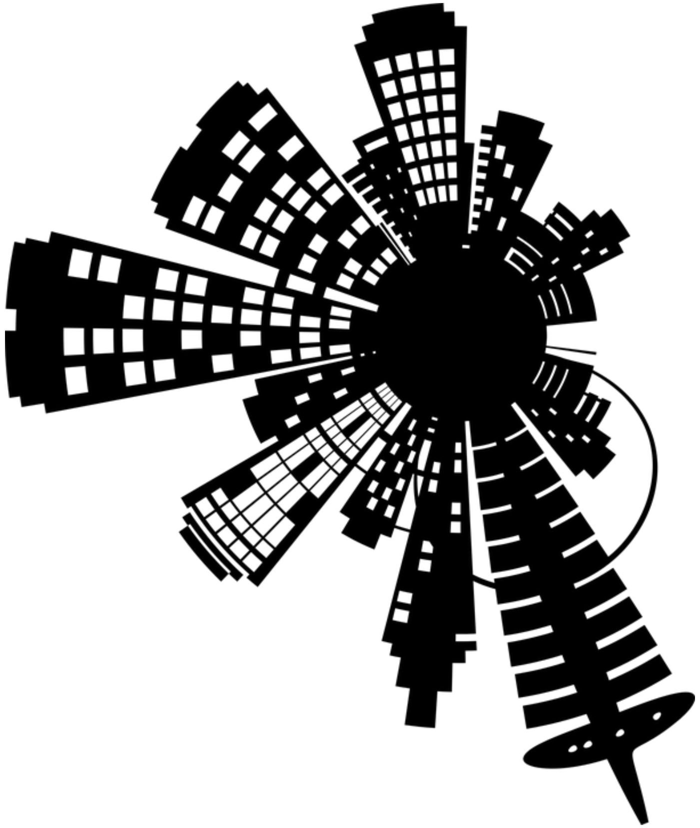
CISO Platform Virtual Summit

17-18 July 2020

Workshop on reverse engineering and signature generation



Andrea Marcelli, PhD
Malware Researcher
Cisco Talos
@_S0nn1_

Outline

Part 00 - Introduction (15 mins)

Part 01 - PE file format (35 min)

- PE walkthrough
- Packers

Part 10 - Malware analysis and automation (40 min)

- Python scripting
- FIRST
- GhIDA

Part 11 - Automatic signature generation (60 min)

- The theory
- YaYaGen

whoami

- PhD in computer Engineering from Politecnico di Torino, Italy
- Malware Research Engineer at Cisco Talos since 2019
- Previously at Hispasec Sistemas, working on Android malware analysis and automation
- Interests: malware analysis, phishing detection, semi-supervised modeling

  https://jimmy-sonny.github.io/

Part 00

Introduction

SACON 2020

About reverse engineering

Reverse Engineering is a process where a man-made product is dissected and deconstructed to its original design, architecture, code
- going back through the development cycle

Goals:
- gain knowledge
- create compatible products
- make interoperation more effective
- uncover undocumented features
- …

Well-known examples:
- Samba software re-implements the SMB networking protocol for file sharing
- Wine project for Windows API
- AMD Am9080 reverse-engineered the Intel 8080 processor

Binary analysis

Binary analysis is the art of understanding binaries (i.e., executable programs)

Binary compilation

- Processor executes machine instructions (arch. specific)

- Compiled vs interpreted language

- Machine code (e.g., C, C++, GO) vs bytecode (e.g., Java, C#)
  - machine code is architecture specific
  - bytecode is runtime environment specific

- For (machine code) compiled languages, several steps are involved:
  - Preprocessing
  - Compilation
  - Assembling
  - Linking

Binary compilation is the process that transform the source code into an executable binary.

Binary analysis

It’s hard, because of the semantic gap

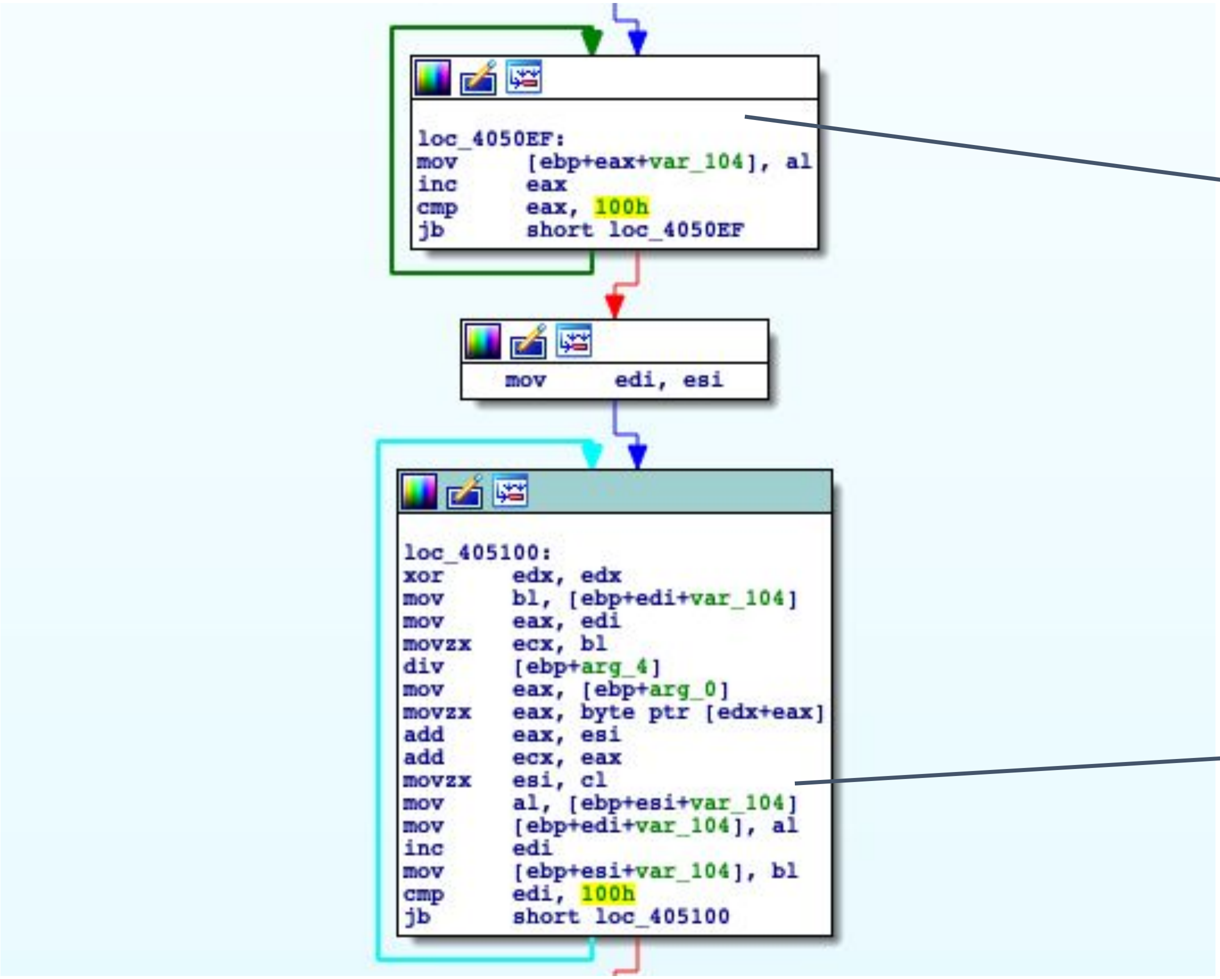
Some languages may be easier than others

Obfuscation complicates the RE process

Tools are essentials, but experience plays a big role.

Binary analysis

Binary analysis



Create and initialize the substitution box

Scramble the substitution box

Binary analysis

- There are two loops with 256 (0x100) iterations.
- The first loop initializes an array with values from 0 to 255.
- …

It’s the initialization of the **RC4 algorithm**.

