**Progress Report 1**

Please note that this is not a comprehensive guide, and not every aspect of the program has been covered, as that would be too extensive. I encourage you to explore the program independently and conduct your own research to develop a solid understanding of its overall structure and functionality.

**Objective:**

* Effectively set up and configure a development environment in VS Code, establishing a solid foundation for future projects.

**Setting Up Python-RAT in Visual Studio Code (VS Code)**

This guide will help you set up a Python Remote Access Tool (RAT) project using Visual Studio Code (VS Code). This includes installing necessary software, configuring the environment, and running server-client script.

**For this project, we’ll be using Visual Studio Code. Why Visual Code?**

* To gain hands-on experience VS Code it’s widely used, lightweight, and has powerful features such as IntelliSense, debugging, Git integration, etc.
* Extensibility – has loads of extensions, to use
* Support integrated terminal VS Code allows users to run shell commands
* It’s a cross platform so whether you’re using it on Mac OS, Linux, etc it’ll work smoothly. If you’re offended by this and you prefer using tools like Vim or other text editors on Linux, you're welcome to do so—feel free to choose the environment that best suits your workflow.

**Step 1: Prepare Your Environment**

1. Disable Windows Defender: This program may be flagged as a reverse shell by Windows Defender. Turn off:
   * Real-Time Protection
   * Cloud-Delivered Protection
   * Automatic Sample Submission
   * Tamper Protection
   * Windows Firewalls
2. **Install Visual Studio Code (VS Code):**
   * Download from Visual Studio Code's website.
   * We’re using VS Code because it’s lightweight, widely used, and has excellent features like IntelliSense, debugging, and Git integration.
3. **Install Python 3.10.0:**
   * Download from Python’s website.
   * This version is used to prevent compatibility issues that may arise with newer Python versions.
4. Add Python to System PATH:
   * Adding Python to PATH ensures the terminal recognizes Python commands.
   * Navigate to This PC > Properties > Advanced System Settings > Environment Variables.
   * Under System Variables, edit the Path variable to include the path to your Python installation (usually C:\Users\<username>\AppData\Local\Programs\Python\Python310).

**Step 2: Cloning the Git Repository**

1. Open Visual Studio Code
2. Press Ctrl + Shift + P to open command palette.
3. Type “Git: Clone” and select it.
4. Note if command does not populate that means you need to install git

**Download Git here:**

<https://git-scm.com/downloads/win>

1. After installing Git, repeat the step to search for Git: Clone.
2. When prompted, enter the repository URL

<https://github.com/FZGbzuw412/Python-RAT>

1. After entering the URL, you will be asked to choose a location on your computer where you want to clone the repository.
2. Choose a folder on your computer where you want to clone the repository.
3. Once done, Visual Studio Code will load the project from the cloned repository.

**Step 3: Setting up the Virtual Environment**

To set up the project virtual environment, you'll need to create and activate a virtual environment. Follow these steps:

1. In the terminal, navigate to the project folder if you have not. If you don’t know the project folder location, it’s the directory you selected during the Git cloning process
2. Run the following command to create a virtual environment:

**python -m venv .venv**

1. Then active the virtual environment

**.venv\Scripts\activate**

A screen shot of a computer program

Description automatically generated

1. If you encounter error related to execution policy, you’ll have to set the execution policy to allow running scripts
2. Open up PowerShell as administrator
3. Follow the commands below or just the second

A computer screen with white text

Description automatically generated

1. Command

**Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope LocalMachine**

* + Will set execution policy for the entire machine which is permanent across all users and sessions.

