Write-up du challenge forensic "Sur écoute" - MidnightCTF 2022

Auteur : Abyss Watcher

Sous-titre : Exfiltration de données

Description : Un camarade et vous-même avez réussi à vous infiltrer sur le réseau d'une organisation ennemie au parti. Votre acolyte a gagné l'accès sur le serveur central, et a pu récupérer des données sensibles sur ce dernier. Vous, n'avez réussi à compromettre que le commutateur, et avez lancé une capture du traffic.
Il était convenu d'échanger les données via une messagerie sécurisée, mais celle-ici a été piratée entre temps ! Vous ne pouvez plus communiquer avec votre camarade, et avez besoin de ces informations au plus vite. Tout ce dont vous disposez est votre capture, et l'appel d'un mystérieux inconnu s'étant trompé de numéro. Vous avez néanmoins relevé ces mots, qui résonnent de manière étrange :
"Il Conviendrait Maintenant Pour Moi De Chasser Ce Numéro De Mon Carnet, Et Sans Perdre De Temps !"

PS : MCTF{flag1flag2}

Première analyse de la capture fournie

Il nous est donné ici une capture au format "pcapng". Nous nous empressons de l'ouvrir avec Wireshark, afin de l'analyser. En premier lieu, nous regardons le nombre de paquets capturés : 1221. Il va donc falloir agir méthodiquement, afin d'éviter de s'éparpiller et de scroller à la main indéfiniment.
Une bonne technique est d'ouvrir directement le menu "Statistics" puis de cliquer sur "Protocol Hierarchy" :

Protocol	Percent Packets	Packets	Percent Bytes
Frame	100.0	1221	100.0
Ethernet	100.0	1221	4.1
Internet Protocol Version 4	88.5	1081	5.2
Transmission Control Protocol	62.7	765	77.2
Transport Layer Security	20.7	253	50.7
Hypertext Transfer Protocol	1.6	20	30.4
Online Certificate Status Protocol	1.0	12	1.1
Line-based text data	0.2	2	0.1
Media Type	0.1	1	15.7
JPEG File Interchange Format	0.1	1	11.9
User Datagram Protocol	20.1	245	0.5
Domain Name System	7.7	94	1.8
Data	6.7	82	5.0
Simple Service Discovery Protocol	1.7	21	0.8
Network Time Protocol	1.4	17	0.2
Routing Information Protocol	1.1	13	0.1
Dynamic Host Configuration Protocol	0.7	8	0.6
Syslog message	0.4	5	0.1
Multicast Domain Name System	0.3	4	0.1
Link-local Multicast Name Resolution	0.1	1	0.0
Internet Control Message Protocol	5.4	66	0.3
Internet Group Management Protocol	0.4	5	0.0
Address Resolution Protocol	9.4	115	1.2
Internet Protocol Version 6	2.0	24	0.2
User Datagram Protocol	1.6	19	0.0
Data	1.1	14	2.2
Multicast Domain Name System	0.3	4	0.1
Link-local Multicast Name Resolution	0.1	1	0.0
Internet Control Message Protocol v6	0.4	5	0.0
Logical-Link Control	0.1	1	0.0
Cisco Discovery Protocol	0.1	1	0.0

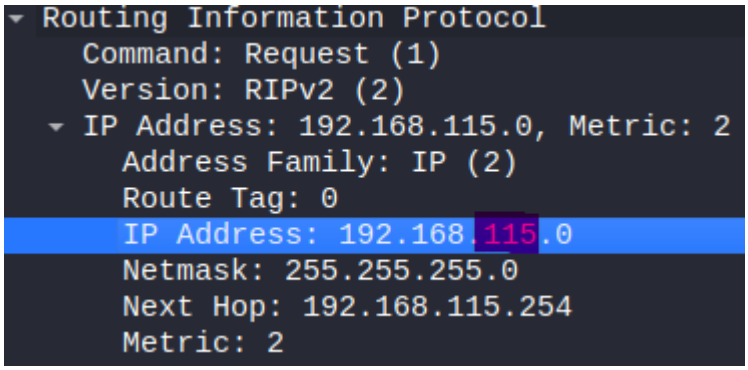
De part l'énoncé du challenge, nous savons que les données ont été exfiltrées de manière "discrète". Ce type de technique nécessite la plupart du temps de nombreux paquets, de part le peu d'information envoyé à chaque fois.
Néanmoins, notre oeil est attiré par la présence du protocole HTTP. Il est possible d'accéder, soit en filtrant, soit en allant dans "File->Export Objects->HTTP" aux fichiers présents dans la capture. Ce sont malheureusement des fausses pistes.

