

# Carving the IcedId - Part 3

By techevo

Published: 2024-03-17 · Archived: 2026-04-05 16:47:47 UTC

Welcome back to this series, analysing IcedId malware artefacts.

This is part 3 in the series, you can check out [part 1](#) and [part 2](#) 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 [part 2](#).

The data for this case was published by [@malware\\_traffic](#) over at **Malware Traffic Analysis**<sup>1</sup>. You can download all the samples from this case from [here](#).

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 [here](#).

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, [here](#) 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](#) and for dynamic analysis [x64dbg](#) 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.

```
[Exports]
nth  paddr      vaddr      bind  type size lib          name          demangled
-----
1  0x00000420 0x814e361020 GLOBAL FUNC 0    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    msys-edit-0.dll trl_print_completions_horizontally
6  0x000192f0 0x814e379ef0 GLOBAL FUNC 0    msys-edit-0.dll trl_qsort_string_compare
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 t__next_word
10 0x00000800 0x814e361400 GLOBAL FUNC 0    msys-edit-0.dll t__prev_word

[ TRUNCATED ]

152 0x000177a0 0x814e3783a0 GLOBAL FUNC 0    msys-edit-0.dll tistory_expand

[ TRUNCATED ]

430 0x00016fb0 0x814e377bb0 GLOBAL FUNC 0    msys-edit-0.dll there_history
```

```
431 0x000177a0 0x814e3783a0 GLOBAL FUNC 0 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, [here](#).

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.

```
[Imports]
nth vaddr      bind type lib      name
-----
1 0x814e391860 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

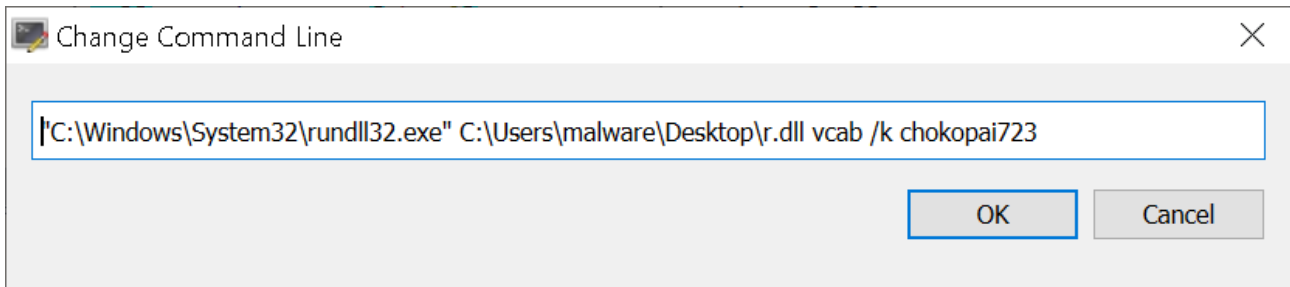
For the dynamic analysis parts of this you will require some working knowledge of **x64dbg**. Primarily around setting breakpoints, although the commands are provided, just knowing what a breakpoint is and how to set it should be enough. If something isn't clear feel free to reach out and ask!

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*.



*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 `0x814e378887` and viewing the sample statically we can see the ASCII characters `0x6B` and `0x2F` being moved into a memory region, as shown in *Figure 2*.

```
0x814e378887 [of]
; CODE XREF from fcn.814e378863 @ 0x814e3788b9(x)
; '/k'
; [0x6b2f:2]=0xffff
mov word [rax + 0x18], 0x6b2f
; arg1
mov rsi, rcx
jmp 0x814e378865

V
|
[0x814e378865]
; CODE XREF from fcn.814e378863 @ 0x814e378890(x)
mov byte [rax + 0x1a], 0
call fcn.814e378b95;[ob]
jmp 0x814e378870

V
```

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

An instruction at `0x0814E378AAB` then copies these two bytes into the `RDX` register. The command line string is then iterated over scanning for the `/k` 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` <sup>4</sup> 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.

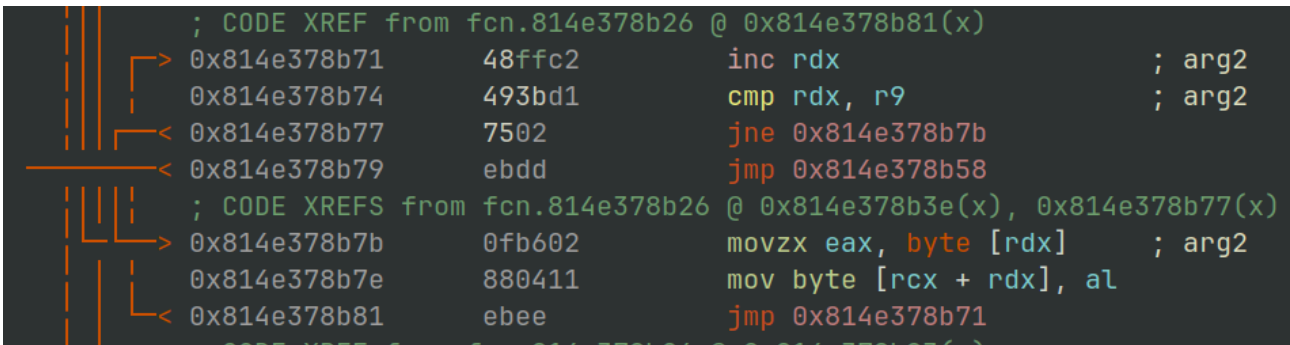


Figure 3: radare2 - IcedId memcpy shellcode routine.

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`

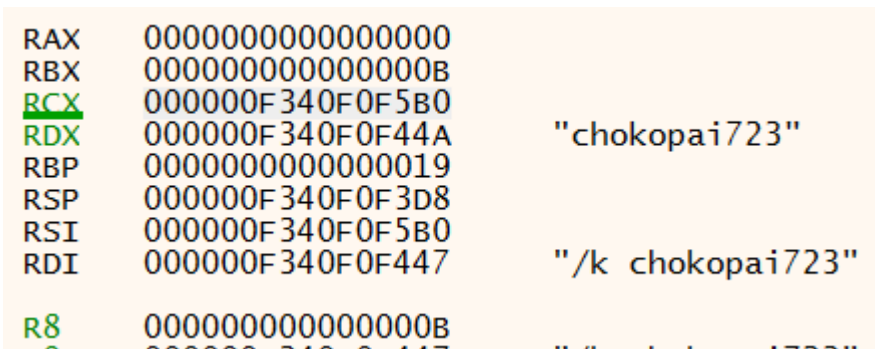


Figure 4: x64dbg - Memory copy routine register usage

Allowing the execution to proceed, the debugger will *break* at a call to `VirtualAlloc`<sup>5</sup>. 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 constants<sup>5 6</sup> 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 `memcpy` 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)
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 [here](#).

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` one-liner.

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
```

```

98: fcn.00000001 (int64_t arg1, int64_t arg3, int64_t arg7);
; arg int64_t arg1 @ rdi
; arg int64_t arg3 @ rdx
; arg int64_t arg7 @ xmm0
0x00000001 39db      cmp ebx, ebx
0x00000003 4855      push rbp
0x00000005 5e        pop rsi
0x00000006 6f        outsd dx, dword [rsi]
0x00000007 3316     xor edx, dword [rsi] ; arg3
0x00000009 fb        sti
0x0000000a 84c9     test cl, cl
0x0000000c 29fb     sub ebx, edi ; arg1
0x0000000e 8e433e   mov es, word [rbx + 0x3e]
0x00000011 f734c0   div dword [rax + rax*8]
0x00000014 9f        lahf
0x00000015 3b44e09f cmp eax, dword [rax + rix*8 - 0x61]
0x00000019 3b44d89f cmp eax, dword [rax + rbx*8 - 0x61]
0x0000001d 3b44d05f cmp eax, dword [rax + rdx*8 + 0x5f]
0x00000021 f623     mul byte [rbx]
0x00000023 309f3b54c06c xor byte [rdi + 0x6cc0543b], bl ; [0x6cc0543b:1]=255 ; arg1
0x00000029 b207     mov dl, 7
0x0000002b 00c6     add dh, al
0x0000002d fb        sti
0x0000002e 8e4b6e   mov cs, word [rbx + 0x6e]
0x00000031 fb        sti
0x00000032 8a4bce   mov cl, byte [rbx - 0x32]
0x00000035 fb        sti
0x00000036 8e43ce   mov es, word [rbx - 0x32]
0x00000039 fa        cli
0x0000003a bfa57211a5 mov edi, 0xa51172a5
0x0000003f 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			
000000814E3784E6	EB 12	jmp r.814E3784FA			
000000814E3784E8	3041 FF	xor byte ptr ds:[rcx-1],al	xor routine memory region 1		
000000814E3784E9	3BD6	cmp edx,esi			
000000814E3784ED	72 F5	jb r.814E3784E4			
000000814E3784EF	EB 76	jmp r.814E378567			
000000814E3784F1	FFC2	inc edx			
000000814E3784F3	0F64438 2C	movzx eax,byte ptr ds:[rax+rdi+2C]	byte ptr ds:[rax+rdi*1+2C]:tistory_expand+90A		
000000814E3784F8	EB EE	jmp r.814E3784E8			
000000814E3784FA	48:8D49 01	lea rcx,qword ptr ds:[rcx+1]			

