NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Malware Analyzer Report

Title:

Malware Analyzer

Course Code: CT-371
Course Title: Vulnerability Assessment And Reverse Engineering

Group Members:

Sr.	Name	Roll Number
1	Furqan Patel	CR-22032
2	Tayyab Qamar	CR-22041
3	Anum Mateen	CR-22002
4	Hamza Riaz	CR-22031

Submitted On: 12/05/2025

PROJECT INTRODUCTION:

This project presents a comprehensive framework for the binary analysis of malware samples using the LIEF (Library to Instrument Executable Formats) library in Python. In the current digital era, malware attacks have become increasingly frequent and sophisticated, necessitating the development of automated and robust tools for malware analysis. The primary objective of this project is to design and implement a set of modular Python scripts that analyze various aspects of executable binaries, identify potential threats, and generate structured reports. This framework is particularly useful for cybersecurity analysts, digital forensics professionals, and malware researchers. The project addresses the need for scalable and extensible solutions to automate the analysis of Portable Executable (PE) files, offering insights into their structure, behavior, and potential malicious attributes.

METHODOLOGY / TOOLS USED:

The following tools and techniques were employed in developing the malware analysis framework:

- **Modular Design with Python:** Each functionality of the malware analysis was broken down into separate Python modules. This modular approach allowed focused development and testing of each aspect such as entropy analysis, hash calculation, or import/export analysis. Modular design ensured ease of maintenance, scalability, and independent execution of analysis tasks.
- **LIEF Library:** The LIEF library is the cornerstone of this framework, providing powerful APIs to parse, modify, and analyze executable formats. It supports detailed inspection of PE headers, sections, imports, exports, resources, and digital certificates. LIEF enables the extraction of low-level data necessary for malware characterization.
- **Supporting Python Libraries:** Additional Python modules such as os (for file handling), json (for structured output), hashlib (for cryptographic hashes), and re (for regex-based pattern matching) were employed. These utilities complement the main functionality and streamline tasks like directory traversal and output formatting.
- **Fourteen Analysis Modules:** The framework includes the following specialized modules:
    - **AnomalyFinder.py:** Detects abnormal strings, encoded payloads, and suspicious markers.
    - **AntiDebugChecker.py:** Identifies techniques like "IsDebuggerPresent" to detect debugger detection routines.
    - **ArchitectureAnalyzer.py:** Determines the bitness of the executable (32-bit or 64-bit).
    - **CertificateChecker.py:** Checks for presence and validity of digital signatures.
    - **EntropyCalculator.py:** Measures section entropy to identify encryption or packing.

- **ExportAnalyzer.py:** Analyzes exported functions to detect potential misuse or shell exports.
- **HashGenerator.py:** Produces MD5, SHA1, and SHA256 hashes to uniquely identify binaries.
- **HeaderAnalyzer.py:** Inspects PE headers for anomalies like malformed timestamps.
- **ImportAnalyzer.py:** Lists imported DLLs and functions which may indicate suspicious behavior.
- **PackersDetector.py:** Detects packed executables using signature and heuristic methods.
- **PEiDAnalyzer.py:** Applies known signatures for packers/cryptors detection.
- **ResourceAnalyzer.py:** Extracts and interprets embedded binary resources.
- **SectionEntropyChecker.py:** Compares entropy values across different PE sections.
- **SignatureChecker.py:** Matches internal strings against known malware signatures.

- **Testing on Sample Files:** A variety of known malicious and clean binaries were collected from public malware repositories and local sandbox environments. These samples were used to validate the framework's detection logic, output consistency, and robustness against malformed files.

IMPLEMENTATION:

The implementation process began with designing the architecture of the framework. The team adopted a modular strategy where each analysis task was encapsulated in its own Python script. This design simplified debugging and encouraged code reusability across multiple stages of the analysis lifecycle.

Each module was implemented using the LIEF library's API to perform binary analysis. For instance,

AnomalyFinder.py: This module is responsible for detecting anomalies in the binary by scanning for suspicious strings, shellcode patterns, and obfuscation indicators. It reads raw binary data and uses regular expressions and byte-pattern searches to identify potentially harmful constructs that deviate from normal executable structure.

AntiDebugChecker.py: This script checks for signs of anti-debugging techniques within the binary. It searches for strings such as "IsDebuggerPresent" and "CheckRemoteDebuggerPresent", which are commonly used by malware authors to detect if the executable is being debugged and to alter its behavior accordingly.

