usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

ret2dir: Rethinking Kernel Isolation

Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis,
Columbia University

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kemerlis

This paper is included in the Proceedings of the
23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7

ret2dir: Rethinking Kernel Isolation

Vasileios P. Kemerlis Michalis Polychronakis Angelos D. Keromytis
Columbia University
{vpk, mikepo, angelos}@cs.columbia.edu

Abstract

Return-to-user (ret2usr) attacks redirect corrupted kernel pointers to data residing in user space. In response, several kernel-hardening approaches have been proposed to enforce a more strict address space separation, by preventing arbitrary control flow transfers and dereferences from kernel to user space. Intel and ARM also recently introduced hardware support for this purpose in the form of the SMEP, SMAP, and PXN processor features. Unfortunately, although mechanisms like the above prevent the explicit sharing of the virtual address space among user processes and the kernel, conditions of implicit sharing still exist due to fundamental design choices that trade stronger isolation for performance.

In this work, we demonstrate how implicit page frame sharing can be leveraged for the complete circumvention of software and hardware kernel isolation protections. We introduce a new kernel exploitation technique, called *return-to-direct-mapped memory (ret2dir)*, which bypasses *all* existing ret2usr defenses, namely SMEP, SMAP, PXN, KERNEXEC, UDEREF, and kGuard. We also discuss techniques for constructing reliable ret2dir exploits against x86, x86-64, AArch32, and AArch64 Linux targets. Finally, to defend against ret2dir attacks, we present the design and implementation of an exclusive page frame ownership scheme for the Linux kernel that prevents the implicit sharing of physical memory pages with minimal runtime overhead.

1 Introduction

Although the operating system (OS) kernel has always been an appealing target, until recently attackers focused mostly on the exploitation of vulnerabilities in server and client applications—which often run with administrative privileges—as they are (for the most part) less complex to analyze and easier to compromise. 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, 140 more than in 2012 [73]. Admittedly, the exploitation of user-level software has become much harder, as recent versions of popular OSes come with numerous protections and exploit mitigations. The principle of least privilege is better enforced in user accounts and system services, compilers offer more protections against common software flaws, and highly targeted applications, such as browsers and document viewers, have started to employ sandboxing. 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 [63]. 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 [32].

Naturally, instead of putting significant effort to exploit applications fortified with numerous protections and sandboxes, attackers often turn their attention to the kernel. By compromising the kernel, they can elevate privileges, bypass access control and policy enforcement, and escape isolation and confinement mechanisms. 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 [5, 74].

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 [14, 23, 26], NULL pointer and pointer arithmetic errors [10, 12], memory disclosure vulnerabilities [13, 19], use-after-free and format string bugs [25, 27], signedness errors [17, 24], integer overflows [10,16], race conditions [11,15], as well as missing authorization checks and poor argument sanitization vulnerabilities [18, 20–22]. The exploitation of these bugs is particularly effective, despite the existence of kernel protection mechanisms, 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 into the address space of each process for performance reasons. This design allows an attacker with non-root access 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)* [57], affect all major OSes, including Windows and Linux, and are applicable in x86/x86-64, ARM, and other popular architectures.

In response to ret2usr attacks, several protections have been developed to enforce strict address space separation, such as PaX's KERNEXEC and UDEREF [77] and kGuard [57]. Having realized the importance of the problem, Intel introduced Supervisor Mode Execute Protection (SMEP) [46] and Supervisor Mode Access Prevention (SMAP) [54], two processor features that, when enabled, prevent the execution (or access) of *arbitrary* user code (or data) by the kernel. ARM has also introduced Privileged Execute-Never (PXN) [4], a feature equivalent to SMEP. These features offer similar guarantees to software protections with negligible runtime overhead.

Although the above mechanisms prevent the explicit sharing of the virtual address space among user processes and the kernel, conditions of *implicit* data sharing still exist. Fundamental OS components, such as physical memory mappings, I/O buffers, and the page cache, can still allow user processes to influence what data is accessible by the kernel. In this paper, we study the above problem in Linux, and expose design decisions that trade stronger isolation for performance. Specifically, we present a new kernel exploitation technique, called *return-to-direct-mapped memory (ret2dir)*, which relies on inherent properties of the memory management subsystem to bypass existing ret2usr protections. This is achieved by leveraging a kernel region that directly maps part or all of a system's physical memory, enabling attackers to essentially "mirror" user-space data within the kernel address space.

