kGuard

Lightweight Kernel Protection

VASILEIOS P. KEMERLIS, GEORGIOS PORTOKALIDIS,
ELIAS ATHANASOPOULOS, AND ANGELOS D. KEROMYTIS

Vasileios Kemerlis is a PhD student in the Department of Computer Science at Columbia University. His research interests are mainly in software and systems security, with a focus on automated software hardening. vpk@cs.columbia.edu

Georgios Portokalidis is a postdoctoral researcher in the Department of Computer Science at Columbia University. He obtained his doctorate from the Vrije Universiteit in Amsterdam. His research interests are mainly around the area of systems security, but extend to network monitoring, operating systems, and virtualization technologies. porto@cs.columbia.edu

Elias Athanasopoulos holds a BS in physics from the University of Athens, and an MS and PhD from the University of Crete. He is currently a Marie Curie postdoctoral fellow with Columbia University. elathan@cs.columbia.edu

Angelos D. Keromytis is an Associate Professor of Computer Science at Columbia University. His research interests revolve around systems and software security and reliability. He received his PhD in 2001 from the University of Pennsylvania. angelos@cs.columbia.edu

Kernel exploits have become increasingly popular over the past several years. We have developed kGuard, a cross-platform system that defends the operating system (OS) against a widespread class of kernel attacks. We describe how these attacks work and how kGuard protects the kernel with only a small decrease in performance.

The OS kernel is becoming an attractive target for attackers. The rising number of kernel vulnerabilities discovered and reported attest to this (see Figure 1). The reasons behind this trend are numerous. First, the number of applications running (continuously) with administrative privileges has significantly decreased, meaning that an attacker compromising such programs remotely gains only limited power over the underlying system. Additionally, programs have become harder to exploit due to the various defense mechanisms already adopted by modern OSes, such as address space layout randomization and stack smashing protection. The most interesting reason is probably that vulnerabilities such as NULL pointer dereference bugs, which were thought to be impractical, hard to exploit, and had not received significant attention by the security community, can be used with ease against the kernel to gain elevated privileges. In fact, some researchers proclaimed 2009 as "the year of the kernel NULL pointer dereference flaw" [2]. Last, exploiting kernel bugs has the added benefit of allowing attackers to mask their presence on the compromised systems (e.g., by hiding processes or files).

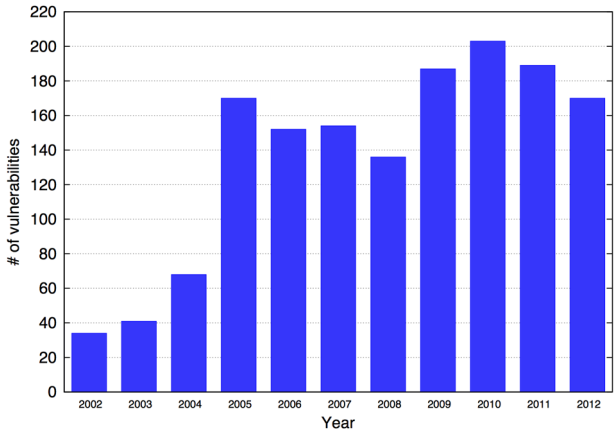

Figure 1: Kernel vulnerabilities (per year) reported to NIST. Over the past decade, the distinct number of CVE identifiers assigned to kernel vulnerabilities has increased by a factor of 5.

Kernel attacks are facilitated by the fact that user and kernel space (i.e., the memory area where user applications and the kernel reside, respectively), are weakly separated in modern OSes. As a result, direct transitions from more to less privileged protection domains (i.e., kernel to user space) are permissible, even though the reverse is not. This is what transforms NULL pointer dereference bugs from system instability vulnerabilities to privilege escalation threats. When exploited successfully, they enable local users to execute arbitrary code with kernel privileges, by redirecting the control flow of the kernel to user-controlled memory. Such return-to-user (ret2usr) attacks have affected all major OSes, including Windows, Linux, and the BSDs. These attacks are not limited to x86/x86-64 systems, but have also targeted the ARM, DEC, and PowerPC architectures.

