Contents

Driver Security Guidance

12/15/2022 • 2 minutes to read • Edit Online

This section contains information on enhancing driver security.

In this section

TOPIC	DESCRIPTION
Driver security checklist	This topic provides a driver security checklist for driver developers.
Threat modeling for drivers	Driver writers and architects should make threat modeling an integral part of the design process for any driver. This topic provides guidelines for creating threat models for drivers.
Windows security model for driver developers	This topic describes how the Windows security model applies to drivers and explains what driver writers must do to improve the security of their devices.
Evaluate HVCI driver compatibility	This topic describes how to implement Hypervisor-protected Code Integrity (HVCI) compatible code.
Development security practices for Windows driver developers	This topic describes how to write secure code for Windows drivers to prevent abuse and tampering by malicious actors.

Driver security checklist

12/15/2022 • 33 minutes to read • Edit Online

This article provides a driver security checklist for driver developers to help reduce the risk of drivers being compromised.

Driver security overview

A security flaw is any flaw that allows an attacker to cause a driver to malfunction in such a way that it causes the system to crash or become unusable. In addition, vulnerabilities in driver code can allow an attacker to gain access to the kernel, creating a possibility of compromising the entire OS. When most developers are working on their driver, their focus is on getting the driver to work properly, and not on whether a malicious attacker will attempt to exploit vulnerabilities within their code.

After a driver is released, however, attackers can attempt to probe and identify security flaws. Developers must consider these issues during the design and implementation phase in order to minimize the likelihood of such vulnerabilities. The goal is to eliminate all known security flaws before the driver is released.

Creating more secure drivers requires the cooperation of the system architect (consciously thinking of potential threats to the driver), the developer implementing the code (defensively coding common operations that can be the source of exploits), and the test team (proactively attempting to find weakness and vulnerabilities). By properly coordinating all of these activities, the security of the driver is dramatically enhanced.

In addition to avoiding the issues associated with a driver being attacked, many of the steps described, such as more precise use of kernel memory, will increase the reliability of your driver. This will reduce support costs and increase customer satisfaction with your product. Completing the tasks in the checklist below will help to achieve all these goals.

Security checklist: *Complete the security task described in each of these topics.*

❒ Confirm that a kernel driver is required

❒ Use the driver frameworks

❒ Control access to software only drivers

❒ Do not production sign test driver code

❒ Perform threat analysis

❒ Follow driver secure coding guidelines

❒ Implement HVCI compatible code

❒ Follow technology specific code best practices

❒ Perform peer code review

❒ Manage driver access control

❒ Enhance device installation security

❒ Execute proper release driver signing

❒ Use code analysis in Visual Studio to investigate driver security

❒ Use Static Driver Verifier to Check for Vulnerabilities

❒ Check code with BinSkim Binary Analyzer

❒ Use code validation tools

❒ Review debugger techniques and extensions

❒ Understand how drivers are reported using the Microsoft Vulnerable and Malicious Driver Reporting Center

❒ Review secure coding resources

Summary of key takeaways

Confirm that a kernel driver is required

Security checklist item #1: *Confirm that a kernel driver is required and that a lower risk approach, such as Windows service or app, is not a better option.*

Drivers live in the Windows kernel, and having an issue when executing in kernel exposes the entire operating system. If any other option is available, it likely will be lower cost and have less associated risk than creating a new kernel driver. For more information about using the built in Windows drivers, see Do you need to write a driver?.

For information on using background tasks, see Support your app with background tasks.

For information on using Windows Services, see Services.

Use the driver frameworks

Security checklist item #2: *Use the driver frameworks to reduce the size of your code and increase its reliability and security.*

Use the Windows Driver Frameworks to reduce the size of your code and increase its reliability and security. To get started, review Using WDF to Develop a Driver. For information on using the lower risk user mode framework driver (UMDF), see Choosing a driver model.

Writing an old fashion Windows Driver Model (WDM) driver is more time consuming, costly, and almost always involves recreating code that is available in the driver frameworks.

The Windows Driver Framework source code is open source and available on GitHub. This is the same source code from which the WDF runtime library that ships in Windows 10 is built. You can debug your driver more effectively when you can follow the interactions between the driver and WDF. Download it from https://github.com/Microsoft/Windows-Driver-Frameworks.

Control access to software only drivers

Security checklist item #3: *If a software-only driver is going to be created, additional access control must be implemented.*

Software-only kernel drivers do not use plug-and-play (PnP) to become associated with specific hardware IDs, and can run on any PC. Such a driver could be used for purposes other than the one originally intended, creating an attack vector.

Because software-only kernel drivers contain additional risk, they must be limited to run on specific hardware (for example, by using a unique PnP ID to enable creation of a PnP driver, or by checking the SMBIOS table for the presence of specific hardware).

For example, imagine OEM Fabrikam wants to distribute a driver that enables an overclocking utility for their

systems. If this software-only driver were to execute on a system from a different OEM, system instability or damage might result. Fabrikam's systems should include a unique PnP ID to enable creation of a PnP driver that is also updatable through Windows Update. If this is not possible, and Fabrikam authors a Legacy driver, that driver should find another method to verify that it is executing on a Fabrikam system (for example, by examination of the SMBIOS table prior to enabling any capabilities).

Do not production sign test code

Security checklist item #4: *Do not production code sign development, testing, and manufacturing kernel driver code.*

Kernel driver code that is used for development, testing, or manufacturing might include dangerous capabilities that pose a security risk. This dangerous code should never be signed with a certificate that is trusted by Windows. The correct mechanism for executing dangerous driver code is to disable UEFI Secure Boot, enable the BCD "TESTSIGNING", and sign the development, test, and manufacturing code using an untrusted certificate (for example, one generated by makecert.exe).

Code signed by a trusted Software Publishers Certificate (SPC) or Windows Hardware Quality Labs (WHQL) signature must not facilitate bypass of Windows code integrity and security technologies. Before code is signed by a trusted SPC or WHQL signature, first ensure it complies with guidance from Creating Reliable Kernel-Mode Drivers. In addition the code must not contain any dangerous behaviors, described below. For more information about driver signing, see Release driver signing later in this article.

Examples of dangerous behavior include the following:

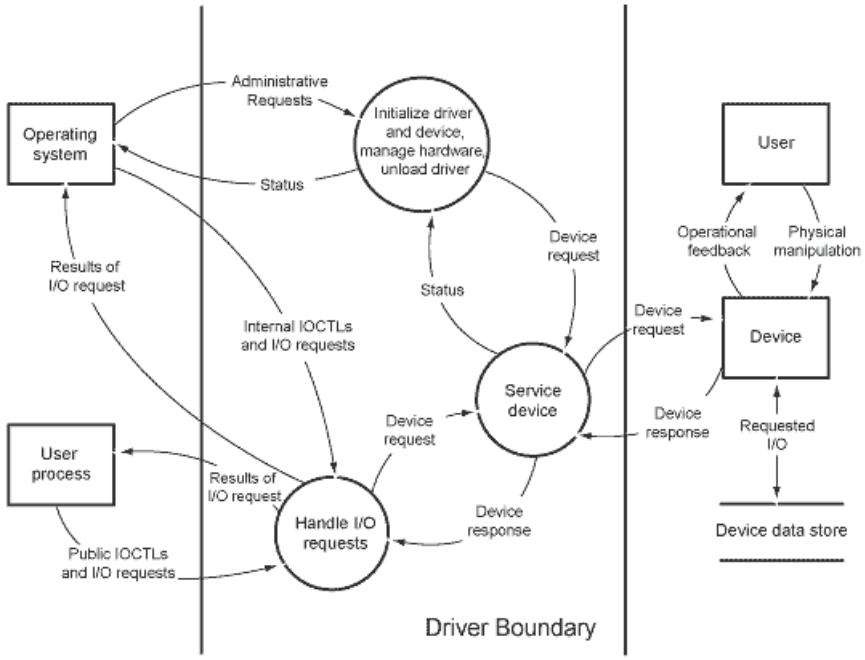
- Providing the ability to map arbitrary kernel, physical, or device memory to user mode.
- Providing the ability to read or write arbitrary kernel, physical or device memory, including Port input/output (I/O).
- Providing access to storage that bypasses Windows access control.
- Providing the ability to modify hardware or firmware that the driver was not designed to manage.

Perform threat analysis

Security checklist item #5: *Either modify an existing driver threat model or create a custom threat model for your driver.*

In considering security, a common methodology is to create specific threat models that attempt to describe the types of attacks that are possible. This technique is useful when designing a driver because it forces the developer to consider the potential attack vectors against a driver in advance. Having identified potential threats, a driver developer can then consider means of defending against these threats in order to bolster the overall security of the driver component.

This article provides driver specific guidance for creating a lightweight threat model: Threat modeling for drivers. The article provides an example driver threat model diagram that can be used as a starting point for your driver.



Security Development Lifecycle (SDL) best practices and associated tools can be used by IHVs and OEMs to improve the security of their products. For more information see SDL recommendations for OEMs.

Follow driver secure coding guidelines

Security checklist item #6: *Review your code and remove any known code vulnerabilities.*

The core activity of creating secure drivers is identifying areas in the code that need to be changed to avoid known software vulnerabilities. Many of these known software vulnerabilities deal with keeping strict track of the use of memory to avoid issues with others overwriting or otherwise comprising the memory locations that your driver uses.

The Code Validation Tools section of this article describes software tools that can be used to help locate known software vulnerabilities.

Memory buffers

- Always check the sizes of the input and output buffers to ensure that the buffers can hold all the requested data. For more information, see Failure to Check the Size of Buffers.
- Properly initialize all output buffers with zeros before returning them to the caller. For more information, see Failure to Initialize Output Buffers.
- Validate variable-length buffers. For more information, see Failure to Validate Variable-Length Buffers. For more information about working with buffers and using **ProbeForRead** and **ProbeForWrite** to validate the address of a buffer, see Buffer Handling.

Use the appropriate method for accessing data buffers with IOCTLs

One of the primary responsibilities of a Windows driver is transferring data between user-mode applications and a system's devices. The three methods for accessing data buffers are shown in the following table.

IOCTL BUFFER TYPE	SUMMARY	FOR MORE INFORMATION
METHOD_BUFFERED	Recommended for most situtations	Using Buffered I/O
METHOD_IN_DIRECT or METHOD_OUT_DIRECT	Used in some high speed HW I/O	Using Direct I/O

IOCTL BUFFER TYPE	SUMMARY	FOR MORE INFORMATION
METHOD_NEITHER	Avoid if possible	Using Neither Buffered Nor Direct I/O

In general buffered I/O is recommended as it provides the most secure buffering methods. But even when using buffered I/O there are risks, such as embedded pointers that must be mitigated.

For more information about working with buffers in IOCTLs, see Methods for Accessing Data Buffers.

Errors in use of IOCTL buffered I/O

- Check the size of IOCTL related buffers. For more information, see Failure to Check the Size of Buffers.
- Properly initialize output buffers. For more information, see Failure to Initialize Output Buffers.
- Properly validate variable-length buffers. For more information, see Failure to Validate Variable-Length Buffers.
- When using buffered I/O, be sure and return the proper length for the OutputBuffer in the IO_STATUS_BLOCK structure Information field. Don't just directly return the length directly from a READ request. For example, consider a situation where the returned data from the user space indicates that there is a 4K buffer. If the driver actually should only return 200 bytes, but instead just returns 4K in the Information field an information disclosure vulnerability has occurred. This problem occurs because in earlier versions of Windows, the buffer the I/O Manager uses for Buffered I/O is not zeroed. Thus, the user app gets back the original 200 bytes of data plus 4K-200 bytes of whatever was in the buffer (non-paged pool contents). This scenario can occur with all uses of Buffered I/O and not just with IOCTLs.

Errors in IOCTL direct I/O

Handle zero-length buffers correctly. For more information, see Errors in Direct I/O.

Errors in referencing user-space addresses

- Validate pointers embedded in buffered I/O requests. For more information, see Errors in Referencing User-Space Addresses.
- Validate any address in the user space before trying to use it, using APIs such as **ProbeForRead** and **ProbeForWrite** when appropriate.

MSR model-specific register reads and writes

Compiler intrinsics, such as __readmsr and __writemsr can be used to access the model-specific registers. If this access is required, the driver must always check that the register to read or write to is constrained to the expected index or range.

For more information, and code examples, see Providing the ability to read/write MSRs in Development Security Best Practices for Windows driver developers.

TOCTOU vulnerabilities

There is a potential time of check to time of use (TOCTOU) vulnerability when using direct I/O (for IOCTLs or for Read/Write). Be aware that the driver is accessing the user data buffer, the user can simultaneously be accessing it.

To manage this risk, copy any parameters that need to be validated from the user data buffer to memory that is solely accessibly from kernel mode (such as the stack or pool). Then once the data can not be accessed by the user application, validate and then operate on the data that was passed-in.