1. Why do we have to set?
   * Because it’ll allow you to run PowerShell scripts. The bypass argument is making sure nothing is blocked, and no warnings are promoted.
2. If you want to change the execution policy later, you can do it yourself. However, for the sake of this project I’m going to set it to be permanent.
3. After adjusting the execution policy, go back to Visual Studio Code and activate the virtual environment again:
4. You’ll know it’s working if you see the virtual environment name in green 

**Step 4: Installing Requirements**

1. The project comes with pre-defined requirements. Notes to whoever the developer is, they’re very knowledgeable.
2. To ensure the project functions properly, we need to install the necessary packages.
3. Install using command

**pip install -r client\_requirements.txt**



**pip install -r server\_requirements.txt**



1. You can verify installation by command

**pip list**

A screenshot of a computer

Description automatically generated

This will show you all the installed packages.

**Step 5: Running the Server and Client**

1. To run the server and client:
2. Split terminal to open another terminal
3. Make sure you enable the virtual environment for both terminals when you split. You can tell when it’s green.
4. In the current terminal, run the server:

**python server.py**

1. In the second terminal, run the client script

**python client.pyw**

1. If you encounter any missing module errors ex: (ModuleNotFoundError: No module named 'keyboard'), install the missing module using:
2. pip install <module\_name>

**ex: pip install keyboard** A screenshot of a computer program

Description automatically generated

1. Once modules are all installed and no missing.
2. Run the client.pyw script and if you see a connection in the server it’s working. You can test out the commands as well.

A screenshot of a computer program

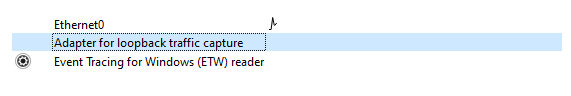
Description automatically generated

That’s it! You should now have the Python-RAT project up and running in Visual Studio Code.

**Objective:**

* Gain insights into TCP/IP socket communication, learning how data is encoded, transmitted, and interpreted between client and server applications.

**Understanding the Client-Server Communication (Using Wireshark)**

1. In your VM install and download Wireshark.
2. In Wireshark, select the loopback adapter to capture packets. This allows you to see the traffic between the client and server on 127.0.0.1.

**Why Select the Loopback Adapter in Wireshark?**

* The client and server communicate locally using the loopback address 127.0.0.1, meaning all data exchange happens within the same computer. The server listens on port 4444 at 127.0.0.1, while the client connects using a dynamically assigned port (an ephemeral port that changes with each session, like **50194** in our example).
* By selecting the loopback adapter in Wireshark, we can capture and view the entire communication between the client and server, including both the fixed server port (4444) and the dynamic client port. This setup lets us see all packets exchanged through the internal loopback interface, providing a complete view of the client-server interaction on the local machine.

**Activate the Virtual Environment and Start the Server**:

* In VS code terminal active virtual environment and run the server

A computer screen shot of white text

Description automatically generated

**Open a Split Terminal and Start the Client**:

* In the other split terminal enable virtual environment and run the client

A black screen with white text

Description automatically generated

**Verify the Connection**:

* If the connection is successful, you’ll see a confirmation message in the server terminal. This indicates that the client has connected to the server

A screenshot of a computer screen

Description automatically generated

**Testing Commands**:

* Now, try running commands in the sever terminal, like **help**, **dir** and **isuseradmin**. These commands should be sent to the client, which will respond with the appropriate output. This data transmission allows you to see how commands and results are sent back and forth between the client and server.

A screenshot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screenshot of a computer screen

Description automatically generated



**Analyzing with Wireshark**:

1. Open Wireshark and use the loopback adapter to capture local traffic. You’ll see the packets associated with the client-server communication on 127.0.0.1.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Understanding the TCP Three-Way Handshake and Ports**

* The first packet you see will likely be the **TCP Three-Way Handshake**. This process establishes a connection between the client and server, ensuring they’re ready to exchange data. During the handshake, you'll notice:
* **Source Port (50194)**: This is the dynamic port assigned to the client by the OS.
* **Destination Port (4444)**: This is the fixed server port where the server listens for incoming connections on 127.0.0.1.

**Checking Ports and Process IDs with netstat**

* I’m curious as to what those ports actually are.

