Rapport de projet

Implémentation d'un IDS

Grégoire Roumache
Florian Fichet

Développement – Sécurité des systèmes
Deuxième année, groupe C-2
Année académique 2020-2021

3 Janvier 2021

1 Introduction

Pour le cours de Développement, nous avons dû implémenter un IDS (Intrusion Detection System) qui analyse le trafic qui passe par une interface réseau de la machine. Il permet de détecter des activités suspectes en fonction d'un certain nombre de règles données au préalable. En cas d'anomalie, le programme peut alerter l'utilisateur en la signalant dans les logs du système.

Nous nous sommes organisés en utilisant la plateforme github, comme recommandé, et voici le lien vers notre répertoire :

`https://github.com/groumache/Intrusion-Detection-System`

2 Organisation du travail et configuration des outils

2.1 Les issues sur github

Les issues sont utilisées pour s'organiser et garder une trace des tâches, des améliorations et des bugs dans un projet sur github.

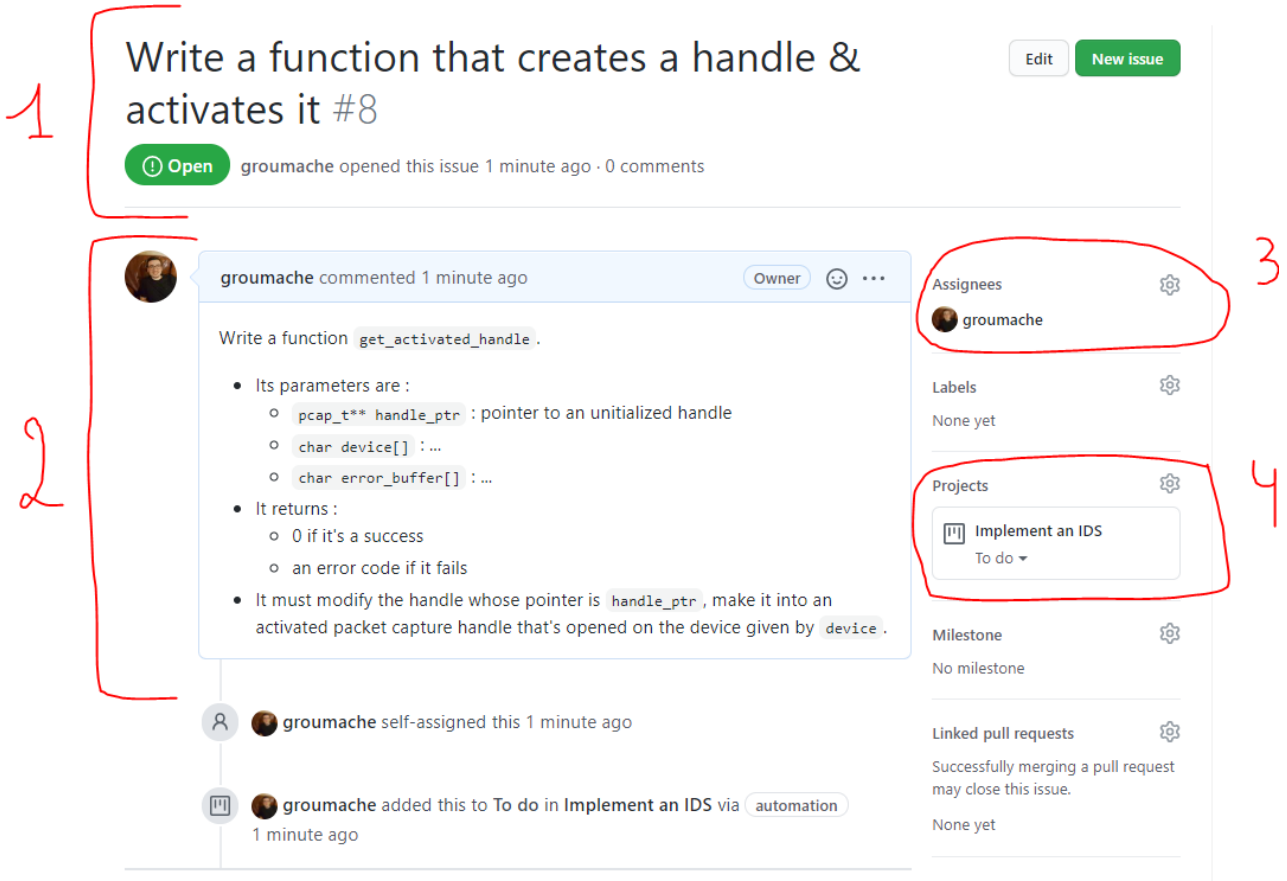

Figure 1 – Une issue sur github

Explication de la figure 1 :

1. Titre de l'issue.
2. Explication/description de l'issue.
3. On peut attribuer une issue à des personnes qui seront responsables pour résoudre l'issue.
4. Le projet dans lequel l'issue va apparaître.

2.2 Utilisation du kanban sur github

Comme nous l'avons précisé dans l'introduction, nous nous sommes organisés en utilisant github. Nous n'avons pas seulement mis le code sur github mais aussi utilisé le tableau kanban avec 4 sections :

1. **Ideas** : un endroit pour ajouter des notes/issues qui dépassent les exigences du projet.
2. **To do** : un endroit où les issues sont placées quand elles sont créées (correspond aux critères minimum de réussite).
3. **In progress** : l'endroit où les issues vont quand on travaille dessus.
4. **Done** : là où les issues vont quand elles sont fermées ou la pull request liée a été fusionnée.