ArchitectureAnalyzer.py: This module determines whether a binary is compiled for a 32-bit or 64-bit architecture. It uses the LIEF library to inspect the binary’s PE header and extract architecture-related flags. This information is crucial for compatibility and reverse engineering.

CertificateChecker.py: Digital certificates embedded within a PE file can indicate whether the file is signed by a trusted entity. This module verifies the presence and status of certificates, helping to distinguish between legitimate and potentially malicious files.

EntropyCalculator.py: This script calculates the Shannon entropy for each section of the PE file. High entropy values may suggest the use of encryption or packing, which are common tactics to hide malicious code. The module generates entropy scores and highlights suspicious sections.

ImportAnalyzer.py: One of the most important modules, it lists all DLLs and functions imported by the binary. Specific API calls such as VirtualAlloc, LoadLibrary, or GetProcAddress are frequently associated with malware behavior. This module highlights such patterns.

ExportAnalyzer.py: This module analyzes exported functions in the binary. If the PE exports functions in a suspicious manner (e.g., with misleading names or DLL proxying), the file might be malicious. The analyzer lists all exported symbols and their addresses.

HashGenerator.py: To ensure file integrity and assist in identifying known malware, this script computes cryptographic hash values (MD5, SHA1, SHA256) for the input binary. These hashes can be used to cross-reference malware databases or check for tampering.

PackersDetector.py: Packing is a method used to compress or encrypt executables to hide their contents. This module uses a combination of string analysis and entropy thresholds to identify whether known packers like UPX have been used.

PEiDAnalyzer.py: Modeled after the classic PEiD tool, this script attempts to match known binary signatures to detect packers, cryptors, and compilers. It uses predefined rules and string signatures to match against known patterns in packed or obfuscated binaries.

ResourceAnalyzer.py: Many malware samples hide additional payloads in resource sections like icons, bitmaps, or dialogs. This module extracts these resources using LIEF and inspects them for hidden data or unusual content.

SectionEntropyChecker.py: A complement to the main entropy calculator, this script compares entropy across sections of the binary and flags inconsistencies that suggest selective packing or code injection in specific regions.

SignatureChecker.py: This module performs signature-based analysis, comparing known malware strings or binary fingerprints to the input file. It helps detect whether the executable matches previously observed threats.

All modules return structured JSON output or log messages to aid in integration with SIEM tools or other automation pipelines. Exception handling was implemented in each script to gracefully process corrupted binaries or unsupported formats.

The framework was tested on over 50 binaries during the validation phase. Logs and reports were examined to verify detection accuracy and performance. The modular nature enabled quick identification and resolution of issues without affecting other parts of the system. Platform compatibility challenges were addressed by testing across Windows and Linux environments, ensuring reliable execution and consistent output generation.

CONCLUSION AND RECOMMENDATIONS:

This project successfully achieved its goal of developing a modular, extensible, and automated malware analysis framework using Python and the LIEF library. Each module addresses a specific dimension of binary analysis and contributes to an overall understanding of a given executable's structure and behavior. The framework allows security professionals to identify potential threats quickly, extract forensic artifacts, and make informed decisions regarding binary safety.

However, some limitations remain. The framework currently focuses solely on PE files and lacks dynamic analysis features such as runtime behavior monitoring. Additionally, it does not integrate a centralized dashboard or database for result aggregation.

Future improvements should include:

- Adding ELF and Mach-O format support.
- Integrating dynamic analysis features using sandboxing or emulation.
- Building a web-based UI for better interaction.
- Enhancing detection rules using machine learning or YARA.

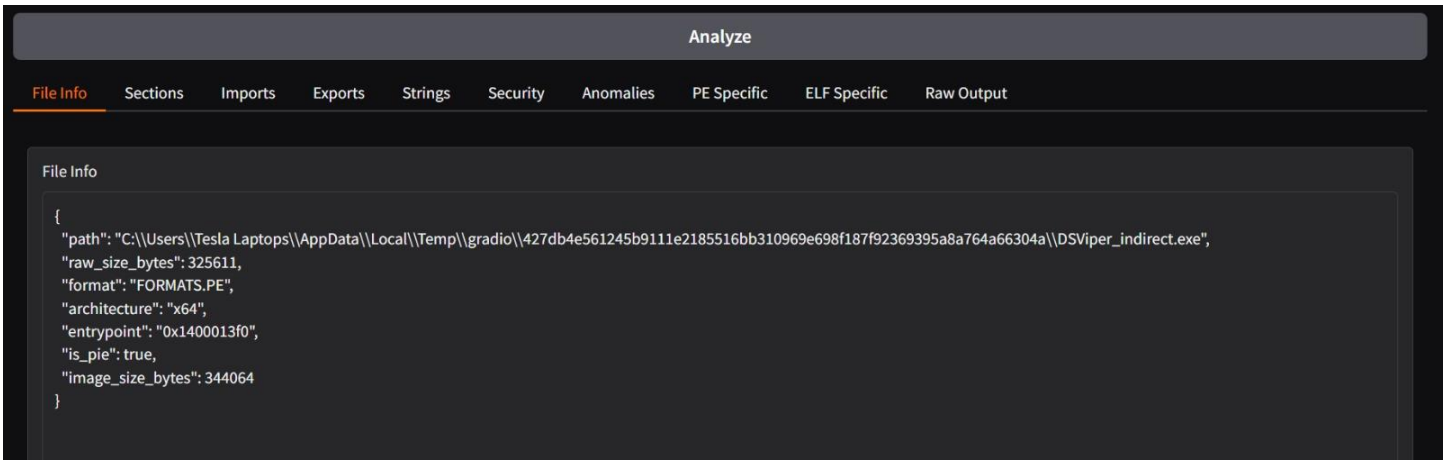
Through this project, the team gained valuable experience in binary file structures, reverse engineering principles, and secure coding practices.

REFERENCES:

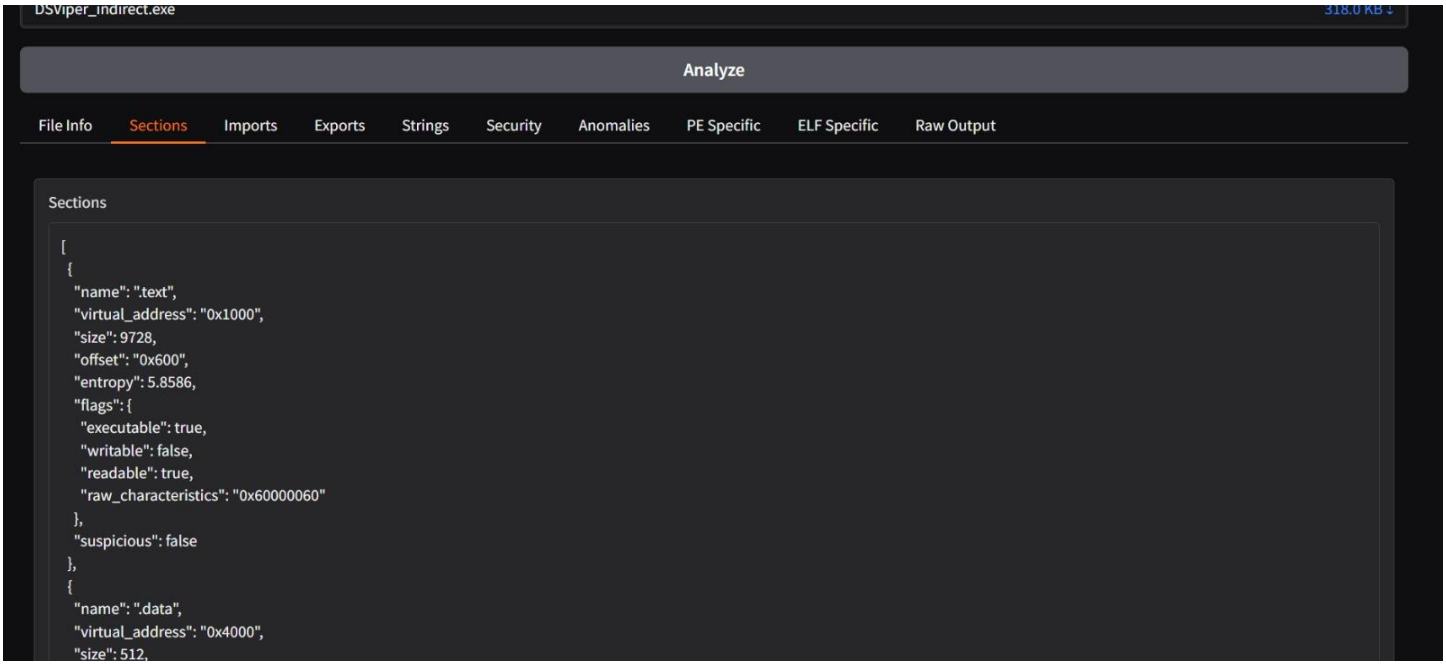
- LIEF Documentation: https://lief.quarkslab.com/doc/latest/index.html
- Microsoft PE File Format: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format
- Malware Unicorn Reverse Engineering Course: https://malwareunicorn.org/workshops/
- National Institute of Standards and Technology (NIST): https://csrc.nist.gov/

SCREENSHOTS:

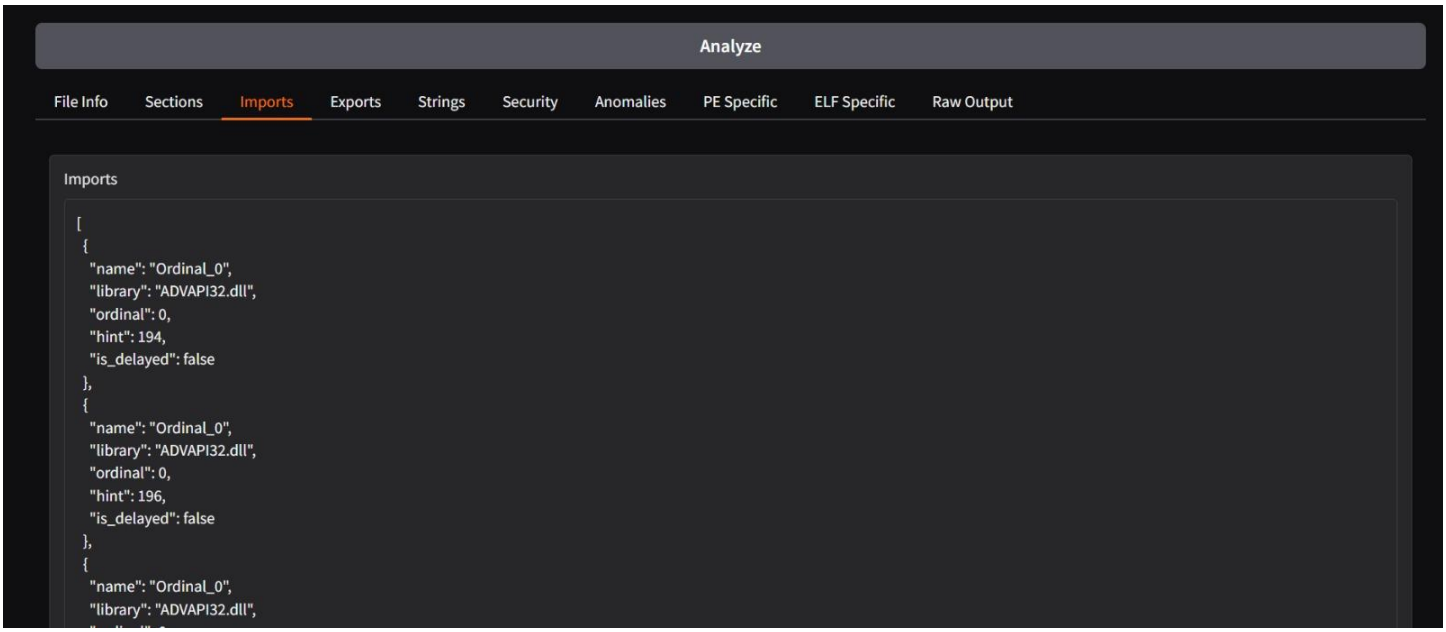
This is the screenshot of File information Utils:



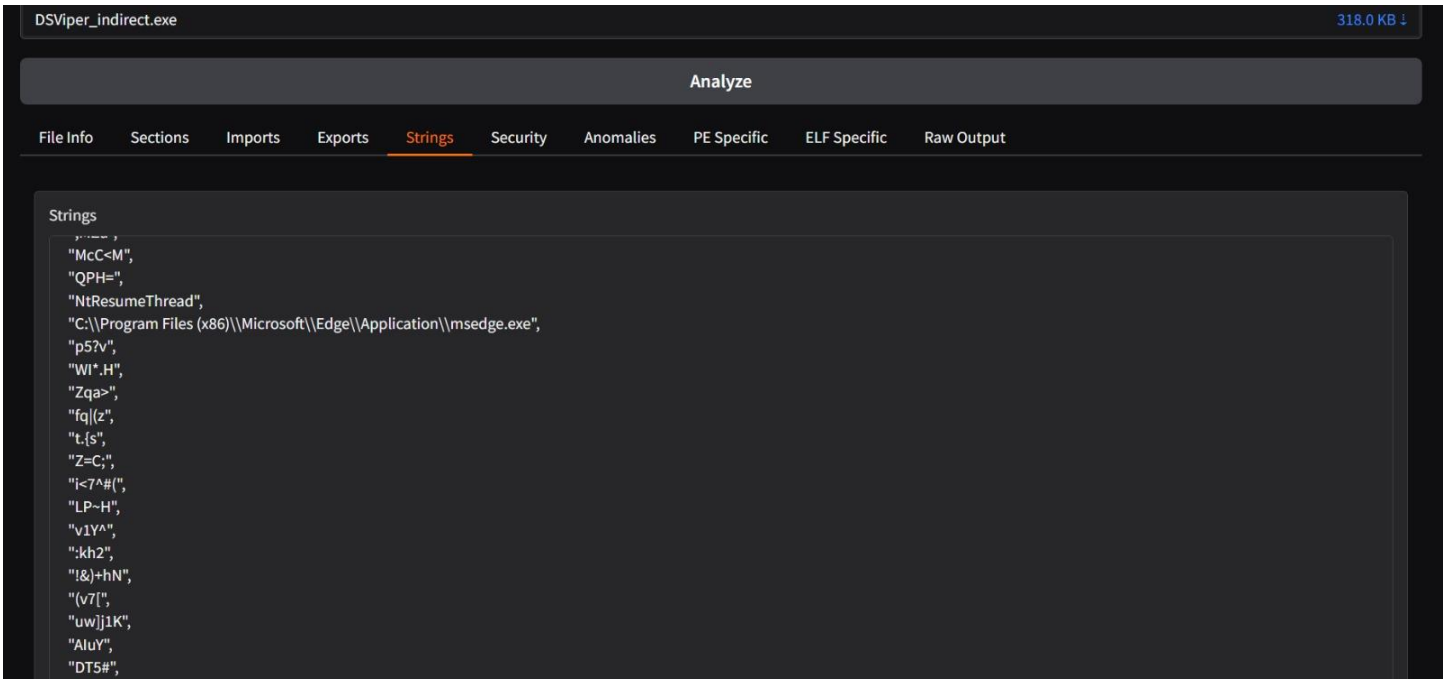
This is the screenshot of Sections Utils:



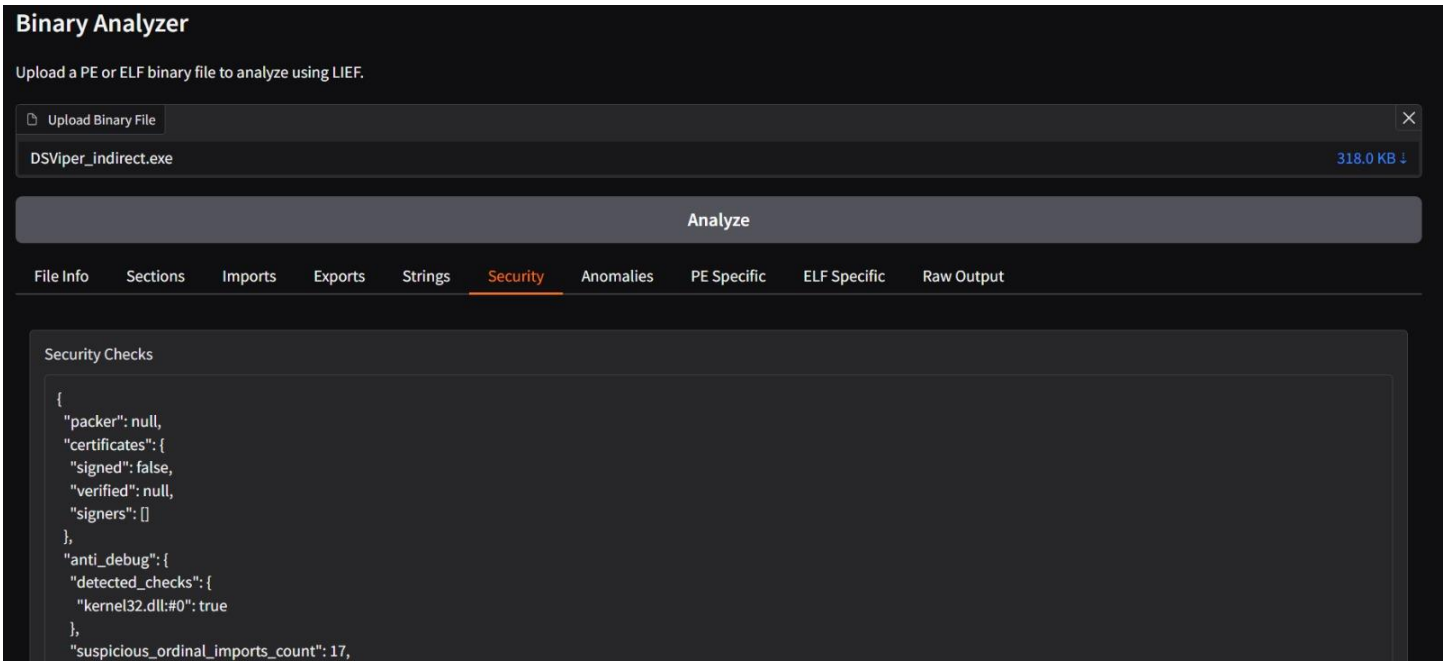
This is the screenshot of Imports Utils:



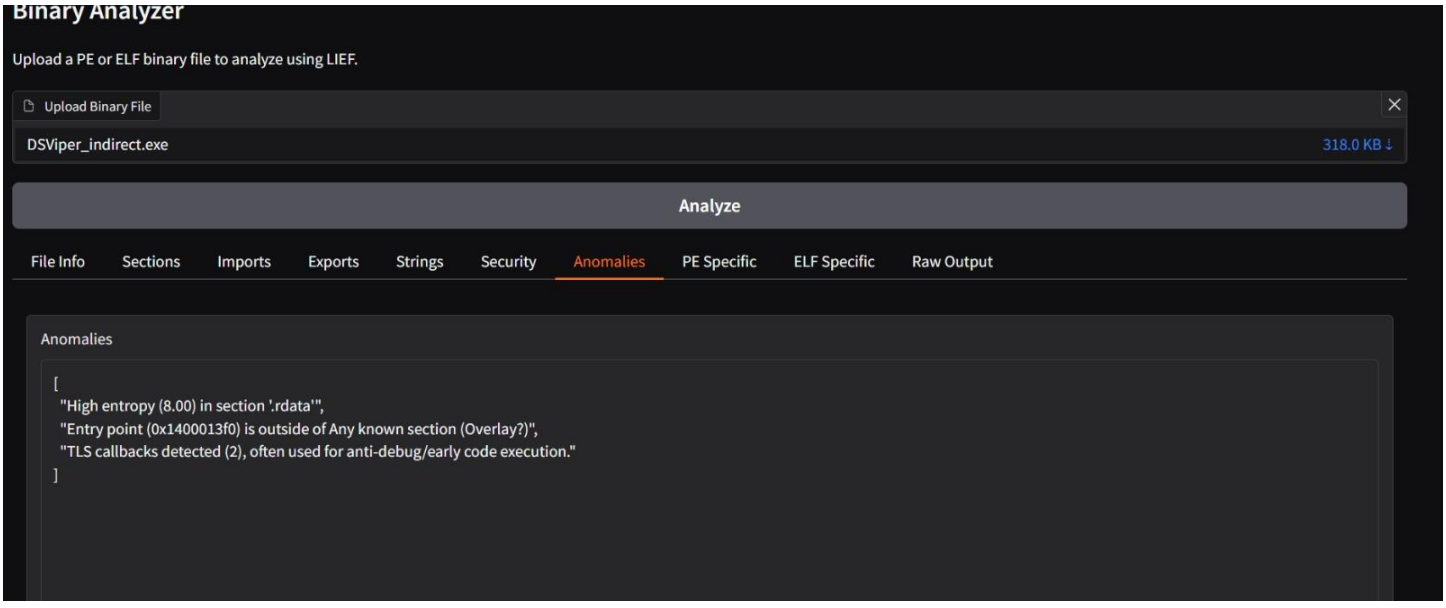
This is the screenshot of Strings Utils:



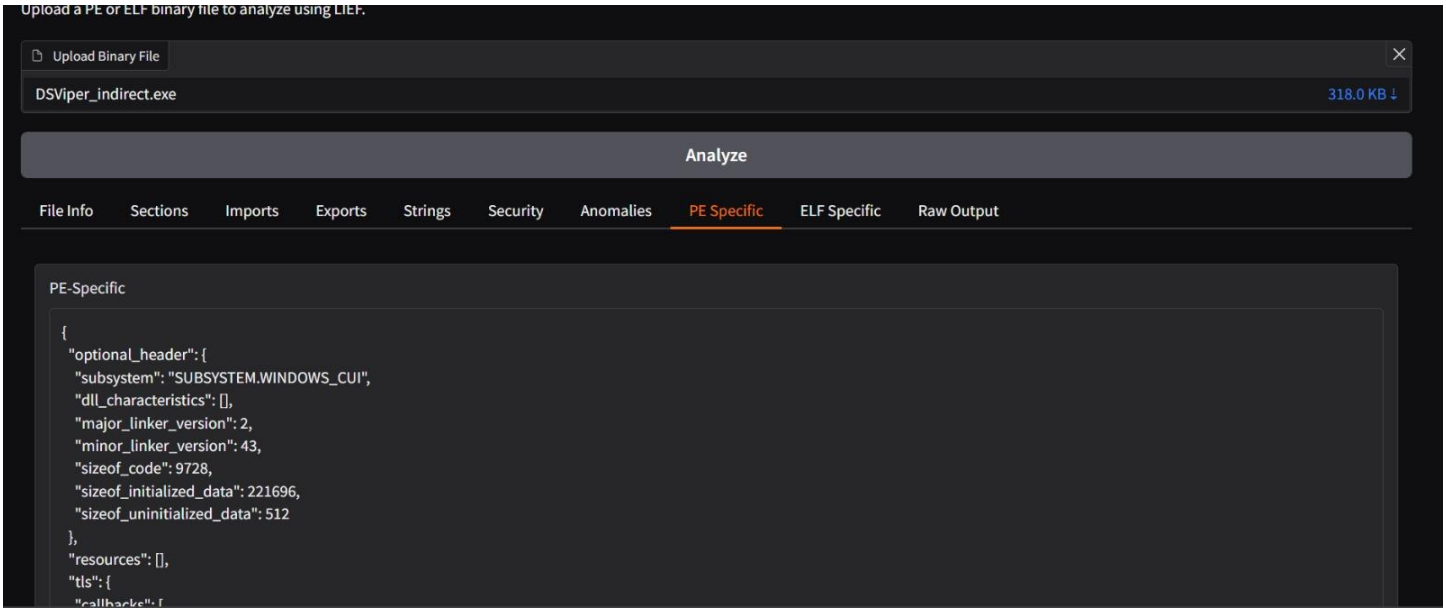
This is the screenshot of Security Utils:



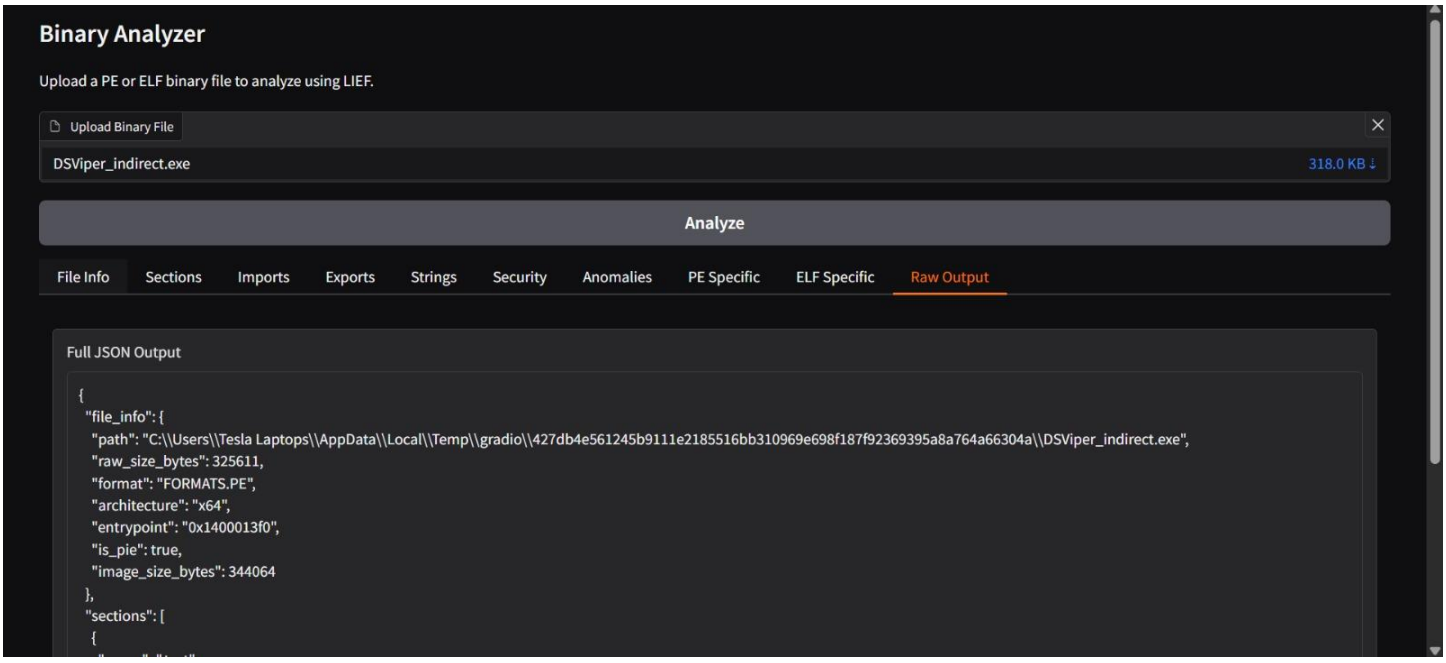
This is the screenshot of Anomalies Utils:

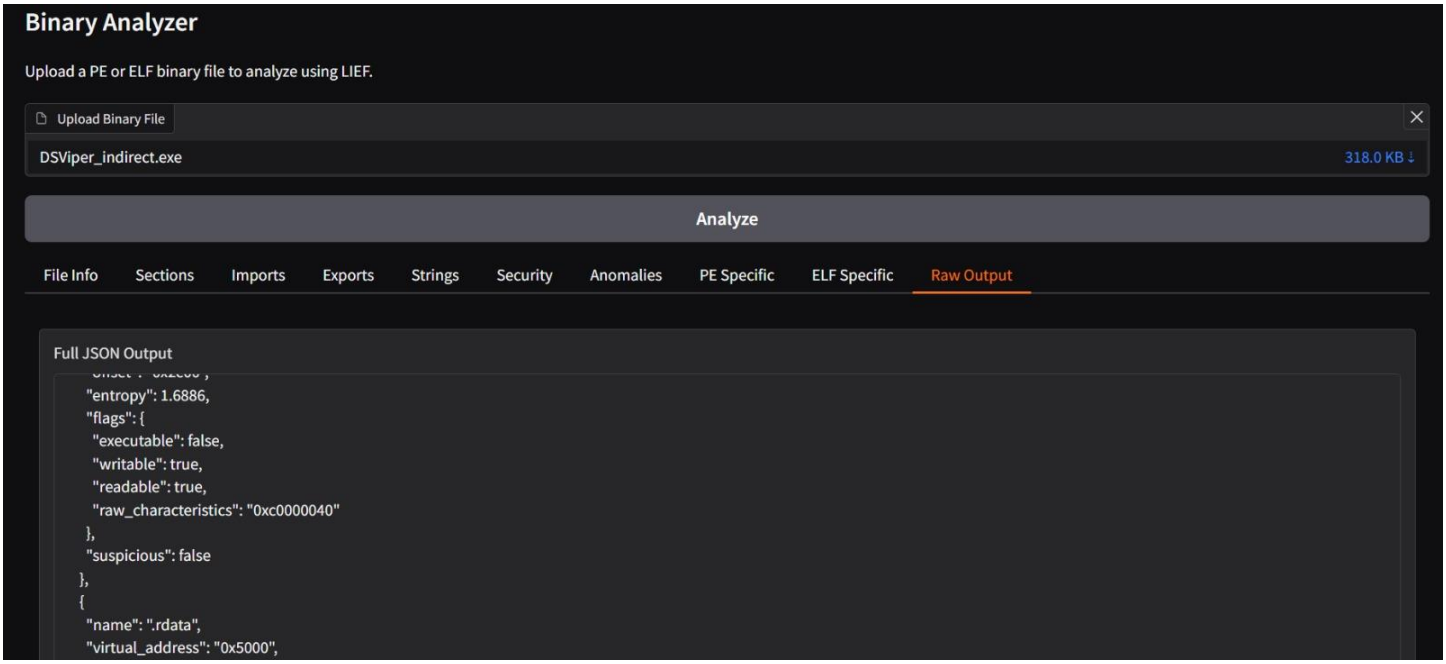


This is the screenshot of PE Specific Utils:



This is the screenshot of Raw Output Utils:




Binary Analyzer
Upload a PE or ELF binary file to analyze using LIEF.
Upload Binary File
DSViper_indirect.exe
318.0 KB
Analyze
File Info
Sections
Imports
Exports
Strings
Security
Anomalies
PE Specific
ELF Specific
Raw Output
Full JSON Output
"entropy": 1.6886,
"flags": {
"executable": false,
"writable": true,
"readable": true,
"raw_characteristics": "0xc0000040"
},
"suspicious": false
},
{
"name": ".rdata",
"virtual_address": "0x5000",