Previous approaches to the problem are either impractical for deployment in certain environments or can be easily circumvented. For example, 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). This scheme has been circumvented by various means and is not backwards compatible. The PaX [8] patch for x86 and x86-64 Linux kernels does not exhibit the same shortcomings, but greatly increases system call and I/O latency. 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; 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. Running a complex VMM 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. Addressing the problem in hardware is the most efficient solution, but even though Intel has recently announced a new CPU feature, named SMEP [5], to thwart such attacks, hardware extensions are oftentimes adopted slowly by OSes. More importantly, other vendors have not publicly announced similar extensions.

kGuard is a lightweight solution to the problem. kGuard consists of a compiler plugin that augments kernel code with control-flow assertions, which ensure that privileged execution remains within its valid boundaries and does not cross to user space. This is achieved by identifying all exploitable control transfers during compilation, and injecting compact dynamic checks to attest that the kernel remains confined. kGuard is to some extent related to previous research on control-flow integrity (CFI) [1]; however, CFI is not effective against ret2usr attacks, because its integrity is only guaranteed if the attacker cannot overwrite the code of the protected binary or execute data. (During a ret2usr attack the control flow is redirected into memory pages whose contents and permissions are fully controlled by the attacker.)

Background

Virtual Memory Organization

Commodity OSes 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 memory model for minimizing the overhead of operations that cross protection domains, such as system calls, interrupts, and exceptions. Specifically, Windows and UNIX-like OSes divide

virtual memory into user and kernel space. The former hosts user processes, while the latter holds kernel code and data, kernel extensions (modules), and device drivers. In most architectures, the separation between the two spaces is assisted and enforced by the following hardware features: CPU modes (or protection rings), a memory management unit (MMU), and special-purpose instructions. The x86/x86-64 instruction set architecture (ISA) supports four protection rings, with the kernel running in the most privileged one (ring 0) and user applications in the least privileged (ring 3). In fact, modern x86/x86-64 CPUs have more than four rings; hardware-assisted virtualization and System Management Mode are colloquially known as ring -1 and -2, respectively. Similarly, the PowerPC and MIPS platforms have two CPU modes, SPARC has three, and ARM seven. All these architectures feature an MMU, typically programmed using privileged special-purpose instructions, which implements virtual memory and ensures that memory assigned to a certain ring is not accessible by the less privileged ones.

Kernel Exploitation

Code running in user space cannot directly access or jump into the kernel, and hence, special-purpose instructions and hardware facilities (i.e., interrupts and exceptions) are provided for crossing the user/kernel boundary. Nevertheless, while executing privileged code, complete and unrestricted access to all memory and system objects is available. For example, when servicing a system call for a process (or during interrupt/exception handling) the kernel executes within the context of a preempted process and can directly access user memory to store the result of the call or read user data.

At the same time, OS kernels, which are mostly written in type-unsafe languages and assembly, suffer the same software flaws that plague applications. For instance, buffer and integer overflows, pointer arithmetic bugs, use-after-free vulnerabilities, and signedness errors can all be exploited to corrupt kernel memory and hijack control flow, thus executing arbitrary code with elevated privileges. The ability to trigger such a bug in the kernel, from a local process, provides a unique standpoint to attackers who totally control (i.e., both in terms of permissions and contents) part of the address space available to the kernel at any given time. In other words, "shellcode" can be executed with kernel rights by hijacking a privileged execution path and redirecting it to user space.

ret2usr Attacks

ret2usr attacks have become the most popular kernel exploitation method, for which a wealth of defensive mechanisms exists [7, 8, 5]. They are manifested by overwriting kernel data with user space addresses, after exploiting memory safety bugs in kernel code. As expected, attackers typically aim for control data [10], such as return addresses, jump tables, and function pointers, since these facilitate arbitrary code execution; however, pointers to critical data structures, frequently stored in kernel stack or heap, are also favored targets, since their contents can be tampered with by mapping fake copies in user space [9]. Most exploits of that kind target data structures that contain function pointers, or data that affect kernel execution, so as to diverge the control flow to arbitrary (typically user-controlled) places.

The end effect of these attacks is that the kernel is hijacked and control is redirected to user space code. Typically, ret2usr exploits use a multi-stage shellcode,

where the first stage lies in user space and glues together kernel functions (i.e., the second stage shellcode) that perform privilege escalation or execute a rootshell. We refer to this type of exploitation as return-to-user [7] because it resembles the older return-to-libc [4] technique that redirected control to existing code in the C library. ret2usr attacks are yet another incarnation of the confused deputy problem [6], where a user fools the kernel (deputy) into misusing its authority and executing arbitrary, non-kernel code with elevated privileges.

kGuard

kGuard consists of a cross-platform GCC plugin that enforces address space segregation without relying on special hardware or architecture-specific features [8, 5]. It protects the kernel from ret2usr attacks with low-overhead by building on the following security primitives: *inline monitoring* and *code diversification*.

