ERESI : une plate-forme d'analyse binaire au niveau noyau

Anthony Desnos, Sebastien Roy, and Julien Vanegue

The ERESI Team
team@eresi-project.org

Résumé *ERESI* est une interface pour l'analyse de code machine sur les systèmes d'exploitation UNIX. Nous présentons ses fonctionnalités d'analyse statique et dynamique directement utilisables depuis un langage spécifique au domaine du reverse engineering, du debugging et de l'audit de programmes sans les sources. Notre interface est organisée en sous-ensembles de commandes permettant d'exécuter des routines de programmes écrites en langage *ERESI*, ce qui facilite l'automatisation des tâches de détection de vulnerabilités, d'extension d'exécutables et de bibliothèques, de vérification d'intégrité, ou de journalisation des évènements internes des programmes. Les techniques de manipulation binaire sur disque ou en mémoire exportées par *ERESI* permettent l'analyse d'environnements protégés tel un Linux compilé avec des protections d'exécution de la memoire (PaX) ou lors d'introduction d'aléa dynamique dans l'espace d'adressage (ASLR). Dans cet article, nous montrons comment nous avons étendu notre analyse au mode superviseur en permettant d'inspecter l'état du noyau Linux en temps réel, de le modifier et de le débugger, d'y injecter du code C compilé, et de détourner les fonctions du noyau même si celle-ci ne sont pas exportées, et ce directement dans le langage *ERESI*. Nous avons décliné ces nouvelles fonctionnalités en deux nouveaux outils : *Ke2dbg* (noyau Embedded ERESI debugger) et *Kernsh* (The noyau shell).

1 Introduction

Le reverse (code) engineering (*ingénierie inverse*, *retro-ingénierie*, ou plus simplement *RE*) est l'étude des systèmes logiciels et matériels fermés et la compréhension de leurs comportements. Bien que le code source des logiciels soit parfois disponible, nous considérons le code binaire comme la matière première de notre analyse.

Les besoins qui mènent à la mise en œuvre de cette activité sont variés et peuvent englober le portage d'un pilote d'une plateforme vers une autre, la recherche de violations de propriété intellectuelle, ou encore l'audit de la sécurité d'une infrastructure.

Des systèmes à étudier et des besoins, les méthodes et outils mis en œuvre vont dès lors évoluer. Chacune des interfaces fournie par *ERESI* donne une perspective différente sur le programme étudié, mais avec certaines contraintes, étant donne la spécialisation de chacune des primitives ERESI. Chacun des outils permet d'obtenir les informations internes de programmes et d'éxpliciter les interactions d'un logiciel avec son environnement : entrées utilisateurs, flux réseaux, évènements d'exécution...

1.1 Analyse de programmes

Deux approches de *RE* coexistent : D'une part, l'analyse statique qui consiste à étudier l'image du programme exécutable (binaire, bibliothèque..) à travers ses différents aspects : flot exécution,

flots de données, typage manuel ou infèrence automatique. L'analyse dynamique consiste quant à elle à s'interesser au comportement de ce système en cours d'exécution au moyen d'un debugger ou d'un traceur. L'ensemble de cet outil permet donc de réaliser l'analyse de logiciels sous ces deux aspects.

Analyse statique :
L'analyse de code statique consiste á étudier la structure d'un logiciel en désassemblant son code exécutable. Cette analyse permet de distinguer les différents blocs logiques de code machine d'un binaire afin de les séparer en entités liées entre elles par une relation de transfert : appel de procédure, saut conditionnel. De cette manière, on obtient une représentation logique du graphe d'exécution. Il s'agit de l'ensemble des différents chemins possibles que peut suivre un programme lors de son exécution. De cette représentation, on peut extraire différentes informations comme l'ensemble des fonctions appelées ainsi que leur organisation, les conditions rencontrées, les itérations ... L'analyse statique fournie par ERESI n'est pas le thème de cet article, un futur artile dédié à l'analyse statique dans ERESI est en cours de préparation.

Analyse dynamique :
L'analyse dynamique consiste à exécuter un logiciel et à l'interrompre à un instant t afin d'en connaître l'état. Le mécanisme de déclenchement de l'interruption peut-être est choisi par l'utilisateur. Cela peut être l'appel d'une fonction, le déclenchement d'un signal suite à un point d'arrêt. Il est alors possible de récupérer l'état du processus, le contenu de sa mémoire, l'organisation de ses structures de données grâce au typage fourni par la *libaspect*. Ces informations complèteront une analyse statique pour offrir une meilleure représentation du systéme à étudier. Cet article présente les fonctionnalités d'analyse dynamique de ERESI sur le noyau Linux.

2 La plate-forme ERESI

ERESI est un ensemble d'outils, logiciels et bibliothèques, dediés au reverse engineering des binaires ELF sur différentes architectures. Au cœur de ceux-ci, on trouve une interface unifiée basée sur un langage de script commun à tous les outils.

2.1 Les bibliothèques dans ERESI

Historiquement, l'interface ERESI se compose de différents modules qui ont été développés progressivement afin d'enrichir le logiciel du shell ELF (*ELFsh*). Au fil des années, les fonctionalités ont cependant évoluées bien au delà de la gestion du format binaire ELF en prenant une dimension logique indépendante au support du format de fichier, de l'architecture, du système d'exploitation, ou de l'environnement d'exécution. Les différents outils et bibliothèques d'*ERESI* permettent d'offrir un cadre de manipulation et d'analyse des binaires mettant ainsi à disposition des utilisateurs la possibilité de réaliser manuellement et de programmer de nombreuses opérations d'analyse logicielle sans leur code source.

