GuLoader: Peering Into a Shellcode-based Downloader | CrowdStrike

By Umesh Wanve

Archived: 2026-04-05 20:25:00 UTC

GuLoader, a malware family that emerged in the wild late last year, is written in Visual Basic 6 (VB6), which is just a wrapper for a core payload that is implemented as a shellcode. It is distributed via spam email campaigns with archived attachments that contain the malware. The majority of malware downloaded by GuLoader is commodity malware, with AgentTesla, FormBook and NanoCore being the most predominant.

This downloader typically stores its encrypted payloads on Google Drive. CrowdStrike has observed that GuLoader downloads its payloads from Microsoft OneDrive and also from compromised or attacker-controlled websites. By utilizing legitimate file-sharing websites, GuLoader can evade network-based detection, as these services are not generally filtered or inspected in corporate environments. In addition, the downloaded payloads are encrypted with a hard-coded XOR key embedded in the malware, making it difficult for file-sharing service providers to identify the payload as malicious. GuLoader is an advanced downloader that uses shellcode wrapped in a VB6 executable that changes in each campaign to evade antivirus (AV) detections. The shellcode itself is encrypted and later heavily obfuscated, making static analysis difficult. In this blog, we cover GuLoader's internal details, including its main shellcode, anti-analysis techniques and final payload delivery mechanism.

Analysis

GuLoader is often distributed through spam campaigns that contain the malware embedded in archived attachments. An example of GuLoader spam email is shown in Figure 1.

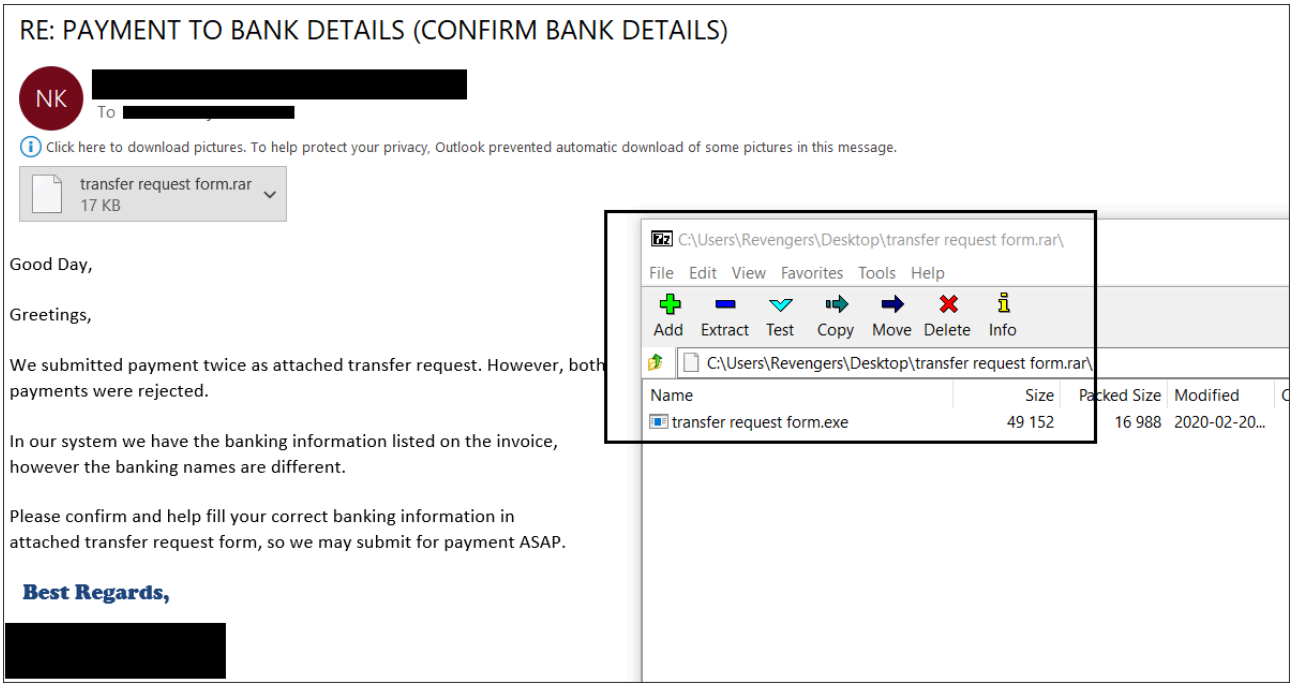


Figure 1: Sample spam email with RAR attachment (click image to enlarge)

The attachment contains a malicious executable file named `transfer request form.exe`. The sample is a PE32 file written in Microsoft Visual Basic (just a wrapper for a shellcode that implements the main functionality), as shown in Figure 2. Strings present inside the sample don't reveal much as the binary is packed. The sample contains numerous calls to meaningless VB functions that can slow down the analysis. By stepping through the assembly code, we will land into some block of code that is eventually used to decrypt the main shellcode, as shown in Figure 2.

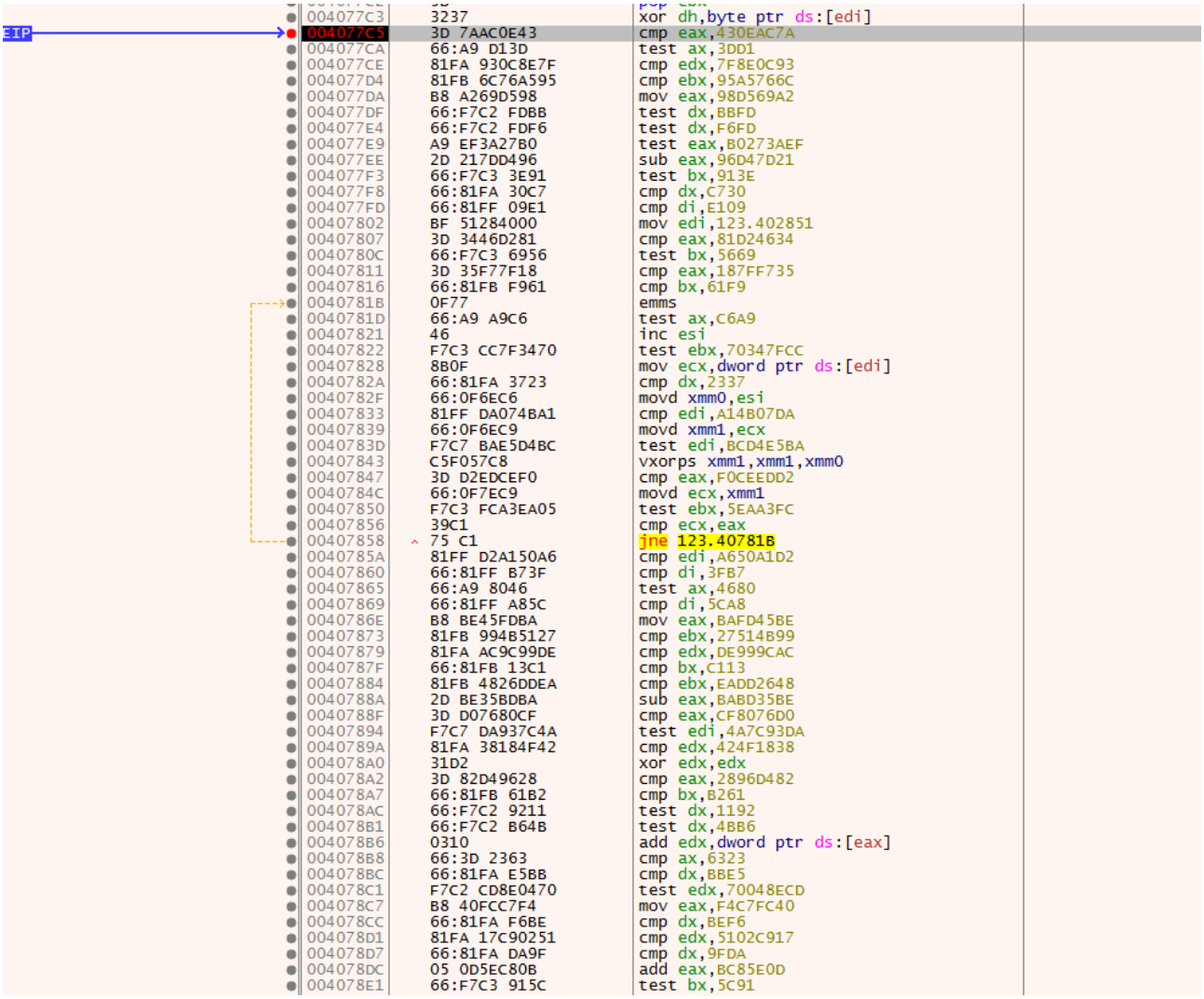


Figure 2: Block of code used to decrypt main shellcode (click image to enlarge)

The snippet above contains junk code inserted within legitimate instructions to thwart analysis. After analyzing and understanding this code further, we see that this code is responsible for decrypting the main shellcode in memory. It uses a 4-byte XOR key to decrypt the packed code to extract the final shellcode. The sample takes the first 4 bytes of encrypted data, XORs it with the ESI register and compares it with the value `0x200EC81`, as shown in Figure 3.

