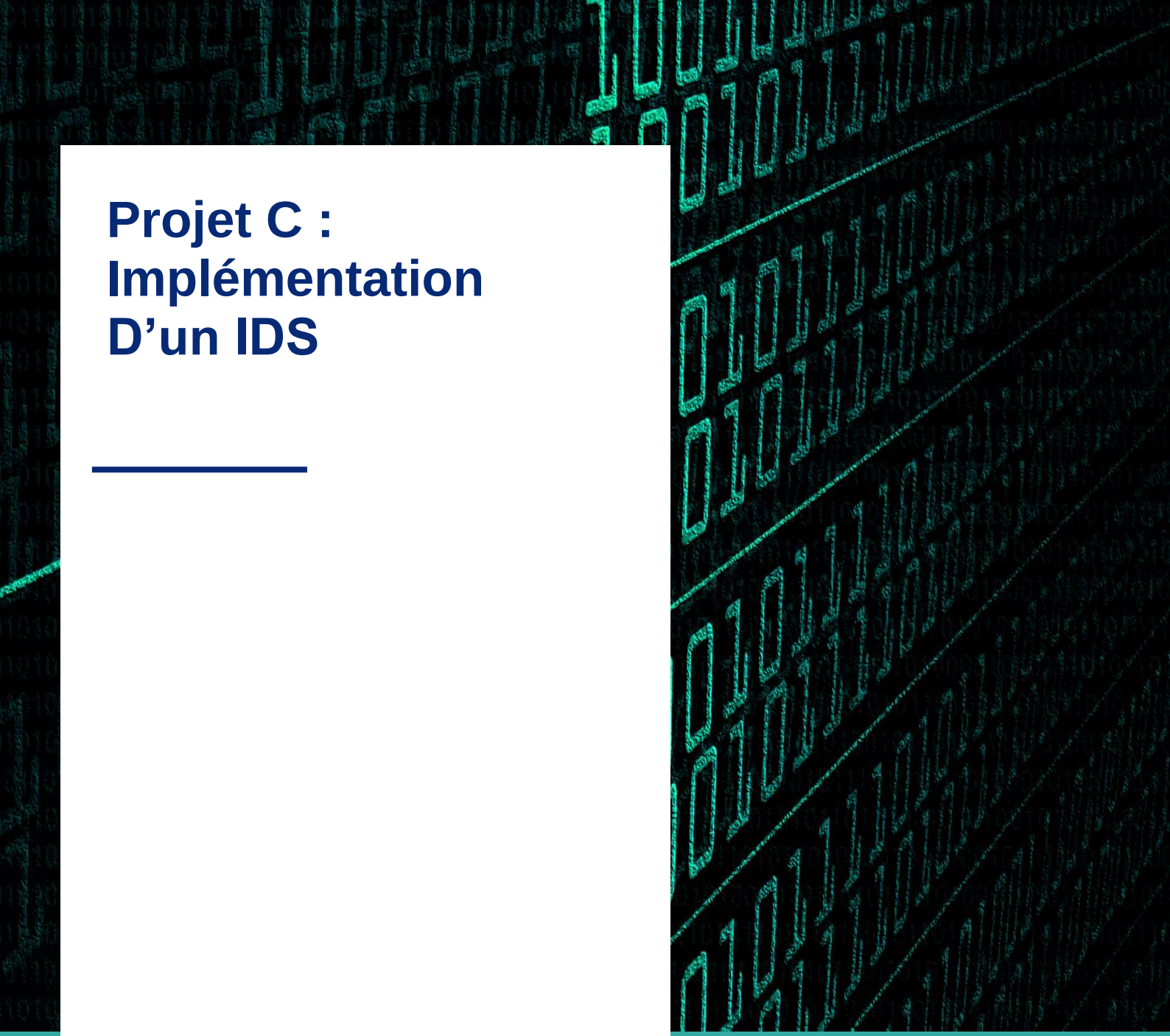


Projet C : Implémentation D'un IDS

BAC2 2020-2021

DEPREZ Martin
BOUTERFASS Hicham

IR209

Table des matières

1. Introduction

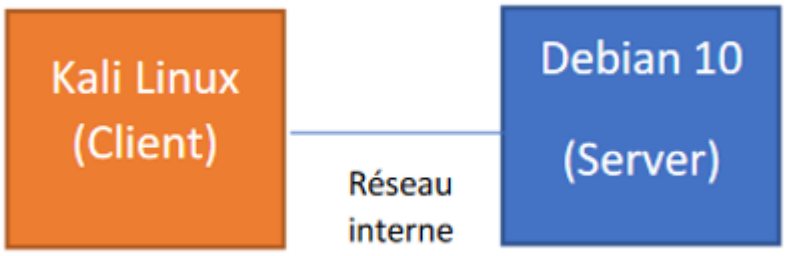
Dans le cadre du cours de Développement, nous avons été amenés à implémenter un IDS *(Intrusion Detection System)* en C. Ce système nous permettra de détecter des activités suspectes en écoutant le réseau, déclenchant ainsi une alerte en cas de menace où d'utilisation de protocoles non sécurisés.

