Analysis Report

Sample	172AED81C4FDE1CF23F1615ACEDFAD65
Author	Marion Marschalek [marion-m@live.at]
Date	24/03/2013

Abstract

The analyzed sample, detected as Backdoor.Win32.Banito, is multi-threaded malware that infects files on the disk, communicates to a remote server under the domain ns.dns3-domain.com and provides extended spying and system control functionalities. Its code is obfuscated and it implements a row of anti-analysis measures.

It was in-the-wild around late 2010 / early 2011 and its origins are believed to be Chinese.

Summary

The analyzed sample is a tricky piece of malware that replicates itself and communicates to a remote Command & Control server (C&C). It is a non-polymorphic file infector, replacing executable images in the file system with a copy of itself, but still hiding and starting the original applications when needed.

It implements various anti-analysis measures, as for example invoking intentional exceptions or checking for the dwFlags value of the GetStartupInfo() API call. It implements a huge amount of junk code and is highly obfuscated. It is implemented in object oriented C++ and makes heavy usage of virtual function calls that make analysis significantly harder than usual. Besides the sample constructs an API offset jump table on startup, which is used throughout execution for resolution of system calls.

The malware is designed as multi-threaded application that divides process control, file infection and sending and receiving messages to and from the C&C server in different threads to share execution load.

The communication to the remote server is handled via one single domain which is hard coded in the sample. The domain is ns.dns3-domain.com, the according network address is 125.34.39.47. The server is believed to be out of operation, as it does not answer to any message of the analyzed sample.

The malwares capabilities are extensive, it is able to spy on the system as well as control system operation. It can produce screenshots and screen captures, report file listings, rename, copy or delete files, command a system shutdown, execute files and numerous other operations.

Besides, functionality can be found to perform a self-update, most likely to a new version of the same malware. It can disinfect the system of its copies.

Contents

1. Overview

1.1 File Details

File Type	Portable Executable 32 / Microsoft Visual C++ 6.0
File Size	269.42 KB (275883 bytes)
MD5	172AED81C4FDE1CF23F1615ACEDFAD65
SHA-1	C47FAF863FD93A310408848F829090F4E783E74C
Detections	Backdoor.Win32.Banito (Kaspersky) TrojanDownloader.Win32.Unruy (Microsoft) Trojan.Artilyb (Symantec)

The analyzed sample is not packed or encrypted. It is highly obfuscated, all strings are built at runtime as well as most of the imports are resolved dynamically at runtime. Therefore, no informative strings can be extracted through initial, static analysis. The sample is written in C++ and is object oriented. This was found proofed by the use of virtual function tables and the extensive use of ecx for passing the this object pointer.

Automated, dynamic analysis as provided by Anubis Sandbox fails due to anti-simulation and/or anti-debugging measures.

Used tools for static and dynamic analysis are IDA Pro 6.1, CFF Explorer 1.0, Wireshark 1.4.1 and several applications from Sysinternals Toolsuite. The used analysis machine is a Windows XP SP3, running in VMware.

2. Anti-Analysis Measures

2.1 dwFlags in _STARTUPINFO Structure

Shortly after startup of the executable and before entering the WinMain function the malware performs a first intent to crash a present debugger. By calling to GetStartupInfoA the current _STARTUPINFO structure is retrieved, which contains a value called dwFlags. This value is 1 in case of a started GUI application. Anyway, in case of a debugger environment it is not 1, which causes the test instruction at 4345C0 to set the zero flag and lead execution to execute the out instruction shown in code block two.

```
.text:004345BA call    ds:GetStartupInfoA
.text:004345C0 test    byte ptr [ebp-30h], 1
.text:004345C4 jz      short loc_4345D7
```

```
.text:004345D7 loc_4345D7:
.text:004345D7 out     dx, al
.text:004345D8 stosb
```

The out-instruction is used for data transfer to I/O-ports, which are not directly accessible from user mode. An exception of type “Privileged Instruction” with the code c0000096 occurs, which is followed by the termination of the debugged process.

The solution in this case is to patch the executable at runtime, to achieve execution.

2.2 SEH for Obfuscation of the Execution Path

SEH is short for Structured Exception Handling and describes a structured way for Win32 applications to handle exceptions that occur at runtime. And, what is more, SEH is a way for the programmer to define custom handlers and link them into a chain of structured exception handlers, all of which are executed when the application encounters an exception. At registration, a new exception handler is linked on top of the handler chain, so it will be the first handler to be executed in case of an exception. Important to mention is, that the handler who accepts to handle an exception gets to decide where execution will resume after the handler was executed. It can point execution basically to any executable code in memory.

A handler chain is always present per thread and an according pointer is to be found in the thread information block (TIB) at offset 0. As the FS register always points to the TIB, the handler chain can be accessed via FS:0.

Information about the exception handler to execute is stored in a structure called __ehfuncinfo (compare source [2]), which in turn is provided as an argument for the SEH frame handler. This structure is also called exception record. Amongst other entries __ehfuncinfo contains a pointer named pTryBlockMap, which maps the try/catch blocks in an application. Also it leads to the array of handlers which are invoked, when the according exception handler is executed.

So essentially, when an exception occurs the steps to take are:

1. Find the registration of the custom exception handler, if there is any. It can easily be identified, if modification of FS:0 can be spotted. Another way would be to check the SEH pointer in TIB or to spot the offset of the handler function on the stack.
2. Determine the offset of the frame handler in the code, which is pushed on the stack before the modification.
3. Find the offset of the __ehfuncinfo structure, which is an argument for the frame handler; in case of MS Visual Studio compiled executables as argument in eax register to __CxxFrameHandler.
4. Follow the pTryBlockMap pointer, which is the fourth entry (excluding the magic number) to find the TryBlockMapEntries, which contain a pointer to the handler array at the fifth position.
5. Following mentioned pointer, the offset of each handler function can be determined.
6. Determine the offset, where execution will continue after the handler callback.

For more information on reversing of Win32 SEH see source [2]. Luckily, in the case of the given sample only two intentional exceptions were thrown and the structures of the exception records were considerably simple.

2.2.1 Exception in WinMain

The first exception is invoked by accessing unreadable memory as documented in the following listing:

```
.text:00401D98 mov     ecx, 69805h
.text:00401D9D call    ecx
```

The called address does not contain executable instructions, the call fails with an exception of type “Access Violation”. Considering the malicious nature of the analyzed sample it is at hand to search for the according exception handler registration at the beginning of the main function.

```
.text:00401C85 push    offset _WinMain@16_SEH
.text:00401C8A mov     eax, large fs:0
.text:00401C90 push    eax
.text:00401C91 mov     large fs:0, esp
```

The offset, which is pushed onto the stack before the registration process, is the last registered frame handler. It is actually the one to be invoked first, when the mentioned exception occurs.

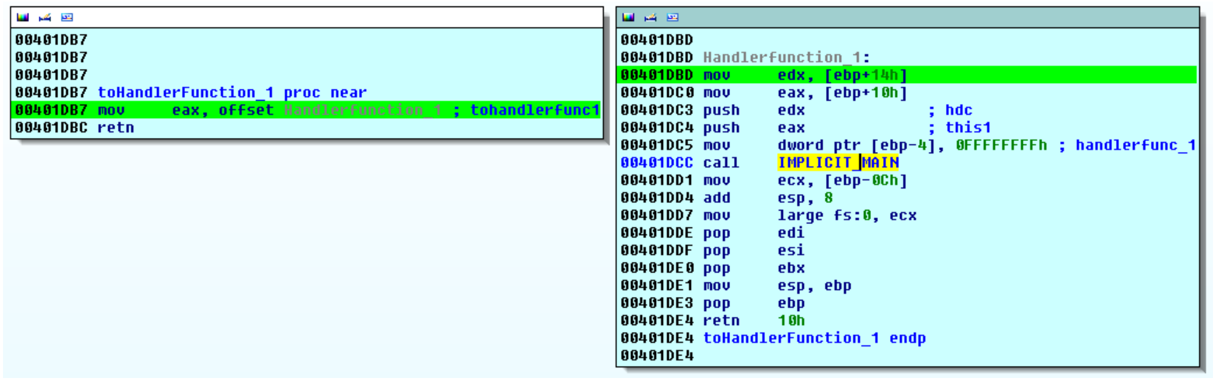
```
.text:00435080 _WinMain@16_SEH proc near
.text:00435080 mov     eax, offset stru_43AC90
.text:00435085 jmp     __CxxFrameHandler
.text:00435085 _WinMain@16_SEH endp
```

The offset of the exception record will be passed to the frame handler via eax. So what is left to do is to follow the exception record offset and find the address of the handler function. This is achieved by walking down the structure as shown in the following listing: Trymap -> HandlerArray -> offset toHandlerFunction_1.

```
.rdata:0043AC90 stru_43AC90 dd 19930520h                    ; Magic
.rdata:0043AC90 dd 2                             ; Count
.rdata:0043AC90 dd offset stru_43AC90.Info       ; unwindmap
```

```
.rdata:0043AC90 dd 1                                        ; trycount
.rdata:0043AC90 dd offset stru_43ACC0                       ; Trymap
.rdata:0043AC90 dd 3 dup(0)                                 ; _unk
.rdata:0043AC90 dd -1                                       ; Info.Id
.rdata:0043AC90 dd 0                                        ; Info.Proc
.rdata:0043AC90 dd -1                                       ; Info.Id
.rdata:0043AC90 dd 0                                        ; Info.Proc
.rdata:0043ACC0 stru_43ACC0 dd 0, 0, 1                                  ; _unk
.rdata:0043ACC0                                             ; DATA XREF:
.rdata:stru_43AC90
.rdata:0043ACC0 dd 1                                        ; Count
.rdata:0043ACC0 dd offset stru_43ACD8                       ; HandlerArray
.rdata:0043ACD4 dd 0
.rdata:0043ACD8 stru_43ACD8 _msRttiDscr <0, 0, 0, offset toHandlerFunction_1>
```

