A crack on the glass

Evaluation of consumer Windows OS security architecture and its feasibility as a potential isolation provider for "Qubes Windows Lite"

"Well, dreams, they feel real while we're in them right? Its only when we wake up then we realize that something was actually strange." -- Cobb, *Inception*

Abstract

In this paper we describe our attempt at creating Qubes Windows Native Isolation system, which is a collection of isolated application containers. We discuss the project requirements, what technical means are needed to implement such system and why the available Windows APIs and system architecture is not appropriate for the task.

Rafał Wojdyła, Invisible Things Lab

omeg@invisiblethingslab.com

1. Introduction to Qubes OS

Qubes OS strives to provide strong desktop security by isolation [1]. Having separate domains dedicated to different tasks helps to mitigate potential attacks – a PDF exploit coming from an untrusted site that compromises "public" domain will not affect "work" domain provided they are properly isolated. Qubes R2 is built on top of a Linux host and Xen hypervisor – domains are implemented as lightweight VMs administered from the trusted one, traditionally called *dom0*. This of course implies we trust that Xen-provided isolation is good, but one needs to start somewhere.

The next iteration of Qubes is built on top of *the Odyssey framework* [2] – the goal of which is to decouple Qubes core from Xen. Qubes Odyssey uses the *libvirt* [3] project to implement a hypervisor abstraction layer. That means we can replace Xen with a different *virtual machine manager* (VMM) – Hyper-V, Virtual Box, VMWare or others. Such architectural change provides a great deal of flexibility. If someone doesn't want to be running Linux as the host OS, they can use Windows. Going even further, one can think of an environment where Qubes domains would be implemented as sandboxed containers inside a normal Windows installation without an actual VMM. Of course such system wouldn't be as secure as one using a proper VMM because the attack surface is larger – constricting the whole Windows API is a much more difficult task than using a relatively narrow set of VMM interfaces. However, it would be significantly easier to adapt for users while still being more secure than "just Windows". Thus the idea of Windows Native Isolation for Qubes was born.

2. Windows Native Isolation for Qubes

Windows Native Isolation (or WNI for short) is an attempt to create Qubes version that would use a single Windows instance as a pseudo-hypervisor for separate domains. Since there is no actual VMM involved it requires a custom libvirt driver that would encapsulate WNI-specific functionality and expose it to the Qubes core. This driver would be responsible for implementing containers for actual domain isolation. Let's see what are the requirements for such environment:

- Processes running in an isolated domain can't change state of the host OS or other domains without explicit user action. Restricted behaviors should include:
  - Changing global host OS settings.
  - Cross-domain inter-process communication (IPC) of any kind.
  - Access to file system and Windows registry outside of dedicated domain space.
  - Access to kernel objects created by processes in other domains [4].
  - Access to windows created by processes in other domains.
  - Access to clipboard.
  - Access to physical screen and input devices.
  - Access to network interfaces.
  - Access to other peripheral devices.
- Only the administrative domain (dom0) can create, destroy and manage other domains.
- All visible GUI windows and elements created by a particular domain should have non-spoofable graphical indicators identifying the owning domain.
- Ability to run reasonable number of domains simultaneously.
- Small performance overhead.
- Resistance to DOS attacks in domains by resource exhaustion.
- Applications that do not belong to the Qubes core can't run in the administrative domain.
- Supported Windows editions: Windows 7 and above, 64-bit.

There is a significant amount of research already performed on the subject of Windows application sandboxes [5]. We can utilize some of that research because Qubes WNI is conceptually a set of application containers where domains are security boundaries. Let's examine in details how we can (or cannot) accomplish the requirements outlined above.

3. Technical analysis of Qubes WNI requirements and possible implementations

Implementing basic domain security boundaries

We chose to use separate Windows user accounts as the basis for domain isolation. Each domain is represented by a separate restricted user account that is managed by the Qubes core. Windows security architecture ensures that we can apply discretionary access control to files, registry keys and kernel objects as needed [6].

Administrative domain (dom0) is represented by the interactive user. This user account does not need to have administrative rights as long as the user can securely invoke administrative functions of the Qubes core (implemented as system services and/or kernel drivers). The interactive user owns the graphical shell process (usually explorer.exe) in the interactive session. Applications running in other domains should be clearly marked as so, as stated in the requirements e.g. via use of colorful, non-spoofable window decorations (title bars, frames).

Preventing changes to global host OS settings

Most of this restriction can be implemented by proper Local Security Policy (LSP) [7] definitions and file/registry Access Control Lists (ACLs) [8]. However there are Windows APIs affecting global system state that are not covered by LSP, those require additional steps to ensure they are properly restricted. For example, `SystemParametersInfo` function can be used to change a wide variety of global settings (accessibility options like sticky/filter keys, desktop wallpaper, desktop work area, mouse speed etc.) [9]. Most of that functionality is available to any account without restriction. These settings are usually mapped to an appropriate registry key and as such can be regulated by a proper ACL entries, but this is a) not documented and b) not reliable. Not all of the `SystemParametersInfo` functionality maps to registry keys. So how we can deal with it?

The function is exported by `user32.dll` library, which is the user-mode entry for most APIs that deal with graphical user interface (GUI) and windowing subsystem. Many of the readers are probably thinking "API hooking!" [10] Indeed, by inserting our own code at the API's entry point we can in principle monitor its usage and enforce additional security policies. There is a problem however. Like for most USER APIs, `SystemParametersInfo`'s real functionality is implemented in the kernel-mode module – the infamous `win32k.sys` and the user-mode entry point is just a wrapper that does some preprocessing of the function parameters. Even if we properly hook the user-mode API (which is by no means easy to do without allowing any bypass methods) nothing stops the (potentially malicious) process from calling the kernel-mode API directly by an appropriate *system call* or syscall for short [11]. Here's how the final user-mode stub for our function looks like:

```
NtUserSystemParametersInfo proc near
mov     r10, rcx
mov     eax, 1042h
syscall
retn
NtUserSystemParametersInfo endp
```

Any process can use that code and directly call into `win32k.sys` bypassing our user-mode hook.

Well, why not employ a kernel-mode hook then? For one, implementing such thing is much harder than a user-mode API hook. Any error can easily lead to a *bug check* [12] that brings the whole system down. There is a more fundamental problem though – beginning from Windows Vista (6.0) Microsoft introduced a mechanism called *Kernel Patch Protection* (KPP for short or, informally, *PatchGuard*) [13]. KPP aims to prevent or at least discourage patching the Windows kernel and other critical system areas. `win32k.sys` is not protected by KPP in Windows 7, but it is in Windows 8. Of course it is possible to bypass KPP [14], but it's obvious that a legitimate product should not use hacks and dirty tricks, at the very least because of security and reliability concerns, but also because of potential legal implications.

Job Objects [15] can be used to prevent processes from using `SystemParametersInfo` function, but there are other APIs with system-wide effects that can't be limited that way and are callable from any process (e.g. `SetSysColors` that is also just a system call stub) [16].

Other, non-intrusive solutions are possible but are definitely not trivial. They will be briefly explained in later sections.

Restricting cross-domain IPC

Windows provides a multitude of options for inter-process communication and interaction. In Qubes WNI we need to ensure that no such activity can occur without explicit permission. Below we review all available IPC channels.

