# Interception de l'information sur le format ELF

Julien Vanegue

*may@epita.fr*
*Ecole Pour l'Informatique et les Techniques Avancees [EPITA]*
*Laboratoire Systeme et Securite [LSE]*
*Laboratoire de Recherche en Informatique Appliquee [LERIA]*

**Résumé**

Cet article explicite les problématiques d'interception de l'information au sein d'un binaire ELF (Executable and Linking Format). Le shell ELFsh [ELFsh] illustre les solutions apportés aux enjeux etudiés. Cette boîte à outil de manipulation d'objets ELF se compose d'un interpreteur de commandes (le shell) et d'une librarie de manipulation ELF (libelfsh). La bibliothèque de manipulation ELF (*libelfsh*) met a disposition plus de 250 opérations sous forme de fonctions C et permet la lecture et l'écriture des objets manipulés. Nous allons étudier les techniques implémentées par le logiciel pour insérer du code C compilé dans un binaire éxistant, dans le but de détourner ses fonctions vers nos propres routines. Le lecteur est invité a la lecture de la référence TIS [ELFTIS], qui pourra être utilisée comme première source d'information sur la norme ELF.

This article introduces information hijacking problematics in the ELF (Executable and Linking Format) environment. The ELF shell [ELFsh] illustrates the solutions and algorithms presented in this paper. The software is a swiss army knife for ELF objects manipulation, it is composed of a commands interpreter (the shell) coupled to the ELF manipulation library (libelfsh) and provides an ELF reverse engineering interface. The complete framework contains more than 250 operations, allowing advanced read/write capabilities : new interception and residency techniques are explained for inserting C compiled code into an executable as if the file was not yet linked. The novice audience is invited to consult the ELF TIS [ELFTIS] reference as a first approach.

## 1. Présentation des besoins

Les besoins de manipulation ELF que nous allons introduire ont nécessité le développement d'interface bas niveau, en délaissant les composants existants, comme *libbfd* [Binutils]. *Libbfd* est en effet une bibliothèque de translation binaire dont le backend ELF reste assez pauvre dès que l'on veut disposer de capacités de reverse engineering, comme le détournement de fonctions, la relocation d'object non relogeable, et tout autre capacité dont l'implémentation est hautement dépendante de l'architecture.

Globalement, les outils GNU ont été codés pour garantir une interface unique sur un grand nombre de formats, ce qui est parfait du point de vue de l'éditeur de liens Ces dernières années, le besoin de translation binaire se fait de moins en moins ressentir, puisque les principaux UNIX tendent à implementer ELF comme format principal.

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 intégralement des sources du systeme d'exploitation (et de ses packages) sur la machine locale. Il est aussi fréquent de ne pas pouvoir transférer ces sources intégralement dans un laps de temps raisonnable — elles ne sont parfois tout simplement pas disponible — parfois même la recompilation n'est pas désirée. 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 : la résidence (etude de l'injection) et l'interception (étude du détournement).

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

# 2. Interception du flux de contrôle

Le flux de contrôle est le chemin d'exécution du programme. Il est intéressant de l'étudier pour plusieurs raisons. Dans le cadre de la manipulation binaire, la mise en place de *hooks* (point de d'etournement) est particulierement interessante et ses applications sont multiples : audit, logging, déclenchement de fonctionalités supplémentaires, etc.

Certains modèles de hooks [IA32hooks] sont intéressants et peuvent être appliqués en partie sur des binaires ELF. L'inconvénient de [IA32hooks] est que ce modèle utilise un système 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'exécution, puisque les fonctions d'un objet ELF sont contenues dans les segments exécutables en lecture seule (c'est-à-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 éviterons de l'utiliser puisqu'il est en général filtré par les protections non-exécutable comme PaX. [PaXdoc]

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'exécution, en utilisant les sections `.got` et `.plt` [ELFsh, ELFfixup] : nous l'expliciterons dans la section 2.2.

## 2.1. Interception statique

Dans un premier temps, voyons les points d'entrées statiques du binaire. Ils existent quelque soit le contenu de l'exécutable, et peuvent être modifiés avec les commandes *write* et *set* dans ELFsh [ELFsh]. On distingue le point d'entrée du programme (champ `e_entry` dans le header ELF, qui pointe sur la fonction `_start`, généralement située au début 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`.

