Smoke Loader – downloader with a smokescreen still alive

blog.malwarebytes.com/threat-analysis/2016/08/smoke-loader-downloader-with-a-smokescreen-still-alive/

Malwarebytes Labs August 5, 2016

This time we will have a look at another payload from recent RIG EK campaign. It is Smoke Loader (Dofoil), a bot created several years ago – one of its early versions was advertised on the black market in 2011. Although there were some periods of time in which it was not seen for quite a while, it doesn't seems to plan retirement. The currently captured sample appears to be updated in 2015.

This small application is used to download other malware. What makes the bot interesting are various tricks that it uses for deception and self protection.

We will walk through the used techniques and compare the current sample with the older one (from 2014).

Analyzed samples

Main focus of this analysis is the below sample, which is dropped by Rig EK:

The above sample downloads:
Payload:

> f60ba6b9d5285b834d844450b4db11fd – (it is an IRC bot, C&C: med-global-fox[DOT]com)

Updated Smoke Loader:

During the analysis it will be compared against the old sample, first seen in September 2014

Behavioral analysis

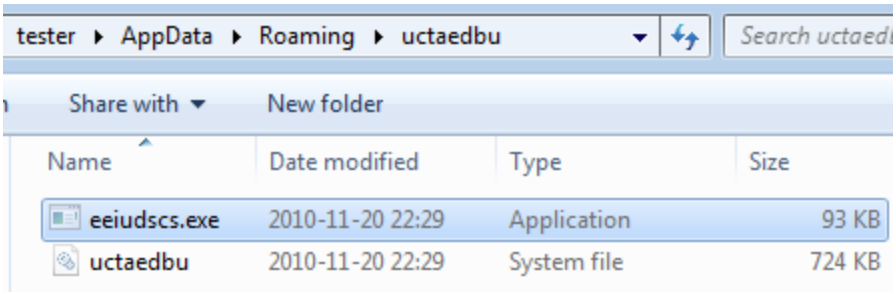
After being deployed, Smoke Loader inject itself into *explorer.exe* and deletes the original executable. We can see it making new connections from inside the *explorer* process.

Installation and updates

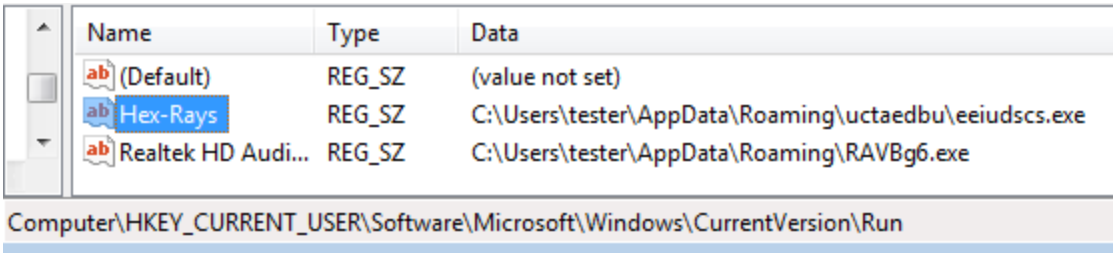
Smoke Loader not only installs its original sample but also replaces it with a fresh version, which is downloaded from the C&C – path: ***http://<CnC address>/system32.exe***. This trick makes detection more difficult – updated samples are repacked by a different crypter, may also have their set of C&Cs changed.

During the current analysis, the initial sample of Smoke Loader dropped the following one: bc305b3260557f2be7f92cbbf9f82975

Sample is saved in a hidden subfolder, located in %APPDATA%:



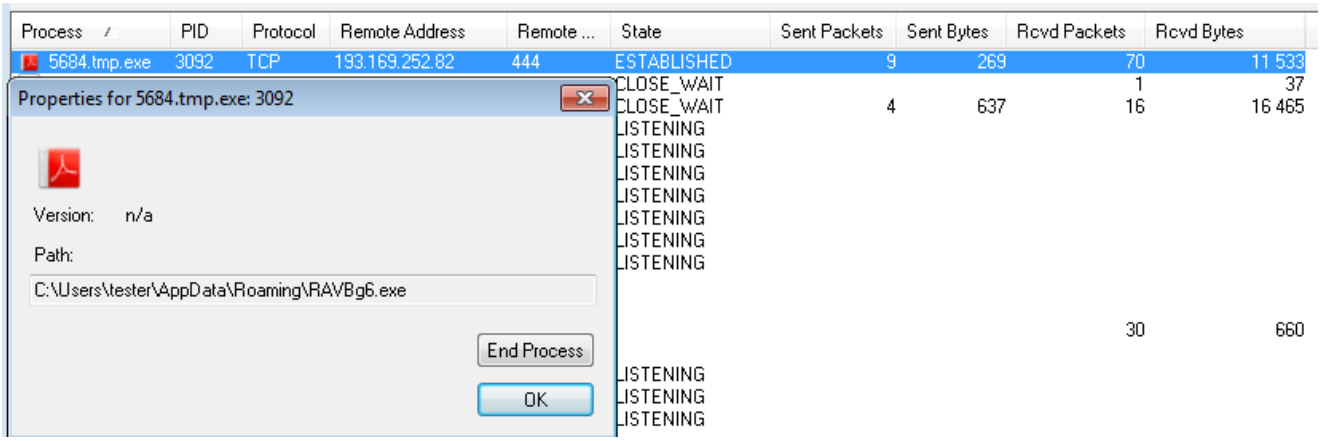
Smoke Loaded adds its current sample and all other downloaded executables to the Windows registry. Names of the keys are randomly chosen among the names of existing entries:



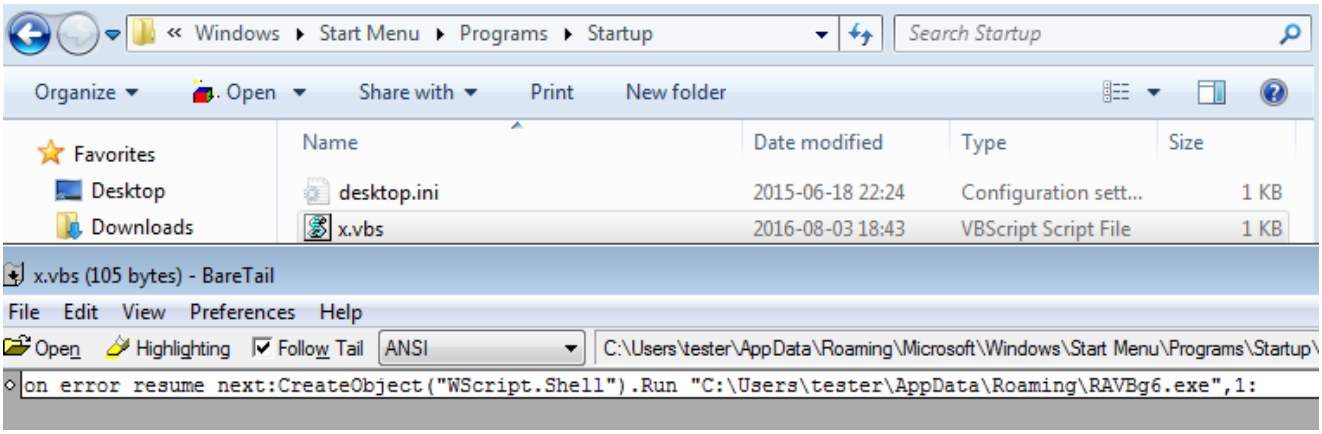
This persistence method is pretty simple (comparing i.e. with Kovter), however there are some countermeasures taken against detection of the main module. The timestamp of the dropped executable is changed, so that malware cannot be found by searching recently modified files. Access to the file is blocked – performing reading or writing operations on it is not possible.

