Securing The Kernel Via Static Binary Rewriting, Program Shepherding and Partial Control Flow Integrity

Piotr Bania
www.piotrbania.com

2011

Project status:	strong working proof of concept completed
Supported archs:	X86
Supported OS:	Microsoft Windows XP, Vista, 7

Abstract

Recent Microsoft security bulletins show that kernel vulnerabilities are becoming more and more important security threats. Despite the pretty extensive security mitigations many of the kernel vulnerabilities are still exploitable. Successful kernel exploitation typically grants the attacker maximum privilege level and results in total machine compromise.

To protect against kernel exploitation, we have developed a tool which statically rewrites the Microsoft Windows kernel as well as other kernel level modules. Such rewritten binary files allow us to monitor control flow transfers during operating system execution. At this point we are able to detect whether selected control transfer flow is valid or should be considered as an attack attempt.

Our solution is especially directed towards preventing remote kernel exploitation attempts. Additionally, many of the local privilege escalation attacks are also blocked (also due to additional mitigation techniques we have implemented). Our tool was tested with Microsoft Windows XP, Windows Vista and Windows 7 (under both virtual and physical machines) on IA-32 compatible processors. Our apparatus is also completely standalone and does not require any third party software.

1 Introduction

Our tool uses the program shepherding technique [10] (monitoring control flow transfers). In our approach, however, we are not using a dynamic binary instrumentation engine, but instead all the selected binary files (Windows kernel and modules) are statically rewritten[1]. By monitoring the control flow transfers we can ignore the complexities of various vulnerabilities and focus on preventing the execution of malicious code. We are also using partial Control Flow Integration [1] technique for monitoring function returns. Section 2 describes the design of our tool.

2 Design

Our apparatus consists of four main modules:

Integration[2] Responsible for disassembling, analyzing and rewriting binary data[3];

Monitoring Responsible for filtering and monitoring the control flow transfers;

Configuration Responsible for configuring, loading and unloading the monitoring module;

Installer Responsible for replacing the rewritten kernel and necessary drivers.

The following subsections describe each module more thoroughly.

[1] Using dynamic binary instrumentation engines on kernel level code is surely harder to implement and more dangerous to use.

[3] In this paper integration process is limited to kernel and kernel mode components.

2.1 Integration module

Integration module is the most important and complex part of this project. It can be divided in two separate submodules: analyzer module and rewriting module.

2.1.1 Analyzer module

The analyzer module is responsible for disassembling and analyzing binary programs. In our case the binary programs are written in the Portable Executable (PE) file format. The analyzer engine is a slightly modified version of the one used in our previous automated binary differential analysis project called AutoDiff [14]. This component is completely standalone and does not rely on other disassembly engines, like the widely used commercial product IDA Pro [9].

The analyzer provides structured information for the integration module. This includes the information about the instructions, basic blocks, functions and other important data. The analyzer engine also divides the analyzed code into two major categories:

Solid code Obtained in a result of recursive traversal disassembly and other heuristic techniques based on the Portable Executable file format characteristics;

Prospect code Obtained in any other fashion, e.g., by using relocation information or by testing solid instruction operands.

Since our engine is using recursive traversal disassembly, dividing the analyzed code into those two categories helps us to avoid further code vs data misunderstandings. Additional heuristics mechanisms are also used, since the complexity of the executable binary files is often high. We also try to improve the code coverage by using the Microsoft symbol files. The main rule we have been using at this point is that it is better to confuse code as data than vice versa. This will be further explained in the next subsection.

As a result we achieve very good code coverage together with excellent overall performance (see Section 3.2.1).

2.1.2 Rewriting module

With the results gathered during the previous step, the rewriting module can perform the static code rewriting process. Basing on our previous experiences with static code rewriting like in project Aslan [2] or SpiderPig [3], we have decided to use a more secure (stable) approach. In Aslan [2] the approach was to interfere with all the original instructions and data. This of course often required manual interaction because of the code vs data dilemma, which cannot be totally resolved by the static analysis. Such an approach is not really usable, as it cannot be fully automated. Thus, we have decided to modify and use SpiderPig [3] instead (it is easier and more secure to perform).

Code Integration Method Our tool can rewrite the binary files in two general ways. The first way is an non-invasive one, where the rewritten code is placed in a separate file. In other words, the original files are not modified. This approach requires the additional driver module to load the integrated code into the operating system and patch all the necessary original modules in virtual memory. This approach is considered to be more secure (in terms of stability) since the original files are not changed. The second way (which is currently implemented and used in our tool) is more invasive since it modifies (rebuilds) the original Portable Executable files. In other words, the operating system boots up with an already modified kernel and selected kernel modules. We will describe our code integration algorithm in reference to the currently used method (invasive).

