# *CS7038 - Malware Analysis - Wk05.1*

# Static Analysis

Coleman Kane
kaneca@mail.uc.edu

February 6, 2018

## Static Analysis Model

In general, static analysis is often a *feature extraction* problem for the analyst. You may extract meaningful attributes/metadata about the program, a.k.a. *structured data*. Additionally, you may also be extracting features lacking meaning (or, moreover, who's meaning is as-yet uncertain). These often fall into the *unstructured metadata* class.

Some structured data examples we've discussed in class:

- Program start address
- PDF/Document author
- Compiler version
- Import libraries / functions

Some unstructured data examples we've discussed in class:

- Strings
- Bytecode sequences (*why?*)

## Unstructured data extraction

A common place to begin is to review `strings` stored in the executable. It is common to store data within an executable in a human-readable format. This utility can help separate this information from the binary content of the file, and emit the tokens one per line - assisting with additional processing.

It is also of note that, in many operating systems, especially Windows, it is common to use the UTF-16 string format (wide string), in addition to ASCII or UTF-8. You'll want to adjust use of `strings` to extract both sets:

- `strings my.exe > strings_ascii.txt`
- `strings -el my.exe > strings_utf16.txt`

## Yara

The **Yara** project is a relatively new system, introduced around 2009-2010, and has a large following of community support. This community in Yara is focused considerably more on using it for the purposes of malware research and analysis, and less focused on malware detection and alerting. However, it works well for both!


UNIVERSITY OF Cincinnati

## Yara Rules

Yara is centered around a custom signature-building language. The syntax provided by Yara is considerably readable, and versatile. The side-effect of this functionality is significant per-rule overhead - leading many users to focus on its use in malware analysis, where delayed results aren't as impactful.

Documentation: `http://yara.readthedocs.io/en/v3.7.1/index.html`

```
rule evil_pdf_rule {
 meta:
  author = "Coleman Kane"
  revision = 12
  description = "Detect evil.pdf sample"

 strings:
  $a = "\"Do not show this message again\"" nocase
  $r = /if exist.*template\.pdf/
  $b = { 706c65617365207469636b207468652022446f206e6f74 }
  $pt1 = "start " nocase
  $pt2 = "cd " nocase
  $pt3 = "exist " nocase
  $pt4 = "cmd.exe" nocase

 condition:
  $a or $b or $r or 2 of ($pt*)
}
```

## Yara Modules

There are also a number of modules for Yara that enable it to combine the language used for unstructured data matching with structured data values, to enable more capabilities.

`http://yara.readthedocs.io/en/v3.7.1/modules.html`

- `pe` - Windows PE32/32+ metadata
- `elf` - ELF (Linux and other UNIX executable) metadata
- `cuckoo` - Utilize an external dynamic-execution report format in addition to file content
- `magic` - File type auto-detection logic
- `math` - Facilitate complex mathematical analysis on content/values
- `dotnet` - EXE metadata for .NET/C# programs
- `time` - Utilize time and temporal conditions

# Yara Rules With Modules

```
import pe /* Import module */

rule evil_exe_rule {
 meta:
  author = "Coleman Kane"
  description = "Detect evil.exe"

 strings:
  $a = "Connecting to %d .. ." nocase
  $b = { 706c6561736520746963207468652022446f206e6f74 }

 condition:
  $a and $b and pe.imports("kernel32.dll", "WriteProcessMemory"
}
```

## LibYara C API

In addition to its functionality as a command-line scanning tool like `vscan` and `clamscan`, the real power inherent in Yara is that it is a C library at its core, and it can be incorporated to extend other projects.

Documentation for this C API is here: `http://yara.readthedocs.io/en/v3.7.1/capi.html`

At a minimum, initialization consists of the following calls:

- `yr_initialize()` - Initialize library
- `yr_compiler_create(YR_COMPILER**)` - Create new compiler
- `yr_compiler_add_file(YR_COMPILER*, FILE*, NULL, char*)` - read a file and compile the rules it contains (*multiple calls possible*)
- `yr_compiler_get_rules(YR_COMPILER*, YR_RULES**)` - Load a pointer to now-compiled rules into local program scope

# LibYara C API (Scanning)

Scanning involves creating your own custom *callback function* and then executing one or more of the `yr_rules_scan_*` functions, iterating if needed.

The scanner will run until it has exhaustively searched the buffer or file you provided, and will call your *callback function* one or more times during the run, providing an opportunity for your custom code to react to the scanner findings.

When a rule hit occurs, you will receive a pointer to the following data structure in the `void *message_data` parameter:

```
struct {
  const char *identifier; /* "no_rule" / "yes_rule"  for us */
  const char *tags; /* Rule-defined tags */
  YR_META *metas;   /* Rule-defined metadata key=value pairs */
  YR_STRING *strings; /* strings from rule */
};
```


UNIVERSITY OF Cincinnati

## LibYara Python API

Additionally, another helpful feature of yara is that there's a really useful Python module for it. In most cases, you can use `pip` or `pip3` to install it:
`pip install yara-python`

Unlike the C API, the Python context can be instantiated rather quickly:
`rules = yara.compile(filepath="yara_rules.yar")`

And then, scanning is implemented in a manner very similar to the popular `re`(`https://docs.python.org/3/library/re.html`) module:
`matches = rules.match(data=input_data)`