University Mohammed VI Polytechnic

College of Computing

Static Analysis of Windows PE Executable

Lab 2 Part 1

EL HANI Mohammed- Rida

Day 2 - Diving Into the Rabbit Hole

Contents

1 Introduction

This report presents a comprehensive static analysis of a suspicious Windows PE executable. After getting our hands on this suspicious sample identified by (SHA256 hash: `d551a474ecf0b7d1ba6dda319e3b77fdecf39489eaf14b7d8837002f2d31387b.exe`), we decided to poke it with static analysis tools to see what secrets it's hiding. The executable weighs in at a hefty 1,307,648 bytes (roughly 1.25 MiB for those who prefer round numbers), and from the moment we ran `file` on it, we knew we were dealing with a PE32 executable targeting Intel 80386 with Mono/.NET assembly for MS Windows. The executable comes with 3 sections, which sounds innocent enough until you start digging deeper and realize that one of those sections is basically suspicious and screaming "I'm packed !".

The goal of this analysis was straightforward: figure out what this thing is, what it wants, and how badly it wants it. We threw every static analysis tool we could find at it, from `rabin2` to Detect It Easy (DIE), and even fired up radare2 for some quick automated analysis. What we discovered was a 32-bit Windows GUI application written in C# targeting .NET Framework v4.5.2, compiled (allegedly) back in 2017, with a suspicious amount of obfuscation, high entropy in the `.text` section, and enough red flags to make a Soviet parade jealous. But let's not get ahead of ourselves. Time to break down what we found, section by section, byte by byte, and suspicion by suspicion.

2 Sample Properties and Initial Fingerprinting

Let's start with the basics. Running `rabin2 -I` on the sample gave us a wealth of information about the binary's architecture and security features. The sample is confirmed to be a 32-bit PE executable (`arch x86`, `bits 32`) with a base address (`baddr`) of `0x400000`, which is pretty standard for Windows PE files. The binary size (`binsz`) is 1,307,648 bytes, and the binary type (`bintype`) is, unsurprisingly, `pe`. The subsystem is listed as Windows GUI, meaning this isn't some sneaky console application trying to hide in the background. No, this malware wants you to see it , while it encrypts your files and asks for Bitcoin.

Now, here's where things get interesting. The `rabin2` output shows that the sample has stack canaries enabled (`canary: true`), NX (No Execute) protection enabled (`nx: true`), and position-independent code (`pic: true`). These are all modern exploit mitigation techniques, which suggests that whoever wrote this malware isn t playing around. However, the binary is not signed (`signed: false`), which is a massive red flag. The compilation timestamp is listed as `Sat Jul 22 07:53:55 2017`, which seems oddly specific until you remember that attackers love to forge timestamps to throw off forensic timelines (Anti forensics method). So yeah, this information is unreliable until we discover other clues.

One of the most damning pieces of evidence is the leaked PDB (Program Database) path: `C:\Users\Ben\Desktop\Panel\project\Wanna\Wanna\obj\Debug\Wanna.pdb`. This is basically the malware author's resume. We now know that the developer's name is probably Ben (or at least that's what he called his user folder), he was working on a project called "Wanna" (subtle naming, very original), and he compiled this in Debug mode, which means he left all the debugging symbols intact. This is what we call weak

OPSEC, and it's a gift to reverse engineers everywhere. The fact that the PDB path is still embedded in the binary tells us that this malware was either rushed, made by an amateur, or both.

The language is listed as CIL (Common Intermediate Language), which confirms that this is a .NET assembly. Specifically, the runtime version is .NET Framework v4.5.2 with CLR v4.0.30319, as confirmed by Detect It Easy. The toolchain used was likely Microsoft's Visual Studio, which makes sense given the project structure visible in the PDB path. The entry point is located at `0x005409aa`, which translates to an RVA (Relative Virtual Address) of `0x001409aa` when you account for the base address of `0x00400000`. This entry point placement is worth noting because it's located very late in the `.text` section, which is suspicious and often indicates stub or loader-style code.

