Architecture matérielle

Cyberenquête — Trames Ethernet

TP n° 2

Contexte

Cyberenquête

Le service informatique du lycée surveille en permanence l'activité du réseau afin de détecter d'éventuelles anomalies.
Hier soir, l'administrateur réseau a remarqué un comportement inhabituel : un ordinateur du laboratoire informatique a envoyé plusieurs messages vers un autre poste, mais ces communications ne correspondent à aucun protocole connu utilisé dans l'établissement.
Pour comprendre ce qui s'est passé, les administrateurs ont utilisé un outil d'analyse réseau (appelé sniffer) afin de capturer les trames Ethernet circulant sur le réseau local pendant quelques minutes.
Au total, 200 trames Ethernet ont été enregistrées.
Après une première inspection, ils soupçonnent qu'un utilisateur a tenté de cacher un message secret dans certaines trames réseau, probablement pour transmettre une information discrètement à un autre ordinateur du réseau.
Ne disposant pas du temps nécessaire pour analyser toutes ces données, ils vous confient la mission.
Votre mission
Vous travaillez comme analyste réseau junior au sein de l'équipe informatique du lycée.
Votre objectif est de découvrir si un message caché a réellement été transmis, et si oui, de le reconstituer.
Pour cela, vous devrez :

- Comprendre la structure d'une trame Ethernet
- Analyser les trames capturées
- Identifier les machines qui communiquent
- repérer les trames suspectes
- Extraire les données utiles pour reconstruire le message secret complet

Si votre analyse est correcte, vous devriez être capable de répondre à la question suivante :
Quel message secret a été transmis sur le réseau du lycée ?

Structure d'une trame Ethernet

Encapsulation des données sur le réseau

Lorsqu'un ordinateur envoie des données, celles-ci sont **encapsulées** en couches successives. Voici la structure réelle d'une trame Ethernet transportant un paquet IP.

En-tête Ethernet	Données (Payload)	FCS
14 octets	Paquet IP — 46 à 1500 octets	4 octets

↓ *contient*

MAC dest.	MAC src.	EtherType
6 octets	6 octets	2 octets

↓ *contient un paquet IP*

En-tête IP	Données
$\geq$ 20 octets	variable

↓ *l'en-tête IP contient*

Autres champs	IP source	IP destination
12 octets	4 octets (32 bits)	4 octets (32 bits)

Détail de « Autres champs » :

- **Version + IHL** (1 oct.) : version du protocole et longueur de l'en-tête
- **ToS / DSCP** (1 oct.) : qualité de service (priorité des paquets)
- **Longueur totale** (2 oct.) : taille du paquet IP complet
- **Identification + Flags + Fragment offset** (4 oct.) : gestion de la fragmentation
- **TTL** — Time To Live (1 oct.) : nombre maximal de routeurs pouvant être traversés
- **Protocole** (1 oct.) : protocole encapsulé (6 = TCP, 17 = UDP, 1 = ICMP…)
- **Checksum** (2 oct.) : détection d'erreurs dans l'en-tête IP

Conclusion : les adresses IP ne sont *pas* dans la trame Ethernet elle-même — elles se trouvent dans l'**en-tête IP**, encapsulé dans le payload. L'EtherType `0x0800` signale que le payload contient un paquet IPv4.

Modèle simplifié pour ce TP

Pour ce TP, on utilise un **modèle pédagogique aplati** qui réunit en-tête Ethernet et en-tête IP en une seule structure de **184 bits** :

MAC dest.	MAC source	EtherType	IP source	IP dest.	Data
48 bits, bits 0–47	48 bits, bits 48–95	16 bits, bits 96–111	32 bits, bits 112–143	32 bits, bits 144–175	8 bits, bits 176–183

Le champ **Data** contient ici un seul caractère ASCII (8 bits), là où une vraie trame pourrait transporter jusqu'à 1 500 octets.

Exemple de lecture d'une trame

Voici une trame complète de 184 bits, avec l'identification de chaque champ :

```
1010 1010 1011 1011 1100 1100 1101 1101 1110 1110 1111 1111  ← MAC destination (bits 0-47)
0001 0001 0010 0010 0011 0011 0100 0100 0101 0101 0110 0110  ← MAC source (bits 48-95)
0000 1000 0000 0000                                          ← EtherType (bits 96-111)
1100 0000 1010 1000 0000 0001 0000 1111                      ← IP source (bits 112-143)
1100 0000 1010 1000 0000 0001 0000 1010                      ← IP destination (bits 144-175
0100 1000                                                    ← Données (bits 176-183)
```

IP source (bits 112–143) :
1100 0000 = 192
1010 1000 = 168
0000 0001 = 1
0000 1111 = 15
→ `192.168.1.15`

Données (bits 176–183) :
0100 1000 = $64 + 8 = 72$
→ `chr(72)` = caractère **H**

Partie 1 — Analyse de trames (sans ordinateur)

Rappel : conversion binaire → décimal → ASCII

Un **octet** = 8 bits. Pour convertir en décimal, on multiplie chaque bit par la puissance de 2 correspondante (de droite à gauche : $2^0, 2^1, \ldots, 2^7$).

