I am packer and so can you

Mike Sconzo
Agenda*

*Powered by business synergy
It’s all about me!

- Threat Research at Bit9 + Carbon Black
- Enjoy static analysis, machine learning, network forensics, and eating BBQ and pie (yes, I live in TX)
- SecRepo.com
- @sooshie
What’s the problem?

Detecting packers, compilers and various artifacts from the development cycle can be really useful when looking at malware or files in general. However, the de facto standard (PEiD) was created over 10 years ago, and there are very few recent signature updates. Is it time for something new and different?
Moar?

- Zero (or near zero) trust in prior solutions
  - Approach this from a clustering perspective
- Easy to generate signatures
  - Non-experts want to play too
- Cross platform
  - `<Insert Mac Fanboi Statement>`
- Simple to understand and extend
- Fuzzy Matching (similarity)
  - Understand signature overlap
  - Percent of each signature matched
Refresher

Terms, File Structure, 101
PE Simplified
# PE format

PE (Portable Executable) format is a file format for many different operating systems used to store executable programs and libraries. It is commonly used on Windows but has also been extended to be used on other operating systems. The PE format is designed to be portable and to allow for easy manipulation of files, making it a popular choice for executable files.

## PE File Format Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>File Header</td>
</tr>
<tr>
<td>0x00000010</td>
<td>PE Signature</td>
</tr>
<tr>
<td>0x00000020</td>
<td>PE Header</td>
</tr>
<tr>
<td>0x00000030</td>
<td>Optional Header</td>
</tr>
<tr>
<td>0x00000040</td>
<td>Section Header</td>
</tr>
<tr>
<td>0x00000050</td>
<td>Section Table (each section header is 40 bytes)</td>
</tr>
</tbody>
</table>

## PE File Format Diagram

The PE format is structured in a hierarchical manner, starting with the PE header, followed by optional headers, and then section headers. Each section header contains metadata about the section it describes.

### PE Header Fields

- **Signature**: A 4-byte signature that identifies the file as a PE file.
- **Machine**: A 2-byte field indicating the architecture of the file.
- **NumberOfSections**: A 2-byte field indicating the number of section headers.
- **TimeDateStamp**: A 4-byte field containing the file's timestamp and date.
- **PointerToSymbolTable**: A 4-byte field pointing to the symbol table.
- **PointerToEntryPoint**: A 4-byte field pointing to the entry point of the file.
- **ImageBase**: A 4-byte field containing the base address of the file.
- **SizeOfImage**: A 4-byte field indicating the size of the executable file.
- **SizeOfHeaders**: A 4-byte field indicating the size of the header information.
- **SizeOfOptionalHeader**: A 4-byte field indicating the size of the optional header.
- **Characteristics**: A 4-byte field containing flags indicating the file's characteristics.

### Section Header Fields

- **Name**: A variable-length field containing the section's name.
- **VirtualSize**: A 4-byte field indicating the size of the section's memory.
- **VirtualAddress**: A 4-byte field indicating the section's virtual address.
- **SizeOfRawData**: A 4-byte field indicating the size of the section's raw data.
- **PointerToRawData**: A 4-byte field pointing to the raw data for the section.
- **PointerToVirtualAddressTable**: A 4-byte field pointing to the virtual address table.
- **PointerToDataDirectory**: A 4-byte field pointing to the data directory.
- **NumberOfRelocations**: A 4-byte field indicating the number of relocations.
- **NumberOfLines**: A 4-byte field indicating the number of lines in the section.
- **Characteristics**: A 4-byte field containing flags indicating the section's characteristics.

### Data Directory Fields

- **Data Directory**: An array of data directories, each describing a specific type of data in the file, such as import addresses, export addresses, or debug information.

The PE format is designed to support various features such as debugging, linking, and loading, making it a versatile and powerful file format for executable files.
Things we care about (today)

- LinkerMajorVersion
- LinkerMinorVersion
- NumberOfSections
Tool chain

- Set of tools used to develop software
  - IDE
  - Compiler
  - Linker
  - Etc…

- This talk will touch on compiler/build environment detection
  - Information provided by the linker, etc… will also be used
Packer

- “Program within a program”
- Generally used to compress or obfuscate PE information
  - Evade AV
  - Make static analysis harder
Packers, their parts.

- **Packer**
  - Compresses/obfuscates the original executable and creates a new executable complete with decompression/deobfuscation code

- **Unpacker**
  - A.k.a Stub
  - Run when the new executable is executed and is responsible for producing the original executable
Unpackers, how do they work?

