Statistical Inconspicuousness

// Brandon McGrath

// Michael Ranaldo

// Steelcon 2024

Art by Dan Mumford

$ whoami

- Brandon McGrath
- Red Teamer / Targeted Operations @ **TrustedSec**
- Developer @ **pre.empt**
- twitter.com/**__mez0__**
- Second time SteelCon Speaker

$ whoami

- Michael Ranaldo
- Senior Security Consultant @ **Trustmarque**
- Developer @ **pre.empt**
- twitter.com/**michaeljranaldo**
- First time SteelCon Speaker

$ Background

- Maelstrom series in 2022, focusing on implant development and detection
- Looking at the wider malware marketplace - “what is the statistically average binary”
- Thinking about incorporating ML to analyse this
- EMBER stood out as a heavily influential dataset for ML
- Some of the points of interest on a binary from this dataset as part of our analysis

Brandon McGrath, Michael Ranaldo: Maelstrom #1: An Introduction
Brandon McGrath: From Chaos to Clarity: Organizing Data with Structured Formats
Hyrum S. Anderson, Phil Roth: EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models
Shay Banon: Welcome Endgame: Bringing Endpoint Security to the Elastic Stack

$ Research Goal

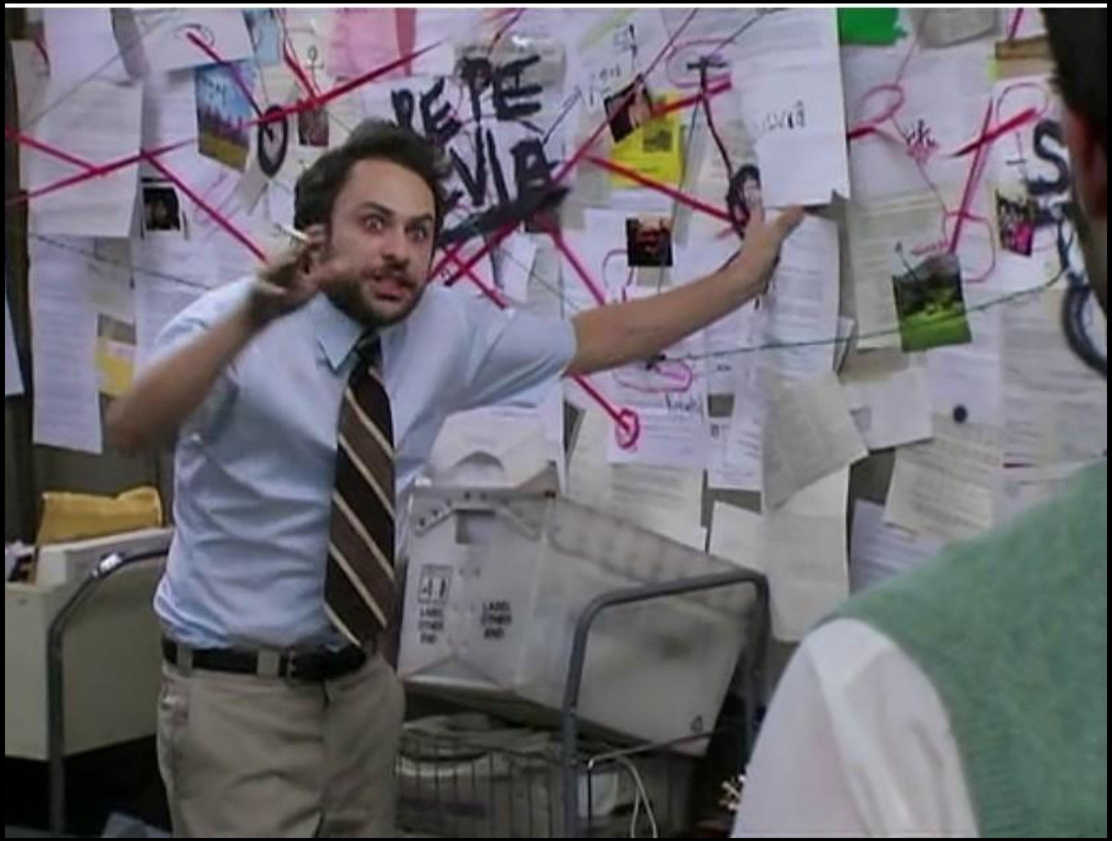
- This is a subset of a bigger goal we've been working on since Summer 2022
- Our research was focused on:
  - Looking at binaries en mass
  - Can we get a comprehensive list of as many samples as possible
  - What are their commonalities when considering their "points of interest"
  - What can we learn, both offensively and defensively, for building red team implants
- We will explain our terminology and data sources over the next slides!

$ Today's Goal

- We are going to show some preliminary findings from our research
- Storytime: *Three implants appear on the desk of a reverse engineer*
- Using the points of interest from our research:
  - What are common steps people take when writing malware to evade detection?
  - How can these in turn make an implant stand out more?
  - What improvements can be made to better the implant?
  - What should defenders be aware of for the future?
    - *More of a vendor issue, maybe?*
- We want to make you paranoid about every payload you ever write

Our Data, Definitions, and Cast

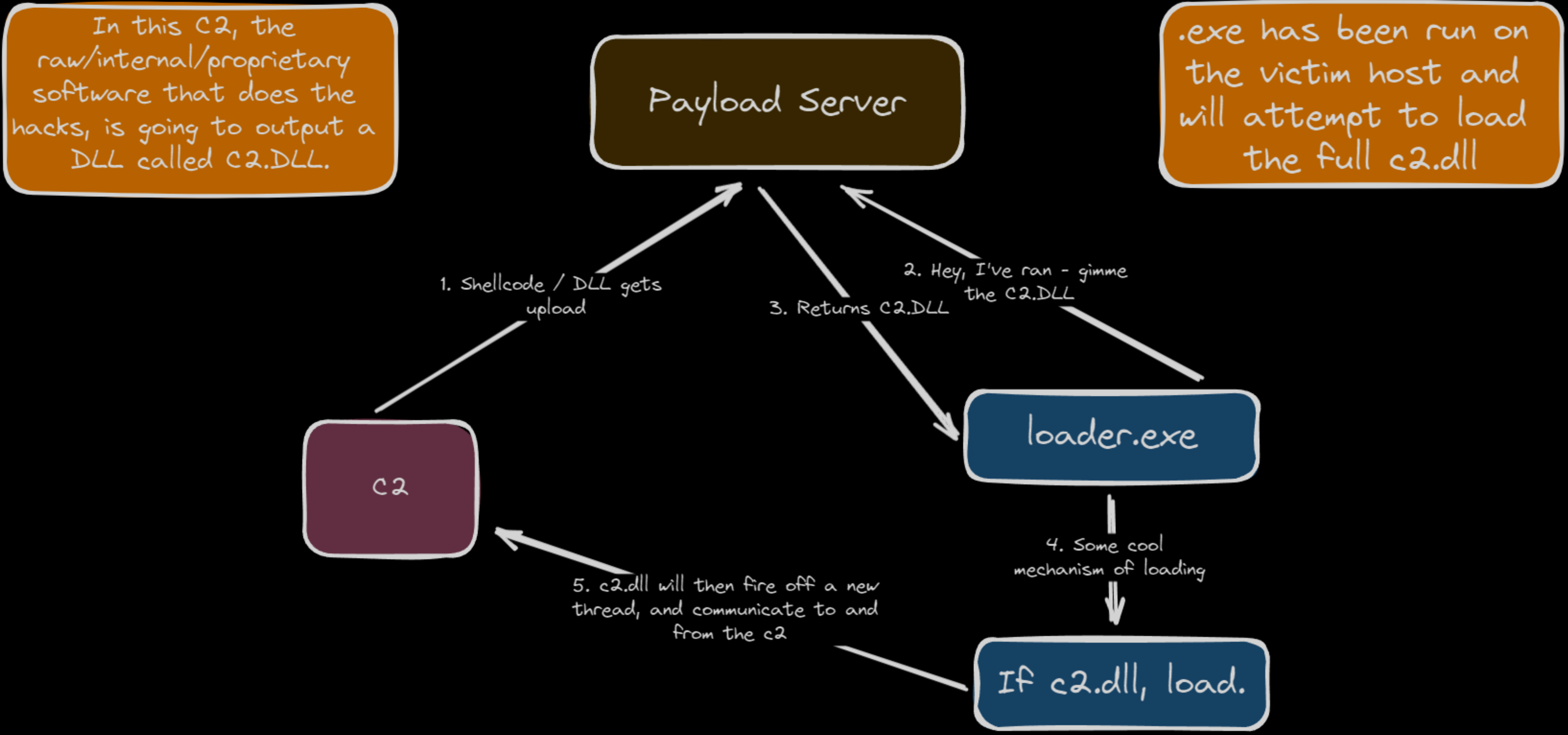
$ When you say “Implant”...



beacon
implant
cradle
loader
demon
dropper
packer

Jong-Wouk Kim, Yang-Sae Moon, Mi-Jung Choi: An Efficient Multi-Step Framework for Malware Packing Identification
Arne Swinnen, Alaeddine Mesbahi: One packer to rule them all

$ So just to be clear



In this C2, the raw/internal/proprietary software that does the hacks, is going to output a DLL called C2.DLL.
Payload Server
.exe has been run on the victim host and will attempt to load the full c2.dll
1. Shellcode / DLL gets upload
3. Returns C2.DLL
2. Hey, I've ran - gimme the C2.DLL
C2
loader.exe
4. Some cool mechanism of loading
5. c2.dll will then fire off a new thread, and communicate to and from the c2
If c2.dll, load.

Any.Run: Loader

$ Types of binary

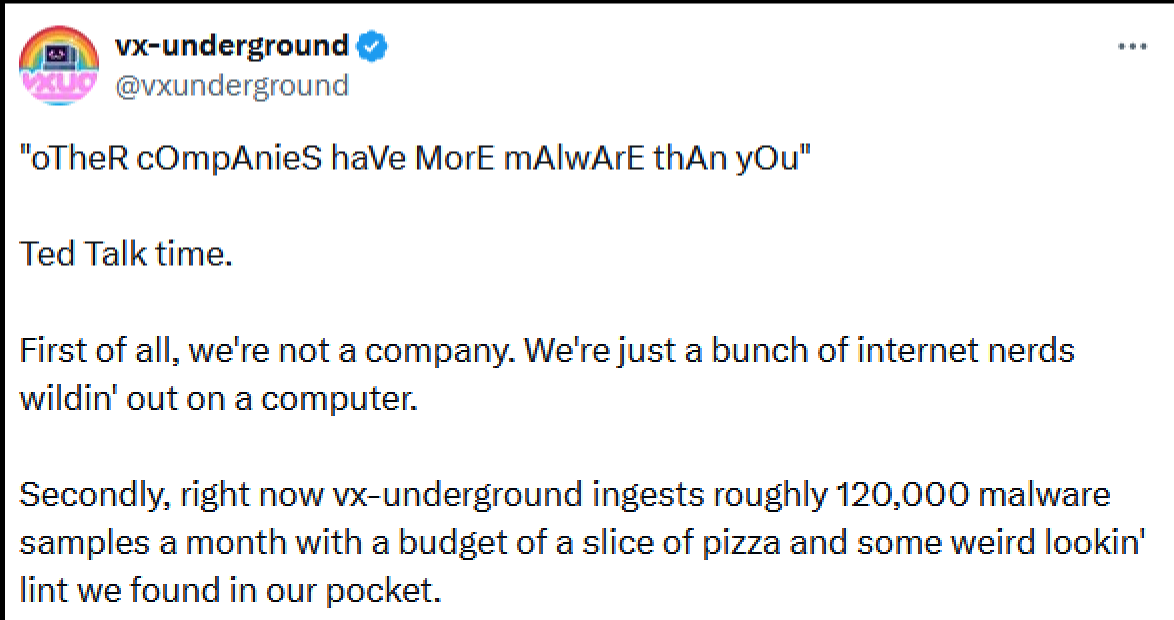
Malware

Goodware

Winware

$ Data Sources: Malware

- Sophos SOREL
- Malware Bazaar
- *Didn't* scrape vx-underground out of respect for smelly



vx-underground
@vxunderground

"oTheR cOmpAnieS haVe MorE mAlwArE thAn yOu"

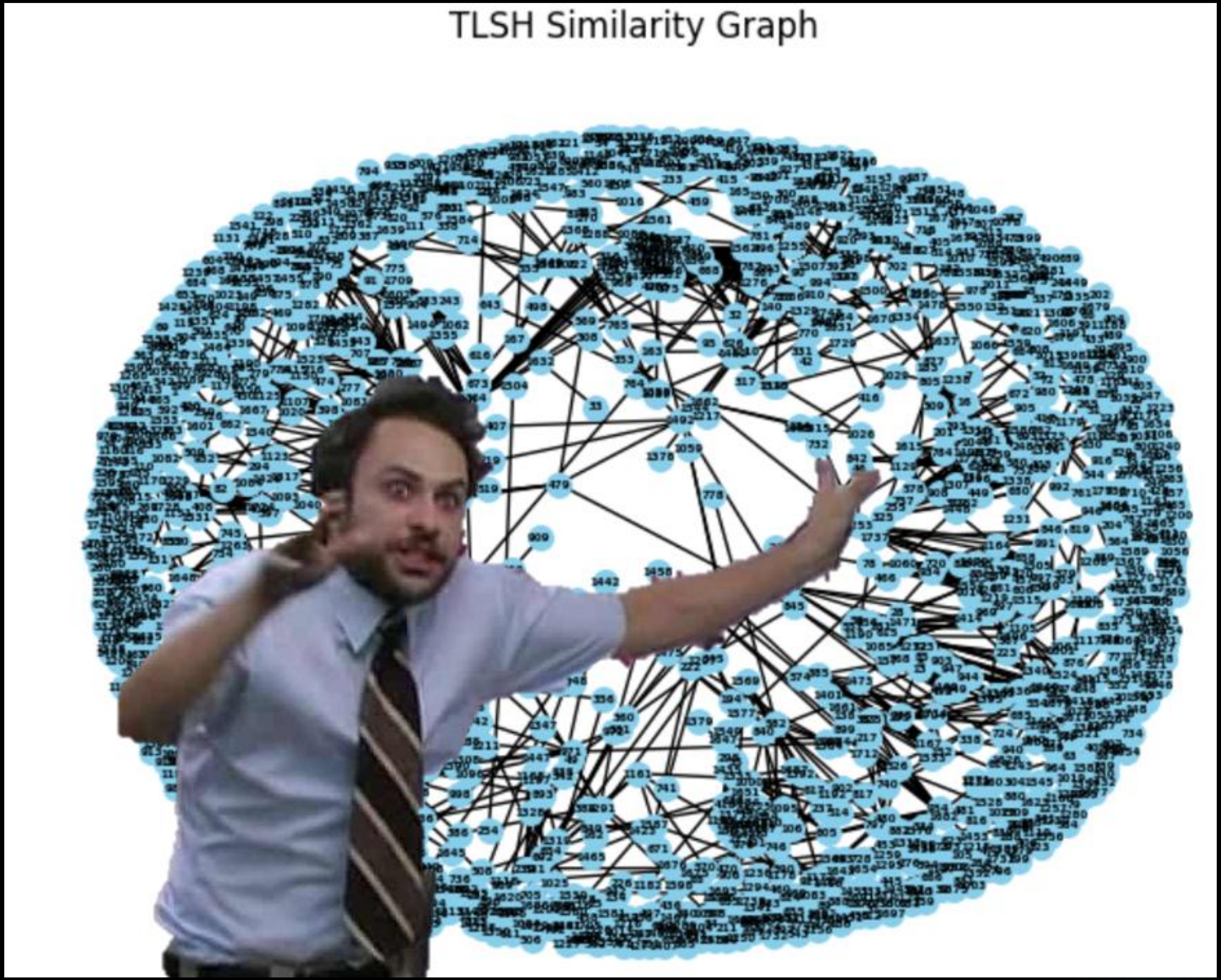
Ted Talk time.

First of all, we're not a company. We're just a bunch of internet nerds wildin' out on a computer.

Secondly, right now vx-underground ingests roughly 120,000 malware samples a month with a budget of a slice of pizza and some weird lookin' lint we found in our pocket.

