



Code Breakers Journal

© The CodeBreakers-Journal, Vol. 1, No. 1 (2004)
<http://www.CodeBreakers-Journal.com>

Processless Applications - Remotethreads on Microsoft* Windows 2000, XP and 2003

Thomas Kruse
Universitas Virtualis

Abstract

The shown technique is able to run on all Windows operation systems. In order to avoid virus creation on it's best, this technique is shown for W2K/XP/2K3 only. NT4 systems doesnt know several of the used API's, also it is possible to rewrite them. Non NT-based systems need other techniques to detect the correct process to inject the code.

This essay was created while searching for new software protections to make "crackers life" even harder. Based on "WatchDog theory" - another way to protect applications - the idea is to create threads outside the main application which are able to continue workflow also if the main application terminates. This essay will show up a way to display a messagebox from process "Explorer.Exe", which is available on all OS.

The created application is "processless" in that way that the main application becomes terminated after creating the external thread.

The shown source code is in Microsoft Assembler style (MASM [3]).

Keywords: Microsoft Operating Systems, Software Protection, Assembly Programming, System Internals

*The author **Thomas Kruse** has his main research focus on operating-system independent code optimising and software-protection development. He is Associate Editor of Assembly-Programming-Journal and CodeBreakers-Journal.*

I. Introduction

Before we start to create our application, we need some basic knowledge about processes and threads. And how they interact.

Whenever an application becomes executed, it creates a process and it's main thread. This application is able to run until:

- it's process becomes terminated
- the number of threads of it's process becomes NULL

The last term will show up a new point: we are able to create more then one thread for an application. And this technique is used for WatchDog protected applications.

System processes for example have many threads running inside one process. Also if one thread terminates, the process is still running. And in addition to this, new threads could be created while the process is running.

But there is another point which we need to inspect: While debugging[2] an application which creates more then one thread (e.g. WatchDog protected app), it is possible to trace the thread assembler code! The reason for this is simple:

- each created thread is allocating memory inside the calling process memory area
- the calling process memory area is available for reading

New memory could be given by using VirtualAlloc[1] API from KERNEL32.DLL, which is available on all OS. If we want to make "crackers life" harder, we have to leave our calling process memory area and create threads outside our application. This could be realized by using VirtualAllocEx[1].

II. General Idea

The idea to create a "better protected" application is based on the given leaks shown before. We need some name conventions to understand the workflow:

- *carrier application*: application which will be executed. It creates the remote thread, inject the code and terminates itself.
- *target application*: application which recives the new thread. It's avail if the carrier starts.

Lets have a look at the timescheduled workflow:

carrier application	target application
-----	-----
- becomes executed	- is already running
- searches the target application	
- grant access to target application	- return access rights
- allocates memory in target area	- return memory address
- inject code in target memory area	
- create remote thread in target app	- increase threads
- resume remote thread	- new thread executes*
- terminate itself**	

* When the thread becomes executed, it contains code, but didn't know correct API function calls. There are two ways to avoid this leak:

- resolve the function calls via carrier application
- resolve the function calls via target application

** The carrier application and it's threads become eliminated. But the created thread belongs to the target application. In case of that, this thread isn't affected while terminating the carrier application!

III. Resolving API calls

This topic sounds hard. But it isn't. Again we need to know a technique which NT-based operating systems use to "sort" dynamic link libraries in memory.

When a remote thread becomes executed, it is able to access the Process Environment Block (PEB) of it's main process. In our case, it is able to access the PEB of our target application. And each PEB contains entry's about:

- InLoadOrderModuleList
- InMemoryOrderModuleList
- InInitializationOrderModuleList

The first two named contain the application itself as first module, followed by needed DLL's (and accessible ones). The last is more interesting.

NT-based operation systems need the NTDLL.DLL. This DLL is the first entry in InInitializationOrderModuleList. And the next module is; KERNEL32.DLL! Exactly what we need, because this library contains the API functions for two important calls: GetProcAddress[1] and LoadLibraryA[1]. By resolving these two calls we are able to import what ever we want to. But where to store the resolved addresses?

