

MASTER 1 CRYPTIS - INFORMATIQUE
FACULTÉ DES SCIENCES ET TECHNIQUES

Projet Audit et Sécurité Réseau

COMMUNICATION IP PAR UTILISATION DE CANAL CACHÉ SÉCURISÉ

Encadré par :
Pr. Pierre-François BONNEFOI
Pr. Emmanuel CONCHON

Réalisé par :
Lansana DIAKITE
Ruslan KASHEPAROV

Année Universitaire 2023 - 2024

Table des matières

1	Introduction	2
2	Présentation théorique des canaux cachés et des techniques de dissimulation	2
3	Contraintes et choix techniques	3
3.1	Contraintes techniques	3
3.2	Choix techniques	3
4	Implémentation Pratique	3
4.1	Configuration du Réseau	3
4.2	Création des Interfaces veth	4
4.3	Configuration IPTables	5
4.4	Traitement des Paquets avec Scapy	7
5	Test du canal caché	10
5.1	Avec Socat et le mode de transport UDP	10
5.2	Mode de Transport TCP : Établissement de la Connexion	12
6	Sécurisation du tunnel	17
6.1	Sécurisation par l'obfuscation des données	17
6.2	Chiffrement via AES	17
7	Conclusion	19

1 Introduction

L'objectif principal de ce projet est de développer un outil permettant de communiquer entre deux machines au travers d'un « *canal caché* », c'est-à-dire de manière invisible aux outils de surveillance réseau susceptibles d'intercepter les paquets IP utilisés pour cette communication.

La mise en œuvre de ce projet repose sur plusieurs contraintes techniques et choix méthodologiques. Premièrement, le canal caché doit se comporter comme un tunnel, utilisant la couche transport de TCP/IP. Les utilisateurs doivent pouvoir établir ou attendre des échanges basés sur TCP ou UDP en utilisant un outil de communication standard tel que *socat*. Les datagrammes liés à ces communications doivent être transportés de manière sécurisée à travers le canal caché.

Pour réaliser ce tunnel, nous avons utilisé des interfaces Linux de type « *veth* » à deux interfaces liées. Une seule de ces interfaces reçoit une configuration IP et correspond au « *canal* », tandis que l'autre sert uniquement à injecter des paquets vers la première. L'adressage réseau privé a été employé pour les deux extrémités du tunnel, permettant ainsi le routage automatique du trafic.

La protection du contenu des datagrammes est essentielle pour garantir la sécurité de la communication. Nous avons exploré des méthodes d'obfuscation, telles que l'inversion des bits du contenu du datagramme avec un *XOR*, ainsi que le chiffrement avec *AES*, en partageant une clé symétrique entre les interlocuteurs. Le protocole de communication choisi pour dissimuler notre canal caché est *ICMP*, en raison de sa capacité à transporter une « *charge utile* » et de son autorisation de circulation entre les machines des interlocuteurs.

L'outil *Scapy* a été utilisé pour intercepter les paquets à cacher depuis l'interface « *veth* » configurée en IP, encapsuler ces paquets dans des paquets ICMP, protéger ces paquets par chiffrement ou obfuscation, et envoyer les paquets ICMP sur le réseau. L'outil permet également d'intercepter les paquets ICMP sur l'interface réseau de la machine réceptrice, d'extraire les paquets cachés, et de les injecter sur l'interface « *veth* » non configurée en IP.

Ce rapport détaillera les différentes étapes de la mise en œuvre de ce projet, en commençant par une présentation théorique des canaux cachés et des techniques de dissimulation, suivie d'une explication des contraintes et des choix techniques effectués. Nous décrirons ensuite l'implémentation pratique de notre solution, illustrée par des captures d'écran et des extraits de code. Enfin, nous concluons par une évaluation des performances et de la sécurité de notre solution, ainsi que des suggestions pour de futures améliorations.

2 Présentation théorique des canaux cachés et des techniques de dissimulation

Un canal caché, également appelé *covert channel*, est une méthode de communication utilisée pour transférer des informations de manière furtive en exploitant des caractéristiques non prévues d'un système informatique ou d'un protocole de communication. Ces canaux sont souvent utilisés pour contourner les politiques de sécurité et les mécanismes de surveillance, en transmettant des données de manière invisible.

Les techniques de dissimulation peuvent inclure :

- **Tunneling** : Utilisation de protocoles légitimes pour encapsuler les données. Par exemple, encapsuler des données TCP ou UDP dans des paquets ICMP.
- **Obfuscation** : Modification des données pour les rendre difficiles à reconnaître ou à analyser. Une méthode simple est l'inversion des bits des données avec une opération *XOR*.
- **Chiffrement** : Utilisation d'algorithmes de chiffrement pour protéger les données contre les interceptions non autorisées. Dans ce projet, nous avons utilisé l'algorithme de chiffrement symétrique *AES*.

Le choix du protocole ICMP (Internet Control Message Protocol) est particulièrement pertinent pour notre projet car il est largement utilisé pour les diagnostics réseau et peut transporter une « *charge utile* ». En encapsulant les données dans des paquets ICMP, nous pouvons exploiter cette fonctionnalité pour créer un canal caché.

3 Contraintes et choix techniques

La mise en œuvre d'un canal caché sécurisé présente plusieurs défis et nécessite des choix techniques soigneux :

3.1 Contraintes techniques

- **Invisibilité** : Le canal caché doit être indétectable par les outils de surveillance réseau classiques. Cela nécessite de dissimuler les données de manière efficace.
- **Sécurité** : Les données transmises doivent être protégées contre les interceptions non autorisées. Cela inclut l'utilisation de techniques de chiffrement robustes.
- **Compatibilité** : Le canal doit pouvoir transporter divers types de données (TCP, UDP) sans perturber les protocoles de communication existants.

3.2 Choix techniques

- **Utilisation des interfaces veth** : Nous avons utilisé des interfaces Linux de type *veth* à deux interfaces liées. Une interface reçoit une configuration IP pour le *canal*, tandis que l'autre sert à injecter des paquets vers la première.
- **Encapsulation dans ICMP** : Les paquets de données sont encapsulés dans des paquets ICMP, ce qui permet de tirer parti de la capacité de transport de charge utile d'ICMP.
- **Chiffrement AES** : Les données encapsulées sont chiffrées avec AES, utilisant une clé symétrique partagée entre les interlocuteurs pour garantir la confidentialité.
- **Outil Scapy** : Scapy est utilisé pour intercepter, encapsuler, chiffrer et envoyer les paquets ICMP, ainsi que pour recevoir, déchiffrer et réinjecter les paquets de données sur l'interface *veth*.

