Protecting Commodity Operating Systems through Strong Kernel Isolation

Vasileios P. Kemerlis

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2015

ABSTRACT

Protecting Commodity Operating Systems through Strong Kernel Isolation

Vasileios P. Kemerlis

Today's operating systems are large, complex, and plagued with vulnerabilities that allow perpetrators to exploit them for profit. The constant rise in the number of software weaknesses, coupled with the sophistication of modern adversaries, make the need for effective and adaptive defenses more critical than ever. In this dissertation, we develop a set of novel protection mechanisms, and introduce new concepts and techniques to secure *commodity* operating systems against attacks that exploit vulnerabilities in kernel code.

Modern OSes opt for a *shared* process/kernel model to minimize the overhead of operations that cross protection domains. However, this design choice provides a unique vantage point to local attackers, as it allows them to control—both in terms of permissions and contents—part of the memory that is accessible by the kernel, easily circumventing protections like kernel-space ASLR and `W^X`. Attacks that leverage the weak separation between user and kernel space, characterized as *return-to-user* (ret2usr) attacks, have been the de facto kernel exploitation technique in virtually every major OS, while they are not limited to the x86 platform, but have also targeted ARM and others.

Given the multi-OS and cross-architecture nature of ret2usr threats, we propose kGuard: a kernel protection mechanism, realized as a cross-platform compiler extension, which can safeguard *any* 32- or 64-bit OS kernel from ret2usr attacks. kGuard enforces *strong* address space segregation by instrumenting exploitable control transfers with dynamic Control-Flow Assertions (CFAs). CFAs, a new *confinement* (inline monitoring) concept that we introduce, act as guards that prevent the unconstrained transition of privileged execution paths to user space. To thwart attacks against itself, kGuard also incorporates two novel *code diversification* techniques: code inflation and CFA motion. Both countermeasures randomize the location of the inline guards, creating a moving target for an attacker that tries to pinpoint their exact placement to evade kGuard. Evaluation results indicate that kGuard provides comprehensive ret2usr protection with negligible overhead (~1%).

Furthermore, we expose a set of additional kernel design practices that trade stronger isolation for performance, all of which can be harnessed to *deconstruct* kernel isolation. To demonstrate the significance of the problem, we introduce a new kernel exploitation technique, dubbed *return-to-direct-mapped memory* (ret2dir), which relies on inherent properties of the memory management (sub)system of modern OSes to bypass *every* ret2usr defense to date. To illustrate the effectiveness of ret2dir, we outline a principled methodology for constructing *reliable* exploits against hardened targets. We further apply it on real-world kernel exploits for x86, x86-64, and ARM Linux, transforming them into ret2dir-equivalents that bypass deployed ret2usr protections, like Intel SMEP and ARM PXN.

Finally, we introduce the concept of *eXclusive Page Frame Ownership* (XPFO): a memory management approach that prevents the implicit sharing of page frames among user processes and the kernel, ensuring that user-controlled content can no longer be injected into kernel space using ret2dir. We built XPFO on Linux and implemented a set of optimizations, related to TLB handling and page frame content sanitization, to minimize its performance penalty. Evaluation results show that our proposed defense offers effective protection against ret2dir attacks with low runtime overhead (<3%).

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The security of a computer system can only be as good as that of the underlying Operating System (OS) kernel [120]. By compromising the kernel, attackers can escape isolation and confinement mechanisms, bypass access control and policy enforcement, and elevate privileges. Nevertheless, until recently, attackers focused mostly on the exploitation of vulnerabilities in server and client applications, which often run with administrative rights, as they are (for the most part) less complex to analyze and easier to compromise [201, 4].

During the past few years, however, the kernel has become an equally attractive target. Continuing the increasing trend of the previous years, in 2013 there were 355 reported kernel vulnerabilities according to the National Vulnerability Database (NVD) [155], 140 more than in 2012, as shown in Figure 1.1. Admittedly, the exploitation of user-level software has become much harder, as recent versions of popular OSes come with numerous protections and exploit mitigations [7]: the principle of least privilege [185] is better enforced in user accounts and system services [134, 177, 176, 216], compilers offer more protections against common software flaws [55, 146, 160], and highly targeted applications, such as browsers and document viewers, have started to employ sandboxing [149, 154, 21, 19, 20, 136].

On the other hand, the kernel has a huge codebase and an attack surface that keeps increasing due to the constant addition of new features [123]. Indicatively, the size of the Linux kernel in terms of lines of code has more than doubled, from 6.6 MLOC in v2.6.11 to 16.9 MLOC in v3.10 [53]. Naturally, instead of putting significant effort to exploit applications fortified with numerous protections and sandboxes, attackers often turn

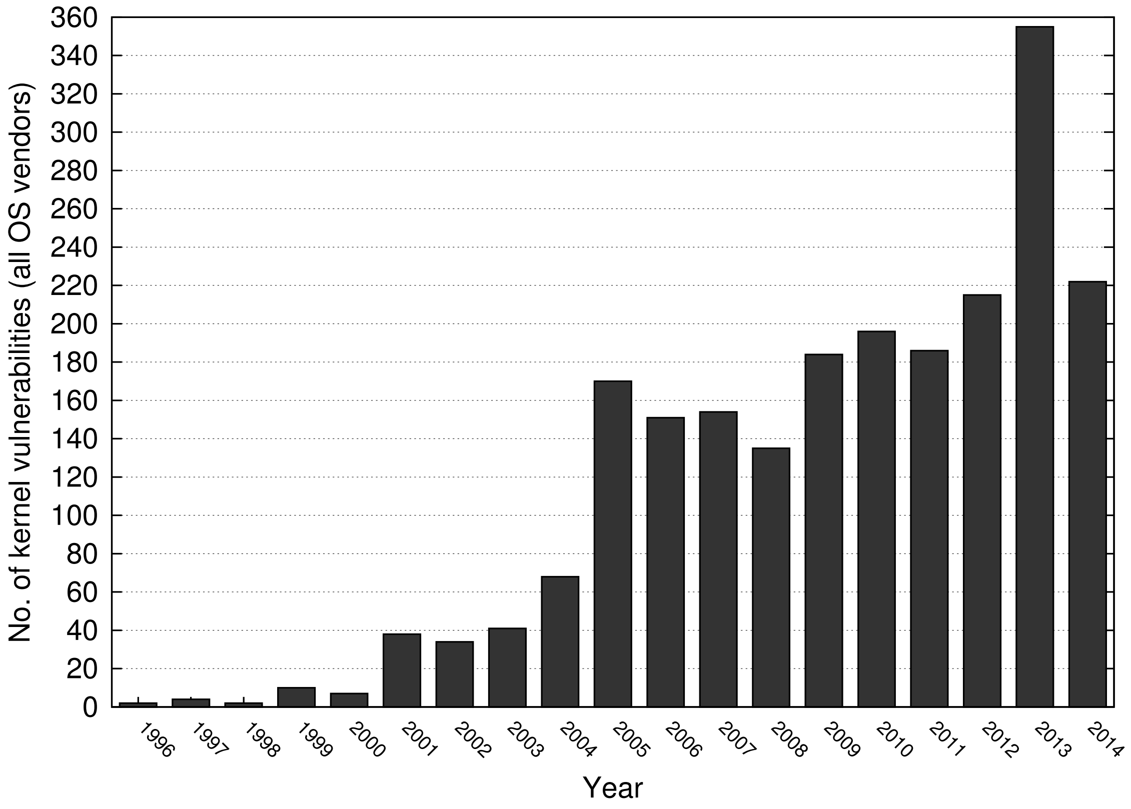


Figure 1.1: Kernel vulnerabilities (per-year) according to NVD [155].

their attention to the kernel [211]. For instance, in recent exploits against Chrome and Adobe Reader, after successfully gaining code execution, the attackers exploited kernel vulnerabilities to break out of the respective sandboxed processes [157, 98].

Opportunities for kernel exploitation are abundant. As an example consider the Linux kernel, which has been plagued by common software flaws, such as stack and heap buffer overflows [40, 29, 43], NULL pointer and pointer arithmetic errors [26, 22], use-after-free and format string bugs [44, 42], memory disclosure vulnerabilities [35, 28], integer overflows [32, 22], signedness errors [41, 33], race conditions [31, 23], as well as missing authorization checks and argument sanitization vulnerabilities [38, 34, 39, 37].

The exploitation of these bugs is particularly effective, despite the existence of kernel protection mechanisms (*e.g.,* kernel-space ASLR [66] and `W^X` [208, 127]), due to the *weak separation between user and kernel space.* Although user programs cannot access directly kernel code or data, the opposite is not true, as the kernel is mapped inside the address space of each process for performance reasons. This design allows an attacker to execute code in privileged mode, or tamper-with critical kernel data structures, by exploiting a kernel vulnerability and redirecting the control or data flow of the kernel to code or data in user space. Attacks of this kind, known as *return-to-user (ret2usr)* [115], affect all major OSes, like Linux [24, 27], Windows [191], and the BSDs [30, 190, 193], while they are not limited to the x86/x86-64 architecture [36], but also impact ARM [70], DEC Alpha [76], and PowerPC [179].

1.1 Hypothesis

Empirical evidence [68, 78, 79, 77, 73, 71, 75, 72, 75, 72, 69, 76, 74, 158, 70] suggests that the execution model imposed by a virtual memory layout that weakly separates the kernel from user processes makes kernel exploitation a fundamentally different craft from the exploitation of userland software [201, 173]. To that end, we hypothesize that the security posture of modern OSes can be improved by employing a synergy of defenses and exploit prevention techniques, which guarantee the *strong isolation* between user and kernel space.

1.2 Thesis Statement

This thesis argues that compiler-assisted *control-flow confinement* and *code diversification*, coupled with a physical memory management approach that enforces the *explicit* use of page frames by different security contexts (user vs. kernel), can efficiently and effectively protect the OS kernel against attacks that leverage the weak segregation of address spaces.

1.3 Contributions

1. We present the design and implementation of kGuard: a compiler plugin that protects commodity OS kernels from ret2usr attacks. kGuard operates by instrumenting exploitable control flow transfers with dynamic Control-Flow Assertions (CFAs). CFAs, a new *confinement* (inline monitoring) concept that we introduced, act as "guards" that prevent (at runtime) the unconstrained transition of privileged execution paths to user space. Our approach does not entail additional software (*e.g.,* hypervisor or VMM) or special hardware.

2. We introduce two novel *code diversification* techniques: *(a.)* code inflation and *(b.)* CFA motion. Both of them randomize the location of CFA-protected branches, during compilation and at run time, creating a moving target for an attacker that tries to pinpoint the exact placement of the fine-grained inline guards and bypass them to evade kGuard.

3. We implement kGuard as a GCC extension, which is *publicly* available. Its maintenance cost is low and can successfully compile functional x86/x86-64 Linux and BSD kernels. More importantly, it can be easily combined with other compiler-based protection mechanisms.

4. We assess the effectiveness of kGuard using real privilege escalation attacks against 32- and 64-bit Linux kernels. In all cases, kGuard was able to successfully detect and prevent the respective exploitation attempt.

5. We evaluate the performance of kGuard using a set of macro- and micro-benchmarks. Our technique incurs minimal runtime overhead on both x86 and x86-64 Linux. Specifically, we show negligible impact on real-life applications, and moderate overhead on system call and I/O latency.

6. We expose a fundamental *design weakness* in the memory management (sub)system of modern OSes, by introducing the concept of *return-to-direct-mapped memory (ret2dir)* attacks. ret2dir attacks exploit the kernel's direct-mapped RAM region(s) to bypass existing ret2usr protection (*i.e.,* SMEP, SMAP, PXN, PAN, KERNEXEC, UDEREF).

7. We introduce a *detailed methodology* for mounting reliable ret2dir attacks against x86, x86-64, AArch32, and AArch64 Linux systems, along with two techniques for forcing user-space exploit payloads to "emerge" within the kernel's direct-mapped physical memory area and accurately pinpointing their location.

8. We experimentally evaluate the effectiveness of ret2dir attacks using a set of nine (eight real-world and one artificial) exploits against different Linux kernel configurations and protection mechanisms. In all cases, our transformed exploits bypass the deployed ret2usr protections.

9. We present the design, implementation, and evaluation of an *exclusive page frame ownership* (XPFO) scheme for the Linux kernel, which protects against ret2dir attacks with negligible (in most cases) runtime overhead.

10. We make *publicly* available: *(i)* our prototype implementation of XPFO, and *(ii)* the complete set of our ret2dir exploits. The latter can be used to educate the community regarding ret2dir attacks and evaluate the effectiveness of future ret2dir defenses.

1.4 Dissertation Roadmap

Chapter 2 provides background information on virtual memory organization and kernel exploitation, and surveys the (state-of-the-art) kernel protection mechanisms in modern OSes. Chapter 3 covers the design, implementation, and evaluation of kGuard. Chapter 4 introduces the concept of ret2dir attacks along with a detailed methodology for mounting such attacks in a plethora of different systems and configurations. Chapter 5 presents the design, implementation, and evaluation of XPFO, which provides protection against attacks that (ab)use the implicit sharing of physical memory between user processes and the kernel. Finally, this dissertation concludes in Chapter 6.

Chapter 2

Background and Related Work

2.1 Virtual Memory Organization

Private virtual address spaces are the standard OS abstraction for separating and confining user processes. Programs, however, cannot run in absolute isolation as they need to *interact* with the OS to perform I/O, acquire or share memory, request device access, communicate with other processes, *etc.* Designs for safely combining different protection domains (*e.g.,* the OS kernel and user programs) range between two extremes. At one extreme, experimental OSes, like Microsoft's Singularity [101], follow a single address space design, by placing the kernel and all user processes into a single, privileged address space and establishing boundaries using software isolation. At the other extreme, OSes that build upon microkernel-based designs [2], such as MINIX [99] and L4 [129], confine user process and kernel components into separate, hardware-enforced address spaces.

Commodity OSes, such as Linux and Linux-derived spin-offs (Android [94], Chrome OS [95], Firefox OS [153], Tizen [132], Sailfish OS [105]), the BSDs (FreeBSD [203], NetBSD [205], OpenBSD [206]), Oracle Solaris [163], and Microsoft Windows [145], adopt a more coarse-grained variant of the latter design approach, and divide the virtual address space into two parts; *kernel* and *user* space. The first part is assigned to kernel code and data, kernel modules, and dynamic kernel memory, while the second to user processes (*e.g.,* program code and data, heap and stack memory, shared libraries).

In most platforms, the separation between the two parts (*i.e.,* the user/kernel split) is enforced by a combination of two hardware features: *(i)* a set of CPU modes (protection rings) [106, 186]; and *(ii)* a memory management unit (MMU). The x86/x86-64 architecture supports four basic protection rings, with the kernel running in the most privileged one (ring 0) and user applications in the least privileged (ring 3). Similarly, PowerPC and MIPS have two basic CPU modes, SPARC has three, and ARM seven.[1] The MMU, programmed using privileged, special-purpose instructions, implements hierarchical memory protection and ensures that memory assigned to a certain ring is not accessible by code executing in *less* privileged rings. The net effect of the above is the complete isolation of kernel space from user programs (*i.e.,* code running in user mode cannot access the kernel directly).

Although some platforms support fully separate kernel and user address spaces, like Oracle Solaris on UltraSPARC [139], contemporary OSes running on popular CPUs (*e.g.,* x86 and ARM), employ a *shared* layout, in which the kernel is mapped inside the address space of every user process. For example, in the x86 and 32-bit ARM (AArch32) architectures, the Linux kernel is typically mapped to the upper 1GB part of the virtual address space, a split also known as "3G/1G" [45, 117].[2] In x86-64 and 64-bit ARM (AArch64) the Linux kernel is placed in the upper *canonical half* ([`0xFFFF800000000000` : $2^{64} - 1$] in x86-64 [119] and [`0xFFFFFF8000000000` : $2^{64} - 1$] in AArch64 [138]). Similar splits are used in BSD [141], Oracle Solaris (x86) [139], and Microsoft Windows [150, 148]. This design minimizes the overhead of crossing protection domains and facilitates fast user-kernel interactions: when servicing a system call, or handling an exception, the kernel is running within the *context* of a preempted process; flushing the TLB is not necessary [152], while the kernel can access user space *directly* to read user data or write the result of a system call.

[1]Modern CPUs support additional modes (rings): x86/x86-64 offers two more, colloquially known as ring -1 (hardware-assisted virtualization) and -2 (system management mode). SPARC has up to four additional modes (reset, error, debug, hyper-privileged), MIPS two (debug, supervisor), and PowerPC one (hypervisor).

[2]Linux also supports 2G/2G and 1G/3G splits. A patch for a 4G/4G split in x86 exists [152], but it was never included in the mainline kernel for performance reasons (it requires a full TLB flush per system call).

2.2 Kernel Exploitation

Despite the fact that both kernel code and userland software are plagued by common types of vulnerabilities, such as stack and heap buffer overflows [43, 40, 29], `NULL` pointer and pointer arithmetic errors [26], memory disclosure vulnerabilities [35, 28], use-after-free and format string bugs [44, 42], signedness errors [41, 33], integer overflows [32, 22], race conditions [31, 23], as well as missing authorization checks and poor argument sanitization bugs [38, 34, 39, 37] (interested readers are referred to the studies of Chen *et al.* [17], Chou *et al.* [18], and Palix *et al.* [166], for a detailed analysis of kernel bugs in Linux and OpenBSD), the execution model imposed by the shared virtual memory layout, between the kernel and user programs, makes kernel exploitation (in commodity OSes) a fundamentally different craft from the exploitation of userland software.

In particular, the shared address space provides a unique *vantage point* to local attackers (*i.e.,* attackers with the ability to execute programs on the OS), as it enables them to control—both in terms of contents and permissions—part of the memory accessible by the kernel [201]. Simply put, attackers can execute shellcode with kernel rights by hijacking a (privileged) kernel control path and redirecting it to user space, effectively invalidating the protection(s) offered by standard defenses, like kernel-space ASLR [66] and `W^X` [127, 208]. Attacks of this kind, known as *return-to-user* (`ret2usr`), can be traced back to the early 1970's, as demonstrated by the PDP-10 "address wraparound" fault [175]. Over the past decade, however, ret2usr has been promoted to the de facto kernel exploitation technique in modern OSes [218, 4, 108, 68, 67].

In a ret2usr attack, kernel data is overwritten with user-space addresses, usually after the exploitation of memory corruption bugs in kernel code [173]. Attackers primarily aim for control data, such as return addresses [192], dispatch tables [78, 68], and function pointers [79, 77, 76, 73], because they directly facilitate arbitrary code execution [200]. Pointers to critical data structures stored in the kernel's heap [71] or the global data section [78] are also common targets, since they allow attackers to tamper with critical data contained in these structures by mapping fake copies in user space [75, 72, 71].

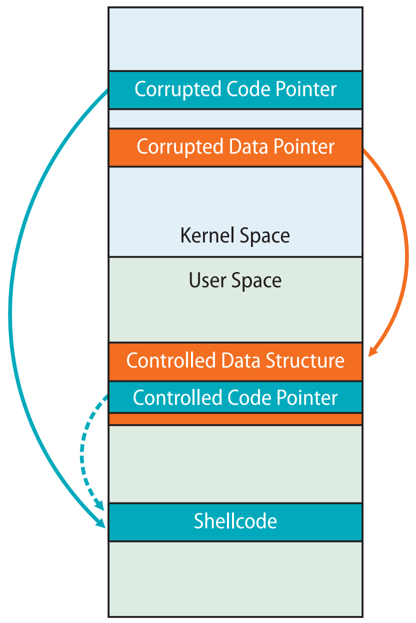


Figure 2.1: Operation of ret2usr attacks. Kernel code or data pointers are hijacked and redirected to controlled code or data in user space (tampered-with data structures may further contain pointers to code).

Typically, the targeted data structures contain function pointers or data that affect the control flow of the kernel, so as to diverge execution to arbitrary locations. The culmination of all ret2usr attacks, as illustrated in Figure 2.1, is the following: *the control/data flow of the kernel is hijacked and redirected to user space code/data* [115].

Most ret2usr exploits use a multi-stage shellcode, with a first stage that lies in user space and "glues" together kernel functions (*i.e.,* the second stage) to perform privilege escalation. Technically, ret2usr expects the kernel to run within the context of a process controlled by the attacker for exploitation to be reliable. However, kernel bugs have been identified and exploited in interrupt service routines [151]. In such cases, where the kernel is either running in *interrupt context* or in a process context beyond the attacker's control [184, 69], the respective shellcode has to be injected in kernel space or constructed using code gadgets from the kernel's executable segments using Return-/Jump-Oriented Programming (ROP/JOP) [16, 100, 195]. The latter approach is gaining popularity in real-world exploits, due to the increased adoption of kernel hardening techniques [137, 49, 220, 127, 209, 208].

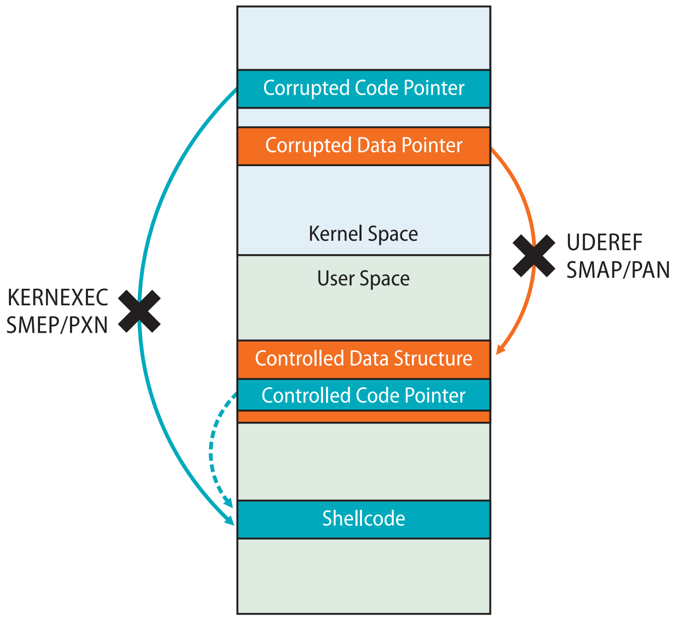


Figure 2.2: ret2usr protection mechanisms. KERNEXEC/UDEREF, SMEP/SMAP, and PXN/PAN prevent arbitrary kernel-to-user control transfers and pointer dereferences.

Interestingly, software bugs that are only a source of instability in user programs, like `NULL` pointer dereferences, can have more dire effects when found in an OS kernel that *weakly* (asymmetrically) separates kernel from user space. Brad Spengler [198] demonstrated how a `NULL` pointer dereference bug, triggered by invoking a system call with specially-crafted parameters, can be (ab)used to perform a ret2usr attack.

Earlier, it was generally thought that such flaws could only be used to perform denial-of-service (DoS) attacks [65], but Spengler's exploit showed how mapping asymmetrically-separated code segments (*i.e.,* kernel code and data are inaccessible from code running in user mode, but the kernel has complete and unrestricted access to the whole address space, which includes user code and data), with different rights, inside the same scope can be exploited to execute arbitrary user code with kernel privilege. A detailed analysis of the kernel bug exploited by Spengler, along with a step-by-step walk-through of a `NULL` function pointer vulnerability [25] that affected all Linux kernel versions released between May 2001 and August 2009 (v2.4.4/v2.6.0 – v2.4.37/v2.6.30.4), can be found in Appendix A.

2.3 Kernel Protection

Return-to-user attacks are yet another incarnation of the confused deputy problem [97]. Due to the weak virtual address space separation, a user cheats the kernel (deputy) to misuse its *ambient authority* and execute (or access) arbitrary, non-kernel code (or data) with elevated privilege. Given the multi-architecture [76, 179] and multi-OS [67, 193, 191, 190, 30, 58] nature of the problem, several defense mechanisms exist for it. In the remainder of this section, we discuss ret2usr defenses that are available in modern OSes and considered standard practice in kernel protection, with the help of Figure 2.2.

2.3.1 `mmap`

The first ret2usr mitigation strategy adopted by Linux [61, 180], {Free, Open}BSD [204], and Microsoft Windows [144], is based on restricting the ability to memory-map the first pages of the virtual address space (typically 4KB – 64KB). This is achieved by modifying the `mmap` system call (or `VirtualAlloc` in Windows [147]) to apply the respective restrictions and preventing binaries from requesting low address mappings.

Unfortunately, this approach has several limitations. First and foremost, it does not solve the actual problem, which is the weak separation of spaces. Disallowing access to lower logical addresses is a protection scheme only against exploits that rely on `NULL` pointer bugs [200, 179], as it does not defend against exploits where the control, or data, flow is hijacked and redirected to memory pages above the forbidden region (*e.g.,* by nullifying one or two bytes of a kernel pointer, or overwriting a branch target with an arbitrary value) [78, 74]. Second, if an attacker bypasses the restriction imposed by `mmap`, she can still orchestrate a ret2usr attack [207]. Third, it breaks compatibility with applications that demand access to low logical addresses [8, 217, 64, 81].

2.3.2 `KERNEXEC` and `UDEREF`

KERNEXEC and UDEREF are two features of the PaX [168] Linux hardening patch set, which prevent control transfers and pointer dereferences from kernel to user space.

In x86, KERNEXEC and UDEREF rely on memory segmentation[3] to map kernel space into a 1GB segment (assuming a 3G/1G split) that returns a memory fault whenever privileged code tries to dereference pointers to, or fetch instructions from, non-kernel addresses [169].

In x86-64, due to the lack of segmentation support, UDEREF/amd64 [170] remaps user space memory into a different, non-executable (shadow) area when execution enters the kernel (and restores it on exit), to prevent user-space pointer dereferences. Additionally, the shadow area is marked as non-executable to disallow control-flow transfers to user code. As the overhead of remapping memory is significant, an alternative for x86-64 systems is to enable KERNEXEC/amd64 [171], which has much lower overhead, but offers protection against only control-flow hijacking attacks. KERNEXEC/amd64 is implemented as a GCC plugin that instruments kernel code with small snippets that confine exploitable control transfers (*i.e.,* indirect `call` and `ret` instructions) to the upper canonical half of the virtual address space. Overall, KERNEXEC and UDEREF provide comprehensive protection against ret2usr attacks, but they are platform-specific (available only in Linux and mainly in x86/x86-64), and require kernel patching. Yet, ret2usr attacks have been demonstrated in non-x86 Linux architectures, including DEC Alpha [76] and PowerPC [179]. Recently, both features were ported to the ARM architecture [199], but the respective patches added support only for AArch32.

