Dissimulation de données dans un flux TCP, implémentation dans le noyau Linux

Fabien Terrace David Vanderhaeghe Cédric Vincent

13 décembre 2004

Table des matières

1 Introduction **2**
1.1 Sujet . . . 2
1.2 Travaux relatifs . . . 2

2 Protocole TCP **2**
2.1 Étude . . . 2
2.1.1 Transmission . . . 2
2.1.2 Codage . . . 2
2.1.3 Structure . . . 4
2.2 Implantation . . . 4
2.2.1 Branchement dans le noyau . . . 4
2.2.2 Émetteur A . . . 6
2.2.3 Récepteur B . . . 6
2.2.4 Exemple de capture . . . 7
2.3 Problèmes et difficultés . . . 7
2.3.1 Codage des données . . . 7
2.3.2 Fast Path . . . 7
2.3.3 Tests . . . 7
2.3.4 Complexité . . . 10

3 Module **10**

4 Codage **10**

5 Concrètement **10**
5.1 Sécurité . . . 10
5.1.1 Améliorations . . . 10

6 Annexes **10**
6.1 Installation . . . 10
6.2 Utilisation . . . 11

1 Introduction

Dans le cadre de notre cursus en Maîtrise Informatique à l'Université Claude Bernard Lyon 1, nous devons réaliser un projet pour le contrôle continue du cours de Codage dispensé par monsieur Yvan Gérard.

1.1 Sujet

Le sujet de ce projet est *inspiré* d'un passage du livre *Cryptographie Appliquée* [15] :

> Une autre méthode est qu'Alice prenne un message banal et le passe par un code correcteur d'erreur. Puis elle peut introduire des erreurs correspondant au message secret chiffré. A la réception, Bernard peut extraire les erreurs pour reconstruire le message secret et le déchiffrer.

Nous souhaitons donc *parasiter* l'émission d'un flux de données TCP/IP en ajoutant des erreurs codant un message *caché*. Ces erreurs seront judicieusement insérées pour que l'on puisse extraire le message dissimulé *e*t reconstruire le flux de données original.

Finalement, on peut considérer que nous parasitons un système de codage pour faire notre propre système de codage :).

1.2 Travaux relatifs

Comme nous l'avons vu à la section 1.1, l'idée de dissimuler des données dans un flux n'est pas nouvelles. En effet il existe de nombreuses études [12] [5] [16] sur les techniques de dissimulation de donnée dans un flux TCP/IP et sur les moyens de les détectés.

Il existe des outils [12][8] qui permettent aussi ce genre de techniques, il suffit de jeter un coup d'oeil sur des site de sécurité informatique tel que : http ://www.securityfocus.com/tools/category/97

Le problème est que toutes ses études et outils créent *explicitement* un flux de communication (genre tunneling), or nous souhaitons plus de discrétion et les données transitées doivent être *légitime*, notre approche est donc différente

2 Protocole TCP

2.1 Étude

2.1.1 Transmission

Le protocole TCP [9] consiste à transmettre des data-grammes, que l'on appelle *segment*, sans perte ni duplication. Ceci est réalisé à l'aide du système d'*accusé de réception* (**ack**nolegment). Chaque segment est émis avec un numéro de séquence qui sert au récepteur B à envoyer un **ack**. Ainsi l'émetteur A sait si l'information qu'il voulait transmettre est « arrivée à bon port ».

Á chaque envoie de segment, l'émetteur arme une temporisation afin d'attendre l'accusé de réception. Lorsque la temporisation expire et que l'émetteur n'a pas reçu le **ack** correspondant, il considère que le segment est perdu et donc le ré-émet (figure 1 b et c).

Il se peut qu'un segment soit ré-émis alors que le récepteur l'a bien reçu (figure 1 c), cela ce produit par exemple en cas de problème de congestion ou par la perte du **ack**. Heureusement le récepteur garde une trace des numéros de séquence reçus, ce qui lui permet d'éliminer les doublons, cela permet aussi de gérer les cas où les segments arrive dans le désordre (figure 1 d).

2.1.2 Codage

La figure 2 représente le codage d'un segment TCP.

La somme de contrôle est *simplement* le complément à 1 de la somme des compléments à 1 sur 16 bits, ce calcul s'effectuant sur l'en-tête et les données.

