Ydays 2021 - 2022
Laboratoire sécurité des systèmes d'information

Rapport de projet fil rouge

WireFish

Antoine SANSON et Benjamin DELSOL

M1 Cybersécurité

Dirlabs : Steven DIAS et Lucas MOUCHAGUE

Mardi 3 mai 2022

Introduction

En quelques mots, notre projet intitulé « **Wirefish** », est un cadre de travail permettant le développement de modules spécialisés d'analyse de trafic réseau, cette idée de module sera bien évidemment développée dans la suite de ce rapport.

Vous l'aurez remarqué, le nom « **Wirefish** » fait bien évidemment référence à « **Wireshark** », qui est probablement l'outil de capture et d'analyse réseau le plus connu et le plus ouvert au grand public, avec une interface assez simple à prendre en main, et une license open-source, ce qui rend son utilisation gratuite..
Le « fish » (poisson) de « **Wirefish** » est bien moins puissant que le « shark » (requin) de « **Wireshark** », bien que ces deux animaux évoluent dans le même domaine avec une idée commune.

Ce projet est né de l'idée selon laquelle la majeure partie des outils d'analyse de trafic réseau visent à fournir des informations exhaustives sur les paquets passant par les différentes interfaces de la machine hôte, sans nécessairement permettre une analyse très ciblée et « paramétrable », concernant un certain type de trafic (par exemple, le trafic entre un client et un serveur web).

Dans un premier temps, nous allons présenter comment nous en sommes venus à cette idée et au choix crucial des technologies utilisées, puis nous allons au travers de schémas simples, expliquer brièvement l'architecture globale de **Wirefish**.
Suite à cela, nous présenterons comment installer cet outil, puis nous allons expliquer le choix des technologies utilisées, avant de fournir une documentation fonctionnelle reprenant l'ensemble du travail réalisé.

Enfin, nous aborderons les problèmes et difficultés rencontrées tout au long du développement, quels axes d'améliorations peuvent être envisagés et enfin nous allons conclure sur les apports de ce projet d'un point de vue personnel.

Bonne lecture !

Justifications et réflexions autour du projet

Idée de base

Notre idée d'origine était de réaliser un outil de capture réseau, ce qui nous aurait poussé à nous documenter et à comprendre en profondeur le fonctionnement de la pile TCP/IP et de la couche OSI, le niveau de détail aurait été déterminé par notre volonté de s'appuyer ou non sur des bibliothèques déjà existantes, et donc de se rapprocher plus ou moins de la couche physique, au plus bas niveau.
Cependant, plusieurs grandes interrogations se sont posées :

- Avons nous les compétences, la détermination et le temps pour être capables de nous documenter et de comprendre les subtilités du traitement des paquets afin de rendre quelque chose de fonctionnel et de présentable dans les délais ?
- Serait-il possible de tirer parti de notre outil afin d'extraire de la valeur ajoutée ? Autrement dit, est-ce que nous allons nous contenter de refaire, probablement en moins efficace et en moins complet, quelque chose qui n'existe pas déjà ?
- Quelles sont les perspectives d'amélioration potentielles du projet, outre le fait d'ajouter sans cesse le support de nouveaux protocoles ? Y aurait-il un aspect communautaire ?

Nouvelle direction du projet

Après un certain temps de réflexion, nous avons décidé que la meilleure chose à faire serait de légèrement changer l'orientation du projet. Étant tous les deux intéressés par la sécurité des systèmes informatiques, pourquoi ne pas se focaliser sur l'analyse de paquets afin d'identifier des données sensibles transitant dans le réseau ? Ainsi, les 3 questions ci-dessus ont trouvé réponse :

- Nous n'avons pas les compétences à l'heure actuelle, mais il est toujours possible de les acquérir, encore faut-il du temps et beaucoup de détermination. Nous allons plutôt appliquer une couche d'abstraction sur le modèle OSI à l'aide d'une bibliothèque d'analyse réseau et nous focaliser sur des protocoles applicatifs.
- Oui, en fournissant une structure et des outils pour extraire des informations sensibles du réseau, il y aurait un réel intérêt à utiliser **Wirefish**.
- Une fois la base du projet construite, on souhaite que chaque fonctionnalité soit sous la forme d'un module (plug-in) développé par quiconque en aurait l'envie.

On souhaite que la création de nouveaux modules soit simple et efficace, et que l'on n'ait pas besoin de comprendre précisément comment fonctionne **Wirefish** pour créer des modules fonctionnels. Bien sûr, il y aura toujours des améliorations potentielles à apporter au cœur de l'outil, que ce soit des ajouts de fonctionnalités pour les modules, des corrections de bugs et des optimisations, mais le gros du travail sera lié au développement des modules, qui pourrait représenter l'écosystème de **Wirefish**.

Cas d'utilisation

Lorsqu'on réfléchit aux cas d'utilisations que **Wirefish** pourrait avoir, on peut avoir plusieurs idées, selon notre relation avec la sécurité des systèmes informatiques.

Dans un premier temps, nous avons pensé qu'il pourrait être efficace à des fins d'expérimentation dans un cadre local, afin d'explorer comment notre machine interagit avec les autres, en étant restreint aux modules disponibles bien évidemment.

Une autre idée intéressante pourrait être, dans le cadre d'un test d'intrusion, d'installer l'outil sur un poste compromis silencieusement, afin d'écouter le trafic et d'éventuellement progresser dans le test d'intrusion en trouvant de nouvelles informations sensibles transitant entre la machine compromise et un serveur quelconque. Ceci pourrait être efficace étant donné que **Wirefish** est complètement passif sur le réseau, puisqu'il se contente d'écouter et n'émet aucun trafic. Cependant, cette idée vient avec différentes problématiques, puisque notre outil requiert un certain environnement pour fonctionner correctement.
Déjà, il faut qu'un interpréteur **Python3** soit disponible, il faut avoir une connexion internet pour télécharger les dépendances et surtout, avoir un accès administrateur pour écouter le trafic passant par les différentes interfaces. Pour les deux premiers problèmes, il pourrait toujours être possible de créer une version exécutable du programme (avec **pyinstaller** par exemple), même si cela ne serait pas très efficace et on ne serait pas à l'abri de bugs et de crashs une fois l'outil utilisé en conditions réelles, à cause de spécificités de l'environnement de la machine hôte.
De plus, le fait de devoir avoir un accès administrateur est un gros inconvénient : autant utiliser cet accès pour effectuer des actions plus critiques sur le poste ou sur le réseau.

