

De l'invisibilité des rootkits : application sous Linux

Eric Lacombe¹ Frédéric Raynal² Vincent Nicomette³

¹eric.lacombe@{laas.fr,security-labs.org}
LAAS-CNRS - EADS-CCR

²frederic.raynal@security-labs.org
Sogeti ESEC - MISC Magazine

³nicomett@laas.fr
LAAS-CNRS

31 mai 2007



Introduction

Définition de *rootkit*

Ensemble de modifications permettant à un attaquant de *maintenir dans le temps un contrôle frauduleux* sur un système informatique.

Plan

- 1 Architecture et élaboration d'un rootkit
 - Évolution des rootkits
 - Architecture fonctionnelle d'un rootkit
- 2 Exemple de construction d'un rootkit "furtif"
 - Principe du vecteur d'interaction avec le noyau
 - Installation du rootkit
 - Dissimulation de l'activité système
- 3 Conclusion

Plan

- 1 Architecture et élaboration d'un rootkit
 - Évolution des rootkits
 - Architecture fonctionnelle d'un rootkit
- 2 Exemple de construction d'un rootkit "furtif"
 - Principe du vecteur d'interaction avec le noyau
 - Installation du rootkit
 - Dissimulation de l'activité système
- 3 Conclusion

Évolution des rootkits

Évolution des rootkits

Jusqu'alors : démarche de factorisation afin de toucher le plus de monde avec un minimum de modifications.

- D'abord modification des commandes du système.
- Ensuite factorisation en espace utilisateur via la modification de DSO - Dynamic Shared Object, compilateur, etc.
- Enfin au sein de l'entité partagé par tous les éléments du système : le noyau.

Maintenant : stade d'externalisation du code malicieux.

- Les *rootkit-hyperviseurs* se place en dessous du noyau en exploitant les technologies de virtualisation.

Évolution des rootkits

Évolution des rootkits

Jusqu'alors : démarche de factorisation afin de toucher le plus de monde avec un minimum de modifications.

- D'abord modification des commandes du système.
- Ensuite factorisation en espace utilisateur via la modification de DSO - Dynamic Shared Object, compilateur, etc.
- **Enfin au sein de l'entité partagé par tous les éléments du système : le noyau.**

Maintenant : stade d'externalisation du code malicieux.

- Les *rootkit-hyperviseurs* se place en dessous du noyau en exploitant les technologies de virtualisation.

Plan

- 1 Architecture et élaboration d'un rootkit
 - Évolution des rootkits
 - Architecture fonctionnelle d'un rootkit
- 2 Exemple de construction d'un rootkit "furtif"
 - Principe du vecteur d'interaction avec le noyau
 - Installation du rootkit
 - Dissimulation de l'activité système
- 3 Conclusion

Architecture fonctionnelle d'un rootkit

Quatre éléments fonctionnels

- Un **injecteur** : installer le rootkit.
- Un **module de protection** : protéger le rootkit et les activités de l'attaquant.
- Une **backdoor** : communiquer avec le rootkit.
- Des **services** : effectuer des opérations malicieuses.

Définitions des éléments fonctionnels d'un rootkit (1/2)

Un injecteur

- Mécanisme que l'attaquant emploie afin d'insérer le rootkit dans le système.
- Ne sert qu'une fois lors de l'installation du rootkit.

Un module de protection

Objectif : rendre le rootkit "tenace" sur le système tout le temps nécessaire à l'attaquant.

Stratégies envisageables :

- *Dissimulation* : cacher la présence du rootkit sur le système (parasitage de processus, etc.) et l'activité de l'attaquant.
- *Résistance* : faire en sorte que le rootkit ne puisse être enlevé même s'il est découvert (chiffrement de données critiques pour le bon comportement du système).
- *Persistance* : survie au redémarrage du système (i.e. dissimulation dans des mémoires non-volatiles comme dans `/boot/vmlinuz` sur le disque).

