Which YARA Rules Rule: Basic or Advanced?

GIAC (GCIA) Gold Certification and RES 5500

Author: Christopher S. Culling, csculling1@gmail.com
Advisor: Sally Vandeven

Accepted: July 29, 2018

Abstract

YARA rules, if used effectively, can be a powerful tool in the fight against malware. However, it appears that the majority of individuals who use YARA write only the most basic of rules, instead of taking advantage of YARA’s full functionality. Basic YARA rules, which focus primarily on identifying malware signatures via detection of predetermined strings within the target file, folder, or process, can be evaded as malware variants are created. Advanced YARA rules, on the other hand, which often include signatures as well, also focus on the malware’s behavior and characteristics, such as size and file type. While it is not uncommon for strings within malware to change, it is much rarer that its primary behavior will. After analyzing multiple samples of two different malware strains within the same family, it became clear that using both basic and advanced YARA rules is the most effective way for users and analysts to implement this powerful tool. As there are a large number of advanced capabilities contained within YARA, this paper will focus on easy-to-use, advanced features, including YARA's Portable Execution (PE) module, to highlight some of the more powerful aspects of YARA. While it takes more time and effort to learn and utilize advanced YARA rules, in the long run, this method is a worthwhile investment towards a safer networking environment.
1. Introduction

YARA is a recursive acronym which, according to its founder, stands for either Yet Another Recursive Acronym or Yet Another Ridiculous Acronym. It is a tool used to identify and classify malware through the use of signature-based rules and other target characteristics that users can run against files, folders, and processes. There are basic YARA rules, such as searching for a particular text string within a file, and more advanced YARA rules, such as searching for data at a specific virtual memory address in a running process. YARA syntax closely resembles the C language (Alvarez, 2018).

1.1. Purpose

While it is possible to find articles and resources which explain basic or advanced usages of YARA rules, one of the topics that was missing from the literature was a comparative analysis of the basic and advanced YARA rules against each other. The literature on YARA also does not address whether or not the extra time it takes to write advanced rules will be more beneficial in the long run, compared to the efficiency of only utilizing the basic, easier-to-write rules. To account for this gap, the focus of this research will be on analyzing this comparison to determine which YARA rules rule - basic or advanced.

The research question that this paper will ultimately attempt to answer is: When attempting to identify malware, how much more effective, if at all, is the utilization of more complex, advanced YARA rules than the use of easier-to-write, basic rules?

1.2. Significance

This research question is worth being answered because it appears, through both discussions with those who write YARA rules, and reviews of rules posted on the Internet, that most YARA users do not take advantage of its more advanced capabilities. Instead, they mostly rely on YARA’s more basic features. According to Robert M. Lee, CEO of Dragos, Inc., and SANS Certified Instructor, as a result of “what I’ve seen, folks I’ve taught, and the YARA rules that get published by vendors […] not many researchers take advantage of YARAs extendable nature” (R. Lee, personal correspondence, May 12,

Christopher S. Culling, csculling1@gmail.com
Which YARA Rules Rule: Basic or Advanced?

2017). Additionally, an anonymous YARA superuser provided several reasons to explain why many YARA users only rely on its basic functions:

1. The basic usage of YARA is good enough.

2. Users don’t care about, or don’t understand, the concept of what I call resilient rules. Writing rules looking for a combination of unique strings is good, but all it takes is those unique strings to change and your rule is not going to catch the new code. This is what I call a low-resilience rule. Instead, where it makes sense, I like to write rules which are harder for an attacker to evade. This often relies upon the more advanced features of YARA and is possibly more time consuming to write.

3. The "more advanced" features are newer, possibly buggier, and a little harder to wrap your brain around. The YARA syntax around strings and how to use them in conditions is easier for a non-programmer to understand.

4. YARA is only a small piece in the chain when it comes to defense. If you can use a "less resilient" rule to catch a piece of malware and unravel the entire kill chain from there, you can find more resilient ways to track the actor in the future that doesn't rely solely upon unchanging malware.

5. Lastly, and this is just a counter-point to the arguments above, I find that if you talk to people privately, they may have a better rule that they don't want to share publicly. So, people do write really nice rules but are keeping them amongst trusted peers because it is a more resilient rule (Anonymous, personal correspondence, May 11, 2017).

Most of the existing literature on this topic does not explore the more advanced aspects of YARA. This research will show that the utilizing both basic and advanced YARA features results in better identification of malware. This research will propose that more YARA users should take the time to learn about these advanced features and incorporate them into their rules. Additionally, more documentation needs to be produced by the YARA-using community that details use cases for advanced YARA rules and how to use them more effectively.

Christopher S. Culling, csculling1@gmail.com
2. Research Method

The research for this paper was conducted on a fresh installation of Linux Ubuntu 18.04 running in VMWare Workstation 14 Professional. All of the updates, upgrades, and installations of required components for both Ubuntu and the software used in this research were made as needed.

2.1. Tools to Aid in Writing and Executing YARA Rules

Malware must first be analyzed to determine its contents and attributes before YARA rules targeting that specific malware can be created. There are numerous tools that can be utilized to do this and which also can aid in the writing and execution of YARA rules. Several of them were used during this research, including the YARA tool (described in the Introduction), yarGen (Roth, 2018), pe (Te-k, 2018), Simple Static Malware Analyzer (SSMA) (Khasaia, 2018), and Joe Sandbox Cloud (Joe Security, 2018). Several collections of tools that analysts can use in the examination of malware, which were not used for this paper but are worth mentioning, are REMnux and the SANS Investigative Forensics Toolkit (SIFT). REMnux, a “free Linux toolkit for assisting malware analysts with reverse-engineering malicious software” (Zeltser, n.d.) is an excellent open-source platform for users who are interested in malware reverse-engineering and analysis. SIFT is also an excellent open-source collection of incident response and forensic tools that can be incorporated into REMnux (SANS, 2018). Users should be sure to update and upgrade the tools in both collections before first use.

While REMnux and SIFT contain a multitude of different tools, and are excellent resources, starting with YARA, yarGen, pe, SSMA, and Joe Sandbox Cloud (or another open-source malware sandbox) can provide plenty of data from which to begin writing quality YARA rules.

2.1.1. yarGen

yarGen is a YARA rule generator used in this research, which when run against a file, will output potential malware strings. What separates it from other YARA-related tools is the large goodware strings and opcode database that comes with it. These features allow for the distinction between malware strings and strings that can, for the most part,
be ignored. yarGen then takes its output and generates a YARA rule for the file, and possibly a super rule when scanning multiple, similar files at the same time (Roth, 2018). yarGen outputs rules that are sufficient to use as-is. However, to optimize them so that they are "sufficiently generic" to match more than one sample, users should read the three-part series entitled, “How to Write Simple but Sound Yara Rules” (Roth, 2015a, 2015b, 2016a).

2.1.2. pe

pe is a tool that delves into the Portable Executable (PE) file, which is found within several different file types and contains information that allows the Windows Operating System loader to work with the wrapped executable code (Revers3r, 2018). pe can extract data from a PE file, search for a string within a PE file, or check to see if anything in the PE file is out of the ordinary (Te-k, 2018).

2.1.3. Simple Static Malware Analyzer (SSMA)

SSMA is a simple analyzer that provides static malware analysis. One of its many capabilities is to scan the malware with its comprehensive YARA rules database which searches for the existence of well-known software packers, cryptographic algorithms and evasion processes, and looks for Windows functions commonly used by malware (Khasaia, 2018).

2.1.4. Joe Sandbox Cloud

Joe Sandbox Cloud is a dynamic malware analyzer which “executes files […] in a controlled environment and monitors the behavior of applications and the operating system for suspicious activities” and produces comprehensive reports in multiple formats (Joe Security, 2018). Appendix C contains the full report of a scan of one of the malware samples to show the amount of information one of these reports can provide. This report can be used to create YARA rules, determine firewall rules, and take various other network defense measures.

2.2. Static Analysis of Malware Samples

The malware samples used for this research consisted of six samples of Equation Group’s malware strain EquationLaser, and 261 samples of their FannyWorm malware.
strain (Shalev, 2017). Equation Group is thought to have been formed anywhere between 1996 and 2002 and has infected systems in multiple sectors around the world ever since (GReAT, 2015). EquationLaser malware was last seen in use between 2001 and 2004, while FannyWorm was on the scene from 2008 to 2011 (Zetter, 2015).

