Biuletyn WAT
Vol. LXV, Nr 4, 2016

CryptXXX V3 –Analiza użytego mechanizmu szyfrowania

MICHAŁ GLET

Wojskowa Akademia Techniczna, Wydział Cybernetyki, Instytut Matematyki i Kryptologii,
00-908 Warszawa, ul. gen. S. Kaliskiego 2, michal.glet@wat.edu.pl

Streszczenie. Celem artykułu jest analiza sposobu działania złośliwego oprogramowania typu ransomware w kontekście wykorzystywanych mechanizmów kryptologicznych. Inżynierii wstecznej poddana zostaje najnowsza w danym momencie wersja oprogramowania CryptXXX. Odkryte zostają wykorzystywane mechanizmy kryptograficzne, wskazane zostają ich słabości oraz możliwości poprawy.
Słowa kluczowe: ransonware, inżynieria oprogramowania, inżynieria wsteczna, RC4, RSA, złośliwe oprogramowanie
DOI: 10.5604/12345865.1228958

1. Opis problemu

Jest piątek, 20.05.2016 r., parę minut po godzinie 10. W biurze panuje względny spokój. Nagle do open space'u wchodzi znajomy, szybkim i nerwowym krokiem zmierza w kierunku kolegi. W ręku trzyma pendrive'a. Z rozmowy wynika, że potrzebuje aktualnych plików projektu, bo jego zostały zaszyfrowane. Pliki te mają pomóc w odszyfrowaniu zawartości. Znowu zapada spokój. Po chwili okazuje się, że skopiowane pliki nic nie dały — deszyfrator nie odnalazł klucza. Zaczyna się delikatna panika, wszyscy dostają maila z zaleceniem, aby pobrać oprogramowanie Symanteca i przeskanować natychmiast komputery. Zalecenie mówi również o utworzeniu backupu. Mijają kolejne godziny. Ponownie pojawia się znajomy z pytaniem, czy ktoś ma bitcoiny. Potrzeba 1,2 bitcoina na „okup". Po ich przelaniu na wskazany adres twórcy wirusa mają udostępnić klucz szyfrujący oraz oprogramowanie odszyfrowujące. Trwa poszukiwanie kryptowaluty. Około godziny 16 kryptowalutowy

okup zostaje przelany na wskazany adres. Teraz wystarczy poczekać 30-40 minut na kilka potwierdzeń transakcji i po problemie — będzie klucz i oprogramowanie. Sytuacja wydaje się być opanowana, choć pewien niesmak pozostaje — w końcu kryptowaluta przekazana została cyberprzestępcom. Kończy się dzień pracy, zaczyna się weekend. Wszystko jak co tydzień, z jedną drobną różnicą — około godziny 19 dostaję SMS od znajomego o treści „Decrypt error – 4. :(". Czyli mamy problem — wiele dokumentów i projektów być może zostało właśnie bezpowrotnie straconych, gdyż nie było ich backupów.

Tak właśnie rozpoczęła się moja przygoda z oprogramowaniem CryptXXX w wersji 3. Chciałem pomóc koledze i przy okazji zobaczyć, jak tego typu oprogramowanie jest zbudowane, jak działa oraz jakie algorytmy kryptograficzne zostały w nim użyte. Ciekawił mnie też sposób zastosowania algorytmów kryptograficznych. Powszechnie wiadomo, że nieprawidłowe zastosowanie nawet najbezpieczniejszego algorytmu może być przyczyną kompromitacji systemu. I tak zapewne złamane zostały poprzednie wersje CryptXXX, dla których istnieją bezpłatne deszyfratory, udostępniane przez twórców oprogramowania antywirusowego.

Poprosiłem zatem o podesłanie kilku par plików (plik zaszyfrowany, plik oryginalny), otrzymanego klucza oraz programu deszyfrującego.

2. Analiza

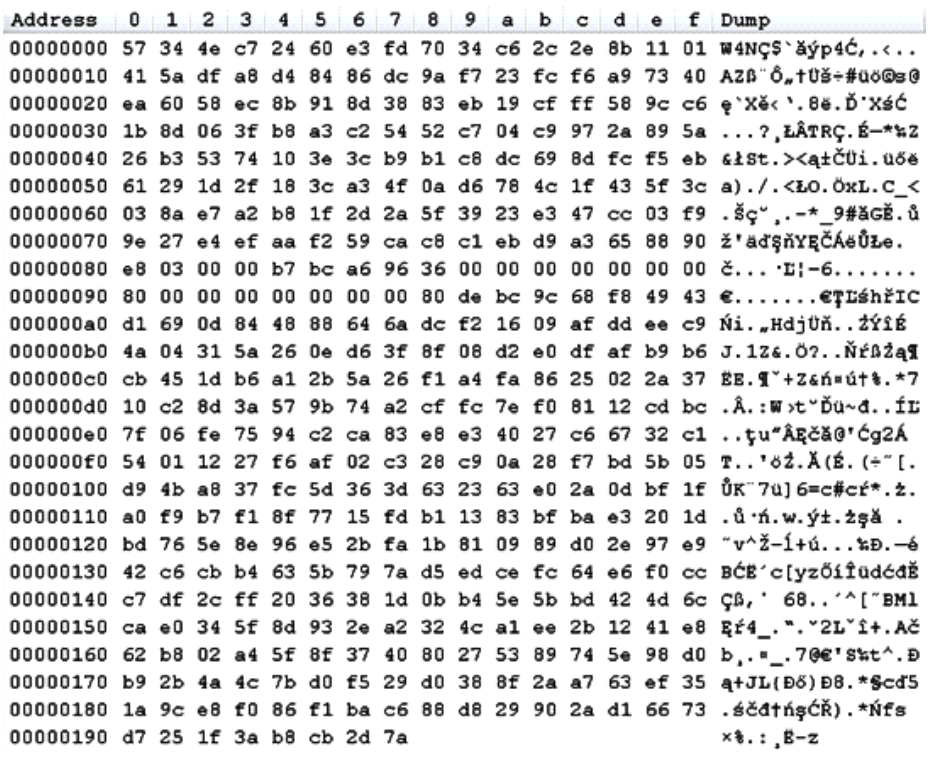
Analizę rozpocząłem 20.05.2016 około godziny 21. Z uwagi, że korzystam z systemu operacyjnego OS X, całą analizę przeprowadziłem na systemie operacyjnym Windows 7 pracującym w środowisku wirtualnym VirtualBox. Numerację będę zaczynał od 0. Korzystał będę z notacji hexadecymalnej. Dzięki temu będzie można łatwiej odnosić opisy do rysunków oraz kodów źródłowych.

2.1. Analiza otrzymanych par plików

Pracę rozpocząłem od analizy otrzymanych par plików. Każda para składała się z pliku zaszyfrowanego oraz pliku oryginalnego (przed zaszyfrowaniem). Do analizy wykorzystałem narzędzie Notepad++ z dodatkową wtyczką Hex-Editor. Poniżej prezentuję zawartość binarną pliku oryginalnego (gif) oraz pliku zaszyfrowanego.

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	47	49	46	38	39	61	08	00	08	00	d0	90	01	00	d0	be	GIF89a....Đ...ĐI
00000010	d0	be	d0	be	c2	a4	c2	a4	c2	a4	21	d1	89	04	01	0a	ĐIĐIÂ¤Â¤Â¤!Ń‰...
00000020	00	01	00	2c	00	00	00	00	08	00	08	00	00	02	0d	04	...,............
00000030	12	d0	96	d0	aa	d0	97	0f	d0	b3	63	e2	80	98	31	d0	.Đ–ĐŞĐ—.Đłcâ€˜1Đ
00000040	99	5f	01	00	3b												™_..;

