



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.

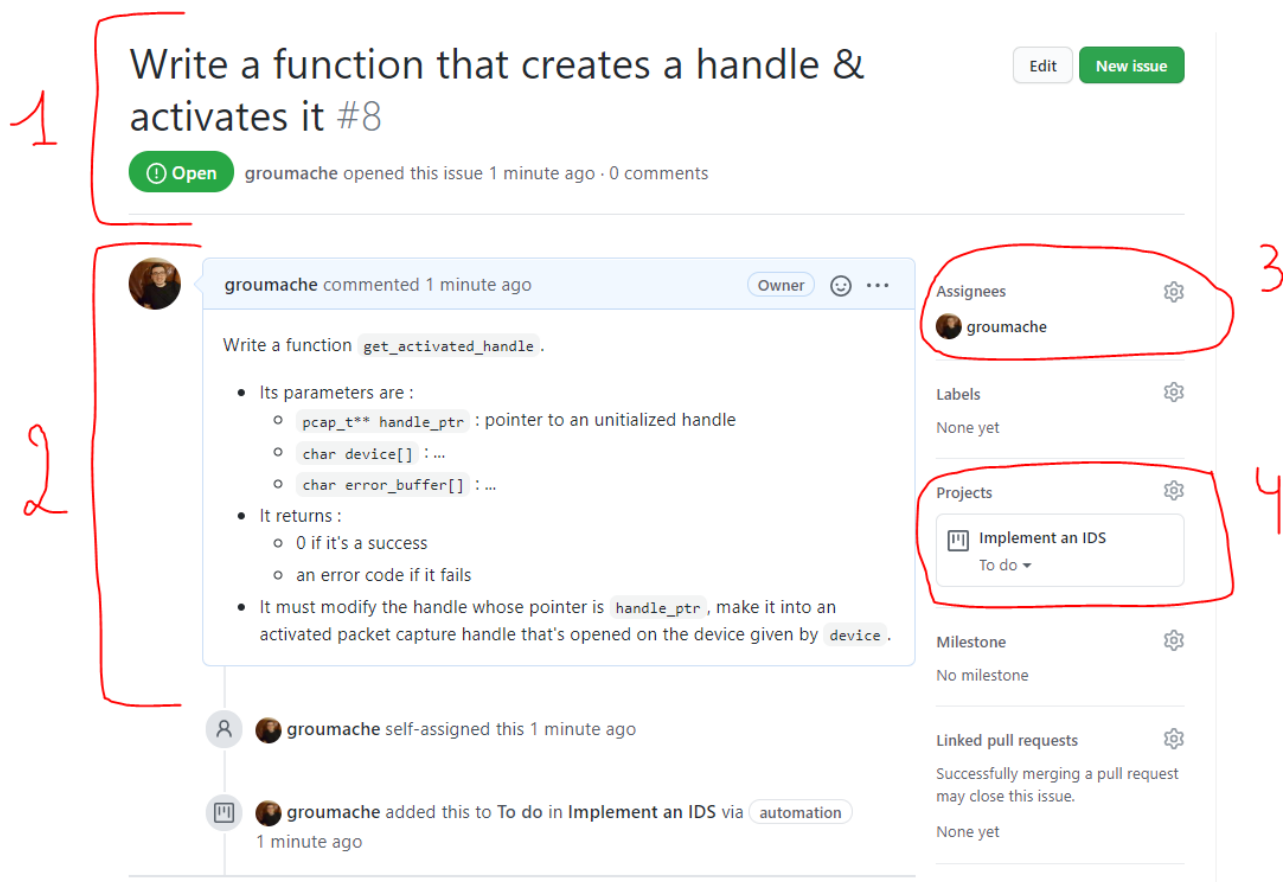


FIGURE 1 – Une issue sur github

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.

