MALWARE ANALYSIS REPORT

Report Details

- **Report ID:** [Unique Report Identifier]
- **Date:** [Date of analysis]
- **Analyst:** [Name of the analyst]
- **Sample Name:** [Name or identifier of the malware sample]

INTRODUCTION

- **Background:** How the malware was discovered or why it's being analyzed.
- **Purpose of Analysis:** Detailed description of the goals of this analysis.
- **Tools Used:** List of tools used during the analysis process.

STATIC ANALYSIS

1. General Information

- **File Type:** [Executable, script, document, etc.]
- **File Size:** [Exact size in bytes]
- **Compilation Timestamp:** [Indication of when the malware was possibly compiled]

2. Hashes

- **MD5:**
- **SHA1:**
- **SHA256:**

3. Packers and Obfuscation

- Identified packers or obfuscation methods.

4. Strings Analysis

- Notable embedded strings.
- Potential configuration data, hardcoded paths, or commands.
- Network indicators: URLs, IP addresses.
- String analyser

5. Imports/Exports Analysis

- List of imported libraries and key functions.

- Notable or suspicious API calls.

6. Resource Analysis

- Embedded files or additional code.
- Metadata, version information, or manifest details.

7. Code Analysis

- Disassembled or decompiled code analysis.
- Identification of key routines, algorithms, or control flow.

File types:

The binary header of a file, often simply called the "file header," is a specific sequence of bytes at the beginning of the file that indicates its format or type. This sequence is also commonly known as the "magic number." The purpose of a file header is to provide a consistent and recognizable way for software (like operating systems or applications) to determine the type and format of a file without having to understand or interpret its entire contents.

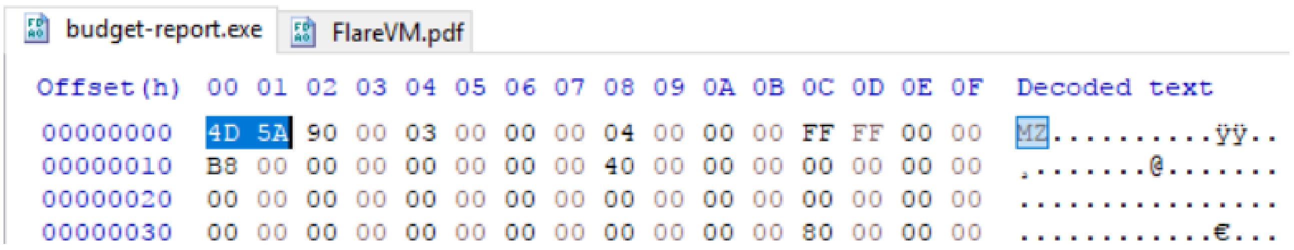
Here's a deeper look:

1. **Identification**: Headers allow quick identification of the file type. When a program opens a file, it often checks the header to ensure it's dealing with the expected format.
2. **Metadata**: In addition to identifying the file type, headers often contain metadata about the file, such as version information, file size, creation date, and more.
3. **Standardization**: Having standard headers for file types means that software developers have a consistent way to design their applications to handle those files. This consistency is why, for example, JPEG images can be viewed across a wide range of devices and software applications.
4. **Protection & Integrity**: File headers can sometimes help detect corrupted files. If the header doesn't match the expected format, the file might be damaged or not be what it purports to be.

Tool: HxD -Hex editor

1. Portable Executable (PE)

- **Binary Header:** `4D 5A`



```
budget-report.exe   FlareVM.pdf

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............€...
```

- **Indication:** This signature indicates Windows executables and DLLs.

2. ELF (Executable and Linkable Format)

- **Binary Header:** `7F 45 4C 46`
- **Indication:** Common executable format on UNIX systems.

3. Java Class File

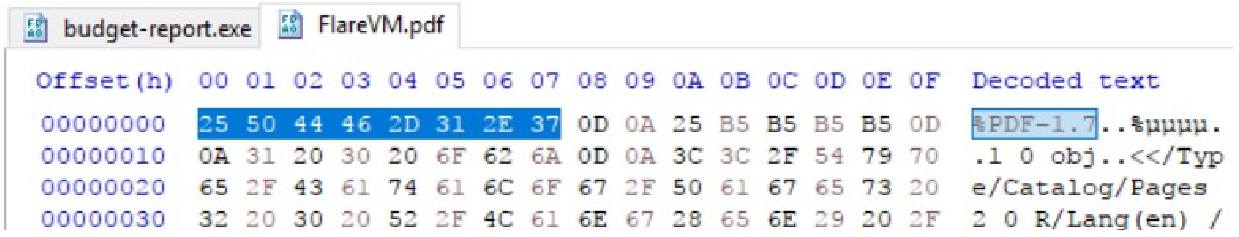
- **Binary Header:** `CA FE BA BE`
- **Indication:** Compiled Java bytecode.

4. Java JAR Archive

- **Binary Header:** `50 4B 03 04`
- **Indication:** Java Archive (compressed ZIP format).

5. PDF

- **Binary Header:** `25 50 44 46`



```
budget-report.exe   FlareVM.pdf

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  25 50 44 46 2D 31 2E 37 0D 0A 25 B5 B5 B5 B5 0D  %PDF-1.7..%µµµµ.
00000010  0A 31 20 30 20 6F 62 6A 0D 0A 3C 3C 2F 54 79 70  .1 0 obj..<</Typ
00000020  65 2F 43 61 74 61 6C 6F 67 2F 50 61 67 65 73 20  e/Catalog/Pages 
00000030  32 20 30 20 52 2F 4C 61 6E 67 28 65 6E 29 20 2F  2 0 R/Lang(en) /
```

- **Indication:** Adobe's document format; can contain embedded scripts or malicious payloads.

6. Office Open XML (DOCX, XLSX, PPTX)

- **Binary Header:** `50 4B 03 04`
- **Indication:** Modern Microsoft Office formats; ZIP containers that can contain XML and other data.

7. Rich Text Format (RTF)

- **Binary Header:** `7B 5C 72 74 66`
- **Indication:** Document format that can be used to deliver exploits.

