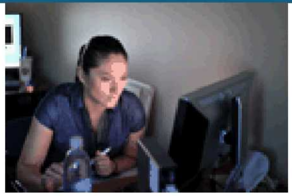
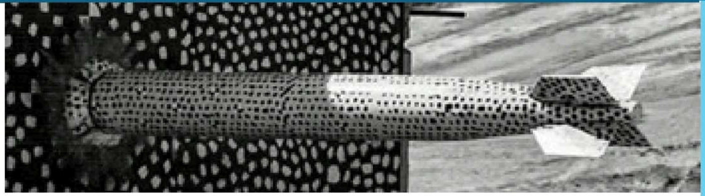
SAND2019-11305PE

Signature Matching for Incident Response





PRESENTED BY

Charles Smutz

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Outline

Intro

Theory

Implementations

- Yara
- Suricata (Snort)
- grep
- BPF

Signature Writing Advice

Lab

Intro: Charles Smutz

2006: BS IT BYU
- First TA for IT digital forensics course

2007: Lockheed Martin: Computer Incident Response Team
- Systems for targeted threats: Network Sensors, Malware Analysis, Data Analysis

2016: PhD IT GMU
- Dissertation: Countering Malicious Documents and Adversarial Learning

2016: Sandia National Labs: Cyber Security R&D
- R&D to support Computer Network Defense

Open Source Activity: Minor contribution or extension to most projects discussed

https://github.com/VirusTotal/yara/issues/108
https://redmine.openinfosecfoundation.org/users/210
https://github.com/lmco/vortex-ids/blob/master/libbsf/libbsf.c

Intro: Sandia National Labs

Albuquerque, NM (also Livermore, CA and other small sites)

Manhattan Project: non-nuclear portions of nuclear weapons

"exceptional service in the national interest" – Harry S Truman

R&D: Applied research, usually funded by US Gov

Reasons to work at a National Lab:

- Benefits, work-life balance
- Time and resources to do true research (difficult in industry)
- Data necessary to fulfill important missions (difficult in academia)

Signature Matching Theory

Signature Matching Theory

Motivation
- Understand how pattern matching works -> how to write efficient signatures
- Prepare for bugs (features), tweaks, and extensions

Naïve String Matching

Single String Matching (Boyer-Moore)

Multi-String Matching (Aho-Corasick/state machine)

Regular Expressions

Bit-parallel approaches

Other approaches

Example Notation

Text

- T = “The quick brown fox jumps over the lazy dog”
- n = len(T)
- i = text counter (T[i], i = 0..n)

Pattern

- P = “over”
- m = len(P)
- j = pattern counter (P[i], i = 0..m)

C and python inspired, heavily simplified pseudocode

- Goal is concise and simple as possible

Big O notation

Naïve Single String Search

```
for(i=0; i < n - m; i++)
  for(j=0; j < m; j++)
    if (T[i + j] != P[j])
      break
    if (j == m - 1)
      print("match at %i\n",i)
```

```
The quick brown fox jumps over the lazy
0,0 "T" != "o"
1,0 "h" != "o"
2,0 "e" != "o"
3,0 " " != "o"

11,0 "r" != "o"
12,0 "o" == "o"
12,1 "w" != "v"
13,0 "w" != "o"
```

Partial Match

```
25,0 " " != "o"
26,0 "o" == "o"
26,1 "v" == "v"
26,1 "e" == "e"
26,1 "r" == "r"
```

Match

Best Case: O (n)

Worst Case: O (nm)

Faster Single String Matching (Boyer-Moore simplified)

o	0
v	1
e	2
r	3

```
for(j=0; j < m; j++)
  L[P[i]] = i
```

```
for(i = m - 1; m < n - m; i += m)
  if (not T[i] in L)
    continue
  else
    k = i - L[T[i]]
    for(j = 0; j < m; j++)
      if (T[k + j] != P[j])
        break
      if (j == m - 1)
        print("match at %i\n",k)
```

```
The quick brown fox jumps over the lazy
3,0 " " not in "over"
7,0 "c" not in "over"
11,0 "r" in "over"
8,0 "k" != "o"
15,0 " " not in "over"
19,0 " " not in "over"
23,0 "p" not in "over"
27,0 "v" in "over"
26,0 "o" == "o"
26,1 "v" == "v"
26,2 "e" == "e"
26,3 "r" == "r"
```

Partial Match (11,0 "r" in "over" / 8,0 "k" != "o")

Match (26,0 … 26,3)

Faster Single String Matching (Boyer-Moore simplified) (cont)