2.3.3 `SMEP/PXN` and `SMAP/PAN`

Supervisor Mode Execute Protection (SMEP) [87] and Supervisor Mode Access Prevention (SMAP) [54] are two recent features of Intel processors that facilitate stronger (symmetric) address space separation (latest Linux kernels support both features [220, 49]). Once enabled, by setting `CR4.SMEP` and `CR4.SMAP`, they result into a page fault whenever an instruction fetch (or data access) is attempted from a linear address whose entry in the page table has the `User/Supervisor` bit asserted (*i.e.,* marked as user mode page). SMEP provides analogous protection to KERNEXEC, without relying on segmentation or code instrumentation, whereas SMAP operates similarly to UDEREF, in absence of the performance penalty of page table manipulation upon kernel entry and exit [152].

[3]In x86, UDEREF restricts `SS`, `DS`, and `ES`. `CS` is taken care by KERNEXEC.

However, both hardware features are vendor-specific—AMD and other x86-compatible CPU vendors have not yet announced similar extensions—and do not protect legacy systems. Specifically, SMEP/SMAP is available only on Intel Ivy Bridge/Haswell (and later) CPUs, and support for it was added in Linux kernel v3.5/v3.7 [220, 49], OpenBSD v5.3 [161], FreeBSD v9 (SMEP only) [9], and Microsoft Windows v8 (SMEP only) [202]. NetBSD and Oracle Solaris have yet to add support for both SMEP and SMAP. Likewise, ARM introduced Privileged Execute-Never (PXN) [130] and Privileged Access-Never (PAN) [13], two features that operate similarly to SMEP and SMAP.

2.4 Other Work

2.4.1 Control-flow Confinement

Program Shepherding [118], Strata [188], and Control-Flow Integrity (CFI) [1], employ dynamic binary instrumentation and binary rewriting for retrofitting control-flow confinement capabilities into unmodified binaries. Specifically, they rewrite programs so that branch targets are given labels, and every indirect branch instruction is prepended with a check, which ensures that the target's label is in accordance with a precomputed control-flow graph (CFG). Regardless of the large performance overhead that these approaches typically incur—Program Shepherding exhibits ∼100% overhead on the SPEC benchmark, while CFI has an average overhead of 16% (21% when a shadow stack is used), and a maximum of 45%, on the same test suite—CFI-based techniques are not effective against ret2usr attacks.

The integrity of a label-based CFI mechanism is guaranteed as long as the attacker cannot overwrite the code of the protected binary or execute user-provided data. However, during a ret2usr attack, the attacker completely controls user space memory, both in terms of contents and rights (see Section 2.2), and thereby can subvert CFI by prepending user-provided shellcode with the respective label. As an example consider Listing A.1 and assume that the attacker has managed to overwrite the function pointer `sendpage` with a user space address. CFI will prepend the instruction that invokes `sendpage` with an inline check that fetches a label ID (placed right before the first instruction of every function that `sendpage` can point to) and compares it with the allowed label ID(s). If the two labels match, the control transfer will be authorized. As the attacker controls the contents and rights of the memory that `sendpage` points at in user space, she can prepend her code with the label ID that will authorize the control transfer.

More importantly, coarse-grained CFI schemes that rely on an imprecise CGF (*e.g.,* CCFIR [221], binCFI [222], kBouncer [167]) have recently shown to be vulnerable to code-reuse attacks [92, 15, 57, 93], and, as Petroni and Hicks [174] observed, computing in advance a precise CFG for a modern kernel is a nontrivial task, due to the rich control structure and the several levels of interrupt handling and concurrency of kernel code.

2.4.2 Hypervisor-based Protection

Garfinkel and Rosenblum proposed Livewire [83], which was one of the first systems to use a virtual machine monitor (VMM) for implementing invariant-based kernel protection. Similarly, Grizzard used a VMM for monitoring kernel execution and validating control flow [96]. For LMBench, Grizzard reported an average overhead of 30%, and a maximum of 74%, atop VMM's performance penalty. SecVisor [194] is a tiny hypervisor that ensures the integrity of commodity OS kernels. SecVisor relies on physical memory virtualization for protecting against code injection attacks and kernel rootkits, by allowing only approved code to execute in kernel mode and ensuring that such code cannot be modified. However, it requires a modern CPU that supports hardware virtualization, as well as kernel patching to add the respective hypercalls that authorize module loading. NICKLE [181] offers similar guarantees, without requiring any OS modification, by relying on an innovative memory shadowing scheme and real-time kernel code authentication via VMM introspection. Petroni and Hicks proposed state-based CFI (SBCFI) [174], which reports violations of the kernel's control flow due to the presence of rootkits. Similarly, Lares [172] and HookSafe [215] protect kernel hooks (including function pointers) from being manipulated by kernel malware.

Chapter 3

kGuard

3.1 Overview

It is our belief that compiler-assisted *control-flow confinement* and *code diversification* can effectively, and efficiently, protect commodity OS kernels against ret2usr attacks. In support of this claim, in this chapter, we present the design, implementation, and evaluation of kGuard: a cross-platform (*i.e.,* multi-OS and many-architecture), compiler-based solution that enforces strong address space segregation, without relying on special hardware features [169, 87, 137] or custom hypervisors [194, 181].

kGuard instruments, at compile time, exploitable control transfers with dynamic *control-flow assertions* (CFAs) that, at runtime, prevent the unconstrained transition of privileged control paths to user space. The injected CFAs perform a small runtime check to verify that the target address of every (privileged) indirect branch instruction is always in kernel space, as shown in Figure 3.1. If the assertion is true, execution continues normally, while if it fails because of a violation, execution is transferred to a handler that was inserted during compilation. The default handler appends a warning message to the kernel log and halts the system. We choose to coerce assertion failures into a kernel fail-stop to prevent unsafe conditions, such as leaving the OS into an inconsistent state (*e.g.,* by aborting an in-flight kernel thread that holds locks or other resources). In Section 3.5, we discuss how kGuard can be extended to support custom handlers for facilitating forensic analyses, error virtualization [196], selective confinement, and protection against persistent attacks.

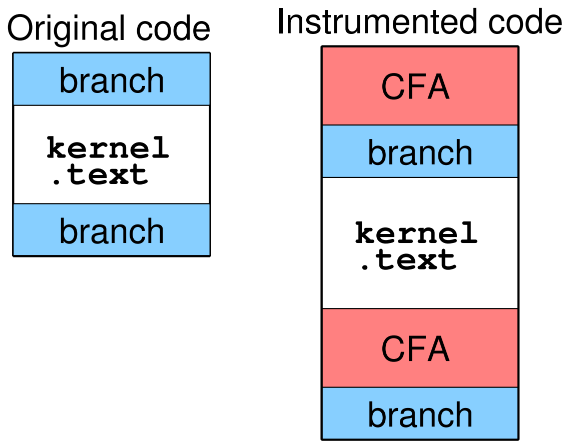


Figure 3.1: CFA-based control-flow confinement. The inserted CFAs perform a small run-time check to verify that the target address of every (privileged) indirect branch instruction is always in kernel space.

After compiling a kernel with kGuard, its execution is *confined* to the privileged address space segment (*e.g.,* addresses higher than `0xC0000000` in x86 Linux and {Free, Net}BSD; see Section 2.1). However, note that kGuard does not rely on any mapping restriction; the previously-disallowed low virtual memory addresses can be dispensed to processes, lifting the compatibility issues with various applications [8, 81, 217, 64]. Furthermore, the inserted CFAs cannot be bypassed using `mmap` tricks [207], nor can they be circumvented by elaborate exploits that jump to memory pages above the forbidden low memory region [78, 74]. More importantly, the kernel can still read and write user memory, so its functionality remains unaffected.

3.1.1 Threat Model

We ascertain an adversary that is able to completely overwrite, partially corrupt (*e.g.,* zero out only certain bytes), or nullify control data stored inside the address space of the kernel. Note that overwriting certain data (*i.e.,* code pointers) with arbitrary values, differs significantly from overwriting *arbitrary kernel memory*. kGuard does not deal with such an adversary. In addition, we assume that the attacker can tamper-with whole data structures (*e.g.,* by mangling data pointers; see Figure 2.1), which in turn may contain control data.

Our technique is straightforward and guarantees that ret2usr attacks are prevented. However, it is not a panacea that protects the kernel from all types control-flow hijacking attacks. For instance, kGuard does not address direct code-injection in kernel space, nor does it thwart code-reuse attacks that utilize ROP/JOP [100, 16]. Nevertheless, note the following. First and foremost, our approach is orthogonal to many solutions that do protect against such threats [126, 6, 194, 169, 107, 55]. Second, the unique nature of address space sharing casts many protection schemes, for the aforementioned problems, as ineffective. As an example, consider again the case of ROP/JOP or code-injection in the kernel setting. No matter what anti-ROP [1, 126, 159], ASLR [66], or `W^X` [127, 208] techniques have been utilized, the attacker can still execute arbitrary code, as long as there is no strict user/-kernel separation, by mapping her code to user space and transferring control to it (*e.g.,* after hijacking a privileged control path).

Finally, in order to protect kGuard from being subverted, we utilize two lightweight diversification techniques for the `.text` segment of the kernel/modules, which can also mitigate kernel-level attacks that use code "gadgets" in a ROP/JOP fashion (see Section 3.2.2.2). Nevertheless, the aim of kGuard is not to provide strict control-flow integrity for the kernel, but rather protect against attacks that leverage the weak segregation of address spaces.

3.2 Design

In this section, we discuss the fundamental aspects of kGuard using examples based on x86 Linux. However, note that kGuard is by no means restricted to 32-bit systems or Linux; it can protect *any* OS kernel that suffers from ret2usr attacks, running both on 32- and 64-bit CPUs. kGuard "guards" indirect control transfers from exploitation. In the x86 instruction set architecture (ISA), such control transfers are performed using the `call`/`jmp` instruction with a register or memory operand, and the `ret` instruction, which always takes an implicit memory operand (*i.e.,* the saved return address). kGuard injects CFAs in all such cases to *assert* that the branch target is located in kernel space.

```
81 fb 00 00 00 c0 ;             cmp   $0xc0000000,%ebx
73 05             ;             jae   call_lbl
bb f1 f8 5a c0    ;             mov   $0xc05af8f1,%ebx
ff d3             ; call_lbl:   call  *%ebx
```

Listing 3.1: `CFA`$_\texttt{R}$ applied on an indirect `call` in x86 Linux (*drivers/cpufreq/cpufreq.c*). The `call` instruction is instrumented with 3 additional instructions (`cmp`, `jae`, `mov`).

```
register void *target_address;
...
if (target_address < 0xC0000000)
    target_address = &violation_handler;
call *target_address;
```

Listing 3.2: `CFA`$_\texttt{R}$ in C-like code (x86).

Specifically, kGuard uses two different CFA types, namely `CFA`$_\texttt{R}$ and `CFA`$_\texttt{M}$, depending on whether the instruction that we want to confine uses a register or memory operand. Listing 3.1 shows a `CFA`$_\texttt{R}$ example. The code is from the `show()` routine of the `cpufreq` driver. kGuard instruments the indirect `call` (`call *%ebx`) with 3 additional instructions. First, the `cmp` instruction compares the `%ebx` register with the lowest kernel address `0xC0000000`.[1] If the assertion is true, the control transfer is *authorized* by jumping to the `call` instruction. Otherwise, the `mov` instruction loads the address of the violation handler (`0xC05AF8F1`; `kguard_panic()`) into the branch register and proceeds to execute `call`, which will result into invoking our violation handler. In C-like code, this is equivalent to injecting the statements shown in Listing 3.2.

Similarly, a `CFA`$_\texttt{M}$ confines indirect branches that use memory operands. Listing 3.3 illustrates how kGuard instruments an indirect control transfer in `sock_sendpage()`. The indirect `call` (`call *0x50(%ebx)`) is prepended by a sequence of 10 instructions that perform two distinct assertions. `CFA`$_\texttt{M}$ not only asserts that the branch target is within

[1]The same is true for x86 FreeBSD/NetBSD, whereas for x86-64 Linux the check should be with address `0xFFFF800000000000`. OpenBSD maps the kernel to the upper 512MB of the virtual address space, and hence, its base address in x86 CPUs is `0xD0000000`.

```
57                    ;             push %edi
8d 7b 50              ;             lea  0x50(%ebx),%edi
81 ff 00 00 00 c0     ;             cmp  $0xc0000000,%edi
73 06                 ;             jae  kmem_lbl
5f                    ;             pop  %edi
e8 99 e2 0f 00        ;             call 0xc05af8f1
5f                    ; kmem_lbl:   pop  %edi
81 7b 50 00 00 00 c0  ;             cmpl $0xc0000000,0x50(%ebx)
73 05                 ;             jae  call_lbl
c7 43 50 f1 f8 5a c0  ;             movl $0xc05af8f1,0x50(%ebx)
ff 53 50              ; call_lbl:   call *0x50(%ebx)
```

Listing 3.3: `CFA`$_M$ applied on an indirect `call` in x86 Linux (*net/socket.c*). The `call` instruction is prepended with a sequence of 10 instructions (`push` – `movl`).

```
register void *target_address_loc;
         void *target_address;
...
target_address_loc = &target_addr;
if (target_address_loc < 0xC0000000)
    call violation_handler;
if (target_address < 0xC0000000)
    target_address = &violation_handler;
call *target_address;
```

Listing 3.4: `CFA`$_M$ in C-like code (x86).

the kernel address space, but also *ensures that the memory address where the branch target is loaded from is also in kernel space*. The latter is necessary for protecting against cases where the attacker has managed to hijack a data pointer to a structure that contains code pointers (see Figure 2.1). Listing 3.4 illustrates how this can be represented in C-like code. In order to perform this dual check, we first need to spill one of the registers in use, unless the basic block where the CFA is injected has spare registers, so that we can use it as a temporary variable (`%edi` in our example). The address of the memory location that stores the branch target (`%ebx + 0x50`), is dynamically resolved via an

```
81 7b 50 00 00 00 c0 ;            cmpl $0xc0000000,0xc123beef
73 05                ;            jae  call_lbl
c7 43 50 f1 f8 5a c0 ;            movl $0xc05af8f1,0xc123beef
ff 53 50             ; call_lbl:  call *0xc123beef
```

Listing 3.5: Optimized `CFA`$_{\texttt{M}}$. The operand of the `call` instruction is loaded from the `.data` segment of the kernel.

arithmetic expression entailing registers and constant offsets. We load its effective address into `%edi` (`lea 0x50(%ebx),%edi`), and proceed to verify that it points in kernel space. If a violation is detected, the spilled register is restored and control is transferred to the runtime violation handler (`call 0xc05af8f1`). Otherwise, we proceed with restoring the spilled register and confine the indirect branch instruction similarly to the `CFA`$_{\texttt{R}}$ case.

3.2.1 Optimizations

In certain cases, we can statically determine if the address of the memory location that holds a branch target is always mapped in kernel space. Examples include a branch operand read from the kernel stack (assuming that the attacker has not seized control of the stack pointer), or taken from a global data structure mapped at a fixed address inside the `.data` or `.rodata` segment of the kernel. In such cases, the first assertion of a `CFA`$_{\texttt{M}}$ guard will always be true, since the branch operand is kernel memory. We optimize such scenarios by removing the redundant assertion, effectively reducing the size of the CFA to 3 instructions, as shown in Listing 3.5.

3.2.2 Mechanism Protection

CFAs, as presented thus far, provide reliable protection against ret2usr attacks, only if the attacker exploits a vulnerability that allows her to partially control a computed branch target. However, vulnerabilities that allow overwriting kernel code pointers with arbitrary values do exist [34]. When such flaws are present, ret2usr exploits could attempt to bypass kGuard. This section discusses how kGuard defends itself against such threats.

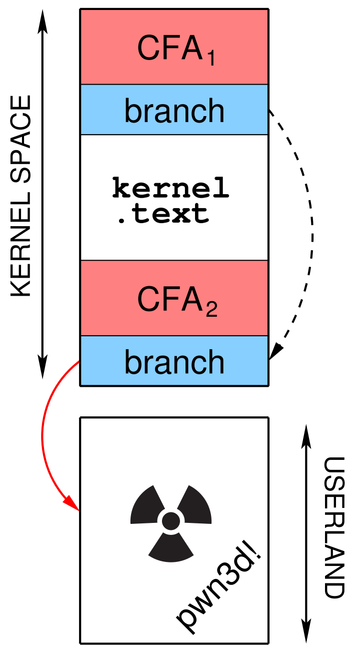


Figure 3.2: Subverting kGuard using bypass trampolines. `CFA`$_1$ succeeds since the address of the second branch (trampoline) is located in kernel space. `CFA`$_2$ is completely bypassed by jumping directly to the branch instruction.

3.2.2.1 Bypass Trampolines

To subvert kGuard, an attacker has to be able to determine the address of an indirect branch instruction inside the `.text` segment of the kernel/modules. Moreover, she should also be able to reliably control the value of its operand (branch target). We refer to that branch instruction as *bypass trampoline*. Note that in ISAs with overlapping variable-length instructions it is possible to find an embedded opcode sequence that translates directly to a branch in user space [100]. By overwriting the target of a protected branch instruction with the address of a bypass trampoline the attacker can successfully execute a jump to user space, as illustrated in Figure 3.2. `CFA`$_1$, corresponding to the initially exploited branch, will succeed, since the address of the trampoline remains inside the privileged memory segment, while `CFA`$_2$, which guards the bypass trampoline, is completely bypassed by jumping directly to the branch instruction.

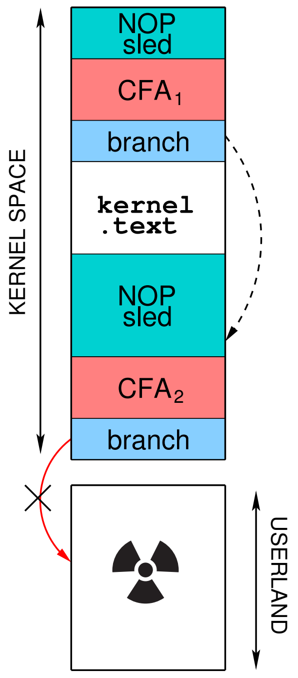


Figure 3.3: Code inflation reshapes the `.text` area of the kernel/modules by inserting `NOP` sleds of random length at the beginning of each CFA.

3.2.2.2 Code Diversification against Bypasses

kGuard implements two *code diversification* techniques that aid in thwarting attacks exploiting bypass trampolines.

① **Code Inflation.** This technique *reshapes* the `.text` area of the kernel/modules, as shown in Figure 3.3. We begin with randomizing the starting address of the `.text` segment. This is achieved by inserting a `NOP` sled of *random length* at its beginning, which effectively shifts all executable instructions by an arbitrary offset. Next, we continue with inserting `NOP` sleds of random length at the beginning of each CFA. The end result is that the location of every indirect control transfer instruction is randomized, forcing the attacker to guess the (exact) address of a confined branch to use as a bypass trampoline. The effects of the sleds are cumulative because each one pushes all instructions and `NOP` sleds following, further to higher memory addresses. The size of the initial sled is chosen by kGuard based on the target architecture. In Linux and {Free, Net}BSD the kernel space is at least 1GB, and hence, we can achieve $\geq$ 20-bit of entropy without notably consuming address space.

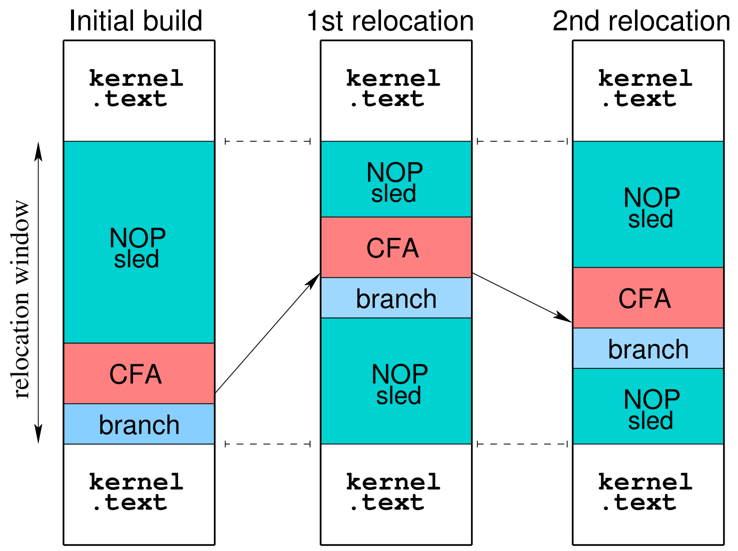


Figure 3.4: CFA motion. kGuard relocates each inline guard and protected branch, within a certain window, by routinely rewriting the `.text` segment of the kernel/modules.

The size of the per-CFA `NOP` sled is randomly selected from a user-configured range. By specifying the range, users can trade higher overhead (both in terms of space and speed), for a smaller probability that an attacker can reliably obtain the address of a bypass trampoline. An important assumption of the aforementioned technique is the secrecy of the kernel's image and symbols. If the attacker has access to the binary image of the protected kernel or is armed with a kernel-level memory disclosure vulnerability [72, 197], the probability of successfully guessing the address of a bypass trampoline increases. We posit that assigning safe file permissions to the kernel's image, modules, and debugging symbols is not a limiting factor.[2] In fact, this is considered standard practice in OS hardening, and is automatically enabled in PaX and similar patches, as well as the latest Ubuntu Linux releases. Also note that the kernel should harden access to the system message buffer (`dmesg`), in order to prevent the leakage of kernel addresses.[3]

[2]This can be trivially achieved by changing the permissions in the file system to disallow reads, from non-administrative users, in `/boot` and `/lib/modules` in Linux/FreeBSD, `/bsd` in OpenBSD, *etc.*

[3]In Linux this can be performed by asserting the `kptr_restrict` [183] sysctl option, which hides exposed kernel pointer addresses in `/proc` interfaces.

② **CFA Motion.** The basic idea behind this technique is the "continuous" relocation of the protected branch instructions and injected guards, by rewriting the `.text` segment of the kernel/modules. Figure 3.4 illustrates the concept. During compilation, kGuard emits information regarding each injected CFA, which can be used later for relocating the respective code snippets. Specifically, kGuard logs the exact location of the CFA inside `.text`, the type and size of the guard, the length of the prepended `NOP` sled, as well as the size of the branch instruction. Armed with that information, we can migrate every CFA and indirect branch instruction separately, by moving them to a random location inside the following window: `sizeof(nop_sled) + sizeof(cfa) + sizeof(branch)`.

Currently, our prototype implementation supports CFA motion during kernel bootstrap only. In Linux, we perform the relocation at the end of the `decompress_kernel()` routine [11], right before jumping to the main entry point of the (decompressed) ELF image. In the BSDs, we do it after the `boot` program has been executed and right before transferring control to the machine-dependent initialization routines (`mi_startup()` in FreeBSD; `main()` in {Net, Open}BSD) [142]. Lastly, note that CFA motion can be performed at runtime, on a live system, by trading runtime overhead for safety. In Section 3.6.1, we discuss how we can expand our current implementation, with moderate engineering effort, to support real-time CFA migration, effectively creating a moving target for the attacker [125].

To further protect against evasion, kGuard can be combined with techniques that secure kernel code against code-injection [128] and code-reuse attacks [1, 126, 159, 56]. That said, note that ret2usr violations are detected at runtime. Hence, one failed attempt is enough to identify the attacker and restrict her capabilities. In Section 3.5.2, we discuss how kGuard can deal with persistent threats.

3.3 Implementation

We implemented kGuard as a set of modifications to the pipeline of a C compiler. Specifically, we instrument the intermediate representation (IR) used during the translation process, in order to perform the CFA-based confinement discussed in Section 3.2. Our implementation consists of a plugin for the GNU Compiler Collection (GCC).

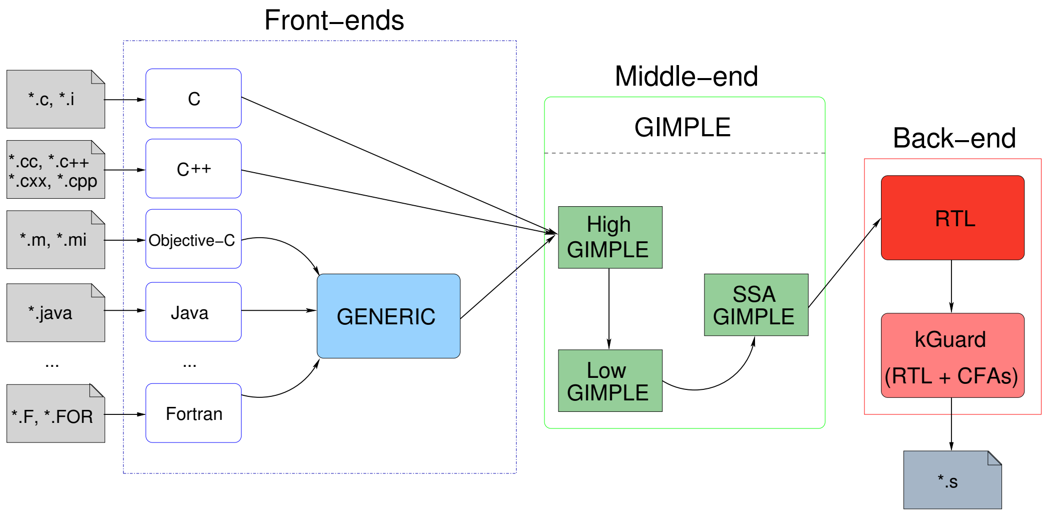


Figure 3.5: Architectural overview of GCC. The compilation process involves 3 distinct translators (frond-end, middle-end, back-end), and more than 250 optimization passes. kGuard is implemented as a back-end optimization pass.

