Kernel Data and Filtering Support for Windows Server 2008

Document Version: 1.62 (May 1, 2007)

Contacts:
Kernel Patch Protection Input (*kppinput@microsoft.com)*

1 Introduction

This is an initial draft made available for review and feedback. There may be changes to the content and functionality after ISV feedback is received. Furthermore, this document is limited in scope to changes being considered for Microsoft Windows Server® 2008. The APIs were determined based on technical discussions with the ISV community and were constructed while following the Kernel Evaluation Criteria

There are a number of management and security applications that need the ability to access kernel information or filter system calls. Some of these operations involve objects like processes and threads. Traditionally, ISVs have resorted to hooking the Object Manager (OB) APIs in the System Service Dispatch Table (SSDT) whenever they wanted to monitor/modify/block operations that are not supported by existing filtering interfaces provided by the operating system. Since hooking the SSDT was never officially supported and could potentially destabilize the system, it is beneficial for the platform to provide documented, supported ways of filtering and accessing kernel information. Also, with the introduction for *Kernel Patch Protection* in x64 platform, ISVs are no longer able to hook the SSDT as they did in 32-bit x86 systems.

2 Scope

A number of conference calls and face-to-face meetings have taken place with ISVs over the past several months, to understand the underlying technical requirements that ISVs have for specific security and management functions. This document reflects the current top requests for new functionality based on discussions with several dozen security ISVs.

The primary areas identified during the ISV discussions are as follows:

- Create & open process/thread control
- Memory write access control
- Load control of images containing executable code
- Self-Protection of processes containing security software

Each functional area will be addressed in this document.

3 Definitions and Assumptions

This document assumes that the reader is familiar with the current filtering model exposed by the registry and the file-system.

KPP – Kernel Patch Protection
OB – Object Manager
PS – Process Management
SSDT – System Service Dispatch Table
WHQL - Windows Hardware Quality Labs

4 Requirements and Constraints

To reduce the chance of misuse of the APIs by malicious software, several requirements will need to be satisfied in order to utilize the API, and several constraints will be placed on the level of filtering possible in the API.

Kernel mode software that is using the new API must be digitally signed, on 32-bit and 64-bit systems, which enables identification and some degree of integrity. Likewise, if there is user mode code that configures or provides policy to kernel mode software, it too must be digitally signed. The code needs to be digitally signed consistent with the Authenticode signing procedures for mandatory driver signing on x64 platforms. This process does not currently require the usage of WHQL for certification purposes -- the ISV may sign their own code without submitting it to Microsoft.

- All ISV kernel and user mode modules must be linked with the **/integritycheck** flag set at link time. This will cause the memory manager to enforce a signature check at load time for the referenced code. This link.exe setting is present in recent versions of the Microsoft Visual Studio® linker. If the ISV is using alternate tools to link or edit their produced binaries, this flag setting has the effect of setting (ON) IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY (0x0800) in the image PE header OptionalHeader.DllCharacteristics field.
- Digitally signed user mode modules must have cryptographic per-page hash catalogs. These are generated using the /PH command-line setting to the signtool utility.
- The following link contains more information on the code-signing process, which is already mandatory for kernel mode software that loads on x64 based systems: http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx

If ISVs employ a user mode system service that is used to configure or control kernel mode software employing the new API, that system service must utilize the new Windows Service Hardening framework introduced in Microsoft Windows Vista®. This will allow the service to employ least-privilege programming principals, in addition to allowing the ISV to govern who is granted access to service/driver specific resources. For example, the Windows Service Hardening framework can be used to enforce that only the ISV provided user mode service has access to a driver object surfacing IOCTL interfaces.

- The microsoft.com site has information on the new service hardening framework, as does the following whitepaper: http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Vista_Services.doc

Any user mode system services should also be digitally signed.

ISV supplied callback function entry points must reside within valid code segments in the ISV provided digitally signed driver. It is not permissible to provide callback addresses that reside in dynamically allocated memory.

For OB filtering the proposed design allows waiting on and signaling events; it is critical that any OB calls are done in the separate thread outside of the callout. The callout can synchronize using an event but note that lengthy synchronous waits due to extensive processing on worker threads will be a delay in the application/system performance. ISVs should also consider using timeout based wait operations in the callback. To avoid issues with deadlocking, care should be taken to minimize the amount of processing done in the callout directly. For cases where the processing requires waiting on state changes (e.g, polling for policy changes), it should be offloaded to a user mode service and the callout should continue asynchronously.

