Malware Analysis Without Looking At Assembly Code

John Franco

Electrical and Computer Engineering
Computer Science
Malware

What:

**Virus**: computer program, hidden in another program, plants copies of itself in other programs and usually performs a malicious action

**Worm**: small, self-contained, self-replicating program, invades networked computers and usually performs a malicious action

**Trojan Horse**: seemingly useful program, has concealed instructions which perform a malicious action when executed (remotely)

**Spyware**: installed without a user's knowledge, transmits information about user activities over the Internet

**Adware**: transmit activities to advertisers

**Backdoor**: allows attacker in without credentials

**Rootkit**: backdoor in a now modified OS
Malware Analysis

Classes of COTS Tools:

**Sniffer**: monitor and analyze network traffic

**Disassembler**: generate assembly code from binary

**Debugger**: allows observation of code execution as it runs

**Decompiler**: generate readable high level code from binary

**Special Purpose**: lots
Malware Analysis

Goals of malware analysis:

Understand how a particular malware works so defenses against it can be developed

- where did it come from and how did it get here?
- who is the intended target?
- what does the malware do?
- how does the malware interact with the network?
- how does the malware interact with the attacker?
Malware Analysis

Types of malware analysis:

- **Static Analysis**: look at and walk through code

- **Dynamic Analysis**: look at behavior of the code
  - Is there a command-and-control channel?
  - What exactly gets installed?

Both types of analysis are needed to get a complete picture of what the malware is trying to do and how it may be stopped.
Malware Analysis

Malware may employ one or more packers:

A packer compresses code in a normal way but the code is decompressed directly into RAM when executed, it is not decompressed into a file.

Packers are used to make the malware less detectable

Anti-virus software may not be able to detect the malware

Unfortunately, there are hundreds of packers that can be used and the AV software can't manage that number, let alone new ones

Encryption is also used – even if AV software can unpack, it probably won't be able to decrypt – anyway, no need to use packing if strong encryption is employed
Malware Analysis

Malware may employ encryption:

Any significant strings in the malware are encrypted using a custom encryption scheme. This means:

1. command and control domains can be hard-coded in the malware instead of having to be generated by the malware (such generators provide signatures)
2. names of functions used by the malware are decrypted at runtime. An analyst must figure out the encryption before progress can be made

Communications to the attacker may be encrypted

1. network analysis is made more difficult
2. changing the encryption is easy for the attacker
Entropy

What:
Entropy is a measure of the randomness in a string of bytes. Alternatively, it is the probability of predicting a character in a string from a given position in the string. It is also known as Information Density.

Definition:
Let $X$ be an alphabet of $n$ letters, let $p_i$ be the probability that the $i^{th}$ letter appears in a string. Then the entropy of $X$ is

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Examples:
Random, 4 letters: $H(X) = -4 \times (1/4) \times (-2) = 2$
Next letter completely known: $H(X) = 0$ (log 1 = 0)
Use in Malware Detection:

Calculate entropy of sections of a file – sections are determined from the file headers (exe file).

If the entropy of a section is high, file may be encrypted or compressed.

Distinction between sections is blurred if file is packed.

Binaries have large blocks of 0s, biasing the entropy – they must be removed from consideration.

Compute average and high entropy and compare against entropy results in a database.

Try: prompt> binwalk -E <file>
Use in Malware Signatures and Detection:

Roughly, there are 13 sections in an executable – 8 sections due to the PE format and 5 sections due to packing algs.

Experiments can produce a chromatographic-like separation based on percentage occurrence of entropy greater than some threshold amount, such as that on the right – a section is indicated as a color and the length of the color is the number of times the section had high entropy divided by the total number of times any section had high entropy over a large set of samples.

This can be done for malware sample sets and good sample sets.

Gives clues:
1) whether malware is suspect
2) where it might be – then use Ghidra
Hashing

What:
A mapping from a long string to a small number of bytes say 20 or 40 bytes that is very hard to reverse and it is highly improbable that two strings (from exe files) map to the same hash value

Properties of a Cryptographic Hash:
1. any bit in output should be 1 about half the time
2. any output should have roughly half its bits set to 1
3. any two outputs should be statistically uncorrelated

Algorithms for Cryptographic Hashing:
SHA-1  20 bytes
SHA-2  28 to 64 bytes
SHA-3  28 to 64 bytes
MD5    16 bytes
Hashing

Examples:

On: “Now is the time for all good men to come to the aid”
MD5: 651f0fb6d21c296b0aa1382fa70527d9
SHA-256: b9cb110d62db60cdden0391dc8034e0870edeb443f219100c739be33806405

On: “Now is the time for all good_men to come to the aid”
MD5: 3c8a07e525d79c591865759030fa4072
SHA-256: 34312299db641e8e427f2b9d98049967ebd530f3b0bf1a8a6aa65a8c377d6c6c

