Code Reuse Across Packers and DLL Loaders

Blog.reversinglabs.com/blog/code-reuse-across-packers-and-dll-loaders



Threat Research | April 1, 2021



Blog Author Robert Simmons, Independent malware researcher and threat researcher at ReversingLabs. <u>Read More...</u>



One of the core tenets of computer science is code reuse. Why write something new, when code that already exists can be repurposed or changed slightly and then reused for a different situation. This is no different in the world of malware. SystemBC is a family of remote access trojans used to provide access to the local network of a victim and are a beachhead for lateral movement inside that network ¹. SystemBC has been observed using a variety of packers ². One specific sample ³ has multiple stages of unpacking which eventually lead to an unpacker stub that has nearly complete code overlap with the unpacking stub used in DLL loaders that are found to deliver Ursnif, IceID, DanaBot, Dridex, Zloader, HanciTor, Valak, and a single example of TrickBot. What follows is a detailed analysis of the packed SystemBC sample up to the unpacker stub in question. From that stub a large set of DLL loaders is discovered via YARA hunting. Finally, the generalized process for dumping the payload from these DLLs is shown.

Packed SystemBC Sample

The first stage of the packer in this sample has some extraneous code in addition to the code that performs initial unpacking. Other than this extraneous code, there are a few key points of interest. The first one being a mutex, "guessHi", that is checked for near the start of execution in the main function. This mutex loaded from a hard coded string along with the call to OpenMutexW is shown in Figure 1.

		<u> </u>	
0041632e 0041632f	53 52	push push	ebx {saved_ebx} {0x0} edx {var_8}
00416330	a324794500	mov	dword [data_457924], eax
00416335	e896fbffff	call	mal_415ed0
0041633a	83c404	add	esp, 0x4
0041633d	68083c4300	push	0x433c08 {"guessHi"}
00416342	6a00	push	0x0 {var_c}
00416344	6a01	push	0x1 {var_10}
00416346	8bd8	mov	ebx, eax
00416348	ff152c224300	call	dword [OpenMutexW@IAT]
0041634e	8a4c2408	mo∨	<pre>cl, byte [esp+0x8 {arg1}]</pre>
00416352	b809000000	mov	eax, 0x9
00416357	eb07	jmp	0x416360

Figure 1: Code Block with "guessHi" String and OpenMutexW

Another interesting feature of the file is a cryptographic signature block at the end of the file, but according to the PE header, there is no signature directory content. Because of this missing data in the header, this file is not properly signed. The data directories from the PE header with the empty security directory highlighted is shown in Figure 2.

🔍 Analysis [Data Directories]							
Name	Offset	Size					
Export Directory RVA	00000168	4	0000000				
Export Directory Size	0000016C	4	0000000				
Import Directory RVA	00000170	4	00038924				
Import Directory Size	00000174	4	00000118				
Resource Directory RVA	00000178	4	0005C000				
Resource Directory Size	0000017C	4	0000042E				
Exception Directory RVA	00000180	4	0000000				
Exception Directory Size	00000184	4	0000000				
Security Directory Offset	00000188	4	00000000				
Security Directory Size	0000018C	4	0000000				
Relocation Directory RVA	00000190	4	0000000				
Relocation Directory Size	00000194	4	0000000				
Debug Directory RVA	00000198	4	000324B0				

Figure 2: No Security Directory Referenced in PE Header

However, looking at the very end of the file in a hex editor reveals that there is a blob of DER encoded binary that is clearly a cryptographic signature for the file. Because this DER data is not referenced in the header as shown above, this signature may have been copied from a different file. The start of this signature blob is shown in Figure 3.

