Malware Analysis Report - Group 2

Omar Barkhadle, Advaita Bhargava, Siddhant More,
Yash Nimbalkar, Gauraksh Vernekar

April 2025

1 Introduction

Hank recently fell victim to a ransomware attack that encrypted his personal files. What the ransomware utilised was, at that time, an unknown encryption scheme that rendered his data inaccessible. Fortunately for him, we obtained a sample of the malware used in the attack from dark web sources. To understand the malware's behaviour, we performed a forensic investigation, which led us to decrypting Hank's files.

First, we reverse-engineered the malware to understand its behaviour. Then, we identified the encryption algorithm and parameters used. These include the key and initialisation vector. After that, we confirmed the recovered decryption key and IV by analysing the malware's runtime behaviour and code. With all this, we then developed a Python script that was successfully able to decrypt Hank's encrypted backup folder. We've provided all this information through clear documentation of our findings and methodology to ensure reproducibility and understandability.

Below is a table that covers our entire forensic process and what each element involved:

Analysis Type	**Description**
Static Analysis	This involved disassembling the malware to examine its code, structure, and indicators of compromise (IoCs) without execution.
Dynamic Analysis	In this part we ran the malware in a controlled environment to observe its runtime behaviour and encryption process.
Key Recovery	To recover the key, we derived it from the malware's execution and the code analysis.
Decryption Tool Development	Then, we used the recovered key to design and implement a Python tool that decrypted the affected files.

2 Reverse Engineering Analysis

2.1 Tools and Methodology

Tools used:

- Windows 10 VM → Used as our runtime environment
- Kali Linux VM → Used for binary disassembly and decryption tool development
- Ghidra, IDA → Used for static analysis of the binary
- PEStudio → Used for gathering static info about the binary
- x64dbg → Used for debugging the binary at runtime

We followed a hybrid malware analysis approach, combining both static analysis and dynamic analysis to reverse engineer the ransomware sample effectively.

2.2 Static Analysis

Static File Info

The initial analysis began with inspecting the binary sample named **mYSCpPoHAih**. The following hash values were computed for integrity verification and reference:
MD5: 1046f940c0cb8566fc6ca5445a7949f3
SHA-1: 227c0a54d2cc0f984eed50a3738267077a626432
SHA-256: 58d67e89e24ed64238f44b1295b03a4e54dfdabcbc53534d573ec496cac4434d

Using PEStudio, we examined the file headers and confirmed that the sample is a Portable Executable (PE) format as indicated by the MZ signature (0x4D 0x5A) at the beginning of the file. Further inspection revealed that it is a 64-bit executable. The binary also imports two libraries and those are SHLWAPI.dll and KERNEL32.dll. Refer to Figure 1 and 2 for more details.

Function-Level Static Analysis

Upon disassembling the malware, the following core functionalities of the sample were identified:

- **WinMain_0:** This is the main entry point of the binary. It first retrieves the current working directory in which the binary is present by calling the `GetCurrentDirectoryW` function and then calls the function `ProcessAndEncryptFiles` and this function is responsible for the file handling and the encryption logic. After the encryption process, the malware sleeps for 10 seconds and finally executes a self-deletion routine via `PingAndDelete`.
- **ProcessAndEncryptFiles:** This is the core payload handler. This uses a double loop method to process the files.

  In the first loop:

  - It appends a wildcard to the input path and goes through all the files using `FindFirstFileW` and `FindNextFileW`.
  - Then it skips directories, files starting with `~en`, and the binary itself (the self-exclusion check ensures the malware doesn't encrypt itself). And for each eligible file it:
    * Constructs the full path to the file.
    * Builds a new path using a `~en` prefix.
    * Calls the function `encryptFiles()` to encrypt the content and write it to the new file.
    * Deletes the original file using `DeleteFileW`.

  In the second loop:

  - It searches for all files in the directory starting with the `~en` prefix.
  - For each such file:
    * It then removes the `~en` prefix by constructing a new filename with the first 3 characters stripped.
    * Then renames the file using `MoveFileW`, effectively replacing the original file with the encrypted version under its original name.

- **encryptFiles:** This function handles the actual file encryption routine. It operates as follows:
  - Opens the input file for reading and then creates an output file for writing the encrypted content.
  - Allocates a 1008-byte memory buffer and reads the original file in chunks.
  - For each chunk:
    * Writes a fixed 16-byte IV (stored in memory at `unk_140086010`) to the output file.
    * Writes the length of the chunk as a 4-byte integer.
    * Performs AES-CBC encryption using a predefined key and IV (stored in memory at `unk_140086000` and `unk_140086010` respectively) via internal calls to `sub_1400025D1` and `sub_140001613`.
    * Writes the ciphertext chunk to the destination file.
  - This continues until the entire file is processed, after which all handles and memory are freed.

- **PingAndDelete:** This function executes the self-deletion mechanism. It constructs the following command line string using `cmd.exe`:

```
cmd.exe /C ping 127.1.247.73 –n 1 –w 1123 > Nul
\& Del "<path\_to\_sample>"
```

  This utilizes the ping command as a delay mechanism followed by the deletion of the binary itself. The command is executed in a hidden window using `CreateProcessW` with the `CREATE_NO_WINDOW` flag.

Static Analysis Outcome

The malware encrypts user files located in the current directory, chunking them into 1008-byte segments, each encrypted using AES in CBC mode. The IV and length are prepended before each encrypted chunk. After encryption, the original files are deleted, and the encrypted versions are renamed to match the original filenames. The binary then self-deletes to avoid detection and hinder analysis.

Upon examining the memory locations `unk_140086000` and `unk_140086010`, we were able to recover the hardcoded encryption key and initialization vector (IV) used by the malware. These values are:

- **Key:** `8d02e65e508308dd743f0dd4d31e484d` (see Fig. 3)
- **IV:** `0a0b0c0d0e0fa0b0c0d0e0f0aabbccdd` (see Fig. 4)

To confirm that this IV is indeed used during encryption, we examined the first 16 bytes of an encrypted file using the command:

```
xxd <encrypted_file> | head
```

The output clearly showed that the first 16 bytes matched the IV obtained from `unk_140086010`, thereby validating our finding (see Fig. 5)

To verify that the recovered key is correct and is actually used for encryption, we used dynamic analysis technique, which is explained in the next section.

2.3 Dynamic Analysis

For the dynamic analysis, the malware sample was executed in a controlled and isolated environment using a Windows virtual machine. This ensured the safety of the host system and allowed for precise behavioral monitoring of the malware.

Key Observations

We set breakpoints at points responsible for loading the hardcoded key and IV values during execution.

The following observations were made:

1. The addresses of the variables `unk_14006000` (the hardcoded key) and `unk_14006010` (the initialization vector) are loaded into `rdx` and `r8` respectively, immediately before the encryption process begins.
2. The IV stored at `unk_14006010` is consistently passed alongside the plaintext buffer to the encryption function.
3. Memory inspection during runtime confirmed that the values stored at these addresses matched those identified through static analysis (see Fig. 6).
    - **Key:** `8d02e65e508308dd743f0dd4d31e484d`
    - **IV:** `0a0b0c0d0e0fa0b0c0d0e0f0aabbccdd`

This confirms that the ransomware does not generate dynamic encryption parameters and instead relies on a hardcoded cryptographic key, making it easier to decrypt the affected files.

3 Decryption Tool Development

Based on insights from the malware's encryption behavior, we developed a custom decryption tool to recover the affected files. The malware used AES in CBC mode with a hardcoded 128-bit key and IV to encrypt the files in the directory.

Initially, our attempts to decrypt the files using the recovered key resulted in padding errors for every file and the files not getting decrypted completely. After inspecting the encryption process again, it became clear that each encrypted chunk was preceded by:

- A 16-byte IV,
- A 4-byte value representing the original (unpadded) chunk length,
- The ciphertext, padded to the nearest 16-byte boundary.

We then adapted our decryption script to this structure by dynamically computing the padded chunk size and truncating the decrypted output accordingly. Our script then processes all encrypted files in a directory, retrieves the IV from the first 16 bytes of the file, decrypts each file using the provided key, and restores the original files.

To run the Decryption tool in your local Windows environment:

- Make sure to install the required dependencies using the following command:

  ```
  pip install pycryptodome
  ```

- Place the `decryptScript.py` script in the same directory where all the encrypted files are present.
- Then run the following command to decrypt all the files in the directory:

  ```
  python decryptScript.py
  ```

Note: The complete decryption script is provided in Appendix 5.

4 Conclusion

Through a combination of static and dynamic analysis techniques, we were able to successfully reverse engineer the provided sample of malware. Our investigation revealed that the malware employed a hardcoded AES-CBC encryption with fixed key and IV values stored in the binary. This provided us with a practical opportunity to develop a working decryption tool to recover the original files and thus mitigate the issue.

5 Appendix

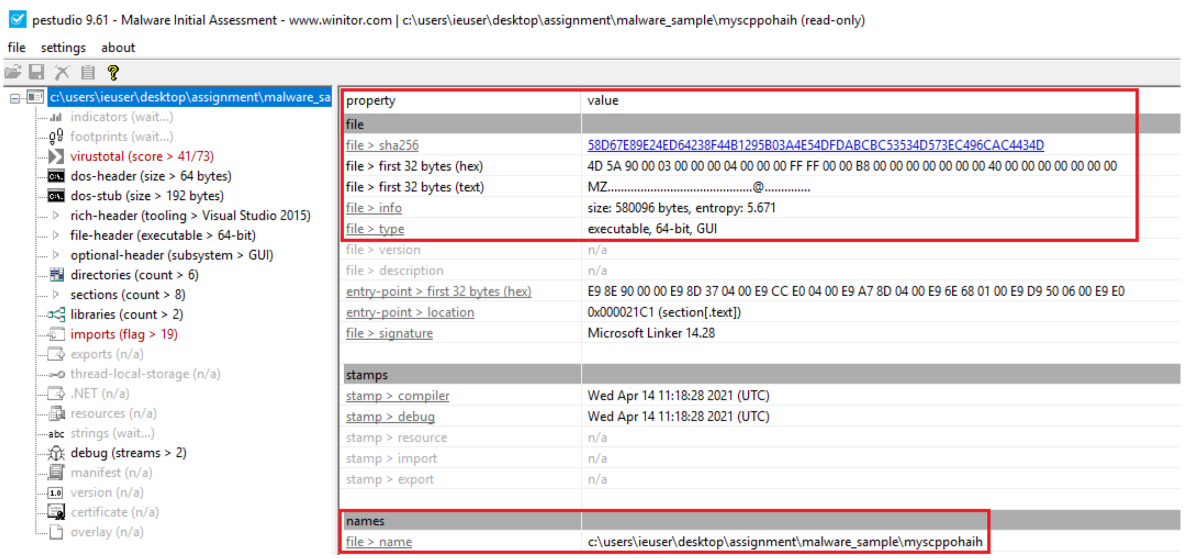

Figure 1: PEStudio: Static File Info

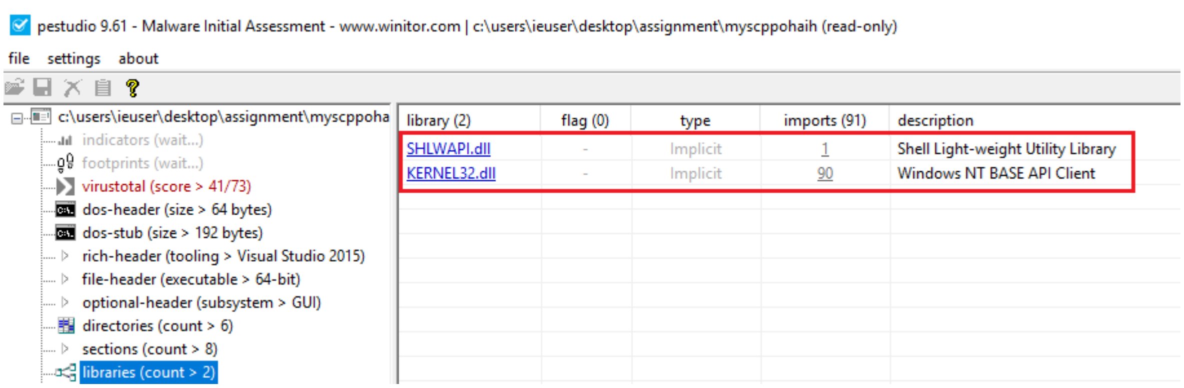

Figure 2: PEStudio: Library Imports

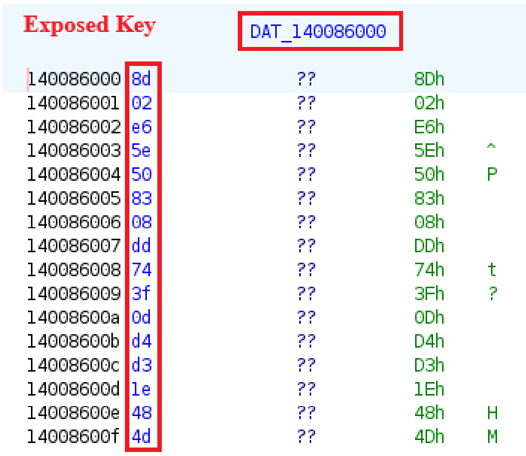

Figure 3: Ghidra: Exposed Key

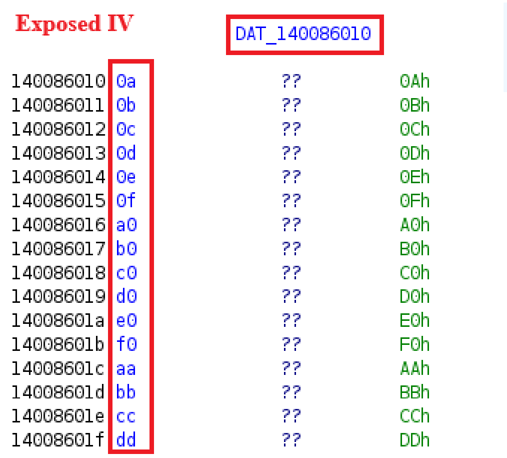

Figure 4: Ghidra: Exposed IV

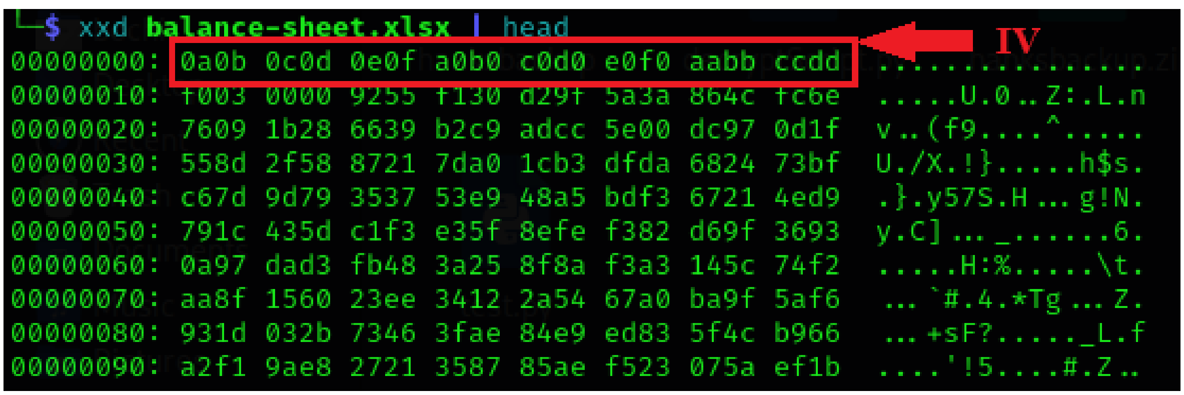

Figure 5: Static Analysis: xxd Output

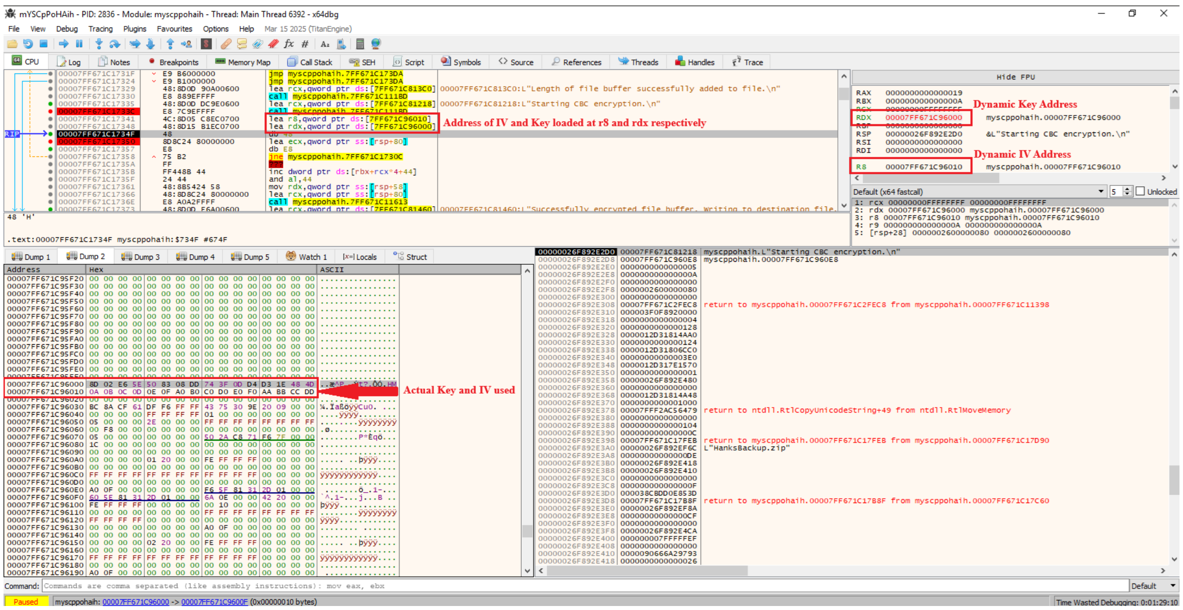

Figure 6: Dynamic Analysis: Actual Key & IV confirmed

decryptScript.py

```python
import os
import struct
from Crypto.Cipher import AES

key = bytes.fromhex("8D02E65E508308DD743F0DD4D31E484D")

def decrypt_chunk(cipher, chunk_len, ciphertext):
    decrypted = cipher.decrypt(ciphertext)
    return decrypted[:chunk_len] # Remove padding beyond original chunk length

def decrypt_file(input_file, key):
    with open(input_file, 'rb') as f_in:
        output_file = f"{input_file}.decrypted"
        with open(output_file, 'wb') as f_out:
            while True:
                iv = f_in.read(16) # Get the IV directly from the file
                if len(iv) < 16:
                    break

                length_bytes = f_in.read(4) # Read original chunk length (
                    before padding)
                if len(length_bytes) < 4:
                    break

                chunk_len = struct.unpack("<I", length_bytes)[0]

                # Calculate the padded size of the encrypted chunk
                padded_len = ((chunk_len + 15) // 16) * 16
```

```python
                ciphertext = f_in.read(padded_len)
                if len(ciphertext) < padded_len:
                    break

                cipher = AES.new(key, AES.MODE_CBC, iv)
                decrypted_data = decrypt_chunk(cipher, chunk_len, ciphertext)

                f_out.write(decrypted_data)

        print(f"Decryption completed for {input_file}.")
    return output_file

def process_files_in_directory(directory, key):
    # Loop through all the files in the directory
    for filename in os.listdir(directory):
        file_path = os.path.join(directory, filename)

        # Ignore any folders and the decryption script
        if os.path.isdir(file_path) or filename == "decryptScript.py":
            continue

        # Decrypt only the files present
        if os.path.isfile(file_path):
            decrypted_file_path = decrypt_file(file_path, key)
            os.remove(file_path) # Delete original encrypted file
            os.rename(decrypted_file_path, file_path) # Rename decrypted file to original filename

directory_path = os.getcwd() # Getting the current working directory
process_files_in_directory(directory_path, key)
```