Pour cela, nous avons décidé d'écouter le trafic réseau sur notre machine avec un outil permettant de capturer les paquets transitant sur le réseau.
Le programme sera créé et tournera en tâche de fond sur une VM Kali Linux, les deux VM seront configurés en réseau interne afin d'éviter tout trafic extérieur. L'objectif sera de générer divers types de trafic sur notre serveur Debian afin de le capturer.

Architecture utilisée :
Un client sous Kali avec le programme et un serveur Debian qui va nous permettre de générer du trafic.

Trafics générés :

- UDP
- TCP
- HTTP
- ICMP
- FTP



[1]

[1] **https://moodle.henallux.be/pluginfile.php/314793/mod_resource/content/1/Test%20du%20programme.pdf**

2. Analyse des différentes fonctions

a. fonction main :

```
Rule *rules_ds = NULL;
int count = 0;
```

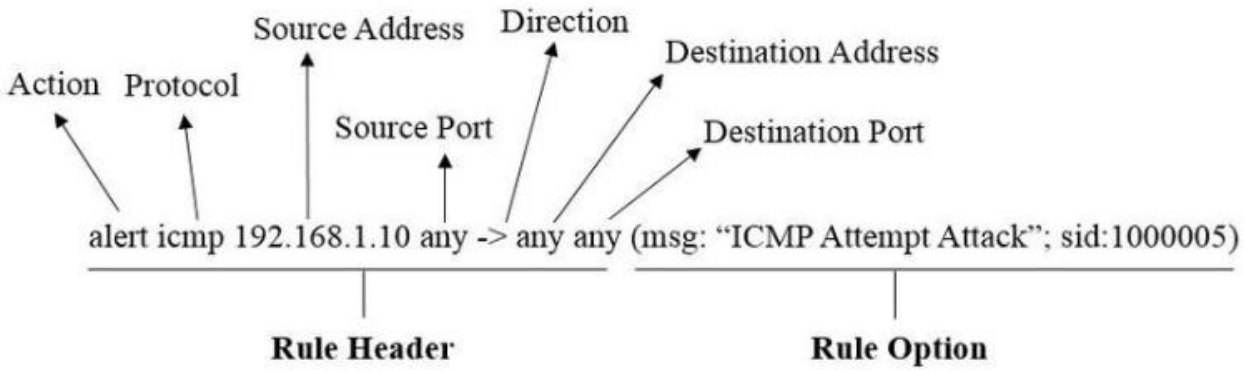
Rule *rules_ds = déclaration + initialisation de notre structure de règles
Int count = 0 = déclaration + initialisation contient le nombre de lignes de notre fichier

Utilisation de 2 variables globales afin d'éviter la modification de nos prototypes de fonctions, à utiliser avec précautions car créent des liens invisibles entres les fonctions.

```
typedef struct ids_rule
{
        char type[10];
        char protocol[10];
        char source_IP[IP_ADDR_LEN_STR];
        char port_source[10];
        char direction[10];
        char destination_IP[IP_ADDR_LEN_STR];
        char port_destination[10];
        Option ids_option;
}Rule;
```

```
typedef struct ids_option
{
   char *key;
   char *values;
} Option;
```

Cette structure va contenir toutes nos règles qui se trouvent dans le IDS.rules
Structure d'une règle :



2

[2] https://moodle.henallux.be/pluginfile.php/307569/mod_resource/content/3/Enonce_projet_IR209_2020_v.1.1.pdf

Pour compiler le programme il faut utiliser la commande :

```
gcc -Wall main.c populate.c rules_func.c -o ids -lpcap
```

Pour lancer le programme :

```
./ids eth0 ids.rules &
```

Le nom de l'interface et le nom du fichier doivent être dans le bon ordre.

```
int main(int argc, char *argv[])
{
        FILE *file = fopen(argv[2],"r");
        if (file == NULL)
            exit(EXIT_FAILURE); // == exit(1)
```

Pour déclarer un fichier on va déclarer un pointeur vers un flux de type FILE :
`FILE *fic = fopen(const char *filename, const char *mode );`
Cette fonction renvoie un pointeur sur le fichier ouvert.
Ce dernier prend comme paramètre le nom du fichier ainsi que le mode ouverture.

Principaux modes d'ouverture :

- *« r »* lecture seule, le fichier doit exister
- *« w »* écriture seule
- *« a »* ajout fin de fichier

Si on n'arrive pas à l'ouvrir, le fichier pointera vers NULL et nous quitterons donc le programme.