Voici le chemin d'exécution d'un programme sous *UNIX* :

```
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 très peu documentée [ELFrtld] : 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.

## 2.2. 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'exécution. En d'autres termes, il permet de résoudre les adresses des fonctions et variables externes à l'objet courant lorsque le programme le nécéssite.

En utilisant *ELFsh*, on remarque que les adresses dans la *SHT* (Section Header Table) et la *PHT* (Program Header Table) sont relatives par rapport à l'adresse base de la bibliothèque, 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`. Ces objets ont une étape de relocation plus complèxe que les binaires `ET_EXEC`. On distingue notamment des sections de relocation supplémentaire comme `.rel.text` ou `.rel.data` dans ces bibliothèques.

Quant à la relocation à la volée, elle 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 mappage initial. En fait, elle se résume à la mise à jour de pointeurs dans un tableau (la section `.got`) de manière indépendante pour chacune des entrées, lors de leur première utilisation.

Chaque entrée de `.got` correspond à l'adresse d'une fonction externe à l'objet dont il dépend. Par exemple, un programme qui dépend 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 celle cité précédemment, c'est à dire qu'une bibliothèque peut être mappée n'importe ou dans le processus.

Les trois premières entrées de.got sont des entrées spéciales, dont nous allons éclaircir le fonctionnement. En fait, chaque entrée de .got pointe par défaut (c'est-à-dire quand la .got n'a pas encore été 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 [ELFrtld].

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 de la .plt est spéciale sur les 2 architectures (les 4 premieres pour SPARC)

Les mechanismes sont similaires d'une architecture a une autre, excepté le fait que la section .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 à 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. La valeur par défaut de l'entrée de la .got est alors utilisée par l'entrée courante de la *.plt*. On constate que l'adresse qu'elle contient pointe sur la deuxiéme instruction de l'entrée correspondante dans la .plt. Cette deuxième 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.

Enfin, la première entrée (entrée spéciale) 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 l'éditeur de liens dynamiques. Cette troisième entrée .got [Binutils] est également réservée : elle est renseignée avant le lancement de la fonction main(). La fonction chargée de la mise à jour de la .got s'appelle en général dl-resolve(), même si ce n'est pas standardisé.

Quand l'entrée de la .got est remplie, la fonction de résolution à la volée appelle 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 la dl-resolve() ne sera jamais appelée pour cette entrée, et une fonction tierce pourra être appelée à la place.

À noter qu'il est possible de forcer le renseignement complet de la .got avant que la fonction main ne s'exécute. Pour cela, il suffit de créer une variable d'environnement LD_BIND_NOW et de la mettre à 1, elle sera directement interprétée par l'éditeur de liens dynamiques au lancement du programme. Dans ce cas, toutes les entrées de la .got sont mises à 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 externe [ELFplt] et ainsi utiliser un pointeur autre que celui renseigné par le linker dynamique dans la section .got.

Sur architecture SPARC, l'infection .plt est toujours nécessaire, puisque cette architecture n'utilise pas la .got pour la résolution d'adresse de fonction externe. Il est également possible de dupliquer la section .plt et utiliser la technique ALTPLT [CerberusELF] pour disposer d'une interface formelle

et portable pour le détournement de fonction.

# 3. Résidence

Voyons comment nous pouvons insérer du code et des données excédentaires dans un binaire ELF sans les sources du programme.

## 3.1. Première approche : le code PIC

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

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

Toutefois, il devient plus délicat d'utiliser les fonctions de l'objet hôte, ou les fonctions exportées de bibliothèques 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 sur les adresses dont il a besoin.

Cette méthode est naïve et devient trop complexe pour des injections de moyenne ou grosse envergure. Elle est également extrémement dépendante de l'architecture, puisque tout le code injecté doit être developpé en langage assembleur. Nous proposons plusieurs méthodes alternatives.

