE : les en-têtes de sections non mappées ont ete retirées]

On distingue clairement que les adresses dans la SHT et la PHT sont relatives par rapport à l'adresse base de la librairie, c'est à dire par rapport au début du fichier (offset de fichier 0), ce qui explique que l'étape de relocation soit obligatoire pour les librairies ET_DYN, et donc un peu plus complexe que la relocation des binaires ET_EXEC. On distinguera notamment des sections de relocation supplémentaire comme *.rel.text* ou *.rel.data* dans ces bibliothèques.

Quant à elle, la relocation à la volée s'effectue aussi bien sur les objets de type ET_EXEC que sur les objets ET_DYN, en utilisant les sections *.got*, *.plt*, *.rel.got*, *.rel.plt*, *.dynsym* et *.dynstr*. Cette étape de relocation va permettre de modifier l'image mémoire de l'objet au moment ou le programme en a besoin, et non au moment du mapping initial. En fait, elle se resume à la mise a jour de pointeurs dans un tableau (la section `.got`) de manière indépendante pour chacune des entrée, lors de leur première utilisation.

Chaque entrée de .got correspond à l'adresse d'une fonction externe a l'objet, mais dont il depend. Par exemple, un programme qui depend de la libc, devra

reloger les entrées de *.got* pour toutes les fonctions de la libc qu'il utilise. Il n'est pas possible de prédire ces adresses à la compilation pour la même raison que cités precedemment, c'est à dire qu'une librairie peut être mappée n'importe ou dans le processus. Voici à quoi ressemble une section `.got` dans le binaire :

```
$ elfsh -f /bin/ls -got '0]'

[Global Offset Table .::. GOT]
[Object /bin/ls]

[000] 0x080535B8        .dynamic
[010] 0x08049036        _obstack_begin + 6
[020] 0x080490D6        __errno_location + 6
[030] 0x08049176        fputs + 6
[040] 0x08049216        printf + 6
[050] 0x080492B6        __lxstat64 + 6
[060] 0x08049356        isatty + 6
[070] 0x080493F6        fwrite + 6

$ elfsh -f /bin/ls -got | wc -l
    85
$
```

4.3 Edition de liens dynamique

Les trois premières entrées de .got sont des entrées spéciales, dont nous allons éclaircir le fonctionnement. En fait, chaque entrée de .got pointe par default (c'est a dire quand la .got n'a pas encore ete relogée dans le binaire) sur l'entrée de la section .plt pour cette fonction, exceptés les trois premières, qui sont respectivement utilisées pour contenir l'adresse de la section *.dynamic*, l'adresse de *dl-resolve*, et l'adresse de la structure *linkmap* de l'objet courant [Van01c] .

Voyons à quoi ressemble une section *.plt* sur architecture `INTEL` :

```
08048FB0 .plt + 0                       push        .got + 4
08048FB6 .plt + 6                       jmp         *.got + 8
08048FBC .plt + 12                      add         %al,(%eax)
08048FBE .plt + 14                      add         %al,(%eax)

08048FC0 .plt + 16                      jmp         *.got + 12
08048FC6 .plt + 22                      push        $0
08048FCB .plt + 27                      jmp         .plt
```

```
08048FD0 .plt + 32                      jmp          *.got + 16
08048FD6 .plt + 38                      push         $8
08048FDB .plt + 43                      jmp          .plt

08048FE0 .plt + 48                      jmp          *.got + 20
08048FE6 .plt + 54                      push         $10
08048FEB .plt + 59                      jmp          .plt

[...]

08049020 .plt + 112                     jmp          *.got + 36
08049026 .plt + 118                     push         $30
0804902B .plt + 123                     jmp          .plt>

08049030 .plt + 128                     jmp          *.got + 40
08049036 .plt + 134                     push         $38
0804903B .plt + 139                     jmp          .plt

[...]
```

Voici la meme chose sous SPARC :

```
[...]

00031484 <.plt>:
  31484:       00 00 00 00     unimp  0
  31488:       00 00 00 00     unimp  0
  3148c:       00 00 00 00     unimp  0

  31490:       00 00 00 00     unimp  0
  31494:       00 00 00 00     unimp  0
  31498:       00 00 00 00     unimp  0

  3149c:       00 00 00 00     unimp  0
  314a0:       00 00 00 00     unimp  0
  314a4:       00 00 00 00     unimp  0

  314a8:       00 00 00 00     unimp  0
  314ac:       00 00 00 00     unimp  0
  314b0:       00 00 00 00     unimp  0

  314b4:       03 00 00 30     sethi  %hi(0xc000), %g1
  314b8:       30 bf ff f3     b,a   0x31484
  314bc:       01 00 00 00     nop
```

```
   314c0:       03 00 00 3c     sethi  %hi(0xf000), %g1
   314c4:       30 bf ff f0     b,a   0x31484
   314c8:       01 00 00 00     nop

   314cc:       03 00 00 48     sethi  %hi(0x12000), %g1
   314d0:       30 bf ff ed     b,a   0x31484
   314d4:       01 00 00 00     nop

 [...]
```

Bien qu'elle contienne du code, on peut dire que la *.plt* est une table, puisque chacune de ses entrées fait 16 octets (12 octets sous `SPARC`) et contient 3 instructions [jmp, push, jmp] ou [sethi, ba, nop] sous `SPARC`. On notera que la première entrée est speciale sur les 2 architectures (les 4 premieres pour SPARC), nous allons expliquer en quoi tout de suite en prenant pour temoin, la machine INTEL.

Les mechanismes sont similaires, exceptes le fait que la *.got* n'est pas utilisée pour le transfert de contrôle sous *SPARC*, mais que l'entrée de la *.plt* est patchée directement.

Lors de l'appel d'une fonction externe a l'objet (par exemple, l'appel à printf de la libc), le programme doit passer par un 'call' sur l'entrée de la *.plt* correpondante à la fonction. Par exemple, la derniere entrée affichée (adresse 08049030) correspond à la dixième entrée de la *.got* ; le premier *jmp* utilise la valeur de got[10] . On appelle ce type de jump *indirect* car il n'utilise pas une valeur immédiate codée dans l'instruction elle même, mais une valeur lue à l'adresse indiquée par son opérande.

Ainsi, la valeur par default de l'entrée de la *.got* va etre utilisée par l'entrée courante de la *.plt* . On constate que l'adresse qu'elle conient pointe sur la deuxiéme instruction de l'entrée correspondante dand la *.plt*. Cette 2e instruction 'push' permet de sauver sur la pile l'offset de l'entrée de relocation correspondant à l'entrée de la *.got* que nous devons patcher.

L'offset $38 correspond à un décalage de 0x38 octets par rapport à l'adresse basse de la table de relocation (qui est disponible dans le segment *PT_DYNAMIC* pour le linker dynamique). Chaque entrée de la table (*.rel.plt*) est codée sur sizeof(Elf32_Rel), ou sizeof(Elf32_Rela), l'entrée a l'offset 0x38 sera utilisée par dl-resolve() pour connaître les informations de relocation relative a cette entrée.

Enfin, la premiere entrée (entrée speciale) de la *.plt* est appelée par le deuxième *jmp*, elle même utilise la troisième entrée de la *.got* pour invoquer la fonction de relocation à la volée, qui se trouve dans le linker dynamique. Cette troisieme entrée got[2] est également réservée : elle est renseignée avant le lancement de la fonction main(). La fonction chargée de la mise-a-jour de la

.got s'appelle en general dl-resolve(), même si ce n'est pas standardisé.

Quand l'entrée de la *.got* est remplie, la fonction de resolution a la volée appele la fonction externe du programme, dans notre cas printf() . Ainsi les prochains appels à printf() utiliseront directement la valeur renseignée de la *.got*, sans repasser par le linker dynamique et sa fonction *dl-resolve*, puisque le premier *jmp* de l'entrée de la *.plt* utilisera maintenant la valeur renseignée de l'entrée de la *.got* pour printf.

Il est donc aisé de détourner une fonction en modifiant son entrée dans la GOT, ainsi dl-resolve() ne sera jamais appelé pour cette entrée, et une fonction tierce pourra etre appelée a la place.

Détournons malloc :

```
[ELFsh-0.5b5]$ load /bin/ls

[*] New object /bin/ls loaded on Thu May  1 00:18:15 2003

[ELFsh-0.5b5]$ got malloc

[Global Offset Table .::. GOT]
[Object /bin/ls]

[023] 0x08049106        <malloc + 6>

[ELFsh-0.5b5]$ set 1.got[23] 0x42424242

[*] Field set succesfully

[ELFsh-0.5b5]$ save /tmp/ls.bad

[*] Object /tmp/ls.bad save successfully

[ELFsh-0.5b5]$ quit

[*] Unloading object 1 (/bin/ls) *

        Good bye ! .::. The ELF shell 0.5b5

$ elfsh -f /tmp/ls.bad -got 42424242

[*] Object /tmp/ls.bad has been loaded (O_RDONLY)
```

```
[Global Offset Table .::. GOT]
[Object /tmp/ls.bad]

[023] 0x42424242        <?>

[*] Object /tmp/ls.bad unloaded

 $ /tmp/ls.bad
Segmentation fault (core dumped)
$ gdb /tmp/ls.bad --core=/tmp/core

[...]

#0  0x42424242 in ?? ()
(gdb) bt
#0  0x42424242 in ?? ()
#1  0x0804e1d6 in strcpy () at ../sysdeps/generic/strcpy.c:31
#2  0x0804a29c in strcpy () at ../sysdeps/generic/strcpy.c:31
#3  0x08049634 in strcpy () at ../sysdeps/generic/strcpy.c:31
#4  0x400502eb in __libc_start_main (main=0x80495a8 <strcpy+408>, argc=1, ...)
(gdb)
```

A noter qu'il est possible de forcer le renseignement complet de la *.got* avant que le main() ne s'execute, pour cela, il suffit de creer une variable d'environnement *LD_BIND_NOW* et de la mettre à 1, elle sera directement interpretée par le linker dynamique au lancement du programme. Dans ce cas, toutes les entrées de la *.got* sont mise a jour au démarrage du programme, et le détournement par *.got* n'est pas possible. Il faut alors patcher la section *.plt* directement pour détourner une fonction dynamique [Ces00], et ainsi utiliser une *.got* alterne plutôt que celle renseignée au début du processus.

Chapitre 5

Assembleur et sémantique

5.1 Désassemblage et Réassemblage

5.1.1 Format des instructions

Une instruction peut débuter par certains préfixes. Ces préfixes sont au nombre de onze, classées en trois types.

Préfixe de segmentation Normalement, les instructions opèrent sur des données réferencées par un registre défini (à savoir `ds`) pour la manipulation de données. Il est cependant possible de forcer les accès par un autre sélecteur de segment au moyen de ces préfixes :

- `ds` (0x3e)
- `cs` (0x2e)
- `es` (0x26)
- `ss` (0x36)
- `fs` (0x64)
- `gs` (0x65)

Par exemple :

```
26 a1 20 56 04 08        mov %es:0x8045620, %eax
```

Préfixe de répetition : Il est destiné à répéter les instructions de manipulation de chaînes, à savoir :

`movsb` (0xa4), `stosb` (0xaa), `lodsb` (0xac), `cmpsb` (0xa6), `scasb` (0xae), etc.
Ces instructions effectuent N iterations, où N est stocké dans l'accumulateur `ECX`.

Les préfixes possibles sont :

- `REPE` (0xf2). condition d'arrêt : zf[1] = 0
- `REPNE` (0xf3). condition d'arrêt : zf = 1

Par exemple :

```
f2 ae                    repnz    scas %es:(%edi),%al
```

Préfixe d'altération de la taille des données : Celui-ci joue un rôle lors de l'interprétation de la taille de certaines opérandes ; celles de type `ASM_OTYPE_VECTOR` peuvent être utilisées comme des opérandes de 16 bits ou 32 bits selon qu'elles sont prefixées ou non par 0x66 et selon le mode par défaut du processeur (16 ou 32 bits).

- `OPSIZE` (0x66). Exemple :
  ```
  50                       push %eax
  66 50                    push %ax
  ```
- `ADDSIZE` (0x67).
  Les opérandes de type `ASM_OTYPE_ADDRESS` sont de 32 bits. Cependant si l'instruction est préfixée par 0x67, alors la taille des opérandes est de 48 bits.

- `LOCK` (0xf0).
  Le préfixe `LOCK` sert à réaliser des accès concurrentiels sur des architectures SMP. Lorsque une instruction est prefixée par `LOCK`, l'instruction est executée atomiquement, ce qui permet d'utiliser de la mémoire partagée entre différents processeurs sur une architecture SMP.

Opcodes Après les préfixes se trouve alors l'opcode de l'instruction. Pour chaque octet de `0x00` à `0xff` est associé une instruction ou un groupe dínstructions donné.

Certains opcodes englobent l'instruction et ses paramètres comme `0x91` : `xchg %eax, %ecx`

Pour d'autres instructions, les oérandes sont encodées dans les octets qui suivent l'opcode :
Branchement conditionnel x86 permettant d'effectuer un branchement à plus 127 ou -128 octets :

```
74 42                    jz +42
```

[1] flag zéro

Le jeu d'instruction x86 a été complété au fur et à mesure par de nouvelles instructions. Ainsi, à partir du *i386*, de nouvelles instructions préfixées par `0x0f` ont été implémentées, dont certaines sont des extensions des instructions existant déjà.

Branchement conditionnel permettant d'effectuer un branchement à n'importe quel offset dans un espace d'adressage 32 bits :

```
0f 84 42 01 00 00       jz +142
```

Structure d'une instruction dans *libasm* Voici la structure d'une instruction telle qu'elle est décrite pas *libasm* :

```
struct s_asm_instr
{
   /*    pointer to instruction buffer   */
 u_char                *ptr_prefix;
 u_char *ptr_instr;
   /*    instruction/operands full lengh */
 u_int                 len;
   /*    internal processor structure    */
 asm_processor         *proc;
   /*    instruction                     */
 int                   instr;
   /*    instruction type                */
 int                   type;
   /*    instruction prefix              */
 int                   prefix;
   /*    operands                        */
 asm_operand           op1;
 asm_operand           op2;
 asm_operand           op3;
};
```

5.1.2 Format des opérandes

Après l'instruction suivent les éventuelles opérandes. Les registres sont définis de la manière suivante pour la suite de l'article :

```
enum e_regset_r32
{
 ASM_REG_EAX,/* 000 */
 ASM_REG_ECX,/* 001 */
 ASM_REG_EDX,/* 010 */
```

```
  ASM_REG_EBX,/* 011 */
  ASM_REG_ESP,/* 100 */
  ASM_REG_EBP,/* 101 */
  ASM_REG_ESI,/* 110 */
  ASM_REG_EDI /* 111 */
 } e_asm_reg32;
```

Chaque jeu de registres est défini sur 3 bits, d'où un maximum de 8 registres pour chaque catégorie : registres 8 bits, 16 bits, 32 bits, registre mmx, de contrôle, de segment, ...

Il existe deux grandes catégories d'opérandes : celles intégrées à l'instruction et celles qui sont distinctes de l'opcode de l'instruction. Il a été nécessaire de les typer différemment afin d'offrir plus de souplesse dans les capacités d'altération de certaines operandes.

Les opérandes fixées à l'instruction : Certaines instructions admettent des opérandes fixes et invariables. Par exemple, les opcodes 0x91 à 0x97 correspondent aux instructions

`xchg %eax, %ecx` à `xchg %eax, %edi`

Il existe d'autres cas d'instructions pour lesquelles la singularité des opérandes ne permet ni ne justifie un typage :

```
06        push %es
07        pop %es
```

Pour ces instructions, le typage choisi est `ASM_OTYPE_FIXED`.

Les registres intégrés à l'instruction : Ici, on observe que les 3 bits de poids faible de l'opcode désignent le registre utilisé :

```
50      inc %eax
51      inc %ecx
57      inc %edi
```

On trouve de nombreuses instructions respectant cet encodage :

`inc %e*` : 0x40 (`inc %eax`) $\longrightarrow$ 0x47 (`inc %edi`)
`dec %e*` : 0x48 (`dec %eax`) $\longrightarrow$ 0x4f (`dec %edi`)
`push %e*` : 0x50 (`push %eax`) $\longrightarrow$ 0x57 (`push %edi`)
`push %e*` : 0x58 (`pop %eax`) $\longrightarrow$ 0x5f (`pop %edi`)

Pour ces instructions, le typage choisi est `ASM_OTYPE_OPMOD`.

