Getting gooey with GULOADER: deobfuscating the downloader

Subscribe

Overview

Elastic Security Labs continues to monitor active threats such as GULOADER, also known as CloudEyE – an evasive shellcode downloader that has been highly active for years while under constant development. One of these recent changes is the addition of exceptions to its Vectored Exception Handler (VEH) in a fresh campaign, adding more complexity to its already long list of anti-analysis tricks.

While GULOADER's core functionality hasn't changed drastically over the past few years, these constant updates in their obfuscation techniques make analyzing GULOADER a time-consuming and resource-intensive process. In this post, we will touch on the following topics when triaging GULOADER:

- Reviewing the initial shellcode and unpacking process
- Finding the entrypoint of the decrypted shellcode
- Discuss update to GULOADER's VEH that obfuscates control flow
- Provide a methodology to patch out VEH

Initial Shellcode

In our sample, GULOADER comes pre-packaged inside an NSIS (Nullsoft Scriptable Install System) installer. When the installer is extracted, the main components are:

NSIS Script - This script file outlines all the various configuration and installation aspects.

Name	Date modified	Type	Size
$PLUGINSDIR	11/20/2023 10:35 ...	File folder	
Bonanzas	11/20/2023 10:35 ...	File folder	
Gudesakes	11/20/2023 10:35 ...	File folder	
Landbrugsstyring	11/20/2023 10:35 ...	File folder	
[NSIS].nsi	10/17/2023 10:06 ...	NSI File	12 KB

Extracted NSIS contents

System.dll - Located under the `$PLUGINSDir`. This file is dropped in a temporary folder to allocate/execute the GULOADER shellcode.

Address	Type	Ordinal	Symbol
10001000	Export	1	Alloc
100016BD	Export	2	Call
10001058	Export	3	Copy
100015B3	Export	4	Free
1000161A	Export	5	Get
1000180D	Export	6	Int64Op
100010E0	Export	7	Store
1000103D	Export	8	StrAlloc

System.Dll exports

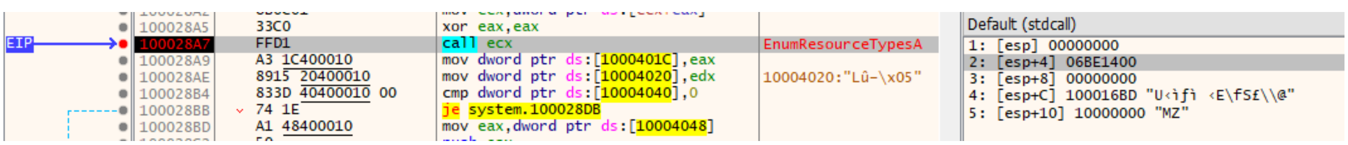
Shellcode - The encrypted shellcode is buried into a nested folder.

One quick methodology to pinpoint the file hosting the shellcode can be done by monitoring `ReadFile` events from SysInternal's Process Monitor after executing GULOADER. In this case, we can see that the shellcode is read in from a file (`Fibroms.Hag`).

proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	68...	ReadFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Gudesakes\Orkestrere\Tilbagetrkningen\Galavants\Fibroms.Hag
proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	68...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability
proctoring.exe	65...	CreateFile	C:\Users\REM\AppData\Local\Temp\slagborenes\Saxofonists85\Forfdning\Bonanzas\Cutability

Shellcode Retrieved from File

GULOADER executes shellcode through callbacks using different Windows API functions. The main reasoning behind this is to avoid detections centered around traditional Windows APIs used for process injection, such as `CreateRemoteThread` or `WriteProcessMemory`. We have observed `EnumResourceTypesA` and `CallWindowProcW` used by GULOADER.