Loading other executables

During its presence in the system it keeps downloading additional modules – “plugins”. First, the downloaded module is saved in %TEMP% under a random name and run. Then, it is moved to %APPDATA%. Below, we can see that the payload established connection with its own separate C&C:



There is also a script in Autostart for deploying the payload:



Network communication

To make analysis of the traffic harder, along with communicating with the C&C bot generates a lot of redundant traffic, sending requests to legitimate domains.

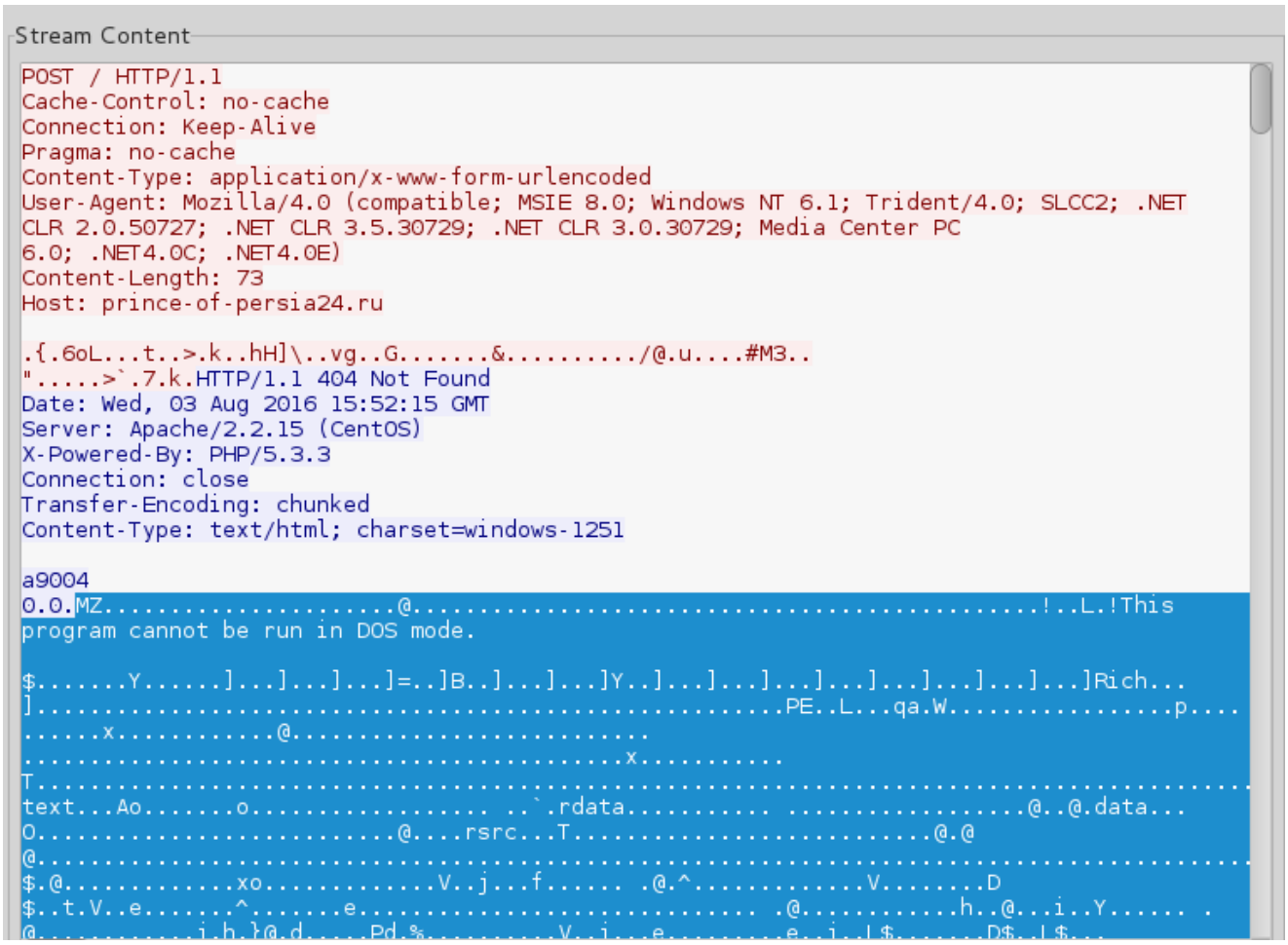
The current sample's C&C addresses:

- ***smoktruefalse.com***
- ***prince-of-persia24.ru***

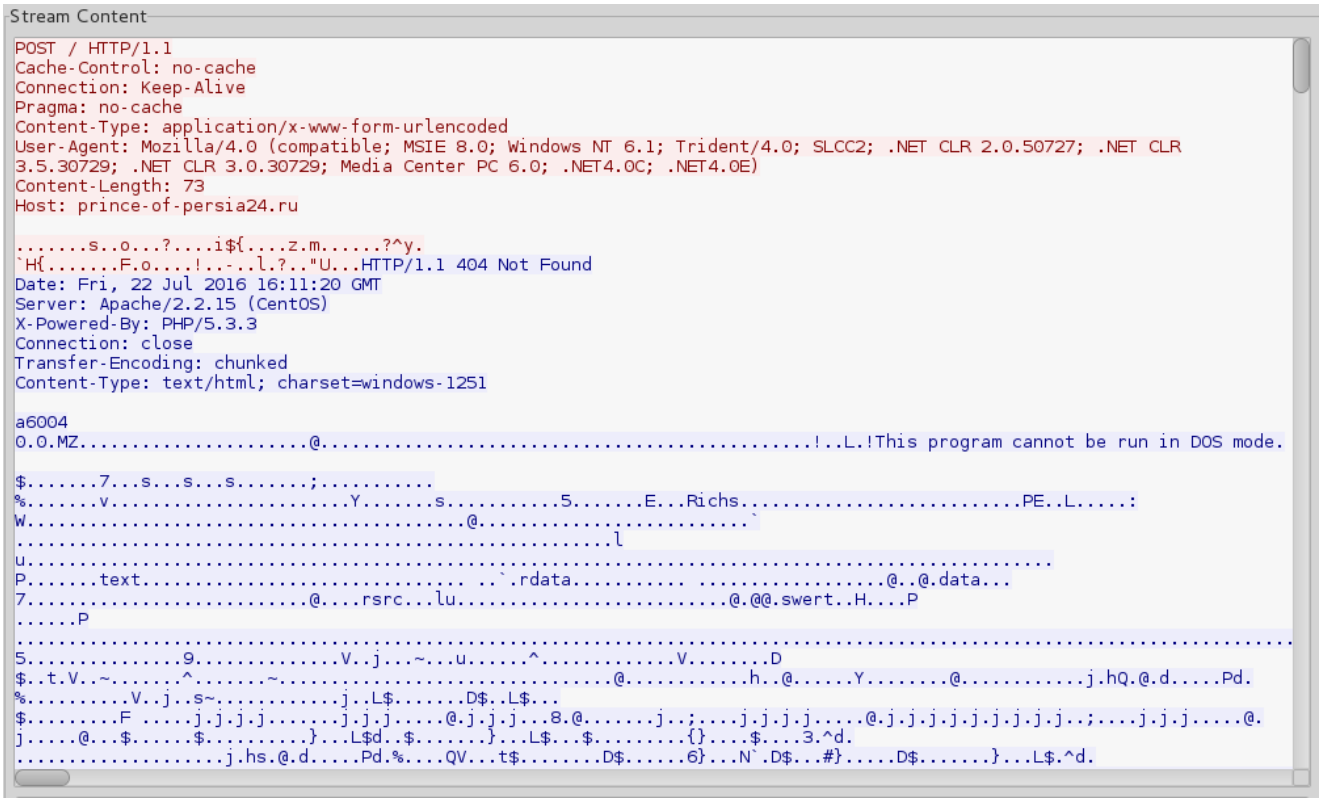
Traffic is partially encrypted.