Les valeurs immédiates : Il existe parfois des valeurs immédiates fixées comme par exemple :

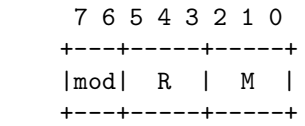
```
cd | 80                int $0x80
68 | 85 b4 07 08       push $0x0807b485
83 | c4 | 10           add $0x10, %esp
```

Pour ces instructions, le typage choisi est `ASM_OTYPE_IMMEDIATE`.

Registre encodé dans l'octet *mod R/M* : Si l'opérande est de type `ASM_OTYPE_GENERAL`, elle possède un registre général encodé dans le champ `M` de l'octet *mod R/M* suivant l'opcode de l'instruction.

Si l'operande est de type `ASM_OTYPE_REGISTER`, elle possède un registre général encodé dans le champ `R` de l'octet *mod R/M* suivant l'opcode de l'instruction.

Opérande encodée par un octet *mod R/M* : L'opérande est de type `ASM_OTYPE_ENCODED`.



On se réfèrera à la table *32bit mod R/M byte* en annexe pour visualiser l'ensemble des possibilités d'encodage.

Deux champs sont à associer : les 2 bits du champ `mod` qui définissent le type de l'opérande et les bits `M` qui permettent de spécifier le registre de base utilisé par l'instruction.

Le champ `R` est quant à lui utilisé soit pour définir l'instruction encodée comme dans les instructions préfixées par 0x80, soit pour une autre opérande de l'instruction.

Lorsque le champ `mod` vaut `00`, le champ `M` est utilisé comme registre de référencement. L'instruction opère alors sur le contenu de l'adresse spécifiée par `M`.

```
89 e5           mov %eax,(%edx)
```

```
   +---+---+---+   mod: 11 M
   |mod| R | M |   R  : EAX
e5:| 00|000|010|   M  : ESP
   +---+---+---+
```

Lorsque le champ mod vaut `01`, le champ `M` est utilisé comme registre de base et un offset d'un octet y est ajouté :

```
   8b 73 08          mov 0x8(ebx), %esi

   +---+---+---+   mod: 1 -> [M + sbyte]
   |mod| R | M |   R  : ESI
73:| 01|010|011|   M  : EBX
   +---+---+---+
```

Lorsque le champ mod vaut `10`, le champ `M` est utilisé comme registre de base et un offset de 4 octets est ajouté au contenu du registre de base.

Lorsque le champ mod vaut `11`, le champ `M` est utilisé comme registre de base et c'est son contenu sur lequel opère l'instruction.

```
   89 e5             mov %esp,%ebp

   +---+---+---+   mod: 11 M
   |mod| R | M |   R  : EBP
e5:| 11|110|101|   M  : ESP
   +---+---+---+
```

L'octet SIB Lorsque le champ `M` de l'octet `mod R/M` contient la valeur 4 (`ESP`) et que le champ `mod` n'a pas pour contenu `00`, nous avons $SIB = base + index \times 2^{scale}$.

```
7 6 5 4 3 2 1 0
+---+-----+-----+
|sc.|index|base |
+---+-----+-----+
```

base : registre de base

index : registre d'indexation

scale : rapport d'indexation

Par exemple :

```
   8d 74 0a ff       lea    0xffffffff(%edx,%ecx,1),%esi

   +---+---+---+   mod: 1 -> [M + sbyte]
   |mod| R | M |   R  : ESI
74:| 01|010|110|   M  : ESP
   +---+---+---+
   +---+---+---+   mod: 00 -> [M + sbyte]
   |mod| R | M |   R  : ECX
0a:| 00|001|010|   M  : EDX
   +---+---+---+
```

Les Offset Le type d'opérande `ASM_OTYPE_OFFSET` ne suit que les opcode 0xa0 à 0xa3. Elle spécifie l'adresse de la valeur à charger dans l'accumulateur.

Par exemple :

```
mov 0x8054e80, %eax
```

Cette instruction met dans a savoir 0x8054e80.

Les adresses Les opérandes de type `ASM_OTYPE_ADDRESS` sont utilisées par les instructions de branchement qui effectuent un saut à l'adresse spécifiée directement après l'opcode. Par exemple :

```
9a | 67 45 23 01 call 0x01234567
```

Cette instruction effectue un appel à la procédure située à l'adresse spécifiée dans l'opérande.

Les offset relatifs de branchement Le type d'opérande `ASM_OTYPE_JUMP` est une valeur immédiate qui sera ajoutée à `EIP` afin d'effectuer un branchement. Il suit les instructions `CALL` et `JMP` ainsi que toutes les instructions de branchement conditionnel.

```
80498f4: e9 | 03 fc ff ff   jmp    0x80494fc
8049910: e8 | e7 fb ff ff   call   0x80494fc
8049922: 74 | 42            jz 0x8049964
```

Les registres de segment, contrôle, debug Pour chacun des types `ASM_OTYPE_CONTROL`, `ASM_OTYPE_DEBUG`, `ASM_OTYPE_SEGMENT`, le champ `R` de l'octet `mod R/M` suivant l'opcode de l'instruction désigne un registre de contrôle, de debug ou de segment. Par exemple :

```
    0f 22 c0     mov    %eax,%cr0

    +---+---+---+   mod: 1 -> [M + sbyte]
    |mod| R | M |   R  : CR0
 c0:| 11|000|000|   M  : EAX
    +---+---+---+
```

Opérande adressée par `ds :esi` ou `es :edi` Les opérandes de type `ASM_OTYPE_XSRC` ou `ASM_OTYPE_YDEST` ne sont utilisées que par les instructions de traitement de chaine : `MOVSB`, `LODSB`, `CMPSB`, `SCASB`, ...Elles sont directement integrées à l'instruction et ne peuvent être modifiées.

Il existe d'autres types d'opérandes notamment pour les instructions MMX et les operandes du FPU mais celles-ci ne sont actuellement pas prises en compte dans l'analyse de code de *libasm*.

Voici la structure sur laquelle est codée une opérande dans *libasm*. Chaque instruction contient jusqu'à 3 opérandes, chacune de type different :

```
struct s_asm_op
{
  /* operand lenght (expressed in R byte from ModRM byte have a null size) */
 u_int         len;

  /* pointer to operand in buffer */
 u_char        *ptr;

  /* operand type : may contain flag listed below */
 u_int         type;
 u_int         size;
 u_int         content;

  /* register set: 8/16/32 bits general registers, segment registers */
```

```
  int           regset;
  int           prefix;

   /* Libasm metadata information */
  asm_processor *proc;

   /* Immediate value extracted from operand */
  int           imm;

   /* Base register */
  int           base_reg;

   /* Index register */
  int           index_reg;

   /* Scale factor */
  unsigned int  scale;
};
```

5.2 Typage des instructions

Afin d'automatiser l'analyse de code, un typage des instructions a été mis en place. L'utilisation d'un typage d'instruction pour l'analyse permet d'effectuer une analyse á un niveau d'abstraction plus élevé que celui de lássembleur machine et de rendre ainsi l'analyse aisemment portable ´a différentes architectures.

Les objectifs visés sont l'obtention d'une représentation manipulable automatiquement des flux de contrôle et de données. Ceci nécessite d'une part un découpage en blocs logiques de l'ensemble du code à analyser. Par ailleurs, afin de réaliser une différenciation des opérandes utilisées en tant que pointeurs des opérandes utilis'ées en tant que valeurs, ou de déterminer certains appels de procédures indirects -utilisation de pointeurs sur fonctions-, il est nécessaire de conserver une trace des transactions effectuées avec la mémoire

Cette représentation symbolique des flux de contróle et de données débouche sur une forme de méta-assembleur indépendante de l'architecture.

5.2.1 Modificateur du flux de contrôle

Ce sont les instructions qui interrompent le flux d'exécution. Il y en a plusieurs types qui interagissent différemment avec le flux de données.

`ASM_TYPE_IMPBRANCH` : L'instruction interrompt le flux de contrôle pour effectuer un branchement direct.

`ASM_TYPE_CONDBRANCH` : L'instruction peut interrompre le flux de contrôle pour effectuer un branchement en fonction du résultat des instructions précédentes. Par exemple sur l'architecture IA32, le registre EFLAGS permet de conserver des informations sur le résultat de l'instruction précédente. Ce sont les bits de ce registre que les instructions de branchement conditionnel vont tester afin de prendre une décision de branchement. Le tableau qui suit liste une partie des instructions de branchement de l'architecture IA32 avec les conditons nécessaires.

Instruction		Jump if	Condition
`JO`	$\longrightarrow$	Overflow	`OF = 1`
`JNO`	$\longrightarrow$	Not Overflow	`OF = 0`
`JE/JZ`	$\longrightarrow$	Equal	`ZF = 1`
`JNE/JNZ`	$\longrightarrow$	Not Equal	`ZF = 0`
`JA`	$\longrightarrow$	Above	`CF = 0 AND ZF = 0`
`JNA`	$\longrightarrow$	Not Above	`CF = 1 OR ZF = 1`
`JB`	$\longrightarrow$	Below	`CF = 1`
`JNB`	$\longrightarrow$	Not Below	`CF = 0`
`JBE`	$\longrightarrow$	Below or Equal	`CF = 1 OR ZF = 1`
`JS`	$\longrightarrow$	Signed	`SF = 1`
`JNS`	$\longrightarrow$	Not Signed	`SF = 0`
`JL`	$\longrightarrow$	Lower	`SF <> OF`
`JLE`	$\longrightarrow$	Lower or Equal	`ZF = 1 AND SF <> OF`
`JGE`	$\longrightarrow$	Greater or Equal	`SF = OF`
`JG`	$\longrightarrow$	Greater	`ZF = 0 AND SF = OF`
`JPE`	$\longrightarrow$	Parity is Even	`PF = 1`
`JPO`	$\longrightarrow$	Parity is Odd	`PF = 0`
`JCZX`	$\longrightarrow$	ECX = 0	`ECX = 0`
`LOOP`	$\longrightarrow$	Counter not null	`ECX <> 0`
`LOOPE`	$\longrightarrow$	" " and Equal	`ECX <> 0 AND ZF = 1`
`LOOPNE`	$\longrightarrow$	" " and Not Equal	`ECX <> 0 AND ZF = 0`

`ASM_TYPE_CALLPROC` : L'instruction interrompt le flux de controle, sauvegarde l'adresse de la prochaine instruction a exécuter apres le retour de la procédure et appelle une sous procédure.

`ASM_TYPE_RETPROC` : L'instruction interrompt le flux de contrôle, et restaure dans le registre `EIP` l'adresse de la prochaine instruction à exécuter. Celle-ci est obtenue en dépilant la valeur immédiatement acessible ùr la pile.

5.2.2 Accès à la mémoire

Il est nécessaire de tracer les différents accès à la mémoire afin détablir une représentation des flux de données. Cette représentation se heurte à plusieurs problèmes notamment celui de l'*aliasing*. En effet, plusieurs accès au même endroit de la mémoire peuvent être référencés de différentes manières :

```
push %ebp
mov %esp, %ebp
mov 0x8(%ebp), %eax
mov $0, (%eax)
mov 0x4(%esp), %ebx
mov $42, %(ebx)
```

Dans la séquence d'instruction précédante, les 2 `mov` affectent une valeur à la meme adresse. En effet, `0x8(%ebp)` et `0x4(%esp)` référencent le premier paramètre de la fonction mais ne sont pas accédées symboliquement de la même manière.

`ASM_TYPE_MEM_READ` : Instruction d'accès en lecture à la mémoire.

`ASM_TYPE_MEM_WRITE` : Instruction d'accès en ecriture à la mémoire.

5.2.3 Opérations arithmétiques

`ASM_TYPE_ARITH` : Ces instructions sont présentes sur toutes les architectures et permettent de réaliser des opérations arithmétiques de base : additions, soustractions, ...

`ASM_TYPE_ARCH` : Ces instructions dépendent de l'architecture et jouent un rôle particulier lors de l'exécution du binaire.

Par exemple : `INT` est une instruction provoquant une interruption dont le code est dans le système d'exploitation. Sous Linux et BSD, l'interruption 0x80 est l'interruption qui permet d'appeler les différentes fonctions du système : ouverture de fichier, lecture, écriture, exécution de binaire, etc.

La combinaison des différents flags permet d'évaluer dans quelle mesure l'instruction altère le flux de données dans les accumulateurs et la mémoire, et permet d'offrir une analyse plus globale des différentes instructions désassemblées.

5.3 Axes de recherche sur l'analyse des flux

L'analyse de flux est une tâche complexe, qui consiste a formaliser a la fois le flux d'execution (le transfert de contrôle entre les blocks atomiques de code) et les transferts de données (l'intéraction entre les variables du programme) . Nous allons détailler en quoi cette étape est nécessaire a l'analyse d'un programme, tant pour effectuer des prédictions de branchement, une protection ASLR (voir chapitre ASLR), ou tout simplement pour automatiser une partie de l'audit de sécurité. Nous insisterons sur l'arithmétique symbolique, le *forward tracing* (analyse de flux linéaire), le *backward tracing* (analyse de flux inverse). Enfin, nous proposons des algorithmes et une implémentation pour certaines de ces problématiques.

Nous allons travailler sur la fonction `func` suivante. Elle est extraite de `disasm_func.c` fournit en annexe B.

```
int func(int fd, char *str, int len)
{
  void (*ptr)();
  char buffer[256];
  int addr;

  addr = strtoul(str, 0, 16);
  ptr = extract_func(str, len);
  if (!addr)
    ptr = (void (*)()) 0;
  else
    snprintf(buffer, len, "%s", str);
  if (!ptr)
    return (0);
  ptr();
}
```

5.3.1 Le flux de contrôle : découpage des blocs

On désigne par bloc une entitée désignant un ensemble d'instructions pour lesquelles le flux d'exécution n'est pas interrompu.

Le découpage en bloc s'effectue grace à plusieurs vecteurs, le parcours linéaire des instructions d'une part, la résolution des adresses des instructions de branchement d'autre part.

Notre analyse portera sur plusieurs aspects sur flux d'exécution dont le Algorithme de decoupage en bloc :

Parcours linéaire des instructions

L'analyse linéaire du flux de contrôle est réalisée grâce aux flags présents dans le type d'instruction. En fonction du type d'instruction, si celle-ci interrompt le flux d'instruction, elle est alors considérée comme séparateur entre deux blocs d'instructions.

```
Decoupage_Bloc
[
  . Le bloc courant n'est pas defini (current_block = NULL)
  . L'adresse du bloc precedent est mis a 0.
```

```
Pour chaque instruction:
[
  Si le bloc courant n'est pas defini:
  [
    S'il n'existe pas de bloc debutant a l'adresse courante:
    [
      - Creer un nouveau bloc courant de taille nulle ayant pour adresse
le pointeur d'instruction courant.
      Si l'adresse de l'instruction precedente est non nulle
      [
        - Rajouter au bloc courant l'adresse de l'instruction
          appelante.
      ]
    ]
    Sinon
    [
      - Initialiser la taille du bloc courant a 0
    ]
  ]

  Sinon
  [
    S'il existe un bloc a l'adresse du bloc courant
    [
      - Le bloc precedent est de type Cont
      - Son adresse contigue est definie a l adresse courante.
      - L'instruction courante est la premiere d'un nouveau
      - Initialiser la taille du bloc a 0
    ]
  ]

  Initialisation de L'adresse du bloc precedant a l'adresse du bloc courrant.
  Ajout de la taille de l'instruction courrante a la taille du bloc courrant.

  Si le bloc courrant n'appartient pas a la liste des blocs
  [
    - l'inserer dans la liste de blocs
  ]
  Si l'instruction est de type ASM_TYPE_CONDBRANCH
  [
    - Definir le type du block a CALLER_JUMP
    - Le champ contig du bloc courrant est initialise a l'adresse courrante
      + la longueur de l'instruction courrante.
    - Le champ altern du bloc courrant est initialise a l'adresse resolue.
    - Le bloc courrant est fini
```

```
    ]
    ou Si l'instruction est de type ASM_TYPE_IMPBRANCH
    [
      - Definir le type du block a CALLER_JUMP
      - Le champ contig du bloc courrant est initalise a 0
      - Le champ altern du bloc courrant est initialise a l'adresse resolue.
      - Le bloc courrant est fini
      - initialiser l'adresse de l'instruction precedente a 0
    ]
    ou Si l'instruction est de type ASM_TYPE_CALLPROC
    [
      - Definir le type du block a CALLER_CALL
      - Le champ contig du bloc courrant est initialise a l'adresse courrante
        a laquelle est ajoutee la longueur de l'instruction courrante.
      - Le champ altern du bloc courrant est initialise a l'adresse resolue.

      - Le bloc courrant est fini

    ]
    ou Si l'instruction est de type ASM_TYPE_RETPROC
    [
      - Definir le type du block a CALLER_RET
      - Le champ contig du bloc courrant est initalise a 0
      - Le champ altern du bloc courrant est initialise a 0

      - Le bloc courrant est fini
      - initialiser l'adresse de l'instruction precedente a 0
      ]
  ]
]
```

L'utilisation de cet algorithme à lui seul ne permet pas d'identifier tous les blocs. En effet, certains blocs parcourus linéairement peuvent être interrompus par des instructions de branchement.

