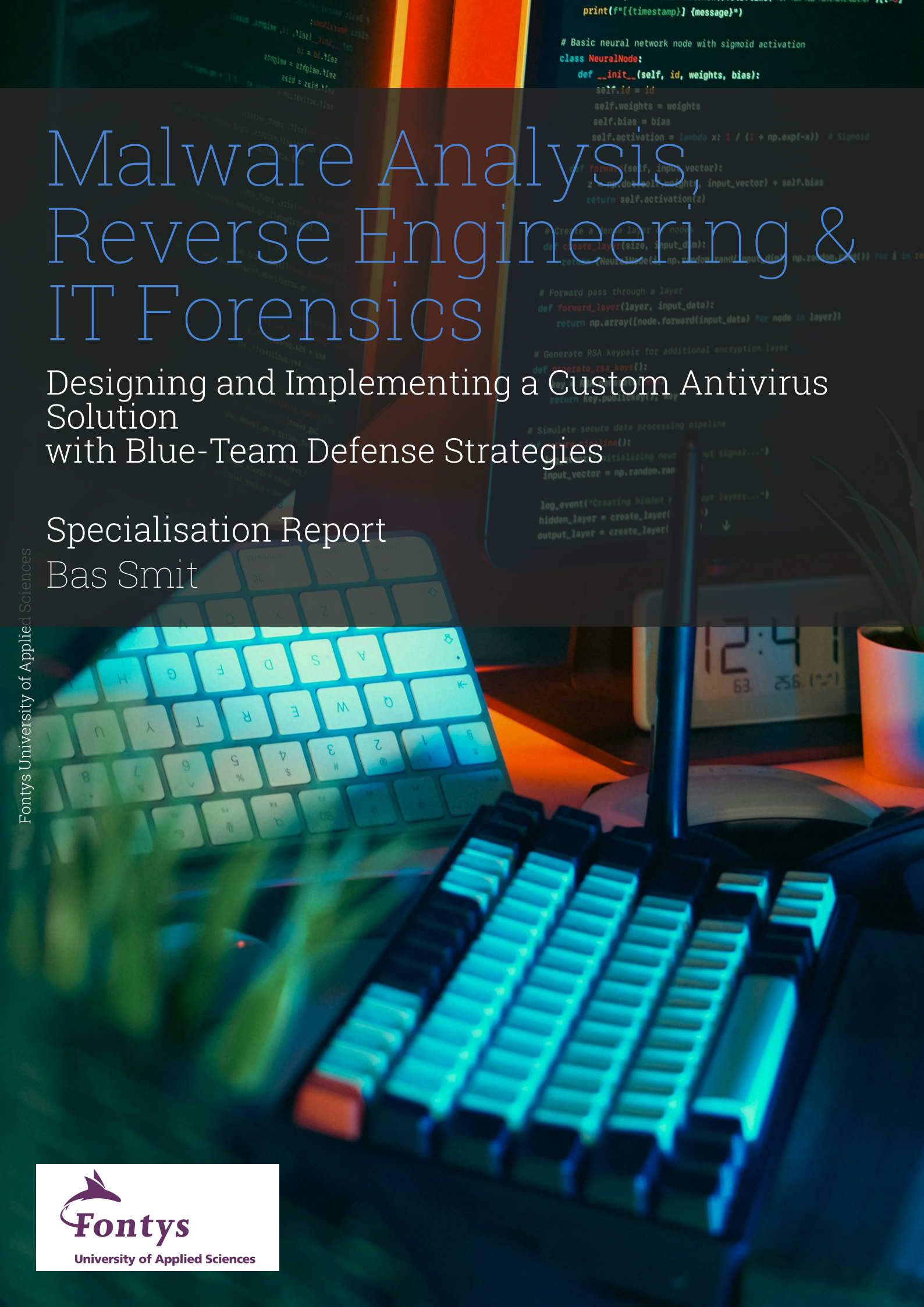
Malware Analysis, Reverse Engineering & IT Forensics

Designing and Implementing a Custom Antivirus Solution with Blue-Team Defense Strategies

Specialisation Report

Bas Smit

Fontys University of Applied Sciences



Fontys
University of Applied Sciences

Malware Analysis, Reverse Engineering & IT Forensics

Designing and Implementing a Custom Antivirus Solution with Blue-Team Defense Strategies

by

Bas Smit

Instructors:	Oosterhout, Mandy M. van Roijers, Stefan S.E.
Project Duration:	September 2024 - January 2025
Faculty:	Faculty of ICT, Eindhoven

Cover:	Malware analysis workspace featuring code decompilation, forensic tools, and cybersecurity monitoring interfaces
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

Fontys HOGESCHOOL ICT

Abstract

This research explores the design and implementation of a custom antivirus solution that combines malware analysis, reverse engineering, and digital forensics from a blue-team perspective. The primary objective is to demonstrate how signature-based and behavior-based detection techniques can be integrated to identify malicious files on Windows systems, while also providing insight into the technical mechanisms that underpin modern antivirus software.

The study examines multiple detection approaches including PE file structure analysis, API import inspection, entropy checks, and heuristic rule evaluation. Reverse engineering techniques are employed to understand malware behavior at a low level, revealing how obfuscation, packing, and evasion methods are used by threat actors. Additionally, forensic principles are applied to identify artifacts left by malware, such as volatile memory data, file system traces, event logs, and network activity patterns.

A functional prototype antivirus was developed in Python, incorporating both local signature databases and online validation through VirusTotal and MalwareBazaar. The prototype was tested against real-world malware samples as well as custom-created malicious executables. Results demonstrate that the combination of multiple detection methods significantly improves classification accuracy compared to single-method approaches.

The research confirms that fundamental antivirus concepts can be successfully implemented and validated through practical experimentation. While the prototype is not production-ready, it serves as proof-of-concept that core detection mechanisms are achievable with proper understanding of malware behavior, PE structures, and forensic analysis. Future improvements could include real-time monitoring capabilities, behavioral sandboxing, machine learning integration, and more sophisticated heuristic engines.

This work provides hands-on experience in malware development, detection strategy design, and defensive forensics, contributing valuable knowledge for incident response and cybersecurity defense operations.

Contents

1

Introduction

For this specialisation, I focused on malware analysis, reverse engineering, and IT forensics from a blue-team perspective. My main goal was to understand how malware actually works at a low level, develop skills to analyze and reverse engineer malicious software, and learn forensic techniques that help investigate security incidents.

I chose this specialisation for two main reasons. First, I wanted to build the knowledge and technical skills needed to defend against cyberattacks that keep getting more sophisticated. As attackers evolve their techniques, defenders need to understand how malware functions internally to create effective countermeasures. Second, the investigative side of cybersecurity really interested me—uncovering digital traces, reconstructing what happened during an attack, and gathering forensic evidence to support incident response.

Modern malware uses various tricks to avoid detection by traditional antivirus software. These include code obfuscation, runtime packing, polymorphic behavior, and sophisticated anti-debugging mechanisms. To counter these threats, I needed to understand not only how to detect malware signatures, but also how to analyze suspicious behavior patterns, reverse engineer packed executables, and extract forensic artifacts from compromised systems.

1.1. Leading Antivirus Solutions Feature Comparison

To understand what modern antivirus software actually does, I looked at the three market leaders: Microsoft Defender, Malwarebytes, and Bitdefender. Table 1.1 compares their core features to show what users expect from antivirus solutions today.

Table 1.1: Feature Comparison of Leading Antivirus Solutions

Feature	Microsoft Defender	Malwarebytes	Bitdefender
Signature-Based Detection	✓	✓	✓
Heuristic Analysis	✓	✓	✓
Behavioral Monitoring	✓	✓	✓
Real-Time Protection	✓	✓	✓
Cloud-Based Scanning	✓	✓	✓
Ransomware Protection	✓	✓	✓
Firewall Integration	✓	×	✓
Automatic Updates	✓	✓	✓
Quarantine Management	✓	✓	✓
Custom Scan Options	✓	✓	✓
VPN Service	×	Premium only	Premium only
Password Manager	×	×	Premium only
Cost	Free (Built-in)	Free/Premium	Free/Premium

Microsoft Defender comes free with Windows and includes all the basic detection features plus Windows Firewall integration. Malwarebytes focuses specifically on anti-malware and is known for catching stubborn threats that other antivirus software might miss. It has both a free scanner and a paid version with real-time protection. Bitdefender is considered enterprise-grade with consistently high detection rates in independent tests, though you need to pay for the full feature set.

What's important here is that all three use the same core detection methods: signature-based detection for known threats, heuristic analysis for suspicious patterns, and behavioral monitoring to catch malicious actions while they're happening. Since these are standard across all major antivirus solutions, I made sure to include them in my own project.

1.2. Antivirus Market Landscape

Understanding the current antivirus market provides context for this research and highlights the competitive landscape of malware detection solutions. According to Security.org's annual antivirus consumer report Security.org, 2025, the market shows distinct trends in user preferences and brand adoption over recent years.

1.2.1. Desktop Operating System Distribution

Before diving into specific antivirus solutions, I wanted to understand what operating systems people actually use. Table 1.2 shows the global desktop operating system market share, which makes it clear why Windows dominates the antivirus market.

Table 1.2: Desktop Operating System Market Share (Global)

Operating System	Market Share
Windows	66.4%
Unknown	16.0%
OS X	7.74%
macOS	4.75%
Linux	3.86%
Chrome OS	1.24%

Table 1.3: Desktop Operating System Market Share (Source: StatCounter StatCounter, 2025a)

Windows dominates with 66.4% market share—that's about two-thirds of all desktop computers worldwide. This makes Windows the primary target for both malware developers and antivirus solutions. When attackers can potentially compromise hundreds of millions of systems with a single

exploit, it makes sense that they focus on Windows. The antivirus industry follows the same logic, which is why most development effort goes into Windows protection.

For my project, I decided to focus on Windows for exactly these reasons. Building a Windows-focused antivirus means tackling the platform with the most malware diversity and the highest volume of threats. It's also where the work would be most relevant since the majority of users are on Windows. Plus, understanding Windows malware detection gives me skills that apply to protecting the largest number of potential users.

Table 1.4 presents the primary antivirus brands used by consumers across three years (2022-2025), revealing significant shifts in the competitive landscape.

Table 1.4: Antivirus Market Share by Brand (2022-2025)

Antivirus Brand	2022	2024	2025
Microsoft Defender	29%	28%	23%
McAfee	15%	17%	18%
Norton	13%	14%	13%
Malwarebytes	11%	9%	9%
Avast	10%	9%	8%
AVG	5%	7%	6%
Kaspersky	4%	3%	3%
Bitdefender	3%	4%	4%
Webroot	2%	3%	2%
XProtect (Mac)	2%	2%	2%
Other	7%	4%	4%

The data in Table 1.4 shows that Microsoft Defender is still the most popular antivirus from 2022 to 2025, though its market share dropped from 29% to 23%. This makes sense because Defender comes pre-installed and automatically enabled on every Windows system, so users get protection immediately without downloading anything extra. The 6 percentage point drop suggests more people are actively choosing third-party alternatives. McAfee benefited most from this shift, growing steadily from 15% to 18% and becoming the second most popular option. This tells me that users are becoming more aware of security options and are willing to pay for dedicated antivirus software even when a free, built-in option exists. The "Other" category also shrunk from 7% to 4%, which indicates people are sticking with well-known brands instead of niche products.

