

Sneaky

Lets start by firing up wireshark. First thing we notice is that there are around 500 packets here to deal with

Upon checking the `protocol hierarchy` we can see that we are dealing with the following protocols:

- DNS
- HTTP
- HTTPS
- ICMP
- SMB

Since everything is soo "random" and its getting hard to analyse anything, we sort everything by `Protocol` to get a clear view of what sort of traffic is in the pcap.

Trolls

Casually sifting through the traffic, we some files like `flag.txt` and `tryharder.png`, which seem to be just trolls at this stage as the text inside flag.txt just says `TryHarder` and nothing hidden inside the image. Just to make it a little more interesting.

Sneaky hidden stuff

Hidden DNS Data

Since most of the traffic is encrypted so for the time being it would only be a complete waste of time to look through it. We start by looking through DNS traffic to look for any abnormal looking queries.

We notice something weird in one of the dns queries:

The image shows a Wireshark packet capture interface. At the top, a packet list shows three packets: packet 326 (DNS), packet 336 (DNS), and packet 337 (ICMP). Packet 336 is selected, and its details pane shows it is a Standard query response from 192.168.145.35 to 192.168.145.30. The packet length is 598 bytes. Below the details pane, the raw packet data is shown in hexadecimal and ASCII. The ASCII portion contains a string that appears to be a base64-encoded message: "#5.5.4.i.....emote.UEsDBBQDAQ AAABJHJ1EqnDRpPw AAADMAAAAEAAAAZm x.hZy6-Y".

No.	Time	Source	Destination	Protocol	Length	Info		
326	0.000161	2.704004	192.168.145.10	192.168.145.35	DNS	53	52902	140 Standard query response 0x0500 AAAA googleads.g.doubleclick.net CNAME
336	0.968670	3.672674	192.168.145.30	192.168.145.35	DNS	53	53	598 Standard query 0x0000 Unknown (13869) <Unknown extended label>
337	0.002970	3.675644	192.168.145.30	192.168.145.35	ICMP	53	53	590 Destination unreachable (Port unreachable)

Frame 336: 598 bytes on wire (4784 bits), 598 bytes captured (4784 bits) on interface 0
Ethernet II, Src: HonHaiPr_11:9d:6b (a8:6b:ad:11:9d:6b), Dst: Vmware_d5:ee:c5 (00:0c:29:d5:ee:c5)
Internet Protocol Version 4, Src: 192.168.145.30, Dst: 192.168.145.35
User Datagram Protocol, Src Port: 53, Dst Port: 53
Domain Name System (query)

```
00 00 0c 29 d5 ee c5 a8 6b ad 11 9d 6b 08 00 45 00 ..)....k...k..E-
10 02 48 00 01 00 00 40 11 d5 11 c0 a8 91 1e c0 a8 .H....@.....
20 91 23 00 35 00 35 02 34 ef 69 00 00 00 00 00 01 .#5.5.4.i.....
30 00 00 00 00 00 06 72 65 6d 6f 74 65 2e 55 45 .....r emote.UE
40 73 44 42 42 51 44 41 51 41 41 41 42 4a 48 4a 31 sDBBQDAQ AAABJHJ1
50 45 71 6e 44 52 70 50 77 41 41 41 44 4d 41 41 41 EqnDRpPw AAADMAAA
60 41 45 41 41 41 41 5a 6d 78 2e 68 5a 79 36 2d 59 AEAAAAZm x.hZy6-Y
```

39	77	53	39	ID-XQ4_t	HcFc9wS9
53	4f	39	63	NpJNG7BD	jF14S09c
57	61	77	51	ar2-L.nZ	-0nggawQ
78	70	6a	75	mgGKcmh_	tXaWxpju
59	31	39	72	VDVCmgKY	VDGki19r
44	41	51	41	9.VBLAQI	_AxQDAQA
70	50	77	41	AABJHJ1E	qnDRpPwA
41	2e	41	41	AADMAAAA	EACQA.AA
41	41	41	41	AAAAAAII	CkgQAAAA
41	41	41	41	BmbGFnCg	AgAAAAAA
75	53	45	31	ABABgAgN	N.4BuSE1
59	44	54	65	gEAl0II5	ITWAYDTe
57	41	41	41	AbkhNYBU	EsFBgAAA
41	47	45	41	AABAA.EA	VgAAAGEA
79	6c	65	67	AAAAAA.r	eallyleg
2e	2e	2e	2e	it.com..
2e	2e	2e	2e

That looks fishy, a query with a domain starting with `remote...` and ending with `...reallylegit.com` with some weird encoded data. Now we simply copy this payload from here and put it into a file for further analysis.

```
remote.UEsDBBQDAQAAABJHJ1EqnDRpPwAAADMAAAAEAAAAZmx.hZy6-YID-XQ4_t
HcFc9wS9NpJNG7BDjF14S09car2-L.nZ-0nggawQmgGKcmh_tXaWxpjuVDVCmgKYV
DGki19r9.VBLAQI_AxQDAQAAABJHJ1EqnDRpPwAAADMAAAAEACQA.AAAAAAAAIICK
gQAAAABmbGFnCgAgAAAAAABABgAgNN.4BuSE1gEAl0II5ITWAYDTeAbkhNYBUESF
BgAAAAABAA.EAVgAAAGEAAAAAAA.reallylegit.com
```