The task of mounting a ret2dir attack is complicated due to the different kernel layouts and memory management characteristics of different architectures, the partial mapping of physical memory in 32-bit systems, and the unknown location of the "mirrored" user-space data within the kernel. We present in detail different techniques for overcoming each of these challenges and constructing reliable ret2dir exploits against hardened x86, x86-64, AArch32, and AArch64 Linux targets.

To mitigate the effects of ret2dir attacks, we present the design and implementation of an exclusive page frame ownership scheme for the Linux kernel, which prevents the implicit sharing of physical memory among user processes and the kernel. The results of our evaluation show that the proposed defense offers effective protection with minimal (<3%) runtime overhead.

The main contributions of this paper are the following:

1. We expose a fundamental design weakness in the memory management subsystem of Linux by introducing the concept of *ret2dir* attacks. Our exploitation technique bypasses all existing ret2usr protections (SMEP, SMAP, PXN, KERNEXEC, UDEREF, kGuard) by taking advantage of the kernel's direct-mapped physical memory region.

2. 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 RAM area and accurately pinpointing their location.

3. 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 successfully the deployed ret2usr protections.

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

2 Background and Related Work

2.1 Virtual Memory Organization in Linux

Designs for safely combining different protection domains range from putting the kernel and user processes into a single address space and establishing boundaries using software isolation [52], to confining user process and kernel components into separate, hardware-enforced address spaces [2, 50, 66]. Linux and Linux-based OSes (Android [47], Firefox OS [72], Chrome OS [48]) adopt a more coarse-grained variant of the latter approach, by dividing the virtual address space into *kernel* and *user* space. In the x86 and 32-bit ARM (AArch32) architectures, the Linux kernel is typically mapped to the upper 1GB of the virtual address space, a split also known as "3G/1G" [28].[1] In x86-64 and 64-bit ARM (AArch64) the kernel is located in the upper *canonical half* [60, 69].

This design minimizes the overhead of crossing protection domains, and facilitates fast user-kernel interactions. When servicing a system call or handling an ex-

[1] Linux also supports 2G/2G and 1G/3G splits. A patch for a 4G/4G split in x86 [53] exists, but was never included in the mainline kernel for performance reasons, as it requires a TLB flush per system call.

ception, the kernel is running within the *context* of a preempted process. Hence, flushing the TLB is not necessary [53], while the kernel can access user space *directly* to read user data or write the result of a system call.

2.2 Return-to-user (ret2usr) Exploits

Although kernel code and user software have both been plagued by common types of vulnerabilities [9], the execution model imposed by the shared virtual memory layout between the kernel and user processes makes kernel exploitation noticeably different. The shared address space provides a unique vantage point to local attackers, as it allows them to control—both in terms of permissions and contents—part of the address space that is accessible by the kernel [91]. Simply put, attackers can easily execute shellcode with kernel rights by hijacking a privileged execution path and redirecting it to user space.

Attacks of this kind, known as return-to-user (ret2usr), have been the de facto kernel exploitation technique (also in non-Linux OSes [88]) for more than a decade [36]. In a ret2usr attack, kernel data is overwritten with user space addresses, typically after the exploitation of memory corruption bugs in kernel code [81], as illustrated in Figure 1. Attackers primarily aim for control data, such as return addresses [86], dispatch tables [36, 44], and function pointers [40, 42, 43, 45], as they directly facilitate arbitrary code execution [89]. Pointers to critical data structures stored in the kernel's heap [38] or the global data section [44] are also common targets, as they allow attackers to tamper with critical data contained in these structures by mapping fake copies in user space [38, 39, 41]. Note that the targeted data structures typically contain function pointers or data that affect the control flow of the kernel, so as to diverge execution to arbitrary points. The end effect of all ret2usr attacks is that *the control or data flow of the kernel is hijacked and redirected to user space code or data* [57].

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) for performing privilege escalation or executing a rootshell. 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 also been identified and exploited in interrupt service routines [71]. In such cases, where the kernel is either running in *interrupt context* or in a process context beyond the attacker's control [37, 85], the respective shellcode has to be injected in kernel space or be constructed using code gadgets from the kernel's text in a ROP/JOP fashion [8, 51, 87]. The latter approach is gaining popularity in real-world exploits, due to the increased adoption of kernel hardening techniques [31, 65, 68, 92, 93, 95].