Table 1.5: Windows Version Market Share (Global Desktop) December 2025 (Source: StatCounter StatCounter, 2025b)

Windows Version	Market Share
Windows 10	50.68%
Windows 11	44.64%
Windows 7	3.92%
Windows XP	0.42%
Windows 8	0.17%
Windows 8.1	0.16%

Table 1.5 shows an interesting pattern in Windows version adoption as of December 2025. Even though Microsoft officially ended support and security updates for Windows 10 in October 2025, it still has the majority market share at 50.68%, just barely ahead of Windows 11's 44.64%. This surprised me at first, but it makes sense when you consider that Windows 11 has strict hardware requirements like TPM 2.0 and Secure Boot. A lot of perfectly functional computers can't upgrade even if users wanted to. Organizations are also hesitant to migrate critical systems, and many users simply prefer sticking with what they know. The gap between Windows 10 and 11 is closing though, probably because people are starting to worry about security without updates. What's really interesting is that legacy systems like

Windows 7 (3.92%), Windows XP (0.42%), and Windows 8/8.1 (0.33% combined) are still out there. This highlights how difficult it is to get everyone on the latest version, especially in enterprise environments with legacy applications or users with older hardware.

2

Theoretical and Practical Background

This chapter covers the theory and practical knowledge I needed before building my antivirus. Understanding what malware is, how it works, how antivirus programs detect it, and what reverse engineering techniques reveal about malicious files gave me the foundation to make smart design choices throughout the project.

2.1. Malware Types and Characteristics

To build effective detection, I first needed to understand what types of malware exist and how they behave. Different malware categories have distinct patterns that help identify them. Viruses attach themselves to other files and spread when those files are executed. Worms are standalone programs that replicate themselves across networks. Trojans pretend to be legitimate software but do malicious things in the background. Ransomware encrypts your files and demands payment to decrypt them. Spyware silently collects information about what you're doing. Rootkits hide their presence while maintaining unauthorized access to the system.

Each type creates different detection challenges. Viruses modify existing files, so you need integrity checking and signature matching. Worms generate unusual network traffic patterns. Trojans rely on tricking users and often look like normal applications, so you need heuristic analysis to spot suspicious behavior. Ransomware makes obvious changes to the file system and uses cryptographic APIs. Understanding these differences helped me create detection rules that look for type-specific indicators.

2.2. Detection Approaches

Antivirus programs use multiple detection methods because no single approach catches everything. Each method has strengths and weaknesses, so combining them gives the best coverage.

2.2.1. Signature-Based Detection

Signature-based detection compares files against a database of known malware. Signatures are usually cryptographic hashes (MD5, SHA1, SHA256) or specific byte patterns unique to particular malware families. This method is very accurate for known threats and rarely gives false positives. The problem is it completely fails against new malware, packed executables, or polymorphic variants that change their appearance while doing the same malicious things.

2.2.2. Heuristic Detection

Heuristic detection looks for suspicious characteristics without needing an exact match. It evaluates things like entropy levels (high entropy often means encryption or packing), suspicious API imports (functions like VirtualAlloc, WriteProcessMemory, and CreateRemoteThread are common in malware), weird PE structures (invalid headers or unusual section names), and behavioral indicators (modifying registry keys or injecting into processes). Heuristics can catch threats that aren't in the signature

database yet, but they also trigger more false positives and need careful tuning to work well.

2.2.3. Behavioral Detection

Behavioral detection watches what a program actually does when it runs. It monitors process behavior, file system changes, registry modifications, and network connections. This approach is great for catching zero-day threats and polymorphic malware that signatures can't identify. The downside is you have to actually execute the file in a controlled environment (sandbox), which uses more resources and takes more time.

2.3. Reverse Engineering Concepts

Reverse engineering is how you figure out what a program does when you don't have the source code. Since malware developers obviously don't share their code, reverse engineering becomes essential for understanding threats. Learning these techniques helped me understand what to look for when analyzing suspicious files.

2.3.1. PE File Structure

Windows uses the Portable Executable (PE) format for .exe files, DLLs, and other system binaries. Understanding how PE files are structured is fundamental to analyzing malware because it tells you a lot about what the program does before you even run it.

PE Components

A PE file has several important parts:

DOS Header and Stub: These are legacy components from ancient Windows versions that maintain backward compatibility with DOS. Every valid executable still needs them even though they're mostly useless now. The DOS stub usually just displays "This program cannot be run in DOS mode" if someone tries to run it in a DOS environment.

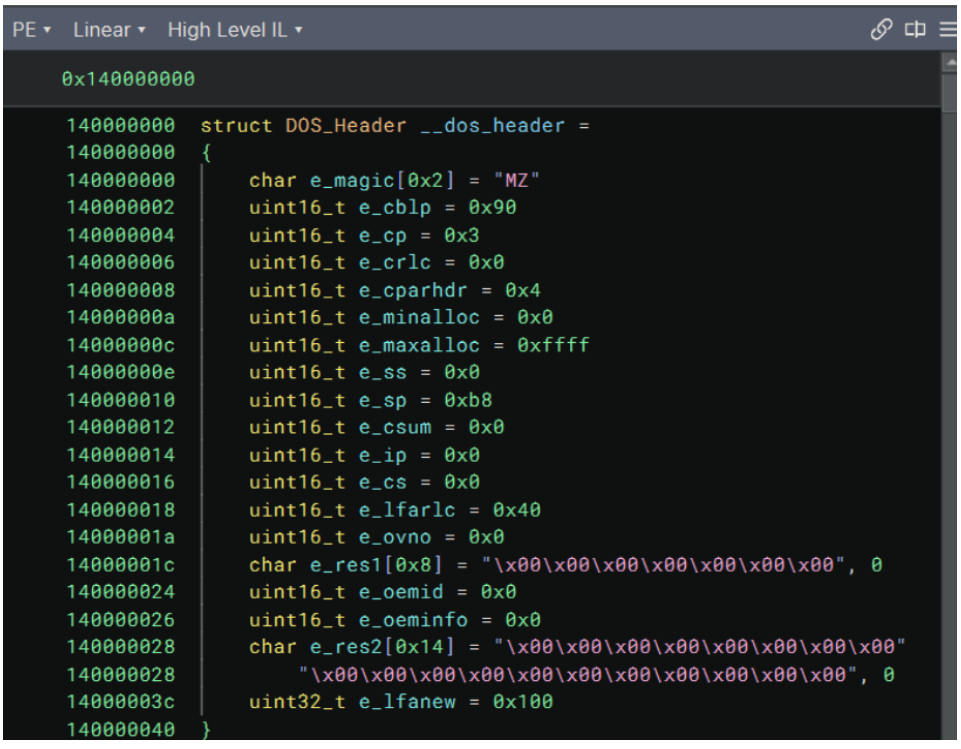


Figure 2.1: DOS Header structure showing legacy compatibility components

NT Headers: This part has metadata like what kind of file it is (32-bit or 64-bit), where execution starts (the entry point), and whether it's a console or GUI application. It also includes the image base address,

section alignment, and what Windows version it needs.

```
140000100  struct COFF_Header __coff_header =
140000100  {
140000100      char magic[0x4] = "PE\x00", 0
140000104      enum coff_machine_1 machine = IMAGE_FILE_MACHINE_AMD64
140000106      uint16_t numberOfSections = 0x7
140000108      uint32_t timeDateStamp = 0x6907c2c4
14000010c      uint32_t pointerToSymbolTable = 0x0
140000110      uint32_t numberOfSymbols = 0x0
140000114      uint16_t sizeOfOptionalHeader = 0xf0
140000116      enum coff_characteristics_1 characteristics = IMAGE_FILE
140000118  }
140000118  struct PE64_Optional_Header __pe64_optional_header =
140000118  {
140000118      enum pe_magic_1 magic = PE_64BIT
14000011a      uint8_t majorLinkerVersion = 0xe
14000011b      uint8_t minorLinkerVersion = 0x2c
14000011c      uint32_t sizeOfCode = 0x2be00
140000120      uint32_t sizeOfInitializedData = 0x26c00
140000124      uint32_t sizeOfUninitializedData = 0x0
140000128      uint32_t addressOfEntryPoint = 0xda30
14000012c      uint32_t baseOfCode = 0x1000
140000130      uint64_t imageBase = 0x140000000
140000138      uint32_t sectionAlignment = 0x1000
14000013c      uint32_t fileAlignment = 0x200
140000140      uint16_t majorOperatingSystemVersion = 0x6
140000142      uint16_t minorOperatingSystemVersion = 0x0
140000144      uint16_t majorImageVersion = 0x0
140000146      uint16_t minorImageVersion = 0x0
140000148      uint16_t majorSubsystemVersion = 0x6
14000014a      uint16_t minorSubsystemVersion = 0x0
14000014c      uint32_t win32VersionValue = 0x0
140000150      uint32_t sizeOfImage = 0x5c000
140000154      uint32_t sizeOfHeaders = 0x400
140000158      uint32_t checkSum = 0x81ae08
14000015c      enum pe_subsystem_1 subsystem = IMAGE_SUBSYSTEM_WINDOWS_
14000015e      enum pe_dll_characteristics_1 dllCharacteristics = IMAGE
140000160      uint64_t sizeOfStackReserve = 0x1e8480
140000168      uint64_t sizeOfStackCommit = 0x1000
140000170      uint64_t sizeOfHeapReserve = 0x100000
140000178      uint64_t sizeOfHeapCommit = 0x1000
140000180      uint32_t loaderFlags = 0x0
140000184      uint32_t numberOfRvaAndSizes = 0x10
```

Figure 2.2: NT Header containing critical executable metadata

Section Table: This lists the different sections of the executable. The `.text` section has the actual code, `.data` stores variables, `.rdata` holds read-only data like imports and constants, and `.rsrc` contains resources like icons and version strings.

```
Type: PE
Platform: windows-x86_64
Architecture: x86_64

Libraries:
  USER32.dll
  COMCTL32.dll
  KERNEL32.dll
  ADVAPI32.dll
  GDI32.dll

Compiler(s) Used:
  Imported Functions (159 objects)
  VS2022 17.5.4 build 32217 and VS2010 v10.0 SP1 build 40219 (216 obj
  VS2022 17.5.4 build 32217 and VS2010 v10.0 SP1 build 40219 (69 obje
  VS2022 17.5.4 build 32217 and VS2010 v10.0 SP1 build 40219 (28 obje

Segments:
r--  0x140000000-0x140000400
r-x  0x140001000-0x14002cd80  {Code}
r--  0x14002d000-0x140040968  {Data}
rw-  0x140041000-0x140041e00  {Data}
rw-  0x140041e00-0x1400460b0  {Data}
r--  0x140047000-0x1400493dc  {Data}
rw-  0x14004a000-0x14004a100  {Data}
r--  0x14004b000-0x14005a41c  {Data}
r--  0x14005b000-0x14005b774  {Data}
---  0x14005b780-0x14005bc10
---  0x14005bc10-0x14005bc40

Sections:
0x140001000-0x14002cd80  .text  {Code}
0x14002d000-0x140040968  .rdata  {Read-only data}
0x140041000-0x1400460b0  .data  {Writable data}
0x140047000-0x1400493dc  .pdata  {Read-only data}
0x14004a000-0x14004a100  .fptable  {Writable data}
0x14004b000-0x14005a41c  .rsrc  {Read-only data}
0x14005b000-0x14005b774  .reloc  {Read-only data}
0x14005b780-0x14005bc10  .extern  {External}
0x14005bc10-0x14005bc40  .synthetic_builtins  {External}
```

Figure 2.3: Section Table displaying executable sections and their properties

Import Address Table (IAT): This shows which Windows DLL functions the program needs. Malware analysts pay close attention to the IAT because certain functions are commonly used in malicious behavior. Functions like `VirtualAlloc`, `WriteProcessMemory`, and `CreateRemoteThread` are often signs of process injection.