)					😸 sys	tembc_c1d31.e	xe		
3EFDC	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	
3F000	481D0000	00020200	30821D35	06092A86	4886F70D	010702A0	821D2630	821D2202	0101310B	H 0.5 *.H &0. "1
3F024	30090605	2B0E0302	1A050030	4C060A2B	06010401	82370201	04A03E30	3C301706	0A2B0601	0 + 0L + .7 .>0<0 +
3F048	04018237	02010F30	09030100	A004A202	80003021	30090605	2B0E0302	1A050004	14C7B2C0	.7 0 0!0 +
3F06C	7C695E45	AD7FFE98	B1FCF62E	38487571	01A08218	7E308205	42308204	2AA00302	01020211	li^E8Huq ~0. B0. *.
3F090	00B4F42E	2C153C90	4FDA64C9	57ED7E10	28300D06	092A8648	86F70D01	010B0500	307C310B	, <.0.d.W.~ (0 *.H 0 1
3F0B4	30090603	55040613	02474231	1B301906	03550408	13124772	65617465	72204D61	6E636865	0 U GB1 0 U Greater Manche
3FØD8	73746572	3110300E	06035504	07130753	616C666F	72643118	30160603	55040A13	0F536563	ster1 0 U Salford1 0 U Sec
3F0FC	7469676F	204C696D	69746564	31243022	06035504	Ø3131B53	65637469	676F2052	53412043	tigo Limited1\$0" U Sectigo RSA C
3F120	6F646520	5369676E	696E6720	4341301E	170D3230	31303230	30303030	30305A17	0D323131	ode Signing CA0 201020000000Z 211
3F144	30323032	33353935	395A3081	84310B30	09060355	04061302	534B310E	300C0603	5504110C	020235959Z01 0 U SK1 0 U
3F168	05383131	30383113	30110603	5504070C	ØA427261	7469736C	61766131	18301606	03550409	811081 0 U Bratislava1 0 U
3F18C	0C0F5AC3	A1687261	646EC3AD	636B6120	36311A30	18060355	040A0C11	4E4F4E4F	2073706F	Zhradncka 61 0 U NONO spo
3F1B0	6C2E2073	20722E6F	2E311A30	18060355	04030C11	4E4F4E4F	2073706F	6C2E2073	20722E6F	l. s r.o.1 0 U NONO spol. s r.o
3F1D4	2E308201	22300D06	092A8648	86F70D01	01010500	0382010F	00308201	0A028201	0100BFB3	.0. "0 *.H 0
3F1F8	7AC3011C	97677862	B30661EA	34101A02	71C4091B	36044500	9200A0FC	05159F8A	28A69551	zgxb. a.4 q. 6 E(Q
3F21C	82153753	A9B4B631	BCFA3B00	199C6AAF	19AB537E	D87BA2DC	4B03C209	09E73316	4B207AFB	. 751; .jS~.{K3 K z.
35240	DQ401024	E7707856	0476CE05	\$2D\$C5E6	3CEAC76R	C42ER7E1	A814EA10	R7E1ADRR	03DED5AR	

Figure 3: Cryptographic Signature Blob

Looking more closely at the content in this signature, a Gmail address is revealed: "draskovicnono[@]gmail[.]com". This email address is highlighted in Figure 4.

)					📙 sys	tembc_c1d31.e	xe		
51510	. LOCLLID	000001 100	01012200	01 0021 33	00001100	0.0.5255	12120101		02030201	I LIBECTIGOTION/ SECTIONS/COUCSIGNING
3F414	43412E63	726C3073	06082B06	01050507	01010467	3065303E	06082B06	01050507	30028632	CA.crl0s + g0e0> + 0.2
3F438	68747470	3A2F2F63	72742E73	65637469	676F2E63	6F6D2F53	65637469	676F5253	41436F64	<pre>http://crt.sectigo.com/SectigoRSACod</pre>
3F45C	65536967	6E696E67	43412E63	72743023	06082B06	01050507	30018617	68747470	3A2F2F6F	eSigningCA.crt0# + 0 . http://o
3F480	6373702E	73656374	69676F2E	636F6D30	22060355	1D11041B	30198117	64726173	6B6F7669	csp.sectigo.com0" U 0 . draskovi
3F4A4	636E6F6E	6F40676D	61696C2E	636F6D30	0D06092A	864886F7	0D01010B	05000382	0101007A	cnono@gmail.com0 *.H z
3F4C8	2D90CD1A	AD28148D	E1CCA5AC	88B3D511	B8A6234C	8710F01F	2DE71D2E	85E840E3	7464AD07	(#L@.td.
3F4EC	76F05465	5B8DB9DB	DB8EF20B	ØB891139	9323B77B	AB85315D	B1D8F9E3	4253F833	C5F0F965	v.Te[BS.3e
35510	EA12125C	35778026	E15D2D5B	RE244149	0D7E324E	57CE4376	Ø8285C21	C1RDE5EE	98FAD162	\5w & 7_F \$A 2NW (y +\1 b

Figure 4: Gmail Address in Signature

Additionally, this same email address can be found using the search feature on the extracted strings list in the Titanium Platform. This string search is shown in Figure 5.



Figure 5: Gmail Address in Extracted Strings

Another important string from this file is the program database string $\frac{4}{2}$. In this file that string is "c:\lawHeart\costforward\pagepushwritten.pdb". One can find this particular string in the A1000 under the CodeViews feature. This string is shown in Figure 6.