Key Optimization: at each test in T, check against all chars in P, skip ahead by m

Requires up front processing
- O(m)

Best case: O(n/m)
- Search time ~ 1/m

Worst case: O(nm)

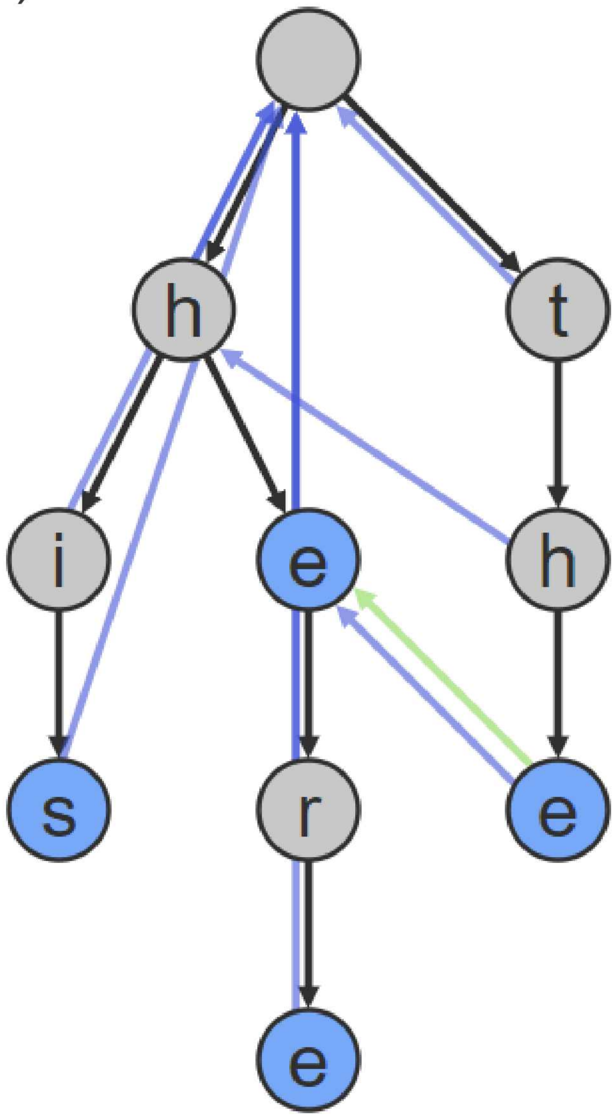
Simplification:
- Boyer-Moore checks last char in P against T as primary check
- Overlap/Repeated Patterns in P: good suffix rule
- Repeated checks: Galil rule

Fast Multiple Pattern Matching (Aho-Corasick, Automata)

Match multiple strings
- Single pass through text
- Simultaneously match all patterns

Aho-Corasick constructs Trie with patterns
- Follow down trie (black lines) when characters matched
- Go back up suffix links (blue lines) when next char doesn't match
- Some nodes/states (blue nodes) represent matches
- Green lines show suffixes that are also matches

Example Text: this hat is the best



Patterns: his here he the

Generator: https://daniel.lawrence.lu/blog/y2014m03d25/
Demo Video: https://www.youtube.com/watch?v=d24CyiU1JFk

Fast Multiple Pattern Matching (Aho-Corasick, Automata) (cont)

State Machine or Automata

Construction: $O(m)$ (m = total length of patterns)

Base Case: $O(n)$

Worst Case: $O(n + z)$ (z = number of matches)

Still competitive, widely used today

Important conclusion: Number of fixed strings doesn't affect scan speed

Regular Expressions

Regular language important part of compiler theory
- lexer splits up input to tokens

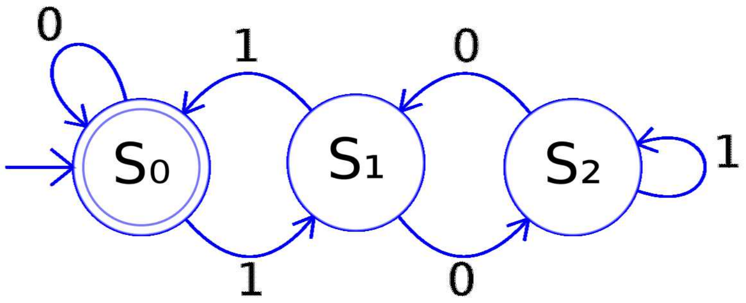
Definition: constructed by concatenation, alternation, or repetition of characters

