Tracing fresh Ryuk campaigns itw

blog.virustotal.com/2020/10/tracing-fresh-ryuk-campaigns-itw.html

Ryuk is one of the most dangerous Ransomware families. It is (allegedly) run by a specialized cybercrime actor that during the last 2 years mainly focused on targeting enterprise environments. The amount of bitcoins demanded in their ransom attacks varies depending on the target. Some of the wallets used by the group to collect the ransom payments reached millions of dollars in a few weeks.

Protecting against such attacks is one of the main priorities for any CISO or security team. This is a problem that should be approached from different perspectives, being prevention (likely) the most relevant one.

Now, what can be done in terms of prevention? Information is power, the first thing we need is understanding how the new campaigns are operating. Is this distributed through phishing or exploiting any vulnerabilities? Do they use brute force attacks? Maybe all together?

In addition to the TTPs described above, we want as many technical details as possible. This will result in very valuable Indicators of Compromise we can use for protecting our infrastructure: deploying networking indicators to disrupt malware communication, making sure our Yara rules will detect all the components of the attack, launching regular scans in our infrastructure to detect any artefact used in the campaign.

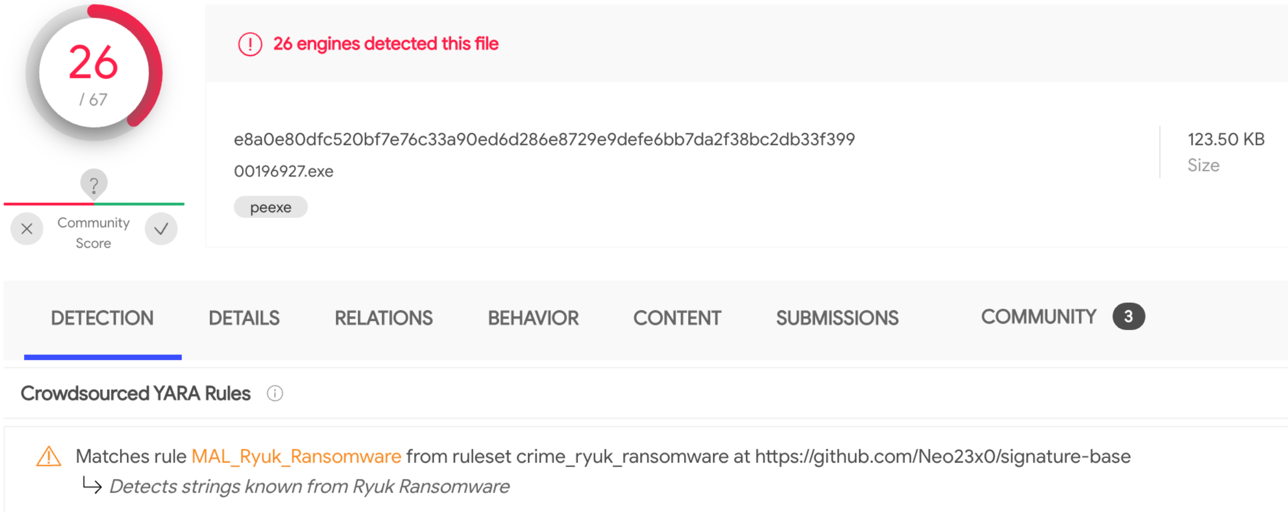
We need to quickly deploy our fishing nets to catch everything related to fresh new campaigns! And then to keep monitoring for a while to make sure we keep our systems updated as attackers evolve.

In this blogpost we will describe how we used VirusTotal to detect and monitor new Ryuk activity. However this is a very specific case where we want to show how our IDA plugin can save us a lot of time when dealing with certain samples.

If you want to learn more about how you can keep your organization safe from ransomware and how to easily leverage VirusTotal to monitor ransomware activity, please join us for our next Anti-ransomware workshop - English (Live November 4th, 1pm ET) and Spanish (Live October 28th, 17:00 CEST) versions available.

Starting the investigation

Two weeks ago new files were uploaded to VirusTotal (1, 2). According to the crowdsourced YARA rule that identified them, these files looked like Ryuk malware.