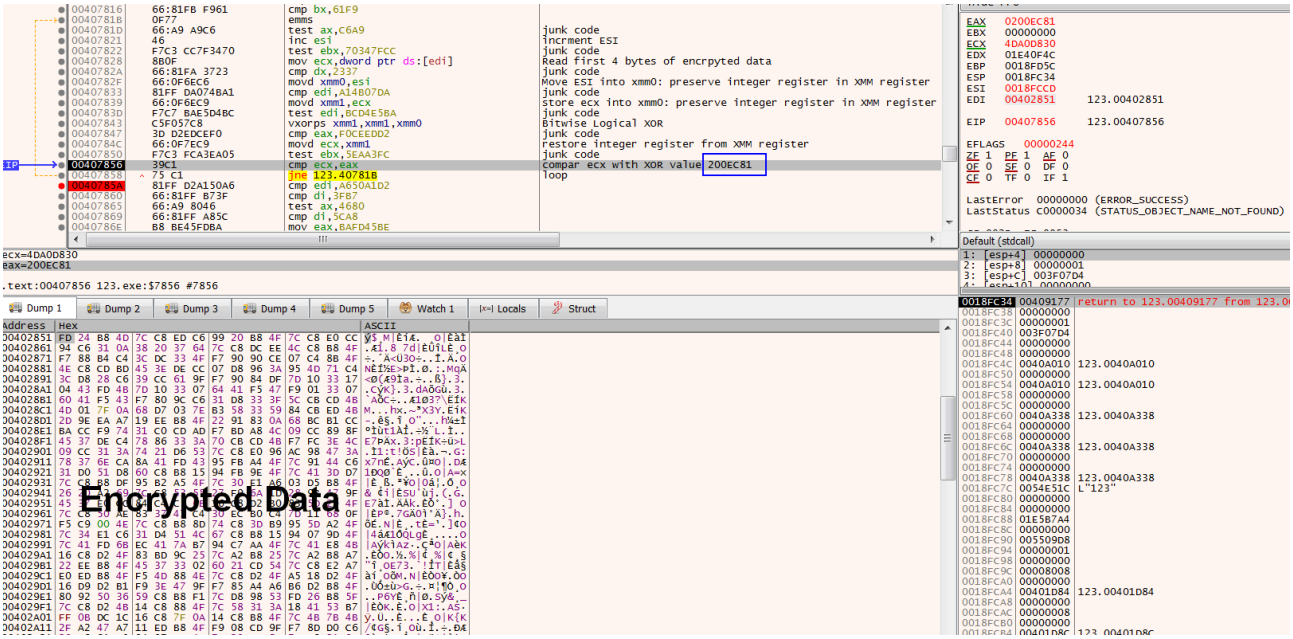


Figure 3: XOR key operation routine (click to enlarge image)

If it does not match, it keeps incrementing ESI and performs an XOR operation until the result matches the expected value. The value `0x200EC81` , read as little-endian, translates into the instruction `sub esp, 0x200` , which is the actual start of the final shellcode.

`(First 4 bytes of encrypted data in little endian`

`XOR 0x200EC81) = XOR Key` which for this sample becomes: `(0x4DB824FD XOR 0x200EC81) = 0x4FB8C87C` After this, the decryption routine will call `VirtualAlloc()` to allocate memory and start decrypting the final shellcode into the newly allocated memory by XORing encrypted data with key `0x4FB8C87C` , as shown in Figure 4.

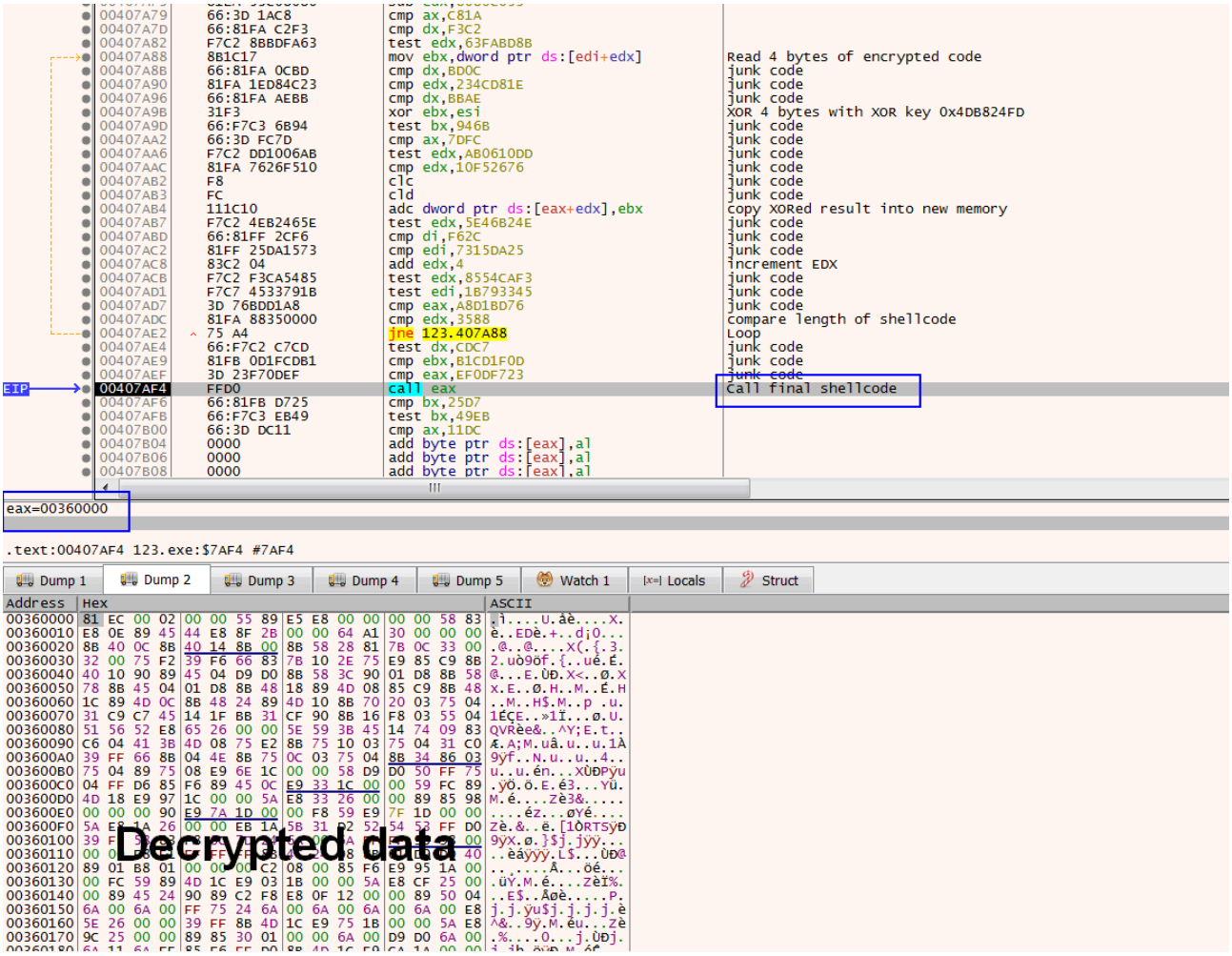


Figure 4: Decrypted data in memory (click image to enlarge)

Once the shellcode is decrypted, the code will jump into that new shellcode for further execution. Since the decryption routine has decrypted our shellcode, a memory dump of that newly allocated region gives us lots of interesting strings, including API names and the final encrypted payload hosted on Google Drive, as shown below.