Nous allons parcourir l'ensemble de ces modules á partir du cœur pour finir vers les outils haut niveau, dont les nouveaux outils *kernsh* et *ke2dbg* qui feront l'objet d'une présentation plus complète.

Libelfsh :
La *Libelfsh* est historiquement le premier composant à avoir été developpé. Face aux limitations des outils existants, le développement d'une bibliothèque s'est imposé pour offrir les primitives de manipulations statiques de binaires 32 et 64 bits. *Libelfsh* offre entre autre l'accès 'a l'ensemble des informations contenus dans un fichier ELF mais aussi la possiblité de modifier un binaire pour manipuler le flux d'exécution gràce á un ensemble de techniques d'injection (*DT_DEBUG*, *injection ET_REL* statique ou á l'exécution, *EXTSTATIC*) et de détournement : (*ALTPLT*, *ALTGOT*, *CFLOW*). Ces techniques ont déjà fait l'objet de plusieurs articles á l'époque de leur developpement[17] [11].

Libasm :
La bibliothèque *Libasm* a été à l'origine développée afin de fournir à *ELFsh* le désassemblage des différentes sections de code d'un binaire. Avec le temps, ce composant s'est enrichi afin de supporter plusieurs architectures (IA32, Sparc, Mips), tout en offrant une classification sémantique des instructions et de leurs opérandes á l'aide d'attributs, afin de rendre certaines analyses statiques indépendante de l'architecture. Ce module a été présenté lors du SSTIC 2003 : [18].

Libmjollnir :
La *Libmjollnir* est l'évolution du module *modflow* présenté à l'édition 2003 du SSTIC. Cette bibliothèque permet de décompiler le flux d'exécution d'un programme et de réaliser des empreintes de fonctions grâce à un hash md5 du code. C'est de ce composant dont dépendent l'analyse de graphes de controle.

Librevm :
La *Librevm* implémente l'interpréteur du langage de script *ERESI*. Elle fournit une interface capable d'exécuter les scripts *ERESI* de listing de commandes, de lancer les fonctions d'analyse, et de gérer les variables de script. Les différentes commandes proposées permettent de disposer d'un langage souple, d'un environnement configurable avec la possibilité d'enregistrer des sessions et de sauver une analyse en cours pour la reprendre ultérieurement.

Libaspect :
La libaspect met à disposition un ensemble de d'outils de modélisation d'objets complexes : tableaux associatifs, vecteurs de portabilité et système de types dépendents. L'API des vecteurs consistue l'interface privilégiée du cadre ERESI pour rendre celui-ci modulaire.
Différents vecteurs sont déjà implémentés dans ERESI pour fournir le squelette de certaines fonctionnalités independemment des details d'implementation : désassemblage, méthodes d'interceptions du flot d'exécution ou d'injection spécifique à chaque architecture. Les primitives de decompilation de types sont implementé à ce niveau afin d'être disponibles dans tous les composants du cadre *ERESI*.

Libstderesi :
La *Libstderesi* est la bibliothèque qui va permettre aux différents composants de haut niveau d'*ERESI* d'accéder à toute l'interface du cadre *ERESI*. Elle englobe les commandes du langage permettant l'accès aux fonctionalités de libelfsh, libasm, libmjollnir et librevm.

Libedfmt :
Ce composant permet de traiter les informations de debugging (si celles-ci sont disponibles dans les binaires) afin de les mettre à disposition par une interface unifiée. Si la présence de ces informations dépend des options choisies lors de la compilation d'une application et ne doivent pas être essentielles au reverse engineering, elles peuvent néanmoins fournir des renseignements complémentaires sur le logiciel analysé tels que des noms de fonction, de variables ou le format de structures. Le support de ces formats rendent donc plus lisibles les resultats d'analyse, sans être indispensable pour l'utilisateur aguérri. Les formats de symboles de debug supportés actuellement sont les formats *STABS* et *Dwarf2*.

libetrace :
La bibliothèque *libetrace* regroupe les primitives ERESI pour le tracing de processus. Le traceur est designé pour être plus léger que le debugger tout en gardant une approche d'analyse dynamique bien adaptable à l'utilisateur. Il est ainsi possible de déclarer des listes de fonctions devant etre tracées et activer des options de tracing à cette granularité selon des évènements de l'exécution. Notre technique de tracing est embarquée : le code du traceur réside dans le même processus que le code trace. Cela nous permet d'avoir une performance optimale de tracing, portable sous *Linux* et *FreeBSD*.

libe2dbg La bibliothèque *libe2dbg* est un module agrégat : en plus d'implanter une interface pour le debugging embarqué de processus utilisateurs, elle est liée aux autres bibliothèques afin de fournir un interpréteur de langage ERESI totalement embarqué dans le debugger. Il est ainsi possible d'exécuter des scripts ERESI analysant le processus sans générer de changements de contextes, comme expliqué dans notre article précédent [14].

liballocproxy :
La bibliothèque *liballocproxy* est un élément clef de la couche bas-niveau dans l'architecture d'*ERESI*. En effet, lors de l'injection de notre debugger embarqué dans une application, il est nécessaire de ne pas altérer les données dynamiques de celui-ci. Afin d'éviter celà, la plate-forme *ERESI* utilise une méthode nommée *allocation proxy* qui utilise l'appel système *mmap* pour créer sa propre zone mémoire et y gèrer son propre sous-espace de mémoire dynamique. Le cadre *Eresi* utilise ses propres fonctions *malloc*, *calloc*, *free*, *realloc* etc.. basées sur l'implémentation *ptmalloc2*. L'utilisation de cette bibliothèque consiste donc à utiliser deux allocateurs concurrents dans un même processus analysé. Elle est portable sur un grand nombre de systèmes d'exploitations et globalement indépendante de l'architecture.