Voici une séquence d'instructions pour laquelle le parcours linéaire des instructions ne permet pas de découper exhaustivement l'ensemble des blocs :

```
 080483DA func + 60  V   jne           <func + 71>
- bloc func_62 ----------------------------------------------
 080483DC func + 62  |   mov           $0,FFFFFFF4(%ebp)
 080483E3 func + 69  V   jmp           <func + 97>
- bloc func_71 ----------------------------------------------
 080483E5 func + 71  |   push          C(%ebp)
 [...]               |
 080483F7 func + 89  V   call          <snprintf>
- bloc func_94 ----------------------------------------------
```

```
 080483FC func + 94  |   add           $10,%esp
- bloc func_97 ------|---[ debut de bloc non detecte ]----
 080483FF func + 97  |   cmp           $0,FFFFFFF4(%ebp)
 08048403 func + 101 V   jne           <func + 115>
- bloc func_103 ----------------------------------------
 08048405 func + 103 |   mov           $0,FFFFFEE0(%ebp)
[...]                V
```

Chaque instruction de branchement est bien utilisé pour séparer un bloc d'un autre ; cependant, l'instruction de branchement située à l'adresse 080483E3 qui charge 080483FF dans `EIP` interrompt un autre bloc qui démarre a l'adresse 080483FC et se finit à l'adresse 08048404. C'est ici qu'intervient l'utilisation du second algorithme.

Résolution des adresses statiques.

Pour chaque instruction de branchement rencontrée, si l'adresse de branchement peut être obtenue (opérande statique), nous allons rechercher la présence d'un bloc à cette adresse.

Si aucun bloc n'est présent, un bloc de type 'inconnu' [Unkn] est créé et il est mis dans la liste des blocs triée.

Sinon, si l'instruction de branchement interrompt un bloc d'instruction, ce bloc est tronqué, et un nouveau bloc est créé ayant pour adresse l'adresse du branchement précédamment résolue.

```
Resolution_Adresse
[
  Pour chaque instruction de branchement
  [
    Si l'operande de destination n'est pas une reference (ASM_TYPE_REFERENCE)
    [
      - Lire le contenu de l'operande.
      - Calculer l'adresse de branchement.

      Si l'adresse de branchement n'appartient pas a un bloc
      [
        -  Creer un nouveau bloc de type Unkn ayant pour adresse l'adresse de
           branchement.
        -  on definit le nouveau bloc comme bloc courrant.
      ]
      Sinon
      [
        - On definit le bloc trouve comme bloc courrant
```

```
      ]

      Si l'adresse de branchement n'est pas l'adresse du debut du bloc
      [
          - Creer un nouveau bloc
          - ce nouveau bloc herite des champs altern, type et contig
            du bloc courrant.
          - la taille du nouveau bloc =
            adresse de branchement - adresse du bloc courrant
          - le champ contig du bloc courrant est initialise a l'adresse
            du nouveau bloc
          - le champ altern du bloc courrant est initialise a 0
          - la taille du bloc courrant est reduite de la taille du
            nouveau bloc.
          - le nouveau bloc est defini en tant que bloc courrant
      ]

        - Ajouter au bloc courrant l'adresse de l'instruction
          courrante qui lui transmet le flux d'execution.
    ]
    Sinon
    [
      - L'adresse de branchement est definie a -1
    ]
  ]
  Retourne adresse de branchement.
]
```

Ces deux algorithmes nous permettent de découper une fonction afin d'y trouver les différents chemins déxécutions qui peuvent être empruntés lors de léxécution du programme.

Voici la fonction `func` désassemblée et découpée à láide des deux algorithmes précédants :

```
- bloc func_0 ------------------------------------------------
 0804839E func + 0       push          %ebp
 0804839F func + 1       mov           %esp,%ebp
 080483A1 func + 3       sub           $128,%esp
 080483A7 func + 9       sub           $4,%esp
 080483AA func + 12      push          $10
 080483AC func + 14      push          $0
 080483AE func + 16      push          C(%ebp)
 080483B1 func + 19      call          <strtoul>
- bloc func_24 -----------------------------------------------
```

```
 080483B6 func + 24      add           $10,%esp
 080483B9 func + 27      mov           %eax,FFFFFEE4(%ebp)
 080483BF func + 33      sub           $8,%esp
 080483C2 func + 36      push          10(%ebp)
 080483C5 func + 39      push          C(%ebp)
 080483C8 func + 42      call          <extract_func>
- bloc func_47 ------------------------------------------
 080483CD func + 47      add           $10,%esp
 080483D0 func + 50      mov           %eax,FFFFFFF4(%ebp)
 080483D3 func + 53      cmp           $0,FFFFFEE4(%ebp)
 080483DA func + 60      jne           <func + 71>
- bloc func_62 ------------------------------------------
 080483DC func + 62      mov           $0,FFFFFFF4(%ebp)
 080483E3 func + 69      jmp           <func + 97>
- bloc func_71 ------------------------------------------
 080483E5 func + 71      push          C(%ebp)
 080483E8 func + 74      push          $<_IO_stdin_used + 4>
 080483ED func + 79      push          10(%ebp)
 080483F0 func + 82      lea           FFFFFEE8(%ebp),%eax
 080483F6 func + 88      push          %eax
 080483F7 func + 89      call          <snprintf>
- bloc func_94 ------------------------------------------
 080483FC func + 94      add           $10,%esp
- bloc func_97 ------------------------------------------
 080483FF func + 97      cmp           $0,FFFFFFF4(%ebp)
 08048403 func + 101     jne           <func + 115>
- bloc func_103 -----------------------------------------
 08048405 func + 103     mov           $0,FFFFFEE0(%ebp)
 0804840F func + 113     jmp           <func + 120>
- bloc func_115 -----------------------------------------
 08048411 func + 115     mov           FFFFFFF4(%ebp),%eax
 08048414 func + 118     call          *%eax
- bloc func_120 -----------------------------------------
 08048416 func + 120     mov           FFFFFEE0(%ebp),%eax
 0804841C func + 126     leave
 0804841D func + 127     ret
```

Le bloc `func_0` se termine par une instruction d'appel de procédure (`CALL`). Si la procédure appelée s'interrompt correctement par une instruction `RET`, alors le flux d'exécution reprendra son cours au bloc `func_24`.

Le bloc `func_24` se termine lui aussi par une instruction d'appel de procédure ; Le flux d'exécution reprendra éventuellement son court au bloc `func_47` si la procédure appelée s'interrompt au moyen de l'instruction `RET`.

Le bloc `func_47` se termine par une instruction de branchement conditionnel

(JNE). Si la condition est vraie, le flux d'exécution est transmis au bloc func_71 sinon au bloc func_62.

Le bloc func_62 se termine par une se termine par une instruction de branchement (JMP). Il se poursuit au bloc func_97.

Le bloc func_71 se termine par une instruction arithmétique (ADD) Il est interrompu par une instruction d'appel de procédure. Il se poursuit au bloc func_97.

Le bloc func_94 se termine par une instruction normale. En effet, une instruction de branchement transmet le flux d'exécution à l'instruction suivante.

Le bloc func_97 se termine par une instruction de branchement conditionnel (JNE). Il se poursuit au bloc func_103 ou func_115

Le bloc func_103 se termine par une instruction de branchement (JMP) il se poursuit au bloc func_120.

Le bloc func_115 se termine par une instruction d'appel de procedure (CALL). Si cette procédure effectue un RET
, alors le flux d'exécution est transmis au bloc func_120

Le bloc func_120 se termine par une instruction de fin de procedure (RET)

Il est possible de reconstruire un graphe simple de la procédure analysée à partir de ces informations.

Implémentation

L'analyse du flux d'exécution a été implémenté sous la forme d'un module *modflow* dans *ELFsh*. Ce module découpe une partie de code en fonction des deux algorithmes précédant afin de conserver par la suite ces informations dans le binaire.

A chaque bloc est associé un symbole de type *STT_BLOCK* permettant de retrouver ce bloc dans la section *ċontrol* qui se prśente sous la forme d'une liste de blocs, éventuellement complétes par un nombre variable de références.

Par ailleurs, les symboles des blocs sont nommés en fonction de leur type : si ceux ci ont été invoqués par une instruction dáppel de procédure (call), ils sont alors préfixés par "function_", sinon par "block_" suivi de l'adresse de début de bloc.

```
typedef struct s_elfshblock
{
```

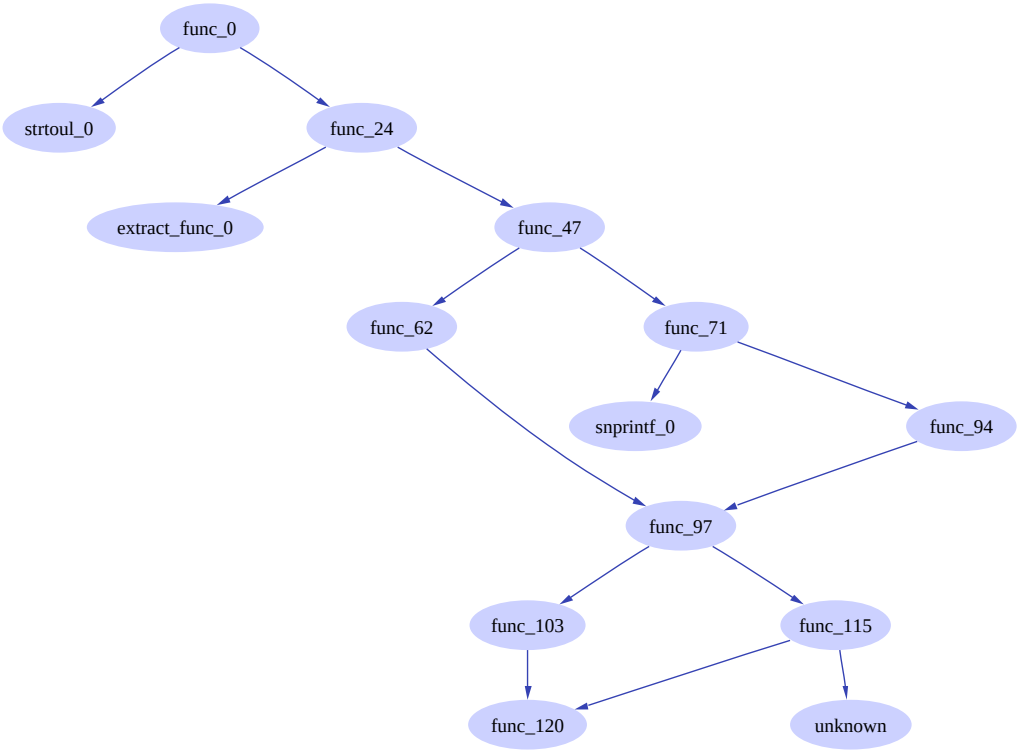
```
  u_int vaddr;          /* virtual address of block     */
  u_int size;           /* size of block                */
  u_int contig;         /* contiguous execution path    */
  u_int altern;         /* alternate execution path     */
  u_int altype;         /* execution breaking instr.    */
} elfshblk_t;

typedef struct s_elfshblref
{
  u_int vaddr;          /* address of instruction invoking block        */
  u_int type;           /* type of invokation                           */
} elfshblkref_t;


         Welcome to The ELF shell 0.5b9 .::.

         .::. This software is under the General Public License
         .::. Please visit http://www.gnu.org to know about Free Software


[ELFsh-0.5b9]$ load disass_func


 [*] New object disass_func loaded on Sun Jun  8 10:05:15 2003


[ELFsh-0.5b9]$ modload modflow

 [*] ELFsh modflow(Jun  8 2003/03:21:05) fini -OK-

     Added commands:
        inspect <vaddr> : inspect block at vaddr
        flow <symbol>   : build a .control section

[ELFsh-0.5b9]$ modload modgraph

 [*] ELFsh modgraph loaded

     Added commands:
        graph <file> : dump graphviz file to file

[ELFsh-0.5b9]$ flow func

[MODFLOW] loading code... vaddr = 0804839e foffset = 926 len = 128
 [*] Entry point: 080482e4
 [*] starting disassembly
```

```
[MODFLOW] done

[ELFsh-0.5b9]$ sym BLOCK

 [SYMBOL TABLE]
 [Object func.blocks]

 [066] (nil)      BLOCK    function_080482b4 sz:000028 foff:000000 scop:Local
 [067] 0x1c       BLOCK    function_080482d4 sz:000028 foff:001456 scop:Local
 [068] 0x38       BLOCK    function_08048394 sz:000028 foff:001484 scop:Local
 [069] 0x54       BLOCK    block_0804839e  sz:000020 foff:001512 scop:Local
 [070] 0x68       BLOCK    block_080483b6  sz:000028 foff:001532 scop:Local
 [071] 0x84       BLOCK    block_080483cd  sz:000028 foff:001560 scop:Local
 [072] 0xa0       BLOCK    block_080483dc  sz:000028 foff:001588 scop:Local
 [073] 0xbc       BLOCK    block_080483e5  sz:000028 foff:001616 scop:Local
 [074] 0xd8       BLOCK    block_080483fc  sz:000028 foff:003177 scop:Local
 [075] 0xf4       BLOCK    block_080483ff  sz:000028 foff:003205 scop:Local
 [076] 0x110      BLOCK    block_08048405  sz:000028 foff:003233 scop:Local
 [077] 0x12c      BLOCK    block_08048411  sz:000028 foff:003261 scop:Local
 [078] 0x148      BLOCK    block_08048416  sz:000028 foff:003289 scop:Local

[ELFsh-0.5b9]$ graph func.graph

 [*] Graph description dumped in func.graph
```

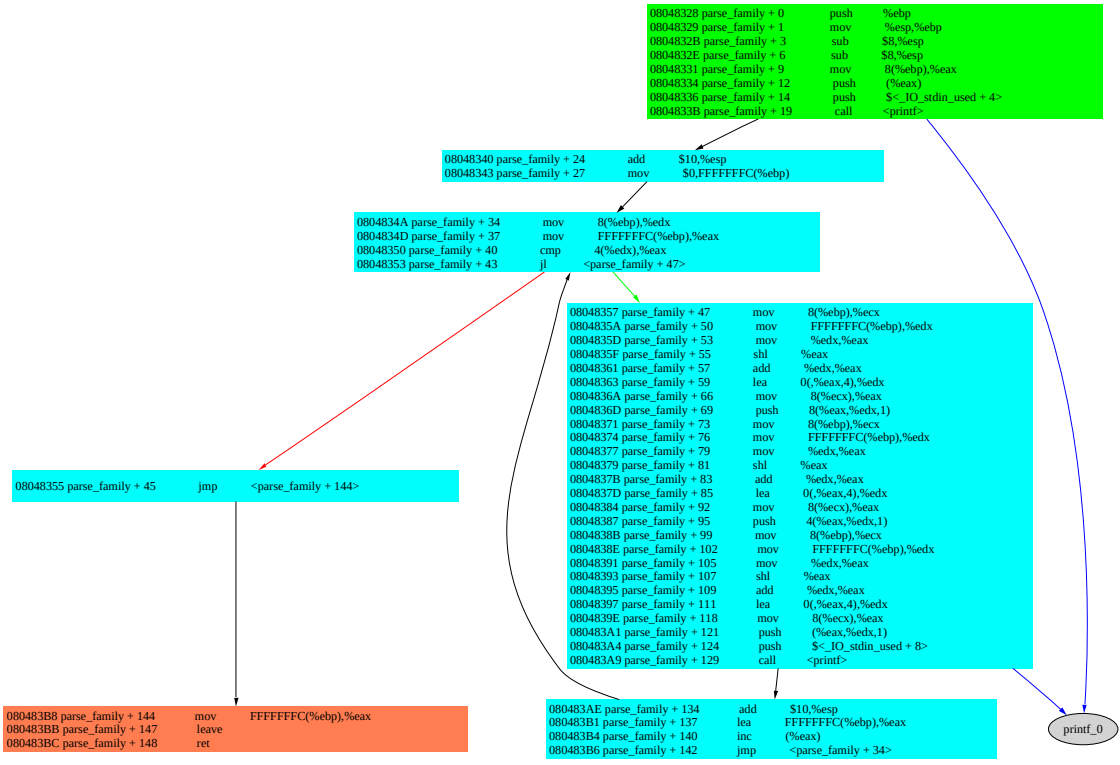
Gràce à l'outil *graphviz*, nous obtenons un rendu de graphe visuel permettant d'observer facilement les différents chemins d'exécution.

```
$ dot -Tps func.graph > func.ps
```

Voici le graphe obtenu à partir de l'analyse de la fonction `func()` :



Si nous discriminons le premier et le dernier bloc de la fonction (en couleur) et que nous incluons le code de chaque bloc, nous commencons a distinguer la structure du programme etudié. Voyons cela sur une petite fonction :



Une fois cette analyse du flux d'execution menée à bien, la seconde étape va consister à analyser les flux de données. L'analyse du flux de données peut éventuellement nous permettre de prédire le flux d'exécution d'une part pour les branchements conditionnels, d'autre part pour les instructions de branchement utilisant comme adresse une donnée dynamique -pointeur sur fonction-.

