Introduction à la sécurité

Sylvain Conchon

Laboratoire Méthodes Formelles

Université Paris-Saclay, CNRS, ENS Paris-Saclay

sylvain.conchon@universite-paris-saclay.fr

Contenu du cours

- Introduction
- Principales cyberattaques
- SYN Flooding
- XSS crossing
- Buffer overflow

Évaluation du cours

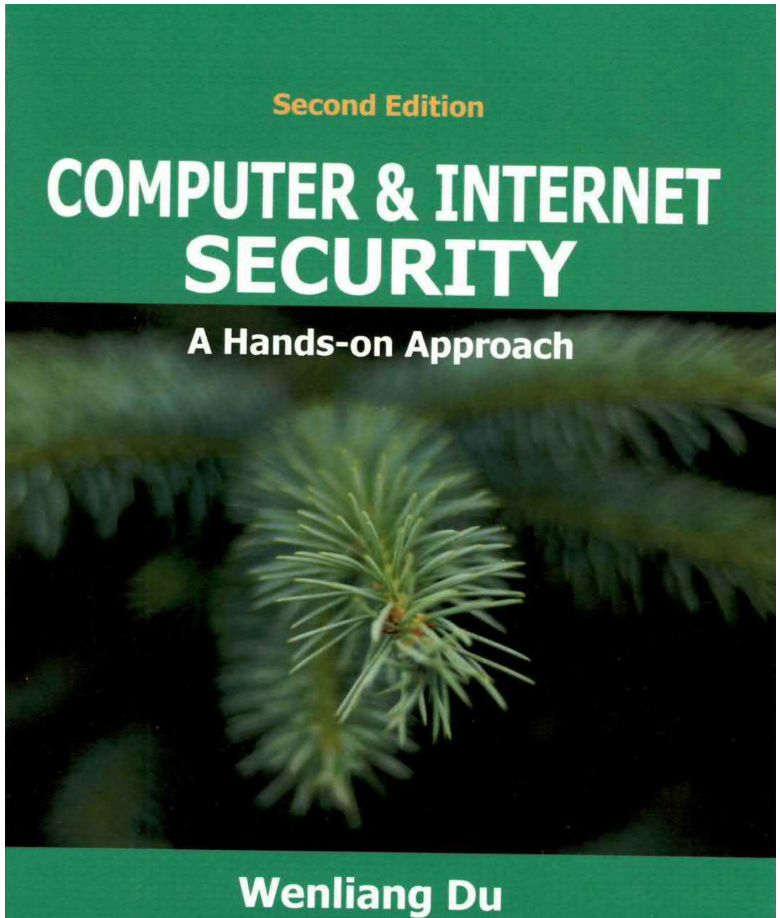
Un exposé sur une attaque en binôme (20 minutes + questions)

Livres et articles supports du cours

Computer security (Third edition), Dieter GOLLMANN, Wiley 2010.

Cryptographie appliquée (Second édition), Bruce SCHNEIER, Wiley 2001

Computer security : a hands-on approach, Wenliang Du, 2022.



INTRODUCTION

La sécurité informatique

La sécurité informatique est un domaine très large qui implique d'avoir des connaissances en cryptographie, systèmes (voire architecture), réseaux, programmation, etc.

Il s'agit essentiellement d'un domaine qui s'intéresse au flot de l'information dans un système informatique.

Politique de sécurité

Une politique de sécurité définit qui est autorisé à accéder à quoi.
Par exemple,

- qui peut lire ou modifier des informations dans une base de données,
- qui peut écouter, envoyer ou modifier des communications sur un réseau ;
- etc.

On pourra ainsi définir les attaquants ou pirates comme ceux qui ne sont pas autorisés à accéder à certaines informations.

Confidentialité

Informations non divulguées aux attaquants

(en anglais) confidentiality, secrecy

Techniques : chiffrement symétrique ou asymétrique (cryptographie)

Notez que la confidentialité ne signifie pas que l'existence des informations privées est gardée secrète, seulement leur contenu.

Intégrité

Si un attaquant altère de l'information, alors les entités autorisées en sont informées

(en anglais) integrity

Technique : hachage cryptographique

Authentification

Un attaquant est incapable de falsifier une identité

Technique : cryptographie asymétrique et tiers de confiance (pour fournir ou vérifier les clés cryptographiques)

Sur un canal de communication, l'authentification a un sens particulier. Un canal est authentifié si chaque message qui y circule :

1. a été envoyé par l'expéditeur prétendu
2. qui avait l'intention de l'envoyer au destinataire, et
3. n'a pas été reçu hors séquence ou en double.

Notez que l'intégrité est implicite dans ce sens.

Non-répudiation

Un acteur ne peut nier avoir effectué une action particulière

On parle de non-répudiation dès qu'il existe une preuve infalsifiable pour

- la rédaction d'un message, ou
- l'envoi d'un message, ou
- la réception d'un message

Technique : signatures numériques (cryptographie asymétrique, fonctions de hachage cryptographiques)

Disponibilité

L'information et les ressources sont accessibles aux parties autorisées

Il s'agit d'une propriété importante : sans elle un système informatique sécurisé est celui qui est éteint.

CYBERATTAQUES

Cyberattaques

Voici un liste non-exhaustive d'attaques informatiques bien connues

- Déni de service (DoS)
- Man in the Middle (MitM)
- Harponnage (Phishing)
- Téléchargement furtif (Drive by download)
- Mot de passe
- Injection SQL
- Cross-site scripting (XSS)
- Écoutes illicites
- Anniversaires
- Logiciels malveillants
- Débordement de buffer (Buffer Overflow)
- ...

DoS (1/2)

Objectif : provoquer la panne (ou suspension de service) du système d'information d'une entreprise

L'idée principale est de submerger de requêtes un serveur afin qu'il ne puisse plus répondre aux demandes de service.

TCP SYN Flood : saturation de demandes de connexion initiale (SYN) pour établir une liaison TCP. Le serveur répond avec un A/R (SYN-ACK) à chaque demande, mais aucun A/R final (ACK) n'est envoyé par le pirate. Le serveur maintient ainsi un très grand nombre de port semi-ouverts, ce qui finit par l'empêcher de fonctionner (plus de ports de disponibles).

Teardrop : exploite des bugs dans le ré-assemblage des paquets TCP/IP fragmentés (mauvais champ d'offset, chevauchement).

DoS (1/2)

Smurf : saturation de paquets Internet Control Message Protocol (echos ICMP) usurpés avec une adresse IP source qui est celle de la victime (via un routeur et une adresse de diffusion) qui se retrouve noyée de réponses.

Ping of death : envois de paquets (fragmentés) de type *ping* dont la taille une fois ré-assemblés est supérieure à la taille maximale de 64ko. Cela peut provoquer des débordements de buffers ou autres plantages.

Botnets : attaque DoS réalisée par des milliers de machines infectées par un logiciel malveillant (DDoS) et coordonnées par un (ou des) pirate(s).

MitM

Un pirate intercepte les communications entre deux parties, en écoutant ou se faisant passer pour un participant légitime.

Détournement de session : le pirate prend le contrôle d'une session de communication TCP entre deux machine. Avec TCP, l'authentification a lieu uniquement à l'ouverture de la session où le serveur envoie un identifiant de session (qui sert d'authentification pour la suite de la session).

Usurpation d'IP : modification de l'adresse source dans les en-têtes de paquets IP.

Relecture : l'attaquant intercepte, enregistre d'anciens messages et les rejoue plus tard.

Harponnage

Attaque par e-mails semblant provenir de sources fiables dont le but est d'obtenir des informations personnelles ou d'effectuer des opérations sensibles.

$\rightarrow$ Ingénierie sociale, stratagème technique

phishing : attaques de masse, non ciblées

Spear phishing : attaques ciblées (recherches menées sur les cibles, messages personnalisés)

Techniques : usurpation adresses électroniques (faiblesse de SMTP dans l'authentification des adresses dans les en-têtes d'e-mails), domaines similaires, clonage de sites web, etc.

Selon le CESIN (club d'experts en sécurité informatique), en 2022, 74% des entreprises déclarent le phishing comme premier vecteur d'entrée pour les attaques subies.

Drive by download

Téléchargement involontaire de logiciels malveillants depuis un serveur Internet (Web, e-mail, etc.).

$\rightarrow$ Exploitation des vulnérabilités des navigateurs, des plug-ins, ou logiciels obsolètes.