Pipes

Named or anonymous pipes are perhaps the best known method for native IPC in Windows environment. They are a simple client-server mechanism for data transfer backed by shared memory. Pipes are implemented as securable objects by the kernel so it's easy to protect them using appropriate ACLs [17].

Mailslots

Not as widely known as pipes, mailslots are used for one-way communication. They are also securable objects and may have ACLs applied to them [18].

Sections (shared memory)

Also known as file mappings on the user-mode side, they can be used to share regions of virtual memory. Sections are securable objects and may have ACLs applied to them [19].

Events, mutexes, semaphores, waitable timers

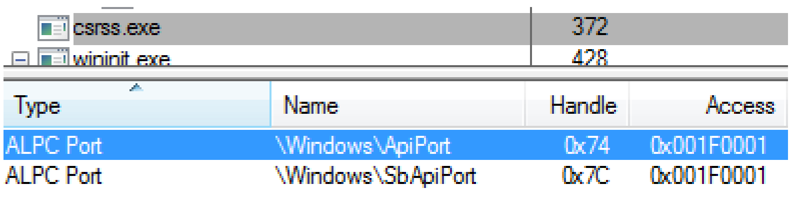
Used for inter-thread and inter-process synchronization. They are all securable objects and may have ACLs applied to them [20], [21], [22], [23].

Debugging and process/thread manipulation APIs

Processes and threads are of course securable. By default processes and threads created by one user do not have access to processes and threads created by another user (unless the requesting process has administrative rights, but WNI domain users do not have administrative rights).

Advanced Local Procedure Call (ALPC)

Often also referred to as Advanced Local Interprocess Communication, ALPC is a high-speed IPC mechanism implemented by the Windows kernel. Its details are undocumented but it's used by many system components for internal communication. ALPC ports are implemented as securable kernel objects [24], [25].



Type	Name	Handle	Access
ALPC Port	\Windows\ApiPort	0x74	0x001F0001
ALPC Port	\Windows\SbApiPort	0x7C	0x001F0001

Fig. 1, ALPC ports owned by the CSRSS process

The problem with ALPC ports is that they are mostly owned by critical system services and are used by various Windows APIs. Access to some of them is needed for successful working of any application. This problem will be described in more detail in the section dealing with kernel objects.

Window messages

GUI processes that create windows can communicate using various window messages. Windows and other GUI elements are *not* securable objects. This will be discussed in detail in the section talking about GUI isolation.

We need to remember that even though kernel objects are securable, applications that create them may not necessarily follow our security policies. Qubes WNI core could change ACLs on existing objects, but applying proper security policies to *newly created* objects would require modifications to the Object Manager (Windows kernel component described below).

Restricting file and registry access

This is simple in principle, because the system registry and NTFS file system are both securable. Practical implementation is not easy though, because we need to create policies with actual ACL values for the registry and file system. WNI domains run as different logon sessions in the same interactive session and special care must be taken to restrict access to resources that are accessible to all interactive processes by default.

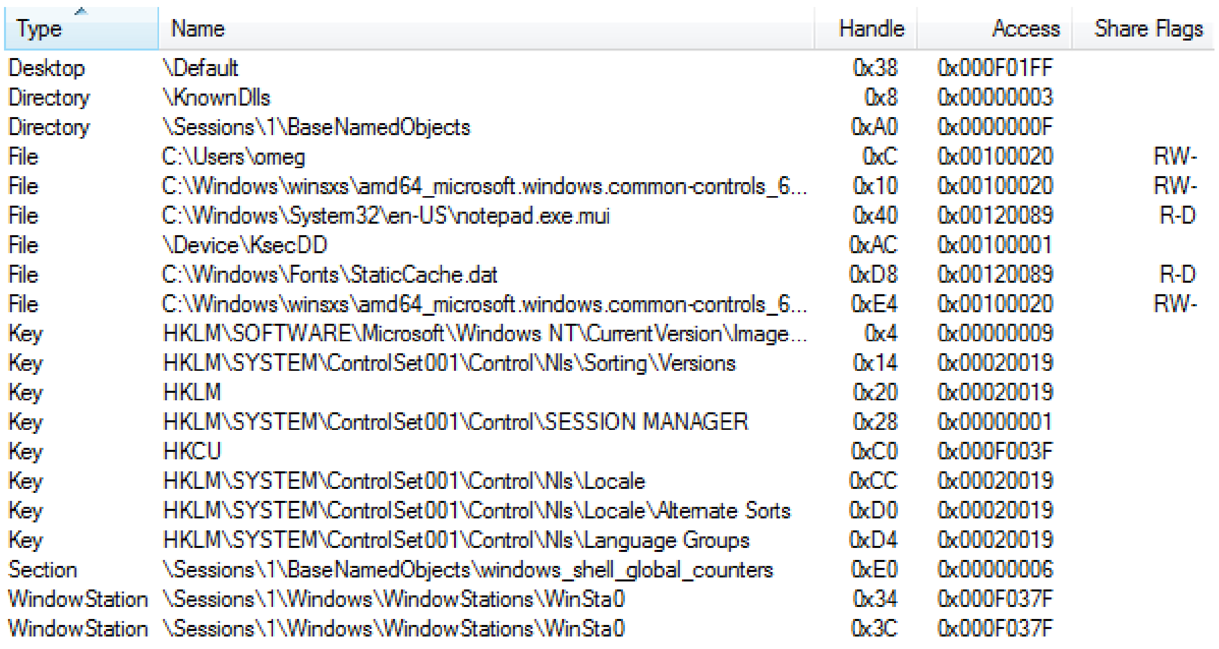
Restricting access to kernel objects

Kernel objects (or executive objects to be more precise) encapsulate various types of resources managed by the executive part of the Windows kernel [26]. Example object types are files, processes, events or desktops. There are 42 object types in a default Windows 7 system, all of them are enumerated below in a dump from a test system:

```
42 possible object types
INDEX     POOL_TYPE ACCESSMASK  COUNT TYPE_NAME
0x02: NonPagedPool 0x001f0001      0 Type
0x03:    PagedPool 0x000f000f    149 Directory
0x04:    PagedPool 0x000f0001      5 SymbolicLink
0x05:    PagedPool 0x001f01ff    241 Token
0x06: NonPagedPool 0x001f001f      3 Job
0x07: NonPagedPool 0x001fffff    319 Process
0x08: NonPagedPool 0x001fffff   1539 Thread
0x09: NonPagedPool 0x000f0003      1 UserApcReserve
0x0a: NonPagedPool 0x000f0003      0 IoCompletionReserve
0x0b: NonPagedPool 0x001f000f      0 DebugObject
0x0c: NonPagedPool 0x001f0003   5615 Event
0x0d: NonPagedPool 0x001f0000      0 EventPair
0x0e: NonPagedPool 0x001f0001    482 Mutant
0x0f: NonPagedPool 0x001f0001      0 Callback
0x10: NonPagedPool 0x001f0003   1238 Semaphore
0x11: NonPagedPool 0x001f0003    217 Timer
0x12: NonPagedPool 0x000f0001      0 Profile
0x13:    PagedPool 0x001f0003     66 KeyedEvent
0x14: NonPagedPool 0x000f037f    100 WindowStation
0x15: NonPagedPool 0x000f01ff     95 Desktop
0x16: NonPagedPool 0x000f00ff    254 TpWorkerFactory
0x17: NonPagedPool 0x001f01ff      0 Adapter
0x18: NonPagedPool 0x001f01ff      0 Controller
0x19: NonPagedPool 0x001f01ff      0 Device
```

```
0x1a: NonPagedPool 0x001f01ff      0 Driver
0x1b: NonPagedPool 0x001f0003    152 IoCompletion
0x1c: NonPagedPool 0x001f01ff   1356 File
0x1d: NonPagedPool 0x000f003f      9 TmTm
0x1e: NonPagedPool 0x001f007f      0 TmTx
0x1f: NonPagedPool 0x001f007f     18 TmRm
0x20: NonPagedPool 0x000f001f      0 TmEn
0x21:    PagedPool 0x001f001f    514 Section
0x22: NonPagedPool 0x001f0003     14 Session
0x23:    PagedPool 0x001f003f   1416 Key
0x24: NonPagedPool 0x001f0001    908 ALPC Port
0x25: NonPagedPool 0x001f0000      5 PowerRequest
0x26: NonPagedPool 0x001f0fff     30 WmiGuid
0x27: NonPagedPool 0x00120fff   2007 EtwRegistration
0x28: NonPagedPool 0x00120fff      6 EtwConsumer
0x29: NonPagedPool 0x001f0001      5 FilterConnectionPort
0x2a: NonPagedPool 0x001f0001      5 FilterCommunicationPort
0x2b:    PagedPool 0x001f0003     22 PcwObject
```