libkernsh :
La dernière avancée de la plate-forme ERESI consiste en libkernsh. Cette bibliothèque comporte l'API de base pour les fonctionalités en rapport au noyau du système d'exploitation. *Libkernsh* est developée sous *Linux*, mais certaines de ses interfaces sont aussi compatibles sous FreeBSD et NetBSD. Nous allons rentrer dans les détails de ces nouvelles capacités dans le chapitre dédié a Kernsh. Grossièrement, la libkernsh permet de rendre utilisable le moteur de l'analyse ERESI sur le noyau du système, en utilisant des interfaces dépendantes du système d'exploitation analysé.

libke2dbg :
La bibliotheque *libke2dbg* fournissent les API de debugging des programmes. Ils permettent de travailler sur le noyau en cours d'exécution sur le système. Alors que *libetrace* ne permet pas le lancement de script ERESI lors d'evènements (elle permet seulement l'appel de routines écrites en C), cette bibliothèque sont modelisées pour être en mesure de lancer des analyses complexes écrites en langage ERESI. Une des contributions de cet article est la description de l'analyse noyau dans ERESI.

2.2 Le scripting de haut-niveau dans *ERESI*

ERESI offre plusieurs outils basées sur ces bibliothèques qui exploitent les techniques implémentées afin de libérer les auditeurs des contraintes liées au développement de module en C en profitant d'interpréteur de commandes. Voici les différents outils de l'environnement d'analyse et une brève introduction des tâches auxquels ils sont dédiés.

***Elfsh* : Manipulation statique de binaires** :
Elfsh est un interpréteur de commandes interactif et modulaire qui permet la manipulation des binaires elf : exécutables, bibliothèques partagées. Il implémente les différentes fonctionnalités offertes par la bibliothèque *libelfsh* afin de mettre à portée de script une partie de l'API évoquée précédemment. Grâce à cet outil, il est possible d'obtenir une description détaillée d'un binaire et de modifier celui-ci sans aucune information de debug et sans l'exécuter. Grâce a son moteur de redirection, il permet aussi de modifier un binaire afin de détourner son flux d'exécution et détudier ou de modifier ses mécanismes. En outre, certaines méthodes employées permettent de passer outre les restrictions que peuvent imposer certaines protections d'espace d'adressage (ASLR, non-exec) implémentées dans les derniers noyaux.

***E2dbg* : Debugging embarqué** :
e2dbg est l'implémentation du debugger du projet *ERESI*. Celui ci s'exécute à l'interieur du processus à debugger. Il y est injecté grâce à la variable d'environnement LD_PRELOAD qui permet de charger notre outil directement dans l'espace mémoire du processus sans nécessiter l'appel à un programme externe utilisant l'appel système *ptrace*. Il permet de poser des breakpoints, de consulter l'état des registres ou de backtracer. Les autres fonctionnalités déjà offertes par Elfsh comme l'injection de code, le détournement de fonctions sont disponibles aussi. *e2dbg* a déj'a fait l'objet d'un article : Next Generation Debuggers for Reverse Engineering [14].

***Etrace* : Tracing embarqué** :
Etrace est le composant d'*ERESI* dédié au tracing embarqué d'un logiciel. Grâce ''a un système de *hook* et á l'injection de code dans les binaires, il permet de suivre le déroulement d'un programme en cours d'exécution en laissant á l'utilisateur la possibilité d'insérer ses propres fonctions au sein d'un logiciel et ainsi de filtrer les entrées et sorties en fonction des besoins d'audit. Cette technique embarquée permet de s'affranchir de l'utilisation de l'appel système *ptrace()*. On pourra se réferer à l'árticle cité précedement [14].

Dans la suite de cet article, nous aborderons les méthodes développées afin d'analyser le noyau linux en cours d'exécution. Dans ce cadre, les techniques développées dans les bibliothèques d'*ERESI*

pour les logiciels en mode utilisateur, comme l'interception de fonctions ou l'injection de code source compilé, restent valables. Cependant, des contraintes inhérentes à la nature même du noyau se posent maintenant. En effet, toute altération d'un noyau en cours d'exécution peut entraîner une erreur fatale au systeme, ce qui rend cette analyse du contexte superviseur bien délicate. L'utilisation d'une machine virtuelle pour developper et utiliser ces outils est fortement preconisèe, si ce n'est indispensable.

3 Analyse dynamique du noyau Linux : *Kernsh*

Le noyau, lorsque son nom est prononcé, en affame certains car il ouvre à lui seul toutes les voies pour un pirate ou un administrateur. Ses dernières années, il a été torturé dans tous les sens possibles et imaginables (injection de code, *backdoor*, redirection de fonction, etc).

Kernsh, premier du nom [6], avait pour vocation de créer un outil de communication user-land $\rightleftarrows$ noyau-land. Réecrit dans *ERESI*, il a pu bénificier de toutes les bibliothèques déjà disponibles pour augmenter ses fonctionnalités, et s'offrir en bonus un plus grand support de portabilité et de maintenabilité.

3.1 État de l'art en analyse noyau

L'analyse noyau n'en n'est pas à ses balbutiements. En effet la modification des appels systèmes, de fonctions noyau par un LKM [7] ou par /dev/(k)mem [8] est bien connue. Injecter du code en effectuant les relocations nécessaires peut être aussi très pratique et a été déjà developpé pour patcher le noyau static [4] ou bien encore à la volée [1]. .