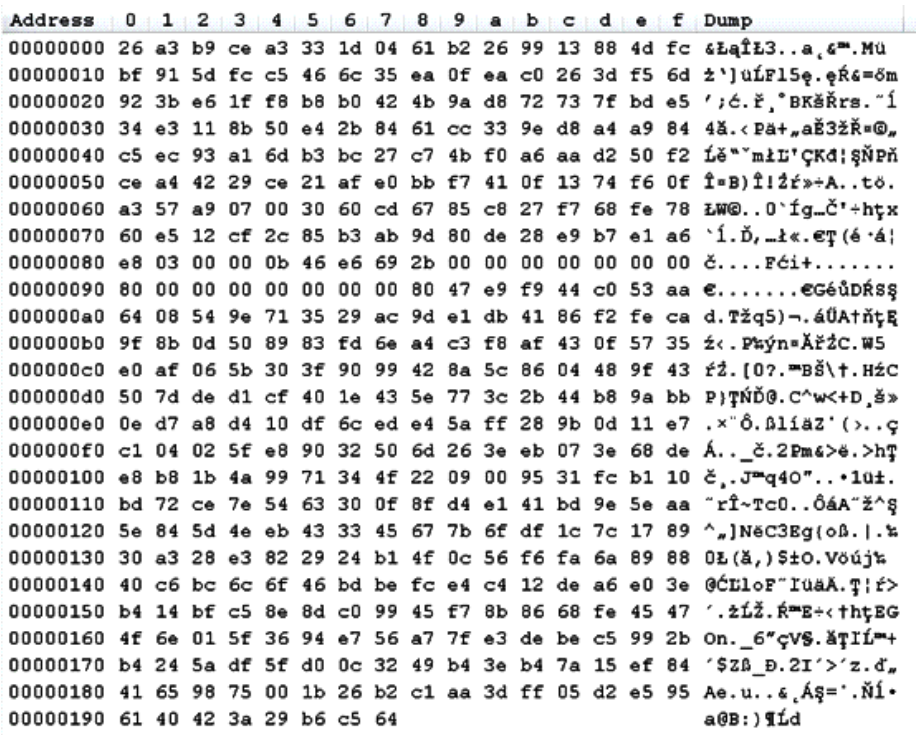
Rys. 1. Plik oryginalny



Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	57	34	4e	c7	24	60	e3	fd	70	34	c6	2c	2e	8b	11	01	W4NÇ$`ăýp4Ć,.‹..
00000010	41	5a	df	a8	d4	84	86	dc	9a	f7	23	fc	f6	a9	73	40	AZß¨Ô„†Üš÷#üö©s@
00000020	ea	60	58	ec	8b	91	8d	38	83	eb	19	cf	ff	58	9c	c6	ę`Xě‹ '.8ë.Ď˙XśĆ
00000030	1b	8d	06	3f	b8	a3	c2	54	52	c7	04	c9	97	2a	89	5a	...?¸ŁÂTRÇ.É—*‰Z
00000040	26	b3	53	74	10	3e	3c	b9	b1	c8	dc	69	8d	fc	f5	eb	&łSt.><ąąČÜi.üőë
00000050	61	29	1d	2f	18	3c	a3	4f	0a	d6	78	4c	1f	43	5f	3c	a)./.<ŁO.ÖxL.C_<
00000060	03	8a	e7	a2	b8	1f	2d	2a	5f	39	23	e3	47	cc	03	f9	.Šç˘¸.-*_9#ăGĚ.ů
00000070	9e	27	e4	ef	aa	f2	59	ca	c8	c1	eb	d9	a3	65	88	90	ž'ädşńYĘČÁëŮŁe.
00000080	e8	03	00	00	b7	bc	a6	96	36	00	00	00	00	00	00	00	č...·¼¦–6.......
00000090	80	00	00	00	00	00	00	00	80	de	bc	9c	68	f8	49	43	€.......€ŢĽśhřIC
000000a0	d1	69	0d	84	48	88	64	6a	dc	f2	16	09	af	dd	ee	c9	Ńi.„HdjÜň..ŻÝîÉ
000000b0	4a	04	31	5a	26	0e	d6	3f	8f	08	d2	e0	df	af	b9	b6	J.1Z&.Ö?..ŇŕßŻą¶
000000c0	cb	45	1d	b6	a1	2b	5a	26	f1	a4	fa	86	25	02	2a	37	ËE.¶˘+Z&ń¤ú†%.*7
000000d0	10	c2	8d	3a	57	9b	74	a2	cf	fc	7e	f0	81	12	cd	bc	.Â.:W›t˘Ďü~đ..ÍĽ
000000e0	7f	06	fe	75	94	c2	ca	83	e8	e3	40	27	c6	67	32	c1	..ţu”ÂĘăč@'Ćg2Á
000000f0	54	01	12	27	f6	af	02	c3	28	c9	0a	28	f7	bd	5b	05	T..'öŻ.Ă(É.(÷˝[.
00000100	d9	4b	a8	37	fc	5d	36	3d	63	23	63	e0	2a	0d	bf	1f	ŮK¨7ü]6=c#cŕ*.ż.
00000110	a0	f9	b7	f1	8f	77	15	fd	b1	13	83	bf	ba	e3	20	1d	.ů·ń.w.ý±.żşă .
00000120	bd	76	5e	8e	96	e5	2b	fa	1b	81	09	89	d0	2e	97	e9	˝v^Ž–Í+ú...‰Đ.—é
00000130	42	c6	cb	b4	63	5b	79	7a	d5	ed	ce	fc	64	e6	f0	cc	BĆË´c[yzŐíÎüdćđĚ
00000140	c7	df	2c	ff	20	36	38	1d	0b	b4	5e	5b	bd	42	4d	6c	Çß,˙ 68..´^[˝BMl
00000150	ca	e0	34	5f	8d	93	2e	a2	32	4c	a1	ee	2b	12	41	e8	Ęŕ4_.“.˘2L˘î+.Ač
00000160	62	b8	02	a4	5f	8f	37	40	80	27	53	89	74	5e	98	d0	b¸.¤_.7@€'S‰t^.Đ
00000170	b9	2b	4a	4c	7b	d0	f5	29	d0	38	8f	2a	a7	63	ef	35	ą+JL{Đő)Đ8.*§cď5
00000180	1a	9c	e8	f0	86	f1	ba	c6	88	d8	29	90	2a	d1	66	73	.śčđ†ńşĆŘ).*Ńfs
00000190	d7	25	1f	3a	b8	cb	2d	7a									×%.:¸Ë-z

Rys. 2. Plik zaszyfrowany

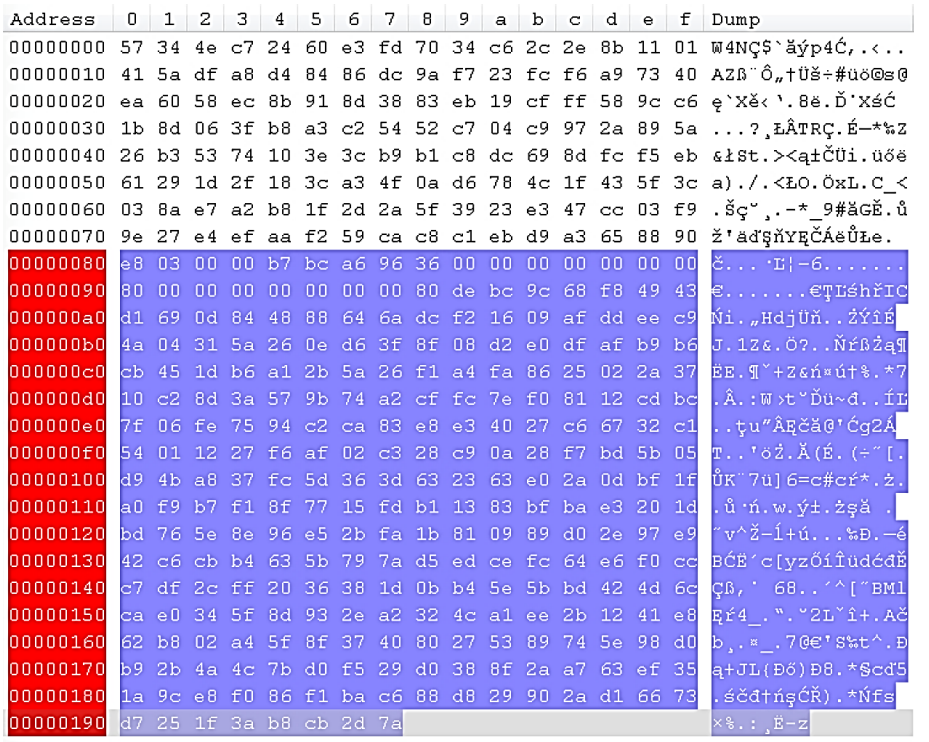
Rozmiar pliku oryginalnego wynosił 0x36 bajtów, rozmiar pliku zaszyfrowanego to 0x197 bajtów. To co na pierwszy rzut oka było widać w pliku zaszyfrowanym, to pewna regularność zaczynająca się od bajtu numer 0x89. Fragment ten nie wyglądał jak fragment szyfrogramu. Analizując dalej, regularność tę można było powiększyć — zaczynała się od bajtu numer 0x80. Regularność ta wskazywała, że od tego miejsca plik nie był zaszyfrowany, co mogło wskazywać na to, że były to jakieś dodatkowe informacje, na bazie których atakujący będą w stanie odszyfrować zawartość pliku. Moje przypuszczenia zostały potwierdzone analizą kilku kolejnych zaszyfrowanych plików.



Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	26	a3	b9	ce	a3	33	1d	04	61	b2	26	99	13	88	4d	fc	&ŁąÎŁ3..a¸&™.Mü
00000010	bf	91	5d	fc	c5	46	6c	35	ea	0f	ea	c0	26	3d	f5	6d	ż‘]üĹFl5ę.ęŔ&=őm
00000020	92	3b	e6	1f	f8	b8	b0	42	4b	9a	d8	72	73	7f	bd	e5	’;ć.ř¸°BKšŘrs.˝Í
00000030	34	e3	11	8b	50	e4	2b	84	61	cc	33	9e	d8	a4	a9	84	4ă.‹Pä+„aĚ3žŘ¤®„
00000040	c5	ec	93	a1	6d	b3	bc	27	c7	4b	f0	a6	aa	d2	50	f2	Ĺě“¡m³ĽˇÇKđ¦ŞŇPň
00000050	ce	a4	42	29	ce	21	af	e0	bb	f7	41	0f	13	74	f6	0f	Î¤B)Î!Żŕ»÷A..tö.
00000060	a3	57	a9	07	00	30	60	cd	67	85	c8	27	f7	68	fe	78	ŁW©..0`Íg…Č'÷hţx
00000070	60	e5	12	cf	2c	85	b3	ab	9d	80	de	28	e9	b7	e1	a6	`Í.Ď,…ł«.€Ţ(é·á¦
00000080	e8	03	00	00	0b	46	e6	69	2b	00	00	00	00	00	00	00	č....Fći+.......
00000090	80	00	00	00	00	00	00	00	80	47	e9	f9	44	c0	53	aa	€.......€GéůDŔSş
000000a0	64	08	54	9e	71	35	29	ac	9d	e1	db	41	86	f2	fe	ca	d.Tžq5)¬.áŰA†ňţĘ
000000b0	9f	8b	0d	50	89	83	fd	6e	a4	c3	f8	af	43	0f	57	35	ź‹.P‰ýn¤ĂřŻC.W5
000000c0	e0	af	06	5b	30	3f	90	99	42	8a	5c	86	04	48	9f	43	ŕŻ.[0?.™BŠ\†.HźC
000000d0	50	7d	de	d1	cf	40	1e	43	5e	77	3c	2b	44	b8	9a	bb	P}ŢŃĎ@.C^w<+D¸š»
000000e0	0e	d7	a8	d4	10	df	6c	ed	e4	5a	ff	28	9b	0d	11	e7	.×¨Ô.ßlíäZ˙(›..ç
000000f0	c1	04	02	5f	e8	90	32	50	6d	26	3e	eb	07	3e	68	de	Á.._č.2Pm&>ë.>hŢ
00000100	e8	b8	1b	4a	99	71	34	4f	22	09	00	95	31	fc	b1	10	č¸.J™q4O"..•1ü±
00000110	bd	72	ce	7e	54	63	30	0f	8f	d4	e1	41	bd	9e	5e	aa	˝rÎ~Tc0..ÔáA˝ž^ş
00000120	5e	84	5d	4e	eb	43	33	45	67	7b	6f	df	1c	7c	17	89	^„]NëC3Eg{oß.|.‰
00000130	30	a3	28	e3	82	29	24	b1	4f	0c	56	f6	fa	6a	89	88	0Ł(ă‚)$±O.Vöújˆ
00000140	40	c6	bc	6c	6f	46	bd	be	fc	e4	c4	12	de	a6	e0	3e	@ĆĽloF˝IüäÄ.Ţ¦ŕ>
00000150	b4	14	bf	c5	8e	8d	c0	99	45	f7	8b	86	68	fe	45	47	´.żĹŽ.Ŕ™E÷‹†hţEG
00000160	4f	6e	01	5f	36	94	e7	56	a7	7f	e3	de	be	c5	99	2b	On._6”çV§.ăŢIĹ™+
00000170	b4	24	5a	df	5f	d0	0c	32	49	b4	3e	b4	7a	15	ef	84	´$Zß_Đ.2I´>´z.ď„
00000180	41	65	98	75	00	1b	26	b2	c1	aa	3d	ff	05	d2	e5	95	Ae.u..&¸Áş=˙.ŇĺŇ
00000190	61	40	42	3a	29	b6	c5	64									a@B:)¶Ĺd

Rys. 3. Inny plik zaszyfrowany

Ze wstępnej analizy wynikało zatem, że każdy zaszyfrowany plik posiadał dodatkowe 0x118 bajtów doklejonych do jego końca. W dalszej części artykułu bajty te będę nazywał stopką.



Rys. 4. Zaznaczone dodatkowe bajty w pliku gif

W dalszym ciągu nieznane było znaczenie tych bajtów. Porównując je z bajtami innych zaszyfrowanych plików, spróbowałem określić format doklejonej stopki.

Tabela 1

Format doklejonej stopki

Pozycja startowa (oznaczenie)	Długość (bajty)	Znaczenie	Przykład
0x00 (A)	4	Początek stopki.	0xe8 0x03 0x00 0x00
0x04 (B)	4	Nieznane, powtarzalne dla większości plików pochodzących z tego samego katalogu.	0xb7 0xbc 0xa6 0x96
0x08 (C)	4	Rozmiar pliku oryginalnego.	0x36 0x00 0x00 0x00
0x0C (D)	4	Nieznane, zawsze zera. Być może jest to dalsza część pola C.	0x00 0x00 0x00 0x00

cd. tabeli 1

0x10 (E)	4	Rozmiar pliku zaszyfrowanego pomniejszony o długość stopki (0x118 bajtów).	0x80 0x00 0x00 0x00
0x14 (F)	4	Nieznane, zawsze zera. Być może jest to dalsza część pola E.	0x00 0x00 0x00 0x00
0x18 (G)	1	Nieznane, zawsze 0x80.	0x80
0x19 (H)	0x80	Nieznane, wyglądają na pseudolosowy ciąg bajtów albo szyfrogram, w każdym pliku inne.	0xde 0xbc 0x9c 0x68 0xf8 0x49 …..
0x99 (I)	0x7F	Nieznane, wyglądają na pseudolosowy ciąg bajtów albo szyfrogram, powtarzalne dla większości plków pochodzących z tego samego katalogu.	0x13 0x83 0xbf 0xba 0xe3 0x20 …

Analiza pola C oraz E wykazała, że rozmiar pliku zaszyfrowanego pomniejszony o rozmiar stopki (0x118 bajtów) powiększał się o 0x40 bajtów na każde 0x2000 bajtów pliku oryginalnego. Dodatkowo, dla plików oryginalnych o rozmiarze mniejszym niż 0x40 bajtów rozmiar E wynosił 0x80 bajtów. Oznaczało to, że oprócz standardowego powiększenia o 0x40 bajtów, mały plik był dopełniany do długości 0x40 bajtów. Dla większych plików dopełnienie nie było przeprowadzane. Analiza prowadziła do kilku wniosków:

1. Minimalna długość danych przetwarzana przez mechanizm kryptograficzny wynosiła 0x40 bajtów.
2. Pliki oryginalne o rozmiarze mniejszym niż 0x40 bajtów dopełniane były do 0x40 bajtów. W tym przypadku E = 0x80.
3. Pliki oryginalne o rozmiarze większym niż 0x40 bajtów nie były dopełniane. W tym przypadku E = C + 0x40 * ((C div 0x2000)+ 1). Operacja div oznacza dzielenie całkowitoliczbowe.
4. Dane pliku oryginalnego przetwarzane były w blokach o długości około 0x2000 bajtów.

