David Lariviere
Hardware/Software Formal Verification
Final Report

Windows Driver Verification

Introduction:

The purpose of my project was to understand and evaluate currently available formal verification tools from Microsoft for the purpose of verifying window drivers. I eventually applied combinations of the Microsoft Verification tools to four separate drivers. The first was an extremely simple "Hello World" driver, based on a simple introduction to Windows Drivers found online. The second was a PortTalk, an open source driver for communicating with the parallel port that I first proposed as the target of my project. The third was an example driver included in the Microsoft Driver Kit for use with SDV. Finally, the fourth was another custom driver, based off of several helpful online driver tutorials, accompanied with a user-mode test program for communicating with and testing the driver.

Software: The project required a large number of software programs, both from Microsoft and third parties.

Non-Verification Related Software:

- Microsoft Windows Driver Development Kit (DDK): Build 3790.1830
- Microsoft Windows Driver Kit (WDK): Build 6000
- Microsoft Platform Development Kit: The primary source of Microsoft APIs/Documentation.
- Microsoft Visual Studio .NET: IDE used for writing the driver.
- DbgView: a tool used for viewing kernel-level debug statements.
- OS Loader: a third party tool for loading/unloading and installing/uninstalling drivers.
- PortTalk: an open-source Windows driver for accessing the parallel port.

Verification Tools:

- PREfast:
- Static Driver Verifier:

Case Study 1: Verification of “HelloWorld” Driver.

The first driver I wrote was based off of a tutorial describing the absolute simplest driver possible, HelloWorld. It consists of nothing more than a Driver Entry function, the driver equivalent of main() for user mode programs.

The complete code is shown below:

```
#include <ntddk.h>

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath)
{
    DbgPrint("Hello, World\n");

    return STATUS_SUCCESS;
}
```

First for the project, I downloaded the publicly available Windows Driver Development Kit (DDK). I had previously investigated windows driver development in the past, so I knew about the DDK as the driver equivalent of the Windows Platform Software Development Kit (PSDK), a familiar required installation for Windows Programmers.

The first thing that confused me in getting the development environment set up was the slightly ambiguous naming Microsoft had for the different packages. A googling of DDK brings one to here. The page, titled “Windows Driver Development Kit” offers a direct download of the DDK. It also contained a reference at the bottom about the “Windows Driver Kit.” Slightly confused, I proceeded to download and install the DDK.

With the DDK installed, it was now time to compile my very first driver. Following the directions of [1], it compiled without any trouble or warnings. With the DDK documentation in hand, I looked up the directions for using PREfast. It was actually very simple, instead of running “build” one executes “prefast build”. PREfast actually works as part of the build process. PREfast, to my joy and amazement, reported “no defects were detected during execution of the command.”

Now it was time to actually install and run my first driver. Next I downloaded and installed OSLoader, in order to actually install the driver and start it. Drivers are considered just another service within the Windows framework. OSLoader takes care of installing/uninstalling the driver as a service within windows. “Hello” installed without errors. Executing “net start hello” on the windows command line started the (functionless) driver without error.

The surprise came when I tried to stop the driver in order to build and install a slightly modified version. Executing “net stop hello” reported an error. I also tried using the commands in OSLoader to either stop the driver, or just uninstall it completely. Both refused to do so. At this point I was somewhat confused. Reading the second half of the tutorial in [1], however, I realized it was because the driver lacked a DriverUnload function, which is analogous to a deconstructor in C++.

At this point, I was slightly disturbed by the fact that the PREfast had not generated any warnings. It was, perhaps, understandable that the Microsoft compiler was not complaining, but I would've figured PREfast would catch something as egregious as a missing and very much required Unload function.

