Hunting for Ransomware

blog.reversinglabs.com/blog/hunting-for-ransomware

Security Operations | January 24, 2020


Blog Author

Robert Simmons, Independent malware researcher and threat researcher at ReversingLabs. Read More...

November Update:

Here's your opportunity to hear directly from Rob Simmons, Threat Researcher involved in #Ryuk ransomware research.

Join us to learn:

- The current state of Ransomware and how it is becoming more targeted
- How to use the A1000 to hunt for threats using YARA
- How to bring new visibility about file risks into your SOC process
- How to apply this new intelligence on Ryuk to actively update your defenses

Register for our November 17 webinar here: https://reversinglabs.zoom.us/webinar/register/6215881027977/WN_X6tAd0-NTeSRllyjtEQS0g

Many ransomware families have changed their tactics and victim-targeting in recent years. Rather than indiscriminate attacks against anyone they're able to infect, they have moved to a process called "big game hunting". The motivation underlying this change of tactics is to increase the potential payout by targeting an organization rather than an individual. The adversary performs extensive reconnaissance on the target to determine what they may be able to pay. Rather than small ransom demands in thousands of dollars, by targeting businesses, they are aiming for payouts in the hundreds of thousands to millions of dollars.

One malware family in particular, Ryuk [1], has been attributed to the GRIM SPIDER [2] threat actor group. According to malpedia.io, this group has been operating the Ryuk ransomware

since August of 2018 [3]. In recent months, a staged attack dubbed “triple threat” [4] has emerged with the initial access to the network achieved by the Emotet [5] malware family. Once initial access is achieved, the next stage, TrickBot [6], delivered inside the target organization. TrickBot has capabilities to steal credentials and to move laterally within the organization’s network. The third stage of the attack is to execute Ryuk ransomware on as many workstations and servers as possible via the lateral movement of TrickBot.
To hunt for and identify Ryuk samples, many YARA [7] rules search for strings that are hard-coded in the sample. However, this type of strings-based rule may be prone to false positives. An excellent conference talk that includes this topic given by Lauren Pierce at ShmooCon 2017 should be watched for more information about this concept. [8] Rather than hunting for these hard-coded strings, one should be hunting for code patterns in the sample. Rules of this type do more damage to the adversary’s intrusion set according to David Bianco’s Pyramid of Pain. [9] More painful code changes are needed to avoid detection by this paradigm of YARA rule. Here, we examine a single algorithm that Ryuk uses in the latest 64bit variant to generate a random string. This string is part of the filename that Ryuk uses when dropping a copy of itself during the installation phase of intrusion.

Looking at the execution of the Ryuk sample [10] in x64dbg, [11] we see that the first step taken is to gather entropy from the tick count of the victim’s computer. In Figure 1, we see the library function call to GetTickCount to gather this randomness.

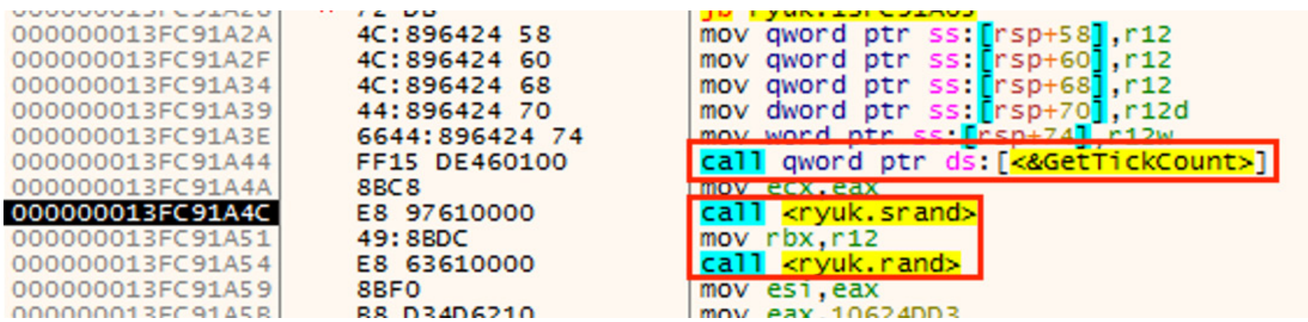


Figure 1: Entropy Input From Tick Count

According to Microsoft’s documentation, GetTickCount returns “the number of milliseconds that have elapsed since the system was started.” [12] The function called immediately after is a C library function, srand. This function takes a seed value and initializes the random number generator. The srand and rand functions’ identities were detected using Ghidra’s [13] function signatures during its code analysis process.