Figure 6: x64dbg - XOR operation memory region #1

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

```

> 0x814e3784e8  3041ff  xor byte [rcx - 1], al
  0x814e3784eb  3bd6    cmp edx, esi
< 0x814e3784ed  72f5    jb 0x814e3784e4
< 0x814e3784ef  eb76    jmp 0x814e378567
; CODE XREF from sym.msys_edit_0.dll_tistory_expand @ 0x814e378501(x)
> 0x814e3784f1  ffc2    inc edx
  0x814e3784f3  0fb644382c 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 shown 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` then 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 `0x0814E378BD4` or file offset `0x17FD4`, as shown in *Figure 8*.

```

00017F50 C8 8B CA 66 3B C0 74 06 49 8B C2 C3 EB E2 48 0B ÈËÈf;Àt.I<ÀÀëàH.
00017F60 C8 49 0B C8 EB C2 4C 8B D1 4E 8D 0C 02 3A ED 74 ÈI.ÈèÀL<ÑN...:it
00017F70 DD 48 FF C2 49 3B D1 75 02 EB DD 0F B6 02 88 04 ÝHýÀI;ÑNu.ëÝ.¶.^ .
00017F80 11 EB EE 48 83 C2 04 49 3B D1 75 02 EB A4 8B 02 .ëiHfÀ.I;ÑNu.ë«< .
00017F90 89 04 11 EB EE EB 00 65 48 8B 04 25 30 00 00 00 %..ëië.eH<.%0...
00017FA0 C3 4B 69 6E 69 74 00 03 A2 8B 01 00 E5 49 03 00 ÀKinit.c<...ãI..
00017FB0 4A 00 00 00 2F 4A 03 00 8F 01 00 00 2A 34 03 00 J.../J.....*4..
00017FC0 BB 15 00 00 BE 4B 03 00 4E 0C 00 00 0C 58 03 00 »...%K..N....X..
00017FD0 AD 1E 00 00 D6 B2 07 00 C8 49 89 C4 E8 2F 36 00 ....Ö*..ÈI%Àè/6.
00017FE0 00 4C 89 F9 48 89 45 D0 E8 23 36 00 00 41 0F B6 .L%ùH%EDè#6..A.¶
00017FF0 1E 48 89 C7 84 DB 0F 84 26 05 00 00 48 8B 55 D0 .H%Ç..Û...&...H<UB
00018000 45 0F B6 0F 88 5D C0 4D 89 F5 89 D8 48 29 FA 44 E.¶.^]ÀM%õ%ØH)úD
00018010 89 CB 48 89 55 00 4C 89 FA 4D 89 F7 49 89 F6 48 %ÈH%U.L%úM%÷I%õH
00018020 89 D6 EB 11 0F 1F 40 00 41 0F B6 45 01 49 83 C5 %Öè...@.A.¶E.IfÀ
00018030 01 84 C0 74 33 38 D8 75 EF 49 89 F8 48 89 F2 4C .,Àt38ØuìI%õH%òL
00018040 89 E9 E8 D9 35 00 00 85 C0 75 DD 8B 55 E8 4C 03 %éèÛ5...ÀuÝ<UèL.
00018050 65 00 85 D2 74 12 49 01 FD 41 0F B6 45 00 84 C0 e...Òt.I.ÝA.¶E..À

```

Figure 8: hxd - hexadecimal dump of potential configuration block

Shown in the **GREEN** box, is the XOR key. Also within short proximity, shown in **BLUE** there are the sizes (in little endian<sup>2</sup>) 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 [here](#)

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

```

74: fcn.00000000 (int64_t arg3, int64_t arg4, int64_t arg6);
    ; arg int64_t arg3 @ rdx
    ; arg int64_t arg4 @ rcx
    ; arg int64_t arg6 @ r9
    ; var int64_t var_30h @ rsp+0x30
    0x00000000 4c8bdc mov r11, rsp
    ; DATA XREF from fcn.00000609 @ 0x8a9(r)
    0x00000003 4883ec68 sub rsp, 0x68
    0x00000007 33c0 xor eax, eax
    0x00000009 4983c9ff or r9, 0xffffffff ; arg6
    0x0000000d 498943e8 mov qword [r11 - 0x18], rax
    0x00000011 4533c0 xor r8d, r8d
    ; DATA XREF from fcn.00000609 @ 0x95c(r)
    0x00000014 498943e0 mov qword [r11 - 0x20], rax
    0x00000018 498943d8 mov qword [r11 - 0x28], rax
    0x0000001c 498943d0 mov qword [r11 - 0x30], rax
    ; DATA XREFS from fcn.00000609 @ 0xa06(r), 0xad9(r)
    0x00000020 89442430 mov dword [var_30h], eax
    ; DATA XREF from fcn.00000609 @ 0x70d(r)
    0x00000024 498953c0 mov qword [r11 - 0x40], rdx ; arg3
    ; DATA XREFS from fcn.00000609 @ 0x7de(r), 0x81b(r), 0xa1f(r), 0xb2b(r)
    0x00000028 ba00000010 mov edx, 0x10000000
    0x0000002d 49894bb8 mov qword [r11 - 0x48], rcx ; arg4
    0x00000031 498d4b18 lea rcx, [r11 + 0x18]
    0x00000035 49894318 mov qword [r11 + 0x18], rax
    0x00000039 48b8a5a4a3a2. movabs rax, 0xa1a2a3a4a5
    0x00000043 ffd0 call rax
    0x00000045 4883c468 add rsp, 0x68
    0x00000049 c3 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.