Pages web compromises (site web non sécurisés où un script malveillant est installé dans le code HTTP ou PHP) ou malicieuses avec un iframe invisible qui permet d'intégrer une page web au sein d'une autre.

L'iframe charge secrètement la page pirate qui recherche une faille de sécurité sur le système du client pour y installer un malware.

Mot de passe

Attaque pour trouver le mot de passe d'un utilisateur.

Force brute : tester l'une après l'autre, toutes les combinaisons possibles d'un mot de passe.

Attaque par dictionnaire : utilisation d'un dictionnaire des mots de passe les plus utilisés.

Injection SQL

Attaque ciblant les sites Web adossés à des bases de données.

Exécution d'une requête SQL modifiée par l'injection d'un morceau de requête non prévu, provenant la plupart du temps d'un formulaire.

```
"SELECT * FROM users WHERE account =" + Number + ";"
```

Si le numéro du compte utilisateur `Number` renseigné par le formulaire est `'1' or '1'`, alors la requête renvoie les informations de tous les utilisateurs.

XSS

Exploite les failles d'une application Web en injectant un code malveillant via les paramètres d'entrée côté client.

$\rightarrow$ vol de cookies, jetons de session, exécution de malware, etc.

Par exemple,

```
function toto(s){
  return '<script>console.log(" '+s+' ");</script>';
}
```

Cette fonction est attaquable en saisissant la valeur suivante :

```
Bonjour");</script><script>code malveillant</script>//
```

Une application sur deux contiendrait ce genre de vulnérabilité

Écoutes illicites

Interception du trafic réseau

$\rightarrow$ obtention de mots de passe, numéros de cartes bancaires, etc.

Anniversaires

Attaque contre les algorithmes de chiffrement, en particulier les signatures numériques basées sur des fonctions de hachage, qui repose sur le paradoxe des anniversaires pour trouver des collisions.

Paradoxe : quelle est la probabilité que deux étudiants d'une classe de 30 fêtent leur anniversaire le même jour ?

Anniversaires

Attaque contre les algorithmes de chiffrement, en particulier les signatures numériques basées sur des fonctions de hachage, qui repose sur le paradoxe des anniversaires pour trouver des collisions.

Paradoxe : quelle est la probabilité que deux étudiants d'une classe de 30 fêtent leur anniversaire le même jour ?

$$1 - \frac{365 \times 364 \times \cdots \times 336}{365^{30}}$$

```
x = 1
for i in range(336,366): x = x * i
print(1 - x / 365**30)
```

Logiciels malveillants

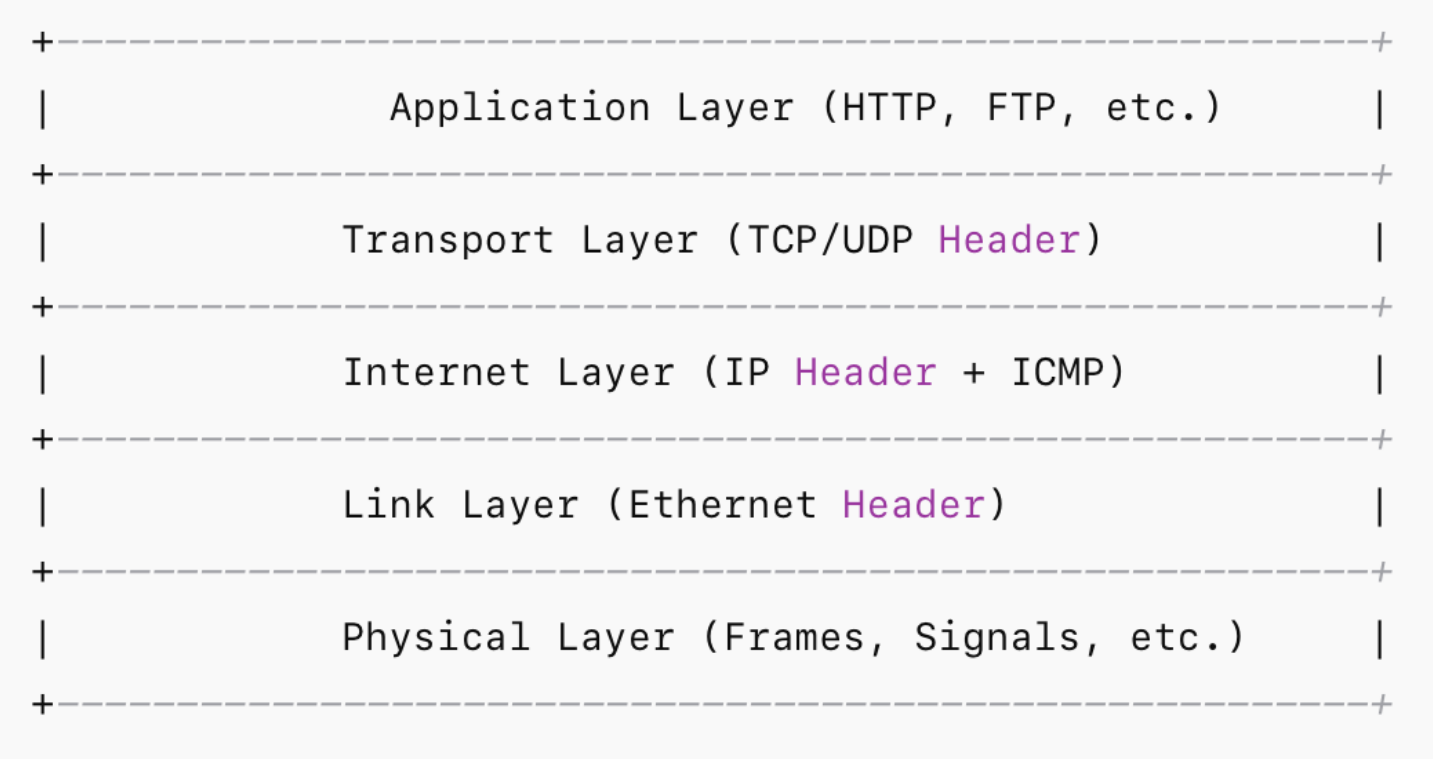
Les logiciels malveillants sont utilisés par les cybercriminels pour :

- inciter les victimes à fournir des données personnelles (vol d'identité, données financières, etc.)
- prendre le contrôle d'ordinateurs afin de lancer des attaques (DoS, minage, rançons, etc.)

Quelques types de logiciels malveillants : virus, ransomware, faux logiciels de sécurité (scareware), ver, logiciels espion (spyware), cheval de troiex

L’ATTAQUE SYN FLOODING

Rappels : couches modèle TCP/IP



TCP/UDP : établissement de la connexion entre les hôtes, la gestion des flux de données (transmission fiable avec TCP ou non fiable avec UDP, ajouts des numéros de port source et destination dans paquets TCP).

IP : ajout des adresses IP pour le routage des paquets

Rappels : les sockets

La couche application est principalement programmée en utilisant des sockets qui permettent la communication (généralement sur un réseau) entre différents processus.

Chaque socket a généralement deux composants importants pour identifier une connexion :

Adresse IP : L'adresse du réseau (source ou destination).

Numéro de port : Le port spécifique de la machine pour la communication

Schéma standard de communication via un socket (TCP) :

1. Le serveur crée un socket et écoute sur un port particulier.
2. Le client crée un socket et se connecte au serveur (adresse IP + port) (pas nécessaire si UDP).
3. Échange de données entre client et serveur.
4. Fermeture de la connexion.

Rappels : ouverture d'un socket

La fonction socket() en C est utilisée pour créer un nouveau socket. Elle permet de spécifier le type de communication (par exemple, TCP ou UDP), ainsi que la famille d'adresses (IPv4 ou IPv6).

```
int socket(int domain, int type, int protocol);
```

- `domain` : La famille d'adresses (par exemple, `AF_INET` pour IPv4, `AF_INET6` pour IPv6).
- `type` : Le type de socket (par exemple, `SOCK_STREAM` pour TCP, `SOCK_DGRAM` pour UDP).
- `protocol` : Le protocole à utiliser (généralement 0 pour laisser le système choisir automatiquement, ou un protocole spécifique comme `IPPROTO_TCP`, `IPPROTO_UDP` pour UDP, `IPPROTO_ICMP` pour ICMP).