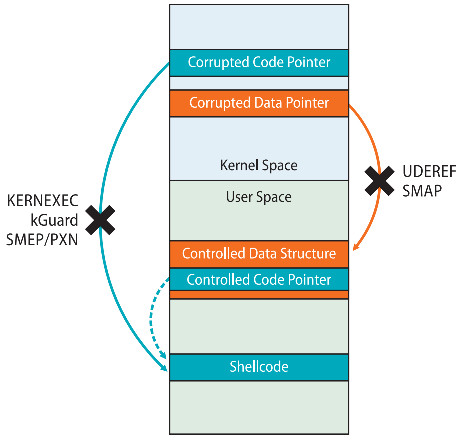

Figure 1: Operation of ret2usr attacks. A kernel code or data pointer is hijacked and redirected to controlled code or data in user space (tampered-with data structures may further contain pointers to code). Various protection mechanisms (KERNEXEC, UDEREF, kGuard, SMEP, SMAP, PXN) prevent arbitrary control flow transfers and dereferences from kernel to user space.

2.3 Protections Against ret2usr Attacks

Return-to-user attacks are yet another incarnation of the confused deputy problem [49]. Given the multi-architecture [42, 83] and multi-OS [88] nature of the problem, several defensive mechanisms exist for it. In the remainder of this section, we discuss the ret2usr defenses available in Linux with the help of Figure 1.

PaX: KERNEXEC and UDEREF are two features of the PaX [77] hardening patch set that prevent control flow transfers and dereferences from kernel to user space. In x86, KERNEXEC and UDEREF rely on memory segmentation [78] to map the kernel space into a 1GB segment that returns a memory fault whenever privileged code tries to dereference pointers to, or fetch instructions from, non-kernel addresses. In x86-64, due to the lack of segmentation support, UDEREF/amd64 [79] remaps user space memory into a different (shadow), non-executable area when execution enters the kernel (and restores it on exit), to prevent user-space dereferences. As the overhead of remapping memory is significant, an alternative for x86-64 systems is to enable KERNEXEC/amd64 [80], which has much lower overhead, but offers protection against only control-flow hijacking attacks. Recently, KERNEXEC and UDEREF were ported to the ARM architecture [90], but the patches added support for AArch32 only and rely on the deprecated MMU domains feature (discussed below).

SMEP/SMAP/PXN: Supervisor Mode Execute Protection (SMEP) [46] and Supervisor Mode Access Prevention (SMAP) [54] are two recent features of Intel processors that facilitate stronger address space separation (latest kernels support both features [31,95]). SMEP provides analogous protection to KERNEXEC, whereas SMAP operates similarly to UDEREF. Recently, ARM added support for an SMEP-equivalent feature, dubbed Privileged Execute-Never (PXN) [4], but Linux uses it only on AArch64. More importantly, on AArch32, PXN requires the MMU to operate on LPAE mode (the equivalent of Intel's Physical Address Extension (PAE) mode [55]), which disables MMU domains. Therefore, the use of KERNEXEC/UDEREF on AArch32 implies giving up support for PXN and large memory (> 4GB).

kGuard: kGuard [57] is a cross-platform compiler extension that protects the kernel from ret2usr attacks without relying on special hardware features. It enforces lightweight address space segregation by augmenting (at compile time) potentially exploitable control transfers with dynamic control-flow assertions (CFAs) that (at runtime) prevent the unconstrained transition of privileged execution paths to user space. The injected CFAs perform a small runtime check before every computed branch to verify that the target address is always located in kernel space or loaded from kernel-mapped memory. In addition, kGuard incorporates code diversification techniques for thwarting attacks against itself.

3 Attack Overview

Linux follows a design that trades weaker kernel-to-user segregation in favor of faster interactions between user processes and the kernel. The ret2usr protections discussed in the previous section aim to alleviate this design weakness, and fortify the isolation between kernel and user space with minimal overhead. In this work, 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 Linux memory management subsystem (`mm`), which can be abused to weaken 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.

3.1 Threat Model

We assume a Linux kernel hardened against ret2usr attacks using one (or a combination) of the protection mechanisms discussed in Section 2.3. Moreover, we assume an *unprivileged* attacker with local access, who seeks to elevate privileges by exploiting a kernel-memory corruption vulnerability [10–27] (see Section 2.2). Note that we do not make any assumptions about the type of corrupted data—code and data pointers are both possible targets [36, 40, 42–45, 86]. Overall, the adversarial capabilities we presume are identical to those needed for carrying out a ret2usr attack.