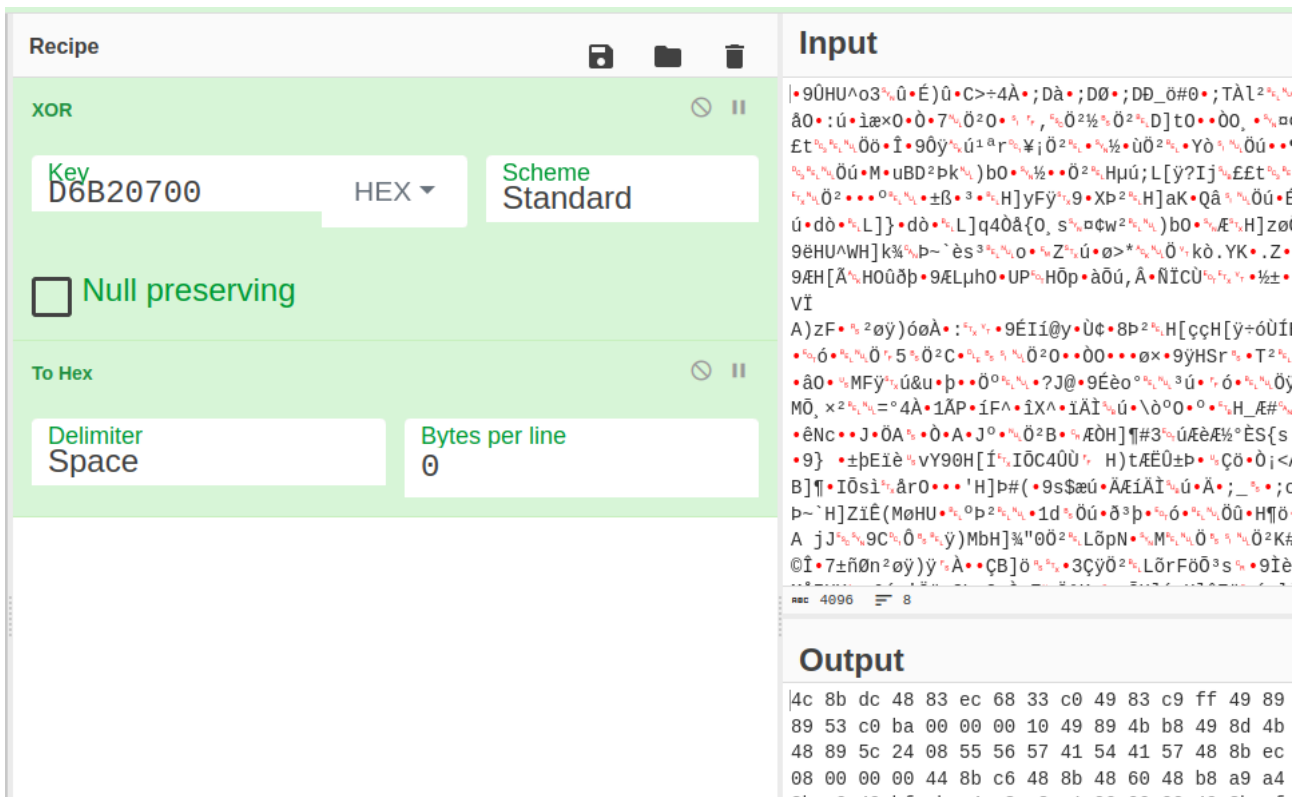


Figure 10: CyberChef - XOR routine.

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.

### 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 `VirtualProtect`, 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 `0x15BB` are located in the output of *Figure 8* underneath the `0x4A` 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
```

```

20: fcn.00000000 ();
   0x00000000 4ce3ab jrcxz 0xfffffffffffffae
   0x00000003 23e6 and esp, esi
   0x00000005 286925 sub byte [rcx + 0x25], ch
   0x00000008 be722b363e mov esi, 0x3e362b72 ; 'r+6>'
   0x0000000d 382a cmp byte [rdx], ch
   0x0000000f 3b31 cmp esi, dword [rcx]
   0x00000011 3428 xor al, 0x28
   0x00000013 61 invalid
   ; DATA XREF from fcn.00000000 @ +0x9f5(r)
   0x00000014 7364 jae 0x7a
   0x00000016 2be5 sub esp, ebp
   0x00000018 034fcf add ecx, dword [rdi - 0x31]
   0x0000001b 38e0 cmp al, ah
   0x0000001d 855733 test dword [rdi + 0x33], edx
   ; DATA XREFS from fcn.00000000 @ +0x6a3(r), +0x6bf(r), +0x12dd(r)
   0x00000020 336320 xor esp, dword [rbx + 0x20]
   ; DATA XREF from fcn.00000000 @ +0xdf7(r)
   0x00000023 e49a in al, 0x9a
   ; DATA XREF from fcn.00000000 @ +0x683(r)
   0x00000025 60 invalid
   ; DATA XREF from fcn.00000000 @ +0x1433(r)
   0x00000026 59 pop rcx
   ; XREFS: DATA 0x00000087 DATA 0x00000a8a DATA 0x00000c30 DATA 0x00001081 DATA 0x00001383 DATA 0x000015a9
   0x00000027 11d1 adc ecx, edx
   0x00000029 8e0cb3 mov cs, word [rbx + rsi*4]
   0x0000002c 5f pop rdi
   ; DATA XREFS from fcn.00000000 @ +0x41(x), +0xb0b(r), +0xf0a(r)
   0x0000002d f22e0 and ah, al
   0x00000030 96 xchg esi, eax
   ; DATA XREFS from fcn.00000000 @ +0x867(r), +0x144c(r)
   0x00000031 3dea997bb9 cmp eax, 0xb97b99ea
   ; DATA XREFS from fcn.00000000 @ +0xb54(r), +0xb8b(r)
   0x00000036 d9 invalid
   0x00000037 8be4 mov esp, esp

```

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
```

```

0x00000000 4c8bc4      mov r8, rsp                ; int64_t arg_28h
; DATA XREF from fcn.00000000 @ 0x580(r)
0x00000003 48895808   mov qword [rax + 8], rbx
0x00000007 4c894018   mov qword [rax + 0x18], r8
; DATA XREF from fcn.000005f4 @ 0x81b(r)
0x0000000b 55         push rbp
0x0000000c 56         push rsi                   ; arg2
0x0000000d 57         push rdi                   ; arg1
0x0000000e 4154      push r12
; DATA XREF from 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 488bf1     mov rsi, rcx                ; int64_t arg2
0x00000025 0f2970b8   movaps xmmword [rax - 0x48], xmm6
0x00000029 b93e803c9a mov ecx, 0x9a3c803e        ; int64_t arg_20h
0x0000002e 4d8bf9     mov r15, r9                 ; arg6
0x00000031 4d8bf0     mov r14, r8
0x00000034 4c8bea     mov r13, rdx                ; arg3
0x00000037 e88c100000 call fcn.000010c8
0x0000003c b91808cc67 mov ecx, 0x67cc0818        ; int64_t arg_20h
0x00000041 488945b0   mov qword [var_50h], rax
0x00000045 e87e100000 call fcn.000010c8
0x0000004a b91f1e5ad4 mov ecx, 0xd45a1e1f        ; int64_t arg_20h
0x0000004f 488bd8     mov rbx, rax
0x00000052 e871100000 call fcn.000010c8
0x00000057 65488b0c2530. mov rcx, qword gs:[0x30]
; CALL XREF from fcn.000008ac @ 0xfe0(x)
0x00000060 4883cfff   or rdi, 0xffffffffffffff
0x00000064 418b5708   mov edx, dword [r15 + 8]
; CALL XREFS from 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 [this](#) 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 fcn.00000000 @ 0x580(r)
0x00000003 48895808    mov qword [rax + 8], rbx
0x00000007 4c894018    mov qword [rax + 0x18], r8
; DATA XREF from fcn.000005f4 @ 0x81b(r)
0x0000000b 55          push rbp
0x0000000c 56          push rsi                  ; arg2
0x0000000d 57          push rdi                  ; arg1
0x0000000e 4154       push r12
; DATA XREF from fcn.000008ac @ 0xc13(r)
0x00000010 4155       push r13
0x00000012 4156       push r14
0x00000014 4157       push r15
0x00000016 488d6c24a0 lea rbp, [rsp - 0x60]
0x0000001b 4881ec60100 sub rsp, 0x160
0x00000022 488bf1     mov rsi, rcx              ; arg4
0x00000025 0f2970b8  movaps xmmword [rax - 0x48], xmm6
0x00000029 b93e803c9a mov ecx, 0x9a3c803e      ; int64_t arg_20h ; NtCreateThreadEx or ZwCreateThreadEx
0x0000002e 4d8bf9     mov r15, r9              ; arg6
0x00000031 4d8bf0     mov r14, r8
0x00000034 4c8bea     mov r13, rdx             ; arg3
0x00000037 e88c100000 call mw_resolve_api_hash_location
0x0000003c b91808cc67 mov ecx, 0x67cc0818      ; int64_t arg_20h ; RtlAllocateHeap
0x00000041 488945b0   mov qword [var_50h], rax
0x00000045 e87e100000 call mw_resolve_api_hash_location
0x0000004a b91f1e5ad4 mov ecx, 0xd45a1e1f      ; int64_t arg_20h ; RtlFreeHeap
0x0000004f 488bd8     mov rbx, rax
0x00000052 e871100000 call mw_resolve_api_hash_location
    
```

Figure 13: radere2 - API hashes being resolved.

Once the API's have been resolved, the routine `RtlAllocateHeap` <sup>9</sup> is called using the `call rbx` instruction, and `0x335B` (13147) bytes are requested.

