YARA: An Introduction

26th annual **FIRST** conference

BOSTON

MASSACHUSETTS

JUNE 22—27, 2014

BACK TO THE 'ROOT' OF INCIDENT RESPONSE

Andreas Schuster

June 25, 2014
Boston

Important Remarks - Read this first!

- This hands-on tutorial will cover advanced topics. If you still have to write your first YARA rule, this tutorial will not be helpful at all.
- This slide deck is split in two parts:
  - The first part covers some basic concepts. **You should already have written some YARA rules on your own and applied some of these techniques a number of times before coming to class.** However, the virtual machine image (see below) includes the materials for the basic exercises, too, so you can work on them at your own pace.
  - The second part, starting from the „Advanced Topics“ tile slide, will be covered in our tutorial.
- Please download the VMware image from http://r.forens.is/bos1st/. **Ensure your environment works properly before coming to class.**

Logistics
Agenda

- Morning session
  - Writing YARA rules
  - Building rules based on magic numbers
  - Memory analysis with Volatility and YARA

Introduction

Introduction

What is YARA?

- „The pattern matching swiss knife for malware researchers (and everyone else)“
- Hosted on GitGub
  http://plusvic.github.io/yara/
- **Pattern matching**:
  - ➔ strings (ASCII, UCS-2)
  - ➔ regular expressions
  - ➔ binary patterns (hex strings)
- **Classification**:
  - ➔ on input: combination of strings
  - ➔ on output: tags, metadata

Introduction
What is YARA?

```
rule my_example : tag1 tag2 tag3
{
    meta:
        description = "This is just an example"
        thread_level = 3
        in_the_wild = true

    strings:
        $a = { 6A 40 68 00 30 00 00 6A 14 8D 91 }
        $b = /[0-9a-f]{32}/
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

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

Introduction

What YARA is NOT

- Not a virus scanner
- Not a correlation engine
- Not a bayesian classifier
- No artifical intelligence (AI) involved

Introduction

How can YARA help me?

- A „better grep“
- Use cases:
  - Finding interesting entries on pastebin.com ...
  - Triage data
  - Preprocess files to direct reverse engineering efforts
- Integrate it into your projects:
  - C library
  - Python bindings
https://github.com/plusvic/yara/tree/master/yara-python
  - Ruby bindings
https://github.com/SpiderLabs/yara-ruby

Introduction

How can YARA help me?

- YARA rules are supported by security products and services
  - ➔ FireEye appliances
  - ➔ Fidelis XPS
  - ➔ RSA ECAT
  - ➔ Volatility
  - ➔ ThreadConnect threat intelligence exchange
  - ➔ VirusTotal Intelligence
  - ➔ ...

Writing YARA Rules

Hello World!
Your First YARA Rule

Your first YARA rule

Starting the VM

- Start VM
- Log in as user „training“, password is „training“
- „training“ also is your sudo password
- You may want to customize the keyboard layout:
  - ➔ System > Preferences > Keyboard
  - ➔ Select „Layouts“ tab
- Open a terminal window

Your first YARA rule

Getting help

```
$ yara
usage:  yara [OPTION]... [RULEFILE]... FILE
options:
  -t <tag>                    print rules tagged as <tag> and ignore the
                              rest. Can be used more than once.
  -i <identifier>             print rules named <identifier> and ignore the
                              rest. Can be used more than once.
  -n                          print only not satisfied rules (negate).
  -g                          print tags.
  -m                          print metadata.
  -s                          print matching strings.
  -d <identifier>=<value>     define external variable.
  -r                          recursively search directories.
  -f                          fast matching mode.
  -v                          show version information.
```

Your first YARA rule

Check the installed version

- There are slight differences between YARA versions 1.4 to 1.7 and 2.0, see http://code.google.com/p/yara-project/source/browse/trunk/ChangeLog and https://github.com/plusvic/yara/commits/master for details
- User manual is in /yara/doc of this VM
- What version does the VM provide?

```
$ yara -v
```

- You should see the result:

```
yara 1.6
```

Your first YARA rule

Editors

- The following editors are available:
  - vim (with simple syntax highlighting)
  - gvim (with GUI and syntax highlighting)
  - emacs
  - gedit

Your first YARA rule

A minimalist rule

- cd /yara/Lab_1
- Create a file named „hello.yara“ with the following contents:

```
rule Hello_World
{
        condition:
                true
}
```

- Now let the computer greet you:

```
$ yara hello.yara /yara/malware/somefile.txt
```

Your first YARA rule

Passing external data to YARA

- Review the file greeting.yara

```
rule GoodMorning
{
        condition:
                hour < 12 and hour >= 4
}
```

- Now pass different values for „hour“ to the rule set:

```
$ yara -d hour=8 greeting.yara /yara/malware/somefile.txt
GoodMorning /yara/files/somefile.txt

$ yara -d hour=20 greeting.yara /yara/malware/somefile.txt
GoodEvening /yara/files/somefile.txt
```

- What happens when you pass a string (e.g. „noon“) or no value at all?

Identify Executable Files

Identify executable files

A simple specification for PE files

- Task: To find any files in Portable Executable („PE“) format
- Simple specification: File must contain the strings „MZ“ and „PE“

```
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 c8 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  65 cd 43 c7 21 ac 2d 94  21 ac 2d 94 21 ac 2d 94  |e.C.!.-.!.-.!.-.|
00000090  21 ac 2c 94 25 ac 2d 94  e2 a3 70 94 24 ac 2d 94  |!.,.%.-...p.$.-.|
000000a0  c9 b3 26 94 23 ac 2d 94  52 69 63 68 21 ac 2d 94  |..&.#.-.Rich!.-.|
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000c0  00 00 00 00 00 00 00 00  50 45 00 00 4c 01 03 00  |........PE..L...|
```

Identify executable files

Rule skeleton

- cd /yara/Lab_2
- Create a new file, named „executable.yara“
- Start with a blank rule:

```
rule PE_file
{
}
```

Identify executable files

Adding strings

- Now add the two strings:

```
rule PE_file
{
        strings:
                $mz = "MZ"
                $pe = "PE"
}
```

- Note: Strings are case-sensitive by default!

Identify executable files

Adding the condition

- A portable executable file MUST contain both strings. So, add the proper condition:

```
rule PE_file
{
        strings:
                $mz = "MZ"
                $pe = "PE"
        condition:
                $mz and $pe
}
```

- Test your rule file:

```
$ yara -r executable.yara /yara/malware
```

Identify executable files

Refining the condition

- More constraints:
  - „MZ“ at offset 0
  - UInt32 at offset 0x3c points to „PE“
- Refine your condition section:

```
        condition:
                (\$mz at 0) and
                (\$pe at (uint32(0x3c)))
```

- Test your rule file again:

```
$ yara -r executable.yara /yara/malware
```

Identify executable files

The final rule

- This is how your rule should look like:

```
rule PE_file
{
        strings:
                $mz = "MZ"
                $pe = "PE"

        condition:
                ($mz at 0) and
                ($pe at (uint32(0x3c)))
}
```

Obfuscation: Move Single Byte

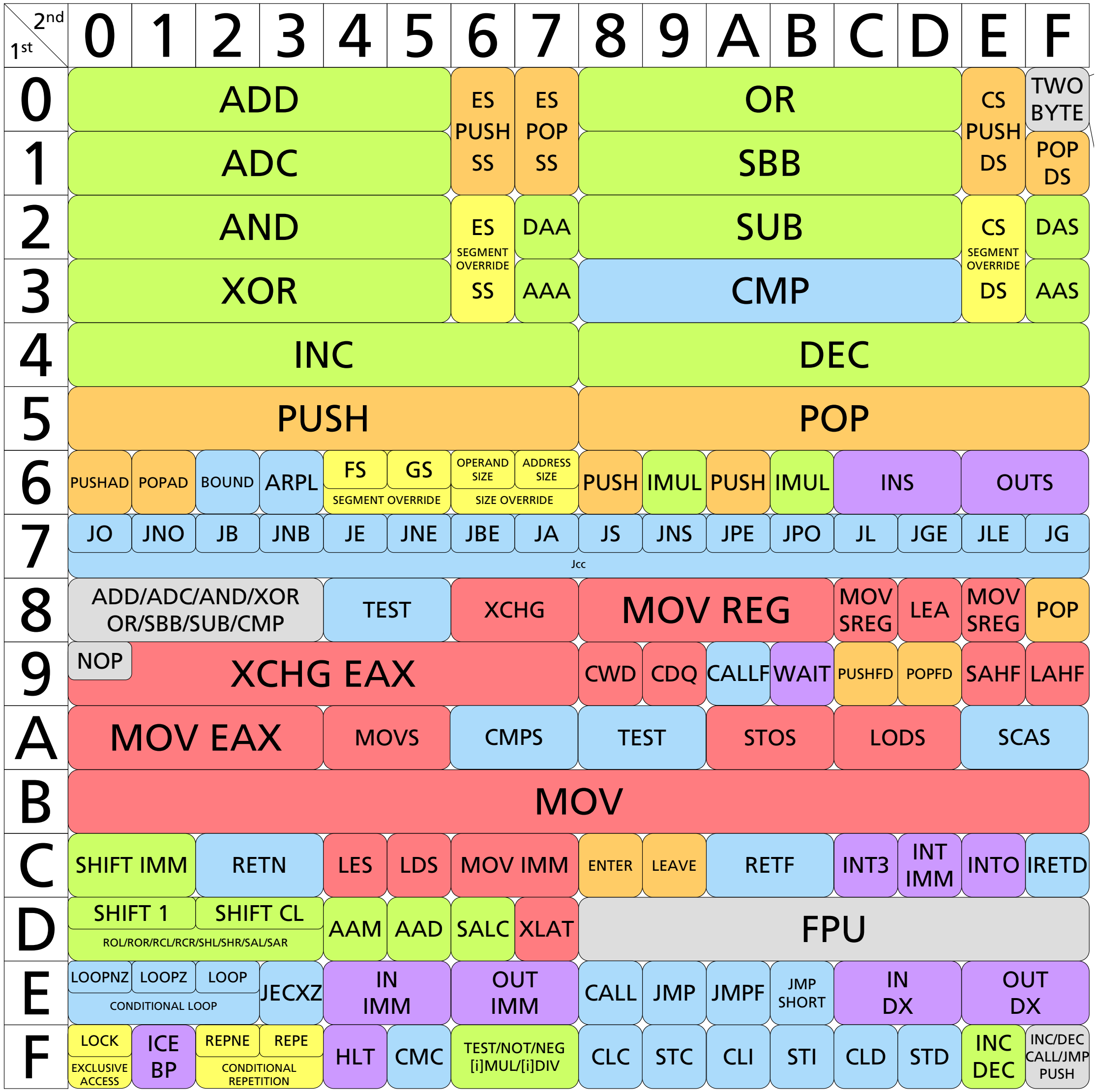
Obfuscation: Move Single Byte

- Can you spot the registry key name?

