HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT 3

Nguyễn Đức Tâm 20210767

Topic: Building a sandbox to support malware analysis

Supervisor: MSc.Bùi Trọng Tùng

School: Information and Communications Technology

HANOI, 2024

TABLE OF CONTENTS

List of Figures

List of Tables

1 Introduction

In today's world, the Internet and digital technology are everywhere. People use computers, smartphones, and other connected devices every day for work, entertainment, shopping, and communication. While these technologies make life easier, they also bring new dangers, such as malware. Malware is a type of harmful software that attackers use to steal information, damage systems, or cause disruptions. It includes viruses, worms, ransomware, spyware, and other forms of software designed to attack or exploit systems.

Malware has become a big problem because it is constantly changing and improving. Cybercriminals create new types of malware every day, making it harder for security teams to keep up. Some malware targets businesses and governments, stealing sensitive data or causing financial loss. Other malware attacks individuals by stealing personal information, such as passwords and credit card numbers. Major attacks, such as the WannaCry ransomware incident, have caused billions of dollars in damage and affected organizations around the world. These threats show why it is important to study and understand how malware works.

The topic of building a sandbox for malware analysis is important because malware is a growing problem. A sandbox is a safe, controlled environment where malware can be tested and studied without harming real systems. When malware runs in a sandbox, it is possible to observe its actions, such as how it tries to steal data or damage files. This information helps researchers and security professionals to create better defenses against future attacks.

One reason for choosing this topic is that many existing tools for analyzing malware are either very expensive or limited in what they can do. By building a sandbox, it is possible to create a tool that fits specific needs and is more affordable or easier to use. Another reason is that some types of malware are very tricky and can hide their true behavior from traditional analysis tools. A sandbox can help uncover these hidden actions by letting the malware run naturally in a fake environment.

This project is also important because it supports the overall goal of improving cybersecurity. By studying how malware works, it is easier to stop future attacks, protect systems, and keep data safe. This project can help many people in the cyber secutiry learn more about malware and how to defend against it.

Additionally, the reason I chose this topic comes from the relevance to the work I am currently doing, as well as my future career orientation which is to become an incident response specialist and forensic investigator. At the place where I am currently working, the main job is to learn about how a malware works which can be useful in implementing more rules to detect malware in the future. Therefore, the need for a tool that can automatically collect behavior information from the malware can be very effective and productive.

Based on the realistic and personal reason above, my topic in this project is about "Building a sandbox to support malware analysis". This is a big and complex problem, but due to the limit time of this subject, in this semester I will only focus on researching the APIs that ransomware usually uses and building the monitor module from small part of the research

results. This module is also the core of the sandbox. It hooks into the malware and logs its information about the APIs of the malware for further analysis.

The structure of this report will be as follows. For the first part, I will introduce my topic and why I chose it. For the second part, there will be a requirement survey and analysis of the topic. In this part, I will mention the goals that I want to achieve in this topic, as well as the tasks that I was assigned during the development by my supervisor. I also briefly outline some advantages and disadvantages of existing tools for supporting malware analysis. In the third part, I will describe the structure of the hold sandbox and the implementation of the monitor module. The fourth part will be about testing the module. The last one will summarize all the work that I have done in this project and what I have learned through this. I also mention the future work of this topic in this part as well.

2 Requirement Survey and Analysis

2.1 Goal and Task

This topic is about developing a sandbox to support malware analysis, but as I mentioned above due to the complexity of the topic and the limited time, the goal of this project is to research the sandbox structure and focus on the APIs of one type of malware which is ransomware. Additionally, I also develop the monitor module for monitoring the behavior of the malware at runtime. The primary functionalities of the module in this project will be as follows. First, the module needs to be able to hook into the target APIs of the malware process. Secondly, the monitor needs to be able to monitor the target hooked APIs. Finally, the monitor needs to be able to log all the input parameters before and after running the APIs along with its return results in json format for further analysis.

During the process of implementing this project, my supervisor, Master Bui Trong Tung, assigned me tasks that helped me complete the proposed project goals. The first task is to learn about the topic in general, which help me thoroughly understand the topic, know what to do next, what is the purpose of this topic. The second task is to learn about the structure of the sandbox in general. The third task is to learn about the APIs that the ransomwares usually use most. The fourth task is to learn about the technique to hook into another process which help to place the monitor module into the malware process at runtime. The fifth task is implementing the monitor module based on the findings of the third and fourth tasks. The last task is to write a report that detail the progress of implementation this topic.

2.2 Pros and Cons of Some Existing Malware Analysis Tools

Malware analysis tools and sandboxes play a critical role in identifying and understanding malicious software in the cybersecurity landscape. These tools provide analysts with the ability to examine malware behavior in controlled environments, detect potential vulnerabilities, and develop effective countermeasures. While numerous tools are available, each has its strengths and weaknesses that make them suitable for specific use cases. This section evaluates three commonly used tools: Cuckoo Sandbox, VirusTotal, and ANY.RUN, listing their advantages and limitations.

For the Cuckoo Sandbox, it is a widely recognized open-source malware analysis platform that enables dynamic analysis of various file types, including executables, documents, and scripts. Its key strength lies in its flexibility and extensibility of being open source. Cuckoo allows users to customize the tool to meet their specific needs, making it a preferred choice for researchers and organizations. Additionally, it integrates well with other analysis tools, such as YARA for signature matching and Volatility for memory analysis, thereby enhancing its capabilities. However, it has several significant limitations. First, setting up and configuring Cuckoo can be overly complicated, especially for beginners. Its reliance on multiple depen-

dencies and complex installation processes can lead to errors and delays. Another limitation is its inability to fully analyze advanced malware such as ransomware that can detect when it is being executed in a sandbox environment. Such malware may hide its malicious behavior, rendering the analysis incomplete.

VirusTotal, a cloud-based malware analysis service, is another popular choice for both individual users and organizations. It aggregates results from multiple antivirus engines and sandbox environments, providing a quick and comprehensive overview of a file's potential threats. One of its primary advantages is its ease of use. Being a web-based platform, VirusTotal requires no installation, making it accessible to users regardless of their technical expertise. However, its primary limitation is its lack of depth. VirusTotal relies on antivirus engines and basic sandbox tools to provide results, which means it cannot offer the detailed behavioral insights required for advanced malware analysis. Additionally, VirusTotal's policy of sharing uploaded files publicly raises serious privacy concerns. Users uploading sensitive or proprietary files risk exposing confidential information. Finally, the tool does not allow users to modify or customize its functionality, making it a poor choice for specialized or in-depth malware studies. These limitations make VirusTotal more suitable as a quick scanning tool.

ANY.RUN is a real-time interactive malware analysis platform that offers dynamic and user-friendly features. Unlike other sandboxes, ANY.RUN allows analysts to interact directly with the environment during the analysis process, enabling them to explore malware behavior more comprehensively. This interactivity is particularly beneficial for analyzing malware that requires user actions, such as clicking on links or opening attachments. The tool's detailed reports make it accessible to every users. Additionally, ANY.RUN supports a wide range of file types and operating systems. However, it also has some limitations. The tool is not free, and its advanced features are locked for a subscription. This makes it inaccessible to everyone. Furthermore, while ANY.RUN is dynamic analysis, it lacks strong capabilities for static analysis, such as inspecting the malware's code or structure without execution. This limitation can prevent users from fully understanding the inner workings of certain types of malware. Additionally, like other sandbox environments, advanced malware can sometimes bypass detection or avoid exhibiting its full range of malicious behavior when running inside ANY.RUN.

while Cuckoo Sandbox, VirusTotal, and ANY.RUN each provide valuable features for malware analysis, they also have distinct limitations. Cuckoo is customizable and integration but requires significant resources, expertise and lacking of update. VirusTotal offers quick and collaborative threat detection but lacks analysis capabilities and raises privacy concerns. ANY.RUN is interactivity and user-friendly interface but comes with cost barriers and limited static analysis features. These limitations also highlight the need for continuous improvement in malware analysis technologies to address current limitations and enhance effectiveness.