One even more disturbing hint came from the MSDN docs which state that:

> *The Unload routine is required for WDM drivers and **optional for non-WDM** drivers. A driver's Unload routine, if supplied, should be named XxxUnload, where Xxx is a driver-specific prefix. The driver's DriverEntry routine must store the Unload routine's address in DriverObject->DriverUnload. (If no routine is supplied, this pointer must be NULL.)*

The Hello driver, as far as I can tell based on tutorials and documentation, is not a WDM driver specific driver. It doesn't include <wdm.h>, but rather the older <ntddk.h>, found for drivers dating back to pre WDM Windows NT drivers. Nonetheless, without the unload routine, Windows failed to stop the driver.

Unable to stop the driver, I restarted my machine in order to unload the driver. Following the directions of [1] I modified the "Hello" driver, adding DriverUnload function, and registering it within the DriverEntry function. After recompiling the driver again, I could now install/uninstall and start/stop the driver without problems.

At this point I thought it might be a good time to move onto a more complicated driver.

Case Study 2: Verification of the PortTalk driver.
Overview: PortTalk [5] is an open-source driver written by Craig Peacock. Its intention was to allow Windows programmers to access the parallel port on Windows NT derived machines which implemented protected memory management, and thus caused software which functioned correctly on Win9x and earlier to fail on those versions of Windows which implemented protected memory (Win NT, Win2k, WinXP, and up).

Implementation: PortTalk was originally written for Windows NT4 and therefore actually adheres to a slightly older version of Windows drivers predating the Windows Driver Model. Luckily WDM encompasses all of the older functionality that PortTalk uses. For communication between user mode and the driver, PortTalk relies upon the Win32 DeviceIoControl system call. It is a method intended to allow user level programs to communicate directly with Windows drivers. As such, besides the required Driver entry and exit points required by all drivers, PortTalk implements a DeviceIoControl handler which then processes the user's requested action.

For the actual communication with the Parallel port, PortTalk still does not utilize direct access to the memory-mapped locations corresponding to the parallel ports. Rather it takes advantage of two WDK macros, READ_PORT_UCHAR and WRITE_PORT_UCHAR, for writing and reading bytes to and from the designated. It is

worth mentioning that both macros, according to the documentation, can be used during any IRQL, thereby avoiding any restrictions within the driver utilizing them, in terms of IRQL requirements.

Compiling PortTalk:

I've used pre-compiled versions of PortTalk before on some personal hobby projects involving ADCs directly wired up to the parallel port. Unfortunately, in the past I was never able to get the actual driver compiling, since it required more than the standard windows build environment provided by Visual Studio + Platform SDK.
With the DDK now having been previously installed for "Hello", however, I was good to go.

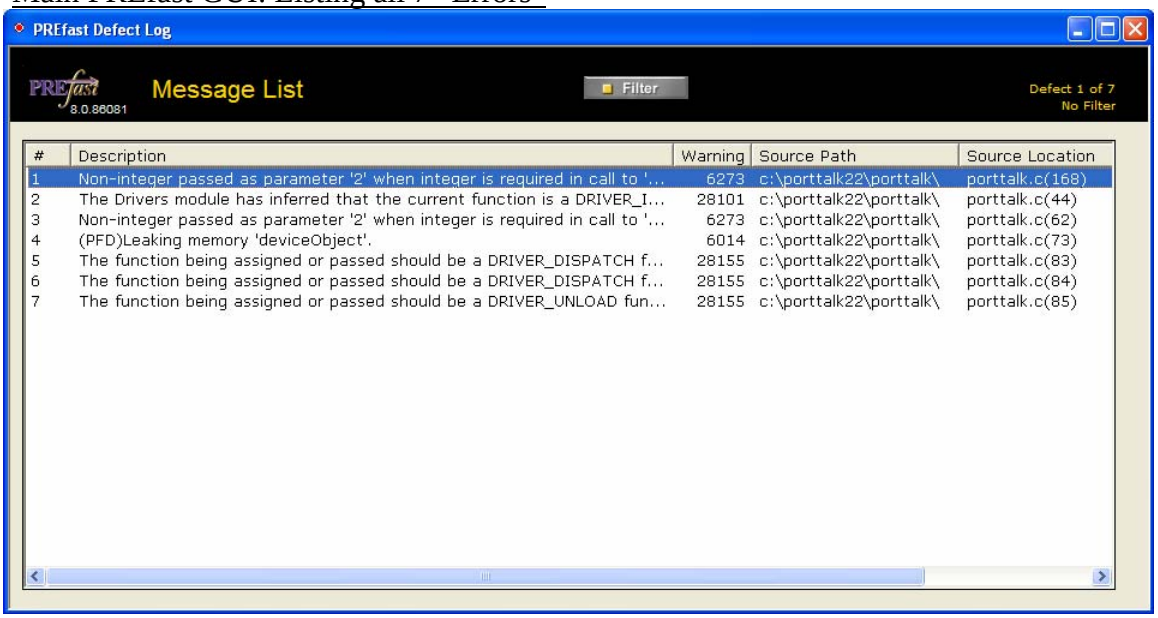
PortTalk compiled without any problems. The next stop was to verify the driver with PREfast and SDV.

