Robustness Testing of the Windows DDK[*]

Manuel Mendonça
University of Lisboa, Portugal
manuelmendonca@msn.com

Nuno Neves
University of Lisboa, Portugal
nuno@di.fc.ul.pt

Abstract

Modern computers interact with many kinds of external devices, which have lead to a state where device drivers (DD) account for a substantial part of the operating system (OS) code. Currently, most of the systems crashes can be attributed to DD because of flaws contained in their implementation. In this paper, we evaluate how well Windows protects itself from erroneous input coming from faulty drivers. Three Windows versions were considered in this study, Windows XP and 2003 Server, and the future Windows release Vista. Our results demonstrate that in general these OS are reasonably vulnerable, and that a few of the injected faults cause the system to hang or crash. Moreover, all of them handle bad inputs in a roughly equivalent manner, which is worrisome because it means that no major robustness enhancements are to be expected in the DD architecture of the next Windows Vista.

1 Introduction

Personal computers are common tools on today's modern life, not only for business, but for leisure and learning. Currently they interconnect all kinds of consumer electronic devices (e.g., cameras, MP3 players, printers, cell phones). In order to support the constant innovation of these products, operating systems (OS) had also to evolve in their architectures to become, as much as possible, independent of the hardware. Their flexibility and extensibility is achieved by the virtualization offered by device drivers (DD), which basically act as the interface between the software and the hardware. Given the typical short life cycle of chipsets and motherboards, system designers have to constantly develop new DD and/or update the existing ones. For these reasons they are the most dynamic and largest part of today's OS.

Even tough current DD are mostly written in a high level language (e.g., C), they continue to be difficult to build and verify. The development of a driver requires knowledge from a set of disparate areas, including chips, OS interfaces, compilers, and timing requirements, which are often not simultaneously mastered by the programmers, leading to both design and implementation errors. Consequently, DD are becoming one of the most important causes of system failures. A recent report showed that 89% of the Windows XP crashes are due to 3rd party DD [21]. Another analysis carried out on Linux demonstrated that a significant portion of failures can be pointed to faulty drivers [4].

As a result, commercial and open source OS are both committed in efforts to deploy more robust drivers. As an example, Microsoft has several tools to assist developers that write code in kernel-mode (e.g., Driver Verifier [16]). Other projects like [3, 6, 22, 25] also propose ways to improve the error containment capabilities of the OS.

In this paper, we want to study the behavior of three Windows versions, XP, 2003 Server and the future Windows release Vista, when they receive erroneous input from a faulty driver. We want to understand for instance if this input can frequently cause the crash of the OS, and if most functions process the input in a safe way or if they are mostly unprotected. We would like also to know the impact of the file system, FAT32 or NTFS, on the observed failure modes. This type of data is important because it helps to understand the extent of the problem, and what solutions need to be devised and applied to ameliorate the robustness of current systems.

Additionally, in the past, the origin of the bad input has been mainly from accidental nature. This situation will probably change in the future, as DD turn into the targets of the malicious attacks, especially because the

[*] This work was partially supported by the EU through project IST-4-027513-STP (CRUTIAL) and NoE IST-4- 026764-NOE (RESIST), and by the FCT through project POSC/EIA/61643/2004 (AJECT) and the Large- Scale Informatic Systems Laboratory (LASIGE).

most common avenues of attack are becoming increasingly difficult to exploit. If this scenario ever occurs, one might end up in a position where many drivers have vulnerabilities, and our only defense is the OS own abilities to protect itself.

The paper uses robustness testing to measure how well these OS handle the inputs from a DD [1, 5, 12]. A group of functions from the Windows interface (for kernel-mode DD, these functions are defined in the Device Driver Toolkit (DDK)) was selected and experimentally evaluated. The tests emulated a range of programming flaws, from missing function initializations to outside range parameter values.

Our results show that in general the three OS are relatively vulnerable to erroneous input, and that only a few routines made an effective checking of the parameters. A few experiments resulted in an OS hang and several caused the system to crash. When the OS installation used the FAT32 file system, some files ended up being corrupted during the crash. This problem was not observed with the NTFS file system. The minidump diagnosis mechanism was also analyzed, and it provided valuable information in most cases. Overall, the three OS versions showed a roughly equivalent behavior.

2 DDK Test Methodology

In a robustness testing campaign one wants to understand how well a certain interface withstands erroneous input to its exported functions. Each test basically consists on calling a function with a combination of good and bad parameter values, and on observing its outcome in the system execution. As expected, these campaigns can easily become too time consuming and extremely hard to perform, specially if the interface has a large number of functions with various parameters, since this leads to an explosion on the number of tests that have to be carried out. This kind of problem occurs with the Windows DDK because it exports more than a thousand functions. However, from the group of all available functions, some of them are used more often than others, and therefore these functions potentially have more impact in the system. Moreover, in most cases, (good) parameter values are often restricted to a small subset of the supported values of a given type.

Based on these observations, we have used the approach represented in Figure 1 in the tests. The DevInspect tool performs an automatic analysis of the target system to obtain a list of available DD. Then, it measures the use of each imported function from the DDK by each driver.