It is the responsibility of the ISV to determine their preferred behavior in error conditions. For example, if a low-memory condition is encountered, the ISV may choose to fail secure or fail open (where on detecting an error an action may be allowed or deferred to a later time). Note the operating system will not define this, as ISV requirements around failure handling may differ, or may be based on security policy. For the purposes of illustration, the ISV may encounter a low memory condition during their callback processes. The following two cases indicate possible handling modes:

1. The ISV chooses to fail secure. For Ob callback processing, the ISV may choose to filter out specific access bits. For Ps callback processing, the ISV may choose to block the process creation operation.
2. The ISV chooses to fail open. In this particular case, the ISV does not filter or block the operation, and may instead opt to log an event or alert the user. This alerting may be queued/deferred until the low-memory condition subsides.

Programs which are using the API must be discoverable by the administrator. A standard administrative uninstall procedure needs to be supported. The code needs to be visible during kernel module enumeration operations.

It must <u>not</u> be possible to create a situation where processes or resources are completely invisible to the administrator. For example, all processes should be visible in the Task Manager.

It must be possible to perform security scanning and compliance checking operations (e.g., anti-virus scanning, integrity checking, etc.) against code using the new APIs, as well as processes/code that is being influenced by the new API.

Future enhancements to the APIs may come at a later date than Windows Server 2008, including requirements that higher assurance (e.g., EV code signing) certificates must be used to sign code, as well as additional code signing enforcement and associated requirements for security software. ISVs will be notified of any new requirements and have an opportunity to comment before they are introduced.

Additional API usage criteria may also be outlined in the “Kernel Evaluation Criteria” document that is published by Microsoft.

5 Create & Open Process/Thread Control

A high priority ISV request concerned the ability to block create calls and filter out DesiredAccess properties for open calls. Filtering the DesiredAccess property at the time of the Open request enables blocking of other API calls that operate on open handles. Supporting this involves API additions to the Object Manager to support OpenProcess and DuplicateHandle and API enhancements to the existing Process Management APIs.

5.1 OB Filtering Overview

To enable the existing security and management applications, the Object Manager will provide new callbacks to allow filtering operations on specific objects. The remainder of this section outlines the scope of this filtering mechanism and the supported objects and operations that can be filtered. The appendix of this document contains detailed structure definitions pertinent to the OB filtering API.

The OB filtering model will support -

- **Layered model to allow multiple filter drivers to co-exist –** The OB filter manager will allow multiple filter drivers to be registered simultaneously. For a given driver, the presence of drivers in other layers will be logically hidden by providing seamless call transitions between the layers (all the way through to the native OB subsystem).

- **Ability to modify parameters to *OpenProcess* and *OpenThread* calls –** A filter driver will be able to intercept calls to the *OpenThread* and *OpenProcess* APIs and modify the *DesiredAccess* parameter before the call is passed on to next layer below. This allows the caller to reduce the allowed handle access permissions to a process or thread object, which has the effect of blocking specific "downstream" operations on an open handle value. Note that it is not possible to entirely block the Open calls; this avoids creating a situation where invisible or unmanageable processes exist in the system, as mentioned in Section 4. (See Limitations section for more info about restrictions on modifying DesiredAccess)

- **Prevent Re-entry from callbacks** - Making OB calls (directly or indirectly) from a callback will be prohibited.

5.2 OB Architecture Overview

5.2.1 Callback Registration

A driver can register filtering callbacks by calling *ObRegisterCallbacks*. The altitude architecture exposed by the OB will be consistent with the existing filesystem/registry filtering model and will have the following properties.

- **The altitude specified during the registration will uniquely identify the position of the callbacks in the stack** - The higher the altitude, the further away the callbacks are from the native OB code.

- **No two registrations can have the same altitude** - The first callback registration with the given altitude wins. All other registrations for the same altitude will fail. Note that the OB filter manager will not perform any verification to validate if the caller is the approved owner of the particular altitude.

- **Pre and Post Registration** - A driver can register separate callbacks for pre and post operation notification. It can also choose to register one without the other.

- **No upper limit on the number of simultaneous registrations** - Altitudes will usually fall within one of the pre-defined altitude categories (The Filesystem team has already defined a set that should cover most scenarios). To allow for future extension of the size of any given category, the altitude is a real (discrete) number represented as a string. It may contain decimal points (though it is not expected this will ever be used). For instance a callback that registered at altitude 1.11 will be higher on the stack than the callback that registered with 1.1. Similarly categories can be added or deleted based on future need.

- **Multiple altitudes simultaneously** – The same callback can be registered at multiple altitudes simultaneously.

- **Routines can be registered per object-type** - Drivers can register callbacks for any combination of the supported operations (currently supported operations are handle create, handle open and handle duplicate) for any of the supported object types (currently supported object types are process and thread).

This filtering model will require ISVs to request a specific altitude on the stack. The existing process used by ISVs for registering and obtaining a specific altitude on the file-system/registry stack (http://whdc.microsoft.com/minifilter/default.aspx) has been extended to include the OB support. Once an ISV submits a request for a specific altitude via the registration website, based on their requirements, Microsoft will allocate a unique altitude from one of the existing categories (or will create a new one if none exists).

