

kBouncer: Efficient and Transparent ROP Mitigation

Vasilis Pappas
Columbia University
vpappas@cs.columbia.edu

April 1, 2012

Abstract

The wide adoption of non-executable page protections in recent versions of popular operating systems has given rise to attacks that employ return-oriented programming (ROP) to achieve arbitrary code execution without the injection of any code. Existing defenses against ROP exploits either require source code or symbolic debugging information, impose a significant runtime overhead, which limits their applicability for the protection of third-party applications, or may require to make some assumptions about the executable code of the protected applications. We propose kBouncer, an efficient and fully transparent ROP mitigation technique that does not require source code or debug symbols. kBouncer is based on runtime detection of abnormal control transfers using hardware features found on commodity processors.

1 Problem Description

The introduction of non-executable memory page protections led to the development of the return-to-libc exploitation technique [11]. Using this method, a memory corruption vulnerability can be exploited by transferring control to code that already exists in the address space of the vulnerable process. By jumping to the beginning of a library function such as `WinExec()`, the attacker can for example spawn a shell without the need to inject any code. Frequently though, especially for remote exploitation, calling a single function is not enough. In these cases, multiple return-to-libc calls can be “chained” together by first returning to a short instruction sequence such as `pop reg; pop reg; ret;` [27, 26]. When arguments need to be passed through registers, a few short instruction sequences ending with a `ret` instruction can be chained directly to set the proper registers with the desired arguments, before calling the library function [20].

In the above code-reuse techniques, the executed code consists of one or a few short instruction sequences followed by a large block of code belonging to a library function. Hovav Shacham demonstrated that using only a carefully selected set of short instruction sequences ending with a `ret` instruction, known as *gadgets*, it is possible to achieve arbitrary computation, obviating the need for calling library functions [30]. This powerful technique, dubbed *return-oriented programming* (ROP), in essence gives the attacker the same level of flexibility offered by arbitrary code injection without injecting any code at all—the injected payload comprises just a sequence of gadget addresses intermixed with any necessary data arguments.

In a typical ROP exploit, the attacker needs to control both the program counter and the stack pointer: the former for executing the first gadget, and the latter for allowing its `ret` instruction to transfer control to subsequent gadgets. Depending on the vulnerability, if the ROP payload is injected in a memory area other than the stack, then the stack pointer must first be adjusted to the beginning of the payload through a stack pivot [12, 34]. In a follow up work [7], Checkoway et al. demonstrated that the gadgets used in a ROP exploit need not necessarily end with a `ret` instruction, but with any other indirect control transfer instruction.

The ROP code used in recent exploits against Windows applications is mostly based on gadgets ending with `ret` instructions, which conveniently manipulate both the program counter and the stack pointer, although a couple of gadgets ending with `call` or `jmp` are also used for calling library functions. In all publicly available Windows exploits so far, attackers do not have to rely on a fully ROP-based implementation for the whole malicious code that needs to be executed. Instead, ROP code is used only as a first stage for

bypassing DEP [25]. Typically, once control flow has been hijacked, the ROP code allocates a memory area with write and execute permissions by calling a library function like `VirtualAlloc`, copies into it some plain shellcode included in the attack vector, and finally jumps to the copied shellcode which now has execute permission [12].

2 Current Solutions

Attackers are able to a priori pick the right code pieces because parts of the code image of the vulnerable application remain static across different installations. Address space layout randomization (ASLR) [25] is meant to prevent this kind of code reuse by randomizing the locations of the executable segments of a running process. However, in both Linux and Windows, parts of the address space do not change due to executables with fixed load addresses [15], or shared libraries incompatible with ASLR [34]. Furthermore, in some exploits, the base address of a DLL can be either calculated dynamically through a leaked pointer [21, 33], or brute-forced [31].

Other defenses against code-reuse attacks complementary to ASLR include compiler extensions [22, 28], code randomization [14, 5, 19], control-flow integrity [3], and runtime solutions [10, 8, 9]. In practice, though, most of these approaches are almost never applied for the protection of the COTS software currently targeted by ROP attacks, either due to the lack of source code or debugging information, or due to their increased overhead. In particular, from the above techniques, those that operate directly on compiled binaries, e.g., by permuting the order of functions [5, 19] or through binary instrumentation [3], require precise and complete extraction of all code and data in the executable sections of the binary. This is possible only if the corresponding symbolic debugging information is available, which however is typically stripped from production binaries. On the other hand, techniques that do work on stripped binary executables using dynamic binary instrumentation [10, 8, 9], incur a significant runtime overhead that limits their adoption. At the same time, instruction set randomization (ISR) [18, 4] cannot prevent code-reuse attacks, and current implementations also rely on heavyweight runtime instrumentation or code emulation frameworks.