In the examples below, we can see how the bot downloads from the C&C other executables.

1 – Updating the main bot with a new sample of Smoke Loader:

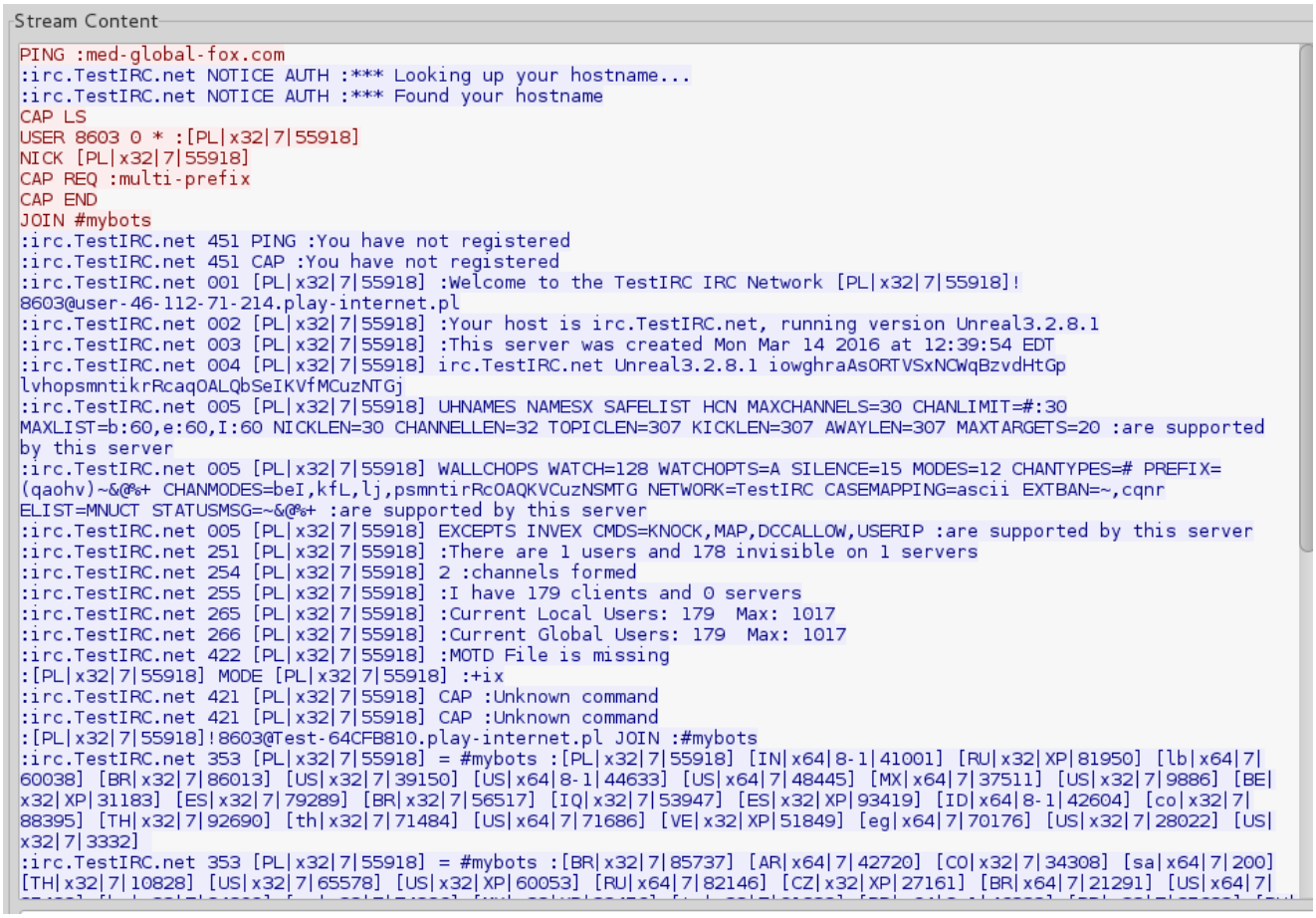


2 – Downloading the additional payload (“plugin”):



Payload traffic

Smoke Loader deploys the downloaded sample, so after some time we can see traffic generated by the payload (connecting to ***med-global-fox.com***). By its characteristics, we can conclude that this time the “plugin” is an IRC bot:



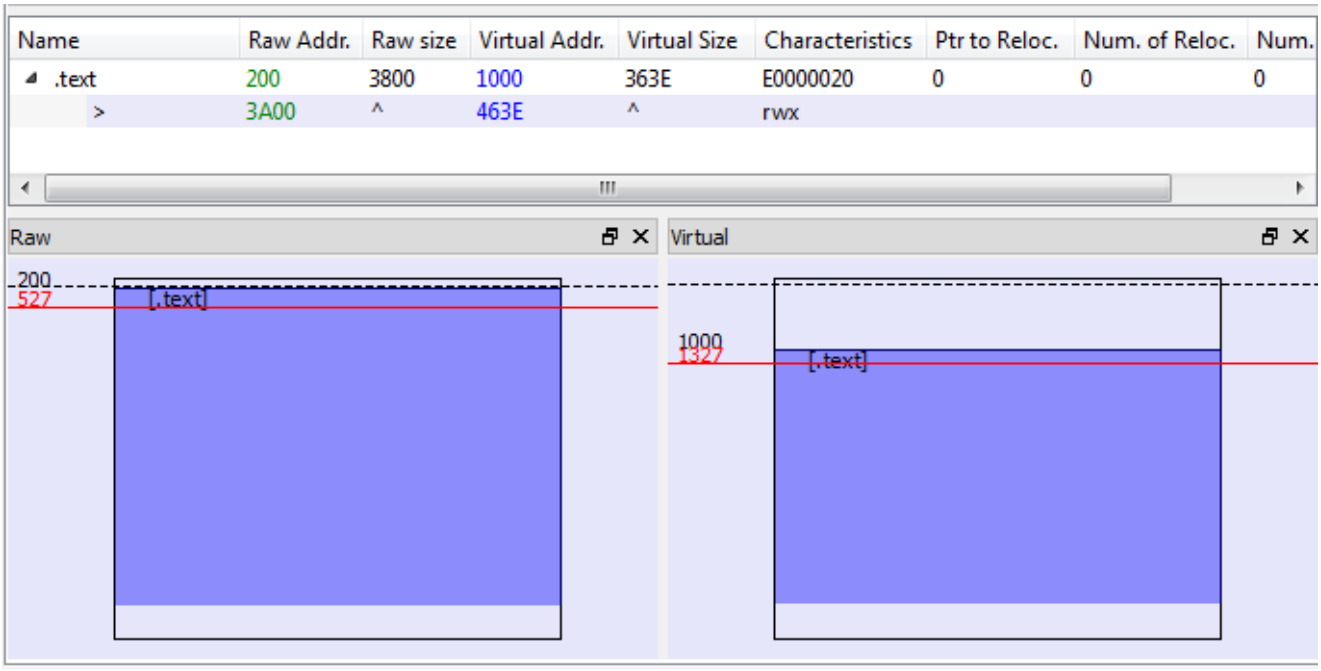
Inside

Like most of the malware, Smoke Loader is distributed packed by some crypter that provides the first layer of defense against detection.

