Global Information Assurance Certification Paper

Intelligence-Driven Incident Response with YARA

GIAC (GCFA) Gold Certification

Author: Ricardo Dias, ricardo.dias@live.com
Advisor: Richard Carbone

Accepted: October 4, 2014

Abstract

Given the current cyber threat landscape, organizations are now beginning to acknowledge the inexorable law that decrees that they will be compromised. Threat actors' tactics, techniques, and procedures demand intelligence-driven incident response, which in turn, depend upon methodologies capable of yielding actionable threat intelligence in order to adapt to each threat. The process to develop such intelligence is already in motion, heavily relying on behavioral analysis, and has given birth to cyber threat indicators as a means of fingerprinting and thus identifying new and unknown threats. This paper will focus on YARA, a malware identification and classification tool used as a scan engine, whose features will be explored in order to deploy indicators at the endpoint.

"It is even better to act quickly and err than to hesitate until the time of action is past."

Carl von Clausewitz: On War, 1832

1. Introduction

The concept of threat intelligence is gaining momentum in the cyber-security arena. As targeted attacks increase in number and sophistication, organizations are beginning to develop and integrate the concept of threat intelligence into their cyber-defensive strategies. By doing so, organizations are taking the next step forward to respond to cyber-attacks. Recent threat reports reveal promising results. In 2012, Mandiant stated that, on average, attackers remain undetectable for 416 days (Mantiant, 2012). In 2013, one year later, the same vendor revealed a significant reduction to 243 days (Mandiant, 2013). While the number is still daunting, the trend denotes an improvement in the way organizations respond to incidents. Still, in Forrester *Planning for Failure*, the incident response process is described as one of the most overlooked areas in information security. The report shows that only 18% of breached organizations increased spending on incident response programs. Notably, 43% invested in additional security and audit requirements, and 30% stated that nothing had changed in the past 12 months because of security breaches. Breached organizations with an inadequate incident response plan are often criticized for failing to notify the public in a timely fashion or for minimizing its effects, and ultimately will face financial, operational, and reputational losses (Holland, 2014).

Sophisticated attacks, commonly known as advanced persistent threats (APT), often involve targeted, multi-stage attacks with specific tactical objectives. Another characteristic of this class of attacks is the ability of the adversary to establish a foothold in the organization in order to facilitate data exfiltration and lateral movement (Barnum, 2014). The level of sophistication of APTs clearly denotes threat actor's maturity concerning its tactics, techniques, and procedures (TTP) when engaging in a cyber-attack. It is imperative that organizations facing APTs proceed with an in-depth intrusion analysis during incident response. The incident response plan should cover memory, timeline and file system forensic analysis. Also known as deep dive forensics, the main

Ricardo Dias;ricardo.dias@live.com

goals are to determine how the intrusion occurred and reveal indicators that map with the attack progression (SANS, 2013).

The suggestion of mapping attack progression in order to derive intelligence for leveraging cyber-defenses is not new. Specifically, Eric Hutchins, Michael Cloppert, and Rohan Amin, in *Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains*, proposed an Intelligence-Driven strategy in order to leverage advantage over APTs. In essence, the proposed strategy encompasses a phase-based model, also known as the intrusion kill chain, composed of cyber threat indicators directly related to APT activity. By understanding threat actors' TTP, organizations that choose to follow the intrusion kill chain will be able to apply strategic courses of action for stopping APT progression (Hutchins, Cloppert, & Amin, 2011).

When organizations adopt cyber threat intelligence, they do it using standards and tools. Standards enable intelligence to be exchanged in a structured manner across sharing partners. This is important since one organization alone is not capable of determining the full scope of sophisticated attacks. In Forrester's *Five Steps To Build An Effective Threat Intelligence Capability*, intelligence sharing is pointed to as a relevant aspect of the cyber threat intelligence strategy (Holland, 2014). A notable example of a standard for structuring threat intelligence is MITRE's Structured Threat Information eXpression (STIX). STIX is an XML language "for the specification, capture, characterization, and communication of standardized cyber threat information" (Barnum, 2014). Moreover, STIX features the intrusion kill chain and supports foreign signature formats like YARA and SNORT.

YARA is an open source tool aimed to help malware researchers identify and classify malware samples. With YARA, you can create descriptions of malware families based on textual or binary patterns. Each description consists of a set of strings and a Boolean expression that determine its logic (Álvarez, 2014). Along with a flexible, and yet simple rule syntax, the YARA engine scanning capabilities include: file objects, processes, and external variables. Furthermore, it is also possible to develop custom modules thus extending the engine's capabilities. Widely adopted by malware analysts,

honeypots, and sandbox technologies, it is also featured in innovative cyber-security solutions. By integrating YARA signatures with STIX, organizations with a mature threat intelligence plan are able not only to share YARA based indicators but also to scan their endpoints for traces of intrusion.

1.1. Cyber Threat Indicators

Indicators are the elements of intelligence in the kill chain model. They are any piece of information that relates with the intrusion and can be classified as Atomic, Computed, and Behavioral (Hutchins, Cloppert, & Amin, 2011).

Atomic indicators are small pieces of data specific to the intrusion. Examples include IP and email addresses, static strings in a covert Command-and-Control (C2) channel or fully qualified domain names. Computed indicators are those that are in some fashion computed. The most common amongst these are malicious file hashes, but can also include specific data in decoded custom C2 protocols. Behavioral indicators are those that combine other indicators — including other behaviors to form a profile. To borrow some parlance, these are also referred to as TTPs.

1.2. Intrusion Kill Chain

The intrusion kill chain is a phase-based model for actionable intelligence. This model defines the seven phases of the APT intrusion in reconnaissance, weaponization, delivery, exploitation, installation, command and control, and actions on objectives. "The essence of an intrusion is that the aggressor must develop a payload to breach a trusted boundary, establish a presence inside a trusted environment, and from that presence, take actions toward their objectives. Actions include laterally moving inside the environment violating the confidentiality, integrity, or availability of a system in the environment" (Hutchins, Cloppert, & Amin, 2011). In order to achieve actionable intelligence, technology capable of ingesting indicators must be aligned with the enterprise's defensive capabilities. Technology spans from host to network based, which include intrusion detection and packet inspection, respectively. If the kill chain analysis successfully detects an intrusion, the analyst will be able to determinate its progression and move forward with its analysis. Late detection presumes a successful intrusion and thus requires deeper analysis of the previous intrusion phases in order to detect the APT

sooner. In turn, early detection denotes resilience and allows the analyst to synthetize the subsequent stages of the attack. Figure 1 illustrates the phases of the intrusion kill chain, technology, and actions for the detection stages:

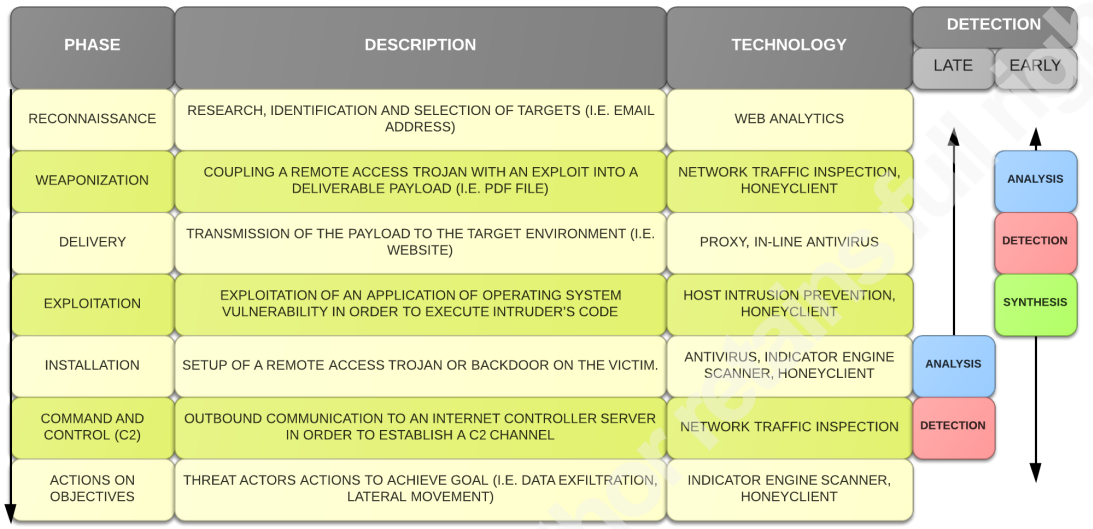


Figure 1 - Kill Chain Process (source: Hutchins et al. 2010)

Attack synthesis is a key step as it allows the analyst to study the malicious code, extrapolate the attack progression, and release additional indicators. Honey-clients and malware sandboxes that support indicators play a significant role in this step, as they are able to interact and identify malicious code activity. They allow the analyst to repeatedly test and fine-tune indicators. Some notable open source projects are Thug and Cuckoo Sandbox.

Thug "is a Python low-interaction honey client aimed at mimicking the behavior of a web browser in order to detect and emulate malicious contents" (Dell'Aera, 2014). It supports YARA based indicators and is particularly efficient when dealing with exploit kits. If detection happens upon a malicious website at the delivery stage, the analyst will be able to run Thug against the website and retrieve malicious content that can reveal additional indicators. Cuckoo Sandbox is an automated malware analysis system that uses virtual machines to analyze malicious files and websites (Guarnieri, 2014). It also supports YARA based indicators. If detection happens upon a malicious email attachment at the delivery phase, the analyst will be able to submit the file to Cuckoo Sandbox and analyze its results to reveal additional indicators.

Ricardo Dias;ricardo.dias@live.com

2. Scope and Assumptions

The research herein explores YARA as a cyber threat indicator scanner for the enterprise. While YARA is best known as a file based scanning tool, this research will introduce its features and how the tool can be leveraged in order to integrate a cyber-threat intelligence platform. This approach will allow the analyst to develop cyber threat indicators, according to the kill chain, which will be organized and distributed across the enterprise.

Additionally, the research goal is to address the topic of cyber threat indicator scanning at the endpoint. While reports on cyber threat intelligence are abundant, they usually approach the concept at a higher level. This results in scarce documentation and resources to help organizations make use of their indicators.

2.1. Structure

The first part of the research consists of introducing the YARA rule syntax and scan engine. The second part demonstrates how to integrate the scan engine in the incident response process. This is necessary in order to define which steps of the incident response process are more suitable to accommodate both YARA scans and rule creation. The third part covers a selection of core features that render the scan engine ready for host scanning while featuring the kill chain model. The fourth part introduces extended features by leveraging the scan engine within scripting languages that enable input and output control of data flowing in and out of YARA. The research culminates at the fifth part with the creation of a distributable package using McAfee ePolicy Orchestrator in order to make use of the scan engine across the enterprise.

2.2. Proof of concept code

The author has developed Python code, released in this paper, as a proof of concept to support the models presented later in the paper. Albeit the code was tested in order to ensure its quality and stability, as in any other proof of concept, it should be revised and tuned before moving it into production. Therefore, Python proficiency is expected when attempting to interpret and execute the code herein.

Ricardo Dias;ricardo.dias@live.com

3. The YARA tool

3.1. Rule Syntax

In plain text format, the rule's syntax resembles a C-like struct declaration and is composed of *metadata*, *strings* and *condition* sections. Each rule begins with the keyword 'rule' followed by the rule identifier, which must be unique. Additionally, following the rule identifier, rule tags can be defined. The metadata section contains descriptive information about the rule. Defined with the keyword 'meta', it contains identifier/value pairs. The strings section is where code sequences for pattern matching are stated. Under the section defined with the keyword 'strings', each string has an identifier consisting of a $ character followed by a sequence of alphanumeric characters and underscores. YARA supports hexadecimal strings, text strings, and regular expressions. The condition section contains the Boolean expression that defines the rule logic. It supports typical operators like 'and', 'or', 'not' and relational operators '>=', '<=', '<', '>', '==' and '!=' (Álvarez, 2014). Figure 2 depicts a common YARA rule:

```
rule my_rule_id: myTag
{
  meta:
    desc = "description"
  strings:
    $str01 = "psexec" nocase     //nocase = case insensitive
    $str02 = "procmon" nocase
  condition:
    $str01 or $str02
}
```

Figure 2 - Common YARA rule: strings.yar

3.2. Scan Engine

YARA scan engine is available for Windows, MacOS X and Linux. Typically, the engine is invoked via command line or via Python with the YARA-Python extension. In its simplest operation, the engine parses a file object or a process' memory against a signature file with one or more rules in the YARA format. Rule matches are sent directly to standard output, meaning the console where the script was invoked. In addition, YARA is also featured in a wide range of cutting-edge cyber security vendors including VirusTotal, CrowdStrike, Fidelis, FireEye, Blue Coat and RSA. This accounts for the engine's maturity and stability.