Analiza pola B oraz I pokazała, że ich wartości powiązane były z folderami, w których znajdowały się zaszyfrowane pliki. W tamtej chwili nie było znane znaczenie i zastosowanie tych pól. Być może były to wartości inicjacyjne (IV) dla mechanizmów kryptograficznych.

Analiza pola G wykazała, że było ono takie samo dla wszystkich zaszyfrowanych plików.

Analiza pola I wykazała, że miało ono zawsze 0x80 bajtów długości oraz było różne w zaszyfrowanych plikach. W związku z tym przyjąłem założenie, że właśnie w polu I zapisany został klucz użyty do zaszyfrowania pliku.

2.2. Analiza otrzymanego klucza

Zawartość pliku z przesłanym kluczem w pierwszym momencie sugerowała, że był to certyfikat X509 zapisany w formacie PEM i zakodowany algorytmem Base64.

```
1  -----BEGIN CERTIFICATE-----
2  BwIAAACkAABSU0EyAAQAAAEAAQBnbNpwv3xXg2XcIluTP9ZMz3N53g5h9Mzka+rKAs1T
3  jj2lsBWyu0IaZCaP5BwcjtAdX7IK3N5k1VwWsqBAh765Y54ugP0ZqNRSBNpWPP5UIcQ3
4  lHGT+1k5RcH2t+8XshVHVncqRRxlm4AYb4OiRzv93Wtgq9v/7zXi5hX5ptTcvb0z0cqd
5  EAwvwZCxHhhelHunggcdIOYmf1DBREGVobJlVYIRRM3wr1iuIus4SDkPEpX6Lly9Ph7d
6  U180C2makOrzkJwYESWVeXIuV1w0EVVp1z+Rtc5X/JzGwFvKOra8OsI/ef505R/19pKD
7  jdPRSHqT8VGeA59Kh3Nf20xIdzbPCSP/9JnNvtnCePbTwHjePLHtoJG65HAI9YWP4/p2
8  N7wxPZv1QhaHPm4E3qnlgjj0je97brTNK+aBS9OHdNKrtrWIb+p7oishYwCItP6dcPnz
9  Al1CJ+nKh0gqVuuyB+zS/VNh167bRHnt2kmrlO91Yk7s5RdNIptN+Ug4LI+b8BFQVBsd
10 ADCu5AipnJYbMoJcxb8Nvx/7PNas+mAaORkkqKTq/qF9hDwBAZhzG3uU9b5aUgGkv4gO
11 DBTiDleBnmqMieM7BeRA5/vxjDOxeeIsEy0ktSaAPOJD5yMEA6h9t6kd8uMCYLHAUK5O
12 1f11mamuPeMrbQwC58XwOrHLyayk7aZEbVAG0lxmLjQeNKUMP4S3tjrIGWU+uszJCSZP
13 cOYs0f9Vw4cpOmPh2zdWuOXuN7zwKLvbEqrJRtsmaSl+RzU=
14 -----END CERTIFICATE-----
```

Rys. 5. Klucz

Niestety, dokładniejsza analiza, a także próba parsowania zgodnie ze standardem ASN.1 wykazała, że nie był to format PEM. Atakujący mogli utworzyć swój własny sposób zapisu klucza albo skorzystać z innej standardowej metody. Aby odpowiedzieć na to pytanie, rozpocząłem analizę danych klucza po wykonaniu dekodowania algorytmem Base64.

```
Address  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  Dump
00000000 07 02 00 00 00 a4 00 00 52 53 41 32 00 04 00 00 .....¤..RSA2....
00000010 01 00 01 00 67 6c da 70 bf 7c 57 83 65 dc 22 5b ....glÚpż|WeÜ"[
00000020 93 3f d6 4c cf 73 79 de 0e 61 f4 cc e4 6b ea ca "?ÖLĎsyŢ.aôĚäkęĘ
00000030 02 cd 53 8e 3d a5 b0 15 b2 bb 42 1a 64 26 8f e4 .ÍSŽ=Ą°.¸»B.d&.ä
00000040 1c 1c 8e d0 1d 5f b2 0a dc de 64 d5 5c 16 b2 a0 ..ŽĐ._¸.ÜŢdŐ\.¸.
00000050 40 87 be b9 63 9e 2e 80 fd 19 a8 d4 52 04 da 56 @‡ľącž.€ý.¨ÔR.ÚV
00000060 3c fe 54 21 c4 37 94 71 93 fb 59 39 45 c1 f6 b7 <ţT!Ä7"q"űY9EÁö·
00000070 ef 17 b2 15 47 56 77 2a 45 1c 65 9b 80 18 6f 83 ď.¸.GVw*E.e›€.o
00000080 a2 47 3b fd dd 6b 60 ab db ff ef 35 e2 e6 15 f9 ˘G;ýÝk`«Ű˙ď5âć.ů
00000090 a6 d4 dc bd bd 33 d1 ca 9d 10 0c 2f c1 90 b1 1e ¦ÔÜ˝˝3ŃĘ.../Á.±.
000000a0 18 5e 94 7b a7 82 07 1d 20 e6 26 7f 50 c1 44 41 .^"{§,.. ć&.PÁDA
000000b0 95 a1 b2 65 55 84 d1 44 cd f0 af 58 ae 22 eb 38 •˙¸eU„ŃDÍđŻX®"ë8
000000c0 48 39 0f 12 95 fa 2e 5c bd 3e 1e dd 53 5f 34 0b H9..•ú.\˝>.ÝS_4.
```

Rys. 6. 0xd0 pierwszych bajtów otrzymanego klucza po zdekodowaniu algorytmem Base64

Pierwsze, zaznaczone na rysunku, bajty klucza wskazywały na wykorzystanie standardowej metody zapisu danych klucza. Dodatkowo pojawił się napis „RSA2", co raczej nie mogło być przypadkiem. Dalsza analiza zawartości klucza wskazywała na to, że jest to wyeksportowany klucz prywatny algorytmu RSA zapisany w strukturze PRIVATEKKEYBLOB obsługiwanej przez biblioteki kryptograficzne dostarczane z systemem operacyjnym Windows.

```
PUBLICKEYSTRUC  publickeystruc;
RSAPUBKEY rsapubkey;
BYTE modulus[rsapubkey.bitlen/8];
BYTE prime1[rsapubkey.bitlen/16];
BYTE prime2[rsapubkey.bitlen/16];
BYTE exponent1[rsapubkey.bitlen/16];
BYTE exponent2[rsapubkey.bitlen/16];
BYTE coefficient[rsapubkey.bitlen/16];
BYTE privateExponent[rsapubkey.bitlen/8];
```

Rys. 7. Struktura PRIVATEKEYBLOB
(https://msdn.microsoft.com/en-us/library/windows/desktop/aa387453(v=vs.85).aspx)

W celu sparsowania otrzymanego klucza oraz weryfikacji jego kryptograficznej poprawności napisałem program w języku Java. W poniższej tabeli przedstawione są najważniejsze wyniki parsowania.

Tabela 2

Tabela z wynikami parsowania

Moduł (P*Q)	bddcd4a6f915e6e235efffdbab606bddfd3b47a2836f18809b651c452a77564715b217efb7f6c1453959fb93719437c42154fe3c56da0452d4a819fd802e9e63b9be8740a0b2165cd564dedc0ab25f1dd08e1c1ce48f26641a42bbb215b0a53d8e53cd02caea6be4ccf4610ede7973cf4cd63f935b22dc6583577cbf70da6c67
Wykładnik publiczny	10001
Wykładnik prywatny	35477e296926db46c9aa12dbbb28f0bc37eee5b85637dbe1633a2987c355ffd12ce6704f2609c9ccba3e6519c83ab6b7843f0ca5341e342e665cd206506d44a6eda4acc9cbb13af0c5e7020c6d2be33daea99975fdd54eae50c0b16002e3f21da9b77da8030423e743e23c8026b5242d132ce279b1338cf1fbe740e4053be389
Liczba pierwsza Q	ea909a690b345f53dd1e3ebd5c2efa95120f394838eb22ae58aff0cd44d1845565b2a1954144c1507f26e6201d0782a77b945e181eb190c12f0c109dcad133bd
Liczba pierwsza P	cf3677484cdb5f73874a9f039e51f1937a48d1d38d8392f6e51fe574fe793fc23abcb63aca5bc0c69cfc57ceb5913fd7695511345c57ee7479952511189c90f3

Analiza uzyskanych wyników wykazała, że otrzymany klucz jest poprawnym pod względem kryptograficznym kluczem prywatnym algorytmu RSA z modułem długości 0x80 bajtów (1024 bity). W związku z tym dane, które miały podlegać deszyfrowaniu, powinny były mieć długość 0x80 bajtów, co odpowiadało długości pola H ze stopki zaszyfrowanego pliku. W związku z tym podjąłem próbę

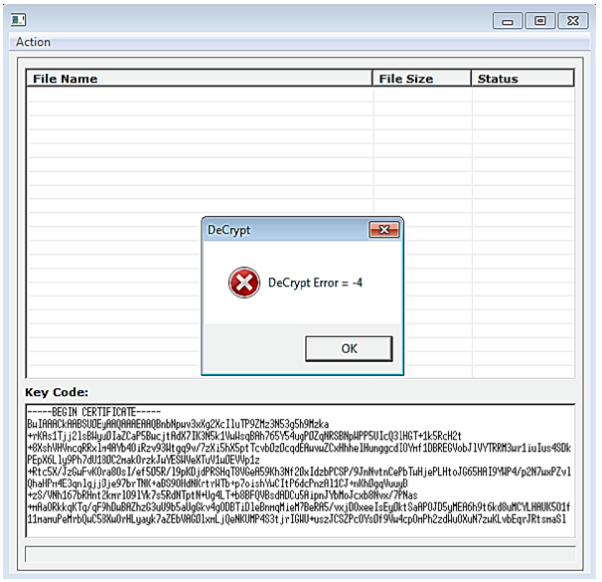
odszyfrowania zawartości pola H, korzystając z algorytmu RSA wraz z algorytmem dopełnienia PKCS#1 w wersji 1.5. Odszyfrowanie zakończyło się pomyślnie, czyli dane nie były większe od modułu i dopełnienie było zgodne z formatem PKCS#1. W wyniku odszyfrowania otrzymałem ciąg 0x40 znaków ASCII postaci: m.R.@kt”KBt*N9>XJYp_L>qkPO[S?ZdPbWjqpNfO0%MJy.o[m_+Y)!Kq”!RRU_^A

W przypadku innych zaszyfrowanych plików w wyniku odszyfrowania pola H również otrzymywałem ciągi złożone z 0x40 znaków ASCII.