```
00415393  C6 45 CC 53 C6 45 CD 6F C6 45 CE 66 C6 45 CF 74  .E.S.E.o.E.f.E.t
004153A3  C6 45 D0 77 C6 45 D1 61 C6 45 D2 72 C6 45 D3 65  .E.w.E.a.E.r.E.e
004153B3  C6 45 D4 5C C6 45 D5 4D C6 45 D6 69 C6 45 D7 63  .E.\.E.M.E.i.E.c
004153C3  C6 45 D8 72 C6 45 D9 6F C6 45 DA 73 C6 45 DB 6F  .E.r.E.o.E.s.E.o
004153D3  C6 45 DC 66 C6 45 DD 74 C6 45 DE 5C C6 45 DF 57  .E.f.E.t.E.\.E.W
004153E3  C6 45 E0 69 C6 45 E1 6E C6 45 E2 64 C6 45 E3 6F  .E.i.E.n.E.d.E.o
004153F3  C6 45 E4 77 C6 45 E5 73 C6 45 E6 5C C6 45 E7 43  .E.w.E.s.E.\.E.C
00415403  C6 45 E8 75 C6 45 E9 72 C6 45 EA 72 C6 45 EB 65  .E.u.E.r.E.r.E.e
00415413  C6 45 EC 6E C6 45 ED 74 C6 45 EE 56 C6 45 EF 65  .E.n.E.t.E.V.E.e
00415423  C6 45 F0 72 C6 45 F1 73 C6 45 F2 69 C6 45 F3 6F  .E.r.E.s.E.i.E.o
00415433  C6 45 F4 6E C6 45 F5 5C C6 45 F6 52 C6 45 F7 75  .E.n.E.\.E.R.E.u
00415443  C6 45 F8 6E                                      .E.n
```

Obfuscation: Move Single Byte

Find the opcode for 0xc6



2nd 1st	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD						ES PUSH SS	ES POP SS	OR						CS PUSH DS	TWO BYTE
1	ADC								SBB							POP DS
2	AND						ES SEGMENT OVERRIDE SS	DAA	SUB						CS SEGMENT OVERRIDE DS	DAS
3	XOR							AAA	CMP							AAS
4	INC								DEC							
5	PUSH								POP							
6	PUSHAD	POPAD	BOUND	ARPL	FS SEGMENT OVERRIDE	GS	OPERAND SIZE SIZE OVERRIDE	ADDRESS SIZE	PUSH	IMUL	PUSH	IMUL	INS		OUTS	
7	JO Jcc	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
8	ADD/ADC/AND/XOR OR/SBB/SUB/CMP				TEST		XCHG		MOV REG				MOV SREG	LEA	MOV SREG	POP
9	NOP	XCHG EAX							CWD	CDQ	CALLF	WAIT	PUSHFD	POPFD	SAHF	LAHF
A	MOV EAX				MOVS		CMPS		TEST		STOS		LODS		SCAS	
B	MOV															
C	SHIFT IMM		RETN		LES	LDS	MOV IMM		ENTER	LEAVE	RETF		INT3	INT IMM	INTO	IRETD
D	SHIFT 1 ROL/ROR/RCL/RCR/SHL/SHR/SAL/SAR		SHIFT CL		AAM	AAD	SALC	XLAT	FPU							
E	LOOPNZ CONDITIONAL LOOP	LOOPZ	LOOP	JECXZ	IN IMM		OUT IMM		CALL	JMP	JMPF	JMP SHORT	IN DX		OUT DX	
F	LOCK EXCLUSIVE ACCESS	ICE BP	REPNE CONDITIONAL REPETITION	REPE	HLT	CMC	TEST/NOT/NEG [i]MUL/[i]DIV		CLC	STC	CLI	STI	CLD	STD	INC DEC	INC/DEC CALL/JMP PUSH

Source:
Extract from „x86 Opcode Structure and Instruction Overview“
by Daniel Plohmann, Fraunhofer FKIE

Obfuscation: Move Single Byte

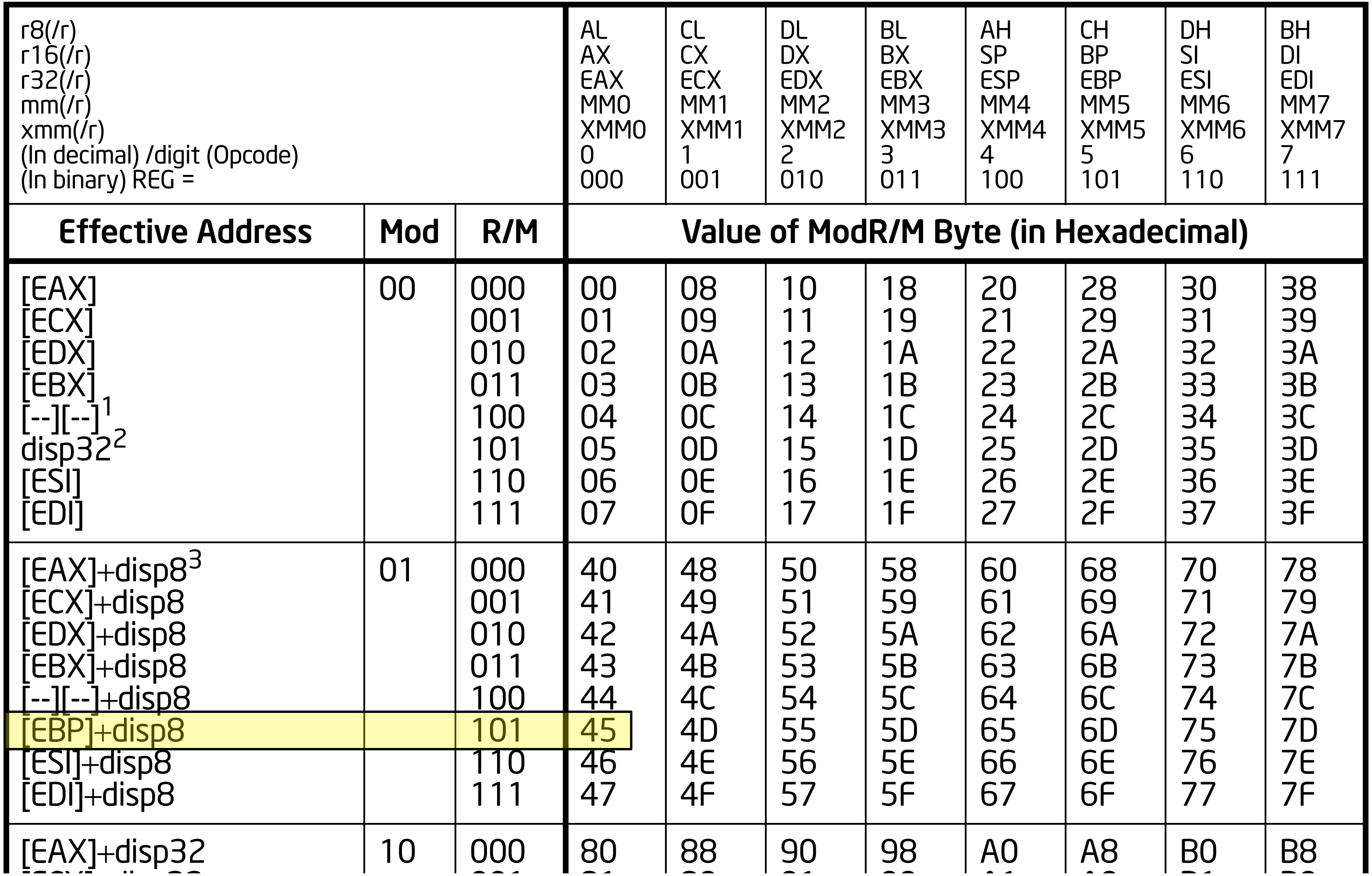
Read the manual page for MOV

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + A3	MOV moffs64*,RAX	D	Valid	N.E.	Move RAX to (*offset*).
B0+ *rb*	MOV *r8, imm8*	E	Valid	Valid	Move *imm8* to *r8.*
REX + B0+ *rb*	MOV r8***, imm8	E	Valid	N.E.	Move *imm8* to *r8.*
B8+ *rw*	MOV *r16, imm16*	E	Valid	Valid	Move *imm16* to *r16.*
B8+ *rd*	MOV *r32, imm32*	E	Valid	Valid	Move *imm32* to *r32.*
REX.W + B8+ *rd*	MOV *r64, imm64*	E	Valid	N.E.	Move *imm64* to *r64.*
C6 */0*	MOV *r/m8, imm8*	F	Valid	Valid	Move *imm8* to *r/m8.*
REX + C6 */0*	MOV r/m8***, imm8	F	Valid	N.E.	Move *imm8* to *r/m8.*
C7 */0*	MOV *r/m16, imm16*	F	Valid	Valid	Move *imm16* to *r/m16.*
C7 */0*	MOV *r/m32, imm32*	F	Valid	Valid	Move *imm32* to *r/m32.*
REX.W + C7 */0*	MOV *r/m64, imm32*	F	Valid	N.E.	Move *imm32 sign extended to 64-bits* to *r/m64.*

Obfuscation: Move Single Byte

Find the register and addressing mode for 0x45

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte



r8(/r)
r16(/r)
r32(/r)
mm(/r)
xmm(/r)
(In decimal) /digit (Opcode)
(In binary) REG =
			AL
			AX
			EAX
			MM0
			XMM0
			0
			000
				CL
				CX
				ECX
				MM1
				XMM1
				1
				001
					DL
					DX
					EDX
					MM2
					XMM2
					2
					010
						BL
						BX
						EBX
						MM3
						XMM3
						3
						011
							AH
							SP
							ESP
							MM4
							XMM4
							4
							100
								CH
								BP
								EBP
								MM5
								XMM5
								5
								101
									DH
									SI
									ESI
									MM6
									XMM6
									6
									110
										BH
										DI
										EDI
										MM7
										XMM7
										7
										111

Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8

Obfuscation: Move Single Byte

Reveal the string

- Single byte MOVes are a common technique to obfuscate strings.

```
0000:00415393      mov     [ebp+SubKey],     'S'   ; C6 45 CC 53
0000:00415397      mov     [ebp+SubKey+1],   'o'   ; C6 45 CD 6F
0000:0041539B      mov     [ebp+SubKey+2],   'f'   ; C6 45 CE 66
0000:0041539F      mov     [ebp+SubKey+3],   't'   ; C6 45 CF 74
0000:004153A3      mov     [ebp+SubKey+4],   'w'   ; C6 45 D0 77
0000:004153A7      mov     [ebp+SubKey+5],   'a'   ; C6 45 D1 61
0000:004153AB      mov     [ebp+SubKey+6],   'r'   ; C6 45 D2 72
0000:004153AF      mov     [ebp+SubKey+7],   'e'   ; C6 45 D3 65
0000:004153B3      mov     [ebp+SubKey+8],   '\'   ; C6 45 D4 5C
0000:004153B7      mov     [ebp+SubKey+9],   'M'   ; C6 45 D5 4D
0000:004153BB      mov     [ebp+SubKey+0Ah], 'i'   ; C6 45 D6 69
0000:004153BF      mov     [ebp+SubKey+0Bh], 'c'   ; C6 45 D7 63
0000:004153C3      mov     [ebp+SubKey+0Ch], 'r'   ; C6 45 D8 72
0000:004153C7      mov     [ebp+SubKey+0Dh], 'o'   ; C6 45 D9 6F
0000:004153CB      mov     [ebp+SubKey+0Eh], 's'   ; C6 45 DA 73
0000:004153CF      mov     [ebp+SubKey+0Fh], 'o'   ; C6 45 DB 6F
0000:004153D3      mov     [ebp+SubKey+10h], 'f'   ; C6 45 DC 66
0000:004153D7      mov     [ebp+SubKey+11h], 't'   ; C6 45 DD 74
```

Obfuscation: Move Single Byte

Develop a signature

- Signature:
  - ➔ 0xC6 0x45 is a constant (opcode and r/m8)
  - ➔ disp8 (index) is variable, but restricted to a single byte
  - ➔ the character (imm8) is variable, but also restricted to a single byte
- Pattern: C6 45 ?? ?? C6 45 ?? ?? C6 45 ...

Obfuscation: Move Single Byte
Create and test your signature

- cd /yara/Lab_2
- Create a file named „obfuscation.yara“ and a signature „single_byte_mov“
- Add the pattern as a string. Note: hex strings are enclosed in curly braces.
- Add the proper condition.
- Test your signature:

```
$ yara -r obfuscation.yara /yara/malware
```

- How many files contain at least one obfuscated string?

Obfuscation: Move Single Byte
Create and test your signature

- This is how your rule file should look like:

```
rule single_byte_mov
{
        strings:
                $a = { c6 45 ?? ?? c6 45 ?? ?? c6 45 }

        condition:
                $a
}
```

Obfuscation: Move Single Byte

Improve your signature

- Pattern using wildcards:
  C6 45 ?? ?? C6 45 ?? ?? C6 45

- Pattern using jumps:
  C6 45 [2] C6 45 [2] C6 45

- Jumps are:

  - → easier to read and maintain

  - → flexible, variable in length: [n-m]

Obfuscation: Move Single Byte

Improve your signature

- Modify your signature to use jumps instead of multi-byte wildcards
- Test your signature again. Are there any differences?
- Can you tell the number of obfuscated strings (not files!) from the output?
- Bonus question:
  - If you know PCRE well, then rewrite the pattern to match the *whole* obfuscated string. (see /yara/doc/yara/pcre.txt for a syntax refresher)
  - Issue `yara -s -r obfuscation.yara /yara/malware`
  - How many obfuscated strings are there?

Obfuscation: Move Single Byte

Improve your signature

- Again, this is how your rule should look like:

```
rule single_byte_mov
{
        strings:
                $a = { c6 45 [2] c6 45 [2] c6 45 }

        condition:
                $a
}
```

Obfuscation: Move Single Byte
Bonus question

- And here is the answer to the bonus question:

```
rule single_byte_mov3
{
        strings:
                $a = /(\xc6\x45..){3,}/

        condition:
                $a
}
```

- Count of matching strings:

```
$ yara -s -r obfuscation.yara /yara/malware/antivirus.exe | wc -l
4
```

- The first line is the matching rule (and file name), so the answer is: „3 strings were obfuscated“

Magic Numbers

Magic numbers

General approach

- Look for constants that are important for an algorithm
- The longer, the better (reduces false positives!)
- Examples:
  - ➔ static substitution box (s-box) of DES
  - ➔ MD5 init and transform constants
  - ➔ polynomial for Cyclic Redundancy Check
- Be aware of endianess issues
  0x1234 can be stored as 0x12 0x34 or 0x34 0x12
- Consider breaking up long numbers, loading into different registers, optimizations by compiler

Magic numbers
A Random Number Generator

- Linear Congruential Generator (LCG)
  - $x_n+1 = (ax_n + c) \bmod m$
  - Pierre L'Ecuyer: Tables of linear congruential generators of different sizes and good lattice structure (1999) http://dimsboiv.uqac.ca/Cours/C2012/8INF802_Hiv12/ref/paper/RNG/TableLecuyer.pdf
  - William H. Press: „Numerical recipes: the art of scientific computing“ (3rd ed., 2007), Chapter 7

Magic numbers

The RNG of PoisonIvy

```
0000:00000DA5 rand_init:
0000:00000DA5                 lea     esi, [ebp+base]
                              ; seed with CPU tick counter
0000:00000DAB                 rdtsc
0000:00000DAD                 xchg    eax, edx
0000:00000DAE                 xor     ecx, ecx
0000:00000DB0
0000:00000DB0 rand_loop:
                              ; LCG x := (x * 2891336453 + 1) mod 2^32
0000:00000DB0                 imul    eax, 2891336453
0000:00000DB6                 add     eax, 1
0000:00000DB9                 mov     [esi+ecx*4+8D9h], eax
0000:00000DC0                 add     ecx, 1
0000:00000DC3                 cmp     ecx, 34
0000:00000DC6                 jb      short rand_loop
```

Magic numbers

Camellia‘s magic constants

- cd /yara/Lab_3
- There you‘ll find a copy of RFC 3713, which specifies the Camellia encryption algorithm.
- Review the specification and try to find good magic numbers. Do NOT even try to understand the algorithm!
- You are explicitly allowed (and encouraged) to collaborate with your neighbours!
- State the magic number (or page number, variable name, etc.)

Magic numbers

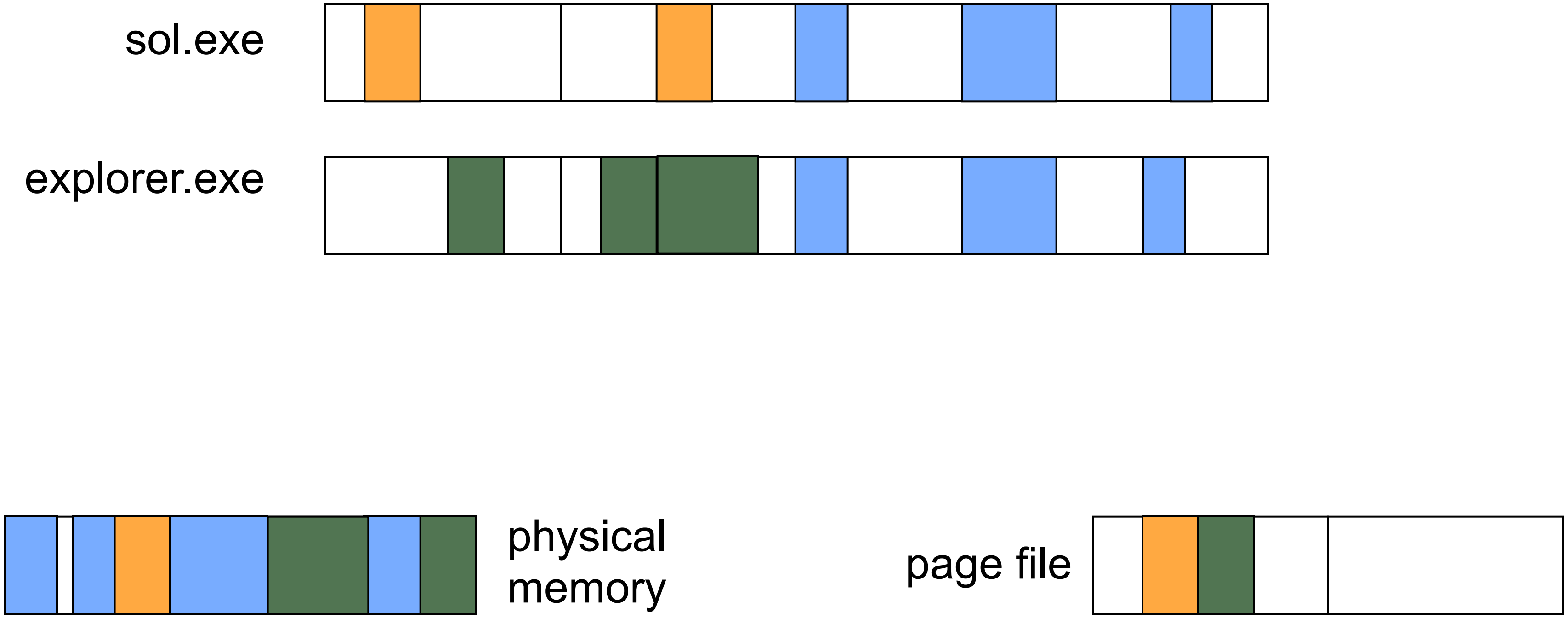
Camellia‘s magic constants

- Write one or multiple rules to check for your magic number(s)
- Test your rule(s) on /yara/malware, as before
- What file(s) are likely to contain the Camellia algorithm?

Memory Analysis

Memory analysis

Virtual and physical memory



Memory analysis
Scanning physical memory

- advantages:
  - fast
  - best coverage (you may want to scan the pagefile, too)

- disadvantages:
  - memory fragmentation can break your signatures
  - search hits can‘t be attributed to a process

Memory analysis
Scanning virtual memory

- advantages:
  - ➔ attribution is easy
  - ➔ defragmented memory image
- disadvantages:
  - ➔ slow
  - ➔ does not cover unallocated („free“) memory

Memory analysis
Proposed method

1. Obtain physical memory dump and pagefile

   ➔ suspend VM and copy .vmem file
   or use a memory dumper, like win32dd

   ➔ mount .vmdk using your tool of choice or
   extract pagefile from live system using FTK Imager

2. Use Volatility to scan each virtual address space or the kernel address space

3. Use YARA to scan pagefile and memory dump in order to cover unallocated and paged memory areas.

Memory analysis
Using Volatility

- Specialized „yarascan“ command
- Option -Y builds YARA rule on the fly,
  accepts either string, hex or regular expression