Rappels : la structure `sockaddr_in`

La struct `sockaddr_in` est utilisée pour spécifier une adresse **IPv4** dans les sockets. Elle est définie dans `<netinet/in.h>`

Elle contient les champs suivants :

- `sin_family` : Famille d'adresses, généralement `AF_INET` pour IPv4.
- `sin_port` : Le port (en format réseau, converti avec `htons()`).
- `sin_addr` : L'adresse IP, de type `struct in_addr`.
- `sin_zero` : Champ réservé, généralement 0.

La structure `sin_addr` n'a qu'un seul champ `uint32_t s_addr` qui contient une adresse IP en format réseau. On utilise la fonction `inet_addr("...")` pour convertir une chaîne de caractères en adresse IP.

Rappels : la structure `sockaddr`

`struct sockaddr` est une structure générique utiliser pour représenter une adresse de socket.

Elle est utilisée de manière abstraite pour tout type de socket (IPv4, IPv6, etc.) et permet de manipuler des informations d'adresse de manière uniforme.

Elle est généralement utilisée comme type de base pour les autres structures spécifiques aux protocoles (comme `sockaddr_in` pour IPv4 ou `sockaddr_in6` pour IPv6).

```
struct sockaddr {
    unsigned short sa_family;
    char sa_data[14];
};
```

Le champ `sa_data` contient les données spécifiques à la famille de l'adresse.

Rappels : envoi de données sur un socket UDP

La fonction sendto() est utilisée pour envoyer des données via un socket UDP. Il n'est pas nécessaire d'établir une connexion au préalable.

```
ssize_t sendto(int sockfd,
               const void *buf,
               size_t len,
               int flags,
               const struct sockaddr *dest_addr,
               socklen_t addrlen);
```

- `sockfd` : descripteur du socket (créé avec socket())
- `buf` : tampon contenant les données à envoyer
- `len` : taille des données à envoyer
- `flags` : Options de transmission (souvent 0).
- `dest_addr` : adresse de destination (structure sockaddr contenant l'adresse IP et le port)
- `addrlen` : taille de l'adresse de destination.

Rappels : implémentation en C d'un simple client UDP

Voici une implémentation en langage C d'un simple client UDP qui envoie une chaine de caractères à un serveur*.

```c
#include <string.h>
#include <arpa/inet.h>

int main(){
  struct sockaddr_in dest = {0};
  const char *data = "UDP message\n";
  int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

  dest.sin_family = AF_INET;
  dest.sin_addr.s_addr = inet_addr("172.18.88.214"); // ifconfig
  dest.sin_port = htons (9090);

  sendto(sock,data,strlen(data),0,
         (struct sockaddr *) &dest, sizeof(dest));
  return 0;
}
```

Le serveur peut être lancé à l'aide de la commande `nc -luc 9090` (NetCat) qui écoute les paquets UDP sur le port 9090

* Utilisez la commande `ifconfig` pour connaitre votre adresse IP

Rappels : liaison d'un socket

La fonction `bind()` est utilisée pour associer un socket à une adresse et un port spécifiques sur la machine locale. Cela est nécessaire pour que le système sache sur quelle interface réseau et quel port écouter ou envoyer des données.

```
int bind(int sockfd,
         const struct sockaddr *addr,
         socklen_t addrlen);
```

- `sockfd` : descripteur de fichier du socket
- `addr` : pointeur vers une structure sockaddr qui contient l'adresse IP et le port auxquels le socket sera lié
- `addrlen` : taille de la structure sockaddr.

UDP : permet d'écouter les messages entrants sur un port spécifique

TCP : à faire avant `listen()` pour accepter des connexions entrantes

Rappels : réception de données sur un socket UDP

La fonction recvfrom() permet de recevoir des données à partir d'un socket UDP. Elle permet également de récupérer l'adresse de l'expéditeur du message.

```
ssize_t recvfrom(int sockfd,
                 void *buf,
                 size_t len,
                 int flags,
                 struct sockaddr *src_addr,
                 socklen_t *addrlen);
```

- `sockfd` : descripteur du socket
- `buf` : tampon où les données reçues seront stockées
- `len` : taille maximale du tampon
- `flags` : mettre à 0
- `src_addr` : pour stocker l'adresse de l'expéditeur
- `addrlen` : pointeur vers la taille de `src_addr`

Rappels : implémentation en C d'un simple serveur UDP

Voici une implémentation en langage C d'un simple serveur UDP qui affiche une chaine de caractères envoyée par un client.

```c
#include <stdio.h>
#include <unistd.h>
#include <arpa/inet.h>

int main() {
    int sockfd;
    struct sockaddr_in client_addr,server_addr = {0};
    char buffer[1024];
    socklen_t addr_len = sizeof(client_addr);

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr("172.18.88.214");
    server_addr.sin_port = htons(9090);

    bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr));
    ssize_t len = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0,
                           (struct sockaddr *)&client_addr, &addr_len);
    printf("%s\n", buffer);
    close(sockfd);
    return 0;
}
```

Rappels : connexion TCP

Avant d'envoyer des données sur un socket TCP, il faut se connecter à une autre machine.

La fonction `connect()` est utilisée pour établir une connexion réseau entre un socket de type TCP (`SOCK_STREAM`) et un serveur a une certaine adresse.

```
int connect(int sockfd,
            const struct sockaddr *addr,
            socklen_t addrlen);
```

- `sockfd` : descripteur de socket
- `addr` : Pointeur vers une structure sockaddr qui contient l'adresse (et éventuellement le port) du serveur auquel se connecter.
- `addrlen` : taille de la structure sockaddr.

Rappels : envoi de données sur un socket TCP

La fonction send() en C est utilisée pour envoyer des données sur un socket SOCK_STREAM de manière fiable.

```
ssize_t send(int sockfd,
             const void *buf,
             size_t len,
             int flags);
```

- `sockfd` : descripteur du socket (créé avec `socket()` ou `accept()`)
- `buf` : tampon contenant les données à envoyer
- `len` : taille des données à envoyer
- `flags` : Options de transmission (souvent 0).

Rappels : implémentation en C d'un simple client TCP

```c
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(){

  int sockfd = socket(AF_INET, SOCK_STREAM, 0);
  struct sockaddr_in dest = {0};
  char buf[100];

  dest.sin_family = AF_INET;
  dest.sin_addr.s_addr = inet_addr("172.18.88.214"); // ifconfig
  dest.sin_port=htons(9090);

  connect(sockfd, (struct sockaddr *) &dest, sizeof(dest));

  while (1) {
    fgets(buf, sizeof(buf), stdin);
    if (strcmp(buf,"end\n") == 0) break;
    write(sockfd, buf, strlen(buf));
  };

  close(sockfd);
  return 0;
}
```

Utilisez `nc -l 9090 -v` pour lancer un serveur TCP

Rappels : socket TCP en mode écoute

La fonction `listen()` est utilisée pour configurer un socket TCP en mode ”écoute” pour les connexions entrantes. Elle permet au serveur d'attendre et d'accepter des connexions des clients.

```
int listen(int sockfd, int backlog);
```

- `sockfd` : descripteur du socket
- `backlog` : nombre maximum de connexions en attente (en file d'attente) que le système peut gérer avant que de nouvelles connexions soient refusées. Ce nombre définit la taille de la file d'attente pour les connexions entrantes non encore acceptées par le serveur.

Rappels : connexion entrante sur socket TCP

La fonction `accept()` est utilisée pour accepter une connexion entrante sur un socket en mode écoute. Elle crée un nouveau socket pour gérer la communication avec le client une fois que la connexion a été établie.

```
int accept(int sockfd,
           struct sockaddr *addr,
           socklen_t *addrlen);
```

- `sockfd` : descripteur du socket serveur
- `addr` : pointeur vers une structure sockaddr qui recevra l'adresse du client connecté
- `addrlen` : pointeur vers la taille de la structure `sockaddr`.