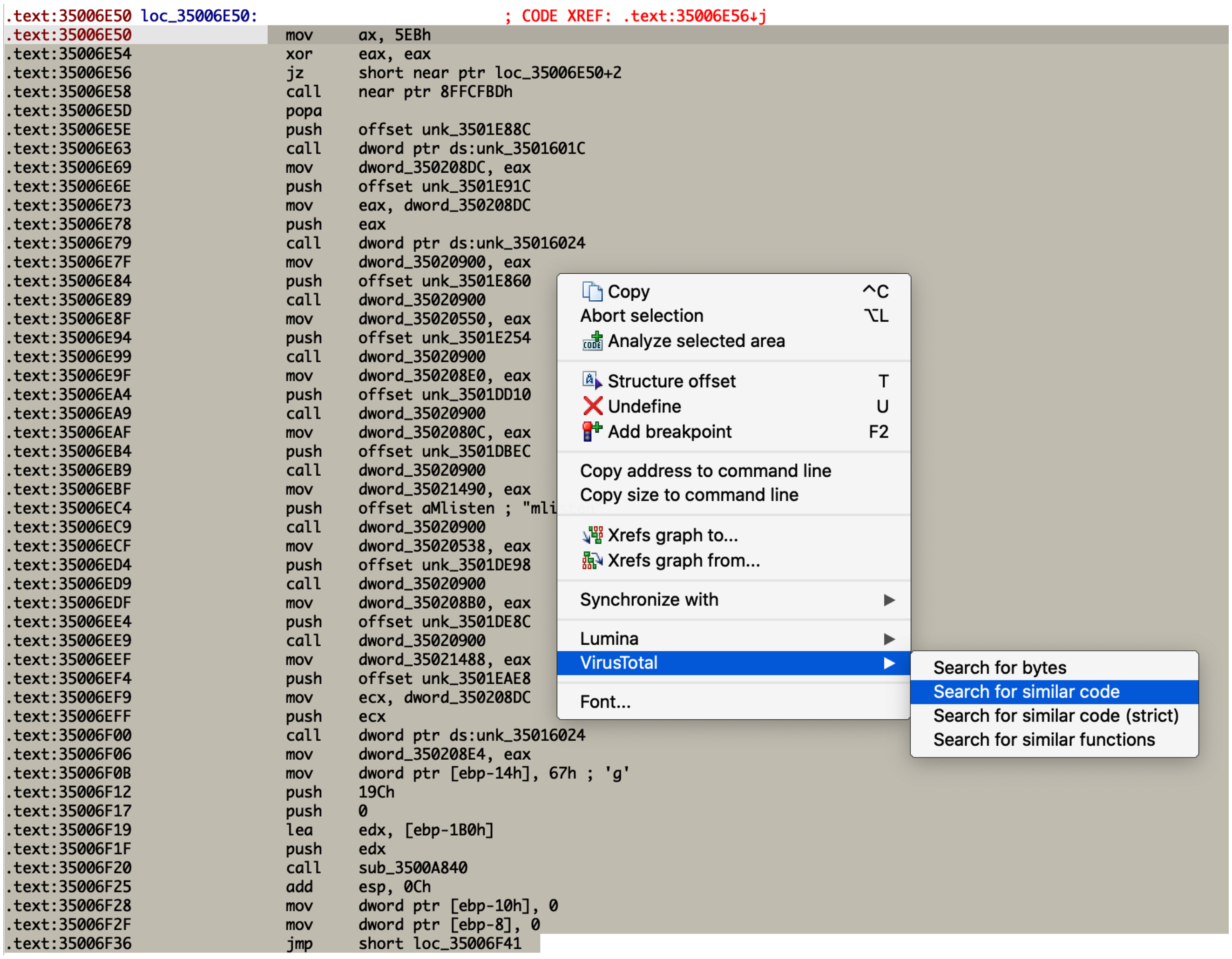
A closer look revealed that these samples have been probably dumped from memory: the disassembled code showed plenty of memory mapped addresses, the import table was missing and the samples crashed when executed - they were definitively corrupted PE files.

