Which YARA Rules Rule: Basic or Advanced?

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 ...

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, 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 }
```

```
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):

```
/*
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:

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

```
 condition:
    $a1
}
```

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] where 0 <= X <= Y
```

For example (Alvarez, 2018):

```
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:

```
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:

```
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:

```
rule ModifierTextExample
{
  strings:
    $wide_and_nocase_string = "Texas" wide nocase
  condition:
    $wide_and_nocase_string
}
```

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)
}
```

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 = "6d484daba3927fc0744b1bbd7981a56ebef95790"
    hash1 = "d4071272cc1bf944e3867db299b3f5dce126f82b"
    hash2 = "6c7c8b804cc76e2c208c6e3b6453cb134d01fa41"
```

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

```
Usage: yara [OPTION]... [NAMESPACE:]RULES_FILE... FILE | DIR | PID

Mandatory arguments to long options are mandatory for short options too.

  -t,  --tag=TAG                      print only rules tagged as TAG
  -i,  --identifier=IDENTIFIER        print only rules named IDENTIFIER
  -c,  --count                        print only number of matches
  -n,  --negate                       print only not satisfied rules (negate)
  -D,  --print-module-data            print module data
  -g,  --print-tags                   print tags
  -m,  --print-meta                   print metadata
  -s,  --print-strings                print matching strings
  -L,  --print-string-length          print length of matched strings
  -e,  --print-namespace              print rules' namespace
  -p,  --threads=NUMBER               use the specified NUMBER of threads to scan a directory
  -l,  --max-rules=NUMBER             abort scanning after matching a NUMBER of rules
  -d VAR=VALUE                        define external variable
  -x MODULE=FILE                      pass FILE's content as extra data to MODULE
  -a,  --timeout=SECONDS              abort scanning after the given number of SECONDS
  -k,  --stack-size=SLOTS             set maximum stack size (default=16384)
       --max-strings-per-rule=NUMBER  set maximum number of strings per rule (default=10000)
  -r,  --recursive                    recursively search directories
  -f,  --fast-scan                    fast matching mode
  -w,  --no-warnings                  disable warnings
       --fail-on-warnings             fail on warnings
  -v,  --version                      show version information
  -h,  --help                         show this help and exit
```

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):

```
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 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

parameter can be any expression returning an unsigned integer, including the return value of one the *uintXX* functions itself (Alvarez, 2018).

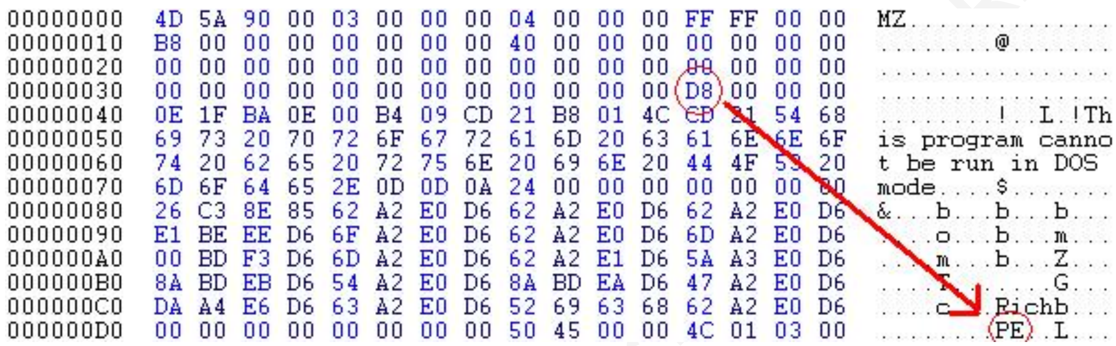
As some analysts may have no problem understanding how to use this feature, many may not. To that end, the following use case is provided to show how this powerful function may be effectively utilized.

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):

```
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.

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):

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

```
        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 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:

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

```
hash2 =
"58e78c653b2a92469963759fc88029c4badc7138e7654005dd1c5904fae163d5"
hash3 =
"a3b324cefbf81d3f1dbd573e64c453cb4d8a53ac54687d0c4caa0d1cbc409a51"
hash4 =
"c5642a2135fd315e754f8af20f92117bba50b17682021e7448019e043aa1edc9"
hash5 =
"fecfe25aaeec3911fee183ff0988ea9045a30d6c1620ed57b1ad134d86dc2ee3"
hash6 =
"ec2a717739947d3512513889bbecd9a0dac3fb65f8e171f8a0835abe8c1537e3"
strings:
  $s1 = "lsasrv32.dll and lsass.exe" fullword wide
  $s2 = "lsasrv32.dll" fullword ascii
  $s3 = "Failed to get Windows version" fullword ascii
  $s4 = "\\\\%s\\mailslot\\%s" fullword ascii
  $s5 = "%d-%d-%d %d:%d:%d Z" fullword ascii
  $s6 = ":#:/:E:J:\\:f:" fullword ascii
  $s7 = " !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!" fullword ascii
condition:
  uint16(0) == 0x5a4d and filesize < 400KB and pe.imphash() ==
  "ee845c2ebf05004bb904724010b3d898" and all of them
}
```

When yarGen was next run against the 261 FannyWorm samples, 37 individual rules were generated along with one super rule. Of the individual rules, 22 were unique, one had five matches, three had two matches, and one had four matches. The super rule, which follows, was edited in the same manner as the previous rule:

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:
```

```
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

d3b1ea2ef9bf92af1c15f8a0426a73fbec43cef2f35695d316d41991e5116c3d

81d3f13409fb76f973fdb090b945eca7b2cdea16e5ee0d7bae70acb6bc90e5c1

3ee093ba4872dc47d28b2437cc5fa404f69209339cc75e0d172b7fd38d324410

e6a54eedfdfdd2edd9c86ae211a37f7b7742bb573b4ecb523e56006291aa2b50

e9e130eec84985f18e6f5c69a222e575acd7976f804fb224a622e34aa93bd495

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 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}
                        Plugins
    shell               Launch ipython 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
csculling@ubuntu:~$ pe check /home/csculling/Desktop/EquationLaser/EquationLaser_8E2C06B52F530C9F9B5C2C743A5BB28A
Running checks on /home/csculling/Desktop/EquationLaser/EquationLaser_8E2C06B52F530C9F9B5C2C743A5BB28A:
[+] Abnormal section names: Shared
[+] Suspicious entropy in the following sections:
        - .data - 7.610638
        - Shared - 0.000000
[+] 4 extra bytes in the file
[+] PeID packer: Armadillov1xxv2xx
csculling@ubuntu:~$ pe search Shared /home/csculling/Desktop/EquationLaser/EquationLaser_8E2C06B52F530C9F9B5C2C743A5BB28A
Position in the file : 0x270
csculling@ubuntu:~$ pe checksize /home/csculling/Desktop/EquationLaser/EquationLaser_8E2C06B52F530C9F9B5C2C743A5BB28A
Name       VirtSize  VirtAddr  RawSize   RawAddr   Entropy  md5
.text      0x1a8b0   0x1000    0x400     0x1aa00   6.5792   542606a0ac9bc0c21fb965438921fceb
.rdata     0x185f    0x1c000   0x1ae00   0x1a00    5.3319   c595146db5a7811cd0ba2d4dcc2264e2
.data      0x4bfb8   0x1e000   0x1c800   0x1800    7.6106   4356caba9211586296347c65c32cc533
Shared     0x118     0x6a000   0x1e000   0x200     0.0000   bf619eac0cdf3f68d496ea9344137e8b
.rsrc      0x418     0x6b000   0x1e200   0x600     2.5508   f0ddc37fff16a7b42285f672ee799e87
.reloc     0x1c56    0x6c000   0x1e800   0x1e00    5.7657   6abbbb5d83042ccdddc319c9771351d7

4 bytes of extra data (132612 while it should be 132608)
csculling@ubuntu:~$ pe info /home/csculling/Desktop/EquationLaser/EquationLaser_8E2C06B52F530C9F9B5C2C743A5BB28A
Metadata
================================================================================
MD5:           8e2c06b52f530c9f9b5c2c743a5bb28a
SHA1:          8edeeb4cccf4bb7f7243565fd3ac829bae890ae8
SHA256:        a3b324cefbf81d3f1dbd573e64c453cb4d8a53ac54687d0c4caa0d1cbc409a51
Imphash:       ee845c2ebf05004bb904724010b3d898
Size:          132612 bytes
Type:          PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
DLL File!
Compile Time:  2004-10-18 12:24:05 (UTC - 0x4173B5E5)
Entry point:   0x1001b801 (section .text)