Export Table: This lists functions the binary makes available to other programs (mainly relevant for DLLs). Malicious DLLs sometimes export functions with misleading names.

Resource Section: This contains non-executable stuff like icons, dialogs, and version strings. Malware sometimes hides encrypted payloads or configuration data in the resource section to avoid detection.

Anomaly Indicators

Malware authors often mess with PE structures to avoid detection or make analysis harder. Common red flags include:

- Weird section names that don't follow normal conventions (like `.text`, `.data`)

- Invalid or zeroed timestamps that look deliberately obfuscated
- Section sizes that don't match up (virtual vs. raw size)
- High entropy values suggesting compression or encryption
- Suspicious imports or dynamic API resolution (using `LoadLibrary` and `GetProcAddress`)
- Entry points that are outside the `.text` section where they should be

Understanding Entropy: Entropy measures how random or unpredictable data in a file is. It's calculated as a value between 0 and 8, where 0 means completely uniform data (like a file full of zeros) and 8 means completely random data. Normal executable code usually has entropy between 5 and 6 because code has patterns and structure. But encrypted or compressed data looks random and has entropy close to 7 or 8.

For example, a simple text string like "AAAAAAAAAA" has very low entropy because it's completely predictable. An encrypted string like "x3$mP9kLq2" has high entropy because each character seems random. Malware authors often encrypt or compress their payloads to hide them from signature-based detection. When a PE section shows entropy above 7, that's a strong sign the content is packed, encrypted, or otherwise hidden. My antivirus uses entropy calculation to spot suspicious sections that might contain hidden malicious code.

Detection systems can flag files with these weird characteristics as potentially malicious even before running them.

Reversing Compiled Python Executables

More and more malware is written in Python and compiled into executables. Python-based malware often uses PyInstaller to package everything into a single .exe file. Understanding how to reverse engineer these shows why obfuscation is so important.

PyInstaller bundles Python scripts with the interpreter and all dependencies into a standalone executable. Using a tool called `pyinstxtractor`, you can extract the PyInstaller archive and reveal compiled Python bytecode (`.pyc` files).

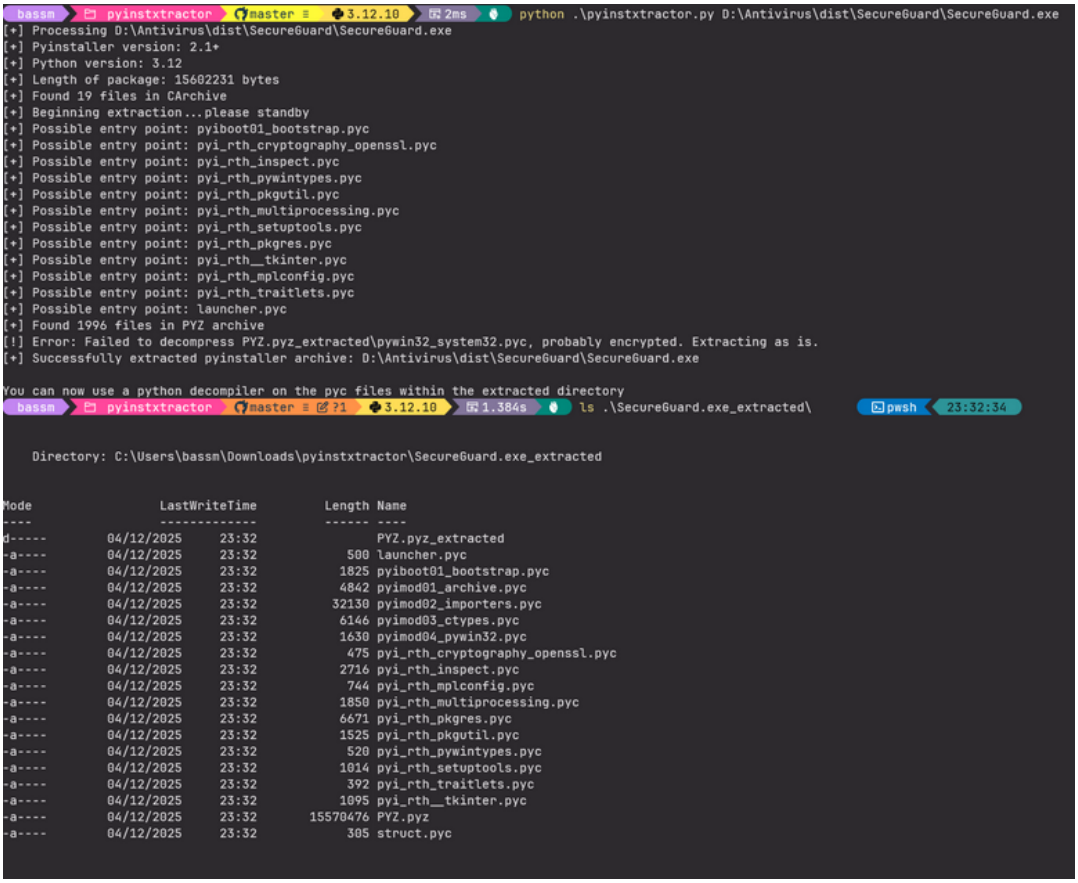


Figure 2.4: Extracting PyInstaller executable to reveal compiled Python bytecode files

These bytecode files are mostly unreadable, but you can decompile them using tools like `pycdc` or online services like `pylingual.io` to get back the original source code.

Figure 2.5: Compiled Python bytecode (.pyc) files before decompilation

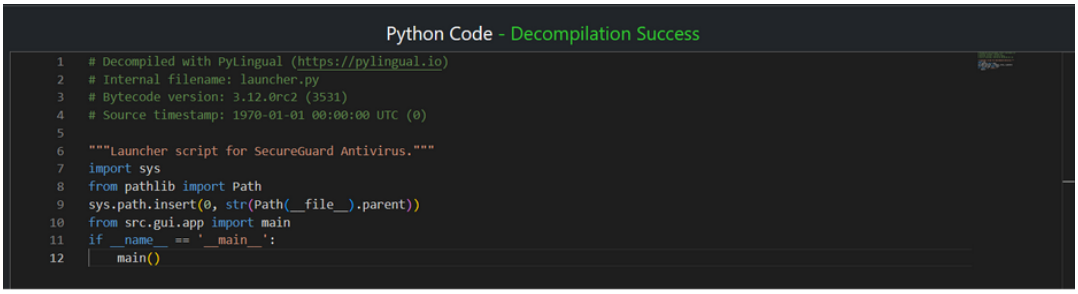


Figure 2.6: Decompiled Python source code recovered from bytecode

This shows why obfuscation and protection mechanisms matter so much in security applications. If you can just decompile something and get the original code back, all your secrets are exposed. That's why malware authors use code obfuscation, packing, and anti-debugging techniques to prevent trivial decompilation.

2.3.2. Disassembly

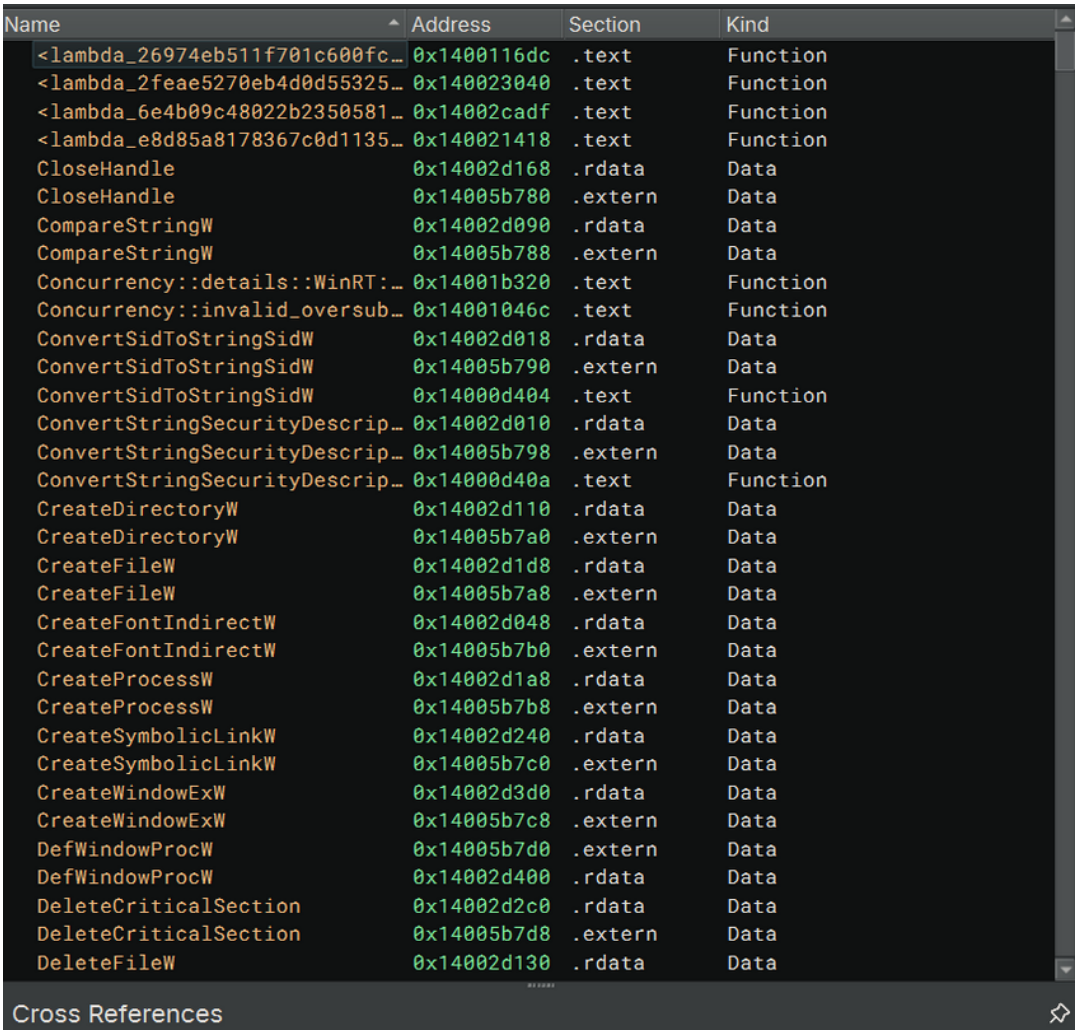
Disassembly converts machine code into human-readable assembly language, which lets you see what a program does at a very low level. Tools like Ghidra, IDA Pro, Binary Ninja, Radare2, and objdump help with this when you're investigating malware.