Static analysis of these samples was conducted using yarGen to produce initial YARA rules. yarGen created an individual rule for each piece of EquationLaser malware along with one super rule for the group, along with 37 individual rules and one super rule from the FannyWorm samples. SSMA and pe were then run against each piece of EquationLaser malware and six randomly-chosen FannyWorm samples to discover the internal characteristics of each against which basic and advanced YARA rules could be crafted.

2.3. Dynamic Analysis of Malware Samples

The research then progressed to dynamic analysis of the malware samples by scanning the six previously analyzed files from each malware strain with Joe Sandbox Cloud.

The results of the static and dynamic analysis of the malware samples and the associated analysis of the data generated from the tools used to analyze the malware are detailed in the following section.

3. Findings and Discussion

Many of the articles reviewed during the research for this paper regarding YARA rules often rehash the official documentation posted by YARA’s creator, Victor Alvarez. And even those articles primarily discussed the more basic aspects of YARA. Additionally, there were no relevant articles in the EBSCOhost research database and very few scholarly articles in Google Scholar on the topic, most of which only mentioned the existence of YARA rules. There were, however, a handful of YARA superusers, such as Florian Roth and Ricardo Dias, who wrote about how to utilize YARA’s more advanced functions and who described uses of particular features in ways not found in...
Mr. Alvarez's original YARA documentation. Their writings will form the basis for much of this research paper and future research.

The most current official YARA documentation can be found in HTML (VirusTotal, n.d.) or PDF format (Alvarez, 2018). It covers YARA installation, how to write YARA rules, YARA modules (add-on features with advanced functionality), how to write modules, running YARA from the command line and via Python, and utilizing the C API to integrate YARA into C/C++ projects.

While the intent of this paper is not to teach users how to use YARA, the concept of how YARA rules work is necessary to understand the research that was conducted. To that end, the writing of YARA rules, executing YARA rules, and what would constitute basic rules and advanced rules will be covered, as it is presented in the official documentation and by several YARA superusers. For instructions on how to install YARA, and for a full description of all of YARA’s capabilities, see the official documentation (Alvarez, 2018).

3.1. Introduction to Writing YARA Rules

Every YARA rule begins with the keyword rule, followed by the name of the rule. The rule itself is enclosed by curly brackets { }, within which lies the parameters of the rule. Rules are primarily made up of two sections. The first, which contains specific strings (text, hexadecimal, or regular expressions), can be omitted if the rule does not include a string. The second, the condition, which will define what triggers the rule, is a requirement for all YARA rules. A simple example rule, taken from the official YARA documentation (Alvarez, 2018) appears as follows:

```
rule ExampleRule
{
    strings:
        $my_text_string = "text here"
        $my_hex_string = { E2 34 A1 C8 23 FB }
```

Christopher S. Culling, csculling1@gmail.com
condition:

$my_text_string or $my_hex_string

If a file that contained either the identified text or the specific hex string had this rule run against it, it would indicate a match, due to the use of or in the condition. If the text or hex string were located within a piece of malware, YARA would indicate that it made a positive match.

Rules can also have comments added to them following C coding comment rules (Alvarez, 2018):

```c
/*
   This is a multi-line comment ...
*/

rule CommentExample  // ... and this is single-line comment
```

### 3.1.1. Strings

Three types of strings are allowed in YARA rules: hexadecimal, text, and regular expression (Alvarez, 2018). A basic YARA rule would be one that primarily relied on the use of strings to identify a piece of malware.

Hexadecimal strings can be used with wild-cards, jumps, and alternatives. An example of using wild-cards (or placeholders signified by a question mark) in a rule is as follows:

```yara
rule Example_Wildcard
{
    strings:
        $a1 = { 55 3? AB ?? 67 }

    $a1 = { 55 3? AB ?? 67 }
```

Christopher S. Culling, csculling1@gmail.com
When a user knows the exact number of missing hex characters, wild-cards are the option to use. However, when the exact number of missing characters is not known, jumps would be used instead of wild-cards. Jumps follow the pattern of (Alvarez, 2018):

\[ [X - Y] \text{ where } 0 \leq X \leq Y \]

For example (Alvarez, 2018):

```yara
rule JumpExample
{
    strings:
        $hex_string = { F4 23 [4-6] 62 B4 }
    condition:
        $hex_string
}
```

In this case, either four, five, or six sets of hex characters could be contained within the \[ \] brackets.

Alternative hex strings resemble regular expressions, such as this example:

```yara
rule Example_Hex_String
{
    strings:
        $hex_string = { AB 23 ( 62 5? | 65 | 8C ?? ?? ) 21 }
    condition:
        $hex_string
}
```

Christopher S. Culling, csculling1@gmail.com
In addition to hex strings, text strings may be used. The simplest use of a text string would be the following:

```yara
rule Example_Text_String
{

    strings:
        $a1 = "Missouri"

    condition:
        $a1
}
```

The following modifiers can appear at the end of a text string (Alvarez, 2018):

- `nocase` = makes the text string, which is normally case-sensitive, case-insensitive
- `wide` = searches for text strings encoded with two bytes per character
- `ascii` = searches for text strings in ascii format (this is the assumed default)
- `xor` = searches for text strings with a single byte XOR applied
- `fullword` = only matches text string if delimited by non-alphanumeric characters

An example of the use of some of these modifiers is as follows:

```yara
rule ModifierTextExample
{

    strings:
        $wide_and_nocase_string = "Texas" wide nocase

    condition:
        $wide_and_nocase_string
}
```

Christopher S. Culling, csculling1@gmail.com
This rule would indicate a positive match if the word “Texas” was encoded with two bytes per character and if it appeared in any form of upper and lower-case characters.

Regular expressions can also be used as strings and are enclosed in forward slashes / instead of quotes like the text strings. The specific regular expression syntax allowed when creating a YARA rule can be found in the official documentation (Alvarez, 2018). While regular expressions provide a wide range of flexibility when creating rules, they should be used sparingly as they significantly slow down YARA’s evaluation of the target file. Users should try to use hex strings with wild-cards and jumps if they can be used instead (Roth, 2016b).

3.1.2. Conditions

The second part of a YARA rule, and the only required component within the rule, is the condition. Conditions are Boolean expressions that contain the operators and, or, and not, relational operators such as >= and ==, arithmetic operators, and bitwise operators, such as >>. Conditions define what will cause the rule to activate on the target file, folder, or process (Alvarez, 2018).

For example, in the following rule, the condition defines what strings will return a positive hit on the target:

```
rule Example_Condition
{
  strings:
    $string1 = "text1"
    $string2 = "text2"
    $string3 = "text3"
    $string4 = "text4"
  condition:
    ($string1 or $string2) and ($string3 or $string4)
}
```

Christopher S. Culling, csculling1@gmail.com
In this case, if the string “a” or” b” and the string “c” or “d” are present in the target, YARA will indicate their presence.

3.1.3. Metadata

In addition to strings and conditions, rules can also contain metadata information. The only use of the metadata section is to store additional data about the rule and is indicated by the word meta. Similar to strings, each piece of metadata begins with an identifying phrase, followed by an equals sign, followed by the information. The following shows how the metadata section is used (Roth, 2015a):

```
rule Enfal_Generic
{
    meta:
        description = "Auto-generated rule - from 3 different files"
        author = "YarGen Rule Generator"
        reference = "not set"
        date = "2015/02/15"
        super_rule = 1
        hash0 = "6d484daba3927fc0744b1bbd7981a56ebeff95790"
        hash1 = "d4071272cc1bf944e3867db299b3f5dce126f82b"
        hash2 = "6c7c8b804cc76e2c208c6e3b6453cb134d01fb41"
```

Once the user has defined the strings (based on the analysis of the malware sample), and has determined the conditions and any optional metadata, he or she is ready to run the rule(s) against the target.

3.2. Executing YARA Rules

To run YARA against a file, folder, or process, the user would apply the following command line syntax (obtained via the “yara -h” command):

Christopher S. Culling, csculling1@gmail.com
The scan uses rules that can be found in source code or be compiled. One or multiple YARA rule files can be run against the target. More in-depth details and examples regarding how to execute YARA rules are found in the official YARA documentation (Alvarez, 2018).

### 3.3. Basic YARA Rules

As previously stated, basic YARA rules search for predefined strings within the target file, folder, or process. These rules are primarily concerned with the detection of a signature within the target that matches the assigned string or strings.