  ```
  $ vol.py -f memory.img yarascan -Y "rm6.org"
  $ vol.py -f memory.img yarascan -Y "rm6.org" -W
  $ vol.py -f memory.img yarascan -Y "/[0-9a-fA-F]{32}/"
  $ vol.py -f memory.img yarascan -Y "{ c6 45 [2] c6 45 [2] c6 45 }"
  ```

- Option -y reads YARA rules from a file
- Option -K searches the kernel address space instead of process address spaces
- Option -p searches only the address space of process identified by its PID
- Option -D dumps responsive memory areas to disk

Memory analysis

Hands on: Step 1

- cd /yara/Lab_4
- Data to analyze:
  - memory.dmp is a physical memory dump obtained from Windows XP SP2
  - pagefile.sys was copied off the „physical disk“ using FTK Imager
- Rule sets:
  - dyndns.yara: names of well-known Dynamic DNS zones
  - camellia.yara: magic numbers of Camellia encryption algorithm

Memory analysis

Hands on: Step 2

- Search all process address spaces for artifacts of the Camellia encryption algorithm.
- Take a note of the responsive PIDs
- Bonus: Can you find any traces of Camellia in kernel memory?

Memory analysis

Hands on: Step 2

```
training@ubuntu:/yara/Lab_4$ vol.py -f memory.dmp yarascan -y camellia.yara
Volatile Systems Volatility Framework 2.2
Rule: Camellia_Sigma
Owner: Process svchost.exe Pid 1080
0x5d10c764  a0 9e 66 7f 3b cc 90 8b b6 7a e8 58 4c aa 73 b2   ..f.;....z.XL.s.
0x5d10c774  c6 ef 37 2f e9 4f 82 be 54 ff 53 a5 f1 d3 6f 1c   ..7/.O..T.S...o.
0x5d10c784  10 e5 27 fa de 68 2d 1d b0 56 88 c2 b3 e6 c1 fd   ..'..h-..V......
0x5d10c794  5d 83 c7 08 8b 44 24 30 8b 98 11 01 00 00 ff 90   ]....D$0........
...
Rule: Camellia_tables
Owner: Process svchost.exe Pid 1116
0x2010cc87  10 10 20 20 10 10 30 30 00 00 20 20 00 00 10 10   ......00........
0x2010cc97  30 30 00 00 20 20 10 10 20 20 00 00 30 30 55 8b   00..........00U.
0x2010cca7  ec 56 51 8b 75 08 8b 9e d1 08 00 00 8b 8e d5 08   .VQ.u...........
0x2010ccb7  00 00 8b 94 33 d9 08 00 00 8b 84 33 dd 08 00 00   ....3......3....
Rule: Camellia_Sigma
Owner: Process explorer.exe Pid 1400
0x01380764  a0 9e 66 7f 3b cc 90 8b b6 7a e8 58 4c aa 73 b2   ..f.;....z.XL.s.
0x01380774  c6 ef 37 2f e9 4f 82 be 54 ff 53 a5 f1 d3 6f 1c   ..7/.O..T.S...o.
0x01380784  10 e5 27 fa de 68 2d 1d b0 56 88 c2 b3 e6 c1 fd   ..'..h-..V......
0x01380794  5d 83 c7 08 8b 44 24 30 8b 98 11 01 00 00 ff 90   ]....D$0........
...
```

Memory analysis
Hands on: Step 2

- Infected processes:
  - svchost.exe PID 1080
  - svchost.exe PID 1116
  - VMwareService.exe PID 1652
  - explorer.exe PID 1400
  - IEXPLORE.EXE PID 464

Memory analysis

Hands-on: Step 3

- Search the kernel address space for DynDNS names and dump the results to disk.

Memory analysis
Hands-on: Step 3