After removing the crypter layer, we can see the main Smoke Loader executable. However, more unpacking needs to be done in order to reach the malicious core. For the sake of convenience, I will refer to the code section of the unpacked sample as ***Stage#1***. Its execution starts in the Entry Point of the main executable and its role is to provide additional obfuscation. It also serves as a loader for the most important piece: ***Stage#2*** – this is a DLL, unpacked to a dynamically allocated memory and run from there.

Stage#1

Interesting feature of this bot is that often its executables have one section only and no imports. Below you can see the visualization of sections layout (Entry Point is marked red):



Code at Entry Point is obfuscated and difficult to follow. It contains many redundant jumps, sometimes an address of a next jump is calculated on the fly – that's why tools for static analysis cannot resolve them. Also, to make analysis more difficult, the code modifies itself during execution.

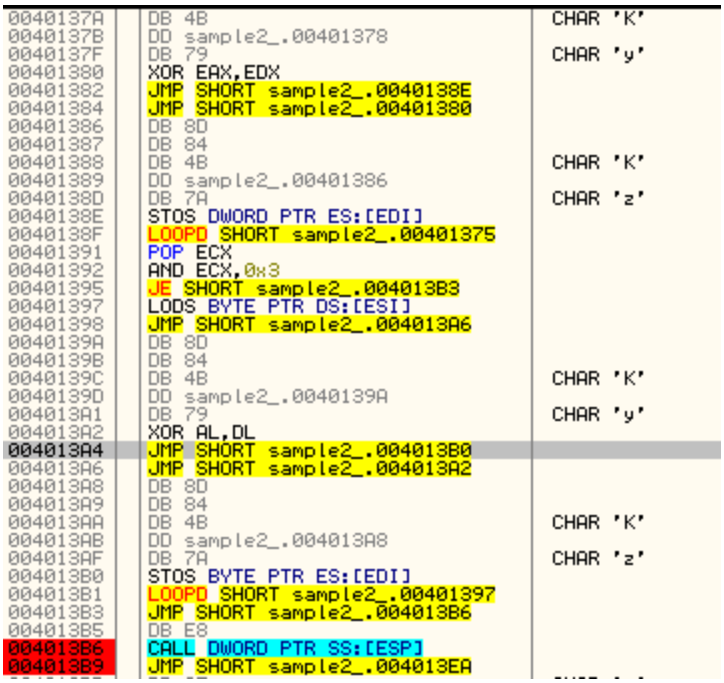
The initial routine decrypts selected parts of the code section using XOR with a hardcoded value:

```
0040133E
0040133E loc_40133E:
0040133E push    0CBDAC235h
00401343 mov     edx, 0CB9AD292h
00401348 xor     [esp+enc_code], edx
0040134B pop     esi
0040134C jmp     short loc_401355
```

And then it it calls it:

```
004013B6
004013B6 loc_4013B6:
004013B6 call    [esp+enc_code]
004013B9 jmp     short loc_4013EA
```

This is not the only way Smoke Loader modifies itself. In the unpacked part, we can see some more tricks. This code uses many tiny jumps followed by XOR and LODS instructions to modify and displace code after every few steps of execution. In between, junk instructions have been added to make it less readable:

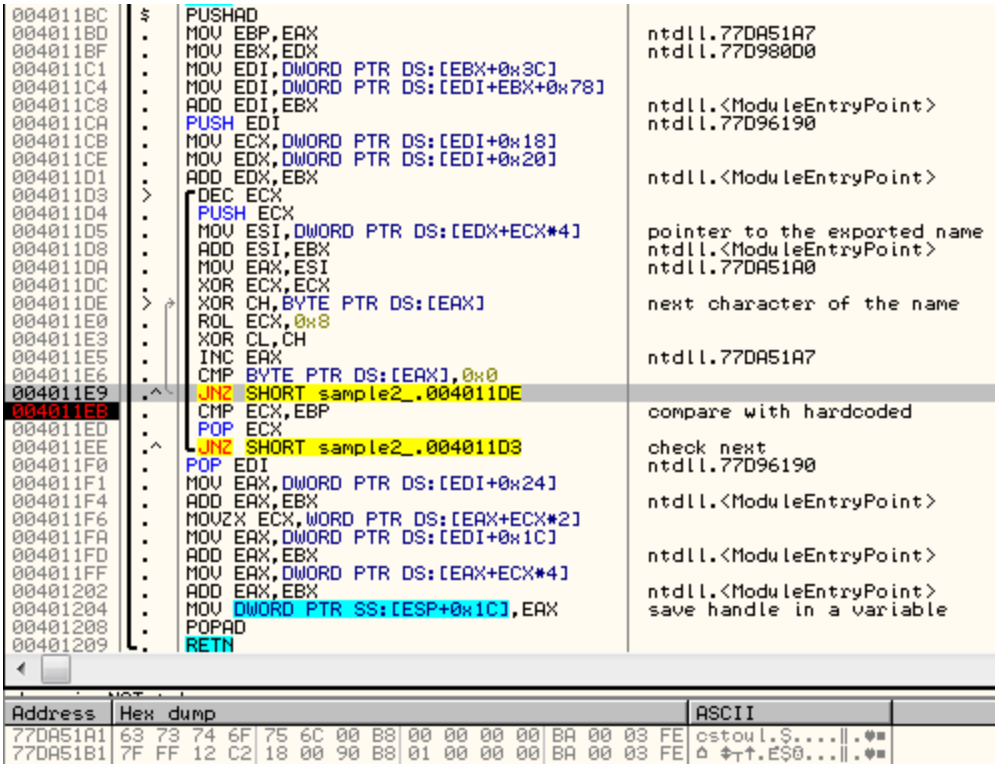


The bot loads all the necessary imports by its own. To achieve this goal, it deploys a variant of a popular method: searching function handles in the loaded modules by calculating checksum of their names and comparing them with hardcoded values. First, a handle to the loaded module is fetched with the help of Process Environment Block (PEB)*:

```
MOV ESI, FS:[30] ; copy to ESI handle to PEB
MOV ESI, DS:[ESI+0xC] ; struct _PEB_LDR_DATA *Ldr
MOV ESI, DS:[ESI+0x1C] ; ESI = Flink = Ldr->InLoadOrderModuleList
MOV EBP, DS:[ESI+0x8] ; EBP = Flink.DllBaseAddress
```

