GuLoader's Anti-Analysis Techniques

By Hido Cohen

Published: 2021-06-29 · Archived: 2026-04-05 23:20:36 UTC

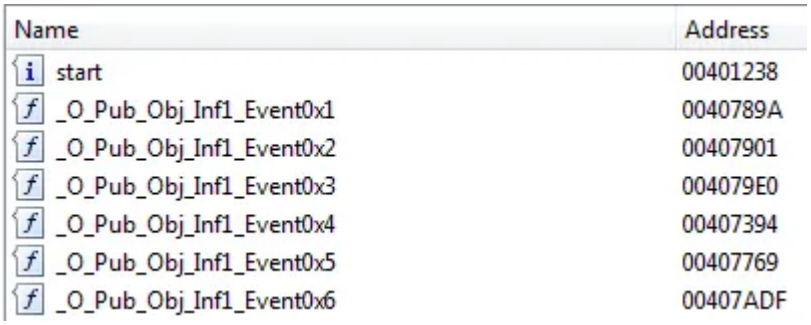
7 min read

Jun 29, 2021

GuLoader is a VB5/6 shellcode-based downloader, with many anti-analysis techniques used to make our lives, as malware researchers, much harder. In this post, I'll show you how I reversed engineer the loading mechanism and explain the different anti-analysis methods used.

Working with Visual Basic

Visual Basic isn't the most pleasant language to reverse engineer. It interacts with only one DLL, `MSVBVM60.DLL` and has a different structure than `C, C++` or any other famous programming language. Fortunately, IDA has an `idc` script written by Reginald Wong which parses the VB headers which identifies the form event functions.



Name	Address
start	00401238
_O_Pub_Obj_Inf1_Event0x1	0040789A
_O_Pub_Obj_Inf1_Event0x2	00407901
_O_Pub_Obj_Inf1_Event0x3	004079E0
_O_Pub_Obj_Inf1_Event0x4	00407394
_O_Pub_Obj_Inf1_Event0x5	00407769
_O_Pub_Obj_Inf1_Event0x6	00407ADF

Event functions

Where's *VirtualAlloc* ?

It isn't new that VB malwares are sometimes used for injecting another malicious code (PE/Shellcode). So, I decided to set up a breakpoint at `VirtualAlloc` and see if this malware is also part of the group. And indeed, I noticed that a shellcode is being written into a newly allocated memory section.

Looking at the code in IDA didn't see any reference to `VirtualAlloc` , which means that the function resolved dynamically. I located the function call in my debugger, and moved back to IDA to get a better understanding of what happened.

The first thing I've noticed is that the malware uses the following techniques in order to harden my analysis:

- High amount of `JMP` s

- Inflating redundant instructions, for example, `pushfd` followed by `popfd` and `mov ebx, ebx`
- Obfuscated values using predefined calculations

Press enter or click to view image in full size

```
Python>0x0FF86B556 + 0x0A94578 + 0x0D7198B + 0x8D0B95 - 0x10ACBAE + 0x7060D
0x100905a4d
```

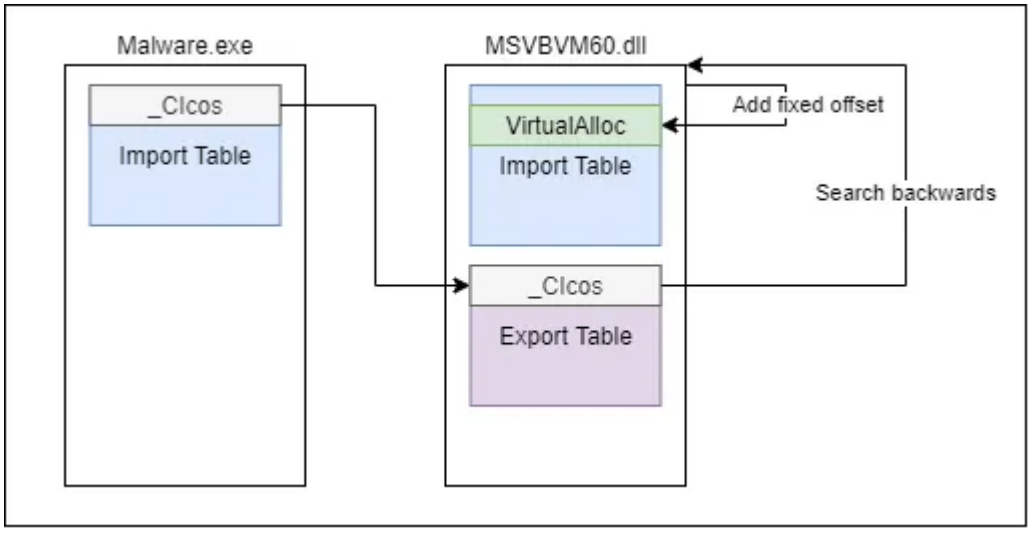
Press enter or click to view image in full size

```
Python>0x438160 - 0x40C26D + 0x2D998F ^ 0x48B21 ^ 0x74C3A3
0x401000
```