Key Disassembly Patterns

Certain instruction sequences and API calls are tell-tale signs of malicious behavior:

- **Memory manipulation:** Calls to `VirtualAlloc`, `VirtualProtect`, or `WriteProcessMemory` suggest shellcode injection or process hollowing
- **Process injection:** `CreateRemoteThread`, `NtCreateThreadEx`, or `QueueUserAPC` indicate code injection into other processes
- **Keylogging loops:** Tight loops calling `GetAsyncKeyState` or `SetWindowsHookEx`
- **Encryption routines:** XOR loops, AES implementations, or cryptographic API usage
- **Network communication:** Socket creation, HTTP requests, or DNS queries to suspicious domains

When you disassemble packed or obfuscated executables, they look chaotic with nonsensical instructions, excessive jumps, and deliberately confusing control flow. That chaos itself is an indicator of malicious intent.



Name	Address	Section	Kind
<lambda_26974eb511f701c600fc…	0x1400116dc	.text	Function
<lambda_2feae5270eb4d0d55325…	0x140023040	.text	Function
<lambda_6e4b09c48022b2350581…	0x14002cadf	.text	Function
<lambda_e8d85a8178367c0d1135…	0x140021418	.text	Function
CloseHandle	0x14002d168	.rdata	Data
CloseHandle	0x14005b780	.extern	Data
CompareStringW	0x14002d090	.rdata	Data
CompareStringW	0x14005b788	.extern	Data
Concurrency::details::WinRT:…	0x14001b320	.text	Function
Concurrency::invalid_oversub…	0x14001046c	.text	Function
ConvertSidToStringSidW	0x14002d018	.rdata	Data
ConvertSidToStringSidW	0x14005b790	.extern	Data
ConvertSidToStringSidW	0x14000d404	.text	Function
ConvertStringSecurityDescrip…	0x14002d010	.rdata	Data
ConvertStringSecurityDescrip…	0x14005b798	.extern	Data
ConvertStringSecurityDescrip…	0x14000d40a	.text	Function
CreateDirectoryW	0x14002d110	.rdata	Data
CreateDirectoryW	0x14005b7a0	.extern	Data
CreateFileW	0x14002d1d8	.rdata	Data
CreateFileW	0x14005b7a8	.extern	Data
CreateFontIndirectW	0x14002d048	.rdata	Data
CreateFontIndirectW	0x14005b7b0	.extern	Data
CreateProcessW	0x14002d1a8	.rdata	Data
CreateProcessW	0x14005b7b8	.extern	Data
CreateSymbolicLinkW	0x14002d240	.rdata	Data
CreateSymbolicLinkW	0x14005b7c0	.extern	Data
CreateWindowExW	0x14002d3d0	.rdata	Data
CreateWindowExW	0x14005b7c8	.extern	Data
DefWindowProcW	0x14005b7d0	.extern	Data
DefWindowProcW	0x14002d400	.rdata	Data
DeleteCriticalSection	0x14002d2c0	.rdata	Data
DeleteCriticalSection	0x14005b7d8	.extern	Data
DeleteFileW	0x14002d130	.rdata	Data

Cross References

Figure 2.7: Disassembled code showing instruction sequences and API calls

2.3.3. Obfuscation Techniques

Malware employs various obfuscation methods to complicate analysis and evade detection:

Packing: Executables are compressed or encrypted, with unpacking code that reveals the actual payload only at runtime. Popular packers include UPX, ASPack, and Themida. Detection systems identify packed executables through high entropy sections and stub patterns.

Encryption: Payloads, strings, and configuration data are encrypted to prevent static analysis. Decryption occurs dynamically during execution.

Dynamic API Resolution: Instead of importing functions directly, malware uses `LoadLibrary` and `GetProcAddress` to resolve APIs at runtime, concealing intentions from IAT inspection.

Control Flow Obfuscation: Code includes junk instructions, opaque predicates (conditions always evaluating to the same result), and convoluted jump patterns that complicate disassembly and static analysis.

String Obfuscation: URLs, file paths, registry keys, and C2 addresses are encoded or encrypted, preventing simple string searches from revealing malicious indicators.

Understanding these techniques informs heuristic detection rules that identify obfuscation indicators as suspicious characteristics.

2.3.4. Dynamic Debugging

Dynamic debugging observes malware behavior during execution, revealing actions that static analysis cannot detect. Debuggers like x64dbg, WinDbg, and OllyDbg enable instruction-level execution control, memory inspection, and breakpoint setting.

Observable Behaviors

Dynamic analysis reveals:

- Registry modifications for persistence (Run keys, service creation)
- File system operations (dropped files, encrypted documents)
- Process spawning and injection attempts
- Network connections and C2 communication
- Anti-debugging checks and evasion techniques
- Unpacking routines revealing hidden payloads

Tools like Process Monitor, Procmon, Wireshark, and automated sandboxes (Cuckoo, Any.Run) log system interactions comprehensively. While full dynamic analysis exceeds typical antivirus scope, understanding these principles shapes behavioral detection logic.

2.4. Forensic Principles

Digital forensics examines system artifacts to reconstruct malware activity and identify compromise indicators. These principles inform antivirus detection strategies.

2.4.1. Volatile Data Collection

Volatile data resides in RAM and disappears upon system shutdown. Memory contains decrypted malware payloads, active network connections, injected code, running processes, unpacked binaries, and cached credentials. Tools like Volatility analyze memory dumps (`.raw` files), extracting process lists, network connections, loaded modules, and injected code.

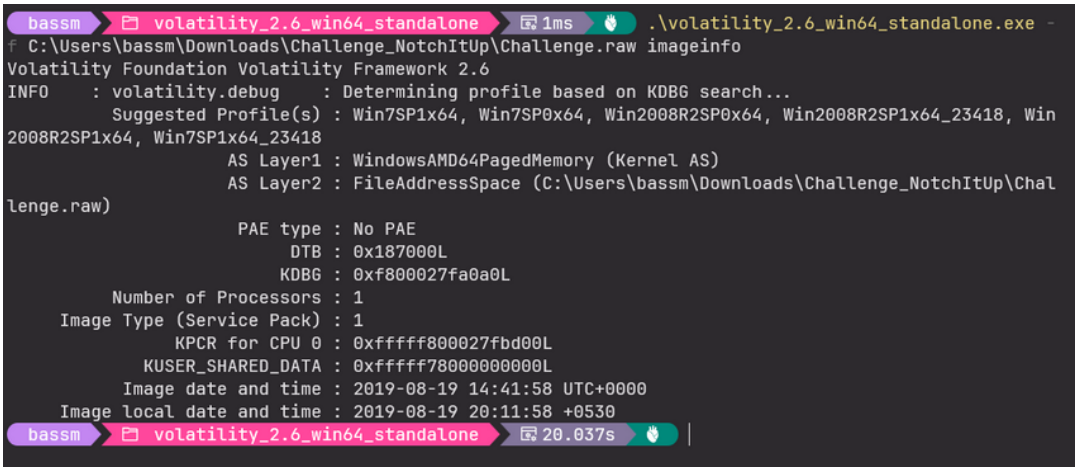


Figure 2.8: Volatility framework analyzing memory dump files

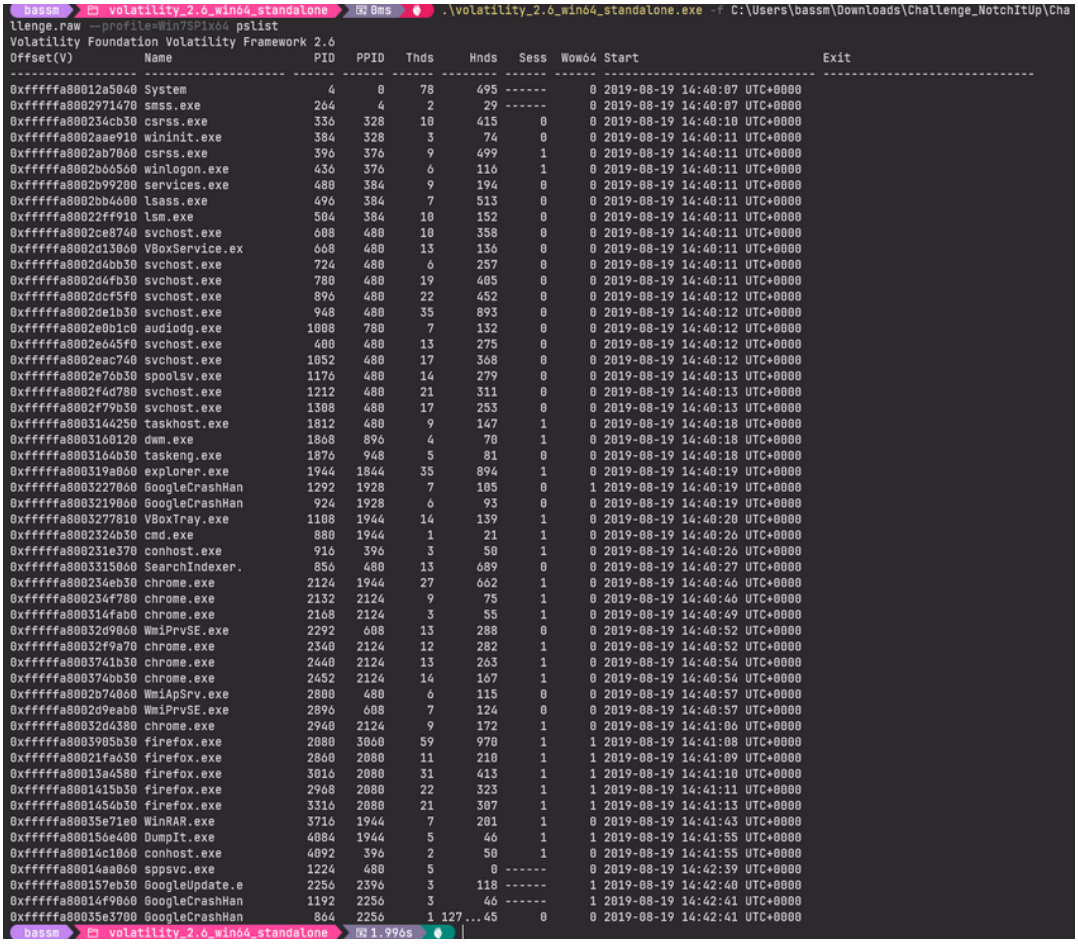


Figure 2.9: Extracting process information from RAM dump using Volatility

Memory forensics reveals information unavailable through static analysis, explaining why behavioral detection and sandboxing complement signature-based approaches.