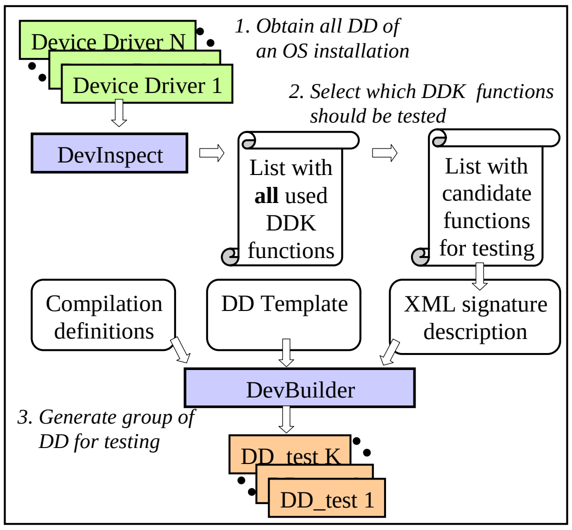
Device Driver N
Device Driver 1
1. Obtain all DD of an OS installation
2. Select which DDK functions should be tested
DevInspect
List with all used DDK functions
List with candidate functions for testing
Compilation definitions
DD Template
XML signature description
DevBuilder
3. Generate group of DD for testing
DD_test K
DD_test 1

Figure 1: Generating the test DD.

Using this data, one can select a group of functions for testing, the *candidate list*. A XML file is manually written to describe the prototype of each function, which also includes the fault load (e.g., the bad values that should be tried).

Next, the DevBuilder tool takes as input the information contained in the XML file, a template of a device driver code and some compilation definitions, and generates the workload utilized to exercise the target system and to observe its behavior. The workload includes for each function test a distinct DD that injects the faulty input.

Other approaches could have been employed to implement the tests (e.g., a single DD injects all faulty data). This solution was chosen because: First, the control logic of each driver and management tool becomes quite simple. Second, the interference between experiments basically disappears since an OS reboot is performed after a driver test. Last, one can determine if the DD loading and unloading mechanisms are damaged by the injected faults.

2.1 Selecting the Candidate Functions

Windows stores drivers in the portable executable file format [15], which contains a table with the functions that are exported and imported. In the case of drivers, the imported functions are the ones provided by the DDK. Therefore, one can discover the DD currently available in a system by looking for .sys modules placed in \system32\drivers. Then, by examining the table of imported functions of the drivers, one can collect statistics about which DDK functions are utilized in practice.

We have installed Windows XP and Server 2003 with FAT32 and NTFS file systems and Windows Vista with NTFS file system in a DELL Optiplex 170L computer. Table 1 shows the number of drivers found in each Windows installation. Each line identifies the OS name and file system, the number of drivers that were found and that were running when the boot sequence completed, and the number of functions called by these drivers. It is possible to observe, for instance, that Windows Vista calls many more functions than Server 2003 for roughly the same number of drivers (2400 instead of 1463).

From the analysis of these drivers (both total and running), it was visible that a small group of functions was called by a majority of the DD, and that most of the rest of the functions were infrequently utilized (e.g., around 900 functions were only called by 1 or 2 drivers). These results indicate that if one of the most called functions unsafely treats its parameters, then almost every DD is potentially affected.

For this work, the functions that were chosen for the candidate list were the ones utilized by the majority of the drivers. We have established the following selection criterion: *the tested functions had to be present in at least 95% of all running drivers*. Table 2 displays the first group of the most used functions that satisfied this criterion. In each line, the table presents our internal identifier, the name of the function and its alias (to reduce the size of the rest of the tables). We have found out that this list changes very little when this criteria is applied to all existing drivers and not only the running ones. Table 3 displays the driver coverage by this group of functions in each OS configuration.

We considered other criteria to select the candidate list, such as the static or dynamic frequency of function calls. Static frequency picks functions that appear many times in the code without taking into account the logic under it – a function may appear repeatedly in the code but may never be executed. Dynamic frequency chooses the functions that are called most often during the execution of a given workload. Therefore, if the workload has a high file activity then disk drivers would run more, and their functions would be selected for the candidate list. This will bias the analysis towards the elected workload, which is something we wanted to avoid in these experiments.

2.2 Tested Faulty Values

The main responsibility of the DevBuilder tool is to write a number of DD based on the template code, each one carrying out a distinct function test (see Figure 1).

Table 1: Drivers in a Windows installation.

Windows	File System	Drivers		Run Drivers Functions
		Total	Run	
XP	FAT32	259	93	1490
	NTFS	260	94	1494
Server 2003	FAT32	189	93	1463
	NTFS	189	92	1463
Vista	NTFS	250	113	2400

Table 2:Top 20 called DDK functions.