00007FFB42D8F301	45:8bc4	mov r8d,r12d			
00007FFB42D8F304	44:8965 D8	mov dword ptr ss:[-rbp-28],r12d			
00007FFB42D8F308	48:8849 30	mov rcx,qword ptr ds:[rcx+30]			
00007FFB42D8F30C	FFD3	call rbx	call rtlAllocateHeap		
00007FFB42D8F30E	6548:880c25 60000000	mov rcx,qword ptr ds:[60]			
00007FFB42D8F317	48:8808	mov rbx,rax	rbx:RtlAllocateHeap, rax:RtlFree		
00007FFB42D8F31A	48:8945 D0	mov qword ptr ss:[-rbp-30],rax	rax:RtlFreeHeap		
00007FFB42D8F31E	45:33c0	xor r8d,r8d			
00007FFB42D8F321	48:8851 18	mov rdx,qword ptr ds:[rcx+18]			
00007FFB42D8F325	48:8842 10	mov rax,qword ptr ds:[rdx+10]	rax:RtlFreeHeap		
00007FFB42D8F329	48:8848 30	mov rcx,qword ptr ds:[rax+30]	qword ptr ds:[rax+30]:RtlFreeHea		
00007FFB42D8F32D	48:85c9	test rcx,rcx			
00007FFB42D8F330	74 34	je dpx.7FFB42D8F366			
00007FFB42D8F332	48:38CE	cmp rcx,rsi			
00007FFB42D8F335	74 05	je dpx.7FFB42D8F33C			
00007FFB42D8F337	48:8800	mov rax,qword ptr ds:[rax]	rax:RtlFreeHeap, qword ptr ds:[r		
00007FFB42D8F33A	EB ED	jmp dpx.7FFB42D8F329			
00007FFB42D8F33C	48:88c8	mov rcx,rbx	rbx:RtlAllocateHeap		
00007FFB42D8F33F	48:8850 50	mov rdx,qword ptr ds:[rax+50]	qword ptr ds:[rax+50]:RtlFreeHea		

Figure 14: x64dbg - RtlAllocate 0x335b Bytes

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

```

000001BF75AB9850 43 00 3A 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 \.r...d.l.l.".
000001BF75AB9890 43 00 3A 00 5C 00 57 00 69 00 6E 00 64 00 6F 00 C:.\w.i.n.d.o.
000001BF75AB98A0 77 00 73 00 5C 00 53 00 79 00 73 00 74 00 65 00 w.s.\S.y.s.t.e.
000001BF75AB98B0 6D 00 33 00 32 00 5C 00 72 00 75 00 6E 00 64 00 m.3.2.\r.u.n.d.
000001BF75AB98C0 6C 00 6C 00 33 00 32 00 2E 00 65 00 78 00 65 00 l.l.3.2...e.x.e.
000001BF75AB98D0 22 00 20 00 43 00 3A 00 5C 00 55 00 73 00 65 00 ".C:.\U.s.e.
000001BF75AB98E0 72 00 73 00 5C 00 6D 00 61 00 6C 00 77 00 61 00 r.s.\m.a.l.w.a.
000001BF75AB98F0 72 00 65 00 5C 00 44 00 65 00 73 00 6B 00 74 00 r.e.\D.e.s.k.t.
000001BF75AB9900 6F 00 70 00 5C 00 72 00 2E 00 64 00 6C 00 6C 00 o.p.\r...d.l.l.
000001BF75AB9910 20 00 76 00 63 00 61 00 62 00 20 00 2F 00 6B 00 .v.c.a.b. ./k.
000001BF75AB9920 20 00 63 00 68 00 6F 00 6B 00 6F 00 70 00 61 00 .c.h.o.k.o.p.a.
000001BF75AB9930 69 00 37 00 32 00 33 00 00 00 00 00 00 00 00 00 i.7.2.3.....
000001BF75AB9940 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

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`.

```

00007FFB42D90778 <dpx> 4C:894424 18 mov qword ptr ss:[rsp+18],r8 mw_memcpy
00007FFB42D9077D 48:895424 10 mov qword ptr ss:[rsp+10],rdx [qword ptr ss:[rsp+10]]:Rt1A
00007FFB42D90782 48:894C24 08 mov qword ptr ss:[rsp+8],rcx [qword ptr ss:[rsp+8]]:ti_p
00007FFB42D90787 48:83EC 18 sub rsp,18 [qword ptr ss:[rsp]]:DpxRest
00007FFB42D9078B 48:8B4424 20 mov rax,qword ptr ss:[rsp+20] [qword ptr ss:[rsp+20]]:ti_p
00007FFB42D90790 48:890424 mov qword ptr ss:[rsp],rax [qword ptr ss:[rsp]]:DpxRest
00007FFB42D90794 48:8B4424 28 mov rax,qword ptr ss:[rsp+28]
00007FFB42D90799 48:894424 08 mov qword ptr ss:[rsp+8],rax
00007FFB42D9079E 48:837C24 30 00 cmp qword ptr ss:[rsp+30],0 number of bytes to copy comp
00007FFB42D907A4 74 34 je dpx.7FFB42D907DA [qword ptr ss:[rsp]]:DpxRest
00007FFB42D907A6 48:8B0424 mov rax,qword ptr ss:[rsp] [qword ptr ss:[rsp]]:DpxRest
00007FFB42D907AA 48:8B4C24 08 mov rcx,qword ptr ss:[rsp+8] ptr source
00007FFB42D907AF 8A09 mov byte ptr ds:[rcx] store 1 byte in RCX LSB
00007FFB42D907B1 8B08 mov byte ptr ds:[rax],cl ptr destination
00007FFB42D907B3 48:8B0424 mov rax,qword ptr ss:[rsp] [qword ptr ss:[rsp]]:DpxRest
00007FFB42D907B7 48:FFC0 inc rax increment index 1 byte
00007FFB42D907BA 48:890424 mov qword ptr ss:[rsp],rax store index back on stack
00007FFB42D907BE 48:8B4424 08 mov rax,qword ptr ss:[rsp+8]
00007FFB42D907C3 48:FFC0 inc rax
00007FFB42D907C6 48:894424 08 mov qword ptr ss:[rsp+8],rax
00007FFB42D907CB 48:8B4424 30 mov rax,qword ptr ss:[rsp+30]
00007FFB42D907D0 48:FFC8 dec rax decrement number of bytes re
00007FFB42D907D3 48:894424 30 mov qword ptr ss:[rsp+30],rax store bytes remaining count
00007FFB42D907D8 EB C4 jmp dpx.7FFB42D9079E [qword ptr ss:[rsp+20]]:ti_p
00007FFB42D907DA 48:8B4424 20 mov rax,qword ptr ss:[rsp+20]
00007FFB42D907DF 48:83C4 18 add rsp,18

```

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 `0x0814E394BBE` (file offset `0x2BBBE`) into memory region 1, where it 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 `memcpy` 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`

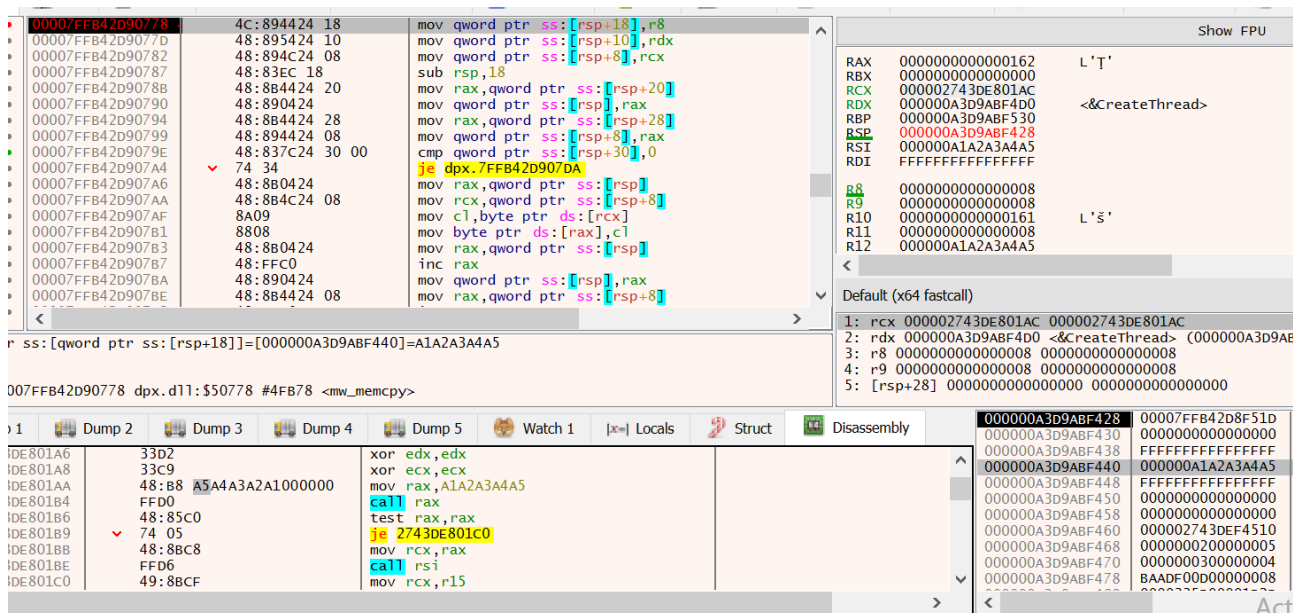


Figure 17: x64dbg - Shell code patching routine, before patch.

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

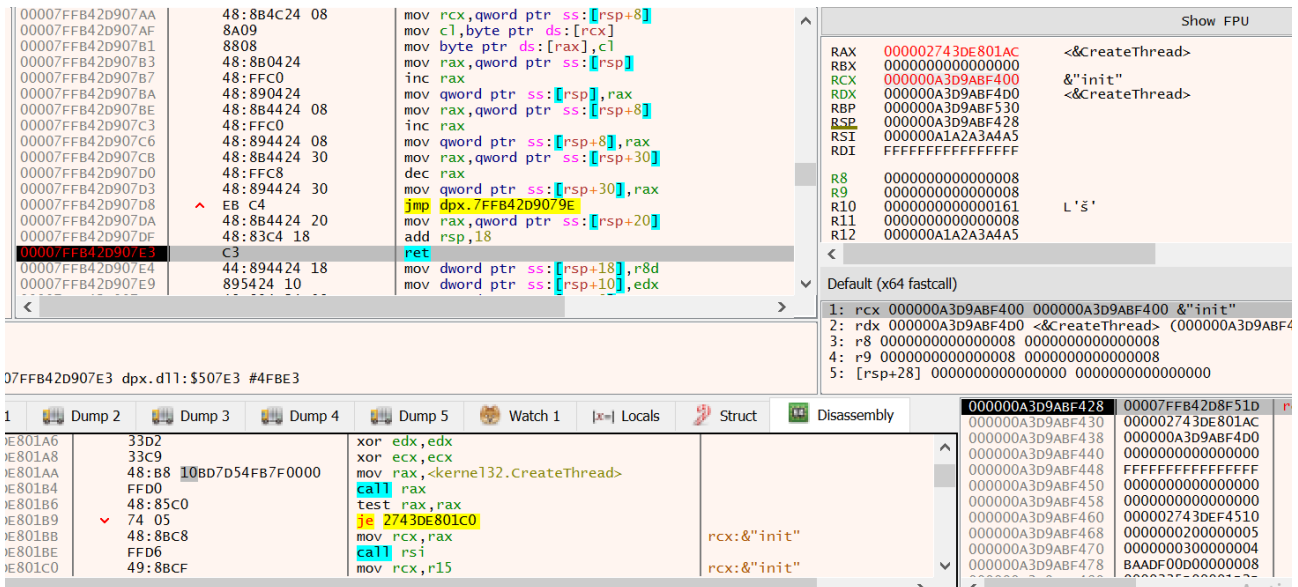


Figure 18: x64dbg - Shell code patching routine, after patch.

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 `0xB` (11) bytes from a stack based address into a location within the first allocated memory region.

