Defeating Guloader Anti-Analysis Technique

By Mark Lim

Published: 2022-10-28 · Archived: 2026-04-05 21:17:45 UTC

Executive Summary

Unit 42 researchers recently discovered a Guloader variant that contains a shellcode payload protected by anti-analysis techniques, which are meant to slow human analysts and sandboxes processing this sample. To help speed analysis for this sample and others like it, we are providing a complete Python script to deobfuscate the Guloader sample that is available on GitHub.

In early September 2022, we discovered a Guloader variant with low VirusTotal detection. Guloader (also known as CloudEye) is a malware downloader first discovered in December 2019.

We analyzed the control flow obfuscation technique used by this Guloader sample to create the IDA Processor module extension script so researchers can deobfuscate the sample automatically. The script can be applied to other malware families like Dridex, which utilize similar anti-analysis techniques.

Palo Alto Networks customers receive protections from malware families using similar anti-analysis techniques with Cortex XDR or the Next-Generation Firewall with cloud-delivered security services, including WildFire and Advanced Threat Prevention.

Guloader Control Flow Obfuscation Technique

The Guloader sample in question uses the control flow obfuscation technique to hide its functionalities and evade detection. This technique impedes both static and dynamic analysis.

First, let’s look at how this threat hampers static analysis. In short, it uses CPU instructions that trigger exceptions, resulting in unintelligible code during static analysis.

After peeling away the packer layer of our Guloader sample, we see that its code is obfuscated. Using static analysis tools such as IDA Pro, we observe many 0xCC bytes (or int3 instructions) littered throughout the sample, as shown in Figure 1.

Following the 0xCC bytes are junk instructions. These added bytes disrupt the static analysis tool’s disassembly process, resulting in the wrong disassembly listing.

```
.text:00401451 E8 EF C4 00+call    sub_40D945
.text:00401451 00
.text:00401456 89 5D 18     mov     [ebp+18h], ebx
.text:00401459 8B 4D 18     mov     ecx, [ebp+18h]
.text:0040145C CC           int     3                               ; Trap to Debugger
.text:0040145D A5           movsd
.text:0040145E 00 8E 7B 8E+add     [esi+6F348E7Bh], cl
.text:0040145E 34 6F
.text:00401464 CD 4C        int     4Ch                             ; Z100 - Slave 8259 - S100 v
.text:00401466 22 B3 BA 57+and     dh, [ebx-7465A846h]
.text:00401466 9A 8B
.text:0040146C A5           movsd
.text:0040146D E8 18 BA 00+call    sub_40CE8A
.text:0040146D 00
.text:00401472 89 85 98 00+mov     [ebp+98h], eax
.text:00401472 00 00
.text:00401478 CC           int     3                               ; Trap to Debugger
.text:00401479 A7           cmpsd
.text:0040147A 7E 00        jle     short $+2
.text:0040147C
.text:0040147C              loc_40147C:                             ; CODE XREF: sub_4013FC+7E↑j
.text:0040147C 72 8D        jb      short loc_40140B
.text:0040147E F6 C8 87     test    al, 87h
.text:00401481 1E           push    ds
.text:00401482 F4           hlt
```

Figure 1. Obfuscated code blocks.

0xCC bytes are CPU instructions that trigger an exception *EXCEPTION_BREAKPOINT* (0x80000003), which pauses the execution of a process. The CPU will pass the code flow to the handler function before the execution continues. The handler function is responsible for moving the instruction pointer to the correct address.

The presence of these same 0xCC bytes make it so that using a debugger during dynamic analysis would crash the Guloader sample. Debuggers insert 0xCC bytes as software breakpoints to halt the execution of the sample. The debugger handles the exception instead of the handler function.

Before understanding what happens in the handler function, we first have to locate its address.

Guloader uses the AddVectoredExceptionHandler function to register the handler function, as shown in Figure 2. The second argument of the AddVectoredExceptionHandler function points to the address of the handler function.

```
PVOID AddVectoredExceptionHandler(
  ULONG                       First,
  PVECTORED_EXCEPTION_HANDLER Handler
);
```

Figure 2. Function prototype of AddVectoredExceptionHandler.

Using a debugger as shown in Figure 3, we locate the address of the handler function registered by the Guloader sample. With the address information, we can examine its code. Notably, this ExceptionHandler is registered with the order of 1, meaning it is the first handler to be invoked.

```
.text:0041069A call    sub_4106E3                       0018FC78  00000001
.text:0041069F call AddVectoredExceptionHandler         0018FC7C  00410944  sub_40136B+F5D9
.text:0041069F                                          0018FC80  0018FF84  Stack[00000410]:0018FF84
```

Figure 3. Debugging the call to AddVectoredExceptionHandler in Guloader sample.

