

Cyber Security Coursework - Protocol Analysis

James Donohue - james.donohue@bbc.co.uk

Introduction

This is my report for the 'protocol analysis' part of the Cyber Security coursework assignment.

The third-party tools used in this report were:

- `file(1)` - command-line tool common on UNIX-based systems for identifying file types
- Wireshark v2.2.4 - graphical network protocol analyser
- [Scapy](#) 2.3.2 - packet manipulation library for Python
- [aircrack-ng](#) - WEP password cracking tool

`capture1.pcap`

The `file` tool identifies this capture as follows:

```
tcpdump capture file (little-endian) - version 2.4 (Ethernet, capture length 65535)
```

From this we can learn that the capture was created on a little-endian processor (such as the Intel x86 family). The 'capture length' indicates that captured data was limited to 65535 bytes per packet (Lamping et al. 2014).

Opening the file in Wireshark shows it contains 44 packets. Using the *Statistics > Endpoints* report we can see the packets relate to 6 different Ethernet addresses and 5 IPv4 addresses.

Find the username & password from the HTTP Basic type of authentication

By using the `http` view filter in Wireshark to show only HTTP packets (see Figure 1) we can see two separate HTTP connections are made from host 192.168.0.3 to port 80 on host 192.168.0.1. The first of these connections contains a single request (`GET /`) which results in a `401 Access Denied` response.

The second connection (coloured in purple on Figure 1) contains two requests made using HTTP Basic authentication. The content of these requests can be seen most easily using the **Follow > TCP Stream** feature (see Figure 2).

Basic HTTP authentication, originally specified in HTTP/1.0, allows for simple challenge-response authentication of web clients using the `Authorization` header. The value after the scheme name `Basic` is a Base64 encoding of the username and password, separated by a colon. As no encryption is performed on the password, the Basic scheme is explicitly non-secure (Berners-Lee et al. 1996). In this case the client sends the following header:

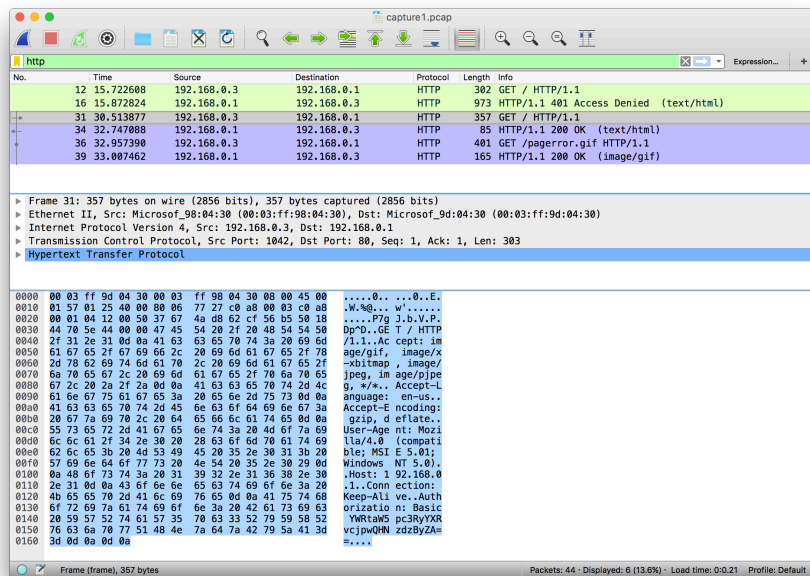


Figure 1: Wireshark filtering for only HTTP packets in capture1.pcap

Authorization: Basic YWRtaW5pc3RyYXRvcjpwQHNzdzByZA==

The server responds with HTTP/1.1 200 OK, indicating that the authentication attempt was successful. The plaintext of the Base64 value can easily be determined using a trivial fragment of Python:

```
>>> import base64
>>> encoded = 'YWRtaW5pc3RyYXRvcjpwQHNzdzByZA=='
>>> base64.b64decode(encoded)
'administrator:p@ssw0rd'
```

Therefore the username sent was administrator and the password was p@ssw0rd.

Which hosts appear to be sending broadcast IP packets?