The callbacks can be unregistered by calling *ObUnRegisterCallbacks*. OB will ensure that all currently executing callbacks have completed before the callback is unregistered.

It is the responsibility of the driver that calls *ObRegisterCallbacks* to call *ObUnRegisterCallbacks* before the driver is unloaded.

ObRegisterCallbacks

The **ObRegisterCallbacks** routine adds a list of driver-supplied callback routines to a list of routines to be called whenever a process/thread is created or accessed.

```
NTSTATUS
ObRegisterCallbacks (
    IN POB_CALLBACK_REGISTRATION CallBackRegistration,
    OUT PVOID *RegistrationHandle
);
```

Parameters

CallBackRegistration
A pointer to a registration structure that contains information for registering callbacks for operation filtering including filter altitude, a registration context, and an array of operations for which callbacks are to be defined. The supplied callback routine addresses must reside within a code section of a validly loaded and digitally signed kernel driver. It is not permissible to supply callback addresses that reside in non-paged pool or other dynamically allocated memory.

RegistrationHandle
A pointer to a PVOID that receives a handle that identifies this instance of the registration. When the callbacks are unregistered, the value should be passed in to ObUnRegisterCallbacks.

Return Value

ObRegisterCallbacks can return one of the following:

STATUS_SUCCESS
The given callbacks are now registered with the system.

STATUS_FLT_INSTANCE_ALTITUDE_COLLISION
The calling driver or another driver has already registered callbacks for the specified altitude.

STATUS_INVALID_PARAMETER
One or more of the parameters specified in the registration was invalid. This error may be returned, for instance, if incorrect versions are specified, or if registration is attempted for object types that do not support callbacks.

STATUS_INSUFFICIENT_RESOURCES
An attempt to allocate memory failed.

Headers

Declared in TBD

Comments

The **ObRegisterCallbacks** routine is available only on Microsoft Windows Server 2008 and later operating systems.

A driver must de-register all callback routines before it unloads. The callbacks can be deregistered by calling the **ObUnRegisterCallbacks** function.

See Also

ObUnRegisterCallbacks

ObUnRegisterCallbacks

The **ObUnRegisterCallbacks** routine unregisters all *Callback* routines that an **ObRegisterCallbacks** call previously registered.

```
VOID
ObUnRegisterCallbacks (
    IN PVOID RegistrationHandle
);
```

Parameters

RegistrationHandle
A PVOID value that identifies the callback registration. **ObRegisterCallbacks** provided this value when a driver registered the callbacks.

Return Value

None**.**

Headers

Declared in tbd

Comments

The **ObUnRegisterCallbacks** routine is available only on Microsoft Windows Server 2008 and later operating systems.

A driver that calls **ObRegisterCallbacks** should call **ObUnRegisterCallback** before the driver is unloaded.

See Also

ObRegisterCallbacks

5.2.2 Callback Invocation

The filter manager supports layered operation by passing notifications sequentially down the callback stack. Since the filter manager notifies every driver that has registered for callbacks with a unique pre-notification, from the perspective of a given callback, the presence of other callbacks above and below it on the stack would be logically abstracted by the filter manager.

5.2.2.1 Callback Routine

The pre-notifications are invoked before the actual operation is processed the OB, while the post-notifications are invoked after the call is processed by OB but before it is returned to the caller.

Post-notifications are informational only. The return code or parameters cannot be changed.

5.2.2.2 Modes of Operation

The filter can either modify or just monitor the parameters. The filter manager passes the same buffer down the stack, and before calling the native API, only the changes to parameters allowed to be modified are picked up. The pre-operation callback must return a status of OB_PREOP_SUCCESS.

5.2.2.3 Callback Context

The filtering model also includes support for associating a callback context with each pre and post-notification. This is achieved by including a *pvoid CallContext* in each notification structure. Each driver is free to store its callback specific context here. Since context can change as the call traverses through the stack, the filter manager guarantees that a context stored by any given callback in a pre-notification will be the same context used in the corresponding post-notification for that callback.

The following describes the context and relevant information for the callbacks:
OB OPEN_HANDLE will be handled by OB Create_Handle

OB CREATE_HANDLE

Callback context: Notify routine is called and executed in the context of the originating process and thread

Originating (calling) Process and Thread: PsGetCurrentProcess, PsGetCurrentProcessId, PsGetCurrentThread, PsGetCurrentThreadId can be used in the callout

Process or Thread to be created: The object pointer of the Process or Thread to create/open is provided in OB_PRE_CREATE_HANDLE_INFORMATION.

Notes: The OB functions are agnostic to the object type so the object pointer is provided and not the ProcessID or ThreadID as all objects do not have IDs.

