

Do Not truSt

Thomas Vantroys and Xavier Redon

1 Description du projet

L'objectif du projet consiste à développer un serveur mandataire pour les requêtes DNS (autrement dit un *proxy DNS*). Ce type de service logiciel est utilisé en cybersécurité aussi bien d'un point de vue défensif que offensif. En mode défensif, il sera utilisé pour enregistrer toutes les demandes de résolution de noms réalisées par une cible et ainsi étudier son comportement. Cela permet également de filtrer certaines requêtes. De manière offensive, un *proxy DNS* permet de leurrer une machine pour la rediriger vers une autre destination.

2 Organisation du projet

Afin de garantir un travail régulier de votre part, nous allons noter chacune des étapes du projet. À la fin d'une étape, nous fournirons si besoin à ceux qui le demande, une proposition d'implémentation afin que tous puissent continuer à avancer dans le projet. La construction du projet sera ainsi réalisée de manière incrémentale avec des refactorisations entre chaque étape.

Chaque binôme créera un dépôt `git` sur le serveur <https://archives.plil.fr>. Le dépôt sera nommé `PSR_Nom-binome1_Nom-binome2`. Vous configurerez le dépôt afin que les deux enseignants possèdent les droits en écriture et lecture. Ce dépôt doit être créé dès maintenant. Le dépôt `git` ne contiendra pas de fichiers compilés.

3 Étape 0 : proxy fonctionnel

La première étape du projet consiste à réaliser un proxy minimaliste qui va rediriger tous les paquets de type `query` qu'il reçoit vers un véritable serveur DNS et qui redirige les réponses vers les demandeurs initiaux. Dans le répertoire `etape0`, vous devez créer un fichier `Makefile` et un fichier `proxy.c` qui contiendra votre code. Pour cette première étape, nous tolérerons que tout le code soit dans la fonction `main`. Il n'est pas obligatoire de réaliser une version multithreadée. Afin de simplifier le développement, nous vous fournissons le fichier `dns.h` qui contient quelques constantes relatives au fonctionnement du DNS. Le fichier `dns_server.h` contient quelques adresses de serveur DNS utilisables pour la redirection. Pour les constantes supplémentaires, définition de types, de structures, *etc* dont vous pourriez avoir besoin, vous les placerez dans un fichier intitulé `dns_etape0.h`.

Cette étape sera évaluée directement après la première séance (22/02/2022).

4 Étape 1 : refactorisation

Maintenant que vous avez réalisé une première version fonctionnelle d'un proxy DNS, vous allez effectuer une première refactorisation de votre code. En effet, pour l'instant votre projet n'est constitué que d'un seul et unique fichier C. Afin d'augmenter la réutilisabilité de votre code, vous allez le réorganiser et créer différentes bibliothèques.

À la racine du dépôt `git`, vous allez créer un répertoire `etape1`. À l'intérieur de ce répertoire, vous allez pouvoir créer des répertoires selon le schéma suivant :

```
etape1
├── bin
├── libs
│   ├── static
│   ├── Log
│   ├── Reseau
│   └── Threads
└── src
```

Le répertoire `libs` contient différents sous-répertoires correspondants à différentes bibliothèques qui seront développées lors du tutorat. Le répertoire `libs/static` contiendra les versions compilées des différentes bibliothèques. Les codes sources de chaque bibliothèque seront dans un répertoire spécifique. Chaque répertoire contenant du code source, contiendra un fichier `Makefile` permettant la compilation et la création de la bibliothèque. Le répertoire `src`

contiendra le code source de la fonction principale (`main`) et un fichier `Makefile` pour la compilation. Le répertoire `etape1/bin` contiendra l'exécutable de votre proxy DNS.

