Kernel-22

Mike McCarl
ICSA Labs
1000 Bent Creek Blvd., Suite 200
Mechanicsburg, PA 17050

ICSA Labs, an independent division of Verizon Business

The idea of spoofing DLLs is not new. It is a technique used for analysis tools as well as malicious programs. By offering the same set of functions as another DLL, a calling program can unknowingly provide the means to load and execute alternate code, which can then completely change the actions of a program for good or bad purposes. In the world of malware analysis, a handy use for spoofing is to simply log each time a function in a DLL is called, which can help determine what a malicious program is attempting. But there is more than one way to spoof a DLL, and some DLLs are easier to spoof than others.

A basic spoofing technique is to offer each of the functions of another DLL – completely implementing the behavior for each function. With this technique, a C header file or other documentation for the functions of the DLL is usually available. A DLL named the same as the DLL that is to be spoofed is then built from scratch.

Consider a DLL that implements the functions of ws2_32.dll. The header file "winsock2.h", available in the Microsoft Platform SDK, provides the function prototypes and data structures for the functions; therefore it is just a matter of inserting the bodies for each function. Listing 1 shows excerpts from the source files used to build such a DLL. The file ws2_32.c contains the DllMain function as well as each of the functions declared in the winsock2.h file. The file ws2_32.def defines the functions to be exported and the ordinals for those exported functions. By giving our new DLL the name "ws2_32.dll" and placing it in a directory that is at the beginning of the search path, applications that use ws2_32.dll will now use the spoofing module instead.

ws2_32.c

```
#include <stdio.h>
#define _WIN32_WINNT 0x0501
#define WINSOCK_API_LINKAGE
#include <winsock2.h>
#include <ws2tcpip.h>
#define WSAAPI FAR PASCAL
#define WSPAPI WSAAPI
        .
        .
        .
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL,
    DWORD fdwReason,
    LPVOID lpvReserved)
{
        .
        .
        .
}

SOCKET PASCAL FAR accept (
    IN SOCKET s,
    OUT struct sockaddr FAR *addr,
    IN OUT int FAR *addrlen)
{
     // implementation of accept
}
        .
        .
        .
       etc.
```

ws2_32.def

```
; Module definition for wsock32
LIBRARY ws2_32.dll
EXPORTS
    accept @1
    bind @2
    closesocket @3
    connect @4
    getpeername @5
    getsockname @6
    getsockopt @7
    htonl @8
    htons @9
    ioctlsocket @10
    inet_addr @11
    inet_ntoa @12
    listen @13
    ntohl @14
    ntohs @15
    recv @16
    recvfrom @17
    select @18
         .
         .
         .
        etc.
```

Listing 1: Implementation of ws2_32

But there are plenty of drawbacks to this technique. A header file or documentation needs to be available in order to determine what to implement. Also, every function needs to be fully implemented from scratch – even to just "fake" functionality. It would be

ws2_32.c

```
#include <stdio.h>
#define _WIN32_WINNT 0x0501
#define WINSOCK_API_LINKAGE
#include <winsock2.h>
#include <ws2tcpip.h>
#define WSAAPI FAR PASCAL
#define WSPAPI WSAAPI

HMODULE ghModule;
FILE *gpLog;

BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD fdwReason,LPVOID lpvReserved)
{
    return TRUE;
}

HMODULE LoadRealDll ()
{
        if (!ghModule) {
                ghModule = LoadLibrary ("\\windows\\system32\\ws2_32.dll");
                gpLog = fopen ("ws2_32.log", "a");
                fprintf (gpLog, "*********** New Dll Instance ***********\n");
        }
        return ghModule;
}

FARPROC GetRealProcAddress (char *pFunction)
{
        if (LoadRealDll ()) {
                if (gpLog) {
                        fprintf (gpLog, "pid: %d tid: %d Entering %s\n",
                                 GetCurrentProcessId (),
                                 GetCurrentThreadId (),
                                 pFunction);
                        fflush (gpLog);
                }
                return GetProcAddress (ghModule, pFunction);
        }
        return NULL;
}

SOCKET PASCAL FAR accept (
    IN SOCKET s,
    OUT struct sockaddr FAR *addr,
    IN OUT int FAR *addrlen)
{
        FARPROC pProc;
        SOCKET rc = -1;

        pProc = GetRealProcAddress ("accept");
        if (pProc) rc =(pProc)(s, addr, addrlen);

        fprintf (gpLog, pid: %d tid: %d Leaving %s rc: %d\n",
                 GetCurrentProcessId (),
                 GetCurrentThreadId (),
                 "accept",
                 rc);
        fflush (gpLog);

        return rc;
}
```

Listing 2: Alternate implementation of ws2_32

nicer if we could leverage the existing DLL so that only the functions we wish to modify need to be re-coded. Fortunately, there is a relatively simple way to accomplish this, too. The key is to load and access the "real" version of the DLL being spoofed. Then, in

the implementation, call the appropriate routine in the real module, returning those values to the caller. Listing 2 shows what source files for such an implementation might look like.

