Carving the IcedId - Part 3

blog.techevo.uk/analysis/binary/2024/03/17/carving-the-icedid-part-3.html

techevo

March 17, 2024

Welcome back to this series, analysing IcedId malware artefacts.

This is part 3 in the series, you can check out <u>part 1</u> and <u>part 2</u> to follow along from the beginning.

This post will focus on analysing a DLL file that was downloaded using a PowerShell script analysed in previously in <u>part 2</u>.

The data for this case was published by <u>@malware_traffic</u> over at **Malware Traffic Analysis**¹. You can download all the samples from this case from <u>here</u>.

This analysis has really stretched my learning regarding unpacking, it has by far been the most challenging and rewarding sample I've come across to date. If there are any errors that you spot, I'd really welcome the feedback to understand better how this sample works.

In order to make this walk through as accessible as possible, I will once again be storing artefacts and output in a GitHub repository <u>here</u>.

The GitHub repository contains the extracted shellcode as seen in the various commands for your own experimentation, as well as the final payload.

TL;DR

This post is fairly detailed and as a result quite long. A quick overview of how the sample executes is listed below to provide some quick insight. If you want a more guided tour of the execution and other interesting observations, skip this section.

- 1. rundll32.exe executes a export on the dll.
- 2. The DLL routine allocates some memory and copies and unpacks data into shellcode from the .reloc section of the DLL.
- 3. The unpacking consists of a 4 byte XOR as well as the supplied string on the command line, for various stages.
- 4. The unpacked shellcode is patched with function addresses and creates some syscall stubs to avoid ntdll.dll hooks.

- 5. The rundll32.exe process opens svchost.exe and injects a payload using shared mapped views of sections and NtQueueUserThread
- 6. The svchost.exe process further unpacks a PE file which is then injected into memory at a fixed location.
- 7. The injected payload is then executed.
- 8. The final payload can be downloaded from the Bazaar or GitHub

In the previous post, a PowerShell script was used to download a DLL named r.dll from a compromised WordPress instance.

Part of the script appended varying amounts of bytes to the file, ensuring the cryptographic hash changes with each download. You can find a copy of the DLL file on the Malware Bazaar, <u>here</u> The SHA1 hash for the copy we will be looking at in this post is: 1c6e76af95f2a17b8e518965d62b3c9d7ecba6d5

For this explanation of the malware delivery, both static and dynamic analysis will be used in conjunction.

For static analysis I am using **radare2**² and for dynamic analysis **x64dbg**³ both are freely available.

Binary File Triage

From the Powershell script we know there must be an export named vcab, we can use a **radare2** one-liner to show the various exports.

\$ r2 -c 'iE' r.dll

[Exp nth dema	paddr angled	vaddr	bind	type	size	lib	name
1	0x00000420	0x814e361020	GLOBAL	FUNC	Θ	msys-edit-0.dll	t_gcc_deregister_frame
2	0x00000400	0x814e361000	GLOBAL	FUNC	0	msys-edit-0.dll	t_gcc_register_frame
3	0x000151e0	0x814e375de0	GLOBAL	FUNC	0	msys-edit-0.dll	tel_fn_complete
4	0x000192c0	0x814e379ec0	GLOBAL	FUNC	0	msys-edit-0.dll	trl_abort_internal
5	0x00026338	0x814e38a138	GLOBAL	FUNC	0	<pre>msys-edit-0.dll</pre>	
trl_	_print_compl	letions_horizo	ontally				
6	0x000192f0	0x814e379ef0	GLOBAL	FUNC	0	msys-edit-0.dll	<pre>trl_qsort_string_compare</pre>
7	0x00016bf0	0x814e3777f0	GLOBAL	FUNC	0	msys-edit-0.dll	tdd_history
8	0x000169a0	0x814e3775a0	GLOBAL	FUNC	0	msys-edit-0.dll	tppend_history
9	0x00000880	0x814e361480	GLOBAL	FUNC	0	msys-edit-0.dll	tnext_word
10	0×00000800	0x814e361400	GLOBAL	FUNC	0	msys-edit-0.dll	tprev_word
[TF	RUNCATED]						
152	0x000177a0	0x814e3783a0	GLOBAL	FUNC	0	msys-edit-0.dll	tistory_expand
[TF	RUNCATED]						
430	0x00016fb0	0x814e377bb0	GLOBAL	FUNC	0	msys-edit-0.dll	there_history
431	0x000177a0	0x814e3783a0	GLOBAL	FUNC	Θ	msys-edit-0.dll	vcab

The above output is truncated, however you can see there are 431 exports on this DLL. The final export listed is the vcab export we already know about. You can find a full output of the command in the GitHub repository for this blog posts, <u>here</u>.

As well as the export names, the virtual addresses are also quite interesting. Looking at the export tistory_expand, ordinal 152, we can see it has the same virtual address as the vcab export.

Given the large amount of exports I believe this is likely a legitimate DLL file that has been modified with some additional functionality. Searching for the DLL name msys-edit-0.dll also shows this is possibly related to the msys2 project.

Since we've looked at **Exports**, lets look at **Imports**, using the following command.

\$ r2 -c 'ii' r.dll [Imports] nth vaddr bind type lib name

- -

. -

^{1 0}x814e391860 NONE FUNC KERNEL32.dll GetModuleHandleA

One import is not a lot to go off for understanding the functionality. The lack of imports is also quite suspicious, and something that indicates this DLL should be investigated further.

Statically analysing the DLL functions proved a little harder than expected. Forcing **Ghidra** to decompile the bytes was possible, but readability was not amazing.

To explore this sample further, I will be combining both static and dynamic analysis techniques.

Debugger Setup

For the dynamic analysis parts of this you will require some working knowledge of **x64dbg**. Primarily around setting breakpoints, although the commands are provided, just knowing what a breakpoint is and how to set it should be enough. If something isn't clear feel free to reach out and ask!

As well as the vcab entry point being supplied on the command line, a flag /k and string parameter were also provided as shown below.

rundll32 r.dll, vcab /k chokopai723

To look into the execution of the DLL I'll be using **x64dbg**. It is possible to use the **x64dbg** DLL host binary, however for this analysis, debugging will be done with rundll32.exe executable in order to mimic the execution environment precisely.

Once you have opened the binary C:\Windows\System32\rundll32.exe with **x64dbg** change the command line to include the additional parameters as shown in *Figure 1*.

🗊 Change Command Line	×
C:\Windows\System32\rundll32.exe" C:\Users\malware\Desktop\r.dll vcab /k chokopai72	!3
ОК	Cancel

Figure 1: x64dbg - Additional command line parameters.

I find it helpful when analysing a new sample to setup breakpoints on DLL loads, which helpfully is a built in feature.

Navigating to **Options** and then **Preferences** you can enable the settings User DLL Load and System DLL Load.

Execute until the r.dll is loaded and then issuing the following command in will set a breakpoint on the vcab entry point.

bp r.vcab

We should also set some breakpoints for interesting API calls before starting, using the following commands. These API's specifically have been selected because VirtualAlloc is common in packed samples to aid in unpacking, and since the number of Imports was limited to a single Kernel32.dll library, there is a chance the sample will attempt to load more modules manually.

bp VirtualAlloc
bp LoadLibraryA

Command Line Validity Check

The first routine to highlight during this walk through is a check that the /k was supplied on the command line. Setting a breakpoint at $0 \times 814e378887$ and viewing the sample statically we can see the ASCII characters $0 \times 6B$ and $0 \times 2F$ being moved into a memory region, as shown in *Figure 2*.



Figure 2: radare2 - r.dll command line check routine.

An instruction at $0 \times 0814E378AAB$ then copies these two bytes into the RDX register. The command line string is then iterated over scanning for the <u>/k</u> flag being present. If its not then the execution flow exits.

Memory Copy Routine

The next routine of interest is located at virtual address 0x0814E378B26.

This routine is used throughout this portion of the loader to essentially move bytes from one location to another, much like the $memcpy^4$ function.

The function prototype for memcpy is shown below, and this is also used by the routine within the sample.

In x86_64 assembly the registers RCX, RDX and R8 are used to store the destination , source and count (size) parameters.

```
void *memcpy(
    void *dest,
    const void *src,
    size_t count
);
```

Although the function is located at 0x0814E378B26, the primary loop that moves data between source and destination can be seen at 0x814E378B71. The disassembly for this routine is shown in *Figure 3* below. The register RDX is used as an index to then increment as it loops through the bytes being copied.



Figure 3: radare2 - IcedId memcpy shellcode routine.

Setting a breakpoint at 0x0814E378B26 will allow us to inspect the various bytes being moved around.

bp 0x0814E378B26

If we allow execution until the memory copy routine breakpoint, we first see a call to copy the string chokopai723 from one area on the stack to another stack based memory location.

Figure 4 shows the source address 0x0F340F0F44A, destination 0x0F340F0F5B0 and the number of bytes 0xB

RAX	00000000000000000						
RBX	0000000000000000						
RCX	000000F340F0F5B0						
RDX	000000F340F0F44A	"chokopai723"					
RBP	0000000000000019						
RSP	000000F340F0F3D8						
RSI	000000F340F0F5B0						
RDI	000000F340F0F447	"/k chokopai723"					
R8	0000000000000000						
Figure 4: x64dbg - Memory copy routine register usage							

Allowing the execution to proceed, the debugger will *break* at a call to VirtualAlloc⁵. If we examine the supplied parameters we can mock-up a call to VirtualAlloc with the following values.

VirtualAlloc(NULL, 0xE27, 0x3000, 0x4);

Converting some of the inputs to their constants⁵ $\frac{6}{2}$ makes it a little easier to understand what is happening.

VirtualAlloc(NULL, 0xE27, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);

Here we can see at least 0xE27 (3623) bytes of memory is being requested, to be committed and reserved, with the page protection of Read and Write.

