Deep Analysis of SmokeLoader

June 21, 2020

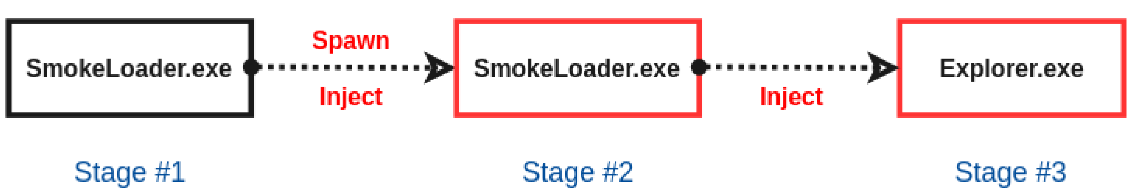
Abdallah Elshinbary

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+flNewProtect] ; 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.

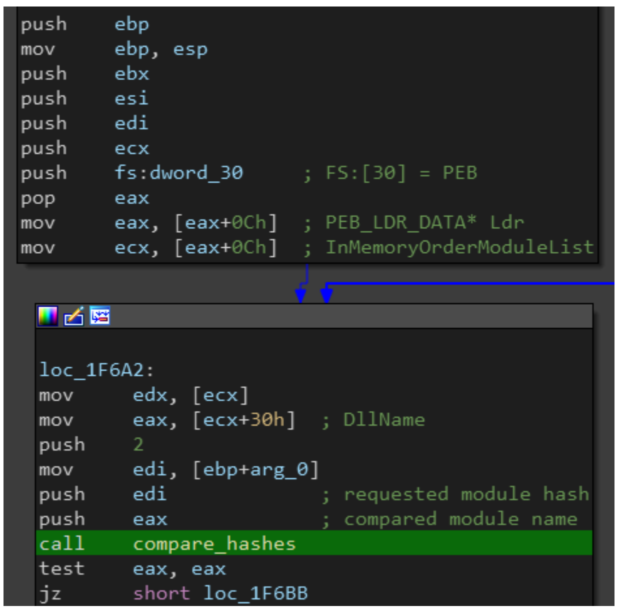
```
int calc_hash(char* name) {
    int x, hash = 0;
    for(int i=0; i<strlen(name); i++) {
        x = name[i] | 0x60;
        hash = 2 * (x + hash);
    }
    return hash;
}
```

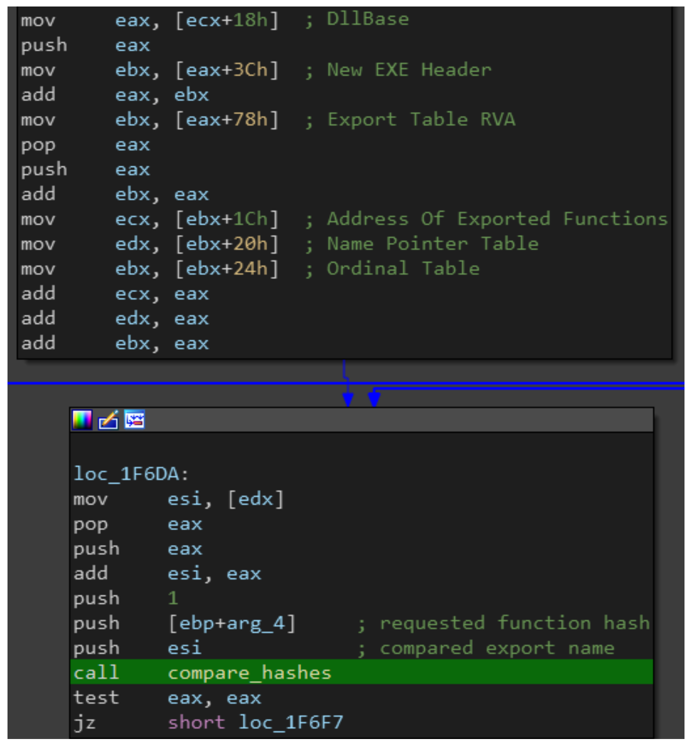
The shellcode uses `PEB traversal` technique for finding a function.