ASCII Strings

```
00001A7F
```

```
hxxps<:>//drive.google.com/uc?export=download&id=1THD-itP7iOm05w_6SQSb-C3tgd3cLMzO 00001ADE
```

```
Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko 0001B28 wininet.dll 00001B3B
InternetOpenA 00001B4E InternetSetOptionA 00001B68 InternetOpenUrlA 00001B7E InternetReadFile 00001B94
InternetCloseHandle 00001BCB ntdll 00001BD6 NtCreateSection 00001BEB NtMapViewOfSection 00001C03
NtClose 00001C10 NtGetContextThread 00001C29 NtSetContextThread 00001C43 NtProtectVirtualMemory
00001C5F NtAllocateVirtualMemory 00001C7C NtWriteVirtualMemory 00001C98 NtOpenFile 00001CA9
NtResumeThread 00001CBD DbgBreakPoint 00001CD0 DbgUiRemoteBreakin 00001CE8 NtSetInformationThread
00001D05 kernel32 00001D13 WaitForSingleObject 00001D2D LoadLibraryA 00001D40 CreateProcessInternalW
00001D5C GetLongPathNameW 00001D73 TerminateProcess 00001D8A CreateThread 00001D9C
AddVectoredExceptionHandler 00001DBD TerminateThread 00001DD2 CreateFileW 00001DE5 WriteFile 00001DF5
```

`GetFileSize 00001E07 ReadFile 00001E15 CloseHandle 00001E26 Sleep 00001E31 advapi32 00001E3F RegCreateKeyExA 00001E54 RegSetValueExA 00001E68 user32 00001E74 EnumWindows 0000210F Startup key 00002120 Software\Microsoft\Windows\CurrentVersion\RunOnce 000021A0 shell32 000021AD SHCreateDirectoryExW 000021C8 ShellExecuteW`

Analyzing Shellcode

```
00360000    81EC 00020000       sub esp,200
00360006    55                  push ebp
00360007    89E5                mov ebp,esp
00360009    E8 00000000         call 36000E
0036000E    58                  pop eax
0036000F    83E8 0E             sub eax,E
00360012    8945 44             mov dword ptr ss:[ebp+44],eax
00360015    E8 8F2B0000         call 362BA9
0036001A    64:A1 30000000      mov eax,dword ptr fs:[30]
00360020    8B40 0C             mov eax,dword ptr ds:[eax+C]
00360023    8B40 14             mov eax,dword ptr ds:[eax+14]
00360026    8B00                mov eax,dword ptr ds:[eax]
00360028    8B58 28             mov ebx,dword ptr ds:[eax+28]
0036002B    817B 0C 33003200    cmp dword ptr ds:[ebx+C],320033
00360032  ^ 75 F2               jne 360026
00360034    39F6                cmp esi,esi
00360036    66:837B 10 2E       cmp word ptr ds:[ebx+10],2E
0036003B  ^ 75 E9               jne 360026
0036003D    85C9                test ecx,ecx
```

Figure 5: Entry point of the main shellcode (click image to enlarge)

This entire shellcode is heavily obfuscated, contains lots of junk code and also contains anti-analysis and anti-debugging tricks to make shellcode analysis more difficult. The shellcode starts with a few lines that prepare the stack and registers for use within the function before an interesting `call 362BA9` instruction, as shown in Figure 6.

```
00362BA9  $  39DB            cmp ebx,ebx                 Heavens_gate
00362BAB  .  64:8B1D C0000000 mov ebx,dword ptr fs:[C0]  Reserved for Wow64. Contains a pointer to FastSysCall in Wow64.
00362BB2  .  F8              clc
00362BB3  .  83FB 00         cmp ebx,0
00362BB6  .v 74 1F           je 362BD7                   Jump if not x64
00362BB8  .v EB 1E           jmp 362BD8
00362BBA     39D2            cmp edx,edx                 edx:sub_362BC2+15
00362BBC  $  58              pop eax
00362BBD     F8              clc
00362BBE     90              nop
00362BBF  .v EB 11           jmp 362BD2
00362BC1     FC              cld
00362BC2  r$ 5A              pop edx                     edx:sub_362BC2+15
00362BC3  .  FC              cld
00362BC4  .  66:BB 3300      mov bx,33                   32-bit is 0x23 and 64-bit is 0x33
00362BC8  .  66:53           push bx
00362BCA  .  50              push eax
00362BCB  .  89E0            mov eax,esp
00362BCD  .  83C4 06         add esp,6
00362BD0  .  FF28            jmp far fword ptr ds:[eax]  jump far pointer
00362BD2  >  E8 EBFFFFFF     call <sub_362BC2>
00362BD7  L> C3              ret
00362BD8  >  E8 DFFFFFFF     call 362BBC
00362BDD     50              push eax
```

Figure 6: Heaven's Gate technique (click to enlarge image)

The code in Figure 6 applies the Heaven's Gate technique, the technique for executing code from x86 to x64 with the far `JMP` command. The code checks the `FS:<0xC0>` register value to see whether the system is x64 or not. If it is x64, the shellcode uses the Heaven's Gate call technique.

Accessing Kernel Imports via PEB

When a malware injects a payload into memory, it needs to determine which API calls to use; this is done by using the Process Environment Block (PEB), which is always located at offset `0x30` within the Threat Information Block (TIB), which in turn is referenced by the segment register `FS:<0x00>`. For example, a common method is

to find the `kernel32.dll` address from the loaded modules and enumerate the export table of `kernel32.dll` to find `GetProcAddress()` and start loading the API addresses required for its operation. Figure 7 shows the code that does this after the Heaven's Gate function call.

```
00360015    E8 8F2B0000     call 362BA9                     Heaven's gate
0036001A    64:A1 30000000  mov eax,dword ptr fs:[30]       get a pointer to the PEB (Process Environment Block)
00360020    8B40 0C         mov eax,dword ptr ds:[eax+C]    get PEB_LDR_DATA structure
00360023    8B40 14         mov eax,dword ptr ds:[eax+14]   get InMemoryOrderModuleList
00360026    8B00            mov eax,dword ptr ds:[eax]
00360028    8B58 28         mov ebx,dword ptr ds:[eax+28]   get pointer to next modules name
0036002B    817B 0C 33003200 cmp dword ptr ds:[ebx+C],320033 compare for 32 string (kernel32.dll)
00360032  ^ 75 F2           jne 360026
00360034    39F6            cmp esi,esi
00360036    66:837B 10 2E   cmp word ptr ds:[ebx+10],2E     compare for . in string
0036003B  ^ 75 E9           jne 360026
0036003D    85C9            test ecx,ecx
0036003F    8B40 10         mov eax,dword ptr ds:[eax+10]   Base Address of Kernel32.dll
00360042    90              nop
00360043    8945 04         mov dword ptr ss:[ebp+4],eax    store the address
00360046    D9D0            fnop
00360048    8B58 3C         mov ebx,dword ptr ds:[eax+3C]   RVA of PE signature
0036004B    90              nop
0036004C    01D8            add eax,ebx                     address of PE signature: eax = eax (kernel32 base) + RVA of PE
0036004E    8B58 78         mov ebx,dword ptr ds:[eax+78]   RVA of Export Table
00360051    8B45 04         mov eax,dword ptr ss:[ebp+4]    base address of kernel
00360054    01D8            add eax,ebx
00360056    8B48 18         mov ecx,dword ptr ds:[eax+18]   Number of functions exported by a module
00360059    894D 08         mov dword ptr ss:[ebp+8],ecx    store the count
0036005C    85C9            test ecx,ecx
0036005E    8B48 1C         mov ecx,dword ptr ds:[eax+1C]   RVA of Address Table - addresses of exported functions
00360061    894D 0C         mov dword ptr ss:[ebp+C],ecx    stroe rva
00360064    8B48 24         mov ecx,dword ptr ds:[eax+24]   RVA of Ordinal Table - function order number as listed in the table
00360067    894D 10         mov dword ptr ss:[ebp+10],ecx   store rva
0036006A    8B70 20         mov esi,dword ptr ds:[eax+20]   RVA of Name Pointer Table - addresses of exported function names
0036006D    0375 04         add esi,dword ptr ss:[ebp+4]
00360070    31C9            xor ecx,ecx
00360072    C745 14 1FBB31CF mov dword ptr ss:[ebp+14],CF31BB1F Hash of GetProcAddress
00360079    90              nop
0036007A    8B16            mov edx,dword ptr ds:[esi]      move next API into EDX
0036007C    F8              clc
0036007D    0355 04         add edx,dword ptr ss:[ebp+4]    add base address
00360080    51              push ecx                        push counter
00360081    56              push esi
00360082    52              push edx                        push api name
00360083    E8 65260000     call <djb_hash_0x1505>          calculate djb hash value for api string
00360088    5E              pop esi
00360089    59              pop ecx
0036008A    3B45 14         cmp eax,dword ptr ss:[ebp+14]   compre EAX hash value with CF31BBF
0036008D  v 74 09           je 360098                       jump if GetProcAddress found
0036008F    83C6 04         add esi,4
00360092    41              inc ecx                         increment counter
00360093    3B4D 08         cmp ecx,dword ptr ss:[ebp+8]    compare API counter
00360096  ^ 75 E2           jne 36007A
00360098    8B75 10         mov esi,dword ptr ss:[ebp+10]
0036009B    0375 04         add esi,dword ptr ss:[ebp+4]
0036009E    31C0            xor eax,eax
```