Alternate definition: accepted by finite state machine

Not a parser: no context or state
- Can check basic structure: validate pair of parenthesis
- Can't check state: can't check nested parenthesis

Catastrophic (pathologic) backtracking
- Some regex engines backtrack to try less greedy quantifiers
  - two quantifiers results in $O(n^2)$ operation to try all combinations

concatenation	alternation	repetition
ab	(a|ε)	a*



state machine to match multiples of 3

Regular Expressions (cont)

Many possible underlying implementations

PCRE (snort)

- Backtracking: unpredictable worst case performance and memory
- Full featured: including backreferences and lookarounds (superset of theoretical regular expressions)
- Well optimized for common use cases

RE2 (~yara)

- State machine based: predicable run time and memory
- Limited functionality: no backreferences/lookarounds
- Goal of safe from untrusted user input

Other features of regex implementations

- Capturing groups
- Character classes
- Lookaheads/lookbehinds

Bit-parallel approaches (leveraging modern processors)

Bit-parallel approaches leverage bitwise logic capabilities of modern processors

- became big research topic ~1990

Special instructions: SIMD, MMX, SSE*, AVX*

- large registers
- multiple operations in a single instruction
- operate on data in parallel
- cache/pipeline friendly

Approximate state machines

- State tracked in bit vectors
- AND/OR operation depending on input
- Shift to advance state

Bit-parallel (Backward Nondeterministic DAWG Matching algorithm)

```
B = 0*256
s = 1
for(j = m - 1; j >= 0; j--)
  B[P[j]] |= s
  s <<= 1

i = 0
while (i <= n - m)
  j = m - 1
  last = m
  d = ~0  //set to all 1s
  while (j >= 0 && d != 0)
    d &= B[T[i+j]]
    j--
    if (d != 0)
      if (j >= 0)
        last = j+1
      else
       printf("match at %i\n",i)
    d <<= 1
  i += last
```

o	1000
v	0100
e	0010
r	0001
*	0000

```
The quick brown fox jumps over the lazy do

0,3 B[" "] == 0000, d == 1111 => 0000
4,3 B["c"] == 0000, d == 1111 => 0000
8,3 B["r"] == 0001, d == 1111 => 0001
8,2 B["b"] == 0000, d == 0010 => 0000
11,3 B["n"] == 0000, d == 1111 => 0000

26,3 B["r"] == 0001, d == 1111 => 0001
26,2 B["e"] == 0010, d == 0010 => 0010
26,1 B["v"] == 0100, d == 0100 => 0100
26,0 B["o"] == 1000, d == 1000 => 1000
```

Partial Match (8,3 – 8,2)

Match (26,0)

Bit-parallel approaches

Complexity doesn’t change (categorically)

- Still approximation of state machine based approach

Current State of art: Hyperscan (~2016)

- Integrated into Suricata, optional pattern matcher, well received
- Bit parallel, SIMD based NFA: extended Shift-OR, can also use AVX2
- Decomposition of patterns into regex + static strings

Challenges/Limitations

- Thermal management (expensive instructions may cause throttling)
- Reliance on speculation

Other approaches

Software

- N-grams, hashing, fuzzy matching

Hardware Specific

- GPGPU, ASIC, FPGA, etc: largely not used due to cost of transfer
  - GPGPU integrated into Suricata, never gained large-scale adoption

Offline Searching

- Inverted indexes (full text search: ex. google, splunk, ELK, etc)

Signature Matching Implementations

Implementations

Yara

Suricata (Snort)

grep

BPF

For each tool:

- Basic Overview
- Implementation details
- Additional features

Yara: Overview

Popular open signature format for files/malware

2008 by VirusTotal

Embedded in many security tools

Allows searching combinations of strings

Pattern types

- static strings
- hex strings (basic wildcards, jumps)
- regex

```
rule Example
{
    strings:
        $a = "text1"
        $b = "text2"
        $c = "text3"
        $d = "text4"

    condition:
        ($a or $b) and ($c or $d)
}
```

Yara: Implementation

Selects 4 byte "atom" from each string, used in Aho-Corasick automaton

- If quality atom doesn't exist, scanning is slowed

Custom Regex engine based on Thompson (VM) as explained by Russ Cox

- Doesn't support backreferences
  - Can be worked around using conditions

Yara: Other features

Operate on attributes of string matches
- count
- location

Access raw data of file (at given offsets)
- Ex. calculate offset

Loops (probably a full Turing machine)