In this implementation, functions LoadRealDll and GetRealProcAddress have been added along with 2 global variables gpLog and ghModule. LoadRealDll calls the LoadLibrary function to load the real ws2_32.dll (from \windows\system32) and saves the instance handle in the global variable ghModule. Once ghModule is set, LoadRealDll won't call LoadLibrary again. LoadRealDll also opens a logging file saving the FILE pointer in gpLog. The GetRealProcAddress function calls LoadRealDll to ensure the real ws2_32.dll has been loaded, and then calls GetProcAddress to resolve the address of the passed function name within the real ws2_32.dll. The accept function (and all other functions) just have to call GetRealProcAddress and can call just about anything else to augment the behavior of the function. In this case, it calls the real accept function then writes some basic information to the log file. Since all functions are now implemented in essentially the same manner, it is much easier to implement all the functions of ws2_32.dll.

Although we now have a simpler method for implementing the functions, it still relies on having a header file that provides the signature of each function. This information is essential in order to construct the call to the "real" functions. In practice, though, if you want to log the calls to an arbitrary DLL, there won't be a header file or even a LIB file to help. The only thing you will have is the DLL itself. Fortunately, a DLL can be subjected to a utility like "dumpbin" to get a list of the functions it implements. Listing 3 shows an excerpt of a dumpbin output for ws2_32.dll. From this output, it is easy to create a DEF file as in Listing 1, and it seems likely that it should be equally simple to create a "C" implementation file – but we need to come up with bodies for each function to write a log entry then call the real function.

```
File Type: DLL

  Section contains the following exports for WS2_32.dll

           0 characteristics
    41107EDA time date stamp Wed Aug 04 02:14:50 2004
        0.00 version
           1 ordinal base
         500 number of functions
         117 number of names

    ordinal hint RVA      name

          27    0 00002B0B FreeAddrInfoW
          24    1 00002899 GetAddrInfoW
          25    2 0000C4EC GetNameInfoW
         500    3 00012105 WEP
          28    4 00011CA7 WPUCompleteOverlappedRequest
          29    5 00010DA9 WSAAccept
          30    6 000091F6 WSAAddressToStringA
          31    7 000062B2 WSAAddressToStringW
         102    8 0000EA2B WSAAsyncGetHostByAddr
         103    9 0000E985 WSAAsyncGetHostByName
         105    A 0000EAD5 WSAAsyncGetProtoByName
         104    B 0000E2AB WSAAsyncGetProtoByNumber
```

Listing 3: Dumpbin output

Since we'll be exporting the same functions as the original DLL (ws2_32.dll), our functions will receive control when called from an application, but aside from knowing where the return address is stored on the stack, we don't know anything about the number or type of parameters being passed in. If the calling convention of the function we're implementing is "cdecl", we would probably be safe calling another function to log the event then execute a return statement back to the calling application. However, if the calling convention is "stdcall" or "fastcall", our function would be responsible for clearing the stack when returning. Since we don't

know what has been pushed onto the stack, we can't execute the proper "retn" statement. Furthermore, no matter what the calling convention, we could not properly reconstruct the stack to call the real function.

But if you think about it for just a moment, you'll realize the parameters are already on the stack in the right order, just below any automatic variables and registers saved when the stack frame was constructed by the prolog code generated by the compiler (Figure 1).