The random number generator initialized by srand is subsequently used by rand function calls to generate random data. The goal of generating this data is to produce a random string. However, not all the bytes of randomness generated can be used in a filename, so a subsequent function checks the output to verify that the generated byte is an alphabet character and therefore valid for a filename. This function has been labelled as “isalpha” in

Figure 2.

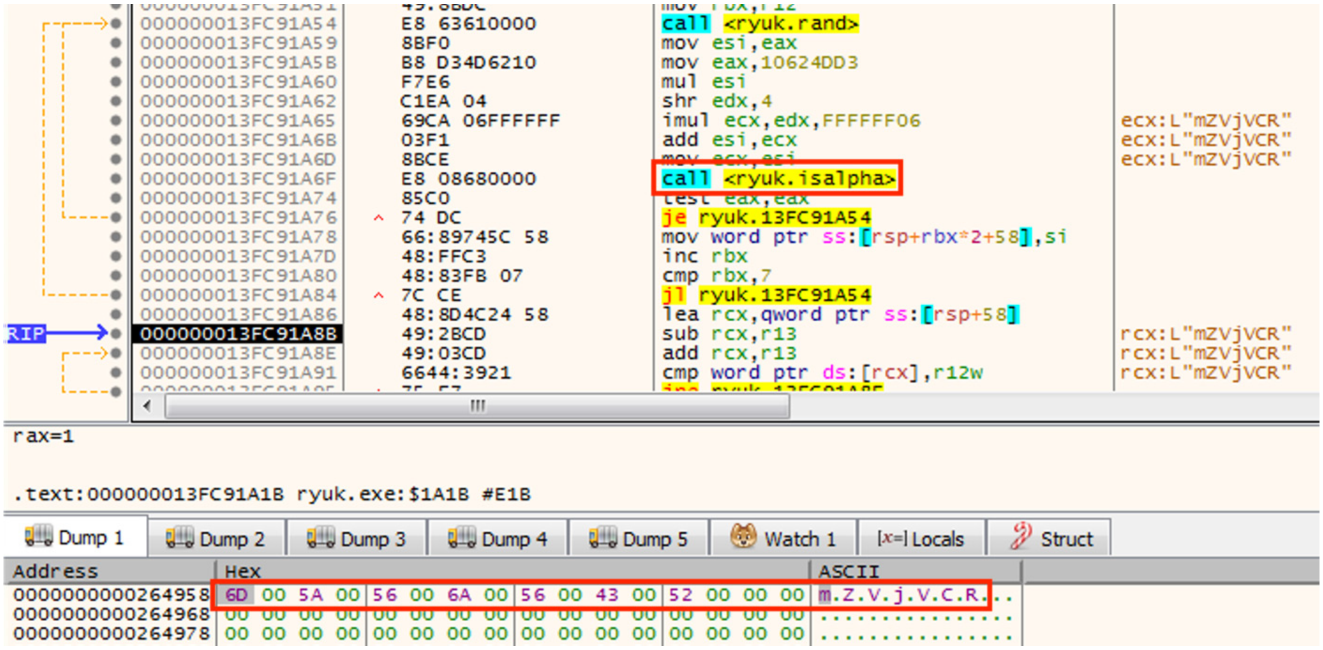


Figure 2: Repeat Until String is Alphabet Characters

If the byte fails this test, execution jumps back to the rand function and a new random byte is generated. This loop continues until all the characters are alphabet characters, and the generated string is therefore usable as a filename.

To write an effective YARA rule for detecting this algorithm, first we examine the srand function and find a hexadecimal string that can be used to match the function. Figure 3 shows the srand function in the debugger's disassembler.

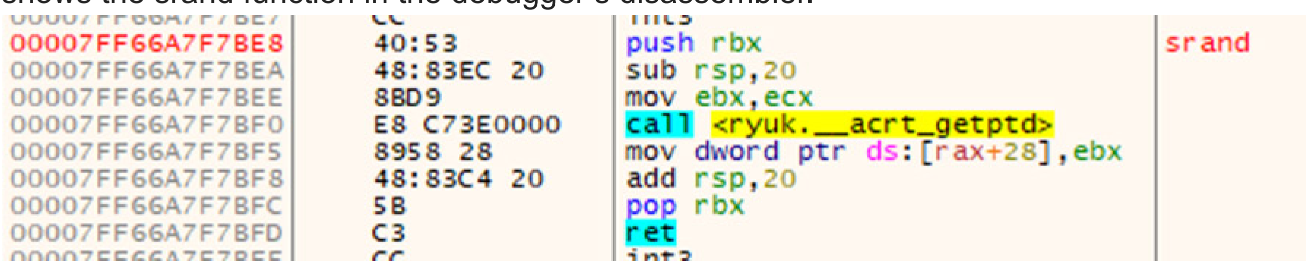


Figure 3: Disassembled srand Function

The goal is to identify enough bytes from this function to differentiate it from other functions in the sample, but still allow enough wiggle room for slight changes due to the compiler.