Supports limited modules
- PE parsing

Decoding of file formats done externally
- Ex. decompress zip files

No metadata collection

```
rule CountExample
{
    strings:
        $a = "dummy1"
        $b = "dummy2"

    condition:
        #a == 6 and #b > 10
}
```

```
rule IsPE
{
  condition:
     // MZ signature at offset 0 and ...
     // ... PE signature at offset 0x3C
     uint16(0) == 0x5A4D and
     uint32(uint32(0x3C)) == 0x00004550
}
```

Suricata (Snort): Overview

Popular open signature format for network packets

Snort 1998

Suricata 2010
- Threading
- More metadata collection

String matching on payloads of packets

Keywords for protocol attributes like tcp port and http uri

TLS everywhere limiting effectiveness of network intrusion detection systems

```
alert tcp any any -> 192.168.1.0/24 80 (content: "cgi-bin/phf";
offset: 3; depth: 22; msg: "CGI-PHF access";)
```

Suricata: Implementation

Selects string from each rule for fast pattern matcher (Aho-Corasick)

- After match in fast pattern matcher, test rest of rule

Use libpcre to validate pcre rules

Use Boyer-Moore for single string validation

Support for Hyperscan

Suricata: Other Features

Concept of direction
- HOME_NET: list of networks monitored

Flow bits
- Track state in a flow

Basic protocol decoding
- Ex. HTTP compression

IPS mode

Lua (scripted packet analysis)

File Extraction
- Files format processing externally
  - LaikaBOSS, STOQ, etc

```
alert tcp 192.168.x.x 21 -> any any (msg:
"Successful FTP Login"; content:"Logged on";
flowbits:set,logged_in; flowbits:noalert;
sid:1000012; rev:1;)

alert tcp 192.168.x.x 21 -> !192.168.y.y any
(msg:"Unauthorized File Download";
content:"download from";
flowbits:isset,logged_in; sid:1000013; rev:1;)
```

grep: Overview

Ubiquitous command-line tool for searching files

Ken Thompson in 1974

- Father of Unix philosophy

Used to search (text) files

- logs, configs, etc
- test signatures for malware

```
grep -w "80" /etc/services
```

grep: Implementation and Other Features

grep: Thompson NFA

fgrep: (fixed string) Aho-Corasick multi string matcher

pcregrep: libpcre regex

zgrep: search gzip files

counts, offsets, etc

match context

pipelined with other commands such as awk, sed, sort, etc.

BPF: Overview

Popular open packet filtering syntax

BPF published in 1992, Lawrence Berkeley National Lab

eBPF in linux kernel in 2015

Uses virtual machine in kernel to implement packet filters

- Filter syntax compiled into program for BPF virtual machine
- Filter program run on each packet

```
tcp port 80
```

BPF: Implementation

```
0000    fe ff 20 00 01 00 00 00 01 00 00 00 08 00 45 00
0010    00 30 0f 41 40 00 80 06 91 eb 91 fe a0 ed 41 d0
0020    e4 df 0d 2c 00 50 38 af fe 13 00 00 00 00 70 02
0030    22 38 c3 0c 00 00 02 04 05 b4 01 01 04 02
```

Ethernet
IPv4
TCP

```
~/bpf$ tcpdump -r http.cap -d "host 65.208.228.223"
reading from file http.cap, link-type EN10MB (Ethernet)
(000) ldh      [12]
(001) jeq      #0x800          jt 2    jf 6
(002) ld       [26]
(003) jeq      #0x41d0e4df     jt 12   jf 4
(004) ld       [30]
(005) jeq      #0x41d0e4df     jt 12   jf 13
... { Removed: stuff for ARP and RARP }
(012) ret      #65535
(013) ret      #0
```

Check Ethertype for IPv4

Check IPv4 Source

Check IPv4 Dest

Return True
Return False

BPF: Other Features

Storage in registers and data structures

Not just packets

- System calls filtering, etc

Cloudflare “BPF eats the world”

- DoS protection
- Load balancing and port dispatching
- Metrics and debugging

Easy way to run custom code in kernel: was used in Spectre attacks on linux

Signature Writing Advice

Signature Writing Advice

Understand what must be, what can change in data formats

- Seek to write signatures that match malicious activity categorically and comprehensively