3.2 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 1. 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). 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, Solaris uses the `seg_kpm` mapping facility to provide a direct mapping of the whole RAM in 64-bit architectures [70].

As physical memory is allotted to user processes and the kernel, the existence of `physmap` results in *address aliasing*. Virtual address aliases, or *synonyms* [62], 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 synonym.

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 [36, 40, 42–45, 86] or a data pointer [38, 39, 41], the payload will consist of either shellcode, or controlled (tampered-with) data structures, as shown in Figure 2. Whenever the `mm` subsystem allocates (dynamic) memory to user space, it actually 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 [7], 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

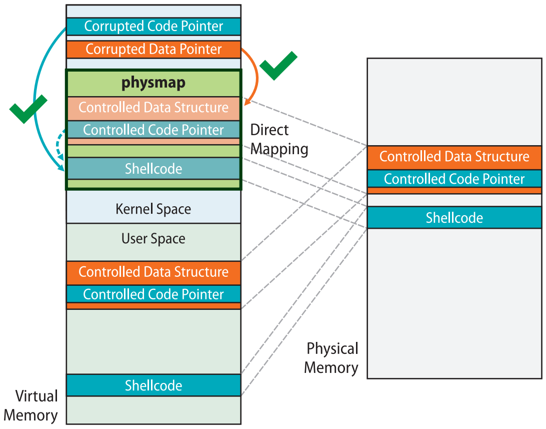

Figure 2: Overall ret2dir operation. The physmap (direct-mapped RAM) area enables a hijacked kernel code or data pointer to access user-controlled data, without crossing the user-kernel space boundary.

frame to the attacking process. Page frames are managed by mm using a *buddy allocator* [61]. 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 2. Although the kernel never uses such synonyms directly, mm keeps the whole RAM premapped 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 4.1 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 Demystifying physmap

A critical first step in understanding the mechanics of ret2dir attacks is to take a look at how the address space of the Linux kernel is organized—we use the x86 platform as a 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. Kernel space occupies the upper half (0xFFFF800000000000 – 0xFFFFFFFFFFFFFFFF), and is further divided into six regions [60]: 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).[2] For the purposes of ret2dir, in the following, we focus only on the direct-mapped region.

4.1 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 kernel's page tables every time memory is (de)allocated to map or unmap the respective page frames to or from the vmalloc arena. This not only incurs additional overhead, but results in increased TLB thrashing [67]. For these reasons, the majority of kernel components use kmalloc. 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

[2] 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.

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 1: physmap characteristics across different architectures (x86, x86-64, AArch32, AArch64).

over a region[3] that pre-maps the entire RAM (or part of it) for the following reasons [7]. First, kmalloc (de)allocates memory without touching the kernel's page table. This not only reduces TLB pressure significantly, but also removes high-latency operations, like page table manipulation and TLB shootdowns [70], 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 [64].

4.2 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) [35] is employed.

Table 1 lists physmap's properties of interest for the platforms we consider. In x86-64 systems, the 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 [69]. Conversely, in x86 systems, the kernel directly maps only a portion of the available 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 [59], 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 up to sizeof(physmap)/sizeof(PAGE) pages are mapped directly, starting from the first page frame.

4.3 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:[4] a utility in the form of a kernel module that exports page tables through the debugfs pseudo-filesystem [29]. 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 more conservative RW mapping. Finally, AArch32 and AArch64 map physmap with RWX permissions in every kernel version we tested (up to v3.12).

[3] 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 [6], and as of kernel v3.12, it supports three variants: SLAB, SLUB (default), and SLOB.

[4] kptdump resembles the functionality of Arjan van de Ven's patch [94]; unfortunately, 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.

5 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 that 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 information leakage from the kernel.

5.1 Leaking PFNs (via `/proc`)

The `procfs` pseudo-filesystem [58] has a long history of leaking security-sensitive information [56, 76]. 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 [34] (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 [1]. 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/4096) * sizeof(uint64_t)` (assuming 4KB pages).

Armed with the PFN of a given `uaddr`, denoted as `PFN(uaddr)`, its synonym `SYN(uaddr)` in `physmap` can be located using the following formula: `SYN(uaddr) = PHYS_OFFSET + 4096 * (PFN(uaddr) - PFN_MIN)`. `PHYS_OFFSET` corresponds to the known, fixed starting kernel virtual address of `physmap` (values for different configurations are shown in Table 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 can be "pinned" to RAM using `mlock`.

