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 - <u>English</u> (Live November 4th, 1pm ET) and <u>Spanish</u> (Live October 28th, 17:00 CEST) versions available.

Starting the investigation

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

26	(!) 26 e	() 26 engines detected this file												
7 67 ? X Community V Score		e8a0e80dfc520bf7e76c33a90ed6d286e8729e9defe6bb7da2f38bc2db33f399 123.5 00196927.exe Size												
DETECTION	DETAILS	RELATIONS	BEHAVIOR	CONTENT	SUBMISSIONS	COMMUNITY 3								
Crowdsourced YAR	A Rules 🕕													
		ansomware from r from Ryuk Ranson		k_ransomware a	t https://github.com/Ne	eo23x0/signature-base								

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 <u>VirusTotal IDA plugin</u> (see original blog post <u>announcement</u>) 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.

.text:35006E50	loc_35006E50:	; CO	DE XREF: .text:35006E56∔j		
.text:35006E50	mov	ax, 5EBh			
.text:35006E54	xor	eax, eax			
.text:35006E56	jz	short near ptr loc_3	5006E50+2		
.text:35006E58	call	near ptr 8FFCFBDh			
.text:35006E5D	рора				
.text:35006E5E	push	offset unk_3501E88C			
.text:35006E63	call	dword ptr ds:unk_350	1601C		
.text:35006E69	mov	dword_350208DC, eax			
.text:35006E6E	push	offset unk_3501E91C			
.text:35006E73	mov	eax, dword_350208DC			
.text:35006E78	push	eax			
.text:35006E79	call	dword ptr ds:unk_350	16024		
.text:35006E7F	mov	dword_35020900, eax			
.text:35006E84	push	offset unk_3501E860	Comu	^c	
.text:35006E89	call	dword_35020900	Сору	-	
.text:35006E8F	mo∨	dword_35020550, eax	Abort selection	TL	
.text:35006E94	push	offset unk_3501E254	🚮 Analyze selected area		
.text:35006E99	call	dword_35020900			
.text:35006E9F	mov	dword_350208E0, eax	A Structure offset	т	
.text:35006EA4	push	offset unk_3501DD10			
.text:35006EA9	call	dword_35020900	X Undefine	U	
.text:35006EAF	mov	dword_3502080C, eax	🚰 Add breakpoint	F2	
.text:35006EB4	push	offset unk_3501DBEC			
.text:35006EB9	call	dword_35020900	Copy address to command line		
.text:35006EBF	mov	dword_35021490, eax	Copy size to command line		
.text:35006EC4	push	offset aMlisten ; "m	li		
.text:35006EC9	call	dword_35020900	📲 Xrefs graph to		
.text:35006ECF	mov	dword_35020538, eax			
.text:35006ED4	push	offset unk_3501DE98	🚮 Xrefs graph from		
.text:35006ED9	call	dword_35020900			
.text:35006EDF	mov	dword_350208B0, eax	Synchronize with	▶	
.text:35006EE4	push	offset unk_3501DE8C			
.text:35006EE9	call	dword_35020900	Lumina		
.text:35006EEF	mov	dword_35021488, eax	VirusTotal		Search for bytes
.text:35006EF4	push	offset unk_3501EAE8			-
.text:35006EF9	mov	ecx, dword_350208DC	Font		Search for similar code
.text:35006EFF	push	ecx			Search for similar code (strict)
.text:35006F00	call	dword ptr ds:unk_350	16024		Search for similar functions
.text:35006F06	mov	dword_350208E4, eax			
.text:35006F0B	mov	dword ptr [ebp-14h],	67h ; 'g'		
.text:35006F12	push	19Ch			
.text:35006F17	push	0			
.text:35006F19	lea	edx, [ebp-1B0h]			
.text:35006F1F	push	edx			
.text:35006F20	call	sub_3500A840			
.text:35006F25	add	esp, 0Ch			
.text:35006F28	mov	dword ptr [ebp-10h],			
.text:35006F2F	mov	dword ptr [ebp-8], 0			
.text:35006F36	jmp	short loc_35006F41			