OB DUPLICATE_HANDLE

Callback context: Notify Routine is called and executed in the context of the originating process and thread

Originating (calling) Process and Thread: PsGetCurrentProcess, PsGetCurrentProcessId, PsGetCurrentThread, PsGetCurrentThreadId can be used in the callout

Process or Thread handle to be duplicated: The process in which the object pointer with the Process or Thread to duplicate is provided by SourceProcess in OB_PRE_DUPLICATE_HANDLE_INFORMATION.

Notes: The destination process that will receive the duplicated handle is provided by TargetProcess in OB_PRE_DUPLICATE_HANDLE_INFORMATION.

Pre Callback Routines

A driver's pre callback routine can monitor or modify an object operation.

```
typedef OB_PREOP_CALLBACK_STATUS
( *POB_PRE_OPERATION_CALLBACK) (
__IN PVOID RegistrationContext,
__INOUT POB_PRE_OPERATION_INFORMATION OperationInformation
);
```

Parameters

RegistrationContext
The value that the driver passed as the *RegistrationContext* parameter (in the registration OB_REGISTRATION structure) when it registered this routine. This will allow context information to be passed on a per-altitude basis.

OperationInformation
A pointer to a data structure which contains the type of operation, as well as input parameters for the operation that this callback is being invoked for. Some of these parameters can be modified by the callback.

Return Value

An enumerated value. Currently, the only supported values is OB_PREOP_SUCCESS.

Headers

Declared in TBD.

Comments

To be notified of process/thread operations, a kernel-mode component (such as the driver component of an anti-malware software package) can call **ObRegisterCallbacks** to register callbacks for these operations.

The OB callback is always made in the context of the originating process & thread.

See Also

Post Callback Routines

A driver's post callback routine can monitor an object operation.

```
typedef VOID
( *POB_POST_OPERATION_CALLBACK) (
__IN PVOID RegistrationContext,
__IN POB_POST_OPERATION_INFORMATION OperationInformation
);
```

Parameters

RegistrationContext
The value that the driver passed as the *RegistrationContext* parameter (in the registration OB_REGISTRATION structure) when it registered this routine. This will allow context information to be passed on a per-altitude basis.

OperationInformation
A pointer to a data structure which contains the type of operation as well as the some parameters related to the operation that was just performed.

Return Value

None.

Headers

Declared in TBD.

Comments

To be notified of process/thread operations, a kernel-mode component (such as the driver component of an anti-malware software package) can call **ObRegisterCallbacks** to register callbacks for these operations.

The OB callback is always made in the context of the originating process & thread.

See Also

5.2.3 Limitations

In the current version, the only input parameter that can modified (when operating in the modify mode) is the *DesiredAccess* ACCESS_MASK for create, open, and duplicate operations on handles for processes and threads. In addition it can only be modified to request a subset of the ACCESS_MASK that was passed in to the callback. A read-only copy of the originally requested *DesiredAccess* is also provided to the callback. Parameters may not be modified if the original call is made from kernel mode.

5.3 PS Extended Callbacks

An enhanced process creation callback that provides additional flexibility during process creation will be provided. Additional information will be provided in the callback including: full path image name and the command line. In addition to the information in the callback, the call will enable recipients to block process creation such that no user mode code will ever run.

As in the existing system, any driver that receives a creation callback will receive the termination callback. However, the converse is not true; a driver may receive a termination callback for a thread or process for which it never saw the creation. This could occur if the driver registers the callback after a process has been created, or an earlier callback vetoes the process creation (via return value from CREATE_PROCESS_NOTIFY_ROUTINE_EX), thereby prematurely ending the notification processing.

The limit on the maximum number of callbacks will also be extended beyond the current 8 slot limit to 32.

The thread creation and termination callbacks will remain unchanged.

The following describes the context and relevant information for the callbacks:
PsSetCreateProcessNotifyRoutineEx

Callback context: Notify Routine is called and executed in the context of the originating process and thread (but subject to change in future OSes)

Originating (calling) Process and Thread: ProcessID and ThreadID are provided by CreatingThreadId in PS_CREATE_NOTIFY_INFO.

Newly created Process: Both the Process pointer and ProcessID of the newly created process are passed as arguments to the callout routine

Process ParentID: Passed as argument to callout in PS_CREATE_NOTIFY_INFO structure.

Notes: No user code will be executed before the callout is made. The callout is made just after the initial thread is created for the new process. Under Windows Server 2008, if a veto is made by a callout, PsSetCreateProcessNotifyRoutines callouts that have not been executed will not be executed. The pointer to the newly created Process is provided for efficiency in not having to look this up in the callback.

PsSetCreateProcessNotifyRoutineEx (delete process)

Callback context: Notify Routine is called and executed in the context of the last remaining thread and the process that is being deleted.