> 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 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:

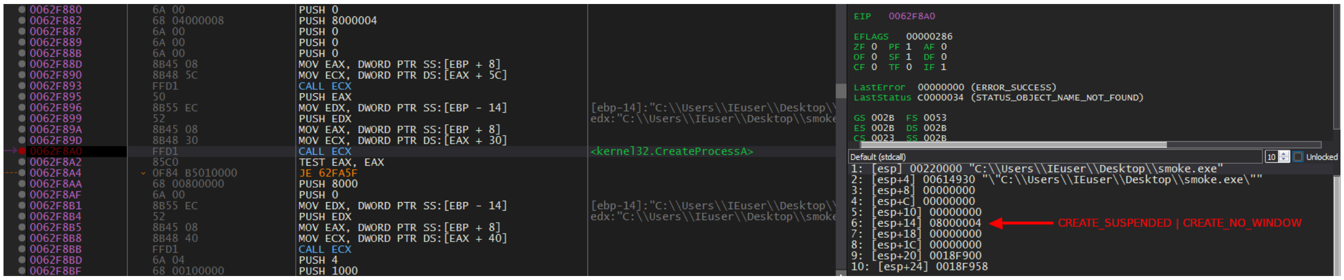
Expand to see more

- 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

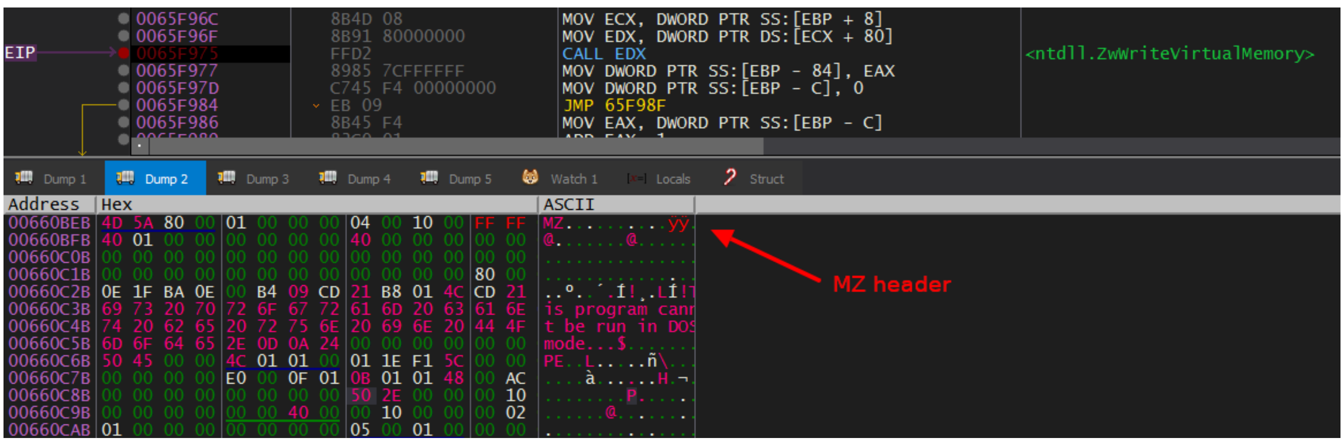
Process Hollowing

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



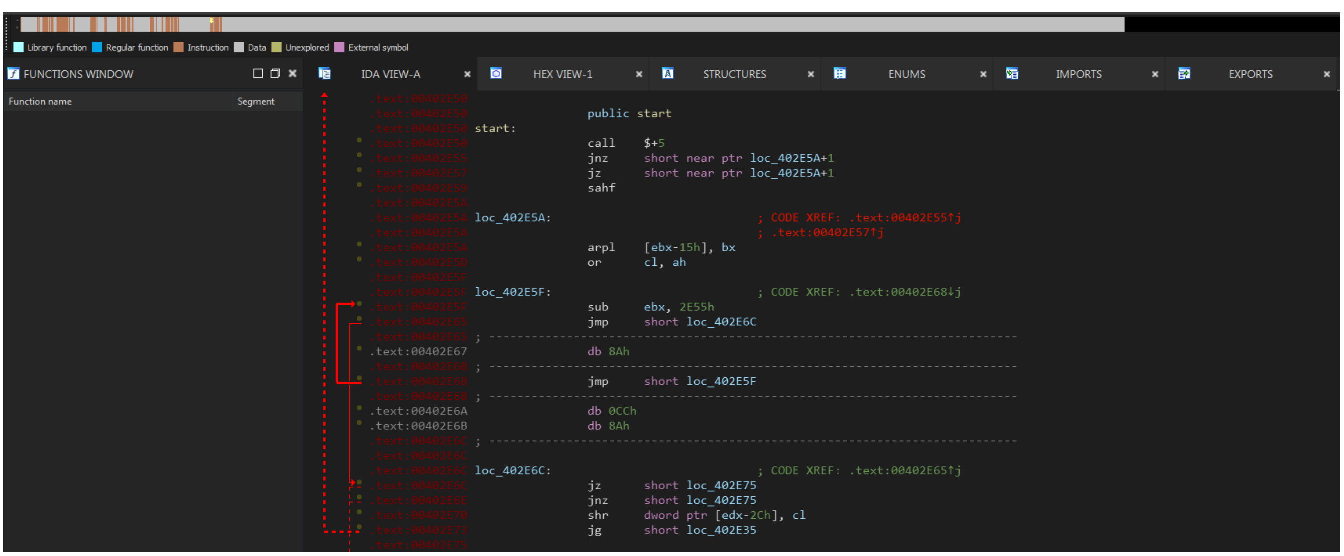
Next, it hollows out the memory at `0x400000` using `ZwUnmapViewOfSection()` and then allocates it again using `VirtualAllocEx()` with `RWX` permissions.