$srand = { 40 53 48 83 ?? 20 8B ?? E8 [4] 89 }

The hexadecimal string seen above identifies the srand function, but leaves room for the destination registers to change and still match the function. [14] These wildcards are represented as "??". The four byte jump "[4]" allows for the address of the "__acrt_getptd" function to change locations.

Next we repeat the same process by examining the rand function in the debugger.

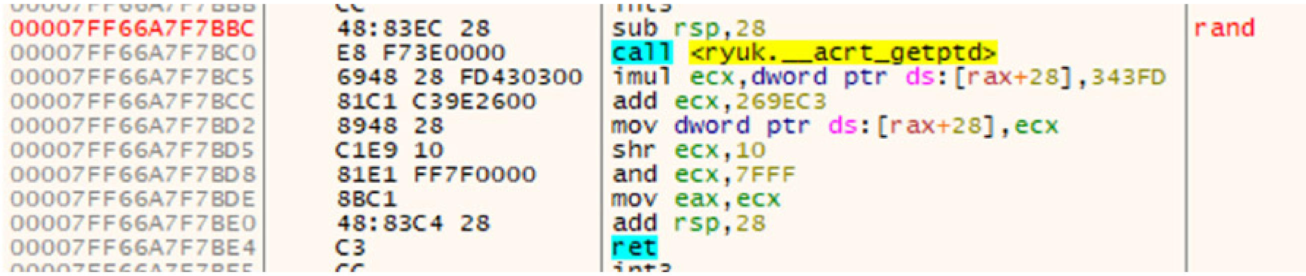


Figure 4: Disassembled rand Function

For this function the following hexadecimal string identifies it and differentiates it from other similar functions in the sample.

$rand = { 48 83 ?? 28 E8 [4] 69 }

Again, wildcards are used to allow for changes in destination registers as well as a four byte jump that allows for the location of the called function to change.

Next we analyze the “isalpha” function that is called to check if the random byte is an alphabet character. This function is not a library function. It is adversary written code and a control flow graph of the function is seen in Figure 5.