Analyzing the Vectored Exception Handler Function

The first step of analyzing the handler function is to apply its type information, as shown in Figure 4.

```
LONG PvectoredExceptionHandler(
  [in] _EXCEPTION_POINTERS *ExceptionInfo
)
{ ... }
```

Figure 4. Type information for the handler function.

Next, we apply the type information for three Windows data structures (shown in Figure 5) used by the handler function.

```
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT          ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;


typedef struct _EXCEPTION_RECORD {
  DWORD                    ExceptionCode;
  DWORD                    ExceptionFlags;
  struct _EXCEPTION_RECORD *ExceptionRecord;
  PVOID                    ExceptionAddress;
  DWORD                    NumberParameters;
  ULONG_PTR                ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;


typedef struct _CONTEXT
{
     ULONG ContextFlags;
     ULONG Dr0;
     ULONG Dr1;
     ULONG Dr2;
     ULONG Dr3;
     ULONG Dr6;
     ULONG Dr7;
    ...
} CONTEXT, *PCONTEXT;
```

Figure 5. Type information of three Windows data structures to be applied on the handler function.

With the type information applied, we can examine how the function handled the exceptions caused by the 0xCC bytes. Figure 6 shows the decompiled handler function (Func_VectoredExceptionHandler) annotated with comments.

```
1 void __cdecl Func_VectoredExceptionHandler(_EXCEPTION_POINTERS *a1)
2 {
3   PCONTEXT Context; // eax
4   _BYTE *Eip; // edx
5   _BYTE *Offset; // edx
6   _BYTE *i; // ecx
7
8   if ( a1->ExceptionRecord->ExceptionCode == STATUS_BREAKPOINT )// check type of exception raised
9   {
10    Context = a1->ContextRecord;
11    if ( !Context->Dr0 && !Context->Dr1 && !Context->Dr2 && !Context->Dr3 && !Context->Dr6 && !Context->Dr7 )// Check for Hardware Breakpoint
12    {
13      Eip = (_BYTE *)Context->Eip;
14      if ( *Eip == 0xCC )                   // CC byte raising exception
15      {
16        Offset = (_BYTE *)(Eip[1] ^ 0xA9);   // Decode Offset byte
17        for ( i = &Offset[Context->Eip - 1]; (_BYTE *)(Context->Eip + 2) != i; --i )// loop to check for Software Breakpoint
18        {
19          if ( *i == 0xCC )                 // Exit Handler function if Software Breakpoint is found
20            return;
21        }
22        Context->Eip += (unsigned int)Offset;  // Update EIP with offset
23      }
24    }
25  }
26 }
```

Figure 6. Decompiled handler function.

The handler function begins with anti-debugging checks. It will terminate execution when hardware or software breakpoints are found. Next, the offset value is computed by XOR decoding the byte after the 0xCC byte with 0xA9. Finally, the offset value is added to the instruction pointer before the code execution resumes. Code execution continues at the address pointed to by the updated instruction pointer.

After understanding how the obfuscation is carried out, we can identify the legitimate instructions and discard the unwanted ones, as shown in Figure 7.

```
.text:00401D7D 89 85 B3 01+mov     [ebp+1B3h], eax              ; legit instruction
.text:00401D7D 00 00
.text:00401D83 CC          int     3                            ; trigger exception
.text:00401D83             ; ---------------------------------------------------------------
.text:00401D84 AF          db 0AFh                              ; encoded offset value
.text:00401D85             ; ---------------------------------------------------------------
.text:00401D85 03 E3       add     esp, ebx                     ; junk instruction
.text:00401D87 19 3E       sbb     [esi], edi                   ; junk instruction
.text:00401D89 89 D0       mov     eax, edx                     ; legit instruction
.text:00401D8B CC          int     3                            ; trigger exception
.text:00401D8B             ; ---------------------------------------------------------------
.text:00401D8C A2          db 0A2h                              ; encoded offset value
.text:00401D8D             ; ---------------------------------------------------------------
.text:00401D8D CE          into                                 ; junk instruction
.text:00401D8E 5E          pop     esi                          ; junk instruction
.text:00401D8F FD          std                                  ; junk instruction
.text:00401D90 70 A8       jo      short near ptr dword_401C84+0B6h ; junk instruction
.text:00401D92 D5 70       aad     70h ; 'p'                    ; junk instruction
.text:00401D92             ; ---------------------------------------------------------------
.text:00401D94 09          db    9                              ; junk instruction
.text:00401D95 4C          db   4Ch ; L                         ; junk instruction
.text:00401D96             ; ---------------------------------------------------------------
.text:00401D96 50          push    eax                          ; legit instruction
.text:00401D97 8B 85 B3 01+mov     eax, [ebp+1B3h]              ; legit instruction
.text:00401D97 00 00
.text:00401D9D CC          int     3                            ; Trap to Debugger
```

Figure 7. Labeled code block.