The value returned in the EAX register is going to be one to keep an eye on. This value is the address of an allocated region of memory. As this value changes from execution to execution I will refer to this as "memory region 1" throughout this post.

This allocated region of memory is then populated using the malware's implementation of memory already covered (0x0814E378B26). The routine is called a total of 3 times, the total number of bytes copied matches the requested region size of 0xE27 (3623) bytes.

Each time, the source of the data is located in the .reloc section of the DLL.

The table below describes the source virtual address, the file physical offset, and number of bytes copied.

Source Virtual Address	File Offset	Byte Count
0x0814E3949E5	0x2B9E5	0x4A (74)
0x0814E394A2F	0x2BA2F	0x18F (399)

Source Virtual Address File Offset Byte Count

0x0814E394BBE 0x2BBBE 0xC4E (3150)

Table 1: Virtual Address and file offset mappings

The file offset can be calculated using the source address seen in the debugger, minus the virtual address of the section (.reloc). Then identifying the physical address of the section within the PE file using the headers, and adding the difference back.

Using **x64dbg**'s memory map tab you can save this memory region to a file, you can find a copy of the file rundll32_memory_region_1.bin in the Github repository <u>here</u>.

Either using the offsets identified or by dumping the memory region, we can examine the data copied in more detail. Data mysteriously copied into un-backed memory region has potential to be shellcode.

We can test this theory by attempting to disassemble the bytes in using this **radare2** oneliner.

Figure 5 shows the interpretation of the bytes as assembly. It appears to be junk as there is no obvious flow of execution present.

```
$ r2 -AA -c 'pd' rundll32_memory_region_1.bin
```

<pre></pre>	arg1, int64_t	arg3, int64_t arg7);
; arg int64_t a	rg1 @ rdi	
; arg int64_t a	rg3 @ rdx	
; arg int64_t a	rg7 @ xmm0	
0x0000001	39 db	cmp ebx, ebx
0x0000003	4855	push rbp
0x0000005	5e	pop rsi
0x0000006	6f	outsd dx, dword [rsi]
0x0000007	3316	xor edx, dword [rsi] ; arg3
0x0000009	fb	sti
0x000000a	84c9	test cl, cl
0x000000c	29 fb	sub ebx, edi ; arg1
0x000000e	8e 433e	mov es, word [rbx + 0x3e]
0x0000011	f734c0	div dword [rax + rax*8]
0x0000014	9f	lahf
0x0000015	3b44e09f	cmp eax, dword [rax + riz*8 - 0x61]
0x0000019	3b44d89f	cmp eax, dword [rax + rbx*8 - 0x61]
0x000001d	3b44d05f	cmp eax, dword [rax + rdx*8 + 0x5f]
0x0000021	f623	mul byte [rbx]
0x00000023	309f3b54c06c	xor byte [rdi + 0x6cc0543b], bl ; [0x6cc0543b:1]=255 ; arg1
0x0000029	b207	mov dl, 7
0x000002b	00 c 6	add dh, al
0x000002d	fb	sti
0x0000002e	8e 4b6e	mov cs, word [rbx + 0x6e]
0x0000031	fb	sti
0x0000032	8a 4b ce	mov cl, byte [rbx - 0x32]
0x00000035	fb	sti
0x0000036	8e 43 ce	mov es, word [rbx – 0x32]
0x00000039	fa	cli
0x000003a	bfa5 72 11a5	mov edi, 0xa51172a5
0x000003f	a1d6b207ff06.	movabs eax, dword [0xc484fa06ff07b2d6]

Figure 5: radare2 - Disassembly view of allocated memory region #1

It's a good idea at this point to set an **Access** breakpoint on the memory region to see if there are any routines that may transform it in some way.

Executing the process again will break when the process attempts to **access** an address within the allocated region of memory.

The cause of this is an XOR operation at 0x0814E3784E8 as shown in *Figure 6*.

					_						
000000814E3784E4		8BC2	mov eax,edx		~	uid	EDU				
000000814E3784E6	V	EB 12	jmp r.814E3784FA			HTU	e Fro				
000000814E3784E8		3041 FF	xor byte ptr ds:[rcx-1],al	xor routine memory region 1		AX 0000000000000000	'ö'				
000000814E3784EB		3BD6	cmp edx,esi			BX 0000000000000000	•				
000000814E3784ED	^	72 F5	ib r.814E3784E4		B	CX 000001A200C20001					
000000814E3784EF	~	EB 76	jmp r.814E378567		R	DX 00000000000000000					
000000814E3784F1		FFC2	inc edx		R	BP 000000299508FB79					
000000814E3784F3		0FB64438 2C	<pre>movzx eax,byte ptr ds:[rax+rdi+2C]</pre>	byte ptr ds:[rax+rdi*1+2C]:tistory_expand+90A	R	SP 000000299508FAE0					
000000814E3784F8	^	EB EE	imp r.814E3784E8		R	SI 00000000000022/	L'3'				
000000814E3784FA		48:8D49 01	lea rcx,qword ptr ds:[rcx+1]		н	DI 000000814E3/8BA8	r.000000814E3/8BA8				
			- Flaure 6' x64aba - 2	XOR operation memory red	ดเด	n #1					

The screenshot in *Figure 6* above and in *Figure 7* below show this XOR taking place both from a dynamic and static perspective.

Г.	0x814e3784e8	3041ff	xor byte [rcx - 1], al
	0x814e3784eb	3bd6	cmp edx, esi
	0x814e3784ed	72 f5	jb 0x814e3784e4
	0x814e3784ef	eb 76	jmp 0x814e378567
	; CODE XREF from	<pre>sym.msys_edit_0</pre>	.dll_tistory_expand @ 0x814e378501(x)
	0x814e3784f1	ffc2	inc edx
	0x814e3784f3	0fb6 44382c	movzx eax, byte [rax + rdi + 0x2c]
	0x814e3784f8	ebee	jmp 0x814e3784e8
	; CODE XREF from	sym.msys_edit_0	.dll_tistory_expand @ 0x814e3784e6(x)
	0x814e3784fa	488d4901	lea rcx, [rcx + 1]
	0x814e3784fe	83e003	and eax, 3
	0x814e378501	ebee	jmp 0x814e3784f1

Figure 7: radare2 - XOR operation memory region #1

The AL register in this case is the lower 8 bytes of the EAX register.

The register pane on the right in *Figure 7* shows this to contain the value 0xD6.

The address the operation is being carried out on in this case is shows as ds: [rcx-1] which if we take a look at the value in the RCX register should contain the address of the second byte within memory region 1, the -1 them refers to the first byte of our mystery data.

If we step through the next few operations hitting the XOR instruction we eventually see the same 4 bytes rotating through the AL register: 0xD6B20700

This raises an interesting question, where are these bytes coming from and can locate them within the DLL file?

We know from observing the routine, that the bytes used for the XOR key is being set in the EAX (AL) register.

Within the screen shot shown in *Figure 7* you may notice the operation at 0x0814E3784F3, also shown below.

```
movzx eax,byte ptr ds:[rax+rdi+2C]
```

This is the operation setting the value of the EAX/AL register prior to the XOR operation. If we follow the address calculated at RAX + RDI + 2C in a dump we can see the 4 bytes at the address $0 \times 0814E378BD4$ or file offset $0 \times 17FD4$, as shown in *Figure 8*.

00017F50	сÞ	8B	CA	66	ЗB	СО	74	06	49	8B	C2	СЗ	EΒ	E2	48	OB	Á< Éf; Àt. I< ÂĂëâH.
00017F60	C8	49	OB	C8	EВ	C2	4C	8B	D1	4E	8D	OC	02	ЗA	ED	74	ĚI.ÈëÂL<ÑN:ít
00017F70	DD	48	FF	C2	49	ЗB	D1	75	02	EΒ	$\mathtt{D}\mathtt{D}$	OF	Β6	02	88	04	ÝHÿÂI;Ñu.ëÝ.¶.^.
00017F80	11	EΒ	EE	48	83	C2	04	49	ЗB	D1	75	02	EΒ	A4	8B	02	.ëîHfÂ.I;Ñu.ë¤<.
00017F90	89	04	11	ΕB	EE	ΕB	00	65	48	8B	04	25	30	00	00	00	‱. <u>.ëîë.</u> eH<.%O
00017FA0	C3	4B	69	6E	69	74	00	03	A2	8B	01	00	E5	49	03	00	Äkinit .¢<åI
00017FB0	4A	00	00	00	ΖF	4A	03	00	8F	01	00	00	2 A	34	03	00	J/J*4
00017FC0	BB	15	00	00	ΒE	4B	03	00	4E	0C	00	00	OC	58	03	00	»¾KNX
00017FD0	AD	1E	00	00	D6	В2	07	00	C8	49	89	C4	E8	2F	36	00	Ö²ÈI‱Äè/6.
00017FE0	00	4C	89	F9	48	89	45	DO	E8	23	36	00	00	41	OF	Β6	.L‰ùH‰EĐè#6A.¶
00017FF0	1E	48	89	C7	84	DB	OF	84	26	05	00	00	48	8B	55	DO	.H‱Ç <i>"</i> Û. <i>"</i> હH< UÐ
00018000	45	OF	Β6	OF	88	5D	СО	4D	89	F5	89	D8	48	29	FA	44	E.¶.^]ÀM‰õ‰ØH)úD
00018010	89	СВ	48	89	55	00	4C	89	FA	4D	89	F7	49	89	F6	48	‰ËH‱U.L‱úM‱÷I‰öH
00018020	89	D6	EΒ	11	OF	1F	40	00	41	OF	Β6	45	01	49	83	C5	‱Öë0.A.¶E.IfÅ
00018030	01	84	СО	74	33	38	D8	75	EF	49	89	F8	48	89	F2	4C	."Àt38ØuïI‰øH‱òL
00018040	89	E9	E8	D9	35	00	00	85	СО	75	DD	8B	55	E8	4C	03	‰éèÙ5ÀuÝ‹UèL.
00018050	65	00	85	D2	74	12	49	01	FD	41	OF	Β6	45	00	84	СО	eÒt.I.ýA.¶E."À
	Fig	ure	8: ľ	nxd	- he	xad	lecir	nal	dun	пр о	f pc	oten	tial	con	figu	ratio	n block

Shown in the **GREEN** box, is the XOR key. Also within short proximity, shown in **BLUE** there are the sizes (in little endian^T) of the data transferred into the first allocated memory region.