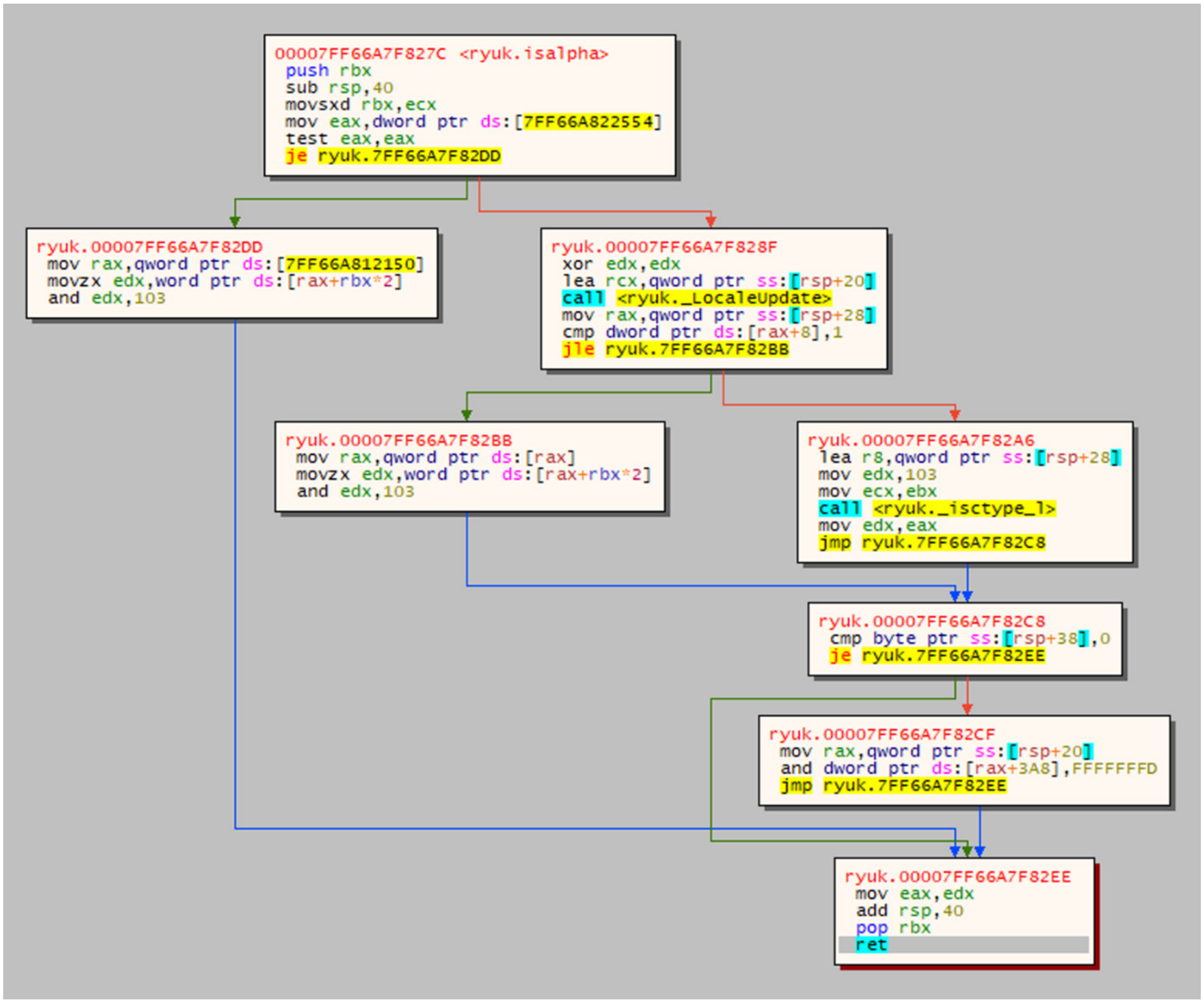


Figure 5: Control Flow Graph of the isalpha Function

To develop a signature that detects this function, we look more closely at the very first code block.

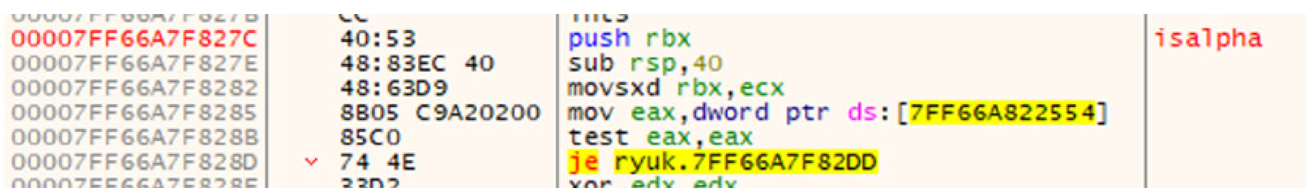


Figure 6: First Code Block of isalpha Function

Following the same methodology to write a signature as above, destination registers except for the opcode "movsxd" are replaced with wildcards. Then the location of the pointer from the "mov" instruction is replaced with a four byte jump. The resulting hexadecimal string is as follows:

$isalpha = { 40 53 48 83 EC ?? 48 63 D9 8B 05 [4] 85 C0 74 4E }

Armed with the locations of the three functions, we return to analyzing the code that is used to call them. This code can be split into two separate opcode signatures. This will allow for variation in the code between these two code snippets thereby still identifying this adversary code even if it changes slightly as the ransomware code is developed for new variants of Ryuk.

