[Case Study: Latrodectus] Analyzing and Implementing String Decryption Algorithms

0x0d4y.blog/case-study-analyzing-and-implementing-string-decryption-algorithms-latrodectus/

May 9, 2024



This article has a slightly different objective than the last ones I published, it is not about an analysis of specific malware.

Today's article is about a case study of the <u>Latrodectus</u> string decryption algorithm (analyzed in the **previous research**). The objective is to study how to identify a string decryption algorithm when reverse engineering a malware, and how we can implement it in *Python* to statically decrypt them. Specifically, we will cover how to identify whether malware is using a custom decryption algorithm to obfuscate strings.

So, let's go!

Why Do Adversaries Encrypt Strings?

It may seem obvious, but it's good to clarify why adversaries encrypt strings to use in their Malware.

The short and thick answer is, to achieve the technical objective of **Obfuscation**! The implementation of encryption to obfuscate strings allows adversaries to hide strings that reveal the objective of the actions that the malware will perform. This technique is registered with MITRE ATT&CK as <u>Obfuscated Files or Information: Encrypted/Encoded File</u> [T1027.013].

Let's use the example of <u>*WannaCry*</u>, which does not implement the string encryption technique. Below, we can see the simple use of the *strings* command, present in every *Linux* system.



As you can see in the image above, without implementing this obfuscation technique, the malware is more vulnerable to detections by security products, and our analysis is simpler :,)

Now let's do the same test on a *Latrodectus* sample.

📶 🖢 ~/Á/F/M/L/String Decry	ptor strings	d1e2e287c96c290e161c553d99	a115e7d72f83f23c850621
169a27cca936f51b.bin grep '	http'		
📶 🖕 ~/Á/F/M/L/String Decry	ptor strings	d1e2e287c96c290e161c553d99	a115e7d72f83f23c850621
<u>169a27cca936f51b.bin grep '</u>	.exe'		
📶 🖢 ~/Á/F/M/L/String Decry	ptor strings	d1e2e287c96c290e161c553d99	a115e7d72f83f23c850621
<u> 169a27cca936f51b.bin tail -</u>	n 35		
D\$0H			
Т\$8Н			
D\$(H			
D\$ H			
T\$OH			
Mb=Lk			
.text\$mn			
.idata\$5			
.rdata			
.rdata\$zzzdbg			
.xdata			
.edata			
.idata\$2			
.idata\$3			
.idata\$4			
.idata\$6			
.data			
.bss			
.pdata			
UpdaterTag.dll			
extra			
follower			
scub			
CreateMutexW			
PeekNamedPipe			
GetLastError			
KERNEL32.dll			
MessageBoxA			
MessageBeep			
USER32.dll			
&a(F*Y.@.N0E2 4v6^8]:0<=>			
l g"0\$J&C(*H.Y.			
0v2V4A6			
8}:R <n>0@-B"D<f h(j&l(nop-rst<="" td=""><td>3V>X7Z?\.^+`</td><td></td><td></td></f></n>	3V>X7Z?\.^+`		
bCdJf1hIiDl/n0p^r0t0v			
I ► ~/Á/F/M/L/String Decry	ptor		

And as we can see above, it was not possible to detect with the strings command any inconsistencies in *URLs* or *binaries*, as we analyzed *Latrodectus* previously, we know that it communicates with two *C2* servers, so this means one thing, *Latrodectus* implements a encryption to obfuscate strings, and decrypts at runtime!

Now that we understand why adversaries implement this obfuscation technique, let's understand how we can identify the use of a decryption algorithm in malware.

How to Identify the Implementation of a Decryption Algorithm?

Firstly, there is no single correct answer to this question, secondly, here comes the famous 'it depends'!

Adversaries will always seek to implement custom encryption algorithms to obfuscate strings and some payloads. This is because using known algorithms, whether through Windows APIs or by manually implementing a known algorithm, can reduce the adversary's chances of passing through *detections* and slowing down our rate of analysis of their sample. This is because the algorithm is already known to us, therefore, it is easier to identify constants or the flow of a given algorithm.