4 Implémentation Pratique

4.1 Configuration du Réseau

Tout d'abord, nous avons récupéré l'architecture NetLab habituelle qui nous a permis de mettre en place un réseau de test. La configuration de ce réseau est présentée dans l'image ci-dessous :

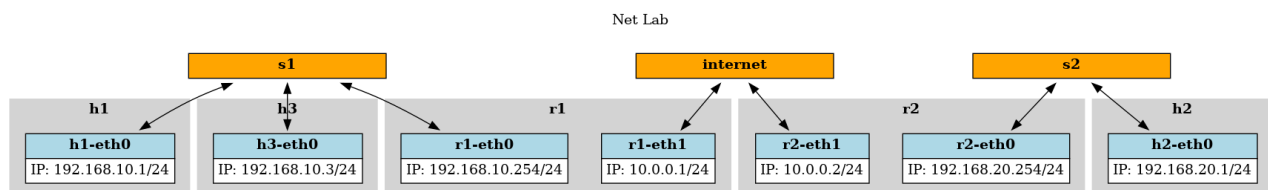


Fig. 1. Image du réseau

Cette configuration inclut plusieurs hôtes et routeurs connectés entre eux, simulant un environnement réseau réaliste.

4.2 Création des Interfaces veth

Ensuite, grâce aux commandes fournies dans le sujet, nous avons créé les interfaces *veth* sur les hôtes *h1* et *h3* (à noter que cela fonctionnera de la même manière entre *h1* et *h2* ou entre *h2* et *h3*). Les interfaces *veth* sont des paires d'interfaces virtuelles qui permettent de relier deux points de terminaison au sein du même système ou de systèmes différents.

Voici le script utilisé pour créer les interfaces *veth* :

```
#!/bin/bash
NETWORK=10.87.87
MACHINE=$1
INTERFACE_ENTREE=injection
INTERFACE_SORTIE=canal
sudo ip l add dev $INTERFACE_ENTREE type veth peer name
    ↪ $INTERFACE_SORTIE
sudo ip a add dev $INTERFACE_SORTIE $NETWORK.$MACHINE/30
sudo ip l set dev $INTERFACE_ENTREE up
```

Ce script configure une paire d'interfaces *veth*, nommées *injection* et *canal*, et leur assigne des adresses IP dans le réseau spécifié.

- **Interface canal** : Cette interface est responsable de la transmission des paquets encapsulés à travers le réseau. Elle est configurée avec une adresse IP pour permettre la communication avec les autres nœuds du réseau.
- **Interface injection** : L'autre interface de la paire *veth* ne reçoit pas de configuration IP et ne sert qu'à injecter des paquets vers l'interface *canal*. Lorsqu'un paquet est injecté dans l'interface *injection*, il est immédiatement transféré à l'interface *canal*, où il peut être traité et envoyé sur le réseau.

Nous avons exécuté ce script sur les hôtes *h1* et *h3* en fournissant des valeurs appropriées pour *MACHINE* afin de configurer les interfaces avec les bonnes adresses IP.

```
ubuntu@ubuntu2004:~/ICMPCanalCache$ [h1] ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: canal@injection: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
   link/ether e6:2a:72:f0:1a:2f brd ff:ff:ff:ff:ff:ff
   inet 10.87.87.1/30 scope global canal
       valid_lft forever preferred_lft forever
   inet6 fe80::e42a:72ff:fef0:1a2f/64 scope link
       valid_lft forever preferred_lft forever
3: injection@canal: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
   link/ether 52:98:c6:d4:10:e7 brd ff:ff:ff:ff:ff:ff
   inet6 fe80::5098:c6ff:fed4:10e7/64 scope link
       valid_lft forever preferred_lft forever
9: h1-eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
   link/ether c6:c6:21:3e:9b:8c brd ff:ff:ff:ff:ff:ff link-netnsid 0
   inet 192.168.10.1/24 scope global h1-eth0
       valid_lft forever preferred_lft forever
   inet6 fe80::c4c6:21ff:fe3e:9b8c/64 scope link
       valid_lft forever preferred_lft forever
```

Fig. 2. Configuration Machine H1

```

ubuntu@ubuntu2004:~/ICMPCanalCache$ [h3] ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: canal@injection: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 72:c1:29:dd:ff:02 brd ff:ff:ff:ff:ff:ff
    inet 10.87.87.2/30 scope global canal
        valid_lft forever preferred_lft forever
    inet6 fe80::70c1:29ff:fedd:ff02/64 scope link
        valid_lft forever preferred_lft forever
3: injection@canal: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 9a:a7:ef:2c:58:95 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::98a7:efff:fe2c:5895/64 scope link
        valid_lft forever preferred_lft forever
13: h3-eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether b6:af:b7:19:fc:e0 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.10.3/24 scope global h3-eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::b4af:b7ff:fe19:fce0/64 scope link
        valid_lft forever preferred_lft forever

```

Fig. 3. Configuration Machine H3

Donc sur le réseau, nous avons deux hôtes, h1 et h3, avec les configurations suivantes :

- **h1** : Son interface h1-eth0 est configurée avec l'adresse IP 192.168.10.1 et son canal caché est à l'adresse IP 10.87.87.1.
- **h3** : Son interface h3-eth0 est configurée avec l'adresse IP 192.168.10.3 et son canal caché est à l'adresse IP 10.87.87.2.

Actuellement, ces hôtes peuvent se communiquer (ping) via leurs interfaces en 192.168.10.0 mais pas via le canal caché en 10.87. L'objectif est donc d'utiliser une communication indirecte via ICMP (*Internet Control Message Protocol*) avec les adresses en 192.168.10.0 pour transporter toutes les données des protocoles UDP ou TCP du canal caché.