IV. Preparing memory areas

We were able to resolve needed API calls, but we need to find a way to store them. One way is to store them on stack, another to create a separate memory area for the data of our thread. But by using the second technique, we need to tell our remote thread where to find the data. Lets have a closer look to API function CreateRemoteThread[1]:

```
HANDLE CreateRemoteThread(  
    HANDLE hProcess,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    DWORD dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter,  
    DWORD dwCreationFlags,  
    LPDWORD lpThreadId  
);
```

- *lpStartAddress*: Pointer to the application-defined function of type LPTHREAD_START_ROUTINE to be executed by the thread and represents the starting address of the thread in the remote process. The function must exist in the remote process.
- *lpParameter*: Specifies a single 32-bit value passed to the thread function.

If our thread becomes executed, *lpParameter* is still on stack. It is available via following codepart of our injected threadcode:

```
; -- access the given parameter from Create(Remote)Thread  
mov    eax,dword ptr [ebp+0Ch]    ; get parameter  
push  eax                        ; store this value on stack
```

We are able to split data and code in two memory areas. We simply use the API VirtualAllocEx twice:

- create data area in target application
- store the memory address, we need it as lpParameter for CreateRemoteThread
- inject data in data area
- create code area in target application
- store the memory address, we need it as lpStartAddress
- inject code in code area

- create the remote thread in suspended mode

By using two memory areas - for code and data - we were able to create address independent code during development! We are leaving the carrier application process memory range and have - during startup of our thread in target - no idea about address ranges in our target application process. And by using this technique, the code is target independent, regardless which application is affected.

V. Thread Data Part

The data part is created via `VirtualAllocEx` and `WriteProcessMemory[1]` inside the process memory range of our target application. By using `VirtualAllocEx`, we were able to name the target process within the first parameter of this API call. While debugging the carrier application, this memory area isn't easy to trace, because it belongs to another process and the debugger won't know this memory areas! For our example, we use the following data part:

```
.data
_sizeofInject      dd offset _inject_end - offset _inject
_sizeofData        dd offset _ENDOFDATA - offset _dwThreadID
_dwThreadID        dd 0
_dwThreadHandle    dd 0
_lpCode            dd 0
_lpData            dd 0
_lpRsrc            dd 0
_dwExitCode        dd 0
_Kernel32          dd 0
_User32            dd 0
_GetProcAddress   dd 0
_GetExitCodeThread dd 0
_ExitThread        dd 0
_szMessageBoxA     db "MessageBoxA",0
_szExitThread      db "ExitThread",0
_szExitCodeThread  db "GetExitCodeThread",0
_szText            db "Hello from Thread!",0
_szCaption         db "Thread:",0
_ENDOFDATA         dd 0
```

The first two and the last shown entry were NOT implemented! They only describe the size to inject. The data part to inject starts with `_dwThreadID` and ends up with `_szCaption` as last entry.

The data part contains several placeholders for resolved API function calls and strings (for `MessageBox` call and additional API functions to resolve). There are several placeholders which are not used in our example, but this data part is prepared for further projects, too.

VI. Thread Code Part

We are able to imagine the thread code workflow by analyzing this given data part:

- resolve `GetProcAddress` API from `KERNEL32.DLL`
- resolve `GetExitCodeThread` API from `KERNEL32.DLL`
- resolve `ExitThread` API from `KERNEL32.DLL`
- resolve `MessageBoxA` API from `User32.DLL`
- prepare `MessageBoxA` parameters on stack
- show the messagebox
- prepare `GetExitCodeThread` parameter on stack (`_dwThreadHandle`)
- get the exitcode
- exit the thread (thread terminates, process continues!)

And again: In order to make "crackers life" even harder, we use several tricks to realize this code! The code shown in Section IX needs some "brain gymnastics", but it is documented and - as time goes by - easy to understand.

VII. Carrier Application

The prepared thread code is useless without the carrier application. Lets have a closer look on what to do:

- check the operating system (do not execute on 9X)
- resolve USER32.DLL base address for our thread (*)
- create a snapshot of running processes
- find the target application (explorer.exe)
- if not found, terminate with error message
- store target processID
- free snapshot
- grant access to target application via OpenProcess[1]
- store the target process handle
- create two memory areas via VirtualAllocEx
- create a remote thread in suspended mode
- inject the parts via WriteProcessMemory
- resume the remote thread to become executed via ResumeThread[1]
- close the target process handle
- terminate the carrier via ExitProcess[1]

* The reader who understand the shown thread code in Section IX might have seen, that the base address of USER32.DLL is not resolved by our thread code! It's up to you to change it.

Knowing this workflow we are able to create the carrier application general part. It contains needed data and included functions and is shown in Section X.

The code part shown in Section XI realises exactly what we want to do.

And this code could be used as general template to create such applications! If you want to use resources later on, simply create another memory region by using VirtualAllocEx and store the memory address inside the thread data (_lpRsrc).

VIII. Conclusions

The shown technique uses one target which executes the injected code. But it is also possible to:

- inject code in DIFFERENT target applications and communicate between these targets using the technique from carrier application to access a process
- WALK from one target application to another, destroying itself after inject needed parts to new target by using polymorphic code
- WALK from one target application to another, leaving itself intact and create "splitted" applications which communicate together
- INTERACT with still running carrier application as kind of "external" WatchDog from target application(s)
- all COMBINATIONS of shown examples above

In combination with already existing software protections it is possible to avoid "readable" thread data. By using polymorphic code and crypto algorithms the reversing approach becomes nearly impossible.

References

- [1] Microsoft Corporation, *Microsoft Developer Network*, <http://msdn.microsoft.com>
- [2] Yuschuk, O., *Olly Debugger*, <http://home.t-online.de/home/ollydbg>
- [3] Hutchenson, S., *MASM V8*, <http://www.masmforum.com>

IX. SourceCode Thread.inc

We use several structures for resolving the export address table from KERNEL32.DLL. These structures are changed into correct code during compilation and in case of that, we didn't need to make life harder than it is.

```
;- thread.inc (data and code)
.data
_sizeofInject      dd offset _inject_end - offset _inject
_sizeofData        dd offset _ENDOFDATA - offset _dwThreadID
_dwThreadID        dd 0
_dwThreadHandle    dd 0
_lpCode            dd 0
_lpData            dd 0
_lpRsrc            dd 0
_dwExitCode        dd 0
_Kernel32          dd 0
_User32            dd 0
_GetProcAddress   dd 0
_GetExitCodeThread dd 0
_ExitThread        dd 0
_szMessageBoxA     db "MessageBoxA",0
_szExitThread      db "ExitThread",0
_szExitCodeThread  db "GetExitCodeThread",0
_szText            db "Hello from Thread!",0
_szCaption         db "Thread:",0
_ENDOFDATA        dd 0

.code
_inject:           ; start offset of code to inject

; -- access the given parameter from Create(Remote)Thread
mov  eax,dword ptr [ebp+0Ch] ; get parameter
push eax           ; store this value on stack

; resolve the kernel32 base via PEB (NT-based!)
mov  eax,7FFDF00Ch ; pointer to PEB_LDR
mov  eax,[eax]
add  eax,1Ch       ; pointer to InInitOrder_First
mov  eax,[eax]     ; NTDLL.DLL InitOrder_Next
mov  eax,[eax]     ; K32.dll InitOrder_Next
add  eax,8h        ; pointer to base address
mov  eax,[eax]     ; get the base address
mov  ebx,[esp]     ; get base data
add  ebx,18h       ; kernel32 address place
mov  [ebx],eax     ; store kernel32 base
mov  ebx,eax       ; to ebx

; -- GetProcAddress in another style
push  edi          ; store edi
push  esi          ; store esi
push  ebp          ; store ebp
push  edx          ; store edx
mov  eax,ebx       ; kernel32
mov  ebp,eax       ; ebp becomes base address
```

```
assume    eax: ptr IMAGE_DOS_HEADER
add      eax,[eax].e_lfanew
assume    eax:ptr IMAGE_NT_HEADERS
mov      edi,[eax].OptionalHeader.DataDirectory[0].VirtualAddress
add      edi,ebp          ; add the base value
assume    edi:ptr IMAGE_EXPORT_DIRECTORY
mov      esi,[edi].AddressOfNames
add      esi,ebp          ; add the base value
xor      edx,edx          ; clear edx
assume    eax:nothing     ; avoid errors

_loopstart:
mov      eax,[esi]
add      eax,ebp          ; add the base value

cmp      word ptr [eax+0Ch], "ss"
jne      _nextloop