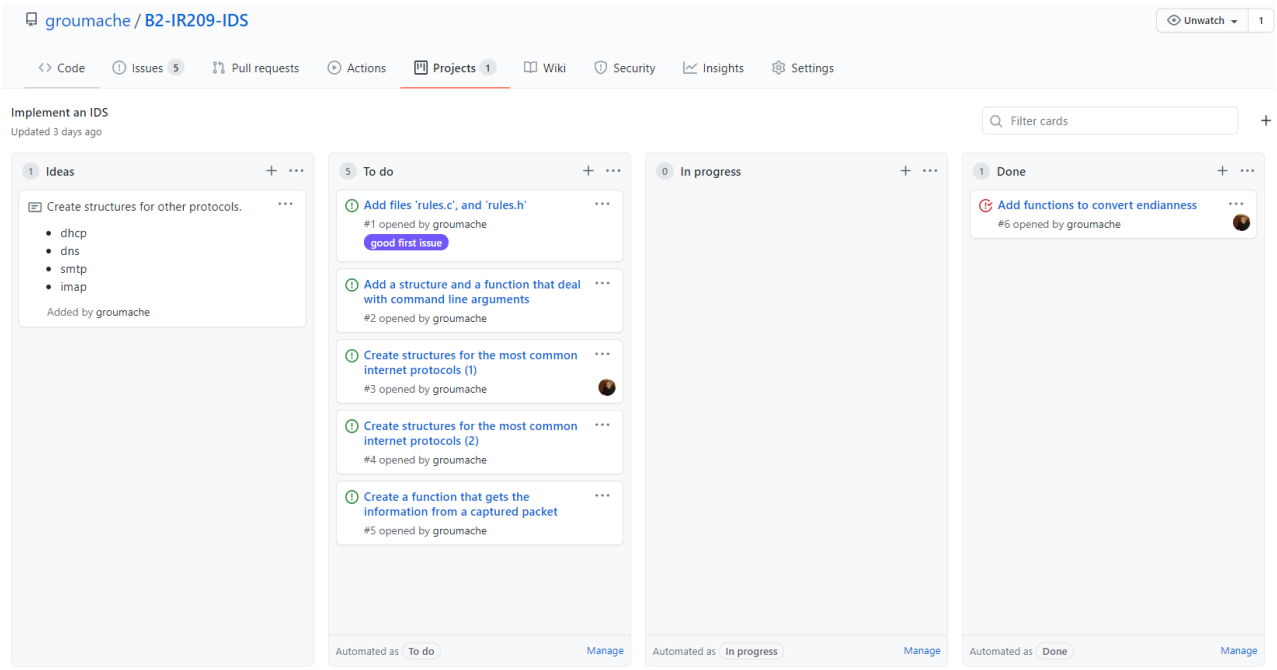

FIGURE 2 – Kanban sur github

Sur le schéma de la figure 3, on voit le network graph. Cette image illustre bien comment nous avons travaillé, c-à-d en créant des nouvelles branches à chaque fois que nous avons eu besoin de résoudre une issue. Une fois que le code a été ajouté, nous avons fait des pull requests pour fusionner les branches.

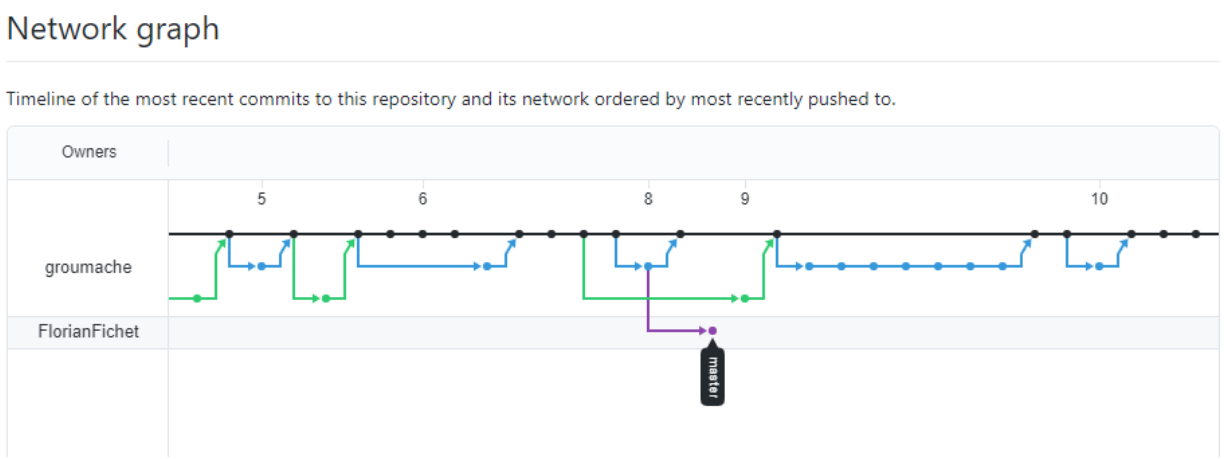

FIGURE 3 – Network graph montrant les commits et comment les branches se rejoignent

2.3 Configuration du debugger

Les fichiers de configurations sont présentés sur les figures 4 et 5.

1. Sur la figure 4, on voit le fichier *tasks.json* où on configure la compilation du projet :
   — on voit que le paramètre *command* donne bien le compilateur gcc,
   — les arguments sont précisés par le paramètre *args*, ce sont ceux recommandés dans l'énoncé,
   — si on combine le tout, c'est l'équivalent de la commande :
   ```
   gcc -Wall -o ids main.c populate.c rules.c -lpcap
   ```
2. Sur la figure 5, on voit le fichier *launch.json*, il sert à configurer le lancement du programme :
   — dans *program*, on voit qu'on a bien précisé *ids*,
   — et dans *args*, on voit les arguments du programme.
   — si on combine le tout, c'est l'équivalent de la commande :
   ```
   ids -p -d eth1 -r ids.rules -n 10
   ```

Remarquez qu'il faut lancer le programme avec des privilèges plus élevé que l'utilisateur standard. Pour réussir à faire cela, on peut lancer visual studio code avec ces mêmes privilèges.

launch.json | tasks.json

developpement-Q1 > projet > .vscode > tasks.json > ...