Starting with v4.5.1, GCC has been re-designed for facilitating better isolation between its components, and allowing the use of plugins for dynamically adding features to translators without modifying them. Figure 3.5 illustrates the internal architecture of GCC. The compilation pipeline consists of 3 distinct components, namely the front-end, middle-end, and back-end, which transform the input into various IRs (*i.e.,* GENERIC, GIMPLE, and RTL). The kGuard plugin consists of ~1000 lines of code in C and builds into a position-independent (`PIC`) dynamic shared object that is loaded by GCC. Upon initializing kGuard, the plugin manager of GCC invokes `plugin_init()` (*i.e.,* the initialization callback assumed to be exported by every plugin), which parses the plugin arguments (if any) and registers `pass_ branchprot` as a new "optimization" pass.[4] Specifically, we chain our instrumentation callback, namely `branchprot_instrument()`, after the `vartrack` RTL optimization pass, by calling GCC's `register_callback()` function and requesting to hook with the pass manager (see Figure 3.5).

[4]Currently, kGuard accepts 3 parameters: `stub`, `nop`, and `log`. `stub` provides the name of the runtime violation handler, `nop` stores the maximum size of the random `NOP` sled inserted before each CFA, and `log` is used to define an instrumentation logfile for CFA motion.

The reasons for choosing to implement the instrumentation logic at the RTL level, and not as annotations to the GENERIC or GIMPLE IR, are mainly the following. First, by applying our assertions after most of the important optimizations have been performed, which may result into moving or transforming instructions, we guarantee that we instrument only relevant code. For instance, we do not inject CFAs for dead code or control-flow transfers that, due to optimization transformations like inline expansion, do not need to be confined. Second, we secure implicit control transfers that are exposed later in the translation (*e.g.,* after the High-GIMPLE IR has been "lowered"). Third, we tightly couple the CFAs with the corresponding unsafe control-flow transfers. This way, we protect the guards from being removed or shifted from the respective points of check, due to subsequent optimization passes (*e.g.,* code motion). For more information regarding the internals of RTL instrumentation, interested readers are referred to Appendix B.

3.4 Evaluation

In this section, we present the results from the evaluation of kGuard both in terms of performance and effectiveness. Our testbed consisted of a single host, equipped with two 2.66GHz quad-core Intel Xeon X5500 CPUs and 24GB of RAM, running Debian GNU/Linux v6 ("squeeze" with kernel v2.6.32). Note that while conducting our performance measurements, the host was idle with no other user processes running apart from the evaluation suite. Moreover, the results presented here are mean values, calculated after running 10 iterations of each experiment; the error bars correspond to 95% confidence intervals.

kGuard and the corresponding Linux kernels were compiled with GCC v4.5.1, and unless otherwise noted, we used Debian's default configuration that results into a complete build of the kernel, including all modules and device drivers. Finally, we configured kGuard to use a random `NOP` sled of maximum 32 (average 16) instructions. Note that we also measured the effect of various `NOP` sled sizes, which was insignificant for the range 0 – 32.

Vulnerability	Description	Impact	Exploit	
			x86	x86-64
CVE-2009-1897	NULL *function* pointer dereference in *drivers/net/tun.c* due to compiler optimization	2.6.30–2.6.30.1	✓	N/A
CVE-2009-2692	NULL *function* pointer dereference in *net/socket.c* due to improper initialization	2.6.0–2.6.30.4	✓	✓
CVE-2009-2908	NULL *data* pointer dereference in *fs/ecryptfs/inode.c* due to negative reference counting (function pointer affected via tampered-with data flow)	2.6.31	✓	✓
CVE-2009-3547	*data* pointer corruption in *fs/pipe.c* due to a use-after-free bug (function pointer under user control via tampered-with data structure)	$\leq$ 2.6.32-rc6	✓	✓
CVE-2010-2959	*function* pointer overwrite via integer overflow in *net/can/bcm.c*	2.6.{27.x, 32.x, 35.x}	✓	N/A
CVE-2010-4258	*function* pointer overwrite via arbitrary kernel memory nullification in *kernel/exit.c*	$\leq$ 2.6.36.2	✓	✓
EDB-15916	*function* pointer overwrite via a signedness error in the Phonet protocol (function pointer affected via tampered-with data structure)	2.6.34	✓	✓
CVE-2009-3234	ret2usr via kernel stack buffer overflow in *kernel/perf_counter.c* (return address is overwritten with a user space address)	2.6.31-rc1	✓	✓

✓: detected and prevented successfully, N/A: exploit unavailable

Table 3.1: kGuard effectiveness evaluation suite. We instrumented 10 x86/x86-64 vanilla Linux kernels, ranging from v2.6.18 to v2.6.34; kGuard successfully detected and prevented all the listed exploits.

	x86 kernel			x86-64 kernel		
	`call`	`jmp`	`ret`	`call`	`jmp`	`ret`
$\texttt{CFA}_{\texttt{M}}$	20767	1803		17740	1732	
$\texttt{CFA}_{\texttt{M}}$`(opt)`	2253	12	113053	1789	0	105895
$\texttt{CFA}_{\texttt{R}}$	6325	0		8780	0	
Total	**29345**	**1815**	**113053**	**28309**	**1732**	**105895**

Table 3.2: Number of indirect branches instrumented by kGuard (Linux kernel v2.6.32.39).

3.4.1 Effectiveness

The main goal of the effectiveness evaluation is to apply kGuard on commodity OSes, and determine whether it can detect and prevent real-life ret2usr attacks. Table 3.1 summarizes our test suite, which consisted of a collection of 8 exploits from the Exploit Database (EDB) [158] that cover a broad spectrum of different flaws, including direct `NULL` pointer dereferences, control hijacking via tampered-with data structures (data pointer corruption), function and data pointer overwrites, arbitrary kernel-memory (data) nullification, and ret2usr via kernel stack-smashing.

We instrumented 10 different vanilla Linux kernels, ranging from v2.6.18 up to v2.6.34, both in x86 and x86-64 architectures. Additionally, in this experiment, we used a home-grown violation handler for demonstrating the customization features of kGuard. Upon the detection of a ret2usr attack, the handler takes a snapshot of the memory that contains the user-provided code for analyzing the behavior of the offending process. Such a feature could be useful in a honeypot setup for performing malware analysis and studying new ret2usr exploitation vectors. All kernels were compiled with and without kGuard, and tested against the respective set of exploits. In every case, we were able to successfully detect and prevent the corresponding exploitation attempt. Also note that the tested exploits circumvented the page mapping restriction(s) that we discussed in Section 2.3.1 [207].

3.4.2 Translation Overhead

We first quantify the additional time needed to inspect the RTL IR and emit the CFAs (see Section 3.3). Specifically, we used the Unix `time` utility to measure the total build time when compiling Linux kernel v2.6.32.39 natively and with kGuard. On average, we observed a 0.3% increase on total build time on the x86 architecture, and 0.05% on x86-64. Moreover, the size of the kernel image/modules increased by 3.5%/0.43% on the x86, and 5.6%/0.56% on the x86-64.

Table 3.2 shows the number of exploitable branches instrumented by kGuard, categorized by architecture, and confinement and instruction type. As expected, `ret` instructions dominate the computed branches. Note that both in x86 and x86-64 scenarios, we were able to optimize approximately 10% of the total indirect calls via memory locations, using the optimization scheme presented in Section 3.2.1. Overall, the `drivers/` subsystem was the one with the most instrumentations, followed by `fs/`, `net/`, and `kernel/`. Additionally, a significant amount of instrumented branches was due to architecture-dependent code (`arch/`) and "inlined" C functions (`include/`).

3.4.3 Performance Overhead

The injected CFAs introduce runtime latency. We evaluated kGuard to quantify this overhead and establish a set of performance bounds for different types of system services. Moreover, we used the overhead imposed by PaX (*i.e.,* UDEREF [169, 170] and KERNEXEC [171]) as reference. Recall that on x86, PaX offers protection against ret2usr attacks by utilizing the segmentation unit for isolating the kernel from user space. In x86-64 CPUs, where segmentation is not supported by the hardware, it temporarily remaps user space into a different (shadow) location with non-execute permissions.

3.4.3.1 Macro-Benchmarks

We begin with a set of real-life applications that represent different workloads. In particular, we used a kernel build and two popular server applications: the Apache web server, which performs mainly I/O; and the MySQL RDBMS that is both I/O driven and CPU intensive.

We run all the respective tests over a vanilla Linux kernel (v2.6.32.39), and the same kernel patched with PaX and instrumented with kGuard.

First, we measured the time taken to build a vanilla Linux kernel (v2.6.32.39), using the Unix `time` utility. On the x86, the PaX-protected kernel incurs a 1.26% runtime overhead, while on the x86-64 the overhead is 2.89%. In contrast, kGuard ranges between 0.93% on x86-64, and 1.03% on x86. Next, we evaluated MySQL v5.1.49 using its own benchmark suite (`sql-bench`). The suite consists of four different tests, which assess the completion time of various DB operations, like table creation and modification, data selection and insertion, *etc.* On average, the slowdown of kGuard ranges from 0.85% (x86-64) to 0.93% (x86), while PaX lies between 1.16% (x86) and 2.67% (x86-64). Finally, we measured Apache's performance using its own utility `ab` and static HTML files of different size. We used Apache v2.2.16 and configured it to pre-fork all the worker processes (pre-forking is a standard multiprocessing module), in order to avoid high fluctuations in performance, due to Apache spawning extra processes for handling the incoming requests at the beginning of our experiments. We chose files with sizes of 1KB, 10KB, 100KB, and 1MB, and measured the average throughput in requests per second (req/sec). All other options were left to their default setting. The kernel patched with PaX incurs an average slowdown that ranges between 0.01% and 0.09% on x86, and 0.01% and 1.07% on x86-64. In antithesis, kGuard lies between 0.001% and 0.01%. Overall, our results indicate that in both x86 and x86-64 Linux the impact of kGuard on real-life applications is negligible (~1%).

3.4.3.2 Micro-Benchmarks

We used the LMbench [143] microbenchmark suite to measure the impact of kGuard on the performance of core kernel system calls and facilities. We focus on both latency and bandwidth. For the first, we measured the latency of entering the OS, by investigating the `NULL` system call (`syscall`) and the most frequently used I/O-related calls: `read`, `write`, `fstat`, `select`, `open`/`close`. Additionally, we measured the time needed to install a signal handler with `sigaction`, inter-process communication (IPC) latency with `socket` and `pipe`, and process creation latency with `fork`+{`exit, execve, /bin/sh`}.

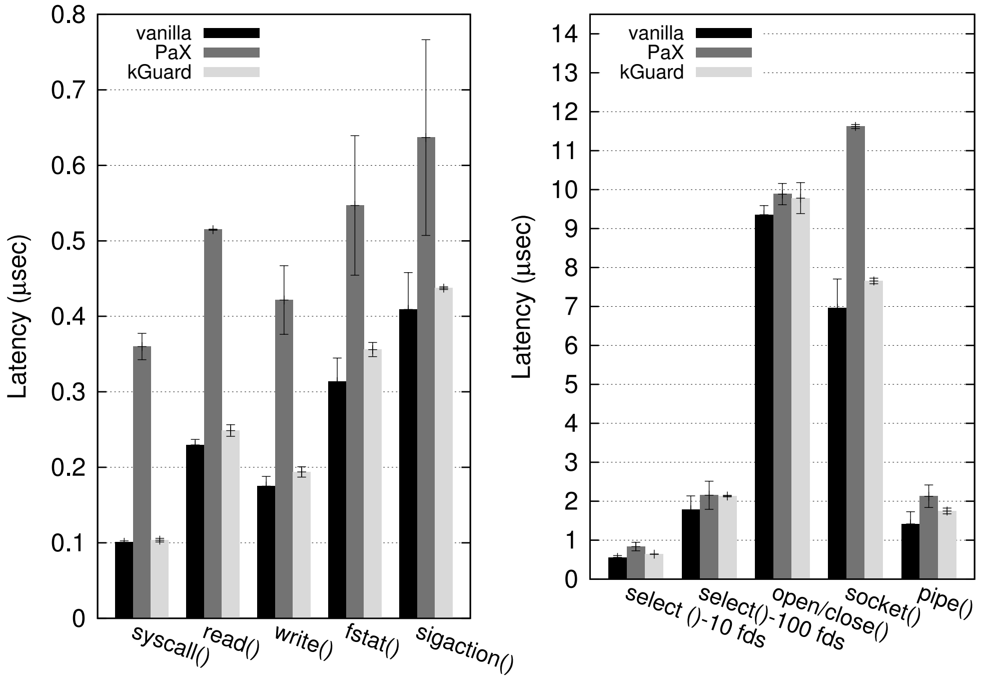


Figure 3.6: Latency overhead incurred by kGuard and PaX on essential system calls (x86 Linux).

Figure 3.6 and Figure 3.7 summarize the latency overhead of kGuard in contrast to the vanilla Linux kernel, and a kernel with the PaX patch applied and enabled. Note that the time is measured in microseconds (μsec). kGuard ranges from 2.7% to 23.5% in x86 (average 11.4%), and 2.9% to 19.1% in x86-64 (average 10.3%). In contrast, the PaX-protected kernel exhibits a latency ranging between 5.6% and 257% (average 84.5%) on the x86, whereas on x86-64, the latency overhead ranges between 19% and 531% (average 172.2%). Additionally, the overhead of kGuard for process creation (in both architectures) lies between 7.1% and 9.7%, whereas PaX ranges from 8.1% to 56.3%.

As far as bandwidth is concerned, we measured the degradation imposed by kGuard and PaX in the maximum achieved bandwidth of popular IPC facilities, such as sockets and pipes. Figure 3.8 shows our results (bandwidth is measured in MB/s). The slowdown of kGuard ranges between 3.2% – 10% on x86 (average 6%), and 5.25% – 9.27% on x86-64 (average 6.6%). PaX lies between 19.9% – 58.8% on x86 (average 37%), and 21.7% – 78% on x86-64 (average 42.8%).

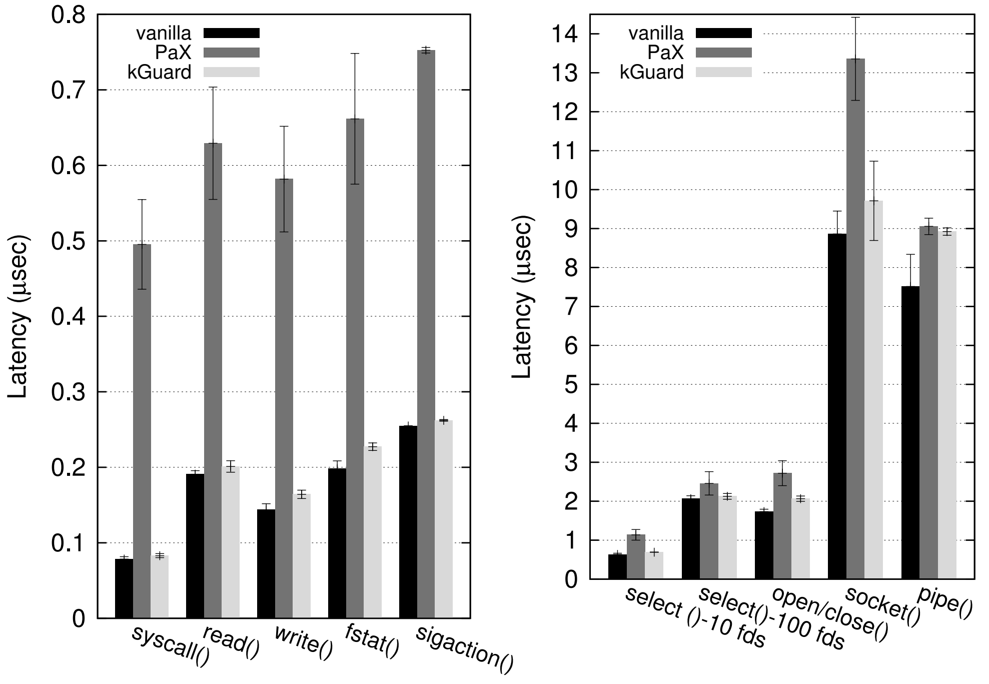


Figure 3.7: Latency overhead incurred by kGuard and PaX on essential system calls (x86-64 Linux).

Overall, kGuard exhibits lower overhead on x86-64, due to the fewer $\texttt{CFA}_\texttt{M}$ guards (see Table 3.2)—$\texttt{CFA}_\texttt{R}$ confinement can be performed with just 3 additional instructions, and hence incurs less runtime overhead; $\texttt{CFA}_\texttt{M}$ might need up to 10 instructions (*e.g.,* when we cannot optimize). However, the same is not true for PaX, as the lack of segmentation in x86-64 results in higher performance penalty due to frequent page table manipulations.

3.5 Discussion

3.5.1 Custom Violation Handlers

The default violation handler of kGuard appends a message to the system log and halts the OS. We coerce assertion violations into a kernel fail-stop to prevent brute-force attempts against our code diversification schemes (code inflation, CFA motion; see Section 3.2.2.2), and avoid leaving the OS in an inconsistent state (*e.g.,* by aborting an in-flight kernel thread that holds locks). Nonetheless, kGuard can be configured to use custom handlers.

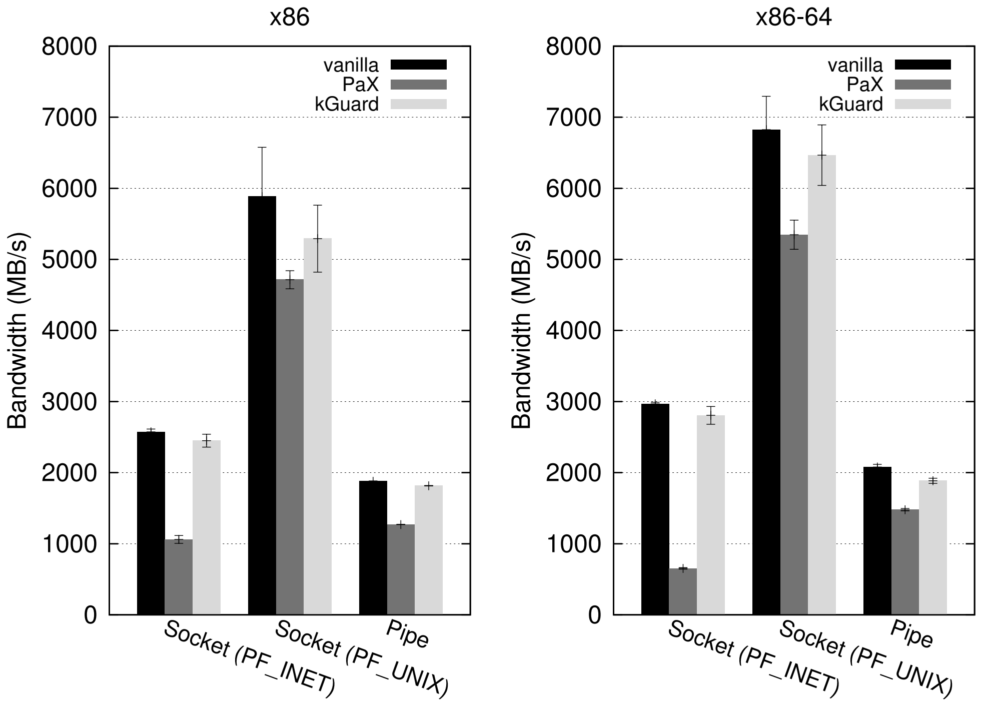


Figure 3.8: IPC bandwidth achieved by kGuard and PaX, using TCP (`PF_INET`), Unix sockets (`PF_UNIX`), and pipes.

Upon enabling this option, our confinement instrumentation becomes slightly different. Instead of overwriting the computed branch target with the address of the default handler, when a violation occurs, we push the *offending* target address to the stack and invoke the custom handler directly. In the case of a $\texttt{CFA}_\texttt{R}$ this means that the `mov` instruction (see Listing 3.1) is replaced by a `push` and `call`, as shown in Listing 3.6; $\texttt{CFA}_\texttt{M}$ guards are modified accordingly. This instrumentation increases slightly the size of our guards, but does not incur additional overhead, as the extra instruction is on the error path.

```
81 fb 00 00 00 c0 ;             cmp  $0xc0000000,%ebx
73 06             ;             jae  call_lbl
53                ;             push %ebx
e8 81 e6 0f 00    ;             call 0xc05af8f1
ff d3             ; call_lbl:   call *%ebx
```

Listing 3.6: Alternative $\texttt{CFA}_\texttt{R}$ applied on an indirect `call` in x86 Linux.

The custom violation handler has access to the location where the violation occurred (*i.e.,* the failed CFA), by reading the return address of the callee (pushed to the stack from `call`), as well as to the offending branch target (explicitly pushed to the stack by `push`). Armed with that information, one can customize kGuard's violation-handling policy and implement adaptive defense mechanisms [86], like selective confinement (*e.g.,* deal with VMware's I/O backdoor that needs to "violate" protection domains) and error virtualization [196], or even perform forensic analyses (*e.g.,* dump the exploit shellcode). The latter can be useful in honeypot setups for studying new ret2usr exploitation vectors.

3.5.2 Persistent Threats

The violation-handling policy of kGuard can be tailored to need, by leveraging its ability to support custom handlers. To that end, we developed a handler that actively responds to persistent threats (*e.g.,* users that try to brute-force the code diversification schemes of kGuard by repeatedly attempting the bypass presented in Section 3.2.2.1).

Once invoked, due to a violation, this handler performs the following. First, it checks the execution context of the kernel to identify if it runs inside a user-level process or an interrupt handler. If the violation occurred while executing an interrupt service routine, or if the current execution path is holding locks[5], then it fail-stops the kernel. Else, if the kernel is preemptible, it terminates all processes with the same `uid` with the offending process, and prevents the user from logging in again (*i.e.,* by disabling the respective account). Other possible approaches include inserting an exponentially-increasing delay to user logins (*i.e.,* make the brute-force attack slow and impractical), activate CFA motion, *etc.*

3.6 Future Work

3.6.1 Live CFA Motion

Currently, we investigate ways to apply the CFA motion technique (see Section 3.2.2.2), while a kernel is running and the OS is live. To that end, we have crafted an early prototype,

[5]In Linux, we can easily check if a kernel control path is holding locks by looking at the value of the `preempt_count` variable [135].

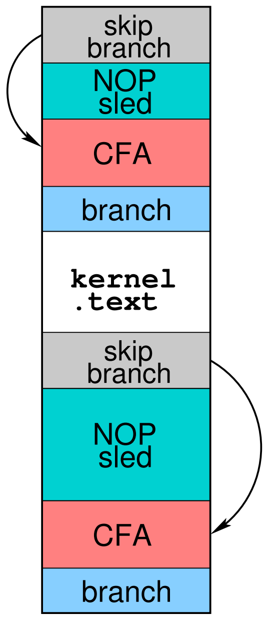


Figure 3.9: Low-latency code inflation inserts "skip branches" before every `NOP` sled to avoid executing the `NOP` instructions and directly transfer control to the CFA that follows.

which upon certain conditions, *freezes* the kernel and performs code (`.text`) rewriting. Thus far, we have achieved CFA relocation in a coarse-grained manner, by exploiting the `stop_machine` facility of the Linux kernel [5]. Specifically, we bring the system to pre-suspend state, preventing any kernel code from being invoked during the relocation phase. Possible events to initiate live CFA motion are the number of executed system calls or interrupts (*i.e.,* diversify the kernel every N invocation events), CFA violations, or external intrusion events (*e.g.,* a ret2usr violation alert generated from a different system inside the same security domain or "application community") [133].

However, our end goal is to be able to perform CFA motion in a more fine-grained, non-interruptible and efficient manner, without "locking" the whole OS. The Linux kernel recently gained support for live kernel patching [52], through the kpatch [50] and kGraft [51] projects. In the future, we plan to investigate whether mechanisms like the ones above can provide the foundation for implementing dynamic code diversification schemes, such as live CFA motion, without interrupting the normal operation of the kernel at all.

3.6.2 Low-latency Code Inflation

In Section 3.4, among other things, we investigated the impact of code inflation on kernel performance. Specifically, we measured the effect of different `NOP` sled sizes on system latency, and observed negligible increase for the range 0 – 32. However, recall that the maximum size of the `NOP` sleds is tunable (see Section 3.3; `nop` parameter). Hence, if (significantly) larger sleds are used (*e.g.,* 0 – 1024 or 0 – 4096), the overhead of code inflation becomes noticeable, as the extra `NOP` instructions consume CPU cycles and their impact on latency cannot be "masked" by the subsequent CFAs. Figure 3.9 sketches a potential solution to this problem, which we plan to explore in the future. The main idea is to prepend every `NOP` sled with a "skip branch" (*i.e.,* a direct branch instruction) that directly transfers control to the respective CFA and avoids executing the `NOP`s.

Availability

Our prototype implementation of kGuard is freely available, under the GNU General Public License (v3) [80], at: `https://www.cs.columbia.edu/~vpk/research/kguard/`. Interested readers are referred to Appendix C for more information on how to use it.

Chapter 4

ret2dir

4.1 Overview

Commodity OSes follow a design that weakly (asymmetrically) separates kernel from user space, in favor of faster interactions between user processes and the kernel, as we discussed in Chapter 2. The ret2usr protections outlined in Section 2.3, along with kGuard, aim to alleviate this design weakness, and fortify kernel isolation with minimal overhead. In this chapter, we seek to assess the security offered by these protections and investigate whether certain performance-oriented design choices can render them ineffective.

Our findings indicate that there exist fundamental decisions, deeply rooted into the architecture of the memory management (sub)system (`mm`) of modern OSes, which can be abused for weakening the isolation between kernel and user space. We introduce a novel kernel exploitation technique, named *return-to-direct-mapped memory* (ret2dir), which allows an attacker to perform the equivalent of a ret2usr attack on a hardened system.