But when the adversary goes down the path of implementing its own algorithm, it runs the risk of implementing something simpler than what already exists. This is because the adversary certainly does not want to disrupt the functioning of the malware. Another risk that the adversary may run is that, once identified by a researcher, *detections* to monitor the presence of a particular malware family will come into existence, in addition to automated extractors using scripts. Therefore, these custom algorithms are short-lived.

But how can we identify that a malware has a string decryption routine? Let's go.

As I said earlier, there is no single method of identification. So here are some tips, and then we will analyze how the behavior is observed in *Latrodectus*.

- A single function is called dozens or hundreds of times during code execution (the amount will depend on the malware);
- The function will probably be receiving as an argument an offset that points to a block of data, probably encrypted.
- When looking at the function execution flow graph, you will probably find one or more loops that perform operations on the data. The likely operation you will encounter will be the **XOR** operation, which will have the *encrypted data* and the *XOR key* as protagonists of this operation.
- Another tip I can give is to identify in your favorite *Disassembler* if there are several blocks of data that are called by the same function during code execution.

Below we can see that in Latrodecuts it is possible to identify some of these patterns that I mentioned above.



Function **sub_18000ae78** is called **121 times** during *Latrodectus* execution. Another pattern that we can detect is that this function receives a set of data as an argument, and stores the result of the function's manipulation in a buffer.

Below we can observe the encrypted data block that is passed as an argument (**data_18000fa00**), in addition to being able to observe the other encrypted data blocks.

PE▼ Linear▼ Pseudo C▼

0x18000f9b0 .data {0x18000f000-0x180011320} Writable data

18000f9b0 7e d0 dd ad 7c d0 7f 80 18000f9b8 data_18000f9b8: 18000f9b8 7e d0 dd ad 64 d0 0d 80 ~...d... 18000f9c0 f4 82 ed 84 e1 86 eb 88-e5 8a b8 8c bf 8e a1 90 18000f9d0 f4 92 eb 94 f0 96 97 98 18000f9d8 data_18000f9d8: 7e d0 dd ad 66 d0 5d 80 ~...f.]. 18000f9d8 18000f9e0 a4 82 f0 84 a7 86 ab 88-a9 8a ae 8c fe 8e af 90 18000f9f0 b4 92 e0 94 95 96 00 00 18000f9f8 data_18000f9f8: 18000f9f8 20 00 00 00 00 00 00 00 18000fa00 data_18000fa00: 18000fa00 7e d0 dd ad 6e d0 0d 80-f4 82 ed 84 eb 86 f2 88 ~...n..... 18000fa10 e7 8a ec 8c 8d 8e 00 00 18000fa18 data_18000fa18: 7e d0 dd ad 6e d0 45 80 18000fa18 ~...n.E. 18000fa20 f6 82 f7 84 e3 86 e5 88-eb 8a fa 8c 8d 8e 00 00 18000fa30 data_18000fa30: 18000fa30 2c 00 00 00 00 00 00 00 18000fa38 data_18000fa38: 18000fa38 7e d0 dd ad 78 d0 5a 80 ~...x.Z. 18000fa40 e5 82 83 84 00 00 00 00 18000fa48 data_18000fa48: 18000fa48 7e d0 dd ad 74 d0 5a 80 ~...t.Z. 18000fa50 f2 82 a6 84 f6 86 87 88 18000fa58 data_18000fa58: 7e d0 dd ad 73 d0 19 e9 18000fa58 ~...s... 18000fa60 ed e7 f0 ab e7 f6 a9 ec-e8 fe 8b 00 00 00 00 00 18000fa70 data_18000fa70: 18000fa70 7e d0 dd ad 6a d0 5a 80-f2 82 df 84 a0 86 e3 88 ~...j.Z..... 18000fa80 a7 8a ef 8c e1 8e e3 90-91 92 00 00 00 00 00 00 18000fa90 data_18000fa90: 18000fa90 7e d0 dd ad 70 d0 5a 80-e5 82 ad 84 e1 86 e6 88 ~...p.Z..... 18000faa0 fd 8a 8b 8c 00 00 00 00 18000faa8 data_18000faa8: 7e d0 dd ad 72 d0 5a 80 ~...r.Z. 18000faa8 18000fab0 f2 82 df 84 a0 86 f4 88-89 8a 00 00 00 00 00 00 18000fac0 data_18000fac0: 18000fac0 7e d0 dd ad 58 d0 16 80-ef 82 ea 84 f1 86 a7 88 ~...X...... 18000fad0 a4 8a f1 8c f7 8e f5 90-eb 92 ae 94 b7 96 b2 98 18000fae0 ea 9a c7 9c b8 9e ec a0-83 a2 a3 a4 00 00 00 00 18000faf0 data_18000faf0: 18000faf0 7e d0 dd ad 78 d0 19 f2-ee ec f7 84 00 00 00 00 ~...x..... 18000fb00 data_18000fb00: 18000fb00 7e d0 dd ad 76 d0 50 e6-e8 ee e6 f7 aa 86 00 00 ~...v.P..... 18000fb10 data_18000fb10:

If we look at the execution flow in graphical mode, we will also detect another pattern, a loop that manipulates data through **XOR** operations.



Now that we have been able to identify the string decryption function, let's analyze how it works statically and dynamically.

Analyzing Latrodectus Decryption Algorithm

Here, we begin our hands-on adventure. First, when we are going to do our dynamic analysis as a complement to the static one, we need to locate the exact decryption function in the debugger, so as not to get lost. In the debugger we do not have the Decompiler crutch, so it is important that during dynamic analysis using a debugger, you have the disassembler/decompiler open.

In the decompiler, we can see below the exact moment when our decryption function is called for the first time in the code.

sub_180003868()						
			18000391a e8c92f0000 18000391f 85c0 180003921 750a	call sub_1 test eax, jne 0x180	800068e8 eax 00392d	
	18000392d 4880d542440 180003932 488d0dc7c00000 180003939 e83a750000 180003930 4885c0 180003941 740c	lea ro lea ro call xo test ra je 0:	 dx, [rsp+0x40 {decrypted_strin cx, [rel encrypted_data_block] pr_decrypt ux, rax <18000394f	ng_out_buffer}]]		
-					, ,	
d442440 lea rax, [rsp+0x40 (decryp 9442430 mov qword [rsp+0x30 {var_5	oted_string_out_buffer}] 8_1}], rax {decrypted_string_	out_buffer}	180003943 488d442440 180003948 4889442430 18000394d eb0a	lea rax mov qwo jmp 0x1	, [rsp+0x40 {decrypted_; rd [rsp+0x30 {var_58_1} 80003959	string_out_buffer], rax {decrypted
			,			

And below, we can observe the same moment. As our notes will not be present in the debugger, it is recommended that you set seven breakpoints and write comments, to remember where each action is done.



To validate where we are, below we can see the content of the encrypted data block that the xor_decrypt function (renamed by me, for documentation purposes) receives as an argument.

			Ţ			
	188 189 180 180 180	0003920 4884542440 003932 48840647500000 003939 e83a750000 003939 4885c0 0003941 740c	lea rdx, lea rcx, call xor test ra, je 0/18	[rsp+0x40 {decr [rel encrypted_ decrypt rax 000394f	ypted_string_out_buffer} data_block]]
8f 488d442440 lea 54 4889442430 mov	rax, [rsp+0x40 {decrypted_ qword [rsp+0x30 {var_58_1}	string_out_buffer}] }, rax {decrypted_string	g_out_buffer}	180003943 488d 180003948 4889 18000394d eb0a	442440 lea r 442430 mov q jmp 0	ax, [rsp+ word [rsp x18000395
Linear 🔻 Disassembly 🔻				******		
9x18000fa00 .data {0x18000f000-	•0x180011320} Writable data					
18000fa00 encrypted data block	k •					
18000fa00 7e d0 dd ad 6e d0 0d	d 80-f4 82 ed 84 eb 86 f2 a	88-e7 8a ec 8c 8d 8e 00	00	~	.n	
18000fa18 data_18000fa18: 18000fa18 18000fa20 f6 82 f7 84 e3 86 e5	5 88-eb 8a fa 8c 8d 8e 00	00	7e d0 dd ad (5e d0 45 80		.n.E.
18000fa30 data_18000fa30: 18000fa30 18000fa38 data_18000fa38:		2c 00 00 00 00 00 00	00			
18000fa38 18000fa40 e5 82 83 84 00 00 00 18000fa48 data_18000fa48:	8 88		7e d0 dd ad 1	78 d0 5a 80 		
18000fa48	7e d0 dd ad 74 d0 5a	80-f2 82 a6 84 f6 86 87	88			
18000fa58 data_18000fa58: 18000fa58			7e d0 dd ad 3	73 d0 19 e9	~	.s

If we do the same thing with address 7FFF11D2FA00, we will observe the same data.



When we enter the function (Step-In in the debugger), we can also validate that the execution flow through graphical mode is the same. You can observe the comparison in the sequence of images, and see that we are in fact in the correct function.





We can also use Decompiler to give us a hand in analyzing this algorithm. In our pseudocode, we can see that first there is an *XOR* operation between some bytes within the data block itself. Then, **rcx_1** is used as a conditional for the *while loop* to continue executing, as long as **var_14** (set to **0**) is less than **rcx_1**. This is where we can assume from experience that right now the algorithm is calculating the value of the block of data that will be decrypted. After all, the block needs to have an end.

18000ae78	int64_t xor_encrypt(int32_t* arg1, int64_t arg2)
180002094	$int32 \pm var c = \star arc1$
190000005	$int16 \pm rev = 1 = var = w \wedge arg1[1] w$
10000000000	$intio_t = var_t = var_t = 0$
Teeesee	Inclo_t var_14 = 0
18000aede	while (zx.d(var_14) s< zx.d(rcx_1))
18000aeee	uint64_t rax_9
18000aeee	rax_9.b = *(arg1 + 6 + zx.q(var_14))
18000aef1	char var_18_1 = rax_9.b
18000aeff	uint64_t rax_10
18000aeff	rax_10.b = *(arg1 + 6 + zx.q(var_14))
18000af14	char var_17_2 = rax_10.b + var_18_1 + 0xa
18000af21	var_c = sub_180009040(var_c)
18000af46	*(arg2 + zx.q(var_14)) = *(arg2 + zx.q(var_14)) + var_18_1 + 0xa
18000af5f	*(arg2 + zx.q(var_14)) = var_18_1 ^ var_c.b
18000aecd	var_14 = var_14 + 1
18000af70	return arg2

To validate, we can check in the debugger. Below, we can see the suspicions of what we saw in the pseudo-code above. The algorithm selected two bytes present in the data block, **0x7e** and **0x6e**, and performed an **XOR** operation between these two values.

兴 rur	ndll32	exe - PID: 35	516 - Mo	dule: emb	edded_latr	odectus.dl	I - Thread: N	Main Thre	ad 1920 -	x64dbg	[Elevated]						
File	View	Debug T	racing	Plugins f	avourites	Options	Help Mar	[.] 8 2024 (T									
•	0	■ →	•	† o	🧆 🦂	¥ ‡	→ 1 8	🖊	= 🧳	• 🥒	fx #	A2	. I	9			
<u>60</u>	CPU	🍃 Log	•	Notes	• Brea	kpoints	📟 Memo	ory Map	🗐 c	all Stack	· 2 3	SEH	o Script	1	Symbols	Source	e 🔎 References
RIP		00007F 00007F 00007F 00007F 00007F 00007F 00007F 00007F 00007F 00007F	FF11D1 FF11D1 FF11D1 FF11D1 FF11D1 FF11D1 FF11D1 FF11D1 FF11D1 FF11D1	AE92 AE94 AE98 AE9D AEA1 AEA5 AEA7 AEA7 AEA7 AEA8 AEB7 AEB7 AEB7	8B00 8944 48:8 0FB7- 8B4C 33C8 8B4C 33C8 8B5C1 66:8 48:8 48:8 48:8 48:8 48:8	24 2C 34424 4 40 04 24 2C 94424 2 34424 4 3C0 06 94424 4	0 8 0 0	mov mov mov mov xor mov mov add mov	eax,du dword rax,qu zx eax ecx,du ecx,ea eax,ea word rax,qu rax,6 qword	word p ptr s word p word p ax cx ptr ss word p ptr s	tr ds: s:[rsp tr ss: ptr ds tr ss: :[rsp+ tr ss: s:[rsp	[rax] +2C],e [rsp+4 :[rax+ [rsp+2 28],ax [rsp+4 +40],r	ax 01 4c] 0] ax		(r. (r. (r. 1s. (r. F. (r. F. (r. F. (r. F. (r. F. (r.) (r.)	AX 0000 BX 0000 DX 0000 SP 0000 SP 0000 SI 0000 DI 0000	000000000006E 000000000000 0000ADDDD07E 0087483EFA0 0087483EFA0 0087483EFA0 000001C022A 0000000000000

The value of this **XOR** operation was **0x10**, as we can see in the **RCX** register.

🐺 rundll32.exe - PID: 3516 - Module: embedded_latrodectus.dll - Thread: Main Thread 1920 - x64dbg [Elevated]	
File View Debug Tracing Plugins Favourites Options Help Mar 8 2024 (TitanEngine)	
🕮 CPU 🔋 Log 📫 Notes 🔹 Breakpoints 🚥 Memory Map 🗐 Call Stack 🗣 SEH 👩 Script 🔮 Symbols 🔇	🔾 Source 🏸 References
00007FFf11D1AE92 8800 mov eax,dword ptr ds:[rax] 00007FFf11D1AE94 894424 2C mov dword ptr ss:[rsp+2C].eax	
00007FFF11D1AE98 48:884424 mov rax,qword ptr ss:[rsp+40] [r RAX 000007FFF11D1AE9D 0FB740 04 movzx eax,word ptr ds:[rax+4] [RX	0000000000000006E 000000000000000000
O0007FFF11D1AEA1 8B4C24 2C mov ecx,dword ptr ss:[rsp+2C] O0007FFF11D1AEA5 33C8 xor ecx.eax	00000000ADDD0010
RIP 00007FFF11D1AEA7 8BC1 mov eax,ecx	00000087483EFAE0
00000/FFF11D1AEA9 06:894424 28 mov word ptr ss:[rsp+28],ax RBP 00007FFF11D1AEAE 48:884424 40 mov rax,qword ptr ss:[rsp+40] [r. RSP 00007FFF11D1AEAE 48:880424 40 add rax,6 1s RSP 00007FFF11D1AEB3 48:880424 40 mov qword ptr ss:[rsp+40],rax [r. RDI	00000087483EFCA0 00000087483EFA60 00000000001C022A 0000000000000000000

If we check in our Disassembler, byte **0x7e** is the first byte of the every data block, and **0x6e** is the fifth byte of this specific data block. In the image below, we can also redo the operation through the *Binary Ninja Python console*, where the value will also give **0x10**, which in decimal is **16**. And if we further analyze the block of data in question, we will also be able to

observe that **0x10** is the *exact size of this specific data block*, before null values. In other words, in fact, the algorithm sets the size of the current data block that will be decrypted, and uses the value of its size as a conditional for the while loop.