```
training@ubuntu:/yara/Lab_4$ mkdir dump
training@ubuntu:/yara/Lab_4$ vol.py -f memory.dmp yarascan -y dyndns.yara -D dump/
Volatile Systems Volatility Framework 2.2
Rule: DynDNS_free
Owner: Process winlogon.exe Pid 624
0x7f77861e  72 00 6d 00 36 00 2e 00 6f 00 72 00 67 00 00 00   r.m.6...o.r.g...
0x7f77862e  3e f4 00 00 00 00 10 8b 85 a0 00 00 00 00 00 00   >...............
0x7f77863e  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x7f77864e  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
Rule: DynDNS_free
Owner: Process services.exe Pid 668
0x004d09c2  72 00 6d 00 36 00 2e 00 6f 00 72 00 67 00 00 00   r.m.6...o.r.g...
0x004d09d2  00 00 2f 00 00 00 dc 59 1e 00 20 00 00 00 10 00   ../....Y........
0x004d09e2  00 00 02 00 04 00 14 00 00 00 00 00 00 00 1c 00   ................
0x004d09f2  00 00 00 00 00 00 cc 4a d8 92 64 6f 6d 61 69 6e   .......J..domain
...
training@ubuntu:/yara/Lab_4$ ls dump/
process.0x80fa53c0.0x4d09c2.dmp    process.0xff492750.0x1e617a.dmp
process.0xff4f1c38.0x7cb25edb.dmp  process.0xff578a18.0x5cb901af.dmp
process.0x80fa53c0.0x4d0a03.dmp    process.0xff492750.0x1e6d37.dmp
process.0xff4f1c38.0x7cb25ef2.dmp  process.0xff578a18.0x5cb90d00.dmp
process.0xff492750.0x170198.dmp    process.0xff492750.0x1e761e.dmp
process.0xff4f1c38.0x7cf25edb.dmp  process.0xff580a98.0x1c5b27.dmp
...
```

Conclusion

Conclusion

Strings

- Text
  - → make use of modifiers: nocase, fullword, ascii, wide
- Hex
  - → make use of wildcards and jumps
- Perl compatible regular expressions

Conclusion

Condition

- Sets
    - ➔ 2 of ($a,$b,$c)
    - ➔ any of them
    - ➔ all of them
- Count number of string matches: #*string*
- Iterator „for“
- Offsets:
    - ➔ at *offset*
    - ➔ entrypoint
- Access raw bytes: int8..int32, uint8..uint32
- Keep your rules simple, reference other rules

Conclusion
Metadata

- Define metadata
  - string
  - integer
  - boolean

- Examples:
  - weight (count of matching bits)
  - architecture
  - algorithm
  - endianess

- Use „-m“ command line option to display metadata

Conclusion
Maintain a set of rules

- One-file-to-keep-them-all doesn‘t work well

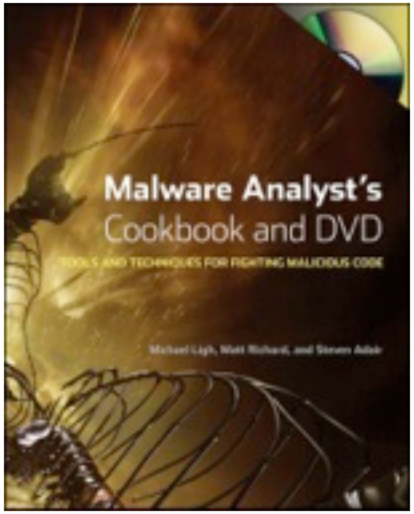
- Refactor your rules
  - ➔ write rules for each common expression („primitives“)
  - ➔ separate files by topic, make use of „include“

- Rule modifiers:
  - ➔ „global“ makes rule a prerequisite for all other rules (e.g. PE header check)
  - ➔ „private“ suppresses output

- Make use of tags and „-t“ command line option to select rules
  - ➔ my tags commonly reflect metadata

Conclusion
More information

- YARA manuals and wiki at
  http://code.google.com/p/yara-project/
- Malware Analyst‘s Cookbook
  - Chapter 3:
    - identify packers
    - sniffer detection
    - malware capabilities
  - Chapter 7: XOR de-obfuscation
- YARA Daemon
  if you need to run many queries with the same rule set (saves compile time!)
  https://github.com/jaimeblasco/AlienvaultLabs/tree/master/yarad


Malware Analyst's
Cookbook and DVD
TOOLS AND TECHNIQUES FOR FIGHTING MALICIOUS CODE
Michael Ligh, Matt Richard, and Steven Adair

Conclusion
Share your knowledge!

- Forum dedicated to the discussion and sharing of YARA rules
  - ➔ Repository on GitHub
  - ➔ Web service to test new rules, scan and download malware
- Founded and moderated by Mila Parkour and Andre DiMino (DeepEndResearch)
- Membership is vetted (send application from your professional email address)
- Active participation is expected and required.
- For details, please see http://www.deependresearch.org/2012/08/yara-signature-exchange-google-group.html

Yara Exchange

Thank you for your attention!

Andreas Schuster

a.schuster@yendor.net
http://computer.forensikblog.de/

YARA: Advanced Topics

26th annual FIRST conference

BOSTON

MASSACHUSETTS

JUNE 22—27, 2014

BACK TO THE 'ROOT' OF INCIDENT RESPONSE

Andreas Schuster

June 25, 2014
Boston

Logistics
Agenda

- Logistics and remarks
- Hands-on: Matching on machine code

- 14:30 - 15:00 Break

- Hands-on: Parsing the PE header
- Remarks on transition from v1.x to v2.x

- 17:00 End

Logistics

Courseware

- Evaluation form
- YARA Cheat Sheet
- Participation in hands-on exercises requires
  - ➔ Lab VM Image
    - on USB thumbdrives (please return as soon as you‘re done!)
  - ➔ VMware {Player, Workstation, Fusion}
  - ➔ VirtualBox may work, too (untested!)
    1. Create new machine
    2. Select RAM (512 MB)
    3. Select „existing disk“ and point to .vmdk

Logistics
Starting the VM

- Start VM
- Log in as user „training“, password is „training“
- „training“ also is your sudo password
- You may want to customize the keyboard layout:
    - ➔ System > Preferences > Keyboard
    - ➔ Select „Layouts“ tab
- Open a terminal window

Logistics

Materials in the VM

- Documentation (in /yara/doc):
  - Intel 64 and IA-32 Architectures Software Developer's Manual
  - x86 Opcode Structure and Instruction Overview by Daniel Plohmann
  - PE format description
  - Perl Compatible Regular Expression Manual
  - YARA Cheat Sheet
  - YARA v1.6 User's Manual

Logistics
Materials in the VM

- Exercises:
  - /yara/Lab_1
    ...
  - /yara/Lab_6
  - /yara/Labs_restore.zip: archived lab materials in case something goes wrong
  - /yara/malware: live malware
- Slide deck
- **WARNING! Virtual machine image contains live malware samples. Do not extract and expose to Microsoft Windows (or an emulator).**

Matching on Machine Code

Objective

- During this hands-on lab, you will learn
  - ➔ a workflow to gradually improve your rules
  - ➔ how to build binary signatures that match on x86 machine code
  - ➔ how to organize a repository based on a categorization by YARA
  - ➔ how to consolidate rules

About the malware samples

- cd /yara/Lab_5
- Directory „incoming“ holds several new malware samples for us to analyze. All samples are backdoors belonging to the Hoardy / Vilsel /Phindolp / Ke3chang family. Your overall task is to categorize these samples based on their decryption routine.
- For selected samples you will find some disassembly listings.

The first decryption routine

- Disassembly of sample 44efa4accc42aa55d7843ec69161c8ca:

```
.text:00401723                          decrypt:
.text:00401723 89 45 E8                 mov     [ebp+0BB4h+var_BCC], eax
.text:00401726 3B C7                    cmp     eax, edi
.text:00401728 7D 18                    jge     short end
.text:0040172A 8A 88 F0 E8 40 00        mov     cl, buffer[eax]
.text:00401730 32 C8                    xor     cl, al
.text:00401732 2A C8                    sub     cl, al
.text:00401734 80 E9 5A                 sub     cl, 5Ah
.text:00401737 88 88 F0 E8 40 00        mov     buffer[eax], cl
.text:0040173D 83 C0 01                 add     eax, 1
.text:00401740 EB E1                    jmp     short decrypt
```

The first decryption routine

- Create a rule file named „hoardy.yara“.
- Create a YARA rule which matches on the bytes that are typeset in bold letters (see previous page).
- Name your rule „crypto1“ and tag it as „category“.
- Name the string „$crypto1“, too.
- Try your rule on all the samples in „incoming“. How many samples match your rule?

Find the „Known Unknowns“

There are known knowns; there are things we know that we know.

There are known unknowns; that is to say there are things that, we now know we don't know.

But there are also unknown unknowns – there are things we do not know we don't know.
—United States Secretary of Defense, Donald Rumsfeld (2002-02-12)

Find the „Known Unknowns“

- Create a rule named „unknown“. This rule shall match on all samples that are NOT detected by rule „crypto1“.
- Remember:
  - ➔ A rule does not have to contain a „strings“ section.
  - ➔ A rule can refer back to rules defined earlier.

Exploring the „Unknowns“

- We pick one of the „unknown“ samples, e.g. 026936afbbbdd9034f0a24b4032bd2f8 and disassemble it:

```
.text:004033A1                          decrypt:
.text:004033A1 3B C3                    cmp     eax, ebx
.text:004033A3 7D 18                    jge     short end
.text:004033A5 8A 88 C0 E5 40 00        mov     cl, buffer[eax]
.text:004033AB 32 C8                    xor     cl, al
.text:004033AD 2A C8                    sub     cl, al
.text:004033AF 80 E9 7C                 sub     cl, 7Ch
.text:004033B2 88 88 C0 E5 40 00        mov     buffer[eax], cl
.text:004033B8 83 C0 01                 add     eax, 1
.text:004033BB EB E4                    jmp     short decrypt
```

- Compare samples 026936afbbbdd9034f0a24b4032bd2f8 and 44efa4accc42aa55d7843ec69161c8ca.

- Why does rule „crypto1“ not match? What has changed?

Exploring the „Unknowns“

- Create a rule named „crypto2“ with tag „category“ that matches on the decryption routine of sample 44efa4accc42aa55d7843ec69161c8ca.
- How many samples are detected by this rule?
- Update your rule „unknown“. What samples are still not identified?

Exploring the „Unknowns“

- Repeat this workflow, until all samples are accounted for.
  - ➔ Create rule „crypto3“ from disassembly of sample 057cb5a62199afbb49a98b3a93f2149d
  - ➔ Create rule „crypto4“ from disassembly of sample 072af79bb2705b27ac2e8d61a25af04b
  - ➔ Create rule „crypto5“ from disassembly of sample 4c46abe77c752f21a59ee03da0ad5011
  - ➔ Attach the tag „category“ to all of these rules.

Organize your repository

- „repo“ is your - still empty - repository.