Originating (calling) Process and Thread: Not available

Process to be deleted: Both the Process pointer and ProcessID are passed as arguments to the callout routine

Process ParentID: Passed as argument to callout in PS_CREATE_NOTIFY_INFO structure.

Notes: Callback is called (deletion indicated) when the last thread is marked for deletion and the process is to be deleted. There is no way to block the process termination at this point. The callback is made before the thread and process is deleted (so it is PRE-operation in that sense), but the callback is notified only meaning it can not block the termination event (so it is POST-operation in the sense of callback control).

PsSetCreateThreadNotifyRoutine

Callback context: Notify Routine is called and executed in the context of the originating process and thread (but subject to change in future OSes)

Originating (calling) Process and Thread: PsGetCurrentProcess, PsGetCurrentProcessId, PsGetCurrentThread, PsGetCurrentThreadId can be used in the callout

Newly created Thread: The ProcessID and newly created ThreadID are passed as arguments to the callout routine.

Notes: No changes to this API are planned for Windows Server 2008

PsSetCreateThreadNotifyRoutine (delete thread)

Callback context: Notify Routine is called and executed in the context of the thread that is to be deleted after system call completes

Originating (calling) Process and Thread: Not available

Thread to be deleted: The ProcessID and ThreadID of the thread to be deleted are passed as arguments to the callout routine

Notes: No changes to this API are planned for Windows Server 2008.

PsSetCreateProcessNotifyRoutineEx

The **PsSetCreateProcessNotifyRoutineEx** routine adds a driver-supplied callback routine to, or removes it from, a list of routines to be called whenever a process is created or deleted.

```
NTSTATUS
  PsSetCreateProcessNotifyRoutineEx(
    IN PCREATE_PROCESS_NOTIFY_ROUTINE_EX  NotifyRoutine,
    IN BOOLEAN  Remove
    );
```

Parameters

NotifyRoutine
Specifies the entry point of a caller-supplied process-creation callback routine.

Remove
Indicates whether the routine specified by *NotifyRoutine* should be added to or removed from the system's list of notification routines. If FALSE, the specified routine is added to the list. If TRUE, the specified routine is removed from the list.

Return Value

PsSetCreateProcessNotifyRoutineEx can return one of the following:

STATUS_SUCCESS
The given *NotifyRoutine* is now registered with the system.

STATUS_INVALID_PARAMETER
The given *NotifyRoutine* has already been registered so this is a redundant call, or the system has reached its limit for registering process-creation callbacks.

Headers

Declared in ntddk.h. Include ntddk.h

Comments

Highest-level drivers can call **PsSetCreateProcessNotifyRoutineEx** to set up their process creation notify routines, declared as follows:

```
VOID
(*PCREATE_PROCESS_NOTIFY_ROUTINE_EX) (
   __inout PEPROCESS Process,
   __in HANDLE ProcessId,
   __in_opt PPS_CREATE_NOTIFY_INFO CreateInfo
   );
```

The *Process* and *ProcessID* parameters identify the new process, and the *CreateInfo* parameter contains information on the process creation and also indicates whether the process was created or deleted.

An IFS or highest-level system-profiling driver might register a process-creation callback to track the system-wide creation and deletion of processes against the driver's internal state. The system can register up to 32 process-creation callbacks.

A driver must remove any callbacks it registers before it unloads. You can remove the callback by calling **PsSetCreateProcessNotifyEx** with Remove = TRUE.

The driver is called with *CreateInfo* defined as:

```
typedef struct _PS_CREATE_NOTIFY_INFO {
    __in SIZE_T Size;
    union {
        __in ULONG Flags;
        struct {
            __in ULONG FileOpenNameAvailable : 1;
            __in ULONG Reserved : 31;
        };
    };
    __in HANDLE ParentProcessId;
    __in CLIENT_ID CreatingThreadId;
    __inout struct _FILE_OBJECT *FileObject;
    __in PCUNICODE_STRING ImageFileName;
    __in_opt PCUNICODE_STRING CommandLine;
    __out NTSTATUS CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;
typedef CONST PS_CREATE_NOTIFY_INFO *PCPS_CREATE_NOTIFY_INFO;
```

After a driver-supplied routine is registered, it is called with a non-NULL CreateInfo just after the initial thread is created within the newly created process designated by the input *ProcessId* handle. The input *ParentProcessId* handle identifies the parent process of the newly created process. Note that the parent process may not be the same as the creating process, since Windows Vista (and other Microsoft Windows NT® based systems) allows a process to be created from a template process (for inheritance of certain attributes). Note that a driver can instruct the kernel to halt the start of the new process by writing a failure NTSTATUS code to the *CreationStatus* field of the structure.

A driver's process-notify routine can also be called with a NULL CreateInfo to indicate that the callback is being done in a terminate path. The callback is made when the last thread within a process has terminated and the process address space is about to be deleted.