8. HTML and XML

- **Binary Header:** Varies, but common sequences include `3C 68 74 6D 6C>` for HTML and `3C 3F 78 6D 6C` for XML.
- **Indication:** Web content; can contain scripts.

9. Windows Script File (WSF)

- **Binary Header:** Typically starts with XML-like syntax such as `3C 6A 6F 62>`.
- **Indication:** Windows Scripting.

10. GIF

- **Binary Header:** `47 49 46 38`
- **Indication:** Graphics Interchange Format, sometimes used for steganography.

11. JPEG

- **Binary Header:** `FF D8 FF`
- **Indication:** Image format; can be used for steganography or to deliver exploits.

12. PNG

- **Binary Header:** `89 50 4E 47 0D 0A 1A 0A`
- **Indication:** Image format; potential for steganography.

13. ZIP Archive

- **Binary Header:** `50 4B 03 04`
- **Indication:** Compressed data; can contain malicious payloads.

14. RAR Archive

- **Binary Header:** `52 61 72 21`
- **Indication:** Compressed data.

15. Python Compiled File (PYC)

- **Binary Header:** Varies, but often `03 F3 0D 0A` for Python 3.x.
- **Indication:** Compiled Python code.

16. SQLite Database File

- **Binary Header:** `53 51 4C 69 74 65 20 66`
- **Indication:** SQLite database, sometimes used for configuration or data storage by malware.

17. Windows Shortcut (LNK)

- **Binary Header:** `4C 00 00 00`
- **Indication:** Windows shortcut file, sometimes used in malicious campaigns.

18. BASH Shell Script

- **Binary Header:** `23 21 2F 62 69 6E 2F 62 61 73 68` or similar.
- **Indication:** UNIX shell script, indicated by shebang.

19. Mach-O Executable

- **Binary Header:** `FE ED FA CE` or `CE FA ED FE`

- **Indication:** Executable format for macOS.

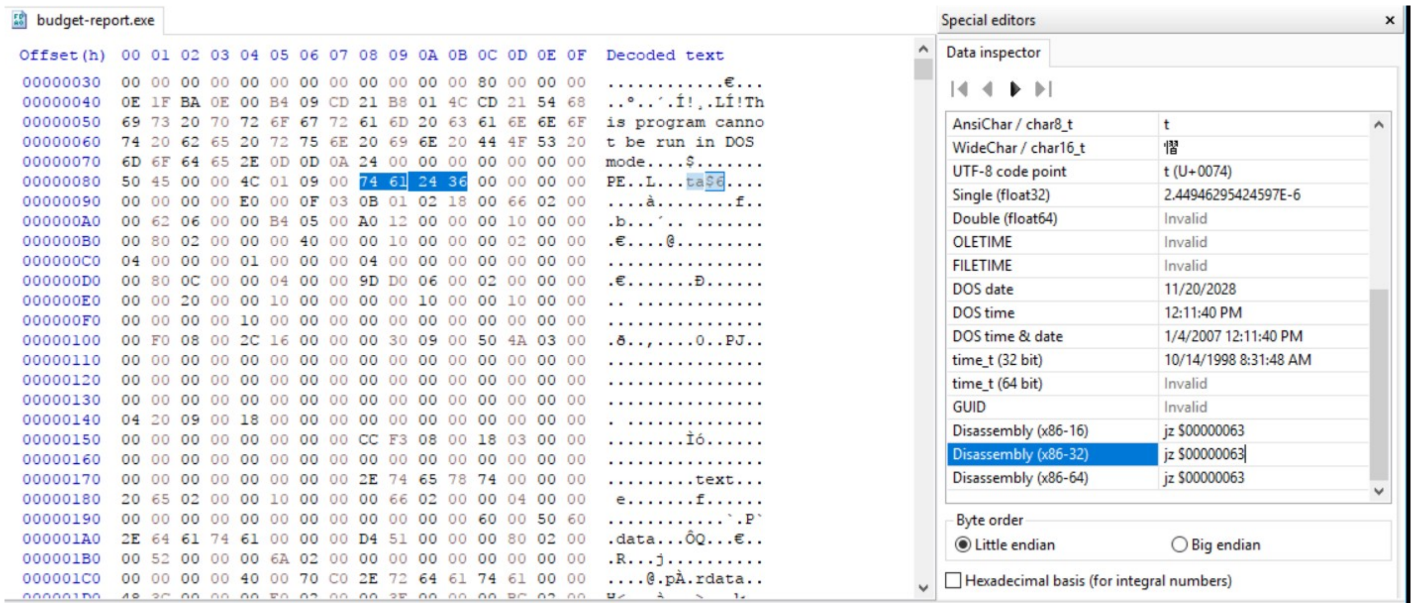
20. Flash (SWF)

- **Binary Header:** `46 57 53` or `43 57 53`
- **Indication:** Flash animation, sometimes used for exploits.

Compilation Timestamp

To check the Compilation Timestamp in an executable, specifically for Portable Executable (PE) files on Windows (e.g., `.exe`, `.dll`), you'll need to navigate to the PE header in the file's hexadecimal representation. Here's a step-by-step guide:

1. **Open the File in a Hex Editor**:
    - Use a hex editor like HxD, 010 Editor, or any other preferred tool.
2. **Locate the PE Signature**:
    - Most PE files start with an MS-DOS header, which is usually followed by the letters "PE" indicating the start of the PE header (in hexadecimal, this appears as `50 45`).
    - Just before the `PE` signature, there's a pointer (often offset `3C`) that points to the location of the PE header.
3. **Find the Timestamp**:
    - Directly after the PE signature (`50 45`), there's the COFF File Header.
    - The timestamp is an 8-byte value located 4 bytes into this COFF File Header. So, it's typically 4 bytes after the `PE` signature.
4. **Extract and Convert the Timestamp**:
    - Once you find the 8-byte (or 4-byte in some cases) timestamp, this is a UNIX-style timestamp, which counts seconds since the epoch (Jan 1, 1970). Convert this to a human-readable date/time. There are many online tools available for UNIX timestamp conversion.
5. **Interpret the Timestamp**:
    - Be cautious! Malware authors can easily manipulate this timestamp, so it might not be accurate. It can be backdated or set to arbitrary values.