Running PREfast with "prefast build" ran quite quickly, on the order of a few seconds. The complete print out was as follows:

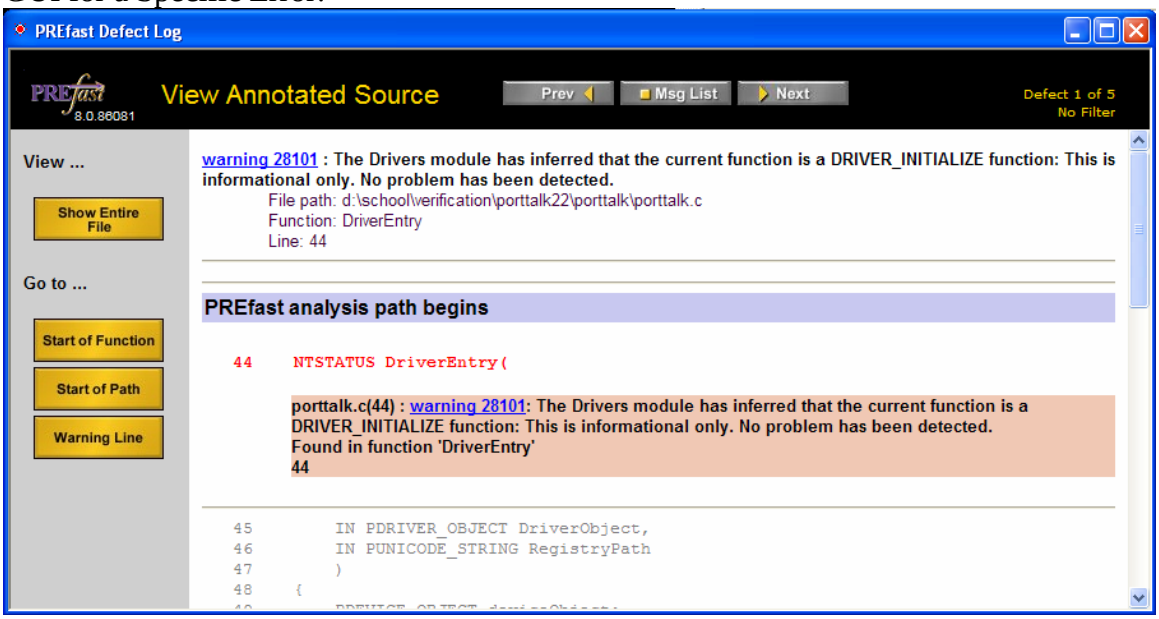
```
C:\porttalk22\Porttalk>prefast build
------------------------------------------------------------------------------
Microsoft (R) PREfast Version 8.0.86081.
Copyright (C) Microsoft Corporation. All rights reserved.
------------------------------------------------------------------------------
BUILD: Compile and Link for x86
BUILD: Loading c:\winddk\6000\build.dat...

BUILD: Computing Include file dependencies:
BUILD: Start time: Thu Dec 21 19:21:59 2006
BUILD: Examining c:\porttalk22\porttalk directory for files to compile.
    c:\porttalk22\porttalk
BUILD: Saving c:\winddk\6000\build.dat...
BUILD: Compiling and Linking c:\porttalk22\porttalk directory
_NT_TARGET_VERSION SET TO WINXP
Compiling - porttalk.rc
Compiling - porttalk.c
Compiling - porttalk.c
Linking Executable - i386\porttalk.sys
BUILD: Finish time: Thu Dec 21 19:22:03 2006
BUILD: Done

    5 files compiled - 2 Warnings -   228 LPS
    1 executable built
------------------------------------------------------------------------------
Removing duplicate defects from the log...
------------------------------------------------------------------------------
PREfast reported 7 defects during execution of the command.
------------------------------------------------------------------------------
Enter PREFAST LIST to list the defect log as text within the console.
Enter PREFAST VIEW to display the defect log user interface.
```