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.



Stage 2

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



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.

```
int start()
{
  int result; // eax

  result = __readfsdword(0x30u);                  // PEB
  if ( *(result + 0xA4) >= 6 )                    // PEB->OSMajorVersion
    result = 0x2DF2 * (*(result + 2) + 1) + 0x400000;// PEB->BeingDebugged
  return result;
}
```

What's interesting here is that these checks are used to calculate the return address. If the `OSMajroVersion` 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.

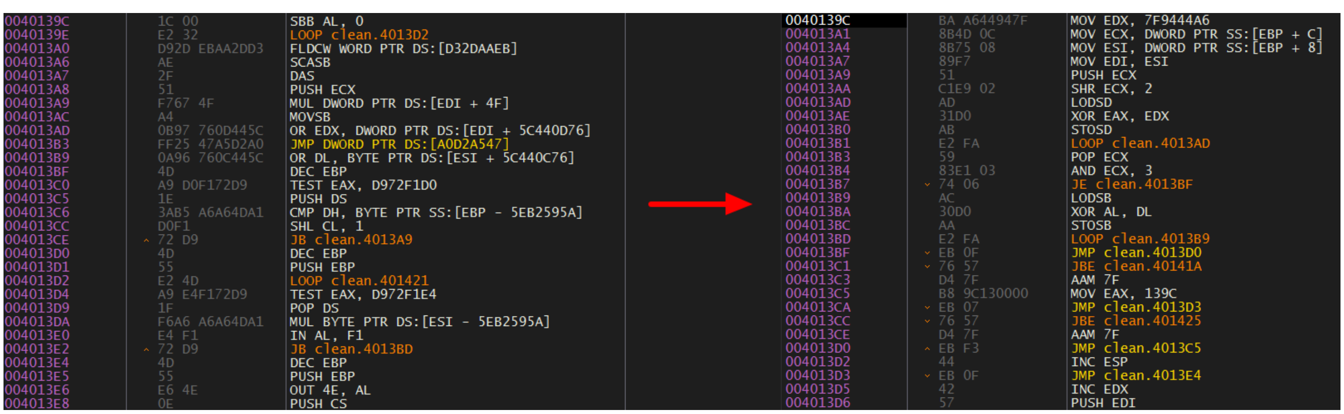
The function takes an offset and a size, it XORes the chunk at that offset with a single byte `(0xA6)`.

```
char __usercall xor_chunk@<al>(int offset@<eax>, int n@<ecx>)
{
  char *chunk; // esi
  _BYTE *xored; // edi
  char byte; // al
  char result; // al

  chunk = (offset + 0x400000);
  xored = (offset + 0x400000);
  do
  {
    byte = *chunk++;
    result = byte ^ 0xA6;
    *xored++ = result;
    --n;
  }
  while ( n );
  return result;
}
```

