Revealing the Machine

A Study of the Rich Header and Respective Malware Triage

George Webster

Technical University of Munich

Specializes in developing scalable methods to perform cyber analytics, dynamic and static analysis techniques, and distributed systems

Julian Kirsch

Technical University of Munich

Specializes in reverse engineering, binary exploitation, and Virtual Machine Introspection

Why Are We Here?

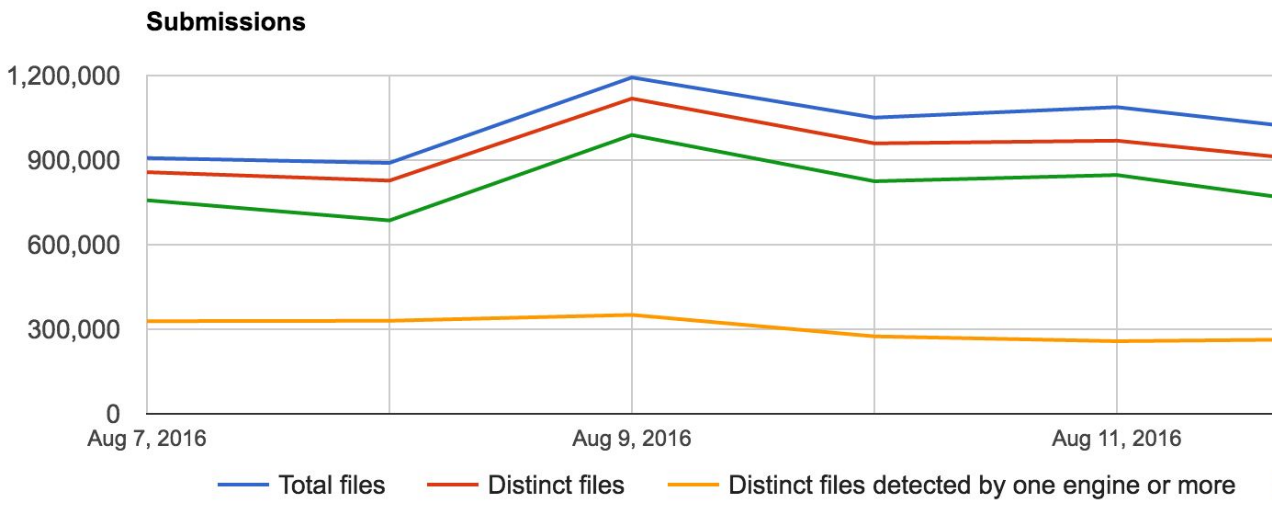
01

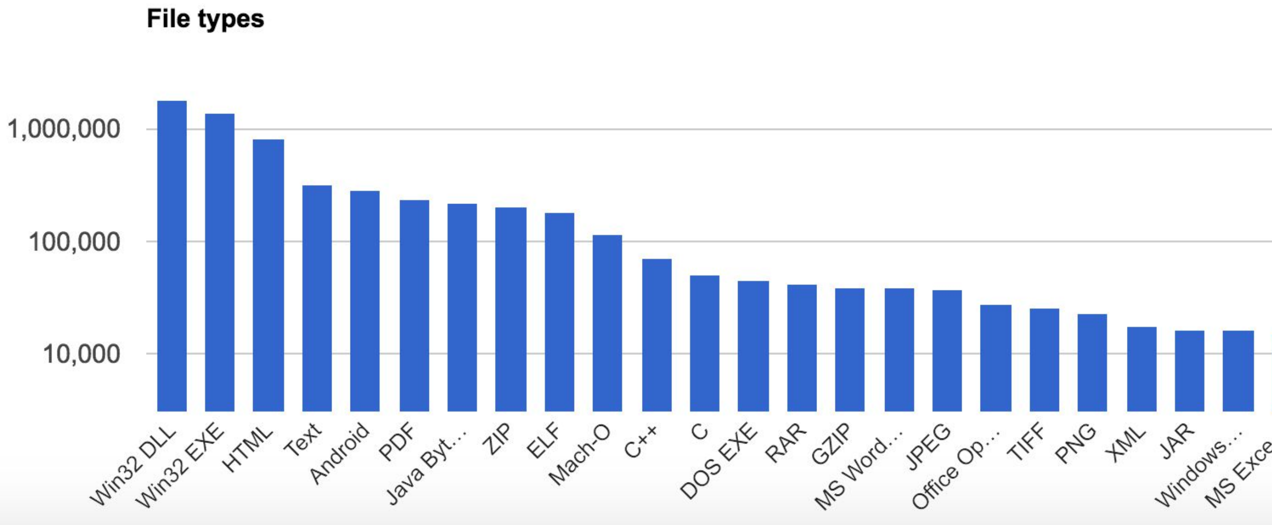
Problem
Why Do You Care

Data, Data, Everywhere

How do you:

- Triage?
- Find related data?
- Make sense of everything?





What about this obscure PE32 field?

