Backdoor Reverse Shell Using Python

Client-Server Communication in Python

This project focuses on building a basic, multi-threaded "echo server" in Python. This type of server is a foundational building block for all network applications. It demonstrates how to handle multiple connections at once and process requests, in this case by "echoing" a message back to the client or serving a file.

Objective

To develop a Python script that implements a multi-threaded TCP server. The server will be able to accept connections from multiple clients simultaneously, receive text commands, and respond by either "echoing" the text or (if requested) sending a file's contents back to the client.

A client-server model is the basis of the internet. A "server" is a program that listens on a specific network port (e.g., port 80 for web pages). A "client" is a program (like your web browser) that connects to that port to make a request. In this project, we build a simple server that listens on port 9009 and a client that connects to it, demonstrating the core principles of network socket programming.

Working Process

1. **Server (echo_server.py):**
   - The server script starts and "binds" to port 9009, listening on all network interfaces (0.0.0.0).
   - It enters an infinite loop, waiting for a client to connect using s.accept().
   - When a client connects, the server creates a **new thread** to handle that specific client (using threading.Thread). This allows the main loop to immediately go back to listening for *other* clients.
   - The client's thread enters its own loop, waiting for data (conn.recv()).
   - If the client sends quit, the thread sends "Goodbye" and closes the connection.
   - If the client sends GETFILE filename, the server reads that file, base64 encodes it (to ensure safe transmission), and sends it to the client.
   - For any other text, the server simply sends "ECHO: " plus the original text back.
2. **Client (echo_client.py):**
   - The client script starts and connects to the server's IP address and port (127.0.0.1:9009).
   - It enters an infinite loop, prompting the user for input (>).
   - It sends the user's command to the server (s.sendall()).
   - It then waits for a response (s.recv()).

- If the response starts with "FILEBEGIN", it knows a file is coming. It reads all the data until "FILEEND", decodes the base64 data, and saves it as downloaded_file.
- Otherwise, it just prints the server's "echo" response to the screen.

Tools and Libraries

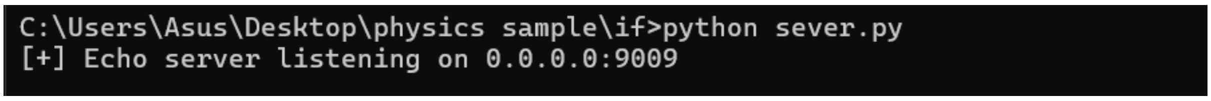
Tool	**Purpose**
Python	The core programming language.
socket	Standard library for all low-level network communication (connecting, listening, sending, receiving).
threading	Standard library that allows the server to handle multiple clients at the same time without freezing.
os	Standard library used by the server to check file paths and prevent directory traversal attacks.
base64	Standard library used to encode binary files (like images) into text so they can be sent safely over a text-based stream.

Code Explanation

- **handle_client(conn, addr)**: This function is the "mini-server" that runs in its own thread for each connected client. conn is the socket object for that specific client.
- **s.bind((HOST, PORT))**: Tells the server "You own this port." 0.0.0.0 means "listen for connections from *anywhere* (other computers on the network, not just your own machine)."
- **s.listen(5)**: Puts the socket into listening mode. The 5 is the "backlog," or how many clients can be waiting in line before the server starts refusing new connections.
- **s.accept()**: This function *blocks* (pauses the script) until a new client connects. It then returns the client's connection object (conn) and their address (addr).
- **conn.sendall(b"...")**: Sends data (as bytes, note the b prefix) to a connected client.
- **conn.recv(4096)**: Receives up to 4096 bytes of data from a client.

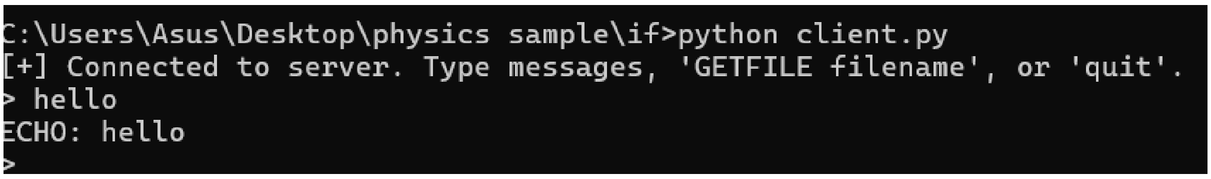
Sample Input & Output

Server Terminal:

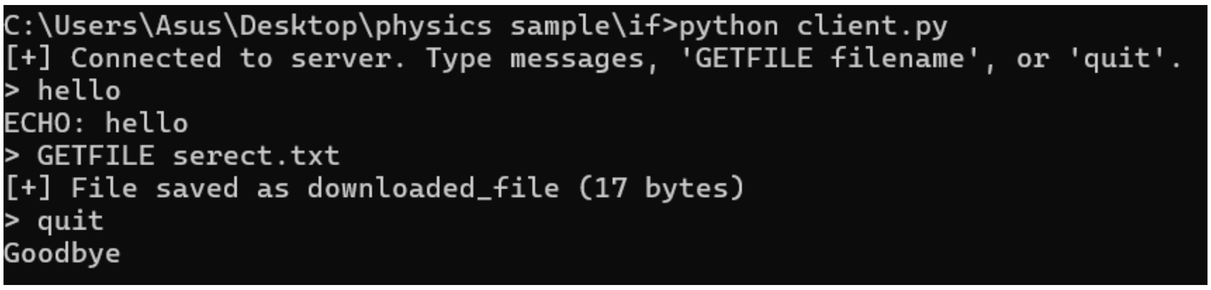


```
C:\Users\Asus\Desktop\physics sample\if>python sever.py
[+] Echo server listening on 0.0.0.0:9009
```

Client Terminal:


```
C:\Users\Asus\Desktop\physics sample\if>python client.py
[+] Connected to server. Type messages, 'GETFILE filename', or 'quit'.
> hello
ECHO: hello
>
```

Results


```
C:\Users\Asus\Desktop\physics sample\if>python client.py
[+] Connected to server. Type messages, 'GETFILE filename', or 'quit'.
> hello
ECHO: hello
> GETFILE serect.txt
[+] File saved as downloaded_file (17 bytes)
> quit
Goodbye
```

The server and client scripts work perfectly as a pair. The server successfully starts and listens for connections. The client can connect, send an arbitrary message (hello), and receive the correct "echo" response. When the client requests a file (GETFILE secret.txt), the server correctly identifies the command, reads the file, encodes it, and sends it. The client successfully receives the stream of data, decodes it, and saves it to downloaded_file.

Defensive Security Concepts

- **Path Traversal Prevention:** The server includes a vital security check.
- safe_path = os.path.abspath(os.path.join(WORKDIR, filename))
- if not safe_path.startswith(os.path.abspath(WORKDIR)):
-     # Access Denied

This prevents a malicious client from sending GETFILE ../../some_other_folder/important.txt. This attack is called "Path Traversal" and this code correctly stops it.

- **How this differs from a Shell:** Your "echo server" only responds to **pre-defined commands** (GETFILE or echo). A "shell" (like a reverse shell) is much more dangerous because it is designed to execute **any command** the client sends (e.g., whoami, ipconfig, or rm -rf /).

Conclusion

This project is a successful demonstration of the fundamental principles of client-server networking in Python. It correctly uses sockets for communication and threading to achieve concurrency, allowing it to serve multiple clients at once. It also includes a basic, but critical, security check to prevent unauthorized file access.