Définitions des éléments fonctionnels d'un rootkit (2/2)

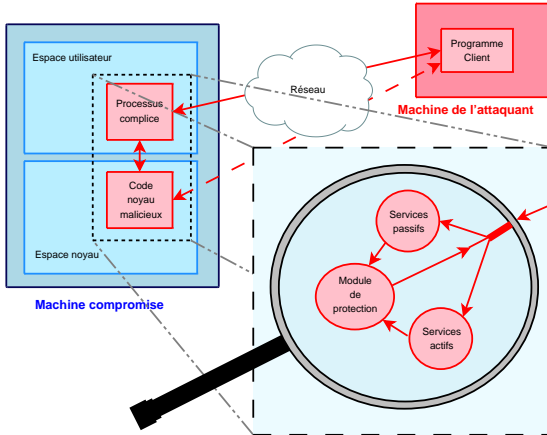
Une backdoor

- **Rôles :**
 - Medium de communication entre l'attaquant et le système compromis.
 - Accéder aux services fournis par le rootkit.
- **Vecteurs d'interaction :**
 - Entre l'attaquant et le système compromis.
 - Entre le système compromis et les services du rootkit.

Des services

- **Services passifs** (espionnage) : keylogger, écoute réseau, etc.
- **Services actifs** (modifie perceptiblement le comportement du système) : lancement d'un DoS, suppression de fichiers, déroutage des ressources de calcul du système, etc.

Représentation graphique de l'architecture fonctionnelle d'un rootkit



Plan

- 1 Architecture et élaboration d'un rootkit
 - Évolution des rootkits
 - Architecture fonctionnelle d'un rootkit
- 2 Exemple de construction d'un rootkit "furtif"
 - Principe du vecteur d'interaction avec le noyau
 - Installation du rootkit
 - Dissimulation de l'activité système
- 3 Conclusion

Fonctionnement des appels système sous Linux

```
system_call:
```

```
...
```

```
call * sys_call_table (, %eax, 4)
```

```
...
```

offset dans
la table

@sys_restart_syscall
@sys_exit
@sys_fork
@sys_read
@sys_write

Fonctionnement

► Depuis le mode utilisateur :

- Préparation de la requête.
- `eax` contient le numéro de l'appel système.
- Autres registres : contiennent les paramètres.

► On bascule en mode noyau :

- Levée de l'interruption `0x80`.
- Autre technique : l'instruction `sysenter`.

Fonctionnement et détournement de l'appel système 0

SCT (SysCall Table)

@sys_restart_syscall

sys_restart_syscall ()

```
{  
  ...  
  restart = &current_ti() -> restart_block;  
  return restart->fn(restart);  
}
```

thread_info
(structure propre à
chaque processus)

restart_block

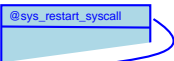
long (*fn)(restart_block *)
int arg0, arg1, ...

L'appel système 0

- Point d'entrée vers l'espace noyau.
- **Rôle** : employé par le noyau pour reprendre certains appels système interrompus (propre au processus interrompu).
- **Fonctionnement** : prend en arguments l'adresse de l'appel système à relancer et ses paramètres, ajustés si besoin.

Fonctionnement et détournement de l'appel système 0

SCT (SysCall Table)



```
sys_restart_syscall ()
```

```
{  
  ...  
  restart = &current_ti() -> restart_block;  
  return restart->fn(restart);  
}
```

thread_info
(structure propre à
chaque processus)



ON CHANGE CETTE ADRESSE

L'appel système 0

- Point d'entrée vers l'espace noyau.
- **Rôle** : employé par le noyau pour reprendre certains appels système interrompus (propre au processus interrompu).
- **Fonctionnement** : prend en arguments l'adresse de l'appel système à relancer et ses paramètres, ajustés si besoin.

Design de notre rootkit, fondé sur le détournement de l'appel système 0

La backdoor en espace noyau : minimisation des modifications

- Modification de la sémantique de l'appel système 0 pour appeler n'importe quelle fonction située en espace noyau.
- Permet l'injection de code arbitraire si une fonctionnalité n'est pas directement fournie par le noyau.

La backdoor en espace utilisateur : le point d'entrée dans le noyau

- Le processus complice : un mécanisme de relais via l'appel système 0.