An example of a basic rule would be the following (AlienVault Labs, 2017):

```yara
rule LIGHTDART_APT1
{
    meta:
        author = "AlienVault Labs"
        info = "CommentCrew-threat-apt1"
}
```

Christopher S. Culling, csculling1@gmail.com
strings:

$s1 = "ret.log" wide ascii
$s2 = "Microsoft Internet Explorer 6.0" wide ascii
$s3 = "szURL Fail" wide ascii
$s4 = "szURL Successfully" wide ascii
$s5 = "%s&sdate=%04ld-%02ld-%02ld" wide ascii

condition:

all of them

An example of a basic rule with a more complex condition is (AlienVault Labs, 2017):

rule CCREWBACK1

{

meta:

    author = "AlienVault Labs"
    info = "CommentCrew-threat-apt1"

strings:

    $a = "postvalue" wide ascii
    $b = "postdata" wide ascii
    $c = "postfile" wide ascii
    $d = "hostname" wide ascii
    $e = "clientkey" wide ascii
    $f = "start Cmd Failure!" wide ascii
    $g = "sleep:" wide ascii
$h = "downloadcopy:" wide ascii

$i = "download:" wide ascii

$j = "geturl:" wide ascii

$k = "1.234.1.68" wide ascii

condition:

4 of ($a,$b,$c,$d,$e) or $f or 3 of ($g,$h,$i,$j) or $k

While there are many useful rules in this ruleset (70 rules in total targeting APT1 malware), none of them move beyond this paper’s definition of a basic rule.

3.4. Advanced YARA Rules

Advanced YARA rules, as opposed to basic rules, are geared more toward the behavior or characteristics of the target, versus a string-based signature. They are designed to be more "resilient," making it harder for an attacker to evade them (Anonymous, personal correspondence, May 11, 2017).

While many advanced rules may still search for strings, they will contain additional features in the condition section. As previously mentioned, YARA rules do not require any strings to be considered a valid rule and can run on condition statements alone. However, if the user does decide to create strings, which strings they use, the relative importance applied to each one, and how they apply conditions to them can also elevate a rule from a basic to an advanced level (Roth, 2015a, 2015b, 2016a).

3.4.1. Magic Number

One condition variable that can elevate a rule from basic to advanced is the magic number variable. The magic number is used by applications and operating systems to determine the type of file with which it is working and is located at the beginning of the file. For example, the hex value 4D 5A at the beginning of a file indicates a Windows/DOS executable file. The values 4D 5A in hex equate to the characters MZ, or the initials of Mark Zbikowski, the individual who designed the DOS executable file.

Christopher S. Culling, csculling1@gmail.com
format. Additionally, the hex values 25 50 44 46 at the beginning of a file would indicate that the file is a PDF. Therefore, if the file type is known when the user is crafting the YARA rule, the addition of the magic number variable in the condition will allow the rule to ignore those files which don't match, speeding up the search process. There are many locations on the Internet where lists of file types and their matching hex signatures can be found, with one very comprehensive list that is maintained by Gary Kessler (2018).

3.4.2. Locating Data at a Given Offset or Virtual Address

YARA uses the following functions to search for a particular string or value at a given offset within a file or virtual memory address:

- `int8(<offset or virtual address>)`
- `int16(<offset or virtual address>)`
- `int32(<offset or virtual address>)`
- `uint8(<offset or virtual address>)`
- `uint16(<offset or virtual address>)`
- `uint32(<offset or virtual address>)`
- `int8be(<offset or virtual address>)`
- `int16be(<offset or virtual address>)`
- `int32be(<offset or virtual address>)`
- `uint8be(<offset or virtual address>)`
- `uint16be(<offset or virtual address>)`
- `uint32be(<offset or virtual address>)`

The official YARA documentation describes this functionality as:

The `intXX` functions read 8, 16, and 32 bits signed integers from `<offset or virtual address>`, while functions `uintXX` read unsigned integers. Both 16 and 32-bit integers are considered to be little-endian. If you want to read a big-endian integer use the corresponding function ending in `be`. The

Christopher S. Culling, csculling1@gmail.com
parameter can be any expression returning an unsigned integer, including the return value of one the `uintXX` functions itself (Alvarez, 2018).

As some analysts may have no problem understanding how to use this feature, many may not. To that end, the following use case is provided to show how this powerful function may be effectively utilized.

If the malware that needed to be detected was a Windows executable, the MZ file signature (indicating a Windows/DOS file) and PE file signature (indicating an executable file) hex values would both need to be located and matched. A YARA rule written to accomplish this would appear as such (Alvarez, 2018):

```yara
rule IsPE
{
    condition:
    // MZ signature at offset 0 and...
    uint16(0) == 0x5A4D and
    // ... PE signature at offset stored in MZ header at 0x3C
    uint32(uint32(0x3C)) == 0x00004550
}
```

As the first comment after the condition statement above indicates, the MZ file signature ((which is a two-byte, unsigned integer (uint16) and little-endian)) should be located at file offset 0 and will be written in reverse order in the rule due to its endianness (5A4D versus 4D5A, or ZM versus MZ). The example graphic below, which puts this process into perspective (Wikibooks, 2018), shows that this is an MZ file (note the 4D 5A located at offset 0). Next, the hex for the PE file signature ((which is a four-byte, unsigned integer (uint32) and little-endian)), when translated reads PE/0/0 (or 00EP, as shown in the example above due to its endianness). The uint32(0x3C) address is first located in the MZ header and contains the hex value D8. If this is an actual PE file, location 0xD8 should contain the PE file indicator 0x50450000.

Christopher S. Culling, csculling1@gmail.com
As we can see from the above graphic, the hex value 0xD8 located at offset uint32(0x3C) does, in fact, point to the hex value for a PE file. Adding this short but effective condition to all YARA rules that are designed to detect Windows executable files can increase its effectiveness by cutting down on false positives and speeding up the detection process.

3.4.3. Filesize

Another advanced condition variable is the filesize variable. This variable can only be used with targets that are files and that can be appended with KB or MB which will multiply the number by 1024 or 2^20, respectively (Alvarez, 2018). An example of the filesize variable follows:

rule Example_Filesize {

    condition:

    filesize <= 300KB

}

In the above example, this rule will detect any file that is less than or equal to 300KB. As many pieces of malware are often quite small, defining the size of the file that is being detected can greatly increase the speed at which YARA performs its search, as the search pool has just been reduced.

Christopher S. Culling, csculling1@gmail.com
3.4.4. Portable Executable (PE) Module

YARA has external modules that provide additional functionality on top of the base program. These include the PE, Executable and Linkable Format (ELF), Cuckoo, Magic, Hash, Math, dotnet, and Time Modules. Due to the length constraints of this research paper, only the PE module will be explored, but it is recommended to study the other modules and their uses from the official YARA documentation (Alvarez, 2018).

The PE module is an excellent place to start creating advanced YARA rules as the various tools discussed in this paper can yield a vast amount of information found in the PE header against which to write YARA rules. As Alvarez states, “The PE module allows you to create more fine-grained rules for PE files by using attributes and features of the PE file format. This module exposes most of the fields present in a PE header and provides functions which can be used to write more expressive and targeted rules” (Alvarez, 2018). The vast amount of condition statements that can be crafted into YARA rules regarding fields and characteristics of a PE file that stem from the PE module makes this resource an important one to learn and incorporate into advanced YARA rules.

To use the PE module in a YARA rule, or set of rules, the user must first activate the module by adding the command `import "pe"` to the start of the rule file. The arguments used with the PE module all begin with `pe` and are found within the `condition` section of the rule.

An example of the PE module usage in a rule follows (Alvarez, 2018):

```plaintext
import "pe"
rule single_section
{
    condition:
        pe.number_of_sections == 1
}
rule control_panel_applet
{

Christopher S. Culling, csculling1@gmail.com
```
condition:

pe.exports("CPlApplet")

}

rule is_dll
{
    condition:

    pe.characteristics & pe.DLL

}

There are PE module arguments whereby the data for the argument can be easily collected using tools such as SSMA and pe. One argument that will play a prominent role in the findings component of this research is `pe.imphash`, which refers to the PE file’s import hash. As found in the FireEye security blog, an unnamed writer from the company Mandiant states “Imports are the functions that a piece of software calls from other files (typically various DLLs that provide functionality to the Windows operating system) (Mandiant, 2014). Additionally, they go on to explain that the imphash can be used to identify malware samples that are related (Mandiant, 2014).

