Intercepting System API Calls

by Seung-Woo Kim

May 13, 2004
The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel, the Intel logo, Pentium, Intel Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2005 Intel Corporation
Introduction

There are many cases where it is necessary for software developers or testers to intercept system function calls in order to instrument code or to extend operating-system functionality. There are a few packages available that provide this functionality, such as the Detours* library from Microsoft or Syringe* from OK Thinking Software. On the other hand, developers may wish to implement this functionality themselves, without implementing third-party software.

This article describes various ways of function interception and presents a generic method to achieve this task without relying on commercial software packages or being bound to GNU licensing. All materials in this paper were either developed by Intel or modified from MSDN sample code.

Two Basic Techniques for Intercepting System Function Calls

Most methods of intercepting arbitrary function calls work by preparing a DLL that replaces the target function to be intercepted and then injecting the DLL to the target process; upon attaching to the target process, the DLL hooks itself to the target function. This technique is suitable, because the source code for the target application is not available most of the time, and it is relatively simple to write a DLL that contains the replacement function, separating it from the rest of the software.

Two intercepting methods have been studied and analyzed. Syringe works by modifying the function import entries (thunking table). On the other hand, the Detours library directly modifies the target function (in the target process space) to make an unconditional jump to the replacement function. Optionally, it provides a trampoline function that can call the original function.

The Detours technique follows this latter method because Syringe has trouble finding the thunks in many cases, and it does not provide trampoline capability to call the original function. Injecting the DLL works the same way in both cases.

The overall workflow to intercept system function calls is as follows:

**DLL Injection**: First, the main software opens the target process and forces it to load the DLL that contains the replacement functions.

**Target Function Modification**: When the DLL attaches to the process, it modifies the target function in the target process space so that it directly jumps to the replacement function in the DLL. Optionally, a trampoline function can call the original function.

**Target Function Intercepted**: When the target function is called, it directly jumps to the replacement function in the DLL. If the developer wishes to invoke the original functionality, he or she calls the trampoline function.
**DLL Injection**

This section is entirely based on the MSDN article, "Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities," which includes downloadable source code. `Inject.cpp` and `Inject.h` are available as an appendix to this article. They are customized for easy integration – just include them in a project and call `InjectLib`. The algorithm to force the target process to load the DLL works as follows:

Open the target process by calling `OpenProcess`.

Allocate memory in the target process by calling `VirtualAllocEx`. Write to the allocated memory the name of the DLL to be injected using `WriteProcessMemory`.

Get the address of `LoadLibrary` by calling `GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");`

Call `CreateRemoteThread`, specifying the entry point of `LoadLibrary` and the name of the DLL (in step 2) as its argument. The target process will load the DLL.

Free the allocated memory using `VirtualFreeEx`. It is not needed anymore.

`Inject.cpp` incorporates a great deal more functionality, including substantial security features, but the preceding steps are sufficient to illustrate core concepts.

**Target-Function Modification**

Target-function modification is self-modifying code that is well documented on MSDN*, although there are a few pitfalls in injecting jmp into the process memory. This section shows almost complete sample code to avoid confusion. The two aspects of target-function modification are replacement and trampoline functions.

The following code snippet is an example DLL to intercept the `GetSystemPowerStatus` API:

```c
BOOL WINAPI GetSystemPowerStatusReplaced(LPSYSTEM_POWER_STATUS lpSystemPowerStatus)
{
    // Your replacement code goes here.
    return TRUE;
}
```

```c
BOOL InterceptAPI(HMODULE hLocalModule, const char* c_szDllName, const char* c_szApiName, DWORD dwReplaced)
```
{  
  DWORD dwOldProtect;

  DWORD dwAddressToIntercept = (DWORD)GetProcAddress(
      GetModuleHandle((char*)c_szDllName), (char*)c_szApiName);

  BYTE *pbTargetCode = (BYTE *) dwAddressToIntercept;
  BYTE *pbReplaced = (BYTE *) dwReplaced;