Sections
================================================================================
Name       VirtSize  VirtAddr  RawSize   RawAddr   Entropy  md5
.text      0x1a8b0   0x1000    0x400     0x1aa00   6.5792   542606a0ac9bc0c21fb965438921fceb
.rdata     0x185f    0x1c000   0x1ae00   0x1a00    5.3319   c595146db5a7811cd0ba2d4dcc2264e2
.data      0x4bfb8   0x1e000   0x1c800   0x1800    7.6106   4356caba9211586296347c65c32cc533
Shared     0x118     0x6a000   0x1e000   0x200     0.0000   bf619eac0cdf3f68d496ea9344137e8b
.rsrc      0x418     0x6b000   0x1e200   0x600     2.5508   f0ddc37fff16a7b42285f672ee799e87
.reloc     0x1c56    0x6c000   0x1e800   0x1e00    5.7657   6abbbb5d83042ccdddc319c9771351d7
```

Christopher S. Culling, csculling1@gmail.com

```
Imports
================================================================================
WS2_32.dll
        0x1001c334 WSACleanup
        0x1001c338 gethostname
        0x1001c33c gethostbyname
        0x1001c340 closesocket
        0x1001c344 sendto
        0x1001c348 recv
        0x1001c34c recvfrom
        0x1001c350 WSAStartup
        0x1001c354 ioctlsocket
        0x1001c358 setsockopt
        0x1001c35c select
        0x1001c360 __WSAFDIsSet
        0x1001c364 getsockopt
        0x1001c368 WSAGetLastError
        0x1001c36c socket
        0x1001c370 bind
        0x1001c374 getsockname
KERNEL32.dll
        0x1001c074 SetThreadPriority
        0x1001c078 GetCurrentThread
        0x1001c07c CloseHandle
        0x1001c080 DeviceIoControl
        0x1001c084 SleepEx
        0x1001c088 ResumeThread
        0x1001c08c TerminateThread
        0x1001c090 WaitForMultipleObjects
        0x1001c094 GetVersion
        0x1001c098 ReleaseSemaphore
        0x1001c09c InterlockedDecrement
        0x1001c0a0 InterlockedIncrement
        0x1001c0a4 CreateFileA
```

```
0x10003154 ?a73957838_2@@YAXXZ 1
0x10003154 ?a84884@@YAXXZ 2
0x10003154 ?b823838_9839@@YAXXZ 3
0x10003154 ?e747383_94@@YAXXZ 4
0x10003154 ?e83834@@YAXXZ 5
0x10003154 ?e929348_827@@YAXXZ 6

Resources:
================================================================================
Id          Name    Size     Lang          Sublang            Type         MD5
16-1-1033   None    960 B    LANG_ENGLISH  SUBLANG_ENGLISH_US data         8481356adacdd6195bcd089232212efc
```

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.

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.

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.

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, csculling1@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).

Christopher S. Culling, csculling1@gmail.com

References

AlienVault Labs. (2017, January 21). *rules/malware/APT_APT1.yar*. Retrieved from https://github.com/Yara-Rules/rules/blob/master/malware/APT_APT1.yar

Alvarez, V. (2018, June 19). *yara Documentation, Release 3.7.0.* Retrieved from https://media.readthedocs.org/pdf/yara/latest/yara.pdf

Dias, R. (2016a, February 10). *Unleashing YARA - Part 1*. Retrieved from https://countuponsecurity.com/2016/02/10/unleashing-yara-part-1/

Dias, R. (2016b, February 18). *Unleashing YARA - Part 2*. Retrieved from https://countuponsecurity.com/2016/02/18/unleashing-yara-part-2/

Dias, R. (2016c, March 9). *Unleashing YARA - Part 3*. Retrieved from https://countuponsecurity.com/tag/malware-analysis/

GReAT. (2015, February 16). *Equation: The Death Star of Malware Galaxy.* Retrieved from https://securelist.com/equation-the-death-star-of-malware-galaxy/68750/

InQuest. (2018, June 13). *A curated list of awesome YARA rules, tools, and people.* Retrieved from https://github.com/InQuest/awesome-yara#rules

Joe Security. (2018, n.d.) *Joe Sandbox Cloud.* Retrieved from https://www.joesecurity.org/joe-sandbox-cloud

Kessler, G. (2018, February 23) *File Signatures Table.* Retrieved from https://www.garykessler.net/library/file_sigs.html?utm_source=tool.lu

Khasaia, L. (2018, April 1). *SSMA - Simple Static Malware Analyzer*. Retrieved from https://github.com/secrary/SSMA

Mandiant. (2014, January 23). *Tracking Malware with Import Hashing.* Retrieved from https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html

Revers3r. (2018). *Malware Researcher's Handbook (Demystifying PE File).* Retrieved from https://resources.infosecinstitute.com/2-malware-researchers-handbook-demystifying-pe-file/

Roth, F. (2015a, February 16*). How to Write Simple but Sound Yara Rules*. Retrieved from https://www.bsk-consulting.de/2015/02/16/write-simple-sound-yara-rules/

Christopher S. Culling, csculling1@gmail.com

Roth, F. (2015b, October 17). *How to Write Simple but Sound Yara Rules - Part 2*. Retrieved from https://www.bsk-consulting.de/2015/10/17/how-to-write-simple-but-sound-yara-rules-part-2/

Roth, F. (2016a, April 15). *How to Write Simple but Sound Yara Rules - Part 3.* Retrieved from https://www.bsk-consulting.de/2016/04/15/how-to-write-simple-but-sound-yara-rules-part-3/

Roth, F. (2016b, February). *YARA Performance Guidelines*. Retrieved from https://gist.github.com/Neo23x0/e3d4e316d7441d9143c7

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

SANS. (2018, n.d.) *SIFT Workstation*. Retrieved from https://digital-forensics.sans.org/community/downloads

Shalev, S. (2017, March 6). *theZoo/malwares/Binaries/EquationGroup/EquationGroup.zip.* Retrieved from https://github.com/ytisf/theZoo/blob/master/malwares/Binaries/EquationGroup/EquationGroup.zip

Te-k. (2018, May 28). *CLI tool to analyze PE files*. Retrieved from https://github.com/Te-k/pe

VirusTotal. (n.d.). *Welcome to YARA's documentation!* Retrieved from https://yara.readthedocs.io/en/v3.7.1/

Wikibooks. (2018, June 25). *X86 Disassembly/Windows Executable Files.* Retrieved from https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files#/media/File:RevEngPeSig.JPG

Zeltser, L. (n.d.) *REMnux: A Linux Toolkit for Reverse-Engineering and Analyzing Malware.* Retrieved from https://remnux.org

Zetter, K. (2015, February 16). *Suite of Sophisticated Nation-State Attack Tools Found with Connection to Stuxnet.* Retrieved from https://www.wired.com/2015/02/kapersky-discovers-equation-group/

Christopher S. Culling, csculling1@gmail.com

Appendix A

Simple Static Malware Analyzer Results - Equation Laser

```
csculling@ubuntu:~/SSMA$ python3 ssma.py /home/csculling/Desktop/EquationLaser/EquationLaser_DE356F2A55B25E047424
23B5EC56DE93

SSMA  Simple
      Static
      Malware
      Analyzer

File Details:
        File: /home/csculling/Desktop/EquationLaser/EquationLaser_DE356F2A55B25E04742423B5EC56DE93
        Size: 132612 bytes
        Type: application/x-dosexec
        MD5:  de356f2a55b25e04742423b5ec56de93
        SHA1: 446323d945ce713859ad3451cb4a5db0541020d6
        Date: Mon Oct 18 08:24:05 2004
        PE file entropy: 6.68497087497457

================================================================================
Number of Sections: 6

Section VirtualAddress VirtualSize SizeofRawData Sections_MD5_Hash Section_Entropy
.text   0x1000              108720        109056 542606a0ac9bc0c21fb965438921fceb 6.579159815610209
.rdata  0x1c000               6239          6656 c595146db5a7811cd0ba2d4dcc2264e2 5.3319319633434175
.data   0x1e000             311224          6144 4356caba9211586296347c65c32cc533 7.610637884216575
Shared  0x6a000                280           512 bf619eac0cdf3f68d496ea9344137e8b 0.0
.rsrc   0x6b000               1048          1536 f0ddc37fff16a7b42285f672ee799e87 2.5508121523378002
.reloc  0x6c000               7254          7680 6abbbb5d83042ccdddc319c9771351d7 5.765710311641036

SUSPICIOUS size of the section ".data" when stored in memory - 311224

Very high or very low entropy means that file/section is compressed or encrypted since truly random data is not c
ommon.