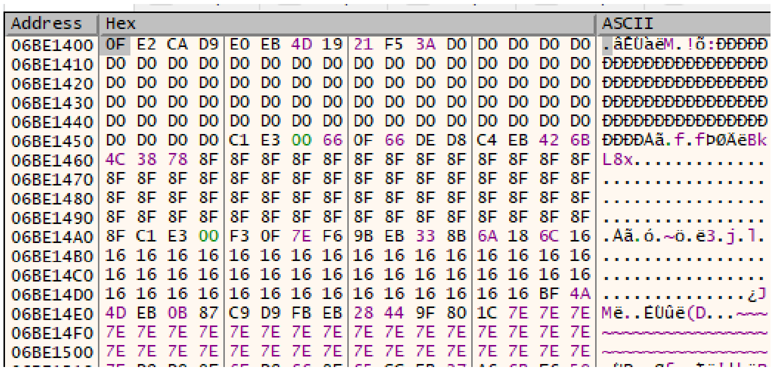


EnumResourceTypesA Function Call inside GULOADER

By reviewing the MSDN documentation for [EnumResourceTypesA](), we can see the second parameter expects a pointer to the callback function. From the screenshot above, we can see that the newly allocated shellcode is placed into this argument.

```cpp
BOOL EnumResourceTypesA(
  [in, optional] HMODULE          hModule,
  [in]           ENUMRESTYPEPROCA lpEnumFunc,
  [in]           LONG_PTR         lParam
);
```

EnumResourceTypesA Function Parameters

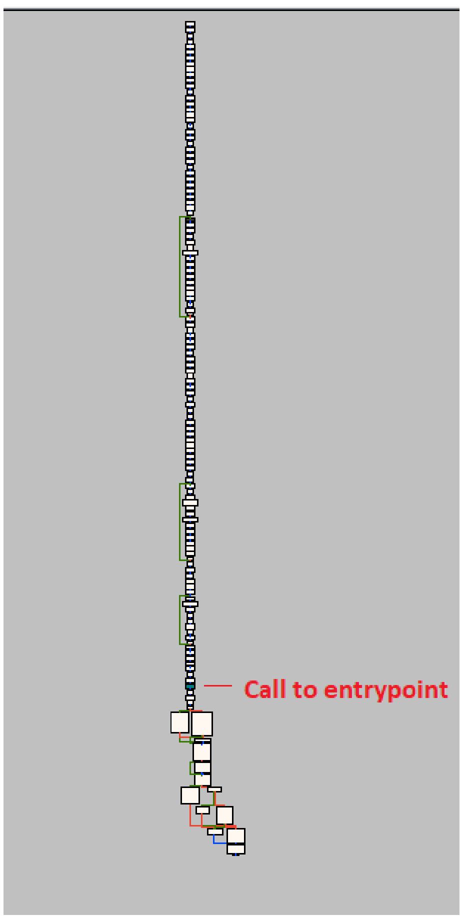


Shellcode from second parameter
EnumResourceTypesA call

Finding Main Shellcode Entrypoint

In recent samples, GULOADER has increased the complexity at the start of the initial shellcode by including many different junk instructions and jumps. Reverse engineering of the downloader can require dealing with a long process of unwinding code obfuscation designed to break disassembly and control flow in some tooling, making it frustrating to find the actual start of the core GULOADER shellcode.

One methodology for finding the initial call can be leveraging graph view inside x64dbg and using a bottom-to-top approach to look for the `call eax` instruction.



Graph view for GULOADER main entrypoint call

Another technique to trace the initial control flow involves leveraging the reversing engineering framework Miasm**.** Below is a quick example where we can pass in the shellcode and disassemble the instructions to follow the flow:

```
from miasm.core.locationdb import LocationDB
from miasm.analysis.binary import Container
from miasm.analysis.machine import Machine

with open("proctoring_06BF0000.bin", "rb") as f:
    code = f.read()

loc_db = LocationDB()
c = Container.from_string(code, loc_db)

machine = Machine('x86_32')
mdis = machine.dis_engine(c.bin_stream, loc_db=loc_db)
mdis.follow_call = True
mdis.dontdis_retcall = True
asm_cfg = mdis.dis_multiblock(offset=0x1400)
```

Miasm cuts through the 142 `jmp` instructions and navigates through the junk instructions where we have configured it to stop on the call instruction to EAX (address: `0x3bde`).

```
JMP        loc_3afd
->      c_to:loc_3afd
loc_3afd
MOV        EBX, EAX
FADDP      ST(3), ST
PANDN      XMM7, XMM2
JMP        loc_3b3e
->      c_to:loc_3b3e
loc_3b3e
SHL        CL, 0x0
PSRAW      MM1, MM0
PSRLD      XMM1, 0xF1
JMP        loc_3b97
->      c_to:loc_3b97
loc_3b97
CMP        DL, 0x3A
PADDW      XMM3, XMM5
PXOR       MM3, MM3
JMP        loc_3bde
->      c_to:loc_3bde
loc_3bde
CALL       EAX
```

Tail end of Miasm

GULOADER's VEH Update

One of GULOADER's hallmark techniques is centered around its Vectored Exception Handling (VEH) capability. This feature gives Windows applications the ability to intercept and handle exceptions before they are routed through the standard exception process. Malware families and software protection applications use this technique to make it challenging for analysts and tooling to follow the malicious code.