Ricardo Dias;ricardo.dias@live.com

Figure 3 depicts the execution of YARA using the rule strings.yar (see Figure 2):

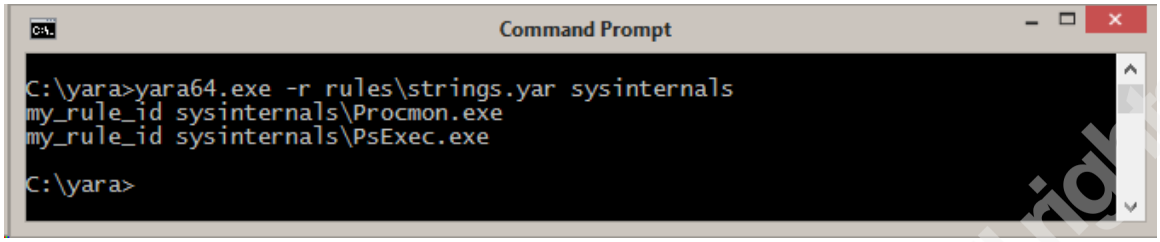


Figure 3 - YARA scan execution

While the scan engine is very lightweight, with roughly 150KB, it is highly versatile (Nemček, 2013). As of version 3.0, YARA began supporting external modules for advanced parsing capabilities. Two modules that ship by default are detailed in Section 3.4.

Naturally, the simplicity inherent in YARA poses some limitations. Perhaps the most notable one is the lack of output to a file. Although this may be irrelevant during intrusion analysis, it is cumbersome when scans are required to run simultaneously on multiple hosts. Section 3.5.1 details how to extend the scan engine in order to overcome this limitation.

3.3. Intelligence-driven Incident Response

The inevitable path of the incident response process towards an intelligence-driven strategy introduced in Section 1, naturally entails modifications within the process itself. The conventional incident response process-based framework includes six phases: Preparation, Identification, Containment, Eradication, Recovery and Follow-up (SANS, 2013).

In the Preparation phase, the incident response team plans how to react to incidents. Some key tasks involve defining a strategy, team managing, creating documentation, and training (Newman, 2007). At heart, this is a key phase of incident response as it impacts upon all subsequent phases. This phase embraces the kill chain particularly well because of its planning element. The incident response team ingests threat intelligence from private communities, feeds and threat reports in order to reveal new indicators for the organization. Here, YARA plays an important role as it provides the scan engine and its flexible rule syntax helps to accommodate these indicators.

Ricardo Dias;ricardo.dias@live.com

The Identification phase begins when the incident is detected. One of the key tasks is proper incident scoping in order to identify all affected systems. If the kill chain model is in use and indicators detected the attack, the incident response team will be able to determine the attack progression. If detection happens at an early phase of the kill chain and results in attack disruption, the incident response team will synthetize the attack as described in Section 1. Likewise, the YARA scan engine will be used to conduct an enterprise wide scan in order to determine the full scope of affected systems. This phase benefits if a CTI framework is already in place, where indicators are organized and ready to be used.

The Containment phase includes the intrusion analysis of the attack that originated the incident and intelligence gathering resulting from forensic analysis (SANS, 2013). It directly maps with the Analysis phase in the kill chain model, and indicators should be revealed. The reverse engineering team will ingest all suspicious unknown code revealed during forensic analysis to reveal new indicators. If new indicators are revealed or existing ones are fine-tuned, the incident response team will return to the Identification phase and conduct an enterprise wide scan.

The Eradication phase, also known as Remediation, includes the intrusion clean-up tasks. Remediation actions include the blocking of malicious IP addresses, enterprise password changes, and verification of remediation activities (SANS, 2013). This phase relies on neither the kill chain model nor the YARA scan engine.

The Recovery phase includes the reestablishment of affected systems back into the organization. It also involves the revision of key items like patch-management, awareness training, and password policy in order to improve organizational resilience against future attacks. Similar to Eradication, this phase does not feature the YARA scan engine.

The Follow-up phase, also known as ‘Lessons Learned’, marks the end of the incident response process. It ensures that the incident has been mitigated, the attacker removed, and additional counter measures are being implemented (SANS, 2013). This phase takes particular advantage of the kill chain model, as all revealed indicators throughout the process will be imported into technology that features the YARA scan

engine. Additional enterprise-wide sweeps with the YARA scan engine will also help to ensure that the threat was removed.

By integrating the intrusion kill chain model within the process the incident response team will not only make use of indicators as actionable intelligence but also contribute to the maturity of the kill chain model by revealing additional indicators. This forces Threat Actors to shift tactics thus making their campaign more expensive and less appealing.

Figure 4 illustrates the incident response process when integrated with YARA:

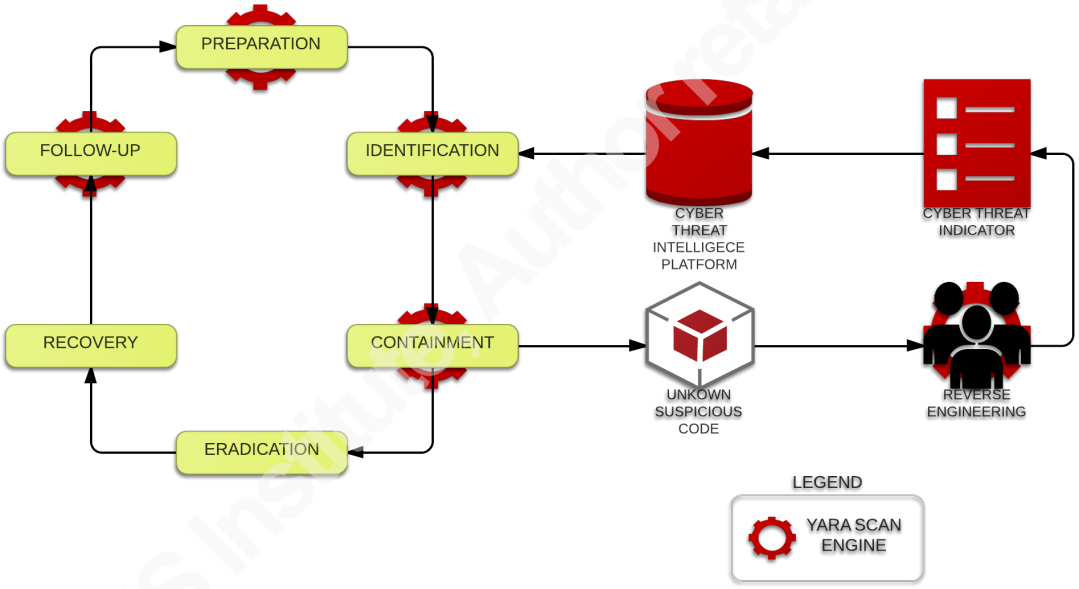


Figure 4 - YARA in the Incident Response Process

Ricardo Dias;ricardo.dias@live.com

3.4. Core Features

Natively, YARA offers a wide range of features that extend beyond simple file object parsing. A description of the most relevant features follows.

3.4.1. Portable executable file parsing

Introduced in version 3.0, as a custom module, YARA now is capable of parsing portable executable (PE) files. This enables the creation of fine-grained rules for PE header features and characteristics. Figure 5 illustrates a rule featuring the PE module by matching the Machine field:

```
import "PE"

rule pei386
{
    condition:
        pe.machine == pe.MACHINE_I386
}
```

Figure 5 – Rule featuring YARA PE module

The rule above will match all PE files designed to run on i386 architectures. This feature greatly contributes to rule simplicity and readability. Additionally, this module provides access to most fields of the PE header including the import and export functions. Combined rules featuring both string and PE module are supported. Figure 6 illustrates a combined rule:

```
import "PE"

rule suspicious_pe
{
    Strings:
        $str = "procmon"
    condition:
        pe.imports( "kernel32.dll", "CreateProcess" ) and
        pe.imports( "wininet.dll", "HttpSendRequest" ) and
        $str
}
```

Figure 6 - Combined YARA rule

The above rule will match when PE header imports, functions, and string conditions are satisfied. This greatly contributes to the extension of rule flexibility. Also, it renders the rule less prone to false positives. The PE module is a notable example of the customization supported by the YARA scan engine.

Ricardo Dias;ricardo.dias@live.com

3.4.2. Process scanning

The process scanning feature allows YARA to scan a process' entire memory, which in turn renders the engine immune to file obfuscation techniques. This feature enables YARA rules created during memory forensic analysis to reveal indicators that integrate into the install phase of the intrusion kill chain. Figure 7 depicts YARA process scanning:

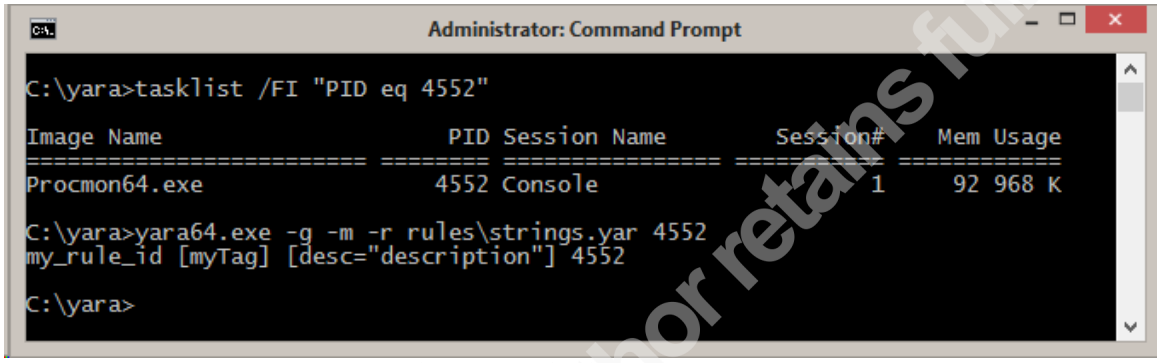


Figure 7 – YARA memory scan

Rules for memory analysis are common among the users of the Volatility framework. The Volatility 'yarascan' plugin scans the process address space against YARA rules. Process mutexes are frequently used to identify malware families. A notable example is the Zeus Banking Trojan, which was known for creating '_AVIRA_' or '__SYSTEM__' as mutexes (Ligh, Adair, Hartstein, & Richard , 2010). Figure 8 illustrates a sample Zeus mutexes rule:

```
rule zeus_mutex
{
    strings:
        $mtx01 = "_AVIRA_"
        $mtx02 = "__SYSTEM__"
    condition:
        $mtx01 or $mtx02
}
```

Figure 8 – YARA Zeus Mutex Rule

During intrusion analysis, memory forensic analysis can provide indicators used to leverage YARA rules for enterprise wide scans. However, it should be noted that YARA does not provide a method to scan all running processes. For that reason, a single

process ID must be always passed as an argument to the engine. Section 3.5.2 provides details extending the scan engine in order to overcome this limitation.

3.4.3. Rule metadata and tags

Perhaps one of the most overlooked features of YARA is the metadata section of the rule. This section is where the analyst describes a variety of details about the rule. Although the metadata section does not affect the rule's logic, it can be used for post-processing tasks. Metadata identifiers can be of type string, integer, or Boolean (true or false values). Additionally, YARA rules also support tags. Rule tags can be used in post-processing tasks, like output filtering, and rule management. Figure 9 depicts a rule featuring the metadata section while Figure 10 depicts the scan engine output:

```
rule RULE_META: systools
{
    meta:
        description = "Found Sysinternals Autorunsc"
        version = 1
        block = true
    strings:
        $str = "autoruns.pdb"
    condition:
        $str

}
```

Figure 9 – YARA Rule featuring metadata and tags

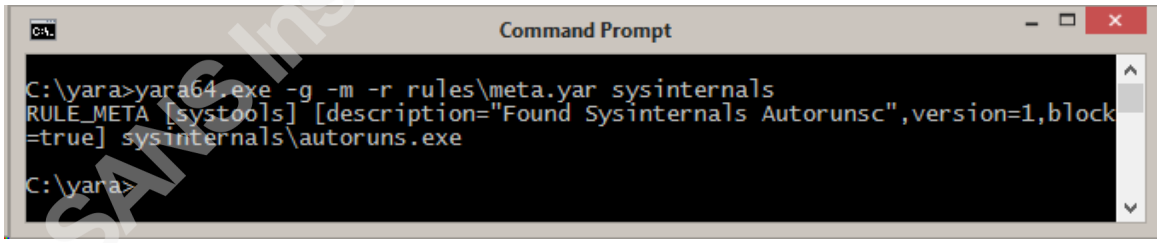


Figure 10 – YARA output featuring metadata

As depicted in Figure 10, the '-m' switch instructs the scan engine to include metadata information in the output, whereas '-g' displays rule tags in the output. The description field is important because it renders the rule human readable if the scan engine finds a match. The version field integer helps to identify the rule version. The block Boolean field in this example instructs post-processing frameworks that this rule shall not result in a blocking action. For example, if this rule were matched against an

email attachment the action would be to block the email. Primarily, the metadata section allows the analyst to create as many fields as needed according to post-processing requirements.