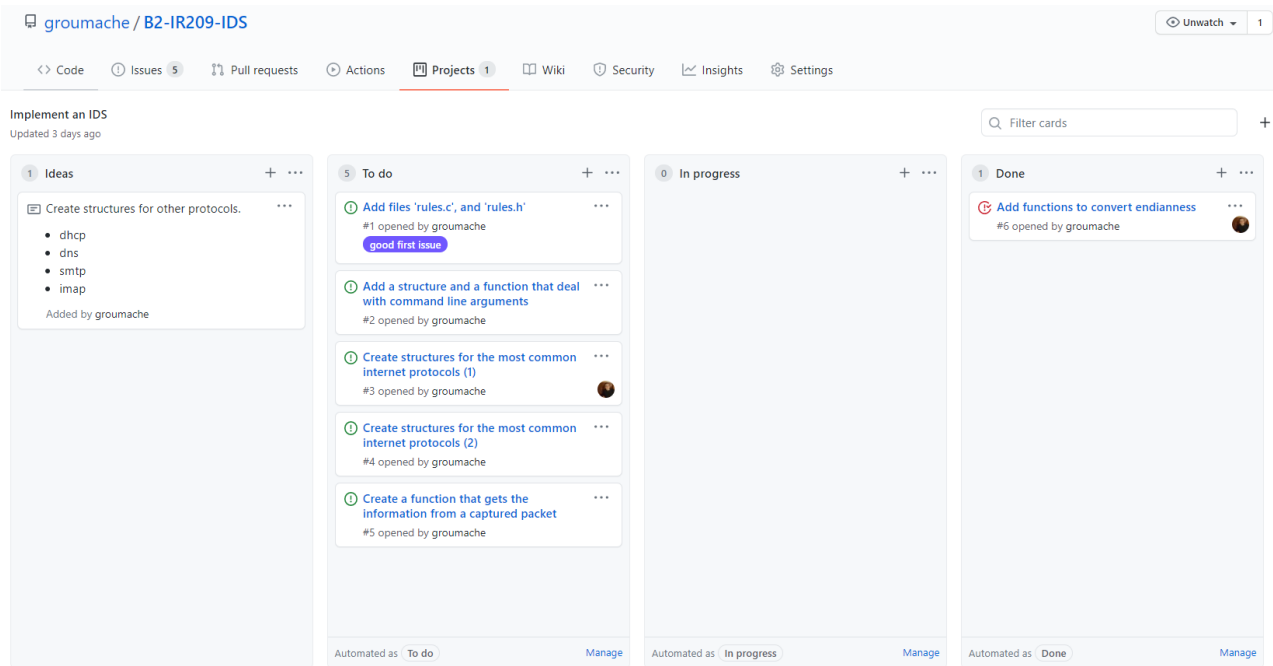


FIGURE 2 – Kanban sur github

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.

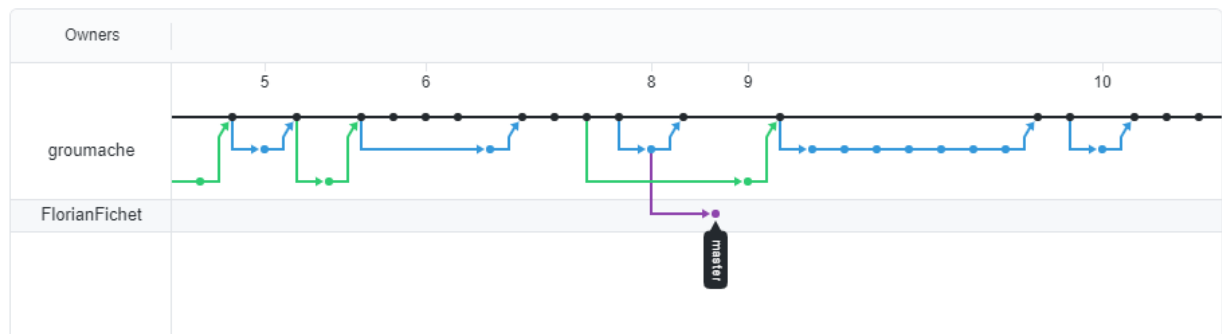


FIGURE 3 – Network graph montrant les commits et comment les branches se rejoignent

