You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code

Michael Backes
Saarland University
MPI-SWS
backes@mpi-sws.org

Thorsten Holz
Ruhr-Universität Bochum
thorsten.holz@rub.de

Benjamin Kollenda
Ruhr-Universität Bochum
benjamin.kollenda@rub.de

Philipp Koppe
Ruhr-Universität Bochum
philipp.koppe@rub.de

Stefan Nürnberger
Saarland University
nuernberger@cs.uni-saarland.de

Jannik Pewny
Ruhr-Universität Bochum
jannik.pewny@rub.de

ABSTRACT

Code reuse attacks allow an adversary to impose malicious behavior on an otherwise benign program. To mitigate such attacks, a common approach is to disguise the address or content of code snippets by means of randomization or rewriting, leaving the adversary with no choice but guessing. However, disclosure attacks allow an adversary to scan a process—even remotely—and enable her to read executable memory on-the-fly, thereby allowing the just-in-time assembly of exploits on the target site.

In this paper, we propose an approach that fundamentally thwarts the root cause of memory disclosure exploits by preventing the inadvertent reading of code while the code itself can still be executed. We introduce a new primitive we call *Execute-no-Read* (XnR) which ensures that code can still be executed by the processor, but at the same time code cannot be read as data. This ultimately forfeits the self-disassembly which is necessary for *just-in-time code reuse attacks* (JIT-ROP) to work. To the best of our knowledge, XnR is the first approach to prevent memory disclosure attacks of executable code and JIT-ROP attacks in general. Despite the lack of hardware support for XnR in contemporary Intel x86 and ARM processors, our software emulations for Linux and Windows have a run-time overhead of only 2.2% and 3.4%, respectively.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized Access*; D.4.6 [**Operating Systems**]: Security and Protection—*Information Flow Controls*

General Terms

Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660378.

Keywords

return-oriented programming, information leaks, memory disclosure exploits, buffer overflows, code reuse attacks

1. INTRODUCTION

Buffer overflows, dangling pointers and related memory corruption vulnerabilities constitute an important class of security vulnerabilities. Despite such control-flow hijacking attacks being known for more than two decades, they are still one of the three most prevalent attack vectors, e.g. through vulnerable PDF viewers, browsers, or operating system services [30, 46]. Over the last few years, a number of attack techniques have been proposed (e.g., [7, 11, 20, 27, 42]). In practice, particularly so-called *code reuse attacks* are relevant, since they enable an attacker to redirect control flow through a program with the intent of imposing malicious behavior on an otherwise benign program. More specifically, an attacker does not need to inject her own code into a vulnerable program, but she can reuse existing code fragments (so-called *gadgets*) that perform malicious computations of her choice by chaining several gadgets together (dubbed *gadget chains* [36]).

As part of the typical arms race in computer security, a large number of potential defenses have been proposed (e.g., [4–6, 37, 49]). In practice, techniques such as *address space layout randomization* (ASLR) and *data execution prevention* (DEP) are most widely deployed: these two techniques are available in virtually all modern operating systems. More specifically, ASLR has been invented to make it impossible to predict where specific code resides [34, 35]. While the technique has been widely adopted, researchers demonstrated that it is often ineffective in practice [38, 39]: Since only the code segment's base address is randomized, a leaked pointer is sufficient to infer all code addresses.

To overcome this problem, several sophisticated randomization techniques were proposed that perturb the code such that the location of gadgets inside the code segment becomes unpredictable. The basic idea of these fine-grained, load-time ASLR schemes is to either slice the code into chunks and shuffle them [18, 24], or to reorder the instructions in the code segment [47] or within individual basic blocks [32]. The resulting systems have a rather small performance overhead and they effectively protect against an adversary that uses precomputed gadget chains during an attack.

However, a more powerful adversary can also compute the gadget chains on-the-fly: Snow et al. introduced *just-in-time code reuse* attacks [40] that enable an attacker to find ROP gadgets in spite of fine-grained randomization of the process' memory. The attack exploits a known memory disclosure vulnerability, which allows reading arbitrary memory and ultimately disassembling the disclosed memory of the vulnerable process. Based on this information, she can then dynamically compute gadget chains. Such just-in-time code reuse (shortened as *JIT-ROP* in the following) attacks bypass existing defenses and represent one of the most sophisticated attacks proposed up to now.

In this paper, we introduce a way to systematically prevent JIT-ROP attacks by addressing the root cause. Our approach is based on the insight that such attacks require a disclosure vulnerability that enables an adversary to read arbitrary memory locations, which allow searching for gadgets. As such, a viable approach to prevent JIT-ROP attacks is to forbid code from being read and hence disassembled. Consequently, our "Execute-no-Read" primitive prohibits an attacker from constructing a gadget chain on-the-fly. Unfortunately, contemporary Intel x86 processors are designed in such a way that memory pages must always be readable in order to be executable. The same applies to ARM processors since these CPUs also only feature one read/write bit, an eXecute Never (XN) bit, and one bit to distinguish between user and kernel mode [1].

However, we can implement and enforce our primitive by modifying the operating system and extending the memory management system. In spirit, this is similar to the first implementations of the $W \oplus X$ (**W**ritable **xor** e**X**ecutable) security model [34]. We have implemented a prototype of our new primitive as a kernel-level modification for both Linux and Windows. Evaluation results show that XnR successfully thwarts memory disclosure attacks with the intent of disassembling code, while benign programs continue to run unaffectedly. Furthermore, the overhead introduce by our new primitive is reasonable: in micro and macro benchmark we found that average run-time increase is only 2.2% for Linux and 3.4% for our Windows implementation. The security parameter we introduce defines a trade-off between security and performance.

In summary, we make the following three contributions:

- We systematically study the root causes behind disclosure vulnerabilities. Our insight is that current processors only allow memory to be marked as non-writable or executable. However, code that is supposed to be executed must remain readable in memory and hence poses a risk for disclosure attacks.

- We propose the primitive "Execute-no-Read" (XnR) that maintains the ability to execute code but prevents reading code as data, which is necessary to disassemble code and finally find ROP gadgets (especially when they are constructed on-the-fly).

- We implemented a prototype of our approach in software as a kernel-level modification for Linux and Windows. We achieve such hardware emulations by patching the memory management system in order to detect inadvertant reads of executable memory. Our prototype is available for both Linux and Windows and introduces only a small performance overhead.

2. PREVENTING DISCLOSURE EXPLOITS

We now first review the intuition behind our approach and define the threat model we use throughout this paper. Afterwards, we provide an overview of the design of our contribution to prevent ROP-style exploitation attacks and also explain the rationale and interplay of XnR in this task.