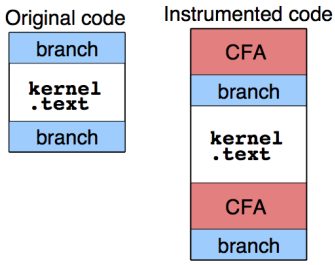

Figure 2a: CFA-based confinement. The injected guards perform a small runtime check before each computed branch to verify that the target address is in kernel space.

```
            cmp  $0xc0000000,%ebx
            jae  call_lbl
            mov  $0xc05af8f1,%ebx
call_lbl:   call *%ebx
```

Figure 2b: CFA_R guard gets applied on an indirect `call` in x86 Linux (drivers/cpufreq/cpufreq.c).

Inline Monitoring

kGuard augments exploitable control transfers, at compile time, with dynamic control-flow assertions (CFAs) that, at runtime, prevent the unconstrained transition of privileged execution paths to user space. Figure 2a illustrates the concept. The injected CFAs perform a small runtime check before each indirect branch 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; however, custom handlers are also supported for facilitating forensic analysis (e.g., dumping the shellcode for studying new ret2usr exploitation vectors), selective confinement (i.e., avoiding instrumenting "legitimate" boundary violators such as VMware's I/O back door), and providing protection against persistent threats.

CFA guards come in two flavors, namely CFA_R and CFA_M, depending on whether the protected control transfer uses a register or memory operand. Figure 2b shows an example of a CFA_R guard. The code is from the `show()` routine of the cpufreq driver in x86 Linux. kGuard instruments the computed branch (`call *%ebx`) with three additional instructions. First, the `cmp` instruction compares the `ebx` register with the lower bound kernel address 0xC0000000. The same is also true for x86 FreeBSD/NetBSD (OpenBSD maps the kernel to the upper 512 MB of the virtual address space, and hence, its base address in x86 CPUs is located at 0xD0000000),

whereas for x86-64 the check should be with address 0xFFFFFFFF80000000. In case 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 the `call`, which will invoke the violation handler.

Similarly, CFA_M guards confine indirect branches that use memory operands; however, these guards not only assert that the branch target is within the kernel address space, but also ensure 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 attackers have managed to tamper with data structures that contain control data, by overwriting data pointers to such structures with user space addresses and mapping fake copies in user space. Interested readers are referred to our recent USENIX Security paper for more information regarding the CFA_M guards [7].

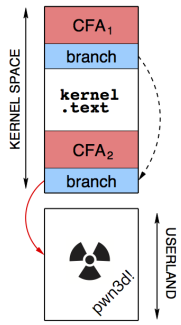

Figure 3: Subverting kGuard using bypass trampolines. CFA_1 succeeds since the address of the second branch (trampoline) is in kernel space. CFA_2 is completely bypassed by jumping directly to the branch instruction.

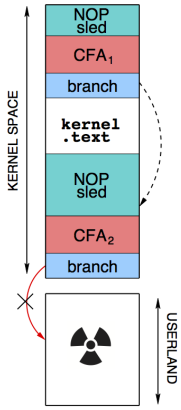

Figure 4: Code inflation reshapes the kernel's text area by inserting NOP sleds of random length at the beginning of each CFA.

Code Diversification

CFA_R and CFA_M guards provide reliable protection against ret2usr attacks only if the attacker exploits a kernel bug that allows him partially to control a computed branch target (e.g., by zeroing out certain bytes); however, vulnerabilities where the attacker can overwrite kernel memory with arbitrary values also exist [3]. When such flaws are present, exploits could attempt to bypass kGuard.

BYPASS TRAMPOLINES

To subvert kGuard, an attacker must be able to determine the address of a (indirect) control transfer instruction inside the text segment of the kernel. Moreover, she should also be able to control the value of its operand reliably (i.e., its branch target). We refer to that branch as a *bypass trampoline*. Note that in ISAs with overlapping variable-length instructions, finding an embedded opcode sequence that translates directly to a control branch in user space is possible. 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, as depicted in Figure 3. 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.

CODE INFLATION

This technique reshapes the kernel's text area (see Figure 4). kGuard begins 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, it continues by inserting NOP sleds of random length at the beginning of each CFA. The end result is that the location of every computed 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 to higher memory addresses. The size of the initial sled is chosen by kGuard based on the target architecture.

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, the probability of successfully guessing the address of a bypass trampoline increases; however, assigning safe file permissions to the kernel's text, modules, and debugging symbols is not a limiting factor. 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. In fact, this is considered standard practice in OS hardening, and is automatically enabled in PaX and similar patches, as well as in the latest Ubuntu Linux releases. Also note that the kernel should harden access to the system message ring buffer (`dmesg`) and certain files in the `proc` pseudo-file system, in order to prevent the leakage of kernel addresses.