`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 1 for other setups), 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., as we have discussed, 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 perspective of an attacker, 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 the formula presented above. 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; additional tasks create memory pressure, "pushing" the attacker's allocations to the desired zones.

Contiguous synonyms: Certain exploits may 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.

5.2 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 identify 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 [33]. The base address and length of the physmap area is known in advance (Table 1). The latter corresponds to PFN_MAX - PFN_MIN page frames, shared among all user processes and the kernel. If the attacking process manages to copy the exploit payload into N memory-resident pages (in the physical memory range mapped by physmap), then the probability (P) that an arbitrarily chosen, page-aligned physmap address will point to the exploit payload is: P = N / (PFN_MAX - PFN_MIN). Our goal is to maximize P.

Spraying: Maximizing 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 5.1. 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 whether 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, simulating 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.

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. The results of our experimental security evaluation (Section 7) show that, depending on the configuration, the probability of success can be as high as 96%.

6 Putting It All Together

6.1 Bypassing SMAP and UDEREF

We begin with an example of a ret2dir attack against an x86 system hardened with SMAP or 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 [38,39,41]. 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 in Section 5.1). 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`.

6.2 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 [40, 42, 43, 45]. 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 [89]. In most cases, this boils down to executing something similar to `commit_creds(prepare_kernel_cred(0))`. 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 1), and maps the whole RAM using regular pages (4KB). Hence, 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 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 privileges. 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.

Non-executable physmap: In the above example, we took 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 7, even in this case, ret2dir attacks are possible through the use of return-oriented programming (ROP) [8,51,87].

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: an implementation of the same functionality consisting solely of a chain of code fragments ending with ret instructions, known as gadgets, which are located in the kernel's (executable) text segment. To trigger the ROP chain, kfptr is overwritten with an address that points to a *stack pivoting* gadget, 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 [3,81].

7 Security Evaluation

7.1 Effectiveness

We evaluated the effectiveness of ret2dir against kernels hardened with ret2usr protections, using real-world and custom exploits. We obtained a set of eight ret2usr exploits from the Exploit Database (EDB) [75], covering a wide range of kernel versions (v2.6.33.6–v3.8). We ran each exploit on an unhardened kernel to verify that it works, and that it indeed 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 the technique(s) presented in Section 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.

Table 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, or *(iii)* relevant against certain types of exploits. For instance, PXN is available only in ARM architectures, while SMEP and SMAP are Intel processor features. Furthermore, support for SMEP was added in kernel v3.2 and for SMAP in v3.7. Note that depending on the permissions of the physmap area (see Table 1), we had to modify some of the exploits that relied on plain shellcode to use a ROP payload, in order to achieve arbitrary code execution (although in ret2usr exploits attackers can give executable permission 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 section of the respective kernel.

As we mentioned in Section 2.3, KERNEXEC and UDEREF were recently ported to the AArch32 architecture [90]. 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. 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

[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 2: Tested exploits (converted to use the ret2dir technique) and configurations.

```
          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 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).

```
                 /* 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: Example of an x86 ROP payload (kernel v3.8) used in our ret2dir exploits for elevating privilege.

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 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 successfully 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 3 shows the shellcode we used in x86-64 and AArch32 architectures. The shellcode is *position independent*, 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 6.2. 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 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 shows an example of an x86 ROP payload used in our exploits. The first gadgets preserve

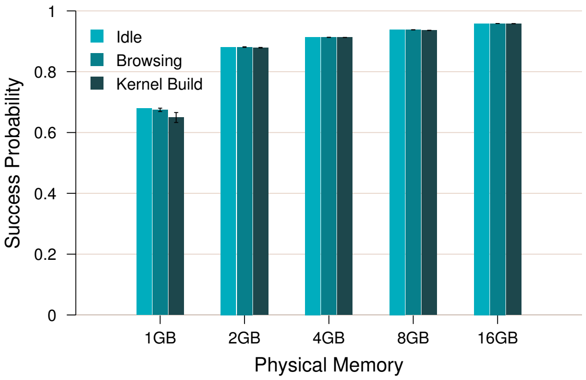

Figure 5: Probability that a selected physmap address will point to the exploit payload (successful exploitation) with a single attempt, when using physmap spraying, as a function of the available RAM.

the esp and ebp registers to facilitate reliable continuation (as discussed in Section 6.2). 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 essentially the same code as the shellcode to elevate privilege.