2.1 Motivation

Traditional return-oriented programming (ROP) attacks can (mostly) be contained with the help of *address space layout randomization* (ASLR): this technique makes it harder for an adversary to guess or brute-force addresses that are needed for ROP gadgets. However, ASLR can be bypassed if the attacker can exploit an information leakage vulnerability, since such a vulnerability inadvertently reveals a valid, current address inside the running program. Hence, an adversary learns an address used inside a process, which allows her to circumvent ASLR. Due to the fact that conventional ASLR moves the entire code segment en bloc (which leaves the relative locations of objects and functions intact), *only one* such leaked address from the code segment is enough to calculate the address of *every* instruction inside a process. This effectively enables an attacker to infer all other objects or functions relative to the leaked address because the relative distances between functions stay exactly the same.

In general, information leaks represent a challenging problem that is hard to solve. To prevent the actual exploitation of such leaked pointers, several mitigations have been recently proposed in the literature. For example, *binary stirring* makes it impossible to reliably use ROP gadgets by substituting instructions with semantically equivalent instructions, thereby effectively concealing known patterns that could be used for a ROP attack [47]. An alternative approach is to apply finer randomization such that a leaked pointer reveals as little information as possible about its surrounding code [18, 26, 48].

Unfortunately, in the light of a disclosure vulnerability, an attacker may read the address space during runtime. This enables her to disassemble a running process with the intent of finding ROP gadgets. Snow et al. [41] demonstrated the practical viability of such an attack: given a memory disclosure vulnerability, it is possible to assemble ROP gadgets on-demand by dynamically compiling a gadget chain. Their *just-in-time code reuse* attack repeatedly exploits a memory disclosure vulnerability to map portions of a process' address space with the objective of reusing the so-discovered code in a malicious way.

Our solution prevents such JIT-ROP attacks by eliminating its root cause, namely by detecting and preventing the exploited disclosure vulnerability. More specifically, as soon as a process tries to read its own code as data, XnR consideres this illegal behavior. This prevents the first necessary step of disclosure vulnerability. We demonstrate that this primitive can be implemented and enforced with a reasonable overhead on contemporary computer systems.

2.2 Threat Model

We make several assumptions about an attacker. First, we assume a commodity operating system (i.e., Linux or Windows) that runs a user mode process that contains a memory corruption vulnerability. The attacker's goal is to exploit this vulnerability in order to divert the control flow and execute arbitrary code of her choice. Furthermore, the

attacker knows the process' binary executable and the OS version of the attacked system. Hence, she can precompute potential gadget chains in advance and use them during the attack.

Second, we assume that the process has at least one memory disclosure vulnerability, which makes the process read from an arbitrary memory location chosen by the attacker and report the value at that location. This vulnerability can be exploited any number of times during the runtime of the process. Note that the process itself performs the read attempt: both address space and permissions are implied to belong to the process.

Third, the attacker can control the input of all communication channels to the process, especially including file content, network traffic, and data entered over the user interface. However, we assume that the attacker has not gained prior access to the operating system's kernel and that the program's binary is not modified. Apart from that, the computational power of the attacker is unlimited. In particular, she can memorize disclosed memory, disassemble it, search it for gadgets, and find meaningful chainings of those gadgets.

2.3 Assumptions

Given this attacker model, we aim at preventing JIT-ROP attacks. As discussed before, the main challenge is that an attacker can exploit an information leak to dynamically construct gadget chains on-the-fly. When attempting to prevent such an attack, we can thus prohibit the actual information collection phase. Before elaborating on the details of our solution, we discuss the assumptions that are necessary for XnR to protect the system in a holistic way.

While XnR is a powerful primitive, we expect it to be used in conjunction with two other security mechanisms: (i) $W \oplus X$ and (ii) fine-grained load time ASLR.

First, we expect that the $W \oplus X$ security model holds, which states that memory cannot be writeable and executable at the same time. This prevents an attacker from modifying existing code to suit her needs, or writing data and executing it as shellcode afterwards (at least without a syscall in between). On modern operating systems, the enforcement of $W \oplus X$ is a standard precaution based on *data execution prevention* (DEP [29]).

When the $W \oplus X$ primitive is enforced, an attacker has to take advantage of code that is already available in memory. As discussed earlier, diverting small fragments of code from their intended purpose is usually called return-oriented programming, and ASLR [35] is effective against traditional ROP attacks: the location of gadgets changes at the load time of a process, which makes it infeasible for an attacker to guess their position (even if she has precomputed potential gadget chains in advance). Again, this state-of-the-art protective measure is enabled by default on modern operating systems.

Unfortunately, conventional ASLR only changes the base address of the entire code segment and hence a single leaked pointer might uncover all the gadget's addresses since the relative addresses between them did not change. Therefore, we require a more fine-grained variant of ASLR that also changes the *relative* address among gadgets. Several such fine-grained load time ASLR methods were recently proposed in the literature [18,24,47] and we assume that one of these methods is used in conjunction with XnR.

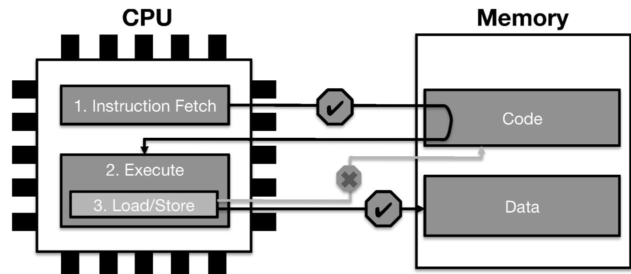

Figure 1: The XnR primitive distinguishes between legitimate code execution (instruction fetch) and illegal access to code using load/store instructions.

2.4 The Execute-No-Read Primitive

The goal of our "Execute-no-Read" (XnR) primitive is preventing attacks that leverage just-in-time code reuse for exploitation. To this end, XnR prevents the step of dynamically gathering gadgets, which is a necessary precondition in the context of fine-grained ASLR, as the gadgets and their positions are not known to the attacker.

Since contemporary processors all feature a von-Neumann memory architecture that mixes code and data, the determination whether a particular piece of memory contains code is challenging. Moreover, the concept of non-readable, but executable, memory does not exist: memory permissions only allow to toggle the ability to write to memory or the ability to execute memory, where executable permissions imply read permissions. As a result, XnR cannot be implemented with current hardware.

Emulating XnR in Software.

To counteract this deficiency, we propose a way to emulate XnR in software by extending the memory management system of the operating system. To demonstrate the practical feasibility of this approach, we modified up-to-date versions of the two mainstream operating systems Windows and Linux. Our experiments (see Section 4) show the effectiveness of our solution against memory disclosure attacks and that legitimate execution is not hindered, except for a small performance overhead.