Magic bytes and first import address calculation

`VirtualAlloc` resolving process is:

1. Calculate the address of the first import from `MSVBVM60.dll` (as shown in the figure above)
2. Locate `VirtualAlloc` inside `MSVBVM60.dll`'s import table — this done by searching backward from the function address in step (1) to the magic value calculated in the figure above. Once, the base address found the malware adds hardcoded value of `0x10CC` which points to `VirtualAlloc` import table entry
3. Copy the address of `VirtualAlloc` for the import table



Shellcode’s Decryption and Execution

The shellcode is stored encrypted inside a global variable, which is copied into the newly created section.

```
loc_402E49:
                        nop
                        push    offset G_encrypted_shellcode
                        mov     ecx, ecx
                        nop
                        jmp     short loc_402E81
; ---------------------------------------------------------------------------
                        dd 0Bh dup(0FFFFFFFFh)
                        db 0FFh
; ---------------------------------------------------------------------------

loc_402E81:
                        mov     ecx, ecx
                        pop     esi
                        mov     esi, esi
                        nop
                        jmp     short loc_402EFB
```

Load the encrypted shellcode's address into ESI in an unusual way

Then, the malware decrypts the shellcode:

Press enter or click to view image in full size

```
do
{
  *(shellcode_address + i) ^= 0xD3329873;
  i += 4;
}
while ( i != new_section_data->dwSize );
```

XOR-decryption

At the end, the malware jumps to the new section address and starts executing the shellcode.

Shellcode Analysis

Looking at the strings inside the shellcode suggest that the shellcode also uses some kind of strings obfuscation and dynamic function loading/resolving:

```
C:\Program Files\Qemu-ga\qemu-ga.exe
C:\Program Files\qga\qga.exe
Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
Set W = CreateObject("WScript.Shell")\r\nSet C = W.Exec ("
Software\Microsoft\Windows\CurrentVersion\RunOnce
Msi.dll
wininet.dll
Publisher
kernel32
```

```
advapi32
user32
ntdll
shell32
windir=
msvbvm60.dll
\syswow64\
\METEROLO
```

Before continuing my research, I wanted to understand if and how the malware resolves those obfuscated strings and functions. I saw that the malware uses DJB hashing algorithm:

```
unsigned int DJBHash(const char* str, unsigned int length)
{
   unsigned int hash = 5381;
   unsigned int i    = 0;

   for (i = 0; i < length; ++str, ++i)
   {
      hash = ((hash << 5) + hash) + (*str);
   }

   return hash;
}
```

Press enter or click to view image in full size

```
023F3 mw_hash_function_name proc near         ; COD
023F3                                         ; sub
023F3
023F3 arg_0           = dword ptr  4
023F3
023F3                 mov     esi, [esp+arg_0]
023F7                 mov     eax, 5381
023FC
023FC loc_23FC:                               ; COD
023FC                 cmp     bx, bx
023FF                 cmp     byte ptr [esi], 0A4h
02402                 jnb     short loc_2419
02404                 mov     ebx, eax
02406                 shl     eax, 5
02409                 add     eax, ebx
0240B                 movzx   ecx, byte ptr [esi]
0240E                 add     eax, ecx
02410                 inc     esi
02411                 cmp     byte ptr [esi], 0
02414                 jnz     short loc_23FC
02416                 retn    4
02419 ; ---------------------------------------
02419
02419 loc_2419:                               ; COD
02419                 test    bl, bl
0241B                 xor     eax, eax
0241D                 retn    4
0241D mw_hash_function_name endp
0241D
```

DJB hash function — source code VS malware implementation

Using that hashing algorithm, the malware parses the PE headers of the requested DLL and look for the function name hash in the export table:

Press enter or click to view image in full size

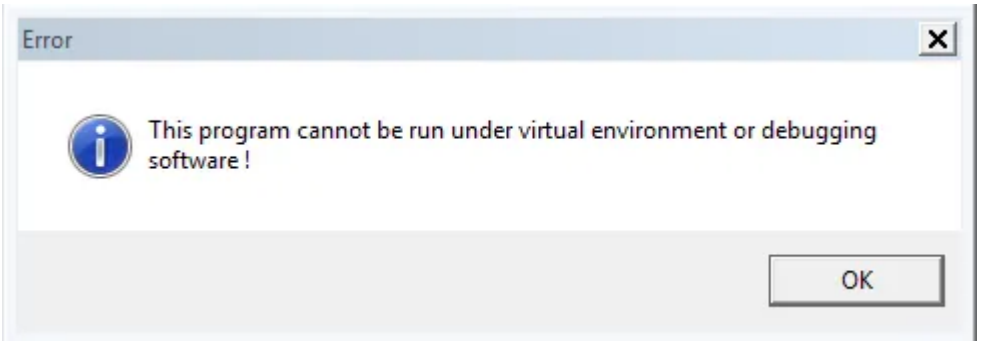
```
function_data->DllBaseAddress = base_address;
export_directory = (PIMAGE_EXPORT_DIRECTORY)(*(_DWORD *)(*(_DWORD *)(base_address + 0x3C) + base_address + 0x78)
                                           + function_data->DllBaseAddress);
function_data->Name = export_directory->NumberOfNames;
function_data->AddressOfFunctions = export_directory->AddressOfFunctions;
function_data->AddressOfNameOrdinals = export_directory->AddressOfNameOrdinals;
address_of_names = (_DWORD *)(function_data->DllBaseAddress + export_directory->AddressOfNames);
index = 0;
do
{
  function_name_hash = mw_djb2_hash((_BYTE *)(function_data->DllBaseAddress + *address_of_names));
  if ( function_name_hash == function_name_hash_to_find )
    break;
  ++address_of_names;
  ++index;
}
while ( index + 1 != function_data->Name );
```

Loop for every export table entry and look for name with the suitable hash

After labeling the function that is responsible for dynamic function loading and the hash function I was ready to move forward to reveal the different anti-analysis techniques used by the malware.

Anti-Analysis Techniques

Running the malware without any changes will result with the following message:



Error message — VM/Debugger detected

And the malware will terminate itself.

How did it find out that it is running inside a VM or there’s a debugger attached?

#1 — VM Detection 1 — Memory Scan

The malware scans the process’s virtual memory address space using `ZwQueryVirtualMemory` WinAPI and uses pre-calculated string hashes (again, using DJB hash function).

```
while ( 1 )
{
  BaseAddress += 0x1000;
  if ( BaseAddress == 0x7FFFF000 )
    break;
  ZwQueryVirtualMemory(-1, BaseAddress, 0, &MemoryInformation, 28, 0);
  if ( MemoryInformation.Protect == PAGE_EXECUTE
    || MemoryInformation.Protect == PAGE_EXECUTE_READ
    || MemoryInformation.Protect == PAGE_EXECUTE_READWRITE
    || MemoryInformation.Protect == PAGE_READONLY
    || MemoryInformation.Protect == PAGE_READWRITE )
  {
    function_address = 0x1000;
    while ( function_address )
    {
      --function_address;
      if ( !*(_BYTE *)(BaseAddress + function_address) )
      {
        index = -1;
        function_address = BaseAddress;
        while ( ++index < function_address )
        {
          if ( *(_BYTE *)(BaseAddress + index) )
          {
            string_hash = mw_djb2_hash((_BYTE *)(index + BaseAddress));
            hash_index = 0;
            while ( hashes_to_find[++hash_index] )
            {
              if ( hashes_to_find[hash_index] == string_hash )
              {
                msvbvm60_base_address = (int)mw_find_dll_name_by_hash(0xB95DDAB0);
```

Memory scan routine

Once the malware detects one of the following string hashes it will display the message box and terminate itself.

```
0x2D9CC76C
0xDFCB8F12
0x27AA3188
0xF21FD920
0x3E17ADE6
0xA7C53F01 \\ "VBoxTrayToolWndClass"
0x7F21185B \\ "HookLibraryx86.dll"
0xB314751D \\ "vmtoolsdControlWndClass"
```

#2 — VM Detection 2— Hypervisor Feature Bit

In this method, the malware utilizes the `cpuid` instruction with `EAX=1` . This means that the result will be stored inside `ECX` and `EDX` , where the 31st bit of `ECX` register is the hypervisor feature bit.