If someone tries to modify a file that we have a good hash for, we will be able to determine this happened by taking a hash of the modified file and comparing
Hashing

Examples:

On: “Now is the time for all good men to come to the aid”
MD5: 651f0fb6d21c296b0aa1382fa70527d9
SHA-256: b9cb110d62db60cdff0391dc8034e0e870ededb443f219100cba739be33806405

On: “Now is the time for all good _ men to come to the aid”
MD5: 3c8a07e525d79c591865759030fa4072
SHA-256: 34312299db641efe427f2b9d98049967ebd530f3b0bf1a8a6aa65a8c377d6c6c

But how well does this work when trying to see if two files are similar? Not very
This is what we want to do when attackers slightly change Their software
Hash on sections of a file to get a class signature

Examples:

Use uniform sections – say 13 bytes
On: “now is the time for all good men to come to the aid”
MD5: 1aff8176559f08244fbc247ff491fc0f
  7b87c6da8571d58751c54d7a3d8aab49
  4207a13d9563243dbf8fabc2a5ea2d21
  e8ccff22fa89baea5924366fae1bd5e0

On: “now is the time for all good_men to come to the aid”
MD5: 1aff8176559f08244fbc247ff491fc0f
  7b87c6da8571d58751c54d7a3d8aab49
  f0c3ca5031446605e80d32b2f185063c
  e8ccff22fa89baea5924366fae1bd5e0

Now we need a good comparison algorithm!
Hashing

Rolling Hash:

**Given:** hash function that produces 4 bytes
Accumulate hashes over the last n bytes in a given string
x, y, z, c, d: integers; w: array of s elements

At any position p in the input, the state of the rolling hash will depend only on the last 4 bytes of the file.

The rolling hash function F is constructed so that it is possible to remove the influence of one of the terms. Thus, given hash r at position p, it is possible to compute r at position p+1 by removing the influence of byte p-q (f(b_{p-q})), and adding the influence of byte p+1 (g(b_{p+1})).
Rolling Hash:

**Given:** hash function that produces 4 bytes

Accumulate hashes over the last n bytes in a given string

\[ x, y, z, c, d: \text{integers}; \quad w: \text{array of } s \text{ elements} \]

```plaintext
update (d) {
    y = y - x
    y = y + s * d
    x = x + d
    x = x - w[c mod s]
    w[c mod s] = d
    c = c + 1
    z = z << 5
    z = z ⊕ d
    return (x + y + z)
}
```
# Hashing

## Rolling Hash Example:

\( s = 4, \)

Input: “now is the time for all good men to come to the aid”

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6e</td>
<td>6e</td>
<td>1b8</td>
<td>6e</td>
<td>1</td>
<td>6e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>294</td>
</tr>
<tr>
<td>6f</td>
<td>dd</td>
<td>306</td>
<td>daf</td>
<td>2</td>
<td>6e</td>
<td>6f</td>
<td>0</td>
<td>0</td>
<td>1192</td>
</tr>
<tr>
<td>77</td>
<td>154</td>
<td>405</td>
<td>1b597</td>
<td>3</td>
<td>6e</td>
<td>6f</td>
<td>77</td>
<td>0</td>
<td>1baf0</td>
</tr>
<tr>
<td>20</td>
<td>174</td>
<td>331</td>
<td>36b2c0</td>
<td>4</td>
<td>6e</td>
<td>6f</td>
<td>77</td>
<td>20</td>
<td>36b765</td>
</tr>
<tr>
<td>69</td>
<td>16f</td>
<td>361</td>
<td>6d65869</td>
<td>5</td>
<td>69</td>
<td>6f</td>
<td>77</td>
<td>20</td>
<td>6d65d39</td>
</tr>
<tr>
<td>73</td>
<td>173</td>
<td>3be</td>
<td>dacb0d53</td>
<td>6</td>
<td>69</td>
<td>73</td>
<td>77</td>
<td>20</td>
<td>dacb1284</td>
</tr>
<tr>
<td>20</td>
<td>11c</td>
<td>2cb</td>
<td>5961aa40</td>
<td>7</td>
<td>69</td>
<td>73</td>
<td>20</td>
<td>20</td>
<td>5961ae27</td>
</tr>
<tr>
<td>74</td>
<td>170</td>
<td>37f</td>
<td>2c354874</td>
<td>8</td>
<td>69</td>
<td>73</td>
<td>20</td>
<td>74</td>
<td>2c354d63</td>
</tr>
<tr>
<td>68</td>
<td>16f</td>
<td>3af</td>
<td>86a90ee8</td>
<td>9</td>
<td>68</td>
<td>73</td>
<td>20</td>
<td>74</td>
<td>86a91406</td>
</tr>
<tr>
<td>65</td>
<td>161</td>
<td>3d4</td>
<td>d521dd65</td>
<td>10</td>
<td>68</td>
<td>65</td>
<td>20</td>
<td>74</td>
<td>d521e29a</td>
</tr>
</tbody>
</table>
## Hashing