Actually, in this case the only operation the mentioned toHandlerFunction_1 performs is, to set the point of resumption after handling the exception. Execution is defined to continue at address 401DBD, which is a code stub that invokes a function which will be titled IMPLICIT_MAIN throughout the analysis, as it controls startup and eventually shutdown of the malware functionality.



```
00401DB7
00401DB7
00401DB7
00401DB7 toHandlerFunction_1 proc near
00401DB7 mov     eax, offset HandlerFunction_1 ; tohandlerfunc1
00401DBC retn
```

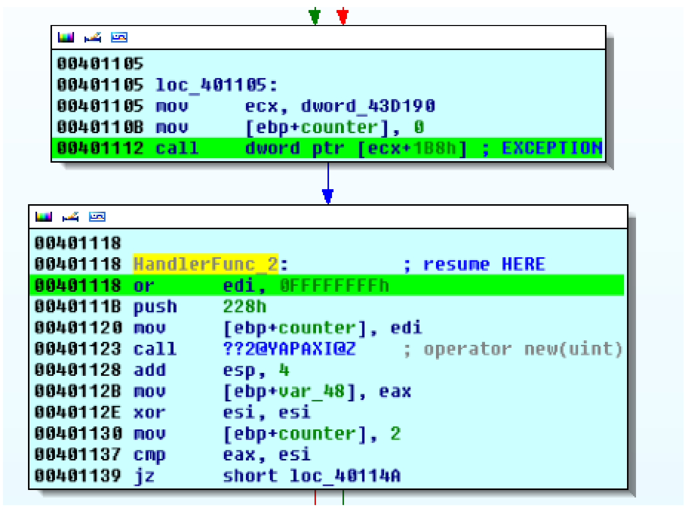
```
00401DBD
00401DBD Handlerfunction_1:
00401DBD mov     edx, [ebp+14h]
00401DC0 mov     eax, [ebp+10h]
00401DC3 push    edx                  ; hdc
00401DC4 push    eax                  ; this1
00401DC5 mov     dword ptr [ebp-4], 0FFFFFFFFh ; handlerfunc_1
00401DCC call    IMPLICIT_MAIN
00401DD1 mov     ecx, [ebp-0Ch]
00401DD4 add     esp, 8
00401DD7 mov     large fs:0, ecx
00401DDE pop     edi
00401DDF pop     esi
00401DE0 pop     ebx
00401DE1 mov     esp, ebp
00401DE3 pop     ebp
00401DE4 retn    10h
00401DE4 toHandlerFunction_1 endp
00401DE4
```

Illustration 1 – Exception Handler Function

2.2.2 Exception in IMPLICIT_MAIN

The second intentional exception is invoked shortly after the beginning of the IMPLICIT_MAIN function, and actually resumes execution exactly after the instruction that caused the exception. The exception itself actually occurs, because a call is carried out on empty memory, which again causes an access violation.

The exception record looks nearly as simple as with the first exception, the offset to the handler function can be found the same way as described above. The handler, identically to exception number one, only sets the offset to resume execution after the exception. And this offset, as mentioned before, points right to the instruction that follows the faulty one which caused the exception.



```
00401105
00401105 loc_401105:
00401105 mov     ecx, dword_43D190
0040110B mov     [ebp+counter], 0
00401112 call    dword ptr [ecx+1B8h] ; EXCEPTION
```

```
00401118
00401118 HandlerFunc_2:              ; resume HERE
00401118 or      edi, 0FFFFFFFFh
0040111B push    228h
00401120 mov     [ebp+counter], edi
00401123 call    ??2@YAPAXI@Z     ; operator new(uint)
00401128 add     esp, 4
0040112B mov     [ebp+var_48], eax
0040112E xor     esi, esi
00401130 mov     [ebp+counter], 2
00401137 cmp     eax, esi
00401139 jz      short loc_40114A
```

Illustration 2 – Exception Handler Function 2

2.3 Simulation Check with GetHostByName

At the beginning of IMPLICIT_MAIN the malware invokes a function which intends to resolve the hostname "…". The function gethostbyname returns always null for this request, obviously the host … can't be resolved to a valid address.

Taking a closer look at what happens next it becomes clear that this call to gethostbyname, if successful, would lead execution directly to the end of IMPLICIT_MAIN function. This is equal to the end of execution, as after IMPLICIT_MAIN there is nothing more than program termination. The conclusion lies at hand, that this name resolution is a check for an automated simulation environment, which would eventually return a standard address to any name resolution.

2.4 Obfuscation and Confusion Tricks

2.4.1 Junk Code

The analyzed sample contains a fair amount of junk code. The term junk code refers to executable instructions that either are executed but have no effect on the behavior of the application, or are never executed at all. In case of the analyzed sample most of the identified junk code is never executed. The following listing is meant to explain one case of obfuscation by use of junk code (some irrelevant instructions are omitted to shorten the listing).

```
.text:0040F2E3 mov     [esp+7Ch+var_78], ecx
.text:0040F2ED lea     eax, [esp+7Ch+var_78]
.text:0040F2F1 lea     ecx, [esp+7Ch+var_78]
.text:0040F2F5 imul    eax, ecx
.text:0040F2F8 lea     edx, [esp+7Ch+var_78]
.text:0040F2FC lea     ecx, [esp+7Ch+var_78]
.text:0040F300 push    ebx
.text:0040F301 sub     edx, ecx
.text:0040F303 push    ebp
.text:0040F304 push    esi
```

```
.text:0040F305 cmp      edx, eax
.text:0040F307 push     edi
.text:0040F312 jnz      short loc_40F35A
```

The cmp instruction at 40f305 will never set the zero flag because the registers cannot contain the same value. The ecx register as the this pointer will most likely not be zero at the time Var_78 is initialized, hence multiplication and subtraction operation will produce different results. The jnz instruction (jump if not zero) will always be taken.

This kind of obfuscation is used intensely, especially in the initialization phase of the malware. A lot of jump instructions, even if there is no useful branch to be taken, let the code appear non-linear and functions with few instructions look scarily huge. A graph mode as IDA Pro includes offers great help in understanding disassembly that's bloated with junk code.

The following screenshots show some of the functions that include code parts that are never being executed. The yellow areas mark executed code, the white areas are useless instructions.

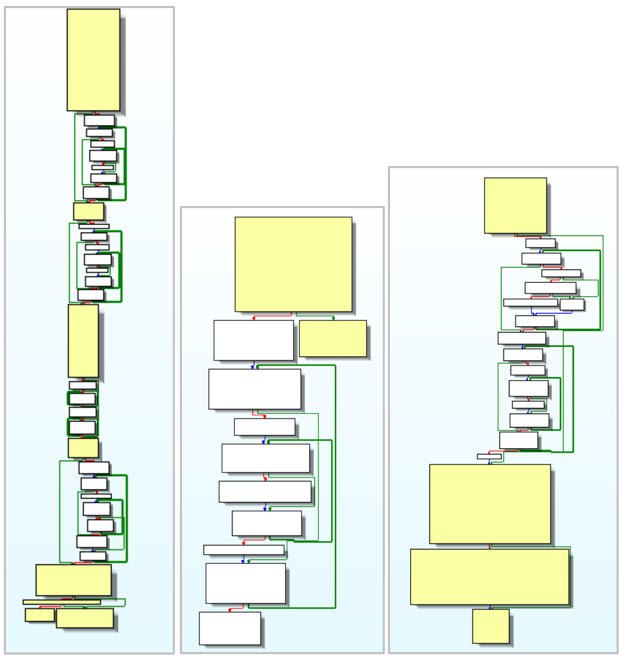
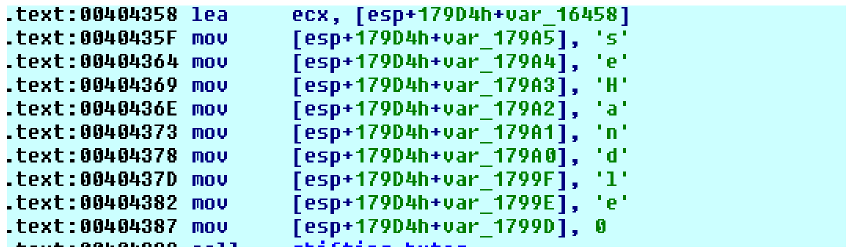


Illustration 3 – Junk Code in Simple Methods

2.4.2 String Construction at Runtime

To additionally complicate static analysis the sample at hand does not include strings, which could be easily found with the most basic tools. Any string that is used at runtime is built character by character and written to memory for later or immediate use.

