Telecom Nancy

Université de Lorraine

RSA

Proxy HTTP

Auteurs :
Vincent Albert

1er mai 2016

Table des matières

1 Présentation du sujet

1.1 L'objectif

L'objectif du projet est de réaliser un proxy transparent relayant les requêtes du navigateur au serveur désiré. Le proxy doit être capable de gérer des requêtes HTTP et HTTPS en IPv4 et en IPv6.

1.2 Les outils utilisés

Le projet a été codé en langage C sous systèmes de type Unix. Afin de faciliter la gestion des versions et le partage des fichiers, le gestionnaire de version git a été utilisé. Wireshark a été utilisé afin de vérifier les échanges TCP et HTTP.

2 Analyse du fonctionnement d'un proxy

2.1 Analyse d'échanges TCP et HTTP sans proxy

Afin de ne pas crouler sous les données et de pouvoir analyser agréablement l'échange, nous avons utilisé le filtre : "(ip.src==192.168.1.76 or ip.dst==192.168.1.76) and (ip.src==193.54.21.201 or ip.dst==193.54.21.201) and tcp" afin de ne récupérer que les données TCP échangées entre le client et le serveur (adresse de telecomnancy.eu). HTTP est une surcouche à TCP, les paquets sont donc aussi affichés.

La requête engendre tout d'abord un three-way handshake (SYN, SYN-ACK, ACK) afin d'initialiser la connexion TCP. Après ces 3 paquets TCP, le premier paquet HTTP est envoyé par le navigateur.

Ce paquet, en plus des données d'un paquet TCP, contient aussi l'en-tête HTTP de la requête. On trouve notamment les informations concernant le type de la requête (GET), sur le navigateur, sur les cookies et enfin sur la page demandée.

Ensuite, de multiples échanges TCP se font à nouveau. Il s'agit de la réponse à la requête en fragments, comme on peut le déduire depuis l'annotation "TCP segment of a reassembled PDU". Ceci nous indique que le paquet contient des données d'un protocole étant une surcouche à TCP (HTTP dans notre cas). Il y a aussi les paquets des ACK envoyés par le navigateur.

Ensuite tous ces segments sont recomposés en un seul paquet HTTP. Ce paquet indique en effet qu'il résulte de la recomposition de 8 autres segments TCP. Ce segment contient à nouveau les données TCP et l'en-tête de la réponse HTTP qui nous indique que la requête a bien été exécutée (200 OK) mais aussi la taille des données de la réponse. Car en effet, le paquet contient en plus les données HTML correspondant à la réponse de la requête.

On remarque que malgré le fait que la requête n'ait été exécutée qu'une seule fois, plusieurs GET sont effectués. Ceci est dû à la réponse HTML, qui importe des fichiers depuis d'autres emplacements du serveur (css, javascript, ...). Le navigateur va donc effectuer des requêtes pour les récupérer afin de pouvoir afficher la page normalement.

2.2 Analyse d'échanges TCP et HTTP avec proxy

Afin d'analyser les échanges avec un proxy, nous avons modifié les paramètres de notre navigateur pour nous connecter au proxy gratuit d'adresse 162.223.88.243. Nous avons aussi modifié le filtre *Wireshark*, étant donné que nous ne contactons plus directement le serveur de destination : "(ip.src==192.168.1.76 or ip.src==162.223.88.243) and (ip.dest==192.168.1.76 or ip.dst==162.223.88.243) and tcp"

La connexion TCP est à nouveau débutée par un three-way handshake, mais cette fois-ci entre le client et le proxy. Ensuite, le navigateur envoie la requête HTTP. N.B. : On remarque la mauvaise qualité de la connexion par les paquets dupliqués.

Ensuite, l'échange de paquets TCP contenant des fragments de la réponse HTTP a de nouveau lieu entre le client et le proxy. Le paquet HTTP est reconstitué avec son en-tête et ses données HTML.

L'échange entre le client le proxy est donc semblable à l'échange entre le client et le proxy. Le serveur proxy reçoit donc la requête vers le serveur, l'exécute, reçoit les données et les envoie de la même manière qu'il les a reçues au serveur web.