Mais toutes ces méthodes de détournement de flot, ne peuvent être mises enœuvre bien souvent qu'après une compromission. Et bien souvent cell- ci passe par une vulnérabilité du noyau. L'exploitation de ces failles a fait l'objet d'articles divers [10] [3].

Nous avons voulu apporter de nouveaux outils pour explorer plus simplement cette dimension de l'analyse binaire, donnant ainsi (re)naissance à *Kernsh* et *Ke2dbg*.

3.2 Introduction à *Kernsh*

Kernsh se présente sous la forme d'un shell userland, interactif, modulaire et directement scriptable en language *ERESI*. En utilisant une bibliothèque appelée *Libkernsh* qui permet la communication de l'*userland* vers le *noyauland*, il permet l'accès dynamiquement ou statiquement l'accès à l'image du noyau ou à sa mémoire .

Kernsh peut appliquer presque toutes les fonctionnalités *userland* de ERESI au niveau noyau. De plus, *Kernsh* rend possible l'obtention et la modification d'information noyau directement depuis le language *ERESI*. Il peut accéder à la table des appels systèmes, la table des descripteurs d'interruptions, la table globale de descripteurs, les symboles, insérer des modules en mémoire sans le support des *Loadable Kernel Module*, rediriger des fonctions noyau, visualiser la mémoire des processus.. En fait, les possibilités d'analyse repoussent les limites de la créativite de leur utilisateurs grâce a l'interface de script dans le langage ERESI .

Étant intégré dans le *cadre ERESI*, *Kernsh* peut créer des nouvelles définitions de types et associer á ces nouveaux types des adresses de la mémoire noyau. Donc, potentiellement tous les objets noyaus peuveut être accéés depuis les scripts *ERESI* une fois que leur format a été défini par l'utilisateur (des types prédéfinis sont fournis pour le noyau Linux 2.6) .

Pour assurer toutes ces fonctions, la bibliothèque *Libkernsh* doit dialoguer avec le noyau. On a donc plusieurs moyens de faire cela :

- Le périphérique de la mémoire virtuelle : */dev/kmem*
- Le périphérique de la mémoire physique : */dev/mem*
- Le core dump du noyau : */proc/kcore* (seulement en lecture)
- Loadable Kernel Module, via un appel système ou le système de fichier */proc*

Les périphériques */dev/kmem* et */dev/mem* sont bien connus et peuvent être désactivés par des patchs comme grsecurity [12] pour prévenir l'installation de certains rootkits. Mais le support des *LKM* peut être absent et donc nous fournissons un module permettant d'accéder à la mémoire via le système de fichier /proc ou un appel système.

De plus, étant intégré dans ERESI, nous bénificions donc des types. Nous pouvons ainsi recréer certaines structures du noyau, les consulter, les annoter, ou modifier leur valeur dans la mémoire du noyau. Il devient ainsi donc trivial de changer la valeur des appels systèmes, des descripteurs d'interruptions, etc..

3.3 Fonctionnalités de *Kernsh*

Kernsh met en place un système de communication user-land $\rightleftarrows$ noyau-land. Nous avons donc implémenté directement la lecture et l'écriture dans l'espace noyau en language ERESI, ainsi que l'injection de LKM, et aussi la redirection de fonctions noyau. Le fait que nous travaillions sur le noyau est transparent pour *Kernsh*, ceci au prix d'une modification (en fait, d'une modularisation) du cadre ERESI. Seule la fonction permettant la lecture et l'écriture de la mémoire a ete modularisée pour la rendre adaptable á n'importe quel environnement mémoire. Il est possible á l'utilisateur de rajouter le support d'un autre vecteur d'entrée-sortie pour obtenir l'ensemble de l'API ERESI á sa disposition par ce nouveau point d'entrée á la mémoire.

Mais Kernsh ne s'arrête pas la et propose d'autres fonctionnalités :

- Affichage de la table des appels systèmes, des symboles.
- Affichage des descripteurs d'interruptions.
- Affichage des Global / Local Descriptor Tables (GDT, LDT).
- Désassemblage de la mémoire du noyau.
- Lecture et modification de l'image statique du noyau.
- Allocation et libération de memoire (non contiguë ou contiguë) dans le noyau.
- Hash du code des fonctions du noyau ou de portions de code.
- Redirection de la fonction d'initialisation des LKM (modules noyau).
- Lecture et écriture dans la mémoire virtuelle des processus.
- Dump des VMA des processus (structures internes de l'allocateur du noyau Linux).

Organisation de l'API : En utilisant la bibliothèque *libaspect*, *Kernsh* emploie les vecteurs, ce qui lui permet de supporter et d'ajouter très facilement de nouvelles fonctionnalités selon les architectures,

les systèmes d'exploitations, leur version ou bien encore diverses options. Par exemple la fonction de lecture en mémoire est :

```
int     kernsh_readmem(unsigned long, void *, int);
```

Elle prend en paramètre, l'adresse où lire, le buffer où écrire l'information et sa taille. Ainsi quelque soit les paramètres de la machine et le mode d'accès à la mémoire, elle reste la même car les vecteurs s'occupent de retrouver quelles fonctions de lecture de la mémoire a été enregistré pour cette configuration.