Figure 7: Accessing kernel imports via PEB (click image to enlarge)

DJB2 Hashes for Windows API Resolution

When GuLoader needs to call a Windows API function, it must first resolve the function's address, as it does not have an Import Address Table (IAT). The code shown in Figure 7 iterates through export functions of `kernel32.dll` one by one, calculates the DJB2 hashes for each export API and compares those with the hardcoded hash value `CF31BB1F` (DJB2 hash of `GetProcAddress` API).

Python Snippet for DJB2 Hash Calculation

1. `val = 0x1505`
2. `inString = "GetProcAddress"`
3. `for ch in inString:`
4. `val += (val << 5)`
5. `val &= 0xFFFFFFFF`
6. `val += ord(ch)`
7. `val &= 0xFFFFFFFF`
8. `print(hex(val).upper().lstrip("0X").rstrip("L"))`

Once the shellcode matches the hash for the string name `GetProcAddress`, it will calculate its API address from `kernel32.dll`. Then it will start resolving the required APIs shown in the appendix at the end of this blog.

Anti-Sandbox/Anti-Emulation

GuLoader also checks the number of application windows to detect an analysis environment. This check uses the function `EnumWindows` to enumerate and count all top-level windows on the screen. If the number of windows is less than 12, the malware calls `TerminateProcess` with its own process handle as the parameter to terminate. This might have been done to evade sandboxes or emulator environments.

Anti-Attach: Patching DbgBreakPoint and DbgUIRemoteBreakin

The Windows API functions `DbgBreakPoint` and `DbgUiRemoteBreakin` are called when a debugger attaches to a running process. The shellcode patches these two APIs by replacing the `INT3` opcode of `DbgBreakPoint` with `opcode 90` (NOP, or "no-operation,"

to do nothing), and replacing the first few bytes of `DbgUIRemoteBreakin` with a dummy call (to cause a crash). This is done to prevent a debugger from attaching to the process, as shown in Figure 8.

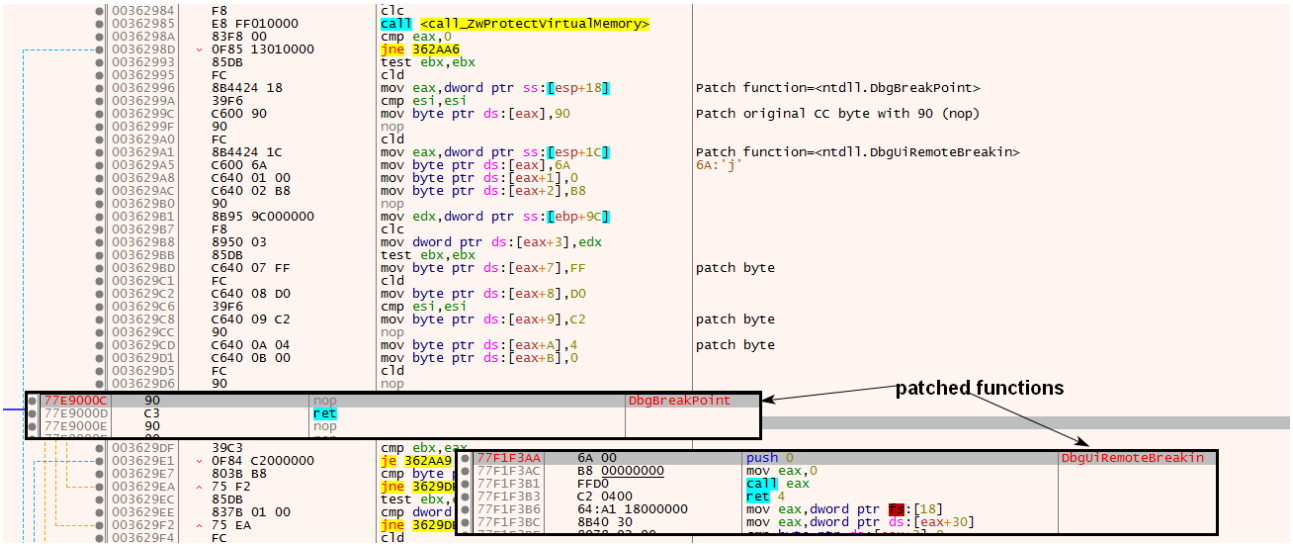


Figure 8: Patching `DbgBreakPoint` and `DbgUIRemoteBreakin` (click image to enlarge)

Unhooking API Hooks

The shellcode performs some pattern matching in the `NTDLL` API's code functions — for example, searching for the byte pattern "\xb8\x00.{3}\xb9," which represents `NTDLL` calls to system calls. Many security products like AV, endpoint detection and response (EDR) and sandbox software put their hooks here, so they can detour the execution flow into their engines to monitor and intercept API calls and block anything suspicious. Basic user-mode API hooks by AV/EDR are often created by modifying the first 5 bytes of the API call with a jump (`JMP`) instruction to another memory address pointing to the security software. Considering this hooking mechanism, the shellcode scans for all such system calls and then restores its first 5 bytes to the original bytes in `NTDLL` , as shown in Figure 9.