Il convient désormais de filtrer méthodiquement chaque protocole pouvant contenir des données cachées (NTP, DNS, RIP etc.), en regardant brièvement d'un paquet à l'autre si des champs changent de manière récurrente.

Nous passerons ici sur cette démarche d'analyse, qui, faite sans chercher la complexité, permet de tomber sur un comportement étrange des paquets RIPv2.

Analyse des paquets RIPv2 (seconde partie du flag)

Même sans connaître le fonctionnement de ce protocole, nous pouvons remarquer qu'une valeur change drastiquement d'un paquet à l'autre, au niveau de l'adresse de réseau :



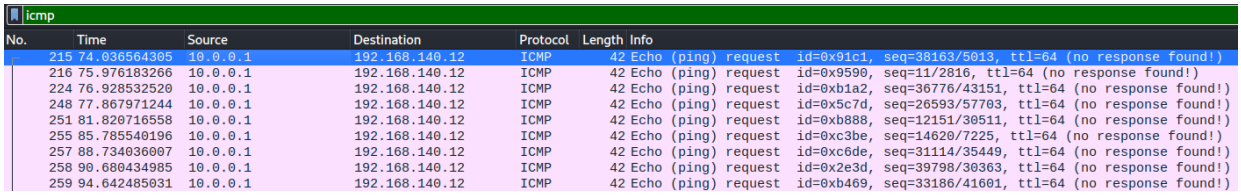
En utilisant le site https://www.rapidtables.com/convert/number/ascii-hex-bin-dec-converter.html, nous pouvons tester si cette valeur décimale donnerait un caractère imprimable intéressant (lorsque l'on ne connaît pas la table ASCII par coeur). Nous obtenons la lettre "s". Il suffit ensuite de tester le paquet d'avant et d'après, pour se rendre compte que cela concorde, et forme sans aucun doute un message une fois tous les paquets RIPv2 "décodés".
Il est possible de le faire à la main, cela n'est pas très long, ou de coder un petit script (python ici) :

```
import pyshark

filename = 'capture.pcapng'

# On filtre uniquement les paquets RIP, puis on convertit la troisième
# partie de l'adresse en son caractère ASCII correspondant.
with pyshark.FileCapture(filename, display_filter=('rip')) as packets:
    flag = ''
    packets.load_packets()
    for packet in packets:
        flag += chr(int(packet.rip.ip.split('.')[2]))
print(flag)
```

Nous obtenons : "s1ckl3_s3c0nd". Cela est la seconde partie du flag, il ne reste plus qu'à trouver la première !

Recherche de la première partie du flag

Nous pouvons réfléchir au sens de la phrase "Il Conviendrait Maintenant Pour Moi De Chasser Ce Numéro De Mon Carnet, Et Sans Perdre De Temps !". Sans chercher compliqué, nous pouvons relever le mot "temps", qui est probablement un message caché pour nous indiquer de faire attention à l'horodatage des paquets.
En cherchant un peu compliqué (mais pas trop), on remarque "**Il** **C**onviendrait **M**aintenant **P**our" -> ICMP. Les données seraient donc cachées dans les paquets ICMP, au niveau temporel :