To completely deobfuscate the Guloader sample, we need to replace all the 0xCC bytes with a JMP short instruction (0xEB) and the following byte with the decoded offset value.

Because doing all this manually is time consuming, in the next section we will show you how to write an [IDA Processor module extension] to automate the deobfuscation process.

Writing an IDA Processor Module Extension

IDA Processor module extensions allow us to influence the disassembler logic in IDA Pro. These extensions are written using Python to enable us to filter and manipulate how IDA Pro disassembles the instructions in the sample.

The Python script extends the [ev_ana_insn] method in the [IDP_Hooks] class. It starts by checking if the current instruction is the 0xCC byte. Next, the 0xCC byte is replaced with the JMP short instruction (0xEB). Finally, the following byte is replaced with the decoded offset value.

Figure 8 shows the function in the Python script where this deobfuscation is implemented.

```
class guloader_veh_hook(idaapi.IDP_Hooks):

    def __init__(self):
        idaapi.IDP_Hooks.__init__(self)

    def ev_ana_insn(self, insn):

        b = bytes(idaapi.get_bytes(insn.ea, 2))      #read 2 bytes
        f = idaapi.get_flags(insn.ea)                #read flags of current instruction
        if idaapi.is_tail(f):                        #check if current byte is inside existing instruction/data to prevent false positive
            return False

        if b[0] ==0xCC:                              #look for CC byte that trigger VEH
            offset = b[1] ^ 0xA9                     #decode offset value
            idaapi.put_byte(insn.ea, 0xEB)           #patch CC byte with JMP instruction
            idaapi.put_byte(insn.ea+1, offset-2)     #patch encoded offset with decoded offset
            print("Patched bytes at: 0x%X" % insn.ea)
            return True

        return True
```

Figure 8. Extending the ev_ana_insn() to deobfuscate the sample.

After applying the Python script, IDA Pro can deobfuscate the Guloader sample automatically, as shown in Figure 9.

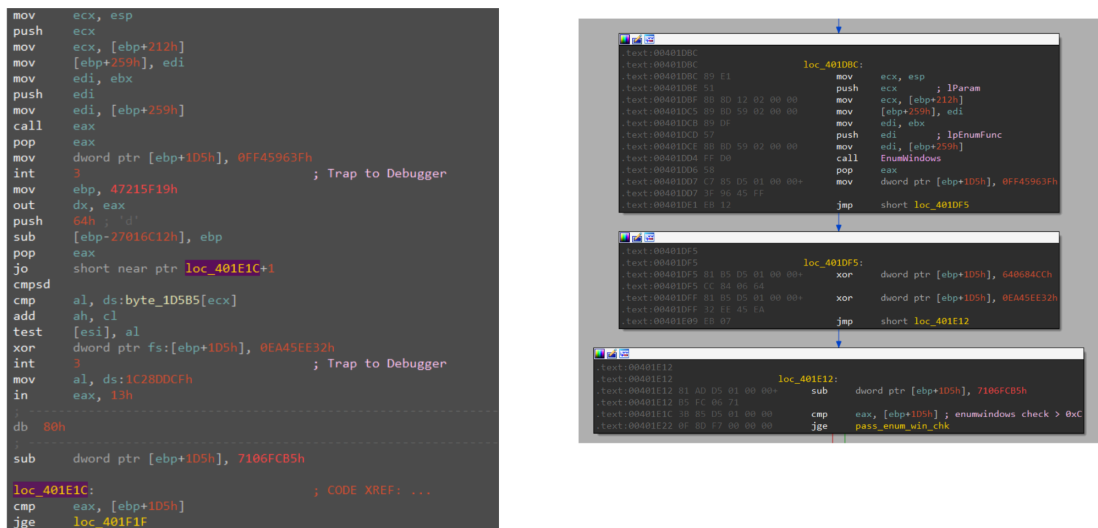

Figure 9: Obfuscated code (left) and code block after deobfuscation (right).

Conclusion: Malware Analysts vs. Malware Authors

Malware authors often include obfuscation techniques, hoping that they will increase the time and resources required for malware analysts to process their creations. Using the steps above, you can reduce the time needed to analyze these malware samples from Guloader, as well as those of other families using similar techniques.

Palo Alto Networks customers receive protections from malware families using similar anti-analysis techniques with Cortex XDR or the Next-Generation Firewall with cloud-delivered security services, including WildFire and Advanced Threat Prevention.

Indicators of Compromise

SQ21002728.IMG:
SHA256: fb8e52ec2e9d21a30d7b4dee8721d890a4fbec48103a021e9c04dfb897b71060
SQ21002764

SQ21002728.vbs:
SHA256: 56cdfaa44070c2ad164bd1e7f26744a2ffe54487c2d53d3ae318d842c6f56178
SQ21002764

Source: https://unit42.paloaltonetworks.com/guloader-variant-anti-analysis/