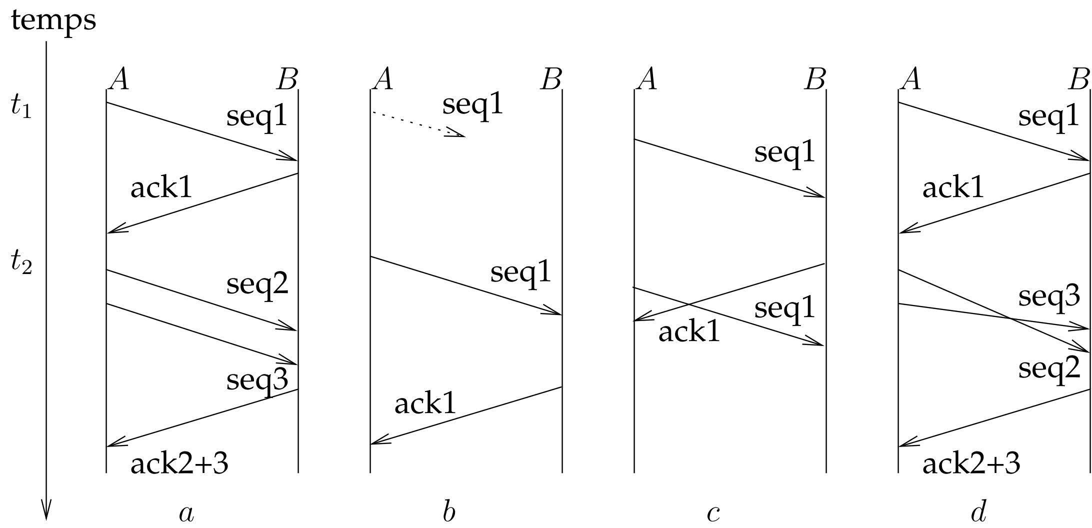


FIG. 1 – Séquences TCP.

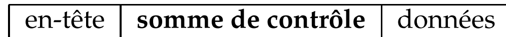


FIG. 2 – Codage d'un segment TCP.

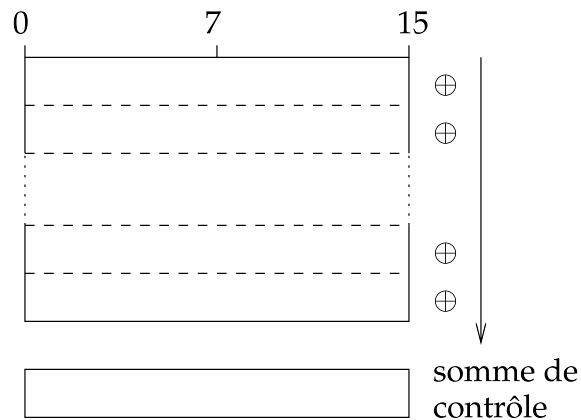


FIG. 3 – Somme de contrôle TCP.

Ce système de codage n'est pas correcteur, et ne détecte qu'un nombre impaire d'erreur sur une même position (alignée sur 16 bits), c'est-à-dire que s'il y a une erreur à la position $a \times 16 + x$ et une erreur à la position $b \times 16 + x$ dans un même segment, alors elle ne sera pas détectée...

Par manque de temps, nous n'avons pas introduit des erreurs dans le résultat de la somme de contrôle, comme nous voulions le faire, mais dans l'en-tête du paquet TCP. En fait, cela revient à un problème équivalent !

2.1.3 Structure

La figure 4 représente la structure d'un segment TCP.

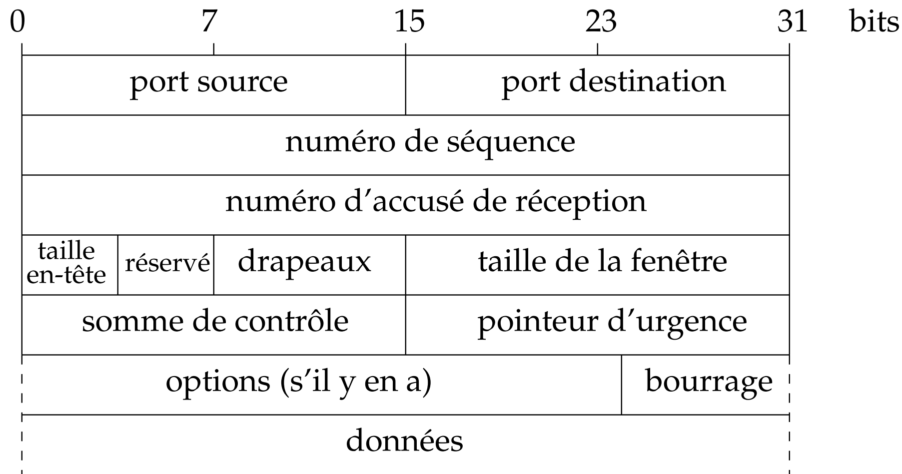


FIG. 4 – Paquet TCP.

