Tricks of the Hackers: API Hooking and DLL Injection

Dr. Wolfgang Koch

Friedrich Schiller University Jena

Department of Mathematics and Computer Science

Jena, Germany

wolfgang.koch@uni-jena.de

API Hooking

2

Intercepting API calls is a mechanism for
testing
monitoring
and reverse engineering

as well as for altering the behavior of the operating system

or of 3rd party products,

without having their source code available.

API Hooking

3

Intercepting API calls is a mechanism for

altering the behavior of programs or of the operating system

widely used by hackers and other "bad guys"

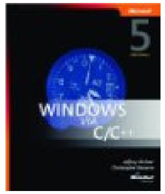
Literature Books

4

"The Windows-API Book":

Jeffrey Richter,
Christophe Nasarre

WINDOWS via C/C++
5th edition



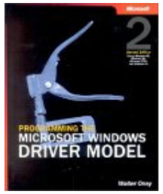
Redmond, Wash : Microsoft Press, 2008
ISBN-13: 978-0-7356-2424-5

820 p. + Companion content Web page

Literature Books

5

The WDM Bible:



Walter Oney
Programming the Microsoft Windows Driver Model
2nd edition

Redmond, Wash : Microsoft Press, 2003
ISBN: 0-7356-1803-8

846 p. + CD-ROM

Literature

6

Ivo Ivanov: API hooking revealed, 2002

http://www.codeproject.com/KB/system/hooksys.aspx

Robert Kuster: Three Ways to Inject Your Code into Another Process, 2003

http://www.codeproject.com/KB/threads/winspy.aspx

Seung-Woo Kim - Intel® Software Network: Intercepting System API Calls, 2004

http://software.intel.com/en-us/articles/intercepting-system-api-calls/

Literature

7

Anton Bassov:

Process-wide API spying - an ultimate hack, 2004

http://www.codeproject.com/KB/system/api_spying_hack.aspx

Kernel-mode API spying - an ultimate hack, 2005

http://www.codeproject.com/KB/system/kernelspying.aspx

Newsgroups:

comp.os.ms-windows.programmer.nt.kernel-mode
microsoft.public.development.device.drivers

Literature Executable File Format

8

Microsoft MSDN **man pages** and "**white papers**":

http://msdn.microsoft.com/library

An In-Depth Look into the Win32 Portable Executable File Format, Matt Pietrek, MSDN Magazine, Feb. 2002

http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

Microsoft Portable Executable and Common Object File Format Specification, Revision 8.0 - May 2006

http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

API Hooking

API - **Application Programming Interface**

Wikipedia:

API - is an **interface** that defines the ways by which an application program may **request services** from **libraries** and/or **operating systems**.

An API determines the vocabulary and **calling conventions** the programmer should employ to use the services. It may include specifications for **routines**, **data structures**, object classes and protocols …

API Hooking

API Routine Call:

```
...
push Parameter_2
push Parameter_1
call <Addr_of_API_Routine>
mov  eax, Result
...
```

```
//API Routine (in DLL)
push ebp
mov  esp, ebp
...  // service
ret
```

API Hooking

API Routine Call - hooked:

```
...
push Parameter_2
push Parameter_1
call <Addr_of_new_Routine>
mov  eax, Result
...
```

```
//proxy function
push ebp
... // other stuff
ret
```

```
//API Routine (in DLL)
push ebp
mov  esp, ebp
...  // service
ret
```

We would have to overwrite every call to the API routine in the code

API Hooking

API Routine Call – hooked (2):

```
...
push Parameter_2
push Parameter_1
call <Addr_of_new_Routine>
mov  eax, Result
...
```

```
//proxy function
push ebp
... // pre proc.
call original API
... // post proc.
ret
```

```
//API Routine (in DLL)
push ebp
mov  esp, ebp
...  // service
ret
```

- Parameters of the API routine
- Win32 API: `_stdcall - ret n`

API Hooking

13

API Routine Call – hooked

different way – overwriting code

```
...
push Parameter_2
push Parameter_1
call <Addr_of_API_Routine>
mov  eax, Result
...
```

```
//proxy function
push ebp
... // other stuff
ret
```

```
//API Routine (in DLL)
jmp <Addr_of_proxy_
     Routine>
... // service
ret
```

Difficulties when calling or jumping to the original API routine.

API Hooking

14

Hooking API Routine Calls

`call <Addr_of_API_Routine>      (0xE8)`

we would have to overwrite **every** call to the API routine in the code

In Windows executables **indirect calls** are applied:

`call ind(Ptr_To_Addr_of_API_Routine)`

The Pointer points to the address of the API Routine in the **Import Address Table** (IAT), where the addresses of all functions, imported by the module, are stored.