**Identify Active Ports and Processes**:

* Open command prompt and run

**netstat -aon | findstr :50194**

This command shows details for the client port 50194. The output will include the type of protocol, local address, foreign address, connection, Process ID

A computer screen with white text

Description automatically generated

**Understanding the Output:**

* **TCP:** The protocol used for communication.
* **Local Address (127.0.0.1:4444):** Indicates the server’s IP and port, where it listens for connections.
* **Foreign Address (127.0.0.1:50194):** Refers to the client’s dynamically assigned port.
* **ESTABLISHED:** This shows an active, open connection between the client and server.
* **PID (Process ID):** **4224, 3440** the process ID associated with the ports.

**Identify Processes Using PIDs**:

Now you might think I’m curious as to what programs are using these ports.

* To find which programs are using these ports, type:

**tasklist | findstr 4224**

**tasklist | findstr 3440**

* In this case, python.exe processes are running the server and client, respectively.

**Now let’s examin the Code to Understand the Server and Client**

* In the client’s code (look for the line near the main function), you’ll see

A computer screen shot of text

Description automatically generated

**rat = RAT\_CLIENT('127.0.0.1', 4444)**

* This line specifies that the client will connect to the server at 127.0.0.1 on port 4444.
* We can see that the rat.build\_connection() method is called which is the method to build the connection lets peek at the definition

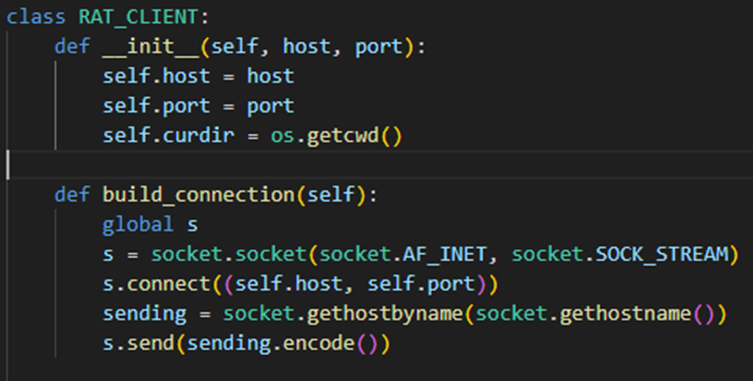
A screen shot of a computer program

Description automatically generated

In the build\_connection method, we can see how the client sets up its connection to the server:

1. **self as the Instance Reference:**
   * The self-parameter refers to the specific instance of the RAT\_CLIENT class that calls this method. This allows self to access attributes unique to each instance, like self.host and self.port.
2. **Global Variable s:**
   * The method uses a global variable s, which means it can be accessed outside this method if needed.
3. **Creating the Socket:**
   * The line s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) creates a new socket object s.
     + AF\_INET specifies that the socket will use IPv4 addresses.
     + SOCK\_STREAM indicates it’s a TCP socket
4. **Connecting to the Server:**
   * The method then attempts to connect to the server using s.connect((self.host, self.port)). This line uses the server address and port saved in self.host and self.port.
   * When the function connect Is called, the operating system automatically assigns an available, temporary (ephemeral) port to the client for this connection. This port is not defined in the code by the developer; instead, they want the OS to dynamically picks one from the ephemeral port range (in the range 49152–65535 on Windows).
   * In this case, the OS assigned port **50194** as the ephemeral port for the client.
   * Source: https://www.ncftp.com/ncftpd/doc/misc/ephemeral\_ports.html
5. **Sending the Client's IP Address:**
   * After connecting, the client retrieves its hostname, converts it to an IP address, then to bytes, and finally sends this IP address to the server using s.send(sending.encode()).

