Automatically Classifying Unknown Bots by The REGISTER Messages

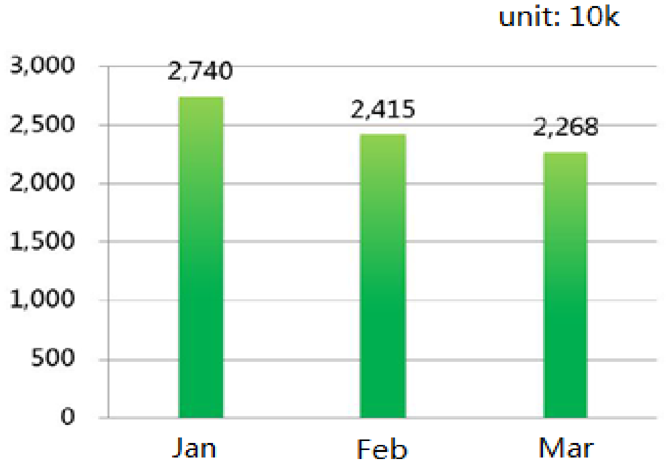
Ya Liu
liuya@360.cn
Network Security Research Lab, Qihoo 360

Outline

- Polymorphism and malware classification
- C&C protocol based classification
- REGISTER message based classification
- Evaluation
- Pitfalls
- Conclusions

Polymorphic Malwares

- A great many of new samples are captured every day



*Malware samples captured in Quarter 1 of 2015

- Most of them are polymorphic variants of known malwares

[*] “2015 first quarter Chinese Internet Security Report”, http://zt.360.cn/1101061855.php?dtid=1101062370&did=1101272883

Malware Classification

- The aim is to classify large number of samples into a relative small number of families
  - e.g., zbot, darkshell, gh0st, ...
- Static sample signatures are heavily used by anti-virus products to build virus signature databases
  - e.g., size, strings, binary code snippets
- It has FP/FN issues when dealing with modern polymorphic malwares

C&C Protocol based Classification

- Most of modern malwares are distributed to build botnets
- It’s proved effective to classify botnets /malware based on their C&C protocols
  - message types/formats/interactions are used
- Detailed C&C protocol specification is a pre-condition
  - Manual RE is necessary in most cases
- Scalability issue

The REGISTER Message

- The first message exchanged in a C&C session, which MUST be sent by the bot
  - It’s also called login, hello, call-home
- Its main usage is to tell the controller:
  - the bot’s machine configs, e.g., OS version, CPU, memory size, net speed
  - hardcoded info copied from sample for verifying
- Many known botnets use this message in their protocols

Common DDoS Bots' REGISTERs

360 INTERNET SECURITY CENTER

bot name	supported OS	OS info in REGISTER?	CPU info in REGISTER	memory info in REGISTER
darkshell	win	yes	yes	yes
elknot	linux/win	yes	yes	yes
XOR DDoS	linux	yes	yes	no
chinaz	linux/win	yes	yes	yes
mayday	linux	yes	yes	no
dofloo	linux/win	yes	yes	yes