Note : une fois que nous n'avons plus besoin du fichier, il faudra donc fermer celui-ci afin d'éviter une fuite mémoire.
➔**`fclose(file);`**

b. count_lines_from_file() :

```
count = count_lines_from_file(file);
```

Ici, nous appelons une fonction qui va compter le nombre de lignes présentes dans notre fichier de règles.

```
int count_lines_from_file(FILE *fic);
```

Cette fonction prend comme paramètre le fichier de règles et renvoie un entier,
Celui-ci sera égal au nombre de lignes comptées dans le fichier **fic*.

```
int count_lines_from_file(FILE *fic)
{
    int c = 0;
    int lines = 0;
    while(!feof(fic))
    {
        c = fgetc(fic);
        if(c == '\n')
            lines ++;
    }
    rewind(fic); // remetre le curseur au debut du fichier
    return lines;
}
```

Nous allons parcourir le fichier tant que nous ne sommes pas à la fin → *while(!feof(fic)).*
Cette fonction va déterminer s'il reste des octets à lire dans le flux.
Nous appelons ensuite la fonction « *fgetc* » qui va lire un caractère, si celui-ci est égal à *'\n'* , alors on incrémente notre variable *lines*.
Quant à la fonction *rewind*(), celle-ci va remettre le curseur en début de fichier.
Pour finir, on retourne le nombre de lignes présentent dans le fichier.

```
rules_ds = malloc(count * sizeof(*rules_ds));
```

A l'adresse de *rules_ds,* nous allons créer un tableau de structure de manière dynamique à l'aide de la fonction *malloc*.
Count = le nombre de règles que nous avons calculées avant * la taille de notre structure *rules_ds.*

c. read_rules() :

```
read_rules(file, rules_ds, count);
```

Cette fonction va lire les règles et les placer dans notre tableau de structure *rules_ds.*

```
void read_rules(FILE * file, Rule *rules_ds, int count);
```

Elle prend comme paramètre le fichier, la structure de règle, et le nombre de ligne du fichier.

```
char texte[256];
for (int i = 0; i < count; i++)
{
    fgets(texte,256,file);
```

Dans un premier temps, nous allons parcourir le fichier, tant qu'on est plus petit que *count* on va récupérer la première ligne de notre fichier (la première règle) avec la fonction *« fgets() »*

Cette fonction va prendre la ligne et la placer dans la variable texte avec une taille de 256 caractères.

Si la chaine lue est plus longue que le buffer de réception, aucun dépassement en mémoire ne sera fait.

```
//séparer la chaine en 2 header + option
char find = '(';
const char *ptr = strchr(texte,find);
int index = ptr - texte; //trouver index pour separer header et option
char rules_header[100]; // alert http any any -> any any
char rules_options[100]; //(msg:"shell attack"; content:"malware.exe";)
```

`char *strchr(const char *string, int searchedChar );`

Cette fonction va rechercher la première occurrence du caractère passé en second paramètre, si ce dernier match, elle retournera un pointeur vers ladite occurrence, si pas, *NULL* sera retourné.

NB : Ne pas confondre ***strchr*** avec ***strrchr***, la seconde va retourner la dernière occurrence, la fonction effectuant une recherche dans la chaine en allant de la droite vers la gauche.

```
memset(rules_header,'\0',sizeof(rules_header));
strncpy(rules_header,texte,index); //strncpy(<destination>,<source>,<n>)

memset(rules_options,'\0',sizeof(rules_header));
strncpy(rules_options,texte + index,strlen(texte));
```

La fonction *memset* va remplir la zone mémoire avec des valeurs. Pourquoi avoir utiliser cette fonction ? Car lors de nos tests et de l'affichage du header, nous avions remarqué qu'il y avait des résidus dans la mémoire.

```
rules_ds[i].ids_option.key = (char*)calloc(100,sizeof(char));
rules_ds[i].ids_option.values = (char*)calloc(100,sizeof(char));
```

Ici, nous parcourons notre tableau de structure et allouons de manière dynamique les *keys* et les *valeurs* avec la fonction *calloc .*

```
//alert http any any -> any any
sscanf(rules_header,"%s %s %s %s %s %s %s",rules_ds[i].type, rules_ds[i].protocol, rules_ds[i].source_IP,
    rules_ds[i].port_source, rules_ds[i].direction, rules_ds[i].destination_IP, rules_ds[i].port_destination);
```

Pour le *header*, nous n'avons pas besoin de faire de modification au niveau de la chaine parce que nous pouvons récupérer toutes les valeurs.

L'utilisation de *sscanf()* permet d'extraire les données à partir d'une chaine de caractère, ici, ce sera le *header*. Ensuite, un formatage s'applique sous forme de *« %s »* pour une chaine, *« %d »* pour un entier, etc…
Nous pourrons ainsi le placer dans notre structure d'indice ***i***, car à chaque itération de ***i***, une nouvelle règle sera stockée dans notre structure.

NB : Attention à l'ordre des éléments ! Par exemple :

« alert » va être placé dans *rules_ds[i].type*

Le dernier *« any »* lui dans *rules_ds[i].port_destination*

```
//(msg:"shell attack"; content:"malware.exe";)
const char *delimiter = ("(");";

char *token = strtok(rules_options,delimiter);
strcpy(rules_ds[i].ids_option.key,token);

token = strtok(NULL,delimiter);
strcpy(rules_ds[i].ids_option.values,token);
```

Ensuite, il nous reste à traiter les options. Un problème s'oppose à nous : les parenthèses.