Overlooked, poorly documented, inaccurate assessments

Rich Header

ooked and poorly

32

tifier for major and generic library

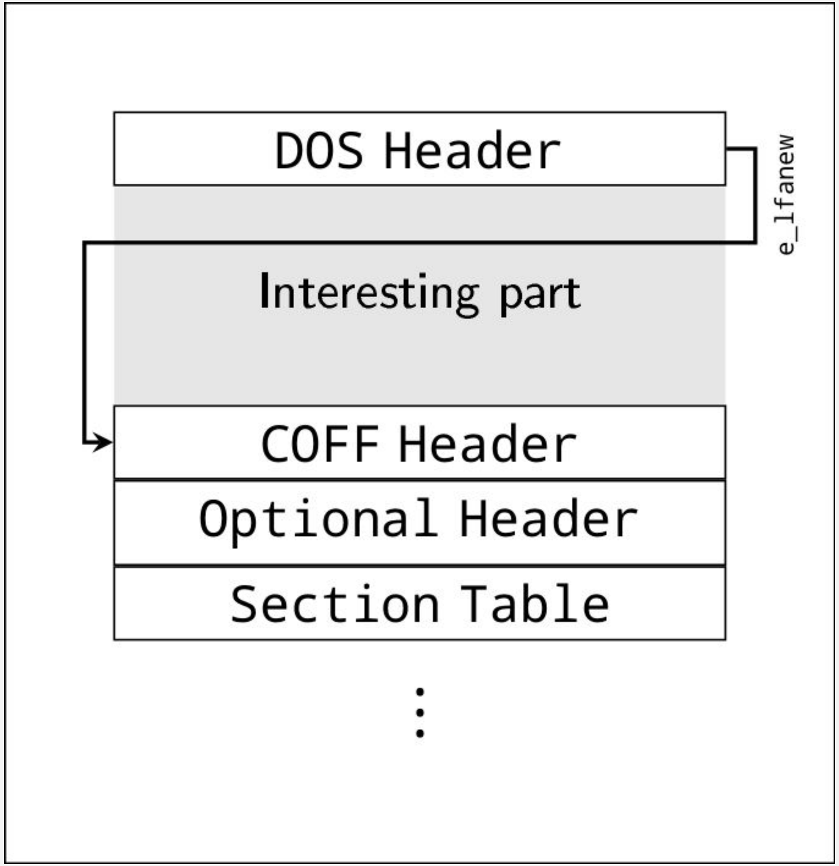
ion information, and compiler flags,

- ords the number of times the linker
  oduct

02

PE32 File Format
Compiler Tool Chain

Background



PE File

Stub between DOS and COFF header containing two things:

01

DOS program printing "This program cannot be run in DOS mode"

- Documented by Microsoft
- Can be replaced by any valid MS-DOS application using MSVC's `/STUB` compiler flag

02

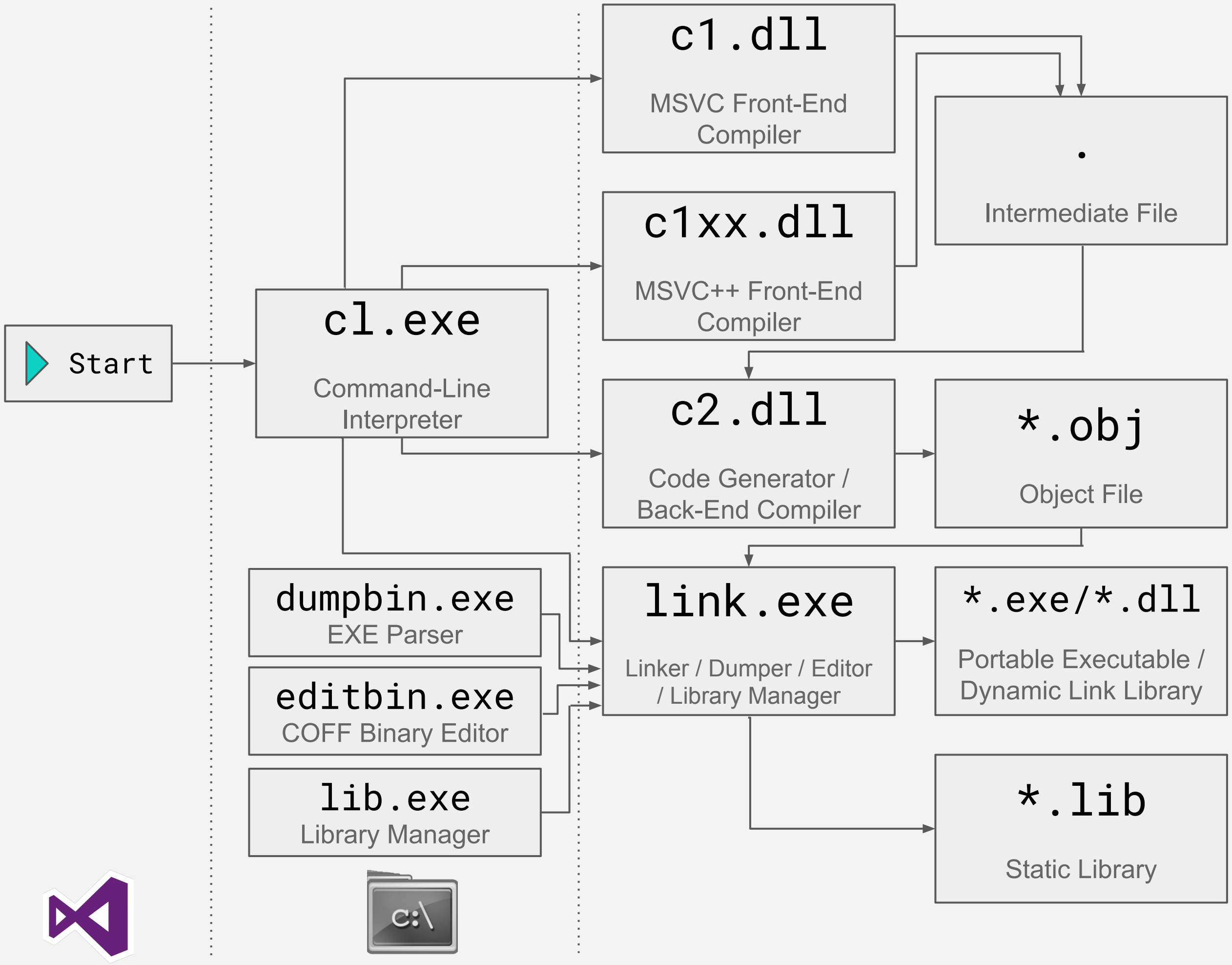
RICH header containing unknown bytes terminated by the string "Rich" and a magic number