You, 4 days ago | 1 author (You)

```
{       You, 4 days ago • add code to close the handle and free
    "tasks": [
        {
            "type": "cppbuild",
            "label": "Build project",
            "command": "/usr/bin/gcc-10",
            "args": [
                "-Wall",
                "-g",
                "-o",
                "/home/user/Downloads/project/ids",
                "/home/user/Downloads/project/main.c",
                "/home/user/Downloads/project/populate.c",
                "/home/user/Downloads/project/rules.c",
                "-lpcap",
            ],
            "options": {
                "cwd": "${workspaceFolder}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "detail": "Task generated by Debugger."
        },
```

FIGURE 4 – Fichier de configuration du debugger, pour la compilation du programme

launch.json | tasks.json

developpement-Q1 > projet > .vscode > launch.json > Launch Targets > { } gcc-10 - Build

You, a day ago | 1 author (You)

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug project",
            "type": "cppdbg",
            "request": "launch",
            "program": "/home/user/Downloads/project/ids",
            "args": [
                "-p",
                "-d", "eth1",
                "-r", "/home/user/Downloads/project/ids.rules",
                "-n", "10",
            ],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
```

FIGURE 5 – Fichiers de configuration du debugger, pour lancer le programme

2.4 Utilisation de valgrind

Valgrind est un outil de programmation qui sert principalement à découvrir des fuites de mémoire. Par exemple, si nous allouons un bloc de mémoire dans le tas (heap) et que nous ne le libèrons pas, valgrind va nous rappeler à l'ordre et même nous dire à quelle ligne le bloc mémoire à été alloué (si le programme a été compilé avec un debug flag).

Lancement du programme avec valgrind (rappel : ids doit être lancé avec des privilèges) :

```
valgrind --leak-check=full \
    --show-leak-kinds=all \
    --track-origins=yes \
    --verbose \
    ./ids
```

L'objectif est donc d'obtenir un résultat comme celui à la figure 6.

```
==2110==
==2110== HEAP SUMMARY:
==2110==     in use at exit: 0 bytes in 0 blocks
==2110==   total heap usage: 165 allocs, 165 frees, 283,249 bytes allocated
==2110==
==2110== All heap blocks were freed -- no leaks are possible
==2110==
==2110== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

FIGURE 6 – Résultat de valgrind

2.5 Configuration du formatage automatique

L'extension C/C++ de visual studio code nous permet de faire du formatage automatique de notre code comme illustré par la figure 7. La configuration se fait au en suivant la syntaxe clang-format, voici celle que nous avons utilisé :

```
{
    BasedOnStyle: Google,
    IndentWidth: 4,
    MaxEmptyLinesToKeep: 2,
    IndentPPDirectives: BeforeHash
}
```

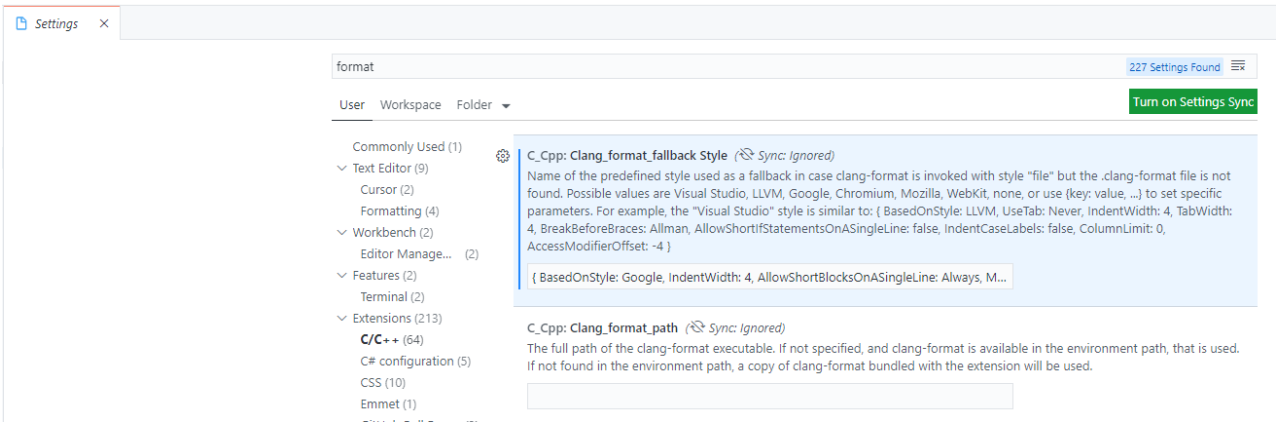

FIGURE 7 – Paramètre de formatage dans visual studio code

3 Architecture du projet

3.1 Architecture du système

Comme on peut voir sur la figure 8, l'ids est divisé en quatre parties logiques.

1. *read_rules* : cette partie est responsable d'extraire les règles du fichier pour les mettre dans les structures appropriées
2. *populate_packet* : cette partie doit extraire le paquet pour le placer dans les structures appropriées
3. *main* : cette partie a 4 responsabilités :
   — analyse les arguments du programme
   — faire appel aux parties *read_rules*, et *populate_packet*
   — vérifier si les packets correspondent aux règles
   — si ils correspondent aux règles, elle doit écrire un message dans syslog

