Sniff'eirb, sniffer intelligent

Rapport final

Nicolas Verdier Nicolas Retrain Gabrielle Schmidt
Benjamin Vandenberghe

Responsable : M. Pierre Lalet & M. Mathieu Blanc

12 janvier 2013

Table des matières

Introduction

Dans le cadre du projet de sécurité, nous avions le choix sur notre sujet.

Chapitre 1

Les besoins et l'état de l'art

1.1 L'état de l'art

Il existe un grand nombre de logiciels et de librairies qui proposent des fonctionnalités de captures réseaux de bas niveau et d'analyse de paquets (d'un point de vu utilisateur final). Nous pouvons notamment citer libpcap, Scapy, Winpcap ou encore tcpdump. Ces outils ont été un modèle d'inspiration pour tenter de réaliser notre propre *Sniffer*[1] réseau.

1.2 Nos objectifs

1.2.1 Premières étapes

L'objectif premier était tout d'abord de retrouver les fonctionnalités principales de Wireshark, comme par exemple afficher une liste des paquets circulant sur le réseaux ainsi que les informations s'y rapportant (timestamp, adresse IP source, adresse IP destination, port, protocole...), tout en apportant une solution différente et plus simple d'utilisation à un utilisateur néophyte. Ainsi nous avons défini trois premiers objectifs :

- Récupérer les communications[2] en clair d'un réseau.
- Stocker les paquets de cette communication.
- Afficher les différents flux[3] possibles de cette communication.

1. Terme anglicisé désginant un dispositif permettant d'analyser le trafic d'un réseau
2. Une communication est définie par un échange réseau entre deux paires (Adresse IP ; Port).
3. Un flux est défini comme un assemblage possible des paquets d'une communication.

1.2.2 La valeur ajoutée de Sniff'eirb

Le but de ce projet n'étant pas de reproduire un Wireshark, nous devions y apporter des atouts significatifs. Plusieurs fonctionnalités ont donc été implémentées :

L'arbre des possibilités pour les flux TCP : Lors de la capture des paquets circulant sur le réseau pour une même communication, des événements inatendus, comme la répétition de paquets ou encore la perte de certains, pouvaient survenir. La capture de tous les paquets permet donc de connaître les différentes façons de les réassembler pour obtenir un flux décrivant la communication.

La gestion du protocole HTTP : La reconstruction des pages webs visitées par un utilisateur sur le réseau était le fil conducteur du projet. L'analyse des paquets circulant sur le réseau est le principal outil pour arriver à cette fin.

1.2.3 Les idées abandonées ou non implémentées

En parallèle, beaucoup d'idées durent être abandonnée du fait d'un manque de temps et de connaissances pour les traiter. Voici une liste non exhaustive de ces idées :

Chapitre 2

L'architecture

2.1 L'architecture générale

Nous avons choisi dès le départ de suivre une architecture de type MVC, et nous nous y sommes tenu tout au long du projet. Afin de nous faciliter le découpage du code et de définir nos principales interfaces, nous avons conçu au démarrage du projet une architecture globale. Elle est représentée dans la figure 2.1.

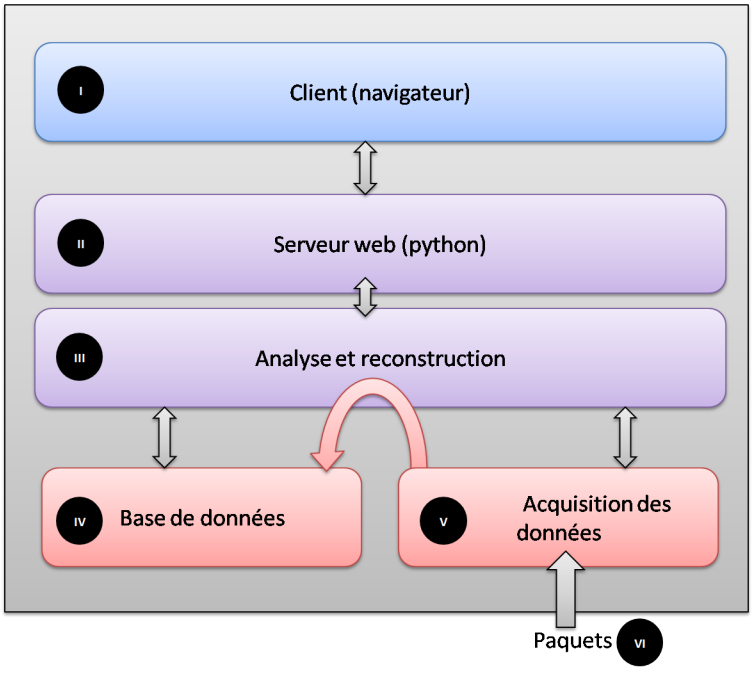

FIGURE 2.1 – Architecture de Sniffeirb

Nous avons choisi d'utiliser le langage Python pour notre projet, en partie pour ses nombreuses bibliothèques, y compris pour les captures réseaux. De plus, nous avions tous envie d'approfondir nos connaissances dans un langage de programmation autre que le C, C++ ou Java.