CodeViews

Timestamp	Mon Nov 06 11:46:40 2006
Guid	0600769D-8E81-4D7A-AA08-C57285876790
Pdb Path	c:\lawHeart\costforward\pagepushwritten.pdb
Revision	0x0000001

Figure 6: Code Views with Program Database Path

Using any of these strings or by pivoting using the ReversingLabs Hash Algorithm (RHA) reveals one other file that is related to the SystemBC sample being analyzed $\frac{5}{2}$. The results of an RHA pivot is shown in Figure 7.

c1d31fa7484170247564e89c97cc325d1f317 Preview Sample					1aliciou	IS	\sim	All Local TiCle	uud				
Size: 259.3 KB Type: PE / Exe							Time	Threat	Name	Format	<u>Files</u>	Size	
Format: Threat: • Win32.Trojan.SystemBC First scop (cloud) 2020 11 12 02:15 UTC					•	2020-12-20 20:41 UTC	Win32.Trojan.Johnnie	ddba178573653eff198f8421dd41eef7230deda0	PE/Exe	2	263.3 KB	≡	
Last seen (cloud): 2020-11-12 02:15 0 TC			۵	•	2020-12-10 20:38 UTC	Win32.Trojan.System	cld31fa7484170247564e89c97cc325d1f317fb8c8	. PE/Exe	2	259.3 KB	≡		
ЯНП Malicious Suspicion Pivoting ● 2 ● 1		Suspicious	uspicious Known										
		<u>• 1</u>	• 0										

Figure 7: RHA Pivoting Results

However, on closer inspection comparing the bytes of the two files in Hex Fiend 6 , the only major difference is an additional 4 kilobytes of data which is just a second copy of the already existing file info data. No other significant differences are found, so these two files are effectively the same file. This difference is shown in Figure 8.

258024	00000000	00000000	00000000	00000000	00000000	00000000	3E400	6F6D3A61	736D2E76	3122206D	616E6966	65737456	65727369
258048	00000000	00000000	04000400	00000200	10000000	20000080	3E418	6F6E3D22	312E3022	3E0D0A3C	2F617373	656D626C	793E0000
258072	18000000	6000080	00000000	00000000	04000400	00000100	3E430	00000000	00000000	00000000	00000000	00000000	00000000
258096	01000000	38000080	00000000	00000000	04000400	00000100	3E448	00000000	00000000	00000000	00000000	00000000	00000000
258120	09040000	50000000	A8D00500	38030000	E4040000	00000000	3E460	00000000	00000000	00000000	00000000	00000000	00000000
258144	00000000	00000000	04000400	00000100	01000000	78000080	3E478	00000000	00000000	00000000	00000000	00000000	00000000
258168	00000000	00000000	04000400	00000100	09040000	90000000	3E490	00000000	00000000	00000000	00000000	00000000	00000000
258192	E0D30500	56000000	E4040000	00000000	00000000	00000000	3E4A8	00000000	00000000	00000000	00000000	00000000	00000000
258216	38033400	00005600	53005F00	56004500	52005300	49004F00	3E4C0	00000000	00000000	00000000	00000000	00000000	00000000
1: Replace	1 byte at offset 0	0xf6 with 1 byte											
2: Replace	1 byte at offset (0x141 with 1 byte	9										
3: Replace	1 byte at offset (0x179 with 1 byte	e										
4: Replace	2 bytes at offset	0x17c with 2 by	rtes										
5: Delete 3	bytes at offset 0	0x288											
6: Delete 1	byte at offset 0x	(291											
7: Delete 2	bytes at offset 0	x295											
8: Delete 1	byte at offset 0x	299											
9: Delete 2	bytes at offset 0	0x29d											
10: Delete 1	0: Delete 1 byte at offset 0x2ac												
11: Replace	1 byte at offset	0x2af with 11 by	tes										
12 Delete	4 kilobytes at off	set 0x3e436											

Figure 8: Difference in the Two Samples

The function that specifically performs the unpacking routine in this file is found at the address 0x414ED0. This function contains a set of three calls to kernel32.Sleep. These are a basic anti-analysis technique and need to be circumvented to make analysis easier. These three API function calls are shown in Figure 9 with one of them shown in the disassembler view.

Figure 9: API Calls to Kernel32.Sleep

In the debugger, the number of milliseconds for each of these Sleeps can be modified to zero them out. This is shown in Figure 10.



Figure 10: Zero Out Sleeps

After the sleeps are neutralized, the first stage of the unpacker writes the next stub to the .data section of the module in memory. The call into that code is shown in Figure 11.