[Sophos: Sophos Reversing Labs SOREL 20 million sample malware dataset]
[Abuse.ch: MalwareBazaar]
[VX Underground]

$ Data Sources: Malware



$ Data Sources: Goodware

- NIST Software Quality Group + HybridAnalysis
- Chocolatey
- winget – excluding anything published by Microsoft
- Ninite
- Also didn’t get historic chocolatey, just everything currently available at the latest version

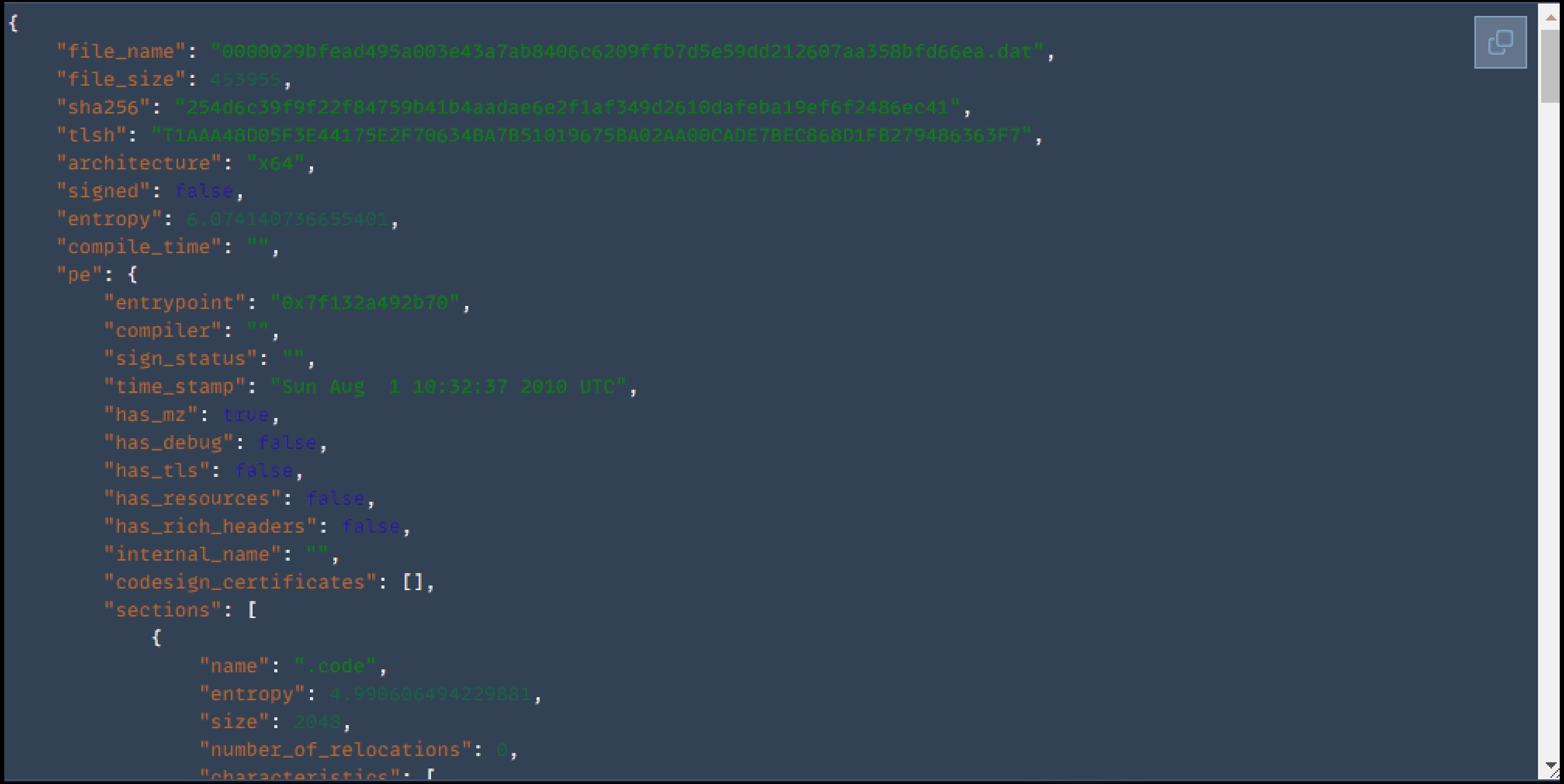
NIST: Current RDS Hash Sets
Hybrid Analysis
Chocolatey
Ninite

$ Data Sources: Winware

- Windows 10
- Windows 11
- Windows Server 2022
- winget – filtering for just Microsoft in the ID
- Everything including Visual Studio and Office and all server roles and optional features

Microsoft: Microsoft | Store
Wikipedia: Microsoft Windows

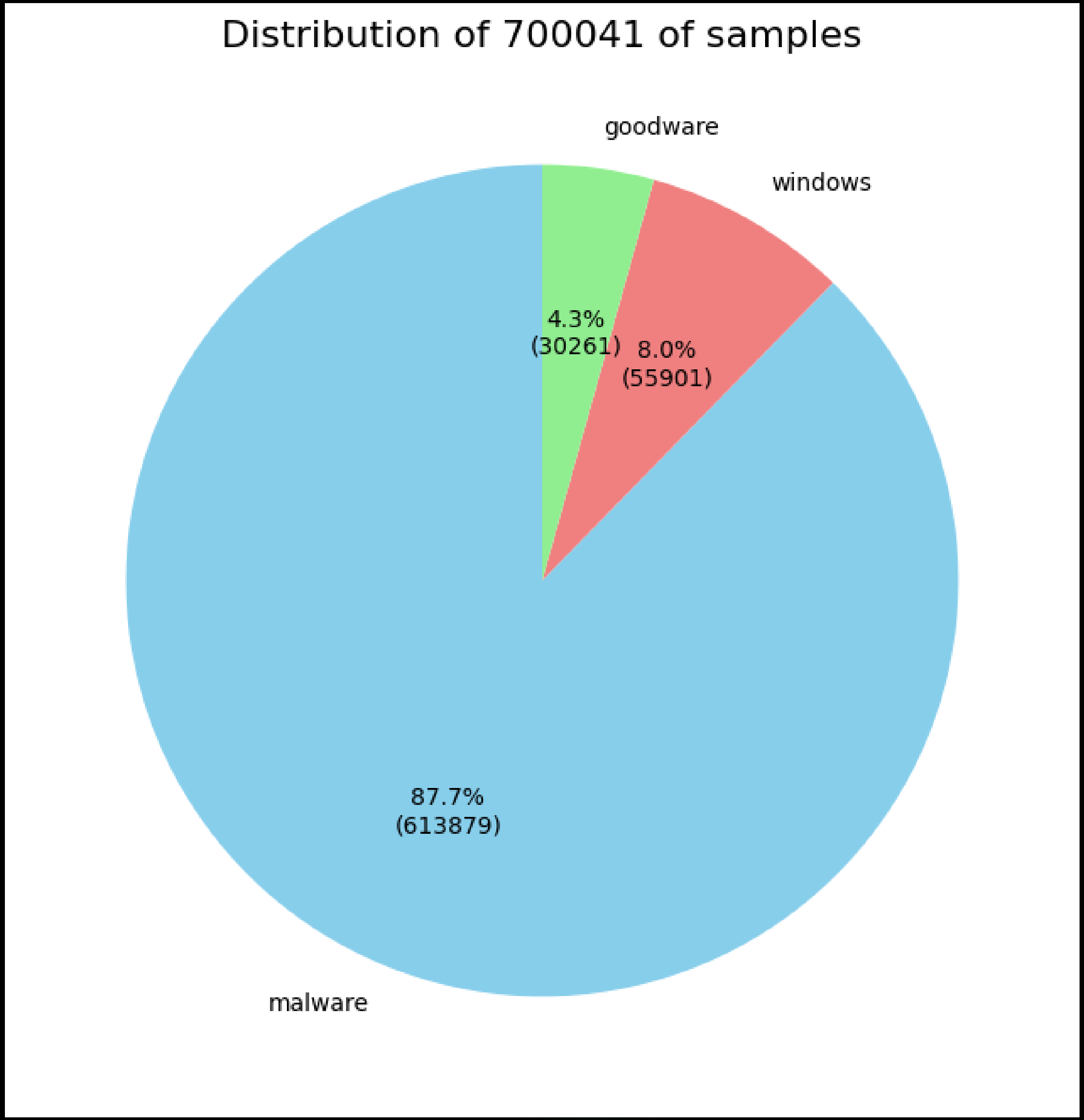
$ Sample



```
{
    "file_name": "0000029bfead495a003e43a7ab8406c6209ffb7d5e59dd212607aa358bfd66ea.dat",
    "file_size": 453955,
    "sha256": "254d6c39f9f22f84759b41b4aadae6e2f1af349d2610dafeba19ef6f2486ec41",
    "tlsh": "T1AAA48D05F3E44175E2F70634BA7B51019675BA02AA00CADE7BEC868D1FB279486363F7",
    "architecture": "x64",
    "signed": false,
    "entropy": 6.074140736655401,
    "compile_time": "",
    "pe": {
        "entrypoint": "0x7f132a492b70",
        "compiler": "",
        "sign_status": "",
        "time_stamp": "Sun Aug  1 10:32:37 2010 UTC",
        "has_mz": true,
        "has_debug": false,
        "has_tls": false,
        "has_resources": false,
        "has_rich_headers": false,
        "internal_name": "",
        "codesign_certificates": [],
        "sections": [
            {
                "name": ".code",
                "entropy": 4.990606494229881,
                "size": 2048,
                "number_of_relocations": 0,
                "characteristics": [
```

$ Data

- Goodware: 30,261
- Winware: 55,901
- Malware: 613,879



$ Our Cast

Meter

er*

Exploring

Art by Dan Mumford

$ Points of Interest

- Entropy
- File Size
- Imports
- Exports
- Code Signing
- Compiler

$ The task

- You’re on a red team
- Something something assumed breach
- You need a loader but you’re unsure
- So you go to Discord
- …

$ Solution 1: Encryption

$ Solution 1: Entropy

- Using Shannon Entropy, we measured the total entropy of a file as well as per-section
- Even back in 2006, 80 to 90% of detected malware was found to use encryption or packing
- Old technique, but still works. Literally the oldest trick in the book.

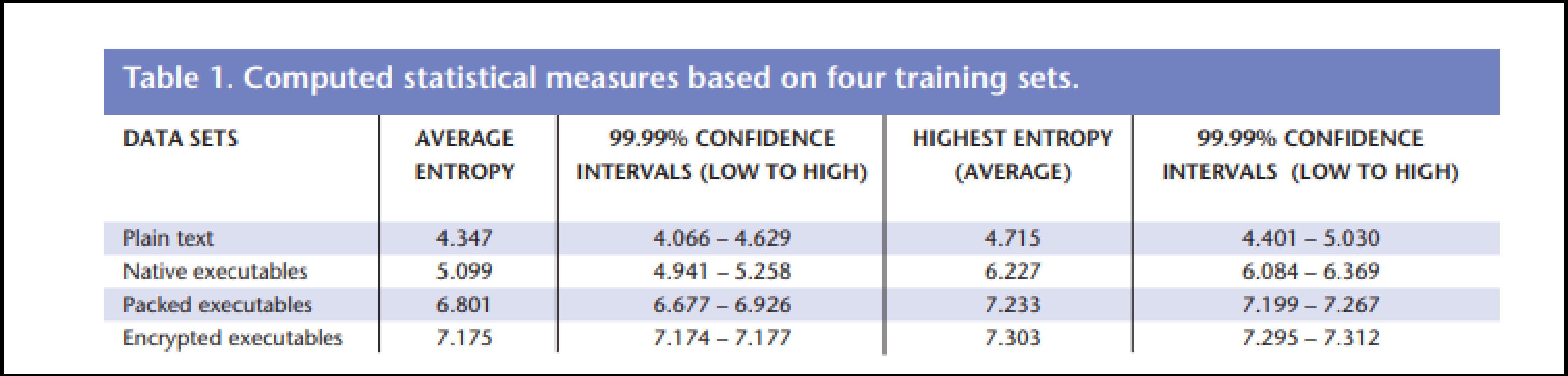


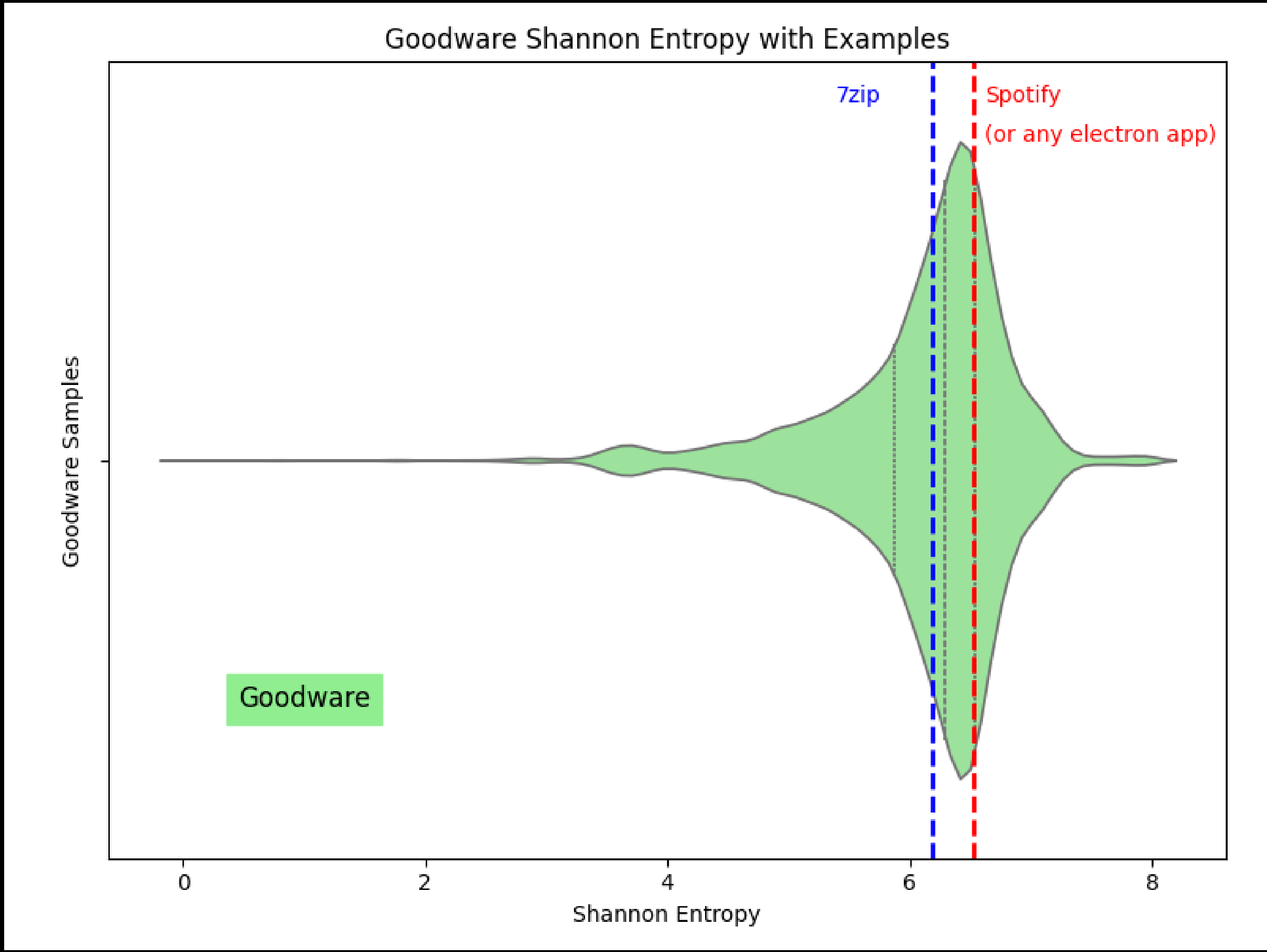
Table 1. Computed statistical measures based on four training sets.

DATA SETS	AVERAGE ENTROPY	99.99% CONFIDENCE INTERVALS (LOW TO HIGH)	HIGHEST ENTROPY (AVERAGE)	99.99% CONFIDENCE INTERVALS (LOW TO HIGH)
Plain text	4.347	4.066 – 4.629	4.715	4.401 – 5.030
Native executables	5.099	4.941 – 5.258	6.227	6.084 – 6.369
Packed executables	6.801	6.677 – 6.926	7.233	7.199 – 7.267
Encrypted executables	7.175	7.174 – 7.177	7.303	7.295 – 7.312