Driver code must make correct use of memory

- All driver pool allocations must be in non-executable (NX) pool. Using NX memory pools is inherently more secure than using executable non-paged (NP) pools, and provides better protection against overflow attacks.

- Device drivers must properly handle various user-mode, as well as kernel to kernel I/O, requests.

To allow drivers to support HVCI virtualization, there are additional memory requirements. For more information, see Implement HVCI compatible code later in this article.

Handles

- Validate handles passed between user-mode and kernel-mode memory. For more information, see Handle Management and Failure to Validate Object Handles.

Device objects

- Secure device objects. For more information, see Securing Device Objects.
- Validate device objects. For more information, see Failure to Validate Device Objects.

IRPs

WDF and IRPs

One advantage of using WDF, is that WDF drivers typically do not directly access IRPs. For example, the framework converts the WDM IRPs that represent read, write, and device I/O control operations to framework request objects that KMDF/UMDF receive in I/O queues.

If you are writing a WDM driver, review the following guidance.

Properly manage IRP I/O buffers

The following articles provide information about validating IRP input values:

DispatchReadWrite Using Buffered I/O

Errors in Buffered I/O

DispatchReadWrite Using Direct I/O

Errors in Direct I/O

Security Issues for I/O Control Codes

Consider validating values that are associated with an IRP, such as buffer addresses and lengths.

If you chose to use Neither I/O, be aware that unlike Read and Write, and unlike Buffered I/O and Direct I/O, that when using Neither I/O IOCTL the buffer pointers and lengths are not validated by the I/O Manager.

Handle IRP completion operations properly

A driver must never complete an IRP with a status value of STATUS_SUCCESS unless it actually supports and processes the IRP. For information about the correct ways to handle IRP completion operations, see Completing IRPs.

Manage driver IRP pending state

The driver should mark the IRP pending before it saves the IRP, and should consider including both the call to **IoMarkIrpPending** and the assignment in an interlocked sequence. For more information, see Failure to Check a Driver's State and Holding Incoming IRPs When A Device Is Paused.

Handle IRP cancellation operations properly

Cancel operations can be difficult to code properly because they typically execute asynchronously. Problems in the code that handles cancel operations can go unnoticed for a long time, because this code is typically not executed frequently in a running system. Be sure to read and understand all of the information supplied under Canceling IRPs. Pay special attention to Synchronizing IRP Cancellation and Points to Consider When Canceling IRPs.

One recommended way to minimize the synchronization problems that are associated with cancel operations is to implement a cancel-safe IRP queue.

Handle IRP cleanup and close operations properly

Be sure that you understand the difference between **IRP_MJ_CLEANUP** and **IRP_MJ_CLOSE** requests. Cleanup requests arrive after an application closes all handles on a file object, but sometimes before all I/O requests have completed. Close requests arrive after all I/O requests for the file object have been completed or canceled. For more information, see the following articles:

DispatchCreate, DispatchClose, and DispatchCreateClose Routines

DispatchCleanup Routines

Errors in Handling Cleanup and Close Operations

For more information about handling IRPs correctly, see Additional Errors in Handling IRPs.

Other security issues

- Use a lock or an interlocked sequence to prevent race conditions. For more information, see Errors in a Multiprocessor Environment.
- Ensure that device drivers properly handle various user-mode as well as kernel to kernel I/O requests.
- Ensure that no TDI filters or LSPs are installed by the driver or associated software packages during installation or usage.

Use safe functions

- Use safe string functions. For more information, see Using Safe String Functions.
- Use safe arithmetic functions. For more information, see Safe Integer Library Routines
- Use safe conversion functions.

Additional code vulnerabilities

In addition to the possible vulnerabilities covered here, this article provides additional information about enhancing the security of kernel mode driver code: Creating Reliable Kernel-Mode Drivers.

For additional information about C and C++ secure coding, see Secure coding resources at the end of this article.

Manage driver access control

Security checklist item #7: *Review your driver to make sure that you are properly controlling access.*

Managing driver access control - WDF

Drivers must work to prevent users from inappropriately accessing a computer's devices and files. To prevent unauthorized access to devices and files, you must:

- Name device objects only when necessary. Named device objects are generally only necessary for legacy reasons, for example if you have an application that expects to open the device using a particular name or if you're using a non-PNP device/control device. Note that WDF drivers do not need to name their PnP device FDO in order to create a symbolic link using WdfDeviceCreateSymbolicLink.
- Secure access to device objects and interfaces.

In order to allow applications or other WDF drivers to access your PnP device PDO, you should use device interfaces. For more information, see Using Device Interfaces. A device interface serves as a symbolic link to your device stack's PDO.

One of the betters way to control access to the PDO is by specifying an SDDL string in your INF. If the SDDL string is not in the INF file, Windows will apply a default security descriptor. For more information, see Securing Device Objects and SDDL for Device Objects.

For more information about controlling access, see the following articles:

Controlling Device Access in KMDF Drivers

Names, Security Descriptors and Device Classes - Making Device Objects Accessible... and SAFE from the *January February 2017 The NT Insider Newsletter* published by OSR.

Managing driver access control - WDM

If you are working with a WDM Driver and you used a named device object you can use IoCreateDeviceSecure and specify a SDDL to secure it. When you implement IoCreateDeviceSecure always specify a custom class GUID for DeviceClassGuid. You should not specify an existing class GUID here. Doing so has the potential to break security settings or compatibility for other devices belonging to that class. For more information, see WdmlibIoCreateDeviceSecure.

For more information, see the following articles:

Controlling Device Access

Controlling Device Namespace Access

Windows security model for driver developers

Security Identifiers (SIDs) risk hierarchy

The following section describes the risk hierarchy of the common SIDs used in driver code. For general information about SDDL, see SDDL for Device Objects, SID Strings, and SDDL String Syntax.

It is important to understand that if lower privilege callers are allowed to access the kernel, code risk is increased. In this summary diagram, the risk increases as you allow lower privilege SIDs access to your driver functionality.

```
SY (System)
\/
BA (Built-in Administrators)
\/
LS (Local Service)
\/
BU (Built-in User)
\/
AC (Application Container)
```

Following the general least privilege security principle, configure only the minimum level of access that is required for your driver to function.

WDM Granular IOCTL security control

To further manage security when IOCTLs are sent by user-mode callers, the driver code can include the IoValidateDeviceIoControlAccess function. This function allows a driver to check access rights. Upon receiving an IOCTL, a driver can call IoValidateDeviceIoControlAccess, specifying FILE_READ_ACCESS, FILE_WRITE_ACCESS, or both.

Implementing granular IOCTL security control does not replace the need to manage driver access using the techniques discussed above.

For more information, see the following articles:

Defining I/O Control Codes

Implement HVCI compatible code

Security checklist item #8: *Validate that your driver uses memory so that it is HVCI compatible.*

Memory usage and HVCI compatibility

HVCI uses hardware technology and virtualization to isolate the Code Integrity (CI) decision-making function from the rest of the operating system. When using virtualization-based security to isolate CI, the only way kernel memory can become executable is through a CI verification. This means that kernel memory pages can never be Writable and Executable (W+X) and executable code cannot be directly modified.

To implement HVCI compatible code, make sure your driver code does the following:

- Opts in to NX by default
- Uses NX APIs/flags for memory allocation (NonPagedPoolNx)
- Does not use sections that are both writable and executable
- Does not attempt to directly modify executable system memory
- Does not use dynamic code in kernel
- Does not load data files as executable
- Section alignment is a multiple of 0x1000 (PAGE_SIZE). E.g. DRIVER_ALIGNMENT=0x1000

For more information about using the tool and a list of incompatible memory calls, see Implement HVCI compatible code.

For more information about the related system fundamentals security test, see HyperVisor Code Integrity Readiness Test and Hypervisor-Protected Code Integrity (HVCI).

Follow technology-specific code best practices

Security checklist item #9: *Review the following technology-specific guidance for your driver.*

File Systems

For more information, about file system driver security see the following articles:

Introduction to File Systems Security

File System Security Issues

Security Features for File Systems

Coexistence with other File System Filter Drivers

NDIS - Networking

For information about NDIS driver security, see Security Issues for Network Drivers.

Display

For information about display driver security, see <Content Pending>.

Printers

For information related to printer driver security, see V4 Printer Driver Security Considerations.

Security Issues for Windows Image Acquisition (WIA) Drivers

For information about WIA security, see Security Issues for Windows Image Acquisition (WIA) Drivers.

Enhance device installation security

Security checklist item #10: *Review driver inf creation and installation guidance to make sure you are following best practices.*

When you create the code that installs your driver, you must make sure that the installation of your device will

always be performed in a secure manner. A secure device installation is one that does the following:

- Limits access to the device and its device interface classes
- Limits access to the driver services that were created for the device
- Protects driver files from modification or deletion
- Limits access to the device's registry entries
- Limits access to the device's WMI classes
- Uses SetupAPI functions correctly

For more information, see the following articles:

Creating Secure Device Installations

Guidelines for Using SetupAPI

Using Device Installation Functions

Device and Driver Installation Advanced Topics

Perform peer code review

Security checklist item #11: *Perform peer code review, to look for issues not surfaced by the other tools and processes*

Seek out knowledgeable code reviewers to look for issues that you may have missed. A second set of eyes will often see issues that you may have overlooked.

If you don't have suitable staff to review you code internally, consider engaging outside help for this purpose.

Execute proper release driver signing

Security checklist item #12: *Use the Windows partner portal to properly sign your driver for distribution.*

Before you release a driver package to the public, we recommend that you submit the package for certification. For more information, see Test for performance and compatibility, Get started with the Hardware program, Hardware Dashboard Services, and Attestation signing a kernel driver for public release.

Use code analysis in Visual Studio to investigate driver security

Security checklist item #13: *Follow these steps to use the code analysis feature in Visual Studio to check for vulnerabilities in your driver code.*

Use the code analysis feature in Visual Studio to check for security vulnerabilities in your code. The Windows Driver Kit (WDK) installs rule sets that are designed to check for issues in native driver code.

For more information, see How to run Code Analysis for drivers.

For more information, see Code Analysis for drivers overview. For additional background on code analysis, see Visual Studio 2013 Static Code Analysis in depth.

To become familiar with code analysis, you can use one of the sample drivers for example, the featured toaster sample, https://github.com/Microsoft/Windows-driver-samples/tree/main/general/toaster/toastDrv/kmdf/func/featured or the ELAM Early Launch Anti-Malware sample https://github.com/Microsoft/Windows-driver-samples/tree/main/security/elam.

1. Open the driver solution in Visual Studio.
2. In Visual Studio, for each project in the solution change the project properties to use the desired rule set. For example: Project >> Properties >> Code Analysis >> General, select *Recommended driver rules*. In

addition to using the recommenced driver rules, use the *Recommended native rules* rule set.

3. Select Build >> Run Code Analysis on Solution.
4. View warnings in the **Error List** tab of the build output window in Visual Studio.

Select the description for each warning to see the problematic area in your code.

Select the linked warning code to see additional information.

Determine whether your code needs to be changed, or whether an annotation needs to be added to allow the code analysis engine to properly follow the intent of your code. For more information on code annotation, see Using SAL Annotations to Reduce C/C++ Code Defects and SAL 2.0 Annotations for Windows Drivers.

For general information on SAL, refer to this article available from OSR.
https://www.osr.com/blog/2015/02/23/sal-annotations-dont-hate-im-beautiful/

Use Static Driver Verifier to check for vulnerabilities

Security checklist item #14: *Follow these steps to use Static Driver Verifier (SDV) in Visual Studio to check for vulnerabilities in your driver code.*

Static Driver Verifier (SDV) uses a set of interface rules and a model of the operating system to determine whether the driver interacts correctly with the Windows operating system. SDV finds defects in driver code that could point to potential bugs in drivers.

For more information, see Introducing Static Driver Verifier and Static Driver Verifier.

Note that only certain types of drivers are supported by SDV. For more information about the drivers that SDV can verify, see Supported Drivers. Refer to the following pages for information on the SDV tests available for the driver type you are working with.

- Rules for WDM Drivers
- Rules for KMDF Drivers
- Rules for NDIS Drivers
- Rules for Storport Drivers
- Rules for Audio Drivers
- Rules for AVStream Drivers

To become familiar with SDV, you can use one of the sample drivers (for example, the featured toaster sample: https://github.com/Microsoft/Windows-driver-samples/tree/main/general/toaster/toastDrv/kmdf/func/featured).

1. Open the targeted driver solution in Visual Studio.
2. In Visual Studio, change the build type to *Release*. Static Driver Verifier requires that the build type is release, not debug.
3. In Visual Studio, select Build >> Build Solution.
4. In Visual Studio, select Driver >> Launch Static Driver Verifier.
5. In SDV, on the *Rules* tab, select *Default* under *Rule Sets*.

   Although the default rules find many common issues, consider running the more extensive *All driver rules* rule set as well.
6. On the *Main* tab of SDV, select *Start*.
7. When SDV is complete, review any warnings in the output. The *Main* tab displays the total number of

defects found.

8. Select each warning to load the SDV Report Page and examine the information associated with the possible code vulnerability. Use the report to investigate the verification result and to identify paths in your driver that fail a SDV verification. For more information, see Static Driver Verifier Report.

Check code with the BinSkim Binary Analyzer

Security checklist item #15: *Follow these steps to use BinSkim to double check that compile and build options are configured to minimize known security issues.*

Use BinSkim to examine binary files to identify coding and building practices that can potentially render the binary vulnerable.

BinSkim checks for:

- Use of outdated compiler tool sets - Binaries should be compiled against the most recent compiler tool sets wherever possible to maximize the use of current compiler-level and OS-provided security mitigations.
- Insecure compilation settings - Binaries should be compiled with the most secure settings possible to enable OS-provided security mitigations, maximize compiler errors and actionable warnings reporting, among other things.
- Signing issues - Signed binaries should be signed with cryptographically-strong algorithms.

BinSkim is an open source tool and generates output files that use the Static Analysis Results Interchange Format (SARIF) format. BinSkim replaces the former BinScope tool.

For more information about BinSkim, see the BinSkim User Guide.

Follow these steps to validate that the security compile options are properly configured in the code that you are shipping.

1. Download and install the cross platform .NET Core SDK.
2. Confirm Visual Studio is installed. For information on downloading and installing Visual Studio see Install Visual Studio.
3. There are a number of options to download BinSkim, such as a NuGet package. In this example we will use the git clone option to download from here: https://github.com/microsoft/binskim and install it on a 64 bit Windows PC.
4. Open a Visual Studio Developer Command Prompt window and create a directory, for example `C:\binskim-master`.

   ```
   C:\> Md \binskim-master
   ```

5. Move to that directory that you just created.

   ```
   C:\> Cd \binskim-master
   ```

6. Use the git clone command to download all of the needed files.

   ```
   C:\binskim-master> git clone --recurse-submodules https://github.com/microsoft/binskim.git
   ```

7. Move to the new `binskim` dirctory that the clone command created.

```
C:\> Cd \binskim-master\binskim
```

8. Run **BuildAndTest.cmd** to ensure that release build succeeds, and that all tests pass.

```
C:\binskim-master\binskim> BuildAndTest.cmd

Welcome to .NET Core 3.1!
---------------------
SDK Version: 3.1.101

...

C:\binskim-master\binskim\bld\bin\AnyCPU_Release\Publish\netcoreapp2.0\win-x64\BinSkim.Sdk.dll
1 File(s) copied
C:\binskim-master\binskim\bld\bin\AnyCPU_Release\Publish\netcoreapp2.0\linux-x64\BinSkim.Sdk.dll
1 File(s) copied

...
```

9. The build process creates a set of directories with the BinSkim executables. Move to the win-x64 build output directory.

```
C:\binskim-master\binskim> Cd \binskim-master\bld\bin\AnyCPU_Release\Publish\netcoreapp2.0\win-x64>
```

10. Display help for the analyze option.

```
C:\binskim-master\binskim\bld\bin\AnyCPU_Release\Publish\netcoreapp2.0\win-x64> BinSkim help analyze

BinSkim PE/MSIL Analysis Driver 1.6.0.0

--sympath                      Symbols path value, e.g.,
SRV*http://msdl.microsoft.com/download/symbols or Cache*d:\symbols;Srv*http://symweb. See
                              https://learn.microsoft.com/windows-hardware/drivers/debugger/advanced-
symsrv-use for syntax information. Note that BinSkim will clear the
                              _NT_SYMBOL_PATH environment variable at runtime. Use this argument for
symbol information instead.

--local-symbol-directories     A set of semicolon-delimited local directory paths that will be
examined when attempting to locate PDBs.

-o, --output                   File path to which analysis output will be written.

--verbose                      Emit verbose output. The resulting comprehensive report is designed to
provide appropriate evidence for compliance scenarios.

...
```

Setting the symbol path

If you are building all the code you are analyzing on the same machine you are running BinSkim on, you typically don't need to set the symbol path. This is because your symbol files are available on the local box where you compiled. If you are using a more complex build system, or redirecting your symbols to different location (not alongside the compiled binary), use `--local-symbol-directories` to add these locations to the symbol file search. If your code references a compiled binary that is not part of your code, the Window debugger sympath can be used to retrieve symbols in order to verify the security of these code dependencies. If you find an issue in these dependencies, you may not be able to fix them. But it can be useful to be aware of any possible security risk you are accepting by taking on those dependencies.

TIP

When adding a symbol path (that references a networked symbol server), add a local cache location to specify a local path to cache the symbols. Not doing this can greatly compromise the performance of BinSkim. The following example, specifies a local cache at d:\symbols. `--sympath Cache*d:\symbols;Srv*http://symweb` For more information about sympath, see Symbol path for Windows debuggers.

1. Execute the following command to analyze a compiled driver binary. Update the target path to point to your complied driver .sys file.

```
C:\binskim-master\binskim\bld\bin\AnyCPU_Release\Publish\netcoreapp2.0\win-x64> BinSkim analyze
"C:\Samples\KMDF_Echo_Driver\echo.sys"
```

2. For additional information add the verbose option like this.

```
C:\binskim-master\binskim\bld\bin\AnyCPU_Release\Publish\netcoreapp2.0\win-x64> BinSkim analyze
"C:\Samples\KMDF_Echo_Driver\osrusbfx2.sys" --verbose
```

NOTE

The --verbose option will produce explicit pass/fail results for every check. If you do not provide verbose, you will only see the defects that BinSkim detects. The --verbose option is typically not recommended for actual automation systems due to the increased size of log files and because it makes it more difficult to pick up individual failures when they occur, as they will be embedded in the midst of a large number of 'pass' results.

3. Review the command output to look for possible issues. This example output shows three tests that passed. Additional information on the rules, such as BA2002 is available in the BinSkim User Guide.

```
Analyzing...
Analyzing 'osrusbfx2.sys'...
...

C:\Samples\KMDF_Echo_Driver\osrusbfx2.sys\Debug\osrusbfx2.sys: pass BA2002: 'osrusbfx2.sys' does not
incorporate any known vulnerable dependencies, as configured by current policy.
C:\Samples\KMDF_Echo_Driver\Debug\osrusbfx2.sys: pass BA2005: 'osrusbfx2.sys' is not known to be an
obsolete binary that is vulnerable to one or more security problems.
C:\Samples\KMDF_Echo_Driver\osrusbfx2.sys: pass BA2006: All linked modules of 'osrusbfx2.sys'
generated by the Microsoft front-end satisfy configured policy (compiler minimum version
17.0.65501.17013).
```

4. This output shows that test BA3001 is not run as the tool indicates that the driver is not an ELF binary.

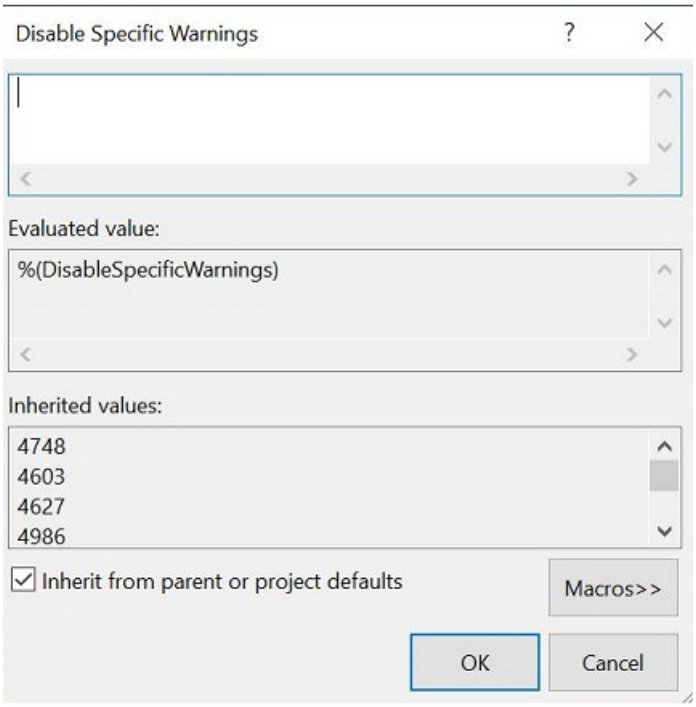
```
...
C:\Samples\KMDF_Echo_Driver\Debug\osrusbfx2.sys: notapplicable BA3001: 'osrusbfx2.sys' was not
evaluated for check 'EnablePositionIndependentExecutable' as the analysis is not relevant based on
observed metadata: image is not an ELF binary.
```

5. This output shows an error for test BA2007.

```
...

C:\Samples\KMDF_Echo_Driver\Debug\osrusbfx2.sys: error BA2007: 'osrusbfx2.sys' disables compiler
warning(s) which are required by policy.
A compiler warning is typically required if it has a high likelihood of flagging memory corruption,
information disclosure, or double-free vulnerabilities.
To resolve this issue, enable the indicated warning(s) by removing /Wxxxx switches (where xxxx is a
warning id indicated here) from your command line, and resolve any warnings subsequently raised
during compilation.
```

To enable these warnings in Visual Studio, under C/C++ in the property pages for the project, remove the values that you don't wish to exclude in **Disable Specific Warnings**.



The default compile options in Visual Studio for driver projects can disable warnings such as the following. These warnings will be reported by BinSkim.

C4603 - 'name': macro is not defined or definition is different after precompiled header use

C4627 - 'description': skipped when looking for precompiled header use

C4986 - 'declaration': exception specification does not match previous declaration

For more information about the compiler warnings, see Compiler Warnings by compiler version.

Use additional code validation tools

Security checklist item #16: *Use these additional tools to help validate that your code follows security recommendations and to probe for gaps that were missed in your development process.*

In addition to Visual Studio Code analysis, Static Driver Verifier, and Binskim discussed above, use the following tools to probe for gaps that were missed in your development process.

Driver Verifier

Driver Verifier allows for live testing of the driver. Driver Verifier monitors Windows kernel-mode drivers and graphics drivers to detect illegal function calls or actions that might corrupt the system. Driver Verifier can subject the Windows drivers to a variety of stresses and tests to find improper behavior. For more information, see Driver Verifier.

Hardware compatibility program tests

The hardware compatibility program includes security related tests can be used to look for code vulnerabilities. The Windows Hardware Compatibility Program leverages the tests in the Windows Hardware Lab Kit (HLK). The HLK Device Fundamentals tests can be used on the command line to exercise driver code and probe for weakness. For general information about the device fundamentals tests and the hardware compatibility program, see Windows Hardware Lab Kit.

The following tests are examples of tests that may be useful to check driver code for some behaviors associated with code vulnerabilities:

DF - Fuzz random IOCTL test (Reliability)

DF - Fuzz sub-opens test (Reliability)

DF - Fuzz zero length buffer FSCTL test (Reliability)

DF - Fuzz random FSCTL test (Reliability)

DF - Fuzz Misc API test (Reliability)

You can also use the Kernel synchronization delay fuzzing that is included with Driver Verifier.

The CHAOS (Concurrent Hardware and Operating System) tests run various PnP driver tests, device driver fuzz tests, and power system tests concurrently. For more information, see CHAOS Tests (Device Fundamentals).

The Device Fundamentals Penetration tests perform various forms of input attacks, which are a critical component of security testing. Attack and Penetration testing can help identify vulnerabilities in software interfaces. For more information, see Penetration Tests (Device Fundamentals).

Use the Device Guard - Compliance Test, along with the other tools described in this article, to confirm that your driver is HVCI compatible.

Custom and domain-specific test tools

Consider the development of custom domain-specific security tests. To develop additional tests, gather input from the original designers of the software, as well as unrelated development resources familiar with the specific type of driver being developed, and one or more people familiar with security intrusion analysis and prevention.

Review debugger techniques and extensions

Security checklist item #17: *Review these debugger tools and consider their use in your development debugging workflow.*

Security related debugger commands

The !acl extension formats and displays the contents of an access control list (ACL). For more information, see Determining the ACL of an Object and **!acl**.

The !token extension displays a formatted view of a security token object. For more information, see **!token**.

The !tokenfields extension displays the names and offsets of the fields within the access token object (the TOKEN structure). For more information, see **!tokenfields**.

The !sid extension displays the security identifier (SID) at the specified address. For more information, see **!sid**.

The !sd extension displays the security descriptor at the specified address. For more information, see **!sd**.

Microsoft Vulnerable and Malicious Driver Reporting Center

Anyone can submit a questionable driver using the Microsoft Vulnerable and Malicious Driver Reporting Center.

Refer to this blog entry for information on how drivers are submitted for analysis - Improve kernel security with the new Microsoft Vulnerable and Malicious Driver Reporting Center

The Reporting Center can scan and analyze Windows drivers built for x86 and x64 architectures. Vulnerable and malicious scanned drivers are flagged for analysis and investigation by Microsoft's Vulnerable Driver team. After vulernable drivers are confirmed, an appropriate notification occurs, they are added to the vulnerable driver blocklist. For more information about that, see Microsoft recommended driver block rules. These rules are aplied by default to Hypervisor-protected code integrity (HVCI) enabled devices and Windows 10 in S mode.

Review secure coding resources

Security checklist item #18: *Review these resources to expand your understanding of the secure coding best practices that are applicable to driver developers.*

Review these resources to learn more about driver security

Secure kernel-mode driver coding guidelines

Creating Reliable Kernel-Mode Drivers

Secure coding organizations

Carnegie Mellon University SEI CERT

Carnegie Mellon University SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition).

CERT - Build Security In

MITRE - Weaknesses Addressed by the CERT C Secure Coding Standard

Building Security In Maturity Model (BSIMM) - https://www.bsimm.com/

SAFECode - https://safecode.org/

OSR

OSR provides driver development training and consulting services. These articles from the OSR newsletter highlight driver security issues.

Names, Security Descriptors and Device Classes - Making Device Objects Accessible... and SAFE

You've Gotta Use Protection -- Inside Driver & Device Security

Locking Down Drivers - A Survey of Techniques

Meltdown and Spectre: What about drivers?

Case Study

From alert to driver vulnerability: Microsoft Defender ATP investigation unearths privilege escalation flaw

Books

24 deadly sins of software security : programming flaws and how to fix them by Michael Howard, David LeBlanc and John Viega

The art of software security assessment : identifying and preventing software vulnerabilities, Mark Dowd, John McDonald and Justin Schuh

Writing Secure Software Second Edition, Michael Howard and David LeBlanc

The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities, Mark Dowd and John McDonald

Secure Coding in C and C++ (SEI Series in Software Engineering) 2nd Edition, Robert C. Seacord

Programming the Microsoft Windows Driver Model (2nd Edition), Walter Oney

Developing Drivers with the Windows Driver Foundation (Developer Reference), Penny Orwick and Guy Smith

Training

Windows driver classroom training is available from vendors such as the following:

- OSR
- Winsider
- Azius

Secure coding online training is available from a variety of sources. For example, this course is available from coursera:

https://www.coursera.org/learn/software-security.

SAFECode offers free training as well:

SAFECode.org/training

Professional Certification

CERT offers a Secure Coding Professional Certification.

Summary of key takeaways

Driver security is a complex undertaking containing many elements, but here are a few key points to consider:

- Drivers live in the windows kernel, and having an issue when executing in kernel exposes the entire operating system. Because of this, pay close attention to driver security and design with security in mind.
- Apply the principle of least privilege:

  a. Use a strict SDDL string to restrict access to the driver

  b. Further restrict individual IOCTL's
- Create a threat model to identify attack vectors and consider whether anything can be restricted further.
- Be careful with regard to embedded pointers being passed in from usermode. They need to be probed, accessed within try except, and they are prone to time of check time of use (ToCToU) issues unless the value of the buffer is captured and compared.
- If you're unsure, use METHOD_BUFFERED as an IOCTL buffering method.
- Use code scanning utilities to look for known code vulnerabilities and remediate any identified issues.
- Seek out knowledgeable code reviewers to look for issues that you may have missed.
- Use driver verifiers and test your driver with multiple inputs, including corner cases.

Threat modeling for drivers

12/15/2022 • 19 minutes to read • Edit Online

Driver writers and architects should make threat modeling an integral part of the design process for any driver. This topic provides guidelines for creating threat models for Windows drivers.

Security should be a fundamental design point for any driver. Any successful product is a target. If you are writing a driver for Windows, you must assume that sometime, somewhere, someone will try to use your driver to compromise system security.

Designing a secure driver involves:

- Identifying the points at which the driver could be vulnerable to an attack.
- Analyzing the types of attacks that could be mounted at each such point.
- Ensuring that the driver is designed in such a way as to thwart such attacks.

Threat modeling is a structured approach to these important design tasks. A threat model is a way of categorizing and analyzing the threats to an asset. From a driver writer's perspective, the assets are the hardware, software, and data on the computer or network.

A threat model answers the following questions:

- Which assets need protection?
- To what threats are the assets vulnerable?
- How important or likely is each threat?
- How can you mitigate the threats?

Threat modeling is an important part of software design because it ensures that security is built into the product, rather than addressed as an afterthought. A good threat model can help find and prevent bugs during the design process, thus eliminating potentially costly patches later and possible reputational damage to your organization.

This section applies the principles of threat modeling to driver design and provides examples of threats to which a driver might be susceptible. For a more complete description of threat modeling for software design, refer to these resources.

- The Microsoft SDL Web site:

  https://www.microsoft.com/sdl

- Simplified Implementation of the Microsoft SDL:

  Download White Paper

- This blog entry describes how to download a free copy of *The Security Development Lifecycle: SDL*, by Michael Howard and Steve Lipner:

  https://blogs.msdn.microsoft.com/microsoft_press/2016/04/19/free-ebook-the-security-development-lifecycle/

Create threat models for drivers

Creating a threat model requires a thorough understanding of the driver's design, the types of threats to which the driver might be exposed, and the consequences of a security attack that exploits a particular threat. After

creating the threat model for a driver, you can determine how to mitigate the potential threats.

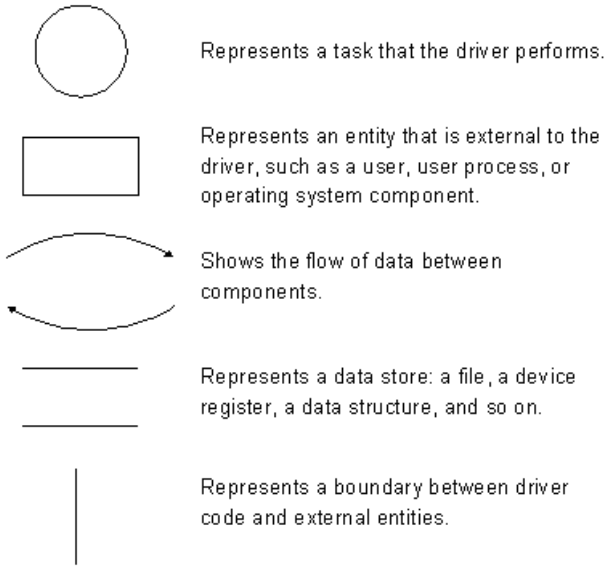
Threat modeling is most effective when performed in an organized, structured way during driver design, rather than haphazardly during coding. A structured approach increases the likelihood that you will discover vulnerabilities in the design, thereby helping to ensure that the model is comprehensive.

One way to organize a threat modeling effort is to follow these steps:

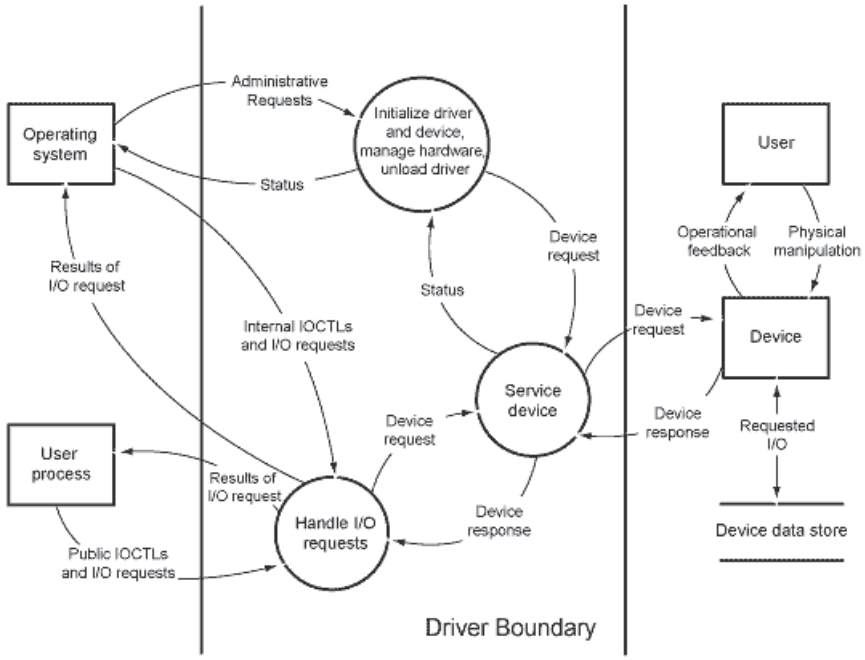
1. Create a structured diagram showing data flow through the driver. Include all possible tasks that the driver performs and the source and destination of all input and output from the driver. A formal data flow diagram, or similar structured diagram, can help you to analyze the path of data through your driver and to identify the driver's external interfaces, boundaries, and interactions.
2. Analyze the potential security threats, based on the data flow diagram.
3. Assess the threats that you identified in the previous step and determine how to mitigate them.

Create a data flow diagram

A data flow diagram shows in conceptual form the flow of data between the driver and the external entities with which it interacts—typically the operating system, a user process, and the device. A formal data flow diagram uses the following symbols:



The following figure shows a sample data flow diagram for a hypothetical kernel-mode Windows Driver Model (WDM) driver. Regardless of the architecture for your particular type of driver, the conceptual model is the same: show all data paths and identify each source of data that enters or exits the driver.



Note The previous figure shows data flowing directly between a user process and the driver, and omits any intermediate drivers. However, in reality, all requests pass through the I/O manager and may traverse one or more higher-level drivers before reaching a particular driver. The figure omits these intermediate steps to emphasize the importance of the original source of the data and the context of the thread that supplied the data. Kernel-mode drivers must validate data that originates in user mode.

Information enters the driver because of requests from the operating system, requests from a user process, or requests (typically interrupts) from the device.

The driver in the previous figure receives data from the operating system in several types of requests:

- Requests to perform administrative tasks for the driver and its device, through calls to **DriverEntry**, **DriverUnload**, and **AddDevice** routines
- Plug and Play requests (IRP_MJ_PNP)
- Power management requests (IRP_MJ_POWER)
- Internal device I/O control requests (IRP_MJ_INTERNAL_DEVICE_CONTROL)

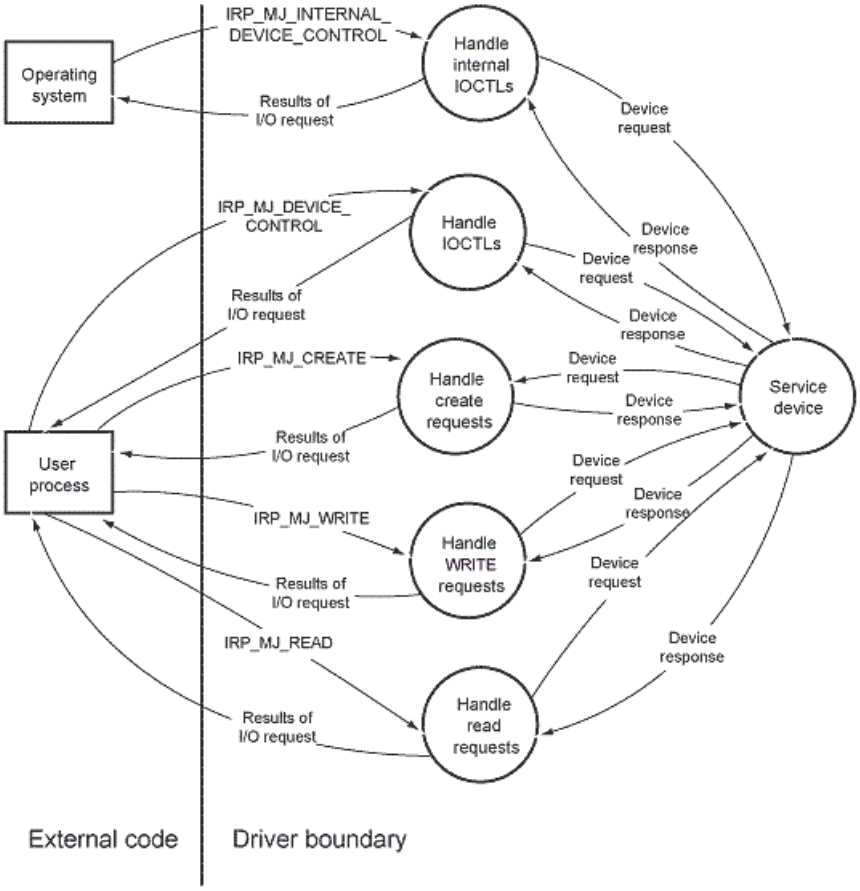
In response to these requests, data flows from the driver back to the operating system as status information. The driver in the figure receives data from a user process in the following types of requests:

- Create, read, and write requests (IRP_MJ_CREATE, IRP_MJ_READ, or IRP_MJ_WRITE)
- Public device I/O control requests (IRP_MJ_DEVICE_ CONTROL)

In response to these requests, output data and status information flow from the driver back to the user process.

Finally, the driver receives data from the device because of device I/O operations or user actions (such as opening the tray on a CD drive) that change device status. Likewise, data from the driver flows to the device during I/O operations and changes in device status.

The previous figure shows driver data flow at a broad conceptual level. Each circle represents a relatively large task and lacks detail. In some cases, a one-level diagram such as the sample is adequate for understanding the data sources and paths. If your driver handles many different types of I/O requests from varying sources, you might need to create one or more additional diagrams that show more detail. For example, the circle labeled "Handle I/O Requests" might be expanded into a separate diagram, similar to the following figure.



The second diagram shows separate tasks for each type of I/O request in the first diagram. (For simplicity, data paths to the device have been omitted.)

The external entities and the types of input and output shown in the diagram may vary, depending on the type of device. For example, Windows supplies class drivers for many common device types. A system-supplied class driver works with a vendor-supplied minidriver, which typically is a dynamic link library (DLL) that contains a set of callback routines. User I/O requests are directed to the class driver, which then calls the routines in the minidriver to perform specific tasks. The minidriver typically does not receive the entire I/O request packet as input; instead, each callback routine receives only the data that is required for its specific task.

As you create the data flow diagrams, remember the variety of sources for driver requests. Any code that is run on a user's computer could generate an I/O request to a driver, from well-known applications such as Microsoft Office to freeware, shareware, and Web downloads of potentially dubious origin. Depending on your specific device, you might also need to consider media codecs or third-party filters that your company ships to support its device. Possible data sources include:

- IRP_MJ_XXX requests that the driver handles
- IOCTLs that the driver defines or handles
- APIs that the driver calls
- Callback routines
- Any other interfaces that the driver exposes
- Files that the driver reads or writes, including those used during installation
- Registry keys that the driver reads or writes
- Configuration property pages, and any other information provided by the user that the driver consumes

Your model should also cover the driver installation and update procedures. Include all the files, directories, and registry entries that are read or written during driver installation. Consider also the interfaces exposed in device installers, co-installers, and property pages.

Any point at which the driver exchanges data with an external entity is potentially vulnerable to attack.

Analyze potential threats

After you identify the points at which a driver might be vulnerable, you can determine which types of threats could occur at each point. Consider the following types of questions:

- What security mechanisms are in place to protect each resource?
- Are all transitions and interfaces properly secured?
- Could improper use of a feature unintentionally compromise security?
- Could malicious use of a feature compromise security?
- Do default settings provide adequate security?

The STRIDE approach to threat categorization

The acronym STRIDE describes six categories of threats to software. This acronym is derived from:

- **S**poofing
- **T**ampering
- **R**epudiation
- **I**nformation disclosure
- **D**enial of service
- **E**levation of privilege

Using STRIDE as a guide, you can pose detailed questions about the kinds of attacks that could be targeted at a driver. The goal is to determine the types of attacks that could be possible at each vulnerable point in the driver and then to create a scenario for each possible attack.

- **Spoofing** is using someone else's credentials to gain access to otherwise inaccessible assets. A process mounts a spoofing attack by passing forged or stolen credentials.

- **Tampering** is changing data to mount an attack. For example, a driver might be susceptible to tampering if the required driver files are not adequately protected by driver signing and access control lists (ACLs). In this situation, a malicious user could alter the files, thus breaching system security.

- **Repudiation** occurs when a user denies performing an action, but the target of the action has no way to prove otherwise. A driver might be susceptible to a repudiation threat if it does not log actions that could compromise security. For example, a driver for a video device could be susceptible to repudiation if it does not log requests to change characteristics of its device, such as focus, scanned area, frequency of image capture, target location of captured images, and so forth. The resulting images could be corrupted, but system administrators would have no way to determine which user caused the problem.

- **Information disclosure** threats are exactly as the name implies: the disclosure of information to a user who does not have permission to see it. Any driver that passes information to or from a user buffer is susceptible to information disclosure threats. To avoid information disclosure threats, drivers must validate the length of each user buffer and zero-initialize the buffers before writing data.

- **Denial-of-service** attacks threaten the ability of valid users to access resources. The resources could be disk space, network connections, or a physical device. Attacks that slow performance to unacceptable levels are also considered denial-of-service attacks. A driver that allows a user process to monopolize a system resource unnecessarily could be susceptible to a denial-of-service attack if the resource consumption hinders the ability of other valid users to perform their tasks.

  For example, a driver might use a semaphore to protect a data structure while executing at IRQL = PASSIVE_LEVEL. However, the driver must acquire and release the semaphore within a

KeEnterCriticalRegion/KeLeaveCriticalRegion pair, which disables and re-enables the delivery of asynchronous procedure calls (APCs). If the driver fails to use these routines, an APC could cause the operating system to suspend the thread that holds the semaphore. As a result, other processes (including those created by an administrator) would be unable to gain access to the structure.

- An **elevation-of-privilege** attack can occur if an unprivileged user gains privileged status. A kernel-mode driver that passes a user-mode handle to a **ZwXxx** routine is vulnerable to elevation-of-privilege attacks because **ZwXxx** routines bypass security checks. Kernel-mode drivers must validate every handle that they receive from user-mode callers.

  Elevation-of-privilege attacks can also occur if a kernel-mode driver relies on the **RequestorMode** value in the IRP header to determine whether an I/O request comes from a kernel-mode or user-mode caller. In IRPs that arrive from the network or the Server service (SRVSVC), the value of **RequestorMode** is **KernelMode**, regardless of the origin of the request. To avoid such attacks, drivers must perform access control checks for such requests instead of simply using the value of **RequestorMode**.

Driver analysis techniques

A simple way to organize the analysis is to list the vulnerable areas along with the potential threats and one or more scenarios for each type of threat.

To perform a thorough analysis, you must explore the possibility of threats at every potentially vulnerable point in the driver. At each vulnerable point, determine each category of threat (spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege) that might be possible. Then create one or more attack scenarios for each plausible threat.

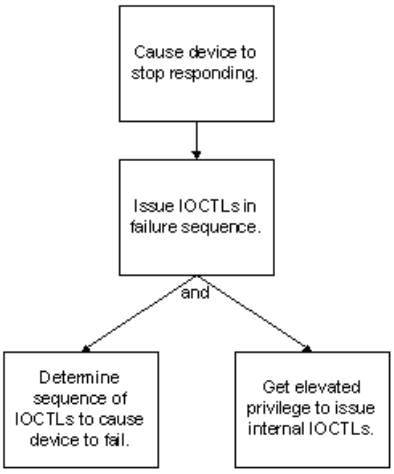
For example, consider the data flow for IRP_MJ_DEVICE_CONTROL requests as shown in the preceding figure. The following table shows two types of threats that a driver could encounter when processing such requests:

VULNERABLE POINT	POTENTIAL THREAT (STRIDE)	SCENARIO
IRP_MJ_DEVICE_CONTROL requests	Denial of service Elevation of privilege	User process issues a sequence of IOCTLs that causes the device to fail. User process issues an IOCTL that permits FILE_ANY_ACCESS.

One threat is often related to another. For example, an attack that exploits an elevation-of-privilege threat can result in information disclosure or denial of service. Furthermore, some types of attacks depend on a sequence of events. A malicious user might start by exploiting an elevation-of-privilege threat. Then, with the added capabilities that come with elevated privilege, the user might find and exploit additional vulnerabilities.

Threat trees and outlines can be useful in modeling such complex scenarios.

A threat tree is a diagram that shows a hierarchy of threats or vulnerabilities; in essence, a threat tree mimics the malicious user's steps in mounting an attack. The ultimate goal of the attack is at the top of the tree. Each subordinate level shows the steps required to carry out the attack. The following figure is a simple threat tree for the denial-of-service scenario in the preceding example.



The threat tree shows the required steps to mount a particular attack and the relationships between the steps. An outline is an alternative to a threat tree.

An outline simply lists in hierarchical order the steps to attack a particular threat. For example:

1.0 Cause device to stop responding.

1.1 Issue IOCTLS in failure sequence.

1.1.1 Determine sequence that causes device to fail.

1.1.2 Get elevated privilege to issue internal IOCTLs.

Either technique can help you to understand which threats are most dangerous and which vulnerabilities in your design are most critical.

Fast path threat modeling

If resources are limited, instead of creating a complete threat model diagram, a summary outline can be created to help assess security risks to the driver. For example the text below describes some of the surface areas diagramed in the example driver described in the previous example.

The driver receives data from the operating system in several types of requests:

- Requests to perform administrative tasks for the driver and its device, through calls to **DriverEntry**, **DriverUnload**, and **AddDevice** routines
- Plug and Play requests (IRP_MJ_PNP)
- Power management requests (IRP_MJ_POWER)
- Internal device I/O control requests (IRP_MJ_INTERNAL_DEVICE_CONTROL)

In response to these requests, data flows from the driver back to the operating system as status information. The driver receives data from a user process in the following types of requests:

- Create, read, and write requests (IRP_MJ_CREATE, IRP_MJ_READ, or IRP_MJ_WRITE)
- Public device I/O control requests (IRP_MJ_DEVICE_ CONTROL)

In response to these requests, output data and status information flow from the driver back to the user process.

Using this basic understanding of the data flow to your driver, you can examine each input and output area for possible threats.

The DREAD approach to threat assessment

Determining how and where a driver might be attacked is not enough. You must then assess these potential

threats, determine their relative priorities, and devise a mitigation strategy.

DREAD is an acronym that describes five criteria for assessing threats to software. DREAD stands for:

- **D**amage
- **R**eproducibility
- **E**xploitability
- **A**ffected users
- **D**iscoverability

To prioritize the threats to your driver, rank each threat from 1 to 10 on all 5 of the DREAD assessment criteria, and then add the scores and divide by 5 (the number of criteria). The result is a numeric score between 1 and 10 for each threat. High scores indicate serious threats.

- **Damage**. Assessing the damage that could result from a security attack is obviously a critical part of threat modeling. Damage can include data loss, hardware or media failure, substandard performance, or any similar measure that applies to your device and its operating environment.
- **Reproducibility** is a measure of how often a specified type of attack will succeed. An easily reproducible threat is more likely to be exploited than a vulnerability that occurs rarely or unpredictable. For example, threats to features that are installed by default, or are used in every potential code path, are highly reproducible.
- **Exploitability** assesses the effort and expertise that are required to mount an attack. A threat that can be attacked by a relatively inexperienced college student is highly exploitable. An attack that requires highly skilled personnel and is expensive to carry out is less exploitable.

  In assessing exploitability, consider also the number of potential attackers. A threat that can be exploited by any remote, anonymous user is more exploitable than one that requires an onsite, highly authorized user.
- **Affected Users**. The number of users that could be affected by an attack is another important factor in assessing a threat. An attack that could affect at most one or two users would rate relatively low on this measure. Conversely, a denial-of-service attack that crashes a network server could affect thousands of users and therefore would rate much higher.
- **Discoverability** is the likelihood that a threat will be exploited. Discoverability is difficult to estimate accurately. The safest approach is to assume that any vulnerability will eventually be taken advantage of and, consequently, to rely on the other measures to establish the relative ranking of the threat.

Sample: Assessing threats using DREAD

Continuing with the example discussed above, the following table shows how a designer might assess the hypothetical denial-of-service attack:

DREAD CRITERION	SCORE	COMMENTS
Damage	8	Disrupts work temporarily, but causes no permanent damage or data loss.
Reproducibility	10	Causes the device to fail every time.
Exploitability	7	Requires a focused effort to determine the command sequence.
Affected users	10	Affects every model of this device on the market.

DREAD CRITERION	SCORE	COMMENTS
Discoverability	10	Assumes that every potential threat will be discovered.
Total:	**9.0**	**Mitigating this problem is high priority.**

Mitigating Threats

Your driver design should mitigate against all the threats that your model exposes. However, in some cases, mitigation might not be practical. For example, consider an attack that potentially affects very few users and is unlikely to result in loss of data or system usability. If mitigating such a threat requires several months of additional effort, you might reasonably choose to spend additional time testing the driver instead. Nevertheless, remember that eventually a malicious user is likely to find the vulnerability and mount an attack, and then the driver will require a patch for the problem.

Including threat modeling in a broader Security Development Lifecycle process

Consider including the threat modeling process in a broader Secure Development Lifecycle - SDL.

The Microsoft SDL process provides a number of recommended software development process that can be modified to fit any size of organization - including a single developer. Consider adding components of the SDL recommendations to your software development process.

For more information, see Microsoft Security Development Lifecycle (SDL) – Process Guidance.

Training and organizational capabilities - Pursue software development security training to expand your ability to recognize and remediate software vulnerabilities.

Microsoft makes its four core SDL Training classes available for download. Microsoft Security Development Lifecycle Core Training classes

For more detailed information about SDL training, see this white paper. Essential Software Security Training for the Microsoft SDL

Requirements and design - The best opportunity to build trusted software is during the initial planning stages of a new release or a new version, because development teams can identify key objects and integrate security and privacy, which minimizes disruption to plans and schedules.

A key output in this phase is to set specific security goals. For example, deciding that all of your code should pass the Visual Studio code analysis "All Rules" with zero warnings.

Implementation - All development teams should define and publish a list of approved tools and their associated security checks, such as compiler/linker options and warnings.

For a driver developer most of the useful work is done in this phase. As code is written it is reviewed for possible weakness. Tools such as code analysis and driver verifier are used to look for areas in the code that can be hardened.

Verification - Verification is the point at which the software is functionally complete and is tested against security goals outlined in the requirements and design phase.

Additional tools such as binscope and fuzz testers can be used to validate that security design goals have been met and the code is ready to ship

Release and response - In preparation for releasing a product, it is desirable to create an incident response

plan that describes what you will do to respond to new threats and how you will service the driver after it has shipped. Doing this work in advance will mean that you will be able to respond faster if security issues are identified in code that has shipped.

For more information about the SDL process, see these additional resources:

- This is the primary Microsoft SDL site and provides an overview of SDL. https://www.microsoft.com/sdl
- This blog describes how to download a free copy of *The Security Development Lifecycle: SDL*, by Michael Howard and Steve Lipner. https://blogs.msdn.microsoft.com/microsoft_press/2016/04/19/free-ebook-the-security-development-lifecycle/
- This page provides links to additional SDL publications. https://www.microsoft.com/SDL/Resources/publications.aspx

Call to action

For driver developers:

- Make threat modeling part of driver design.
- Take steps to efficiently mitigate threats in your driver code.
- Become familiar with the security and reliability issues that apply to your driver and device type. For more information, see the device-specific sections of the Windows Device Driver Kit (WDK).
- Understand which checks the operating system, I/O manager, and any higher-level drivers perform before user requests reach your driver — and which checks they do not perform.
- Use tools from the WDK, such as driver verifier to test and verify your driver.
- Review public databases of known threats and software vulnerabilities.

For additional driver security resources, see Driver Security Checklist.

Software Security Resources

Books

Writing Secure Code, Second Edition by Michael Howard and David LeBlanc

24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them, First Edition by Michael Howard, David LeBlanc and John Viega

The art of software security assessment : identifying and preventing software vulnerabilities, by Mark Dowd, John McDonald and Justin Schuh

Microsoft Hardware and Driver Developer Information

Cancel Logic in Windows Drivers white paper

Windows security model: what every driver writer needs to know

Microsoft Windows Driver Development Kit (DDK)

See Driver Programming Techniques in Kernel-Mode Driver Architecture

Test Tools

See Windows Hardware Lab Kit in Test for performance and compatibility

Public databases of known threats and software vulnerabilities

To expand your knowledge of software threats, review the available public databases of known threats and software vulnerabilities.

- Common Vulnerabilities and Exposures (CVE): https://cve.mitre.org/
- Common Weakness Enumeration: https://cwe.mitre.org/
- Common Attack Pattern Enumeration and Classification: https://capec.mitre.org/index.html
- NIST maintains a site that describes how vulnerabilities are cataloged: https://samate.nist.gov/BF/

See Also

Driver security checklist

Windows security model for driver developers

12/15/2022 • 15 minutes to read • Edit Online

The Windows security model is based on securable objects. Each component of the operating system must ensure the security of the objects for which it is responsible. Drivers, therefore, must safeguard the security of their devices and device objects.

This topic summarizes how the Windows security model applies to kernel-mode drivers.

Windows security model

The Windows security model is based primarily on per-object rights, with a small number of system-wide privileges. Objects that can be secured include, —but are not limited to, —processes, threads, events and other synchronization objects, as well as files, directories, and devices.

For each type of object, the generic read, write, and execute rights map into detailed object-specific rights. For example, for files and directories, possible rights include the right to read or write the file or directory, the right to read or write extended file attributes, the right to traverse a directory, and the right to write an object's security descriptor.

The security model involves the following concepts:

- Security identifiers (SIDs)
- Access tokens
- Security descriptors
- Access Control Lists (ACLs)
- Privileges

Security Identifiers (SIDs)

A security identifier (SID, also called a *principal*) identifies a user, a group, or a logon session. Each user has a unique SID, which is retrieved by the operating system at logon.

SIDs are issued by an authority such as the operating system or a domain server. Some SIDs are well-known and have names as well as identifiers. For example, the SID S-1-1-0 identifies Everyone (or World).

Access tokens

Every process has an access token. The access token describes the complete security context of the process. It contains the SID of the user, the SID of the groups to which the user belongs, and the SID of the logon session, as well as a list of the system-wide privileges granted to the user.

By default, the system uses the primary access token for a process whenever a thread of the process interacts with a securable object. However, a thread can impersonate a client account. When a thread impersonates, it has an impersonation token in addition to its own primary token. The impersonation token describes the security context of the user account that the thread is impersonating. Impersonation is especially common in Remote Procedure Call (RPC) handling.

An access token that describes a restricted security context for a thread or process is called a restricted token. The SIDs in a *restricted token* can be set only to deny access, not to allow access, to securable objects. In addition, the token can describe a limited set of system-wide privileges. The user's SID and identity remain the same, but the user's access rights are limited while the process is using the restricted token. The CreateRestrictedToken function creates a restricted token.

Security descriptors

Every named Windows object has a security descriptor; some unnamed objects do, too. The security descriptor describes the owner and group SIDs for the object along with its ACLs.

An object's security descriptor is usually created by the function that creates the object. When a driver calls the IoCreateDevice or IoCreateDeviceSecure routine to create a device object, the system applies a security descriptor to the created device object and sets ACLs for the object. For most devices, ACLs are specified in the device Information (INF) file.

For more information Security Descriptors in the kernel driver documentation.

Access Control Lists

Access Control Lists (ACLs) enable fine-grained control over access to objects. An ACL is part of the security descriptor for each object.

Each ACL contains zero or more Access Control Entries (ACE). Each ACE, in turn, contains a single SID that identifies a user, group, or computer and a list of rights that are denied or allowed for that SID.

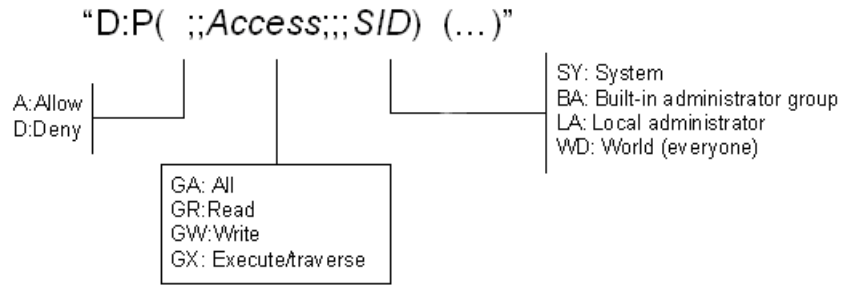
ACLs for device objects

The ACL for a device object can be set in any of three ways:

- Set in the default security descriptor for its device type.
- Created programmatically by the **RtlCreateSecurityDescriptor** function and set by the **RtlSetDaclSecurityDescriptor** function.
- Specified in Security Descriptor Definition Language (SDDL) in the device's INF file or in a call to the **IoCreateDeviceSecure** routine.

All drivers should use SDDL in the INF file to specify ACLs for their device objects.

SDDL is an extensible description language that enables components to create ACLs in a string format. SDDL is used by both user-mode and kernel-mode code. The following figure shows the format of SDDL strings for device objects.



The Access value specifies the type of access allowed. The SID value specifies a security identifier that determines to whom the Access value applies (for example, a user or group).

For example, the following SDDL string allows the System (SY) access to everything and allows everyone else (WD) only read access:

```
"D:P(A;;GA;;;SY)(A;;GR;;;WD)"
```

The header file wdmsec.h also includes a set of predefined SDDL strings that are suitable for device objects. For example, the header file defines SDDL_DEVOBJ_SYS_ALL_ADM_RWX_WORLD_RWX_RES_RWX as follows:

```
"D:P(A;;GA;;;SY)(A;;GRGWGX;;;BA)(A;;GRGWGX;;;WD)(A;;GRGWGX;;;RC)"
```

The first segment of this string allows the kernel and operating system (SY) complete control over the device.

The second segment allows anyone in the built-in Administrators group (BA) to access the entire device, but not to change the ACL. The third segment allows everyone (WD) to read or write to the device, and the fourth segment grants the same rights to untrusted code (RC). Drivers can use the predefined strings as is or as models for device-object-specific strings.

All device objects in a stack should have the same ACLs. Changing the ACLs on one device object in the stack changes the ACLs on the entire device stack.

However, adding a new device object to the stack does not change any ACLs, either those of the new device object (if it has ACLs) or those of any existing device objects in the stack. When a driver creates a new device object and attaches it to the top of the stack, the driver should copy the ACLs for the stack to the new device object by copying the **DeviceObject.Characteristics** field from the next lower driver.

The **IoCreateDeviceSecure** routine supports a subset of SDDL strings that use predefined SIDs such as WD and SY. User-mode APIs and INF files support the full SDDL syntax.

Security checks using ACLs

When a process requests access to an object, security checks compare the ACLs for the object against the SIDs in the caller's access token.

The system compares the ACEs in a strict top-down order and stops on the first relevant match. Therefore, when creating an ACL, you should always put denial ACEs above the corresponding grant ACEs. The following examples show how the comparison proceeds.

Example 1: Comparing an ACL to an access token

Example 1 shows how the system compares an ACL to the access token for a caller's process. Assume that the caller wants to open a file that has the ACL that is shown in the following table.

Sample File ACL

PERMISSION	SID	ACCESS
Allow	Accounting	Write, delete
Allow	Sales	Append
Deny	Legal	Append, write, delete
Allow	Everyone	Read

This ACL has four ACEs, which apply specifically to the Accounting, Sales, Legal, and Everyone groups.

Next, assume the access token for the requesting process contains SIDs for one user and three groups, in the following order:

User Jim (S-1-5-21...)

Group Accounting (S-1-5-22...)

Group Legal (S-1-5-23...)

Group Everyone (S-1-1-0)

When comparing a file ACL to an access token, the system first looks for an ACE for user Jim in the file's ACL. None appears, so next it looks for an ACE for the Accounting group. As shown in the previous table, an ACE for the Accounting group appears as the first entry in the file's ACL, so Jim''s process is granted the right to write or delete the file and the comparison stops. If the ACE for the Legal group instead preceded the ACE for the

Accounting group in the ACL, the process would be denied write, append, and delete access to the file.

Example 2: Comparing an ACL to a restricted token

The system compares an ACL to a restricted token in the same way that it compares those in a token that is not restricted. However, a denial SID in a restricted token can match only a Deny ACE in an ACL.

Example 2 shows how the system compares a file's ACL with a restricted token. Assume the file has the same ACL shown in the previous table. In this example, however, the process has a restricted token that contains the following SIDs:

User Jim (S-1-5-21...) Deny

Group Accounting (S-1-5-22...) Deny

Group Legal (S-1-5-23...) Deny

Group Everyone (S-1-1-0)

The file's ACL does not list Jim's SID, so the system proceeds to the Accounting group SID. Although the file's ACL has an ACE for the Accounting group, this ACE allows access; therefore, it does not match the SID in the process's restricted token, which denies access. As a result, the system proceeds to the Legal group SID. The ACL for the file contains an ACE for the Legal group that denies access, so the process cannot write, append, or delete the file.

Privileges

A privilege is the right for a user to perform a system-related operation on the local computer, such as loading a driver, changing the time, or shutting down the system.