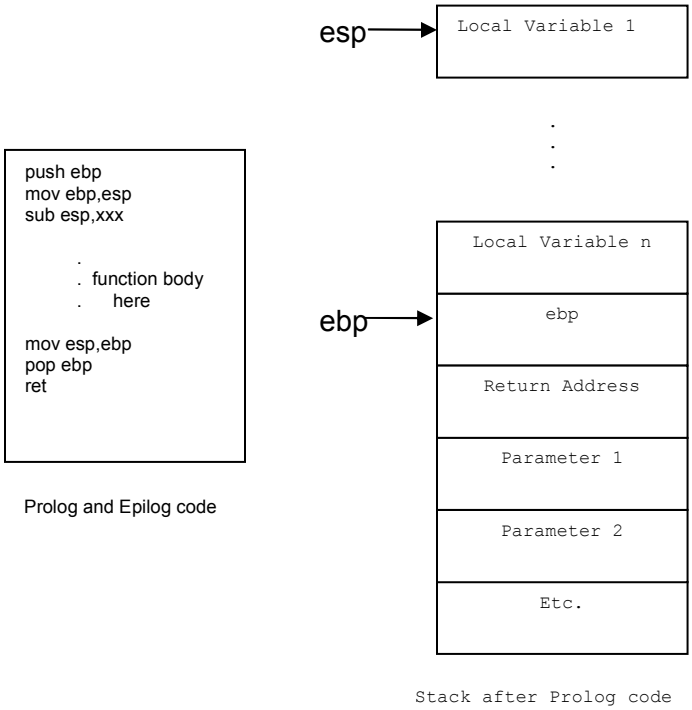

Figure 1: Prolog and Epilog Code

By adjusting the stack pointer to ebp and popping ebp (we need to properly restore ebp as the epilog would), we could jump to the real function which would execute the appropriate ret (or retn) instruction to return to the caller. The epilog code would not be executed. But if we're not going to execute it, why have it there in the first place? We can eliminate it (which would also make stack adjustment easier) if we use the "declspec(naked)" calling convention, which tells the compiler to not generate prolog and epilog code. Listing 4 shows an implementation of this technique. Notice that the return type of the function is "void" and there are no parameters. Because we're letting the called function do the return and the calling function builds the stack, we don't have to worry about either. We can even take this one step further and create a macro that can be used to implement any function. Listing 5 shows such a macro and the implementation of several functions using it. Note that the value returned from the call to GetRealProcAddress is not assigned to any variable because, by convention, the function returns its result in the eax register. We can take advantage of this fact and jump to the address in the eax register (as long as GetRealProcAddress always returns a valid address!).

```
void __declspec(naked) accept ()
{
    LogFunctionCall ("accept");
    GetRealProcAddress ("accept");
    _asm {
        jmp eax
    }
}
```

Listing 4: Naked function

So, before moving on, let's take an inventory of files in a complete project. The project consists of 2 source files: ws2_32.c and ws2_32.def. Listing 6 shows the compiler and linker statements generated by Microsoft Visual C++ (version 7.1) used to build the project.

Granted, this ws2_32.dll doesn't do much other than identify the functions that were called, but having such a simple source code makes it very easy to create new implementations of functions that will show much more detail In malware analysis, the huge variety of malware makes such an easily modifiable utility a necessity.

```
#define IMPLEMENT_SPOOF(func) \
void __declspec (naked) func () \
{ \
        LogFunctionCall (#func); \
        GetRealProcAddress (#func); \
        _asm \
        { \
                jmp eax \
        } \
}

IMPLEMENT_SPOOF(accept)
IMPLEMENT_SPOOF(FreeAddrInfoW)
IMPLEMENT_SPOOF(GetAddrInfoW)
IMPLEMENT_SPOOF(GetNameInfoW)
IMPLEMENT_SPOOF(WEP)
```

Listing 5: Implementation via Macro

Now, in case you've forgotten, the title of this paper is "Kernel-22", but not one thing about a kernel or "22" has yet been mentioned. For certain, the inspiration for this paper came from attempting to develop a spoofing module for kernel32.dll, but the interesting items aren't the ideas that succeeded – the interesting things are the failures and their resolutions. In the remainder of this paper, I'd like to share those "interesting things" with you while still providing the source for a useful spoofing kernel32.dll.

For a malware researcher, the idea of spoofing kernel32 is quite appealing. For starters, it could not only show the functions being called, it could be configured to limit the total number of calls into the kernel before automatically terminating the program, thereby preventing the malware from completing its mission and possibly infecting other components of the analysis network. Output gathered could then be reviewed to determine where breakpoints could be set in a debugger for further analysis of the malware. Other options include modifying functions like GetVersion to return values not normally associated with the particular OS version actually running. The list of potential features is almost endless.

```
cl /Od /I "..\PVLib" /D "WIN32" /D "_DEBUG" /D
"_WINDOWS" /D "_USRDLL" /D "WS2_32_EXPORTS" /D
"_WINDLL" /D "_MBCS" /Gm /EHsc /RTC1 /MTd /Fo"Debug/"
/Fd"Debug/vc70.pdb" /W3 /nologo /c /Wp64 /ZI /TC ws2_32.c
```

```
link /OUT:"Debug/ws2_32.dll" /INCREMENTAL /NOLOGO
/DLL /DEF:"ws2_32.def" /DEBUG /PDB:"Debug/ws2_32.pdb"
/SUBSYSTEM:WINDOWS /IMPLIB:"Debug/ws2_32.lib"
/MACHINE:X86 kernel32.lib
ws2_32.obj  user32.lib gdi32.lib winspool.lib comdlg32.lib
advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib
odbc32.lib odbccp32.lib
```