Another argument, `pe.entry_point`, refers to the address where the PE loader starts to run the executable portion of the file (Revers3r, 2018). This is a common location for software packers to begin their code. Both the entry point and imphash values can be found using the `info` argument when running the tool `pe`. If the number of imports or exports in the PE is known, `pe.number_of_imports` or `pe.number_of_exports` can be used. The official YARA documentation contains nine pages of PE arguments that can be used in the condition statement of a rule, and the PE module is a good place to start learning about YARA module usage and capabilities.

3.4.5. YARA Performance Guidelines

The guidance provided in “YARA Performance Guidelines” (Roth, 2016b), covers ways to craft YARA rules to achieve the highest level of performance from them. This section has already touched upon several of the topics. Some of the additional

Christopher S. Culling, csculling1@gmail.com
subjects Roth covers are global rules, the most efficient ways to write strings, and condition statements which use a newer YARA feature called *short-circuit evaluation*, which can potentially improve the execution time of a YARA rule depending on the order in which the condition statement is written.

### 3.4.6. Advanced YARA Rules Use Case

In Ricardo Dias’ three-part series, “Unleashing YARA” (2016a, 2016b, 2016c), he discusses the usefulness of YARA in an Incident Response Team and walks the reader through a very detailed, advanced YARA use case. This is highly recommended reading for any user who is serious about improving their YARA rule writing abilities.

### 3.5. Static Analysis Findings

#### 3.5.1. yarGen Findings

For the initial static malware analysis, yarGen was run against the six EquationLaser malware samples using the [-z 0] option to see both malware and goodware strings. After removing the goodware strings, the generated YARA rules for all six samples were the same, including the super rule for the set. What follows is the super rule, which was edited to remove non-essential information, the goodware strings, and the condition, which previously contained “and 8 of them” (referring to the strings) and was changed to “and all of them” as was found in the individual rules:

```yara
rule _EquationLaser {
    meta:
        description = "EquationLaser"
        author = "yarGen Rule Generator"
        reference = "https://github.com/Neo23x0/yarGen"
        date = "2018-06-30"
    hash1 = "5e97f0cc3407c56ee5e6233b7573bd6eb05ffe22949bd12c1d1a26b2ab21d827"
}
```

Christopher S. Culling, csculling1@gmail.com
Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com
rule FannyWorm {

meta:

    author = "yarGen Rule Generator"
reference = "https://github.com/Neo23x0/yarGen"
date = "2018-06-30"
hash1 - hash 261
description = "FannyWorm"

strings:

    $x1 = "c:\windows\system32\kernel32.dll" fullword ascii
    $s2 = "\shelldoc.dll" fullword ascii
    $s3 = "\system32\win32k.sys" fullword wide
    $s4 = "dll_installer.dll" fullword ascii
    $s5 = "32.exe" fullword ascii
    $s6 = "Global\RPCMutex" fullword ascii
    $s7 = "System\\CurrentControlSet\\Services\\PartMgr\\Enum" fullword ascii
    $s8 = "x:\fanny.bmp" fullword ascii
    $s9 = "d:\fanny.bmp" fullword ascii
    $s10 = "Q:\__?__.lnk" fullword ascii
    $s11 = "=:=2=:=d=" fullword ascii /* hex encoded string '-' */
    $s12 = "fseek(SEEK_SET) failed" fullword ascii
    $s13 = "file size = %d bytes" fullword ascii
    $s14 = "4%5F5J5N5R5V5Z5^5b5f5j5n5r5v5z5~5" fullword ascii
    $s15 = "Software\Microsoft\MSNetMng" fullword ascii

condition:

Christopher S. Culling, csculling1@gmail.com
uint16(0) == 0x5a4d and filesize < 500KB and pe.imphash() ==
"1f5e76572fad36553733428ca3571f53" and all of them
}

Some observations about the yarGen results:

1. yarGen identified and added the MZ file signature (uint16(0) == 0x5a4d) and
filesize to each malware sample, reinforcing the importance of using those
conditions when writing rules.

2. Each set of malware samples, regardless of whether or not the individual rules
contained the same strings or not, had the same PE import hash (pe.imphash).
This indicates that the behavior of each file was the same, regardless of the file’s
overall hash value or the individual strings which were identified for it. A YARA
rule containing only that condition should provide a positive match every time on
a piece of malware from that particular strain.

3. Removing the pe.imphash statement from the condition did not change the
overall results in that each super rule positively identified every malware sample
in each set as being malware. However, because each sample was able to be
analyzed prior to running YARA rules against it, the advantage fell to the
researcher. Future variants of either of these malware strains may have different
identifying strings, which would not be caught by the strings used in this research.
However, because the PE import hash remained constant, that condition alone
would more than likely to catch future variants.

3.5.2. SSMA Findings

The SSMA scans for all six EquationLaser and six randomly selected
FannyWorm malware samples were identical within each group. Examples of the scan
results are found in Appendices A and B. The FannyWorm sample hashes were as
follows:

f4bff0768e2e548aad03a51b00077c30c1865c54385b060ed8f4325812da13aa
d3b1ea2ef9bf92af1c15f8a0426a73f8e43ceff2f35695d316d41991e5116c3d

Christopher S. Culling, csculling1@gmail.com
The SSMA results for EquationLaser pointed out a suspicious PE file `.data` section size, two PE file sections (.data and Shared) with either very high or very low entropy numbers (indicating compression or encryption), a PE file section suspiciously named Shared, and the presence of four bytes of overlay data, or extra data often associated with malware. SSMA then lists a number of Windows functions commonly used by malware and is followed by positive matches using its internal YARA rule collection. SSMA’s YARA rules were positive for the existence of well-known malware, software packers, cryptographic algorithms, and anti-debug/anti-virtualization processes within the malware samples.

The SSMA results for FannyWorm contained less information than for EquationLaser. However, both sets of malware almost had the same positive hits from the YARA rule scans, which is a strong indication that they both belong to the same overall malware family.

One point to highlight from the SSMA findings is that SSMA uses a YARA rules database that is not designed to discover specific strains of malware but instead is designed to identify typical characteristics of malware, such as the presence of software packers and cryptographic algorithms.

### 3.5.3. pe Findings

The following graphic shows the findings of a pe scan on one of the EquationLaser samples. The usage, as shown at the top of the below figure, is easy to use. For this research, the arguments check, search, checksize, and info were used. While it shows similar data as SSMA, one new piece of information that it provides is the PE file `entrypoint` when using the info argument. The second figure below shows the beginning of the Imports section, which contains information that is useful for the PE Module, and

Christopher S. Culling, csculling1@gmail.com
the third figure shows the six files that every EquationLaser malware sample exported during execution.

```
csculling@ubuntu$: pe -h
usage: pe [-h] {shell,check,dump,search,info,checksize} ...

positional arguments:
  {shell,check,dump,search,info,checksize}
    shell  Launch python shell to analyze the PE file
    check  Check for stuff in the file
    dump   Dump resource or section of the file
    search  Search for a string in a PE file
    info    Extract info from the PE file
    checksize  Check size of the PE file

optional arguments:
    -h, --help    show this help message and exit

Running checks on /home/csculling/Desktop/EquationLaser/EquationLaser_OE2C80852F530C9F852C2743A5BB2BA:
[1] Abnormal section names: Shared
[1] Suspicious entropy in the following sections:
  .data     7.01632
  .shared   0.00000
[1] 4 extra bytes in the file
[1] PE ID packer: ArnadilloVxSx

csculling@ubuntu$: pe check /home/csculling/Desktop/EquationLaser/EquationLaser_OE2C80852F530C9F852C2743A5BB2BA

Position in the file : 0x270

Running searches:
  Shared /home/csculling/Desktop/EquationLaser/EquationLaser_OE2C80852F530C9F852C2743A5BB2BA

Name     VirtSize VirtualAddr RawSize RawAddr Entropy md5
.text     0x1a00 0x1000 0x1a00 0x1000 6.6792 526206a0ac6ac6c2f59653892f4ec
.data     0x1b5f 0x1c00 0x1ae0 0x1e00 5.3139 c595146db5a7811cd6b2a24d4d22b4e
.data     0x4bfb8 0x1e00 0x1c00 0x1e00 7.1016 4356caba9211586296374e65c32cc533
.Shared   0x118 0x6a00 0x1c00 0x200 0.0000 bf619eac0df36460e9a9344173d8
.rsrc     0x1418 0x6b00 0x1e20 0x600 2.5508 fodd3c7f9f2a47b24825f76372eeb79987
.relloc   0x15c6 0x6f00 0x1e80 0x1e00 5.7657 6abbbbb5d83042ccddc319c977713517

4 bytes of extra data (132612 while it should be 132698)

csculling@ubuntu$: pe info /home/csculling/Desktop/EquationLaser/EquationLaser_OE2C80852F530C9F852C2743A5BB2BA

Metadata

MDS: 0e2c80b2f530c9f852c2743a5bb2ba
SH1: 8edeec4f4f4b77243565fd3d3a290e4900e
SH2: aef712f9efbf8f131f8b971f644c3c4b08da315c4487d80e4acaab01c8493a3
ImpHash: 33450e6b90d95724e1d3e6888
Size: 132612 bytes
Type: PE32 executable (DLL) (GUI) Intel 80386, for MS Windows

DLL File:
  Compile Time: 2004-10-18 12:24:05 (UTC -0x17385E5)
  Entry point: 0x1001b801 (section .text)

Sections

Name     VirtSize VirtualAddr RawSize RawAddr Entropy md5
.text     0x1a00 0x1000 0x1a00 0x1000 6.6792 526206a0ac6ac6c2f59653892f4ec
.data     0x1b5f 0x1c00 0x1ae0 0x1e00 5.3139 c595146db5a7811cd6b2a24d4d22b4e
.data     0x4bfb8 0x1e00 0x1c00 0x1e00 7.1016 4356caba9211586296374e65c32cc533
.Shared   0x118 0x6a00 0x1c00 0x200 0.0000 bf619eac0df36460e9a9344173d8
.rsrc     0x1418 0x6b00 0x1e20 0x600 2.5508 fodd3c7f9f2a47b24825f76372eeb79987
.relloc   0x15c6 0x6f00 0x1e80 0x1e00 5.7657 6abbbbb5d83042ccddc319c977713517
```
3.6. Dynamic Analysis Findings