Na bazie analizy zawartości klucza oraz poprawnie odszyfrowanej zawartości pola stopki H wnioskowałem, że w polu tym zapisany był klucz użyty w jakimś algorytmie symetrycznym, którym zaszyfrowany został plik. Algorytm ten mógł być szyfrem blokowym, szyfrem strumieniowym lub jakimś autorskim mechanizmem atakujących. Analizując jednak format pliku zaszyfrowanego oraz format udostępnionego klucza, bardziej prawdopodobne wydawało się zastosowanie znanego algorytmu blokowego albo strumieniowego niż własnego pomysłu atakujących.

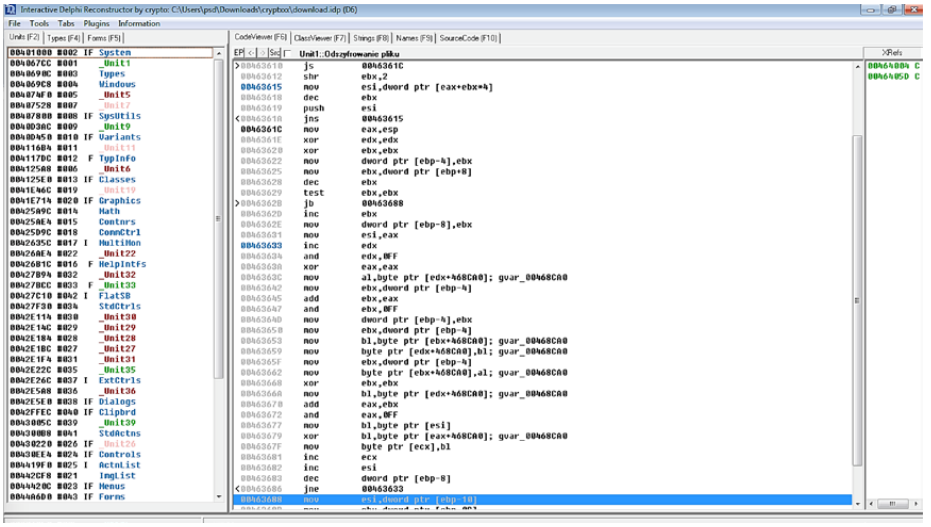
2.3. Analiza otrzymanego programu deszyfrującego

Miałem już klucz prywatny do algorytmu RSA oraz klucz do algorytmu symetrycznego. Brakowało mi jednak informacji, jaki algorytm symetryczny został wykorzystany. Nie wiedziałem też, w jaki sposób ten algorytm został użyty. Dodatkowo chciałem się dowiedzieć, dlaczego podczas próby odszyfrowania pojawiał się komunikat „DeCrypt Error = -4”.



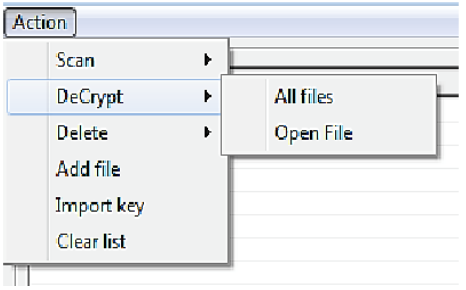
Rys. 8. Błąd programu deszyfrującego

Dalszą pracę rozpocząłem od analizy postaci mnemonicznej (kod assembler) uzyskanej za pomocą narzędzi typu disasembler. Wstępna analiza programu deszyfrującego wykazała, że został on napisany w języku Delphi. Do uzyskania kodu assemblera wykorzystałem aplikację Interactive Delphi Reconstructor w najnowszej dostępnej na stronie producenta wersji.



Rys. 9. IDR

Wstępny przegląd uzyskanego kodu wskazywał, że w przypadku narzędzia do deszyfrowania atakujący nie użyli żadnych technik zaciemniania kodu. Dzięki temu kod był w miarę łatwy do czytania oraz analizy. W pierwszej kolejności zająłem się szukaniem funkcji, która jest wywoływana po kliknięciu w menu opcji „DeCrypt->All files".



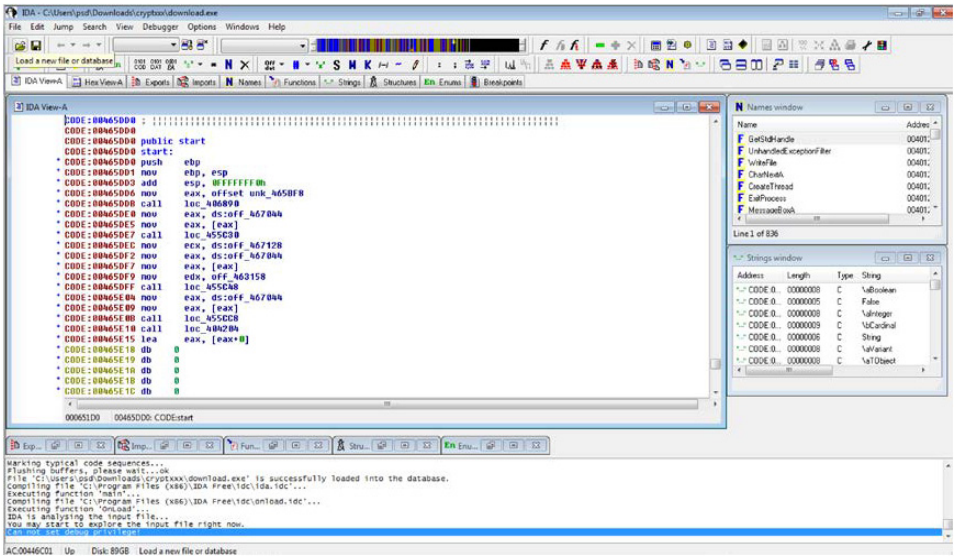
Rys. 10. Opcja decrypt

Funkcję znalazłem pod adresem 00465B0C.

```
Unit1::Opcja DeCrypt->All files
00465B0C    push        ebp
00465B0D    mov         ebp,esp
00465B0F    add         esp,0FFFFFFF8
00465B12    xor         ecx,ecx
00465B14    mov         dword ptr [ebp-8],ecx
00465B17    xor         eax,eax
00465B19    push        ebp
00465B1A    push        465B76
00465B1F    push        dword ptr fs:[eax]
00465B22    mov         dword ptr fs:[eax],esp
00465B25    lea         edx,[ebp-8]
00465B28    mov         eax,[00468C8C]; gvar_00468C8C:TForm1
00465B2D    mov         eax,dword ptr [eax+2F8]
00465B33    call        TControl.GetText
00465B38    mov         edx,dword ptr [ebp-8]
00465B3B    mov         eax,468C98; gvar_00468C98:AnsiString
00465B40    call        @LStrAsg
00465B45    call        004641A0
00465B4A    push        0
00465B4C    push        0
00465B4E    lea         eax,[ebp-4]
00465B51    push        eax
00465B52    mov         ecx,465724
00465B57    xor         edx,edx
00465B59    xor         eax,eax
00465B5B    call        BeginThread
00465B60    xor         eax,eax
00465B62    pop         edx
00465B63    pop         ecx
00465B64    pop         ecx
00465B65    mov         dword ptr fs:[eax],edx
00465B68    push        465B7D
00465B6D    lea         eax,[ebp-8]
00465B70    call        @LStrClr
00465B75    ret
<00465B76   jmp         @HandleFinally
<00465B7B   jmp         00465B6D
00465B7D    pop         ecx
00465B7E    pop         ecx
00465B7F    pop         ebp
00465B80    ret
```

Rys. 11. Funkcja All files

W następnej kolejności postanowiłem uruchomić program deszyfrujący z podłączonym do niego narzędziem typu debugger. Chciałem zobaczyć, co i jak dokładnie jest robione. W tym celu skorzystałem z oprogramowania IDA firmy Hex-Rays. Była to darmowa wersja v5.0.



Rys. 12. IDA

Trwająca kilka godzin analiza wykonania programu deszyfrującego umożliwiła mi poznanie algorytmu szyfrującego pliki oraz wyjaśniła, dlaczego program zwraca komunikat „DeCrypt Error = -4.". Wersja oprogramowania, którą analizowałem, szukała w pliku zaszyfrowanym pola A ze stopki w bajcie numer 0x104, licząc od końca pliku. Jeżeli pole A nie zostało tam odnalezione, program zwracał wspomniany wcześniej komunikat i kończył przetwarzanie pliku. Postanowiłem więc ręcznie zmodyfikować zaszyfrowany plik w taki sposób, aby format pliku był kompatybilny z programem. Dzięki temu przetwarzanie pliku przebiegało dalej i umożliwiło poznanie mechanizmu szyfrowania. Zmodyfikowany plik został przetworzony przez program deszyfrujący, jednak odszyfrowane dane były nieprawidłowe.

Analiza zachowania programu i przyczyn zwracanego błędu pozwalała sądzić, że program ten służył do odszyfrowywania plików zaszyfrowanych wirusem CryptXXX w wersji wcześniejszej niż 3. W związku z tym, z punktu widzenia użytkownika, program był całkowicie bezużyteczny. Z mojego punktu widzenia dostarczał jednak wiedzy, jak mechanizm szyfrowania działał kiedyś. Zakładając, że w wersji 3 mechanizm uległ jedynie drobnym modyfikacjom, można było z programu uzyskać wiele przydatnych informacji.

2.4. Odczytane algorytmy z programu deszyfrującego

Analiza kodu programu deszyfrującego podczas jego działania umożliwiła uzyskanie informacji, w jaki sposób działała poprzednia wersja mechanizmu szyfrującego pliki. Udało mi się ustalić, że:

1. Do każdego zaszyfrowanego pliku doklejana była stopka o długości 0x104 bajtów. Stopka zaczynała się sekwencją bajtów: 0xe8 0x03 0x00 0x00.
2. Korzystano jedynie z 0x80 bajtów stopki, w których zapisany był zaszyfrowany algorytmem RSA (1024 bity) klucz. Klucz ten, po odszyfrowaniu, składał się z 0x40 znaków ASCII. Pozostała część danych w stopce nie była używana.
3. Odszyfrowany klucz używany był do inicjalizacji nieznacznie zmodyfikowanej wersji algorytmu RC4.
4. Dane pliku dzielone były na bloki długości 0x2000 bajtów i szyfrowane algorytmem RC4 (alternatywa wykluczająca z wygenerowanym strumieniem klucza). Dla każdego bloku resetowany był indeks wygenerowanego bajtu w algorytmie RC4 na 1. Wydaje się, że ta słabość mogła zostać wykorzystana w narzędziu udostępnionym przez firmę Kaspersky do odszyfrowywania plików zaszyfrowanych CryptXXX bez znajomości klucza prywatnego RSA.
5. Do obsługi kryptografii asymetrycznej (RSA) atakujący korzystali z API systemu Windows (m.in. funkcja CryptDecrypt).
6. Do obsługi kryptografii symetrycznej atakujący korzystali z własnej implementacji szyfru RC4. Opisana w punkcie 4 modyfikacja może wynikać z błędnej implementacji, a nie celowej modyfikacji.
7. Długość zaszyfrowanego pliku była równa sumie długości pliku oryginalnego oraz stopki (0x104 bajty).

Poniżej przedstawiony został znaleziony w kodzie algorytm inicjalizacji RC4 na podstawie klucza zapisanego w polu H stopki.

```
0046356C  push      ebx
0046356D  push      esi
0046356E  push      edi
0046356F  add       esp,0FFFFFE00
00463575  mov       esi,eax
00463577  lea       edi,[esp]
0046357A  xor       ecx,ecx
0046357C  mov       cl,byte ptr [esi]
0046357E  inc       ecx
0046357F  rep movs  byte ptr [edi],byte ptr [esi]
00463581  xor       edx,edx
00463583  mov       eax,468CA0; gvar_00468CA0
00463588  mov       byte ptr [eax],dl
0046358A  inc       edx
0046358B  inc       eax
```

```
0046358C  test    dl,dl
0046358E  jne     00463588
00463590  mov     al,1
00463592  movzx   esi,byte ptr [esp]
00463596  mov     dl,0
00463598  lea     ecx,[esp+100]
0046359F  xor     ebx,ebx
004635A1  mov     bl,al
004635A3  mov     bl,byte ptr [esp+ebx]
004635A6  mov     byte ptr [ecx],bl
004635A8  xor     ebx,ebx
004635AA  mov     bl,al
004635AC  cmp     esi,ebx
004635AE  jne     004635B2
004635B0  xor     eax,eax
004635B2  inc     eax
004635B3  inc     ecx
004635B4  dec     dl
004635B6  jne     0046359F
004635B8  mov     dl,0
004635BA  lea     esi,[esp+100]
004635C1  mov     edi,468CA0; gvar_00468CA0
004635C6  and     eax,0FF
004635CB  xor     ebx,ebx
004635CD  mov     bl,byte ptr [esi]
004635CF  add     eax,ebx
004635D1  mov     cl,byte ptr [edi]
004635D3  xor     ebx,ebx
004635D5  mov     bl,cl
004635D7  add     eax,ebx
004635D9  and     eax,0FF
004635DE  xor     ebx,ebx
004635E0  mov     bl,al
004635E2  mov     bl,byte ptr [ebx+468CA0]; gvar_00468CA0
004635E8  mov     byte ptr [edi],bl
004635EA  xor     ebx,ebx
004635EC  mov     bl,al
004635EE  mov     byte ptr [ebx+468CA0],cl; gvar_00468CA0
004635F4  inc     edi
004635F5  inc     esi
```

```
004635F6  dec     dl
004635F8  jne     004635C6
004635FA  add     esp,200
00463600  pop     edi
00463601  pop     esi
00463602  pop     ebx
00463603  ret
```

Poniżej przedstawiony został znaleziony w kodzie algorytm RC4.

```
00463604  push    ebp
00463605  mov     ebp,esp
00463607  add     esp,0FFFFFFF8
0046360A  push    ebx
0046360B  push    esi
0046360C  mov     ebx,edx
0046360E  test    ebx,ebx
>00463610  js     0046361C
00463612  shr     ebx,2
00463615  mov     esi,dword ptr [eax+ebx*4]
00463618  dec     ebx
00463619  push    esi
0046361A  jns     00463615
0046361C  mov     eax,esp
0046361E  xor     edx,edx
00463620  xor     ebx,ebx
00463622  mov     dword ptr [ebp-4],ebx
00463625  mov     ebx,dword ptr [ebp+8]
00463628  dec     ebx
00463629  test    ebx,ebx
0046362B  jb      00463688
0046362D  inc     ebx
0046362E  mov     dword ptr [ebp-8],ebx
00463631  mov     esi,eax
00463633  inc     edx
00463634  and     edx,0FF
0046363A  xor     eax,eax
0046363C  mov     al,byte ptr [edx+468CA0]; gvar_00468CA0
00463642  mov     ebx,dword ptr [ebp-4]
00463645  add     ebx,eax
```

```
00463647   and      ebx,0FF
0046364D   mov      dword ptr [ebp-4],ebx
00463650   mov      ebx,dword ptr [ebp-4]
00463653   mov      bl,byte ptr [ebx+468CA0]; gvar_00468CA0
00463659   mov      byte ptr [edx+468CA0],bl; gvar_00468CA0
0046365F   mov      ebx,dword ptr [ebp-4]
00463662   mov      byte ptr [ebx+468CA0],al; gvar_00468CA0
00463668   xor      ebx,ebx
0046366A   mov      bl,byte ptr [edx+468CA0]; gvar_00468CA0
00463670   add      eax,ebx
00463672   and      eax,0FF
00463677   mov      bl,byte ptr [esi]
00463679   xor      bl,byte ptr [eax+468CA0]; gvar_00468CA0
0046367F   mov      byte ptr [ecx],bl
00463681   inc      ecx
00463682   inc      esi
00463683   dec      dword ptr [ebp-8]
00463686   jne      00463633
00463688   mov      esi,dword ptr [ebp-10]
0046368B   mov      ebx,dword ptr [ebp-0C]
0046368E   mov      esp,ebp
00463690   pop      ebp
00463691   ret      8
```

W wyniku przeprowadzonej analizy poznałem sposób działania CryptXXX w wersji 2. Na tamtą chwilę pozostawała mi nadzieja, że mechanizm szyfrowania w wersji 3 zmienił się tylko nieznacznie. Na podstawie przeprowadzonej do tej pory analizy mogłem wnioskować o podobieństwach oraz różnicach pomiędzy CryptXXX w wersji 2 oraz w wersji 3.

Możliwe podobieństwa:

1. Ten sam mechanizm (RSA 1024 bity) szyfrowania klucza użytego do zaszyfrowania pliku.
2. Każdy plik szyfrowany innym kluczem (wniosek z istnienia stopki w CryptXXX w wersji 2).
3. Zaszyfrowany algorytmem RSA klucz użyty do zaszyfrowania pliku zapisywany był jako fragment doklejanej do końca zaszyfrowanego pliku stopki.
4. Klucz użyty do szyfrowania składał się z 0x40 znaków ASCII.

Możliwe różnice:

1. Inny sposób szyfrowania pliku. W wersji 3 długość szyfrogramu była dłuższa niż długość tekstu jawnego. W wersji 2 długość szyfrogramu była równa długości tekstu jawnego.

2. Różne formaty stopki. W wersji 2 w stopce nie występowały pola B, C, D, E oraz F. Stopka w wersji 2 była krótsza o 0x14 bajtów.

Na bazie kluczy używanych do szyfrowania pliku spodziewałem się, że w wersji 3 użyty został algorytm RC4 z drobnymi modyfikacjami, które powodowały przyrost długości szyfrogramu.

2.5. Próba dojścia do sposobu szyfrowania użytego w wersji v3

Aby zweryfikować przypuszczenia z poprzedniego rozdziału, napisałem program w języku Java, w którym próbowałem stosować różne metody wykorzystania algorytmu RC4 do odszyfrowania pliku. Niestety, wielogodzinne próby nie przyniosły spodziewanego efektu — dalej nie znałem sposobu szyfrowania plików przez CryptXXX w wersji 3.

Poniżej prezentuję niektóre próby deszyfrowania, które przetestowałem na zaszyfrowanym pliku. Próby zakładały, że dodatkowe 0x40 bajtów, o które testowy szyfrogram był dłuższy od tekstu jawnego, były wykorzystywane w RC4 albo były nieużywane (analogicznie do niektórych bajtów w dodawanej do zaszyfrowanego pliku stopce).

Pracowałem na pliku o zawartości:

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	57	34	4e	c7	24	60	e3	fd	70	34	c6	2c	2e	8b	11	01	W4NÇ$`ăýp4Ć,.‹..
00000010	41	5a	df	a8	d4	84	86	dc	9a	f7	23	fc	f6	a9	73	40	AZß¨Ô„†Üš÷#üö©s@
00000020	ea	60	58	ec	8b	91	8d	38	83	eb	19	cf	ff	58	9c	c6	ę`Xě‹'.8ë.Ď˙XśĆ
00000030	1b	8d	06	3f	b8	a3	c2	54	52	c7	04	c9	97	2a	89	5a	...?¸ŁÂTRÇ.É—*‰Z
00000040	26	b3	53	74	10	3e	3c	b9	b1	c8	dc	69	8d	fc	f5	eb	&łSt.><ąŁČÜi.üőë
00000050	61	29	1d	2f	18	3c	a3	4f	0a	d6	78	4c	1f	43	5f	3c	a)./.<ŁO.ÖxL.C_<
00000060	03	8a	e7	a2	b8	1f	2d	2a	5f	39	23	e3	47	cc	03	f9	.Šç˘¸.-*_9#ăGĚ.ů
00000070	9e	27	e4	ef	aa	f2	59	ca	c8	c1	eb	d9	a3	65	88	90	ž'ädŞňYĘČÁëŮŁe.
00000080	e8	03	00	00	b7	bc	a6	96	36	00	00	00	00	00	00	00	č...·L¦–6.......
00000090	80	00	00	00	00	00	00	00	80	de	bc	9c	68	f8	49	43	€.......€ŢLśhřIC
000000a0	d1	69	0d	84	48	88	64	6a	dc	f2	16	09	af	dd	ee	c9	Ńi.„HdjÜň..ŻÝîÉ
000000b0	4a	04	31	5a	26	0e	d6	3f	8f	08	d2	e0	df	af	b9	b6	J.1Z&.Ö?..ŇŕßŻą¶
000000c0	cb	45	1d	b6	a1	2b	5a	26	f1	a4	fa	86	25	02	2a	37	ËE.¶˘+Z&ń¤ú†%.*7
000000d0	10	c2	8d	3a	57	9b	74	a2	cf	fc	7e	f0	81	12	cd	bc	.Â.:W›t˘Ďü~đ..ÍĽ
000000e0	7f	06	fe	75	94	c2	ca	83	e8	e3	40	27	c6	67	32	c1	..ţu”ÂĘčă@'Ćg2Á
000000f0	54	01	12	27	f6	af	02	c3	28	c9	0a	28	f7	bd	5b	05	T..'öŻ.Ă(É.(÷˝[.
00000100	d9	4b	a8	37	fc	5d	36	3d	63	23	63	e0	2a	0d	bf	1f	ŮK¨7ü]6=c#cŕ*.ż.
00000110	a0	f9	b7	f1	8f	77	15	fd	b1	13	83	bf	ba	e3	20	1d	.ů·ń.w.ý±.żşă .
00000120	bd	76	5e	8e	96	e5	2b	fa	1b	81	09	89	d0	2e	97	e9	˝v^Ž–Í+ú...‰Đ.—é
00000130	42	c6	cb	b4	63	5b	79	7a	d5	ed	ce	fc	64	e6	f0	cc	BĆË´c[yzŐíÎüdćđĚ
00000140	c7	df	2c	ff	20	36	38	1d	0b	b4	5e	5b	bd	42	4d	6c	Çß,˙ 68..´^[˝BMl
00000150	ca	e0	34	5f	8d	93	2e	a2	32	4c	a1	ee	2b	12	41	e8	Ęŕ4_.“.˘2L˘î+.Ač
00000160	62	b8	02	a4	5f	8f	37	40	80	27	53	89	74	5e	98	d0	b¸.¤_.7@€'S‰t^˜Đ
00000170	b9	2b	4a	4c	7b	d0	f5	29	d0	38	8f	2a	a7	63	ef	35	ą+JL{Đő)Đ8.*§cď5
00000180	1a	9c	e8	f0	86	f1	ba	c6	88	d8	29	90	2a	d1	66	73	.śčđ†ńşĆŘ).*Ńfs
00000190	d7	25	1f	3a	b8	cb	2d	7a									×%.:¸Ë-z

Rys. 13. Zawartość pliku

Plik podzieliłem na następujące części.

Tabela 3

Podział pliku

Nazwa części	Początek	Długość (bajty)
czesc_1	0x00	0x40
czesc_2	0x40	0x80
stopka	0x80	0x118

W dalszej części rozdziału będę używał funkcji *RC4(k, ct)*, gdzie *k* jest kluczem, a *ct* szyfrogramem. Klucz odczytany ze stopki będę oznaczał literą *K*.

Poniżej prezentuję kilka z przeprowadzonych prób odszyfrowania pliku.

Tabela 4

Wykorzystane schematy

Schemat	Opis	Wynik
RC4(K, ct)	Schemat działania zgodny z CryptXXX w wersji 2.	Nie udało się prawidłowo odszyfrować pliku.
RC4(K xor czesc_1, czesc_2)	Schemat, w którym klucz ma postać K xor czesc_1.	Nie udało się prawidłowo odszyfrować pliku.
RC4(K xor czesc_2, czesc_1)	Schemat, w którym klucz ma postać K xor czesc_2.	Nie udało się prawidłowo odszyfrować pliku.
RC4(RC4(K, czesc_2), czesc_1)	Schemat, w którym klucz ma postać RC4(K, czesc_2).	Nie udało się prawidłowo odszyfrować pliku.
RC4(RC4(K, czesc_1), czesc_2)	Schemat, w którym klucz ma postać RC4(K, czesc_1).	Nie udało się prawidłowo odszyfrować pliku.
RC4(K || czesc_1, czesc_2)	Schemat, w którym klucz ma postać konkatenacji K i czesc_1.	Nie udało się prawidłowo odszyfrować pliku.
RC4(K || czesc_2, czesc_1)	Schemat, w którym klucz ma postać konkatenacji K i czesc_2.	Nie udało się prawidłowo odszyfrować pliku.
RC4(czesc_1 || K, czesc_2)	Schemat, w którym klucz ma postać konkatenacji czesc_1 i K.	Nie udało się prawidłowo odszyfrować pliku.
RC4(czesc_2 || K, czesc_1)	Schemat, w którym klucz ma postać konkatenacji czesc_2 i K.	Nie udało się prawidłowo odszyfrować pliku.

Niestety, żadna z podjętych prób odszyfrowania pliku nie zakończyła się powodzeniem. Aby osiągnąć sukces, musiałem poznać algorytm szyfrowania CryptXXX

w wersji 3. Podjąłem zatem decyzję o próbie odtworzenia algorytmu poprzez analizę postaci mnemonicznej wirusa CryptXXX w wersji 3.

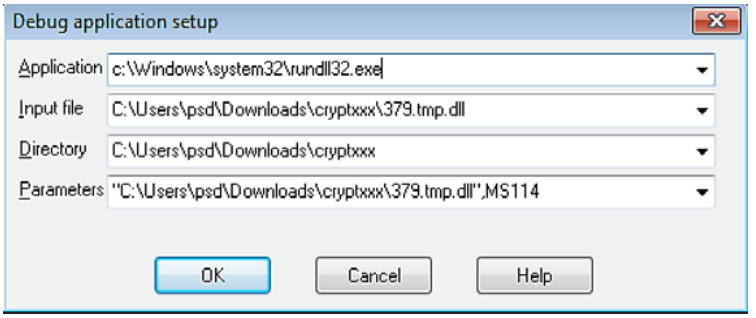
2.6. Analiza wirusa

Analizę wirusa planowałem przeprowadzić w środowisku wirtualnym poprzez statyczną analizę kodu. W tym celu wykorzystałem posiadaną maszynę wirtualną z systemem operacyjnym Windows 7, pracującą pod kontrolą VirtualBoxa. Do analizy kodu planowałem skorzystać z oprogramowania IDA. Miałem niestety jeden poważny problem — nie posiadałem kodu wirusa. Skorzystałem więc z forum internetowego na stronie http://bleepingcomputer.com. Znajduje się tam osobny wątek poświęcony wirusowi CryptXXX. Wirus w postaci pliku dll uzyskałem po około 12 godzinach od zamieszczenia prośby. Mogłem zatem przystąpić do analizy kodu.

Analizę rozpocząłem od wygenerowania postaci mnemonicznej za pomocą narzędzia IDA. Program IDR już się do tego nie nadawał, gdyż wirus nie został napisany w języku Delphi. Pierwsze etapy analizy statycznej nie wykazały żadnych niepokojących elementów. Plik DLL nie wyglądał na złośliwe oprogramowanie, które potrafi szyfrować pliki na dyskach.

Rys. 14. Importy z pliku

W funkcjach importowanych przez wirusa z pliku advapi32.dll nie było np. funkcji CryptEncrypt, której użycia się spodziewałem do szyfrowania klucza wykorzystanego do szyfrowania pliku. Inną funkcją, której się spodziewałem, była funkcja CryptExportKey, za jej pomocą utworzona została opisywana przeze mnie wcześniej struktura PRIVATEKEYBLOB. Tej funkcji też nie było. W pierwszej chwili zacząłem się zastanawiać, czy mam do czynienia z prawidłowym plikiem wirusa CryptXXX — w końcu dostałem go od użytkownika z forum. Postanowiłem zrobić prosty test. Utworzyłem snapshota obecnego stanu mojej wirtualnej maszyny i uruchomiłem kod zawarty w pliku DLL za pomocą programu rundll32.exe. Punktem wejściowym do pliku DLL było wejście MS114. Proces utworzony został z poziomu IDA tak, aby zachować przynajmniej częściową kontrolę nad wirusem.



Rys. 15. Uruchomienie wirusa

Po chwilowym śledzeniu kodu wirusa postanowiłem pozwolić na wykonywanie instrukcji w trybie ciągłym. Po chwili większość plików uzyskała rozszerzenie crypt, a na pulpicie pojawił się plik tekstowy opisujący sposób na odszyfrowanie plików. Empirycznie zatem potwierdziłem, że posiadany przeze mnie plik DLL to rzeczywiście plik z wirusem. Odtworzyłem więc utworzony parę chwil wcześniej snapshot i rozpocząłem wszystko od początku, starając się prześledzić dokładne działanie wirusa.

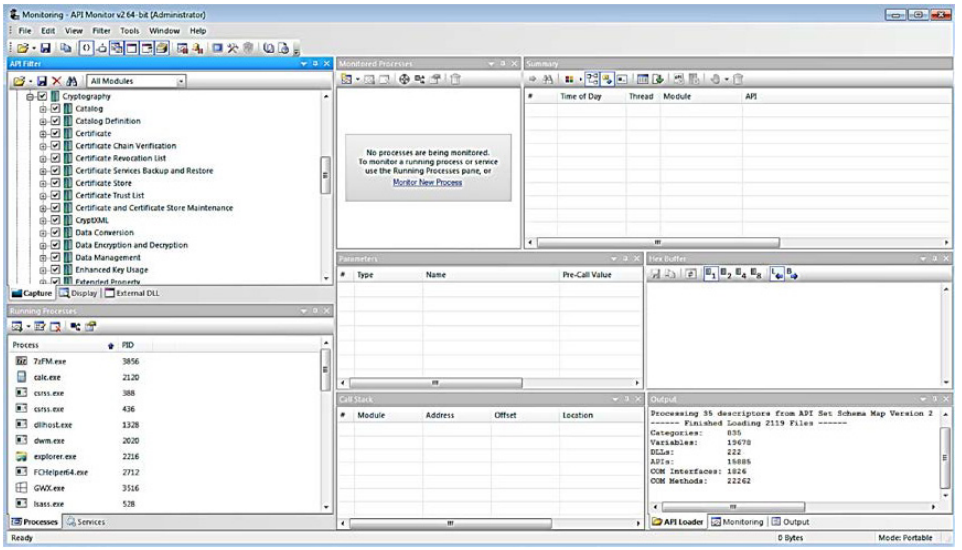
Kod wirusa był dużo trudniejszy do analizy niż kod deszyfratora. Wynikało to zapewne z użytych technik zaciemniania kodu. Kod wirusa zmieniał się podczas wykonywania, więc statyczna analiza nie była w stanie pokazać prawdziwego sposobu działania. Po kilku godzinach analiz wykonywanego kodu udało mi się ustalić, że wirus działa w następujący sposób:

1. Skopiowanie do folderu tymczasowego pliku rundll32.exe.
2. Zmiana nazwy pliku na svchost.exe.
3. Utworzenie nowego procesu (wywołanie z pliku kernel32.dll funkcji CreateProcessInternalW) w trybie uśpienia. Do procesu przekazane zostało polecenie …\svchost.exe „…..\379.tmp.dll",MS111.
4. Modyfikacja wczytanego do pamięci kodu z pliku 379.tmp.dll.
5. Wznowienie utworzonego w punkcie 3 procesu.
6. Złośliwe działania takie jak np. szyfrowanie plików wykonywane są przez osobny proces utworzony w punkcie 3 i wznowiony w punkcie 5.

Niestety, zastosowane przeze mnie narzędzie (IDA) nie umożliwiało debugowania wielu procesów jednocześnie. Próby podłączania się do nowo utworzonego procesu nie przynosiły pożądanych efektów. Być może wynikało to z mojego stanu wiedzy — to mój pierwszy analizowany wirus. Jak się później okazało, po przeczytaniu kilku publikacji w Internecie, zastosowana technika tworzenia procesu w trybie uśpienia jest jedną z technik mających na celu uniemożliwienie/utrudnienie debugowania kodu wirusa. Potwierdzam — to było utrudnienie.

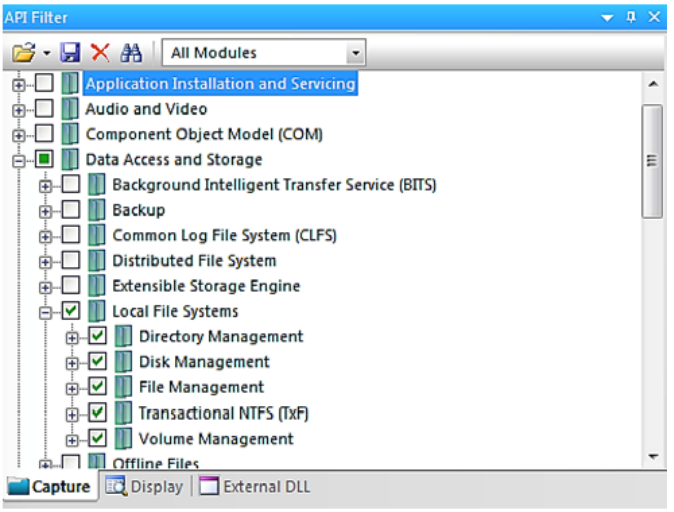
Przed wdrożeniem w życie poznanych właśnie technik debugowania takiego typu oprogramowania postanowiłem za pomocą programu APIMonitor prześledzić wywołania funkcji API systemu Windows, z których korzysta CryptXXX w wersji 3.

APIMonitor śledzi rzeczywiste, wykonane w trakcie działania wywołania API. Dodatkowo pokazuje wartości argumentów, z którymi wywołana została dana funkcja API.

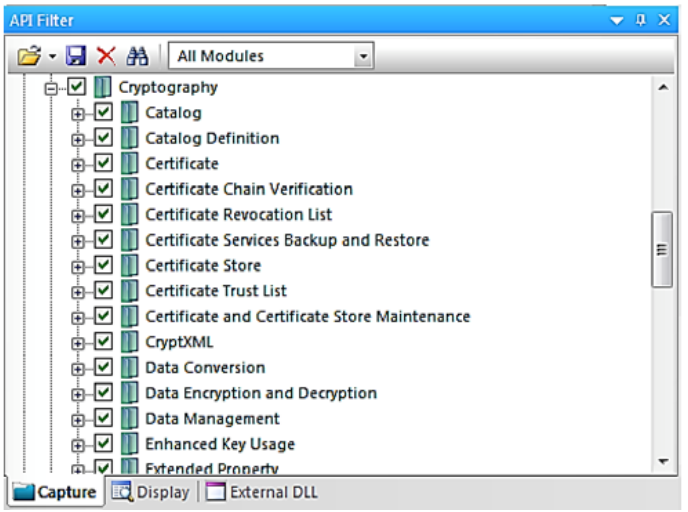


Rys. 16. APIMonitor

Odtworzyłem więc snapshot, uruchomiłem APIMonitor i wskazałem zakres funkcji API, który chcę monitorować. Wybrałem funkcje z zakresu dostępu do plików lokalnych oraz z zakresu kryptografii.