3.4.4. Rule variables

External variables enable the creation of rules that receive values from the outside. This feature is particularly useful when creating rules that rely on computed indicators, such as a file hash. Figure 11 depicts a rule featuring the external variable in the condition section while Figure 12 depicts a scan engine execution and its output:

```
rule my_rule_id
{
    strings:
        $str01 = "procmon" nocase
    condition:
        $str01 and
        proc matches /^procmon$/
}
```

Figure 11 – YARA rule featuring external variables

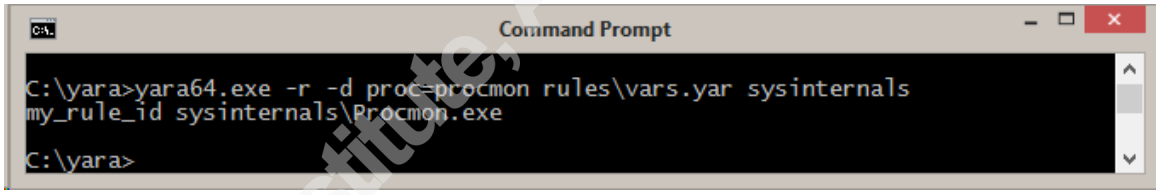


Figure 12 – YARA external variable declaration

The use of external variables is specified by the argument "-d" and is limited to certain operators. For example, external variables of type string only accept 'matches' or 'contains' operators. While the 'contains' operator returns true if the provided string contains the specified substring, the 'matches' operator returns true if the provided string matches a regular expression. Since the scan engine does not support the 'equal' operator for string based variables, the workaround would be to use the 'matches' operator featuring regular expression delimiters with '^' for the beginning of a string and '$' for the end of a string.

The use of external operators is extremely powerful since it allows the external injection of variables in order to influence the rule match. Particularly, when executing YARA within the context of scripting languages, one will be able to fetch values from

various sources and use them when invoking the scan engine. Section 13.2 demonstrates how the hash function 'imphash', available in the PEFILE module, can be passed to YARA in order to cluster malware analysis.

3.4.5. Rule referencing

YARA allows the creation of rules that reference other rules. This way a rule matching condition will depend on references rules. Specifically, rule referencing enables the creation of behavior-based signatures, thereby containing both atomic and computed indicators. Figure 13 depicts a rule referencing in the condition section while Figure 14 depicts a scan engine execution and output:

```
rule RULE_EVILMD5
{
    meta:
        description = "Suspicious MD5 Hash"
    condition:
        md5 matches /^3bb34a700e8d21acfdfe0f09208a7c01$/
}

rule RULE_EVILPATH
{
    meta:
        description = "Suspicious File Path"
    condition:
        path contains "/appdata/roaming/"
}
rule RULE_SUSP_BHV
{
    meta:
        description = "Suspicious Behavior"
    condition:
        RULE_EVILMD5 and RULE_EVILPATH
}
```

Figure 13 – YARA rule featuring rule referencing

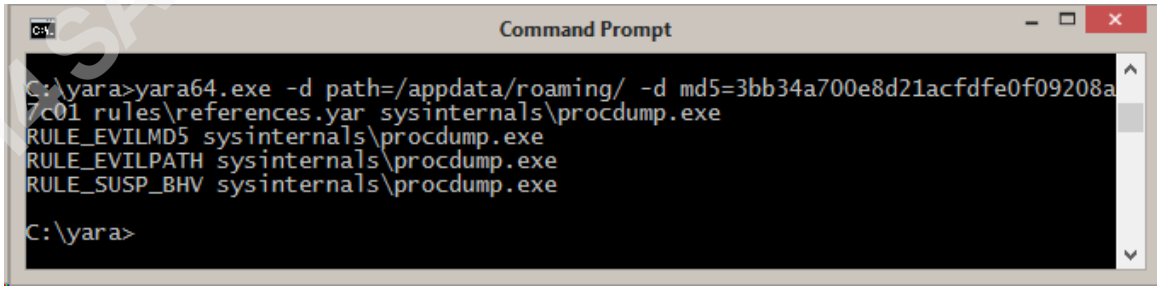


Figure 14 – YARA output

The above example demonstrates that both the parent rule 'RULE_SUSP_BHV' and the referenced rules are reported upon a match. Still, if the rule is too complex,

because it contains multiple referenced rules, this behavior can easily fill the output with superfluous information, overshadowing the parent rule. This is where private and global rules come in.

3.4.6. Private and global rules

YARA private rules do not report any output upon a match. This feature solves the problem identified in the previous section, as referenced rules can be defined as private in order to output only the main rule. Global rules affect all other rules in the same rule file, without the need to be referenced within every rule. Figure 15 depicts a rule featuring private referencing in the condition section while Figure 16 depicts such a scan engine execution and output.

```
private rule RULE_EVILMD5
{
    meta:
        description = "Suspicious MD5 Hash"
    condition:
        md5 matches /^3bb34a700e8d21acfdfe0f09208a7c01$/
}

private rule RULE_EVILPATH
{
    meta:
        description = "Suspicious File Path"
    condition:
        path contains "/appdata/roaming/"
}

rule RULE_SUSP_BHV
{
    meta:
        description = "Suspicious Behavior"
    condition:
        RULE_EVILMD5 and RULE_EVILPATH
}
```

Figure 15 – YARA rule referencing private rules

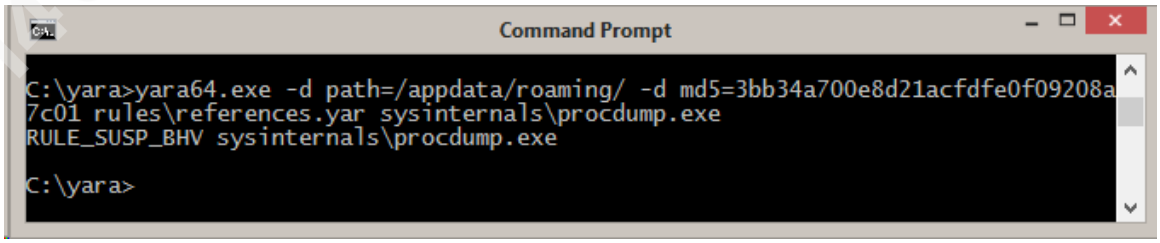


Figure 16 – YARA output

In the above example, private rules are omitted from the output. Compared with the output depicted in Figure 14, which lacked private rules, the output is no longer cluttered by the referenced rules.

3.4.7. Rule compilation and privacy

Before YARA begins parsing streams of data it must first compile the defined working set of rules. When dealing with repositories consisting of thousands of rules, precompiled rules will greatly enhance overall scan time. Rule compilation occurs by two methods. The first is upon conventional YARA scan engine execution; the engine parses the file and stores a compiled version in memory. The second method is by evocation of the YARAC binary which ingests clear text rules and outputs them in a binary object format ready to be consumed by the scan engine. The latter not only offers benefits regarding scanning speed, but also keeps the rule structure hidden from prying eyes. Figure 17 illustrates how to compile a rule by using the 'yarac' executable:

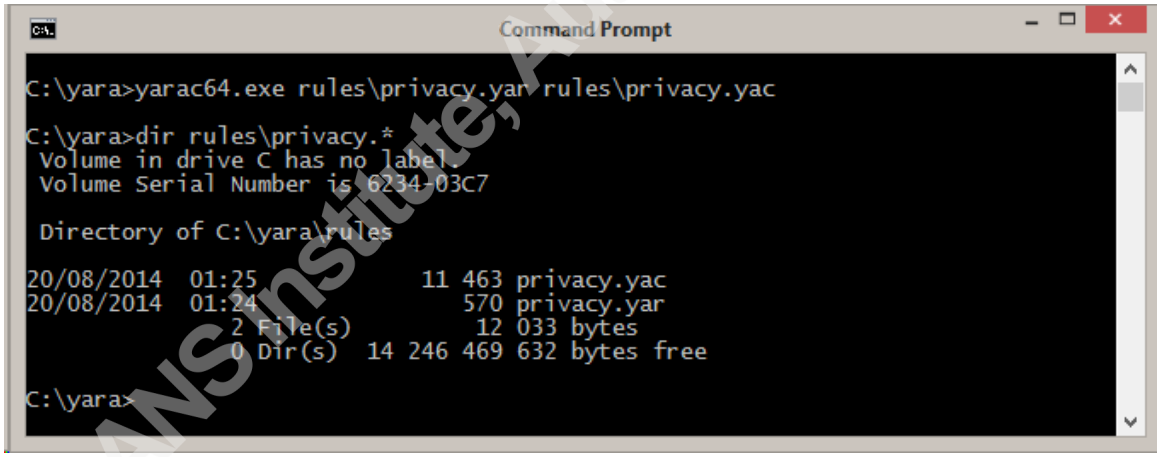


Figure 17 – YARA rule compilation

The subject of rule privacy can be relevant since the rule file might be exposed to unsafe environments controlled by threat actors. Although the rule compilation mitigates the problem there is another consideration to have in mind. The YARA scan engine features the ability to print matching strings using the '-s' switch. This feature is used in debugging or in some cases may be of use in post-processing tasks. Nonetheless, by exposing the matched strings portion of the rule the structure will be disclosed. The same principle applies if the '-m' switch for metadata retrieval is used. As a result, rules

containing matching criteria should be defined as private in order to safeguard rule privacy.

Figure 18 depicts a rule featuring rule-private referencing in the condition section. Figure 19 depicts scan engine execution and output:

```
rule OPEN_ATOMIC {
    meta:
        description = "Open Atomic Indicator"
    strings:
        $str = "autoruns.pdb"
    condition:
        $str
}

rule SUSPICIOUS_OPEN {
    meta:
        description = "Suspicious File"
    condition:
        OPEN_ATOMIC
}

private rule CLOSED_ATOMIC {
    meta:
        description = "Closed Atomic Indicator"
    strings:
        $str = "autoruns.pdb"
    condition:
        $str
}

rule SUSPICIOUS_CLOSED {
    meta:
        description = "Suspicious File"
    condition:
        CLOSED_ATOMIC
}
```

Figure 18 –YARA rule referencing private rules

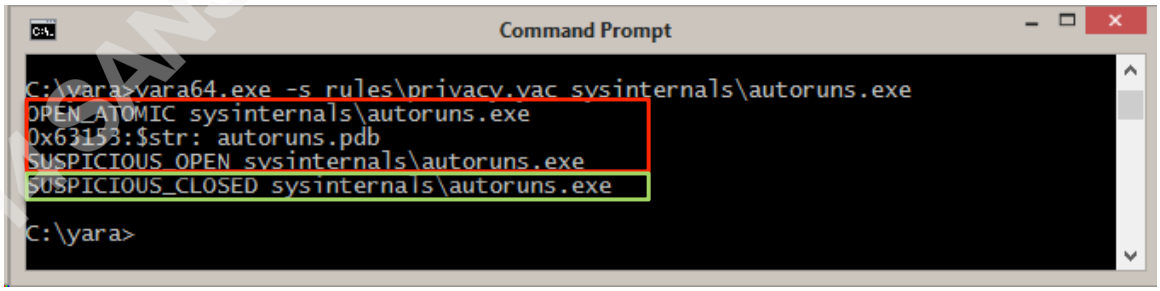


Figure 19 – YARA output

The YARA output illustrated in Figure 18 clearly shows that while the 'Atomic' rules are identical, the fact that the 'closed' one is kept private; it will protect the rule from potential reverse engineering. The red colored output pertains to the open rule,

Ricardo Dias;ricardo.dias@live.com

disclosing rule details. The green colored output pertains to the closed rule which discloses only the details of the intentionally generic referencing rule.