In IPv4, broadcast packets are those sent to a special address which causes them to be received by all hosts on a given subnet. The special address has all its unmasked bits set to 1 (Mogul 1984), which means ending in a sequence of one or more 255s when printed in typical 'dotted quad' form.

The Wireshark *Statistics > Conversations* view (shown in Figure 3) shows that four hosts are sending packets to 192.168.0.255, which is the broadcast address for the 192.168.0.0/24 network.

The hosts that appear to be sending broadcast IP packets (via UDP) are therefore:

```
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.100
```

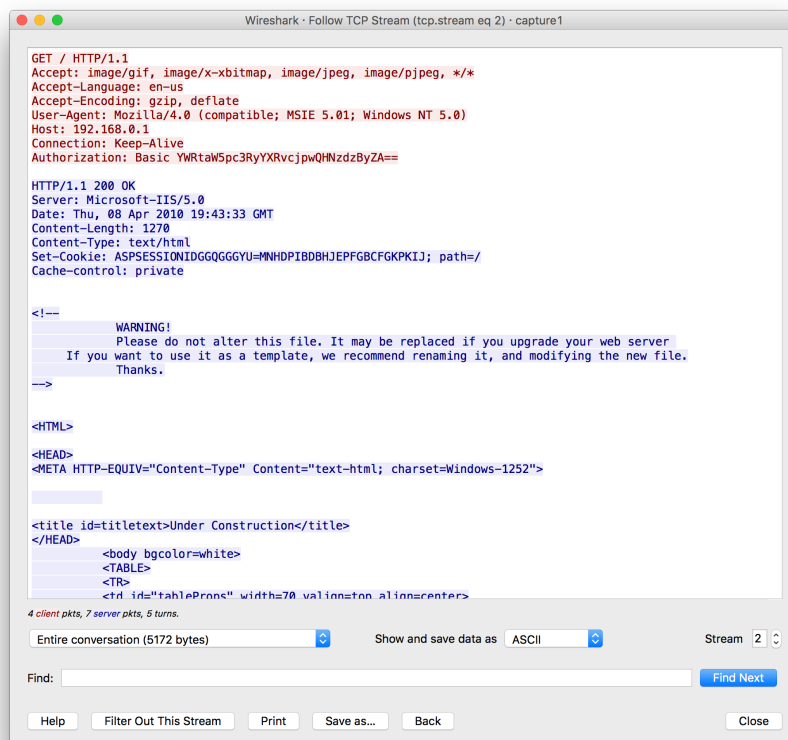


Figure 2: Wireshark Follow TCP Stream showing HTTP Basic authentication in capture1.pcap

The image shows the Wireshark 'Conversations' view for a capture file named 'capture1.pcap'. The 'IPv4' tab is selected, showing a list of conversations between IP addresses. A red box highlights the first five rows, which represent broadcast traffic from 192.168.0.1 to 192.168.0.100, 192.168.0.255, 192.168.0.2, 192.168.0.3, and 192.168.0.100.

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
192.168.0.1	192.168.0.3	24	10 k	12	9025	12	
192.168.0.1	192.168.0.100	7	548	4	298	3	
192.168.0.1	192.168.0.255	3	276	3	276	0	
192.168.0.2	192.168.0.255	2	335	2	335	0	
192.168.0.3	192.168.0.255	3	276	3	276	0	
192.168.0.100	192.168.0.255	1	243	1	243	0	

Figure 3: Wireshark ‘Conversations’ view for capture1.pcap with broadcast traffic highlighted

The packets are UDP datagrams on ports 137 and 138 (NetBIOS), meaning we can infer that these hosts are likely to be running Windows.

capture2.pcap

The Address Resolution Protocol (ARP) (Plummer 1982) is used to determine the hardware (MAC) address of a target host based on its IP address. In normal operation, the host attempting to resolve a MAC address sends a broadcast ‘request’ message containing the IP address that is being queried. This will be received by all network interface cards (NICs) on the same subnet. The target host then sends a ‘reply’ message containing its own MAC address to the host that made the request, which caches it for later reuse.