Parmi les champs [11] de cette structure, nous avons principalement utilisé :

numéro de séquence : donne la position du segment dans le flux de données envoyées par l'émetteur, c'est-à-dire la place dans ce flux du premier octet de données transmis dans ce segment.

numéro d'accusé de réception : contient en fait le numéro de séquence suivant que le récepteur s'attend à recevoir, c'est-à-dire le numéro de séquence du dernier octet reçu avec succès plus 1. De manière précise, TCP n'acquitte pas un à un chaque segment qu'il reçoit, mais acquitte l'ensemble du flot de données jusqu'à l'octet $k - 1$ en envoyant un acquittement de valeur k.

pointeur d'urgence : c'est **ici** que nous introduisons nos codes cachés. Il s'agit en fait d'un offset positif qui, ajouté au numéro de séquence du segment indique le numéro du dernier octet de donnée urgente. Il faut également que le drapeau URG soit positionné à 1 pour indiquer que le pointeur est valide, et pour que la pile TCP/IP passe le plus rapidement possible les données à l'application associée à la connexion.

2.2 Implantation

2.2.1 Branchement dans le noyau

Afin de capturer et parasiter les paquets entrants et sortants, nous avons modifié les fichiers **linux/net/ipv4/tcp_input.c** et **linux/net/ipv4/tcp_output.c** des source du noyau Linux, ceux-ci s'occupant de l'envoie et de la réception des paquets TCP/IP.

Notre système de branchement est simple :

– dans un premier temps nous déclarons une fonction de parasitage dans le noyau que l'on définie à NULL (figure 5) ;
– puis, lorsque notre module se charge (ou décharge), nous définissons correctement cette fonction de parasitage (figure 6) ;
– enfin, dans le noyau lui-même, nous avons un branchement sur cette fonction si elle est correctement définie, c'est-à-dire lorsque notre module est chargé en mémoire (figure 7).

```
#if defined(CONFIG_TCP_HIDE)
int (*tcphide_parasite_1)(struct sock *sk,
                          struct tcphdr *th,
                          struct sk_buff *skb,
                          struct tcp_opt *tp) = NULL;
#endif
```

FIG. 5 – Déclaration et définition à NULL de la fonction de parasitage dans le noyau.

```
static int __init tcphide_init(void)
{
    tcphide_parasite_1 = tcphide_send;
    tcphide_parasite_2 = tcphide_recv;
    printk("TCPhide init\n");
    return 0;
}

static void __exit tcphide_cleanup(void)
{
    tcphide_parasite_1 = NULL;
    tcphide_parasite_2 = NULL;
    printk("TCPhide exit\n");
}
```

FIG. 6 – Définition de la fonction de parasitage dans le module.

```
#if defined(CONFIG_TCP_HIDE)
// Parasitage de la pile TCP/IP.
if(NULL != tcphide_parasite_1)
    tcphide_parasite_1(sk, th, skb, tp);
#endif
```

FIG. 7 – Définition de la fonction de parasitage dans le module.

2.2.2 Émetteur A

Dans cette section, nous définissons les structures et fonctions (en pseudo-code) utilisées par le parasitage de paquets sortants de l'émetteur A.

List<Packet>packets_out : Cette liste *ordonnée* contient tous les paquets TCP/IP parasités ayant pour destination B. Nous les conservons afin de *ré-injecter* le même morceau de message caché dans un segment émis plusieurs fois (figure 1 b et c).

FIFO<Msg>outgoing : Cette file *ordonnée* contient les morceaux de messages à transmettre à B.

sender_send() : Lors de l'*envoie* d'un paquet, dans un premier temps nous vérifions que ce paquet doit être parasité (ligne 5), c'est-à-dire s'il fait parti d'une connexion *établie* entre A et B, et que ces drapeaux nous conviennent. Ensuite nous vérifions si ce paquet a déjà été envoyé (ligne 6), si tel est le cas on lui affecte le morceaux de message que lui avait déjà affecté (ligne 7), sinon on lui affecte un morceau de message *potentiel* à transmettre (lignes 9 à 12).