4.1.1 Threat Model

We assume a kernel hardened against ret2usr attacks using one (or a combination) of the protection mechanisms discussed in Section 2.3, including kGuard. Moreover, we assume an *unprivileged* attacker with local access to the OS, who seeks to elevate privileges by exploiting a kernel-memory corruption vulnerability [35, 28, 40, 29, 43, 26, 22, 44, 42, 41, 33, 32, 31, 23, 38, 34, 39, 37] (see Section 2.2).

We do not make any assumptions regarding the type of corrupted data; code and data pointers are both possible targets [192, 78, 68, 79, 77, 76, 73]. Overall, the adversarial capabilities we presume are identical to those needed for carrying out a ret2usr attack.

4.1.2 Attack Strategy

In a kernel hardened against ret2usr attacks, the hijacked control or data flow can no longer be redirected to user space in a direct manner—the respective ret2usr protection scheme(s) will block any such attempt, as shown in Figure 2.2. However, the implicit physical memory sharing between user processes and the kernel allows an attacker to *deconstruct* the isolation guarantees offered by ret2usr protection mechanisms, and redirect the kernel's control or data flow to *user-controlled* code or data.

A key facility that enables the implicit sharing of physical memory is *`physmap`*: a large, contiguous virtual memory region inside kernel address space that contains a direct mapping of part or all (depending on the architecture) physical memory. This region plays a *crucial* role in enabling the kernel to allocate and manage dynamic memory, as fast as possible (we discuss the structure of `physmap` in Section 4.2). We should stress that although in this study we focus on Linux—one of the most widely used OSes—direct-mapped RAM regions exist (in some form) in many OSes, as they are considered standard practice in physical memory management. For instance, OpenBSD and Oracle Solaris use the `seg_kpm` and `pmap` mapping facility, respectively, to provide a direct mapping of the whole RAM in 64-bit architectures [140, 162].

As physical memory is allotted to user processes and the kernel, the existence of `physmap` results in *address aliasing*. Virtual address aliases, or *synonyms* [122], occur when two (or more) different virtual addresses map to the same physical memory address. Given that `physmap` maps a large part (or all) of physical memory within the kernel, the memory of an attacker-controlled user process is accessible through its kernel-resident synonyms.

The first step in mounting a ret2dir attack is to map, in user space, the exploit *payload*. Depending on whether the exploited vulnerability enables the corruption of a code pointer [192, 78, 68, 79, 77, 76, 73] or a data pointer [75, 72, 71], the payload will consist of either shellcode, or controlled (tampered-with) data structures, as shown in Figure 4.1.

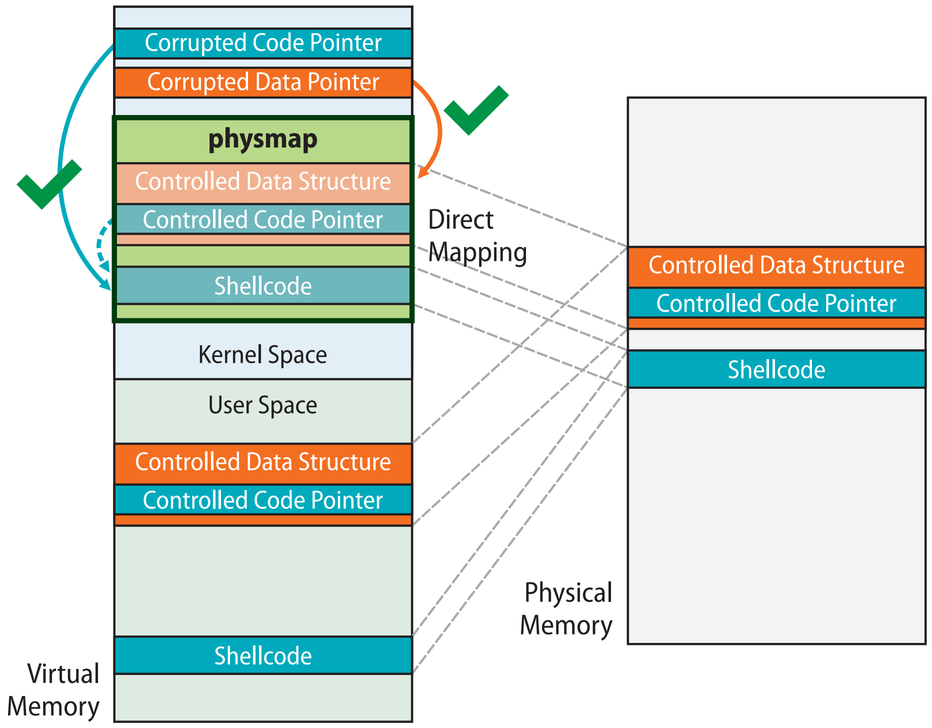


Figure 4.1: Overall ret2dir operation. The `physmap` (direct-mapped RAM) area enables a hijacked kernel code or data pointer to access user-controlled data, without crossing the user-kernel space boundary.

Whenever the `mm` (sub)system allocates (dynamic) memory to user space, it defers giving page frames until the very last moment. Specifically, physical memory is granted to user processes in a lazy manner, using the *demand paging* and *copy-on-write* methods [12], which both rely on page faults to actually allocate RAM. When the content of the payload is initialized, the MMU generates a page fault, and the kernel allocates a page frame to the attacking process. Page frames are typically managed by `mm` using a *buddy allocator* [121].

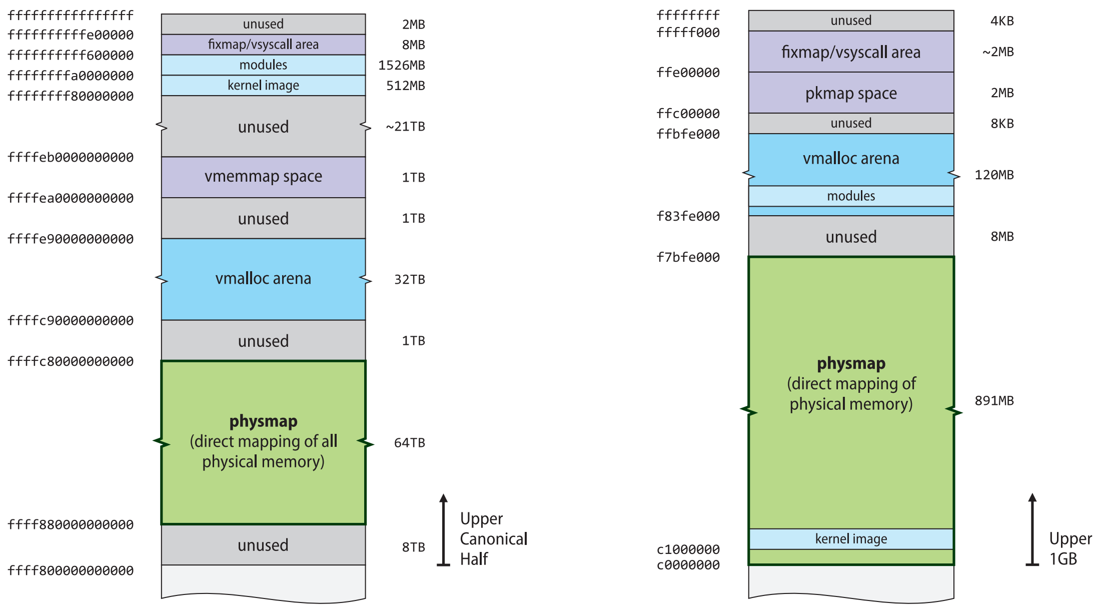
Given the existence of `physmap`, the moment the buddy allocator provides a page frame to be mapped in user space, `mm` effectively creates an alias of the exploit payload in kernel space, as shown in Figure 4.1. Although the kernel never uses such synonyms directly, `mm` keeps the whole RAM pre-mapped in order to boost page frame reclamation. This allows newly deallocated page frames to be made available to the kernel instantly, without the need to alter page tables (see Section 4.2.1 for more details).

Overall, ret2dir takes advantage of the implicit data sharing between user and kernel space (due to `physmap`) to redirect a hijacked kernel control or data flow to a set of kernel-resident synonym pages, effectively performing the equivalent of a ret2usr attack without reaching out to user space. It is important to note that the malicious payload "emerges" in kernel space the moment a page frame is given to the attacking process. The attacker does not have to explicitly "push" (copy) the payload to kernel space (*e.g.,* via pipes or message queues), as `physmap` makes it readily available. The use of such methods is also much less flexible, as the system imposes strict limits to the amount of memory that can be allocated for kernel-resident buffers, while the exploit payload will (most likely) have to be encapsulated in certain kernel data objects that can affect its structure.

4.2 Demystifying `physmap`

A critical first step in understanding the mechanics of ret2dir attacks is to take a look at how kernel address space is organized internally; we use the Linux x86/x86-64 platform as reference. The x86-64 architecture uses 48-bit virtual addresses that are sign-extended to 64 bits (*i.e.,* bits `[48:63]` are copies of bit `[47]`). This scheme natively splits the 64-bit virtual address space in two canonical halves of 128TB each. As shown in Figure 4.2a, kernel space occupies the upper half (`0xFFFF800000000000` – `0xFFFFFFFFFFFFFFFF`), and is further divided into six regions [119]: the `fixmap` area, modules, kernel image, `vmemmap` space, `vmalloc` arena, and `physmap`. In x86, on the other hand, the kernel space can be assigned to the upper 1GB, 2GB, or 3GB part of the address space, with the first option being the default. As kernel virtual address space is limited, it can become a scarce resource, and certain regions collide to prevent its waste (*e.g.,* modules and `vmalloc` arena, kernel image and `physmap`; see Figure 4.2b).[1] For the purposes of ret2dir, in the following, we focus only on the direct-mapped region.

[1]To access the contents of a page frame, the kernel must first map that frame in kernel space. In x86, however, the kernel has only 1GB – 3GB virtual addresses available for managing (up to) 64GB of RAM.



(a) x86-64 (b) x86

Figure 4.2: The internal layout of Linux kernel space in (a) x86-64 and (b) x86 (as of kernel version 3.12). In x86-64, the kernel space is further divided in six distinct regions, namely `fixmap`, modules, kernel image, `vmemmap` space, `vmalloc` arena, and `physmap`. In x86, where kernel addresses are a scarce resource, the distinct regions are only four; some of them collide to reduce space waste (*i.e.,* modules and `vmalloc` arena, kernel image and `physmap`). What is important to note, however, is that in both architectures there exists a *direct* mapping of all (or part of) RAM inside kernel space at a *fixed* location (values shown for the default 3G/1G user/kernel split).

4.2.1 Functionality

The `physmap` area is a mapping of paramount importance to the performance of the kernel, as it facilitates dynamic kernel memory allocation. At a high level, `mm` offers two main methods for requesting memory: `vmalloc` and `kmalloc`. With the `vmalloc` family of routines, memory can only be allocated in multiples of page size and is guaranteed to be virtually contiguous but *not* physically contiguous. In contrast, with the `kmalloc` family of routines, memory can be allocated in byte-level chunks, and is guaranteed to be *both* virtually and physically contiguous.

As it offers memory only in page multiples, `vmalloc` leads to higher internal memory fragmentation and often poor cache performance. More importantly, `vmalloc` needs to alter the page tables of the kernel every time memory is (de)allocated to map or unmap the respective page frames to or from the `vmalloc` arena. (see Figure 4.2). This not only incurs additional overhead, but results in increased TLB thrashing [135]. For these reasons, the majority of kernel components use `kmalloc`; the use of `vmalloc` is limited to performance-insensitive tasks, such as module loading. However, given that `kmalloc` can be invoked from any context, including that of interrupt service routines, which have strict timing constraints, it must satisfy a multitude of different (and contradicting) requirements. In certain contexts, the allocator should never sleep (*e.g.,* when locks are held). In other cases, it should never fail, or it should return memory that is guaranteed to be physically contiguous (*e.g.,* when a device driver reserves memory for DMA).

Given constraints like the above, *`physmap` is a necessity, as the main facilitator of optimal performance.* The `mm` developers opted for a design that lays `kmalloc` over a region[2] that pre-maps the entire RAM (or part of it) for the following reasons [12]. First, `kmalloc` (de)allocates memory without touching the page table(s) of the kernel. This not only reduces TLB pressure significantly, but also removes high-latency operations, like page table manipulation and TLB shootdowns [140], from the fast path. Second, the linear mapping of page frames results in virtual memory that is guaranteed, by design, to be always physically contiguous. This leads to increased cache performance, and has the added benefit of allowing drivers to directly assign `kmalloc`'ed regions to DMA devices that can only operate on physically contiguous memory (*e.g.,* when there is no IOMMU support). Finally, page frame accounting is greatly simplified, as address translations (virtual-to-physical and vice versa) can be performed using solely arithmetic operations [124].

[2]`kmalloc` is not directly layered over `physmap`. It is instead implemented as a collection of geometrically distributed (32B–4KB) *slabs*, which are in turn placed over `physmap`. The slab layer is a hierarchical, type-based data structure caching scheme. By taking into account certain factors, such as page and object sizes, cache line information, and memory access times (in NUMA systems), it can perform intelligent allocation choices that minimize memory fragmentation and make the best use of a system's cache hierarchy. Linux adopted the slab allocator of SunOS [10], and as of kernel v3.12, it supports three variants: `SLAB`, `SLUB` (default), and `SLOB`.

Architecture		**PHYS_OFFSET**	**Size**	**Prot.**
x86	(3G/1G)	0xC0000000	891MB	RW
	(2G/2G)	0x80000000	1915MB	RW
	(1G/3G)	0x40000000	2939MB	RW
AArch32	(3G/1G)	0xC0000000	760MB	RW**X**
	(2G/2G)	0x80000000	1784MB	RW**X**
	(1G/3G)	0x40000000	2808MB	RW**X**
x86-64		0xFFFF880000000000	64TB	RW(**X**)
AArch64		0xFFFFFFC000000000	256GB	RW**X**

Table 4.1: The characteristics of `physmap` in Linux (x86, x86-64, AArch32, AArch64).

4.2.2 Location and Size

The `physmap` region is an architecture-independent feature (this should come as no surprise given the reasons we outlined above) that exists in all popular Linux platforms. Depending on the memory addressing characteristics of each ISA, the size of `physmap` and its exact location may differ. Nonetheless, in all cases: *(i)* there exists a *direct* mapping of part or all physical memory in kernel space, and *(ii)* the mapping starts at a *fixed*, known location. The latter is true even in the case where kernel-space ASLR (KASLR) [66] is employed.

Table 4.1 lists the properties of `physmap` for the platforms we consider. In x86-64 systems, `physmap` maps directly, in a 1:1 manner, starting from page frame zero, the entire RAM of the system into a 64TB region. AArch64 systems use a 256GB region for the same purpose [138]. Conversely, in x86 systems, the kernel directly maps only a portion of RAM.

The size of `physmap` on 32-bit architectures depends on two factors: *(i)* the user/kernel split used (3G/1G, 2G/2G, or 1G/3G), and *(ii)* the size of the `vmalloc` arena. Under the default setting, in which 1GB is assigned to kernel space and the `vmalloc` arena occupies 120MB, the size of `physmap` is 891MB (1GB - `sizeof(vmalloc + pkmap + fixmap + unused)`) and starts at `0xC0000000`. Likewise, under a 2G/2G (1G/3G) split, `physmap` starts at `0x80000000` (`0x40000000`) and spawns 1915MB (2939MB). The situation in AArch32 is quite similar [117], with the only difference being the default size of the `vmalloc` arena (240MB).

Overall, in 32-bit systems, the amount of directly mapped physical memory depends on the size of RAM and `physmap`. If `sizeof(physmap)` $\geq$ `sizeof(RAM)`, then the entire RAM is direct-mapped—a common case for 32-bit mobile devices with up to 1GB of RAM. Otherwise, only `sizeof(physmap)/sizeof(PAGE_SZ)` pages are mapped directly.

4.2.3 Access Rights

A crucial aspect for mounting a ret2dir attack is the memory access rights of `physmap`. To get the protection bits of the kernel pages that correspond to the direct-mapped memory region, we built `kptdump`:[3] a utility in the form of a kernel module that exports page tables through the `debugfs` pseudo-filesystem [46]. The tool traverses the kernel page table, available via the global symbols `swapper_pg_dir` (x86/AArch32/AArch64) and `init_level4_pgt` (x86-64), and dumps the flags (`U/S`, `R/W`, `XD`) of every kernel page that falls within the `physmap` region.

In x86, `physmap` is mapped as "readable and writeable" (`RW`) in all kernel versions we tried (the oldest one was v2.6.32, released on Dec. 2009). In x86-64, however, the permissions of `physmap` are *not* in sane state. Kernels up to v3.8.13 violate the `W^X` property by mapping the entire region as "readable, writeable, and executable" (`RWX`)—only very recent kernels ($\geq$ v3.9) use the conservative `RW` mapping. AArch32 and AArch64 map `physmap` with `RWX` permissions in every kernel version we tested (up to v3.12).

[3] `kptdump` resembles the functionality of Arjan van de Ven's patch [210]; we had to resort to a custom solution, as that patch is only available for x86/x86-64 and cannot be used "as-is" in any other architecture.

4.3 Locating Synonyms

The final piece for mounting a ret2dir exploit is finding a way to reliably pinpoint the location of a synonym address in the `physmap` area, given its user-space counterpart. For legacy environments, in which `physmap` maps only part of the system's physical memory, such as a 32-bit system with 8GB of RAM, an additional requirement is to ensure that the synonym of a user-space address of interest exists.

We have developed two techniques for achieving both goals. The first relies on page frame information available through the `pagemap` interface of the `/proc` filesystem, which is currently accessible by non-privileged users in all Linux distributions we studied. As the danger of ret2dir attacks will (hopefully) encourage system administrators and Linux distributions to disable access to `pagemap`, we have developed a second technique that does not rely on any kernel information leakage.

4.3.1 Leaking PFNs (via **`/proc`**)

The `procfs` pseudo-filesystem [116] has a long history of leaking information that is security-sensitive [165, 102, 189]. Starting with kernel v2.6.25 (Apr. 2008), a set of pseudo-files, including `/proc/<pid>/pagemap`, were added in `/proc` to enable the examination of page tables for debugging purposes. To assess the prevalence of this facility, we tested the latest releases of the most popular distributions according to DistroWatch [63] (*i.e.,* Debian, Ubuntu, Fedora, and CentOS). In all cases, `pagemap` was enabled by default.

For every user-space page, `pagemap` provides a 64-bit value, indexed by (virtual) page number, which contains information regarding the presence of the page in RAM [131]. If a page is present in RAM, then bit `[63]` is set and bits `[0:54]` encode its page frame number (PFN). That being so, the PFN of a given user-space virtual address `uaddr`, can be located by opening `/proc/<pid>/pagemap` and reading eight bytes from file offset `(uaddr/PAGE_SZ) * sizeof(uint64_t)` (assuming `PAGE_SZ` = 4KB).

Armed with the PFN of a given `uaddr`, denoted as `PFN(uaddr)`, its synonym in `physmap`, `SYN(uaddr)`, can be located using the following formula:

$$\texttt{SYN(uaddr) = PHYS\_OFFSET + PAGE\_SZ * (PFN(uaddr) - PFN\_MIN)} \quad (4.1)$$

`PHYS_OFFSET` corresponds to the known, fixed starting virtual address of `physmap` (values for different configurations are shown in Table 4.1), and `PFN_MIN` is the first page frame number—in many architectures, including ARM, physical memory starts from a non-zero offset (*e.g.,* `0x60000000` in Versatile Express ARM boards, which corresponds to `PFN_MIN = 0x60000`). To prevent `SYN(uaddr)` from being reclaimed (*e.g.,* after swapping out `uaddr`), the respective user page(s) can be "pinned" to RAM using `mlock`.

4.3.1.1 `sizeof(RAM)` > `sizeof(physmap)`

For systems in which part of RAM is direct-mapped, only a subset of PFNs is accessible through `physmap`. For instance, in an x86 system with 4GB of RAM, the PFN range is `0x0`–`0x100000`. However, under the default 3G/1G split, the `physmap` region maps only the first 891MB of RAM (see Table 4.1 for alternative configurations), which means PFNs from `0x0` up to `0x37B00` (`PFN_MAX`). If the PFN of a user-space address is greater than `PFN_MAX` (the PFN of the last direct-mapped page), then `physmap` does not contain a synonym for that address. Naturally, the question that arises is whether we can *force* the buddy allocator to provide page frames with PFNs less than `PFN_MAX`.

For compatibility reasons, `mm` splits physical memory into several *zones*. In particular, DMA processors of older ISA buses can only address the first 16MB of RAM, while some PCI DMA peripherals can access only the first 4GB. To cope with such limitations, `mm` supports the following zones: `ZONE_DMA`, `ZONE_DMA32`, `ZONE_NORMAL`, and `ZONE_HIGHMEM`. The latter is available in 32-bit platforms and contains the page frames that cannot be directly addressed by the kernel (*i.e.,* those that are not mapped in `physmap`). `ZONE_NORMAL` contains page frames above `ZONE_DMA` (and `ZONE_DMA32`, in 64-bit systems) and below `ZONE_HIGHMEM`. When only part of RAM is direct-mapped, `mm` orders the zones as follows: `ZONE_HIGHMEM > ZONE_NORMAL > ZONE_DMA`.

Given a page frame request, mm will try to satisfy it starting with the highest zone that complies with the request (*e.g.,* the direct-mapped memory of ZONE_NORMAL is preferred for kmalloc), moving towards lower zones as long as there are no free page frames available.

From the attacker's perspective, user processes get their page frames from ZONE_HIGHMEM first, as mm tries to preserve the page frames that are direct-mapped for dynamic memory requests from the kernel. However, when the page frames of ZONE_HIGHMEM are depleted, due to increased memory pressure, *mm inevitably starts providing page frames from ZONE_NORMAL or ZONE_DMA.* Based on this, our strategy is as follows.

The attacking process repeatedly uses mmap to request memory. For each page in the requested memory region, the process causes a page fault by accessing a single byte, forcing mm to allocate a page frame (alternatively, the MAP_POPULATE flag in mmap can be used to pre-allocate all the respective page frames). The process then checks the PFN of every allocated page, and the same procedure is repeated until a PFN less than PFN_MAX is obtained. The synonym of such a page is then guaranteed to be present in physmap, and its exact address can be calculated using formula 4.1. Note that depending on the size of physical memory and the user/kernel split used, we may have to spawn additional processes to completely deplete ZONE_HIGHMEM. For example, on an x86 machine with 8GB of RAM and the default 3G/1G split, up to three processes might be necessary to guarantee that a page frame that falls within physmap will be acquired. Interestingly, the more benign processes are running on the system, the easier it is for an attacker to acquire a page with a synonym in physmap. The additional tasks create memory pressure, "pushing" the allocations of the attacker to the desired zones.

4.3.1.2 Contiguous synonyms

Certain exploits require more than a single page for their payload(s). Pages that are virtually contiguous in user space, however, do not necessarily map to page frames that are physically contiguous, which means that their synonyms will not be contiguous either. Yet, given physmap's linear mapping, two pages with consecutive synonyms have PFNs that are sequential. Therefore, if 0xBEEF000 and 0xFEEB000 have PFNs 0x2E7C2 and 0x2E7C3, respectively, then they are contiguous in physmap despite being ~64MB apart in user space.

To identify consecutive synonyms, we proceed as follows. Using the same methodology as above, we compute the synonym of a random user page. We then repeatedly obtain more synonyms, each time comparing the PFN of the newly acquired synonym with the PFNs of those previously retrieved. The process continues until any two (or more, depending on the exploit) synonyms have sequential PFNs. The exploit payload can then be split appropriately across the user pages that correspond to synonyms with sequential PFNs.

4.3.2 **`physmap`** Spraying

As eliminating access to `/proc/<pid>/pagemap` is a rather simple task, we also consider the case in which PFN information is not available. In such a setting, given a user page that is present in RAM, there is no direct way of determining the location of its synonym inside `physmap`. Recall that our goal is to identify a kernel-resident page in the `physmap` area that "mirrors" a user-resident exploit payload. Although we cannot pinpoint the synonym of a given user address, it is still possible to proceed in the opposite direction: pick an *arbitrary* `physmap` address, and ensure (to the extent possible) that its corresponding page frame is mapped by a user page that contains the exploit payload.

This can be achieved by exhausting the address space of the attacking process with (aligned) copies of the exploit payload, in a way similar to heap spraying [62]. The base address and length of the `physmap` area is known in advance (Table 4.1). The latter corresponds to $\texttt{PFN\_MAX} - \texttt{PFN\_MIN}$ page frames, shared among all user processes and the kernel. If the attacking process manages to copy the exploit payload into $\texttt{N}$ memory-resident pages (in the physical memory range mapped by `physmap`), then the probability ($\texttt{P}_{\texttt{succ}}$) that an arbitrarily chosen, page-aligned `physmap` address will map the exploit payload is:

$$\texttt{P}_{\texttt{succ}} = \frac{\texttt{N}}{\texttt{PFN\_MAX} - \texttt{PFN\_MIN}} \tag{4.2}$$

Our goal is to maximize $\texttt{P}_{\texttt{succ}}$.

4.3.2.1 Spraying

Maximizing $\texttt{N}$ is straightforward and boils down to acquiring as many page frames as possible. The technique we use is similar to the one presented in Section 4.3.1.

The attacking process repeatedly acquires memory using `mmap` and "sprays" the exploit payload into the returned regions. We prefer using `mmap`, over ways that involve `shmget`, `brk`, and `remap_file_pages`, due to system limits typically imposed on the latter. `MAP_ANONYMOUS` allocations are also preferred, as existing file-backed mappings (from competing processes) will be swapped out with higher priority compared to anonymous mappings. The copying of the payload causes page faults that result in page frame allocations by mm (alternatively `MAP_POPULATE` can be used). If the virtual address space is not enough for depleting the entire RAM, as is the case with certain 32-bit configurations, the attacking process must spawn additional child processes to assist with the allocations.