4.3 Configuration IPTables

Pour le fonctionnement du tunnel caché, nous devons exécuter les règles iptables suivantes :

- Sur h1

```

sudo modprobe iptable_raw
sudo iptables -A PREROUTING -t raw -p tcp -j ACCEPT

# redirect packet to nfqueue rule
#paquets entrants
sudo iptables -t raw -A PREROUTING -s 192.168.10.3 -p icmp -j
    ↪ NFQUEUE --queue-num 2

#paquets sortants
sudo iptables -t raw -A OUTPUT -d 10.87.87.2 -j NFQUEUE --queue-num
    ↪ 0

```

— Sur h3

```
#authorize raw packet
sudo modprobe iptable_raw
sudo iptables -A PREROUTING -t raw -p tcp -j ACCEPT

# redirect packet to nfqueue rule
#paquets entrants
sudo iptables -t raw -A PREROUTING -s 192.168.10.1 -p icmp -j
    ↪ NFQUEUE --queue-num 2

#paquets sortants
sudo iptables -t raw -A OUTPUT -d 10.87.87.1 -j NFQUEUE --queue-num
    ↪ 0
```

Explication des Commandes :

1. Autoriser les Paquets Bruts :

La première commande permet l'acceptation des paquets bruts. Cette étape est nécessaire car nous prévoyons d'intercepter le trafic ICMP entrant et TCP ou UDP sortant via Scapy et de créer de nouveaux paquets. Nous activons donc le traitement des paquets bruts pour faciliter cette opération.

2. Récupérer les Paquets ICMP Entrants :

La seconde commande vise à récupérer les paquets ICMP entrants, qui sont susceptibles de contenir le canal caché. Nous les interceptons pour les analyser et extraire les données à injecter dans le canal caché.

3. Gérer les Paquets Sortants vers le Canal Caché :

Cette commande gère les paquets sortants à destination du canal caché. Elle les intercepte pour les encapsuler dans des paquets ICMP avant de les envoyer. Cela permet d'utiliser ICMP pour transporter les données des protocoles UDP ou TCP via le canal caché

Nous avons choisi d'utiliser la table raw d'Iptables pour ces règles car elle nous permet de manipuler les paquets (bruts) au niveau le plus basique avant qu'ils ne soient affectés par les autres tables de filtrage ou qu'ils soient routés. Les NFQUEUE nous permettent de déléguer la gestion des paquets marqués à nos scripts Scapy, offrant ainsi une flexibilité dans le traitement du trafic réseau.

4.4 Traitement des Paquets avec Scapy

À ce stade, lorsqu'un utilisateur se trouve par exemple sur l'hôte **h1** et lance l'outil **socat** avec la commande suivante :

```
socat - UDP4:10.87.87.2:6789,bind=10.87.87.1:6789
bonjour
```

Les datagrammes UDP créés à destination de l'hôte **h3** (10.87.87.2) seront interceptés par la règle Iptables et placés dans la file NFQUEUE 0 de **h1**. Ainsi, il est nécessaire de développer un script Scapy qui interceptera ces paquets et les encapsulera dans des paquets ICMP à destination de **h3** (192.168.10.3). Voici le code Scapy faisant ce travail :

```
from scapy.layers.inet import IP, TCP, UDP
from netfilterqueue import NetfilterQueue
import icmp_builder

def tracker(pkt):
    packet = IP(pkt.get_payload())
    if packet.haslayer(TCP):
        #afficher_infos_paquet(packet)
        handle_tcp_packet(packet)
    elif packet.haslayer(UDP):
        handle_udp_packet(packet)
    else:
        pass
    #print(packet.summary())
    pkt.drop()

nfqueue = NetfilterQueue()
nfqueue.bind(0, tracker) # Le numero de file d'attente 0 pour
    ↪ correspondre aux regles iptables
try:
    nfqueue.run()
except KeyboardInterrupt:
    nfqueue.unbind()
```


Et voici les fonctions qui vont se charger de construire le payload ICMP selon que le trafic depuis Socat soit en UDP ou TCP :

```
def handle_tcp_packet(packet):
    # Extraire les couches TCP et IP du paquet
    tcp_layer = packet[TCP]
    ip_layer = packet[IP]

    # Recuperer les adresses IP et les ports source et destination
    src_ip = ip_layer.src
    src_port = tcp_layer.sport
    dst_ip = ip_layer.dst
    dst_port = tcp_layer.dport

    # Creer un nouveau paquet IP avec la charge utile TCP
    payload = IP(src=src_ip, dst=dst_ip) / tcp_layer / tcp_layer.
        ➔ payload

    # Envoyer le paquet ICMP avec la charge utile TCP comme payload
    icmp_builder.envoyer_paquets(payload)

    # Afficher les informations du paquet TCP envoye
    print(f"Sending the TCP Packet from {src_ip}:{src_port} to {
        ➔ dst_ip}:{dst_port}")

def handle_udp_packet(packet):
    # Extraire les couches UDP et IP du paquet
    udp_layer = packet[UDP]
    ip_layer = packet[IP]

    # Recuperer les adresses IP et les ports source et destination
    src_ip = ip_layer.src
    src_port = udp_layer.sport
    dst_ip = ip_layer.dst
    dst_port = udp_layer.dport

    # Creer un nouveau paquet IP et UDP avec la charge utile UDP
    payload = IP(src=src_ip, dst=dst_ip) / UDP(sport=src_port, dport
        ➔ =dst_port) / udp_layer.payload

    # Afficher les informations du paquet UDP recu
    print(f"UDP Packet from {src_ip}:{src_port} to {dst_ip}:{
        ➔ dst_port}")

    # Envoyer le paquet ICMP avec la charge utile UDP comme payload
    icmp_builder.envoyer_paquets(payload)
```

Et maintenant, le code qui prend le payload, l'encapsule dans un paquet ICMP et l'envoie vers h3 :

```
from scapy.all import *

# destination IP address (dans le cadre de h1)
dst_ip = "192.168.10.3"
# Fonction pour envoyer des paquets ICMP avec un payload
def envoyer_paquets(payload):
    # Payload personnalisée
    custom_payload = payload
    ping_packet = IP(dst=dst_ip)/ICMP()/custom_payload
    # Envoi du paquet
    send(ping_packet)
```