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 -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.

```

{
  "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

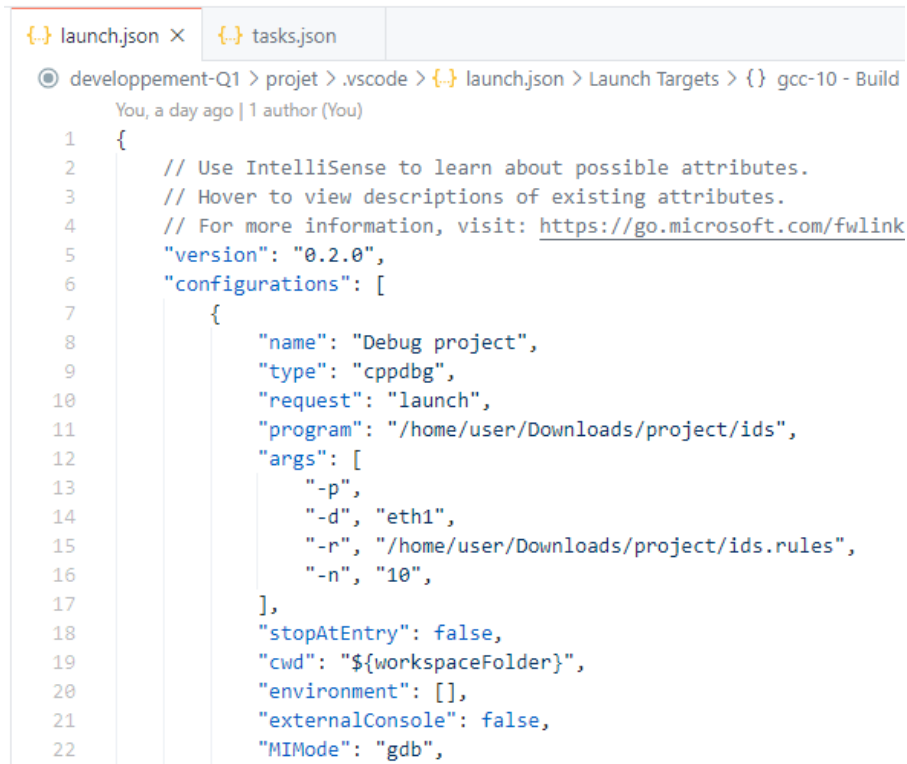


FIGURE 5 – Fichiers de configuration du debugger, pour lancer le programme

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
}
```