Lastly within the **RED** box, there is a **NULL** terminated string of **init**. This could be a useful marker for what might turn out to be some kind of stored configuration.

If we allow the XOR routine to complete its rounds across the data, and repeat the steps from earlier to dump, and then attempt to show the disassembly it now prints some pretty convincing shellcode.

The file rundll32_memory_region_1_xor.bin can also be found in the GitHub repository <u>here</u>

\$ r2 -AA -c 'pd' rundll32_memory_region_1_xor.bin

Γ 74: fcn.0	0000000 (int64_t	arg3, int64_t a	arg4, int64_t arg6);
	; arg int64_t a	rg3 @ rdx	
	; arg int64_t a	rg4 @ rcx	
	; arg int64_t a	rg6 @ r9	
	; var int64_t va	ar_30h <mark>@ rsp+0x</mark> 3	30
	0x00000000	4c8bdc	mov r11, rsp
	; DATA XREF from	m fcn.00000609 (0x8a9(r)
	0x0000003	4883ec68	sub rsp, 0x68
	0x00000007	33c0	xor eax, eax
	0x00000009	49 83c9ff	or r9, 0xfffffffffffffffffffffffff
	0x0000000d	49 89 43 e8	mov qword [r11 - 0x18], rax
	0x00000011	4533c0	xor r8d, r8d
	; DATA XREF from	m fcn.00000609 (@ 0x95c(r)
	0x00000014	49 89 43 e0	mov qword [r11 - 0x20], rax
	0x00000018	498943d8	mov qword [r11 - 0x28], rax
	0x0000001c	49 89 43 d0	mov qword [r11 - 0x30], rax
	; DATA XREFS fro	om fcn.00000609	@ 0xa06(r), 0xad9(r)
	0x00000020	89442430	mov dword [var_30h], eax
	; DATA XREF from	m fcn.00000609 (@ 0x70d(r)
	0x00000024	49 89 53 c0	mov qword [r11 - 0x40], rdx ; arg3
	; DATA XREFS fro	om fcn.00000609	@ 0x7de(r), 0x81b(r), 0xa1f(r), 0xb2b(r)
	0x00000028	ba000000 10	mov edx, 0x10000000
	0x0000002d	49894bb8	mov qword [r11 - 0x48], rcx ; arg4
	0x00000031	49 8d 4b 18	lea rcx, [r11 + 0x18]
	0x00000035	49 89 43 18	mov qword [r11 + 0x18], rax
	0x00000039	48 b8a5a4a3a2.	movabs rax, 0xala2a3a4a5
	0x00000043	ffd0	call rax
	0x00000045	4883c468	add rsp, 0x68
L	0x00000049	с3	ret
	0x0000004a	48895c2408	mov qword [rsp + 8], rbx
	0x0000004f	55	push rbp
	0x00000050	56	push rsi
	0x00000051	57	push rdi
	0x00000052	4154	push r12
	0x00000054	4157	push r15

Figure 9: radare2 - Shell code disassembly

We can validate that the XOR key is correct by applying it to the memory dump file we created previously and comparing the output. *Figure 10* shows the recipe required. You will notice the hexadecimal output matches the instruction bytes in the disassembly above, in *Figure 9*.



Figure 10: CyberChef - XOR routine.

If we remember the call to VirtualAlloc previously, the region was requested with PAGE_READWRITE protection, restricting the ability for execution. There are two possibilities for the shellcode now, the first is it will be executed in its current location or it will be copied somewhere else before executing.

Wherever the shellcode will be executed, the memory region will need its execute permission set. Just as VirtualAlloc was used to allocate the region, we can set a break point on VirtualProtect as shown below.

bp VirtualProtect

Sacrificial DLL Loading

Pressing on with the unpacking, there is a call to LoadLibraryA with the parameter to load the DLL dpx.dll from the default C:\Windows\System32 directory.

Loading the dpx.dll library is followed by locating an exported function named dpx.DpxCheckJobExists. Based on my loose understanding of how the function is located, I believe this is chosen simply because it is the first function listed in the exports. This technique would allow the malware authors to potentially swap the dpx.dll for another fairly easily...

The address returned from for dpx.DpxCheckJobExists is then passed to VirtualProtect⁸, executed via a call r15 instruction at 0x0814E3786BE.

The arguments passed to VirtualProtect can be arranged as shown.

This function call will mark 0x15BB (5563) bytes as PAGE_READWRITE starting at the address of dpx.DpxCheckJobExists.

VirtualProtect(dpx.CheckJobExists, 0x15BB, 0x4)

The original protection was PAGE_EXECUTE_READ, so the additional permission to allow writing is enough to know we likely want to keep an eye on this region.

Moving on, we hit a familiar breakpoint for the malware's memcpy routine. This time, 0x15BB bytes are being moved from the address 0x0814E39342A once again located in the .reloc section, to the address of dpx.DpxCheckJobExists. The file offset for this data is 0x2A42A.

Rather interestingly the bytes representing the amount of data transferred $0 \times 15BB$ are located in the output of *Figure 8* underneath the $0 \times 4A$ byte.

Extracting the 0x15BB bytes from the newly copied location, we can take a look and see what the original code for dpx.DpxCheckJobExists has been replaced with.

\$ r2 -AA -c 'pd' rundll32_dpx_checkjobexists.bin

г ²⁰ :	0 <u>000000 ();</u>		
	0x00000000	4c e3ab	jrcxz 0xfffffffffffffffae
	0x0000003	23e6	and esp, esi
		286925	sub byte [rcx + 0x25], ch
		be 722b363e	mov esi, 0x3e362b72 ; 'r+6>'
	0x0000000d	382a	cmp byte [rdx], ch
	0x000000f	3b31	cmp esi, dword [rcx]
	0x00000011	3428	xor al, 0x28
L	0x00000013	61	
	; DATA XREF from		0 +0x9f5(r)
	0x00000014	7364	jae 0x7a
	0x00000016	2be5	sub esp, ebp
	0x00000018	03 4f cf	add ecx, dword [rdi - 0x31]
	0x0000001b	38e0	cmp al, ah
	0x0000001d	85 5733	test dword [rdi + 0x33], edx
	; DATA XREFS fro		@ +0x6a3(r), +0x6bf(r), +0x12dd(r)
	0x00000020	336320	xor esp, dword [rbx + 0x20]
	; DATA XREF from		
	0x00000023	e49a	in al, 0x9a
	; DATA XREF from		0 +0x683(r)
	0x00000025	60	
	; DATA XREF from		@ +0x1433(r)
	0x00000026	59	pop rox
	; XREFS: DATA 0	x00000087 DATA	0x00000a8a DATA 0x00000c30 DATA 0x00001081 DATA 0x00001383 DATA 0x000015a9
	0x00000027	11d1	adc ecx, edx
	0x00000029	8e0cb3	mov cs, word [rbx + rsi*4]
	0x0000002c	5 f	pop rdi
	; DATA XREFS fro		@ +0x41(x), +0xb0b(r), +0xf0a(r)
	0x0000002d	f2 22 e0	and ah, al
	0x00000030		xchg esi, eax
	; DATA XREFS fro		@ +0x867(r), +0x144c(r)
	0x00000031	3dea997bb9	стр еах, 0хb97b99еа
	; DATA XREFS fro		
_	0x00000036	d9	
	0x00000037	8be4	mov esp, esp
	0,00000070	75/6	

Figure 11: radare2 - Dpx.CheckJobExists overwritten data

It doesn't look shellcode, so likelihood is there will be an additional routine to de-obfuscate it.

Through setting some access breakpoints you will stumble elegantly upon yet another routine with an XOR instruction located at 0x0814E3786E1. This routine iterates over the dpx.DpxCheckJobExists location using the string chokopai723 as a key for all 0x15BB bytes.

The string chokopai732 was passed into the process via the command line flag /k.

If we take a look at the dpx.DpxCheckJobExists contents shown in *Figure 12*, once the XOR has been applied we get something more resembling shellcode.

```
$ r2 -AA -c 'pd' rundll32_dpx_checkjobexists_xor.bin
```

0x0000000	4c8bc4	mov r8, rsp	; int64_t	: arg_28h
; DATA XREF from	m fcn.00000000 (@ 0x580(r)		
0x0000003	48895808	mov qword [rax + 8], rbx		
0x00000007	4c 89 40 18	mov qword [rax + 0x18], r8		
; DATA XREF from	m fcn.000005f4 (∂ 0x81b(r)		
0x0000000b	55	push rbp		
0x000000c	56	push rsi	; arg2	
0x0000000d	57	push rdi	; arg1	
0x0000000e	4154	push r12		
; DATA XREF from	m fcn.000008ac (@ 0xc13(r)		
0x00000010	4155	push r13		
0x00000012	4156	push r14		
0x00000014	4157	push r15		
0x00000016	488d6c24a0	lea rbp, [rsp - 0x60]		
0x0000001b	4881ec600100.	sub rsp, 0x160		
0x00000022	48 8bf1	mov rsi, rcx	; int64_t	: arg2
0x00000025	0f 2970 b8	movaps xmmword [rax - 0x48],		
0x00000029	b9 3e 80 3c 9a	mov ecx, 0x9a3c803e	; int64_t	: arg_20h
0x0000002e	4d 8bf9	mov r15, r9	; arg6	
0x00000031	4d 8bf0	mov r14, r8		
0x00000034	4c8bea	mov r13, rdx	; arg3	
0x00000037	e88c100000			
0x0000003c	b91808cc 67	mov ecx, 0x67cc0818	; int64_t	: arg_20h
0x00000041	488945b0	mov qword [var_50h], rax		
0x00000045	e8 7e10 0000			
0x0000004a	b91f1e5ad4	mov ecx, 0xd45a1e1f	; int64_t	: arg_20h
0x0000004f	48 8bd8	mov rbx, rax		
0x00000052	e8 71 100000			
0x00000057	65488b0c2530.	mov rcx, qword gs:[0x30]		
; CALL XREF from	m fcn.000008ac (0 OxfeO(x)		
0x00000060	4883cfff	or rdi, Oxfffffffffffffffff		
0x00000064	418b5708	mov edx, dword [r15 + 8]		
; CALL XREFS fr	om fcn.000008ac	@ 0xc7e(x), 0x104b(x)		
0x00000068	458b6628	mov r12d, dword [r14 + 0x28]		
0x0000006c	81c260080000	add edx. 0x860		

Figure 12: radare2 - Dpx.DpxCheckJobExists shellcode

The sample then makes another call to VirtualProtect, restoring the page protection on dpx.DpxCheckJobExists back to PAGE_EXECUTE_READ.