As we proceed, the decryption algorithm calls a function that we can also observe in the pseudo-code. This function simply adds *1 byte* (going from **0x7e** to **0x7f**) to the first byte of the encrypted data block.

🗄 rundll32.exe - PID:	: 7980 - Mc	idule: emb	edded_latrodectus	dll - Thread: Main Thr	read 5316 - x64dbg	[Elevated]										
File View Debug	Tracing	Plugins F	Favourites Option	s Help Mar 8 2024 (
📫 🖸 🔳 🚽	• II		🛥 🌲 1	-1 🔟 🖊	🚍 🛷 🥔		ս 👢 🛙 🖩	9								
🖾 CPU 🌔 La	og 📫	Notes	Breakpoints	🚥 Memory Map	Call Stack	9 <u>8</u> SEH	Script	Symbols		References	🗢 Threads	🔒 Handles				
		0000	7FFF11D2AF18 7EFF11D2AF1C	884C24 2C		nov ecx.dwc	rd ptr ss: ed latrode	frsp+2Cl Ectus, 7FFF110	29040 XOR	Key + 1						Show FPU
RIP		00000 00000 00000 00000 00000 00000 0000	7FFF1102AF21 7FFF1102AF25 7FFF1102AF26 7FFF1102AF26 7FFF1102AF34 7FFF1102AF38 7FFF1102AF38 7FFF1102AF41	894424 2C OFB74424 2 OFB64C24 2 48:885424 OFB60402 8D4408 0A OFB74C24 2 48:885424	4 0 48 14 48	nov dword p novzx eax,w novzx ecx,b novzx ecx,b novzx eax,dw novzx eax,dw novzx ecx,w novzx ecx,w	tr ss:[rsp ord ptr ss yte ptr ss nd ptr ss yte ptr ds ord ptr ds ord ptr ss ord ptr ss	++2C],eax s:[rsp+24] s:[rsp+20] s:[rsp+48] s:[rdx+rax] s:[rsp+24] s:[rsp+24] s:[rsp+48]						RAX RBX RCX RDX RBP RSP RSI	00000000AD00D07F 00000000000000000 00000000AD0D07E 0000007A9628F140 0000007A9628F600 0000007A9628F080 0000000001E0236	

Next, the algorithm will perform an **XOR** operation with the byte appended to *1* (**0x7f**) and with byte **0x0d**, which is the *seventh byte of the encrypted data block*.

86	rundii	62.exe	e - PID:	1976 - N	lodule:	embedde	d_latrode	ctus.dll -	Thread:	Main Th	ead 357	2 - x64db	g (Elevat	ted]												
Fik	We		ebug	Tracing	Plugins		rites Op	tions H	elp Ma																	
-	0	-	14				• ¥.		4 18	/	=	4 🥜			A+	L II										
E	OR		De La	9	Note		Breakpo							SEH			2		References							
			0000	7FFF11 7FFF11	02AF41 02AF46	48	1:885424 040A			iov rdx. iov byti	, qword	ptr ss: ls:[rdx:	[rsp+	48] al												Show FPU
			0000	7FFF11 7FFF11	D2AF49 D2AF4E	0F 0F	864424 864C24	20 2C		ovzx el ovzx el	sx,byte □x,byte	e ptr se ptr se	i:[rsp i:[rsp	+20] +2C]											RAX	000000000000000000000000000000000000000
RIF			000000	788811	02 AF 5 3													: XOR Key			e Encrypt			on)	KBA DC V	00000000000000075
			0000		DZAFSS	OF	B74C24	24		iovzx_ei	CK, WORK	i ptr si		+24											PDY	0000005956285600
			0000	7FFF11	DZAFSA	48	:885424	48		iov rdx	,quord	ptr ss:	[rsp+	48]											REP	0000005956286890
			0000	7FFF11	DZAFSF	88	040A			IOV DYT	e ptr e	IS I LINDKA	HCX]	al CCC110											RSP	000000595628£650
Γ.			0000	755511	024567		-084474	40		empi		acroae	10000	401											RST	000000000270236
			0000		024660	20	1004424	*		dd ese	18														RDT	000000000000000000

We can validate this information in our Disassembler, where it is possible to observe that the algorithm skips the initial *6 bytes* of the encrypted data block.