budget-report.exe

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000030  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............€...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  50 45 00 00 4C 01 09 00 74 61 24 36 00 00 00 00  PE..L...ta$6....
00000090  00 00 00 00 E0 00 0F 03 0B 01 02 18 00 66 02 00  ....à........f..
000000A0  00 62 06 00 00 B4 05 00 A0 12 00 00 00 10 00 00  .b...´.. .......
000000B0  00 80 02 00 00 00 40 00 00 10 00 00 00 02 00 00  .€....@.........
000000C0  04 00 00 00 01 00 00 00 04 00 00 00 00 00 00 00  ................
000000D0  00 80 0C 00 00 04 00 00 9D D0 06 00 02 00 00 00  .€.......Ð......
000000E0  00 00 20 00 00 10 00 00 00 00 10 00 00 10 00 00  .. .............
000000F0  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00  ................
00000100  00 F0 08 00 2C 16 00 00 00 30 09 00 50 4A 03 00  .ð..,....0..PJ..
00000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000140  04 20 09 00 18 00 00 00 00 00 00 00 00 00 00 00  . ..............
00000150  00 00 00 00 00 00 00 00 CC F3 08 00 18 03 00 00  ........Ìó......
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
00000180  20 65 02 00 00 10 00 00 00 66 02 00 00 04 00 00   e.......f......
00000190  00 00 00 00 00 00 00 00 00 00 00 00 60 00 50 60  ............`.P`
000001A0  2E 64 61 74 61 00 00 00 D4 51 00 00 00 80 02 00  .data...ÔQ...€..
000001B0  00 52 00 00 00 6A 02 00 00 00 00 00 00 00 00 00  .R...j..........
000001C0  00 00 00 00 40 00 70 C0 2E 72 64 61 74 61 00 00  ....@.pÀ.rdata..
```

Special editors

Data inspector

AnsiChar / char8_t	t
WideChar / char16_t	慴
UTF-8 code point	t (U+0074)
Single (float32)	2.44946295424597E-6
Double (float64)	Invalid
OLETIME	Invalid
FILETIME	Invalid
DOS date	11/20/2028
DOS time	12:11:40 PM
DOS time & date	1/4/2007 12:11:40 PM
time_t (32 bit)	10/14/1998 8:31:48 AM
time_t (64 bit)	Invalid
GUID	Invalid
Disassembly (x86-16)	jz $00000063
Disassembly (x86-32)	jz $00000063
Disassembly (x86-64)	jz $00000063

Byte order: [✓ Little endian] Big endian

☐ Hexadecimal basis (for integral numbers)

The value `74 61 24 36` you've provided appears to be a little-endian hexadecimal representation. When interpreting data structures like the `TimeDateStamp` in the PE header, values are stored in little-endian order due to the x86 architecture's little-endian nature. This means we need to read it backward to understand it in typical hexadecimal representation.

If we reverse the bytes from `74 61 24 36` to `36 24 61 74`, we get the hexadecimal value `0x36246174`.

This value represents the number of seconds since January 1, 1970 (the Unix epoch). Converting `0x36246174` to its decimal equivalent gives us `907,244,916` seconds.

Using this value to calculate the date:

- Epoch timestamp: 907,244,916 corresponds to 12/10/1998 @ 7:48pm (UTC).

So, based on the timestamp you've provided, this file was possibly compiled on December 10, 1998, at 7:48 PM UTC.

A file hash

A file hash is a value generated from a string of text in a way that is nearly impossible to turn back into the original string. It's derived from the contents of a file and represents them as a fixed-size series of bytes. The process to create the hash value is deterministic, meaning the same input will always produce the same hash, but even a tiny change in the input will produce a very different-looking hash. Hashing is used in various applications such as password storage, data integrity verification, and digital signatures.

Some common properties of cryptographic hash functions:

1. **Fixed Size**: Regardless of the size of the input data, the hash value size remains constant.
2. **Fast Computation**: For any given input, the hash function should be capable of returning the hash value in a short amount of time.
3. **Pre-image Resistance**: Given a hash, it should be computationally difficult to retrieve the original input data.
4. **Small Changes, Big Impact**: Even a tiny change in input should produce such a drastic change in output that the new hash looks completely different.
5. **Collision Resistance**: It should be very hard to find two different inputs that produce the same hash.

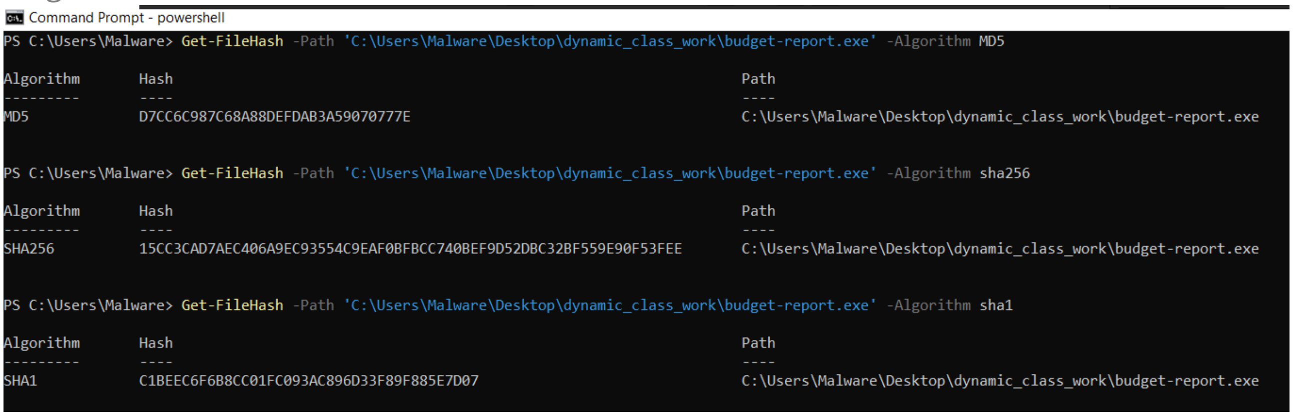
Common hashing algorithms include:

- **MD5**: Produces a 128-bit hash value, often represented as a 32-character hexadecimal number.
- **SHA-1**: Produces a 160-bit hash value, represented as a 40-digit hexadecimal number.
- **SHA-256**: Part of the SHA-2 family, it produces a 256-bit hash.
- **SHA-3**: The latest member of the Secure Hash Algorithm family.

Simple technique to create file hash – basic power shell command: **Get-HashFile**