```
4C 8B D1 B8 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<sup>10</sup>](#) recipe to `Disassemble 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 [here](http://www.ired.team) over at [www.ired.team](http://www.ired.team)

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.

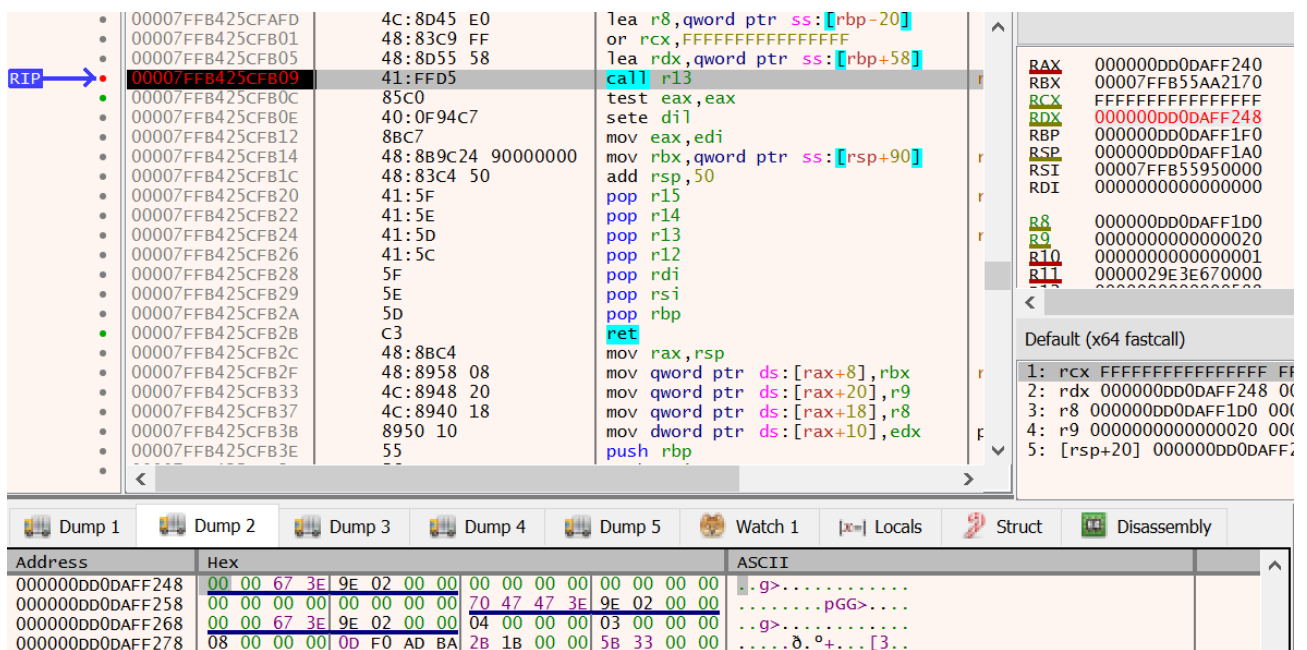


Figure 19: x64dbg - `ZwProtectVirtualMemory` from `R13` register

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 `0x5` 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 `0x36` resolves to `NtQuerySystemInformation` [11](#)

Taking a look at the documentation for `NtQuerySystemInformation` [here](#) 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 [here](#)

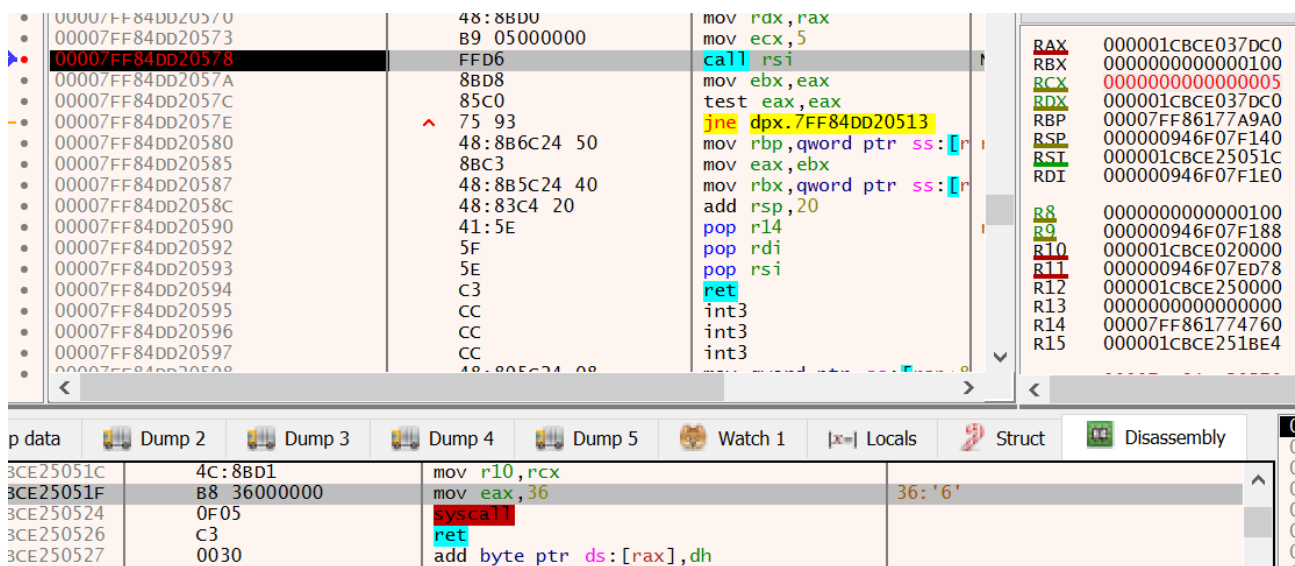


Figure 20: x64dbg - NtQuerySystemInformation native syscall

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.

