Assignment 3

Software Re-Engineering

Reverse Engineering Using Open-Source Tools

***Submitted to*: Dr Shahela Saif**

***Submitted by*:**

Haadiya Sajid 21I-1216

Umar Farooq 21I-1143

Date of Submission:

17th November 2024

Section:

SE-P

Table of Contents

Introduction

This document presents an analysis of the 7-Zip software using Ghidra tool, focusing on its two main components: **7zFM.exe (File Manager)** and **7zG.exe (GUI Application)**, detailing their functionality and structure. Additionally, we analyzed a malware executable to compare patterns in assembly and decompiled code, highlighting key differences between well-designed software and malicious programs.

Project Objectives

1. **Static Analysis** of decompiled code obtained from 7zip's key exe files.
    a. Analyze positive indicators of good practices
    b. Find vulnerabilities and issues

2. **Studying Ghidra's decompiler working** and its ability to replicate the original code given the exe.
    a. Comparison with original source code

3. Comparison with **Malicious software**
    a. Analyze malicious patterns with Ghidra

Summary of Selected Software

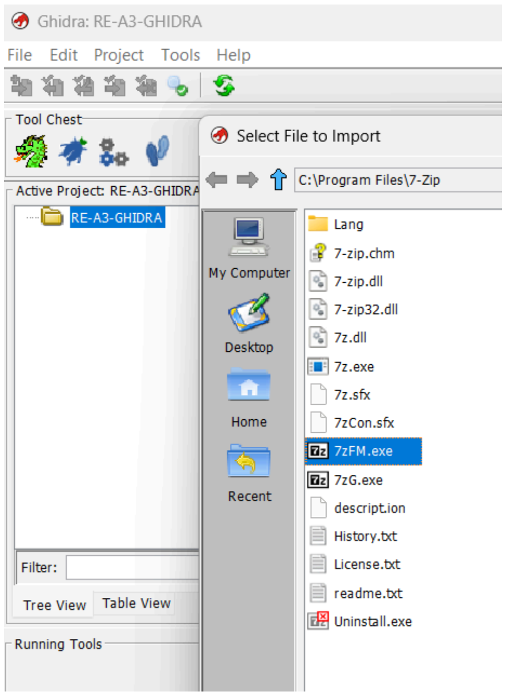
- 7-Zip is a free, open-source file compression and extraction software known for its high compression ratios using LZMA/LZMA2 algorithms.
- The original code of this open source project "7ZIP" consists of both C and CPP code which is compiled together to create the whole project.
- It supports a wide range of formats for both packing (e.g., 7z, ZIP, TAR) and unpacking (e.g., RAR, ISO, NTFS).
- It offers AES-256 encryption, a robust command-line interface, Windows shell integration, and localizations for 87 languages.

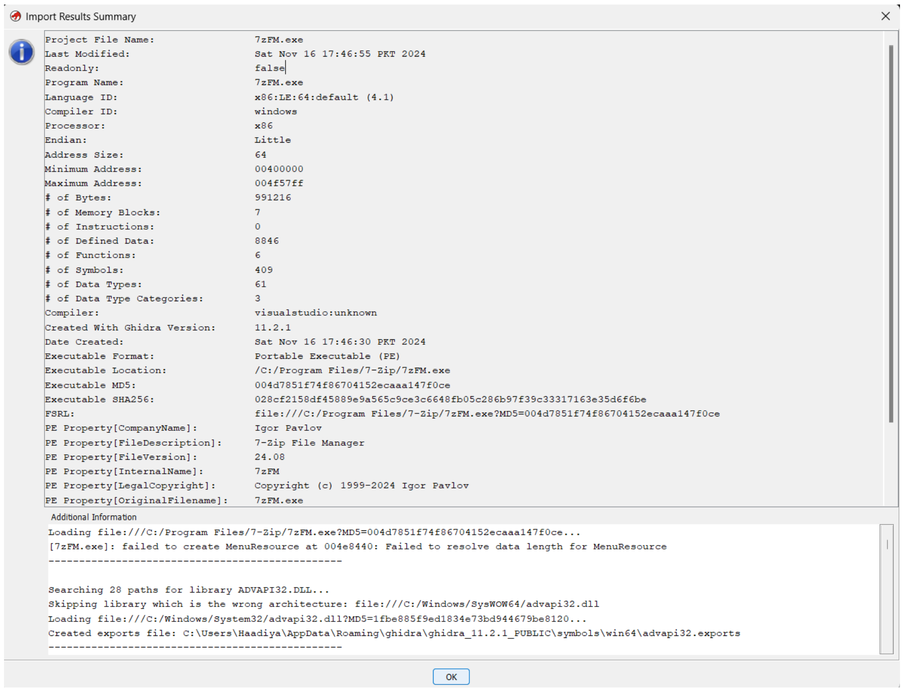
1. Static Analysis of 7zip

In this section we will perform static analysis of the decompiled 7zip code obtained through Ghidra. We focused on the decompiled code obtained from **7zFM.exe** (File Manager) and **7zG.exe** (Graphical User Interface) as they represent the primary user-facing components of 7-Zip.

These executables provide the graphical interface for interacting with files, such as browsing archives and executing compression/extraction operations. By analyzing these, we gain insights into the usability and design of the core functionality that most users interact with, while other components (like DLLs, uninstall.exe) serve lower-level or backend roles.

1.1. Summary of statistics





```
Project File Name:              7zFM.exe
Last Modified:                  Sat Nov 16 17:46:55 PKT 2024
Readonly:                       false
Program Name:                   7zFM.exe
Language ID:                    x86:LE:64:default (4.1)
Compiler ID:                    windows
Processor:                      x86
Endian:                         Little
Address Size:                   64
Minimum Address:                00400000
Maximum Address:                004f57ff
# of Bytes:                     991216
# of Memory Blocks:             7
# of Instructions:              0
# of Defined Data:              8846
# of Functions:                 6
# of Symbols:                   409
# of Data Types:                61
# of Data Type Categories:      3
Compiler:                       visualstudio:unknown
Created With Ghidra Version:    11.2.1
Date Created:                   Sat Nov 16 17:46:30 PKT 2024
Executable Format:              Portable Executable (PE)
Executable Location:            /C:/Program Files/7-Zip/7zFM.exe
Executable MD5:                 004d7851f74f86704152ecaaa147f0ce
Executable SHA256:              028cf2158df45889e9a565c9ce3c6648fb05c286b97f39c33317163e35d6f6be
FSRL:                           file:///C:/Program Files/7-Zip/7zFM.exe?MD5=004d7851f74f86704152ecaaa147f0ce
PE Property[CompanyName]:       Igor Pavlov
PE Property[FileDescription]:   7-Zip File Manager
PE Property[FileVersion]:       24.08
PE Property[InternalName]:      7zFM
PE Property[LegalCopyright]:    Copyright (c) 1999-2024 Igor Pavlov
PE Property[OriginalFilename]:  7zFM.exe
```

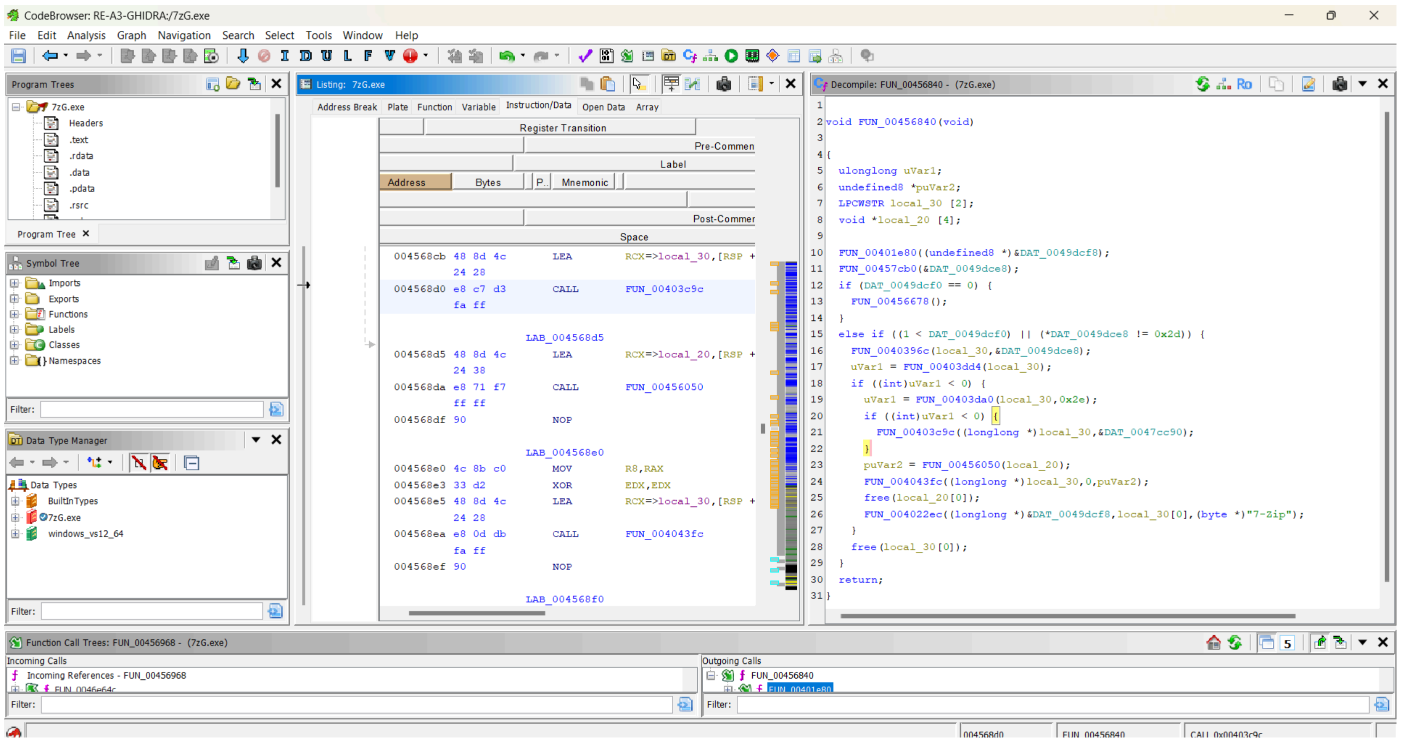
```
Loading file:///C:/Program Files/7-Zip/7zFM.exe?MD5=004d7851f74f86704152ecaaa147f0ce...
[7zFM.exe]: failed to create MenuResource at 004e8440: Failed to resolve data length for MenuResource
------------------------------------------------

Searching 28 paths for library ADVAPI32.DLL...
Skipping library which is the wrong architecture: file:///C:/Windows/SysWOW64/advapi32.dll
Loading file:///C:/Windows/System32/advapi32.dll?MD5=1fbe885f9ed1834e73bd944679be8120...
Created exports file: C:\Users\Haadiya\AppData\Roaming\ghidra\ghidra_11.2.1_PUBLIC\symbols\win64\advapi32.exports
------------------------------------------------
```

Here we can see a history of the results.