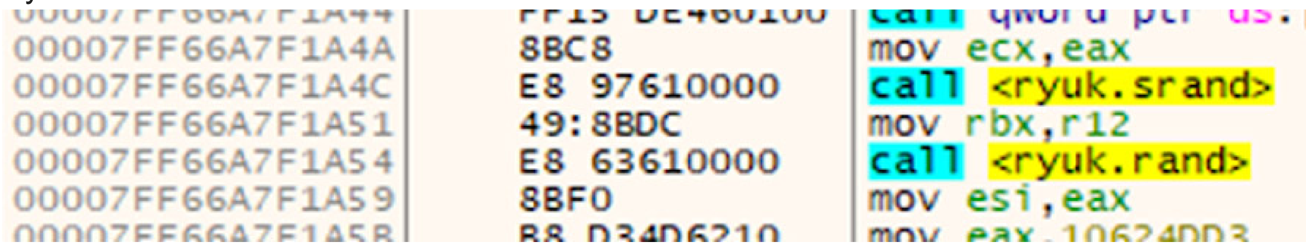


Figure 7: First Set of Instructions as Seen in Debugger

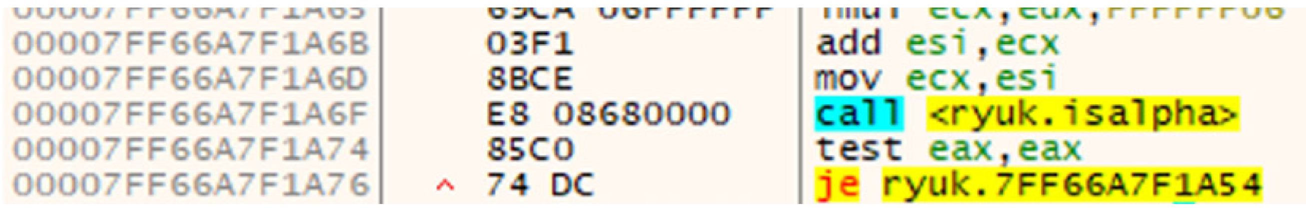


Figure 8: Second Set of Instructions as Seen in Debugger

By following the process of allowing for variation in destination registers as well as the location of the called functions, the following two opcode signatures are developed:

$op1 = { E8 [4] 49 8B ?? E8 [4] ?? }
$op2 = { 03 ?? 8B ?? E8 [4] 85 C0 74 ?? }

Now that we have signatures for the functions that are called as well as signatures for the code that calls them, we tie these together by comparing the bytes found in the opcode that calls the function with the location of the called function. This is done by using YARA condition statements to calculate the locations. This first condition statement verifies that the first opcode calls the srand function:

uint32(@op1 + 1) + @op1 + 5 == @srand

This condition verifies that the first opcode then calls the rand function:

uint32(@op1 + 9) + @op1 + 13 == @rand

And this condition verifies that the second opcode calls the isalpha function:

uint32(@op2 + 5) + @op2 + 9 == @isalpha

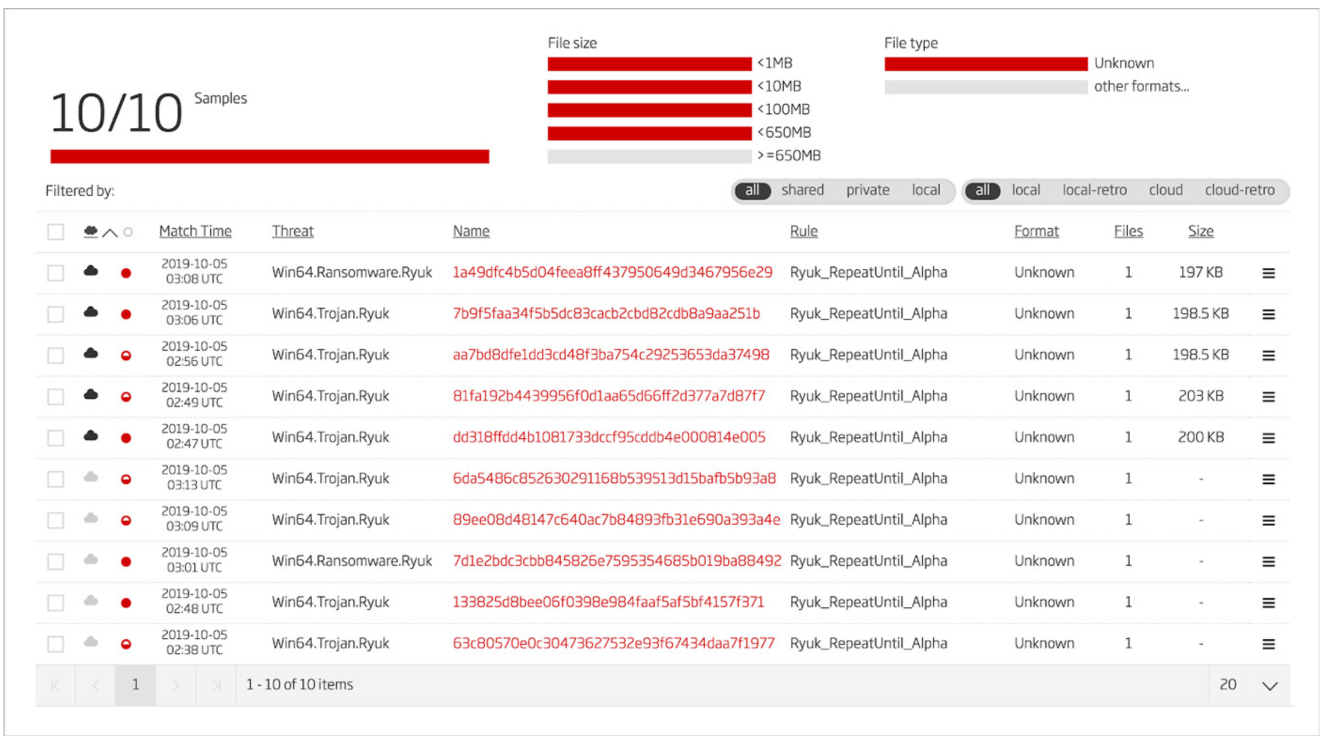
The last instruction seen in Figure 8 is a jump that leads back to the rand function to generate a new byte of random data if the previous byte was not an alphabet character. The following condition reflects this jump and allows for the landing address of the jump to change based on differences at compile time or code changes between the two opcode snippets.