```
Get-FileHash -Path 'path-to-file' -Algorithm <MD5, SHA1, SHA256>
```

Get-FileHash -Path 'C:\Users\Malware\Desktop\dynamic_class_work\budget-report.exe' -Algorithm sha1

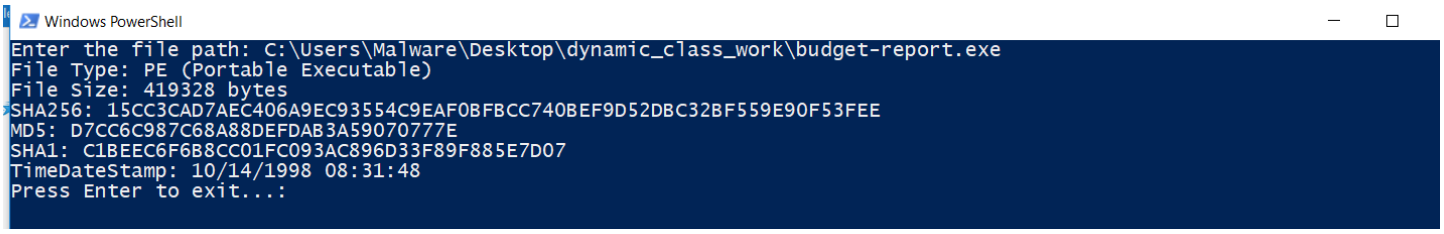


```
Command Prompt - powershell
PS C:\Users\Malware> Get-FileHash -Path 'C:\Users\Malware\Desktop\dynamic_class_work\budget-report.exe' -Algorithm MD5

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
MD5             D7CC6C987C68A88DEFDAB3A59070777E                                       C:\Users\Malware\Desktop\dynamic_class_work\budget-report.exe


PS C:\Users\Malware> Get-FileHash -Path 'C:\Users\Malware\Desktop\dynamic_class_work\budget-report.exe' -Algorithm sha256

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA256          15CC3CAD7AEC406A9EC93554C9EAF0BFBCC740BEF9D52DBC32BF559E90F53FEE       C:\Users\Malware\Desktop\dynamic_class_work\budget-report.exe


PS C:\Users\Malware> Get-FileHash -Path 'C:\Users\Malware\Desktop\dynamic_class_work\budget-report.exe' -Algorithm sha1

Algorithm       Hash                                                                   Path
---------       ----                                                                   ----
SHA1            C1BEEC6F6B8CC01FC093AC896D33F89F885E7D07                               C:\Users\Malware\Desktop\dynamic_class_work\budget-report.exe
```

You can use my PowerShell script as take path to suspicious file and output:

file type, file size, timestamp, hashes

https://github.com/anpa1200/Malware_analysis/blob/main/file_header_analyzer.ps1