Elknot’s REGISTER

- This bot is also called Billgates
- It has variable length and binary format

```
00000000  01 00 00 00 6c 00 00 00  00 f4 01 00 00 32 00 00  ....l... .....2..
00000010  00 e8 03 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
00000020  00 00 01 01 00 00 00 00  01 00 00 00 c0 a8 38 66  ........ ......8f
00000030  c0 a8 38 66 c0 a8 38 66  c0 a8 38 66 c0 a8 38 66  ..8f..8f ..8f..8f
00000040  ff ff 01 00 00 00 00 00  3a 00 02 00 00 00 f9 0d  ........ :.......
00000050  00 00 e0 07 00 00 4c 69  6e 75 78 20 33 2e 31 31  ......Li nux 3.11
00000060  2e 30 2d 31 32 2d 67 65  6e 65 72 69 63 00 47 2d  .0-12-ge neric.G-
00000070  33 2e 30 00                                      3.0.
```

```
00000000  01 00 00 00 76 00 00 00  00 f4 01 00 00 32 00 00  ....v... .....2..
00000010  00 e8 03 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
00000020  00 00 01 01 00 00 00 00  01 00 00 00 c0 a8 38 66  ........ ......8f
00000030  c0 a8 38 66 c0 a8 38 66  c0 a8 38 66 c0 a8 38 66  ..8f..8f ..8f..8f
00000040  ff ff 01 00 00 00 00 00  62 75 79 61 6f 63 61 6f  ........ buyaocao
00000050  77 6f 3a 00 02 00 00 00  f9 0d 00 00 e0 07 00 00  wo:..... ........
00000060  4c 69 6e 75 78 20 33 2e  31 31 2e 30 2d 31 32 2d  Linux 3. 11.0-12-
00000070  67 65 6e 65 72 69 63 00  47 32 2e 30 30 00        generic. G2.00.
```

```
00000000  01 00 00 00 73 00 00 00  00 f4 01 00 00 32 00 00  ....s... .....2..
00000010  00 e8 03 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
00000020  00 00 01 01 00 00 00 00  01 00 00 00 c0 a8 38 66  ........ ......8f
00000030  c0 a8 38 66 c0 a8 38 66  c0 a8 38 66 c0 a8 38 66  ..8f..8f ..8f..8f
00000040  ff ff 01 00 00 00 00 00  43 6c 75 73 74 65 72 3a  ........ Cluster:
00000050  00 02 00 00 00 f9 0d 00  00 e0 07 00 00 4c 69 6e  ........ .....Lin
00000060  75 78 20 33 2e 31 31 2e  30 2d 31 32 2d 67 65 6e  ux 3.11. 0-12-gen
00000070  65 72 69 63 00 47 34 2e  30 30 00                 eric.G4. 00.
```

```
struct register_msg {
    msg_hdr hdr;
    u8 conf[0x40];
    std::string description;
    u32 cpu_num;
    u32 cpu_spd;
    u32 mem_size;
    std::string os;
    std::string magic;
};
```

Dofloo's REGISTER

- It's also called Mr. Black
- It has text format of *"VERSONEX:%s|%d|%d|%s"*
  - *VERSONEX:Linux-3.11.0-12-generic|2|3576 MHz|2016MB|634MB|Hacker*
  - *VERSONEX:Windows XP|1|3582|Mr.Black*
  - *VERSONEX:Windows XP|1|3582 MHz|1024MB|245MB|Hacker*

Shannon Entropy

- *"Entropy is a measure of unpredictability of information content.* "
  - From *en.wikipedia.org*

$$\mathrm{H}(X) = \sum_i \mathrm{P}(x_i)\,\mathrm{I}(x_i) = -\sum_i \mathrm{P}(x_i) \log_b \mathrm{P}(x_i)$$

- Shannon entropy can be used to measure how statistically similar 2 messages are

Sample REGISTER Statistics

Family name	Length	Entropy
Kelihos	164	4.6~4.8
XOR DDoS	272	3.22~3.29
mayday	401	0.4~0.6
elknot	variable	2.5~2.8

Classification based on REGISTER

- Rich information included in REGISTER messages
  - length, entropy value, format, semantics fields
- A new classification that is based on the similarities among REGISTERs in statistics/structure
- It is scalable because the REGISTER message is easy to get