Now the code is executable again, the sample executes the newly laid out shellcode by call rsi operation at 0x0814E378421. This can be intercepted by setting a breakpoint on the dpx.DpxCheckJobExists symbol.

Executing the shellcode located at dpx.DpxCheckJobExists, it uses an internal routine labelled below as mw_resolve_api_hash_location to locate the procedure addresses for 3 API's. The use of API hashes to resolve routines is quite common in malware, as it makes it much harder to see what is being used.

The hash values are usually fairly static, although there a few different methods employed, "search engine-ing" the hexadecimal values is the first step.

Special thanks to <u>this</u> GitHub project by **hidd3ncod3s** for supplying the hashes and corresponding API routines.

From the following disassembly we can see 3 values being moved into ECX before the function mw_resolve_api_hash_location is used. The labels in the disassembly, show the methods being passed:

- NtCreateThreadEx (0x9a3c803e)
- RtlAllocateHeap (0x67cc0818)
- RtlFreeHeap (0xd45a1e1f)

0x00000000	4c8bc4	mov r8, rsp	; int64_t arg_28h	
; DATA XREF from	n fcn.00000000 (@ 0x580(r)		
0x00000003	48895808	mov qword [rax + 8], rbx		
0x00000007	4c894018	mov qword [rax + 0x18], r8		
; DATA XREF from		@ 0x81b(r)		
	55	push rbp		
	56	push rsi	; arg2	
0x0000000d	57	push rdi	; arg1	
	4154	push r12		
; DATA XREF from		0 0xc13(r)		
0x00000010	4155	push r13		
0x00000012	4156	push r14		
0x00000014	4157	push r15		
0x00000016	48 8d 6c24 a0	lea rbp, [rsp - 0x60]		
0x0000001b	4881ec600100.	sub rsp, 0x160		
0x00000022	48 8bf1	mov rsi, rcx	; arg4	
0x00000025	0f 2970 b8	movaps xmmword [rax - 0x48],		
0x00000029	b9 3e 80 3c 9a	mov ecx, 0x9a3c803e	; int64_t arg_20h ;	
0x0000002e	4d8bf9	mov r15, r9	; arg6	
0x00000031	4d 8bf0	mov r14, r8		
0x00000034	4c8bea	mov r13, rdx	; arg3	
0x00000037	e88c100000			
0x0000003c	b91808cc 67	mov ecx, 0x67cc0818	; int64_t arg_20h ;	
0x00000041	488945b0	mov qword [var_50h], rax		
0x00000045	e8 7e10 0000			
0x0000004a	b91f1e5ad4	mov ecx, 0xd45a1e1f	; int64_t arg_20h ;	
0x0000004f	488bd8	mov rbx, rax		
0x00000052	e8 71 100000			
 		. [

Figure 13: radere2 - API hashes being resolved.

Once the API's have been resolved, the routine RtlAllocateheap⁹ is called using the call rbx instruction, and $0\times335B$ (13147) bytes are requested.

00007FFB42D8F301 00007FFB42D8F304	45:8BC4 44:8965 p8	mov r8d,r12d		^	Hic	le FPU
00007FFB42D8F308	48:8849 30	mov rcx, qword ptr ds: [rcx+30]		RAX	00007FFB55974760	<pre><ntdll.rtlereeheap></ntdll.rtlereeheap></pre>
00007FFB42D8F30C	FFD3	call rbx	call rtlAllocateHeap	RBX	00007FFB5597A9A0	<ntdll.rtlallocateheap></ntdll.rtlallocateheap>
00007FFB42D8F30E	6548:8B0C25 6000000	mov rcx, qword ptr gs: [60]		RCX	0000011FBC370000	
00007FFB42D8F317	48:8BD8	mov rbx,rax	rbx:RtlAllocateHeap, rax:RtlFree	RDX	000000000000008	
00007FFB42D8F31A	48:8945 D0	mov qword ptr ss:[rbp-30],rax	rax:RtlFreeHeap	RBP	0000001D8BE9F8E0	
00007FFB42D8F31E	45:33c0	xor r8d,r8d		RSP	0000001D8BE9F/E0	<&RtIAllocateHeap>
00007FFB42D8F321	48:8B51 18	<pre>mov rdx,qword ptr ds:[rcx+18]</pre>		RSI	00000814E360000	r.000000814E360000
00007FFB42D8F325	48:8B42 10	<pre>mov rax,qword ptr ds:[rdx+10]</pre>	rax:RtlFreeHeap	RDI	FFFFFFFFFFFFFFFF	
00007FFB42D8F329	48:8B48 30	mov rcx, gword ptr ds: [rax+30]	qword ptr ds:[rax+30]:RtlFreeHea	P 8	00000000003358	
00007FFB42D8F32D	48:85C9	test rcx,rcx		R9	000000000000000000000000000000000000000	
00007FFB42D8F330	✓ 74 34	je dpx.7FFB42D8F366		R10	00000000E290FD31	
00007FFB42D8F332	48:3BCE	cmp rcx,rsi		R11	00007FFB55ABC4D0	ntd11.00007FFB55ABC4D0
00007FFB42D8F335	✓ 74 05	je dpx.7FFB42D8F33C		R12	00000000000335в	
00007FFB42D8F337	48:8B00	mov rax, gword ptr ds:[rax]	rax:RtlFreeHeap, gword ptr ds:[r	R13	000000814E378BA2	"init"
00007FFB42D8F33A	▲ EB ED	jmp dpx.7FFB42D8F329		R14	0000001D8BE9FA20	
00007FFB42D8F33C	48:8BCB	mov rcx,rbx	rbx:RtlAllocateHeap	RT2	0000001D8BE8E8E8C0	
00007FFB42D8F33F	48:8B50 50	mov rdx, qword ptr ds:[rax+50]	qword ptr ds:[rax+50]:RtlFreeHea	/ <		>
				1		

Figure 14: x64dbg - RtlAllocate 0x335b Bytes

Once the region is allocated, the shellcode then accesses its own processes Process Envonment Block aka the PEB, to retrieve the full command line given.

000001BF75AB9850	43	00	ЗA	00	5C	00	55	00	73	00	65	00	72	00	73	00	C.:.∖.U.s.e.r.s.
000001BF75AB9860	5C	00	6D	00	61	00	6C	00	77	00	61	00	72	00	65	00	∖.m.a.l.w.a.r.e.
000001BF75AB9870	5C	00	44	00	65	00	73	00	6B	00	74	00	6F	00	70	00	\.D.e.s.k.t.o.p.
000001BF75AB9880	5C	00	72	00	2E	00	64	00	6C	00	6C	00	20	00	22	00	\.rd.l.l".
00001BF75AB9890	43	00	3A	00	5C	00	57	00	69	00	6E	00	64	00	6F	00	C.:.\.W.i.n.d.o.
00001BF75AB98A0	77	00	73	00	5C	00	53	00	79	00	73	00	74	00	65	00	w.s.\.S.y.s.t.e.
00001BF75AB98B0	6D	00	33	00	32	00	5C	00	72	00	75	00	6E	00	64	00	m.3.2.∖.r.u.n.d.
00001BF75AB98C0	6C	00	6C	00	33	00	32	00	2E	00	65	00	78	00	65	00	1.1.3.2e.x.e.
000001BF75AB98D0	22	00	20	00	43	00	3A	00	5C	00	55	00	73	00	65	00	"C.:.\.U.s.e.
00001BF75AB98E0	72	00	73	00	5C	00	6D	00	61	00	6C	00	77	00	61	00	r.s.∖.m.a.l.w.a.
00001BF75AB98F0	72	00	65	00	5C	00	44	00	65	00	73	00	6в	00	74	00	r.e.∖.D.e.s.k.t.
00001BF75AB9900	6F	00	70	00	5C	00	72	00	2E	00	64	00	6C	00	6C	00	o.p.\.rd.1.1.
000001BF75AB9910	20	00	76	00	63	00	61	00	62	00	20	00	2F	00	6B	00	.v.c.a.b/.k.
00001BF75AB9920	20	00	63	00	68	00	6F	00	6в	00	6F	00	70	00	61	00	.c.h.o.k.o.p.a.
00001BF75AB9930	69	00	37	00	32	00	33	00	00	00	00	00	00	00	00	00	i.7.2.3
00001BF75AB9940	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
accord TE COLO		22	20		00	<u>^</u> ,	20	0.01	0.0	22	00	60	00	<u> </u>		. ^ ^	

Figure 15: x64dbg - Command line copied from Process Environment Block

Probably not surprisingly, this second shellcode also implements a memcpy routine, as shown in *Figure 16*.