- Take control of AddressOfEntryPoint
- Run the unpacking routine
  - Find the packed data
  - Restore the data contents
  - Perform relocation fixes
  - Resolve imports since the original executable isn’t being loaded by the Windows loader
- Jump into the original program

The popular kids

- **PEiD**
  - “PEiD detects most common packers, cryptors, and compilers for PE files.”

- **YARA**
  - “YARA is a tool aimed at (but not limited to) helping malware researchers to identify and classify malware samples”

- **RDG Packer Detector**
  - “RDG Packer Detector is a detector of packers, cryptors, Compilers, Packers Scrambler, Joiners, Installers”
Back to Reality (Data)

Now on to the more exciting stuff, even though it’s ugly at times.
Data

- 3977 PEiD signatures used for testing

- File sets
  - Sony – 9 Executables
  - Chthonic – 11 Executables
  - Backoff – 22 Executables
  - Volatile Ceader – 36 Executables
  - Carbanak – 74 Executables
  - APT1 – 281 Executables
  - ZeuS – 6774 Executables
  - Random – 411340 Executables
Data analysis

- Basic exploration of the ZeuS dataset
- Some of the possible attributes/features we can look at
- Clustering
# ZeuS + PEiD

<table>
<thead>
<tr>
<th>PEiD Label</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>4600</td>
</tr>
<tr>
<td>UPX v0.89.6 - v1.02 / v1.05 -v1.24 -&gt; Markus &amp; Laszlo [overlay]</td>
<td>781</td>
</tr>
<tr>
<td>UPX 2.90 [LZMA] -&gt; Markus Oberhumer, Laszlo Molnar &amp; John Reiser</td>
<td>318</td>
</tr>
<tr>
<td>PureBasic 4.x -&gt; Neil Hodgson</td>
<td>267</td>
</tr>
<tr>
<td>Microsoft Visual Basic v5.0/v6.0</td>
<td>166</td>
</tr>
<tr>
<td>Armadillo v1.71</td>
<td>164</td>
</tr>
<tr>
<td>Microsoft Visual C++ 8</td>
<td>148</td>
</tr>
<tr>
<td>UPX 2.93 - 3.00 [LZMA] -&gt; Markus Oberhumer, Laszlo Molnar &amp; John Reiser</td>
<td>61</td>
</tr>
<tr>
<td>MingWin32 v?.? (h)</td>
<td>45</td>
</tr>
<tr>
<td>Microsoft Visual C++ 7.0 MFC</td>
<td>44</td>
</tr>
</tbody>
</table>
ZeuS + PEiD
Samples per PEiD signature
ZeuS + PEiD

Correlation between the PEiD signatures
### Highly correlated

<table>
<thead>
<tr>
<th>Associated Software</th>
<th>Correlation Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASPack v2.12</td>
<td>1.0</td>
</tr>
<tr>
<td>Alexey Solodovnikov</td>
<td></td>
</tr>
<tr>
<td>ASProtect V2.X DLL</td>
<td>1.0</td>
</tr>
<tr>
<td>Alexey Solodovnikov</td>
<td></td>
</tr>
</tbody>
</table>
PDB strings

- C:\Users\Samim\Desktop\Stab\stb\Release\stb.pdb
- Y:\DnijJVgd\pitWxRX\ctoerrwx\RtpjVeb.pdb
- H:\RJmq\HYkAuHH\lsvyudBS\yMgpHF\obzwwn.pdb
- C:\answer\record\These\Answer\Dry\Lay\since\Since\mean\Tree\Music.pdb
- X:\DEVELOPMENT\VC++\Cryptor_Evolution_old\release\main.pdb
- c:\temp\debug.pdb
- F:\zmapHjyf\tGQkckQ\UrmgircgraBwwX\nAjaGbB.pdb
- C:\Users\M4x\Documents\Programmieren\PECRYPT\Client\EXECUTABLE\Stub\Release\Stub.pdb
## Linker versions

<table>
<thead>
<tr>
<th>Major.Minor Linker Versions</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>2067</td>
</tr>
<tr>
<td>10.0</td>
<td>1064</td>
</tr>
<tr>
<td>9.0</td>
<td>793</td>
</tr>
<tr>
<td>6.0</td>
<td>717</td>
</tr>
<tr>
<td>5.0</td>
<td>235</td>
</tr>
<tr>
<td>5.12</td>
<td>231</td>
</tr>
<tr>
<td>8.0</td>
<td>201</td>
</tr>
<tr>
<td>7.10</td>
<td>155</td>
</tr>
<tr>
<td>0.0</td>
<td>85</td>
</tr>
<tr>
<td>1.1</td>
<td>73</td>
</tr>
</tbody>
</table>
ASM mnemonics

Symbolic name for a machine instruction

- add
- mov
- nop
- xor
- ...