3 PE Header Analysis and Section Layout

Let's talk about the PE header, because that's where the structure of this malware really starts to reveal itself. The optional header shows us that this is a PE32 binary (Magic number `0x010b`), with an ImageBase of `0x00400000`, which is the default for Windows executables. The AddressOfEntryPoint is `0x001409aa`, and the SectionAlignment and FileAlignment values are `0x2000` (8 KiB) and `0x200` (512 bytes), respectively. These are completely standard values, which means the PE header itself hasn't been tampered with or corrupted. The SizeOfCode field reports `0x0013ea00`, which is approximately 1.24 MiB, meaning almost the entire binary is occupied by the `.text` section. That's a huge code section, and it's our first hint that something is packed or compressed inside.

The section table extracted with `rabin2 -S` shows three sections: `.text`, `.rsrc`, and `.reloc`. The `.text` section is massive, with a virtual size of `0x13e9b0` bytes (1,305,008 bytes) and permissions set to `-r-x` (read and execute). This section contains the bulk of the executable code, and as we'll see later, it's also where the packing and obfuscation live. The `.rsrc` section is tiny at `0x58c` bytes (1,420 bytes) with permissions `-r-` (read-only), and it stores the embedded resources like images and strings. The `.reloc` section is even smaller at `0xc` bytes (12 bytes), also read-only, and it contains the relocation information needed for ASLR (Address Space Layout Randomization). The presence of the `.reloc` section confirms that the binary advertises ASLR support, although the relocation table is suspiciously small with only one relocation block. This suggests that while ASLR is technically enabled, the relocations are limited, which might mean the binary doesn't fully leverage ASLR or that the packing process stripped most relocations.

Now, let's talk about that entry point placement, because it's genuinely weird. The `.text` section starts at RVA `0x2000` and has a virtual size of `0x13e9b0`, which means it ends around `0x1409b0`. The entry point RVA is `0x1409aa`, which places it just 6 bytes before the end of the `.text` section. This is highly unusual for a normal executable, where you'd expect the entry point to be near the beginning of the code section. When the entry point is shoved all the way to the end like this, it's typically a sign of a loader stub or unpacking routine. The idea is that execution starts at this tiny stub, which then unpacks or decrypts the real payload hidden elsewhere in the binary. This suspicion is further supported by the fact that radare2's automated analysis only found 7 functions in a 1.25 MiB binary, which is absurdly low and suggests that most of the code is obfuscated or hidden from static analysis.

4 Packing, Obfuscation, and Entropy Analysis

Here's where things get really fun. Detect It Easy flagged the binary with a heuristic packer hint: "Compressed or packed data [High entropy + Section 0 ('.text')]". This is basically DIE's way of saying "yeah, this thing is definitely hiding something." High entropy in a code section is a dead giveaway for packing or compression, because normal executable code has relatively low entropy due to its structured nature. Encrypted or compressed data, on the other hand, looks like random noise and has very high entropy. The entropy analysis shows that the `.text` section has an entropy of approximately 7.99022, which is extremely high (the maximum entropy is 8.0 for truly random data). Meanwhile, the PE header has an entropy of only 2.78985, and the `.rsrc` and `.reloc` sections have entropies of 4.02761 and 0.08436, respectively. The overall packing percentage shown in the entropy visualization is 99%, which means almost the entire binary is packed.

So what does this mean in practice? It means that the majority of the `.text` section is not actually executable code in the traditional sense. Instead, it's compressed or encrypted data that will be unpacked at runtime. This is a common obfuscation technique used by malware authors to evade static analysis and signature-based detection. When you run the binary, the loader stub at the entry point will decompress or decrypt this packed data, load it into memory, and then transfer execution to the unpacked code. This is why radare2's automated analysis only found 7 functions and why the import table shows only 1 import (specifically, `_CorExeMain` from `mscoree.dll`, which is the .NET runtime entry point). The real functionality is hidden inside that packed blob, waiting to be unleashed at runtime.