1. **Executable Information**:
   - **Program Name:** `7zFM.exe` (7-Zip File Manager).
   - **Executable Format:** Portable Executable (PE), common for Windows applications.
   - **File Version:** 24.00, indicating the version of 7-Zip File Manager.
   - **Language ID:** `x86:LE:64:default (4.1)`, meaning the binary is compiled for 64-bit Windows systems.
   - **Compiler ID:** Likely compiled using Visual Studio (exact version unknown).
2. **Memory and Instruction Details**:
   - **# of Functions:** 61, showing a moderate complexity level, allowing for meaningful reverse engineering.
   - **# of Symbols:** 409, indicating a mix of named functions, global variables, and debug information available.
3. **Metadata**:
   - **Copyright:** Igor Pavlov (1999–2024), confirming it is a legitimate, open-source project.
   - **Internal Name:** `7zFM`, consistent with the purpose of the executable.
4. **Potential Issues and Observations**:
   - **MenuResource Issue:** Failed to resolve the data length for MenuResource. This might be related to incomplete or missing resource tables.
   - **Library Search:** The ADVAPI32.DLL library (Windows API) was found, but it seems to have some discrepancies with symbols.
   - **Export Table Creation:** Ghidra has successfully created an exports file, aiding further static analysis.
5. **Architecture and Endianness**:

- **Processor Architecture:** `x86`, confirming it targets Intel-compatible processors.
- **Endian:** Little-endian, standard for x86 architecture.

1.2. Code Extraction



```
004568cb 48 8d 4c        LEA      RCX=>local_30,[RSP +
         24 28
004568d0 e8 c7 d3        CALL     FUN_00403c9c
         fa ff

                     LAB_004568d5
004568d5 48 8d 4c        LEA      RCX=>local_20,[RSP +
         24 38
004568da e8 71 f7        CALL     FUN_00456050
         ff ff
004568df 90              NOP

                     LAB_004568e0
004568e0 4c 8b c0        MOV      R8,RAX
004568e3 33 d2           XOR      EDX,EDX
004568e5 48 8d 4c        LEA      RCX=>local_30,[RSP +
         24 28
004568ea e8 0d db        CALL     FUN_004043fc
         fa ff
004568ef 90              NOP

                     LAB_004568f0
```

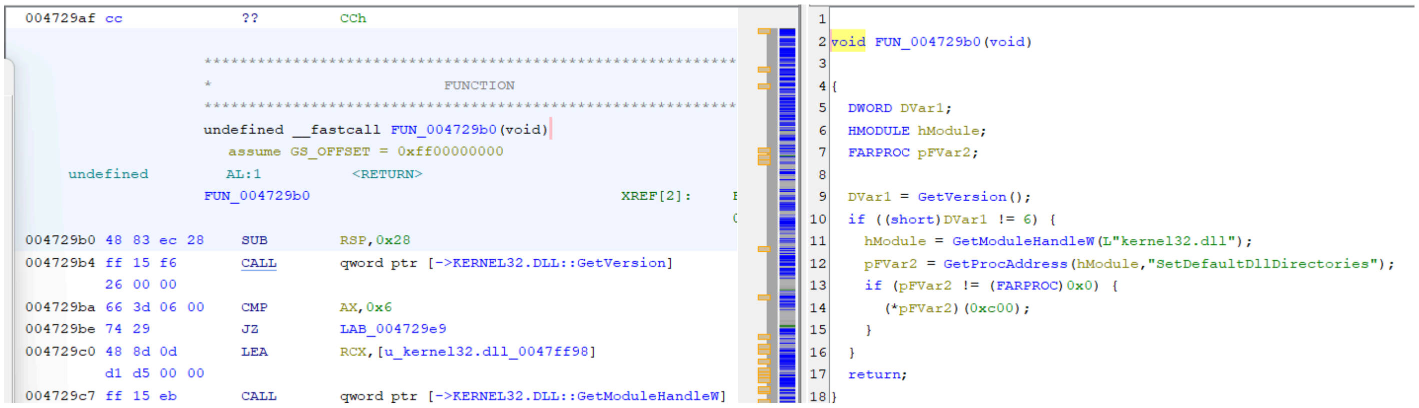
```c

void FUN_00456840(void)

{
  ulonglong uVar1;
  undefined8 *puVar2;
  LPCWSTR local_30 [2];
  void *local_20 [4];

  FUN_00401e80((undefined8 *)&DAT_0049dcf8);
  FUN_00457cb0(&DAT_0049dce8);
  if (DAT_0049dcf0 == 0) {
    FUN_00456678();
  }
  else if ((1 < DAT_0049dcf0) || (*DAT_0049dce8 != 0x2d)) {
    FUN_0040396c(local_30,&DAT_0049dce8);
    uVar1 = FUN_00403dd4(local_30);
    if ((int)uVar1 < 0) {
      uVar1 = FUN_00403da0(local_30,0x2e);
      if ((int)uVar1 < 0) {
        FUN_00403c9c((longlong *)local_30,&DAT_0047cc90);
      }
      puVar2 = FUN_00456050(local_20);
      FUN_004043fc((longlong *)local_30,0,puVar2);
      free(local_20[0]);
      FUN_004022ec((longlong *)&DAT_0049dcf8,local_30[0],(byte *)"7-Zip");
    }
    free(local_30[0]);
  }
  return;
}
```

Ghidra is able to extract assembly level code as well as C-like high level code for the application. In this analysis, in order to limit the scope we analyzed the decompiled code of both 7zG.exe, the GUI component of 7-Zip, and 7zFM.exe which is the file manager. We were focusing on security, architecture, and code quality aspects.

1.3. Positive Indicators in Code

1.3.1. *Secure DLL Loading*

- Implements `SetDefaultDllDirectories` with flag 0xc00
- Common vulnerability: DLL hijacking
- This is a combined flag (`0x800 + 0x400 = 0xC00`)
    - represents **specific rules** for DLL search paths
    - each bit in the number corresponds to a different rule
    - **`LOAD_LIBRARY_SEARCH_SYSTEM32` (0x800)**
    - **`LOAD_LIBRARY_SEARCH_USER_DIRS` (0x400)** (allow DLLs from any directories the program explicitly specifies, but not from any other including the current working directory)
- This code prevents DLL hijacking by setting secure search paths (0xc00 flag).
- Shows awareness of Windows security best practices

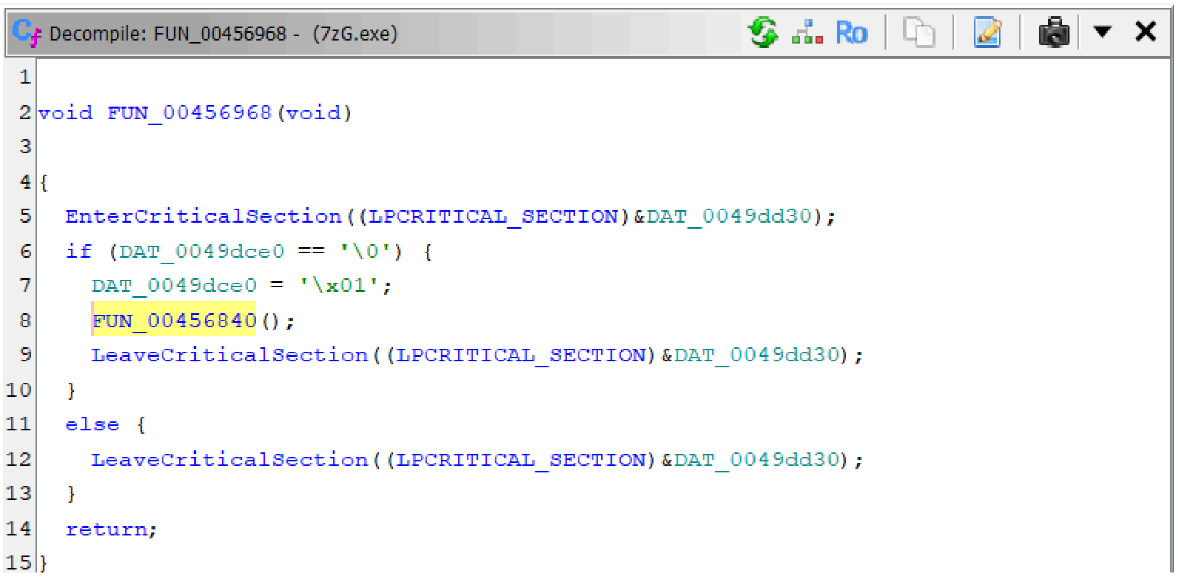


```
004729af cc              ??          CCh

                    ******************************************************
                    *                          FUNCTION
                    ******************************************************
                    undefined __fastcall FUN_004729b0(void)
                      assume GS_OFFSET = 0xff00000000
     undefined        AL:1            <RETURN>
                    FUN_004729b0                                XREF[2]:
004729b0 48 83 ec 28     SUB         RSP,0x28
004729b4 ff 15 f6        CALL        qword ptr [->KERNEL32.DLL::GetVersion]
         26 00 00
004729ba 66 3d 06 00     CMP         AX,0x6
004729be 74 29           JZ          LAB_004729e9
004729c0 48 8d 0d        LEA         RCX,[u_kernel32.dll_0047ff98]
         d1 d5 00 00
004729c7 ff 15 eb        CALL        qword ptr [->KERNEL32.DLL::GetModuleHandleW]
```

```c

void FUN_004729b0(void)

{
  DWORD DVar1;
  HMODULE hModule;
  FARPROC pFVar2;

  DVar1 = GetVersion();
  if ((short)DVar1 != 6) {
    hModule = GetModuleHandleW(L"kernel32.dll");
    pFVar2 = GetProcAddress(hModule,"SetDefaultDllDirectories");
    if (pFVar2 != (FARPROC)0x0) {
      (*pFVar2)(0xc00);
    }
  }
  return;
}
```