Le travail suivant consiste à refactoriser votre code, c'est à dire créer une véritable bibliothèque `Reseau` permettant d'encapsuler les différentes fonctions du réseau. Cela doit permettre une meilleure réutilisation de votre code. Votre bibliothèque sera nommée `libreseau.a` et contiendra a minima les fonctions suivantes :

- `int initialisationSocketUDP(char *service)` pour la création d'une socket d'écoute sur le port spécifié en paramètre. La fonction renvoie le numéro de la socket créée.
- `int messageUDP(...)` pour l'envoi d'un message et la réception d'un message UDP. À vous de définir les paramètres qui vous semblent pertinents.
- `int boucleServeurUDP(int s, pointeur de fonction)`. Cette fonction écoute sur une socket UDP passée en premier paramètre. À chaque réception d'un message, elle appelle la fonction passée en paramètre. À vous de déterminer ce paramètre.

Une fois la bibliothèque créée, vous devez modifier refactoriser le programme principal en le plaçant dans le fichier `main.c` qui sera situé dans le répertoire `src`.

Une fois que votre proxy DNS refactorisé est opérationnel, vous modifierez le programme principal afin de pouvoir lui passer des arguments. La gestion des arguments sera codée dans le fichier `args.c` situé dans le répertoire `src`. Le traitement des arguments sera effectué en utilisant la fonction `getopt_long`. A minima, vous traiterez les options suivantes :

- `-h` et `--help` : affiche le message d'aide du programme qui explique toutes les options possibles
- `-p PORT` et `--port=PORT` : permet de modifier le port par défaut (53) utilisé par le programme pour recevoir les questions DNS
- `-s SERVEUR` et `--serveur=SERVEUR` : permet de spécifier un serveur DNS particulier à la place du serveur DNS par défaut

5 Étape 2 : génération de log

Dans l'étape 2 du tutorat, vous allez ajouter des fonctionnalités de *logging* à votre proxy DNS. En effet, un des intérêts d'utiliser un proxy DNS consiste à sauvegarder les différentes requêtes DNS utilisées par une machine. Cela permet par exemple d'étudier le comportement d'un logiciel malveillant. Différentes stratégies peuvent être mises en place pour générer et sauvegarder les traces de fonctionnement du proxy DNS. Afin d'avoir une solution générique permettant d'évoluer facilement au cours du temps, vous allez devoir écrire des bibliothèques dynamiques et vous inspirer du patron de conception "stratégie".

En génie logiciel, Les patrons de conceptions représentent des solutions à des problèmes fréquemment rencontrés lors des développements logiciels. Vous trouverez sur wikipedia (https://fr.wikipedia.org/wiki/Patron_de_conception) une description plus détaillée. Dans notre cas, nous souhaitons pouvoir changer dynamiquement le comportement du système de gestion des logs du proxy DNS sans devoir le recompiler. Pour cela, nous allons mettre en œuvre le patron de conception "stratégie". L'interface du service de *logging* doit donc être identique pour chaque implémentation. Elle est définie dans le fichier `log_dns.h`. Chaque service sera implémenté sous la forme d'une bibliothèque dynamique (extension `.so`). Les deux fonctions de chaque bibliothèque sont :

- `int initLog(void *)` : cette fonction permet d'initialiser le service de *logging*
- `int addLog(logMsg_t *)` : cette fonction envoie les données à traiter par le service de *logging*. Cette fonction sera donc appelée pour chaque nouveau message de *log*.

Vous devez développer *a minima* deux bibliothèques. La première sauvegardera dans un fichier tous les messages DNS reçus par le proxy. Chaque message correspondra à une ligne du fichier et sera écrit octet par octet (`printf("%x", ...)`). Le nom du fichier sera passé en paramètre de la fonction `initLog`. La seconde bibliothèques affichera à l'écran chaque nom de domaine se trouvant dans un paquet de type `Query`.