Par la suite, lorsque ce trafic ICMP, émis par h1, parvient à h3, il est à son tour placé dans la file NFQUEUE 2 de h3. Par conséquent, un autre script Scapy doit être mis en place sur h3 pour capturer ces paquets ICMP, vérifier la présence de la charge utile (payload) et l'injecter dans le canal caché si nécessaire. Voici ce code :

```
from scapy.all import *
from netfilterqueue import NetfilterQueue
import traceback

INTERFACE_INJECTION='injection'
INTERFACE='canal'
ADRESSE_MAC=get_if_hwaddr(INTERFACE)

def handle_icmp_packet(pkt):
    # Verifier si le paquet contient la couche ICMP
    try:
        packet = IP(pkt.get_payload())
        # print("Packet:", packet.summary())
        if ICMP in packet:
            icmp_layer = packet[ICMP]
            if Raw in icmp_layer:
                # Recuperer le champ de charge utile (payload) du
                #   ↳ paquet ICMP
                payload = icmp_layer[Raw].load

                # Extraire les couches IP et TCP
                ip_layer = IP(payload)
                if TCP in ip_layer:
                    tcp_layer = ip_layer[TCP]

                # Injecter le payload
                sendp(Ether(src=RandMAC(),dst=ADRESSE_MAC)/IP(
                    ↳ src=ip_layer[IP].src,dst=ip_layer[IP].dst)
                    ↳ /tcp_layer/tcp_layer.payload,iface=
                    ↳ INTERFACE_INJECTION)

                # Verifier si le paquet(payload) encapsule de l'UDP
                if UDP in ip_layer:
```

```

        udp_layer = ip_layer[UDP]

        # L'injecter
        sendp(Ether(src=RandMAC(), dst=ADRESSE_MAC)/IP(
            ↪ src=ip_layer[IP].src, dst=ip_layer[IP].dst)
            ↪ /UDP(sport=ip_layer[UDP].sport, dport=
            ↪ ip_layer[UDP].dport)/udp_layer.payload,
            ↪ iface=INTERFACE_INJECTION)

    else:
        print("No Raw layer in ICMP")

except Exception as e:
    print(e)
    print(traceback.format_exc())
    pkt.drop() # Abandonner le paquet en cas d'erreur

nfqueue = NetfilterQueue()
nfqueue.bind(2, handle_icmp_packet) # Le numero de file d'attente
    ↪ doit correspondre a celui specifie dans les regles iptables
try:
    nfqueue.run() # Demarrer le traitement des paquets
except KeyboardInterrupt:
    nfqueue.unbind() # Deliaage de la file d'attente en cas d'
    ↪ interruption

```

Le trafic retour, de H3 vers H1, doit également subir le même traitement. Les mêmes scripts seront utilisés, mais en changeant l'adresse IP de destination pour qu'elle soit celle de H1. Pour plus de détails, veuillez consulter le dossier du projet.

5 Test du canal caché

5.1 Avec Socat et le mode de transport UDP

Une fois qu'on a lancé les deux scripts scapy chez H1 et H3 ; nous allons utiliser socat pour tester le bon fonctionnement du canal caché avec socat en essayant une communication UDP entre h1 (10.87.87.1) et h3 (10.87.87.2). Nous assistons au bon fonctionnement du tunnel caché, ce que prouve les deux figures suivantes : (à noter qu'il n'y a aucune connectivité entre (10.87.87.1) et (10.87.87.2) ; tout le trafic a été envoyé via ICMP).

```

ubuntu@ubuntu2004:~/ICMPCanalCache$ [h1] socat -dd - UDP4:10.87.87.2:6789,bind=10.87.87.1:6789
2024/05/24 17:39:06 socat[110515] N reading from and writing to stdio
2024/05/24 17:39:06 socat[110515] N opening connection to AF=2 10.87.87.2:6789
2024/05/24 17:39:06 socat[110515] N successfully connected from local address AF=2 10.87.87.1:6789
2024/05/24 17:39:06 socat[110515] N starting data transfer loop with FDs [0,1] and [5,5]
test canal caché
un autre test h3 vers h1
finalement de h1 vers h3

```

Fig. 4. H1 Socat UDP Communication

```
ubuntu@ubuntu2004:~/ICMPCanalCache$ [h3] socat -dd - UDP4:10.87.87.1:6789,bind=10.87.87.2:6789
2024/05/24 17:40:02 socat[110521] N reading from and writing to stdio
2024/05/24 17:40:02 socat[110521] N opening connection to AF=2 10.87.87.1:6789
2024/05/24 17:40:02 socat[110521] N successfully connected from local address AF=2 10.87.87.2:6789
2024/05/24 17:40:02 socat[110521] N starting data transfer loop with FDs [0,1] and [5,5]
test canal caché
un autre test h3 vers h1
finalement de h1 vers h3
```

Fig. 5. H3 Socat UDP Communication

Durant ce trafic, nous avons effectué un tcpdump afin d'observer les paquets ICMP envoyés et reçus. On constate que des requêtes d'écho ont bien été envoyées, mais qu'il n'y a pas de réponses d'écho. Cela s'explique par le fait que, sur h3, lorsque la requête ping est reçue, elle est placée dans une file d'attente gérée par Scapy. Dans le code Scapy, une fois que nous avons extrait la charge utile (payload), nous n'avons plus besoin du paquet, donc nous le supprimons. Si h3 souhaite répondre, son trafic sera également encapsulé dans une requête d'écho. On a trois echo requests pour les trois messages échangés (l'horodatage le confirme également).

```
ubuntu@ubuntu2004:~/ICMPCanalCache$ [h1] sudo tcpdump -nvee -i h1-eth0 icmp
tcpdump: listening on h1-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C17:40:15.807244 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 88: (tos 0x0, ttl 64, id 1, offset 0
, flags [none], proto ICMP (1), length 74)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 54
17:40:56.282433 b6:af:b7:19:fc:e0 > c6:c6:21:3e:9b:8c, ethertype IPv4 (0x0800), length 95: (tos 0x0, ttl 64, id 1, offset 0,
flags [none], proto ICMP (1), length 81)
    192.168.10.3 > 192.168.10.1: ICMP echo request, id 0, seq 0, length 61
17:41:09.868517 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 96: (tos 0x0, ttl 64, id 1, offset 0,
flags [none], proto ICMP (1), length 82)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 62

3 packets captured
3 packets received by filter
```

Fig. 6. Paquets ICMP capturés avec tcpdump sur h1

Nous avons également affiché un paquet summary du trafic ICMP reçu lors d'un message UDP : On voit bien que dans le payload, on arrive à voir notre message (juste après les en-têtes UDP) et il s'agit aussi d'un echo request.

```
^Cubuntu@ubuntu2004:~/ICMPCanalCache/netlab$ [h1] sudo python3 ../alice/icmp_extractor.py
(icmp_extractor.py:110753): dbind-WARNING **: 17:54:50.627: Couldn't connect to accessibility bus: Failed to connect to socke
t /tmp/dbus-TBz7nIFb0f: Connexion refusée
Packet: IP / ICMP 192.168.10.3 > 192.168.10.1 echo-request 0 / Raw
Payload ICMP: b'E\x00\x005\x00\x01\x00\x00@\x11\x06\x06\x06\x02\x01\x1a\x85\x1a\x85\x00!\x0c\x01un autre test h3 vers h1\
n'
Taille du payload: 53
IP Layer: IP / UDP 10.87.87.2:6789 > 10.87.87.1:6789 / Raw
.
Sent 1 packets.
```

