Advanced Engineering Forum *Online: 2012-09-26*
ISSN: 2234-991X, Vols. 6-7, pp 26-31
doi:10.4028/www.scientific.net/AEF.6-7.26

Research on Key Technologies of Windows CE API Interception

Chen Xuhui, Chen Jia, Zhu Ping

College of Mathematics & Computer Science, Wuhan Textile University, Hubei, 430064, China

wh.chxh@gmail.com

Keywords: Windows CE, API Interception, Dynamically Linked Library, System Call.

Abstract. Windows CE API interception technology can be used for monitoring various system services calls such as file systems, registry. API interception has a very important role on the Windows CE system software debugging and performance analysis. This paper introduces the main methods of the Windows CE API interception, discusses a number of key technologies involved, how to use these methods flexible, and gives a practical example.

Introduction

Most of the functions of the application programming interface (API) provided by Windows CE operating system is implemented in the system Dynamically Linked Library (DLL) files. The API is based specification intended to be used as an interface by software components to communicate with each other and the operating system. API interception means hooking application or operating system calls to system services, and thus to probe the internal structure of application or operating system. There are many tools and toolkits for API interception in desktop operating system. Such as Process Monitor is a monitoring tool that shows Win32 APIs calls and Detours is software package for re-routing Win32 APIs underneath applications for Windows. But there are few tools and discussion in embedded system, such as Windows CE system. This paper summarizes the principle as well as some key technologies of API interception in Windows CE system, compares the different of the interception methods, and gives a specific case.

The rest of the paper is organized in the following way. Section II surveys previous work in API interception. Section III presents the principle of API interception. Section IV presents details of some API interception methods. In Section V, a practical example is given and the results are analyzed. Finally, a few remarks and discussion are sketched as a conclusion in Section VI.

Related Work

No matter what platforms, the basic principle of API interception is same. Some methods can be applied to both PC and embedded system. But in different system, the technology of API interception may have its own characteristics, which is decided by the realization of the system. Article [7] introduces the principle of system calls in Windows CE and a method to use the principle to intercept API. Article [5] also describes this principle of system calls. The test code given by article [7] is effective but the method to load DLL is a bit complicated and its compatibility is not good. In Windows CE 4.0 and later, a new function is implemented to load a DLL into the kernel's address space. This function can be used to load the DLL that intercept system calls.

The principle of API interception

API interception means hooking one or more system services that application or operating system calls. The hooked APIs are called target functions. The target function is called through a pointer that contains the address of the target function. API interception can be achieved by replacing this address. The target function arguments are generally passed through the stack or registers. Through the arguments in stack or register, we can analyze the function or modify its return results. The

functions that replace the original APIs are called pseudo-functions. The number and the types of the actual arguments in a pseudo function should be same as the original API. The pseudo function may call the original API or not. This is shown in Fig 1.

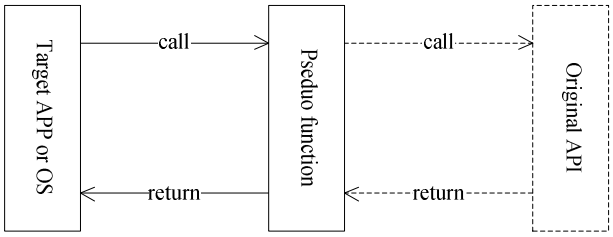

Figure 1. The principle of API interception.

Key Technologies

Replacing the system DLL where the API in. An executable file uses a DLL in two ways in Windows CE, one way is explicit linking and the other way is implicit linking. Explicit linking is referred to as dynamic load or run-time dynamic linking. With explicit linking, the executable using the DLL must make function calls to explicitly load DLL through the system call 'LoadLibrary' and unload DLL through 'FreeLibrary'. And to get the address of DLL's exported functions through 'GetProcAddress'. The client executable must call the exported functions through a function pointer.

Implicit linking is referred to as static load or load-time dynamic linking. With implicit linking, the executable using the DLL links to an import library provided by the maker of the DLL. The operating system loads the DLL when the executable is loaded. The client executable calls the DLL's exported functions just as if the functions were contained within the executable [1].

Regardless of what kinds of linking to a DLL, the name of the DLL must be specified in executable file. The difference is in explicit linking the name of the DLL specified by programmer, and in implicit linking the name of the DLL specified by linker. We can modify the name of the original DLL to another name. The new DLL is called pseudo DLL. If we implement all of the function which exported by the original DLL in the pseudo DLL, this means that all functions of the original DLL are intercepted. The function of the original DLL can still be called by pseudo DLL. This is shown in Fig. 2. If the DLL exists in the application's directory, 'LoadLibrary' ignores the specified path and loads the DLL from the application's directory. So it is not need to modify the target application, the name of the pseudo DLL and the original DLL can be same. It is only need the target programs and pseudo DLL exist in the same directory.

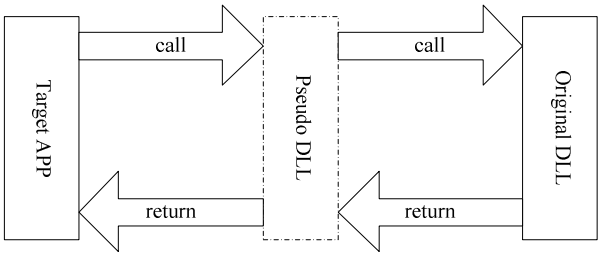

Figure 2. Replacing the dynamic library.

Cross-process code injection. When an executable uses a DLL in implicit linking in Windows CE system, it calls API, for example 'LocalFree', as follows:

```
        LDR    R12, =__imp_LocalFree
        LDR    R12, [R12]
        BX     R12
off_xxxx  DCD __imp_LocalFree
```

'__imp_LocalFree' contains the address of the API after relocated. This address can be modified to achieve the purpose of interception, but this method does not have the versatility. It needs not only analyze the PE file format of the executable but also know where to call the API when executable uses a DLL in implicit linking.

When an executable links a DLL using explicit linking, it is difficult to find somewhere to call the export functions of a DLL. An executable can call function pointer anywhere, and it has no obvious features to find the code.

Another method is to find the entrance of the API in the DLL, save a few bytes of code at the entrance and then modify the code to jump to the pseudo function. When pseudo function wants to call the original API, the saved code is executed firstly then the rest code of the original API is executed. This method is shown in Fig. 3.

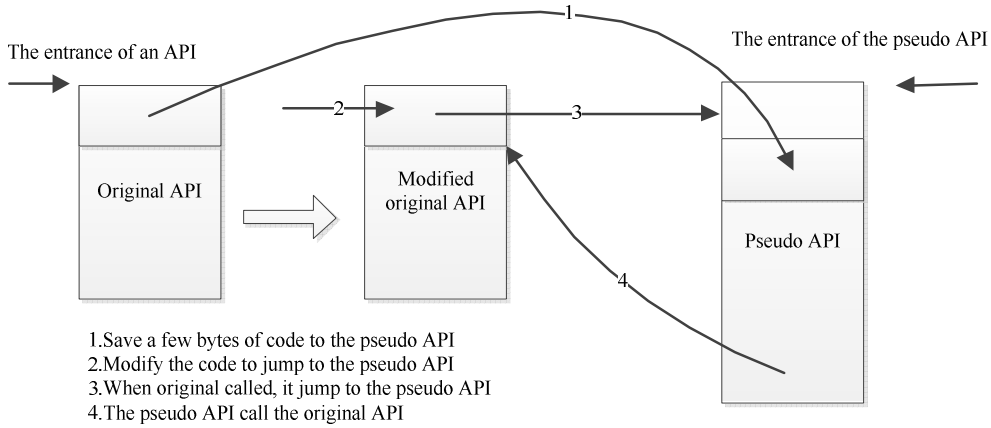

Figure 3. Directly modify the API code.

When an executable loads a DLL firstly, the system maps the DLL module into the virtual address space of the process and increments the reference count. If the DLL whose code is already mapped into the virtual address space of the calling process, only the DLL reference count will be incremented. When Windows CE loads a DLL, it reserves space for it in every process. So a DLL in different processes has the same address space, that is, the virtual address of a DLL's each function in different process is same. Therefore, it is very easy to find the entrance of an API in the other processes as long as we know it in a process. Because the pseudo DLL need to be in the target process's address space, the pseudo DLL must be injected into the target process. This step must be done for each process [2][3][4].

Windows CE system calls. Most of the Windows CE APIs is in coredll.dll. Coredll.dll implements some APIs, the others are done by the coredll.dll calling the operating system and then return to the caller. In order to trap into the system kernel, coredll.dll calls the address pointer from 0xf0000000 to 0xf0010000. This address range is not accessible in the system memory map. Such as 'GetTickCount' call is implemented in a coredll.dll on the ARM processor like following:

```
LDR    R1, =0xF000FFCC
MOV    LR, PC
MOV    PC, R1
……
```

The exception named prefetch abort is triggered when processor executes instruction from those addresses. When the operating system captures the exception and finds the exception address range is between above, it knows that this is a system call.

Windows CE API Set is divided into explicitly API set and handle-based API set. The API set has 32 collections. Its structure is defined as CINFO, the pointer, SystemAPISets, points to this collection. Each explicit API can be called directly by the collection index and the location index in the collection, Fig. 4 shows this structure [5].

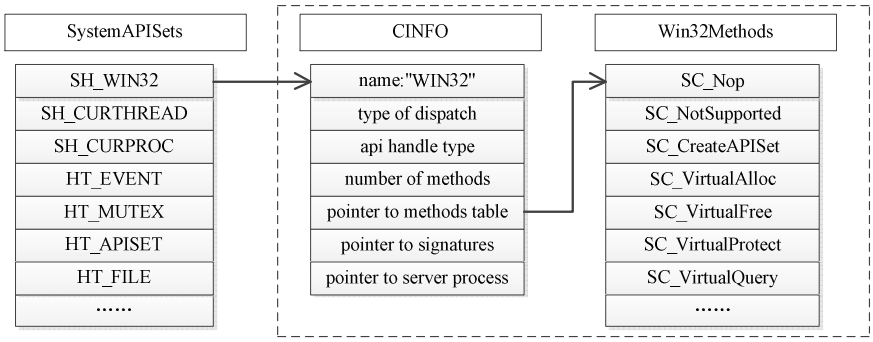

Figure 4. The structure of system call in Windows CE.

The handle-based APIs associated with kernel objects, such as files, events, etc., it is called through the handle and the position index. Such as 'CloseHandle', it can close the different kernel objects. For example, if the handle is a file, it calls the close function of the file system. If the handle is an event, it calls the close function of the event system.

The address of the system call 'GetTickCount' is 0xF000FFCC, how it is calculated? The header psyscall.h in Windows CE.Net the Platform Builder defines the following macro:

```
#define IMPLICIT_CALL(hid, mid) (FIRST_METHOD - ((hid)<<HANDLE_SHIFT | (mid))*APICALL_SCALE)
#define FIRST_METHOD 0xF0010000
#define HANDLE_SHIFT 8
#define APICALL_SCALE 4
#define SH_WIN32      0
#define W32_GetTickCount   13
```

The hid is the index of API Set, the mid is index in this API Set.

'GetTickCount' is in SH_WIN32 API Set and its index in this API set is 13. Therefore, the call address of 'GetTickCount' is IMPLICIT_CALL (SH_WIN32, W32_GetTickCount), it equals to 0xF0010000 – (0<<8 | 13)*4 = 0xF000FFCC。

The pointer of SystemAPISets can be got through the global variable KDataStruct in Windows CE. KDataStruct can be accessed by applications using the fixed address 0xFFFFC800 on the ARM processor. To intercept an API routine can be by replacing a pointer to an API set in the SystemAPISets table or a pointer to an API in methods table. But the methods table may be located in ROM. They are read only, so the best choice is to replace a pointer to an API set in the SystemAPISets table. We allocate a copy of the original CINFO structure and methods table, and then replace the address of the original API set in the SystemAPISets table with the address of the new CINFO and the address of the original method table in the new CINFO with the address of new methods table. Replacing the pointer to an API in the duplicate methods table with the pointer to an interceptor routine can intercept an API [6][7]. This is shown in Fig. 5.

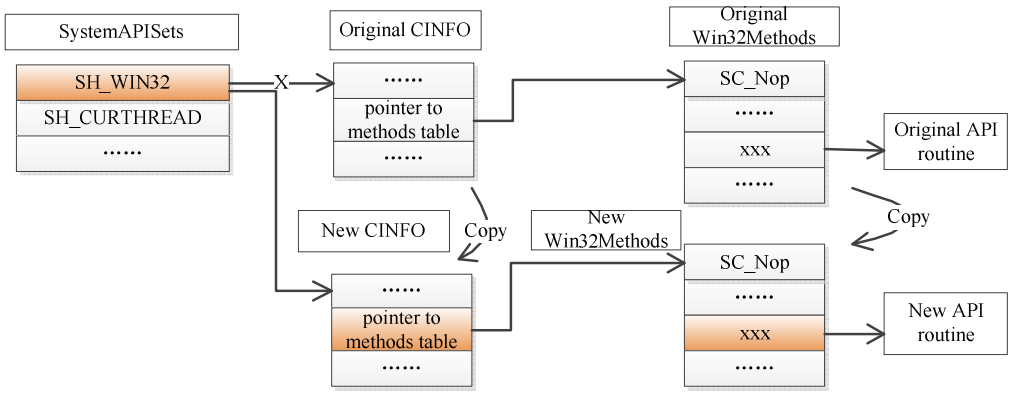

Figure 5. Replacing the system call.

Coredll.dll obtains the pointer of Win32Methods and ExtraMethods using 'GetRomFileInfo' routine when it is loaded every time, then calls these methods directly instead of using traps. So the 'GetRomFileInfo' routine must be intercepted to return the replaced pointer of Win32Methods and ExtraMethods [8].

'LoadKernelLibrary' routine can be used to simplify the code and be able to achieve good result. The 'LoadKernelLibrary' function loads a DLL into the kernel's address space, so the pseudo DLL can be loaded into system kernel easily. But there are some restrictions to using this function. This function loads a DLL that can only be unloaded by restarting the device. In addition, the library loaded by this function cannot have any dependent DLLs. Every function must be statically linked [1].

A practical example using system call to intercept APIs. An application gets the device ID as the unique identifier of the device to verify the operating environment by calling system function 'KernelIoControl'. If we want to the application can run on different devices, we can intercept 'KernelIoControl' and return a device ID that the application can run normally instead of a true one.

The address of a system call may be different in different Windows CE version. So the best way to get the address of a system call is disassembling the function in coredll.dll. For example, disassembling function 'KernelIoControl' in certain version of coredll.dll, we know the address of its system call is 0xF000FE74. According to the above macro defined IMPLICIT_CALL, having the following formula:

APISet = (((FIRST_METHOD - FAULT_ADDR) / APICALL_SCALE) >> HANDLE_SHIFT) & HANDLE_MASK.

MethodIndex = ((FIRST_METHOD -FAULT_ADDR) / APICALL_SCALE) & METHOD_MASK.

So we can know the APISet of 'KernelIoControl' is 0 and the MenthodIndex is 99.

We allocate a copy of CINFO structure pointed by SystemAPISets[0], and a copy of methods table pointed by this CINFO structure, the copy named as new_cinfo and new_mothods respectively. Then we replace the address of the methods table in new_cinfo to new_methods, Replace the address of CINFO structure in SystemAPISets[0] to new_cinfo. A function named new_KernelIoControl is created, and it has same formal parameter as 'KernelIoControl'. The 99th method address in new_methods is replaced to new_ KernelIoControl.

The system call address of 'GetRomFileInfo' is 0xF000FE74 in the coredll.dll. The interception is same as above.

This implementation has an executable file and a DLL file. Their codes are simple. Here is the pseudo code of the executable file:

```
    LoadKernelLibrary ("DLL name");
```

Here is the pseudo code of the DLL file:

```
BOOL new_GetRomFileInfo (…) {
    Call original GetRomFileInfo;
    If want to get the address of Win32Methods and Win32Methods is replaced Return new address of Win32Methods ;}
BOOL new_ KernelIoControl (…) {
    Call original KernelIoControl;
     If want to get device id,  Return fake device id ;}
BOOL DLLEntry (…) {
    Backup original system calls address;
    Replaces system calls address ;}
```

It can be seen from above code, the implementation is simple. Table 1 shows the comparison of these methods.

TABLE 1. Windows CE API Interception method comparison

Methods	advantages	disadvantages
Pseudo DLL	Application-level programming.	Most functions in replaced DLL must be implemented in pseudo DLL.
Cross-process code injection	Only need to implement intercepted function.	Programming is complex.
Intercepting system calls	Code is simple.	System call address may be different in different OS version.

Conclusion

There are several ways to intercept APIs in Windows CE system. But Windows CE is a customizable embedded operating system. The customized systems are not identical. So those methods have advantages and disadvantages respectively.

References

[1] http://msdn.microsoft.com

[2] Chen Xiangqun, Wang Lei, Ma Hongbing, Xiang Yong, “Windows CE.NET system analysis and experimental tutorial,” China machine press, 2003

[3] John Murray, “Inside Microsoft Windows CE,” Microsoft Press, 1998

[4] Jeffrey Richter, “Programming Application for Microsoft Windows,” Beijing: Mechanical Industry Press, 2008

[5] San, “Hacking Windows CE,” Phrack Magazine, 6(63), July 2005.

[6] http://itsme.home.xs4all.nl/projects/xda

[7] Dmitri Leman, “Spy: A Windows CE API Interceptor,” Dr. Dobb's Journal October 2003.

[8] Microsoft, “Windows CE Source Code and Advanced Debugger Commands,” May 07, 2000.