2.4.2. File System Artifacts

Malware leaves persistent traces on disk:

- **Temporary files:** Created during execution, often in `%TEMP%` or `%APPDATA%`
- **Prefetch files:** Windows generates `.pf` files in `C:\Windows\Prefetch` when executables run, confirming program execution
- **Registry modifications:** Persistence mechanisms in `HKLM\Software\Microsoft\Windows\CurrentVersion\Run` or service entries
- **Dropped payloads:** Additional malware components, configuration files, or exfiltrated data
- **Modified system files:** Infected or replaced legitimate executables

Antivirus systems leverage this knowledge to identify suspicious file system patterns supporting malicious activity detection.

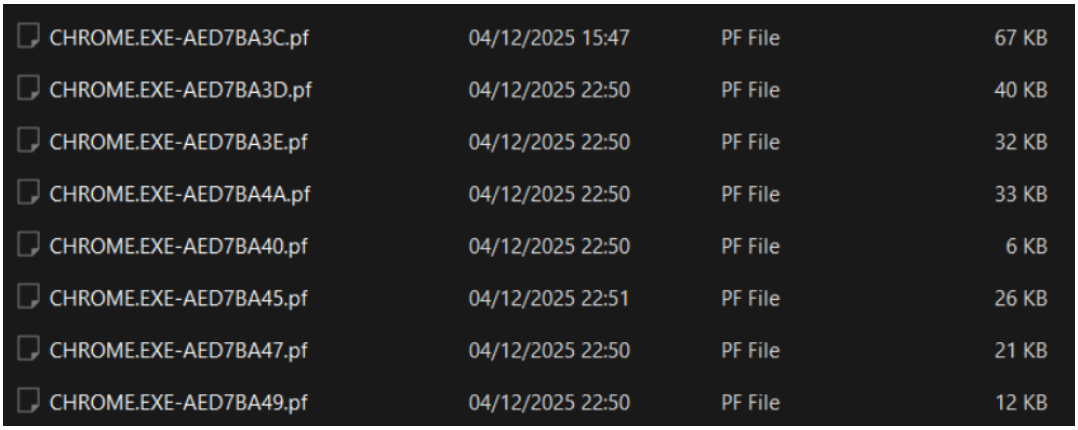


Figure 2.10: Windows Prefetch folder containing execution artifacts

2.4.3. Event Logs

Windows Event Viewer records system activity in detailed logs exposing malware behavior:

- Executable launch events (Security logs, Sysmon)
- Service crashes or unexpected startups
- Failed authentication attempts (brute force indicators)
- UAC elevation prompts (privilege escalation attempts)
- PowerShell script execution (common attack vector)

While comprehensive log parsing exceeds typical antivirus scope, understanding which events correlate with malicious actions informs heuristic detection rules.

2.4.4. Network Traces

Modern malware frequently communicates with command-and-control (C2) servers for instructions or data exfiltration. Network indicators include:

- Connections to unknown or suspicious domains
- Traffic over unusual ports (non-standard HTTP/HTTPS ports)
- DNS queries for algorithmically generated domains (DGA)
- Encrypted traffic to non-standard destinations
- Large data uploads to external servers

Network behavioral awareness enables antivirus systems to detect infections before visible system damage occurs.

2.5. Antivirus Architecture Theory

Antivirus software comprises multiple components working together to detect, analyze, and contain threats. Understanding architectural principles is essential for designing functional detection systems.

2.5.1. Scanning Engine

The scanning engine forms the antivirus core, responsible for file inspection and threat evaluation. It implements multiple detection methods including hash-based lookups, YARA rule matching, IAT inspection, and PE structure analysis. Efficient engines employ caching, incremental scanning, and optimized hashing to minimize resource consumption while maintaining accuracy. Engine performance directly impacts system responsiveness and detection effectiveness.

2.5.2. File Monitoring

Real-time monitoring extends protection beyond scheduled scans by observing system activity continuously. Monitoring components intercept file system operations (writes, executions), registry modifications, and process creation events. Windows provides APIs like FileSystemWatcher for user-space monitoring, while kernel-level protection requires driver development. Early detection prevents malware execution before damage occurs.

2.5.3. Quarantine System

When threats are detected, quarantine systems isolate infected files in controlled environments preventing execution or harm. Quarantined files are encrypted or renamed to block accidental access. The system records metadata including detection reason, original file path, hash values, and timestamps, enabling restoration if false positives occur or facilitating further investigation. Reliable quarantine protects users without data loss.

2.5.4. Privilege Management

Antivirus systems balance protection capabilities with security risks. User-space file scanning requires minimal permissions, but advanced threat detection often demands elevated access for system-wide process monitoring, protected registry access, and memory scanning. However, excessive privileges create attack surface—if the antivirus is compromised, attackers gain system-level access. Modern architectures isolate high-privilege tasks in protected services while maintaining user-facing components with limited permissions, implementing the principle of least privilege.

2.5.5. Architectural Integration

These components form an integrated system: the scanning engine identifies threats using multiple detection methods, file monitoring provides real-time protection, the quarantine system safely contains malicious files, and proper privilege management ensures secure operation. Understanding these architectural principles justifies design decisions in the custom antivirus implementation detailed in subsequent chapters.

3

Research Questions

This chapter defines the main research question and sub-questions that guide this specialisation project, along with the research approach based on the DOT (Development Oriented Triangulation) framework.

3.1. Research Goal

The goal of this research is to design, implement, and evaluate a custom antivirus solution that demonstrates understanding of malware behavior, detection methods, and incident analysis techniques. This research also aims to deepen expertise in reverse engineering, static/dynamic malware analysis, and forensic investigation through practical implementation and validation.

3.2. Main Research Question

"What methods can be used to design an antivirus system capable of detecting and analyzing malicious behavior on Windows systems?"

This question addresses the core challenge of building a functional antivirus prototype that can effectively identify malware through multiple detection techniques while maintaining practical usability and accuracy.

3.3. Sub-Questions

The main research question is decomposed into five focused sub-questions that each address a specific aspect of the antivirus development process:

3.3.1. Sub-Question 1: Detection Techniques

How can signature-based and behavior-based detection techniques be applied in a custom antivirus?

This question investigates the implementation of fundamental detection methods. Signature-based detection relies on known malware patterns and hash databases, while behavior-based detection identifies suspicious activities through heuristic rules and anomaly detection.

3.3.2. Sub-Question 2: Analysis Methods

What static and dynamic analysis techniques can be incorporated to inspect potentially malicious files?

This question explores the technical methods used to examine executable files. Static analysis involves inspecting file structures, headers, imports, and code without execution, while dynamic analysis observes runtime behavior in controlled environments.

3.3.3. Sub-Question 3: Reverse Engineering

How can reverse engineering support the accuracy and reliability of malware classification?

This question examines how disassembly, decompilation, and structural analysis of PE files can reveal malicious intent, obfuscation techniques, and behavioral characteristics that improve detection logic.

3.3.4. Sub-Question 4: Forensic Artifacts

What forensic artifacts can be extracted to support incident response?

This question investigates which traces malware leaves on systems, including volatile memory data, file system modifications, registry changes, event logs, and network activity patterns that aid in post-incident investigation.

3.3.5. Sub-Question 5: Validation and Effectiveness

How effective is the implemented antivirus in detecting real-world malware samples compared to baseline methods?

This question addresses validation through testing against known malware databases (MalwareBazaar) and custom-created samples, measuring detection rates and comparing results with established detection approaches.

3.4. Research Approach Using DOT Framework

The DOT (Development Oriented Triangulation) framework structures this research across three dimensions: the domains (what), trade-offs (why), and strategies (how). This framework ensures comprehensive coverage of both theoretical knowledge and practical implementation.

SQ	Question	Strategy	Methods
1	How can signature-based and behavior-based detection techniques be applied?	**Library** **Lab**	Literature review of detection methods Implementation and testing with malware samples
2	What static and dynamic analysis techniques can be incorporated?	**Library** **Lab**	Study of PE structure, disassembly techniques Practical analysis of malware samples
3	How can reverse engineering support malware classification?	**Library** **Lab**	RE theory and methodologies Hands-on reverse engineering experiments
4	What forensic artifacts can be extracted?	**Library**	Forensic methodologies study Literature on artifact types and extraction
5	How effective is the implemented antivirus?	**Lab**	Controlled testing environment Comparison with VirusTotal/MalwareBazaar Detection rate measurement

Table 3.1: DOT Framework strategies applied to research questions

4

Methodology

This chapter explains how I built and tested the custom antivirus solution. I'll describe the development process, the tools I used, how the system works, and how I validated that everything functions correctly.

Note on AI Usage: Throughout this project, I used AI tools extensively to assist with development and writing. GitHub Copilot and ChatGPT helped me generate code, debug issues, create YARA rules, and implement complex features like behavioral monitoring and the GUI. Additionally, AI was used to reformulate and improve the text throughout this report to ensure clarity and proper academic presentation of the technical concepts.

4.1. Development Setup

For this project, I developed a Windows-based antivirus using Python because Python offers excellent libraries for file analysis and system monitoring. I chose to work on Windows 11 since that's where most malware targets operate, and it made sense to build the solution on the actual platform it would protect. The entire project was coded in Visual Studio Code, and I used Git to track all my changes throughout development.

The implementation uses several important Python libraries that handle specific tasks. The `requests` library connects to the VirusTotal API VirusTotal, 2025a for cloud-based threat checking. The `pefile` library parses Windows executable files so I can inspect their structure. For pattern matching, I integrated `yara-python` VirusTotal, 2025b which runs YARA rules against suspicious files. The `psutil` library monitors running processes and system behavior. For storing malware signatures locally, I used Python's built-in `sqlite3` database. Finally, `tkinter` provided the framework for building the graphical user interface.

4.2. How the Antivirus Works

The antivirus is built from six main modules that each handle specific tasks. I designed it this way so each component can be developed and tested independently, and they all work together to provide layered protection.

The `config.py` module stores all the settings in one place. This includes the VirusTotal API key, file size limits (I set a maximum of 100 MB for scanning), detection thresholds for heuristic and behavioral analysis, and paths to the database and YARA rules. Having everything in a config file makes it easy to adjust settings without changing the actual detection code.

The `signature_db.py` module manages the SQLite database where I store known malware signatures. The database has two tables: one for full file hashes (SHA256, SHA1, or MD5) and another for partial hashes. Each signature includes the malware name, when it was added, and where it came from (either manually added, imported from VirusTotal, or loaded from a file). This local database is important because it lets the antivirus quickly check files without constantly querying VirusTotal, which has rate limits.