Given these were fresh samples, we certainly wanted to know more about them, especially if they were part of a bigger campaign. In such cases, one of our best allies is looking for similar samples that could also be part of the attack. However, when working with memory dumps we need to be careful, given that probably some segments and memory mapped addresses will be execution specific. If we include any of such specifics in our search, we won't be able to find other samples.

IDA plugin to the rescue

One of the options would be to rebuild the samples we found, which is an extremely time consuming process. Instead, we can use the VirusTotal IDA plugin (see original blog post announcement) to help us search for the original sample. Using the "search for similar code" functionality we can create a query that will ignore all the memory mapped addresses, being a perfect choice for our problem.

Taking a look at the samples with IDA, we can see there are many functions that aren't properly identified by the disassembler engine given the use of anti-disassembly techniques. Precisely for this reason, they are good choices for searching for code similarity.



We just need to select the code, right-button, and search for similar code. The resulting query will take care of ignoring all the memory mapped addresses we wanted to get rid of.

FILES 13

	Detections	Size	First seen	Last seen	Submitters
C952328213342E03FEBA9C61ED7F968D402439ED56555389232CBD0C59D7B155 C:\Users\<USER>\AppData\Local\Temp\VfaNvpXYhlan.exe peexe	44 / 70	128.50 KB	2020-09-12 20:42:42	2020-09-12 20:42:42	1
53EE2767E91EE2AD7A9A9DA8CA28D6DC56EEC47E4F407B5DF0BDB515F5BDD20F /data/modqi/samples/sum/53ee2767e91ee2ad7a9a9da8ca28d6dc56eec47e4f407b5df0bdb515f5bdd20f peexe	33 / 71	128.00 KB	2020-09-21 01:13:24	2020-09-27 01:07:45	2
E2F4417566A1D3BE7FF4CE5EE8857073668DFACF3D3B740F3E1858945816260A peexe	37 / 70	127.00 KB	2020-09-23 06:10:05	2020-09-23 06:10:05	1
D7333223DCC1002AAE04E25E31D8C297EFA791A2C1E609D67AC6D9AF338EFBE8 097cb948a1f011f5de11579849a08db5.virus peexe	46 / 70	120.50 KB	2020-09-26 12:20:10	2020-09-26 12:20:10	1
D0D7A8F588693B7CC967FB4069419125625EB7454BA553C0416F35FC95307CBE e8673c8a299d1647ead6f3da4565ac54.virus peexe	38 / 70	127.50 KB	2020-09-26 12:26:37	2020-09-26 12:26:37	1
E72C36DFFCE4BB998EC1A35DA6C57156A08897EB5196DC969D5F8066E5AD5E13 e72c36dffce4bb998ec1a35da6c57156a08897eb5196dc969d5f8066e5ad5e13.bin peexe	28 / 71	123.50 KB	2020-09-29 05:43:35	2020-09-29 05:43:35	1
CFE1678A7F2B949966D9A020FAAFB46662584F8A6AC4B72583A21FA858F2A2E8 00228416.exe peexe	29 / 70	123.50 KB	2020-09-30 02:58:04	2020-09-30 02:58:04	1
E8A0E80DFC520BF7E76C33A90ED6D286E8729E9DEFE6BB7DA2F38BC2DB33F399 00196927.exe peexe	26 / 67	123.50 KB	2020-09-30 04:38:00	2020-09-30 04:38:00	1
CFDC2CB47EF3D2396307C487FC3C9FE55B3802B2E570BEE9AEA4AB1E4ED2EC28 e8a0e80dfc520bf7e76c33a90ed6d286e8729e9defe6bb7da2f38bc2db33f399.bin_unmapped peexe overlay	10 / 70	118.00 KB	2020-09-30 13:42:21	2020-09-30 13:42:21	1
96E2DDB2F05EF68B6B2E31DC33D75530F3E4E3932F0A1E619D00D8BA7990E088 v2.exe peexe	31 / 70	124.50 KB	2020-10-01 17:40:01	2020-10-01 17:40:01	1
5212B76A3E238759AFFDAD53865553E9D80F6DAFDAD27A57950E6450987371F2 v2.exe peexe	45 / 70	124.50 KB	2020-10-01 17:55:20	2020-10-01 17:55:20	1

The resulting listing with all the files found shows very close first submission time. Also, some of them report behaviour activity, meaning they executed in the sandboxes without crashing: maybe one of them could be our original sample.