2.3 Algorithme général d'un proxy

```
Data: serverSocket, clientSocket, webSocket, requete, response, servAddr
Result: Réception d'une requête d'un client, envoi au serveur et transfert de la réponse au client
while true do
    Attente de la connexion TCP d'un client;
    requete += recevoir(socketClient) ; //On récupère la requête du client
    servAddr = trouverHost(requete) ; //On récupère l'adresse du serveur
    webSocket = créerSocket(servAddr) ; //On crée la socket sur le serveur de destination
    envoyer(requete, webSocket) ; //On envoie la requête au serveur
    connexion(webSocket) ; //On se connecte au serveur
    while données à lire sur webSocket do
        if taille des données == 0 then
            //On ferme la connexion
            close(serverSocket);
            close(clientSocket) ;
        else
            response = lire(webSocket) ; // On lit un morceau de la réponse envoyer(response,
            clientSocket) ; //On transfère la donnée
        end
    end
end
```

Algorithm 1: Algorithme d'un proxy

Toutes les fonctions utilisées ici sont déjà implémentées en C, hormis trouverHost(requete). Nous allons donc nous attacher à expliquer son fonctionnement ici. Nous savons que l'en-tête d'une requête sera toujours de la forme "GET http ://hostname/pageRecherchée.html HTTP/1.1". Notons qu'hostname peut être de la fome www.host.com ou seulement host.com sans que cela n'ait d'incidence. De plus, "pageRecherchée.html" peut être vide. Afin d'isoler la méthode utilisée, nous pouvons donc simplement lire la chaine de la requête jusqu'à rencontrer le premier espace. Ensuite, on peut supprimer la partie "http ://" en incrémentant le pointeur. Puis on peut lire le hostname jusqu'au premier "/". Au final nous aurons donc :

```
Data: requete, hostname, i
Result: trouverHost(requete)
while requete != ' ' do
    requete++ ; //On supprime la méthode de la requête
end
requete += 7 ; //On supprime 'http ://'
i = 0 ; while requete[i] != '/' do
    hostname[i++] = requete[i++]
end
retourner hostname ;
```

Algorithm 2: Fonction trouverHost

3 Développement d'un proxy

3.1 Version standard du proxy

La première version du proxy n'est que l'adaptation de l'algorithme au langage. Il n'accepte qu'un seul client à la fois. Toute autre demande durant le traitement d'un client est ignorée.

Certains morceaux de code redondant ont été transformés en fonction dans les fichiers util.c et socketutil.c du dossier utils. Le premier rassemble les fonctions de traitement de chaînes de caractère et des fichiers de log, tandis que le second regroupe les fonctions de traitement de socket (création de socket, ajout d'un client, ...).

Afin d'avoir un aperçu global du bon fonctionnement du proxy, un affichage exhaustif de son comportement a été réalisé (connexion/déconnexion client, requête reçue, fermeture de la connexion avec le serveur web, réponse envoyée, ...).

On affiche aussi l'adresse Ip et le port de lancement. On remarque qu'étant donné que le serveur attend autant de l'IPv4 que de l'IPv6, son adresse est " : :".

3.2 Version multi-utilisateur

Afin de faciliter les modifications de code dans les deux versions par la suite, de nouvelles fonctions ont été ajoutées à util.c et socketutil.c pour y mettre les morceaux de code communs (initialisation des sockets d'écoute, ...).

Dans cette seconde version, afin de gérer plusieurs clients, nous utilisons un tableau de descripteurs pour les sockets clients et pour les sockets web. Nous initialisons par défaut ces tableaux au nombre maximal de descripteur d'un set de descripteur (soit 1024) étant donné que le select ne pourra pas en gérer plus (on considère que le nombre de descripteurs déjà utilisé pour les sockets d'écoute et les entrée/sortie standard est négligeable). Afin de faciliter la correspondance entre les sockets clients et les websockets, on les associera par le même indice (un client d'indice i attend la réponse de la websocket d'indice i).