Because ARP has no built-in support for authenticating such messages, it is possible for a malicious user to use it to disrupt network activity. In ‘ARP poisoning’, an attacker sends fake ARP requests or replies with the intention of either intercepting network traffic between two hosts (man-in-the-middle attack) or producing a denial-of-service (DoS) (Mishra 2016).

Method of detection

ARP poisoning can be suspected when a host begins sending ARP packets containing a ‘sender’ hardware (MAC) address that is different to a MAC address already associated with the given protocol address (IP) in earlier traffic.

Therefore, a simple implementation of an ARP poisoning sensor is to record the sender MAC address associated with the IP address in each ARP packet. When a new ARP packet is received, if a different

sender MAC is given for the IP in address in question than one previously recorded, an alert can be issued and any other required action taken.

In order to demonstrate this approach using the supplied packet capture file, a Scapy script (poisonarp.py) was written. This script implements the above approach by iterating over each packet in the capture and storing a mapping of IP address to MAC address in a dictionary object.

The script can be executed as follows (assuming Scapy has been installed):

```
$ python poisonarp.py cwk_pcaps/capture2.pcap
```

Example script output

```
Read 328 packets from cwk_pcaps/capture2.pcap
WARNING: suspected ARP poisoning attack at packet 298
  Attacker: 00:03:ff:98:98:02 (last seen using 192.168.0.2)
  Victim: 00:00:00:00:00:00 (192.168.0.1)
  Hijacked resource: 00:03:ff:98:98:03 (192.168.0.3)
```

Limitations

To identify the victim, this script uses the ‘target’ hardware and protocol addresses given in the ARP packet, which is sufficient for the packet capture given. However, ARP poisoning can also be performed by broadcasting malicious ARP replies (so-called Gratuitous ARP or GARP). In this scenario the intended victim may not be immediately apparent.

Also, there are some scenarios in which the MAC address for an IP address may change legitimately (for example due to replacement of a part). These events will produce ‘false positives’ in a sensor that uses this approach.

Alternatives: Wireshark

Wireshark also identifies this type of attack using the message ‘Duplicate IP address detected’ (see yellow text in Figure 4).

capture3.pcap

The file tool identifies this capture as follows:

```
capture3.pcap: tcpdump capture file (little-endian) - version 2.4 (802.11, capture length 65535)
```

This reveals that unlike the other capture files (which were captured on an Ethernet network) this capture was created using a wireless (802.11) network.

Wireshark reports that the capture file contains 25895 packets. Using the display filter `wlan.fc.protected == 1` reveals that 59.5% of the packets (100% of data frames) are flagged as protected (see Figure 5). This bit indicates that the packet data is encrypted using an 802.11 encryption mechanism such as WEP (Wired Equivalent Privacy) or TKIP (Orebaugh et al. 2006).

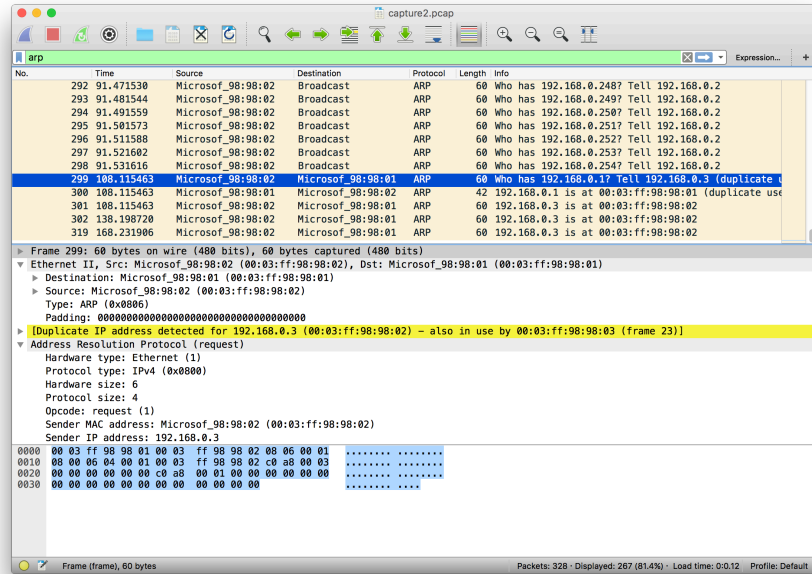


Figure 4: Wireshark showing duplicate IP address usage in capture2.pcap

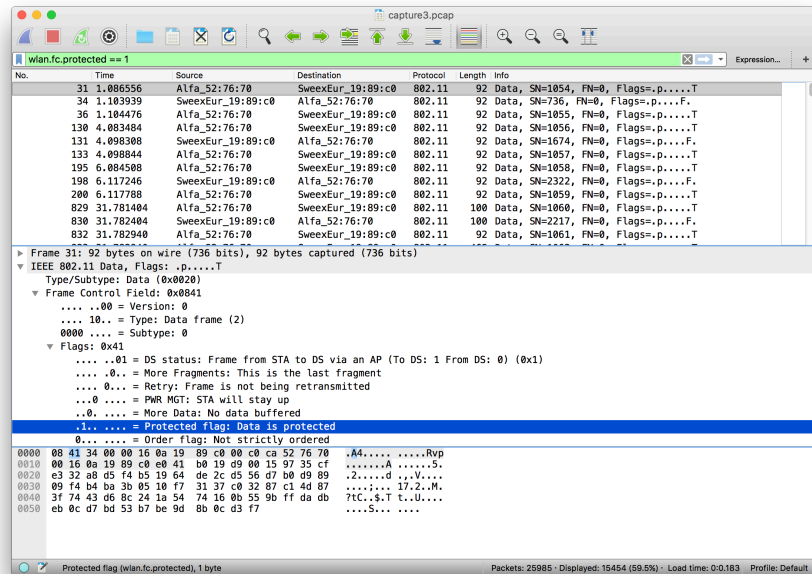


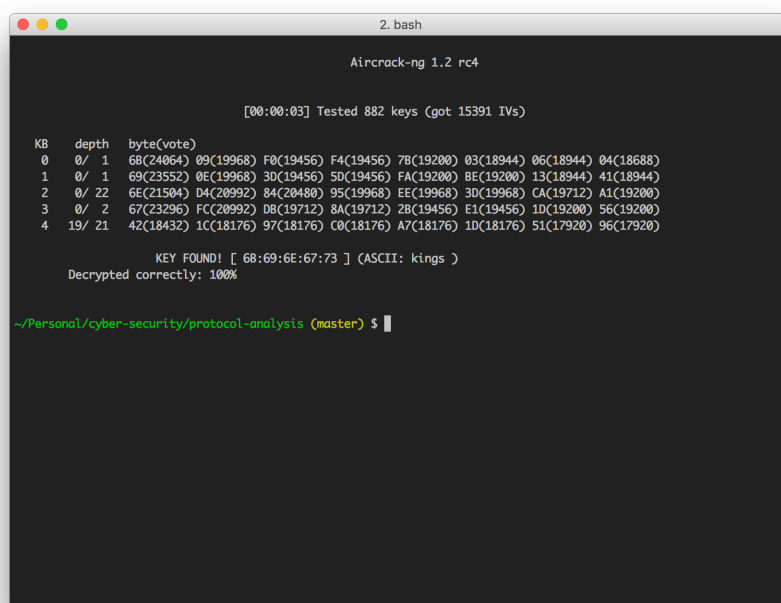
Figure 5: Wireshark showing encrypted Wifi packets in capture3.pcap

This means that it will not be possible to simply use Wireshark's protocol dissection functionality to extract passwords from the capture, as the relevant data is encrypted. However, the display filter `wlan.wep.iv`, which identifies frames that include the WEP Initialization Vector (IV), shows the same list of packets, meaning that all of these packets are encrypted using WEP.

The stream cipher used by WEP has been shown to be insecure and vulnerable to passive attack by an eavesdropper (Fluhrer et al. 2001), given access to sufficient IVs. The tool `aircrack-ng` can be used to crack WEP encryption using a packet capture file as follows:

```
$ aircrack-ng cwk_pcaps/capture3.pcap
```

This successfully extracts the WEP key `6B:69:6E:67:73` ("kings") from the capture file as shown in figure 6.



```
2. bash
Aircrack-ng 1.2 rc4

[00:00:03] Tested 882 keys (got 15391 IVs)

KB  depth  byte(vote)
0   0/ 1    68(24064) 09(19968) F0(19456) F4(19456) 7B(19200) 03(18944) 06(18944) 04(18688)
1   0/ 1    69(23552) 0E(19968) 3D(19456) 5D(19456) FA(19200) BE(19200) 13(18944) 41(18944)
2   0/ 22    6E(21504) D4(20992) 84(20480) 95(19968) EE(19968) 3D(19968) CA(19712) A1(19200)
3   0/ 2    67(23296) FC(20992) 08(19712) 8A(19712) 2B(19456) E1(19456) 1D(19200) 56(19200)
4  19/ 21    42(18432) 1C(18176) 97(18176) C0(18176) A7(18176) 1D(18176) 51(17920) 96(17920)

KEY FOUND! [ 6B:69:6E:67:73 ] (ASCII: kings )
Decrypted correctly: 100%

~/Personal/cyber-security/protocol-analysis (master) $
```

Figure 6: Output from aircrack-ng after cracking WEP on capture3.pcap

If we input this key in hexadecimal form into Wireshark (using *Preferences > Protocols > IEEE 802.11 > Decryption Keys*) it will automatically decrypt the WEP-encrypted packets. Using the `http.request` display filter we can see that the HTTP requests being sent include the header (Figure 7):

```
Authorization: Basic c3dlZXg6bXlzd2VleA==
```

Using the same Python fragment used for `capture1.pcap` above we can therefore discover that the HTTP Basic username was 'sweex' and the password was 'mysweex'.

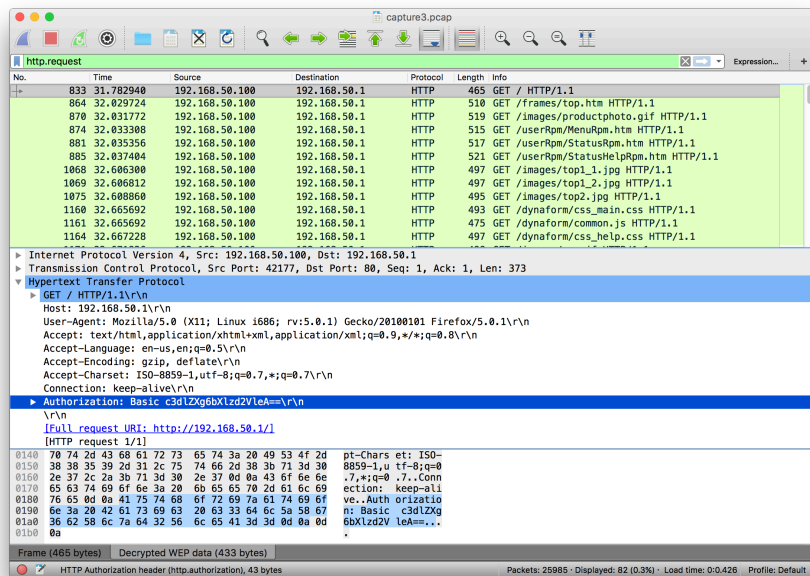


Figure 7: Wireshark showing decrypted HTTP traffic from capture3.pcap

capture4.pcap

Fast-flux

Fast-flux is a method of evasion sometimes used by cyber-criminals to prolong the lifespan of websites and other services used for illegal purposes such as phishing, often by using rapidly-changing DNS records to frustrate efforts to identify and block the ultimate source of the activity (ICANN 2008). A *fast-flux network* is a service layer built on an existing botnet of compromised Internet-connected hosts whose owners are typically unaware of their involvement.

Fast-flux can take advantage of using the Domain Name System (DNS) to return multiple A records for a given fully-qualified domain name (FQDN) with ‘round robin’ ordering (Aitchison 2011), and updating these records dynamically with short time-to-live (TTL) values. The IP addresses of multiple botnet hosts acting as *flux agents* are constantly to added and removed from a list depending on whether they are active (for example, they are accepting TCP connections on a particular port).

Upon receiving traffic from a potential victim (such as HTTP requests for a phishing site) the flux agent proxies it to a concealed origin server (sometimes known as the *mother ship*). When round robin DNS is combined with the ability of applications such as web browsers to ‘fail over’ to other IP addresses if one does not respond, this provides the fast-flux network with a degree of both anonymity and high availability (HA) and makes it difficult to block without unregistering the domain name.

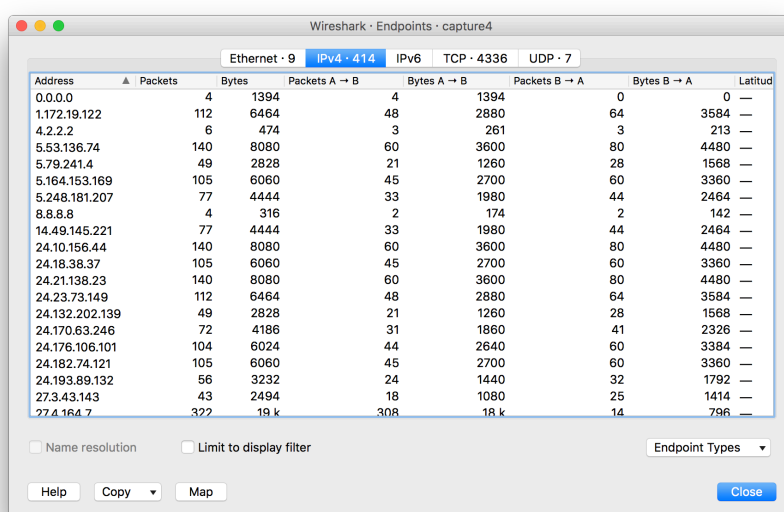
The Wikipedia entry for ‘fast flux’ notes that the term may refer more generally to “the combination of peer-to-peer networking, distributed command and control, web-based load balancing and proxy redirection used to make malware networks more resistant to discovery and counter-measures.” (2017).

Initial inspection

Wireshark shows the capture file contains 33348 packets. Using the display filter:

```
ip.addr == 172.16.253.132
```

we can see that 99.8% of the packets involve the IPv4 address 172.16.253.132. Since we are looking for evidence of fast flux we will focus on traffic involving this host, which will shall refer to as the suspected *victim host*. By using the *Protocol Hierarchy* and *Conversations* view we can see the majority of traffic uses the TCP protocol, specifically TCP connections originating from the victim to over 400 different Internet hosts (Figure 8).



The image shows the 'Endpoints' window in Wireshark for capture4.pcap. The window title is 'Wireshark · Endpoints · capture4'. It has tabs for 'Ethernet · 9', 'IPv4 · 414', 'IPv6', 'TCP · 4336', and 'UDP · 7'. The 'IPv4' tab is selected. Below the tabs is a table with columns: 'Address', 'Packets', 'Bytes', 'Packets A → B', 'Bytes A → B', 'Packets B → A', 'Bytes B → A', and 'Latitud'. The table lists various IP addresses and their corresponding packet and byte counts. The last row is highlighted in blue.

Address	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Latitud
0.0.0.0	4	1394	4	1394	0	0	—
1.172.19.122	112	6464	48	2880	64	3584	—
4.2.2.2	6	474	3	261	3	213	—
5.53.136.74	140	8080	60	3600	80	4480	—
5.79.241.4	49	2828	21	1260	28	1568	—
5.164.153.169	105	6060	45	2700	60	3360	—
5.248.181.207	77	4444	33	1980	44	2464	—
8.8.8.8	4	316	2	174	2	142	—
14.49.145.221	77	4444	33	1980	44	2464	—
24.10.156.44	140	8080	60	3600	80	4480	—
24.18.38.37	105	6060	45	2700	60	3360	—
24.21.138.23	140	8080	60	3600	80	4480	—
24.23.73.149	112	6464	48	2880	64	3584	—
24.132.202.139	49	2828	21	1260	28	1568	—
24.170.63.246	72	4186	31	1860	41	2326	—
24.176.106.101	104	6024	44	2640	60	3384	—
24.182.74.121	105	6060	45	2700	60	3360	—
24.193.89.132	56	3232	24	1440	32	1792	—
27.3.43.143	43	2494	18	1080	25	1414	—
27.4.164.7	322	19 k	308	18 k	14	796	—

At the bottom of the window, there are checkboxes for 'Name resolution' and 'Limit to display filter', both of which are unchecked. There is also a dropdown menu for 'Endpoint Types' and buttons for 'Help', 'Copy', 'Map', and 'Close'.