```
Windows PowerShell
Enter the file path: C:\Users\Malware\Desktop\dynamic_class_work\budget-report.exe
File Type: PE (Portable Executable)
File Size: 419328 bytes
SHA256: 15CC3CAD7AEC406A9EC93554C9EAF0BFBCC740BEF9D52DBC32BF559E90F53FEE
MD5: D7CC6C987C68A88DEFDAB3A59070777E
SHA1: C1BEEC6F6B8CC01FC093AC896D33F89F885E7D07
TimeDateStamp: 10/14/1998 08:31:48
Press Enter to exit...:
```

Packers and Obfuscation

Packers and obfuscation are prevalent techniques used in the malware world to evade detection and make analysis more challenging. Understanding them is crucial for anyone diving into malware analysis.

• **What are Packers?** Packers, often termed as software protectors, are tools that can take an executable and transform it into a new binary, making the original code and data hidden or encrypted. When executed, this packed binary will typically decrypt or unpack itself in memory and then execute the original code. The purpose of packing is not only to reduce the size of the binary but also to make reverse engineering and signature-based detection more difficult.

Examples of common packers include UPX, PECompact, and ASPack.

• **What is Obfuscation?** Obfuscation refers to the practice of deliberately modifying the code to make it more confusing and challenging to read, especially when decompiled or disassembled. It doesn't change the logic or the outcome of the program but makes it cryptic. Techniques might include renaming variable and function names to meaningless labels, introducing dead code, or using confusing control flow constructs.

• **Identifying Packers and Obfuscation**: Malware analysts use a variety of methods to detect the use of packers or obfuscation techniques:

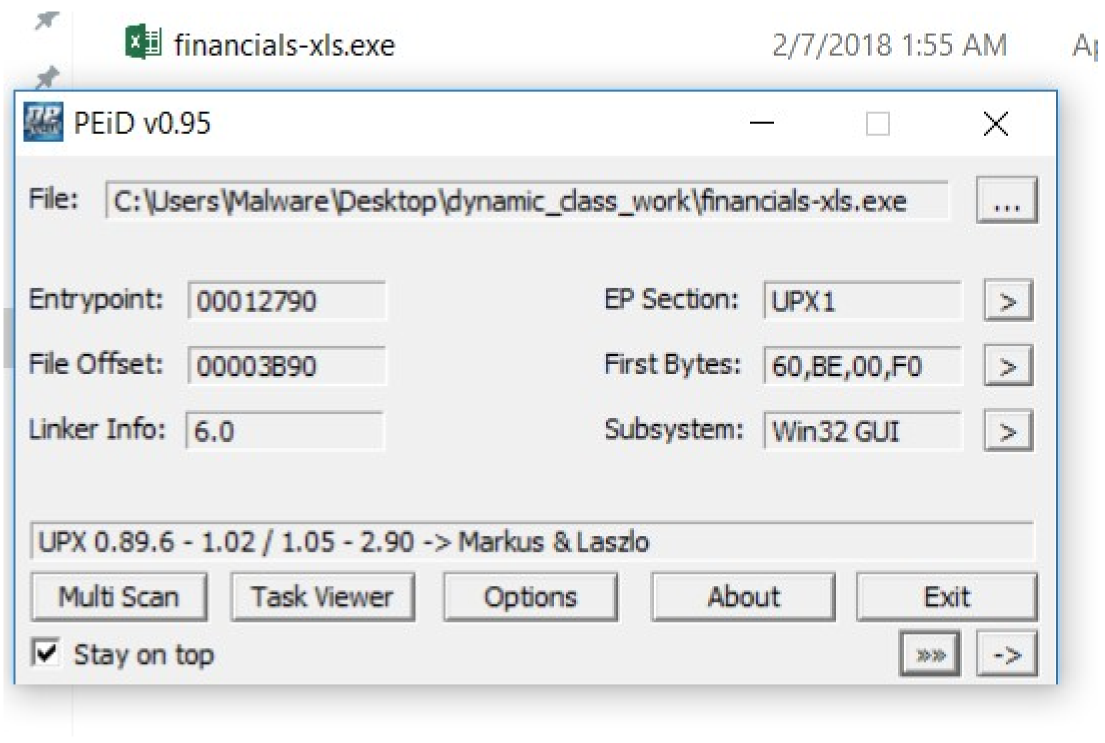
- **Entropy Analysis**: High entropy indicates that data is packed or encrypted. Tools like PEiD or Detect It Easy can give an entropy overview.
- **Signature-Based Detection**: Many tools can recognize signatures of known packers. PEiD, for instance, comes with a database of signatures for many common packers.
- **String Analysis**: As in the "String Analyser" tool, a lack of meaningful strings or very high obfuscated strings can be an indicator of packing or obfuscation.
- **Control Flow Analysis**: Obfuscated code might have a convoluted or illogical control flow, which can be spotted in tools like IDA Pro or Ghidra.

• **Why It Matters?**: Identifying packers and obfuscation is a crucial step in malware analysis. Once recognized, the analyst may decide to unpack a sample to study the original code, or employ dynamic analysis techniques to examine the malware's behavior post-decryption. Understanding the level of effort the malware authors have put into hiding their code can also give clues about the malware's origin, purpose, and potential sophistication.

Example:

I have file to analyze: financials.xls.exe.

* Use PeiD tool:


financials-xls.exe 2/7/2018 1:55 AM

PEiD v0.95

File: C:\Users\Malware\Desktop\dynamic_class_work\financials-xls.exe

Entrypoint: 00012790

EP Section: UPX1

File Offset: 00003B90

First Bytes: 60,BE,00,F0

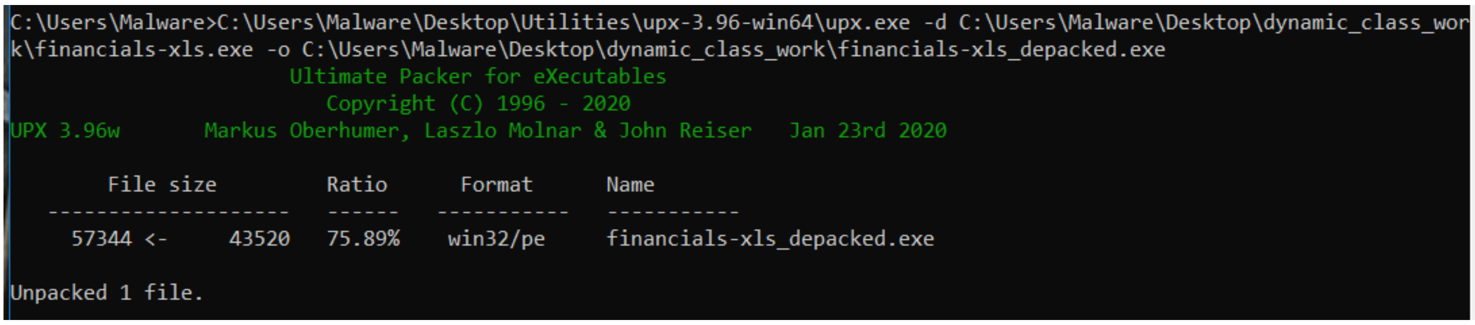
Linker Info: 6.0

Subsystem: Win32 GUI

UPX 0.89.6 - 1.02 / 1.05 - 2.90 -> Markus & Laszlo

* File was packed with UPX

* To unpack this file use standart upx tool.



```
C:\Users\Malware>C:\Users\Malware\Desktop\Utilities\upx-3.96-win64\upx.exe -d C:\Users\Malware\Desktop\dynamic_class_wor
k\financials-xls.exe -o C:\Users\Malware\Desktop\dynamic_class_work\financials-xls_depacked.exe
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2020
UPX 3.96w       Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     57344 <-     43520   75.89%    win32/pe     financials-xls_depacked.exe

Unpacked 1 file.
```

*Without unpacking you cant analyze the file correctly!

Unpack techniques:

1. UPX

To unpack a UPX-packed executable in Windows, follow these steps:

1. **Download UPX**:
    - First, you'll need to have UPX on your Windows machine. You can download it from the official website: https://upx.github.io/
2. **Install/Extract UPX**:
    - UPX is portable and doesn't require installation. Simply extract the ZIP file to a location of your choice. Remember the path, as you'll use the command prompt to navigate there.
3. **Unpack the Executable**:
    - Open the Command Prompt (you can search for `cmd` in the Windows search bar).
    - Navigate to the directory where you extracted UPX using the `cd` command. For instance, if you extracted UPX to `C:\Tools\UPX`, you'd type:

      bash

- `cd C:\Tools\UPX`
- To unpack an executable, use the following command:

```
upx -d path_to_packed_file.exe -o path_to_unpacked_output.exe
```

Replace `path_to_packed_file.exe` with the path to the packed executable you want to unpack and `path_to_unpacked_output.exe` with the desired path for the unpacked file. If you don't specify the `-o` option, UPX will overwrite the packed file.

For example:

mathematica

- `upx -d C:\MalwareSamples\packed_sample.exe -o C:\MalwareSamples\unpacked_sample.exe`

- If the unpacking is successful, UPX will display a message indicating the file has been unpacked.

Yo can use https://www.unpac.me/ for other types of packed files

Strings

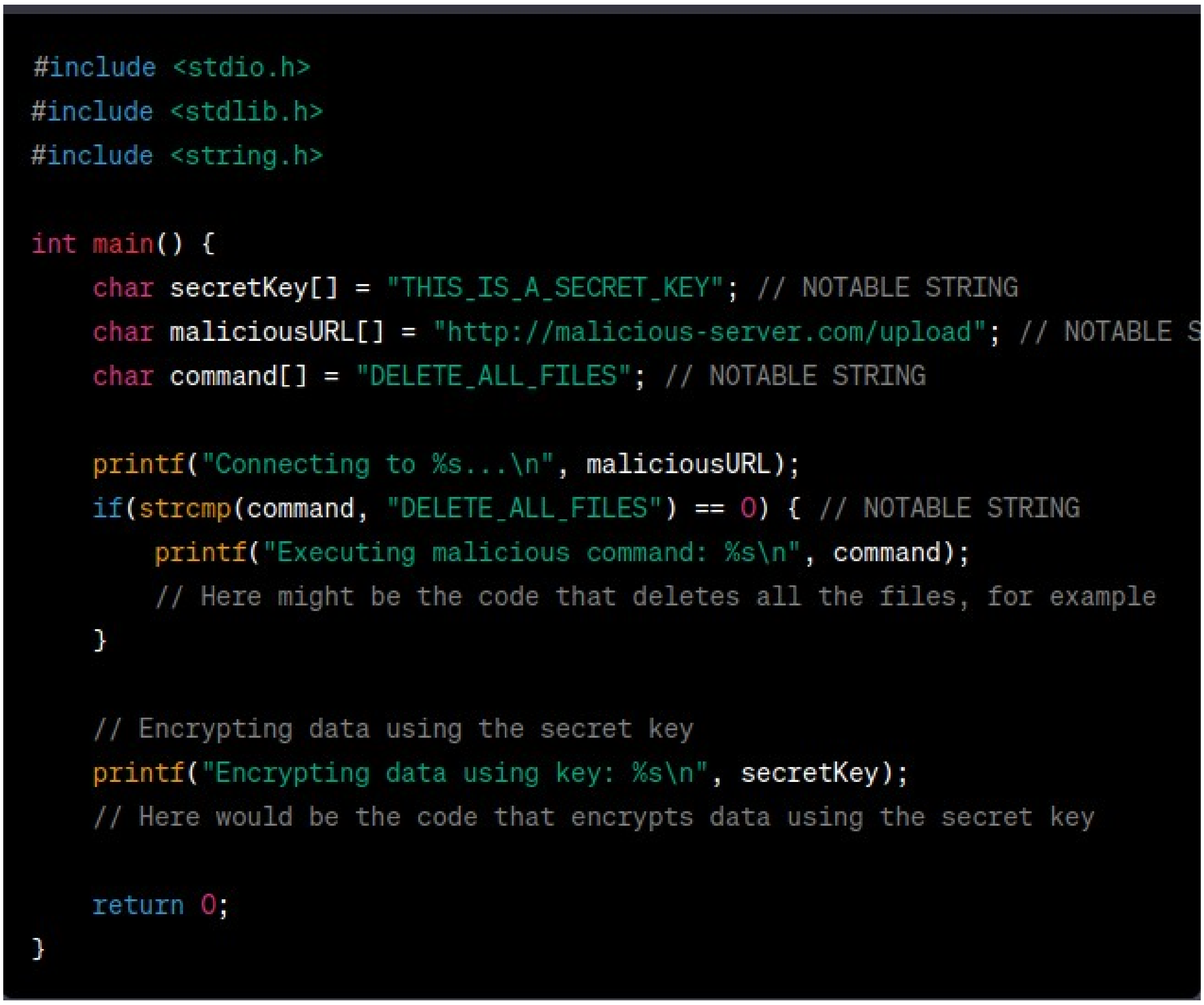
Notable Embedded Strings refers to specific sequences of readable characters (strings) found within a binary or program that provide meaningful or interesting information about its functionality, purpose, or origin.

In the context of malware analysis, "notable" typically means the strings have potential relevance for understanding the malware's behavior, identifying its authorship, or other aspects of its function and context. Here's a breakdown:

1. **What Are Strings?**
    - Strings are sequences of readable characters. In binary files or executables, they might be commands, messages, file paths, URLs, or other text that the program uses or references.
2. **Why Are They Notable?**
    - **Behavior Indicators:** Strings can give clues about what the malware might do. For example, a string like "cmd.exe /c" might indicate the malware intends to execute commands.
    - **Configuration or Command & Control (C2) Information:** Malware might contain strings that are URLs or IP addresses, indicating where it communicates for instructions or data exfiltration.
    - **Authorship or Attribution:** Sometimes, strings can give away who wrote the malware, especially if the author left in comments or used recognizable naming conventions.
    - **Decryption Keys or Algorithms:** Some malware contains strings that hint at how they decrypt other parts of their code or data.
    - **File Paths:** These can show where malware might try to write files or which files it tries to access.
    - **Error Messages:** These can give clues about the malware's function or what it expects its environment to be like.
3. **How Are They Found?**
    - Analysts often use tools to extract strings from a binary. A popular tool is named `strings`, part of the Unix and Unix-like operating systems. For Windows, Sysinternals also has a tool by the same name.
4. **Why Are They Important?**
    - Strings can provide a quick way to get insights into a binary without having to dive deep into code analysis. While they won't give a comprehensive view, they can quickly point an analyst in the right direction.
5. **Caveats:**

- Just because a string is present doesn't mean the malware uses it in an overt way. Some strings could be decoys or remnants from copied code.
- Advanced malware might encode, encrypt, or obfuscate strings to hide them from basic analysis.

Example: In this code, there are three "notable strings" that can provide hints about what the software does or its intentions when encountered during analysis: the first string is an encryption key, the second is a web address for an external server (which could be a Command and Control server), and the third is a command to delete files.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char secretKey[] = "THIS_IS_A_SECRET_KEY"; // NOTABLE STRING
    char maliciousURL[] = "http://malicious-server.com/upload"; // NOTABLE S
    char command[] = "DELETE_ALL_FILES"; // NOTABLE STRING

    printf("Connecting to %s...\n", maliciousURL);
    if(strcmp(command, "DELETE_ALL_FILES") == 0) { // NOTABLE STRING
        printf("Executing malicious command: %s\n", command);
        // Here might be the code that deletes all the files, for example
    }

    // Encrypting data using the secret key
    printf("Encrypting data using key: %s\n", secretKey);
    // Here would be the code that encrypts data using the secret key

    return 0;
}
```