- Never officially mentioned by Microsoft
- No consistent explanation available

MSVC Compiler Toolchain

Consisting of:

- Command-Line Interpreter
- C/C++ Frontend
- Code Generator
- (Multi Purpose) Linker



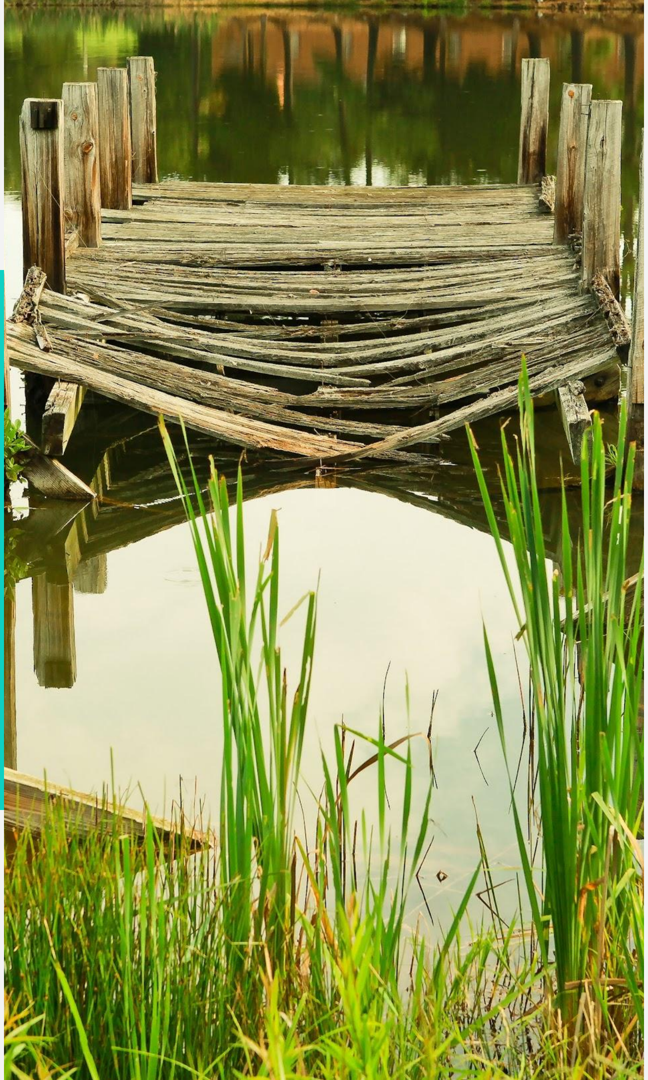
Rich Header

03

What does it contain?
What are these `@comp.id`s?
How is it created?
How is it extracted?