**You might be wondering how an instance is created and initialized in Python.**

****

* In the code, if you look above the build\_connection method, you’ll see the \_\_init\_\_ method, which is a function in Python called a constructor. Like constructors in Java, this method is automatically called when you create an instance of the class.
* The \_\_init\_\_ method takes (self, host, and port) as parameters.
* self refers to the instance being created. It allows the method to set up attributes specific to that instance.
* self.host = host and self.port = port save the host and port values to this instance, so they can be used throughout the instance’s lifecycle.

For example, when you instantiate RAT\_CLIENT like this:

**rat = RAT\_CLIENT('127.0.0.1', 4444)**

the values **'127.0.0.1'** and **4444** are passed to **\_\_init\_\_**. Inside **\_\_init\_\_**, these values are assigned to **self.host** and **self.port**, making them accessible to other methods in the class, such as **build\_connection**.

So, whenever self is used, it refers to this specific instance (**rat** in this example), allowing it to access its own **host** and **port** values.

**Final Confirmation of Roles:**

* Based on this analysis:
  + **PID 4224** is the **server** (listening on port 4444).
  + **PID 3440** is the **client** (using the dynamic port 50194).

A computer screen with white text

Description automatically generated

**Now that we've identified which processes correspond to which ports, let's return to Wireshark for further analysis.**

A screenshot of a computer

Description automatically generated

I was able to capture 13 packets.

Packet 1: 50194 -> 4444 [SYN]

* Port 50194 is the client as we have already determined that it initiates a connection to the server (port 4444) by sending a SYN (synchronize) packet. This is the first step of the TCP Three-Way Handshake.

Packet 2: 4444 -> 50194 [SYN, ACK]

* The server (port 4444) responds with a SYN-ACK packet, acknowledging the client’s connection request. This is the second step.

Packet 3: 50194 -> 4444 [ACK]

* The client responds with an ACK packet, confirming the connection is established. This completes the Three-Way Handshake, and the connection is now established.

As we look at packet 4: 50194 -> 4444 [PSH, ACK]

* In Packet 4 (50194 → 4444 [PSH, ACK]), we can see data being sent from the client to the server. The data content (31302e302e302e313138) converts to ASCII as 10.0.0.118, which is the client’s local IP address.
* Now what is 10.0.0.118? To understand this, let’s look at the client code once again.

A screen shot of a computer program

Description automatically generated

* Here, we see sending = socket.gethostbyname(socket.gethostname()). This line retrieves the client’s local IP address (10.0.0.118 in this example) and sends it to the server.

**How to Find Your Local IP Address**

* Go to cmd and type ipconfig

A screen shot of a computer

Description automatically generated

* In this output, the IPv4 Address shows 10.0.0.118, which matches the data sent in Packet 4. Note that the AF\_INET parameter in the code specifies the use of IPv4, which is why we’re retrieving an IPv4 address (10.0.0.118) for the client.
* Therefore, based on this analysis packet 4 is performing this exact operation retrieving the client’s local IP address (10.0.0.118) and sending it to the server over the established TCP connection.

Packet 5: 4444 -> 50194 [ACK]

* As expected, Packet 5 is sent from the server to the client with an ACK flag. This acknowledgment confirms that the server successfully received the data sent in Packet 4, which included the client’s IP address (10.0.0.118).

Packet 6: 4444 → 50194 [PSH, ACK]

* In Packet 6, we see data being sent from the server to the client. The PSH flag instructs the client to push this data directly to the application layer without buffering.
* The ACK flag indicates that the server has received all previous data from the client.
* This packet contains 3 bytes of data, represented by 646972 in hexadecimal. Converting 646972 to ASCII reveals the command dir, which is the command we sent.
* Therefore, this packet is sending the dir command to the client telling it to execute the command and provide the output.

**You might be wondering, "Wait, didn't we first send a help command? Why don’t I see it in the packet analysis before the dir command? Shouldn’t the help command data appear first?"**

* To clarify: Yes, we did execute the help command initially. However, the help command was processed locally on the server. It simply listed the available commands without needing to send any data to the client, so no packet for help was transmitted over the network.