3.6.1. Joe Sandbox Cloud Findings

While an analyst can obtain a lot of useful information by performing a static analysis of a piece of malware, more data may be found when they dynamically analyze the malware by executing it in a contained environment.

Christopher S. Culling, csculling1@gmail.com
The results of the analysis performed by Joe Sandbox Cloud were quite detailed. The reports revealed many different malware characteristics from which quality YARA rules could be generated. For example, they list files that the malware may drop onto the target computer which can then be separately analyzed to create more detailed, granular YARA rules. Additionally, the reports reveal the characteristics of the malware while it is executing, providing more points of reference from which to create advanced YARA rules than static malware analysis alone can provide. A report of the analysis conducted on one of the FannyWorm malware samples can be found in Appendix C. Each set of malware that was run through the Sandbox produced mostly similar results. It is assumed that variations in results between the malware in each strain occurred because the malware was only run once and only for several minutes. Additionally, the malware contained malware analysis system evasion processes, anti-virus detection, and other protections, which could cause each malware sample to behave differently in the Sandbox, even if all of them essentially perform the same function.

While the results of the Joe Sandbox Cloud analysis of the 12 pieces of malware ultimately was not used to inform the final recommendations of this research, they do play a crucial part in providing information above and beyond what any static malware analysis could provide. For example, one Joe Sandbox report stated that the malware sample dropped PE files which had not been started and that the Sandbox should also run those files for analysis.

The amount of information that dynamic malware analysis provides that can be used in writing advanced YARA rules should not be overlooked and learning how to perform malware analysis should be part of any serious YARA rule-user’s skillset.

4. Recommendations and Implications

Upon beginning this research, the question - which YARA rules were more effective, basic or advanced - appeared to be an either/or proposition. However, as it turns out, the entire spectrum of YARA rules are needed to ensure complete coverage against malware threats.

Christopher S. Culling, csculling1@gmail.com
4.1. Recommendations for Use in the Field

Basic YARA rules can be easily assembled based on the first identified piece of malware in a matter of minutes— and they should be, in order to quickly deploy them into the ever-growing, various network defense components that accept YARA rules as one of their Indicator of Compromise (IOC) inputs (such as Tanium and Nessus). If they are not initially written in a manner with will limit false positives, they should eventually be updated accordingly. Guidance for doing this can be found in Florian Roth’s “How to Write Simple but Sound YARA Rules” series (2015a, 2015b, 2016a).

However, as the research has shown, the strings that basic YARA rules rely upon can change, making the current, basic YARA rules ineffective. To counter this, further analysis of the malware samples must be taken to understand their behavior and characteristics, which are less likely to change compared to their string signatures. Using the magic number and filesize parameters in every YARA rule written will provide an immediate advantage as those are variables that are unlikely to change over time. While strings may change, a malware’s core behavior should remain consistent. As the PE file contains the “brains” of the executable, and as the research has shown that it remains remarkably consistent within individual malware strains, utilizing YARA’s PE Module is an excellent, advanced usage of YARA. Breaking down the PE file with various tools such as SSMA, pe, and Joe Sandbox Cloud should yield a multitude of different attributes from which to craft advanced YARA rules.

Once more advanced, “resilient” rules are created for a malware strain, the chances of it slipping through a network’s defenses are lessened. And, as previously stated, YARA rules should also be tuned to perform most effectively (Roth, 2016b).

Lastly, YARA can be used proactively to scan the network to look for files that contain well-known software packers, cryptographic algorithms, and anti-debug/anti-virtualization techniques that malware may use to hide from discovery, as demonstrated by SSMA.

Christopher S. Culling, csculling1@gmail.com
4.2. Implications for Future Research

Developing a reference that contains multiple use cases involving all levels of YARA rules would be the most beneficial future YARA rule research. Robert M. Lee, who teaches the use of YARA rules in his SANS courses, states, “There’s a lot of functionality that folks aren’t aware of and many ways to use it that aren’t clearly documented or explored” (R. Lee, personal correspondence, May 12, 2017). While the official documentation explains how to use YARA, only a handful of YARA superusers have shown how to use YARA rules in specific instances or how to take true advantage of its advanced features. One document or site which captures use cases or YARA’s advanced features would be most useful, allowing researchers or analysts to determine which types of YARA rules would work best in their situation.

Another worthwhile subject for future research into YARA rules would include the development of best practices and techniques to employ YARA rules in threat hunting situations, as suggested by Robert M. Lee (R. Lee, personal communication, May 7, 2017). While YARA rules were initially developed mainly for malware classification and incident handling, they are adaptable enough to be used as one more tool in a red team’s arsenal.

Qualitative research that employs surveys to discover how the YARA-using community actually uses the rules would be another informative research topic, allowing for the assessment of gaps which could be explored in further research.

Documenting what tools exist that would benefit a malware researcher throughout the entire YARA rule-creation process and ranking them based on their effectiveness via comparative demonstrations and analysis, would also be useful to the YARA-using community.

Finally, Dr. Johannes Ullrich, SANS Senior Instructor (J. Ullrich, personal communication, March 5, 2018), suggested a worthwhile subject to explore would be the use of YARA rules to detect malware utilizing obfuscation techniques. This is an especially important area for research as malware is increasingly becoming more and more sophisticated in its makeup.

Christopher S. Culling, csulling1@gmail.com
5. Conclusion

More often than not, analysts who utilize YARA rules in their discovery and classification of malware resort to using the most basic features and functionality of YARA. This conclusion led to this paper’s research question: When attempting to identify malware, how much more effective, if at all, is the utilization of more complex, advanced YARA rules than the use of easier-to-write, basic rules?

According to the research conducted, the entire range of YARA rules, from basic to advanced, have their part to play when searching for malware, and every level of rule has value to add. Initially, developing basic rules to catch the first wave of a new malware strain might be all that’s needed. The ease and speed with which these rules can be created will allow network defenders to quickly add an additional layer of detection and protection to their networks. However, it should be emphasized that as malware evolves, and as different variants are created, they may not continue to be detected by YARA’s basic rules. In this case, the need to develop the skills to utilize YARA’s more advanced functionality by searching for characteristics of behavior versus string matches would be a worthwhile endeavor.

For a comprehensive list of YARA rules, tools, services, people, and much more, please see “A curated list of awesome YARA rules, tools, and people” (InQuest, 2018).
Which YARA Rules Rule: Basic or Advanced?

