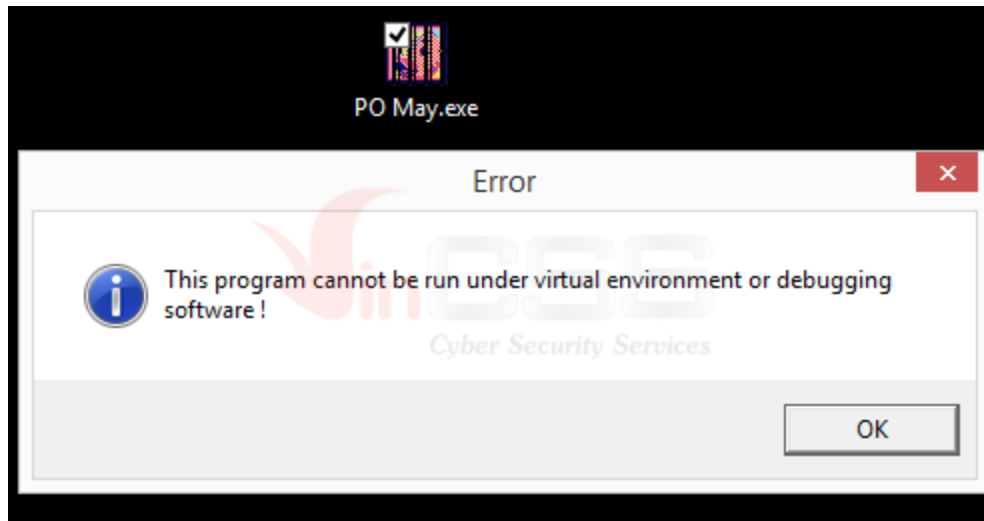


[RE014] GuLoader AntiVM Techniques

blog.vincss.net/2020/05/re014-guloader-antivm-techniques.html



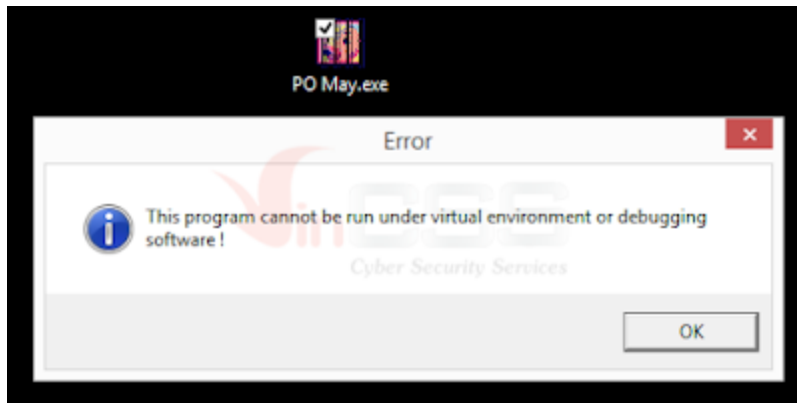
Thời gian gần đây, Twitter của các chuyên gia nước ngoài liên tục xuất hiện hash tag **#GuLoader**, đây là một downloader phổ biến đang được các nhóm tin tặc sử dụng rộng rãi nhằm tải về mã độc chính lợi dụng các dịch vụ cloud của Google Drive và Microsoft OneDrive. GuLoader rất nhỏ, nhẹ, được viết bằng VB6 và thường được nén trong file .rar. Khi người dùng vô tình thực thi loader này, nó sẽ tải xuống Trojans (RAT) hoặc các dòng mã độc đánh cắp thông tin như Agent Tesla, FormBook, NanoCore RAT, Netwire RAT, Remcos RAT, ...

Ở Việt Nam cũng không ngoại lệ, chúng tôi đã tiếp cận và có bài phân tích chi tiết về downloader này. Mới đây, khách hàng của chúng tôi tiếp tục nhận được email có file đính kèm lạ, qua kiểm tra, phân tích và so sánh, chúng tôi nhận thấy đây chính là một biến thể của GuLoader. Nhiệm vụ của nó là tải về NanoCore RAT để thực thi trên máy người dùng. Trong bài viết này, chúng tôi sẽ không đi vào phân tích chi tiết mà chỉ tập trung vào các kỹ thuật Anti-VM được sử dụng trong shellcode.