As can be seen above, 7 defects were found. Executing "prefast view", I launched the PREfast GUI to examine the problems.

Main PREfast GUI: Listing all 7 “Errors”



GUI for a Specific Error:



The PREfast GUI is very nicely done. It is straight forward, completely intuitive, and easy to use. Click on each error in the list brings up another screen (see above) which displays where in the code the error is occurring, and a description of the error.

Analysis of PREfast errors:

1. The first Warning/Error found by PREfast simple states that:

> **The Drivers module has inferred that the current function is a DRIVER_INITIALIZE function: This is informational only. No problem has been detected.0**
> **Found in function 'DriverEntry'**

At first I was slightly perplexed and amused as to why PREfast would always bother to report this warning message. Later, through experience with SDV, I realized that the reason was to inform the programmer that PREfast was indeed able to correctly determine the DriverEntry function. Just as with C programs, the compiler generally assumes a certain set of default function names for where the primary entry point into the code should be. This parameter is usually configurable, however, and therefore it is possible that the user could be using a custom name for the initial DriverEntry function. PREfast reporting the entry informs the programmer that it was indeed able to correctly identify what the entry function was, rather than perhaps not finding the entry, and therefore not scanning it, and incorrectly reporting that no errors were found as a result of not entering the code rather than from the DriverEntry actually being correct.

2. The second warning was about a possible memory leak, with respect to passing a pointer of a variable allocated on the stack to a function. In this specific case, however, it was not actually a memory error. One common programming paradigm, especially for Microsoft within its APIs, is for programmers to pass in double pointers to system calls, so that the system call can set the pointer to a created object. This is as opposed to perhaps relying on return values, which Microsoft most often uses strictly for return codes indicating success or failure.

This is one example that I am somewhat surprised PREfast does not explicitly know to avoid. Virtually every driver must use this function, and checking a lot of the example drivers included with DDK, all of them use this and will generate the error if verified with PREfast. This goes back to PREfast's history as a general tool for finding common programming mistakes, rather than specifically for drivers. Nonetheless, hopefully future versions will be better tuned toward common driver situations.

3. Two of the warnings referred to the operand type ("%X") being used in a KdPrint (similar to printf) call, in which the corresponding argument was a pointer, not an integer. While this may seem trivial at first, for 64bit systems it could actually have a significant impact. For years Windows systems, and thus windows driver writers, have gotten used to 32bit systems being standard. As such the standard 4byte integer and pointers both had the same size in memory and could also be used interchangeably (though sometimes incorrectly) with casting. With 64bit systems, however, this is no longer the case, as the pointers are now 64 bits long. The same issue even applies to newer chips which, although still 32-bit, have extended memory features which expand the memory space of the system to 64 bits. If a 64 bit pointer was passed to a function expecting a 32 bit integer, it could cause problems.