C'est donc après cette réflexion que l'on a jugé que le cas d'utilisation le plus pertinent serait le cas où on analyse le trafic réseau en temps réel dans le cas d'une attaque **Man In The Middle**, où **Wirefish** est exécuté sur la machine de l'attaquant et analyse à la volée le trafic intercepté à la recherche de données sensibles (mots de passes ou tokens d'authentification en clair, transfert de fichiers, etc.).

État de l'art

En revanche, avant de se lancer dans cette voie et de commencer à taper frénétiquement du code sur nos claviers, il nous a paru judicieux d'effectuer un état de l'art couvrant l'ensemble des outils et applications opérant dans le domaine de la capture et/ou de l'analyse du trafic réseau, afin de penser à une méthodologie et de découvrir des technologies adaptées.

La liste des logiciels évoqués ci-dessous n'est pas exhaustive, elle couvre simplement les 3 plus connus, mais il en existe bien d'autres qui ont chacun leur spécificité.

Wireshark

L'outil le plus connu de tous, il est doté d'une interface graphique simple et de nombreuses fonctionnalités de tri, de recherche, ou encore de suivi de flux réseau. Il existe depuis plusieurs décennies et son développement est basé sur l'open-source, ce qui prouve encore une fois la puissance de l'open-source. Il embarque également de nombreuses fonctionnalités avancées, comme le déchiffrement de certains protocoles, la décompression des données à la volée, ou encore l'analyse de VoIP (Voice Over IP).
Il existe également en version ligne de commande : **tshark**.
Il est possible d'écrire des scripts en langage **Lua** pour étendre les capacités de **Wireshark**, mais le but est surtout de fournir un cadre d'expérimentation propice au développement de nouvelles fonctionnalités qui pourraient être intégrées nativement dans le futur.

tcpdump

Il s'agit d'un utilitaire historique en ligne de commande, disponible nativement sur la plupart des distributions **Linux**. Il permet d'effectuer des actions simples afin de résoudre un problème particulier. Par exemple, afficher l'ensemble des interfaces réseau, capturer le trafic d'une interface, appliquer des filtres plus ou moins complexes et afficher à l'écran ou bien enregistrer les paquets dans un fichier de capture réseau afin de les analyser plus tard, avec un outil plus adapté à l'analyse, tel que **Wireshark**.
Son équivalent sous **Windows** est **WinDump**.

NetworkMiner

Il s'agit d'un outil d'analyse forensique réseau, qui peut être utilisé pour capturer le réseau afin de détecter des données diverses : systèmes d'exploitation, noms d'hôtes, ports ouverts, et bien d'autres. La partie intéressante est qu'il supporte de nombreux protocols applicatifs comme **FTP**, **HTTP**, **SMB**, **SMTP** et qu'il permet d'extraire, entre-autres, des fichiers transférés par le biais de ces protocoles.
Il s'agit plus ou moins là de ce que l'on cherche à faire, et cet outil est une découverte intéressante. Ceci nous a donné des idées concernant des fonctionnalités (modules) à développer pour **Wirefish**.

Démarrage du projet

Une fois ce travail de recherche effectué, nous avions déjà une bonne idée de ce vers quoi nous souhaitions faire tendre notre projet. N'ayant trouvé aucun autre outil reprenant strictement l'idée de notre projet, nous avons décidé de commencer la partie programmation, tout en sachant qu'on trouverait largement assez de documentation parmi les forums, les documentations officielles et les articles explicatifs.

Schémas fonctionnels

Avant d'entrer un peu plus dans les détails techniques, voici deux schémas permettant de comprendre rapidement comment fonctionne **Wirefish**.

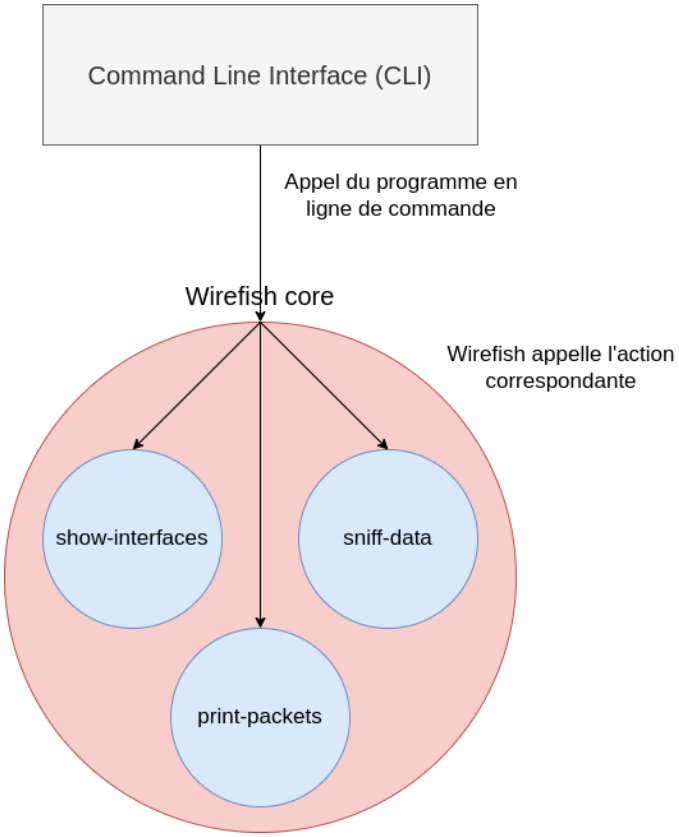


Schéma illustrant l'architecture générale de **Wirefish**

Dans un premier temps, on appelle le programme en lui passant des paramètres en ligne de commande. Le premier paramètre est spécial, il désigne l'action à exécuter et les paramètres suivants vont dépendre de ce premier paramètre, puisqu'ils sont spécifiques à l'action choisie. 3 actions sont actuellement disponibles :

- show-interfaces : affiche l'ensemble des interfaces réseau disponibles, avec des détails comme l'adresse MAC ou l'adresse IPv4 associée.
- print-packets : permet d'écouter une ou plusieurs interfaces et affiche des informations élémentaires sur les paquets en ligne de console. Il est possible d'appliquer un filtre optionnel (par exemple, uniquement les paquets dont la destination est 192.168.1.2 sur le port 80). Cette action n'apporte pas grand chose en soi, elle permet surtout de vérifier que **Wirefish** fonctionne bien sur le système.
- sniff-data : cette action est la fonctionnalité principale de **Wirefish**, c'est elle qui permet d'utiliser les différents modules disponibles afin d'analyser le trafic en temps réel et d'en extraire les données sensibles.