Objects are managed by the *Object Manager* and are made available for use to the Windows API by various system calls (`NtCreateFile`, `NtOpenProcess`, `NtSetEvent`, `NtUserSwitchDesktop` etc). Every running process accesses from a few to thousands of kernel objects.



Type	Name	Handle	Access	Share Flags
Desktop	\Default	0x38	0x000F01FF	
Directory	\KnownDlls	0x8	0x00000003	
Directory	\Sessions\1\BaseNamedObjects	0xA0	0x0000000F	
File	C:\Users\omeg	0xC	0x00100020	RW-
File	C:\Windows\winsxs\amd64_microsoft.windows.common-controls_6...	0x10	0x00100020	RW-
File	C:\Windows\System32\en-US\notepad.exe.mui	0x40	0x00120089	R-D
File	\Device\KsecDD	0xAC	0x00100001	
File	C:\Windows\Fonts\StaticCache.dat	0xD8	0x00120089	R-D
File	C:\Windows\winsxs\amd64_microsoft.windows.common-controls_6...	0xE4	0x00100020	RW-
Key	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image...	0x4	0x00000009	
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	0x14	0x00020019	
Key	HKLM	0x20	0x00020019	
Key	HKLM\SYSTEM\ControlSet001\Control\SESSION MANAGER	0x28	0x00000001	
Key	HKCU	0xC0	0x000F003F	
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Locale	0xCC	0x00020019	
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Locale\Alternate Sorts	0xD0	0x00020019	
Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Language Groups	0xD4	0x00020019	
Section	\Sessions\1\BaseNamedObjects\windows_shell_global_counters	0xE0	0x00000006	
WindowStation	\Sessions\1\Windows\WindowStations\WinSta0	0x34	0x000F037F	
WindowStation	\Sessions\1\Windows\WindowStations\WinSta0	0x3C	0x000F037F	

Fig. 2, Named kernel objects owned by an instance of Notepad

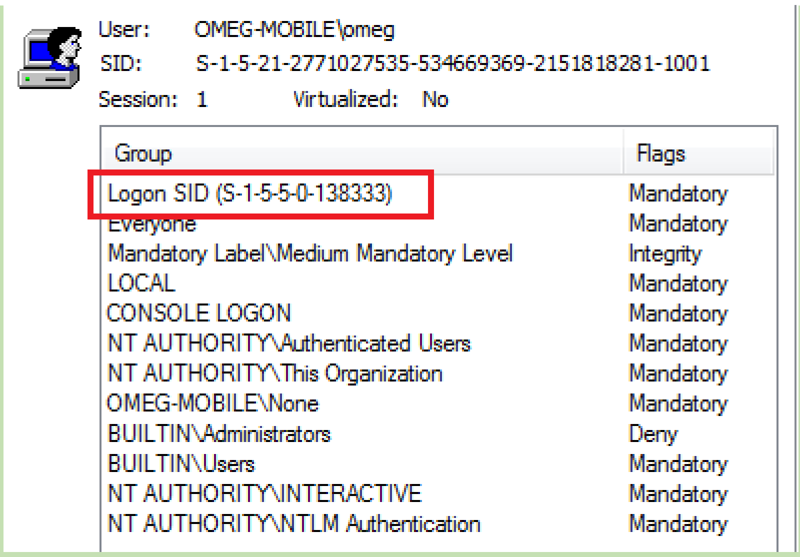
Kernel objects are securable – that means we can protect them with ACLs. Here, problem solved, right? Not really. But to see why we need to clarify a few things first.

Intermission: tokens, permissions, impersonation, oh my

Security Identifier (SID) is a system-wide unique value that identifies a *trustee* – a system account, group or *logon session.* SIDs are primary entities to which access control can be applied [27].

Access token is a kernel object that describes identity and privileges of a process or thread [28]. It contains, amongst other things, SIDs for the owner account and group, and a *logon SID*. Every child process inherits the token of its parent. Assignment and creation of arbitrary tokens is a security-critical operation and requires appropriate privileges, only assigned to the SYSTEM account by default.

Logon SID is a SID that the system creates when a user is authenticated during logon, interactive or otherwise. `LogonUser` function returns an access token that contains a logon SID [29].



User: OMEG-MOBILE\omeg
SID: S-1-5-21-2771027535-534669369-2151818281-1001
Session: 1 Virtualized: No

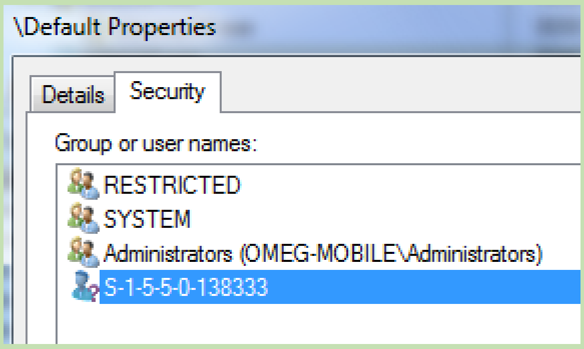
Group	Flags
Logon SID (S-1-5-5-0-138333)	Mandatory
Everyone	Mandatory
Mandatory Label\Medium Mandatory Level	Integrity
LOCAL	Mandatory
CONSOLE LOGON	Mandatory
NT AUTHORITY\Authenticated Users	Mandatory
NT AUTHORITY\This Organization	Mandatory
OMEG-MOBILE\None	Mandatory
BUILTIN\Administrators	Deny
BUILTIN\Users	Mandatory
NT AUTHORITY\INTERACTIVE	Mandatory
NT AUTHORITY\NTLM Authentication	Mandatory

Fig. 3, Logon SID in the access token of Notepad that was launched by the interactive user

Logon session is a collection of processes that have the same logon SID assigned [30]. Usually it's created by Winlogon process during an interactive logon, but can also be started by calling the `LogonUser` function. A logon session is destroyed when the last process referencing its logon SID is terminated.