1.3.2. *Initialization and Critical Section Security*

- In multiple places, code snippets show proper use of critical sections for thread-safe initialization (example below)
- Structured error handling and resource management
- Safe command-line argument parsing with quotation handling



```c

void FUN_00456968(void)

{
  EnterCriticalSection((LPCRITICAL_SECTION)&DAT_0049dd30);
  if (DAT_0049dce0 == '\0') {
    DAT_0049dce0 = '\x01';
    FUN_00456840();
    LeaveCriticalSection((LPCRITICAL_SECTION)&DAT_0049dd30);
  }
  else {
    LeaveCriticalSection((LPCRITICAL_SECTION)&DAT_0049dd30);
  }
  return;
}
```

- **EnterCriticalSection**: This function locks a critical section to ensure that only the current thread can access the code inside the critical section. It prevents race conditions when multiple threads try to modify shared data concurrently.
- **LeaveCriticalSection**: This function unlocks the critical section, allowing other threads to acquire the lock and execute the protected code.
- **Critical Section Locking for Initialization**: The initialization logic (e.g., setting `DAT_0049dce0`) is protected by this lock, ensuring that only one thread performs the initialization even if multiple threads are attempting to do so simultaneously.

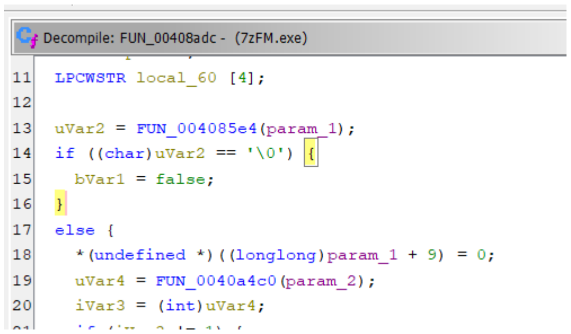
- **Ensuring Safe Exit**: The use of `LeaveCriticalSection` in both branches ensures that the critical section is always released, even if the initialization is skipped. This avoids a deadlock or situation where other threads are indefinitely blocked from entering the critical section.

1.3.3. *Secure File Operation Patterns*

A typical secure file operation in 7-Zip follows this pattern:

1. Pre operation checks
2. File operation execution
3. Cleanup

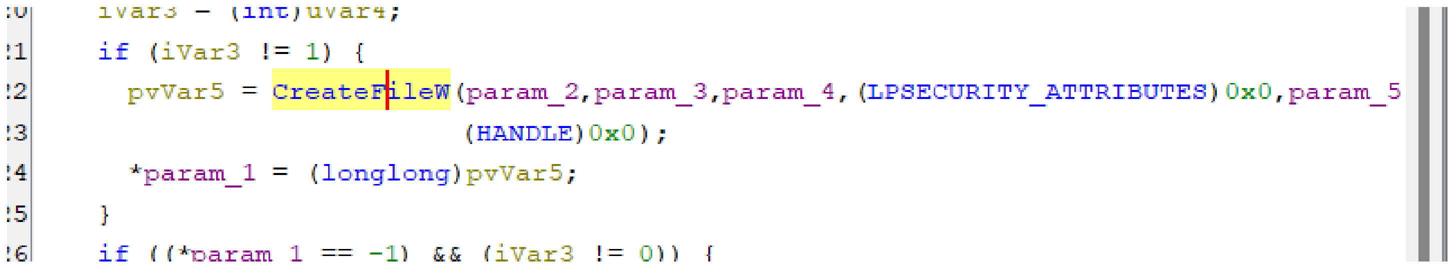
Pre-operation Checks Example:



```
LPCWSTR local_60 [4];

uVar2 = FUN_004085e4(param_1);
if ((char)uVar2 == '\0') {
  bVar1 = false;
}
else {
  *(undefined *)((longlong)param_1 + 9) = 0;
  uVar4 = FUN_0040a4c0(param_2);
  iVar3 = (int)uVar4;
```

Here, `FUN_004085e4` likely performs validation or setup before proceeding with the file operation. If the check fails, the function exits early (`bVar1 = false`), avoiding unsafe execution.

File Operation Execution with Secure Practices

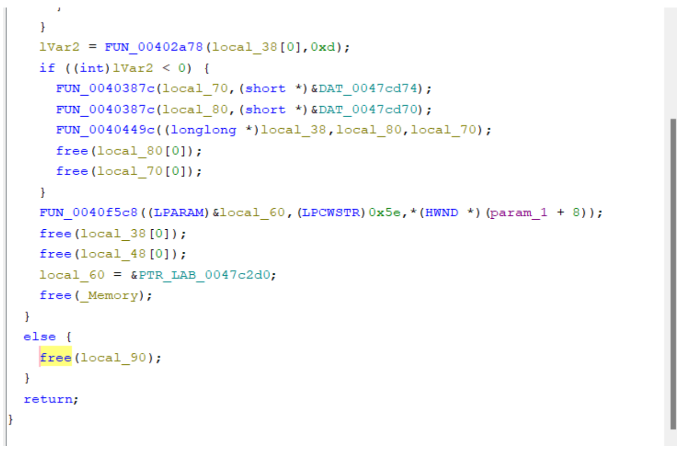


```
  iVar3 = (int)uVar4;
  if (iVar3 != 1) {
    pvVar5 = CreateFileW(param_2,param_3,param_4,(LPSECURITY_ATTRIBUTES)0x0,param_5
                         (HANDLE)0x0);
    *param_1 = (longlong)pvVar5;
  }
  if ((*param_1 == -1) && (iVar3 != 0)) {
```

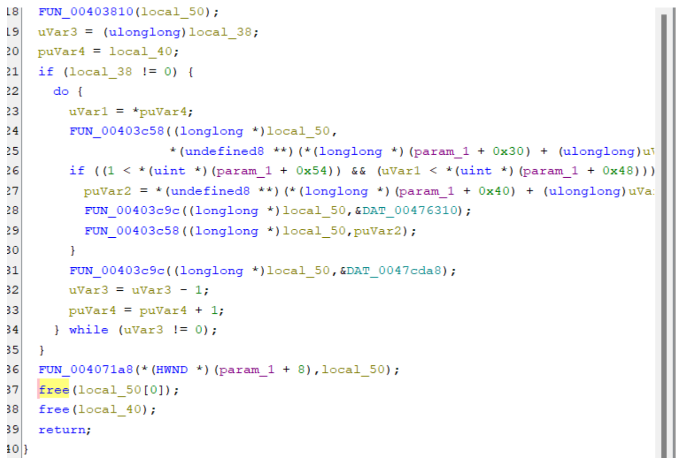
The `CreateFileW` API in the given code is used to open files securely, and its implementation shows good practices for controlled and secure file access.

Cleanup Example

Cleanup ensures no resources are left dangling, which could lead to memory leaks, handle leaks, or undefined behavior.



```
    }
    lVar2 = FUN_00402a78(local_38[0],0xd);
    if ((int)lVar2 < 0) {
      FUN_0040387c(local_70,(short *)&DAT_0047cd74);
      FUN_0040387c(local_80,(short *)&DAT_0047cd70);
      FUN_0040449c((longlong *)local_38,local_80,local_70);
      free(local_80[0]);
      free(local_70[0]);
    }
    FUN_0040f5c8((LPARAM)&local_60,(LPCWSTR)0x5e,*(HWND *)(param_1 + 8));
    free(local_38[0]);
    free(local_48[0]);
    local_60 = &PTR_LAB_0047c2d0;
    free(_Memory);
  }
  else {
    free(local_90);
  }
  return;
}
```



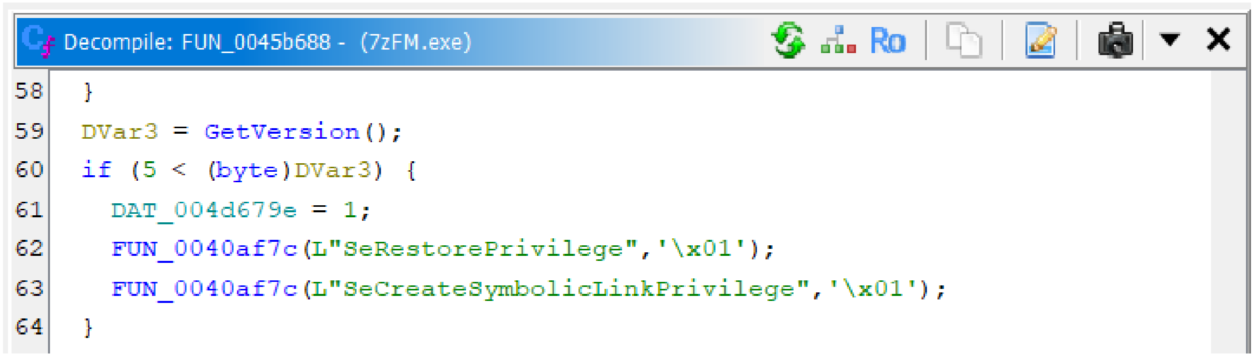
```
  FUN_00403810(local_50);
  uVar3 = (ulonglong)local_38;
  puVar4 = local_40;
  if (local_38 != 0) {
    do {
      uVar1 = *puVar4;
      FUN_00403c58((longlong *)local_50,
                   *(undefined8 **)(*(longlong *)(param_1 + 0x30) + (ulonglong)u
      if ((1 < *(uint *)(param_1 + 0x54)) && (uVar1 < *(uint *)(param_1 + 0x48)))
        puVar2 = *(undefined8 **)(*(longlong *)(param_1 + 0x40) + (ulonglong)uVa
        FUN_00403c9c((longlong *)local_50,&DAT_00476310);
        FUN_00403c58((longlong *)local_50,puVar2);
      }
      FUN_00403c9c((longlong *)local_50,&DAT_0047cda8);
      uVar3 = uVar3 - 1;
      puVar4 = puVar4 + 1;
    } while (uVar3 != 0);
  }
  FUN_004071a8(*(HWND *)(param_1 + 8),local_50);
  free(local_50[0]);
  free(local_40);
  return;
}
```