Packet 7: 50194 → 4444 [ACK]

* This packet is sent from the client to the server, indicating that the client has received the previous packet (Packet 6) from the server, which contained the dir command.

I forgot to point out what Seq, Ack, Win, and len means

**Explaining Seq, Ack, Win, and Len**A screenshot of a computer code

Description automatically generated

Let’s go back to Packet 6 to understand what each of these fields means:

Seq (Sequence Number):

* This represents the starting byte position of the data being sent in the current packet. In Packet 6, the sequence number is 1, meaning the server’s data stream starts from byte position 1.

Since Packet 6 contains 3 bytes of data, the next expected sequence number will be 1 + 3 = **4**, which is what the client acknowledges in Packet 7.

Ack (Acknowledgment Number):

* This indicates the next byte that the sender (in this case, the server) expects to receive from the other side (the client).
* In Packet 6, the acknowledgment number is 11, meaning the server has received all data from the client up to byte 10 and is ready for more data starting at byte 11.

Win (Window Size):

* The window size (Win) specifies how much more data the sender is willing to receive before expecting an acknowledgment.
* In Packet 6, the window size is 2619648, which is the amount of buffer space available on the server for receiving data.

Len (Length):

* This represents the amount of data (in bytes) being carried in the packet. In Packet 6, Len=3, meaning there are 3 bytes of data in this packet, which is the dir command.

Let look at packet 7

A screenshot of a computer

Description automatically generated

Here we see

**Len=0**: packet contains no additional data, as it’s purely an acknowledgment request.

**Seq=11**: This is the current sequence number for the client, indicating where its data stream is at this point.

**Ack=4**: This acknowledgment number confirms that the client has received all data from the server up to byte 4, which corresponds to the 3 bytes of data containing the dir command sent by the server in **Packet 6**.

**Calculated window size**: 2619648

Packet 8: 50194 → 4444 [PSH, ACK]

* Packet 8 is the client’s response to the server’s dir command. The PSH flag ensures that the data (725 bytes) is immediately pushed to the server’s application layer, making it available for processing without delay. This data contains the directory listing from the client.
* The ACK flag confirms that the client has received the previous dir command from the server, maintaining a reliable connection.

Packet 9 follows this by sending an ACK request back from the server, confirming that it received the directory listing data from the client.

Packet 10: 4444 -> 50194 [PSH, ACK]

* Packet 10 indicates that the server is sending the command isuseradmin to the client. Similar to packet 6 how the dir command is sent.

Packet 11:

* Just like packet 7. Packet 11 is sent from the client to the server, indicating that the client has received the previous packet (Packet 10) from the server, which contained the isuseradmin command.

Packet 12:

* The client is sending a message (10.0.0.118 is not admin) back to the server as a response to the isuseradmin command. This message provides the client’s privilege level. If VS Code is not running as admin, it returns "not admin"; if VS Code is running with admin privileges, it returns "admin."
* The data content of this packet 31302e302e302e313138206973206e6f742061646d696e, which translates to ASCII as "10.0.0.118 is not admin".

**End of Progress 1**

Now that we have successfully set up the project environment and grasped how the client and server communicate over the network, we will transition to the next phase.

In Progress 2, we will delve deeper into the code to explore its functionality, implement additional commands to enhance the RAT, and observe how the program interacts with the operating system.

**Progress 2**

Discuss further changes, completion percentage, reasons for decisions, challenges faced, and strategies employed to overcome them. Update us on any group composition modifications due to class changes.

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.

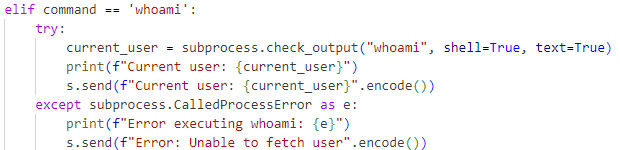
Clear: Clears the screen. I decided to implement this because when I needed to clear the screen there was no clear command.