0041463D 8955 F4 00414640 8845 F4 00414643 83C0 1F 00414646 2845 0C 00414647 0305 04B04300 0041464F A3 04B04300	<pre>mov dword ptr ss:[ebp-C],edx mov eax,dword ptr ss:[ebp-C] add eax,1F sub eax,dword ptr ss:[ebp+C] add eax,dword ptr ds:[43B004] mov dword ptr ds:[43B004] eax</pre>
00414654 FF55 F8 00414657 0FB60D 1CB04300 0041465E 0FB615 09B04300 00414665 2BCA 00414667 81E9 24030000	<pre>call dword ptr ss:[ebp-8] movzx ecx,byte ptr ds.[43801C] movzx edx,byte ptr ds:[438009] sub ecx,edx cmp ecx 324</pre>

Figure 11: Call into Next Stage of Unpacker

As shown in Figure 12, the destination of this call is in the initialized data section of the module with the name .data.

0021/000	OOTE2000	Reserved (00200000)		PRV		-KW
00400000	00001000	systembc_c1d31.exe		IMG	-R	ERWC-
00401000	00031000	".text"	Executable code	IMG	ER	ERWC-
00432000	00009000	".rdata"	Read-only initialized data	IMG	-R	ERWC-
0043B000	00021000	".data"	Initialized data	IMG	-RW	ERWC-
0045c000	00001000	".rsrc"	Resources	IMG	-R	ERWC-
00460000	00035000	Reserved		PRV		-RW
00405000	00000000			DD1/	-BW-C	_DM

Figure 12: Initialized Data Section

The first set of instructions in this next stage is a small loop that decodes the rest of the stub in place. This loop is highlighted in Figure 13.

EIP > •	0043EEA8	81EB 8111C188	sub ebx,88C11181	
•	0043EEAE	E8 0000000	call systembc_c1d31.43EEB3	call \$0
•	0043EEB3	5B	pop ebx	
•	0043EEB4	8D43 30	lea eax.dword ptr ds:[ebx+30]	
	0043EEB7	BF AD1DD78E	mov edi.8ED71DAD	
	0043EEBC	B9 63090000	mov ecx.963	
	0043EEC1	89 FA	mov edx.edi	
	0043EEC3	31DB	xor ebx.ebx	
>	0043EEC5	89CE	mov esi.ecx	
	0043FEC7	83F6 03	and esi.3	
	0043FECA	× 75 0D	ine systembc c1d31.43FED9	
	0043FECC	89 FB	mov ebx.edi	
	0043FECE	66:01DA	add dx bx	
	0043FED1	6BD2_03	imul edx.edx.3	
	0043FED4	C1CA 04	ror edx 4	
	0043FED7	8907	mov edi edx	
	0043FED9	3010	xor byte ptr ds:[eax].d]	
	0043EEDB	40	inc eax	
	0043EEDC	▲ F2 F7	loon systembc c1d31 43FEC5	
	0043EEDE	× F9 C3040000	$\lim_{x \to x} \frac{1}{2} \int \frac{1}$	
	0043FFF3	50	non ehn	
	0043EEE4	× 7A 55	in systembc c1d31 43EE3B	
	00435556	4B	dec eby	
	0043EEE7	BE 29202CAC	mov esi Ac2c2029	
	00435556	012000	add dword ntr ds: [eaviedy*8] ehn	
	0043EEEC	012000	Land amora per us. [eaxTeax o], ebp	

Figure 13: Decoding Loop

After the decoding loop has written out the rest of the stub, the resulting instructions are used to write a YARA rule. The specific instructions used are highlighted in Figure 14.

EIP 0	0043F3A6	E8 00000000	call systembc_c1d31.43F3AB	call \$0
• 0	D043F3AB	5B	pop ebx	
• 0	0043F3AC	81EB FD148000	sub ebx,8014FD	
• 0	0043F3B2	8D83 00108000	<pre>lea eax,dword ptr ds:[ebx+801000]</pre>	
• 0	0043F3B8	8983 CC148000	mov dword ptr ds:[ebx+8014CC].eax	
• 0	0043F3BE	8DB3 39108000	lea esi,dword ptr ds:[ebx+801039]	
• 0	0043F3C4	89B3 A0148000	mov dword ptr ds:[ebx+8014A0].esi	
• 0	0043F3CA	8B46 3C	mov eax.dword ptr ds:[esi+3C]	
• 0	0043F3CD	8983 D0148000	mov dword ptr ds:[ebx+8014D0].eax	
• 0	0043F3D3	8DB3 D8148000	lea esi.dword ptr_ds:[ebx+8014D8]	
• 0	0043F3D9	56	push esi	
• 0	0043F3DA	8DB3 7D108000	lea esi.dword ptr ds:[ebx+80107D]	
• 0	0043F3E0	56	push esi	
• 0	0043F3E1	6A 07	push 7	
• 0	0043F3E3	68 884E0D00	push D4E88	
• 0	0043F3E8	8DBB 9D108000	lea edi.dword ptr ds:[ebx+80109D]	
• 0	0043F3EE	FFD7	call edi	
• 0	0043F3F0	8DB3 F4148000	<pre>lea esi.dword ptr ds:[ebx+8014F4]</pre>	
• 0	0043F3F6	56	push esi	
• 0	0043F3F7	8DB3 99108000	lea esi.dword ptr ds:[ebx+801099]	
	0435350	56	nuch aci	