Le prochain schéma va s'intéresser de plus près à **sniff-data**.

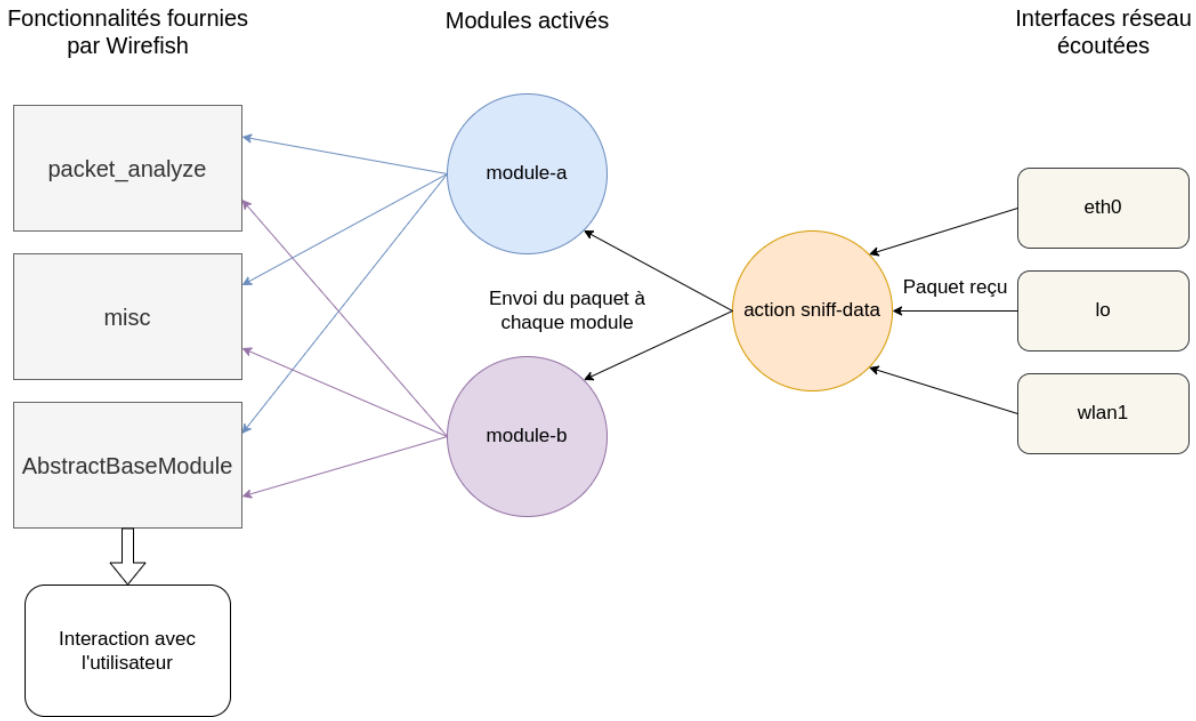


Schéma montrant les relations entre les différents éléments dans le cadre de l'utilisation de l'action sniff-data

Lorsque cette action est utilisée, une chaîne d'opérations va être exécutée, rythmée par l'arrivée et la distribution des paquets interceptés par *sniff-data* aux différents modules :

1. Dans un premier temps, on commence par écouter les différentes interfaces spécifiées.
2. Lorsqu'un paquet est intercepté, peu importe l'interface d'où il provient, celui-ci va être transmis aux différents modules, un à un, qui feront ensuite des traitements sur celui-ci. On se rend donc bien compte que chaque module est cloisonné et qu'il ne communique pas avec les autres : cela force la spécialisation des modules.
3. Les modules font appel à des fonctionnalités partagées pour traiter de paquet reçu :

    - packet_analyze est un module Python intégré au projet qui permet d'analyser et d'extraire des données ou d'interpréter le payload de certains paquets. Il faut voir ça comme une extension de **scapy** pour le traitement des paquets.
    - misc est un module Python intégré au projet qui contient des fonctions utilitaires mais qui n'ont pas de lien direct avec le traitement des paquets.
    - AbstractBaseModule est la mère de tous les modules, elle fournit des méthodes et des propriétés utiles aux classes dérivées (modules créés). Chaque module doit implémenter correctement les éventuels champs abstraits de cette classe, sans quoi le module ne sera pas chargé et donc inutilisable. Elle fournit au passage des fonctionnalités de logging et d'écriture de fichiers.
4. Les modules n'ont rien à faire pour signaler qu'ils ont terminé leur traitement, à part attendre le prochain paquet.

Documentation

Installation

Pour installer **Wirefish**, il n'y a rien de bien compliqué. Il suffit pour cela d'avoir **Python3** d'installé sur le système, de préférence la version **3.8.10** puisque c'est avec celle-ci qu'a été développé le projet, mais il y a de grandes chances que cela fonctionne avec toutes les versions de **Python3**. Si ce prérequis est rempli, alors on peut passer à la suite.

Dans un premier temps, il vous faudra cloner le dépôt **GitHub** :
https://github.com/Schrubitteflau/WireFish/

Il existe un script d'installation **setup.sh**, qui fonctionne bien sous **Linux** et sur **MacOS**. Pour **Windows**, il faudra procéder à une installation manuelle.

Wirefish s'appuie sur un environnement virtuel Python, qui est un contexte d'exécution Python isolé du système global. Cela signifie que l'ensemble des dépendances sera installé dans cet environnement virtuel (**venv**), et le système ne sera pas pollué de dépendances globales inutiles qui pourraient d'ailleurs potentiellement entrer en conflit avec d'autres.
Le script d'installation se contente donc d'automatiser la création de cet environnement virtuel, l'installation des dépendances, effectue des vérifications et ajoute notamment le she-bang à **main.py** pour s'assurer qu'il soit bien exécuté dans le contexte de l'environnement virtuel.

setup.sh attend un paramètre : **install** ou **reinstall**. Le paramètre **install** n'effectuera l'installation que si le dossier du venv n'existe pas déjà, alors que **reinstall** va supprimer le dossier du venv s'il existe, avant de procéder à l'installation.