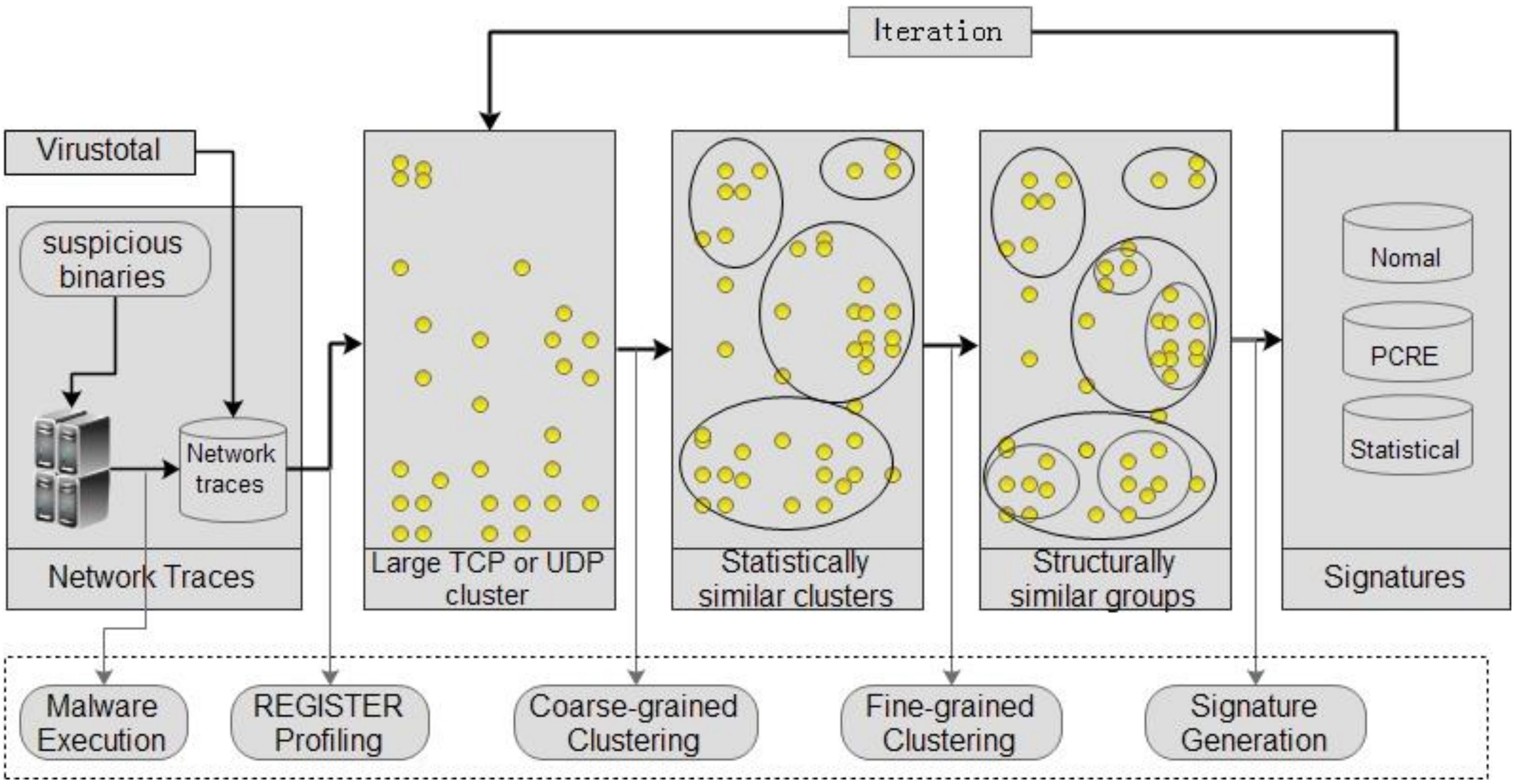
Objectives

- To classify unlabeled samples based on their REGISTER messages
  – Simplify the sample analysis work
- What we really need to do is to find out the number of REGISTER families, and generate signatures for later identification

What We not do

- Will not tell you which cluster of REGISTERs are malicious, and which are not
- Will not classify HTTP based REGISTERs
  - Good solution exists
  - there is so much classification info (e.g., method, uri, headers) that we think it’s better to classify them in a separate solution

The Architecture



REGISTER Profiling

- Creating REGISTERs from network traces
  - Mainly parsing PCAP files
- Setting REGISTER attributes for later clustering and signature generating
  - Length, entropy, binary/text format, semantic strings

Sample Profiles

360 INTERNET SECURITY CENTER


```
{
  "bin":1,
  "length": 260,
  "entropy": 0.703393,
  "strings": [
    {
      "offset":4,
      "size":64,
      "content":"Windows XP",
      "semantics": ”os”
    },
    {
      "offset":68,
      "size":128,
      "content":"1 * 3187MHz",
      "semantics": “cpu”
    },
    {
      "offset":196,
      "size":32,
      "content":"128MB",
      "semantics": “memory”
    }
  ]
}
```

```
{
  “bin": 1,
  "length": 127,
  "entropy": 2.949660 ,
  "strings": [
    {
      "offset": 55,
      "size": 18,
      "content": "08:00:27:6D:C8:C5",
      "semantics": "mac"
    },
    {
      "offset": 73,
      "size": 14,
      "content": "Ubuntu 13.10 ",
      "semantics": "os"
    },
    {
      "offset": 87,
      "size": 40,
      "content": "Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz",
      "semantics": "cpu"
    }
  ]
}
```