Figure 8: Wireshark showing endpoints in capture4.pcap

However we can also observe some UDP packets which are DNS queries and responses for the Russian domain names `wowrizep.ru` and `papcybop.ru`, shown in (Figure 9).

After resolving up `wowrizep.ru` it makes an HTTP connection to it on port 80 (85.217.248.130, actually apparently in Bulgaria) and downloads a file `instcod.exe` (842k). This appears to be a Windows executable as it starts with the characteristic character sequence MZ. Later a second executable, `calc.exe` (498k), is downloaded from 76.166.170.170.

We victim reconnects 11 more times to 85.217.248.130 (on port 80), approximately every 2 minutes, but does not sends or receive any data.

5 seconds after finishing downloading `instcod.exe` (packet 939), the victim tries to connect to 89.149.103.8 (Moldova) on port 80; after retransmitting SYN packets twice, it seems to give up. It then tries connecting to several further IPs on port 80 (93.79.183.88, 93.177.160.235, 212.87.181.1, etc., in Ukraine and Georgia) until it finds a host that accepts the connection by responding with a SYN-ACK packet (81.198.250.32 in Latvia, at packet 962). With the connection open, packets 962-973 show an unusual exchange of binary data, apparently some form of request and response.

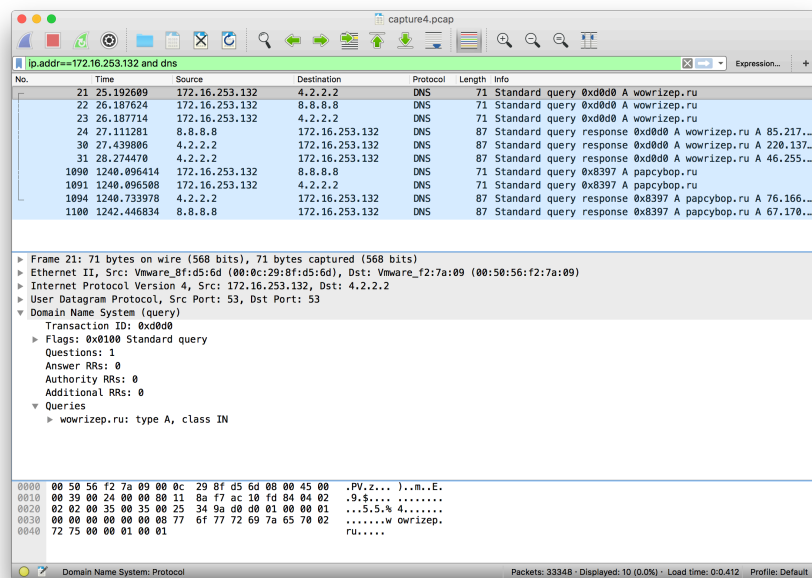


Figure 9: Wireshark showing DNS queries in capture4.pcap

Request excerpt:

```
00000000 dc dd 27 22 be de 27 22 ce de 27 22 ef e3 27 22  ..'"..' '"..' '"
00000010 f4 e2 27 22 f4 e3 27 22  ...
```

Response excerpt:

```
00000000 c9 dd 3a 22 34 de 3a 22 44 de 3a 22 e5 de 3a 22  ...:"4.:" D.:"..."
00000010 6a e2 3a 22 c5 d9 3a 22  ...
```

We can note the similar structure with the repeating hex code 0x22. This binary exchange format would be a good target for further examination.

The victim attempts a series of further connections on port 80, including some which resemble legitimate HTTP requests (GET /index.htm, GET /start.htm, etc.). However they are unusual in that they often include a binary payload after the HTTP request (which HTTP GET requests typically do not).