Attention toutefois, **Wirefish** n'a été testé que sous Linux, plus précisément une distribution **Ubuntu 20.04**, et des problèmes ont été rencontrés avec **MacOS**.

Documentation fonctionnelle

Fonctionnement global

Lorsqu'on exécute notre programme depuis la ligne de commande, diverses combinaisons de paramètres sont possibles, selon l'action sélectionnée (1er paramètre). Tout ceci a pu être géré efficacement et simplement à l'aide du module **argparse**, intégré dans **Python**. Celui-ci permet de s'assurer que les combinaisons de paramètres sont viables, que ceux qui sont requis ont bien une valeur, d'afficher des messages d'aide, etc :

```
$> ./main.py
usage: main.py [-h] {show-interfaces,print-packets,sniff-data} ...
main.py: error: the following arguments are required: command_action
```

Pour la suite de cette documentation, nous allons nous focaliser sur l'action **sniff-data** et son fonctionnement, pour voir des exemples d'utilisation, se référer à l'annexe. Voici les paramètres qu'elle accepte :

```
$> ./main.py sniff-data -h
usage: main.py sniff-data [-h] -I INTERFACES -M MODULES

optional arguments:
  -h, --help            show this help message and exit
  -I INTERFACES, --interfaces INTERFACES
                        A comma-separated list of the interface(s) to listen on, or the value 'ALL'. For example : 'wlan0,lo,eth1'
  -M MODULES, --modules MODULES
                        A comma-separated list of the modules to use. For example : 'http.post_credentials,ftp.credentials'
```

Pour charger les modules, on se base sur leur nom qui correspond au chemin d'import **Python**. Par exemple, un module nommé *http.post_credentials* sera importé depuis une classe nommée *Module* définie dans un fichier *post_credentials.py* présent dans un dossier *http*, lui-même dans le dossier racine des modules nommé *sniff_modules*. Pour charger un module, on se contente donc d'importer dynamiquement et d'instancier la classe qui correspond, à l'aide de la fonction native **__import__()**. Le code correspondant est situé dans *actions/sniff_data.py*.

Pour spécifier les modules à charger, on utilisera le paramètre **–modules**, ou son équivalent **-M**, qui attend comme valeur le nom de chaque module à charger, séparés par une virgule. Si le chargement d'un module échoue, la raison sera affichée et le programme n'utilisera simplement pas ce module :

```
$> ./main.py sniff-data -I wlp164s0,lo -M http.post_credentials,ftp.credentials,ftp.transfer_files
Sniffing interface(s) ['wlp164s0', 'lo'] using filter '<no filter>' and module(s) ['http.post_credentials', 'ftp.credentials', 'ftp.transfer_files']
Loading module http.post_credentials : Success
Loading module ftp.credentials : Can't instantiate abstract class Module with abstract methods on_receive_packet
Loading module ftp.transfer_files : Success
```

Ici, l'erreur indique que le module *ftp.credentials* est invalide car il n'implémente pas correctement la méthode **on_receive_packet()**, ce module sera donc ignoré. En effet, un module doit, pour être valide, implémenter la classe abstraite *AbstractBaseModule*, ce qui signifie implémenter ses champs (propriétés et méthodes) abstraits. Il est possible de dériver *AbstractBaseModule* pour créer des bases de modules plus spécifiques, comme c'est le cas avec les modules HTTP qui héritent de *AbstractHTTPModule*, afin de fournir des fonctionnalités communes à tous les modules qui opèrent au niveau du protocole HTTP.

Spécifier les modules à utiliser, c'est une chose, mais encore faut-il indiquer sur quelles interface réseau écouter. C'est à ça que sert le paramètre **–interfaces**, ou **-I**, qui accepte les noms des interfaces (que l'on peut obtenir en invoquant l'action **show-interfaces**) séparés par des virgules, ou bien la valeur spéciale *ALL* qui signifie « toutes les interfaces ».

Le code du projet est assez lisible et, bien que manquant de commentaires, il est facile de s'y retrouver et de le comprendre simplement grâce aux noms des variables et fonctions qui sont explicites. Nous n'allons donc pas détailler l'ensemble du code dans la suite de cette documentation, mais plutôt présenter chacun des 4 modules développés à ce jour.

Présentation des modules

Tout d'abord, il est bon de noter que les quatre modules développés traitent deux protocoles différents : **FTP** et **HTTP**. Ces deux protocoles sont très courants, très connus, très utilisés, très documentés et surtout, ils sont basés sur du texte.

ftp.credentials

Ce module est le premier développé et a un peu servi de Proof Of Concept pour la suite du projet. Il intercepte simplement les noms d'utilisateurs et mots de passe transmis d'un client vers un serveur, qui sont sous la forme de commandes **USER** <*username*>, et **PASS** <*password*>.
Pour faciliter le traitement des payloads FTP, nous avons développé une class *FTPPacketParser*, qui est ici utilisée de cette façon :

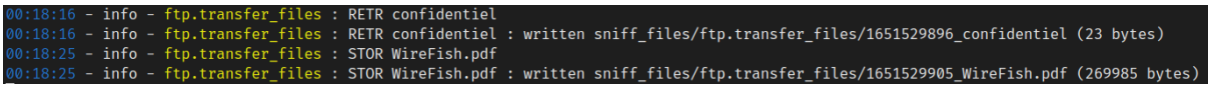
```
ftp_packet_parser = FTPPacketParser(packet=packet)
if ftp_packet_parser.is_command("USER"):
    self.log_message("FTP username : %s " % (
        self.to_str_safe(ftp_packet_parser.get_parameters())
))
```

Voici un exemple de retour de cette commande avec la connexion à un serveur **FTP** :

```
00:03:10 - info - ftp.credentials : FTP username : ftpuser
00:03:10 - info - ftp.credentials : FTP password : passw0rd
```

ftp.transfer_files

Ce module permet de détecter le transfert de fichiers par le biais du protocole **FTP**, que ce soit en envoi ou en téléchargement, et d'effectuer une copie de ces fichiers en local :