The above examples show that proper cleanup is being performed, using the **free** keyword. Overall we see though the decompiled code that there is comprehensive error checking in file operations

1.4. Potential Vulnerabilities

1.4.1. *Legacy Support*

- Contains version checks for older Windows versions
- Maintains compatibility code paths
- Could benefit from modernization

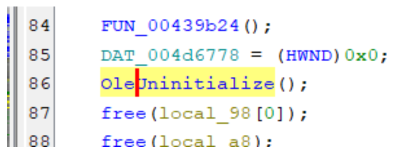


```
  }
  DVar3 = GetVersion();
  if (5 < (byte)DVar3) {
    DAT_004d679e = 1;
    FUN_0040af7c(L"SeRestorePrivilege",'\x01');
    FUN_0040af7c(L"SeCreateSymbolicLinkPrivilege",'\x01');
  }
```

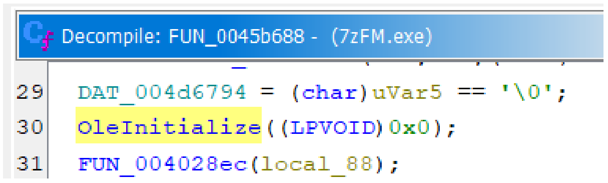
What This Does:

- `GetVersion()` checks the Windows version. The conditional `if (5 < (byte)DVar3)` indicates that the program differentiates behavior depending on whether the OS is older or newer than Windows XP (version 5.1).
- For newer versions (e.g., Vista and above), additional privileges are requested.
- Intention: maintaining backward compatibility with outdated Windows environments, eg. XP.
- However, `GetVersion()` is now **deprecated in modern Windows APIs**, and its use is discouraged in favor of `VerifyVersionInfo` or feature detection methods.

1.4.2. OLE Initialization and Cleanup



```
FUN_00439b24();
DAT_004d6778 = (HWND)0x0;
OleUninitialize();
free(local_98[0]);
free(local_a8);
```



Decompile: FUN_0045b688 - (7zFM.exe)

```
DAT_004d6794 = (char)uVar5 == '\0';
OleInitialize((LPVOID)0x0);
FUN_004028ec(local_88);
```

Unfortunately, the code explicitly (manually) initializes and uninitializes OLE resources as shown above. Object Linking and Embedding (OLE). is a technology developed by Microsoft that allows embedding and linking to documents and other objects. Manually managing these resources can lead to resource leaks or improper cleanup if not handled correctly.

Solution: Use RAII (Resource Acquisition Is Initialization) or scoped resource wrappers to manage OLE resources automatically. RAII ties the lifecycle of OLE resources to the scope of objects, ensuring deterministic cleanup when these objects go out of scope. This ensures that resources are always properly released, reducing the risk of resource leaks and making the code easier to manage.

Note: Project is still maintained and updated as of less than 6 months ago. Therefore, improvements can be expected.

1.5. Conclusion of Static Analysis

The code demonstrates professional-grade software engineering practices with a strong focus on security and stability. The architecture follows Windows programming best practices, showing signs of mature development processes and security awareness. While some code appears to maintain legacy compatibility, this is appropriate for widely-used utility software.

2. Analysis of Ghidra's decompilation

In order to analyze Ghidra's decompilation process, we used the Understand tool. We analyzed both the decompiled code obtained in Part 1 as well as the original source code of 7Zip. This helped us understand the working of the decompiler better, which was important as it's one of the major functionalities of Ghidra.

2.1. Analysis of Code Breakdown

2.1.1. *Source Code:*

- The decompiled code has a higher percentage of source code (80%) compared to the original code (61%). This might be due to the removal of comments and preprocessing directives during decompilation.

2.1.2. *Number of files/folders*:

- The decompiler does not maintain any semblance of the folder structure or files but merely puts all the decompiled code in a single file.

2.1.3. *Blank lines*:

- Interestingly, the percentage of blank lines is higher in the decompiled code (19%) compared to the original code (14%).
- This could be a result of how the decompilation process formats the code, possibly inserting more blank lines for readability.

2.1.4. *Comments*:

- The decompiled code shows 0% for comments
- This shows that comments are stripped out during the compilation process and are not present in the decompiled output.

2.1.5. *Inactive Code*:

- There is no mention of inactive code in the summary
- This shows that the decompiler would not include conditional compilation segments that were not compiled into the binary.

2.1.6. *PreProcessor Directives*:

- Similarly, preprocessor directives are not present in the decompiled code since they are processed before compilation.

2.2. Analysis of Large Entities

2.2.1. *Naming Conventions*:

- **Original Code**: Uses meaningful names like `CPanel` and `CHandler`, which provide insight into their roles and responsibilities within the codebase.
- **Decompiled Code**: Uses placeholder names like `FUN_0047b694`, reflecting the lack of source-level context in the decompiled output.

2.2.2. *Size Representation:*

- **Original Code**: Sizes of entities (classes or functions) are based on their source-level components, such as classes, functions, and modules, that reflect the developer's intended design.
- **Decompiled Code**: Sizes of entities are reflected by the binary structure and decompilation process, which may result in broader or more fragmented representations.

2.2.3. *Code Structure and Clarity*:

- **Original Code**: Easier to understand and maintain due to meaningful naming and organized structure.
- **Decompiled Code**: More challenging to interpret due to generic function names and lack of comments or documentation.

2.3. Comparison of Architecture and Design

2.3.1. *Original Source Code*

Analyzing the folder structure of 7-Zip original code shows that the project follows a **layered**, **modular design**, and uses **platform-specific** code to ensure cross-platform compatibility. It organizes the functionality into clearly defined modules (compression, encryption, UI, etc.) and separates core logic from platform-specific implementations.

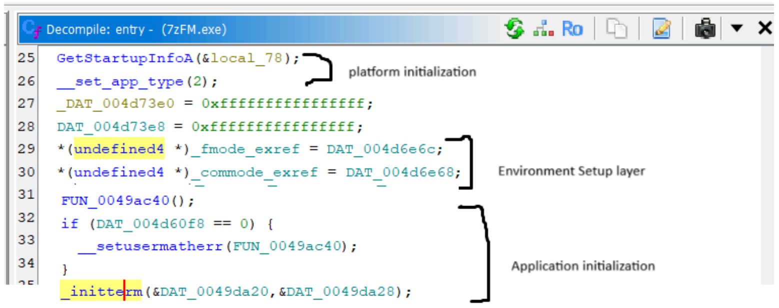
1. **Layered Architecture**:
    - The project is structured into different layers (e.g., **core compression logic**, **file handling**, **UI**, **platform-specific code**). Each subfolder and file in the project handles a specific concern of the application, which is a typical characteristic of **layered architecture**.
    - For example:
        - **Compression algorithms** are handled in the `compress` folder.
        - **Encryption** is handled in the `crypto` folder.
        - **User interface** is handled in the `UI` folder.
        - **Platform-specific logic** is in the `Windows` folder (likely only for Windows OS).
2. **Modular Design**:
    - The project is broken down into **modules** like compression, encryption, UI, etc., with clear boundaries between them. Each module is self-contained and can be worked on or tested independently. This is a sign of **modular design**, where each module has a specific responsibility.
    - For example, all the code related to file compression is likely in the `compress` folder, while the user interface is handled separately in the `UI` folder.
3. **Cross-Platform Support**:
    - The presence of an **`asm`** folder with subfolders for **ARM**, **ARM64**, and **x86** indicates that 7-Zip is designed to support multiple processor architectures, which is important for cross-platform compatibility.
    - The **Windows** folder in the `cpp` directory suggests that 7-Zip includes platform-specific code for Windows. This reflects a **cross-platform** or **platform-specific modular** approach, where code is optimized for specific operating systems or hardware architectures but shares common logic.
4. **Separation of Concerns**:
    - The project separates different concerns into different folders, which helps with code maintainability and readability. For instance, **compression algorithms**, **file handling**, and

encryption are all managed separately in their own modules, following the **separation of concerns** pattern.

2.3.2. *Preserved Patterns in Decompiled Code*

Layered Design

- Layered design becomes Scrambled and distorted, as seen in the example of the initialization sequence
- No separation of UI and core logic
- Abstraction layers flattened



```
GetStartupInfoA(&local_78);
__set_app_type(2);
_DAT_004d73e0 = 0xffffffffffffffff;
DAT_004d73e8 = 0xffffffffffffffff;
*(undefined4 *)_fmode_exref = DAT_004d6e6c;
*(undefined4 *)_commode_exref = DAT_004d6e68;
FUN_0049ac40();
if (DAT_004d60f8 == 0) {
  __setusermatherr(FUN_0049ac40);
}
_initterm(&DAT_0049da20,&DAT_0049da28);
```

Modularity

- The ability to alter functions without impacting other functions is retained in the decompiled code (for the most part)
- However Ghidra does not inherently group related functions together during decompilation because its primary goal is to reconstruct machine code into human-readable pseudo-code.