The following screenshot shows how a part of the string CloseHandle is written to memory. For the resolution of API addresses all the names of the requested APIs are constructed similarly and written to the heap (see next section). Also names of events, temporary files, the domain of the command and control server or other keywords that are used at runtime are generated this way. This reveals two facts. First, no string is to be found via string scanning of the file; and second, all the strings are contained hardcoded in the executable, they just need proper extraction.



```
.text:00404358 lea     ecx, [esp+179D4h+var_16458]
.text:0040435F mov     [esp+179D4h+var_179A5], 's'
.text:00404364 mov     [esp+179D4h+var_179A4], 'e'
.text:00404369 mov     [esp+179D4h+var_179A3], 'H'
.text:0040436E mov     [esp+179D4h+var_179A2], 'a'
.text:00404373 mov     [esp+179D4h+var_179A1], 'n'
.text:00404378 mov     [esp+179D4h+var_179A0], 'd'
.text:0040437D mov     [esp+179D4h+var_1799F], 'l'
.text:00404382 mov     [esp+179D4h+var_1799E], 'e'
.text:00404387 mov     [esp+179D4h+var_1799D], 0
```

Illustration 4 – String Construction

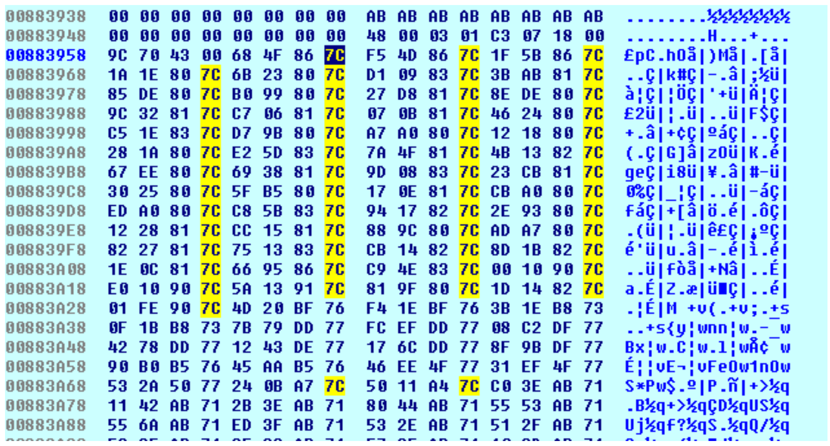
2.4.3 API Address Resolution at Runtime

The sparse import table of the sample tells the analyst at first glance already that most likely more API functions will be resolved at run time. From kernel32.dll just 4 functions are imported, two of which are needed for dynamic loading of libraries and API offsets. Said two functions are LoadLibraryA and GetProcAddress. These are invoked in a loop by the executable, feeding them all the names of the APIs which are constructed before. This way the addresses of the desired functions are retrieved and later stored in a separate memory region which will be the reference table for every API call in the future. For invoking an API function the address of the API offset object is loaded into a register and the offset according to the desired function is added, like shown in the following example.

```
0040F502 mov      edx, dword_43D190
0040F51D call     dword ptr [edx+68h] ; createeventa in kernel32 7C83089D
```

Dword_43D190 contains the pointer to the resolved API addresses throughout execution. The following graphic shows the memory region where the addresses are stored, marked 7C.. addresses belong to kernel32.dll on the used WinXP system.

A table of resolved functions and their offsets can be found as attachment to this document (see attachment [1]).



```
00883938  00 00 00 00 00 00 00 00  AB AB AB AB AB AB AB AB  ........½½½½½½½½
00883948  00 00 00 00 00 00 00 00  48 00 03 01 C3 07 18 00  ........H...+...
00883958  9C 70 43 00 68 4F 86 7C  F5 4D 86 7C 1F 5B 86 7C  £pC.hOå|)Må|.[å|
00883968  1A 1E 80 7C 6B 23 80 7C  D1 09 83 7C 3B AB 81 7C  ..Ç|k#Ç|-.â|;½ü|
00883978  85 DE 80 7C B0 99 80 7C  27 D8 81 7C 8E DE 80 7C  à¦Ç|¦ÖÇ|'+ü|Ä¦Ç|
00883988  9C 32 81 7C C7 06 81 7C  07 0B 81 7C 46 24 80 7C  £2ü|¦.ü|..ü|F$Ç|
00883998  C5 1E 83 7C D7 9B 80 7C  A7 A0 80 7C 12 18 80 7C  +.â|+¢Ç|ºáÇ|..Ç|
008839A8  28 1A 80 7C E2 5D 83 7C  7A 4F 81 7C 4B 13 82 7C  (.Ç|G]â|zOü|K.é|
008839B8  67 EE 80 7C 69 38 81 7C  9D 08 83 7C 23 CB 81 7C  geÇ|i8ü|¥.â|#-ü|
008839C8  30 25 80 7C 5F B5 80 7C  17 0E 81 7C CB A0 80 7C  0%Ç|_¦Ç|..ü|-áÇ|
008839D8  ED A0 80 7C C8 5B 83 7C  94 17 82 7C 2E 93 80 7C  fáÇ|+[â|ö.é|.ôÇ|
008839E8  12 28 81 7C CC 15 81 7C  88 9C 80 7C AD A7 80 7C  .(ü|¦.ü|ê£Ç|¡ºÇ|
008839F8  82 27 81 7C 75 13 83 7C  CB 14 82 7C 8D 1B 82 7C  é'ü|u.â|-.é|ì.é|
00883A08  1E 0C 81 7C 66 95 86 7C  C9 4E 83 7C 00 10 90 7C  ..ü|fòå|+Nâ|..É|
00883A18  E0 10 90 7C 5A 13 91 7C  81 9F 80 7C 1D 14 82 7C  a.É|Z.æ|ü■Ç|..é|
00883A28  01 FE 90 7C 4D 20 BF 76  F4 1E BF 76 3B 1E B8 73  .¦É|M +v(.+v;.+s
00883A38  0F 1B B8 73 7B 79 DD 77  FC EF DD 77 08 C2 DF 77  ..+s{y¦wnn¦w.-_w
00883A48  42 78 DD 77 12 43 DE 77  17 6C DD 77 8F 9B DF 77  Bx¦w.C¦w.l¦wÅ¢¯w
00883A58  90 B0 B5 76 45 AA B5 76  46 EE 4F 77 31 EF 4F 77  É¦¦vE¬¦vFeOw1nOw
00883A68  53 2A 50 77 24 0B A7 7C  50 11 A4 7C C0 3E AB 71  S*Pw$.º|P.ñ|+>½q
00883A78  11 42 AB 71 2B 3E AB 71  80 44 AB 71 55 53 AB 71  .B½q+>½qÇD½qUS½q
00883A88  55 6A AB 71 ED 3F AB 71  53 2E AB 71 51 2F AB 71  Uj½qf?½qS.½qQ/½q
```

Illustration 5 – API Call Jumptable

This addressing method prevents IDA Pro from automatically naming the function and from listing the expected arguments for each call. The arguments had to be looked up via MSDN library and corresponding comments had to be added by hand, if needed.

2.5 Indirect Function Calls

Most core functionality of the malware is laid out in object oriented design with numerous virtual functions. Virtual functions are a concept of object oriented programming, to realize polymorphic design of classes and linking of virtual functions at runtime. The virtual functions of a class are listed in the virtual function table, short vftable. This table maps methods to according implementations of functions in memory. When deriving a class it derives the superclass' virtual functions, which can be overloaded, if desired.

As with polymorphic objects it is not always clear at compile time which offset is to be called when a virtual function is invoked. Therefore indirect addressing is used. When a virtual function is called at runtime, the vftable in the object is resolved to load the right offset.

This is needed because in polymorphic programming an object is always of one or more types, as to say it is of its own class as well as its superclass, if there is any, and eventually other superclasses as well.

The following code snippet shows an example, picked out of the analyzed sample. The call in line 6 invokes a virtual function, of an object initially pointed to by esi. The references in the vftables are resolved, until the right offset is found. It is very hard to determine the offset to be called through static analysis, as the memory layout with all vftables of all classes derived from the same superclass to be considered.

```
.text:0042238A mov     eax, esi
.text:0042238C mov     esi, [esi]
.text:0042238E mov     edi, [eax+8]
.text:00422391 mov     ecx, edi
.text:00422393 mov     eax, [edi]
.text:00422395 call    dword ptr [eax+8]
```

As this is a design attribute of object oriented programming it is not considered an obfuscation method. But certainly, virtual function calls complicate analysis of malicious code significantly.

2.6 Timing Attacks using GetTickCount

In the executable there exist 7 calls to the GetTickCount function, which retrieves the number of milliseconds that have elapsed since system startup. It is a commonly known trick of malware, to detect the presence of debuggers by regularly checking the tick count, as to say by calculating the ticks that are passing in between the calls. If the number is too high, there is presumably a debugger halting the process in between two calls to GetTickCount.

The malware at hand would run perfectly fine inside the debugger, after the initial anti-debugging trick of checking dwFlags. Just when single stepping inside one of the multiple threads it would not resume without catching an exception sooner or later, from which it could not recover.

The calls to GetTickCount happen in different pieces of the code, the value checks are supposedly done somewhere later in the code. Anyway, it is not quite necessary to find and patch them all, as most debuggers are prepared for this trick. For IDA Pro there is a plugin called IDA Stealth, an equivalent for OllyDebug is PantOm Plugin. These plugins incorporate several stealth mechanisms, which aim to hide the debugger from the debuggee.

Also they can fake return values for GetTickCount. For the analyzed sample a random delta of 30.000 was found sufficient.

The sample also uses WinMM library to create time callback events. This way the malware could also check for the right timing, but it is assumed that this is not the case as no behavior was noticed that would support this theory.

3. Malware Startup

The analyzed sample is a non-polymorphic file infector that compromises executable files on the system. It replaces the original executable with its own image, barely altered, steals the icon to look genuine and stores the original file, hidden and without extension, in the same folder. When invoked, the malware checks if there is an original counterpart to its own filename, without extension. If so, this executable has to be started too.