The hypervisor feature bit indicate a hypervisor present, which means that this value is always zero for physical-CPUs.

```
025BE mw_vm_detection_cpuid_1 proc near       ; CODE XREF: m
025BE                                         ; mw_vm_detect
025BE                 call    mw_read_timestamp_counter
025C3                 mov     esi, edx
025C5                 pusha
025C6                 mov     eax, 1
025CB                 cpuid
025CD                 bt      ecx, 31
025D1                 jb      mw_vm_detected
025D7                 popa
025D8                 call    mw_read_timestamp_counter
025DD                 sub     edx, esi
025DF                 cmp     edx, 0
025E2                 jle     short mw_vm_detection_cpuid_1
025E4                 retn
025E4 mw_vm_detection_cpuid_1 endp
```

Check the 31st bit of ECX after CPUID instruction

#3 — VM Detection 3— QEMU Guest Agent

The malware checks if QEMU related files exist on the infected system:

```
176B mw_check_file_and_terminate proc near   ; CC
176B                                         ; su
176B                 pop     ebx
176C                 push    ebx
176D                 call    mw_open_file_handle
1772                 cmp     eax, 0FFFFFFFFh
1775                 jnz     mw_vm_detected
177B                 cmp     bl, al
177D                 retn
```

Check if the file can be opened

It searches in the following files:

- C:\Program Files\Qemu-ga\qemu-ga.exe
- C:\Program Files\qga\qga.exe

#4 — Anti-Sandbox 1 — RTDSC Wrapper

The `rtdsc` instruction is used to determine how many CPU ticks took place since the processor was reset, which can be used for Sandbox/VM detection mechanism.

Get Hido Cohen's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

In the case of that malware, as we can see at #2, if the readings are the same, the malware enters into an endless loop.

#5— Anti-Sandbox 2— Windows Enumeration

The malware uses `EnumWindows` WinAPI in order to count the top-level windows running on the system, if the number is lower than 12, it will call `TerminateProcess`.

#6— Anti-Sandbox 3— Long Delays

The malware calls to the same function many times in order to extend the execution time of the program before revealing its real intentions. For example,

```
for ( delay_counter = 0x186A0; delay_counter != 1; --delay_counter )
{
  mw_vm_detection_cpuid_1();
  if ( v3 <= 0x31 )
    ++data->DelayCounter;
  v1 += v3;
}
```

Calling 100,000 times to the same function

This is technique used in order to evade sandboxes since they're usually time-limited.

#7— Anti-Attaching— Patch Important Functions

The malware patches `DbgBreakPoint` and `DbgUiRemoteBreakin` calls which are being used by debuggers and disrupting their actions.

```
02963                   push    PAGE_EXECUTE_READWRITE ; new_permissions
02965                   call    mw_wrap_ZwProtectVirtualMemory
0296A                   push    esi
0296B                   mov     esi, 836F406Fh
02970                   cmp     esi, 836F406Fh
02976                   jnz     loc_19D
0297C                   pop     esi
0297D                   cmp     eax, 0
02980                   jnz     locret_2AE6
02986                   test    dl, al
02988                   mov     eax, [esp+DbgBreakPoint_addr]
0298C                   mov     byte ptr [eax], 90h ; nop
0298F                   test    cl, 4Ah
02992                   mov     eax, [esp+DbgUiRemoteBreakin_addr]
02996                   mov     byte ptr [eax], 6Ah ; 'j'
02999                   mov     byte ptr [eax+1], 0 ; push 0
0299D                   mov     byte ptr [eax+2], 0B8h
029A1                   test    cx, cx
029A4                   test    dx, ax
029A7                   mov     edx, [ebp+ExitProcess]
029AD                   mov     [eax+3], edx    ; mov eax, <ExitProcess>
029B0                   mov     byte ptr [eax+7], 0FFh
029B4                   test    edi, 8FFE0B68h
029BA                   mov     byte ptr [eax+8], 0D0h ; call eax
029BE                   mov     byte ptr [eax+9], 0C2h
029C2                   mov     byte ptr [eax+0Ah], 4 ; ret 4
```

Change protection and patch functions

Before patching the functions, the malware needs to change the page protection of `ntdll.dll` to `RWX`.