For XnR to work, we need to distinguish memory accesses between legitimate access to data and read attempts to code (see also Figure 1 for an illustration). The memory management unit (MMU) present in virtually all modern processors such as x86 and ARM introduced the notion of a *process*, which is a complete address space that exists from each process' point of view. Each such address space can have memory regions marked as writable and others as read-only. While the MMU can detect write attempts to any part of a process' memory, detecting read attempts is not supported. Read attempts can only be detected by declaring a certain memory region to be *non-present* in the MMU. However, a *non-present* memory region cannot be executed anymore.

To overcome this challenge, our solution makes use of the so-called *page fault handler*. Every time the MMU detects a memory access violation in a process, the page fault handler of the operating system kernel is called. An access violation may occur when a process tried to read memory that is marked as *non-present* or when a process tried to write to memory that is marked as read-only. The granularity at which memory regions can have *writable* and *present*

attributes is defined by the hardware as so-called memory pages (usually 4 kB). When a page fault occurs, the process is halted and control is switched to the kernel, which tries to handle the page fault. A page fault is no exception in the ordinary run of a program but happens thousands of times during normal execution. The reason is *demand paging*, a performance feature that starts every process with an empty address space and only maps pages that are actually accessed. Demand paging is handy for our solution since every first access to a page is caught due to the initially empty process space.

Our modified page fault handler checks the violation conditions and decides whether to continue normally (i.e., to map the missing page into the address space) or to terminate execution if a memory disclosure was detected. Each page fault is provided with additional information such as the address where the fault occurred and whether the access was generated during an instruction fetch. The latter is crucial information for our XnR solution: If the CPU was trying to execute an instruction in a memory page that was *non-present*, this constitutes a legitimate operation and we let the usual *demand paging* routine of the kernel fetch the page and mark it *present*. If, on the other hand, the access violation did not occur due to an instruction fetch, then the processor was trying to read memory as data. In this case, XnR has to distinguish whether the violating address is indeed inside a valid region of data or points to code. If the address lies inside a data region of the process, our page fault handler continues normally by mapping the missing page. Otherwise, the process tried to read from a code region, which is illegal, as we assume executable code not to be readable. In this case we terminate the process with an error and prevent the attack.

Distinguishing between data and code regions in a process is achieved by interpreting the executable file formats, which provide information as to which memory region is executable (code) and which is readable (data).

To overcome the problem that only the first access to every page is detected by the page fault handler, we need to make pages *non-present* again after data access in a particular page has finished. However, the halting problem dictates that it is not generally possible to decide when and if a program finished executing a particular memory region. Our solution is to wait until another page than the currently used one is accessed and then we mark the last used page as being *non-present*. This guarantees new page faults will be triggered whenever an already accessed page is accessed again. Whenever execution runs outside of one memory page into another, the last page gets inaccessible while the new page is set to *present* in the same atomic operation.

Sliding Window.

Instead of keeping only one page *present* at all times, we introduce a security and performance parameter, which keeps the last recently used n pages *present* while setting all the others to *non-present* (see Figure 2). This allows the kernel to keep more than one page active at the same time, which reduces the chance of an congestion because the code is constantly jumping between two pages.

The parameter n constitutes a trade-off between performance and security. For $n = 1$, only the page in which execution currently takes place is mapped at any time. This is the only page that does not trigger a page fault when being accessed and hence is the only page that could potentially be read by an attacker without being noticed by our XnR detection. Since the attacker can only read the page in which execution takes place, she has to know the current execution address (instruction pointer) at the time of the exploit. Even if it were possible to guess the execution address correctly and the attacker exploits the disclosure vulnerability, she would only have access to the single page in which the code for the disclosure vulnerability resides because accessing another position of code would trigger an XnR exception.

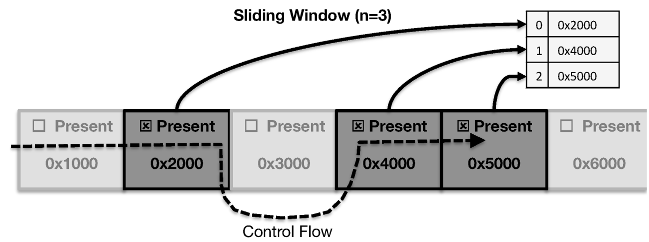

Figure 2: To improve performance, the parameter n adjusts the window length of the sliding window that keeps up to n pages in the address space. At the same time n is the security parameter.

For $n > 1$ mapped pages, the situation is more complex as there are now n pages which can be accessed without our XnR detection mechanism noticing. If the attacker can now determine the address of one of those pages at runtime, she can read that page and search for gadgets. However, as the last visited pages depend on the past control-flow, she is either limited to the same n pages each time (in case of invariant control-flow) or runs the risk of picking a path, which was not executed this time and hence crashes the program. If n is equal to the total number of code pages of a process, then all pages stay mapped in the process' address space forever. This would resemble XnR not being active at all.

3. IMPLEMENTATION

As discussed above, XnR would be a very helpful hardware feature, but unfortunately this primitive is not present in any modern processor. To show the general feasibility and effectiveness of our approach, we implemented a software-based prototype of our XnR solution for two mainstream operating systems, i.e., Microsoft Windows and GNU Linux.

The two implementations share the same concept, namely using hardware page faults to emulate an XnR hardware feature. However, their specific implementations differ vastly due to the different natures of Windows and Linux. While the Linux kernel is freely available as open source, the Windows kernel is closed source and was not designed for third parties to modify fundamentals such as the memory management. This naturally led to different approaches with respect to *how* and *where* XnR was engaged in the specific kernels. In general, the fact that the Linux kernel is provided in source code allows for a proper integration with the existing memory management and process run-time system, whereas Windows prohibits any modifications in the first place and hence forces the Windows implementation of XnR to be an outer shell around the immutable Windows kernel functions.

The concept that both our implementations share is to intercept any access to code or data, and to then decide whether that access was triggered due to an instruction fetch (i.e., the processor is executing code) or due to data access (i.e., load and store operations on data). In order to decode and execute instructions, the processor must read them from memory. This so-called *instruction fetch* resembles implicit access to memory. Additionally, an instruction being executed by the processor might also explicitly access memory by means of *load/store* operations (i.e., reading memory to a processor register or storing a processor register to memory).

This leaves us with three types of memory access that we need to distinguish:

Instruction Fetch: The processor fetches a byte from memory in order to decode and execute the instruction that it resembles. This constitutes a legal operation that takes place during code execution.