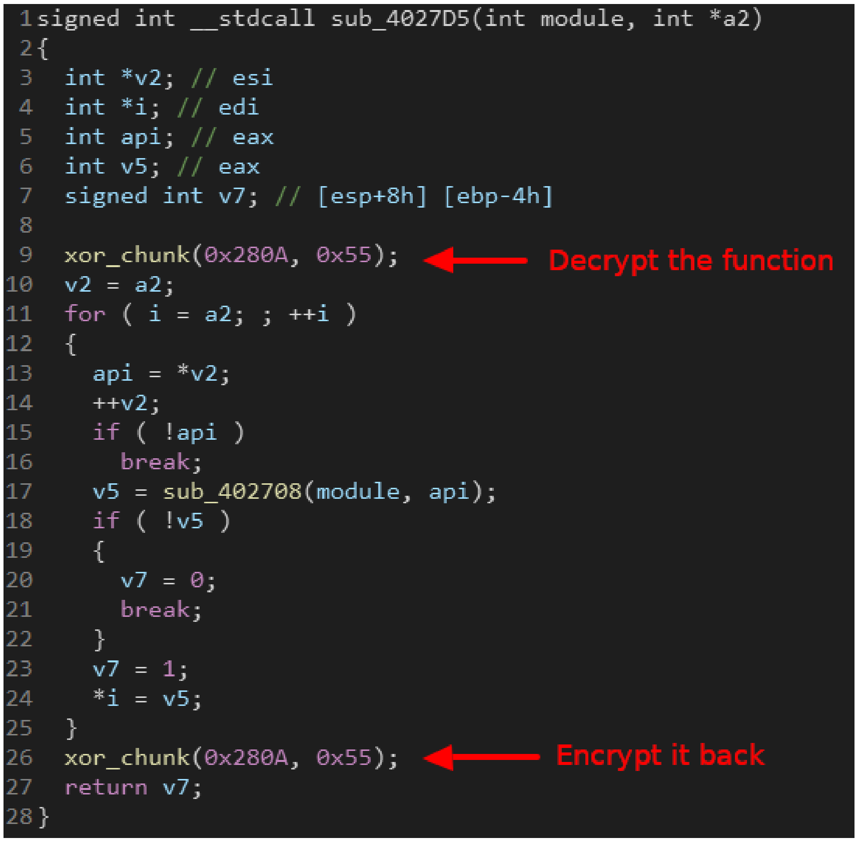
We can use `IDAPython` again to decrypt the encrypted chunks:

```
import idc
import idautils

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 idautils.CodeRefsTo(xor_chunk_addr, 0):
        mov_addr = list(idautils.CodeRefsTo(xref, 0))[0] - 5
        n = idc.get_operand_value(mov_addr, 1)
        offset = (xref + 5) - 0x400000
        xor_chunk(offset, n)
```

After the decryption:

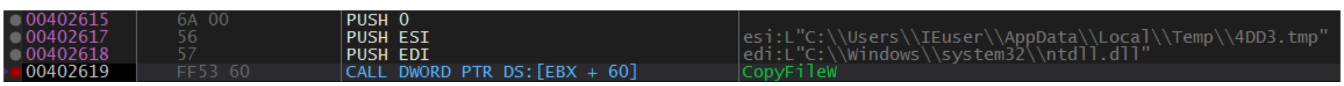


One thing to note here, SmokeLoader tries to keep as many encrypted code as possible. So once it’s done with the decrypted functions, it encrypts it again.



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.



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 (0x306cceb7)
        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 (0xeda647bb)
        LocalFree (0x742c61b2)
        MapViewOfFile (0x4db4c713)
        Sleep (0xd156a5be)
        WaitForSingleObject (0x8681d8fa)
        lstrcatW (0x2ab51a99)
        lstrcmpA (0x2abb9b4b)
    user32.dll
        EnumChildWindows (0x9a8897c9)
        EnumPropsA (0x8f0f57cf)
        GetForegroundWindow (0x5a6c9878)
        GetKeyboardLayoutList (0x04e9de30)
        GetShellWindow (0xd454e895)
        GetWindowThreadProcessId (0x576a5801)
        SendMessageA (0x41ecd315)
        SendNotifyMessageA (0xc6123bae)
        SetPropA (0x90bc10d3)
        wsprintfW (0x0bafd3f9)
    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