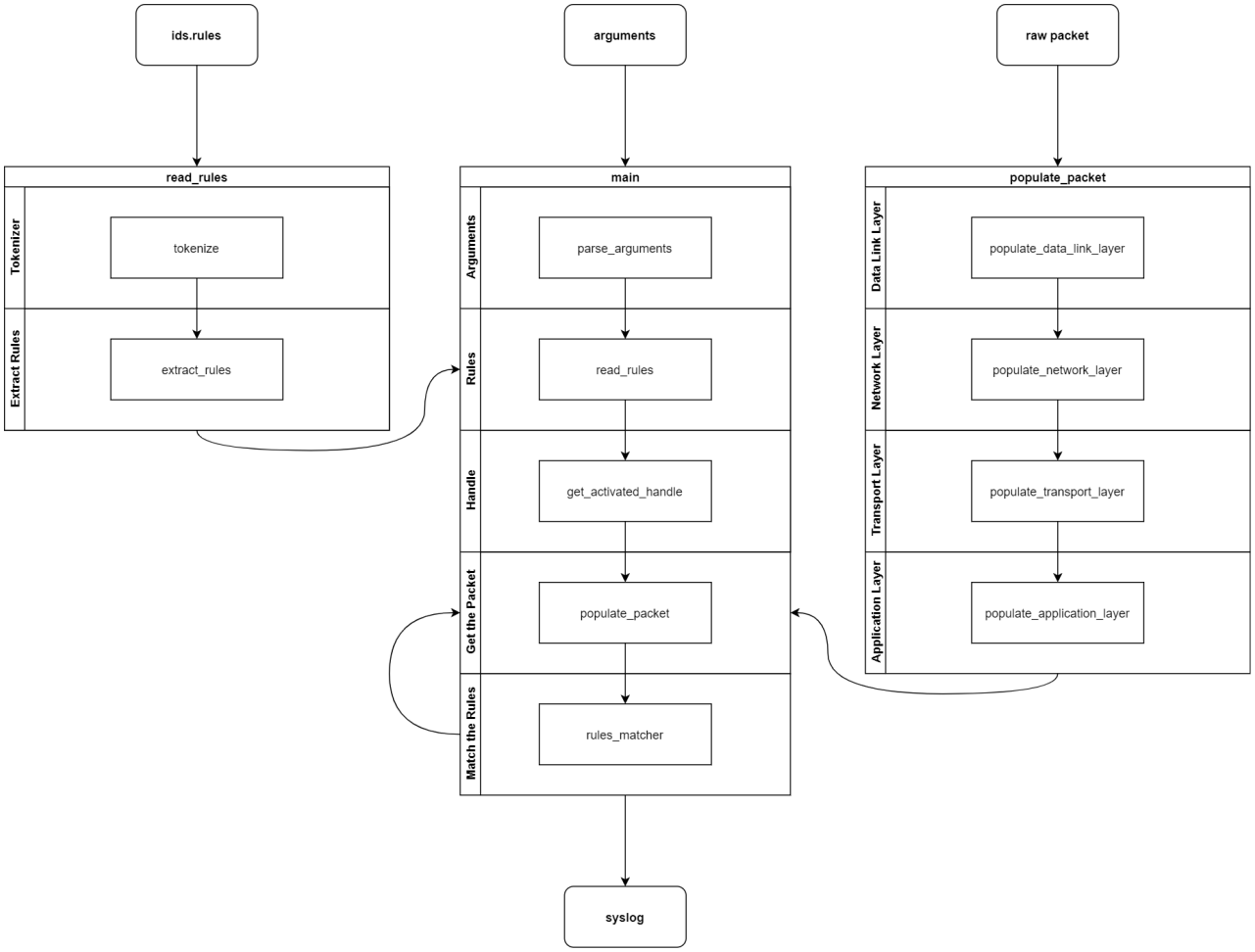

FIGURE 8 – Architecture du système

3.2 Structure du projet

Pour mieux comprendre l'organisation du projet, on peut regarder la figure 9 sert à montrer quels fichiers sont inclus dans quels autres.

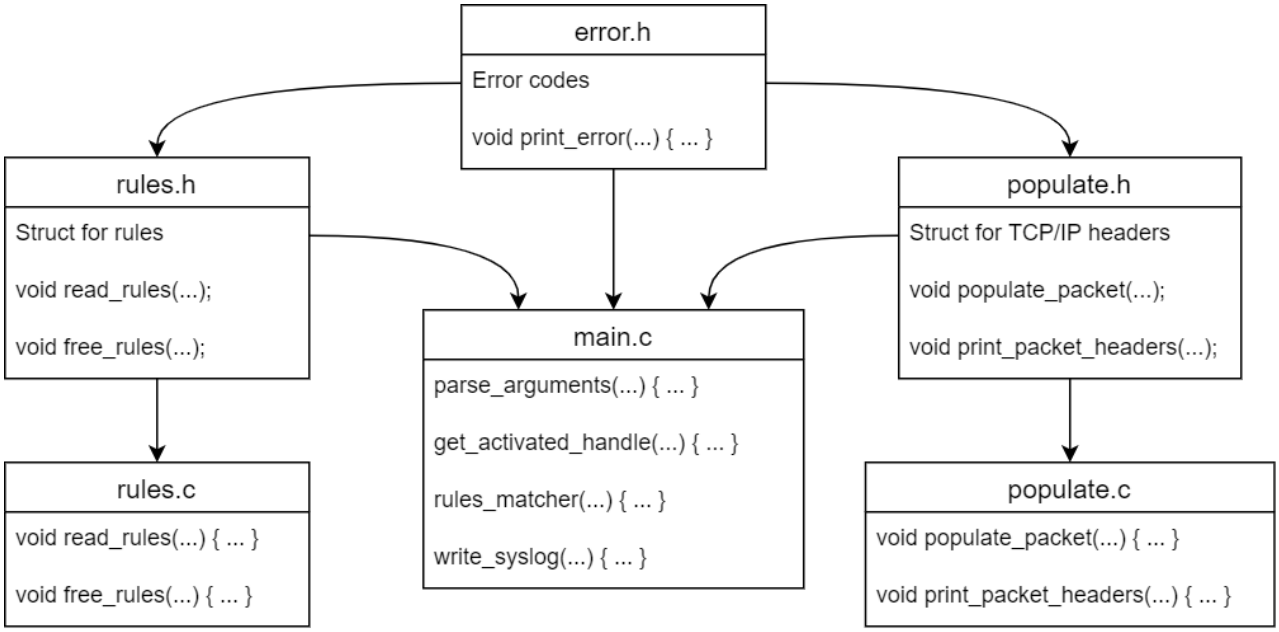

FIGURE 9 – Structure du projet

4 Organisation générale du code

Dans cette section, nous allons expliquer comment le code fonctionne de manière générale mais sans entrer dans le détail. Les choix d'implémentation spécifiques seront expliqués dans la section suivante (section 5).

4.1 Fichier *populate.c*

Les 3 fonctions de ce fichier qui ont leur définition dans le header *populate.h* sont :
- *populate_packet*, organise les informations à partir de la trame brute
- *print_packet_headers*, affiche les headers des protocols utilisés dans le paquet
- *print_packet_data*, affiche les données contenues dans le protocole de dernière couche (HTTP, TLS, TCP ou UDP)

La fonction la plus importante de ce fichier est évidemment la fonction *populate_packet* qui va appeler d'autres fonctions *populate* comme représenté sur la figure 10.