Load/Store of Data: An instruction may accesses memory that either contains code or data. If the load/store instruction targets data, this constitutes a legal operation that is necessary to operate on data.

Load from Code: However, if a load instruction targets code, this constitutes a programmatic disassembly that we consider illegal.

Since it is impossible to efficiently intercept each access to memory in software, we chose to leverage the already existing memory management unit (MMU) of modern processors. The MMU implements virtual memory and thereby enables process isolation. The operating system, in concert with the MMU, allows for the illusion of a contiguous virtual address space for every process. Only the used parts of each address space (i.e., each process) are actually mapped to physical memory, which is done completely transparent to each process.

The MMU divides memory in the smallest addressable unit, a memory page, for which a translation from virtual memory address to physical memory pages can be set up for each page and for each process. The MMU also allows trapping access to a certain memory page. In particular, operating systems detect illegal access to memory (e.g., writing to read-only portions of memory) that way. Moreover, dynamic growth of memory portions can be implemented using those access traps. This way the stack can grow dynamically and only consumes as much memory as is actually used. The so-called *demand paging* is responsible for mapping memory only when it is first accessed, thereby saving memory that is never accessed or whose size is even unknown a priori.

Because the *demand paging* already facilities a framework to detect access to memory, it was a suitable position to entrench our XnR solution in the Linux kernel. The closed-source nature of Windows, however, does did not allow us to change already defined page mappings (*present/non-present*) and we had to resort to marking pages kernel-only in order to trigger access violations from user mode.

3.1 Linux Kernel

The advantage of an implementation in the *demand paging* subsystem of Linux is that an illegal access can be detected before it can actually happen, i.e., before the targeted code is accessible by the user mode process. As a basis for our implementation we chose the recent stable Linux kernel version 3.13.7 for 64-bit x86 CPUs. The Linux memory management is fairly sophisticated and uses page faults not only to detect illegal access to memory but to transparently implement *demand paging*, Copy-on-Write (COW), and to map files to memory.

A general overview of how XnR is integrated in the Linux kernel is given in Figure 3. A typical XnR check works as follows: For every page fault, Linux first checks if the fault is due to access to invalid memory (i.e., the accessed position is not supposed to contain memory) or if the fault can be gracefully resolved by mapping a new page of memory and continuing execution. If the page fault handler detects that the allegedly non-present memory is actually supposed to be present, *demand paging* is invoked to pretend the accessed page existed in the first place. This way, the address space of a process can be built on demand, rather than wasting memory and time by pre-loading the entire address space at program start.

We added XnR detection and decision logic as well as bookkeeping to the commodity *demand paging* procedure. Before the actual *demand paging* procedure is invoked, we check whether the page fault was caused due to an instruction fetch operation or due to a load instruction that read memory (2.). For this purpose, the x86 CPU pushes a `word` to the stack that encodes the type of access violation (e.g., read, write, user, kernel, instruction fetch, data access). If an instruction fetch happened, we engage bookkeeping that logs which page we allowed to execute and let the *demand paging* logic do its conventional job of mapping the particular page that was not present (4.). If the target address is data, it is checked to which logical area it belongs to (3a.). If the faulting address lies in a segment that is marked executable but not readable, the process tried to read instructions from memory.

In order to know the access permissions of a particular memory area, we use the access type information provided in the section meta data of the executable file or shared library file. For dynamically allocated memory(`malloc(), mmap()`), the permissions are provided with the respective syscalls. This flexible method allows the developer to selectively protect single executables, single shared libraries or even specific `mmap`'ed memory inside a process. The latter is particularly useful in the case of Just-in-Time compilation, such as Java or Mono.

To ensure that pages that have already been paged into the address space of a process (e.g., by *demand paging*) will be checked again when they are accessed again, they need to trigger a page fault again, otherwise accessing them might evade XnR detection. To this end, we implemented a *sliding window* that evicts pages from a process' address space in order to ensure that at most n pages are mapped into the address space at any time. This guarantees that access to memory outside the n mapped pages will be caught and can be checked by our XnR implementation. At the same time, the number of simultaneous pages n is a parameter for both security and performance. If n is set to 1, only one page is active at any time and hence every access to another location in memory will trigger a page fault and therefore an XnR check.

The Linux implementation is designed as a patch against the 3.13.7 kernel and works with 64 bit and 32 bit programs running on an x86-64 kernel. The patch modifies the

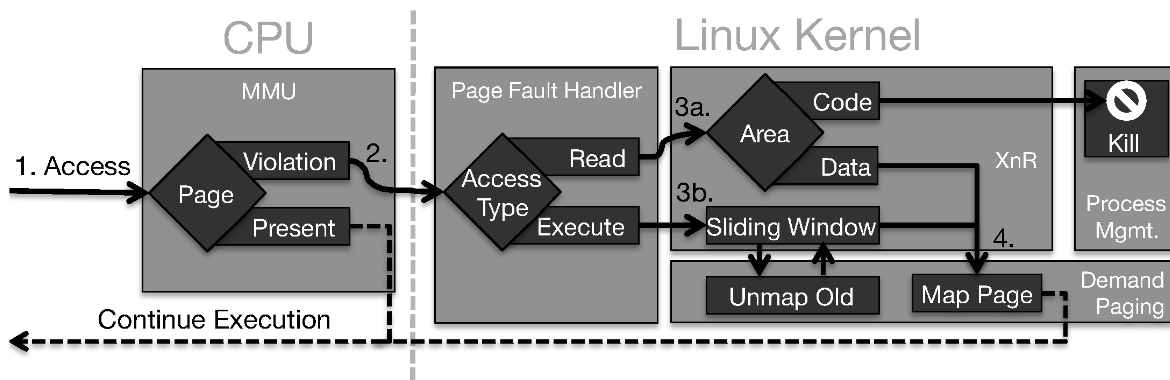

Figure 3: Flow diagram of how the CPU, MMU and parts of the Linux kernel interact in order to implement XnR.

existing minor page fault handler as well the file mapping part of *demand paging* and adds the "Execute-no-Read"-sub-subsystem to the memory management subsystem of Linux. In total, we have modified 570 lines of code.

3.2 Windows Kernel

Lacking the ability to modify the Windows kernel as freely as the Linux kernel, we opted for a minimally invasive approach. Our custom page fault handler is invoked before the normal Windows handler by inserting it directly into the Windows kernel's interrupt descriptor table (IDT). Since Patchguard [28] prevents modifications of the IDT, our proof-of-concept implementation can only be run in *test mode*. We applied the Windows kernel patch to the 64 bit version of Windows. Our Windows XnR implementation protects 64 bit and 32 bit processes. All applications need to explicitly activate our protection using control functions exposed by our created virtual device.