It is first used to copy 0x1EAD (7853) bytes from 0x0814E39580C (file offset 0x2C80C within the .reloc section) to a heap allocated region. *Figure 8* above contains the value 0x1EAD within the configuration block at offset 0x17FD0.

For future reference, the screen shot below shows the destination address in the RCX register as 0x023D5D94A0B0.



Figure 16: radare2 - DPX.dll shellcode memory routine.

Extracting the data that was just copied reveals not too much, and you might be able to spot a familiar pattern occurring.

Shellcode Patching

Moving on to the next call of the memcpy routine, the sample copies 0xC4E (3150) bytes from the very first allocated memory region to the tail of the data written into the heap region previously described.

This second chunk of data being copied was originally transferred from $0 \times 0814E394BBE$ (file offset $0 \times 2BBBE$) into memory region 1, where is was then de-obfuscated.

The data copied into this heap region becomes very relevant later on. At this stage there is some missing information so don't dump the memory region just yet. To clarify, the first chunk is obfuscated in some way, the second chunk is valid shellcode.

The next call the memcpy routine is used to copy a more 4 bytes containing the value 0x5B330000 into a location within the first allocated memory region. If we swap the endianness of 0x5B330000 we get 0x335B, matching the size of a previously copied segment of shellcode... very interesting...

Next, the shellcode's routine for locating a procedure based on its hash is used to locate CreateThread. This location is then used to patch the shellcode that was written into the first region of allocated memory, using the memory routine.

Figure 17 shows the start of the memcpy routine with the shellcode to be patched in the lower pane. Currently, the 8 bytes to be patched contains 0xA1A2A3A4A5

				_							
	00007FFB42D90778	4C:894424 18	mo∨ qwo	rd ptr ss:[r:	sp+18],r8		~				Show FPU
	00007FFB42D907782	48:894C24 08	mov qwo	rd ptr ss:[r	sp+8, rcx			PAX 00000	000000162	1.171	
•	00007FFB42D90787	48:83EC 18	sub rsp	,18	· •			RBX 000000	000000000000000000000000000000000000000	L !	
•	00007FFB42D9078B	48:8B4424 20	mov rax	,qword ptr s	s:[rsp+20]			RCX 00000	2743DE801AC		
1	00007FFB42D90790	48:890424	mo∨ qwo	rd ptr ss:	sp],rax			RDX 000000		<&Create1	hread>
	00007FFB42D90794	40:004424 20	mov rax	, dword ptr s	s: [rsp+20]			RSP 00000	DA3D9ABF428		
	00007FFB42D9079F	48:837024 30 (rd ptr ss:	sp+301.0			RSI 00000	DA1A2A3A4A5		
	00007FFB42D907A4	✓ 74 34	je dpx.	7FFB42D907DA	-p <mark>.</mark> ,.			RDI FFFFF	FFFFFFFFFF		
	00007FFB42D907A6	48:8B0424	mov rax	,qword ptr s	s:[rsp]			R8 00000	0000000008		
	00007ffb42d907aa	48:8B4C24 08	mov rcx	,qword ptr s	s:[rsp+8]			R9 00000	800000000008		
	00007FFB42D907AF	8A09	mov cl,	byte ptr ds:	[rcx]			R10 00000	0000000161	L'š'	
	0000/FFB42D90/B1	8808	mov byt	e ptr ds:[ra:	x],cl			R11 000000	00000000008		
	00007FFB42D907B5	40:080424	inc rax	, qword ptr S	s:[rsp]			K12 000000	JAIAZAJA4AJ		
	00007FFB42D907BA	48.890424	mov awo	rd ntr ss.	spl rax			`			
	00007FFB42D907BE	48:8B4424 08	mov rax	,qword ptr s	s:[rsp+8]		\sim	Default (x64 fast	call)		
	(1.				>	1. max 00000	27420590146 00	0002742059	01.40
-		1011 5000000.200						2: rdx 00000	12743DE801AC 00	CreateThre	ad> (000000430948
r	ss:[qword ptr ss:[r	sp+18]]=[000000A3D9A	BF440]=A1A2A3A4	1A5				3: r8 000000	0000000008 000	00000000000	000000000000000000000000000000000000000
								4: r9 000000	0000000008 000	0000000000	008
0	07FFB42D90778 dpx.dl	1:\$50778 #4FB78 <mw_< td=""><td>memcpv></td><td></td><td></td><td></td><td></td><td>5: [rsp+28]</td><td>000000000000000000000000000000000000000</td><td>00 0000000</td><td>00000000</td></mw_<>	memcpv>					5: [rsp+28]	000000000000000000000000000000000000000	00 0000000	00000000
									00000.00		
)	1 🚛 Dump 2 🚛	Dump 3 🛛 🚛 Dump 4	🚛 Dump 5	💮 Watch 1	x= Locals	Struct	<u>.</u>	Disassembly	000000A3D	ABF428 00	000000000000000000000000000000000000000
l.	DE80146 33D2		vor edv edv						000000A3D	ABF438 FF	FFFFFFFFFFFFF
	DE801A8 33C9		xor ecx.ecx						^ 000000A3D9	ABF440 00	00000A1A2A3A4A5
30	DE801AA 48:B8	A5A4A3A2A1000000	mov rax, A1A2/	A3A4A5					000000A3D9	ABF448 FF	FFFFFFFFFFFFF
BE	DE801B4 FFD0		call rax						000000A3D9	ABF450 00	000000000000000000000000000000000000000
BE	DE801B6 48:850	c0	test rax,rax						000000A3D9	ABF458 00	000000000000000000000000000000000000000
BE	≥ 20189	- 9	je 2743DE8010	0					000000A3D9	ABF400 00	10002/430EF4310
	JEOUISE 48:88	10	mov rcx, rax						000000A3D	ABE470 00	00000300000004
	DE801C0 49.8B	CE	mov rcx r15						V 000000A3D	ABF478 BA	ADF00D0000008
	45.00		1 100 1 00,115			1		`		in [
											Act

Figure 17: x64dbg - Shell code patching routine, before patch.

Figure 18 shows the shellcode after being patched, containing the address of CreateThread ready for it to be copied into RAX and then called.

00007FFB42D907AA 00007FFB42D907AF	48:8B4C24 08 3A09	<pre>mov rcx,qword ptr ss:[rsp+8 mov cl,byte ptr ds:[rcx]</pre>				Show FPU
00007FFB420907B1 00007FFB420907B3 00007FFB420907B7 00007FFB420907BA 00007FFB420907B4 00007FFB420907C3 00007FFB420907C6 00007FFB420907C6	3808 18:8B0424 18:FFC0 18:890424 18:8B4424 08 18:FFC0 18:894424 08 18:894424 30	<pre>mov byte ptr ds:[rax].el mov rax,qword ptr ss:[rsp] inc rax mov qword ptr ss:[rsp],rax mov qword ptr ss:[rsp+8] inc rax mov qword ptr ss:[rsp+8],ra: mov qword ptr ss:[rsp+8],ra;</pre>] ×	RAX 00 RBX 00 RCX 00 RDX 00 RDX 00 RBP 00 RSP 00 RSI 00 RDI FF	0002743DE801AC 00000000000000 0000A3D9ABF400 0000A3D9ABF4D0 0000A3D9ABF330 0000A3D9ABF428 0000A1A2A3A4A5 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	<&CreateThread> &"init" <&CreateThread>
00007FF842D907D0 00007FF842D907D3 00007FF842D907D8 00007FF842D907D8 00007FF842D907DF 00007FF842D907DF 00007FF842D907E4	48:FFC8 48:894424 30 EB C4 48:884424 20 48:83C4 18 E3 44:894424 18	<pre>dec rax mov qword ptr ss:[rsp+30],r jmp dpx.7FFB42D9079E mov rax,qword ptr ss:[rsp+20 add rsp,18 ret mov dword ptr ss:[rsp+18],r</pre>	ax 0] 8d	R8 00 R9 00 R10 00 R11 00 R12 00 <	0000000000008 0000000000008 0000000000	L'š'
00007FFB42D907E9	395424 10	mov dword ptr ss:[rsp+10],e	dx 🗸	Default (x64	fastcall)	
07FFB42D907E3 dpx.dll:\$507	E3 #4FBE3			1: rcx 00 2: rdx 00 3: r8 000 4: r9 000 5: [rsp+2	00000A3D9ABF400 000 0000A3D9ABF4D0 <& 0000000000008 0000 000000000008 0000 (8] 000000000000000000000000000000000000	0000A3D9ABF400 &"init" CreateThread> (000000A3D9ABF4 000000000008 000000000008 00 0000000008
1 🚛 Dump 2 🚛 Dump 3	3 📖 Dump 4	📖 Dump 5 🛛 💮 Watch 1 🛛 🖛 Loc	als 🦻 Struct 🛄	Disassembly	000000A3D9	ABF428 00007FFB42D8F51D r ABF430 000002743DE801AC
1E801A6 33D2 1E801A8 33C9 1E801A8 33C9 1E801A8 48:188 10BD7 1E801B4 FFD0 1E801B6 48:185C0 1E801B8 48:186C8 1E801B8 48:186C8 1E801B8 FFD6 1E801C0 49:188CF	D54FB7F0000	<pre>xor edx,edx xor ecx,ecx mov rax,kernel32.CreateThread> call rax test rax,rax je 27430E801C0 mov rcx,rax call rsi mov rcx,r15</pre>	rcx:&"init" rcx:&"init"		 ▲ 000000A3D9 ● 00	ABF43 000000A3D9ABF4D0 ABF440 0000000000000 ABF440 0000000000000 ABF450 00000000000000 ABF450 0000000000000 ABF450 0000000000000 ABF458 00000000000000 ABF458 0000002703000000 ABF470 00000030000004 ABF478 BAAPF0000000005 ABF478 BAAPF0000000008

Figure 18: x64dbg - Shell code patching routine, after patch.

