

Projet Rootkit

Dieu-Donne Alatin
Nikola Ostojic
Kevin Pons

Contexte

- Dans le cadre de comprendre plus le fonctionnement du kernel après le cours
- Rootkit = un type de logiciel capable de maintenir un accès privilégié tout en cachant sa présence
- Plusieurs types: UM (LD PRELOAD), KM (module), FW based (UEFI) etc.
- Ici KM sur un système Linux, dépend fortement des versions

Cahier des charges

- LFS qcow2 prête à utilisation avec potentiellement le rootkit embarqué
- Rapport avec les fonctionnalités et méthodes choisies
- Instructions et commandes
- Codes sources et login des machines

Fonctionnalités

Rootkit LKM avec diverses fonctionnalités:

- Acquérir un shell root
- Dissimulation des fichiers et répertoires à base d'un préfix
- Cacher des parties d'un fichier sur la base d'un tag
- Empêcher la suppression de fichiers cachés (dans le cas où ils sont découverts)
- Backdorer le système (partiellement fait)
- Cacher des processus, cacher le module
- Cacher les ports ouverts et contrôler le flux réseau pour dissimuler des packets (non abouti)
- Contrôler les logs kernel afin de dissimuler nos traces
- Anti reverse

Techniques choisies

- Plusieurs existes (comme réécriture du registre CR0, ftrace)
- On peut hooker des appels systèmes, des méthodes virtuel d'opération (files_operations, symbols, des callbacks de routine de notification)
- Hook des symboles via **ftrace**
- Utilisation des kernels probes pour retrouver les symboles à résoudre via ftrace

Cacher des processus

- Requiert la gestion du cycle de vie du processus, sa visibilité et la gestion de sa consommation CPU

A chaque processus créé est associé une structure task avec des flags

On attache un attribut via les flags non utilisés au processus et à base on prend une décision

```
/*  
 * Per process flags  
 */  
#define PF_VCPU 0x00000001 /* I'm a virtual CPU */  
#define PF_IDLE 0x00000002 /* I am an IDLE thread */  
#define PF_EXITING 0x00000004 /* Getting shut down */  
#define PF_IO_WORKER 0x00000010 /* Task is an IO worker */  
#define PF_WQ_WORKER 0x00000020 /* I'm a workqueue worker */  
#define PF_FORKNOEXEC 0x00000040 /* Forked but didn't exec */  
#define PF_MCE_PROCESS 0x00000080 /* Process policy on mce errors */  
#define PF_SUPERPRIV 0x00000100 /* Used super-user privileges */  
#define PF_DUMPCORE 0x00000200 /* Dumped core */  
#define PF_SIGNALED 0x00000400 /* Killed by a signal */  
#define PF_MEMALLOC 0x00000800 /* Allocating memory */  
#define PF_NPROC_EXCEEDED 0x00001000 /* set_user() noticed that RLIMIT_NPROC was exceeded */  
#define PF_USED_MATH 0x00002000 /* If unset the fpu must be initialized before use */  
#define PF_NOFREEZE 0x00008000 /* This thread should not be frozen */  
#define PF_FROZEN 0x00010000 /* Frozen for system suspend */  
#define PF_KSWAPD 0x00020000 /* I am kswapd */  
#define PF_MEMALLOC_NOFS 0x00040000 /* All allocation requests will inherit GFP_NOFS */  
#define PF_MEMALLOC_NOIO 0x00080000 /* All allocation requests will inherit GFP_NOIO */  
#define PF_LOCAL_THROTTLE 0x00100000 /* Throttle writes only against the bdi I write to,  
 * I am cleaning dirty pages from some other bdi. */  
#define PF_KTHREAD 0x00200000 /* I am a kernel thread */  
#define PF_RANDOMIZE 0x00400000 /* Randomize virtual address space */  
#define PF_SWAPWRITE 0x00800000 /* Allowed to write to swap */  
#define PF_NO_SETAFFINITY 0x00000000 /* Userland is not allowed to meddle with cpus_mask */  
#define PF_MCE_EARLY 0x00800000 /* Early kill for mce process policy */  
#define PF_MEMALLOC_PIN 0x10000000 /* Allocation context constrained to zones which allow long term pinning. */  
#define PF_FREEZER_SKIP 0x40000000 /* Freezer should not count it as freezable */  
#define PF_SUSPEND_TASK 0x80000000 /* This thread called freeze_processes() and should not be frozen */
```