Le programme client : siège de la logique de l'attaquant

- Construction des opérations de l'attaquant à partir des fonctions noyau et du code injecté.

Plan

- 1 Architecture et élaboration d'un rootkit
 - Évolution des rootkits
 - Architecture fonctionnelle d'un rootkit
- 2 Exemple de construction d'un rootkit "furtif"
 - Principe du vecteur d'interaction avec le noyau
 - **Installation du rootkit**
 - Dissimulation de l'activité système
- 3 Conclusion

L'injecteur du rootkit

Description

► Installation d'un *bootstrap* :

- 1 Recherche de l'adresse de la fonction noyau `get_page` par *pattern matching* sur `/dev/kmem`.
- 2 Injection au fond de la pile noyau d'un bootstrap `B` qui appelle cette fonction.
- 3 Appel de `B` via l'appel système 0 depuis l'espace utilisateur et obtention d'une page mémoire `P`.

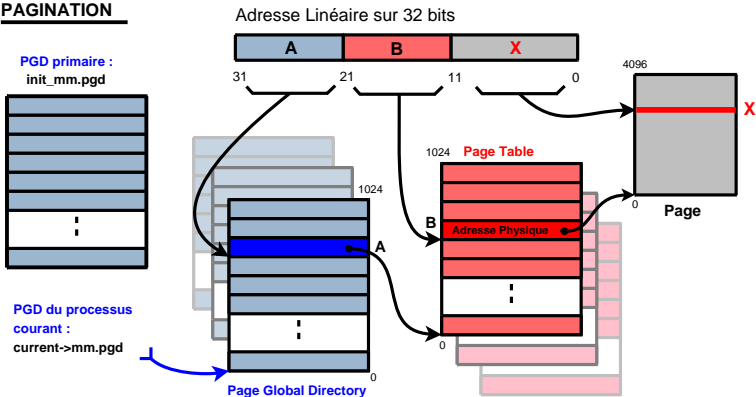
► Installation de la *backdoor* BK en espace noyau dans `P` :

- 1 BK est l'interface entre les fonctions du noyau et l'appel système 0 servant à les exécuter.
 - Nouvelle *sémantique* de l'appel système 0 : les registres `eax`, `ebx`, etc. contiennent l'adresse de la fonction noyau à exécuter et ses paramètres.
- 2 Écriture dans le `thread_info` du processus complice de l'adresse de BK dans le champ employé par l'appel système 0.

Dissimulation de la backdoor noyau du rootkit (1/2)

Rappels

PAGINATION



L'allocateur mémoire non-contiguë VMALLOC

- VMALLOC ne modifie que la PGD primaire.
- Mise à jour de la PGD d'un processus lors de l'accès à la zone allouée.

Plan

- 1 Architecture et élaboration d'un rootkit
 - Évolution des rootkits
 - Architecture fonctionnelle d'un rootkit
- 2 Exemple de construction d'un rootkit "furtif"
 - Principe du vecteur d'interaction avec le noyau
 - Installation du rootkit
 - Dissimulation de l'activité système
- 3 Conclusion

Dissimulation de processus (1/2)

Méthodes

Camoufler son descripteur

- Un processus est caractérisé par une structure `task_struct`.
- Ce descripteur est lié de multiples façons avec ces pairs et certains sous-systèmes du noyau (mécanismes des signaux, ptrace, etc.).
- Pour le camoufler : on l'enlève de toutes les listes le référençant afin que le noyau "nie" son existence.
- Deux approches pour le délier :
 - *A priori* : duplication de `do_fork` et suppression des opérations de liaison.
 - *A posteriori* : suppression de ses liens après sa création.
- **Attention : Il doit rester dans les listes de l'ordonnanceur (au moins temporairement) pour s'exécuter.**

Dissimulation de processus (2/2)

Observations

Problème

- Le descripteur de processus est toujours présent en mémoire.
- On peut facilement le trouver en parcourant la mémoire physique.

Solution

- Le parasitage mobile : le vol de cycles d'exécution.

Exemples d'utilisation

- Exécution de codes malicieux sans création de `task_struct`.
- Conception d'une backdoor originale (cf. l'article pour plus de détails).