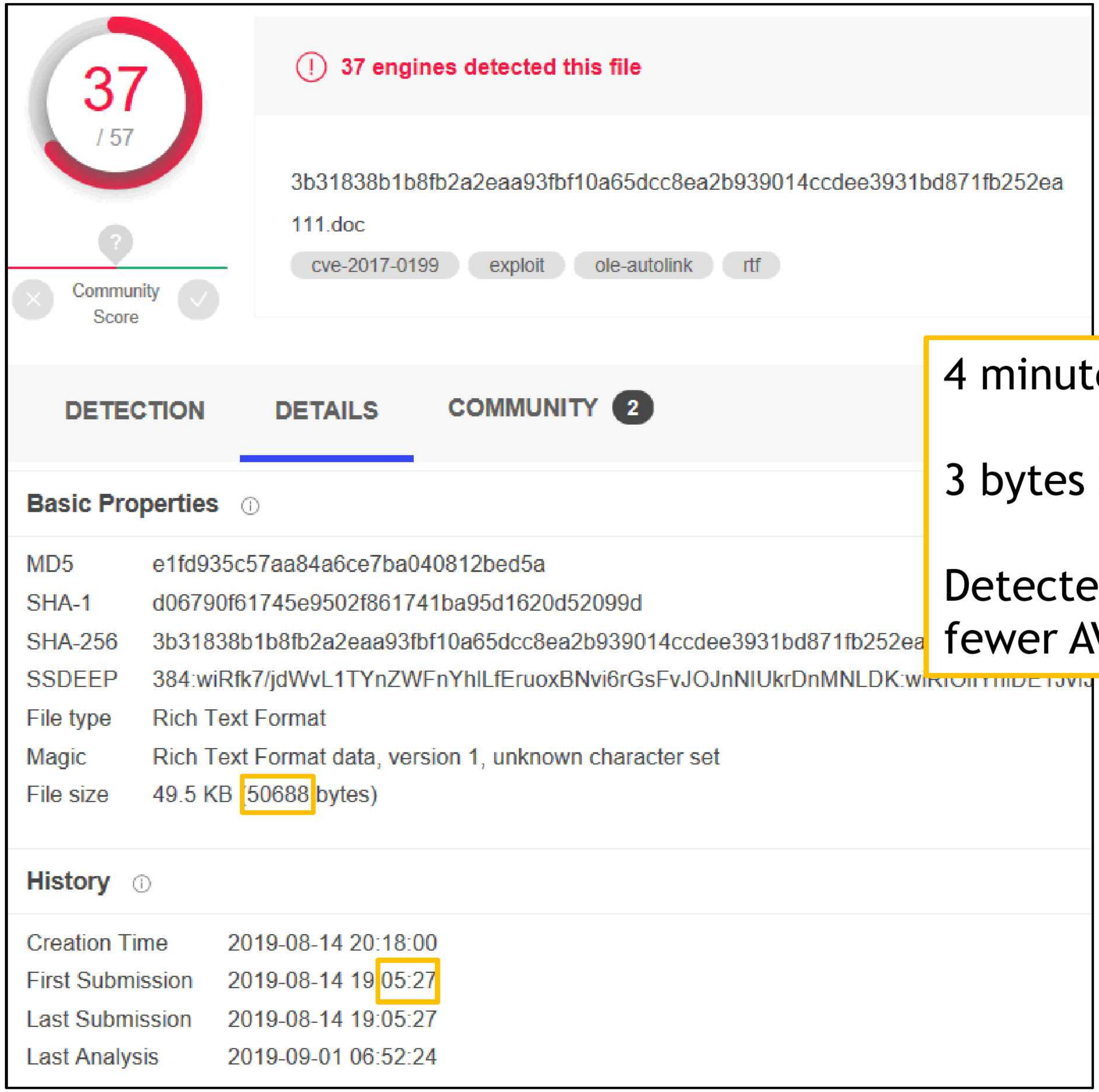
Look for consistencies in malicious data

- Even if coincidental or easy to change

Turn obfuscation/evasion against attackers

Use long, static, unique strings where possible (performance)