In prior work we introduced a technique called in-place code randomization, which can be applied on third-party (stripped) binaries and practically incurs no performance overhead [29]. We showed that our system can successfully prevent a number of real-world ROP exploits and two automated ROP payload construction toolkits. However, although quite effective as a standalone mitigation, in-place code randomization is not meant to be a complete prevention solution, as it offers probabilistic protection and thus cannot deliver any protection guarantees.

3 Approach

The main idea behind our approach is to *perform runtime checks for abnormal control flow transfers* that usually appear when ROP code is executing. However, checking all control transfers at runtime, no matter how fast it is performed, is definitely going to introduce a high performance overhead, as indirect control flow transfers are very frequent in executing code. Based on this fact and the observation that malicious executing code will eventually need to perform system calls to achieve something meaningful, we refine the set of the control transfers that need to be checked to only the ones within the last part of paths that lead to the execution of a system call.

To define the “abnormal” control transfers we take a closer look into how ROP code executes. Before ROP code starts executing, the register that holds the stack pointer is set to the beginning of the ROP payload. Then, each gadget ends with a return instruction, which advances the stack pointer to the address of the next gadget and transfers control to it. At runtime, the return instructions of ROP code can be easily distinguished from the legitimate return instructions of the actual program, which are paired with call instructions and transfer control to the instruction following the call. Thus, a return instruction that transfers control to an instruction not preceded by a call is considered abnormal.

Figure 1 depicts the same idea. The rectangles correspond to memory snapshots, and the arrows that connect them correspond to control transfers. The upper part of the memory is the kernel space and the bottom one is the user space. The vertical dashed line denotes the point in time at which the flow of control is transferred from user space to kernel space (usually through a system call using `syscall`, `sysenter` or

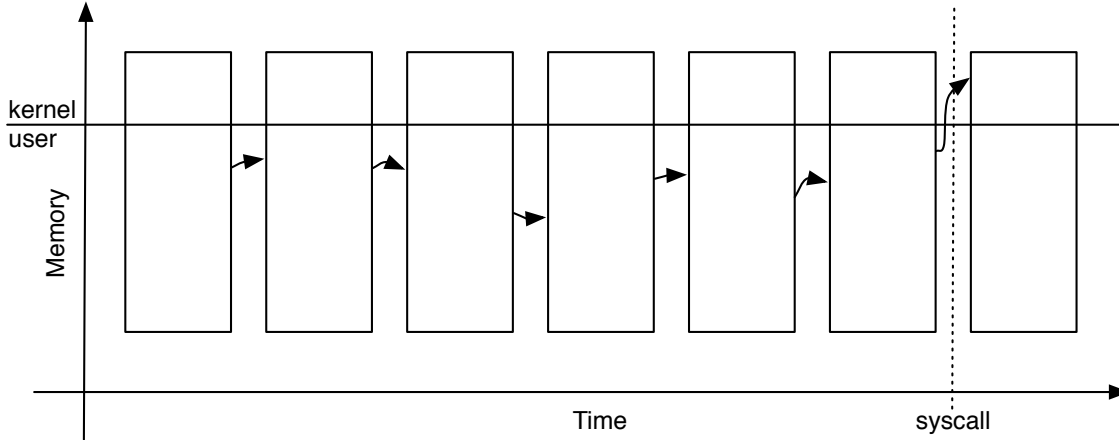


Figure 1: Whenever control is transferred from user to kernel space (vertical dashed line), our system checks the last N indirect branches to decide whether this is a benign system call or part of a ROP exploit.

`int 0x80`). This is the point where we check the control flow path for abnormal control transfers and decide whether it corresponds to a benign system call, or is part of a ROP exploit. In other words, our system acts like a bouncer for entering the kernel, thus the name kBouncer.

There are several different approaches that can be followed for applying such checks, each of them having different requirements, performance overhead, transparency, and deployment effort. Probably the easiest way would be to enhance the compiler to generate and embed the checks in the executable binary at compile time. But, as there is no way to tell which set of control transfers lead to system calls, we have to check all of them all. Also, deployment requires a huge effort as all programs have to be recompiled. Another option is binary rewriting. Its main advantage over compiler-level techniques is that no source code is required, but only debug symbols (.pdb files). Still, all control transfers need to be checked. Even worse, it breaks self-checksumming or signed code and cannot be applied to self-modifying programs. Dynamic instrumentation is another alternative that can handle even stripped binaries (no need for source code or debug symbols), but the runtime performance overhead of existing binary instrumentation frameworks slows down the normal execution of an application by a factor of a few times.