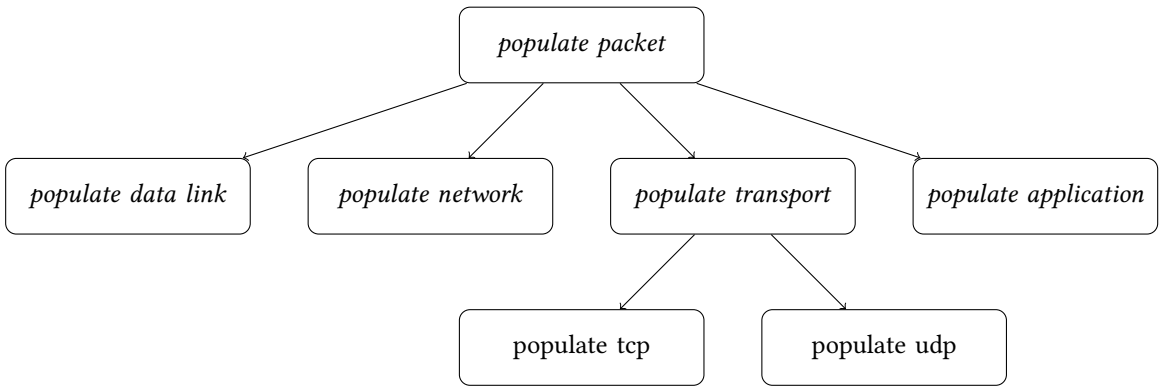

FIGURE 10 – Organisation des fonctions *populate*

4.2 Fichier *rules.c*

Les fonctions de ce fichier qui ont leur définition dans le header *rules.h* sont :
- *read_rules*

— *free_rules*

La fonction la plus importante de ce fichier est *read_rules* qui va faire la tokenisation du fichier de règles (voir section 5.1 pour des explications sur les tokens) puis en extraire les règles, ceci est illustré par la figure 11.

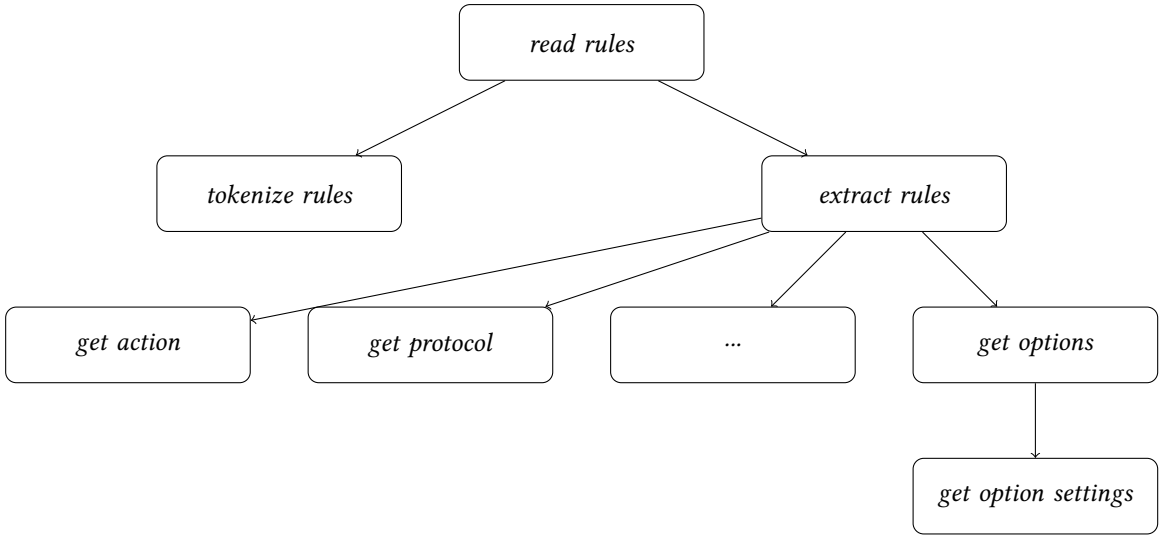

FIGURE 11 – Organisation des fonctions utilisées pour extraire les règles

Les fonctions pour obtenir les IPs et les ports sont plus compliquées parce que nous avons suivi le format de règle de *surricata* [2]. Ces formats sont représentés sur les figures 12 et 13.

Opérateur	Description
../..	range d'IPs (notation CIDR)
!	exception/négation
[.., ..]	groupement

FIGURE 12 – Format d'IP acceptable dans le fichier de règles

Opérateur	Description
:	range de ports
!	exception/négation
[.., ..]	groupement

FIGURE 13 – Format de port acceptable dans le fichier de règles

4.3 Fichier *main.c*

Il y a deux structures dans *main.c* :

— *IdsArguments*, sert à stocker les paramètres donnés en ligne de commande à l'IDS,

— *UserArgsPacketHandler*, sert à stocker des paramètres pour la fonction *packet_handler* tels que les règles du programme et les arguments de l'IDS lui-même.

Il y a aussi quelques fonctions à noter dans le fichier *main.c* :

— *main*, c'est le point d'entrée du programme, elle va appeler les autres fonctions de l'ids,

— *packet_handler*, fonction appellée par *pacp_loop* quand un paquet est capturé, elle doit utiliser les fonctions *populate_packet* et *rules_matcher* pour vérifier si le paquet correspond à une règle,

— *rules_matcher*, vérifie si le paquet correspond à une règle,

— *parse_arguments*, analyse les arguments donnés en ligne de commande,

— *write_syslog*, écrit un message dans syslog.

4.4 Détection de l'encryption du payload

Dans la section *critères minimum de réussite* de l'énoncé, il nous est demandé de pouvoir faire la *détection de l'encryption du payload*. Cependant, c'est une tâche difficile si on doit le faire pour tous les paquets. D'autant plus que des données compressées ressemblent souvent à des données cryptées [3].