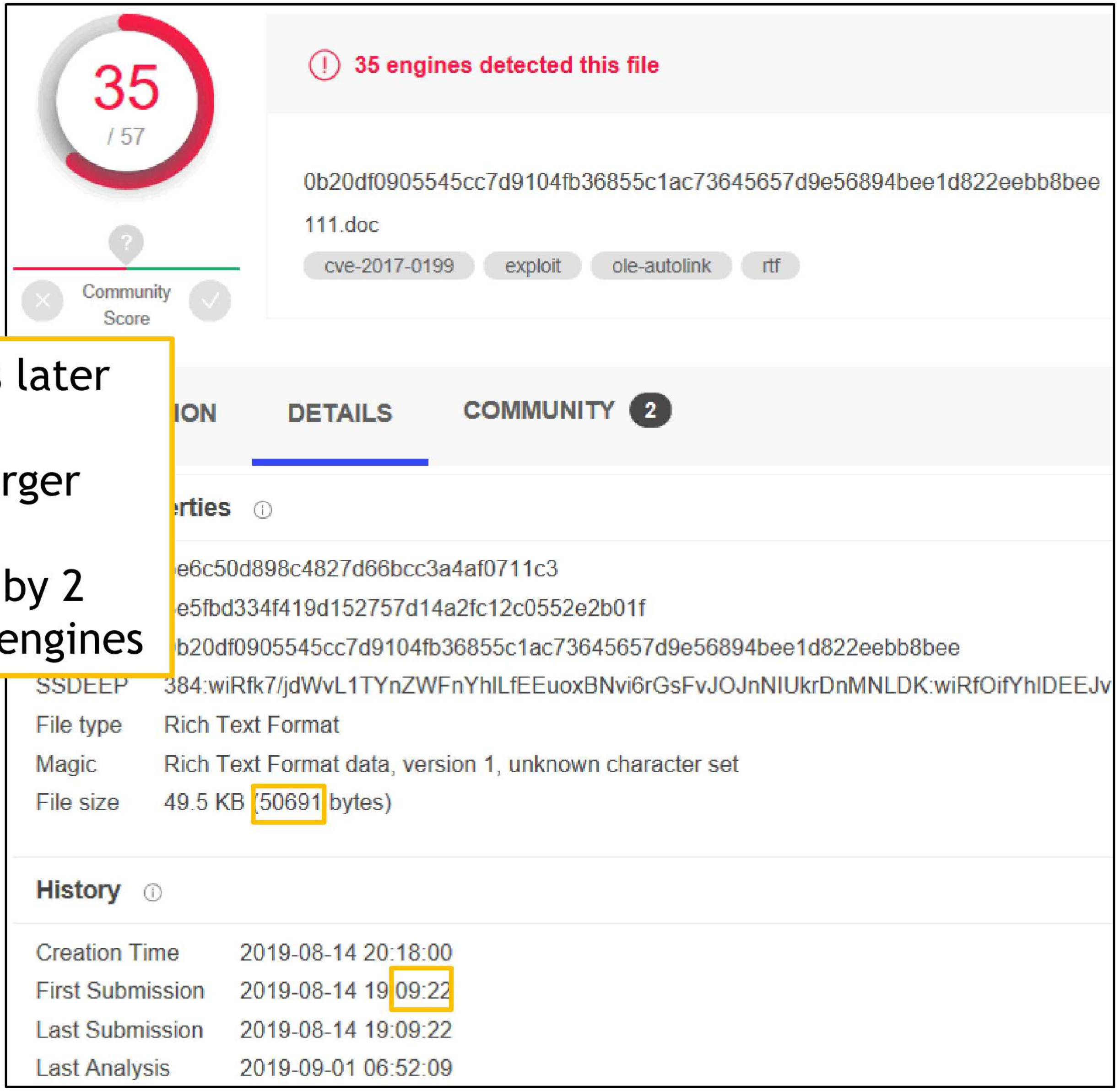
Signature for Signature Evasion Example



4 minutes later

3 bytes larger

Detected by 2 fewer AV engines



Signature for Signature Evasion Example

```
00002cd0 | ..{\object\obja|
00002ce0 |utlink\objupdate|
00002cf0 |\rsltpict\objw93|
00002d00 |56\objh450{\*\ob|
00002d10 |jclass Word.Docu|
00002d20 |ment.8}{\*\objda|
00002d30 |ta 0105000002000|
00002d40 |000090000004f4c4|
00002d50 |5324c696e6b00000|
00002d60 |0000000000000000|
00002d70 |a0000..d0cf11e0a|
00002d80 |1b11ae1000000000|
```

4 byte change

Escaping to hide character in key word

b -> \'62

```
00002cd0 | ..{\object\obja|
00002ce0 |utlink\objupdate|
00002cf0 |\rsltpict\objw93|
00002d00 |56\objh450{\*\o\|
00002d10 |'62jclass Word.D|
00002d20 |ocument.8}{\*\ob|
00002d30 |jdata 0105000002|
00002d40 |0000000900000004f|
00002d50 |4c45324c696e6b00|
00002d60 |0000000000000000|
00002d70 |000a0000..d0cf11|
00002d80 |e0a1b11ae1000000|
```

Signature for Signature Evasion Example

Goal: detect escaped "objclass"

Regex implementation is direct

- Also expensive

```
00002d00 |56\objh450{\*\o\|
00002d10 |'62jclass Word.D|
00002d20 |ocument.8}{\*\ob|
```