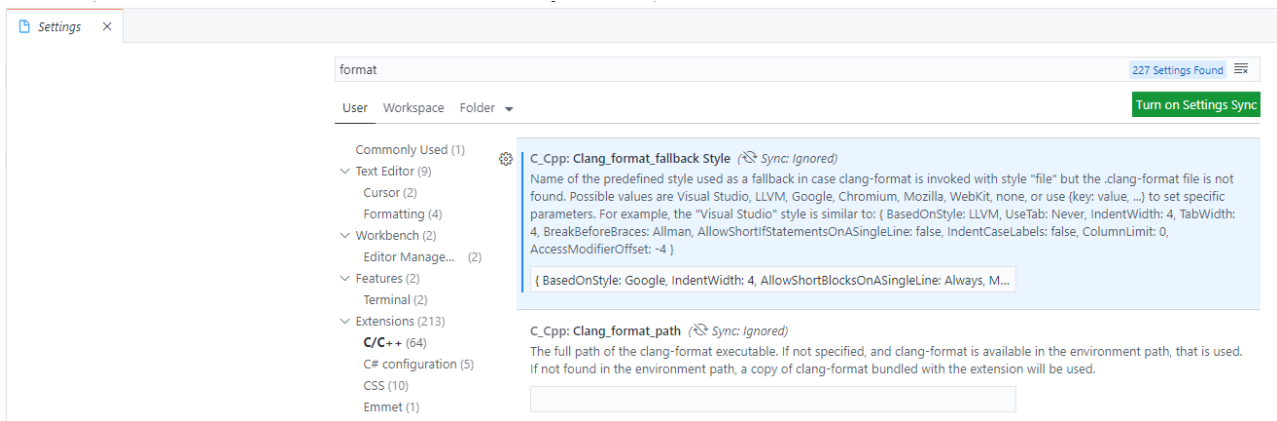


FIGURE 7 – Paramètre de formatage dans visual studio code

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

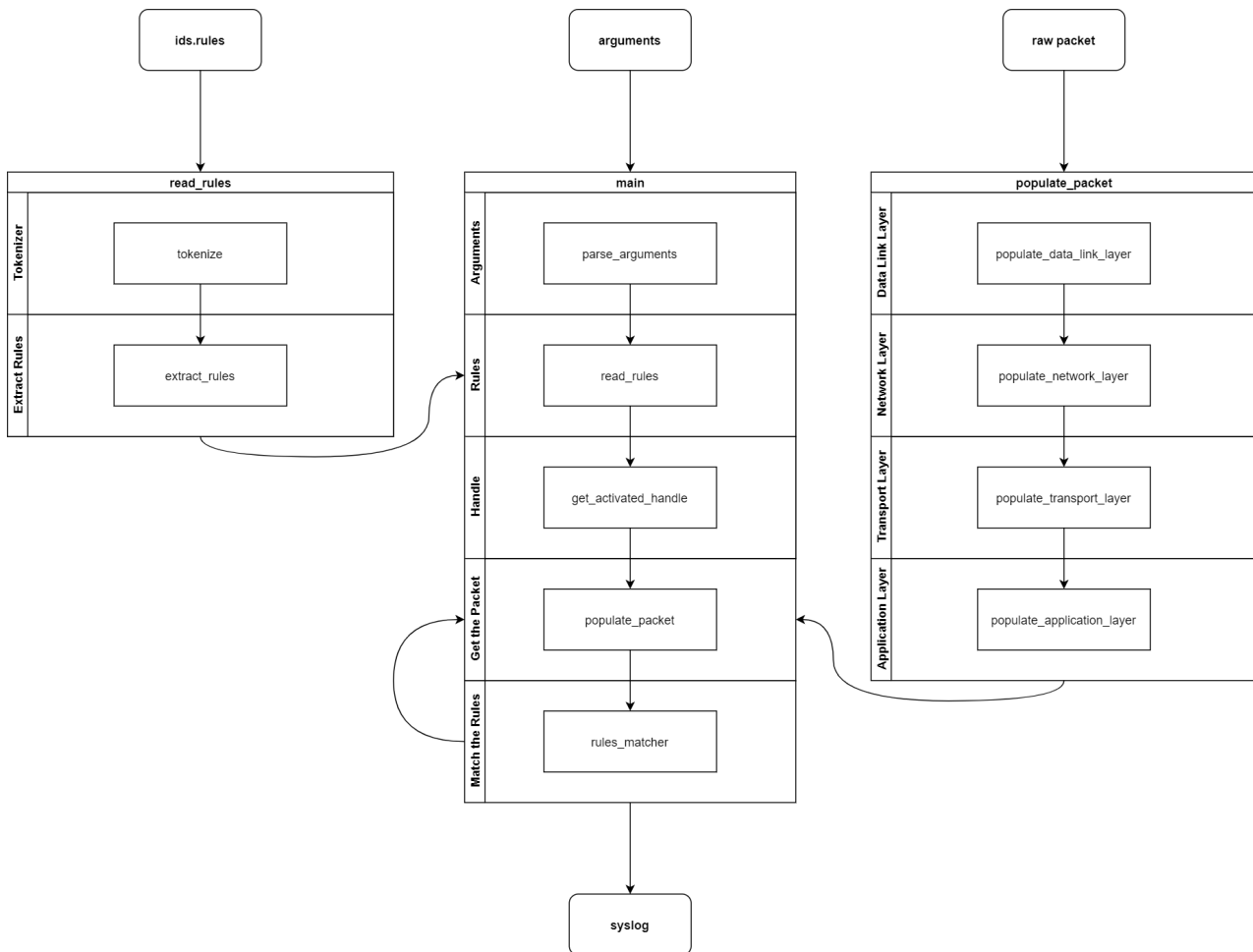


FIGURE 8 – Architecture du système

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.