Coarse-grained Clustering

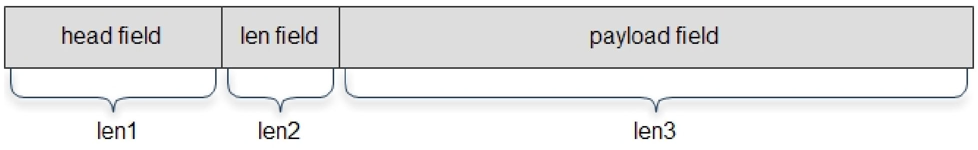
- To group statistically similar REGISTERs
  - k-means algorithm is used to cluster vectors of <length, entropy>
- To reduce the computation cost
  - A $O(N^2)$ computation cost is needed if we attempt to directly find out structurally similar REGISTERs

Finding Semantic Strings

- A heuristic deduction procedure
  - OS: “linux”, “Ubuntu 13.10”,”Win XP”
  - Memory: “2016MB”, “2016 MB”
  - CPU: “Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz”, “MHz: 3576,3576”, “3582 MHz”
- Every semantic string has following attributes:
  - semantics
  - offset
  - size

Length Field

- 3 types of semantics
  - len_value=len3
  - len_value=len2+len3
  - len_value=len1+len2+len3



- Field size
  - 32-bit/16-bit/8-bit
- Byte order
  - Host-byte-order or network-byte-order

Fine-grained Clustering

- To find out structurally similar REGISTERs
- 2 REGISTERs are considered as structurally similar if and only if:
  1. Having similar entropy values
  2. Sharing the same set of semantics strings and their placing order
  3. Sharing the same format of length field
  4. Sharing the same encoding format
     - binary or text
  5. Having similar length

Sample Signatures


```
{
    "name":"L172O0S11T1448427268.607769",
    "ordinal":[0.5, 1.0],
    "type":"normal",
    "length":172,
    "entropy":3.48832,
    “patterns":
    [
        {"type":"rawbytes", "offset":0, "length":11, "content":"31 36 38 00 6C 6C 7C 27 7C 27 7C"}
    ]
}

{
    "name":"L126O4S16T1448427256.926312",
    "ordinal":[0.5, 1.0],
    "type":"normal",
    "length":126,
    "entropy":2.74977,
    “patterns":
    [
        {"type":"rawbytes", "offset":4, "length":16, "content":"76 65 72 73 69 6F 6E 00 00 00 00 00 66 00 00 00"}
    ]
}
```

Signature Generation

- For each group of structurally similar REGISTERs a set of signatures are generated
- Generation steps includes:
  1. Finding out frequent items of (offset, byte_value)
  2. Merging offset continuing items
  3. Normalizing them into valid signatures
- Some policies:
  - AT_LEAST_OCCURS, default is 3
  - AT_LEAST_SIG_BYTES, default is 4
  - AT_LEAST_CONTINUOUS_SIG_BYTES, default is 1

Apriori/FP-Growth and Sig Bytes

- *"Apriori is an algorithm for frequent item set mining and association rule learning over transactional databases."*
  - From *en.wikipedia.org*
- Currently we use Apriori to find the frequent items of (offset, byte_value) among REGISTERs
- We will update our solution to FP-Growth for better performance