GULOADER starts this process by adding the VEH using `RtlAddVectoredExceptionHandler`. Throughout the execution of the GULOADER shellcode, there is code purposely placed to trigger these different exceptions. When these exceptions are triggered, the VEH will check for hardware breakpoints. If not found, GULOADER will modify the EIP directly through the CONTEXT structure using a one-byte XOR key (changes per sample) with a one-byte offset from where the exception occurred. We will review a specific example of this technique in the subsequent section. Below is the decompilation of our sample's VEH:

```
  if ( ExceptionInfo->ExceptionRecord->ExceptionCode != EXCEPTION_ACCESS_VIOLATION )
  {
    ExceptionCode = ExceptionInfo->ExceptionRecord->ExceptionCode;
    exception_code = EXCEPTION_ILLEGAL_INSTRUCTION;

    if ( ExceptionCode != EXCEPTION_ILLEGAL_INSTRUCTION )
    {
      exception_code = EXCEPTION_PRIV_INSTRUCTION;
      if ( ExceptionCode != EXCEPTION_PRIV_INSTRUCTION )
      {
        exception_code = EXCEPTION_SINGLE_STEP;
        if ( ExceptionCode != EXCEPTION_SINGLE_STEP )
        {
          exception_code = EXCEPTION_BREAKPOINT;
          if ( ExceptionCode != EXCEPTION_BREAKPOINT )
            return sub_76B3FA5(ExceptionInfo);
        }
      }
    }
LABEL_8:
    cxt_record = des_MonitorHardwareBreakpoints(exception_code);
    des::modify_EIP(&cxt_record->_Eip, (cxt_record->_Eip + 7), v4);
    return -1;
  }

  exception_code = 0x10000;
  if ( SLODWORD(ExceptionInfo->ExceptionRecord->ExceptionInformation[0]) <= 0x10000 )
    goto LABEL_8;
  return sub_76B3FA5(ExceptionInfo);
}
```

Decompilation of VEH

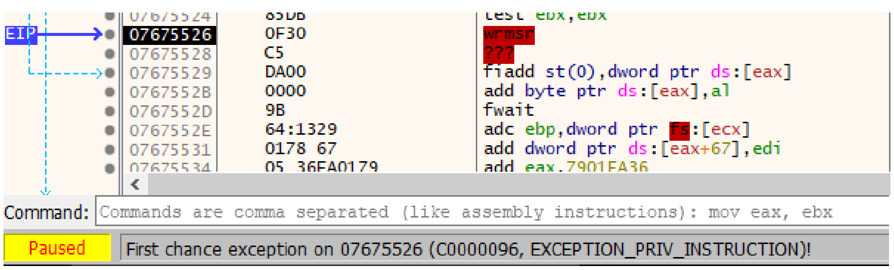
Although this technique is not new, GULOADER continues to add new exceptions over time; we have recently observed these two exceptions added in the last few months:

- `EXCEPTION_PRIV_INSTRUCTION`
- `EXCEPTION_ILLEGAL_INSTRUCTION`

As new exceptions get added to GULOADER, it can end up breaking tooling used to expedite the analysis process for researchers.

EXCEPTION_PRIV_INSTRUCTION

Let's walk through the two recently added exceptions to follow the VEH workflow. The first exception (`EXCEPTION_PRIV_INSTRUCTION`), occurs when an attempt is made to execute a privileged instruction in a processor's instruction set at a privilege level where it's not allowed. Certain instructions, like the example below with WRSMR expect privileges from the kernel level, so when the program is run from user mode, it will trigger the exception due to incorrect permissions.