In contrast to Linux, Windows does not provide a framework to selectively page-in memory that is currently accessed or to page-out memory that is not recently used. Instead, our Windows implementation resorts to setting the `privileged` flag for memory pages that contain code. This in turn causes page faults whenever a user application tries to access them. The raised page fault handler then decides whether to allow or deny access. In case of a legitimate execution (instruction fetch), the `privileged` flag is removed, resulting in unaffected execution. In case of an XnR violation, the exception is passed to the Windows handler, which results in termination of the faulting program. Analogously to the Linux implementation, we use a *sliding window* to keep at most n pages accessible at all times, while all the remaining pages have their `privileged` bit set. Our Windows implementation has a total of 1,540 lines of code.

4. EVALUATION

In this section, we show the effectiveness of XnR. To this end, we demonstrate that memory disclosure attacks are successfully caught by XnR (no false negatives), while benign programs are not affected by our modifications (no false positives). Moreover, we conducted performance evaluations that demonstrate the low overhead of our XnR solution.

4.1 Precision and Effectiveness

To evaluate the precision, we conducted several experiments that on the one hand ensure that memory disclosure attacks are successfully caught by XnR. This is achieved by showing that a typical exploit fails. On the other hand, we need to ensure that benign programs are not affected, i.e., legitimate code reads are still possible and no false positives occur.

Detection of Exploits.

To demonstrate the security of our solution, we exploited a memory disclosure vulnerability in the standard Linux `netcat` program, which is a powerful utility for establishing or listening on TCP/UDP network connections. To this end, we weakened the `netcat` source by implementing a memory disclosure vulnerability that can be triggered by TCP packets that are too long. In the course of this buffer overrun, the attacker's input data overwrites an internal buffer pointer. This buffer pointer is used to assemble network packets. This enables an attacker to craft malicious packets in such a way as to intentionally overwrite the buffer pointer and thereby directing the TCP response buffer to arbitrary memory.

When our modified version of `netcat` is run on an unprotected Linux, we can successfully direct the memory disclosure vulnerability to return arbitrary data and code to the attacker. With enabled XnR protection, however, the offending `netcat` process is killed as soon as the memory disclosure vulnerability is directed to a region containing executable code. Since JIT-ROP relies on the necessary precondition of reading memory such that gadgets can be constructed on-the-fly, such an exploitation technique can be successfully detected and prevented by our approach.

Legitimate Code Reads.

Since XnR prevents all access if it detects reads from the code segment, one must ponder whether there are legitimate reasons to read from the code segment. If a benign program reads its own code segment as data for a legitimate reason, blocking such access would constitute a false positive.

In our evaluation, we found that both common Linux programs and standard Windows DLL functions attempted to read code during normal program execution. For the open source Linux programs it was easy to find and fix the reason for their behavior so that they work with XnR. For Windows, we introduced a heuristic that decides whether accessing code is legitimate or illegal. In the following, we explain both aspects in detail.

Code Reads on Linux.

All code read attempts of Linux programs that we observed during our evaluation were accessing the header of the ELF executable file or the header of an ELF shared library. This header is parsed by library functions that iterate over the loaded sections (PHDRs). In contemporary Linux ELF executables, this header resides in the `.text` segment, which is strictly speaking semantically wrong because the header (data) is not supposed to be executed. Should the program have a vulnerability, the ELF header unnecessarily resembles ROP gadgets.

For XnR, storing the header in an executable segment does no longer work. Reading the header in memory triggers an illegal access in XnR because it belongs to `.text`, which is marked executable.

The fact that the ELF header resides in the loaded `.text` segment is a result of file size optimizations by the linker stemming from an era that was not aware of the security implications of a header that is executable.

After modifying the standard Linux linker script that creates executables and shared libraries, the ELF headers reside in the read-only data section (`.rodata`) and can be accessed without triggering an XnR violation. As a side effect, this prevents the ELF headers from being executable.

Unaffected Execution.

To ensure that benign programs are not affected by XnR, we used 352 standard command line programs for Linux that are packed in the *busybox* [2] project. All 352 commands were executed successfully on an XnR-enabled Linux 3.13.7 without any illegal read detected by the XnR kernel subsystem. On Windows we performed a similar test using the 231 utilities provided by Cygwin64 [3], all of which succesfully executed. That is a very good indication that XnR does not affect the normal execution of programs.

Code Reads on Windows.

On Windows, we also found cases where a benign executable reads its own code. This happens, for example, when an application uses `GetModuleHandle` and `GetProcAddress` to dynamically obtain function addresses. Starting with Windows 7, common API functions can be accessed not only via the plain DLL name they are contained in, but also via a special API layering naming scheme beginning with "API-" [44]. For example `kernel32.dll` imports multiple functions from `api-ms-win-core-memory-l1-1-0.dll`. The prefix is stored as a static string inside the `.text` section of `ntdll` and not, as one would expect, inside `.data` or `.rodata`. Thus, any call to `GetModuleHandle` would lead to a false positive and termination of the application. A semantically correct solution would be, analogously to Linux, to place the offending data in the proper section. Since the system libraries are closed source we cannot recompile them. Instead, we introduce a heuristic to account for occuring code reads. A counter is increased for every code read and reduced for each first legitimate access to a code page. Should the counter reach a configurable threshold, the current code read is considered illegal. Thus a limited number of code reads are permitted, while an excessive scan of the .text section is detected and prevented.

This heuristic enables legacy applications to run with XnR protection. However, programs that are available in source can be forced to store all data in a separate region, similar to Linux programs with a modified linker script. Those specifically compiled Windows programs can benefit from the full security guarantees of XnR without needing to resort to a heuristic for compatibility. As part of our future work, we plan to revise and extend the support of XnR on Windows beyond the current prototype implementation.

4.2 Performance Evaluation

To evaluate the efficiency of XnR, we used the de facto standard SPEC CPU2006 integer benchmark suite. All benchmarks on Linux were performed on an Intel Core i7-3770 CPU running at 3.4 GHz with 4 GB of RAM. This particular CPU features four hardware cores with two symmetric hardware threads each (*HyperThreading*).

The run-time overhead of XnR is mainly introduced by the modified page fault handlers. Both implementations ensue that the number of pages that are mapped into an XnR-enabled process are at most n at any time. Usually, compilers optimize code layout so that the locality of code is high, i.e., functions that call each other or functions that resemble hot spots reside next to each other in the compiled code. For programs that heavily jump between different code positions, the probability is higher that they access different memory pages. If their jumping is constrained to at most n pages, our XnR virtually has no impact as no page faults will be triggered during execution. However, if a program continuously accesses more than n pages, this results in constant eviction of pages from the process' address space and results in a performance degradation compared to a stock Linux kernel without XnR checks.