Obfuscated, Undocumented, Part of the PE Header

Included in MS Toolchain since Visual Studio 6 (1998) and maybe even earlier. First discussed in 2004 and reverse engineered in 2008 by Daniel Pistelli



01

Added by the Microsoft Linker

02

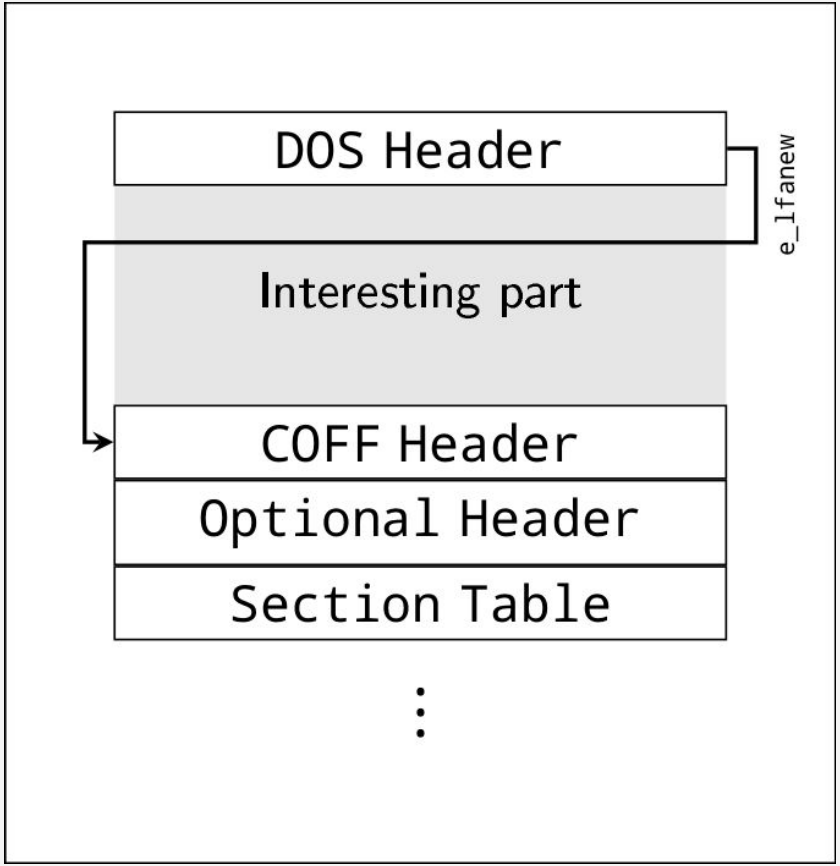
Each iteration of the Microsoft Toolchain adjusts how the Rich Header is generated and updates product mapping

03

Contains information about how the binary was created

PE32 Structure

Let's dive into it!



e_lfanew

```
00000000: 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 |MZ..............|
00000010: b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 |........@.......|
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 f0 00 00 00 |................|
00000040: 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 |........!..L.!Th|
00000050: 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f |is program canno|
00000060: 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 |t be run in DOS |
00000070: 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 |mode....$.......|
00000080: 16 f7 59 40 52 96 37 13 52 96 37 13 52 96 37 13 |..Y@R.7.R.7.R.7.|
00000090: 8f 69 f9 13 50 96 37 13 5f c4 ea 13 50 96 37 13 |.i..P.7._...P.7.|
000000a0: 5f c4 e8 13 50 96 37 13 5f c4 d7 13 46 96 37 13 |_...P.7._...F.7.|
000000b0: 5f c4 d6 13 5b 96 37 13 8f 69 fc 13 5b 96 37 13 |_...[.7..i..[.7.|
000000c0: 52 96 36 13 48 97 37 13 5f c4 de 13 33 96 37 13 |R.6.H.7._...3.7.|
000000d0: 5f c4 ec 13 53 96 37 13 5f c4 e9 13 53 96 37 13 |_...S.7._...S.7.|
000000e0: 52 69 63 68 52 96 37 13 00 00 00 00 00 00 00 00 |RichR.7.........|
000000f0: 50 45 00 00 64 86 06 00 df ba 90 55 00 00 00 00 |PE..d......U....|
...
```

DOS Header

COFF Header

Rich Header

Header Structure

```
00000080: 16 f7 59 40 52 96 37 13 52 96 37 13 52 96 37 13 |..Y@R.7.R.7.R.7.|
00000090: 8f 69 f9 13 50 96 37 13 5f c4 ea 13 50 96 37 13 |.i..P.7._...P.7.|
000000a0: 5f c4 e8 13 50 96 37 13 5f c4 d7 13 46 96 37 13 |_...P.7._...F.7.|
000000b0: 5f c4 d6 13 5b 96 37 13 8f 69 fc 13 5b 96 37 13 |_...[.7..i..[.7.|
000000c0: 52 96 36 13 48 97 37 13 5f c4 de 13 33 96 37 13 |R.6.H.7._...3.7.|
000000d0: 5f c4 ec 13 53 96 37 13 5f c4 e9 13 53 96 37 13 |_...S.7._...S.7.|
000000e0: 52 69 63 68 52 96 37 13 00 00 00 00 00 00 00 00 |RichR.7.........|
```

