Cyber Defense Overview

Malware Analysis Without Looking At Assembly Code

John Franco
Electrical Engineering and Computing Systems

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 typically employs a packer:

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 typically employs 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 are 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

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: $H(X) = -4*(1/4)*(-2) = 2$

Next letter completely known: $H(X) = 0$ (log 1 = 0)

Entropy

Use in Malware Signatures and Detection:

Calculate entropy of sections of a file – sections are determined from the file header if it is an `exe` file.

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

If the file has been packed, the distinction between sections Is blurred.

Also executables have large blocks of 0 values, biasing the entropy – they need to be removed from consideration.

Also compute average and high entropy and compare against entropy results in a database.

Entropy

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 IDA Pro

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: `b9cb110d62db60cdff0391dc8034e0e870edeb443f219100cba739be33806405`

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

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: `b9cb110d62db60cdff0391dc8034e0e870edeb443f219100cba739be33806405`

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

Hashing

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`: 4 bytes; `w`: array of `s` elements

```
update (d) {
  y = y − x
  y = y + size * 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”

	d	x	y	z	c	w[0]	w[1]	w[2]	w[3]	r
n	6e	6e	1b8	6e	1	6e	0	0	0	294
o	6f	dd	306	daf	2	6e	6f	0	0	1192
w	77	154	405	1b597	3	6e	6f	77	0	1baf0
	20	174	331	36b2c0	4	6e	6f	77	20	36b765
i	69	16f	361	6d65869	5	69	6f	77	20	6d65d39
s	73	173	3be	dacb0d53	6	69	73	77	20	dacb1284
	20	11c	2cb	5961aa40	7	69	73	20	20	5961ae27
t	74	170	37f	2c354874	8	69	73	20	74	2c354d63
h	68	16f	3af	86a90ee8	9	68	73	20	74	86a91406
e	65	161	3d4	d521dd65	10	68	65	20	74	d521e29a

Hashing

Rolling Hash Example:

s = 4,

Input: “now7is the time for all good men to come to the aid”

	d	x	y	z	c	w[0]	w[1]	w[2]	w[3]	r
n	6e	6e	1b8	6e	1	6e	0	0	0	294
o	6f	dd	306	daf	2	6e	6f	0	0	1192
w	77	154	405	1b597	3	6e	6f	77	0	1baf0
7	37	18b	38d	36b2d7	4	6e	6f	77	20	**36b7**ef
i	69	186	3a6	6d65a89	5	69	6f	77	20	**6d65**fb5
s	73	18a	3ec	dacb5153	6	69	73	77	20	**dacb**56c9
	20	133	2e2	596a2a40	7	69	73	20	20	**596**a2e55
t	74	170	37f	2d454874	8	69	73	20	74	2d4**54d63**
h	68	16f	3af	a8a90ee8	9	68	73	20	74	a8**a91406**
e	65	161	3d4	1521dd65	10	68	65	20	74	1**521e29a**

Hashing

Elements of the Spamsum Algorithm:

```
b = compute initial block size(input);

sig[0] = “”; sig[1] = “”;
mark[0] = 0; mark[1] = 0; i = 0;
while ((d = input[i++]) != NULL) {
    r = update(d)
    if (r % b = b – 1) {
        sig[0] += md5sum(input[mark[0])...input[i]) % 64;
        mark[0] = i;
    }
    if (r % (b*2) = b*2 – 1) {
        sig[1] += md5sum(input[mark[1])...input[i]) % 64;
        mark[1] = i;
    }
}

signature = b+ ":" +sig[0]+ ":" +sig[1]
```

Fuzzy Hashing

Comparing Spamsum Results:

Sample signature:

```
96:
RVZs5AHNMGXq08UrOaOl/7U25wTyTjH+dUW557B5RE8shXMn+ca9WagVQR3m46Pq:
RvuGHCUS/7U25wTynH+dUWP7C8sh8nJU,
"~/Courses/C6055/Lectures/MalwareAnalysis/create-sigs.txt"
```

Signature of similar file:

```
96:
RVZs5AHNZGXq0TUrOaOl/7U25wJTjH+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]? cost

1. a character may be removed from some signature 1
2. a character may be inserted into some signature 1
3. a character may be changed to a different character 3
4. 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

#insertions + #deletions = $|l_0 - l_1|$

#changes + #swaps < min (l_0, l_1) - a swap of two means neither should be changed

Edit Distance:

E = #insertions+#deletions+3#changes+5#swaps

Similarity Score:

$M = 100 (1 – E/(l_0 + l_1))$

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
- 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

http://hexacorn.com/examples/2014-12-24_santas_bag_of_mutants.txt
Also: ~/Courses/C6055/Lectures/MalwareAnalysis/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
A collection of strings may reveal a password guesser
A string may reveal the place of origin of the malware

Malware Analysis

Yara:

Identify 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.readthedocs.org/en/latest/writingrules.html`

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 can be exploited by any malware!!

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 LKM and KLD 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 assymetric 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)
`http://www.cryptovirology.com/cryptovfiles/newbook.html`

Removal:
forget removal – just reinstall OS