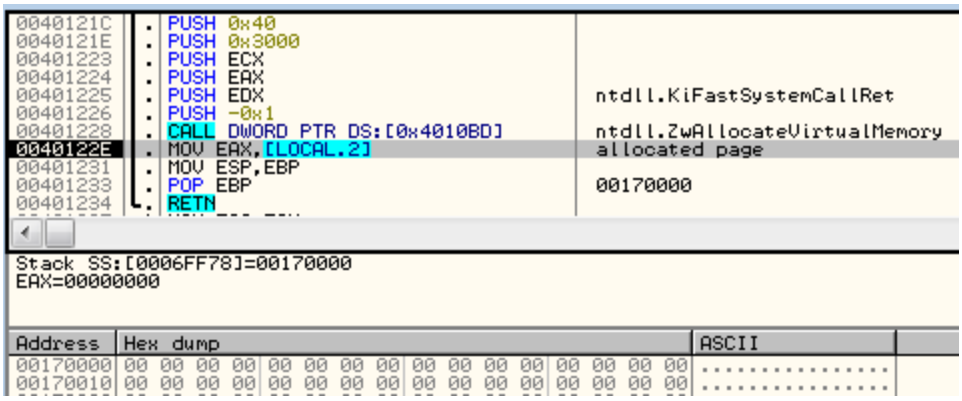
** read more about it here*

Below we can see the fragment of code that walks through exported functions of *ntdll.dll* searching for a handle to the function: *ZwAllocateVirtualMemory* (using it's checksum: 0x976055C), and then saving the found handle in a variable:



Thanks to this trick Smoke Loader can operate without having any import table. (The same method is utilized by *Stage#2* to fill its imports).

The stored handle is used to make an API call and allocate additional memory:

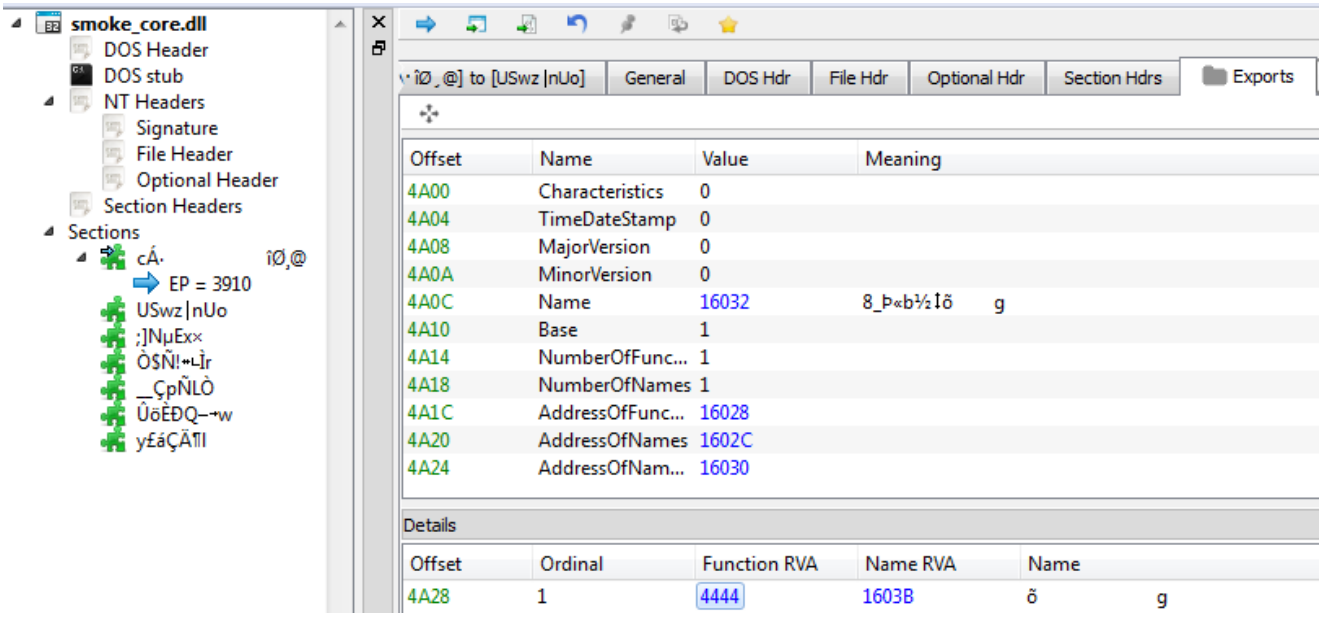


In this added memory space, *Stage#2* is being unpacked. This new module is a PE file with headers removed (it is a common anti-dumping technique). Below, you can see the part that was erased at the beginning of the file (marked red):

_00020000.mem

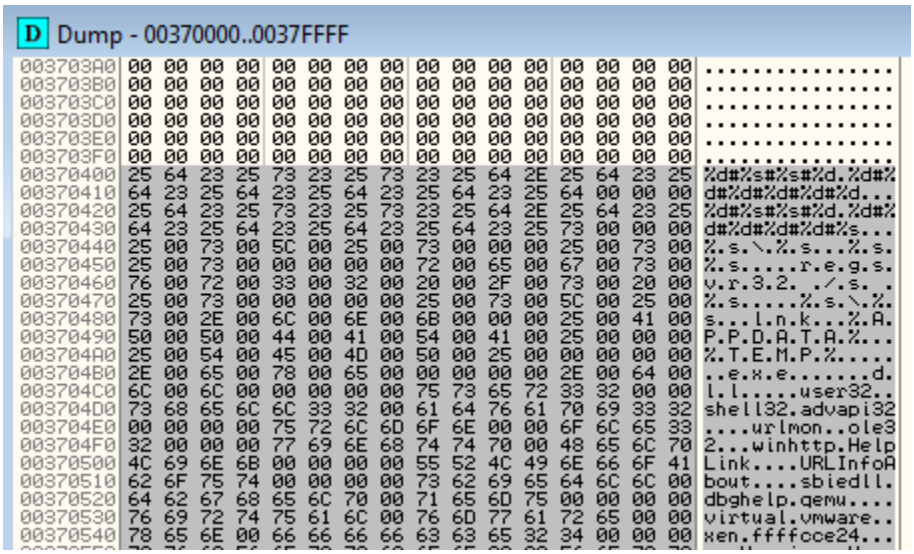
```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00  MZP.........ˇˇ..
00000010   B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00  ¸.......@.......
00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030   00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ................
00000040   BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90  ş....´.Í!¸.LÍ!..
00000050   54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73  This program mus
00000060   74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57  t be run under W
00000070   69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00  in32..$7........
00000080   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000100   50 45 00 00 4C 01 07 00 19 5E 42 2A 00 00 00 00  PE..L....^B*....
00000110   00 00 00 00 E0 00 8E A1 0B 01 02 19 00 36 00 00  ....ŕ.Ž˙.....6..
00000120   00 1A 00 00 00 00 00 00 10 45 00 00 00 10 00 00  .........E......
00000130   00 50 00 00 00 00 40 00 00 10 00 00 00 02 00 00  .P....@.........
00000140   04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00  ................
00000150   00 90 01 00 00 04 00 00 00 00 00 00 02 00 01 00  ................
00000160   00 00 00 00 00 00 00 00 00 00 00 10 00 00 10 00  ................
00000170   00 00 00 00 10 00 00 00 00 60 01 00 40 00 00 00  .........`..@...
```

If we add the missing part, we can parse it as a typical PE file. It turns out to be a DLL exporting one function. Exactly the same technique was used before by older versions of Dofoil. In the past, the name of the module was *Stub.dll* and the exported function was *Works*. Now the names are substituted by garbage.