[Talos Blog: RC4 Encryption in Malware]

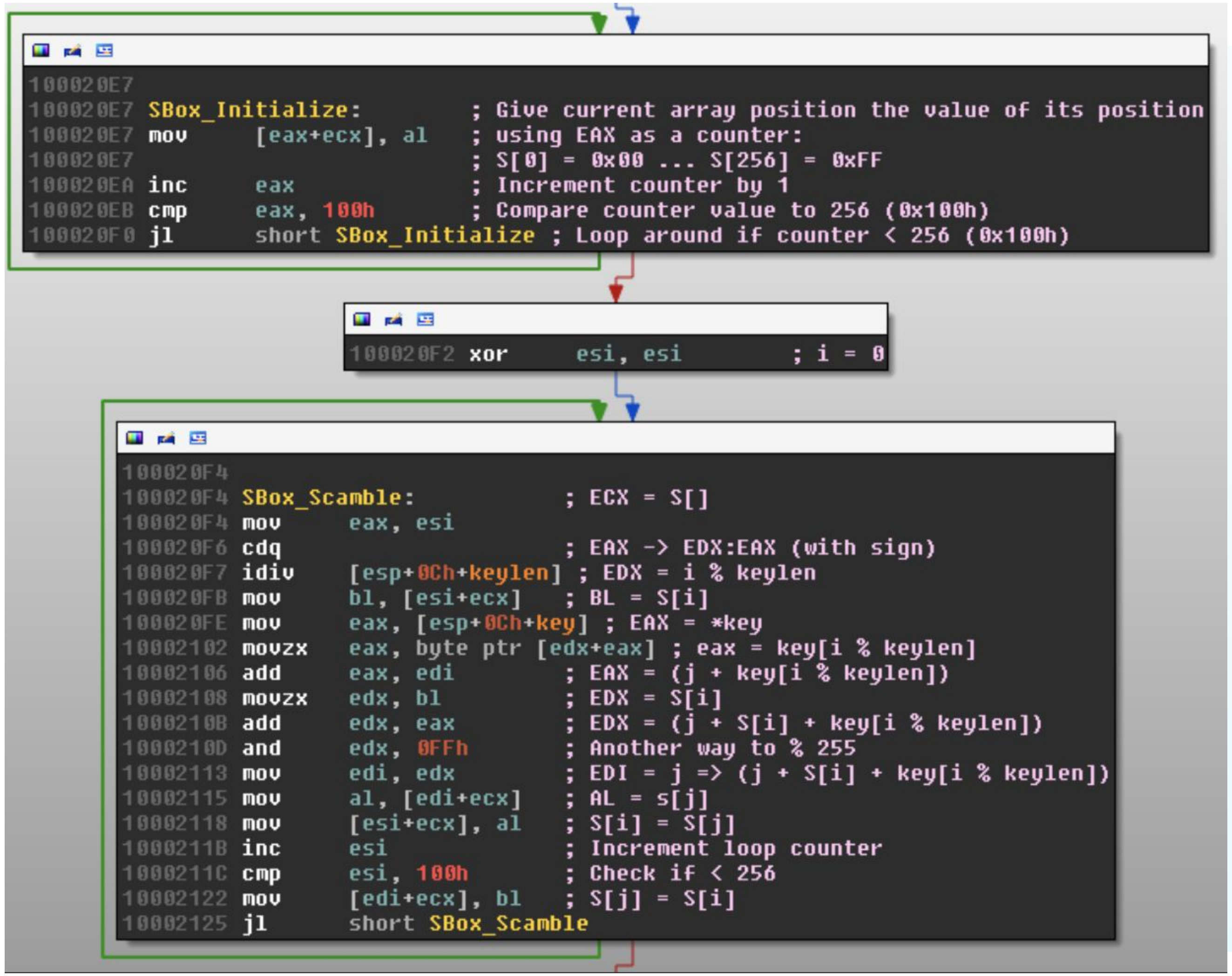
Binary analysis

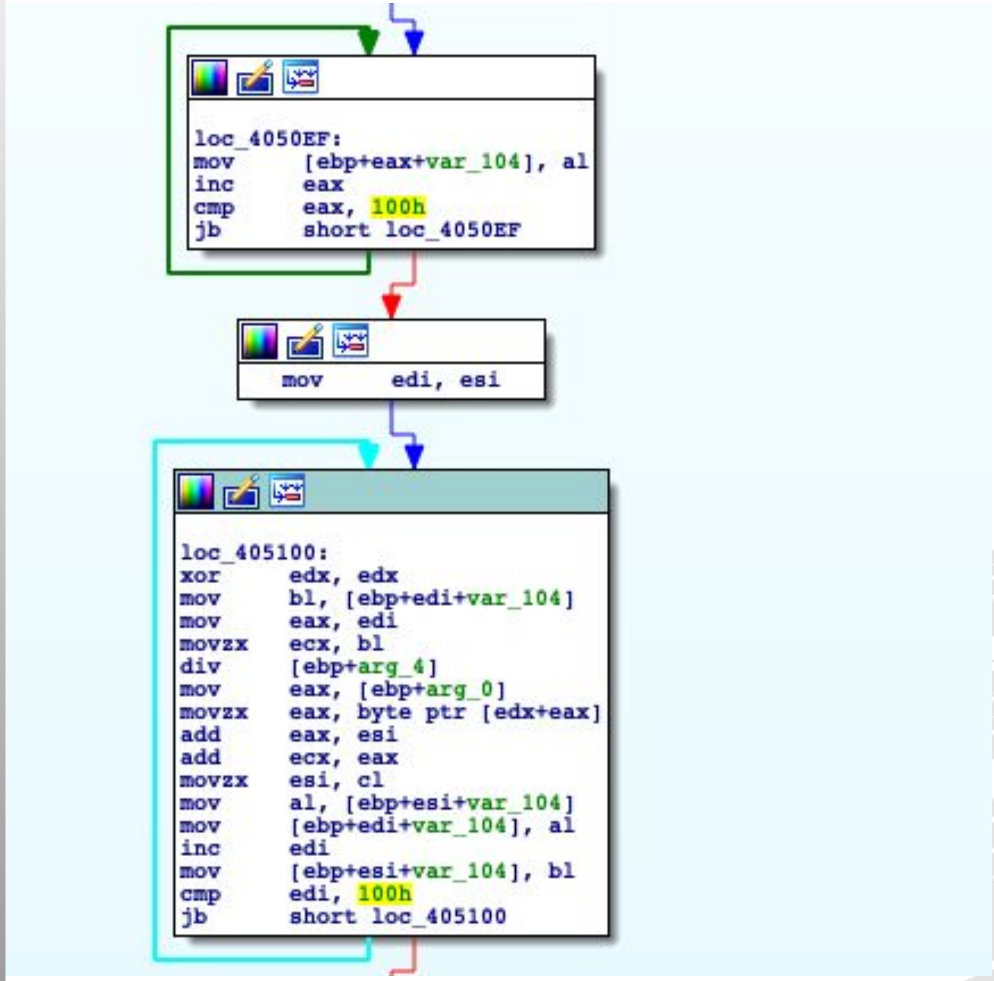
```
for i from 0 to 255
    S[i] := i
endfor


j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

It’s the initialization of the **RC4 algorithm**.

[Talos Blog: RC4 Encryption in Malware]

Binary analysis





Binary analysis

RC4 in PowerEmpire

