



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.

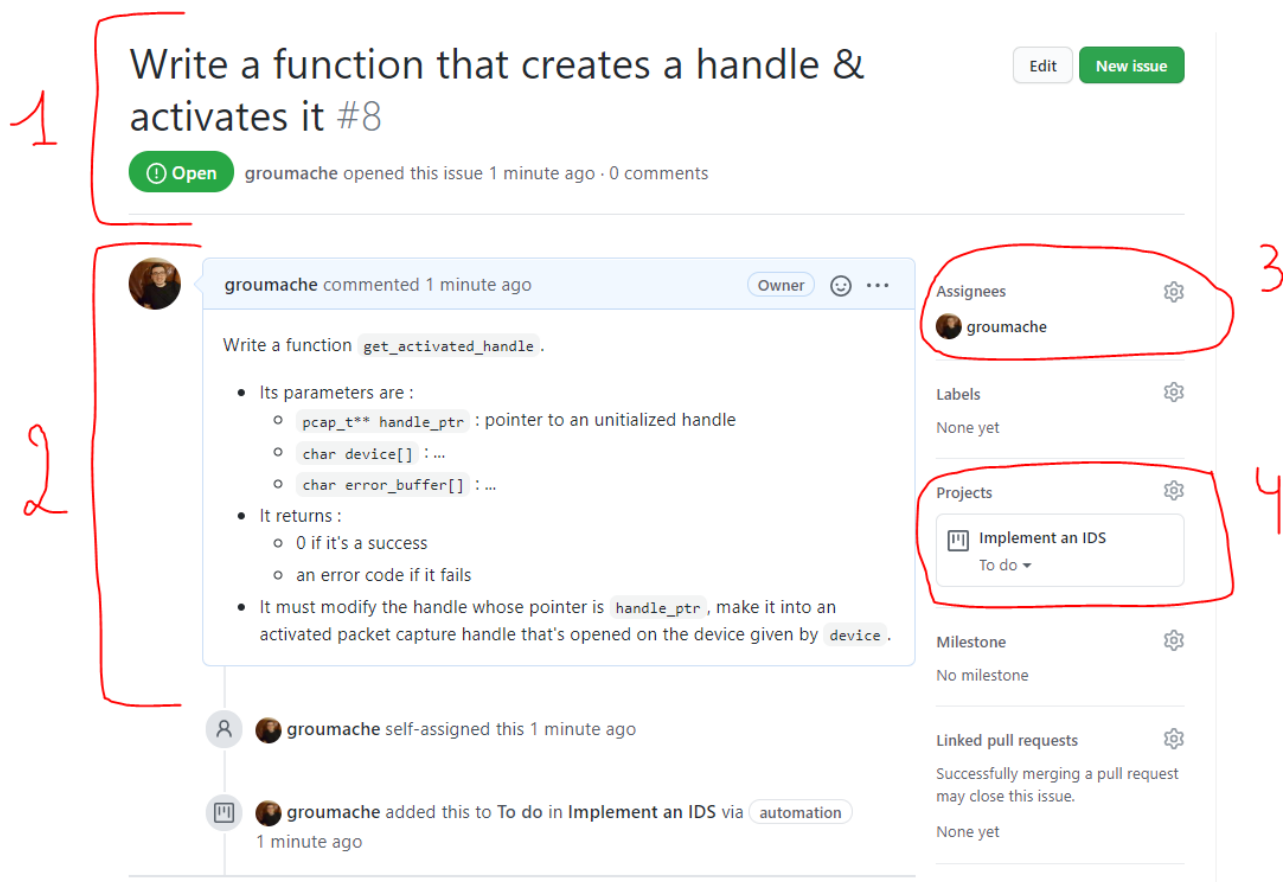


FIGURE 1 – Une issue sur github

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.

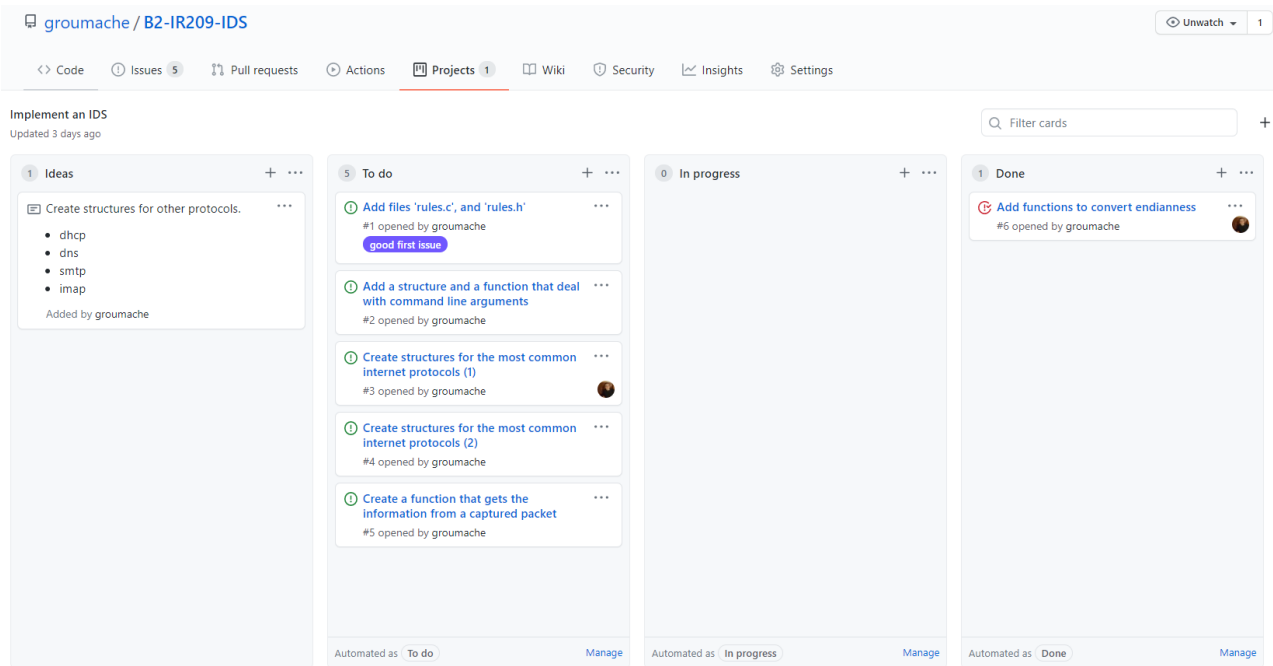


FIGURE 2 – Kanban sur github

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.

Network graph

Timeline of the most recent commits to this repository and its network ordered by most recently pushed to.