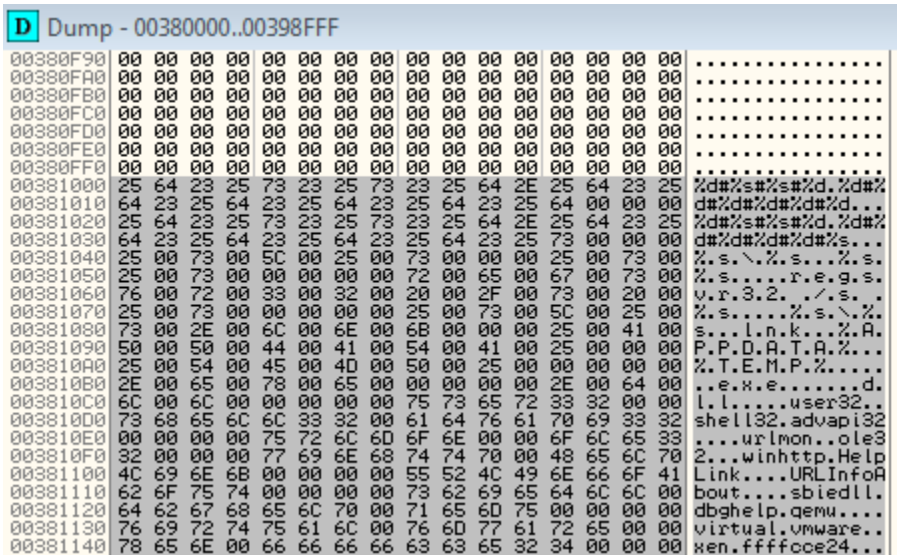


This piece is loaded by the dedicated function inside *Stage#1*, that takes care of all the actions typically performed by the Windows Loader.

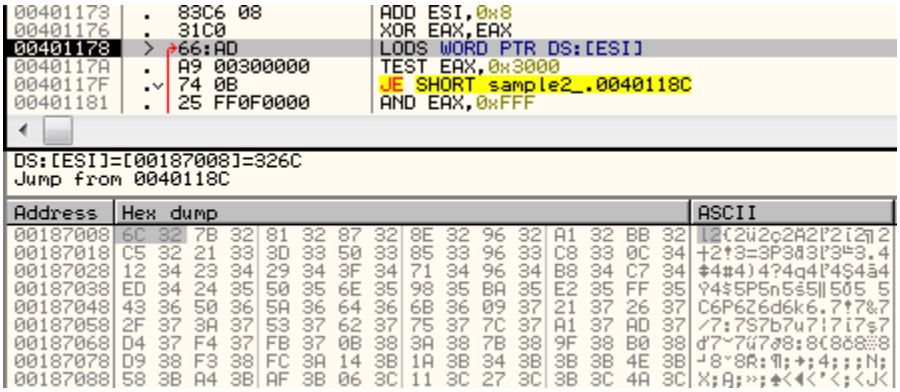
First the unpacked content is in raw format (Size of Headers: 0x400, File Alignment: 0x200):



Then, the same content is realigned to a virtual format (unit size: 0x1000):

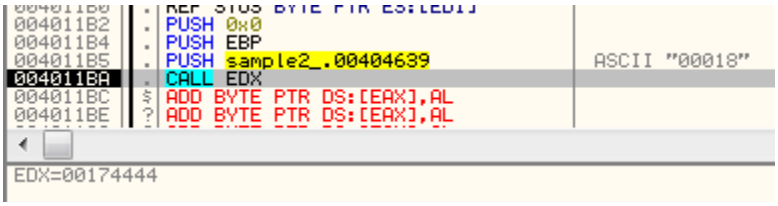


Another subroutine parses and applies relocations. As we can see below, it is a typical relocations table known from PE format. Entries are stored as a continuous array of WORDs:

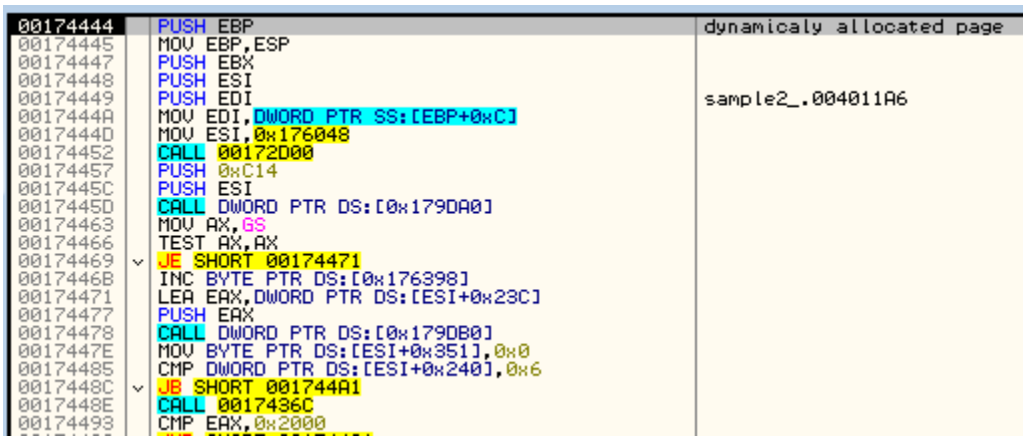


The loader processes them one by one. First, it checks if the entry type is “32-bit field” (by **TEST EAX,0x3000**) – it is the only format supported in this case. Then, it fetches the relocation offset (**AND EAX,0xFFF**), gets the pointed address and performs calculation – by removing old ImageBase (it’s value is hardcoded) and applying the new base – offset to the dynamically allocated memory where the unpacked code was copied).

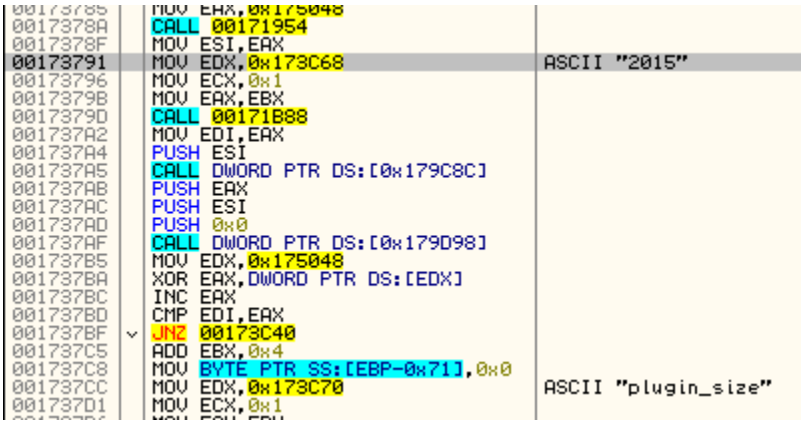
Finally execution flow can be redirected to the new code. *Stage#1* calls the exported function form the *Stage#2* DLL with three parameters.The first one is a string, different for each sample (this time it is “00018”):



The execution of *Stage#2* starts inside the dynamically allocated section:



At this stage we can see some of the strings known from previous editions of Smoke Loader. String “2015” may suggest that this version has been written in 2015 (however, compilation timestamp of the sample is more recent: 10-th June 2016).