```
$R={$D,$K=$ARGs;
$S=0..255;
0..255|%{$J=($J+$S[$_]+$K[$_%$K.COuNT])%256;
$S[$_],$S[$J]=$S[$J],$S[$_]};
$D|%{$I=($I+1)%256;
$H=($H+$S[$I])%256;
$S[$I],$S[$H]=$S[$H],$S[$I];
$_-bXOr$S[($S[$I]+$S[$H])%256]}};
```

https://plaintext.do/AV-Evasion-Converting-PowerEmpire-Stage-1-to-CSharp-EN/

Binary analysis

Static analysis:

- Extract strings, symbols and API calls
- Disassembler: from machine code to assembly

Dynamic analysis:

- Debugger: debug the environment
- Instrumentation frameworks: inject code in the program execution
- Sandbox: capture the interaction with the OS

Part 01

The portable executable file format

PE walkthrough

The PE:

- is the native standard for Microsoft Windows 32 and 64 bit executable file
- it was introduced in Windows NT 3.1

Contains the ***DOS header***, the ***PE header***, the ***Sections table*** and the ***Sections***

DOS header

It starts with 0x4D 0x5A “MZ”, the initials of Mark Zbikowski

lfanew offset to the PE header

```
Offset   0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

000000  4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00   MZ..............
000010  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00   ........@.......
000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000030  00 00 00 00 00 00 00 00  00 00 00 00 E0 00 00 00   ................
000040  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68   ........!..L.!Th
000050  69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F   is program canno
000060  74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20   t be run in DOS 
000070  6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00   mode....$.......
000080  EC 85 5B A1 A8 E4 35 F2  A8 E4 35 F2 A8 E4 35 F2   ..[...5...5...5.
000090  6B EB 3A F2 A9 E4 35 F2  6B EB 55 F2 A9 E4 35 F2   k.:...5.k.U...5.
0000A0  6B EB 68 F2 BB E4 35 F2  A8 E4 34 F2 63 E4 35 F2   k.h...5...4.c.5.
0000B0  6B EB 6B F2 A9 E4 35 F2  6B EB 6A F2 BF E4 35 F2   k.k...5.k.j...5.
0000C0  6B EB 6F F2 A9 E4 35 F2  52 69 63 68 A8 E4 35 F2   k.o...5.Rich..5.
0000D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
0000E0  50 45 00 00 4C 01 03 00  87 52 02 48 00 00 00 00   PE..L....R.H....
0000F0  00 00 00 00 E0 00 0F 01  0B 01 07 0A 00 78 00 00   .............x..
```

(Beginning of **NOTEPAD.EXE**; **Windows™ XP Pro SP-3**; **April 14, 2008**, **4:00:00 AM**, **69,120** bytes.)

Figure 3.

PE header

Begins with 0x50, 0x45, 0x00, 0x00 (“PE00”)

Contains the *FileHeader* and the *OptionalHeader*

- FileHeader
  Machine, NumberOfSections and ***SizeOfOptionalHeader***

- OptionalHeader
  AddressOfEntryPoint *and* ***ImageBase***
  SectionAlignment and ***FileAlignment***
  SizeOfImage is the overall size of the PE image in memory
  DataDirectory used for import table and export table

The Section Table

Section table contains information about each section:

- the ***NumberOfSections*** is located in the *FileHeader*
- Sections are sorted according to their RVA (Relative Virtual Address)
- Most section names start with ".", but this is not a requirement
  e.g., *.text*, *.data*, *.reloc*

Some fields:

- ***SizeOfRawData*** is the size of the section on the disk
  - rounded up to next multiple of ***FileAlignment***
- ***VirtualSize*** is the size of the section when it's loaded in memory
- ***VritualAddress*** is the address of the first byte of the section relative to the ***ImageBase***
  e.g., VA: *0x1000*, PE loaded at *0x400000*, the section will be loaded at *0x401000*
- ***Characteristics*** indicate whether the section contains code, initialized data, *rwe* permissions.

The Sections

.text: Executable Code Section
The linker concatenates all the .text sections from the object files into one big .text section.
Contains the program Entry Point and the Jump Table.

.data or **.rdata**: Data Section
global and static variables initialized at compile time

.bss:
uninitialized data, including all the variables declared *static* or *global*

.rsrc: Resource Section
data is structured into a resource tree. Most common resources are Icons and GUI.

The Sections

.edata: Export Data Section
list of functions and data exported to other modules. Used by DLLs.

.idata: Import Data Section
contains the Import Directory and the Import Address Table
when calling a function in a DLL, the call transfer controls to a jmp instruction

.debug: Debug Information Section

Relocations

The linker makes an assumption about where the file will be mapped into memory:

- In Win32 the default Base Address is 0x400000

The binary may be loaded somewhere else:

- *.reloc* section contains the information to fix the addresses

Usually *jmp* and *call* instructions use relative offsets.

- Relocations are needed for instructions that reference some data.

The windows loader

The loader:

- Creates a virtual address space for the process
- Using the Section headers, it maps in memory the sections of the file.
  Page attributes are set according to the section ***Characteristic***.
- It performs relocations if the load address is not equal to the preferred base address
- ***ImportTable*** is used to add required DLLs. Address in the IAT are fixed to point to the address of the imported functions
- Creates the stack and the heap
- Creates the initial thread and it passes the execution to the program entry point

Modifying a PE file

- Adding code to an existing section
- Enlarge an existing section
- Add a section
- Add an overlay: append data at the end of the PE file.

Part 01

Packers

Packers

Originally designed to
- reduce the size of an executable (compression)
- protect intellectual property (encryption)

Encrypts or *compress* an executable file:
- It may change the PE sections
- *AddressOfEntryPoints* points to the unpacking routine

It's one of the most common techniques to obfuscate a binary:
- To evade static detection (AV signature)
- Make analysis more difficult

Unpacking problems

Packer complexity varies

- Compressor (LZMA) / encryptor (XOR, RC4, AES) / protector (anti-analysis tricks)
- Code virtualization (e.g., VMProtect)
- Nested packing

General unpacking is a problem:

- Goal: extract the payload in an automated way
- Packer logic varies
- Custom packers

Unpacking

ClamAV automatically unpacks UPX, FSG and Petite

https://www.clamav.net/

Packer detection

How to detect a packer?

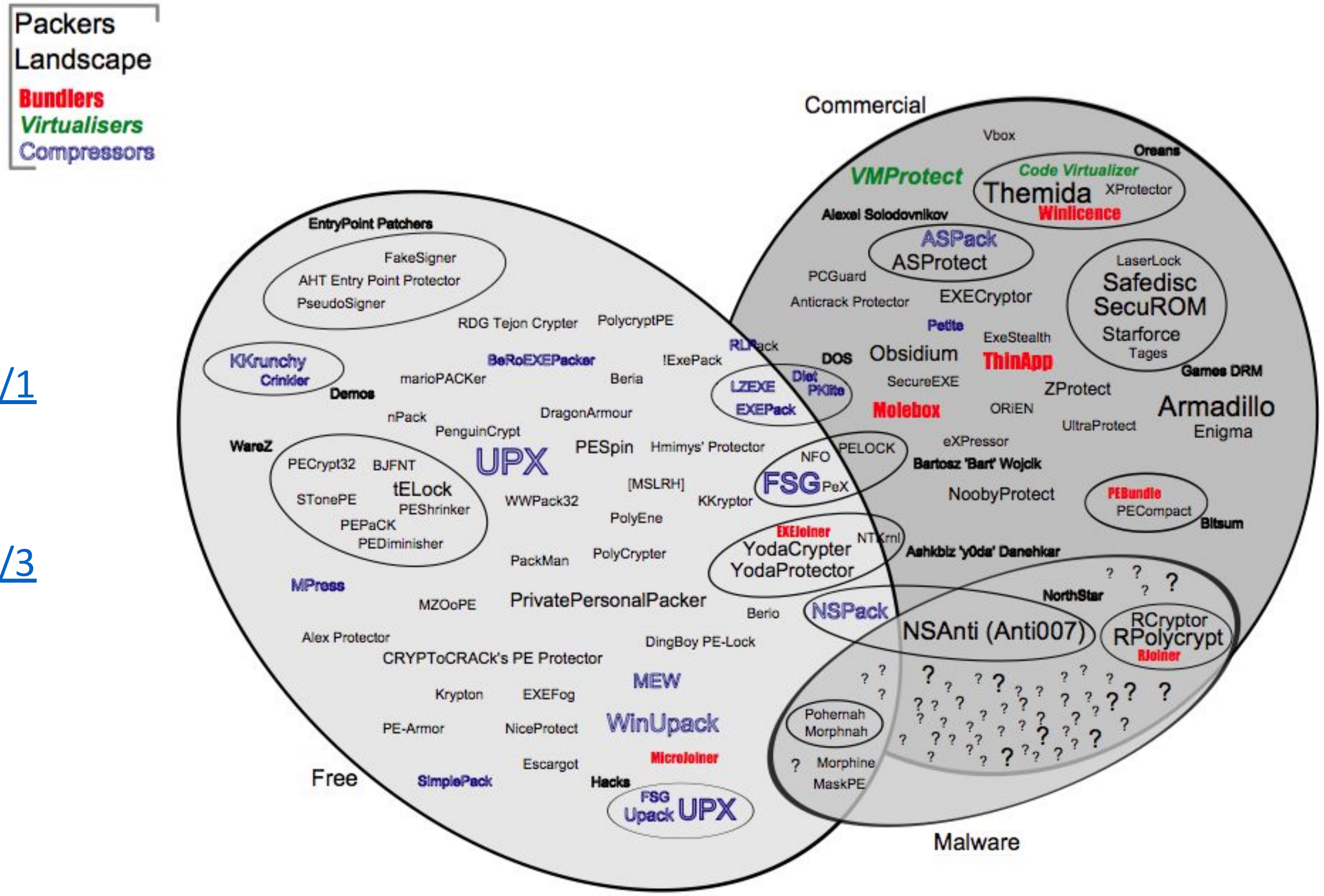
- Look at section names
- Check section permissions
- Check imports
- Check strings
- Check *RawSize* and *VirtualSize*
- Check sections’ entropy
- Packers signature (e.g., PEid - but be aware of FPs)

Packers complexity

Example:

- UPX. Type 1 packer. 2 layers
  https://www.packerinspector.com/example/1

- Themida. Type 3 packer. 41 layers
  https://www.packerinspector.com/example/3



SACON 2020

UPX

DEMO - Microsoft Write