3.5. Extended Features

Due to its intrinsic simplicity, YARA is designed to be a lean and fast scan engine, leaving out features like output control. While input core features provide the ability to parse file and memory objects, the engine yields scanning results directly to standard output, which can be problematic when post-processing is needed. To overcome these limitations and extend the engine functionality the author developed the YARA-Python extension.

The YARA-Python extension allows developing Python scripts featuring all YARA core features described in Section 3.4, and benefit from the vast diversity of Python modules. Likewise, Python scripts can later be compiled using packages like py2exe in order to execute them in systems without Python installed. By wrapping YARA into a Python executable one will be able to develop enhanced YARA scan engines. The next two sections describe how Python can be used to address the limitations described in the last sections.

The capabilities described in the following subsections were developed and prototyped by the current author, who has decided to share them with the community.

3.5.1. YARA featuring REST

As detailed in Section 3.4.1, YARA outputs scan results directly to the command line. Although it is possible to redirect output to a file, the process is cumbersome since files would need post processing in order to facilitate analysis. Furthermore, one would need to move files from the host to a central location (for example using a file share). Another important aspect is rule management. If rules are distributed along with the YARA scan, the distribution package needs to be updated every time a rule is added or updated. Consequently , the ideal scenario relies on placing both rules and reports in a central location.

In order to meet this objective, a Python script featuring YARA and REQUESTS is used. The REQUESTS module provides the ability to communicate with a

Representational State Transfer (REST) application-programming interface (API) using HTTP requests. Figure 20 depicts the structure of the Python script:

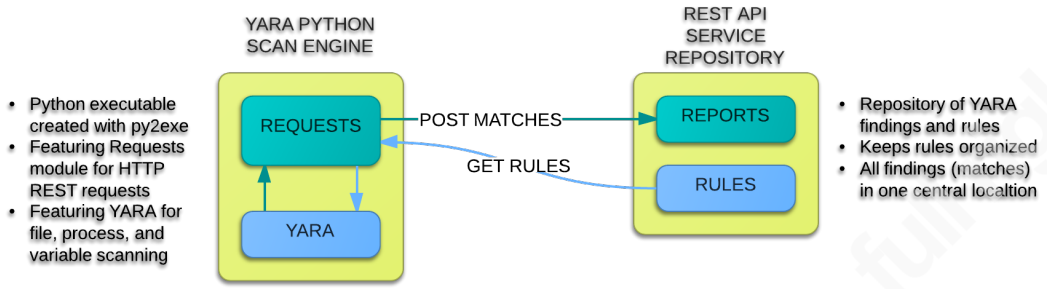


Figure 20 - YARA REST scan engine

By integrating both YARA and REQUEST modules in a Python script, one will be able to get rules and post scan results via REST API. This approach is highly efficient since it relies on a lightweight protocol to exchange data between the host and REST API server. Essentially, each time a YARA package is invoked it fetches the rule, initiates scanning, and uploads every match. By doing so reports are updated in real-time, since scan matches will be uploaded as they occur, otherwise results would only be available at the end of the scan.

As Microsoft Windows does not feature a Python runtime by default, the script need to be converted into a single executable file featuring the Python runtime. Py2exe can be used to convert Python scripts into executables.

A proof of concept of the YARA REST implementation, featuring the script conversion, is available in section 8.

3.5.2. YARA featuring WMI

As detailed in section 3.4.2, YARA's ability to scan a process is limited to a single process ID. In order to work around this limitation YARA must be invoked automatically for each running process. Additionally a special consideration should be made concerning how YARA reports matched processes. When a rule matches a process, it only returns the process identification number. Thence, the method used to list running processes should be able to fetch process details, including full path and arguments. Microsoft Windows operating systems provide an infrastructure for management entitled Windows Management Instrumentation (WMI). Microsoft WMI provides a wide range of

services for local and remote management, including the list of running processes and their details.

As with YARA REST, a Python package featuring YARA and WMI modules are used in order to address the above stated issues. Figure 21 depicts a diagram of YARA featuring a WMI client:

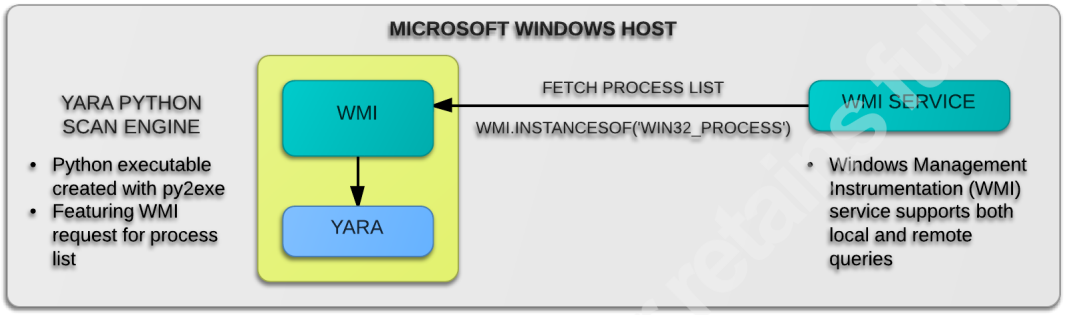


Figure 21 - YARA WMI scan engine diagram

By using Python's WMI module one will be able to fetch running process details, including process ID, process name, and process path. YARA will then run against each running process ID. Additional information including process name or path can be used to replace process ID as the default output.

A proof of concept of the YARA WMI implementation for scanning all running processes is available in section 9.

Alternatively, process details fetched via WMI can be used for rogue process detection. Also known as processes that mimic legitimate processes, rogue process identification is the first step of the memory forensics process (Tilbury, 2012). In so doing, rather than scanning a given process ID, one must pass the process name and path as external variables to a YARA rule. Figure 22 depicts a rule capable of detecting rogue svchost.exe processes:

```
rule SUSP_SVCHOST
{
    condition:
        name matches /svchost.exe/is and
        not path matches /\\windows\\system32\\svchost.exe/is
}
```

Figure 22 - Rule for detecting rogue svchost.exe processes

It should be noted that rules depending solely on external variables do not require a strings section. These rules are special because they do not depend on data scanned by YARA. This is relevant because it provides an alternative to traditional YARA scanning use cases. Take the example above, where Python's WMI module fetches two variables that are passed to YARA.

A proof of concept for the YARA WMI implementation of rogue process detection is available in Section 10.

4. YARA in the Enterprise

While both core and extended features demonstrate YARA's capabilities to accomplish host based indicator scanning, there is still a topic that needs to be addressed: how to deploy and execute the scan engine across the enterprise. In order to accomplish enterprise wide scanning with YARA, a package management platform must be in place. This type of platform usually operates with an agent-based architecture in order to provide deployment and central management for applications. By encapsulating the YARA scan engine into a deployable package the management platform will be able to push and execute it on multiple clients.

When the management console deploys the package to a client computer, it instructs the console agent to invoke the scan engine. Given the scan engine limitations already detailed in Section 3, an extended YARA package must used for deployment. The YARA REST package described in Section 3.5.1 will be adopted for enterprise scanning. Figure 23 illustrates the expected process when deploying YARA packages through the enterprise:

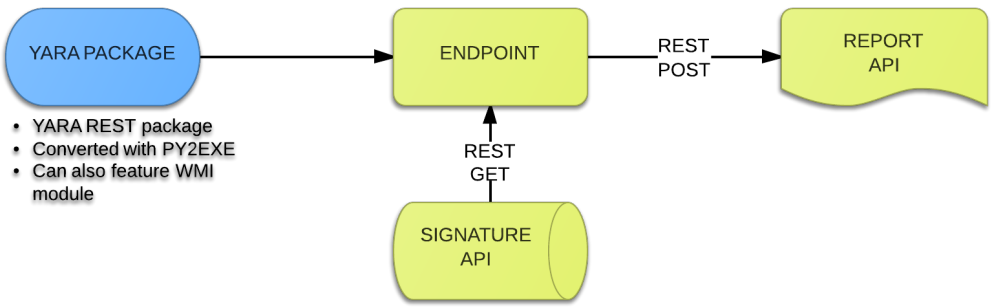


Figure 23 - YARA package deployment process

Ricardo Dias;ricardo.dias@live.com

Distributing the YARA REST package greatly simplifies the deployment and scanning process. In the first place, all rules are centralized in the REST server. This means that all endpoints will fetch the same rule version. Secondly, all endpoints post matches in real-time to the REST server. This enables real-time processing of all findings. The fact that YARA REST is a single executable also simplifies the deployment process.

Within this mindset, the following section will introduce McAfee ePolicy Orchestrator (ePO) as a management framework capable of deploying the YARA package.

4.1. Deploying Yara with McAfee ePO

McAfee ePO is a management framework that provides centralized management for endpoint and network security solutions (McAfee, 2013a). It supports the deployment of security products, centralized policy management, and comprehensive reporting. Endpoint management is supported by the McAfee Agent that contacts the central console via a secure channel in order to retrieve policies and tasks. The agent supports a wide range of operating systems including Microsoft Windows, Mac OS X, HP-UX, IBM AIX, Solaris, RedHat, and other popular Linux distributions (McAfee, 2013b). The extensive list of operating systems supported by the agent makes it the ideal vehicle to distribute YARA packages to systems other than Microsoft Windows. In addition, McAfee ePO tasks support the input of arguments to be passed to the package upon execution at the endpoint. This feature makes it possible to pass arguments including the rule and report path needed by the YARA package.

While the ePO console is well known for managing McAfee security products, like antivirus and host intrusion prevention, it also supports the deployment of custom packages. In order to create custom packages McAfee provides ePO Endpoint Deployment Kit (EEDK). McAfee EEDK is able to create software packages ready to be imported into the ePO software repository (McAfee, 2012). When a package is imported, it can be deployed across all endpoints managed by the console. By using McAfee EEDK, we will be able to create a deployable YARA package in order to scan the endpoint.

Ricardo Dias;ricardo.dias@live.com

Figure 24 illustrates the relation between McAfee EEDK, ePO and its agents when creating and deploying a YARA custom package:

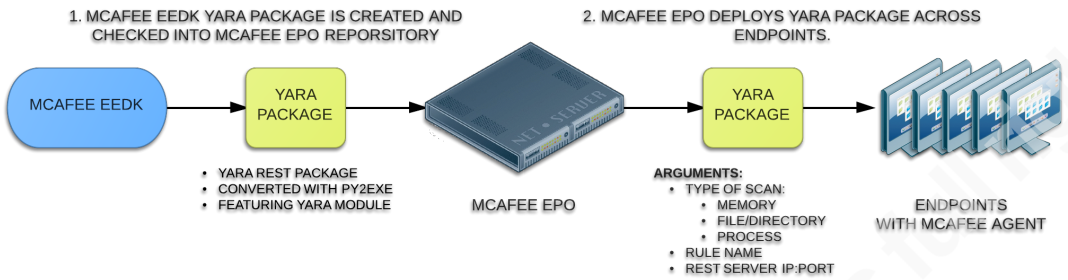


Figure 24 - McAfee EEDK package creation and deployment

The YARA package created by McAfee EEDK embodies the YARA REST scan engine. When McAfee ePO distributes the package to the endpoint, the agents execute the scan engine with the arguments passed by the McAfee ePO task. This means that multiple scanning tasks can be defined for different sets of YARA rules or scanning objects.

Upon YARA distribution to the endpoint, the ePO agent creates a temporary folder to store the package contents. Then, the package is extracted and the scan engine executed by the agent with the command line arguments defined in the task. When the scan tasks finishes, the McAfee agent deletes the temporary folder. An important aspect to consider is that YARA will be executed under the SYSTEM account, used by McAfee Agent. This considerably enhances the scan engine's ability to scan running processes and file objects without the need to rely on a specific domain account.

By using McAfee EEBK, ePO, and the YARA REST, one will be capable of conducting enterprise wide scanning. This process enables the execution and evaluation of indicators gathered during intrusion analysis or external threat intelligence services (i.e. APT reports). The repeated execution of scanning rules, for instance on a weekly basis, will ensure the maturity of the indicators, subsequently improving its quality. Since McAfee ePO's primary purpose is for management of McAfee security tools, the YARA scanning task will provide an extra layer of security on the endpoint defense strategy.

In the context of the kill chain detailed in Section 1.2, the rules appropriate to be featured in the endpoint scan are the ones featured in the 'Installation', 'Command and

Control' and 'Actions on Objectives' phases. This is reasonable since endpoint scans will only be able to gather data at rest, like artifacts in memory and file objects.

Section 11 guides the reader through the process of creating and deploying the YARA REST package across McAfee ePO managed systems.