Fig. 7. ICMP trafic

Finalement, en regardant l'état de notre table raw d'iptables, on observe que les règles s'appliquent bien :

```
ubuntu@ubuntu2004:~/ICMPCanalCache$ [h1] sudo iptables -t raw -L -v -n
```

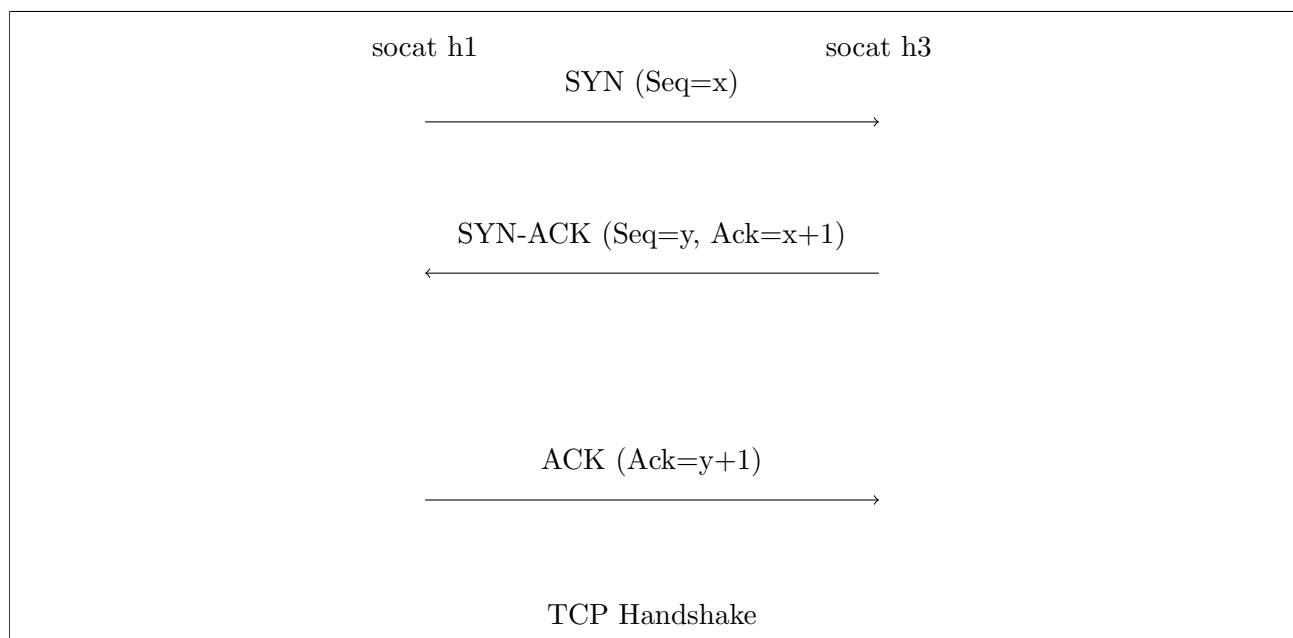
Chain PREROUTING (policy ACCEPT 56 packets, 5414 bytes)									
pkts	bytes	target	prot	opt	in	out	source	destination	
124	8570	ACCEPT	tcp	--	*	*	0.0.0.0/0	0.0.0.0/0	
114	10498	NFQUEUE	icmp	--	*	*	192.168.10.3	0.0.0.0/0	NFQUEUE num 2

Chain OUTPUT (policy ACCEPT 65 packets, 5914 bytes)									
pkts	bytes	target	prot	opt	in	out	source	destination	
160	9496	NFQUEUE	all	--	*	*	0.0.0.0/0	10.87.87.2	NFQUEUE num 0

Fig. 8. Etat de la table raw de H1

5.2 Mode de Transport TCP : Établissement de la Connexion

Pour le mode TCP, la tâche est un peu plus complexe en raison de la nécessité d'établir une connexion entre les deux pairs via le processus de Handshake TCP. Ce processus implique un échange de paquets SYN, SYN/ACK, et ACK afin de synchroniser les numéros de séquence et d'acquittement entre les deux parties.



Dans notre configuration, la machine *h3* joue le rôle du serveur en mode *listen*, donc on exécutera la commande suivante :

```
socat -dd - TCP-LISTEN:6789,bind=10.87.87.2,reuseaddr,fork
# -dd ---> verbose verbose mode
```

Tandis que *h1* joue le rôle du client :

```
socat -dd - TCP:10.87.87.2:6789
# -dd ---> verbose verbose mode
```

Il est crucial que les deux parties s'accordent sur leurs rôles respectifs pour initier correctement la connexion.

Afin d'assurer une gestion correcte des paquets de contrôle TCP (SYN, SYN/ACK, ACK), il est nécessaire d'intercepter ces paquets, de les modifier si besoin, et de les renvoyer avec les numéros de séquence et d'acquittement appropriés. Cette approche nous permet de maintenir la synchronisation entre les deux hôtes tout en encapsulant les paquets dans des messages ICMP pour les dissimuler.

Pour faciliter le suivi et la vérification du processus d'établissement de la connexion, nous avons ajouté une fonction qui affiche des informations détaillées sur chaque paquet TCP intercepté. Cette fonction, *afficher_infos_paquet*, permet de diagnostiquer et de valider le bon déroulement de la connexion TCP.

```
def afficher_infos_paquet(paquet):  
    # Verifier si le paquet contient une couche IP  
    if IP in paquet:  
        ip_src = paquet[IP].src  
        ip_dst = paquet[IP].dst  
        tos = paquet[IP].tos  
  
        print(f"IP Source: {ip_src}")  
        print(f"IP Destination: {ip_dst}")  
        print(f"TOS: {tos}")  
  
    # Verifier si le paquet contient une couche TCP  
    if TCP in paquet:  
        flags = paquet[TCP].flags  
        seq = paquet[TCP].seq  
        ack = paquet[TCP].ack  
  
        print(f"TCP Flags: {flags}")  
        print(f"TCP Sequence Number: {seq}")  
        print(f"TCP Acknowledgment Number: {ack}")
```

Cette fonction permet de capturer et d'afficher les informations clés des paquets TCP, telles que les adresses IP source et destination, le champ TOS (Type of Service), ainsi que les drapeaux TCP, les numéros de séquence et d'acquittement.