The `scanner.py` module is the main coordinator that runs files through all the detection checks. When you scan a file, it first validates that the file exists and isn't too large. Then it calculates the SHA256 hash and checks the local database. If there's no match, it calculates partial hashes from different parts of the file to catch malware that has been padded with junk data. Next, if heuristic analysis is enabled, it inspects the file structure for suspicious characteristics. If the file still isn't identified and VirusTotal is enabled, it queries their API. Finally, if behavioral analysis is requested, it can execute the file in a monitored environment to watch what it does, though I only use this for safe test files.

The `heuristic_engine.py` module performs static analysis on Windows PE files. It examines the file structure looking for red flags like packed sections, unusually high entropy (suggesting encryption), suspicious API imports like VirtualAlloc or WriteProcessMemory, and other indicators that professionals look for when analyzing malware. I also integrated YARA rules that pattern-match against known malware behaviors. The engine assigns points for each suspicious finding, and if the total score exceeds 60 points, it flags the file as potentially malicious.

The `behavioral_engine.py` module is designed to monitor what a file actually does when executed. It tracks things like network connections, child processes being spawned, memory usage, and suspicious patterns in process memory. However, I only implemented this for testing benign programs because running actual malware is too risky without proper isolation. The behavioral engine would execute a file, monitor it for 30 seconds, and score its activities based on how suspicious they appear.

The `gui.py` module provides a modern graphical interface so users can easily scan files and folders. The GUI uses a dark theme and shows real-time results in a color-coded table (red for threats, green for clean files). Users can enable or disable different detection methods, see detailed information about each scan result, and track statistics about how many files were scanned and how many threats were found. I built it with threading so the interface doesn't freeze during long scans.

4.3. Detection Methods

The antivirus implements multiple detection techniques that work together. I'll explain how each one works and why I included it.

Signature-based detection is the foundation. When a file is scanned, the system calculates its SHA256 hash and checks if it matches any known malware signatures in the database. This method is extremely accurate for known threats and very fast since it's just a database lookup. When VirusTotal VirusTotal, 2025a identifies something as malicious, I automatically add that signature to the local database so future scans of the same file don't need to query the API again.

Partial hashing addresses a common evasion technique where attackers append random data to a malware file to change its hash. My implementation samples five different parts of the file: the first 8 KB, positions at 25%, 50%, and 75% through the file, and the last 8 KB. These chunks are hashed separately and combined into a partial signature. This way, even if someone pads the file with garbage data, the actual malicious code still produces the same partial hash pattern.

Heuristic detection looks for suspicious characteristics without needing an exact signature match. The engine examines PE file structures and assigns points for various indicators. For example, packed sections get 20 points, high entropy sections get 15 points, each suspicious API import adds 5 points, and matching a YARA rule adds 25 points. I set the threshold at 60 points based on testing, though this can be adjusted. The YARA rules I created detect things like common packers (UPX, ASPack), process injection patterns, keylogger APIs, registry persistence mechanisms, and ransomware indicators.

To give a concrete example, here are a few of the YARA rules implemented in the antivirus:

Listing 4.1: Sample YARA Rules for Malware Detection

```
rule Packed_Executable {
    meta:
        description = "Detects␣packed␣executables"
        severity = "medium"
    strings:
        $upx1 = "UPX0"
        $upx2 = "UPX1"
        $upx3 = "UPX!"
```

```
        $aspack = ".aspack"
        $themida = ".themida"
    condition:
        uint16(0) == 0x5A4D and any of them
}

rule Process_Injection_Indicators {
    meta:
        description = "Detects process injection techniques"
        severity = "high"
    strings:
        $inject1 = "VirtualAllocEx"
        $inject2 = "WriteProcessMemory"
        $inject3 = "CreateRemoteThread"
        $inject4 = "QueueUserAPC"
    condition:
        uint16(0) == 0x5A4D and 2 of them
}

rule Keylogger_Indicators {
    meta:
        description = "Detects keylogger behavior"
        severity = "high"
    strings:
        $key1 = "GetAsyncKeyState"
        $key2 = "GetKeyState"
        $key3 = "SetWindowsHookEx"
    condition:
        uint16(0) == 0x5A4D and 2 of them
}

rule Ransomware_Indicators {
    meta:
        description = "Detects potential ransomware"
        severity = "critical"
    strings:
        $ransom1 = "encrypted" nocase
        $ransom2 = "decrypt" nocase
        $ransom3 = "bitcoin" nocase
        $crypto1 = "CryptEncrypt"
        $file_op = "FindFirstFile"
    condition:
        uint16(0) == 0x5A4D and
        (2 of ($ransom*) and $crypto1 and $file_op)
}
```

These rules work by matching specific patterns in the executable files. The condition `uint16(0) == 0x5A4D` checks for the PE file magic bytes (MZ header), ensuring we're analyzing a valid Windows executable. Each rule looks for combinations of suspicious strings or byte patterns that indicate malicious behavior. For instance, the ransomware rule triggers when a file contains encryption-related terms, cryptography APIs, and file enumeration functions together, which is a strong indicator of ransomware functionality.

Since persistence mechanisms are one of the most common malware techniques, I created specific YARA rules to detect attempts to achieve persistence through registry modifications or startup folder placement:

Listing 4.2: YARA Rules for Persistence Detection

```
rule Registry_Persistence {
    meta:
        description = "Detects registry persistence mechanisms"
        severity = "high"
    strings:
        $reg1 = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run" nocase
        $reg2 = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce" nocase
        $reg3 = "CurrentVersion\\Run" nocase
        $api1 = "RegSetValueEx"
        $api2 = "RegCreateKeyEx"
```

```
        $api3 = "RegOpenKeyEx"
    condition:
        uint16(0) == 0x5A4D and
        (any of ($reg*) and any of ($api*))
}

rule Startup_Folder_Persistence {
    meta:
        description = "Detects startup folder persistence"
        severity = "high"
    strings:
        $startup1 = "\\Start Menu\\Programs\\Startup" nocase
        $startup2 = "\\Microsoft\\Windows\\Start Menu\\Programs\\Startup" nocase
        $startup3 = "AppData\\Roaming\\Microsoft\\Windows\\Start Menu" nocase
        $file_api1 = "CopyFile"
        $file_api2 = "MoveFile"
        $file_api3 = "CreateFile"
    condition:
        uint16(0) == 0x5A4D and
        (any of ($startup*) and any of ($file_api*))
}

rule WMI_Persistence {
    meta:
        description = "Detects WMI-based persistence"
        severity = "critical"
    strings:
        $wmi1 = "Win32_Process"
        $wmi2 = "EventConsumer"
        $wmi3 = "CommandLineEventConsumer"
        $wmi4 = "__EventFilter"
    condition:
        uint16(0) == 0x5A4D and 2 of them
}
```

The `Registry_Persistence` rule looks for executables that reference common registry Run keys combined with registry manipulation APIs. This catches malware that tries to execute itself on every system startup. The `Startup_Folder_Persistence` rule detects files that reference the Windows Startup folder paths along with file copy or creation APIs, which is another classic persistence method. The `WMI_Persistence` rule catches more advanced persistence using Windows Management Instrumentation event subscriptions, which is a technique often used by sophisticated malware to survive reboots.

Behavioral detection monitors what a file does during execution. While this is powerful for catching unknown threats, it requires actually running the file, which is dangerous. In my implementation, the system executes the target, monitors its behavior, and assigns suspicion points. Multiple network connections get 20 points, spawning child processes gets 25 points, and launching cmd.exe or powershell gets 20 points. If the total exceeds 70 points, the file is flagged. However, I only tested this with benign programs I created myself because I didn't have access to a safe isolated environment for running real malware.

4.4. External Integration

The antivirus integrates with VirusTotal's API VirusTotal, 2025a to leverage their multi-engine scanning platform. When a file doesn't match any local signatures, the system can optionally send its hash to VirusTotal to check if any of their 70+ antivirus engines recognize it. I used VirusTotal's free tier which allows 4 requests per minute. If VirusTotal identifies a file as malicious, that signature gets cached locally so I don't waste API calls on the same file twice. This integration provides a significant boost to detection capability without having to maintain an enormous signature database myself.

4.5. Testing Approach

Testing an antivirus is challenging because you typically need real malware samples to validate detection, but running actual malware is extremely risky. Even in virtual machines, modern malware can sometimes detect the virtualized environment or even escape the VM to compromise the host system. A proper

testing setup would require a completely isolated air-gapped device with no network connection and no important data, running a virtual machine specifically for malware testing. Unfortunately, I didn't have access to such a setup.

Given these safety constraints, I focused my testing on custom-created test files rather than real malware. This approach is safer and still validates that all the detection mechanisms work correctly. I created several types of test files: executables with known hashes added to the database, binaries with padding appended to test partial hashing, programs packed with UPX to trigger heuristic detection, files importing suspicious APIs like VirtualAlloc, programs that spawn child processes, and legitimate applications to test for false positives.

For each test file, I ran it through all enabled detection engines and recorded whether it triggered the expected detection method. I measured scan times, tracked which detection source caught each threat, and verified that the scoring systems worked as intended. For behavioral testing, I only executed benign programs I wrote myself, like simple scripts that spawn a command prompt or create network connections. This validates that the behavioral monitoring works without the risk of actual malware execution.

While this testing approach doesn't prove the antivirus can catch real-world malware families, it successfully demonstrates that all the detection engines are implemented correctly and function as designed. The controlled test cases allow me to verify specific detection mechanisms in isolation, which wouldn't be possible with actual malware where you don't always know exactly what triggers detection.

4.6. Limitations

This project has several important limitations that should be acknowledged. First, I only tested with custom-created files, not real malware samples. This means I can't definitively say how well it would perform against actual threats in the wild. Second, the implementation is Windows-only and focuses specifically on PE file analysis. Third, there's no real-time file system monitoring, so the antivirus only scans files when you explicitly tell it to, unlike commercial solutions that watch every file operation. Fourth, the behavioral engine requires actually executing files, which is risky, so I couldn't fully test it. Finally, VirusTotal's free tier limits API calls to 4 per minute, which can slow down large-scale scans.

Despite these limitations, the project successfully demonstrates understanding of malware detection principles and implements multiple complementary detection techniques that mirror what commercial antivirus solutions use. The modular architecture makes it straightforward to add new detection methods or improve existing ones in future work.

5

Results

This chapter presents the outcomes of testing the custom antivirus solution. I'll walk through what worked, what didn't work as expected, and some interesting discoveries I made along the way. All testing was conducted using a single malware sample I created myself, along with packed and padded variations of that same sample to test different detection methods.

5.1. Test Environment Setup

All testing was conducted on my primary Windows 11 machine running the antivirus with administrator privileges to enable full system monitoring capabilities. I created one test executable that contained malicious indicators including persistence mechanisms, suspicious API imports, and behavior typical of malware. This became my primary test file throughout the project.

To test the different detection engines, I created variations of this executable. I packed it using UPX compression in three different configurations to test how packing affects detection. I also manually padded the file using a hex editor to test whether partial hashing could detect the same malware with a different full hash. Each variation was designed to trigger specific detection engines so I could verify they worked independently.