EXCEPTION_PRIV_INSTRUCTION triggered by wrmsr instruction

EXCEPTION_ILLEGAL_INSTRUCTION

This exception is invoked when a program attempts to execute an invalid or undefined CPU instruction. In our sample, when we run into Intel virtualization instructions such as `vmclear` or `vmxon`, this will trigger an exception.

```
EIP ──────────────► 07697630   66:0FC730        vmclear qword ptr ds:[eax]
                    07697634   0000             add byte ptr ds:[eax],al
                    07697636   008A BB88BAE2    add byte ptr ds:[edx-1D457745],cl
                    0769763C   09B0 7FEF8B13    or dword ptr ds:[eax+138BEF7F],esi
                    07697642   88FD             mov ch,bh
                    07697644   2953 3D          sub dword ptr ds:[ebx+3D],edx
                    07697647   DD08             fisttp qword ptr ds:[eax],st(0)
                    07697649   46               inc esi
                    0769764A   8A45 E7          mov al,byte ptr ss:[ebp-19]
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
Paused   First chance exception on 07697630 (C000001D, EXCEPTION_ILLEGAL_INSTRUCTION)!
```

EXCEPTION_ILLEGAL_INSTRUCTION triggered by vmclear instruction

Once an exception occurs, the GULOADER VEH code will first determine which exception code was responsible for the exception. In our sample, if the exception matches any of the five below, the code will take the same path regardless.

- `EXCEPTION_ACCESS_VIOLATION`
- `EXCEPTION_ILLEGAL_INSTRUCTION`
- `EXCEPTION_PRIV_INSTRUCTION`
- `EXCEPTION_SINGLE_STEP`
- `EXCEPTION_BREAKPOINT`

GULOADER will then check for any hardware breakpoints by walking the CONTEXT record found inside the **EXCEPTION_POINTERS** structure. If hardware breakpoints are found in the different debug registers, GULOADER will return a `0` into the CONTEXT record, which will end up causing the shellcode to crash.

```
_CONTEXT *__usercall des_MonitorHardwareBreakpoints@<eax>(_EXCEPTION_POINTERS *ExceptionInfo@<eax>)
{
  _CONTEXT *ContextRecord; // eax

  ContextRecord = ExceptionInfo->ContextRecord;
  if ( ContextRecord->Dr0
    || ContextRecord->Dr1
    || ContextRecord->Dr2
    || ContextRecord->Dr3
    || ContextRecord->Dr6
    || ContextRecord->Dr7 )
  {
    JUMPOUT(0x76B3FA5);
  }
  return ContextRecord;
}
```

GULOADER monitoring hardware breakpoints

If there are no hardware breakpoints, GULOADER will retrieve a single byte which is 7 bytes away from the address that caused the exception. When using the last example with `vmclear`, it would retrieve byte (`0x8A`).

```
seg000:07697630 66 0F C7 30                     vmclear qword ptr [eax]
seg000:07697630                 ; ---------------------------------------
seg000:07697634 00                              db    0
seg000:07697635 00                              db    0
seg000:07697636                 ; ---------------------------------------
seg000:07697636 00 8A BB 88 BA E2               add     [edx-1D457745h], cl
seg000:0769763C
```

GULOADER retrieves a single byte, 7 bytes away from the instruction, causing an exception

Then, using that byte, it will perform an XOR operation with a different hard-coded byte. In our case (`0xB8`), this is unique per sample. Now, with a derived offset `0x32` (`0xB8 ^ 0x8A`), GULOADER will modify the EIP address directly from the CONTEXT record by adding `0x32` to the previous address (`0x7697630`) that caused the exception resulting in the next code to execute from address (`0x7697662`).

```
seg000:07697630 66 0F C7 30                vmclear qword ptr [eax]
seg000:07697630                 ; ---------------------------------------------------------------
seg000:07697634 00                         db    0
seg000:07697635 00                         db    0
seg000:07697636                 ; ---------------------------------------------------------------
seg000:07697636 00 8A BB 88 BA E2          add     [edx-1D457745h], cl
seg000:0769763C
seg000:0769763C                 loc_769763C:                       ; CODE XREF: sub_76826A3+14FB9↓j
seg000:0769763C 09 B0 7F EF 8B 13          or      [eax+138BEF7Fh], esi
seg000:07697642 88 FD                      mov     ch, bh
seg000:07697644 29 53 3D                   sub     [ebx+3Dh], edx
seg000:07697647 DD 08                      fisttp  qword ptr [eax]
seg000:07697649 46                         inc     esi
seg000:0769764A 8A 45 E7                   mov     al, [ebp-19h]
seg000:0769764D CC                         int     3               ; Trap to Debugger
seg000:0769764E 29 BC A4 A3 C6 B4          sub     [esp+var_4C4B395D], edi
seg000:0769764E B3
seg000:07697655 83 BE 49 E5 1B 9E          cmp     dword ptr [esi-61E41AB7h], 0FFFFFF96h
seg000:07697655 96
seg000:0769765C 79 DE                      jns     short loc_769763C
seg000:0769765E 94                         xchg    eax, esp
seg000:0769765F 4F                         dec     edi
seg000:07697660 75 32                      jnz     short near ptr loc_7697692+2
seg000:07697662 81 AD E2 01 00 00          sub     dword ptr [ebp+1E2h], 0EEB913B3h
seg000:07697662 B3 13 B9 EE                                        ; CODE XREF: seg000:076976D3↓j
seg000:0769766C 0F 01 1B                   lidt    fword ptr [ebx]
seg000:0769766F C9                         leave
```