Pour remédier à cela, nous allons récupérer dans la *key* : ***« msg : shell attack »*** et dans le *values* : ***« content : malware.exe »***.

Nous allons alors diviser la chaine avec la fonction ***strtok()*** qui prend comme paramètre la chaine en question et le délimiteur.

```
char *strtok(char *string, const char *delimiters );
```

Pour le premier *token*, on commence par le premier élément.

Pour le deuxième, on demande l'élément suivant avec le *NULL*.

IMPORTANT : la chaine d'origine est altérée avec l'utilisation de cette fonction.

Une autre possibilité série d'utiliser propres fonctions.

Mais ici dans la librairie ***<string.h>*** nous avons tout ce qu'il nous faut pour pouvoir traiter les chaines de caractères.

Pour le premier token, nous allons copier la valeur de celui-ci dans notre structure de règle *structure0[i].structure1.key.*

Pareil pour le second mais il se trouvera dans les *values.*

ANALOGIE à un dictionnaire en python :

Key : values

d. check_struct() :

```
if(check_struct(rules_ds,count))
{
    generate_syslog("An error has occured check ids.rules !!","--IDS CRIT--");
    exit(EXIT_FAILURE);
}
fclose(file);
```

Cette fonction va exécuter plusieurs choses, nous allons regarder si notre structure est correcte et qu'il n'y a pas de dépassement.

```
Bool check_struct(Rule *r1, int count);
```

Ici, nous allons prendre comme paramètre la structure à traiter et le nombre de lignes. Cela nous renverra un ***Bool*** qui a été définit avec un *enum* dans le *populate.h*.

```
Bool check_struct(Rule *r1, int count)
{
    //Objectif de la fonction check si la structure est correct
    for (int i = 0; i < count; i ++)
    {
        if ((strlen(r1[i].type) > 10)                          ||
            (strlen(r1[i].protocol) > 10)                      ||
            (strlen(r1[i].source_IP) > IP_ADDR_LEN_STR)        ||
            (strlen(r1[i].port_source) > 10)                   ||
            (strlen(r1[i].direction) > 10)                     ||
            (strlen(r1[i].destination_IP) > IP_ADDR_LEN_STR)   ||
            (strlen(r1[i].port_destination) > 10))
                return true;
    }
    return false;
}
```

Si notre condition *match*, la fonction retournera **true,** sinon, elle retournera **false.**

Dans le cas d'un return **true,** nous appellerons une fonction qui va générer un ***syslog*** avec un message d'alerte.

Cette fonction est utilisée plusieurs fois par le programme, notamment pour l'ouverture et la fermeture de celui. Nous l'utiliserons aussi pour générer des messages d'alerte en fonction des paquets que nous allons capturer, ou bien, en cas d'erreur avec le fichier de règles.

NB : Une amélioration possible serait d'utiliser cette fonction pour capturer les signaux du système.

Un exemple avec la commande :

```
kill -9 process
```

- La commande va générer un ***syslog*** pour prévenir que le programme s'est fait tuer.

e. generate_syslog() :

```
void generate_syslog(char *log_msg, char *log_type);
```

Le premier paramètre sera le message à mettre dans les logs.

Le deuxième paramètre sera le nom, par exemple : *« IDS Alert ».*

```
void generate_syslog(char *log_msg, char *log_type)
{
    openlog(log_type,LOG_PID,LOG_USER);
    syslog(LOG_INFO,log_msg);
}
```

Sur notre machine Kali, les logs se trouvent dans */var/log/syslog.*
Attention : cela peut changer en fonction des distributions.

Format des logs :

```
Dec 26 11:39:22 kali ├────┤ IDS ├────┤[4525]: Succesful Program Launch
Dec 26 11:39:25 kali --- IDS ALERT ---[4525]: msg:"backdoor attack"
Dec 26 11:39:30 kali --- IDS ALERT ---[4525]: msg:"UDP traffic bind port is forbidden"
Dec 26 11:39:33 kali --- IDS ALERT ---[4525]: msg:"shell attack"
Dec 26 11:39:44 kali --- IDS ALERT ---[4525]: msg:"Unsecure protocol use detected"
Dec 26 11:39:44 kali ├────┤ IDS ├────┤[4525]: Succesful Program Close
```

Openlog ➔ Ouvre et/ou réouvre la connexion avec le *syslog* afin de préparer l'envoi d'un message.

LOG_PID ➔ Insère le PID du process dans les logs, par exemple : *[4525]*.

LOG_USER ➔ Par défaut, utilisateur générique.

LOG_INFO ➔ Simple information.

A la fin du programme, il ne faut pas oublier de fermer la connexion avec *syslog* en utilisant la fonction ***closelog()***.