The "**String Analyser"** (**https://github.com/anpa1200/Malware_analysis/blob/main/string_analyzer.exe**) is a open source specialized tool tailored for detecting and categorizing embedded strings within binary files or malware samples. By extracting and analyzing these strings, the tool provides insights into potential behaviors, functions, and communication endpoints of a piece of malware.

Key Features include:

1. **Windows API Detection**: Identify calls to known Windows APIs, which can indicate specific functionalities the malware might be leveraging.

2. **Command Recognition**: Highlight system commands that may indicate file operations, system manipulation, or other suspicious activities.
3. **Network Endpoint Extraction**: Extract potential URLs and IP addresses, which can point to command and control servers or other malicious endpoints.
4. **File Path and DLL Parsing**: Unearth embedded file paths and DLL names, giving clues to malware persistence or injection techniques.

By using **"String Analyser"**, you can quickly gain a preliminary understanding of a malware's capabilities and intentions, allowing for a more targeted and effective subsequent analysis.

Or you can use s**trings.exe** fro sysinternals.

Imports/Exports Analysis

An essential step in the process of analyzing a given executable (often in the context of malware analysis). It refers to studying the functions an executable imports from other dynamic-link libraries (DLLs) and the functions it exports (if any). This analysis can provide insights into the capabilities and intentions of a given piece of software.

Why is it important?

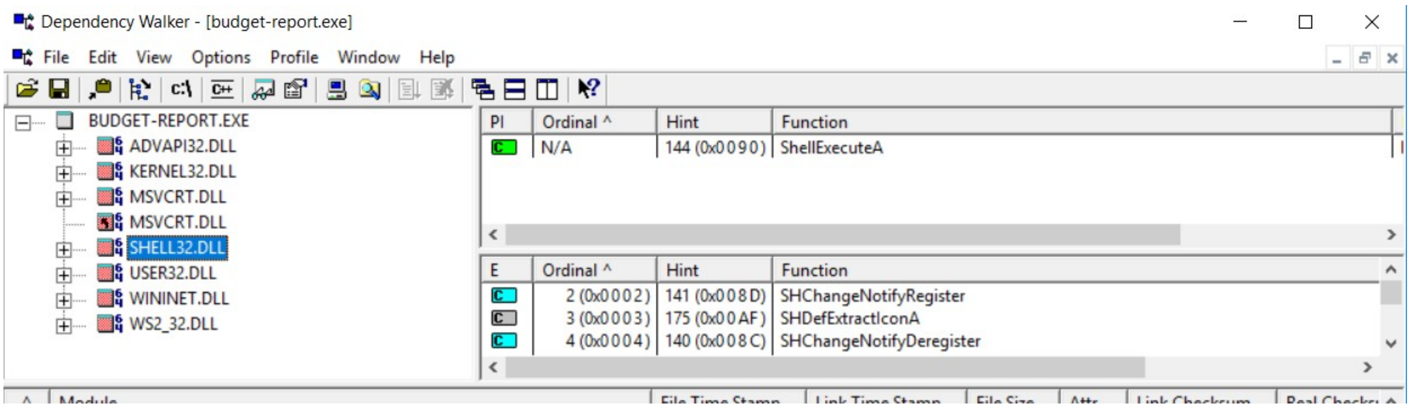
1. **Functionality Insight**: The functions an executable imports can often give clues about what the software might be designed to do. For instance, an executable that imports networking functions is likely involved in some form of network communication.

2. **Dependency Determination**: Understanding what libraries an executable relies upon can be essential for determining if it will run on a given system or under specific conditions.

3. **Detection and Classification**: Recognizing patterns in imports can assist in categorizing malware into families or recognizing known benign software.

4. **Red Flags**: Certain imports, such as functions to inject code into other processes, can be indicators of malicious intent.

Example with tool **Dependency walker**


Dependency Walker - [budget-report.exe]
File Edit View Options Profile Window Help

```
BUDGET-REPORT.EXE
  ADVAPI32.DLL
  KERNEL32.DLL
  MSVCRT.DLL
  MSVCRT.DLL
  SHELL32.DLL
  USER32.DLL
  WININET.DLL
  WS2_32.DLL
```

PI	Ordinal ^	Hint	Function
C	N/A	144 (0x0090)	ShellExecuteA

E	Ordinal ^	Hint	Function
C	2 (0x0002)	141 (0x008D)	SHChangeNotifyRegister
C	3 (0x0003)	175 (0x00AF)	SHDefExtractIconA
C	4 (0x0004)	140 (0x008C)	SHChangeNotifyDeregister