2.2 Explication du rôle des différentes couches

2.2.1 Client

Le client est soit le terminal dans lequel on lance le programme, soit un navigateur web. Nous n'offrons pas tout à fait les mêmes services sur ces deux interfaces, ceci est lié à notre volonté d'avoir une interface utilisateur simple. De ce fait, le terminal permet seulement de lancer une capture, ou de lancer le programme avec différentes options (lancement du serveur web, choix d'une ancienne capture à charger, choix du nom de la nouvelle capture, chargement d'un fichier pcap..). L'interface web permet de lancer une capture, d'afficher les flux des différentes communications, de reconstruire les documents et de changer certains paramètres.

Le navigateur utilise des requêtes AJAX afin d'interagir avec le programme. Dans le but de fournir un programme utilisable sur le plus grand nombre de navigateur possible, nous avons utilisé des frameworks reconnus dans le milieu. Nous avons donc choisi d'utiliser JQuery pour nos requêtes, Bootstrap pour la mise en page et le CSS, et DataTables pour la présentation des flux sous forme de tableau.

2.2.2 Serveur web

Le serveur web est l'interface entre les requêtes HTML provenant du client et les différentes actions possibles du sniffer. Après avoir lancé une action, le serveur renvoie si besoin est des données au navigateur au format JSON. Nous n'avons pas utilisé de serveur déjà fourni dans un framework Python, trop lourd pour l'utilisation que nous en faisons. Nous avons choisi d'utiliser JSON pour plusieurs raisons. C'est un format de données structuré, moins verbeux que le XML, et qui est très bien intégré avec le JavaScript et le Python. De plus le format JSON est identique au format BSON utilisé par la base de données.

2.2.3 Analyse et reconstruction

Ce composant analyse une communication pour déterminer son protocole, reconstruit les flux possibles et reconstruit les documents qui transitent en donnant un pourcentage de validité. Nous assurons seulement la reconstruction des documents sur le protocle HTTP conformément à nos spécifications fonctionnelles.

2.2.4 Base de données

La base de données sert à stocker les paquets réseaux regroupés par communication. Stocker les données nous permet de rejouer des échanges de données (soit pour le débugage, soit suite à une amélioration algorithmique du projet). Nous avons choisi d'utiliser la base NoSQL Mongo pour ses performances et sa souplesse. Elle s'utilise aisément en Python grâce au module Pymongo. Il faut donc installer mongodb et lancer le service mongod pour pouvoir utiliser le sniffer.

2.2.5 Acquisition des données

Ce composant récupère les données réseaux, ou lit un fichier pcap, et les met dans la base de données. Les paquets sont regroupés par communication. Nous utilisons le module Scapy de Python afin de récupérer les paquets. Il est de plus au même niveau que la bibliothèque Libpcap ou WinPcap, ce qui facilite la capture des paquets réseau.

2.2.6 Paquets

Ceci représente le réseau, et les paquets qui circulent. En paramétrant la carte réseau en mode monitor, l'utilisateur peut alors écouter tout le réseau et analyser les paquets sniffés.

Chapitre 3

L'implémentation

3.1 La base du sniffer : la récupération des paquets et leur stockage

La partie principale de notre projet est de capturer un flux réseau, nous nous sommes donc appuyés sur la bibliothèque Scapy. Cependant, la fonction de capture réseau fournie par Scapy permet de lancer une capture sur un nombre fixe de paquet ou un temps donné. C'est également à la fin de la capture que l'utilisateur peut récupérer les données.

Dans notre projet, nous voulions pouvoir sniffer le réseau un temps indéterminé, jusqu'à une interaction de l'utilisateur, et traiter les données au fur et à mesure. Nous avons donc repris le code de la fonction "sniff" de Scapy et modifié de telle sorte qu'une fonction de contrôle soit appelée régulièrement. Ainsi, nous pouvons couper et lancer le sniffer de façon évènementielle, et récupérer les données en temps réel.

En parallèle de la capture, nous stockons les paquets capturés dans une base de données. Celle-ci est composée d'une unique collection. Cette collection comporte plusieurs sessions qui permettent d'identifier les différentes captures réalisées. Ensuite, chaque session est composée de communications, elles-mêmes composées de un ou plusieurs documents dont les champs peuvent varier selon le type de paquets. Un document correspond à une communication [1]. Ce qui signifie que l'on ne prend pas en compte les éventuels paquets qui utilisent un autre protocole qu'IP.

Chaque document possèdent certains champs dans tous les cas. Il s'agit des champs :

- adresses IP source et destination
- ports sources et destination
- protocole (TCP, UDP, ICMP ou "other")
- type (IP)

1. Une communication est définie par un échange réseau entre deux paires (Adresse IP ; Port).

– session (un identifiant de type *sess_date*)

et du document paquets. Les documents paquets étant eux-mêmes composés de tous les champs utiles des paquets, le tout au format BSON.