@op2 + 5 + int8(@op2 + 12) == @op1

Now that we have a fully formed YARA rule, we can hunt for samples of Ryuk using the Titanium Platform that are related to the one that we started with. The complete YARA rule is provided at the bottom.

Hunting for Ryuk

By loading the YARA rule into the ReversingLabs A1000's threat hunting system, we discover that the rule is highly accurate and has matched nine other Ryuk 64bit samples that are all related to the sample that we started with.



Match Time	Threat	Name	Rule	Format	Files	Size
2019-10-05 03:08 UTC	Win64.Ransomware.Ryuk	1a49dfc4b5d04feea8ff437950649d3467956e29	Ryuk_RepeatUntil_Alpha	Unknown	1	197 KB
2019-10-05 03:06 UTC	Win64.Trojan.Ryuk	7b9f5faa34f5b5dc83cacb2cbd82cdb8a9aa251b	Ryuk_RepeatUntil_Alpha	Unknown	1	198.5 KB
2019-10-05 02:56 UTC	Win64.Trojan.Ryuk	aa7bd8dfe1dd3cd48f3ba754c29253653da37498	Ryuk_RepeatUntil_Alpha	Unknown	1	198.5 KB
2019-10-05 02:49 UTC	Win64.Trojan.Ryuk	81fa192b4439956f0d1aa65d66ff2d377a7d87f7	Ryuk_RepeatUntil_Alpha	Unknown	1	203 KB
2019-10-05 02:47 UTC	Win64.Trojan.Ryuk	dd318ffdd4b1081733dccf95cddb4e000814e005	Ryuk_RepeatUntil_Alpha	Unknown	1	200 KB
2019-10-05 03:13 UTC	Win64.Trojan.Ryuk	6da5486c852630291168b539513d15bafb5b93a8	Ryuk_RepeatUntil_Alpha	Unknown	1	-
2019-10-05 03:09 UTC	Win64.Trojan.Ryuk	89ee08d48147c640ac7b84893fb31e690a393a4e	Ryuk_RepeatUntil_Alpha	Unknown	1	-
2019-10-05 03:01 UTC	Win64.Ransomware.Ryuk	7d1e2bdc3cbb845826e7595354685b019ba88492	Ryuk_RepeatUntil_Alpha	Unknown	1	-
2019-10-05 02:48 UTC	Win64.Trojan.Ryuk	133825d8bee06f0398e984faaf5af5bf4157f371	Ryuk_RepeatUntil_Alpha	Unknown	1	-
2019-10-05 02:38 UTC	Win64.Trojan.Ryuk	63c80570e0c30473627532e93f67434daa7f1977	Ryuk_RepeatUntil_Alpha	Unknown	1	-

Figure 9: YARA Hunting Results

Looking at each sample's analysis results, we can additionally see that the ReversingLabs Hash Algorithm [15] has associated these same files together as a cluster.

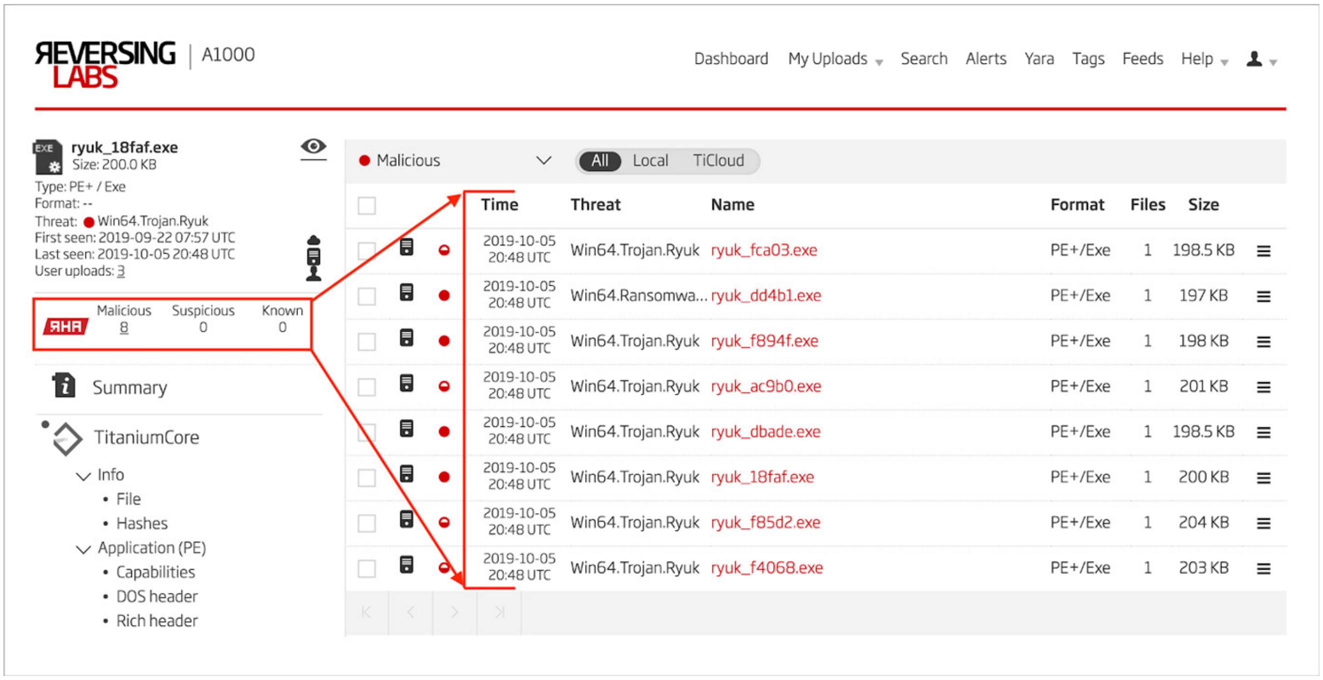


Figure 10: ReversingLabs Hash Algorithm Cluster

Next, we can see that the Titanum Platform has determined the threat name as "Win64.Trojan.Ryuk" for each of the identified samples via the cloud classification system.

Figure 11: Titanium Cloud Classification as Win64.Trojan.Ryuk

Finally, we can drill into the file's indicators and see what has been extracted during analysis by ReversingLabs Titanium Platform. In Figure 12, we see some of the hallmarks of ransomware: tampering with security products to disable them, disabling backups to prevent data recovery, writing and deleting files during the encryption process, stopping services and processes so that more data can be encrypted, and usage of cmd.exe to run CLI commands.