Assume that "objclass" is interesting when used with "Word.Document"

Assume "Word.Document" can not be escaped, is case sensitive, etc.

```
rule objclass_obfuscated
{
    strings:
        $magic = "{\\rt"
        $a =
/(o|\\?'6f)(b|\\?'62)(j|\\?'6A)(c|\\?'63)
(l|\\?'6C)(a|\\?'61)(s|\\?'73)(s|\\?'73)/
        $b = "objclass"
    condition:
        $magic at 0 and $a and not $b
}
```

Signature for Signature Evasion Example

Infer consistencies and differences

Use large corpus of samples

- ~5k benign and ~5k malicious RTF files

Should I include "8" in "Word.Document.*"?

- No, that number changes

Other methods for determining consistencies

- Read specification
  - Danger: implementations don't always follow specification
- Read parsing code/RE parser

```
~/malicious_rtfs$ grep -h -E -o -a
"Word\.Document\.." * | sort | uniq -c
      3 Word.Document.
      3 Word.Document.0
    772 Word.Document.1
     13 Word.Document.3
      1 Word.Document.4
      1 Word.Document.5
    145 Word.Document.6
      1 Word.Document.7
    784 Word.Document.8
      2 Word.Document.9
~/benign_rtfs$ grep -h -E -o -a
"Word\.Document\.." * | sort | uniq -c
      1 Word.Document.
     13 Word.Document.1
      2 Word.Document.6
     76 Word.Document.8
```

Signature for Signature Evasion Example

Look for "Word.Document" sans "objclass"