Junk instructions in between exceptions

With different junk instructions in between, and repeatedly hitting exceptions (we counted 229 unique exceptions in our sample), it's not hard to see why this can break different tooling and increase analyst time.

Control Flow Cleaning

To make following the control flow easier, an analyst can bypass the VEH by tracing the execution, logging the exceptions, and patching the shellcode using the previously discussed EIP modification algorithm. For this procedure, we leveraged TinyTracer, a tool written by @hasherezade that leverages Pin, a dynamic binary instrumentation framework. This will allow us to catch the different addresses that triggered the exception, so using the example above with `vmclear`, we can see the address was `0x7697630`, generated an exception calling `KiUserExceptionDispatcher`, a function responsible for handling user-mode exceptions.

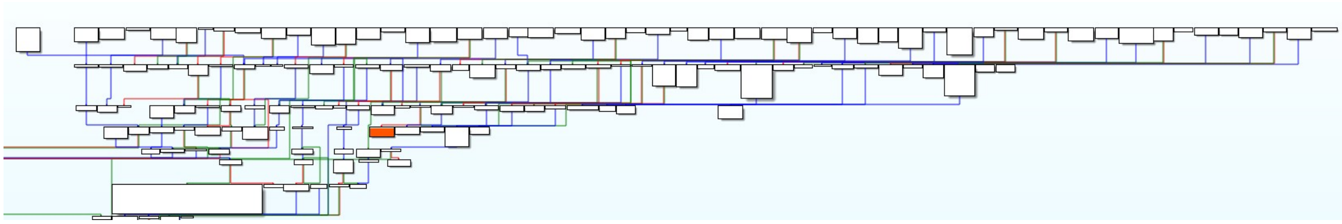
Once all the exceptions are collected and filtered, these can be passed into an IDAPython script where we walk through each address, calculate the offset using the 7th byte over and XOR key (`0xB8`), then patch out all the instructions generating exceptions with short jumps.

The following image is an example of patching instructions that trigger exceptions at addresses `0x07697630` and `0x0769766C`.

```
seg000:07697626 C7 85 E2 01 00 00 39 A7          mov     dword ptr [ebp+1E2h], 53EBA739h
seg000:07697626 EB 53
seg000:07697630 EB 30                            jmp     short loc_7697662
seg000:07697630                   ; ---------------------------------------------------------------
seg000:07697632 C7 30 00 00 00 8A BB 88…         dd 30C7h, 88BB8A00h, 0B009E2BAh, 138BEF7Fh, 5329FD88h
seg000:07697646 3D DD 08 46 8A 45 E7 CC…         dd 4608DD3Dh, 0CCE7458Ah, 0A3A4BC29h, 83B3B4C6h, 1BE549BEh
seg000:0769765A 9E 96 79 DE 94 4F 75 32          dd 0DE79969Eh, 32754F94h
seg000:07697662                   ; ---------------------------------------------------------------
seg000:07697662
seg000:07697662                   loc_7697662:                        ; CODE XREF: sub_7697125+50B↑j
seg000:07697662 81 AD E2 01 00 00 B3 13          sub     dword ptr [ebp+1E2h], 0EEB913B3h
seg000:07697662 B9 EE
seg000:0769766C EB 32                            jmp     short loc_76976A0
```