ID	Name	Alias
1	ntoskrnl::RtlInitUnicodeString	InitStr
2	ntoskrnl::ExAllocatePoolWithTag	AllocPool
3	ntoskrnl::KeBugCheckEx	BugCheck
4	ntoskrnl::IofCompleteRequest	CompReq
5	ntoskrnl::IoCreateDevice	CreateDev
6	ntoskrnl::IoDeleteDevice	DeleteDev
7	ntoskrnl::KeInitializeEvent	InitEvt
8	ntoskrnl::KeWaitForSingleObject	WaitObj
9	ntoskrnl::ZwClose	ZwClose
10	ntoskrnl::IofCallDriver	CallDrv
11	ntoskrnl::ExFreePoolWithTag	FreePool
12	ntoskrnl::KeSetEvent	SetEvt
13	ntoskrnl::KeInitializeSpinLock	InitLock
14	HAL::KfAcquireSpinLock	AcqLock
15	HAL::KfReleaseSpinLock	RelLock
16	ntoskrnl::ObfDereferenceObject	DerefObj
17	ntoskrnl::ZwOpenKey	OpenKey
18	ntoskrnl::ZwQueryValueKey	QryKey
19	IoAttachDeviceToStack	AttachDev
20	ntoskrnl::memset	memset

Table 3: Top 20 Functions Driver coverage.

Windows	File System	Driver coverage
XP	FAT32	96,7%
	NTFS	96,8%
Server 2003	FAT32	96,7%
	NTFS	96,7%
Vista	NTFS	97,3%

To accomplish this task, all relevant data about the functions is provided in a XML signature file, and the DD template has special marks that identify where to place the information translated from XML into code.

The signature file includes the function name, parameter type and values that should be tried out, and expected return values. In addition, for certain functions, it also contains some setup code that is inserted before the function call, to ensure that all necessary initializations are performed. Similarly, some other code can also be included, which is placed after the function call, for instance to evaluate if some parameter had its value correctly changed or to check the returned value.

In order to obtain the relevant data about the functions, we had to resort to the Windows DDK documentation. From the point of view of a DD developer, this documentation corresponds to the specification of the DDK functions. Therefore, if there are errors in documentation, then they may be translated into bugs in the drivers' implementations and also in our tests. Nevertheless, in the worst case, if a problem is observed with a test, at least it indicates that the function description contains some mistake.

The signature file defines seven types of correct and faulty inputs. These values emulate the outcomes of some of the most common programming bugs. They can be summarized as follows:

Acceptable Value: parameter is initialized with a correct value.

Missing local variable initialization: parameter with a random initial value.

Forbidden values: uses values that are explicitly identified in the DDK documentation as incorrect.

Out of bounds value: parameters that exceed the expected range of values.

Invalid pointer assignment: invalid memory locations.

NULL pointer assignment: NULL value passed to a pointer parameter.

Related function not called: this fault is produced by deliberately not calling a setup function, contrarily to what is defined in the DDK documentation.

2.3 Expected Failure Modes

The list displayed in Table 4 represents the possible scenarios that are expected to occur after a DD injects a fault into the OS. Initially we started with a much larger list of failure modes, which was derived from various sources, such as the available works in the literature and expert opinion from people that administer Windows systems. However, as the experiences progressed, we decided to reduce substantially this list because several of the original failure modes were not observed in practice.

Generally speaking there are two major possible outcome scenarios: either the faulty input produces an error (e.g., a crash) or it is handled in some manner. Since the fault handling mechanisms can also have implementation problems, the FM1 failure mode was divided in three subcategories. In order to determine which subcategory applies to a given experiment, the DD verifies the correctness of the return value (if it was different from void) and output parameters of the function.

Table 4: Expected failure modes.

ID	Description
FM1	No problems are detected in the system execution.
FM2	The applications or even the whole system hangs.
FM3	The system crashes and then reboots; the file system is checked and NO corrupted files are found.
FM4	Same as FM3, but there are corrupted files.

Returns ERROR (RErr): The return value from the function call indicates that an error was detected possibly due to invalid parameters. This means that the bad input was detected and was handled properly.

Returns OK (ROk): The return value of the call indicates a successful execution. This category includes two cases: even with some erroneous input, the function executed correctly or did not run but returned OK; all input was correct, for instance because only good parameter values were utilized or the random parameters ended up having acceptable values.

Invalid return value (RInv): Some times several values are used to indicate a successful execution (a calculation result) or an error (reason of failure). When the return value is outside the range of possible output values (at least from what is said in the DDK documentation), this means that either the documentation or the function implementation has a problem.

Whenever crash occurs, Windows generates a minidump file that describes the execution context of the system when the failure took place. The analysis of this file is very important because it allows developers to track the origin of crashes. Although several efforts have been made to improve the capabilities of crash origin identification, still some errors remain untraceable or are detected incorrectly. Whenever an experiment caused a crash, the minidump files were inspected to evaluate their identification capabilities. Four main categories of results were considered:

Identification OK (M1): The minidump file correctly identifies the faulty driver as the source of the crash.

Identification ERROR (M2): The minidump file identifies other module as the cause of failure.

Unidentified (M3): The minidump file could not identify either the driver or other module as the source of the crash.

Memory Corruption (M4): The minidump file detected a memory corruption.

2.4 Experimental Setup

Since the experiments were likely to cause system hangs or crashes, and sometimes these crashes corrupted files, we had to utilize two machines in order to automate most of the tasks (see Figure 2). The target machine hosts the OS under test and the DD workload, and the controller machine is in charge of selecting which tests should be carried out, collecting data and rebooting the target whenever needed.