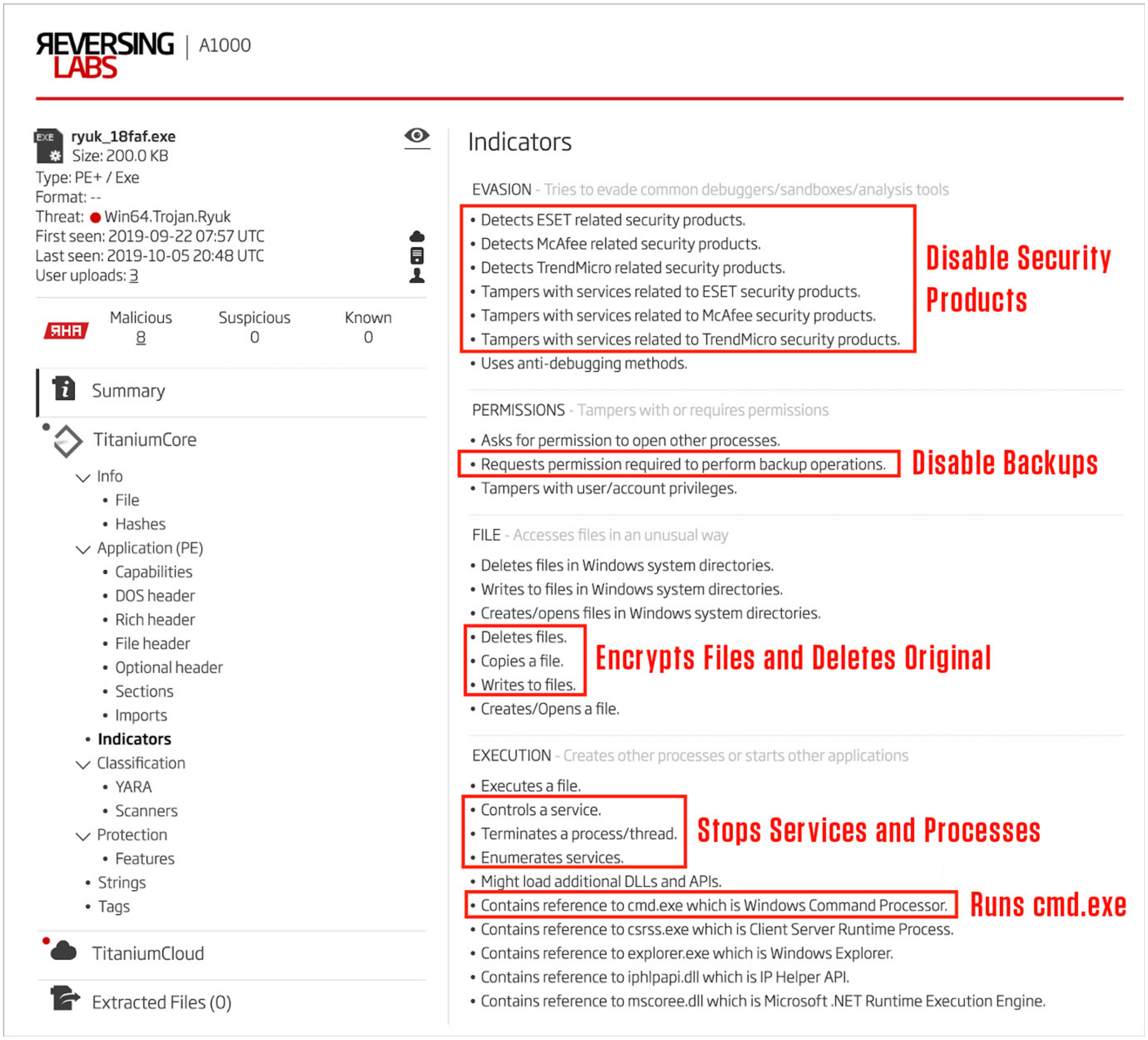


Figure 12: Indicators Detected by Titanium Platform

As we have seen, by starting with one sample, and analyzing its code, a YARA signature can be developed to identify more related samples. Furthermore, by leveraging the Titanium Platform, these related files can be confirmed as being related. If further analysis is warranted, static features analysis in the A1000 allows the researcher to delve deeper into the capabilities of the ransomware and its related samples from a particular campaign.

YARA Rule

Ryuk64

```
rule RepeatUntil_Alpha : Ryuk64
{
meta:
author = "Malware Utkonos"
date = "2019-09-22"
exemplar = "18faf22d7b96bfdb5fd806d4fe6fd9124b665b571d89cb53975bc3e23dd75ff1"
description = "Repeat generation of random data until filename string is all alpha characters"
strings:
$srand = { 40 53 48 83 ?? 20 8B ?? E8 [4] 89 }
$rand = { 48 83 ?? 28 E8 [4] 69 }
$isalpha = { 40 53 48 83 EC ?? 48 63 D9 8B 05 [4] 85 C0 74 4E }
$op1 = { E8 [4] 49 8B ?? E8 [4] ?? }
$op2 = { 03 ?? 8B ?? E8 [4] 85 C0 74 ?? }
condition:
WindowsPE and all of them and
uint32(@op1 + 1) + @op1 + 5 == @srand and // call srand
uint32(@op1 + 9) + @op1 + 13 == @rand and // call rand
uint32(@op2 + 5) + @op2 + 9 == @isalpha and // call isalpha
@op2 + 5 + int8(@op2 + 12) == @op1 // jump to rand call until all characters are alpha
}
```

[1] https://malpedia.caad.fkie.fraunhofer.de/details/win.ryuk

[2]

[3] ibid.

[4] https://searchsecurity.techtarget.com/news/252461071/Triple-threat-malware-campaign-combines-Emotet-TrickBot-and-Ryuk

[5]

[6] https://malpedia.caad.fkie.fraunhofer.de/details/win.trickbot

[7]

[8] https://youtu.be/_BfLSRjHWo8?t=1252

[9] https://detect-respond.blogspot.com/2013/03/the-pyramid-of-pain.html

[10] 18faf22d7b96bfdb5fd806d4fe6fd9124b665b571d89cb53975bc3e23dd75ff1

[11] https://x64dbg.com/

[12] https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-gettickcount

[13] https://ghidra-sre.org/

[14] Thanks to Wesley Shields https://twitter.com/wxs for this critical signature technique.

[15] https://www.reversinglabs.com/technology/reversinglabs-hash-algorithm

MORE BLOG ARTICLES