Après execution de ces commandes, avec les codes scapy bien mis en place ; on assiste à l'établissement de la connexion entre H1 et H3 ; Voici en détail ce qui s'est passé sur H1 :

```
ubuntu@ubuntu2004:~/ICMPCanalCache$ [h1] sudo python3 packet_tracker.py

(packet_tracker.py:114077): dbind-WARNING **: 15:45:32.544: Couldn't connect to accessibility bus:
s-TBz7nIFb0f: Connexion refusée
IP Source: 10.87.87.1
IP Destination: 10.87.87.2
TOS: 0
TCP Flags: S
TCP Sequence Number: 368658432
TCP Acknowledgment Number: 0
.
Sent 1 packets.
envoi envoie envoie
Sending the TCP Packet from 10.87.87.1:38530 to 10.87.87.2:6789
Payload:
IP Source: 10.87.87.1
IP Destination: 10.87.87.2
TOS: 0
TCP Flags: A
TCP Sequence Number: 368658433
TCP Acknowledgment Number: 2065060977
.
Sent 1 packets.
envoi envoie envoie
Sending the TCP Packet from 10.87.87.1:38530 to 10.87.87.2:6789
Payload:
IP Source: 10.87.87.1
IP Destination: 10.87.87.2
TOS: 0
TCP Flags: PA
TCP Sequence Number: 368658433
TCP Acknowledgment Number: 2065060977
.
Sent 1 packets.
envoi envoie envoie
Sending the TCP Packet from 10.87.87.1:38530 to 10.87.87.2:6789
Payload: b'bonjour depuis h1\n'
IP Source: 10.87.87.1
IP Destination: 10.87.87.2
TOS: 0
TCP Flags: PA
TCP Sequence Number: 368658433
TCP Acknowledgment Number: 2065060977
```

Fig. 9. Paquets envoyés depuis H1

Sur la figure (FIG 9), on voit bien que la machine H1 a envoyé un paquet SYN et un paquet ACK, et entre les deux , on peut voir sur la figure (FIG 10) le paquet SA reçu depuis la machine H3.

```

ubuntu@ubuntu2004:~/ICMPCanalCache/netlab$ [h1] sudo python3 ../icmp_extractor.py

(icmp_extractor.py:114096): dbind-WARNING **: 15:45:46.374: Couldn't connect to accessibility bus:
s-TBz7nIFb0f: Connexion refusée
Packet: IP / ICMP 192.168.10.3 > 192.168.10.1 echo-request 0 / Raw
IP Source: 10.87.87.2
IP Destination: 10.87.87.1
TOS: 0
TCP Flags: SA
TCP Sequence Number: 2065060976
TCP Acknowledgment Number: 368658433
TCP Layer: TCP 10.87.87.2:6789 > 10.87.87.1:38530 SA
Payload:
.
Sent 1 packets.
Packet: IP / ICMP 192.168.10.3 > 192.168.10.1 echo-request 0 / Raw
IP Source: 10.87.87.2
IP Destination: 10.87.87.1
TOS: 0
TCP Flags: PA
TCP Sequence Number: 2065060977
TCP Acknowledgment Number: 368658433
TCP Layer: TCP 10.87.87.2:6789 > 10.87.87.1:38530 PA / Raw
Payload: b'bonjour depuis h1\nbonjour depuis h1\n'

```

Fig. 10. Paquets recus sur H1

En lançant également la commande pour voir l'état des connexions sur H1, on observe bien une connexion établie entre les deux hôtes :

```

Toutes les 1,0s: sudo ss -tu state all
ubuntu2004: Sat May 25 15:46:04 2024

Netid  State  Recv-Q  Send-Q  Local Address:Port  Peer Address:Port Process
tcp    ESTAB  0        0        10.87.87.1:53200    10.87.87.2:6789

```

Fig. 11. H1 TCP Connexions State

Et voici les terminals SOCAT qui prouvent bien aussi l'établissement de la connexion :

```

ubuntu@ubuntu2004:~/ICMPCanalCache$ [h1] socat -dd - TCP:10.87.87.2:6789
2024/05/25 15:46:04 socat[114109] N reading from and writing to stdio
2024/05/25 15:46:04 socat[114109] N opening connection to AF=2 10.87.87.2:6789
2024/05/25 15:46:04 socat[114109] N successfully connected from local address AF=2 10.87.87.1:38530
2024/05/25 15:46:04 socat[114109] N starting data transfer loop with FDs [0,1] and [5,5]
bonjour depuis h1
bonjour depuis h3

```

Fig. 12. Socat TCP Client (H1)


```

ubuntu@ubuntu2004:~/ICMPCanalCache/bob$ [h3] socat -dd - TCP-LISTEN:6789,bind=10.87.87.2,reuseaddr,fork
2024/05/25 15:45:19 socat[114075] N reading from and writing to stdio
2024/05/25 15:45:19 socat[114075] N listening on AF=2 10.87.87.2:6789
2024/05/25 15:46:04 socat[114075] N accepting connection from AF=2 10.87.87.1:38530 on AF=2 10.87.87.2:6789
2024/05/25 15:46:04 socat[114075] N forked off child process 114112
2024/05/25 15:46:04 socat[114075] N listening on AF=2 10.87.87.2:6789
2024/05/25 15:46:04 socat[114112] N starting data transfer loop with FDs [0,1] and [6,6]
bonjour depuis h1
bonjour depuis h3

```

Fig. 13. Socat TCP Server (H3)