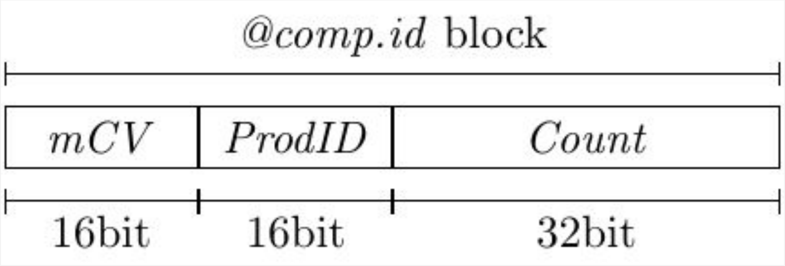
- Footer (8 + x bytes)
  - "Rich" identifier
  - Checksum
  - Zero padding (*presumably* to next multiple of 16)

Header Structure

```
00000080: 44 61 6e 53 00 00 00 00 00 00 00 00 00 00 00 00 |DanS............|
00000090: dd ff ce 00 02 00 00 00 0d 52 dd 00 02 00 00 00 |.........R......|
000000a0: 0d 52 df 00 02 00 00 00 0d 52 e0 00 14 00 00 00 |.R.......R......|
000000b0: 0d 52 e1 00 09 00 00 00 dd ff cb 00 09 00 00 00 |.R..............|
000000c0: 00 00 01 00 1a 01 00 00 0d 52 e9 00 61 00 00 00 |.........R..a...|
000000d0: 0d 52 db 00 01 00 00 00 0d 52 de 00 01 00 00 00 |.R.......R......|
000000e0: 52 69 63 68 52 96 37 13 00 00 00 00 00 00 00 00 |RichR.7.........|
```

- Header (4 + 12 bytes)
  - "DanS"
  - Zero padding (fix!)
- `@Comp.id` Blocks (n x 8 bytes)
  - n `@Comp.id` Blocks
- Footer (8 + x bytes)
  - "`Rich`" identifier
  - Checksum
  - Zero padding (*presumably* to next multiple of 16)

} XORed with Checksum



Structure of `@comp.id`

01 `mCV`

Minor version of the compiler used to make the product

02 `ProdID`

Unique identifier that specifies a specific identify or type of object

03 `Count`

Specifies how often the specific `ProdID` and `mCV` were used by the linker

ProdID	VS Release	Object Type	Generator
0x105	2015	C++	c2.dll
0x104	2015	C	c2.dll
0x103	2015	Assembly	c2.dll
0xff	2015	Resource File	cvtres.exe
0xb4	2010	C++	c2.dll
0x5e	.NET	Resource File	cvtres.exe
0x15	6	C	c2.dll

01

02

ProdID

1) Generic identifier: Identifies the referenced object type and VS Release

2) Unique identifier: Appears to map to major libraries but exact definition is unknown