0x18000fa10	.data	a {0x18	3000f00	0-0x1	8001132	20} 1	Writab	le da	ta	
18000fa00 18000fa00	encry 7e d0	pted_d dd ad	ata_blo 6e d0	ock: 0d 8	0-f4 82	ed	84 eb	86 f2	288	~n.
18000fa10	e7 8a	ec 8c	8d 8e	00 0	9					

When we perform the **XOR** operation between the values 0x7f and 0x0d, the result (stored in **RAX**) will be a string identified as '*r*'.

₩ ru	rundli32.exe - PID; 1976 - Module: embedded_latrodectus.dll - Thread: Main Thread 3572 - x64dbg [Elevated]																	
File	View	Debug Tradi	ng Plugins	Favourites Options	Help													
•	O	■ → II	1 🕈 🤅	▶ ‡ ‡	≁±	S /	= 🥢 🕯	₽ fx		Aa	8. 1	9						
60	CPU	🍃 Log	📫 Notes	 Breakpoints 		Memory Map	🧊 Call Sta	dk	🧏 SEH	O	Script	Symbols	Source	References	٠	Threads	ᡖ Handles	ぞ ^う Trace
	Î l	00007FFF11	D2AF41	48:885424 48 880404		mov rdx,qw	ord ptr ss tr ds:[rdx-	[rsp	48] al									
		00007FFF11	D2AF49	0FB64424 20		movzx eax,	byte ptr s	:[rsp	+20]						RAX	000000	0000000072	ini.
		00007FFF11	D2AF4E	0FB64C24 2C		movzx ecx,	byte ptr s:	::[rsp	+2C]		0	+	VOD K (1-+		RBX	000000	00000000000	
RTP			DZAF55	0FB74C24 24	_	movzx ecx.	x word ptr s				Decryp	ted 1st byte o	t the Encryp	ted String	RCX	000000	000000007F	
		00007FFF11	UZAFSA	48:855424 48	_	mov rax,qw	ora ptr ss	Inspe	481		occi jp		. the the sys			000000)595628F6D0	
		00007FFF11	D2AF5F	88040A		mov byte p	tr ds:[rdx	rcx],	al						RBP	000000	0595628F890	

Having analyzed this behavior, we reached the following conclusion:

- The first byte of all blocks to be decrypted is the initial byte, which will always have its value increased by 1 for each subsequent byte (starting from the seventh byte of the encrypted data block) in which the XOR operation will be performed. That is, each byte will have a different XOR key.
- The *fifth byte* of each encrypted data block will be used together with the *first byte* of the block to calculate the size of the block that must be decrypted
- In other words, the *first six bytes* of each encrypted data block are not decrypted, they are what we can call the *control header*.

Below, we can validate our assumption. Below, I manually made the algorithm execution flow.