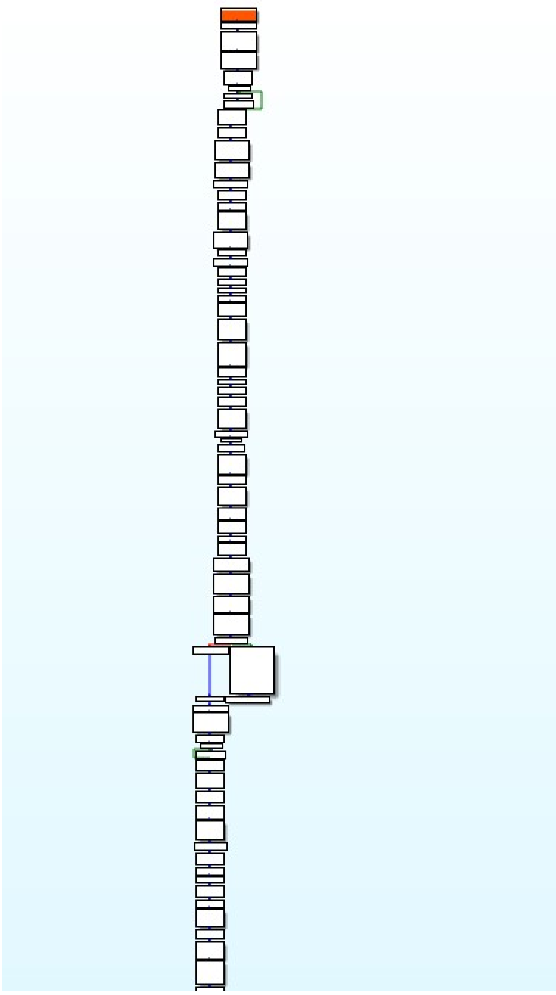
Disassembly of patched instructions

Below is a graphic representing the control flow graph before the patching is applied globally. Our basic block with the `vmclear` instruction is highlighted in orange. By implementing the VEH, GULOADER flattens the control flow graph, making it harder to trace the program logic.



GULOADER's control flow flattening obfuscation

After patching the VEH with `jmp` instructions, this transforms the basic blocks by connecting them together, reducing the complexity behind the flow of the shellcode.



GULOADER's call graph obfuscation

Using this technique can accelerate the cleaning process, yet it's important to note that it isn't a bulletproof method. In this instance, there still ends up being a good amount of code/functionality that will still need to be analyzed, but this definitely goes a long way in simplifying the code by removing the VEH. The full POC script is located here.

Conclusion

GULOADER has many different features that can break disassembly, hinder control flow, and make analysis difficult for researchers. Despite this and the process being imperfect, we can counter these traits through different static or dynamic processes to help reduce the analysis time. For example, we observed that with new exceptions in the VEH, we can still trace through them and patch the shellcode. This process will set the analyst on the right path, closer to accessing the core functionality with GULOADER.

By sharing some of our workflow, we hope to provide multiple takeaways if you encounter GULOADER in the wild. Based on GULOADER's changes, it's highly likely that future behaviors will require new and different strategies. For detecting GULOADER, the following section includes YARA rules, and the IDAPython script from this post can be found here. For new updates on the latest threat research, check out our malware analysis section by the Elastic Security Labs team.

YARA

Elastic Security has created different YARA rules to identify this activity. Below is an example of one YARA rule to identify GULOADER.

```
rule Windows_Trojan_Guloader {
    meta:
        author = "Elastic Security"
        creation_date = "2023-10-30"
        last_modified = "2023-11-02"
        reference_sample = "6ae7089aa6beaa09b1c3aa3ecf28a884d8ca84f780aab39902223721493b1f99"
        severity = 100
        arch = "x86"
        threat_name = "Windows.Trojan.Guloader"
        license = "Elastic License v2"
        os = "windows"
    strings:
        $djb2_str_compare = { 83 C0 08 83 3C 04 00 0F 84 [4] 39 14 04 75 }
        $check_exception = { 8B 45 ?? 8B 00 38 EC 8B 58 ?? 84 FD 81 38 05 00 00 C0 }
        $parse_mem = { 18 00 10 00 00 83 C0 18 50 83 E8 04 81 00 00 10 00 00 50 }
        $hw_bp = { 39 48 0C 0F 85 [4] 39 48 10 0F 85 [4] 39 48 14 0F 85 [7] 39 48 18 }
        $scan_protection = { 39 ?? 14 8B [5] 0F 84 }
    condition:
        2 of them
}
```

Observations

All observables are also available for download in both ECS and STIX format.

The following observables were discussed in this research.

Observable	Type	Name	Reference
6ae7089aa6beaa09b1c3aa3ecf28a884d8ca84f780aab39902223721493b1f99	SHA-256	Windows.Trojan.Guloader	GULOADER downloader
101.99.75[.]183/MfoGYZkxZIl205.bin	url	NA	GULOADER C2 URL
101.99.75[.]183	ipv4-addr	NA	GULOADER C2 IP

References