Cacher des processus

- Hooks `copy_creds()` pour attacher des attributs au moment d'un `fork()`
- Hooks `exit_creds()` pour detacher l'attribut
- Hooks `find_task_by_vpid()` utilisé par la plupart des fonctions qui requete le task associé à un PID , ainsi on filtre les task via les attributs
- Hooks `tgid_iter()` pour filter les répertoires `/proc/<PID>` des processus qui doivent rester cachés
- Hooks `vfs_getattr()` (utilisé pour retrouver le PID via par exemple le fichier en cours d'exécution)
- Hooks `sys_kill()` pour éviter que le processus soit tué
- Hooks `account_process_tick()` pour filtrer la consommation CPU

Cacher des fichiers et répertoires

- Hooks de fonctions très bas niveau (en regardant par exemple à quelle fonction getdents fait appel etc.)

```
#ifdef CONFIG_HIDE_DIR_AND_FILES
    HOOK("filldir", rk_filldir, &orig_filldir),
    HOOK("filldir64", rk_filldir64, &orig_filldir64),
    HOOK("fillonedir", rk_fillonedir, &orig_fillonedir),
    HOOK("compat_fillonedir", rk_compat_fillonedir, &orig_compat_fillonedir),
    HOOK("compat_filldir", rk_compat_filldir, &orig_compat_filldir),
    HOOK("d_lookup", rk_d_lookup, &orig_d_lookup),
    HOOK("__d_lookup", rk__d_lookup, &orig__d_lookup),
    HOOK("user_path_at_empty", rk_user_path_at_empty, &orig_user_path_at_empty),
#endif
```


Monitorer les logs kernels

- Audit, utilisé par le kernel pour auditer des appels systèmes
- Hooks `audit_alloc()` pour supprimer l'auditing

On peut également récupérer des logs via dmesg, syslog

- Hook `do_syslog()`, filter les logs relatifs au rootkit
- Hook `devkmsg_read()` qui est utilisé pour lire les logs kernels (`dmesg` lit par exemple depuis `/dev/kmsg` ou `/proc/kmsg`), a permis de filter même le premier message qui indique qu'un module vient d'être insérer

Cacher des parties d'un fichier

- Hooks de `vfs_read()` utilisé en interne par l'appel `read`

Communication avec le programme companion

- Communique via `/dev/zero` qui est un device qui normalement n'accepte que des lectures (il renvoie autant de null bytes que voulu), écrire dans ce device revient à écrire dans `/dev/null`
- Hooks de sa fonction d'écriture qui est encore celui de `/dev/null` (`write_zero = write_null ()`)

Obtenir un shell root

- Classique `commit_creds(prepare_kernel_cred(NULL));`

Reverse shell

- Un script est lancé à l'initialisation du module, on peut y mettre ce qu'on veut à exécuter à l'initialisation (on a pas pu testé, qemu ne marchait pas avec le réseau)

Anti-reverse engineering

- Notre module est chiffré (un simple `XOR + ROL`) puis embarqué dans un autre module qui s'occupera de le lancer via `sys_init_module()`
- Ensuite le module obtenu est à nouveau chiffré puis embarqué dans un binaire qu'il sera possible de lancer depuis le userland
- Possibilité de lancer des modules depuis le userland via la fonction `init_module()`

Demo time

Bilan

- Fonctionnement du kernel linux et possibilités
- Compréhension accrue du fonctionnement des makefiles
- Fonctionnement de Kconfig
- Build de LFS

Perspectives ?

- Gérer correctement le réseau
- Ajouter de nouvelles fonctionnalités plus sophistiqués (bypass de SELinux, App Armor chroot etc), pivoting dans le réseau si possible, récupération de creds

Merci

@Rootkit2600