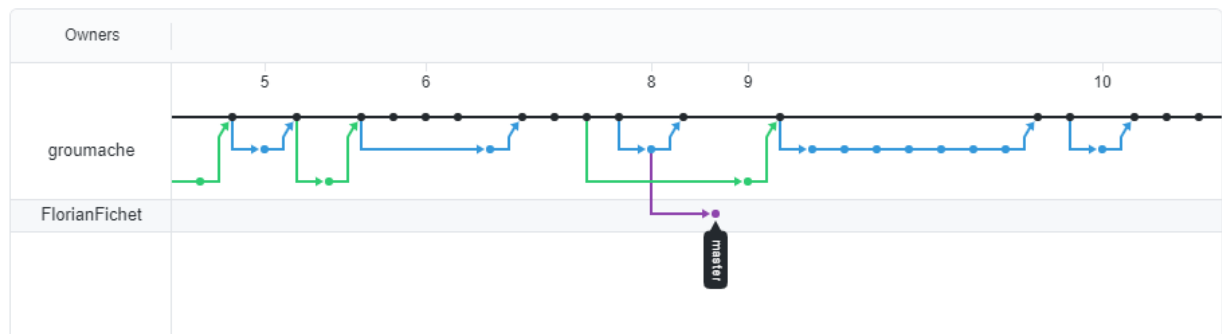


FIGURE 3 – Network graph montrant les commits et comment les branches se rejoignent

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 error.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 --print-logs --print-all`

Remarques :

- 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.
- Les chemins des fichiers devront être modifiés si on clone le répertoire github sur une autre machine.

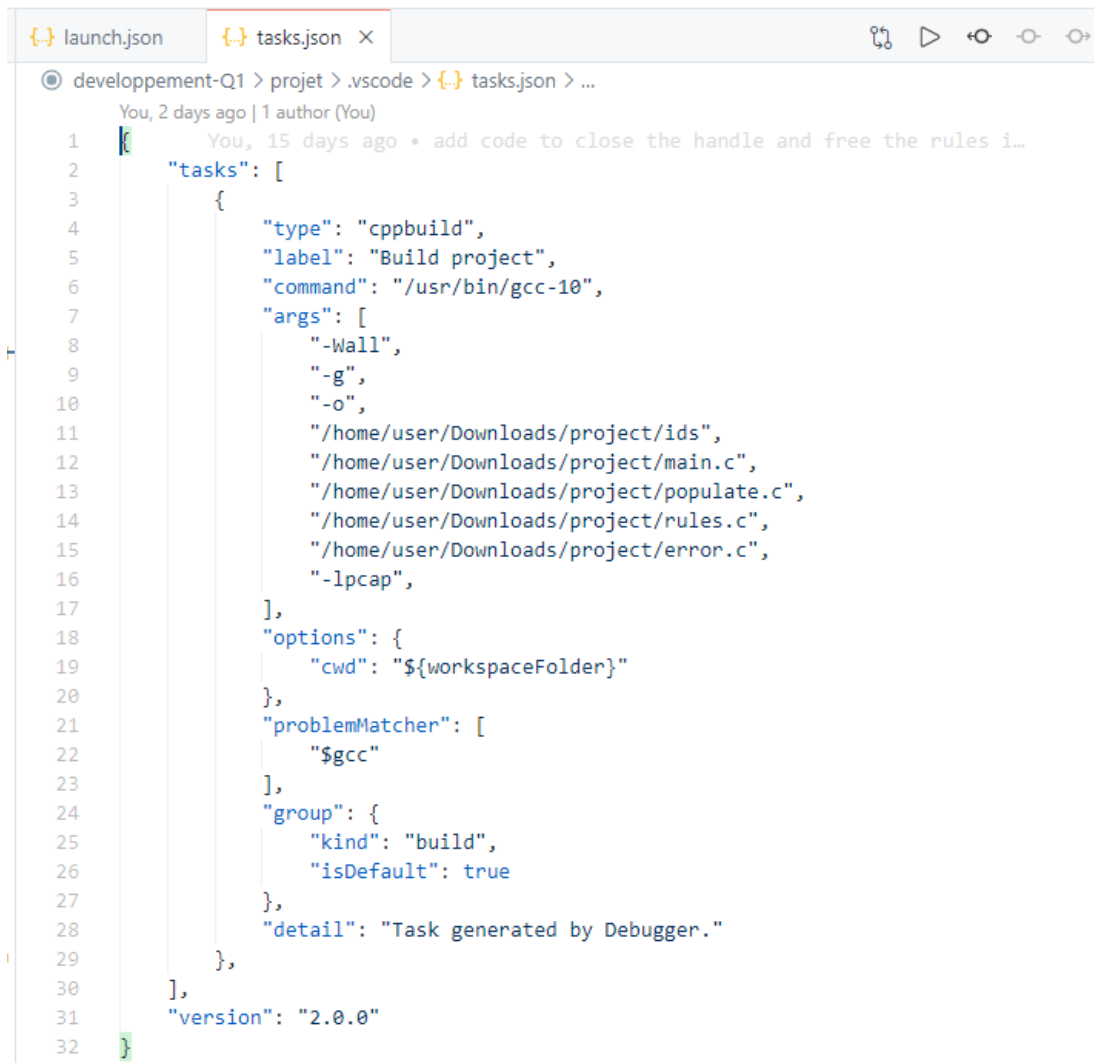


FIGURE 4 – Fichier de configuration du debugger, pour la compilation du programme



```

1  {
2      // Use IntelliSense to learn about possible attributes.
3      // Hover to view descriptions of existing attributes.
4      // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
5      "version": "0.2.0",
6      "configurations": [
7          {
8              "name": "Debug project",
9              "type": "cppdbg",
10             "request": "launch",
11             "program": "/home/user/Downloads/project/ids",
12             "args": [
13                 // "-h",
14                 "-p",
15                 "-d", "eth1",
16                 "-r", "/home/user/Downloads/project/ids.rules",
17                 "-n", "10",
18                 "--print-logs",
19                 "--print-all",
20             ],
21             "stopAtEntry": false,
22             "cwd": "${workspaceFolder}",
23             "environment": [],
24             "externalConsole": false,
25             "MIMode": "gdb",
26             "setupCommands": [
27                 {
28                     "description": "Enable pretty-printing for gdb",
29                     "text": "-enable-pretty-printing",
30                     "ignoreFailures": true
31                 }
32             ],
33             "preLaunchTask": "Build project",
34             "miDebuggerPath": "/usr/bin/gdb"
35         },
36     ],
37 }