3 Implementation

3.1 Sandbox Architecture

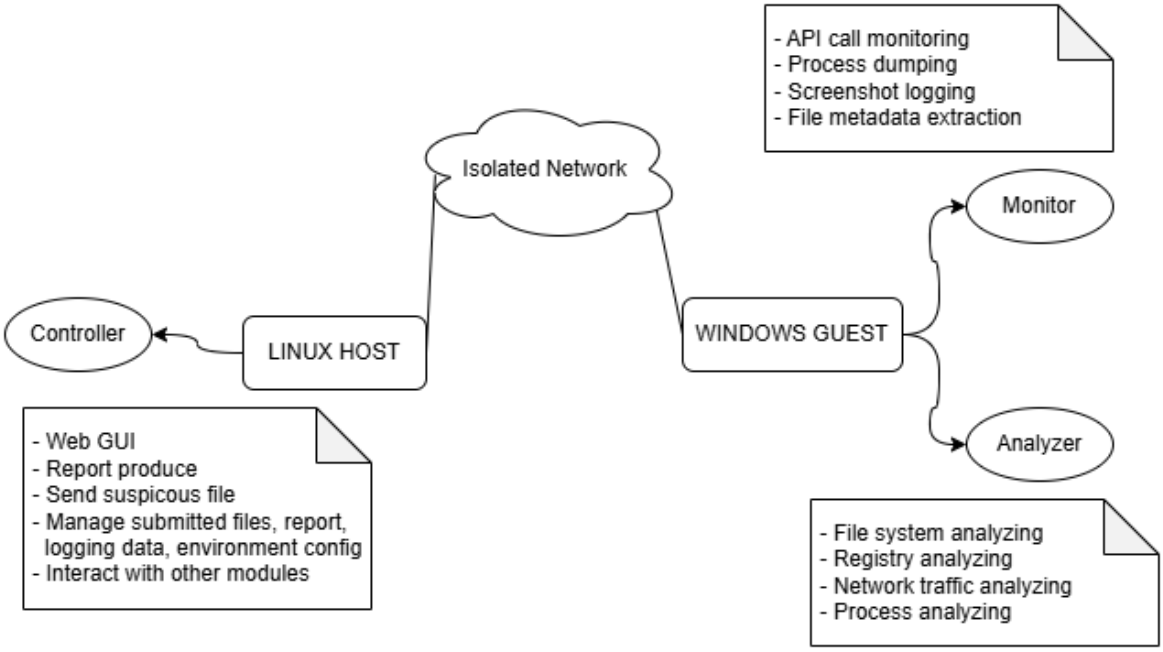

Fig. 1. Sandbox Architecture

The sandbox architecture consists of two interconnected machines. The host machine, running a Linux operating system, serves as the central management and control point for the sandbox environment. It contains a crucial module known as the Controller, which is responsible for controlling the entire sandbox workflow. The Controller provides a web-based graphical user interface (Web GUI) that allows users to interact with the sandbox efficiently. Through this interface, users can submit potentially malicious files for analysis, review detailed reports generated by the system, and manage various aspects of the sandbox environment. The Controller also handles the storage and tracking of submitted files, ensuring each file is properly logged and associated with its respective analysis report. The Controller is responsible for managing the configuration of the entire sandbox system. This includes setting up virtual machines, configuring the isolated network, and defining the environment in which the malicious files are executed.

The guest machine, running a Windows operating system, serves as the execution and analysis environment for the submitted malicious files. It is specifically designed to emulate a real-world system, allowing malware to operate as intended within a controlled and secure environment. This machine is isolated from the host and external networks to ensure that any potentially harmful activity is contained. The guest machine contains two key modules, Monitor and Analyzer, which work together to execute and analyze the behavior of the

malicious file.

The Monitor module is the first stage in the guest machine's analysis pipeline and is responsible for actively monitoring and capturing the behavior of the malicious file during its execution. The module intercepts and logs all system API calls made by the malicious file. This includes function calls to the Windows API related to file operations, network communications, process creation, memory manipulation, and registry modifications. These logs are critical for understanding how the malware interacts with the system. Next, the Monitor module captures the state of the process memory at various stages of execution. This allows the system to identify any code injections, unpacked payloads, or dynamically loaded modules that the malware may utilize. The third task of the module is, periodically take screenshots of the system's graphical interface to document any visual changes caused by the malware, such as pop-up messages or fake error screens. The module also extracts metadata from files. This includes file names, paths, sizes, timestamps, and hash values, providing forensic details for further analysis. The Monitor module functions as the foundation of data collection and the output of this module serves as the raw input for the Analyzer module.

The Analyzer module processes the data collected by the Monitor module, applying filtering and parsing to extract meaningful information about the malicious file. The Analyzer analyzes registry modifications performed by the malware. For instance, the Analyzer may detect attempts to create persistent entries, such as adding keys to the "Run" registry path for automatic execution upon system startup. It also inspects any network communications initiated by the malware to see if it communicate with any commandandcontrol server. After completing its analysis, the Analyzer module generates a report summarizing the behavior posed by the malicious file. This report is securely transmitted back to the Controller module on the host machine for further processing and user presentation.

The communication between the host machine and the guest machine occurs over an isolated network, which is a critical security feature of the sandbox architecture. This isolated network ensures that any potentially malicious actions performed by the analyzed files on the guest machine cannot propagate beyond the sandbox environment. This setup guarantees that even if the malicious file attempts to establish a network connection (e.g., to exfiltrate data or communicate with a command-and-control server), these attempts will remain inside the sandbox. Moreover, the network isolation ensures that the host machine is shielded from malware-related risks, such as infection, unauthorized access, or data compromise. The isolated network is also configured to allow the Controller module on the host machine to securely send files and commands to the Monitor and Analyzer modules on the guest machine. Simultaneously, it facilitates the return of analysis results and logs from the guest machine back to the host. This controlled, bidirectional communication ensures seamless interaction between the two machines. Additionally, the isolated network also simulate specific network conditions, such as fake DNS servers, HTTP servers, to observe the malware's behavior in a more natural context. This additional layer helps in identifying network-based malicious actions, such as data exfiltration or propagation attempts.

3.2 API Hooking Technique with Detours on Windows

In the process of malware analysis, understanding how malicious software interacts with system resources is crucial. One powerful technique to gain information about these behaviors is API hooking. By intercepting and modifying the behavior of application programming interfaces (APIs), analysts can monitor or alter the execution flow of a program, providing valuable information about its behavior. In this section, I will describe the API hooking technique in general and the API hooking technique that Detours library implements, a widely used method.