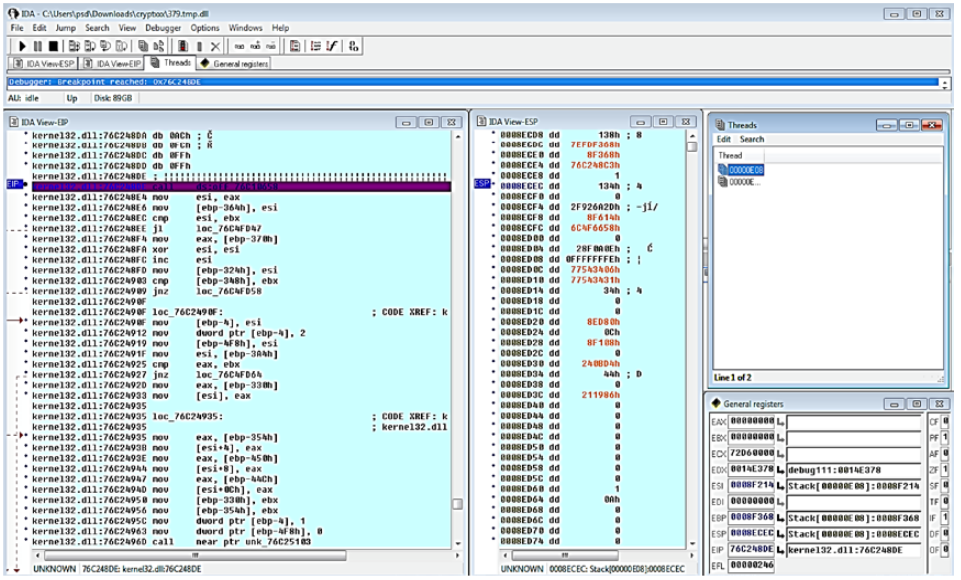


Rys. 17. Dostęp do plików



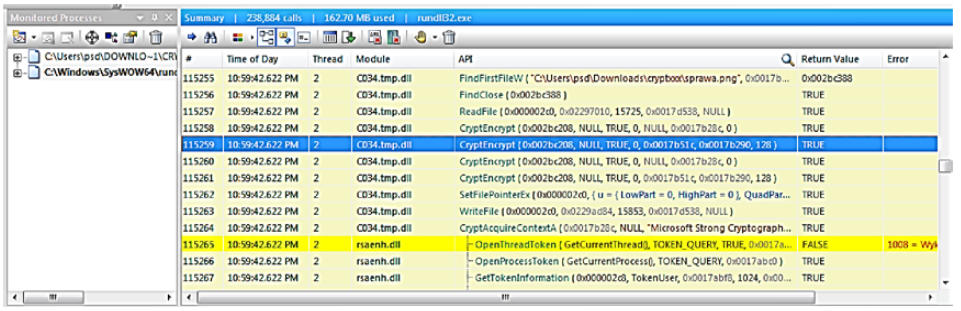
Rys. 18. Kryptografia

Następnie uruchomiłem wirusa z poziomu programu IDA. Breakpoint ustawiłem tuż przed wywołaniem procedury, która wznawia nowo utworzony proces wirusa. Przetwarzanie wirusa zatrzymało się po utworzeniu procesu, ale jeszcze przed jego wznowieniem. Dzięki temu bez problemu byłem w stanie wskazać w APIMonitor proces, który ma być monitorowany.



Rys. 19. Zatrzymany proces tuż przed wznowieniem procesu

Po wskazaniu w APIMonitor procesu, który zamierzałem monitorować, wznowiłem wykonywanie kodu wirusa w trybie ciągłym. Pliki na moim dysku zostały zaszyfrowane, a w APIMonitor pojawiło się wiele wpisów związanych z wywołanymi funkcjami API.



Rys. 20. Odnotowane wywołania API

Dokładna analiza odnotowanych wywołań funkcji API pozwoliła na wyciągnięcie nowych wniosków:

1. CryptXXX w wersji 3 wywołuje funkcję CryptEncrypt (w tym przypadku jest to szyfrowanie RSA) dla różnych plików różną liczbę razy. Dokładniejsza analiza wykazała, że liczba wywołań funkcji CryptEncrypt jest bezpośrednio związana z rozmiarem szyfrowanego pliku.
2. Każde nowe wywołanie CryptEncrypt pojawia się raz na każde mniej więcej 0x2000 bajtów szyfrowanego pliku.
3. Minimalna liczba wywołań funkcji CryptEncrypt to 2. Jedno z tych wywołań służy do zaszyfrowania klucza dla algorytmu RC4.

Dzięki APIMonitor udało się zdobyć kolejne informacje związane z działaniem CryptXXX w wersji 3.

Łącząc uzyskane informacje związane z działaniem wirusa CrytpXXX w wersji 3 z informacjami zdobytymi wcześniej, można wnioskować, że:

1. Algorytm szyfrowania został zmodyfikowany poprzez dodanie wywołania szyfrowania RSA raz na każde 0x2000 bajtów przetwarzanego pliku.
2. Wykorzystanie RSA do szyfrowania fragmentu danych pliku tłumaczy przyrost długości szyfrogramu o 0x40 bajtów dla każdych 0x2000 bajtów tekstu jawnego — szyfrując 0x40 bajtów algorytmem RSA (moduł 1024 bity) z dopełnieniem PKCS#1, otrzymujemy szyfrogram o długości 0x80 bajtów.

Otwartą kwestią pozostawało pytanie, czy w dalszym ciągu CryptXXX korzystał z algorytmu RC4. Jeżeli nie, to należałoby wrócić do procesu debugowania kodu wirusa.

2.7. Własne oprogramowanie do odszyfrowania

Zaobserwowane powyżej zachowanie zostało przetestowane w aplikacji Java. W pierwszej kolejności algorytm RSA użyty został do odszyfrowania pierwszych 0x80 bajtów zaszyfrowanego pliku. Odszyfrowywanie przebiegło pomyślnie i pozwoliło odzyskać pierwszy fragment pliku. Następnie spróbowałem odszyfrować pozostałą część pliku za pomocą algorytmu RC4. Udało się odzyskać kolejne 0x1FFF bajtów. Następne bajty były nieprawidłowe. Spróbowałem więc kolejne 0x80 bajtów odszyfrować algorytmem RSA. Udało się — bajty zostały odzyskane. Poprzez powtarzanie tego schematu udało się odzyskać całą zawartość pliku.

W następnej kolejności napisałem oprogramowanie, które przeszukuje dysk twardy w poszukiwaniu plików z rozszerzeniem „crypt" i je deszyfruje. Oprogramowanie z powodzeniem zostało wykorzystane przez znajomego, którego zaatakował CryptXXX w wersji 3.

3. Podsumowanie

3.1. Jak zatem działał CryptXXX w wersji 3

W trakcie odszyfrowywania okazało się, że klucz do algorytmu RC4 jest wielokrotnie używany dla różnych plików. Działo się tak w większości przypadków, w których pola stopki B oraz I były identyczne. Stanowiło to bardzo poważną słabość mechanizmu szyfrowania plików. Poniższa tabela przedstawia klucz szyfrowania dla przykładowych plików posiadających te same wartość w polach stopki B oraz I.

Tabela 5

Wartość pola stopki B oraz I

Pole stopki	Wartość (bajty)
B	0x49 0x58 0x4a 0x28
I	0x02 0x00 0x00 0x40 0x71 0x61 0x00 0xc8 0xbf 0x58 0x00 0x09 0x1a 0xfe 0xb6 0x00 0x00 0x00 0x00 0xc6 0xda 0x2b 0x4e 0x95 0x00 0x00 0x00 0x2a 0x01 0x2c 0x01 0xcc 0x09 0x64 0x00 0xfe 0xff 0xff 0xff 0x02 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x2a 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x2a 0x01 0x00 0x00 0x22 0x8f 0x25 0x00 0xcc 0x09 0x64 0x00 0x74 0x40 0xb0 0x01 0x2a 0x01 0x08 0x02 0xf8 0x8d 0x25 0x00 0xf6 0x0a 0x64 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xc3 0x53 0x5f 0x00 0x03 0x00 0x00 0x00 0x2a 0x01 0x00 0x00 0x48 0x8c 0x25 0x00 0xa8 0xa0 0x35 0x8f 0x9f 0xf3 0xeb 0x7b 0xfc 0x8c 0x25 0x00 0x20 0xf2 0x62 0x00

Tabela 6

Przykładowe pliki posiadające taką samą wartość pola stopki B oraz I

Typ pliku	Wartość pola stopki C	Wartość pola stopki E	Klucz szyfrujący dla algorytmu RC4
ASPX	0x9b	0xdb	XRK2K:[Y”HGdYX$Ri!k>(JXICA06QDp^OoT<y336cY *$ELx^PqNeP%e1&:3~C$+7
ASPX	0x84	0xc4	`p1I)O@tOaFWs@Q?_~CNbz!RR5$c)_26D”SUFY,RXx KRlP_hXyz^Kl8d^(?mQBvf
ASPX	0xad	0xed	`p1I)O@tOaFWs@Q?_~CNbz!RR5$c)_26D”SUFY,RXx KRlP_hXyz^Kl8d^(?mQBvf
ASPX	0xa5	0xe5	`p1I)O@tOaFWs@Q?_~CNbz!RR5$c)_26D”SUFY,RXx KRlP_hXyz^Kl8d^(?mQBvf
ASPX	0x9e	0xd9	O]ls`YWXD&j6HjENP!MJ9TI(3~P*g%>D~X@_Q”1ZBs :U”(8oD[kVc_XW%UM4yP`;
ASPX	0x9a	0xda	O]ls`YWXD&j6HjENP!MJ9TI(3~P*g%>D~X@_Q”1ZBs :U”(8oD[kVc_XW%UM4yP`;
ASPX	0xa9	0xe9	O]ls`YWXD&j6HjENP!MJ9TI(3~P*g%>D~X@_Q”1ZBs :U”(8oD[kVc_XW%UM4yP`;
ASPX	0xa5	0xe5	O]ls`YWXD&j6HjENP!MJ9TI(3~P*g%>D~X@_Q”1ZBs :U”(8oD[kVc_XW%UM4yP`;
ASPX	0xad	0xe5d	lK&?[mxZ%@KMN?FiQ~<F,[L^hvqBFLOfE+7;oQnmk! OltmF%BdJA/nGM`KX;<QJ9
ASPX	0x94	0xd4	lK&?[mxZ%@KMN?FiQ~<F,[L^hvqBFLOfE+7;oQnmk! OltmF%BdJA/nGM`KX;<QJ9
ASPX	0xaf	0xef	lK&?[mxZ%@KMN?FiQ~<F,[L^hvqBFLOfE+7;oQnmk! OltmF%BdJA/nGM`KX;<QJ9
ASPX	0x8f	0xcf	lK&?[mxZ%@KMN?FiQ~<F,[L^hvqBFLOfE+7;oQnmk! OltmF%BdJA/nGM`KX;<QJ9
ASPX	0x9f	0xdf	lK&?[mxZ%@KMN?FiQ~<F,[L^hvqBFLOfE+7;oQnmk! OltmF%BdJA/nGM`KX;<QJ9
ASPX	0x9c	0xdc	_ZB&R<^^)DCL<JGeS/?Yw3)bu.Z*%aE$$@gS_B/qxCZ BsFD<HkF9(R7F!”J,1,Br
ASPX	0x8f	0xcf	_ZB&R<^^)DCL<JGeS/?Yw3)bu.Z*%aE$$@gS_B/qxCZ BsFD<HkF9(R7F!”J,1,Br
ASPX	0xad	0xed	_ZB&R<^^)DCL<JGeS/?Yw3)bu.Z*%aE$$@gS_B/qxCZ BsFD<HkF9(R7F!”J,1,Br
ASPX	0x93	0xd3	_ZB&R<^^)DCL<JGeS/?Yw3)bu.Z*%aE$$@gS_B/qxCZ BsFD<HkF9(R7F!”J,1,Br
ASPX	0xa1	0xe1	_ZB&R<^^)DCL<JGeS/?Yw3)bu.Z*%aE$$@gS_B/qxCZ BsFD<HkF9(R7F!”J,1,Br
ASPX	0x7f	0xb7	oM(PMys%KOCT%>M:w0/E`+yz`F[V!8>&&pB&R5bP L7q*!T,o<cS?W$$*.)]d[Zix