The same process of locating a function, and then patching shellcode is also carried out for additional functions.

The complete list of functions resolved and patched is:

- CreateThread
- LoadLibraryA

- ReadProcessMemory
- VirtualProtect
- RtlAllocateHeap
- NtClose
- ZwCreateThreadEx

Next comes a routine that appears (at least to me), to parse the ntdll.dll module for the various syscall operations.

Continuing the execution again we hit another call to the memcpy routine, this time copying $0 \times B$ (11) bytes from a stack based address into a location within the first allocated memory region.

4C 8B D1 B8 00 00 00 00 0F 05 C3

At first glance the purpose of the byte sequence is not obvious, it's certainly not an address as previously observed. If you continue to view the disassembler during the memcpy routine, you would have seen a patch applied to call a syscall directly.

We can quickly check the above hexadecimal opcodes using the **CyberChef**¹⁰ recipe to **Disasemble X86** or use the following **rasm2** command.

\$ rasm2 -a x86 -b 64 -d '4C 8B D1 B8 00 00 00 00 0F 05 C3'
mov r10, rcx
mov eax, 0
syscall
ret

This syscall related activity has a lot of similarities with what is described <u>here</u> over at <u>www.ired.team</u>

Once the syscalls stubs have been copied over, the function ZwAllocateVirtualMemory, is then used to request 0x3841 (14401) bytes of memory with the protection constant PAGE_WRITECOPY, this region will be labelled and hence forth known as memory region 2.

Figure 19 shows the call to ZwAllocateVirtualMemory being made. The registers RDX and R8 are being used to provide the address and protection flags. As can be seen in the display, RCX contains the location of memory, which contains the location in memory that is being altered....aka a pointer.

The address being altered here is stored in little-endian, and is 0x29E3E670000 as shown in the lower dump 2 pane.

 00007FFB425CFAFD 00007FFB425CFB01 	4C:8D45 E0 48:83C9 FF	lea r8,qword ptr ss:[rbp-20] or rcx.FFFFFFFFFFFFFFFF	^	
• 00007FFB425CFB05	48:8D55 58	<pre>lea rdx,qword ptr ss:[rbp+58] call n13</pre>		RAX 000000D0D0DAFF240
 00007FFB425CFB0C 00007FFB425CFB0E 00007FFB425CFB12 00007FFB425CFB14 00007FFB425CFB1C 	85C0 40:0F94C7 8BC7 48:889C24 90000000 48:83C4 50	test eax,eax sete dil mov eax,edi mov rbx,qword ptr ss:[rsp+90] add rsp,50	r	RBX 0000/FESSA21/0 RCX FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF RDX 000000D0DAFF248 RBP 000000D0DAFF1F RSP 000000D0DAFF1A0 RSI 000000D0DAFF1A0 RSI 000000000000000000000000000000000000
 00007FFB425CFB20 00007FFB425CFB22 00007FFB425CFB28 00007FFB425CFB26 00007FFB425CFB28 00007FFB425CFB28 	41:5F 41:5D 41:5D 41:5C 5F	pop r15 pop r14 pop r13 pop r12 pop rdi	r r	RE 000000000000000000000000000000000000
 00007FFB425CFB29 00007FFB425CFB28 00007FFB425CFB2B 00007FFB425CFB2C 00007FFB425CFB3F 00007FFB425CFB37 00007FFB425CFB38 00007FFB425CFB38 	5D C3 48:8BC4 48:8958 08 4C:8948 20 4C:8940 18 8950 10 55	<pre>pop rs1 pop rs1 mov rax,rsp mov qword ptr ds:[rax+8],rbx mov qword ptr ds:[rax+20],r9 mov qword ptr ds:[rax+18],r8 mov dword ptr ds:[rax+10],edx puch rbs</pre>	r E	C Default (x64 fastcall) 1: rcx FFFFFFFFFFFFFFFFFF 2: rdx 000000D0DAFF248 0(3: r8 00000D0D0AFF1D0 00(4: r9 000000000000020 00(5: [rsp-20] 000000D0DAFF1
• <		publi rop	>	. [
🚛 Dump 1 🚛 Dump 2 🚛	Dump 3 🛛 🚛 Dump 4	Dump 5 👹 Watch 1 🗵 🖛 Locals	🥬 Stru	ct 🔤 Disassembly
Address Hex 000000D0DAFF248 00 00 67 3 000000D0DAFF258 00 00 00 0 000000D0DAFF268 00 00 67 3 000000D0DAFF278 08 00 00 0	E 9E 02 00 00 00 00 00 00 0 00 00 00 00 70 47 47 3E E 9E 02 00 00 0 0D FO AD BA 2B 1B 00 00	ASCII 00 00 00 00 9E 02 00 00 03 00 00 00 5B 33 00 00 		

Figure 19: x64dbg - ZwProtectVirtualMemory from R13 register

After building the syscall routines and patching the shellcode in memory region 1, more API's are resolved.

- NtOpenProcess
- NtClose
- RtlFreeHeap

The malware went to a lot of trouble to generate the syscall stubs, it finally begins to use them starting with a call via the **RSI** register.

Setting an execution breakpoint on the region of memory containing the syscall stubs will allow you to step through the next procedure.

Figure 20 shows the call via the RSI register, with a value of 0×5 being passed in on the RCX register. In the disassembly view in the bottom pane, you can see the syscall ID being loaded into RAX, the value 0×36 resolves to NtQuerySystemInformation¹¹

Taking a look at the documentation for NtQuerySystemInformation <u>here</u> provided by Geoff Chappell, the value 0x5 is the constant for SystemProcessInformation. This is being used to generate a process listings, more details can be found <u>here</u>

 000071 00071 00071	FF84DD20570 FF84DD20573 FF84DD2057A FF84DD2057A FF84DD2057C FF84DD2057C FF84DD20580 FF84DD20585 FF84DD20587 FF84DD20587 FF84DD20582 FF84DD20590 FF84DD20593 FF84DD20594 EF84DD20595	48:88D0 B9 05000000 FFD6 88D8 85C0 75 93 48:886C24 50 88C3 48:885C24 40 48:83C4 20 41:5E 5F 5E C3 C6	<pre>mov rdx,rax mov ecx,5 call rsi mov ebx,eax test eax,eax jne dpx.7FF84DD2051: mov rbp,qword ptr s mov eax,ebx mov rbx,qword ptr s add rsp,20 pop r14 pop rdi pop rsi ret int3</pre>	1 55:[r 1	BAX 000001CBCE037DC0 RBX 000000000000100 BCX 00000000000000 BDX 00000000000000 BDX 000001CBCE037DC0 RBP 00007FF86177A9A0 RSP 000001CBCE25051C RDI 000000946F07F140 RSI 000000946F07F1E0 B8 00000000000100 B9 000000946F07F188 B10 000001CBCE020000 B11 000000946F07F1278 R12 000001CBCE250000 B13 000000000000000
• 00007f • 00007f • 00007f	FF84DD20596 FF84DD20597	CC CC 48:805 c34_08	int3 int3	>	R14 0000/FF861//4/60 R15 000001CBCE251BE4
p data 🚦	🛄 Dump 2 🛛 🚛 Dump 3	🚛 Dump 4 🛛 🚛 Dump 5	👹 Watch 1 🛛 🖛 Locals	🎾 Str	uct 🗰 Disassembly
3CE25051C 3CE25051F 3CE250524 3CE250526 3CE250527	4C:8BD1 B8 36000000 OF05 C3 0030	mov r10,rcx mov eax,36 syscall ret add byte ptr ds:[rax]	36 , dh	:'6'	

Figure 20: x64dbg - NtQuerySystemInformation native syscall

Once the PID for explorer.exe is located, it is passed to the NtOpenProcess syscall. Opening the rundll32.exe process in **ProcessHacker** we can see the handle to explorer.exe has been opened, as shown in *Figure 21*.

✓ Hide unnamed h	handles			
Type Key Directory File Mutant Directory Semaphore Semaphore Key Key Key Key Key Key WindowStation Desktop WindowStation File Section Key Process	Name HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersio \KnownDlls C:\Windows\System32 \Sessions\1\BaseNamedObjects\SM0:5716:304:WilStagi \Sessions\1\BaseNamedObjects\SM0:5716:304:WilStagi \Sessions\1\BaseNamedObjects\SM0:5716:304:WilStagi \Sessions\1\BaseNamedObjects\SM0:5716:304:WilStagi HKLM HKLM\SYSTEM\ControlSet001\Control\VIIs\Sorting\Versi HKLM HKLM\SOFTWARE\Microsoft\Ole HKLM HKCU\Software\Classes\Local Settings \Sessions\1\Windows\WindowStations\WinSta0 \Default \Sessions\1\Windows\WindowStations\WinSta0 C:\Windows\System32\en-US\rundll32.exe.mui \Windows\Theme318023683 \Sessions\1\Windows\Theme552054757 HKLM\SYSTEM\ControlSet001\Control\Session Manager explorer.exe (2780)	Handle [^] 0x8 0x38 0x44 0x4c 0x58 0x5c 0x5c 0x88 0x8c 0x9c 0xa0 0xa4 0xa8 0x108 0x108 0x100 0x114 0x114 0x120 0x144 0x164	Handle Properties General Security Basic information Name: explorer.exe (2780) Type: Process Object address: 0xffff828176 Granted access: 0x1000 (Qui References References; 519699 Handles: 24	64d5080 ery limited information) Quota charges Paged: 4096 Non-paged: 3144

Figure 21: ProcessHacker - Handle to explorer process opened.

The handle on explorer.exe is then used by a call to NtOpenProcessToken. The returned handle for the token is passed to NtQueryInformationToken before being closed with NtClose.

The syscall NtSystemQueryInformation is then used as it was previously to generate a list of processes running on the system.