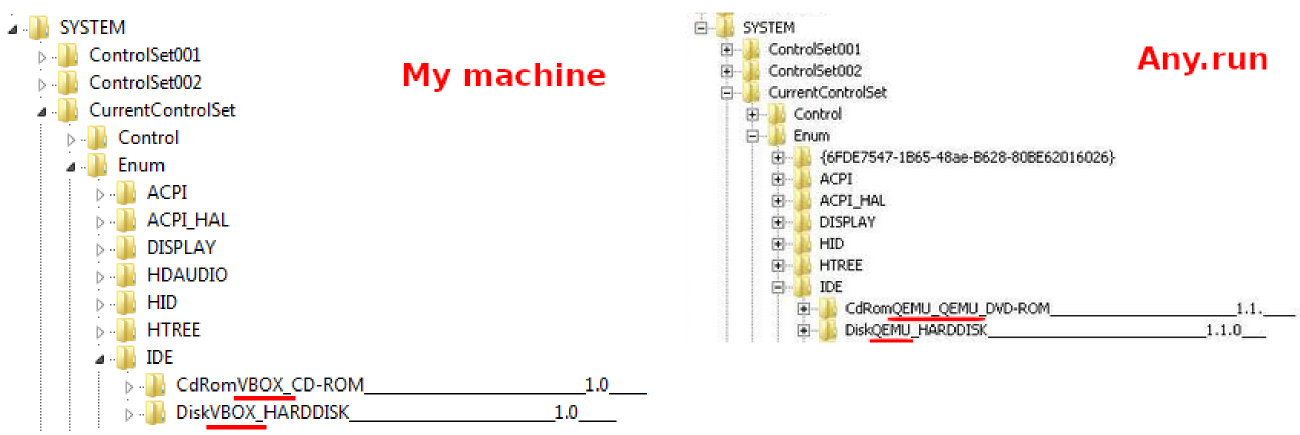
SmokeLoader enumerates all the subkeys of these keys:

- 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
```

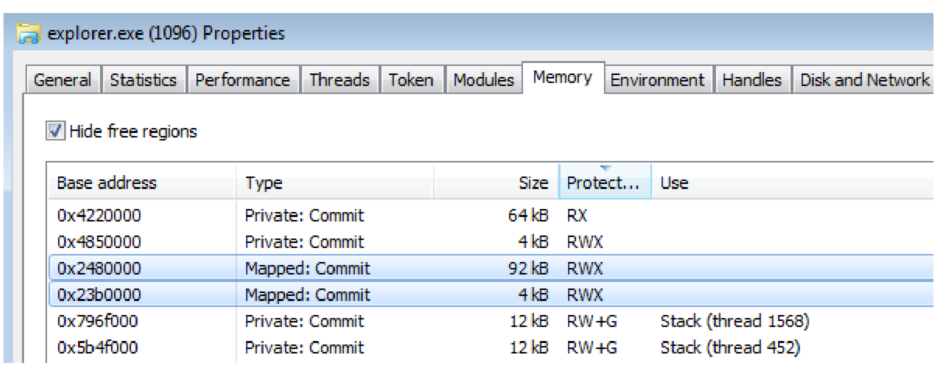
The injection process starts by creating two shared sections between the current process and explorer process (one section for the modified property and the other for the next stage's code), then SmokeLoader maps the created sections to the current process and explorer process memory space (so any changes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

```
push    edi
push    SEC_COMMIT
push    40h
push    eax
push    edi
push    SECTION_ALL_ACCESS
push    esi
call    dword ptr [ebx+0B4h]  ; ZwCreateSection
test    eax, eax
jnz     loc_401AEE

lea     eax, [ebp-38h]
mov     [eax], edi
lea     ecx, [ebp-3Ch]
push    dword ptr [ebp-8]
pop     dword ptr [ebp-3Ch]
push    PAGE_EXECUTE_READWRITE
push    edi
push    1
push    ecx
push    edi
push    edi
push    edi
push    eax
push    dword ptr [ebp-48h]   ; explorer handle
push    dword ptr [esi]       ; section handle
call    dword ptr [ebx+0B8h]  ; ZwMapViewOfSection
test    eax, eax
jnz     loc_401AEE
```