References


Christopher S. Culling, csculling1@gmail.com


Roth, F. (2018, February) *yarGen is a generator for YARA rules.* Retrieved from https://github.com/Neo23x0/yarGen/


Christopher S. Culling, csculling1@gmail.com
Appendix A

Simple Static Malware Analyzer Results - Equation Laser

Christopher S. Culling, cssculling1@gmail.com

© 2018 The SANS Institute, Author retains full rights.
Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com

as part of a victim survey or to select between different offsets for undocumented structures that have changed between different versions of Windows.

- LoadLibraryA: Loads a DLL into a process that may not have been loaded when the program started. Imported by nearly every Win32 program.

- CreateMutexA: Creates a mutual exclusion object that can be used by malware to ensure that only a single instance of the malware is running on a system at any given time. Malware often uses fixed names for mutexes, which can be good host-based indicators to detect additional installations of the malware.

- MapViewOfFile: Maps a file into memory and makes the contents of the file accessible via memory addresses. Launchers, loaders, and injectors use this function to read and modify PE files. By using MapViewOfFile, the malware can avoid using WriteFile to modify the contents of a file.

- CreateFileMappingA: Creates a new file or opens an existing file.

- CreateFileMappingA: Creates a handle to a file mapping that loads a file into memory and makes it accessible via memory addresses. Launchers, loaders, and injectors use this function to read and modify PE files.

- GetWindowsDirectoryA: Returns the file path to the Windows directory (usually C:\Windows). Malware sometimes uses this call to determine into which directory to install additional malicious programs.

- SetFileTime: Modifies the creation, access, or last modified time of a file. Malware often uses this function to conceal malicious activity.

- GetModuleHandleA: Obtains a handle to an already loaded module. Malware may use GetModuleHandle to locate and modify code in a loaded module or to search for a good location to inject code.

- WideCharToMultiByte: Used to convert a Unicode string into an ASCII string.

- LoadLibraryA: Loads a DLL into a process that may not have been loaded when the program started. Imported by nearly every Win32 program.

- OpenProcess: Opens a handle to another process running on the system. This handle can be used to read or write to the other process memory or to inject code into the other process.

- GetTempPathA: Returns the temporary file path. If you see malware call this function, check whether it reads or writes any files in the temporary file path.

- CreateFileA: Creates a new file or opens an existing file.

- CreateProcessA: Creates and launches a new process. If malware creates a new process, you will need to analyze the new process as well.

- CreateProcessA: Creates and launches a new process. If malware creates a new process, you will need to analyze the new process as well.

- GetStartupInfoA: Retrieves a structure containing details about how the current process was configured to run, such as where the standard handles are directed.

- GetTickCountA: Retrieves the number of milliseconds since bootup. This function is sometimes used to gather timing information as an anti-debugging technique. GetTickCount is often added by the compiler and is included in many executables, so simply seeing it as an imported function provides little information.

- SetWindowsHookExA: Sets a hook function to be called whenever a certain event is called. Commonly used with keyloggers and spyware, this function also provides an easy way to load a DLL into all GUI processes on the system. This function is sometimes added by the compiler.

- CallNextHookExA: Used within code that is hooking an event set by SetWindowsHookEx. CallNextHookEx calls the next hook in the chain. Analyze the function calling CallNextHookEx to determine the purpose of a hook set by SetWindowsHookEx.

- RegOpenKeyExA: Opens a handle to a registry key for reading and editing. Registry keys are sometimes written as a way for software to achieve persistence on a host. The registry also contains a whole host of operating system management settings information.

- OpenProcessToken: Opens a handle to another process running on the system. This handle can be used to read and write to the other process memory or to inject code into the other process.

- RegOpenKeyExA: Opens a handle to a registry key for reading and editing. Registry keys are sometimes written as a way for software to achieve persistence on a host. The registry also contains a whole host of operating system management settings information.

- RegQueryInfoKey: Opens a handle to a registry key for reading and editing. Registry keys are sometimes written as a way for software to achieve persistence on a host. The registry also contains a whole host of operating system management settings information.

- OpenProcessHandleA: Opens a handle to the service control manager. Any program that installs, modifies, or controls a service must call this function before any other service-manipulation function.

Scan file using Yara-rules. With Yara rules you can create a "description" of malware families to detect new samples.

For more information: https://virustotal.github.io/yara/

These Yara rules specialized on the identification of well-known malware.

Result:

Str.Win32.Winsock2.Library - Match Winsock 2 API library declaration
These YARA rules are aimed to detect well-known software packers, that can be used by malware to hide itself.

**Result:**
- Armadillo_v1xx_v2xx_addditional
- Microsoft_Visual_CPP_60_DLL_additonal
- Microsoft_Visual_CPP_v97_DLL
- Microsoft_Visual_CPP_v80v68_MFC
- Microsoft_Visual_CPP_60_DLL_Debug
- Armadillo_v1xx_v2xx
- Microsoft_Visual_CPP_v80_DLL
- Microsoft_Visual_CPP_60_DLL
- Microsoft_Visual_CPP_60

These YARA rules aim to detect the existence of cryptographic algorithms.

**Detected cryptographic algorithms:**
- CRC32_poly_constant - Look for CRC32 [poly]
- CRC32_table - Look for CRC32 table

These YARA Rules aim to detect anti-debug and anti-virtualization techniques used by malware to evade automated analysis.

**Result:**
- win_mutex - Create or check mutex
- win_registy - Affect system registries
- win_token - Affect system token
- win_private_profile - Affect private profile
- win_files_operation - Affect private profile
- win_hook - Affect hook table

Ups... That's all ;)}
Appendix B

Simple Static Malware Analyzer Results - FannyWorm

This file contains a list of Windows functions commonly used by malware.

For more information use the Microsoft documentation.

- **CreateMutexA** - Creates a mutual exclusion object that can be used by malware to ensure that only a single instance of the malware is running on a system at any given time. Malware often uses names for mutexes, which can be good host-based indicators to detect additional installations of the malware.

- **OpenMutexA** - Opens a handle to a mutual exclusion object that can be used by malware to ensure that only a single instance of malware is running on a system at any given time. Malware often uses names for mutexes, which can be good host-based indicators.

- **GetTempPathA** - Returns the temporary file path. If you see malware call this function, check whether it reads or writes any files in the temporary file path.

- **GetVersionExA** - Returns information about which version of Windows is currently running. This can be used as part of a victim survey or to select between different offsets for undocumented structures that have changed between different versions of Windows.

- **CreateMutexA** - Creates a mutual exclusion object that can be used by malware to ensure that only a single instance of the malware is running on a system at any given time. Malware often uses names for mutexes, which can be good host-based indicators to detect additional installations of the malware.

- **OpenMutexA** - Opens a handle to a mutual exclusion object that can be used by malware to ensure that only a single instance of malware is running on a system at any given time. Malware often uses names for mutexes, which can be good host-based indicators.

- **LoadLibraryExA** - Loads a DLL into a process that may not have been loaded when the program started. Imported by nearly every Win32 program.

- **CreateFileA** - Creates a new file or opens an existing file.

- **MapViewOfFile** - Maps a file into memory and makes the contents of the file accessible via memory addresses. Loaders, launchers, and injectors use this function to read and modify PE files. By using `MapViewOfFile`, the malware can avoid using `WriteFile` to modify the contents of a file.

- **CreateFileMappingA** - Creates a new file or opens an existing file.

Christopher S. Culling, csculling1@gmail.com
Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com

PE files.

**LoadLibraryA** - Loads a DLL into a process that may not have been loaded when the program started. Imported by nearly every Win32 program.

**GetProcAddress** - Retrieves the address of a function in a DLL loaded into memory. Used to import functions from other DLLs in addition to the functions imported in the PE file header.

**CreateFileA** - Creates a new file or opens an existing file.

**SetFiletime** - Modifies the creation, access, or last modified time of a file. Malware often uses this function to conceal malicious activity.

**CreateProcessA** - Creates and launches a new process. If malware creates a new process, you will need to analyze the new process as well.

**FindResourceA** - Used to find a resource in an executable or loaded DLL. Malware sometimes uses resources to store strings, configuration information, or other malicious files. If you see this function used, check for a .rsrc section in the malware's PE header.

**LoadResource** - Loads a resource from a PE file into memory. Malware sometimes uses resources to store strings, configuration information, or other malicious files.

**OpenProcessToken** - Opens a handle to another process running on the system. This handle can be used to read and write to the other process memory or to inject code into the other process.

**RegistryKeyExA** - Opens a handle to a registry key for reading and editing. Registry keys are sometimes written as a way for software to achieve persistence on a host. The registry also contains a whole host of operating system and application setting information.