To confirm this suspicion, we used DIE's extractor feature to see what resources were embedded in the binary. The extractor revealed multiple sections containing zlib-compressed data, as well as several PNG images and even a JPEG. The presence of embedded image resources makes sense given that this is a GUI application (more on that later), but the fact that everything is stored in compressed zlib format confirms that the malware is using compression to hide its components and payload. The binary decompresses these resources at runtime, which is consistent with our entropy analysis and explains why the `.text` section is so bloated.

5 Strings Analysis and Attribution

Now let's talk about the strings, the fun part, where this malware author left us some breadcrumbs. Running strings analysis on the binary revealed a treasure trove of information, most of it in Turkish. The very first string we noticed was "Ooops, your files have been encrypted!", which is basically the malware equivalent of leaving a signed confession at the crime scene. Other notable strings include "Send $7000 worth of bitcoin to this address:", followed by the Bitcoin address `1Hydrt8E3ybCwS7YBtggFFNn1AyrqWaBzz`. The presence of a hardcoded Bitcoin address is both hilarious and incredibly useful for attribution, because it means we can potentially track payments made to this address on the blockchain.

The Turkish strings are particularly interesting because they give us insight into either the author's native language or the intended victim group. Strings like "leminiz Kontrol

Edilecektir" (It will be checked), "deme" (don't say), and "olduysa <Decrypt> butonu aktif olacaktır" (If it worked, the <Decrypt> button would be active) are written in an informal, almost conversational tone. This suggests that the author is either Turkish or targeting Turkish-speaking victims, and that they're not a highly professional threat actor. Professional malware operations typically use more formal language and avoid revealing linguistic characteristics that could aid in attribution.

Other strings reveal the workflow of the ransomware: "Check Payment", "Decrypt button will be active if it worked", and "Programdan bize bildirim gelecektir" (We will receive a notification from the program). These strings outline a classic ransomware lifecycle: encrypt the victim's files, display a ransom note demanding Bitcoin payment, provide a "Check Payment" button that supposedly verifies whether the ransom has been paid, and then activate a "Decrypt" button if the payment is confirmed. The fact that the malware claims it will "receive a notification" suggests that it's checking payment status locally, possibly by querying a Bitcoin blockchain API, rather than using a sophisticated command-and-control (C2) infrastructure.

The GUI-related strings like "pictureBox1", "label1", and "Microsoft Sans Serif" confirm that this is a Windows Forms application with a graphical user interface. This is consistent with the subsystem type (Windows GUI) reported by the PE header, and it means the ransomware is designed to interact directly with the victim through a visible window. This is typical of the WannaCry ransomware family, which displays a prominent ransom note with countdown timers and payment instructions. Speaking of which, the internal naming of the project ("Wanna") and the similarity of the GUI strings to known WannaCry variants strongly suggest that this malware is either inspired by or directly imitates WannaCry.

6 WannaCry Attribution and GUI Comparison

After analyzing the strings and static artifacts, we had a strong suspicion that this sample was related to the WannaCry ransomware family. To confirm this hypothesis, we did some research on known WannaCry attacks, read some writeups on reversing WannaCry samples, and compared the behavior and GUI with documented WannaCry variants. The results were striking. The ransom note GUI that this malware displays is almost identical to the one used by WannaCry, down to the layout, color scheme (dark red background with yellow text), countdown timers, and the presence of "Check Payment" and "Decrypt" buttons.

The main differences are the language (this variant uses Turkish instead of English or other languages) and some minor details in the threat message, which suggests that this is either a localized version targeting Turkish victims or a clone/variant created by a Turkish-speaking threat actor. The internal project name "Wanna" (visible in the PDB path and file version strings) and the executable name "Wanna.exe" are dead giveaways. The original WannaCry ransomware used similar naming conventions, and it's clear that this sample is paying homage to (or straight-up ripping off) the original.