Dans l'exemple ci-dessous, on peut voir un document de notre collection où l'on retrouve tous les élements indiqués, classés par ordre alphabétique.


```
{
        "_id" : ObjectId("50edc0fe10b95715ceddf583"),
        "dport" : 80,
        "dst" : "50.57.168.107",
        "initTS" : 1357758718.609804,
        "packets" : [
                {
                        "ack" : 0,
                        "flags" : "S",
                        "ts" : 1357758718.609804,
                        "seq" : NumberLong("3047389275")
                },
                {
                        "ack" : NumberLong("3368619615"),
                        "flags" : "A",
                        "ts" : 1357758718.736191,
                        "seq" : NumberLong("3047389276")
                },
                {
                        "ack" : NumberLong("3368619615"),
                        "flags" : "A",
                        "ts" : 1357758722.540476,
                        "seq" : NumberLong("3047389276")
                }
        ],
        "proto" : "TCP",
        "session" : "sess_09-01-2013-201054",
        "sport" : 53879,
        "src" : "192.168.1.101",
        "type" : "IP"
}
```


3.2 La reconstruction

Un des principaux objectifs de Sniff'eirb était de rendre l'exploitation des données capturées la plus évidente possible. Nous avons donc décidé de permettre l'affichage des échanges HTTP, notamment celui des pages HTML et des images. Pour ce faire, nous avons commencé par reconstruire les différents flux de données.

3.2.1 Recontruction d'un flux

Comme nous ne pouvions pas implémenter la reconstruction de flux de la même façon pour tous les protocoles de couche 4 (UDP et TCP) et que nous voulions présenter un résultat démonstratif même pour le néophyte, nous nous sommes consacrés au protocole TCP, puis au média HTTP.

Reconnaissance du protocole et du média

Il a donc tout d'abord fallu reconnaître le protocole qui nous intéresse et le média, avant d'en reconstruire le flux. Comme nous l'avons vu dans la partie précédente, le protocole est facilement retrouvable directement dans la base de données (puisqu'il s'agit d'un champ du paquet IP). Pour le média, cela a été plus technique.

Au départ, nous l'avons simplement déterminé grâce au port (source ou destination). En effet, traditionnellement, certains ports sont utilisés pour certains médias (par exemple, 80 pour HTTP). Mais en ne nous concentrant que sur cette information, nous aurions manqué toutes les communications d'un média sur d'autres ports (comme 8080 pour HTTP), qui, d'un point de vue de la sécurité, pouvaient être les plus intéressantes.

Nous avons donc décidé de parser la charge utile du paquet TCP à la recherche d'expression régulière trahissant le HTTP(S), comme les en-têtes $HTTP/numero\_de\_version$ et les mots clés GET ou $POST$ ou des URLs.

La multiplicité des flux possibles

À ce moment-là, nous disposons donc de l'ensemble des paquets appartenant à la même communication ainsi que du média qui correspond. Comme le protocole est TCP, il paraît assez facile de reconstruire le flux : on classe les paquets par numéro de séquence croissant et on concatène les charges utiles.

Cependant, cette démarche n'est pas possible. En effet, il arrive parfois que plusieurs paquets arrivent avec le même numéro de séquence, voire qu'il manque des paquets au milieu de la communication. C'est pourquoi nous avons dû prévoir de reconstruire tous les flux possibles.

La reconstruction de tous ces flux

Pour ce faire, nous avons créé une structure de liste doublement chaînée que nous avons ensuite parcouru pour en tirer tous les flux possibles. Pour remplir cette chaîne doublement chaînée, baptisée $LianaTree$, nous avions simplement parcouru la liste des paquets d'une communication en commençant par celui qui a le plus petit numéro de séquence et nous cherchions tous ses successeurs.

Le successeur d'un paquet ayant pour numéro de séquence la somme de la taille de sa charge utile et du numéro de séquence de son prédecesseur (modulo 2^{32}).

Cependant, nous avions un problème dans certains cas, car le numéro de séquence TCP est pris au hasard parmi 2^{32} possibilités et il était donc pos-

sible d'avoir la fin du flux avant le début (si le numéro de séquence du début de la communication était proche de $2^{32} - 1$). Finalement, nous recherchons simplement les successeurs et prédecesseurs de chaque paquet via le numéro de séquence et la taille de la charge utile.

Une fois cette liste créée, nous devons la parcourir en entier en partant du paquet qui n'a pas de prédécesseur. Si plusieurs paquets n'ont pas de prédécesseur, nous recommençons le parcours autant de fois que nécessaire. Nous réalisons un parcours en largeur de la liste qui découle de ce paquet et nous concaténons les charges utiles au fur et à mesure. À la fin, nous éliminons les doublons et renvoyons le résultat.

Le choix du meilleur flux

Dans le cas où nous aurions plusieurs flux, nous avons décidé de les afficher tous, mais dans l'ordre où ils nous paraissent le plus cohérent. Cette cohérence est définie spécifiquement pour le type de document. Dans le cas de document HTML en texte, nous calculons le pourcentage de balises bien fermées par rapport au nombre total de balises.

3.2.2 Recontruction d'un Document

-reconstruction des pages html (.html et .gzip !) -reconstruction des images

3.3 L'affichage

-création d'un serveur web minimaliste -template html -jquery datatables bootstrap -Ajax

Conclusion

Annexes