As we can see, a *user SID* is not equivalent to *logon SID* – if we call `LogonUser` two times with the same user credentials we will get two different logon SIDs that can be subject to different access control rules.

As we said before, Qubes WNI uses separate user accounts as a basis for domain separation. How do we start a process as another user? There are several ways. The most straightforward way is the `CreateProcess…` family of functions. The most convenient of them is `CreateProcessWithLogonW`: it takes both user credentials and executable details and performs both logon and process creation in one step [31]. Perfect! All may seem to work well but there is one minor detail that can potentially ruin our security model: this function doesn't create a new logon SID but uses logon SID *of the caller* instead. Why does this matter? After all, the new process still has a different owner SID. That's true, but most dynamic kernel objects in an interactive session are protected by default with ACLs that rely on *logon SID*. We could add explicit ACCESS_DENIED entries containing the new domain's user SID to all objects but this won't help us secure the future objects created by 3rd party applications.



\Default Properties

Group or user names:

RESTRICTED
SYSTEM
Administrators (OMEG-MOBILE\Administrators)
S-1-5-5-0-138333

Fig. 4, SIDs in a default security descriptor for an interactive desktop

This behavior of `CreateProcessWithLogonW` is not really made clear in its MSDN documentation. It's only mentioned in the following section:

***Windows XP with SP2 and Windows Server 2003**: You cannot call CreateProcessWithLogonW from a process that is running under the "LocalSystem" account, because the function uses the logon SID in the caller token, and the token for the "LocalSystem" account does not contain this SID. As an alternative, use the CreateProcessAsUser and LogonUser functions.*

…but this is true for Windows 7 as well.

The proper way is to use `CreateProcessAsUser` function [32] if our code runs with SYSTEM rights (because that function requires *SeAssignPrimaryToken* privilege) or `CreateProcessWithTokenW` if we "only" have administrative rights [33]. Both of these functions require a separate call to `LogonUser` (that creates a new logon SID) and additional calls to prepare user environment, but the end result is a process with a logon SID different than the interactive logon SID.

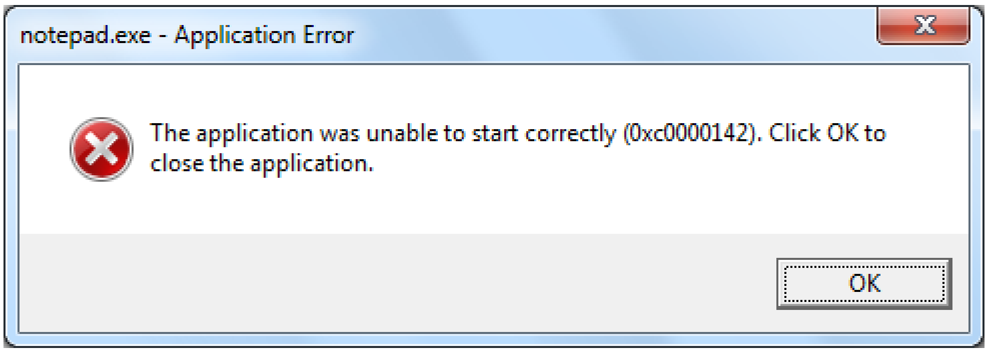
Back to kernel objects. As we've seen above most kernel objects are protected by ACLs that rely on logon SID of the interactive user. That's usually enough because the interactive session is meant to be the real security boundary. If we have several users logged on at the same time to a single system (using Remote Desktop on a server or Fast User Switching on consumer Windows) they cannot access each other's kernel objects normally. That's because interactive sessions implement *kernel object namespace separation* [34].

Named kernel object namespace looks like a tree structure similar to a typical file system. O*bject directories* can contain objects or other directories. There are symbolic links that point to other objects or object directories (object directories are just a special object type). To browse through the object namespace one can use a tool like WinObj [35].

Applications don't have direct access to kernel objects, they typically access them by using appropriate high-level API functions (`CreateFile`, `OpenProcess`, `SetEvent`, `SwitchDesktop` etc.) All those functions operate on *handles*, which are process-specific identifiers mapped to particular objects in process *handle tables*.

Each interactive user session has its own private kernel object namespace to prevent name collisions when multiple users are logged on to the system at the same time. When a process running in session 1 refers to an event named `SomeEvent`, the Object Manager transparently maps that name to `\Sessions\1\BaseNamedObjects\SomeEvent` (starting from Windows 6.0 (Vista) session 0 is reserved for system services and interactive session IDs start at 1).

What stops applications from using fully-qualified object names to access objects from other sessions? Default ACL of the `\Sessions\x\BaseNamedObjects` object directory only allows access to processes with a logon SID that started the current interactive session, administrators and the SYSTEM account. All child objects inherit that ACL by default. It may seem that this is just fine for the purpose of isolating processes that should run as different users in the same interactive session. But is it?


notepad.exe - Application Error

The application was unable to start correctly (0xc0000142). Click OK to close the application.

Fig. 5, Notepad launched by the CreateProcessWithTokenW API as a different user

The error above is a result of a `CreateProcessWithTokenW` call where the token's logon SID is different than the interactive logon SID. 0xC0000142 means "DLL initialization failed" – the process couldn't start because one of the DLLs it depends on failed to initialize. Debugging such problems can be tricky as the process itself never starts – Widows loader aborts the operation. We can guess that the problem most likely has to do with kernel object permissions.

One way to see what fails is by using Global Flags – a global system variable that allows developers to enable various built-in diagnostic mechanisms [36]. One of the options ("Show loader snaps") enables verbose debugging output for the image loader. After setting it using the standalone *GFlags* utility or through debugger (`!gflag +sls` command in WinDbg) we can examine the output and find this:

```
0614:06c4 @ 12725329 - LdrpRunInitializeRoutines - ERROR: Init routine
0000000077256B88 for DLL "C:\Windows\system32\USER32.dll" failed during
DLL_PROCESS_ATTACH
0614:06c4 @ 12725329 - LdrpInitializeProcess - ERROR: Running the init
routines of the executable's static imports failed with status 0xc0000142
```

We know now that USER32's `DllMain` is failing, but we don't know why yet. We know the function's address so we can debug it, but it's trickier than it looks: the actual failure point lies deep inside kernel code.

```
kd> k
Child-SP          RetAddr           Call Site
fffff880`03fa4598 fffff960`000d3463 win32k!UserTestForWinStaAccess
fffff880`03fa45a0 fffff960`000a54be win32k!xxxResolveDesktop+0x42b
fffff880`03fa4950 fffff960`000a220b win32k!xxxCreateThreadInfo+0x89e
fffff880`03fa4b20 fffff960`00071ac3 win32k!UserThreadCallout+0x21b
fffff880`03fa4b70 fffff800`03107de2 win32k!W32pThreadCallout+0xff
fffff880`03fa4bc0 fffff800`028124ba nt!PsConvertToGuiThread+0x1fe
fffff880`03fa4bf0 fffff800`02818fdf nt!KiConvertToGuiThread+0xa
fffff880`03fa4c20 000007fe`ff03704a nt!KiSystemServiceExit+0x1c4
00000000`0025de98 000007fe`fefdd3e4 GDI32!ZwGdiInit+0xa
00000000`0025dea0 00000000`76f47311 GDI32!GdiDllInitialize+0x124
00000000`0025e000 00000000`7705f216 USER32!UserClientDllInitialize+0x749
kd> dt _unicode_string @rcx
ntdll!_UNICODE_STRING
 "\Sessions\1\Windows\WindowStations\WinSta0"
kd> gu
win32k!xxxResolveDesktop+0x42b:
fffff960`000d3463 8bf0            mov     esi,eax
kd> r rax
rax=00000000c0000022
```

Bingo. 0xC0000022 status code means “access denied”. What happens is GDI32.DLL issues a win32k system call during its initialization and the system service dispatcher tries to convert the thread to a GUI thread (basically to perform some initial setup for win32k). Somewhere in that setup code there is a check if the interactive window station is accessible.