```
unpacked: e46620bd4eb048fcb2a8f1541d2dbda8299e38e01a4eef9c4e7c3c43b96d0629
packed:   98667da25a8d0b08b360d919ca3a32d4f20d38b43aa38ad354d9366540367ec1
```

UPX

Ultimate Packer for eXecutables

https://github.com/upx/upx

```
./upx -1 write.exe
```

Name	Entropy	SizeOfRawData	VirtualSize	VirtualAddress
.text	5.628278	4096	3780	4096
.data	0.419103	512	1784	8192
.pdata	1.442970	512	168	12288
.rsrc	4.620039	3584	3496	16384
.reloc	0.221676	512	56	20480

Name	Entropy	SizeOfRawData	VirtualSize	VirtualAddress
UPX0	0.000000	0	24576	4096
UPX1	7.265725	3584	4096	28672
.rsrc	4.438244	4096	4096	32768

UPX packed

Consequences of packers

`Packed != malicious`

This is an old experiments (2003), but still gives the idea:
https://sarvamblog.blogspot.com/2013/05/nearly-70-of-packed-windows-system.html

7,983 samples from different versions of Windows packed with 4 packers submitted to VT, looking for 10+ detections

Packer	Total # of Packed Exes	# of Packed files with at least 10 AV labels	Corresponding %
UPX	4694	0	0
Upack	5250	5244	99.88
NsPack	5191	5125	98.72
BEP	1528	1109	72.78

*VT should not be used for comparing AV products

Packers - References

SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers

https://www.packerinspector.com/about (currently offline)

https://www.usenix.org/node/208120

Part 10

Malware analysis and automation

Demo

Python scripting with pefile and python-idb

https://pypi.org/project/pefile/

https://github.com/williballenthin/python-idb

FIRST architecture

FIRST

FIRST: Function Identification & Recovery Signature Tool

Collaborative platform for reverse engineering

Functions and metadata are saved on a DB

Server-side similarity engines look up similar function

Official IDA Pro plugin and un-official R2 plugin.

FIRST architecture

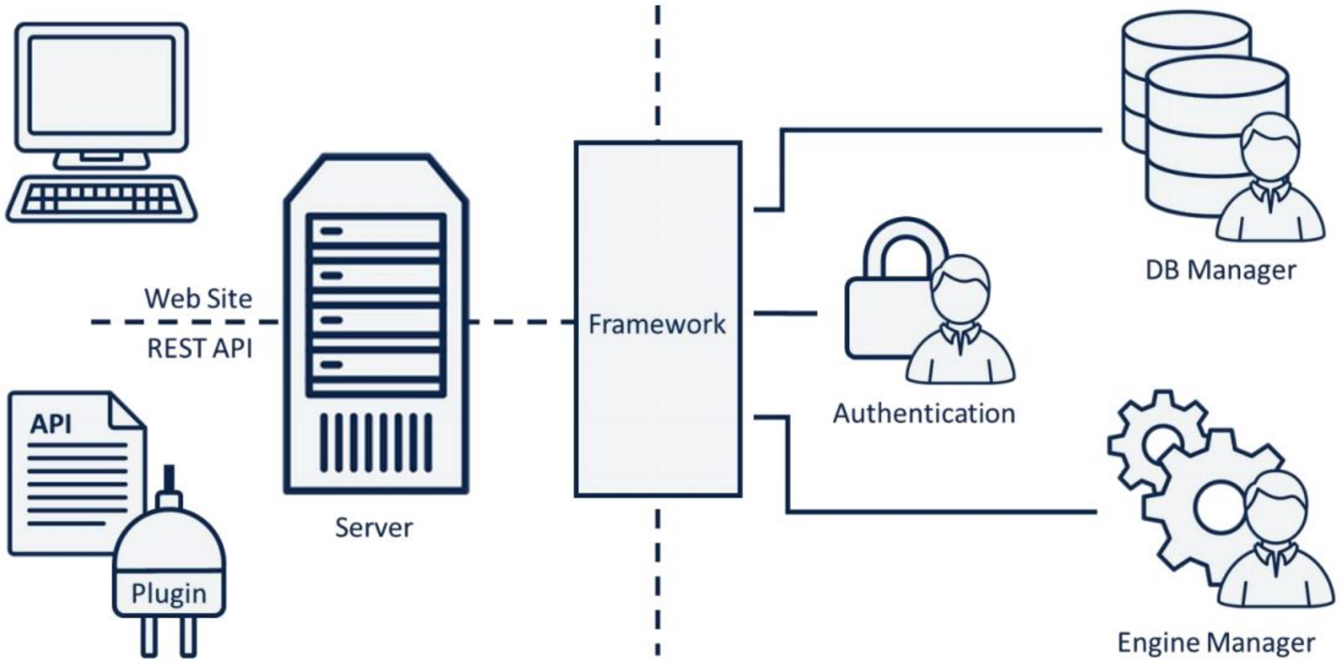


Figure 1: FIRST framework overview

FIRST APIs

api/sample/checkin/<api_key>
sample check-in

api/metadata/add/<api_key>
add function metadata

api/metadata/history/<api_key>
function metadata history

api/metadata/applied/<api_key>
metadata applied

api/metadata/unapplied/<api_key>
metadata unapplied

api/metadata/delete/<api_key>/<id>
delete function metadata

api/metadata/created/<api_key>/<page>
metadata created

api/metadata/get/<api_key>
get function metadata

api/metadata/scan/<api_key>
scan the entire binary

FIRST architecture

The DB includes more than 350k functions

OpenSSL, 7zip, aPLib, ucl, LibreSSL 2.3.1, Mimikatz, aPackage, UPX, Alina Spark, Dexter, Grum, Pony, Zeus, HackingTeam RCS

3 implemented engines: exact match, basic masking, mnemonic hash

1 experimental engine: Catalog1 from @xorpd

FIRST

Register to use:

https://first.talosintelligence.com/

Get the code:

https://github.com/vrtadmin/FIRST-plugin-ida

https://github.com/vrtadmin/FIRST-server

Read the docs:

https://first-plugin-ida.readthedocs.io/

Demo

Phobos malware:

`4c347d78da2c29cd84a298dd2a463c381bc13da95cdb9782c6bc65256eae1576`

GhIDA and Ghidraaas

GhIDA: Ghidra Decompiler for IDA Pro
https://github.com/Cisco-Talos/GhIDA

Ghidraaas: Ghidra analysis through REST APIs
https://github.com/Cisco-Talos/Ghidraaas

Ghidraaas

Docker with Ghidra installed and web-server with REST APIs

3 Ghidra plugins to analyze, list the functions and decompile

5 generic APIs and 3 GhIDA specific.

Ghidraaas

api/analyze_sample/
Submit a sample for the analysis

api/get_functions_list/<sha256>
Request the list of functions

api/get_functions_list_detailed/<sha256>
Request the list of functions with additional details

api/get_decompiled_function/<sha256>/<offset>
Request to decompile a function

api/analysis_terminated/<sha256>
Remove the *.gpr file and *.rep files.

GhIDA

IDA Pro 7.x plugin. Requires Python 2.7

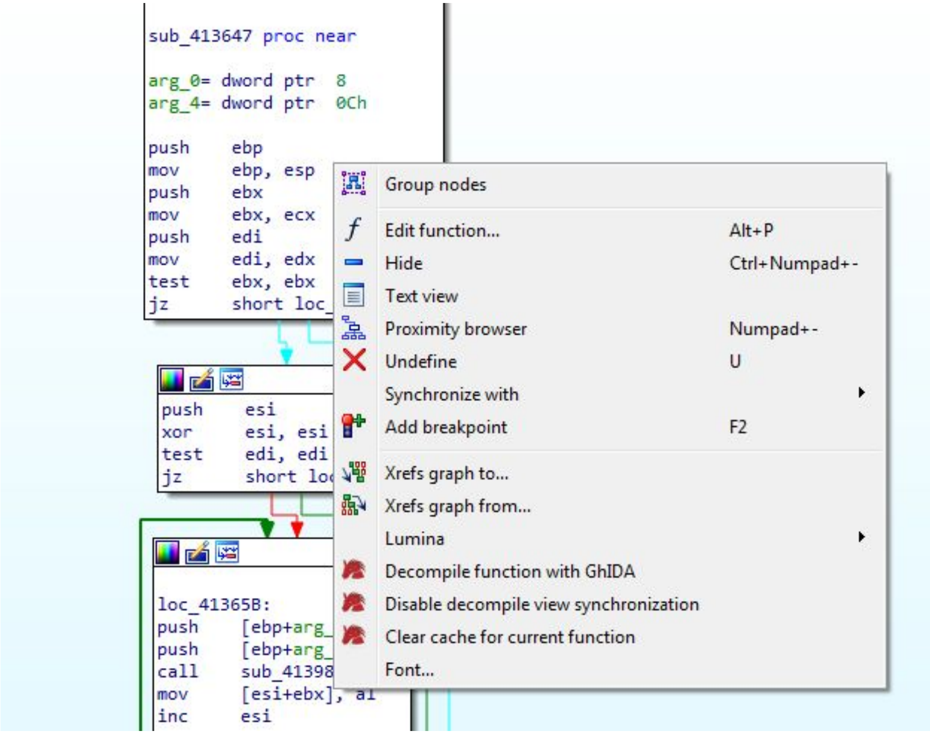