The ransom amount of $7000 worth of Bitcoin is interesting because it's relatively high compared to typical ransomware demands, which usually range from a few hundred to a few thousand dollars. This could indicate that the malware is targeting businesses or organizations rather than individual home users, or it could simply be that the author is

feeling ambitious. The Ransom note threatens permanent data loss (“your passwords are gone”, “data cannot be recovered”)is a classic social engineering tactic used to pressure victims into paying quickly without thinking. The message also includes reassurances like "no damage to NAS or terminals", which is meant to make the victim feel like the attacker is reasonable and trustworthy (spoiler: they’re not) .

7 .NET Metadata and Managed Code Analysis

Since this is a .NET assembly, we also took a look at the managed metadata to understand the structure of the managed code. Using a .NET metadata viewer, we confirmed that the binary contains valid .NET metadata tables, including TypeRef (78 entries), TypeDef (6 entries), Field (22 entries), MethodDef (34 entries), MemberRef (109 entries), and Assembly/AssemblyRef tables. The presence of these tables confirms that this is not just a native PE executable with a .NET stub; it’s a full-fledged managed application written in C#.

The assembly manifest shows that the application is targeting .NET Framework v4.5.2 with CLR v4.0.30319, and the requested execution level is "asInvoker" with UI access set to false. This means the malware doesn’t require administrator privileges to run, which makes it easier to execute on victim machines without triggering UAC (User Account Control) prompts. The assembly identity shows that the version is 1.0.0.0, and the assembly name is "MyApplication.app", which is a generic default name that suggests the author didn’t bother to change it from the Visual Studio project template.

However, despite the presence of managed metadata, static analysis of the managed code is difficult because of the packing/obfuscation we identified earlier. The high entropy of the `.text` section suggests that the managed assemblies or methods are encrypted or compressed, and they’re only decrypted at runtime. This is a common technique used by .NET malware to evade static analysis tools that rely on parsing the managed metadata. To fully understand the managed code, we would need to either unpack the binary manually (Insane difficulty, because it’s using a custom packer *-*) or perform dynamic analysis to capture the unpacked code in memory.

8 Security Mitigations and Defensive Features

Let’s talk about the security features that this binary advertises, because they’re actually kind of interesting. The PE header shows that the binary has several modern exploit mitigation features enabled, including DEP/NX (Data Execution Prevention / No Execute), ASLR (Address Space Layout Randomization), and even High Entropy ASLR. These flags are visible in the DLL characteristics field of the optional header: IMAGE_DLLCHARACTERISTICS_HIGH_ENTROPY_VA, IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE, IMAGE_DLLCHARACTERISTICS_NX_COMPAT, IMAGE_DLLCHARACTERISTICS_NO_SEH, and IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE.

DEP/NX prevents code execution from data segments, which makes it harder for attackers to exploit buffer overflows and similar vulnerabilities. ASLR randomizes the base address of the executable at load time, making it harder to predict memory addresses for

exploitation. High Entropy ASLR increases the randomization space, further improving security. The NO_SEH flag indicates that the binary doesn't use Structured Exception Handling, which is fine for managed code. The TERMINAL_SERVER_AWARE flag means the binary is compatible with Terminal Services (Remote Desktop), which could be relevant if the malware is designed to spread across RDP sessions.

However, as we noted earlier, the relocation table is suspiciously small with only one relocation block containing 12 bytes. This suggests that while ASLR is technically enabled, the binary might not have many relocations to randomize, which could limit the effectiveness of ASLR. Additionally, the binary is not signed (signed: false), which is a massive red flag. Legitimate software is almost always digitally signed by the publisher to prove authenticity and integrity. The lack of a digital signature makes it trivial to identify this as potentially malicious software.

