AIR UNIVERSITY

Islamabad, Pakistan

Building Executable with Zero Imports

A Data Structures Approach to Understanding PE File Format and Dynamic Linking

PROJECT REPORT

Submitted By:

Name	Roll Number
M Gulraiz Khan	241531
Haider Mustafa	242380

Submitted To:

Mam Mehmoona Jabeen
Course Instructor

Date: 20 December 2024

Submission Date: January 28, 2026

Contents

List of Figures

1 Introduction

Modern antivirus solutions work primarily through two main approaches: **static analysis** and **dynamic analysis**. Static analysis focuses on detecting known malicious signatures by comparing them against a database of known threats. This method also identifies suspicious characteristics and patterns in malware behavior by analyzing behavioral signatures.

Dynamic analysis takes a different approach by monitoring the program's behavior in real-time to determine if it exhibits malicious activity.

This report concentrates on the first method, exploring techniques that help break common signatures such as suspicious strings and imports that malware typically contains. The discussion begins with understanding how PE (Portable Executable) files resolve imports through various **data structures**, how the Import Address Table (IAT) functions as a lookup table, and how to build malware that has zero imports by dynamically resolving all Windows APIs.

1.1 Data Structures in PE Files

PE files utilize several interconnected data structures to manage dynamic linking:

- **Import Directory Table** - Root structure containing metadata about imports
- **Import Lookup Table (ILT)** - Array of function references
- **Import Address Table (IAT)** - Array of function pointers (runtime resolution)
- **Hint/Name Table** - Hash table-like structure for function name lookups
- **Export Address Table** - Function pointer array in DLLs
- **Process Environment Block (PEB)** - Linked list of loaded modules

2 Linking Mechanisms

2.1 Static Linking

When developing a program using Windows APIs, the functions in your application are linked against their respective import libraries during the compilation process. The operating system's loader resolves these function addresses at compile time.

Static Linking Process

Compile Time:

1. Linker embeds function references in binary
2. Import Address Table populated with placeholder values
3. Dependencies hardcoded into PE headers

Load Time:

1. Loader reads Import Directory Table
2. All dependencies must be present
3. Addresses resolved before execution begins

If any functions are missing, the application will not run at all. This creates a **hard dependency** on external libraries.

2.2 Dynamic Linking

Dynamic linking offers a more flexible approach by using `GetModuleHandle` and `GetProcAddress` to resolve function addresses during program execution (runtime rather than compile time).

Dynamic Linking Advantages

- **No hard dependencies** - Application can handle missing DLLs gracefully
- **Runtime flexibility** - Load different implementations based on conditions
- **Evasion capability** - Functions not revealed until execution
- **Reduced IAT footprint** - Minimal static import signatures

This approach does not create a hard dependency on a DLL. When the DLL or a specific function is missing, your application can handle the error gracefully.

2.3 Dynamic Linking Algorithm

The dynamic linking process follows a well-defined algorithm:

3 PE Data Structures Analysis

3.1 Import Directory Table Structure

The import information of a PE file begins with the **Import Directory Table**. This is the root data structure that orchestrates the entire import resolution process.

Algorithm 1 PE Dynamic Linking Process

```
1:  Input: PE binary file
2:  Output: Resolved function addresses
3:
4:  PE_Loader inspects binary
5:  dependencies ← Parse_Import_Directory()
6:  for each DLL in dependencies do
7:      Load_DLL_Into_Memory(DLL)
8:  end for
9:
10: for each function_reference in Import_Lookup_Table do
11:     symbol ← function_reference
12:     address ← Resolve_Symbol(symbol)
13:     Update_IAT(address)
14: end for
15:
16: return Program ready for execution
```