Exports an IDA Pro project in xml format, then calls Ghidra in headless mode

It works either a local installation of Ghidra or the Ghidraaas server

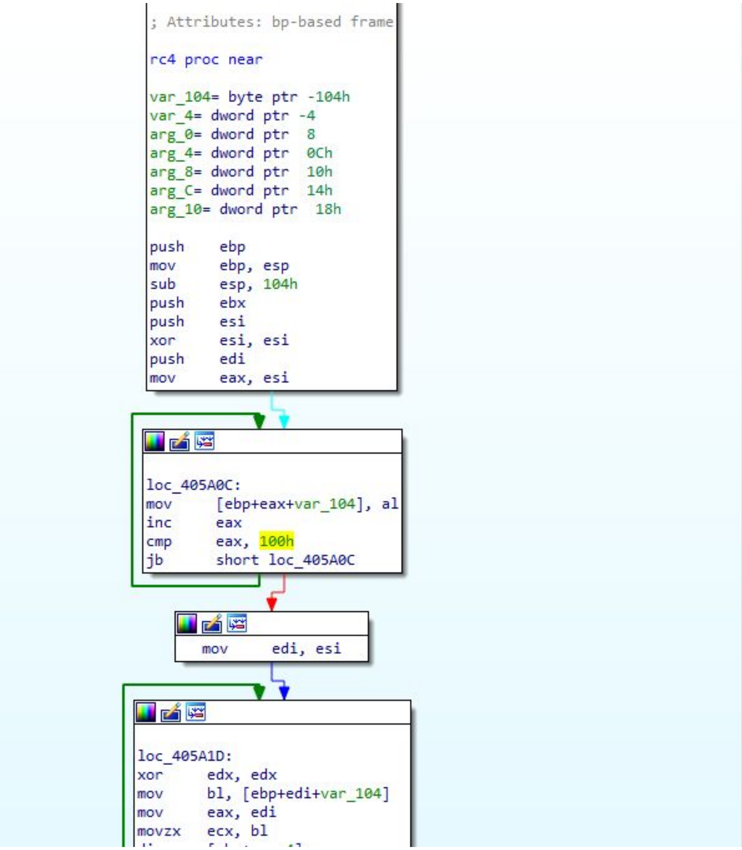
The plugin correctly handles x86 and x64 PE and ELF binaries.

There are other plugins that directly integrate the Ghidra decompiler
e.g., https://github.com/cseagle/blc

GhIDA



sub_413647 proc near
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
push ebp
mov ebp, esp
push ebx
mov ebx, ecx
push edi
mov edi, edx
test ebx, ebx
jz short loc_
push esi
xor esi, esi
test edi, edi
jz short lo
loc_41365B:
push [ebp+arg_
push [ebp+arg_
call sub_41398
mov [esi+ebx], al
inc esi
Group nodes
Edit function... Alt+P
Hide Ctrl+Numpad+-
Text view
Proximity browser Numpad+-
Undefine U
Synchronize with
Add breakpoint F2
Xrefs graph to...
Xrefs graph from...
Lumina
Decompile function with GhIDA
Disable decompile view synchronization
Clear cache for current function
Font...



; Attributes: bp-based frame
rc4 proc near
var_104= byte ptr -104h
var_4= dword ptr -4
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
arg_8= dword ptr 10h
arg_C= dword ptr 14h
arg_10= dword ptr 18h
push ebp
mov ebp, esp
sub esp, 104h
push ebx
push esi
xor esi, esi
push edi
mov eax, esi
loc_405A0C:
mov [ebp+eax+var_104], al
inc eax
cmp eax, 100h
jb short loc_405A0C
mov edi, esi
loc_405A1D:
xor edx, edx
mov bl, [ebp+edi+var_104]
mov eax, edi
movzx ecx, bl

byte * rc4(int param_1,uint param_2,int param_3,byte *param_4,byte *param_5)
{
byte bVar1;
uint uVar2;
int iVar3;
uint uVar4;
byte local_108;
dword local_8;
iVar3 = (int)param_4;
uVar4 = 0;
uVar2 = 0;
do { // Key-Scheduling Algorithm - Initialization
(&local_108)[uVar2] = (byte)uVar2;
uVar2 = uVar2 + 1;
} while (uVar2 < 0x100);
uVar2 = 0;
do {
bVar1 = (&local_108)[uVar2];
uVar4 = (uint)bVar1 + (uint)*(byte *)(uVar2 % param_2 + param_1) + uVar4 & 0xff;
(&local_108)[uVar2] = (&local_108)[uVar4];
uVar2 = uVar2 + 1;
(&local_108)[uVar4] = bVar1;
} while (uVar2 < 0x100);
uVar4 = 0;
uVar2 = 0;
if (param_4 != (byte *)0x0) {
param_4 = param_5;
do { // Stream Generation
uVar2 = uVar2 + 1 & 0xff;
bVar1 = (&local_108)[uVar2];
uVar4 = (uint)bVar1 + uVar4 & 0xff;
(&local_108)[uVar2] = (&local_108)[uVar4];
(&local_108)[uVar4] = bVar1;
*param_4 = (&local_108)[(uint)(byte)((&local_108)[uVar2] + bVar1)] ^
param_4[param_3 - (int)param_5];
param_4 = param_4 + 1;
iVar3 = iVar3 + -1;
} while (iVar3 != 0);
}
return param_5;
}

GhIDA’s features

Synchronization of the disassembler view with the decompiler view

Decompiled code syntax highlight

Code navigation by double-clicking on symbols' name

Add comments in the decompiler view

Symbols renaming (limited to XML exported symbols and few others)

Symbols highlight on disassembler and decompiler view

Decompiled code and comments cache

Store setting options

Demo

Phobos malware:

4c347d78da2c29cd84a298dd2a463c381bc13da95cdb9782c6bc65256eae1576

Part 11

Automatic signature generation

What is a malware signature?

A combination of patterns that indicate the presence of malicious code

As malware evolves, new signatures need to be generated frequently

Static signatures are based on unique sequences of instructions or strings
* this is where the most of the existing tools and researches focus on

Behavioural signatures provides an abstraction of the program behavior
In this context, malware “signatures” and “rules” have the same meaning.

About YARA and ClamAV

ClamAV and YARA are the most-used languages to write malware signatures

"YARA is to files what Snort is to network traffic" *Victor M. Alvarez*

They natively supports static signatures (strings + regex + hex)

YARA Signatures can be extended through custom modules
Similarly, ClamAV bytecode signatures supports complex matching logic.

Example of YARA signature