Rappels : réception de données sur un socket TCP

La fonction recv() permet de recevoir des données à partir d'un socket TCP.

```
ssize_t recv(int sockfd,
             void *buf,
             size_t len,
             int flags);
```

- `sockfd` : descripteur du socket depuis lequel les données doivent être lues
- `buf` : tampon où les données reçues seront stockées
- `len` : taille maximale du tampon
- `flags` : mettre à 0

Rappels : implémentation en C d'un simple serveur TCP

Voici une implémentation en langage C d'un simple serveur TCP qui affiche les chaine de caractères envoyées par un client.

```c
#include <stdio.h>
#include <unistd.h>
#include <arpa/inet.h>

int main() {
    int server_fd, client_fd;
    struct sockaddr_in server_addr={0}, client_addr;
    char buf[1024];
    socklen_t addr_len = sizeof(client_addr);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr("172.18.88.214");
    server_addr.sin_port = htons(9090);

    bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
    listen(server_fd, 1);

    client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addr_len);

    while (1) {
        if (recv(client_fd, buf, sizeof(buf) - 1, 0) <= 0) break;
        printf("%s", buf);
    }
    close(client_fd);
    close(server_fd);
    return 0;
}
```

Rappels : serveur TCP avec multiples connexions

Voici un serveur TCP autorisant un nombre quelconque de connexions.

```c
#include <stdio.h>
#include <unistd.h>
#include <arpa/inet.h>

int main() {
    int server_fd, client_fd;
    struct sockaddr_in server_addr={0};
    char buf[1024];

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr("172.18.88.214");
    server_addr.sin_port = htons(9090);

    bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
    listen(server_fd, 5);

    while (1) {
      client_fd = accept(server_fd, NULL, NULL);
      if (client_fd == -1) break;
      if (fork() == 0) {
        close(server_fd);
        while (1) {
          if (recv(client_fd, buf, sizeof(buf) - 1, 0) <= 0) break;
          printf("%s", buf);
        }
      }
      else { close(client_fd); }
    }
    close(server_fd);
    return 0;
}
```

État des connexions

Il est possible de voir l'état des connexions TCP avec la commande `netstat -p tcp -l -n` (à adapter selon l'OS) :

```
→  synflooding git:(master) ✗ netstat -p tcp -l -n                    ~/enseignements/securite/M1/synflooding
Active Internet connections
Proto Recv-Q Send-Q  Local Address                      Foreign Address                    (state)
tcp4       0      0  192.168.1.144.58138               129.175.15.19.585                  SYN_SENT
tcp4       0      0  192.168.1.144.58137               129.175.38.59.585                  SYN_SENT
tcp4       0      0  192.168.1.144.58136               129.175.38.59.585                  SYN_SENT
tcp4       0      0  192.168.1.144.9090                192.168.1.144.58135                ESTABLISHED
tcp4       0      0  192.168.1.144.58135               192.168.1.144.9090                 ESTABLISHED
tcp4       0      0  192.168.1.144.9090                192.168.1.144.58134                ESTABLISHED
tcp4       0      0  192.168.1.144.58134               192.168.1.144.9090                 ESTABLISHED
tcp6       0      0  2a01:e0a:d1a:61b0:79b5:f72b:10dd:818.58103  2a00:1450:400c:c06::6d.993  ESTABLISHED
tcp6       0      0  fe80::ce2:5aba:b5f:acc8%en0.58099  fe80::100d:e6bf:c447:367%en0.49153  ESTABLISHED
tcp6       0      0  fe80::ce2:5aba:b5f:acc8%en0.58051  fe80::18f9:b4b2:ec0d:5ad0%en0.59393  ESTABLISHED
tcp6       0      0  2a01:e0a:d1a:61b0:79b5:f72b:10dd:818.58093  2a00:1450:400c:c06::6d.993  ESTABLISHED
```

L'état ESTABLISHED indique qu'une connexion est active et que les données peuvent être échangées.

Limitations des sockets TCP ou UPD

Les sockets TCP/UDP masquent les détails du réseau ce qui complique l'envoi de paquets malveillants ou personnalisés

Par exemple, il n'est pas possible de manipuler les en-têtes des paquets envoyés : c'est le système d'exploitation qui gère l'encapsulation, l'en-tête, les retransmissions et l'intégrité des paquets.

Par ailleurs, c'est également le système qui gère l'établissement des connexions (handshake) TCP, la vérification des données (checksum) et le contrôle de congestion : tout cela rend très difficile la réalisation d'attaques par (D)DoS.

Enfin, les sockets TCP/UDP sont soumis à des politiques de filtrage (pare-feu, NAT, etc.) qui empêche l'envoi de paquets falsifiés (spoofing) ou des attaques par saturation.

Les sockets Raw

Un socket RAW un type de socket qui permet de manipuler directement les paquets réseau sans abstraction, à un niveau bas du modèle OSI.

Il permet aux applications d'envoyer et de recevoir des paquets avec un contrôle total sur tous les aspects du paquet, y compris l'en-tête du protocole réseau. Par exemple, un socket RAW permet d'envoyer des paquets IP avec un en-tête personnalisé.

On peut déclarer un socket RAW de la manière suivante :

```
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

Configuration d'un socket RAW

Pour envoyer des paquets personnalisés, il faut tout d'abord définir les options du socket RAW à l'aide de la fonction `setsockopt()`

Par exemple, pour définir un socket qui va permettre d'envoyer des paquets au niveau IP, on écrira :

```
int enable = 1;
setsockopt(sock,
           IPPROTO_IP,
           IP_HDRINCL,
           &enable, sizeof(enable));
```

- `IPPROTO_IP` est le niveau de protocole du socket `sock`
- `IP_HDRINCL` indique que l'en-tête du paquet sera inclus explicitement par l'utilisateur
- La variable `enable` permet de définir la valeur de l'option passée à `setsockopt` (1 = activation)

Header IP

Lorsqu'on utilise un socket RAW, il faut définir manuellement l'en-tête des paquets que l'on souhaite envoyer.

4	4	8	16
Version	HL	Type of service	Total length
Identification			Fragment offset
TTL		Protocol	Checksum
Source IP Address			
Destination IP Address			

La structure du header IP est définie dans le fichier `netinet/ip.h`

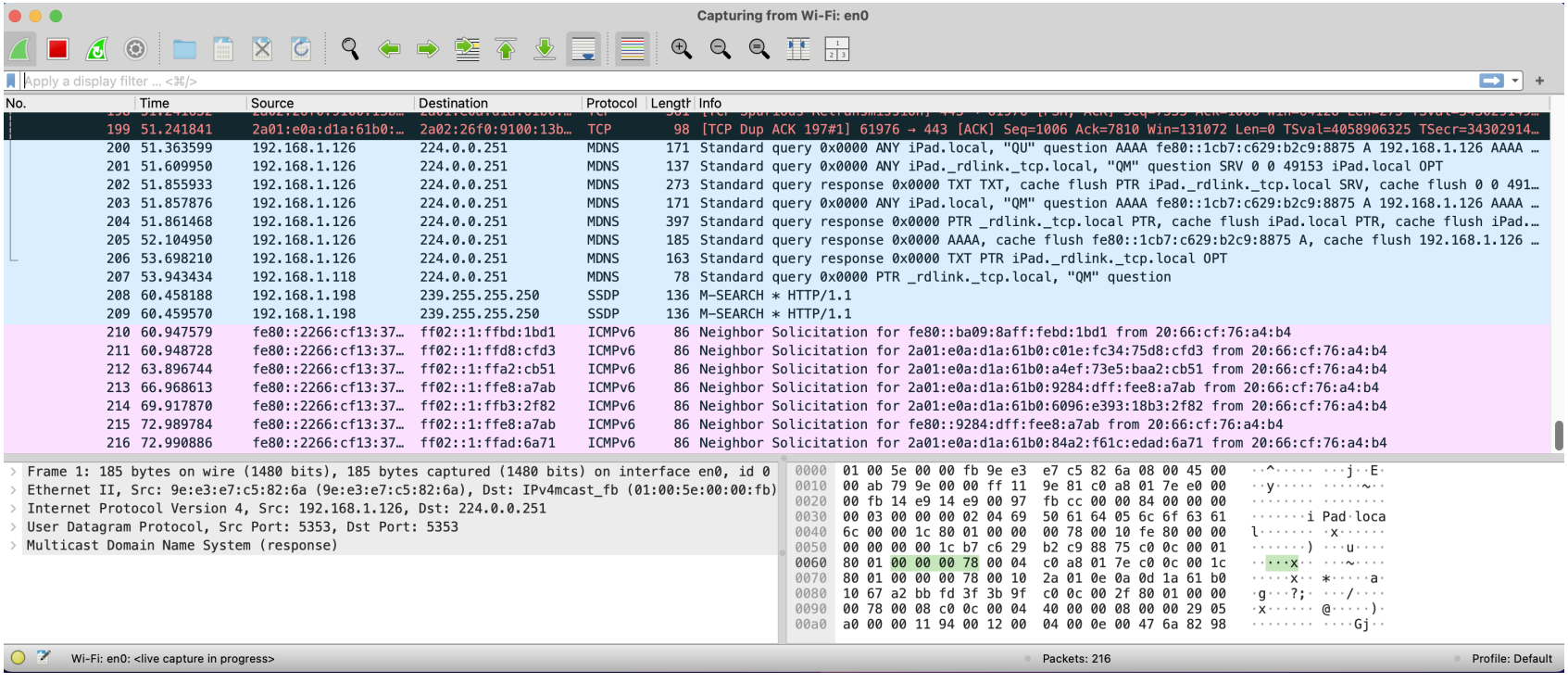
```
struct ip {
  u_int   ip_v:4,                    /* version IPv4 = 4 */
          ip_hl:4;                   /* header length (en mots de 4 octets) */
  u_char  ip_tos;                    /* type of service (0 = best effort)*/
  u_short ip_len;                    /* total length (header + données)*/
  u_short ip_id;                     /* identifiant unique du paquet*/
  u_short ip_off;                    /* fragment offset field */
  u_char  ip_ttl;                    /* time to live */
  u_char  ip_p;                      /* protocol */
  u_short ip_sum;                    /* checksum */
  struct  in_addr ip_src, ip_dst;    /* source and dest address */
};
```

Envoi d'un paquet IP avec un socket RAW

