DarkGate: From AutoIT to Shellcode Execution

vmray.com/cyber-security-blog/darkgate-from-autoit-to-shellcode-execution/

Join us as we explore the malicious tactics and activities of the DarkGate malware family.

05 January 2024

Table of Contents

Overview

The DarkGate malware family is known for its variety of features including the download and execution of malicious payloads, information stealing and keylogging abilities, as well as employing multiple evasion techniques. It is being sold as a service to cybercriminals and

has been active since at least 2018, but only recently gained in popularity after the Qakbot infrastructure was taken down by law enforcement. What stands out is its rather complex delivery methods and multitude of evasion tactics to avoid detection, one of which is the abuse if AutoIt scripts to execute native code and not just commands.

AutoIt, commonly used to automate tasks within the Windows environment, such as simulating mouse clicks or keystrokes on the GUI, is abused to execute a malicious shellcode in DarkGate's hands. This technique attempts to let the malware operate under the radar by betting on static analysis tools inability to parse compiled and obfuscated AutoIt scripts.

Recently, we have taken an in-depth look into the DarkGate malware family to gain insights into the inner workings of this malware family as well as to improve detection and configuration extraction. In this blog post, we want to specifically highlight the interesting way by which DarkGate accomplishes executing malicious native code via AutoIt scripts.

Infection chain of DarkGate

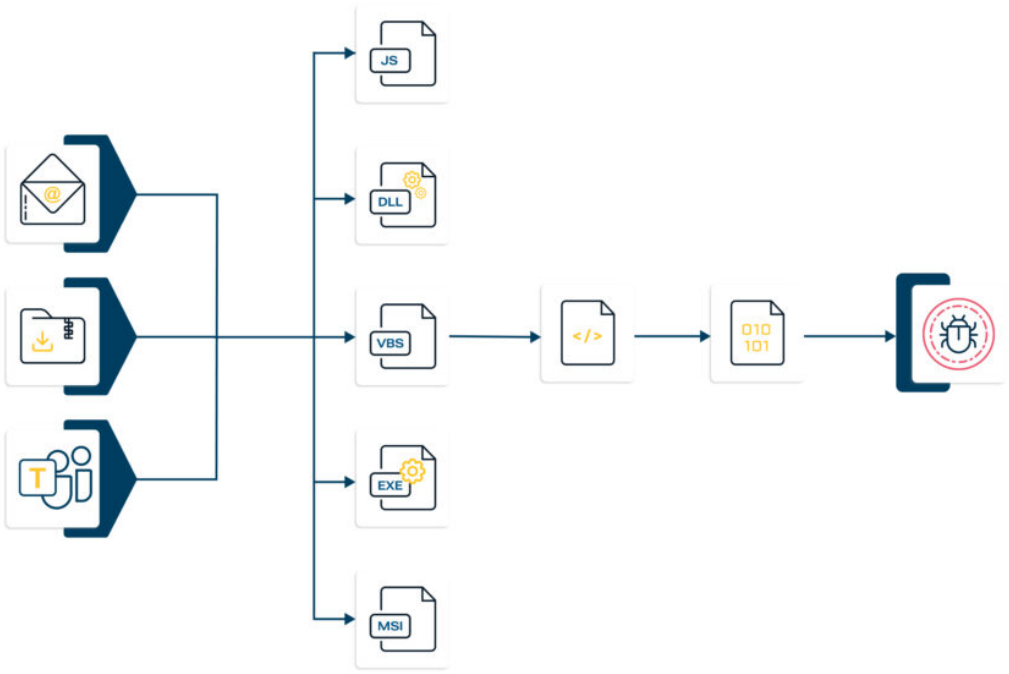

Figure 1: Overview of the DarkGate delivery chain we have spotted so far.

DarkGate's infection chain can start with multitude of file types, including DLLs, JScript, VBScript, EXE and MSI files (see Figure 1 for a common delivery chain).

This visualization highlights the journey from the initial delivery file to the subsequent stages. It progresses through the AutoIt interpreter to the execution of shellcode, ending in the execution of the actual DarkGate Loader.