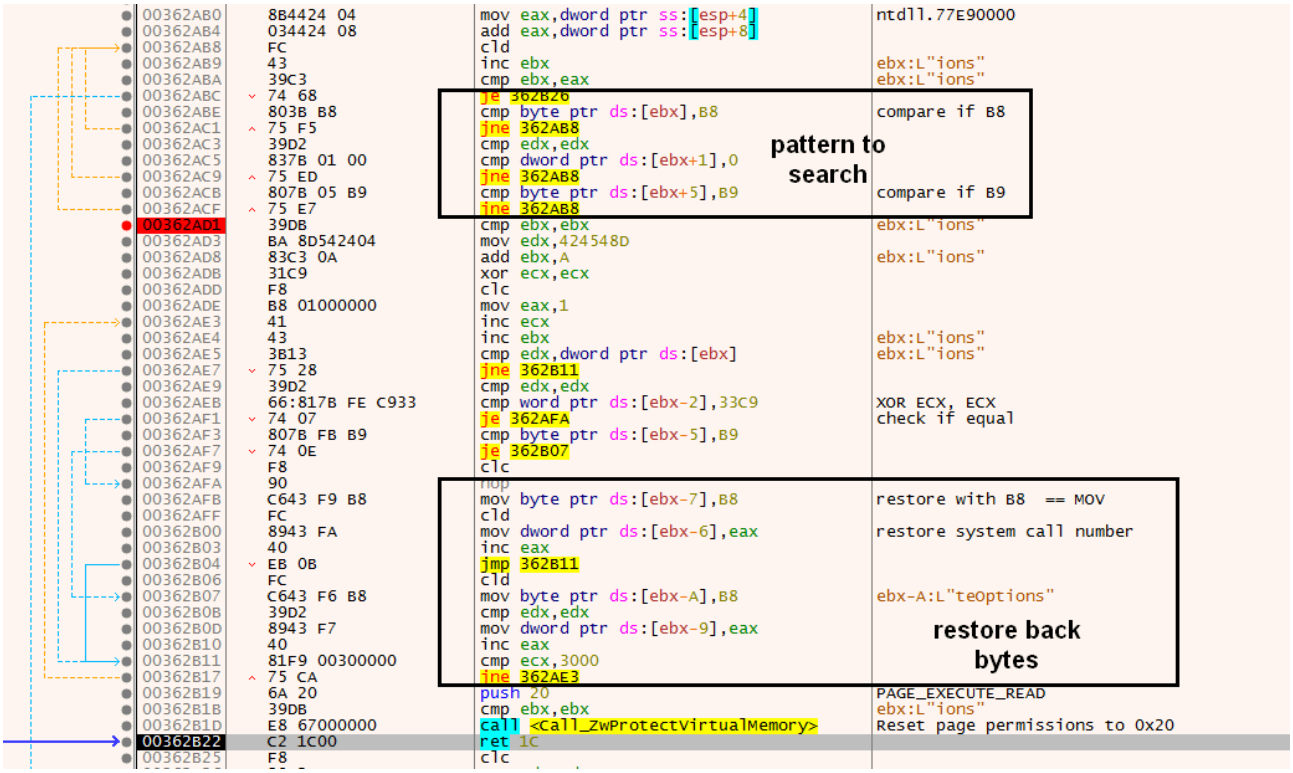


Figure 9: Unhooking API hooks code (click image to enlarge)

As a result, GuLoader bypasses any hooks installed by anti-malware software. Lastly, it resets the `NTDLL`'s memory permissions back to `PAGE_EXECUTE_READ` only.

Anti-debug (NtSetInformationThread)

Next, the shellcode calls the `NtSetInformationThread` function with `ThreadHideFromDebugger` (`0x11`) as the second parameter for hiding the thread from a debugger, as shown in Figure 10.

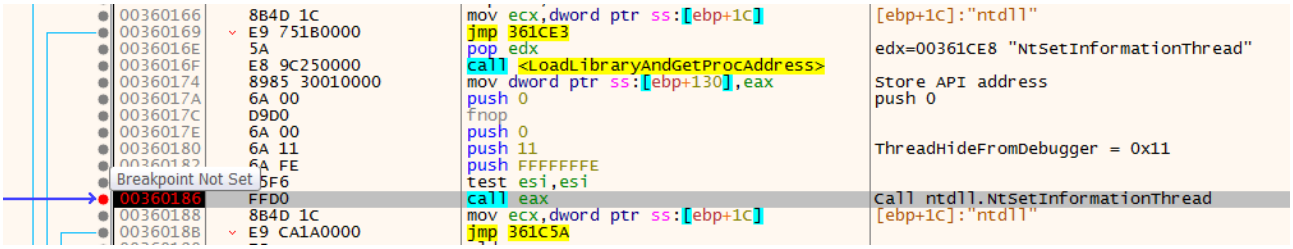


Figure 10: `NtSetInformation` thread function with `ThreadHideFromDebugger` parameter (click image to enlarge)

This causes a crash in the debugged application when a breakpoint is hit in the hidden thread or when the debugger steps through the instructions.

Anti-Analysis/Debug Techniques

The shellcode uses several anti-debugging techniques. The shellcode detects if hardware breakpoints or software breakpoints have been set, each time it calls several key API functions, as shown in Figure 11.

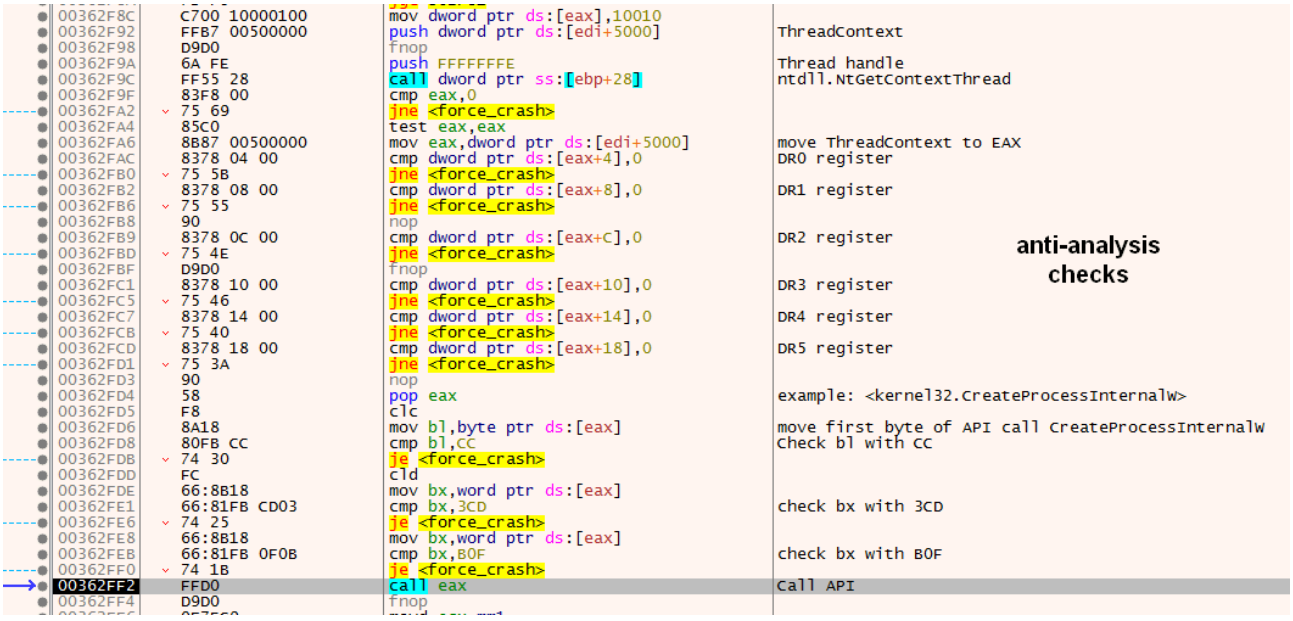


Figure 11: Software and hardware breakpoint checks (click image to enlarge)

During their malware analysis, analysts often use hardware or software breakpoints at the beginning of suspicious API calls — for example, by patching the first byte of `CreateProcessInternalW` with `0xCC`.

By calling the `NtGetContextThread` function, debug registers (`DR0` through `DR7`) can be used to detect hardware breakpoints, while `0xCC` , `0X3CD` and `0xB0F` opcodes are used to detect software breakpoints (if present) at the beginning of the API calls.