Une fois toutes les vérifications effectuées, nous pourrons fermer le fichier et commencer à utiliser la librairie *libpcap*.

f. main préparation du sniffer

```
char *device = argv[1];
char error_buffer[PCAP_ERRBUF_SIZE];
pcap_t *handle;

handle = pcap_create(device,error_buffer);
pcap_set_timeout(handle,10);
pcap_activate(handle);
int total_packet_count = 20;

generate_syslog("Succesful Program Launch",LOG_PROG);

pcap_loop(handle, total_packet_count, my_packet_handler, NULL);
free_memory(rules_ds, count);

generate_syslog("Succesful Program Close",LOG_PROG);
closelog();
return 0;
```

Char *device : nom de l'interface sur laquelle nous allons écouter, celle-ci sera renseignée comme argument à notre programme sous *« **eth0** »*.

`pcap_t *pcap_create(const char *source, char *errbuf)`

Cette fonction va créer un outil de capture et de manipulation des paquets sur le réseau**,** *source prendra comme paramètre l'interface à écouter.
Pour capturer tous les paquets de toutes les interfaces, il suffit de mettre l'argument *« **any** »* ou ***NULL.***

`int pcap_loop(pcap_t *p, int cnt,pcap_handler callback, u_char *user);`

Nous aurons une boucle qui va utiliser une fonction de rappel.
Pour chaque itération, nous récupérons un paquet et nous appelons une fonction de *callback* dans le code. Ici, le nombre de paquets sera 20.

g. my_packet_handler() :

```
void my_packet_handler(u_char *args,const struct pcap_pkthdr *header,const u_char *packet)
{
    ETHER_Frame custom_frame;
    populate_packet_ds(header,packet,&custom_frame);
    rule_matcher(rules_ds,&custom_frame,count);
}
```

Cette fonction prend comme paramètres :

- ***args** ➔ Pointeur vers les arguments de l'utilisateur.
- ***header** ➔ Pointeur vers le header du paquet reçu.
- ***packet** ➔Ce pointeur va pointer vers le premier bit de donnée contenue dans le paquet sniffé par ***pcap_loop().***

Nous allons créer un ***custom_frame*** pour pouvoir facilement manipuler les paquets que nous allons capturer.

ETHER_Frame est une structure qui contient toutes les informations dont nous avons besoin pour les soumettre à notre structure de règles.

Ensuite, nous appellerons la fonction ***populate_packet()*** qui va remplir notre *custom frame*.

Enfin pour chaque paquet capturé, nous ferons appel à la fonction ***rule_matcher()*** qui regardera si le paquet *match* avec les règles que nous avons stockées dans notre structure.

Si une règle match nous appelons la fonction ***generate_syslog()***.

h. populate_packet_ds() :

```
int populate_packet_ds(const struct pcap_pkthdr *header, const u_char *packet, ETHER_Frame *custom_frame)
{
```

Cette fonction prend comme paramètres :

- ***packet** ➔ Pointeur vers le début de la zone mémoire.
- ***custom_frame** ➔ Pointeur vers la structure que nous allons compléter.
- ***header** ➔ Pointeur vers le header paquet reçu.

```
const struct sniff_ethernet *ethernet; /* The ethernet header */
const struct sniff_ip *ip; /* The IP header */
const struct sniff_tcp *tcp; /* The TCP header */
const struct sniff_udp *udp;/* THe UDP Header */
unsigned char *payload; /* Packet payload */


ethernet = (struct sniff_ethernet*)(packet);
```

Nous allons caster le paquet avec la structure ***ethernet,*** nous recevrons ensuite une variable ***ethernet*** qui est du type de cette structure.

```
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
```

Nous devons à nouveau caster le paquet, mais avec la taille de l'header ***ethernet***.
Nous pourrons ainsi garnir notre structure ***ip.***

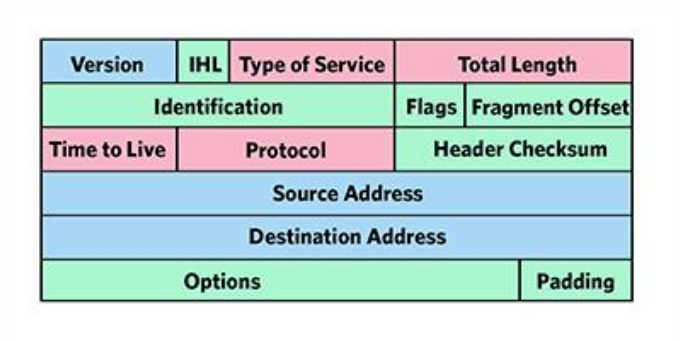
```
udp = (struct sniff_udp*)(packet + SIZE_ETHERNET + size_ip);


tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
```

De la même manière que vu plus haut, nous complèterons notre structure ***udp*** et ***tcp***.

Packet ➔ Contient le pointeur initial, la taille du header, la taille header IP.
Ces données vont nous positionner au début de la partie qui est dédié à TCP ou UDP.

Comment savoir si on utilise de l'UDP ou bien du TCP ?



Dans l'entête IP, le champ protocole, qui est codé sur 8 bits, va indiquer quel type de protocole se trouve derrière l'entête IP.