## 3.2. Zones d'alignement

Chaque segment de type `PT_LOAD` doit être aligné sur une page sur architecture INTEL (4096 octets) et sur architecture SPARC (8192 octets) . Cela laisse de la place pour un parasite dans la zone d'alignement d'un segment exécutable `PT_LOAD` . Comme cette zone n'est pas physiquement inséré dans le fichier mais que l'éditeur de liens s'assure que l'espace d'adressage est respecté, une telle résidence a pour conséquence le fait de devoir décaler tous les offsets de fichier pour chaque section se trouvant après le parasite, mais pas les adresses virtuelles des segments.

Par contre, il n'est pas envisageable d'utiliser les zones d'alignement de chacune des sections, la taille maximale de chacune d'entre elle étant 3 octets — car l'addresse basse d'une section doit etre alignée sur 4 — nous ne pourrions même pas coder une instruction de branchement telle que le `jmp` indirect, qui permettrait de passer le contrôle à une partie du parasite placée dans le padding des sections suivantes.

## 3.3. Ajout de section

Synchroniser la perspective par sections de l'éditeur de liens est intéressant dès lors que l'on souhaite reloger des objets entre eux, ou débugger le binaire hôte de manière décente. Nous pouvons

injecter les sections de plusieurs façons :

- section mappée de code ;
- section mappée de données ;
- section non mappée.

Les sections non mappées sont les plus simples à injecter puisque l'espace d'adressage ne doit pas etre modifié dans la `PHT`. Ces sections sont utiles pour stocker des informations, comme des tables de références, des tables de symboles excédentaires, ou toutes autres tables qui peuvent faciliter l'analyse par la suite.

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

Les sections de code peuvent être mappées à divers endroits. Une première approche consiste à les mapper dans la zone d'alignement du segment exécutable, mais cela limite la taille de notre code injecté. Nous pouvons également les injecter au début du fichier, entre la première section (section `NULL`) et la deuxième (généralement `.interp`). Cette deuxième 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 décaler les adresses virtuelles des sections suivantes.

Bien sûr, nous devons synchroniser la perspective du système, en modifiant la `PHT` (particulièrement les champs de l'en-tête des segments mappés) afin que les données des sections injectées soient inclues dans le segment approprié. L'algorithme d'insertion de section se resume à la création et l'insertion de l'en-tête dans la `SHT` et á l'insertion de la nouvelle section physiquement dans le fichier.

À chaque insertion de section, les tables de symboles doivent être synchronisées, plus precisément le champ d'index de section (`st_sctndx`) doit être mis à jour pour chaque entrée de la table des symboles. Ce champ est notamment utilisé pour indiquer que le symbole est non-defini (`SHN_UNDEF`) ou *commun* (`SHN_COMMON`), autrement il indique la section parente de l'objet référencé par le symbole. De même, l'index de la section des noms de sections `.shstrtab` doit être mis à jour dans l'en-tête ELF (`e_shstrndx`) si la section injectée a été inserée apres celle ci.

## 3.4. Extension de segments

Dans le cas d'une injection de section post-bss, nous devons fixer le segment `PT_LOAD` autorisé en écriture (`rw-`) pour qu'il prenne en compte l'insertion du bss (en modifiant la taille physique `p_filesz` par la valeur de la taille mémoire `p_memsz` de l'entrée 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 écriture (`aw-`)
    - Changer `p_filesz` pour la meme valeur que `p_memsz`
    - Étendre `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 de base du segment `PT_LOAD` exécutable (`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'éviter de reloger l'exécutable lui-même à chaque insertion, et n'admet pas de limite de taille, contrairement à l'injection par remplissage 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 à `p_filesz` et `p_memsz`
    - Soustraire la taille de la section injectée à `p_vaddr` et `p_paddr`
  - Sinon si le segment est de type `PT_PHDR`
    - Soustraire la taille de la section injectée à `p_vaddr` et `p_paddr`
  - Sinon
    - Ajouter la taille de la section injectée à `p_offset`

L'avantage ultime de cette méthode est sa compatibilité avec les environnements de type non-executable comme PaX [PaXdoc], qui refusent toute exécution de code en dehors des segments exécutables `PT_LOAD` (dans le but principal de protéger des exploitations de failles de sécurité de type *buffer overflow* dont le *shellcode* s'exécute dans la pile ou le tas). Il n'existe à priori, pas d'autre méthode pour insérer plus d'une page de code dans ce segment dans un environnement non exécutable. Si l'environnement n'est pas non-exécutable, il est possible d'utiliser l'injection post-bss pour insérer du code.