sender_recv() : Lors de la *réception* d'un paquet, nous supprimons tous les éléments de la liste **packets_out** que ce paquet reçu acquitte (lignes 15 à 19). En effet, nous sommes maintenant certains qu'ils ne seront plus jamais ré-émis.

```
1  List<Packet>packets_out
2  FIFO<Msg>outgoing
3
4  sender_send(pkt)
5          if(true == parasitable(pkt))
6                  if(true == belong_to(packets_out, pkt))
7                          set_msg(pkt, get_msg(get(packets_out, pkt)))
8                  else
9                          msg = get(outgoing)
10                         if(undef != msg)
11                                 set_msg(pkt, msg)
12                                 put_sorted(packets_out, pkt)
13
14 sender_recv(pkt)
15         elt = packets_out.head
16         while(undef != elt && elt.ack_seq < pkt.seq)
17                 next = elt.next
18                 del(packets_out, elt)
19                 elt = next
```

FIG. 8 – Émission de données cachées.

2.2.3 Récepteur B

Dans cette section, nous définissons les structures et fonctions (en pseudo-code) utilisées par le parasitage de paquets entrants du récepteur B.

List<Packet>packets_in : Cette liste *ordonnée* contient tous les paquets TCP/IP parasités en provenance de A Nous les conservons afin de les remettre dans l'ordre et de garantir leur unicité (figure 1 c et d). **incoming**.

FIFO<Msg>incoming : Cette file *ordonnée* contient les morceaux de messages reçus de A.

receiver_recv() : Lors de la *réception* d'un paquet, si celui-ci à été parasité (mêmes conditions qu'un paquet parasitable) (ligne 24), nous le mettons dans la liste **packets_in** (ligne 25).

receiver_send() : Lors de l'*envoie* d'un paquet, nous supprimons tous les éléments de la liste **packets_in** en sauvant les morceaux de message reçu dans **incoming** que ce paquet émis acquitte (lignes 29 à 34). En effet, nous sommes maintenant certains que l'on ne recevra plus jamais de paquet précédents ceux que l'on vient d'acquitter.

```
20 List<Packet>packets_in
21 FIFO<Msg>incoming
22
23 receiver_recv(pkt)
24         if(true == parasited(pkt))
25                 put_unic_sorted(packets_in, pkt)
26
27 receiver_send(pkt)
28         elt = packets_in.head
29         while(undef != elt && elt.seq < pkt.seq_ack)
30                 msg = get_msg(elt)
31                 put(incoming, msg)
32                 next = elt.next
33                 del(packets_in, elt)
34                 elt = next
```

FIG. 9 – Réception de données cachées.

2.2.4 Exemple de capture

Nous présentons dans cette section une capture de trois segments TCP (non parasités) que nous avons effectué lors du seul moment où nous avions à notre disposition deux machines réellement en réseau.

La figure 10 représente ce que l'on capture à l'émission sur la machine émettrice, et la figure 11 nous montre ce qu'à reçu la machine réceptrice. Afin de ne pas surcharger, nous n'avons pas inclus les paquets émis par le récepteur et reçu par l'émetteur.

Comme on peut le constater, nous somme dans la même situation qu'à figure 1 *b*, où il y a perte de segments.

2.3 Problèmes et difficultés

2.3.1 Codage des données

Le premier problème que nous avons rencontré venait du codage des données. En effet, nous n'avons pas fait attention que nos PCs stockent [10] les octets à la manière des »petit-boutiste« alors que les octets qui circulent sur le réseau sont stockés à la manière des »gros-boutiste« . Nous avons avons trouvé dans le noyau des macros permettant de remettre dans l'ordre nos données.

2.3.2 Fast Path

Nous avons eu aussi des problèmes avec le flot d'instructions du noyau Linux [6]. En effet, il existe une technique nommée « Fast Path » qui consiste à optimiser les cas les plus fréquents. Par exemple, dans la pile TCP/IP il existe des branchements pour tester les drapeaux des paquets afin d'accélérer le transfert des données entre le réseau et l'application. Il en résulte, qu'au début, certains segments nous échappaient, ce problème a été résolu en étudiant minutieusement le flot d'instuctions du noyau Linux.