cmp      dword ptr [eax+04h], "Acor"
jne      _nextloop
cmp      dword ptr [eax+00h], "PteG"
jne      _nextloop
cmp      dword ptr [eax+08h], "erdd"
jne      _nextloop

mov      eax,[edi].AddressOfNameOrdinals
add      eax,ebp          ; add the base value
movzx   ebx,word ptr [edx*2+eax]; get ordinal

mov      eax,[edi].AddressOfFunctions
add      eax,ebp          ; add the base value
mov      ebx,[ebx*4+eax]
add      ebx,ebp          ; ebx holds value of function
jmp      _found          ; quit searching

_nextloop:
add      esi,4
inc      edx
cmp      edx,[edi].NumberOfNames
jne      _loopstart

_found:
assume    edi:nothing     ; avoiding errors
pop      edx              ; restore edx
pop      ebp              ; restore ebp
pop      esi              ; restore esi
pop      edi              ; restore edi

mov      eax,[esp]        ; get base data
add      eax,20h          ; GetProcAddress placeholder
mov      [eax],ebx        ; store API address

mov      eax,[esp]        ; init eax with base data

; -- resolve all needed API's for this thread
```

```

mov     ebx,[eax+18h]    ; base address kernel32
mov     edx,[eax+20h]    ; function GetProcAddress
push   eax              ; store base data on stack
add     dword ptr [esp],38h ; pointer to ExitThread String
push   ebx              ; the kernel32 base
call   edx              ; call GetProcAddress
xchg   eax,ebx
pop    eax              ; restore data base
add     eax,28h          ; placeholder of ExitThread
mov     dword ptr [eax],ebx ; store the resolved address
sub     eax,28h          ; restore the original data base
push   eax              ; store it on stack
mov     ebx,[eax+18h]    ; base address kernel32
mov     edx,[eax+20h]    ; function GetProcAddress
push   eax              ; store base data on stack
add     dword ptr [esp],43h ; GetExitCodeThread String
push   ebx              ; the kernel32 base
call   edx              ; call GetProcAddress
xchg   eax,ebx
pop    eax              ; restore data base
add     eax,24h          ; placeholder of ExitThread
mov     dword ptr [eax],ebx ; store the resolved address
sub     eax,24h          ; restore the original data base
push   eax              ; store it on stack
mov     ebx,[eax+1ch]    ; base address user32
mov     edx,[eax+20h]    ; function GetProcAddress
push   eax              ; store base data on stack
add     dword ptr [esp],2ch ; pointer to MessageBoxA String
push   ebx              ; the user32 base
call   edx              ; call GetProcAddress
xchg   eax,ebx
pop    eax              ; restore data base
add     eax,1ch          ; placeholder MessageBoxA (User32)
mov     dword ptr [eax],ebx ; store the resolved address
sub     eax,1ch          ; restore the original data base

; -- start visible "action" inside the created thread
push   eax              ; store data base
mov     ebx,eax          ; ebx becomes data base
mov     edx,[eax+1Ch]    ; function MessageBoxA
add     ebx,68h          ; pointer to caption
push   0                ; MB_OK
push   ebx              ; caption
sub     ebx,13h          ; pointer to text
push   ebx              ; text
push   0                ; handle
call   edx              ; MessageBoxA
pop    eax              ; restore data base/correct stack!

; -- final cleanup and destroying the created thread
push   eax              ; store data base
mov     ebx,eax
add     ebx,14h          ; pointer to ExitCode
push   ebx              ; address to recieve the ExitCode

```

```
sub    ebx,10h           ; pointer to ThreadHandle
push   dword ptr[ebx]
add    ebx,20h           ; pointer to GetExitCodeThread
call   dword ptr[ebx]    ; GetExitCodeThread
xchg   eax,ebx
pop    eax               ; restore data base
mov    ebx,eax
add    ebx,14h           ; pointer to ExitCode
push   dword ptr[ebx]    ; ExitCode for this thread
add    ebx,14h           ; pointer to ExitThread
call   dword ptr[ebx]    ; ExitThread
_inject_end:            ; end offset of code to inject
```