2.3.3. *Lost Architectural Elements After Decompilation*

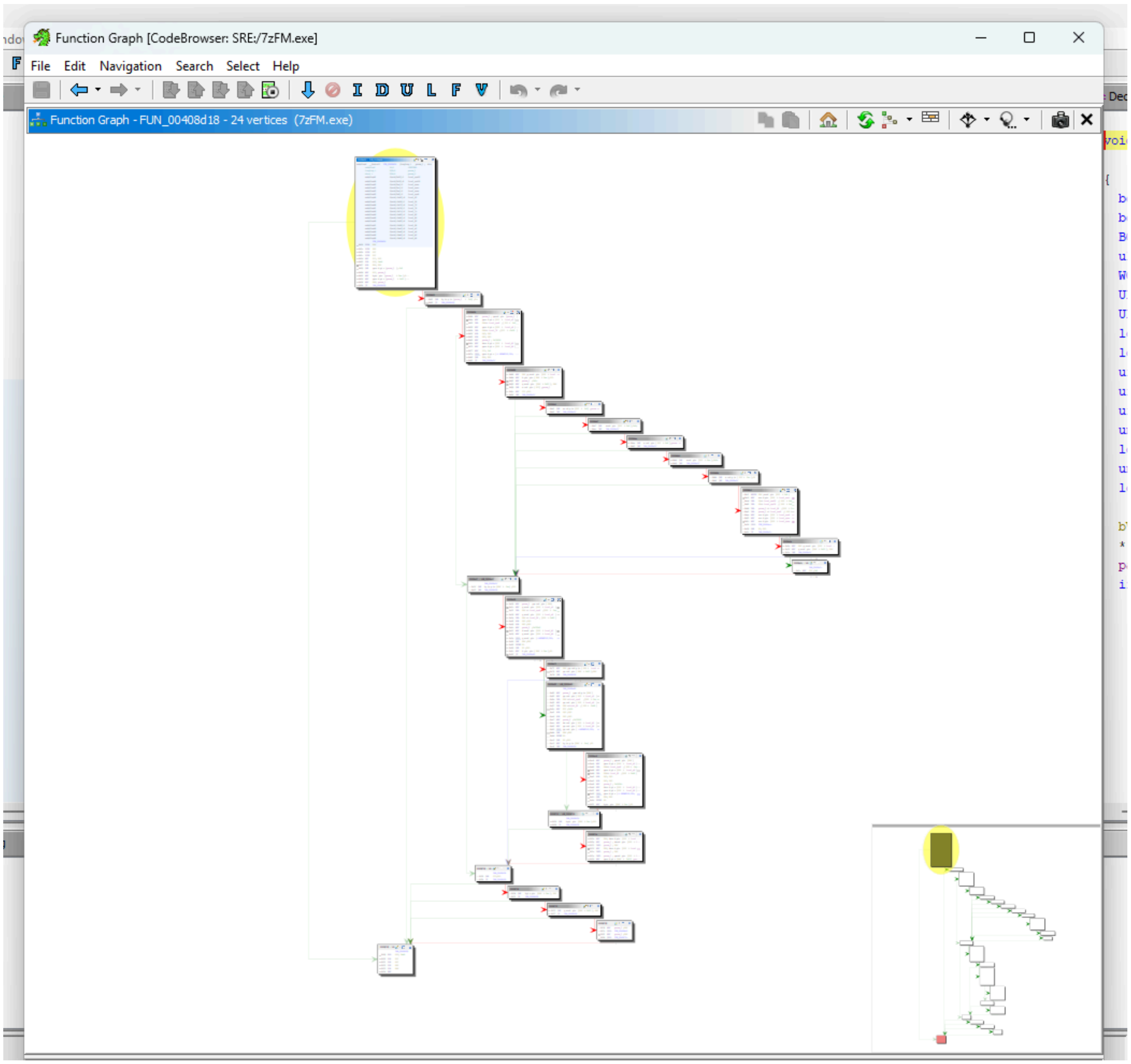
File Organization

- Original directory structure lost
- Module boundaries less clear
- Package relationships obscured

Cross-Platform Elements

- Platform-specific code mixed
- Abstraction barriers less visible
- Implementation details exposed

2.3.4. *Function Graph Analysis*


Function Graph [CodeBrowser: SRE:/7zFM.exe]
File Edit Navigation Search Select Help
Function Graph - FUN_00408d18 - 24 vertices (7zFM.exe)

When utilizing Ghidra for code analysis, the function graph feature provides a visual representation of the control flow within a selected function. This graph highlights the relationships between different code blocks by showing the paths and branches.

In most cases, the overall code structure is valid and follows good practices, but I have highlighted here a particular function that may have some issues regarding its design. As can be seen in the function graph generated by Ghidra, there are numerous nested function calls. Although modern compilers optimize function calls, excessive nesting can still impact performance. Function calls introduce overhead, and deep nesting can lead to slower execution times due to frequent context switching.

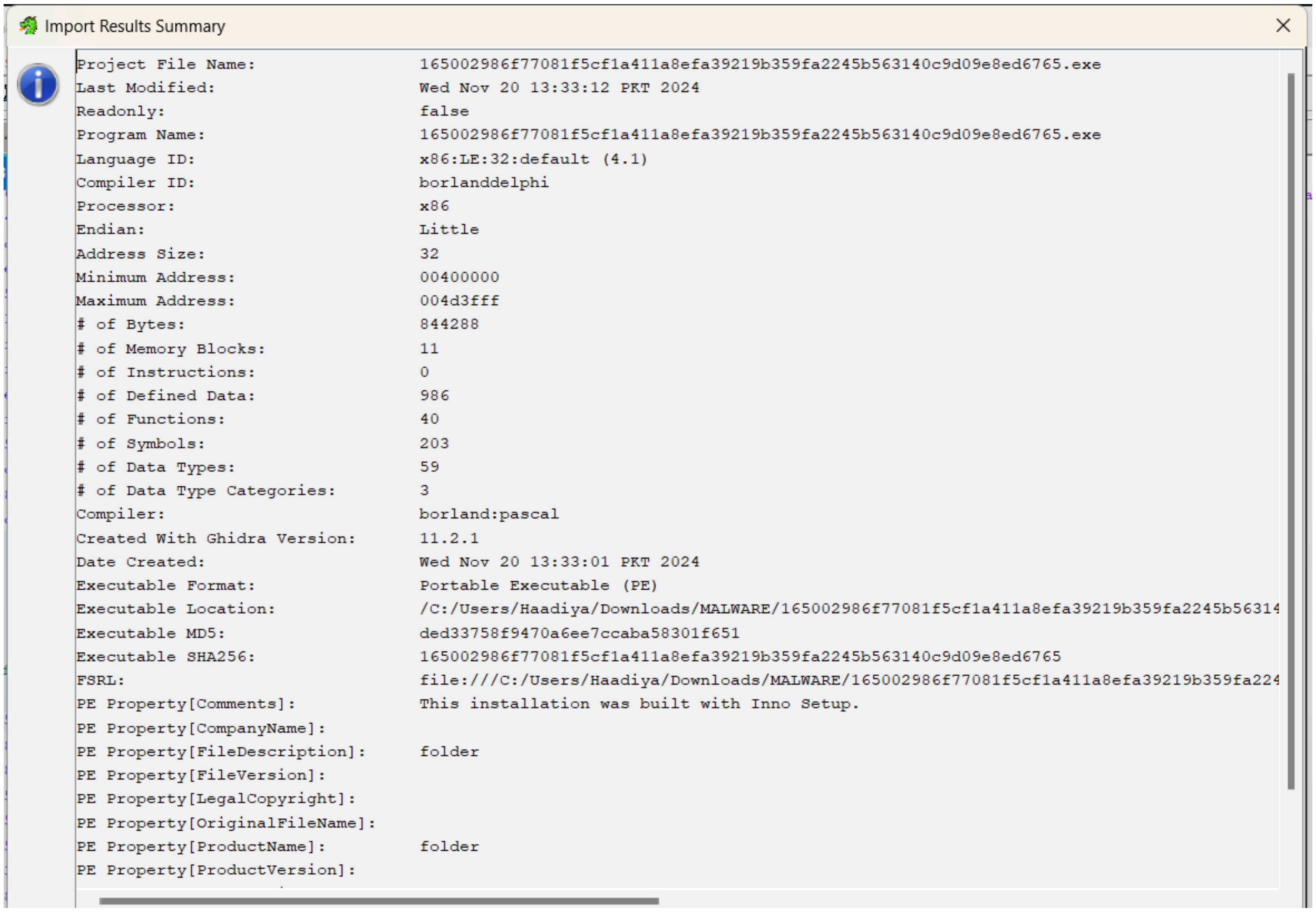
3. Malware Analysis

3.1. Selected Malware

The malware was found through MalwareBazaar, a platform dedicated to sharing malware samples with the infosec and cyber security community, antivirus vendors, and threat intelligence providers. The malware selected is an adware.