Afin de pouvoir charger dynamiquement la bibliothèque choisie, vous devez développer un gestionnaire de chargement. Ce dernier sera sous la forme d'une bibliothèque dynamique et contiendra trois fonctions différentes. Les prototypes de fonctions sont définis dans le fichier `genericLog.h` :

- `int loadStrategy(char *)` : cette fonction charge en mémoire la bibliothèque dont le nom est passé en paramètre. Elle renvoie -1 en cas d'erreur ou 0 en l'absence de problème ;
- `int initStrategy(void *)` : cette fonction permet d'initialiser le service de *logging*. Elle appelle donc la fonction `initLog` avec les éventuels paramètres ;

- `int logStrategy(logMsg_t *)` : cette fonction permet d'envoyer les données à traiter. Elle appelle la fonction `addLog`.

Le gestionnaire est donc un *wrapper* de la bibliothèque choisie. Pour comprendre le fonctionnement du chargement dynamique de bibliothèque, vous pouvez vous baser sur les explications se trouvant à l'adresse <https://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html>.

Pour choisir la stratégie de log utilisée, vous devez modifier le programme principal pour y ajouter les options :

- `-l STRATEGIE` et `--logstrategy=STRATEGIE` : permet de choisir la stratégie à charger
- `-i INIT_ARGS_STRATEGIE` et `--initlogstrategy=INIT_ARGS_STRATEGIE` : permet de passer des paramètres d'initialisation à la stratégie choisie

Si aucune stratégie n'est passée en argument, aucune sauvegarde des messages DNS n'est réalisée.

6 Étape 3 : multithreading

Pour l'instant, votre proxy DNS traite séquentiellement les requêtes qu'il reçoit. Pour améliorer ce comportement, vous allez maintenant utiliser des threads pour gérer les requêtes ainsi que pour gérer les messages de log. Chaque fois qu'une requête parvient, vous devez créer un thread qui gèrera le traitement complet de la requête (envoi vers un serveur DNS et renvoi de la réponse vers le client). Afin d'améliorer le traitement des messages de *logging*, un thread spécifique sera chargé du traitement des messages. La communication entre les threads de traitement des requêtes et le thread de traitement du *logging* sera réalisée au moyen d'une mémoire partagée.

Pour simplifier la correction, copiez votre répertoire `etape2` en le nommant `etape3`

6.1 Gestion des threads

Pour la gestion des threads, vous allez créer une bibliothèque dynamique nommée `libthread.so`. Cette bibliothèque, qui se situera dans le répertoire `libs/Threads`, contiendra a minima la fonction suivante :

- `int launchThread(void *(*fonction)(void *), void *arg, size_t argSize)` : cette fonction générique permet de créer un nouveau thread et gère les éventuelles allocations mémoires nécessaires.

Si d'autres fonctions génériques liées au threads sont nécessaires, c'est à vous des les ajouter dans cette bibliothèques.

6.2 Gestion de mémoire partagée

Pour gérer la communication entre les threads qui gère les requêtes et le thread qui gère le *logging*, nous allons passer par une mémoire partagée et mettre en œuvre le principe classique de producteur-consommateur. Comme il est possible de gérer la mémoire partagée de différentes manières, nous allons utiliser une interface générique possédant les fonctions suivantes :

- `int allocateMemory(size_t size)` : cette fonction permet de créer un espace mémoire de taille `size` octets. Elle renvoie 0 quand tout se passe bien, -1 en cas d'erreur ;
- `int writeMemory(void *data, uint8_t size)` : cette fonction ajoute dans l'espace mémoire les `size` octets se trouvant à partir de l'adresse `data`. Cette fonction doit vérifier qu'il reste suffisamment de place avant de réaliser l'écriture. Si ce n'est pas le cas, elle renvoie immédiatement une valeur retour -1. Si tout se passe bien, elle renvoie 0. Nous considérons ici que la zone mémoire est contigüe. Cette fonction écrira donc en mémoire un octet correspondant à la taille des données, puis les données ;
- `void *readMemory(uint8_t *size)` : cette fonction permet de lire le message courant se trouvant dans la mémoire. Cette fonction gère les allocations mémoires afin de renvoyer les octets du message ainsi que le nombre d'octets du message.