7.2 Spraying 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 5.2, the probability of randomly selecting a physmap address that indeed points 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.

Figure 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 included a single host armed with two 2.66GHz quad-core Intel Xeon X5500 CPUs and 16GB of RAM, running 64-bit Debian Linux v7. Each bar denotes the average value over 5 repetitions and error bars correspond to 95% confidence intervals. On every repetition we count the percentage of the *maximum* number of physmap-resident page frames that we managed to aquire, using the spraying technique (Section 5.2), over the size of physmap. We used three different workloads of increasing memory pressure: an idle system, a desktop-like workload with constant browsing activity in multiple tabs (Facebook, Gmail, Twitter, YouTube, and the USENIX website), and a distributed kernel compilation 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 so that their working set remains *hot* (worst-case scenario), otherwise the attacking ret2dir processes would 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 all 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 5.1, 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.

8 Defending Against ret2dir Attacks

Restricting access to /proc/<pid>/pagemap, or disabling the feature completely (e.g., by compiling the kernel without support for PROC_PAGE_MONITOR), is a simple first step that can hinder, but not prevent, ret2dir attacks. In this section, we present an eXclusive Page Frame Ownerwhip (XPFO) scheme for the Linux kernel that provides effective protection with low overhead.

8.1 XPFO 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 [84]).

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 [7], 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 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.

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 5 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.

8.2 Implementation

We implemented XPFO in the Linux kernel v3.13. 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. To that end, 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 [30]. Notice 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 below).

Whenever page frames are released to the buddy system, XPFO checks if bit `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 below). 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 5.1). 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 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.

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 [53], 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 in turn 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 path that required a clean page—clear_page is invoked by every kernel path that requires a zero-filled page frame.

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 through system objects like pipes and message queues.

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 3: XPFO performance evaluation results using macro-benchmarks (upper part) and micro-benchmarks (lower part) from the Phoronix Test Suite.

8.3 Evaluation

To evaluate the effectiveness of the proposed protection scheme, we used the ret2dir versions of the real-world exploits presented in Section 7.1. We back-ported our XPFO patch to each of the six kernel versions used in our previous evaluation (see Table 2), and tested again our ret2dir exploits when XPFO was enabled. In all cases, XPFO prevented the exploitation attempt.

To assess the performance overhead of XPFO, we used kernel v3.13, and a collection of macro-benchmarks and micro-benchmarks from the Phoronix Test Suite (PTS) [82]. PTS puts together *standard* system tests, like apachebench, pgbench, kernel build, and IOzone, typically used by kernel developers to track performance regressions. Our testbed was the same with the one used in Section 7.2; Table 3 summarizes our findings. Overall, XPFO introduces a minimal (negligible in most cases) overhead, ranging between 0.18–2.91%.

9 Conclusion

We have presented ret2dir, a novel kernel exploitation technique that takes advantage of direct-mapped physical memory regions to bypass existing protections against ret2usr attacks. To improve kernel isolation, we designed and implemented XPFO, an exclusive page frame ownership scheme for the Linux kernel that prevents the implicit sharing of physical memory. The results of our experimental evaluation demonstrate that XPFO offers effective protection with negligible runtime overhead.

Availability

Our prototype implementation of XPFO and all modified ret2dir exploits are available at: http://www.cs.columbia.edu/~vpk/research/ret2dir/

Acknowledgments

This work was supported by DARPA and the US Air Force through Contracts DARPA-FA8750-10-2-0253 and AFRL-FA8650-10-C-7024, respectively, with additional support from Intel Corp. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, the Air Force, or Intel.

References

[1] pagemap, from the userspace perspective, December 2008. https://www.kernel.org/doc/Documentation/vm/pagemap.txt.

[2] ACCETTA, M. J., BARON, R. V., BOLOSKY, W. J., GOLUB, D. B., RASHID, R. F., TEVANIAN, A., AND YOUNG, M. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of USENIX Summer* (1986), pp. 93–113.

[3] ARGYROUDIS, P. Binding the Daemon: FreeBSD Kernel Stack and Heap Exploitation. In *Black Hat USA* (2010).

[4] ARM® ARCHITECTURE REFERENCE MANUAL. ARM® v7-A and ARM® v7-R edition. Tech. rep., Advanced RISC Machine (ARM), July 2012.

[5] BEN HAYAK. The Kernel is calling a zero(day) pointer - CVE-2013-5065 - Ring Ring, December 2013. http://blog.spiderlabs.com/2013/12/the-kernel-is-calling-a-zeroday-pointer-cve-2013-5065-ring-ring.html.