SUSPICIOUS section names: Shared
================================================================================
Overlay Data is present which is often associated with malware
 Start offset: 0x00020600
 Size:         0x00000004 4 bytes 0.00%
 MD5:          e704cdc9befef9192cf88e9d889382a4
 SHA-256:      7122bdbeb9d04c1ab4924e8d941c84c5043e8f6f8831cd07f83d4cbd3259b805
 MAGIC:        b'c2aa573b66' ªW;f
 PE file without overlay:
  MD5:          752af597e6d9fd70396accc0b9013dbe
  SHA-256:      9412a66bc81f51a1fa916ac47c77e02ac1a7c9dff543233ed70aa265ef6a1e76
================================================================================
This file contains a list of Windows functions commonly used by malware.
For more information use the Microsoft documentation.

        gethostname - Retrieves the hostname of the computer. Backdoors sometimes use gethostname as part of a su
rvey of the victim machine.
        gethostbyname - Used to perform a DNS lookup on a particular hostname prior to making an IP connection to
 a remote host. Hostnames that serve as command- and-control servers often make good network-based signatures.
        sendto - Sends data to a remote machine. Malware often uses this function to send data to a remote comman
d-and-control server.
        recv - Receives data from a remote machine. Malware often uses this function to receive data from a remot
e command-and-control server.
        recvfrom - Receives data from a remote machine. Malware often uses this function to receive data from a r
emote command-and-control server.
        WSAStartup - Used to initialize low-level network functionality. Finding calls to WSAStartup can often be
 an easy way to locate the start of network-related functionality.
        DeviceIoControl - Sends a control message from user space to a device driver. DeviceIoControl is popular
with kernel malware because it is an easy, flexible way to pass information between user space and kernel space.
        ResumeThread - Resumes a previously suspended thread. ResumeThread is used as part of several injection t
echniques.
        CreateFileA - Creates a new file or opens an existing file.
        GetVersionExA - Returns information about which version of Windows is currently running. This can be used
```

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.
        GetProcAddress - Retrieves the address of a function in a DLL loaded into memory. Used to import function
s from other DLLs in addition to the functions imported in the PE file header.
        LoadLibraryA - Loads a DLL into a process that may not have been loaded when the program started. Importe
d 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, wh
ich 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 addresse
s. 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 access
ible 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 somet
imes 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 fun
ction to conceal malicious activity.
        GetModuleHandleA - Used to obtain a handle to an already loaded module. Malware may use GetModuleHandle t
o 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.
        LoadLibraryW - Loads a DLL into a process that may not have been loaded when the program started. Importe
d by nearly every Win32 program.
        OpenProcess - Opens a handle to another process running on the system. This handle can be used to read an
d 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 r
eads or writes any files in the temporary file path.
        CreateFileW - 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 a
nalyze the new process as well.
        CreateProcessW - Creates and launches a new process. If malware creates a new process, you will need to a
nalyze the new process as well.
        GetStartupInfoA - Retrieves a structure containing details about how the current process was configured t
o run, such as where the standard handles are directed.
        GetStartupInfoW - Retrieves a structure containing details about how the current process was configured t
o run, such as where the standard handles are directed.
        GetTickCount - Retrieves the number of milliseconds since bootup. This function is sometimes used to gath
er 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 w
ith keyloggers and spyware, this function also provides an easy way to load a DLL into all GUI processes on the s
ystem. This function is sometimes added by the compiler.
        CallNextHookEx - 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.
        RegOpenKeyExW - Opens a handle to a registry key for reading and editing. Registry keys are sometimes wri
tten 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.
        OpenProcessToken - Opens a handle to another process running on the system. This handle can be used to re
ad 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 wri
tten 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.
        RegOpenKeyA - Opens a handle to a registry key for reading and editing. Registry keys are sometimes writt
en as a way for software to achieve persistence on a host. The registry also contains a whole host of operating s
ystem and application setting information.
        OpenSCManagerA - Opens a handle to the service control manager. Any program that installs, modifies, or c
ontrols 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 specialised on the identification of well-known malware.
Result:
         Str_Win32_Winsock2_Library - Match Winsock 2 API library declaration
```

Christopher S. Culling, csculling1@gmail.com

```
================================================================================
These Yara Rules aimed to detect well-known sofware packers, that can be used by malware to hide itself.
Result:
        Armadillo_v1xx_v2xx_additional
        Microsoft_Visual_Cpp_60_DLL_additional
        Microsoft_Visual_Cpp_v70_DLL
        Microsoft_Visual_Cpp_v50v60_MFC
        Microsoft_Visual_Cpp_60_DLL_Debug
        Armadillo_v1xx_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 algoritms.
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:
        win_mutex - Create or check mutex
        win_registry - 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 :)
```

Christopher S. Culling, csculling1@gmail.com

Appendix B

Simple Static Malware Analyzer Results - FannyWorm

```
csculling@ubuntu:~/SSMA$ python3 ssma.py /home/csculling/Desktop/FannyWorm/FannyWorm_2C029BE8E3B0C9448ED5
E88B52852ADE

SSMA   Simple
       Static
       Malware
       Analyzer

File Details:
        File: /home/csculling/Desktop/FannyWorm/FannyWorm_2C029BE8E3B0C9448ED5E88B52852ADE
        Size: 184320 bytes
        Type: application/x-dosexec
        MD5:  2c029be8e3b0c9448ed5e88b52852ade
        SHA1: cd6fddaa492a38629fd80c46bb2c2d37ec80444c
        Date: Mon Jul 28 01:11:35 2008
        PE file entropy: 6.372905768797145

================================================================================
Number of Sections: 5

Section VirtualAddress VirtualSize SizeofRawData Sections_MD5_Hash Section_Entropy
.text   0x1000               50613         53248 6d75617229827b2cfd28f8c7d4fae49c 6.204662699717051
.rdata  0xe000                7090          8192 611bf51687e24fbdbb68797ac29a46e6 5.113443284517251
.data   0x10000               2156          4096 2cef86abb489ad5788edf99844ed4ab8 2.9325260938167017
.rsrc   0x11000             108344        110592 b3e9018bec3f078148601de6a52ce323 6.540798356733707
.reloc  0x2c000               3520          4096 46431dede6ae9f44f5bdd5c03595bc5c 4.831299025167482

================================================================================
 No overlay Data Present
================================================================================
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 fixed 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 th
at only a single instance of malware is running on a system at any given time. Malware often uses fixed n
ames for mutexes, which can be good host-based indicators.
        GetTempPathA - Returns the temporary file path. If you see malware call this function, check whet
her it reads or writes any files in the temporary file path.
        OpenProcess - 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.
        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 th
at have changed between different versions of Windows.
        CreateMutexW - 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.
        OpenMutexW - Opens a handle to a mutual exclusion object that can be used by malware to ensure th
at only a single instance of malware is running on a system at any given time. Malware often uses fixed n
ames for mutexes, which can be good host-based indicators.
        GetModuleHandleA - Used to obtain a handle to an already loaded module. Malware may use GetModule
Handle to locate and modify code in a loaded module or to search for a good location to inject code.
        LoadLibraryExA - Loads a DLL into a process that may not have been loaded when the program starte
d. Imported by nearly every Win32 program.
        CreateFileW - Creates a new file or opens an existing file.
        LoadLibraryW - Loads a DLL into a process that may not have been loaded when the program started.
 Imported by nearly every Win32 program.
        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 MapV
iewOfFile , 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 i
t accessible via memory addresses. Launchers, loaders, and injectors use this function to read and modify
```

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 n
eed to analyze the new process as well.
        FindResourceA - Used to find a resource in an executable or loaded DLL. Malware some- times 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 s
tore strings, configuration information, or other malicious files
        OpenProcessToken - Opens a handle to another process running on the system. This handle can be us
ed 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 somet
imes written as a way for software to achieve persistence on a host. The registry also contains a whole h
ost 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_Win32_Winsock2_Library - Match Winsock 2 API library declaration

================================================================================
These Yara Rules aimed to detect well-known sofware packers, that can be used by malware to hide itself.
Result:
         Armadillo_v1xx_v2xx_additional
         Microsoft_Visual_Cpp_60_DLL_additional
         Microsoft_Visual_Cpp_v70_DLL
         Microsoft_Visual_Cpp_v50v60_MFC
         Microsoft_Visual_Cpp_60_DLL_Debug
         Armadillo_v1xx_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 algoritms.
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 a
utomated analysis.
Result:
         antisb_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