## 3.5. La section `.dynamic`

Cette section permet d'indexer toutes les tables nécessaires á l'édition de liens dynamiques. Nous trouvons depuis cette table les adresses virtuelles des tables de relocation, tables des symboles, tables de hash des symboles. Nous trouvons également des entrées de type `DT_NEEDED` qui sont intéressantes pour la résidence, puisqu'elles indexent les dépendances (bibliothèques : type ELF `ET_DYN`) du binaire. Nous pouvons donc :

- ajouter des entrées de type `DT_NEEDED`;
- modifier une entrée existante pour la transformer en `DT_NEEDED`.

L'ajout d'une entrée peut être problématique. En effet, la section `.dynamic` se trouve juste avant le `.bss`, ou de manière générale, entre les sections de données initialisées (`.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 problématique de l'ASLR statique (Address Space Layout Randomization) [ELFINTEL] .

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, associée à l'entrée. Dans le cas d'une entrée `DT_NEEDED`, le champ `d_val` contient un offset en octets depuis le début de la section `.dynstr`, où se trouve la chaîne de caractéres du nom de la bibliothèque.

Certaines entrées ne sont pas obligatoires pour que l'exécutable puisse etre lancé, par exemple les entrées de type `DT_DEBUG` . Si nous changeons le `d_tag` pour y mettre `DT_NEEDED`, et que nous changeons `d_val` pour y mettre l'offset d'une chaîne de caractères valide en tant que nom de bibliothèque, nous pouvons donc rajouter une dépendance au binaire ELF `ET_EXEC` sans injection de données supplementaires, sans changer sa taille et sans relocation. Il faut pour cela trouver un nom de bibliothèque valide pour le système, terminé par `NUL`, et réutiliser partiellement la chaine de caractère déja présente dans l'executable.

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 excédentaire dans une bibliothèque.

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 excédentaires. Est-il aisé d'injecter des modules relogeables dans un objet `ET_EXEC` non-relogeable ? La réponse est oui.

### 3.6. Injection `ET_REL` dans `ET_EXEC`

Cette méthode est très 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 à insérer des copies des sections mappées du module dans l'exécutable, 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 à la section suivante.

La deuxième passe permet de reloger chaque section injectée en utilisant sa table de relocation dans l'objet relogeable `ET_REL`. Étant donné que l'exécutable lui-même n'a pas besoin d'etre relogé (puisque ses sections mappées ne sont pas deplacées dans l'espace d'adressage lors de l'insertion d'un ou plusieurs modules), cette implementation à l'avantage de ne pas générer de faux-positifs de relocation. Voici l'algorithme de cette deuxiéme 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 injectée à 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 à la section suivante.
  - Sinon passer à la section suivante.

Le fait d'utiliser toutes les tables des symboles permet de relinker à 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. De même, cette implémentation fonctionne sur les systèmes protegés par PaX [PaXdoc], puisque toutes les injections de code étendent directement la zone exécutable du binaire, décrite par l'entrée de la PHT dont les droits sont `r-x`.

## 4. Conclusion et perspectives

Grâce à ces algorithmes, l'ajout de fonctionnalités dans un programme dont la source n'est pas disponible est devenu presque aussi simple que pour un projet dont la source est publique. Toutes les fonctionnalités d'injection et de détournement présentées dans cet article sont compatibles *INTEL* et *SPARC*. Le code est disponible pour les systémes Linux, NetBSD, FreeBSD et Solaris. Nous collaborons avec d'autres concepteurs sur le portage á d'autres architectures.

## Références

[ELFsh] The ELF shell, elfsh.devhell.org

[Binutils] Binary utilities of the GNU project

[ELFTIS] ELF TIS portable format specifications

[PaXdoc] Documentation for the PaX project, pageexec.virtualave.net

[IA32hooks] IA32 advanced function hooking, Phrack Magazine 58

[ELFfixup] ELF runtime fixup, elfsh.devhell.org

[ELFrtld] Understanding Linux ELF RTLD internals, elfsh.devhell.org

[ELFplt] Shared Library Redirection via ELF PLT infection, Phrack Magazine 56

[ELFINTEL] Reverse engineering des systémes ELF/INTEL, SSTIC03

[CerberusELF] The Cerberus ELF interface, Phrack Magazine 61