3.2. Analysis of Malicious Patterns from decompiled code

3.2.1. *Summary*



Import Results Summary

```
Project File Name:              165002986f77081f5cf1a411a8efa39219b359fa2245b563140c9d09e8ed6765.exe
Last Modified:                  Wed Nov 20 13:33:12 PKT 2024
Readonly:                       false
Program Name:                   165002986f77081f5cf1a411a8efa39219b359fa2245b563140c9d09e8ed6765.exe
Language ID:                    x86:LE:32:default (4.1)
Compiler ID:                    borlanddelphi
Processor:                      x86
Endian:                         Little
Address Size:                   32
Minimum Address:                00400000
Maximum Address:                004d3fff
# of Bytes:                     844288
# of Memory Blocks:             11
# of Instructions:              0
# of Defined Data:              986
# of Functions:                 40
# of Symbols:                   203
# of Data Types:                59
# of Data Type Categories:      3
Compiler:                       borland:pascal
Created With Ghidra Version:    11.2.1
Date Created:                   Wed Nov 20 13:33:01 PKT 2024
Executable Format:              Portable Executable (PE)
Executable Location:            /C:/Users/Haadiya/Downloads/MALWARE/165002986f77081f5cf1a411a8efa39219b359fa2245b56314
Executable MD5:                 ded33758f9470a6ee7ccaba58301f651
Executable SHA256:              165002986f77081f5cf1a411a8efa39219b359fa2245b563140c9d09e8ed6765
FSRL:                           file:///C:/Users/Haadiya/Downloads/MALWARE/165002986f77081f5cf1a411a8efa39219b359fa224
PE Property[Comments]:          This installation was built with Inno Setup.
PE Property[CompanyName]:
PE Property[FileDescription]:   folder
PE Property[FileVersion]:
PE Property[LegalCopyright]:
PE Property[OriginalFileName]:
PE Property[ProductName]:       folder
PE Property[ProductVersion]:
```

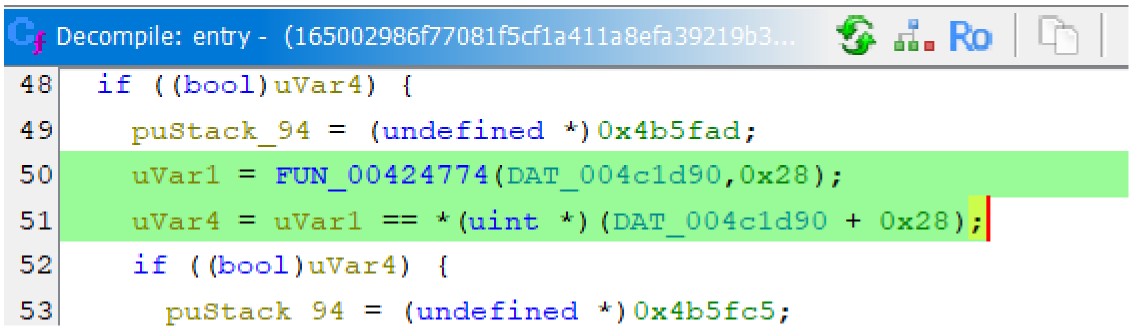
The information shown in the Ghidra summary itself does not directly indicate maliciousness. All the technical indicators shown are normal characteristics that could be found in legitimate software:

1. The use of Borland Delphi/Pascal is a legitimate programming environment
2. Inno Setup is a legitimate installer creation tool used by many legitimate software packages
3. The file size (844,288 bytes), memory layout, and number of functions/symbols appear normal

4. Having PE (Portable Executable) properties and folder descriptions is common for Windows installers

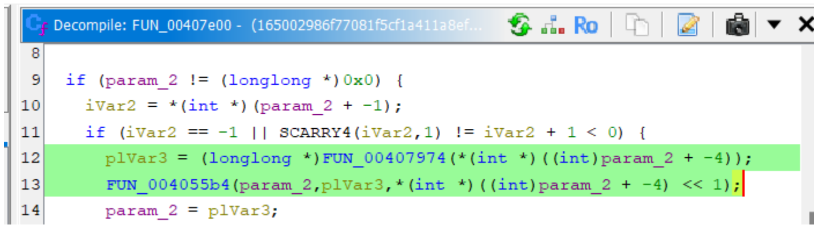
Therefore, further analysis is needed to determine its malicious characteristics. For this purpose, the remaining sections will analyze the decompiled code of the entry function of the software.

3.2.2. *Suspicious Memory Operations*

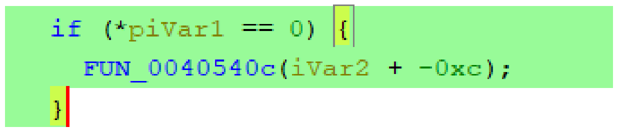

Decompile: entry - (165002986f77081f5cf1a411a8efa39219b3...

```
if ((bool)uVar4) {
  puStack_94 = (undefined *)0x4b5fad;
  uVar1 = FUN_00424774(DAT_004c1d90,0x28);
  uVar4 = uVar1 == *(uint *)(DAT_004c1d90 + 0x28);
  if ((bool)uVar4) {
    puStack_94 = (undefined *)0x4b5fc5;
```

This appears to be a memory comparison check that could be used for anti-debugging or anti-analysis techniques. Legitimate software rarely performs such direct memory comparisons.


Decompile: FUN_00407e00 - (165002986f77081f5cf1a411a8ef...

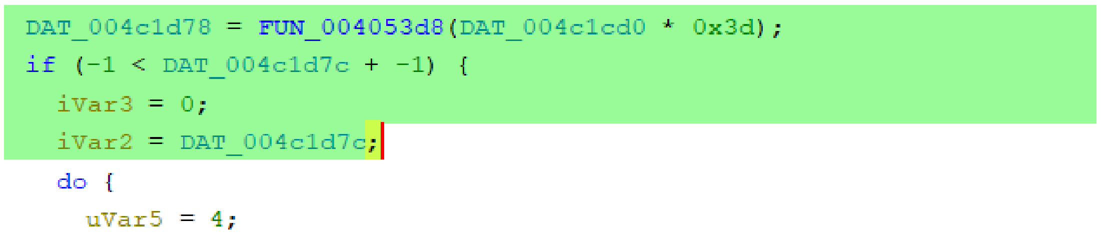
```

if (param_2 != (longlong *)0x0) {
  iVar2 = *(int *)(param_2 + -1);
  if (iVar2 == -1 || SCARRY4(iVar2,1) != iVar2 + 1 < 0) {
    plVar3 = (longlong *)FUN_00407974(*(int *)((int)param_2 + -4));
    FUN_004055b4(param_2,plVar3,*(int *)((int)param_2 + -4) << 1);
    param_2 = plVar3;
```



```
  if (*piVar1 == 0) {
    FUN_0040540c(iVar2 + -0xc);
  }
```

The above is another suspicious example where the program Directly manipulates memory addresses using pointer arithmetic and uses bitwise shifting (<< 1) which could be used for, buffer overflows, heap manipulation or perhaps memory corruption.

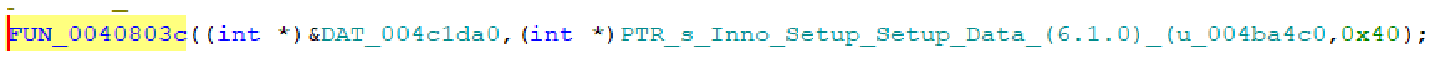
3.2.3. *Improper Resource Handling*



```
DAT_004c1d78 = FUN_004053d8(DAT_004c1cd0 * 0x3d);
if (-1 < DAT_004c1d7c + -1) {
  iVar3 = 0;
  iVar2 = DAT_004c1d7c;
  do {
    uVar5 = 4;
```

The highlighted portion represents memory allocation without proper error checking or cleanup mechanisms. This could lead to memory leaks or be used for malicious purposes like buffer overflows.

3.2.4. *Suspicious String Operation*



```
FUN_0040803c((int *)&DAT_004c1da0,(int *)PTR_s_Inno_Setup_Setup_Data_(6.1.0)_(u_004ba4c0,0x40);
```

This appears to be string manipulation involving Inno Setup, but the fixed size operation (0x40) without bounds checking is suspicious and could be used for buffer overflow attacks.

Interestingly, InnoSetup is a legitimate software installer creation tool, often abused by malware authors because:

1. It's trusted by Windows
2. Can bypass some security controls
3. Can execute custom code during installation

So, the malware is perhaps masking itself as an Inno Setup installer in order to appear legitimate.

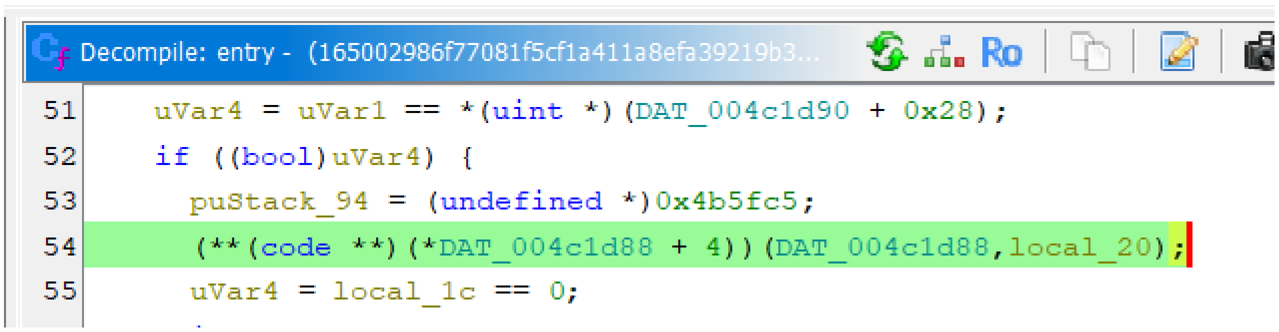
3.2.5. *Missing Security Measures*

Viewing the code in comparison to the 7Zip code clearly shows the following:

- No evidence of proper DLL loading security checks
- Lack of privilege level verification
- No proper exception handling
- Absence of resource cleanup in error paths

In contrast, the 7Zip decompiled code ensured secure practices in all of the above aspects.

3.2.6. *Obfuscated Function Calls*



```
  uVar4 = uVar1 == *(uint *)(DAT_004c1d90 + 0x28);
  if ((bool)uVar4) {
    puStack_94 = (undefined *)0x4b5fc5;
    (**(code **)(*DAT_004c1d88 + 4))(DAT_004c1d88,local_20);
    uVar4 = local_1c == 0;
```

The use of indirect function calls through pointers is suspicious , and commonly used to hide API calls or implement shellcode. This is suspicious because:

1. It makes code harder to analyze by hiding what functions are being called
2. Can be used to dynamically call Windows APIs to avoid detection
3. Common technique to hide malicious functions like:
    - Process injection
    - System modifications

Instead of directly calling `CreateProcess()` or `WriteFile()`, malware might use these indirect calls to make it harder for security tools to detect what it's doing.