5.3.2 Prédiction de branchement

Il n'est pas toujours immédiat de retrouver l'adresse de branchement d'une instruction *call* ou *jmp*. En effet, ces instructions peuvent etre directes (dans ce cas l'opérante seule permet de determiner l'adresse de branchement) ou indirectes (l'adresse de branchement se trouve en memoire ou dans un registre, il faut donc connaitre la valeur de la memoire ou du registre a ce moment precis de l'execution du programme).

Reprenons le bloc de la procédure func à l'offset 155 :

```
- bloc func_115 ------------------------------------------
 08048411 func + 115     mov           FFFFFFF4(%ebp),%eax
 08048414 func + 118     call          *%eax
- bloc func_120 ------------------------------------------
```

Nous voyons que le contenu d'une variable locale indexée par le registre EBP est copié dans le registre EAX, puis cette valeur est utilisée comme adresse de branchement absolue par l'instruction *call*. Pour déterminer la destination de ce type de branchement, il faut tracer l'utilisation du registre de maniére inverse, nous désignons cette technique par *backward tracing*, puisqu'elle consiste a remonter le flux d'exécution pour connaître la valeur d'une variable (registre ou memoire) au moment du branchement.

Cet example illustre les liens entre le flux de contrôle et les flux de donnés. Il n'est pas toujours possible de prédire les adresses de branchements de manière statique quand celles ci sont determinee dynamiquement. Par exemple, une instruction JMP indirecte de la section *.plt* ne pourra pas révéler son adresse de branchement automatiquement, puisqu'elle dépend de la valeur de l'entrée correspondante de la section *.got* au moment du branchement (voir chapître précédent).

Il existe des methodes dynamiques pour resoudre ce probleme. Le projet Dynamorio [Kir03] introduit l'une d'elle en déterminant ces adresses par utilisation de points d'arrêt (breakpoint) sur chaque instruction de branchement indirect. Lorsque le point d'arrêt est atteint, l'instruction de branchement est executée pas a pas, l'adresse de branchement est alors déterminée par la nouvelle valeur du registre *EIP*, le pointeur d'instruction.

5.3.3 Flux de données :

L'étude des flux de données constituent notre recherche courante et nous sommes à ce sujet au stade de developpement des algorithmes. Plutot que de soumettre des algorithmes trop spécifiques à la portée trop limitée, nous allons introduire les fonctionnalités nécéssaires à l'étude des flux de données en dégageant plusieurs problématiques.

Tout d'abord, nous voulons identifier les variables globales et locales. Alors que les premieres sont accédées directement par leur adresse, ces dernieres (ainsi que les arguments des fonctions) peuvent être accédées de deux manières différentes en fonction des options de compilations.

Référencement des variables locales et des arguments par `EBP` Lorsque un objet est compilé avec le support des cadres de pile, c'est à dire sans l'option `-fomit-frame-pointer`, les variables locales et les arguments d'une fonction

sont référencés par rapport a %ebp.

Chaque fonction accède à ses arguments en utilisant comme référence le registre `EBP`. Il est ansi aisé de retrouver parmi les instructions composant une fonction celles qui utilisent un argument :

```
Pour chaque instruction
[
  Pour chaque op\'erande de l'instruction
   [
    Si cette operande est une reference positive par rapport a EBP
      et que l'offset est un multiple de 4
      [
 operande de la forme (n + 1 * 4)(%ebp)
           la fonction accede au n-ieme arguments
      ]
   ]
]
```

En reprenant l'exemple précédant, aux adresses `080483C2` et `080483C5`, se trouve les instructions qui empilent les paramètres de la fonction `extract_func()` qui sont deux arguments de la fonction `func`.

```
080483C2 func + 36      push          10(%ebp)
080483C5 func + 39      push          C(%ebp)
080483C8 func + 42      call          <extract_func>
```

En annexe B est fourni un code source analysant de manière linéaire une fonction et chacune des opérandes de ses instructions jusqu'au premier `RET` trouvé, afin de détecter le nombre de paramètres de la fonction.

Après compilation et exécution, ce programme détecte que la fonction analysée n'accède qu'à son deuxième et troisième paramètre :

```
$ gcc -o disass_func disass_func.c -lasm
$ ./disass_func func
found argument 2
found argument 3
found argument 2
found argument 2
found argument 3
RET encountered
$
```

Référencement des variables locales et des arguments par `ESP` Lorsque le binaire est compilé avec l'option `-fomit-frame-pointer`, les fonctions sont

compilées sans cadre de pile. Les variables locales ainsi que les arguments de la fonction sont alors référencés par rapport à `ESP`

```
08048528 <func>:
 8048528: 81 ec 10 01 00 00     sub    $0x110,%esp
 804852e: 57                    push   %edi
 804852f: 56                    push   %esi
 8048530: 53                    push   %ebx
 8048531: 8b 9c 24 20 01 00 00  mov    0x120(%esp,1),%ebx
[1] 8048537: c6 03 00                movb   $0x0,(%ebx)
[...]
```

Il est nécessaire d'analyser les instructions modifiant le registre `ESP` afin de garder une trace de son contenu, puisque les références aux variables locales et aux arguments dépendent de sa valeur.

5.3.4 Arithmétique symbolique

L'interprétation en arithmétique symbolique consiste a reprśenter une variable (registre ou mémoire) sous forme composite, et non sous la forme immediate absolue. Il est ainsi bien plus aisé de retrouver les dépendances aux registres de chaque instruction.

Le bloc d'instruction précèdent sous forme arithmétique est le suivant :

```
 D\'ebut de la procedure.

     ESP = ESP - 0x110 | ESP = old_ESP - 0x110
     ESP = ESP - 0x4 | ESP = old_ESP - 0x110 - 0x4
     [ESP] = EDI | [old_ESP - 0x114] = old_EDI
     ESP = ESP - 0x4 | ESP = old_ESP - 0x114 - 0x4
     [ESP] = ESI | [old_ESP - 0x118] = old_ESI
     ESP = ESP - 0x4 | ESP = old_ESP - 0x118 - 0x4
     EBX = [ESP + 0x120] | EBX = [old_ESP - 0x11c + 0x120]
 [1] [EBX] = 0 | [old_ESP + 0x4] = 0
```

On constate ici que l'instruction `movb $0x0, (%ebx)` de destination l'adresse stockée dans le premier argument de la fonction. Si l'utilisateur contrôle cette valeur, il peut alors ecrire un `00` là où il le désire en mémoire.

`[old_ESP + 0x4]` désigne le premier argument de la fonction en cours d'analyse.

Par compilation des instructions sous forme d'expressions symboliques, il devient ainsi possible de connaître l'origine de certaines opérandes et d'évaluer

dans quelle mesure l'utilisateur peut détourner le flux d'exécution.

Si un erreur survenait lors de l'exécution de l'instruction en [1], et que l'utilisateur contrôle le premier argument passé à la fonction, alors il lui est possible de spécifier une adresse ou écrire en mémoire.

5.3.5 Tracage avant (Forward tracing)

Ce type d'analyse consiste à suivre l'utilisation d'une variable (registre ou mémoire) dans le sens de l'execution, et ce pour determiner son type, ainsi que son influence sur le flux d'execution, au cours d'un bloc ou d'une procédure. Dans *ELFsh*, ce type de de tracage est necessaire afin de determiner si les tables de references (de relocation) sont valides, pour effectuer une protection ASLR sur un binaire donne ; nous allons expliciter cette protection dans le chapitre suivant.

Chapitre 6

ASLR

6.1 Problématique

Pour contrer les protections non-executables, de nouvelles techniques comme le return-into-libc ont ete inventées [Des97] . Cette attaque consiste á modifier le pointeur de pile, et le déplacer sur une séquence d'adresses de procédures executables, de manière à simuler des frames de fonctions dont le code est deja présent dans le processus. Pour cela, les adresses des fonctions et de certains de leurs paramètres sont nécessaires, et c'est au moment du 'ret' de la fonction vulnérable que la séquence de fonctions est executée. Ainsi il n'est plus nécessaire d'injecter du code dans le processus, et les protections non-executable sont contournées.

La defense contre cette attaque est simple et efficace, puisqu'elle consiste a génerer un alea sur toutes les adresses bases des librairies en modifiant le comportement de mmap. Cependant, certaines zones du processus restent mappées à des adresses fixes, comme les segments PT_LOAD du binaire base lui meme, et certaines sections de ces segments se sont averées particulierement intéressantes [Ner01] a la génération de séquences de frames. Ainsi les fonctions, plus generalement les groupes d'instructions, dans les sections .plt ou .text peuvent etre utilisés.

Il est complexe de stopper cette attaque, car un grand nombre d'adresses absolues sont codées en dur dans un binaire, par exemple dans son en-tete, ses sections de code (.text, .plt), ses sections de donnees (.got, .data, .rodata), sa section dynamique (.dynamic), sa section de symboles dynamiques (.dynsym), sa section got (.got) et autres sections de constructeurs et destructeurs. De plus, les sections de relocation pour un objet binaire executable de type ET_EXEC ont ete grandement supprimées par l'éditeur de liens, il est toutefois possible de les conserver avec ld par l'option –emit-relocs - mais ce n'est le cas pour les binaires installes par la grande majorite des distributions.

Grace au projet PAX [Tea02] , il existe une protection contre ces attaques. Elle consiste à mapper 2 fois le binaire dans le processus, l'un aux adresses du binaire, dont l'acces va etre filtrer par le mecanisme de page fault, et l'autre à une adresse aléatoire. Quand l'instance du code remappée accede aux segments d'origine par ses instructions d'adressage absolue, la reference est patchee dynamiquement depuis le handler de PF, ainsi l'execution peut se dérouler malgre une chute de la performance. Cette protection n'est pas parfaite [p5903] mais semble très difficile à contourner en pratique par buffer overflow.