This example was probably one of the best examples of where PREfast can be quite helpful in pointing obscure points that one might not otherwise thinking about, especially if running on a 32 bit machine, or perhaps unaware that Microsoft offered a "pointer-type" specific operand just for insuring future architecture independence. Not only did it provide the error, it even suggested the correct solution, utilizing %p instead.

4. The last set of errors consisted of three of the same warning. They were complaining about the return type of the various Driver functions for handling specific events. Two were declared as having a return type of NT_STATUS, and the third with void. PREfast stated that the functions should have a return type of DRIVER_DISPATCH or DRIVER_UNLOAD, respectively. Unfortunately, after making the changes, the driver refused to compile. As will be discussed later, the same original warnings came up on verifying the last driver as well, but changing them there still led to a successful compile. Why the difference? The PortTalk driver, as mentioned before, is based on an older driver framework, utilizing "ntddk.h." It most likely does not have the definitions for the DRIVER_DISPATCH or DRIVER_UNLOAD, which were added later. Unfortunately, PREfast does seem to be completely backwards compatible with older drivers, as some of its warnings suggests changes that will break the build.

Interesting side note: when I first ran PREfast against PortTalk, I did so using the Windows Driver Development Kit (DDK). While writing up this report, however, I ran it again in order to grab the screenshots and output. In doing it discovered 3 more errors than the last time, specifically the last set just mentioned. The reason was because I first started with the DDK, but later, for reasons which will be explained further on, I had to acquire the Windows Driver Kit (which despite the extremely similar name is quite separate and different from the Windows Driver Development Kit). This shows that Microsoft is continuing work on PREfast and has been adding additional error checks to the latest versions of PREfast.

With all of the suggested changes that made sense or would still compile made, the PREfast verification stage was done.

Verification with Static Driver Verifier:

Whereas PREfast ran in a matter of seconds, SDV took roughly an hour and a half to complete, running several minutes for each of a few dozen rules, most which didn't even apply to the PortTalk driver. After the long await, anxiously awaiting the results, SDV reported… "66 Not Applicable", meaning none of the 66 relevant SDV rules applied to the PortTalk driver.

Thus PortTalk, according to SDV, PortTalk passes, albeit by default.

Case 3, Example SDV Failed Driver:

In learning how to use SDV, I followed one of the provided examples, a driver called "failed_1." It is a very simple driver with several function handlers for various driver events, each which implements a distinct one or two line common mistake which SDV checks for.

Testing:

- CancelSpinLock
- IrpProcessingCompleteLock
- LowerDriverReturn
- NullExFreePool
- SpinLock

The tests include examples like ensuring that the user either remembered to release a lock before returning, or that actually acquired the lock before trying to release it, and that null pointers (but only null pointers, not necessarily a valid pointer) cannot be freed.

The example failed drivers, and the fact that they did indeed fail in according to SDV, would be useful for in trying to verify case 4.

Case 4, Eightball Driver:

As part of the project, I decided to try to implement my own driver. After googling extensively, I came upon very few good resources or tutorials related to driver development. Luckily, at least one of them, was a gold mind. In the Device Drivers section of The Code Project [3], a community website for programmers, was a 6 part tutorial on driver writing.

The tutorials walked through the process of writing a file-based driver that one could open up in a user-mode program, and make read and write calls to.

As a target application, I decided on a toy example named the "Eightball" driver. The idea was to have a user mode program write questions to the eightball driver using standard file write calls, for example "Will it rain tomorrow?". The user mode program then makes a subsequent read call to the driver to receive an answer, "Ask again later," for example.

Implementation:

I developed the eightball driver in stages. First by combining snippets of code from the tutorial with my Hello World driver, and then by implementing more and more of the features. During each round of changes, I would verify the driver with PREfast, make the recommend changes until PREfast was satisfied, and then continue developing the program.