```
push    40h
push    edi
push    1
push    ecx
push    edi
push    edi
push    edi
push    eax
push    0FFFFFFFFh            ; current process handle
push    dword ptr [esi]       ; section handle
call    dword ptr [ebx+0B8h]  ; ZwMapViewOfSection
test    eax, eax
jnz     loc_401AEE
```

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



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:

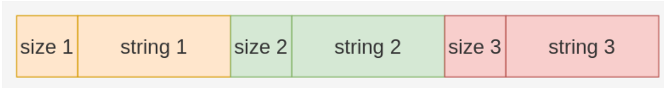
```
0xD384255C  →  Autoruns.exe
0x76BDCBAB  →  procexp.exe
0xA159E6BE  →  procexp64.exe
0x7E9CCCA5  →  procmon.exe
0xA24B8E63  →  procmon64.exe
0x63B3D1A4  →  Tcpview.exe
0xA28974F3  →  Wireshark.exe
0xA9B5F897  →  ProcessHacker.exe
0x6893EBAB  →  ollydbg.exe
0xF5FD94B7  →  x32dbg.exe
0xCBFD99B0  →  x64dbg.exe
0x8993DEE5  →  idaq.exe
0x8993D8CF  →  idaw.exe
0x8C083960  →  idaq64.exe
0xB6223960  →  idaw64.exe
```

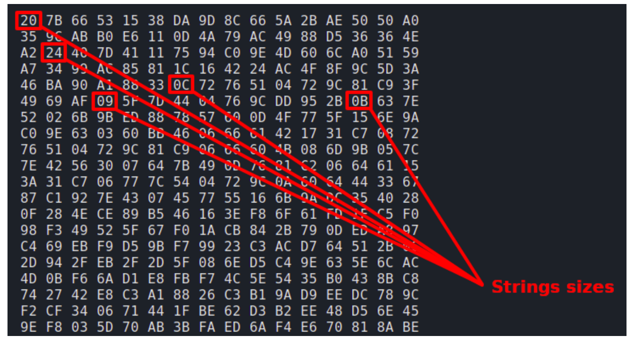
If one of these processes is found to be running, `explorer.exe` will exit.

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:





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

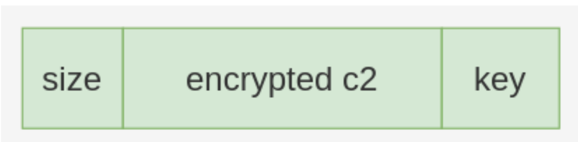
regsvr32 /s %s
regsvr32 /s /n /u /i:"%s" scrobj
%APPDATA%
%TEMP%
.exe
.dll
:Zone.Identifier
POST
Content-Type: application/x-www-form-urlencoded
runas
Host: %s
PT10M
1999-11-30T00:00:00
NvNgxUpdateCheckDaily_{%08X-%04X-%04X-%04X-%08X%04X}
Accept: */*
Referer: %S

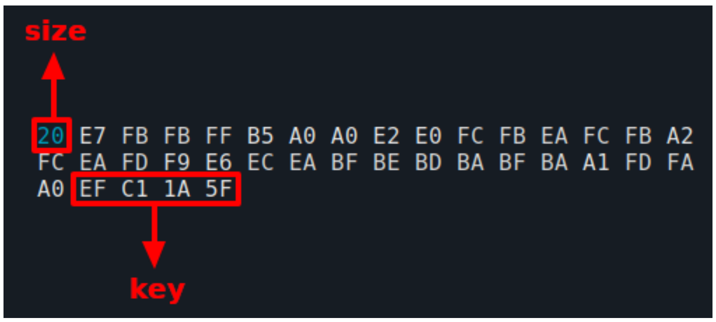
Encrypted C2 Domains

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:



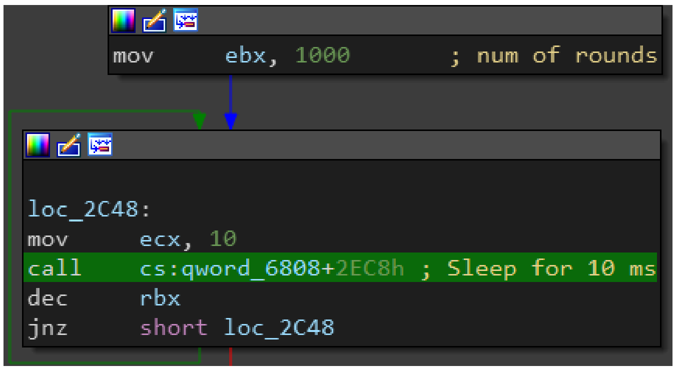


We can easily decrypt the domains:

```
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("E7FBFBFFB5A0A0E2E0FCFBEAFCFBA2FCEAFDF9E6ECEABFBEBDBABFBAA1FDFAA0", "EFC11A5F")
# http://mostest-service012505.ru/
```

C2 Communications

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



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.

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

SmokeLoader fc20b03299b8ae91e72e104ee4f18e40125b2b061f1509d1c5b3f9fac3104934

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://mirtest-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/