3.1 Synchronization Methods for Multiple Instances

Also, the sample checks if the system has been compromised before by the same malware. This is indicated by the presence of two Windows system events, shortly named event2 and event3 (the hardcoded names of extracted events are listed in attachment [2]). These two events are used for synchronization of running instances of the malware, as can be seen at a call to WaitForMultipleObjects in START_MW function, followed by a switch statement (see illustration 6).

A second check verifies if there is a fingerprint file present in the systems %TEMP% directory (extracted filenames are also hardcoded and listed in attachment [2]). The content of this file is checked against a value, generated by a hardcoded algorithm, without system specific values. Presumably all variants of the same version of the analyzed sample produce the same fingerprint file. This could change with later updates.

The generation of said file happens after successful malware startup in the function START_MW. Also, creation of event2 and event3 happen at this time.

The following graphic simplifies the malware initialization phase in IMPLICIT_MAIN function. When executing, the sample will either terminate or startup the malware functionality and run as multi-threaded process in background. The different checks before startup are marked in yellow.

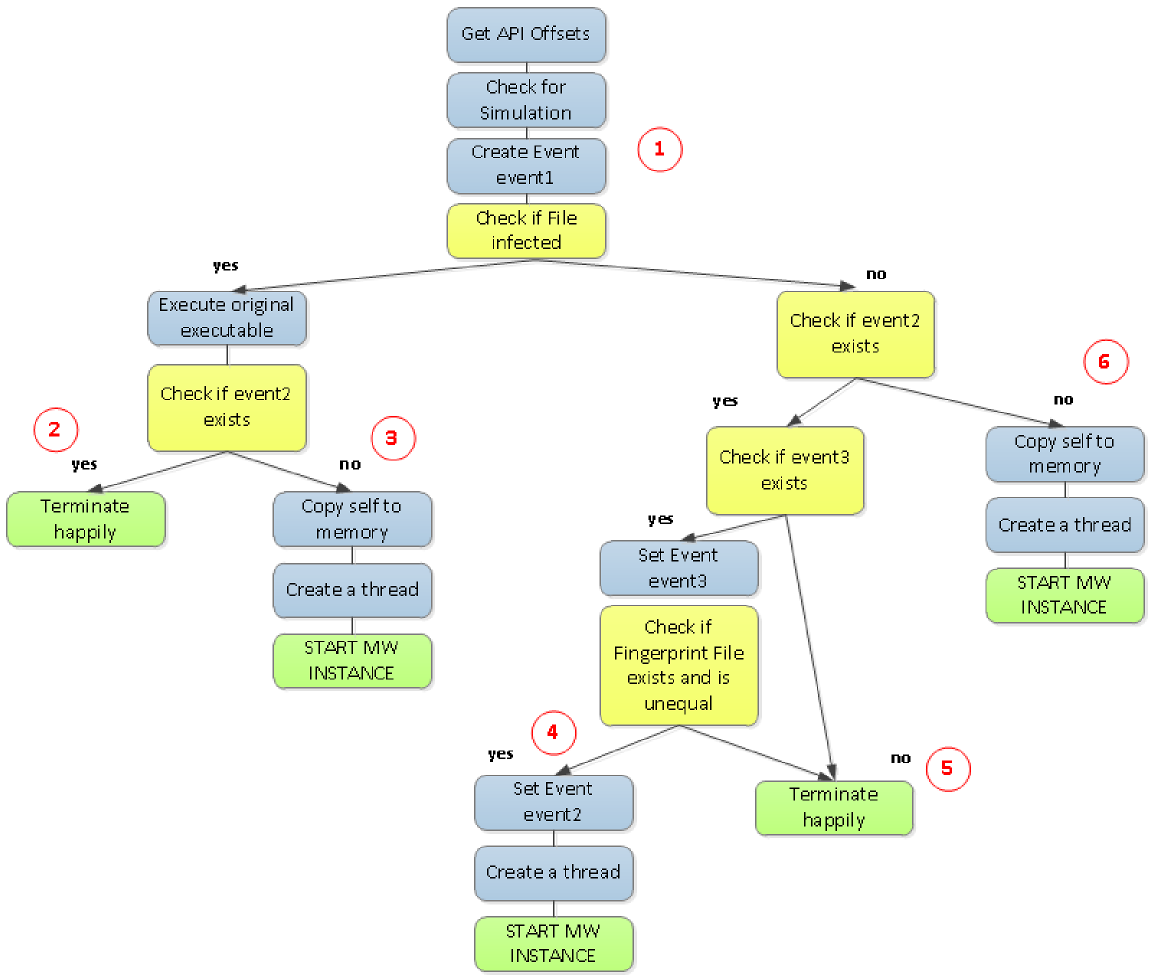


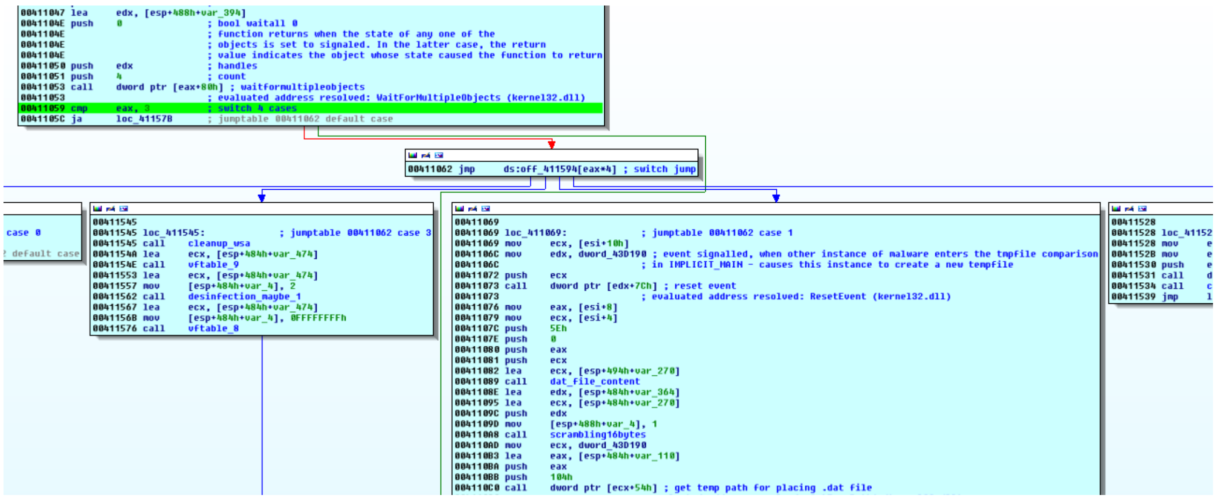
Illustration 6 – Malware Startup Flowchart

1. Initialization – The API object is created, the gethostbyname call is done to check for simulation and a named event is created, shortly termed event 1. If createevent fails, because event1 exists already, this information is saved for performing checks later but does not influence execution immediately.
2. If an original counterpart can be found it is executed, if additionally a handle to event2 can be retrieved the sample performs some cleanup code and then terminates execution.
3. If an original is found and executed, but the openevent call on event2 fails the sample initiates a normal startup procedure similar to point 6, which would be the initial infection routine on a clean system.
4. If no original counterpart is found and a handle for event2 can be retrieved the malware enters one single function for checking if a handle for event3 can be retrieved. If so, the event is set to signaled state, the fingerprint file, if existent, is read, deleted, and the content is compared to the self-generated fingerprint value. So if event3 exists, the fingerprint file exists and the two fingerprints are not equal the malware takes course number 4: event2 is set to signaled state and usual startup procedure is initiated. Signaling of event2 presumably causes the former, running instance of the malware to terminate.
5. If event3 does not exist, the fingerprint file cannot be found or the fingerprint values are equal the malware terminates execution.

6. In case no original executable can be found and event2 is not present the system is supposedly clean and initial infection routine is started. This includes creating a copy of the image of the own executable in memory and creation of the first thread, that will be the file infection routine.

Having this flow in mind some conclusions can be drawn. The mentioned events are used to coordinate the malware, when multiple infected executables are started at the same time. The fingerprint file could be some sort of version management. When a malware instance is running, and a new infector starts up, which calculates a different fingerprint, the former instance is stopped and the new infector is started. As mentioned before, signaling event2 causes an instance which has reached the WaitForMultipleObjects statement in START_MW function to terminate gracefully.