The stack canary protection (canary: true) is also worth mentioning, although it's more relevant for native code than managed code. Stack canaries are used to detect stack buffer overflows by placing a random value on the stack before the return address and checking it before the function returns. If the canary value has been corrupted, it indicates a buffer overflow attack, and the program terminates. This is a useful defensive feature, but it's less critical for .NET applications where memory safety is largely handled by the CLR.

9 Radare2 Analysis and Function Discovery

Finally, let's talk about what radare2 found when we ran automated analysis on the binary. The results were... underwhelming. Radare2 reported 7 functions, 48,797 xrefs (cross-references), 6 calls, 22 strings, 1 import, 1 symbol, and a code size of 1,310,720 bytes. The function count of 7 is absurdly low for a 1.25 MiB binary, which strongly supports our suspicion that most of the code is packed or obfuscated. Normal executables of this size would typically have hundreds or even thousands of functions, so the fact that radare2 only found 7 means that its automated analysis couldn't properly disassemble the packed code.

The 48,797 xrefs number is also interesting because it's very high relative to the function count. This could indicate that radare2 is picking up a lot of data references or that the packed code contains a lot of internal cross-references that don't correspond to actual function boundaries. The 6 calls and 1 import numbers are consistent with our earlier findings: the only import is `_CorExeMain` from `mscoree.dll`, and there are very few native function calls because this is a managed .NET application.

The code coverage reported by radare2 is 457 bytes out of 1,310,720 total bytes, which gives a coverage percentage of 0%. This means that radare2's automated analysis barely scratched the surface of the binary's code. The vast majority of the `.text` section remains unexplored because it's packed and can't be properly disassembled without unpacking it first. This is exactly what we expected based on the entropy analysis and packing hints from DIE.

10 Conclusions and Next Steps

So, what have we learned from this static analysis? Quite a lot, actually. We've confirmed that this is a 32-bit Windows GUI executable written in C# targeting .NET Framework v4.5.2, likely compiled with Visual Studio in 2017 (We found "get_Screen_Shot_2017_05_12_at_4_05_23_PM" in strings). The binary is heavily packed or compressed, with a high-entropy `.text` section that hides the real functionality. The entry point is located at the very end of the `.text` section, which suggests a loader stub that unpacks the real code at runtime. The binary advertises modern security features like DEP, ASLR, and stack canaries, but it's not signed, which immediately marks it as suspicious.

The strings analysis revealed that this is ransomware, specifically a WannaCry-style variant that demands $7000 in Bitcoin, displays a GUI ransom note , that includes Turkish language strings that suggest either Turkish authorship or targeting. The leaked PDB path gives us strong attribution to a developer named "Ben" (suspicious mails and names were leaked too in strings) working on a project called "Wanna", and the weak OPSEC (leaving debug symbols intact, hardcoded Bitcoin address, informal language) suggests this is the work of a less sophisticated threat actor.

The .NET metadata confirms that this is a managed application, but static analysis of the managed code is hindered by the packing. Radare2's automated analysis only found 7 functions in a 1.25 MiB binary, which tells us that most of the code is hidden and will only reveal itself at runtime. The entropy analysis and DIE extractor confirm that the binary uses zlib compression to store its components and payload.

What we couldn't determine from static analysis alone are the exact encryption algorithms used to encrypt victim files, the decryption keys, the specific runtime unpacking behavior, any C2 communication or network activity, and the persistence mechanisms. All of these questions require dynamic analysis, which means we need to run this malware in a controlled isolated environment (CAPEv2 Sandbox for example) and observe its behavior at runtime.

The next steps are clear: set up a safe analysis environment, execute the malware under observation, capture the unpacked code, analyze the encryption routines, trace the GUI behavior, monitor network activity, and document everything. But that's a story for another day (too lazy to build a sandbox) . For now, we can confidently say that this sample is a WannaCry-inspired ransomware written by someone named Ben who really should have stripped those debug symbols.