[6] BONWICK, J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proc. of USENIX Summer* (1994), pp. 87–98.

[7] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, 3^{rd} ed. 2005, ch. Memory Management, pp. 294–350.

[8] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-Oriented Programming without Returns. In *Proc. of CCS* (2010), pp. 559–572.

[9] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. of APsys* (2011), pp. 51–55.

[10] COMMON VULNERABILITIES AND EXPOSURES. CVE-2005-0736, March 2005.

[11] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-1527, May 2009.

[12] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-2698, August 2009.

[13] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-3002, August 2009.

[14] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-3234, September 2009.

[15] COMMON VULNERABILITIES AND EXPOSURES. CVE-2009-3547, October 2009.

[16] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-2959, August 2010.

[17] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-3437, September 2010.

[18] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-3904, October 2010.

[19] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-4073, October 2010.

[20] COMMON VULNERABILITIES AND EXPOSURES. CVE-2010-4347, November 2010.

[21] COMMON VULNERABILITIES AND EXPOSURES. CVE-2012-0946, February 2012.

[22] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-0268, December 2013.

[23] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-1828, February 2013.

[24] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-2094, February 2013.

[25] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-2852, April 2013.

[26] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-2892, August 2013.

[27] COMMON VULNERABILITIES AND EXPOSURES. CVE-2013-4343, June 2013.

[28] CORBET, J. Virtual Memory I: the problem, March 2004. http://lwn.net/Articles/75174/.

[29] CORBET, J. An updated guide to debugfs, May 2009. http://lwn.net/Articles/334546/.

[30] CORBET, J. How many page flags do we really have?, June 2009. http://lwn.net/Articles/335768/.

[31] CORBET, J. Supervisor mode access prevention, October 2012. http://lwn.net/Articles/517475/.

[32] CORBET, J., KROAH-HARTMAN, G., AND MCPHERSON, A. Linux Kernel Development. Tech. rep., Linux Foundation, September 2013.

[33] DING, Y., WEI, T., WANG, T., LIANG, Z., AND ZOU, W. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks. In *Proc. of ACSAC* (2010), pp. 327–336.

[34] DISTROWATCH. Put the fun back into computing. Use Linux, BSD., November 2013. http://distrowatch.com.

[35] EDGE, J. Kernel address space layout randomization, October 2013. http://lwn.net/Articles/569635/.

[36] EXPLOIT DATABASE. EBD-131, December 2003.

[37] EXPLOIT DATABASE. EBD-16835, September 2009.

[38] EXPLOIT DATABASE. EBD-14814, August 2010.

[39] EXPLOIT DATABASE. EBD-15150, September 2010.

[40] EXPLOIT DATABASE. EBD-15285, October 2010.

[41] EXPLOIT DATABASE. EBD-15916, January 2011.

[42] EXPLOIT DATABASE. EBD-17391, June 2011.

[43] EXPLOIT DATABASE. EBD-17787, September 2011.

[44] EXPLOIT DATABASE. EBD-20201, August 2012.

[45] EXPLOIT DATABASE. EBD-24555, February 2013.

[46] GEORGE, V., PIAZZA, T., AND JIANG, H. Technology Insight: Intel©Next Generation Microarchitecture Codename Ivy Bridge, September 2011. http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf.

[47] GOOGLE. Android, November 2013. http://www.android.com.

[48] GOOGLE. Chromium OS, November 2013. http://www.chromium.org/chromium-os.

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

[50] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. MINIX 3: A Highly Reliable, Self-

Repairing Operating System. *SIGOPS Oper. Syst. Rev. 40*, 3 (July 2006), 80–89.

[51] Hund, R., Holz, T., and Freiling, F. C. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proc. of USENIX Sec* (2009), pp. 384–398.

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

[53] Ingo Molnar. 4G/4G split on x86, 64 GB RAM (and more) support, July 2003. `http://lwn.net/Articles/39283/`.

[54] Intel® 64 and IA-32 Architectures Software Developer's Manual. Instruction Set Extensions Programming Reference. Tech. rep., Intel Corporation, January 2013.

[55] Intel® 64 and IA-32 Architectures Software Developer's Manual. System Programming Guide, Part 1. Tech. rep., Intel Corporation, September 2013.

[56] Jana, S., and Shmatikov, V. Memento: Learning Secrets from Process Footprints. In *Proc. of IEEE S&P* (2012), pp. 143–157.