Une protection ASLR statique à pour but de patcher l'executable binaire de maniere à le mapper à un autre endroit qu'aux adresses pour lesquelles il a ete compilé. Pour cela, il est necessaire de reconstruire une partie des informations de relocations du binaire, afin de savoir ou les références absolues doivent etre mise a jour lors du déplacement des segments. Une premiere approche naive consiste a lire toutes les sections mappées, et verifier pour chaque mot de la taille du pointeur, si sa valeur correspond à une adresse mappée du binaire. Dans ce cas, nous devons créer une entrée de relocation pour ce pointeur. Meme si cette methode n'est pas fiable, et nous allons voir pourquoi dans les prochains paragraphes, elle a le merite de fonctionner sur /bin/ls :

```
~quiet
~load /bin/ls
~modload ../../modules/modremap.so
~e

[ELF HEADER]
[Object /bin/ls, MAGIC 0x464C457F]

Architecture       :        Intel 80386   ELF Version        :            1
Object type        :  Executable object   SHT strtab index   :           26
Data encoding      :      Little endian   SHT foffset        :        44336
PHT foffset        :                 52   SHT entries number :           28
PHT entry size     :                 32   ELF header size    :           52
Entry point        :          0x8049420   (.text)
{PAX FLAGS = 0x0}
PAX_PAGEEXEC       :           Disabled   PAX_EMULTRAMP      : Not emulated
PAX_MPROTECT       :         Restricted   PAX_RANDMMAP       :   Randomized
PAX_RANDEXEC       :     Not randomized   PAX_SEGMEXEC       :      Enabled


~s
[SECTION HEADER TABLE .::. SHT is not stripped]
[Object /bin/ls]
```

```
[00] (nil)      ---                 foffset(00000000) size(00000244)
[01] 0x80480f4  a-- .interp         foffset(00000244) size(00000019)
[02] 0x8048108  a-- .note.ABI-tag   foffset(00000264) size(00000032)
[03] 0x8048128  a-- .hash           foffset(00000296) size(00000600)
[04] 0x8048380  a-- .dynsym         foffset(00000896) size(00001296)
[05] 0x8048890  a-- .dynstr         foffset(00002192) size(00000875)
[06] 0x8048bfc  a-- .gnu.version    foffset(00003068) size(00000162)
[07] 0x8048ca0  a-- .gnu.version_r  foffset(00003232) size(00000128)
[08] 0x8048d20  a-- .rel.got        foffset(00003360) size(00000016)
[09] 0x8048d30  a-- .rel.bss        foffset(00003376) size(00000040)
[10] 0x8048d58  a-- .rel.plt        foffset(00003416) size(00000560)
[11] 0x8048f88  a-x .init           foffset(00003976) size(00000037)
[12] 0x8048fb0  a-x .plt            foffset(00004016) size(00001136)
[13] 0x8049420  a-x .text           foffset(00005152) size(00024636)
[14] 0x804f45c  a-x .fini           foffset(00029788) size(00000028)
[15] 0x804f480  a-- .rodata         foffset(00029824) size(00012092)
[16] 0x80533bc  aw- .data           foffset(00041916) size(00000188)
[17] 0x8053478  aw- .eh_frame       foffset(00042104) size(00000004)
[18] 0x805347c  aw- .ctors          foffset(00042108) size(00000008)
[19] 0x8053484  aw- .dtors          foffset(00042116) size(00000008)
[20] 0x805348c  aw- .got            foffset(00042124) size(00000300)
[21] 0x80535b8  aw- .dynamic        foffset(00042424) size(00000168)
[22] 0x8053660  -w- .sbss           foffset(00042592) size(00000000)
[23] 0x8053660  aw- .bss            foffset(00042592) size(00000680)
[24] (nil)      --- .comment        foffset(00042592) size(00000988)
[25] (nil)      --- .note           foffset(00043580) size(00000520)
[26] (nil)      --- .shstrtab       foffset(00044100) size(00000235)
[27] (nil)      --- .symtab         foffset(00044335) size(00000000)

~p
[Program header table .::. PHT]
[Object /bin/ls]

[00] 0x08048034 -> 0x080480F4 r-x memsz(000192) foff(000000052) filesz(000192)
[01] 0x080480F4 -> 0x08048107 r-- memsz(000019) foff(000000244) filesz(000019)
[02] 0x08048000 -> 0x080523BC r-x memsz(041916) foff(000000000) filesz(041916)
[03] 0x080533BC -> 0x08053908 rw- memsz(001356) foff(000041916) filesz(000676)
[04] 0x080535B8 -> 0x08053660 rw- memsz(000168) foff(000042424) filesz(000168)
[05] 0x08048108 -> 0x08048128 r-- memsz(000032) foff(000000264) filesz(000032)

~findrel
[*] EXTRA relocs for /bin/ls
[*] Total number of absolute references : 1606
[*] Section                     srcref[0] dstref[0]
[*] Section .interp             srcref[0] dstref[0]
```

```
[*] Section .note.ABI-tag        srcref[0] dstref[0]
[*] Section .hash                srcref[0] dstref[0]
[*] Section .dynsym              srcref[0] dstref[0]
[*] Section .dynstr              srcref[0] dstref[0]
[*] Section .gnu.version         srcref[0] dstref[0]
[*] Section .gnu.version_r       srcref[0] dstref[0]
[*] Section .rel.got             srcref[0] dstref[0]
[*] Section .rel.bss             srcref[0] dstref[0]
[*] Section .rel.plt             srcref[0] dstref[0]
[*] Section .init                srcref[0] dstref[1]
[*] Section .plt                 srcref[72] dstref[6]
[*] Section .text                srcref[705] dstref[741]
[*] Section .fini                srcref[0] dstref[0]
[*] Section .rodata              srcref[812] dstref[101]
[*] Section .data                srcref[17] dstref[1]
[*] Section .eh_frame            srcref[0] dstref[0]
[*] Section .ctors               srcref[0] dstref[2]
[*] Section .dtors               srcref[0] dstref[1]
[*] Section .got                 srcref[0] dstref[72]
[*] Section .dynamic             srcref[0] dstref[0]
[*] Section .sbss                srcref[0] dstref[0]
[*] Section .bss                 srcref[0] dstref[444]
[*] Section .comment             srcref[0] dstref[0]
[*] Section .note                srcref[0] dstref[0]
[*] Section .shstrtab            srcref[0] dstref[0]
[*] Section .symtab              srcref[0] dstref[0]

~remap 0x11223344
[*] Base address adapted to be congruent pagesize
[*] Delta is 091DB000
[*] MODREMAP : Section .sbss wont be relocated
[*] MODREMAP : Section .bss wont be relocated
[*] MODREMAP : Section .symtab wont be relocated
[*] Remapping at base 11223000 -OK-
[*] Total number of modified references : 1878
        PHT relocation : 12
        SHT relocation : 23
        ENT relocation : 1
        RAW relocation : 1842

~e

[ELF HEADER]
[Object /bin/ls, MAGIC 0x464C457F]

Architecture       :        Intel 80386   ELF Version        :             1
```

```
Object type        :  Executable object   SHT strtab index   :           26
Data encoding      :      Little endian   SHT foffset        :        44336
PHT foffset        :                 52   SHT entries number :           28
PHT entries number :                  6   SHT entry size     :           40
PHT entry size     :                 32   ELF header size    :           52
Entry point        :         0x11224420   (.text)
{PAX FLAGS = 0x0}
PAX_PAGEEXEC       :           Disabled   PAX_EMULTRAMP      : Not emulated
PAX_MPROTECT       :         Restricted   PAX_RANDMMAP       :   Randomized
PAX_RANDEXEC       :     Not randomized   PAX_SEGMEXEC       :      Enabled

~s
[SECTION HEADER TABLE .::. SHT is not stripped]
[Object /bin/ls]

[00] (nil)      ---                foffset(00000000) size(00000244)
[01] 0x112230f4 a-- .interp        foffset(00000244) size(00000019)
[02] 0x11223108 a-- .note.ABI-tag  foffset(00000264) size(00000032)
[03] 0x11223128 a-- .hash          foffset(00000296) size(00000600)
[04] 0x11223380 a-- .dynsym        foffset(00000896) size(00001296)
[05] 0x11223890 a-- .dynstr        foffset(00002192) size(00000875)
[06] 0x11223bfc a-- .gnu.version   foffset(00003068) size(00000162)
[07] 0x11223ca0 a-- .gnu.version_r foffset(00003232) size(00000128)
[08] 0x11223d20 a-- .rel.got       foffset(00003360) size(00000016)
[09] 0x11223d30 a-- .rel.bss       foffset(00003376) size(00000040)
[10] 0x11223d58 a-- .rel.plt       foffset(00003416) size(00000560)
[11] 0x11223f88 a-x .init          foffset(00003976) size(00000037)
[12] 0x11223fb0 a-x .plt           foffset(00004016) size(00001136)
[13] 0x11224420 a-x .text          foffset(00005152) size(00024636)
[14] 0x1122a45c a-x .fini          foffset(00029788) size(00000028)
[15] 0x1122a480 a-- .rodata        foffset(00029824) size(00012092)
[16] 0x1122e3bc aw- .data          foffset(00041916) size(00000188)
[17] 0x1122e478 aw- .eh_frame      foffset(00042104) size(00000004)
[18] 0x1122e47c aw- .ctors         foffset(00042108) size(00000008)
[19] 0x1122e484 aw- .dtors         foffset(00042116) size(00000008)
[20] 0x1122e48c aw- .got           foffset(00042124) size(00000300)
[21] 0x1122e5b8 aw- .dynamic       foffset(00042424) size(00000168)
[22] 0x1122e660 -w- .sbss          foffset(00042592) size(00000000)
[23] 0x1122e660 aw- .bss           foffset(00042592) size(00000680)
[24] (nil)      --- .comment       foffset(00042592) size(00000988)
[25] (nil)      --- .note          foffset(00043580) size(00000520)
[26] (nil)      --- .shstrtab      foffset(00044100) size(00000235)
[27] (nil)      --- .symtab        foffset(00044335) size(00000000)

~p
[Program header table .::. PHT]
```

```
[Object /bin/ls]

[00] 0x11223034 r-x memsz(0000000192) foff(0000000052) filesz(0000000192)
[01] 0x112230F4 r-- memsz(0000000019) foff(0000000244) filesz(0000000019)
[02] 0x11223000 r-x memsz(0000041916) foff(0000000000) filesz(0000041916)
[03] 0x1122E3BC rw- memsz(0000001356) foff(0000041916) filesz(0000000676)
[04] 0x1122E5B8 rw- memsz(0000000168) foff(0000042424) filesz(0000000168)
[05] 0x11223108 r-- memsz(0000000032) foff(0000000264) filesz(0000000032)

~save /tmp/ls.remapped
[*] Object /tmp/ls.remapped save successfully

~exec /tmp/ls.remapped

CVS
graph-tests.esh
regression-tests.esh
regression-tests.out
remap_script.esh
remap_script.esh~
remap_script.out
strings_tests.esh
strings_tests.out

[*] Command executed successfully
```

Ce n'est malheureusement pas si facile pour tous les binaires, et les gros executables sont en general mal relogés par cette methode. Nous allons nous pencher sur les problèmes de mauvaises entrées de relocation, aussi appelés faux-positifs (False positives ou FP).

6.2 Faux positifs et heuristiques de detection

Voyons le resultat de la methode naive sur un binaire tel que ssh :

```
$ /tmp/ssh.remapped
/tmp/ssh.remapped: error while loading shared librairies: unexpected PLT reloc
type 0x24
$
```

Le code de reconstruction de table de relocation a mal interpreté certains mots, et les a reloger alors qu'ils n'étaient pas des références absolues. Il est en effet probable que certaines séquences d'octets puissent, par coincidence, représenter un pointeur valide sans en être un, comme ce fut le cas cette fois pour 4

octets dans une des entrée de relocation de la table *.rel.plt* .

Pour se prévenir d'une telle confusion, nous devons associer à chaque type de section sa propre fonction de reconstruction. Ainsi, selon le type (sh_type) de la section, nous utiliserons une fonction différente, avec un comportement approprié :

1. Analyser entrée par entrée dans les sections .rel* et .dynamic
2. Analyser mot par mot dans .got, .ctors, .dtors et .hash
3. Analyser symbole apres symbole dans .symtab et .dynsym
4. Analyser instruction par instruction dans les sections executables
5. Analyser de manière naive pour les autres sections (.data, .rodata)

Pour chaque section de relocation, nous nous contenterons d'examiner le champ r_offset, pour chaque symbole nous utiliserons le champ st_value, et pour les entrées de dynamic le champ d_ptr.

Voyons maintenant si ssh peut etre remappé :

```
$ /tmp/ssh.remapped mayhem@segfault.net
Enter passphrase for key '/home/mayhem/.ssh/id_dsa': Segmentation fault
$
```

L'erreur intervient avant la première frappe au clavier, voyons pourquoi :

```
$ gdb -q /tmp/ssh.remapped --core=core
(no debugging symbols found)...Core was generated by `/tmp/ssh.remapped
[...]
#0  0x11237e0b in EVP_get_digestbyname ()
(gdb) bt
#0  0x11237e0b in EVP_get_digestbyname ()
#1  0x00000000 in ?? ()
(gdb) x/1i $eip
0x11237e0b <EVP_get_digestbyname+71943>:        testb  $0x11,0x25b7df0c
(gdb)
```

L'opérande de cette instruction semble etre un FP, voyons les logs de notre reconstructeur de relocation pour cette adresse virtuelle :

```
$ elfsh -f /tmp/ssh.remapped -D text | grep 11237E0B

11237E0B [foff: 85515] .text + 71915   test $11,25B7DF0C  F6 05 0C DF B7 25 11

$ elfsh -f /tmp/ssh.remapped -findrel | grep text | grep 11237E0
```

```
[1887] From .text + 71918 TO .bss  + 25503 [ 11237E0E -> 1125B7DF ]

$
```

Nous avons creé une entrée de relocation en croyant que les 4 derniers octets de cette instruction formaient un pointeur. On confirme cette erreur manuellement puisqu'on peut voir que le fameux pointeur appartient à deux opérandes distinctes de l'instruction. Nous devons nous assurer que chaque entrée de relocation reconstruite pointe sur une - et une seule - operande lors de l'analyse des sections executables.

Malgre cette verification supplémentaire, il existe toujours certains cas particulier, qui genere des fausses entrées de relocation :

movl $immed, %reg ; put $immed in %reg

Dans cette construction, et dans le cas ou $immed représente une adresse valide du binaire, le comportement par default est de reconstruire une entree de relocation, qui passe le test d'alignement d'opérande. Toutefois, $immed n'est pas toujours une adresse, la valeur peut etre tout simplement une valeur immediate qui, par coincidence, pointe dans l'espace d'adressage.

Afin de detecter ces construction particulieres, il nous faut tracer le flux de données, c'est a dire l'utilisation du registre %reg. Pour cela, le typage des instructions comme présenté dans le chapitre 3 doit etre exhaustif afin de savoir si le registre est utilisé comme une valeur immediate, ou comme un pointeur (par exemple si %reg est utilisé en registre de base).

Dans cette approche de detection de faux-positifs par suivi des variables, il est possible de rencontrer des fonctions dans lesquelles des pointeurs (donc des entrees de relocation) sont stockés, et utilisés par la suite. Ce cas se présente lors de manipulation de structure de données comme des tableaux de pointeurs, liste chainées de pointeurs, ou toute autre structure dynamiques, comme les graphs. Dans ce cas, la difficulté supplémentaire est celle d'analyser l'utilisation des variables aussi abstraites que ces structures, et ce au dela d'une seule procedure.

Les faux négatifs sont egalement un problème, par exemple lors de la construction d'adresse au moment de l'execution. Cette technique est notamment utilisée par les virus, dans le but de cacher leurs références, et ainsi rendre la tâche difficile aux programmes d'analyse automatique. Elle consiste à construire les adresses par des opérations arithmétiques, dans le but de ne pas la coder en dure, et ainsi echapper à la detection. Toutefois, un compilateur ne genere pas ce type de code, il est donc possible de laisser ce cas particulier de coté.

6.3 Remappage ET_EXEC

L'algorithme de remapping d'objets ET_EXEC est indépendant de l'algorithme de relocation, meme s'il en dépend. Apres avoir mis a jour le point d'entrée dans l'en-tete ELF, nous devons parcourir la PHT et la SHT afin de déplacer chaque segment et chaque section mappée dans l'espace d'adressage virtuel.

```
- Calculer la différence entre les anciennes et nouvelles adresses bases
- Ajouter la différence a e_entry dans le header ELF
- Pour chaque segment mappé
[
     - Ajouter la différence a p_vaddr
     - Ajouter la différence a p_paddr
]
- Pour Chaque section mappée
[
      - Ajouter la différence a sh_addr
]
```

Toutefois, cette méthode est limitée par le choix de la nouvelle adresse basse, qui sera maintenant toujours utilisée, a moins que l'on remappe plusieurs fois le meme objet. Pour éviter le multiple remapping, il est intéressant de transformer un objet executable ET_EXEC en une librairie partagée de type ET_DYN. Ainsi nous utiliserons le mécanisme de randomisation dynamique de mmap() pour générer notre aléa au cours de chaque exécution.

6.4 Relinkage ET_EXEC en ET_DYN

Cette fonctionnalité n'est pas implementée car elle est bridée par un tracage de flux insuffisant, présenté dans le paragraphe précédent. Tout comme le remapping ET_EXEC, l'algorithme associé est independant de la reconstruction de relocation, mais il en depend.

Quelles sont les différences majeures entre un objet ET_EXEC et un objet ET_DYN ?

- L'adressage relatif

Afin de transformer un adressage absolue en adressage relatif, il suffit de soustraire l'adresse basse du binaire à chaque référence. Il faut alors synchroniser la table des symboles et la tables des symboles dynamique avec le nouvel

espace d'adressage.

- La relocation au moment du mapping

Contrairement aux executables, les librairies doivent etre reloger une première fois avant d'être executée, notamment en relogeant toutes les références de type relatives :

```
#define R_386_RELATIVE  8               /* Adjust by program base */
#define R_SPARC_RELATIVE 22             /* Adjust by program base */
```

Cela veut dire que notre librairie va comporter une gros nombre d'entrées de relocation supplémentaires (une pour chaque référence que nous avons passée en relative dans le binaire), et donc qu'il faut etendre les sections de relocation. L'extension d'une section .rel ne pose pas de probleme, cela dit on distingue un cas particulier : la *.plt* .

Le format de la *.plt* est différent pour les objets de type ET_DYN, car elle admet une forme PIC, dont l'adressage est entierement basé sur la valeur du registre `EBX` (sous `INTEL`) :

```
0001B160 [foff: 110944] .plt + 0                    push        4(%ebx)
0001B166 [foff: 110950] .plt + 6                    jmp         *8(%ebx)
0001B16C [foff: 110956] .plt + 12                   add         %al,(%eax)
0001B16E [foff: 110958] .plt + 14                   add         %al,(%eax)

0001B170 [foff: 110960] .plt + 16                   jmp         *C(%ebx)
0001B176 [foff: 110966] .plt + 22                   push        $0
0001B17B [foff: 110971] .plt + 27                   jmp         <.plt>

0001B180 [foff: 110976] .plt + 32                   jmp         *10(%ebx)
0001B186 [foff: 110982] .plt + 38                   push        $8
0001B18B [foff: 110987] .plt + 43                   jmp         <.plt>

0001B190 [foff: 110992] .plt + 48                   jmp         *14(%ebx)
0001B196 [foff: 110998] .plt + 54                   push        $10
0001B19B [foff: 111003] .plt + 59                   jmp         <.plt>
```

La première entré est toujours spéciale, comme décrit dans le chapitre 4. Cette spécificité est importante pour la translation ET_EXEC en ET_DYN, car il n'est pas envisageable de conserver le registre (l'adresse du debut de la GOT) durant toute l'execution du processus, ainsi le format de la PLT d'un objet binaire n'est pas executable tel quel si l'executable est relinké en librairie.