Ce que nous avons décidé de faire est donc de laisser l'utilisateur écrire des règles pour des protocoles de chiffrement des données comme TLS qui seront détectés en fonction du port (source ou destination) utilisé. Notre implémentation diffère en deux points de la recommendation donnée (figure 14) :

1. Nous ne cherchons pas à détecter la poignée de main TLS sur le port 443 car c'est le seul protocole qui utilise ce port [5], si un paquet est transmis avec ce port, il est directement signalé en tant que paquet utilisant TLS.
2. Nous ne gardons pas trace de la communication en utilisant les numéro de séquence de paquet TCP mais nous utilisons uniquement le port utilisé. Ceci découle du point précédent, c-à-d qu'un paquet est signalé en fonction de son port, et puisque une communication utilisant TCP ou UDP ne peut pas changer de port car elle est identifiée en utilisant ces ports [4], il n'y a pas besoin de garder une trace de la communication avec une variable dans le code (le port utilisé est suffisant pour identifier la communication).

Il y a deux problèmes à cette approche :

— On imagine facilement que le protocole TLS peut être utilisé sur d'autres ports sans difficultés mais sa détection devient ardue puisqu'il faut dès lors chercher la présence de header TLS dans tous les paquets peut importe leur port. Ceci augmente aussi le risque de signalement de paquets n'utilisant pas le TLS. Nous avons décidé de ne pas prendre ce risque.
— On peut aussi penser que du traffic légitime passerait sur un port réservé au protocole TLS. Cependant, il y a peu de chance que cela arrive puisque le port est réservé.

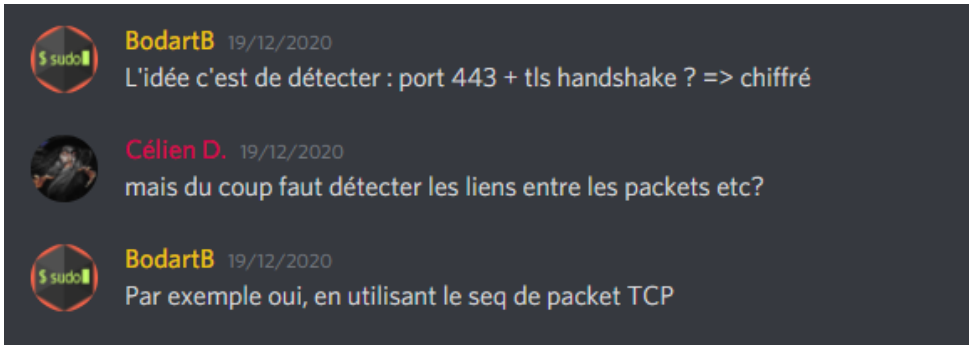

FIGURE 14 – Recommendation pour détecter et garder trace des paquets chiffrés

5 Choix d'implémentation et problèmes rencontrés

5.1 Tokens

La tokenisation c'est la séparation d'un texte en plus petites unités appelées tokens (comme illustré sur la figure 15). L'utilité de la tokenisation dans ce projet est que l'on a séparé deux choses :

— le traitement des informations contenues dans le texte,
— et la recherche des informations.

Ainsi, la fonction *extract_rules* (appellée par *read_rules* après la tokenisation) ne doit pas chercher où commence et finit un protocole dans le fichier de règle puisque ça a été pris en charge par la fonction *tokenize_rules*, elle est seulement responsable d'extraire l'information du token.

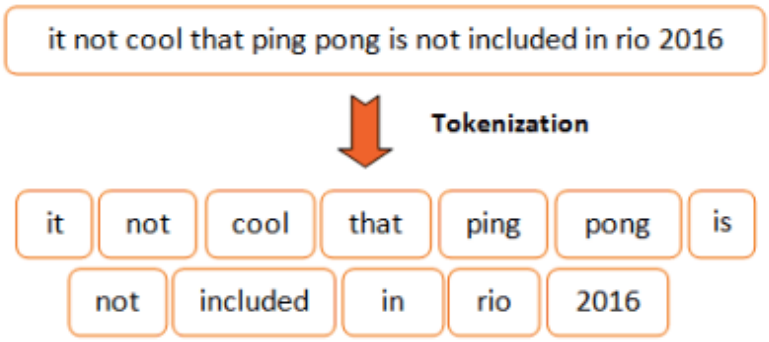

FIGURE 15 – Illustration de la tokenisation [6]

Nous avons mis ci-dessous la fonction *extract_rules* utilisée dans notre code pour extraire les règles des tokens et les mettre dans des structures *Rule*. La variable *i* est utilisée pour itérer sur les tokens. Cependant il y a un problème : si une IP, par exemple, est en fait une liste d'IP qui est placée sur plusieurs tokens (voir figure 12 sur les formats d'IP), de combien doit-on incrémenter la variable *i* ?

Pour résoudre ce problème, le pointeur de la variable *i* est passé en paramètre des fonctions qui servent à obtenir la règle (comme *get_rule_action* ou *get_rule_source_ip*). Ce sont désormais ces fonctions qui sont responsables d'incrémenter *i*. Ainsi, si une liste d'IP source prend vingt tokens, la fonction *get_rule_source_ip* devra incrémenter *i* de vingt.

```
1  void extract_rules(Rule **rules_ptr, int *nb_rules, Tokens *tokens) {
2      int i = 0;
3      while (i < tokens->nb_tokens) {
4          // add an empty rule to the list
5          ...
6
7          // initialize the rule
8          ...
9
10         // get the rule
11         get_rule_action(rule, tokens, &i);
12         get_rule_protocol(rule, tokens, &i);
13         get_rule_source_ip(rule, tokens, &i);
14         get_rule_source_port(rule, tokens, &i);
15         get_rule_direction(rule, tokens, &i);
16         get_rule_destination_ip(rule, tokens, &i);
17         get_rule_destination_port(rule, tokens, &i);
18         get_rule_options(rule, tokens, &i);
19
20         // NOTE: no need to increase i (i++) as it is incremented in the
21         // functions used above
22     }
23 }
```

5.2 Structures de données dynamiques

Dans ce projet, nous nous sommes souvent retouvés face à des quantités inconnues. Par exemple, combien de règles allons nous avoir dans le fichier de règles ? Pour résoudre ce problème, nous avons ajouter des fonctions, telle que *increase_nb_rules* ci-dessous. Elles sont chargées de ré-allouer la mémoire afin que la "liste" d'éléments puisse grandir dynamiquement en fonction des besoins du programme.

```
1  void increase_nb_rules(Rule **rules_ptr, int *nb_rules) {
2      (*nb_rules)++;
3      (*rules_ptr) = realloc((*rules_ptr), (*nb_rules) * sizeof(Rule));
```

```
}
```

5.3 Problèmes d'endianness

Il y a deux manières de représenter un nombre qui occupe plusieurs octets (voir figure 16) :
- placer les octets de poids fort dans l'ordre des adresses (big endian),
- ou bien les placer dans l'ordre inverse des adresses (little endian).