2.3.3 Tests

Nous nous sommes heurtés à des difficultés techniques pour effectuer des tests, du fait que nous n'avons pas à notre disposition plusieurs machines en réseaux (et l'interface *loopback* n'est pas utilisable). Pour y remédier, nous avons utilisé un module du noyau *simulant* deux réseaux distincts [13]. Cependant il s'est avéré que cette interface ne répondait pas à nos attente, en effet après comparaison avec un vrai réseau, son comportement n'était pas fidèle. En conséquences, nous n'avons pas pu finaliser notre travail…

```
Outgoing paquet :
ip source : 0.0.0.0
ip destination : 192.168.0.132
TCP header :
sport : 22
dport : 32773
seq : 4201650433
ack_seq : 298660614
check : 19687
urg_ptr : 0
flags : PSH ACK

Outgoing paquet :
ip source : 0.0.0.0
ip destination : 192.168.0.132
TCP header :
sport : 22
dport : 32773
seq : 4201650497
ack_seq : 298660662
check : 25873
urg_ptr : 0
flags : ACK

Outgoing paquet :
ip source : 0.0.0.0
ip destination : 192.168.0.132
TCP header :
sport : 22
dport : 32773
seq : 4201650497
ack_seq : 298660662
check : 28395
urg_ptr : 0
flags : PSH ACK

Outgoing paquet :
ip source : 0.0.0.0
ip destination : 192.168.0.132
TCP header :
sport : 22
dport : 32773
seq : 4201650545
ack_seq : 298660710
check : 61662
urg_ptr : 0
flags : PSH ACK
```

FIG. 10 – Capture des paquets sortants de l'émetteur.

```
Incoming paquet :
ip source : 192.168.0.133
ip destination : 0.0.0.0
TCP header :
sport : 22
dport : 32773
seq : 4201650433
ack_seq : 298660614
check : 19687
urg_ptr : 0
flags : PSH ACK

Incoming paquet :
ip source : 192.168.0.133
ip destination : 0.0.0.0
TCP header :
sport : 22
dport : 32773
seq : 4201650497
ack_seq : 298660662
check : 25873
urg_ptr : 0
flags : ACK

Incoming paquet :
ip source : 192.168.0.133
ip destination : 0.0.0.0
TCP header :
sport : 22
dport : 32773
seq : 4201650545
ack_seq : 298660710
check : 61662
urg_ptr : 0
flags : PSH ACK
```

FIG. 11 – Capture des paquets entrants du récepteur.

2.3.4 Complexité

Même si nous avions à disposition un article [14] sur le *hacking* de la pile TCP/IP du noyau Linux, nous avons du lire énormément de documentations [2] [6] [13] [3] [4] pour comprendre plus en détails le fonctionnement du noyau Linux.

3 Module

Partie de Fabien.

4 Codage

Partie de David.

5 Concrètement

5.1 Sécurité

Il faut avouer que notre système n'assure en rien la sécurité des données transférer, mais uniquement la discrétion. Avec des outils de capture réseau [7][1], il est facile de remarquer qu'il y a un problème dans l'émission des données...

Il faut tout de même remarquer que si l'on doit transférer des données fortement cryptées (grande clef), il est préférable de ne pas alerter tous ses *voisins* en les diffusant telles quelles, sans en ce sens que notre système doit être utilisé : pour la discrétion.

5.1.1 Améliorations

Nous avons des tas d'améliorations en tête, mais il nous reste à finaliser tout d'abord notre projet. En suite, nous irons toujours plus loin en adoptant un système de cryptage et en recherchant des systèmes de dissimulation encore plus malins.

6 Annexes

6.1 Installation

Dans un premier temps, récupérer le patch et appliquez-le aux sources du noyau :

```
cd /usr/src/
wget http://freethecube.free.fr/TCPhide-2.4.26-2.tar.bz2
tar -xjf TCPhide-2.4.26-2.tar.bz2
cp TCPhide/src/kernel-space/TCPhide-2.4.26-2.patch .
patch -p0< ~/TCPhide-2.4.26-2.patch
cd linux-2.4.26
```