La solution consiste à ajouter des entrées de relocation qui pointent dans la PLT du binaire, qui fait plusieurs référence absolues, comme décrit dans le

chapître d'interception de flux de contrôle dynamique. Comme la PLT est une section de code qu'il n'est en principe pas modifiée au cours de l'execution, il faut préciser un paramètre de comportement à l'éditeur de liens dynamique, afin qu'il passe les pages de code en écriture lors de la relocation de la section. Ce comportement est spécifié par une entrée dans la section dynamique, l'entrée DT_TEXTREL :

```
#define DT_TEXTREL      22              /* Reloc might modify .text */
```

L'éditeur de lien dynamique vérifie la valeur de ce paramètre au cours de l'execution si il est amené a modifier une section en lecture seule, effectuer un appel a mprotect(2) dans le but de changer les droits de la page, a ce moment il effectue la relocation, et enfin restaure les anciens droits restreints de la page.

Chapitre 7

Residence

Voyons comment nous pouvons inserer du code et des donnees excedentaires dans un binaire ELF.

7.1 Premiere approche : le code PIC

Nous pouvons choisir la solution simple du code indépendant de sa position (PIC), pour qu'il puisse s'executer n'importe ou dans le processus. Dans ce cas, nous n'avons pas à nous soucier de reloger le code avant de l'executer.

Cette méthode est également utilisée pour les shellcodes, c'est a dire les codes injectés lors d'exploitation de type buffer overflow. Ainsi le code peut s'executer aussi bien sur la pile que dans le tas (heap), sans se soucier de son adresse base.

Toutefois, il devient plus délicat d'utiliser les fonctions de l'objet hôte, ou les fonctions exportées de librairie dans ce type de code. Il est alors nécéssaire de disposer d'un propre moteur de relocation statique, et ainsi reloger certaines parties du code injecté, pour le renseigner des éventuelles adresses absolues dont il a besoin.

Dans ce chapitre, nous expliciterons en quoi cette méthode est naive, et devient trop complexe pour des injections de moyenne ou grosse envergure. Nous proposerons plusieurs méthodes alternes.

7.2 Zones d'alignement

Chaque segment de type PT_LOAD doit etre aligné sur une page sur architecture INTEL (4096 bytes) et sur architecture SPARC (8192 bytes) . Cela laisse de la place pour un parasite qui viendrait se loger dans le padding du

PT_LOAD executable. Comme le padding n'est pas physiquement inséré dans le fichier, une telle résidence aura pour consequence de décaler tous les offsets de fichier pour chaque section se trouvant apres le parasite, mais pas les adresses virtuelles des segments. Voyons les PT_LOAD decrit par la PHT sous SPARC :

```
[02] PT_LOAD 0x10000 r-x memsz:0070100 foff:00000000 filesz:0070100
[03] PT_LOAD 0x311d8 rwx memsz:0002624 foff:00070104 filesz:0001816
```

.. et sous INTEL :

```
[02] PT_LOAD 0x08048000 r-x memsz:041916 foff:000000 filesz:041916
[03] PT_LOAD 0x080533BC rw- memsz:001356 foff:041916 filesz:000676
```

L'éditeur de liens s'assure que l'espace d'adressage du padding est respecté.

Par contre, il n'est pas vraiment envisageable d'utiliser le padding de chacune des sections, la taille maximale du padding etant 3 bytes - car l'addresse d'une section doit etre alignée sur 4 bytes - nous ne pourrions meme pas coder une instruction de branchement tel que le jmp indirect, qui permettrait de passer le contrôle a une partie du parasite placée dans le padding des sections suivantes.

7.3 Ajout de section

Synchroniser la perspective par sections de l'éditeur de liens est interessant dès lors que l'on souhaite relinker des objets entre eux, ou debugger le binaire hôte de manière décente. Nous pouvons injecter les sections de plusieurs facons :

1. Section mappée de code
2. Section mappée de données
3. Section non mappée

Les sections non mappées sont les plus simples a injecter, elles sont souvent utiles pour stocker des informations, comme des tables de références, des tables de symboles excedentaires, ou toutes autres tables qui peuvent faciliter l'analyse par la suite.

```
$ elfsh -f fake_aout -q -s
[SECTION HEADER TABLE .::. SHT is not stripped]
[Object fake_aout]

[00] (nil)      ---                 foffset(00000000) size(00000000)
[01] 0x80480f4  a-- .interp         foffset(00000244) size(00000019)
[...]
[21] 0x804963c  aw- .bss            foffset(00001596) size(00000024)
[22] (nil)      --- .stab           foffset(00001596) size(00002388)
```

```
[23] (nil)      --- .stabstr        foffset(00003984) size(00006498)
[24] (nil)      --- .comment        foffset(00010482) size(00000228)
[25] (nil)      --- .note           foffset(00010710) size(00000120)
[26] (nil)      --- .shstrtab       foffset(00010830) size(00000232)
[27] (nil)      --- .symtab         foffset(00012252) size(00001264)
[28] (nil)      --- .strtab         foffset(00013516) size(00000569)
[29] (nil)      --- .newsect        foffset(00014095) size(00000022) *

$
```

Les sections de données sont en general ajoutées apres la section .bss, qui est la derniere section du segment de données. La *.bss* n'est pas physiquement dans le fichier, il possède seulement une entrée dans la *SHT* entre la dernière section mappée et la premiere section non mappée, ce qui permet a l'éditeur de liens de l'identifier comme la zone des données non initialisées.

```
$ elfsh -q -f fake_aout -s
[SECTION HEADER TABLE .::. SHT is not stripped]
[Object fake_aout]

[00] (nil)      ---                 foffset(00000000) size(00000000)
[01] 0x80480f4  a-- .interp         foffset(00000244) size(00000019)
[...]
[20] 0x8049614  aw- .got            foffset(00001556) size(00000040)
[21] 0x804963c  aw- .bss            foffset(00001596) size(00000024)
[22] 0x8049654  a-x .newsect        foffset(00001620) size(00000030) *
[23] (nil)      --- .stab           foffset(00001650) size(00002388)
[...]
[28] (nil)      --- .symtab         foffset(00012306) size(00001264)
[29] (nil)      --- .strtab         foffset(00013570) size(00000569)

$
```

Les sections de code peuvent etre mappée a divers endroits. Une premiere approche consiste a les mapper dans le padding du segment executable, mais cela limite la taille de notre code injecte. Nous pouvons egalement les injecter en premier, entre la première section (section NULL) et la deuxième (generalement *.interp*). Cette deuxiéme approche est intéressante du fait qu'elle permet d'injecter autant de données que l'on souhaite, en étendant l'espace d'adressage par le haut, sans decaler les adresses virtuelles des sections suivantes.

```
$ elfsh -f fake_aout -q -s
[SECTION HEADER TABLE .::. SHT is not stripped]
[Object fake_aout]

[00] (nil)      ---                 foffset(00000000) size(00000000)
```

```
[01] 0x80470f4  a-x .newsect        foffset(00000244) size(00004096) *
[02] 0x80480f4  a-- .interp         foffset(00004340) size(00000019)
[03] 0x8048108  a-- .note.ABI-tag   foffset(00004360) size(00000032)
[04] 0x8048128  a-- .hash           foffset(00004392) size(00000056)
[05] 0x8048160  a-- .dynsym         foffset(00004448) size(00000144)
[...]
[27] (nil)      --- .shstrtab       foffset(00014910) size(00000231)
[28] (nil)      --- .symtab         foffset(00016332) size(00001264)
[29] (nil)      --- .strtab         foffset(00017596) size(00000569)

$
```

Bien sûr, nous devons synchroniser la perspective du systeme, en modifiant la PHT (particulierement les champs de l'en-tête des segments mappes) afin que les données des sections injectées soient inclues dans le segment approprié. L'algorithme d'insertion de section se resume à :

1. La creation de l'en-tête
2. L'insertion de l'en-tête dans la SHT (présente dans une zone nonmappée)
3. L'insertion des données initiales dans la nouvelle section

A chaque insertion de section, les tables de symboles doivent etre synchronisées, plus precisement le champ d'index de section (st_sctndx) doit etre mis a jour pour chaque entrée de la table des symboles. Ce champ est notamment utilisé pour indiquer que le symbole est non definie (SHN_UNDEF) ou common (SHN_COMMON), autrement il indique la section parente de l'objet référence par le symbole. De meme, l'index de la section des noms de sections doit etre mis a jour dans l'en-tete ELF (e_shstrndx) si la section injectée a été inserée apres .shstrtab.

7.4 Extension de segments

Dans le cas d'une injection de section post-bss, nous devons fixer le segment PT_LOAD writable (rw-) pour qu'il prenne en compte l'insertion physique du bss (en modifiant la taille physique p_filesz par la valeur de la taille memoire p_memsz de l'entree de la *PHT*) et l'insertion de la nouvelle section.

```
Pour Chaque entrée de la PHT
[
       Si le segment est de type PT_LOAD et accessible en ecriture (aw-)
       [
          - Changer p_filesz pour la meme valeur que p_memsz
          - Etendre p_filesz et p_memsz de la taille de la section injectée
```

```
        ]
]
```

Dans le cas d'une injection de section pre-interp, nous devons décrémenter l'adresse base du segment PT_LOAD executable (r-x) pour qu'il prenne en compte l'insertion de la nouvelle section de code au dessus de la section .interp. Cette methode permet d'eviter de reloger l'executable lui meme a chaque insertion, et n'admet pas de limite de taille contrairement a l'injection par padding de segment.

```
Pour Chaque entrée de la PHT
[
      Si le segment est de type PT_LOAD executable (a-x)
       [
          - Ajouter la longueur de la section injectée a p_filesz et p_memsz
          - Soustraire la taille de la section injectée a p_vaddr et p_paddr
       ]
      Sinon si le segment est de type PT_PHDR
       [
          - Soustraire la taille de la section injectée a p_vaddr et p_paddr
       ]
      Sinon
       [
          - Ajouter la taille de la section injectée a p_offset
       ]
]
```

L'avantage ultime de cette méthode, est qu'elle est compatible avec les environnement de type non-executable, qui refusent toute execution de code dans le processus, en dehors des segments exécutables PT_LOAD (dans le but principal de protéger des buffer overflow dont le shellcode s'exécute dans sur la pile ou le tas). Il n'existe pas d'autre méthode pour étendre plus d'une page de code dans ce segment dans un environnement non executable. Si l'environnement n'est pas non-executable, il est possible d'utiliser l'injection post-bss pour insérer du code.

7.5 La section .dynamic

Cette section permet d'indexer toutes les tables nécessaires au linkage dynamique, telles que l'adresse et les tailles des tables de relocation ou de symboles

dynamique. Voici sa representation :

```
$ elfsh -f /bin/ls -d

[*] Object /bin/ls has been loaded (O_RDONLY)

[SHT_DYNAMIC]
[Object /bin/ls]

[00] Name of needed library            =>          librt.so.1 {DT_NEEDED}
[01] Name of needed library            =>           libc.so.6 {DT_NEEDED}
[02] Address of init function          =>          0x08048F88 {DT_INIT}
[03] Address of fini function          =>          0x0804F45C {DT_FINI}
[04] Address of symbol hash table      =>          0x08048128 {DT_HASH}
[05] Address of dynamic string table   =>          0x08048890 {DT_STRTAB}
[06] Address of dynamic symbol table   =>          0x08048380 {DT_SYMTAB}
[07] Size of string table              =>           821 bytes {DT_STRSZ}
[08] Size of symbol table entry        =>            16 bytes {DT_SYMENT}
[09] Debugging entry (unknown)         =>          0x00000000 {DT_DEBUG}
[10] Processor defined value           =>          0x0805348C {DT_PLTGOT}
[11] Size in bytes for .rel.plt        =>           560 bytes {DT_PLTRELSZ}
[12] Type of reloc in PLT              =>                  17 {DT_PLTREL}
[13] Address of .rel.plt               =>          0x08048D58 {DT_JMPREL}
[14] Address of .rel.got section       =>          0x08048D20 {DT_REL}
[15] Total size of .rel section        =>            56 bytes {DT_RELSZ}
[16] Size of a REL entry               =>             8 bytes {DT_RELENT}
[17] SUN needed version table          =>          0x08048CA0 {DT_VERNEED}
[18] SUN needed version number         =>                   2 {DT_VERNEEDNUM}
[19] GNU version VERSYM                =>          0x08048BFC {DT_VERSYM}

[*] Object /bin/ls unloaded

$
```

Nous trouvons depuis cette table les addresses virtuelles des tables de relocation, tables des symboles, tables de hash des symboles. Nous trouvons également des entrées de type DT_NEEDED qui sont interessantes pour la résidence, puisqu'elles indexent les dépendances (librairies : type ELF ET_DYN) du binaire. Nous pouvons donc :

1. Ajouter des entrées de type DT_NEEDED
2. Modifier une entrée existante pour la transformer en DT_NEEDED

L'ajout d'une entrée peut etre problématique. En effet, la section .dynamic se trouve (en general) juste avant le .bss, de maniere generale, entre les sections

de données initialisees (.data, .rodata, etc) et la section des données non initialisées (.bss). Ainsi, il faudra décaler la section .bss, et mettre à jour toutes ses références dans les autres sections (notamment dans .text), ce qui n'est pas toujours trivial, comme explicité dans la partie sur la problématique de l'ASLR.

La modification d'entrée est plus subtile. Chaque entrée est structurée comme telle :

```
typedef struct
{
 Elf32_Sword   d_tag;                  /* Dynamic entry type */
 union
   {
     Elf32_Word d_val;                 /* Integer value */
     Elf32_Addr d_ptr;                 /* Address value */
   } d_un;
} Elf32_Dyn;
```

Le champ d_tag contient le type d'entrée (dans notre cas : DT_NEEDED), le champ val (ou ptr) contient la valeur du pointeur, ou d'index associé a l'entrée. Dans le cas d'une entrée DT_NEEDED, le champ d_val contient un offset en octets depuis le debut de la section .dynstr, ou se trouve la chaine de caracteres du nom de la librairie.

Nous pouvons remarquer que certaines entrées ne sont pas obligatoires pour que l'executable puisse etre lancé, comme par exemple les entrées de type DT_DEBUG (index 9) . Si nous changeons d_tag pour y mettre DT_NEEDED, et que nous changeons d_val pour y mettre l'offset d'une chaine de caractères valide en tant que nom de librairie, nous pouvons donc rajouter une dependance au binaire ELF ET_EXEC sans injection de données supplementaires :

1. Sans changer sa taille
2. Sans relocation

Il faut pour cela trouver un nom de librairie valide pour le systeme, termine par NUL, voyons dans ls :

```
$ elfsh -f /bin/ls -X dynstr | grep so
.dynstr + 16  6C 69 62 72 74 2E 73 6F 2E 31 00 63 6C 6F 63 6B librt.so.1.clock
.dynstr + 48  69 6E 5F 75 73 65 64 00 6C 69 62 63 2E 73 6F 2E in_used.libc.so.
.dynstr + 176 72 6E 61 6C 00 71 73 6F 72 74 00 6D 65 6D 63 70 rnal.qsort.memcp
.dynstr + 784 65 65 00 6D 62 73 69 6E 69 74 00 5F 5F 64 73 6F ee.mbsinit.__dso
$
```

```
$ cat > /tmp/newlib.c
function()
{
printf("my own fonction \n");
}
$ gcc -shared /tmp/newlib.c -o /lib/rt.so.1
$ elfsh

        Welcome to The ELF shell 0.5b5 .::.

        .::. This software is under the General Public License
        .::. Please visit http://www.gnu.org to know about Free Software

[ELFsh-0.5b5]$ load /bin/ls

  [*] New object /bin/ls loaded on Mon Apr 28 23:09:55 2003

[ELFsh-0.5b5]$ d DT_NEEDED|DT_DEBUG

[SHT_DYNAMIC]
[Object /bin/ls]

[00] Name of needed library            =>          librt.so.1 {DT_NEEDED}
[01] Name of needed library            =>           libc.so.6 {DT_NEEDED}
[09] Debugging entry (unknown)         =>          0x00000000 {DT_DEBUG}

[ELFsh-0.5b5]$ set 1.dynamic[9].tag DT_NEEDED

  [*] Field set succesfully

[ELFsh-0.5b5]$ set 1.dynamic[9].val 19

  [*] Field set succesfully

[ELFsh-0.5b5]$ save /tmp/ls.new

  [*] Object /tmp/ls.new saved successfully

[ELFsh-0.5b5]$ quit

  [*] Unloading object 1 (/bin/ls) *

        Good bye ! .::. The ELF shell 0.5b5

$
```

Nous avons créer un nouveau ls, en changeant une entrée facultative DT_DEBUG de la section .dynamic, en une entrée DT_NEEDED. L'executable reste valide, seule une dépendance a ete ajoutée.

```
$ ldd /tmp/ls.new
librt.so.1 => /lib/librt.so.1 (0x40021000)
libc.so.6 => /lib/libc.so.6 (0x40033000)
rt.so.1 => /lib/rt.so.1 (0x40144000)
libpthread.so.0 => /lib/libpthread.so.0 (0x40146000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ ./ls.new
AcroOlAAFj  ELFSH_DEBUG  ls.new  newlib.c
$ elfsh -f ls.new -d DT_NEEDED

[*] Object ls.new has been loaded (O_RDONLY)

[SHT_DYNAMIC]
[Object ls.new]

[00] Name of needed library            =>          librt.so.1 {DT_NEEDED}
[01] Name of needed library            =>           libc.so.6 {DT_NEEDED}
[09] Name of needed library            =>             rt.so.1 {DT_NEEDED}

[*] Object ls.new unloaded

$
```

Il est egalement interessant de permuter les entrés DT_NEEDED, ainsi il est utile de donner la priorité a certaines de nos fonctions de la librarie excendentaire lors de la resolution des symboles. En effet, chaque objet du processus est représenté par un element dans une liste chaînée de structure linkmap par l'éditeur de lien dynamique. Lors de la relocation (notamment lors de la modification de la *.got* au moment de l'execution), cette liste est parcourue et la valeur du premier symbole correspondant trouvé est utilisé. Ainsi, nous pouvons forcer l'utilisation de nos propres symboles, détourner le flux d'execution, et ajouter une dépendence en modifiant seulement 3 octets du fichier.