D'autres fonctions peuvent être utiles à implémenter dans cette bibliothèque, mais ne sont pas obligatoire, mais votre code rester clair et structurés. Voici quelques exemples :

- `bool memoryIsEmpty(void)`
- `bool memoryIsFull(void)`
- `uint8_t availableMemory(void)`

Le code source de cette bibliothèque se trouvera dans le répertoire `libs/Memoire`. La bibliothèque se nommera `libmemoire.so`. Pour cette étape, cette bibliothèque sera implémentée sous la forme d'un buffer circulaire qui sera alloué sur le tas. D'autres variantes seront possibles dans la suite du projet.

6.3 Modifications de l'existant

Une fois que vos bibliothèques sont réalisées, vous devez modifier le code existant afin de créer les différents threads.

7 Étape 4 : refactorisation

Après tous les développements réalisés, il est maintenant temps de réaliser la refactorisation de votre projet et de tester le bon fonctionnement du proxy. Une partie du travail consiste à garantir la généricité des bibliothèques. Il faudra par exemple supprimer toute référence directe à des fonctions bas-niveau dans le code du programme principal. En effet, jusqu'à présent, il reste un appel à `sendto` dans le fichier `main.c` et le prototypage de `boucleServeurUDP` expose indument la structure `struct sockaddr_storage`. De même, la gestion des sémaphores sera encapsulée dans la bibliothèque des threads. Le fichier principal, `main.c`, ne devra inclure aucun des fichiers d'entête (`.h`) relatifs aux sockets et aux pthreads.

Un des intérêts de développer des bibliothèques séparées est de pouvoir les tester indépendamment les unes des autres. Il est donc possible de réaliser des tests unitaires (voir https://fr.wikipedia.org/wiki/Test_unitaire). Afin de vérifier le bon fonctionnement des bibliothèques, vous ajouterez dans chaque répertoire de bibliothèque, un répertoire `tests` qui comprendra votre programme de tests unitaires. Par exemple, le répertoire `etape3/libs/Memoire/tests` contiendra le fichier `testMemory.c` qui testera différents cas de figure d'utilisation comme par exemple écrire une quantité de données supérieure à la taille de la mémoire ou lire une mémoire vide. Voici un début de programme de test à étendre :

```
#include <stdio.h>
#include <assert.h>

#include "libmemory.h"

void test_allocateMemory(size_t size)
{
    printf("Test allocateMemory, size=%d :", (int)size);
    assert(allocateMemory(size) == 0);
    printf("Success\n");
}

int main(void)
{
    test_allocateMemory(256);
    return 0;
}
```

Vous pouvez, si vous le souhaitez/préférez, utiliser la bibliothèque de tests unitaires CUNIT (<http://cunit.sourceforge.net/>).

Une fois toutes les bibliothèques refactorisées et testées individuellement, vous pourrez tester votre proxy DNS à l'aide du logiciel `valgrind` afin de vérifier que vous gérez correctement la mémoire.

8 Étape 5 : redirection

Un usage possible d'un proxy DNS est de pouvoir détourner les clients qui effectuent des requêtes en leurs envoyant de fausses réponses pour certains noms de domaine. Pour cela, vous allez utiliser un fichier de configuration dans lequel chaque ligne représentera une redirection. Le format sera le suivant :

```
nom.de.domaine=adresse_IPv4,adresse_IPv6,adresse_server_mail
```

Voici un exemple de fichier possible :

```
tvantroys.plil.fr=172.26.145.55,2001:660:4401:6050:5000::5,172.26.145.55
www.google.com=172.26.145.56,,
*.stackoverflow.com=172.26.145.55,2001:660:4401:6050:5000::5,
```