Listing 6: Build Commands

So let's get started! Using the concepts already discussed, we get the list of exports from kernel32.dll and create 2 files: kernel32.c and kernel32.def. The kernel32.c file will contain DllMain, GetRealProcAddress, LoadRealDll, LogFunctionCall, NotImplemented, the IMPLEMENT_SPOOF macro, and all the kernel32 functions from the dumpbin output. The kernel32.def file will contain the LIBRARY statement followed by the EXPORTS statement with all the functions (and ordinals) from the dumpbin output. The kernel32.c file contains functions much like those in Listing 2, but they have been condensed a bit. Listing 7 contains excerpts from these files.

```
;; Module definition for kernel32.dll
LIBRARY  kernel32.dll
EXPORTS
ActivateActCtx @1
AddAtomA @2
AddAtomW @3
AddConsoleAliasA @4
AddConsoleAliasW @5
AddLocalAlternateComputerNameA @6
```

```
#define IMPLEMENT_SPOOF(func) \
__declspec (naked) func () { \
        LogFunctionCall (#func); \
        GetRealProcAddress (#func); \
        _asm { \
                jmp eax \
        } \
}

HANDLE ghKernelKernel = NULL;

LogFunctionCall (char *pFunction)
{
  // write function name to log file …
}

__declspec(naked) NotImplemented () {
  _asm {
    int 3
    ret
  }
}

void * GetRealProcAddress (char *lpProcName) {
  void *addr;

  if (!ghKernelKernel)
    ghKernelKernel = LoadLibrary (
            "\\windows\\system32\\kernel32.dll");
  }
  addr = GetProcAddress (ghKernelKernel,
                         lpProcName);

  addr ? return addr : return NotImplemented;
}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD
fdwReason, LPVOID lpvReserved) {
  return TRUE;
}

IMPLEMENT_SPOOF(ActivateActCtx)
IMPLEMENT_SPOOF(AddAtomA)
IMPLEMENT_SPOOF(AddAtomW)
IMPLEMENT_SPOOF(AddConsoleAliasA)
IMPLEMENT_SPOOF(AddConsoleAliasW)
IMPLEMENT_SPOOF(AddLocalAlternateComputerNameA)
IMPLEMENT_SPOOF(AddLocalAlternateComputerNameW)
```

Listing 7: Kernel32.dll source files

Let's now consider how this program might actually execute. When loaded, the DllMain function will be called to perform any initialization. In this particular version of the program, we might consider initializing a global variable to hold the handle of the "real" kernel32.dll obtained via LoadLibrary, but the SDK documentation is pretty clear that you may not call the LoadLibrary function from DllMain. Therefore, we'll save the initialization of that for later. Since we have nothing else to do, DllMain can just return TRUE.

So what might happen next? Presumably, the application will eventually call a kernel32 function. It doesn't matter which, the macro guarantees the same thing is going to happen: LogFunctionCall, GetRealProcAddress, then jump to the real implementation of the called function. For now, let's assume that LogFunctionCall can output the passed string to a file. The GetRealProcAddress should draw your attention. Notice the first thing it does is check if the real kernel32 has been loaded. If not, it calls LoadLibrary… Ok, you saw this coming. If you call LoadLibrary here, it's going to call the function in *this* DLL, not the one we actually want to call. The LoadLibrary in this DLL would call LogFunctionCall then GetRealProcAddress then LoadLibrary and so on, and so on.

I suppose instead of calling the LoadLibrary function in kerne32.dll, we could call the LdrLoadDll function in ntdll.dll, but I really don't want to dig up the DDK (the .lib file ntdll.lib is not available in the Platform SDK). Besides, it's more fun to work through this problem. (In all honesty, I did try using ntdll functions, but for lots of reasons, this attempt failed miserably).