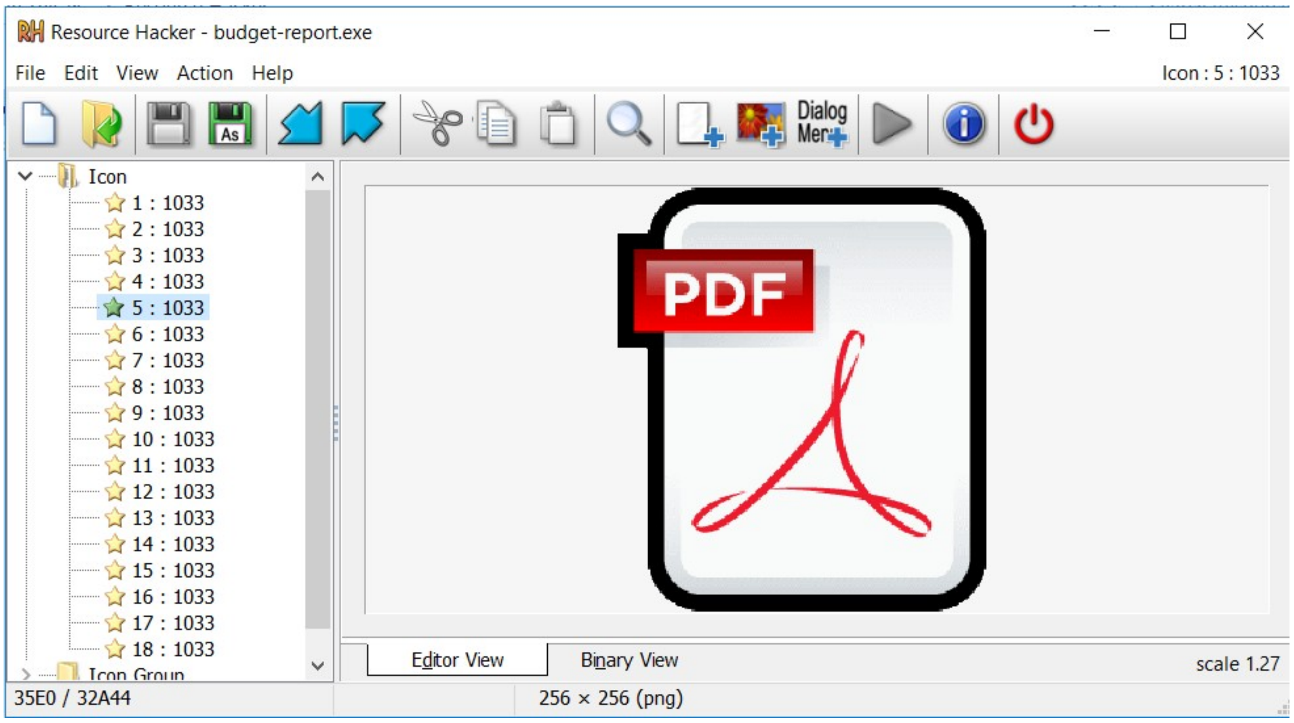
DLLs with common functions

`kernel32.dll`	Core DLL for most Windows operations. Functions like `CreateProcess`, `WriteFile`, and `VirtualAlloc` can be leveraged for process injection, writing malware payloads, etc.
`user32.dll`	Interacts with the Windows UI. Malware may use functions like `FindWindow` or `SendInput` for tasks such as keylogging or evading user detection.
`gdi32.dll`	Though graphical, malware could leverage it to create screenshots for espionage purposes.
`ntdll.dll`	Provides a lot of low-level system functionalities. Malware can use `NtQueryDirectoryFile` for file enumeration or `NtUnmapViewOfSection` for process hollowing.
`advapi32.dll`	Central to Windows security. Malware can use `CryptEncrypt` for ransomware purposes or `OpenSCManager` to interact with system services.
`ws2_32.dll`	Used for network communications. `socket` or `connect` functions can be crucial for malware's command & control (C&C) communications.
`comdlg32.dll`	Can be used by malware to present misleading dialog boxes to users, possibly as a social engineering tactic.
`ole32.dll`	OLE functionalities can be abused to launch macros or embedded malicious payloads in documents.
`shell32.dll`	`ShellExecute` could be used to run malicious scripts or commands.
`comctl32.dll`	Not as common in malware context but can be leveraged for UI spoofing attacks.
`msvcrt.dll`	While primarily for C runtime, certain string manipulation functions might be employed to modify or generate malware configurations.
`wininet.dll`	Provides Internet functionalities. Malware might use `InternetOpenUrl` or `HttpSendRequest` to fetch payloads or communicate with a C&C server.
`crypt32.dll`	Beyond standard encryption, malware uses it to validate certificates, potentially mimicking legitimate applications.
`shlwapi.dll`	Contains URL and path manipulation functions. Might be used by malware to parse and generate malicious URLs.
`oleaut32.dll`	If a malware employs automation to exploit applications like MS Office, they might interact with this DLL.
`imm32.dll`	While mainly for input methods, some keyloggers might tap into its functionalities.
`version.dll`	Malware might check versions of specific files to decide on the exploitation technique to employ.
`setupapi.dll`	Relevant for malware that interacts with device drivers or aims to persist via hardware and devices.
`rasapi32.dll`	Could be used by malware that wants to manipulate VPN or dial-up connections, potentially for traffic redirection.
`msimg32.dll`	Less common in a malware context but could be used in very specific graphical-based attack scenarios.

Resource analysis

Resource analysis, in the context of malware research and reverse engineering, refers to the examination of embedded resources within a binary or executable file. Many Windows applications (and malware) embed resources such as icons, images, audio clips, configuration files, or even other executables within the main binary. These resources can be a goldmine of information for an analyst and can sometimes contain malicious payloads or clues about an executable's functionality and origin.

Example from Resource Hacker tool



Resource Hacker - budget-report.exe
File Edit View Action Help
Icon : 5 : 1033
Dialog Mer
Icon
1 : 1033
2 : 1033
3 : 1033
4 : 1033
5 : 1033
6 : 1033
7 : 1033
8 : 1033
9 : 1033
10 : 1033
11 : 1033
12 : 1033
13 : 1033
14 : 1033
15 : 1033
16 : 1033
17 : 1033
18 : 1033
Icon Group
PDF
Editor View
Binary View
scale 1.27
35E0 / 32A44
256 × 256 (png)