**Rolling Hash Example:**

\[ s = 4, \]

Input: “now is the time for all good men to come to the aid”

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6e</td>
<td>6e</td>
<td>1b8</td>
<td>6e</td>
<td>1</td>
<td>6e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>294</td>
</tr>
<tr>
<td>6f</td>
<td>dd</td>
<td>306</td>
<td>daf</td>
<td>2</td>
<td>6e</td>
<td>6f</td>
<td>0</td>
<td>0</td>
<td>1192</td>
</tr>
<tr>
<td>77</td>
<td>154</td>
<td>405</td>
<td>1b597</td>
<td>3</td>
<td>6e</td>
<td>6f</td>
<td>77</td>
<td>0</td>
<td>1baf0</td>
</tr>
<tr>
<td>37</td>
<td>18b</td>
<td>38d</td>
<td>36b2d7</td>
<td>4</td>
<td>6e</td>
<td>6f</td>
<td>77</td>
<td>20</td>
<td>36b7ef</td>
</tr>
<tr>
<td>69</td>
<td>186</td>
<td>3a6</td>
<td>6d65a89</td>
<td>5</td>
<td>69</td>
<td>6f</td>
<td>77</td>
<td>20</td>
<td>6d65fb5</td>
</tr>
<tr>
<td>73</td>
<td>18a</td>
<td>3ec</td>
<td>dacb5153</td>
<td>6</td>
<td>69</td>
<td>73</td>
<td>77</td>
<td>20</td>
<td>dacb56c9</td>
</tr>
<tr>
<td>20</td>
<td>133</td>
<td>2e2</td>
<td>596a2a40</td>
<td>7</td>
<td>69</td>
<td>73</td>
<td>20</td>
<td>20</td>
<td>596a2e55</td>
</tr>
<tr>
<td>74</td>
<td>170</td>
<td>37f</td>
<td>2d454874</td>
<td>8</td>
<td>69</td>
<td>73</td>
<td>20</td>
<td>74</td>
<td>2d454d63</td>
</tr>
<tr>
<td>68</td>
<td>16f</td>
<td>3af</td>
<td>a8a90ee8</td>
<td>9</td>
<td>68</td>
<td>73</td>
<td>20</td>
<td>74</td>
<td>a8a91406</td>
</tr>
<tr>
<td>65</td>
<td>161</td>
<td>3d4</td>
<td>1521dd65</td>
<td>10</td>
<td>68</td>
<td>65</td>
<td>20</td>
<td>74</td>
<td>1521e29a</td>
</tr>
</tbody>
</table>
Locality Sensitive Hashing

**SpamSum:** algorithmic technique that hashes similar input items into the same "buckets" with high probability. The number of buckets is much smaller than the universe of possible input items.

Since similar items end up in the same buckets, this technique can be used for data clustering and nearest neighbor search.

It differs from conventional hashing techniques in that hash collisions are maximized, not minimized.
Locality Sensitive Hashing

Elements of the Spamsum Algorithm:

\[ b = \text{compute initial block size}(\text{input}); \]

\[ \text{sig}[0] = \text{""}; \text{sig}[1] = \text{""}; \]
\[ \text{mark}[0] = 0; \text{mark}[1] = 0; i = 0; \]

while (\( (d = \text{input}[i++]) \neq \text{NULL} \)) {
    \[ r = \text{update}(d) \]
    if (\( r \% b = b - 1 \)) {
        \[ \text{sig}[0] += \text{md5sum}(\text{input}[\text{mark}[0]]...\text{input}[i]) \% 64; \]
        \[ \text{mark}[0] = i; \]
    }
    if (\( r \% (b*2) = b*2 - 1 \)) {
        \[ \text{sig}[1] += \text{md5sum}(\text{input}[\text{mark}[1]]...\text{input}[i]) \% 64; \]
        \[ \text{mark}[1] = i; \]
    }
}

signature = \( b+ \text{"::"} +\text{sig}[0]+ \text{"::"} +\text{sig}[1] \)
Fuzzy Hashing

Comparing Spamsum Results:

Sample signature:
96:
RVZs5AHNMGXq08UrOa0l/7U25wTyTjH+dUW557B5RE8shXMn+ca9WagVQR3m46Pq:
RvuGHCUS/7U25wTyH+dUWP7C8sh8nJU,
"~/Courses/C6055/Lectures/MalwareAnalysis/create-sigs.txt"

Signature of similar file:
96:
RVZs5AHNZGXq0TUrOa0l/7U25wJTTjH+dUW557B5RE8shXMn+pa9WagVQR3m46Piu:
RvuGHLuh/7U25wJnH+dUWP7C8sh8niao,
"~/Courses/C6055/Lectures/MalwareAnalysis/create-sigs-1.txt"

How to compare the two?
What can be done for sig[0] to match sig[1]?