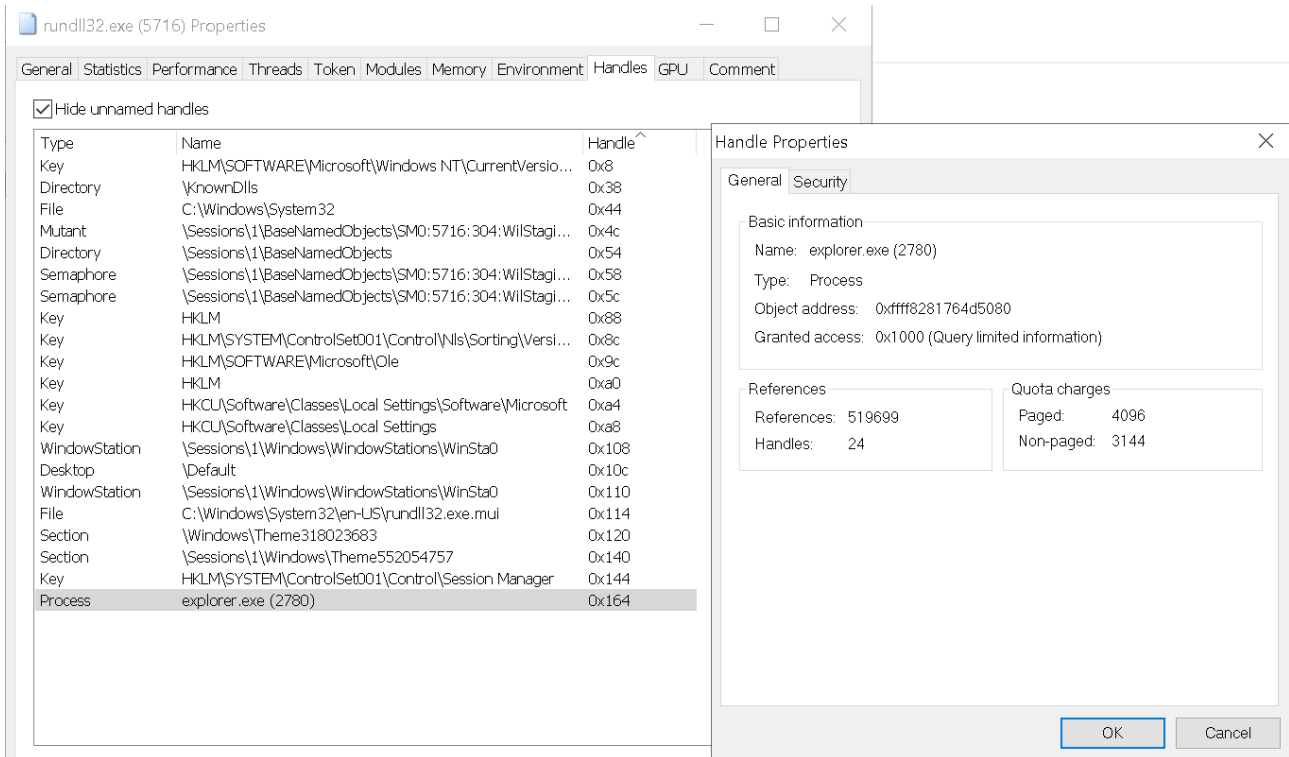


Figure 21: ProcessHacker - Handle to explorer process opened.

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

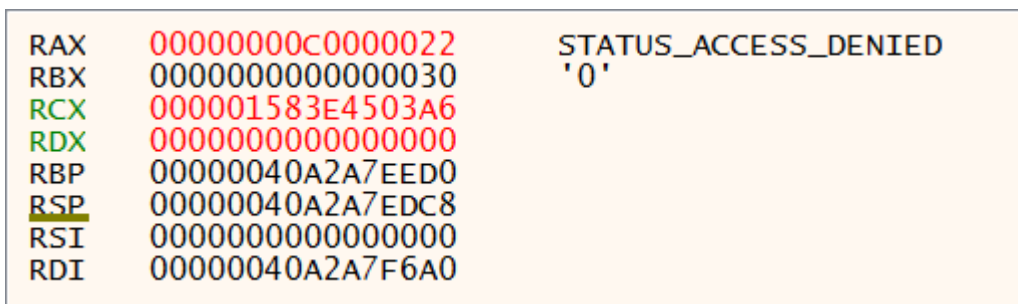


Figure 22: x64dbg - NtOpenProcess Access Denied.

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

Once a handle to an `svchost.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 `svchost.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 180B0000000000	mov r15, B18
000001583E4400A0	49:8BCF	mov rcx, r15
000001583E4400A3	48:B8 50CC7C5FF87F00	mov rax, <kernel32.ReadProcessMemory>
000001583E4400AD	44:8BCE	mov r9d, esi
000001583E4400B0	48:8BD3	mov rdx, rbx
000001583E4400B3	FFD0	call rax

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:8Bb1	mov r10, rcx	50: 'p'
000001583E45078F	B8 50000000	mov eax, 50	NTProtectVirtualMemory
000001583E450794	0F05	syscall	
000001583E450796	C3	ret	
000001583E450797	0070 DA	add byte ptr ds:[rax-26], dh	
000001583E45079A	7E 61	jle 1583E4507FD	
000001583E45079C	F8	c1c	
000001583E45079D	7F 00	jg 1583E45079F	
000001583E45079F	00c8	add b1, c1	
000001583E4507A1	4A:94	xcfg rsp, rax	
000001583E4507A3	894C8B D1	mov dword ptr ds:[rbx+rcx*4-2F],	
000001583E4507A7	B8 51000000	mov eax, 51	51: 'q'
000001583E4507AC	0F05	syscall	
000001583E4507AE	C3	ret	
000001583E4507AF	0090 DA7E61F8	add byte ptr ds:[rax-79E8126], d	
000001583E4507B5	7F 00	jg 1583E4507B7	
000001583E4507B7	00A1 41E0744C	add byte ptr ds:[rcx+4C74E041], a	
000001583E4507BD	8BD1	mov edx, ecx	

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.

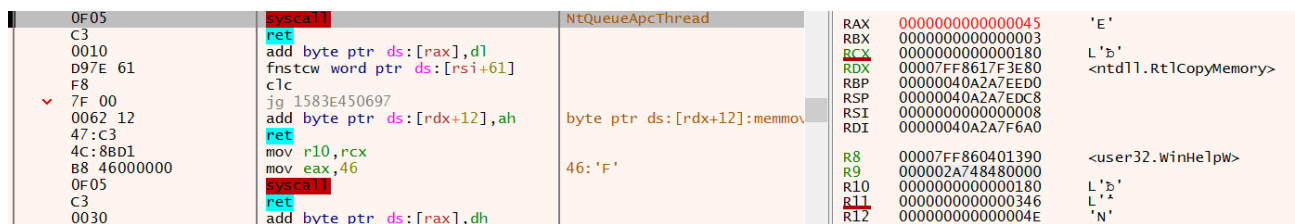


Figure 25: x64dbg - `WinHelpW` execution after `NtDelayExecution`

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 `svchost.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

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

```

74: fcn.00000000 (int64_t arg3, int64_t arg4, int64_t arg6);
    ; arg int64_t arg3 @ rdx
    ; arg int64_t arg4 @ rcx
    ; arg int64_t arg6 @ r9
    ; var int64_t var_30h @ rsp+0x30
    0x00000000      4c8bdc      mov r11, rsp
    0x00000003      4883ec68    sub rsp, 0x68
    0x00000007      33c0        xor eax, eax
    0x00000009      4983c9ff    or r9, 0xffffffff ; arg6
    ; DATA XREF from _x0000004f @ +0x219(r)
    0x0000000d      498943e8    mov qword [r11 - 0x18], rax
    0x00000011      4533c0      xor r8d, r8d
    0x00000014      498943e0    mov qword [r11 - 0x20], rax
    0x00000018      498943d8    mov qword [r11 - 0x28], rax
    0x0000001c      498943d0    mov qword [r11 - 0x30], rax
    0x00000020      89442430    mov dword [var_30h], eax
    0x00000024      498953c0    mov qword [r11 - 0x40], rdx ; arg3
    0x00000028      ba00000010 mov edx, 0x10000000
    0x0000002d      49894bb8    mov qword [r11 - 0x48], rcx ; arg4
    0x00000031      498d4b18    lea rcx, [r11 + 0x18]
    0x00000035      49894318    mov qword [r11 + 0x18], rax
    0x00000039      48b880e87e61. movabs rax, 0x7ff8617ee880
    0x00000043      ffd0        call rax ; ZwCreateThreadEx
    0x00000045      4883c468    add rsp, 0x68
    0x00000049      c3          ret
[0x00000000]>

```

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.