5. Conclusion

Slowly, as societies adopt a way of life supported by information technology, the threat landscape grows in complexity, with many motives and opportunities threat actors can use. It is imperative that defenders in the cyber ecosystem accept the challenge, and thus evolve, in order to cope with every new threat.

Cyber threat intelligence standards and frameworks enable organizations to develop a solid ground against sophisticated attacks. Traditional CTI research papers often focus in the adoption of standards used for structuring and exchanging cyber threat intelligence. While these are key features of every CTI adoption, organizations must apply intelligence at the endpoint in order render the process useful. There is no point in gathering intelligence if no one is able to use it. During the writing of this paper the absence of cyber threat indicator scanning research was startling. For that reason, the primary goal of this research was to address the indicator scan engine topic.

Due to its simplicity and flexibility, YARA blends perfectly in the CTI concept. By choosing YARA as a scan engine, organizations benefit not only from the fact that it is open source, but also has a relatively low learning curve for the creation of indicators. The implicit rule flexibility and the engine core features combined with a management framework are enough to take the first steps towards an intelligence driven response. YARA is an emergent project, with amazing potential, and it is gaining ever-growing acceptance by both security vendors and professionals. A notable example of a YARA community effort is the Deep End Research's 'Yara Exchange'[1] group (Research, DeepEnd, 2013).

[1] http://www.deependresearch.org/2012/08/yara-signature-exchange-google-group.html

As demonstrated, YARA achieves greater flexibility when executed as a Python module. In so doing, developers will be able to extend the scan engine's functionality. The scenarios explored in Section 3.5 clearly demonstrate how both data input and output can be structured in order to achieve specific goals. This was clearly the case with REST, WMI, and IMPHASH versions of YARA as seen in sections 8, 9, and 13.2, respectively.

In conclusion, cyber-security is starting to provide positive feedback to the APT reality. Every organization that adopts CTI to leverage incident response has considerable advantage over those who remain idle when an intrusion occurs.

Ricardo Dias;ricardo.dias@live.com

6. References

Álvarez, V. M. (2014). *YARA User's Manual.* Retrieved from http://yara.readthedocs.org/en/latest/

Barnum, S. (2014). *Standardizing Cyber Threat Intelligence Information with the Structured Threat Information eXpression.* Retrieved from https://stix.mitre.org/about/documents/STIX_Whitepaper_v1.1.pdf

Dell'Aera, A. (2014). *Thug - Python low-interaction honeyclient*. Retrieved from Github: https://github.com/buffer/thug

Guarnieri, C. (2014). *Cuckoo Sandbox*. Retrieved from http://www.cuckoosandbox.org

Holland, R. (2014). *Five Steps To Build An Effective Threat Intelligence Capability.* Retrieved from https://www.forrester.com/Five+Steps+To+Build+An+Effective+Threat+Intelligence+Capability/fulltext/-/E-RES83841

Hutchins, E., Cloppert, M., & Amin, R. (2011). *Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains.* Retrieved from http://www.lockheedmartin.com/content/dam/lockheed/data/corporate/documents/LM-White-Paper-Intel-Driven-Defense.pdf

Kindervag, J., & Holland, R. (2011). *Planning For Failure.* Retrieved from https://www.forrester.com/Planning+For+Failure/fulltext/-/E-RES60564?objectid=RES60564

Ligh, M., Adair, S., Hartstein, B., & Richard , M. (2010). *Malware Analyst's Cookbook.* Wiley.

Luttgens, J., Pepe, M., & Mandia, K. (2014). *Incident Response & Computer Forensics, Third Edition* . McGraw-Hill.

Mandiant. (2013). *M-Trends 2013.* Retrieved from https://www.mandiant.com/resources/mandiant-reports/

Ricardo Dias;ricardo.dias@live.com

Mandiant. (2014). *Tracking Malware with Import Hashing.* Retrieved from https://www.mandiant.com/blog/tracking-malware-import-hashing/

Mantiant. (2012). *M-Trends 2012.* Retrieved from https://www.mandiant.com/resources/mandiant-reports/

McAfee. (2012). *ePO Endpoint Deployment Kit 9.4 - Community Edition.* Retrieved from https://community.mcafee.com/docs/DOC-3401

McAfee. (2013b). *McAfee Agent 4.8.0 - Product Guide.* Retrieved from https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/24000/PD24333/en_US/MA_480_ProductGuide.pdf

McAfee. (2013a). *McAfee ePolicy Orchestrator 5.0.0.* Retrieved from https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/24000/PD24350/en_US/epo_5_0_product_guide_en-us.pdf

Nemček, P. (2013). *Analysis of Malware Classification Schemas.* Retrieved from http://is.muni.cz/th/256476/fi_m/nemcek_mgrthesis.pdf

Newman, R. (2007). *Computer Forensics: Evidence Collection and Management* . Auerbach.

Research, DeepEnd. (2013). *Yara Resources*. Retrieved from http://www.deependresearch.org/2013/02/yara-resources.html

SANS. (2013). *SANS FOR508.1 Book.* SANS.

Tilbury, C. (2012). *SANS Memory Forensics Cheat Sheet.* Retrieved from https://blogs.sans.org/computer-forensics/files/2012/04/Memory-Forensics-Cheat-Sheet-v1_2.pdf

Wicherski, G. (2009). *peHash: A Novel Approach to Fast Malware Clustering.* Retrieved from https://www.usenix.org/legacy/event/leet09/tech/full_papers/wicherski/wicherski.pdf

Ricardo Dias;ricardo.dias@live.com

7. Appendix A – YARA rule naming scheme

In order to keep the process of rule management linear YARA rule identifications must follow a nomenclature. This not only aids in the creation of new rules but also enables the directly mapping of a rule with the kill chain phase and indicator type where it belongs. In order to properly map the rule within these two contexts, detailed in Section 1, two keywords, or tokens, must be part of the rule ID; they are the category and indicator type. Figure 25 illustrates the proposed YARA rule nomenclature for rule identifications:

```
CATEGORY_CAMPAIN_INDICATORID_INDICATORTYPE
```

Figure 25 - Proposed rule identification nomenclature

Figure 26 and Table 1 demonstrate the nomenclature application for a fictitious APT named 'Freedom':

INDICATOR TYPE	TOKEN
ATOMIC	ATM
COMPUTED	CPT
BEHAVIOURAL	BHV

Figure 26 - Indicators and Yara rule mapping

Table 1 - Kill Chain and Yara rule mapping

KILL CHAIN PHASE	CATEGORY	EXAMPLE
Reconnaissance	REC	REC_FREEDOM_001_ATM
Weaponization	WPN	WPN_FREEDOM_001_CPT
Delivery	EXP	EXP_FREEDOM_001_ATM
Install	INS	INS_FREEDOM_001_BHV
C2	CNC	CNC_FREEDOM_001_ATM
Actions	AOO	AOO_FREEDOM_001_BHV

Ricardo Dias;ricardo.dias@live.com

8. Appendix B – YARA REST Proof of concept

8.1. Requirements

- Python v2.7.8;
- Python-YARA module v3.0;
- Python-requests module v2.3.0;
- Python-bottle module v0.12;
- Python-py2exe module v0.6.9.

8.2. Proof of concept code

8.2.1. YARAREST.PY

File yararest.py is a Python script that features YARA and REQUEST modules for file object scanning and REST communications, respectively. The goal of this script is to demonstrate how to fetch rules and upload reports to a central location. Figure 27 depicts the proof of concept code for yararest.py:

```
Import os
import sys
import yara
from requests import post

if len(sys.argv) < 2:
    sys.exit("yara_rest folder yararule")

# fetch yara signature file
try:
    rule_payload = { 'rulename' : sys.argv[2] }
    r = post( 'http://192.168.1.65:8080/get', data = rule_payload )
except:
    sys.exit("- Failed to download yara rule")

# fetch the rules
rules = yara.compile( sources = { 'namespace' : r.text } )

for root, dirs, filenames in os.walk( sys.argv[1] ):
    for name in filenames:
        try:
            file_path = os.path.join( root, name )
            matches = rules.match( filepath = file_path )

            if matches:
                payload = { "rulename" : matches[0],
                            "filename" : file_path,
                            "hostname" : os.environ['COMPUTERNAME'] }
                p = post( 'http://192.168.1.65:8080/put', data = payload )
        except:
            continue
```

Figure 27 - yararest.py proof of concept

Ricardo Dias;ricardo.dias@live.com

The yararest.py script receives two arguments from the command line. The first argument is the scanning folder and the second argument is the YARA rule name. Upon execution, the script attempts to download the YARA rules from the REST server. Then it iterates over the files found in the scanning folder and subfolders. Every time a rule matches a file it uploads the rule name, file path, and hostname to the REST server.

In order to render this code executable in systems without the Python runtime, it is necessary to convert the script into an executable. The next section details this process.

8.2.2. SETUP.PY

File setup.py is a Python script featuring py2exe module that will accomplish Python to executable conversion. Figure 28 depicts the setup.py script, and Figure 29 depicts the setup.py execution:

```
from distutils.core import setup
import py2exe

setup(
    options = {'py2exe': {'bundle_files': 1, 'compressed': True}},
    console = [{'script': "yararest.py"}],
    zipfile = None
)
```

Figure 28 - setup.py featuring py2exe module

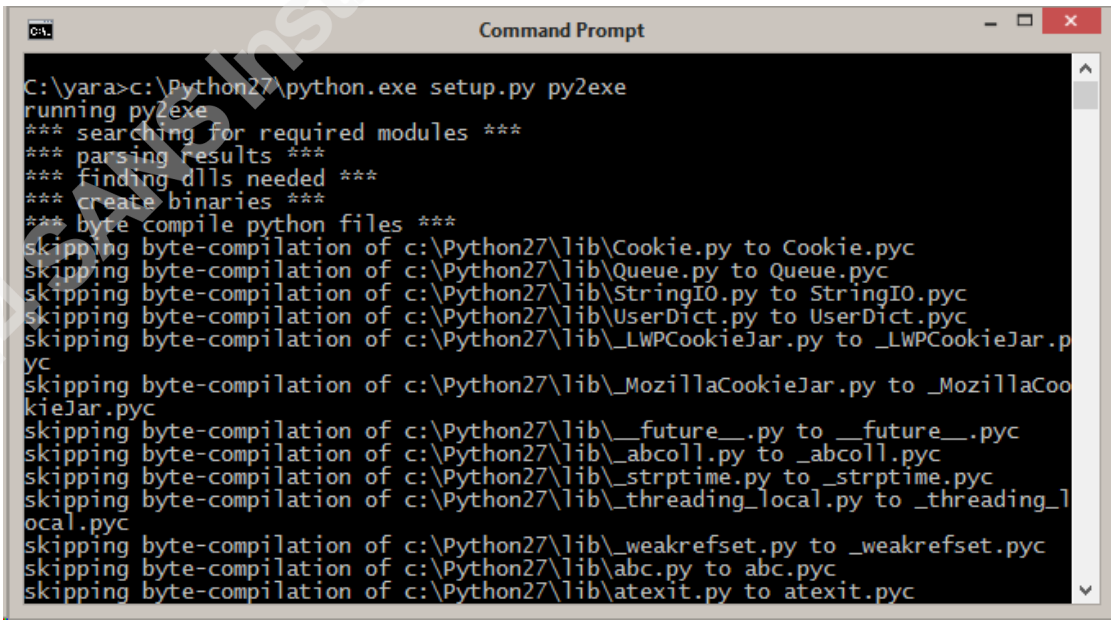


Figure 29 - python conversion to executable with py2exe

Ricardo Dias;ricardo.dias@live.com

8.2.3. RESTSERVER.PY

File restserver.py is a Python script API, featuring TIME and BOTTLE modules. The BOOTLE module is responsible for handling HTTP requests from yararest.exe. Figure 30 depicts the proof of concept code for restserver.py:

```
from time import gmtime, strftime
from bottle import route, get, run, post, request

@post('/get')
def index():
    rulename = request.forms.get('rulename')
    f = open(rulename,'rb')
    rule = f.read()
    f.close()
    return rule

@post('/put')
def index():
    recivedt = strftime('%Y-%m-%d %H:%M:%S', gmtime())
    rulename = request.forms.get('rulename')
    filename = request.forms.get('filename')
    hostname = request.forms.get('hostname')
    f = open("results.txt","a")
    f.write("%s,%s,%s,%s\r\n" % (recivedt,hostname,rulename,filename))
    f.close()
    return ""

run(host='192.168.1.65', port=8080)
```

Figure 30 - restserver.py proof of concept

The restserver.py API handles GET and PUT requests. GET requests attempt to read the rule file and return its contents while PUT requests read the variables from the HTTP post and write them to the 'results.txt' report file.