1. Anti-VM

1.1. Kết hợp ZwQueryVirtualMemory với các hash đã tính toán trước

Thông thường, để phân tích mã độc, người phân tích sẽ thực thi mã độc trên một môi trường riêng biệt nhằm thu thập các thông tin tương tác với hệ thống trong quá trình thực thi. Tuy nhiên, với các biến thể mới của Guloader, khi thực thi sẽ nhận được thông báo sau:



Hình 1. Thông báo lỗi khi thực thi trên môi trường ảo hóa hoặc thông qua debugger

Debug loader sẽ tới được shellcode. Shellcode thực hiện push một loạt các hash đã tính toán trước lên stack:

```
push 0xB314751D
push 0xA7C53F01
push 0x7F21185B
push 0x3E17ADE6
push 0xF21FD920
push 0x27AA3188
push 0xDFCB8F12
push 0x2D9CC76C
```

Tiến hành resolve hàm API **ZwQueryVirtualMemory** ứng với hash đã tính toán trước là **0x8802EDAC**.

```
ZwQueryVirtualMemory = f_get_api_func_w(a1→ntdll, ZwQueryVirtualMemory_hash);
```

Hình 2. Lấy địa chỉ hàm API ZwQueryVirtualMemory

Tiếp theo sử dụng vòng lặp để quét toàn bộ vùng nhớ từ **0x00010000** tới **0x7FFFF000**, gọi hàm **ZwQueryVirtualMemory** kiểm tra access protection của các vùng nhớ này. Nếu vùng nhớ thỏa mãn điều kiện, sẽ quét vùng nhớ đó, gặp chuỗi sẽ gọi hàm tính toán hash cho chuỗi đó và so sánh với các hash đã thiết lập trên Stack. Khi trùng hash, loader lấy địa chỉ base của **msvbvm60.dll** để tìm hàm **MessageBoxA**, giải mã chuỗi "This program cannot be run under virtual environment or debugging software !" và hiển thị thông báo như Hình 1.

```

MemRegion = 0xF000;
LABEL_2:
while ( 1 )
{
    MemRegion = MemRegion + 0x1000; // 0x00010000
    if ( MemRegion == 0x7FFF000 )
    {
        return 0;
    }
    ZwQueryVirtualMemory(0xFFFFFFFF, MemRegion, 0, &MemoryInformation, 0x1Cu, 0);
    if ( access_protection == PAGE_EXECUTE
        | access_protection == PAGE_EXECUTE_READ
        | access_protection == PAGE_EXECUTE_READWRITE
        | access_protection == PAGE_READONLY
        | access_protection == PAGE_READWRITE )
    {
        delta = 0x1000;
        while ( delta )
        {
            if ( !*(MemRegion + --delta) )
            {
                i = -1;
                while ( ++i < delta )
                {
                    if ( *(MemRegion + i) )
                    {
                        v13 = delta;
                        v12 = i;
                        v11 = MemRegion;
                        hash_calced = calc_hash(MemRegion + i);
                        MemRegion = v11;
                        i = v12;
                        delta = v13;
                        j = 0;
                        while ( 1 )
                        {
                            j += 4;
                            if ( *(&retaddr + j) == -1 )
                            {
                                break;
                            }
                            if ( *(&retaddr + j) == hash_calced )
                            {
                                JUMPOUT(loc_87B6);
                            }
                        }
                    }
                }
            }
        }
        goto LABEL_2;
    }
}

```

Hình 3. Tính toán hash của các chuỗi trên vùng nhớ

Hàm tính toán hash mà loader sử dụng chính là thuật toán djb2:

```

int __cdecl calc_hash(_DWORD *szInputString)
{
    _BYTE *szStr; // esi
    int hash_calced; // eax

    szStr = szInputString;
    hash_calced = 0x1505;
    while ( *szStr < 0xA4u )
    {
        hash_calced = *szStr++ + 0x21 * hash_calced;
        if ( !*szStr )
        {
            return hash_calced;
        }
    }
    return 0;
}

```

Hình 4. Hàm tính toán hash

Với máy sử dụng để debug loader, chúng tôi có được chuỗi “vmtoolsdControlWndClass” trùng với một hash đã được loader tính toán trước là 0xB314751D. Các hash còn lại theo phỏng đoán của chúng tôi có thể liên quan đến VirtualBox hoặc môi trường sandbox khác.