```

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 libérons pas, valgrind va nous rappeler à l'ordre et même nous dire à quelle ligne le bloc mémoire a 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
}

```

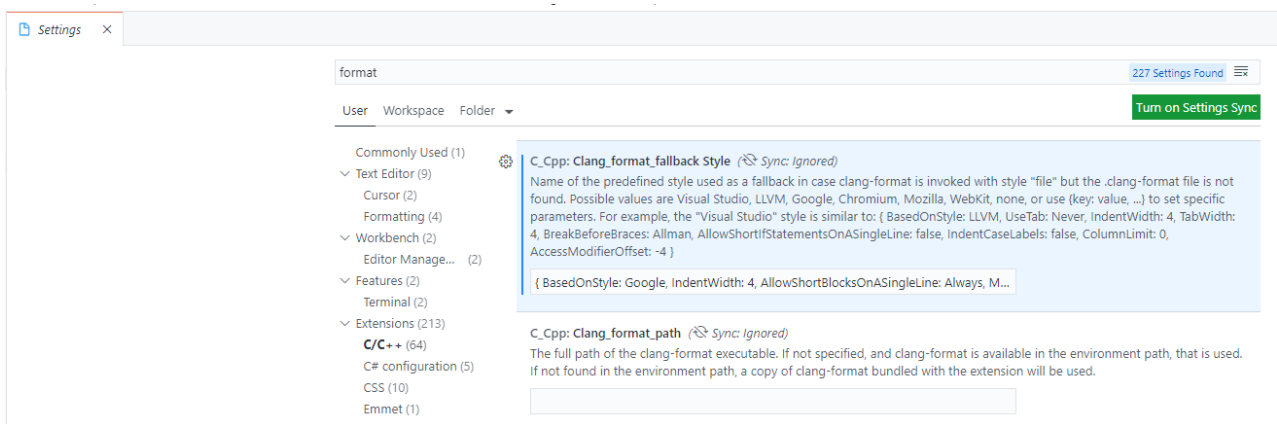


FIGURE 7 – Paramètre de formatage dans visual studio code

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

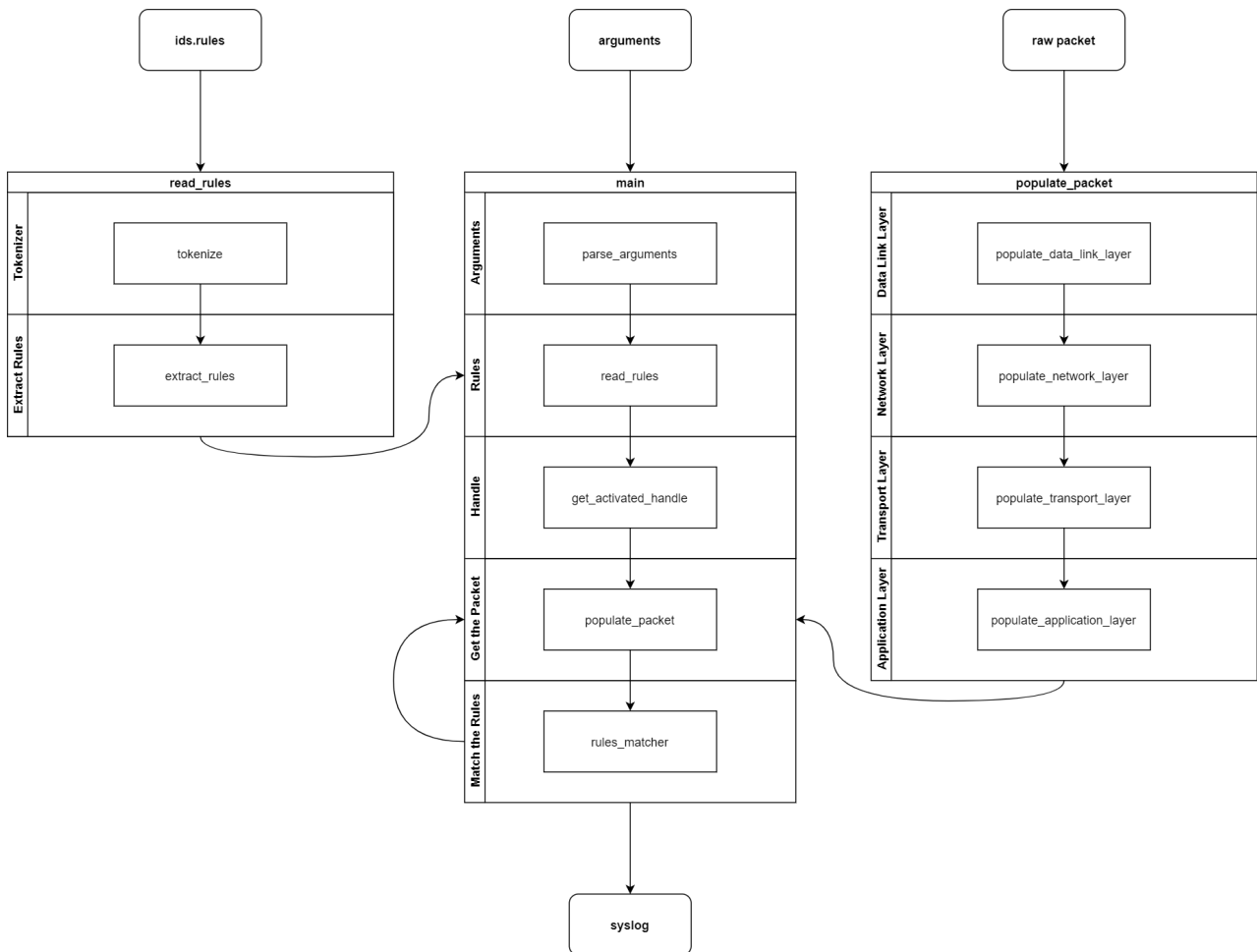


FIGURE 8 – Architecture du système

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.

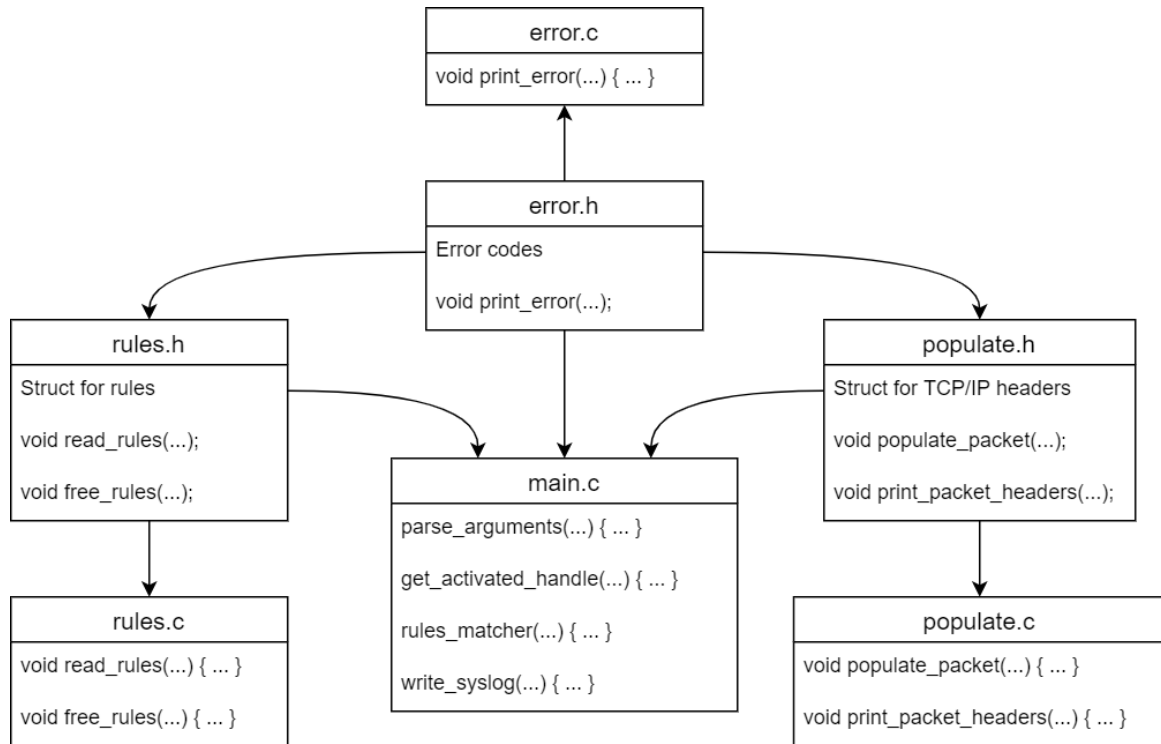


FIGURE 9 – Structure du projet

4 Organisation générale du code

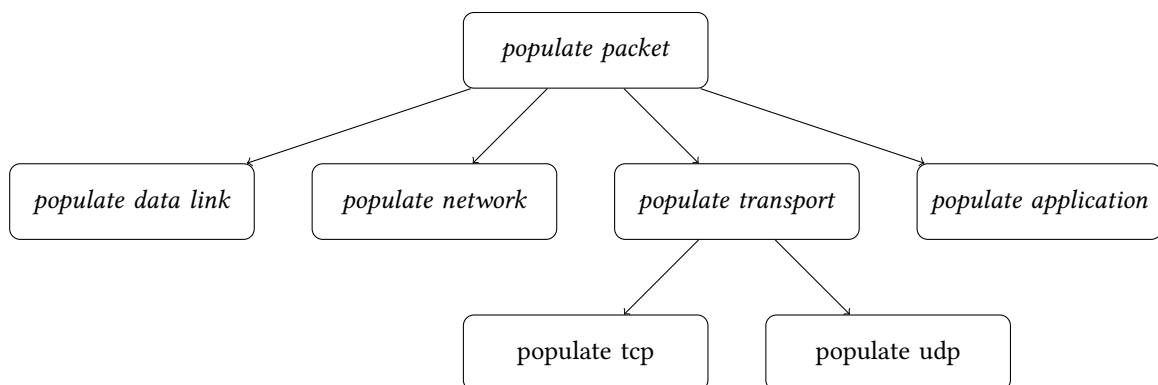
Dans cette section, nous allons expliquer comment le code fonctionne de manière générale mais sans entrer dans le détail. Les choix d'implémentation spécifiques seront expliqués dans la section suivante (section 5).

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 protocoles utilisés dans le paquet
- *print_packet_data*, affiche les données contenues dans le protocole de dernière couche (HTTP, TLS, FTP, 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.

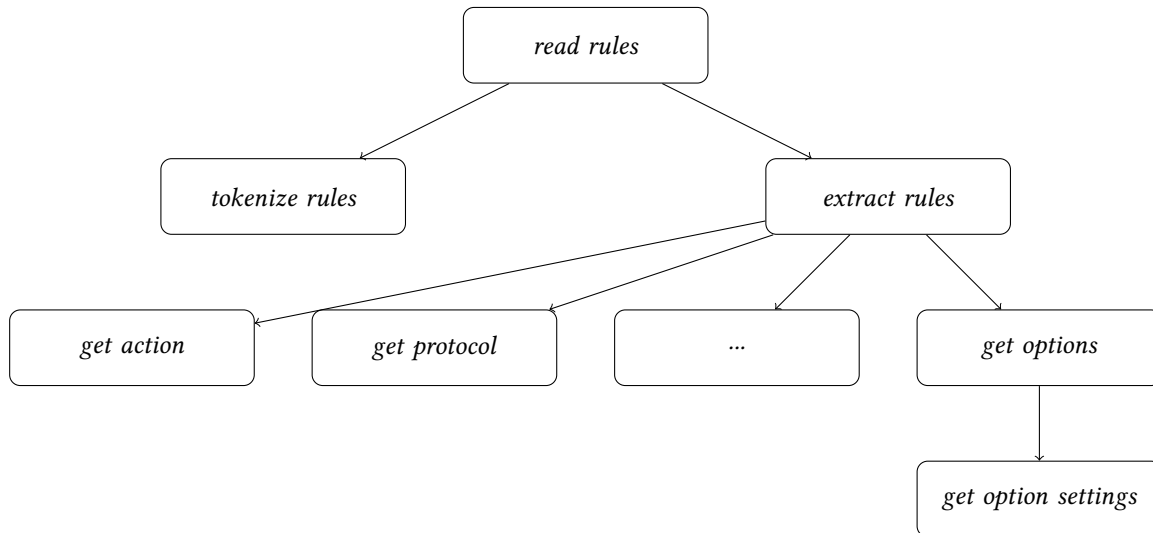


FIGURE 11 – Organisation des fonctions utilisées pour extraire les règles

Les fonctions pour obtenir les IPs et les ports sont plus compliquées parce que nous avons suivi le format de règle de *suricata* [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 appelée 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 Fichier `error.c`

Ce fichier ne comporte qu'une seule fonction, `print_error`. Cette fonction a pour but d'afficher un message d'erreur en rapport à l'erreur rencontré dans le programme IDS.

Cette fonction a sa définition dans le fichier `error.h`. Celui-ci comporte également :

- Une énumération `Error_type`, qui liste les différentes erreurs que le programme peut rencontrer.