Notepad is an application that creates a window. That means it needs to access the interactive desktop and the window station. But the default ACLs on those objects don't allow access to SIDs other than the interactive logon SID, however. We have a problem.

What can we do? Any GUI application needs access to the window station and desktop to run correctly. This is potentially risky because Windows offers little in the way of protecting GUI resources. We will discuss GUI isolation in a later section.

However even if we manage to solve GUI issues or just allow applications running in restricted Qubes domains access to the window station and desktop it will not be enough. Every non-trivial application creates and uses various kernel objects through Windows APIs. As was said before most new kernel objects are created in a session-specific object directory that is not accessible to processes with a logon SID different than the interactive one. Trying to solve that problem and allowing access to `\Sessions\x\BaseNamedObjects` directory may cause security issues though – a process running in one domain can perform a *name squatting* attack. Because the object namespace is shared in such scenario, a malicious process could create an object named just like something that’s normally used by another process, a shared memory section for example, before the “legitimate” process does so. If the malicious process fills said object with malformed data and the second process tries to use the object without proper validation, Bad Things can happen. This attack can also be used as a form of denial of service if the malicious process blocks another process from creating object(s).

As we see, GUI processes need access to the window station/desktop as well as other named objects (mainly ALPC ports) that are created by system services and need to be shared by all interactive processes in a session. Without access to these ports applications may behave incorrectly or just crash, as illustrated on the images below.

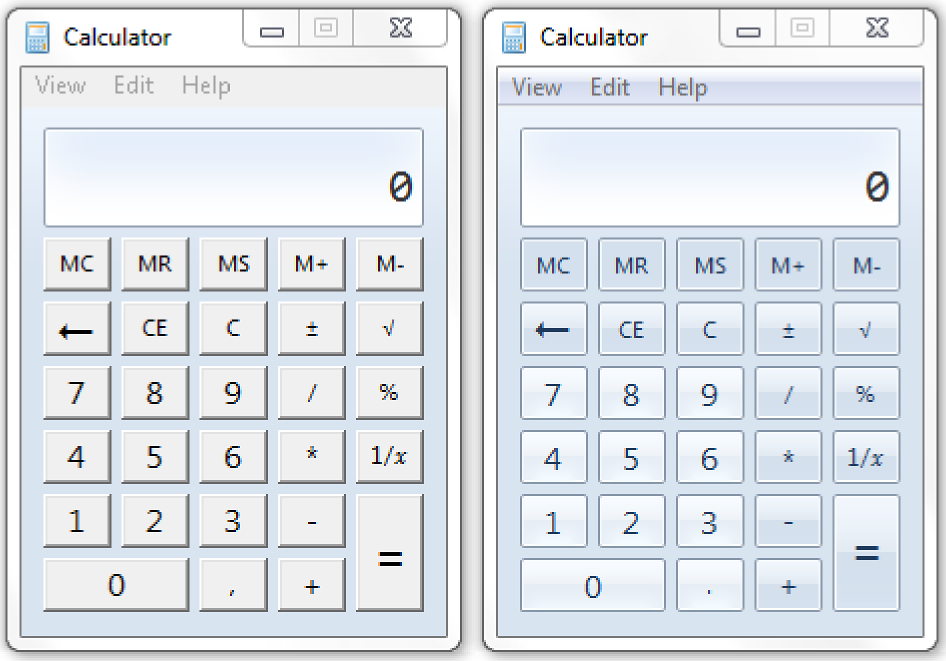


Fig. 6, Window of a process without access to \ThemeApiPort object (left) and with access to that port (right)

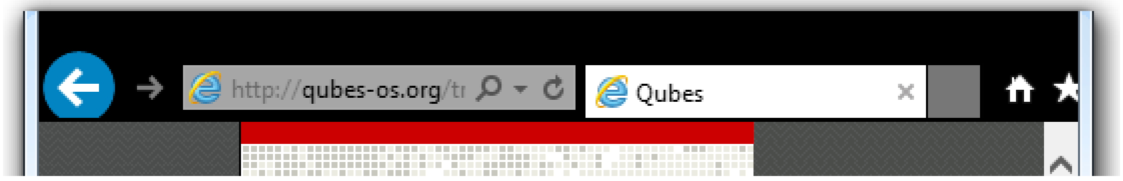


Fig. 7, Internet Explorer's title bar when the process doesn't have access to the Desktop Window Manager's ALPC API port

Ideally we would like to separate object namespaces completely, but that's not really possible without running processes in another interactive session. We *cannot* use separate interactive sessions for WNI domains. Sessions would solve most of our problems with object namespaces, because the separation mechanism is already implemented there. However, consumer Windows versions support only one active session at a time. It's possible to circumvent that limitation but such actions are against Windows EULA. Even if we could use multiple sessions there are fundamental issues preventing us from developing seamless GUI integration in such scenario (discussed in detail later).

What are the possible solutions?

- We can grant access to all these per-session system objects to WNI domains. That, however, would open an unchecked communication channel between domains. Those objects are created in *dom0*, the administrative domain. ALPC port interface is officially undocumented along with actual communication protocols used by system components. Exposing internal windows API ports to untrusted domains is a bad idea, especially when there are documented vulnerabilities in such mechanisms [37].
- We can implement our own object namespace separation. This isn't too hard if we deal with *newly created* objects only (although still requires kernel mode code and some dirty tricks with the Object Manager). Even then the problem with shared system objects remain.
- We can try to create our own implementation of interactive sessions without actually using the system implementation. That would allow us to have multiple copies of system services and processes inside (or alongside) one "real" interactive session. Such task is of course very hard as the session implementation details are not well documented and can change between operating system versions. Our research suggests that this *might* be possible and is an eventual avenue for future attempts at implementing Qubes WNI.

Conclusion: domain kernel object isolation *without significantly impacting usability (or highly non-trivial development)* is an open issue.

Implementing GUI isolation and seamless inter-domain integration

The fact that windows and other GUI elements are not securable is perhaps the biggest flaw in the Windows security model. It's a result of maintaining backwards compatibility with pre-NT editions where security wasn't really considered in system's design. By default, processes running as different users can affect each other by using various windows messages. Any GUI process can potentially spoof things like password input boxes because raw access to the desktop is not restricted – if an application can show its window, it can draw anything on the desktop. Clipboard is shared between all processes belonging to an interactive window station. Processes can synthesize keyboard and mouse input in a way that can affect other processes. Basically, it's a mess.

First, we need some definitions.

Window station is a kernel object that belongs to an interactive session. It contains a clipboard, an atom table, and several desktop objects. Only one window station, named `WinSta0`, is interactive – that means it can receive user input and display user interface [38].