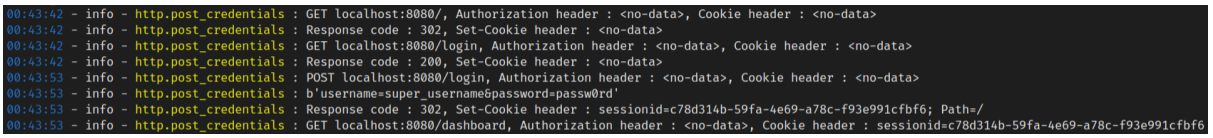
La création de ce module s'est tout de suite avérée est plus complexe qu'on ne l'aurait pensé, puisqu'il faut commencer à interpréter les paquets en plus de les décoder. Il existe deux modes différents pour le transfert de données avec FTP : le mode passif et le mode actif. Étant donné que les données ne sont pas transmises sur le port d'écoute du serveur (21 généralement) mais bien sur un autre port ouvert pour l'occasion, cela peut poser des soucis au niveau de la configuration du pare-feu chez le client ou le serveur. Lorsque le mode passif est utilisé, le client va se connecter à un port indiqué par le serveur, pour que les données soient transmises (soit dans un sens, soit dans l'autre), avant que la connexion ne soit terminée. L'avantage de cette méthode est que le client initie la connexion, donc son pare-feu ne devrait pas empêcher celle-ci. Le mode actif correspond à l'inverse, c'est le serveur qui se connecte au client, sur un port haut et non utilisé, par exemple 34567. Attention, ce n'est pas parce que le client se connecte au serveur que c'est le client qui va envoyer les données. Voici le processus complet :

- Le client envoie une requête **PASV** afin d'ouvrir une connexion passive
- Le serveur répond « très bien, connecte-toi à l'IP <x.x.x.x> au port x »
- Le client initie la connexion en arrière plan
- Le client effectue une requête FTP (sur le port d'écoute du serveur) qui nécessite un transfert de données. Par exemple, **RETR fichier.txt**
- Le serveur va envoyer les données à travers la connexion passive précédemment établie
- Le client et le serveur peuvent toujours communiquer sur le port du serveur **FTP**, et le serveur indique au client lorsqu'il a terminé d'envoyer les données. La connexion passive est alors coupée et le client a téléchargé **fichier.txt**.

Puisque le mode passif est le plus utilisé, nous avons choisi de le prendre en charge au détriment du mode actif, bien qu'une bonne partie du code créé pourrait être réutilisée. Le code qui gère toute cette logique se trouve dans *util/packet_analyze/ftp.py*.

http.post_credentials

Rien de bien sorcier ici, on analyse chaque paquet à la recherche de requêtes ou de réponses **HTTP** et on extrait les éléments les plus intéressants : cookies et tokens d'authentification dans les en-têtes, et données de formulaire dans le corps des requêtes. Voici une capture d'écran qui illustre bien un parcours utilisateur (sur l'application de démo située dans *demo/file-sharing*) :



Dans un premier temps, l'utilisateur accède à la racine, mais il n'est pas connecté donc est redirigé vers **/login**. Il remplit et soumet un formulaire via la méthode **POST**, on voit que son nom d'utilisateur est **super_username** et que son mot de passe est **passw0rd**. Le serveur redirige l'utilisateur vers **/dashboard** en lui envoyant un cookie de session **sessionid**.

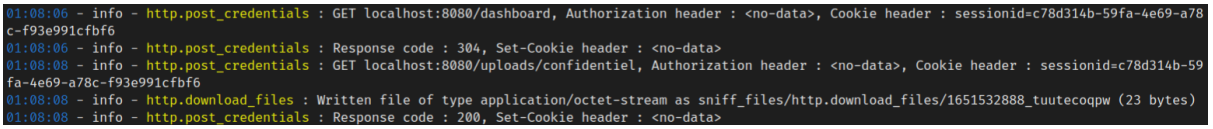
Par chance, **Scapy** fournit déjà des fonctionnalités pour analyser les requêtes et réponses **HTTP**, ce qui nous a facilité la tâche. Nous avons simplement dû gérer manuellement la détection de payloads correspondant au protocole **HTTP**, à l'aide d'expressions régulières :

```
def _guess_payload_class(self, payload) -> PayloadClass:
    """ Decides if the payload is an HTTP Request or Response, or something else
"""
    try:
        crlfIndex = payload.index("\r\n".encode())
        req = payload[:crlfIndex].decode("utf-8")
            if http_request_regex.match(req):
                return PayloadClass.HTTP_REQUEST
            elif http_response_regex.match(req):
                return PayloadClass.HTTP_RESPONSE
    except:
        pass
        return PayloadClass.UNKNOWN
```

Après quelques manipulations supplémentaires, on obtient des instances de *HTTPRequest* ou *HTTPResponse*, classes fournies par **Scapy**, et il nous suffit alors d'accéder à des propriétés comme **request.Authorization**, **request.Host**, **response.Set_Cookie**, etc.

http.download_files

Le dernier module en date, **http.download_files**, permet de conserver une copie locale de tous les fichiers téléchargés par le client, selon certaines conditions. Le contenu d'une réponse **HTTP** peut être très facilement accédé à l'aide des fonctions fournies par **Scapy** et de la classe *HTTPResponse*.
Il nous a suffit d'utiliser la valeur de l'en-tête *Content-Type* de la réponse pour déterminer si on souhaite ou non faire une copie du fichier en local. Par manque de temps, nous n'avons pas eu le temps de développer correctement ce module et celui-ci ne fonctionnera malheureusement pas pour les fichiers transmis sur plusieurs paquets, à cause de leur taille trop importante. De plus, le lien entre requête et réponse n'est pas effectué, ce qui fait que l'on a pas de certitude quant à la source du fichier en question, on perd donc notamment son nom, c'est pourquoi il est vivement recommandé de garder une trace des requêtes effectuées :



On comprend ici que l'utilisateur accède à **/dashboard**, puis qu'il télécharge un fichier **/uploads/confidentiel**. Celui-ci est intercepté par ce *ftp.download_files* et est enregistré localement sous le nom de *1651532888_tuutecoqpw*, qui est la combinaison du timestamp et d'une chaîne aléatoire.