Ces paquets sont par ailleurs particulièrement suspects, puisque notre camarade a oublié d'activer la réponse au ping sur son serveur. Nous remarquons que l'écart entre les paquets est irrégulier,

mais que certaines valeurs de delta (écart) reviennent souvent (1,2,3 et 4 secondes). Il est possible qu'un delta soit égal à un équivalent binaire ! 4 secondes, cela équivaut à 4 possibilités : (00,01,10,11).
Voici le script permettant d'extraire la première partie du flag :

```
import pyshark
import math
import itertools
import collections
import re

filename = 'capture.pcapng'

# On filtre uniquement les paquets ICMP de type requête, provenant du
# serveur central, puis on calcule l'écart de temps (delta) entre deux
# paquets.
with pyshark.FileCapture(filename, display_filter=('icmp.type==8 and
ip.src==10.0.0.1')) as packets:
    delta_extract = []
    packets.load_packets()
    for i in range(len(packets) - 1):
        delta_extract.append(math.ceil(
            float(packets[i + 1].sniff_timestamp) -
float(packets[i].sniff_timestamp)))

# Ici, on peut déjà afficher les deltas. On peut remarquer que seulement 4
# sont utilisés. Cela signifie : '00','01','10','11'.
print(f'Deltas extraits : {delta_extract}')

# On génère toutes les combinaisons possibles vis-à-vis des deltas
# extraits (#1), puis les valeurs de bits possibles (#2) avant de créer
# toutes les permutations possibles entre les deltas et les bits (#3).
sorted = sorted(collections.Counter(delta_extract))  # 1
binary_combinations = [
    ''.join(x) for x in itertools.product('01', repeat=2)]  # 2
binary_delta = [list(zip(x, sorted))  # 3
                for x in itertools.permutations(binary_combinations,
len(sorted))]

# Ici, on va désormais remplacer chaque delta (nombre entier) par une
# valeur correspondante dans le tableau des bits ('00','01','10','11').
for permutation in binary_delta:
    intermediate_delta_extract = delta_extract
    for binary, delta in permutation:
        intermediate_delta_extract = [
            binary if x == delta else x for x in
```

```
intermediate_delta_extract]

    # Une fois cela fait, il ne reste plus qu'à extraire chaque paquet de 8
bits de la suite de bits, avant de les décoder.
    # Si le décodage comprend des caractères hors de l'UTF-8, il est peu
    # probable que ce soit le flag. On affiche alors uniquement les
décodages
    # UTF-8 valides.
    extracted = ''
    for binary in re.findall('[01]{8}',
''.join(intermediate_delta_extract)):
        extracted += format(int(binary, 2), '02x')
    try:
        print(
            f'Found printable string {bytes.fromhex(extracted).decode("utf-
8").encode()} with permutation {permutation} !')
    except:
        continue
```

Sortie :

```
Deltas extraits : [2, 1, 1, 4, 4, 3, 2, 4, 2, 1, 3, 2, 2, 1, 3, 2, 4, 3, 4,
3, 2, 3, 4, 1, 2, 2, 3, 3, 2, 1, 2, 1, 4, 3, 4, 2, 2, 3, 4, 1, 2, 3, 4, 3,
2, 3, 2, 4, 2, 2, 3, 3, 4]
Found printable string b')q,,w6\x0f"t671\x0f' with permutation [('10', 1),
('00', 2), ('11', 3), ('01', 4)] !
Found printable string b'h4mm3r_f1rst_' with permutation [('10', 1), ('01',
2), ('11', 3), ('00', 4)] !
Found printable string b"=a88f'\n3d'&!\n" with permutation [('11', 1),
('00', 2), ('10', 3), ('01', 4)] !
Found printable string b'|$yy"cZw!cbdZ' with permutation [('11', 1), ('01',
2), ('10', 3), ('00', 4)] !
```

Nous remarquons la chaîne "h4mm3r_f1rst", qui correspond à la première partie du flag !

Nous assemblons les deux parties, en ajoutant le formatage du MidnightCTF :

MCTF{h4mm3r_f1rst_s1ckl3_s3c0nd}

Nous espérons que ce challenge vous aura plu !