After booting the targeting machine, DevInject contacts DevController to find out which driver should be used in the next experiment. Then, DevInject loads the driver, triggers the fault, checks the outcome and, if everything went well, removes the driver. DevController is informed of each step of the experiment, so that it can tell DevInject what actions should be performed. This way, the target file system is not used to save any intermediate results or keep track of the experience, since it might end up being corrupted. The target file system is however utilized to store the minidump files and the corrupted files that were found. After a reboot, DevInject transfers to DevController this information using FTP.

3 Experimental Results

All measurements were taken on a prototype system composed by two x86 PCs linked by an Ethernet network. The target machine was a DELL Optiplex computer with 512Mb and 2 disks. Three OS versions and two distinct file systems, FAT32 and NTFS, were evaluated. The outcome was five different configurations (Vista was not tested with FAT32). The exact OS versions were: Windows XP Kernel Version 2600 (SP 2), *built*: 2600.xpsp_sp2_gdr.050301-1519, Windows Server 2003 Kernel Version 3790 (SP 1), *built*: 3790.srv03_sp1_rtm.050324-1447 and Windows Vista Kernel Version 5600, *built*: 5600.16384.x86fre.vista_rc1.060829-2230.

Microsoft provides an equivalent Device Driver Toolkit for all OS. Consequently, the same set of drivers could be used to test the various OS. In every target configuration the initial conditions were the same, the OS were configured to produce similar types

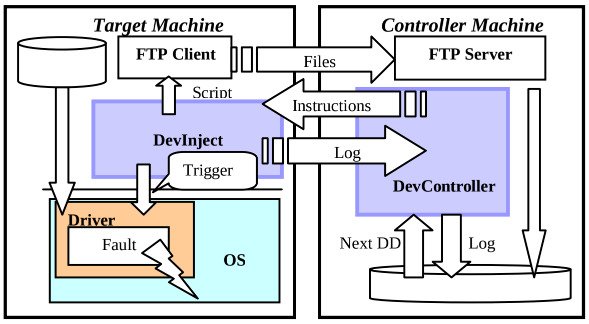

Figure 2: Injecting erroneous input.

of dump files, and the DevInject tool was basically the only user application running.

We decided to carry out the experiments without load to ensure that results were highly repeatable, and therefore to increase the accuracy to the conclusions. In the near future, we intend to complement our analysis with loaded systems, by employing some standard workload.

3.1 Observed Failure Modes

The observed failure modes are displayed in Table 5. The first three columns present the function identifier ID, its alias name and the number of experiments carried out with each function. The failure modes for the various OS configurations are represented in the next four groups of columns, under the headings FM1 to FM4. Each column group presents one value for each OS configuration.

In the 20 functions that were tested, several of them were able to deal at least with a subset of the erroneous input. There were however a few cases where results were extremely bad, indicating a high level of vulnerability. By computing the formula FM1/#DD for each FM1 entry, one can have an idea about the relative robustness of the functions (see Figure 3). Only two functions were 100% immune to the injected faults, 9-ZwClose and 18-QryKey. On the other hand, eight functions had zero or near zero capabilities to deal with the faults.

One reason for this behavior is that some of these functions are so efficiency dependent (e.g., 4-CompReq and 14-AcqLock) that developers have avoided the implementation of built in checks. Another reason is related to the nature of the function, which in the case of 3-BugCheck is to bring down the system in a controlled manner, when the caller discovers an unrecoverable inconsistency.

Table 5: Observed failure modes.

ID	Alias	#DD	FM1: Execution OK					FM2: Hangs					FM3: Crash					FM4: Crash & FCorrupt				
			XP		2003		V	XP		2003		V	XP		2003		V	XP		2003		V
			Fat	Ntfs	Fat	Ntfs	Ntfs	Fat	Ntfs	Fat	Ntfs	Ntfs	Fat	Ntfs	Fat	Ntfs	Ntfs	Fat	Ntfs	Fat	Ntfs	Ntfs
1	InitStr	12	9	9	9	9	9	0	0	0	0	0	2	3	2	3	3	1	0	1	0	0
2	AllocPool	440	416	416	416	416	420	0	0	0	0	0	14	24	13	24	20	10	0	11	0	0
3	BugCheck	12	0	0	0	0	0	0	0	0	0	0	6	12	3	12	12	6	0	9	0	0
4	CompReq	51	0	0	0	0	0	0	0	0	0	0	0	51	26	51	51	51	0	25	0	0
5	CreateDev	96	48	48	48	48	48	0	0	0	0	0	29	48	8	48	48	19	0	40	0	0
6	DeleteDev	4	0	0	0	0	0	0	0	0	0	0	0	4	2	4	4	4	0	2	0	0
7	InitEvt	18	6	6	6	6	6	0	0	0	0	0	8	12	7	12	12	4	0	5	0	0
8	WaitObj	36	18	18	18	18	18	0	0	0	0	0	15	18	2	18	18	3	0	16	0	0
9	ZwClose	3	3	3	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	CallDrv	9	0	0	0	0	0	0	0	0	0	0	0	9	4	9	9	9	0	5	0	0
11	FreePool	16	1	1	1	1	1	0	0	0	0	0	1	15	8	15	15	14	0	7	0	0
12	SetEvt	24	6	6	18	18	9	0	0	0	0	0	15	18	5	6	15	3	0	1	0	0
13	InitLock	3	2	2	2	2	2	0	0	0	0	0	0	1	0	1	1	1	0	1	0	0
14	AcqLock	8	0	0	0	0	0	2	2	2	2	2	2	6	2	6	6	4	0	4	0	0
15	RelLock	48	3	3	1	1	1	0	0	0	0	0	18	45	12	47	47	27	0	35	0	0
16	DerefObj	3	2	2	2	2	0	0	0	0	0	0	1	1	0	1	3	0	0	1	0	0
17	OpenKey	155	104	104	104	104	104	0	0	0	0	0	25	51	47	51	51	26	0	4	0	0
18	QryKey	315	315	315	315	315	315	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	AttachDev	9	0	0	0	0	0	1	1	1	1	1	6	8	2	8	8	2	0	6	0	0
20	memset	48	18	18	27	27	24	0	0	0	0	0	22	30	12	21	24	8	0	9	0	0
Total		**1310**	**951**	**951**	**970**	**970**	**960**	**3**	**3**	**3**	**3**	**3**	**164**	**356**	**155**	**337**	**347**	**192**	**0**	**182**	**0**	**0**
Total / # DD (%)			**72,6**	**72,6**	**74,0**	**74,0**	**73,3**	**0,2**	**0,2**	**0,2**	**0,2**	**0,2**	**12,5**	**27,2**	**11,8**	**25,7**	**26,5**	**14,7**	**0,0**	**13,9**	**0,0**	**0,0**