Certains binaires ne possèdent pas d'entrées facultatives dans .dynamic (DT_DEBUG, ou autres), il n'est donc pas toujours possible d'utiliser la section .dynamic et d'encapsuler le code excedentaire dans une librairie.

De plus, le nombre limité d'entrées facultatives dans la section .dynamic ne permet pas une granularisation des zones résidentes, pour le code et les données excedentaires. Est-il aisé de relinker des modules relogeables dans un objet ET_EXEC non-relogeable ? La réponse est oui.

7.6 Injection ET_REL dans ET_EXEC

Cette méthode est tres efficace et portable. En effet, seule la fonction de relocation est dépendante de l'architecture, mais toutes les primitives de manipulation ELF sont communes à toutes les machines.

Notre algorithme est simple, il s'effectue en 2 passes. La première passe, qui est la plus complexe, et consiste a insérer des copies des sections mappées du module dans l'executable, est la suivante :

```
Pour chaque section du module
[
  Si la section est de type BSS (SHT_NOBITS dans sh_type)
  [
         - Initialiser l'interface interne du BSS si besoin
         - Insérer une nouvelle zone dans la section .bss pour ce module
  ]
  Sinon, si elle est allocatable (SHF_ALLOC dans sh_flags)
  [
      Si elle est writable (SHF_WRITE dans sh_flags) alors:
      [
         - Insérer la section sous le .bss
         - Résoudre et insérer les symboles pointant sur la section
      ]
      Sinon
      [
         - Insérer la section au dessus de .interp
         - Résoudre et insérer les symboles pointant sur la section
      ]
   ]
   Sinon passer a la section suivante.
]
```

La deuxième passe permet de reloger chaque section injectée en utilisant sa table de relocation dans l'objet relogeable ET_REL. Etant donne que l'executable lui meme n'a pas besoin d'etre relogé (puisque ses sections mappees ne sont pas deplacée dans l'espace d'adressage lors de l'insertion d'un de ou plusieurs modules), cette implementation a l'avantage de ne pas générer de faux-positifs de relocation.

Voici l'algorithme de cette deuxieme passe :

```
Pour chaque section du module ET_REL
[
```

```
   Si la section est de type BSS (SHT_NOBITS dans sh_type)
   [
      - Ne pas reloger
   ]
   Sinon, si elle est allocatable (SHF_ALLOC dans sh_flags)
   [
      - Trouver la section injectée correspondante dans l'objet ET_EXEC
      - Trouver la section de relocation correspondante dans l'objet ET_REL
      Si toutes ces sections sont disponibles:
      [
            - Reloger la section injectee a l'aide de :
            * La table de relocation (.rel.*) dans l'objet ET_REL
            * La table des symboles (.symtab) dans l'objet ET_EXEC
            * La table des symboles dynamiques (.dynsym) dans l'objet ET_EXEC
      ]
      Sinon passer a la section suivante.
   ]
   Sinon passer a la section suivante.
]
```

Le fait d'utiliser toutes les tables des symboles permet de relinker a la fois en utilisant les objets du module et les objets du binaire. Ainsi, le code injecté dans le module peut faire appel aux fonctions et utiliser les variables du binaire original.

Voici la sortie d'un script ELFsh ou la commande 'reladd' (insertion d'un module relogeable) est utilisée :

```
~exec cat func.c

int glvar_testreloc = 42;

int func(char *str)
{
  if (!str)
    glvar_testreloc++;
  printf("ET_REL takes control now [%s:%u]\n", str, glvar_testreloc);
  return (0);
}

 [*] Command executed successfully

~exec gcc -c func.c
```

```
 [*] Command executed successfully

~exec cat exec.c

int main()
{
  printf("First_printf\n");
  puts("Second_puts");
  return (0);
}

[*] Command executed successfully

~exec gcc exec.c

[*] Command executed successfully

~exec /tmp/a.out
First_printf
Second_puts

[*] Command executed successfully

~load a.out

[*] New object a.out loaded on Sun Apr  6 00:51:11 2003

s
 [SECTION HEADER TABLE .::. SHT is not stripped]
 [Object a.out]

[000] (nil)      -------                    foffset:00000000 size:00000244
[001] 0x80480f4  a------ .interp            foffset:00000244 size:00000019
[002] 0x8048108  a------ .note.ABI-tag      foffset:00000264 size:00000032
[003] 0x8048128  a------ .hash              foffset:00000296 size:00000052
[004] 0x804815c  a------ .dynsym            foffset:00000348 size:00000128
[005] 0x80481dc  a------ .dynstr            foffset:00000476 size:00000127
[006] 0x804825c  a------ .gnu.version       foffset:00000604 size:00000016
[007] 0x804826c  a------ .gnu.version_r     foffset:00000620 size:00000032
[008] 0x804828c  a------ .rel.dyn           foffset:00000652 size:00000008
[009] 0x8048294  a------ .rel.plt           foffset:00000660 size:00000040
[010] 0x80482bc  a-x---- .init              foffset:00000700 size:00000037
[011] 0x80482e4  a-x---- .plt               foffset:00000740 size:00000096
[012] 0x8048350  a-x---- .text              foffset:00000848 size:00000332
[013] 0x804849c  a-x---- .fini              foffset:00001180 size:00000028
[014] 0x80484b8  a------ .rodata            foffset:00001208 size:00000034
```

```
[015] 0x80494dc  aw----- .data             foffset:00001244 size:00000012
[016] 0x80494e8  aw----- .eh_frame         foffset:00001256 size:00000004
[017] 0x80494ec  aw----- .dynamic          foffset:00001260 size:00000200
[018] 0x80495b4  aw----- .ctors            foffset:00001460 size:00000008
[019] 0x80495bc  aw----- .dtors            foffset:00001468 size:00000008
[020] 0x80495c4  aw----- .got              foffset:00001476 size:00000036
[021] 0x80495e8  aw----- .bss              foffset:00001512 size:00000024
[022] (nil)      ------- .stab             foffset:00001512 size:00001932
[023] (nil)      ------- .stabstr          foffset:00003444 size:00006377
[024] (nil)      ------- .comment          foffset:00009821 size:00000228
[025] (nil)      ------- .note             foffset:00010049 size:00000120
[026] (nil)      ------- .shstrtab         foffset:00010169 size:00000222
[027] (nil)      ------- .symtab           foffset:00011552 size:00001248
[028] (nil)      ------- .strtab           foffset:00012800 size:00000551

~load func.o
 [*] New object func.o loaded on Sun Apr  6 00:51:11 2003

~s
 [SECTION HEADER TABLE .::. SHT is not stripped]
 [Object func.o]

[000] (nil)      -------                   foffset:00000000 size:00000064
[001] (nil)      a-x---- .text             foffset:00000064 size:00000050
[002] (nil)      aw----- .data             foffset:00000116 size:00000004
[003] (nil)      aw----- .bss              foffset:00000120 size:00000000
[004] (nil)      ------- .note             foffset:00000120 size:00000020
[005] (nil)      a------ .rodata           foffset:00000160 size:00000050
[006] (nil)      ------- .comment          foffset:00000210 size:00000038
[007] (nil)      ------- .shstrtab         foffset:00000248 size:00000071
[008] (nil)      ------- .symtab           foffset:00000760 size:00000192
[009] (nil)      ------- .strtab           foffset:00000952 size:00000051
[010] (nil)      ------- .rel.text         foffset:00001004 size:00000032

~relinject 1 2
 [*] ET_REL func.o injected succesfully in ET\_EXEC a.out

~switch 1
 [*] Switched on object 1 (a.out)

~s
 [SECTION HEADER TABLE .::. SHT is not stripped]
 [Object a.out]

[000] (nil)      -------                   foffset:00000000 size:00000244
[001] 0x80460f4  a------ func.o.rodata     foffset:00000244 size:00004096
```

```
[002] 0x80470f4  a-x---- func.o.text        foffset:00004340 size:00004096
[003] 0x80480f4  a------ .interp            foffset:00008436 size:00000019
[004] 0x8048108  a------ .note.ABI-tag      foffset:00008456 size:00000032
[005] 0x8048128  a------ .hash              foffset:00008488 size:00000052
[006] 0x804815c  a------ .dynsym            foffset:00008540 size:00000128
[007] 0x80481dc  a------ .dynstr            foffset:00008668 size:00000127
[008] 0x804825c  a------ .gnu.version       foffset:00008796 size:00000016
[009] 0x804826c  a------ .gnu.version_r     foffset:00008812 size:00000032
[010] 0x804828c  a------ .rel.dyn           foffset:00008844 size:00000008
[011] 0x8048294  a------ .rel.plt           foffset:00008852 size:00000040
[012] 0x80482bc  a-x---- .init              foffset:00008892 size:00000037
[013] 0x80482e4  a-x---- .plt               foffset:00008932 size:00000096
[014] 0x8048350  a-x---- .text              foffset:00009040 size:00000332
[015] 0x804849c  a-x---- .fini              foffset:00009372 size:00000028
[016] 0x80484b8  a------ .rodata            foffset:00009400 size:00000034
[017] 0x80494dc  aw----- .data              foffset:00009436 size:00000012
[018] 0x80494e8  aw----- .eh_frame          foffset:00009448 size:00000004
[019] 0x80494ec  aw----- .dynamic           foffset:00009452 size:00000200
[020] 0x80495b4  aw----- .ctors             foffset:00009652 size:00000008
[021] 0x80495bc  aw----- .dtors             foffset:00009660 size:00000008
[022] 0x80495c4  aw----- .got               foffset:00009668 size:00000036
[023] 0x80495e8  aw----- .bss               foffset:00009704 size:00000024
[024] 0x8049600  aw----- func.o.data        foffset:00009728 size:00000004
[025] (nil)      ------- .stab              foffset:00009732 size:00001932
[026] (nil)      ------- .stabstr           foffset:00011664 size:00006377
[027] (nil)      ------- .comment           foffset:00018041 size:00000228
[028] (nil)      ------- .note              foffset:00018269 size:00000120
[029] (nil)      ------- .shstrtab          foffset:00018389 size:00000260
[030] (nil)      ------- .symtab            foffset:00019892 size:00001248
[031] (nil)      ------- .strtab            foffset:00021140 size:00000551

~got puts
[Global Offset Table .::. GOT]
[Object a.out]

[004] 0x0804830A  <puts@@GLIBC_2.0 + 6>

~set 1.got[4] 1.sht[2].addr
 [*] Field set succesfully

~save /tmp/a.out.injected
 [*] Object /tmp/a.out.injected save successfully

~exec /tmp/a.out.injected
First_printf
ET_REL takes control now [Second_puts:42]
```

```
[*] Command executed successfully
```

Comme vous pouvez le voir, la fonction puts() est detournée vers le module inseré, ce qui permet de conclure que la résidence par injection ET_REL a réussie. Puisque la SHT est synchronisée avec l'image mémoire décrite par la PHT, il est aisé de retirer les modules ainsi inserés, puisque la SHT renseigne de la provenance de chaque section, par son nom. De même, cette implémentation fonctionne sous les systèmes protegés par PAX, puisque toute les injections de code étendent directement la zone executable du binaire, décrite par l'entrée de la PHT dont les droits sont r-x.

Chapitre 8

Le modèle ELFsh

Il est orienté sur deux axes :

1. Le croisement des espaces de noms
2. La virtualisation des composants logiciels et des objets ELF

Les fonctionnalités d'ELFsh peuvent etre appelées depuis le shell, dans une session interractive, par un script shell, ou par un script ELFsh. L'idée d'une interface GUILE [GNU03b] ou PERL a été discutée, mais ne reste qu'au stade de projet pour le moment.

D'un autre cote, chaque composant du shell - notamment les commandes - et chaque structure ELF est virtualisé en objet, qui possède ses propres routines (methodes) et attributs.

8.1 Orienté Objet

Il existe 3 types d'objets ELFsh :

8.1.1 Objet L1

Ce sont les objets de niveau 1 (Level 1), c'est a dire les objets parents.

```
/* ELFsh Level 1 object (= parent object) structure */
typedef struct s_L1handler
{
hash_t *l2list;    /* A ptr on the child L2 hashtable */
 u_int elem_size;  /* Size of one element of this object */

 /* Handlers */
```

```
void *(*get_obj)(void *file, void *arg); /* Read object */

/* Read multiple instanced L1 obj */
void *(*get_obj_idx)(void *file, u_int i, void *a);
void *(*get_obj_nam)(void *file, char *name); /* Read by name */
void *(*get_entptr)(void *file, u_int idx);  /* Get address */
u_int (*get_entval)(void *ptr);  /* Get value */
u_int (*set_entval)(void *ptr, u_int vaddr); /* Set value */
} elfshL1_t;
```

L'objet L1 est utilisé pour representer :

1. L'en-tete ELF [label : hdr]
2. La PHT [label : pht]
3. La table des symboles [label : symtab]
4. La table des symboles dynamiques [label : dynsym]
5. La section dynamique [label : dynamic]
6. La section GOT [label : got]
7. La section CTORS [label : ctors]
8. La section DTORS [label : dtors]
9. Les tables de relocation [label : rel]
10. La liste des sections [label : section]

8.1.2 Objet L2

Ce sont les objets de niveau 2 (Level 2), c'est a dire les objets fils.