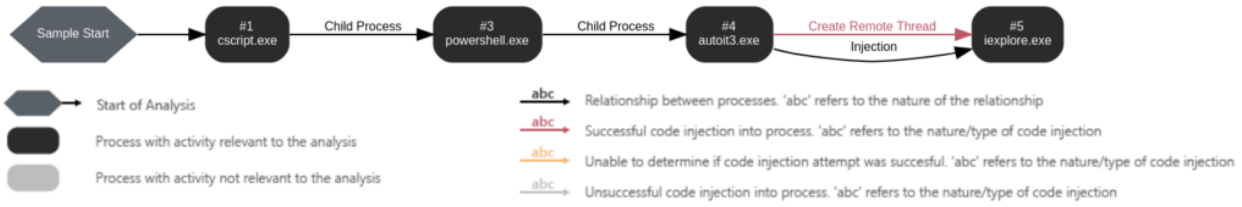

Figure 2: Execution chain from the initial JS file, to the powershell downloader, finally landing in the AutoIt interpreter where the script executes it's shellcode.

Differing from typical malware tactics, DarkGate does not use AutoIt scripts to execute new commands (such as cmd) as evidenced by our process graph in Figure 2. This leads us to the intriguing question: how exactly does DarkGate execute its malicious code?

An important clue is in our process graph, which tells us that the AutoIt interpreter injects code into another process.

DarkGate Using AutoIt Scripts

The choice of AutoIt by DarkGate's developers is strategic: Our investigations reveal that the malware often employs compiled and protected AutoIt scripts, which are additionally obfuscated to further cloak their malicious intent. This level of protection makes it challenging to dissect and understand the malware's inner workings for researchers and static analysis tools alike.

By utilizing tools such as myAut2Exe or Binary Refinery, we can extract the original source code from these obfuscated scripts. This process, albeit requiring some clean-up, does produce a readable source code for our manual reverse engineering purposes. The deobfuscated code in Figure 3 provides us with a pivotal insight into DarkGate's operation: The malware utilizes specific Windows API functions, notably EnumWindows, but in other samples we have also seen a call to CallWindowProc.

DarkGate AutoIt Script

```
#NoTrayIcon
$shellcode="90E9B903000000787653697l4D5061526F4B78..."
$shellcode_buffer=DLLSTRUCTCREATE("byte["&48902&"]")
IF NOT FILEEXISTS("C:\Program Files (x86)\Sophos") THEN
    DllCall("kernel32.dll", "BOOL", "VirtualProtect", "ptr", DllStructGetPtr($shellcode_buffer), "int",
        48902, "dword", 0x40, "dword*", null)
ENDIF
DllStructSetData($shellcode_buffer, 1, BinaryToString("0x"&$shellcode))
DllCall("user32.dll", "int", "EnumWindows", "ptr", DllStructGetPtr($shellcode_buffer), "lparam", 0)
```

Figure 3: Deobfuscated and simplified AutoIt script

DarkGate's Shellcode Execution

The aforementioned API functions, while typically used for legitimate purposes, are repurposed by DarkGate to execute its malicious payload.

CallWindowProc is typically used for customizing actions in a Windows GUI, like modifying button functionality. However, DarkGate calls this function while pointing the first parameter, lpPrevWndFunc, to its shellcode. In effect, Windows then executes the malicious shellcode as if it were a window procedure. This seems to be a known workaround to execute native code via AutoIt scripts at least since 2008.

In some variants of DarkGate, EnumWindows is abused instead, which is a legitimate API for enumerating top-level windows. This function is designed to execute a specified callback function for each window, but DarkGate sets the callback function address to it's shellcode location. Given that there's almost always at least one open window, this ensures the execution of the malicious shellcode at least once.

Any callback-based Windows API function could potentially be abused in a similar way, but specifically executing native code via EnumWindows in AutoIt seems to be new and unique to DarkGate as far as we are aware. While all of this may be hard for static analysis tools to extract, behavior-based analysis allows one to capture this in action. Our execution logs (see Figure 4) clearly show the runtime execution of EnumWindows and the following call to LoadLibraryA executed by the shellcode.

```
[0108.636] RtlAllocateHeap (HeapHandle=0x610000, Flags=0x0, Size=0xc) returned 0x99eeb0
[0108.636] WideCharToMultiByte (in: CodePage=0x0, dwFlags=0x0, lpWideCharStr="EnumWindows", cchWideChar=12,
[0108.636] EnumWindows (lpEnumFunc=0x9a5e78, lParam=0x0)
[0108.640] LoadLibraryA (lpLibFileName="kernel32.dll") returned 0x75c80000
```

Figure 4: VMRay's function log captures runtime execution of EnumWindows by the AutoIt script.