The screenshot shows the assembly view of the `ReadProcessMemory` function in `kernel32.dll`. The instruction list on the left shows a series of `int3` instructions. The registers window on the right shows the following values:

Register	Value	Comment
RAX	00007FFD4AB0CC50	<kernel32.ReadProcessMemory>
RBX	00000290552B9840	
RCX	0000000000000470	L'v'
RDX	00000290552B9840	
RSP	000000CD1B38F9F0	
RSI	000000000000335B	
RDI	0000023BF968BF70	
R8	0000023BF968BF70	
R9	000000000000335B	
R10	0000000000001FFF	
R11	0000000000000000	
R12	0000000000000000	
R13	0000000000000000	
R14	0000000000000000	
R15	0000000000000470	L'v'

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	00 00 00 00	init.....
0000023BF968C780	00 00 00 00	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	00 00 00 00	.....
0000023BF968C7A0	00 00 00 00	00 00 00 00	A0 A0 2B 55	90 02 00 00	00 00 00 00	..... +U....
0000023BF968C7B0	AD 1E 00 00	00 00 00 00	D6 B2 07 00	00 00 00 00	00 00 00 00	..... Ö².....
0000023BF968C7C0	4D BF 2B 55	90 02 00 00	4E 0C 00 00	00 00 00 00	00 00 00 00	M¿+U....N.....
0000023BF968C7D0	70 57 38 21	21 F7 45 0E	0E D8 6A EB	BE E8 0D B0		pw8!!÷E...øje¾è.°
0000023BF968C7E0	20 F6 41 46	46 90 1E 09	0A DD 6F 68	B0 46 2E 29		öAFF....Ýoh°F.)
0000023BF968C7F0	E9 2D 9D AF	AE 78 FA ED	DF EA 9E 8B	8B A9 12 48		é-.™xúíßè...@.H
0000023BF968C800	27 F1 43 44	C4 5A E7 94	C7 F0 EE 81	18 5B C9 5E		'ñCDÄZç.Çðî...[ÉÁ

