

EAI  
VB.net  
Java  
Mainframe  
C/C++  
ABAP  
Oracle  
DB2  
Symbian



# Application Virtualization

- A Whitepaper



turning **ideas**  
into **reality**

**Interra Information Technologies, Inc.**



## Introduction

Application virtualization is the ability to deploy the software without modifying the host computer or making any changes to the local operating system, file system or registry. Application virtualization decouples applications and data from the OS

- Virtualized applications eliminate nearly all of the complexities and support issues associated with delivering and accessing traditional applications for both fat and thin-client deployments. The time and regression testing required to successfully deliver applications and updates is shortened to hours instead of weeks.
- Virtualized application operates exclusively in user mode and therefore the host operating system and other applications are protected from potential corruption by installation modifications.

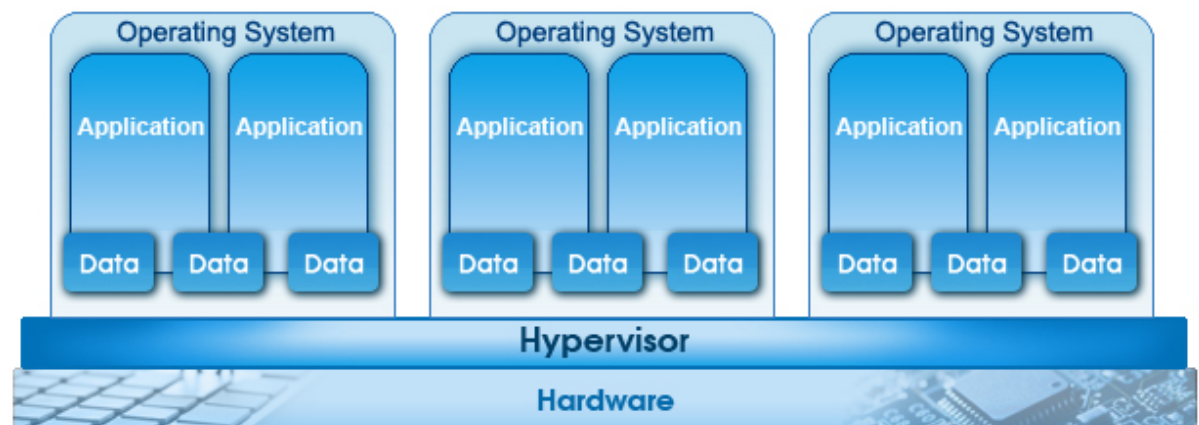
***This paper presents our ideas for “Generic approach for building an Application virtualization environment for standalone windows application.”***

## What is Application Virtualization?

In Nutshell, it is a process by which an ordinary application believes that it is directly interfacing with the operating system and all the resources are managed by it, where as the reality is different.

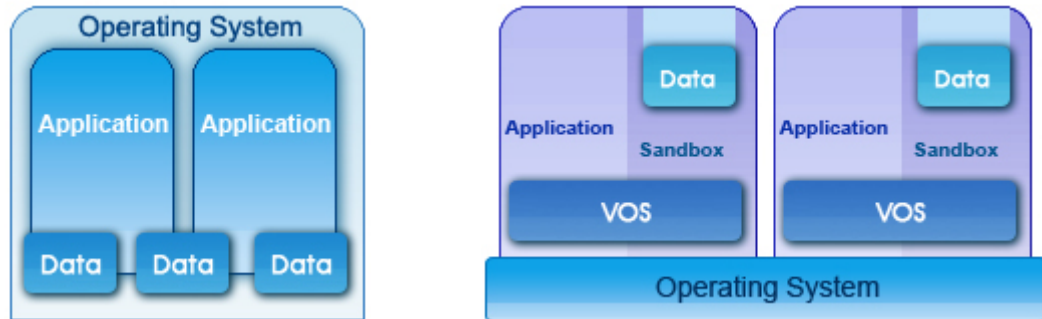
Application virtualization is different from hypervisor technology. Hypervisor creates an abstraction for entire underlying system (i.e. hardware such as memory, CPU, peripherals, video etc). In order to virtualize a single application, layer “of virtualization” is required only between operating system and application itself.

## Hypervisor Technology





## Application Virtualization



To build an application virtualization framework, it's important to understand what kinds of dependencies an application typically has on the operating system from a footprint perspective.

Some of the major dependencies are

- An application stores files on the file system when it installs and makes requests to fetch these files when it executes.
- An application stores configuration and other information in the registry.
- An application also uses environment variables.
- An application may install certain components as “specially handled” elements. E.g. registering binary as a window service.

*Note -: There are some more minor dependencies; this paper only concentrates on above mentioned points.*

Now when application installs, application virtualization framework needs to perform all of the above actions but instead of actually committing those changes to the underlying operating system, they are intercepted and logged separately. The application binaries, including the installer receives information that all the files' copies, registry modifications etc. have been done successfully. But actually all these modifications are logged separately in space called **“Application Delta Catalog”**.

How to fulfill the application's resource expectations when the changes weren't actually committed to the operating system?

By intercepting the applications' request to the file system, the environment, the registry or the windows services and when it asks for a file or a key that doesn't exist on the physical operating system install and check the application delta catalog that was created for the application.

In normal scenario any windows process uses Win32 APIs to make a request to the filesystem or the registry. Application virtualization framework is based on **“How to hook these Win32 APIs”** to route the above requests to other filesystem and registry instead of local filesystem and local registry.

Without knowing the source code of application, how it's possible to intercept all the call for Win32 APIs from an instance of any application (i.e. from windows process).





Let's divide this activity into two parts

- How to intercept and change the behavior of Win32 API calls from any application, for which the source code is available.
- How to make any windows process to intercept and change the behavior of Win32 API calls.