3.2.1 API Hooking Concepts

Before diving into the question, what is API hooking? An understanding of the basic flow that a normal process in Windows runs and calls APIs is needed. In Windows, when a program is run, the loader loads the executable file of it into the memory and the DLLs (Dynamic Link Libraries) are also loaded. DLLs are files that contain code that is meant to be shared between programs. Therefore, many Windows APIs are provided through these DLLs, some of them are very common such as kernel32.dll and user32.dll. When a program needs to use a function from a DLL, it calls that function based on information in the DLL's export table. This table includes the names of the functions and their locations within the memory where the DLL is loaded. Once the correct address is found, the program executes the function from that location. So what if we can modify the functionality of the exported functions to get more controll over the running program. This is where the API hooking technique comes in place.

API hooking is a technique that changes the behavior of these functions. It does this by replacing the first few bytes of the target function with some new instructions (also called opcodes). These new instructions redirect the execution to a different address chosen by the developer. As a result, when the function is called, it first jumps to this new address before continuing. This redirection allows the program to pass the execution to a custom function instead of the original. The custom function can perform additional tasks, such as logging or modifying data, before deciding whether to pass control back to the original API function. This approach allow developers to add new functionality to a Windows API without permanently changing the original function. Unhooking, on the other hand, is the process of removing the hook. It restores the function to its original state by replacing the modified instructions with the ones that were there before. For unhooking to work, the original opcodes must be saved when the function is first hooked so they can be restored later. This ensures that the function behaves exactly as it did before hooking.

3.2.2 Detours Library

Detours is a library designed to intercept binary functions on various machine architectures, including ARM, ARM64, X86, X64, and IA64. It is commonly used to intercept Win32 API

calls in applications. This interception happens dynamically while the program is running.

In a normal call to the exported functions, there are two jump taking place. The first jump redirects the flow of the program to the exported function. The second jump takes place at the end of the exported functions to redirect the flow back to the caller.

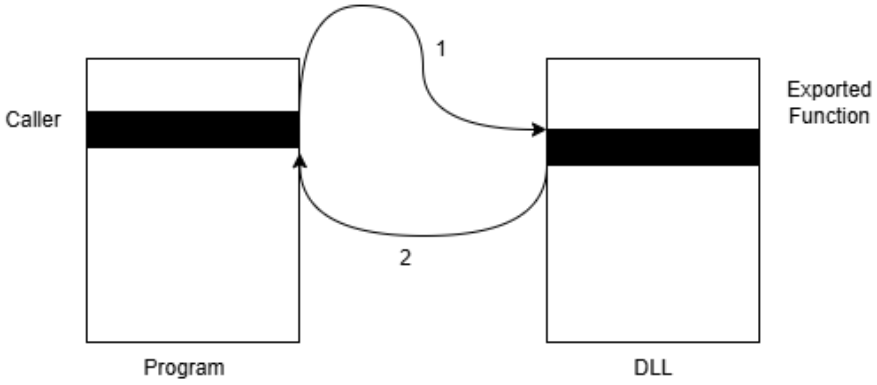

Fig. 2. Normal Call

But if the exported functions have been hooked by Detours, the whole process would look very different. The new flow will be as follows. First, the caller calls and jumps to the exported function as normal. However, since the exported function has been hooked which means the first few bytes of its has been replaced with jump instructions that redirect the flow to the hook function. These new first few bytes are called the "trampoline" due to its main job of immediately jumping to the hook function. This is the second jump. Next, the hook function does what it is supposed to do and jumps to the original exported function. However, the first few bytes of the exported function have been replaced with jumps instruction to jump to hook function, therefore, such a call would cause an infinite loop. To solve this problem, Detours saves these first few bytes of the exported function into other place in memory. As the hook function wants to jump to the original, it can jump to this place. This is the third jump. Detours also adds a jump at the end of the place that stores the first few bytes of the exported function so that it can continue the process of the original exported function. So, the fourth jump takes place right at the end of this place. From there, after the exported function completes its execution, it returns to the hook function in the fifth jump. This is because the hook function is the actual function that has called the exported function. The hook function then continuously does what it is supposed to do and jumps back to the original caller in the sixth jump.

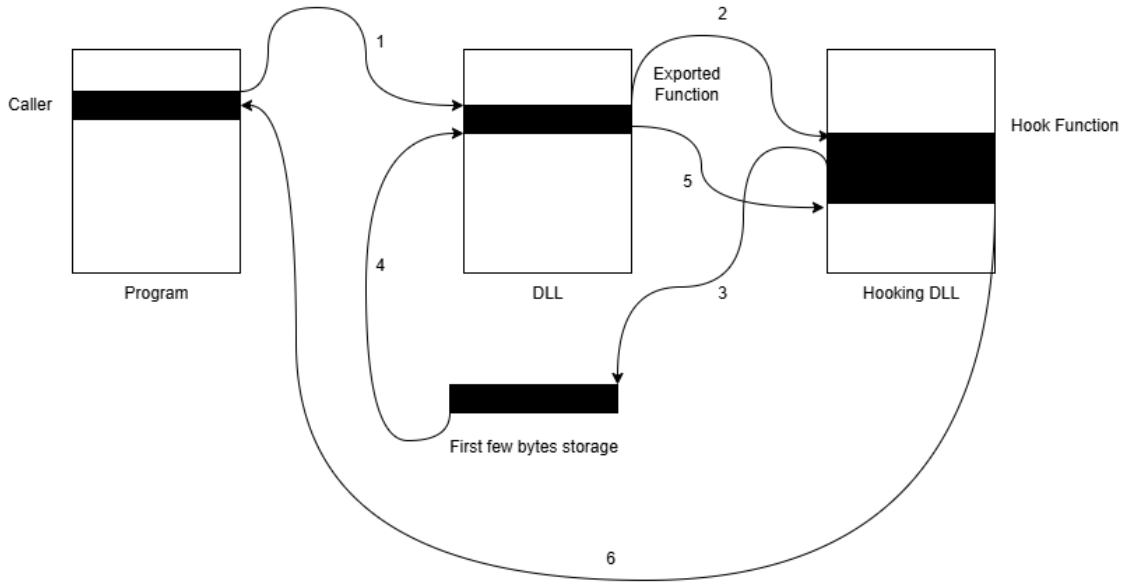

Fig. 3. Hooked Call

To perform hooking with Detours, I utilize the following functions `DetourTransactionBegin`, `DetourUpdateThread`, `DetourAttach` and `DetourTransactionCommit`. Detours works in the transaction manner which means that every hookings is cached untill committed. The function `DetourTransactionBegin` is used to begin a transaction, and the function `DetourTransactionCommit` is used to commit a transaction. The function `DetourUpdateThread` is used to specify the thread to be used to perform the hooking. The actual function which performs the hooking is `DetourAttach`. This function takes two arguments - a double pointer to the target function and the hook function address. For the first argument, Detours will use it in two things. First, it will use this address to locate the location of the target function. Detours then copies these first few bytes of the target function and stores them to another place. Then, it overwrites the first few bytes with a jump instruction to jump to the hook function. The second thing that Detours will do with this address is incrementing the argument to point to the above copy of the first few original bytes. This is done so that the hook function can still call the original function in the future. For the second argument, it is used to be the operand of the jump instruction in the above overwriting jump instruction. Once the hooking is done, any calls to the original target function will redirect to the hook function as described in the previous.