Stage 1: The first stage involved adding just the DriverEntry and DriverUnload functions. Running PREfast first complained that the DriverEntry function was lacking a statement which registered the DriverUnload function. Note: when I ran PREfast on the extremely simple "HelloWorld" driver, it did not make this recommendation. The reason most likely goes back to that driver including "ntddk.h", whereas the eightball driver utilizes the newer "wdm.h". While either way it is actually seems mandatory (contrary to MSDN documentation), the use of WDM seemed to signal PREfast to pay closer attention to this.

After making the recommended change of return type and recompiling with PREfast again, it next complained (as it did in the PortTalk example) that the DriverUnload function should have a return type of DRIVER_UNLOAD. This time, however, making the change still compiled, in contrast to the previous attempt with PortTalk which utilized "ntddk.h".

Rebuilding with PREfast again now responded with no relevant errors (beyond some warnings that always appear which were previously described and are either just informational or false positives).

Stage 2:

With the basic stubs now in place, and being able to install and start the (functionless) driver, I next proceeded to introduce more of the code from the driver tutorial, adding a full DriverEntry which registered handler functions for various calls (read and write, for example), as well as the actual write system calls.

Running PREfast on the updated version resulted in additional warnings. First, it wanted all of the dispatch function calls (for things like read and write) defined with a return type of DRIVER_DISPATCH.

The second error was derived from a series of pragmas which the tutorial had in one of the files. In the tutorial, all of the function calls are marked with:

```
#pragma alloc_text(PAGE, *NameOfFunction*)
```

After googling and researching the MS documentation, it turns out the pragmas indicate to the compiler/OS that the code sections are indeed pageable. This differs from drivers which might be running at a different (lower) IRQ Level, some of which disable all

interrupts, including pagefault interrupts. Drivers running in these IRQLs cannot have pageable code, since if the code sections of the driver become paged with it running in the higher IRQ level, it will be unable to execute the functions and will likely crash the machine.

The recommendation of PREfast, given that the functions were all marked with the pageable pragma, was to insert the PAGED_CODE(); macro at the beginning of every function. This is simply a wrapper for an assert macro which checks that the IRQ Level is not high enough to disable paging, thus preventing this function from running if it was designated as pageable.

I liked this particular example because it seemed to demonstrate another benefit of using PREfast, good coding style. While sometimes the warnings are actually errors, in this case it is more to help the developer follow best practices in writing drivers, and thus make debugging easier down the road.

The last error in this round involved the use of an all encompassing try/catch of EXCEPTION_EXECUTE_HANDLER, which includes all possible exceptions, rather than just the specific two exceptions that a particular function, ProbeForRead, can throw. While not quite sure if the solution was valid, after several failed attempts at catching multiple specific exceptions in a single try catch, I nested two try/catches around the code block which previously just used a single catch. Each level caught one of the two specific exceptions that the code block could raise. When researching the problem, it seemed the reason fro the warning was that one does not want to catch unexpected exceptions, as the reason may be beyond the scope of the program, and it may not be the responsibility of the specific code block to deal with the exception. By not catching other exceptions that are not expected for the specific code, it means that the exception will travel up the stack to other drivers or kernel handlers, one of which may actually responsible for dealing with the exception properly, rather than stopping the exception from being propagated with overly ambiguous catch statements.

Stage 3:

Stage 3 involved implementing the provided user mode program responsible for accessing the driver, in order to make read and write calls to it. As of stage 2, the driver only supported writes to it. Therefore the user mode program passed a string to the driver which it then printed using the kernel debug print functions. After installing the latest version of the compiled driver and starting it, I ran the user mode program. It successfully opened up the driver and wrote a message to the driver, which successfully printed it to the kernel debug (as displayed using [6]).

Stage 4.

In Stage 4, following part 2 of the tutorial, I added Read functionality to the driver, allowing it to return data to the user mode program, rather than just receiving it. Knowing some of the things that PREfast was looking for, I proactively added some code I knew PREfast might otherwise complain about. Running PREfast resulted in no additional warnings.