## How to intercept and change the behavior of Win32 API calls

There is no official way to hook a specific Win32 function or API. Instead there are lots of different ways that were invented by a lot of different people and all these API hooking methods have advantages and disadvantages. *There's no perfect solution.*

This paper only describes the solutions which does not touch the standard/system DLLs on hard disk and works purely in memory.

- **Import Table Patching**

This is the most famous approach to hook Win32 APIs. Each win32 application or DLL (in PE format), has its own "import table", which is basically a list of all APIs that this module calls. Patching the import table is quite easy and safe. This involves changing the API addresses in import table. But in this way we can only patch the statically linked win32 APIs calls. Dynamically linked API calls are not caught by this approach at all.

- **Extended Import Table Patching**

To improve the "Import Table patching", hook the API "LoadLibrary" to notify when any new DLLs are loaded and can be patched at the runtime.

Another way is to hook the API "GetProcAddress" and to return the API address of the callback function instead of the original API. One limitation with this approach is there is no unhooking mechanism available with this approach.

- **Export Table Patching**

Every win32 application or DLL (in PE format), has its own "export table", which lists all functions that this module exports. Patching this export table is easy. But patching an export table has absolutely NO effect on already existing API linking (e.g. static bindings of already loaded DLLs). So using export table patching alone is for the restricted use cases only.

- **Simple Code Overwriting**

Another approach is to directly manipulate the API's binary code in memory. Often used technique is to overwrite the first 5 bytes of the API code with JMP instruction, which then jumps to the callback function.

This is very effective approach. It catches all the Win32 API calls. With this approach, it's not possible to call the original API in your call back function. Because once you do that, the JMP instruction would again jump to your call back function, which would result in an endless loop.

One approach to overcome this drawback is to temporarily undo the hook in order to first restore and then call the original API. This workaround too has some limitations.

Steadily hooking and unhooking consumes precious time.

While you have temporarily unhooked the API, you are in big danger of missing out a lot of API calls.





- **Extended Code Overwriting**

While overwriting the 5 bytes of the API's code, copy these bytes to different location. And whenever original API needs to be called, it should be done from the new location. This is a complex approach to implement.

Each instruction can have a different length. So if we copy 5 bytes from the API, that can be exactly one instruction, or one and a half, or two and a half. But executing half copied instruction will make the program to crash. A little dissembler will identify the exact length of the first instruction. These number of bytes need to be copied to another location.

## How to make any windows process to intercept and change the behavior of Win32 API calls

All the techniques mentioned in the section "How to intercept and change the behavior of Win32 API calls", are only possible if source code of an application is available. What if it's not? For this scenario, some injection techniques are required which are responsible for

- Injecting the DLL (for hooking the API) inside the process address space.
- Execute the DLL (for hooking the API) in remote thread.

### Injecting techniques

- **Registry**

In order to inject a DLL into process that links with user32.DLL, simply add the DLL name to the value of the following registry key:

**HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\Appinit\_DLLs**

Its value contains a single DLL name or group of DLLs separated either by comma or spaces. According to MSDN documentation, all DLLs specified by the value of that key are loaded by each windows application. Actual loading of these DLLs occur as a part of user32's initialization. However this approach only works with applications that use USER32.DLL.

Although it is a harmless way to inject a DLL into windows processes, there are some short comings.

In order to activate and deactivate the injection process operating system needs to be rebooted.

There is no control over the injection process. It means that it is implanted into every single GUI application, regardless it's required or not.

- **System-wide Windows Hooks**

Certainly a very popular technique for injecting DLL into a targeted process relies on Windows Hook. According to MSDN documentation, hook is a trap in a system message-handling mechanism. An application can install a custom filter function to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure. For detail information, refer to MSDN documentation.

There are some disadvantages with this approach as well.

Windows hook can degrade the performance of the system significantly.

This affects the processing of the system and under certain circumstances reboot is required in order to recover it.





- **Injecting DLL by using CreateRemoteThread() API function**

This approach is only supported by NT and Windows 2K operating system. Any process can load a DLL dynamically using LoadLibrary() API. The issue is how to force another process to call LoadLibrary().

The function CreateRemoteThread() creates a remote thread in any remote process. The signature of thread function, whose pointer is passed as parameter (i.e. LPTHREAD\_START\_ROUTINE) to CreateRemoteThread is:

**DWORD WINAPI ThreadProc(LPVOID lpParameter);**

The prototype of LoadLibrary API is:

**HMODULE WINAPI LoadLibrary(LPCSTR lpFileName);**

The signature of both the APIs have identical pattern. They have the same calling convention, accept one parameter and the size of returned value is the same. This gives the flexibility to use the LoadLibrary() as a thread function, which will execute after the remote thread has been created.

## Design Points

Once the requirement specification of Application Virtualization framework is done, there are few design points which need to be taken into account.

- Which application needs to be virtualized?
- How to inject the DLL into target processes?
- Which interception mechanism to use?

Identifying the answers to these questions and with above mentioned approaches, you are set to build your application virtualization framework.





turning **ideas**  
into **reality**

---

**Corporate Office**

2001 Gateway Place,  
Suite-670W, San Jose  
CA 95110, USA

**Delivery Centers**

**Noida**  
SDF #E14, NSEZ,  
Noida  
UP 201 305, India

**Kolkata**  
223 SDF Bldg, Sec V,  
Block GP, Kolkata  
WB 700 091, India

**USA Offices**

Bellevue, WA  
Cerritos, CA  
Chicago, IL  
Dallas, TX  
Fullerton, CA  
Omaha, NE  
Princeton, NJ  
Roseville, CA

**Contact Us**

**Namit Kumar**  
Interra Information  
Technologies, Inc.  
Mob: +1-408-839-0795  
Work: +1-408-451-1715