Then, the malware writes `0x90 (NOP)` into `DbgBreakPoint` and `kernel32!ExitProcess` calls inside `DbgUiRemoteBreakin`.

Press enter or click to view image in full size

Press enter or click to view image in full size

The patched functions

#8 — Defense Evasion— WinAPI Function Hooks Removal

Security products (AV, EDR, etc…) usually insert `JMP` instruction in the first 5 bytes of WinAPI functions to execute their code before the real function being called. This allows them to have better control over the process's actions.

The malware searches for known syscall patterns inside `ntdll`, for example, in the figure below we can see that the malware searches for the pattern:

```
B8 ?? ?? ?? ??
B9 ?? ?? ?? ??
8D 54 24 04
```

```
if ( *current_byte_1 == (char)0xB8 && !*(_DWORD *)(current_byte_1 + 1) && current_byte_1[5] == (char)0xB9 )
{
  v22 = current_byte_1 + 10;
  v23 = 0;
  syscall_num = 1;
  do
  {
    ++v23;
    if ( *(_DWORD *)++v22 == 0x424548D )  // lea edx, [esp+arg_0]
    {
      if ( *((_WORD *)v22 - 1) != 0x33C9 && *(v22 - 5) == (char)0xB9 )
      {
        *(v22 - 10) = 0xB8;
        *(_DWORD *)(v22 - 9) = syscall_num++;
      }
      else
      {
        *(v22 - 7) = 0xB8;
        *(_DWORD *)(v22 - 6) = syscall_num++;
      }
    }
  }
```

Unhooking routine

This pattern matches, for example, to `ZwReleaseMutant` function call:

```
7DE8FBBC                                public ZwReleaseMutant
7DE8FBBC                ZwReleaseMutant proc near               ; CODE XREF: sub_7DEBFD5C+4A2BE↓p
7DE8FBBC                                                        ; sub_7DF54DCC+A↓p
7DE8FBBC                                                        ; DATA XREF: ...
7DE8FBBC
7DE8FBBC                arg_0           = byte ptr  4
7DE8FBBC
7DE8FBBC B8 1D 00 00 00                 mov     eax, 1Dh        ; NtReleaseMutant
7DE8FBC1 B9 07 00 00 00                 mov     ecx, 7
7DE8FBC6 8D 54 24 04                    lea     edx, [esp+arg_0]
7DE8FBCA 64 FF 15 C0 00 00 00           call    large dword ptr fs:0C0h
7DE8FBD1 83 C4 04                       add     esp, 4
7DE8FBD4 C2 08 00                       retn    8
```

ZwReleaseMutant pattern

As we can see, after the malware finds the pattern, it extracts the syscall number and restores the function call to its appropriate structure.

#9 — Anti-Analysis — Check Installed Software and Running Services

The malware utilizes `MsiEnumProduct` and `MsiGetProductInfo` functions in order to iterate over the installed software in the system.

For each installed software, the malware checks if the Publisher is inside its black list.

```
data->MsiGetProductInfoA(lpProductBuf, szAttribute, lpProductBuf + 255);
product_publisher_hash = mw_djb2_hash(lpProductBuf + 255);
if ( product_publisher_hash != 0x7C8AA9FD
  && product_publisher_hash != 0x9B8FFB51
  && product_publisher_hash != 0x555E1691
  && product_publisher_hash != 0xCE81C85D )
```

Publisher black listing

The same thing is done with running services:

Press enter or click to view image in full size

```
OpenSCManagerA = (void *)mw_find_and_load_function(data->advapi32_str, 0xBAEF4789);
mw_call_protected_function(data, OpenSCManagerA, 0, 0, SC_MANAGER_ENUMERATE_SERVICE);
if ( SCManager )
{
  data->hSCManager = SCManager;
  EnumServicesStatusA = (void *)mw_find_and_load_function(data->advapi32_str, 0x19C2C923);
  mw_call_protected_function(
    data,
    EnumServicesStatusA,
    data->hSCManager,
    SERVICE_WIN32,
    SERVICE_STATE_ALL,
    &data->Allocated_Memory->lpServices,
    0x3E800,
    (char *)&data->Allocated_Memory->field_10005 + 3,
    (char *)&data->Allocated_Memory->gap10001 + 3,
    &data->Allocated_Memory->char10000);
  if ( EnumServicesStatusA_result )
  {
    dwBufferSize = 0x3E800;
    lpServices = &data->Allocated_Memory->lpServices;
    while ( 1 )
    {
      v35 = dwBufferSize - 1;
      service_hash = mw_djb2_hash((_BYTE *)lpServices + dwBufferSize - 1);
      dwBufferSize = v35;
      v6 = (char *)&loc_0 + 1;
      if ( (char *)&loc_0 + 1 == (char *)&loc_0 + 1
        && (service_hash == 0x30871D6D || service_hash == 0xD03596C8 || service_hash == 0x1B7912B2) )
      {
        break;
      }
```