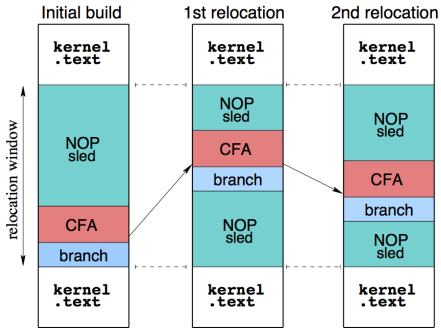

Figure 5: CFA motion synopsis. kGuard relocates each inline guard and protected branch, within a certain window, by routinely rewriting the text segment of the kernel.

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, for more hardening against bypasses. Figure 5 illustrates the concept. During compilation, kGuard emits information regarding each injected CFA, which can be used later to relocate the respective code snippets. Specifically, kGuard logs the exact location of the CFA inside the 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, kGuard 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, kGuard only supports CFA motion during kernel bootstrap. That said, keep in mind that ret2usr violations are detected at runtime, and hence one false guess is enough to identify the attacker and restrict his capabilities (e.g., by revoking his access to prevent brute-force attempts).

Results and Next Steps

The effectiveness of kGuard has been experimentally assessed by instrumenting different vanilla Linux kernels, both in x86 and x86-64 architectures, and testing them against real 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. As expected, kGuard was able to detect and prevent exploitation successfully in all cases. For more information regarding the evaluation suite, please refer to our paper in USENIX Security '12 [7].

kGuard exhibits lower overhead than previous work. On average, it imposes a 11.4% overhead on system call and I/O latency on x86 Linux, and 10.3% on x86-64, as reported by the LMbench micro-benchmark suite. In the case of IPC bandwidth, it exhibits an average slowdown of 6% on x86, and 6.6% on x86-64. Additionally, the size of a kGuard-compiled kernel grows between 3.5% and 5.6%, due to the inserted checks, while the impact on real-life applications, such as the MySQL RDBMS and Apache Web server, is minimal ($\lesssim 1\%$).

Future steps include investigating how to apply the CFA motion technique while a kernel is running and the OS is live. Currently, we have developed a Linux prototype that utilizes a dedicated kernel thread, which upon a certain condition, freezes the kernel and performs rewriting. Thus far, we have achieved 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 to prevent any kernel code from being invoked during relocation (note that the BSD OSes have similar capabilities); however, our end goal is to perform CFA motion in a more fine-grained, non-interruptible and efficient manner, without "locking" the whole OS. Further in the future, we also plan to explore custom fault handlers that perform error virtualization for automatically recovering from attacks.

Conclusion

kGuard is a fast and flexible cross-platform solution that protects the kernel from ret2usr attacks. It works by injecting fine-grained inline guards during the translation phase that are resistant to bypass, and does not require any modification to the kernel or additional software such as a VMM. kGuard can safeguard both 32- and 64-bit OSes that map a mixture of code segments with different privileges inside the same scope and are vulnerable to ret2usr exploits. We believe that kGuard strikes a balance between safety and functionality, and provides comprehensive protection from a widespread class of attacks.

Availability

kGuard is freely available at: http://www.cs.columbia.edu/~vpk/research/kguard/.

Acknowledgments

We thank Georgios Kontaxis for his valuable feedback on earlier drafts of this manuscript. This work was supported by DARPA, the US Air Force, and ONR through Contracts DARPA-FA8750-10-2-0253, AFRL-FA8650-10-C-7024, and N00014-12-1-0166, 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, the Air Force, or ONR.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," *Proceedings of the 12th ACM Conference on Computer and Communications Security* (CCS), 2005, pp. 340–353.

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

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

[4] S. Designer, "Getting Around Non-Executable Stack (and Fix)," August 1997: http://seclists.org/bugtraq/1997/Aug/63.

[5] V. George, T. Piazza, and H. Jiang, "Technology Insight: Intel Next Generation Microarchitecture Codename Ivy Bridge," September 2011: www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf.

[6] N. Hardy, "The Confused Deputy (or Why Capabilities Might Have Been Invented)," . SIGOPS Operating Systems Review, vol. 22, no. 4, October 1988, pp. 36–38.

[7] V.P. Kemerlis, G. Portokalidis, and A.D. Keromytis, "kGuard: Lightweight Kernel Protection Against Return-to-User Attacks," *Proceedings of the 21st USENIX Security Symposium,* USENIX Association, 2012, pp. 459–474.

[8] PaX Team home page: http://pax.grsecurity.net, accessed September 2012.

[9] SecurityFocus, "Linux Kernel 'pipe.c' Local Privilege Escalation Vulnerability," November 2009: http://www.securityfocus.com/bid/36901/info.

[10] Virtual Security Research, "Linux RDS Protocol Local Privilege Escalation," October 2010: http://www.vsecurity.com/resources/advisory/20101019-1/.