Clearly, we have a paradox: we need the real kernel32 to be able to load the real kernel32. I'm reminded of the paradox Yossarian encountered in the novel Catch-22:

"There was only one catch and that was Catch-22, which specified that a concern for one's own safety in the face of dangers that were real and immediate was the process of a rational mind. Orr was crazy and could be grounded. All he had to do was ask; and as soon as he did, he would no longer be crazy and would have to fly more missions. Orr would be crazy to fly more missions and sane if he didn't, but if he was sane he had to fly them. If he flew them he was crazy and didn't have to; but if he didn't want to he was sane and had to. Yossarian was moved very deeply by the absolute simplicity of this clause of Catch-22 and let out a respectful whistle.
'That's some catch, that catch-22,' he observed.
'It's the best there is' Doc Daneeka agreed."[1]

You don't have to be crazy to analyze malware, but it helps.

There must be some way to resolve this paradox. There are 2 ways to load a DLL in a process: explicitly via LoadLibrary or implicitly by the loader when the program is started. The explicit method has already been eliminated, so the answer must be to load it implicitly.

So now, our immediate goal is to find a way to get a module named kernel32 to implicitly load a module named kernel32. The loader isn't going to like that. Our kernel32.dll can load a module named "fred", "alice", or just about anything else, but attempting to load kernel32 would be self-referential. However, the kernel32.dll we want to load is just a file, and files can be copied and renamed, so why not copy the real kernel32.dll to a file named "KernelKernel.dll" (suggesting a kernel for our kernel)? This brings our task within the realm of possibility; we just need a KernelKernel.*lib* that we can supply to the linker so our kernel32.dll will import KernelKernel.dll.

Shipped with Microsoft Visual C++ (also available from other sources) is a utility called lib.exe. This utility is used to manipulate .lib files. Among its features is the ability to create a .lib file from a .def file, and we just created a .def file for our kernel32. So, we can copy it, change the LIBRARY statement to be "LIBRARY KernelKernel.dll" and build a KernelKernel.lib. Just execute the following command:

```
c:\>lib /DEF:KernelKernel.def /MACHINE:x86 /OUT:KernelKernel.lib
```

kernel32stub.c that will be called from kernel32.c. Listing 8 shows the new files.

Beyond the addition of the #include statement for windows.h, examination of the modifications show the initialization of ghKernelKernel has been moved to DllMain. This can be done because it is no longer set from a call to LoadLibrary, but instead from a

[1] Joseph Heller, *Catch-22* chapter 5: *Chief White Halfoat*

call to GetModuleHandle. This will also reduce the overhead whenever GetRealProcAddress is called because it won't have to continuously check if the real module has been loaded.

While these changes should satisfy the compiler, the linker will still have a problem. When kernel32.c is compiled, the object file kernel32.obj will be produced. Inside this object file are the same functions that are in the KernelKernel.lib we built earlier. How does the linker know which to use? The answer is that it doesn't. We're going to have to find a way to differentiate the functions we produce from the functions in the KernelKernel.lib, yet still have our DLL export the those same function names. Another paradox is revealed.

```
;; Module definition for kernel32.dll
LIBRARY  kernel32.dll
EXPORTS
ActivateActCtx @1
AddAtomA @2
AddAtomW @3
AddConsoleAliasA @4
AddConsoleAliasW @5
AddLocalAlternateComputerNameA @6
```

```
//                kernel32.c
#include "kernel32stub.h"

#define JIMPLEMENT(func) \
__declspec (naked) J##func () \
{ \
         LogFunctionCall (#func); \
         GetRealProcAddress (#func); \
         _asm \
         { \
                  jmp eax \
         } \
}

IMPLEMENT_SPOOF(ActivateActCtx)
IMPLEMENT_SPOOF(AddAtomA)
IMPLEMENT_SPOOF(AddAtomW)
IMPLEMENT_SPOOF(AddConsoleAliasA)
IMPLEMENT_SPOOF(AddConsoleAliasW)
IMPLEMENT_SPOOF(AddLocalAlternateComputerNameA)
IMPLEMENT_SPOOF(AddLocalAlternateComputerNameW)
                  .
```

```
//             kernel32stub.h
#ifndef KERNEL32STUB_INCLUDED
#define KERNEL32STUB_INCLUDED
LogFunctionCall (char *pFunction);
LogParams (char *pFormat);
GetRealProcAddress (char *pFunction);

#endif
```

```
//              kernel32stub.c
#include <windows.h>

#define IMPLEMENT_SPOOF(func) \
__declspec (naked) func () { \
         LogFunctionCall (#func); \
         GetRealProcAddress (#func); \
         _asm { \
                  jmp eax \
         } \
}

HANDLE ghKernelKernel = NULL;

LogFunctionCall (char *pFunction)
{
  // write function name to log file …
}

__declspec(naked) NotImplemented () {
  _asm {
    int 3
    ret
  }
}

void * GetRealProcAddress (char *lpProcName)
{
  void *addr;

  addr = GetProcAddress (ghKernelKernel,
                         lpProcName);

  addr ? return addr : return NotImplemented;
}

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD
             fdwReason, LPVOID lpvReserved) {
  ghKernelKernel =
        GetModuleHandle("KernelKernel.dll");

  return (BOOL) ghKernelKernel;
}
```