Signature Types

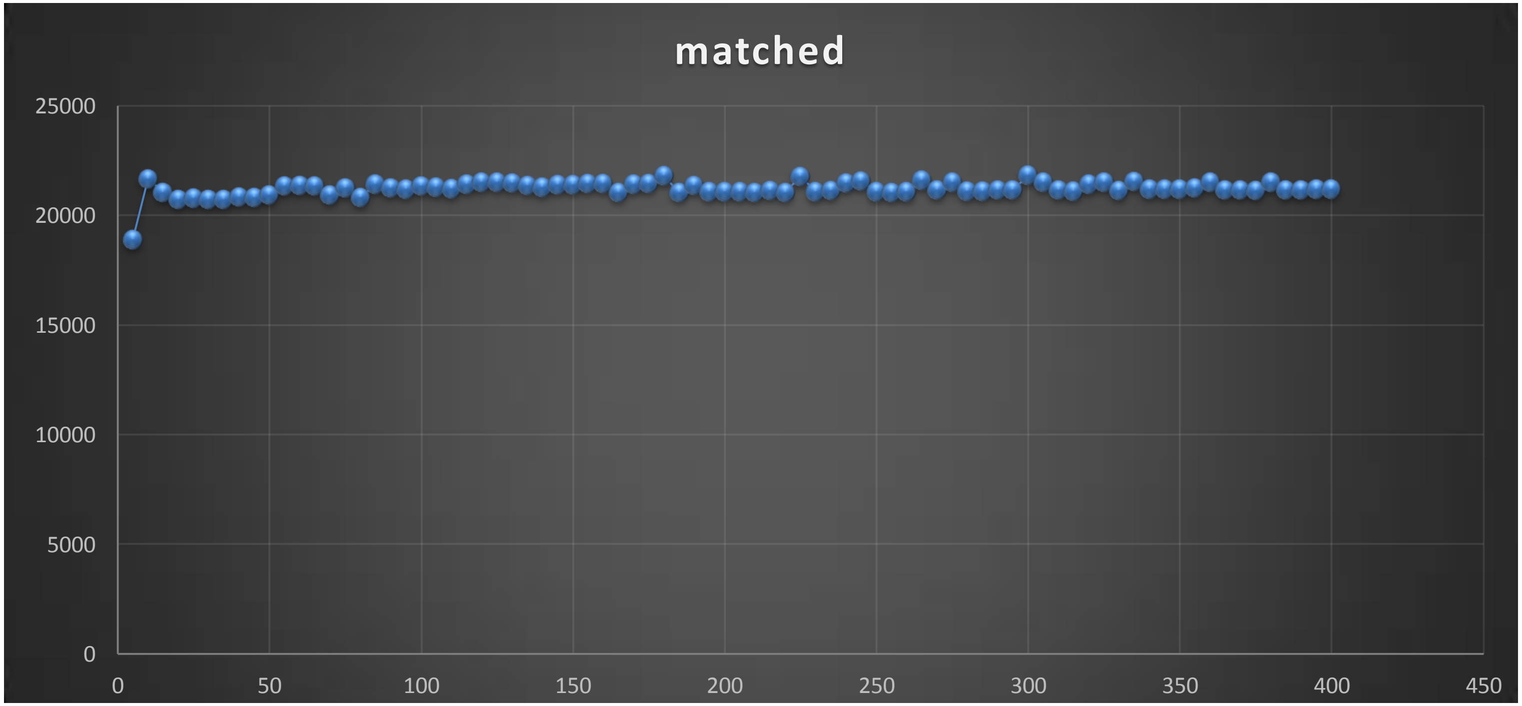
- Normal
  - specific byte patterns exist at specified offsets
- PCRE :
  - replacing semantic patterns with equivilant PCRE expressions, e.g., “Windows\s.*”,
- Entropy:
  - No valid patterns could be generated
  - AND all REGISTERs have the same length and very close entropy values