[57] Kemerlis, V. P., Portokalidis, G., and Keromytis, A. D. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proc. of USENIX Sec* (2012), pp. 459–474.

[58] Killian, T. J. Processes as Files. In *Proc. of USENIX Summer* (1984), pp. 203–207.

[59] King, R. Kernel Memory Layout on ARM Linux, November 2005. `https://www.kernel.org/doc/Documentation/arm/memory.txt`.

[60] Kleen, A. Memory Layout on amd64 Linux, July 2004. `https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt`.

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

[62] Koldinger, E. J., Chase, J. S., and Eggers, S. J. Architecture Support for Single Address Space Operating Systems. In *Proc. of ASPLOS* (1992), pp. 175–186.

[63] Kurmus, A., Tartler, R., Dorneanu, D., Heinloth, B., Rothberg, V., Ruprecht, A., Schröder-Preikschat, W., Lohmann, D., and Kapitza, R. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proc. of NDSS* (2013).

[64] Lameter, C. Generic Virtual Memmap support for `SPARSEMEM` v3, April 2007. `http://lwn.net/Articles/229670/`.

[65] Liakh, S. NX protection for kernel data, July 2009. `http://lwn.net/Articles/342266/`.

[66] Liedtke, J. On μ-Kernel Construction. In *Proc. of SOSP* (1984), pp. 237–250.

[67] Love, R. *Linux Kernel Development*, 2^{nd} ed. 2005, ch. Memory Management, pp. 181–208.

[68] Marinas, C. arm64: Distinguish between user and kernel XN bits, November 2012. `https://forums.grsecurity.net/viewtopic.php?f=7&t=3292`.

[69] Marinas, C. Memory Layout on AArch64 Linux, February 2012. `https://www.kernel.org/doc/Documentation/arm64/memory.txt`.

[70] McDougall, R., and Mauro, J. *Solaris Internals*, 2^{nd} ed. 2006, ch. File System Framework, pp. 710–710.

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

[72] Mozilla. Firefox OS, November 2013. `https://www.mozilla.org/en-US/firefox/os/`.

[73] National Vulnerability Database. Kernel Vulnerabilities, November 2013. `http://goo.gl/GJpw0b`.

[74] Nils and Jon. Polishing Chrome for Fun and Profit, August 2013. `http://goo.gl/b5hmjj`.

[75] Offensive Security. The Exploit Database, November 2013. `http://www.exploit-db.com`.

[76] Ormandy, T., and Tinnes, J. Linux ASLR Curiosities. In *CanSecWest* (2009).

[77] PaX. Homepage of The PaX Team, November 2013. `http://pax.grsecurity.net`.

[78] PaX Team. UDEREF/i386, April 2007. `http://grsecurity.net/~spender/uderef.txt`.

[79] PaX Team. UDEREF/amd64, April 2010. `http://grsecurity.net/pipermail/grsecurity/2010-April/001024.html`.

[80] PaX Team. Better kernels with GCC plugins, October 2011. `http://lwn.net/Articles/461811/`.

[81] Perla, E., and Oldani, M. *A Guide To Kernel Exploitation: Attacking the Core*. 2010, ch. Stairway to Successful Kernel Exploitation, pp. 47–99.

[82] PTS. Phoronix Test Suite, February 2014. `http://www.phoronix-test-suite.com`.

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

[84] Rizzo, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. of USENIX ATC* (2012), pp. 101–112.

[85] Rosenberg, D. Owned Over Amateur Radio: Remote Kernel Exploitation in 2011. In *Proc. of DEF CON®* (2011).

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

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

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

[89] Spengler, B. Enlightenment Linux Kernel Exploitation Framework, July 2013. `https://grsecurity.net/~spender/exploits/enlightenment.tgz`.

[90] Spengler, B. Recent ARM security improvements, February 2013. `https://forums.grsecurity.net/viewtopic.php?f=7&t=3292`.

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

[92] van de Ven, A. Debug option to write-protect rodata: the write protect logic and config option, November 2005. `http://lkml.indiana.edu/hypermail/linux/kernel/0511.0/2165.html`.

[93] van de Ven, A. Add `-fstack-protector` support to the kernel, July 2006. `http://lwn.net/Articles/193307/`.

[94] van de Ven, A. x86: add code to dump the (kernel) page tables for visual inspection, February 2008. `http://lwn.net/Articles/267837/`.

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