Lorsqu'une fonction du programme rencontrera une erreur, elle renverra une valeur de type `int` contenue dans l'énumération. Une fonction qui se termine sans erreur renvoie généralement la valeur 0. Dans l'énumération, `not_error` équivaut à 0, nous pouvons donc renvoyer `not_error` à la fin d'une fonction pour plus de clarté.

4.5 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 recommandation 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 numéros 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 puisqu'une communication TCP/UDP est identifiée en utilisant ces ports, elle ne peut pas les changer [4]. Il n'y a pas non plus besoin de garder une trace de la communication avec une variable (le port utilisé est suffisant pour identifier la communication).

Il y a deux problèmes à cette approche :

- Le protocole TLS peut être utilisé sur d'autres ports mais sa détection devient ardue puisqu'il faut dès lors chercher la présence du header TLS dans tous les paquets peut importe leur port. Ceci augmente aussi le risque de signalement de paquets n'utilisant pas le TLS.
- Du trafic légitime pourrait passer sur un port réservé au protocole TLS. Cependant, il y a peu de chance que cela arrive puisque le port est réservé.

En résumé :

le trafic non-crypté ne doit pas passer sur le port 443	} nous ne vérifions que le port 443
le trafic crypté passant par d'autres ports est trop difficile à vérifier	

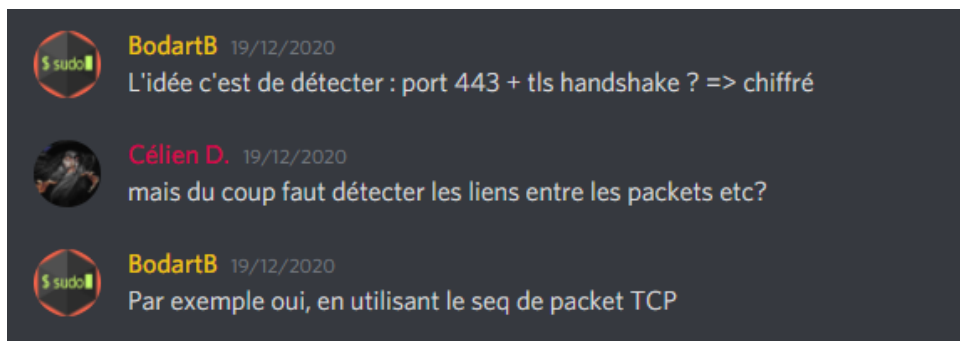


FIGURE 14 – Recommandation pour détecter et garder trace des paquets chiffrés

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* (appelée 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.

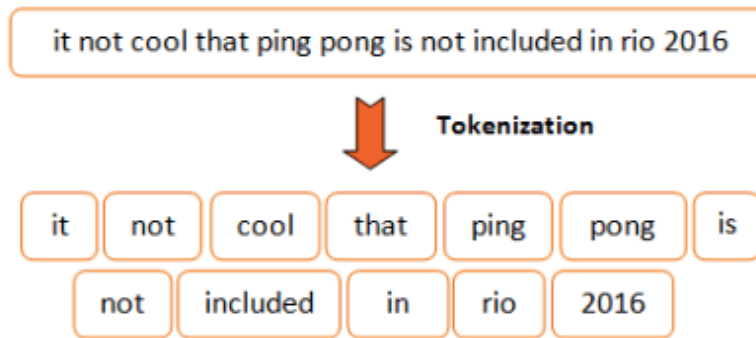


FIGURE 15 – Illustration de la tokenisation [6]

Nous avons mis ci-dessous la fonction *extract_rules* qui extrait les règles des tokens et les met dans les 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 écrite 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 retrouvés face à des quantités inconnues. Par exemple, combien de règles allons nous avoir dans le fichier de règles? Pour résoudre ce problème, nous avons ajouté 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));
4 }