Scan file using Yara-rules.

With Yara rules you can create a "description" of malware families to detect new samples.

For more information: https://virusTotal.github.io/yara/

These Yara rules specialised on the identification of well-known malware.

**Result:**

Str_Wln32_Wlnsck2_Library - Match Winsock 2 API library declaration

These Yara Rules aimed to detect well-known software packers, that can be used by malware to hide itself.

**Result:**

Armadillo_vixx_v2xx_additonal
Microsoft_Visual_Cpp_60_DLL_additonal
Microsoft_Visual_Cpp_V78_DLL
Microsoft_Visual_Cpp_V80V99_MFC
Microsoft_Visual_Cpp_60_DLL_Debug
Armadillo_vixx_v2xx
Microsoft_Visual_Cpp_v60_DLL
Microsoft_Visual_Cpp_60_DLL
Microsoft_Visual_Cpp_60

These Yara rules aimed to detect the existence of cryptographic algorithms.

Detected cryptographic algorithms:

**CRC32_poly_Constant** - Look for CRC32 [poly]

**CRC32_table** - Look for CRC32 table

These Yara Rules aimed to detect anti-debug and anti-virtualization techniques used by malware to evade automated analysis.

**Result:**

antib.threatExpert - Anti-Sandbox checks for ThreatExpert
win_mutex - Create or check mutex
win_registry - Affect system registries
win_token - Affect system token
win_files_operation - Affect private profile

Ups... That's all :)

Christopher S. Culling, csculling1@gmail.com

© 2018 The SANS Institute Author retains full rights.
Appendix C

Joe Sandbox Cloud - FannyWorm

ID: 596380
Sample Name: ZuDBYlOvt4

Christopher S. Culling, csculling1@gmail.com
Table of Contents

Analysis Report

Overview
- General Information
- Detection
- Confidence
- Classification
- Analysis Advice

Signature Overview
- AV Detection:
- Exploits:
- Persistence and Installation Behavior:
- Data Obliviation:
- System Summary:
- HIPS / PFW / Operating System Protection Evasion:
- Anti Debugging:
- Memory Analyze System Evasion:
- Hooking and other Techniques for Hiding and Protection:
- Language, Device and Operating System Detection:

Behavior Graph

Simulations
- Behavior and APIs

Antivirus Detection
- Initial Sample
- Dropped Files
- Unpacked PE Files
- Domains
- URLs

Yara Overview
- Initial Sample
- PCAP (Network Traffic)
- Dropped Files
- Memory Dumps
- Unpacked PEs

Joe Sandbox View / Context
- IPs
- Domains
- ASN
- Dropped Files

Screenshots

Startup

Created / dropped Files

Contacted Domains/Contacted IPs
- Contacted Domains
- Contacted IPs

Static File Info
- General
- File Icon
- Static PE Info

Network Behavior

Code Manipulations

Christopher S. Culling, csculling1@gmail.com
<table>
<thead>
<tr>
<th>Statistics</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior</td>
<td>16</td>
</tr>
<tr>
<td>System Behavior</td>
<td>16</td>
</tr>
<tr>
<td>Analysis Process: loaddl32.exe PID: 3548 Parent PID: 2916</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>General</td>
</tr>
<tr>
<td></td>
<td>File Activities</td>
</tr>
<tr>
<td></td>
<td>File Written</td>
</tr>
<tr>
<td>Analysis Process: rundll32.exe PID: 3556 Parent PID: 3548</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>General</td>
</tr>
<tr>
<td></td>
<td>File Activities</td>
</tr>
<tr>
<td></td>
<td>Registry Activities</td>
</tr>
<tr>
<td>Analysis Process: msupdate.exe PID: 3554 Parent PID: 3556</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>General</td>
</tr>
<tr>
<td></td>
<td>File Activities</td>
</tr>
<tr>
<td></td>
<td>Registry Activities</td>
</tr>
<tr>
<td>Analysis Process: rundll32.exe PID: 3568 Parent PID: 3564</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>General</td>
</tr>
<tr>
<td></td>
<td>File Activities</td>
</tr>
<tr>
<td>Analysis Process: rundll32.exe PID: 3682 Parent PID: 3546</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>General</td>
</tr>
<tr>
<td></td>
<td>File Activities</td>
</tr>
<tr>
<td>Disassembly</td>
<td>19</td>
</tr>
<tr>
<td>Code Analysis</td>
<td>19</td>
</tr>
</tbody>
</table>
### Analysis Report

#### Overview

<table>
<thead>
<tr>
<th>General Information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Jie Sandbox Version</td>
<td>23.0.0</td>
</tr>
<tr>
<td>Analysis ID</td>
<td>a5600e0</td>
</tr>
<tr>
<td>Start time</td>
<td>01:52:29</td>
</tr>
<tr>
<td>Jie Sandbox Product</td>
<td>Cloud</td>
</tr>
<tr>
<td>Start date</td>
<td>01.07.2018</td>
</tr>
<tr>
<td>Overall analysis duration</td>
<td>0h 3m 46s</td>
</tr>
<tr>
<td>Hypervisor based inspection enabled</td>
<td>True</td>
</tr>
<tr>
<td>Report type</td>
<td>light</td>
</tr>
<tr>
<td>Sample file name</td>
<td>DVENY3Y4 (remained file extension from none to dll)</td>
</tr>
<tr>
<td>Cookbook file name</td>
<td>csculling1.txt</td>
</tr>
<tr>
<td>Analysis system description</td>
<td>Windows 7</td>
</tr>
<tr>
<td>Number of analysed new started processes analysed</td>
<td>7</td>
</tr>
<tr>
<td>Number of non started drivers analysed</td>
<td>0</td>
</tr>
<tr>
<td>Number of existing processes analysed</td>
<td>0</td>
</tr>
<tr>
<td>Number of existing drivers analysed</td>
<td>0</td>
</tr>
<tr>
<td>Number of injected processes analysed</td>
<td>0</td>
</tr>
<tr>
<td>Technologies</td>
<td>HCA enabled, EDA enabled, HOC enabled</td>
</tr>
<tr>
<td>Analysis stop reason</td>
<td>Timeout</td>
</tr>
<tr>
<td>Detection</td>
<td>Malicious</td>
</tr>
<tr>
<td>Classification</td>
<td>malware.classический.человек.200</td>
</tr>
<tr>
<td>HCA Information</td>
<td>Successful, rate: 94%</td>
</tr>
<tr>
<td>ECA Information</td>
<td>Successful, rate: 100%</td>
</tr>
<tr>
<td>MOC Information</td>
<td>Successful, rate: 100% (good quality rate 64.7%)</td>
</tr>
<tr>
<td>Cookbook Comments</td>
<td>Adjust boot time, Connecting counters for adjusted boot time, Start process as user (medium integrity level)</td>
</tr>
<tr>
<td>Warnings</td>
<td>Show All</td>
</tr>
</tbody>
</table>

#### Detection

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Score</th>
<th>Range</th>
<th>Reporting</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold</td>
<td>76</td>
<td>6 - 100</td>
<td>Report FD / FN</td>
<td></td>
</tr>
</tbody>
</table>

© 2018 The SANS Institute, Author retains full rights.

Christopher S. Culling, csculling1@gmail.com
Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com
Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com
## Which YARA Rules Rule: Basic or Advanced?