If FileOpenNameAvailable is set, the *ImageFileName* Unicode String specifies the exact file name used to open the process executable file. If this flag is absent, a partial name is specified. In either case, the file object actually backing the memory mapped section for the executable is specified, and the driver can retrieve additional properties through filter manager APIs like *FltGetFileNameInformationUnsafe*.

Callers of **PsSetCreateProcessNotifyRoutineEx** must be running at IRQL = PASSIVE_LEVEL.

See Also

6 Memory Write Access Control

The specific ISV requests in this area pertain to the ability to block calls to the WriteProcessMemory() and associated API. Microsoft has investigated this issue and has had detailed conversations with ISVs, and concluded that providing fine-grained API filtering on WriteProcessMemory() is not required. However, Microsoft is supporting blocking of these operations through alternate means, which can also be used to prevent a large range of other issues, such as that of creating a remote thread in a monitored process.

To provide additional context, there are a series of functions that we term “upstream” and others referred to as “downstream”. The downstream functions operate on handles returned from upstream functions. For example, TerminateProcess() would be a downstream API operation that depends on a handle returned by an upstream call to OpenProcess() with at least PROCESS_TERMINATE access. By filtering the DesiredAccess granted during the upstream OpenProcess() operation, effective controls can be made on a wide class of downstream functions, such as CreateRemoteThread(), WriteProcessMemory(), and TerminateProcess().

By filtering the DesiredAccess property with the new OB callouts, a higher level, broader range of control is possible. Some examples of the DesiredAccess that are pertinent to memory write access control and process/thread termination are as follows:

- PROCESS_VM_WRITE
- PROCESS_VM_OPERATION
- PROCESS_TERMINATE
- THREAD_TERMINATE

The ability to query and maintain visibility and control within the kernel over processes must be maintained to prevent “runaway” and invisible processes containing malicious software. As a result, it will not be possible to filter or block “query” class operations. Furthermore, the operating system kernel and kernel mode driver software will always have the ability to terminate any process on the system.

7 Load image of executable code

Several ISVs have requested the ability to block the loading of code modules. Released Windows operating system versions do support notification of image loading operations (which does meet some ISV requirements), but do not provide the ability to block the loading operation. For that reason, Microsoft investigated whether a new API would be needed to support this requirement, but ultimately concluded that existing supported functionality could be used to achieve the desired module load blocking behavior.

In particular, a file system mini-filter can be utilized to block the loading of both modules in both user mode (e.g., DLLs) and kernel mode (e.g., device drivers). Intercepting IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION and returning STATUS_ACCESS_DENIED when sections are loaded for PAGE_EXECUTE permission is an appropriate approach. The file-system mini-filter support is documented

at microsoft.com, and a link to a whitepaper is as follows:
http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/FilterDriverDeveloperGuide.doc

As a side note, Microsoft is investigating deprecating usage of the undocumented and unsupported routine NtLoadDriver() from any user mode caller except for the Windows Service Control Manager.

8 Self-protection

Protecting processes hosting security software was the second highest priority requirement from ISVs. This requirement stemmed from the desire to reduce the chances that security software is terminated or interfered with by malicious software. In several cases, the sole reason behind ISV patching of the operating system kernel was done for the purpose of implementing self-protection of security software. Microsoft has investigated the requirements around this functionality, and concluded that the combination of using the new API and existing Windows API can be used to implement various protection controls for processes containing security software. Examples of where that is applicable are as follows:

- The new API for filtering open process/thread operations, which can be used to impede process termination and tampering.
- Control over code loading into the security process as afforded by a mini-filter, which can be used to prevent the loading of unexpected code.
- Protection of registry resources as afforded by the existing registry configuration manager callbacks, to prevent manipulation of security configuration data.
- Usage of the Windows Service Hardening framework to limit access and privilege to private security service state.

Microsoft is also investigating generalizing the notion of a "protected process", for third-party ISV usage. This support would require the ISV to "authenticate" into the protected process, as well as imposing other constraints. The operating system kernel would perform the policy enforcement constraints. This particular support will be investigated for a delivery vehicle beyond Windows Server 2008, and is generally the model that Microsoft would like to adopt. Such a model may be adopted in specific areas over multiple operating system releases, and may ultimately lead to the deprecation of existing kernel functions. ISVs will be apprised well-in-advance of any changes in this area.

9 Appendix

9.1 Structure Definitions

The following section lists the structure definitions for callback registration and invocation.