The screenshot displays a debugger interface with three main sections:

- Disassembly:** Shows assembly instructions for the `calc_hash` function, including `call 00568002`, `mov edx, eax`, `pop esi`, `pop ecx`, `pop ebx`, `xor eax, eax`, `add eax, 0x4`, `cmp dword ptr [esp+eax], -0x1`, `short 0x5687A7`, and `jmp 0xB9510A00`.
- Registers (FPU):** Lists CPU registers. `ECX` is highlighted with the value `0xB314751D`, with a red arrow pointing to it and the label `calculated_hash`.
- Memory Dump:** Shows a hex dump of memory starting at address `005DC090`. The ASCII column shows the string `vmtoolsdControlWndClass`.

Below the memory dump, a list of hashes is shown with their corresponding addresses and values:

Address	Value	Comment
0013F5D8	005601E0	RETURN to 005601E0 from 0056855E
0013F5DC	2D9CC76C	
0013F5E0	DFC88F12	
0013F5E4	27AA3188	
0013F5E8	F21FD920	
0013F5EC	3E17AD86	
0013F5F0	7F21185B	
0013F5F4	A7C53F01	
0013F5F8	B314751D	pre-computed hash
0013F5FC	FFFFFFFF	
0013F600	0013F864	

Hình 5. Phát hiện môi trường ảo hóa sử dụng VMware

1.2. Kiểm tra sự tồn tại của QEMU Guest Agent

Bên cạnh kĩ thuật nói trên, loader cũng thực hiện kiểm tra xem môi trường phân tích có sử

dùng Qemu guest agent (qga) hay không thông qua hàm **CreateFileA**.

```
a2→CreateFileA = f_get_api_func_w(a2→kernel32, CreateFileA_hash);
```

Hình 6. Lấy địa chỉ hàm API CreateFileA

Sử dụng hàm này để kiểm tra sự tồn tại của “C:\ProgramData\qemu-ga\qga.state” trên máy:

Address	Value	Comment
0013F5D4	002743CA	CALL to CreateFileA from 002743C4
0013F5D8	002743D2	FileName = "C:\ProgramData\qemu-ga\qga.state"
0013F5DC	80000000	Access = GENERIC_READ
0013F5E0	00000001	ShareMode = FILE_SHARE_READ
0013F5E4	00000000	pSecurity = NULL
0013F5E8	00000003	Mode = OPEN_EXISTING
0013F5EC	00000000	Attributes = 0
0013F5F0	00000000	hTemplateFile = NULL

Hình 7. Gọi hàm CreateFileA để kiểm tra sự tồn tại của file

Nếu tồn tại file trên hệ thống sẽ hiển thị thông báo như Hình 1:

```
void __usercall f_CreateFileA_w(int a1@<ebp>, _DWORD *edi0@<edi>)  
{  
    HANDLE hmsvbvm; // eax  
    void (*v3)(void); // eax  
    const CHAR *retaddr; // [esp+0h] [ebp+0h]  
  
    // "C:\ProgramData\qemu-ga\qga.state"  
    if ( f_CreateFileA(a1, retaddr) != INVALID_HANDLE_VALUE )  
    {  
        hmsvbvm = get_module_base_addr_from_hash(MSVBVM60_DLL_HASH);  
        f_wipe_sc_decrypt_nag_show_msg(a1, edi0, *(hmsvbvm + 0x4EF));  
        f_wipe_sc_decrypt_nag_show_msg(a1, edi0, v3);  
    }  
}
```