```
rule example {

   meta:
      author = "Andrea Marcelli"

   strings:
      $a = "IEncrypt.dll"

   condition:
       $a and
       pe.image_base == 708640768 and
       pe.resources[6].language == 1030 and
       pe.resources[36].type == 10 and
       pe.resources[37].id == 104 and
       pe.imports("user32.dll","GetCursorPos")
}
```

Examples of ClamAV signature

```
Sig1;Target:0;(0&1&2&3)&(4|1);6b6f74656b;616c61;7a6f6c77;7374656
6616e;deadbeef

Sig2;Target:0;((0|1|2)>5,2)&(3|1);6b6f74656b;616c61;7a6f6c77;737
46566616e

Sig3;Target:0;((0|1|2|3)=2)&(4|1);6b6f74656b;616c61;7a6f6c77;737
46566616e;deadbeef

Sig4;Engine:51-255,Target:1;((0|1)&(2|3))&4;EP+123:33c06834f04100
f2aef7d14951684cf04100e8110a00;S2+78:22??232c2d252229{-15}6e6573
(63|64)61706528;S3+50:68efa311c3b9963cb1ee8e586d32aeb9043e;f9c58
dcf43987e4f519d629b103375;SL+550:6300680065005c0046006900
```

Requirements

The process to generate a signature should be fast (e.g., ~ 5 min for 100 samples)

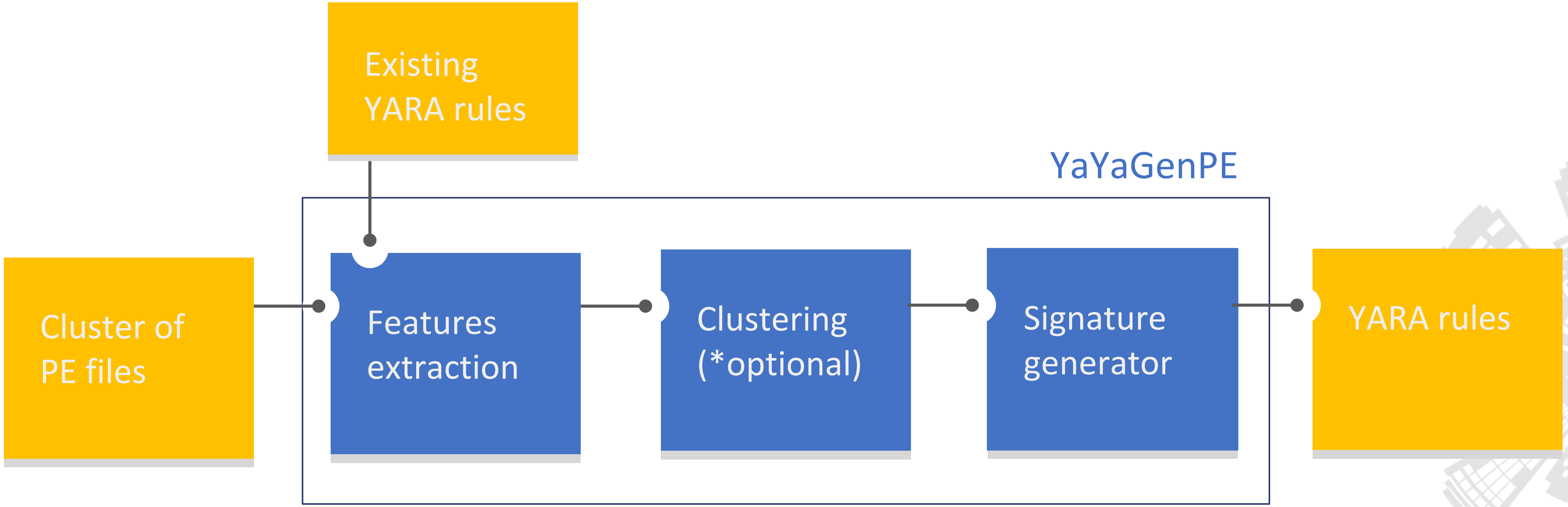
The algorithm should scale up to few thousands of input samples

Limit FPs

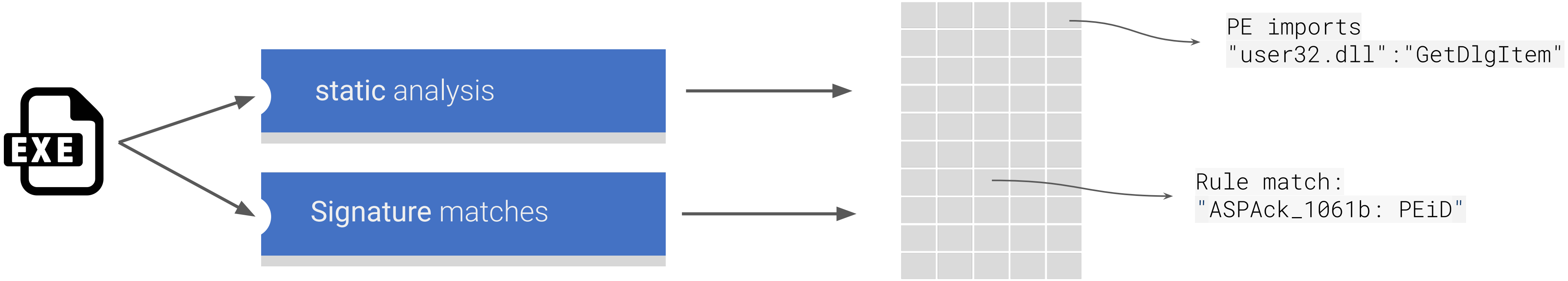
Avoiding FPs should not be related to number of samples input

The signature should catch other variants too.

The framework workflow



1. Features extraction



Each block is a feature extracted through the analysis, or another rule that matches the file

A custom YARA version extract all the supported features

Existing YARA rules add expert knowledge.

2. Clustering

It reduces the complexity of signature generation process
Allow the framework to scale with 1000+ inputs

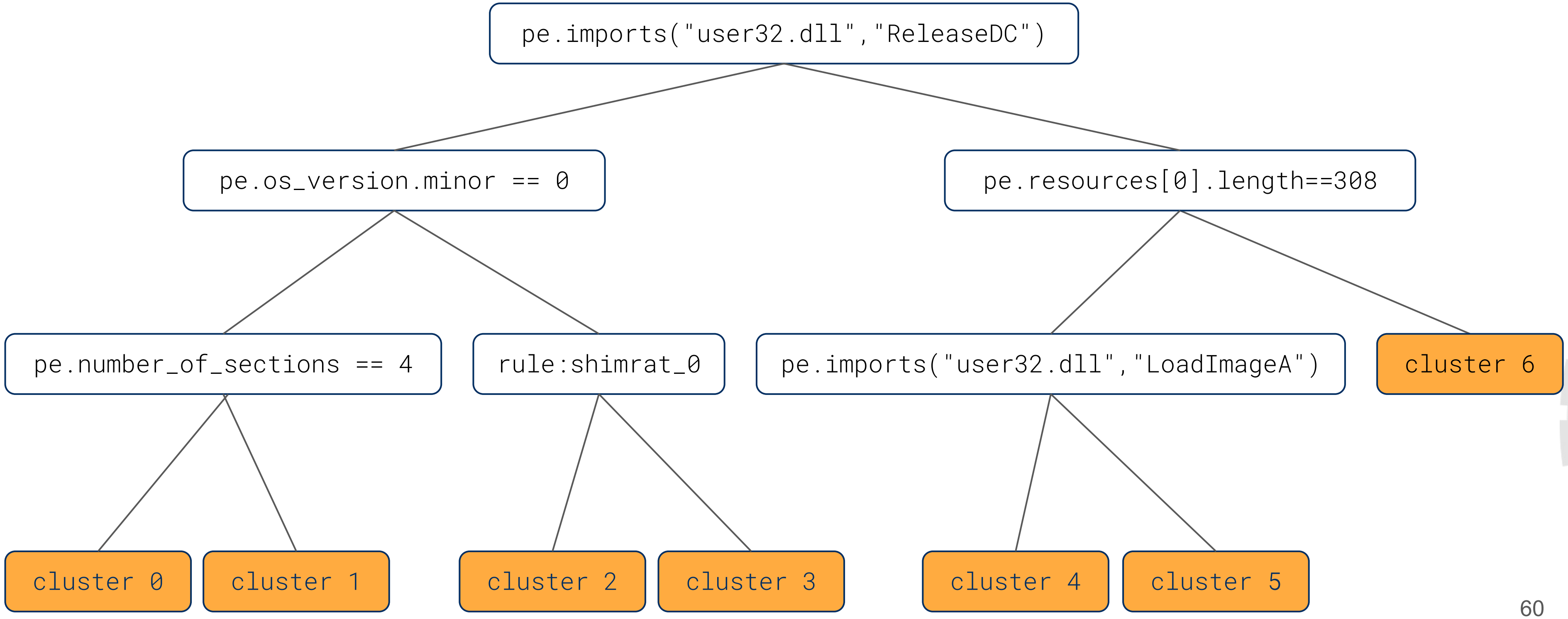
Each cluster is splitted based on the value of a single feature