Dans ce fichier, le proxy redirige toutes les requêtes IPv4 du domaine `tvantroys.plil.fr` vers l'adresse `172.26.145.55`, les requêtes IPv6 vers `2001:660:4401:6050:5000::5` et les requêtes MX (mail) vers `172.26.145.55`. Pour le domaine `www.google.com`, seule l'adresse IPv4 est modifiée. Pour toutes les requêtes qui se terminent par `stackoverflow.com`,

il y a redirection vers 172.26.145.55 ou 2001:660:4401:6050:5000::5 en fonction du type de demande. Afin de prendre en compte ce fichier, il faut ajouter comme option à votre programme `-c CONFIG_FILE` et `--configfile=CONFIG_FILE`, toujours en utilisant `getopt_long`. Afin de pouvoir ajouter ou retirer dynamiquement des redirections, toutes les redirections seront placées dans une zone mémoire partagée. Vous devez écrire un programme, nommé `dnsproxy_mgr`, permettant d'aller dynamiquement modifier, ajouter, retirer des redirections sans devoir relancer le programme principal. C'est à vous d'organiser correctement le découpage de votre code, de réaliser les tests unitaires, *etc.*

9 Checklist du projet

- Séance 1 (4h, 22 février 2022)
 - ☐ Création et configuration du dépôt `git`
 - ☐ Proxy minimal terminé
- Séance 2 (2h, 01 mars 2022)
 - ☐ Création des différents répertoires correspondants aux bibliothèques
 - ☐ Création d'un `Makefile` principal et d'un `Makefile` par répertoire
 - ☐ Création d'une bibliothèque `Reseau`
 - ☐ Modification du programme principal afin d'utiliser la bibliothèque `Reseau`
 - ☐ Ajout de fonctions pour le traitement des options du programme
- Séance 3 (2h, 08 mars 2022) et séance 4 (2h, 14 mars 2022)
 - ☐ Création de la bibliothèque de gestion `libgenericLog.so`
 - ☐ Création d'une bibliothèque simple de *logging*
 - ☐ Création d'une bibliothèque de *logging* des noms de domaines trouvés dans des requêtes `query`
 - ☐ Modification du programme principal afin d'utiliser les bibliothèques pour la génération de log
 - ☐ Ajout de l'option `-l STRATEGIE` et `--logstrategy=STRATEGIE`
 - ☐ Ajout de l'option `-i INIT_ARGS_STRATEGIE` et `--initlogstrategy=INIT_ARGS_STRATEGIE`
 - ☐ Utilisation du gestionnaire de bibliothèque de *log*
 - ☐ Ajout des *logs* des messages reçus par le proxy DNS
- Séance 5 (2h, 21 mars 2022) et Séance 6 (2h, 22 mars 2022)
 - ☐ Création de la bibliothèque de gestion mémoire `libmemoire.so`
 - ☐ Création de la bibliothèque de gestion des threads `libthread.so`
 - ☐ Ajout de `Makefile` et modification de `Makefile` existants
 - ☐ Modification des fonctions existantes afin de créer les threads pour la gestion des requêtes
 - ☐ Développement du thread de gestion des *logs*
- Séance 7 (2h, 28 mars 2022) et Séance 8 (2h, 29 mars 2022)
 - ☐ Généricité/encapsulation de la bibliothèque `Reseau`
 - ☐ Généricité/encapsulation de la bibliothèque `Threads`
 - ☐ Généricité/encapsulation de la bibliothèque `Mémoire`
 - ☐ Tests unitaires de la bibliothèque `Reseau`
 - ☐ Tests unitaires de la bibliothèque `Threads`
 - ☐ Tests unitaires de la bibliothèque `Mémoire`
 - ☐ Tests unitaires de la bibliothèque `Log`
 - ☐ Vérification et suppression des fuites mémoires (utilisation de Valgrind)
- Séance 9 (2h, 25 avril 2022) et Séance 10 (4h, 26 avril 2022)
 - ☐ Ajout de l'option `-c CONFIG_FILE` et `--configfile=CONFIG_FILE`
 - ☐ Développement d'une solution générique et encapsulée
 - ☐ Tests unitaires du code développé