```c
#include <unistd.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>

int main(){
  struct sockaddr_in dest;
  struct ip ip_hdr;

  int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
  int enable = 1;
  setsockopt(sock,IPPROTO_IP,IP_HDRINCL,&enable, sizeof(enable));

  dest.sin_family = AF_INET;
  dest.sin_addr.s_addr = inet_addr("192.168.1.144");

  ip_hdr.ip_hl = 5; // 20 octets header IP / sizeof (uint32_t)
  ip_hdr.ip_v = 4;
  ip_hdr.ip_tos = 0;
  ip_hdr.ip_len = 20; // 20 octets dans un header IP
  ip_hdr.ip_id = htons (1);
  ip_hdr.ip_off = htons(IP_DF);
  ip_hdr.ip_ttl = 255;
  ip_hdr.ip_p = 42; // protocole couche supérieure
  ip_hdr.ip_src.s_addr = inet_addr("1.2.3.4");
  ip_hdr.ip_dst.s_addr = dest.sin_addr.s_addr;
  ip_hdr.ip_sum = 0; // pas important pour sendto

  sendto(sock,&ip_hdr,20,0,
         (struct sockaddr *) &dest, sizeof(dest));

  close (sock) ;
  return 0;
}
```

WireShark

Utilisez Wireshark pour inspecter le trafic réseau et observer vos paquets.



Exemple : les paquets ICMP

La couche ICMP (Internet Control Message Protocol) fait partie de la couche Internet dans le modèle TCP/IP. ICMP est utilisé pour gérer et contrôler les erreurs dans le réseau. Par exemple, `ping` utilise ICMP pour tester la connectivité réseau.

```
> ping google.com
PING google.com (142.250.74.238) : 56 data bytes
64 bytes from 142.250.74.238 : icmp_seq=0 ttl=115 time=3.468 ms
64 bytes from 142.250.74.238 : icmp_seq=1 ttl=115 time=4.495 ms
...
```

ICMP est un protocole sans connexion directement encapsulé dans le protocole IP (il utilise donc les mêmes adresses IP)

Les messages ICMP sont Echo Request / Echo Reply, Destination Unreachable ou Time Exceeded.

Utilisez Wireshark pour visualiser les paquets émis par la commande `ping`

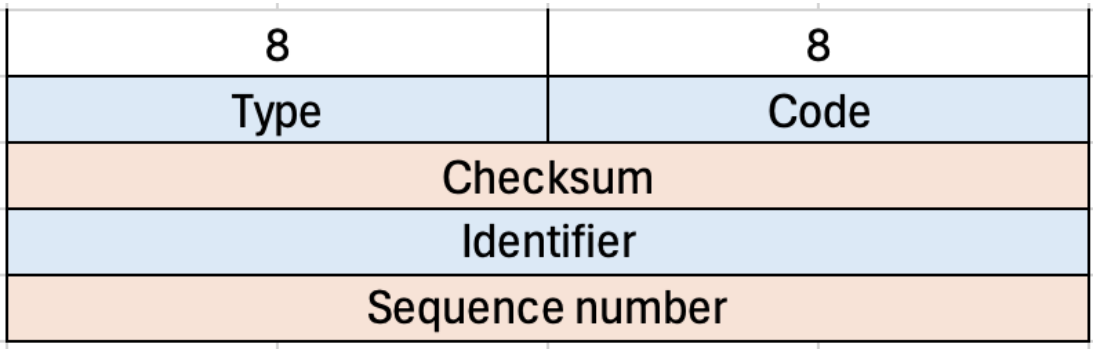
Exercice 1 : Attaque Smurf

L'attaque Smurf est une attaque par amplification DDoS (Distributed Denial of Service) qui exploite le protocole ICMP.

Principe de l'attaque :

- Envoi d'une requête ping usurpée : L'attaquant envoie un grand nombre de requêtes ICMP Echo Request à une adresse de broadcast d'un réseau, en falsifiant l'adresse source pour qu'elle corresponde à celle de la victime.
- Amplification : Tous les hôtes du réseau répondent à la requête ping en envoyant une ICMP Echo Reply à l'adresse source (qui est en réalité celle de la victime).
- Saturation de la bande passante : La victime est submergée par un grand nombre de réponses ICMP, ce qui peut ralentir voire rendre son système indisponible.

Exercice 1 : Attaque Smurf (header ICMP)



La structure `icmp` du header ICMP est définie dans le fichier `netinet/ip_icmp.h`. Elle contient les champs suivants :

```
u_char  icmp_type;  /* type des messages : 8 pour ECHO Request */
u_char  icmp_code;  /* complémente icmp_type (0 pour ECHO Request) */
u_short icmp_cksum; /* intégrité du header (à calculer) */
n_short icmp_id     /* pour identifier la requête */
n_short icmp_seq    /* numéro de séquence pour suivre les requêtes*/
```

Exercice 1 : Attaque Smurf (Checksum)

Il faut obligatoirement calculer le checksum d'un paquet personnalisé pour que ce dernier soit accepté.

```c
uint16_t checksum(uint16_t *data, size_t length) {
    uint32_t sum = 0;

    // Additionner les mots de 16 bits
    while (length > 1) {
        sum += *data++;
        length -= 2;
    }
    // Si la longueur est impaire, ajouter l'octet restant
    if (length == 1) {
        sum += *(uint8_t *)data;
    }
    // Ajouter le carry (si sum dépasse 16 bits)
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    // Complément à 1
    return (uint16_t)~sum;
}
```

Astuce : mettre d'abord 0 dans `icmp_cksum`, puis changer sa valeur en utilisant la fonction `checksum`

Exercice 1 : Attaque Smurf (fin)

En utilisant des sockets RAW, réaliser un programme qui implémente l'attaque Smurf.

Bien sûr, vous ne viserez pas une adresse de broadcast, mais votre propre adresse IP, et vous ne submergerez pas le réseau avec vos paquets ICMP falsifiés ;-)

La bibliothèque `scapy`

Scapy est une bibliothèque Python pour la manipulation de paquets réseau. Elle permet de construire, envoyer, capturer et analyser des paquets de nombreux protocoles (IP, TCP, UDP, ICMP, ARP, etc.).

Installation de scapy avec `pip install scapy`

Documentation :
`https://scapy.readthedocs.io/en/latest/`

Spoofing ICMP avec `scapy`

Le spoofing de paquets (IP, ICMP, etc.) est très facile avec `scapy`. Par exemple, le code ci-dessous permet de spoofer des paquets ICMP.

```
from scapy.all import IP, ICMP, send

ip = IP(src='1.2.3.9', dst='172.18.74.168',chksum=0)
icmp = ICMP()
pkt = ip/icmp

pkt.show()
send(pkt,verbose=0)
```

- Les constructeurs `IP` et `ICMP` permettent de créer les en-têtes des paquets correspondant à ces couches de protocoles.
- L'opérateur (surchargé) `/` permet de concaténer l'en-tête `IP` et l'en-tête `ICMP`. La fonction `send` permet d'envoyer un paquet sur le réseau.

Approche hybride

La bibliothèque `scapy` est très puissante, mais elle peut parfois être trop lente pour construire et envoyer des paquets dans une attaque qui nécessite des envois successifs très rapides.

Une approche hybride consiste à :

1. Fabriquer un modèle de paquets avec `scapy` et de l'enregistrer au format binaire sur disque

```
with open('ip.bin', 'wb') as f :
    f.write (bytes (pkt))
```

2. Relire ce fichier dans un programme C et le stocker dans un tableau d'octets.