**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

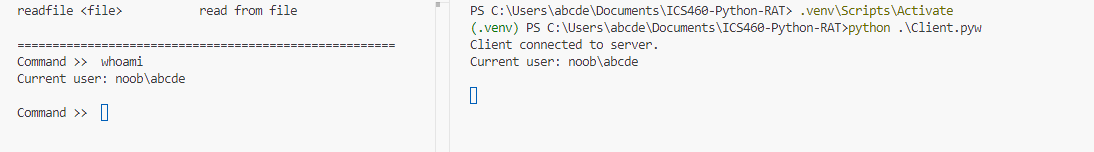


subprocess.check\_output will execute the whoami command in cmd as it has a built in whoami function and capture it as a string instead of bytes and then username is stored in current\_user.

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



Output



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:



Server:

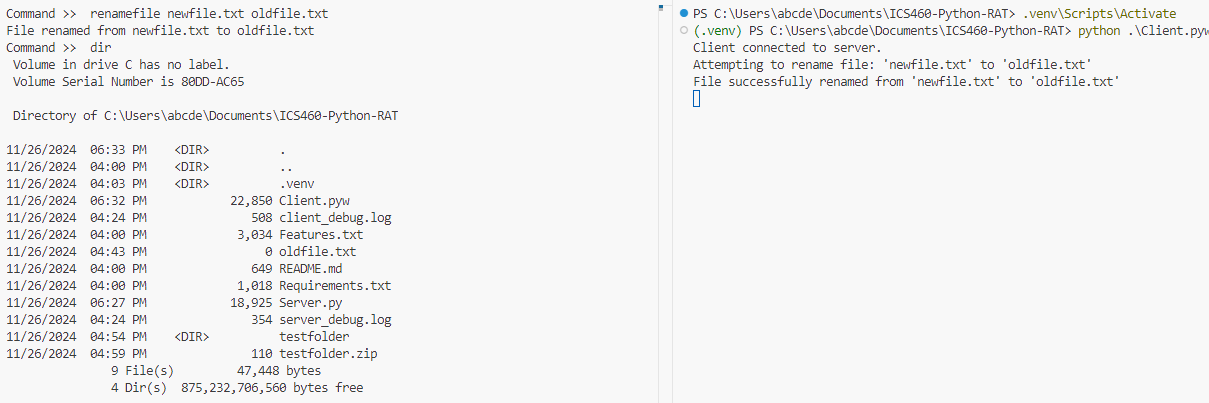
A screenshot of a computer code

Description automatically generated

output:

A black and blue text

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

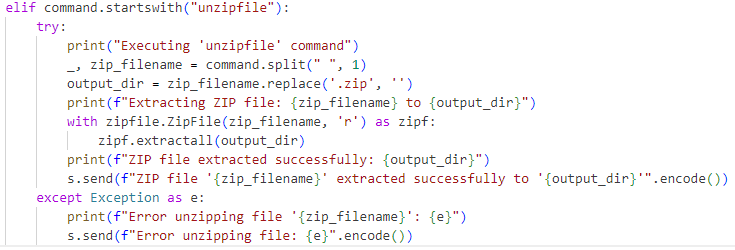
Output:



A screenshot of a computer

Description automatically generated

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



A white background with blue dots

Description automatically generated



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

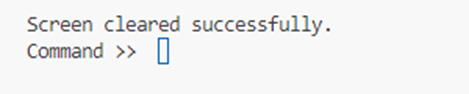
For clear command, all code will be on the server. Windows and Linux has their own build in clear commands.