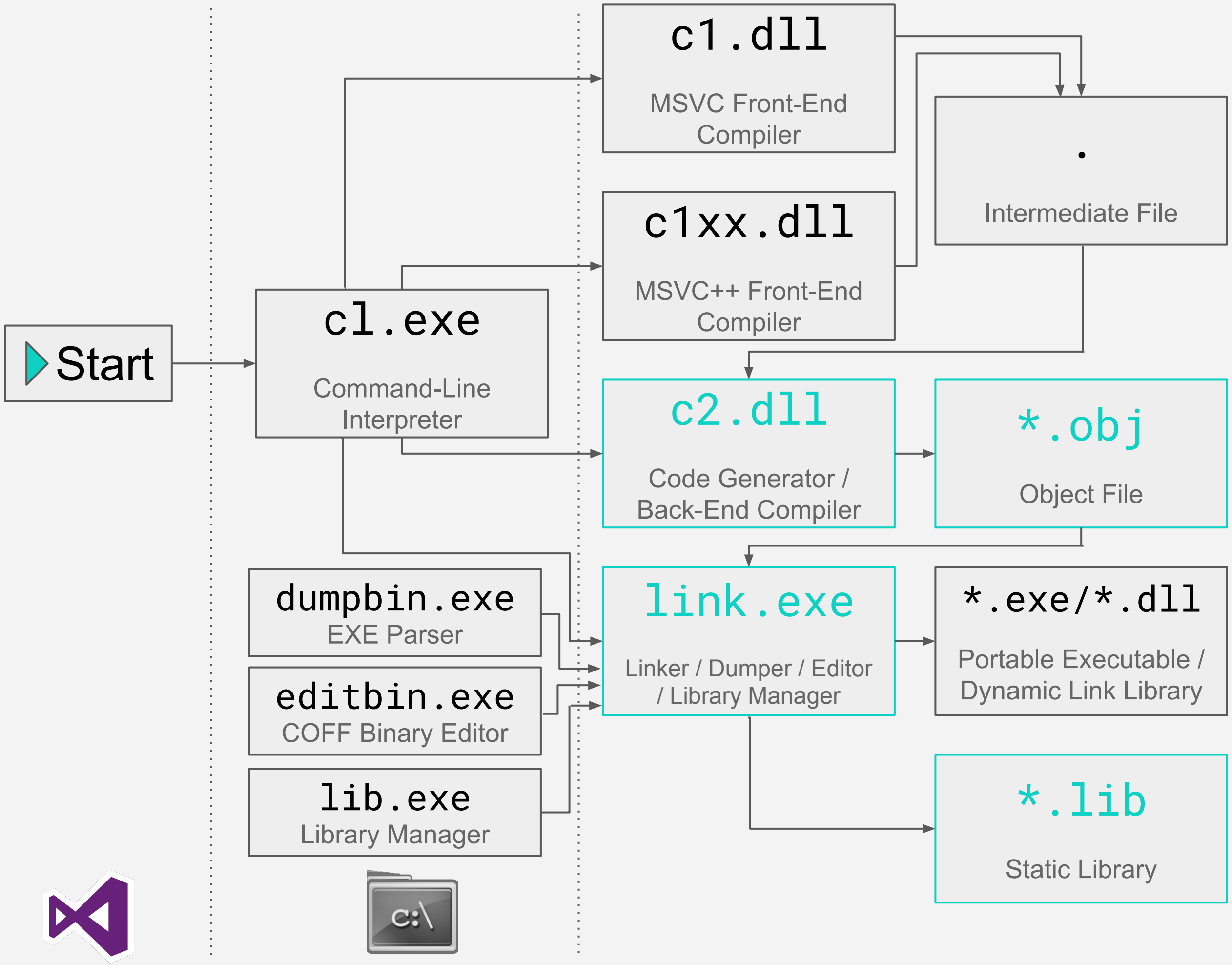
Checksum

- Rotate the DOS Header bytes by their offset
- Rotate contents of `@comp.ids` by their `count`
- Only 37 of the 64 bits per `@comp.id` are checksummed!

```python
## Rotate left helper function
rol32 = lambda v, n: \
        ((v << (n & 0x1f)) & 0xffffffff) | \
        (v >> (32 - (n & 0x1f)))

## raw_dat is a bytearray containing the exe's data
## compids is the list of deciphered @compid structs
## off is the offset to the start of the Rich Header
def calc_csum(raw_dat, compids, off):
    csum = off

    for i in range(off):
        ## skip e_lfanew as it's not initialized yet
        if i in range(0x3c, 0x40):
            continue
    csum += rol32(raw_dat[i], i)

    for c in compids:
        csum += rol32((c['prodid'] << 16) | c['mcv'], \
                c['count'])

    return csum & 0xffffffff
```

Insertion of Rich Header

- Back-End Compiler generates one `@comp.id` per object
- Linker collects `@comp.id`s from objects and puts them into the PE.



04

Statistics

Samples with Rich Header
Samples without the Rich Header

71%

Random

1 million samples. Including packed and obfuscated malware

98%

APT1

298 samples from a popular APT actor

37%

Zeus-Citadel

1928 samples from a popular criminal malware variant

2%

Mediyes

1873 samples from a dropper that contains a valid signature

The Microsoft Linker always adds the Rich Header

No Header:

- .Net
- MinGW
- GCC
- dUP

With Header:

- Visual Studios
- Intel
- UPX*
- ASPack*
- Nullsoft*

* More to come!

So What?

05

Identifying Suspicious Binaries
Similarity Matching
Demonstration
Discussion

Discrepancies are GREAT!

Corrupt Checksum

Post modified binary

Duplicate Entries

Packing Error

Fast!

Very inexpensive check to perform. Out of 1 million samples, identified 22% were packed

Can we do more?

With only the data in the Rich Header can we create the following:

Fast

Return the results in near real-time

Similarity Matching

Identify binaries that are similar. Potentially different versions or baked in

Fingerprint Actors

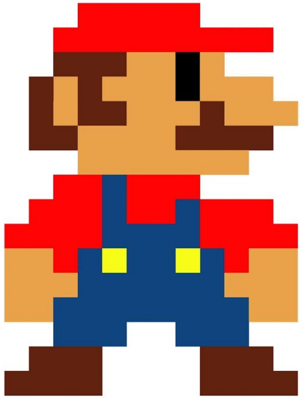
Identify binaries that were created under similar build environments

Dimension Reduction

Stacked Autoencoder

Benefits:

- Easier: denser lower-dimensional space
- Efficient: reduced memory requirement



VS.