```
FILE *f = fopen("ip.bin", "rb");
unsigned char ip[SIZE];
int n = fread (ip, 1, SIZE, f);
```

3. Personnaliser ce modèle pour spoofer des paquets rapidement.

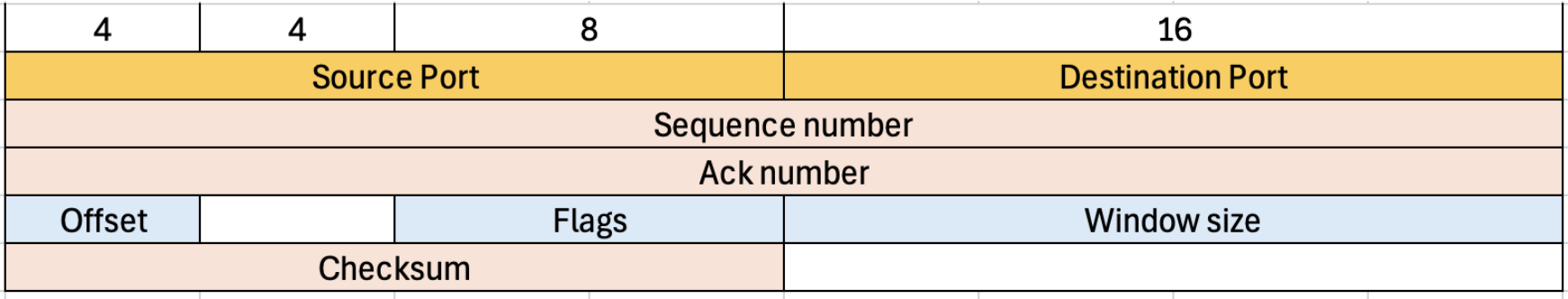
Exercice 2 : Attaque Smurf par approche hybride

Refaire une attaque Smurf en C en utilisant l'approche hybride décrite ci-dessus.

Attaque SynFlooding

Afin de mettre en place l'attaque Syn Flooding, il est nécessaire de rappeler l'établissement des connexions dans le protocole TCP, ainsi que le contenu des headers TCP/IP.

Rappels : structure d'un header TCP



La structure `tcphdr` du header TCP est définie dans le fichier `netinet/tcp.h`. Elle contient les champs suivants :

```
struct tcphdr {
 unsigned short  th_sport;  /* source port */
 unsigned short  th_dport;  /* destination port */
 tcp_seq th_seq;            /* sequence number */
 tcp_seq th_ack;            /* acknowledgement number */
 unsigned int    th_off:4,  /* data offset (size of header = 5)  */
 unsigned char   th_flags;  /* voir slide suivant */
 unsigned short  th_win;    /* window size (debit) */
 unsigned short  th_sum;    /* checksum */
 unsigned short  th_urp;    /* urgent pointer */
};
```

Rappels : flags d'un header TCP

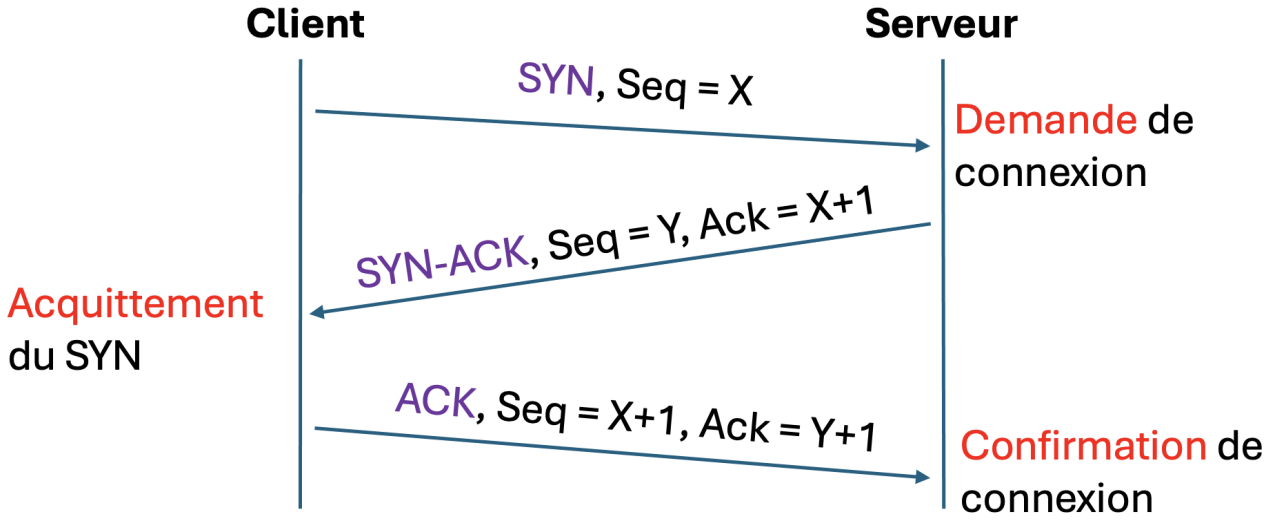
Voici les principaux flags d'un header TCP sont :