Evaluation

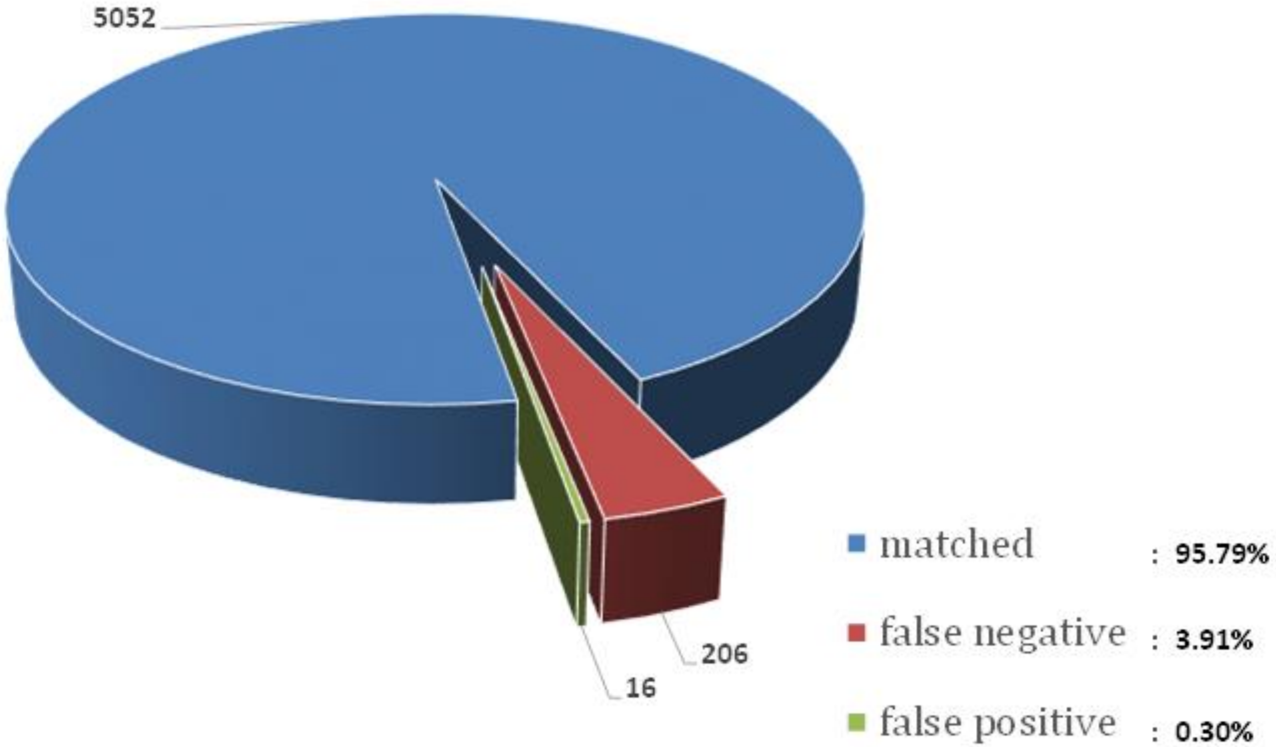
- Our system is implemented in C++ and python
  - About 2,500 lines of C++ code.
- It takes less than 30 minutes to classify 10K REGISTERs
  - Performed on a 4-core Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz machine with 4GB of RAM
  - Single thread

Choice of k

- k=20 is the best choice for k-means clustering when doing coarse-grained clustering