```

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).

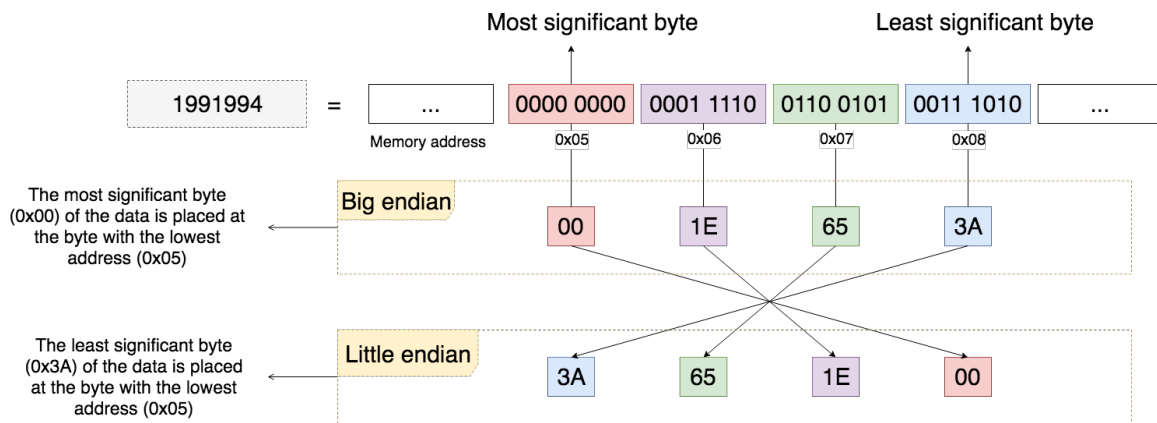


FIGURE 16 – Illustration des deux types d'endianness [7]

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 par les fonctions suivantes :

```

1 uint16_t convert_endianness_16bits(uint16_t nb) {
2     uint16_t result = ((nb >> 8) | ((nb << 8)));
3     return result;
4 }
5 uint32_t convert_endianness_32bits(uint32_t nb) {
6     uint32_t result = ((nb >> 24) // move 1-st byte to 4-th byte
7     | ((nb << 24) // move 4-th byte to 1-st byte
8     | ((nb >> 8) & 0x0000ff00) // move 2-nd byte to 3-rd byte
9     | ((nb << 8) & 0x00ff0000); // move 3-rd byte to 2-nd byte
10    return result;
11 }

```

Exemples :

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

$$\begin{aligned} (nb \gg 8) | (nb \ll 8) &= (0x0008 \gg 8) | (0x0008 \ll 8) \\ &= (0x0000) | (0x0800) \\ &= 0x0800 \end{aligned}$$

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

$$\begin{aligned} nb \gg 24 &= 0x11223344 \gg 24 &= 0x00000011 \\ nb \ll 24 &= 0x11223344 \ll 24 &= 0x44000000 \\ (nb \gg 8) \& 0x0000ff00 &= (0x11223344 \gg 8) \& 0x0000ff00 &= 0x00002200 \\ (nb \ll 8) \& 0x00ff0000 &= (0x11223344 \ll 8) \& 0x00ff0000 &= 0x00330000 \end{aligned}$$