After fiddling around it for a while we notice that this is `Base64-URL` without any padding, so I made a script to decode it the base64

```
import base64

data="UEsDBBQDAQAAABJHJ1EqnDRpPwAAADMAAAAEAAAAZmx.hZy6-YID-XQ4_t
HcFc9wS9NpJNG7BDjF14S09car2-L.nZ-0nggawQmgGKcmh_tXaWxpjuVDVCmgKYV
DGki19r9.VBLAQI_AxQDAQAAABJHJ1EqnDRpPwAAADMAAAAEACQA.AAAAAAAAIICK
gQAAAABmbGFnCgAgAAAAAABABgAgNN.4BuSE1gEAl0II5ITWAYDTeAbkhNYBUESF
BgAAAAABAA.EAVgAAAGEAAAAAAA"

def decode():
    # https://gist.github.com/catwell/3046205
    _data = data.replace('.', '_').replace('-', '/').replace('-', '+')
    padding = len(_data) % 4
    if padding == 2:
        _data += '=='
    elif padding == 3:
        _data += '='
    return base64.urlsafe_b64decode(_data)
```

```

return base64.b64decode(_data)

if __name__ == '__main__':
    decoded_data = decode()
    print(decoded_data)

```

Which outputs some rubbish, but after redirecting the "rubbish" to a file and running a file command on it tells me that its a `zip` file.

```

<[ root@gokuKaioKen 10:19 tess ]>
python decode.py > file
<[ root@gokuKaioKen 10:19 tess ]>
file file
file: Zip archive data, at least v?[0x314] to extract
<[ root@gokuKaioKen 10:19 tess ]>

```

Unfortunately this zip file is protected by a password. We try to bruteforce this using john and rockyou to get the password. To do that we first use `zip2john` to create a hash which we can crack using john.

```

<[ root@gokuKaioKen 10:21 tess ]>
zip2john file.zip
ver 78.8 file.zip/flag PKZIP Encr: cmplen=63, decmplen=51, c
file.zip/flag:$pkzip2$1*1*2*0*3f*33*69349c2a*0*22*0*3f*6934*
429a02985431a48b5f6bf5*$ /pkzip2$:flag:file.zip::file.zip
<[ root@gokuKaioKen 10:21 tess ]>

```

We can see straight away that the password is `narutoshippudengoku`.

```

<[ root@gokuKaioKen 10:22 tess ]>
john hash --wordlist=/usr/share/wordlists/rockyou.txt
Using default input encoding: UTF-8
Loaded 1 password hash (PKZIP [32/64])
Press 'q' or Ctrl-C to abort, almost any other key for status
narutoshippudengoku (file.zip/flag)
ig 0:00:00:01 DONE (2020-09-07 10:22) 0.6451g/s 3351Kp/s 3351Kp/s
Use the "--show" option to display all of the cracked passwords
Session completed
<[ root@gokuKaioKen 10:22 tess ]>

```

unzipping the file we have `flag` in there BUT

The Final Boss

Now comes the brainy part of the challenge which will probably break alot of them !

after decrypting the zip file, we find the `flag` but unfortunately we cannot read it as its `encrypted`(WTF!).

```

decode.py 100.21p flag hash test
<[ root@gokuKaioKen 10:23 tess ]>
file flag
flag: Vim encrypted file data

```

```

flag: Vim encrypted file data
<[ root@gokuKaioKen 10:24 tess ]>
cat flag
VimCrypt~02!-s0grU 90  A  ~X LG

```

If anyone has used `vim` before (which you should), then it should be clear that it's encrypted using one of the algorithms supported by vim. Upon further digging we can find that VimCrypt has 3 encryption modes which are 1 = pkzip, 2 = blowfish and 3 = blowfish2. From the header of the file, we can see `VimCrypt~02!` which suggests that this is `blowfish` encrypted which is another `XOR cipher` encryption. Upon further googling we can find this link (below) which tells us why it's vulnerable and how you can get the plaintext without any key if you have known some part of the plaintext (flag format).

[Vulnerable ViM algorithm](#)

It says that the first 28 bytes are header, made up of the encryption descriptor (12), salt (8) and IV (8), so if you XOR the first block with the plaintext and keep XORing the remaining with the previously XORed block then you can retrieve some part of the plaintext. I made this quick script to retrieve the flag based on this vulnerability.

```

#!/bin/env python

import os
import sys
import itertools

plaintext = bytes("DUCTF{")
print plaintext
blocks = []
keystream = ""

def xor(block, key):
    return ''.join(chr(ord(x) ^ ord(y)) for (x,y) in itertools.izip(block, key))

with open("flag", "rb") as infile:
    header = infile.read(12)
    salt = infile.read(8)
    iv = infile.read(8)

    blocks.append(infile.read(8))

```

```
blocks.append(infile.read(8))
blocks.append(infile.read(8))
blocks.append(infile.read(8))

key = xor(blocks[0],plaintext)

plaintext+=xor(blocks[1],key)
plaintext+=xor(blocks[2],key)
plaintext+=xor(blocks[3],key)

print plaintext
```

Now here we already know that the known part is actually the flag format `DUCTF{` so we run this script with that as the known part.

```
python dec.py
DUCTF{y_H4rD_!EE7}
```

From here we can see that this `H4rD` looks like the word `Hard` and the word before that ends with `y` so we can take an easy guess that it must be the word `Try` so it becomes `Try_Harder`. So now the known part becomes `DUCTF{Tr` and one we run the script again we finally get the flag !

2e
2e
2e
2e
72Tr
2e	yHarder.
2e
2e

```
[root@gokuKaioKen 10:29 temp2 ]# python dec.py
DUCTF{Try_H4rD3R_!EE7}
```

gokuKaioKen