Next I went on to extend the user mode program, having it actually write the data to question (as a string) to the driver, and then read the response. At this point the response was hard coded as “Ask again later.”

Running the latest combo worked as expected.

Stage 5:

In stage 5, I really started trying to change things. I decided to add a global integer variable to keep track of whether the user had asked a question yet, before returning a response. The idea being that the eightball will only answer (return a valid “Eightball” response) if the user had written a question beforehand. Otherwise it would return a message telling the user mode program that it should ask a question first.

Unfortunately it was around this point that things soon went bad. Running PREfast did not come up with any additional errors.

First I decided to test the new code by modifying the user mode program to test each situation. The first version of the program made an immediate read to the Eightball driver without first writing anything (i.e. a question). This test worked, as the driver properly returned `"You didn't ask Eightball a question yet!"`

The next case involved making sure a valid eightball response was still returned if the user mode program had indeed written data to the eightball driver first. This test also passed, as the usermode wrote a question, read the response and printed to the console: `"It is certain".`

Feeling slightly more confident, I next compiled a version of the user mode program to test both cases, first writing a question, then reading the response, but then trying to read second response without having made another write. The result was bad…

My very first self-caused Blue Screen of Death…

The most disturbing thing about it was that PREfast had no in wa complained. I had a bad feeling about using a global variable in a driver, given all the issues with paging I had read about. After getting an OK from PREfast, however, I felt more confident about it. Obviously, too confident.

With the use of PREfast at an end. I next turned to SDV, hoping it would shed light on where I went wrong.

Verify with SDV:
With a BSOD as proof that something was quite wrong with my driver, I next ran SDV, confident that it would come back and explain the list of events leading to a possible crash, or at least point out some horrible coding rule I had violated.

One and a half hours later… no rules were applicable. Nothing my program had done matched any of the rules of SDV. Worst yet, it took an hour and a half to tell me something that grep could've done in under a second.

My original intention was to use properties of the question string to pick the response, even if it was a simple as using some value in it as a seed to a random number generator to pick an index into an array for a proper response. The problem with this was that it involved string functions, which I knew from beforehand, are often a big problem area even in user mode programs, let alone in drivers.

I first tried using strlen(), a standard C function that while having been around for decades, can cause all hell on a bad piece of data, for example if the string is not NULL-terminated.

What surprised me most was that running both PREfast and SDV on the new version of the code (which removed the newest logic which led to the BSOD) produced no warnings. I found this quite odd… impossible in fact.

One of the rules in SDV is the SafeStrings: which explicitly checks to make sure "safe" Microsoft-added versions of common string functions are usd in place of the common ones. In researching the problem, I even came upon an article [3] by a Professoinal Driver Writer on both the importance and pains of using proper strings.

I at first wondered if SDV was perhaps not finding the particular function that was including the strlen function. I therefore wrote some unsafe (using standard C functions) code in the main DriverEntry. SDV still said that the SafeStrings rule did not apply.

Even more confused, I now seriously wondered if it was even scanning my project. Relying upon tried and tested bad examples from the failed_driver examples included with SDV, I pasted the CancelSpinLock into the beginning of the DriverEntry. To my complete shock, SDV not only didn't report that the CancelSpinLock rule didn't apply, it in fact said the rule passed!

Utterly perplex at this point, I decided to paste yet another known bad code segment from the examples, this time relying on the NullExFreePool rule, which frees a NULL pointer. Not knowing what to expect, SDV finally reported that it had failed the NullExFreePool.

At this point, I decided to conclude my driver writing adventure. I was hoping that SDV would help guide me along the way, preventing future Blue screens. After see it blatantly pass a rule that was clearly failing, however, I lost faith in it. Not wanting to further damage my computer, I decided to call it quits for the time being.