Dans notre programme, nous avons :

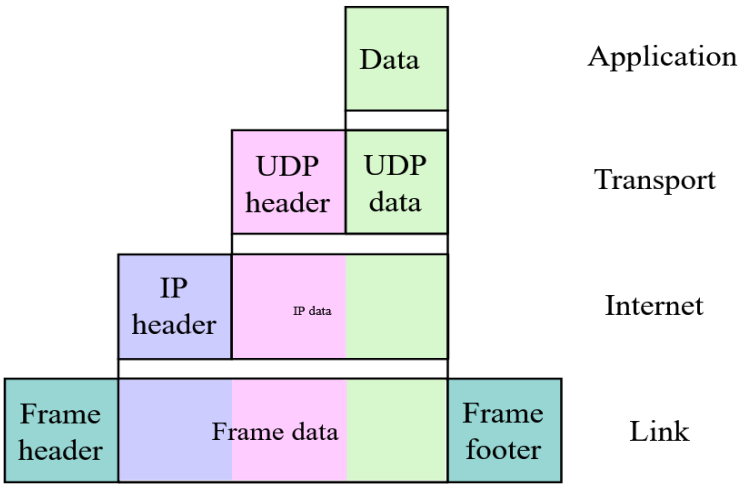
```
#define UDP_PROTOCOL 17
#define TCP_PROTOCOL 6
#define ICMP_PROTOCOL 1
```

Pour plus d'informations, la liste des protocoles que l'on peut retrouver dans l'entête IP se trouve ici.

```
payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
```

Pour finir, nous pouvons récupérer le ***payload***. Ce dernier va contenir les ***data*** avec le pointeur du paquet, le header ***ethernet***, le header ***ip*** et le header ***tcp***.

Toutes ces données récupérées seront stockées dans notre *custom frame.*
Nous comparerons ensuite cette frame à notre fichier de règles.



i. rule_matcher() :

```
void rule_matcher(Rule *rules_ds, ETHER_Frame *frame, int count)
```

Paramètres de la fonction :

- ***rules_ds*** ➔ Pointeur vers la structure qui contient toutes nos règles.
- ***frame*** ➔ Pointeur vers la structure custom_frame qui a été garnie avec le populate_packet.
- ***count*** ➔ Nombre de lignes dans le fichier de règles.

Explication de la structure ETHER_Frame :

```
/*-------------------------------------------------------
struct custom_ethernet
{
        char source_mac[ETHER_ADDR_LEN_STR];
        char destination_mac[ETHER_ADDR_LEN_STR];
        int ethernet_type;
        int frame_size;
        IP_Packet data;

} typedef ETHER_Frame;
/*-------------------------------------------------------
```

custom_ethernet correspond à l'header ***ethernet.*** Ce dernier contient une structure ***IP_Packet*** mais aussi d'autres informations comme la ***source*** et la ***destination mac***, l'***ethernet_type****(IPV4, IPV6,..).* Pour finir, elle permet aussi d'accéder à l'entête IP via ***struct.data.***

```
/*-------------------------------------------------
struct custom_ip
{
        char source_ip[IP_ADDR_LEN_STR];
        char destination_ip[IP_ADDR_LEN_STR];
        int ip_protocol;
        TCP_Segment data;
        UDP_segment data_UDP;

} typedef IP_Packet;
/*-------------------------------------------------
```

C'est dans cette entête que nous pourrons récupérer les adresses IP, mais aussi , en fonction de la valeur de ***ip_protocol***, nous allons pouvoir déterminer ci c'est de l'***UDP*** ou du ***TCP.***

Pour accéder aux informations UDP ➔ ***strutc.data.data_UDP***

Pour accéder aux information TCP ➔ ***struct.data.data***

```
for (int i = 0 ; i < count ; i ++)
```

La première chose à faire sera de parcourir notre structure de règles.

Ensuite, nous avons décidé d'utiliser un *switch* pour des raisons de facilité mais aussi de propreté, nous avons stocké dans l'entête IP le protocole de couche inférieur qui sera utilisé.

```
switch(frame->data.ip_protocol) //TCP IMCP UDP
{
    case 6: // TCP_PROTOCOL
        if(frame->data.data.destination_port == atoi(rules_ds[i].port_destination)) //TCP msg
            generate_syslog(rules_ds[i].ids_option.key,LOG_ALERTE);
        if(search_web_content((char*)rules_ds[i].ids_option.values,(char*)frame->data.data.data))
            generate_syslog(rules_ds[i].ids_option.key,LOG_ALERTE);
        break;
```

Si c'est du ***TCP***, il y aura plusieurs choses à regarder. Nous devons d'abord comparer le numéro de port qui se trouve dans le *header* ***TCP*** avec le numéro de port qui se trouve dans le fichier de règles. Si les deux numéros *match,* un ***syslog*** sera généré.