The procedure continues until mm starts swapping "sprayed" pages to disk. To pinpoint the exact moment that swapping occurs, each attacking process checks periodically if its sprayed pages are still resident in physical memory, by calling the `getrusage` system call every few `mmap` invocations. At the same time, all attacking processes start a set of background threads that repeatedly write-access the already allocated pages, emulating the behavior of `mlock`, and preventing (to the extent possible) sprayed pages from being swapped out—mm swaps page frames to disk using the LRU policy. Hence, by accessing pages repeatedly, mm is tricked to believe that they correspond to fresh content. When the number of memory-resident pages begins to drop (*i.e.,* the resident-set size (`RSS`) of the attacking process(es) starts decreasing), the maximum allowable physical memory footprint has been reached. Of course, the size of this footprint also depends on the memory load inflicted by other processes, which compete with the attacking processes for RAM.

4.3.2.2 Signatures

As far as `PFN_MAX` − `PFN_MIN` is concerned, we can reduce the set of potential target pages in the `physmap` region, by excluding certain pages that correspond to frames that the buddy allocator will never provide to user space. For example, in x86 and x86-64, the BIOS typically stores the hardware configuration detected during POST at page frame zero. Likewise, the physical address range `0xA0000`–`0xFFFFF` is reserved for mapping the internal memory of certain graphics cards.

In addition, the ELF sections of the kernel image that correspond to kernel code and global data are loaded at known, fixed locations in RAM (*e.g.,* `0x1000000` in x86). Based on these and other predetermined allocations, we have generated *`physmap` signatures* of reserved page frame ranges for each configuration we consider. If a signature is not available, then all page frames are potential targets. By combining `physmap` spraying and signatures, we can maximize the probability that our informed selection of an arbitrary page from `physmap` will point to the exploit payload. Specifically, the results of our experimental security evaluation (Section 4.6.2) show that, depending on the configuration, $\texttt{P}_{\texttt{succ}}$ can be as high as 96%.

4.4 Putting Everything Together

4.4.1 Bypassing SMAP, PAN, and UDEREF

We begin with an example of a ret2dir attack against an x86 system hardened with SMAP or UDEREF (PAN operates similarly to UDEREF). We assume an exploit for a kernel vulnerability that allows us to corrupt a kernel *data* pointer, named `kdptr`, and overwrite it with an arbitrary value [75, 72, 71]. On a system with an unhardened kernel, an attacker can overwrite `kdptr` with a user-space address, and force the kernel to dereference it by invoking the appropriate interface (*e.g.,* a buggy system call). However, the presence of SMAP or UDEREF will cause a memory access violation, effectively blocking the exploitation attempt. To overcome this, a ret2dir attack can be mounted as follows.

First, an attacker-controlled user process reserves a single page (4KB), say at address `0xBEEF000`. Next, the process moves on to initialize the newly allocated memory with the exploit payload (*e.g.,* a tampered-with data structure). This payload initialization phase will cause a page fault, triggering `mm` to request a free page frame from the buddy allocator and map it at address `0xBEEF000`. Suppose that the buddy system picks page frame 1904 (`0x770`). In x86, under the default 3G/1G split, `physmap` starts at `0xC0000000`, which means that the page frame has been pre-mapped at address `0xC0000000 + (4096 * 0x770) = 0xC0770000` (according to formula 4.1).

At this point, `0xBEEF000` and `0xC0770000` are synonyms; they both map to the physical page that contains the attacker's payload. Consequently, any data in the area `0xBEEF000`–`0xBEEFFFFF` is readily accessible by the kernel through the synonym addresses `0xC0770000`–`0xC0770FFF`. To make matters worse, given that `physmap` is primarily used for implementing dynamic memory, the kernel cannot distinguish whether the kernel data structure located at address `0xC0770000` is fake or legitimate (*i.e.,* properly allocated using `kmalloc`). Therefore, by overwriting `kdptr` with `0xC0770000` (instead of `0xBEEF000`), the attacker can bypass SMAP and UDEREF, as both protections consider benign any dereference of memory addresses above `0xC0000000`.

4.4.2 Bypassing SMEP, PXN, KERNEXEC, and kGuard

We use a running example from the x86-64 architecture to demonstrate how a ret2dir attack can bypass KERNEXEC, kGuard, and SMEP (PXN operates almost identically to SMEP). We assume the exploitation of a kernel vulnerability that allows the corruption of a kernel function pointer, namely `kfptr`, with an arbitrary value [79, 77, 76, 73]. In this setting, the exploit payload is not a set of fake data structures, but machine code (shellcode) to be executed with elevated privilege. In real-world kernel exploits, the payload typically consists of a multi-stage shellcode, the first stage of which stitches together kernel routines (second stage) for performing privilege escalation [200]. In most cases, this boils down to executing something similar to `commit_creds(prepare_kernel_cred(NULL))`. These two routines replace the credentials (`(e)uid`, `(e)gid`) of a user task with zero, effectively granting root privileges to the attacking process.

The procedure is similar as in the previous example. Suppose that the payload has been copied to user-space address `0xBEEF000`, which the buddy allocator assigned to page frame 190402 (`0x2E7C2`). In x86-64, `physmap` starts at `0xFFFF880000000000` (see Table 4.1), and maps the whole RAM using regular pages (4KB). Hence, according to formula 4.1, a synonym of address `0xBEEF000` is located within kernel space at address `0xFFFF880000000000 + (4096 * 0x2E7C2) = 0xFFFF87FF9F080000`.

In ret2usr scenarios where attackers control a kernel function pointer, an advantage is that they also control the memory access rights of the user page(s) that contain the exploit payload, making it trivially easy to mark the shellcode as executable. In a hardened system, however, a ret2dir attack allows controling only the *content* of the respective synonym pages within physmap—not their permissions. In other words, although the attacker can set the permissions of the range 0xBEEF000 – 0xBEEFFFF, this will *not* affect the access rights of the corresponding physmap pages.

Unfortunately, as shown in Table 4.1, the W^X property is not enforced in many platforms, including x86-64. In our example, the content of user address 0xBEEF000 is also accessible through kernel address 0xFFFF87FF9F080000 as plain, executable code. Therefore, by simply overwriting kfptr with 0xFFFF87FF9F080000 and triggering the kernel to dereference it, an attacker can directly execute shellcode with kernel privilege. KERNEXEC, kGuard, and SMEP (PXN) cannot distinguish whether kfptr points to malicious code or a legitimate kernel routine, and as long as kfptr $\geq$ 0xFFFF880000000000 and *kfptr is RWX, the dereference is considered benign.

4.4.2.1 Non-executable **physmap**

In the example above, we take advantage of the fact that some platforms map part (or all) of the physmap region as executable (X). The question that arises is whether ret2dir can be effective when physmap has sane permissions. As we demonstrate in Section 4.6.1, even in this case, ret2dir attacks are still possible through the use of ROP [100, 195, 16].

Let's revisit the previous example, this time under the assumption that physmap is not executable. Instead of mapping regular shellcode at 0xBEEF000, an attacker can map an equivalent ROP payload: a sequence of control and regular data that stitch together a chain of code fragments ending with ret instructions, known as gadgets. Gadgets are located in the kernel's (executable) .text segment, and perform a semantically equivalent computation with the regular shellcode (*e.g.,* commit_creds(prepare_kernel_cred(NULL)). To trigger the ROP chain, kfptr is overwritten with an address that points to a *stack pivoting* gadget [187, 197], which is needed to set the stack pointer to the beginning of the ROP payload, so that each gadget can transfer control to the next one.

By overwriting `kfptr` with the address of a pivot sequence, like `xchg %rax, %rsp; ret` (assuming that `%rax` points to `0xFFFF87FF9F080000`), the synonym of the ROP payload now acts as a kernel-mode stack. Note that Linux allocates a separate kernel stack for every user thread using `kmalloc`, making it impossible to differentiate between a legitimate stack and a ROP payload "pushed" in kernel space using ret2dir, as both reside in `physmap`. Finally, the ROP code should also take care of restoring the stack pointer (and possibly other CPU registers) to allow for *reliable kernel continuation* [4, 173].

4.5 Implementation

We have implemented the techniques presented in the previous sections under a common framework for the development of ret2dir exploits. Our implementation is readily available as a shared library, called `libret2dir`, and provides support for x86/x86-64 and AArch32/AArch64 Linux architectures (additional targets can be added with moderate engineering effort). `libret2dir` can be used to aid the development of kernel exploits against ret2usr-hardened targets, by automating the process of injecting the exploit payload in kernel space. In addition, it can be used to convert existing ret2usr exploits to ret2dir-equivalents with minimal effort.

The core part of the API provided by `libret2dir` consists of the following functions:

```
int      ret2dir_init();
caddr_t  ret2dir_get(size_t &pnum);
int      ret2dir_map(void *src, size_t len);
```

At first, the library has to be initialized by invoking `ret2dir_init`. This function launches a discovery phase for determining the features of the target platform that are of interest for ret2dir, such as `PHYS_OFFSET`, `PFN_MAX`, `PFN_MIN`, the user/kernel split used (32-bit architectures only), *etc.* At this stage, `libret2dir` also checks whether `pagemap` is accessible through `/proc`. If it is available, then `libret2dir` uses the PFN leaking technique to locate synonyms. Otherwise, it falls back to the page frame spraying method.

Once the library is initialized, there are two principal methods that an exploit writer can use: *get* and *map*. The former is implemented by `ret2dir_get`, which requests `pnum` contiguous pages in `physmap`. If `pagemap` is available, then the technique presented in Section 4.3.1 is used for obtaining up to `pnum` contiguous synonym pages (the number of *actual* pages obtained is returned back via `pnum`); synonyms are internally represented in the form of `(uaddr, kaddr)` pairs. After locating the synonyms, `ret2dir_get` returns `kaddr_base`, which is the base address of the contiguous kernel-space synonyms. The exploit writer can assume that the payload will be available at `[kaddr_base:kaddr_base + (pnum * PAGE_SZ) - 1]`, and use addresses from this region (instead of user-space addresses) when corrupting kernel data or code poiners. If `pagemap` is not available, then `libret2dir` randomly selects a page-aligned address from within the `physmap` region, by taking into consideration the appropriate `physmap` signature (if available), and returns that as `kaddr_base`. Note that in this case, only a single page is reserved (`pnum = 1`).

Armed with the address of the synonym in kernel space, the exploit writer can now proceed to prepare the payload, which is typically position-dependent (especially in the case of ROP code). Once ready, `ret2dir_map` can be used to inject the payload in kernel space. The exploit writer provides a pointer to the beginning of the payload along with its size. If `pagemap` is available, `libret2dir` makes the payload available at `kaddr_base`, by copying it in to the appropriate user-space locations (*i.e.,* `uaddr` instances), recorded during the invocation of `ret2dir_get`. Otherwise, it begins the page frame spraying process, as described in Section 4.3.2, with the hope that at the end of it `kaddr_base` will map to an instance of the "sprayed" payload.

4.6 Evaluation

4.6.1 Effectiveness

We evaluated the effectiveness of ret2dir against kernels hardened with ret2usr protections, using real-world and custom exploits. We obtained a set of 8 ret2usr exploits from EDB [158], covering a wide range of kernel versions (v2.6.33.6–v3.8). We ran every exploit on an unhardened kernel to verify that works and follows a ret2usr exploitation approach.

Next, we repeated the same experiment with every kernel hardened against ret2usr attacks, and, as expected, all exploits failed. Finally, we transformed the exploits into ret2dir-equivalents, using `libret2dir` (Section 4.5), and used them against the same hardened systems. Overall, our ret2dir versions of the exploits *bypassed all available ret2usr protections*, namely SMEP, SMAP, PXN, KERNEXEC, UDEREF, and kGuard.[4]

Table 4.2 summarizes our findings. The first two columns (EDB-ID and CVE) correspond to the tested exploit, and the third (Arch.) and fourth (Kernel) denote the architecture and kernel version used. The Payload column indicates the type of payload pushed in kernel space using ret2dir, which can be: a ROP payload (`ROP`), executable instructions (`SHELLCODE`), tampered-with data structures (`STRUCT`), or a combination of the above, depending on the exploit. The Protection column lists the deployed protection mechanisms in each case. Empty cells correspond to protections that are not applicable in the given setup, because they may not be: *(i)* available for a particular architecture, *(ii)* supported by a given kernel version, *(iii)* relevant against certain types of exploits. For instance, PXN is available only in ARM architectures [130], while SMEP and SMAP are Intel processor features [87, 54]. Furthermore, support for SMEP was added in kernel v3.5 [220] and for SMAP in v3.7 [49]. Note that depending on the permissions of the `physmap` area (see Table 4.1), we had to modify some exploits that relied on plain shellcode to use a ROP payload, in order to achieve arbitrary code execution (while in ret2usr exploits attackers can give executable permissions to the user-space memory that contains the payload, in ret2dir exploits it is not possible to modify the permissions of `physmap`).[5] Entries for kGuard marked with ⋆ require access to the (randomized) `.text` segment of the kernel.

As we mentioned in Section 2.3, KERNEXEC and UDEREF were recently ported to the AArch32 architecture [199]. In addition to providing stronger address space separation, the authors made an effort to fix the permissions of the kernel in AArch32, by enforcing the `W^X` property for the majority of `RWX` pages in `physmap`.

[4]At the time of writting, we did not have access to a CPU (or CPU emulator) that supported PAN.

[5]Exploit `15285` uses ROP code to bypass KERNEXEC/UDEREF and plain shellcode to evade kGuard. Exploit `26131` uses ROP code in x86 (kernel v3.5) to bypass KERNEXEC/UDEREF and SMEP/SMAP, and plain shellcode in x86-64 (kernel v3.8) to bypass kGuard, KERNEXEC, and SMEP.

EDB-ID	**CVE**	**Arch.**	**Kernel**	**Payload**	**Protection**						**Bypassed**
26131	2013-2094	x86/x86-64	3.5/3.8	ROP/SHELLCODE	KERNEXEC	UDEREF	kGuard	SMEP	SMAP		✓
24746	2013-1763	x86-64	3.5	SHELLCODE	KERNEXEC		kGuard	SMEP			✓
15944	N/A	x86	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*****				✓
15704	2010-4258	x86	2.6.35.8	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*****				✓
15285	2010-3904	x86-64	2.6.33.6	ROP/SHELLCODE	KERNEXEC	UDEREF	kGuard				✓
15150	2010-3437	x86	2.6.35.8	STRUCT		UDEREF					✓
15023	2010-3301	x86-64	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*****				✓
14814	2010-2959	x86	2.6.33.6	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*****				✓
Custom	N/A	x86	3.12	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*****	SMEP	SMAP		✓
Custom	N/A	x86-64	3.12	STRUCT+ROP	KERNEXEC	UDEREF	kGuard*****	SMEP	SMAP		✓
Custom	N/A	AArch32	3.8.7	STRUCT+SHELLCODE	KERNEXEC	UDEREF	kGuard				✓
Custom	N/A	AArch64	3.12	STRUCT+SHELLCODE			kGuard			PXN	✓

Table 4.2: Tested exploits (converted to use the ret2dir technique) and configurations.

However, as the respective patch is currently under development, there still exist regions inside `physmap` that are mapped as `RWX`. In kernel v3.8.7, we identified a ~6MB `physmap` region mapped as `RWX` that enabled the execution of plain shellcode in our ret2dir exploit.

The most recent kernel version for which we found a publicly-available exploit is v3.8. Thus, to evaluate the latest kernel series (v3.12) we used a custom exploit. We artificially injected two vulnerabilities that allowed us to corrupt a kernel data or function pointer, and overwrite it with a user-controlled value (marked as “Custom” in Table 4.2). Note that both flaws are similar to those exploited by the publicly-available exploits. Regarding ARM, the most recent PaX-protected AArch32 kernel that we managed to boot was v3.8.7.

We tested every applicable protection for each exploit. In all cases, the ret2dir versions transfered control *solely* to kernel addresses, bypassing all deployed protections. Figure 4.3 depicts the shellcode we used in x86-64 and AArch32 architectures. The shellcode is *relocatable*, so the only change needed in each exploit is to to replace `pkcred` and `ccreds` with the addresses of `prepare_kernel_cred` and `commit_creds`, respectively, as discussed in Section 4.4.2. By copying the shellcode into a user-space page that has a synonym in the `physmap` area, we can directly execute it from kernel mode by overwriting a kernel code pointer with the `physmap`-resident synonym address of the user-space page. We followed this strategy for all cases in which `physmap` was mapped as executable (corresponding to the entries of Table 4.2 that contain `SHELLCODE` in the Payload column).

For cases in which `physmap` is non-executable, we substituted the shellcode with a ROP payload that achieves the same purpose. In those cases, the corrupted kernel code pointer is overwritten with the address of a stack pivoting gadget, which brings the kernel's stack pointer to the `physmap` page that is a synonym for the user page that contains the ROP payload. Figure 4.4 shows an example of an x86 ROP payload used in our exploits. The first gadgets preserve the `%esp` and `%ebp` registers to facilitate reliable continuation (as discussed in Section 4.4.2.1). The scratch space can be conveniently located inside the controlled page(s), so the addresses `SCRATCH_SPACE_ADDR1` and `SCRATCH_SPACE_ADDR2` can be easily computed accordingly. The payload then executes (semantically) the same code as the shellcode to elevate privilege.

```
          x86-64                        AArch32
------------------------------------------------------------

push   %rbp                  |   push  r3, lr
mov    %rsp,       %rbp      |   mov   r0, #0
push   %rbx                  |   ldr   r1, [pc, #16]
mov    $<pkcred>, %rbx       |   blx   r1
mov    $<ccreds>, %rax       |   pop   r3, lr
mov    $0x0,       %rdi      |   ldr   r1, [pc, #8]
callq  *%rax                 |   bx    r1
mov    %rax,       %rdi      |   <pkcred>
callq  *%rbx                 |   <ccreds>
mov    $0x0,       %rax      |
pop    %rbx                  |
leaveq                       |
ret                          |
```

Figure 4.3: The plain shellcode used in ret2dir exploits for x86-64 (left) and AArch32 (right) targets (`pkcred` and `ccreds` correspond to the addresses of `prepare_kernel_cred` and `commit_creds`).

4.6.2 Performance

In systems without access to `pagemap`, ret2dir attacks have to rely on `physmap` spraying to find a synonym that corresponds to the exploit payload. As discussed in Section 4.3.2, the probability of randomly selecting a `physmap` address that maps to the exploit payload depends on: *(i)* the amount of installed RAM, *(ii)* the physical memory load due to competing processes, and *(iii)* the size of the `physmap` area. To assess this probability, we performed a series of experiments under different system configurations and workloads.

```
                /* save orig. esp              */
0xc10ed359      /* pop    %edx         ; ret    */
<SCRATCH_SPACE_ADDR1>
0xc127547f      /* mov    %eax, (%edx) ; ret    */
                /* save orig. ebp              */
0xc10309d5      /* xchg   %eax, %ebp   ; ret    */
0xc10ed359      /* pop    %edx         ; ret    */
<SCRATCH_SPACE_ADDR2>
0xc127547f      /* mov    %eax, (%edx) ; ret    */
       /* commit_creds(prepare_kernel_cred(0))*/
0xc1258894      /* pop    %eax         ; ret    */
0x00000000
0xc10735e0      /* addr. of prepare_kernel_cred */
0xc1073340      /* addr. of commit_creds'       */
                /* restore the saved CPU state  */
0xc1258894      /* pop    %eax         ; ret    */
<SCRATCH_SPACE_ADDR2>
0xc1036551      /* mov  (%eax), %eax   ; ret    */
0xc10309d5      /* xchg   %eax, %ebp   ; ret    */
0xc1258894      /* pop    %eax         ; ret    */
<SCRATCH_SPACE_ADDR1>
0xc1036551      /* mov  (%eax), %eax   ; ret    */
0xc100a7f9      /* xchg   %eax, %esp   ; ret    */
```

Figure 4.4: Example of an x86 ROP payload (kernel v3.8) used in our ret2dir exploits.

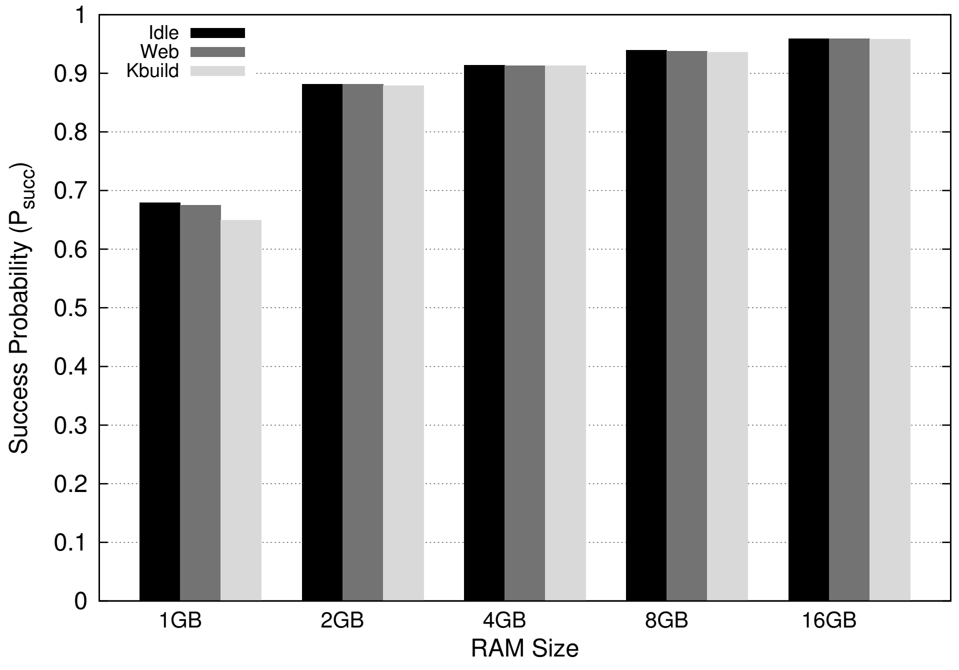


Figure 4.5: Probability (P_{succ}) that a selected `physmap` address maps to the exploit payload (successful exploitation) with a single attempt, when using `physmap` spraying, as a function of the available RAM in the system.

Figure 4.5 shows the probability of successfully selecting a `physmap` address, with a *single* attempt, as a function of the amount of RAM installed in our system. Our testbed consisted of a single host armed with two 2.66GHz quad-core Intel Xeon X5500 CPUs and 24GB of RAM, running 64-bit Debian Linux v7. Each bar denotes the average value over 5 repetitions of the same experiment. On each iteration we count the percentage of the *maximum* number of `physmap`-resident page frames that we managed to aquire, using the spraying technique (Section 4.3.2), over the size of `physmap`. We used three different workloads of increasing memory pressure: an idle system (`Idle`), a desktop-like workload with constant browsing activity (`Web`) in multiple tabs (Facebook, Gmail, Twitter, YouTube, and a mostly-static website), and a distributed kernel compilation (`Kbuild`) with 16 parallel threads running on 8 CPU cores (`gcc`, `as`, `ld`, `make`). Note that it is necessary to maintain continuous activity in the competing processes for their working set to remain *hot*, otherwise the ret2dir processes can easily "steal" their memory-resident pages.

The probability of success increases with the amount of RAM. For the lowest-memory configuration (1GB), the probability ranges between 65–68%, depending on the workload. This small difference between the idle and the intensive workloads is an indication that despite the continuous activity of the competing processes, the ret2dir processes manage to claim a large amount of memory, as a result of their repeated accesses to already allocated pages that in essence "lock" them to main memory. For the 2GB configuration the probability jumps to 88%, and reaches 96% for 16GB.

Note that as these experiments were performed on a 64-bit system, `physmap` always mapped all available memory. On 32-bit platforms, in which `physmap` maps only a subset of RAM, the probability of success is even higher. As discussed in Section 4.3.1.1, in such cases, the additional memory pressure created by competing processes, which more likely were spawned *before* the ret2dir processes, helps "pushing" ret2dir allocations to the desired zones (`ZONE_NORMAL`, `ZONE_DMA`) that fall within the `physmap` area. Finally, depending on the vulnerability, it is quite common that an unsuccessful attempt will not result in a kernel panic, allowing the attacker to run the exploit multiple times.

Availability

The complete set of exploits developed to assess the effectiveness of our ret2dir attack(s) is available at: `https://www.cs.columbia.edu/~vpk/research/ret2dir/`. Interested readers are referred to Appendix D for more information on how to use them.

Chapter 5

XPFO

5.1 Overview

In this chapter, we investigate whether a physical memory management approach that enforces the *explicit* use of page frames by different security contexts (user vs. kernel), can efficiently and effectively protect the OS kernel against ret2dir attacks.

Restricting access to `/proc/<pid>/pagemap` and similar interfaces, or disabling such features completely (*e.g.,* in Linux this can be easily achieved by compiling the kernel without support for `PROC_PAGE_MONITOR`), is a simple step that can hinder, but not prevent, ret2dir attacks. To that end, we present the design, implementation, and evaluation of an eXclusive Page Frame Ownership (XPFO) scheme for Linux, which provides strong kernel isolation and effective ret2dir protection with low overhead.

5.1.1 Threat Model

We assume a kernel hardened against ret2usr attacks using one (or a combination) of the protection mechanisms outlined in Section 2.3. In addition, we assume an *unprivileged* attacker with local access, who seeks to elevate privileges by exploiting a kernel-memory corruption vulnerability [35, 28, 40, 29, 43, 26, 22, 44, 42, 41, 33, 32, 31, 23, 38, 34, 39, 37] (see Section 2.2). Recall that in a kernel hardened against ret2usr attacks, the hijacked control or data flow can no longer be redirected to user space in a direct manner, as the respective ret2usr protection scheme(s) will block any such attempt.

For that reason, we surmise the (ab)use of the implicit physical memory sharing between user processes and the kernel (*e.g.,* through `physmap`), which allows an attacker to *deconstruct* the isolation guarantees offered by ret2usr protection mechanisms, and redirect the kernel's control or data flow to *user-controlled* code or data.