3.3 Monitor Component

As explained in the introduction, due to the complexity of developing a full-featured sandbox and the constraints of time, I have chosen to focus on implementing a specific and limited component of the overall system. In particular, I will concentrate on a subset

of the monitor module, which is tasked with tracking specific API calls that ransomware might leverage during its encryption-decryption process. The APIs I have selected for monitoring are critical functions provided by Microsoft for encryption tasks. These include `CryptAcquireContextA`, `CryptAcquireContextA`, `CryptHashData`, `CryptDeriveKey`, `CreateFileA`, `CryptEncrypt` and `CryptDecrypt`. These functions form the backbone of Microsoft's cryptographic services, enabling applications to perform secure encryption and decryption operations. Unfortunately, these same functions are frequently exploited by malicious software, especially ransomware, to encrypt user data. By monitoring these APIs, my implementation will aim to detect behaviors characteristic of ransomware, such as the creation of encryption keys, hashing of data, derivation of cryptographic keys, and file encryption or decryption. This targeted approach provides a practical starting point for building the sandbox.

The approach I used to monitor these APIs involves capturing and logging critical data before and after each API call is executed. Specifically, I log the input parameters before API call, the output parameters after the API call, and the return value of the call itself. This method ensures that all relevant data about the API's behavior is recorded, providing complete insights into its usage. To implement this functionality, I created a `LogEntry` struct, which serves as a container for the recorded information. The `LogEntry` struct is defined as follows:

```
struct LogEntry {
    std::string apiName;
    std::string pre_call_parameters;
    std::string post_call_parameters;
    std::string result;

    std::string toJSON() const {
        std::ostringstream json;
        json << "{";
        json << "\"apiName\": \"" << apiName << "\", ";
        json << "\"pre_call_parameters\": " <<
           pre_call_parameters << ", ";
        json << "\"post_call_parameters\": " <<
           post_call_parameters << ", ";
        json << "\"result\": " << result;
        json << "}";
        return json.str();
    }
};
```

Each element in the `LogEntry` struct will have the following meaning:

- `apiname`: Stores the name of the API being monitored, allowing easy identification of which function is being logged.
- `pre_call_prameters`: Contains the values of the API's input parameters before the function is invoked. This helps in understanding the context in which the API was called.
- `post_call_parametes`: Captures the values of the parameters after the API call is executed, providing insights into how the call may have modified the input data.
- `result`: Logs the return value of the API call, which is crucial for identifying the outcome or behavior of the function.

To create efficient storage and analysis, the `LogEntry` struct includes a toJSON method. This method converts the struct's data into a JSON-formatted string, making it easier to store and analyze the logs in a structured format. By implementing this structured logging mechanism, I aim to capture a detailed record of each monitored API's behavior. This provides valuable data for analyzing potential malicious activities, particularly in scenarios where ransomware or other malware exploits these APIs for cryptographic operations.

To implement API hooking, a critical component is the hook functions for each targeted API. These hook functions act as intermediaries between the program and the actual Windows API, allowing us to intercept and monitor API calls. Each hook function is designed to capture and log details about the API's execution, providing a complete view of its behavior. The execution flow of a hook function can be described as follows. The hook function is triggered whenever the corresponding API is called by the program. The hook function then perform its pre-hook functionality which captures and logs all the input parameters provided to the API. After logging the pre-call parameters, the hook function passes control to the actual system API. This ensures that the original functionality of the API remains unaffected. Once the API call is complete, the hook function performs its post-hook functionality which logs the state of the parameters after execution. The hook function then captures and logs the return value of the API. This return value provides crucial information about the outcome of the API call. All the logged information, including the pre-call parameters, post-call parameters, and return value, is filled into a log entry object. This log entry is then serialized in JSON format and saved to a file for further analysis.

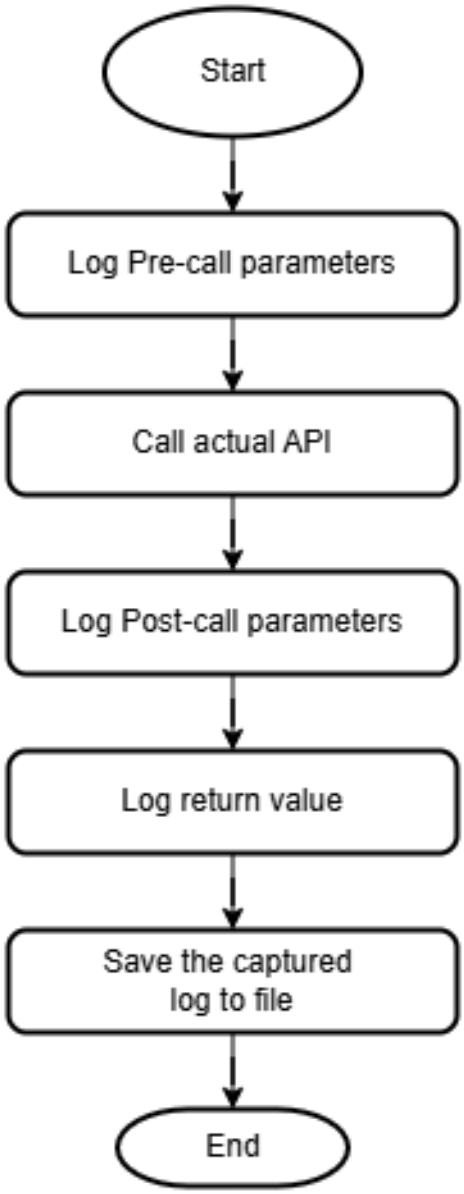

Fig. 4. Hook Function Execution Flow

3.3.1 Installing Hook

To install the hook function for each target API, I have created a function named `InstallHook` to do this task. The flow of it is describe as follow. First, a Detour transaction is initiated with `DetourTransactionBegin()`, which sets up the environment for applying hooks. If this step fails, the function exits early, returning FALSE. Next, the thread in which the hooks will be applied is updated using `DetourUpdateThread(GetCurrentThread())`. This step ensures that the current thread is prepared for modifications. The function then iterates through the originals array, which contains pointers to the original

API functions, and the corresponding hooks array, which contains pointers to the custom hook functions. For each pair, it calls `DetourAttach()` to attach the hook function to the respective original API. If any attachment fails, the function aborts the transaction and returns FALSE. Finally, the function commits the Detour transaction with `DetourTransactionCommit()`, making the hooks active. If this step is successful, the function returns TRUE, signaling that all hooks have been installed correctly. Otherwise, it returns FALSE, indicating an error in the transaction process. By following this structured approach, the `InstallHooks` function ensures a reliable and systematic setup of API hooks for further analysis or modification.