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.

```

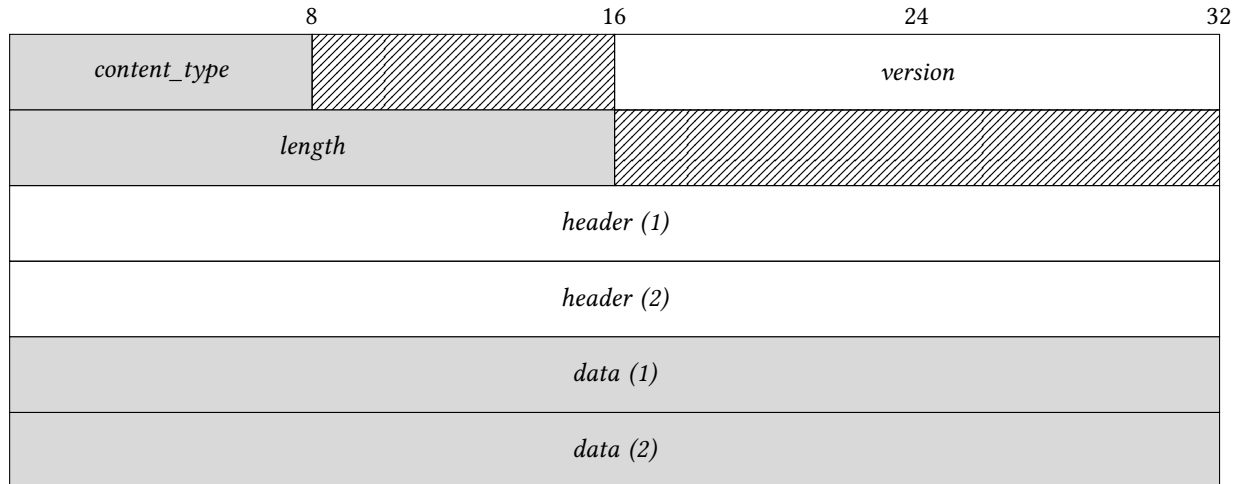
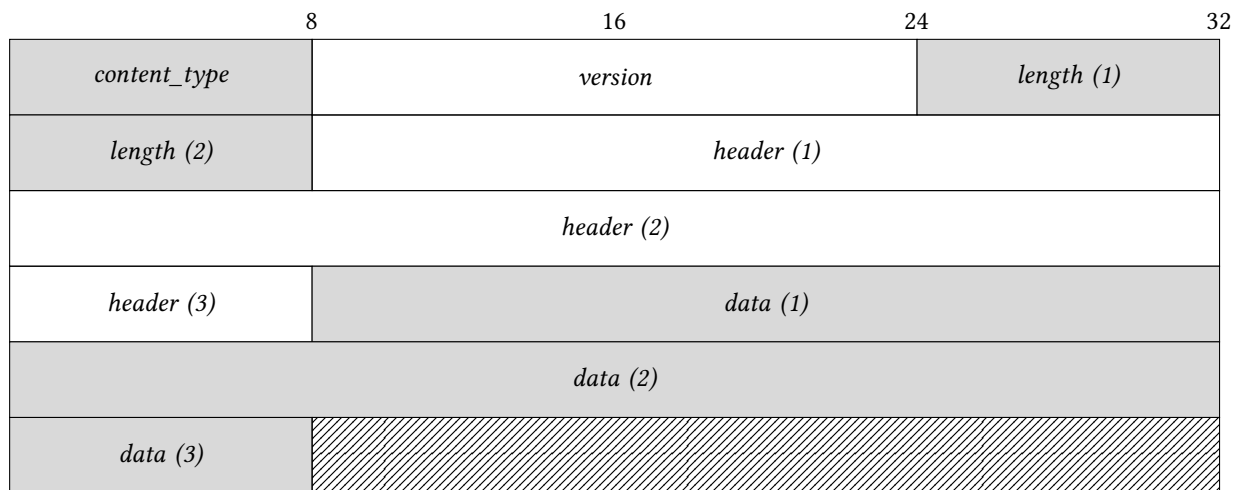
1 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2 // NOTE: the order of the fields might seem weird, it's because of how the //
3 // compiler places the fields //
4 //      0      1      2      3 //
5 //      01234567012345670123456701234567 //
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 été alignés. En fait, ils sont alignés automatiquement par le compilateur en fonction de leurs tailles et de 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 une règle, il faut deux choses : 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 créé la structure *UserArgsPacketHandler*.

```

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

5.6 Fonctions récursives

Les seules fonctions qui font des appels "récursifs" dans ce projet sont dans le fichier *rules.c* et servent à extraire les IP ou les ports à partir des tokens. Par exemple, pour les listes, on pourrait avoir un cas comme ceci (format accepté représenté sur la figure 12) :

$$\langle ip \rangle = [\langle ip \rangle, ![\langle ip \rangle, \langle ip \rangle]]$$

Et donc pour simplifier l'implémentation, on appelle récursivement la fonction *get_rule_ip* après "!", "[" et ";".

Remarque : ce n'est pas une récursivité directe mais une récursivité indirecte puisque les fonctions vont s'appeler comme sur la figure 19.

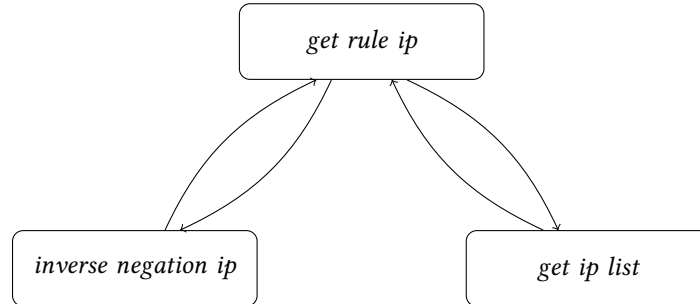


FIGURE 19 – Récursivité indirecte pour extraire les IPs

5.7 Les shifts de bits

Nous utilisons des shifts à plusieurs endroits dans notre code. Il y a notamment une utilisation triviale pour générer un string représentant une IPv4 (fonction *get_ipv4_address_string*, identique à la fonction *generate_ip* donnée dans le code de base avec l'énoncé). Nous avons aussi couvert les fonction qui convertissent l'endianness dans la section 5.3.

Le code suivant est utilisé dans *rules.c* pour faire l'inverse de *get_ipv4_address_string*, c'est-à-dire convertir un string IPv4 en un entier. Exemple :

- `ip_str = "255.0.0.0"`
- `num_byte` est initialisé à 3 et est décrémenté après chaque octet
- le buffer va contenir chaque byte successivement ("255", "0", "0", "0")
- `byte` est la conversion du buffer en entier (255, 0, 0, 0)
-

$$\begin{aligned} ip_int &= (255 \ll (8 * 3)) + (0 \ll (8 * 2)) + (0 \ll (8 * 1)) + (0 \ll (8 * 0)) \\ &= (255 \ll 24) + (0 \ll 16) + (0 \ll 8) + (0 \ll 0) \\ &= 255 \ll 24 \\ &= 0xFF000000 \end{aligned}$$

```

1 // get the number from the buffer
2 int byte = atoi(buffer);
3
4 // add the byte to ip_int
5 (*ip_int) += byte << (8 * num_byte);
6
7 num_byte--;

```

Dans le code suivant, qui vient de la fonction *check_ipv4_match* dans le fichier *main.c*, on va vérifier que l'adresse IPv4 du paquet (entier non-signé de 32-bits) correspond à l'adresse IP de la règle (entier non-signé de 32-bits, avec un netmask au format CIDR). L'idée va être de comparer l'adresse réseau des deux adresses. Exemple :