Choix des technologies

Le langage **Python3** a été choisi pour plusieurs raisons, qui sont les mêmes qui expliquent sa popularité dans le monde de la cybersécurité d'une manière générale.

Tout d'abord, il s'agit d'un langage interprété, ce qui signifie qu'il suffit d'avoir installé l'interpréteur pour pouvoir exécuter le programme. Pas besoin de compiler quoi que ce soit et de s'intéresser aux spécificités des différentes machines pour que cela fonctionne bien.

La seconde raison est sa syntaxe claire et concise. C'est un des langages les plus simples à apprendre, et qui a un très bon ratio de quantité de code écrit par action effectuée. Pour résumer, on peut, en seulement quelques lignes, programmer des comportements assez poussés, ce qui également grâce à la quantité astronomique de bibliothèques qui existent. En s'armant des bonnes bibliothèques, quelqu'un travaillant dans le domaine de la cybersécurité peut rapidement créer des outils répondant à un besoin très spécifique. Par exemple, on peut imaginer mettre en place des simulations d'attaque, créer des exploits complexes, afficher graphiquement de grandes quantités de données, et bien d'autres encore. C'est pour cela que ce langage est si populaire dans la cybersécurité, ainsi que dans d'autres domaines comme la data analyse. On pense également au fait que des outils très puissants et utilisés ont été programmés en **Python**, comme l'outil de forensique **volatility**.

L'une de ses bibliothèques se nomme **Scapy**, qui illustre parfaitement le paragraphe précédent. Il est par exemple possible, en seulement 1 ligne, de créer un paquet de A à Z, en renseignant des informations relatives à plusieurs couches du modèle OSI (voir https://scapy.readthedocs.io/en/latest/usage.html#stacking-layers).

Scapy est un programme Python qui peut être utilisé comme une bibliothèque. Il permet de forger et de disséquer des paquets, pour nous permettre d'en faire ce que l'on veut. C'est donc là sa principale force, ce qui le rend spécial d'après la documentation, puisqu'il ne s'agit d'un outil, d'un framework sur lequel s'appuyer pour créer nos propres outils, donc les seules limites sont notre imagination.

Scapy n'interprète pas les paquets, il nous fournit juste des fonctions et des manières simples pour nous permettre de les manipuler comme on le souhaite, et ce en quelques lignes seulement ! Rien ne nous empêche donc de faire des expérimentations et d'envoyer des paquets invalides par exemple, ce qui serait impossible avec des outils spécialisés qui ont un objectif bien particulier.

Par ailleurs, ça fait toujours plaisir d'apprendre que cet outil a été créé par un ingénieur en sécurité informatique français, **Philippe Biondi**.

Pour plus de détails concernant les possibilités offertes par **Scapy**, se rendre sur la documentation officielle (https://scapy.readthedocs.io/en/latest/introduction.html).

Difficultés rencontrées

Tout au long du développement de **Wirefish**, nous avons été confrontés à divers problèmes auxquels il a fallu trouver une solution appropriée.

Dans l'état actuel des choses, on peut remarquer que certains modules ne fonctionnent pas correctement lorsque de nombreuses données transitent sur le réseau dans un court délai, par exemple lors d'un transfert de fichier volumineux. Après avoir effectué des tests isolés et avoir parcouru certains forums, il s'avère que **Scapy** lui-même souffre de problèmes de performances sous certaines configurations, la conséquence étant la non-capture de certains paquets, pouvant donc engendrer des résultats incohérents ou tout simplement de la perte de données et donc des fichiers corrompus. Pour résoudre partiellement ce problème, nous avons indiqué à Scapy que nous souhaitions utiliser la **libpcap** pour capturer les paquets, qui est programmée en C et donc compilée, ce qui offre de bien meilleures performances que la manière par défaut utilisée par **Scapy** pour l'interception des paquets. L'inconvénient est que cela ajoute une contrainte à notre environnement, puisque cette bibliothèque doit être installée sur le système.
Cependant, ce problème subsiste et il est très difficile de remonter à la source de celui-ci.

Un autre problème quant à lui est plus lié à la méthodologie et à la manière dont est conçu **Wirefish**. Il est plutôt simple d'analyser le payload d'un paquet isolé, surtout s'il s'agit d'un protocole basé sur du texte auquel cas il s'agit généralement d'opérations impliquant des chaînes de caractères et des expressions régulières, puis un peu de logique pour interpréter les résultats.
En revanche, tout cela devient bien plus complexe lorsqu'il s'agit d'interpréter les différents payloads à la suite et que ceux-ci suivent une logique, celle du protocole applicatif en question. Par exemple, il est simple d'analyser une requête HTTP isolée, il suffit d'extraire les informations intéressantes et le tour est joué. Par contre, cela devient plus complexe si on s'attèle à l'analyse d'un trafic FTP, puisque la connexion est bidirectionnelle et il ne s'agit pas simplement d'un schéma requête-réponse. Parfois, le serveur va envoyer des données de lui-même (par exemple, un code timeout), mais un exemple plus parlant est le cas d'un transfert de fichier. Une fois le client connecté au serveur, certains transferts de données, comme les transferts de fichier (dans un sens comme dans l'autre), se feront sur un autre port, un port dédié et utilisé uniquement pour transférer les données en question. Après quoi, la connexion sera coupée et ce port sera fermé.
Il faut donc être capable de comprendre le protocole analysé et programmer ce cas précis, pour pouvoir se dire « dès maintenant je vais intercepter les données sur ce port là et les stocker localement ».

On en vient donc à une dernière difficulté, qui est de savoir quelles sont les limites d'un module. Au fur à mesure qu'on développe des fonctionnalités concernant un protocole précis, on se retrouve finalement à petit à petit programmer une sorte de « client muet », un client capable de comprendre et d'interpréter un protocole mais pas de communiquer. Cela pose des problèmes au niveau de la lisibilité du code qui devient rapidement un champ de bataille, et les fonctionnalités de base fournies par **Wirefish** ne suffisent plus.

Axes d'amélioration

Après analyse des problèmes rencontrés et des limitations de notre outil, différentes perspectives d'amélioration se sont dessinées.

Dans un premier temps, nous avons pensé qu'il pourrait être utile de fournir une option permettant de rediriger les logs vers un fichier de sortie en plus de les afficher dans la console, même si cela pourrait être fait directement à partir du terminal utilisé.