Similarity matching

KNN w/ Ball Tree

Benefits:

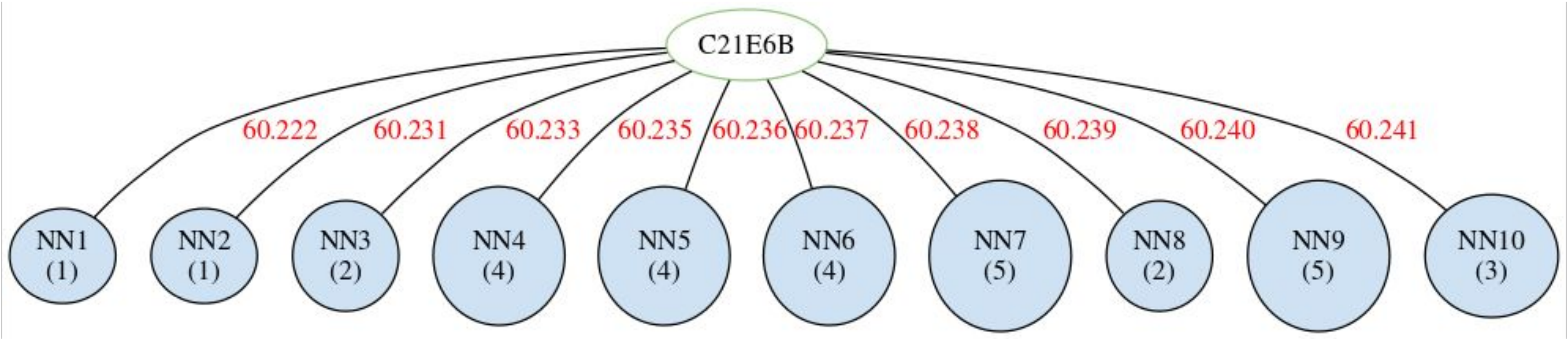
- Less pre-processing
- No predefined number of groups
- Fast lookups: 6.73ms Per 2 million

Demo!

Finding similar malware across a million samples

Visualizing the Demo

Top 10 Nearest Neighbor “clusters”



Case Study APT1

Based on SHA256:
F737829E9AD9A025945AD9
CE803641677AE0FE3ABF43
B1984A7C8AB994923178

All samples have different AV signatures

Matching Rich Header

Detected 1 sample

1:1 Match
Identical functionality
Identical code base

Sha256 difference was from compiler artifacts

Nearest Neighbors

Detected 3 samples

1) Different Build Environment
Library versions were slightly off

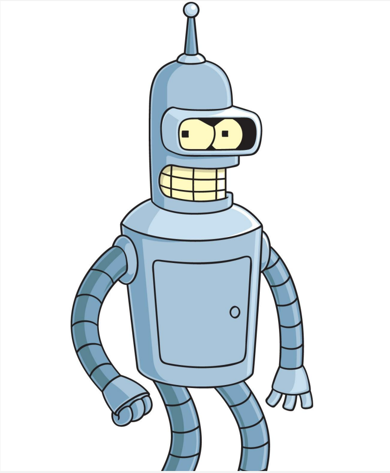
2) Different Version
Adds function “FlushFileBuffers”

3) Version Upgrade
Removes double write by calling strcat

Case Study Zeus

Based on SHA256:
8471A205E1E85080B7230D
B19D773D43A559ECA7A4B8
92E64E74C4E7E0A0D3BD

Most samples have a generic AV signature



Matching Rich Header

Detected 23 samples

1:1 Match
Identical functionality

Assembly equivalent:
- XOR uses a "do while" versus "for" loop
- Code segments reordered