One of our initial assumptions in the process of static binary rewriting was to preserve the original file structure in such a way that the original code and data offsets are not changed. This step is essential for increasing the stability of the rewritten code and avoiding other problems like the already mentioned code vs data dilemma. The newly generated code which includes all the original functions is attached to the end of the original file. Typically the new code resides in the relocation section of the original Portable Executable file. The destination section is expanded and modified in order to handle executable and non-pageable code. Code is rewritten such that all the original functions are instrumented (this will be described in the next para-

graph) and the overall functionality is preserved. Rewritten functions also do not contain shared basic blocks[4]. The binary rewriting process can be divided into three phases:

Instrumentation Responsible for adding instrumentation code and expanding original control transfer instructions;

Calculation Responsible for allocating new relative virtual addresses for the rewritten basic blocks and for generating new relocation entries;

Repairing Responsible for repairing relative offsets of control transfer instructions.

In the *Instrumentation phase* we have two primary objectives. First one is to expand all the short conditional and unconditional jumps in order to avoid further problems with fixing the relative offsets in the repairing phase. Second one is strict requirement of the program shepherding method. In this case we add instrumentation for every instruction that is either a indirect `CALL` or indirect `JMP` instruction. However For the sake of Control Flow Integrity we are instrumenting suitable relative `CALL` and return (`RET`) instructions. Instrumentation is performed in such a way that the added chunk of code is executed before the original instruction. This gives us the possibility to deny the control transfers into the malicious memory region. It is worth noticing that we are not instrumenting `CALL` or `JMP` indirect instructions which refer to imported API functions. Firstly because the import address table is typically read-only (presents low security threat), and secondly due to performance reasons.

The *Calculation phase* is responsible for the calculation of new relative virtual addresses of the newly generated basic blocks. Additionally it is also needed for creating new relocation entries for the generated code. If an original instruction consists of an operand (either immediate or memory immediate) that has a relocation entry, the rewritten instruction requires the corresponding relocation entry as well.

The *Repairing phase* is necessary for fixing the offsets of the control transfer instructions that use relative operands. This is essential for keeping the rewritten correct stable and to ensure that the execution flow will remain in the rewritten code. At this point we don't need to repair other control transfer instructions since we assume the control will be given back due to the emitted function hooks (see next paragraph).

When all the phases are finished the engine is almost ready to produce a new Portable Executable file that contains the rewritten code. However, there is one essential step yet to perform. In order to transfer the execution from original code to the rewritten code, the engine must patch original functions and redirect them to the corresponding generated ones. This is achieved by emitting the `JMP` relative instruction at the original function prologue. This is not so easy to perform since the first basic block of the original function may be smaller than the 5 bytes required for the patch. Additionally, we may confuse code with data and patch some essential kernel structure which in the end will lead to a system crash.

In order to do this safely, our engine decides whether the original function can be patched or not. The decision is based on a few tests. First of all we check the size of first basic block. If it is not large enough original function remains unpatched. For the functions marked as **prospect code** we apply some additional tests. For example, we don't patch functions that consist only of one basic block. We also test if the selected function is entirely built from `ASCII` or `Unicode` characters. As it was mentioned earlier we prefer to redirect less functions and cause no stability issues than vice versa. It is also worth mentioning that the Windows kernel uses self-modifying code at certain places, forcing us to use additional checking mechanisms to address such problems. Even though we don't redirect all the original functions, our tests showed that it is still more than enough for our solution to work successfully. After the original functions are patched, the engine checks the relocation entries one more time in order to make sure none of them overlaps with the 5 byte relative `JMP` instruction. If such relocation entry is found it is simply erased. Lastly the Portable Executable headers are fixed. When all the issues are resolved the new PE file is emitted.

It is important to note that the static binary rewriting process may be performed on a different (remote) machine.

[4]This is done especially for some protection methods we would like to implement in the future [13, 15].