  VirtualProtect((void *) dwAddressToIntercept, 5, PAGE_WRITECOPY, &dwOldProtect);

  *pbTargetCode++ = 0xE9;    // jump rel32

  *((signed int *)(pbTargetCode)) = pbReplaced - (pbTargetCode +4);

  VirtualProtect((void *) dwAddressToIntercept, 5, PAGE_EXECUTE, &dwOldProtect);

  FlushInstructionCache(GetCurrentProcess(), NULL, NULL);

  return TRUE;
}

BOOL WINAPI DllMain(HINSTANCE hInst, DWORD dwReason, LPVOID reserved)
{
  
  if (dwReason == DLL_PROCESS_ATTACH) {
      InterceptAPI(hInst, "kernel32.dll", "GetSystemPowerStatus",
          (DWORD) GetSystemPowerStatusReplaced);
  }

  else if (dwReason == DLL_PROCESS_DETACH) {
      // Cleanup
  }
}
The first thing this code does upon attaching is to call `InterceptAPI`. It requires the name of the module containing the target function, the name of the target function, and the address of the replacement function. `GetSystemPowerStatus` is in `kernel32.dll`. Other basic Win32* APIs, such as `MessageBox` and `PeekMessage`, are available in `user32.dll`. MSDN specifies the module to which each API belongs; a future enhancement could automatically find the correct module for a given API.

`InterceptAPI` overwrites the first five bytes of the target function to an unconditional jump (opcode 0xE9), followed by the displacement to the replacement function as a signed integer (four bytes). The displacement starts at the next instruction; hence, `pbReplaced - (pbTargetCode + 4)` is required. Two cautions are necessary to make this code work:

- Change the protection mode of the region overwritten by `VirtualProtect`. Otherwise, an access-violation error occurs.
- `FlushInstructionCache` is necessary to support those cases where the instructions are already in cache. Otherwise, old code will run from cache, even though the instructions have been changed in memory.

Now, when the `GetSystemPowerStatus` function is called, all it does is to jump to our replacement function, and it returns directly to the caller, successfully intercepting the call.

**Trampoline Function**

In many cases, the replacement function needs to call the original target function in addition to its own code, in order to extend the capability of the API, rather than replacing the whole thing. A trampoline function provides this functionality. The theory behind trampoline functions is as follows:

- Prepare a dummy function that has the same declaration that will be used as the trampoline. Make sure the dummy function is more than 10 bytes long.

- Before overwriting the first five bytes of the target function, copy them to the beginning of the trampoline function.

- Overwrite from the sixth byte of the trampoline with an unconditional jump to the sixth byte of the target function.

- Overwrite the target function as before.

When a trampoline function is called (from the replacement function or anywhere else), it executes the first five bytes of the copied original code, and then jumps to the sixth byte of the real original
code. The control returns to the caller of the trampoline. After optionally completing additional tasks, control returns to the caller of the API.

One additional complication exists, in that the sixth byte of the original code may be part of the previous instruction. In that case, the function overwrites part of the previous instruction and then crashes. In the case of `GetSystemPowerStatus`, the beginning of a new instruction after the first five bytes is the seventh byte. Thus, for this scheme to work, six bytes need to be copied to the trampoline, and the code must adjust this offset accordingly.