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		¢⊅	OD 💥	⊗ =	© ⊻
C952328213342E03FEBA9C61ED7F968D402439ED56555389232CBD0C59D7B155 🅞 🛛 🖂 👍 🗸	Detections Size	First seen	Last seen	Submitters	
C:\Users\-USER-\AppData\Local\TemplVfaNvpXYhlan.exe[]	44 / 70 128.50 KB	2020-09-12 20:42:42	2020-09-12 20:42:42	1	SC. EXE
53EE2767E91EE2AD7A9A90A8CA28060C56EEC47E4F40785DF8008515F580020F /data/modqi/samples/sum/53ee2767e91ee2ad7a9a9da8ca28d6dc56eec47e4f407b5df0bdb515f5bdd20f /data/modqi/samples/sum/53ee2767e91ee2ad7a9a9da8ca28d6dc56eec47e4f407b5df0bdb515f5bdd20f /data/modqi/samples/sum/53ee2767e91ee2ad7a9a9da8ca28d6dc56eec47e4f407b5df0bdb515f5bdd20f	33 / 71 128.00 KB	2020-09-21 01:13:24	2020-09-27 01:07:45	2	EXE
E2F4417566A1D3BE7FF4CE5EE8857073668DFACF3038740F3E1858945816260A	37 / 70 127.00 KB	2020-09-23 06:10:05	2020-09-23 06:10:05	1	EXE
D73332230CC1002AAE04E25E31D8C297EFA791A2C1E609067AC6D9AF338EFBE8 O97cb948atf01tf5de1t579849a08db5.virus	46 / 70 120.50 KB	2020-09-26 12:20:10	2020-09-26 12:20:10	1	exe exe
D0D7A8F58869387CC967F64069419125625EB7454BA553C0416F35FC95387CBE e8673c8a299d1647ead6f3da4565ac54.virus @ peexe	38 / 70 127.50 KB	2020-09-26 12:26:37	2020-09-26 12:26:37	1	exe exe
E72C360FFCE4B8998EC1A350A6C57156A88897EB51960C96905F8866E5A05E13 e72c36dffce4bb998ec1a35da6c57156a08897eb5196dc969d5f80666e5ad5e13.bin © peexe ©	28 / 71 123.50 KB	2020-09-29 05:43:35	2020-09-29 05:43:35	1	exe exe
CFE1678A7F2B9499660D9A820FAAFB46662584F8A6AC4B72583A21FA858F2A2E8 O0228416.exe	29 / 70 123.50 KB	2020-09-30 02:58:04	2020-09-30 02:58:04	1	EXE
E8A0E80DFC520BF7E76C33A90ED6D286E8729E9DEFE6B87DA2F388C2DB33F399 O0196927.exe	26 / 67 123.50 KB	2020-09-30 04:38:00	2020-09-30 04:38:00	1	EXE
CFDC2C847EF302396387C487FC3C9FE558380282E5788EE9AEA4A81E4E02EC28 e8a0e80dfc520bf7e76c33a90ed6d286e8729e9defe6bb7da2f38bc2db33f399.bin_unmapped e peex overlay	10 / 70 118.00 KB	2020-09-30 13:42:21	2020-09-30 13:42:21	1	EXE
96E2D082F05EF688682E31DC33D75530F3E4E3932F0A1E619D00D8BA7990E088 v2.exe	31 / 70 124.50 KB	2020-10-01 17:40:01	2020-10-01 17:40:01	1	EXE
5212B76A3E238759AFFDAD53865553E9D80F6DAFDAD27A57958E6450987371F2 v2.exe	45 / 70 124.50 KB	2020-10-01 17:55:20	2020-10-01 17:55:20	1	exe

The resulting listing with all the <u>files found</u> 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 <u>one</u> of our initial samples and <u>another one</u> with behavioural information, we can see that:

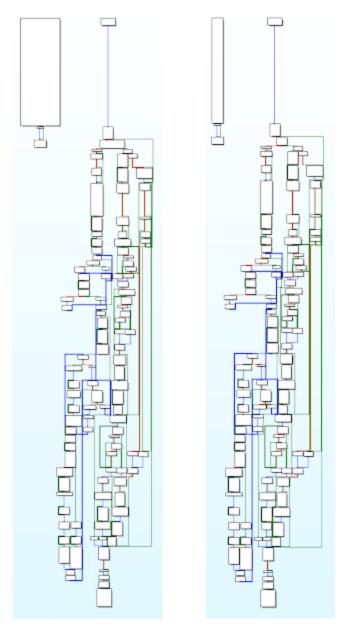
- They don't show up as similar when doing a similarity search (as expected).
- They have some long sequences of bytes in common.

	DIFF	PATT	ERNS	35																?	$\equiv \diamond$	•		; <u>-</u>
																	Ma	tched bytes	Matches		Tags			
35	9C	64	01	35	A4	64	01	35	в0	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	FO	64	01	$5 \cdot d \cdot 5 \cdot d \cdot 5 \cdot d \cdot 5 \cdot d \cdot$								
35	F8	64	01	35	00	65	01	35	0 C	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 · 5 4 e ·								
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	в0	65	01	35	В4	65	01	35	В8	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 •		394	2/2		binary		Q	ĥ
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	FO	65	01	35	F8	65	01	35	04	66	01	35	1C	66	01	5 · e · 5 · e · 5 · f · 5 · f ·								

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 <u>0x35008A60</u>, 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 <u>query generated</u>: 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.

😨 .data:3501ECF4	000002B	С	Picuovphv Bbsg:Es rwojrarkkd Stryjfes x4.3		
🔄 .data: 3501ED20	000000D	С	Ddvdpl440dlm	Copy #C Copy all 수상	
🛐 .data: 3501ED30	00000015	с	FrystDdswirfCqovf{vD		
😴 .data: 3501ED48	00000010	С	FrystFsgctelaui	〒 Quick filter ヘF Modify filters ヘ☆F	
😴 .data: 3501ED58	000000E	С	FrystKbujDaua	🍟 Modify filters ^ ጉ ፍ	
🔄 .data: 3501ED68	00000011	С	FrystGfuvrozHctj	Setup	
🔄 .data: 3501ED7C	00000014	С	FrystUfngasfCqovf{v	Hide column	
🔄 .data: 3501ED90	00000012	С	FrystJfvJasiPcscn	Columns	
🔄 .data:3501EDA9	00000161	С	YfJPuyqKmWxOwvkiLnkhWhVxmEhdcMpbJCvY	VirusTotal >	Search for string
🔄 .data:3501EF20	0000007	С	\$\t.Fs\aV	Font	
😒 .data:3501EF48	000000C	С	VXd[ra;78]In		

Using the VT-IDA plugin we can search for <u>other files</u> 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 <u>following VT-Graph</u>.

This post was co-authored by Vicente Diaz.