Variants of DarkGate

To investigate this further, we have manually selected multiple DarkGate samples dating back to it's initial version in 2018. Through manual clustering based on code similarities, we've identified four distinct variants:

1. **First Variant:** This variant embeds the payload within the compiled AutoIt script, encrypted using XOR and surrounded by the "padoru" keyword. It specifically checks for the presence of Sophos antivirus software and leverages the VirtualProtect call to make the shellcode memory region executable and uses the CallWindowProc API to execute the shellcode.
2. **Second Variant:** Here, the payload is scattered throughout the AutoIt source code as hex codes, which is put together at runtime. This variant switches its strategy to abuse the EnumWindows API instead of CallWindowProc.

3. **Third Variant:** This is similar to the others but with a key difference: it checks if it is running with SYSTEM privileges.
4. **Fourth Variant:** This one is from 2018, has much less complexity as it contains no obfuscation. It creates a shortcut (LNK) to the AU3 file placed in the startup directory and reads the shellcode from a previously dropped ‘shell.txt’ file. Like the first variant, it also abuses CallWindowProc.

We have also noticed that there are differences in how the shellcode was implemented, which we will briefly look into next.

Payloads in DarkGate

The payloads in DarkGate’s various samples typically follow a similar mechanism, primarily focusing on loading the next stage of the malware, which is often tasked with downloading the final DarkGate malware.

One notable technique observed in these payloads is the byte-by-byte construction of the code using the mov instruction, a method likely adopted to evade detection by scanning tools before runtime extraction (see Figure 5).

```
jnz     short loc_9
mov     eax, [rbp-4]
add     esp, 0FFFFF5ACh
push    rbx
push    rsi
push    rdi
lea     eax, [rbp-3A56h]
mov     byte ptr [rax], 4Dh ; 'M'
mov     byte ptr [rax+1], 5Ah ; 'Z'
mov     byte ptr [rax+2], 50h ; 'P'
mov     byte ptr [rax+3], 0
mov     byte ptr [rax+4], 2
mov     byte ptr [rax+5], 0
mov     byte ptr [rax+6], 0
mov     byte ptr [rax+7], 0
mov     byte ptr [rax+8], 4
mov     byte ptr [rax+9], 0
mov     byte ptr [rax+0Ah], 0Fh
mov     byte ptr [rax+0Bh], 0
mov     byte ptr [rax+0Ch], 0FFh
mov     byte ptr [rax+0Dh], 0FFh
```

Figure 5: The shellcode copies a PE file into a memory location byte-by-byte without loops, potentially to avoid detection.

Additionally, some payloads exhibit a deliberate pattern of jumping around the code (see Figure 6).