Figure 14: Decoded Instructions

The process to write the YARA rule starts with writing out the exact bytes of these instructions. Here, just the first few instructions are shown, but in the actual process the whole set of instructions all the way to and including the first function call at 0x43F3EE is used. The example instructions are the following.

E8 0000000 5B 81EB FD148000 8D83 00108000 8983 CC148000

Next, each of the bytes that are specific to locations in this particular file or values that may be unique to this instance of the packer are converted into wildcards and jumps. The bytes that this applies to are shown in red below.

E8 0000000 5B 81EB FD148000 8D83 00108000 8983 CC148000

The resulting byte string with these jumps and wildcards in place is the following.

E8 00 00 00 00 5B 81 EB [4] 8D 83 [4] 89 83 [4] 8D B3 [4] 89 B3 [4] 8B 46 ?? 89 83 [4] 8D B3 [4] 56 8D B3 [4] 56 6A ?? 68 [4] 8D BB [4] FF D7

The full YARA rule using this byte string is provided at the end of this blog.

Related DLLs

Using the YARA rule written using the process above, a retro-hunt is run in the Titanium Platform. The results of this are a large set of hundreds of malicious DLLs that are all packed and utilize the same unpacker stub found in the second stage of the packed SystemBC sample above. These results in the A1000 are shown in Figure 15. This is a very accurate YARA rule in that there are zero false positives found in the result set.

ЯE	VERSIN ABS	IG A1000		D	ashboard	Submissions Searc	h Alerts Yara	a Tags	Feeds Help	v ± v
Тур	pe to searcl	n yara		0						
Sho	owing all 1 resu	lt(s).								
H D	💄 My R	tulesets 🕈 🕇 C	Ordered By Highest Threat							
	🗢 Syste	mBC_Packed	d_Stub	Ali rules 👻	CLOUD	SRETRO 🗱 🔸	211 • 0	• 0	• 45	≡
Filt	264 tered by:	/264	Samples		1MB 10MB 100MB 650MB =650MB	all shared private	e local all	Unknown PE/DII other form	loud cloud	retro
	≜ ∧∘	Match Time	Threat	Name	Rule		Format	Files	Size	
	•	2021-01-03 02:02 UTC	Win32.Trojan.DanaBot	2e6a5a180e757ef69402d20ae21c7dfc5cf96950	System	BC_Packed_Stub	Unknown	1	3.3 MB	≡
	• •	2021-01-03 02:02 UTC	Win32.Trojan.Emotet	97129b734a451d04eaf0cdec99efb9009cf4e7c0	System	BC_Packed_Stub	Unknown	1	476.5 KB	=
	• •	2021-01-03 02:02 UTC	Win32.Trojan.IcedID	0b92f2b52aa49d3ea90f1c1dfc47bde2533f35ed	System	BC_Packed_Stub	Unknown	1	144 KB	=
	• •	2021-01-03 02:02 UTC	Win32.Trojan.Johnnie	34dc41ab625df260f2a46bb9f3e0cd984d1c370b	System	BC_Packed_Stub	Unknown	1	215.5 KB	≡
	• •	2021-01-03 02:02 UTC	Win32.Backdoor.Lotoor	436e253dda5c90d56bda8797d1713437ebb07310) System	BC_Packed_Stub	Unknown	1	362.5 KB	=

Figure 15: Retro-hunt Results

This is a moderately large set of files, so unpacking each one to determine what malware family is being delivered would be time consuming. Therefore a strategy for grouping the files into clusters which can then have representative files analyzed is a good idea. One effective strategy for this particular data set is to group the files by import hash. Figure 16 shows all the DLLs that share an import hash in descending size of the groups, but excluding single member groups.



Figure 16: Files Grouped by Import Hash

Each cluster can then be examined to determine if the members of a cluster are in fact all delivering the same unpacked payload. Figure 17 shows one cluster that has two different detection names according to automation. The fuzzy hash, ssdeep, is also shown as a sanity check to make sure that the structure of the files in the cluster are nearly the same.