The same wait statement is waiting for signaling of event3. When this happens, the fingerprint file is generated anew. The following screenshot shows mentioned switch statement, case 0 or 3 belong to signaling of event2 (termination), case 1 belongs to signaling of event3.



```
00411047 lea     edx, [esp+488h+var_394]
0041104E push    0               ; bool waitall 0
0041104E                         ; function returns when the state of any one of the
0041104E                         ; objects is set to signaled. In the latter case, the return
0041104E                         ; value indicates the object whose state caused the function to return
00411050 push    edx             ; handles
00411051 push    4               ; count
00411053 call    dword ptr [eax+80h] ; waitformultipleobjects
00411053                         ; evaluated address resolved: WaitForMultipleObjects (kernel32.dll)
00411059 cmp     eax, 3          ; switch 4 cases
0041105C ja      loc_41157B      ; jumptable 00411062 default case
```

```
00411062 jmp     ds:off_411594[eax*4] ; switch jump
```

```
case 0
2 default case
```

```
00411545
00411545 loc_411545:             ; jumptable 00411062 case 3
00411545 call    cleanup_wsa
0041154A lea     ecx, [esp+484h+var_474]
0041154E call    vftable_9
00411553 lea     ecx, [esp+484h+var_474]
00411557 mov     [esp+484h+var_4], 2
00411562 call    desinfection_maybe_1
00411567 lea     ecx, [esp+484h+var_474]
0041156B mov     [esp+484h+var_4], 0FFFFFFFFh
00411576 call    vftable_8
```

```
00411069
00411069 loc_411069:             ; jumptable 00411062 case 1
00411069 mov     ecx, [esi+10h]
0041106C mov     edx, dword_43D190 ; event signalled, when other instance of malware enters the tmpfile comparison
0041106C                         ; in IMPLICIT_MAIN - causes this instance to create a new tempfile
00411072 push    ecx
00411073 call    dword ptr [edx+7Ch] ; reset event
00411073                         ; evaluated address resolved: ResetEvent (kernel32.dll)
00411076 mov     eax, [esi+8]
00411079 mov     ecx, [esi+4]
0041107C push    5Eh
0041107E push    0
00411080 push    eax
00411081 push    ecx
00411082 lea     ecx, [esp+494h+var_270]
00411089 call    dat_file_content
0041108E lea     edx, [esp+484h+var_364]
00411095 lea     ecx, [esp+484h+var_270]
0041109C push    edx
0041109D mov     [esp+488h+var_4], 1
004110A8 call    scrambling16bytes
004110AD mov     ecx, dword_43D190
004110B3 lea     eax, [esp+484h+var_110]
004110BA push    eax
004110BB push    104h
004110C0 call    dword ptr [ecx+54h] ; get temp path for placing .dat file
```

```
00411528
00411528 loc_41152
00411528 mov     e
0041152B mov     e
00411530 push    e
00411531 call    d
00411534 call    c
00411539 jmp     l
```

Illustration 7 – thread0 WaitForMultipleObject

As the fingerprint checking is only done when no original counterpart can be found, it is believed that the possible update mechanism is just to be executed by the initial infector, not by its infected copies.

It is important to mention that the malware incorporates a disinfection routine, that's called only in case 3 of shown switch statement. Apparently one event causes the malware to clean the system and terminate afterwards. The disinfection routine is described in more detail in section 7.5.

4. Multi-Threading Model

In a multi-threaded application different tasks are delegated to different threads, which can be virtually or physically parallelized, depending on the processor architecture. For management of the threads and interaction between the threads exist various ways of inter-thread communication. The malicious sample at hand starts, apart from thread 0, 8 more threads during startup, naturally with various purposes. There is one thread with the sole purpose of infecting files, which is coded pretty much straight forward. Other threads implement functionality for sending data to the command and control server (short C&C), receiving data from the C&C and dispatching of commands from the C&C.

The multi-threading model and inter-thread communication was not reconstructed in full detail but just as far as needed for understanding the role of its components.

4.1Inter-Thread Communication

Inter-thread communication and coordination is accomplished with various mechanisms. Four methods could be identified:

- Event-driven thread/process-synchronization
- Message-driven thread communication
- I/O completion port for dispatching of TCP/UDP messages
- Critical sections for management of concurrency

Events in Windows are a synchronization mechanism that works on both levels, process- and thread-level. They can take two states, signaled and non-signaled. When an event is set its state is raised into signaled state. A process or thread that is waiting for this exact object will recognize and start execution. The event will then be reset, automatically or manually.

Windows events, as mentioned in the last chapter, are mainly used for control of the application as a whole. Besides, thread 1, later called *fileinfector*, is also triggered by an event.

Windows Messages are usually the standard mechanism for controlling Windows desktop applications and windows. Communication is handled in message queues, one thread posts a message to another ones message queue, the other thread fetches the message from its queue and starts execution. The analyzed sample uses Windows messages for communication of its working threads, which handle the communication with the C&C.

I/O Completion Port (IOCP) is an API for handling multiple asynchronous input/output operations. This is necessary when multiple threads handle the input or output of one object, like in this case a socket. In fact, the analyzed bot seems to handle the incoming messages from the C&C server using I/O completion port. The intention behind this is perhaps to handle more than one server command at a time. With IOCP the operations of receiving commands and execution of according actions can be separated. It would even possible for the malware to operate multiple sockets and process input of various control servers.

For protection of memory that is eventually accessed by more than one thread the mechanism of critical sections is implemented. This is the most important mechanism for interchanging data between the malware's threads, as possibilities to pass data by Windows messages is limited. Critical sections are initialized to protect access around I/O completion data structures, time event data structures and the C&C domain character string.

4.2 Details about Started Threads

When the malware startup was successful it runs on 9 threads in total, of which one is thread0, one the file infecting thread, two seem to control reception of data from the C&C using I/O completion port mechanism and the remaining four implement functionality for windows message passing. Thereof three are created with an attached time callback thread each. The fourth is actually one of the time callback functions, which runs in its own thread.

The applications threads, in order of their startup (not creation), naming intends to sum up the main purpose of the thread:

0.	thread0	initial application code
1.	timecallback_ptmessage	offset 422426
2.	fileinfector	offset 411AA0
3.	get_queued_compstatus	offset 42B43A
4.	getmessage_loop	offset 42248B
5.	getmessage_loop	offset 42248B
6.	recv_post_queued_compstatus	offset 42B0C5
7.	cnc_cmd_switching	offset 4211E7
8.	getmessage_loop	offset 42248B

Furthermore, the malware is capable of starting several more threads, which are able to receive either TCP or UDP data or handle GDI functionality (MS Graphics Device Interface).

0 – thread0

The initial thread hangs mostly at a WaitFormultipleObjects instruction, waiting for one of four events to be set. These are two named events, event2 and event3 as mentioned before, and two nameless events. Setting of event2 or one of the unnamed events will cause the application to terminate. Setting of event3 tells the application to recreate its fingerprint file, and when setting the second unnamed event an unspecified event will be reset and *thread0* continues to wait.

Thread0 is in charge of controlling the entire process, and terminating it if told so.

1 – timecallback_ptmessage

Timed callbacks are implemented using the WinMM library (Windows Multimedia Library) for creating timers, which call a specified callback function. The timing event is configured with either 10, 1000 or 5000 milliseconds. In the analyzed code time callbacks are always generated along with an associated thread for *getmessage_loop*.

The malware uses for every time callback the same function. This function implements a loop for posting messages to thread message queues, actually with message identifier 401. 401 is an ID for user defined messages. It is assumed that the purpose of this constellation is to post timed notifications to a queue, which are dispatched by an associated instance of *getmessage_loop*.

2 – fileinfector

The *fileinfector* thread is the one which is granted most CPU time, when the other threads are sleeping. It is driven by a Windows event and its purpose is to replace the images of executables on disk with its own malicious code. Details about this thread are to be found in section 5.

3 – get_queued_compstatus

This thread waits for a queued completion status message to be posted to the I/O completion ports queue. At thread creation the thread is passed the only I/O completion object created by the application. It is the same object thread 6, *recv_post_queued_compstatus* is feeding.

The received data is processed, maybe even deobfuscated/decrypted and stored in an object that is protected via a critical section. Eventually some networking routines are called that send a message back to the C&C.

4, 5 & 8 – getmessage_loop

The malware starts three threads with the same entry point to *getmessage_loop* function. These are always created and associated with a timing event callback, which posts thread messages. The *getmessage_loop* function just executes its body when the received message is 401, just as the time callback function would send it.

The functions body is basically a call to a single, virtual function. This virtual function is a perfect example for difficulties of reversing polymorphic functions. The same function call can invoke two different functions, because of being referenced indirectly via vftable of an object.

The first virtual function to be called is a routine to resolve the hardcoded domain ns.dns3-domain.com to a valid network address via DNS and to send an initial message to the C&C. This message contains a GUID for identification purposes, requested from the system.

The second function can either also send a message over the network, or post another thread message. Recipient is thread 7, named *cnc_cmd_switching*, as this is the only other thread waiting for thread messages, besides the *getmessage_loop* threads.

At the time this report was written it is not perfectly clear what the purpose of multiple *getmessage_loop* threads is. Possibilities are either load balancing or confusion of the analyst. Starting multiple threads with the same functionality would only make sense, if one single thread would be overburdened with the work to process. It is assumed that the multi-threaded design is laid out for load balanced processing of C&C instructions, so multiple threads with the same purpose could make sense.

6 - recv_post_queued_compstatus

This thread is settled on the input side of the I/O completion port, defined before along with the *get_queued_compstatus* thread. The threads function itself is a loop for receiving incoming UDP datagrams and passing them via PostQueuedCompletionStatus to the I/O completion object. From there they will be fetched by the *get_queued_compstatus* thread.

7 – cnc_cmd_switching

The command getmessage() is only called twice in all the code, the *getmessage_loop* routine and in the *cnc_cmd_switching* routine. It will also just process a received message if it is of type 401. Main purpose of this thread is to instantiate an object for C&C command processing and to call the according method. The C&C command is presumably passed via a critical section data structure, and not via thread messages.

4.3 Thread Workflow Diagram

The following diagram is just a simplified sketch on how the workflow in between the multiple threads is most likely designed; it does not lay claim to be complete. There definitely exist more threads the malware can possibly start, like two threads for recv/recvfrom functionality, but these are not counted in to the core model.