Appendix C

Joe Sandbox Cloud - FannyWorm

ID: 596380
Sample Name: ZuDBYiOvt4

Christopher S. Culling, csculling1@gmail.com

Table of Contents

Christopher S. Culling, csculling1@gmail.com

Christopher S. Culling, csculling1@gmail.com

Analysis Report

Overview

General Information

Joe Sandbox Version:	23.0.0
Analysis ID:	596380
Start time:	01:52:29
Joe Sandbox Product:	Cloud
Start date:	01.07.2018
Overall analysis duration:	0h 3m 46s
Hypervisor based Inspection enabled:	false
Report type:	light
Sample file name:	ZuDBYiOvt4 (renamed file extension from none to dll)
Cookbook file name:	default.jbs
Analysis system description:	Windows 7 (Office 2010 SP2, Java 1.8.0_40, Flash 16.0.0.305, Acrobat Reader 11.0.08, Internet Explorer 11, Chrome 55, Firefox 43)
Number of analysed new started processes analysed:	7
Number of new started drivers analysed:	0
Number of existing processes analysed:	0
Number of existing drivers analysed:	0
Number of injected processes analysed:	0
Technologies	• HCA enabled • EGA enabled • HDC enabled
Analysis stop reason:	Timeout
Detection:	MAL
Classification:	mal76.evad.expl.winDLL@9/7@0/0
HCA Information:	• Successful, ratio: 94% • Number of executed functions: 0 • Number of non-executed functions: 0
EGA Information:	• Successful, ratio: 100%
HDC Information:	• Successful, ratio: 100% (good quality ratio 94.7%) • Quality average: 84.9% • Quality standard deviation: 26.1%
Cookbook Comments:	• Adjust boot time • Correcting counters for adjusted boot time • Start process as user (medium integrity level) • Adjusted system time to: 29/7/2008 • Stop behavior analysis, all processes terminated
Warnings:	Show All • Exclude process from analysis (whitelisted): conhost.exe, dllhost.exe

Detection

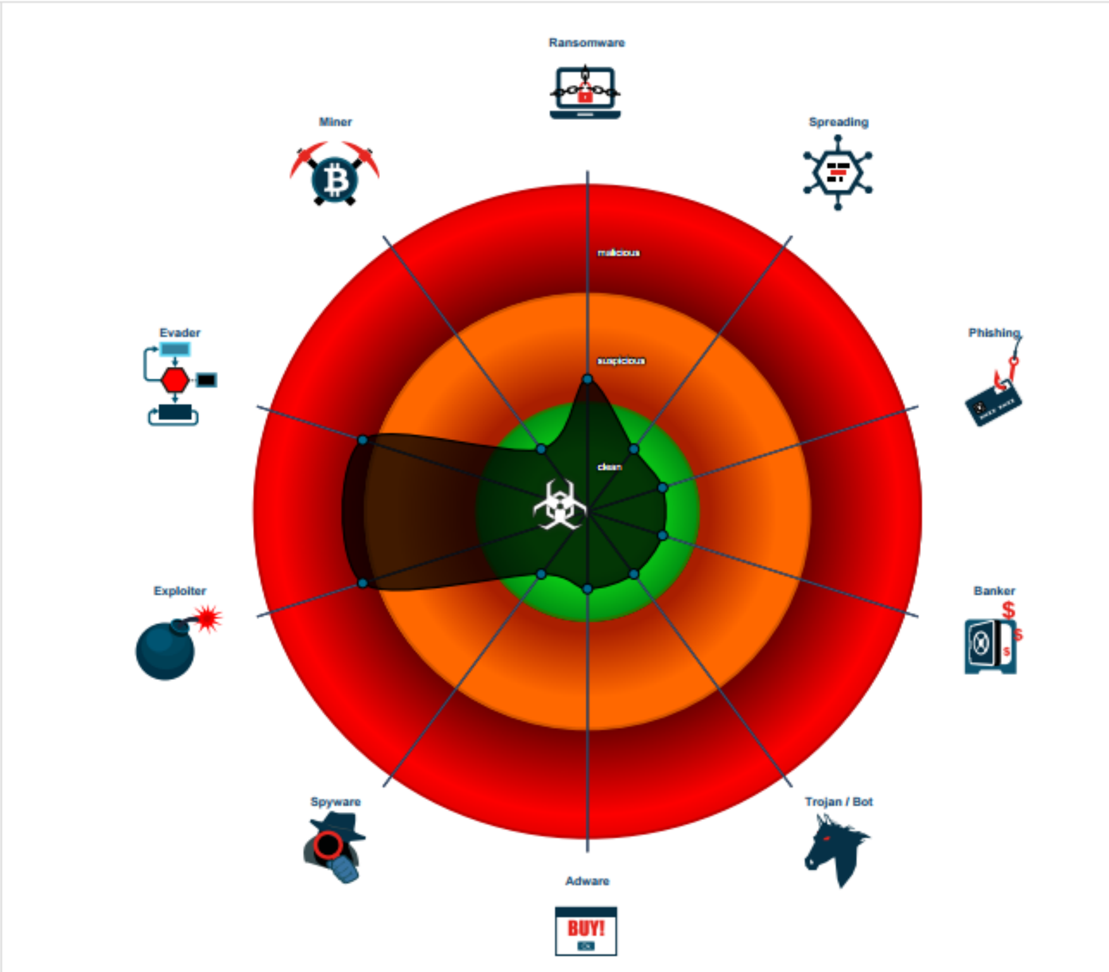
Strategy	Score	Range	Reporting	Detection
Threshold	76	0 - 100	Report FP / FN	MALICIOUS SUSPICIOUS CLEAN UNKNOWN

Confidence

Christopher S. Culling, csculling1@gmail.com

Strategy	Score	Range	Further Analysis Required?	Confidence
Threshold	5	0 - 5	false	99% 80% 60% 40% 20% 5%

Classification



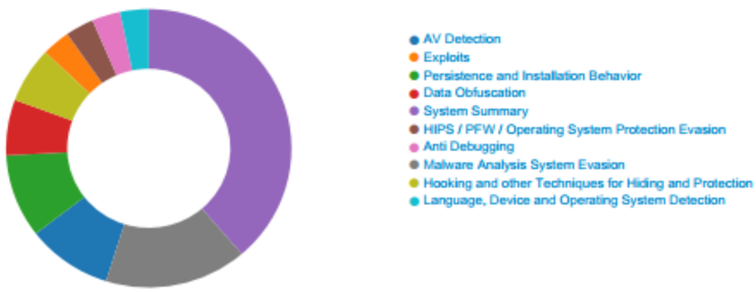
Analysis Advice

Contains functionality to modify the execution of threads in other processes

Christopher S. Culling, csculling1@gmail.com

Sample drops PE files which have not been started, submit dropped PE samples for a secondary analysis to Joe Sandbox

Signature Overview



Click to jump to signature section

AV Detection:

- Antivirus detection for dropped file
- Antivirus detection for submitted file
- Antivirus detection for unpacked file

Exploits:

- Accesses ntoskrnl, likely to find offsets for exploits

Persistence and Installation Behavior:

- Drops files with a non-matching file extension (content does not match file extension)
- Drops PE files
- Drops PE files to the windows directory (C:\Windows)

Data Obfuscation:

- Contains functionality to dynamically determine API calls
- Uses code obfuscation techniques (call, push, ret)

System Summary:

- Creates files inside the system directory
- Detected potential crypto function
- PE file contains executable resources (Code or Archives)
- Classification label
- Contains functionality to load and extract PE file embedded resources
- Creates temporary files
- PE file has an executable .text section and no other executable section
- Reads software policies
- Reads the Windows registered organization settings
- Runs a DLL by calling functions
- Spawns processes

Christopher S. Culling, csculling1@gmail.com

Reads the Windows registered owner settings

HIPS / PFW / Operating System Protection Evasion:

Contains functionality to inject threads in other processes

Anti Debugging:

Contains functionality to dynamically determine API calls

Malware Analysis System Evasion:

Found dropped PE file which has not been started or loaded

Found large amount of non-executed APIs

Program exit points

Queries a list of all running drivers

Queries a list of all running processes

Hooking and other Techniques for Hiding and Protection:

May modify the system service descriptor table (often done to hook functions)