Server:

A screen shot of a computer program

Description automatically generated

Output



**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:

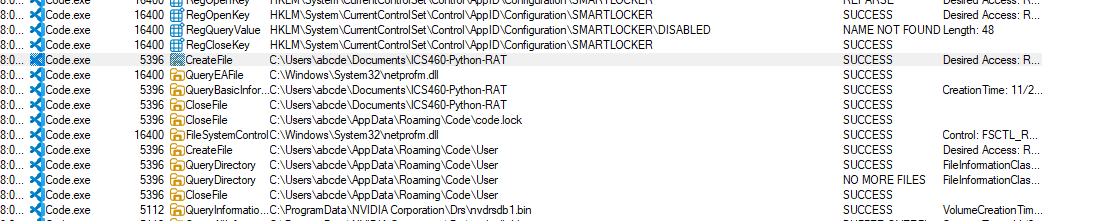
The first thing I’m going to do is exit out of visual code, and start proc mon.

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.

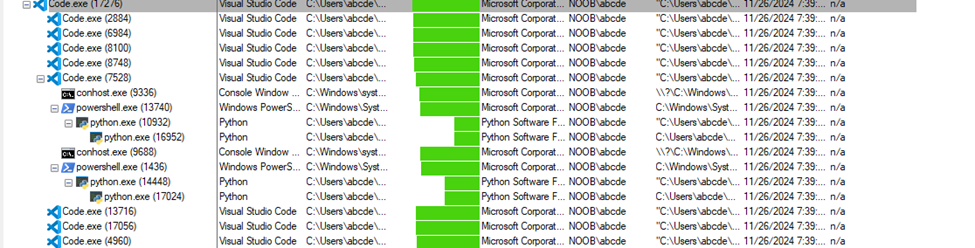


Switching to the Process Tree:

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:





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.