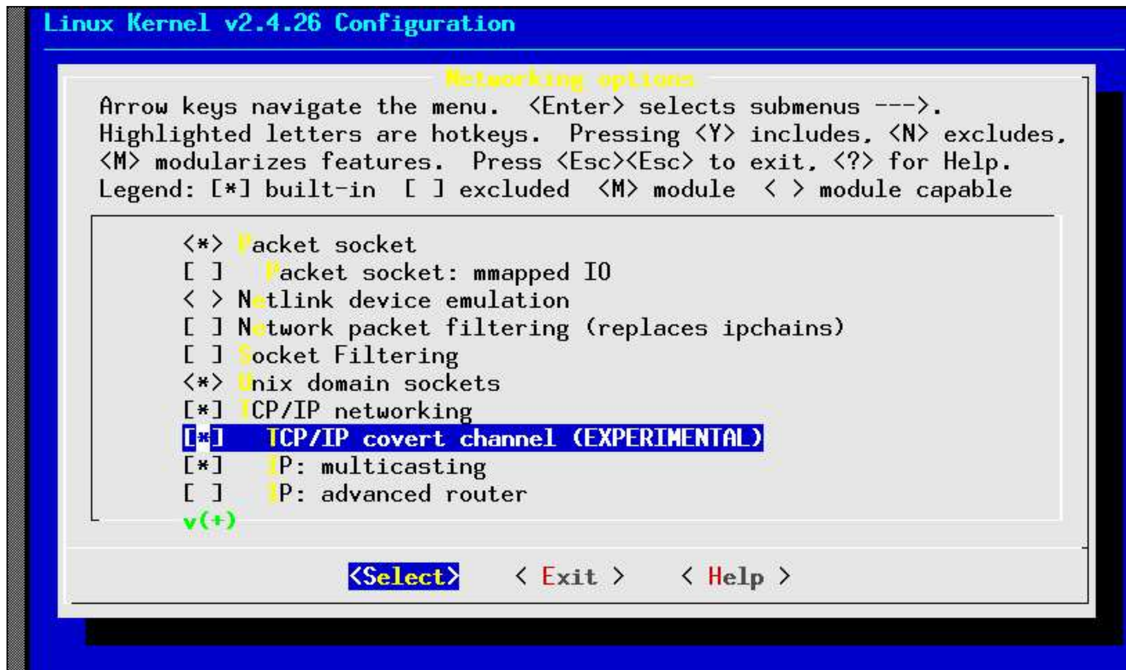
Commencez une nouvelle configuration :

```
make clean
make menuconfig
```

Choisissez les options :

```
Code maturity level options
---> Prompt for development and/or incomplete code/drivers

Networking options
---> TCP/IP covert channel (EXPERIMENTAL)
```



Compilez et installez ce nouveau noyau, ainsi que ses modules :

```
make dep
make bzImage
make install
make modules modules_install
```

Puis relancez la machine. . .

Enfin, compilez le module tcphide et chargez le en mémoire :

```
cd TCPhide/src/kernel-space
make
make load
```

6.2 Utilisation

Lorsque l'on aura fusionné tous nos codes, l'émetteur n'aura qu'à faire :

```
cat message.txt >/dev/tcphide
```

et le récepteur :

```
cat /dev/tcphide > message.txt
```

Références

[1] *TCPDump man page*. http ://www.tcpdump.org/tcpdump_man.html.

[2] Concepts fondamentaux et structure du noyau linux. *GNU/Linux Magazine hors-série*, (16), 09/10 2003.

[3] Introduction à la programmation noyau. *GNU/Linux Magazine hors-série*, (16), 09/10 2003.

[4] Le noyau et le réseau : comment repousser les limites de la connectivité. *GNU/Linux Magazine hors-série*, (17), 11/12 2003.

[5] Kamran Ahsan. Covert channel analysis and data hiding in tcp/ip. Master's thesis, University of Toronto, 2002.

[6] Daniel P. Bovet and Marco Cesati. *Le noyau Linux*. O'Reilly, 07 2001.

[7] Ulisses Alonso Camaró. *Capture network traffic using PACKET MMAP*. http ://pusa.uv.es/ ulisses/packet_mmap/.

[8] Simon Castro. Covert channel tunneling tool. Technical report, GRAY-WORLD.NET, 09 2003.

[9] Defense Advanced Research Projects Agency Information Processing Techniques Office and Information Sciences Institute University of Southern California, http ://www.faqs.org/rfcs/rfc793.html. *RFC 793 - Transmission Control Protocol*, 09 1981.

[10] Greg Kroah-Hartman. Writing portable device drivers. *Embedded Linux Journal Online*, 2002.

[11] Pascal NICOLAS. Cours de réseaux maîtrise d'informatique.

[12] Craig H. Rowland. Covert channels in the tcp/ip protocol suite. *First Monday*, 05 1997.

[13] Alessandro Rubini and Jonathan Corbet. *Pilotes de périphériques sous Linux*. O'Reilly, http ://www.xml.com/ldd/chapter/book/, 01 2002.

[14] Shyamjithe C S. Network protocol stack and tcp hacking. *Linux Gazette*, 02 2004.

[15] Bruce Schneier. *Cryptographie appliquée*. Vuibert, 01 2001.

[16] Sanjeev J. Wagh, Prashant M Yawalkar, and T. R. Sontakke. Eliminating covert channels in tcp/ip using active wardens.