<table>
<thead>
<tr>
<th>Rule</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process the Windows registered owner settings</strong></td>
<td></td>
</tr>
<tr>
<td><strong>HIPS / PPW / Operating System Protection Evasion:</strong></td>
<td></td>
</tr>
<tr>
<td>Contains functionality to inject threads in other processes</td>
<td></td>
</tr>
<tr>
<td><strong>Anti Debugging:</strong></td>
<td></td>
</tr>
<tr>
<td>Contains functionality to dynamically determine API calls</td>
<td></td>
</tr>
<tr>
<td><strong>Malware Analysis System Evasion:</strong></td>
<td></td>
</tr>
<tr>
<td>Found dropped PE file which has not been started or loaded</td>
<td></td>
</tr>
<tr>
<td>Found large amount of non-executed APIs</td>
<td></td>
</tr>
<tr>
<td>Program exit points</td>
<td></td>
</tr>
<tr>
<td>Queries a list of all running drivers</td>
<td></td>
</tr>
<tr>
<td>Queries a list of all running processes</td>
<td></td>
</tr>
<tr>
<td><strong>Hooking and other Techniques for Hiding and Protection:</strong></td>
<td></td>
</tr>
<tr>
<td>May modify the system service descriptor table (often done to hook functions)</td>
<td></td>
</tr>
<tr>
<td>Disables application error messages (SelfDestrWrite)</td>
<td></td>
</tr>
<tr>
<td><strong>Language, Device and Operating System Detection:</strong></td>
<td></td>
</tr>
<tr>
<td>Contains functionality to query windows version</td>
<td></td>
</tr>
</tbody>
</table>

**Behavior Graph**
Simulations

**Behavior and APIs**

<table>
<thead>
<tr>
<th>Time</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:53:21</td>
<td>API Interrupt</td>
<td>Sleep call for process: rundll32.exe modified</td>
</tr>
<tr>
<td>01:53:22</td>
<td>API Interrupt</td>
<td>Sleep call for process: mauplay.exe modified</td>
</tr>
</tbody>
</table>

**Antivirus Detection**

**Initial Sample**

<table>
<thead>
<tr>
<th>Source</th>
<th>Detection</th>
<th>Scanner</th>
<th>Label</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\Old\Y\Private\ deadly.dll</td>
<td>136%</td>
<td>Avira</td>
<td>TVDrp.Adrgent_vah</td>
<td></td>
</tr>
</tbody>
</table>

**Dropped Files**

<table>
<thead>
<tr>
<th>Source</th>
<th>Detection</th>
<th>Scanner</th>
<th>Label</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\dirt\Windows\sys\deadly.dll</td>
<td>136%</td>
<td>Avira</td>
<td>TVDrp.Adrgent_vah</td>
<td></td>
</tr>
<tr>
<td>C:\dirt\User1\mauplayLocal\temp\\deadly.exe</td>
<td>136%</td>
<td>Avira</td>
<td>TVDrp.Adrgent_vah</td>
<td></td>
</tr>
<tr>
<td>C:\Windows\System32\deadly.dll</td>
<td>136%</td>
<td>Avira</td>
<td>TVDrp.Adrgent_vah</td>
<td></td>
</tr>
</tbody>
</table>

**Unpacked PE Files**

<table>
<thead>
<tr>
<th>Source</th>
<th>Detection</th>
<th>Scanner</th>
<th>Label</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.mauplay.exe.40006.3.unpacked</td>
<td>136%</td>
<td>Avira</td>
<td>TVDrp.Adrgent_vah</td>
<td></td>
</tr>
<tr>
<td>7.1.33.mauplay.exe.40006.3.1.unpacked</td>
<td>136%</td>
<td>Avira</td>
<td>TVDrp.Adrgent_vah</td>
<td></td>
</tr>
</tbody>
</table>
### Yara Overview

**Initial Sample**

No yara matches

**PCAP (Network Traffic)**

No yara matches

**Dropped Files**

No yara matches

**Memory Dumps**

No yara matches

**Unpacked PEs**

No yara matches

### Joe Sandbox View / Context

**IPs**

No context

**Domains**

No context

**ASN**

No context

**Dropped Files**

No context

---

Christopher S. Culling, csculling1@gmail.com
Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com
Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com
## Which YARA Rules Rule: Basic or Advanced?

### C:\x3fcs

| Malicious: | false |
| Reputation: | low |

### C:\x3fcs.1

| Process: | C:\\user\\AppData\\roaming\\tempupdate.\exe |
| File Type: | data |
| Size (bytes): | 5127 |
| Timestamp (UTC): | 148244312899385 |
| Encrypted: | false |
| MD5: | 7160672b01b7a59213a9386f2e5f77f3269531 |
| SHA256: | D507755176C2B254643C83734E1AD6A6AFC1BC88B30B53C8F120A0221B25ED3F3361 |
| SHA1: | 86ebe3b70fd6c73a5a03ef6d6a193e1a6572d751269068f22e53041e35b5e01f77a7f07 |
| Malicious: | false |
| Reputation: | low |

### stdout

| Process: | C:\Windows\System32\cmd.exe |
| File Type: | ASCII text, with CRLF line terminators |
| Size (bytes): | 121 |
| Encrypted: | false |
| MD5: | C254A1846A5499A17A0B7E6BD0698B |
| SHA1: | 3E69C8CA2E2A04204D5E4A0159844AC3542 |
| SHA256: | 052724C2C1D061E33D2D6971E3D30E33618867AD66F7D3AB2837D615CA85 |
| SHA1: | FE1138C17F576109D797C1A088B8445E57034E69FAC1B59939882B148872CFB5615B4A4F23F8F48E40D |
| Malicious: | false |
| Reputation: | low |

### Contacted Domains/Contacted IPs

#### Contacted Domains

No contacted domains info

#### Contacted IPs

No contacted IP info

### Static File Info

**General**

- File type: PE 32 executable (DLL) [GUI] Intel x86, for MS Windows
- File size: 844320
- MD5: 1b27ec7238a3360a1366e8d534f0d
- SHA1: e10ebe5c530f9d2330230e9958303849
- SHA256: e84d1093f0e03639d5cc145117f3e1f7eb5c0e84b5a932c47f
- SHA1: 44557d28f5d7786b7795e7c8dedb279b8b256d
- SHA256: 6f06159a6f0f0b32b7727a116e6138a2e2547

---

Christopher S. Culling, csculling1@gmail.com
Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com
Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com
No network behavior found

Code Manipulations

Statistics

Behavior

System Behavior

**Analysis Process: loadlib32.exe PID: 3548 Parent PID: 2916**

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start time:</td>
</tr>
<tr>
<td>Start date:</td>
</tr>
<tr>
<td>Path:</td>
</tr>
<tr>
<td>Would execute (White):</td>
</tr>
<tr>
<td>Commandline:</td>
</tr>
<tr>
<td>Imagehash:</td>
</tr>
<tr>
<td>File size:</td>
</tr>
<tr>
<td>MD5 hash:</td>
</tr>
<tr>
<td>Has administrator privileges:</td>
</tr>
<tr>
<td>Programmed in:</td>
</tr>
<tr>
<td>Reputation:</td>
</tr>
</tbody>
</table>

**File Activities**

<table>
<thead>
<tr>
<th>File Written</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Path</td>
</tr>
</tbody>
</table>

Christopher S. Culling, csculling1@gmail.com
### Which YARA Rules Rule: Basic or Advanced?

Christopher S. Culling, csculling1@gmail.com
## Upcoming Training

<table>
<thead>
<tr>
<th>Event</th>
<th>Location</th>
<th>Dates</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS vLive - SEC503: Intrusion Detection In-Depth</td>
<td>SEC503 - 202008,</td>
<td>Aug 10, 2020 - Sep 16, 2020</td>
<td>vLive</td>
</tr>
<tr>
<td>SANS Essentials Live Online 2020</td>
<td></td>
<td>Aug 17, 2020 - Aug 22, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Baltimore Fall 2020</td>
<td>Baltimore, MD</td>
<td>Sep 08, 2020 - Sep 13, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Baltimore Fall 2020 - Live Online</td>
<td>Baltimore, MD</td>
<td>Sep 08, 2020 - Sep 13, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Network Security 2020 - Live Online</td>
<td>Las Vegas, NV</td>
<td>Sep 20, 2020 - Sep 25, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS London October 2020 - Live Online</td>
<td>London, United Kingdom</td>
<td>Oct 12, 2020 - Oct 17, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS October Singapore 2020 - Live Online</td>
<td>Singapore, Singapore</td>
<td>Oct 12, 2020 - Oct 24, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS Dallas Fall 2020</td>
<td>Dallas, TX</td>
<td>Oct 19, 2020 - Oct 24, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Dallas Fall 2020 - Live Online</td>
<td>Dallas, TX</td>
<td>Oct 19, 2020 - Oct 24, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS London November 2020</td>
<td>London, United Kingdom</td>
<td>Nov 02, 2020 - Nov 07, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS London November 2020 - Live Online</td>
<td>London, United Kingdom</td>
<td>Nov 02, 2020 - Nov 07, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS San Diego Fall 2020 - Live Online</td>
<td>San Diego, CA</td>
<td>Nov 16, 2020 - Nov 21, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS San Diego Fall 2020</td>
<td>San Diego, CA</td>
<td>Nov 16, 2020 - Nov 21, 2020</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Cyber Defense Initiative 2020 - Live Online</td>
<td>Washington, DC</td>
<td>Dec 14, 2020 - Dec 19, 2020</td>
<td>CyberCon</td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>Online</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s Only</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>