

Malicious DLL Injection Detection using Ghidra: A Static Code Analysis Process

A Comprehensive Study on Malware Detection

Table of Contents

Table of Contents	2
Abstract	4
Introduction	4
Background on DLLs and DLL Injection	4
Why do we need to detect malicious DLLs	4
Why use Ghidra & Static Code Analysis	5
Literature review	6
DLL Injection Techniques	6
Existing Detection Methods	7
Role of Static Analysis in Malware Detection	8
Tools Like Ghidra in Cybersecurity	8
Limitations in Existing Literature	9
Methodology	10
Setting up the analysis environment	10
Static code analysis process	10
Criteria for Identifying Malicious Behavior	11
Detailed Analysis & Case Study	12
Identifying suspicious functions & API calls	12
Table 1: List of Suspicious APIs and their Functions	14
Compare the malicious with non-malicious to identify patterns	15
Case Study 1: Remote Thread Injection - Lab12-01.exe	15
Image 1: Analyzing the binary manually using Ghidra's Decompiler	15
Image 2: Output from analyzing the binary and searching for suspicious API call patterns	16
Image 3: Confirming suspicious API call pattern is found	16
Image 4: Added functionality of bookmarking functions which contain suspicious API call pattern	17
Case Study 2: Thread Manipulation Injection - Lab12-02.exe	17
Image 5: Output from analyzing the binary and searching for Suspicious API patterns in the file	18
Image 6: Confirming suspicious API call pattern is found	18
Image 7: Bookmarks the function which contains suspicious API call pattern	19
Case Study 3: Windows Hook Manipulation Injection - Lab12-03.exe	19
Image 8: Analysis of the binary to search for suspicious patterns and confirmation of suspicious API call patterns in the binary	20
Image 9: Bookmarks the function which contains suspicious API call pattern	20
Case Study 4: Injection by Privilege Escalation - Lab12-04.exe	21
Image 10: Analysis of the binary to search for suspicious patterns and	

confirmation of suspicious API call patterns in the binary	21
Image 11: Bookmarks the function which contains suspicious API call pattern	22
Case Study 5: Reflective DLL Injection - inject.exe	22
Image 12: Analysis of the binary to search for suspicious patterns	22
Image 13: List of the binary's system API calls	23
Case Study 6: Benign Binaries and the Importance of Patterns in DLL Injection	23
Image 14: Analysis of the benign binary to search for suspicious patterns	24
Image 15: No Suspicious Sequence Found, an indication of benign binary, and no false positives are detected	24
Image 16: No Suspicious Sequence Found, an indication of benign binary, and no false positives are detected	25
Results & Accuracy Check	26
Accuracy Metrics and Their Relevance	26
Extracting Necessary Information from Analyzing Binaries	26
Using the Accuracy Metrics to Compare Malicious and Benign Binaries	26
Final Accuracy Results	27
Conclusion	28
Appendix: Glossary of Technical Terms	29
Appendix: Scripts Used	30

Abstract

The report, *Malicious DLL Injection Detection using Ghidra: A Static Code Analysis Process*, explores using the Ghidra reverse engineering tool for detecting DLL injection techniques. DLL injection, often exploited for malicious purposes, involves executing unauthorized code within legitimate processes. The study identifies key DLL injection patterns by analyzing API call sequences and develops a Ghidra-based script to detect these patterns in binaries. Through six case studies, the project demonstrates its script's ability to identify malicious activities, achieving a detection accuracy of 97% with an F1-Score of 0.89. The report highlights the strengths and limitations of static code analysis, recommending the integration of dynamic techniques and obfuscation detection for more robust malware detection.

Introduction

Background on DLLs and DLL Injection

According to Microsoft's official documentation, a Dynamic-Link Library (DLL) is a module that contains functions and data that can be used by multiple programs simultaneously. This modular approach facilitates code reuse and simplifies updates to standardized applications. For instance, modifying the source code of a complex application like Microsoft Word can be intricate and time-consuming. In contrast, injecting a DLL allows for more straightforward modifications, as updates can be made within the DLL without altering the entire application. This technique is also prevalent in video games, where DLLs are injected to modify gameplay based on player preferences or to introduce new features.

DLLs contribute to the modularization of programs, which is particularly beneficial in large software projects. Dividing a program into separate DLLs facilitates easier maintenance and updates, as modifying a specific DLL can propagate changes throughout the application. Additionally, this modular structure can enhance load times and overall performance.

Why do we need to detect malicious DLLs

However, despite their advantages, DLLs can be exploited for malicious purposes. Cybercriminals utilize DLL injection techniques to execute unauthorized code within legitimate processes, compromising system security. A notable example is the 2010 Stuxnet attack, which targeted Iran's nuclear facilities by manipulating centrifuge operations through malicious DLL injections. More recently, the SolarWinds attack infiltrated U.S. federal institutions using similar methods. These incidents underscore the critical need for effective detection mechanisms to identify and mitigate malicious DLL injections.

Detecting malicious DLL injections poses significant challenges due to the sophisticated techniques attackers use to conceal their activities. Reflective DLL loading, for instance, allows DLLs to be loaded directly into memory without leaving traces on disk, bypassing traditional file-based detection methods. Additionally, attackers often inject malicious DLLs into trusted

processes, enabling their code to run under the guise of legitimate software while inheriting the privileges of the host process. The wide range of injection methods, such as DLL hijacking, process hollowing, and APC injection, further complicates detection efforts, as each technique requires tailored identification strategies. Furthermore, advanced memory manipulation techniques, such as overwriting legitimate process memory, can obscure the presence of injected DLLs from forensic tools. Existing detection methods often fall short because static signature-based systems are ineffective against obfuscated or modified threats, while behavioral analysis tools struggle with malware that delays or conceals its actions. Real-time monitoring solutions face performance trade-offs, and memory forensics, though powerful, demands significant resources and expertise. To address these gaps, advanced solutions integrating memory analysis, anomaly detection, and tools like Ghidra are essential for identifying and mitigating DLL injections effectively and this report proposes a solution by developing a script that employs static code analysis using Ghidra to detect malicious DLL injections.

Why use Ghidra & Static Code Analysis

Ghidra is a powerful open-source reverse engineering tool developed by the National Security Agency (NSA), providing advanced capabilities for analyzing executable files. Its primary strength lies in its ability to decompile assembly-level code into a human-readable high-level language, such as C, enabling detailed inspection of program logic and structure. This functionality makes Ghidra particularly useful for malware analysis, as it allows analysts to reverse-engineer malicious software, uncovering hidden behaviors and vulnerabilities. Additionally, Ghidra's modular and extensible architecture supports scripting and automation, which can significantly enhance the efficiency of analyzing large or complex codebases. Its graphical user interface and suite of analytical tools provide a streamlined environment for exploring binaries, identifying anomalies, and understanding program flow, making it an indispensable tool for static analysis of potentially malicious DLLs.

Static analysis, in turn, is a critical method for examining software without executing it, allowing for a safe and thorough evaluation of its code. By using static analysis, security analysts can detect vulnerabilities, malicious payloads, or unauthorized modifications within a program before it is executed, reducing the risk of triggering harmful behavior during the investigation process. This technique is particularly advantageous in identifying malicious DLLs because it enables the detection of suspicious patterns, hardcoded malicious instructions, or known indicators of compromise embedded in the code. Unlike dynamic analysis, which requires running the software in a controlled environment, static analysis avoids the complexities and potential risks associated with monitoring runtime behavior. Together, Ghidra and static analysis provide a robust framework for dissecting and understanding the inner workings of DLLs, offering a proactive approach to detecting and mitigating threats.

Dynamic-link libraries (DLLs) are modular components containing functions and data that can be used by multiple programs simultaneously, facilitating code reuse and simplifying updates. While they enhance performance and modularization in large software projects, DLLs are also vulnerable to exploitation through injection techniques. Cybercriminals use DLL injection to

execute unauthorized code within legitimate processes, as seen in high-profile attacks like Stuxnet and SolarWinds. These attacks highlight the critical need for detection mechanisms, as successful DLL injection can compromise system integrity, leak sensitive data, and enable further unauthorized access. Detecting these injections is challenging due to techniques like reflective DLL loading, process hollowing, and DLL hijacking, which manipulate memory to evade detection. Existing methods relying on static signatures or runtime behavior analysis often fail against obfuscated threats, making advanced solutions necessary.

This report addresses these challenges by leveraging Ghidra, an open-source reverse engineering tool developed by the NSA, to perform static code analysis and detect malicious DLL injections. Ghidra's ability to decompile assembly code into human-readable formats enables detailed analysis of binaries without executing them, reducing risks and simplifying the identification of malicious patterns. Its extensible architecture and automation capabilities further streamline the analysis process. Static analysis, combined with Ghidra, offers a proactive approach to detecting vulnerabilities, identifying suspicious behaviors, and mitigating the risks posed by malicious DLL injections, contributing to enhanced cybersecurity measures in both industry and academia.

Literature review

DLL injection has been a focus of cybersecurity research due to its dual use in legitimate applications and malicious exploits. Attackers leverage DLL injection techniques to manipulate processes, execute unauthorized code, and evade detection, making it a critical area for study. This section explores common DLL injection techniques, existing detection methods, the role of static analysis in malware detection, the application of tools like Ghidra, and identified gaps in current research.

DLL Injection Techniques

DLL injection encompasses various techniques, each with unique mechanisms and levels of complexity.

1. **Remote Thread Injection**

This is one of the simplest and most commonly used methods. It involves injecting a DLL into a target process by creating a remote thread that is called `LoadLibrary`. APIs such as `OpenProcess`, `VirtualAllocEx`, and `CreateRemoteThread` are used to allocate memory and execute the injection. While this method is effective, it is easier to detect compared to more sophisticated techniques.

2. **Reflective DLL Injection**

Reflective DLL injection eliminates the use of traditional API calls like `LoadLibrary` by embedding a custom loader within the DLL itself. The loader parses the DLL headers and manually maps the DLL into memory. This method is highly stealthy, as it avoids leaving artifacts on disk and relies on manual memory mapping. As such, it is particularly challenging to detect with conventional tools.

3. **Process Hollowing**

Process hollowing involves creating a legitimate process in a suspended state, replacing its memory with malicious code, and resuming it to execute under the legitimate process's name. This technique uses APIs like `CreateProcess`, `ZwUnmapViewOfSection`, `VirtualAllocEx`, and `WriteProcessMemory`. Its ability to masquerade as a legitimate process makes it difficult to identify without advanced behavioral analysis.

4. **Thread Context Manipulation**

This method involves suspending a thread in a target process, modifying its execution context to point to malicious code, and resuming it. It requires precise manipulation of thread registers using APIs like `GetThreadContext` and `SetThreadContext`. The technique is effective for executing malicious payloads covertly, as the thread resumes under the context of the legitimate application.

5. **Process Doppelgänger**

Process doppelgänger leverages the transactional nature of NTFS to create a process from a transacted (and deleted) file. The malicious code is written to the transacted file and executed, bypassing many detection mechanisms by avoiding persistent artifacts. This sophisticated technique exploits advanced file system features, making it a significant challenge for security tools.

These techniques illustrate the breadth of DLL injection methods and their increasing complexity, highlighting the need for equally sophisticated detection strategies.

Existing Detection Methods

Current detection methods for DLL injection are categorized into static and dynamic analysis:

- **Static Analysis** Static analysis involves examining executable files and their code without running them. This method relies on signature-based detection to identify known threats, as well as heuristic analysis to detect suspicious patterns. It is safe and proactive, allowing analysts to uncover threats before execution. However, static analysis can struggle against obfuscated or encrypted code that conceals malicious intent.
- **Dynamic Analysis** Dynamic analysis monitors the behavior of processes during execution to identify anomalies, such as unauthorized API calls (`VirtualAllocEx`, `WriteProcessMemory`, `CreateRemoteThread`). This approach is effective at detecting runtime behavior that static analysis might miss, especially for threats that rely on delayed execution. However, dynamic analysis is resource-intensive and can be bypassed by malware designed to mimic benign behavior or delay execution until after the monitoring period.

However, despite their strengths, both methods have limitations. Static analysis may miss runtime-dependent threats, while dynamic analysis is less effective against advanced stealth techniques.

Role of Static Analysis in Malware Detection

Static code analysis plays an important role in malware detection by enabling safe and detailed evaluation of software without execution. For DLL injection scenarios, static analysis is particularly effective at identifying suspicious API calls, hardcoded instructions, and known indicators of compromise. Compared to dynamic analysis, static methods avoid the risks associated with executing potentially harmful code, offering a proactive defense mechanism. This approach is especially useful for dissecting obfuscated binaries and detecting threats embedded deep within the code structure.

Tools Like Ghidra in Cybersecurity

Ghidra, an open-source reverse engineering framework developed by the National Security Agency (NSA), has revolutionized static code analysis and malware detection. It offers a comprehensive suite of tools for analyzing executable files, making it a cornerstone in modern cybersecurity workflows. Ghidra's robust feature set has been instrumental in helping researchers and analysts uncover hidden vulnerabilities, dissect complex malware, and enhance their understanding of software behavior.

Key Features of Ghidra:

1. **Decompilation and Disassembly**

One of Ghidra's standout features is its ability to decompile low-level assembly code into high-level languages like C, significantly reducing the complexity of reverse engineering. Its integrated disassembler provides a detailed breakdown of machine code, enabling analysts to examine program logic and understand its functionality.

2. **Automation and Scripting**

Ghidra's extensible architecture supports custom scripts written in Java or Python. This capability allows analysts to automate repetitive tasks, such as identifying suspicious patterns or extracting function calls, thereby saving time and increasing efficiency in large-scale analyses.

3. **Graphical User Interface (GUI)**

The tool's intuitive GUI facilitates the visualization of code flow, dependencies, and control structures, making it easier to analyze intricate binaries. Graph-based representations of function calls and control flow enhance the user's ability to trace the execution path of software.

4. **Collaboration Features**

Ghidra supports multi-user collaboration, allowing teams to work on the same project simultaneously. This feature is particularly useful in large-scale investigations where multiple analysts contribute to dissecting and documenting malware.

5. **Cross-Platform Support**

Ghidra is compatible with various operating systems, including Windows, macOS, and Linux, ensuring accessibility for analysts regardless of their environment.

6. **Built-in Emulation and Debugging Tools**

While primarily focused on static analysis, Ghidra includes some debugging and

emulation capabilities. These features allow analysts to simulate execution environments, providing additional context for understanding software behavior.

Ghidra has been widely adopted in the cybersecurity industry and academia for various purposes:

- **Deobfuscating Malware:** Researchers have used Ghidra to analyze obfuscated malware, such as GuLoader and TrickBot, uncovering hidden functionalities and understanding the obfuscation techniques employed.
- **Analyzing Ransomware:** Ghidra has been applied to dissect ransomware like Ryuk and Conti, allowing analysts to identify encryption algorithms and potential weaknesses in the malware's implementation.
- **Reverse Engineering Advanced Persistent Threats (APTs):** Security professionals have leveraged Ghidra to investigate APTs, which often utilize sophisticated DLL injection techniques to achieve persistence and evade detection.
- **Uncovering Supply Chain Attacks:** Ghidra has been instrumental in analyzing attacks like SolarWinds, enabling a better understanding of how malicious DLLs were injected into legitimate processes and how the supply chain was compromised.

While tools like IDA Pro and Radare2 are also popular in the cybersecurity community, Ghidra's open-source nature and rich feature set provide a unique combination of accessibility and functionality. Unlike IDA Pro, which requires a paid license for advanced features, Ghidra offers comparable capabilities for free, making it a preferred choice for many organizations and researchers. Radare2, while lightweight and scriptable, lacks the user-friendly interface and advanced decompilation capabilities that Ghidra provides.

Despite its strengths, Ghidra is not without its limitations. Its decompiler may struggle with heavily obfuscated or packed binaries, requiring additional manual intervention. Moreover, while Ghidra supports scripting, it has a steeper learning curve compared to tools with simpler automation frameworks. However, its active community and extensive documentation mitigate these challenges, making it an increasingly versatile and powerful tool in the fight against malware.

The adoption of Ghidra has democratized access to advanced reverse engineering tools, enabling even small teams or independent researchers to perform in-depth analyses of malware. By lowering the barrier to entry, Ghidra has fostered innovation and collaboration within the cybersecurity community. Its integration into workflows for detecting and analyzing DLL injection techniques exemplifies its value in addressing real-world threats. As new techniques and attack vectors emerge, Ghidra's adaptability ensures it remains a critical asset in cybersecurity defense.

Limitations in Existing Literature

While existing methods and tools provide a foundation for detecting DLL injections, several gaps remain:

1. **Limited Coverage of Advanced Techniques:** Advanced techniques like Reflective DLL Injection and Process Doppelgänger rely heavily on memory manipulation and obfuscation, making them challenging to detect with traditional methods.
2. **Integration Challenges:** Many static analysis tools are not well integrated into modern reverse engineering workflows, limiting their usability for large-scale or complex analyses.
3. **Scalability and Automation:** Manual analysis remains time-consuming, particularly for large codebases or highly obfuscated binaries.

This project aims to address these gaps by leveraging Ghidra to develop a static analysis script specifically designed to detect malicious DLL injections. The proposed approach combines automation, pattern recognition, and memory analysis to provide a scalable and efficient solution. By focusing on both known and emerging injection techniques, this work seeks to enhance detection accuracy and contribute to the broader field of malware analysis.

Methodology

Setting up the analysis environment

The analysis environment was established on a system running Kali Linux, with Ghidra installed as the primary reverse engineering tool. Ghidra's decompiler and script manager were utilized to analyze binaries and develop scripts capable of detecting malicious patterns indicative of DLL injection techniques. This environment provided a robust platform for both static code analysis and script execution, ensuring compatibility with the required tools and datasets.

The dataset comprised a total of 39 binaries, including 5 malicious and 34 benign samples. The malicious binaries were primarily sourced from the lab files of the Practical Malware Analysis book, specifically Chapter 12, which focuses on DLL Injection techniques. These binaries included [lab12-01.exe](#), [lab12-02.exe](#), [lab12-03.exe](#), and [lab12-04.exe](#). Additionally, one malicious binary was obtained from the GitHub repository of Stephen Fewer's Reflective DLL Injection project, which is a widely referenced implementation of this advanced injection technique.

For benign binaries, a combination of files from the Windows32 folder and other non-malicious samples from the Practical Malware Analysis book's lab sections were used. Examples of benign binaries included [lab11-01.exe](#) and [lab10-01.exe](#), among others. This balanced dataset ensured that the scripts could be evaluated for accuracy in distinguishing between benign and malicious behavior, providing a foundation for reliable detection mechanisms.

Static code analysis process

The static code analysis process utilized Ghidra's reverse engineering tools to identify malicious DLL injection patterns through the analysis of API call sequences within binaries. The analysis began with the identification of suspicious external function calls in the binary's symbol table.

API functions such as `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` were specifically targeted, as they are frequently associated with DLL injection techniques. The addresses of these API calls were extracted and logged to serve as the foundation for further analysis.

To detect malicious behavior, the script processed the extracted API calls and compared them against predefined DLL injection patterns. Each pattern consisted of at least eight API calls, including sequences like `OpenProcess` → `VirtualAllocEx` → `WriteProcessMemory` → `CreateRemoteThread`. The deliberate inclusion of a minimum of eight APIs in each pattern was aimed at reducing false negatives by ensuring sufficient complexity and precision in detection. The script was designed to account for minor variations, such as gaps between calls, enabling it to identify obfuscated or slightly altered injection patterns.

After detecting suspicious sequences, the script generated detailed outputs, highlighting the identified patterns along with their associated API calls and memory addresses. This comprehensive reporting allowed for an efficient review of the detected behaviors and facilitated further analysis of potentially malicious binaries.

By focusing on robust pattern detection and leveraging Ghidra's static analysis capabilities, this approach ensured the accurate identification of DLL injection techniques. The inclusion of complex, multi-API patterns minimized false negatives, while the systematic analysis of binaries enhanced the reliability of the detection process.

Criteria for Identifying Malicious Behavior

The detection of malicious DLL injection behavior in this project is based on a systematic analysis of system API calls and their sequences, utilizing static code analysis in Ghidra. The criteria are designed to ensure the accurate identification of injection patterns while minimizing false positives and negatives. These criteria incorporate insights from known DLL injection techniques, including Remote Thread DLL Injection, Thread Manipulation, Windows Process Hooking, and Privilege Escalation.

The primary criterion involves analyzing the sequence of system API calls within the binary. The script identifies and extracts these calls, assigning an index to each based on its location in the execution flow. Known patterns associated with DLL injection are then matched against the extracted sequences. To reduce false negatives, each pattern consists of at least eight system API calls, ensuring sufficient complexity for detection. For example, a pattern for Remote Thread DLL Injection includes calls such as `GetCurrentDirectoryA`, `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. Similarly, other patterns are designed to match behaviors associated with thread manipulation, process hooking, and privilege escalation.

A key aspect of sequence matching is the handling of gaps between suspicious API calls. The script allows a maximum gap of 100 instructions, ensuring that minor variations in execution do

not prevent detection while avoiding overly long gaps that could introduce false positives. Furthermore, the script accounts for repeated calls to the same API within a short span, skipping redundant entries while maintaining the integrity of the sequence.

To enhance accuracy, the script also checks for signs of code obfuscation, such as irregular or complex instructions surrounding suspicious API calls. While this step is optional, it adds a layer of analysis for binaries employing advanced evasion techniques. Detected sequences are flagged, and their associated addresses are reported for easy navigation, enabling detailed manual investigation where necessary.

Global and local API hooks are also considered as part of the analysis. Global hooks target all processes, while local hooks are specific to individual processes. For instance, the use of `SetWindowsHookEx` is closely examined, as it requires a DLL to be injected to monitor or modify system events like keystrokes and mouse movements, making it a common target for malicious activity.

The effectiveness of these criteria has been evaluated by comparing the script's outputs against a labeled dataset of executables, including both benign and malicious samples. The evaluation includes metrics such as true positives, false positives, false negatives, and true negatives, with the performance summarized using metrics like accuracy, precision, and F1-score. While the current implementation has shown high accuracy, further refinement is planned, including the expansion of the dataset, integration of dynamic analysis techniques, and additional functionalities such as bookmarking suspicious functions within Ghidra for enhanced usability.

By focusing on well-defined patterns and incorporating flexibility in sequence matching, the criteria provide a robust framework for identifying malicious DLL injection behavior in static analysis environments. This approach balances precision and efficiency, ensuring that the script can effectively detect a wide range of injection techniques while remaining adaptable to evolving threats.

Detailed Analysis & Case Study

Identifying suspicious functions & API calls

Identifying suspicious API calls relied on insights from Chapter 12 of the *Practical Malware Analysis* book. The book highlights that malicious DLL injections often follow specific patterns of system API calls, which vary depending on the injection method employed. For this analysis, we focused on commonly exploited APIs that play a significant role in facilitating DLL injection techniques. These APIs are frequently observed in malware analysis and are widely documented as indicators of potential malicious activity.

System APIs like `VirtualAllocEx`, `WriteProcessMemory`, `CreateRemoteThread`, and `SetWindowsHookEx` were prioritized because they represent key steps in most injection scenarios. For example, `VirtualAllocEx` is used to allocate memory in a target process, while `WriteProcessMemory` writes malicious code into that memory, and `CreateRemoteThread` triggers execution. When executed in specific sequences, these functions serve as clear indicators of potential malicious behavior. Additionally, APIs such as `OpenProcess` and `SetThreadContext` are often exploited for manipulating processes and threads during injection.

To provide clarity and ease of navigation, the following table summarizes the suspicious APIs considered in this project, categorized by their primary function:

API Name	Function	Role
<code>CreateProcess</code> , <code>CreateProcessA</code>	Creates a new process.	Used in process hollowing to create a suspended process for code replacement.
<code>OpenProcess</code>	Opens a handle to another process.	Enables access to a target process's memory or threads.
<code>VirtualAllocEx</code> , <code>VirtualAlloc</code>	Allocates memory in the target process.	Creates space for malicious code or DLL path.
<code>WriteProcessMemory</code>	Writes data into allocated memory.	Injects the malicious code or DLL path.
<code>CreateRemoteThread</code> , <code>NtCreateThreadEx</code>	Creates a thread in the target process.	Executes the malicious payload or DLL in the target process.
<code>SetThreadContext</code> , <code>ResumeThread</code>	Modifies and resumes the context of a suspended thread	Used in Thread Context Manipulation to hijack execution.
<code>NtUnmapViewOfSection</code> , <code>NtMapViewOfSection</code>	Unmaps or maps memory sections in a process.	Overwrites legitimate code with malicious payloads during process hollowing.
<code>VirtualProtectEx</code>	Modifies memory protection settings.	Ensures the injected code is executable.
<code>FlushInstructionCache</code>	Clears the CPU instruction cache.	Allows newly written code to execute properly.
<code>LoadLibraryA</code> , <code>LoadLibraryW</code> ,	Loads a DLL into the process memory.	Commonly used in Remote DLL Injection.

API Name	Function	Role
LoadLibraryExA, LoadLibraryExW		
GetProcAddress	Resolves the address of a function within a loaded DLL.	Dynamically retrieves addresses of required functions.
SetWindowsHookExA, SetWindowsHookExW	Installs hooks for system events, such as keyboard or mouse inputs.	Exploited in Windows Hooking to inject DLLs into GUI applications.
OpenThread, SuspendThread, GetThreadContext	Manipulates threads within a process	Allows suspension and redirection of threads for injection purposes.
AdjustTokenPrivileges, LookupPrivilegeValueA	Escalates privileges of a process.	Ensures elevated access for injecting DLLs into protected processes.
CallNextHookEx, UnhookWindowsHookEx	Manages and removes hooks.	Maintains or terminates malicious hooks installed in the system.
AllocConsole, FindWindowA, ShowWindow, SendMessageA	Manages console and window operations.	Utilized for Windows Hooking or concealing malicious behavior.
NtCreateSection, NtCreateProcessEx	Creates sections or processes within the system.	Used in advanced techniques like Process Doppelg�nging.

Table 1: List of Suspicious APIs and their Functions

This list forms the baseline for identifying potential DLL injection techniques and serves as the first layer of filtering before delving into sequence analysis and pattern matching. By isolating and analyzing these suspicious API calls, the detection script ensures that known patterns for common injection techniques, such as Remote Thread Injection, Thread Context Manipulation, and Process Hollowing, can be flagged effectively. Including multiple APIs for each functionality also accounts for variations in injection methods, thereby reducing the chances of false negatives.

Compare the malicious with non-malicious to identify patterns

Case Study 1: Remote Thread Injection - [Lab12-01.exe](#)

The first binary analyzed, [Lab12-01.exe](#), demonstrated a clear instance of **Remote Thread Injection**, a widely used DLL injection technique. Using the Ghidra script, the entire sequence of suspicious API calls was extracted and matched against the predefined pattern for this method. The script successfully flagged a suspicious sequence, starting with [GetCurrentDirectoryA](#), which sets the context for locating the DLL to be injected.

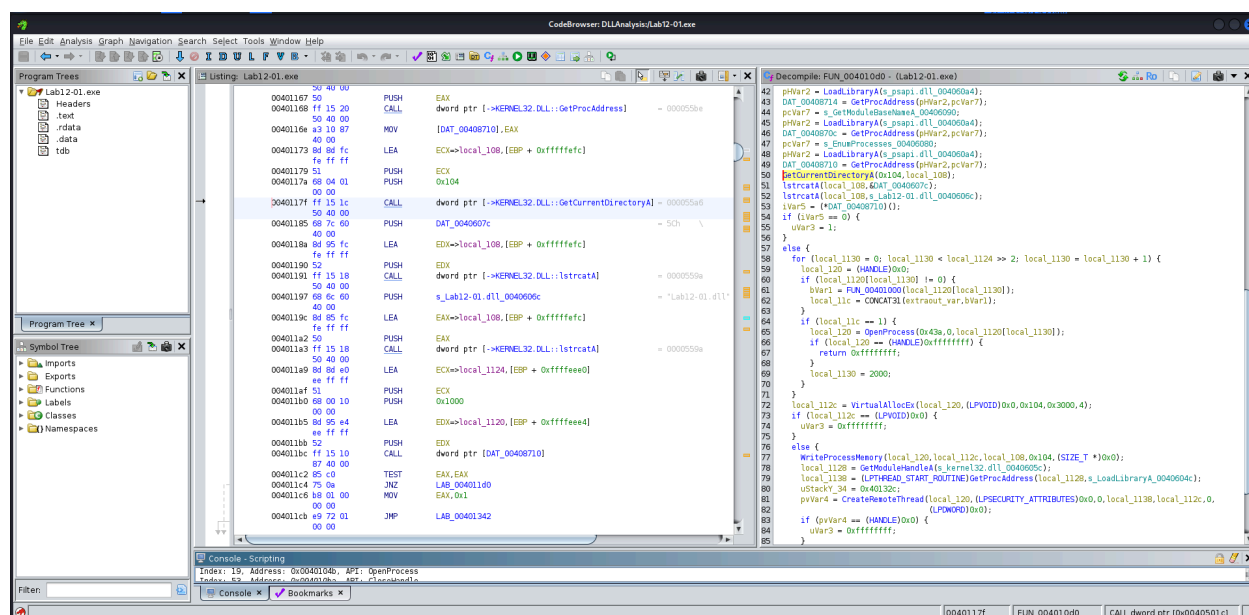


Image 1: Analyzing the binary manually using Ghidra's Decompiler

The subsequent calls formed a coherent chain indicative of Remote Thread Injection. After [lstrcatA](#) at address [0x00401191](#), which appears to handle string concatenation to construct the DLL path, the call to [OpenProcess](#) at [0x00401260](#) confirmed that the binary was attempting to gain access to a target process. Once the process handle was acquired, [VirtualAllocEx](#) at [0x004012a1](#) allocated memory in the remote process to hold the malicious payload.

Following the memory allocation, [WriteProcessMemory](#) at [0x004012da](#) was executed to write the DLL payload into the allocated memory. This was a critical step, as it allowed the malicious code to be transferred into the target process's address space. Once the payload was in place, [GetModuleHandleA](#) and [GetProcAddress](#) were identified at [0x004012e5](#) and [0x004012fd](#) respectively, confirming that the binary resolved the necessary module and function addresses—key prerequisites for creating a remote thread. Finally, [CreateRemoteThread](#) at [0x00401326](#) executed the injected DLL within the target process, completing the Remote Thread Injection process.

```
Console - Scripting
Searching for sequence: ['GetCurrentDirectoryA', 'lstrcatA', 'OpenProcess', 'VirtualAllocEx', 'WriteProcessMemory', 'GetModuleHandleA', 'GetProcAddress', 'CreateRemoteThread']
Starting potential sequence at index 8 (GetCurrentDirectoryA at 0x0040117f)
Found API call: lstrcatA at 0x00401191 (Index 100)
API matches sequence at position 1: lstrcatA
Found API call: lstrcatA at 0x004011a3 (Index 104)
Skipping duplicate API: lstrcatA
Found API call: OpenProcess at 0x00401260 (Index 143)
API matches sequence at position 2: OpenProcess
Found API call: VirtualAllocEx at 0x004012a1 (Index 157)
API matches sequence at position 3: VirtualAllocEx
Found API call: WriteProcessMemory at 0x004012da (Index 171)
API matches sequence at position 4: WriteProcessMemory
Found API call: GetModuleHandleA at 0x004012e5 (Index 173)
API matches sequence at position 5: GetModuleHandleA
Found API call: GetProcAddress at 0x004012fd (Index 178)
API matches sequence at position 6: GetProcAddress
Found API call: CreateRemoteThread at 0x00401326 (Index 190)
API matches sequence at position 7: CreateRemoteThread
Complete sequence found
```

Image 2: Output from analyzing the binary and searching for suspicious API call patterns

```
Suspicious sequences detected:
Sequence:
GetCurrentDirectoryA at address 0x0040117f
lstrcatA at address 0x00401191
OpenProcess at address 0x00401260
VirtualAllocEx at address 0x004012a1
Potential variable assignment: EDX <- EBP
Potential variable assignment: EBP <- 0x7d0
Potential variable assignment: EBP <- EAX
Potential variable assignment: ECX <- EBP
Potential variable assignment: EAX <- EBP
WriteProcessMemory at address 0x004012da
GetModuleHandleA at address 0x004012e5
GetProcAddress at address 0x004012fd
CreateRemoteThread at address 0x00401326
---
```

Image 3: Confirming suspicious API call pattern is found

The script's console output displayed the complete sequence of API calls alongside their corresponding addresses, ensuring transparency in the analysis. Each call in the pattern matched the expected order for Remote Thread Injection, verifying the malicious nature of the binary. Ghidra's decompiler view further reinforced these findings by revealing the function `FUN_004010d0`, where the entire sequence of API calls appeared in logical succession.

Type	Category	Description	Location	Label	Code Unit
Analysis	Found Code	Found code from operand reference	00401f04	LAB_00401f04	MOV ECK,dword ptr [ESP + ...]
Analysis	Found Code	Found code from operand reference	00401fdc	LAB_00401fdc	PUSH EBP
Analysis	Function ID Analyzer	Library Function - Single Match, __exit	004014e5	__exit	PUSH 0x0
Analysis	Function ID Analyzer	Library Function - Single Match, __global_unwind2	00401ee4	__global_unwind2	PUSH EBP
Analysis	Function ID Analyzer	Library Function - Single Match, __local_unwind2	00401f26	__local_unwind2	PUSH EBX
Analysis	Function ID Analyzer	Library Function - Single Match, _malloc	004022a0	_malloc	PUSH dword ptr [DAT_0040...
Analysis	Function ID Analyzer	Library Function - Single Match, _rh_malloc	00402362	_rh_malloc	OMP dword ptr [ESP + _SI...
Analysis	Function ID Analyzer	Library Function - Single Match, _strlen	00402460	_strlen	MOV ECK,dword ptr [ESP + ...]
Analysis	Function ID Analyzer	Library Function - Single Match, _strchr	00402a30	_strchr	XOR EAX,EAX
Analysis	Function ID Analyzer	Library Function - Single Match, _strstr	00402ef0	_strstr	MOV ECK,dword ptr [ESP + ...]
Analysis	Function ID Analyzer	Library Function - Single Match, _strncmp	00402f70	_strncmp	PUSH EBP
Analysis	Function ID Analyzer	Library Function - Single Match, _strncpy	00403eb0	_strncpy	MOV ECK,dword ptr [ESP + ...]
Analysis	Function ID Analyzer	Library Function - Single Match, _memset	004047f0	_memset	MOV EDI,dword ptr [ESP + ...]
Analysis	Address Table	Address table[3] created	00402924	switchdata0_00402924	addr switch0_00402904:c...
Analysis	Address Table	Address table[4] created	00402a08	switchdata0_00402a08	addr switch0_004028f5:c...
Analysis	Address Table	Address table[3] created	00402aac	switchdata0_00402aac	addr switch0_00402a9d:c...
Analysis	Address Table	Address table[4] created	00402ba0	switchdata0_00402ba0	addr switch0_00402a77:c...
Analysis	Address Table	Address table[3] created	00404514	switchdata0_00404514	addr switch0_004044fd:c...
Analysis	Address Table	Address table[4] created	004045f8	switchdata0_004045f8	addr switch0_004044e5:c...
Analysis	Address Table	Address table[3] created	0040469c	switchdata0_0040469c	addr switch0_0040468d:c...
Analysis	Address Table	Address table[4] created	00404790	switchdata0_00404790	addr switch0_00404667:c...
Analysis	Address Table	Address table[2] created	0040542c		addr LAB_004040d9
Analysis	Address Table	Address table[2] created	00405438		addr LAB_0040418d
Analysis	Address Table	Address table[2] created	00405444		addr LAB_00404311
Analysis	Address Table	Address table[2] created	00408340	PTR_DAT_00408340	addr DAT_0040834a
Analysis	Suspicious Sequence	Function FUN_004010d0 contains suspicious A...	004010d0	FUN_004010d0	PUSH EBP
Info	PE Header	IMAGE_DIRECTORY_ENTRY_IMPORT	0040544c	DWORD_0040544c	dword_5474h
Info	PE Header	IMAGE_DIRECTORY_ENTRY_IAT	00405000	PTR_CloseHandle_00405000	addr KERNEL32.DLL:Close...

Image 4: Added functionality of bookmarking functions which contain suspicious API call pattern

The detection script's bookmarks proved particularly useful, allowing us to navigate directly to the addresses of each API call. This feature enabled detailed inspection of [GetCurrentDirectoryA](#), [lstrcatA](#), [OpenProcess](#), [VirtualAllocEx](#), [WriteProcessMemory](#), [GetModuleHandleA](#), [GetProcAddress](#), and [CreateRemoteThread](#), all of which collectively confirmed the injection attempt.

This pattern, consisting of at least 8 API calls, adheres to the established baseline for identifying Remote Thread Injection. By including these specific APIs in the detection sequence, false negatives are minimized, as shorter or incomplete sequences could potentially be benign.

The analysis of [Lab12-01.exe](#) successfully highlighted a classic Remote Thread Injection mechanism. The detection script accurately identified and flagged the full sequence of suspicious API calls, demonstrating its effectiveness in analyzing and isolating malicious behavior within binaries.

Case Study 2: Thread Manipulation Injection - [Lab12-02.exe](#)

In the second case study, the analysis focuses on detecting the *Thread Manipulation* DLL injection technique. This method manipulates the execution context of a thread to execute malicious code, a behavior that involves APIs such as [CreateProcessA](#), [VirtualAlloc](#), [GetThreadContext](#), [WriteProcessMemory](#), [SetThreadContext](#), and [ResumeThread](#). These APIs work together to allocate memory, inject the DLL, and redirect the thread's execution flow to the malicious payload.

```
Console - Scripting

Searching for sequence: ['GetCurrentDirectoryA', 'Istrcata', 'OpenProcess', 'VirtualAllocEx', 'WriteProcessMemory', 'GetModuleHandleA', 'GetProcAddress', 'CreateRemoteThread']
Searching for sequence: ['CloseHandle', 'CreateProcessA', 'VirtualAlloc', 'GetThreadContext', 'GetModuleHandleA', 'VirtualAllocEx', 'WriteProcessMemory', 'SetThreadContext', 'ResumeThread']

Starting potential sequence at index 1 (CloseHandle at 0x004010dd)
Found API call: CreateProcessA at 0x0040115f (Index 123)
API matches sequence at position 1: CreateProcessA
Found API call: VirtualAlloc at 0x0040117b (Index 130)
API matches sequence at position 2: VirtualAlloc
Found API call: GetThreadContext at 0x00401195 (Index 138)
API matches sequence at position 3: GetThreadContext
Found API call: GetModuleHandleA at 0x004011e1 (Index 157)
API matches sequence at position 4: GetModuleHandleA
Found API call: GetProcAddress at 0x004011e8 (Index 159)
Found API call: VirtualAllocEx at 0x00401222 (Index 180)
API matches sequence at position 5: VirtualAllocEx
Found API call: WriteProcessMemory at 0x00401251 (Index 195)
API matches sequence at position 6: WriteProcessMemory
Found API call: WriteProcessMemory at 0x004012b1 (Index 227)
Skipping duplicate API: WriteProcessMemory
Found API call: WriteProcessMemory at 0x004012d5 (Index 240)
Skipping duplicate API: WriteProcessMemory
Found API call: SetThreadContext at 0x004012f5 (Index 250)
API matches sequence at position 7: SetThreadContext
Found API call: ResumeThread at 0x004012ff (Index 253)
API matches sequence at position 8: ResumeThread
Complete sequence found
Potential variable assignment: EAX <- EBP
Potential variable assignment: EDX <- ECX
Potential variable assignment: ECX <- EBP
Potential variable assignment: EAX <- EDX
Potential variable assignment: EDX <- EBP
Potential variable assignment: ECX <- EBP
Potential variable assignment: EAX <- EBP
Potential variable assignment: EBP <- EAX
Potential variable assignment: EBP <- EAX

Searching for sequence: ['AllocConsole', 'FindWindowA', 'ShowWindow', 'GetModuleHandleA', 'SetWindowsHookExA', 'GetMessageA', 'UnhookWindowsHookEx']
Searching for sequence: ['GetCurrentProcess', 'OpenProcessToken', 'LookupPrivilegeValueA', 'CloseHandle', 'AdjustTokenPrivileges', 'LoadLibraryA', 'GetProcAddress', 'OpenProcess', 'CreateRemoteThread']

Starting potential sequence at index 15 (GetCurrentProcess at 0x00401c62)
Next API in sequence not found within max_gap
Incomplete sequence
```

Image 5: Output from analyzing the binary and searching for Suspicious API patterns in the file

The analysis begins with identifying a suspicious sequence of API calls within the binary. The sequence starts with **CreateProcessA**, which creates a suspended process, followed

```
Suspicious sequences detected:
Sequence:
  CloseHandle at address 0x004010dd
  CreateProcessA at address 0x0040115f
  VirtualAlloc at address 0x0040117b
  GetThreadContext at address 0x00401195
  GetModuleHandleA at address 0x004011e1
  VirtualAllocEx at address 0x00401222
  Potential variable assignment: EAX <- EBP
  Potential variable assignment: EDX <- ECX
  Potential variable assignment: ECX <- EBP
  Potential variable assignment: EAX <- EDX
  Potential variable assignment: EDX <- EBP
  Potential variable assignment: ECX <- EBP
  Potential variable assignment: EAX <- EBP
  Potential variable assignment: EBP <- EAX
  Potential variable assignment: EBP <- EAX
  SetThreadContext at address 0x004012f5
  ResumeThread at address 0x004012ff
  ...

Analysis Complete.
extractdll.py> Finished!
```

Image 6: Confirming suspicious API call pattern is found

by **VirtualAlloc** for memory allocation. **GetThreadContext** is then used to retrieve the thread's context, and **WriteProcessMemory** writes the malicious payload into the target process's allocated memory. Subsequently, **SetThreadContext** modifies the instruction pointer (IP) or register to point to the injected DLL, and **ResumeThread** resumes the execution of the thread, effectively loading the DLL.

Bookmarks - (26 bookmarks)						
Type	Category	Description	Location	Label	Code Unit	
Analysis	Found Code	Found code from operand reference	004024c8	LAB_004024c8	MOV ECX, dword ptr [ESP + 0x4]	
Analysis	Found Code	Found code from operand reference	004025a0	LAB_004025a0	PUSH EBP	
Analysis	Function ID Analyzer	Library Function - Single Match, _memset	00401990	_memset	MOV EDI, dword ptr [ESP + _Size]	
Analysis	Function ID Analyzer	Library Function - Single Match, _strncat	00401930	_strncat	MOV ECX, dword ptr [ESP + _Co...	
Analysis	Function ID Analyzer	Library Function - Single Match, _strlen	00401a60	_strlen	MOV ECX, dword ptr [ESP + _Str]	
Analysis	Function ID Analyzer	Library Function - Single Match, __exit	00401c41	__exit	PUSH 0x0	
Analysis	Function ID Analyzer	Library Function - Single Match, __global_unwind2	004024a8	__global_unwind2	PUSH EBP	
Analysis	Function ID Analyzer	Library Function - Single Match, __local_unwind2	004024ea	__local_unwind2	PUSH EBX	
Analysis	Function ID Analyzer	Library Function - Single Match, _malloc	00402930	_malloc	PUSH dword ptr [DAT_00405438]	
Analysis	Function ID Analyzer	Library Function - Single Match, __nh_malloc	00402942	__nh_malloc	OPB dword ptr [ESP + _Size],...	
Analysis	Function ID Analyzer	Library Function - Single Match, _strcpy	00403660	_strcpy	MOV ECX, dword ptr [ESP + _Co...	
Analysis	Address Table	Address table[3] created	00401654	switchdata0_00401654	addr switch0_0040163d::case0_1	
Analysis	Address Table	Address table[4] created	00401738	switchdata0_00401738	addr switch0_00401625::case0_0	
Analysis	Address Table	Address table[3] created	004017dc	switchdata0_004017dc	addr switch0_004017cd::case0_1	
Analysis	Address Table	Address table[4] created	004018d0	switchdata0_004018d0	addr switch0_004017a7::case0_0	
Analysis	Address Table	Address table[3] created	00403b84	switchdata0_00403b84	addr switch0_00403b6d::case0_1	
Analysis	Address Table	Address table[4] created	00403c68	switchdata0_00403c68	addr switch0_00403b55::case0_0	
Analysis	Address Table	Address table[3] created	00403d0c	switchdata0_00403d0c	addr switch0_00403cf1::case0_1	
Analysis	Address Table	Address table[4] created	00403e00	switchdata0_00403e00	addr switch0_00403cd7::case0_0	
Analysis	Address Table	Address table[2] created	00404424	addr LAB_00403889		
Analysis	Address Table	Address table[2] created	00404430	addr LAB_0040393d		
Analysis	Address Table	Address table[2] created	0040443c	addr LAB_00403ac1		
Analysis	Suspicious Sequence	Function 'FUN_004010ea' contains suspicious API sequence.	004010ea	FUN_004010ea	PUSH EBP	
Info	PE Header	IMAGE_DIRECTORY_ENTRY_IMPORT	00404444	DWORD_00404444	ddw 446ch	

Image 7: Bookmarks the function which contains suspicious API call pattern

Ghidra's scripting output, as shown in the screenshots, highlights this sequence of calls and their corresponding addresses within the binary. The output logs confirm a suspicious pattern match, flagging the function and marking it for further review. This sequence aligns with known Thread Manipulation techniques, reinforcing the method's malicious intent.

Case Study 3: Windows Hook Manipulation Injection - [Lab12-03.exe](#)

The third case study of [Lab12-03.exe](#) demonstrates the **Hook Manipulation** technique, a common approach for DLL injection where system hooks are exploited to inject malicious code into targeted processes. This method begins with a call to **AllocConsole** at address **0x0040100d**, followed by subsequent calls to **FindWindowA** and **ShowWindow**. These calls suggest that the program is locating and interacting with a specific window, possibly to prepare the environment for further actions.

The critical point in this sequence is the call to **SetWindowsHookExA** at address **0x0040105b**, a key indicator of hook-based DLL injection. This function installs a hook procedure that allows the malicious DLL to intercept system events like keyboard inputs or mouse movements. Immediately after, calls to **GetMessageA** and **UnhookWindowsHookEx** are observed, which retrieve system messages and remove the hook, respectively, completing the injection process. The function then invokes **CallNextHookEx**, ensuring the smooth transfer of control to the next hook in the chain.

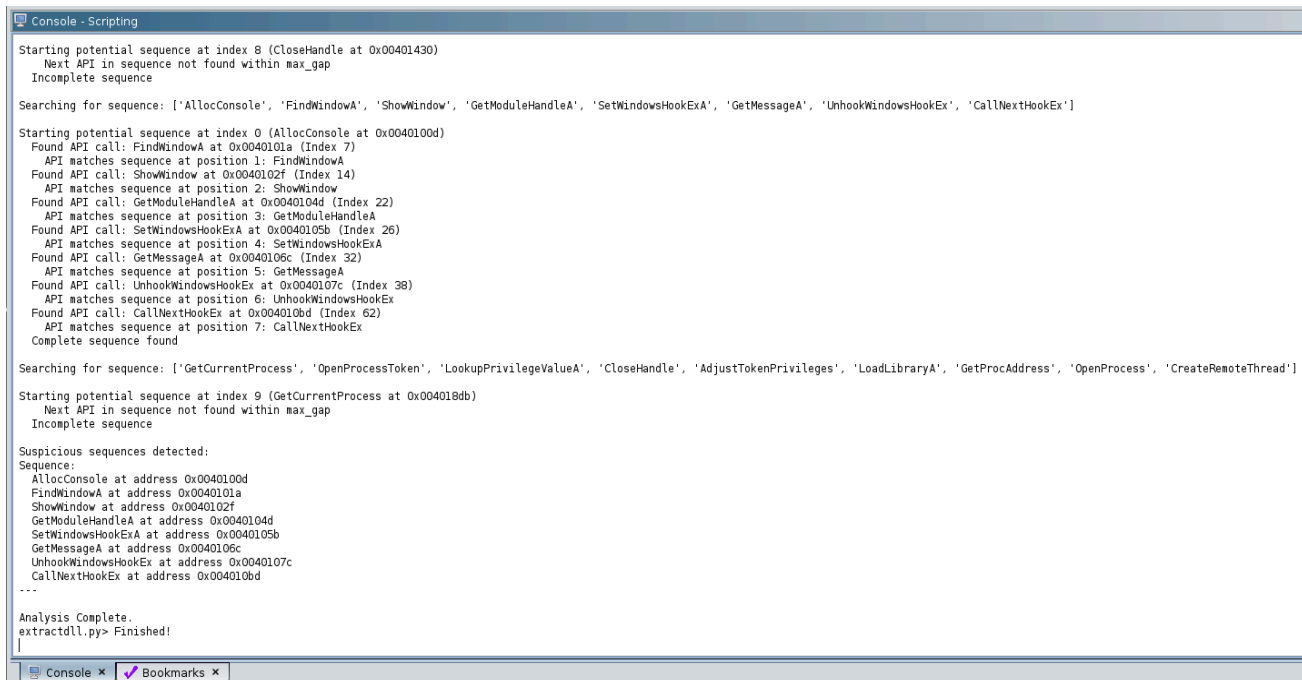


Image 8: Analysis of the binary to search for suspicious patterns and confirmation of suspicious API call patterns in the binary

The Console - Scripting output in image 8 clearly shows the sequence of API calls, including their addresses, as the script matches them against the predefined Hook Manipulation pattern. The output indicates that the complete sequence was detected successfully, with each API call aligned sequentially, confirming malicious activity. Additionally, the Bookmarks section in image 9 highlights the function **FUN_004010fc** where the suspicious sequence resides, providing a precise location within the binary for further investigation.

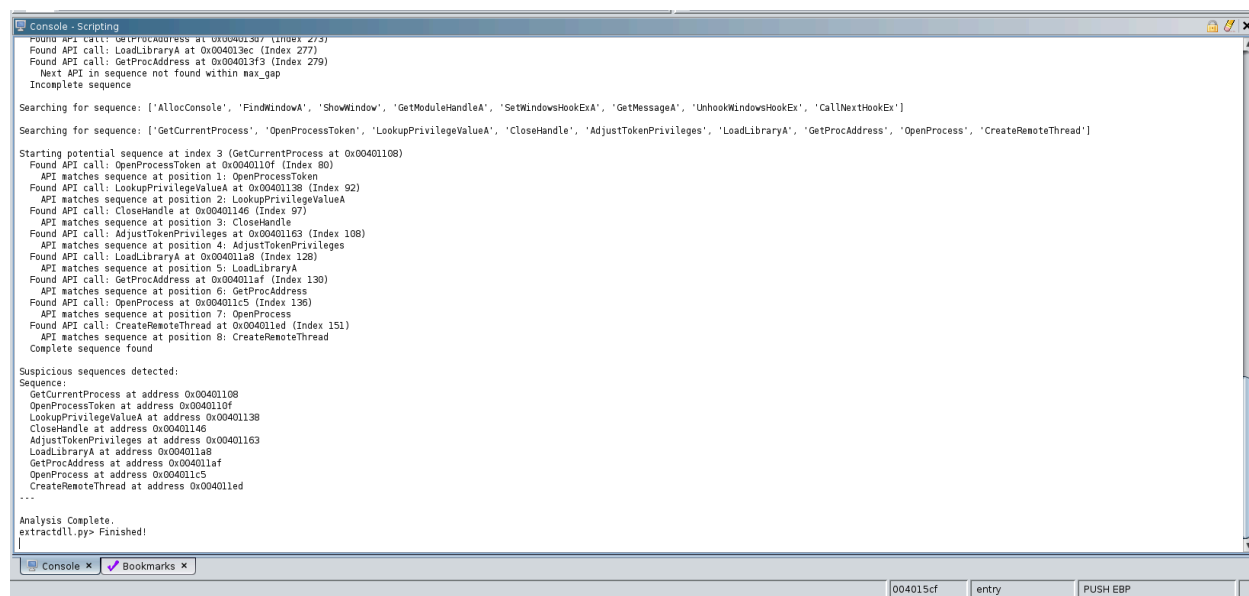
Bookmarks - (6 bookmarks)						
Type	Category	Description	Location	Label	Code Unit	
Analysis	Found Code	Found code from operand reference	004016fe	LAB_004016fe	XOR EAX, EAX	
Analysis	Address Table	Address table[2] created	0040209c		addr LAB_004016bf	
Analysis	Suspicious Sequence	Function 'FUN_004010fc' contains suspicious API sequence.	004010fc	FUN_004010fc	PUSH EBP	
Info	PE Header	IMAGE_DIRECTORY_ENTRY_IMPORT	004020a4	DWORD_004020a4	ddw 2104h	
Info	PE Header	IMAGE_DIRECTORY_ENTRY_RESOURCE	00404000	IMAGE_RESOURCE_DIRECTORY_00...	IMAGE_RESOURCE_DIRECTORY	
Info	PE Header	IMAGE_DIRECTORY_ENTRY_IAT	00402000	PTR_OpenProcessToken_004020...	addr ADVAPI32.DLL::OpenProce...	

Image 9: Bookmarks the function which contains suspicious API call pattern

By capturing and correlating API calls such as **SetWindowsHookExA** and **UnhookWindowsHookEx**, the script effectively identifies this form of DLL injection. This case study underscores the tool's reliability in pinpointing hook manipulation techniques by analyzing the flow of system APIs and recognizing malicious behavior based on predefined patterns. The detection process is streamlined, with both images offering clear evidence of how the suspicious sequence was located and flagged within the binary.

Case Study 4: Injection by Privilege Escalation - [Lab12-04.exe](#)

For the fourth and the last malicious binary case study, **Injection by Privilege Escalation** has been identified in [Lab12-04.exe](#). The analysis highlights a clear sequence of suspicious system API calls associated with privilege escalation and DLL injection. The script first detects the invocation of **OpenProcessToken**, a commonly used API to obtain access tokens of processes. This is immediately followed by **LookupPrivilegeValueA** and **AdjustTokenPrivileges**, which adjust process permissions—often exploited to escalate privileges.



```
Console - Scripting
Found API call: GetProcAddress at 0x0040130f (Index 273)
Found API call: LoadLibraryA at 0x004013ec (Index 277)
Found API call: GetProcAddress at 0x004013f3 (Index 279)
Next API in sequence not found within max_gap
Incomplete sequence

Searching for sequence: ['AllocConsole', 'FindWindow', 'ShowWindow', 'GetModuleHandleA', 'SetWindowsHookEx', 'SendMessage', 'UnhookWindowsHookEx', 'CallNextHookEx']

Searching for sequence: ['GetCurrentProcess', 'OpenProcessToken', 'LookupPrivilegeValueA', 'CloseHandle', 'AdjustTokenPrivileges', 'LoadLibraryA', 'GetProcAddress', 'OpenProcess', 'CreateRemoteThread']

Starting potential sequence at index 3 (GetCurrentProcess at 0x00401108)
Found API call: OpenProcessToken at 0x0040110f (Index 80)
API matches sequence at position 1: OpenProcessToken
Found API call: LookupPrivilegeValueA at 0x00401138 (Index 92)
API matches sequence at position 2: LookupPrivilegeValueA
Found API call: CloseHandle at 0x00401146 (Index 97)
API matches sequence at position 3: CloseHandle
Found API call: AdjustTokenPrivileges at 0x00401163 (Index 108)
API matches sequence at position 4: AdjustTokenPrivileges
Found API call: LoadLibraryA at 0x004011a8 (Index 128)
API matches sequence at position 5: LoadLibraryA
Found API call: GetProcAddress at 0x004011af (Index 130)
API matches sequence at position 6: GetProcAddress
Found API call: OpenProcess at 0x004011c5 (Index 136)
API matches sequence at position 7: OpenProcess
Found API call: CreateRemoteThread at 0x004011ed (Index 151)
API matches sequence at position 8: CreateRemoteThread
Complete sequence found

Suspicious sequences detected:
Sequence:
GetCurrentProcess at address 0x00401108
OpenProcessToken at address 0x0040110f
LookupPrivilegeValueA at address 0x00401138
CloseHandle at address 0x00401146
AdjustTokenPrivileges at address 0x00401163
LoadLibraryA at address 0x004011a8
GetProcAddress at address 0x004011af
OpenProcess at address 0x004011c5
CreateRemoteThread at address 0x004011ed
...

Analysis Complete.
extractdll.py> Finished!
```

Image 10: Analysis of the binary to search for suspicious patterns and confirmation of suspicious API call patterns in the binary

The script logs the progression of these calls, showcasing how the suspicious sequence develops. After the privileges are modified, **LoadLibraryA** is used to load a dynamic link library, a critical operation in DLL injection. Subsequent calls to **GetProcAddress** and **CreateRemoteThread** complete the injection process, as the malicious code is executed within the target process.

The output console in image 10 marks this progression, listing each matched API call and the corresponding addresses where they occur within the binary. Additionally, the bookmark view in image 11 confirms the identification of the suspicious sequence under the function

FUN_00401000, facilitating easy navigation to the flagged areas for further investigation.

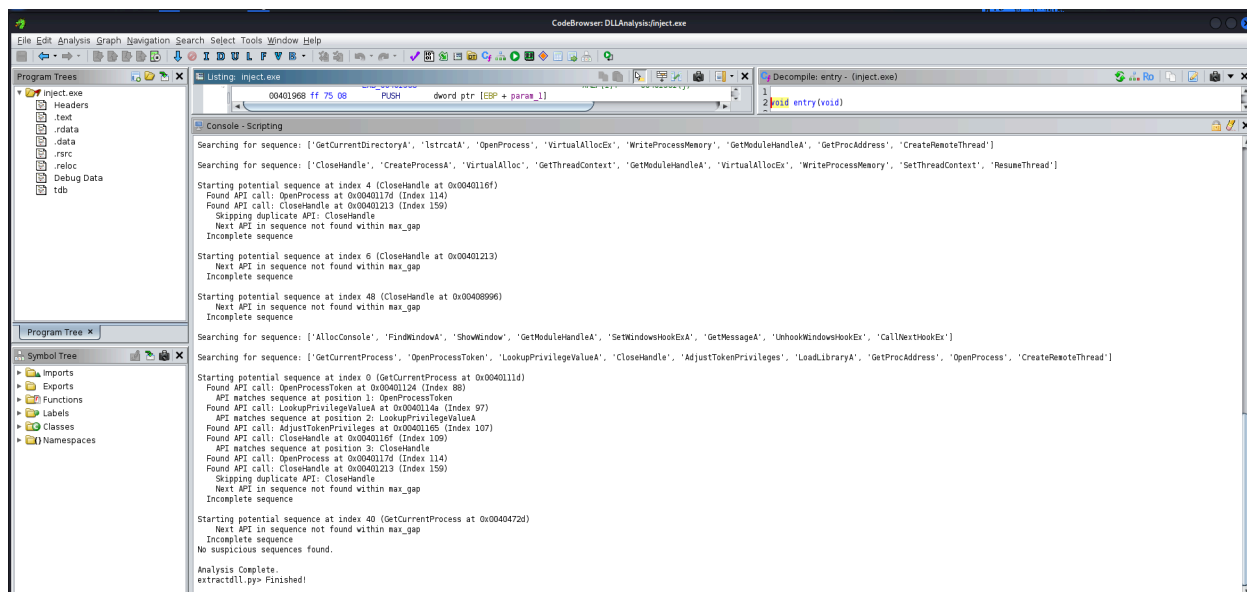
Bookmarks - (25 bookmarks)						
Type	Category	Description	Location	Label	Code Unit	
Analysis	Found Code	Found code from operand reference	00402144	LAB_00402144	MOV ECX, dword ptr [ESP + 0x4]	
Analysis	Found Code	Found code from operand reference	0040221c	LAB_0040221c	PUSH EBP	
Analysis	Function ID Analyzer	Library Function - Single Match, _memset	004014f0	_memset	MOV EDI, dword ptr [ESP + _Size]	
Analysis	Function ID Analyzer	Library Function - Single Match, _strncpy	00401550	_strncpy	MOV ECX, dword ptr [ESP + _Co...	
Analysis	Function ID Analyzer	Library Function - Single Match, _strlen	00401650	_strlen	MOV ECX, dword ptr [ESP + _Str]	
Analysis	Function ID Analyzer	Library Function - Single Match, _strcmp	004016d0	_strcmp	MOV EDI, dword ptr [ESP + _Str1]	
Analysis	Function ID Analyzer	Library Function - Single Match, _exit	004018ba	_exit	PUSH 0x0	
Analysis	Function ID Analyzer	Library Function - Single Match, __global_unwind2	00402124	__global_unwind2	PUSH EBP	
Analysis	Function ID Analyzer	Library Function - Single Match, __local_unwind2	00402166	__local_unwind2	PUSH EBX	
Analysis	Function ID Analyzer	Library Function - Single Match, _malloc	004025a0	_malloc	PUSH dword ptr [DAT_00405cc8]	
Analysis	Function ID Analyzer	Library Function - Single Match, __nh_malloc	004025b2	__nh_malloc	CMV dword ptr [ESP + _Size],...	
Analysis	Address Table	Address table[3] created	00402a64	switchdata0_00402a64	addr switch0_00402a4d::case0_1	
Analysis	Address Table	Address table[4] created	00402b48	switchdata0_00402b48	addr switch0_00402a35::case0_0	
Analysis	Address Table	Address table[3] created	00402bec	switchdata0_00402bec	addr switch0_00402bdc::case0_1	
Analysis	Address Table	Address table[4] created	00402ce0	switchdata0_00402ce0	addr switch0_00402b77::case0_0	
Analysis	Address Table	Address table[3] created	00403a24	switchdata0_00403a24	addr switch0_00403a0d::case0_1	
Analysis	Address Table	Address table[4] created	00403b08	switchdata0_00403b08	addr switch0_004039f5::case0_0	
Analysis	Address Table	Address table[3] created	00403bac	switchdata0_00403bac	addr switch0_00403b9d::case0_1	
Analysis	Address Table	Address table[4] created	00403ca0	switchdata0_00403ca0	addr switch0_00403b77::case0_0	
Analysis	Address Table	Address table[2] created	0040440c	addr LAB_00403732		
Analysis	Address Table	Address table[2] created	00404418	addr LAB_004037e6		
Analysis	Address Table	Address table[2] created	00404424	addr LAB_004039e6		
Analysis	Suspicious Sequence	Function 'FUN_00401000' contains suspicious API sequence.	00401000	FUN_00401000	PUSH EBP	
Info	PE Header	IMAGE_DIRECTORY_ENTRY_IMPORT	0040442c	DWORD_0040442c	ddw 4468h	
Info	PE Header	IMAGE_DIRECTORY_ENTRY_IAT	00404000	PTR_GetModuleHandleA_00404000	addr KERNEL32.DLL::GetModule...	

Image 11: Bookmarks the function which contains suspicious API call pattern

This analysis underscores the methodical sequence of privilege escalation leading to DLL injection, where the manipulation of process tokens and dynamic library loading serve as strong indicators of malicious activity.

Case Study 5: Reflective DLL Injection - inject.exe

In this case study, the binary under analysis demonstrates Reflective DLL Injection. However, unlike previous cases, the script fails to flag this injection technique as malicious due to its inability to match a sufficiently large sequence of suspicious API calls.



`adjustTokenPrivileges`, `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. These APIs are well-known components of privilege escalation and injection processes. However, the absence of an extended, contiguous chain of these calls prevents the script from categorizing the behavior as definitively malicious.

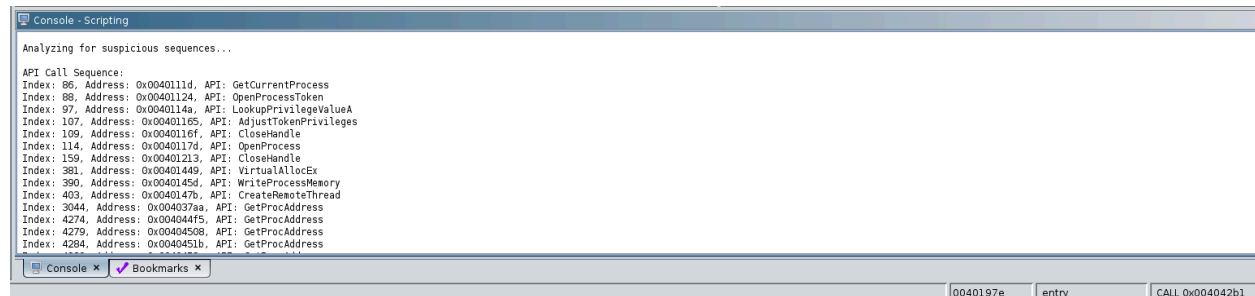


Image 13: List of the binary's system API calls

The script relies on identifying patterns that consist of a specific sequence of API calls, as established in earlier case studies. In this case, while the individual API calls are suspicious on their own, the gap between consecutive calls or the presence of other benign functions might disrupt the pattern recognition logic. This disruption results in incomplete sequence detection, as highlighted in the output.

This case emphasizes the limitations of strict pattern-matching techniques when analyzing binaries that exhibit fragmented execution flows or non-linear API calls. To address such issues, the script could be improved by implementing loose sequence matching with configurable thresholds for allowable gaps between suspicious API calls. Additionally, considering the frequency of certain high-risk APIs, such as `VirtualAllocEx` and `WriteProcessMemory`, could help prioritize suspicious behaviors even in fragmented sequences.

By analyzing Image 13, it is evident that the reflective injection does employ injection-related APIs, but their execution order and gaps within the sequence result in the script failing to recognize the overall malicious activity. This reinforces the need for more adaptive detection mechanisms capable of handling incomplete or irregular patterns.

Case Study 6: Benign Binaries and the Importance of Patterns in DLL Injection

This case study focuses on benign binaries such as `rundll32.exe` and `rrinstaller.exe`, which utilize system API calls as part of their legitimate functionality. Unlike the malicious binaries previously analyzed, these binaries follow API call patterns that do not align with malicious DLL injection techniques. While all system API calls observed in both benign and malicious binaries are inherently legitimate, it is the sequence and combination of these calls that determine whether the behavior indicates a potential DLL injection attack.

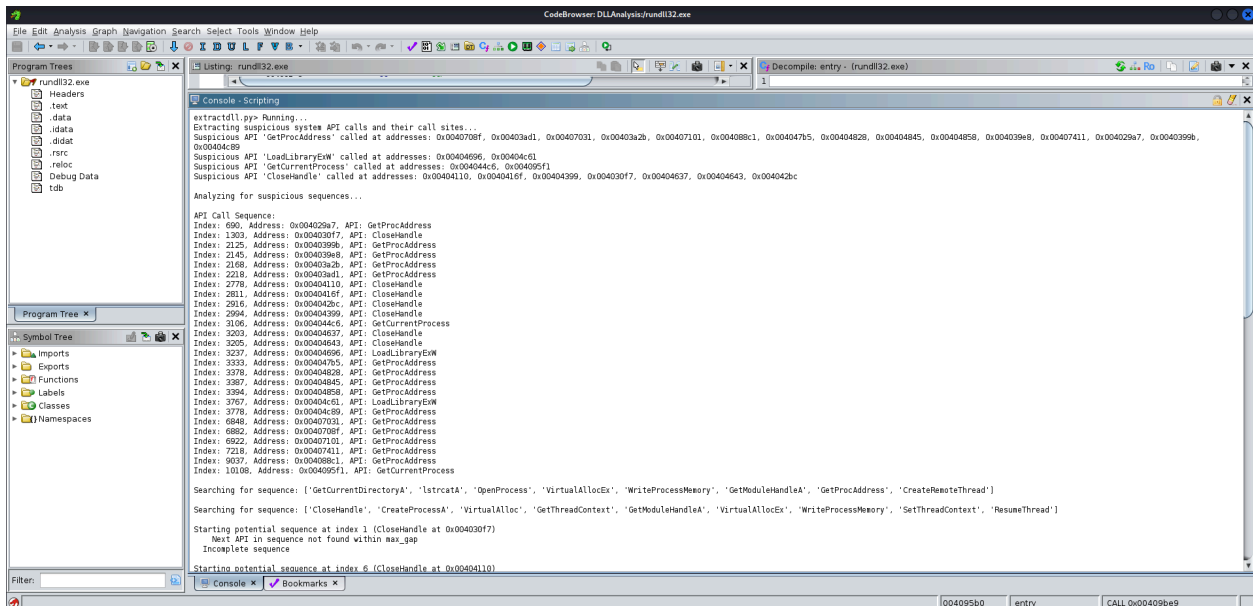


Image 14: Analysis of the benign binary to search for suspicious patterns

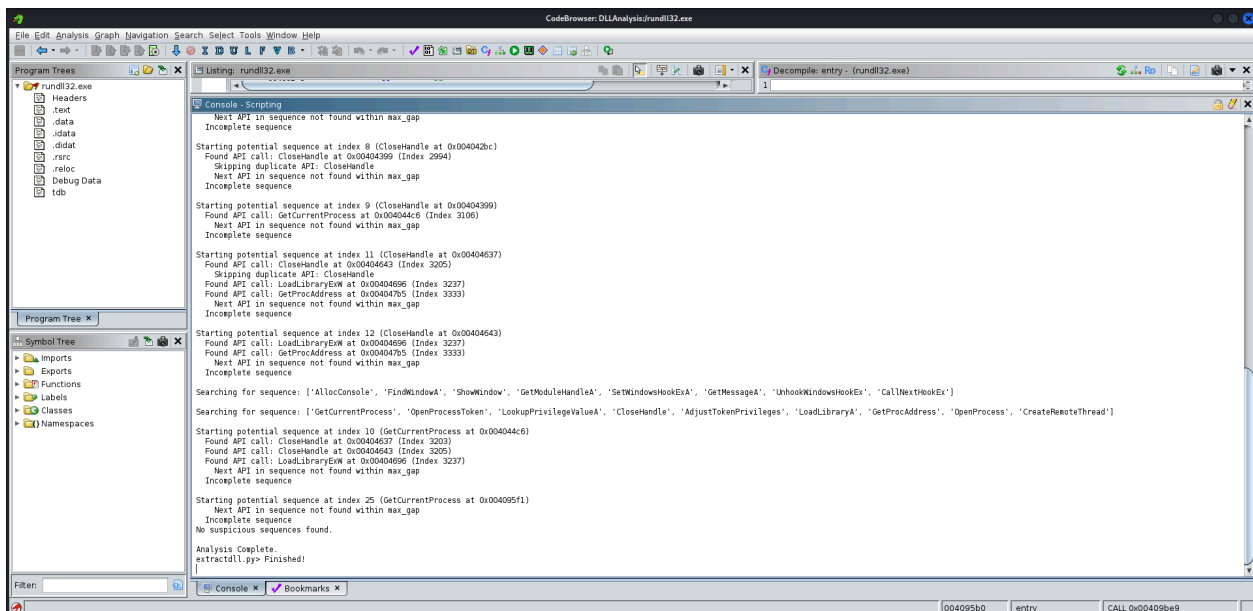


Image 15: No Suspicious Sequence Found, an indication of benign binary, and no false positives are detected

In the case of `rundll32.exe`, captured in Image 14 and Image 15, the analysis reveals frequent usage of APIs like `LoadLibraryExW`, `GetProcAddress`, and `CloseHandle`. These APIs are standard for dynamically loading libraries and resolving function addresses during normal operations. While `CloseHandle` and `GetProcAddress` also appear in malicious binaries, the absence of complementary calls such as `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` distinguishes benign behavior from malicious patterns.

The second binary, `rrinstaller.exe`, analyzed in Image 16, also demonstrates API sequences involving `LoadLibraryExA` and `FlushInstructionCache`. These calls are necessary for ensuring the successful loading and execution of code in memory. However, there is no evidence of thread manipulation or external memory writing, which are hallmarks of malicious injections.

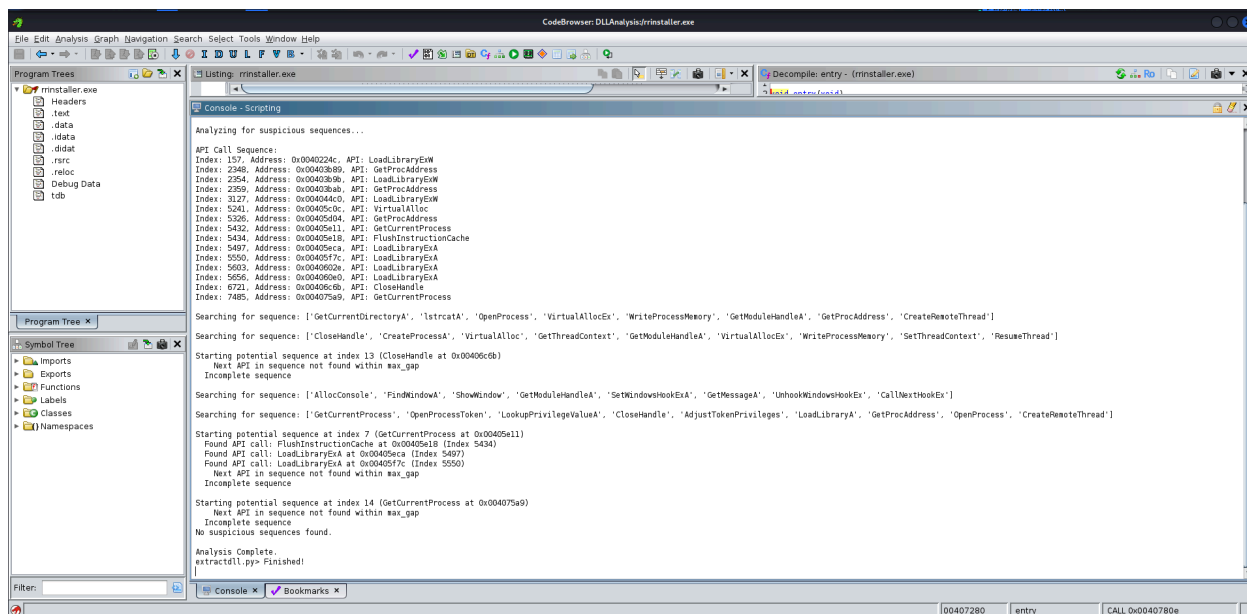


Image 16: No Suspicious Sequence Found, an indication of benign binary, and no false positives are detected

The analysis of `rundll32.exe` and `rrinstaller.exe` highlights this distinction. While some suspicious APIs are present, they are not part of a malicious sequence. This finding underscores the importance of analyzing API call patterns rather than flagging individual calls in isolation. As seen in Image 14, Image 15, and Image 16, the behavior is consistent with legitimate tasks required for loading and executing DLLs within the application's context.

By understanding the patterns that distinguish benign binaries from malicious ones, the analysis avoids false positives and ensures more accurate identification of malicious behavior. This distinction reinforces that DLL injection detection relies not on the APIs themselves but on how they are used together.

The analysis of both malicious and benign binaries here demonstrates that no single system API call is inherently malicious. Instead, it is the sequence and combination of calls, such as memory allocation, process manipulation, and thread creation, that signals a potential DLL injection attack. By identifying these patterns, we can distinguish between legitimate processes and malicious activities, as demonstrated across the case studies.

Results & Accuracy Check

Accuracy Metrics and Their Relevance

To evaluate the performance of the detection system, key accuracy metrics such as True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) are utilized. True Positives indicate correctly identified malicious binaries, while True Negatives account for benign binaries accurately classified as non-malicious. False Positives refer to benign binaries incorrectly flagged as malicious, and False Negatives represent malicious binaries that go undetected.

These metrics are critical because they feed into other evaluation parameters like Precision, Recall, and the F1-Score. Precision measures the proportion of correct positive predictions against the total predicted positives, ensuring the reliability of identifying malicious binaries. Recall evaluates the system's ability to detect all malicious binaries by analyzing the ratio of True Positives to the sum of True Positives and False Negatives. The F1-Score, which combines Precision and Recall into a single value, balances these aspects and provides a comprehensive measure of system performance.

Extracting Necessary Information from Analyzing Binaries

The analysis begins with identifying suspicious patterns of API calls indicative of DLL injections. A custom script, `extractdll.py`, is employed within Ghidra to search for sequences of system API calls known to signal DLL injection behavior. To streamline the process, Ghidra's Headless Analyzer is leveraged through a shell script, which automates the analysis of binaries and generates a CSV output file containing detection results.

This shell script executes `extractnimport2csv.py`, which focuses solely on extracting filenames and their corresponding detection results (0 for benign and 1 for malicious). The output is saved in a file named `detection_results.csv`. To enhance clarity, a filtered version of the file is created, `filtered.csv`, containing only the filename and detection values. This filtered data serves as the predicted results for comparison against a ground-truth file, `actual_labels.csv`, which contains the actual classifications of the binaries.

Using the Accuracy Metrics to Compare Malicious and Benign Binaries

To measure the accuracy of the detection system, the filtered results are compared to the actual labels. A Python program, `calculate_f1_score.py`, merges the filtered predictions with the ground truth data and calculates the accuracy metrics. The confusion matrix and evaluation metrics, including Precision, Recall, and F1-Score, are derived from this comparison.

The results demonstrate that 34 True Negatives and 4 True Positives were identified, with 1 False Negative, as shown in the output. Notably, no False Positives were reported, which means that no benign binaries were incorrectly flagged as malicious. The False Negative indicates that a malicious binary went undetected, highlighting a gap in the detection capability.

```

(kali@kali)-[~/Desktop]
$ python3 calculate_f1_score.py
Loaded 39 records from filtered.csv.
Loaded 39 records from actual_labels.csv.
Merged DataFrame has 39 records.

Merged DataFrame Preview:

```

	Filename	Detection	Actual
0	bitsadmin.exe	0	0
1	bridgeunattend.exe	0	0
2	ByteCodeGenerator.exe	0	0
3	CloudExperienceHostBroker.exe	0	0
4	CredentialEnrollmentManager.exe	0	0

```

Confusion Matrix:
True Negatives (TN): 34
False Positives (FP): 0
False Negatives (FN): 1
True Positives (TP): 4

Evaluation Metrics:
Precision: 1.00
Recall: 0.80
F1 Score: 0.89

Classification Report:

```

	precision	recall	f1-score	support
0	0.97	1.00	0.99	34
1	1.00	0.80	0.89	5
accuracy			0.97	39
macro avg	0.99	0.90	0.94	39
weighted avg	0.98	0.97	0.97	39

Image 17: Output of the accuracy check

The final accuracy achieved stands at 97%, which is significant but falls short of perfection due to the False Negative. The Precision of the system is measured at 1.00, indicating that every positive detection was accurate. However, the Recall is slightly reduced to 0.80 due to the undetected malicious file. Consequently, the combined F1-Score is 0.89, reflecting a strong overall performance with room for improvement.

Final Accuracy Results

The evaluation reveals that the detection system performs reliably with high accuracy, as seen in the confusion matrix and classification report. Nevertheless, the False Negative underscores the need for refining detection patterns to identify more subtle malicious behaviors. Expanding the dataset and incorporating advanced indicators such as obfuscation patterns could further enhance the system's robustness.

Conclusion

The analysis conducted in this study provides valuable insights into detecting DLL injection techniques by analyzing patterns of system API calls. While the detection system demonstrates promising results, achieving an overall accuracy of 97% with an F1-Score of 0.89, several challenges and limitations are identified that pave the way for further improvements.

One of the primary challenges encountered during this study was the reliance on static code analysis. While static analysis excels at identifying known patterns in binary files, it struggles with obfuscated code or techniques that dynamically resolve API calls at runtime. This limitation occasionally leads to False Negatives, where malicious binaries exhibiting unconventional behavior evade detection. Addressing this requires integrating dynamic analysis techniques to complement static methods, providing a more comprehensive detection system.

In summarizing the findings, the study successfully demonstrated that patterns of legitimate system API calls can serve as strong indicators of DLL injection. By analyzing these sequences through a custom script, the detection system was able to differentiate between benign and malicious binaries with high precision. However, as observed in the results, the absence of certain patterns and the presence of False Negatives highlight areas for refinement.

The significance of these findings becomes even more evident in dynamic environments, where real-time detection of DLL injection is critical. Security systems that integrate both static and dynamic analysis can leverage the observed API patterns to flag potential threats with greater accuracy. Additionally, incorporating obfuscation detection mechanisms and expanding the dataset to include diverse binaries will enhance the robustness of the detection system.

In conclusion, this work lays a foundation for identifying DLL injections through API call patterns while acknowledging the limitations of static analysis. The insights gained from this study not only contribute to improving detection techniques but also highlight the importance of combining static and dynamic methods to effectively combat sophisticated injection attacks in real-world scenarios. By addressing the challenges identified and refining the system further, this approach can play a crucial role in securing systems against evolving threats.

Appendix: Glossary of Technical Terms

Term	Definition
API (Application Programming Interface)	A set of tools and functions that let software programs interact with an operating system or other applications, making it easier to develop new applications.
DLL (Dynamic-Link Library)	A type of file that contains code and data that multiple programs can use at the same time, enabling modular program design and efficient code reuse.
DLL Injection	A method used to insert malicious code into a running process by forcing it to load a malicious DLL file.
Ghidra	An advanced open-source tool developed by the NSA for reverse engineering software, capable of decompiling executable files into more understandable programming languages like C.
Reverse Engineering	The process of breaking down software to understand how it works, often to find vulnerabilities or analyze malicious programs.
Static Analysis	The practice of examining a program's code or binary files without actually running the software, allowing the identification of potential threats in a controlled manner.
Reflective DLL Injection	A stealthy DLL injection technique where the DLL is loaded directly into memory without leaving any trace on the disk, making it harder to detect.
API Hooking	A process of intercepting API calls made by a program, often used for debugging or to manipulate program behavior, including injecting malicious code.
Obfuscation	A technique to make code more complex and difficult to understand, often employed by attackers to evade detection tools.
False Positive	When a detection system incorrectly identifies safe software as malicious.
False Negative	When a detection system fails to identify malicious software as a threat.

Appendix: Scripts Used

Script Name	Purpose	Functionality
ExtractDLL.py	Identifies suspicious sequences of API calls in binary files to detect DLL injection techniques.	Uses Ghidra scripting to extract and flag potentially harmful API call patterns from binaries.
ExtractnImport2CSV.py	Converts the analysis results into a CSV file for easy data handling and review.	Processes file names and detection outcomes, organizing the results into a structured CSV format.
CalculateF1Score.py	Computes performance metrics like Precision, Recall, and F1-Score by comparing detection results with known classifications.	Uses Python to analyze and evaluate the detection system's effectiveness through statistical measures.
Headless Analysis Shell Script - run_analysis.sh	Automates batch processing of binaries in Ghidra without the need for a graphical interface.	Coordinates the execution of the above scripts to streamline the analysis process and generate results efficiently.