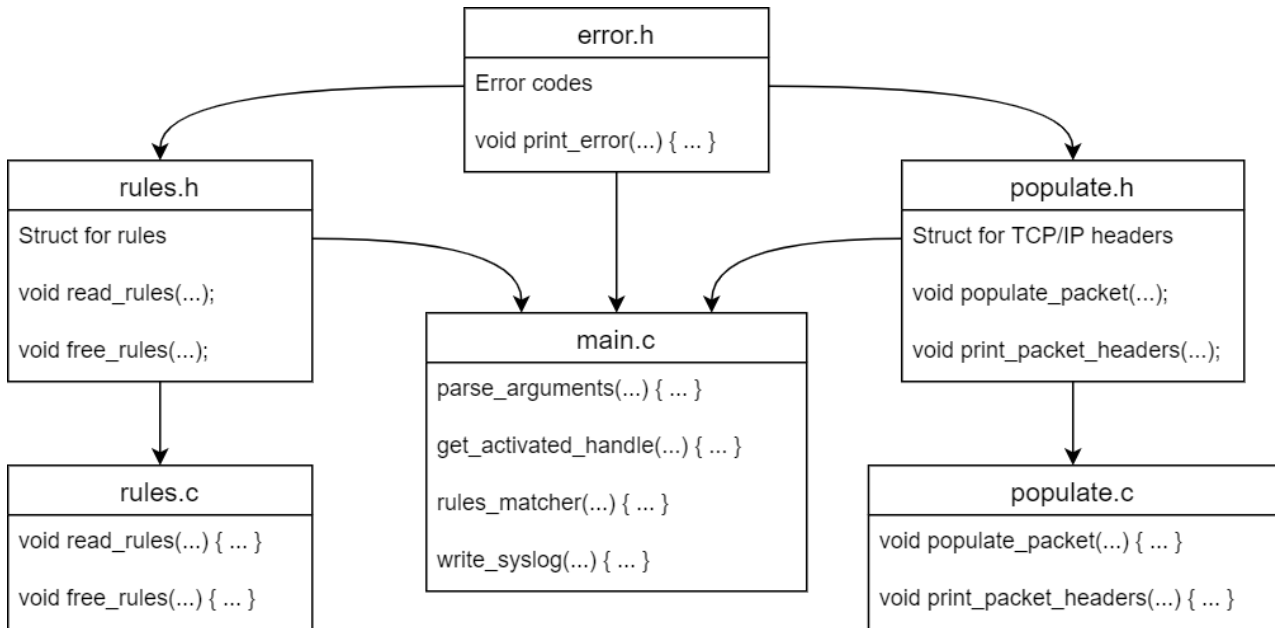


FIGURE 9 – Structure du projet

4 Organisation générale du code

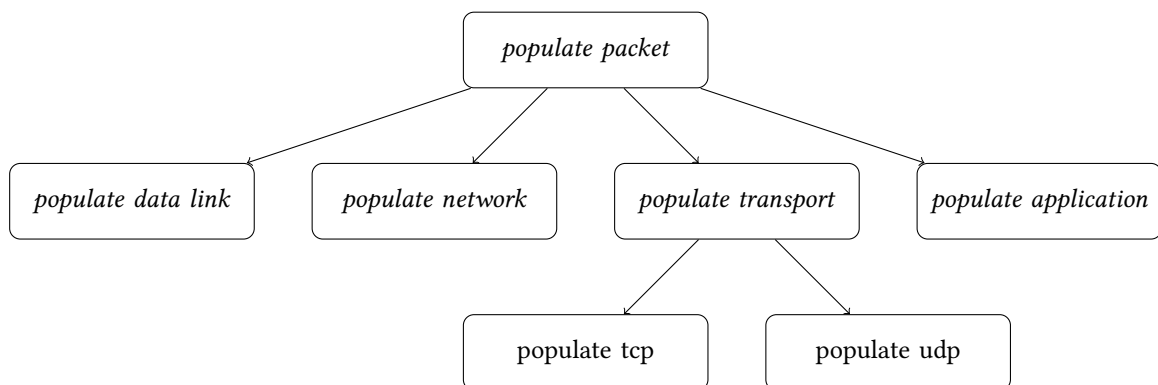
Dans cette section, nous allons expliquer comment le code fonctionne de manière générale mais sans entrer dans le détail. Les choix d'implémentation spécifiques seront expliqués dans la section suivante (section 5).

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 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.