Services black listing

The malware enumerates the running services and searches for the following service names hash (probably security products):

```
0x30871D6D
0xD03596C8
0x1B7912B2
```

#10— Anti-Debugging 1 — Protected Function Calls

The malware implemented a Win32 API function calls wrapper which perform checks before calling the actual function:

Press enter or click to view image in full size

```
if ( !((int (__cdecl *)(int, PCONTEXT))a1->NtGetContextThread)(-2, allocated_memory->ThreadContext) )
{
  ThreadContext_1 = allocated_memory->ThreadContext;
  if ( !ThreadContext_1->Dr0 && !ThreadContext_1->Dr1 && !ThreadContext_1->Dr2 )
  {
    sub_302D();
    if ( !ThreadContext->Dr3
      && !ThreadContext->Dr6
      && !ThreadContext->Dr7
      && *(_BYTE *)Win32API_call != 0xCC
      && *(_WORD *)Win32API_call != 0x3CD
      && *(_WORD *)Win32API_call != 0xB0F )
    {
      Win32API_call();
```

Checks the malware does before calling a certain function

Using `NtGetThreadContext` , the malware checks if there are any hardware breakpoints (by inspecting the `DRx` registers values) and software breakpoint (represented by `0xCC` , `0x3CD` and `0xB0F`).

The arguments to the Win32API function pushed earlier to the stack.

#11 — Anti-Debugging 2— ThreadHideFromDebugger

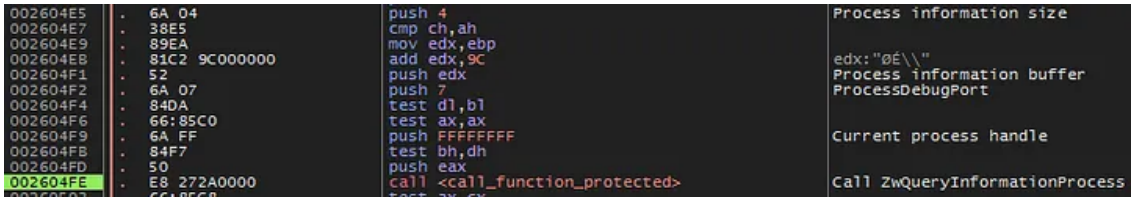
The malware calls to `NtSetInformationThread` with `THREADINFOCLASS.ThreadHideFromDebugger` flag.

```
push    0
push    0
push    11h             ; ThreadHideFromDebugger
push    0FFFFFFFEh      ; Current thread macro
call    eax             ; NtSetInformationThread
```

#12 — Anti-Debugging 3— ProcessDebugPort

The malware uses `ZwQueryInformationProcess` to detect if a debugger is attached to the process.

Press enter or click to view image in full size



According to MSDN:

> *Retrieves a DWORD_PTR value that is the port number of the debugger for the process. A nonzero value indicates that the process is being run under the control of a ring 3 debugger.*

Conclusions

GuLoader implements many anti-analysis techniques and uses different methods which makes the analysis harder. After reading this post you have the knowledge of how to overcome those techniques. Those techniques used in many other malwares which you now be able to identify in your research.

Hope you found this post useful :)

References

[1] http://sandsprite.com/vb-reversing/

[2] https://theartincode.stanis.me/008-djb2/

[3] https://en.wikipedia.org/wiki/CPUID#EAX.3D1:_Processor_Info_and_Feature_Bits

[4] https://www.dimva2019.org/wp-content/uploads/sites/31/2019/06/DIMVA19-slides-13.pdf

[5] https://www.crowdstrike.com/blog/guloader-malware-analysis/