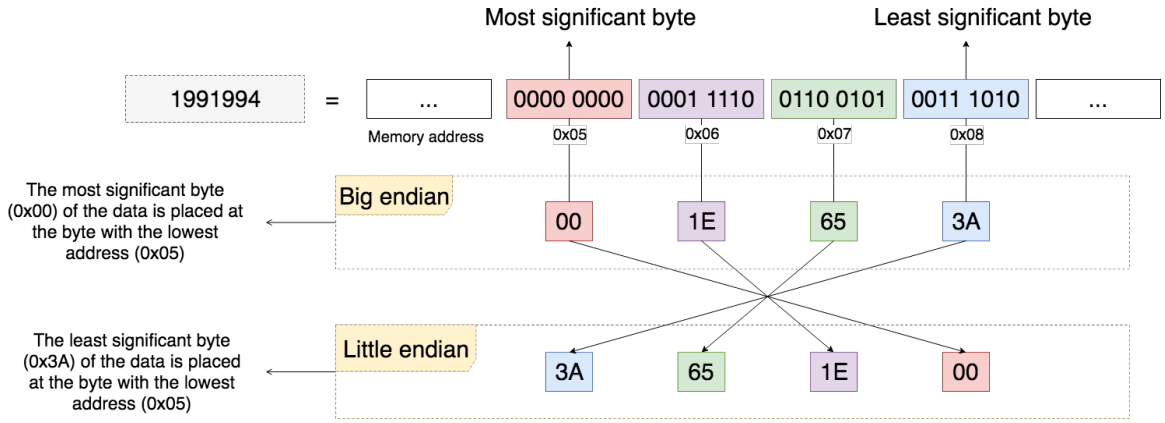

FIGURE 16 – Illustration des deux types d'endianness [7]

Le problème vient que les protocoles internet sont codés en big endian, alors que l'architecture x86 code les nombres avec le format little endian. Cela pose problème quand, par exemple, on reçoit le type de protocole dans une trame ethernet. Si on reçoit un paquet IPv4, on aura :
- 0x0800 = 2048, en big endian,
- qui sera interprété comme étant : 0x0008 = 8, en little endian !

Nous devons donc changer l'ordre des octets pour pouvoir interpréter correctement les informations reçues. Ceci est fait pas les fonctions suivantes :

```c
uint16_t convert_endianess_16bits(uint16_t nb) {
    uint16_t result = ((nb >> 8)) | ((nb << 8));
    return result;
}
uint32_t convert_endianess_32bits(uint32_t nb) {
    uint32_t result = ((nb >> 24))                  // move 1-st byte to 4-th byte
                    | ((nb << 24))                  // move 4-th byte to 1-st byte
                    | ((nb >> 8) & 0x0000ff00)      // move 2-nd byte to 3-rd byte
                    | ((nb << 8) & 0x00ff0000);     // move 3-rd byte to 2-nd byte
    return result;
}
```

Exemples :

1. Si on veut changer l'endianness de 0x0008, le résultat sera :

$$\begin{aligned}(\text{nb} >> 8) \mid (\text{nb} << 8) &= (0\text{x}0008 >> 8) \mid (0\text{x}0008 << 8)\\ &= (0\text{x}0000) \mid (0\text{x}0800)\\ &= 0\text{x}0800\end{aligned}$$

2. Si on veut changer l'endianness de 0x11223344, on aura :

nb >> 24	= 0x11223344 >> 24	= 0x00000011
nb << 24	= 0x11223344 << 24	= 0x44000000
(nb >> 8) & 0x0000ff00	= (0x11223344 >> 8) & 0x0000ff00	= 0x00002200
(nb << 8) & 0x00ff0000	= (0x11223344 << 8) & 0x00ff0000	= 0x00330000

Et quand on combine le tout :

0x00000011 | 0x44000000 | 0x00002200 | 0x00330000 = 0x44332211

5.4 Structures pour les headers des protocoles

Comme pour les problèmes d'endianness (figure 5.3), le placement des champs dans la structure va influencer l'information qu'on obtiendra. Dans l'exemple suivant, nous avons été obligés de changer l'ordre des champs de quatre bits *ip_header_length* et *ip_version* parce que le compilateur les plaçait dans l'ordre inverse par rapport aux champs du protocole IPv4.

```
1  ////////////////////////////////////////////////////////////////////////////
2  // NOTE: the order of the fields might seem weird, it's because of how the  //
3  //       compiler places the fields                                         //
4  //         0       1       2       3                                        //
5  //         01234567012345670123456701234567                                 //
6  // ip:     |ver|len|...                                                     //
7  // struct: |len|ver|...                                                     //
8  ////////////////////////////////////////////////////////////////////////////
9  struct ipv4_datagram {
10     u_char ip_header_length : 4;
11     u_char ip_version : 4;
12
13     // autres champs de la structure
14     ...
15 } typedef Ipv4Datagram;
```

Et avec la structure suivante (*TlsData*), nous avons dû ajouter l'attribut *packed* pour que le compilateur gcc place les champs de la structure correctement.

```
1 struct tls_data {
2     u_char content_type;
3     u_short version;
4     u_short length;
5     void *header;
6     void *data;
7 } __attribute__((packed)) typedef TlsData;
```

Pour comprendre pourquoi nous devons utiliser l'attribut *packed*, il suffit de comparer les figures 17 et 18. Sur la première, on voit que les champs de la structures ont étés alignés. En fait, ils sont alignés automatiquement par le compilateur avec l'alignement le plus grand possible sur l'architecture de la machine ciblée. Ceci est fait pour optimiser le temps du CPU [8]. Sur la seconde figure, on voit que tous les champs sont collés les uns aux autres. Puisque les données des protocoles réseaux se transmettent d'une machine à une autre, elles ne sont pas optimisées en fonction d'une architecture mais elles sont collées, nous devons donc utiliser l'attribut *packed*.