```
#define TH_FIN  0x01  // L'émetteur souhaite terminer la connexion.
#define TH_SYN  0x02  // Pour initier une connexion TCP (voir prochain slide)
#define TH_RST  0x04  // Pour réinitialiser la connexion TCP
#define TH_ACK  0x10  // Indique que le champ "Ack Number" est valide
```

Rappels : établissement des connexions TCP

L'établissement d'une connexion TCP suit un processus en trois étapes appelé 3-Way Handshake.

Ce mécanisme permet aux deux parties de synchroniser leurs *numéros de séquence* et de s'assurer que la connexion est ouverte et prête à échanger des données.



État des connexions (suite)

On peut suivre les étapes d'une connexion TCP grâce à la commande `netstat -p tcp -l -n` (à adapter selon l'OS).

Voici les états possibles lors de l'établissement d'une connexion :

SYN_SENT	SYN envoyé, en attente de SYN-ACK
SYN_RCVD	SYN reçu, en attente de ACK
ESTABLISHED	ACK reçu, connexion établie

Réception d'un paquet TCP

Lorsqu'il reçoit un paquet SYN, le serveur crée un Transmission Control Block (TCB) pour stocker les informations de connexion.

Tant que la connexion reste semi-ouverte (en attente de l'ACK du client), ce TCB est placé dans une file dédiée.

Si le serveur reçoit l'ACK final, il déplace le TCB ailleurs. Sinon, il réémet le SYN-ACK et, en cas d'absence de réponse, le TCB finit par expirer et être supprimé.

Exercice 3 : Attaque SynFlooding

L'attaque SYN Flooding est une attaque DoS qui exploite la phase d'établissement de connexion TCP en surchargeant le serveur de paquets SYN et sans jamais répondre aux SYN-ACK envoyés par le serveur. Cela a pour effet de saturer la file d'attente des connexions semi-ouvertes

En utilisant les techniques de spoofing vues précédemment, implémenter une attaque Syn Flooding :

- en Python avec `scapy`
- avec la méthode hybride C + Python
- en C directement.

Vous utiliserez WireShark et `netstat` pour visualiser les paquets envoyés et l'effet de l'attaque sur l'états des connexions.

Exercice 4 : contre-mesures pour Syn Flooding

En effectuant des recherches sur Internet, listez et expliquez les principales contre-mesures mises en place pour se protéger contre les attaques SYN Flooding.

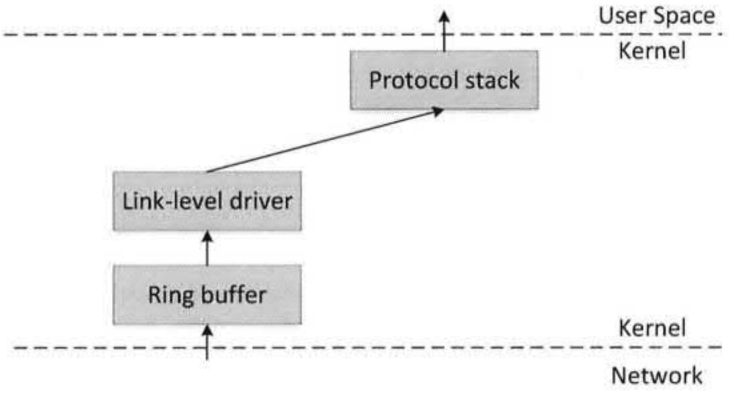
Techniques de sniffing

Le sniffing est une technique de surveillance du trafic réseau utilisée pour capturer et analyser les paquets de données circulant entre les appareils.

Cette technique peut être utilisée à des fins légitimes (diagnostic réseau, cybersécurité) comme dans l'outil WireShark, ou malveillantes (espionnage, vol d'informations sensibles) pour réaliser des attaques.

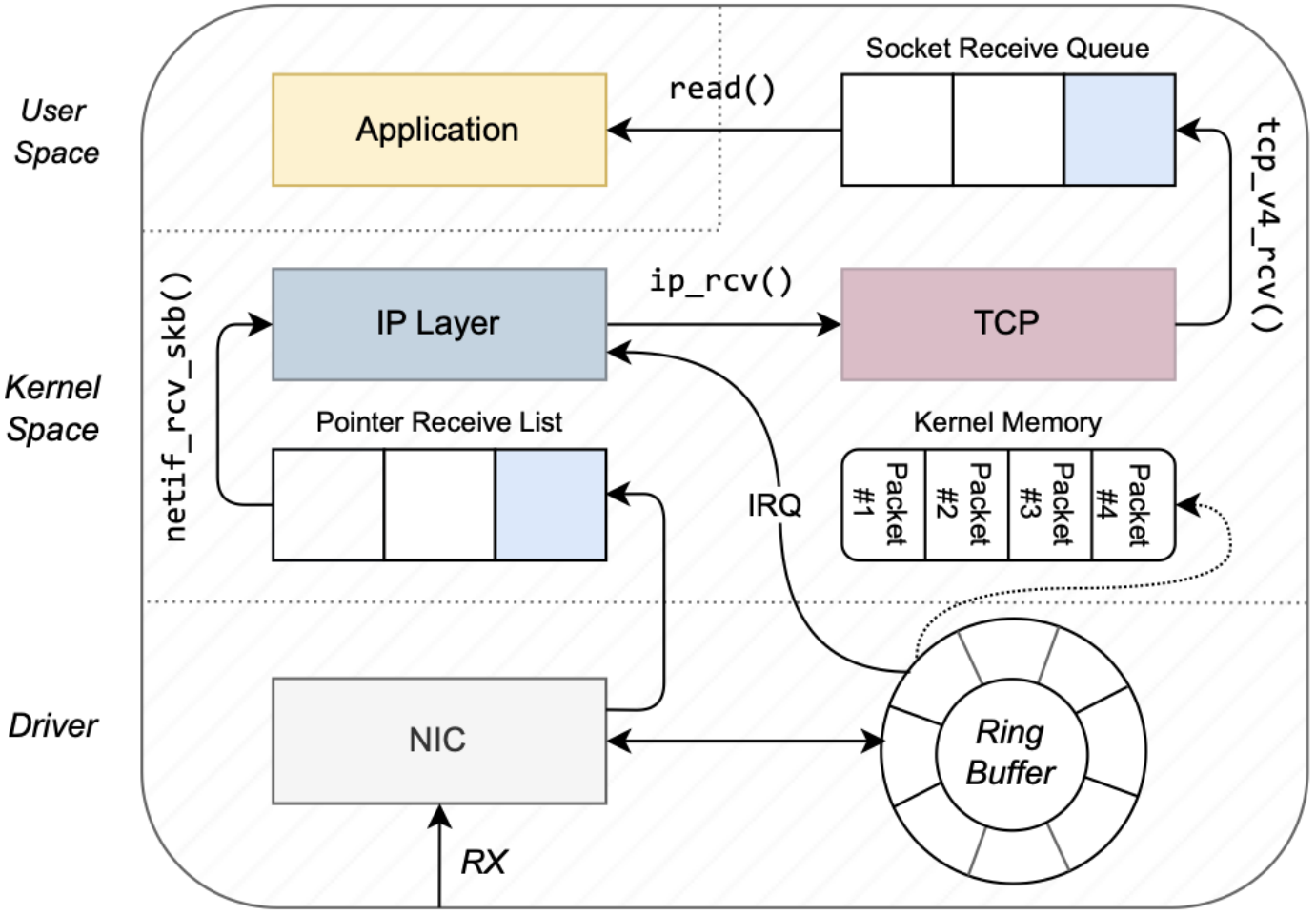
Interface réseau

Sur un réseau local (Ethernet ou WiFi), les machines sont connectées via une carte réseau (NIC) qui a une adresse MAC. Sur un réseau sur Ethernet ou Wifi utilise un medium de communication partagé où les paquets sont *broadcastés* à toutes les machines.



La carte réseau écoute les trames sur le réseau et, lorsqu'une trame correspond à son adresse MAC, elle la stocke dans son Ring Buffer. La carte interrompt ensuite le processeur pour signaler la présence d'un nouveau paquet, qui est copié dans une file d'attente du pilote de liaison (Link-level driver). Le noyau traite ensuite ces paquets et les envoie aux programmes en espace utilisateur.

Parcours d'un paquet



Le mode promiscuité

Les trames qui ne sont pas destinées à une carte réseau (NIC) donnée sont rejetées au lieu d'être envoyées au processeur pour traitement.

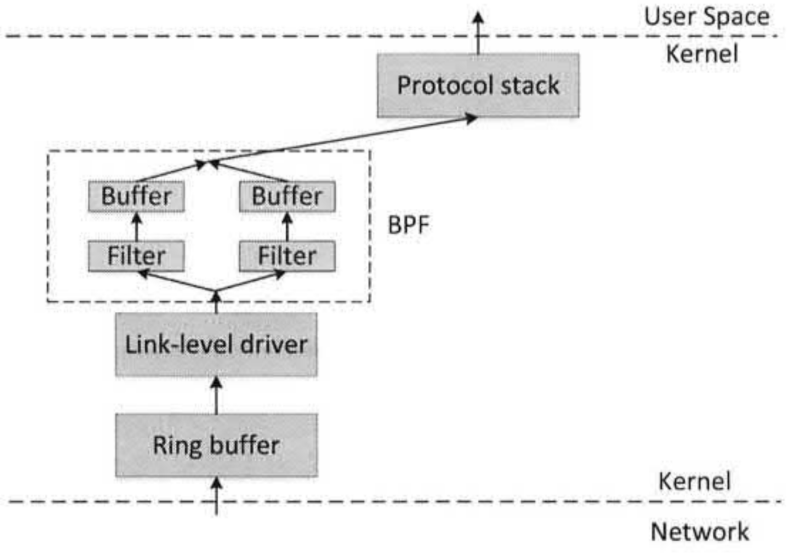
Le mode promiscuous (ou "mode promiscuïté") fait référence à une configuration de la carte réseau qui lui permet d'accepter toutes les trames qu'elle reçoit, même si elles ne lui sont pas adressées.

Si un sniffer (programme de capture) est enregistré auprès du noyau, toutes ces trames seront finalement transmises par le noyau au *sniffer*.

Il faut généralement avoir des privilèges élevés (par ex. root) pour mettre une carte réseau dans ce mode.

Les filtres de paquets BPF

Pour filtrer efficacement les paquets réseau, on utilise des filtres BPF (Berkeley Packet Filter), qui sont des programmes injectés depuis l'espace utilisateur vers le noyau.



Les filtres BPF correspondent à du code bas-niveau interprété par un pilote BPF dans le noyau. Pour simplifier l'écriture de ces filtres, on utilisera l'API `pcap`, qui dispose d'un compilateur permettant de construire des filtres BPF à partir d'une représentation de haut niveau.

API `pcap`

L'API `pcap` (*packet capture*) fournit une interface indépendante de la plateforme pour capturer efficacement des paquets.

`pcap` inclut un compilateur permettant aux programmeurs de spécifier des règles de filtrage à l'aide d'expressions booléennes lisibles ; le compilateur traduit ces expressions en pseudo-code BPF, qui peut ensuite être utilisé par le noyau.

Pour installer libpcap : `sudo apt-get install libpcap-dev`

Pour utiliser libpcap en C : `#include <pcap.h>`

Pour compiler : `gcc -o <exec> <source.c> -l pcap`

Ouverture d'une interface pour la capture

La fonction `pcap_open_live` ouvre un socket RAW en mode promiscuité sur une interface réseau donnée.