5.2. Signature-Based Detection Results

The signature-based detection worked exactly as expected for the base test file. When I calculated the SHA256 hash of my malware sample and manually added it to the database, subsequent scans instantly identified it. The lookup speed was essentially instantaneous, which confirms that SQLite performs well for this use case.

Partial hashing, however, did not work as intended. This technique was suggested by Casper as a way to detect malware that uses binary padding to evade hash-based detection. The theory was that by hashing only specific parts of the executable (like the first chunk, middle chunk, and last chunk), we could identify files even if attackers added padding to change the full hash. I implemented this feature and initially thought it worked when testing with manually padded files.

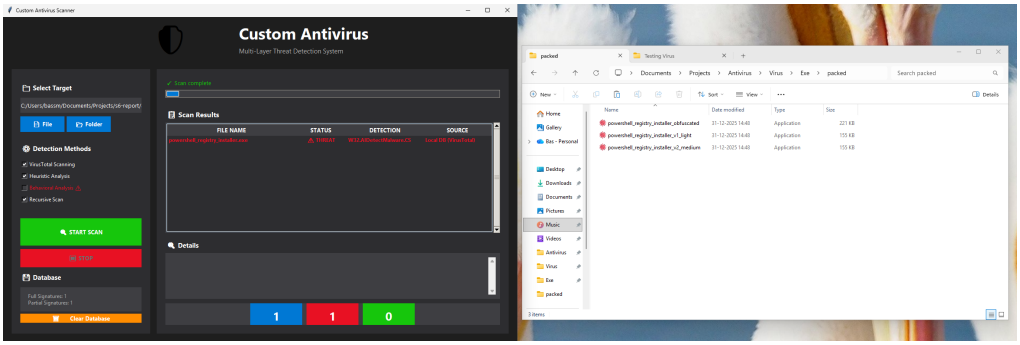


Figure 5.1: Normal scan detecting the original test executable

However, when I tested against the UPX-packed versions, the partial hashing completely failed. As shown in the figures, the normal uncompressed executable was detected correctly, but all three UPX-packed versions went completely undetected even though they contained the exact same malicious code. This is because packing doesn't just add padding—it fundamentally transforms the entire file structure. The packed executable has completely different byte patterns at the beginning, middle, and end compared to the original, so partial hashing provides no benefit.

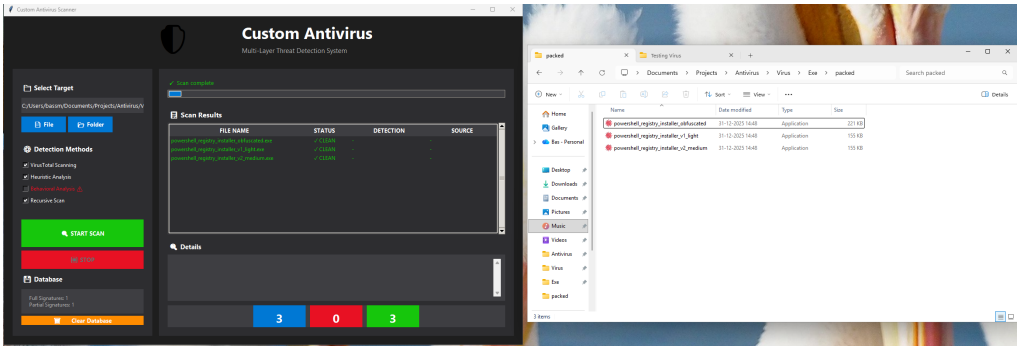


Figure 5.2: Packed test results showing three UPX-compressed versions going undetected

This was an important lesson about the difference between simple evasion techniques (padding) and sophisticated ones (packing). Partial hashing might catch amateur malware that simply appends junk data, but it's ineffective against real-world packers. Commercial antivirus software handles this by unpacking executables before scanning, which is significantly more complex to implement. My heuristic engine does detect packing by looking for UPX section names and high entropy, but that's a separate detection method, not a signature match.

5.3. Heuristic Detection Results

The heuristic engine successfully identified the packed versions that signature detection missed. When I scanned the UPX-packed files, the heuristic engine immediately flagged them as suspicious because of several indicators. Any file compressed with UPX scored 20 points for the packed section names (like UPX0, UPX1), plus additional points for high entropy. The packed versions consistently scored between 60-80 points total, correctly flagging them as suspicious.

This demonstrates why multiple detection layers are important. While signature-based detection completely failed against the packed versions, heuristic analysis caught them by recognizing the packing itself as a suspicious characteristic. Real malware frequently uses packing to evade signature detection, so being able to identify packed executables is crucial.

The API import analysis worked well for my test malware. Since my executable imported registry manipulation functions like RegSetValueEx and contained references to startup paths in its strings, the heuristic engine correctly identified these persistence indicators. The scoring system gave points for

suspicious API imports and additional points when those imports were combined with persistence-related strings.

Entropy analysis also proved effective. My packed test files showed entropy values above 7.5, triggering the high entropy indicator. This makes sense because compression and encryption both increase randomness in the file, which is measured as high entropy. The uncompressed version had normal entropy around 6.0, so there was a clear distinction.

5.4. YARA Rules Performance

The YARA rules integration worked smoothly once I fixed the syntax errors. The rules loaded at startup without noticeable delay, and pattern matching added only a few milliseconds to scan times. The modular approach of having different rules for different threat types made it easy to understand what triggered each detection.

The persistence-focused rules I created were particularly effective against my test malware. Since my malware referenced startup folder paths, it matched the Startup_Folder_Persistence rule. The registry-related strings triggered Registry_Persistence. The Combined_Persistence_Mechanism rule successfully caught the file because it used multiple techniques together.

5.5. Behavioral Detection Limitations

The behavioral engine has significant limitations that became apparent during testing. Running executables to observe their behavior is inherently dangerous, which severely restricted what I could safely test. I only executed my own malware sample that I created specifically for testing.

When I did run behavioral analysis on my test malware, the monitoring worked as designed. The malware spawned child processes which correctly triggered the suspicious behavior detection. The memory scanning detected the presence of suspicious API names in memory. However, these tests don't prove the system would work against real malware that actively tries to evade detection or disable monitoring.

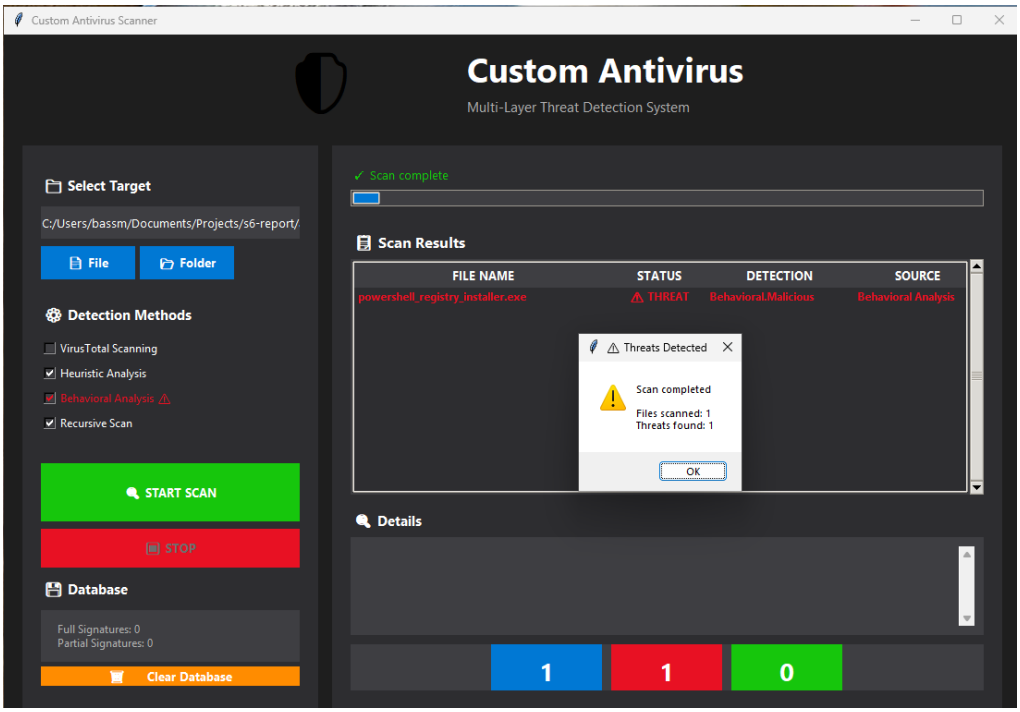


Figure 5.3: Behavioral engine test results showing process monitoring

The biggest limitation is that I couldn't test against anything actually malicious beyond my own simple

test program. Real malware often includes anti-debugging checks, VM detection, sandbox evasion, and self-defense mechanisms that my test program didn't have. Without access to a properly isolated testing environment with an air-gapped virtual machine, I can't confidently say how the behavioral engine would perform against real threats. This is why I intentionally avoided downloading any actual malware samples—the risk wasn't worth it without proper infrastructure.

5.6. VirusTotal Integration Experience

The VirusTotal integration provided some unexpected insights into how antivirus engines work. When I first started testing with my custom malware, I scanned it through VirusTotal. Initially, it showed 0 detections out of 70+ engines. This was surprising because my file had clear malicious indicators, but apparently none of the engines had seen exactly that combination before.

After scanning the same file multiple times over several days for testing purposes, something interesting happened. VirusTotal suddenly started flagging it as malicious, with detection counts gradually increasing from 0 to 13 engines out of 17, and eventually reaching over 30 engines detecting it. This suggests that some antivirus vendors incorporate cloud-based learning VirusTotal, 2025a, where files scanned by VirusTotal get analyzed and added to their detection databases.