Note that we do not make any assumption(s) regarding the type of corrupted data; both code and data pointers are possible targets [192, 78, 68, 79, 77, 76, 73]. Overall, the adversarial capabilities we presume are identical to those needed for carrying out the ret2dir attacks introduced in Chapter 4.

5.2 Design

XPFO is a thin management layer that enforces *exclusive ownership* of page frames by either the kernel or user-level processes. Specifically, under XPFO, page frames can *never* be assigned to both kernel and user space, unless a kernel component explicitly requests that (*e.g.,* to implement zero-copy buffers [182]).

We opted for a design that does not penalize the performance-critical kernel allocators, at the expense of low additional overhead whenever page frames are allocated to (or reclaimed from) user processes. Recall that physical memory is allotted to user space using the demand paging and copy-on-write (COW) methods [12], both of which rely on page faults to allocate RAM. Hence, user processes already pay a runtime penalty for executing the page fault handler and performing the necessary bookkeeping. XPFO aligns well with this design philosophy, and increases only marginally the management and runtime overhead of user-space page frame allocation.

Crucially, the `physmap` area is left untouched, and the slab allocator, as well as kernel components that interface directly with the buddy allocator, continue to get pages that are guaranteed to be physically contiguous and benefit from fast virtual-to-physical address translations, as there are no extra page table walks or modifications (see Section 4.2.1).

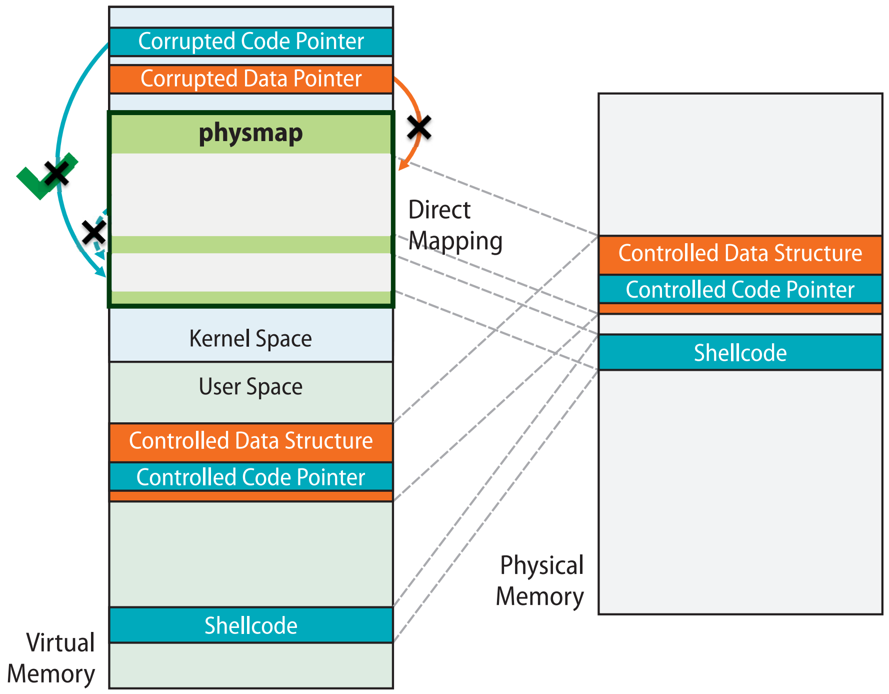


Figure 5.1: Overall XPFO operation. Whenever page frames are assigned to a user process, XPFO unmaps their respective synonyms from `physmap`, breaking unintended aliasing and ensuring that malicious content can no longer be "injected" to kernel space using ret2dir.

Figure 5.1 summarizes the operation of XPFO. Whenever a page frame is assigned to a user process, XPFO unmaps its respective synonym from `physmap`, thus breaking unintended aliasing and ensuring that malicious content can no longer be "injected" to kernel space using ret2dir. Likewise, when a user process releases page frames back to the kernel, XPFO maps the corresponding pages back in `physmap` to proactively facilitate dynamic (kernel) memory requests.

A key requirement here is to *wipe out* the content of page frames that are returned by (or reclaimed from) user processes, before making them available to the kernel. Otherwise, a non-sanitizing XPFO scheme would be vulnerable to the following attack. A malicious program spawns a child process that uses the techniques presented in Section 4.4 to map its payload. Since XPFO is in place, the payload is unmapped from `physmap` and cannot be addressed by the kernel. Yet, it will be mapped back once the child process terminates, making it readily available to the malicious program for mounting a ret2dir attack.

5.3 Implementation

We implemented XPFO in the Linux kernel v3.13. We devoted a significant amount of time to engineer our patch to fulfill the design requirements described in Section 5.2, and make as few and localized kernel modifications as possible—our aim is to to integrate XPFO in future Linux releases. Our implementation (~500LOC) keeps the management and runtime overhead to the minimum, by employing a set of optimizations related to TLB handling and page frame cleaning, and handles appropriately *all* cases in which page frames are allocated to (and reclaimed from) user processes. Specifically, XPFO deals with: *(a)* demand paging frames due to previously-requested anonymous and shared memory mappings (`brk`, `mmap/mmap2`, `mremap`, `shmat`), *(b)* COW frames (`fork`, `clone`), *(c)* explicitly and implicitly reclaimed frames (`_exit`, `munmap`, `shmdt`), *(d)* swapping (both swapped out and swapped in pages), *(e)* NUMA frame migrations (`migrate_pages`, `move_pages`), and *(f)* huge pages and transparent huge pages.

Handling the above cases is quite challenging. We first extended the system's page frame data structure (`struct page`) with the following fields: `xpfo_kmcnt` (reference counter), `xpfo_lock` (spinlock) and `xpfo_flags` (32-bit flags field)—`struct page` already contains a flags field, but in 32-bit systems it is quite congested [47]. Note that although the kernel keeps a `struct page` object for *every* page frame in the system, our change requires only 3MB of additional space per 1GB of RAM (~0.3% overhead). Moreover, out of the 32 bits available in `xpfo_flags`, we only make use of three: "Tainted" (`T`; bit 0), "Zapped" (`Z`; bit 1), and "TLB-shootdown needed" (`S`; bit 2).

Next, we extended the buddy system. Whenever the buddy allocator receives requests for page frames destined to user space (requests with `GFP_USER`, `GFP_HIGHUSER`, or `GFP_HIGHUSER_MOVABLE` set to `gfp_flags`), XPFO unmaps their respective synonyms from `physmap` and asserts `xpfo_flags.T`, indicating that the frames will be allotted to userland and their contents are not trusted anymore. In contrast, for page frames destined to kernel space, XPFO asserts `xpfo_flags.S` (optimization; see Section 5.3.1).

Whenever page frames are released to the buddy system, XPFO checks if `xpfo_flags.T` was previously asserted. If so, the frame was mapped to user space and needs to be wiped out. After zeroing its contents, XPFO maps it back to `physmap`, resets `xpfo_flags.T`, and asserts `xpfo_flags.Z` (optimization; more on that in Section 5.3.1). If `xpfo_flags.T` was not asserted, the buddy system reclaimed a frame previously allocated to the kernel itself and no action is necessary (fast-path; no interference with kernel allocations). Note that in 32-bit systems, the above are not performed if the page frame in question comes from `ZONE_HIGHMEM`—this zone contains page frames that are not direct-mapped.

Finally, to achieve complete support of cases *(a)–(f)*, we leverage `kmap/kmap_atomic` and `kunmap/kunmap_atomic`. These functions are used to temporarily (un)map page frames acquired from `ZONE_HIGHMEM` (see Section 4.3.1.1). In 64-bit systems, where the whole RAM is direct-mapped, `kmap/kmap_atomic` returns the address of the respective page frame directly from `physmap`, whereas `kunmap/kunmap_atomic` is defined as NOP and optimized by the compiler. If XPFO is enabled, all of them are re-defined accordingly.

As user pages are (preferably) allocated from `ZONE_HIGHMEM`, the kernel wraps *all* code related to the cases we consider (*e.g.,* demand paging, COW, swapping) with the above functions. Kernel components that use `kmap` to operate on page frames not related to user processes do exist, and we distinguish these cases using `xpfo_flags.T`. If a page frame is passed to `kmap/kmap_atomic` and that bit is asserted, this means that the kernel tries to operate on a frame assigned to user space via its kernel-resident synonym (*e.g.,* to read its contents for swapping it out), and thus is temporarily mapped back in `physmap`. Likewise, in `kunmap/kunmap_atomic`, page frames with `xpfo_flags.T` asserted are unmapped. Note that in 32-bit systems, the XPFO logic is executed on `kmap` routines only for direct-mapped page frames (see Table 4.1). `xpfo_lock` and `xpfo_kmcnt` are used for handling recursive or concurrent invocations of `kmap/kmap_atomic` and `kunmap/kunmap_atomic` with the same page frame.

5.3.1 Optimizations

The overhead of XPFO stems mainly from two factors: *(i)* sanitizing the content of reclaimed pages, and *(ii)* TLB shootdown and flushing (necessary since we modify the kernel page table). We employ three optimizations to keep that overhead to the minimum. As full TLB flushes result in prohibitive slowdowns [152], in architectures that support single TLB entry invalidation, XPFO selectively evicts only those entries that correspond to synonyms in `physmap` that are unmapped; in x86/x86-64 this is done with the `INVLPG` instruction.

In systems with multi-core CPUs, XPFO must take into consideration TLB coherency issues. Specifically, we have to perform a TLB shootdown whenever a page frame previously assigned to the kernel itself is mapped to user space. XPFO extends the buddy system to use `xpfo_flags.S` for this purpose. If that flag is asserted when a page frame is alloted to user space, XPFO invalidates the TLB entries that correspond to the synonym of that frame in `physmap`, in *every* CPU core, by sending IPI interrupts to cascade TLB updates. In all other cases (*i.e.,* page frames passed from one process to another, reclaimed page frames from user processes that are later on alloted to the kernel, and page frames allocated to the kernel, reclaimed, and subsequently allocated to the kernel again), XPFO performs only local TLB invalidations.

To alleviate the impact of page sanitization, we exploit the fact that page frames previously mapped to user space, and subsequently reclaimed by the buddy system, have `xpfo_flags.Z` asserted. We extended `clear_page` to check `xpfo_flags.Z` and avoid clearing the frame if the bit is asserted. This optimization avoids zeroing a page frame twice, in case it was first reclaimed by a user process and then subsequently allocated to a kernel control path that required a clean page—`clear_page` is invoked by every kernel execution path that requires a zero-filled page frame.

Benchmark	**Metric**	**Original**	**XPFO (%Overhead)**
Apache	Req/s	17636.30	17456.47 **(%1.02)**
NGINX	Req/s	16626.05	16186.91 **(%2.64)**
PostgreSQL	Trans/s	135.01	134.62 **(%0.29)**
Kbuild	sec	67.98	69.66 **(%2.47)**
Kextract	sec	12.94	13.10 **(%1.24)**
GnuPG	sec	13.61	13.72 **(%0.80)**
OpenSSL	Sign/s	504.50	503.57 **(%0.18)**
PyBench	ms	3017.00	3025.00 **(%0.26)**
PHPBench	Score	71111.00	70979.00 **(%0.18)**
IOzone	MB/s	70.12	69.43 **(%0.98)**
tiobench	MB/s	0.82	0.81 **(%1.22)**
dbench	MB/s	20.00	19.76 **(%1.20)**
PostMark	Trans/s	411.00	399.00 **(%2.91)**

Table 5.1: XPFO performance evaluation results using micro- and macro-benchmarks from the Phoronix Test Suite.

5.4 Evaluation

5.4.1 Effectiveness

To evaluate the effectiveness of the proposed protection scheme, we used the ret2dir versions of the real-world exploits presented in Section 4.6.1. We back-ported our XPFO patch to each of the six kernel versions used in our previous evaluation (see Table 4.2), and tested again our ret2dir exploits when XPFO was enabled. In all cases, XPFO prevented the respective exploitation attempt.

5.4.2 Performance

To assess the performance overhead of XPFO we used kernel v3.13, and a collection of micro- and macro-benchmarks from the Phoronix Test Suite (PTS) [178]. PTS puts together *standard* system tests, like `apachebench`, `pgbench`, kernel build, and `IOzone`, which are typically used by kernel developers to track performance regressions. Our testbed was the same with the one used in Section 4.6.2; Table 5.1 summarizes our findings (the numbers reported are average values over 5 repetitions). Overall, XPFO introduces a minimal (negligible in most cases) overhead, ranging between 0.18–2.91%.

5.5 Discussion

5.5.1 Limitations

XPFO provides protection against ret2dir attacks, by braking the unintended address space sharing between different security contexts. However, it does not prevent generic forms of data sharing between kernel and user space, such as user-controlled content pushed to kernel space via I/O buffers, the page cache, or system objects like pipes and message queues.

5.5.2 Alternative Design Approaches

5.5.2.1 Physical Memory Partitioning

An alternative solution to XPFO is to split physical memory in two *mutually exclusive* parts. Pages frames in the first part are permanently assigned to the kernel and are pre-mapped in the `physmap` area, while frames from the second part are alloted to user processes on demand. This scheme guarantees by design that there are no synonyms of user-space addresses in kernel space: page frames belong either to the kernel domain or the user domain. An implementation of this scheme would require minimal changes in `mm` to ensure that user-domain page frames are always assigned from the part of RAM that is not mapped in `physmap`. As discussed in Section 4.3.1.1, in 32-bit systems `ZONE_HIGHMEM` currently contains page frames that cannot be directly addressed by the kernel through `physmap`, so this or a similar zone could be leveraged for this purpose.

Likewise, the kernel will use only frames from zone(s) that are included in the `physmap` area. The main drawback of this approach is resource underutilization, as any unused page frame in either part will not be available to the other part when the latter has depleted its memory. For instance, although the kernel might still have available page frames for its own use, it will refuse to provide them to user processes in case all user page frames have been allocated. A more resource-efficient approach would be to dynamically adjust the size of the two domains, although the overhead for doing so needs to be explored.

5.5.2.2 `physmap` Layout Randomization

Another potential mitigation against page frame spraying is to randomize the location of `physmap`, by taking advantage of the ample virtual address space in 64-bit systems. The simplest scheme would be to divide `physmap` into RAM-sized slots, and map the entire RAM into one of them at boot time. In x86-64 the current size of `physmap` is 64TB (see Table 4.1). Therefore, assuming a machine with 64GB RAM, the aforementioned scheme will result in 1024 slots (*i.e.,* 10 bits of randomization entropy). Even after successfully spraying and occupying 96% of RAM, the attacker would now have to guess the right one among the 1024 slots.

As the amount of RAM in future systems increases, however, randomization entropy will drop, while (depending on the exploit) an attacker might be able to launch repeated ret2dir attempts. In addition, this solution is not applicable in 32-bit systems, as in most cases the size of the physical memory is close to (or exceeds) the size of `physmap`.

Availability

Our XPFO prototype, mainly implemented as a patch for Linux kernel v3.13, is freely available at: `https://www.cs.columbia.edu/~vpk/research/xpfo/`. Interested readers are referred to Appendix E for more information on how to use it.

Chapter 6

Conclusion

6.1 Summary

In this dissertation, we investigated the hypothesis that the security posture of modern OSes can be improved by employing a synergy of defenses and exploit prevention techniques, which guarantee the strong isolation between user and kernel space.

Towards this goal, we presented kGuard: a lightweight, compiler-based mechanism that protects the kernel from ret2usr threats. Unlike previous work, kGuard is fast, flexible, and offers cross-platform support. It works by injecting fine-grained inline guards during the translation phase, which are resistant to bypass, and it does not require any modification to the kernel. kGuard can safeguard 32- or 64-bit OSes that map a mixture of code segments with different privileges inside the same scope and are susceptible to ret2usr attacks. We believe that it strikes a balance between safety and functionality, and provides comprehensive protection, as demonstrated by our extensive evaluation with real exploits against Linux.

In addition, we presented ret2dir: a novel kernel exploitation technique that takes advantage of direct-mapped physical memory regions in kernel space to bypass deployed ret2usr protections. To improve kernel isolation, we designed and implemented XPFO: an exclusive page frame ownership scheme for the Linux kernel, which prevents the implicit sharing of physical memory, thereby ensuring that user-controlled content can no longer be injected into kernel space using ret2dir. The results of our experimental evaluation demonstrate that XPFO offers effective protection with negligible runtime overhead.

6.2 Future Directions

As commodity software is increasingly used in mission-critical settings, improving the *trustworthiness* of key elements in the software stack becomes an issue of utmost importance. To this end, we introduced new techniques to protect essential software components, like OS kernels (kGuard [115, 114], XPFO [113]). However, the bulk of the work still lies ahead. Going forward, we plan to address the security challenges of the unique software ecosystem that emerging technologies, like mobile and cloud computing, have started shaping. The following research directions capture our vision towards next-generation software protection.

6.2.1 Agile Software Hardening

Despite years of research, software hardening still revolves around the concept of defense in depth. The basic premise behind this strategy to protection is that by stacking together different exploit mitigation techniques, software compromise becomes harder. However, this "blanket" approach to software security (*i.e.,* "protect everything, the same way, all the time, at the same intensity") works well only with defences that have negligible, or close to zero, performance overhead and are oblivious to the setting(s) in which the hardened software is used. Hence, as security-critical components, like OS kernels, language runtime environments, cryptographic libraries, *etc.*, are utilized verbatim in a multitude of diverse domains—ranging from mobile settings to cloud computing environments—software providers are coerced to employ "cheap" exploit mitigations with lax security guarantees, rather than modern protection primitives with assured security; the latter typically inflict non-negligible runtime slowdowns or operate only on platforms with specific characteristics. Consequently, the majority of the software stack remains armored with rudimentary broad-spectrum defense schemes, such as ASLR, NX memory, and compiler-based tripwires (*e.g.,* stack canaries, memory poisoning), despite the latest advances in the development of protections with guaranteed security properties.

Looking forward, we believe that we should rethink how we harden our systems. As essential software components move rapidly from one domain to another to meet the demands of emerging software ecosystems (*i.e.,* app stores and mashup cloud applications), which rely heavily on software of unknown quality and provenance, the problems of the blanket approach will only be exacerbated. The preclusion of every protection mechanism that does not align with the "always-on, applicable-everywhere" mode of operation culminates in a monoculture of defenses (*i.e.,* the same exploit mitigations are used repeatedly in environments with fundamentally different security requirements and characteristics), which allows adversaries to automate their techniques for bypassing protections and reliably exploit software running in dissimilar settings with minimal (additional) effort.

To address these challenges, we envision a next-generation security architecture that enables defenses to be constantly in flux, fostering the deployment and use of protection mechanisms like Data Flow Tracking (DFT) [112]—heavyweight, yet principled and very effective—as and where needed in a strategic manner. Our goal is to infuse systems with the ability to dynamically change their defenses along several dimensions, by elevating hardening *rectification* and *agility* to first-rate principles. The benefits of such an approach to software security are manifold. First, hardening agility creates a diversified and unpredictable environment, which naturally breaks the monoculture of defenses and hinders the ability of adversaries to use standard recipes for bypassing exploit mitigations. Second, hardening rectification allows software to make the best use of the hardening capabilities that a particular setting offers, and dynamically adapt the deployed defences to meet certain needs. For instance, intensify the hardening level to protect against an active exploitation attempt, or turn off redundant protections to increase performance and preserve power.

To support this effort, we plan to build upon our prior work on Virtual Application Partitioning [86] and investigate compiler-assisted techniques that efficiently, and transparently (to the extent possible), enable systems to adapt the *(a.)* granularity (coarse-grain vs. fine-grain), *(b.)* intensity, and *(c.)* layering, of the deployed protection mechanisms, based on the *(i)* environment in which the software is executing (*e.g.,* mobile device, cloud platform), *(ii)* data that the application operates upon (high-value vs. low-value), *(iii)* innate properties of particular code parts (high-privileged vs. deprivileged/sandboxed), and *(iv)* external events. In addition, we plan to explore minimal hardware extensions for boosting certain kinds of heavyweight runtime protections, such as CFI and DFT [112, 103]. Finally, in the longer term, our aim is to leverage automated program verification techniques for applying defense mechanisms strategically, in a very fine-grained and targeted manner (*e.g.,* only in code parts that certain safety properties cannot be proven to hold, or only when an application operates on inputs that are proven to be unsafe), in a spirit similar to that of Taint Flow Algebra [104].

6.2.2 Kernel Security

Kernel protection is a fertile and challenging research area that we plan to continue working on, as kernel exploitation is currently turning into one of the primary methods for compromising (hardened) systems. ret2usr [115, 114] and ret2dir [113]—the two attacks we introduced as part of this dissertation—signified the importance of proper isolation between the OS kernel and user processes, by demonstrating how the weak separation of different protection domains (*i.e.,* kernel space vs. user space) can be leveraged to completely subvert state-of-the-art kernel defenses. Going forward, we plan to explore kernel designs, and hardware extensions, which can safely combine different protection domains without trading security for performance, or vice versa.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proc. of CCS*, pages 340–353, 2005.

[2] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of USENIX Summer*, pages 93–113, 1986.

[3] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[4] Patroklos Argyroudis. Binding the Daemon: FreeBSD Kernel Stack and Heap Exploitation. In *Black Hat USA*, 2010.

[5] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. In *Proc. of EuroSys*, pages 187–198, 2009.

[6] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proc. of CCS*, pages 38–49, 2010.

[7] Dennis Batchelder, Joe Blackbird, David Felstead, Paul Henry, Jeff Jones, Aneesh Kulkarni, John Lambert, Marc Lauricella, Ken Malcolmson, Matt Miller, Nam Ng, Daryl Pecelj, Tim Rains, Vidya Sekhar, Holly Stewart, Todd Thompson, David Weston, and Terry Zink. How Vulnerabilities are Exploited. *Microsoft Security Intelligence Report*, 16:6–8, December 2013.

[8] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proc. of USENIX ATC*, pages 41–46, 2005.

[9] Konstantin Belousov. CVS-200910282103. `https://svnweb.freebsd.org/base?view=revision&revision=242433`, November 2012. [Online; accessed June-2015].

[10] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proc. of USENIX Summer*, pages 87–98, 1994.

[11] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, chapter System Startup, pages 835–841. OReilly Media, 3rd edition, 2005.

[12] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, chapter Memory Management, pages 294–350. OReilly Media, 3rd edition, 2005.

[13] David Brash. The ARMv8-A architecture and its ongoing development. `https://community.arm.com/groups/processors/blog/2014/12/02/the-armv8-a-architecture-and-its-ongoing-development`, December 2014. [Online; accessed June-2015].

[14] Javier Martinez Canillas. Kbuild: the Linux Kernel Build System. *Linux Journal*, 2012(224), December 2012.

[15] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proc. of USENIX Sec*, pages 385–399, 2014.

[16] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *Proc. of CCS*, pages 559–572, 2010.

[17] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. of APsys*, pages 51–55, 2011.

[18] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *Proc. of SOSP*, pages 73–88, 2001.

[19] Chromium OS. Linux Sandboxing. `https://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design`. [Online; accessed June-2015].

[20] Chromium OS. OSX Sandboxing. `https://code.google.com/p/chromium/wiki/LinuxSandboxing`. [Online; accessed June-2015].

[21] Chromium OS. Sandbox. `https://www.chromium.org/developers/design-documents/sandbox`. [Online; accessed June-2015].

[22] Common Vulnerabilities and Exposures. CVE-2005-0736, March 2005.

[23] Common Vulnerabilities and Exposures. CVE-2009-1527, May 2009.

[24] Common Vulnerabilities and Exposures. CVE-2009-1897, June 2009.

[25] Common Vulnerabilities and Exposures. CVE-2009-2692, August 2009.

[26] Common Vulnerabilities and Exposures. CVE-2009-2698, August 2009.

[27] Common Vulnerabilities and Exposures. CVE-2009-2908, August 2009.

[28] Common Vulnerabilities and Exposures. CVE-2009-3002, August 2009.

[29] Common Vulnerabilities and Exposures. CVE-2009-3234, September 2009.

[30] Common Vulnerabilities and Exposures. CVE-2009-3527, October 2009.

[31] Common Vulnerabilities and Exposures. CVE-2009-3547, October 2009.

[32] Common Vulnerabilities and Exposures. CVE-2010-2959, August 2010.

[33] Common Vulnerabilities and Exposures. CVE-2010-3437, September 2010.

[34] Common Vulnerabilities and Exposures. CVE-2010-3904, October 2010.

[35] Common Vulnerabilities and Exposures. CVE-2010-4073, October 2010.

[36] Common Vulnerabilities and Exposures. CVE-2010-4258, November 2010.

[37] Common Vulnerabilities and Exposures. CVE-2010-4347, November 2010.

[38] Common Vulnerabilities and Exposures. CVE-2012-0946, February 2012.

[39] Common Vulnerabilities and Exposures. CVE-2013-0268, December 2013.

[40] Common Vulnerabilities and Exposures. CVE-2013-1828, February 2013.

[41] Common Vulnerabilities and Exposures. CVE-2013-2094, February 2013.

[42] Common Vulnerabilities and Exposures. CVE-2013-2852, April 2013.

[43] Common Vulnerabilities and Exposures. CVE-2013-2892, August 2013.

[44] Common Vulnerabilities and Exposures. CVE-2013-4343, June 2013.

[45] Jonathan Corbet. Virtual Memory I: the problem. `http://lwn.net/Articles/75174/`, March 2004. [Online; accessed June-2015].

[46] Jonathan Corbet. An updated guide to `debugfs`. `http://lwn.net/Articles/334546/`, May 2009. [Online; accessed June-2015].

[47] Jonathan Corbet. How many page flags do we really have? `http://lwn.net/Articles/335768/`, June 2009. [Online; accessed June-2015].

[48] Jonathan Corbet. On vsyscalls and the vDSO. `http://lwn.net/Articles/446528/`, June 2011. [Online; accessed June-2015].