X. SourceCode inject.inc

```
; -- inject.inc
.386                ; minimum required CPU
.model flat,stdcall ; win 32 app
option casemap:none ; case sensitive

include windows.inc ; stuff we need

incboth macro incl  ; macro for less typing
    include    incl.inc
    includelib incl.lib
endm

incboth kernel32    ; for kernel32 API's
incboth user32      ; for user32 API's

include thread.inc  ; thread related data

.data
szErrorText        db "Process EXPLORER.EXE not found",0
szErrorOS          db "You need an NT-based OS!",0
szErrorCaption     db "Error occured:",0
szExplorer         db "explorer.EXE",0
szUser32           db "user32.dll",0

.data?
hInstance          dd ?      ; application instance handle
_myProcessID       dd ?      ; resolved process ID
_myProcessHandle   dd ?      ; resolved process handle
_BytesWritten      dd ?      ; for WriteProcessMemory
hSnapshot          dd ?      ; snapshot handle
myProcess          PROCESSENTRY32 <>
```

XI. SourceCode inject.asm

```
; -- inject.asm
include inject.inc           ; additional data and includes

.code
start:                       ; application entry point
    ; -- carrier application
    invoke GetModuleHandle,NULL ; get application module handle
    mov    hInstance,eax      ; and store it

    assume fs:nothing        ; needed for MASM
    mov    eax,fs:[18h]      ; get TEB self pointer
    mov    ebx,fs:[30h]     ; get PEB pointer

    .if    eax!=7FFDE000h || ebx!=7FFDF000h ; NT check
        invoke MessageBox,NULL,addr szErrorOS,addr szErrorCaption,MB_OK
        jmp    _exitApp
    .endif
    invoke LoadLibrary,addr szUser32
    mov    _User32,eax
    invoke FreeLibrary,_User32
    invoke CreateToolhelp32Snapshot, TH32CS_SNAPALL, 0
    mov    hSnapShot, eax      ; store the snapshot handle
    mov    myProcess.dwSize, sizeof myProcess
    invoke Process32First,hSnapShot, ADDR myProcess
    .while eax
        lea    eax,myProcess.szExeFile
        lea    esi,szExplorer ; the name to search
        invoke lstrcmpi,eax,esi ; compare both strings
        .if    eax==0         ; if the same...
            push myProcess.th32ProcessID
            pop  _myProcessID ; get the process ID and
            jmp    @F         ; leave search routine
        .else
            invoke Process32Next,hSnapShot, ADDR myProcess
        .endif
@@:
    .endw
    invoke CloseHandle,hSnapShot ; close snapshot handle

    mov    eax,_myProcessID ; get the process ID
    test   eax,eax          ; is there an ID
    jne    _goOn           ; if so, continue
    invoke MessageBox,NULL,addr szErrorText,\
        addr szErrorCaption, MB_OK
    jmp    _exitApp        ; leave application

_goOn:
    invoke OpenProcess,PROCESS_ALL_ACCESS,NULL,_myProcessID
    mov    _myProcessHandle,eax ; store the process handle

    ; -- create two mem parts: one for code, one for data
    invoke VirtualAllocEx,_myProcessHandle,NULL,1000h,MEM_COMMIT,\
```

```

                                PAGE_EXECUTE_READWRITE
mov     _lpCode,eax             ; store code base in thread data
invoke VirtualAllocEx,_myProcessHandle,NULL,100h,MEM_COMMIT,\
                                PAGE_EXECUTE_READWRITE
mov     _lpData,eax            ; store data base in thread data

; -- create the thread in suspended mode!
; -> the 4th (here: _lpData) parameter is important; it
;     guarantees an memory independent inject code...
invoke CreateRemoteThread,_myProcessHandle,NULL,NULL,_lpCode,\
                                _lpData, CREATE_SUSPENDED,\
                                addr _dwThreadID
mov     _dwThreadHandle,eax     ; store handle in thread data

; -- write code section
invoke WriteProcessMemory,_myProcessHandle,_lpCode,\
                                addr _inject, _sizeofInject,\
                                addr _BytesWritten

; -- write data section
invoke WriteProcessMemory,_myProcessHandle,_lpData,\
                                addr _dwThreadID, _sizeofData,\
                                addr _BytesWritten
invoke ResumeThread,_dwThreadHandle ; start the thread
invoke CloseHandle,_myProcessHandle ; close process handle
_exitApp:
invoke ExitProcess,NULL          ; exit application
end start                        ; end of application code section
```