API Hooking

15

In Windows executables indirect calls are applied:

```
call ind(Ptr_To_Addr_of_API_Routine)
                              (0xFF,0x15)
```

Reason:

The actual address of the API Routine is not known at compile time.

The Import Address Table is filled in, when a Library (a DLL) is loaded.

Now we can change the addresses in the IAT (only one per API routine) to our proxy functions.

API Hooking Locating the IAT

16

see Portable Executable File Format:

```
IMAGE_DOS_HEADER * dosheader = (IMAGE_DOS_HEADER *) hMod;

IMAGE_OPTIONAL_HEADER * opthdr = (IMAGE_OPTIONAL_HEADER *)
          ((BYTE*)hMod + dosheader->e_lfanew + 24);

IMAGE_IMPORT_DESCRIPTOR * descriptor =
     (IMAGE_IMPORT_DESCRIPTOR *) ((BYTE*) hMod + opthdr->
      DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].
      VirtualAddress );

while(descriptor->FirstThunk){
  char * dllname = (char *)((BYTE*)hMod + descriptor->Name);
```

API Hooking IAT_Entry_Addresses

17

```
while(descriptor->FirstThunk){
  char * dllname = (char *)((BYTE*)hMod + descriptor->Name);
  IMAGE_THUNK_DATA * thunk = (IMAGE_THUNK_DATA *)
     ((BYTE*) hMod + descriptor->OriginalFirstThunk);

  int x=0;
  while(thunk->u1.Function) {
   char  * functionname = (char*) ((BYTE*) hMod +
          (DWORD)thunk->u1.AddressOfData + 2);
   DWORD * IATentryaddress = (DWORD*) ((BYTE*) hMod +
          descriptor->FirstThunk) + x;
   x++; thunk++; }
descriptor++; }
```

API Hooking IAT_Entry_Addresses

18

```
IATentryaddress = (DWORD*)
   ((BYTE*) hMod + descriptor->FirstThunk) + x;
```

Now we can overwrite IAT entries of the target module with the addresses of our user-defined proxy functions.

Each IAT entry replacement normally requires a separate proxy function - a proxy function must know which particular API function it replaces so that it can invoke the original callee.

Anton Bassov proposed the following method

API Hooking

19

Anton Bassov proposed the following method

4 (unique) functions, one structure :

`ProxyProlog(), ProxyEpilog()` - Assembler

`Prolog(), Epilog()` - normal C code

```
struct RelocatedFunction{
  DWORD  proxyptr;
  DWORD  functionptr;
  char * dllname;
  char * functionname;
};
```

API Hooking

20

4 (unique) functions, one structure,

22 bytes of allocated (executable) memory per IAT entry:

0xFF15	Addr_RelFkt	struct RelocatedFunction
2	4	16 Bytes

`0xFF15 Addr_RelFkt -`

indirect call: `call ind(Addr_RelFkt)`

i.e. calls `RelocatedFunction.proxyptr`

Trick: The return address of this proxy function – on top of the stack – is the address of our `struct RelocatedFunction` !!!

API Hooking Modifying IAT Entries

21

```
while(thunk->u1.Function) {
  char  * functionname = ... ;
  DWORD * IATentryaddress = ... + x;

  if ( alter(functionname) ){
     <allocate memory>
     <fill in memory and store original API address>
     <modify  *IATentryaddress >
  }

  x++; thunk++;
}
```

API Hooking Modifying IAT Entries

22

```
uchar * mptr = malloc(22); // <allocate memory>

// <fill in memory and store original API address>
struct  RelocatedFunction * reloc =
        (struct RelocatedFunction *) (mptr+6);

 reloc->proxyptr     = (DWORD) ProxyProlog;
 reloc->functionptr  = *IATentryaddress;   // orig.
 reloc->functionname = functionname;
 reloc->dllname      = dllname;

 mptr[0]= 0xFF; mptr[1]= 0x15;   // JMP ind
 memmove( &mptr[2], &reloc, 4);
```

API Hooking Modifying IAT Entries

23

```
uchar * mptr = malloc(22);
. . .
// < modify  *IATentryaddress >

DWORD  byteswritten;

WriteProcessMemory( GetCurrentProcess(),
                    IATentryaddress,
                    &mptr, 4, &byteswritten);
```

Usually the IAT is writeable (it is filled in, when a Library is loaded), but not the Export Directory.