In contrast to the above approaches, we take advantage of a hardware feature that has been introduced in recent processors. The facility we used to apply the checks is a branch trace mechanism called Last Branch Recording (LBR) [17] (Section 17.4), supported by recent Intel CPUs. When LBR is enabled, the CPU stores the last executed branches in a set of model specific registers (MSR), on-chip. The values of these registers can be read using a special instruction, called `rdmsr`, only from privileged mode. In addition, the branches to be stored can be filtered based on their type: relative/indirect calls/jumps, returns, and so on. For our case, LBR provides the combined advantages of all the previous alternatives, without having any of their disadvantages. More precisely, it incurs zero-overhead for storing the branches; it is fully transparent to the running applications; it does not cause any incompatibility issues as it is completely decoupled from the actual execution; it does not require source code or debug symbols; and it can be dynamically enabled for any running application—there is no need for recompilation or any other type of intrusive instrumentation. Table 1 shows a summarized comparison of the alternative strategies.

Table 1: Comparison of alternative approaches for performing runtime control transfer checks.

	Performance	Requirements	Self-Modifying/ Checksumming	Deployment
Compiler-level	med	source	some	hard
Binary rewriting	med	pdb	no	med
Dynamic Instr.	high	-	yes	med
HW branch trace	low	-	yes	easy

4 Implementation

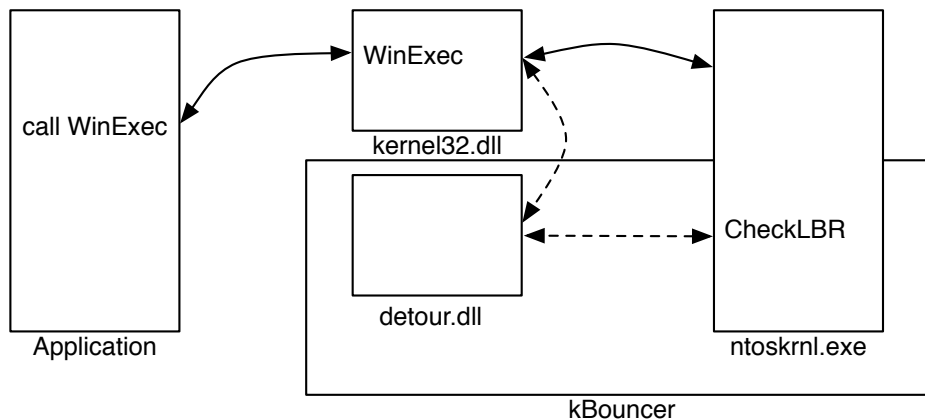


Figure 2: Instead of performing the checks on system call entry, kBouncer detours the execution when a Windows API is called, performs the checks in kernel-mode and then returns control back to the application.

To assess and demonstrate the effectiveness of our kBouncer, we developed a prototype implementation for the x86 64-bit version of Windows 7 Professional SP1. Ideally, our system would consist of a single kernel-level component that upon each system call entry would iterate the contents of the LBR registers and check the last control transfers. However, it turned out that this was not possible with the current version of the operating system.

A recent Windows kernel feature for the 64-bit version called PatchGuard [13] was added in order to protect against kernel-level rootkits by preventing any changes to critical data structures, such as the Service Description Table (SDT). Unfortunately, although effective against rootkits, PatchGuard removed the ability for legitimate applications, like AVs, to intercept system calls. Microsoft added a set of kernel-level APIs to filter network and file-system operations (Windows Filtering Platform [24]). Hopefully, future versions of the operating system will provide system call filtering capabilities as well. As system call interposition is currently not an option for the recent 64-bit versions of Windows, our checks had to be performed at a different point. Instead of checking the contents of the LBR registers upon system call entry, the checks are performed when Windows API functions are called, using the excellent Detours framework [16]. At the same time, our “detour” code maintains a communication channel with the kernel-level component using `DeviceIoControl`. Each time a Windows API function is called, the detour code sends an IO control to the kernel component, which informs it to check the contents of the LBR registers for abnormal control transfers. This process is depicted in Figure 2. Solid lines correspond to normal execution and dashed lines show how the detour code performs the check before `WinExec` is called.

More precisely, when a new application that we want to protect is starting, an appropriate IO control is sent to the kernel component to inform it to enable the LBR feature. Currently, the type of branches stored in the LBR are filtered so that only return instructions are recorded. The MSR registers (both the configuration and the ones that hold the branches) are considered part of the running process context and are preserved during context switches. Each time a detoured Windows API function is called, an IO control is sent to the kernel-level component to inform it to perform the check. For each destination address in the LBR registers, we fetch some bytes from the memory locations just before that address and try to decode them. If a call instruction does not precede the instruction at the destination address, then this is considered abnormal and further action is taken, such as informing the user or terminating the application.

5 Evaluation

In this section we present the results of our evaluation of the performance overhead, the false positive rate, and the effectiveness of our technique in preventing real-world ROP exploits. All the experiments were

performed on a single computer with the following specifications: Intel i7 2600S CPU, 8GB RAM and 128GB SSD. The operation system was the 64-bit version of Windows 7 Professional SP1.

We started by performing micro-benchmarks to measure the average time needed for checking the LBR register contents. The results are shown in Table 2. The first row corresponds to the net time of checking, that is, given a pair of addresses, we check whether the source decodes to a return instruction, and the instruction right before the destination to a call. We see that this can be performed very fast—it takes around 10ns on our system. Then, we tried to benchmark how a real check costs, that is, issue a `DeviceIoControl` system call, read the LBR registers, and check their contents. The duration for performing a batch of 10 million such operations was changing dramatically each time (as shown in the last row, the standard deviation is comparable to the average time). In order to find out the cause of that fluctuation, we performed a few more benchmarks. The second row shows the time needed to perform null `DeviceIoControl` system calls. In the third row, we also iterate over the contents of the LBR registers, which still results to stable measurements. However, if we also access the addresses from the LBR registers, then the measured duration starts to fluctuate. We are not sure if this is normal behavior, or if it is a result of non-optimal use of the kernel API. Still, the average check time is in the order of a few microseconds.

Table 2: Micro-benchmarks: Average time of checking the contents of the LBR registers for abnormal return control transfers.

Type	# Requests	Avg. Total Time ms (stddev)	Avg. Single ns
UserCheck	1B	10889.9 (312.9)	10.8
SysNull	10M	5145.0 (66.0)	514.5
SysLBR	10M	19981.8 (504.5)	1998.1
SysRead	10M	47267.7 (30925.6)	4726.7
SysCheck	10M	51520.2 (35998.8)	5152.0

Next, we switched to macro-benchmarks for measuring the false positives rate and estimate the overall overhead when an application is protected by kBouncer. A false positive occurs when our system is not able to decode a call instruction before the destination of a return instruction, but still, no ROP code is executing. To gather all the executed return control transfers we used the Pin dynamic instrumentation toolkit [32, 23]. We used three applications: Windows Media Player 12 (30 second music playback), Internet Explorer 9 (load <http://www.microsoft.com>) and Adobe Reader v9.3 (open Intel’s manual, 4128 pages). For each case, our Pin tool dumps information about all the call and return instructions and system calls. To better simulate the LBR, the Pin tool buffers a number of return branches and, whenever the buffer gets full, then copies the bytes from the source-destination locations and dumps everything to disk. The total number of calls/returns and system calls is shown in Table 3. Internet Explorer exhibits a lower number of call-return pairs and system calls because we were only tracking the parent process. As shown in the same table, we encountered no false positives. We should note here that whenever a return instruction targets a writable and executable memory region, then no check is performed. This is because these type of memory regions are mostly used for just-in-time (JIT) compilation and, as our checks happen after the instructions have executed, the contents may have already changed. A few of these cases showed up while analyzing Windows Media Player’s data (the WX buffer was allocated by MF.dll).

To estimate the additional overhead of the checks before each system call is performed, we counted the total number of system calls (fifth column in Table 3). Also, we measured the real, user, and system time for each of the application using the `GetProcessTimes` API. As we can see in the last two columns, the measured overhead is very low. We note here that this overhead corresponds to the worst case scenario where all system calls are checked. Ideally, we could skip the checks for “harmless” system calls such `GetSystemTime`, which are not useful for the attackers.

In the last part of our evaluation, we tested whether our prototype is able to prevent two real-world ROP exploits. The first one targets Adobe Reader v9.3.4 and exploits a stack based overflow in `CoolType.dll` [1]. The second one targets MPlayer Lite r33064 and exploits a buffer overflow that overwrites a SEH pointer [2]. Both exploits work on the latest and up-to-date (as of April 1st, 2012) version of Windows 7 Professional SP1 64-bit. We note here that we had to manually update the Mplayer ROP payload to correctly calculate

Table 3: Macro-benchmarks: False positive rate and estimated runtime overhead.

Type	Times			Calls/ Returns	Syscalls	False Positives	Overhead	
	real	user	sys				ms	%
wmplayer	30.73	0.37	0.21	30.8M	194K	0*	33.8	6
iexplore	7.24	0.06	0.04	1.5M	31K	0	5.4	5
reader	4.11	1.32	0.24	35.3M	107K	0	18.7	1

the offset of `VirtualProtect` within `kernel32.dll`, as the public version of the exploit was written based on a previous version of `kernel32.dll`.

The Adobe Reader exploit creates a file (`CreateFileA`), maps it in RWX mode (`CreateFileMappingA`, `MapViewOfFile`), copies the shellcode in the newly mapped area, and executes it. Similarly, the MPlayer exploit changes the permissions of the memory region where the shellcode resides to RWX (`VirtualProtect`) and executes it. In both cases, the shellcode simply invokes the Calculator using `WinExec`. Our prototype successfully identified abnormal return control transfers when the `CreateFileA` and `VirtualProtect` functions were called in Adobe Reader and MPlayer, respectively.

6 Limitations

The main limitation of the current prototype is that it does not prevent Jump-Oriented Programming (JOP) [6]. JOP is similar to ROP, but it uses indirect jump instructions instead of returns to transfer control from gadget to gadget. Clearly, constructing JOP programs is far more difficult than ROP because the attacker has to maintain a memory region with the payload and manually advance the pointer to the next gadget and transfer control to it. The same functionality is conveniently achieved by a single return instruction in ROP. Although JOP is not covered by our current prototype, forcing the attackers to switch from ROP to JOP significantly raises the bar. In our future work, we plan to further examine the properties of control transfers within a JOP program and try to extract abnormal behaviour patterns that could be detected by our system.

Another way to evade our system is by using ROP code to overwrite the contents of the LBR registers with legitimate call-return control transfers just before performing the system call. Although this may sound simple as an idea, it is extremely hard to realise. One way would be to only use gadgets that are preceded with a call instruction, but that would significantly shrink the number of available gadgets, thus making it very difficult to construct useful payloads. Another way would be to repeatedly execute a special gadget that calls a function and returns control back using a jump instruction instead of a return, which again requires extra effort and is not guaranteed to be satisfied.

References

- [1] CVE-2010-2883. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2883>.
- [2] Mplayer (r33064 lite) buffer overflow + rop exploit. <http://www.exploit-db.com/exploits/17124/>.
- [3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*, 2005.
- [4] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*, 2003.
- [5] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

- [6] Tyler Bletsch, Xuxian Jiang, Vince Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.
- [7] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, 2010.
- [8] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, 2009.
- [9] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC)*, 2009.
- [10] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A practical protection tool to protect against return-oriented programming. In *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.
- [11] Solar Designer. Getting around non-executable stack (and fix). <http://seclists.org/bugtraq/1997/Aug/63>.
- [12] Úlfar Erlingsson. Low-level software security: Attack and defenses. Technical Report MSR-TR-07-153, Microsoft Research, 2007. <http://research.microsoft.com/pubs/64363/tr-2007-153.pdf>.
- [13] Scott Field. An introduction to kernel patch protection. <http://blogs.msdn.com/b/windowsvistasecurity/archive/2006/08/11/695993.aspx>.
- [14] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.
- [15] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.
- [16] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.
- [17] Intel. Intel 64 and ia-32 architectures software developers manual, volume 3b: System programming guide, part 2. <http://www.intel.com>.
- [18] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*, 2003.
- [19] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.
- [20] Sebastian Kraemer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. <http://www.suse.de/~kraemer/no-nx.pdf>.
- [21] Haifei Li. Understanding and exploiting Flash ActionScript vulnerabilities. CanSecWest, 2011.
- [22] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with “return-less” kernels. In *Proceedings of the 5th European conference on Computer Systems (EuroSys)*, 2010.

- [23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.
- [24] Microsoft. Windows filtering platform. [http://msdn.microsoft.com/en-us/library/windows/desktop/aa366510\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/aa366510(v=vs.85).aspx).
- [25] Matt Miller, Tim Burrell, and Michael Howard. Mitigating software vulnerabilities, July 2011. <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26788>.
- [26] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), December 2001.
- [27] Tim Newsham. Non-exec stack, 2000. <http://seclists.org/bugtraq/2000/May/90>.
- [28] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.
- [29] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, 2012.
- [30] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, 2007.
- [31] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS)*, 2004.
- [32] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. Dynamic program analysis of microsoft windows applications. In *International Symposium on Performance Analysis of Software and Systems*, 2010.
- [33] Peter Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. <http://vreugdenhilresearch.nl/Pwn2Own12010-Windows7-InternetExplorer8.pdf>.
- [34] Dino A. Dai Zovi. Practical return-oriented programming. SOURCE Boston, 2010.