4. Key Findings & Conclusion

This analysis of 7-Zip using Ghidra, alongside a comparative examination of malware, demonstrates how structured software development contrasts with malicious code practices.

4.1. Static Analysis of 7-Zip: Findings

1. The 7-Zip codebase reflects high-quality software engineering with security measures.
2. The modular, layered architecture evident in the original source code is partially preserved in decompiled output, though some abstraction and organizational clarity are lost.
3. Potential vulnerabilities like legacy compatibility and manual OLE management were identified but do not pose immediate security risks given proper maintenance.

4.2.Capabilities and Limitations of Ghidra: Findings

1. Ghidra effectively recovers assembly-level and C-like high-level code, preserving structural patterns and identifying core functions.
2. However, it cannot restore comments, preprocessor directives, or the original folder structure, which affects the readability and maintainability of decompiled code.
3. Decompiled function names and modular boundaries lack context, emphasizing the limitations of reverse engineering in recovering complete architectural designs.

4.3. Analysis of Malware: Findings

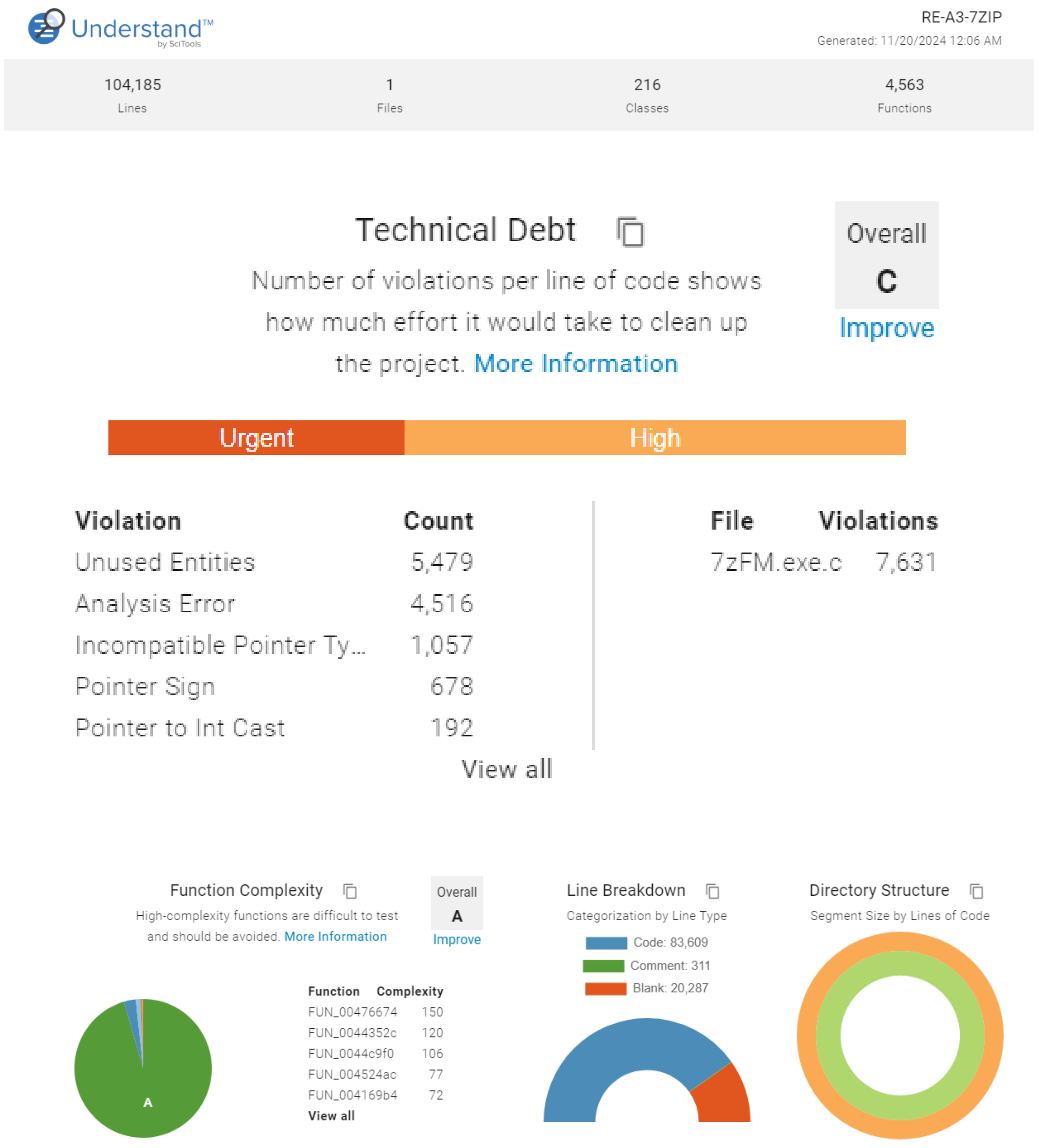
1. Malware exhibited suspicious patterns which were contrasting sharply with 7-Zip's secure practices.
2. This helped us understand how Ghidra can be used for Malware analysis.

4.4 Conclusion

After this project, we not only got insights into 7-Zip's codebase but also illustrated Ghidra's utility as a reverse engineering tool and the critical distinctions between legitimate software and malicious executables.

Appendix A

Understand Reports for Decompiled 7Zip Code



Understand™ by SciTools

RE-A3-7ZIP
Generated: 11/20/2024 12:06 AM

104,185	1	216	4,563
Lines	Files	Classes	Functions

Technical Debt

Number of violations per line of code shows how much effort it would take to clean up the project. More Information

Overall C Improve

Violation	Count
Unused Entities	5,479
Analysis Error	4,516
Incompatible Pointer Ty...	1,057
Pointer Sign	678
Pointer to Int Cast	192

View all

File	Violations
7zFM.exe.c	7,631

Function Complexity

High-complexity functions are difficult to test and should be avoided. More Information

Overall A Improve

Function	Complexity
FUN_00476674	150
FUN_0044352c	120
FUN_0044c9f0	106
FUN_004524ac	77
FUN_004169b4	72