3.3 Fichiers de logs

Afin d'avoir un meilleur aperçu de l'utilisation du proxy, les deux versions renseignent des fichiers de logs (qui se trouvent dans le dossier logs).

Le premier fichier (log_visits) enregistre l'adresse IP d'un client qui se connecte au proxy. Si celui-ci n'est pas déjà dans le fichier de log, alors on l'ajoute en mettant le compteur de visite à 1. Si le client se trouve déjà dans le fichier, le compteur est incrémenté. Ceci permet d'avoir une vue globale de ses visiteurs et du nombre de leurs visites, et pourrait aussi permettre d'identifier les clients abusifs.

Le second fichier (log_requests) enregistre chaque requête effectuée auprès du proxy avec l'IP et le port d'origine de la demande. Afin de trouver l'IP d'un client à partir de son descripteur, la fonction *getpeername* a été utilisée. Ce fichier permet d'avoir une liste exhaustive des requêtes faites par le proxy et ainsi d'isoler les requêtes illégales et/ou les clients effectuant trop de requêtes.

3.4 Ce qui pourrait être encore fait : le time-out

On peut remarquer lors de l'utilisation du proxy que les déconnexions et reconnexions sont très fréquentes. Ceci est sans doute due au fait que les connexions ne soient pas conservées. On pourrait imaginer vérifier dans la réponse la durée de vie de la requête (keep-alive) afin de faire perdurer les connexions qui le doivent.

3.5 Version HTTPS

Nous tenons à remercier à Nicolas Bédrine pour cette partie. En effet, les forums sur lesquels nous sommes tombés proposaient d'utiliser la bibliothèque C openSSL. Cependant nous n'avons pas réussi à implémenter le code de manière fonctionnelle avec cette méthode. Avec le recul, cette bibliothèque est sans doute plus utile pour un serveur ou un client HTTPS que pour un proxy. C'est donc Nicolas qui nous a expliqué le fonctionnement de HTTPS. Nous suivons donc le schéma suivant : lorsque nous recevons un paquet CONNECT, nous savons qu'un client veut initier une connexion HTTPS. Pour ce faire, nous lui créons une websocket vers le serveur sur le port 443 et nous renvoyons un 200 OK au client pour signifier à ce dernier qu'il peut envoyer ses données. Ensuite, lors de la phase d'échange, lorsque nous recevons une donnée du serveur, nous la transmettons immédiatement au client. Et lorsque l'on reçoit un paquet de type inconnu de la part d'un client, on renvoie directement et sans traitement le paquet au serveur. En effet, étant donnée que les données sont cryptées, nous ne pouvons récupérer aucune information sur l'échange.

Références et autres sources d'inspiration

`http://blog.hikoweb.net/index.php?post/2011/11/06/Exemple-de-rapport-en-LaTeX`
Source pour le template LaTeX ayant servi à la rédaction de ce rapport.

`http://livre.g6.asso.fr/`
Site expliquant en détail le fonctionnement de l'IPv6 et l'utilisation de getaddrinfo

`http://stackoverflow.com/questions/20472072/c-socket-get-ip-adress-from-filedescriptor-retur`
Site ayant permis de comprendre le fonctionnement de getpeername

`http://ipv6test.google.com/`
Site ayant permis de tester la capacité du serveur à gérer l'IPv6

`http://man7.org/linux/man-pages/man7/ipv6.7.html`
Site ayant fréquemment servi pour retrouver les structures utilisées.

`http://simplestcodings.blogspot.fr/2010/08/secure-server-client-using-openssl-in-c.html`
`http://stackoverflow.com/questions/16255323/make-an-https-request-using-sockets-on-linux`
Sites expliquant le traitement du HTTPS en C

`https://www.openssl.org/docs/manmaster/ssl`
Documentation d'OpenSSL

`http://www.palaiszelda.com`

`http://perdu.com/`

`http://lost.com/`

`http://www.jeuxvideo.com/`

Liste de sites ayant subis mes assauts inconsidérés durant la phase de test :)