Vecteurs	Descriptions
Mémoire	Ouvrir, Fermer, Lire, écrire dans la mémoire noyau
Table des appels système	Affichage, Appel
Table des descripteurs d'interruptions	Affichage
Table globale de descripteurs	Affichage
Variables noyaus	Récuperation de variable importante du noyau
Symbols	Récuperation de l'adresse ou d'un nom d'un symbol noyau
Gestion mémoire	Allocation et Libération de memoire noyau (contiguë ou non)
Autotypes	Création automatique des structures noyaus sans les headers
Modules	Lien, Infection, Chargement, Déchargement des modules
Mémoire processus	Lire, écrire dans la mémoire virtuelle des processus
Virtual Memory Area	Récupération et Dump des VMA d'un processus

Il peut être surprenant de ne pas voir des fonctions simples comme la modification des adresses des appels systèmes ou des interruptions dans les tables correspondantes, mais comme dit plus haut tout cela est fait directement avec le language ERESI et l'annotation des types. Par exemple si l'on veut détourner l'appel système *write* par une adresse de notre choix :

```
#kernsh

type table = sys :int[320]              ⟵ Creation du type, ici un simple tableau d'entier
inform table tableinform $sct           ⟵ Annotation du nouveau type sur le symbole tableinform
                                          (syscall table)

mode dynamic                            ⟵ Passage en mode dynamique (runtime / memoire)
set 1.table[tableinform].sys[4] addr    ⟵ Changement de l'appel systeme write par l'adresse addr
```

Fig. 1: Exemple d'annotation manuelle de types sur la mémoire du noyau

Injection de code dans le noyau : L'injection de code permet l'insertion d'un object relogeable (*ET_REL*) dans l'espace d'adressage d'un programme. C'est une méthode puissante qui permet

d'injecter du code et des données autant que nécessaire. Pour cela *Kernsh* réutilise la technique de relocation des *ET_REL* [17], développée dans *Libelfsh* et donc ici transposée à la mémoire noyau.

Comme *libelfsh* est très modulaire, son algorithme de relocation ne nécessite que peu de modifications pour s'interfacer avec la mémoire runtime du noyau. Ainsi, seuls 3 endroits sont à modifier (correspondant a quelques lignes de C dans ERESI) :

- En adaptant la fonction de récupération d'adresse qui retourne la bonne adresse en mode static ou dynamic (*elfsh_get_raw*) en renvoyant une adresse en espace noyau
- En écrivant l'objet dans la mémoire noyau (via la fonction *kernsh_writemem*)
- En allouant de l'espace dans la mémoire noyau (par la fonction *kernsh_alloc*)

Redirection de fonction noyau : Kernsh peut là encore bénificier de la redirection de fonction de type *CFLOW* [11] intégrée dans *Libelfsh* pour les fonctions du noyau. Ce type de redirection permet en insérant de simple jump au début des fonctions de rediriger l'appel d'une routine vers une autre de notre choix (nouvellement alloué par exemple).

Mémoire virtuelle des processus : Pour pouvoir modifier la mémoire virtuelle d'un processus (codes, données), *Kernsh* fournit un module noyau réalisant cette fonction. Les *Virtual Memory Area* sont des régions contigües d'espace d'adresses virtuelles. Ces régions sont crées pendant la vie des processus quand un programme demande une projection mémoire d'un fichier, un lien vers un segment de mémoire partagées ou bien encore alloue de l'espace au tas.

Kernsh permet de récupérer ces *VMA*, de les lire et également d'y écrire. Mais il intègre directement des fonctions de *dump* automatique de ces sections sur le disque. Ainsi un programme ne peut plus tromper au niveau utilisateur la récupération d'informations comme avec l'appel système *ptrace*. *Kernsh* propose donc plusieurs commandes permettant d'utiliser toutes ces fonctionnalités (pour plus de détails sur les paramètres [15]) :

Commandes	Paramètres	Descriptions
openmem	/	Ouverture de l'accès à la mémoire noyau
closemem	/	Fermeture de l'accès à la mémoire noyau
mode	static/dynamic	Changement du contexte
kmem_read	adresse taille	Lecture dans la mémoire noyau
kmem_write	adresse buffer	écriture dans la mémoire noyau
kmem_disasm	adresse taille	Désassemblage de la mémoire noyau
kmem_info	/	Affichage des options en mode *humain*
kmem_hash	adresse taille	Hash d'une partie de la mémoire noyau
kmem_chash	hash | fichier	Vérification d'un hash
sct	/	Affichage de la table des appels systèmes
idt	/	Affichage de la table des descripteurs d'interruptions
gdt	/	Affichage de la table globale de descripteurs
alloc	taille	Allocation d'un espace contigue en mémoire noyau
free	adresse	Libération d'un espace contigue en mémoire noyau
alloc_nc	taille	Allocation d'un espace non contigüe en mémoire noyau
free_nc	adresse	Libération d'un espace non contigüe en mémoire noyau
kmodule_relink	orig new res	Link deux modules
kmodule_infect	module orig evil	Changement de la fonction d'initialisation
kmodule_load	module	Chargment d'un module
kmodule_unload	module	Déchargement d'un module
ksym	symbol	Récupération de l'adresse d'un symbole
kvirtm_loadme	module	Chargement du module d'accès mémoire automatiquement
kvirtm_read_pid	pid adresse taille	Lecture dans la mémoire virtuelle d'un processus
kvirtm_write_pid	pid adresse buffer	écriture dans la mémoire virtuelle d'un processus
kvirtm_disasm_pid	pid adresse taille	Désassemblage de la mémoire virtuelle d'un processus
kvirtm_task_pid	pid	Récupération de champs dans la task_struct d'un processus
kdump_get_vma	pid	Récupération des VMA d'un processus
kdump_vma	pid	Dump sur le disque des VMA d'un processus
reladd	ws(.o) ws(mem)	Injection de nouveau source code dans la mémoire noyau
redir	hooked hooking	Redirige une fonction vers une autre fonction

3.4 Inconvénients de *Kernsh*

Malgré ses points positifs, *Kernsh* a des côtés négatifs. À chaque commande lancée, on doit changer de contexte, passer du contexte utilisateur en contexte noyau, et donc cela entraîne des problèmes de performances, de fiabilitées de l'information récupérée et peut également provoqué des bugs en multiprocesseur. C'est pourquoi nous avons développés un autre outil d'analyse noyau, corrigeant ces points.