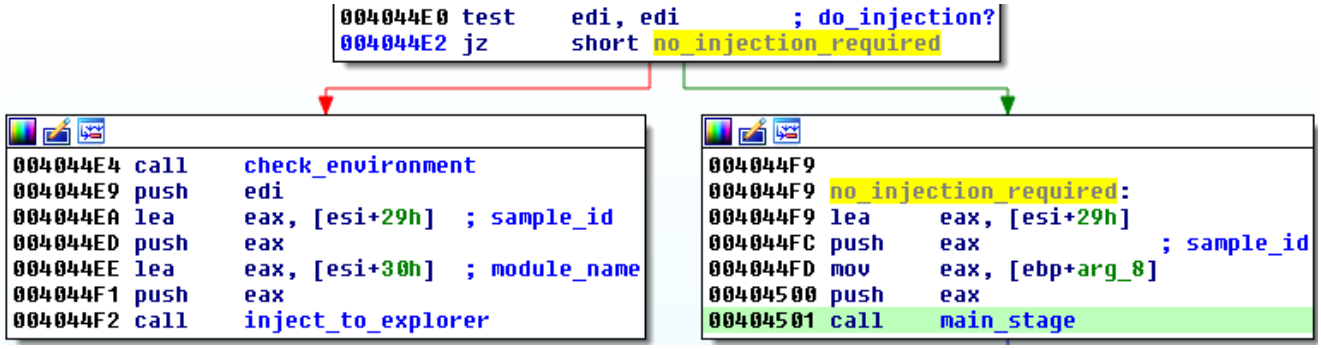
Stage#2

While the previous stage was just a preparation, at *Stage#2* the malicious functions are deployed. Its entry lies within the exported function that has the following header:

```
int __stdcall Work(char* sample_id, bool do_injection, char* file_path);
```

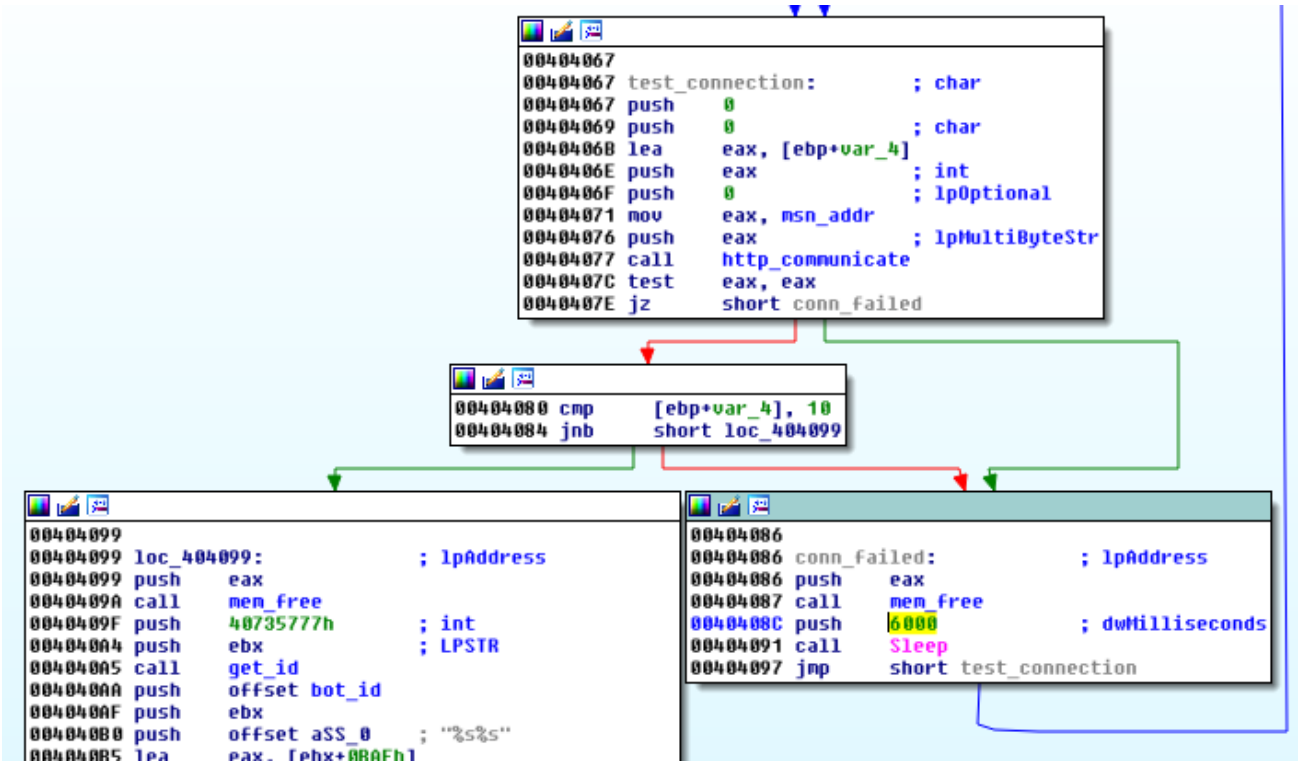
Basing on those parameters, the executable recognize its current state and the execution path to follow.

Before executing the real mission, the bot prepares a disguise – injecting its code into a legitimate process – *explorer.exe* (more about it will be explained later). Whether this path should be deployed or not, it is specified by the second parameter (denoted as *do_injection*).