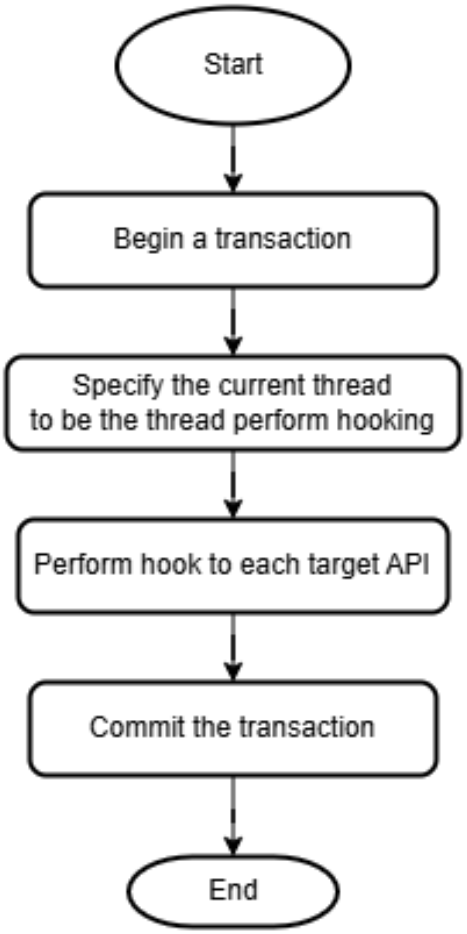

Fig. 5. Install Hook Function Execution Flow

3.3.2 CryptAcquireContextA Hooking

The first target API function is `CryptAcquireContextA`. This function is for finding a cryptographic service provider (CSP). If the CSP is found, the function attempts to find a key container within the CSP based on its parameters. With the appropriate setting, this function can also create and destroy key containers and can provide access to a CSP with a

temporary key container if access to a private key is not required.

```
BOOL CryptAcquireContextA(
  [out] HCRYPTPROV *phProv,
  [in]  LPCSTR     szContainer,
  [in]  LPCSTR     szProvider,
  [in]  DWORD      dwProvType,
  [in]  DWORD      dwFlags
);
```

Brief meaning of each parameter is described in the table below.

Parameter Name	**Description**
phProv	A pointer to a handle of a CSP.
szContainer	The key container name. This is a null-terminated string that identifies the key container to the CSP.
szProvider	A null-terminated string that contains the name of the CSP to be used.
dwProvType	Specifies the type of provider to acquire.
dwFlags	Specifies the flag value. This parameter is usually set to zero.

Table 1: CryptAcquireContextA Parameter Description

To serve the purpose of hooking `CryptAcquireContextA` function, a hook function named `Hook_CryptAcquireContextA` is created. The `Hook_CryptAcquireContextA` function intercepts the `CryptAcquireContextA` API call to log its input parameters, output parameters, and result for monitoring purposes. The flow begins by initializing a `LogEntry` structure and logging the pre-call parameters, including pointers, container names, provider names, and flags, in a structured JSON format. This involves converting the values into a human-readable hexadecimal representation for precision. Next, the actual `CryptAcquireContextA` API call is executed using the original function stored in `CryptAcquireContextAActual`. Following the call, the function logs the post-call state of the parameters, including any modifications to pointers or memory. Finally, the return result of the API is recorded in the log. The complete log entry, which contains the API name, pre-call parameters, post-call parameters, and result, is then converted to JSON format and appended to a file `api_logs.json`. The function returns the result of the original API call, ensuring the functionality of the intercepted API remains unchangeable.

```
BOOL Hook_CryptAcquireContextA(HCRYPTPROV* phProv, LPCSTR
   szContainer, LPCSTR szProvider, DWORD dwProvType, DWORD
   dwFlags) {
    //Initialize LogEntry object for storing logs
    // Pre-call parameter logging
    // Actual API call
    BOOL result = CryptAcquireContextAActual(phProv,
       szContainer, szProvider, dwProvType, dwFlags);
    // Post-call parameter logging
    // Result logging
    // Log the entry to the file "api_logs.json"
    return result;
}
```

3.3.3 CryptCreateHash Hooking

The second target Windows API is `CryptCreateHash`. The CryptCreateHash function initiates the hashing of a stream of data. It creates and returns to the calling program a handle to a cryptographic service provider hash object. This handle is used in subsequent calls to CryptHashData and CryptHashSessionKey to hash session keys and other streams of data.

```
BOOL CryptCreateHash(
  [in]  HCRYPTPROV hProv,
  [in]  ALG_ID     Algid,
  [in]  HCRYPTKEY  hKey,
  [in]  DWORD      dwFlags,
  [out] HCRYPTHASH *phHash
);
```

The meaning of each parameter is briefly described in the table below.

Parameter Name	Description
hProv	This is a handle to a CSP created by a previous call to CryptAcquireContext.
Algid	Specify the hash algorithm to use. Valid values for this parameter vary, depending on the CSP that is used
hKey	Specify the key using in the hash if the hash is keyed type such as Hash-Based Message Authentication Code (HMAC) or Message Authentication Code (MAC) algorithm.
dwFlags	This parameter is not using anymore.
phHash	A pointer points to a handle to the new hash object

Table 2: CryptCreateHash Parameter Description

The function `Hook_CryptCreateHash` is used as the hook function for the `CryptCreateHash` API. The flow begins by initializing a `LogEntry` structure and recording the pre-call parameters, including the cryptographic provider handle `hProv`, algorithm identifier `Algid`, cryptographic key handle `hKey`, flags `dwFlags`, and hash handle pointer `phHash`. These values are formatted into a JSON-compatible string for structured logging. The actual `CryptCreateHash` API call is then executed using the original function stored in `CryptCreateHashActual`. After the call, the function logs the post-call state of the parameters, capturing any changes to pointers or handles. The return result of the API call is also logged. Finally, the complete log entry, including the API name, pre-call parameters, post-call parameters, and result, is converted to JSON format and appended to a log file `api_logs.json`. The function ensures the behavior of the intercepted API remains unaffected by returning the original API's result.

```
BOOL Hook_CryptCreateHash(HCRYPTPROV hProv, ALG_ID Algid,
   HCRYPTKEY hKey, DWORD dwFlags, HCRYPTHASH* phHash)
{
    //Initialize LogEntry object for storing logs
    // Pre-call parameter logging
    // Actual API call
    BOOL result = CryptCreateHashActual(hProv, Algid, hKey,
       dwFlags, phHash);
    // Post-call parameter logging
    // Result logging
    // Log the entry to the file "api_logs.json"

    return result;
}
```

3.3.4 CryptHashData Hooking

The third Windows API that is targeted is the `CryptHashData`. The main job of this function is to fill the hash data provided by the program to the hash object created in the previous call of `CryptCreateHash`.

```
BOOL CryptHashData(
  [in] HCRYPTHASH hHash,
  [in] const BYTE *pbData,
  [in] DWORD      dwDataLen,
  [in] DWORD      dwFlags
);
```

The short explanation of each parameter is listed in the table below.

Parameter Name	Description
hHash	Specify the handle of the hash object. This is a handle created by a previous call to CryptCreateHash.
pbData	Specify a pointer to a buffer that contains the data to be added to the hash object.
dwDataLen	Specify the number of bytes of data to be added.
dwFlags	This parameter is not using anymore.