The victim also uses a number of different user-agent strings, apparently to mask easy identification/blocking, e.g.:

```
Mozilla/4.0 (compatible; MSIE 5.5; Windows 95)
Mozilla/5.0 (X11; U; Linux i686; de-AT; rv:1.7.5) Gecko/20041222 Firefox/1.0 (Debian package 1.0-4)
Mozilla/5.0 (Windows; U; Windows NT 6.1; ja; rv:1.9.2a1pre) Gecko/20090403 Firefox/3.6a1pre
Mozilla/5.0 (Windows; U; Windows NT 6.0; zh-HK; rv:1.8.1.7) Gecko Firefox/2.0
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2; Win64; x64; SV1)
Mozilla/5.0 (X11; I; SunOS sun4u; en-GB; rv:1.7.8) Gecko/20050713 Firefox/1.0.4
```

We can therefore assume that instcod.exe (and possibly also calc.exe) is the malware responsible

for initiating fast flux connections to geographically distributed IP addresses.

Analysis script using Scapy

A Scapy script (`poisonarp.py`) was written to support the above findings and provide further detail about the capture file. It can be executed as follows (assuming Scapy has been installed):

```
$ python fastflux.py cwk_pcaps/capture4.pcap
```

Example script output (extracts)

```
Read 33349 packets from cwk_pcaps/capture4.pcap
== Analysis of domain names
DNS query from 172.16.253.132 to 4.2.2.2: wowrizep.ru.
DNS query from 172.16.253.132 to 8.8.8.8: wowrizep.ru.
...
== Analysis of IP addresses
= IPs attempting to establish TCP connections:
172.16.253.132: 4456 packet(s)
= IPs and ports being connected to (in order of frequency):
106.1.153.162:80: 26 packet(s)
67.249.109.170:80: 23 packet(s)
...
== Analysis of malware

Possible HTTP malware request to 85.217.248.130:
GET /instcod.exe HTTP/1.0
Host: wowrizep.ru
...
== Analysis of sessions
= TCP sessions where data is sent or received:
172.16.253.132:4846 71.203.178.250:80: (16 packets)
172.16.253.132:2560 173.212.117.87:80: (31 packets)
172.16.253.132:3399 31.192.26.202:80: (31 packets)
...
```

References

- Aitchison, R., 2011. *HOWTO - Configure Load Balancing* [online]. Available from: <http://www.zytrax.com/books/dns/ch9/rr.html> [Accessed 18 Feb 2017].
- Berners-Lee, T., Fielding, R., and Frystyk, H., 1996. *RFC1945: Hypertext Transfer Protocol -- HTTP/1.0* [online]. Available from: <https://tools.ietf.org/search/rfc1945> [Accessed 3 Feb 2017].
- Fluhrer, S., Mantin, I., and Shamir, A., 2001. Weaknesses in the Key Scheduling Algorithm of RC4. *Selected Areas of Cryptography*, 2259, 1–24.
- ICANN, 2008. *SAC 025: SSAC Advisory on Fast Flux Hosting and DNS* [online]. Available from: <https://www.icann.org/en/sac/sac025>

[//www.icann.org/en/system/files/files/sac-025-en.pdf](http://www.icann.org/en/system/files/files/sac-025-en.pdf) [Accessed 18 Feb 2017].

Lamping, U., Sharpe, R., and Warnicke, E., 2014. *Wireshark User's Guide: Capturing with tcpdump for viewing with Wireshark* [online]. Available from: https://www.wireshark.org/docs/wsug_html_chunked/AppToolstcpdump.html [Accessed 3 Feb 2017].

Mishra, C., 2016. *Mastering Wireshark*. Birmingham, UK: Packt Publishing.

Mogul, J., 1984. *RFC919: Broadcasting Internet Datagrams* [online]. Available from: <https://tools.ietf.org/html/rfc919> [Accessed 3 Feb 2017].

Orebaugh, A., Ramirez, G., and Beale, J., 2006. *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Rockland, MA: Syngress.

Plummer, D. C., 1982. *RFC829: An Ethernet Address Resolution Protocol* [online]. Available from: <https://tools.ietf.org/html/rfc826> [Accessed 11 Feb 2017].

Wikipedia, 2017. *Fast flux* [online]. Available from: https://en.wikipedia.org/wiki/Fast_flux [Accessed 18 Feb 2017].