4 Debugging du noyau Linux : *Ke2dbg*

Contrairement à *kernsh* qui est un outil s'exécutant entièrement en contexte *utilisateur*, un debugger noyau s'exécute totalement en contexte *superviseur*. Dans le cadre ERESI, cela signifie que l'interpréteur du langage ERESI est entierement mappé dans le noyau. Cette technique est

complémentaire a celle de *kernsh* : elle est plus performante car aucun changement de contexte n'est nécessaire (puisque tout s'exécute en contexte noyau). Cependant, elle est plus intrusive (et potentiellement plus instable), car elle nécessite l'injection de nouveau code dans le noyau. Sans précaution, un debugger noyau peut s'avérer inutilisable pour certaines tâches. Par exemple, si le but d'un tel debugger est d'analyser la distribution des allocations mémoire du noyau, la technique d'*allocation proxying* doit nécessairement être portée au debugger noyau, pour que les allocations du debugger n'affectent pas les allocations du noyau.

4.1 État de l'art en debug noyau

Un Debugger noyau doit fournir des moyens d'examiner la mémoire et les structures de données du noyau pendant que le système est en fonctionnement. Il doit également permettre de facilement ajouter, d'afficher des données (pile de la tâche courant, désassemblage) , des structures du système, d'avoir un contrôle complet sur ses opérations, stopper sur une instruction spécifique ou bien sur une modification de la mémoire, d'un registre, etc... Les *Kernel Hackers* sous UNIX ont la possibilité d'utiliser 3 types de débugger noyau :

- Debugging noyau depuis les sources (kdg [9], kgdb [16])
- Debugging noyau distant (kgdb [16])
- Debugging noyau par un module (rr0d [2])

Les deux premiers types de debugging nécessitent de patcher les sources du noyau pour pouvoir analyser le système durant son exécution. Le second type permet en plus d'effectuer le debugging à distance par le port série ou ethernet (comme par *gdb*). Il a pour désavantage de devoir utiliser une seconde machine en plus de la machine soumise à l'analyse. Le dernier type quand à une approche plus intéressante car il ne nécessite pas de patcher les sources du noyau, et s'utilise comme un *LKM*, et supporte plusieurs systèmes d'exploitations (Linux, *BSD, Windows). C'est sur *rr0d* que l'architecture de *ke2dbg* est calquée.

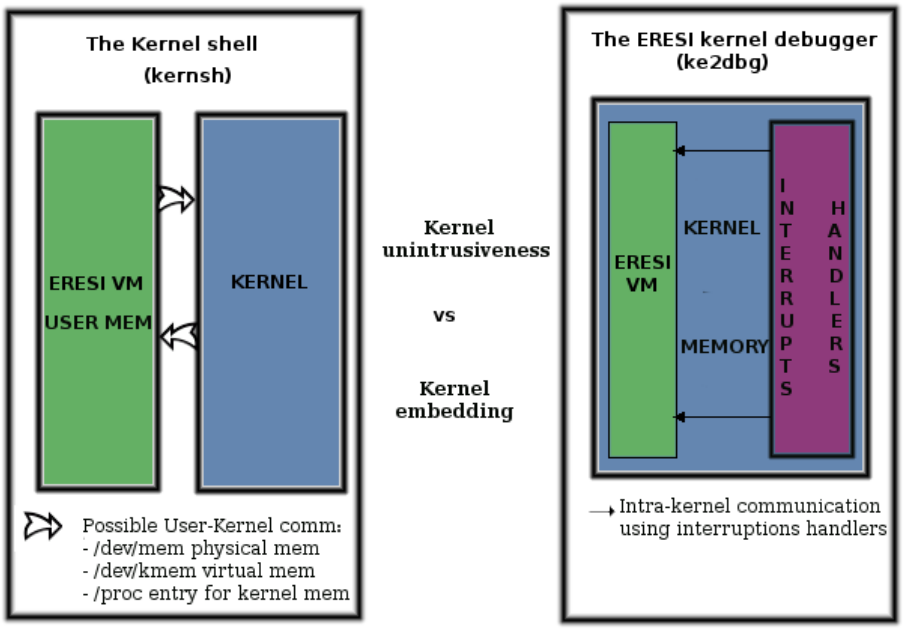
Notre cible ne concerne que les systèmes Linux et BSD car les autres OS disposent déjà de debuggers noyau très intéressant :

- Le debugger *Windbg* [5] de *Microsoft* pour le systeme d'exploitation *Windows*.
- Le debugger *Kmdb* [13] de *Sun Microsystem* pour le systeme d'exploitation *Solaris*.

Ke2dbg est cependant le premier debugger noyau capable d'interpréter des scripts ERESI.