The best best splitting feature is the one that maximise the distance among centroids
Cluster centroids are approximated, and Jaccard distances is used

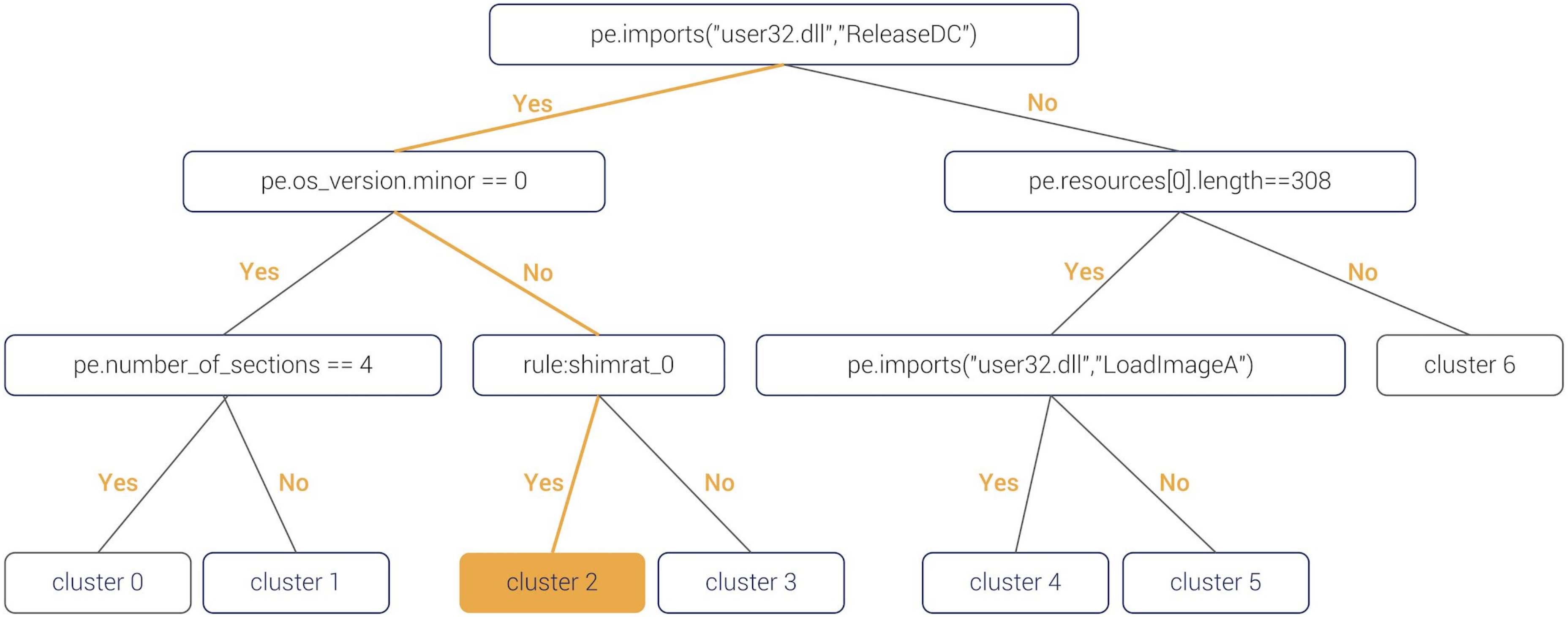
The stopping criterion is the distance between centroids (experimentally set)

The splitting feature can be easily added to the generated rules.

UDT clustering



UDT clustering



3. The signature generation

Finding the optimal attributes subsets is the goal of the signature generation process

The problem can be reduced to a variant of the set cover problem (NP-complete)

A dynamic greedy algorithm builds the signature as a disjunction of clauses.

Signature anatomy

$$\underbrace{(l_1 \wedge l_2 \wedge l_3)}_{\text{clause}} \vee (\underbrace{l_4}_{\text{literal}} \wedge l_5)$$

Each signature can be expressed in DNF

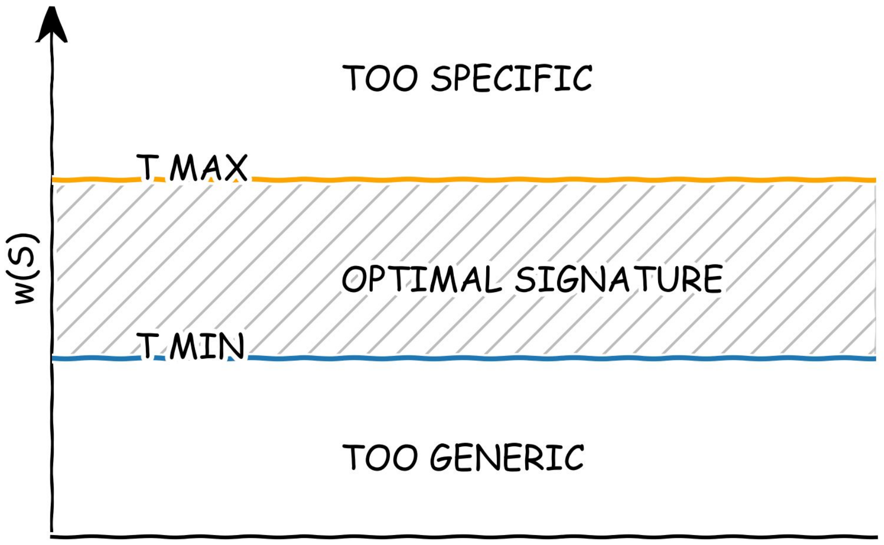
$$S = \bigvee_{i=0}^{n} c_i \quad c_i = \bigwedge_{j=0}^{m(i)} l_{i,j}$$

Each clause is a valid YARA rule
Each clause can be weighed.

The simplex method is used to assign the weights.

$$w(c_i) = \sum_{j=0}^{m(i)} w(l_{i,j})$$

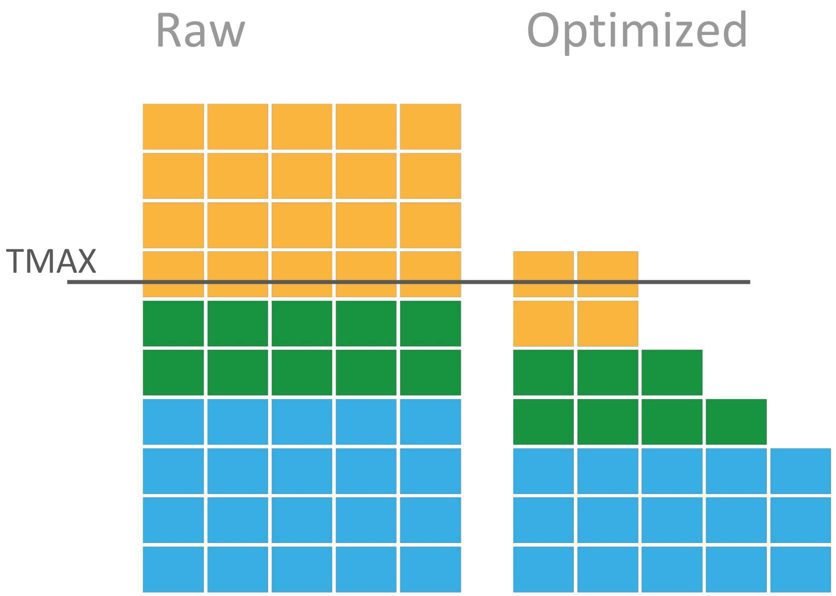
Scoring system



The weight of a signature is the lowest among its clauses

$$w(S) = \min_{\forall i} w(c_i)$$

Signature optimization



Rules could be over-specific

We need to study which combinations of attributes create a better rule

We introduced two optimizers: hill-climber- and EA-based.

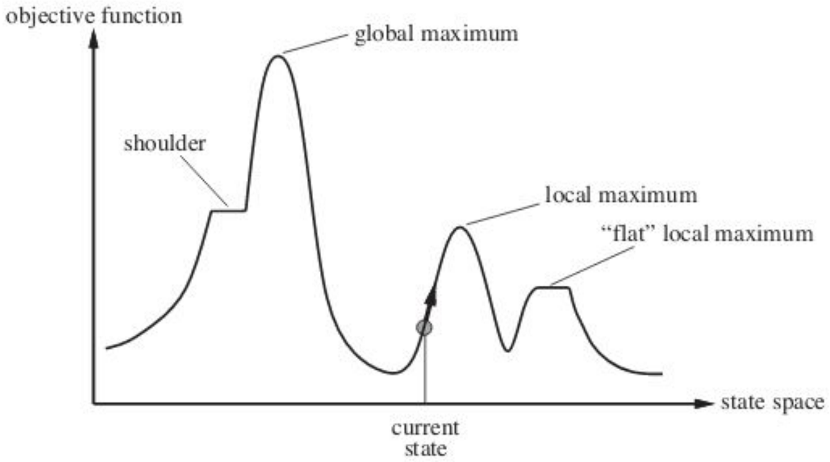
Hill-climber

It's a *local search* technique

It makes incremental changes until no better solutions can be found

For *non-convex* problems it will only find *local optima*

Variants include *Tabu search* or *Simulated annealing*.