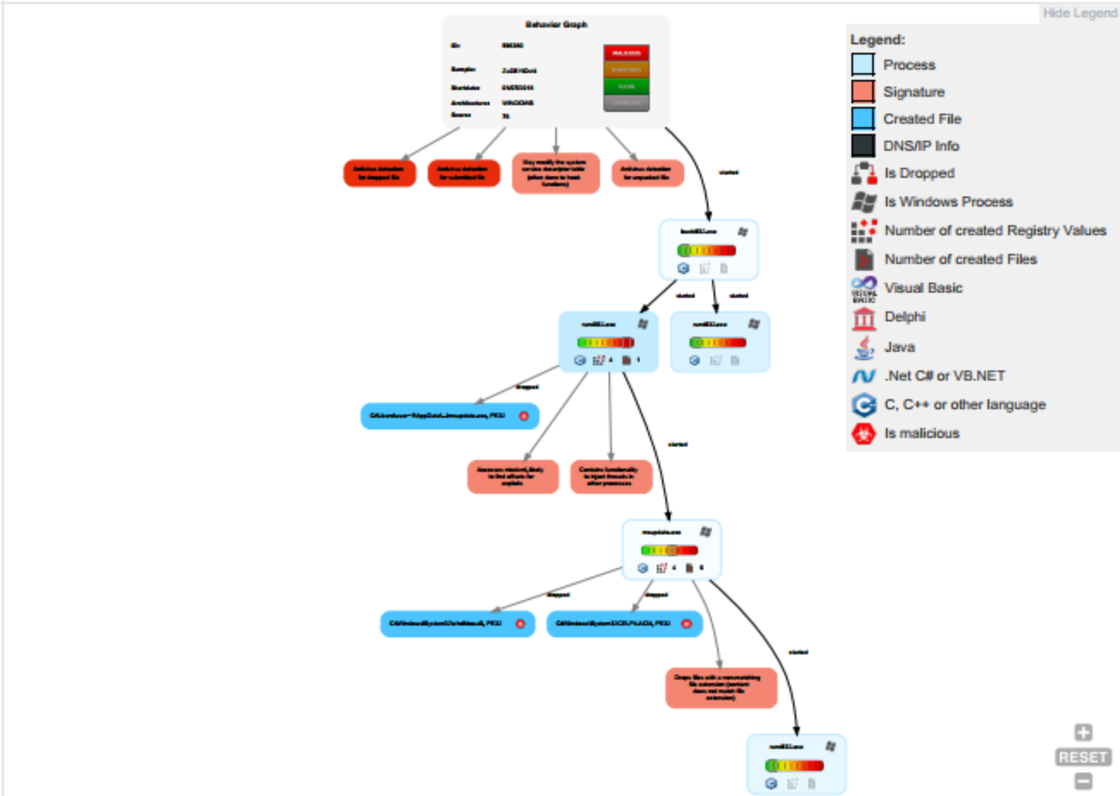
Disables application error messsages (SetErrorMode)

Language, Device and Operating System Detection:

Contains functionality to query windows version

Behavior Graph

Christopher S. Culling, csculling1@gmail.com



Simulations

Behavior and APIs

Time	Type	Description
01:53:21	API Interceptor	1x Sleep call for process: rundll32.exe modified
01:53:22	API Interceptor	1x Sleep call for process: msupdate.exe modified

Antivirus Detection

Initial Sample

Source	Detection	Scanner	Label	Link
ZuDBYiOvt4.dll	100%	Avira	TR/Drop.Agent.aeb	

Dropped Files

Source	Detection	Scanner	Label	Link
C:\Windows\System32\shelldoc.dll	100%	Avira	TR/PWS.Wsgame.9769.A	
C:\Users\user~1\AppData\Local\Temp\msupdate.exe	100%	Avira	TR/Drop.Agent.aeb	
C:\Windows\System32\ECELP4.ACM	100%	Avira	TR/Horse.BQE	

Unpacked PE Files

Source	Detection	Scanner	Label	Link
3.0.msupdate.exe.400000.3.unpack	100%	Avira	TR/Drop.Agent.aeb	
2.1.rundll32.exe.10000000.1.unpack	100%	Avira	TR/Drop.Agent.aeb	

Copyright Joe Security LLC 2018 Page 8 of 19

Christopher S. Culling, csculling1@gmail.com

Source	Detection	Scanner	Label	Link
3.0.msupdate.exe.400000.0.unpack	100%	Avira	TR/Drop.Agent.aeb	
2.1.rundll32.exe.10000000.3.unpack	100%	Avira	TR/Drop.Agent.aeb	
3.0.msupdate.exe.400000.2.unpack	100%	Avira	TR/Drop.Agent.aeb	
3.1.msupdate.exe.400000.0.unpack	100%	Avira	TR/Drop.Agent.aeb	
2.1.rundll32.exe.10000000.2.unpack	100%	Avira	TR/Drop.Agent.aeb	
3.0.msupdate.exe.400000.5.unpack	100%	Avira	TR/Drop.Agent.aeb	
3.0.msupdate.exe.400000.4.unpack	100%	Avira	TR/Drop.Agent.aeb	
3.2.msupdate.exe.400000.1.unpack	100%	Avira	TR/Drop.Agent.aeb	
2.1.rundll32.exe.10000000.0.unpack	100%	Avira	TR/Drop.Agent.aeb	
3.0.msupdate.exe.400000.1.unpack	100%	Avira	TR/Drop.Agent.aeb	

Domains

No Antivirus matches

URLs

No Antivirus matches

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

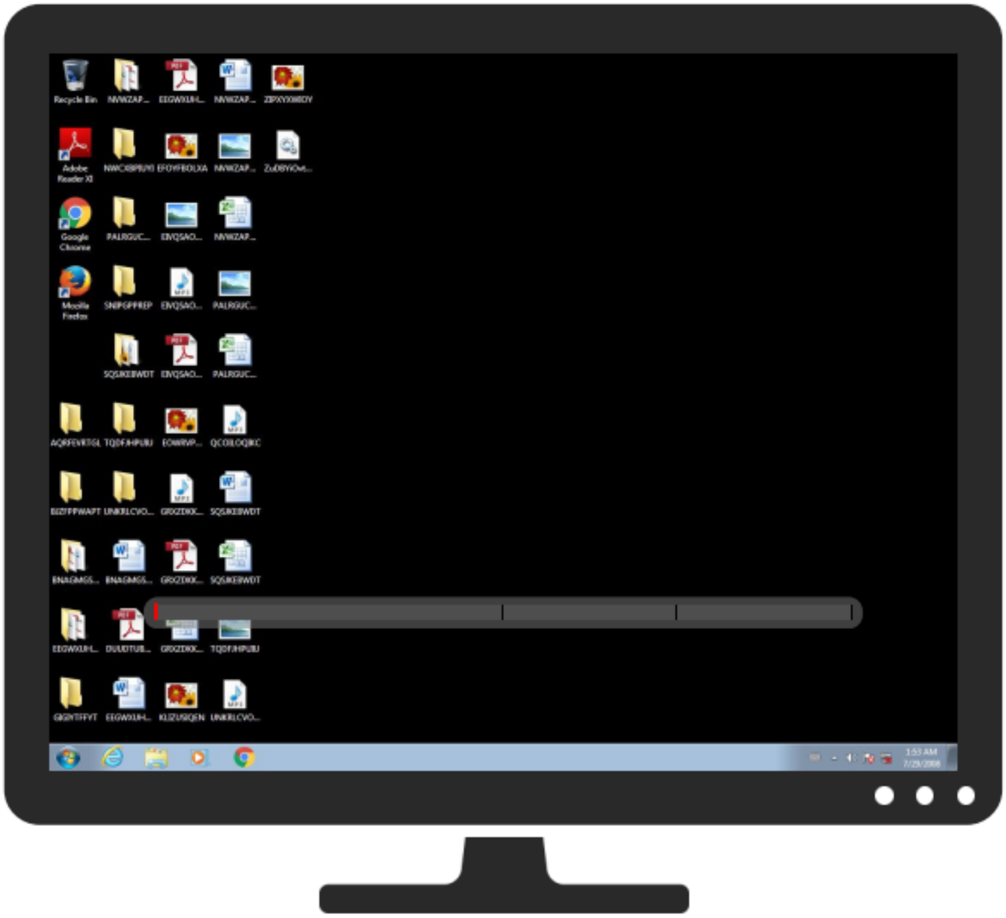
No context

Dropped Files

Christopher S. Culling, csculling1@gmail.com

No context

Screenshots



Startup

- System is w7_1
- loaddll32.exe (PID: 3548 cmdline: loaddll32.exe 'C:\Users\user\Desktop\ZuDBYiOvt4.dll' MD5: D2792A55032CFE825F07DCD4BEC5F40F)
  - rundll32.exe (PID: 3556 cmdline: rundll32.exe C:\Users\user\Desktop\ZuDBYiOvt4.dll,#1 MD5: 51138BEEA3E2C21EC44D0932C71762A8)
    - msupdate.exe (PID: 3564 cmdline: C:\Users\user~1\AppData\Local\Temp\msupdate.exe MD5: DDC9915A5158A9560AD1EF2F16CCE9A6)
      - rundll32.exe (PID: 3588 cmdline: rundll32 C:\Windows\MSAgent\AGENTCPD.DLL _start@16 0 MD5: 51138BEEA3E2C21EC44D0932C71762A8)
  - rundll32.exe (PID: 3692 cmdline: rundll32.exe C:\Users\user\Desktop\ZuDBYiOvt4.dll,#2 MD5: 51138BEEA3E2C21EC44D0932C71762A8)
- cleanup

Created / dropped Files

C:\Users\user~1\AppData\Local\Temp\msupdate.exe	
Process:	C:\Windows\System32\rundll32.exe
File Type:	PE32 executable (console) Intel 80386, for MS Windows
Size (bytes):	106496
Entropy (8bit):	6.5229026614713135