Time 🗸	Import Hash	File Size 🗸	ReversingLabs.Threat Name	ssdeep	Filename
Oct 1, 2020 @ 22:21:23.000	8b9db6a8971f1f6955399a095f4baa4b	355,842	Win32.Worm.Cridex	6144:3iPezoVfjCMoEeHqHY2C4MlKNWsxCbVX31CQV9kfYU:7o5CMoEeHq04ZfxCbVH1fHQYU	3ecff41c2026ee2ab744bf2ecc10f6f032c9c645e4c6ac5e33312b977b2bff0e
Oct 15, 2020 @ 03:04:00.000	8b9db6a8971f1f6955399a095f4baa4b	355,840	Win32.Worm.Cridex	6144:3iPezyVfjCMoEeHqHY2C4MlKNWsxCbVX31CQV9kfY:7y5CMoEeHq04ZfxCbVH1fHQY	91645a7414c2e7082118364980d22f4062c14cbfc786f07e59f4fcbc96928ed3
Jul 30, 2020 @ 11:16:06.000	8b9db6a8971f1f6955399a095f4baa4b	355,840	Win32.Trojan.IcedID	6144:3iPez8VfjCMoEeHqHY2C4MlKNWsxCbVX31CQV9kfY:705CMoEeHq04ZfxCbVH1fHQY	5c164db3a3028d0b83b94f4938a8c9d71f51d731c23346565ede67620786c56d
Jul 30, 2020 @ 11:15:09.000	8b9db6a8971f1f6955399a095f4baa4b	355,840	Win32.Trojan.IcedID	6144:3iPezNVfjCMoEeHqHY2C4MlKNWsxCbVX31CQV9kfY:7N5CMoEeHq04ZfxCbVH1fHQY	f2771e7307f0a2ce36ff0fd0d7c5f6e810c1610ec6dabbf92e472db071fadf8b
Jul 30, 2020 @ 11:15:08.000	8b9db6a8971f1f6955399a095f4baa4b	355,840	Win32.Trojan.IcedID	6144:3iPezpVfjCMoEeHqHY2C4MlKNWsxCbVX31CQV9kfY:7p5CMoEeHq04ZfxCbVH1fHQY	bdfe12d66d23d2dc816e53e3c6529ca8bad831593d713460d1e95dcbae6ef237
Jul 30, 2020 @ 11:15:07.000	8b9db6a8971f1f6955399a095f4baa4b	355,840	Win32.Trojan.IcedID	6144:3iPezhVfjCMoEeHqHY2C4MlKNWsxCbVX31CQV9kfY:7h5CMoEeHq04ZfxCbVH1fHQY	d482df2c7867179901e983eea0e8aae0801e3ac547bcfa9a93470b6a37005473
Jul 30, 2020 @ 11:15:06.000	8b9db6a8971f1f6955399a095f4baa4b	355,840	Win32.Trojan.IcedID	6144:31PezNVfjCMoEeHqHY2C4M1KNWsxCbVX31CQV9kfY:7N5CMoEeHq04ZfxCbVH1fHQY	f974fc781325113d7c24d37a7d778cd0d841ff9702f5d37612b4666a6607de13

Figure 17: Cluster of Files Sharing One Import Hash

Two Basic Flavors

Among all these files, there are two basic flavors of packing. The payload binary is written to allocated memory in all cases, but in one case this payload is uncompressed and the other is compressed. The uncompressed payload can simply be dumped directly and then analyzed. However, in the case of the compressed payload, one needs to determine the compression

algorithm and then decompress the data before analyzing the resulting binary. This can be done a few different ways. First, the unpacker itself will decompress the binary and overwrite the original DLL's module. After that, the DLL can be dumped and analyzed. Alternatively, one can, as noted earlier, determine the algorithm and decompress the data. However, there is an easier, more straightforward method using the Titanium Platform. The first step is to open the DLL in x64dbg and run the executable up to the entry point. From there, one sets a breakpoint at the return instruction in kernel32.VirtualAlloc. This breakpoint is shown in Figure 18.