API Hooking Modifying IAT Entries

24

Usually the IAT is writeable …

```
if (! WriteProcessMemory( GetCurrentProcess(),
      IATentryaddress, &mptr, 4, NULL)) {
  DWORD dwOldProt;
  if (VirtualProtect(IATentryaddress, 4,
             PAGE_READWRITE, &dwOldProt)) {
     WriteProcessMemory( GetCurrentProcess(),
         IATentryaddress, &mptr, 4, NULL);
     VirtualProtect(IATentryaddress, 4, dwOldProt,
                    &dwOldProt);  }
}
```

API Hooking ProxyProlog

25

ProxyProlog() saves registers and flags and calls Prolog().
Then it restores the registers and flags and calls (in fact: returns to) the original API routine (Prolog modifies the return address).

```
__declspec(naked) void ProxyProlog()
{
 _asm{ push eax; push ebx; push ecx; push edx }

 _asm{ mov ebx,esp; pushf; add ebx,16 }
 _asm{ push ebx; call Prolog; pop ebx; popf }

 _asm ( pop edx; pop ecx; pop ebx; pop eax; ret)
}
```

API Hooking ProxyProlog

26

ProxyProlog() saves registers and flags and calls Prolog().

```
__declspec(naked) void ProxyProlog()
```

creates pure function code without a stack frame
– intended for assembler code.

I work with cygwin and gcc, there is no `__declspec(naked)`.
I just define `void ProxyProlog()`
and instad of `reloc->proxyptr = (uint) ProxyProlog;`
I have

```
reloc->proxyptr = (uint)((BYTE*)ProxyProlog +3);
```

API Hooking ProxyProlog

27

ProxyProlog() saves registers and flags and calls Prolog().

```
 _asm{ mov ebx,esp; pushf; add ebx,16 }
 _asm{ push ebx; call Prolog; pop ebx; popf }
```

Prolog is defined as `void Prolog(DWORD * stackptr);`
– it has one parameter –
`ebx` , which holds the address of the top of stack
(the contents of `esp`) just before saving the registers –
i.e. the address of the return address of ProxyProlog().

As you remember this in fact is the address of the corresponding `struct RelocatedFunction` which contains all necessary data.

API Hooking Prolog

28

Prolog() does the preprocessing part to the hooked routines
– for example monitoring all API calls.
It has access to the actual parameters of the routine
as well as to the DLL name and the function name.

If necessary Prolog() prepares for the call of the
post-processing part – ProxyEpilog() and Epilog()

Prolog() returns to ProxyProlog(), which in turn has no valid return address. Prolog replaces it with the original address of the hooked routine – so return is just a jump to this routine.

API Hooking Prolog

29

Prolog() has access to the actual parameters of the routine as well as to the DLL name and the function name.

It has one parameter – the address of the return address of ProxyProlog(), which in fact is the address of the companion `struct RelocatedFunction`.

```
void Prolog(DWORD * stackptr) {
  struct RelocatedFunction *reloc_prol, *reloc_epil;
  reloc_prol = (struct RelocatedFunction *) *stackptr;

  // use  reloc_prol->functionname,  reloc_prol->dllname);
```

API Hooking Prolog

30

```
Stack: |     ...     |
       +-------------+        void Prolog(DWORD * stackptr);
       |     EAX     |
       +-------------+
       | Ret ProxyPr |     <==== stackptr
       +-------------+
       |  Ret  API   |
       +-------------+
       |   Param 1   |     // DWORD param1 = *(stackptr+2);
       +-------------+
       |   Param 2   |     // DWORD param2 = *(stackptr+3);
       +-------------+
  high |     ...     |
```

API Hooking Prolog

31

```
Stack: |     ...     |
       +-------------+        void Prolog(DWORD * stackptr);
       |     EAX     |
       +-------------+
       | Ret ProxyPr |     <==== stackptr
       +-------------+
       |  Ret  API   |
       +-------------+
       |   Param 1   |
       +-------------+
       |   Param 2   |
       +-------------+
  high |     ...     |
```

Prolog replaces the return address of ProxyProlog with the original address of the hooked routine → jmp to API

```
*stackptr =
        reloc_prol->functionptr;
```

If there is no Epilog, API returns regularly using `RET API`

API Hooking Prolog

32

… API returns regularly using `RET API`.

If necessary Prolog() prepares for the call of the post-processing part – ProxyEpilog() and Epilog()

– it overwrites `RET API` with the address of a new `struct RelocatedFunction` (in fact of all the 22 bytes) which works the same way as with ProxyProlog and holds data, important for Epilog().