A series of calls to NtOpenProcess is then issued against all svchost.exe processes until one can be successfully opened. As the process is running in a non-privileged context, calls to svchost.exe processes running as NT AUTHORITY\SYSTEM are responded to with an access denied value in EAX as shown in *Figure 22*

RBX 000000000000000000000000000000000000	000C0000022 STATUS_ACCESS_DENIED 000000000000 '0' 1583E4503A6 00000000000 040A2A7EED0 040A2A7EDC8 0000000000 040A2A7F6A0
--	---

Figure 22: x64dbg - NtOpenProcess Access Denied.

Note: The *sihost.exe* process is also attempted if the *svchost.exe* process list becomes exhausted.

Once a handle to an sychost.exe process is opened, the token information is harvested using NtOpenProcessToken and NtQueryInformationToken.

To determine if the target svchost.exe process is the correct architecture, NtQueryInformationProcess is used to check the ProcessWow64Information details.

For each thread on the sychost.exe process the following routines are called:

- NtOpenThread
- NtCreateEvent
- NtDuplicateObject
- NtQueueApcThread
- SetEvent

Once each thread has been setup, there is a call to NtQuerySystemTime.

The shellcode residing in memory region 1, is further patched with the value 0xB18 forming the first argument to ReadProcessMemory as shown in *Figure 23*.

000001583E440090	48:8BF8	mov rdi,rax
000001583E440093	4C:8BC0	mov r8,rax
000001583E440096	49:BF 180B00000000	mov r15,B18
000001583E4400A0	49:8BCF	mov rcx, r15
000001583E4400A3	48:B8 50CC7C5FF87F0	<pre>mov rax, <kernel32.readprocessmemory></kernel32.readprocessmemory></pre>
000001583E4400AD	44:8BCE	mov r9d,esi
000001583E4400B0	48:8BD3	mov rdx, rbx
000001583E4400B3	FFD0	call rax
000001502-4400-5	40 20-27-61-07-04	

Figure 23: x64dbg - Length value being patched in shellcode

Using the handle to svchost.exe, the rundll32.exe process makes a call to NtVirtualProtect targeting the address of WinHelpW from user32.dll.

Looking at the R9 register in *Figure 24* you can see the value 0x40, which corresponds to the memory protection constant PAGE_EXECUTE_READWRITE.

000001583E45078C	4C:8BD1 R8 5000000	mov r10,rcx	50+ 'p'	^	Hide FPU	
000001583E450794 000001583E450796 000001583E450797 000001583E450797 000001583E450796 000001583E450796 000001583E45079F 000001583E4507A1	0F05 C3 0070 DA 7E 61 F8 7F 00 00CB 4A:94	inv cut, i ret add byte ptr ds:[rax-26],dh je 1583E4507FD clc jg 1583E45079F add b),cl xchg rsp,rax	NtProtectVirtualMemory	R/ RE RI RI RS RS	xx 000000000000000000 xx 00000000000000	
000001583e4507A3 000001583e4507A7 000001583e4507Ac 000001583e4507Ae 000001583e4507AF 000001583e4507B5 000001583e4507B5	894C88 D1 8851000000 0F05 C3 0090 DA7E61F8 7F 00 00A1 41E0744C 88D1 50500000	<pre>mov dword ptr ds:[rbx+rcx*4-2F] mov eax,51 ret add byte ptr ds:[rax-79E8126],d jg 1583E450787 add byte ptr ds:[rcx+4C74E041],a mov edx,ecx</pre>	51; 'Q'		00000040A2A7EF70 000000000000040 "Q" 00000000000000164 L"T" 1 000000000000386 L"A" 2 00000000000004 "N" 3 000000000000000 "N" 4 00000000000003 S 5 00007FF860401390 <user32.winhelpt< td=""></user32.winhelpt<>	W>

Figure 24: x64dbg - NtVirtualProtect WinHelpW

Payload Transfer

The rundll32.exe process then calls NtCreateSection to create a section within the svchost.exe process. This section is then mapped into view of the rundll32.exe process using NtMapViewOfSection.

With the section accessible to the rundll32.exe process, the memcpy implementation is called twice. The first transfer copies 0x4A bytes, and the second transfers 0x18F bytes from the first memory region.

You'll notice the byte sizes align with the blocks of data transferred from the .reloc section into "memory region 1", which has been decoded and subsequently patched.

The original bytes from both WinHelpW (0x4A) and WinHelpA (0x18F) are copied into a location of memory, possibly for restoring later.

Once data has been written by the rundll32.exe process, NtUnMapviewofSection is called on the section.

Using the handle to the svchost.exe process, the section is mapped into memory using NtMapViewOfSection.

Now comes a really interesting process, to avoid using heavily monitored API's the rundll32.exe process such as WriteProcessMemory.

The rundll32.exe processes calls the NtQueueApcThread routine to schedule an execution of RtlCopyMemory within the svchost.exe process. The source parameter is the location of the mapped memory region of the shared section, the destination parameter contains the address of the WinHelpW routine within user32.dll.

Thus when the queued APC routine executes, the WinHelpW routine will be replaced with shellcode.

The setup for this can be seen in *Figure 25* below.



Figure 25: x64dbg - WinHelpW execution after NtDelayExecution

The same technique is then used to copy data from the mapped section, to overwrite the WinHelpA routine. The shellcode at WinHelpW is then scheduled to execute using the NtQueueApcThread routine as well as Sleep and a call to NtDelayExecution.

Both the WinHelpW and WinHelpA locations have their memory protection restored back to PAGE_EXECUTE_READ using NtVirtualProtectMemory, and the section becomes unmapped in the sychost.exe process with a call to NtUnMapviewofSection.

Execution from this point will continue from within the perspective of the svchost.exe process.

Setting a breakpoint on the WinHelpW routine, we can examine this further.

Executing WinHelpW Shellcode

```
74: fcn.00000000 (int64_t arg3, int64_t arg4, int64_t arg6);
         ; arg int64_t arg3 <u>0</u> rdx
         ; arg int64_t arg4 @ rcx
         ; arg int64_t arg6 @ r9
         ; var int64_t var_30h @ rsp+0x30
         0x00000000
                        4c8bdc
                                       mov r11, rsp
                        4883ec68
                                       sub rsp, 0x68
                        33c0
                                       4983c9ff
                                       mov qword [r11 - 0x18], rax
         0x00000011
                        4533c0
                                       xor r8d, r8d
                        498943e0
498943d8
498943d0
89442430
498953c0
                                      mov qword [r11 - 0x20], rax
                                       mov qword [r11 - 0x28], rax
                                       mov qword [r11 - 0x30], rax
                                       mov dword [var_30h], eax
                                       mov qword [r11 - 0x40], rdx ; arg3
                                       mov edx, 0x1000000
                                       mov qword [r11 - 0x48], rcx ; arg4
                        49894bb8
                        498d4b18
                        49894318
                                       mov qword [r11 + 0x18], rax
         0x00000035
                                      movabs rax, 0x7ff8617ee880
                        48b880e87e61.
                        ffd0
                                       call rax
         0x00000045
                        4883c468
                                       add rsp, 0x68
         0x00000049
                                       ret
```

\$ r2 -AA -c 'pdf' svchost_user32_injected.bin

Figure 26: radare2 - svchost.exe User32.dll WinHelpW Shellcode

Calls to OpenProcess on the rundll32.exe process. Then ReadProcessMemory from the rundll32.exe process, the heap allocated data previously described.

00007FFD4AB0CC50 <ker 00007FFD4AB0CC57</ker 	<pre>- jmp qword ptr ds:[<readprocessmemory>] int3</readprocessmemory></pre>	ReadProcessMemory			Hide FPU
00007FFD4AB0CC58	int3		RAX	00007FFD4AB0CC50	<kernel32.readprocessmemory></kernel32.readprocessmemory>
00007FFD4AB0CC59	int3		RBX	00000290552B9840	
00007FFD4AB0CC5A	int3		RCX	000000000000470	L'Ψ'
00007FFD4AB0CC5B	int3		RDX	00000290552в9840	
00007FFD4AB0CC5C	int3		RBP	000000CD1B38F9F0	
00007FFD4AB0CC5D	int3		RSP	000000CD1B38F9B8	
00007FFD4AB0CC5E	int3		RSI	000000000000335B	
00007FFD4AB0CC5F	int3		RDI	0000052853098510	
00007FFD4AB0CC60	int3		P.S.	00000238E9688E70	
00007FFD4AB0CC61	int3		R9	0000000000000335B	
00007FFD4AB0cc62	int3		R10	0000000000001FFF	
00007FFD4AB0cc63	int3		R11	000000000000000000000000000000000000000	
00007FFD4AB0CC64	int3		R12	00000000000000000	
00007FFD4AB0CC65	int3		R13	000000000000000000	
00007FFD4AB0cc66	int3		R14	00000000000000000	. 11
00007FFD4AB0CC67	int3		R15	000000000000470	Γ.Ψ.
00007550140000669	5445 C	•			

Figure 27: x64dbg - ReadProcessMemory called from svchost.exe

As you can see from the screen shot in *Figure 28*, some of the data copied may contain a similar configuration block identified with the init keyword. Further down into the bytes you may also spot the bytes 0xD6, 0xB2, 0x07 and 0x00 which was the XOR key used within the rundll32.exe unpacking staged.