7553F150	8BFF	mov edi,edi	VirtualAlloc
7553F152	55	push ebp	
7553F153	8BEC	mov ebp.esp	
7553F155	51	push ecx	ecx:EntryPoint
7553F156	51	push ecx	ecx:EntryPoint
7553F157	8B45 0C	mov eax.dword ptr ss:[ebp+C]	Lohn Cl + "MZE"
7553F15A	8945 F8	mov dword ntr ss: [ebn-8] eax	Lebn-81:EntryPoint
7553F15D	8845 08	mov eax dword ntr ss: ehn+8	[ebp+8]:EntryPoint
7553F160	8945 FC	mov dword ntr ss lehn-41 eax	
7553F163	56	nush esi	Tebb-41: MZL
7553F164	8500	test eav eav	
75535166	× 74 0C	ie kernelhace 7553E174	
75535168	3805 98565575	$\int \frac{1}{2} \int $	
75535165	0582 61870300	th kernelhace 75577805	
75535174	• 0F82 01870300	push dword ntr. ss: ohn 14	
75525177		mov any dword ptr ss. epp+14	
75535177	2256	lillov eax, uword ptr ss.[epp+to]	
7553F17A	33F0 C0	and easy EFFEECO	
7553F17C	83E0 C0	and eax, FFFFFC0	
/ 333F1/F		push eax	False 97 - Free mondate
7553F180	8D45 F8	Tea eax, dword ptr ss:[ebp-8]	[ebp-8]:EntryPoint
75535103	50	push eax	
75535184		push est	
7553F185	8045 FC	lea eax, dword ptr ss:[ebp-4]	[ebp-4]:"MZ L "
7553F188	50	push eax	
7553F189	6A FF	push FFFFFFF	
7553F18B	FF15 <u>38875F75</u>	call dword ptr ds:[<&ZwAllocateVirtualMemory>]	
7553F191	8500	test eax,eax	
7553F193	✓ /8_0A	Js kernelbase.7553F19F	
7553F195	8B/5 FC	mov esi,dword ptr ss:[ebp-4]	Геbb-41:"MZ L "
7553F198	8BC6	mov eax,esi	
7553F19A	5E	pop esi	
7553F19B	C9	leave	
7553F19C	C2 1000	ret 10	
7553F19F	8BC8	mov ecx,eax	ecx:EntryPoint
7553F1A1	E8 6A29FEFF	call kernelbase./5521B10	
7553F1A6	▲ EB F0	jmp_kernelbase./553F198	
75536148	CC	int?	

Figure 18: Breakpoint on VirtualAlloc

Once set, run the file until that breakpoint is reached. When execution is on this return instruction, observe the address of the newly allocated memory in the EAX register. An example of this is shown in Figure 19 with the address of the newly allocated memory highlighted.

🖾 CPU 🗾	Log 📋 No	otes 🛛 📍 Breakp	oints	Memory Map	🗐 Call Stack	न SEH	Script	🐏 Syı	mbols	<>> Source	₽ Re
	7553F195 7553F198 7553F19A 7553F19B 7553F19F 7553F146 7553F146 7553F148 7553F148 7553F148 7553F148	8B75 FC 8BC6 5E C9 C2 1000 8BC8 E8 6A29FEFI CC CC CC CC CC	-	<pre>mov esi,dw mov eax,es pop esi leave ret 10 mov ecx,ea call kernel int3 int3 int3 int3 int3</pre>	vord ptr ss:[e ;i Phase,75521B10 base,7553F198	bp-4]	/	^ ~	EAX EL EDX EBP ESP < Default 1: [es 2: [es	00350000 71426F2E 7BD80000 0019EF54 0019EF54 0019EF24 (stdcall) p+4] 000000	&L " 00
.text:7553F19	OC kernelbase	e.dll:\$10F19C #	10E59C						3: [es 4: [es 5: [es <	p+C] 000030 p+10] 00000 p+14] 00000	00 040 A66
🛄 Dump 1	🚚 Dump 2	🚛 Dump 3	🛄 Dump 4	🚛 Dump 5	虁 Watch 1	[x=] Locals	🖉 Struct				
Address Icx 00350000 00 00350010 00 00350020 00 00350030 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	0 00 00 00 0 00 00 00 0 00 00 00 0 00 00	00 00 00 00 00 00 00 00 00 00 00 00	ASCII	· · · · · · · · · · · · · · · · · · ·					

Figure 19: Newly Allocated Memory

Once execution has arrived at this point, set one more breakpoint on VirtualProtect. The reason for waiting until execution is deep into the unpacker stubs before setting the breakpoint on VirtualProtect is that often there are benign calls to VirtualProtect that one would need to pass over before even getting to the initial VirtualAlloc, and that can get tedious.

For each call to VirtualAlloc, follow that new memory address in x64dbg's dump then run to the next call to VirtualAlloc. Each time examine the new data that is written to the allocated memory. This is where eventually the payload binary is written to. In the first flavor of unpacking mentioned above, it will be crystal clear when the MZ magic number along with the DOS stub appear in the allocated memory. However, the second flavor where the payload is compressed will also be quite recognizable, if not a bit garbled. An example of this compressed flavor is shown in Figure 20.