```
pcap_t *pcap_open_live(const char *device,
                       int snaplen,
                       int promisc,
                       int to_ms,
                       char *errbuf);
```

- `device` : nom de l'interface réseau à surveiller
- `snaplen` : taille maximale des paquets à capturer
- `promisc` : mode promiscuité (1 on, 0 off)
- `to_ms` : timeout
- `errbuf` : tampon pour message d'erreur.

La fonction renvoie un descripteur qui peut ensuite être utilisé pour lire les paquets capturés.

Capture des paquets

La fonction `pcap_loop` prend en charge la capture de paquets et appelle une fonction de rappel (callback) chaque fois qu'un paquet est capturé.

```
int pcap_loop(pcap_t *ptr,
              int cnt,
              pcap_handler callback,
              u_char *user_data);
```

- `ptr` : pointeur vers un descripteur de capture
- `cnt` : nombre de paquets à capturer
- `callback` : fonction appelée chaque fois qu'un paquet est capturé
- `user_data` : pointeur vers des données qui peuvent être utilisées dans le callback.

Compilation des filtres BPF

La fonction `pcap_compile` compile une expression de filtre en texte clair (comme ”tcp port 80”) en un filtre BPF optimisé.

```
int pcap_compile(pcap_t *ptr,
                 struct pcap_program *fp,
                 const char *str,
                 int optimize,
                 bpf_u_int32 netmask);
```

- `ptr` : descripteur de capture
- `fp` : pointeur vers une structure `pcap_program` pour le filtre compilé
- `str` : expression de filtre à compiler (par exemple, ”tcp”, ”icmp”, ”host 192.168.0.1”, ”port 80”, ou des combinaisons booléennes)
- `optimize` : filtre optimisé (1) ou non (0)
- `netmask` : masque de sous-réseau1

Appliquer un filtre BPF

La fonction `pcap_setfilter` applique un filtre BPF à une session de capture ouverte avec libpcap. Cela permet de ne capturer que les paquets qui correspondent aux critères spécifiés dans le filtre.

```
int pcap_setfilter(pcap_t *ptr,
                   struct bpf_program *fp);
```

- `ptr` : descripteur de capture de paquets
- `fp` : un programme BPF compilé, qui définit le filtre à appliquer.

Exercice 5 : capture des paquets ICMP

La couche ICMP (Internet Control Message Protocol) fait partie de la couche Internet dans le modèle TCP/IP. ICMP est utilisé pour gérer et contrôler les erreurs dans le réseau. Par exemple, `ping` utilise ICMP pour tester la connectivité réseau.

```
> ping google.com
PING google.com (142.250.74.238) : 56 data bytes
64 bytes from 142.250.74.238 : icmp_seq=0 ttl=115 time=3.468 ms
64 bytes from 142.250.74.238 : icmp_seq=1 ttl=115 time=4.495 ms
...
```

ICMP est un protocole sans connexion directement encapsulé dans le protocole IP (il utilise donc les mêmes adresses IP)

Les messages ICMP sont Echo Request / Echo Reply, Destination Unreachable ou Time Exceeded.

Implémentez un sniffer avec `libpcap` pour capturer les paquets émis par la commande `ping`

Sniffer ICMP avec pcap

Implémentation d'un simple sniffer en C avec l'API pcap

```c
#include <pcap.h>

void cb_packet(u_char *args, const struct pcap_pkthdr *header,
               const u_char *packet)
{
  printf("Réception d'un paquet\n");
}

int main(){

  pcap_t *pfd;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program bpf;
  char filter[] = "icmp";

  pfd = pcap_open_live("en0",1024,1,1000,errbuf);
  pcap_compile(pfd,&bpf,filter,0,0);
  pcap_setfilter(pfd,&bpf);
  pcap_loop(pfd,-1,cb_packet,NULL);
  pcap_close(pfd);
  return 0;
}
```

Les paquets Ethernet

La structure des paquets Ethernet est la suivante :

Préambule	SFD	MAC dest	MAC src	Type	Données

MAC dest et MAC src sont les adresses MAC destination et source.

Type indique le protocole de niveau supérieur : `ETHERTYPE_IP`, `ETHERTYPE_ARP`, `ETHERTYPE_IPV6`, etc.

La structure des paquets ethernet est définie dans le fichier `net/ethernet.h`

```
typedef struct  ether_header {
    u_char  ether_dhost[ETHER_ADDR_LEN];
    u_char  ether_shost[ETHER_ADDR_LEN];
    u_short ether_type;
} ether_header_t;
```

Exercice 6 : traitement des paquets Ethernet

Étendre votre sniffer afin d'afficher les paquets IP ou ICMP reçus sur votre machine.

Exercice 6 : solution

```c
void handle_packet(u_char *args, const struct pcap_pkthdr *header,
                   const u_char *packet)
{
  struct ether_header *eth = (struct ether_header *) packet;

  if (ntohs(eth->ether_type) == ETHERTYPE_IP) {
    struct ip *ip = (struct ip *)(packet + sizeof(struct ether_header));
    printf("IP: %s -> %s, Proto: ",
           inet_ntoa(ip->ip_src),
           inet_ntoa(ip->ip_dst));

    switch (ip->ip_p) {
    case IPPROTO_ICMP: printf("ICMP\n"); break;
    case IPPROTO_TCP:  printf("TCP\n"); break;
    case IPPROTO_UDP:  printf("UDP\n"); break;
    default:           printf("Other\n");
    }
  }
}
```

Sniffing avec `scapy`

La bibliothèque `scapy` permet également de sniffer des paquets à l'aide de la fonction `sniff` qui a la signature suivante :

```
sniff(
  iface=None,  # Interface réseau à écouter (ex: "eth0", "wlan0")
  count=0,    # Nombre de paquets à capturer (0 pour infini)
  prn=None,     # Fonction de callback pour chaque paquet
  store=1,      # Stocker les paquets capturés (1=True, 0=False)
  filter=None,   # Filtre BPF (Berkeley Packet Filter)
  lfilter=None,  # Fonction Python pour filtrer les paquets
  offline=None,  # Fichier pcap à lire au lieu de l'interface réseau
  timeout=None,  # Temps maximum pour capturer les paquets (en secondes)
  opened_socket=None, # Utiliser un socket existant au lieu d'en créer un
  stop_filter=None, # Fonction de rappel pour arrêter la capture
  *args, **kwargs)   # Autres arguments
```

La fonction `sniff` offre la possibilité de filtrer les paquets capturés en fonction de divers critères (protocole, adresse IP, port, etc.). Une fois les paquets capturés, `sniff` peut les traiter, analyser ou les enregistrer

La classe `Packet`

Les paquets capturés avec `scapy` sont des instances de la classe `Packet`. Ils contiennent des champs et des méthodes pour accéder aux différentes couches du protocole (Ethernet, IP, TCP, etc.).

L'accès aux couches d'un paquet utilise une surcharge de l'opérateur `[]`. Ainsi, si `pkt` est un paquet de la classe `Packet`, alors `pkt[IP]` permet d'accéder aus informations de la couche `IP` du paquet `pkt`.

Les champs de `pkt[IP]` sont ceux des paquets IP. On trouve par exemple :

- `proto` : Protocole de la couche supérieure
- `src` : Adresse IP source.
- `dst` : Adresse IP de destination.

Par ailleurs, la notation `IP in pkt` permet de savoir si le paquet `pkt` contient une couche IP.

Exercice 7 : sniffer ICMP en `scapy`

Écrire un sniffer en Python avec `scapy` afin d'afficher les paquets IP ou ICMP reçus sur votre machine.

Exercice 7 : solution

```
from scapy.all import *

def print_pkt(pkt):
    protos = { 1 : 'ICMP', 6 : 'TCP', 17 : 'UDP' }
    if IP in pkt:
        print(f'From {pkt[IP].src} to {pkt[IP].dst}')
        print(f'Protocol : {protos.get(pkt[IP].proto,"Unkown")}\n')

pkt = sniff(filter='ip or icmp',prn=print_pkt,iface="en0")
```