- `uint32_t ip = 3232235521 (= 192.168.0.1)`
- `addresses[i].netmask = 24 (= 255.255.255.0)`
- `addresses[i].ip = 3232235774 (= 192.168.0.254)`
- `wildcard_cidr = 32 - addresses[i].netmask = 8`
-

$$\begin{aligned} wildcard &= (1 \ll wildcard_cidr) - 1 \\ &= (1 \ll 8) - 1 \\ &= 256 - 1 = 255 \end{aligned}$$

```

—
uint32_t netmask = -1 - wildcard
                = 4294967295 - wildcard
                = 4294967295 - 255
                = 4294967040 (= 255.255.255.0)

—
(ip & netmask) == (addresses[i].ip & netmask)
= (192.168.0.1 & 255.255.255.0) == (192.168.0.254 & 255.255.255.0)
= 192.168.0.0 == 192.168.0.0

```

```

1 uint32_t wildcard_cidr = 32 - addresses[i].netmask;
2 uint32_t wildcard = (1 << wildcard_cidr) - 1;
3 uint32_t netmask = -1 - wildcard;
4
5 if (wildcard_cidr == 32 ||
6     (ip & netmask) == (addresses[i].ip & netmask)) {
7     ips_match = !addresses[i].negation;
8 }

```

Une dernière remarque par rapport au code ci-dessus : nous devons absolument ajouter la condition *wildcard_cidr == 32* pour vérifier qu'on ne fait pas un shift égal à la taille du nombre en bits (uint32_t est codé sur 32-bits). En effet, faire un shift négatif ou avec un nombre plus grand ou égal à la taille en bits du nombre est un comportement indéfini (undefined behavior) [9].

5.8 Variable statique

Pour pouvoir s'y retrouver plus facilement quand on affiche les header des paquets, nous avons ajouté une variable qui sert à numéroté les paquets dans la fonction *print_packet_headers*. Si nous avions utilisés une variable locale, l'information serait perdue à la fin de la fonction. Pour pouvoir la conserver, nous avons pensé à 3 possibilités :

- utiliser une variable globale,
- converver l'état de la variable dans la structure *UserArgsPacketHandler* qui est renvoyé à la réception de chaque paquet comme argument dans la fonction *packet_handler*,
- la troisième possibilité est d'utiliser une variable statique qui est déclarée dans la fonction mais qui n'est pas réinitialisée à chaque appel.

Nous avons décidé d'utiliser une variable statique comme dans le code ci-dessous.

```

1 void print_packet_headers(Packet *packet) {
2     static int i = 0;
3     printf("\nPacket n°%d:\n", ++i);
4
5     // print the packet headers
6     ...
7 }

```

5.9 Include guard

Les include guards (aussi appelés garde-fous) sont utilisés pour empêcher la double inclusion des fichiers headers [11]. Nous n'en avons eu besoin que pour le fichier *error.h* (voir figure 20). Il est aussi possible d'utiliser la directive de préprocesseur *#pragma once* au lieu des include guards. Cependant c'est une alternative moins portable car non-standard [10].

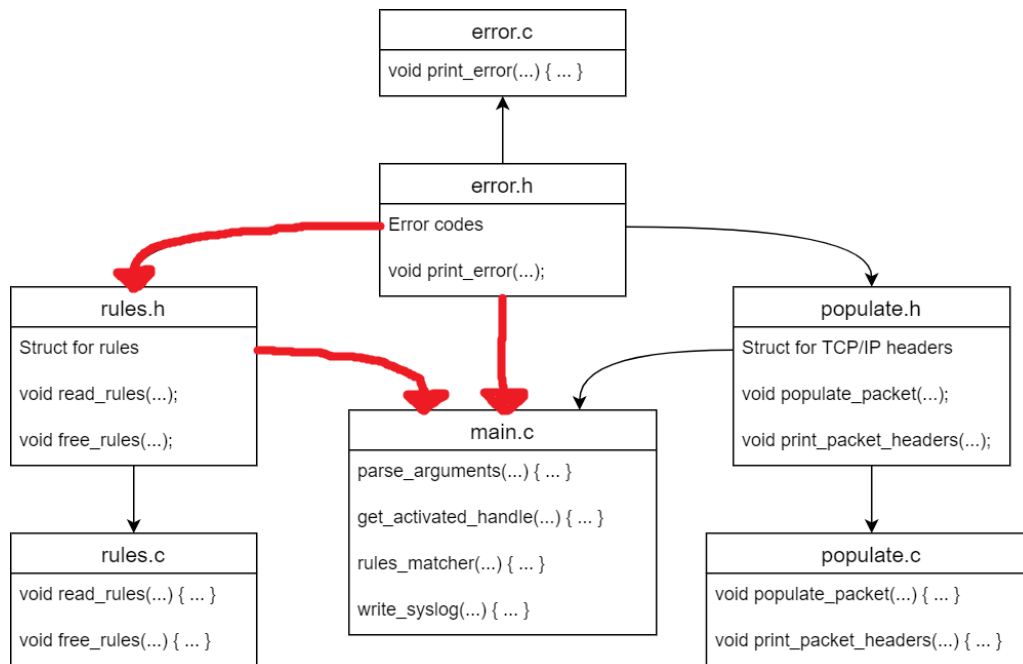


FIGURE 20 – Illustration de la double inclusion

6 Améliorations possibles

6.1 Changement au niveau des règles ou du fichier de règles

- Prendre en charge les adresses des protocoles IPv6 et Ethernet au niveau des règles.

Pour le moment, seules les adresses IPv4 peuvent être traitées par l'IDS. Comme la longueur et le format des adresses de ces trois protocoles (IPv4, IPv6, Ethernet) sont forts différents, on pourrait faire une fonction qui vérifie le format d'adresse utilisé (donc pas besoin que l'utilisateur précise le protocole lui-même).

Il faudrait aussi changer le code dans la fonction `read_rules` pour prendre en charge ces autres types d'adresses.

- Pouvoir écrire un message d'erreur qui précise aussi les numéros de ligne et colonne quand il y a un problème.

Il faudrait pouvoir écrire des message d'erreur quand le fichier `ids.rules` contient des erreurs. Par exemple, si il manque une virgule, une parenthèse fermante, etc. J'ai ajouté la figure 21 pour avoir un exemple de bon message d'erreur. Il montre les numéros de ligne et colonne, il affiche le contenu des lignes entourant l'erreur, et explique ce que le compilateur attendait.

```

error: expected one of `.` , `;` , `?` , `}` , or an operator, found `println`
--> main.rs:3:5
|
2 |     println!("Hello, world!")
|                                     - expected one of `.` , `;` , `?` , `}` , or an operator here
3 |     println!("Happy new year!")
|     ^^^^^^ unexpected token

error: aborting due to previous error
  
```

FIGURE 21 – Exemple de message d'erreur

- Ajouter la possibilité d'écrire des commentaires dans le fichier de règles.

Ce serait bien de pouvoir écrire des commentaires qui expliquent une règle. Par exemple, si on croise un `#` dans le fichier de règles, le reste de la ligne est ignoré par la fonction `tokenize_rules`. Le problème vient

évidemment du fait qu'un utilisateur de l'IDS peut vouloir utiliser un # dans l'option *content* d'une règle pour vérifier que certains paquets ne contiennent pas ce caractère. Il faut donc ajouter une possibilité d'échappement.

