**Progress 2**

To begin, I’m going to review all the commands available in the Python-RAT. After testing these commands, I decided to remove several due to issues such as commands that weren’t working correctly, features that interfered with the program’s usability, or simply because they were unnecessary.

These commands are:

disabletaskmgr

turnoffmon

turnonmon

sendmessage

extendrights

enabletaskmgr

profiles

portscan

profilepwsd

setvalue

delkey

createkey

[Driver]

Webcam

Below is the list of commands I chose to add, along with their intended purpose:

whoami: Displays the current user.

renamefile <oldname> <newname>: Provides a convenient way to rename a file. If the user decides they want to change a file's name, this command can handle it easily by renaming oldname to newname.

zipfile <filename>: Compresses a file or directory into a .zip archive. This can be especially useful when the user wants to upload a file or folder to a server or share it online.

unzipfile <filename>: Extracts the contents of a .zip archive into a directory. Since we’re zipping files I decided why not make an unzip command.

Clear: Clears the screen. I decided to implement this because when I needed to clear the screen there was no clear command.

**Implementations Approach**

Note: all print is for debugging for me personally.

How I approached the whoami command is by researching the subprocess module. Found a good website that explains and shows example of python subprocess <https://www.datacamp.com/tutorial/python-subprocess>. Looking at the other commands, how they’re coded we can do similar by doing an elif command == ‘whoami’: and then a try block in the client.

Below is the code

A screenshot of a computer code

Description automatically generated

subprocess.check\_output will execute the whoami command in cmd as it has a built in whoami function and capture it as a string instead of bytes and then username is stored in current\_user.

For the server, we can do the same as the other command elif and then self.result().



Output

A close up of a computer screen

Description automatically generated

For the renamefile command, first have to import zipfile module, used an elif statement along with a try block. For commands that require arguments, I decided to include a usage message in my server. This approach is particularly useful for me, as it provides clear guidance on the correct command format when needed. Initially, I attempted to use the == tomatch the command but encountered issues, not sure why, so Instead did a. startswith(). This made the implementation work as intended.

Client:

A screenshot of a computer code

Description automatically generated

Server:

A screenshot of a computer code

Description automatically generated

output:

A black and blue text

Description automatically generated

A screenshot of a computer

Description automatically generated

For the zipfile command. I did some research and found a very useful built-in module for zipping archives.

<https://docs.python.org/3/library/zipfile.html>

I structured the implementation with an elif command.startswith() condition and a try block.

Client:A screenshot of a computer program

Description automatically generated

Server:

A screen shot of a computer code

Description automatically generated

Output:



A screenshot of a computer

Description automatically generated

For unzipfile command will be reading and extracting instead of writing.

A screen shot of a computer code

Description automatically generated

A white background with blue dots

Description automatically generated



Forgot to mention that the testfolder included a hi.txt.

For clear command, all code will be on the server. Windows and Linux has their own build in clear commands.

Server:

A screen shot of a computer program

Description automatically generated

Output

A close up of a word

Description automatically generated

**Process Monitor Exploration**

Now that I've added some custom commands, I wanted to see how they function behind the scenes using Process Monitor (Proc Mon). Here's how I conducted my investigation:

The first thing I’m going to do is exit out of visual code, and start proc mon.

Opened Visual Studio Code and then ran both the server and client scripts.

Searched for code.exe (the Visual Studio Code process) and added an "Include" filter for it.

Further narrowed down the results by filtering for my project folder name, ICS460-Python-RAT, since that's where my scripts are located.

However, this method returned too much data—there were too many files and irrelevant entries to parse effectively.

A screenshot of a computer

Description automatically generated

Switching to the Process Tree:

To get a clearer picture, I explored the Process Tree view in Proc Mon.

This allowed me to visualize the hierarchy and relationship between parent and child processes.

Findings in the Process Tree:

A screenshot of a computer

Description automatically generated



code.exe (PID: 7528) is the parent process.

It spawned powershell.exe (PID: 13740), which represents the Visual Studio Code terminal. I found this interesting

Child Processes: Client:

A Python process (PID: 10932) was started when the .venv\Scripts\Activate command was executed. This indicates the activation of the virtual environment and the subsequent launch of the client script.

Server:

A similar sequence occurred for the server, with its Python process being launched after the environment was activated.

**Let’s test out some commands**

Whoami

A screenshot of a computer

Description automatically generated

The whoami command operates by launching cmd.exe, which in turn calls whoami.exe. Quite interesting as it knows the description. It is also short lived.

A screenshot of a computer

Description automatically generated

I was able to track and analyze the behavior of the whoami command and its interaction with the server and client. Here's a step-by-step breakdown:

When the whoami command is executed, Python initiates a new cmd.exe process with the argument /c "whoami", creating a shell to run the command

C:\Windows\system32\cmd.exe /c "whoami".

After the cmd.exe process is created, it performs several registry reads and file accesses to retrieve the necessary information to run the whoami executable and return the current user.

Once the whoami command finishes executing, the result is captured by Python's subprocess module and sent to the server over the TCP connection.

**TCP Receive** indicates the client sending the result (whoami output) to the server on port 4444 (the server's listening port remember in progress 1 4444 is our server).

**TCP Send** confirms the server successfully received the message, completing the communication loop.

Very interesting was a great way to visualize the inner workings and the server-client interactions.

Let’s try another command with wireshark captures:

volumeup

A screenshot of a computer

Description automatically generated

It increased my volume to 100% wow

A blue line on a white surface

Description automatically generated

Capture loopback

A screenshot of a computer

Description automatically generated

A white background with black text

Description automatically generated

A screenshot of a computer

Description automatically generated

A close up of a screen

Description automatically generated

Wireshark captures communication between the server (127.0.0.1:4444) and the client (127.0.0.1:54197). The packet with Len=8 indicates that the server sent 8 bytes of data. This matches the message size observed in Process Monitor. The packets show proper acknowledgment of data received, confirming reliable communication. In one of the packets, the captured data includes the volumeup command, confirming that the server issued the command to the client.

Process Monitor Observations:

TCP Activity: Similar to Wireshark, Process Monitor confirms TCP send and receive operations between ports 4444 and 54197. The Length: 8 in Process Monitor aligns with Wireshark’s Len=8, showing consistency in message size during communication. This length of 8 bytes corresponds to the command "volumeup" being sent from the server to the client. Upon executing the volumeup command, Process Monitor logs show registry queries and accesses related to audio rendering. These interactions demonstrate the client’s efforts to adjust the system’s audio settings programmatically, aligning with the expected behavior for the volumeup command. At the end it shows the communication loop completes successfully, as indicated by the exchange of ack packets shown in Wireshark quite interesting.

It is quite interesting to observe both sides of the interaction—how commands and responses are transmitted over the network and the corresponding activities at the software level.

Overall, the combined efforts in Progress 1 and 2 showcased a comprehensive exploration of the RAT’s capabilities, ranging from setting up the development environment to implementing and analyzing new features. This project not only provided hands-on experience with development setup, TCP/IP communication, and Python scripting but also introduced practical use of tools like Wireshark and Process Monitor for debugging, network traffic analysis, and system-level diagnostics. Overall, was a good experience for me.

Below is the updated github to the server and client.

<https://github.com/JimmieXiong/ICS460-Python-RAT>