Epilog() then must repair the return address `RET API` on the stack.

API Hooking Prolog

33

Prolog() overwrites `RET API` with the address of a new `struct RelocatedFunction.`

We cannot re-use the original `struct`,
since we are in a preemptive, multithreaded environment
– several calls to the same routine may be pending.

A. Bassov uses **thread local storage** (TLS)
- and has a lot of trouble with it (scanning all DLLs, DLL injection)

My method (using malloc) is much easier and just as good.

API Hooking Prolog

34

Prolog() overwrites `RET API` with the address of a new `struct`

```
uchar * mptr = malloc(22);

reloc_epil   = (struct RelocatedFunction *) (mptr+6);

reloc_epil->proxyptr     = (DWORD) ProxyEpilog;
reloc_epil->functionname = reloc_prol->functionname;
reloc_epil->dllname      = reloc_prol->dllname;
reloc_epil->functionptr  = *(stackptr+1);  // RET API

mptr[0]= 0xFF; mptr[1]= 0x15;
memmove( &mptr[2], &reloc_epil, 4);

*(stackptr+1) = (DWORD) mptr;
```

API Hooking Epilog

35

ProxyEpilog() saves registers and flags and calls Epilog().
Then it restores the registers and flags and returns to the caller of the original API routine (Epilog repairs the return address).

```
__declspec(naked) void ProxyEpilog()
{
 _asm{ push eax; push ebx; push ecx; push edx }

 _asm{ mov ebx,esp; pushf; add ebx,16 }
 _asm{ push ebx; call Epilog; pop ebx; popf }

 _asm ( pop edx; pop ecx; pop ebx; pop eax; ret)
}
```

API Hooking Epilog

36

Epilog() does the post-processing part to the hooked routines.

```
void Epilog(DWORD * stackptr);
```

It has access to the parameters of the routine
– though they are of no much use now,
to the result of the original routine (in `eax` – on the stack)
and, again to the DLL name and the function name.

```
DWORD result = *(stackptr-1);
DWORD Param1 = *(stackptr+1);  // +1 now, not +2
```

Epilog repairs the return address of ProxyEpilog()

API Hooking Epilog

37

Epilog repairs the return address of ProxyEpilog()
and frees the memory, used for `struct RelocatedFunction`.

```
struct RelocatedFunction * reloc_epil;

reloc_epil = (struct RelocatedFunction *) *stackptr;

*stackptr = reloc_epil->functionptr;

free((uchar *)reloc_epil -6);
```

API Hooking Multiple Modules

38

In order to intercept every call to certain API routines
on a process-wide base,
we have to walk through all modules that are currently
loaded into the address space of the target process,
and, in each loaded module, overwrite the IAT entries

```
while ( modules ){
  HANDLE hMod = GetModuleHandle(DLL_Name);
  ModuleScanIAT(hMod); }
```

This doesn´t work, if a module is dynamically loaded afterwards,
then we must overwrite the Export Directory of the API module.

API Hooking DLL

39

We put Prolog() etc. in a DLL, which we **inject** into the
target process.
When the DLL is loaded, its `DLLMain` Routine is called
with `reason = DLL_PROCESS_ATTACH`.

```
int WINAPI DllMain(HANDLE h, DWORD reason, void *foo)
{
  if (reason == DLL_PROCESS_ATTACH)
  {
     HANDLE hMod = GetModuleHandle(NULL);
     ModuleScanIAT(hMod);
     < possibly scans of more DLLs and Export Dirs >
} }
```

DLL Injection 4 Ways

40

In order to apply our hooking tool to a (possibly yet running)
target process, we put the tool in a DLL and **inject** the DLL
into that process.

There are different possible ways of injection:

- Using the Registry
- Using Windows Hooks
- Using `CreateRemoteThread()` the way I used
- Using `CreateProcess( …, CREATE_SUSPENDED, … )`
  for a process we launch see A. Bassow

DLL Injection Registry

41

There are different possible ways of injection

Using the Registry

if in the registry key
`HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\`
the value `AppInit_DLLs` contains the name of one or more DLLs, these DLLs are loaded by each Windows-based application running within the current logon session (`LoadAppInit_DLLs = 1`)

- Injection only into processes that use USER32.DLL
- Injection into every single GUI application, regardless we want it

DLL Injection Windows Hooks

42

There are different possible ways of injection

Using Windows Hooks

The primary role of windows hooks is to monitor the message traffic of some thread.

If the hooked thread belongs to another process, the hook procedure must reside in a DLL. The system then maps the entire DLL containing the hook procedure into the address space of the hooked thread.