4.2 Description de *Ke2dbg*

Les differences entre *Kernsh* et *Ke2dbg* sont exprimées par la figure suivante :



Ke2dbg est un debugger modulaire à la *rr0d*, car il ne nécessite pas de patcher les sources du noyau, se lançant donc comme une module noyau. Il ne requiert pas également une autre machine pour debugger. Le code réside entièrement en espace noyau, et ne nécessite donc aucune communication avec l'espace utilisateur. l'intéraction avec ce dernier est effectuée avec une prise en charge basique du clavier et de l'écran.

Le principal intéret de *Ke2dbg* est d'intégrer dans son code toutes les fonctionnalités vu précédemment. En effet il est lié avec toutes les principales bibliothèques du cadre, et peut donc bénéficier de tout son support. On peut donc par exemple directement écrire des scripts en language *ERESI* pour débugger le noyau, ou bien encore bénéficier du code noyau de *Libkernsh* pour effecter des opérations sur la mémoire des processus.

5 Travaux futurs

Dans les évolutions à venir pour la partie noyau de *ERESI*, nous envisageons les extensions suivantes :

- Support du SMP pour les parties noyau, permettant ainsi l'analyse des problèmes de races conditions que nous avons sur ces systemes.
- Interface de *Libmjollnir* pour le noyau, permettant ainsi de générer des graphes de flot de controles pour les fonctions du noyau et de vérifier l'intégrité, ou de faire de la prise d'empruntes de telles fonctions.
- Séparer la mémoire de *Ke2dbg* de celle du système d'exploitation (porter notre technique d'*allocation proxying* sur le noyau).
- Porter nos outils noyau vers d'autres systèmes d'exploitation (notamment *BSD*).
- Rendre ERESI utilisable sur les processeurs *AMD64*.

6 Conclusion

Nous fournissons ici un cadre alternatif complet permettant de travailler dans des environnements hostiles, aussi bien sur des programmes, des processus, que sur des systèmes d'exploitations entiers. Notre implémentation inclue le premier langage dédié à l'analyse statique ou dynamique en reverse engineering et forensics. Elle met á disposition un interpréteur de commandes, un moteur de désassemblage, une bibliothèque de manipulation de binaire, une libraire de prise d'empreintes, de *debugging* et *tracing* embarquée, d'analyse de format de debug, et maintenant s'enrichit donc d'une libraire de manipulation du noyau, mais également d'un analyseur noyau. Nous donnons en annexe un exemple d'utilisation de cet outil.

Références

1. Silvio Cesare. http ://www.phiral.net/other/runtime-kernel-kmem-patching.txt. Runtime Kernel Kmem Patching.
2. Droids Corporation. http ://rr0d.droids-corp.org/. RR0D : Rasta Ring 0 Debugger.
3. Duverger. http ://actes.sstic.org/sstic07/exploitation_espace_noyau/sstic07-article-duverger-exploitation_espace_noyau.pdf. Exploitation espace noyau.
4. jbtzhm. http ://phrack.org/issues.html ?issue=60&id=8#article. Static Kernel Patching.
5. Microsoft. http ://www.microsoft.com/whdc/devtools/debugging/default.mspx. Windbg.
6. Samuel Dralet et Kstat Nicolas Brito. http ://www.kernsh.org/old/. Ancien Kernsh.
7. Plaguez. http ://www.phrack.org/issues.html ?issue=52&id=18#article. Weakening the Linux Kernel.
8. sd and devik. http ://www.phrack.org/issues.html ?issue=58&id=7#article. Linux on-the-fly kernel patching without LKM.
9. SGI. http ://oss.sgi.com/projects/kdb/. KDB.
10. sgrakkyu and twiz. http ://www.phrack.org/issues.html ?issue=64&id=6#article. Attacking the Core : Kernel Exploiting Notes.
11. The ELF shell crew. http ://phrack.org/issues.html ?issue=63&id=9#article. Embedded ELF Debugging : the middle head of Cerberus.
12. Spender. http ://www.grsecurity.net.
13. Sun. http ://www.opensolaris.org/os/community/mdb/architecture/kmdb_overview.html. Kmdb.
14. Eresi Team. http ://s.eresi-project.org/inc/articles/bheu-eresi-article-2007.pdf. Next Generation Debuggers for Reverse Engineering.
15. The ERESI team. http ://www.eresi-project.org. The ERESI Reverse Engineering Software Interface.
16. LynSysSoft Technologies. http ://kgdb.linsyssoft.com/. KGDB.
17. Julien Vanegue. http ://phrack.org/issues.html ?issue=61&id=8#article. The Cerberus ELF Interface.
18. Julien Vanegue and Sebastien Roy. http ://actes.sstic.org/sstic03/reverse_intel_elf/sstic03-art-vanegue_roy-reverse_intel_elf.pdf. Reverse engineering des systemes ELF/INTEL.