Christopher S. Culling, csculling1@gmail.com

C:\Users\user~1\AppData\Local\Temp\msupdate.exe	
Encrypted:	false
MD5:	DDC9915A5158A9560AD1EF2F16CCE9A6
SHA1:	A870BF68AD03940FF2FC61A4BA2C5C1CC53AA252
SHA-256:	9E77FABA3137C28572CE9E7BC62897139451AEB99C68958C457214CF8B1CC723
SHA-512:	7DA029ADB7137B9B4B92E6FEFBBB114EC91A312AB87284E5CD2FF3099E9F7A4AEA478171112755592EB9F99820 A19525C42C62DA91EF0DC80D677BD14F874F49
Malicious:	true
Antivirus:	• Antivirus: Avira, Detection: 100%, Browse
Reputation:	low

C:\Windows\System32\ECELP4.ACM	
Process:	C:\Users\user\AppData\Local\Temp\msupdate.exe
File Type:	PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
Size (bytes):	32768
Entropy (8bit):	4.502829685213646
Encrypted:	false
MD5:	C02C5D4E8D06F8F8ADB3756F061D35D2
SHA1:	C308EA7A2A0F1A1B41C89911D79151FD5CD0CD3C
SHA-256:	787419D07F4BDA9BC91072BB8644ED943089AD44F4BB5920BFE452DC111C244D
SHA-512:	5367412ED994E1E20ECB3C14A86C67B007F00BD72845B84E32783842F08EA190FF3A0AD0313DD7CF69E53FB8A6 CC550ABA8D3312279A5E2418F2C2AF78606707
Malicious:	true
Antivirus:	• Antivirus: Avira, Detection: 100%, Browse
Reputation:	low

C:\Windows\System32\shelldoc.dll	
Process:	C:\Users\user\AppData\Local\Temp\msupdate.exe
File Type:	PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
Size (bytes):	53248
Entropy (8bit):	5.305756001441466
Encrypted:	false
MD5:	03F8CFDF5E6D9ECDFF1CAB3E47D39F44
SHA1:	A9FBFE65A3D44A55BFDF0BD01C6C81F436139447
SHA-256:	6EB00B34D1DAFFA49B2F4C90841705B2C994563BDE672BF35EB1C46CDB19A1ED
SHA-512:	6E7CEC13F27E602BF5105C1F34417D4CEB2CE614A9B4CBE0636A8AF68988348297ED5300ECE7D4AF802C23D476 DCB9D0DC528B0B8EFB85A64302F07B8AF0EA5C
Malicious:	true
Antivirus:	• Antivirus: Avira, Detection: 100%, Browse
Reputation:	low

C:\Windows\Temp\~DE1923.tmp	
Process:	C:\Users\user\AppData\Local\Temp\msupdate.exe
File Type:	data
Size (bytes):	8
Entropy (8bit):	0.5435644431995964
Encrypted:	false
MD5:	829FAD6054098EDC501A7CBCA1F87823
SHA1:	5482540332B93EAFFBB991FC8BCB508822909F83
SHA-256:	C69A9157638E69FB692D827383C3F27E586E0C98989CFFDF8BD4C982AD837A4C
SHA-512:	F1BB9B76F6CB8D6CD284F2385D46787B0314B307414E41485E440D9B3D5C0674F01FADAC748DE41442A0C1C2C2 191DBA319BD36C9C76E7D97BFE69B25A970D6C
Malicious:	false
Reputation:	low

C:\t3fc	
Process:	C:\Users\user\AppData\Local\Temp\msupdate.exe
File Type:	data
Size (bytes):	34540
Entropy (8bit):	7.990279028006206
Encrypted:	true
MD5:	BBA957CD854887C77675535EF816ABD8
SHA1:	161BDFE788107164323440186C0D8A559684D97F
SHA-256:	117A54214DB4E8F3DABA8082B1D99978F367DE5BA85A85E2A2FA8C68DEF17F57
SHA-512:	F6F47041F539EFFB49079CEC1A6E4284FDE11C7C78BBCA8BF1BA593E6AB7C8B3A0BD0AA4755226BF3BC124862 A736CE2589FE21967B138BDDB9DE6999B5ACF0A

Christopher S. Culling, csculling1@gmail.com

C:\t3fc	
Malicious:	false
Reputation:	low

C:\t3fc.1	
Process:	C:\Users\user\AppData\Local\Temp\msupdate.exe
File Type:	data
Size (bytes):	91627
Entropy (8bit):	4.824431138396385
Encrypted:	false
MD5:	DAC4C47F508D11006762B6AF39BBAB15
SHA1:	C44EC666E1941880D14E36FD93FF2C671722B951
SHA-256:	D95755176C2BB544B4C3B37314E1ADAA7C9B83B0BB438130AD5B178B26CC331F
SHA-512:	BC0E8B3074BC73DA3C3EFC16A197AE1A057620725C5B0DE8F525B416EB5BADEB53BFCF70E2C8DF29E1AA7F07 5EC90C7D4FF3319148E2E0E737428DDA8151A4D5
Malicious:	false
Reputation:	low

stdout	
Process:	C:\Windows\System32\loaddll32.exe
File Type:	ASCII text, with CRLF line terminators
Size (bytes):	221
Entropy (8bit):	4.882199539528061
Encrypted:	false
MD5:	C4205A10A845D8A97A87ED85D960986D
SHA1:	32D9CBAC2EA7C04096D455E4A051598844AC5342
SHA-256:	02E77ECE2FC695E373D9DB97182FD0AE3601B5B6947D6AF1BFA5BE07EB15EA85
SHA-512:	E3F13B16755010602077CA7DC88B1445E570340D5FFEA4C8539395B0E0944802CF085E9A8BA44FF234FB4F8C006: 48F1ADCD75BD5DF492A1B50AAA9995993A61
Malicious:	false
Reputation:	low

Contacted Domains/Contacted IPs

Contacted Domains

No contacted domains info

Contacted IPs

No contacted IP infos

Static File Info

General	
File type:	PE32 executable (DLL) (GUI) Intel 80386, for MS Wi ndows
Entropy (8bit):	6.371312870404757
TrID:	• Win32 Dynamic Link Library (generic) (1002004/3) 99.60% • Generic Win/DOS Executable (2004/3) 0.20% • DOS Executable Generic (2002/1) 0.20% • Autodesk FLIC Image File (extensions: flc, fli, cel) (7/3) 0.00%
File name:	ZuDBYiOvt4.dll
File size:	184320
MD5:	1b27ac722847f5a3304e3896f0528fa4
SHA1:	c01ed51535f5c1436fe50f183b3efbb293683499
SHA256:	e84d10c3399e39858ec1d54ef1fd7c3bc9484ba692fa476 16c5df0447c19605a
SHA512:	ab960763f8ef65b87ee83a145b947d6306ce77fbae7c6c6 86a7118919481bc090801f94826e978f6e2a935b26b536t 9ec601b3b70df06c2c772df31b0dbf4a25

Copyright Joe Security LLC 2018 Page 12 of 19

Christopher S. Culling, csculling1@gmail.com

General

File Content Preview:	MZ......................@..!..L.!Th is program cannot be run in DOS mode....$...........w.Y. w.Y.w.Y.k.Y.w.YFk.Y.w.Y.T.Y.w.Y.x.Y.w.Y.w.YNw.Y.h.Y .w.Y.T.Y.w.Y.Q.Y.w.Y•h.Y.w.Y•h.Y.w.Y)q.Y.w.Y•h.Y.w.Y Rich.w.Y.......

File Icon

Static PE Info

General

Entrypoint:	0x1000d41b
Entrypoint Section:	.text
Digitally signed:	false
Imagebase:	0x10000000
Subsystem:	windows gui
Image File Characteristics:	LOCAL_SYMS_STRIPPED, 32BIT_MACHINE, EXECUTABLE_IMAGE, DLL, LINE_NUMS_STRIPPED
DLL Characteristics:	
Time Stamp:	0x488D7F37 [Mon Jul 28 08:11:35 2008 UTC]
TLS Callbacks:	
CLR (.Net) Version:	
OS Version Major:	4
OS Version Minor:	0
File Version Major:	4
File Version Minor:	0
Subsystem Version Major:	4
Subsystem Version Minor:	0
Import Hash:	1f5e76572fad36553733428ca3571f53

Entrypoint Preview

Instruction
push ebp
mov ebp, esp
push ebx
mov ebx, dword ptr [ebp+08h]
push esi
mov esi, dword ptr [ebp+0Ch]
push edi
mov edi, dword ptr [ebp+10h]
test esi, esi
jne 00007F3309310AABh
cmp dword ptr [10010854h], 00000000h
jmp 00007F3309310AC8h
cmp esi, 01h
je 00007F3309310AA7h
cmp esi, 02h
jne 00007F3309310AC4h
mov eax, dword ptr [10010868h]
test eax, eax
je 00007F3309310AABh
push edi
push esi
push ebx
call eax
test eax, eax
je 00007F3309310AAEh
push edi
push esi
push ebx
call 00007F33093109BAh