Ideally, script restserver.py runs on a dedicated server with a Linux operating system. Since the goal of this proof is solely to demonstrate the capabilities of Python as a REST server, the following improvements can be made:

- Store the match results into a SQLITE database;
- Create an additional method for handling result queries (for example return all matches for a give system or rule);
- Create an additional method for handling rule metadata queries. Since only the rule identifier is stored in the results file or database, this method would allow returning rule metadata and tags.

Ricardo Dias;ricardo.dias@live.com

8.2.4. Report format

The report generated by the restserver.py script is stored as a file. The first column is the detection date and time. The second column is the system computer name. The third column is the rule name. Lastly, the fourth column is the matched file path. Figure 31 depicts an excerpt of the report contents:

```
2014-08-20 17:15:09,W8_SYSTEM,suspicious_pe,d:\yara\SAMPLES\AcLayers.dll
2014-08-20 17:15:09,W8_SYSTEM,suspicious_pe,d:\yara\SAMPLES\procexp.exe
```

Figure 31 - restserver.py report excerpt

9. Appendix C – YARA WMI proof of concept for process scanning

9.1. Requirements

- Python v2.7.8;
- Python-YARA module v3.0;
- Python-Win32 module v269;
- Python-WMI module v1.4.9.

9.2. Proof of concept code

9.2.1. YARAWMI_PS.PY

File yarawmi_ps.py is a Python script that features YARA and WMI modules, for file object scanning and WMI requests, respectively. The goal of the script is to scan all running processes. Figure 32 depicts the proof of concept code of yarawmi_ps.py:

```
import os
import sys
import wmi
import yara

myPid = os.getpid()
c     = wmi.WMI ()
rules = yara.compile(filepath=sys.argv[1])

for process in c.Win32_Process ():
    pId = process.ProcessId
    pName = process.Name
    pPath = process.ExecutablePath
    pCmd = process.CommandLine
    pOwner = process.GetOwner()[2]

    if pId != myPid:
        try:
            matches = rules.match(pid=pId)
        except:
            print('Failed inspecting PID: %d' % pId)
            continue

        if matches:
            print("%s matched %s [PID:%s]" % (matches, pPath, pId))
        else:
            print("No matches for %s [PID:%s]" % (pName, pId))
```

Figure 32 - yarawmi_ps.py proof of concept code

The yarawmi_ps.py script receives the YARA rule name as an argument from the command line. The script reads the YARA rule and iterates over the process identifiers

provided by the WMI query. Every time a rule matches a process, the script redirects the output to standard output.

For the sake of simplicity, contrary to the yararest.py proof of concept code, yarawmi_ps.py does not feature the REQUEST module for REST integration. Section 8 provides a proof of concept that should be easily adapted to yarawmi_ps.py.

Ricardo Dias;ricardo.dias@live.com

10. Appendix D – YARA WMI proof of concept for rogue process detection

10.1. Requirements

- Python v2.7.8;
- Python-YARA module v3.0;
- Python-Win32 module v269;
- Python-WMI module v1.4.9.

10.2. Proof of concept code

10.2.1. YARAWMI_RS.PY

File yarawmi_rs.py is a Python script that features YARA and WMI modules, for file object scanning and WMI requests respectively. The goal of the script is to scan for rogue processes. Figure 33 depicts the proof of concept code of yarawmi_rs.py:

```
import os
import wmi
import yara

myPid = os.getpid()
c     = wmi.WMI ()
rules = yara.compile(
    filepath='sample.yar',
    externals={'path': 'path', 'name': 'name'})

for process in c.Win32_Process ():
    pId = process.ProcessId
    pName = process.Name
    pPath = process.ExecutablePath

    if pId != myPid:
        try:
            matches = rules.match(
                data='',
                externals={'path': str(pPath), 'name': str(pName)})
        except:
            continue

        if matches:
            print("%s matched %s [PID:%s]" % (matches, pPath, pId))
        else:
            print("No matches for %s [PID:%s]" % (pName, pId))
```

Figure 33 - yarawmi_rs.py proof of concept code

The yarawmi_rs.py script receives the YARA rule name as argument from the command line. The script reads the YARA rule and iterates over the running processes

provided by the WMI query. Contrary to yarawmi_ps.py where YARA scans the entire process memory, only the process name and path are passed to YARA. Every time a rule matches a process, the script redirects the output to standard output.

As in yarawmi_ps.py, yarawmi_rs.py does not feature the REQUEST module for REST integration. Section 8 provides a proof of concept that could be easily adapted to yarawmi_rs.py.

11. Appendix E – Creating, check-in, and distributing YARA with McAfee ePO and EEDK

11.1. Requirements

- Server with McAfee ePolicy Orchestrator v4.6.x or v5.x;
- Endpoint with McAfee ePO Endpoint Deployment Kit v9.4;
- YARA REST package;
- YARA REST server;
- Endpoint managed by McAfee ePO.

11.2. Package creation

At the endpoint where EEDK was extracted, go to the extract location and execute the file EEDK.EXE with administrative privileges. Figure 34 depicts EEDK graphical user interface:

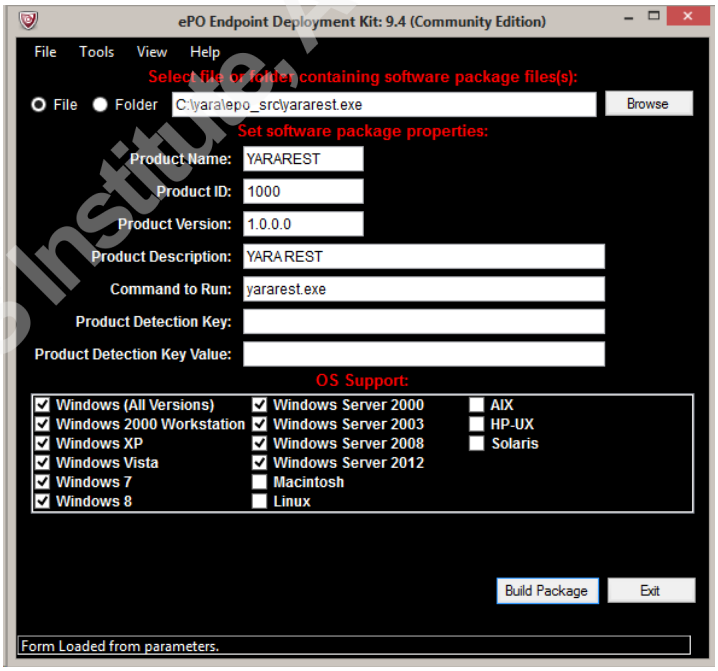


Figure 34 - EEDK main window

Select the file option and click the browse button. Navigate to the folder containing the YARA REST executable and select the file. The next step is the configuration of the package. Set the product name, product ID and product description.

Ricardo Dias;ricardo.dias@live.com

It is important to use definitions that correspond to the package contents. The command to run is also important; make sure to specify the correct executable. Registry keys and values do not need to be defined, so they can remain empty. Lastly, select the supported operating systems. A last change must be made before building the package. At the top menu, select 'tools' and choose 'options'. Define the build folder where the ePO package will be created, and press the save button. Figure 35 depicts the options window:

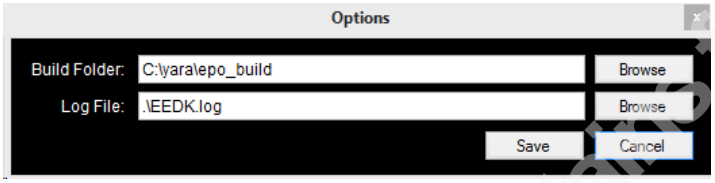


Figure 35 - EEDK options window

Back to the main window, when everything is setup, press the 'build package' button. The process to create the package begins and takes a few seconds to complete. Figure 36 depicts the message displayed when the package build is complete:

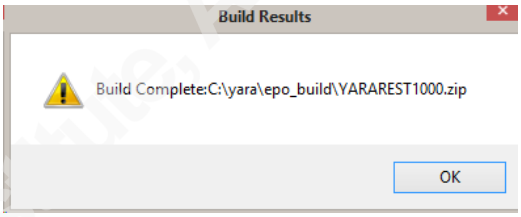


Figure 36 - EEDK notification window

With the ePO package created, the next step is to check-in the package in the ePO console.

11.3. Package check-in

After logging into the ePO console, press 'menu' and select 'master repository'. At the master repository pane, press 'check in package' button. In the check in package wizard select 'product or update', press the 'browse' button, and navigate to the folder containing the package created by EEDK. Figure 37 depicts the check in pane:

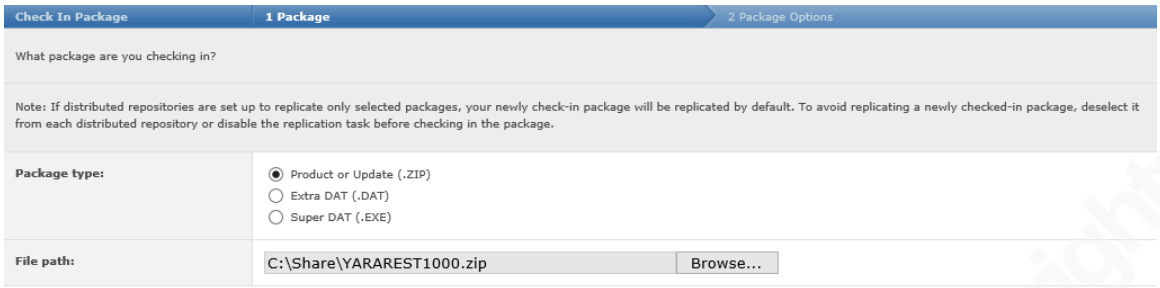


Figure 37 - Master repository package check-in

At the package options pane confirm that the package information meets the name and version defined during package creation. Select the 'current' branch and then press 'save' button to check in the package in the master repository. Figure 38 depicts the package options:

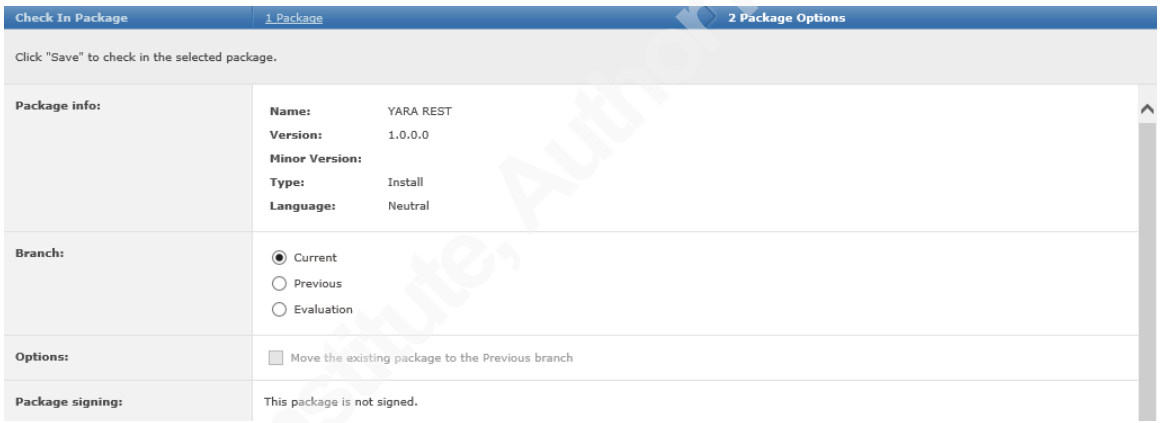
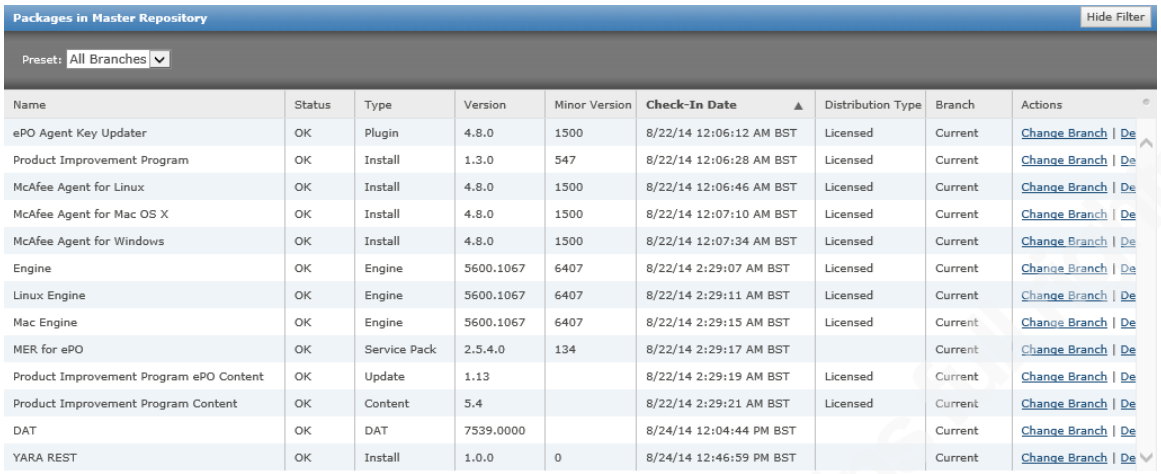