False Negatives/ False Positives



Generated Signature: STUN

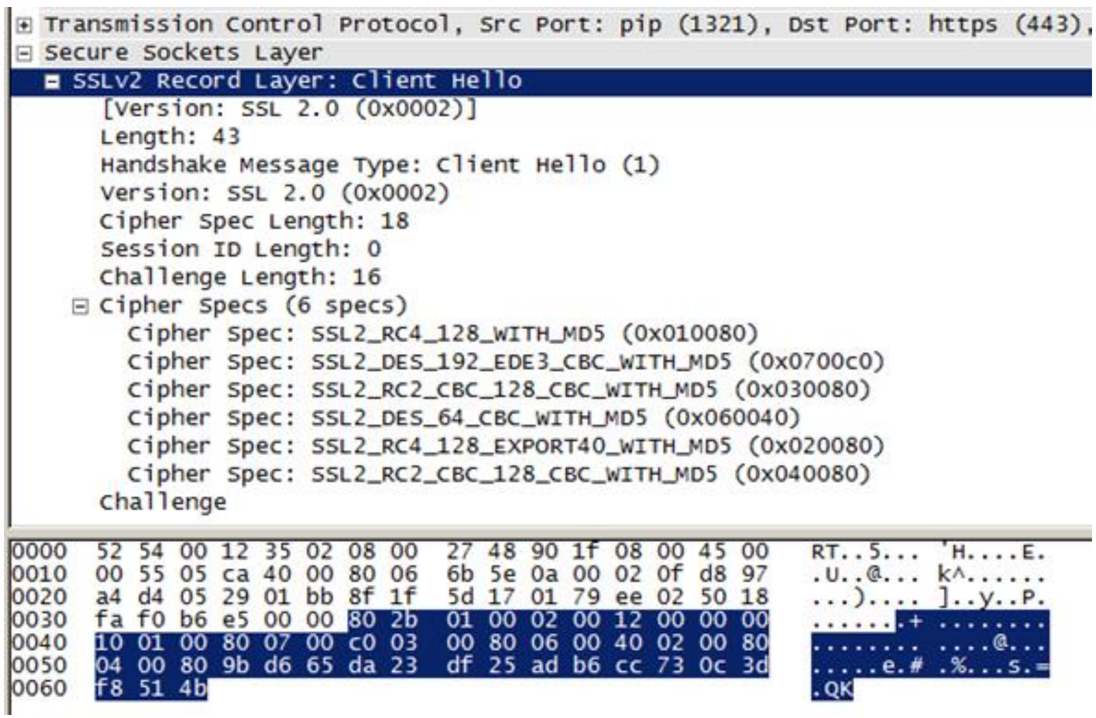

```
{
    "name":"L28O0S4O20S7T1447301150.241028",
    "ordinal":[0.5, 1.0],
    "sigtype":"normal",
    "length":28,
    "entropy":2.79622,
    "blocks":
    [
        {"type":"rawbytes", "offset":0, "length":4, "content":"00 01 00 08"},
        {"type":"rawbytes", "offset":20, "length":7, "content":"00 03 00 04 00 00 00"}
    ]
}
```



```
00000000  00 01 00 08 01 0f 24 53  ce 5c b3 66 a7 33 c9 65 ......$s .\.f.3.e
00000010  7f 6b 8f 4f 00 03 00 04  00 00 00 00             .k.O.... ....
0000001c  00 01 00 08 01 25 87 1d  b0 48 b9 65 4c 0a c4 70 .....%.. .H.eL..p
0000002c  77 5e 58 3a 00 03 00 04  00 00 00 00             w^X:.... ....
00000038  00 01 00 08 01 73 db 76  c6 7d f5 5b a6 6e aa 63 .....s.v .}.[.n.c
00000048  e7 5e aa 7b 00 03 00 04  00 00 00 00             .^.{.... ....
00000054  00 01 00 08 01 3e 1b 17  d5 20 4e 30 b0 76 a6 43 .....>.. . N0.v.C
00000064  bc 55 0c 61 00 03 00 04  00 00 00 00             .U.a.... ....
00000070  00 01 00 08 01 12 f9 64  aa 59 02 0e 8e 73 d8 1c .......d .Y...s..
00000080  27 60 98 16 00 03 00 04  00 00 00 00             '`...... ....
0000008c  00 01 00 08 01 01 5c 33  92 56 21 0e 06 11 d3 77 ......\3 .V!....w
0000009c  1e 6b 7c 48 00 03 00 04  00 00 00 00             .k|H.... ....
    00000000  01 01 00 44 01 01 5c 33  92 56 21 0e 06 11 d3 77 ...D..\3 .V!....w
    00000010  1e 6b 7c 48 00 01 00 08  00 01 e7 cc 5e 17 f0 1b .k|H.... ....^...
    00000020  00 04 00 08 00 01 0d 96  af 06 00 7c 00 05 00 08 ........ ...|....
    00000030  00 01 0d 97 af 06 00 7d  80 20 00 08 00 01 e6 cd .......} . ......
    00000040  5f 16 ac 28 80 22 00 10  56 6f 76 69 64 61 2e 6f _..(."..  Vovida.o
    00000050  72 67 20 30 2e 39 36 00                          rg 0.96.
```

Generated Signature: SSL

{
“name”:”L45O0S29T1447301147.172219”,
“ordinal”:[0.5, 1.0],
“type”:”normal”,
“length”:45,
“entropy”:2.93596,
“blocks”:
[
{“type”:”rawbytes”, “offset”:0, “length”:29, “content”:”80 2B 01 00 02 00 12 00 00 00 10 01 00 80 07 00 C0 03 00 80 06 00 40 02 00 80 04 00 80”}
]
}