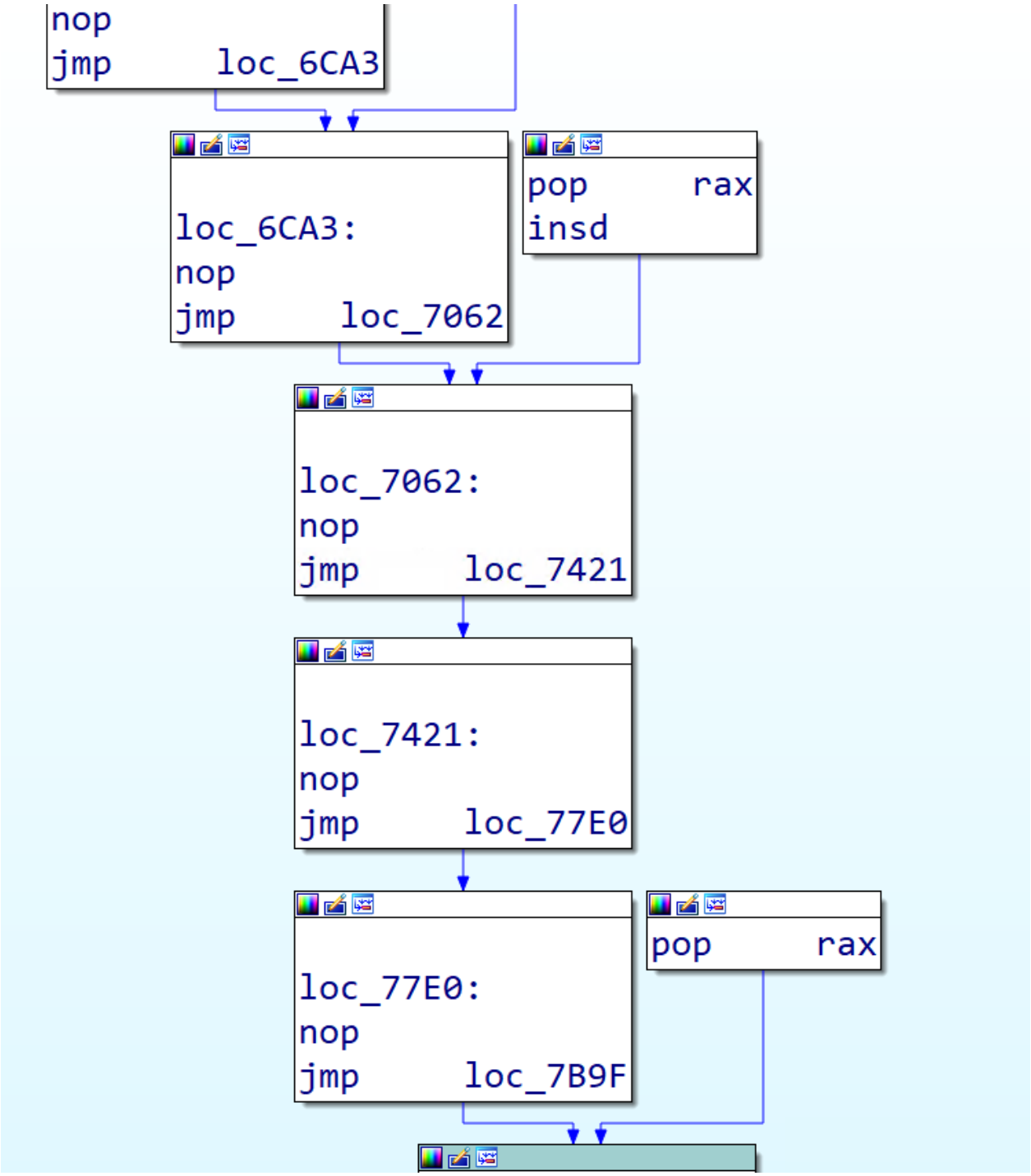

Figure 6: Obfuscated shellcode dropped by DarkGate

This complexity is designed to hinder manual analysis, making it more challenging to dissect and understand the malware's functionality and intent. For dynamic, behavior-based analysis solutions such as VMRay's Platform, none of these obfuscation attempts can hide the malicious actions taken by the sample.

In particular, while such intricacies in the payloads underscore the stealth and sophistication embedded in DarkGate's design, our dynamic approach reveals the executed functions in the function log just the same, regardless of any obfuscation attempts such as jumping

around, calling native functions via AutoIt scripts or employing multi-stage payloads spread over different memory regions.

Conclusion

Despite DarkGate's extensive obfuscation efforts, dynamic, behavior-based analysis proves to be a helpful tool in identifying and understanding this malware. By not solely relying on static analysis, it's possible to trace the entire code execution journey – from initial infection, through the AutoIt3 interpreter stage, to the injection, and finally to the actual DarkGate malware, culminating in the extraction of its configuration.

This case study highlights the lengths to which attackers will go, continually exploring obscure methods to deliver their malware and challenge existing security solutions.

References

IOCs

Hashes:

- 754d7afb2c3454d86ded95668c74c119c5ec4465
- 18f49619d69b057e81163bdf08eab5f355ce662c
- 5629b3684d406e431c6f41c5df56455c3b944c41
- 47718e8df5e7a0d0b2c74f10696ca50cf6e1e0b9
- bb0f4a60bbd8256e42f57d8b0b1269f2ec855428
- eabcef1e27b7452c74acfa0f201e9a937b0dee6d
- c46e52b896bf3b53a6878d2b2386a9dc40377f19
- 29b6a8ae869cdc1a95bae83dd97874e5efa79613
- d25e55d1eed18e55557ee9da7d195748dd2814f0
- 2e0d4798c12a7d71ad45a621dddb750bae0cd23b
- edc5d0dc190dcd0e031e2c5b43026fd3a61caed0
- c90d572f7f160dd8a3ae6e825eeb2a9d6628cef5
- 0d47cbd6d19a17a57077cbc0d0aa659865458672
- f68cc52f19c11d07d72118e71919df20ffabe9f2

URLs:

- hxxp://adhufdauifadhj13[.]com:2351
- hxxp://sftp.bitepieces[.]com:443
- hxxp://sftp.noheroway[.]com:443
- hxxp://saintelzearlava[.]com:80
- hxxp://trans1ategooglecom[.]com:80
- hxxp://sanibroadbandcommunicton[.]duckdns[.]org:5864

- hxxp://faststroygo[.]com

Emre Güler
Threat Researcher