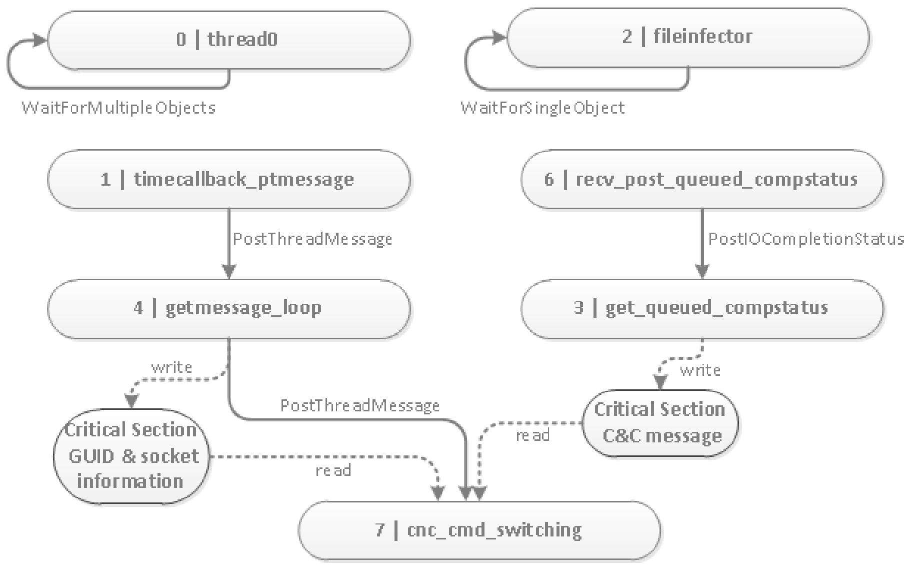
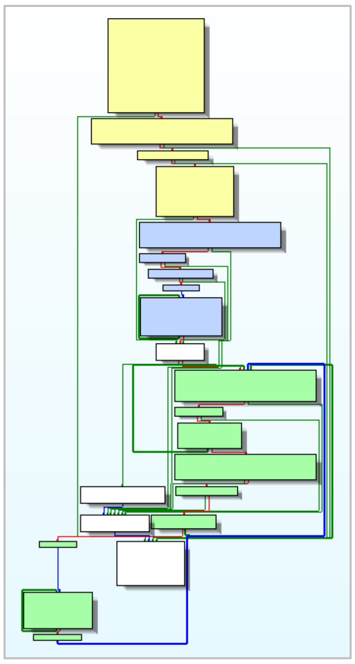


Illustration 8 – Thread Workflow Diagram

5. File Infection



The first thread the malware creates is the file infection routine. Basic functionality of this routine is to enumerate running processes and entries in SOFTWARE\Microsoft\Windows\CurrentVersion\Run to select a set of executables and replace their image on disk with the malware itself. It does this by setting the hidden attribute on the original file and removing the extension, while creating a copy of itself with minor modifications and writing it under the filename of the "infected" file back to disk. It also steals the icon of the former original to look non-suspicious. As mentioned before when describing the malware startup, if the copied malware ever gets loaded, it first starts the original executable before it eventually reaches the START_MW function.

The IDA Pro Graph can actually be used for explaining the different steps of the file infection procedure. The three marking colors indicate what regions are considered to be initial setup (yellow), infection routine (blue) and re-infection loop (green).

5.1 Initial Infection

5.1.1 Check for Chinese AV-Products

When entering the main function of the file infection routine a method is called to check if one or both of two processes is found within a snapshot of all running processes: ZhuDongFangYu.exe from Qihoo360 and RavMond.exe from Rising Antivirus, which are both processes of Chinese anti-malware products. If detected, the whole thread terminates immediately.

As mentioned in the next section these are probably applications that the malware developer ran on his own system and therefore does not want to infect files on systems running Qihoo360 or Rising.

5.1.2 Module Name Filtering

As a setup for file infection the malware first has to select suitable executables. To achieve this it creates a list of two sorts; first all executables that have an entry in SOFTWARE\Microsoft\Windows\CurrentVersion\Run, second all executables which are present in a snapshot of currently running processes.

The applications from the startup registry key are only added if they have the ".exe"-ending.

The modules listed in the process snapshot are filtered by a list of application and folder names, which are apparently excluded from infection. For filtering a number of strings is generated at runtime and matched against either the folder name or are searched for in the whole path. Some of the filter strings are quite interesting as they seem to give some insight on what applications the malware author is using on this own system. Most likely these applications are filtered to avoid infection on his own system. The most interesting ones are listed below:

- Directory should not be desktop or temp
- Pathname should not contain either of the following strings
- :\windows
- netthief
- exebinder
- \qq
- visual studio
- microsoft office\
- \thunder\
- \360
- \aliwangwang\
- \win zip\
- \winrar\
- \globallink\game\
- \qqdoctor\
- \rising\
- \aliim.exe
- \avira\
- \world of warcraft\

Actually the term netthief can be connected to another piece of malware, written by a Chinese author. It is called Netthief RAT (Remote Access Trojan) Some research also revealed that the domain used by the analyzed sample is connected to Netthief.

Further reasons for exclusion of a process from the final list are:

- The filename (without the path) of the running process is equal to the malwares filename
- The paths second and third letter are not equal to ":\", as it is for example with the paths of Windows system services like smss.exe or csrss.exe, starting with \systemroot- or ??\-prefix
- The last four characters of the image name do not equal ".exe"
- The running executable in question is smaller than 10KB

If the module name passes all filtering measures it is added to a list of executables that will be infected by the malware.

5.1.3 The Infection Routine

For actual infection of an executable a copy of the image saved at initialization phase is prepared and modified.

1. Some modifications are made to the image at offset 117A, which are unique to every copy of the malware. Likely, to give the copy a unique identifier or some seed values for obfuscation algorithms embedded in the malicious code.
2. The executable to infect is searched for a resource section, based on the string ".rsrc". If .rsrc is found it is searched for an icon in the right size and shape, which can be copied to the malware image. By copying the icon of the file to the malicious image in memory will make the malware copy look like the genuine file. The changed file size and creation time stamp though could tell there is something wrong with that file.
3. Only if copying the icon was successful and the original executable has not yet been replaced by a malware copy, the actual replacement routine is initiated. Therefore the original is renamed to remove the .exe-extension and its file attributes are altered to set the hidden flag. Then, finally, a file is created with path and filename of the file to infect and the customized malware copy is written to this file handle.

5.2 The Re-Infection Loop

After initial infection the thread enters its final loop, where it periodically checks if the list of infected images has grown, specifically if new processes have started which follow the described preconditions. The routine maintains a list of infected modules. When a new process appears that's not yet on the list it is added and execution switches to a loop, identical to the initial infection loop. There, if not already infected, the image of the started process, respectively images if there was more than one candidate started, is eventually replaced with a malware copy.

After calling the infect method on every recently listed module the routine switches back into looping process snapshot after snapshot, until another candidate appears.

6. Network Communication

The analyzed sample is categorized by anti-virus industry as backdoor, spy or bot; so it should come with extended network functionality. Not much of the communication capabilities can be analyzed dynamically though, the sample only tries to communicate to one single, hardcoded domain. The sample tries to connect to ns.dns3-domain.com, but no answer was ever received from the server. The according network address 125.34.39.47 does not answer, neither ping requests nor requests of the bot.

The domain actually still exists and is registered until July 2013.

```
Whois v1.01 - Domain information lookup utility
Sysinternals - www.sysinternals.com
Copyright (C) 2005 Mark Russinovich

Connecting to COM.whois-servers.net...
Connecting to grs-whois.hichina.com...

Domain Name ..................... DNS3-DOMAIN.COM
Name Server ..................... dns21.hichina.com
                                  dns22.hichina.com
Registrant ID ................... hc564383063-cn
Registrant Name ................. sun rui
Registrant Organization ......... sun rui
Registrant Address .............. tian jin shi he xi qu mei jiang dao 18 hao
Registrant City ................. tian jin shi
Registrant Province/State ....... tian jin
Registrant Postal Code .......... 300221
Registrant Country Code ......... CN
Registrant Email ................ niceday122@126.com
… Output omitted …

Expiration Date ................. 2013-07-11 03:04:33
```

6.1 Sending Messages to the C&C

The analyzed sample implements functionality for use of send and sendto, as well as recv and recvfrom. System calls send/recv are used to operate TCP stream sockets while sendto/recvfrom operate UDP datagram sockets. This means that the analyzed sample has capability to use both, TCP and UDP connections. Most data sending routines in the malware code call to the UDP variant. Actually, message exchange with the C&C server is operated via UDP connections.

The code for sending of UDP datagrams, which is used for most messages to the C&C, is implemented in around 35 methods of which all finally call into one single function that invokes the sendto system call. The numerous preceding methods are presumably obfuscation, checking of parameters or altering the data to be sent. The network message protocol with the C&C was not analyzed further.

The preceding methods to sendto use critical sections quite frequently. This part of the code is believed to be implemented in a thread-safe manner, so that the threads can call into sendto concurrently.

Following the function calls towards the sendto method four main operators could be identified that would eventually use the sendto system call:

1. getmessage_loop – the main method for thread4 when resolving the C&C domain and sending the “HELLO”-message to the server
2. get_queued_compstatus – the main method for thread3 in case of non-failure
3. cnc_cmd_switching – in case of failure of some sort the main function of thread7 sends notifications to the server
4. message_to_cnc – this function was identified as being used predominantly to send messages and data to the C&C server

Point 1-3 depend on a specific thread, point 4 mentions a single method that is used throughout the code. Mostly it is called from thread7, where the processing of C&C commands happens and messages as well as data flow back to the C&C after operation.