6.2 Prise en charge de plus d'actions

- Prendre en charge l'action *drop* pour pouvoir laisser tomber un paquet sans qu'il puisse être utilisé par d'autres programmes.

On pourrait créer des *iptables* pour envoyer les paquets dans une file pour les analyser et décider si on les laisse tomber ou pas, et ce avant qu'ils ne soient analysés [12].

- Prendre en charge l'action *reject* qui *drop* le paquet puis envoie un message "RST/ICMP unreachable error".
- Ajouter une option pour que notre IDS fasse l'équivalent d'une attaque *man in the middle* (mais légitime) afin d'analyser le contenu des paquets cryptés.

Certaines organisations ont une politique de sécurité informatique telle que les données cryptées doivent être décryptées puis analysées avant d'être éventuellement ré-encryptées [13]. Ceci est illustré par la figure 22.

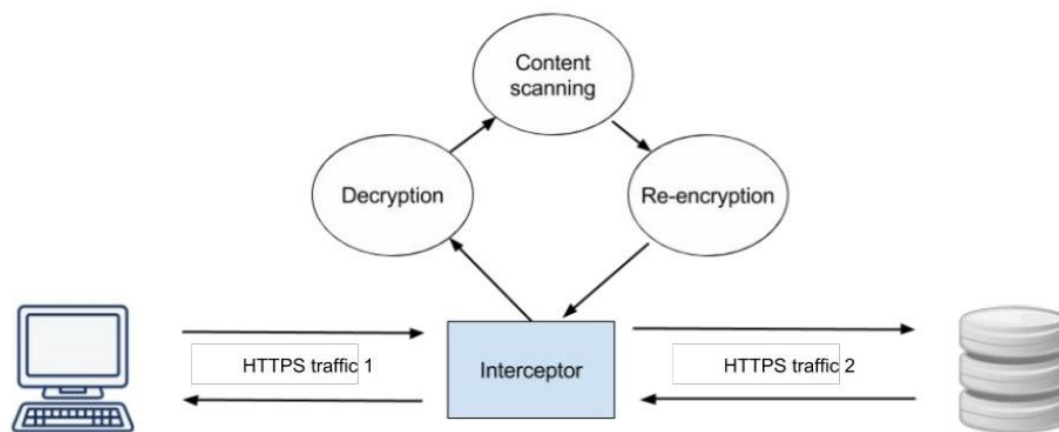


FIGURE 22 – Analyse TLS [13]

6.3 Autres améliorations

- Afficher une erreur quand les arguments donnés à l'IDS en ligne de commande sont incorrects.

Pour le moment, si un mauvais argument est donné à l'IDS, il est simplement ignoré. Ce serait mieux d'afficher un message d'erreur qui aide l'utilisateur à corriger son erreur.

- Ajouter des fichiers pour simplifier l'organisation du code.

Le nombre de lignes de code et de fonctions est plutôt élevé. Le projet gagnerait en simplicité si certaines fonctions, structures ou enums étaient placées dans d'autres fichiers. Par exemple, nous utilisons deux enums pour les protocoles (*RuleProtocol* et *PopulateProtocol*). Notre base de code serait simplifiée si on n'en utilisait qu'une, en la mettant dans un fichier header *protocol.h*.

7 Conclusion

Nous avons beaucoup appris avec ce projet, en particulier quand nous avons essayé de résoudre les problèmes rencontrés. Par exemple, nous savons maintenant comment gérer des structures de données dynamiques, travailler avec du binaire (shift, et/ou logique), utiliser des include guards, etc. Mais en-dehors de la programmation elle-même, nous avons appris beaucoup dans des sujets traités par le projet, comme l'utilisation des headers des protocoles réseau et leur

analyse en sécurité informatique. En conclusion, nous avons appris comment les IDS fonctionnent et nous en avons implémenté un simplifié.

Table des matières

1	Introduction	1
2	Organisation du travail et configuration des outils	1
2.1	Les issues sur github	1
2.2	Utilisation du kanban sur github	2
2.3	Configuration du debugger	3
2.4	Utilisation de valgrind	4
2.5	Configuration du formatage automatique	5
3	Architecture du projet	5
3.1	Architecture du système	5
3.2	Structure du projet	6
4	Organisation générale du code	7
4.1	Fichier <i>populate.c</i>	7
4.2	Fichier <i>rules.c</i>	8
4.3	Fichier <i>main.c</i>	8
4.4	Fichier <i>error.c</i>	9
4.5	Détection de l'encryption du payload	9
5	Choix d'implémentation et problèmes rencontrés	10
5.1	Tokens	10
5.2	Structures de données dynamiques	11
5.3	Problèmes d'endianness	11
5.4	Structures pour les headers des protocoles	12
5.5	Passer des arguments à la fonction <i>packet_handler</i>	13
5.6	Fonctions récursives	13
5.7	Les shifts de bits	14
5.8	Variable statique	15
5.9	Include guard	15
6	Améliorations possibles	16
6.1	Changement au niveau des règles ou du fichier de règles	16
6.2	Prise en charge de plus d'actions	17
6.3	Autres améliorations	17
7	Conclusion	17

Table des figures

1	Une issue sur github	1
2	Kanban sur github	2
3	Network graph montrant les commits et comment les branches se rejoignent	2
4	Fichier de configuration du debugger, pour la compilation du programme	3
5	Fichiers de configuration du debugger, pour lancer le programme	4
6	Résultat de valgrind	5
7	Paramètre de formatage dans visual studio code	5
8	Architecture du système	6
9	Structure du projet	7
10	Organisation des fonctions <i>populate</i>	7
11	Organisation des fonctions utilisées pour extraire les règles	8
12	Format d'IP acceptable dans le fichier de règles	8
13	Format de port acceptable dans le fichier de règles	8

14	Recommandation pour détecter et garder trace des paquets chiffrés	9
15	Illustration de la tokenisation [6]	10
16	Illustration des deux types d'endianness [7]	11
17	Structure <i>TlsData</i> sans l'attribut <i>packed</i>	13
18	Structure <i>TlsData</i> avec l'attribut <i>packed</i>	13
19	Récursivité indirecte pour extraire les IPs	14
20	Illustration de la double inclusion	16
21	Exemple de message d'erreur	16
22	Analyse TLS [13]	17

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://uynghuyen.github.io/2018/04/30/Big-Endian-vs-Little-Endian/>
- [8] <https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Attributes.html>
- [9] <https://stackoverflow.com/a/45433017/10524378>
- [10] <https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html>
- [11] https://fr.wikipedia.org/wiki/Include_guard
- [12] <https://www.linuxquestions.org/questions/linux-networking-3/capture-drop-packets-using-c-716868/#post3498563>
- [13] <https://qr.ae/pNJyLS>