[49] Jonathan Corbet. Supervisor mode access prevention. `http://lwn.net/Articles/517475/`, October 2012. [Online; accessed June-2015].

[50] Jonathan Corbet. The first kpatch submission. `https://lwn.net/Articles/597407/`, May 2014. [Online; accessed June-2015].

[51] Jonathan Corbet. The initial kGraft submission. `https://lwn.net/Articles/596854/`, April 2014. [Online; accessed June-2015].

[52] Jonathan Corbet. A rough patch for live patching. `https://lwn.net/Articles/634649/`, February 2015. [Online; accessed June-2015].

[53] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. Linux Kernel Development. Technical report, Linux Foundation, September 2013.

[54] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, April 2015.

[55] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of USENIX Sec.*, pages 63–78, 1998.

[56] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proc. of IEEE S&P*, pages 292–307, 2014.

[57] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proc. of USENIX Sec*, pages 401–416, 2014.

[58] Theo de Raadt. CVS-200910282103. `http://marc.info/?l=openbsd-cvs&m=125676466108709&w=2`, October 2009. [Online; accessed June-2015].

[59] Debian. "squeeze" Release Information. `https://www.debian.org/releases/squeeze/`, February 2011. [Online; accessed June-2015].

[60] Debian. "wheezy" Release Information. `https://www.debian.org/releases/wheezy/`, May 2013. [Online; accessed June-2015].

[61] Debian Wiki. `mmap_min_addr`. `https://wiki.debian.org/mmap_min_addr`, May 2011. [Online; accessed June-2015].

[62] Yu Ding, Tao Wei, TieLei Wang, Zhenkai Liang, and Wei Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks. In *Proc. of ACSAC*, pages 327–336, 2010.

[63] DistroWatch. Put the fun back into computing. Use Linux, BSD. `http://distrowatch.com`, November 2013. [Online; accessed June-2015].

[64] DOSEMU. DOS Emulation. `http://www.dosemu.org`. [Online; accessed June-2015].

[65] Mark Dowd. Application-Specific Attacks: Leveraging The ActionScript Virtual Machine. Technical report, IBM Corporation, April 2008.

[66] Jake Edge. Kernel address space layout randomization. `http://lwn.net/Articles/569635/`, October 2013. [Online; accessed June-2015].

[67] Sinan Eren. Smashing The Kernel Stack For Fun And Profit. *Phrack*, 6(60), December 2002.

[68] Exploit Database. EBD-131, December 2003.

[69] Exploit Database. EBD-16835, September 2009.

[70] Exploit Database. EDB-9477, August 2009.

[71] Exploit Database. EBD-14814, August 2010.

[72] Exploit Database. EBD-15150, September 2010.

[73] Exploit Database. EBD-15285, October 2010.

[74] Exploit Database. EBD-15704, December 2010.

[75] Exploit Database. EBD-15916, January 2011.

[76] Exploit Database. EBD-17391, June 2011.

[77] Exploit Database. EBD-17787, September 2011.

[78] Exploit Database. EBD-20201, August 2012.

[79] Exploit Database. EBD-24555, February 2013.

[80] Free Software Foundation. GNU General Public License. `https://www.gnu.org/licenses/gpl.html`, June 2007. [Online; accessed June-2015].

[81] FreeBSD. sysutils/vbetool doesn't work with FreeBSD 8.0-RELEASE and STABLE. `http://forums.freebsd.org/showthread.php?t=12889`. [Online; accessed June-2015].

[82] freedesktop.org. `libbsd`. `https://wiki.freedesktop.org/libbsd/`, July 2014. [Online; accessed June-2015].

[83] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. of NDSS*, February 2003.

[84] GCC. 4.5 Release Series. `https://gcc.gnu.org/gcc-4.5/`, April 2010. [Online; accessed June-2015].

[85] GCC. 4.6 Release Series. `https://gcc.gnu.org/gcc-4.6/`, March 2011. [Online; accessed June-2015].

[86] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. Adaptive Defenses for Commodity Software through Virtual Application Partitioning. In *Proc. of CCS*, pages 133–144, 2012.

[87] Varghese George, Tom Piazza, and Hong Jiang. Technology Insight: Intel©Next Generation Microarchitecture Codename Ivy Bridge. `http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf`, September 2011. [Online; accessed June-2015].

[88] GNU. Make. `https://www.gnu.org/software/make/`, April 2006. [Online; accessed June-2015].

[89] GNU. patch. `http://savannah.gnu.org/projects/patch/`, December 2009. [Online; accessed June-2015].

[90] GNU. Binutils. `https://www.gnu.org/software/binutils/`, November 2011. [Online; accessed June-2015].

[91] GNU. GDB: The GNU Project Debugger. `https://www.gnu.org/software/gdb/`, April 2012. [Online; accessed June-2015].

[92] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proc. of IEEE S&P*, pages 575–589, 2014.

[93] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *Proc. of USENIX Sec*, pages 417–432, 2014.

[94] Google. Android. `http://www.android.com`. [Online; accessed June-2015].

[95] Google. Chromium OS. `http://www.chromium.org/chromium-os`. [Online; accessed June-2015].

[96] Julian Bennett Grizzard. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, 2006.

[97] Norm Hardy. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22:36–38, October 1988.

[98] Ben Hayak. The Kernel is calling a zero(day) pointer - CVE-2013-5065 - Ring Ring. `http://blog.spiderlabs.com/2013/12/the-kernel-is-calling-a-zeroday-pointer-cve-2013-5065-ring-ring.html`, December 2013. [Online; accessed June-2015].

[99] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006.

[100] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proc. of USENIX Sec*, pages 384–398, 2009.

[101] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[102] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *Proc. of IEEE S&P*, pages 143–157, 2012.

[103] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking. In *Proc. of CCS*, pages 235–246, 2013.

[104] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In *Proc. of NDSS*, 2012.

[105] Jolla. Sailfish OS. `https://sailfishos.org`. [Online; accessed June-2015].

[106] Paul A. Karger and Andrew J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proc. of IEEE S&P*, pages 2–12, 1984.

[107] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proc. of CCS*, pages 272–280, 2003.

[108] Sylvester Keil and Clemens Kolbitsch. Kernel-mode exploits primer. Technical report, International Secure Systems Lab (isecLAB), November 2007.

[109] Vasileios P. Kemerlis. kGuard (Source Code). `https://www.cs.columbia.edu/~vpk/research/kguard/`, August 2012. [Online; accessed June-2015].

[110] Vasileios P. Kemerlis. Return-to-direct-mapped memory (ret2dir) Exploitation Kit. `https://www.cs.columbia.edu/~vpk/research/ret2dir/`, August 2014. [Online; accessed June-2015].

[111] Vasileios P. Kemerlis. XPFO (Linux Kernel Patch). `https://www.cs.columbia.edu/~vpk/research/xpfo/`, August 2014. [Online; accessed June-2015].

[112] Vasileios P. Kemerlis, Kangkook Jee, Georgios Portokalidis, and Angelos D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proc. of VEE*, pages 121–132, 2012.

[113] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proc. of USENIX Sec*, pages 957–972, 2014.

[114] Vasileios P. Kemerlis, Georgios Portokalidis, Elias Athanasopoulos, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection. *USENIX ;login:*, 37(6):7–14, December 2012.

[115] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proc. of USENIX Sec*, pages 459–474, 2012.

[116] Thomas J. Killian. Processes as Files. In *Proc. of USENIX Summer*, pages 203–207, 1984.

[117] Russell King. Kernel Memory Layout on ARM Linux. `https://www.kernel.org/doc/Documentation/arm/memory.txt`, November 2005. [Online; accessed June-2015].

[118] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution via Program Shepherding. In *Proc. of USENIX Sec*, pages 191–206, 2002.

[119] Andi Kleen. Memory Layout on amd64 Linux. `https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt`, July 2004. [Online; accessed June-2015].

[120] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of SOSP*, pages 207–220, 2009.

[121] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[122] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architecture Support for Single Address Space Operating Systems. In *Proc. of ASPLOS*, pages 175–186, 1992.

[123] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proc. of NDSS*, 2013.

[124] Christoph Lameter. Generic Virtual Memmap support for `SPARSEMEM` v3. `http://lwn.net/Articles/229670/`, April 2007. [Online; accessed June-2015].

[125] Per Larsen, Stefan Brunthaler, and Michael Franz. Security through Diversity: Are We There Yet? *IEEE Security and Privacy*, 12(2):28–35, March 2014.

[126] Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace, and Sina Bahram. Defeating Return-Oriented Rootkits With "*Return-less*" Kernels. In *Proc. of EuroSys*, pages 195–208, 2010.

[127] Siarhei Liakh. NX protection for kernel data. `http://lwn.net/Articles/342266/`, July 2009. [Online; accessed June-2015].

[128] Siarhei Liakh, Michael Grace, and Xuxian Jiang. Analyzing and Improving Linux Kernel Memory Protection: A Model Checking Approach. In *Proc. of ACSAC*, pages 271–280, 2010.

[129] Jochen Liedtke. On μ-Kernel Construction. In *Proc. of SOSP*, pages 237–250, 1984.

[130] ARM Limited. *ARMv8-A Architecture Reference Manual*, March 2015.

[131] Linux Documentation. `pagemap`, from the userspace perspective. `https://www.kernel.org/doc/Documentation/vm/pagemap.txt`, December 2008. [Online; accessed June-2015].

[132] Linux Foundation. Tizen. `https://www.tizen.org`. [Online; accessed June-2015].

[133] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Software Self-Healing Using Collaborative Application Communities. In *Proc. of NDSS*, 2006.

[134] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proc. of USENIX ATC (FREENIX Track)*, pages 29–42, 2001.

[135] Robert Love. *Linux Kernel Development*. Novel Press, 2^{nd} edition, 2005.

[136] Mac Developer Library. About App Sandbox. `https://developer.apple.com/library/mac/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html`. [Online; accessed June-2015].

[137] Catalin Marinas. arm64: Distinguish between user and kernel XN bits. `http://permalink.gmane.org/gmane.linux.kernel.commits.head/348605`, November 2012. [Online; accessed June-2015].

[138] Catalin Marinas. Memory Layout on AArch64 Linux. `https://www.kernel.org/doc/Documentation/arm64/memory.txt`, February 2012. [Online; accessed June-2015].

[139] Richard McDougall and Jim Mauro. *Solaris Internals*, chapter Kernel Virtual Address Maps, pages 965–969. Prentice Hall, 2^{nd} edition, 2006.

[140] Richard McDougall and Jim Mauro. *Solaris Internals*, chapter File System Framework, pages 710–710. Prentice Hall, 2^{nd} edition, 2006.

[141] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*, chapter Overview of the FreeBSD Virtual-Memory System, pages 227–230. Pearson Education, 2^{nd} edition, 2015.

[142] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*, chapter Kernel Boot, pages 782–796. Pearson Education, 2^{nd} edition, 2015.

[143] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. of USENIX ATC*, pages 279–294, 1996.

[144] Microsoft. Enhanced Mitigation Experience Toolkit. `https://technet.microsoft.com/en-us/security/jj653751`. [Online; accessed June-2015].

[145] Microsoft. Microsoft Windows. `http://windows.microsoft.com`. [Online; accessed June-2015].

[146] Microsoft Developer Network. /guard (Enable Control Flow Guard). `https://msdn.microsoft.com/en-us/library/dn919635%28v=vs.140%29.aspx`. [Online; accessed June-2015].

[147] Microsoft Developer Network. Managing Virtual Memory. `https://msdn.microsoft.com/en-us/library/ms810627.aspx`. [Online; accessed June-2015].

[148] Microsoft Developer Network. Memory Limits for Windows and Windows Server Releases. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778(v=vs.85).aspx`. [Online; accessed June-2015].

[149] Microsoft Developer Network. Understanding and Working in Protected Mode Internet Explorer. `https://msdn.microsoft.com/en-us/library/bb250462%28v=vs.85%29.aspx`. [Online; accessed June-2015].

[150] Microsoft Developer Network. Virtual Address Space. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366912%28v=vs.85%29.aspx`. [Online; accessed June-2015].

[151] MoKB. Broadcom Wireless Driver Probe Response SSID Overflow, November 2006.

[152] Ingo Molnar. 4G/4G split on x86, 64 GB RAM (and more) support. `http://lwn.net/Articles/39283/`, July 2003. [Online; accessed June-2015].

[153] Mozilla. Firefox OS. `https://www.mozilla.org/en-US/firefox/os/2.0/`. [Online; accessed June-2015].

[154] MozillaWiki. Security/Sandbox. `https://wiki.mozilla.org/Security/Sandbox`. [Online; accessed June-2015].

[155] National Vulnerability Database. Kernel Vulnerabilities. `https://web.nvd.nist.gov/view/vuln/statistics-results?cves=on&query=kernel&cwe_id=&pub_date_start_month=-1&pub_date_start_year=-1&pub_date_end_month=-1&pub_date_end_year=-1&mod_date_start_month=-1&mod_date_start_year=-1&mod_date_end_month=-1&mod_date_end_year=-1&cvss_sev_base=&cvss_av=&cvss_ac=&cvss_au=&cvss_c=&cvss_i=&cvss_a=`, July 2014. [Online; accessed June-2015].

[156] Kernel Newbies. KernelBuild. `http://kernelnewbies.org/KernelBuild`, April 2013. [Online; accessed June-2015].

[157] Nils and Jon. Polishing Chrome for Fun and Profit. `https://labs.mwrinfosecurity.com/system/assets/538/original/mwri_polishing-chrome-slides-nsc_2013-09-06.pdf`, August 2013. [Online; accessed June-2015].

[158] Offensive Security. The Exploit Database. `http://www.exploit-db.com`. [Online; accessed June-2015].

[159] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries. In *Proc. of ACSAC*, pages 49–58, 2010.

[160] Open Web Application Security Project. C-Based Toolchain Hardening. `https://www.owasp.org/index.php/C-Based_Toolchain_Hardening#Compiler_and_Linker`. [Online; accessed June-2015].

[161] OpenBSD. Changes made between OpenBSD 5.2 and 5.3. `http://www.openbsd.org/plus53.html`, May 2013. [Online; accessed June-2015].

[162] OpenBSD. `pmap` – machine dependent interface to the MMU. `http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man9/pmap.9?query=pmap&sec=9`, June 2015. [Online; accessed June-2015].

[163] Oracle Corporation. Oracle Solaris. `http://www.oracle.com/solaris`. [Online; accessed June-2015].

[164] The Linux Kernel Organization. Linux Kernel v3.13 (Source Code). `https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.13.tar.gz`, January 2014. [Online; accessed June-2015].

[165] Tavis Ormandy and Julien Tinnes. Linux ASLR Curiosities. In *CanSecWest*, 2009.

[166] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *Proc. of ASPLOS*, pages 305–318, 2011.

[167] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proc. of USENIX Sec*, pages 447–462, 2013.

[168] PaX. Homepage of The PaX Team. `http://pax.grsecurity.net`. [Online; accessed June-2015].

[169] PaX Team. UDEREF/i386. `http://grsecurity.net/~spender/uderef.txt`, April 2007. [Online; accessed June-2015].

[170] PaX Team. UDEREF/amd64. `http://grsecurity.net/pipermail/grsecurity/2010-April/001024.html`, April 2010. [Online; accessed June-2015].

[171] PaX Team. Better kernels with GCC plugins. `http://lwn.net/Articles/461811/`, October 2011. [Online; accessed June-2015].

[172] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proc. of IEEE S&P*, pages 233–247, 2008.

[173] Enrico Perla and Massimiliano Oldani. *A Guide To Kernel Exploitation: Attacking the Core*, chapter Stairway to Successful Kernel Exploitation, pages 47–99. Elsevier, 2010.

[174] Nick L. Petroni, Jr. and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proc. of CCS*, pages 103–115, 2007.

[175] Gerald J. Popek and David A. Farber. A Model for Verification of Data Security in Operating Systems. *Commun. ACM*, 21(9):737–749, September 1978.

[176] Niels Provos. Improving Host Security with System Call Policies. In *Proc. of USENIX Sec*, pages 257–272, 2003.

[177] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proc. of USENIX Sec*, pages 231–242, 2003.

[178] PTS. Phoronix Test Suite. `http://www.phoronix-test-suite.com`, February 2014. [Online; accessed June-2015].

[179] Ramon de Carvalho Valle & Packet Storm. `sock_sendpage()` NULL pointer dereference (PPC/PPC64 exploit). `http://packetstormsecurity.org/files/81212/Linux-sock_sendpage-NULL-Pointer-Dereference.html`, September 2009. [Online; accessed June-2015].

[180] RedHat. How do i mitigate against null pointer dereference vulnerabilities? `https://access.redhat.com/articles/20484`, May 2012. [Online; accessed June-2015].

[181] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proc. of RAID*, pages 1–20, 2008.

[182] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. of USENIX ATC*, pages 101–112, 2012.

[183] Dan Rosenberg. `kptr_restrict` for hiding kernel pointers. `http://lwn.net/Articles/420403/`, December 2010. [Online; accessed June-2015].

[184] Dan Rosenberg. Owned Over Amateur Radio: Remote Kernel Exploitation in 2011. In *Proc. of DEF CON®*, 2011. `http://lwn.net/Articles/420403/`.

[185] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proc. of the IEEE*, 63(9):1278–1308, September 1975.

[186] Michael D. Schroeder and Jerome H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Commun. ACM*, 15(3):157–170, March 1972.

[187] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proc. of USENIX Sec*, pages 379–394, August 2011.

[188] Kevin Scott and Jack Davidson. Safe Virtual Execution Using Software Dynamic Translation. In *Proc. of ACSAC*, pages 209–218, 2002.

[189] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat USA*, 2015.

[190] SecurityFocus. BID 36587, October 2009.

[191] SecurityFocus. BID 36939, November 2009.

[192] SecurityFocus. Linux Kernel 'perf_counter_open()' Local Buffer Overflow Vulnerability, September 2009.

[193] SecurityFocus. BID 43060, September 2010.

[194] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proc. of SOSP*, pages 335–350, 2007.

[195] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of CCS*, pages 552–61, 2007.

[196] Stelios Sidiroglou, Oren Laadan, Carlos R. Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. ASSURE: Automatic Software Self-healing Using REscue points. In *Proc. of ASPLOS*, pages 37–48, 2009.

[197] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness

of Fine-Grained Address Space Layout Randomization. In *Proc. of IEEE S&P*, pages 574–588, 2013.

[198] Brad Spengler. On exploiting null ptr derefs, disabling SELinux, and silently fixed linux vulns. `http://seclists.org/dailydave/2007/q1/224`, March 2007. [Online; accessed June-2015].

[199] Brad Spengler. Recent ARM security improvements. `https://forums.grsecurity.net/viewtopic.php?f=7&t=3292`, February 2013. [Online; accessed June-2015].

[200] Brad Spengler. Enlightenment Linux Kernel Exploitation Framework. `https://grsecurity.net/~spender/exploits/enlightenment.tgz`, June 2015. [Online; accessed June-2015].

[201] sqrkkyu and twiz. Attacking the Core: Kernel Exploiting Notes. *Phrack*, 6(64), May 2007.

[202] Ars Technica. Better on the inside: under the hood of Windows 8. `http://arstechnica.com/information-technology/2012/10/better-on-the-inside-under-the-hood-of-windows-8/3/`, October 2012. [Online; accessed June-2015].

[203] The FreeBSD Foundation. FreeBSD. `https://www.freebsd.org`. [Online; accessed June-2015].

[204] The FreeBSD Project. No zero mapping feature. `https://www.freebsd.org/security/advisories/FreeBSD-EN-09:05.null.asc`, October 2009. [Online; accessed June-2015].

[205] The NetBSD Foundation. NetBSD. `http://www.netbsd.org`. [Online; accessed June-2015].

[206] The OpenBSD Foundation. OpenBSD. `http://www.openbsd.org`. [Online; accessed June-2015].

[207] Julien Tinnes. Bypassing Linux NULL pointer dereference exploit prevention (mmap_min_addr). `http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html`, June 2009. [Online; accessed June-2015].

[208] Arjan van de Ven. Debug option to write-protect rodata: the write protect logic and config option. `http://lkml.indiana.edu/hypermail/linux/kernel/0511.0/2165.html`, November 2005. [Online; accessed June-2015].

[209] Arjan van de Ven. Add `-fstack-protector` support to the kernel. `http://lwn.net/Articles/193307/`, July 2006. [Online; accessed June-2015].

[210] Arjan van de Ven. x86: add code to dump the (kernel) page tables for visual inspection. `http://lwn.net/Articles/267837/`, February 2008. [Online; accessed June-2015].

[211] Roman Vasilenko. Unmasking Kernel Exploits. `http://labs.lastline.com/unmasking-kernel-exploits`. [Online; accessed June-2015].

[212] VMware. VMware Fusion. `https://www.vmware.com/products/fusion/`. [Online; accessed June-2015].

[213] VMware. VMware Player. `https://www.vmware.com/products/player/`. [Online; accessed June-2015].

[214] VMware. VMware Workstation. `https://www.vmware.com/products/workstation/`. [Online; accessed June-2015].

[215] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proc. of CCS*, pages 545–554, 2009.

[216] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: practical capabilities for UNIX. In *Proc. of USENIX Sec*, pages 29–45, 2010.

[217] WineHQ. Run Windows applications on Linux, BSD, Solaris and Mac OS X. `http://www.winehq.org`. [Online; accessed June-2015].

[218] Xst3nZ. Windows Kernel Exploitation Basics. `http://poppopret.blogspot.com/2011/07/windows-kernel-exploitation-basics-part.html`, July 2011. [Online; accessed June-2015].

[219] Eric Youngdale. The ELF Object File Format by Dissection. *Linux Journal*, May 1995.

[220] Fenghua Yu. Enable/Disable Supervisor Mode Execution Protection. `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=de5397ad5b9ad22e2401c4dacdf1bb3b19c05679`, May 2011. [Online; accessed June-2015].

[221] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proc. of IEEE S&P*, pages 559–573, 2013.

[222] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proc. of USENIX Sec*, pages 337–352, 2013.

Appendix A

Examples of ret2usr Exploits

ret2usr attacks are usually manifested by overwriting kernel-resident control data (*e.g.,* return addresses, jump tables, function pointers) with user-space addresses (see Section 2.2). In early versions of such exploits, this was accomplished by invoking a (vulnerable) system call with carefully crafted arguments to nullify a function pointer. When the `NULL` function pointer is eventually dereferenced, control is transferred to address zero that resides in user space. Commonly, that address is not used by processes and is unmapped.[1] However, if the attacker has local access to the system, she can build a program with arbitrary data or code mapped at address zero (or any other address in her program for that matter). Note that since the attacker controls the program, its memory pages can be mapped both writable and executable, and placed at known locations (*i.e.,* kernel-space ASLR and `W^X` measures do not apply).

[1]In Linux, accessing an unmapped page, when running in kernel mode, results into a *kernel oops* and subsequently causes the OS to kill the offending process (unless a lock is held, in which case Linux will result into a *kernel panic*). Other OSes fail-stop directly with a kernel panic.

A.1 **sendpage** ret2usr Linux Exploit

```
736 sock  = file->private_data;
737 flags = !(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
738 if (more)
739         flags |= MSG_MORE;
740 /*[!] NULL pointer dereference (sendpage) [!]*/
741 return sock->ops->sendpage(sock, page, offset, size, flags);
```

Listing A.1: `NULL` *function* pointer in Linux (*net/socket.c*).

Listing A.1 shows a straightforward `NULL` function pointer vulnerability that affected all Linux kernel versions released between May 2001 and August 2009 (v2.4.4/v2.6.0 – v2.4.37/-v2.6.30.4) [25]. If the `sendfile` system call is invoked with a socket descriptor belonging to a vulnerable protocol family, the value of the `sendpage` pointer in line 741 is set to `NULL`. This results in an indirect function call to address zero, which can be exploited by attackers to execute arbitrary code with elevated privileges.

Figure A.1 illustrates the steps taken by a malicious process to exploit the vulnerability in x86. It starts by invoking the `sendfile` system call with the offending arguments (*i.e.,* a datagram socket of a vulnerable protocol family, such as `PF_IPX`). The corresponding `libc` wrapper (`0xB7F50D20`) traps to the OS via the `SYSENTER` instruction (`0xB7FE2419`) and the generated software interrupt leads to executing the system call handler of Linux (`sysenter_do_call()`). The handler dynamically resolves the address of `sys_sendfile` (`0xC01D0CCF`) using the array `sys_call_table`, which includes the kernel address of every supported system call, indexed by system call number (`0xC01039DB`).[2] Privileged execution then continues until the offending `sock_sendpage()` routine is invoked. Due to the arguments passed in `sendfile`, the value of the `sendpage` pointer (`0xFA7C8538`) is `NULL` and results in an indirect function call to address zero. This transfers control to the attacker, who can execute arbitrary code with kernel privilege.

[2]The address `0xC03FD3A8` corresponds to the kernel-memory address of `sys_call_table`.

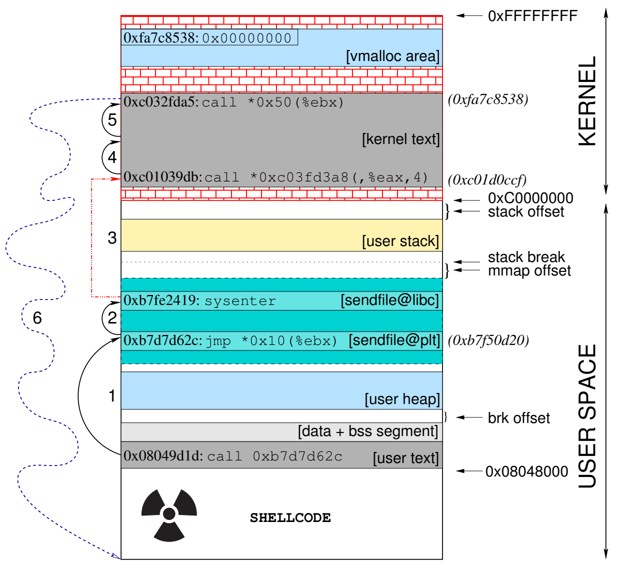


Figure A.1: Control transfers that occur during the exploitation of a ret2usr attack. The `sendfile` system call, on x86 Linux, causes a function pointer in kernel to become `NULL`, illegally transferring control to user space code.