Listing 8: Kernel32.dll source files

The solution this time is found in the .def file and our IMPLEMENT_SPOOF macro. One of the features of a def file is that you can assign an export name to a function that is different from the actual function name. It's as easy as "ExportName=InternalName". If we modify the IMPLEMENT_SPOOF macro to add a prefix to each of the functions, they will be different from those in KernelKernel.lib. Listing 9 shows the changes to the macro and the def file. Note that the changes to the def file are still easily created from the output of the dumpbin utility.

The IMPLEMENT_SPOOF macro now utilizes the "##" concatenation operator to prefix the function name with the letter "J". The def file assigns the export name to the "J" prefixed name.

We have now resolved the linker conflict. What do you think will be next?

Instead of keeping you in suspense, I'll just get to it. After all the work we've done trying to satisfy the compiler and linker, building these files results in an unresolved external for _GetModuleHandleA@4, _GetProcAddress@8, and every other kernel32 function referenced in kernel32stub.c. Obviously, building KernelKernel.lib from a def file which was built from dumpbin output of kernel32.dll didn't work – but why? For certain, dumpbin didn't lie about what is exported by kernel32. Could it be that a def file that maps external names to internal names was used by the developers of kernel32.dll? Well, not exactly. Yes, kernel32 is built using a def file, but not to map export names of kernel32.dll to these weird names prefixed with an underscore and suffixed with an '@' symbol and a number. These weird (decorated) names are actually generated by the compiler as a result of the stdcall calling convention used. The number following the '@' symbol in each name specifies the number of bytes that are cleared from the stack by the retn instruction when the function returns.

```
;; Module definition for kernel32.dll
LIBRARY  kernel32.dll
EXPORTS
ActivateActCtx = JActivateActCtx @1
AddAtomA = JAddAtomA @2
AddAtomW = JAddAtomW @3
AddConsoleAliasA = JAddConsoleAliasA @4
AddConsoleAliasW = JAddConsoleAliasW @5
                   .
                   .
                   .
```

```
#define IMPLEMENT_SPOOF(func) \
__declspec (naked) J##func () { \
        LogFunctionCall (#func); \
        GetRealProcAddress (#func); \
        _asm { \
                jmp eax \
        } \
```

Listing 9: Def and Macro Changes

So it appears that to generate a KernelKernel.lib file, we need to prefix all the functions in our KernelKernel.def file with underscore characters and suffix them with the '@' values. But rather than carefully typing each value, I thought it would be faster to run dumpbin on the kernel32.lib file that is in the Platform SDK then, once again, use the lib utility to generate a new KernelKernel.lib.

Now, the project compiles, links and creates kernel32.dll! Now how do we load it?

Unlike other DLL's like ws2_32.dll, kernel32.dll is a special DLL to Windows. It is a "Known DLL". A Known DLL is a DLL that Windows loads from disk once at boot time, and when any new process requires it, it is not reloaded from disk. The copy already in memory is just shared with the new process.

This is a problem for us because our kernel32.dll will never get loaded, but there is a way around this problem, too. Known DLLs are "known" because they are listed in the registry under "\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Contol\Session Manager\KnownDLLs". When a DLL is listed under this key, it and all DLLs it depends upon, become Known DLLs. This means that a DLL can be a Known DLL even if it isn't listed under this key. For example, ole32.dll is listed as known DLL. It imports msvcrt.dll, ntdll.dll, gdi32.dll, kernel32.dll, user32.dll, advapi32.dll, and rpcrt4.dll. All these DLLs become Known DLLs *even if they are not listed explicitly*. Therefore, you cannot just remove kernel32.dll from the list of known DLLs. You have to *add* it to the value "ExcludeFromKnownDlls" that is found under the key "\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Contol\Session Manager". Make sure to reboot.

Now, if our spoofing kernel is found in the search path before \windows\system32\kernel32.dll, it will be loaded. To keep things simple, I just prefix my search path with ".\;" to make sure the current directory is checked first. To use the spoofing DLL, create an empty directory and copy the spoofing kernel32.dll, KernelKernel.dll (which is a copy of the real kernel32.dll), and the program you want to monitor into the new directory. Start the program from that directory and BOOM! It doesn't work. Actually, less of a BOOM and more of "A Dll Failed to Initialize Properly".