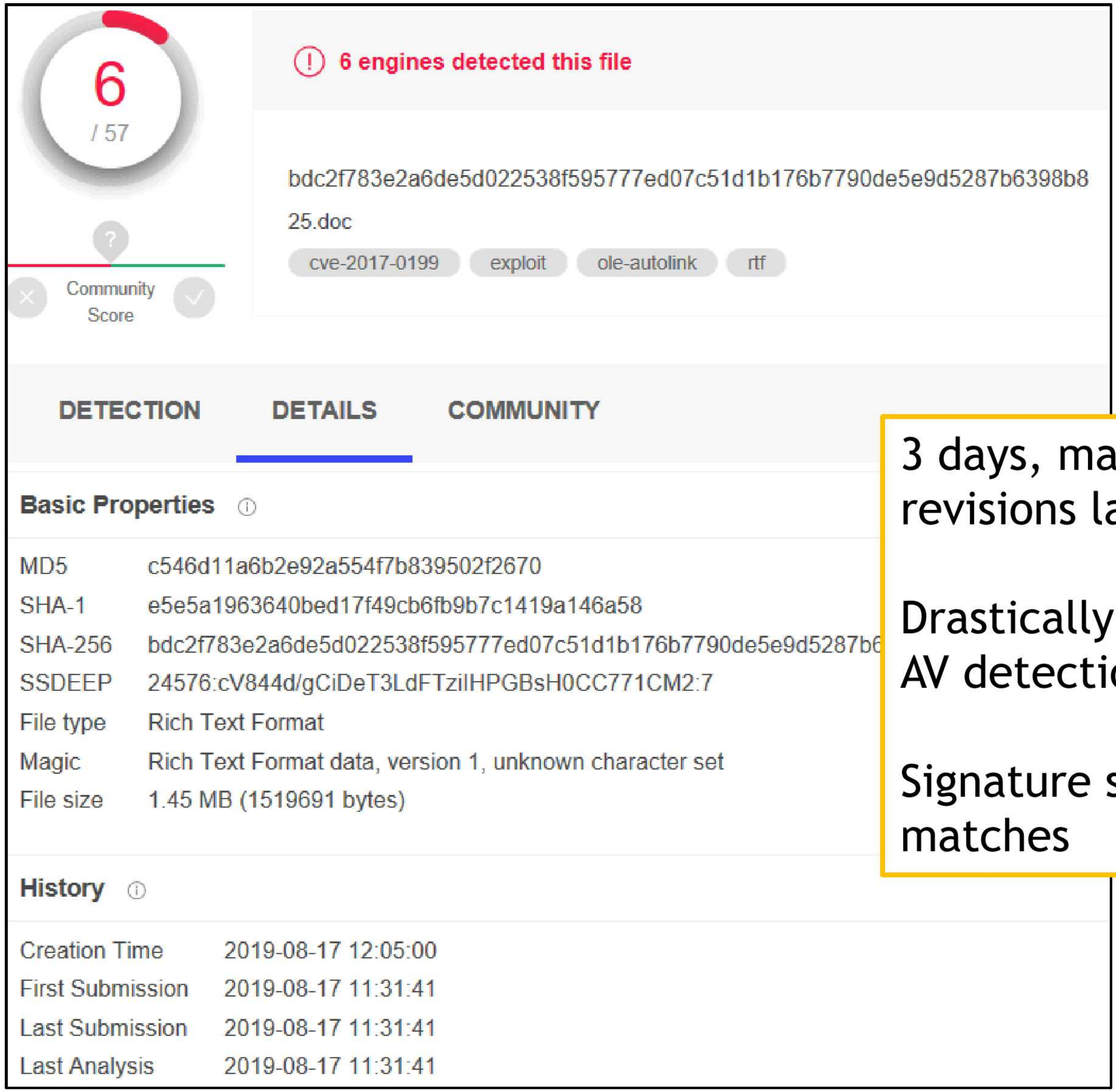
- "objclass" must be obfuscated

Efficient static string matches

```
00002d00 |56\objh450{\*\o\|
00002d10 |'62jclass Word.D|
00002d20 |ocument.8}{\*\ob|
```

```
rule objclass_obfuscated
{
    strings:
        $magic = "{\\rt"
        $a = " Word.Document."
        $b = "objclass"
    condition:
        $magic at 0 and $a and not $b
}
```

Signature for Signature Evasion Example



3 days, many revisions later

Drastically lower AV detection rates

Signature still matches

```
00003280   |.......{\object\|
00003290   |objautlink\objup|
000032a0   |date\rsltpict\ob|
000032b0   |jw9582\objh4047{|
000032c0   |\*\o'62'6Aclass |
000032d0   |Word.Document.8}|
000032e0   |{\*\objdata}....|
000032f0   |..........{\resu|
00003300   |lt {\rtlch\fcs1 |
00003310   |\af0 \ltrch\fcs0|
00003320   | \insrsid1625669|
00003330   |4 {\*\shppict{\p|
00003340   |ict{\*\picprop\s|
00003350   |hplid1027{\sp{\s|
00003360   |n shapeType}{\sv|
```

Signature for Signature Evasion Example

Toy example: signature not broadly applicable

- Only observed associated with small amount of related malware testing on VirusTotal

Oversimplified

- would use offset at location (uint* functions) for greater efficiency

Principles demonstrated are broadly applicable

- Similar evasion techniques are effective
- Signature evasion is a detection opportunity
- Possible to refactor expensive signature to be more efficient
- Use large corpus of samples to test signatures, when possible
  - seek to understand required consistencies and valid variations so signature can be as complete as possible

csmutz@sandia.gov

Preparation

Reading: https://swtch.com/~rsc/regexp/regexp1.html

- Understand how finite state machines apply to pattern matching
  - Input passes through states
- Appreciate performance impact of implementation choices
  - Backtrace or not
  - DFA vs. NFA
- Some historical context
- Motivation for current implementation (re2)
- Optional: understand basic implementation
  - Not necessary to understand C implementation: understand the state machine diagrams

Lab Exercise

Environment

- Recommend common linux distribution such as ubuntu
  - All software should be available as packages: yara, suricata, grep, tcpdump
- Data sources:
  - Text: book from gutenburg such as: https://www.gutenberg.org/ebooks/18
  - Network: pcap from wireshark samples such as: https://wiki.wireshark.org/SampleCaptures?action=AttachFile&do=get&target=http.cap

Yara

- Implement a signature using text strings that looks for a combination of multiple strings
- Demonstrate use of a keyword such as nocase
- Implement the above signature using hex strings
  - Is it possible to implement as a single hex string?
- Implement the above signature using a regular expressions
  - Is it possible to implement as single regular expression?
- Implement signature using uint* functions such file format magic number check

Lab Exercise (cont)

Suricata

- Implement simple packet payload signature that matches packets in a pcap
- Use a protocol keyword in simple signature
- Demonstrate and a more advanced feature of Suricata and explain how it works
  - Ex. file extraction, custom protocol logging, multiple rule signature using flowbits, etc.

grep

- Demonstrate using grep to count instances of patterns/words in large text file
- Demonstrate how at least two different grep variants (grep, egrep, pgrep, etc) handle potentially catastrophic expressions

BPF

- Create a simple packet filter that operates on layer 4 (tcp or udp) attributes
- Dump packet matching code (tcpdump -d) and annotate, explaining meaning of operations