View all

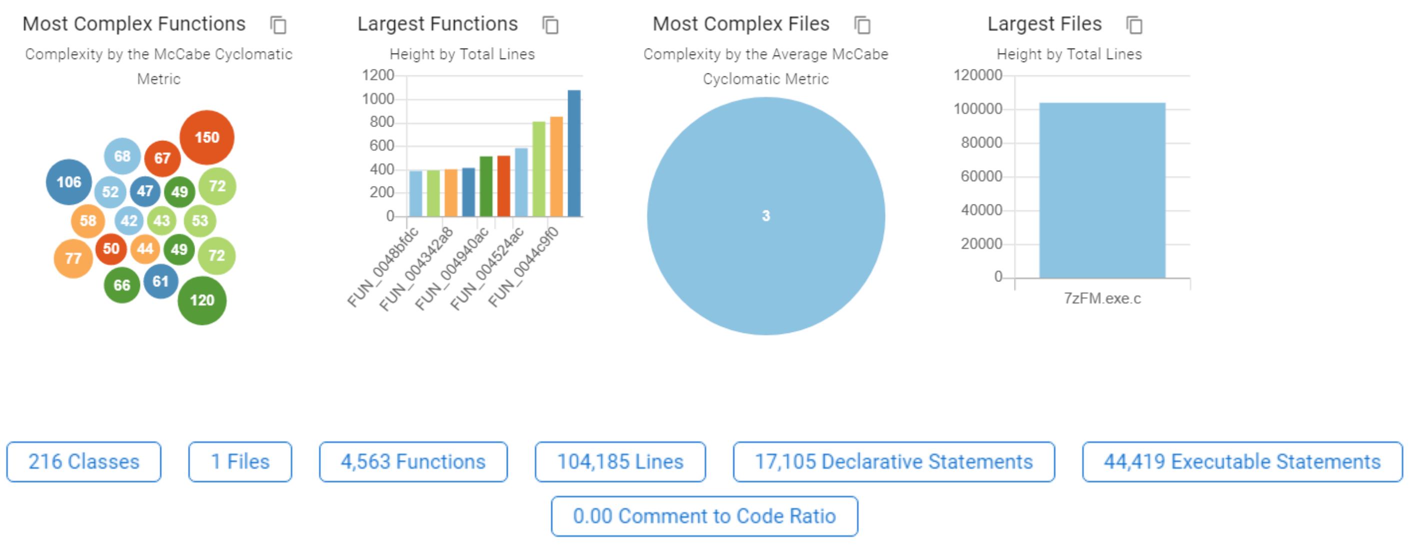
Line Breakdown

Categorization by Line Type

🟦	Code: 83,609
🟩	Comment: 311
🟥	Blank: 20,287

Directory Structure

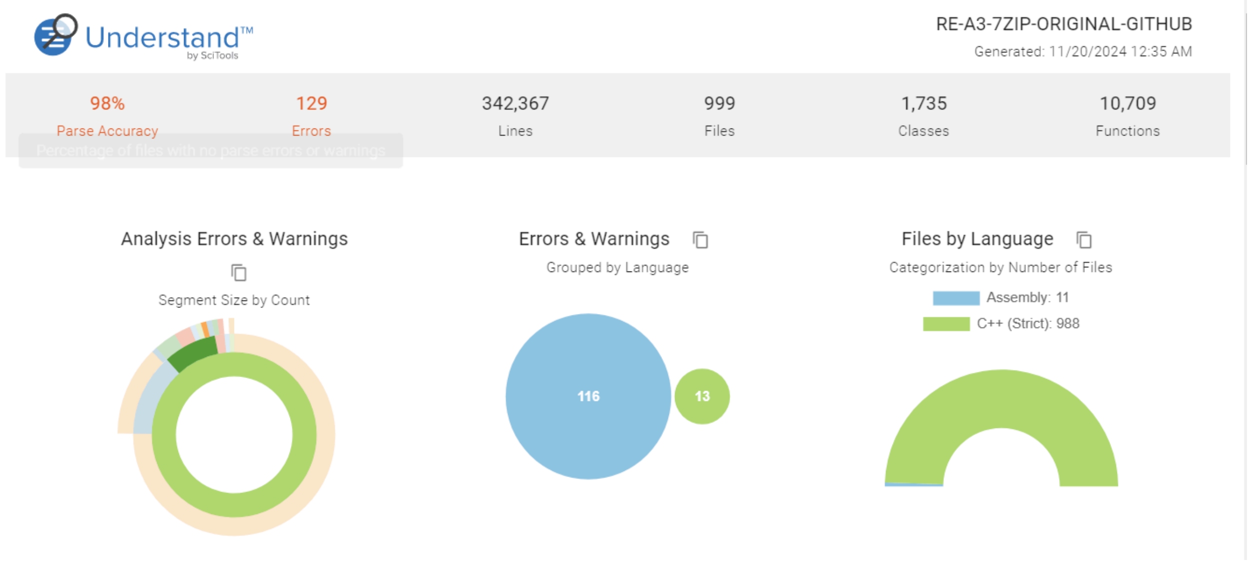
Segment Size by Lines of Code



216 Classes | 1 Files | 4,563 Functions | 104,185 Lines | 17,105 Declarative Statements | 44,419 Executable Statements

0.00 Comment to Code Ratio

Understand Reports for Original 7Zip code

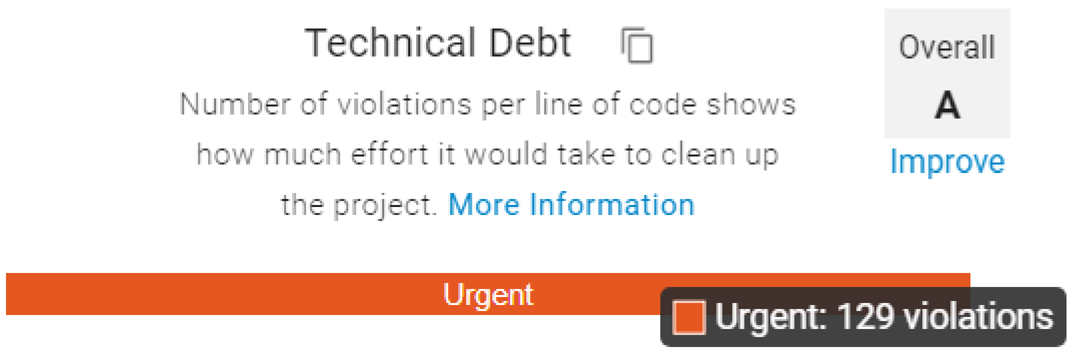


Understand™ by SciTools

RE-A3-7ZIP-ORIGINAL-GITHUB
Generated: 11/20/2024 12:35 AM

98%	129	342,367	999	1,735	10,709
Parse Accuracy	Errors	Lines	Files	Classes	Functions

Percentage of files with no parse errors or warnings



Technical Debt

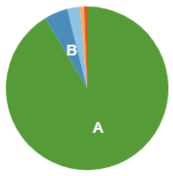
Number of violations per line of code shows how much effort it would take to clean up the project. More Information

Overall
A
Improve

Function Complexity

High-complexity functions are difficult to test and should be avoided. More Information

Overall
A
Improve

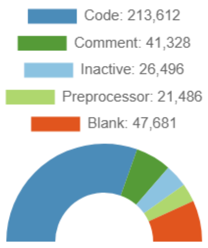


Function	**Complexity**
UpdateItems	254
OpenStream2	230
Bench	229
Update	150
OpenVolume	130

View all

Line Breakdown

Categorization by Line Type

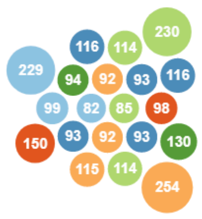


Directory Structure

Segment Size by Lines of Code

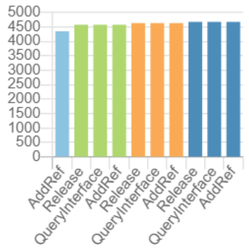
Most Complex Functions

Complexity by the McCabe Cyclomatic Metric



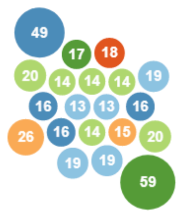
Largest Functions

Height by Total Lines



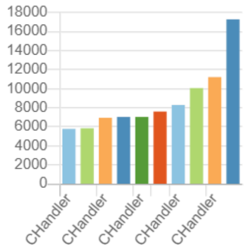
Most Complex Classes

Complexity by the Average McCabe Cyclomatic Metric



Largest Classes

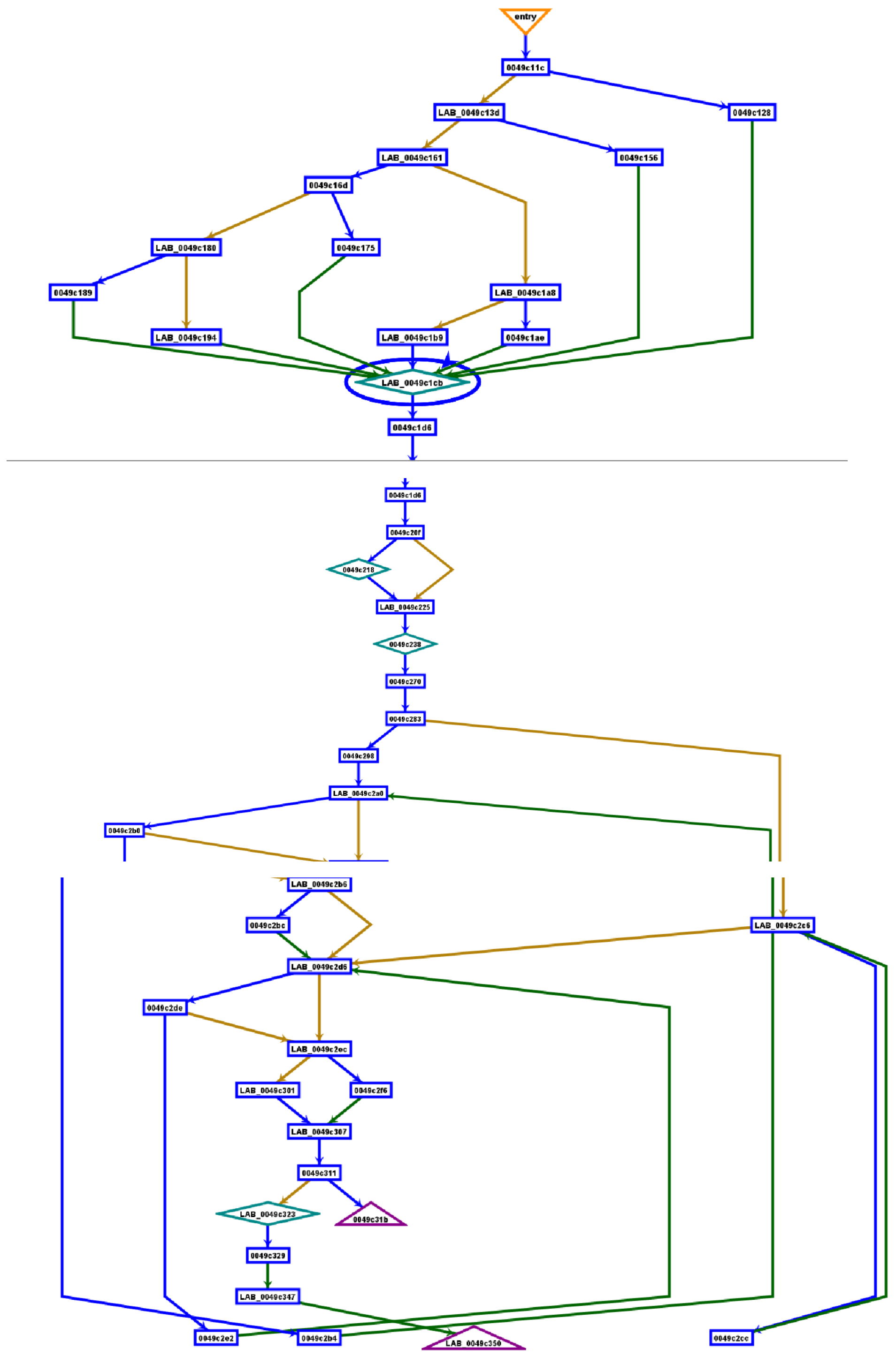
Height by Total Lines



1,735 Classes | 999 Files | 10,709 Functions | 342,367 Lines | 53,190 Declarative Statements | 125,705 Executable Statements

0.19 Comment to Code Ratio

Function graph of entry function (7Zip decompiled)


entry
0049c11c
LAB_0049c13d
0049c128
LAB_0049c161
0049c156
0049c16d
LAB_0049c180
0049c175
0049c189
LAB_0049c1a8
LAB_0049c194
LAB_0049c1b9
0049c1ae
LAB_0049c1cb
0049c1d6
0049c1d6
0049c20f
0049c218
LAB_0049c225
0049c238
0049c270
0049c283
0049c298
LAB_0049c2a0
0049c2b0
LAB_0049c2b6
0049c2bc
LAB_0049c2c6
LAB_0049c2d6
0049c2de
LAB_0049c2ec
LAB_0049c301
0049c2f6
LAB_0049c307
0049c311
LAB_0049c323
0049c31b
0049c329
LAB_0049c347
0049c2e2
0049c2b4
LAB_0049c350
0049c2cc