Using Entropy Analysis to Find Encrypted and Packed Malware

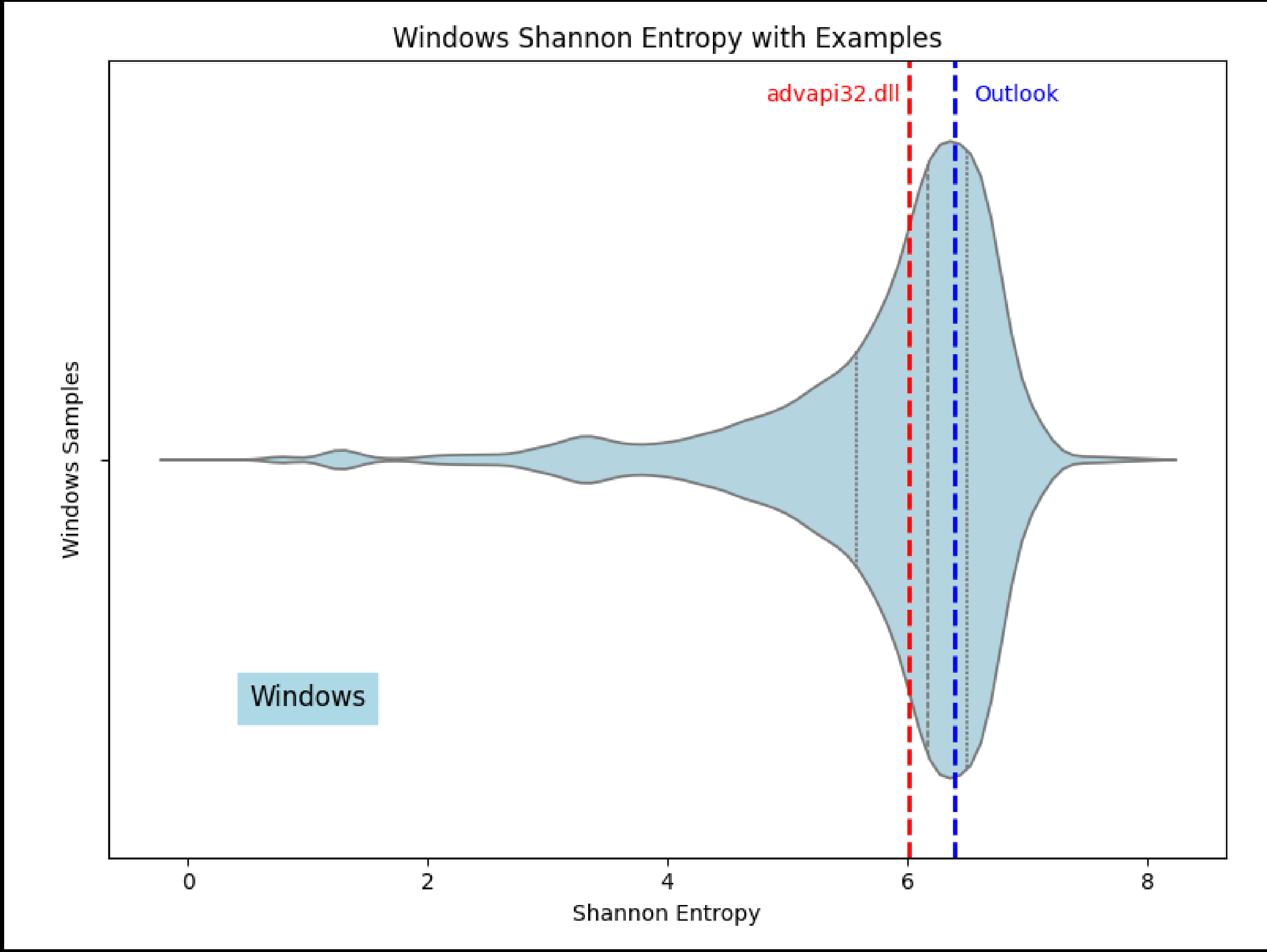
$ Entropy: Goodware

- Interquartile Range:
  - 25%: 5.87
  - 50% (mean): 6.30
  - 75%: 6.54



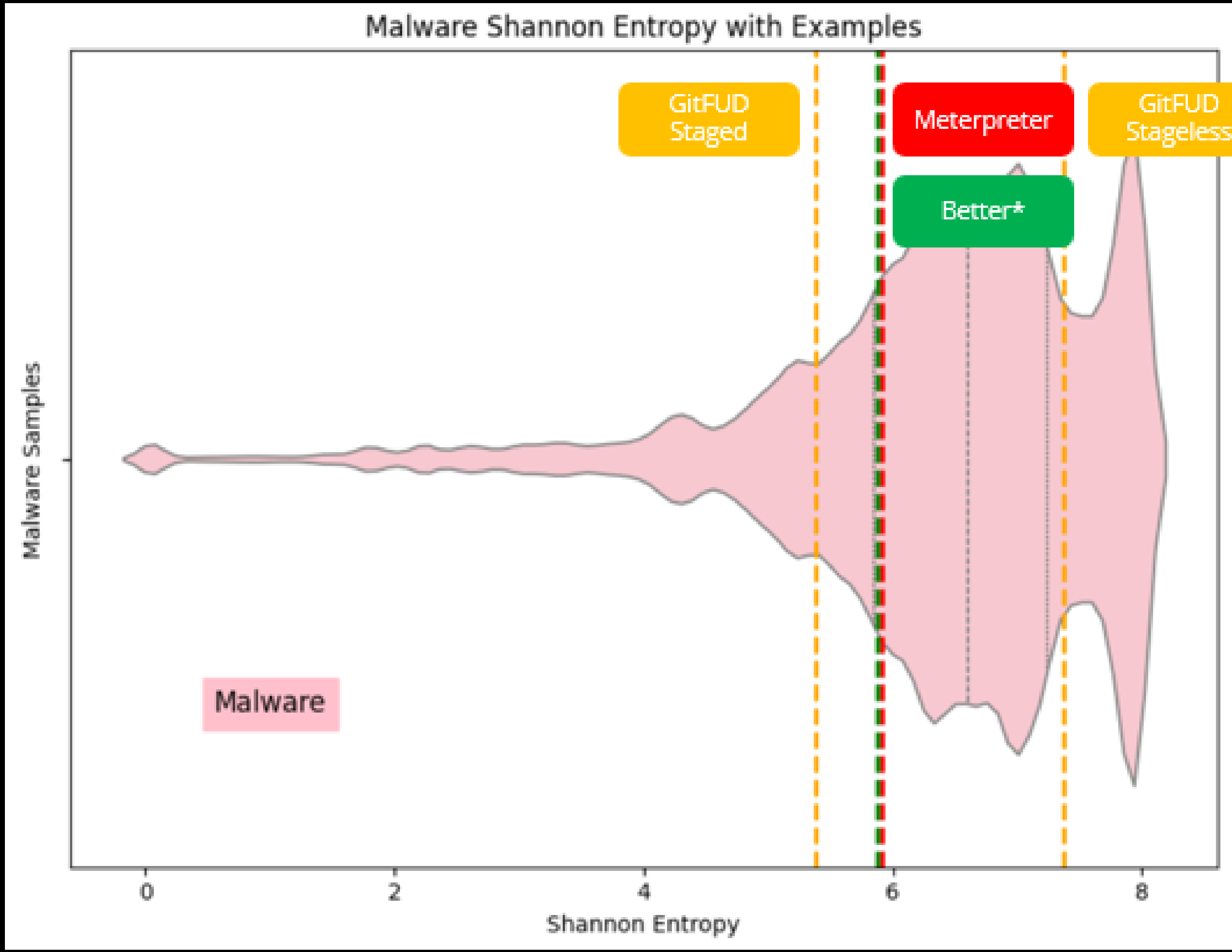
$ Entropy: Winware

- Interquartile Range:
  - 25%: 5.58
  - 50% (mean): 6.16
  - 75%: 6.50



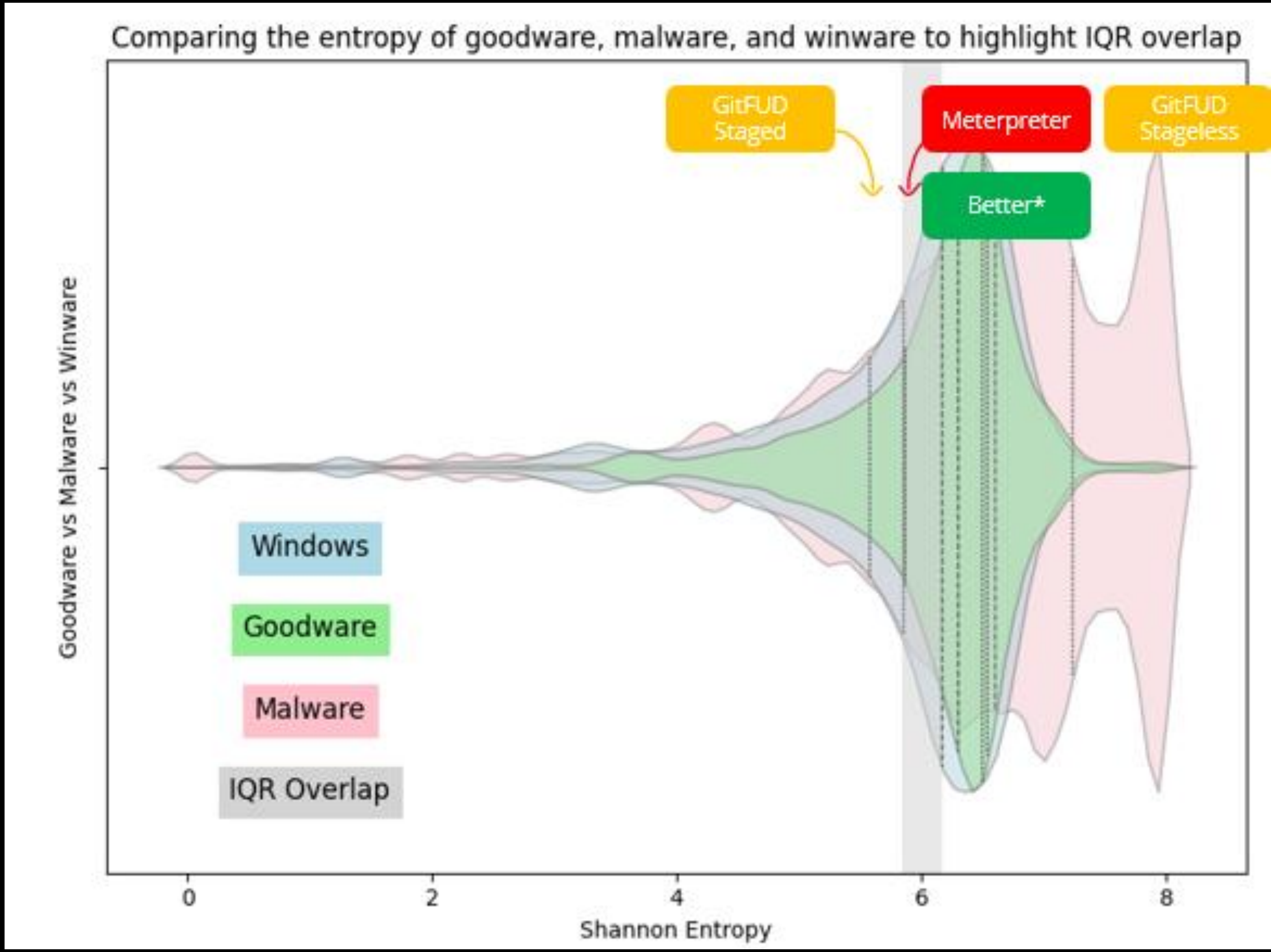
$ Entropy: Malware

- Interquartile Range:
  - 25%: 5.85
  - 50% (mean): 6.60
  - 75%: 7.24



$ Entropy: All together now!

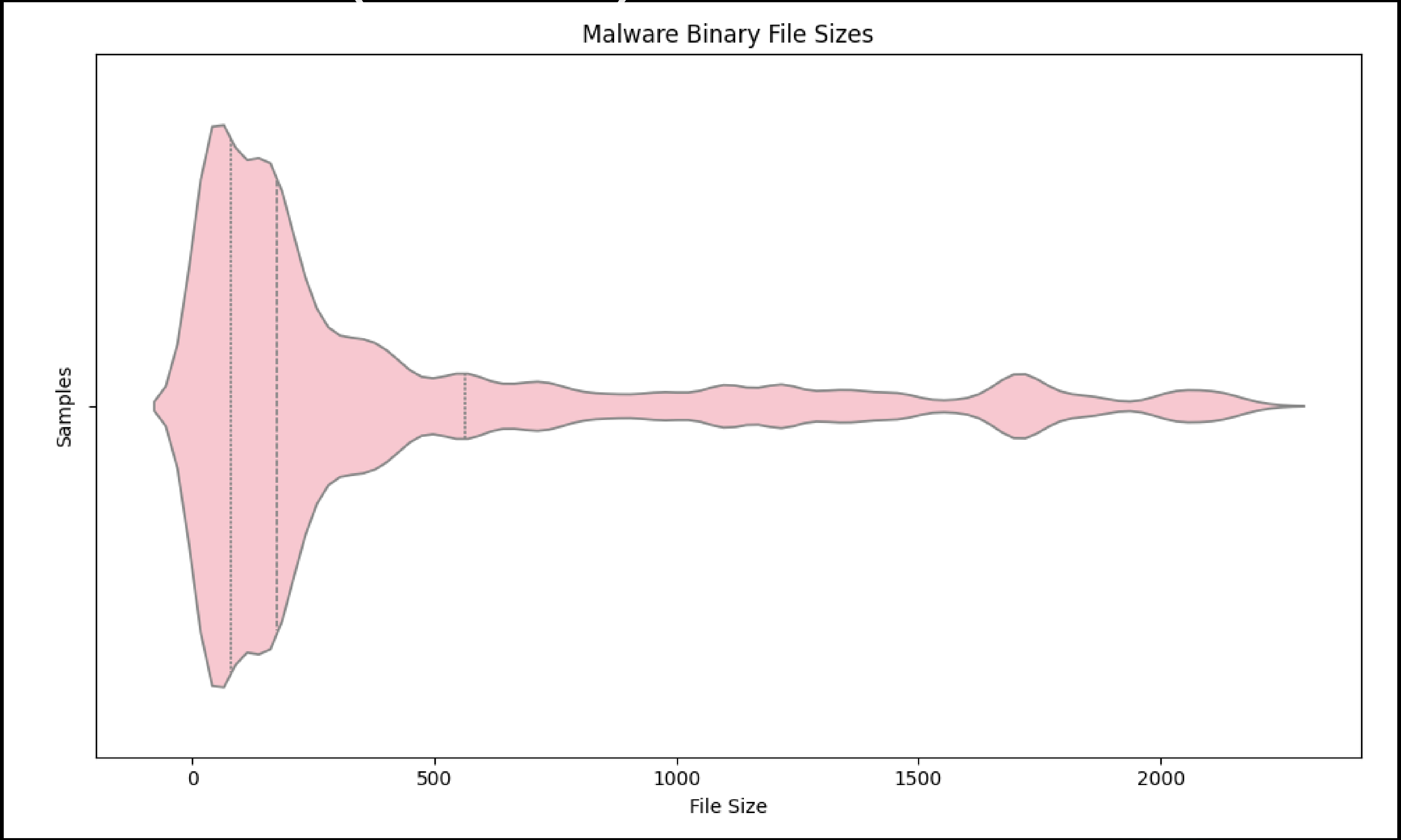
IQR	Malware	Goodware	Winware
25	5.85	5.87	5.58
50	6.6	6.3	6.16
75	7.24	6.54	6.5



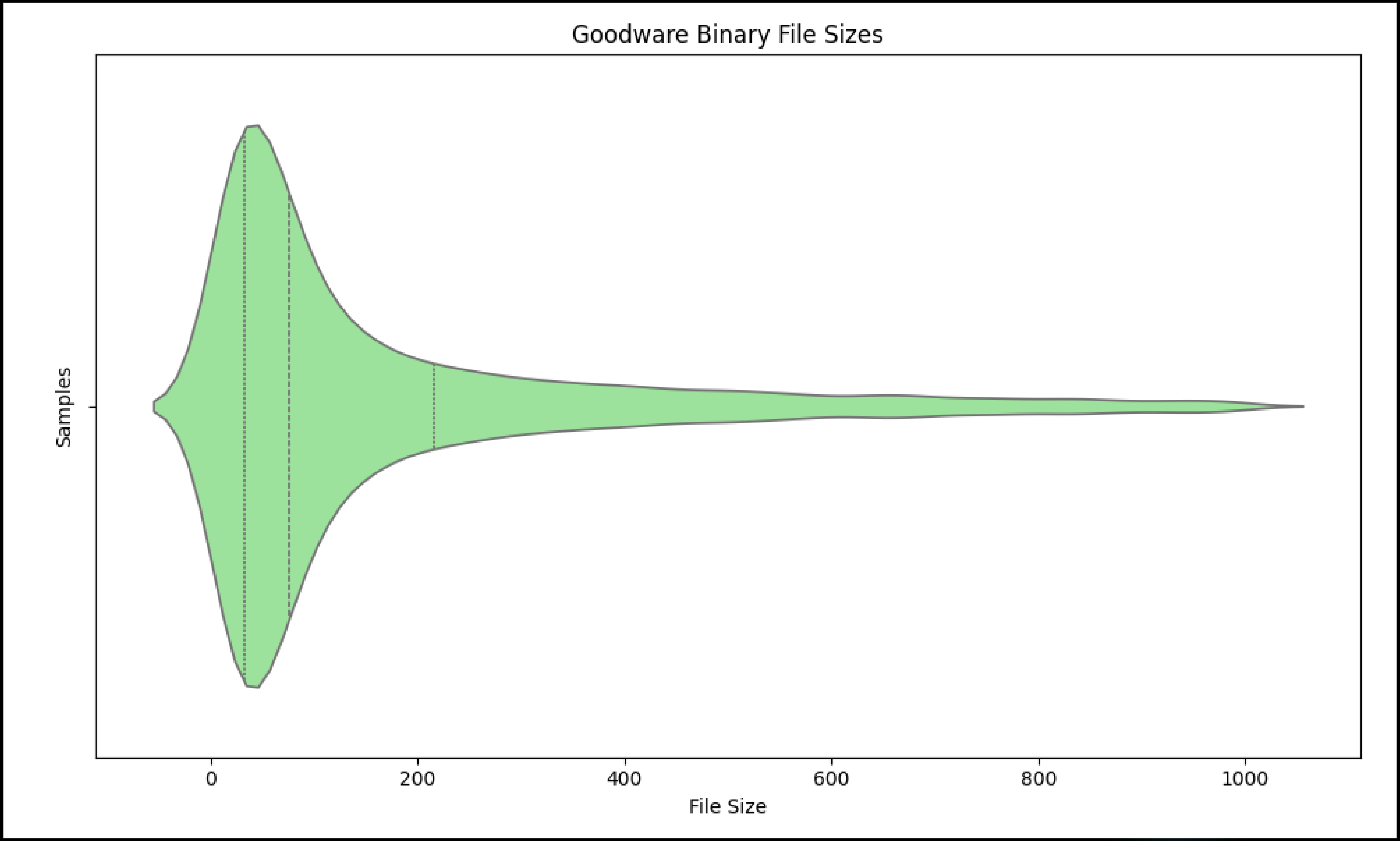
$ Solution 1 (Bonus): File Size

- A natural progression from Entropy
- The size of the binary CAN have implications
- Bloating tools are double edged:
  - Too large to scan
  - Looks super weird