0000023BF968c770	69	6E	69	74	00	00	00	00	00	00	00	00	00	00	00	00	init
0000023BF968C780	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0000023BF968C790	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0000023BF968C7A0	00	00	00	00	00	00	00	00	A0	A0	2в	55	90	02	00	00	+U
0000023BF968C7B0	AD	1E	00	00	00	00	00	00	D6	в2	07	00	00	00	00	00	Ö ²
0000023BF968c7c0	4D	BF	2в	55	90	02	00	00	4E	0C	00	00	00	00	00	00	M¿+UN
0000023BF968C7D0	70	57	38	21	21	F7	45	0E	0E	D8	6A	EB	BE	E8	0D	в0	pW8!!÷EØjë¾è.°
0000023BF968C7E0	20	F6	41	46	46	90	1E	09	0A	DD	6F	68	в0	46	2E	29	öAFFÝoh°F.)
0000023BF968C7F0	E9	2D	9D	AF	AE	78	FA	ED	DF	EA	9E	8в	8B	Α9	12	48	é [—] ®xúíßê©.H
0000023BF968C800	27	F1	43	44	C4	5A	Е7	94	C 7	F0	EE	81	18	5B	C9	5E	'ñCDÄZç.Çðî[É^

Figure 28: x64dbg - svchost.exe init configuration block

Taking a look at the shellcode that was placed at WinHelpA statically in *Figure 29*, we can see it contains the string dpx.dll and will call LoadLibraryA to load it.

It then calls VirtualProtect on the routine DpxCheckJobExists to allow a byte copying routine to overwrite its contents, replicating the behaviour from earlier in the unpacking routine.

\$ r2 -AA -c 's 0xe2; pd 40' svchost_user32_injected.bin



Figure 29: radare2 - LoadLibraryA dpx.dll and overwrite DpxCheckJobExists

If you are viewing this dynamically then, you will observe 0xC4E (3150) bytes from the second chunk of data copied from the rundll32.exe process into dpx.DpxCheckJobExists routine.

A call to CreateThread is then issued with a base address of dpx.DpxCheckJobExists

The shellcode located at dpx.DpxCheckJobExists then kicks of a routine to XOR decode some of the remaining data originally sourced from rundll32.exe.

Payload Decrypting

In Figure 30 below we can see the static disassembly output of the XOR routine used.

\$ r2 -AA -c 's 0x57; pd 72' svchost_dpx_dpxcheckjobexists.bin

0x00000057	448d52ff	lea r10d, [rdx - 1]
; CODE XREF from	n fcn.00000000 (@ 0xc9(x)
0x0000005b	48 8bc1	mov rax, rcx
0x0000005e	48 8d 71 01	lea rsi, [rcx + 1]
0x00000062	4899	cqo
0x00000064	49 f7fe	idiv r14
0x00000067	48 8bc1	mov rax, rcx
0x0000006a	4c63da	movsxd r11, edx
0x0000006d	4899	cqo
0x0000006f	83e203	and edx, <mark>3</mark>
0x00000072	48 03c2	add rax, rdx
0x00000075	83e003	and eax, <mark>3</mark>
0x00000078	482bc2	sub rax, rdx
0x0000007b	4863 c8	movsxd rcx, eax
0x0000007e	43 0fb6040b	movzx eax, byte [r11 + r9]
0x00000083	44 0fb68419 48 .	movzx r8d, byte [rcx + rbx + 0x848]
0x000008c	4433c0	xor r8d, eax
0x000008f	48 8bc6	mov rax, rsi
0x00000092	4899	cqo
0x00000094	49 f7fe	idiv r14
0x00000097	4863 c2	movsxd rax, edx
0x0000009a	42 0fb60c08	movzx ecx, byte [rax + r9]
0x0000009f	442bc1	sub r8d, ecx
0x000000a2	41 81c0000100.	add r8d, 0x100
0x000000a9	41 81e0ff0000.	and r8d, 0x800000ff
0x000000b0	7d 0d	jge Oxbf
0x000000b2	41ffc8	dec r8d
0x000000b5	41 81c800ffff.	or r8d, 0xffffff00 ; 4294967040
0x000000bc	41ffc0	inc r8d
; CODE XREF from	n fcn.00000000 (@ 0xb0(x)
0x000000bf	47 88040b	mov byte [r11 + r9], r8b
0x000000c3	48 8bce	mov rcx, rsi
0x00000c6	493 bf2	cmp rsi, r10
0x000000c9	7 e90	jle 0x5b

Figure 30: radare2 - XOR Routine

This routine is used to reveal the **FINAL** PE file payload in its original memory buffer copied over from rundll32.exe, as shown in *Figure 31* there is an MZ header and DOS stub visible.

000001AC4A052011	AD	1E	00	00	00	34	00	00	00	82	2D	80	4D	5A	90	00	4MZ
000001AC4A052021	03	00	00	00	04	10	FF	FF	00	00	в8	20	C6	00	40	12	ÿÿ, Æ.@.
000001AC4A052031	02	EB	01	00	D0	10	0E	1F	BA	0E	00	в4	09	CD	21	00	.ëĐº´.Í!.
000001AC4A052041	00	00	80	B8	01	4C	CD	21	54	68	69	73	20	70	72	6F	,.LÍ!This pro
000001AC4A052051	67	72	61	6D	20	63	61	6E	6E	6F	74	20	62	65	20	72	gram cannot be r
000001AC4A052061	75	6E	00	00	02	84	20	69	6E	20	44	4F	53	20	6D	6F	un in DOS mo
000001AC4A052071	64	65	2E	0D	0D	0A	24	12	11	21	C9	10	93	65	A8	7E	de\$!Ée~~
000001AC4A052081	C0	16	01	42	6E	05	C 0	82	88	A0	в0	67	20	16	CA	7F	ÀBn.À °g .Ê.
000001AC4A052091	C1	6E	0a	05	7F	C0	4F	20	83	CC	7 A	0A	04	83	CC	7E	ÁnÀO .ÌzÌ~
000001AC4A0520A1	C1	64	0A	02	7C	0a	02	52	69	63	68	06	0E	16	25	50	Ád Rich%P
000001ac4a052081	00	08	40	A9	45	00	00	64	86	07	00	9A	9F	87	63	16	@GFdc.

Figure 31: x64dbg - Decoded DOS stub header

As well as the executable file, there also resides some configuration data that is used to allow shellcode to map the PE into the address space.

Value 0x3400 taken from payload structure and passed to RtlAllocateHeap The PE file is the seemingly copied into this allocated memory region.



Figure 32: x64dbg - MZ header being copied into allocated Heap region

Pausing the debugger here, will allow you to extract the executable file before it gets mapped into memory.

As the shellcode within the dpx.DpxCheckJobExists area executes, it calls VirtualAlloc with a base region of 0x0180000000, a size of 0x3000 (12288) bytes and a page protection flag of 0x40 (PAGE_EXECUTE_READWRITE).

00007FFB14FF8C6F	int3		DAV	
00007FFB14FF8C70	<pre>imp gword ptr ds:[<virtualalloc>]</virtualalloc></pre>	,	DBY	0000000000008200
00007FFB14FF8C77	int3		RCX	00000018000000
00007FFB14FF8C78	int3		RDX	000000000000000000000
00007FFB14FF8C79	int3		RBP	0000006cd797FBE0
00007ffb14ff8c7a	int3		RSP	0000006cd797fb58
00007FFB14FF8C7B	int3		RSI	0000000000009000
00007FFB14FF8C7C	int3		RDI	0000020E36CB5110
00007FFB14FF8C7D	int3		D.Q	000000000000000000000000000000000000000
00007FFB14FF8C7E	int3			000000000000000000000000000000000000000
00007FFB14FF8C7F	int3		R10	00000000000000000
00007FFB14FF8C80	int3		R11	0000000000000246
00007FFB14FF8C81	int3		R12	00007FFB15CD7BD0
00007FFB14FF8C82	int3		R13	00007FFB14FFB630
00007FFB14FF8C83	int3		R14	0000/FFB14FF8C70
00007FFB14FF8C84	mov rax, rsp		RT2	0000020E36CB5040
00007FFB14FF8C87	mov gword ptr ds:[rax+8],rbx	\checkmark	DTD	00007EEB14EE8c70
<		>	KTL.	0000/FFB14FF0C/0
-		-		

Figure 33: x64dbg - VirtualAlloc hardcoded 0x0180000000

Once this very specific location of memory is allocated the PE file is mapped into execute, the process for this is well documented elsewhere.

Once mapped, execution is started using a call to CreateThread using the 0x01800028D4 address as the entry point.



Figure 34: x64dbg - CreateThread hardcoded 0x0180000000

Unpacked Payload

Now we have jumped through the many hoops to unpack the final payload, we can validate the contents by loading it into PE-Bear¹².

As you can see from *Figure 35*, the binary lists some imports from the **WINHTTP.dll** that look like might be worthy some additional analysis.

You can find a copy of the file svchost_icedid_unpacked.bin in the GitHub repository for this blog post here, or on the malware Bazaar here.

🕙 PE-bear v0.6.5.2 [C:/Users/malware/Desktop/icedid/svchost_icedid_unpacked.bin]



Figure 35: PE Bear - Unpacked icedid payload from svchost.exe

Final Words

That's it for this blog post, its been quite in depth and low-level. If you want to understand anything covered, or maybe not covered in this post feel free to reach out.

I'm planning to do a part 4 taking a look into the extracted PE file so keep an eye out for that, and in the meantime keep evolving.

References

- 1. <u>https://www.malware-traffic-analysis.net</u> ←
- 2. https://rada.re/n/ ←
- 4. <u>https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/memcpy-wmemcpy</u> ←
- 5. <u>https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-</u> <u>virtualalloc</u> <u>←</u> <u>←</u>²
- 6. <u>https://learn.microsoft.com/en-us/windows/win32/Memory/memory-protection-</u> <u>constants</u> <u>←</u>
- 7. <u>https://en.wikipedia.org/wiki/Endianness</u> *↔*
- 8. <u>https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-</u> <u>virtualprotect</u> ←
- 9. <u>https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-</u> <u>rtlallocateheap</u> ←
- 10. <u>https://gchq.github.io/CyberChef/</u> ↔
- 11. <u>https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-</u> <u>ntquerysysteminformation</u> <u>←</u>
- 12. <u>https://github.com/hasherezade/pe-bear</u> *↔*