Un autre cas de figure pourrait se présenter lors de la navigation sur une page web, si on retrouve le contenu malware.exe on génère un ***syslog***.
➔ Pour cela, nous avons créé la fonction ***search_web_content***.

```
/*recherche dans les data la values passé
en paramètre*/
if(strlen(values) > 5)
{
    char find[100];
    strcpy(find,values);

    const char *delimiter = ("\": ");
    char *token = strtok(find,delimiter); //content
    token = strtok(NULL,delimiter);  // malware.exe
    if (strstr(data,token) != NULL)
        return true;
}
return false;
```

`const char *strstr(const char *fullString, const char *substring );`

Nous allons regarder si une chaine se trouve dans une autre valeur de retour.

Si ***NULL*** == notre valeur, il n'y a pas de match.

Si notre valeur != ***NULL***, la sous-chaîne est trouvée et renvoie un pointeur visant la première occurrence.

La fonction va nous retourner ***true*** ou ***false*** si elle retourne ***true***, un ***syslog*** sera généré, contenant par exemple : *« msg : shell attack ».*

```
case 17: //UDP_PROTOCOL
    if(frame->data.data_UDP.destination_port == atoi(rules_ds[i].port_destination))
        generate_syslog(rules_ds[i].ids_option.key,LOG_ALERTE);
    break;
```

Ici, nous utilisons la même méthode que pour le ***TCP***, si le numéro de port match et que nous avons de l'UDP, un ***syslog*** sera généré.

```
case 1: //ICMP_PROTOCOL
    if(strcmp(frame->data.source_ip,rules_ds[i].source_IP) == 0)
        generate_syslog(rules_ds[i].ids_option.key,LOG_ALERTE);
    break;
```

Ici, nous avons blacklisté une adresse IP en particulier, si celle-ci tente de ping la machine, un ***syslog*** sera généré.

NB : Une amélioration possible serait qu'à chaque ping reçu, un ***syslog*** serait généré. La solution pour cela serait de compter le nombre de paquet, et qu'au bout de X paquets, générer un ***syslog***.

```
if(frame->data.data.destination_port == 21)
    {
        char *protocol = "ftp";
        if(strcmp(rules_ds[i].protocol,protocol) == 0)
            generate_syslog(rules_ds[i].ids_option.key,LOG_ALERTE);
    }
```

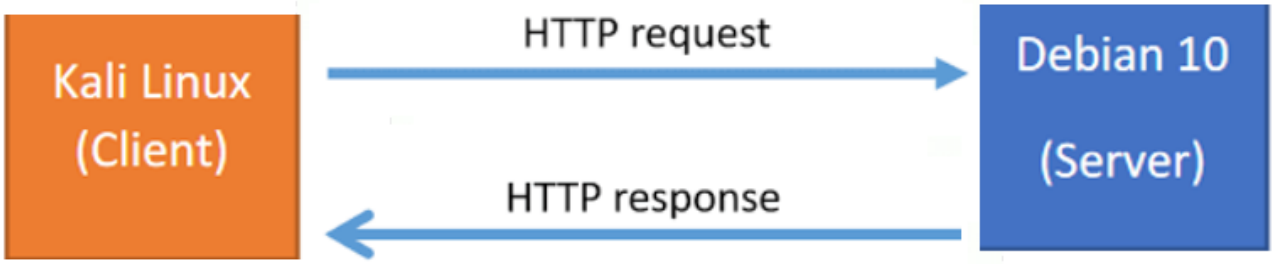
Ici, nous savons que le ***FTP*** fonctionne sur le port 21. Si nous retrouvons ce dit numéro, nous créerons une variable qui va contenir « ***ftp*** » et la comparer à notre fichier de règles. S'il y a *match,* un ***syslog*** sera généré.