Nearest Neighbors

Detected 4 samples

Different version
Identical functionality

XOR algorithm loops >8 times more

Case Study Zeus cont

Based on SHA256:
7F1A07F484A8AE853DB936
4508A7BDFD3718BFA5E311
5AD941B216D0B662A880

Most samples have no signature of generic AV signatures

Matching Rich Header

Detected 36,606 samples

1:1 Match
Identical functionality
A constant value changes

16,123 samples have no AV detection

Nearest Neighbors

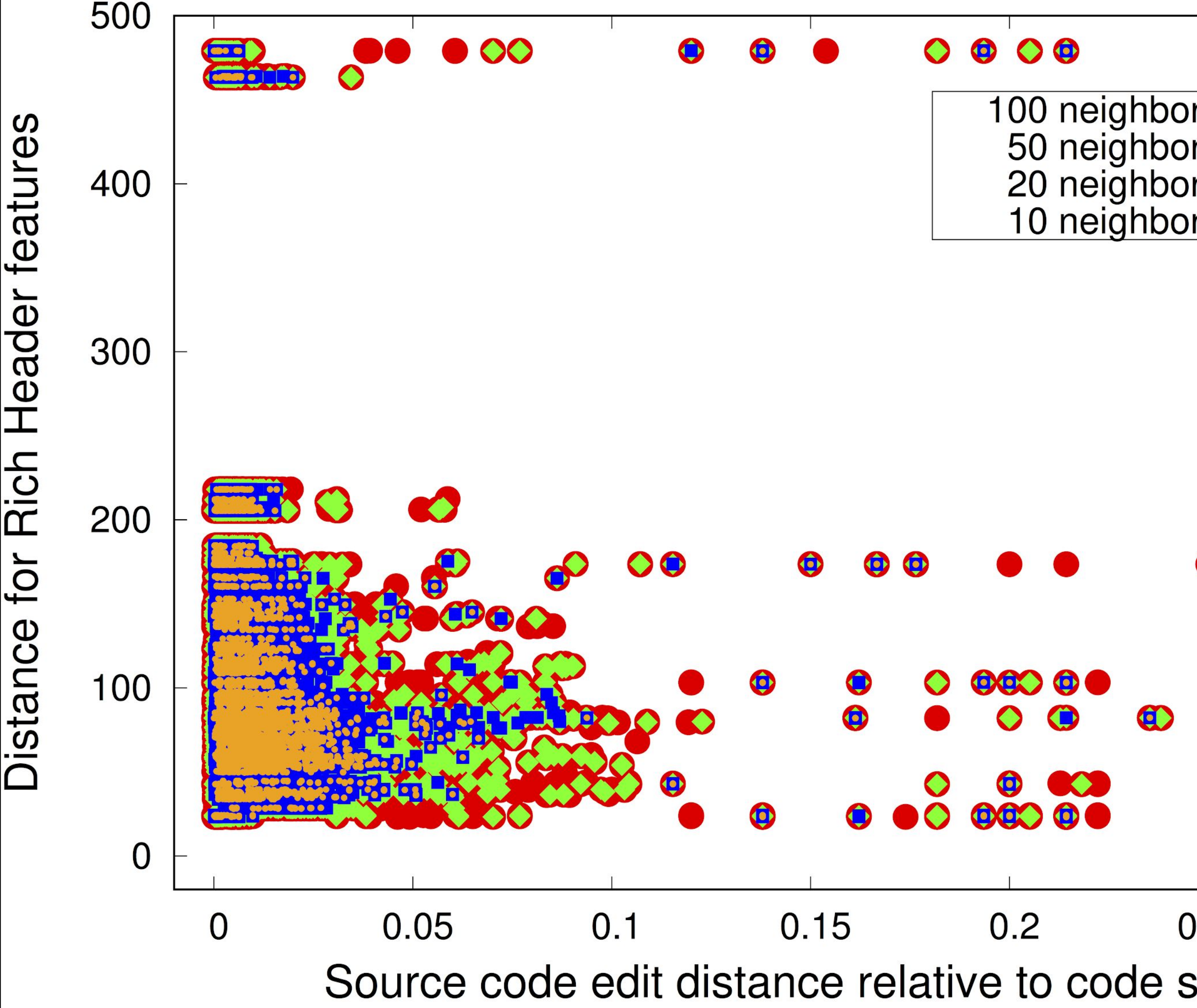
Detected 1,567 samples

Different Build Environment
Identical functionality
Library versions were slightly off

511 samples have no AV detection

Validation

Correlation of IDA generated code across 1 million random samples. Using entropy of source code.



06

Where do we go from here
Conclusion

Conclusion

Future

Rich Header is valuable for triage but future work remains:

- **ProdID:** What are the true mappings?
- **Checksum:** Why is the checksum designed as it is?
- **Purpose:** What was the original intention, why it is maintained, why is it hidden?
- **Combine:** Individual triage methods can be overcome. We need to combine with other algorithms to reach the full potential

Ripe for Research and Incorporation with Existing Methods!

Releasing The Rich Header Extractor

Apache2 License

Docker Service

Ready to use with Holmes

holmesprocessing.github.io

```python
DANS = 0x536E6144 # 'DanS' as dword
RICH = 0x68636952 # 'Rich' as dword

try:
    rich_data = pe.get_data(0x80, 0x80)
    current_pos = 0x80+0x80
    if len(rich_data) != 0x80:
        return None
    data = list(struct.unpack("<32I", rich_data))
except:
    return None

# the checksum should be present 3 times after the DanS signature
checksum = data[1]
if (data[0] ^ checksum != DANS
    or data[2] != checksum
    or data[3] != checksum):
    return None
d['checksum'] = checksum

# add header values
headervalues = []
headerparsed = []
data = data[4:]
found_end = False
while not found_end:
    for i in xrange(len(data) // 2):

        # Stop until the Rich footer signature is found
        if data[2 * i] == RICH:
            found_end = True
            # it should be followed by the checksum
            if data[2 * i + 1] != checksum:
                print('Rich Header corrupted')
            break

        # header values come by pairs
        temp1 = data[2 * i] ^ checksum
        temp2 = data[2 * i + 1] ^ checksum
        headervalues.extend([temp1, temp2])
        headerparsed.append({'id': temp1 >> 16,
                             'version': temp1 & 0xFFFF,
                             'times_used': temp2})
```

Take-aways

Rich Header is valuable for triage

- **Quick Detection:** Identifies packed and post modified binaries
- **Similarity Matching:** Finds binaries with same functionality
- **Build Environment Fingerprinting:** Actors?

We need help! Please send us copies of your C2.dll, cvtres.exe, link.exe, and ml.exe

Thank you

- Bojan Kolosnjaji
- Christian von Pentz
- Marcel Schumacher
- Zachary Hanif
- Apostolis Zarras
- Claudia Eckert

George Webster &
Julian Kirsch
Technical University of Munich
Chair for IT Security

holmesprocessing.github.io