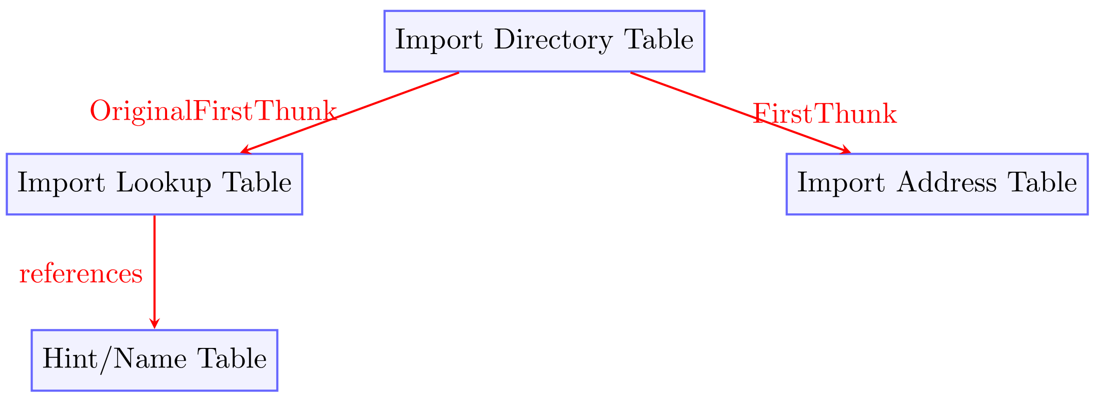


Figure 1: Import Directory Table Pointer Structure

3.2 Image Import Descriptor

The Image Import Directory is a data structure located within a PE file. It plays a crucial role in the dynamic resolution of functions and APIs by allowing programs to import and utilize external libraries during runtime.

3.2.1 Structure Definition and Memory Layout

```c
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;        // 0 for terminating
           null entry
        DWORD   OriginalFirstThunk;     // RVA to ILT
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;              // 0 if not bound
    DWORD   ForwarderChain;             // -1 if no
       forwarders
    DWORD   Name;                       // RVA to DLL name
    DWORD    FirstThunk;                    // RVA to IAT
} IMAGE_IMPORT_DESCRIPTOR;
```

Listing 1: Image Import Descriptor Structure

Structure Members Analysis

Key Data Structure Components:

1. **OriginalFirstThunk** (DWORD - 4 bytes)
    - Stores pointer (RVA) to Import Lookup Table
    - Acts as an array index to function names/ordinals
    - Read-only reference to function identifiers
2. **Name** (DWORD - 4 bytes)
    - Pointer to null-terminated ASCII string
    - Contains DLL name (e.g., "kernel32.dll")
    - Used for module identification
3. **FirstThunk** (DWORD - 4 bytes)
    - Pointer to Import Address Table
    - Initially mirrors ILT structure
    - Overwritten with actual function addresses at runtime
    - Modified by PE loader during import resolution

Table 1: IMAGE_IMPORT_DESCRIPTOR Memory Layout

Field	**Offset**	**Size**	**Purpose**
OriginalFirstThunk	0x00	4 bytes	Pointer to ILT
TimeDateStamp	0x04	4 bytes	Binding timestamp
ForwarderChain	0x08	4 bytes	Forwarder index
Name	0x0C	4 bytes	DLL name pointer
FirstThunk	0x10	4 bytes	Pointer to IAT

3.3 Import Lookup Table (ILT)

The Import Lookup Table, also referred to as the Import Name Table (INT), is an **array data structure** of function names/references that tells the loader which functions are needed from the DLL being imported.

ILT as an Array Structure

Data Structure Type: Array of DWORD/QWORD values
Array Elements:

- Each element is either:
  1. Pointer to Hint/Name structure (if bit 31/63 = 0)
  2. Ordinal value (if bit 31/63 = 1)
- Terminated by NULL entry (0x00000000)

Use Case: Function forwarding through indirection

This table is especially useful in function forwarding, where function calls are redirected from one DLL to another through a level of indirection.

3.4 Hint/Name Table Structure

This structure is defined in the Windows Internals header file `winnt.h`:

```c
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;        // Index into Export Name Pointer
        Table
    CHAR    Name[1];     // Variable-length null-terminated
        string
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Listing 2: Hint/Name Table Structure

Hint/Name Table: Hash-Like Lookup

Structure Type: Optimized lookup structure (hint + fallback)
Lookup Algorithm:

```
1: hint_index ← IMAGE_IMPORT_BY_NAME.Hint
2: if Export_Table[hint_index].Name == IMAGE_IMPORT_BY_NAME.Name then
3:     return Export_Table[hint_index].Address
4: else
5:     return Binary_Search(Export_Table, Name)
6: end if
```

Time Complexity:

- Best case: O(1) - Direct hint match
- Worst case: O(log n) - Binary search fallback

The Hint is a number used to look up a function. It serves as an **index** to the export name pointer table, which contains pointers to functions exported from a DLL. If the index lookup fails, a binary search on the name is performed.

3.5 Import Address Table (IAT)

The Import Address Table is similar to the ILT but serves a different purpose in the resolution process.

IAT vs ILT Comparison

Data Structure Similarities:

- Both are arrays of pointers
- Both indexed by same ordinal/name references
- Same size and alignment

Key Difference:

- **ILT:** Read-only, contains references to names/ordinals
- **IAT:** Writable, overwritten with actual function addresses at runtime

Runtime Modification:

1. Initially: IAT = copy of ILT structure
2. After loader resolution: IAT = array of function pointers
3. Program calls functions through IAT indirection

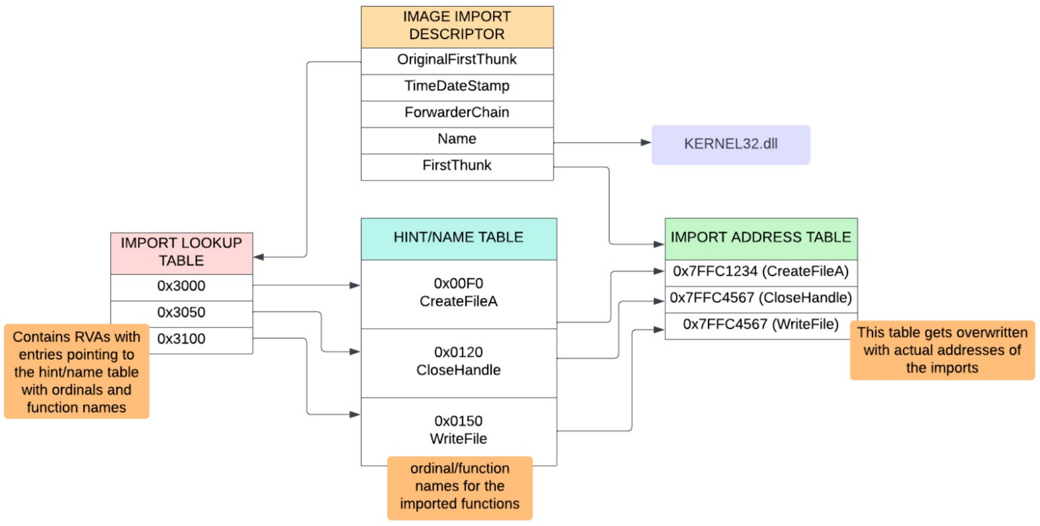


Figure 2: Import Address Table Data Flow Diagram

Table 2: IAT State Transitions

Phase	IAT Content	Access Type
Compile Time	RVA to Hint/Name	Read
Load Time (Before)	RVA to Hint/Name	Read/Write
Load Time (After)	Function Address	Read
Runtime	Function Address	Read (Execute)

4 Process Environment Block (PEB)

4.1 PEB as a Linked List Structure

The Process Environment Block contains critical information about loaded modules in a process. It utilizes a **doubly-linked list** data structure to maintain module information.

PEB_LDR_DATA Structure

Data Structure Type: Doubly-linked circular list
List Entry Structure:

```c
typedef struct _LDR_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;        // Flink, Blink pointers
    LIST_ENTRY InMemoryOrderLinks;
    LIST_ENTRY InInitializationOrderLinks;
    PVOID DllBase;                      // Module base address
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;         // DLL name for comparison
    // ... additional fields
} LDR_DATA_TABLE_ENTRY;
```

Traversal Algorithm:

- Start: PEB → Ldr → InLoadOrderModuleList
- Iterate: Follow Flink pointers until circular list returns to head
- Compare: BaseDllName at each node
- Return: DllBase when match found

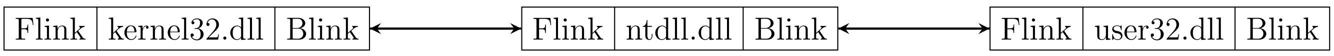


Figure 3: PEB Module List: Doubly-Linked List Structure

5 Function Resolution APIs

5.1 GetModuleHandle & GetProcAddress

`GetModuleHandle` is used to retrieve a handle to a given module. The module can be a DLL, for example `kernel32.dll`. The handle can then be passed to `GetProcAddress` to retrieve the address of an exported function from the DLL.

API as Data Structure Operations

GetModuleHandle:

- Operation: Linked list search
- Input: Module name (string)
- Process: Traverse PEB module list
- Output: Base address (pointer)

GetProcAddress:

- Operation: Hash table lookup (with hint) or binary search
- Input: Module handle + function name/ordinal
- Process: Parse Export Directory Table
- Output: Function address (pointer)

Algorithm 2 GetModuleHandle Implementation