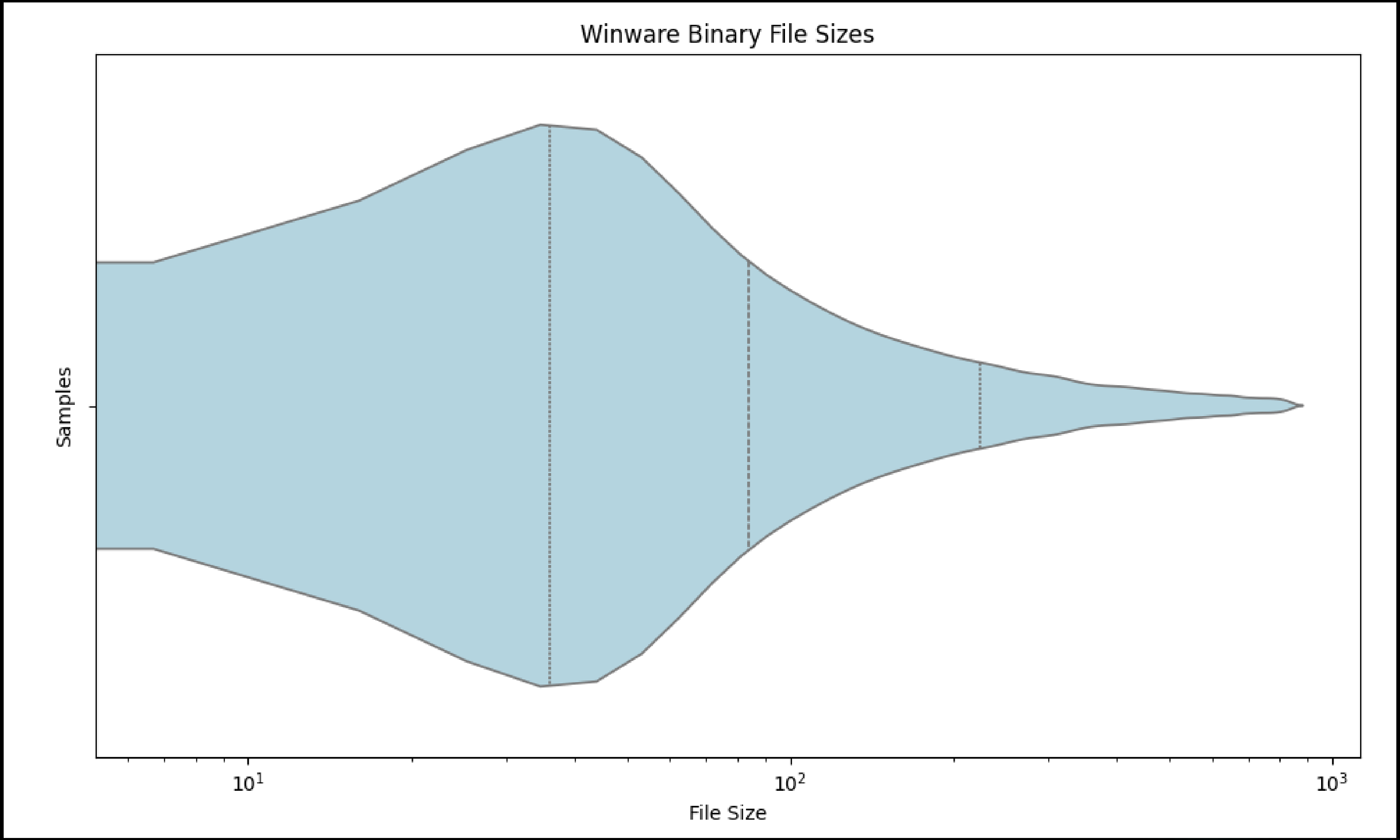
$ Solution 1 (Bonus): File Size



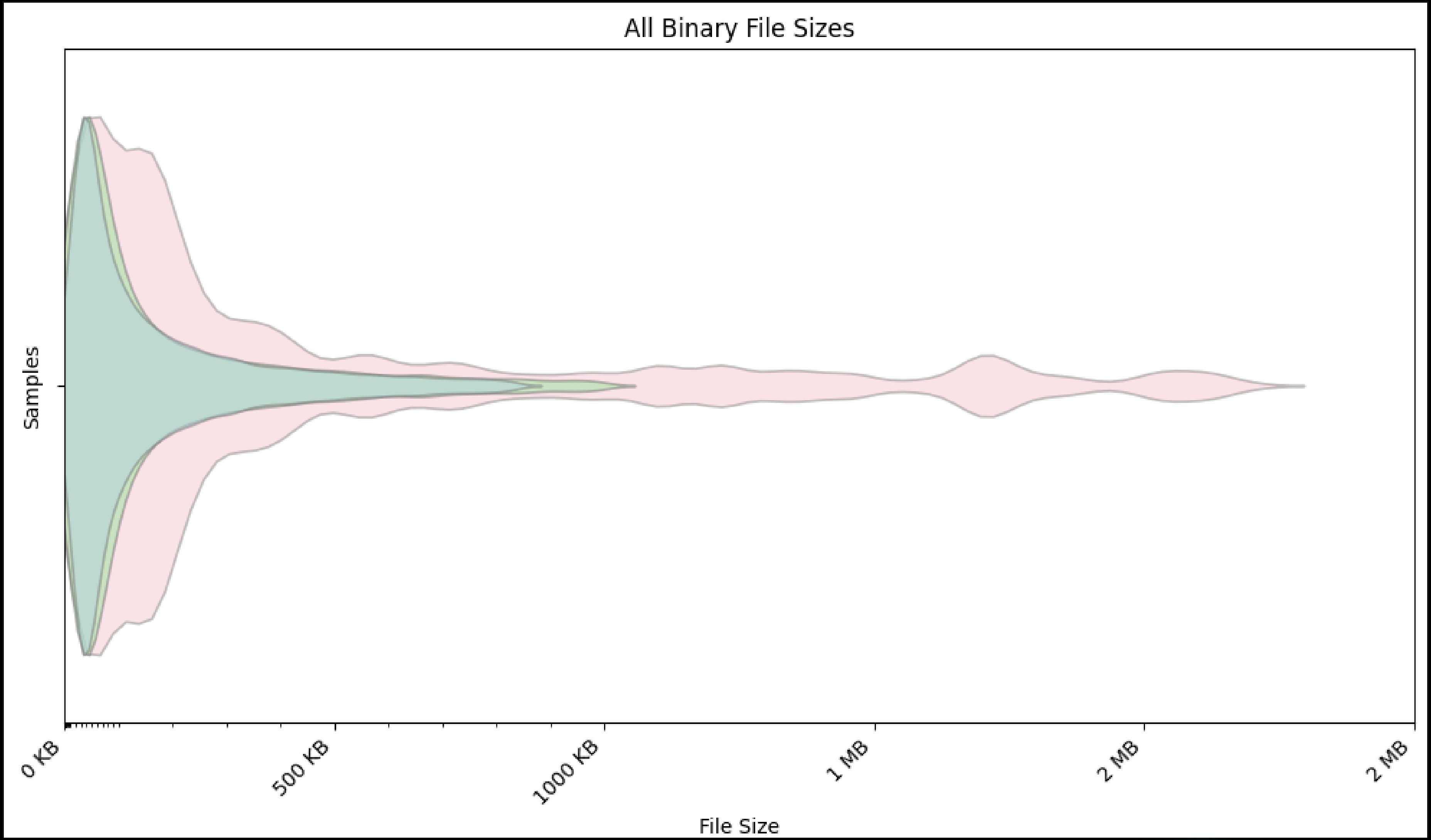
$ Solution 1 (Bonus): File Size



$ Solution 1 (Bonus): File Size



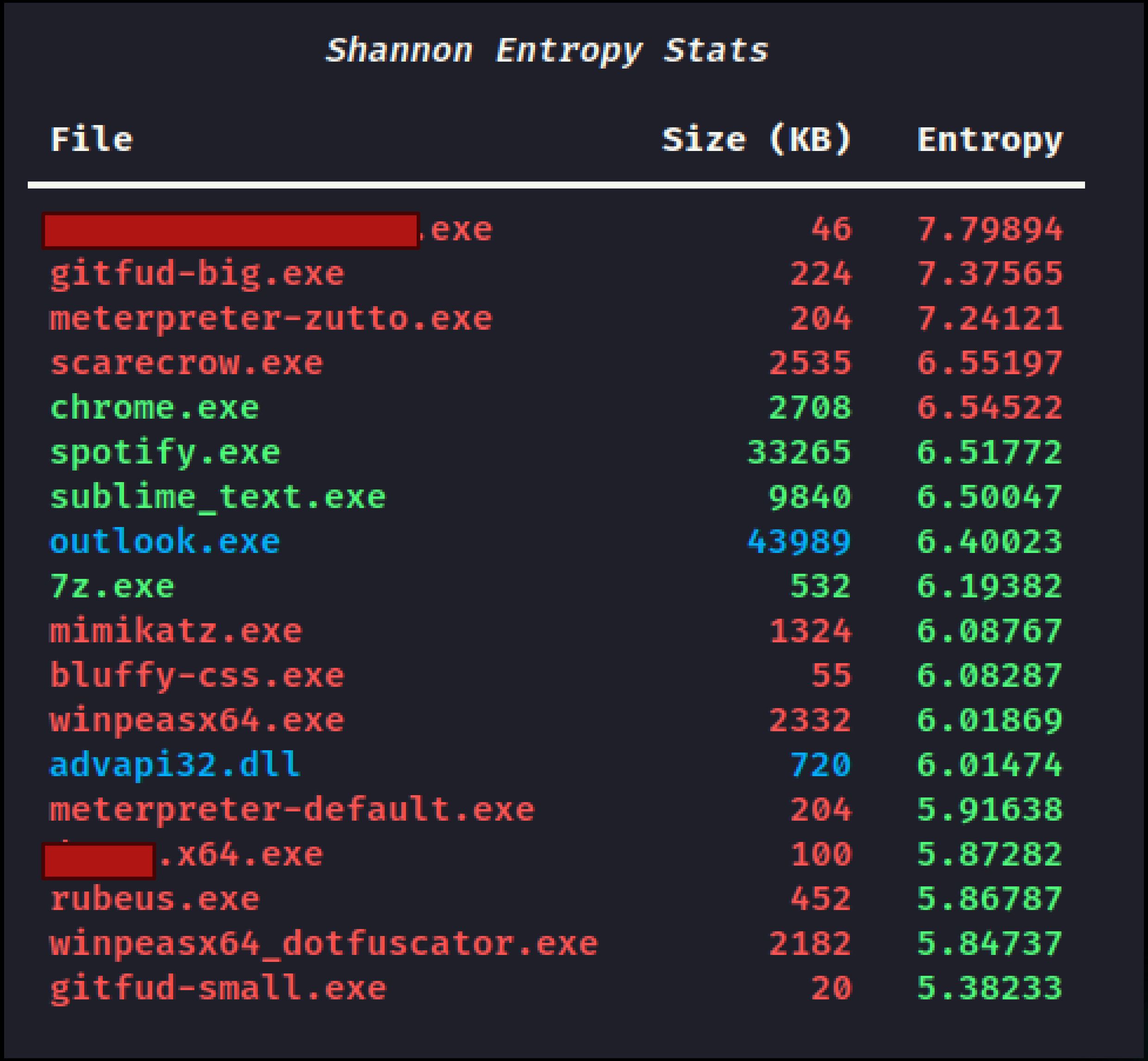
$ Solution 1 (Bonus): File Size



$ Solution 1 (Bonus): File Size

IQR (%)	Malware (KB)	Goodware (KB)	Winware (KB)
25	88	40	40
50	212	100	112
75	940	424	360

$ Solution 1 (Bonus): File Size



```
                  Shannon Entropy Stats

 File                              Size (KB)   Entropy
─────────────────────────────────────────────────────
 [redacted].exe                           46   7.79894
 gitfud-big.exe                          224   7.37565
 meterpreter-zutto.exe                   204   7.24121
 scarecrow.exe                          2535   6.55197
 chrome.exe                             2708   6.54522
 spotify.exe                           33265   6.51772
 sublime_text.exe                       9840   6.50047
 outlook.exe                           43989   6.40023
 7z.exe                                  532   6.19382
 mimikatz.exe                           1324   6.08767
 bluffy-css.exe                           55   6.08287
 winpeasx64.exe                         2332   6.01869
 advapi32.dll                            720   6.01474
 meterpreter-default.exe                 204   5.91638
 [redacted].x64.exe                      100   5.87282
 rubeus.exe                              452   5.86787
 winpeasx64_dotfuscator.exe             2182   5.84737
 gitfud-small.exe                         20   5.38233
```

$ Solution 1: Considerations

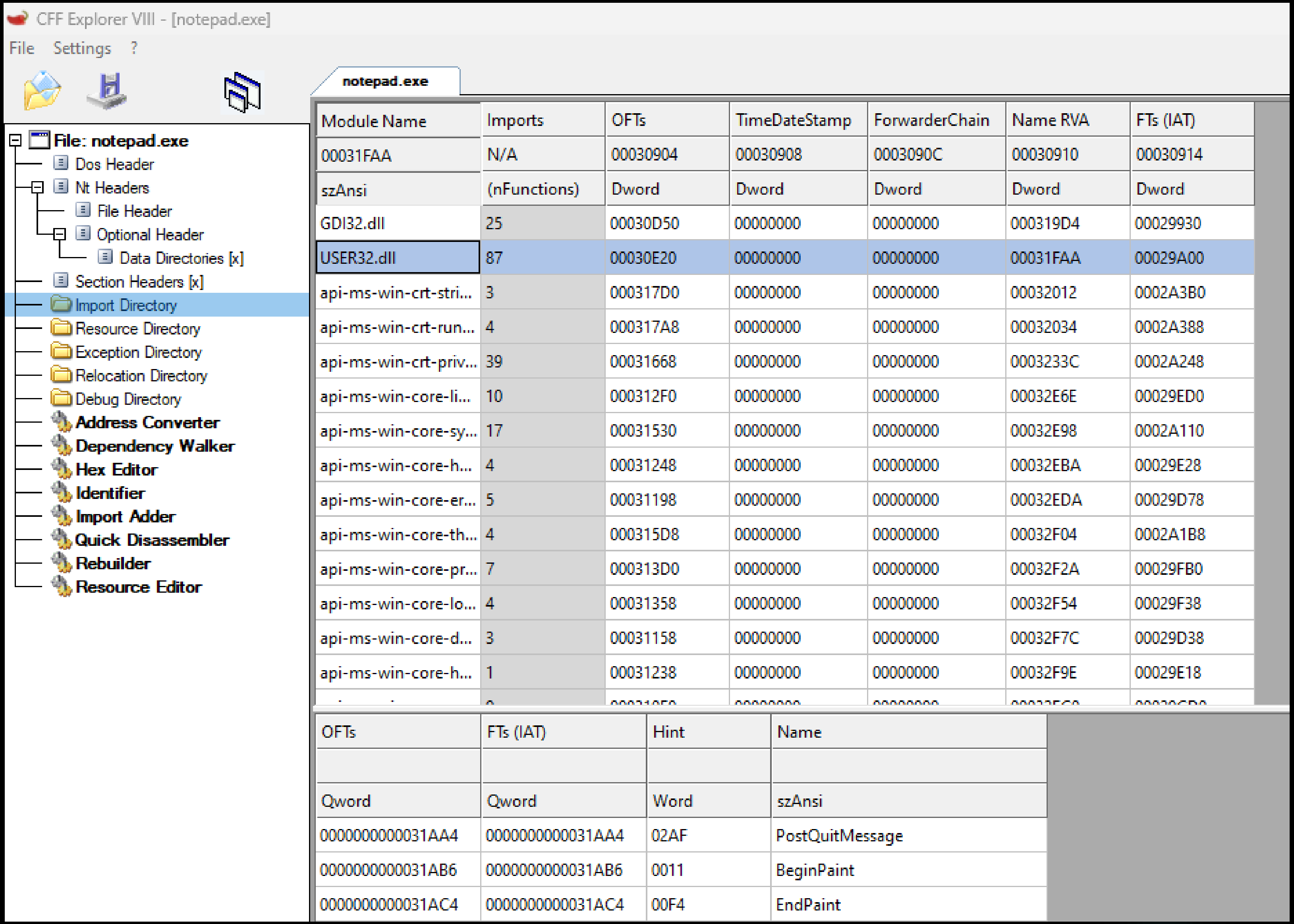
- Don’t put the high entropy blob inside the binary:
  - Hinder Reverse Engineers
  - Kill-switch
- Transform the blob into something like:
  - UUID
  - MAC
  - IPV6
  - SVG
  - CSS
  - PNG Bytes

pre.empt: Bluffy
Caleb Fenton: SentinelOne: Detecting Malware Pre-execution with Static Analysis and Machine Learning
Kurt Baker: CrowdStrike: 10 Malware Detection Techniques

$ Solution 2: Imports

- Sometimes it works, sometimes it doesn’t
- Back to Discord
- …
- *Get called noob*
- …
- “You should dynamically resolve WinAPI Functions”