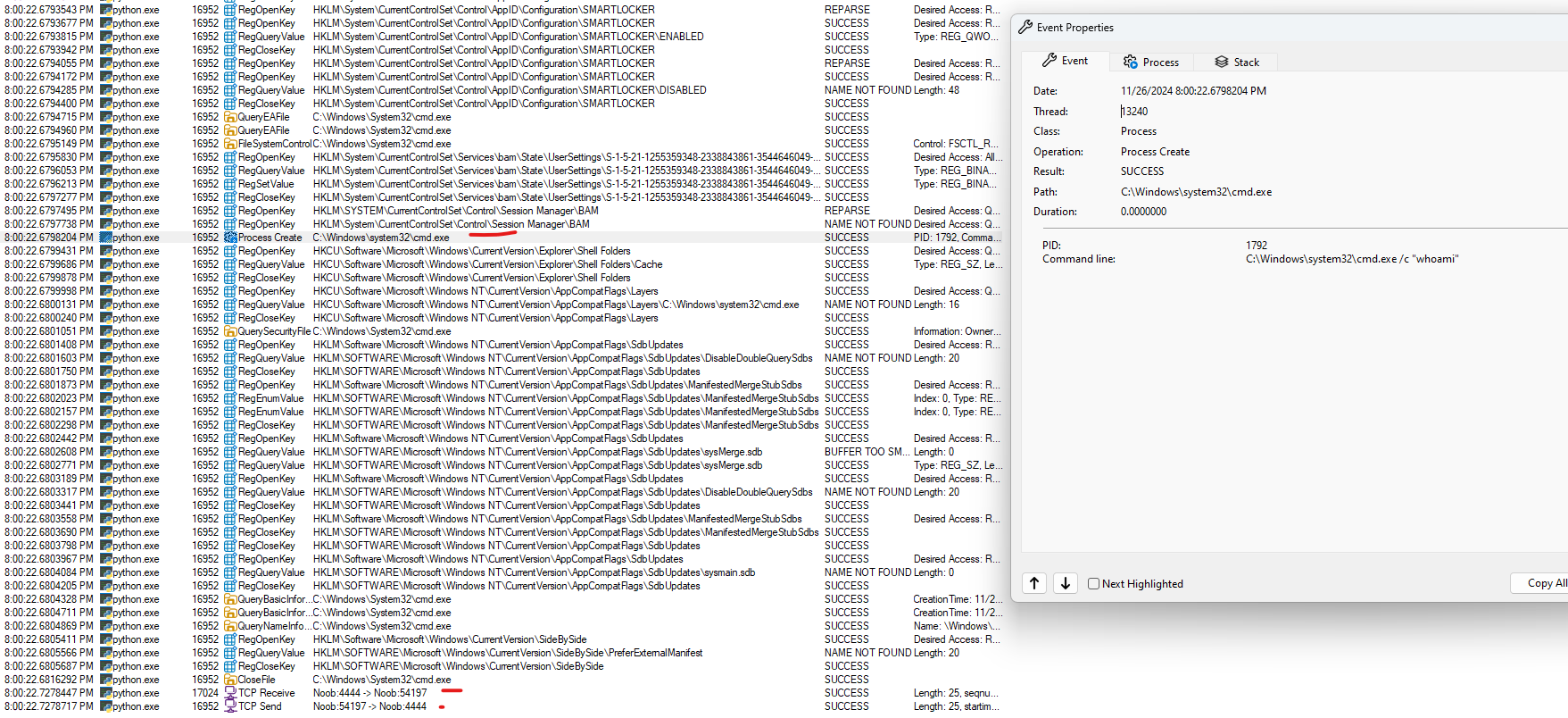
**Let’s test out some commands**

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.



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.

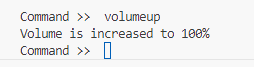
**TCP Receive** indicates the client sending the result (whoami output) to the server on port 4444 (the server's listening port remember in progress 1 4444 is our server).

**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



It increased my volume to 100% wow

A blue line on a white surface

Description automatically generated

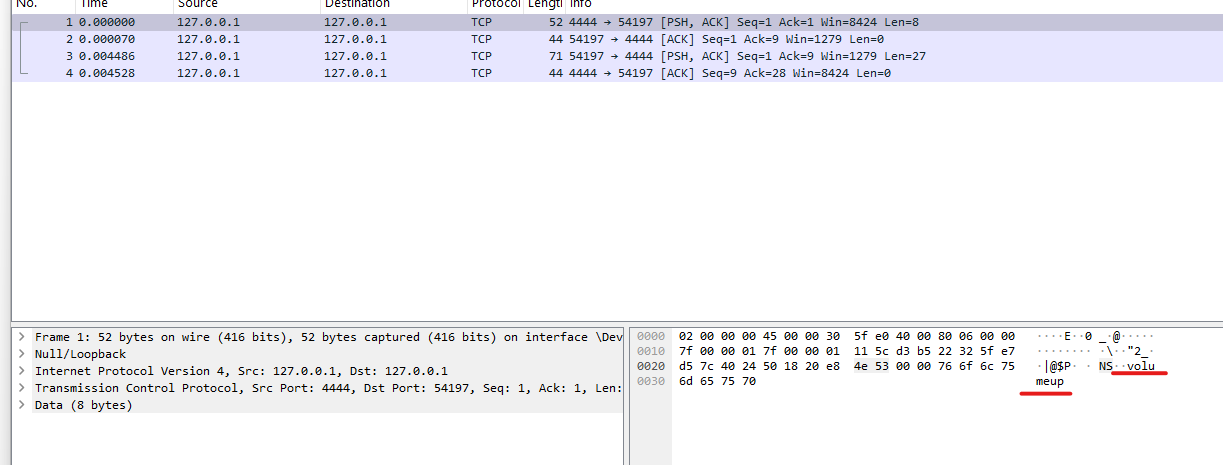
Capture loopback

A screenshot of a computer

Description automatically generated

A white background with black text

Description automatically generated



A close up of a screen

Description automatically generated

Wireshark captures show 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.

Volume Command Sent: 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. Also 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.