Now, a sane person might give up by now, but sanity is overrated.

If you've ever written a DLL before and you've seen this message, you might guess that the error is probably somewhere in the DllMain function. It is possible to debug this using the Microsoft Visual C++ debugger by setting the property values in the project configuration. But, if you do this, you might be surprised to see that the program fails *before* DllMain is ever called by the loader. The Visual C++ debugger won't debug this.

Maybe sanity isn't overrated.

In order to debug this problem, a kernel debugger is necessary. The Microsoft debugging tools are available for this purpose. Included in these tools is "WinDbg.exe", a windows-based debugging tool capable of debugging the kernel. I'll skip the debugging details and get right to the problem. The reason the DLL failed to initialize is because the functions imported by kernel32.dll were not found in KernelKernel.dll. Since the imports in kernel32.dll could not be resolved, DllMain was never called.

But which functions are missing? As it turns out, all of them. If you run dumpbin to list the imports on any program that is dependent upon kernel32.dll, you'll see the same function names that you find when you examine the exports of kernel32.dll. If you run the dumpbin utility against the spoofing kernel32.dll to see what it is importing from KernelKernel.dll, you'll see every function is prefixed with an underscore and suffixed with an '@' symbol and a number. (I wish I did this *before* all those WinDbg sessions). So what happened? The dumpbin on the kernel32.lib from the Platform SDK showed the same exported functions as the KernelKernel.lib created via the lib utility and def file. A more detailed look at the 2 files revealed a subtle difference. The "Microsoft Portable Executable and Common Object File Format Specification" indicates that the format of an import library includes a flag called "Import Type" that can have a setting of "IMPORT_NAME_UNDECORATE". If this flag is set, the linker will remove the leading underscore and "@" suffix when building the import table for an executable (or dll).

Closer examination of the kernel32.lib shows that this bit is set, while in KernelKernel.lib, it is not. So how do we get it be set in our KernelKernel.lib? From what I have been able to determine, this bit cannot be set with just the lib utility and a def file. Another method for creating the KernelKernel.lib file needs to be found.

The only thing I could think of was to actually build a KernelKernel.dll from source code, throw away the resulting DLL and keep the .lib and .exp files that were created in the process. I really don't want to type in all those functions, though. Since the decorated names don't include any information about the return type of the functions, they can be all the same. In fact, it doesn't say anything about the types of arguments, either, so those can all be the same, too. All that needs to be in agreement with the decorated name is the stdcall calling convention and the number of parameters. It turns out that it's pretty easy to create a source file from the exports from kernel32.lib of the Platform SDK. Listing 10 shows an excerpt from the source file. The KernelKernel.def file doesn't change.

```
void __stdcall ActivateActCtx(void *p, void *q) {}
void __stdcall AddAtomA(void *p) {}
void __stdcall AddAtomW(void *p) {}
void __stdcall AddConsoleAliasA(void *p, void *q, void *r) {}
void __stdcall AddConsoleAliasW(void *p, void *q, void *r) {}
void __stdcall AddLocalAlternateComputerNameA(void *p, void *q) {}
void __stdcall AddLocalAlternateComputerNameW(void *p, void *q) {}
void __stdcall AddRefActCtx(void *p) {}
```

Listing 10: Excerpt from KernelKernel.c

Compilation of KernelKernel.c and subsequent linking of the resultant object file with the def file will produce a lib file which has the IMPORT_NAME_UNDECORATE flag set. Discard the KernelKernel.dll produced and link kernel32.obj with the lib file to produce kernel32.dll. Finally, a working spoofing DLL is made!

Now, it wouldn't be wise to assume that if this works for one or two programs, it will work for all programs, and it didn't take long to find a program that failed, but why?

Debugging revealed that the failure occurred shortly after the function GetProcAddress was called to obtain the address of the function "RtlEnterCriticalSection" in the spoofing kernel32. Because this function was not exported, a valid address could not be returned. Perhaps I missed an export from the real kernel32, but when I checked, I found the real kernel32.dll doesn't export this function, either -- at least, not directly.

If we examine the dumpbin output for the real kernel32 more closely (Listing 11), we find that the function "EnterCriticalSection" is exported, but is forwarded to "NTDLL.RtlEnterCriticalSection". The spoofing kernel *does* export EnterCriticalSection, but it is not forwarded in the same manner, so RtlEnterCriticalSection is not. When GetProcAddress (in the real kernel32) is called for RtlEnterCriticalSection, it returns the address of for EnterCriticalSection. We need to make the spoofing DLL behave the same way -- not just for RtlEnterCriticalSection, but for all functions forwarded by kernel32.dll.