Figure 38 - Master repository package options

When the check-in package is complete, the YARA package will be available in the master repository package list. Figure 39 depicts the package list:

Ricardo Dias;ricardo.dias@live.com



Name	Status	Type	Version	Minor Version	Check-In Date	Distribution Type	Branch	Actions
ePO Agent Key Updater	OK	Plugin	4.8.0	1500	8/22/14 12:06:12 AM BST	Licensed	Current	Change Branch | De
Product Improvement Program	OK	Install	1.3.0	547	8/22/14 12:06:28 AM BST	Licensed	Current	Change Branch | De
McAfee Agent for Linux	OK	Install	4.8.0	1500	8/22/14 12:06:46 AM BST	Licensed	Current	Change Branch | De
McAfee Agent for Mac OS X	OK	Install	4.8.0	1500	8/22/14 12:07:10 AM BST	Licensed	Current	Change Branch | De
McAfee Agent for Windows	OK	Install	4.8.0	1500	8/22/14 12:07:34 AM BST	Licensed	Current	Change Branch | De
Engine	OK	Engine	5600.1067	6407	8/22/14 2:29:07 AM BST	Licensed	Current	Change Branch | De
Linux Engine	OK	Engine	5600.1067	6407	8/22/14 2:29:11 AM BST	Licensed	Current	Change Branch | De
Mac Engine	OK	Engine	5600.1067	6407	8/22/14 2:29:15 AM BST	Licensed	Current	Change Branch | De
MER for ePO	OK	Service Pack	2.5.4.0	134	8/22/14 2:29:17 AM BST		Current	Change Branch | De
Product Improvement Program ePO Content	OK	Update	1.13		8/22/14 2:29:19 AM BST	Licensed	Current	Change Branch | De
Product Improvement Program Content	OK	Content	5.4		8/22/14 2:29:21 AM BST	Licensed	Current	Change Branch | De
DAT	OK	DAT	7539.0000		8/24/14 12:04:44 PM BST		Current	Change Branch | De
YARA REST	OK	Install	1.0.0	0	8/24/14 12:46:59 PM BST		Current	Change Branch | De

Figure 39 - Master repository package list

11.4. Package deployment

Before deploying the package to a managed system, first a client task must be created in the client task catalog. At the ePO console, press 'menu' and select 'client task catalog. At the client task catalog, select 'new task', then select 'product deployment' as task type. At the task details pane fill the task name, description, target platforms, and product and components. The product will be the YARA package. Command line parameters are also necessary since YARA REST requires two arguments: 'scanning path' and 'rule name'. Press 'save' button to create the client task. Figure 40 depicts the task configuration pane:

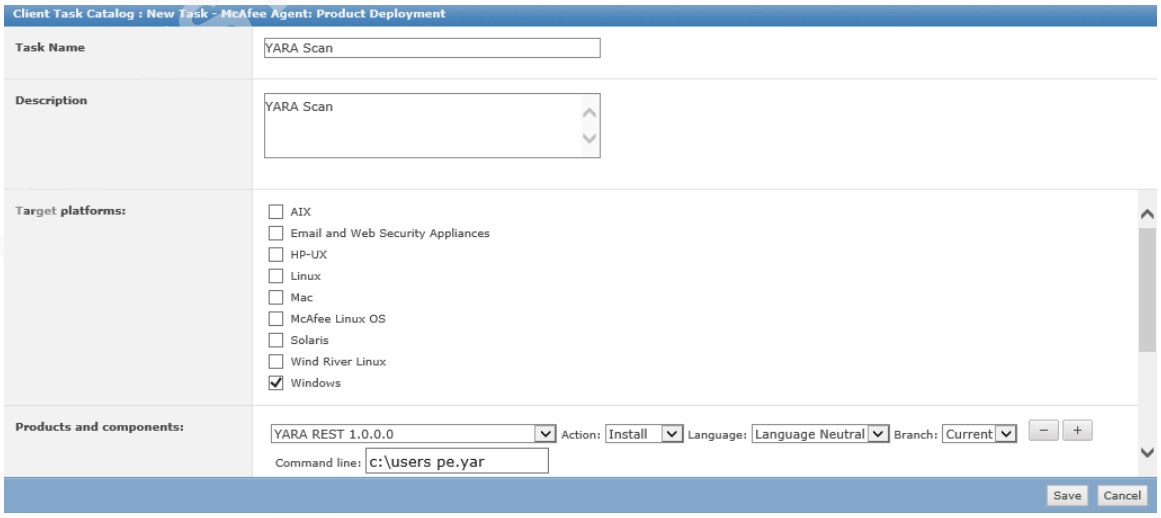


Figure 40 - task configuration pane

Ricardo Dias;ricardo.dias@live.com

Now that the client task is created at the catalog, it can be applied to a managed system. In the ePO console, select ‘menu’ and then ‘system tree’. Navigate the system tree in order to find the target system. Check the target system, press ‘actions’ button, select ‘agent’ from the menu, and select ‘modify tasks on a single system’. Figure 41 depicts the task assignment process:

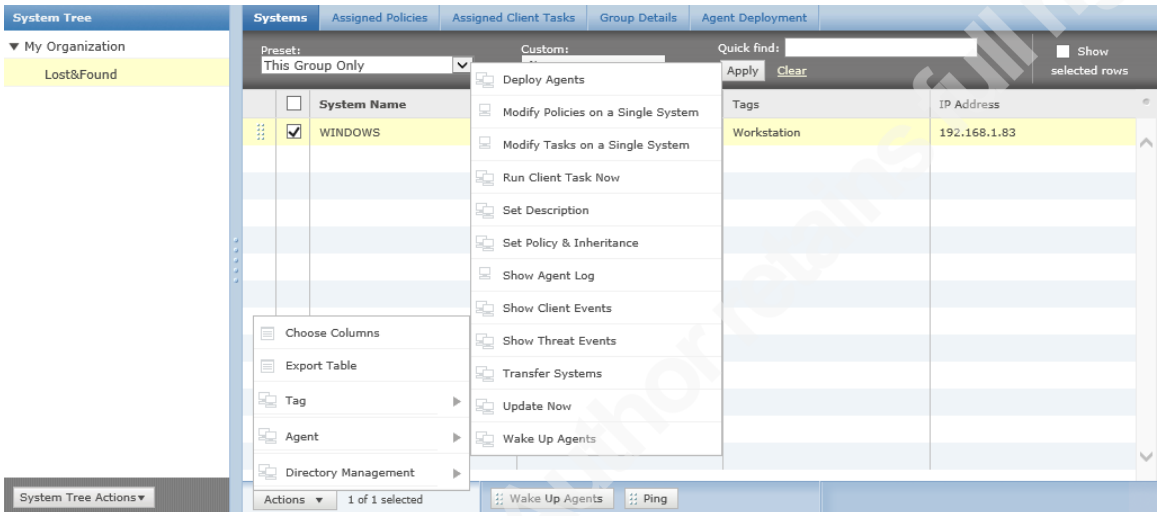


Figure 41 - system task assignment

At the task creation wizard, select ‘mcafee agent’ as a product, ‘product deployment’ as task type, and then the YARA task previously created. Select next to continue with the task scheduling. Figure 42 depicts the task configuration process:

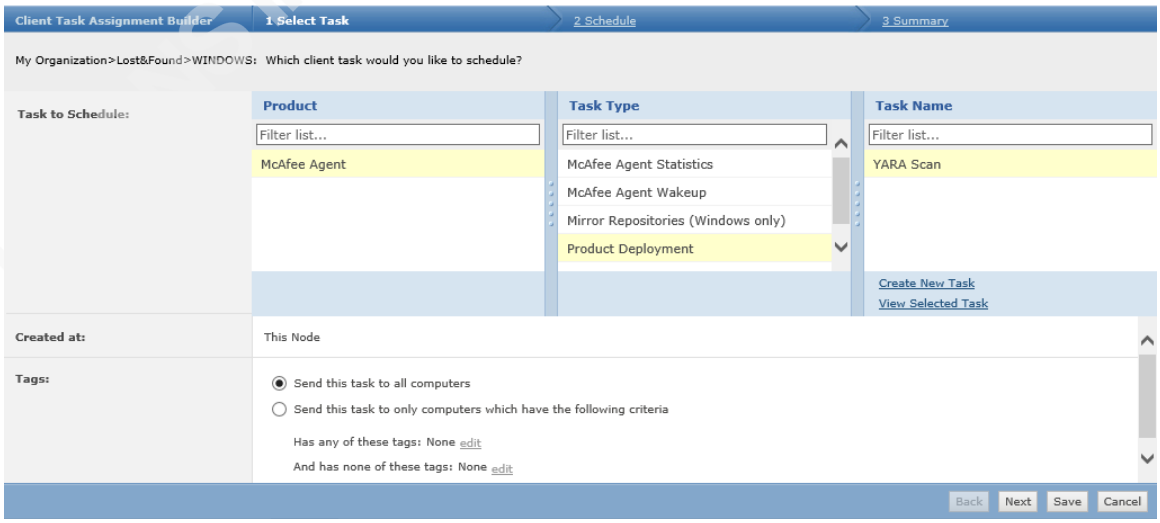


Figure 42 - task configuration process

Ricardo Dias;ricardo.dias@live.com

Finally, ensure that the schedule status is set to 'enable'. For testing purposes, the schedule type will be set to 'run immediately'. Select save to create the task. Figure 43 depicts the task scheduling process:

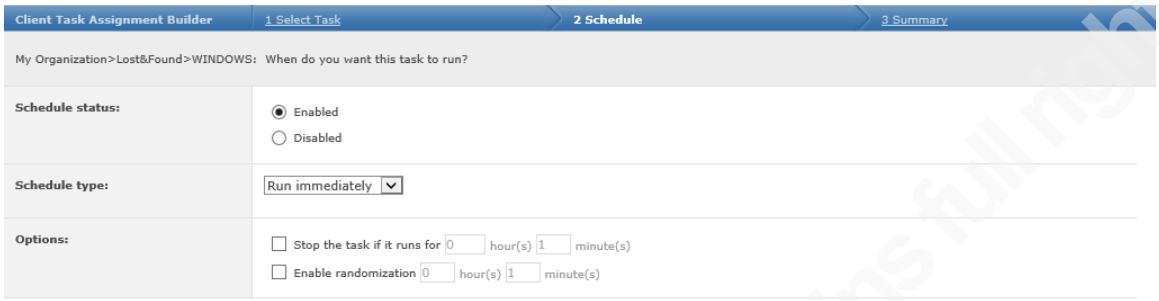


Figure 43 - task scheduling process

When the managed system contacts the ePO server, it downloads the new task and executes it accordingly, that is, immediately. Figure 44 depicts an excerpt of an agent log file while executing the YARA task:

```
24/08/14   16:18:51  Scheduler: Task [YARA Scan] is finished
24/08/14   16:18:51  The task YARA Scan is successful
24/08/14   16:18:38  Update Finished
24/08/14   16:18:38  Update succeeded to version 1.0.0.0.
24/08/14   16:18:25  Agent is looking for events to upload
24/08/14   16:18:17  Verifying yararest.exe.
24/08/14   16:18:08  Downloading yararest.exe.
24/08/14   16:18:08  Installing YARAREST1000.
24/08/14   16:18:07  Loading update configuration from: PkgCatalog.xml
24/08/14   16:18:07  Extracting PkgCatalog.z.
24/08/14   16:18:07  Verifying PkgCatalog.z.
24/08/14   16:18:07  Downloading PkgCatalog.z.
24/08/14   16:18:07  Root key was not specified.
24/08/14   16:18:07  Verifying YARAREST1000-det.mcs.
24/08/14   16:18:06  Loading update configuration from: catalog.xml
24/08/14   16:18:06  Extracting catalog.z.
24/08/14   16:18:06  Verifying catalog.z.
24/08/14   16:18:06  Initializing update...
24/08/14   16:18:05  Scheduler: Invoking task [YARA Scan]...
```

Figure 44 - McAfee agent log excerpt

Ricardo Dias;ricardo.dias@live.com

If the package executed correctly the next step is to check the REST server for scan reports. The scan report format is defined in the REST server. For more details on the REST server and report format, please refer to section 9 of the paper.