L'objet L2 est utilisé pour representer chacun des objets fils des objets L1, c'est a dire concrétement, de la liste des champs de structure de chaque entrée des objets ELF (sht, pht, relocations, symboles, en-tete ELF, etc). L'objet L1 section est différent, en ce sens que ses objets L2 sont abstraits et ne coincide pas avec les champs de leur en-tete (pour cela, l'objet sht a été mis a disposition) mais permet d'indexer les données de la section. De même, les objets GOT, CTORS et DTORS n'admettent pas d'objets fils L2.

```
/* ELFsh Level 2 object (= L1 child) structure */
typedef struct s_L2handler
{

/* For fields */
int           (*get_obj)(void *obj); /* Read object */
int           (*set_obj)(void *par, u_int arg); /* Write object */

/* For names */
char          *(*get_name)(elfshobj_t *, void *obj); /* Get name */
```

```
  int           (*set_name)(elfshobj_t *, void *, char *); /* Set name */

  /* For sections data */
  char          *(*get_data)(elfshsect_t *, u_int off, u_int sizelem);
/* Read data */
  int           (*set_data)(elfshsect_t *, u_int, char *, u_int, u_int); /* Write data */
  char type;                  /* Object type */

  } elfshL2_t;
```

Voici la liste des objets L2 pour chaque objet L1 :

1. hdr [magic class type machine version entry phoff shoff flags ehsize phentsize shentsize phnum shnum shstrndx pax_pageexec pax_emultramp pax_mprotect pax_randmmap pax_randexec pax_segmexec]
2. sht [type offset addr size link info align entsize a w x s m l o]
3. pht [type offset paddr vaddr filesz memsz flags align]
4. symtab/dynsym [name value size bind type other]
5. dynamic [val tag]
6. section [name raw]
7. rel [type sym offset]

8.1.3 Objet abstrait PATH

Cet objet permet d'avoir une represéntation unique d'un objet, quelquesoit son type et sa profondeur. C'est un objet abstrait, aussi appelé meta objet.

int vm_lookup_param(char *path, elfshpath_t *pobj, u_int mode) ;
Cette fonction de l'API interne permet de résoudre un objet abstrait selon son chemin dans l'arborescence à deux niveaux des objets L1 et L2. Il permet egalement de représenter des valeurs immediates.

```
  /* Meta object */
  typedef struct s_elfshpath
  {

  /* Handlers */
  u_long             (*get_obj)(void *parent);
  u_long             (*set_obj)(void *parent, long value);
  char               *(*get_name)(elfshobj_t *, void *obj);
  int                (*set_name)(elfshobj_t *, void *, char *);
  char              *(*get_data)(elfshsect_t *, u_int off, u_int);
  int                (*set_data)(elfshsect_t *, u_int, char *, u_int, u_int);
```

```
elfshobj_t         *root;             /* Root parent */
void               *parent;         /* Direct parent */

u_int              off;               /* Optional byte offset */
u_int              size;              /* Size of the immediate string */
u_int              sizelem;           /* Size of element for OBJRAW */
char               immed;             /* Immediate binary flag */

/* Immediate value of immed flag is set */
union                 immval
{
  char                *str;
  long                ent;
}                     immed_val;

/* Here is the object type list */
#define               ELFSH_OBJINT 0       /* Dword */
#define               ELFSH_OBJSTR 1       /* String */
#define               ELFSH_OBJRAW 2       /* Raw */
#define               ELFSH_OBJUNK 3       /* Unknown */
char type;

}                     elfshpath_t;
```

Toutes les commandes ELFsh utilisent les objets PATH, et n'interagissent aucunement avec les objets L1 et L2. Ceux la sont uniquement connus du code de résolution de chemin interne, c'est à dire par la fonction vm_lookup_param() .

8.2 Modèle Ouvert

ELFsh dispose d'une interface de module simple, ainsi qu'une API interne de gestion de tables de hash et de gestion de commandes.

8.2.1 Modules

Voici un module de base :

```
#include "elfsh.h"

#define CMD_TEST "cmdtest"

int my_cmd_print()
```

```
{
  puts("\n [*] I do block the print command, oh yeah . \n");
  return (0);
}

int my_new_cmd()
{
  puts("\n [*] This is a new command, oh no ! \n");
  return (0);
}

void elfsh_init()
{
  puts(" [*] ELFsh modtest init -OK- \n");
  vm_setcmd(CMD_PRINT, my_cmd_print, ELFSH_ORIG, (u_int) ELFSH_ORIG);
  vm_addcmd(CMD_TEST, my_new_cmd, NULL, 0);
}

void elfsh_fini()
{
  puts(" [*] ELFsh modtest fini -OK- \n");
  vm_setcmd(CMD_PRINT, cmd_print, ELFSH_ORIG, (u_int) ELFSH_ORIG);
  vm_delcmd(CMD_TEST);
}
```

8.2.2 Interface des commandes

Il est possible de rajouter des commandes a la volée, et même de détourner des commandes existantes par cette voie. L'API interne se résume a des fonctions de manipulation d'objets abstraits comme ceux presentés précedemment, et des commandes du shell. Voici le log de la session ELFsh générée par l'utilisation de ce module :

```
[ELFsh-0.5b4]$ print 1 2 3 4

1 2 3 4

[ELFsh-0.5b4]$ cmdtest

 [!] Unknown command cmdtest .::. type 'help' for command list

[ELFsh-0.5b4]$ modload modules/modtest.so

 [*] ELFsh modtest init -OK-
```

```
[ELFsh-0.5b4]$ cmdtest

 [*] This is a new command, oh no !

[ELFsh-0.5b4]$ print toto 42

 [*] I do block the print command, oh yeah .

[ELFsh-0.5b4]$ modunload modules/modtest.so

 [*] ELFsh modtest fini -OK-

 [*] Module modules/modtest.so unloaded on Tue Feb 25 01:56:13 2003

[ELFsh-0.5b4]$ print 1 2 3 4

1 2 3 4

[ELFsh-0.5b4]$ cmdnew

 [!] Unknown command cmdnew .::. type 'help' for command list

[ELFsh-0.5b4]$
```

8.2.3 Tables de hash

En interne, le shell indexe tous ces objets par des tables de hash, dont l'API est également disponible depuis les modules. Dans la version 0.5b6, il est nécessaire d'utiliser directement ces tables de hash pour rajouter des objets L1 et L2 à l'arborescence. Le lecteur est invité à consulter la référence de l'API interne sur la page du projet [sc03] .

8.3 Portabilité

Toutes les fonctionnalités présentées dans ce exposé (injection de sections, détournement de fonctions, etc) sont compatibles INTEL et SPARC depuis la version 0.51 du shell ELF.

L'ASLR a été etudié sur machine INTEL uniquement, du fait des heuristiques de détection de faux positifs de relocation qui sont dépendantes de l'architecture et donc de libasm. Toutefois, les algorithmes de remapping présentés sont indépendants de l'architecture.

Le code est disponible pour Linux, NetBSD, FreeBSD et Solaris. Nous collaborons avec d'autres concepteurs afin de disposer d'une interface d'analyse ala libasm pour l'architecture SPARC.

Bibliographie

[Ces00] Silvio Cesare. Shared library redirection via elf plt infection. *Phrack Magazine*, 56, 2000.

[Con98] TIS Consortium. Elf portable format specifications. *TIS*, 1998.

[Des97] Solar Designer. Getting around non-executable stack. *Bugtraq post on www.securityfocus.com*, 1997.

[GNU03a] GNU. Binary utilities. *gnu.org*, 2003.

[GNU03b] GNU. Project gnu's extension language. *www.gnu.org/software/guile/*, 2003.

[Kir03] Vladimir Kiriansky. Execution model enforcement via program shepherding. *LCS Technical Memos*, 2003.

[Ner01] Nergal. Advanced ret2libc exploits. *Phrack Magazine*, 58, 2001.

[p5903] p59_09@author.phrack.org. Bypassing pax aslr protection. *Phrack magazine*, 59, 2003.

[sc03] The ELF shell crew. The elf shell. *elfsh.devhell.org*, 2003.

[Tea02] The PaX Team. Documentation for the pax project. *pageexec.virtualave.net*, 2002.

[Van01a] Julien Vanegue. Elf runtime fixup. *devhell.org*, 2001.

[Van01b] Julien Vanegue. Ia32 advanced function hooking. *Phrack Magazine*, 58, 2001.

[Van01c] Julien Vanegue. Understanding elf rtld internals. *devhell.org*, 2001.

Chapitre 9

Annexes

9.1 Annexe A

32bit mod R/M byte

r8(/r)	AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)	AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg	ES	CS	SS	DS	FS	GS	res.	res.
eee	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
eee	DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7
/digit (opcode)	0	1	2	3	4	5	6	7
reg=	000	001	010	011	100	101	110	111

effective address	mod	R/M	value of mod R/M byte (hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[sib]		100	04	0C	14	1C	24	2C	34	3C
[sdword]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F

[EAX+sbyte]	01	000	40	48	50	58	60	68	70	78
[ECX+sbyte]		001	41	49	51	59	61	69	71	79
[EDX+sbyte]		010	42	4A	52	5A	62	6A	72	7A
[EBX+sbyte]		011	43	4B	53	5B	63	6B	73	7B
[sib+sbyte]		100	44	4C	54	5C	64	6C	74	7C
[EBP+sbyte]		101	45	4D	55	5D	65	6D	75	7D
[ESI+sbyte]		110	46	4E	56	5E	66	6E	76	7E
[EDI+sbyte]		111	47	4F	57	5F	67	6F	77	7F
[EAX+sdword]	10	000	80	88	90	98	A0	A8	B0	B8
[ECX+sdword]		001	81	89	91	99	A1	A9	B1	B9
[EDX+sdword]		010	82	8A	92	9A	A2	AA	B2	BA
[EBX+sdword]		011	83	8B	93	9B	A3	AB	B3	BB
[sib+sdword]		100	84	8C	94	9C	A4	AC	B4	BC
[EBP+sdword]		101	85	8D	95	9D	A5	AD	B5	BD
[ESI+sdword]		110	86	8E	96	9E	A6	AE	B6	BE
[EDI+sdword]		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

32bit SID byte

r32		EAX	ECX	EDX	EBX	ESP	#1	ESI	EDI
base=		0	1	2	3	4	5	6	7
base=		000	001	010	011	100	101	110	111
scaled index	S	index	value of SIB byte (hex)						
[EAX*1+base]	00	000	00	01	02	03	04	05	06	07
[ECX*1+base]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX*1+base]		010	10	11	12	13	14	15	16	17
[EBX*1+base]		011	18	19	1A	1B	1C	1D	1E	1F

```
|[none*1+base]  |    |  100  | 20  | 21  | 22  | 23  | 24 | 25 | 26 | 27 |
|[EBP*1+base]   |    |  101  | 28  | 29  | 2A  | 2B  | 2C | 2D | 2E | 2F |
|[ESI*1+base]   |    |  110  | 30  | 31  | 32  | 33  | 34 | 35 | 36 | 37 |
|[EDI*1+base]   |    |  111  | 38  | 39  | 3A  | 3B  | 3C | 3D | 3E | 3F |
|---------------+----+-------+-----+-----+-----+-----+----+----+----+----|
|[EAX*2+base]   | 01 |  000  | 40  | 41  | 42  | 43  | 44 | 45 | 46 | 47 |
|[ECX*2+base]   |    |  001  | 48  | 49  | 4A  | 4B  | 4C | 4D | 4E | 4F |
|[EDX*2+base]   |    |  010  | 50  | 51  | 52  | 53  | 54 | 55 | 56 | 57 |
|[EBX*2+base]   |    |  011  | 58  | 59  | 5A  | 5B  | 5C | 5D | 5E | 5F |
|[none*2+base]  |    |  100  | 60  | 61  | 62  | 63  | 64 | 65 | 66 | 67 |
|[EBP*2+base]   |    |  101  | 68  | 69  | 6A  | 6B  | 6C | 6D | 6E | 6F |
|[ESI*2+base]   |    |  110  | 70  | 71  | 72  | 73  | 74 | 75 | 76 | 77 |
|[EDI*2+base]   |    |  111  | 78  | 79  | 7A  | 7B  | 7C | 7D | 7E | 7F |
|---------------+----+-------+-----+-----+-----+-----+----+----+----+----|
|[EAX*4+base]   | 10 |  000  | 80  | 81  | 82  | 83  | 84 | 85 | 86 | 87 |
|[ECX*4+base]   |    |  001  | 88  | 89  | 8A  | 8B  | 8C | 8D | 8E | 8F |
|[EDX*4+base]   |    |  010  | 90  | 91  | 92  | 93  | 94 | 95 | 96 | 97 |
|[EBX*4+base]   |    |  011  | 98  | 99  | 9A  | 9B  | 9C | 9D | 9E | 9F |
|[none*4+base]  |    |  100  | A0  | A1  | A2  | A3  | A4 | A5 | A6 | A7 |
|[EBP*4+base]   |    |  101  | A8  | A9  | AA  | AB  | AC | AD | AE | AF |
|[ESI*4+base]   |    |  110  | B0  | B1  | B2  | B3  | B4 | B5 | B6 | B7 |
|[EDI*4+base]   |    |  111  | B8  | B9  | BA  | BB  | BC | BD | BE | BF |
|---------------+----+-------+-----+-----+-----+-----+----+----+----+----|
|[EAX*8+base]   | 11 |  000  | C0  | C1  | C2  | C3  | C4 | C5 | C6 | C7 |
|[ECX*8+base]   |    |  001  | C8  | C9  | CA  | CB  | CC | CD | CE | CF |
|[EDX*8+base]   |    |  010  | D0  | D1  | D2  | D3  | D4 | D5 | D6 | D7 |
|[EBX*8+base]   |    |  011  | D8  | D9  | DA  | DB  | DC | DD | DE | DF |
|[none*8+base]  |    |  100  | E0  | E1  | E2  | E3  | E4 | E5 | E6 | E7 |
|[EBP*8+base]   |    |  101  | E8  | E9  | EA  | EB  | EC | ED | EE | EF |
|[ESI*8+base]   |    |  110  | F0  | F1  | F2  | F3  | F4 | F5 | F6 | F7 |
|[EDI*8+base]   |    |  111  | F8  | F9  | FA  | FB  | FC | FD | FE | FF |
|-------------------------------------------------------------------------|
|-------------------------------------------------------------------------|
|     note      |                      description                        |
|-------------------------------------------------------------------------|
|-------------------------------------------------------------------------|
|      #1       |if mod=00 then base=sdword                               |
|               |if mod=01 then base=EBP+sbyte                            |
|               |if mod=10 then base=EBP+sdword                           |
+-------------------------------------------------------------------------+
```

9.2 disasm_func.c

```
$ cat > disasm_func.c <<EOF
```

```
/*
** gcc -o disasm_func disasm_func.c -lasm -Lpath/to/libasm
*/

#include <stdlib.h>
#include <libasm.h>
#include <unistd.h>

void *extract_func(char *str, int len) {
  return (exit);
}

/*
 * fonction a analyser.
 */

int func(int fd, char *str, int len) {
  void (*ptr)();
  char buffer[256];
  int addr;

  addr = strtoul(str, 0, 16);
  ptr = extract_func(str, len);
  if (!addr)
    ptr = (void (*)()) 0;
  else
    snprintf(buffer, len, "%s", str);
  if (!ptr)
    return (0);
  ptr();
}

/*
  Cette fonction verifie si l'operande
  est une reference a un argument de la fonction.
  Pour cela, elle verifie tour a tour
  que le type de l'operande est ASM_OTYPE_ENCODED.
  Si oui, on verifie que les champs base et immediate
  sont bien presents.
  Si le registre est de base est EBP et que la
  valeur immediate est modulo et superieure ou egale a 8
  alors il s'agit d'une reference a un argument et
  la fonction renvoie le numero de l'argument.
  func(1, 2, 3, 4, 5, ...);
 */
```

```
int operand_is_argument(asm_operand *op) {

  if ((op->type == ASM_OTYPE_ENCODED) &&
      (op->content & ASM_OP_VALUE) &&
      (op->content & ASM_OP_BASE) &&
      (op->base_reg == ASM_REG_EBP) &&
      (op->regset == ASM_REGSET_R32)) {
    if (!(op->imm % 4) && op->imm >= 8)
      return ((op->imm - 4) / 4);
  }
  return (0);
}

int main (int ac, char **av) {
  asm_processor proc;
  asm_instr instr;
  u_char *ptr;
  int i;
  int len = 1024;
  int cur_len;
  int arg;

  if (ac < 2) {
    printf("Usage: %s \n", av[0]);
    return (-1);
  }

  ptr = (u_char *) func;

  /* initialise processor structure. */
  asm_init_i386(&proc);
  for (i = 0; i < len; i += cur_len) {
    if ((cur_len = asm_read_instr(&instr, ptr + i, len - i, &proc))) {

      /* RET found. leaving. */
      if (instr.instr == ASM_RET) {
printf("RET encountered\n");
break;
      }
      /*
       * Verification des operandes.
       */
      if (arg = operand_is_argument(&instr.op1)) {
printf("argument %i\n", arg);
      }
      if (arg = operand_is_argument(&instr.op2)) {
```

```
printf("argument %i\n", arg);
      }
    } else {
      printf("Disassembly error...\n");
      break;
    }
  }
  return (0);
}
EOF
```