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
```

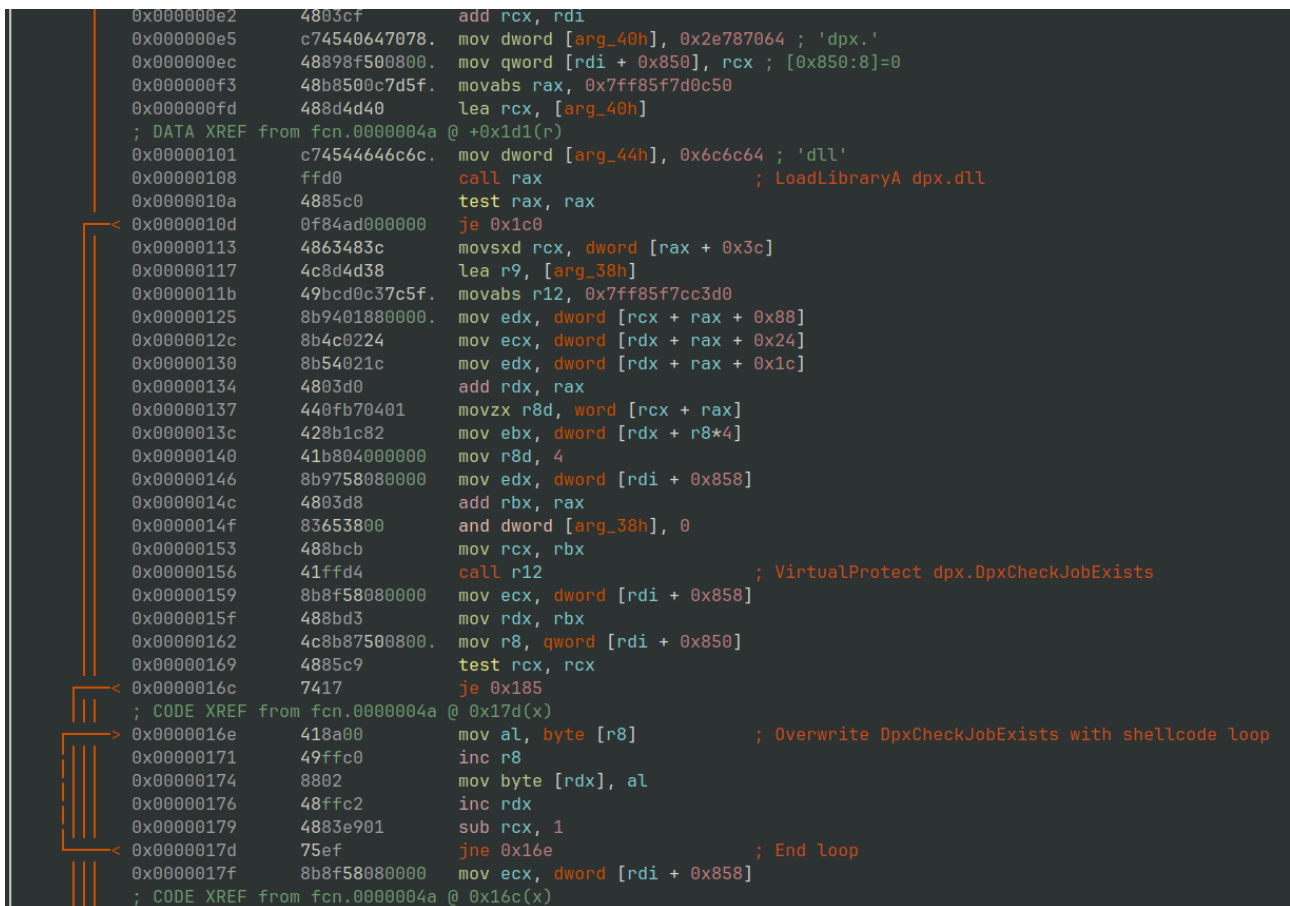


Figure 29: radare2 - LoadLibraryA dpx.dll and overwrite DpxCheckJobExists

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 off 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 fcn.00000000 @ 0xc9(x)
0x0000005b 488bc1 mov rax, rcx
0x0000005e 488d7101 lea rsi, [rcx + 1]
0x00000062 4899 cqp
0x00000064 49f7fe idiv r14
0x00000067 488bc1 mov rax, rcx
0x0000006a 4c63da movsxd r11, edx
0x0000006d 4899 cqp
0x0000006f 83e203 and edx, 3
0x00000072 4803c2 add rax, rdx
0x00000075 83e003 and eax, 3
0x00000078 482bc2 sub rax, rdx
0x0000007b 4863c8 movsxd rcx, eax
0x0000007e 430fb6040b movzx eax, byte [r11 + r9]
0x00000083 440fb6841948. movzx r8d, byte [rcx + rbx + 0x848]
0x0000008c 4433c0 xor r8d, eax
0x0000008f 488bc6 mov rax, rsi
0x00000092 4899 cqp
0x00000094 49f7fe idiv r14
0x00000097 4863c2 movsxd rax, edx
0x0000009a 420fb60c08 movzx ecx, byte [rax + r9]
0x0000009f 442bc1 sub r8d, ecx
0x000000a2 4181c0000100. add r8d, 0x100
0x000000a9 4181e0ff0000. and r8d, 0x800000ff
0x000000b0 7d0d jge 0xbf
0x000000b2 41ffc8 dec r8d
0x000000b5 4181c800ffff. or r8d, 0xffffffff ; 4294967040
0x000000bc 41ffc0 inc r8d
; CODE XREF from fcn.00000000 @ 0xb0(x)
0x000000bf 4788040b mov byte [r11 + r9], r8b
0x000000c3 488bce mov rcx, rsi
0x000000c6 493bf2 cmp rsi, r10
0x000000c9 7e90 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	.....4.....-MZ..
000001AC4A052021	03 00 00 00	04 10 FF FF	00 00 B8 20	C6 00 40 12	.....ÿÿ...Æ.@.
000001AC4A052031	02 EB 01 00	D0 10 0E 1F	BA 0E 00 B4	09 CD 21 00	.ë...ð...°...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....\$..!É...e~
000001AC4A052081	C0 16 01 42	6E 05 C0 82	88 A0 B0 67	20 16 CA 7F	À..Bñ..À...°g.Ê.
000001AC4A052091	C1 6E 0A 05	7F C0 4F 20	83 CC 7A 0A	04 83 CC 7E	Án...ÀO .İz...İ~
000001AC4A0520A1	C1 64 0A 02	7C 0A 02 52	69 63 68 06	0E 16 25 50	Ád... ..Rich...%P
000001AC4A0520B1	00 08 40 A9	45 00 00 64	86 07 00 9A	9F 87 63 16	..@F...d.....c.

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		RAX	FFFFFFFFFFFFFF00
00007FFB14FF8C70	jmp	qword ptr ds:[<VirtualAlloc>]	RBX	0000000000008200
00007FFB14FF8C77	int3		RCX	0000000180000000
00007FFB14FF8C78	int3		RDX	0000000000009000
00007FFB14FF8C79	int3		RBP	0000006CD797FBE0
00007FFB14FF8C7A	int3		RSP	0000006CD797FB58
00007FFB14FF8C7B	int3		RSI	0000000000009000
00007FFB14FF8C7C	int3		RDI	0000020E36CB5110
00007FFB14FF8C7D	int3		R8	0000000000003000
00007FFB14FF8C7E	int3		R9	0000000000000004
00007FFB14FF8C7F	int3		R10	0000000000000000
00007FFB14FF8C80	int3		R11	0000000000000246
00007FFB14FF8C81	int3		R12	00007FFB15CD7BD0
00007FFB14FF8C82	int3		R13	00007FFB14FFB630
00007FFB14FF8C83	int3		R14	00007FFB14FF8C70
00007FFB14FF8C84	mov	rax, rsp	R15	0000020E36CB5040
00007FFB14FF8C87	mov	qword ptr ds:[rax+8], rbx	RIP	00007FFB14FF8C70

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.

00007FFB14FFBD00	int3			RAX	0000000180001378
00007FFB14FFBD0E	int3			RBX	0000000180000000
00007FFB14FFBD0F	int3			RCX	0000000000000000
00007FFB14FFBD10	mov	r11, rsp	CreateThread	RDX	0000000000000000
00007FFB14FFBD13	sub	rsp, 48		RBP	0000002409E7F960
00007FFB14FFBD17	mov	r10d, dword ptr ss:[rsp+70]		RSP	0000002409E7F898
00007FFB14FFBD1C	mov	rax, qword ptr ss:[rsp+78]	[qword ptr ss:[rs	RSI	00000001800000D0
00007FFB14FFBD21	and	r10d, 10004		RDI	0000000180000300
00007FFB14FFBD28	mov	qword ptr ds:[r11-10], rax		R8	00000001800028D4
00007FFB14FFBD2C	and	qword ptr ds:[r11-18], 0		R9	0000000000000000
00007FFB14FFBD31	mov	dword ptr ds:[r11-20], r10d		R10	0000000000000000
00007FFB14FFBD35	mov	qword ptr ds:[r11-28], r9		R11	0000000000000246
00007FFB14FFBD39	mov	r9, r8		R12	0000000000000000
00007FFB14FFBD3C	mov	r8, rdx		R13	0000000000000001
00007FFB14FFBD3F	mov	rdx, rcx		R14	0000000000000007
				R15	0000000004000000

r11=246 L'g'  
rsp=0000002409E7F898

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-Bear<sup>12</sup>.

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](#).

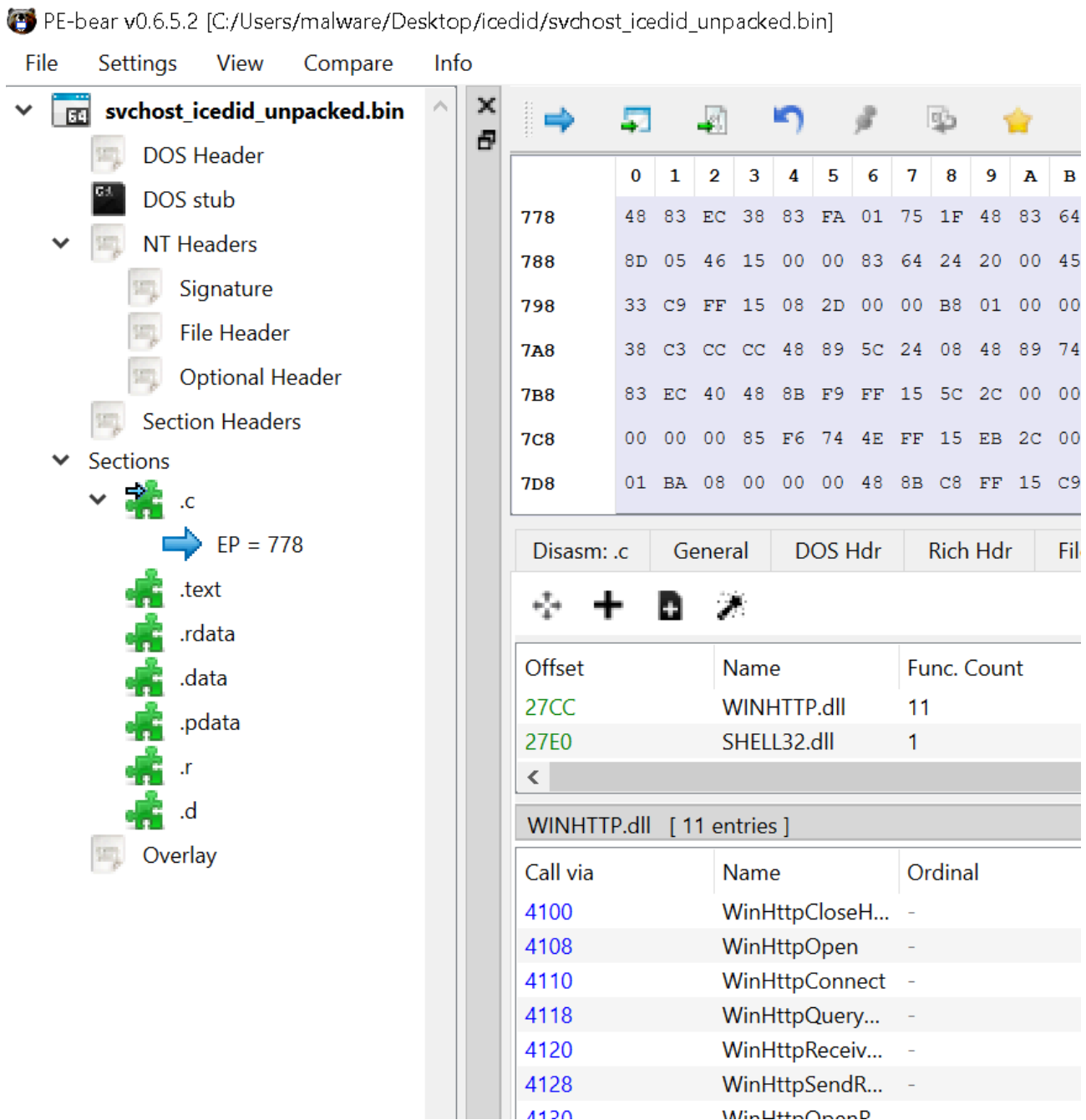


Figure 35: PE Bear - Unpacked icedid payload from svchost.exe

## 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.

[@techevo](#)

## References

Source: <https://blog.techevo.uk/analysis/binary/2024/03/17/carving-the-icedid-part-3.html>