Ricardo Dias;ricardo.dias@live.com

12. Appendix F – Requirements website references

Table 2 - References

Requirement	Website reference
Python v2.7.8	https://www.python.org/
Python-yara module v3.0	http://plusvic.github.io/yara/
Python-requests module v2.3.0	http://docs.python-requests.org/en/latest/
Python-bottle module v0.12	http://bottlepy.org/docs/dev/index.html
Python-py2exe module v0.6.9	http://www.py2exe.org/
Python-win32 module v269	http://sourceforge.net/projects/pywin32/
Python-wmi module v1.4.9	http://timgolden.me.uk/python/wmi/index.html
Python-pefile module v1.2.10-139	http://code.google.com/p/pefile/
McAfee ePO Endpoint Deployment Kit v9.4	https://community.mcafee.com/docs/DOC-3401

Ricardo Dias;ricardo.dias@live.com

13. Appendix G - Case Studies

13.1. Incident response to BANKERBOT with YARA

13.1.1. The incident

By August 1st 2014, at 10:15AM, a notification email, from the incident response platform, drops in the ACME's computer security incident response team (CSIRT) mailbox. An employee from the financial department reported strange behavior after logging on the organization e-banking account. Apparently, the banking institution is offering, via the e-banking portal, a new tool to enhance mobile security. The user is compelled to fill a submission form in order to download and install the mobile application. While checking the behavior with other co-workers the employee finds the behavior suspicious and opens an incident in the CSIRT portal.

13.1.2. The response

ACME's incident response plan resembles the process documented in Section 3.3. The CSIRT first goal is to determine the veracity of the incident and step into the identification phase with the intrusion analysis of the affected system. Both memory and hard drive images are acquired.

Forensic analysis of both images reveals injected executable code in multiple processes. Suspicious mutexes are present across all injected processes. Additionally memory analysis also reveals suspicious domains in the address space of the injected processes. Finally, timeline analysis reveals the creation of suspicious files a couple of hours before the incident was reported. Figure 45 depicts a summary of the intrusion analysis key findings:

```
Files created:
    c:\programdata\moon\hgzdv.dat
    c:\programdata\moon\gqxbg.bck

Domains resolved:
    d4rkl33ch.ru
    client.f4stp1cs.ru

Mutexes created:
    F6A013EC00DA_STOP
    5F04AF3E11CD_INIT
```

Figure 45 - intrusion analysis key findings

Next step of identification is to move forward in order to determine the full scope of the intrusion. That is, if there are more systems infected with the same malware. This involves revealing cyber threat indicators from the key findings. In order to derive cyber threat indicators from the key findings, the CSIRT began creating YARA rules and mapping them to the kill chain.

Analysis of created files 'hgzdv.dat' and 'gqxbg.bck' concluded that the file type was unknown. They seem to be data files used by the malware, perhaps to store the configuration file. Moreover, both files reveal a high density, or entropy, which indicates they are probably encrypted. This also means there are few details within the files in order to create a YARA rule. Therefore, no indicator will be revealed.

Regarding domain names and mutexes, since these are present in clear text within the address space of injected processes, the CSIRT was able to create two YARA rules to match the malware presence. Figure 46 depicts the BANKERBOT YARA rules:

```
rule INS_BANKERBOT_001_ATM: bankerbot mutexes
{
    meta:
        description = "bankerbot mutexes"
        killchain = "install"
    strings:
        $mutex01 = /[A-F0-9]{12}_STOP/
        $mutex02 = /[A-F0-9]{12}_INIT/
    condition:
        $mutex01 and $mutex02
}

rule INS_BANKERBOT_002_ATM: bankerbot c2
{
    meta:
        description = "banker bot c2 domains"
        killchain = "c2"
    strings:
        $dns01 = "d4rkl33ch.ru"
        $dns02 = "client.f4stp1cs.ru"
    condition:
        $dns01 and $dns02
}
```

Figure 46 – BANKERBOT YARA rules

Both rules identification were created using naming scheme detailed in Section 8. With the rules created, the CSIRT was able to perform an enterprise wide scan for the indicators revealed during analysis. Rules were then imported to the REST server so that YARA REST can access them. Finally the CSIRT created an ePO YARA task for scanning all managed systems against the newly created rules. Both the process of

defining the REST server and ePO package deployment are detailed in sections 8 and 11 respectively.

13.1.3. Lessons learned and follow-up

After the eradication and recovery phases of the incident response process, the CSIRT highlighted a few takeaways:

- Two more systems were found to be infected;
- In-depth forensic analysis revealed that infection vector was by phishing email with malicious website. In the context of the kill chain the email message is the delivery mechanism;
- BANKERBOT sample downloaded from the website, was obfuscated by a packer, rendering the creation of a YARA rule useless;
- Memory scanning proved to be the best solution to bypass the malware obfuscation techniques and detect infected machines efficiently;
- Preventive measures involve fine-tuning the anti-spam framework in order to detect this type of phishing messages. If possible using YARA rules directly in the anti-spam appliance;
- Another preventive measure would be to fine-tune the web proxy in order to detect both malicious websites and files related with this malware. If possible, use YARA rules directly in the proxy.

13.2. Malware clustering with YARA and IMPHASH

Malware clustering with hashing functions is not a new concept. Essentially, it focus on the task of grouping multiple instances of the same malware (Wicherski, 2009). In 2014, Mandiant published in their blog a new malware clustering technique named ‘import hash’, or ‘imphash’. Fundamentally, the technique consists in generating a hash based on the PE file imports (Mandiant, 2014).

PEFILE, a python module for PE file parsing capabilities, supports the imphash function. This enables Python scripts featuring PEFILE module to apply further actions

depending on the imphash value. Within this mindset, one can consider the imphash as a value to be passed as an external variable to a YARA rule. Figure 47 depicts the proof of concept of a python script featuring YARA REST with IMPHASH module.

```
import os
import sys
import yara
import pefile
import requests

try:
    rule_payload = { 'rulename' : sys.argv[2] }
    r = requests.post('http://192.168.1.65:8080/get', data=rule_payload)
except:
    sys.exit("failed to download yara rule")

for root, dirs, filenames in os.walk(sys.argv[1]):
    for name in filenames:
        filename = os.path.join(root, name)

        try:
            pe = pefile.PE(filename)
            imphash = pe.get_imphash()
            print("%s:%s" %(imphash,filename))
        except:
            print("- unable to parse %s" % filename)
            continue

        rules = yara.compile(
            sources = { 'namespace' : r.text },
            externals = { 'imphash' : '' } )

        matches = rules.match(
            data='',
            externals = { 'imphash': imphash } )

        if matches:
            print('%s matched yara rule' % filename)
            payload = {
                "rulename" : matches[0],
                "filename" : filename,
                "hostname" : os.environ['COMPUTERNAME']
            }
            p = requests.post('http://192.168.1.65:8080/put', data=payload)
```

Figure 47 - yaraimphash.py proof of concept

The yaraimphash.py is very similar to yararest.py detailed in Section 8. The change lies in the fact that the file's data is not passed to YARA. Instead, the file's data is used by PEFILE in order to compute the imphash value. Next, only the imphash value is passed to YARA as an external variable. Naturally, the YARA rule must be able to accept the variable. Figure 48 depicts a YARA rule featuring the imphash variable:

```
rule suspicious_imphash {
    meta:
        desc = "suspicious imphash file"
    condition:
        imphash contains "ac710138693a6f811a55ad16b930e041"
}
```

Figure 48 - yara rule featuring imphash value

The above rule will match all PE files with an imphash value of "ac710138693a6f811a55ad16b930e041". Just like in the yararest.py, yaraimphash.py will be able to fetch rules and send reports to a REST server. Figure 49 depicts the executable version of 'yaraimphash', while figure 50 depicts the REST report result stored in report.txt file:

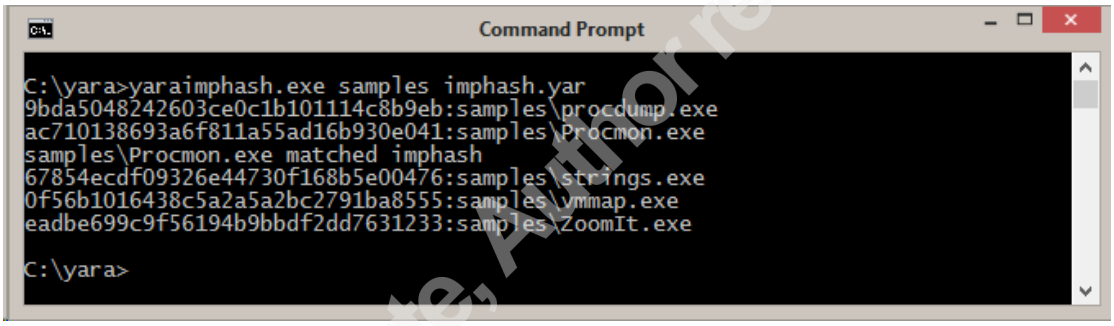


Figure 49 - yaraimphash execution

```
2014-08-20 18:15:12,W8_SYSTEM,suspicious_imphash,samples\Procmon.exe
```

Figure 50 - yaraimphash report.txt on the REST server

14. Appendix H – Fitting all together

In essence, YARA code exposed in the REST, WMI and IMPHASH modules can all fit perfectly, delivering an even more flexible cyber threat indicator solution. The concept builds upon two main components: a cyber threat intelligence framework and the YARA all-in-one scan engine. The cyber threat intelligence framework will feature an REST API with capabilities of delivering YARA rules and receiving matches, as detailed in Section 3.5.1. As for the YARA all-in-one package, besides the scanning module, it will feature the REST client in order to communicate with the cyber threat intelligence framework. Additionally, input modules featuring WMI and PEFILE, enable the scan engine to ingest specific data according to the scan objective. Figure 51 depicts a high-level overview of the concept described above:

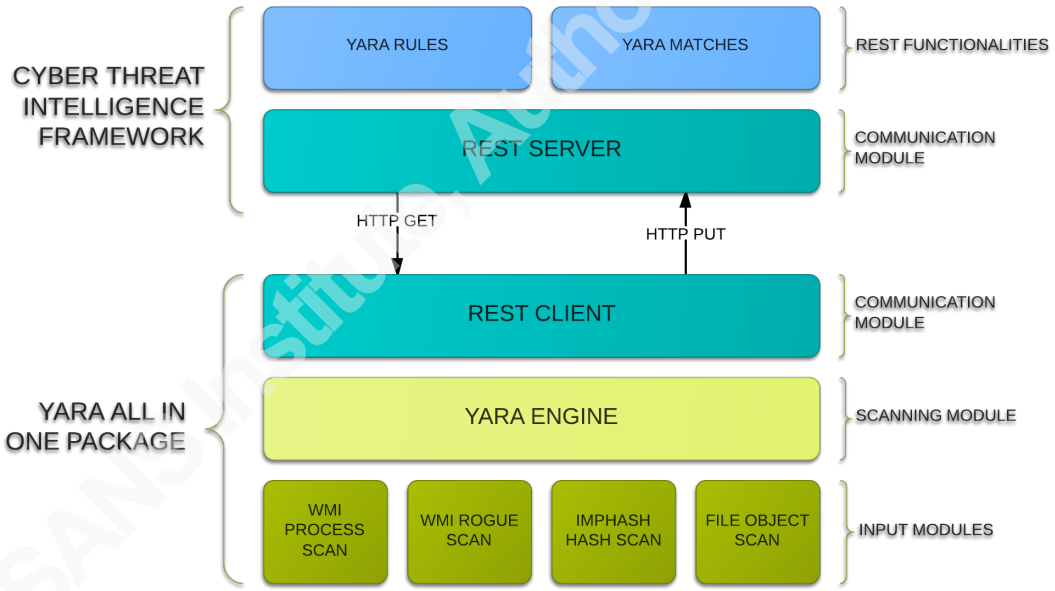


Figure 51 - All-in-one YARA solution