Generated Signature: Bladabindi


```
{
    "name":"L158O0S7O31S1O43S4O51S1O66S1O72S1O81S1O85S1O103S1O134S24T1447301149.680667",
    "ordinal":[0.5, 1.0],
    "type":"normal",
    "length":158,
    "entropy":3.3299,
    "blocks":
    [
        {"type":"rawbytes", "offset":0, "length":7, "content":"6C 76 7C 27 7C 27 7C"},
        {"type":"rawbytes", "offset":31, "length":1, "content":"7C"},
        {"type":"rawbytes", "offset":43, "length":4, "content":"42 4F 4F 4D"},
        {"type":"rawbytes", "offset":51, "length":1, "content":"7C"},
        {"type":"rawbytes", "offset":66, "length":1, "content":"7C"},
        {"type":"rawbytes", "offset":72, "length":1, "content":"30"},
        {"type":"rawbytes", "offset":81, "length":1, "content":"7C"},
        {"type":"rawbytes", "offset":85, "length":1, "content":"7C"},
        {"type":"rawbytes", "offset":103, "length":1, "content":"73"},
        {"type":"rawbytes", "offset":134, "length":24, "content":"7C 27 7C 27 7C 2E 2E 7C 27 7C 27 7C 7C 27 7C 27 7C 5B 65 6E 64 6F 66 5D"}
    ]
}
```



```
00000000  6c 76 7c 27 7c 27 7c 53  47 46 6a 53 32 56 6b 58 lv|'|'|S GFjS2VkX
00000010  7a 5a 44 4e 7a 68 42 4f  55 4d 7a 7c 27 7c 27 7c zZDNzhBO UMz|'|'|
00000020  54 45 51 55 49 4c 41 42  4f 4f 4d 42 4f 4f 4d 7c TEQUILAB OOMBOOM|
00000030  27 7c 27 7c 6a 61 6e 65  74 74 65 64 6f 65 7c 27 '|'|jane ttedoe|'
00000040  7c 27 7c 32 30 31 35 2d  30 39 2d 31 31 7c 27 7c |'|2015- 09-11|'|
00000050  27 7c 55 53 41 7c 27 7c  27 7c 57 69 6e 20 58 50 '|USA|'| '|Win XP
00000060  20 50 72 6f 66 65 73 73  69 6f 6e 61 6c 53 50 33  Profess ionalSP3
00000070  20 78 38 36 7c 27 7c 27  7c 4e 6f 7c 27 7c 27 7c  x86|'|' |No|'|'|
00000080  30 2e 35 2e 30 45 7c 27  7c 27 7c 2e 2e 7c 27 7c 0.5.0E|' |'|..|'|
00000090  27 7c 7c 27 7c 27 7c 5b  65 6e 64 6f 66 5d       '||'|'|[ endof]
0000009E  75 73 5b 65 6e 64 6f 66  5d                      us[endof ]
```

Generated Signature: Nitol

360 INTERNET SECURITY CENTER


```
{
    "name":"LXO1S18T1448629222.519142",
    "ordinal":[0.5, 1.0],
    "type":"normal",
    "entropy":1.2787,
    "blocks":
    [
        {"type":"rawbytes", "offset":1, "length":18, "content":"00 00 00 77 00 00 00 09 04 00 00 57 69 6E 20 58 50 20"}
    ]
}
```


```
00000000  b0 00 00 00 77 00 00 00  09 04 00 00 57 69 6e 20  |....w.......Win |
00000010  58 50 20 53 50 33 00 00  00 00 00 00 00 00 00 00  |XP SP3..........|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000040  00 00 00 00 00 00 00 00  00 00 00 00 31 32 38 20  |............128 |
00000050  4d 42 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |MB..............|
00000060  00 00 00 00 00 00 00 00  00 00 00 00 31 2a 33 33  |............1*33|
00000070  38 36 4d 48 7a 00 00 00  00 00 00 00 00 00 00 00  |86MHz...........|
00000080  00 00 00 00 00 00 00 00  00 00 00 00 31 30 30 20  |............100 |
00000090  4d 62 70 73 00 00 00 00  00 00 00 00 00 00 00 00  |Mbps............|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 00 00  |................|
000000b0  04 00 00 00 8c 25 01 00                           |.....%..|
000000b8
```

Pitfalls

- REGISTER is not always used in C&C protocols
- For UDP based C&C protocol, it’s hard to tell which message is REGISTER because of its statelessness nature
- The same REGISTER may be shared across different C&C protocols
- Our solution is not good at classifying variable-length text format REGISTERs

Conclusions

- Statistical/structural similarities can be used to effectively classify REGISTERs
- REGISTER based classification can complement C&C protocol based classification
- Our solution is good at classifying binary format REGISTERs with fixed lengths

Q&A