Process Hollowing Injection

Process hollowing is a code injection technique used by malware in which the executable code of a legitimate process in memory is replaced with malicious code. By executing within the context of legitimate processes, the malware can bypass security solutions. The shellcode similarly uses process hollowing techniques in order to inject its code into the legitimate process (here `RegAsm.exe` or `MSBuild.exe` or `RegSvcs.exe`) with a slight variation. Here, shellcode doesn't unmap memory code of legitimate processes; instead it uses the `NtCreateSection` API section object to inject its malicious code. The process is as follows: 1) Calls `kernel32.CreateProcessInternalW` to create the Windows legitimate process "C:\Windows\Microsoft.NET\Framework\v2.0.50727\RegAsm.exe" with `CREATE_SUSPENDED(0x00000004)` flags. If it doesn't find `RegAsm.exe` , it will try to find `MSBuild.exe` or `RegSvcs.exe` in the same directory path and loop until it finds one of them. 2) Opens a file handle to the hard-coded file path "C:\Windows\syswow64\mstsc.exe" using `ZwOpenFile`

3) Calls `ntdll.NtCreateSection` on the file handle for `mstsc.exe` . The `ZwCreateSection` function creates a section object that represents a section of memory that can be shared. This file handle is used to create a new section object with the DesiredAccess parameter. 4) The section is then mapped in the targeted process (`RegAsm.exe`) using the function `ntdll.NtMapViewOfSection` with the BaseAddress parameter set to `0x400000` . This maps the section in the base address `0x400000` , which is typically the address used to map the executable file image of the process.

5) Calls `ntdll.NtWriteVirtualMemory` in order to write the shellcode in the newly allocated memory of the targeted process. 6) Calls `ntdll.NtGetContextThread` to obtain information about the main thread within the suspended subprocess. 7) After the shellcode has been written to the memory of the targeted process, the execution needs to be redirected to it. To achieve this, GuLoader makes use of the function `ntdll.NtSetContextThread` to change the context of the only thread running in the targeted process (still in a suspended state). This context change sets the EIP register to the address that points to the beginning of the shellcode, which makes the execution start there. 8) Calls `ntdll.NtResumeThread` to resume the new thread in `RegAsm.exe` to execute the malicious shellcode.

Final Payload

After GuLoader has successfully injected into the `RegAsm.exe`

process, its shellcode will download the final payload from the Google Drive link in memory in an encrypted form, as shown in Figure 12.

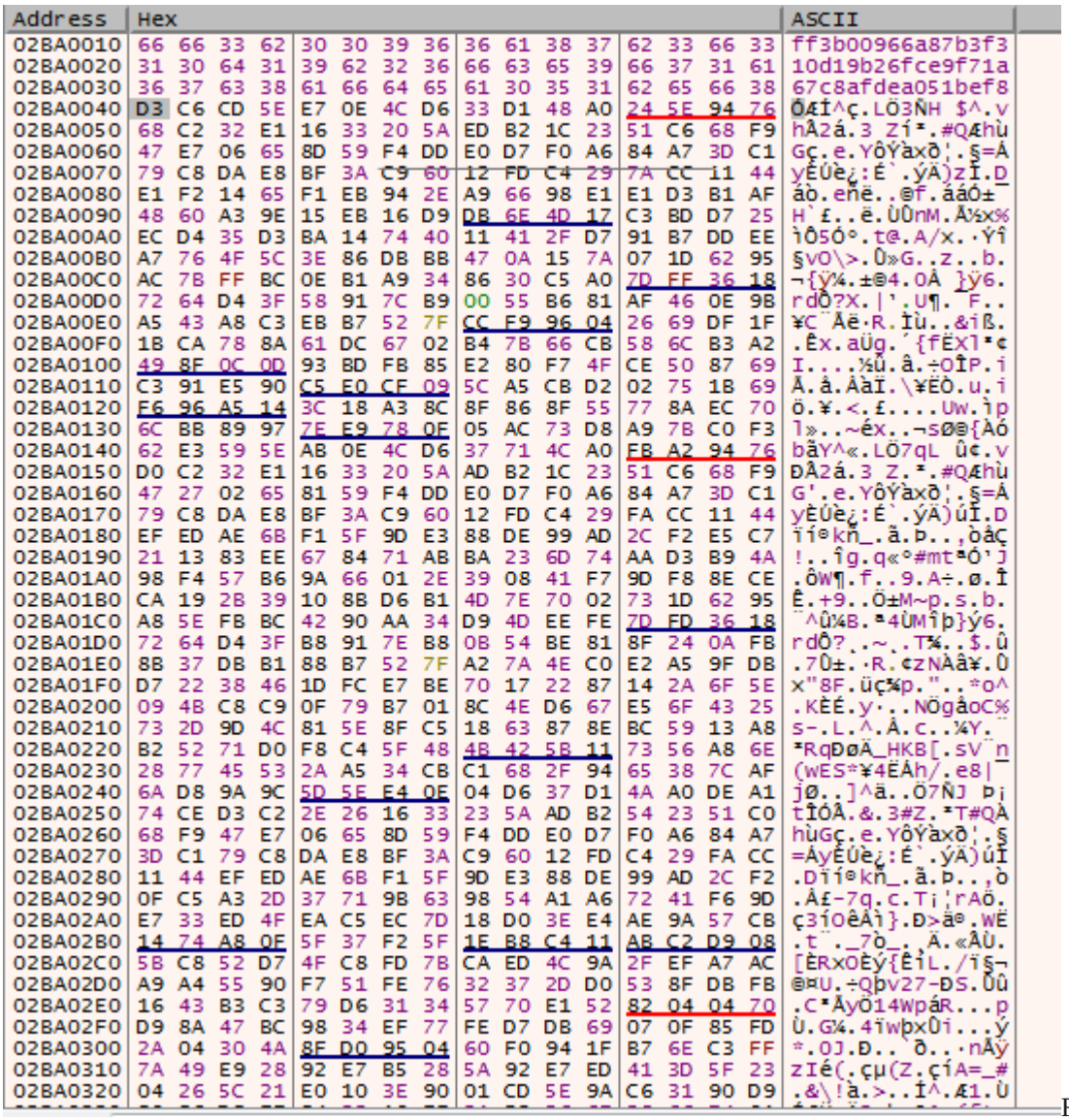
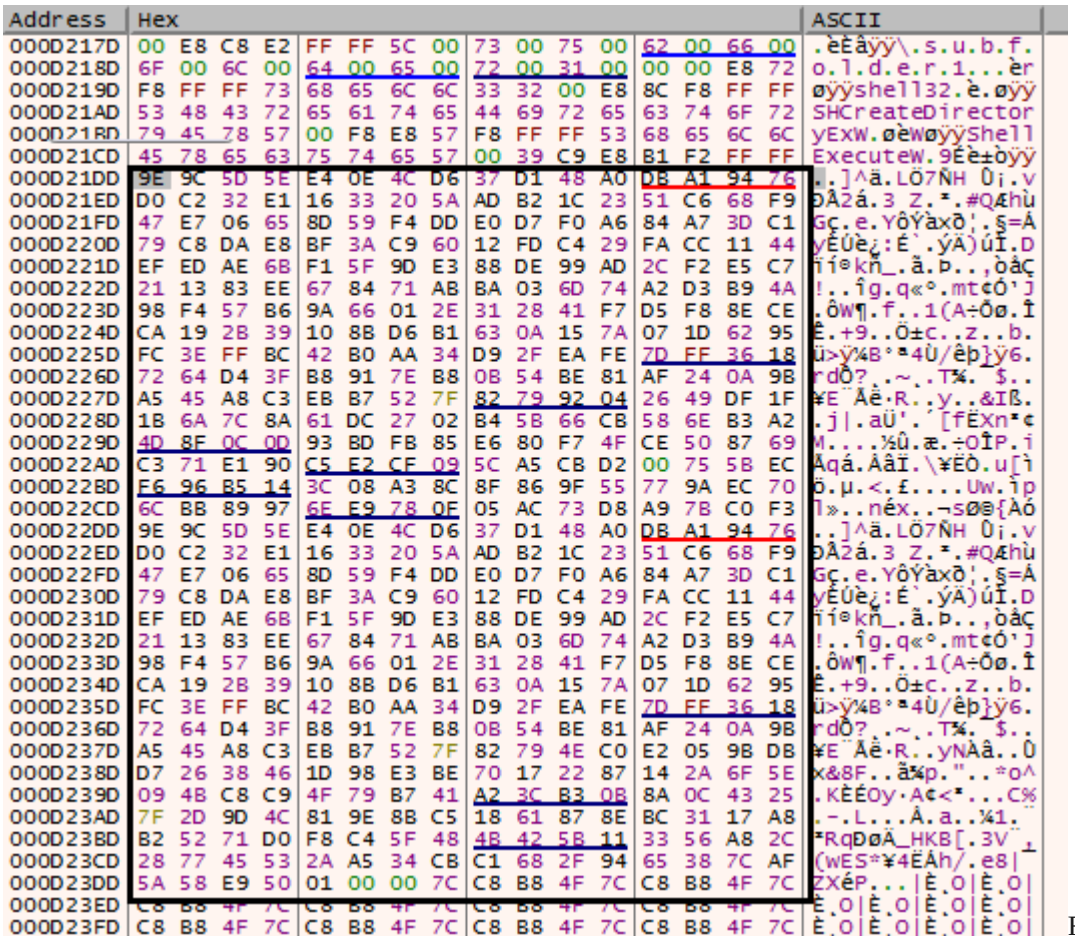


Figure 12: Encrypted final payload downloaded in the memory (click image to enlarge)

The real encrypted payload is appended after the first 64 bytes of random data. The GuLoader shellcode uses a hardcoded XOR key with a length of 517 bytes for this sample (as shown in Figure 13) to decrypt the final payload.