Christopher S. Culling, csculling1@gmail.com

Instruction
test eax, eax
jne 00007F3309310AA6h
xor eax, eax
jmp 00007F3309310AF0h
push edi
push esi
push ebx
call 00007F3309305743h
cmp esi, 01h
mov dword ptr [ebp+0Ch], eax
jne 00007F3309310AAEh
test eax, eax
jne 00007F3309310AD9h
push edi
push eax
push ebx
call 00007F3309310996h
test esi, esi
je 00007F3309310AA7h
cmp esi, 03h
jne 00007F3309310AC8h
push edi
push esi
push ebx
call 00007F3309310985h
test eax, eax
jne 00007F3309310AA5h
and dword ptr [ebp+0Ch], eax
cmp dword ptr [ebp+0Ch], 00000000h
je 00007F3309310AB3h
mov eax, dword ptr [10010868h]
test eax, eax
je 00007F3309310AAAh
push edi
push esi
push ebx
call eax
mov dword ptr [ebp+0Ch], eax
mov eax, dword ptr [ebp+0Ch]
pop edi
pop esi
pop ebx
pop ebp
retn 000Ch
jmp dword ptr [1000E1D0h]
jmp dword ptr [1000E1D8h]
jmp dword ptr [1000E1E0h]
jmp dword ptr [1000E1ECh]
jmp dword ptr [1000E1F0h]
jmp dword ptr [1000E180h]

Data Directories

Name	Virtual Address	Virtual Size	Is in Section
IMAGE_DIRECTORY_ENTRY_EXPORT	0xfb70	0x42	.rdata
IMAGE_DIRECTORY_ENTRY_IMPORT	0xf048	0x78	.rdata
IMAGE_DIRECTORY_ENTRY_RESOURCE	0x11000	0x1a738	.rsrc
IMAGE_DIRECTORY_ENTRY_EXCEPTION	0x0	0x0	
IMAGE_DIRECTORY_ENTRY_SECURITY	0x0	0x0	
IMAGE_DIRECTORY_ENTRY_BASERELOC	0x2c000	0x990	.reloc
IMAGE_DIRECTORY_ENTRY_DEBUG	0x0	0x0	
IMAGE_DIRECTORY_ENTRY_COPYRIGHT	0x0	0x0	
IMAGE_DIRECTORY_ENTRY_GLOBALPTR	0x0	0x0	
IMAGE_DIRECTORY_ENTRY_TLS	0x0	0x0	

Christopher S. Culling, csculling1@gmail.com

Name	Virtual Address	Virtual Size	Is in Section
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	0x0	0x0	
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	0x0	0x0	
IMAGE_DIRECTORY_ENTRY_IAT	0xe000	0x220	.rdata
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	0x0	0x0	
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	0x0	0x0	
IMAGE_DIRECTORY_ENTRY_RESERVED	0x0	0x0	

Sections

Name	Virtual Address	Virtual Size	Raw Size	Xored PE	ZLIB Complexity	File Type	Entropy	Characteristics
.text	0x1000	0xc5b5	0xd000	False	0.481370192308	data	6.20416166568	IMAGE_SCN_MEM_EXECUTE, IMAGE_SCN_CNT_CODE, IMAGE_SCN_MEM_READ
.rdata	0xe000	0x1bb2	0x2000	False	0.382202148438	data	5.11344328452	IMAGE_SCN_CNT_INITIALIZED_DATA, IMAGE_SCN_MEM_READ
.data	0x10000	0x86c	0x1000	False	0.27001953125	data	2.93252609382	IMAGE_SCN_CNT_INITIALIZED_DATA, IMAGE_SCN_MEM_WRITE, IMAGE_SCN_MEM_READ
.rsrc	0x11000	0x1a738	0x1b000	False	0.609718605324	data	6.54062041632	IMAGE_SCN_CNT_INITIALIZED_DATA, IMAGE_SCN_MEM_READ
.reloc	0x2c000	0xdc0	0x1000	False	0.56103515625	data	4.92316510574	IMAGE_SCN_CNT_INITIALIZED_DATA, IMAGE_SCN_MEM_DISCARDABLE, IMAGE_SCN_MEM_READ

Resources

Name	RVA	Size	Type	Language	Country
BINARY	0x110f8	0x62c	Non-ISO extended-ASCII text, with very long lines, with no line terminators		
BINARY	0x11724	0x1	very short file (no magic)	English	United States
BINARY	0x11728	0x1a000	PE32 executable (console) Intel 80386, for MS Windows		
BINARY	0x2b728	0x1	very short file (no magic)	English	United States
BINARY	0x2b72c	0xb	ASCII text, with no line terminators	English	United States

Imports

DLL	Import
WS2_32.dll	htonl
KERNEL32.dll	FindClose, FindFirstFileA, GetVolumeInformationA, WaitForSingleObject, CreateMutexA, OpenMutexA, ReadFile, FindNextFileA, FreeLibrary, GetTempPathA, GetSystemDirectoryA, GetSystemTime, lstrlenA, lstrcpynA, GetComputerNameA, OpenProcess, GetVersionExA, GetModuleFileNameA, LocalFree, LocalAlloc, GetCurrentProcess, GetCurrentThread, CreateMutexW, SetLastError, lstrcpyA, GetVersion, lstrcatA, VirtualFree, ReleaseMutex, VirtualAlloc, OpenMutexW, GetModuleHandleA, LoadLibraryExA, IsBadReadPtr, CreateFileW, GetEnvironmentVariableW, LoadLibraryW, MapViewOfFile, CreateFileMappingA, LoadLibraryA, SetFilePointer, GetProcAddress, CreateThread, Sleep, CreateFileA, WriteFile, CloseHandle, GetFileTime, SetFileTime, CreateProcessA, BeginUpdateResourceA, UpdateResourceA, EndUpdateResourceA, FindResourceA, LoadResource, SizeofResource, LockResource, DeleteFileA, GetLastError, CopyFileA, SetFileAttributesA, GetEnvironmentVariableA, GetCurrentProcessId, GetFileSize
USER32.dll	UnregisterClassA, SetPropA, CreateWindowExW, DestroyWindow, wsprintfA, UnregisterClassW
ADVAPI32.dll	RegDeleteValueA, RegEnumValueA, LookupAccountNameA, RegEnumKeyExA, LookupAccountSidA, IsValidSid, AccessCheck, OpenProcessToken, GetTokenInformation, GetSidSubAuthorityCount, GetSidSubAuthority, GetUserNameW, LookupAccountSidW, AllocateAndInitializeSid, FreeSid, RegOpenKeyExA, RegQueryValueExA, RegCreateKeyExA, RegCloseKey, RegSetValueExA, SetSecurityDescriptorGroup, SetSecurityDescriptorOwner, InitializeSecurityDescriptor, AddAccessAllowedAce, InitializeAcl, DuplicateTokenEx, OpenThreadToken, GetSidIdentifierAuthority, SetSecurityDescriptorDacl
MSVCRT.dll	_mbschr, _local_unwind2, strcmp, wcsncat, wcscpy, strncmp, _wcsicmp, _adjust_fdiv, _itoa, _stricmp, sscanf, strcat, memset, strstr, strncpy, memcpy, strlen, malloc, strcpy, free, strncat, memcmp, _snprintf, _ftol, realloc, _abnormal_termination, wcscmp, wcslen, ??3@YAXPAX@Z, ??2@YAPAXI@Z, _initterm

Possible Origin

Language of compilation system	Country where language is spoken	Map
English	United States	

Network Behavior

Christopher S. Culling, csculling1@gmail.com

No network behavior found

Code Manipulations

Statistics

Behavior

Click to jump to process

System Behavior

Analysis Process: loaddll32.exe PID: 3548 Parent PID: 2916

General

Start time:	01:53:21
Start date:	29/07/2008
Path:	C:\Windows\System32\loaddll32.exe
Wow64 process (32bit):	false
Commandline:	loaddll32.exe 'C:\Users\user\Desktop\ZuDBYiOvt4.dll'
Imagebase:	0x10c0000
File size:	112640 bytes
MD5 hash:	D2792A55032CFE825F07DCD4BEC5F40F
Has administrator privileges:	true
Programmed in:	C, C++ or other language
Reputation:	moderate