A.2 `tee` ret2usr Linux Exploit

```
/*[!] NULL pointer dereference (ops) [!]*/
ibuf->ops->get(ipipe, ibuf);
obuf  = opipe->bufs + nbuf;
*obuf = *ibuf;
```

Listing A.2: NULL *data* pointer in Linux (*fs/splice.c*).

```
struct pipe_buf_operations {
        int     can_merge;
        void*   (*map)(/* ... */);
        void    (*unmap)(/* ... */);
        int     (*pin)(/* ... */);
        void    (*release)(/* ... */);
        int     (*steal)(/* ... */);
        void    (*get)(/* ... */);
};
```

Listing A.3: `struct pipe_buf_operations` (*include/linux/pipe_fs_i.h*).

Listing A.2 shows the Linux kernel bug exploited by Spengler [198]. The `ops` field in line 1334, which is a data pointer of type `struct pipe_buf_operations`, becomes `NULL` after the invocation of the `tee` system call. Upon dereferencing `ops`, the effective address of a function is obtained via `get`, which is mapped to the seventh double word (assuming an x86 architecture) after the address pointed by `ops` (*i.e.,* due to the definition of structure `pipe_buf_operations`; see Listing A.3). Hence, the kernel reads a code pointer from `0x0000001C`, *which is controlled by the user.* This enables an attacker to redirect the control flow of the kernel to an arbitrary address.

Appendix B

kGuard RTL Internals

`branchprot_instrument()`, our instrumentation callback, is invoked by the GCC pass manager for every translation unit after all RTL optimizations have been applied, and exactly before the target code is emitted. At this point, the corresponding translation unit is maintained as graph of basic blocks (BBs) that contain chained sequences of RTL instructions, also known as `rtx` expressions (*i.e.,* LISP-like assembler code for an abstract machine with infinite registers).

GCC maintains a specific graph-based data structure (`call-graph`) that holds information for every internal/external call site. However, indirect control transfers are not represented in it and are assumed to be control-flow neutral. For that reason we perform the following. We begin by iterating over all the BBs and `rtx` expressions of the respective translation unit, selecting only the computed calls and jumps. This includes `rtx` objects of type `CALL_INSN` or `JUMP_INSN` that branch via a register or memory location. Note that `ret` instructions are also encoded as `rtx` objects of type `JUMP_INSN`. Next, we modify the `rtx` expression stream for inserting the $\texttt{CFA}_\texttt{R}$ and $\texttt{CFA}_\texttt{M}$ checks. The $\texttt{CFA}_\texttt{R}$ guards are inserted by splitting the original BB into 3 new ones. The first hosts all the `rtx` expressions before {`CALL`, `JUMP`}`_INSN`, along with the random `NOP` sled and two more `rtx` expressions that match the compare (`cmp`) and jump (`jae`) instructions shown in Listing 3.1. The second BB contains the code for loading the address of the violation handler into the branch register (*i.e.,* `mov` in x86), while the last BB contains the actual branch expression along with the remaining `rtx` expressions of the original BB.

Lastly, the process also involves altering the control-flow graph, by chaining the new BBs accordingly and inserting the proper branch labels to ensure that the injected code remains inlined. $\texttt{CFA}_{\texttt{M}}$ instrumentation is performed in a similar fashion.

Appendix C

kGuard Distribution

Our kGuard bundle [109] contains the following tools:

- **`kguard:`** a GCC plugin that implements *CFA-based control-flow confinement* and *code inflation*, as specified in Section 3.2, Section 3.3, and Appendix B.
- **`kvld:`** a helper tool that detects and reports *unprotected* (exploitable) control transfer instructions in ELF [219] objects, along with a `kguard` instrumentation summary.

Although kGuard is a cross-platform solution, both `kguard` and `kvld` were mainly evaluated in x86/x86-64 Linux. Earlier versions were successfully used with members of the BSD family and the ARM platform (AArch32), but our latest release was solely developed and tested on Debian GNU/Linux v6 [59] and v7 [60].

C.1 Limitations

kGuard is implemented as an RTL IR optimization pass (see Section 3.3 and Appendix B), and as such, it does not handle *assembler* code (both "inline" and external). Hence, any indirect control-flow transfer embedded in assembly snippets is left unprotected. Note, however, that this is not a fundamental limitation of kGuard, but rather an implementation decision. In principle, one can incorporate the techniques presented in Section 3.2 (*i.e.,* CFA-based control-flow confinement, code inflation, and CFA motion) in the assembler, instead of the compiler, as they do not require high-level semantics.

In addition, our released prototype does not include support for the following features:

- Liveness analysis for minimizing register spilling in $\texttt{CFA}_\texttt{M}$ guards (see Section 3.2). Even though initial versions of kGuard included support for that feature, we had to significantly rewrite the respective code when we ported `kguard` from GCC v4.5 [84] to v4.6 [85]. Hence, to avoid fragmentation, and given the limited performance benefits of the feature, we opted for a scheme that always spills a fixed register (*i.e.,* `%edi`/`%rdi` in x86/x86-64), as shown in Listing 3.3.

- CFA motion (see Section 3.2.2.2). Although `kguard` supports the `log` parameter, which records all necessary information to perform CFA relocation, it does not handle the actual rewriting of the `.text` segment of the kernel/modules. The latter is performed through a kernel component that is OS- and architecture-specific. In Section 3.2.2.2, we describe, in detail, how to implement CFA motion in x86 Linux (kernel v2.6.32) and {Free, Net}BSD.

- `NOP` sled in the beginning of the kernel `.text` segment. Code inflation randomizes the starting address of the kernel `.text` segment, by inserting a (huge) `NOP` sled of *random length* its beginning (see Section 3.2.2.2). Again, though original versions of `kguard` included support for that feature, we decided to remove it from our latest release, as many commodity OSes begun employing kernel-space ASLR [66].

C.2 Building kGuard

What follows are the steps to build `kguard` from source. We assume a Debian GNU/Linux distribution (v6 [59] or v7 [60]; x86/x86-64) and the latest release of kGuard (v3.14159alpha at the time of writing).

- Dependencies:
  - GNU make v3.81 (or later) [88], GNU binutils v2.22 (or later) [90].
  - GCC v4.5 [84] or v4.6 [85] (versions $\geq$ 4.7 are not supported).
  - libbsd v0.4.2 (or later) [82].

- Build steps:

  1. Download [109]:

     `wget https://www.cs.columbia.edu/~vpk/research/kguard/kguard-src.tar.gz`

  2. Unpack:

     `tar xzf kguard-src.tar.gz`

  3. Build:

     `cd kguard-src && make`

If the build process is successful, `kguard` (`kguard.so`) will be made available in the *working directory*; `kvld` is implemented as an AWK [3] script, and therefore, no compilation is necessary. In Debian and Debian-derived distributions (*e.g.,* Ubuntu), the build dependencies can be satisfied by installing the following packages: `build-essential`, `gcc-4.{5, 6}-plugin-dev`, `libbsd-dev`. Note that we recommend building `kguard` with the same version of GCC used to compile protected kernels, by setting the `CC` and `CFLAGS` variables, accordingly, in `Makefile`. Lastly, `kguard.h` includes the default value(s) for the starting address of the kernel `.text` segment. Hence, if kGuard is used in experimental settings, `KADDR_DFL` needs to be modified accordingly.[1]

C.3 Using kGuard

C.3.1 kguard

Once `kguard` (`kguard.so`) is successfully built, one can start compiling kGuard-protected kernels by leveraging the `-fplugin` and `-fplugin-arg` parameters of GCC. In Linux, which uses the kBuild system [14], we recommend supplying the respective parameters through the `CFLAGS` variable or by directly editing the main `Makefile` of the kernel and altering `KBUILD_CFLAGS`. As stated in Section 3.3, kGuard accepts 3 parameters: `stub` (address or symbol), `nop` (decimal integer), and `log` (file path). `stub` provides the name of the runtime violation handler, `nop` stores the maximum size of the random `NOP` sled inserted

[1] For example, in x86 OpenBSD `KADDR_DFL` should be set to `0xD0000000`.

before each CFA, and `log` is used to define an instrumentation logfile for CFA motion. In addition, our latest prototype supports the `retprot` (boolean) flag for controlling the instrumentation of `ret` instructions (x86/x86-64 only). The default parameter values are the following: `stub = panic`, `nop = 16`, `log = NULL`, and `retprot = 1`.

In common settings, appending '**`-fplugin=<full-path>/kguard.so`**'[2] to the build flags should be sufficient for compiling a kGuard-protected kernel; specific parameters can be further tailored to need with `-fplugin-arg`. For example, the (maximum) size of the `NOP` sleds can be doubled (*i.e.,* from 16 to 32 `NOP`s), and the runtime violation handler can be redefined (*e.g.,* from `panic` to `chndl`), with '**`-fplugin-arg-kguard-nop=32`**' and '**`-fplugin-arg-kguard-stub=chndl`**', respectively. Note that certain (sub)directories of the kernel source tree may have to be excluded from being instrumented with `kguard`. Examples include early, machine-dependent bootstrap code (*e.g.,* `arch/x86/boot/` in x86/x86-64 Linux), the vDSO [48] (*i.e.,* `arch/x86/vdso` in x86-64 Linux), *etc.*

In recent Linux kernels, the above can be performed with the `CFLAGS_REMOVE` macro of kBuild, whereas in all other cases, every kGuard-related directive should be manually stripped from the build flags. The latest Linux kernel that we successfully built with kGuard was v3.2; we appended '**`-fplugin=<full-path>/kguard.so`**' to `KBUILD_CFLAGS` in the main `Makefile`, and the following snippet(s) to `arch/x86/vdso/Makefile`:

```
CFLAGS_REMOVE_vdso-note.o      = -fplugin=<full-path>/kguard.so
CFLAGS_REMOVE_vclock_gettime.o = -fplugin=<full-path>/kguard.so
CFLAGS_REMOVE_vgetcpu.o        = -fplugin=<full-path>/kguard.so
CFLAGS_REMOVE_vvar.o           = -fplugin=<full-path>/kguard.so
```

C.3.2 kvld

`kvld` can be used with any x86/x86-64 kGuard-instrumented ELF file (`kobj`) as follows: **`objdump -d <kobj> | cut -f3 | kvld`**.

[2] `<full-path>`: *absolute path* of the directory that contains `kguard.so`.

Appendix D

ret2dir Distribution

The ret2dir exploits we developed against *hardened* x86/x86-64 Linux targets (see Section 4.6.1; Table 4.2) are all publicly available, primarily serving two purposes:

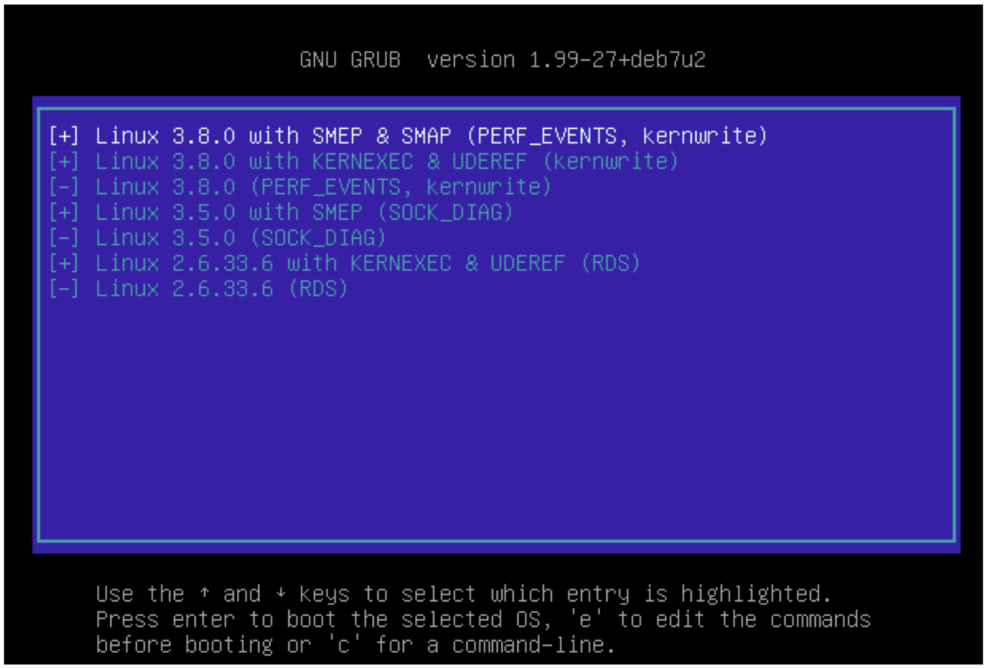
1. Educate the community regarding ret2dir attacks.
2. Provide a testbed for evaluating future ret2dir protection mechanisms.

Our ret2dir distribution [110] contains the following:

- **`ret2dir_{amd64, i386}.vmwarevm`**: two VMs that run atop VMware's hypervisors (Player [213], Workstation [214], Fusion [212]). Both VMs target VMware hardware specification v10 and use Debian GNU/Linux v7 [60] as guest OS.
- **`getpmap`**: a helper tool for querying the `/proc/<pid>/pagemap` interface.

D.1 ret2dir Virtual Machines

`ret2dir_amd64.vmwarevm` is configured to use 2 vCPUs and 2GB of RAM, booting 64-bit kernels, whereas `ret2dir_i386.vmwarevm` is configured to use 2 vCPUs and 4GB of RAM, booting 32-bit kernels, as shown in Figure D.1 and Figure D.2, respectively.



```
GNU GRUB  version 1.99-27+deb7u2

[+] Linux 3.8.0 with SMEP & SMAP (PERF_EVENTS, kernwrite)
[+] Linux 3.8.0 with KERNEXEC & UDEREF (kernwrite)
[-] Linux 3.8.0 (PERF_EVENTS, kernwrite)
[+] Linux 3.5.0 with SMEP (SOCK_DIAG)
[-] Linux 3.5.0 (SOCK_DIAG)
[+] Linux 2.6.33.6 with KERNEXEC & UDEREF (RDS)
[-] Linux 2.6.33.6 (RDS)

Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, 'e' to edit the commands
before booting or 'c' for a command-line.
```

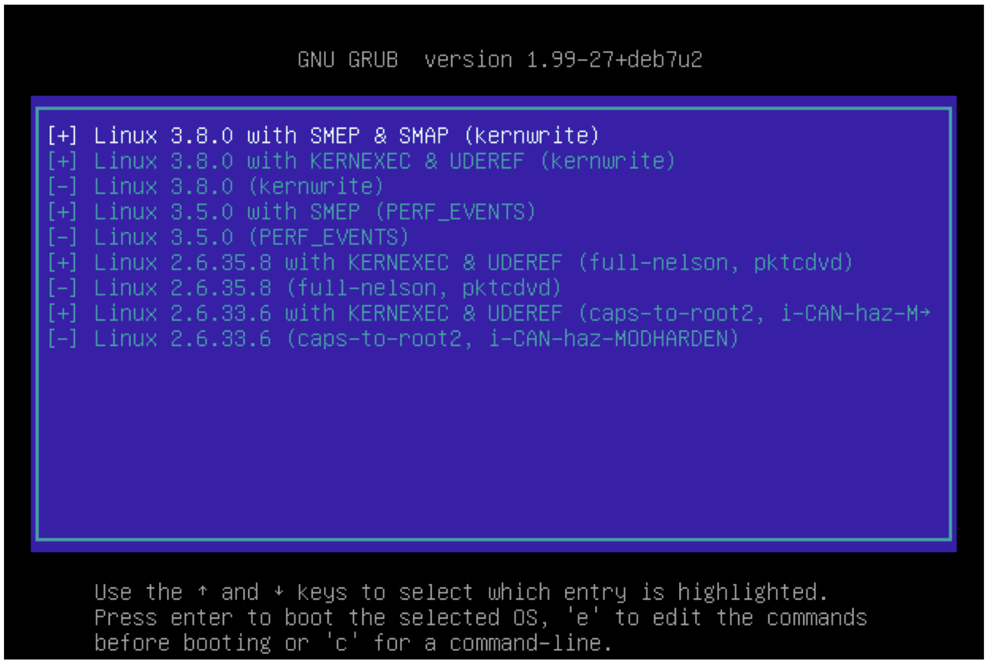
Figure D.1: `ret2dir_amd64.vmwarevm` running over a VMware hypervisor. Entries beginning with "`[-]`" correspond to vanilla Linux kernels without any ret2usr protection. Entries beginning with "`[+]`" indicate Linux kernels hardened against ret2usr attacks—the protection mechanisms enabled are listed after "with". Names in parentheses indicate the exploit(s) that the respective kernel is vulnerable to.

GRUB kernel entries beginning with "`[-]`" correspond to vanilla Linux kernels without any ret2usr protection. Conversely, entries beginning with "`[+]`" correspond to Linux kernels hardened against ret2usr attacks, while the exact ret2usr protection mechanisms enabled are listed after "with". Lastly, names in parentheses indicate the exploit(s) that the respective kernel is vulnerable to.

D.2 ret2dir Exploitation Kit

The credentials to login to the guest OS are the following:

- Username: '**`w00t`**'
- Password: '**`pwn3d`**'



```
GNU GRUB  version 1.99-27+deb7u2

[+] Linux 3.8.0 with SMEP & SMAP (kernwrite)
[+] Linux 3.8.0 with KERNEXEC & UDEREF (kernwrite)
[-] Linux 3.8.0 (kernwrite)
[+] Linux 3.5.0 with SMEP (PERF_EVENTS)
[-] Linux 3.5.0 (PERF_EVENTS)
[+] Linux 2.6.35.8 with KERNEXEC & UDEREF (full-nelson, pktcdvd)
[-] Linux 2.6.35.8 (full-nelson, pktcdvd)
[+] Linux 2.6.33.6 with KERNEXEC & UDEREF (caps-to-root2, i-CAN-haz-M→
[-] Linux 2.6.33.6 (caps-to-root2, i-CAN-haz-MODHARDEN)

   Use the ↑ and ↓ keys to select which entry is highlighted.
   Press enter to boot the selected OS, 'e' to edit the commands
   before booting or 'c' for a command-line.
```

Figure D.2: `ret2dir_i386.vmwarevm` running over a VMware hypervisor. GRUB entries beginning with "`[-]`" correspond to vanilla Linux kernels without any ret2usr protection. Entries beginning with "`[+]`" indicate Linux kernels hardened against ret2usr attacks—the protection mechanisms enabled are listed after "with". Names in parentheses indicate the exploit(s) that the respective kernel is vulnerable to.

We have developed and pre-installed a ret2dir exploitation kit (`ekit`) on both VMs, which automates the process of running the proper exploit(s) on the kernel used to boot the guest OS. Once logged in, `ekit` can be invoked by running **`ekit/runme`**.

`ekit` automatically detects and reports the Linux kernel version used, along with *all* ret2usr protection mechanisms available (if any) and a summary of certain system characteristics (*e.g.,* the CPU model, RAM size, the size of `ZONE_HIGHMEM` in 32-bit kernels). Based on that it will also list the exploits available for the specific kernel version, including the EDB-ID and CVE number(s) of each exploit. Upon selecting a particular exploit to run, `ekit` will provide the option to invoke either the ret2usr or ret2dir variant, suggesting to execute the one that bypasses deployed protections.

The purpose of `ekit` is to enable users experiment with multiple combinations. For instance, a user can invoke a ret2usr exploit on a hardened kernel, to first verify that the respective ret2dir protection mechanisms work, and then execute the ret2dir variant of the same exploit on the same kernel, to observe how the techniques presented in Section 4.3 can bypass deployed ret2dir defenses. The source code of all ret2usr and ret2dir exploits used by `ekit` is available at '**`ekit/ret2usr/`**' and '**`ekit/ret2dir/`**', respectively.[1] Finally, note that both VMs come pre-configured with a GDB [91] kernel stub, allowing the remote debugging of the guest OS and vulnerable kernels. `ret2dir_i386.vmwarevm` can be debugged from the *host* OS by connecting to port `8832` (**`(gdb) target remote localhost:8832`**), whereas `ret2dir_amd64.vmwarevm` can be debugged by connecting to port `8864`.

D.3 getpmap

`getpmap` can be used with any Linux kernel that exports `/proc/<pid>/pagemap` as follows: **`getpmap -p <pid> -a <vaddr>`**. The decimal number returned by `getpmap` is the PFN that corresponds to virtual address `<vaddr>` of process `<pid>`.

[1] For brevity, all our ret2dir exploits use the deterministic technique described in Section 4.3.1.

Appendix E

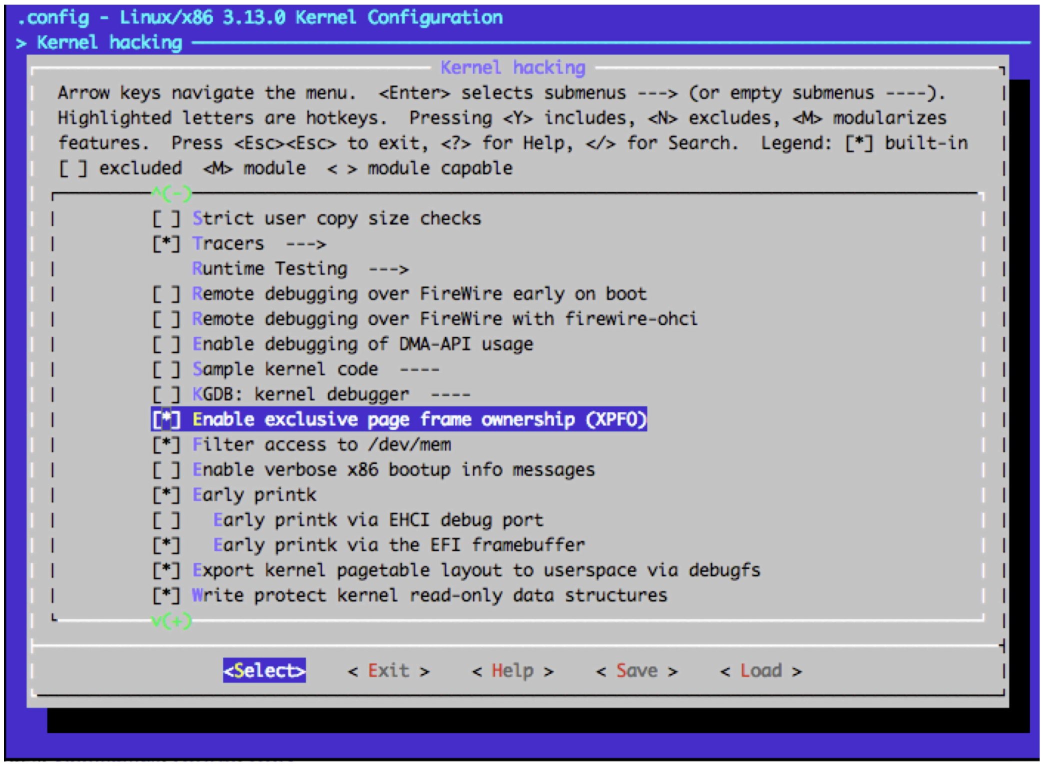
XPFO Distribution

Our XPFO prototype implements the page frame ownership scheme described in Section 5.3, along with the optimizations outlined in Section 5.3.1, and is freely available as a patch for Linux kernel v3.13 [164]. The initial version of XPFO supported both the x86 and x86-64 architectures. However, our latest release has been mainly tested and evaluated on x86-64 Debian GNU/Linux v7 [60].

E.1 Patching the Linux Kernel

What follows are the steps to patch the Linux kernel and add support for XPFO.

- Dependencies:
  - GNU patch v2.6.1 (or later) [89].
- Patch steps (we assume that the working directory is set to the topmost directory of Linux kernel v3.13 source code [164]):
  1. Download the XPFO patch [111]:
     ```
     wget https://www.cs.columbia.edu/~vpk/research/xpfo/linux-3.13-xpfo.patch
     ```
  2. Apply the patch:
     ```
     patch -p1 < linux-3.13-xpfo.patch
     ```



```
.config - Linux/x86 3.13.0 Kernel Configuration
> Kernel hacking
                              Kernel hacking
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in
  [ ] excluded  <M> module  < > module capable
                ^(-)
                [ ] Strict user copy size checks
                [*] Tracers  --->
                    Runtime Testing  --->
                [ ] Remote debugging over FireWire early on boot
                [ ] Remote debugging over FireWire with firewire-ohci
                [ ] Enable debugging of DMA-API usage
                [ ] Sample kernel code  ----
                [ ] KGDB: kernel debugger  ----
                [*] Enable exclusive page frame ownership (XPFO)
                [*] Filter access to /dev/mem
                [ ] Enable verbose x86 bootup info messages
                [*] Early printk
                [ ]   Early printk via EHCI debug port
                [*]   Early printk via the EFI framebuffer
                [*] Export kernel pagetable layout to userspace via debugfs
                [*] Write protect kernel read-only data structures
                v(+)

              <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure E.1: XPFO can be enabled by running **make menuconfig** (once the kernel is patched) and asserting the XPFO kernel configuration option under “Kernel Hacking”.

E.2 Enabling XPFO

After *successfully* patching the kernel, XPFO can be enabled by asserting the XPFO kernel configuration option. This can be performed by appending ‘**CONFIG_XPFO=y**’ to .config, or by invoking **make menuconfig**[1] and enabling XPFO under “Kernel Hacking”, as shown in Figure E.1. Note that XPFO requires DEBUG_KERNEL and X86 configuration options being enabled, and HIBERNATION, DEBUG_PAGEALLOC, and KMEMCHECK, being disabled. Lastly, the XPFO-enabled kernel can be compiled using the standard build procedure [156]. During boot time the following message is appended to system message buffer, indicating that XPFO is active: **XPFO (eXclusive Page Frame Ownership) protection: active**.

[1]Alternatively, make {n, x, g}config can be used.