Address	Hex	ASCII
000D217D	00 E8 C8 E2 FF FF 5C 00 73 00 75 00 62 00 66 00	.èÈâÿÿ\.s.u.b.f.
000D218D	6F 00 6C 00 64 00 65 00 72 00 31 00 00 00 E8 72	o.l.d.e.r.1...èr
000D219D	F8 FF FF 73 68 65 6C 6C 33 32 00 E8 8C F8 FF FF	øÿÿshell32.è.øÿÿ
000D21AD	53 48 43 72 65 61 74 65 44 69 72 65 63 74 6F 72	SHCreateDirector
000D21BD	79 45 78 57 00 F8 E8 57 F8 FF FF 53 68 65 6C 6C	yExW.øèWøÿÿShell
000D21CD	45 78 65 63 75 74 65 57 00 39 C9 E8 B1 F2 FF FF	ExecuteW.9Éè±òÿÿ
000D21DD	9E 9C 5D 5E E4 0E 4C D6 37 D1 48 A0 DB A1 94 76	.]^ä.LÖ7ÑH Û¡.v
000D21ED	D0 C2 32 E1 16 33 20 5A AD B2 1C 23 51 C6 68 F9	ÐÂ2á.3 Z.².#QÆhù
000D21FD	47 E7 06 65 8D 59 F4 DD E0 D7 F0 A6 84 A7 3D C1	Gç.e.YôÝà×ð¦.§=Á
000D220D	79 C8 DA E8 BF 3A C9 60 12 FD C4 29 FA CC 11 44	yÈÚè¿:É`.ýÄ)úÌ.D
000D221D	EF ED AE 6B F1 5F 9D E3 88 DE 99 AD 2C F2 E5 C7	ïí®kñ_.ã.Þ..,òåÇ
000D222D	21 13 83 EE 67 84 71 AB BA 03 6D 74 A2 D3 B9 4A	!..îg.q«°.mt¢Ó'J
000D223D	98 F4 57 B6 9A 66 01 2E 31 28 41 F7 D5 F8 8E CE	.ôW¶.f..1(A÷Õø.Î
000D224D	CA 19 2B 39 10 8B D6 B1 63 0A 15 7A 07 1D 62 95	Ê.+9..Ö±c..z..b.
000D225D	FC 3E FF BC 42 B0 AA 34 D9 2F EA FE 7D FF 36 18	ü>ÿ¼B°ª4Ù/êþ}ÿ6.
000D226D	72 64 D4 3F B8 91 7E B8 0B 54 BE 81 AF 24 0A 9B	rdÔ?¸.~..T¾. $..
000D227D	A5 45 A8 C3 EB B7 52 7F 82 79 92 04 26 49 DF 1F	¥E¨ÃëR..y..&Iß.
000D228D	1B 6A 7C 8A 61 DC 27 02 B4 5B 66 CB 58 6E B3 A2	.j|.aÜ'.´[fËXn³¢
000D229D	4D 8F 0C 0D 93 BD FB 85 E6 80 F7 4F CE 50 87 69	M...½û.æ.÷OÎP.i
000D22AD	C3 71 E1 90 C5 E2 CF 09 5C A5 CB D2 00 75 5B EC	Ãqá.ÅâÏ.\¥ËÒ.u[ì
000D22BD	F6 96 B5 14 3C 08 A3 8C 8F 86 9F 55 77 9A EC 70	ö.µ.<.£....Uwìp
000D22CD	6C BB 89 97 6E E9 78 0F 05 AC 73 D8 A9 7B C0 F3	l»..néx..¬sØ©{Àó
000D22DD	9E 9C 5D 5E E4 0E 4C D6 37 D1 48 A0 DB A1 94 76	..]^ä.LÖ7ÑH Û¡.v
000D22ED	D0 C2 32 E1 16 33 20 5A AD B2 1C 23 51 C6 68 F9	ÐÂ2á.3 Z.².#QÆhù
000D22FD	47 E7 06 65 8D 59 F4 DD E0 D7 F0 A6 84 A7 3D C1	Gç.e.YôÝà×ð¦.§=Á
000D230D	79 C8 DA E8 BF 3A C9 60 12 FD C4 29 FA CC 11 44	yÈÚè¿:É`.ýÄ)úÌ.D
000D231D	EF ED AE 6B F1 5F 9D E3 88 DE 99 AD 2C F2 E5 C7	ïí®kñ_.ã.Þ..,òåÇ
000D232D	21 13 83 EE 67 84 71 AB BA 03 6D 74 A2 D3 B9 4A	!..îg.q«°.mt¢Ó'J
000D233D	98 F4 57 B6 9A 66 01 2E 31 28 41 F7 D5 F8 8E CE	.ôW¶.f..1(A÷Õø.Î
000D234D	CA 19 2B 39 10 8B D6 B1 63 0A 15 7A 07 1D 62 95	Ê.+9..Ö±c..z..b.
000D235D	FC 3E FF BC 42 B0 AA 34 D9 2F EA FE 7D FF 36 18	ü>ÿ¼B°ª4Ù/êþ}ÿ6.
000D236D	72 64 D4 3F B8 91 7E B8 0B 54 BE 81 AF 24 0A 9B	rdÔ?¸.~..T¾. $..
000D237D	A5 45 A8 C3 EB B7 52 7F 82 79 4E C0 E2 05 9B DB	¥E¨ÃëR..yNÀâ..Û
000D238D	D7 26 38 46 1D 98 E3 BE 70 17 22 87 14 2A 6F 5E	×&8F..ã¾p."..*o^
000D239D	09 4B C8 C9 4F 79 B7 41 A2 3C B3 0B 8A 0C 43 25	.KÈÉOy·A¢<³..C%
000D23AD	7F 2D 9D 4C 81 9E 8B C5 18 61 87 8E BC 31 17 A8	.-.L...Å.a..¼1.¨
000D23BD	B2 52 71 D0 F8 C4 5F 48 4B 42 5B 11 33 56 A8 2C	²RqÐøÄ_HKB[.3V¨,
000D23CD	28 77 45 53 2A A5 34 CB C1 68 2F 94 65 38 7C AF	(wES*¥4ËÁh/.e8|¯
000D23DD	5A 58 E9 50 01 00 00 7C C8 B8 4F 7C C8 B8 4F 7C	ZXéP...|È¸O|È¸O|
000D23ED	C8 B8 4F 7C C8 B8 4F 7C C8 B8 4F 7C C8 B8 4F 7C	È¸O|È¸O|È¸O|È¸O|
000D23FD	C8 B8 4F 7C C8 B8 4F 7C C8 B8 4F 7C C8 B8 4F 7C	È¸O|È¸O|È¸O|È¸O|

Figure 13: Embedded XOR key (null terminated) for decrypting final payload (click image to enlarge)

The following piece of code from the shellcode decrypts its encrypted payload back into its original one, as shown in Figure 14.