 77447C6D CC 77447C6E CC 77447C6F CC 77447C70 88 FF 77447C72 55 77447C73 88 EC 77447C75 5D 77447C76 ~ FF25 F8124B77 77447C7C CC 77447C7D CC 77447C7F CC 77447C7F CC 77447C7F CC 						ir ir pu mc pc jn ir ir ir ir	it3 it3 it3 ish e ip eb ip dw it3 it3 it3 it3	li,ec bp p,es p ord	di sp ptr	ds:[<mark><&Vir</mark>	'tualPr	rotect>]	VirtualPi JMP.&Vir	rotect tualProtect
edi=f974f.718EE050 .text:77447C70 kernel32.dll:\$17C70 #8C70 <virtualprotect></virtualprotect>														
🚛 Dump	1 🛄 Du	imp 2	🚛 Du	mp 3	💷 Di	ump 4		Dun	np 5	🤴 Watch	n 1 [x=] Locals	👂 Struct	
Address	Hex									ASCII				
00380000 00380010 00380020 00380030 00380040 00380050 00380060	4D 38 5A 01 40 C2 21 B8 01 67 61 6D 5F 98 69 0A 24 4C 3E 6A 4D	90 38 15 C6 4C C0 87 63 06 44 44 4F 30 09	03 66 D8 09 0A 54 47 6E 4F 7E 01 14 11 78	02 04 1C 0E 68 69 1F 4F 53 03 27 F9 17 48	09 71 1F BA 73 20 74 E7 6D 6F 0B 75 70 AB	FF 8 F8 0 OE 7 62 6 64 6 49 A AB 1	1 B8 0 B4 0 72 5 AF 5 2E A 58 4 C7	C2 09 6F CF 0D 04 C6	91 CD 67 75 89 E3 39	M8Z.8.f .@Â. &Ø ! ,.LÀ.This gam.cGn.Ot i.DO~S.m .\$LDO'ù. >iMOx.H¤	qÿ. Â ø. çbe ï ode. uIªX. o««.CÆ			-

Figure 20: Compressed Payload

This is where the Titanium Platform comes in handy. Just dump this memory to a file and upload it to the A1000. After it has been analyzed, navigate to the extracted files and drill into the extracted file until the payload DLL is revealed. This extracted DLL is shown in Figure 21.



Figure 21: Decompressed Payload

Analysis of all the clusters of DLLs in the retro-hunt results in this manner, along with all the import hashes with only one file, reveals that this unpacker code has been used to deliver many malware families over the past year. These families include Ursnif, IceID, DanaBot, Dridex, Zloader, HanciTor, Valak, and a single example of TrickBot. A full list of file hashes in clusters by payload malware family is provided below.

IOCs

SystemBC Samples

c1d31fa7484170247564e89c97cc325d1f317fb8c8efe50e4d126c7881adf499

6afe08f542426b9662b84907d35870e9714c2755e1da95ed42db33a37aaf33b9

Mutex

guessHi

Email Address

draskovicnono[@]gmail[.]com

PDB Path

c:\lawHeart\costforward\pagepushwritten.pdb

YARA Rule

```
private rule WindowsPE
ł
    condition:
        uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550
}
rule Unpacker Stub
{
    meta:
        author = "Malware Utkonos"
        date = "2020-12-30"
        description = "First bytes in decoded unpacker stub."
        exemplar = "c1d31fa7484170247564e89c97cc325d1f317fb8c8efe50e4d126c7881adf499"
    strings:
        $a = { E8 00 00 00 00 5B 81 EB [4] 8D 83 [4] 89 83 [4] 8D B3 [4] 89 B3 [4]
               8B 46 ?? 89 83 [4] 8D B3 [4] 56 8D B3 [4] 56 6A ?? 68 [4] 8D BB [4] FF D7 }
    condition:
        WindowsPE and $a
}
```

A full list of file hashes in clusters by payload malware family is provided here.

References: ¹<u>https://malpedia.caad.fkie.fraunhofer.de/details/win.systembc</u> ²<u>https://news.sophos.com/en-us/2020/12/16/systembc/</u> ³c1d31fa7484170247564e89c97cc325d1f317fb8c8efe50e4d126c7881adf499 ⁴<u>https://en.wikipedia.org/wiki/Program_database</u> ⁵6afe08f542426b9662b84907d35870e9714c2755e1da95ed42db33a37aaf33b9 ⁶<u>https://ridiculousfish.com/hexfiend/</u>

MORE BLOG ARTICLES