File Activities

File Written

File Path	Offset	Length	Value	Ascii	Completion	Count	Source Address	Symbol

Christopher S. Culling, csculling1@gmail.com

File Path	Offset	Length	Value	Ascii	Completion	Count	Source Address	Symbol
stdout	unknown	221	46 6f 75 6e 64 3a 20 32 20 65 78 70 6f 72 74 73 2c 20 63 61 6c 6c 69 6e 67 0d 0a 43 61 6c 6c 20 65 78 70 6f 72 74 73 20 32 0d 0a 53 75 63 63 65 73 73 66 75 6c 6c 79 20 63 61 6c 6c 65 64 20 63 6d 64 20 6c 69 6e 65 20 72 75 6e 64 6c 6c 33 32 2e 65 78 65 20 43 3a 5c 55 73 65 72 73 5c 6c 75 6b 65 74 61 79 6c 6f 72 5c 44 65 73 6b 74 6f 70 5c 5a 75 44 42 59 69 4f 76 74 34 2e 64 6c 6c 2c 23 31 0d 0a 53 75 63 63 65 73 73 66 75 6c 6c 79 20 63 61 6c 6c 65 64 20 63 6d 64 20 6c 69 6e 65 20 72 75 6e 64 6c 6c 33 32 2e 65 78 65 20 43 3a 5c 55 73 65 72 73 5c 6c 75 6b 65 74 61 79 6c 6f 72 5c 44 65 73 6b 74 6f 70 5c 5a 75 44 42 59 69 4f 76 74 34 2e 64 6c 6c 2c 23 32 0d 0a	Found: 2 exports, calling..Call exports 2..Successfully called cmd line rundll32.exe C:\Users\user\Desktop\ZuDBYiOvt4.dll,#1..Successfully called cmd line rundll32.exe C:\Users\user\Desktop\ZuDBYiOvt4.dll, #2..	success or wait	1	10CA7D2	WriteFile

Analysis Process: rundll32.exe PID: 3556 Parent PID: 3548

General

Start time:	01:53:21
Start date:	29/07/2008
Path:	C:\Windows\System32\rundll32.exe
Wow64 process (32bit):	false
Commandline:	rundll32.exe C:\Users\user\Desktop\ZuDBYiOvt4.dll,#1
Imagebase:	0x6d0000
File size:	44544 bytes
MD5 hash:	51138BEEA3E2C21EC44D0932C71762A8
Has administrator privileges:	true
Programmed in:	C, C++ or other language
Reputation:	moderate

File Activities

File Path	Access	Attributes	Options	Completion	Count	Source Address	Symbol

File Path	Offset	Length	Value	Ascii	Completion	Count	Source Address	Symbol

File Path	Offset	Length	Completion	Count	Source Address	Symbol

Registry Activities

Key Path	Completion	Count	Source Address	Symbol

Key Path	Name	Type	Data	Completion	Count	Source Address	Symbol

Analysis Process: msupdate.exe PID: 3564 Parent PID: 3556

Christopher S. Culling, csculling1@gmail.com

General

Start time:	01:53:21
Start date:	29/07/2008
Path:	C:\Users\user\AppData\Local\Temp\msupdate.exe
Wow64 process (32bit):	false
Commandline:	C:\Users\user~1\AppData\Local\Temp\msupdate.exe
Imagebase:	0x400000
File size:	106496 bytes
MD5 hash:	DDC9915A5158A9560AD1EF2F16CCE9A6
Has administrator privileges:	true
Programmed in:	C, C++ or other language
Reputation:	low

File Activities

File Path	Access	Attributes	Options	Completion	Count	Source Address	Symbol

File Path	Offset	Length	Value	Ascii	Completion	Count	Source Address	Symbol

File Path	Offset	Length	Completion	Count	Source Address	Symbol

Registry Activities

Key Path	Completion	Count	Source Address	Symbol

Key Path	Name	Type	Data	Completion	Count	Source Address	Symbol

Analysis Process: rundll32.exe PID: 3588 Parent PID: 3564

General

Start time:	01:53:22
Start date:	29/07/2008
Path:	C:\Windows\System32\rundll32.exe
Wow64 process (32bit):	false
Commandline:	rundll32 C:\Windows\MSAgent\AGENTCPD.DLL _start@16 0
Imagebase:	0x6d0000
File size:	44544 bytes
MD5 hash:	51138BEEA3E2C21EC44D0932C71762A8
Has administrator privileges:	true
Programmed in:	C, C++ or other language
Reputation:	moderate

File Activities

File Path	Offset	Length	Completion	Count	Source Address	Symbol

Analysis Process: rundll32.exe PID: 3692 Parent PID: 3548

General

Start time:	01:53:24
Start date:	29/07/2008
Path:	C:\Windows\System32\rundll32.exe
Wow64 process (32bit):	false
Commandline:	rundll32.exe C:\Users\user\Desktop\ZuDBYiOvt4.dll,#2

Christopher S. Culling, csculling1@gmail.com

Imagebase:	0x6d0000
File size:	44544 bytes
MD5 hash:	51138BEEA3E2C21EC44D0932C71762A8
Has administrator privileges:	true
Programmed in:	C, C++ or other language
Reputation:	moderate

File Activities

File Path	Offset	Length	Completion	Count	Source Address	Symbol

Disassembly

Code Analysis

Christopher S. Culling, csculling1@gmail.com

Last Updated: August 19th, 2018

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS Chicago 2018	Chicago, ILUS	Aug 20, 2018 - Aug 25, 2018	Live Event
Data Breach Summit & Training 2018	New York City, NYUS	Aug 20, 2018 - Aug 27, 2018	Live Event
SANS Prague 2018	Prague, CZ	Aug 20, 2018 - Aug 25, 2018	Live Event
SANS Virginia Beach 2018	Virginia Beach, VAUS	Aug 20, 2018 - Aug 31, 2018	Live Event
SANS San Francisco Summer 2018	San Francisco, CAUS	Aug 26, 2018 - Aug 31, 2018	Live Event
SANS Copenhagen August 2018	Copenhagen, DK	Aug 27, 2018 - Sep 01, 2018	Live Event
SANS SEC504 @ Bangalore 2018	Bangalore, IN	Aug 27, 2018 - Sep 01, 2018	Live Event
SANS Tokyo Autumn 2018	Tokyo, JP	Sep 03, 2018 - Sep 15, 2018	Live Event
SANS Wellington 2018	Wellington, NZ	Sep 03, 2018 - Sep 08, 2018	Live Event
SANS Amsterdam September 2018	Amsterdam, NL	Sep 03, 2018 - Sep 08, 2018	Live Event
SANS Tampa-Clearwater 2018	Tampa, FLUS	Sep 04, 2018 - Sep 09, 2018	Live Event
SANS MGT516 Beta One 2018	Arlington, VAUS	Sep 04, 2018 - Sep 08, 2018	Live Event
Threat Hunting & Incident Response Summit & Training 2018	New Orleans, LAUS	Sep 06, 2018 - Sep 13, 2018	Live Event
SANS Baltimore Fall 2018	Baltimore, MDUS	Sep 08, 2018 - Sep 15, 2018	Live Event
SANS Alaska Summit & Training 2018	Anchorage, AKUS	Sep 10, 2018 - Sep 15, 2018	Live Event
SANS Munich September 2018	Munich, DE	Sep 16, 2018 - Sep 22, 2018	Live Event
SANS London September 2018	London, GB	Sep 17, 2018 - Sep 22, 2018	Live Event
SANS Network Security 2018	Las Vegas, NVUS	Sep 23, 2018 - Sep 30, 2018	Live Event
SANS DFIR Prague Summit & Training 2018	Prague, CZ	Oct 01, 2018 - Oct 07, 2018	Live Event
Oil & Gas Cybersecurity Summit & Training 2018	Houston, TXUS	Oct 01, 2018 - Oct 06, 2018	Live Event
SANS Amsterdam October 2018	Amsterdam, NL	Oct 08, 2018 - Oct 13, 2018	Live Event
SANS Brussels October 2018	Brussels, BE	Oct 08, 2018 - Oct 13, 2018	Live Event
SANS Northern VA Fall- Tysons 2018	Tysons, VAUS	Oct 13, 2018 - Oct 20, 2018	Live Event
SANS Riyadh October 2018	Riyadh, SA	Oct 13, 2018 - Oct 18, 2018	Live Event
SANS London October 2018	London, GB	Oct 15, 2018 - Oct 20, 2018	Live Event
SANS October Singapore 2018	Singapore, SG	Oct 15, 2018 - Oct 27, 2018	Live Event
SANS Seattle Fall 2018	Seattle, WAUS	Oct 15, 2018 - Oct 20, 2018	Live Event
SANS Denver 2018	Denver, COUS	Oct 15, 2018 - Oct 20, 2018	Live Event
Secure DevOps Summit & Training 2018	Denver, COUS	Oct 22, 2018 - Oct 29, 2018	Live Event
SANS Houston 2018	Houston, TXUS	Oct 29, 2018 - Nov 03, 2018	Live Event
SANS Gulf Region 2018	Dubai, AE	Nov 03, 2018 - Nov 15, 2018	Live Event
SANS Dallas Fall 2018	Dallas, TXUS	Nov 05, 2018 - Nov 10, 2018	Live Event
SANS Krakow 2018	OnlinePL	Aug 20, 2018 - Aug 25, 2018	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced