kGuard: Lightweight Kernel Protection against Return-to-user Attacks

Vasileios P. Kemerlis Georgios Portokalidis Angelos D. Keromytis
Network Security Lab
Department of Computer Science
Columbia University, New York, NY, USA
{vpk, porto, angelos}@cs.columbia.edu

Abstract

Return-to-user (ret2usr) attacks exploit the operating system kernel, enabling local users to hijack privileged execution paths and execute arbitrary code with elevated privileges. Current defenses have proven to be inadequate, as they have been repeatedly circumvented, incur considerable overhead, or rely on extended hypervisors and special hardware features. We present kGuard, a compiler plugin that augments the kernel with compact inline guards, which prevent ret2usr with low performance and space overhead. kGuard can be used with any operating system that features a weak separation between kernel and user space, requires no modifications to the OS, and is applicable to both 32- and 64-bit architectures. Our evaluation demonstrates that Linux kernels compiled with kGuard become impervious to a variety of control-flow hijacking exploits. kGuard exhibits lower overhead than previous work, imposing on average an overhead of 11.4% on system call and I/O latency on x86 OSs, and 10.3% on x86-64. The size of a kGuard-protected kernel grows between 3.5% and 5.6%, due to the inserted checks, while the impact on real-life applications is minimal ($\leq$1%).

1 Introduction

The operating system (OS) kernel is becoming an increasingly attractive target for attackers [30, 60, 61, 64]. Due to the weak separation between user and kernel space, direct transitions from more to less privileged protection domains (*e.g.,* kernel to user space) are permissible, even though the reverse is not. As a result, bugs like NULL pointer dereferences that would otherwise cause only system instability, become serious vulnerabilities that facilitate privilege escalation attacks [64]. When successful, these attacks enable local users to execute arbitrary code with kernel privileges, by redirecting the control flow of the kernel to a user process. Such *return-to-user* (ret2usr) attacks have affected all major OSs, including Windows [60], Linux [16, 18], and FreeBSD [19, 59, 61], while they are not limited to x86 systems [23], but have also targeted the ARM [30], DEC [31], and PowerPC [25] architectures.

There are numerous reasons to why attacks against the kernel are becoming more common. First and foremost, processes running with administrative privileges have become harder to exploit due to the various defense mechanisms adopted by modern OSs [34, 52]. Second, NULL pointer dereference errors had not received significant attention, until recently, exactly because they were thought impractical and too difficult to exploit. In fact, 2009 has been proclaimed, by some security researchers, as "*the year of the kernel NULL pointer dereference flaw*" [15]. Third, exploiting kernel bugs, besides earning attackers administrative privileges, enables them to mask their presence on compromised systems [6].

Previous approaches to the problem are either impractical for deployment in certain environments or can be easily circumvented. The most popular approach has been to disallow user processes to memory-map the lower part of their address space (*i.e.,* the one including page zero). Unfortunately, this scheme has been circumvented by various means [21, 66] and is not backwards compatible [35]. The PaX [52] patch for x86 and x86-64 Linux kernels does not exhibit the same shortcomings, but greatly increases system call and I/O latency, especially on 64-bit systems.

Recent advances in virtualization have fostered a wave of research on extending virtual machine monitors (VMMs) to enforce the integrity of the virtualized guest kernels. SecVisor [62] and NICKLE [56] are two hypervisor-based systems that can prevent ret2usr attacks by leveraging memory virtualization and VMM introspection. However, virtualization is not always practical. Consider smartphone devices that use stripped-down versions of Linux and Windows, which are also vulnerable to such attacks [30]. Running a complex VMM, like

SecVisor, on current smartphones is not realistic due to their limited resources (*i.e.,* CPU and battery life). On PCs, running the whole OS over a VM incurs performance penalties and management costs, while increasing the complexity and size of a VMM can introduce new bugs and vulnerabilities [44, 58, 71]. To address the latter, we have seen proposals for smaller and less error-prone hypervisors [65], as well as hypervisor protection solutions [4, 67]. The first exclude mechanisms such as SecVisor, while the second add further complexity and overhead, and lead to a "turtles all the way down" problem,[1] by introducing yet another software layer to protect the layers above it. Addressing the problem in hardware would be the most efficient solution, but even though Intel has recently announced a new CPU feature, named SMEP [37], to thwart such attacks, hardware extensions are oftentimes adopted slowly by OSs. Note that other vendors have not publicly announced similar extensions.

We present a lightweight solution to the problem. kGuard is a compiler plugin that augments kernel code with control-flow assertions (CFAs), which ensure that privileged execution remains within its valid boundaries and does not cross to user space. This is achieved by identifying all indirect control transfers during compilation, and injecting compact dynamic checks to attest that the kernel remains confined. When a violation is detected, the system is halted by default, while a custom fault handler can also be specified. kGuard is able to protect against attacks that overwrite a branch target to directly transfer control to user space [23], while it also handles more elaborate, two-step attacks that overwrite data pointers to point to user-controlled memory, and hence hijack execution via tampered data structures [20].

Finally, we introduce two novel code diversification techniques to protect against attacks that employ bypass trampolines to avoid detection by kGuard. A trampoline is essentially an indirect branch instruction contained within the kernel. If an attacker manages to obtain the address of such an instruction and can also control its operand, he can use it to bypass our checks. Our techniques randomize the locations of the CFA-indirect branch pairs, both during compilation and at runtime, significantly reducing the attackers' chances of guessing their location. The main contributions of this paper can be summarized in the following:

- We present the design and implementation of kGuard, a compiler plugin that protects the kernel from ret2usr attacks by injecting fine-grained inline guards during compilation. Our approach does not require modifications to the kernel or additional software, such as a VMM. It is also architecture independent by design, allowing us to compile OSs for different target architectures and requires little modifications for supporting new OSs.

- We introduce two code diversification techniques to randomize the location of indirect branches, and their associated checks, for thwarting elaborate exploits that employ bypass trampolines.

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

- 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 exploit.

- 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 architectures. Particularly, we show negligible impact on real-life applications, and an average overhead of 11.4% on system call and I/O latency on x86 Linux, and 10.3% on x86-64. The space overhead of kGuard due to the instrumentation is between 3.5% – 5.6%, while build time increases by 0.05% to 0.3%.

kGuard is to some extent related to previous research on control-flow integrity (CFI) [2]. Similar to CFI, we rely on inline checks injected before every unsafe control-flow transfer. Nevertheless, CFI depends on a precomputed control-flow graph for determining the permissible targets of every indirect branch, and uses binary rewriting to inject labels and checks in binaries.

CFI is not effective against ret2usr attacks. Its integrity is only guaranteed if the attacker cannot overwrite the code of the protected binary or execute data. During a ret2usr attack, the adversary completely controls user space memory, both in terms of contents and rights, and hence, can subvert CFI by prepending his shellcode with the respective label. Additionally, CFI induces considerable performance overhead, thereby making it difficult to adopt. Ongoing work tries to overcome the limitations of the technique [72]. kGuard can be viewed as a lightweight variant of CFI and Program Shepherding [43] that is more suitable and efficient in protecting kernel code from ret2usr threats.

The rest of this paper is organized as follows. In Section 2, we look at how ret2usr attacks work and why the current protection schemes are insufficient. Section 3 presents kGuard. We discuss the implementation of the

[1] `http://en.wikipedia.org/wiki/Turtles_all_the_way_down`

kGuard GCC plugin in Section 4, and evaluate its effectiveness and performance in Section 5. Section 6 discusses possible extensions. Related work is in Section 7 and conclusions in Section 8.

2 Overview of ret2usr Attacks

2.1 Why Do They Work?

Commodity OSs offer process isolation through private, hardware-enforced virtual address spaces. However, as they strive to squeeze more performance out of the hardware, they adopt a "shared" process/kernel model for minimizing the overhead of operations that cross protection domains, like system calls. Specifically, Unix-like OSs divide virtual memory into *user* and *kernel* space. The former hosts user processes, while the latter holds the kernel, device drivers, and kernel extensions (interested readers are referred to Figure 5, in the appendix, for more information regarding the virtual memory layout of kernel and user space in Linux).

In most CPU architectures, the segregation of the two spaces is assisted and enforced by two hardware features. The first is known as *protection rings* or CPU modes, and the second is the memory management unit (MMU). The x86/x86-64 CPU architecture supports four protection rings, with the kernel running in the most privileged one (ring 0) and user applications in the least privileged (ring 3).[2] Similarly, the PowerPC platform supports two CPU modes, SPARC and MIPS three, and ARM seven. All these architectures also feature a MMU, which implements virtual memory and ensures that memory assigned to a ring is not accessible by less privileged ones.

Since code running in user space cannot directly access or jump into the kernel, specific hardware facilities (*i.e.,* interrupts) or special instructions (*e.g.,* `SYS{ENTER,CALL}` and `SYS{EXIT,RET}` in x86/x86-64) are provided for crossing the user/kernel boundary. Nevertheless, while executing kernel code, complete and unrestricted access to *all* memory and system objects is available. For example, when servicing a system call for a process, the kernel has to directly access user memory for storing the results of the call. Hence, when kernel code is abused, it can jump into user space and execute arbitrary code with kernel privileges. Note that although some OSs have completely separated kernel and user spaces, such as the 32-bit XNU and Linux running on UltraSPARC, most popular platforms use a shared layout. In fact, on MIPS the shared address space is mandated by the hardware.

[2]Some x86/x86-64 CPUs have more than four rings. Hardware-assisted virtualization is colloquially known as ring -1, while System Management Mode (SMM) is supposedly at ring -2.

As a consequence, software bugs that are only a source of instability in user space, like NULL pointer dereferences, can have more dire effects when located in the kernel. Spengler [64] demonstrated such an attack by exploiting a NULL pointer dereference bug, triggered by the invocation a system call with specially crafted parameters. Earlier, it was generally thought that such flaws could only be used to perform denial-of-service (DoS) attacks [29], but Spengler's exploit showed that mapping code segments with different privileges inside the same scope can be exploited to execute arbitrary user code with kernel privileges. Note that SELinux [47], the hardened version of the Linux kernel, is also vulnerable to this attack.

2.2 How Do They Work?

ret2usr attacks are manifested by overwriting kernel-level control data (*e.g.,* return addresses, jump tables, function pointers) with user space addresses. In early versions of such exploits, this was accomplished by invoking a 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.[3] However, if the attacker has local access to the system, he can build a program with arbitrary data or code mapped at address zero (or any other address in his program for that matter). Notice that since the attacker controls the program, its memory pages can be mapped both writable and executable (*i.e.,* W^X anti-measures do not apply).

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

Snippet 1: NULL *function* pointer in Linux (*net/socket.c*)

Snippet 1 presents a straightforward NULL function pointer vulnerability [17] that affected all Linux kernel versions released between May 2001 and August 2009 (2.4.4/2.6.0 – 2.4.37/2.6.30.4). In this exploit, 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 742 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 kernel privileges. A more detailed analysis of this attack is presented in Appendix A.

[3]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. Other OSs fail-stop with a kernel panic.

```
1333 /*[!] NULL pointer dereference (ops) [!]*/
1334 ibuf->ops->get(ipipe, ibuf);
1335 obuf  = opipe->bufs + nbuf;
1336 *obuf = *ibuf;
```

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

Snippet 2 shows the Linux kernel bug exploited by Spengler [64], which is more elaborate. 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 read via `get`, which is mapped to the seventh double word (assuming an x86 32-bit architecture) after the address pointed by `ops` (*i.e.,* due to the definition of the structure). Hence, the kernel reads the branch target from 0x0000001C, *which is controlled by the user*. This enables an attacker to redirect the kernel to an arbitrary address.

NULL pointer dereferences are not the only attack vector for ret2usr exploits. Attackers can partially corrupt, or completely overwrite with user space addresses, kernel-level control data, after exploiting memory safety bugs. Examples of common targets include, return addresses, global dispatch tables, and function pointers stored in kernel stack and heap. In addition, other vulnerabilities allow attackers to corrupt arbitrary kernel memory, and consequently any function or data pointer, due to the improper sanitization of user arguments [22, 23]. Use-after-free vulnerabilities due to race conditions in FreeBSD 6.x/7.x and Linux kernels before 2.6.32-rc6 have also been used for the same purpose [19, 20]. These flaws are more complex and require multiple simultaneous kernel entrances to trigger the bug. Once they succeed, the attacker can corrupt a pointer to a critical kernel data structure that grants him complete control over its contents by mapping a tampered data structure at user space memory. If the structure contains a function pointer, the attacker can achieve user code execution.

The end effect of all these attacks is that the kernel is hijacked and control is redirected to user space code. Throughout the rest of this paper, we will refer to this type of exploitation as *return-to-user* (ret2usr), since it resembles the older return-to-libc [27] technique that redirected control to existing code in the C library. Interestingly, ret2usr attacks are yet another incarnation of the confused deputy problem [39], where a user "cheats" the kernel (deputy) to misuse its authority and execute arbitrary, non-kernel code with elevated privileges. Finally, while most of the attacks discussed here target Linux, similar flaws have been reported against FreeBSD, OpenBSD, and Windows [19, 26, 59–61].

2.3 Limitations of Current Defenses

Restricting mmap The mitigation strategy adopted by most Linux and BSD systems is to restrict the ability to map the first pages of the address space to users with administrative privileges only. In Linux and FreeBSD, this is achieved by modifying the `mmap` system call to apply the respective restrictions, as well as preventing binaries from requesting page zero mappings. OpenBSD completely removed the ability to map page zero, while NetBSD has not adopted any protective measures yet.

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 merely a protection scheme against exploits that rely on NULL pointer bugs. If an attacker bypasses the restriction imposed by `mmap`, he can still orchestrate a ret2usr attack. Second, it does not protect against exploits where control is redirected to memory pages above the forbidden `mmap` region (*e.g.,* by nullifying one or two bytes of a pointer, or overwriting a branch target with an arbitrary value). Third, it breaks compatibility with applications that rely on having access to low logical addresses, such as QEMU [5], Wine [70], and DOSEMU [28]. Similar problems have been reported for the FreeBSD distribution [35].

In fact, shortly after these protection mechanisms were set in place, many techniques were developed for circumventing them. The first technique for bypassing `mmap` restrictions used the `brk` system call for changing the location of the program break (marked as `brk offset` in Figure 5), which indicates where the heap segment starts. By setting the break to a low logical address, it was possible to dynamically allocate memory chunks inside page zero. Another technique used the `mmap` system call to map pages starting from an address above the forbidden region and extend the allocated region downwards, by supplying the `MAP_GROWSDOWN` parameter to the call. A more elaborate mechanism utilized the different execution domains supported by Linux, which can be set with the `personality` system call, for executing binaries compiled for different OSs. Specifically, an attacker could set the personality of a binary to SRV4, thus mapping page zero, since SRV4 utilizes the lower pages of the address space [66]. Finally, the combination of a NULL pointer with an integer overflow has also been demonstrated, enabling attackers to completely bypass the memory mapping restrictions [20, 21]. Despite the fact that all the previous techniques were fixed shortly after they were discovered, it is possible that other approaches can (and probably will be) developed by persistent attackers, since the root cause of this new manifestation of control hijacking attacks is the weak separation of spaces.

Hardening with PaX UDEREF [53] and KERNEXEC are two patches included in PaX [52] for hardening the Linux kernel. In particular, they provide protection against dereferencing, or branching to, user space memory. In x86, PaX relies on memory segmentation. It maps kernel space into an expand-down segment that returns a memory fault whenever privileged code tries to dereference pointers to other segments.[4] In x86-64, where segmentation is not available, PaX resorts in temporarily remapping user space memory into a different area, using non-executable rights, when execution enters the kernel, and restoring it when it exits.

PaX has limitations. First, it requires kernel patching and is platform and architecture specific (*i.e.,* x86/x86-64 Linux only). On the other hand, ret2usr attacks not only have been demonstrated on many architectures, such as ARM [30], DEC Alpha [31], and PowerPC [25], but also on different OSs, like the BSDs [19, 26, 59, 61]. Second, as we experimentally confirmed, PaX incurs non-negligible performance overhead (see Section 5). In x86, it achieves strong isolation using the segmentation unit, but the kernel still needs to interact with user-level processes. Hence, PaX modifies the stub that executes during kernel entry for setting the respective segments, and also patches code that copies data to/from user space, so as to temporarily flatten the privileged segment for the duration of the copy. Evidently, this approach increases system call latency. In x86-64, remapping user space requires page table manipulation, which results in a TLB flush and exacerbates the problem [41].

3 Protection with kGuard

3.1 Overview

We propose a defensive mechanism that builds upon *inline monitoring* and *code diversification*. kGuard is a cross-platform compiler plugin that enforces address space segregation, without relying on special hardware features [37, 53] or custom hypervisors [56, 62]. It protects the kernel from ret2usr attacks with low-overhead, by augmenting 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 indirect branches to verify that the target address is always in kernel space. 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 might hold locks or other resources). In Section 6, we discuss how we can implement custom handlers for facilitating forensic analysis, error virtualization [63], selective confinement, and protection against persistent attacks.

After compiling a kernel with kGuard, its execution is limited to the privileged address space segment (*e.g.,* addresses higher than 0xC0000000 in x86 Linux and BSD). kGuard does not rely on any mapping restriction, so the previously restricted addresses can be dispensed to the process, lifting the compatibility issues with various applications [5, 28, 35, 70]. Furthermore, the checks cannot be bypassed using `mmap` hacks, like the ones described in the previous section, nor can they be circumvented by elaborate exploits that manage to jump to user space by avoiding the forbidden low memory addresses. More importantly, the kernel can still read and write user memory, so its functionality remains unaffected.

3.2 Threat Model

In this work, we ascertain that an adversary is able to completely overwrite, partially corrupt (*e.g.,* zero out only certain bytes), or nullify control data that are stored inside the address space of the kernel. Notice that overwriting certain data with arbitrary values, differs significantly from overwriting *arbitrary kernel memory with arbitrary values*. 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), which in turn may contain control data.

Our technique is straightforward and guarantees that *kernel/user space boundary violations 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 inside kernel space, nor it thwarts code-reuse attacks that utilize return-oriented/jump-oriented programming (ROP/JOP) [7, 40]. Nevertheless, note the following. First and foremost, our approach is orthogonal to many solutions that do protect against such threats [4, 14, 42, 45, 53, 62]. For instance, canaries injected by the compiler [34] can be used against ret2usr attacks performed via kernel stack-smashing. Second, the unique nature of address space sharing casts many protection schemes, for the aforementioned problems, ineffective. As an example, consider again the case of ROP/JOP in the kernel setting. No matter what anti-ROP techniques have been utilized [45, 51], the attacker can still execute arbitrary code, as long as there is no strict process/kernel separation, by mapping his code to user space and transferring control to it (after hijacking a privileged execution path).

[4]In x86, UDEREF restricts only the `SS`, `DS`, and `ES` segments. `CS` is taken care by the accompanying KERNEXEC patch.

Finally, in order to protect kGuard from being subverted, we utilize a lightweight diversification technique for the kernel's text, which can also mitigate kernel-level attacks that use code "gadgets" in a ROP/JOP fashion (see Section 3.5). Overall, the aim of kGuard is not to provide strict control-flow integrity for the kernel, but rather to render a realistic threat ineffective.

3.3 Preventing ret2usr Attacks with CFAs

In the remainder of this section, we discuss the fundamental aspects of kGuard using examples based on x86-based Linux systems. However, kGuard is by no means restricted to 32-bit systems and Linux. It can be used to compile any kernel that suffers from ret2usr attacks for both 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` and `jmp` instructions with a register or memory operand, and the `ret` instruction, which takes an implicit memory operand from the stack (*i.e.,* the saved return address). kGuard injects CFAs in both cases to check that the branch target, specified by the respective operand, is inside kernel space.

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

Snippet 3: CFA_R guard applied on an indirect call in x86 Linux (*drivers/cpufreq/cpufreq.c*)

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

Snippet 4: CFA_R guard in C-like code (x86)

We use two different CFA guards, namely CFA_R and CFA_M, depending on whether the control transfer that we want to confine uses a register or memory operand. Snippet 3 shows an example of a CFA_R guard. 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 lower kernel address 0xC0000000.[5] 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; `panic()`) into the branch register and proceeds to execute `call`, which will result into invoking the violation handler. In C-like code, this is equivalent to injecting the statements shown in Snippet 4.

[5]The same is true for x86 FreeBSD/NetBSD, whereas for x86-64 the check should be with address 0xFFFFFFFF80000000. 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 43 d6 2d b8        ; 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)
```

Snippet 5: CFA_M guard applied on an indirect call in x86 Linux (*net/socket.c*)

```
register void *target_address_ptr;
...
target_address_ptr = &target_addr;
if (target_address_ptr < 0xC0000000)
    call <violation handler>;
if (target_address < 0xC0000000)
    target_address = &<violation handler>;
call *target_address;
```

Snippet 6: CFA_M guard in C-like code (x86)

Similarly, CFA_M guards confine indirect branches that use memory operands. Snippet 5 illustrates how kGuard instruments the faulty control transfer of `sock_sendpage()` (the original code is shown in Snippet 1). The indirect `call` (`call 0x50(%ebx)`; Figure 5) is prepended by a sequence of 10 instructions that perform two distinct assertions. CFA_M not only asserts that the branch target is within 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 function pointers (see Snippet 2 in Section 2.2). Snippet 6 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 (*i.e.,* `edi` in our example). The address of the memory location that stores the branch target (`ebx + 0x50` = 0xfa7c8538; Figure 5), is dynamically resolved via an 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 branch target similarly to the CFA_R case.

```
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)
```

Snippet 7: Optimized CFA_M guard

3.4 Optimizations

In certain cases, we can statically determine that the address of the memory location that holds the branch target is always mapped in kernel space. Examples include a branch operand read from the 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 segment of the kernel. In this case, the first assertion of a CFA_M guard will always be true, since the memory operand points within kernel space. We optimize such scenarios by removing the redundant assertion, effectively reducing the size of the inline guard to 3 instructions. For instance, Snippet 7 depicts how we can optimize the code shown in Snippet 5, assuming that `ebx` is loaded with the address of a global symbol from kernel's data segment. `ret` *instructions are always confined using the optimized CFA_M variant*.

3.5 Mechanism Protection

CFA_R and CFA_M guards, as presented thus far, provide reliable protection against ret2usr attacks, only if the attacker exploits a vulnerability that allows him to partially control a computed branch target. Currently, all the well known and published ret2usr exploits, which we analyzed in Section 2 and further discuss in Section 5.1, fall in this category. However, vulnerabilities where the attacker can overwrite kernel memory with arbitrary values also exist [22]. When such flaws are present, exploits could attempt to bypass kGuard. This section discusses how we protect against such attacks.

3.5.1 Bypass Trampolines

To subvert kGuard, an attacker has to be able to determine the address of a (indirect) control transfer instruction inside the text segment of the kernel. Moreover, he should also be able to reliably control the value of its operand (*i.e.,* its branch target). We shall refer to that branch as a *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 control branch in user space [40]. By overwriting the value of a protected branch target with the address of a bypass trampoline, the attacker can successfully execute a jump to user space. The first CFA corresponding to the initially exploited branch will succeed, since the address of the trampoline remains inside the privileged memory segment, while the second CFA that guards the bypass trampoline is completely bypassed by jumping directly to the branch instruction.

Similarly, jumping in the middle of an instruction that contains an indirect branch within, could also be used to subvert kGuard. At this point, we would like to stress that if an attacker is armed with a powerful exploit for a vulnerability that allows him to overwrite arbitrary kernel memory with arbitrary values, he can easily elevate his privileges by overwriting the credentials associated with a process under his control. In other words, the attacker can achieve his goal without violating the control-flow by jumping into user-level shellcode.

3.5.2 Code Diversification Against Bypasses

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

Code inflation This technique *reshapes* the kernel's text area. We begin with randomizing the starting address of the text segment. This is achieved by inserting a random `NOP` sled at its beginning, which effectively shifts all executable instructions by an arbitrary offset. Next, we continue by 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, making it harder for an 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. For example, in Linux and BSD the kernel space is at least 1GB. Hence, we can achieve more than 20 bits of entropy (*i.e.,* the `NOP` sled can be $\geq$ 1MB) without notably consuming address space.

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 text and symbols. If the attacker has access to the binary image of the confined kernel or is armed with a kernel-level memory leak [32], the probability of successfully guessing the address of a bypass trampoline increases. We posit that assigning safe file permissions to the kernel's text, modules, and debugging symbols is not a limiting factor.[6] 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 ring buffer (`dmesg`), in order to prevent the leakage of kernel addresses.[7]

[6]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.*

[7]In Linux, this can be done by asserting the `kptr_restrict` [24] sysctl option that hides exposed kernel pointers in `/proc` interfaces.

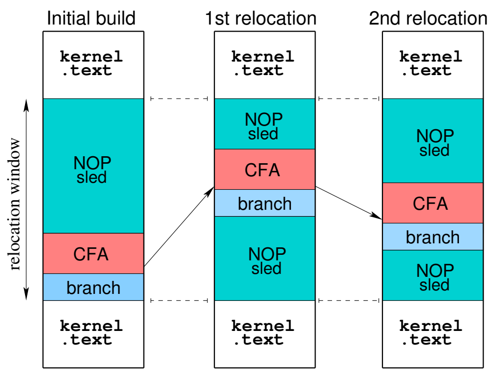


Figure 1: CFA motion synopsis. kGuard relocates each inline guard and protected branch, within a certain window, by routinely rewriting the text segment of the kernel.

CFA motion The basic idea behind this technique is the "continuous" relocation of the protected branches and injected guards, by rewriting the text segment of the kernel. Figure 1 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 kernel's text, the type and size of the guard, the length of the prepended `NOP` sled, as well as the size of the protected branch. Armed with that information, we can then migrate every CFA and indirect branch instruction separately, by moving it inside the following window: `sizeof(nop_sled) + sizeof(cfa) + sizeof(branch)`. Currently, we only support CFA motion during kernel bootstrap. In Linux, this is performed after the boot loader (*e.g.,* LILO, GNU GRUB) extracts the kernel image and right before jumping to the `setup()` routine [8]. In BSDs, we perform the relocation after the `boot` program has executed and right before transferring control to the machine-dependent initialization routines (*i.e.,* `mi_startup()` in FreeBSD and `main()` in {Net, Open}BSD) [49]. Finally, note that CFA motion can also be performed at runtime, on a live system, by trading runtime overhead for safety. In Section 6, we discuss how we can expand our current implementation, with moderate engineering effort, to support real-time CFA migration.

To further protect against evasion, kGuard can be combined with other techniques that secure kernel code against code-injection [46] and code-reuse attacks [45, 51]. That said, mind that ret2usr violations are detected at runtime, and hence one false guess is enough for identifying the attacker and restricting his capabilities (*e.g.,* by revoking his access to prevent brute-force attacks). In Section 6, we further discuss how kGuard can deal with persistent threats.

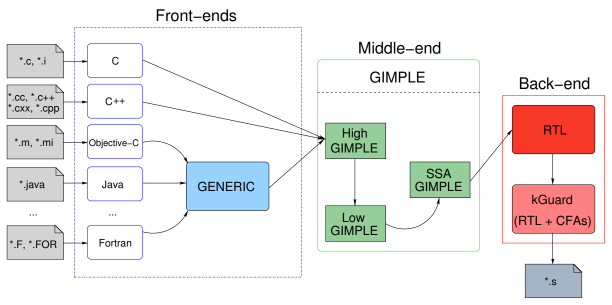


Figure 2: Architectural overview of GCC. The compilation process involves 3 distinct translators (frond-end, middle-end, back-end), and more than 250 optimization passes. kGuard is implemented as a back-end optimization pass.

4 Implementation

We implemented kGuard as a set of modifications to the pipeline of a C compiler. Specifically, we instrument the intermediate language (IL) used during the translation process, in order to perform the CFA-based confinement discussed in Section 3. Our implementation consists of a plugin for the GNU Compiler Collection (GCC) that contains the "de-facto" C compiler for building Linux and BSD kernels. Note that although other compilers, such as Clang and PCC, are capable of building much of Linux/FreeBSD and NetBSD/OpenBSD respectively, they are not officially supported by the corresponding development groups, due to the excessive use of the GNU C dialect in the kernel.

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 the translators without modifying them. Figure 2 illustrates the internal architecture of GCC. The compilation pipeline is comprised by 3 distinct components, namely the front-end, middle-end, and back-end, which transform the input into various ILs (*i.e.,* GENERIC, GIMPLE, and RTL). The kGuard plugin consists of $\sim$1000 lines of code in C and builds into a position-independent (`PIC`) dynamic shared object that is loaded by GCC. Upon loading 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.[8] 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 2).

[8]Currently, kGuard accepts 3 parameters: `stub`, `nop`, and `log`. `stub` provides 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 IL, 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 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 IL has been "lowered"). Third, we tightly couple the CFAs with the corresponding unsafe control 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.

5 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 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 20 instructions on average. Mind you that we also measured the effect of various NOP sled sizes, which was insignificant for the range 0 – 20.

5.1 Preventing Real Attacks

The main goal of the effectiveness evaluation is to apply kGuard on commodity OSs, and determine whether it can detect and prevent real-life ret2usr attacks. Table 1 summarizes our test suite, which consisted of a collection of 8 exploits that cover a broad spectrum of different flaws, including direct NULL pointer dereferences, control hijacking via tampered data structures (data pointer corruption), function and data pointer overwrite, arbitrary kernel-memory nullification, and ret2usr via kernel stack-smashing.

	x86 kernel			**x86-64 kernel**		
	`call`	`jmp`	`ret`	`call`	`jmp`	`ret`
CFA_M	20767	1803	—	17740	1732	—
$CFA_M opt$	2253	12	113053	1789	0	105895
CFA_R	6325	0	—	8780	0	—
Total	29345	1815	113053	28309	1732	105895

Table 2: Number of indirect branches instrumented by kGuard in the vanilla Linux kernel v2.6.32.39.

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 restrictions of Linux, by using one or more of the techniques discussed in Section 2.3.

5.2 Translation Overhead

We first quantify the additional time needed to inspect the RTL IL and emit the CFAs (see Section 4). Specifically, we measured the total build time with Unix's `time` utility, when compiling the v2.6.32.39 Linux kernel natively and with kGuard. On average, we observed a 0.3% increase on total build time on the x86 architecture, and 0.05% on the x86-64. Moreover, the size of the kernel image/modules was increased by 3.5%/0.43% on the x86, and 5.6%/0.56% on the x86-64.

In Table 2, we show 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 3.4. 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/`).

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	√	—
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 a negative reference counter (function pointer affected via tampered 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 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}	√	—
CVE-2010-4258	*function* pointer overwrite via arbitrary kernel memory nullification in *kernel/exit.c*	$\leq$ 2.6.36.2	√	√
EDB-15916	NULL *function* pointer overwrite via a signedness error in Phonet protocol (function pointer affected via tampered data structure)	2.6.34	√	√
CVE-2009-3234	ret2usr via kernel stack buffer overflow in *kernel/perf_counter.c* (return address is overwritten with user space memory)	2.6.31-rc1	√	√

√: detected and prevented successfully —: exploit unavailable

Table 1: Effectiveness evaluation suite. We instrumented 10 x86/x86-64 vanilla Linux kernels, ranging from v2.6.18 to v2.6.34, for assessing kGuard. We successfully detected and prevented all the listed exploits.

5.3 Performance Overhead

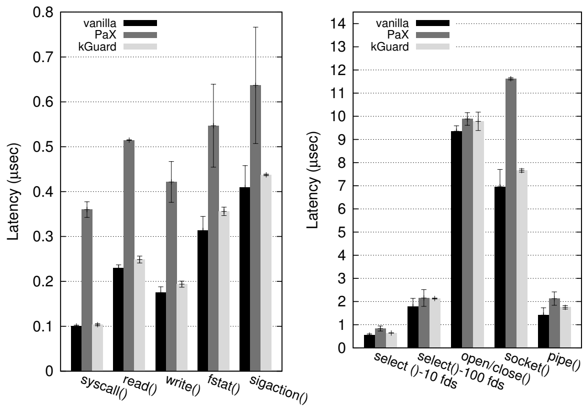
The injected CFAs also introduce runtime latency. We evaluated kGuard to quantify this overhead and establish a set of performance bounds for different types of system services. Moreover, we used the overhead imposed by PaX (*i.e.,* UDEREF [53] and KERNEXEC) as a reference. Mind you that on x86, PaX 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 location with non-execute permissions.

Macro benchmarks We begin with the evaluation of kGuard using 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, the same kernel patched with PaX, and instrumented with kGuard.

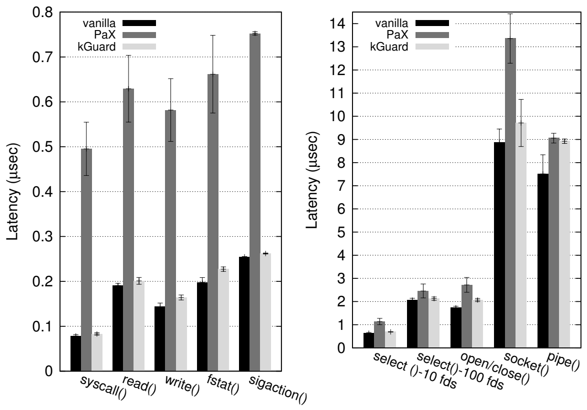
First, we measured the time taken to build a vanilla Linux kernel (v2.6.32.39), using the Unix `time` utility. On the x86, the PaX-protected kernel incurs a 1.26% run-time overhead, while on the x86-64 the overhead is 2.89%. In contrast, kGuard ranges between 0.93% on x86-64, and 1.03% on x86. Next, 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, and so forth. On average, kGuard's slowdown 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 the x86, and 0.01% and 1.07% on x86-64. In antithesis, kGuard's slowdown lies between 0.001% and 0.01%. Overall, our results indicate that in both x86 and x86-64 Linux the impact of kGuard in real-life applications is negligible ($\leq$1%).

Micro benchmarks Since the injected CFAs are distributed throughout many kernel subsystems, such as the essential `net/` and `fs/`, we used the LMbench [50] 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 with `sigaction`, inter-process communication (IPC) latency with `socket` and `pipe`, and process creation latency with `fork+{exit, execve, /bin/sh}`.

Figure 3 summarizes 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, kGuard's overhead for process creation (in both architectures) lies between 7.1% and 9.7%, whereas PaX ranges from 8.1% to 56.3%.



(a) x86



(b) x86-64

Figure 3: Latency overhead incurred by kGuard and PaX on essential system calls (x86/x86-64 Linux).

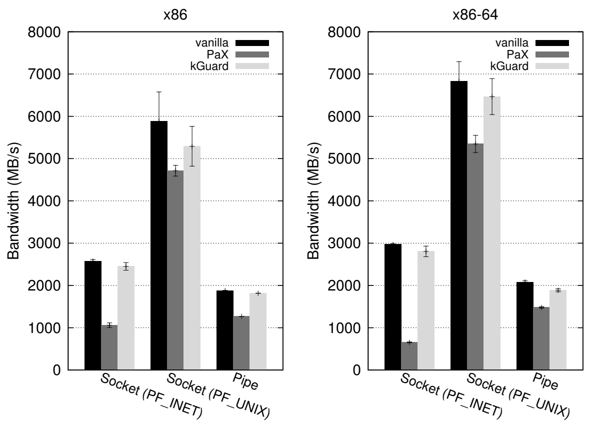


Figure 4: IPC bandwidth achieved by kGuard and PaX, using TCP (PF_INET), Unix sockets (PF_UNIX), and pipes.

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 4 shows our results (bandwidth is measured in MB/s). kGuard's slowdown ranges between 3.2% – 10% on x86 (average 6%), and 5.25% – 9.27% on x86-64 (average 6.6%). PaX's overhead lies between 19.9% – 58.8% on x86 (average 37%), and 21.7% – 78% on x86-64 (average 42.8%). Overall, kGuard exhibits lower overhead on x86-64, due to the fewer CFA_M guards (see Table 2). Recall that CFA_R confinement can be performed with just 3 additional instructions, and hence incurs less run-time overhead, whereas CFA_M might need up to 10 (*e.g.,* when we cannot optimize). However, the same is not true for PaX, since the lack of segmentation in x86-64 results in higher performance penalty.

6 Discussion and Future Work

Custom violation handlers kGuard's default violation handler appends a message in system log and halts the OS. We coerce assertion violations into a kernel fail-stop to prevent brute-force attempts and to avoid leaving the OS in inconsistent states (*e.g.,* by aborting an in-flight kernel thread that holds a lock). However, kGuard can be configured to use a custom handler. Upon enabling this option, our instrumentation becomes slightly different. Instead of overwriting offending branch targets with the address of our handler, we push the value of the branch target into the stack and invoke the handler directly. In the case of a CFA_R guard this means that the `mov` instruction (see Snippet 3) will be replaced with a `push` and `call`. CFA_M guards are modified accordingly.

This instrumentation increases slightly the size of our inline guards, but does not incur additional overhead, since the extra instructions are on the error path. Additionally, the custom violation handler has access to the location where the violation occurred, by reading the return address of the callee (pushed into the stack from `call`), as well as to the offending branch target (passed as argument to the handler). Using that information, one can implement adaptive defense mechanisms, including selective confinement (*e.g.,* deal with VMware's I/O backdoor that needs to "violate" protection domains), error virtualization [63], as well as forensic analysis (*e.g.,* dump the shellcode). The latter can be useful in honeypot setups for studying new ret2usr exploitation vectors.

Persistent threats By building upon the previous feature, we implemented a handler that actively responds to persistent threats (*i.e.,* users that repeatedly try to perform a ret2usr attack). Once invoked, due to a violation, it 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 the current execution path is holding a lock[9], then we fail-stop the kernel. Else, if the kernel is preemptible, we terminate all processes with the same `uid` of the offending process and prevent the user from logging in. Other possible approaches include inserting an exponentially increased delay for user logins (*i.e.,* make the bruteforce attack slow and impractical), activate CFA motion, *etc.*

Future considerations Currently, we investigate how to apply the CFA motion technique (see Section 3.5), while a kernel is running and the OS is live. Our early Linux prototype utilizes a dedicated kernel thread, which upon a certain condition, freezes the kernel and performs rewriting. Thus far, we achieve CFA relocation in a coarse-grained manner, by exploiting the suspend subsystem of the Linux kernel. Specifically, we bring the system to pre-suspend state for preventing any kernel code from being invoked during the relocation (note that the BSD OSs have similar facilities). 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 in the case of smartphone devices, dock station attach and charging. However, our end goal is to perform CFA motion in a more fine-grained, non-interruptible and efficient manner, without "locking" the whole OS.

7 Related Work

kGuard is inspired by the numerous compiler-based techniques that explicitly or implicitly constrain control flow and impose a specific execution policy. StackGuard [14] and ProPolice [34] are GCC patches that extend the behavior of the translator for inserting a canary word prior to the saved return address on the stack. The canary is checked again before a function return is performed, and execution is halted if it has been overwritten (*e.g.,* due to a stack-smashing attack). Stack Shield [1] is a similar extension that saves the return address, upon function entry, into a write-protected memory area that is not affected by buffer overflows and restores it before returning.

Generally, these approaches have limitations [9, 69]. However, they significantly mitigate real-life exploits by assuring that functions will always return to caller sites, incur low performance overhead, and do not require any change to the runtime environment or platform of the protected applications. For these reasons, they have been adopted by mainstream compilers, such as GCC, and enabled by default in many BSD and Linux distributions.

[9]In Linux, we can check if the kernel is holding locks by looking at the `preempt_count` variable in the current process's `thread_info` structure [48].

kGuard operates analogously, by hooking to the compilation process and dynamically instrumenting code with inline guards. However, note that we leverage the plugin API of GCC, and do not require patching the compiler itself, thus aiding the adoption of kGuard considerably. More importantly, since stack protection is now enabled by default, kGuard can be configured to offload the burden of dealing with the integrity of return control data to GCC. If random XOR canaries [14] are utilized, then any attempt to tamper with saved return addresses on the stack, for redirecting the privileged control flow to user space, will be detected and prevented. Hence, the protection of kernel-level `ret` instructions with CFAs can be turned off. Note that during our preliminary evaluation we also measured such a scenario. The average overhead of kGuard, with no `ret` protection, on system call and I/O latency was 6.5% on x86 and 5.4% on x86-64, while its impact on real-life applications was $\leq$ 0.5%. This "offloading" cannot be performed in the case of simple random canaries or terminator canaries. Nevertheless, it demonstrates that our approach is indeed orthogonal to complementary mitigation schemes, and operates nicely with confinement checks injected during compile time.

PointGuard [13] is another GCC extension that works by encrypting all pointers while they reside in memory and decrypting them before they are loaded into a CPU register. PointGuard could provide some protection against ret2usr attacks, especially if a function pointer is read directly from user-controlled memory [20]. However, it cannot deal with cases where an attacker can nullify kernel-level function pointers by exploiting a race condition [19] or supplying carefully crafted arguments to buggy system calls [23]. In such scenarios, the respective memory addresses are altered by legitimate code (*i.e.,* kernel execution paths), and not directly by the attacker. kGuard provides solid protection against ret2usr attacks by policing every computed control transfer for kernel/user space boundary violations.

Other compiler-based approaches include DFI [11] that enforces data flow integrity based on a statically calculated reaching definition analysis. However, the main focus of DFI, and similar techniques [3, 12, 33], is the enforcement of spatial safety for mitigating bounds violations and preventing bounds-related vulnerabilities.

Control-Flow Integrity (CFI) [2], Program Shepherding [43], and Strata [57], employ binary rewriting and dynamic binary instrumentation (DBI) for retrofitting security enforcement capabilities into unmodified binaries. The major issue with such approaches has been mainly the large performance overhead they incur, as well as the reliance on interpretation engines, which complicates their adoption. Program Shepherding exhibits $\sim$100% overhead on SPEC benchmarks, while CFI has an average overhead of 15%, and a maximum of 45%, on the same test suite. CFI-based techniques rewrite programs so that every branch target is given a label, and each 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). Unfortunately, CFI is not effective against ret2usr attacks. The integrity of the 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. Therefore, CFI can be subverted by prepending user-provided shellcode with the respective label.

As an example, consider again Snippet 1 and assume that the attacker has managed to overwrite the function pointer `sendpage` with an address pointing in user space. CFI will prepend the instruction that invokes `sendpage` with an inline check that fetches a label ID (placed right before the first instruction in functions that `sendpage` can point to), and compares it with the allowed label IDs. If the two labels match, the control transfer will be authorized. Unluckily, since the attacker controls the contents and rights of the memory that `sendpage` is now pointing, he can easily prepend his code with the label ID that will authorize the control transfer. Furthermore, Petroni and Hicks [55] noted that 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. CFI-based proposals can be combined with kGuard to overcome the individual limitations of each technique. kGuard can guarantee that privileged execution will always be confined in kernel space, thus leaving no other options to attackers than targeting kernel-level control flow violations, which can be solidly protected by CFI.

Garfinkel and Rosenblum proposed Livewire [36], which was the first system that used a virtual machine monitor (VMM) for implementing invariant-based kernel protection. Similarly, Grizzard uses a VMM for monitoring kernel execution and validating control flow [38]. For LMBench, he reports an average of 30% overhead, and a maximum of 74%, on top of VMM's performance penalty. SecVisor [62] is a tiny hypervisor that ensures the integrity of commodity OS kernels. It 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 modern CPUs that support virtualization in hardware, as well as kernel patching to add the respective hypercalls that authorize module loading. Along the same lines, NICKLE [56] 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) [55], which reports violations of the kernel's control flow due to the presence of rootkits. Similarly, Lares [54] and HookSafe [68] protect kernel hooks (including function pointers) from being manipulated by kernel malware. The focus of those techniques, however, has been kernel attestation and kernel code integrity [10], which is different from the control-flow integrity of kernel code. On the other hand, kGuard focuses on solving a different problem: *privilege escalation* via hijacked kernel-level execution paths. Although VMMs provide stronger security guarantees than kGuard, and SecVisor and NICKLE can prevent ret2usr attacks by refusing execution from user space while running in kernel mode, they incur larger performance penalties and require running the whole OS over custom hypervisors and specialized hardware. It is also worth noting that SecVisor and NICKLE cannot protect against execution hijacking via tampered data structures containing control data [18,20]. kGuard offers solid protection against that type of ret2usr due to the way it handles control data stored in memory.

Supervisor Mode Execution Prevention (SMEP) [37] is an upcoming Intel CPU feature, which prevents code executing in kernel mode from branching to code located in pages without the supervisor bit set in their page table entry. Although it allows for a confinement mechanism similar to PaX with zero performance penalty, it is platform specific (*i.e.,* x86, x86-64), requires kernel patching, and does not protect legacy systems.

8 Conclusions

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

Availability

The prototype implementation of kGuard is freely available at: `http://www.cs.columbia.edu/~vpk/research/kguard/`

Acknowledgments

We thank Michalis Polychronakis and Willem de Bruijn for their valuable feedback on earlier drafts of this paper. 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. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, or the Air Force.

References

[1] Stack Shield. `http://www.angelfire.com/sk/stackshield/`, January 2000.

[2] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity. In *Proceedings of the* 12th *ACM Conference on Computer and Communications Security (CCS)* (2005), pp. 340–353.

[3] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with WIT. In *Proceedings of the* 29th *IEEE Symposium on Security and Privacy (S&P)* (2008), pp. 263–277.

[4] AZAB, A. M., NING, P., WANG, Z., JIANG, X., ZHANG, X., AND SKALSKY, N. C. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the* 17th *ACM Conference on Computer and Communications Security (CCS)* (2010), pp. 38–49.

[5] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the* 7th *USENIX Annual Technical Conference (FREENIX track)* (2005), pp. 41–46.

[6] BICKFORD, J., O'HARE, R., BALIGA, A., GANAPATHY, V., AND IFTODE, L. Rootkits on Smart Phones: Attacks, Implications and Opportunities. In *Proceedings of the* 11th *International Workshop on Mobile Computing Systems and Applications (HotMobile)* (2010), pp. 49–54.

[7] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the* 6th *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2011), pp. 30–40.

[8] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, Sebastopol, CA, USA, 2005, ch. System Startup, pp. 835–841.

[9] BULBA AND KIL3R. Bypassing StackGuard and StackShield. *Phrack 5*, 56 (May 2000).

[10] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *Proceedings of the* 16th *ACM Conference on Computer and Communications Security (CCS)* (2009), pp. 555–565.

[11] CASTRO, M., COSTA, M., AND HARRIS, T. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the* 7th *Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 147–160.

[12] CASTRO, M., COSTA, M., MARTIN, J.-P., PEINADO, M., AKRITIDIS, P., DONNELLY, A., BARHAM, P., AND BLACK, R. Fast Byte-granularity Software Fault Isolation. In *Proceedings of the* 22nd *ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 45–58.

[13] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. PointGuard^{TM}: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the* 12th *USENIX Security Symposium (USENIX Sec)* (2003), pp. 91–104.

[14] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the* 7th *USENIX Security Symposium (USENIX Sec)* (1998), pp. 63–78.

[15] COX, M. J. Red Hat's Top 11 Most Serious Flaw Types for 2009. `http://www.awe.com/mark/blog/20100216.html`, February 2010.

[16] CVE. CVE-2009-1897. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897`, June 2009.

[17] CVE. CVE-2009-2692. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692`, August 2009.

[18] CVE. CVE-2009-2908. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2908`, August 2009.

[19] CVE. CVE-2009-3527. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3527`, October 2009.

[20] CVE. CVE-2009-3547. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3547`, October 2009.

[21] CVE. CVE-2010-2959. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2959`, August 2010.

[22] CVE. CVE-2010-3904. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3904`, October 2010.

[23] CVE. CVE-2010-4258. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4258`, November 2010.

[24] DAN ROSENBERG. `kptr_restrict` for hiding kernel pointers. `http://lwn.net/Articles/420403/`, December 2010.

[25] DE C VALLE, R. Linux `sock_sendpage()` NULL Pointer Dereference (PPC/PPC64 exploit). `http://packetstormsecurity.org/files/81212/Linux-sock_sendpage-NULL-Pointer-Dereference.html`, September 2009.

[26] DE RAADT, T. CVS-200910282103. `http://marc.info/?l=openbsd-cvs&m=125676466108709&w=2`, October 2009.

[27] DESIGNER, S. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63`, August 1997.

[28] DOSEMU. DOS Emulation. `http://www.dosemu.org`, June 2012.

[29] DOWD, M. Application-Specific Attacks: Leveraging The ActionScript Virtual Machine. Tech. rep., IBM Corporation, April 2008.

[30] EDB. EDB-9477. `http://www.exploit-db.com/exploits/9477/`, August 2009.

[31] EDB. EDB-17391. `http://www.exploit-db.com/exploits/17391/`, June 2011.

[32] EDB. EDB-18080. `http://www.exploit-db.com/exploits/18080/`, November 2011.

[33] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 75–88.

[34] ETOH, H. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, August 2005.

[35] FREEBSD. sysutils/vbetool doesn't work with FreeBSD 8.0-RELEASE and STABLE. `http://forums.freebsd.org/showthread.php?t=12889`, April 2010.

[36] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)* (February 2003).

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

[38] GRIZZARD, J. B. *Towards Self-Healing Systems: Reestablishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, 2006.

[39] HARDY, N. The Confused Deputy (or why capabilities might have been invented). *SIGOPS Operating Systems Review 22*, 4 (October 1988), 36–38.

[40] HUND, R., HOLZ, T., AND FREILING, F. C. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium (USENIX Sec)* (2009), pp. 383–398.

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

[42] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)* (2003), pp. 272–280.

[43] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium (USENIX Sec)* (2002), pp. 191–206.

[44] KORTCHINSKY, K. CLOUDBURST: A VMware Guest to Host Escape Story. In *Proceedings of the 12th Black Hat USA* (2009).

[45] LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHRAM, S. Defeating Return-Oriented Rootkits With "*Return-less*" Kernels. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (2010), pp. 195–208.

[46] LIAKH, S., GRACE, M., AND JIANG, X. Analyzing and Improving Linux Kernel Memory Protection: A Model Checking Approach. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)* (2010), pp. 271–280.

[47] LOSCOCCO, P., AND SMALLEY, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the 3rd USENIX Annual Technical Conference (FREENIX track)* (2001), pp. 29–42.

[48] LOVE, R. *Linux Kernel Development*, 2nd ed. Novel Press, Indianapolis, IN, USA, 2005.

[49] MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996, ch. Kernel Services, pp. 49–73.

[50] MCVOY, L., AND STAELIN, C. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1st USENIX Annual Technical Conference (USENIX ATC)* (1996), pp. 279–294.

[51] ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)* (2010), pp. 49–58.

[52] PAX. Homepage of The PaX Team. `http://pax.grsecurity.net`, June 2012.

[53] PAX TEAM. UDEREF. `http://grsecurity.net/~spender/uderef.txt`, April 2007.

[54] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)* (2008), pp. 233–247.

[55] PETRONI, JR., N. L., AND HICKS, M. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (October 2007), pp. 103–115.

[56] RILEY, R., JIANG, X., AND XU, D. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)* (2008), pp. 1–20.

[57] SCOTT, K., AND DAVIDSON, J. Safe Virtual Execution Using Software Dynamic Translation. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)* (December 2002), pp. 209–218.

[58] SECURITYFOCUS. Xbox 360 Hypervisor Privilege Escalation Vulnerability. `http://www.securityfocus.com/archive/1/461489`, February 2007.

[59] SECURITYFOCUS. BID 36587. `http://www.securityfocus.com/bid/36587`, October 2009.

[60] SECURITYFOCUS. BID 36939. `http://www.securityfocus.com/bid/36939`, November 2009.

[61] SECURITYFOCUS. BID 43060. `http://www.securityfocus.com/bid/43060`, September 2010.

[62] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (2007), pp. 335–350.

[63] SIDIROGLOU, S., LAADAN, O., PEREZ, C. R., VIENNOT, N., NIEH, J., AND KEROMYTIS, A. D. ASSURE: Automatic Software Self-healing Using REscue points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009), pp. 37–48.

[64] SPENGLER, B. On exploiting null ptr derefs, disabling SELinux, and silently fixed linux vulns. `http://seclists.org/dailydave/2007/q1/224`, March 2007.

[65] STEINBERG, U., AND KAUER, B. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (2010), pp. 209–222.

[66] TINNES, J. Bypassing Linux NULL pointer dereference exploit prevention (mmap_min_addr). `http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html`, June 2009.

[67] WANG, Z., AND JIANG, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)* (2010), pp. 380–395.

[68] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (2009), pp. 545–554.

[69] WILANDER, J., AND KAMKAR, M. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)* (2003).

[70] WINEHQ. Run Windows applications on Linux, BSD, Solaris and Mac OS X. `http://www.winehq.org`, June 2012.

[71] WOJTCZUK, R. Subverting the Xen hypervisor. In *Proceedings of the* 11th *Black Hat USA* (2008).

[72] ZENG, B., TAN, G., AND MORRISETT, G. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the* 18th *ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 29–40.

A Step-by-step Analysis of the `sendpage` ret2usr Exploit

Figure 5 illustrates the steps taken by a malicious process to exploit the vulnerability shown in Snippet 1 (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-level address of every supported system call indexed by system call number (0xc01039db).[10] 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 privileges.

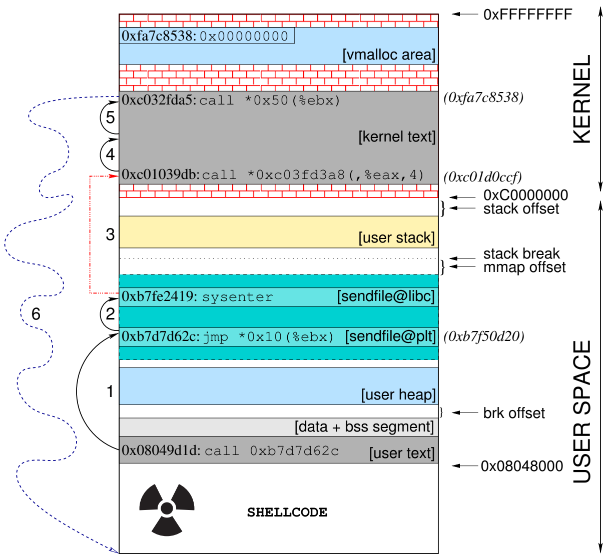


Figure 5: Control transfers that occur during the exploitation of a ret2usr attack. The `sendfile` system call, on x86 Linux, causes a function pointer in kernel to become NULL, illegally transferring control to user space code.

B GCC RTL Instrumentation Internals

`branchprot_instrument()`, our instrumentation callback, is invoked by GCC's pass manager for every translation unit after all the RTL optimizations have been applied, and exactly before target code is emitted. At that 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 CFA_R and CFA_M guards. The CFA_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 Snippet 3. 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. Note that 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. CFA_M instrumentation is performed in a similar fashion.

[10] The address 0xc03fd3a8 corresponds to the kernel-level memory address of `sys_call_table`.