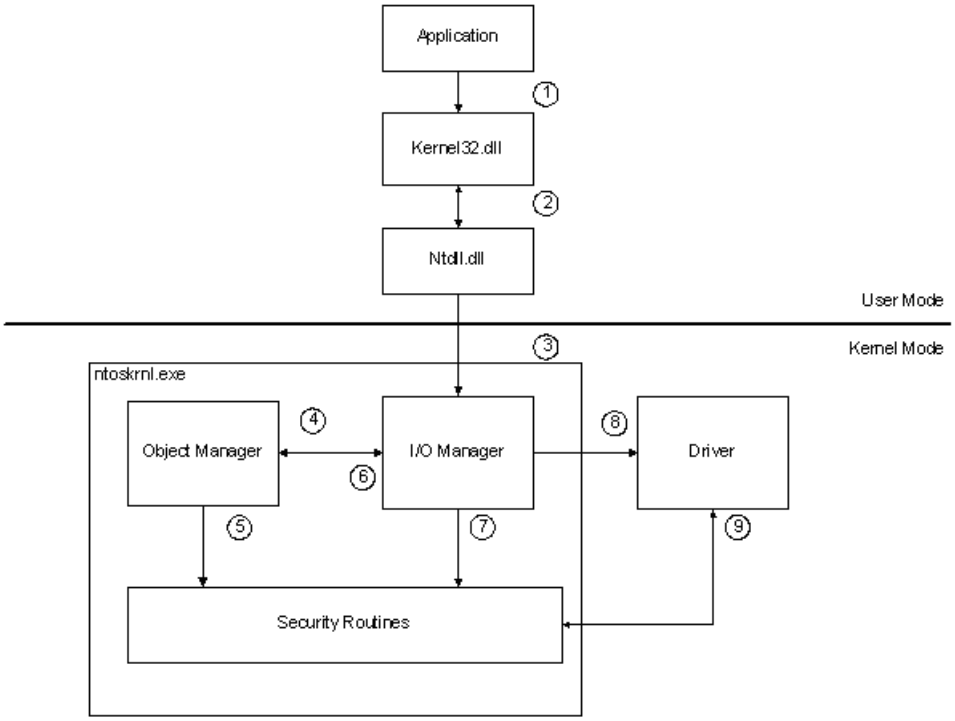
Privileges are different from access rights because they apply to system-related tasks and resources rather than objects, and because they are assigned to a user or group by a system administrator, rather than by the operating system.

The access token for each process contains a list of the privileges granted to the process. Privileges must be specifically enabled before use. For more information on privilges, see Privileges in the kernel driver documentation.

Windows security model scenario: Creating a file

The system uses the security constructs described in the Windows security model whenever a process creates a handle to a file or object.

The following diagram shows the security-related actions that are triggered when a user-mode process attempts to create a file.



The previous diagram shows how the system responds when a user-mode application calls the **CreateFile** function. The following notes refer to the circled numbers in the figure:

1. A user-mode application calls the **CreateFile** function, passing a valid Microsoft Win32 file name.
2. The user-mode Kernel32.dll passes the request to Ntdll.dll, which converts the Win32 name to a Microsoft Windows NT file name.
3. Ntdll.dll calls the **NtCreateFile** function with the Windows file name. Within Ntoskrnl.exe, the I/O Manager handles **NtCreateFile**.
4. The I/O Manager repackages the request into an Object Manager call.
5. The Object Manager resolves symbolic links and ensures that the user has traversal rights for the path in which the file will be created. For more information, see Security checks in the Object Manager.
6. The Object Manager calls the system component that owns the underlying object type associated with the request. For a file creation request, this component is the I/O Manager, which owns device objects.
7. The I/O Manager checks the security descriptor for the device object against the access token for the user's process to ensure that the user has the required access to the device. For more information, see Security checks in the I/O Manager.
8. If the user process has the required access, the I/O Manager creates a handle and sends an IRP_MJ_CREATE request to the driver for the device or file system.
9. The driver performs additional security checks as needed. For example, if the request specifies an object in the device's namespace, the driver must ensure that the caller has the required access rights. For more information, see Security checks in the driver.

Security checks in the Object Manager

The responsibility for checking access rights belongs to the highest-level component that can perform such checks. If the Object Manager can verify the caller's access rights, it does so. If not, the Object Manager passes the request to the component responsible for the underlying object type. That component, in turn, verifies access, if it can; if it cannot, it passes the request to a still-lower component, such as a driver.

The Object Manager checks ACLs for simple object types, such as events and mutex locks. For objects that have a namespace, the type owner performs security checks. For example, the I/O Manager is considered the type owner for device objects and file objects. If the Object Manager finds the name of a device object or file object when parsing a name, it hands off the name to the I/O Manager, as in the file creation scenario presented above. The I/O Manager then checks the access rights if it can. If the name specifies an object within a device

namespace, the I/O Manager in turn hands off the name to the device (or file system) driver, and that driver is responsible for validating the requested access.

Security checks in the I/O Manager

When the I/O Manager creates a handle, it checks the object's rights against the process access token and then stores the rights granted to the user along with the handle. When later I/O requests arrive, the I/O Manager checks the rights associated with the handle to ensure that the process has the right to perform the requested I/O operation. For example, if the process later requests a write operation, the I/O Manager checks the rights associated with the handle to ensure that the caller has write access to the object.

If the handle is duplicated, rights can be removed from the copy, but not added to it.

When the I/O Manager creates an object, it converts generic Win32 access modes to object-specific rights. For example, the following rights apply to files and directories:

WIN32 ACCESS MODE	OBJECT-SPECIFIC RIGHTS
GENERIC_READ	ReadData
GENERIC_WRITE	WriteData
GENERIC_EXECUTE	ReadAttributes
GENERIC_ALL	All

To create a file, a process must have traversal rights to the parent directories in the target path. For example, to create \Device\CDROM0\Directory\File.txt, a process must have the right to traverse \Device, \Device\CDROM0, and \Device\CDROM0\Directory. The I/O Manager checks only the traversal rights for these directories.

The I/O Manager checks traversal rights when it parses the file name. If the file name is a symbolic link, the I/O Manager resolves it to a full path and then checks traversal rights, starting from the root. For example, assume the symbolic link \DosDevices\D maps to the Windows NT device name \Device\CDROM0. The process must have traversal rights to the \Device directory.

For more information, see Object Handles and Object Security.

Security checks in the driver

The operating system kernel treats every driver, in effect, as a file system with its own namespace. Consequently, when a caller attempts to create an object in the device namespace, the I/O Manager checks that the process has traversal rights to the directories in the path.

With WDM drivers, the I/O Manager does not perform security checks against the namespace, unless the Device Object has been created specifying FILE_DEVICE_SECURE_OPEN. When FILE_DEVICE_SECURE_OPEN is not set, the driver is responsible for ensuring the security of its namespace. For more information, see Controlling Device Namespace Access and Securing Device Objects.

For WDF drivers, the FILE_DEVICE_SECURE_OPEN flag is always set, so that there will be a check of the device's security descriptor before allowing an application to access any names within the device's namespace. For more information, see Controlling Device Access in KMDF Drivers.

Windows security boundaries

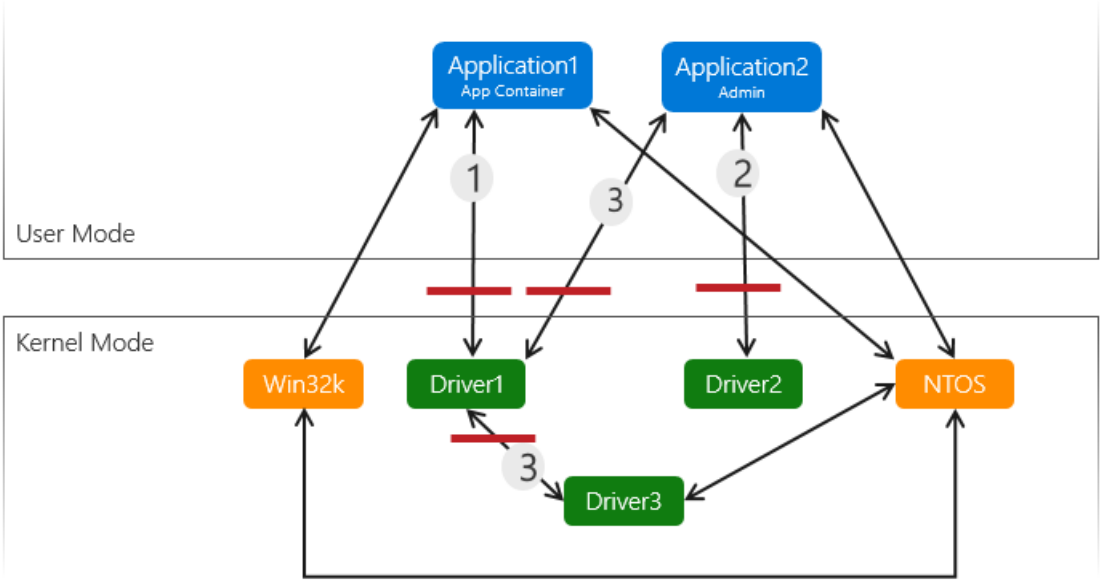
Drivers communicating with each other and to user mode callers of different privilege levels can be considered to be crossing a trust boundary. A trust boundary is any code execution path that crosses from a lower privileged process into a higher privileged process.

The higher the disparity in the privilege levels, the more interesting the boundary is for attackers that want to perform attacks such as a privilege escalation attack against the targeted driver or process.

Part of the process of creating a threat model is to examine the security boundaries and look for unanticipated paths. For more information, see Threat modeling for drivers.

Any data that crosses a trust boundary is untrusted and must be validated.

This diagram shows three kernel drivers, and two apps, one in an app container and one app that runs with admin rights. The red lines indicate example trust boundaries.



As the app container can provide additional constraints, and is not running at admin level, path (1) is a higher risk path for an escalation attack since the trust boundary is between an app container (a very low privilege process) and a kernel driver.

Path (2) is a lower risk path, as the app is running with admin rights and is calling directly into the kernel driver. Admin is already a fairly high privilege on the system so the attack surface from admin to kernel is less of an interesting target to attackers, but still a noteworthy trust boundary.

Path (3) is an example of a code execution path that crosses multiple trust boundaries that could be missed if a threat model is not created. In this example, there is a trust boundary between driver 1 and driver 3, as driver 1 takes input from the user mode app and passes it directly to driver 3.

All inputs coming into the driver from user mode is untrusted and should be validated. Inputs coming from other drivers may also be untrusted depending on whether the previous driver was just a simple pass-through (e.g. data was received by driver 1 from app 1 , driver 1 didn't do any validation on the data and just passed it forward to driver 3). Be sure to identify all attack surfaces and trust boundaries and validate all data crossing them, by creating a complete threat model.

Windows Security Model Recommendations

- Set strong default ACLs in calls to the **IoCreateDeviceSecure** routine.
- Specify ACLs in the INF file for each device. These ACLs can loosen tight default ACLs if necessary.
- Set the FILE_DEVICE_SECURE_OPEN characteristic to apply device object security settings to the device namespace.
- Do not define IOCTLs that permit FILE_ANY_ACCESS unless such access cannot be exploited maliciously.
- Use the **IoValidateDeviceIoControlAccess** routine to tighten security on existing IOCTLS that allow FILE_ANY_ACCESS.
- Create a threat model to examine the security boundaries and look for unanticipated paths. For more

information, see Threat modeling for drivers.

- See Driver security checklist for additional driver security recommendations.

See Also

Securing Device Objects

Driver security checklist

Implement HVCI compatible code

12/15/2022 • 4 minutes to read • Edit Online

This section describes how to implement Hypervisor-protected Code Integrity (HVCI) compatible code.

To implement HVCI compatible code, make sure your driver code does the following:

- Opts in to NX by default
- Uses NX APIs/flags for memory allocation (NonPagedPoolNx)
- Does not use sections that are both writable and executable
- Does not attempt to directly modify executable system memory
- Does not use dynamic code in kernel
- Does not load data files as executable
- Section alignment is a multiple of 0x1000 (PAGE_SIZE). E.g. DRIVER_ALIGNMENT=0x1000

The following list of DDIs that are not reserved for system use may be impacted:

DDI NAME
ExAllocatePool
ExAllocatePoolWithQuota
ExAllocatePoolWithQuotaTag
ExAllocatePoolWithTag
ExAllocatePoolWithTagPriority
ExInitializeNPagedLookasideList
ExInitializeLookasideListEx
MmAllocateContiguousMemory
MmAllocateContiguousMemorySpecifyCache
MmAllocateContiguousMemorySpecifyCacheNode
MmAllocateContiguousNodeMemory
MmCopyMemory
MmMapIoSpace
MmMapLockedPages
MmMapLockedPagesSpecifyCache

DDI NAME
MmProtectMdlSystemAddress
ZwAllocateVirtualMemory
ZwCreateSection
ZwMapViewOfSection
NtCreateSection
NtMapViewOfSection
ClfsCreateMarshallingArea
NDIS
NdisAllocateMemoryWithTagPriority
Storage
StorPortGetDataInBufferSystemAddress
StorPortGetSystemAddress
ChangerClassAllocatePool
Display
DxgkCbMapMemory
VideoPortAllocatePool
Audio Miniport
IMiniportDMus::NewStream
IMiniportMidi::NewStream
IMiniportWaveCyclic::NewStream
IPortWavePci::NewMasterDmaChannel
IMiniportWavePci::NewStream
Audio Port Class
PcNewDmaChannel
PcNewResourceList

DDI NAME
PcNewResourceSublist
IFS
FltAllocatePoolAlignedWithTag
FltAllocateContext
WDF
WdfLookasideListCreate
WdfMemoryCreate
WdfDeviceAllocAndQueryProperty
WdfDeviceAllocAndQueryPropertyEx
WdfFdoInitAllocAndQueryProperty
WdfFdoInitAllocAndQueryPropertyEx
WdfIoTargetAllocAndQueryTargetProperty
WdfRegistryQueryMemory

Use the code integrity tests in the HLK to test HVCI driver compatibility:

For more information about the related system fundamentals security test, see HyperVisor Code Integrity Readiness Test and Hypervisor-Protected Code Integrity (HVCI).