$ Solution 2: Imports

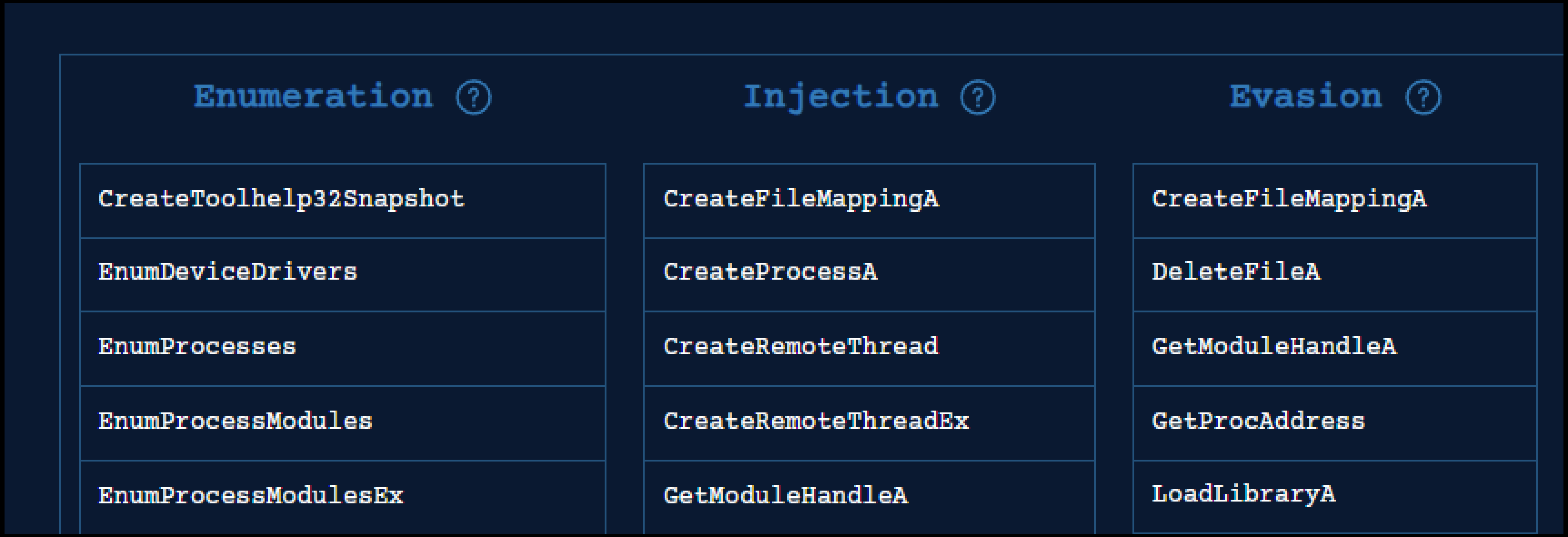


Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
00031FAA	N/A	00030904	00030908	0003090C	00030910	00030914
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
GDI32.dll	25	00030D50	00000000	00000000	000319D4	00029930
USER32.dll	87	00030E20	00000000	00000000	00031FAA	00029A00
api-ms-win-crt-stri...	3	000317D0	00000000	00000000	00032012	0002A3B0
api-ms-win-crt-run...	4	000317A8	00000000	00000000	00032034	0002A388
api-ms-win-crt-priv...	39	00031668	00000000	00000000	0003233C	0002A248
api-ms-win-core-li...	10	000312F0	00000000	00000000	00032E6E	00029ED0
api-ms-win-core-sy...	17	00031530	00000000	00000000	00032E98	0002A110
api-ms-win-core-h...	4	00031248	00000000	00000000	00032EBA	00029E28
api-ms-win-core-er...	5	00031198	00000000	00000000	00032EDA	00029D78
api-ms-win-core-th...	4	000315D8	00000000	00000000	00032F04	0002A1B8
api-ms-win-core-pr...	7	000313D0	00000000	00000000	00032F2A	00029FB0
api-ms-win-core-lo...	4	00031358	00000000	00000000	00032F54	00029F38
api-ms-win-core-d...	3	00031158	00000000	00000000	00032F7C	00029D38
api-ms-win-core-h...	1	00031238	00000000	00000000	00032F9E	00029E18

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
0000000000031AA4	0000000000031AA4	02AF	PostQuitMessage
0000000000031AB6	0000000000031AB6	0011	BeginPaint
0000000000031AC4	0000000000031AC4	00F4	EndPaint

$ Solution 2: Imports

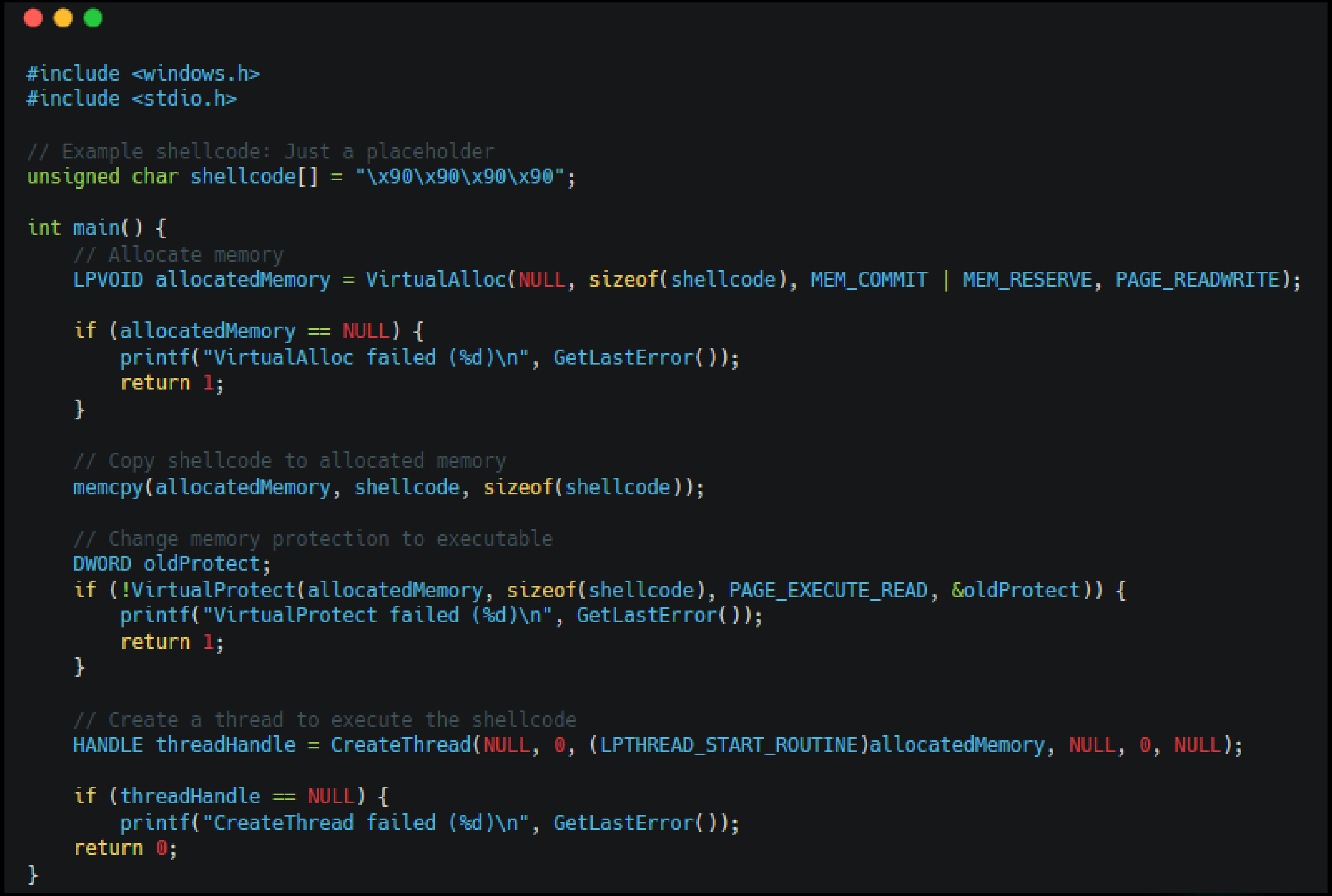
- Common Injection:
  - VirtualAlloc
  - Memcpy
  - VirtualProtect
  - CreateThread
  - WaitForSingleObject



Enumeration ?	Injection ?	Evasion ?
CreateToolhelp32Snapshot	CreateFileMappingA	CreateFileMappingA
EnumDeviceDrivers	CreateProcessA	DeleteFileA
EnumProcesses	CreateRemoteThread	GetModuleHandleA
EnumProcessModules	CreateRemoteThreadEx	GetProcAddress
EnumProcessModulesEx	GetModuleHandleA	LoadLibraryA

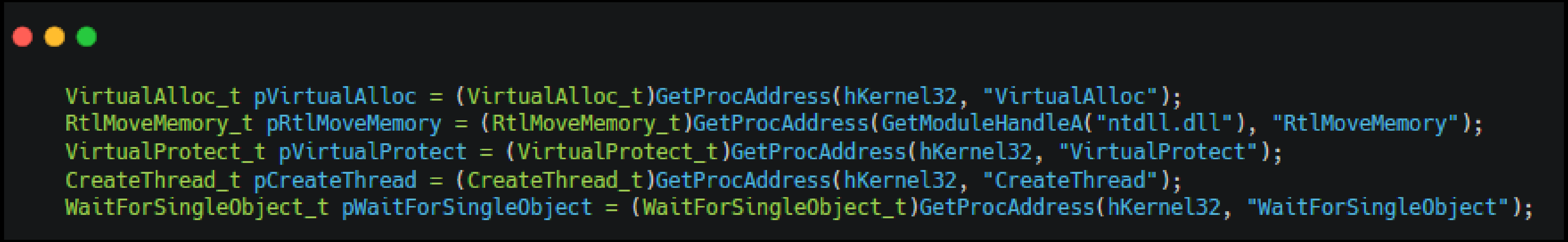
malapi.io

$ Solution 2: Imports



```c
#include <windows.h>
#include <stdio.h>

// Example shellcode: Just a placeholder
unsigned char shellcode[] = "\x90\x90\x90\x90";

int main() {
    // Allocate memory
    LPVOID allocatedMemory = VirtualAlloc(NULL, sizeof(shellcode), MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

    if (allocatedMemory == NULL) {
        printf("VirtualAlloc failed (%d)\n", GetLastError());
        return 1;
    }

    // Copy shellcode to allocated memory
    memcpy(allocatedMemory, shellcode, sizeof(shellcode));

    // Change memory protection to executable
    DWORD oldProtect;
    if (!VirtualProtect(allocatedMemory, sizeof(shellcode), PAGE_EXECUTE_READ, &oldProtect)) {
        printf("VirtualProtect failed (%d)\n", GetLastError());
        return 1;
    }

    // Create a thread to execute the shellcode
    HANDLE threadHandle = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)allocatedMemory, NULL, 0, NULL);

    if (threadHandle == NULL) {
        printf("CreateThread failed (%d)\n", GetLastError());
    return 0;
}
```

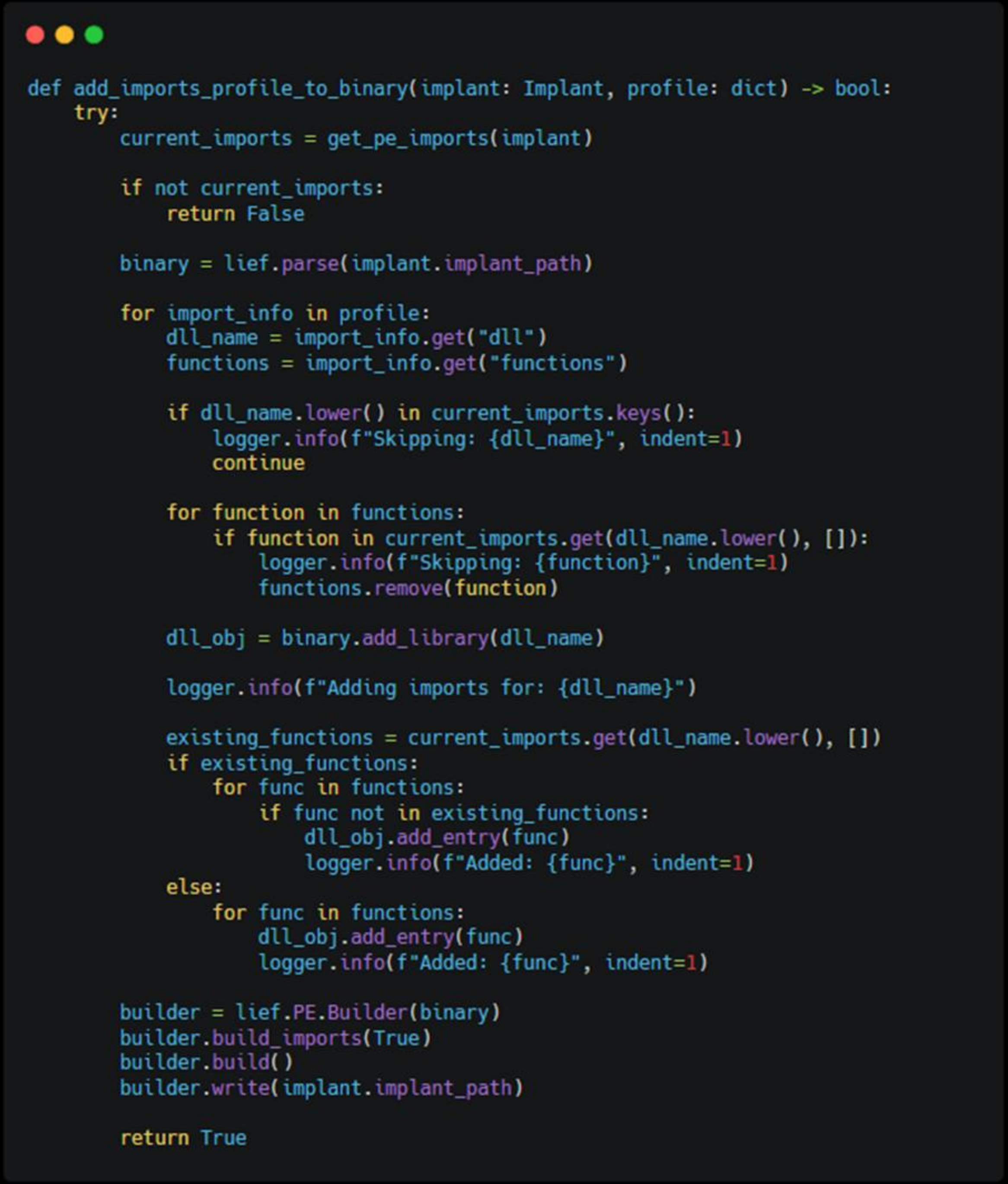
$ Solution 2: Imports



```
    VirtualAlloc_t pVirtualAlloc = (VirtualAlloc_t)GetProcAddress(hKernel32, "VirtualAlloc");
    RtlMoveMemory_t pRtlMoveMemory = (RtlMoveMemory_t)GetProcAddress(GetModuleHandleA("ntdll.dll"), "RtlMoveMemory");
    VirtualProtect_t pVirtualProtect = (VirtualProtect_t)GetProcAddress(hKernel32, "VirtualProtect");
    CreateThread_t pCreateThread = (CreateThread_t)GetProcAddress(hKernel32, "CreateThread");
    WaitForSingleObject_t pWaitForSingleObject = (WaitForSingleObject_t)GetProcAddress(hKernel32, "WaitForSingleObject");
```

$ Solution 2: Considera

- Resolve some functions and not others
- Fake some imports
- Add telemetry generation



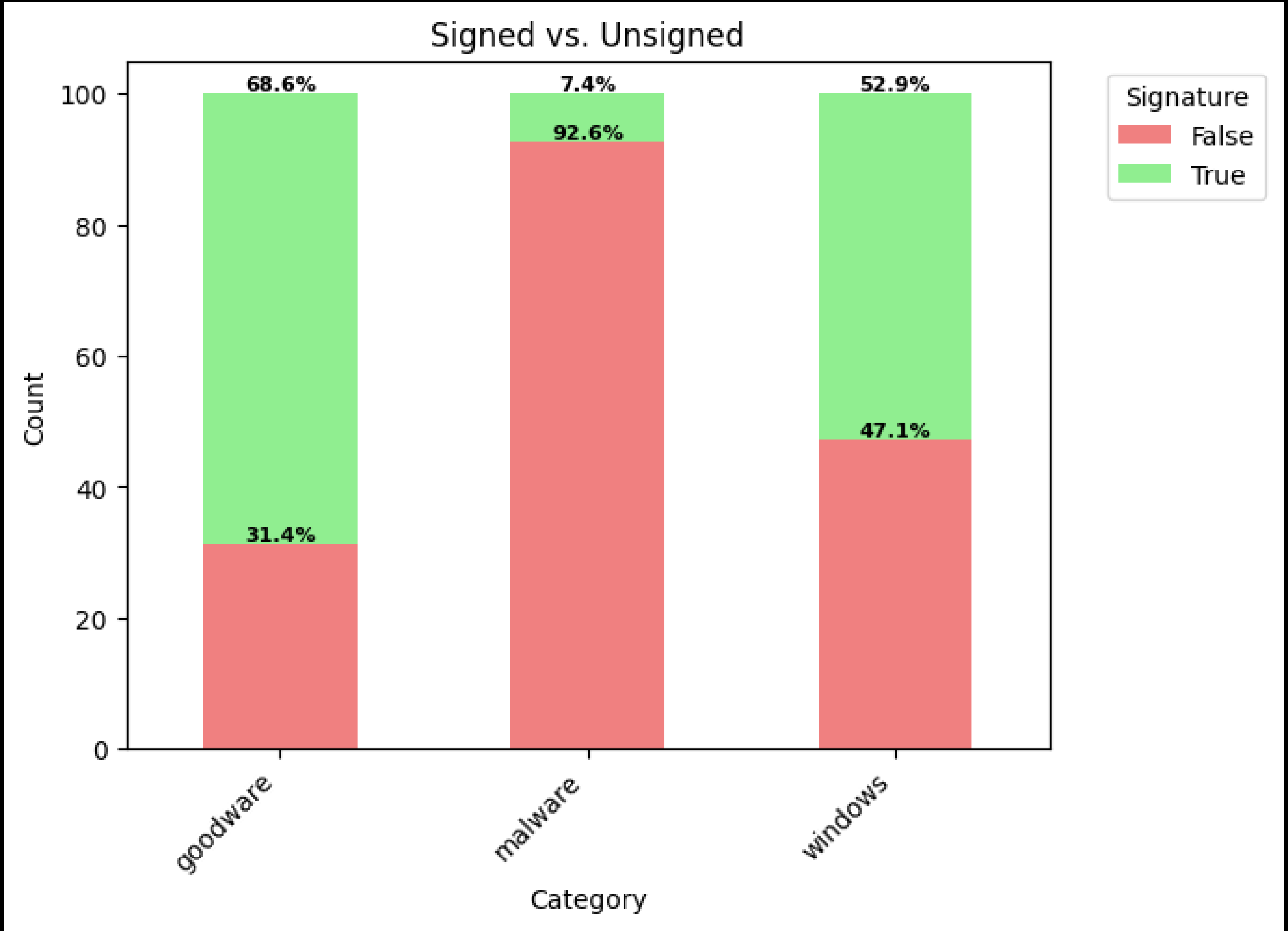
```python
def add_imports_profile_to_binary(implant: Implant, profile: dict) -> bool:
    try:
        current_imports = get_pe_imports(implant)

        if not current_imports:
            return False

        binary = lief.parse(implant.implant_path)

        for import_info in profile:
            dll_name = import_info.get("dll")
            functions = import_info.get("functions")

            if dll_name.lower() in current_imports.keys():
                logger.info(f"Skipping: {dll_name}", indent=1)
                continue

            for function in functions:
                if function in current_imports.get(dll_name.lower(), []):
                    logger.info(f"Skipping: {function}", indent=1)
                    functions.remove(function)

            dll_obj = binary.add_library(dll_name)

            logger.info(f"Adding imports for: {dll_name}")

            existing_functions = current_imports.get(dll_name.lower(), [])
            if existing_functions:
                for func in functions:
                    if func not in existing_functions:
                        dll_obj.add_entry(func)
                        logger.info(f"Added: {func}", indent=1)
            else:
                for func in functions:
                    dll_obj.add_entry(func)
                    logger.info(f"Added: {func}", indent=1)

        builder = lief.PE.Builder(binary)
        builder.build_imports(True)
        builder.build()
        builder.write(implant.implant_path)

        return True
```