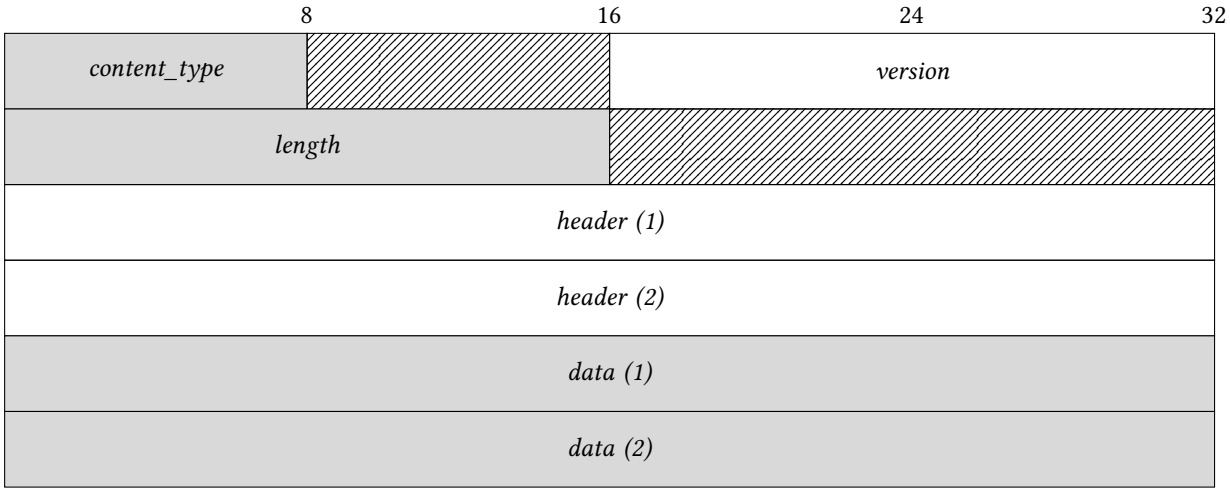

FIGURE 17 – Structure *TlsData* sans l'attribut *packed*

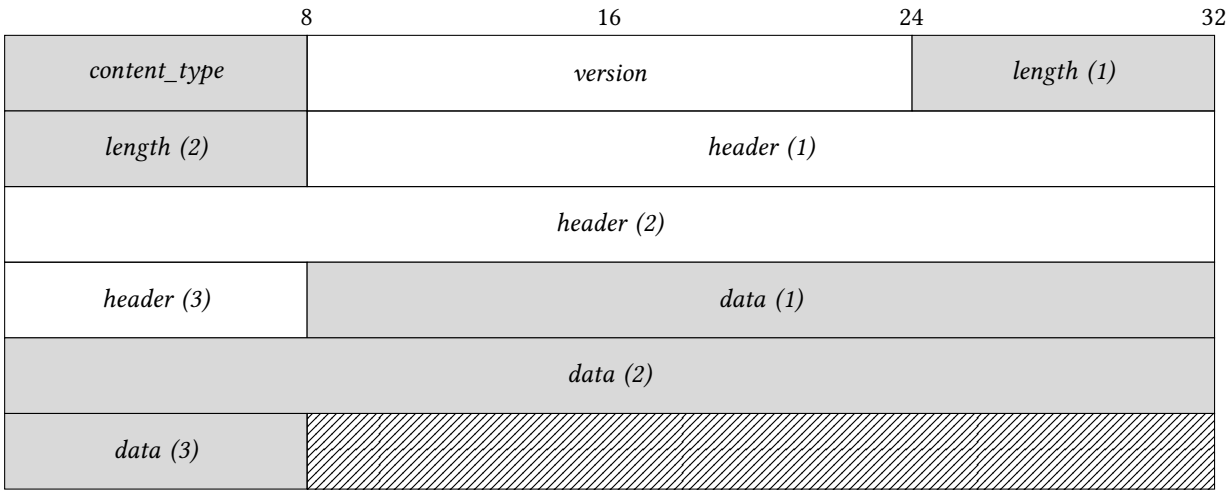

FIGURE 18 – Structure *TlsData* avec l'attribut *packed*

5.5 Passer des arguments à la fonction *packet_handler*

Pour vérifier qu'un paquet n'enfreint pas un règle, il faut deux chose : le paquet et les règles. La fonction *packet_handler* reçoit les paquets en argument mais pas les règles, cependant c'est possible de lui donner des informations en utilisant le paramètre *user* (qui est un pointeur) de la fonction *pcap_loop*. Afin d'organiser les données que nous allons transmettre à *packet_handler*, nous avons créés la structure *UserArgsPacketHandler*.

```
1 struct user_args_packet_handler {
2     int nb_rules;
3     Rule *rules;
4     IdsArguments ids_arguments;
5 } typedef UserArgsPacketHandler;
```

Table des matières

Table des figures

Références

[1] https://github.com/groumache/Intrusion-Detection-System

[2] https://suricata.readthedocs.io/en/latest/rules/intro.html

[3] https://qr.ae/pNSWwR

[4] https://stackoverflow.com/a/16268404/10524378

[5] https://fr.wikipedia.org/wiki/Liste_de_ports_logiciels

[6] https://www.seas.upenn.edu/~cis522/lecture_notes/lec11.pdf

[7] https://uynguyen.github.io/2018/04/30/Big-Endian-vs-Little-Endian/

[8] https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Attributes.html