Note that of all the functions forwarded, not all are forwarded with a different name. For instance, RtlDecodePointer is forwarded as DecodePointer, but RtlCaptureContext is forwarded as the same name, RtlCaptureContext. Therefore, it is only necessary to add

the *different* names to the .def file. Since this is a trivial exercise, I've omitted the listing of the updated .def file.

```
AddVectoredExceptionHandler (forwarded to NTDLL.RtlAddVectoredExceptionHandler)
DecodePointer (forwarded to NTDLL.RtlDecodePointer)
DecodeSystemPointer (forwarded to NTDLL.RtlDecodeSystemPointer)
DeleteCriticalSection (forwarded to NTDLL.RtlDeleteCriticalSection)
EncodePointer (forwarded to NTDLL.RtlEncodePointer)
EncodeSystemPointer (forwarded to NTDLL.RtlEncodeSystemPointer)
EnterCriticalSection (forwarded to NTDLL.RtlEnterCriticalSection)
GetLastError (forwarded to NTDLL.RtlGetLastWin32Error)
HeapAlloc (forwarded to NTDLL.RtlAllocateHeap)
HeapFree (forwarded to NTDLL.RtlFreeHeap)
HeapReAlloc (forwarded to NTDLL.RtlReAllocateHeap)
HeapSize (forwarded to NTDLL.RtlSizeHeap)
InitializeSListHead (forwarded to NTDLL.RtlInitializeSListHead)
InterlockedFlushSList (forwarded to NTDLL.RtlInterlockedFlushSList)
InterlockedPopEntrySList (forwarded to NTDLL.RtlInterlockedPopEntrySList)
InterlockedPushEntrySList (forwarded to NTDLL.RtlInterlockedPushEntrySList)
LeaveCriticalSection (forwarded to NTDLL.RtlLeaveCriticalSection)
QueryDepthSList (forwarded to NTDLL.RtlQueryDepthSList)
RemoveVectoredExceptionHandler (forwarded to NTDLL.RtlRemoveVectoredExceptionHandler)
RestoreLastError (forwarded to NTDLL.RtlRestoreLastWin32Error)
RtlCaptureContext (forwarded to NTDLL.RtlCaptureContext)
RtlCaptureStackBackTrace (forwarded to NTDLL.RtlCaptureStackBackTrace)
RtlFillMemory (forwarded to NTDLL.RtlFillMemory)
RtlMoveMemory (forwarded to NTDLL.RtlMoveMemory)
RtlUnwind (forwarded to NTDLL.RtlUnwind)
RtlZeroMemory (forwarded to NTDLL.RtlZeroMemory)
SetCriticalSectionSpinCount (forwarded to NTDLL.RtlSetCriticalSectionSpinCount)
SetLastError (forwarded to NTDLL.RtlSetLastWin32Error)
TryEnterCriticalSection (forwarded to NTDLL.RtlTryEnterCriticalSection)
VerSetConditionMask (forwarded to NTDLL.VerSetConditionMask)
```

Listing 11: Functions Forwarded by Kernel32.dll

Conclusion

I don't think it's possible to list all the potential uses for spoofing of kernel32, but the source code framework provided makes it very simple to intercept any kernel32 function, insert new processing, and optionally continue with the original processing. All you have to do is remove the function of interest from kernel32.c and implement it (with proper signature and "J" prefix) in kernel32stub.c. It's up to you to decide if the "real" function is called or not.

One of the features I like most is that I don't have to have a specially built system where this is installed to be able to perform analysis of a program. I can install this on any system very quickly by changing 1 registry key and copying a couple files – and it uninstalls just as easily. This is especially nice when examining your friend's home computer because he knows you can fix it for free (there's no sense copying that nasty virus he got onto *your* machine).

One final note: I've attempted to chronicle my own experience when developing the spoofing kernel and tried to be as accurate as possible with respect to the sequence of events and the errors I encountered (or caused). To be sure, I omitted several things either intentionally (like the cursing) or accidentally (you might find these when you try this yourself). There may be instances where I describe a certain error, but it can't be recreated by the code snippets I supplied here. If this occurs, I apologize, but please remember I did get these errors and the prescribed solutions solved them. So, even if you can't recreate one of these errors, you might see it in some other context in the future. If you do, you might now know how to fix it.