Desktop is a kernel object that belongs to a window station. It contains a logical display surface and GUI objects like windows, menus or hooks [39]. Window messages can only be sent between processes on the same desktop, desktop is also a boundary for *message hooks* [40].

We need to implement three things:

- GUI isolation: cross-domain window messages, hooks, clipboard access etc. should not be possible.
- Seamless GUI integration: even if windows created by other domains are physically on some other desktop/session, we need to display them on the interactive (host) desktop and it should be possible to normally use them.
- Non-spoofable window decorations: all windows belonging to a domain should be clearly marked with appropriate color and domain name.

The image below shows GUI isolation on Xen-based Qubes OS where all the above points are met.

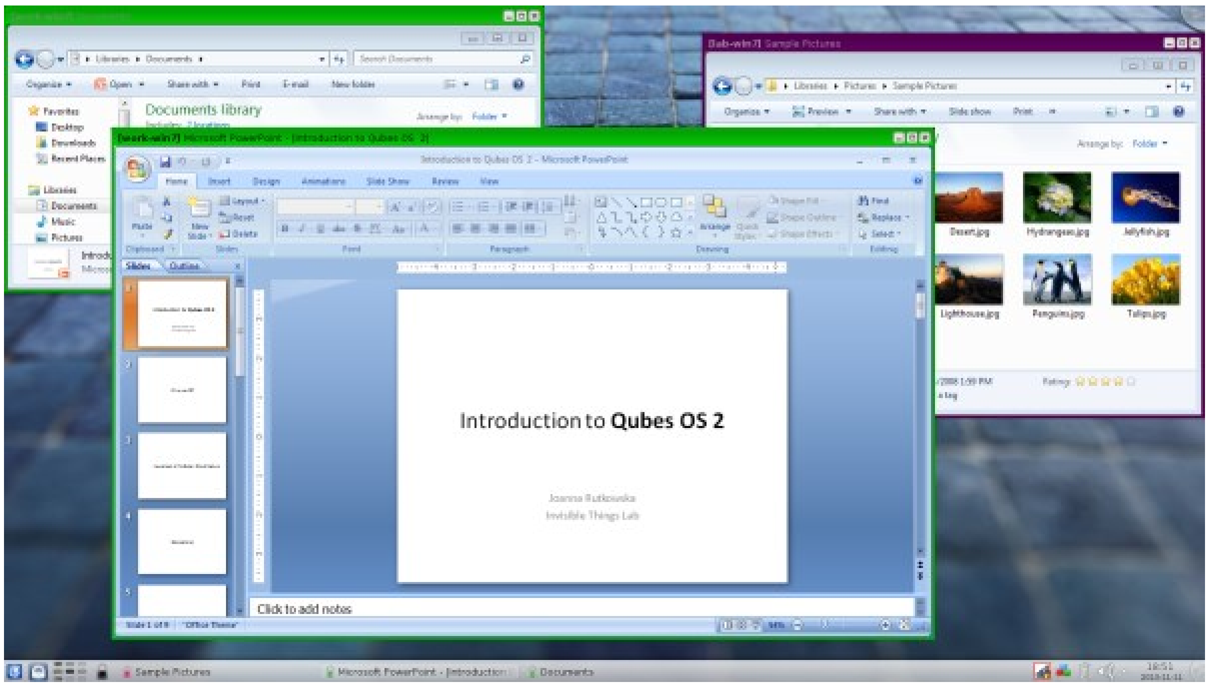


Fig. 8, Windows from multiple domains on a Qubes desktop (Linux host)

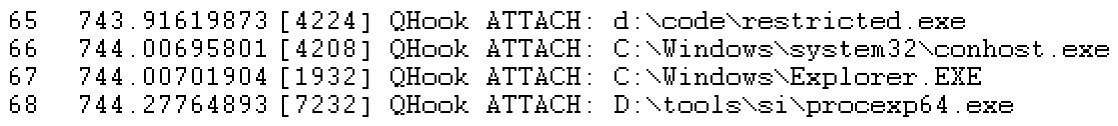
The best solution would be to run each domain in its own interactive session, as discussed in the section where we explained issues with kernel objects. Each session contains an interactive window station and desktop so we would get GUI isolation for free. This is not really possible though:

a) Consumer Windows systems are limited to one active session at a time. Inactive sessions can't be interacted with and they do not render their display contents.
b) There is no documented way to create an interactive session programmatically. It's possible to use Remote Desktop Protocol to connect to the local machine and create a session that way, but again, not on consumer Windows, and RDP sessions differ from "physical" sessions in various ways because display and input devices are virtualized.

What are other options? Only one predefined window station can be interactive so we can't use that as a useful security boundary. Maybe we can use multiple desktops then? Each domain would need to use a separate desktop for its applications. This is not hard to implement but then the task of presenting windows from other domains on a host/*dom0* desktop (like in the image above) is pretty much impossible because of Windows display architecture.

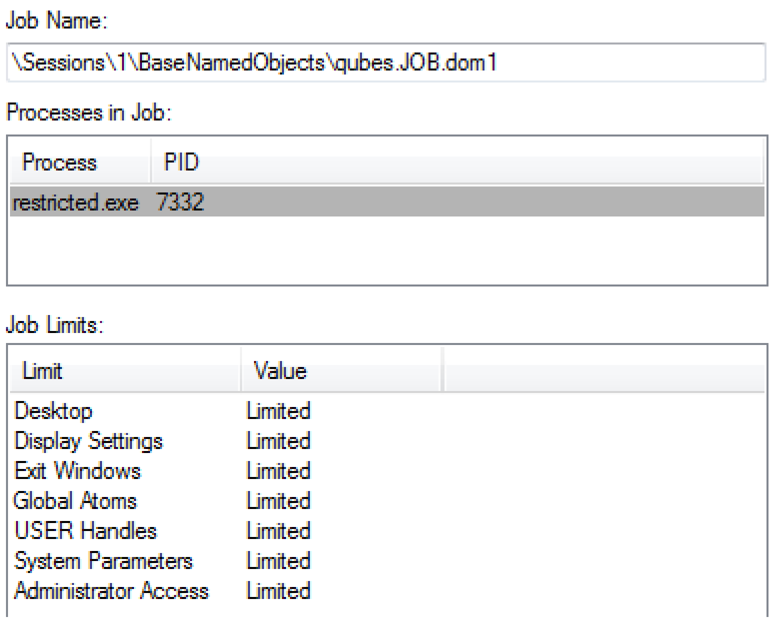
The main problem is that *win32k.sys*, the kernel component responsible for windowing and graphics subsystems, is not capable of simultaneously rendering on multiple desktops at the same time. *Desktops* should not be mistaken for *monitors* – in Windows, a desktop contains monitors, not the other way around. That means we can have a multi-monitor setup, but we can only see one desktop at a time (extending multiple monitors, or mirrored). Only one desktop can be *active* at any time – that is, being actually rendered and running a *raw input thread* that receives user input. This is an architectural limitation that's not avoidable without significant changes to the display/windowing subsystem. Inquisitive readers can examine the inner workings of *win32k.sys* by disassembly and live kernel debugging. ReactOS sources also provide valuable information, although they are not compatible with Windows 7 as of yet [41]. The following paper describes the general architecture of Windows user interface subsystem and several attacks against it [42].