Fournir une interface graphique pourrait également être utile, mais étant donné qu'elle servirait simplement d'intermédiaire avec la ligne de commande, et donc qu'aucune fonctionnalité ne serait ajoutée, nous avons pensé que ce n'était pas très intéressant.

Une autre amélioration, bien plus intéressante, serait d'ajouter la prise en charge d'un fichier de configuration externe, dans un format texte et facilement lisible et éditable, et nous avons directement pensé au format **YAML**. Celui-ci serait utilisé dans un premier temps afin de rendre les différents modules paramétrables, nous pensons par exemple au module **ftp.transfer_files** pour lequel il pourrait être pertinent de spécifier une taille maximum des fichiers à sauvegarder, ou encore de choisir de ne faire une copie locale que de ceux portant une certaines extension (par exemple, ignorer les *.jpg* mais sauvegarder les *.zip* et *.pdf*). Le passage de ces paramètres aux modules ainsi que la gestion des valeurs par défaut serait géré par **Wirefish**.

De plus, pour des questions d'optimisation, il faudrait offrir la possibilité à chaque module de spécifier dans quelles conditions opérer. Pour l'instant, la totalité des paquets est traitée par chaque module, qui effectuent des traitements rapides et choisissent quoi en faire. Il faudrait pouvoir déléguer cette partie de « traitement préliminaire » à **Wirefish**. Par exemple, peut-être qu'un certain module pourrait n'être intéressé que par les paquets destinés à une certaine IP ou à un certain port. Déjà, le code des modules serait plus clair, mais notre outil pourrait effectuer ce premier filtrage en amont, en utilisant par exemple des fonctions plus bas niveau fournies par **Scapy**, ce qui améliorerait les performances.

Enfin, probablement l'amélioration la plus intéressante de toutes, celle qui consiste en la prise en charge de fichiers de capture réseau. Ceci aurait l'avantage de ne pas risquer de perdre des paquets et donc des informations sensibles qui pourraient s'y trouver, à cause des problèmes de performance évoqués précédemment, ou bien à cause de bugs ou de crashs de l'outil. Dans le cadre d'une attaque **Man In The Middle**, la bonne méthodologie à utiliser serait d'enregistrer une capture réseau, puis de passer celle-ci à notre outil. De cette manière, même s'il y a un problème avec **Wirefish**, les potentielles données sensibles ne sont pas perdues et se trouvent dans le fichier de capture. Il existe la possibilité de lire différents types de fichiers de capture réseau avec Scapy, donc peu de travail additionnel serait nécessaire, pour de nombreux avantages. Si nous devions continuer le développement de **Wirefish**, nous nous tournerions sans aucun doute vers l'ajout de cette fonctionnalité.

Conclusion

Pour conclure ce rapport, nous allons parler de ce que ce projet nous a apporté d'un point de vue personnel.

Apports personnels - Antoine

J'ai vraiment apprécié développer ce projet, puisque celui-ci a amené des pistes de réflexion intéressantes, telles que décrites dans ce rapport. J'ai également eu le sentiment de travailler sur quelque chose apportant une valeur ajoutée, au lieu de réitérer ou de réinventer, en moins bien, quelque chose qui existe déjà.

En plus de cela, je suis monté en compétences d'un point de vue technique, puisque j'ai amélioré mes compétences en langage **Python**, me permettant d'écrire du code plus concis, plus lisible et moins sujet à d'éventuels bugs.
De plus, j'ai développé une compréhension basique de la bibliothèque **Scapy** et ai pris connaissance de l'ensemble des possibilités qu'elle offre, ces connaissances vont très probablement me resservir un jour, que ce soit dans un cadre professionnel ou personnel. Quand je vois les possibilités qu'elle offre, je me dis qu'en passant un certain temps à découvrir ses fonctionnalités, il y aurait la possibilité de créer des outils vraiment inédits pour des cas d'utilisation que l'on imagine à peine.

J'ai également trouvé que le développement de modules est intéressant dans le sens où il nous oblige à acquérir une compréhension plus ou moins approfondie du protocole en question, ce qui ne peut être qu'utile et nous pousse à remettre en question nos connaissances.

Enfin, j'aimerais noter que c'est définitivement un projet que je vais mettre en avant, même si je ne continuerais probablement pas son développement, puisque les technologies utilisées et la nature même de ce projet devraient plaire à d'éventuels recruteurs en cybersécurité.

Apports personnels - Benjamin

Pour ma part, ce projet m'a apporté tout d'abord d'un point de vu humain. Je suis arrivé seulement cette année au sein d'Ynov et je ne connaissais personne dans ma promotion. J'ai pu faire la connaissance d'Antoine, une très belle personne qui a su manager l'équipe et garder son sérieux tout au long du projet. Un grand merci à lui, il a pu m'aider lorsque j'en avais besoin. Ce projet pour moi s'est soldé par une très belle rencontre.

De plus ce projet m'a aussi apporté techniquement, j'avais de légères bases en ce qui concerne le langage Python au début du projet j'ai pu les développer et me documenter et ces compétences de programmation ne peuvent qu'être utiles à un moment donné dans la suite de mon parcours.

En ce qui concerne la bibliothèque **Scapy** je ne connaissais pas son existence, elle s'avère très utile et elle nous a permis de gagner du temps en utilisant des fonctions déjà développées.

Certains utilisent tous les jours l'outil **WireShark** dans leur quotidien mais ils ne se rendent pas forcément compte de la programmation complexe qu'il doit y avoir derrière. Avec ce projet ne représentant qu'une infime partie de ce dernier, je n'ose imaginer tout le travail pour obtenir cet outil.