$ Solution 2: Exports (bonus)

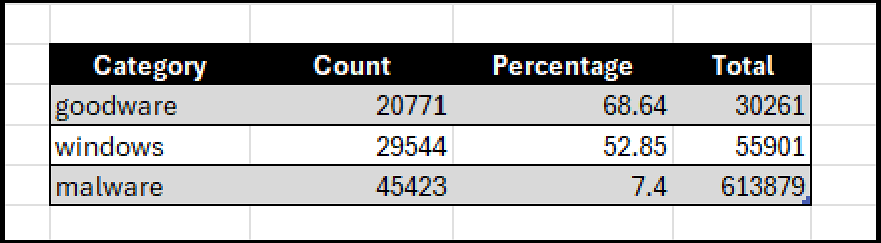
$ Solution 3: Code Signing

- So, you’ve fixed the imports and exports
- You’ve built your new implant
- Securely compiled, non-standard, etc.
- So, you add the final 20% of effort, the coup de grace, and implement the special features people talk about on the walkthroughs

$ Signing

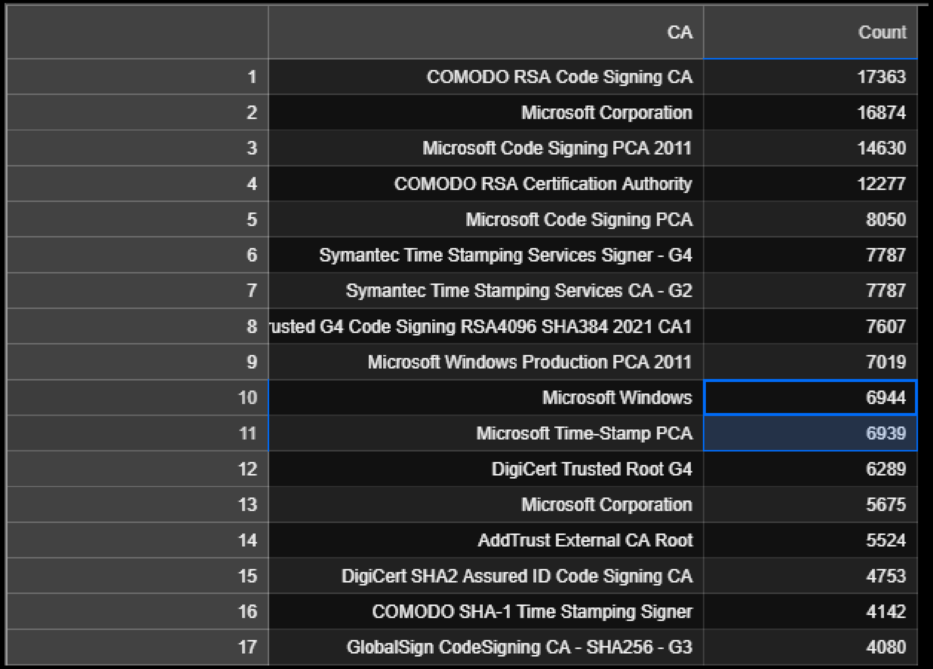

Signed vs. Unsigned
Count
Category
goodware
malware
windows
0
20
40
60
80
100
68.6%
31.4%
7.4%
92.6%
52.9%
47.1%
Signature
🟥 False
🟩 True

$ Signing



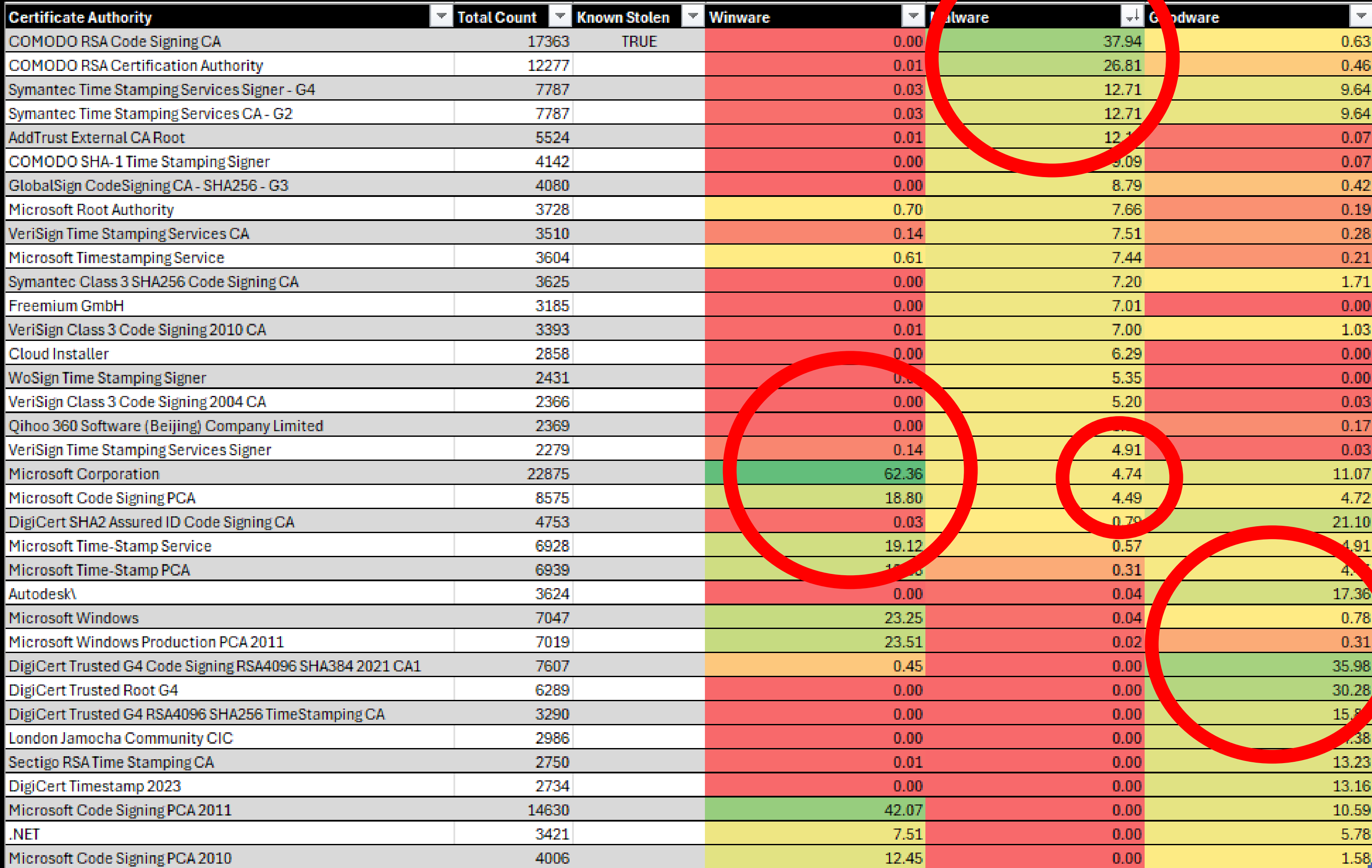
Category	Count	Percentage	Total
goodware	20771	68.64	30261
windows	29544	52.85	55901
malware	45423	7.4	613879

	CA	Malware (%)	Goodware (%)	Winware (%)



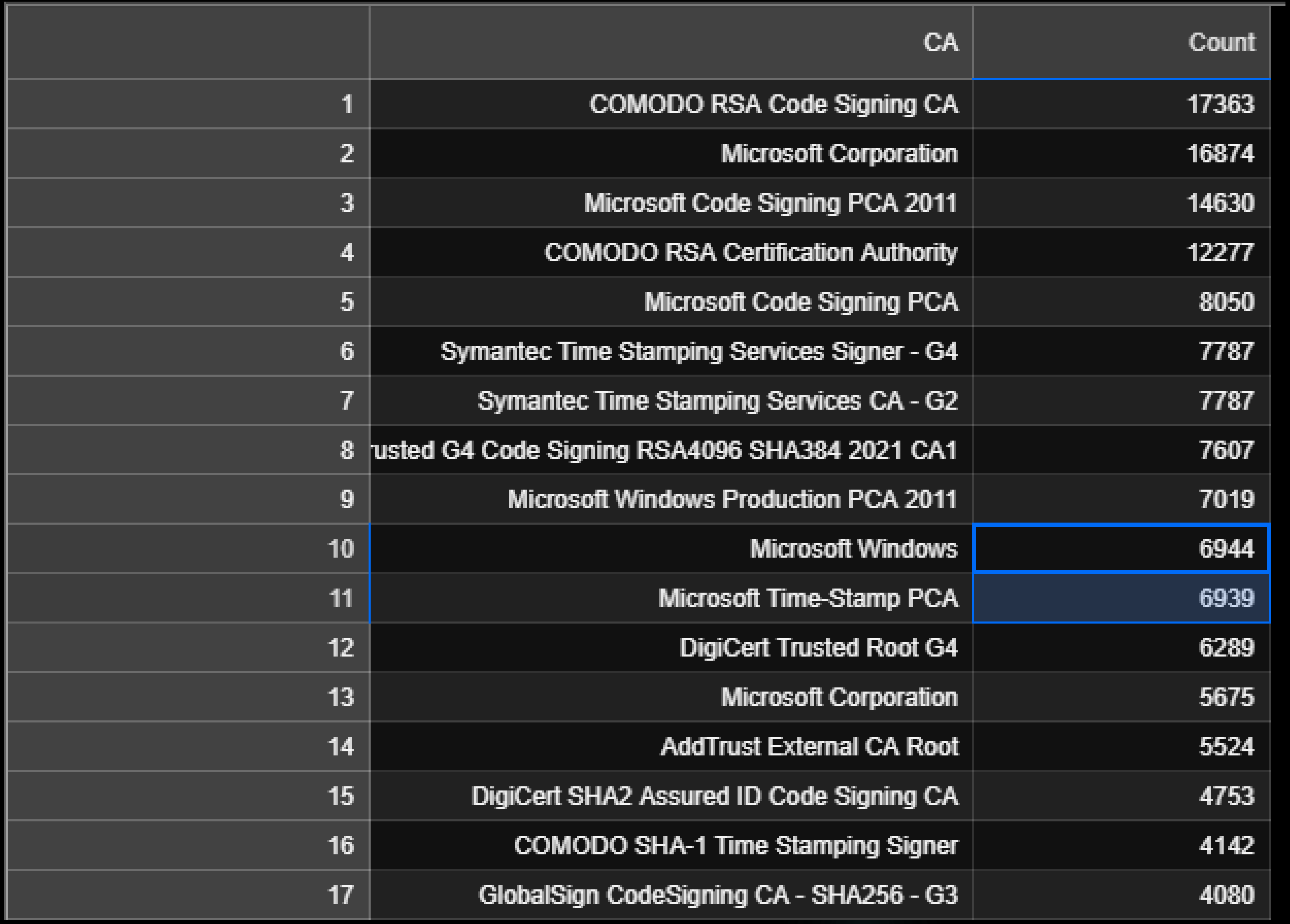
	CA	Count
1	COMODO RSA Code Signing CA	17363
2	Microsoft Corporation	16874
3	Microsoft Code Signing PCA 2011	14630
4	COMODO RSA Certification Authority	12277
5	Microsoft Code Signing PCA	8050
6	Symantec Time Stamping Services Signer - G4	7787
7	Symantec Time Stamping Services CA - G2	7787
8	usted G4 Code Signing RSA4096 SHA384 2021 CA1	7607
9	Microsoft Windows Production PCA 2011	7019
10	Microsoft Windows	6944
11	Microsoft Time-Stamp PCA	6939
12	DigiCert Trusted Root G4	6289
13	Microsoft Corporation	5675
14	AddTrust External CA Root	5524
15	DigiCert SHA2 Assured ID Code Signing CA	4753
16	COMODO SHA-1 Time Stamping Signer	4142
17	GlobalSign CodeSigning CA - SHA256 - G3	4080

$ Signing



Certificate Authority	Total Count	Known Stolen	Winware	Malware	Goodware
COMODO RSA Code Signing CA	17363	TRUE	0.00	37.94	0.63
COMODO RSA Certification Authority	12277		0.01	26.81	0.46
Symantec Time Stamping Services Signer - G4	7787		0.03	12.71	9.64
Symantec Time Stamping Services CA - G2	7787		0.03	12.71	9.64
AddTrust External CA Root	5524		0.01	[illegible]	0.07
COMODO SHA-1 Time Stamping Signer	4142		0.00	9.09	0.07
GlobalSign CodeSigning CA - SHA256 - G3	4080		0.00	8.79	0.42
Microsoft Root Authority	3728		0.70	7.66	0.19
VeriSign Time Stamping Services CA	3510		0.14	7.51	0.28
Microsoft Timestamping Service	3604		0.61	7.44	0.21
Symantec Class 3 SHA256 Code Signing CA	3625		0.00	7.20	1.71
Freemium GmbH	3185		0.00	7.01	0.00
VeriSign Class 3 Code Signing 2010 CA	3393		0.01	7.00	1.03
Cloud Installer	2858		0.00	6.29	0.00
WoSign Time Stamping Signer	2431		[illegible]	5.35	0.00
VeriSign Class 3 Code Signing 2004 CA	2366		0.00	5.20	0.03
Qihoo 360 Software (Beijing) Company Limited	2369		0.00	[illegible]	0.17
VeriSign Time Stamping Services Signer	2279		0.14	4.91	0.03
Microsoft Corporation	22875		62.36	4.74	11.07
Microsoft Code Signing PCA	8575		18.80	4.49	4.72
DigiCert SHA2 Assured ID Code Signing CA	4753		0.03	0.79	21.10
Microsoft Time-Stamp Service	6928		19.12	0.57	4.91
Microsoft Time-Stamp PCA	6939		[illegible]	0.31	[illegible]
Autodesk\	3624		0.00	0.04	17.36
Microsoft Windows	7047		23.25	0.04	0.78
Microsoft Windows Production PCA 2011	7019		23.51	0.02	0.31
DigiCert Trusted G4 Code Signing RSA4096 SHA384 2021 CA1	7607		0.45	0.00	35.98
DigiCert Trusted Root G4	6289		0.00	0.00	30.28
DigiCert Trusted G4 RSA4096 SHA256 TimeStamping CA	3290		0.00	0.00	[illegible]
London Jamocha Community CIC	2986		0.00	0.00	[illegible]
Sectigo RSA Time Stamping CA	2750		0.01	0.00	13.23
DigiCert Timestamp 2023	2734		0.00	0.00	13.16
Microsoft Code Signing PCA 2011	14630		42.07	0.00	10.59
.NET	3421		7.51	0.00	5.78
Microsoft Code Signing PCA 2010	4006		12.45	0.00	1.58

$ Signing



	CA	Count
1	COMODO RSA Code Signing CA	17363
2	Microsoft Corporation	16874
3	Microsoft Code Signing PCA 2011	14630
4	COMODO RSA Certification Authority	12277
5	Microsoft Code Signing PCA	8050
6	Symantec Time Stamping Services Signer - G4	7787
7	Symantec Time Stamping Services CA - G2	7787
8	usted G4 Code Signing RSA4096 SHA384 2021 CA1	7607
9	Microsoft Windows Production PCA 2011	7019
10	Microsoft Windows	6944
11	Microsoft Time-Stamp PCA	6939
12	DigiCert Trusted Root G4	6289
13	Microsoft Corporation	5675
14	AddTrust External CA Root	5524
15	DigiCert SHA2 Assured ID Code Signing CA	4753
16	COMODO SHA-1 Time Stamping Signer	4142
17	GlobalSign CodeSigning CA - SHA256 - G3	4080