OB_CALLBACK_REGISTRATION

The OB_CALLBACK_REGISTRATION structure is used as a parameter to ***ObRegisterCallbacks***.

```
typedef struct _OB_CALLBACK_REGISTRATION {
    __in USHORT                     Version;
    __in USHORT                     OperationRegistrationCount;
    __in UNICODE_STRING             Altitude;
    __in PVOID                      RegistrationContext;
    __in OB_OPERATION_REGISTRATION  *OperationRegistration;
} OB_CALLBACK_REGISTRATION, *POB_CALLBACK_REGISTRATION;
```

Members

Version
Provides version that this filter supports. Filters should set this to OB_FLT_REGISTRATION_VERSION.

OperationRegistrationCount
The number of operations registered via the OperationRegistration array.

Altitude
Altitude assigned to the driver making the registration.

RegistrationContext
A context that the driver can associate with this instance of the registration. The context will be passed to all the callbacks the driver has registered at this altitude.

OperationRegistration
A pointer to an array of structures that registers individual sets of operations. This is defined in detail below.

OB_OPERATION_REGISTRATION

The OB_OPERATION_REGISTRATION is used to register callbacks per object type.

```
typedef struct _OB_OPERATION_REGISTRATION {
    __in POBJECT_TYPE                *ObjectType;
    __in OB_OPERATION                Operations;
    __in POB_PRE_OPERATION_CALLBACK  PreOperation;
    __in POB_POST_OPERATION_CALLBACK PostOperation;
} OB_OPERATION_REGISTRATION, *POB_OPERATION_REGISTRATION;
```

Members

ObjectType
A pointer to the object type for which the callbacks are being registered. Currently supported types are Process and Thread. Drivers should use *PsProcessType* and *PsThreadType*.

Operation
Specifies the operations for which callbacks are desired. Must be a combination (bitwise OR) of the flags OB_OPERATION_CREATE and OB_OPERATION_DUPLICATE.

PreOperation
A pointer to a driver supplied function that will be called by the filter manager before a registered operation takes place.

PostOperation
A pointer to a driver supplied function that will be called by the filter manager after a registered operation takes place.

OB_PRE_OPERATION_INFORMATION

The OB_PRE_OPERATION_INFORMATION is populated by the filter manager and is a parameter to the pre-operation callback.

```
typedef struct _OB_PRE_OPERATION_INFORMATION {
    __in OB_OPERATION           Operation;
    union {
        __in ULONG Flags;
        struct {
            __in ULONG KernelHandle:1;
            __in ULONG Reserved:31;
        };
    };
    __in PVOID                  Object;
    __in POBJECT_TYPE           ObjectType;
    __out PVOID                 CallContext;
    __in POB_PRE_OPERATION_PARAMETERS  Parameters;
} OB_PRE_OPERATION_INFORMATION, *POB_PRE_OPERATION_INFORMATION;
```

Members

Operation
The operation that is being filtered. One of OB_OPERATION_CREATE, OB_OPERATION_OPEN, or OB_OPERATION_DUPLICATE.

Flags
A bitfield of flags that gives information about the current operation

Object
The object that the operation is being performed on.

ObjectType
The type of object that the operation is being performed on.

CallContext
A context that the driver can associate with this operation. This context will be sent to the post callback for this particular pre callback. If the driver allocates any memory for this context in the pre callback, it must free that memory in the associated post callback.

Parameters
A pointer to a union of structures that encapsulate parameters specific to each of the operations that can be filtered. The Operation member indicates which structure is populated.

OB_PRE_OPERATION_PARAMETERS

The OB_PRE_OPERATION_PARAMETERS union encapsulates the parameters the driver needs to filter the handle create operation

```
typedef union _OB_PRE_OPERATION_PARAMETERS {
    __inout OB_PRE_CREATE_HANDLE_INFORMATION        CreateHandleInformation;
    __inout OB_PRE_DUPLICATE_HANDLE_INFORMATION     DuplicateHandleInformation;
} OB_PRE_OPERATION_PARAMETERS, *POB_PRE_OPERATION_PARAMETERS;
```

Members

CreateHandleInformation
Parameters for handle create and open.

DupliateHandleInformation
Parameters for handle duplication.

OB_PRE_CREATE_HANDLE_INFORMATION

The OB_PRE_CREATE_HANDLE_INFORMATION structure encapsulates the parameters the driver needs to filter the handle create operation

```
typedef struct _OB_PRE_CREATE_HANDLE_INFORMATION {
    __inout ACCESS_MASK         DesiredAccess;
    __in ACCESS_MASK            OriginalDesiredAccess;
} OB_PRE_CREATE_HANDLE_INFORMATION, *POB_PRE_CREATE_HANDLE_INFORMATION;
```

Members

DesiredAccess
The access that is about to be granted to the caller. The driver can modify this parameter in keeping with the rules described earlier.

OriginalDesiredAccess
A read-only copy of the DesiredAccess requested by the originating caller.