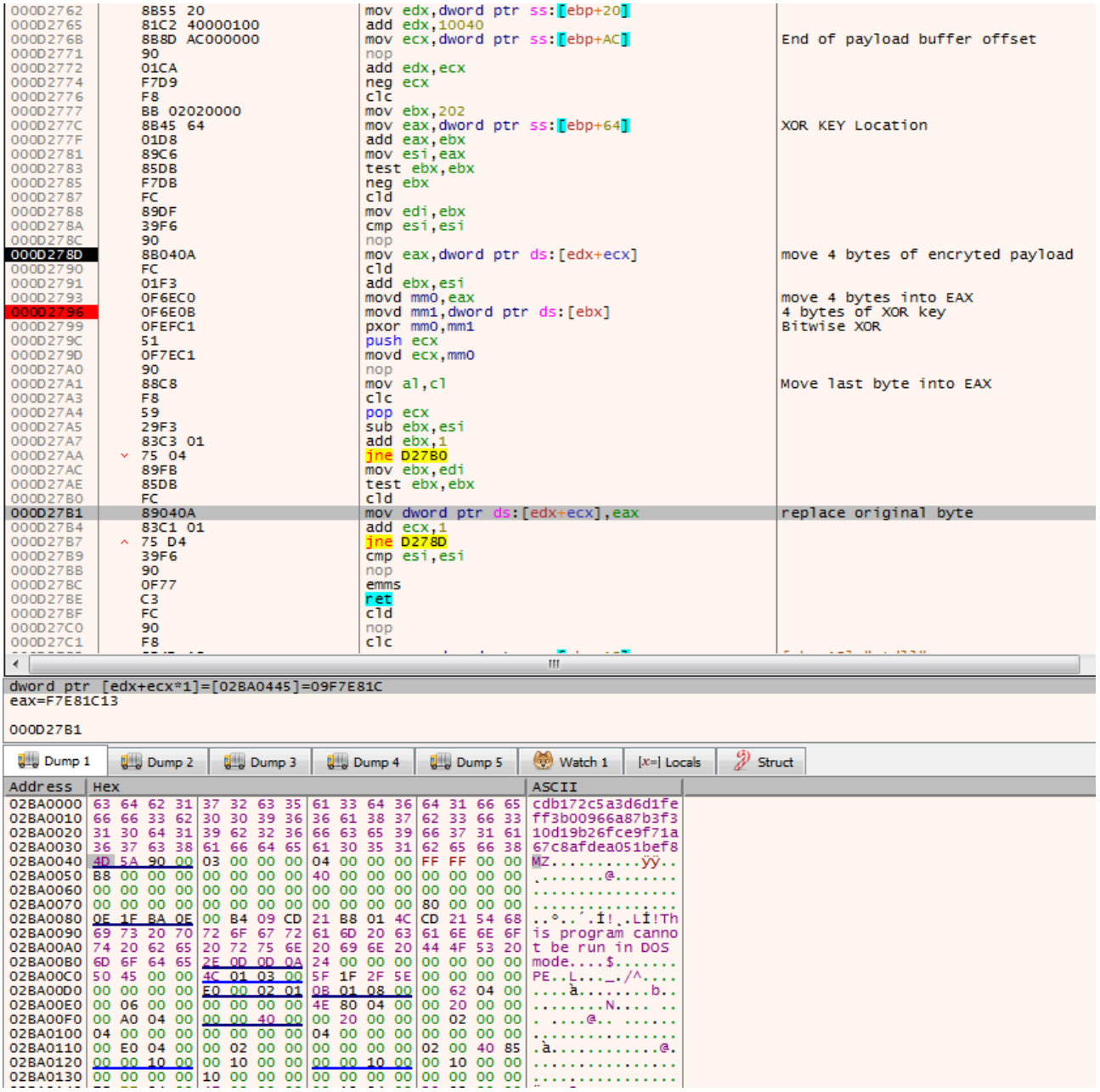


Figure 14: Decryption routine and decrypted final payload (click image to enlarge)

How the CrowdStrike Falcon® Platform Protects Against GuLoader

The CrowdStrike Falcon®® platform has the ability to detect and prevent GuLoader by taking advantage of the behavioral patterns indicated by the malware. By turning on suspicious process blocking, Falcon ensures that GuLoader is killed in the very early stages of execution.

In addition, the CrowdStrike® machine learning (ML) algorithm provides additional coverage against this malware family, as illustrated in Figure 16.

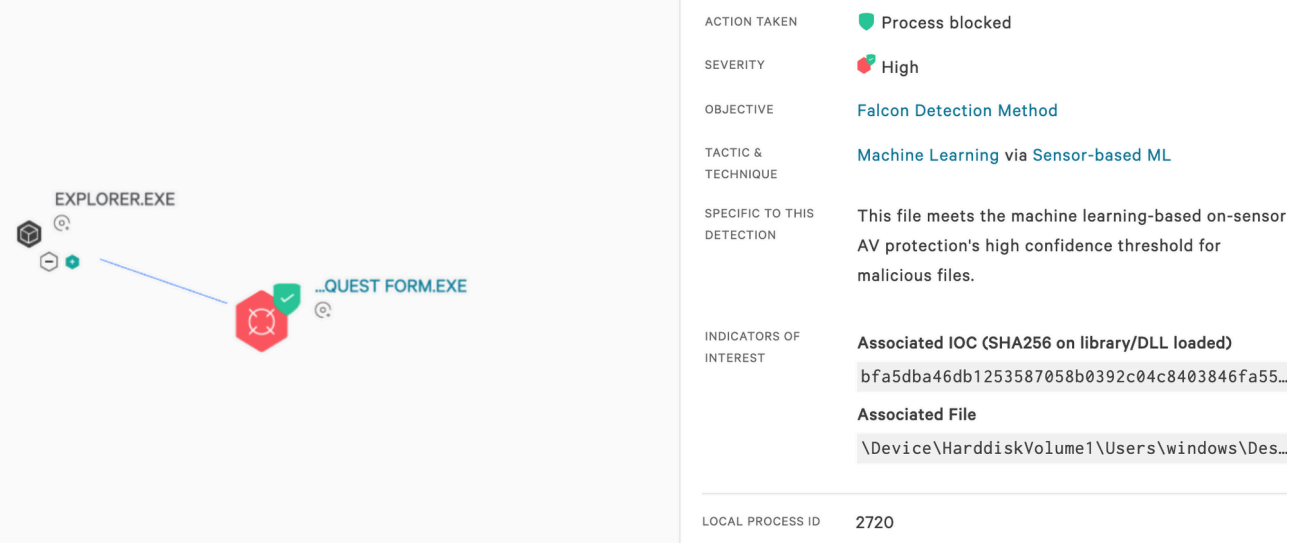


Figure 16: GuLoader process blocked by ML algorithm (click image to enlarge)

Conclusion

GuLoader has been very active in 2020 and is frequently used by criminals to distribute their malware like AgentTesla, FormBook and NanoCore. The use of process hollowing and hosting encrypted payloads on Google Drive is designed to bypass many security solutions — but it doesn't bypass CrowdStrike Falcon®.

Appendix: APIs Resolved by GuLoader

- LoadLibraryA
- TerminateProcess
- EnumWindows
- ZwProtectVirtualMemory
- DbgBreakPoint
- DbgUIRemoteBreakin
- NtGetContextThread
- NtSetContextThread
- NtWriteVirtualMemory
- NtCreateSection
- NtMapViewOfSection
- NtOpenFile
- NtClose
- NtResumeThread
- CreateProcessInternalW
- GetLongPathNameW
- Sleep
- CreateThread
- WaitForSingleObject
- TerminateThread
- AddVectoredExceptionHandler

- CreateFileW
- WriteFile
- CloseHandle
- GetFileSize
- ReadFile
- ShellExecuteW
- SHCreateDirectoryExW
- RegCreateKeyExA
- RegSetValueExA

Indicators of Compromise (IOCs)

Additional Resources

- *Learn more about the CrowdStrike Falcon® platform by visiting the product webpage.*
- *Learn more about CrowdStrike endpoint detection and response by visiting the Falcon Insight™ webpage.*
- *Test CrowdStrike next-gen AV for yourself. Start your free trial of Falcon Prevent™ today.*