I never used Windows Hooks.

DLL Injection CreateRemoteThread

43

There are different possible ways of injection

Using `CreateRemoteThread()`

Any process can load a DLL dynamically using `LoadLibrary`. But, how do we force an external process to call this function?

Answer: `CreateRemoteThread()`

It was originally designed for debuggers and such tools.

It is identical to `CreateThread()` except for one additional parameter: `hProcess`, the process that will own the new thread

DLL Injection CreateRemoteThread

44

`CreateRemoteThread()` is identical to `CreateThread()` except for one additional parameter: `HANDLE hProcess`

We get this handle using `OpenProcess()` with the ProcessID of the target process.

The addresses `LPTHREAD_START_ROUTINE lpStartAddress` and `LPVOID lpParameter` are addresses in the target process now, we have to copy the **thread code** and additional data there using

`VirtualAllocEx(hProcess, …)` and
`WriteProcessMemory(hProcess, …)`.

DLL Injection CreateRemoteThread

45

We have to copy the thread code and additional data there …

```
typedef HINSTANCE (*fpLoadLibrary) (char*);

typedef struct _INJECTSTRUCT
{
  fpLoadLibrary   LoadLibrary;
  char path[120];
} INJECTSTRUCT;
```

`INJECTSTRUCT` will contain additional data:

a function pointer for `LoadLibrary()`

and a string that holds the path of the DLL we want to inject

DLL Injection CreateRemoteThread

46

Thread code:

```
DWORD WINAPI threadcode(LPVOID addr)
{
  INJECTSTRUCT *is = addr;

  is->LoadLibrary(is->path);
  return 0;
}

void threadend()
{
}   // for computing the size of the thread code
```

DLL Injection CreateRemoteThread

47

```
int main()
{
  HANDLE  hProc, hThread;
  LPVOID  start, thread;
  DWORD   funcsize, pid;
  INJECTSTRUCT is;

  is.LoadLibrary = (fpLoadLibrary) GetProcAddress(
                        GetModuleHandle("Kernel32"),
                                      "LoadLibraryA");

  strcpy(is.path, "Hook.dll");
```

DLL Injection CreateRemoteThread

48

```
funcsize = (DWORD)threadend -(DWORD)threadcode;

printf("\n PID: "); scanf("%d", (int *) &pid);

hProc = OpenProcess(PROCESS_ALL_ACCESS,
                              FALSE, pid);
if(!hProc){ printf(" ... %d", pid); return 1; }

start = VirtualAllocEx(hProc, NULL, funcsize +
          sizeof(INJECTSTRUCT),      // 4096
          MEM_RESERVE | MEM_COMMIT,
          PAGE_EXECUTE_READWRITE);
```

DLL Injection CreateRemoteThread

49

```
WriteProcessMemory(hProc, start, &is,
                  sizeof(INJECTSTRUCT), NULL);

thread = (LPVOID)      // new Addr of Thread
         ((DWORD)start + sizeof(INJECTSTRUCT));

WriteProcessMemory(hProc, thread, threadcode,
                          funcsize, NULL);

hThread = CreateRemoteThread(hProc, NULL, 0,
                      thread, start, 0, NULL);
```

DLL Injection CreateRemoteThread

50

```
  hThread = CreateRemoteThread(hProc, ...);

  WaitForSingleObject(hThread, INFINITE);

  VirtualFreeEx(hProc, start, 0, MEM_RELEASE);

  CloseHandle(hThread);
  CloseHandle(hProc);
  return 0;
}
```

we can find the PID using the console program `tasklist`

DLL Injection DebugPrivilege

51

We cannot inject code into every other process, unless the injecting process has “DebugPriviledge”

```
#include <tlhelp32.h>

void EnableDebugPrivilege()
{
  TOKEN_PRIVILEGES priv;
  HANDLE           hToken;
  LUID             luid;   //Locally Unique Identifier

  OpenProcessToken(GetCurrentProcess(),
         TOKEN_ADJUST_PRIVILEGES, &hToken);
```

DLL Injection DebugPrivilege

52

```
{
  OpenProcessToken(GetCurrentProcess(),
          TOKEN_ADJUST_PRIVILEGES, &hToken);

  LookupPrivilegeValue(NULL, "seDebugPrivilege", &luid);

  priv.PrivilegeCount = 1;
  priv.Privileges[0].Luid = luid;
  priv.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

  AdjustTokenPrivileges(hToken, FALSE, &priv, 0,0,0);

  CloseHandle(hToken);
}
```