Picking one of our initial samples and another one with behavioural information, we can see that:

- They don't show up as similar when doing a similarity search (as expected).
- They have some long sequences of bytes in common.

DIFF PATTERNS 35

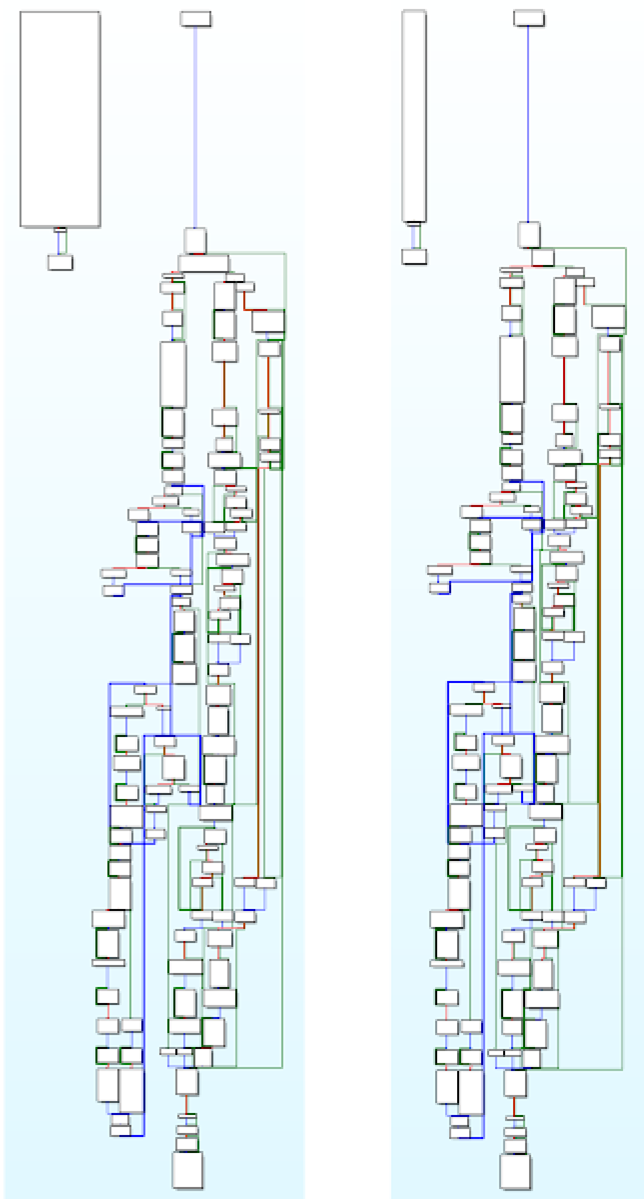
```
35 9C 64 01 35 A4 64 01 35 B0 64 01 35 BC 64 01   5·d·5·d·5·d·5·d·
35 C8 64 01 35 D4 64 01 35 E4 64 01 35 F0 64 01   5·d·5·d·5·d·5·d·
35 F8 64 01 35 00 65 01 35 0C 65 01 35 18 65 01   5·d·5·e·5·e·5·e·
35 22 65 01 35 24 65 01 35 2C 65 01 35 34 65 01   5"e·5$e·5,e·54e·
35 38 65 01 35 3C 65 01 35 40 65 01 35 44 65 01   58e·5<e·5@e·5De·
35 48 65 01 35 4C 65 01 35 50 65 01 35 5C 65 01   5He·5Le·5Pe·5\e·
35 60 65 01 35 64 65 01 35 68 65 01 35 6C 65 01   5`e·5de·5he·5le·
35 70 65 01 35 74 65 01 35 78 65 01 35 7C 65 01   5pe·5te·5xe·5|e·
35 80 65 01 35 84 65 01 35 88 65 01 35 8C 65 01   5·e·5·e·5·e·5·e·
35 90 65 01 35 94 65 01 35 98 65 01 35 9C 65 01   5·e·5·e·5·e·5·e·
35 A0 65 01 35 A4 65 01 35 A8 65 01 35 AC 65 01   5·e·5·e·5·e·5·e·
35 B0 65 01 35 B4 65 01 35 B8 65 01 35 BC 65 01   5·e·5·e·5·e·5·e·
35 C0 65 01 35 C4 65 01 35 C8 65 01 35 CC 65 01   5·e·5·e·5·e·5·e·
35 D0 65 01 35 D4 65 01 35 D8 65 01 35 E4 65 01   5·e·5·e·5·e·5·e·
35 F0 65 01 35 F8 65 01 35 04 66 01 35 1C 66 01   5·e·5·e·5·f·5·f·
```