Hình 8. Hiển thị thông báo nếu có qga.state

1.3. Sử dụng CPUID

Bên cạnh hai kỹ thuật trên, loader còn sử dụng thêm lệnh **CPUID** để kiểm tra xem chương trình có đang thực thi trong môi trường ảo hóa hay không?

```
f_call_to_cpuid proc near                                ; CODE XREF: f_call_cpuid_w+8↑p
                                                         ; f_call_to_cpuid+30↓j
    lfence
    rdtsc
    lfence
    shl     edx, 20h
    or     edx, eax
    mov    esi, edx
    pusha
    mov    eax, 1
    cpuid                                ; call cpuid with argument in EAX
    bt     ecx, 1Fh                       ; On a guest VM it will equal to 1.
    jnb   short $+2
```

Hình 9. Sử dụng CPUID để kiểm tra

Lệnh CPUID được thực thi với giá trị của EAX = 1. Bit thứ 31 của ECX nếu trên máy vật lý sẽ bằng 0. Trên máy ảo, nó sẽ bằng 1.

2. Bonus

Loader này sử dụng kỹ thuật code injection phức tạp nhằm làm khó người phân tích.

- Tạo một tiến trình con là **RegAsm.exe** ở trạng thái suspended.
- Map **msvbvm60.dll** vào tiến trình vừa tạo.
- Thực hiện inject shellcode vào **RegAsm.exe**.
- Thiết lập context của tiến trình nhằm chuyển hướng thực thi tới đoạn code đã inject và gọi hàm **ZwResumeThread** để thực thi.

Shellcode thực thi bên trong tiến trình **RegAsm.exe** sẽ thực hiện tải về, giải mã và thực thi payload. Payload mới được lưu trên Google Drive:



Hình 10. Loader thực hiện tải payload từ Google Drive

Giải mã payload và map vào memory để thực thi. Payload có kích thước **0x32A00**:

Address	Hex dump	ASCII
01A70040	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ?L...J... * ..
01A70050	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	?.....@.....
01A70060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01A70070	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00 €..
01A70080	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	þ?.???L?Th
01A70090	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program cannot
01A700A0	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	be run in DOS
01A700B0	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$......
01A700C0	50 45 00 00 4C 01 03 00 A1 27 E9 54 00 00 00 00	PE..L L.?門...
01A700D0	00 00 00 00 E0 0E 01 0B 01 06 00 00 C8 01 00	...?þ ? -..?
01A700E0	00 60 01 00 00 00 00 00 92 E7 01 00 00 20 00 00 擊 ..
01A700F0	00 00 02 00 00 00 40 00 00 20 00 00 00 02 00 00	..?..@.. ..?
01A70100	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	J.....J.....
01A70110	00 80 03 00 00 02 00 00 00 00 00 00 02 00 00 00	€L..?.....?
01A70120	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	..+.+.+.+.+.+
01A70130	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00	...+.+.+.+.+
01A70140	38 E7 01 00 57 00 00 00 00 20 02 00 A0 5D 01 00	8?.W.... ?.燦 .
01A70150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Hình 11. Payload sau khi tải về được giải mã trên memory

Payload thu được chính là **NanoCore RAT**:

File pos	Mem pos	ID	Text
A 0000000B93C	00000040D73C	0	#Blob
A 0000000FAA0	0000004118A0	0	(.49>Cj)
A 0000000FCC3	000000411AC3	0	IHOt
A 0000000FEF5	000000411CF5	0	NanoCore Client
A 0000000FF05	000000411D05	0	NanoCore Client.exe
A 0000000FF19	000000411D19	0	mscorlib
A 0000000FF25	000000411D22	0	Microsoft.VisualBasic
A 0000000FF38	000000411D38	0	System.Windows.Forms
A 0000000FFAD	000000411DAD	0	System

Hình 12. Payload thu được là NanoCore RAT

Tran Trung Kien (aka m4n0w4r)
Dang Dinh Phuong
R&D Center - VinCSS (a member of Vingroup)