Evolutionary algorithm

Solution representation:

- *the genome* is the signature to optimize
- *the loci* are the literals of the signature

Two individuals are compared based on:

- Num. of matches
- Heuristics
- Score of the rule
- Num. of attributes

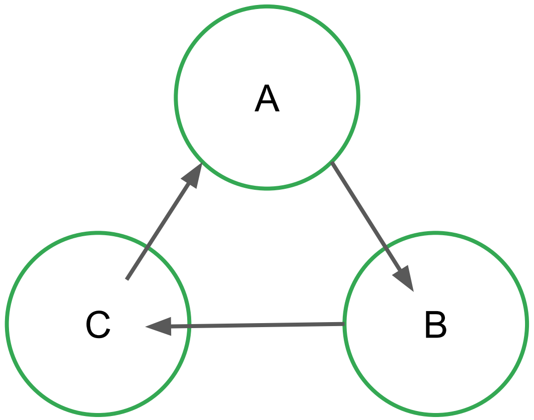
Heuristic fitness

Define your heuristics: e.g. *better more clauses of type x than y*

The comparisons are not “hard”: transitive property is lost

Archive comparison through tournament selection
(each pair, twice comparisons)

Best solution is stored in the archive.



Yet another YARA rule generator

*YaYa is grandma is ES

YaYaGenPE is an extension of the original YaYaGen framework

2 clustering algorithms (HDBSCAN, UDT)

2 algorithms for the rule generation (clot, greedy)

Include new YARA python bindings to directly extract the features.

Supports FP exclusion from rule generation

Written in Python 3.

YaYaGenPE

Fork me on GitHub:

https://github.com/jimmy-sonny/YaYaGen

Demo

Let’s create a signature for a malware family

Signature generation - References

Griffin, Kent, et al. "Automatic generation of string signatures for malware detection." *International workshop on recent advances in intrusion detection*. Springer, Berlin, Heidelberg, 2009.

Preda, Mila Dalla, et al. "A semantics-based approach to malware detection." ACM SIGPLAN Notices 42.1 (2007): 377-388.

Perdisci, Roberto, Wenke Lee, and Nick Feamster. "Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces." NSDI. Vol. 10. 2010.

https://github.com/Xen0ph0n/YaraGenerator

https://github.com/Neo23x0/yarGen

https://github.com/AlienVault-OTX/yabin

https://www.talosintelligence.com/bass

Thanks

anmarcel@cisco.com