In this case, the developers probably preferred to reboot the system even if some parameters were incorrect (but notice that this reboot sometimes was not done in a completely satisfactory way since files ended up being corrupted).

From the various functions, only two caused the system to hang (FM2 ≠ 0). Functions 14-AcqLock and 19-AttachDev caused hangs in all OS configurations, when an invalid pointer was passed as argument. Most of the erroneous inputs that caused failures end up crashing the system (FM3 and FM4). From the various classes of faults that were injected, the most malicious were invalid pointer assignments and NULL values passed in pointer parameters. The first class, invalid pointers, is sometimes difficult to validate, depending on the context (e.g., a buffer pointer that was not properly allocated). On other hand, NULL pointers are easily tested and for this reason it is difficult to justify why they are left un-checked, allowing them to cause so many reliability problems.

In all experiments, we never observed any file corruption with the NTFS file system after a reboot. However, the FAT32 file system displayed in many instances cases of corruption. Traditionally, NTFS has been considered much more reliable than FAT32, and our results contribute to confirm this. The reliability capabilities integrated in NTFS, like transactional operations and logging, have proven to be quite effective in protecting the system during abnormal execution.

The overall comparison of the 3 operating systems, if we restrict ourselves to NTFS or FAT32, shows a remarkable resemblance among them. The last two rows of Table 5 present an average value for the failure modes and OS configurations. On average, OSs had an approximately equivalent number of failures in each mode, with around 73% testes with no problems detected during the system execution. Hangs were a rare event in all OSs. If a finer analysis is made on a function basis (see Figure 3), we observe a similar behavior for most functions. There were only two functions where results reasonably differ, 12-SetEvt and 20-memset. From these results, there is reasonable indication that the 3 operating systems use comparable levels of protection from faulty inputs coming from drives.

These results reinforce the idea that although the Windows NT system has undergone several name changes over the past several years, it remains entirely based on the original Windows NT code base. However, as time went by, the implementation of many internal features has changed._We expected that newer versions of the Windows OS family would become more robust; in practice we did not see this improvement at the driver's interface. Of course, this conclusion needs to be better verified with further experiments.

3.2 Return Values from Functions

As explained previously, even when the system executes without apparent problems, the checking mechanisms might not validate the faulty arguments in the most correct manner and produce fail-silent violations. Therefore, failure mode 1 can be further divided in three sub-categories to determine how well the OS handled the inputs.

Table *6* shows the analysis when the function execution returned a value in the RErr category, i.e., an error was detected by the function. Since some functions do not return any values, their corresponding table entries were filled with "-". The "# Faulty Drivers" column refers to the number of drivers produced by DevBuilder that contained at least one bad parameter. Comparing this column with the following five columns, one can realize that only two functions have a match between the number of faulty drivers and the number of RErr values. The other functions revealed a limited parameter checking capability.

To complement this analysis, Table 7 presents the results for the ROk category (i.e., the return value of the call is a successful execution). Column "Non Faulty Drivers" shows the number of drivers with only correct arguments. Comparing this column with the remaining ones, it is possible to conclude that functions return a successful execution more often then the number of non faulty drivers. However, in some cases this might not mean that there is a major problem. For instance, consider function 2-AllocPool that receives three parameters: the type of pool (P0); the pool size (P1); and a tag value (P2). Depending on the order of parameter checking, one can have the following acceptable outcome: P1 is zero, and 2-AllocPool returns a pointer to an empty buffer independently of the other parameters values.

On the other hand, by analyzing the execution log, we found out that when P1 was less than 100.000*PAGE_SIZE, Windows returned ROk even when a forbidden value was given in P0 (at least, as stated in the DDK documentation). This kind of behavior means that an error was (potentially) propagated back to the driver, since it will be using a type of memory pool different from the expected thus causing a fail silent violation. The table also reveals another phenomenon -- the three versions of Windows handle the faulty parameters differently.