Table 3: CryptHashData Parameter Description

The `Hook_CryptHashData` function is created to intercept the `CryptHashData` API call to log details about its execution. It starts by creating a `LogEntry` object and logging the pre-call parameters, including the hash handle `hHash`, pointer to the data buffer `pbData`, data length `dwDataLen`, and flags `dwFlags`. The data buffer is converted into a hexadecimal string for detailed logging. The actual `CryptHashData` function is then called using the original implementation stored in `CryptHashDataActual`, and its result is captured. The post-call parameters are identical to the pre-call parameters and are also logged. The function's result, along with all the parameters and execution details, is stored in a JSON-compatible format and appended to a log file `api_logs.json`. Finally, the function returns the result of the original API call, ensuring normal operation while enabling comprehensive monitoring.

```
BOOL Hook_CryptHashData(HCRYPTHASH hHash, const BYTE* pbData,
    DWORD dwDataLen, DWORD dwFlags)
{
    // Initialize LogEntry object for storing logs
    // Pre-call parameter logging
    // Actual API call
    BOOL result = CryptHashDataActual(hHash, pbData,
       dwDataLen, dwFlags);
    // Post-call parameter logging
    // Result logging
    // Log the entry to the file "api_logs.json"

    return result;
}
```

3.3.5 CryptDeriveKey Hooking

The `CryptDeriveKey` API is the fourth target function that I aim to. This is very important function. Because it generates cryptographic keys from the hash data provided in the previous steps. This function also guarantees that when the same cryptographic service provider and algorithms are used, the keys generated from the same base data are identical.

```
BOOL CryptDeriveKey(
  [in]       HCRYPTPROV hProv,
  [in]       ALG_ID     Algid,
  [in]       HCRYPTHASH hBaseData,
  [in]       DWORD      dwFlags,
  [in, out] HCRYPTKEY  *phKey
);
```

The description of each paremeter is defined in the table below.

Parameter Name	Description
hProv	Specify a handle of a CSP created by a call to CryptAcquire-Context.
Algid	Specify the symmetric encryption algorithm for which the key is to be generated.
hBaseData	Specify the handle of the hash object which has been filled with hash data to be used as base data.
dwFlags	Specifies the type of key generated.
phKey	Specifies the pointer to the handle of the newly created key.

Table 4: CryptDeriveKey Parameter Description

The hook function for the `CryptDeriveKey` is the function named `Hook_Crypt-DeriveKey`. This function intercepts the `CryptDeriveKey` API to log its execution details while maintaining normal functionality. It begins by creating a `LogEntry` and logs pre-call parameters, including the cryptographic service provider handle `hProv`, algorithm ID `Algid`, base hash handle `hBaseData`, flags `dwFlags`, and the key handle pointer `phKey`. These values are formatted as a JSON-compatible string. The original API function which stored in the variable `CryptDeriveKeyActual` is then called with the provided arguments, and its result is stored. After the call, the post-call parameters are logged, reflecting any changes made by the API. The result, along with the pre- and post-call data, is serialized into a JSON entry and appended to a log file `api_logs.json`. Finally, the function returns the result of the original API, ensuring the API call is executed as intended while capturing detailed runtime information

```
BOOL Hook_CryptDeriveKey(HCRYPTPROV hProv, ALG_ID Algid,
   HCRYPTHASH hBaseData, DWORD dwFlags, HCRYPTKEY* phKey)
{
    // Initialize LogEntry object for storing logs
    // Pre-call parameter logging
    // Actual API call
    BOOL result = CryptDeriveKeyActual(hProv, Algid,
       hBaseData, dwFlags, phKey);
    // Post-call parameter logging
    // Result logging
    // Log the entry to the file "api_logs.json"

    return result;
}
```

3.3.6 CryptEncrypt Hooking

The fifth API that I perform hooking is the `CryptEncrypt` function. This function as its name describes is used to perform encryption on data which converts the plaintext data to ciphertext data. This is a very crucial function which serves as the main part of the encryption process.

```
BOOL CryptEncrypt(
  [in]       HCRYPTKEY  hKey,
  [in]       HCRYPTHASH hHash,
  [in]       BOOL       Final,
  [in]       DWORD      dwFlags,
  [in, out] BYTE        *pbData,
  [in, out] DWORD       *pdwDataLen,
  [in]       DWORD      dwBufLen
);
```

A short description about the `CryptEncrypt` function is in the table below.

Parameter Name	Description
hKey	Specify a handle to the encryption key created by a call to CryptDeriveKey.
hHash	Specify a handle to a hash object created by a call to CryptCreateHash.
Final	A Boolean value that specifies whether this is the last section in a series being encrypted.
dwFlags	Reserved for future use.
pbData	A pointer to a buffer that contains the plaintext to be encrypted. The plaintext in this buffer is overwritten with the ciphertext created by this function.
pdwDataLen	A pointer to a DWORD value that, before execution, contains the length, in bytes, of the plaintext in the pbData buffer. After execution, this DWORD contains the length, in bytes, of the ciphertext written to the pbData buffer.
dwBufLen	Specifies the total size, in bytes, of the input pbData buffer.

Table 5: CryptEncrypt Parameter Description

The `Hook_CryptEncrypt` function hooks into the `CryptEncrypt` API to log detailed execution data while preserving the API's functionality. It begins by creating a `LogEntry` and logs pre-call parameters, including the cryptographic key handle `hKey`, hash handle `hHash`, final block indicator `Final`, flags `dwFlags`, data buffer `pbData`, data length pointer `pdwDataLen`, and buffer length `dwBufLen`. These parameters are serialized into a JSON-compatible string for logging. The original `CryptEncrypt` function which

is store in `CryptEncryptActual` is then invoked with the provided arguments, and its result is captured. Post-call parameters are similarly logged, reflecting any changes made to the data during the encryption process. The result, along with the pre- and post-call data, is serialized into a JSON entry and appended to a log file `api_logs.json`. Finally, the function returns the result of the original API call.

```
BOOL Hook_CryptEncrypt(HCRYPTKEY hKey, HCRYPTHASH hHash, BOOL
    Final, DWORD dwFlags, BYTE* pbData, DWORD* pdwDataLen,
   DWORD dwBufLen)
{
    // Initialize LogEntry object for storing logs
    // Pre-call parameter logging
    // Actual API Call
    BOOL result = CryptEncryptActual(hKey, hHash, Final,
       dwFlags, pbData, pdwDataLen, dwBufLen);
    // Post-call parameter logging
    // Result logging
    // Log the entry the file "api_logs.json"

    return result;
}
```

3.3.7 CryptDecrypt Hooking

The next hooked function is `CryptDecrypt`. This function is reverse with the previous API function. This function is used to convert the ciphertext into plaintext by using the same key used in the previous encryption process.

```
BOOL CryptDecrypt(
  [in]       HCRYPTKEY  hKey,
  [in]       HCRYPTHASH hHash,
  [in]       BOOL       Final,
  [in]       DWORD      dwFlags,
  [in, out] BYTE       *pbData,
  [in, out] DWORD      *pdwDataLen
);
```

The brief description of the input parameters of the function is shown in the table below.

Parameter Name	Description
hKey	Specify a handle to the encryption key created by a call to CryptDeriveKey.
hHash	Specify a handle to a hash object created by a call to CryptCreateHash.
Final	A Boolean value that specifies whether this is the last section in a series being encrypted.
dwFlags	Can be one of two values CRYPT_OAEP or CRYPT_DECRYPT_RSA_NO_PADDING_CHECK
pbData	A pointer to a buffer that contains the data to be decrypted. After the decryption has been performed, the plaintext is placed back into this same buffer.
pdwDataLen	A pointer to a DWORD value that, before execution, contains the length, in bytes, of the ciphertext in the pbData buffer. After execution, this DWORD contains the length, in bytes, of the plaintext written to the pbData buffer.