Le parasitage mobile

Objectifs

Objectifs

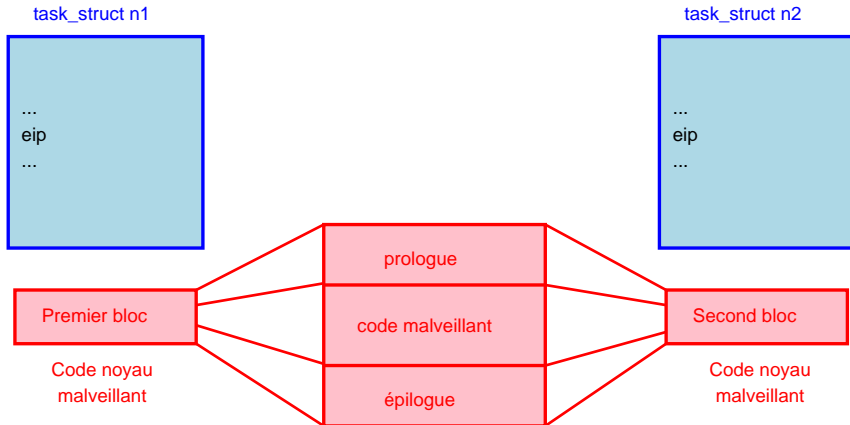
- Exécuter du code par l'intermédiaire de threads noyau.
- Ne pas altérer le travail de ces threads.

Rappel

À chaque changement de contexte, l'ordonnanceur sauvegarde dans le descripteur du processus interrompu la valeur de son compteur d'instructions.

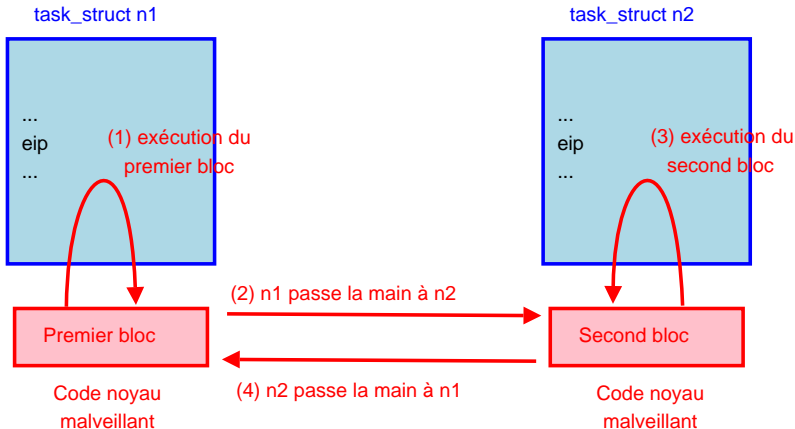
Parasitage mobile dans le cas de deux threads noyau (1/2)

Disposition



Parasitage mobile dans le cas de deux threads noyau (2/2)

Algorithme



Limites et Contributions

Limites

- ▶ **Au niveau du vecteur d'interaction avec le noyau :**
 - Comportement *a priori* étrange du processus complice : exécution de nombreux appels système 0.
 - Forte dépendance du programme client avec l'API interne du noyau.
- ▶ **Au niveau des techniques de dissimulation :**
 - Camouflage via VMALLOC ne résiste pas à une lecture complète de la mémoire physique.
 - Parasitage mobile :
 - Difficulté d'implémentation de la charge malicieuse.
 - Survie dépendante des processus infestés

Limites et Contributions

Contributions

▶ Au niveau du vecteur d'interaction avec le noyau :

- Visibilité *locale* des modifications effectuées sur le système.
- Tirer parti au maximum des fonctionnalités du noyau pour effectuer nos opérations.
- Construction des opérations de l'attaquant, repoussée à l'extérieur du système compromis.

▶ Au niveau des techniques de dissimulation :

- Mise à profit des caractéristiques d'implémentation des mécanismes du noyau (ex : VMALLOC).
- Algorithme de parasitage mobile ne perturbe que temporairement le comportement des threads infestés.

Et maintenant ...

... place aux questions (et au café ;)