$ Sections

Name	Content
.bss	Uninitialized data (free format)
.cormeta	CLR metadata that indicates that the object file contains managed code
.data	Initialized data (free format)
.debug$F	Precompiled debug types (object only)
.debug$P	Debug types (object only)
.drective	Linker options
.edata	Export tables
.idata	Import tables
.idlsym	Includes registered SEH (image only) to support IDL attributes. For information, see "IDL Attributes" in References at the end of this topic.
.pdata	Exception information
.rdata	Read-only initialized data
.reloc	Image relocations
.rsrc	Resource directory
.sbss	GP-relative uninitialized data (free format)
.sdata	GP-relative initialized data (free format)
.srdata	GP-relative read-only data (free format)
.sxdata	Registered exception handler data (free format and x86/object only)
.text	Executable code (free format)
.tls	Thread-local storage (object only)
.tls$	Thread-local storage (object only)
.vsdata	GP-relative initialized data (free format and for ARM, SH4, and Thumb architectures only)
.xdata	Exception information (free format)

MSDN: PE Format

$ Compiler

- ***In our experience***, compilers have changed the outcome of execution

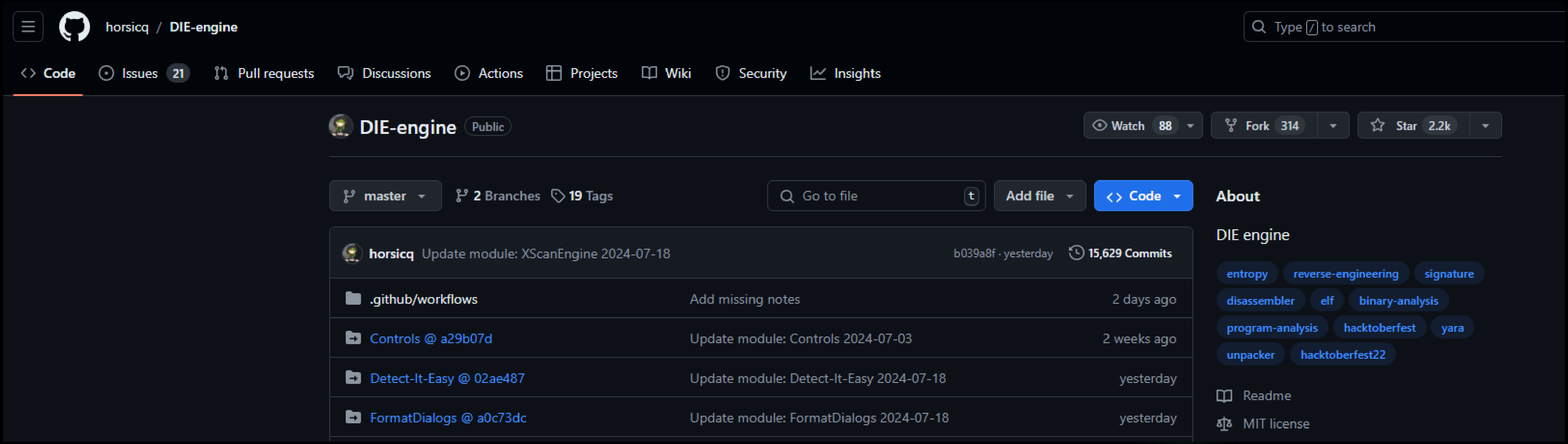
- Compiling with MINGW:
  - **Caught**

- Compiling with MSVC:
  - **Fine**

- Compiling with CLANG:
  - **Mixed**

$ Compiler

- This is a current limitation of the dataset
- *Detect it Easy* could be used to solve this problem



horsicq / DIE-engine

DIE-engine Public

master · 2 Branches · 19 Tags

horsicq Update module: XScanEngine 2024-07-18 · b039a8f · yesterday · 15,629 Commits

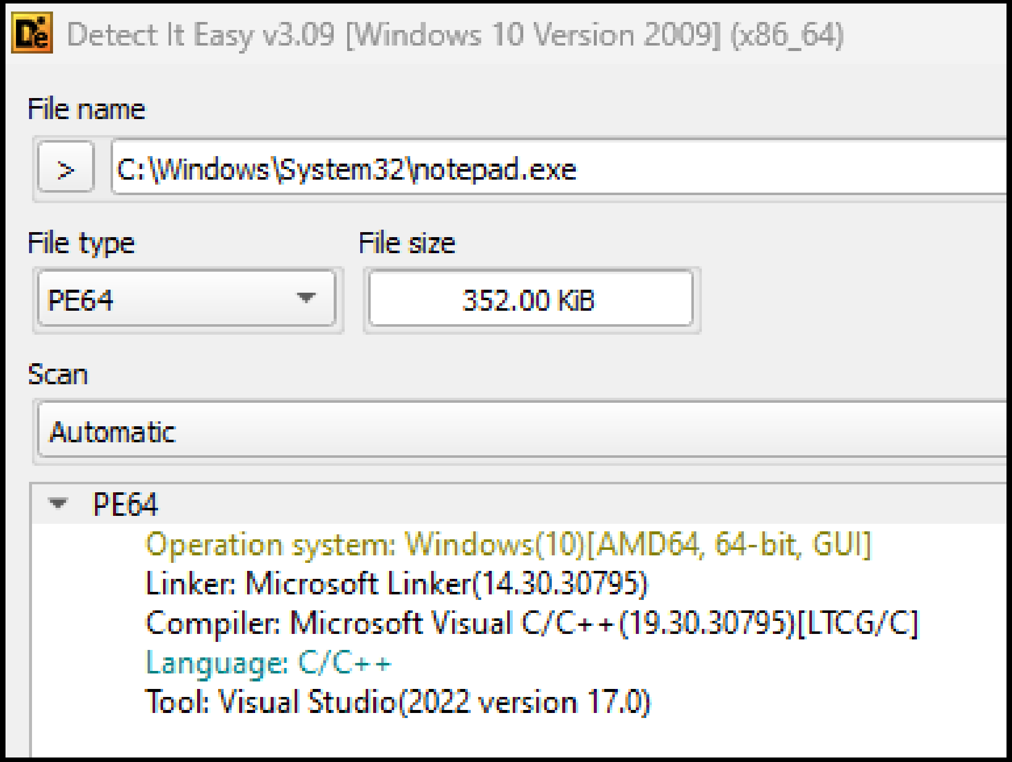
.github/workflows	Add missing notes	2 days ago
Controls @ a29b07d	Update module: Controls 2024-07-03	2 weeks ago
Detect-It-Easy @ 02ae487	Update module: Detect-It-Easy 2024-07-18	yesterday
FormatDialogs @ a0c73dc	Update module: FormatDialogs 2024-07-18	yesterday

About

DIE engine

entropy, reverse-engineering, signature, disassembler, elf, binary-analysis, program-analysis, hacktoberfest, yara, unpacker, hacktoberfest22

$ Compiler



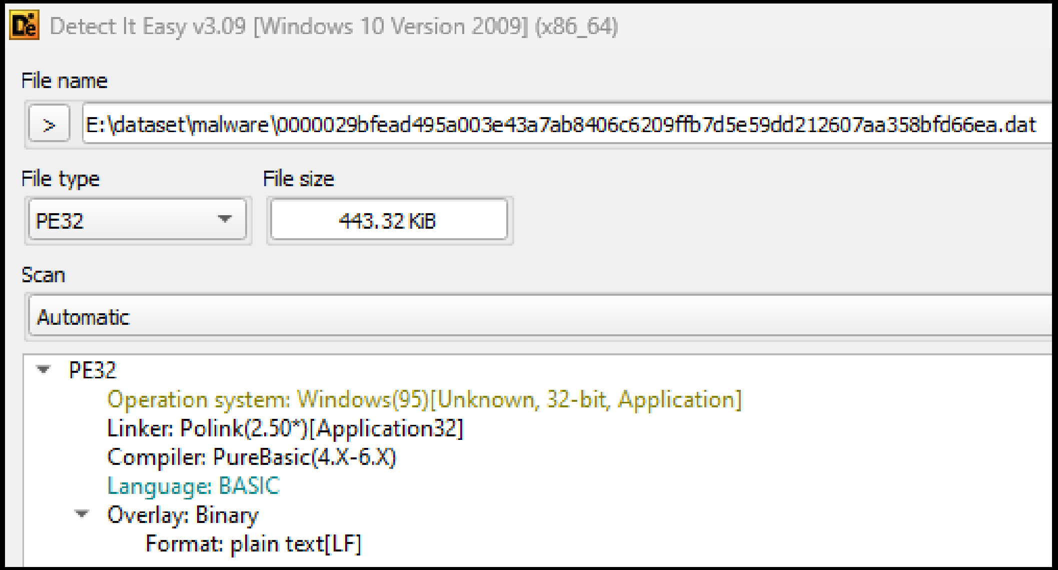
Detect It Easy v3.09 [Windows 10 Version 2009] (x86_64)

File name: C:\Windows\System32\notepad.exe

File type: PE64 | File size: 352.00 KiB

Scan: Automatic

```
PE64
    Operation system: Windows(10)[AMD64, 64-bit, GUI]
    Linker: Microsoft Linker(14.30.30795)
    Compiler: Microsoft Visual C/C++(19.30.30795)[LTCG/C]
    Language: C/C++
    Tool: Visual Studio(2022 version 17.0)
```



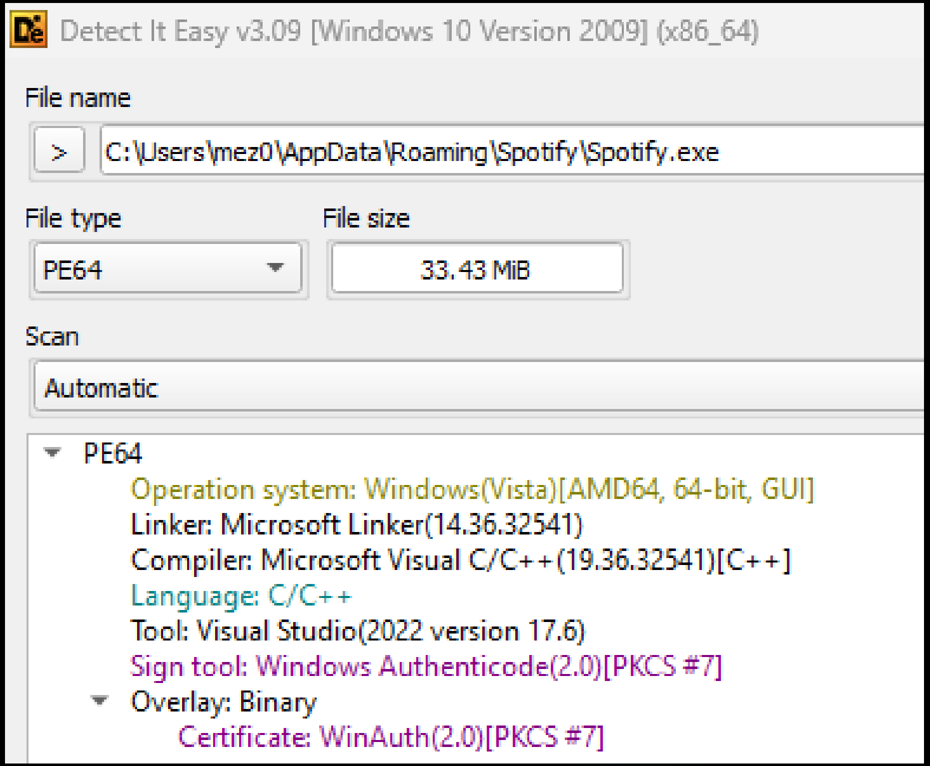
Detect It Easy v3.09 [Windows 10 Version 2009] (x86_64)

File name: E:\dataset\malware\0000029bfead495a003e43a7ab8406c6209ffb7d5e59dd212607aa358bfd66ea.dat

File type: PE32 | File size: 443.32 KiB

Scan: Automatic

```
PE32
    Operation system: Windows(95)[Unknown, 32-bit, Application]
    Linker: Polink(2.50*)[Application32]
    Compiler: PureBasic(4.X-6.X)
    Language: BASIC
    Overlay: Binary
        Format: plain text[LF]
```



Detect It Easy v3.09 [Windows 10 Version 2009] (x86_64)

File name: C:\Users\mez0\AppData\Roaming\Spotify\Spotify.exe

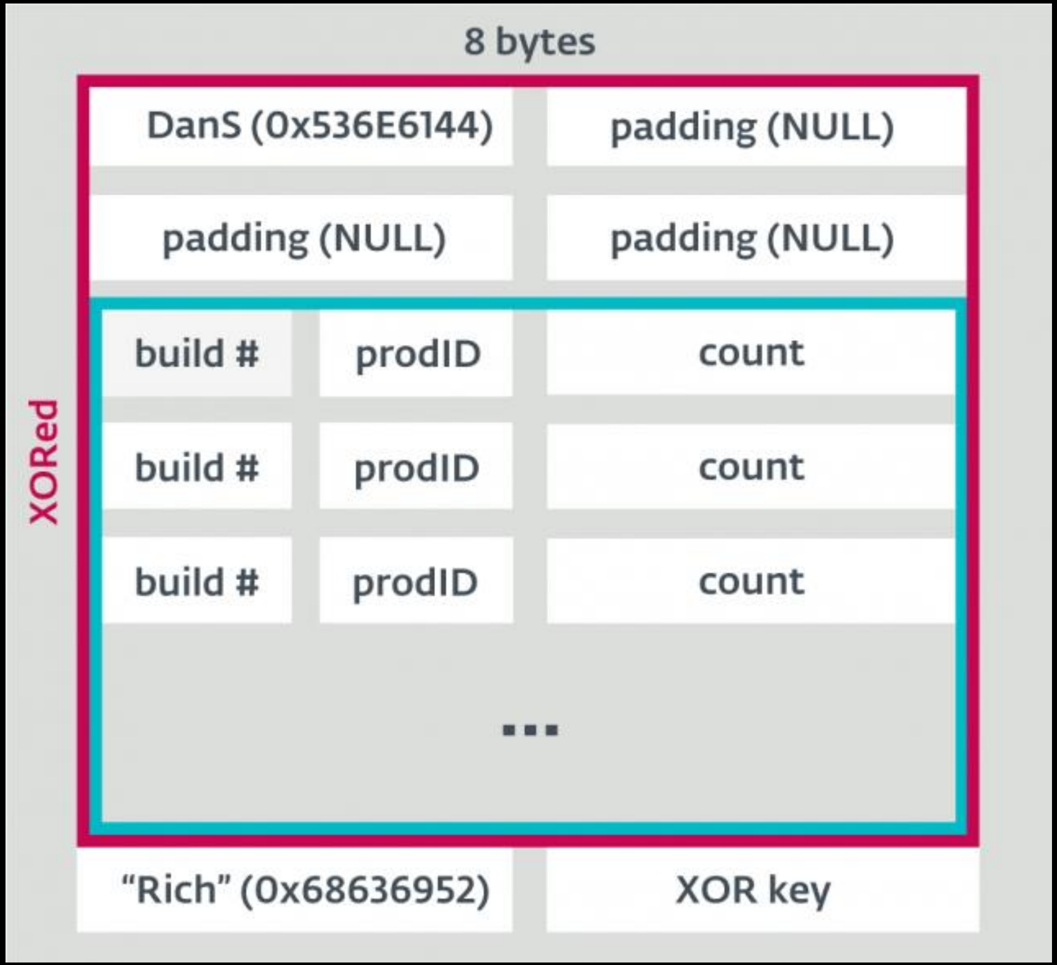
File type: PE64 | File size: 33.43 MiB

Scan: Automatic

```
PE64
    Operation system: Windows(Vista)[AMD64, 64-bit, GUI]
    Linker: Microsoft Linker(14.36.32541)
    Compiler: Microsoft Visual C/C++(19.36.32541)[C++]
    Language: C/C++
    Tool: Visual Studio(2022 version 17.6)
    Sign tool: Windows Authenticode(2.0)[PKCS #7]
    Overlay: Binary
        Certificate: WinAuth(2.0)[PKCS #7]
```