Exemple : `0100 0001` = $0 + 64 + 0 + 0 + 0 + 0 + 0 + 1 = 65$ → caractère **A**

Question 1 — Analyser la trame interceptée

Les administrateurs réseau ont intercepté la trame suivante. Elle provient peut-être de l'ordinateur suspect…

```
1010 1010 1011 1011 1100 1100 1101 1101 1110 1110 1111 1111 1101 1110 1010
1101 1011 1110 1110 1111 1100 1010 1111 1110 0000 1000 0000 0000 1100 0000
1010 1000 0011 1000 0010 1010 1100 0000 1010 1000 0011 1000 0000 0001 0100
0010
```

Question 1. Quelle est l'adresse IP source ?
(convertir chaque groupe de 8 bits en décimal)

. . . . . .

. . . . . .

Question 2. Quelle est l'adresse IP destination ?

. . . . . .

. . . . . .

Question 3. Quel est le caractère transmis dans les données ?
(convertir `0100 0010` en décimal, puis trouver le caractère ASCII)

. . . . . .

. . . . . .

Question 4. L'IP source trouvée correspond-elle à l'ordinateur suspect (`192.168.56.42`) ? Que peut-on en conclure ?

. . . . . .

. . . . . .

En Partie 2, vous écrirez les fonctions `get_source()` et `get_data()` pour automatiser cette lecture.

Partie 2 — Fonctions Python d'analyse

Principe

Dans cette partie, vous allez écrire des fonctions Python permettant d'analyser automatiquement une trame stockée sous forme d'une chaîne de bits.

Exemple

```
trame = "10101010101111011111001100..." # chaine de 184 caractères '0' ou
↪  '1'
```

Question 2 — `get_source(trame)`

Écrire une fonction `get_source(trame)` qui extrait l'adresse IP source (bits 112 à 143), la découpe en 4 octets et renvoie l'adresse sous forme de chaîne.

Indice : `int("01001000", 2)` renvoie `72`.

Exemples

```
>>> get_source(trame)
'192.168.1.15'
>>> get_source(trame_suspect)    # trame analysée en Partie 1
'192.168.56.42'
```

Réponse

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Question 3 — `get_destinataire(trame)`

Écrire une fonction **`get_destinataire(trame)`** qui extrait l'adresse IP destination (bits 144 à 175) et renvoie l'adresse sous forme de chaîne.

Exemple

```
>>> get_destinataire(trame)
'192.168.1.10'
```

Réponse

.....

.....

.....

.....

.....

.....

.....

Question 4 — `get_data(trame)`

Écrire une fonction **`get_data(trame)`** qui extrait les 8 derniers bits de la trame, les convertit en entier, puis renvoie le caractère ASCII correspondant.

Exemple

```
>>> get_data(trame)
'H'
```

Indice : **`chr(n)`** renvoie le caractère dont le code ASCII est **`n`**.

Réponse

.....

.....

.....

.....

.....

Partie 3 — Analyse du trafic réseau

Situation

Vous disposez d'une liste `trames` contenant **200 trames** capturées sur le réseau du lycée.

L'ordinateur suspect possède l'adresse IP : `192.168.56.42`

Cet ordinateur a envoyé un message **caractère par caractère**, en dissimulant chaque caractère dans les données d'une trame distincte.

Question 5 — Retrouver le message secret

Écrire un programme qui :

1. parcourt toutes les trames de la liste `trames` ;
2. sélectionne celles dont l'IP source est `192.168.56.42` ;
3. extrait le caractère transmis avec `get_data()` ;
4. affiche le message complet.

Résultat attendu

```
MESSAGE SECRET : ............
```

Réponse

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Question 6 — Question de réflexion

Pourquoi un message est-il découpé en plusieurs trames sur un réseau ?

. . . . . .

. . . . . .

. . . . . .

*

Annexe — Table de correspondance ASCII

Lire la table

Chaque caractère possède un **code décimal** et une **représentation binaire** sur 8 bits.
Exemple : le caractère **B** a le code décimal **66**, soit `01000010` en binaire.
En Python : `ord('B')` → `66` et `chr(66)` → `'B'`

Lettres majuscules

Déc	Binaire	Car	Déc	Binaire	Car
65	01000001	A	78	01001110	N
66	01000010	B	79	01001111	O
67	01000011	C	80	01010000	P
68	01000100	D	81	01010001	Q
69	01000101	E	82	01010010	R
70	01000110	F	83	01010011	S
71	01000111	G	84	01010100	T
72	01001000	H	85	01010101	U
73	01001001	I	86	01010110	V
74	01001010	J	87	01010111	W
75	01001011	K	88	01011000	X
76	01001100	L	89	01011001	Y
77	01001101	M	90	01011010	Z

Lettres minuscules

Déc	Binaire	Car	Déc	Binaire	Car
97	01100001	a	110	01101110	n
98	01100010	b	111	01101111	o
99	01100011	c	112	01110000	p
100	01100100	d	113	01110001	q
101	01100101	e	114	01110010	r
102	01100110	f	115	01110011	s
103	01100111	g	116	01110100	t
104	01101000	h	117	01110101	u
105	01101001	i	118	01110110	v
106	01101010	j	119	01110111	w
107	01101011	k	120	01111000	x
108	01101100	l	121	01111001	y
109	01101101	m	122	01111010	z

Chiffres et espace

Déc	Binaire	Car	Déc	Binaire	Car
32	00100000	(espace)	49	00110001	1
48	00110000	0	50	00110010	2
51	00110011	3	52	00110100	4
53	00110101	5	54	00110110	6
55	00110111	7	56	00111000	8
57	00111001	9			