- a character may be removed from some signature 1
- a character may be inserted into some signature 1
- a character may be changed to a different character 3
- two characters may be swapped 5
Fuzzy Hashing

Comparing Spamsum Results:

Sanity check:
Let signature 0 have \( l_0 \) length and signature 1 have \( l_1 \) length

\[
\text{#insertions + #deletions} = |l_0 - l_1|
\]

\[
\text{#changes + #swaps} < \min (l_0, l_1) \quad \text{- a swap of two means neither should be changed}
\]

Edit Distance:

\[
E = \text{#insertions} + \text{#deletions} + 3\text{#changes} + 5\text{#swaps}
\]

Similarity Score:

\[
M = 100 \left(1 - \frac{E}{l_0 + l_1}\right)
\]

Two identical files: \( E=0, M=100 \)
Two same size sigs, all characters different: \( E=3*l_0, M= -50 \)
Fuzzy Hashing

Uses:

- Identify files that are not identical but close to each other
  - identify malware variants by matching to known malware samples

- Truncated files can be matched to their originals
  - files missing headers and not viewable can be matched to known files

- File containing pages, each of which is truncated
Malware Analysis

Malware can sometimes be identified by strings it contains:

Malware may use mutex objects so that it will not re-infect an already infected machine.

For a list of mutex objects see

Also: http://gauss.ececs.uc.edu/Courses/c6055/extra/mutants.txt

However, the mutex may be computed from some information that is specific to the machine it is running on such as the product ID. Thus, the mutex may be different on different machines and cannot be used to identify the malware.

Strings can reveal some things about the malware:

http strings may be used to leak information to attacker
https://blog.httpwatch.com/2009/02/20/how-secure-are-query-strings-over-https/
A collection of strings may reveal a password guesser
A string may reveal the place of origin of the malware
Malware Analysis

Yara: (the pattern matching swiss army knife for malware researchers)

Identify and classify malware based on usual and unusual conditions

Write rules:

```
rule BadBoy  {
    strings:
      $a = "win.exe"
      $b = "http://foo.com/badfile1.exe"
      $c = "http://bar.com/badfile2.exe"
    condition:
      $a and ($b or $c)
}
```

files or processes containing the string `win.exe` and any of the two URLs must be reported as **BadBoy**

Rules: https://github.com/Yara-Rules/rules

http://yara-generator.net/ Yara online rule generator
https://blog.malwarebytes.com/threat-analysis/2013/10/using-yara-to-attribute-malware/
Malware Analysis

Rootkits:

cracker installs rootkit after obtaining user-level access, either by exploiting a vulnerability or cracking a password allows attacker to mask intrusion and gain root or privileged access to one computer or other machines on the network
rootkits have been known to come from seemingly innocent DRM components on a SONY audio CD!

rootkits may be exploited by malware if they have vulnerabilities!!

Detection:

1. look for telltale strings
2. look for calls to library functions that are hidden from the Windows API, Master File Table, and directory index
3. look for library calls that are redirected to other functions, or load device drivers

Removal:

Better off just reinstalling clean OS
Malware Analysis

Rootkit detection in ubuntu:

rkhunter -c:

- MD5 hash compare
- look for default files used by rootkits
- wrong file permissions for binaries
- look for suspected strings in Loadable Kernel Module & and Kernel Load modules
- look for hidden files

Other:

* Examine log files for connections from unusual locations
* Look for setuid and setgid files (especially setuid root files)
  
  `find / -user root -perm -4000 -print`
  
  `find / -group kmem -perm -2000 -print`
* Check whether system binaries have been altered.
* Examine all the files that are run by 'cron' and 'at.'
* Check for unauthorized services.
* Examine the /etc/passwd file on the system
* Check system and network config files
* Look for unusual or hidden files
Malware Analysis

Backdoor:
undocumented portal that allows someone, say admin, to access the OS by bypassing the usual credential checks

traditional backdoor: can be accessed by anyone
asymmetric backdoor: can only be accessed by the planter
cannot be detected (for the most part)

kleptographic attack: uses asymmetric encryption to install a cryptographic backdoor
https://en.wikipedia.org/wiki/Kleptography

OpenSSL RSA Backdoor: experimental backdoor planted in RSA key generation (chap 10 below)
https://www.academia.edu/18471944/Chapter_10_An_Elliptic_Curve_Asymmetric_Backdoor_in_OpenSSL_RSA_Key_Generation

Removal:
forget removal – just reinstall OS
Malware Analysis

Virus:
Feature extraction and classification problem