Table 6: Return error (RErr) values.

Alias	**# Faulty Drivers**	**Rerr**				
		XP		**2003**		**V**
		Fat	**Ntfs**	**Fat**	**Ntfs**	**Ntfs**
InitStr	9	0	0	0	0	0
AllocPool	200	20	20	20	20	12
BugCheck	12	-	-	-	-	-
CompReq	51	-	-	-	-	-
CreateDev	76	0	0	0	0	0
DeleteDev	4	-	-	-	-	-
InitEvt	14	-	-	-	-	-
WaitObj	36	0	0	0	0	0
ZwClose	3	3	3	3	3	3
CallDrv	9	0	0	0	0	0
FreePool	15	-	-	-	-	-
SetEvt	20	0	0	0	0	0
InitLock	2	-	-	-	-	-
AcqLock	8	0	0	0	0	0
RelLock	48	-	-	-	-	-
DerefObj	3	-	-	-	-	-
OpenKey	155	104	104	104	104	104
QryKey	315	315	315	315	315	315
AttachDev	9	0	0	0	0	0
Memset	39	0	0	0	0	0

Table 7: Return OK (ROk) values.

Alias	**Non Faulty Drivers**	**ROk**				
		XP		**2003**		**V**
		Fat	**Ntfs**	**Fat**	**Ntfs**	**Ntfs**
InitStr	3	9	9	9	9	9
AllocPool	240	396	396	396	396	408
BugCheck	0	-	-	-	-	-
CompReq	0	-	-	-	-	-
CreateDev	20	48	48	48	48	48
DeleteDev	0	-	-	-	-	-
InitEvt	4	-	-	-	-	-
WaitObj	0	18	18	18	18	18
ZwClose	0	0	0	0	0	0
CallDrv	0	0	0	0	0	0
FreePool	1	-	-	-	-	-
SetEvt	4	6	6	18	18	9
InitLock	1	-	-	-	-	-
AcqLock	0	0	0	0	0	0
RelLock	0	-	-	-	-	-
DerefObj	0	-	-	-	-	-
OpenKey	0	0	0	0	0	0
QryKey	0	0	0	0	0	0
AttachDev	0	1	1	1	1	1
memset	9	18	18	27	27	22

For example, there were several cases in Vista where function 2-AllocPool succeeded while in XP and Server 2003 it caused a crash. In function 12-SetEvt, Server 2003 does not crash when TRUE was passed in one of the parameters, while the other did so (the documentation says that when this value is used, the function execution is to be followed immediately by a call to one of the KeWaitXxx routines, which was not done in either OSs).

In all experiments, we did not observe any return values belonging to the RInv category (i.e., values outside the expected return range).

3.3 Corrupted Files

The last group of results in Table 5 corresponding to FM4, displays the number of times Windows found corrupted files while booting. The Chkdsk utility is called during the booting process to detect these files. Corrupted files were found only in the configurations that used the FAT32 file system. Using the formula FM4/FM3 one can have a relative measure of how sensitive is the file system when a crash occurs. The results presented in Figure 4 shows that when using FAT32 in general, Windows Server 2003 is more sensitive than Windows XP in a majority of the cases.

3.4 Minidump Diagnosis Capabilities

The analysis of the minidump files produced during a system crash allows us to determine how well they identify a driver as the culprit of the failure. These files are fundamental tools for the Windows development teams because they help to diagnose system problems, and eventually to correct them. We have used the Microsoft’s Kernel Debugger (KD) [17] to perform the analysis of these files, together with a tool, DevDump, that automates most of this task. DevDump controls the debugger, passes the minidumps under investigation, and selects a log where results should be stored. After processing all files, DevDump generates various statistics about the detection capabilities of minidumps.

In the experiments, all Windows versions correctly spotted the faulty DD in the majority of times (see Figure 5 and compare it with Table 5). The correct identification of the source of crash (M1) seams to be independent of the file system used. Only in very few cases there was a difference between the two file systems, such as for the 7-InitEvt function where Server 2003 FAT32 identified a different source of crash from Server 2003 NTFS.

In general, the results show that Windows XP is more accurate than the others OS (see 15-RelLock and 20-memset). Still there were cases where other kernel modules were incorrectly identified (functions 1-InitStr, 14-AcqLock and 15-RelLock), as displayed in Figure 6.

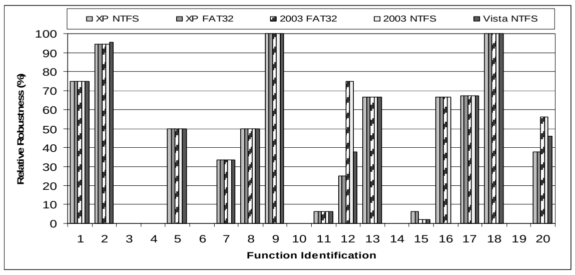

Figure 3: Relative robustness (FM1/#DD).

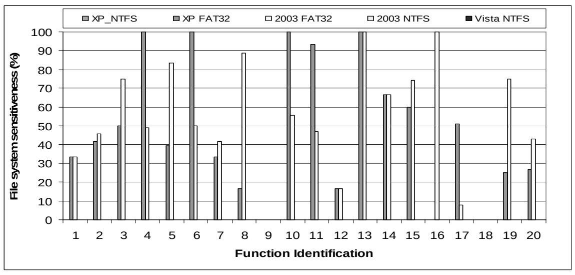

Figure 4: FSystem sensitiveness (FM4/FM3).