A Annexes

```
eresi:/# cat lkm-sys_setdomainname.c
[...]
int asmlinkage new_sys_setdomainname(const char *name, size_t len)
{
        printk("NEW SYS DOMAINNAME\n");
        return old_sys_setdomainname(name, len);
}
[...]

(kernsh-0.81-a8-dev@local) load kernsh/lkm-sys_setdomainname.ko
 [*] Tue Apr  1 19:58:47 2008 - New object loaded :
kernsh/lkm-sys_setdomainname.ko

(kernsh-0.81-a8-dev@local) mode dynamic
[*] kernsh is now in DYNAMIC mode

(kernsh-0.81-a8-dev@local) reladd 1 2
 [*] ET_REL kernsh/lkm-sys_setdomainname.ko injected
succesfully in ET_EXEC /tmp/vmlinux

(kernsh-0.81-a8-dev@local)

eresi:/# ./test_sys_setdomainname
RET 0
eresi:/# dmesg | tail -1
....

(kernsh-0.81-a8-dev@local) redir sys_setdomainname new_sys_setdomainname
 [*] Function sys_setdomainname redirected to addr 0xCFD981E4 <new_sys_setdomainname>

(kernsh-0.81-a8-dev@local) redir
[00] TYPE:CFLOW  [C012F237] <sys_setdomainname> redirected
on [CFD981E4] <new_sys_setdomainname>


eresi:/# ./test_sys_setdomainname
RET 0
eresi:/# dmesg | tail -1
NEW SYS DOMAINNAME
```

```
(kernsh-0.81-a8-dev@local) kvirtm_loadme libkernsh/modules/kernsh-virtm-linux.ko
Loading libkernsh/modules/kernsh-virtm-linux.ko with : /sbin/insmod
libkernsh/modules/kernsh-virtm-linux.ko hijack_sct=1
sct_value=0xc02aa440 free_syscall=31
(kernsh-0.81-a8-dev@local) kdump_get_vma 1
Vma for pid 1 is in list vma_1
(kernsh-0.81-a8-dev@local) kvirtm_read_pid 1 0x8048000%100

Reading pid:1 @ 0x08048000 strlen(100)
0x8048000 | 7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00 | .ELF............
0x8048010 | 02 00 03 00 01 00 00 00  30 9a 04 08 34 00 00 00 | ........0...4...
0x8048020 | d0 75 00 00 00 00 00 00  34 00 20 00 07 00 28 00 | .u......4. ...(.
0x8048030 | 1a 00 19 00 06 00 00 00  34 00 00 00 34 80 04 08 | ........4...4...
0x8048040 | 34 80 04 08 e0 00 00 00  e0 00 00 00 05 00 00 00 | 4...............
0x8048050 | 04 00 00 00 03 00 00 00  14 01 00 00 14 81 04 08 | ................
0x8048060 | 14 81 04 08                                     | ....
(kernsh-0.81-a8-dev@local) kvirtm_read_pid 1 0x08050000%100

Reading pid:1 @ 0x08050000 strlen(100)
0x8050000 | 00 00 00 00 49 00 00 00  2f 65 74 63 2f 73 65 6c | ....I.../etc/sel
0x8050010 | 69 6e 75 78 2f 72 65 66  70 6f 6c 69 63 79 2d 74 | inux/refpolicy-t
0x8050020 | 61 72 67 65 74 65 64 2f  63 6f 6e 74 65 78 74 73 | argeted/contexts
0x8050030 | 2f 66 69 6c 65 73 2f 66  69 6c 65 5f 63 6f 6e 74 | /files/file_cont
0x8050040 | 65 78 74 73 00 00 00 00  00 00 00 00 21 00 00 00 | exts........!...
0x8050050 | 58 01 05 08 04 00 00 00  57 45 53 54 00 00 00 00 | X.......WEST....
0x8050060 | ac 00 05 08                                     | ....

(kernsh-0.81-a8-dev@local) kdump_get_vma 2485
vma_2485[0][vma_2485_0] [vm_start] => [0x8048000] [vm_end] => [0x80c5000]
vma_2485[1][vma_2485_1] [vm_start] => [0x80c5000] [vm_end] => [0x80ef000]
vma_2485[2][vma_2485_2] [vm_start] => [0xb7ead000] [vm_end] => [0xb7f43000]
vma_2485[3][vma_2485_3] [vm_start] => [0xbfaba000] [vm_end] => [0xbfad0000]
vma_2485[4][vma_2485_4] [vm_start] => [0xffffe000] [vm_end] => [0xfffff000]
Vma for pid 2485 is in list vma_2485

(kernsh-0.81-a8-dev@local) kvirtm_disasm_pid 2485 0xb7ead000%1000
Disasambling pid:2485 @ 0xB7EAD000 strlen(1000)
[...]
0xB7EAD168                    sub           $0x18,%esp 83 EC 18
0xB7EAD16B                    mov           (%esi),%edx 8B 16
0xB7EAD16D                    mov           %edx,(%esp,1) 89 14 24
0xB7EAD170                    mov           0x4(%esi),%edx 8B 56 04
0xB7EAD173                    mov           %edx,0x4(%esp,1) 89 54 24 04
0xB7EAD177                    mov           0x8(%esi),%edx 8B 56 08
0xB7EAD17A                    mov           %edx,0x8(%esp,1) 89 54 24 08
0xB7EAD17E                    mov           0x14(%esi),%edx 8B 56 14
0xB7EAD181                    mov           %edx,0xc(%esp,1) 89 54 24 0C
0xB7EAD185                    mov           $0xffffffff,0x10(%esp,1) C7 44 24 10
FF FF FF FF
0xB7EAD18D                    mov           $0x0,0x14(%esp,1) C7 44 24 14 00 00 00 00
0xB7EAD195                    mov           %esp,%ebx 89 E3
0xB7EAD197                    mov           $0x5a,%eax B8 5A 00 00 00
0xB7EAD19C                    int             $0xffffff80 CD 80
0xB7EAD19E                    add           $0x18,%esp 83 C4 18
0xB7EAD1A1                    mov           %eax,%ebx 89 C3
0xB7EAD1A3                    mov           $0x7d,%eax B8 7D 00 00 00
0xB7EAD1A8                    mov           0x4(%esi),%ecx 8B 4E 04
0xB7EAD1AB                   mov           0x8(%esi),%edx 8B 56 08
0xB7EAD1AE                    int             $0xffffff80 CD 80
0xB7EAD1B0                    jmp           0xb7ead07a EB 78
[...]
```