If we can't use separate desktops, can we satisfy our requirements using a single desktop for all domains? We can in principle use *job objects* to contain each domain's processes and impose restrictions on using GUI handles that belong to processes not in that particular job object [43]. Job objects also enable limiting access to *global atoms* – instead of using an atom table shared by all GUI threads belonging to a window station, a job can have its own private atom table. Atom separation increases the overall system security. See e.g. this prior research [44] which describes some attacks possible through a shared atom table. A job object *cannot* restrict setting global message hooks, despite some well-known publications claiming otherwise [45]. This is trivial to demonstrate – images below show that *restricted.exe*, a process with all possible UI restrictions set in its job object, can in fact inject a message hook DLL into other processes. And with arbitrary code execution all bets are off.



```
65   743.91619873 [4224] QHook ATTACH: d:\code\restricted.exe
66   744.00695801 [4208] QHook ATTACH: C:\Windows\system32\conhost.exe
67   744.00701904 [1932] QHook ATTACH: C:\Windows\Explorer.EXE
68   744.27764893 [7232] QHook ATTACH: D:\tools\si\procexp64.exe
```

Fig. 9, Debug output from a hook DLL showing attaches to non-job processes


Job Name:

\Sessions\1\BaseNamedObjects\qubes.JOB.dom1

Processes in Job:

Process	PID
restricted.exe	7332

Job Limits:

Limit	Value
Desktop	Limited
Display Settings	Limited
Exit Windows	Limited
Global Atoms	Limited
USER Handles	Limited
System Parameters	Limited
Administrator Access	Limited

Fig. 10, Job object containing the restricted process

To completely dispel any illusions that a single desktop can contain isolated applications running with different privileges, let's discuss the last sub-requirement of true GUI isolation: *Non-spoofable window decorations. All windows belonging to a domain should be clearly marked with appropriate color and domain name.* This is of course aimed to stop attacks where a malicious application disguises itself as a legitimate password-entry field and similar. Unfortunately if an application can display its window on a desktop, it can obtain a handle to this desktop and use that handle for any drawing operations. This means drawing over other windows and generally tampering with our hypothetical "GUI supervisor" that would draw "secure" window decorations.

Similarly, any window that can receive mouse and keyboard input can simulate such input [46] and possibly craft malicious data affecting other windows. Malicious applications can also intercept input data, for example logging keystrokes through `GetAsyncKeyState` function (which is just a wrapper for a system call, like many other APIs, so restricting it is very difficult) [47].

With Windows Vista Microsoft introduced *User Interface Privilege Isolation (UIPI)* [48] – a mechanism designed to mitigate *shatter attacks* [49]. Shatter attacks are a way of injecting arbitrary code into other processes by using specific window messages. UIPI uses *integrity levels* [50] – special flags in access tokens that specify mandatory access control level. Process with low integrity level can't send most window messages to another process with medium integrity level for example. UIPI mechanism is not a security boundary however – some messages are still allowed. `WM_KEYDOWN` message can cross integrity level boundaries and it can adversely affects the receiving application [51]. Also there are only a few integrity levels available for use, which is not enough for arbitrary number of WNI domains. Lastly, most off-the-shelf applications will not run correctly under low integrity level which is not acceptable for Qubes WNI [52].

It's also worth mentioning that Windows clipboard is shared by all desktops in the interactive window station, so even if we could implement proper multi-desktop environment this would be a problem. Clipboard is implemented in *win32k.sys* so regulating access to it is burdened by the usual gotchas regarding kernel code modification [53].

Conclusion: GUI isolation with seamless domain integration is not currently possible. It may be possible if we abandon the idea of having only one host desktop with domain-created windows seamlessly appearing on and use switchable desktops for each domain. Even then there would be as of yet unresolved issues (clipboard access).

Restricting access to network interfaces and peripheral devices

This requirement is possible to fulfill by creating custom device filter drivers. *NDIS filters* [54] can be used to monitor and filter network traffic. Access to other devices can in principle be restricted in similar way. This was not investigated further because previous unresolved issues are enough to make Qubes WNI impossible to implement to a satisfactory level.

Other requirements

Job objects can be used to limit domain's CPU usage to prevent DOS attacks. Remaining requirements were not analyzed in detail due to severity of previous issues.

4. Summary

Existing successful Windows application sandboxes (e.g. Chromium [5] or Acrobat Reader [45]) rely on the fact that the sandboxed process is specifically designed to run inside a sandbox. In a *supervisor/client* model, the non-privileged client expects to run on a different desktop and with low privileges in its access token for example, and needs to explicitly ask the supervisor to perform any privileged operations. Primary use case for Qubes WNI is isolating unmodified applications, and even supervisor/client sandbox model that's implemented only in user mode can't satisfy *all* of our requirements (main offenders are system calls that affect global system state but are not subject to ACL checks).

In short, Qubes WNI can't be currently implemented due to following architectural limitations of Windows systems:

- Inability to properly isolate Kernel Object spaces for different security domains.
- Limitations in the display and windowing subsystems preventing simultaneous use of more than one desktop at a time. *One desktop to rule them all* seems to be an accurate description of this issue. ☺
- Lack of system-provided mechanism for API and system call restriction. We have some ideas on how to implement such restrictions without the need for patching the OS kernel, but this requires further substantial research.

It's worth mentioning that pretty much all of these issues can be solved by using a server Windows version with Remote Desktop service license. Interactive session is a true security boundary, but multiple active sessions are only supported through Remote Desktop Services. Qubes WNI was supposed to be a solution for consumers – ease of deployment on an existing Windows installation was a primary requirement. Not many people will agree to reinstall their system from scratch *and* purchase an additional license that is orders of magnitude more expensive than their previous one. And even if they did, then we should remember that Windows Server systems come with built-in bare metal hypervisor, Hyper-V, which we could then employ as container provider for Qubes to get orders or magnitude better isolation than any Windows ACLs mechanisms could potentially provide [55].

In closing, Qubes WNI project is put on hold until we come up with some breakthroughs that could solve issues outlined above.

Appendix

Sample WinDbg script that demonstrates parsing internal *win32k.sys* structures. Usage:

```
$$>a< path\to\script\file n
```