Une capture du trafic icmp avec tcpdump sur h1 nous montre également les echo-requests correspondants à l'établissement de la connexion TCP ainsi que les futurs échanges.

```

ubuntu@ubuntu2004:~/ICMPCanalCache$ [h1] sudo tcpdump -nvee -i h1-eth0 icmp
tcpdump: listening on h1-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
15:46:04.546788 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 102: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 88)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 68
15:46:04.636387 b6:af:b7:19:fc:e0 > c6:c6:21:3e:9b:8c, ethertype IPv4 (0x0800), length 102: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 88)
    192.168.10.3 > 192.168.10.1: ICMP echo request, id 0, seq 0, length 68
15:46:04.671271 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 94: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 80)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 60
15:49:26.887073 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 130: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 116)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 96
15:49:27.378006 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 130: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 116)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 96
15:49:27.928472 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 130: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 116)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 96
15:49:28.975566 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 130: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 116)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 96
15:49:31.200121 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 130: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 116)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 96
15:49:35.550661 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 130: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 116)
    192.168.10.1 > 192.168.10.3: ICMP echo request, id 0, seq 0, length 96
15:49:44.003961 c6:c6:21:3e:9b:8c > b6:af:b7:19:fc:e0, ethertype IPv4 (0x0800), length 130: (tos 0x0, ttl 64, id 1, offset 0, flags
[none], proto ICMP (1), length 116)

```

Fig. 14. Capture du trafic sur H1

6 Sécurisation du tunnel

Dans les sections précédentes, nous avons démontré le bon fonctionnement du tunnel en mode UDP et TCP, mais cela montre aussi que des tiers pourraient intercepter les paquets échangés entre les deux interlocuteurs. En utilisant des outils comme Scapy pour inspecter les données, ils pourraient accéder directement à la nature et au contenu des échanges effectués via le canal caché. Cela compromet la sécurité du canal, qui est censé rester invisible.

Pour renforcer la sécurité, nous allons chiffrer nos échanges. Deux options s'offrent à nous : chiffrer uniquement les données, ou chiffrer à la fois les données et les en-têtes. Nous optons pour la seconde solution, car en chiffrant uniquement les données, une tierce personne pourrait encore déduire la nature des échanges et potentiellement les bloquer. En chiffrant à la fois les données et les en-têtes, seuls les deux interlocuteurs peuvent accéder au contenu des communications.

Ainsi, il est nécessaire de chiffrer entièrement le payload avant de l'encapsuler dans un paquet ICMP et de le déchiffrer avant d'analyser la nature du trafic.

Nous continuerons d'utiliser le mode UDP car il est plus léger et ne nécessite pas l'établissement d'une connexion particulière, ce qui est avantageux pour les canaux cachés.

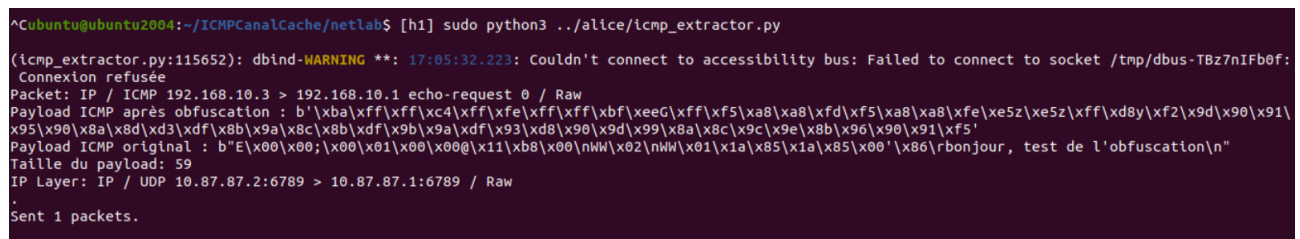
6.1 Sécurisation par l'obfuscation des données

L'obfuscation vise à rendre les données moins compréhensibles pour les tiers non autorisés et offre un avantage crucial dans le contexte des canaux cachés en dissimulant la nature même de la communication, rendant ainsi la détection et l'interprétation du trafic beaucoup plus ardues. Cette technique implique souvent des manipulations telles que l'inversion des bits des données, les rendant ainsi difficilement reconnaissables. Pour faire l'obfuscation, on a :

```
# Obfuscation : Inverser les bits de données avec XOR
def obfusquer_data(payload):

    bytes_payload = bytes(payload)
    return bytes([b ^ 0xFF for b in bytes_payload])
```

Avant d'envoyer les données et après les avoir reçues, nous avons appliqué cette technique d'obfuscation. Pour le test, nous avons lancé le canal en mode UDP comme décrit dans la partie 5.1. Un exemple illustrant l'efficacité de cette méthode est visible dans les données obfusquées reçues depuis h3 sur h1. On observe clairement que le contenu des données obfusquées diffère du contenu original, car les bits ont été inversés. Il est même impossible de lire notre message original avant la dé-obfuscation, démontrant ainsi l'efficacité de l'obfuscation, comme le montre la figure 15.



```
^Cubuntu@ubuntu2004:~/ICMPCanalCache/netlab$ [h1] sudo python3 ../alice/icmp_extractor.py
(icmp_extractor.py:115652): dbind-WARNING **: 17:05:32.223: Couldn't connect to accessibility bus: Failed to connect to socket /tmp/dbus-TBz7nIFb0f:
Connexion refusée
Packet: IP / ICMP 192.168.10.3 > 192.168.10.1 echo-request 0 / Raw
Payload ICMP après obfuscation : b'\xba\xff\xff\xc4\xff\xfe\xff\xbf\xeeG\xff\x55\xa8\xa8\xfd\x55\xa8\xfe\xe5z\xe5z\xff\xd8y\xf2\x9d\x90\x91\x95\x90\x8a\x8d\xd3\xdf\x8b\x9a\x8c\x8b\xdf\x9b\x9a\xdf\x93\x8d\x90\x9d\x99\x8a\x8c\x9c\x9e\x8b\x96\x90\x91\xf5'
Payload ICMP original : b'E\x00\x00;\x00\x01\x00\x00@\x11\xb8\x00nM\x02nM\x01\x1a\x85\x1a\x85\x00'\x86'rbonjour, test de l'obfuscation\n"
Taille du payload: 59
IP Layer: IP / UDP 10.87.87.2:6789 > 10.87.87.1:6789 / Raw
.
Sent 1 packets.
```

Fig. 15. Obfuscation fonctionnel

6.2 Chiffrement via AES

Pour renforcer davantage la sécurité, nous pouvons aller plus loin en mettant en place un chiffrement symétrique avec AES (Advanced Encryption Standard). Car bien que l'obfuscation offre une certaine protection, elle présente également des faiblesses. Tout d'abord, elle ne fournit pas de confidentialité réelle car elle ne masque pas le contenu des données de manière sécurisée. Un attaquant aussi est capable de re-inverser les données pour récupérer les données originales, ce qui n'est pas très coûteux. AES est un algorithme de chiffrement robuste largement utilisé pour sécuriser les données sensibles.