Bibliographie

Code source du projet : https://github.com/Schrubitteflau/WireFish

Logiciels, outils, bibliothèques mentionnés :
Documentation officielle de la bibliothèque **Scapy** :
https://scapy.readthedocs.io/en/latest/index.html
Langage de programmation **Python** : https://www.python.org/
Logiciel **Wireshark** : https://www.wireshark.org/
Langage de programmation **Lua** (ici, scripting **Wireshark**) : https://www.lua.org/
Logiciel **tcpdump** et bibliothèque **libpcap** : https://www.tcpdump.org/
Logiciel **NetworkMiner** : https://www.netresec.com/?page=networkminer
Logiciel **volatility** : https://github.com/volatilityfoundation/volatility
Format de fichier **YAML** : https://yaml.org/

Quelques tutoriels, articles et discussions de forum qui nous ont été utiles :
http://sdz.tdct.org/sdz/manipulez-les-paquets-reseau-avec-scapy.html
https://stackoverflow.com/questions/71936069/scapy-dns-requests-and-responses
https://fr.quora.com/Pourquoi-Python-est-il-utilis%C3%A9-dans-la-cybers%C3%A9curit%C3%A9
https://www.varonis.com/fr/blog/capture-de-paquets
https://www.dnsstuff.com/fr/outils-detection-paquets
https://geekflare.com/fr/network-packet-analyzers/
https://github.com/secdev/scapy/issues/2608
https://github.com/secdev/scapy/issues/1789
https://www.deskshare.com/resources/articles/ftp-how-to.aspx
https://en.wikipedia.org/wiki/List_of_FTP_server_return_codes

Quelques modules **Python** mentionnés :
https://pypi.org/project/netifaces/
https://pyinstaller.org/en/stable/
Logo (libre de droit) du poisson :
https://freepngimg.com/png/85595-largemouth-logo-art-bass-fishing-free-download-png-hd

Annexes

```
$> ./main.py -h
usage: main.py [-h] {show-interfaces,print-packets,sniff-data} ...

positional arguments:
  {show-interfaces,print-packets,sniff-data}
                        Shows the help of the actions
    show-interfaces     Shows all available interfaces and their details
    print-packets       Starts listening the specified interface(s) and prints the summary of the intercepted packets
    sniff-data          Starts listening the specified interface(s) and analyze the packets with the specified module(s)

optional arguments:
  -h, --help            show this help message and exit
```

Capture d'écran du manuel d'aide principal

```
$> ./main.py show-interfaces
Name              MAC                IPv4             IPv6
---------------   -----------------  -------------    ------------------------
lo                00:00:00:00:00:00  127.0.0.1        ::1
enx00e04c684b98   00:e0:4c:68:4b:98
wlp164s0          94:e2:3c:a9:d2:e7  10.188.120.200   fe80::9b6e:ed07:169c:8503
docker0           02:42:30:03:1a:63  172.17.0.1
br-fdb00f49326a   02:42:91:9b:9f:41  172.18.0.1       fe80::42:91ff:fe9b:9f41
veth3aee7c3       2e:75:77:87:f8:f9                   fe80::2c75:77ff:fe87:f8f9
veth82626c8       f6:d4:46:18:a4:5d                   fe80::f4d4:46ff:fe18:a45d
vmnet1            00:50:56:c0:00:01  172.16.91.1      fe80::250:56ff:fec0:1
vmnet8            00:50:56:c0:00:08  192.168.239.1    fe80::250:56ff:fec0:8
```

Capture d'écran de l'action show-interfaces

```
$> ./main.py print-packets --interfaces lo,wlp164s0
Ether / IP / UDP 10.188.147.134:37773 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:37773 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:37773 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.141.181:54915 > 10.188.255.255:54915 / Raw
Ether / IP / UDP 10.188.147.134:53565 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:42146 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:36010 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:37332 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:37332 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:37332 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:33658 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:54554 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:36417 > 239.255.255.250:1900 / Raw
Ether / IP / UDP / DNS Ans "10.188.235.183"
Ether / IPv6 / UDP / DNS Ans "10.188.235.183"
Ether / IP / UDP 10.188.147.134:45189 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:45189 > 239.255.255.250:1900 / Raw
Ether / IP / UDP 10.188.147.134:45189 > 239.255.255.250:1900 / Raw
Ether / ARP who has 169.254.255.255 says 10.188.217.217 / Padding
```

Capture d'écran de l'action print-packets sur les interfaces lo (loopback) et wlp164s0 (Wi-Fi) afin de tester le bon fonctionnement de **Wirefish**

```
01:32:35 - info - http.post_credentials : GET localhost:8080/, Authorization header : <no-data>, Cookie header : websitez_mobile_detector=%7C0%7C008bc52a8dfb41398644c8d74ba5dc93
01:32:35 - info - http.post_credentials : Response code : 302, Set-Cookie header : <no-data>
01:32:35 - info - http.post_credentials : GET localhost:8080/login, Authorization header : <no-data>, Cookie header : websitez_mobile_detector=%7C0%7C008bc52a8dfb41398644c8d74ba5dc93
01:32:35 - info - http.post_credentials : Response code : 200, Set-Cookie header : <no-data>
01:32:35 - info - http.post_credentials : GET localhost:8080/favicon.ico, Authorization header : <no-data>, Cookie header : websitez_mobile_detector=%7C0%7C008bc52a8dfb41398644c8d74ba5dc93
01:32:35 - info - http.post_credentials : Response code : 404, Set-Cookie header : <no-data>
01:32:42 - info - http.post_credentials : POST localhost:8080/login, Authorization header : <no-data>, Cookie header : websitez_mobile_detector=%7C0%7C008bc52a8dfb41398644c8d74ba5dc93
01:32:42 - info - http.post_credentials : b'username=test&password=test'
01:32:42 - info - http.post_credentials : Response code : 302, Set-Cookie header : sessionid=c036caac-f03d-401e-815b-5974be41181a; Path=/
01:32:42 - info - http.post_credentials : GET localhost:8080/dashboard, Authorization header : <no-data>, Cookie header : websitez_mobile_detector=%7C0%7C008bc52a8dfb41398644c8d74ba5dc93; sessionid=c036caac-f03d-401e-815b-5974be41181a
01:32:42 - info - http.post_credentials : Response code : 200, Set-Cookie header : <no-data>
01:32:46 - info - ftp.credentials : FTP username : ftpuser
01:32:46 - info - ftp.credentials : FTP password : passw0rd
01:32:51 - info - ftp.credentials : FTP username : ftpuser
01:32:51 - info - ftp.credentials : FTP password : passw0rd
01:32:51 - info - ftp.transfer_files : RETR 87-preproceedings.pdf
01:32:51 - info - ftp.transfer_files : RETR 87-preproceedings.pdf : written sniff_files/ftp.transfer_files/1651534371_87-preproceedings.pdf (459877 bytes)
```

Capture d'écran d'un exemple de retour console avec les quatre modules activés