6.1.1 Initial “HELLO”-Messages

The getmessage_loop resolves the remote server’s domain and sends “HELLO”-packages for means of registration. Two possible destination ports were identified for these messages, which are 53 (Domain Name Service) and 8000 (Intel Remote Desktop Managemet Interface).

The messages each contain a preceding GUID (Globally Unique Identifier), which is probably used for separation of infected machines on the server side. All together the messages are 25 bytes long, 16 bytes GUID plus 9 bytes custom information.

7. C&C-Command Processing

The method, identified as central node for processing of C&C commands, is shown for illustration purposes in an IDA Pro screenshot.

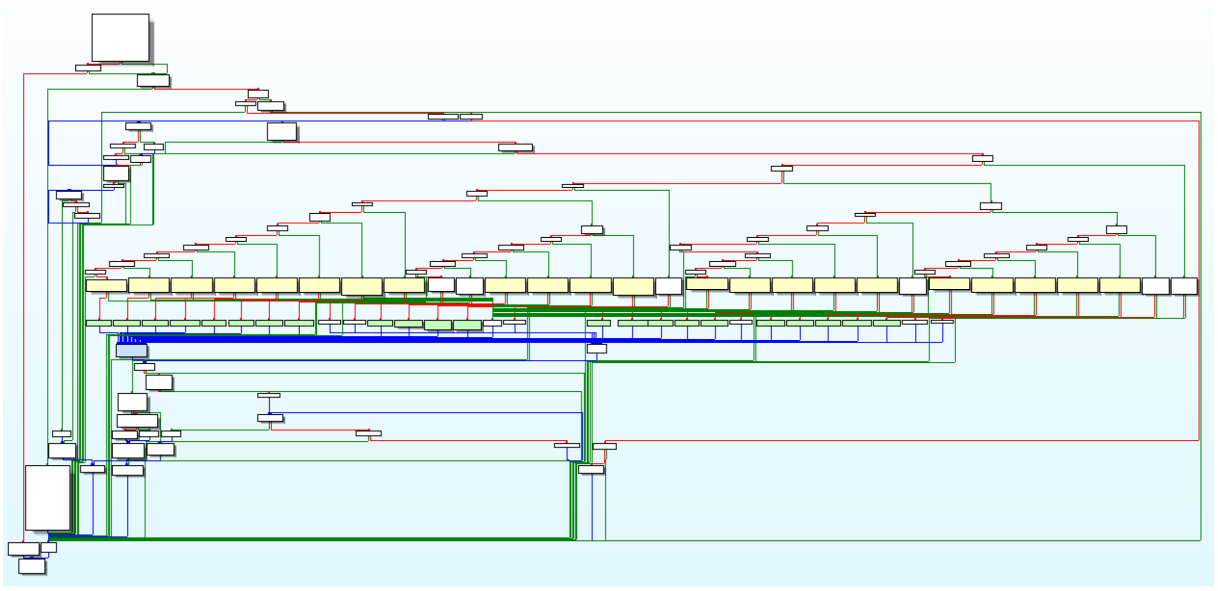


Illustration 9 – C&C Command Processing Method

The C&C instructions are numerical values, which are broken down in the method (called cnc_cmd_switching) to execute the according action. What the numerous branch instructions do is basically to allocate memory (yellow), create an object derived from a C&C command superclass (green) and finally execute the first virtual method (blue).

22 operations were identified, that the execution call (blue) could eventually perform. In succession they are simply listed in their right to left order.

7.1 Control Operations

7.1.1 terminate

The sample can terminate its own process by invoking the following pseudo code: TerminateProcess(OpenProcess(GetCurrentProcessId(), 0, PROCESS_TERMINATE),0).

7.1.2 system_shutdown

The malware grants itself the SeShutdownPriviledge and invokes ExitWindowsEx with the parameter 0Ch. 0CH means that the flags EWX_POWEROFF and EWX_FORCE are set, which will force a system shutdown.

7.1.3 spawn_console_process

This method implements code for spawning a console process with redirected standard handles. This means that the malware can start a child process, but redirect its standard handles to control input and output to and from this process. More information and example code can be found in MS Knowledgebase Article 190351 (see source [4]).

The code for the console process‘ thread is of course located in a separate method. This method consists of a loop, where a file is read and it’s content is compared to a given buffer. At the time of writing this report it was not perfectly clear what the purpose of this action is, but it is believed that the function, when it has found specific attributes, will copy the file content to memory and post a thread message with ID 402.

7.1.4 shellexecute

This method uses the ShellExecute API call to launch an application. If the received parameter is does not point to an executable file then ShellExecute opens the associated application. This is followed by a notification to the C&C server.

7.1.5 notify_cnc

An unidentified message is sent to the C&C server.

7.1.6 notify_cnc2

An unidentified message is sent to the C&C server.

7.2 Multimedia Operations

7.2.1 gdi_capture_window

This method takes a capture of the actual desktop window and saves it to a .tmp-file in %TEMP% directory (file3). The capture is taken by an instance of CreateCaptureWindowA, which implies starting a thread and creating a nameless event. The thread then switches into waiting state until the method is called a second time. Then the same method will set the nameless event and the capture will terminate. The GDI jpeg encoder is used for creation of file3.

Afterwards a message to the C&C server is sent.

7.2.2 gdi_dca_screenshot

This method creates a device context for the display device and produces a graphical snapshot of the actual screen display. Using the GDI jpeg encoder this image is saved in a .tmp-file in %TEMP% (file4).

Afterwards a message to the C&C server is sent.

7.2.3 send_multimedia

This method is designed for compressing and sending of file3 or file4, if they were created before by *gdi_dca_screenshot* or *gdi_capture_window*. In case of success the compressed data is sent to the remote server, otherwise a notification message is sent.

7.3 File System Operations

7.3.1 file_listing

This method accepts a file- or directory name. It walks recursively down the directory tree and saves certain names to a list, which is then packed and sent to the C&C. The sample incorporates code from the zlib compression library, which lets assume that certain amounts of data to be sent to the C&C is compressed. This helps obfuscation and reduces network traffic.

7.3.2 directory_listing

This method walks a given directory and lists file entries plus the information if the containing directory contains a ".." entry. This indicates that it has a parent directory. The final list is not compressed before sending it to the C&C.

7.3.4 directory_listing2

This method accepts a path to a directory and lists all entries plus the information if the object is a file or a directory, indicated by 0 or 1. This list is then compressed and sent to the C&C server.

7.3.5 create_directory

This method accepts a character string as argument and creates an equally named directory on disk. This is followed by a notification to the C&C server.

7.3.6 copy_file

This method can copy a given file or list of files to a given place on disk. Afterwards another message to the C&C server is sent.

7.3.7 delete_file

This method can delete a given file or list of files from disk. Afterwards another message to the C&C server is sent.

7.3.8 rename_file

This method can rename one file on disk. Afterwards another message to the C&C server is sent.

7.3.9 write_to_file

An unidentified character string is written to a file on disk.

7.4 Other Operations

7.4.1 get_volume_info

Requests information about file system volumes and tries all possible drive letters beginning with A:\ to retrieve it. More specifically it requests the volume names and saves them to a buffer. These names are then sent to the C&C server.

7.4.2 get_window_text

This method enumerates the title bar texts of all uppermost desktop windows of each running process, if the window is not hidden or overlapped. This list is packed and sent to the C&C server. This way the attacker can get a clue what desktop applications are running at the moment.

7.4.3 check_for_fingerprint

Reads a file (file2, cmp. attachment[2]) from the systems %TEMP% directory and checks if its content matches the samples fingerprint as mentioned when describing the malwares startup procedure.

7.4.4 smss_sysinu_tempfiles

Reads file2 from the systems %TEMP% directory, modifies its content and copies it back to %TEMP% twice, once called %TEMP%\smss.exe and once %TEMP%\sysinu.dll.

7.4.5 dat_file_createwrite

This function fulfills various purposes. The C&C command is broken down into three defined cases. The method can either create file2, same as used in *smss_sysinu_tempfiles* and *check_for_fingerprint*, in the %TEMP% directory, or split a path down to create directories, so to say create the directory path from scratch and place a new file into it. The file is created empty and filled at a later point in time.

This would be case 3, when the C&C command indicates another execution path, where first a notification is sent to the server and then a predefined value is written to file2.

Creating and filling another fingerprint file could be an additional method for managing the different instances of the malware on the system. Also it could be used helping in a self-update mechanism.

7.5 Desinfection Routine

The disinfection routine is called from *thread0*, the initial thread, when a specific unnamed event is set signaled. The routine starts by replacing its own image on disk with the hidden original file, if there exists one. This is achieved by deleting its own image, removing the hidden attribute from the original and renaming it back to original.exe.

Afterwards it enumerates again all executable images that are registered for being loaded on startup in selected registry keys. Same happens with the enumeration of running processes, just as mentioned in the description of the *file_infector* thread. With this list prepared the routine calls the disinfection method also on other potentially infected images and cleans them.

Finally a method is called that walks recursively through all volumes of the system, checks for files that end with ".exe" and checks for an infection by searching for a hidden original. If infected the file in question is cleaned.

8. Conclusions

The malicious sample at hand is highly-sophisticated in its operation and indeed very interesting. Analysis is challenging due to obfuscation methods and junk code, as well as numerous virtual function calls that are hard to resolve. Debugging is problematic due to the multi-threaded design and has its constraints as no responses from the C&C server were received.