All mnemonics are treated equally from this point forward
Johnny 5 is alive!
Yes. Disassemble.
Capstone Engine

- Standardize on one disassembler for consistent results
- Free
- Awesome
- Multi-Language support
- Cross Platform
ZeuS + ASM

Just because we can, doesn’t mean we should
It’s important
ASM

- Mnemonics describe program behavior
- Mnemonics at AddressOfEntryPoint describe initial program behavior
  - Compiler setup
  - Unpacker stub
- Use this as the basis for a signature
Sets

- Correlation
  - Doesn’t take order into account
  - Doesn’t really help with distance or similarity

- Jaccard Distance
  - Doesn’t take order into account
  - Distance is based on set membership

- Levenshtein Distance
  - Edits determine distance
  - Position is important
Jaccard

['pushal', 'mov', 'lea', 'push', 'jmp', 'nop', 'mov', 'inc', 'mov', 'inc']
['push', 'mov', 'add', 'push', 'mov', 'call', 'mov', 'mov', 'call', 'mov']

Total # of shared elements/Total # of unique elements

[mov push] / [pushal mov lea jump nop inc push add call]

2/8 = .25
Levenshtein

['pushal', 'mov', 'lea', 'push', 'jmp', 'nop', 'mov', 'inc', 'mov', 'inc']

['push', 'mov', 'add', 'push', 'mov', 'call', 'mov', 'mov', 'call', 'mov']

How many things have to change to make the bottom into the top.

[yes no yes no yes yes no yes yes yes yes yes]

yes = 7

Distance = 7
But...

- Code is executed in order
- There might be branches
- Shouldn’t the ASM mnemonics to the ‘left’ be worth more than the ones on the ‘right’
- Where’s the cutoff
- How many instructions should we care about
- What’s the size of the stub
Enter our superhero

(Tapered Levenshtein)
Tapered levenshtein

Position dependent, left edits have a higher weight than right edits

['pushal', 'mov', 'lea', 'push', 'jmp', 'nop', 'mov', 'inc', 'mov', 'inc'
['push', 'mov', 'add', 'push', 'mov', 'call', 'mov', 'mov', 'call', 'mov']

1 - (position/len(set))

1, 0, .8, 0, .6, .5, 0, .3, .2, .1

3.5 (vs. 7 on the non-tapered version)
Now we’re ready to science

- PE files
- MajorLinkerVersion
- MinorLinkerVersion
- Assembly mnemonics
- Fancy algorithms
Workflow

1. Gather samples

2. Static analysis
   1. PEiD
   2. Disassemble
   3. Header features

3. Cluster

4. Closeness according to distance metric (> 90% similar)
   1. Use banded minhash for < O(n²) comparisons

5. Analyze based on cluster groups using 30 mnemonics
   1. 30 mnemonic chain length is based on prior research and exploration

6. Create Signatures
Please accept our chicken sacrifice, Demo Gods.
XXX-mac:packerid XXXXX$ python ./mmpes.py -s ./test.sig -v -t 0.0 ~/data/APT1/
VirusShare_01e0dc079d4e33d8edd050c4900818da
[*] Processing: /Users/XXX/data/APT1/VirusShare_01e0dc079d4e33d8edd050c4900818da
[TEST] (Edits: 4.6666666667 | Similarity: 0.844) (Minor Linker Version Match: True | Major Linker
Version Match: True | Number Of Sections Match: False)

XXX-mac:packerid XXXXX$ python ./mmpes.py -s ./test.sig -v -t 0.0 ~/data/APT1/
VirusShare_002325a0a67fded0381b5648d7fe9b8e
[*] Processing: /Users/XXX/data/APT1/VirusShare_002325a0a67fded0381b5648d7fe9b8e
[TEST] (Edits: 0.0 | Similarity: 1.000) (Minor Linker Version Match: True | Major Linker
Version Match: True | Number Of Sections Match: True)
Signatures

[Microsoft Visual Basic v5.0]

mnemonics =
push, call, add, add, add, xor, add, inc, add, add, add, add, add, add, adc, dec, mov, adc, add, add, add, adc, or, imul, push, and, and