Where *n* is the ID of an interactive session that should be analyzed.

```
$$ t0: session id (script parameter)
r $t0 = ${$arg1}

$$ t1: first process
r $t1 = nt!PsActiveProcessHead

$$ t2: current process
r? $t2 = @$t1

.printf "session: %N\n", @$t0

$$ Search for a window station in the given session.
$$ This can probably be done easier using some global win32k variable...

.do
{
    $$ t3: EPROCESS
    r? $t3 = (nt!_EPROCESS*) (@@masm(@$t2) - #FIELD_OFFSET(nt!_EPROCESS,ActiveProcessLinks))
    $$ Go to the next process.
    r? $t2 = @$t3->ActiveProcessLinks.Flink
    $$ Set context to current process.
    .process /r /p @$t3
    $$ t4: process name
    r? $t4 = @$t3->ImageFileName
    $$ t5: win32 process info
    r? $t5 = (win32k!tagPROCESSINFO*) @$t3->Win32Process
    .if (@$t5 != 0)
    {
        $$ t6: window station
        r? $t6 = @$t5->rpwinsta
        .if (@$t6 != 0)
        {
            $$ t7: session id
            r? $t7 = @$t6->dwSessionId
            .if (@$t7 == @$t0)
            {
                .printf "\nGot process in session %N, %ma\n", @$t0, @$t4
                .break
                $$ now $t6 is a tagWINDOWSTATION, $t5 is tagPROCESSINFO
            }
        }
    }
}
(@$t2 != @$t0)

$$ t9: sizeof(OBJECT_HEADER)
r? $t9 = #FIELD_OFFSET(nt!_object_header,Body)
$$ t9: (negative) offset of OBJECT_HEADER_NAME_INFO for a given object
r? $t9 = @$t9 + sizeof(nt!_object_header_name_info)

$$ t8: first window station
r? $t8 = @$t6

.if (@$t7 == @$t0)
{
    $$ t0: first desktop object
    r? $t0 = @$t6->rpdeskList
    $$ t1: current desktop object
    r? $t1 = @$t0

    $$ Loop through all window stations (t6 = current).
    .do
    {
        r? $t4 = (nt!_object_header_name_info*)(@@masm(@$t6)-@@masm(@$t9))
        $$ %mu expects null-terminated wide string, but buffer in UNICODE_STRING may not be
null-terminated.
        $$ %msu is supposed to take a UNICODE_STRING address but doesn't seem to work at
all...
```

```
        .printf ">WINSTA: %mu\n", @@c++(@$t4->Name.Buffer)

        $$ Loop through all desktops in the window station.
        .do
        {
            r? $t2 = @$t1->pDeskInfo
            r? $t3 = @$t1->pDispInfo
            r? $t4 = (nt!_object_header_name_info*)(@@masm(@$t1)-@@masm(@$t9))
            .printf "Desktop: %mu, %N, id %N, flags %N\n", @@c++(@$t4->Name.Buffer), @$t1,
@@c++(@$t1->dwDesktopId), @@c++(@$t1->dwDTFlags)
            .printf "   Base: %N, window: %N, composited: %N\n", @@c++(@$t2->pvDesktopBase),
@@c++(@$t2->spwnd), @@c++(@$t2->fComposited)
            .printf "   hDev: %N, pmdev: %N, hDevInfo: %N\n", @@c++(@$t3->hDev), @@c++(@$t3-
>pmdev), @@c++(@$t3->hDevInfo)
            .printf "   hdcScreen: %N, hdcBits: %N, dmLogPixels: %N\n", @@c++(@$t3-
>hdcScreen), @@c++(@$t3->hdcBits), @@c++(@$t3->dmLogPixels)
            .printf "   cMonitors: %N, primary: %N\n", @@c++(@$t3->cMonitors), @@c++(@$t3-
>pMonitorPrimary)
            r? $t1 = @$t1->rpdeskNext
        }
        (@$t1 != 0 & @$t0 != @$t1)

        r? $t6 = @$t6->rpwinstaNext
    }
    (@$t6 != 0 & @$t6 != @$t8)
}
.else
{
    .printf "No desktops in session %N\n", @$t0
}
```

References

[1] https://qubes-os.org/
[2] http://theinvisiblethings.blogspot.com/2013/03/introducing-qubes-odyssey-framework.html
[3] http://libvirt.org/
[4] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724485(v=vs.85).aspx
[5] http://www.chromium.org/developers/design-documents/sandbox
[6] http://msdn.microsoft.com/en-us/library/windows/desktop/aa379557(v=vs.85).aspx
[7] http://msdn.microsoft.com/en-us/library/windows/desktop/ms721785(v=vs.85).aspx
[8] http://msdn.microsoft.com/en-us/library/windows/desktop/aa374872(v=vs.85).aspx
[9] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724947(v=vs.85).aspx
[10] http://en.wikipedia.org/wiki/Hooking
[11] http://en.wikipedia.org/wiki/System_call
[12] http://msdn.microsoft.com/en-us/library/windows/hardware/ff551961(v=vs.85).aspx
[13] http://msdn.microsoft.com/en-us/library/windows/hardware/gg487353.aspx
[14] http://www.uninformed.org/?v=3&a=3&t=sumry
[15] http://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx
[16] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724940(v=vs.85).aspx
[17] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365780(v=vs.85).aspx
[18] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365576(v=vs.85).aspx
[19] http://msdn.microsoft.com/en-us/library/windows/desktop/aa366556(v=vs.85).aspx
[20] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682655(v=vs.85).aspx
[21] http://msdn.microsoft.com/en-us/library/windows/desktop/ms684266(v=vs.85).aspx
[22] http://msdn.microsoft.com/en-us/library/windows/desktop/ms685129(v=vs.85).aspx
[23] http://msdn.microsoft.com/en-us/library/windows/desktop/ms687012(v=vs.85).aspx
[24] http://en.wikipedia.org/wiki/Local_Procedure_Call
[25] http://blogs.msdn.com/b/ntdebugging/archive/2007/07/26/lpc-local-procedure-calls-part-1-architecture.aspx
[26] http://en.wikipedia.org/wiki/Object_Manager_(Windows)
[27] http://msdn.microsoft.com/en-us/library/windows/desktop/aa379571(v=vs.85).aspx
[28] http://msdn.microsoft.com/en-us/library/windows/desktop/aa374909(v=vs.85).aspx
[29] http://msdn.microsoft.com/en-us/library/windows/desktop/aa378184(v=vs.85).aspx
[30] http://msdn.microsoft.com/en-us/library/windows/desktop/aa378338(v=vs.85).aspx
[31] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682431(v=vs.85).aspx
[32] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682429(v=vs.85).aspx
[33] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682434(v=vs.85).aspx
[34] http://www.nynaeve.net/?p=86
[35] http://technet.microsoft.com/en-us/sysinternals/bb896657.aspx
[36] http://msdn.microsoft.com/en-us/library/windows/hardware/ff549557(v=vs.85).aspx
[37] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1209
[38] http://msdn.microsoft.com/en-us/library/windows/desktop/ms687096(v=vs.85).aspx
[39] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682573(v=vs.85).aspx
[40] http://msdn.microsoft.com/en-us/library/windows/desktop/ms632589(v=vs.85).aspx
[41] http://doxygen.reactos.org/dir_7fafc42db1aeb26e18427fe0bd7122f8.html
[42] http://www.mista.nu/research/mandt-win32k-paper.pdf
[43] http://msdn.microsoft.com/en-us/library/windows/desktop/ms684152(v=vs.85).aspx
[44] http://mista.nu/research/smashing_the_atom.pdf
[45] http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-—-part-2-—-the-sandbox-process.html
[46] http://msdn.microsoft.com/en-us/library/windows/desktop/ms646310(v=vs.85).aspx
[47] http://msdn.microsoft.com/en-us/library/windows/desktop/ms646293(v=vs.85).aspx
[48] http://en.wikipedia.org/wiki/User_Interface_Privilege_Isolation

[49] http://en.wikipedia.org/wiki/Shatter_attack
[50] http://msdn.microsoft.com/en-us/library/bb625957.aspx
[51] http://theinvisiblethings.blogspot.com/2007/02/running-vista-every-day.html
[52] http://msdn.microsoft.com/en-us/library/bb625960.aspx
[53] http://blogs.msdn.com/b/ntdebugging/archive/2012/03/16/how-the-clipboard-works-part-1.aspx
[54] http://msdn.microsoft.com/en-us/library/windows/hardware/ff556030(v=vs.85).aspx
[55] http://technet.microsoft.com/library/hh831531.aspx