Table 6: CryptDecrypt Parameter Description

The `Hook_CryptDecrypt` function is created to intercept and log the execution of the `CryptDecrypt` API. It initializes a `LogEntry` object, setting the API name to "CryptDecrypt," and captures the pre-call parameters, including the cryptographic key handle `hKey`, hash handle `hHash`, final block indicator `Final`, flags `dwFlags`, data buffer `pbData`, and data length pointer `pdwDataLen`. These parameters are formatted into a JSON-compatible string for logging. The actual decryption process is then performed by invoking the original `CryptDecrypt` function stored in `CryptDecryptActual`, and its result is recorded. Post-call parameters are logged similarly, capturing any changes to the buffer or its length. The result of the decryption call, along with the logged parameters, is serialized into a JSON entry and appended to a log file `api_logs.json`. The function then returns the result of the original API call, maintaining the original functionality while providing detailed runtime monitoring.

```
BOOL Hook_CryptDecrypt(HCRYPTKEY hKey, HCRYPTHASH hHash, BOOL
    Final, DWORD dwFlags, BYTE* pbData, DWORD* pdwDataLen)
{
    // Initialize LogEntry object for storing logs
    // Pre-call parameter logging
    // Actual API Call
    BOOL result = CryptDecryptActual(hKey, hHash, Final,
       dwFlags, pbData, pdwDataLen);
    // Result logging
    // Log the entry the file "api_logs.json"

    return result;
}
```

3.3.8 CreateFileA Hooking

The final target function which I focus on in this project is the `CreateFileA` function. It is used to open or create vaious type of file or I/O device. This function is very widely used by both normal program and malware program. Especially, for ransomware, to encrypt or decrypt the data of the compromised system, the very first steps of it must be open the file to read the data from it.

```
HANDLE CreateFileA(
  [in]           LPCSTR                lpFileName,
  [in]           DWORD                 dwDesiredAccess,
  [in]           DWORD                 dwShareMode,
  [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  [in]           DWORD                 dwCreationDisposition,
  [in]           DWORD                 dwFlagsAndAttributes,
  [in, optional] HANDLE                hTemplateFile
);
```

A short description explaining the purpose of each input parameter is defined in the table below.

Parameter Name	Description
lpFileName	Specify the name of the file or device to be created or opened.
dwDesiredAccess	Specify the requested access to the file or device, which can be summarized as read, write, both or 0 to indicate neither.
dwShareMode	Specify the requested sharing mode of the file or device, which can be read, write, both, delete, all of these, or none.
lpSecurityAttributes	Specify a pointer to a SECURITY_ATTRIBUTES structure. But normally be NULL.
dwCreationDisposition	Specify to whether open or create the file in case it already exists.
dwFlagsAndAttributes	Specify the file or device attributes and flags, FILE_ATTRIBUTE_NORMAL being the most common default value for files.
hTemplateFile	Specifies a valid handle to a template file with the GENERIC_READ access right. But most commonly used value is NULL.

Table 7: CreateFileA Parameter Description

The hook function of the `CreateFileA` is a function named `Hook_CreateFileA`. It intercepts and logs the execution of the `CreateFileA` API call. It begins by initializing a `LogEntry` object and setting the API name to "CreateFileA." Pre-call parameters, including the file name `lpFileName`, desired access flags `dwDesiredAccess`, share mode `dwShareMode`, security attributes `lpSecurityAttributes`, creation disposition `dwCreationDisposition`, flags and attributes `dwFlagsAndAttributes`, and template file handle `hTemplateFile`, are captured and serialized into a JSON-compatible string. If the file name is provided, it is logged as a hexadecimal-encoded string. The function then calls the original `CreateFileA` API which is stored in the variable named `CreateFileAActual` and stores the resulting handle. The pre-call parameters are reused as post-call parameters since no modifications occur during the API execution. Finally, the result and logged parameters are serialized into JSON format and appended to the `api_logs.json` file. The function returns the handle obtained from the original API call, preserving the behavior while providing detailed logging for monitoring and analysis.

```
HANDLE Hook_CreateFileA(LPCSTR lpFileName, DWORD
   dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ATTRIBUTES
   lpSecurityAttributes, DWORD dwCreationDisposition, DWORD
   dwFlagsAndAttributes, HANDLE hTemplateFile)
{
    // Initialize the LogEntry for storing logs
    // Pre-call parameter logging
    // Actual API call
    HANDLE result = CreateFileAActual(lpFileName,
       dwDesiredAccess, dwShareMode, lpSecurityAttributes,
       dwCreationDisposition, dwFlagsAndAttributes,
       hTemplateFile);
    // Result logging
    // Log the entry the file "api_logs.json"

    return result;
}
```

4 Testing

In this section, I will conduct a comprehensive test case evaluation of the implemented Monitor module to verify its functionality and effectiveness in capturing and logging APIs behavior.

4.1 Description

To test the functionality of the Monitor module, I have implemented a testing program. The flow execution of its as follows. The program begins execution by defining a file path pointing to `testfile.txt`, which will be processed for encryption and decryption. The LoadLibraryA function is called to load a dynamic link library `Monitor.dll` which is the implemented Monitor module. Following this setup, the `EncryptDecryptFile` function is called with the file path as its argument. Inside this function, the specified file is opened using the Windows API `CreateFileA` with read and write access. A cryptographic context is then initialized using `CryptAcquireContextA`, allowing the program to utilize cryptographic services. A hash object is created with `CryptCreateHash`, and dummy data is hashed using `CryptHashData` to generate a key with `CryptDeriveKey`. The program reads the file's content into a buffer and encrypts it using `CryptEncrypt`, ensuring sufficient buffer size for encryption padding. The encrypted data is then written back to the file. Subsequently, the program decrypts the same data using `CryptDecrypt`, verifying the encryption and decryption processes. Finally, all allocated resources, including cryptographic keys, hashes, contexts, and file handles, are cleaned up before the program exits. This flow demonstrates file encryption and decryption using Windows CryptoAPI, while dynamically loading the Monitor DLL to monitor the API behaviors.

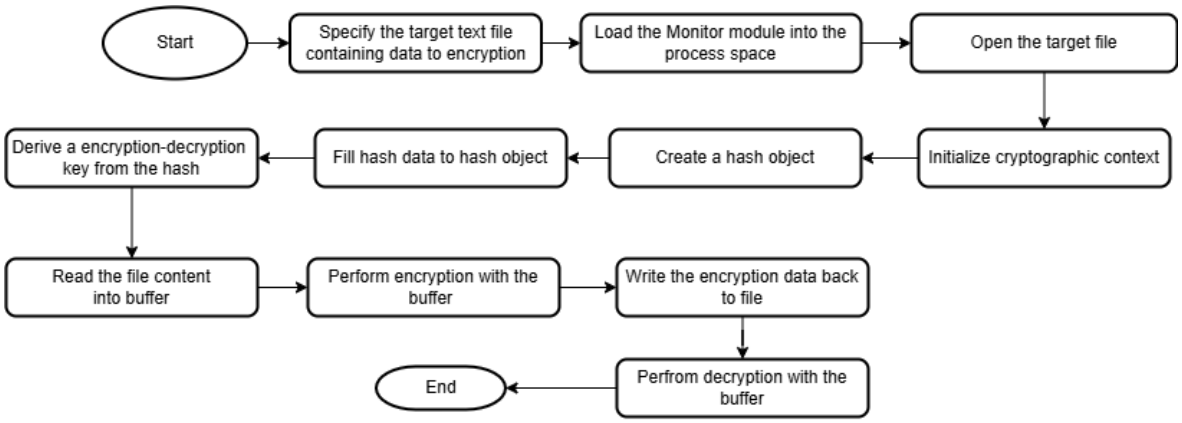

Fig. 6. Testing Program Execution Flow

4.2 Result

So as I mention in the functionality of each hook to an API in the Implement section. After each API is monitored it will write that information into a file name `api_logs.js`. The content of the file is as follows. Observing the result, we can conclude that the monitor has successfully catch all its target APIs once they are called.