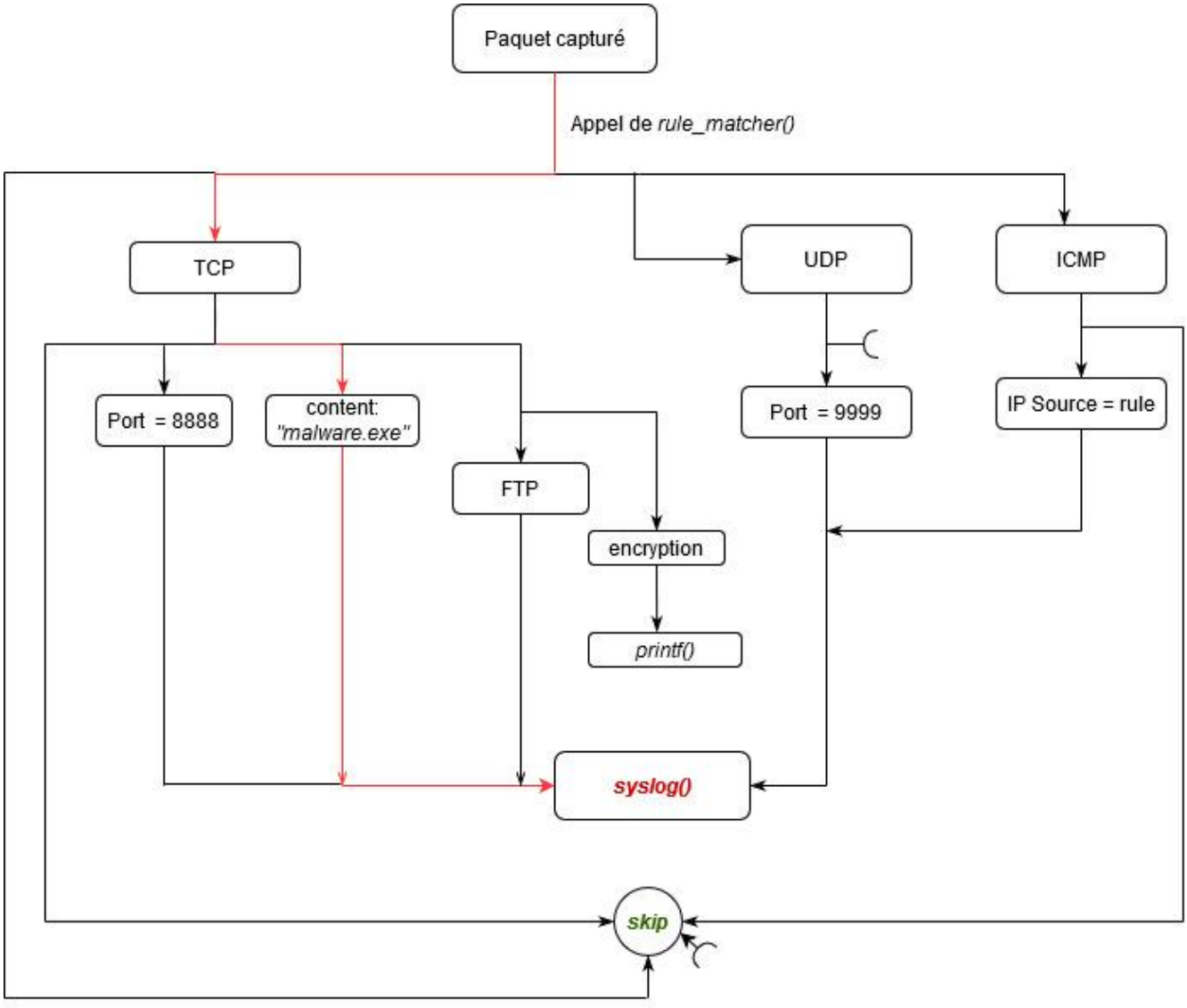
```
if(frame->data.data.destination_port == 443 && strstr((char*)frame->data.data.data,"GET ") == NULL)
    printf("ENCRYPTION DETECTED\n");
```

Cette partie permet de détecter l'encryption du payload. Dans un premier temps, nous regardons le numéro de port HTTPS, égal à 443. Si ce dernier *match*, et que nous n'arrivons pas à trouver « ***GET*** » dans le *payload* avec la fonction ***strstr***, nous aurons donc de l'HTTPS.

3. Fonctionnement du programme

Nous avons décidé de schématiser notre fonction *rule_matcher()* dans laquelle nous pourrions observer le chemin parcouru par un paquet HTTP contenant « malware.exe ».





Voici la liste des différentes règles utilisées :

```
alert http any any -> any any (msg:"shell attack"; content:"malware.exe";)
alert tcp any any -> any 8888 (msg:"backdoor attack";)
alert udp any any -> any 9999 (msg:"UDP traffic bind port is forbidden";)
alert ftp any any -> any any (msg:"Unsecure protocol use detected";)
alert icmp 192.168.10.30 any -> any any (msg:"ICMP Attempt Attack";)
```

4. Conclusion

La réalisation de ce travail qui consistait à implémenter un IDS en C nous a permis d'établir un lien entre la programmation et la sécurité informatique.

De plus, les différentes recherches effectuées, que ce soit aussi bien au niveau du réseau qu'au niveau de la programmation, nous ont permis d'accroître nos compétences dans ces domaines.

Plusieurs constats ont été fait durant la réalisation de ce travail, et certaines améliorations nous semblent envisageables, notamment le traitement d'autres protocoles, la possibilité de gérer les attaques de type *sync flood*, ou encore le développement d'une interface graphique. Nous pourrions également bloquer les IP dans un fichier *blocklist* en cas d'attaque extérieure.

5. Bibliographie

https://koor.fr/C/cstdio/fgets.wp
https://www.tcpdump.org/manpages/pcap_create.3pcap.html
https://koor.fr/C/cstdio/feof.wp
https://moodle.henallux.be/pluginfile.php/314793/mod_resource/content/1/Test%20du%20programme.pdf
https://www.ltam.lu/cours-c/prg-c112.htm
https://www.gnu.org/software/libc/manual/html_node/openlog.html
https://stackoverflow.com/questions/8175827/what-happens-if-i-dont-call-fclose-in-a-c-program
https://www.frameip.com/entete-ip/
https://en.wikipedia.org/wiki/Encapsulation_(networking)
https://koor.fr/C/cstring/strstr.wp
https://www.youtube.com/watch?v=90hGCMC3Chc&list=PLrSOXFDHBtfEh6PCE39HERGgbbaIHhy4j