```
training@ubuntu:/yara/Lab_5$ ls -lR repo/
repo/:
total 20
drwxr-xr-x 2 training training 4096 2014-01-20 00:02 crypto1
drwxr-xr-x 2 training training 4096 2014-01-20 00:02 crypto2
drwxr-xr-x 2 training training 4096 2014-01-20 00:02 crypto3
drwxr-xr-x 2 training training 4096 2014-01-20 00:02 crypto4
drwxr-xr-x 2 training training 4096 2014-01-20 00:02 crypto5

repo/crypto1:
total 0

repo/crypto2:
total 0
...
```

- Your next job is to populate your repository with the new samples from the „incoming“ directory.

Organize your repository

- We limit YARA‘s output to rules tagged with „category“:

```
training@ubuntu:/yara/Lab_5$ yara -r -t category hoardy.yara incoming
crypto2 incoming/1ae06edd0ea2df734e357698bcdf8f30
crypto5 incoming/4c46abe77c752f21a59ee03da0ad5011
crypto2 incoming/5ee64f9e44cddaa7ed11d752a149484d
...
```

- A shell one-liner then moves/copies/links the files into their proper directory:

```
training@ubuntu:/yara/Lab_5$ while read CATEGORY FILE ; \
     do cp ${FILE} repo/${CATEGORY}/ ; \
     done < <(yara -r -t category hoardy.yara incoming)
```

- Use the following commands:
  - `cp` for copying (safe)
  - `mv` for moving (most common case for repositories)
  - `ln` for linking (when one file can exist in multiple categories)

Organize your repository

```
training@ubuntu:/yara/Lab_5$ ls -R repo/
repo/:
crypto1  crypto2  crypto3  crypto4  crypto5

repo/crypto1:
44efa4accc42aa55d7843ec69161c8ca  979c37df230a83ffab32baf03f0536ac
4652d041244c06b8d58084312692b85e  a738badbeca89b6a79b2f098c817bca2

repo/crypto2:
026936afbbbdd9034f0a24b4032bd2f8  5ee64f9e44cddaa7ed11d752a149484d
1ae06edd0ea2df734e357698bcdf8f30

repo/crypto3:
057cb5a62199afbb49a98b3a93f2149d  c2c1bc15e7d172f9cd386548da917bed
277487587ae9c11d7f4bd5336275a906  c718d03d7e48a588e54cc0942854cb9e
34252b84bb92e533ab3be2a075ab69ac  e4d8bb0b93f5da317d150f039964d734
703c9218e52275ad36147f45258d540d

...
```

Consolidate your rules

- Having a multitude of elaborate rules is fine for classification of malware in your lab.
- For detection, e.g. VirusTotal or heavy-duty online traffic monitoring, your priorities shift to small and fast rules.
- Your next task will be to consolidate the five categorization rules into a single rule with at maximum two strings.

Consolidate your rules

- Create a new rule, named „combined“ and tag it with „summary“
- Build its strings section from the binary strings in the five „crypto“ rules.
- Rework the „unknown“ rule as follows:

```
rule unknown: summary
{
    condition:
        not combined
}
```

- Run YARA on your repository and limit its output to rules tagged with „summary“.
- Does „unknown“ match on any files?

Consolidate your rules

- We can now merge strings „crypto1“ and „crypto2“ by using wildcards (this honors the different XOR keys):

  ```
  $crypto1  = { 32 c8 2a c8 80 e9 5a 88 }

  $crypto2  = { 32 c8 2a c8 80 e9 7c 88 }
  ```

  into

  ```
  $crypto12 = { 32 c8 2a c8 80 e9 ?? 88 }
  ```

- Run again with the modified rule and check for missing („unknown“) files:
  $ yara -t summary -r hoardy.yara repo

- Merge „crypto4“ and „crypto5“ in the same way and test (this again affects XOR keys).

- Finally merge „crypto12“ and „crypto45“ and test again (this masks register bits).

Consolidate your rules

- In a last step, merge strings „crypto1245“ and „crypto3“.
- Remember two regex operators:
  - ➔ ( ) groups items
  - ➔ *a* | *b* matches either on *a* or *b*
  - ➔ see /yara/doc/yara/pcre.txt for details
- Run YARA again with the modified rule and one again check for missing („unknown“) files:

  ```
  $ yara -t summary -r hoardy.yara repo
  ```

Summary

- You have written signatures that are:
  - ➔ robust against slightly modified obfuscation schemes (different key)
  - ➔ robust against relocation (different addresses)
  - ➔ robust against usage of differtent registers
    (registers are commonly selected by compiler based on context)
- You have categorized a batch of new malware samples and moved them into your repository.
- You have consolidated a rule set in order to improve speed and maintainability.

Parsing a PE File

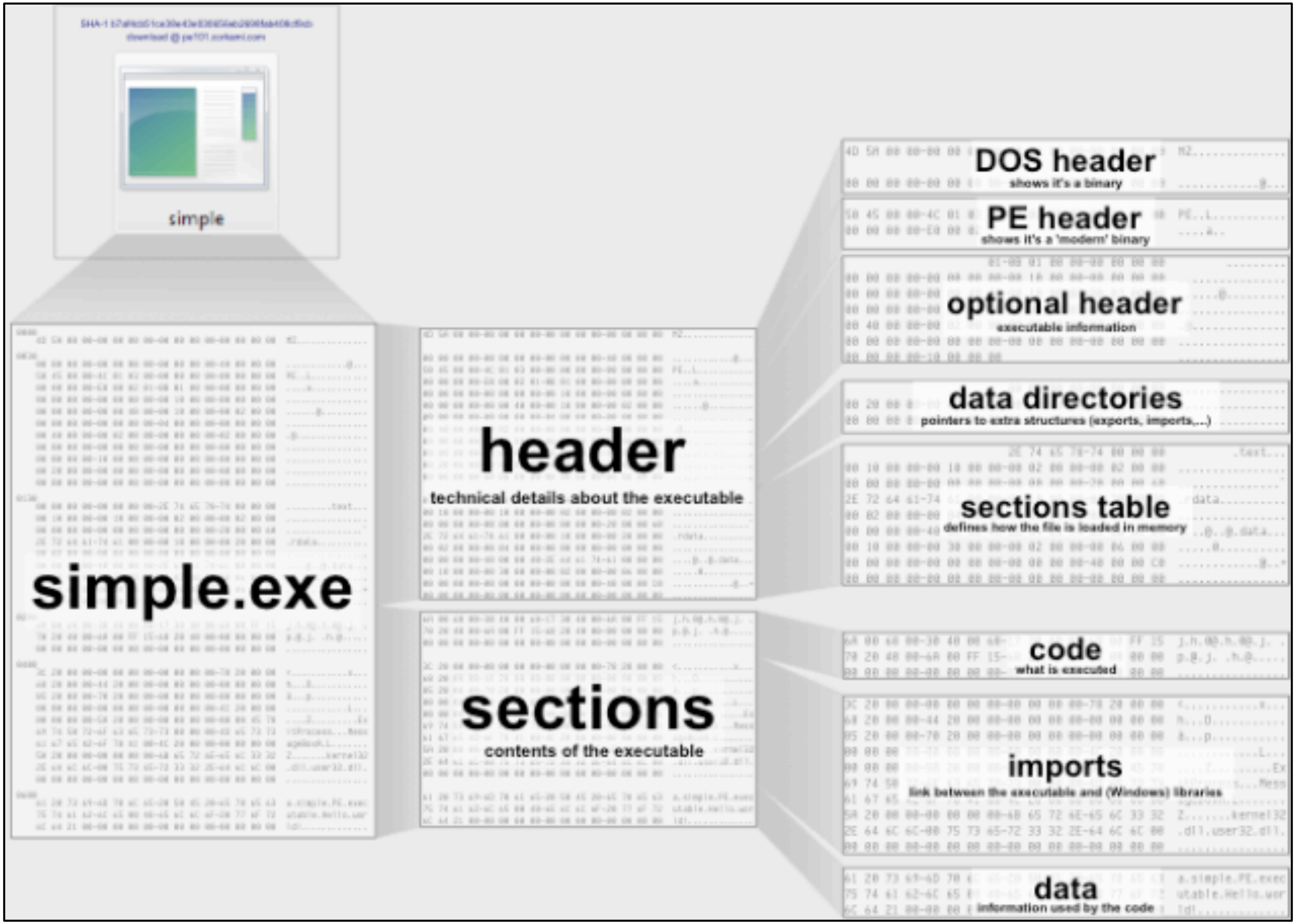
Parsing a PE file

- Overall goal is to limit a search to a certain section of a PE file.
- Suggested steps to go there:
  - Learn about the PE file format
  - Find relevant data in the PE header
  - Rule to identify a dropper limits search to .rsrc, while backdoor rule will search in .data only.

PE format

- PE = Portable Executable
- Structured format for executable files
- Supporting documents in /yara/doc/PE
  - Overview by Ange Albertini
  - Specification v8.3 by Microsoft (2013)

PE format


download @ pe101.corkami.com
simple
simple.exe
header
technical details about the executable
sections
contents of the executable
DOS header
shows it's a binary
PE header
shows it's a 'modern' binary
optional header
executable information
data directories
pointers to extra structures (exports, imports,...)
sections table
defines how the file is loaded in memory
code
what is executed
imports
link between the executable and (Windows) libraries
data
information used by the code

Your first task

- We‘ve implemented a (simplified) detection rule at a malware repository and found a few files. Some are simple droppers, others are the dropped backdoors. In order to speed up processing, we want to categorize our samples with YARA.

- What we know:
  - ➔ All samples contain the string „~ISUN32“.
  - ➔ All samples are PE files for Microsoft Windows, 32bit.
  - ➔ Backdoors contain the string in their .data section.
  - ➔ Droppers carry a backdoor (and hence the string) in their .rsrc section.

- Your first task is to develop a plan:
  - ➔ What information do you need?
  - ➔ Where can you find this information in a PE file?

Learn about the section table

- Information about sections can be found in the section table.
- Review the PE format specification (/yara/doc/PE/pecoff_v83.pdf), section 3, pages 24-26.
- Where can we find the location info? What are the field names, what are their offsets and types?
- Remember: we are dealing with an „executable image“, not an „object“.

One last question

- One last question remains:
  How can we find the proper entry in the section table?
- There are at least two different ways. They also differ in their difficulty (and computational complexity). Try to find a fast and easy solution. You may have to make extra assumptions.
- Write the rule for the dropper first.
- Remember: in order to classify as a „dropper“, the string „~ISUN32“ needs to appear within in .rsrc section.

Searching for backdoors

- Now write a rule to match on backdoors.
- Remember: The string „~ISUN32“ now has to appear in the „.data“ section.
- You may reuse code from the dropper rule ;)

- Test your rules on the samples in /yara/Lab_6/incoming.
- How many droppers and how many backdoors do you find?
- Bonus excercise: populate the repository in /yara/Labs_6/repo with the samples in „incoming“, based on your classification rules.

Summary

- You‘ve used nested uint32() function calls to parse a file, based on its format specification.
  - ➔ Similar functions do exist for 8 and 16 bits, and for signed and unsigned integers.
  - ➔ All of these functions read integers in little endian (Intel) byte order only.
- You‘ve used this method to limit string matching to certain parts of a Portable Executable.
  - ➔ You can use it to access lots of other information from PE files, e.g. linker version and timestamp, DLL vs. EXE, section characteristics
  - ➔ You can parse other file formats that are structured in a similar way, e.g. PNG

Migration from YARA v1 to v2

Migration from YARA v1 to v2

Attention, all C programmers

- Different application binary interface for C library
- No changes required for Python bindings

- Benefit: libyara is now thread-safe and much faster than prior versions.

Migration from YARA v1 to v2

Attention, all script authors

-
  ```
  $ yara -v
  yara 1.6 (rev:129)
  ```

  -
    ```
    $ yara good_rule.yara somefile ; echo $?
    1
    ```

  -
    ```
    $ yara bad_rule.yara somefile ; echo $?
    0
    ```

-
  ```
  $ yara -v
  yara 2.1
  ```

  -
    ```
    $ yara good_rule.yara somefile ; echo $?
    0
    ```

  -
    ```
    $ yara bad_rule.yara somefile ; echo $?
    1
    ```

Migration from YARA v1 to v2

Attention, all script authors

- Exit status codes changed from v1 to v2.
- Exit status codes from v2 onward are POSIX compliant
- Attention all batch/script coders:
  - ➔ check YARA version (yara -v), or
  - ➔ let YARA run on known good and bad rule files and observe status codes

Migration from YARA v1 to v2

Attention, all script authors

```
# Check YARA's return codes for good and broken rules.
YARA_OK := $(shell \
        PROBE=`mktemp ./yaratemp.XXXXXX` || exit 1; \
        printf "YARA probe file\n" > $${PROBE}; \
        RULE=`mktemp ./yaratemp.XXXXXX` || exit 1; \
        printf 'probe' > $${PROBE}; \
        printf 'rule test {condition: true}' > $${RULE}; \
        $(YARA) $${RULE} $${PROBE} 1>$(NULL) 2>$(NULL); GOOD=$$?; \
        echo 'rule test {condition: invalid_keyword}' > $$RULE; \
        $(YARA) $${RULE} $${PROBE} 1>$(NULL) 2>$(NULL); FAIL=$$?; \
        if [ $$GOOD -eq $$FAIL ]; \
        then \
                printf "Fatal: unable to detect broken rules.\n" 1>&2; \
                echo "127"; \
        else \
                echo $${GOOD}; \
        fi; \
        rm $${PROBE} $${RULE}; )
```

Migration from YARA v1 to v2

Boolean shortcut evaluation

- Boolean shorcut evaluation missing in v2.
- Example: Rule ensures that it deals with a PE file, then does some computational expesive processing (e.g. nested loops)

```
condition:
  uint16(0) == 0x5a4d and uint16(uint32(0x3c)) == 0x4550
  and
  for 2 i in (0..(uint16(uint32(@section[1]+20) + 0xc) - 1 )) :
    (for any of ($name_*) :
      ($ at ((uint32(uint32(@section[1]+20) + 0x10 + 8*i) & 0x7fffffff)
        + uint32(@section[1]+20))))
```

- Works in v1, but may take insanely long time in v2!

Migration from YARA v1 to v2
Custom regex engine

- v1.6: PCRE
- v1.7: PCRE or RE2
- v2.0: custom regex engine
  - ➔ no more backreferences
    e.g. <([A-Z][A-Z0-9]*)\b[^>]*>.*?</\1>
  - ➔ no POSIX character classes
    e.g. [:space:]
- Benefit: The new engine is faster than any of the standard libraries.

Migration from YARA v1 to v2

Regex greediness

-
  ```
  $ cat rule.yara
  rule test
  {
  strings:
    $re = /[a-zA-Z ]+/
  condition:
    $re
  }
  ```

-
  ```
  $ cat data.txt
  This is a test
  ```

Migration from YARA v1 to v2

Regex greediness

-
  ```
  $ yara -v
  yara 1.6 (rev:129)
  ```

-
  ```
  $ yara -s rule.yara data.txt
  test data.txt
  0x0:$re: This is a test
  ```

Migration from YARA v1 to v2

Regex greediness

-
  ```
  $ yara -v
  yara 1.7 (rev:167)
  ```

-
  ```
  $ yara -s rule.yara data.txt
  test data.txt
  0x0:$re: This is a test
  0x1:$re: his is a test
  0x2:$re: is is a test
  0x3:$re: s is a test
  0x4:$re: is a test
  0x5:$re: is a test
  0x6:$re: s a test
  0x7:$re: a test
  0x8:$re: a test
  0x9:$re: test
  0xa:$re: test
  0xb:$re: est
  0xc:$re: st
  ```

Solutions

- A PDF with all the exercises and solutions (slides with a red bar) will be available
  - ➔ from Monday June 30, 2014
  - ➔ at http://r.forens.is/first2014sol
- Or send me an email at a.schuster@yendor.net

Thank you for your attention!

Andreas Schuster

a.schuster@yendor.net
http://computer.forensikblog.de/