2.1.3 Control Flow Integrity

Control Flow Integrity rules [1] dictate that software execution must follow a path of Control Flow Graph (CFG) which in our case is obtained by the analyzer module by using static binary analysis. Whenever the control flow deviates from previously calculated path (CFG) the attack is detected. However due to the complexity of executable code and static binary analysis problems (explained before) it is hard to calculate the perfect Control Flow Graph which is crucial for our mechanism. In order to resolve this issue we have developed a special policy which guarantees us the correct Control Flow Graph. We have named is as partial Control Flow Integration policy since first of all we focus only on return instructions (since the indirect calls and jumps are handled differently) and secondly we don't protect functions that are not suitable according to our policy. Additionally in our case we are especially basing on the Call Graph that represent relationships between subroutines (functions). **Our policy states that functions that are referenced by a relocation entry or are exported are not suitable for Control Flow Integration since it is impossible to determine the full Control Flow Graph by using static binary analysis**[5]. This prevents us from protecting functions that are for example executed by 3rd party modules. Every return instruction of a CFI suitable function is then instrumented in a following fashion:

```
PUSH    EAX
MOV     EAX,DWORD PTR SS:[ESP+4]
CMP     DWORD PTR DS:[EAX+3], FUNCTION_KEY
JNZ     SHORT wrong_key
POP     EAX
cfi_ok:
RETN
wrong_key:
CMP     EAX, ORG_CALL1_VA+5
JE      cfi_ok
CMP     EAX, ORG_CALLX_VA+5
JE      cfi_ok
...
corrupted_cfi:
POP     EAX
CALL    report_corrupted_cfi
INT3
```

Listing 1: Function return with CFI instrumentation.

[5]We also require the selected function to not have any shared basic block among other functions

Additionally each relative call to a protected function is instrumented with a special x86 long `NOP` (`0x0F 0x1F`) instruction which allows us to place a 32 bit function key (`FUNCTION_KEY`) as a `NOP` operand. Code emitted at `wrong_key` label is used as a precaution and typically it shouldn't be executed at all. If a CFI protect procedure is called from a location that was not included in the original Call Graph our mechanism will detect an attack attempt.

2.1.4 Original Code Erasion

Our prototype is able to erase original code of protected driver in order to protect from "code reuse" technique. In this case most of the original module code will be filled with `0xCC` bytes. Please read `INSTALL.TXT` for more details.

2.1.5 Random code insertion

In order to change internal offsets of instructions our prototype is able to inject any code at any place in the generated code. The idea here is to keep the produced code different from one generation to another generation. Currently to demonstrate this technique we are injecting a `NOP` instruction randomly on every beginning of the basic block. Please read `INSTALL.TXT` for more details.

2.2 Monitoring module

Monitoring module is developed as a device driver. It acts as a server for the configuration module. It is also responsible for filtering and monitoring the control transfers caused by instrumented instructions. The filtering method is described in Section 2.2.2. Monitoring module contains a memory map structure that includes the information about the currently loaded kernel modules. This memory map is updated after selected device driver is loaded to or unloaded from the kernel memory. It is worth noting that instrumented instructions in the rewritten binary files do not execute the filtering procedure before the monitoring module is not fully initialized. Monitoring module is also responsible for blocking most of the local privilege escalation exploits by utilizing a very simple but yet effective technique (described in the Section 2.2.1).

2.2.1 Mitigation technique for local privilege escalation attacks

Most of the local privilege escalation exploits use the `NtQuerySystemInformation`[6] function to gather the base addresses where the kernel modules are mapped to. On Microsoft Windows systems, device drivers are mapped to different memory addresses each run. Thus, getting their base addresses is very often essential for such exploits to work correctly. Our solution hooks the `NtQuerySystemInformation` function and denies all user-mode requests where the `SystemModuleInformation` class is passed as parameter. According to our tests this method does not influence the operating system stability[7]. At this point the only way for the attacker to succeed is to find the base address using some other method (for example by using another vulnerability).

2.2.2 Detecting exploitation attempts

In order to detect attack attempts, the filtering (monitoring) procedure needs to decide whether selected control transfer is valid or not. Filtering procedure must be fast enough to not cause any major slowdown of the operating system. Our tool uses the memory map structure (mentioned in the previous section) which contains information about the currently loaded device drivers. This information is divided into fast memory page lookup entries which provide the characteristics of the selected page. For example it shows whether the page is writable, executable or whether it is a part of the kernel or any other loaded module. At this point our security policy is very simple. We mark all the control transfers to pages that are not executable and are not the part of kernel or other modules as attack attempts. Since we don't store the information about the userland modules, all control transfers from kernel to usermode space are automatically marked as forbidden.

2.2.3 Reaction to attack

When an attack is detected there are two available reaction options in the monitoring module. First one is to log the attack in a specified file and continue the execution; this is especially useful for honeypot-like systems. The second choice provides the attack logging feature together with immediate system shutdown. It is important to notice that we are operating in the kernel mode. Therefore there is no single task we can securely terminate in comparison to user mode solutions where such action can be typically safely performed.

[6] `EnumDeviceDrivers` is also used however this API function is just a wrapper for `NtQuerySystemInformation`.

[7] Our tests were performed on a default instalations of Microsoft Windows systems.

2.3 Configuration module

The Configuration module loads the monitoring module driver and creates initial memory map for all of the currently loaded kernel modules. It also provides additional information required by the monitoring module. Our policy allows only one configuration attempt after system start. This is done in order to block other potentially malicious configuration requests from the attacker.

2.4 Installer module

This module currently consists of batch scripts and programs that allow one to modify the Microsoft kernel and selected device drivers. On Windows XP we are using a `WINLOGON.EXE` thread injection method to disable the Windows File Protection. This is achieved by executing the `SfcTerminateWatcherThread` API from `SFC_OS.DLL` library. On Microsoft Windows Vista and Windows 7 we need to take file ownership and grant full access control permissions to ourselves. Additionally `WINLOAD.EXE` is copied and patched in order to allow the execution of modified Windows kernel. We are currently working on fully automating described tasks (together with easy file recovery) for all Windows operating systems.

3 Experimental results

This section is divided into two subsections. First one describes the results we have obtained while testing versus selected Windows exploits. Second one shows the performance impact.

3.1 Effectiveness

We have tested our solution with a few publicly available exploits. Due to small number of publicly

released exploits (especially remote ones) targeting kernel and kernel modules, our tests are currently limited. Obtained results are presented below:

- **CVE-2009-3103** (Microsoft Windows SMB2 '`Smb2ValidateProviderCallback`' Remote Code Execution Vulnerability) [4, 16]

  This vulnerability allows remote attackers to execute arbitrary code with system privileges. Reliable and publicly known exploit for this issue (see [16] for technical details on the exploitation process) firstly generates a so called trampoline which is located at fixed BIOS/HAL memory region (which is by default readable, writeable, and executable). After the trampoline is ready the execution is thrown to it using a `CALL EAX` instruction (located in `srv2.sys` module). Our solution is able to detect and block this attack before the generated trampoline is executed.

- **CVE-2010-2743** (Microsoft Windows `win32k.sys` Keyboard Layout Vulnerability) [5, 20, 17]

  This is one of the local vulnerabilities exploited by the Stuxnet worm [21] in order to elevate privileges. Keyboard layout vulnerability is caused by `win32k!xxxKENLSProcs` function that do not properly perform indexing of a function-pointer table during the loading of keyboard layouts from disk. Malicious code in this case is executed by the `CALL _aNLSVKFProc[ecx*4]` instruction. Due to forged index value the code flow is redirected to `0x60636261`. Contents of the memory located at this specific address can be controlled by the attacker. Tests showed that our tool detects and prevents successful exploitation of this vulnerability.

Most of the others local privilege escalation exploits are unable to work at the very first stage. This is caused by the mitigation that was described in the section 2.2.1.

As a "side effect" our apparatus is able to detect some cases of hidden malicious code that operates at the kernel level (like rootkits or bootkits).

3.2 Performance

Following section presents the performance results of our tool. It is divided into two subsections. Where first one describes the performance of the integration module and the second one focuses on testing the protected operated system itself.

3.2.1 Integration performance

Table 1 presents the results obtained by integrating original Microsoft Windows 7 files (listed below). This test was performed on T3400 2.16Ghz (Core2) notebook machine with 2.46GB of RAM. Modules (files) presented in the table were chosen specifically due to potential security threats they create. This does not mean our tool is not able to protect other modules.

The legend for Table 1 is as follows:

- Size_{org} - original file size,
- Size_{int} - file size after integration,
- T_{disasm} - time required for disassembling the selected file[8],
- $\text{T}_{basicblock}$ - time required for creating basic blocks from the disassembly information
- T_{int} - time required for instrumenting, repairing and generating new code[9].

Presented results indicate that the integration process is more than satisfactory in terms of speed and memory usage. Results also show that typically newly generated files are twice as large as the original ones. This is natural considering the modifications we have applied. Also we don't consider this as a drawback at all since nowadays few MBs of additional memory is not really considered expensive. As it was mentioned earlier (see Section 2.1.2), our integration engine may also work on external machines. This gives user the opportunity to perform the binary rewriting process remotely.

[8] Does not include time required for downloading symbol file etc.

[9] Does not include time required for emitting the PE file.

Table 1: Static binary rewriting performance depending on a various files.

File	$Size_{org}$ [MB]	$Size_{int}$ [MB]	T_{disasm} [sec]	$T_{basicblock}$ [sec]	Instructions [#]	Basic blocks [#]	Memory usage [MB]	T_{int} [sec]
afd.sys	0.13208	0.30835	0.050622	0.049928	36351	9742	10.324219	0.059221
http.sys	0.253418	0.559937	0.116081	0.123439	65868	16211	16.167969	0.104105
ipsec.sys	0.071777	0.172485	0.022959	0.018481	21893	5234	11.074219	0.022762
mrxsmb.sys	0.436646	0.998657	0.165068	0.157653	111247	27215	30.175781	0.171291
ndis.sys	0.174194	0.380493	0.078988	0.070438	45990	11649	16.070313	0.071033
ndistapi.sys	0.01001	0.021484	0.002144	0.001546	1971	476	10.011719	0.002589
ndproxy.sys	0.039063	0.088379	0.01116	0.007857	10048	2301	10.585938	0.011456
netbios.sys	0.033081	0.073608	0.014428	0.01353	8811	2451	10.257813	0.018943
netbt.sys	0.155273	0.365356	0.073585	0.072251	44696	11239	16.382813	0.092584
ntkrnlpa.exe	1.974731	4.710938	0.911645	0.59494	518868	132890	122.539063	0.705654
ntoskrnl.exe	2.092407	4.828491	0.743287	0.611262	517767	134932	124.707031	2.559062
srv.sys	0.341309	0.767334	0.108492	0.097917	91164	22616	27.789063	0.130862
tcpip.sys	0.344849	0.806885	0.105241	0.102179	90913	23610	28.527344	0.182019
tdi.sys	0.018188	0.040527	0.004537	0.004298	3978	1005	18.308594	0.008281
win32k.sys	1.781616	4.432373	0.771259	0.631631	538478	137020	121.511719	2.604051

3.2.2 System performance

In the system performance testing we have used our own custom benchmarking tool and also two other solutions for Microsoft Windows systems (NovaBench version 3 [11] and PerformanceTest 7 [12] evaluation version).

Benchmarked machine configuration: Intel Core2 Q9550 2.83GHz; 3327 MB RAM; ATI Radeon HD 5870; Windows 7 (32-bit).

Our benchmark results are presented in Table 2, NovaBench results are presented in Table 3. PerformanceTest benchmark results are presented in Table 4.

The legend for Table 2 is as follows:

- P_1 - protected machine (full instrumentation),
- Ps_1 - slowdown (P_1 versus native configuration),

Each benchmark program tend to produce results that vary during each system run. Our custom benchmarking tool executes four types of tests. The *Process Test* works by creating 50 instances of `calc.exe` program. The time is measured until all of the created processes are fully initialized. The *Write File Test* creates 100 10MB files and fills them with constant data. The *Read File Test* works in analogical way. In both cases time is measured until all files are processed. Last test (*Memory Test*) allocates (commits) 100 memory regions each 10MB wide, fills them with constant data and finally frees the committed regions. Time is measured until this process is finished for all the regions. Each test was performed 30 times for both protected and unprotected configurations. The arithmetic mean of the results was used in the comparison process.

Table 2: Custom benchmark results on native and protected systems.

Test	Native [s]	P_1 [s]	Ps_1 [%]
Process	47,013	49,132	4,314
Write File	212,391	204,903	-3,654
Read File	52,925	57,292	7,621
Memory	19,5143	19,168	-1,807

Our benchmark showed that the largest performance impact was observed in the *Read File Test*.

Table 3: NovaBench 3 results on native and protected system (higher number the better).

Test	Native system	Protected system
RAM Speed (MB/s)	4292	4303
Floating Point (ops/s)	102158140	102160836
Integer (ops/s)	334728164	334641264
MD5 Hashes (gen/s)	932387	932632
CPU Score	404	404
Graphics Tests Score	502	504
Drive Write Speed (MB/s)	84	85
Hardware Tests Score	31	31
NovaBench Score	1040	1042

Table 4: PerformanceTest 7 results on native and protected system (higher number the better).

Test	Native system	Protected system
Graph2D - Solid Vectors	3.9	3.5
Graph2D - Transparent Vectors	3.8	3.4
Graph2D - Complex Vectors	93.9	83.3
Graph2D - Fonts and Text	121.3	111.2
Graph2D - Windows Interface	71.6	62.7
Graph2D - Image Filters	433.4	432.1
Graph2D - Image Rendering	290.2	289.2
Memory - Small Block Alloc	2573.6	2553.3
Memory - Read Cached	2202.4	2202.0
Memory - Read Uncached	2022.5	2030.7
Memory - Write	2109.8	2103.5
Memory - Large RAM	1242.8	1232.6

The slowdown in this case was approximately 7% (P_1 case). It is also worth noticing that the *Write File Test* and the *Memory Test* were actually faster on the protected configuration. On the other hand the Passmark benchmark shows that protected system causes some negative performance effect on 2D graphics tests (Table 4) - where they oscillate from 0% to 14%. However the NovaBench (Table 3) benchmark shows something completely different. Appending to the NoveBench test results protected machine configuration was faster than the native one and scored 1042 points (where native scored only 1040). Additionally it is also worth noticing that performance slowdowns presented in our benchmarks are often not widely manifested.

4 Future Work

Please read `INSTALL.TXT` for more details.

5 Conclusion

This paper presents a method for securing the kernel of the operating system (in this case Microsoft Windows). Our engine uses static binary rewriting and code instrumentation techniques in order to monitor the control flow. We have shown that our protection is capable of detecting and blocking both remote (especially) and local attacks. Our solution, however, does not prevent against exploits that overwrite sensitive data and it also does not protect against vulnerabilities in 3rd party kernel modules (besides the technique presented in section 2.2.1).

We have also described techniques and ideas that may be implemented in the future. Performance impact together with operating system benchmarks was also presented. We believe that currently our solution provides a unique security solution for the Microsoft Windows kernel and does not require any special hardware features. Of course this mechanism cannot solve every security problem completely but it does make kernel exploitation much harder.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13:4:1–4:40, November 2009.

[2] Piotr Bania. Aslan (4514N) Metamorphic Engine. http://www.piotrbania.com/all/4514N/, 2006.

[3] Piotr Bania. SpiderPig - The Data Flow Tracer Project Homepage. http://www.piotrbania.com/all/spiderpig, 2008.

[4] CVE Database. CVE-2009-3103. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3103.

[5] CVE Database. CVE-2010-2743. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2743.

[6] Dino A. Dai Zovi. Practical Return-Oriented Programming. http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf.

[7] Erik Buchanan, Ryan Roemer, Stefan Savage, Hovav Shacham. Return-oriented Programming: Exploitation without Code Injection. https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf.

[8] Fyyre. Disable PatchGuard - the easy/lazy way. http://fyyre.ivory-tower.de/txt/bootloader.txt.

[9] Hex-Rays. Interactive Disassembler Pro. http://www.hex-rays.com/idapro/.

[10] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[11] Novawave Inc. NovaBench 3. http://novabench.com.

[12] PassMark Software. PerformanceTest 7. http://www.passmark.com.

[13] PaX Team. Pax Future. http://pax.grsecurity.net/docs/pax-future.txt.

[14] Piotr Bania. AutoDiff - Automated Binary Differential Analysis Project. http://autodiff.piotrbania.com.

[15] Piotr Bania. Security Mitigations for Return-Oriented Programming Attacks. http://piotrbania.com/all/articles/pbania_rop_mitigations2010.pdf.

[16] Piotr Bania. SMB2: 351 Packets from the Trampoline. http://blog.metasploit.com/2009/10/smb2-351-packets-from-trampoline.html.

[17] Ruben Santamarta. Stuxnet MS10-073/CVE-2010-2743 Exploit. http://reversemode.com/index.php?option=com_content&task=view&id=71&Itemid=1.

[18] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.

[19] Skape and Skywing. Bypassing PatchGuard on Windows x64. http://www.uninformed.org/?v=3&a=3.

[20] VUPEN Vulnerability Research Team. Technical Analysis of the Windows Win32K.sys Keyboard Layout Stuxnet Exploit. http://www.vupen.com/blog/20101018.Stuxnet_Win32k_Windows_Kernel_0Day_Exploit_CVE-2010-2743.php.

[21] Wikipedia. Stuxnet worm. http://en.wikipedia.org/wiki/Stuxnet.