File

mnemonics =
push, call, add, add, add, xor, add, inc, add, add, add, add, add, add, jmp, dec, mov, xor, add, add, add, add, add, add, add, dec, jnp, add

Results

[Microsoft Visual Basic v5.0] (Edits: 2.93333333333 | Similarity: 0.902)
APT1

Because first APT is best APT
APT 1
PEiD vs. ASM
APT 1
Cluster on 30 ASM Mnemonics
APT 1
Sub-clustered on number of sections
APT 1
Sub-clustered on linker versions
APT 1
Sub-clustered on both
Zeus
Malware or Greek God?
ZeuS
PEiD vs. ASM
ZeuS
Cluster on 30 ASM Mnemonics
ZeuS
Sub-clusters on number of sections
ZeuS
Sub-clustered on linker versions
ZeuS
Sub-clustered on both
Random files

Are random
Random
ASM
Fun fact

- 346680 out of 411340 files don’t meet the similarity requirement*
- No 2 files are at least 90% similar

*Actual number may be lower
Random Cluster on 30 ASM Mnemonics
Random
Sub-clustered on linker versions
Random – Non-Unlabeled
Sub-clustered on linker versions
Random
Sub-clustered on number of sections
Random – Non-Unlabeled
Sub-clustered on number of sections
Random
Sub-clustered on both
Random – Non-Unlabeled
Sub-clustered on both
Google Chrome
AKA Why linkers and sections are important

- 97 files match the ASM signature

- 95 have the same linker version: 10.0
- 2 have a linker version of 11.0

- 94 have the same number of sections: 6
  - Two have 4 sections, one has 5 sections

- 94 binaries have BOTH a linker version of 10.0 AND 6 sections
  - All 94 are signed Google Chrome binaries
UPX
Because somebody was going to ask

- 7469 files packed with UPX (looked for UPX0, UPX1, or UPX! in the file)
- 65 different groups identified
  - Includes everything that didn’t match anything else (catch all) – 76 samples
- 31 unique linker versions
- 13 unique number of sections
  - 6902 have 3 sections

<table>
<thead>
<tr>
<th>Group Label</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3564</td>
</tr>
<tr>
<td>1</td>
<td>1694</td>
</tr>
<tr>
<td>0</td>
<td>821</td>
</tr>
<tr>
<td>25</td>
<td>305</td>
</tr>
<tr>
<td>24</td>
<td>297</td>
</tr>
<tr>
<td>83</td>
<td>113</td>
</tr>
<tr>
<td>123</td>
<td>111</td>
</tr>
</tbody>
</table>
# UPX numbers

## PEiD Label

<table>
<thead>
<tr>
<th>PEiD Label</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPX 2.93 - 3.00 [LZMA] -&gt; Markus Oberhumer, Laszlo Molnar &amp; John Reiser:UPX</td>
<td>3453</td>
</tr>
<tr>
<td>3.02:UPX v3.0 (EXE_LZMA) -&gt; Markus Oberhumer &amp; Laszlo Molnar &amp; John Reiser</td>
<td></td>
</tr>
<tr>
<td>UPX v0.89.6 - v1.02 / v1.05 -v1.24 -&gt; Markus &amp; Laszlo [overlay]</td>
<td>1121</td>
</tr>
<tr>
<td>None</td>
<td>506</td>
</tr>
<tr>
<td>UPX v0.89.6 - v1.02 / v1.05 -v1.22 (Delphi) stub</td>
<td>297</td>
</tr>
</tbody>
</table>

## Group Label

<table>
<thead>
<tr>
<th>Group Label</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3564</td>
</tr>
<tr>
<td>1</td>
<td>1694</td>
</tr>
<tr>
<td>0</td>
<td>821</td>
</tr>
<tr>
<td>25</td>
<td>305</td>
</tr>
<tr>
<td>24</td>
<td>297</td>
</tr>
</tbody>
</table>
Solution recap

- Easy to generate signatures
  - Python script with minimal dependencies

- It involves Math, who doesn’t love Math?

- Cross platform.
  - Python works everywhere, right?

- Easy to understand...ish

- It Works!
Future work

The int3s are a side effect of the compiler adding bytes between functions or keeping aligned addresses (http://hooked-on-mnemonics.blogspot.com/2013/08/exploring-functions-with-undefinderpy.html)
Questions

I’m done
References

- Capstone Engine
- PEFile
- Tapered Levenshtein
- ZeuS dataset
- APT1 dataset