The functionality of the malware is found to be considerably dangerous and extensive. The malware can basically take full control over the system. It can spy a great amount of information, from screenshots and live captures, to directory listings and running desktop applications. It can execute other applications, shut down its own process, copy, rename or delete files on disk, eventually even download other executable and update itself.

On the other side some possible weaknesses were identified too. The sample does not make use of a runtime packer or encryption layers. The anti-debugging measures were quite quick to pass by. The file infection routine is not polymorph, it is easy for anti-virus software to detect every sample of this variant.

It is not clear if this was intention of the malware author, but the sample implements various outdated or deprecated Win32 API calls, as for example SHFileOperation.

Various questions are still open and would be of interest for further analysis. The inter-thread communication via critical sections could be analyzed and most likely would reveal a lot more insight on how the threads really operate. The communication protocol from bot to server would be of high interest. Some functionality of the documented C&C command processing methods is still unclear.

Sources

[1]
MSDN Library (Microsoft Software Developer Network)
http://msdn.microsoft.com/

[2]
„A Crash Course on the Depths of Win32 Structured Exception Handling“, Matt Pietrik
http://www.microsoft.com/msj/0197/exception/exceptionfigs.htm#fig4

[3]
„Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI“, OpenRCE Library
http://www.openrce.org/articles/full_view/23

[4]
„Console Processes with Redirected Standard Handles“, MS Knowledgebase
http://support.microsoft.com/kb/190351

Attachments

1_API_Offsets.xls

2_Events_Filenames.txt

3_Imports.png

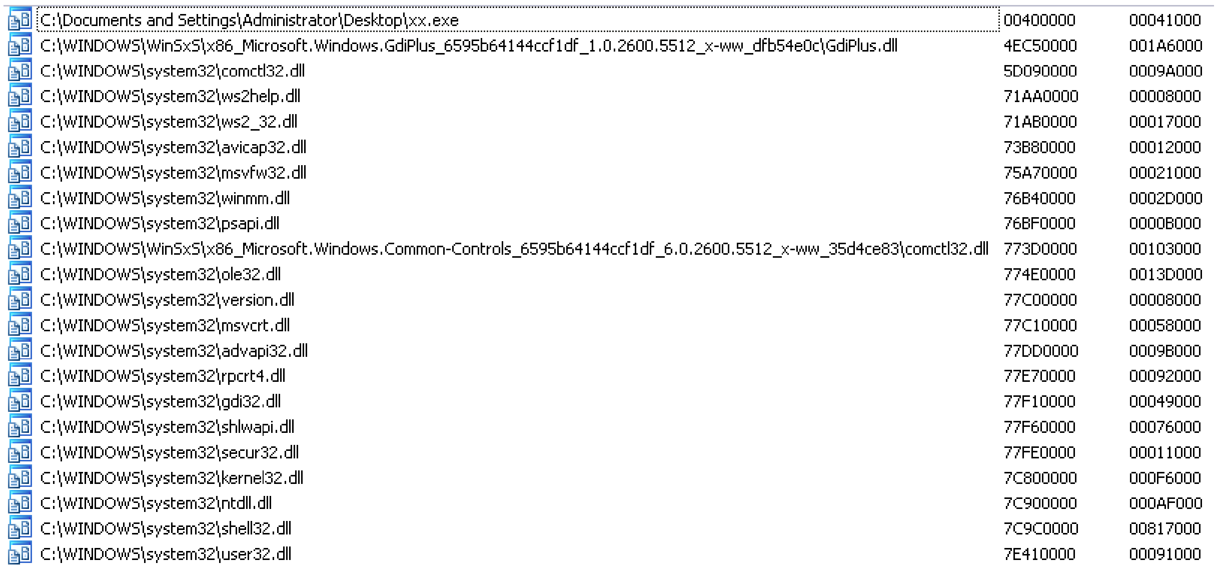
API Offsets

	0	4	8	C
00000000		process32next	process32first	createtoolhelp32snapshot
00000010	terminateprocess	createprocess	openprocess	getexitcodeprocess
00000020	getcurrentprocess	getcurrentprocessid	createpipe	duplicatehandle
00000030	openeventa	createthread	getfilesize	sleep
00000040	deletefilea	closehandle	setevent	readfile
00000050	createfile	gettemppath	getsystemdirectory	getwindowsdirectory
00000060	findclose	findfirstfilea	createevent	terminatethread
00000070	waitforsingleobject	getmodulefilename	writefile	resetevent
00000080	waitformultipleobjects	getshortpathname	createdirectorya	gettickcount
00000090	setfileattributesa	getfileattributes	MultiByteToWideChar	GetQueuedCompletionStatus
000000A0	PostQueuedCompletionStatus	CreateIoCompletion	PortGetDriveTypeA	GetVolumeInformationA
000000B0	SetFilePointer	GetLongPathNameA	FindNextFileA	EnterCriticalSection
000000C0	LeaveCriticalSection	DeleteCriticalSection	InitializeCriticalSection	GetExitCodeThread
000000D0	GetLastError	GetModuleFileNameExA	EnumProcessModules	CreateCaptureWindowA
000000E0	GetDriverDescriptionA	OpenProcessToken	AdjustTokenPrivileges	LookupPrivilegeValueA
000000f0	RegOpenKeyExA	RegQueryInfoKeyA	RegCloseKey	RegEnumValueA
00000100	timeSetEvent	timeKillEvent	CoUninitialize	CoCreateGuid
00000110	CoInitialize	SHFileOperationA	ShellExecuteA	WSAIoctl
00000120	socket	closesocket	bind	gethostbyname
00000130	WSAStartup	WSACleanup	htons	sendto
00000140	ntohs	WSAGetLastError	recvfrom	getsockname
00000150	shutdown	connect	send	recv
00000160	gethostname	DeleteObject	GetDIBits	GetObjectA
00000170	BitBlt	SelectObject	CreateCompatibleDC	CreateCompatibleBitmap
00000180	CreateDCA	DeleteDC	GdipGetImageEncoders	GdipGetImageEncodersSize
00000190	GdipDisposeImage	GdipSaveImageToFile	GdipLoadImageFromFile	GdiplusStartup
000001A0	GdiplusShutdown	GetWindowThreadProcessId	ExitWindowsEx	GetWindowTextA
000001B0	GetWindowLongA	GetWindow	GetDesktopWindow	PostMessageA
000001C0	FindWindowA	GetSystemMetrics	DestroyWindow	SendMessageA
000001D0	IsWindow	PostThreadMessageA	GetMessageA	sprintf
000001E0	strcmp	strstr	strcpy	strrchr
000001F0	strlen	strchr	memcmp	srand
00000200	rand	wcscmp	pow	_strupr
00000210	_strlwr	_strset	strcat	memset
00000220	malloc	free		

Events & Filenames

event1	AB8D393B-9177-440D-B3F8-1C1FE0CF9692
event2	A37340FD-F043-41e3-9C16-2F2632387199
	check for running mw instance
event3	83D33F3A-9482-446F-ABFF-7B69D58C1634
	check for fingerprint file
file1	FF24CF9A-EE48-4CDE-AC10-15D1CE2C272C
	fingerprint file
file2	A041D349-C68A-45C0-9081-536BC43BB0FF
	used in dat_file_createwrite, in check_datfile compared against fingerprint
file3	50030006-9D06-426F-936B-FFE0B81D5913
	file for storing captures made with create capture window
file4	C94A6BBB-4B51-4A8D-A49F-F184A27A972E
	file for storing screenshots, made with GDI DCA
temp1	FBCA78D4-024F-47E8-9851-C42C9626CC5A
	file for temporary use, deleted immediately
temp2	EF724F56-1CBE-4F84-A7AE-D31B2671B616
	file for temporary use, deleted immediately

Imports



C:\Documents and Settings\Administrator\Desktop\xx.exe	00400000	00041000
C:\WINDOWS\WinSxS\x86_Microsoft.Windows.GdiPlus_6595b64144ccf1df_1.0.2600.5512_x-ww_dfb54e0c\GdiPlus.dll	4EC50000	001A6000
C:\WINDOWS\system32\comctl32.dll	5D090000	0009A000
C:\WINDOWS\system32\ws2help.dll	71AA0000	00008000
C:\WINDOWS\system32\ws2_32.dll	71AB0000	00017000
C:\WINDOWS\system32\avicap32.dll	73B80000	00012000
C:\WINDOWS\system32\msvfw32.dll	75A70000	00021000
C:\WINDOWS\system32\winmm.dll	76B40000	0002D000
C:\WINDOWS\system32\psapi.dll	76BF0000	0000B000
C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll	773D0000	00103000
C:\WINDOWS\system32\ole32.dll	774E0000	0013D000
C:\WINDOWS\system32\version.dll	77C00000	00008000
C:\WINDOWS\system32\msvcrt.dll	77C10000	00058000
C:\WINDOWS\system32\advapi32.dll	77DD0000	0009B000
C:\WINDOWS\system32\rpcrt4.dll	77E70000	00092000
C:\WINDOWS\system32\gdi32.dll	77F10000	00049000
C:\WINDOWS\system32\shlwapi.dll	77F60000	00076000
C:\WINDOWS\system32\secur32.dll	77FE0000	00011000
C:\WINDOWS\system32\kernel32.dll	7C800000	000F6000
C:\WINDOWS\system32\ntdll.dll	7C900000	000AF000
C:\WINDOWS\system32\shell32.dll	7C9C0000	00817000
C:\WINDOWS\system32\user32.dll	7E410000	00091000