$ Compiler

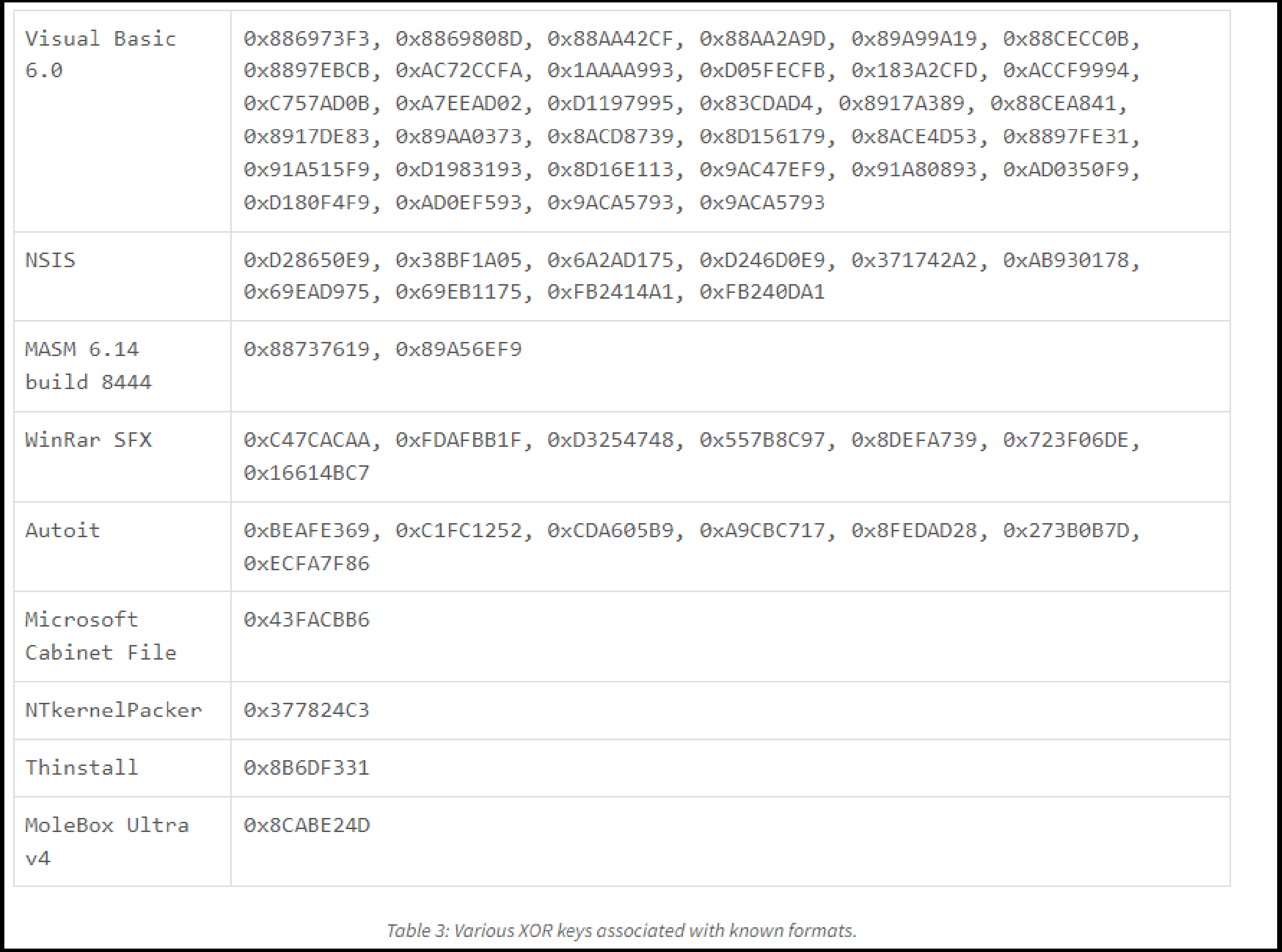
- In 2020, ESET published *Rich Headers: leveraging this mysterious artifact of the PE format*
- Rich Headers were released in VS 97 SP3
- Mostly undocumented
- Contains information about:
  - Product Identifier
  - Build Number
  - And some other stuff



VB2019 paper: Rich Headers: leveraging this mysterious artifact of the PE format

$ Compiler

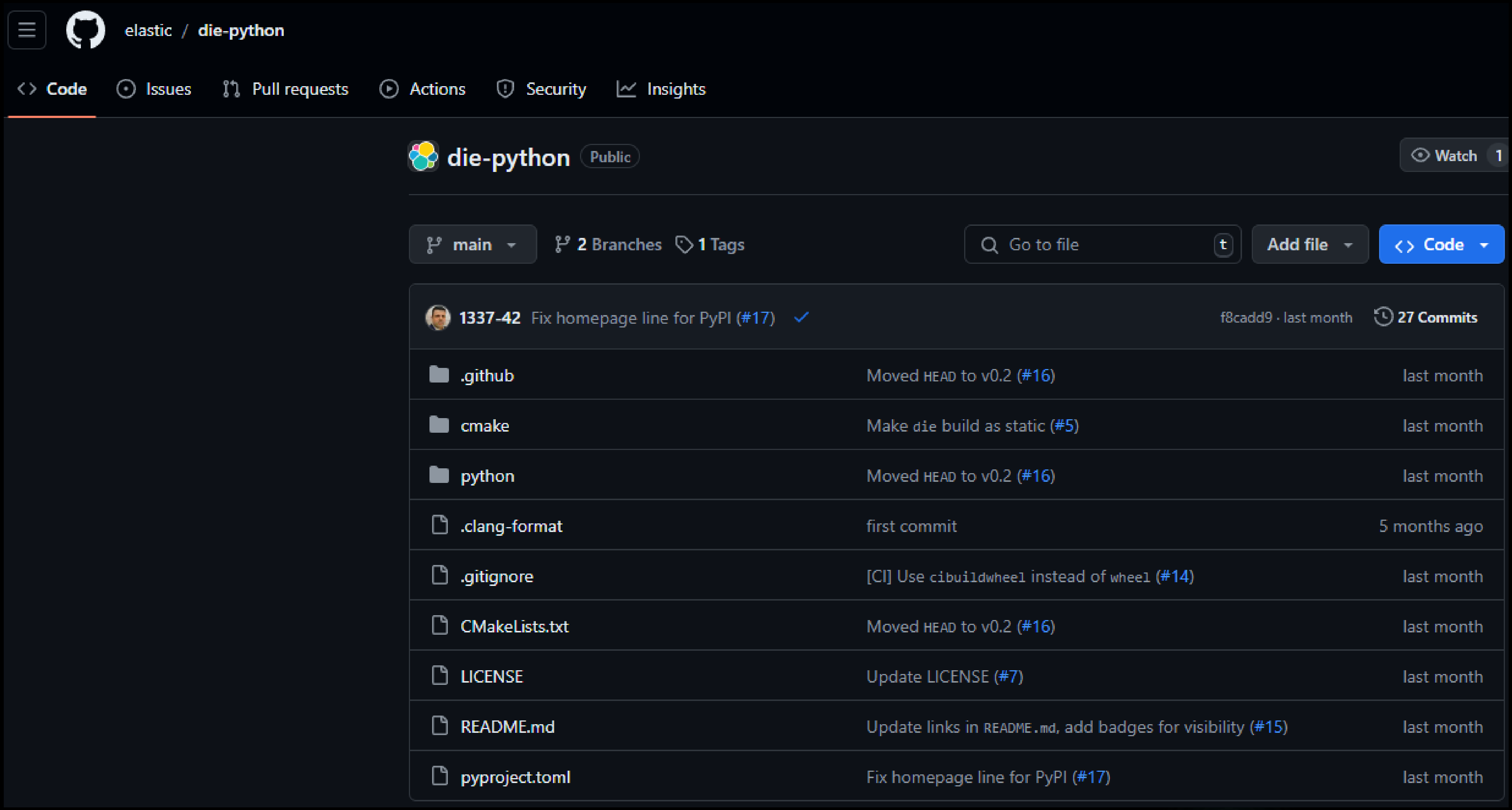
PE 'Rich Headers' were introduced with the release of Visual Studio 97 SP3. Microsoft didn't announce that it had implemented such a feature or give a reason for its introduction, and never released any kind of documentation for it, so we cannot really be sure about its original purpose, but it seems that Microsoft simply wanted to have some sort of development environment fingerprint stored in the executables, or perhaps to help with diagnostics and debugging. Regardless of the original intent, the Rich Header has proved to be a very valuable block of data for malware researchers, where a few hundred bytes, when interpreted correctly, can be used as a very strong factor for attribution and detection.



Visual Basic 6.0	0x886973F3, 0x8869808D, 0x88AA42CF, 0x88AA2A9D, 0x89A99A19, 0x88CECC0B, 0x8897EBCB, 0xAC72CCFA, 0x1AAAA993, 0xD05FECFB, 0x183A2CFD, 0xACCF9994, 0xC757AD0B, 0xA7EEAD02, 0xD1197995, 0x83CDAD4, 0x8917A389, 0x88CEA841, 0x8917DE83, 0x89AA0373, 0x8ACD8739, 0x8D156179, 0x8ACE4D53, 0x8897FE31, 0x91A515F9, 0xD1983193, 0x8D16E113, 0x9AC47EF9, 0x91A80893, 0xAD0350F9, 0xD180F4F9, 0xAD0EF593, 0x9ACA5793, 0x9ACA5793
NSIS	0xD28650E9, 0x38BF1A05, 0x6A2AD175, 0xD246D0E9, 0x371742A2, 0xAB930178, 0x69EAD975, 0x69EB1175, 0xFB2414A1, 0xFB240DA1
MASM 6.14 build 8444	0x88737619, 0x89A56EF9
WinRar SFX	0xC47CACAA, 0xFDAFBB1F, 0xD3254748, 0x557B8C97, 0x8DEFA739, 0x723F06DE, 0x16614BC7
Autoit	0xBEAFE369, 0xC1FC1252, 0xCDA605B9, 0xA9CBC717, 0x8FEDAD28, 0x273B0B7D, 0xECFA7F86
Microsoft Cabinet File	0x43FACBB6
NTkernelPacker	0x377824C3
Thinstall	0x8B6DF331
MoleBox Ultra v4	0x8CABE24D

Table 3: Various XOR keys associated with known formats.

$ Compiler



elastic / die-python

Code · Issues · Pull requests · Actions · Security · Insights

die-python Public

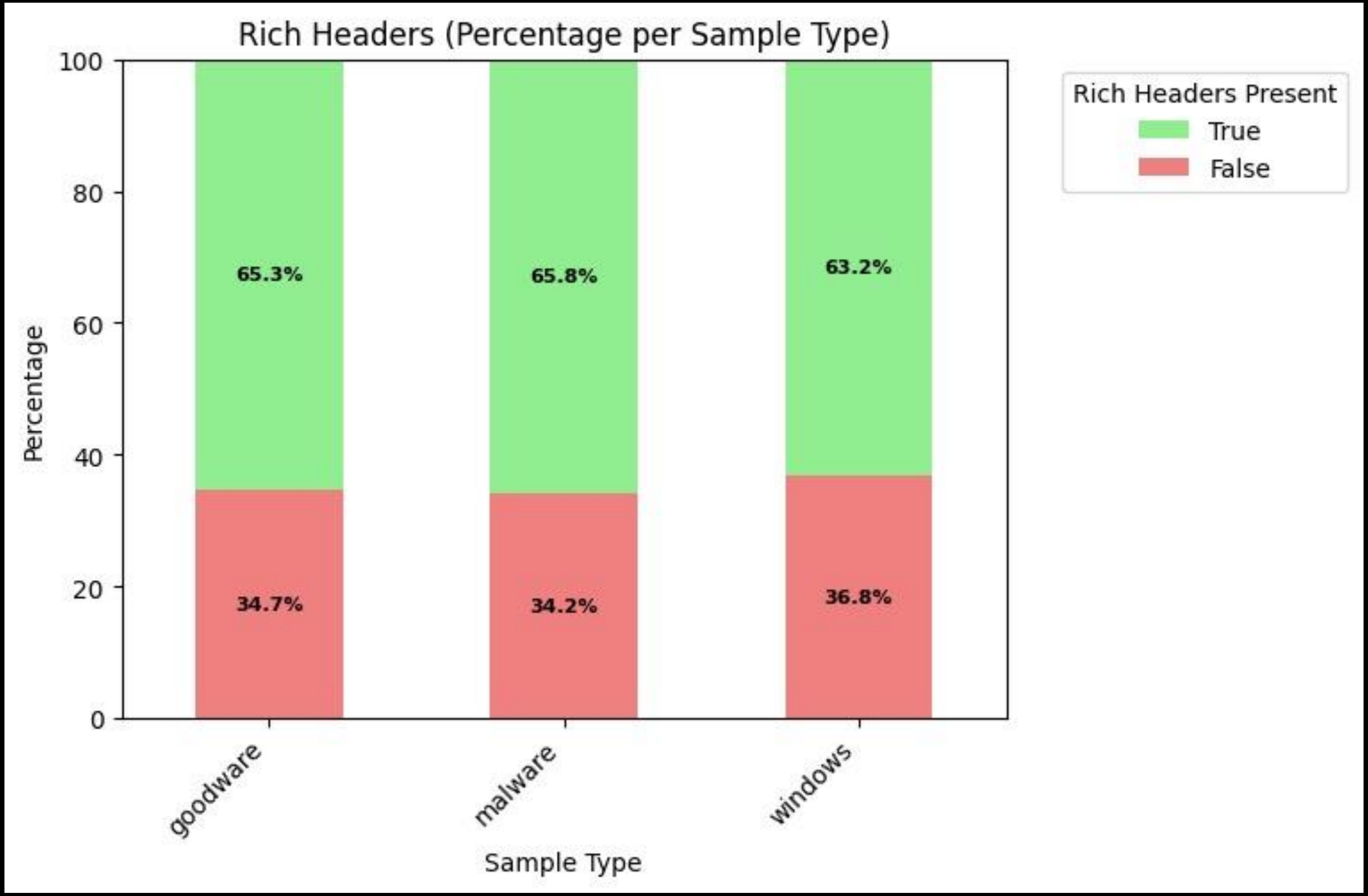
Watch 1

main · 2 Branches · 1 Tags · Go to file · Add file · Code

1337-42 Fix homepage line for PyPI (#17) ✓ f8cadd9 · last month 27 Commits

.github	Moved HEAD to v0.2 (#16)	last month
cmake	Make die build as static (#5)	last month
python	Moved HEAD to v0.2 (#16)	last month
.clang-format	first commit	5 months ago
.gitignore	[CI] Use cibuildwheel instead of wheel (#14)	last month
CMakeLists.txt	Moved HEAD to v0.2 (#16)	last month
LICENSE	Update LICENSE (#7)	last month
README.md	Update links in README.md, add badges for visibility (#15)	last month
pyproject.toml	Fix homepage line for PyPI (#17)	last month

$ Compiler


Rich Headers (Percentage per Sample Type)
Rich Headers Present
🟩 True
🟥 False
Percentage
Sample Type
0
20
40
60
80
100
goodware
malware
windows
65.3%
34.7%
65.8%
34.2%
63.2%
36.8%

Best Practices

Art by Dan Mumford

$ Implant Methodology

Key	Colour
Optional	🟦
Preferred	🟩
Paranoia	🟥

Consideration	Example
Environmental Keying	Domain name
Host Keying	Hostname
External Keying	IPV4 == Expected
Anti-Debug	*Something clever*
Anti-VM	*\HKEY_CURRENT_USER\Software\VMware, Inc.**
Retrieve Payload Remotely	HTTP/DNS
Loading Routine	VirtualAlloc, VirtualProtect, etc.
Execution Routine	CreateThread, Callbacks, etc.
Defensive Impairment	ETW, AMSI, WFP, etc.
Post Execution Cleanup	VirtualFree / DeleteFile, etc
PE Management	Exports, entrypoints, icons, etc.
Binary Post-Processing	Embed Imports, etc.

$ Implant Methodology

	Consideration	Example
Pre-load	Environmental Keying	Domain name
	Host Keying	Hostname
	External Keying	IPV4 == Expected
	Anti-Debug	*Something clever*
	Anti-VM	*\HKEY_CURRENT_USER\Software\VMware, Inc.\**
load	Retrieve Payload Remotely	HTTP/DNS
	Loading Routine	VirtualAlloc, VirtualProtect, etc.
	Execution Routine	CreateThread, Callbacks, etc.
	Defensive Impairment	ETW, AMSI, WFP, etc.
Post-load	Post Execution Cleanup	VirtualFree / DeleteFile, etc
Post-compilation	PE Management	Exports, entrypoints, icons, etc.
	Binary Post-Processing	Embed Imports, etc.

Malware Evasion Techniques Part 2: Anti-VM Blog

Windows: Evasion techniques

$ Implant Methodology

Entropy

File Size

Sections

Imports / Exports

Imports

Signing

Consideration	**Example**
Environmental Keying	Domain name
Host Keying	Hostname
External Keying	IPV4 == Expected
Anti-Debug	*Something clever*
Anti-VM	\HKEY_CURRENT_USER\Software\VMware, Inc.*
Retrieve Payload Remotely	HTTP/DNS
Loading Routine	VirtualAlloc, VirtualProtect, etc.
Execution Routine	CreateThread, Callbacks, etc.
Defensive Impairment	ETW, AMSI, WFP, etc.
Post Execution Cleanup	VirtualFree / DeleteFile, etc
PE Management	Exports, entrypoints, icons, etc.
Binary Post-Processing	Embed Imports, signing, etc.

$ Final thoughts

- Gold Standard:
  - Pretend to be Windows
    - You won’t be able to do this, so much to get right
    - Trying to make this perfect, will create more variance, thus standing out

- Goodware is a lot better to aim for - most malware doesn’t even bother

- Doing a little, but not knowing why, and ending up as gitfud, is worse than just using well-honed loader or even meterpreter when it comes to static points of interest
  - This is not the case for runtime!

$ Final thoughts

- The statisiclyl average binary is

POI	Goodware	Winware	Malware
Entropy			
File Size			
Imports			
Exports			
Signed	TRUE	TRUE	FALSE
Compiler			

$ Final thoughts

__iob_func	CharNextA
_amsg_exit	CoCreateInstance
_CorDllMain	CoInitialize
_initterm	CoTaskMemFree
_lock	CoUninitialize
_unlock	DestroyWindow
abort	DispatchMessageA
calloc	GetKeyboardType
free	GetSystemMetrics
fwrite	LoadStringA
malloc	MessageBoxA
memcpy	OleInitialize
memmove	OleUninitialize
memset	RegCloseKey
realloc	RegOpenKeyExA
strchr	RegQueryValueExA
strcmp	SysAllocStringLen
strlen	SysFreeString
strncmp	SysReAllocStringLen
vfprintf	VariantClear

$ Questions?

Thanks!

Art by Dan Mumford