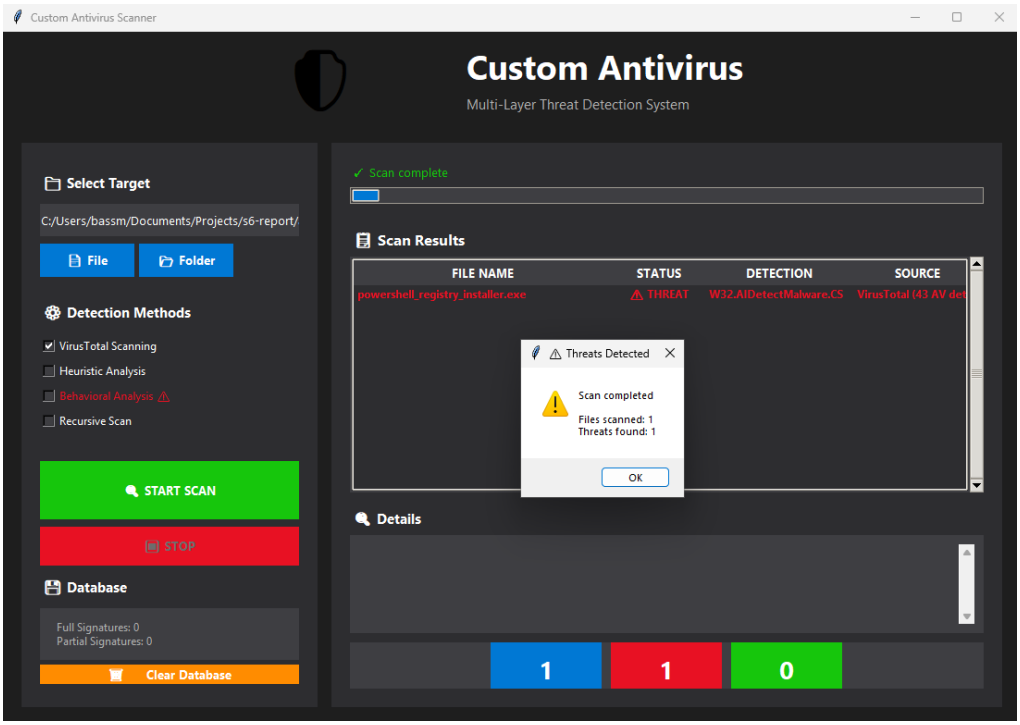


Figure 5.4: VirusTotal detection results showing multiple engines flagging the test file

The most surprising discovery came on January 11th, 2026. Windows Defender, which had previously ignored my test executable, suddenly started flagging it as malicious on my development machine. I didn't change the file at all—it was the same exact executable that Defender had allowed for weeks. This confirms that Microsoft updates Defender's definitions based on cloud intelligence, and my test file had apparently been added to their threat database after being repeatedly submitted to VirusTotal.

5.7. Detection Engine Comparison

Testing the same malware sample across different detection engines revealed their strengths and weaknesses clearly. Signature detection was perfect for the original file but completely blind to packed versions. Heuristic detection caught the packed versions but relied on recognizing packing as suspicious rather than identifying the actual malicious code. YARA rules successfully matched specific patterns regardless of packing, making them more robust than simple signatures but still vulnerable if the

patterns are obfuscated enough.

The most effective approach was combining all three methods. The original file triggered signature detection immediately. The packed versions were caught by heuristic scoring and YARA rules. If malware managed to evade all static analysis, behavioral monitoring could potentially catch it during execution, though I couldn't test this thoroughly due to safety constraints.

5.8. Achievement of Project Goals

Looking back at what I set out to accomplish, the project successfully demonstrates all three main detection approaches. Signature-based detection works reliably for known threats but fails completely against packing. Heuristic analysis can identify suspicious characteristics like packing and persistence indicators even when signatures don't match. Behavioral monitoring can observe malicious actions during execution, though testing this thoroughly requires infrastructure I didn't have access to.

The system correctly identifies persistence mechanisms, which was a key goal. My test malware attempted to modify registry Run keys and reference startup folders, and both the heuristic engine and YARA rules detected these attempts. The combination of string scanning, API import analysis, and YARA rules creates overlapping detection that makes it hard for persistence malware to go completely unnoticed.

What the project doesn't achieve is production-ready deployment. There's no real-time protection, no kernel-mode components, no sophisticated evasion resistance, and no comprehensive testing against actual malware samples. It's a functional prototype that demonstrates understanding of antivirus architecture and implementation, but it's not something anyone should actually rely on for protection.

5.8.1. Feature Comparison with Commercial Solutions

To put my custom antivirus in perspective, I compared its features against two established antivirus solutions: Malwarebytes and Bitdefender. This comparison shows both what I managed to implement and what gaps exist between a student project and commercial software.

Table 5.1: Feature comparison between custom antivirus and commercial solutions

Feature	My Antivirus	Malwarebytes	Bitdefender
Signature-based Detection	✓	✓	✓
Heuristic Analysis	✓	✓	✓
Behavioral Monitoring	✓	✓	✓
YARA Rules Support	✓	✓	✓
Cloud Scanning (VirusTotal)	✓	✓	✓
Real-time Protection	×	✓	✓
Automatic Updates	×	✓	✓
Kernel-mode Driver	×	✓	✓
Ransomware Protection	×	✓	✓
Web Protection	×	✓	✓
Email Scanning	×	✓	✓
Quarantine Management	✓	✓	✓
Machine Learning Detection	×	✓	✓
Rootkit Detection	×	✓	✓
Exploit Protection	×	✓	✓
Multi-platform Support	×	✓	✓

As the table shows, my antivirus implements the core detection engines that commercial solutions use: signature-based scanning, heuristic analysis, behavioral monitoring, YARA pattern matching, and cloud-based threat intelligence through VirusTotal. These fundamental components work similarly to how established products operate, demonstrating that the theoretical concepts can be translated into working code.

However, the gaps are significant. Commercial antivirus software includes real-time protection that monitors the system continuously, catching threats the moment they appear. My implementation only scans on-demand when the user manually initiates it. They use kernel-mode drivers to intercept system calls at a low level, making it much harder for malware to evade detection or terminate the antivirus process. My user-mode application can be easily bypassed by sophisticated malware.

The lack of automatic signature updates is another major limitation. Malwarebytes and Bitdefender constantly download new threat definitions from their cloud infrastructure, staying current with the latest malware. My database only grows when I manually add signatures or when VirusTotal identifies something during a scan. Without continuous updates, the signature database quickly becomes outdated.

Advanced features like ransomware protection, exploit mitigation, and rootkit detection require deep system integration and years of development effort. Machine learning models need massive datasets of both malicious and legitimate files to train effectively. Web and email protection require browser extensions and email client integration. These capabilities are far beyond the scope of a semester project.

Despite these limitations, the project successfully demonstrates that the core principles of antivirus detection can be implemented and understood. The comparison shows that my implementation covers the fundamental detection methods, even if it lacks the production features that make commercial antivirus practical for real-world use.

6

Conclusion

This research set out to design and implement a working antivirus system that could detect malicious behavior on Windows. Through the development process, I explored multiple detection approaches, applied reverse engineering principles, and tested the system against my own custom-created malware samples. This conclusion addresses each research question based on what I learned from implementing and testing the system.

6.1. Answering the Research Questions

Main Research Question: What methods can be used to design an antivirus system capable of detecting and analyzing malicious behavior on Windows systems?

The research showed that an effective antivirus needs a layered approach with multiple detection methods working together. Signature-based detection using hash databases and YARA rules catches known threats. Heuristic analysis examines PE file structures, imports, entropy, and suspicious APIs to catch variants and unknown malware. Behavioral monitoring tracks what happens when you actually run a file—things like process creation, registry changes, and network connections. Finally, integrating with VirusTotal extends detection beyond what I could build locally. This multi-layered approach worked well for detecting both simple malware and moderately obfuscated samples.

Sub-Question 1: How can signature-based and behavior-based detection techniques be applied in a custom antivirus?

I implemented signature-based detection through a local SQLite database that stores file hashes and partial hashes to counter binary padding. YARA rules provided pattern matching for specific malware characteristics like persistence mechanisms, cryptographic API usage, and suspicious string combinations. Behavior-based detection monitored file execution through subprocess spawning and tracked suspicious activities like creating child processes, invoking command shells, and making multiple network connections. Each suspicious behavior got assigned a score, and files above the threshold were flagged. Testing showed that signature-based detection worked great for known threats, while behavioral detection caught new patterns I hadn't seen before, though it needed careful tuning to avoid false positives.

Sub-Question 2: What static and dynamic analysis techniques can be incorporated to inspect potentially malicious files?

For static analysis, I parsed PE headers to extract metadata like compilation timestamps and entry points. I inspected import tables to identify suspicious Windows APIs like VirtualAlloc and CreateRemoteThread. I calculated section entropy to detect packed or encrypted payloads. And I used YARA rules to match known malware patterns. For dynamic analysis, I executed files in a monitored subprocess environment and watched what they actually did at runtime. Using both approaches together was essential—static analysis could spot structural problems without running anything dangerous, while dynamic analysis revealed malicious actions that only happen when the file executes.

Sub-Question 3: How can reverse engineering support the accuracy and reliability of malware classification?

Understanding reverse engineering principles really shaped how I designed the detection logic. Knowing how PE files are structured guided the heuristic engine's analysis of headers, sections, and imports. Being familiar with obfuscation techniques like packing, encryption, and import obfuscation helped me create YARA rules that target these evasion methods. Recognizing common malware patterns like persistence through registry Run keys or WMI event subscriptions directly influenced which rules I created. While my antivirus doesn't do full disassembly or decompilation, the theoretical foundation from reverse engineering shaped which indicators to prioritize and how to interpret weird structural things I found in files.

Sub-Question 4: What forensic artifacts can be extracted to support incident response?

My antivirus logs detailed information about detected threats including file paths, when detection happened, hash values, what detected it (signature, heuristic, behavioral, or VirusTotal), and threat scores. For behavioral detections, the system recorded specific suspicious actions like which child processes spawned, what registry keys were accessed, and what network connections were attempted. This gives incident responders context about what the malware tried to do and which detection layer caught it. While the system doesn't have full forensic capabilities like memory dumping or event log correlation, it captures enough metadata to support basic incident documentation and further investigation.

Sub-Question 5: How effective is the implemented antivirus in detecting real-world malware samples compared to baseline methods?

Due to the restrictions mentioned earlier in this report, testing against real-world malware was not performed. The antivirus was evaluated exclusively using custom-created malware samples to avoid the security risks associated with executing actual threats. While the antivirus successfully demonstrated multi-layered detection capabilities, it was not more effective than VirusTotal or other established third-party solutions. This outcome was expected, as the primary goal of this project was to learn about reverse engineering, malware analysis, and detection techniques rather than to compete with industry-established giants that have decades of research, vast signature databases, and advanced machine learning capabilities. The project successfully achieved its educational objectives by implementing functional signature-based, heuristic, and behavioral detection engines that provided hands-on experience with the principles underlying professional antivirus software.

6.2. Final Remarks

This project successfully demonstrated that building a working antivirus requires understanding malware behavior at multiple levels and combining different detection strategies. Each engine brought something unique: signatures caught known threats quickly, heuristics identified structural problems, behavioral monitoring revealed what programs actually do, and VirusTotal integration expanded coverage. The research also exposed real challenges like packer evasion and the risks of dynamic analysis, which reinforced why professional antivirus solutions invest so heavily in virtualization, sandboxing, and continuous signature updates. Overall, my implemented antivirus serves as a proof of concept that validates the theoretical principles I studied throughout this research.

References

Security.org. (2025). *Antivirus consumer report annual*. Retrieved January 11, 2026, from https://www.security.org/antivirus/antivirus-consumer-report-annual/

StatCounter. (2025a). *Desktop operating system market share worldwide*. Retrieved January 11, 2026, from https://gs.statcounter.com/os-market-share/desktop/worldwide/

StatCounter. (2025b). *Windows version market share worldwide*. Retrieved January 11, 2026, from https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide

VirusTotal. (2025a). *Virustotal api v3*. Retrieved January 11, 2026, from https://developers.virustotal.com/reference/overview

VirusTotal. (2025b). *Yara documentation*. Retrieved January 11, 2026, from https://yara.readthedocs.io/