Since the locality of code and the window size n influence performance, we also conducted benchmarks for different values of n. Figure 4 depicts the performance depending on the window size $n = 2$ pages, $n = 4$, $n = 6$ and $n = 8$ pages.

Even for a small window size of only $n = 2$ pages, the average overhead is a moderate 2.2%. These good performance figures make choosing the right n fairly easy as small values of n allow for high security but remain decent performance. Suppose the security parameter and window size was chosen to be $n = 2$. Then for the attacker to be successful, she needs to know the two addresses of the two active pages. Even in that unlikely case, she would be left with only 8 kB of code to use for an attack.

The low overhead slightly contradicts the traditional assumption that a large working set is necessary for good performance. However, by making pages *non-present*, we do not actually reduce the working set size. In contrast to an unmapped page, removing the *present* bit leaves the con-

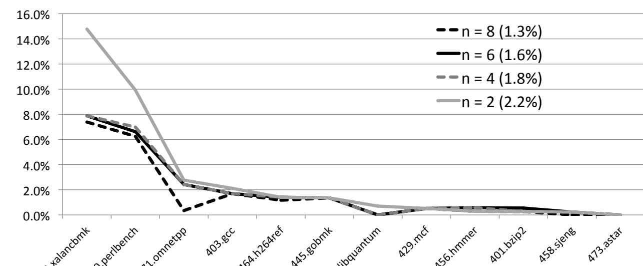

Figure 4: SPECint2006 benchmark suite for Linux showing the performance for each of the 12 benchmarks (and averages) dependent on the parameter n.

tent of a memory page intact and also does not touch any caches. That means that in contrast to a reduced working set size, our XnR solution only re-enables the *present* bit, which causes a TLB miss but at the same time profits from a cache that is still filled. Hence, the overhead is mainly due to the CPU switching to kernel mode after the hardware page fault and switching back to user mode after enabling the *present* bit.

Micro Benchmarks.

As our XnR solution only needs to protect executable pages, the performance overhead is more distinct for applications that heavily jump between many code areas. Examples of such are the *Perl* interpreter (benchmark `400.perlbench`) and XSLT transformers (benchmark `483.xalancbmk`). The other extreme are programs that heavily operate on data like compression algorithms such as *BZip2*. In this case study, we demonstrate the different memory access behavior by picking prominent examples of their respective groups: *Perl* for code-intense programs and *BZip2* for data-intense programs.

As can be seen from Table 1, the different access patterns (code vs. data) already result in different access times for pages faults on an unmodified stock Linux. When run on a Linux without our XnR solution enabled, the difference is even more distinct as the heavy code execution of *Perl* results in a slightly higher performance penalty. Figure 5 shows a distribution of page fault durations for the two distinct programs *BZip2* and *Perl*. The graph shows that the sliding window results in a higher amount of page faults for the code-intense program *Perl*. In contrast, for the compression algorithm *BZip2* only a few more page faults are generated for the relatively steady code access. However, the additional checks on every access result in a slightly broader distribution of page fault times. The total overhead in terms of run-time for the same input is almost negligible for *BZip2* (only 0.3% for $n = 4$), whereas the total runtime given the identical input to *Perl* was increased by 7.0%.

Table 1: Microbenchmarks for data-intensive example (`bzip2`) and code-intensive example (`perl`). Sliding window set to $n = 4$.

Program	∅ Page Fault		Page Faults / s	
	Stock Kernel	With XnR	Stock Kernel	With XnR
Bzip2	9.1 μs	12.9 μs	307	401
Perl	5.4 μs	8.3 μs	108	512

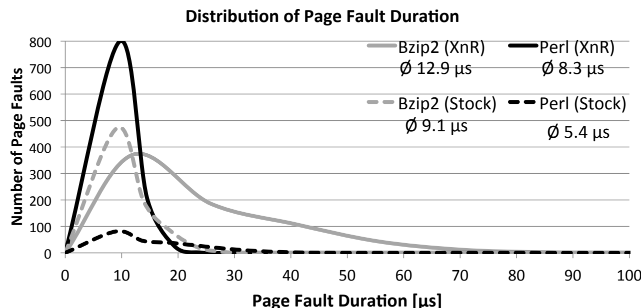

Figure 5: Distribution of the duration of page faults that have been triggered over a period of 2s.

Windows Performance.

We also evaluated the performance impact of our proof-of-concept XnR implementation for Windows. The benchmarks were conducted in a virtual machine with 2 GB of RAM running on one core of an Intel Core i7-950 with 3.06GHz. Our tests show that the current implementation introduces more overhead compared to the Linux version, mostly due to the additional processing required in order to make decisions on whether or not to allow an access before the actual page fault handler is executed (see Figure 6 for details).

On Windows, we used window sizes of $n = 2$, $n = 4$ and $n = 8$ with a selection of SPEC tools to illustrate the effects of varying window sizes. While the performance is not as good as on Linux, the results show that we can also deploy XnR on a proprietary Windows system. With an appropriate window size of $n = 4$, the overhead is on average 3.4%, but decreasing the window to only $n = 2$ results in 8.4% overhead.

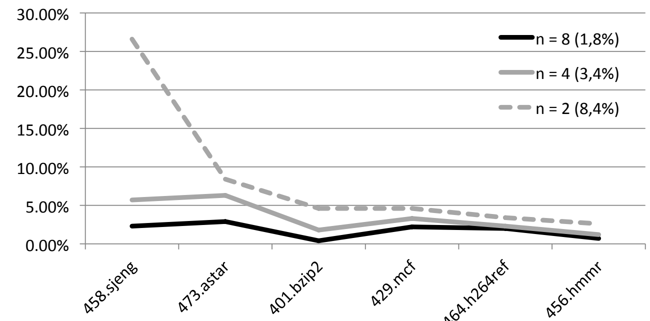

Figure 6: SPECint2006 benchmark suite for Windows showing the performance for each of the 6 benchmarks dependent on the parameter n.

5. RELATED WORK

The broad adoption of non-executable memory (write-xor-execute ($W \oplus X$) for short) successfully mitigates code injection attacks. However, this defense primitive gave rise to a form of attack that reuses existing code by intelligently stitching small code fractions, so-called *gadgets*, together in order to execute arbitrary code [10, 12, 14, 27, 36, 38]. These gadgets are well selected so that they end in an instruction that transfers control to the next gadget, e.g. a `return` instruction which pops its target off the stack. Hence the name *return-oriented programming* (ROP [38]).

ROP Detection and Prevention.

Over the last few years, code reuse attacks and their mitigation has been an ongoing cat and mouse game. Some of the code reuse mitigation techniques address the problem at its roots by preventing buffer overruns [5, 6] or by confining the control flow to the destined control-flow graph [4]. Several efficient approaches for the enforcement of control-flow integrity (CFI) were proposed over the years (e.g., [50, 51]), but recent research results suggest that an adversary can easily bypass them [22].

Other mitigation techniques make it hard for the adversary to guess or brute-force addresses that are necessary for successful execution of malicious code. Besides the more general control flow integrity [4, 16], there are two common

types of defenses proposed in the literature that were designed to prevent or mitigate those type of attacks. (1) Address Space Layout Randomization (ASLR [35]), which was designed to make the addresses of functions and gadgets unpredictable, and (2) ROP gadget elimination, which programmatically replaces basic blocks of a program with semantically equivalent, but different instructions [25, 31, 32, 47].

The first category of defense (1) used finer and finer randomization to counteract the fact that a single leaked pointer may revert low-entropy randomization [39]. The different randomizations solutions have different approaches, resulting in different entropy, and are applied at different stages of the software lifecycle. While Bhatkar et al. [8] makes source code self-randomizing, Kil et al. [26] modify the binaries on disk. Another approach is to randomize the processes at load-time, which gives a different randomization for every process start [18, 47]. Hiser et al. [24] even randomize running programs dynamically, similarly to the execution of a virtual machine.

The other category (2) randomizes instructions and registers within a basic block to mitigate return-oriented programming attacks [32]. However, often those solutions cannot prevent return-into-libc attacks, which have been shown to be Turing-complete [45], since all functions remain at their original position.

Over the last few years, several approaches to detect ROP exploits at runtime such as *kBouncer* [33], *ROPecker* [15], and *ROPGuard* [19] were also proposed. These methods employ different heuristics to detect suspicious branch sequences hinting at an ongoing ROP exploit. Recent research results suggest that such approaches can be bypassed by an attacker since she can construct gadget chains that bypass all proposed heuristics [13, 17, 23].

JIT-ROP.

Recently, Snow et al. [41] showed that given a memory disclosure vulnerability, it is possible to assemble ROP gadgets on-demand without knowing the layout or randomization of a process. They explore the address space of the vulnerable process step-by-step by following the control flow from an arbitrary start position. After they have discovered enough ROP gadgets, they compile the payload so that it incorporates the actual current addresses that were discovered on site.

Snow et al. also proposed potential mitigations of their own attack. However, the proposed solutions are either very specific to their heap spraying exploitation or are as general and slow as frequent re-randomization of a whole process. Note that the latter is potentially bypassable if the attack takes place between two randomization phases.

Blind-ROP.

Bittau et al. [9] presented *blind ROP attacks* against server processes with unknown binaries. Their attack consist of three stages: First, they apply a stack reading attack to bypass stack canaries and find the return address. Second, they infer gadget positions by repeatedly altering the return address on the stack and observing the server's behaviour when executing the code at the altered return address. They end this stage once they found enough gadgets to perform a write()-syscall to transmit the executable memory over the network. Lastly, they scan the dumped binary for gadgets to launch a common ROP attack.

While XnR successfully prevents the full attack (because the third stage cannot read the executable memory), we have to note that the second stage could still execute sucessfully. However, the authors need a large number of request to find even a single gadget, which makes finding enough gadgets for a full ROP chain likely impractical.

Re-Randomization.

The only work that implemented and benchmarked re-randomization was presented by Giuffrida et al. [21]. Based on the LLVM framework, their Minix microkernel can re-randomize itself every x seconds. However, this procedure has a significant run-time overhead of 10% for a randomization every $x = 5$ seconds or even 50% overhead when applied every second. Note that an attacker can potentially abuse this long time window to perform a JIT-ROP attack.

To the best of our knowledge, in this paper we present the first solution that prevents the general problem of memory disclosure attacks conceptually, which makes it the first solution that is secure against the new just-in-time ROP by Snow et al.

Related Techniques.

Our implementation shares some properties with *Shadow Walker* [43]. The common idea is to distinguish between code and data accesses. However, Shadow Walker's goal is to hide code when it is read as data in order to evade detection. Additionally, they use very processor-specific caching manipulation to achieve their goal while we only rely on standard paging mechanisms.

Shadow Walker exploits the fact that the x86 CPUs use two distinct caches, the ITLB (Instruction TLB) and DTLB (Data TLB) to perform lookups of pages. While these are usually transparent to the OS, it is possible to return different page translations depending on whether an access occurred due to a data or code read and thus fill the caches with inconsistent values. These are cached until evicted by more recent entries. However, we chose not to use this technique as it is heavily dependent on the availability of split TLBs and the actual implementation of the caching algorithms. While these can be considered fit for our purpose on basically all modern x86 CPUs, we wanted to show a general solution that can be applied to architectures with one TLB or even no TLB at all.

6. DISCUSSION

Our prototype implementations for Windows and Linux show the general feasibility of our approach. However, similar to $W \oplus X$, to become widespread and usable without restrictions, they need compiler support in the future. Before $W \oplus X$ was introduced, executing the stack was normal[1] and a paradigm shift was needed to mark the stack not executable by default. We have observed that both Windows and Linux programs were linked so that the resulting executable files contained data stored in the code segment. This is not just semantically wrong, but also hindered our XnR solution. While it was easy to fix open source programs by re-compiling them with a modified linker script, the Windows core DLLs remain closed source. With a change of

[1]E.g., trampolines need an executable stack

the default binutils linker script, XnR could become default. The same is true if Microsoft changed their default linker to put data in read-only data sections of the EXE and DLL files, rather than code.

It is noteworthy that XnR prevents memory disclosures, but the protected program might still suffer from a pointer leakage vulnerability. This is possible because the address a pointer represents is data even though it might point to code. Such a leaked pointer (e.g., through a malformed `printf`) may reveal function addresses. Therefore, the most simple form of code reuse attacks—return-to-libc—would potentially be possible. In a broader sense, the attacker could use functions as very coarse grained gadgets. However, return-to-libc attacks can be detected by the callee. The callee needs to check whether the last address on the stack is its own address. Usually, it must be any other address but the callee's address because it represents the return address of the caller who used a regular `call` instruction. A `call` pushes the return location to the stack, whereas a `return` pops the address of the targeted function (*callee*) off the stack. This prevention technique is a very simple form of control-flow integrity checks [4, 16]. It could either be patched to the prologue of every function by means of binary rewriting or require compiler support.

Our XnR solution might hinder debuggers. On a Linux machine, the `gdb` debugger reads bytes from the code section and even overwrites bytes in the code section to place breakpoints. This is prevented by XnR. However, we assume that a developer using a debugger can also control the XnR kernel featue in order to allow debugging. On Windows, debugging is done from a remote process using `ReadProcessMemory` (internally `NtReadVirtualMemory`) APIs and therefore is not impaired by XnR.

7. CONCLUSION

We presented a novel technique that detects and prevents the root cause of memory disclosure exploits. This thwarts a whole attack vector that is based on those types of exploits, most prominently attacks that exploit the disclosure vulnerability to disassemble the address space with the intent of finding usable code for a code reuse attack.

Even though there is no hardware support yet for our "Execute-no-Read" (XnR) primitive, our software emulation shows benign programs are not affected by this new primitive while XnR successfully prevents disclosure exploits before they could take effect. Our performance benchmarks demonstrate that even the software implementation has a rather low overhead of just 2.2% (Linux) and 3.4% (Windows). To stimulate future research on this topic, we are making our XnR implementation available in support of open science.

Acknowledgments.

The research was supported in part by the DFG Research Training Group GRK 1817/1. We also thank the anonymous reviewers and our shepherd Zhiqiang Lin for their valuable insights and comments.

8. REFERENCES

[1] *ARM1136JF-S and ARM1136J-S Technical Reference Manual Revision: r1p5, section 6.5.2*. ARM Limited.

[2] BusyBox: The Swiss Army Knife of Embedded Linux. http://www.busybox.net/.

[3] Cygwin - Posix API and tool collection for Windows. https://www.cygwin.com/.

[4] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2005), ACM, pp. 340–353.

[5] Abadi, M., Budiu, M., Erlingsson, U., Necula, G. C., and Vrable, M. XFI: Software Guards for System Address Spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[6] Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M. Preventing Memory Error Exploits with WIT. *IEEE Symposium on Security and Privacy* (2008).

[7] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine 49*, 14 (1996).

[8] Bhatkar, S., Sekar, R., and DuVarney, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium* (2005), USENIX Association.

[9] Bittau, A., Belay, A., Mashtizadeh, A., MaziÃĺres, D., and Boneh, D. Hacking Blind. In *IEEE Symposium on Security and Privacy* (2014).

[10] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented Programming: A New Class of Code-reuse Attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2011).

[11] blexim. Basic Integer Overflows. *Phrack Magazine 60*, 10 (2002).

[12] Buchanan, E., Roemer, R., Shacham, H., and Savage, S. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security (CCS)* (2008).

[13] Carlini, N., and Wagner, D. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security Symposium* (2014).

[14] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented Programming Without Returns. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[15] Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[16] Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., and Sadeghi, A.-R. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Symposium on Network and Distributed System Security (NDSS)* (2012).

[17] Davi, L., Lehmann, D., Sadeghi, A.-R., and Monrose, F. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security Symposium* (2014).

[18] Davi, L. V., Dmitrienko, A., Nürnberger, S., and Sadeghi, A.-R. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *8th ACM SIGSAC symposium on Information, computer and communications security (ACM ASIACCS 2013)* (2013), ACM, pp. 299–310.

[19] Fratric, I. Runtime Prevention of Return-Oriented Programming Attacks. `http://ropguard.googlecode.com/svn-history/r2/trunk/doc/ropguard.pdf`.

[20] gera. Advances in Format String Exploitation. *Phrack Magazine 59*, 12 (2002).

[21] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 40–40.

[22] Goktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy* (2014).

[23] Göktas, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *USENIX Security Symposium* (2014).

[24] Hiser, J. D., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. ILR: Where'd My Gadgets Go? In *IEEE Symposium on Security and Privacy* (2012).

[25] Jajodia, S., Ghosh, A. K., Subrahmanian, V. S., Swarup, V., Wang, C., and Wang, X. S., Eds. *Moving Target Defense II - Application of Game Theory and Adversarial Modeling*, vol. 100 of *Advances in Information Security*. Springer, 2013.

[26] Kil, C., Jun, J., Bookholt, C., Xu, J., and Ning, P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference (ACSAC)* (2006).

[27] Krahmer, S. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique. `http://users.suse.com/~krahmer/no-nx.pdf`, 2005.

[28] Microsoft. Kernel patch protection for x64-based operating systems. `http://technet.microsoft.com/en-us/library/cc759759(v=ws.10).aspx`.

[29] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[30] MITRE. Common weakness enumeration. `http://cwe.mitre.org/top25/`, November 2012.

[31] Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., and Kirda, E. G-Free: defeating return-oriented programming through gadget-less binaries. In *ACSAC'10, Annual Computer Security Applications Conference* (Dec. 2010).

[32] Pappas, V., Polychronakis, M., and Keromytis, A. D. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *IEEE Symposium on Security and Privacy* (2012).

[33] Pappas, V., Polychronakis, M., and Keromytis, A. D. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security Symposium* (2013).

[34] PaX Team. `http://pax.grsecurity.net/`.

[35] PaX Team. PaX Address Space Layout Randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`.

[36] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security 15*, 1 (Mar. 2012).

[37] Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium* (2010).

[38] Shacham, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[39] Shacham, H., jin Goh, E., Modadugu, N., Pfaff, B., and Boneh, D. On the Effectiveness of Address-space Randomization. In *ACM Conference on Computer and Communications Security (CCS)* (2004).

[40] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy* (2013).

[41] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy* (2013).

[42] Solar Designer. "return-to-libc" attack. Bugtraq, 1997.

[43] Sparks, S., and Butler, J. ShadowWalker: Raising the Bar for Rootkit detection. In *Black Hat Japan* (2005).

[44] Sun, B. Kernel patch protection for x64-based operating systems. `http://blogs.mcafee.com/mcafee-labs/windows-7-kernel-api-refactoring`.

[45] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., and Ning, P. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection* (2011), Springer-Verlag.

[46] Van der Veen, V., Cavallaro, L., Bos, H., et al. Memory errors: the past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012, pp. 86–106.

[47] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[48] Xu, H., and Chapin, S. Address-space layout randomization using code islands. In *Journal of Computer Security* (2009), IOS Press, pp. 331–362.

[49] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *IEEE Symposium on Security and Privacy* (2009).

[50] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy* (2013).

[51] Zhang, M., and Sekar, R. Control flow integrity for cots binaries. In *USENIX Security Symposium* (2013).