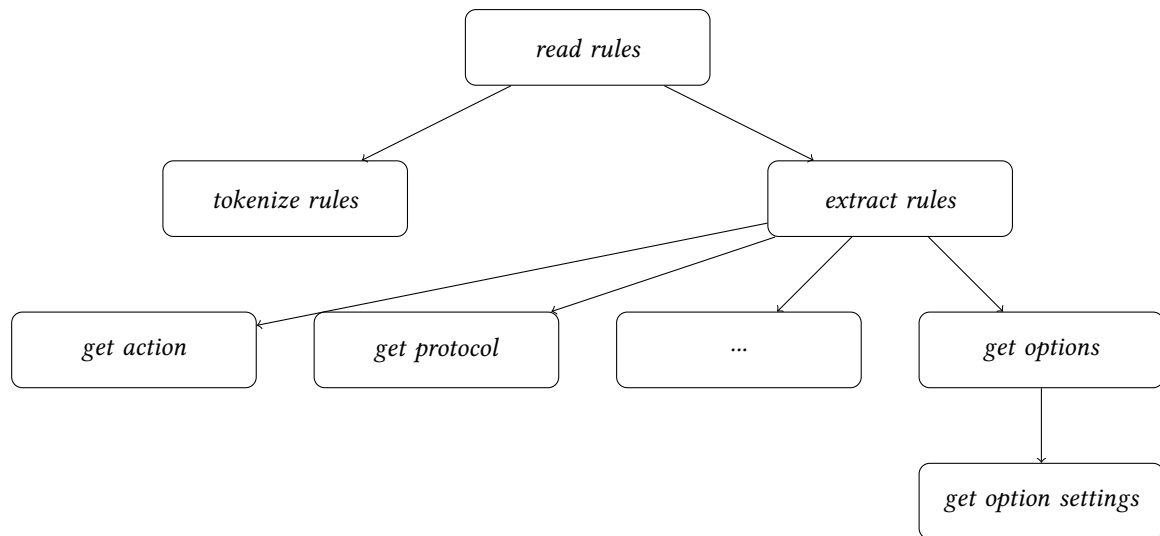


FIGURE 11 – Organisation des fonctions utilisées pour extraire les règles

Les fonctions pour obtenir les IPs et les ports sont plus compliquées parce que nous avons suivi le format de règle de *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 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 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 trafic légitime passerait sur un port réservé au protocole TLS. Cependant, il y a peu de chance que cela arrive puisque le port est réservé.

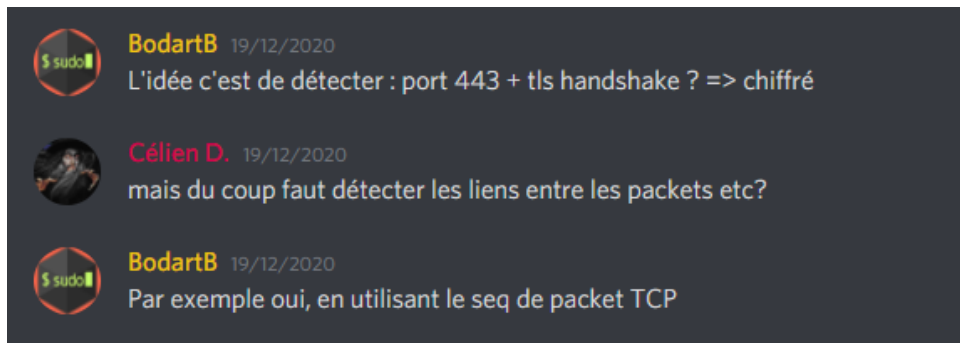


FIGURE 14 – Recommandation pour garder trace des paquets chiffrés

5 Choix d'implémentation et problèmes rencontrés

5.1 Tokens

5.2 Structure de données dynamique

5.3 Problèmes d'endianness

5.4 Structures pour header de protocole

5.5 Passer des arguments à la fonction *packet_handler*

5.6 Fonctions récursives

5.7 Macros et masques

5.8 ...

5.9 Variable statique

5.10 ...

Voici hello world :

```
1 #include <stdio.h>
2
3 // this is hello world
4 int main(void)
5 {
6     printf("Hello World!");
7 }
```

6 Améliorations possibles

7 Conclusion

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	4
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>	7
4.3	Fichier <i>main.c</i>	8
4.4	Détection de l'encryption du payload	9
5	Choix d'implémentation et problèmes rencontrés	10
5.1	Tokens	10
5.2	Structure de données dynamique	10
5.3	Problèmes d'endianness	10
5.4	Structures pour header de protocole	10
5.5	Passer des arguments à la fonction <i>packet_handler</i>	10
5.6	Fonctions récursives	10
5.7	Macros et masques	10
5.8	10
5.9	Variable statique	10
5.10	10
6	Améliorations possibles	10
7	Conclusion	10

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	4
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 garder trace des paquets chiffrés	9

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