For more information about the related device fundamentals test, see Device.DevFund tests.

Use the following table to interpret the output and determine what driver code changes are needed to fix the different types of HVCI incompatibilities.

Warning	Redemption
Execute Pool Type	The caller specified an executable pool type. Calling a memory allocating function that requests executable memory. Be sure that all pool types contain a non executable NX flag.
Execute Page Protection	The caller specified an executable page protection. Specify a "no execute" page protection mask.

Execute Page Mapping	The caller specified an executable memory descriptor list (MDL) mapping. Make sure that the mask that is used contains MdlMappingNoExecute. For more information, see MmGetSystemAddressForMdlSafe
Execute-Write Section	The image contains an executable and writable section.
Section Alignment Failures	The image contains a section that is not page aligned. Section Alignment must be a multiple of 0x1000 (PAGE_SIZE). E.g. DRIVER_ALIGNMENT=0x1000
IAT in Executable Section	The import address table (IAT), should not be an executable section of memory. This issue occurs when the IAT, is located in a Read and Execute (RX) only section of memory. This means that the OS will not be able to write to the IAT to set the correct addresses for where the referenced DLL. One way that this can occur is when using the /MERGE (Combine Sections) option in code linking. For example if .rdata (Read-only initialized data) is merged with .text data (Executable code), it is possible that the IAT may end up in an executable section of memory.

Unsupported Relocs

In Windows 10, version 1507 through Windows 10, version 1607, because of the use of Address Space Layout Randomization (ASLR) an issue can arise with address alignment and memory relocation. The operating system needs to relocate the address from where the linker set its default base address to the actual location that ASLR assigned. This relocation cannot straddle a page boundary. For example, consider a 64-bit address value that starts at offset 0x3FFC in a page. It's address value overlaps over to the next page at offset 0x0003. This type of overlapping relocs is not supported prior to Windows 10, version 1703.

This situation can occur when a global struct type variable initializer has a misaligned pointer to another global, laid out in such a way that the linker cannot move the variable to avoid the straddling relocation. The linker will attempt to move the variable, but there are situations where it may not be able to do so (for example with large misaligned structs or large arrays of misaligned structs). Where appropriate, modules should be assembled using the /Gy (COMDAT) option to allow the linker to align module code as much as possible.

```
#include <pshpack1.h>

typedef struct _BAD_STRUCT {
      USHORT Value;
      CONST CHAR *String;
} BAD_STRUCT, * PBAD_STRUCT;

#include <poppack.h>

#define BAD_INITIALIZER0 { 0, "BAD_STRING" },
#define BAD_INITIALIZER1 \
      BAD_INITIALIZER0      \
      BAD_INITIALIZER0      \
      BAD_INITIALIZER0      \
      BAD_INITIALIZER0      \
      BAD_INITIALIZER0      \
      BAD_INITIALIZER0      \
      BAD_INITIALIZER0      \
      BAD_INITIALIZER0

#define BAD_INITIALIZER2 \
      BAD_INITIALIZER1      \
      BAD_INITIALIZER1      \
      BAD_INITIALIZER1      \
      BAD_INITIALIZER1      \
      BAD_INITIALIZER1      \
      BAD_INITIALIZER1      \
      BAD_INITIALIZER1      \
      BAD_INITIALIZER1

#define BAD_INITIALIZER3 \
      BAD_INITIALIZER2      \
      BAD_INITIALIZER2      \
      BAD_INITIALIZER2      \
      BAD_INITIALIZER2      \
      BAD_INITIALIZER2      \
      BAD_INITIALIZER2      \
      BAD_INITIALIZER2      \
      BAD_INITIALIZER2

#define BAD_INITIALIZER4 \
      BAD_INITIALIZER3      \
      BAD_INITIALIZER3      \
      BAD_INITIALIZER3      \
      BAD_INITIALIZER3      \
      BAD_INITIALIZER3      \
      BAD_INITIALIZER3      \
      BAD_INITIALIZER3      \
      BAD_INITIALIZER3

BAD_STRUCT MayHaveStraddleRelocations[4096] = { // as a global variable
      BAD_INITIALIZER4
};
```

There are other situations involving the use of assembler code, where this issue can also occur.

Driver Verifier code integrity

Use the Driver Verifier code integrity option flag (0x02000000) to enable extra checks that validate compliance with this feature. To enable this from the command line, use the following command.

```
verifier.exe /flags 0x02000000 /driver <driver.sys>
```

To choose this option if using the verifier GUI, select *Create custom settings* (for code developers), select *Next*, and then select *Code integrity checks*.

You can use the verifier command line /query option to display the current driver verifier information.

```
verifier /query
```

See Also

Driver security checklist

Windows drivers security best practices for driver developers

12/15/2022 • 5 minutes to read • Edit Online

This topic summarizes the unsafe development patterns which can lead to exploitation and abuse of your Windows driver code. This topic provides development recommendations and code samples. Following these best practices will help improve the safety of performing privileged behavior in the Windows kernel.

Unsafe driver behavior overview

While it is expected that Windows drivers perform high privileged behavior in kernel mode, not performing security checks and adding constraints on privileged behavior is securely is unacceptable. The Windows Hardware Compatibility Program (WHCP), formerly WHQL, requires new driver submissions to comply with this requirement.

Examples of unsecure and dangerous behavior includes, but is not limited to, the following:

- Providing the ability to read and write to arbitrary machine specific registers (MSRs)
- Providing the ability to read and write to Port input and output
- Providing the ability to read and write kernel, physical, or device memory

Providing the ability to read and write MSRs

Enhancing the security of reading from MSRs

In the first ReadMsr example, the driver allows for unsafe behavior by allowing for any and all registers to be arbitrarily read. This can result in abuse by malicious processes in user mode.

```
Func ReadMsr(int dwMsrIdx)
{
 int value = __readmsr(dwMsrIdx); // Unsafe, can read from any MSR
 return value;
}
```

If your scenario requires reading from MSRs, the driver must always check that the register to read from is constrained to the expected index or range. Two examples of how to implement the safe read operation follow.

```
Func ConstrainedReadMsr(int dwMsrIdx)
{
    int value = 0;
    if (dwMsrIdx == expected_index) // Blocks from reading anything
    {
        value = __readmsr(dwMsrIdx); // Can only read the expected MSR
    }
    else
    {
        return error;
    }
    return value;
}

// OR

Func ConstrainedReadMsr(int dwMsrIdx)
{
    int value = 0;
    if (min_range <= dwMsrIdx <= max_range) // Blocks from reading anything
    {
        value = __readmsr(dwMsrIdx); // Can only from the expected range of MSRs
    }
    else
    {
        return error;
    }
    return value;
}
```

Enhancing the security of writing to MSRs

In the first WriteMsr example, the driver allows for unsafe behavior by allowing for any and all registers to be arbitrarily written to. This can result in abuse by malicious processes to elevate privilege in user mode and write to all MSRs.

```
Func WriteMsr(int dwMsrIdx)
{
 int value = __writemsr(dwMsrIdx); // Unsafe, can write to any MSR
 return value;
}
```

If your scenario requires writing to MSRs, the driver must always check that the register to write to is constrained to the expected index or range. Two examples of how to implement the safe write operation follow.

```
Func ConstrainedWriteMsr(int dwMsrIdx)
{
    int value = 0;
    if (dwMsrIdx == expected_index) // Blocks from reading anything
    {
        value = __writemsr(dwMsrIdx); // Can only write to the expected constrained MSR
    }
    else
    {
        return error;
    }
    return value;
}

// OR

Func ConstrainedWriteMSR(int dwMsrIdx)
{
    int value = 0;
    if (min_range <= dwMsrIdx <= max_range) // Blocks from reading anything
    {
        value = __writemsr(dwMsrIdx); // Can only write to the expected constrained MSR
    }
    else
    {
        return error;
    }
    return value;
}
```

Providing the ability to read and write to Port input and output

Enhancing the security of reading from Port IO

Caution must be used, when providing the ability to ability to read to Port input/output (I/O). This code example is unsafe.

```
Func ArbitraryInputPort(int inPort)
{
 dwResult = __indword(inPort); // Unsafe, allows for arbitrary reading from Input Port
 return dwResult;
}
```

To prevent the abuse and exploit of the driver, the expected input port must be constrained to the required usage boundary.

```
Func ConstrainedInputPort(int inPort)
{
 // The expected input port must be constrained to the required usage boundary to prevent abuse
 if(inPort == expected_InPort)
 {
  dwResult = __indword(inPort);
 }
 else
 {
  return error;
 }
 return dwResult;
}
```

Enhancing the security of writing to Port IO

Caution must be used, when providing the ability to ability to write to Port input/output (I/O). This code example is unsafe.

```
Func ArbitraryOutputPort(int outPort, DWORD dwValue)
{
 __outdword(OutPort, dwValue); // Unsafe, allows for arbitrary writing to Output Port
}
```

To prevent the abuse and exploit of the driver, the expected input port must be constrained to the required usage boundary.

```
Func ConstrainedOutputPort(int outPort, DWORD dwValue)
{
 // The expected output port must be constrained to the required usage boundary to prevent abuse
 if(outPort == expected_OutputPort)
 {
  __outdword(OutPort, dwValue); // checks on InputPort
 }
 else
 {
  return error;
 }
}
```

Providing the ability to read and write kernel, physical, or device memory

Enhancing the security of Memcpy

This sample code shows unconstrained and unsafe use of safe use of physical memory.

```
Func ArbitraryMemoryCopy(src, dst, length)
{
 memcpy(dst, src, length); // Unsafe, can read and write anything from physical memory
}
```

If your scenario requires reading and writing kernel, physical or device memory, the driver must always check that the source and destinations are constrained to the expected indices or ranges.

```
Func ConstrainedMemoryCopy(src, dst, length)
{
 // valid_src and valid_dst must be constrained to required usage boundary to prevent abuse
 if(src == valid_Src && dst == valid_Dst)
 {
  memcpy(dst, src, length);
 }
 else
 {
  return error;
 }
}
```

Enhancing the security of ZwMapViewOfSection

The following example illustrates the unsafe and improper method to read and write physical memory from user mode utilizing the ZwOpenSection and ZwMapViewOfSection APIs.

```
Func ArbitraryMap(PHYSICAL_ADDRESS Address)
{
 ZwOpenSection(&hSection, ... ,"\Device\PhysicalMemory");
 ZwMapViewOfSection(hSection, -1, 0, 0, 0, Address, ...);
}
```

To prevent the abuse and exploit of the driver's read/write behavior by malicious user mode processes, the driver must validate the input address and constrain the memory mapping only to the required usage boundary for the scenario.

```
Func ConstrainedMap(PHYSICAL_ADDRESS paAddress)
{
 // expected_Address must be constrained to required usage boundary to prevent abuse
 if(paAddress == expected_Address)
 {
  ZwOpenSection(&hSection, ... ,"\Device\PhysicalMemory");
  ZwMapViewOfSection(hSection, -1, 0, 0, 0, paAddress, ...);
 }
 else
 {
  return error;
 }
}
```

Enhancing the security of MmMapLockedPagesSpecifyCache

The following example illustrates the unsafe and improper method to read and write physical memory from user mode utilizing the MmMapIoSpace, IoAllocateMdl and MmMapLockedPagesSpecifyCache APIs.

```
Func ArbitraryMap(PHYSICAL_ADDRESS paAddress)
{
 lpAddress = MmMapIoSpace(paAddress, qwSize, ...);
 pMdl = IoAllocateMdl( lpAddress, ...);
 MmMapLockedPagesSpecifyCache(pMdl, UserMode, ... );
}
```

To prevent the abuse and exploit of the driver's read/write behavior by malicious user mode processes, the driver must validate the input address and constrain the memory mapping only to the required usage boundary for the scenario.

```
Func ConstrainedMap(PHYSICAL_ADDRESS paAddress)
{
 // expected_Address must be constrained to required usage boundary to prevent abuse
 if(paAddress == expected_Address && qwSize == valid_Size)
 {
  lpAddress = MmMapIoSpace(paAddress, qwSize, ...);
  pMdl = IoAllocateMdl( lpAddress, ...);
  MmMapLockedPagesSpecifyCache(pMdl, UserMode, ... );
 }
 else
 {
  return error;
 }
}
```

See Also

Driver security checklist