$\overline{\Sigma}$ T ₁ researcher : python3 – Konsole $\vee \land \otimes$	18000fa00	data_18000fa00:	
	18000fa00	7e d0 dd ad 6e d0 0d 80-f4 82 ed 84 eb 86 f2 88	~n.
🗅 Nova aba 🔲 Dividir a ovibicão 🔀	18000fa10	e7 8a ec 8c 8d 8e 00 00	
	18000fa18	data_18000fa18:	
>>> hex(0x7e+1)	18000fa18	7e d0 dd ad 6e d0 45 80	~n.E.
'0x7f'	18000fa20	f6 82 f7 84 e3 86 e5 88-eb 8a fa 8c 8d 8e 00 00	
\rightarrow hex(0x7f ^ 0x0d)	18000fa30	data_18000fa30:	
'0x72'	18000fa30	2c 00 00 00 00 00 00 00	
$\rightarrow \rightarrow her(0x7f+1)$	18000fa38	data_18000fa38:	
	18000fa38	7e d0 dd ad 78 d0 5a 80	~x.Z.
$\rightarrow \rightarrow box(0x80 \land 0x80)$	18000fa40	e5 82 83 84 00 00 00 00	
	18000fa48	data_18000fa48:	
$\sum hox(0x80+1)$	18000fa48	7e d0 dd ad 74 d0 5a 80	~t.Z.
>>> THEX(0X00+1)	18000fa50	f2 82 a6 84 f6 86 87 88	
0x01	18000fa58	data_18000fa58:	
>>> nex(0xo1 0x14)	18000fa58	7e d0 dd ad 73 d0 19 e9	
	18000fa60	ed e7 f0 ab e7 f6 a9 ec-e8 fe 8b 00 00 00 00 00	
>>> hex(0x81+1)	18000fa70	data_18000fa70:	
·0x82·	18000fa70	7e d0 dd ad 6a d0 5a 80-†2 82 d† 84 a0 86 e3 88	~j.Z
>>> hex(0x82 ^ 0x82)	18000fa80	a7 8a ef 8c e1 8e e3 90-91 92 00 00 00 00 00 00	
'0×0'	18000fa90	data_18000fa90:	
>>> hex(0x82+1)	18000fa90	7e d0 dd ad 70 d0 5a 80-e5 82 ad 84 e1 86 e6 88	~p.Z
'0x83'	180001aa0	td 8a 8b 8c 00 00 00 00	
>>> hex(0x83 ^ 0xed)	180001aa8	data_18000Taa8:	
'0x6e'	180001aa8		~r.Z.
>>> hex(0x83+1)	180001200	T2 82 0T 84 80 86 T4 88-89 88 00 00 00 00 00 00	
'0x84'	180001ac0		
>>> hex(0x84 ^ 0x84)	100005-10	7e d0 dd ad 58 d0 16 80-et 82 ea 84 ti 86 a7 88	~
'0×0'	100005-00	a4 8a 11 8c 17 8e 15 90-eb 92 ae 94 b7 96 b2 98	
>>> hex(0x84+1)	100005-50	ea 9a c7 9c b8 9e ec a0-83 a2 a3 a4 00 00 00 00	
'0x85'	100001010	data_100001a10;	
>>> hex(0x85 ^ 0xeb)	100001410	dete 18000fb00;	~
'0x6e'	180001000	7e d0 dd ad 76 d0 50 e6-e8 ee e6 f7 aa 86 00 00	N V P
>>> # and so on	180001000	data 18000fb10	·····
>>> []	19999fb19	7_{0} d0 dd ad 77 d0 22 e0-f5 f6 ef e1 ed f1 97 00	~ ~ ~ ?
	186661010		·····

Upon obtaining a certain set of bytes, I went to **CyberChef** to transform the hex data into readable output, and... *Voilà*!



As we know from the *Latrodectus* analysis in my previous post, the string above is part (I just streamed it in a few bytes, out of laziness) of the **runnung** string, which is used to create the *Mutex* on the infected system.

Latrodectus Decryption Algorithm Flowchart

In order to improve understanding of the algorithm, below is a flowchart I made just to illustrate the flow of executing the *Latrodectus* string decryption algorithm.



Once you understand the algorithm, you can implement this algorithm in a script, with the aim of extracting the strings from the sample you are analyzing.

Python Script for String Decryption and Extraction

I created a Python script that will run the *Latrodectus* decryption algorithm, print the entire flow of its execution for debugging and study.

Below is the video of the execution of the script.

Python-Only – Latrodectus String Extractor

The source code of the script can be found on my *github*, at the link below:

Conclusion

Well, I hope this type of article pleased you, the reader, and that you learned something new!! See you around!