Summary:

The PREfast tool, while sometimes giving false positives, all-in-all made driver development better. It helped guide me towards better programming practices, and caught a few legitimate potential bugs (at least on a 64 bit machine). My one complaint with it is that it doesn't really seem to qualify as a verification tool. In the future, I think it, or a derivative of it, may become part of the compiler itself. PREfast was actually originally developed at Microsoft for finding common C/C++ bugs, independent of the specific usage. Based on [7], it was used internally at Microsoft in several groups outside of the driver development group, including in Office, SQL, and Windows.

What makes drivers special was Microsoft's decision to release the tool for external use. In the future, however, given that Microsoft apparently uses it internally for a variety of uses, I would not at all be surprised if it found its way in the compiler itself. Most of the messages, after all, are very similar to warnings a typical compiler gives. The only difference being that they are specific to the driver platform, rather than to generic language constructs.

SDV was a completely different story. While I can vaguely see what it can be useful in finding bugs, I personally did not find it at all useful. It is either still buggy, or I am doing something very wrong.

My biggest complaint is the running time of the application. SDV actually prints to the command line each time it starts and finishes checking each rule. From this I could tell how long it was spending on each rule. To my great surprise, it wasn't focusing the hour and a half of processing on potentially relevant rules, but was spending roughly equal time on each. This seemed absurd given that a quick search fo the code would reveal that it was absolutely impossible for many of the rules to apply simply because they didn't contain certain things like spin locks or other relevant data structures or system calls.

Conclusion:

I really enjoyed doing this project. After many many hours of browsing the WDK documentation and reading a few very helpful tutorials, I feel like I finally have at least a vague grasp on the different Windows Driver models. It was also very satisfying to be able to compile and even slightly improve a driver I had used in the past.

The only very troubling part was when I bluescreened and thereby crashed my machine. It actually corrupted files on my hard drive, and made me much more hesitant to run any additional driver code. Next time I do driver development, I will either use a dedicated stripped down machine, or setup a virtual machine so that no real damage can be done.

The state of formal verification in commercial software applications is bright. In the case of Windows Drivers, I think the foundation has been laid down, but significant work needs to be done. I also fear the Microsoft may be moving slightly in the wrong direction, aiming at developing extremely complicated tools and rule-checkers to help programmers understand an extremely complex driver development model rather than focusing more effort on simplifying the driver model itself. To Microsoft's credit, they do appear to have started moving in this direction, as Windows Vista will apparently introduce "User-Mode" drivers as part of a new driver framework, the Windows Driver Framework (WDF). The one thing Microsoft absolutely must do is to come up with more distinct names for the different driver frameworks, the Window Driver Development Kit, Windows Driver Kit, and Windows Driver Framework just sound much too similar to be used to refer to completely different things!

References:

[1] Introduction to Device Drivers: http://www.catch22.net/tuts/kernel1.asp
[2] OSLoader: http://www.osronline.com
[3] Opferman, Toby. Driver Development Tutorials. The Code Project. http://www.codeproject.com/system/#Device+Drivers
[4] Little, Gary. Don't String Me Along. http://www.wd-3.com/archive/SafeStrings.htm
[5] Peacock, Greg. PortTalk – A Windows NT/2000 I/O Port Device Driver: http://www.beyondlogic.org/porttalk/porttalk.htm
[6] DbgView. http://www.microsoft.com/technet/sysinternals/utilities/debugview.mspx
[7] Horstmanshof, Stephanie. PREfast: Less Bugs, More Reliability. http://research.microsoft.com/displayArticle.aspx?id=634

Other References:

"MSDN Library." Microsoft Software Developer Network: http://msdn.microsoft.com/library/default.asp
"Quick Start: Static Driver Verifier." http://msdn.microsoft.com/library/default.asp?url=/library/en-us/DevTest_g/hh/DevTest_g/staticdv_fd29cbbc-5ddc-4ab7-8e2c-fbdcf3f499d5.xml.asp
"DDK – Windows Driver Development Kit." http://www.microsoft.com/whdc/devtools/ddk/default.mspx
"Windows Driver Kit": http://www.microsoft.com/whdc/devtools/WDK/default.mspx