```
Input: ModuleName (string)
Output: Module base address

PEB ← Get_Process_Environment_Block()
ModuleList ← PEB.Ldr.InLoadOrderModuleList
current ← ModuleList.Flink

while current $\neq$ ModuleList do
    entry ← CONTAINING_RECORD(current, LDR_DATA_TABLE_ENTRY)
    if Compare(entry.BaseDllName, ModuleName) == 0 then
        return entry.DllBase
    end if
    current ← current.Flink
end while

return NULL
```

6 Implementation

6.1 MessageBoxW Dynamic Resolution Example

The implementation integrates `GetModuleHandle` and `GetProcAddress` to dynamically resolve APIs.

6.1.1 Line 6: Module Handle Retrieval

```c
HMODULE user32Dll = GetModuleHandle(L"user32.dll");
```

This performs a linked-list traversal of the PEB module list to locate `user32.dll` and returns its base address.

6.1.2 Lines 9-15: Function Pointer Typedef

```c
typedef int (WINAPI* MessageBoxWFunc)(
    HWND hwnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType
);
MessageBoxWFunc pMessageBoxW = (MessageBoxWFunc)
   GetProcAddress(
    user32Dll,
    "MessageBoxW"
);
```

Function Pointer as Indirect Reference

Data Structure Concept: Function pointer (indirect reference)
Type Definition:

- Creates function pointer type matching API signature
- Ensures correct calling convention (WINAPI/stdcall)
- Enables type-safe function invocation

Type Casting:

- GetProcAddress returns generic FARPROC
- Cast to specific function pointer type
- Compiler can now validate parameters

6.1.3 Line 22: Indirect Function Call

Finally, we invoke the `pMessageBoxW` function through pointer indirection.

7 Strings Obfuscation

If we check the strings on the compiled program, it reveals API strings that were used. Static analysis can detect these patterns.

Figure 4: Strings output showing detectable API strings

7.1 XOR Encryption as Data Transformation

String Obfuscation Algorithm

Encryption (Compile Time):

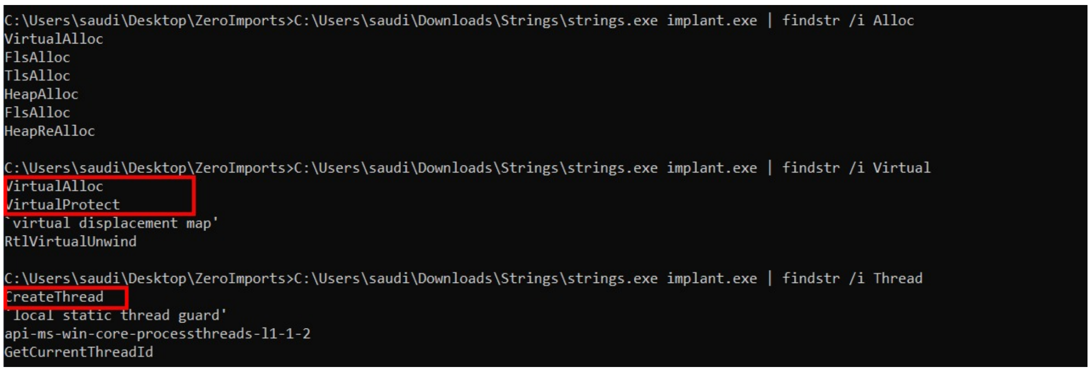
```
1: for each byte $b$ in plaintext_string do
2:     encrypted[i] ← b ⊕ key
3: end for
```

Decryption (Runtime):