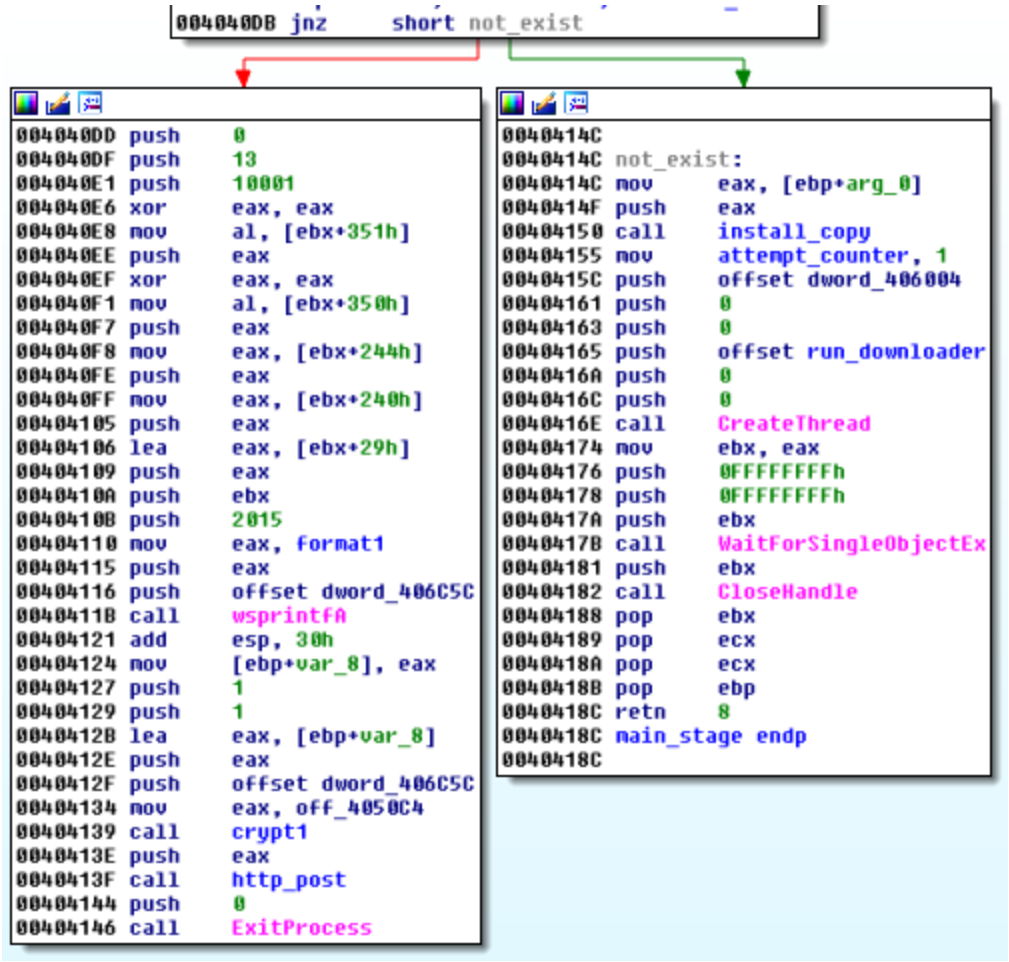


- If *Stage#2* was called with ***do_injection* flag set**, it will inject the code into ***explorer.exe***. Before doing so, the environment is checked for the presence of tools used for malware analysis. If any symptom is detected pointing that the sample is running in the controlled environment, application goes in the infinite sleep loop.
- If *Stage#2* was called with ***do_injection* flag cleared,** it starts proceeding to the *main path* of execution, that includes connecting to the C&C and downloading malicious modules.

If the main path of execution has been chosen, the bot proceeds to communicate with its C&C server. It is a known fact that before making the connection to the real C&C it first checks if the network is reachable. For the purpose of testing, it uses some non-malicious address – in this case it is ***msn.com***. As long as it gets no response, it keeps waiting and re-trying:



Once it found the connection working, next it verifies whether or not the application is already running (using the mutex with a name unique for the particular machine).



- If the mutex exist, program sends report to the C&C server and exits
- If the mutex does not exist (program is not yet running), it installs itself and then starts the main operations.

Injections to other processes

The older version was injecting the code alternatively to *explorer.exe* or *svchost.exe*. Injection to *explorer.exe* employed an interesting trick that triggered a lot of attention from researchers. It is based on a PowerLoader injection technique (*Shell_TrayWnd* / *NtQueueApcThread*).

Injection to *svchost.exe* was just a fail-safe, and followed more classic way similar to this one. Functions used:

```
CreateProcessInternalA
NtCreateSection
NtMapViewOfSection
RtlMoveMemory
NtUnmapViewOfSection
NtQueueApcThread
ResumeThread
```

The current version dropped that idea in favor for another method (similar to this one) – adding a new section to the remote process and copying its own code there. Functions used:

```
CreateProcessInternalA
NtQueryInformationProcess
ReadProcessMemory
NtCreateSection
NtMapViewOfSection
RtlMoveMemory
NtUnmapViewOfSection
ResumeThread
```

Now the only target of the injection is *explorer.exe*.

It patches Entry Point of *explorer* and adds there a code redirecting to the newly added section. That section contains the injected *Stage#2* DLL along with a small loader (similar to the one from *Stage#1*). Again, the loader prepares *Stage#2* and deploys it – this time with different parameters:

```
004010EB  .  REP STOS BYTE PTR ES:[EDI]
004010ED  .  LEA EDI,DWORD PTR DS:[EBX+0x121B]
004010F3  .  MOV ECX,DWORD PTR DS:[EDI+0x150]
004010F9  .  REP STOS BYTE PTR ES:[EDI]
004010FB  .  POPAD
004010FC  .  LEA EDI,DWORD PTR DS:[EBX+0xFFB]    sample_path
00401102  .  PUSH EDI
00401103  .  PUSH 0x0
00401105  .  LEA EDI,DWORD PTR DS:[EBX+0x120F]   ASCII "00018"
0040110B  .  PUSH EDI
0040110C  .  CALL EDX                            call Stage#2
0040110E  >  PUSH 0x64
00401110  .  CALL DWORD PTR DS:[EBX+0x18]
00401113  .^ JMP SHORT patched1.0040110E
00401115  .  RETN
00401116     MOV EAX,EAX
```

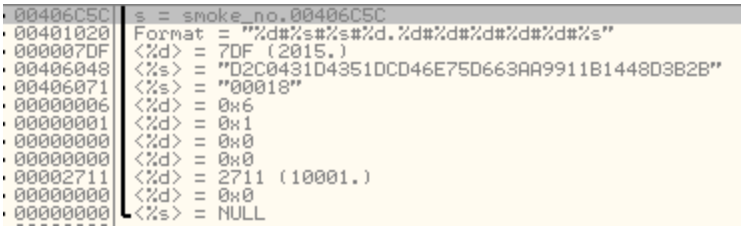
Communication protocol

Old versions of Smoke Loader were using a very descriptive protocol, with commands directly pointing to the functionality. Below are the parameters used by the old version:

```
cmd=getload&login=
&file=
&run=ok
&run=fail
&sel=
&ver=
&bits=
&doubles=1
&personal=ok
&removed=ok
&admin=
&hash=
```

In the current version, the sent beacon looks different – parameters are separated by a delimiter instead of following the typical, more lengthy key-value format:

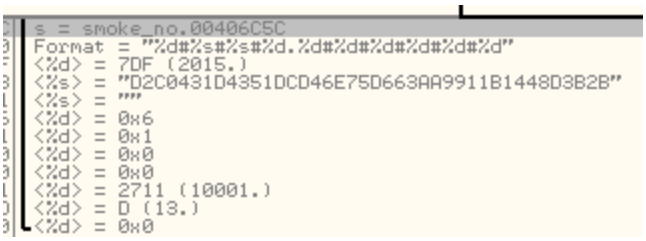
```
"2015#D2C0431D4351DCD46E75D663AA9911B1448D3B2B#00018#6.1#0#0#10001#0#"
```



Reading the beacon, we can confirm that the currently analyzed version is higher than the previous one. The bot also sends its ID, which is generated based on the GUID of particular system and the parameter typical for the particular sample (i.e. “00018”).

The program also reports to the C&C if there was attempt to run it more than once (mutex locked):

```
"2015#D2C0431D4351DCD46E75D663AA9911B1448D3B2B#00018#6.1#0#0#10001#13#0"
```



Conclusion

In the past Smoke Loader was extensively distributed via spam. Now we encountered it carried by an exploit kit.

Many parts of the bot didn't changed over the years, making this malware easy to identify. It still uses the same set of environment checks for its defense. Also, it waits for network accessibility in old style. The protocol used for its communication with the C&C is now less descriptive – it doesn't have so many keywords that identifies its performed actions. Like the previous, traffic is encrypted. The core features also stayed the same and the main role of this malware is to download and deploy other modules.

Appendix

http://stopmalvertising.com/rootkits/analysis-of-smoke-loader.html

https://blog.fortinet.com/2014/11/12/the-rebirth-of-dofoil

This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrzd.wordpress.com.