Matched bytes	Matches	Tags
394	2 / 2	binary

Is this our sample?

At this point we feel confident that the new sample found is the one we were looking for. Indeed, starting from this sample and taking a look at the (undetected) function located at 0x35008A60, we select a large sequence of instructions with the IDA plugin (as we did before) for a new search. This results in only 4 files that match the query generated: our two initial samples, another file that's also corrupted, and the previously chosen sample that detonated in our sandboxes. Therefore, this is the second time that we get this file when looking for similar code.

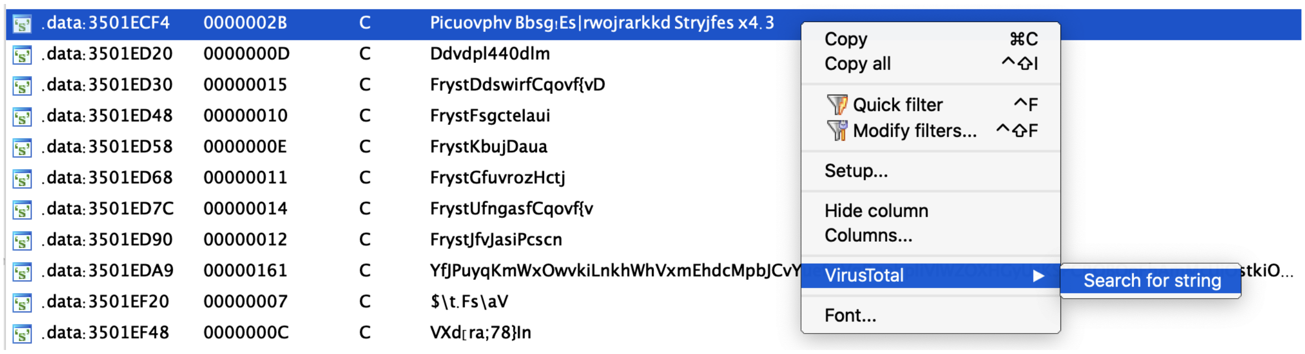
Going deeper, we'll see that it shares the same PE entry point that our two initial corrupted files. Furthermore, their WinMain functions are the same. Initially it looks like a quite simple function, composed of only three blocks of code. But, after overcoming the anti-disassembly trick implemented to confuse IDA, we can compare both function graphs to see the similarity. We conclude that we found the original sample.



Left: original sample, right: initial corrupted sample

What now?

At the time of this research there isn't any Yara rule detecting the original sample and it has 28/71 positives. Inside this file we can find encrypted strings that are extremely useful for pivoting to find additional samples. These strings are included in the corrupted files as well, stored in the ".gfids" segment at the end of the file. In other words, they aren't located in the ".data" segment as seen in the original sample. This new location reveals that probably these strings were initially encrypted and became decrypted after execution, thus they can be seen as footprints of the original sample.



Using the VT-IDA plugin we can search for other files that contain these encrypted strings. As expected, the four files found before are listed now, but there are two other samples that were submitted three days prior to our original sample and can also be investigated.

Moreover, all these new strings can be used to improve the original Yara rule that brought us here, or to create a new one! Remember to keep it running as a LiveHunt to make sure you keep track of any new Indicators of Compromise and to detect anything new attackers use in their campaigns. You can find all the details about the campaign described in this blogpost in the following VT-Graph.

This post was co-authored by Vicente Diaz.