OB_PRE_DUPLICATE_HANDLE_INFORMATION

The OB_PRE_DUPLICATE_HANDLE_INFORMATION structure encapsulates the parameters the driver needs to filter the handle duplicate operation.

```
typedef struct _OB_PRE_DUPLICATE_HANDLE_INFORMATION {
    __inout ACCESS_MASK         DesiredAccess;
    __in ACCESS_MASK            OriginalDesiredAccess;
    __in PVOID                  SourceProcess;
    __in PVOID                  TargetProcess;
} OB_PRE_DUPLICATE_HANDLE_INFORMATION, * POB_PRE_DUPLICATE_HANDLE_INFORMATION;
```

Members

DesiredAccess
The access that is about to be granted to the caller. The driver can modify this parameter in keeping with the rules described earlier.

OriginalDesiredAccess
A read-only copy of the DesiredAccess requested by the originating caller.

SourceProcess
The process from which the handle is being duplicated.

TargetProcess
The process to which the handle is being duplicated.

OB_POST_OPERATION_INFORMATION

The OB_POST_OPERATION_INFORMATION is a parameter to the post-operation callback

```
typedef struct _OB_POST_OPERATION_INFORMATION {
    __in OB_OPERATION  Operation;
    union {
        __in ULONG Flags;
        struct {
            __in ULONG KernelHandle:1;
            __in ULONG Reserved:31;
        };
    };
    __in PVOID         Object;
    __in POBJECT_TYPE  ObjectType;
    __in PVOID         CallContext;
    __in NTSTATUS      ReturnStatus;
    __in POB_POST_OPERATION_PARAMETERS  Parameters;
} OB_POST_OPERATION_INFORMATION,*POB_POST_OPERATION_INFORMATION;
```

Members

Operation
The operation being filtered.

Flags
A bitfield of flags that gives information about the current operation.

Object
The object on which the operation was performed.

ObjectType
The type of the object on which this operation was performed.

CallContext
The call context that was set by the driver in the pre-operation callback. If no context was set, this is NULL.

ReturnStatus
The status of the operation that was performed.

Parameters
A pointer to a union of structures that encapsulate parameters specific to each of the operations that was filtered. The Operation member indicates which structure is populated. This pointer is valid only when the ReturnStatus value is a success status.

OB_POST_OPERATION_PARAMETERS

The OB_POST_OPERATION_PARAMETERS union encapsulates the information supplied to the driver regarding the handle create operation that was just performed.

```
typedef union _OB_POST_OPERATION_PARAMETERS {
    __in OB_POST_CREATE_HANDLE_INFORMATION       CreateHandleInformation;
    __in OB_POST_DUPLICATE_HANDLE_INFORMATION    DuplicateHandleInformation;
} OB_POST_OPERATION_PARAMETERS, *POB_POST_OPERATION_PARAMETERS;
```

Members

CreateHandleInformation
Parameters for handle create and open.

DuplicateHandleInformation
Parameters for handle duplication.

OB_POST_CREATE_HANDLE_INFORMATION

The OB_POST_CREATE_HANDLE_INFORMATION structure encapsulates the information supplied to the driver regarding the handle create operation that was just performed.

```
typedef struct _OB_POST_CREATE_HANDLE_INFORMATION {
    __in ACCESS_MASK            GrantedAccess;
} OB_POST_CREATE_HANDLE_INFORMATION, *POB_POST_CREATE_HANDLE_INFORMATION;
```

Members

GrantedAccess
The access that was granted to the handle.

OB_POST_DUPLICATE_HANDLE_INFORMATION

The OB_POST_DUPLICATE_HANDLE_INFORMATION structure encapsulates the information supplied to the driver regarding the handle duplicate operation that was just performed.

```
typedef struct _OB_POST_DUPLICATE_HANDLE_INFORMATION {
    __in ACCESS_MASK            GrantedAccess;
} OB_POST_DUPLICATE_HANDLE_INFORMATION, * POB_POST_DUPLICATE_HANDLE_INFORMATION;
```

Members

GrantedAccess
The access that was granted to the handle.

9.2 DesiredAccess Property Controls

The following section outlines the DesiredAccess bits that are permissible to filter in the Open/Duplicate callback interfaces:

```
PROCESS_CREATE_PROCESS
PROCESS_CREATE_THREAD
PROCESS_DUP_HANDLE
PROCESS_SET_QUOTA
PROCESS_SET_INFORMATION
PROCESS_SUSPEND_RESUME
PROCESS_TERMINATE
PROCESS_VM_OPERATION
PROCESS_VM_WRITE

THREAD_DIRECT_IMPERSONATION
THREAD_IMPERSONATE
THREAD_SET_CONTEXT
THREAD_SET_INFORMATION
THREAD_SET_LIMITED_INFORMATION
THREAD_SET_THREAD_TOKEN
THREAD_SUSPEND_RESUME
THREAD_TERMINATE
```