```
1: for each byte $e$ in encrypted_string do
2:     decrypted[i] ← e ⊕ key
3: end for
```

Properties:

- Symmetric: Same key for encryption/decryption
- Self-inverse: $x \oplus k \oplus k = x$
- Fast: Single bitwise operation per byte



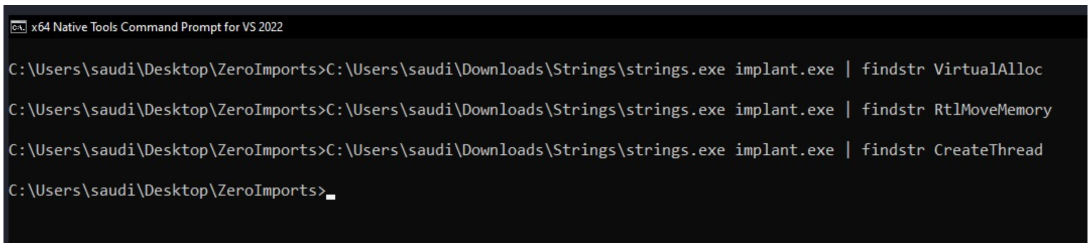
```
C:\Users\saudi\Desktop\ZeroImports>C:\Users\saudi\Downloads\Strings\strings.exe implant.exe | findstr /i Alloc
VirtualAlloc
FlsAlloc
TlsAlloc
HeapAlloc
FlsAlloc
HeapReAlloc

C:\Users\saudi\Desktop\ZeroImports>C:\Users\saudi\Downloads\Strings\strings.exe implant.exe | findstr /i Virtual
VirtualAlloc
VirtualProtect
`virtual displacement map'
RtlVirtualUnwind

C:\Users\saudi\Desktop\ZeroImports>C:\Users\saudi\Downloads\Strings\strings.exe implant.exe | findstr /i Thread
CreateThread
`local static thread guard'
api-ms-win-core-processthreads-l1-1-2
GetCurrentThreadId
```

Figure 5: CyberChef XOR encryption interface

8 Achieving Zero Imports

8.1 Manual Implementation Strategy

The code initially statically links to `GetModuleHandle` and `GetProcAddress`, leaving two imports visible in the IAT. The solution is to manually implement these functions by



x64 Native Tools Command Prompt for VS 2022

```
C:\Users\saudi\Desktop\ZeroImports>C:\Users\saudi\Downloads\Strings\strings.exe implant.exe | findstr VirtualAlloc

C:\Users\saudi\Desktop\ZeroImports>C:\Users\saudi\Downloads\Strings\strings.exe implant.exe | findstr RtlMoveMemory

C:\Users\saudi\Desktop\ZeroImports>C:\Users\saudi\Downloads\Strings\strings.exe implant.exe | findstr CreateThread

C:\Users\saudi\Desktop\ZeroImports>_
```

Figure 6: Strings output after XOR obfuscation - no detectable patterns

directly parsing PE data structures.

Table 3: Import Reduction Strategy

Approach	**IAT Entries**	**Detection Risk**
Standard Linking	50+ imports	High
Dynamic Resolution	2 imports	Medium
Manual Implementation	0 imports	Low

8.2 hlpGetModuleHandle Function

This function manually traverses the PEB data structure.

Manual Module Resolution Algorithm

Data Structures Accessed:

1. Process Environment Block (PEB)
2. PEB_LDR_DATA structure
3. LDR_DATA_TABLE_ENTRY linked list
4. UNICODE_STRING comparison

Algorithm Steps:

```
PEB ← Read from TEB (Thread Environment Block)
Ldr ← PEB.Ldr
ListHead ← Ldr.InLoadOrderModuleList

if ModuleName == NULL then
    return ListHead.First.DllBase                    ▷ Return calling module
end if

for each entry in ListHead do
    if String_Compare(entry.BaseDllName, ModuleName) == 0 then
        return entry.DllBase
    end if
end for
return NULL
```

8.3 hlpGetProcAddress Function

This function parses the PE Export Directory structure.

Manual Function Resolution Algorithm

Data Structures Parsed:

1. IMAGE_DOS_HEADER
2. IMAGE_NT_HEADERS
3. IMAGE_EXPORT_DIRECTORY
4. Export Address Table (array)
5. Export Name Table (array)
6. Export Ordinal Table (array)

Resolution Process:

```
DOS_Header ← ModuleBase
NT_Headers ← ModuleBase + DOS_Header.e_lfanew
Export_Dir ← NT_Headers.OptionalHeader.DataDirectory[EXPORT]

if ProcName is ordinal then
    ordinal ← ProcName & 0xFFFF
    return Export_Address_Table[ordinal]
else
    for i ← 0 to Export_Dir.NumberOfNames do
        name ← Export_Name_Table[i]
        if String_Compare(name, ProcName) == 0 then
            ordinal ← Export_Ordinal_Table[i]
            return Export_Address_Table[ordinal]
        end if
    end for
end if
return NULL
```

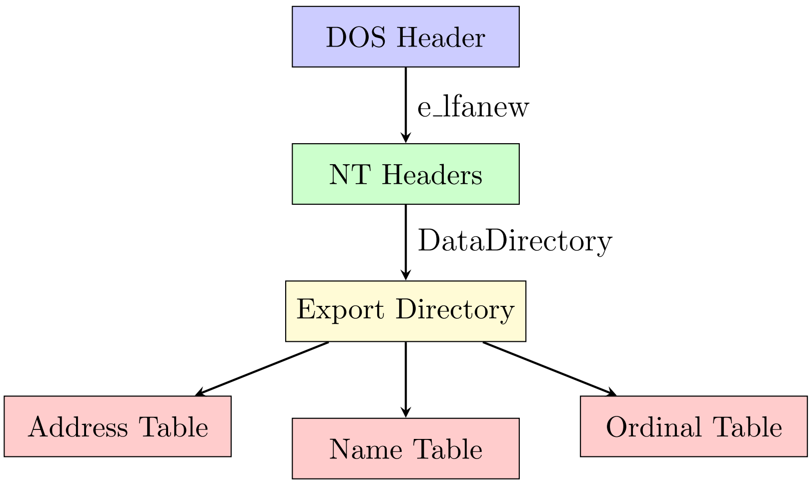


Figure 7: PE Export Directory Structure Navigation

9 Project Files

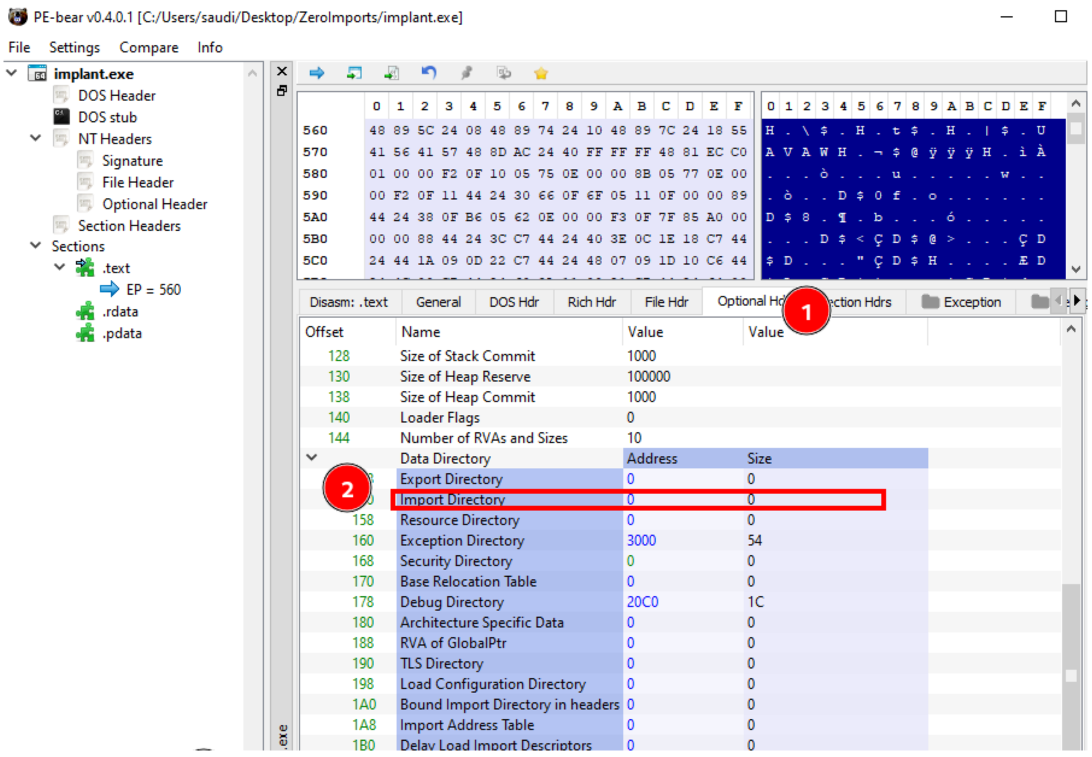
The complete project demonstrates:

- Custom implementations replacing Windows APIs
- Direct parsing of PE data structures
- Manual traversal of linked lists (PEB)
- Custom hash/search algorithms (export resolution)
- XOR-based string encryption arrays

10 Verification Results

10.1 IAT Analysis

Opening the compiled implant in PEBear reveals zero imports:



```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0 1 2 3 4 5 6 7 8 9 A B C D E F
560  48 89 5C 24 08 48 89 74 24 10 48 89 7C 24 18 55  H . \ $ . H . t $ . H . | $ . U
570  41 56 41 57 48 8D AC 24 40 FF FF FF 48 81 EC C0  A V A W H . ¬ $ @ ÿ ÿ ÿ H . ì À
580  01 00 00 F2 0F 10 05 75 0E 00 00 8B 05 77 0E 00  . . . ò . . . u . . . . . w . .
590  00 F2 0F 11 44 24 30 66 0F 6F 05 11 0F 00 00 89  . ò . . D $ 0 f . o . . . . . .
5A0  44 24 38 0F B6 05 62 0E 00 00 F3 0F 7F 85 A0 00  D $ 8 . ¶ . b . . . ó . . . . .
5B0  00 00 88 44 24 3C C7 44 24 40 3E 0C 1E 18 C7 44  . . . D $ < Ç D $ @ > . . . Ç D
5C0  24 44 1A 09 0D 22 C7 44 24 48 07 09 1D 10 C6 44  $ D . . . " Ç D $ H . . . . Æ D
```

Offset	Name	Value	Value
128	Size of Stack Commit	1000	
130	Size of Heap Reserve	100000	
138	Size of Heap Commit	1000	
140	Loader Flags	0	
144	Number of RVAs and Sizes	10	
	Data Directory	Address	Size
[illegible]	Export Directory	0	0
[illegible]	Import Directory	0	0
158	Resource Directory	0	0
160	Exception Directory	3000	54
168	Security Directory	0	0
170	Base Relocation Table	0	0
178	Debug Directory	20C0	1C
180	Architecture Specific Data	0	0
188	RVA of GlobalPtr	0	0
190	TLS Directory	0	0
198	Load Configuration Directory	0	0
1A0	Bound Import Directory in headers	0	0
1A8	Import Address Table	0	0
1B0	Delay Load Import Descriptors	0	0

Figure 8: PEBear showing zero imports - empty IAT

Table 4: Import Comparison - Before and After

Metric	Standard Implant	Zero Import Implant
IAT Entries	50+	0
Visible API Calls	All	None
String Signatures	Plaintext	XOR Encrypted
Static Detection	High	Low
Manual Parsing	No	Yes (PEB, Export Tables)

10.2 Data Structure Complexity Analysis

Table 5: Data Structure Operations - Time Complexity

Operation	Data Structure	Complexity
Module Lookup	PEB Linked List	O(n)
Function Lookup (Hint)	Export Table	O(1)
Function Lookup (Name)	Export Table	O(n)
String Decryption	Array	O(n)
IAT Resolution	Array	O(n)

11 Conclusion

This project demonstrates how understanding PE file data structures enables sophisticated evasion techniques. Key data structure concepts applied:

1. **Linked Lists**: PEB module traversal
2. **Arrays**: Import tables, export tables, string buffers
3. **Hash Tables**: Hint-based function lookups
4. **Pointers**: Indirect function calls, RVA resolution
5. **Binary Search**: Fallback function name resolution

By manually implementing these data structure operations, we achieve zero static imports while maintaining full API functionality.

12 References

Sektor7 Institute - Red Team Operator: Malware Development Intermediate Course
`https://institute.sektor7.net/rto-maldev-intermediate`

Microsoft Documentation - PE Format Specification
`https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`