En utilisant AES, les données sont chiffrées avec une clé partagée entre les parties autorisées, ce qui garantit la confidentialité des données même si elles sont interceptées par des tiers non autorisés.

Nous allons mettre en place un chiffrement AES avec une taille de clé de 256bits en mode CTR car dans le mode CTR, chaque bloc de texte en clair est chiffré de manière indépendante à l'aide d'une opération XOR avec un jeton de chiffrement généré à partir de la clé et un nonce. Ainsi, la perte d'un paquet n'impacte que les données associées à ce paquet spécifique. Les autres paquets restent inchangés car ils ont été chiffrés de manière indépendante (ce qui est utile dans notre cas).

Voici le code faisant ce travail, nous allons utiliser la fonction `urandom` pour générer aléatoirement la clé symétrique ainsi que le vecteur d'initialisation que nous allons stocker dans un fichier. On suppose que cette clé est connue par les deux pairs

```
from cryptography.hazmat.primitives.ciphers import Cipher,
    ↪ algorithms, modes
from cryptography.hazmat.backends import import default_backend
from os import urandom

# read the key and nonce from the file
def read_key():
    with open("aes_key.txt", "rb") as f:
        cle = f.readline().strip()
        iv = f.readline().strip()
    return cle, iv

# Chiffrer des données avec AES en mode CTR
def chiffrer_AES_256_CTR(payload, cle, iv):
    payload = bytes(payload)
    cipher = Cipher(algorithms.AES(cle), modes.CTR(iv), backend=
    ↪ default_backend())
    encryptor = cipher.encryptor()
    donnees_chiffrees = encryptor.update(payload) + encryptor.
    ↪ finalize()
    return donnees_chiffrees

# Dechiffrer des données avec AES en mode CTR
def dechiffrer_AES_256_CTR(donnees_chiffrees, cle, iv):
    cipher = Cipher(algorithms.AES(cle), modes.CTR(iv), backend=
    ↪ default_backend())
    decryptor = cipher.decryptor()
    payload = decryptor.update(donnees_chiffrees) + decryptor.
    ↪ finalize()
    return payload
```

Nous allons relancer la communication via le canal caché entre les deux extrémités :

— Du côté de H1 :

```
ubuntu@ubuntu2004:~/ICMPCanalCache$ [h1] socat -dd - UDP4:10.87.87.2:6789,bind=10.87.87.1:6789
2024/05/25 19:05:24 socat[117833] N reading from and writing to stdio
2024/05/25 19:05:24 socat[117833] N opening connection to AF=2 10.87.87.2:6789
2024/05/25 19:05:24 socat[117833] N successfully connected from local address AF=2 10.87.87.1:6789
2024/05/25 19:05:24 socat[117833] N starting data transfer loop with FDs [0,1] and [5,5]
bonjour envoi sécurisé de h1 vers h3
et de h3 vers h1
```

Fig. 16. Secure tunnel H1 Side

— Du côté de H3 :

```
ubuntu@ubuntu2004:~/ICMPCanalCache$ [h3] socat -dd - UDP4:10.87.87.1:6789,bind=10.87.87.2:6789
2024/05/25 19:04:56 socat[117818] N reading from and writing to stdio
2024/05/25 19:04:56 socat[117818] N opening connection to AF=2 10.87.87.1:6789
2024/05/25 19:04:56 socat[117818] N successfully connected from local address AF=2 10.87.87.2:6789
2024/05/25 19:04:56 socat[117818] N starting data transfer loop with FDs [0,1] and [5,5]
bonjour envoi sécurisé de h1 vers h3
et de h3 vers h1
```

Fig. 17. Secure tunnel H3 Side

On observe la transparence et le bon fonctionnement des communications ; cependant, la communication entre H1 et H3 est bien chiffrée, comme le montre le paquet chiffré reçu sur H1 depuis H3 :

```
ubuntu@ubuntu2004:~/ICMPCanalCache$ [h1] sudo python3 alice/icmp_extractor.py
(icmp_extractor.py:117821): dbind-WARNING **: 19:05:09.930: Couldn't connect to accessibility bus: Failed to connect to socket /tmp/dbus-TBz7n
IFb0f: Connexion refusée
Packet: IP / ICMP 192.168.10.3 > 192.168.10.1 echo-request 0 / Raw
Payload ICMP après chiffrement : b'\xc4\xf1>\xa1n\xf2\x17\xea/Ig\xb8vV\xeeh$5I\xef\x02\x12\x1d&\x1bU\x0f\xa5\x8bc\xb3\xd8jI\xe2\x13\xd35\xd5\x92A/o '
Payload ICMP original : b'E\x00\x00-\x00\x01\x00\x00@\x11\xb8\x0e\nWW\x02\nWW\x01\x1a\x85\x1a\x85\x00\x19I\x9aet de h3 vers h1\n'
Taille du payload: 45
IP Layer: IP / UDP 10.87.87.2:6789 > 10.87.87.1:6789 / Raw
```

Fig. 18. Effective ciphering

7 Conclusion

Ce projet avait pour objectif de développer un outil de communication sécurisée entre deux machines en utilisant un canal caché, rendant les échanges indétectables par les outils de surveillance réseau. En utilisant des interfaces veth pour créer le tunnel et en encapsulant les paquets de données dans des paquets ICMP, nous avons mis au point un canal de communication à la fois efficace et discret. Pour garantir la sécurité des données transmises, nous avons utilisé des techniques d'obfuscation et de chiffrement AES.

Cependant, nous avons rencontré plusieurs défis lors de l'établissement des connexions TCP. L'un des principaux obstacles a été de maintenir la stabilité de la connexion tout en assurant l'invisibilité du canal caché.

Les tests effectués avec les protocoles TCP et UDP ont révélé que les canaux cachés constituent une menace réelle pour la sécurité des systèmes de communication. En contournant les mécanismes de surveillance et de sécurité existants, ils posent des risques significatifs pour la protection des informations sensibles. Ce projet souligne également la nécessité de repenser certains protocoles de communication qui offrent des options de payload, car celles-ci peuvent constituer des failles de sécurité potentielles. Bien que notre solution soit robuste et fonctionnelle, il reste des possibilités d'amélioration. L'optimisation des performances et l'exploration de nouvelles méthodes de dissimulation pourraient accroître davantage la sécurité. En conclusion, ce projet met en lumière l'importance de la vigilance et de l'innovation continue dans le domaine de la sécurité des communications.