cd. tabeli 6

ASPX	0x91	0xd1	oM(PMys%KOCT%>M:w0/E`+yz`F[V!8>&&pB&R5bP L7q*!T,o<cS?W$$*.)]d[Zix
ASPX	0xa4	0xe4	oM(PMys%KOCT%>M:w0/E`+yz`F[V!8>&&pB&R5bP L7q*!T,o<cS?W$$*.)]d[Zix
ASPX	0xa6	0xe6	oM(PMys%KOCT%>M:w0/E`+yz`F[V!8>&&pB&R5bP L7q*!T,o<cS?W$$*.)]d[Zix

Podsumowując powyższe tabele, na 22 pliki posiadające takie same wartości w polach stopki B oraz I:

— cztery pliki zaszyfrowano kluczem: oM(PMys%KOCT%>M:w0/E`+yz`F[V!8>&&pB&R5bPL7q*!T,o<cS?W$$*.)]d[Zix
— pięć plików zaszyfrowano kluczem: _ZB&R<^^)DCL<JGeS/?Yw3)bu.Z*%aE$$@gS_B/qxCZBsFD<HkF9(R7F!”J,1,Br
— pięć plików zaszyfrowano kluczem: lK&?[mxZ%@KMN?FiQ~<F,[L^hvqBFLOfE+7;oQnmk!OltmF%BdJA/nGM`KX;<QJ9
— cztery pliki zaszyfrowano kluczem: O]ls`YWXD&j6HjENP!MJ9TI(3~P*g%>D~X@_Q”1ZBs:U”(8oD[kVc_XW%UM4yP`;
— trzy pliki zaszyfrowano kluczem: `p1I)O@tOaFWs@Q?_~CNbz!RR5$c)_26D”SUFY,RXxKRlP_hXyz^Kl8d^(?mQBvf
— jeden plik zaszyfrowano kluczem: XRK2K:[Y”HGdYX$Ri!k>(JXICA06QDp^OoT<y336cY*$ELx^PqNeP%e1&:3~C$+7

Reasumując, tylko jeden plik z 22 posiadających taką samą wartość pola stopki B oraz I miał inny klucz. Dla pozostałych użyty klucz powtarzał się co najmniej trzy razy. Dalsza analiza wykazała istnienie innego pliku również zaszyfrowanego kluczem: XRK2K:[Y”HGdYX$Ri!k>(JXICA06QDp^OoT<y336cY*$ELx^PqNeP%e1&:3~C$+7 Plik ten miał inną wartość pola stopki B i I.

Na bazie przeprowadzonej analizy udało się ustalić algorytm szyfrowania plików przez wirusa CryptXXX w wersji 3. Algorytm składa się z następujących kroków:

1. Tworzona jest para kluczy RSA (klucz prywatny, klucz publiczny).
2. Dla każdego szyfrowanego pliku:
   2.1. Tworzony (wykorzystywany istniejący) jest 0x40-bajtowy klucz dla algorytmu RC4. Każdy bajt to jeden ze znaków ASCII. Klucz ten zostaje użyty to inicjalizacji algorytmu RC4.

2.2. Plik dzielony jest na bloki o długości 0x20CF bajtów. Bloki o długości mniejszej niż 0x40 bajtów dopełniane są do 0x40 bajtów. Minimalna dopuszczalna długość przetwarzanego bloku to 0x40 bajtów. Każde pierwsze 0x40 bajtów bloku szyfrowane jest algorytmem RSA. Wszystkie kolejne bajty w bloku szyfrowane są algorytmem RC4. Operacja ta jest powtarzana aż do przetworzenia wszystkich danych w pliku.

2.3. Szyfrowany algorytmem RSA jest klucz z punktu 2.1. Tworzona jest stopka zawierająca 0x118 bajtów. Zaszyfrowany klucz umieszczany jest w polu G stopki. Stopka doklejana jest do końca pliku. Format stopki został opisany w jednym z poprzednich rozdziałów.

Dodatkowo, wirus CryptXXX w wersji 3 robi inne szkodliwe rzeczy, takie jak np. kradzież bitcoinów z dysku użytkownika. Inne czynności nie stanowiły jednak tematu artykułu, więc nie zajmowałem się ich analizą.

3.2. Czy wersja 4 może być lepsza?

Analizowany przeze mnie wirus CryptXXX w wersji 3 posiada wiele wad w utworzonych mechanizmach kryptograficznych. Prawidłowe zastosowanie używanych prymitywów kryptograficznych (RC4, RSA) zmniejszyłoby prawdopodobieństwo odzyskania (bez znajomości klucza) nawet części pliku praktycznie do 0.

Jakie zatem są minimalne wymagane zmiany, aby udoskonalić CryptXXX?

1. Każdy plik należy szyfrować innym kluczem (RC4).
2. Klucz powinien mieć bajty z pełnego zakresu (od 0x00 do 0xFF). Używanie bajtów tylko z zakresu znaków ASCII powoduje osłabienie mocy klucza o 1 bit na każdy 1 bajt.
3. Dane dla RC4 powinny stanowić pojedynczy ciąg bajtów, a nie bloki po 0x1FFF bajtów. Rozpoczynanie szyfrowania zawsze od indeksu 0 dla kolejnych bloków wprowadza dodatkową podatność do mechanizmu szyfrowania.

Jakie są zalecenie dla kolejnych wersji?

1. Zamiana algorytmu RC4 na szyfr blokowy, np. AES.
2. W szyfrowaniu treści pliku należy zrezygnować z algorytmu RSA.
3. Każdy plik należy szyfrować innym kluczem o długości 0x20 bajtów.
4. Dla każdego pliku należy wygenerować inny wektor IV o długości 0x10 bajtów.
5. Całą zawartość pliku należy zaszyfrować algorytmem AES w trybie np. CBC z kluczem o długości 0x20 bajtów i wektorem IV o długości 0x10 bajtów.
6. Klucz użyty w algorytmie AES należy zaszyfrować i dokleić do zaszyfrowanego pliku. Do szyfrowania można użyć algorytmu RSA z dopełnieniem PKCS#1 w wersji 1.5 albo OAEP. Jeżeli zależy nam na większym bezpieczeństwie, należy skorzystać z algorytmów opartych o krzywe eliptyczne.

3.3. Odszyfrowanie plików bez znajomości klucza RSA?

Z uwagi na istniejące poważne błędy w mechanizmie szyfrowania plików związane np. z nieprawidłowym wykorzystaniem prymitywów kryptograficznych (w szczególności RC4), możliwe wydaje się być odszyfrowanie części zawartości zaszyfrowanej algorytmem RC4. Wynika to faktu, że na każde 0x203F bajtów pliku oryginalnego 0x40 zaszyfrowane jest algorytmem RSA, a 0x1FFF algorytmem RC4. Odszyfrowanie częściowe pliku będzie możliwe gdy:

1. Istnieje inny zaszyfrowany tym samym kluczem RC4 plik;
2. Rozmiar zaszyfrowanego pliku z punktu 1 jest nie mniejszy niż rozmiar odszyfrowywanego pliku.

Te warunki powinny być wystarczające do podjęcia próby odszyfrowania części pliku. Aby odszyfrować całą zawartość pliku, potrzebujemy uzyskać dostęp do klucza prywatnego RSA.

3.4. Jak się bronić?

Korzystanie z programów antywirusowych niewiele pomaga, jeżeli jesteśmy jednymi z pierwszych ofiar wirusa. Gdy na dysku pojawią się pliki informujące o sposobie odszyfrowania, jest już za późno na reakcję — dane zostały zaszyfrowane. Jak w takim razie się bronić? Polecam skorzystanie z co najmniej jednej z poniższych rad:

1. Przestać korzystać z systemu Windows.
2. Regularnie tworzyć kopie bezpieczeństwa. Ta metoda powinna pomóc do czasu udoskonalenia wirusa, który może w trybie uśpienia działać przez wiele dni i uszkadzać (szyfrować) dane podczas tworzenia kopii zapasowej. W związku z tym do tworzenia kopii polecam narzędzia działające w oparciu o dystrybucje Linuxa, np. w trybie Live-CD. W takim przypadku nawet wirus w trybie uśpionym nie będzie w stanie uszkodzić tworzonej kopii bezpieczeństwa.
3. Wyłączenie rozszerzonego trybu ochrony w systemie Windows poprzez zastosowanie restrykcyjnych polityk, np. związanych z uruchamianiem plików exe z folderów tymczasowych. Ta metoda powinna pomóc do czasu udoskonalenia wirusa, który z powodzeniem może ominąć wprowadzone polityki bezpieczeństwa.
4. Wyłączenie wtyczek typu Adobe Flash Player w przeglądarkach internetowych, regularna aktualizacja systemu operacyjnego, regularna aktualizacja programu antywirusowego. Ta metoda daje nam tylko częściowe bezpieczeństwo. Nie jesteśmy odporni na nowe wirusy oraz exploity korzystające z nowych luk w systemie operacyjnym i oprogramowaniu dodatkowym.
5. Uważne korzystanie z programu pocztowego oraz zasobów sieci Internet.

Źródło finansowania — działalność statutowa Instytutu Matematyki i Kryptologii WAT.

Artykuł wpłynął do redakcji 28.10.2016 r. Zweryfikowaną wersję po recenzjach otrzymano 14.11.2016 r.

LITERATURA

[1] Serge Vaudenay, *A classical introduction to cryptography*, Applications for Communications Security, Springer, 2006.
[2] www.wikipedia.org, stan aktualny w drugiej połowie maja 2016.
[3] http://bleepingcomputer.com, stan aktualny w drugiej połowie maja 2016.
[4] https://www.hex-rays.com, stan aktualny w drugiej połowie maja 2016.
[5] https://www.virustotal.com, stan aktualny w drugiej połowie maja 2016.
[6] http://www.rohitab.com/apimonitor, stan aktualny w drugiej połowie maja 2016.
[7] http://kpnc.org/idr32/en/, stan aktualny w drugiej połowie maja 2016.

M. GLET

Analysis of cryptographic mechanisms used in ransomware CryptXXX v3

Abstract. The main purpose of this paper was to analysis how malicious software is using cryptographic mechanisms. Reverse engineering were applied in order to discover mechanisms used in ransomware CryptXXX v3. At the end were given some useful advices how to improve CryptXXX.
Keyword: ransomware, software engineering, reverse engineering, RC4, RSA, malicious software
DOI: 10.5604/12345865.1228958