The number of bytes that the code needs to copy depends upon the API. It is necessary to look at the original target code (using a debugger or a disassembler) and to count the number of bytes to copy. A future enhancement could automatically detect the correct offset. Assuming that we know the correct offset, the following code shows the extended `InterceptAPI` function that sets up the trampoline function as well:

```c
BOOL InterceptAPI(HMODULE hLocalModule, const char* c_szDllName, const char* c_szApiName,
                  DWORD dwReplaced, DWORD dwTrampoline, int offset)
{
    int i;

    DWORD dwOldProtect;

    DWORD dwAddressToIntercept = (DWORD)GetProcAddress(
        GetModuleHandle((char*)c_szDllName), (char*)c_szApiName);

    BYTE *pbTargetCode = (BYTE *) dwAddressToIntercept;
    BYTE *pbReplaced = (BYTE *) dwReplaced;
    BYTE *pbTrampoline = (BYTE *) dwTrampoline;

    // Change the protection of the trampoline region
    // so that we can overwrite the first 5 + offset bytes.

    VirtualProtect((void *) dwTrampoline, 5+offset, PAGE_WRITECOPY, &dwOldProtect);

    for (i=0; i<offset; i++)
    {
        *pbTrampoline++ = *pbTargetCode++;
    }
}```
pbTargetCode = (BYTE *) dwAddressToIntercept;

// Insert unconditional jump in the trampoline.
*pbTrampoline++ = 0xE9;        // jump rel32
*((signed int *)(pbTrampoline)) = (pbTargetCode+offset) - (pbTrampoline + 4);
VirtualProtect((void *)  dwTrampoline, 5+offset, PAGE_EXECUTE, &dwOldProtect);

// Overwrite the first 5 bytes of the target function
VirtualProtect((void *) dwAddressToIntercept, 5, PAGE_WRITECOPY, &dwOldProtect);
*pbTargetCode++ = 0xE9;        // jump rel32
*((signed int *)(pbTargetCode)) = pbReplaced - (pbTargetCode +4);
VirtualProtect((void *) dwAddressToIntercept, 5, PAGE_EXECUTE, &dwOldProtect);

// Flush the instruction cache to make sure
// the modified code is executed.
FlushInstructionCache(GetCurrentProcess(), NULL, NULL);
return TRUE;
}

**Conclusion**

This article describes a generic method to intercept system function calls, as well as providing trampoline functions to retain the original functionality. Because this paper is a summary of methods, rather than a complete package, some details are not implemented:

Automatic detection of the module containing the target API.

Automatic detection of the offset for the trampoline function.
Removing replacement functions and ejecting the DLL. (For now, the only way to clean up is to close the application.)

Nevertheless, the techniques, explanations, and source code in this article should be sufficient for developers to implement software that can intercept any system function calls without relying on third-party software packages.

About the Author

Seung-Woo Kim received his Ph.D in computer science at University of Minnesota and is currently working as a senior application engineer at Intel. He specializes in the performance optimization for technical and commercial software. He can be reached at seung-woo.kim@intel.com

Additional Resources

Articles

Writing Portable, Adaptable Code Using CompLib

Introduction to Intrinsics

Intel® Extreme Graphics 2: Developer's Guide

Branch and Loop Reorganization to Prevent Mispredicts

Intel Software Forums

http://softwareforums.intel.com/

Appendix

Inject.h

 ifndef _INJECT_H

#define _INJECT_H

 ifndef UNICODE

 #define InjectLib InjectLibW

#else

#endif


#define InjectLib InjectLibA
#endif   // !UNICODE

BOOL WINAPI InjectLibW(DWORD dwProcessId, PCWSTR pszLibFile);
BOOL WINAPI InjectLibA(DWORD dwProcessId, PCSTR pszLibFile);

#endif

Inject.cpp
#include "stdafx.h"
#include "Inject.h"
#include <tchar.h>
#include <malloc.h>        // For alloca
#include <aclapi.h>

#ifdef UNICODE
#define InjectLib InjectLibW
#else
#define InjectLib InjectLibA
#endif   // !UNICODE

BOOL AdjustDacl(HANDLE h, DWORD DesiredAccess)
{ 
  // the WORLD Sid is trivial to form programmatically (S-1-1-0)
  SID world = { SID_REVISION, 1, SECURITY_WORLD_SID_AUTHORITY, 0 }; 

  EXPLICIT_ACCESS ea =
  {
    DesiredAccess,
    SET_ACCESS,
    NO_INHERITANCE,
    {
      0, NO_MULTIPLE_TRUSTEE,
      TRUSTEE_IS_SID,
      TRUSTEE_IS_USER,
      reinterpret_cast<LPTSTR>(&world)
    }
  }

  ACL* pdacl = 0;
  DWORD err = SetEntriesInAcl(1, &ea, 0, &pdacl);
  if (err == ERROR_SUCCESS)
  {
    err = SetSecurityInfo(h, SE_KERNEL_OBJECT, DACL_SECURITY_INFORMATION, 0, 0, pdacl, 0); 
    LocalFree(pdacl);
    return(err == ERROR_SUCCESS);
  }
}
else

    return(FALSE);

}

// Useful helper function for enabling a single privilege

BOOL EnableTokenPrivilege(HANDLE htok, LPCTSTR szPrivilege, TOKEN_PRIVILEGES& tpOld)
{
    TOKEN_PRIVILEGES tp;
    tp.PrivilegeCount = 1;

    tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if (LookupPrivilegeValue(0, szPrivilege, &tp.Privileges[0].Luid))
    {
        // htok must have been opened with the following permissions:
        // TOKEN_QUERY (to get the old priv setting)
        // TOKEN_ADJUST_PRIVILEGES (to adjust the priv)

        DWORD cbOld = sizeof tpOld;

        if (AdjustTokenPrivileges(htok, FALSE, &tp, cbOld, &tpOld, &cbOld))
            // Note that AdjustTokenPrivileges may succeed, and yet
            // some privileges weren’t actually adjusted.
            // You’ve got to check GetLastError() to be sure!

            return(ERROR_NOT_ALL_ASSIGNED != GetLastError());
        else
            return(FALSE);
    }
    else


return(FALSE);

// Corresponding restoration helper function

BOOL RestoreTokenPrivilege(HANDLE htok, const TOKEN_PRIVILEGES& tpOld)
{
    return(AdjustTokenPrivileges(htok, FALSE, const_cast<TOKEN_PRIVILEGES*>(&tpOld), 0, 0, 0));
}

HANDLE GetProcessHandleWithEnoughRights(DWORD PID, DWORD AccessRights)
{
    HANDLE hProcess = ::OpenProcess(AccessRights, FALSE, PID);
    if (hProcess == NULL)
    {
        HANDLE hpWriteDAC = OpenProcess(WRITE_DAC, FALSE, PID);
        if (hpWriteDAC == NULL)
        {
            // hmm, we don't have permissions to modify the DACL...
            // time to take ownership...

            HANDLE htok;
            if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY |
                                  TOKEN_ADJUST_PRIVILEGES, &htok))
                return(FALSE);

            return(FALSE);
        }
    }
}
TOKEN_PRIVILEGES tpOld;

if (EnableTokenPrivilege(htok, SE_TAKE_OWNERSHIP_NAME, tpOld))
{
    // SeTakeOwnershipPrivilege allows us to open objects with
    // WRITEOWNER, but that's about it, so we'll update the owner,
    // and dup the handle so we can get WRITE_DAC permissions.
    HANDLE hpWriteOwner = OpenProcess(WRITE_OWNER, FALSE, PID);
    if (hpWriteOwner != NULL)
    {
        BYTE buf[512]; // this should always be big enough
        DWORD cb = sizeof buf;
        if (GetTokenInformation(htok, TokenUser, buf, cb, &cb))
        {
            DWORD err =
                SetSecurityInfo(
                    hpWriteOwner,
                    SE_KERNEL_OBJECT,
                    OWNER_SECURITY_INFORMATION,
                    reinterpret_cast<TOKEN_USER*>(buf)->User.Sid,
                    0, 0, 0
                );
            if (err == ERROR_SUCCESS)
            {
                // now that we're the owner, we've implicitly got WRITE_DAC
                // permissions, so ask the system to reevaluate our request,
// giving us a handle with WRITE_DAC permissions

if (!DuplicateHandle(

    GetCurrentProcess(),
    hpWriteOwner,
    GetCurrentProcess(),
    &hpWriteDAC,
    WRITE_DAC, FALSE, 0
)
)

hpWriteDAC = NULL;

}

}

// don't forget to close handle
::CloseHandle(hpWriteOwner);

}

// not truly necessary in this app,
// but included for completeness

RestoreTokenPrivilege(htok, tpOld);

}

// don't forget to close the token handle
::CloseHandle(htok);
if (hpWriteDAC)
{
    // we've now got a handle that allows us WRITE_DAC permission
    AdjustDacl(hpWriteDAC, AccessRights);

    // now that we've granted ourselves permission to access
    // the process, ask the system to reevaluate our request,
    // giving us a handle with right permissions
    if (!DuplicateHandle(
        GetCurrentProcess(),
        hpWriteDAC,
        GetCurrentProcess(),
        &hProcess,
        AccessRights,
        FALSE,
        0
    ))
    {
        hProcess = NULL;
    }
    
    CloseHandle(hpWriteDAC);
}
return(hProcess);
}

BOOL WINAPI InjectLibW(DWORD dwProcessId, PCWSTR pszLibFile)
{
    BOOL fOk = FALSE; // Assume that the function fails
    HANDLE hProcess = NULL, hThread = NULL;
    PWSTR pszLibFileRemote = NULL;

    // Get a handle for the target process.
    hProcess =
        GetProcessHandleWithEnoughRights(
            dwProcessId,
            PROCESS_QUERY_INFORMATION |   // Required by Alpha
            PROCESS_CREATE_THREAD     |   // For CreateRemoteThread
            PROCESS_VM_OPERATION      |   // For VirtualAllocEx/VirtualFreeEx
            PROCESS_VM_WRITE          // For WriteProcessMemory
        );
    if (hProcess == NULL)
        return(FALSE);

    // Calculate the number of bytes needed for the DLL’s pathname
    int cch = 1 + lstrlenW(pszLibFile);
int cb = cch * sizeof(WCHAR);

// Allocate space in the remote process for the pathname
pszLibFileRemote =
  (PWSTR) VirtualAllocEx(hProcess, NULL, cb, MEM_COMMIT, PAGE_READWRITE);

if (pszLibFileRemote != NULL)
{
  // Copy the DLL's pathname to the remote process's address space
  if (WriteProcessMemory(hProcess, pszLibFileRemote,
    (PVOID) pszLibFile, cb, NULL))
  {
    // Get the real address of LoadLibraryW in Kernel32.dll
    PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
      GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");
    if (pfnThreadRtn != NULL)
    {
      // Create a remote thread that calls LoadLibraryW(DLLPathname)
      hThread = CreateRemoteThread(hProcess, NULL, 0,
        pfnThreadRtn, pszLibFileRemote, 0, NULL);
      if (hThread != NULL)
      {
        // Wait for the remote thread to terminate
        WaitForSingleObject(hThread, INFINITE);
      }
    }
  }
}
fOk = TRUE; // Everything executed successfully

CloseHandle(hThread);

// Free the remote memory that contained the DLL's pathname
VirtualFreeEx(hProcess, pszLibFileRemote, 0, MEM_RELEASE);

CloseHandle(hProcess);

return(fOk);

}

BOOL WINAPI InjectLibA(DWORD dwProcessId, PCSTR pszLibFile) {

// Allocate a (stack) buffer for the Unicode version of the pathname
PWSTR pszLibFileW = (PWSTR)_alloca((lstrlenA(pszLibFile) + 1) * sizeof(WCHAR));

// Convert the ANSI pathname to its Unicode equivalent
wsprintfW(pszLibFileW, L"%S", pszLibFile);
// Call the Unicode version of the function to actually do the work.

return(InjectLibW(dwProcessId, pszLibFileW));

}