{"apiName": "CreateFileA", "pre_call_parameters": {"lpFileName":
"0x00007FF6C71B3480=0x433a5c55736572735c48505c446f776e6c6f6164735c50726f6a6563745f335c7465737466696c652e747874", "dwDesiredAccess": c0000000, "dwShareMode": 0,
"lpSecurityAttributes": 000000000000000, "dwCreationDisposition": 3, "dwFlagsAndAttributes": 80, "hTemplateFile": 000000000000000}, "post_call_parameters":
{"lpFileName": "0x00007FF6C71B3480=0x433a5c55736572735c48505c446f776e6c6f6164735c50726f6a6563745f335c7465737466696c652e747874", "dwDesiredAccess": c0000000,
"dwShareMode": 0, "lpSecurityAttributes": 000000000000000, "dwCreationDisposition": 3, "dwFlagsAndAttributes": 80, "hTemplateFile": 000000000000000}, "result": 240}
{"apiName": "CryptAcquireContextA", "pre_call_parameters": {"phProv": "0x000000F617B8FC58=0x7fff210c1aaf", "szContainer": 0, "szProvider": 0, "dwProvType": 1,
"dwFlags": 0}, "post_call_parameters": {"phProv": "0x000000F617B8FC58=0x2a3d971d0b0", "szContainer": 0, "szProvider": 0, "dwProvType": 1, "dwFlags": 0}, "result": 1}
{"apiName": "CryptCreateHash", "pre_call_parameters": {"hProv": "0x2a3d971d0b0", "Algid": 8003, "hKey": 0, "dwFlags": 0, "phHash": "0x000000F617B8FC60=0x2a3d971c9b0"},
"post_call_parameters": {"hProv": "0x2a3d971d0b0", "Algid": 8003, "hKey": 0, "dwFlags": 0, "phHash": "0x000000F617B8FC60=0x2a3d971cde0"}, "result": 1}
{"apiName": "CryptHashData", "pre_call_parameters": {"hHash": "0x2a3d971cde0", "pbData": "0x00007FF6C71B3378=0x44756d6d7920646174612066f722068617368696e672e",
"dwDataLen": 17, "dwFlags": 0}, "post_call_parameters": {"hHash": "0x2a3d971cde0", "pbData": "0x00007FF6C71B3378=0x44756d6d7920646174612066f722068617368696e672e",
"dwDataLen": 17, "dwFlags": 0}, "result": 1}
{"apiName": "CryptDeriveKey", "pre_call_parameters": {"hProv": "0x2a3d971d0b0", "Algid": 6602, "hBaseData": "0x2a3d971cde0", "dwFlags": 1, "phKey":
"0x000000F617B8FC68=0x0"}, "post_call_parameters": {"hProv": "0x2a3d971d0b0", "Algid": 6602, "hBaseData": "0x2a3d971cde0", "dwFlags": 1, "phKey":
"0x000000F617B8FC68=0x2a3d971ba50"}, "result": 1}
{"apiName": "CryptEncrypt", "pre_call_parameters": {"hKey": "0x2a3d971ba50", "hHash": 0, "Final": 1, "dwFlags": 0, "pbData":
"0x000002A3D97235A0=0xce2be2cac36ae15970f2e6102c9cd08442840f97af4a964aa974767535cad67f", "pdwDataLen": "0x000000F617B8FC50=20", "dwBufLen": 30}, "post_call_parameters":
{"hKey": "0x2a3d971ba50", "hHash": 0, "Final": 1, "dwFlags": 0, "pbData":
"0x000002A3D97235A0=0x6c32e88ce5904a2cfccdd6b4d952f52dbbaef924cb50b0f90a9aab05778fd2445813a63dbeff14cb", "pdwDataLen": "0x000000F617B8FC50=28", "dwBufLen": 30},
"result": 1}
{"apiName": "CryptDecrypt", "pre_call_parameters": {"hKey": "0x2a3d971ba50", "hHash": 0, "Final": 1, "dwFlags": 0, "pbData":
"0x000002A3D97235A0=0x6c32e88ce5904a2cfccdd6b4d952f52dbbaef924cb50b0f90a9aab05778fd2445813a63dbeff14cb", "pdwDataLen": "0x000000F617B8FC50=28", },
"post_call_parameters": {"hKey": "0x2a3d971ba50", "hHash": 0, "Final": 1, "dwFlags": 0, "pbData":
"0x000002A3D97235A0=0xce2be2cac36ae15970f2e6102c9cd08442840f97af4a964aa974767535cad67f", "pdwDataLen": "0x000000F617B8FC50=20", }, "result": 1}

Fig. 7. Testing Result After Monitoring Testing Program

5 Conclusion

In conclusion, the objective of developing a functional segment of the Monitor module for the sandbox has been successfully achieved. Specifically, I have implemented a Monitor module designed to run on the x64 Windows operating system, capable of tracking the use of key cryptographic and file-handling APIs commonly employed by ransomware. The targeted APIs include `CryptAcquireContextA`, `CryptCreateHash`, `CryptHashData`, `CryptDeriveKey`, `CryptEncrypt`, `CryptDecrypt`, and CreateFileA. While these APIs represent critical functions often used by ransomware for encryption and file manipulation, there are numerous other APIs leveraged by ransomware for various malicious activities that I have not yet incorporated into the hooking mechanism. Expanding support for such APIs will be a priority for future work.

This project has provided me with substantial technical insights and hands-on experience in several important areas. I have gained an in-depth understanding of the Windows operating system's program loading and execution mechanisms, which are essential for designing robust monitoring solutions. Furthermore, I have acquired practical knowledge of sandbox architectures, their components, and their respective roles within the system. The implementation of the Monitor module has significantly enhanced my expertise in API hooking techniques, allowing me to engage in real-world implementation and problem-solving scenarios. Additionally, this project has deepened my understanding of ransomware behavior by exposing me to key APIs that ransomware exploits, further expanding my knowledge base for analyzing malicious programs.

Despite these accomplishments, there is considerable scope for future work. First, I plan to conduct extensive research into additional APIs commonly utilized by ransomware to expand the functionality and versatility of the Monitor module. This will include APIs related to network activity monitoring and process manipulation to provide a more comprehensive analysis of ransomware behavior. Next, I aim to implement the remaining modules of the sandbox architecture, completing the end-to-end process of analyzing malicious programs. This will encompass everything from receiving and processing a malware sample to generating detailed behavioral reports, ensuring the sandbox can be effectively applied in real-world scenarios.

Overall, the knowledge and skills gained through this project form a solid foundation for future development and innovation in the field of cybersecurity.

Reference

1. https://captain-woof.medium.com/api-hooking-with-detours-on-windows-7940a9b93427
2. https://cuckoo.readthedocs.io/
3. https://github.com/microsoft/Detours
4. https://app.any.run/docs/
5. https://github.com/microsoft/Detours/wiki
6. https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptacquirecontexta
7. https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-crypthashdata
8. https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptcreatehash
9. https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptderivekey
10. https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptencrypt
11. https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptdecrypt
12. https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea