SPL-1 Project Report, [2022]

Static Analysis of Malware

SE 305: Software Project Lab-1

Submitted by

Md. Rakibul Islam

BSSE Roll No: 1411

BSSE Session: 2021-2022

Supervised by

Dr. B M Mainul Hossain

Designation: Director and Professor

Institute of Information Technology

Supervisors Approval: _____

Institute of Information Technology
University of Dhaka

[17-12-2023]

Contents

1. Introduction

The "Static Analysis of Malware" project offers an automated solution for detecting malware in executable files. It targets a range of harmful software, including viruses, worms, Trojans, and ransomware. The system uses MD5 hashing to create unique file identifiers, aiding in recognizing known malware. Additionally, it employs a trie-based string matching technique for efficient pattern recognition. PE header parsing is another critical method used to examine executables for unusual characteristics indicative of malware. This combination of MD5 hashing, string matching, and header analysis enables a comprehensive examination of files. By analyzing file behavior and traits, the system identifies and mitigates potential threats. Overall, this project enhances cybersecurity by proactively detecting and addressing malware risks.

1.1 Background Study

Malware poses a significant threat to computer systems and networks. With the increasing sophistication and variety of malware, traditional signature-based detection methods alone are no longer sufficient. Advanced techniques and algorithms are required to accurately detect and mitigate malware infections. This project aims to address this need by implementing an efficient and effective malware detection system.

For doing this project I had to implement a hashing algorithm named MD5, a data structure for matching the malware's hash value with the known malware hash values. PE header parsing of the exe file also had to be done. And an encoding of an exe file.

The main 4 parts of the project are-

➢ **Message Digest 5 (MD5):** Message Digest 5 is a cryptographic hash function that produces a fixed-size output(128-bit hash value) from any input data. It was developed by Rivet Rivest in 1991.

So what exactly is MD5?

MD5 operates on message blocks of 512 bits, and applies a series of logical operations and modular arithmetic operations to each block to generate a 128-bit hash value.

The complete process of evaluating hash value can be divided into 4 segments-

1. Padding: The input message is padded with a 1-bit followed by 0-bits to make its length a multiple of 512 bits. The padding is done in such a way that the length of the padded message is 64 bits less than the next multiple of 512 bits.

2. Initialization: The MD5 hash function initializes a 128-bit state variable, which is used to process each 512-bit block of the padded message. The state variable consists of four 32-bit words, A, B, C, and D.

3. Message Processing: The padded message is divided into 512-bit blocks, and each block is processed using four rounds of operations. Each round consists of 16 operations that operate on a 32-bit word of the block and use the result of the previous operation to compute the next operation.

The operations in each round are as follows:
Round 1: The first round mixes the input data in a simple way using modular addition, bitwise logical operations, and left rotations. The output of each operation is added to the corresponding word of the state variable.
Round 2: The second round uses a more complex mixing function that involves modular addition, bitwise logical operations, and nonlinear functions of the form (B AND C)

4. Finalization: After processing all the blocks of the padded message, the final hash value is computed by concatenating the four words of the state variable, A, B, C, and D, in little-endian order. The resulting 128-bit value is the MD5 hash of the input message.

5. Output: After processing all message blocks, the final state value is transformed into a 128-bit hash value, which is the output of the MD5 hash function.

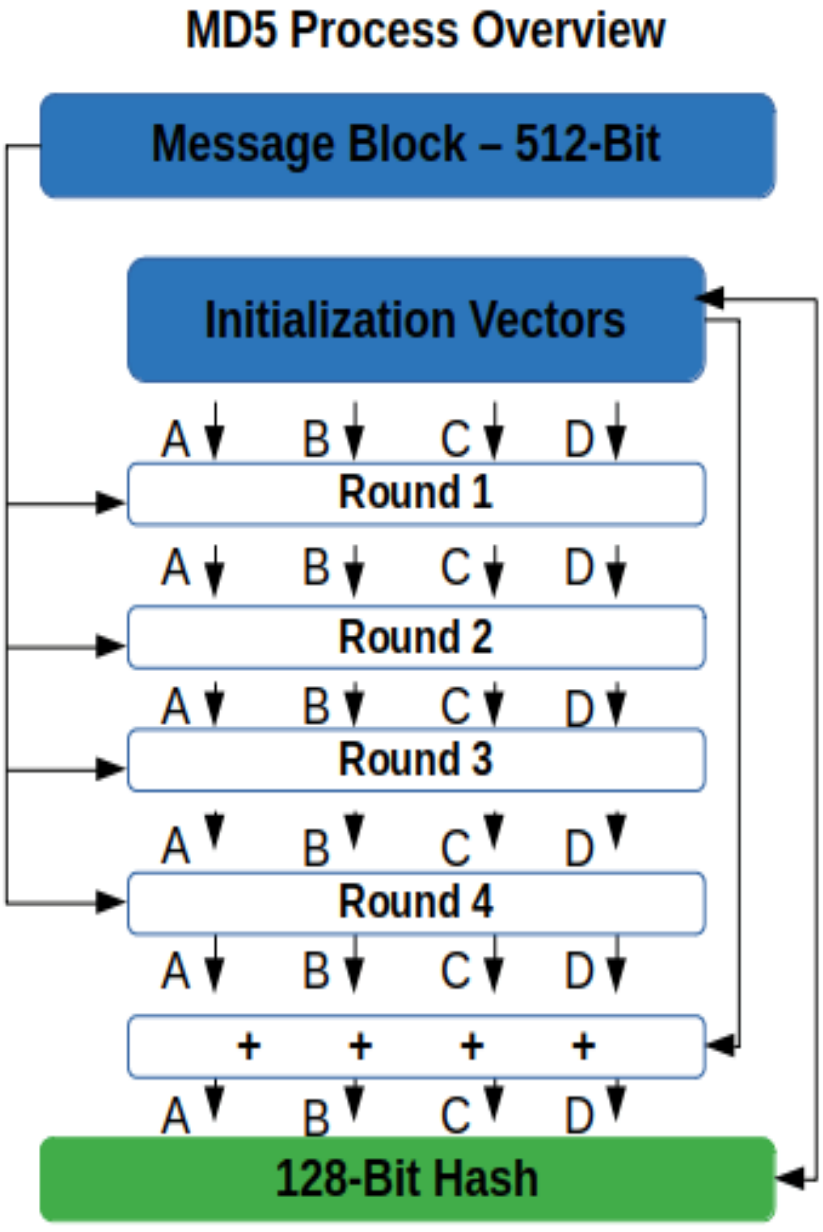

Figure-01: MD5 Hash Algorithm

➢ **Trie data structure for string matching:** The trie data structure, essential in the string matching process for malware detection, operates as a multiway tree designed to store strings across a specific alphabet. Its primary function is to accommodate a vast array of strings, making it ideal for handling extensive databases of known malware hash values. In this context, the trie is first populated with a comprehensive collection of hash values representing known malware signatures. When analyzing a file, the system computes its hash value and searches this trie. If there's a match with any stored hash, the file is immediately flagged as "Malware". This efficient search mechanism, facilitated by the trie's unique structure, accelerates the detection process.

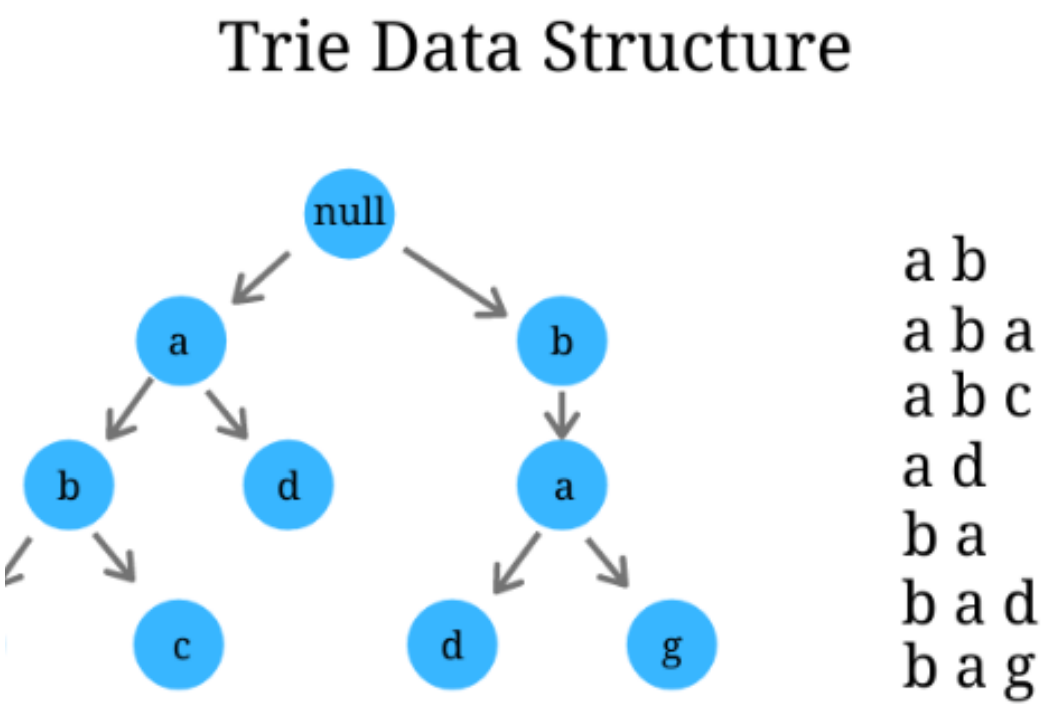

Figure-02: Trie data structure

➢ **PE header parsing:** PE header parsing is a crucial technique in analyzing the Portable Executable (PE) file format, used for executables, DLLs, and object codes in Windows operating systems. This format, encompassing both 32-bit and 64-bit versions, is a structured data format that contains critical information for the operating system's loader to handle the executable code. By parsing the PE header of an executable file, the system can extract its unique characteristics. This process involves comparing these characteristics against known malware profiles. If there's a match, it raises a flag, indicating a potential malware threat.

The format of a PE file is look like this-

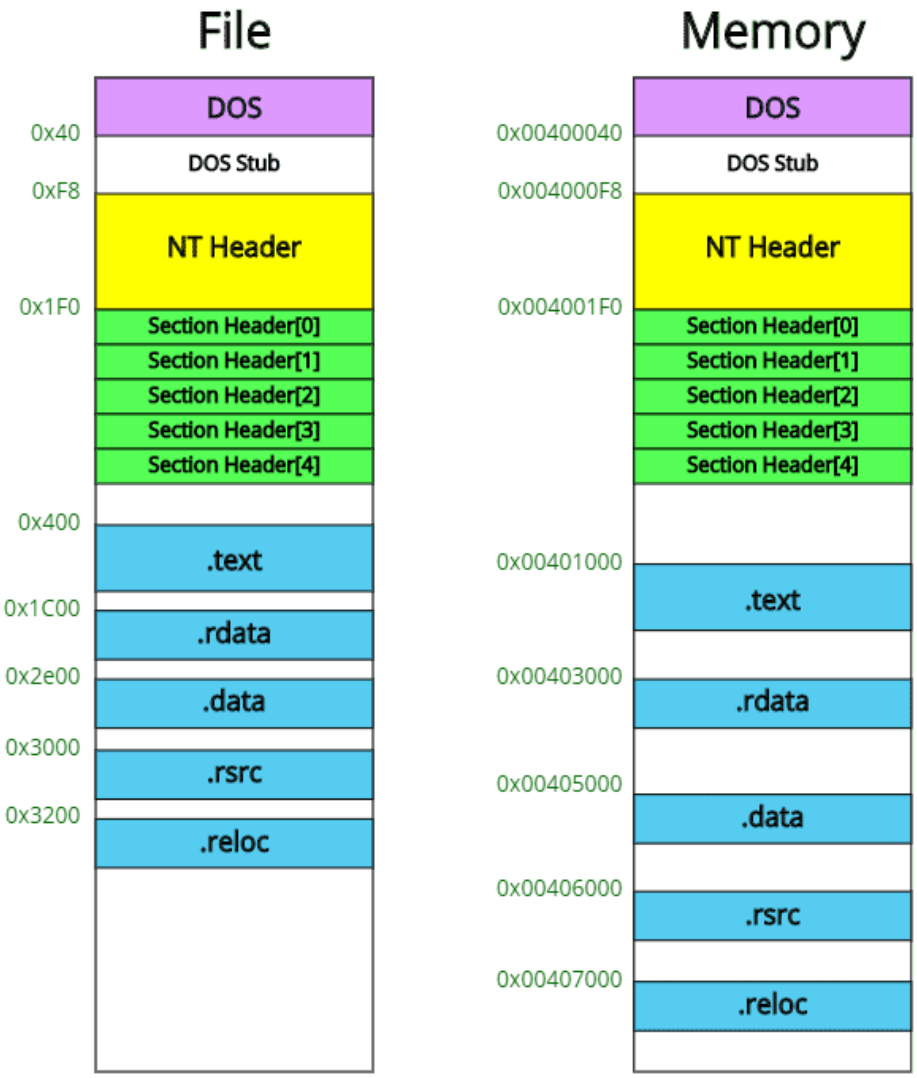

Figure-03: PE file format

➢ **Encoding of an exe file:** In this project, a straightforward bit shifting technique is employed for encoding malware within an executable file. The encoding process involves a 1-bit left shift operation, altering the original binary data of the file. Conversely, decoding is achieved through a 1-bit right shift operation, reverting the data back to its original form. This method of encoding and decoding is simple yet effective, subtly modifying the file to evade basic detection mechanisms. The bit shifting approach thus adds an additional layer of complexity to malware obfuscation and analysis.

1.2 Challenges

Developing a malware detecting system comes with several challenges that require careful consideration and effective solutions. Some of the key challenges in this domain include:

1. Evolving Malware Techniques: A major challenge is keeping pace with the constantly evolving tactics of malware authors. They use obfuscation, encryption, and polymorphism to evade detection, necessitating a dynamic and adaptable detection system. Static analysis, such as PE header parsing, is crucial for identifying new variants of malware.

2. Computational Intensity: Malware detection requires processing large amounts of data, which can be resource-intensive. Designing efficient algorithms and using optimized data structures is vital to manage computational demands without compromising performance.

3. User Interface Design: A user-friendly interface is essential for the effectiveness of a malware detection system. It should provide clear threat information, actionable options, and be intuitive to navigate, enabling users to respond effectively to malware threats.

4. Complexity in Hash Value Calculation: The process of converting input strings to MD5, involving decimal to binary and binary to hexadecimal conversions, is complex and time-consuming, especially when implemented in programming languages like C++.

5. Understanding the PE File Format: Gaining a comprehensive understanding of the PE file format, a key component for executables in Windows, was challenging. The format contains critical information for the OS loader, and fully grasping its structure is essential for effective malware analysis.

6. Difficulties in Parsing Section Headers: Parsing the section header of a PE file and evaluating section names was particularly difficult, as there are no readily available library functions to simplify this task. This required developing custom methods to parse and analyze PE headers.

7. Limited Access to Resources: Acquiring comprehensive and reliable information was a significant challenge. The scarcity of resources necessitated exploring alternative methods and sources to gather essential knowledge and technical guidance.

8. Personal Learning Curve: Having no prior knowledge of the project's specifics required extensive study and understanding of each part. Overcoming these challenges involved in-depth analysis, utilizing various resources, and seeking external help when necessary.

2. Project Overview

The architecture of the project is structured into several distinct but interconnected modules, each playing a crucial role in the system's overall functionality:

⇨ **User Menu Module**

⇨ **File/Folder Input Module**

⇨ **Hash Value Evaluation Module**

⇨ **Hash Matching Module**

⇨ **PE Header Parsing Module**

⇨ **User Interface Module**

⇨ **File Management Module**

Let's discuss the functionalities of each part one by one-

1. User Menu Module: Initially, the system presents a menu to the user, offering the option to test either an individual file or an entire folder for potential malware.

```
cout << "\nDo you want to test a File or a Folder for Malware Detection?" << endl;
cout << "If so then select an option:\n";
cout << "  1. A File\n";
cout << "  2. A Folder\n";
cout << "  3. Exit\n";

int userSelection;
cin >> userSelection;

// Clear the input buffer
cin.ignore( n: numeric_limits<streamsize>::max(), dlm: '\n');

if (userSelection == 1)
{
    cout << "Enter the File Path: ";
    string fileLocation;
    getline( &: cin, &: fileLocation);
    check( s: fileLocation);
}
```

Figure- 04: Code snippet for User Menu

2. File/Folder Input Module: The File/Folder Input Module is designed to facilitate user selection and processing of either individual files or entire folders for malware analysis. This module streamlines the input of data, ensuring smooth integration into the system for subsequent examination. It serves as the initial step in the malware detection process, handling the essential task of data acquisition.

```
if (userSelection == 1)
{
    cout << "Enter the File Path: ";
    string fileLocation;
    getline( &: cin,  &: fileLocation);
    check( s: fileLocation);
}

else if (userSelection == 2)
{
    cout << "Enter the Folder Path: ";
    string directoryPath;
    getline( &: cin,  &: directoryPath);
```

Figure- 05: Code snippet for Input File/Folder

3. Hash Value Evaluation Module: The system computes the MD5 hash value of the provided executable file and then cross-references this hash with a database of known malware hashes, which are sourced from the 'MalwareBazaar' and GitHub website.

```
string appendTo512Bits(const string& msg)
{
    string result = msg;
    int string_size = (int)msg.length();
    result += '1';
    int fullSize;
    if ((string_size % 512) < 448)
    {
        fullSize = 512 * (string_size / 512) + 448 - string_size - 1;
    }

    else
    {
        fullSize = 512 * (string_size / 512) + 511 - string_size + 448;
    }

    for (int i = 0; i < fullSize; i++)
    {
        result += '0';
    }

    string string_size_binary = numberToBinary( number: string_size,  size: -1);

    string result_length;
    for (int i = 0; i < (64 - string_size_binary.size()); i++)
    {
        result_length += '0';
    }

    result_length += string_size_binary;
    result_length = littleEndian64Bits( str: result_length);
    return result + result_length;
```

Figure- 06: Code snippet for Hash Value Evaluation

4. Hash Matching Module: The system employs a trie-based string matching algorithm for efficient hash comparison. It involves two primary operations: Insert and Search. Initially, all known malware hash values, sourced from a file, are inserted into the trie. Following this, the system performs a search operation within this trie. If the hash value of the file being analyzed matches any of the known malware hashes in the trie, a specific output is generated to indicate the detection.

```
bool search(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen( s: key);
    int index;
    struct TrieNode *pCrawl = root;

    for (level = 0; level < length; level++)
    {
        index = CHAR_TO_INDEX( c: key[level]);

        if (!pCrawl->children[index])
        {
            return false;
        }

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl->isEndOfWord);
}
```

Figure- 07: Code snippet for Hash Value Matching

5. PE Header Parsing Module: When an input file does not initially trigger malware detection, the system advances to a secondary analysis known as "PE Header Parsing." This process involves delving deeper into the file's structure to extract crucial information, including:

i) Number of Initialized Data

ii) Major Image Version

iii) DLL Characteristics

iv) Checksum

v) Section Names

The system conducts this analysis by parsing the Image DOS Header, Image File Header, Image Optional Header, and the Section Header. If the values extracted during this process align with known characteristics of malware, the file is then classified as potentially malicious. A specialized decision algorithm integrates all of this evaluated information to conclusively determine if the file is indeed malware.

```
unsigned long ParseDOSHeader(const char* _NAME, FILE* _Ppefile)
{
    const char* NAME;
    FILE* Ppefile;
    NAME = _NAME;
    Ppefile = _Ppefile;
    fseek(Ppefile, 0, SEEK_SET);
    fread( ptr: &PEFILE_DOS_HEADER,  size: sizeof(IMAGE_DOS_HEADER),  nitems: 1,  stream: Ppefile);

    PEFILE_DOS_HEADER_EMAGIC = PEFILE_DOS_HEADER.e_magic;
    PEFILE_DOS_HEADER_LFANEW = PEFILE_DOS_HEADER.e_lfanew;


    return PEFILE_DOS_HEADER_EMAGIC;



}
```

Figure-08: Code snippet for PE Header Parsing

6. User Interface Module: The system features a user-friendly interface that clearly displays the results of the malware detection process. If a file is identified as malware, the interface presents the user with a decision menu. This menu offers options to either delete the file, effectively removing the potential threat from the system, or quarantine it, isolating the file to prevent any harm to the computer system. Additionally, the user has the choice to leave the file unchanged if they decide to retain it despite the identified risks. This aspect of the system ensures that users are fully informed and can take appropriate actions based on the detection results.

```
cout<<"\nDo you want to Delete/Quarantine the file or want to exit the program?"<<endl;
cout<<"Select an option please-"<<endl;
cout<<"  1.Delete"<<endl;
cout<<"  2.Quarantine"<<endl;
cout<<"  3.Exit"<<endl;

int i;
cin>>i;
const char* filePath=fileName;

if(i==1)
{

    deleteMaliciousFile(filePath);

}
```

Figure-09: Code snippet for File Management

7. File Management Module: The system incorporates functionalities for file deletion or quarantine, contingent upon the user's decision. If the user opts to quarantine a detected malware file, the system employs a bit shifting operation to modify the file, rendering it inactive. Subsequently, this neutralized file is relocated to a specifically designated folder named “Quarantined Files”.

```
encodeMaliciousFile(filePath);

int a;
cout<<"Do you have (Decode) the file again?"<<endl<<"Press 1 to Decode the File"<<endl;
cout<<"press 0 to exit"<<endl;
cin>>a;

if(a==1)
{
    decodeMaliciousFile(filePath);
}
```

Figure-10: Code snippet for File Management

Overall output shown here when the file is detected as a malware after PE header analysis-

```
Welcome to 'Static Analysis of Malware'

Do you want to test a File or a Folder for Malware Detection?
If so then select an option:
  1. A File
  2. A Folder
  3. Exit
1
Enter the File Path: malware1.exe
MD-5 Hash Value is: fd1ec4e0dd8213b4b7fc33259acea631
This File is Detected as Malware by PreDefined Hash Value Matching..!!!


Do you want to Delete/Isolate the file or want to exit the program?
Select an option please-
  1.Delete
  2.Isolate
  3.Exit
2
File Encoding Complete.
Do you have (Decode) the file again?
Press 1 to Decode the File
press 0 to exit
0

Do you want to test a File or a Folder for Malware Detection?
If so then select an option:
  1. A File
  2. A Folder
  3. Exit
```

Figure-11: Sample Output

3. User manual

This guide provides detailed steps for acquiring and setting up the "Static Analysis of Malware" software from GitHub. Follow these instructions carefully to ensure a successful installation:

Prerequisites:

1. C++ Compiler: Make sure a C++ compiler is installed on your system for compiling the source code.

2. Code Editor/IDE: Install a code editor or an Integrated Development Environment (IDE) to open and modify the “MalwareDetection.cpp” file.

Installation Instructions:

1. Navigate to the SPL-01-3rd-Semester- project's GitHub repository at https://github.com/Rakibul1411/SPL-01-3rd-Semester- .

2. Click the “Code” button on the repository page and choose “Download Zip” to download the project files as a zip archive.

3. Once downloaded, extract the zip file to a location of your choice on your computer.

4. Using your C++ compiler, compile the “MalwareDetection.cpp” file. This step may vary depending on your specific compiler and development environment.

5. After compiling, run the executable file generated from the compilation.

6. Upon running the program, you will be presented with options to start the malware detection process. Select the desired option to proceed.

4. Conclusion

The "Static Analysis of Malware" stands out as a robust and automated tool designed to combat malware threats in executable files. It harnesses the power of MD5 hashing, trie-based string matching algorithms, and in-depth PE header parsing to proficiently identify and categorize files that may contain malware. Enhanced by its intuitive user interface, the system empowers users to make knowledgeable decisions regarding the management of detected malware, thereby playing a pivotal role in safeguarding the security and integrity of their computing environments.

References

1. A survey on malware and malware detecting systems, Imtithal A.Saeed, Ali selamat, Ali M. A. Abu Ayoub, International Journal of Computer Applications(0975- 8887), April-2013

2. PE-Header-Based Malware Study and Detection, Yibin Liao ,Department of Computer Science The University of Georgia, Athens, GA 30605.

[3]https://www.comparitech.com/blog/information-security/md5-algorithm-with-examples/
update on September 25, 2023

4. MalwareBazaar

5. https://www.virustotal.com/gui/home/upload

6. Predefined MD5 Malicious Hash value taken- https://github.com/ocatak/malware_api_class

7. https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/pe-file-header-parser-in-c++

8. https://www.hybrid-analysis.com/

9. https://0xrick.github.io/win-internals/pe8/#a-dive-into-the-pe-file-format---lab-1-writing-a-pe-parser

10. https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/pe-file-header-parser-in-c++

11. https://tryhackme.com/room/dissectingpeheaders

12. https://www.geeksforgeeks.org/trie-insert-and-search/