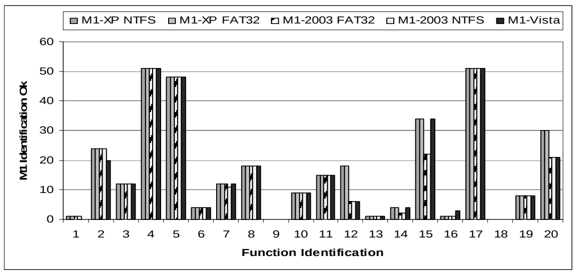

Figure 5: Source identification OK (M1).

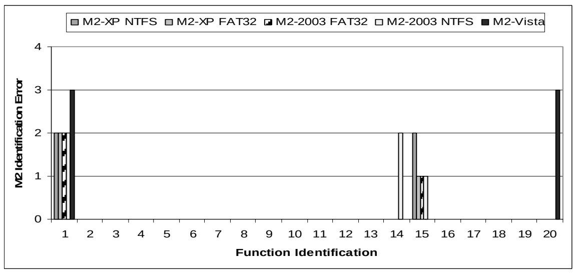

Figure 6: Source identification error (M2).

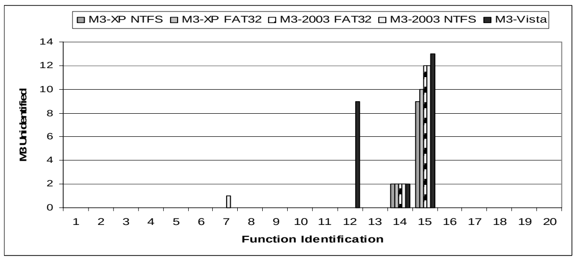

Figure 7: Source of crash unidentified (M3).

These errors are particularly unpleasant because they can lead to waist of time while looking for bugs in the wrong place, and they can reduce the confidence on the information provided by minidumps. In some other cases, Windows was unable to discover the cause of failure. This happened in Vista more frequently than the other OS configurations, for instance in functions 15-RelLock and 12-SetEvt (see Figure 7). In the last function, Vista was the only system that could not diagnose the cause of failure. Only Windows Server 2003 detected memory corruption situations (in functions 14-AcqLock and 15-RelLock). Windows Server 2003 (FAT32 and NTFS) located memory corruptions when faults were injected in functions 14-AcqLock and 15-RelLock.

4 Related Work

Robustness testing has been successfully applied to several software components to characterize their behavior when facing exceptional inputs or stressful environmental conditions. One of the main targets of these studies has been general propose OS, with erroneous inputs being injected at the application interface. Fifteen OS versions that implement the POSIX standard, including AIX, Linux, SunOS and HPUX, were assessed using the Ballista tool [12]. Shelton et al. made a comparative study of six variants of Windows, from 95 to 2000, by injecting faults at the Win 32 interface [20]. Several command line utilities of Windows NT were evaluated by Ghosh et al. [7]. Real time microkernels, such as Chorus and LynxOS, have also been the target of these studies using the MAFALDA tool [2]. Application level software can be tested using robustness techniques by, for instance, generating exceptions and returning bad values at the OS interface [8]. Middleware support systems like CORBA have been examined at the client-side interface of an ORB [19] and internally at the level of the Naming and Event services [14]. Dependability benchmarking has resorted to robustness testing in order to evaluate systems [18, 23, 24]. For example, Kalakech et al. proposed an OS benchmark which provided a comprehensive set of measures, and applied it to the Windows 2000 [11].

To our knowledge, only a few works have assessed the robustness of systems at the level of device drivers. Durães and Madeira described a way to emulate software faults by mutating the binary code of device drivers [5]. Basically, the driver executable is scanned for specific low-level instruction patterns, and mutations are performed on those patterns to emulate high-level faults. These ideas were experimented for 4 types of patterns, on 2 drivers of the Windows NT4, 2000 and XP. Albinet et al. conducted a set of experiments to evaluate the robustness of Linux systems in face of faulty drivers [1]. They intercepted the driver calls to the kernel's DPI (Drivers Programming Interface) functions, and changed the parameters on the fly with a few pre-set number faulty values. Then, the behavior of the system was observed. Johansson and Suri employed a similar methodology to evaluate a Windows CE .Net [10]. In their work, however, they focus on error propagation profiling measures, as facilitator for the selection of places to put wrappers.

Our research is complementary to these previous works, not only because we targeted different OS. The followed methodology has its roots in the original Ballista tool [13], where several test drivers are generated, containing DDK function calls with erroneous arguments. The argument values were selected specifically for each function, and they emulate seven classes of typical programming errors. Our study has looked in a comparative basis at such aspects like error containment, influence of the file system type, and the diagnosis capabilities of minidump files.

5 Conclusions

The paper describes a robustness testing experiment that evaluates Windows XP, Windows Server 2003 and the future Windows release Vista. The main objective of this study was to determine how well Windows protects itself from faulty drivers that provide erroneous input to the DDK routines. Seven classes of typical programming bugs were simulated.

The analysis of the results shows that most interface functions are unable to completely check their inputs – from the 20 selected functions, only 2 were 100% effective in their defense. We observed a small number of hangs and a reasonable number of crashes. The main reason for the crashes was invalid or NULL pointer values. Corruption of files was only observed with the FAT32 file system. The analysis of the return values demonstrates that in some cases Windows completes without generating an error for function calls with incorrect parameters; in particular, Windows Server 2003 seams to be the most permissible one. This behavior suggests a deficient error containment capability of the OS. In most cases, the examined minidump files provided valuable information about the sources of the crashes, something extremely useful for the development teams. However, Windows Vista seems to have more troubles in this identification than the other OS.

The experiments made with Windows Vista reveled that it behaves in a similar way to Windows XP and Server 2003. This probably means that Microsoft intents to continue to use the current DD architecture in its future OS, which is reasonably worrisome since Vista will be most likely the most used OS in the years to come.

6 References

[1] A. Albinet, J. Arlat, and J.-C. Fabre, "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel", *Proceedings of the International Conference on Dependable Systems and Networks*, June 2004

[2] J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, "Dependability of COTS Microkernel-Based Systems", *IEEE Transactions on Computers*, vol. 51, no. 2, 2002, pp. 138-163.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "On u-kernel construction", *Proceedings of the Symposium on Operating Systems Principles*, December 1995, pp. 237–250.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating system errors", *Proceedings of the Symposium on Operating Systems Principles*, October 2001, pp. 73–88.

[5] J. Durães and H. Madeira, "Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation", *Proceedings of the Pacific Rim International Symposium. On Dependable Computing*, December 2002, pp. 201-209.

[6] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The Flux OSKit: a substrate for OS language and resource management", *Proceedings of the Symposium on Operating Systems Principles*, October 1997, pp. 38–51.

[7] A. Ghosh, M. Schmid, and V. Shah, "Testing the robustness of Windows NT software", *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, November 1998, pp. 231-235.

[8] A. K. Ghosh, M. Schmid, "An Approach to Testing COTS Software for Robustness to Operating System Exceptions and Errors", *Proceedings 10th International Symposium on Software Reliability Engineering*, November 1999, pp. 166-174.

[9] R. Gruber, and M. L. Jiang, "Robustness Testing and Hardening of CORBA ORB Implementations", *Proceedings of the International Conference on Dependable Systems and Networks*, June 2001, pp. 141-150.

[10] A. Johansson, and N. Suri, "Error Propagation Profiling of Operating Systems", *Proceedings of the International Conference on Dependable Systems and Networks*, June 2005.

[11] A. Kalakech, T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun, "Benchmarking Operating System dependability: Windows 2000 as a Case Study", *Proceedings Pacific Rim International Symposium on Dependable Computing*, March 2004, pp. 261- 270.

[12] P. Koopman, J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems", *IEEE Transactions on Software Engineering*, vol. 26, no. 9, September 2000, pp. 837-848.

[13] N. Kropp, P. Koopman, and D. Siewiorek, "Automated Robustness Testing of Off-the-Shelf Software Components", *Proceedings of the International Symposium on Fault-Tolerant Computing*, June 1998.

[14] E. Marsden, J.-C. Fabre and J. Arlat, "Dependability of CORBA Systems: Service Characterization by Fault Injection", *Proceedings of the 21st International Symposium on Reliable Distributed Systems*, June 2002, pp. 276-285.

[15] Microsoft Corporation, "Microsoft Portable Executable and Common Object File Format Specification", February 2005.

[16] Microsoft Corporation, "Introducing Static Driver Verifier", May 2006.

[17] Microsoft Corporation, "Debugging Tools for Windows – Overview", December 2006 http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[18] A. Mukherjee and D. P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking", *IEEE Transactions of Software Engineering*, vol. 23, no. 6, 1997, pp. 366-378.

[19] J. Pan, P. J. Koopman, D. P. Siewiorek, Y. Huang, R. Gruber and M. L. Jiang, "Robustness Testing and Hardening of CORBA ORB Implementations", *Proceedings of the Internatinal Conference on Dependable Systems and Networks*, June 2001, pp. 141-150.

[20] C. Shelton, P. Koopman and K. D. Vale, "Robustness Testing of the Microsoft Win32 API", *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000, pp. 261-270.

[21] D. Simpson, "Windows XP Embedded with Service Pack 1 Reliability", *Tech. rep., Microsoft Corporation*, January 2003.

[22] M. Swift, B. Bershad, and H. Levy, "Improving the reliability of commodity operating systems", *Proceedings of the Symposium on Operating Systems Principles*, October 2003, pp. 207–222.

[23] T. K. Tsai, R. K. Iyer, and D. Jewitt, "An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems", *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, June 1996, pp. 314-323.

[24] M. Vieira and H. Madeira, "A Dependability Benchmark for OLTP Application Environments", *Proceedings of the 29th International Conference on Very Large Data Bases*, 2003, pp. 742-753.

[25] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian, "Mach: A new kernel foundation for UNIX development", *Proceedings of the Summer USENIX Conference*, June 1986, pp. 93–113.