Ensuring the Minimality of Included Kernel Components

Dannie M. Stanley
Department of Computer Science
Purdue University
West Lafayette, IN 47907
email: ds@cs.purdue.edu

***Abstract*—Kernels shipped with general-purpose operating systems often contain extraneous code. The unnecessary kernel code is a security liability. The code may contain exploitable vulnerabilities or may be pieced together using return/jump-oriented programming to attack the system. Run-time kernel *minimization* can be used to improve the security of an operating system kernel.**

Our hypothesis is the following: *It is possible to strengthen the defenses of commodity, general-purpose computer operating systems by increasing the diversity of, validating the integrity of, and ensuring the minimality of the included kernel components without modifying the kernel source code. Such protections can therefore be added to existing, widely-used, unmodified operating systems to prevent malicious software from executing in supervisor mode.*

To test our hypothesis we design and implement six distinct kernel security mechanisms, protect many unmodified commodity operating systems kernels using the mechanisms, and assail the protected kernels using common attack techniques including return-oriented programming and kernel rootkits.

I. Introduction

Code injection prevention and authentication techniques, such as those described in [1], are still vulnerable to return and jump oriented programming (ROP and JOP respectively) attacks because the payload executable code is already present in kernel memory. The instructions are merely reused in unintended and malicious ways. With large commodity operating system, the amount of reusable code is abundant.

As described by Bryant et. al. in their work on Poly^2, the operating system kernel often includes code that is unnecessary for the applications that are running on the system [2]. General-purpose operating system vendors include kernel code for many possible hardware profiles and system use cases to cover the most uses with a single piece of software. Though this makes the kernel executable larger, it reduces the number of versions that the vendor must support and update. This extra code, though perhaps convenient for both end-users and vendors is a security liability.

The seminal work on return-oriented programming is titled, in-part, "innocent flesh on the bone," a crude reference to the code flesh left on the bone, the kernel in our case, and available for devouring by attackers [3]. In Poly^2, researchers remove unnecessary code at compile time. Though there has been work done in adding code to a kernel at run-time, such as loadable kernel modules, there has not been work done in removing code from a kernel at run-time. In our work we intend to garner the benefits of the kernel reduction technique described in Poly^2 at run-time and reduce the vulnerability of commodity operating system kernels.

Our hypothesis is that it is possible to improve the security of a kernel against return and jump oriented programming attacks by deactivating extraneous kernel code at run-time, thereby limiting the supply of reusable instructions that can be used to construct return-oriented gadgets.

To test our hypothesis, we introduce two novel techniques for run-time kernel minimization. The first is an out-of-the-box function eviction technique. The second is a kernel-based non-executable page technique. We implement a prototype for the out-of-the-box technique and report the results.

II. Related Work

Work has been done in design-time techniques for minimizing the code contained in a kernel. Minix and Mach are microkernel designs that attempt to include only necessary components in the kernel space [4], [5].

Work has been done in compile-time techniques for reducing the code contained in a kernel. In the Poly^2 framework, kernels are minimized at compile-time based on the kernel mechanisms needed for specific applications to improve security[2]. For the operating system family named Choices, the kernel is minimized at compile-time based on the functions needed for a specific embedded system to make the kernel as small as possible [6].

Work has been done in run-time techniques for optimizing the code contained in a kernel. The Synthesis kernel optimizes system functions at run-time that get reused frequently to improve system performance [7].

III. Organization

The remainder of this paper is organized as follows: we begin in Section IV by laying out the design of our minimization techniques followed by a description of our KIS implementation in Section V. In Section VI, we evaluate the security merits of our approaches. Finally, in Section **??**, we summarize our contributions and findings.

IV. Design

A. Problem

Modern commodity operating systems are equipped with monolithic kernels that, by design, contain code that does not strictly need kernel-level privileges. Since 1991 the Linux kernel has grown in size from 10,000 lines of code (LOC) to 15,004,006 LOC in 2012. From 2010 to 2012 it more than doubled in size. Likewise, the Windows NT Kernel has 50 million LOC by some estimates [1]. This excessive code increases the risk of kernel-level exploitation.

We are motivated to reduce this risk by dynamically applying the *economy of mechanism* to the operating system kernel code; to prune the kernel back to the minimum amount of instructions that are required for each specific system use case and hardware configuration at run-time. To keep with the carnivorous analogy, we intend to remove as much kernel flesh from the bone as possible. We refer to this technique as *run-time kernel minimization*.

B. Approach

In this section we describe the general approach to run-time kernel minimization. We introduce a run-time kernel specialization *security monitor* named KIS. KIS is an acronym borrowed, in-part, from a common design principle named KISS that stands for "keep it simple, stupid." The economy of mechanism security principle can be described colloquially as the KISS principle. For our use, KIS stands for "kernel instructions specialization" *and* "keep it simple." KIS is the mechanism that deactivates and activates code at run-time.

1) Code Deactivation: We have devised two general techniques for deactivating instructions at run-time. The first is function-level code deactivation. The second is page-level code deactivation.

The function-level deactivation technique modifies kernel functions that are resident in memory. Offline, the kernel is analyzed and a profile is generated. The profile is a list of pairs. The first element in each pair is the address of a kernel function. The second element in each pair is the size of the function in bytes. Each pair can be used to define the byte range of every kernel function.

Online, after the kernel has been loaded into memory, for each function that must be deactivated, we will refer to these functions as mutant functions. KIS replaces the mutant function body with alternative instructions. The form of the alternative instructions depends on whether or not the mutant function can be restored to its original. If the mutant function must never be restored, then the mutant function may be filled with a return instruction, a NOP sled, and a final return instruction. If instructions are fetched from anywhere in the mutant function, execution will immediately return to the caller. Depending on the callee-caller contract, some register restoring may also be necessary. As described in [8], the return value of "-1" can be used as an impostor return value that is handled by some caller code. If the mutant function should truly never be used by the system, then the mutant function will never be called legitimately and illegitimate calls will fail gracefully in many cases.

If mutant function restoration is allowed, then the mutant body is replaced with an interrupt instruction (INT3) or series of interrupt instructions that, if executed, would pass control to an interrupt handler. In the case that KIS is located "out of the box" in a hypervisor, the interrupt would be raised to the hypervisor first and KIS could manage the restoration of the mutant. In the case that KIS is located in the kernel, KIS would have to be installed as an interrupt handler.

In both cases, KIS removes the original mutant function bodies. Any return-oriented programming attacks that rely on the original mutant body will hampered, if not completely prevented.

The second code deactivation technique is similar to the NICKLE-KVM approach described in [1]. Rather than deactivating the code at the function-level, we deactivate code at the page-level. If a page contains code that must be deactivated, we call this a mutant page, then the corresponding page table entry is set to non-executable. If instructions located in that page are fetched, control would then be passed to KIS by way of a page fault exception.

Because the page-level deactivation is less granular than the function-level technique previously described, special consideration must be given to pages that contain both code that should be executable and code that should not be executable. If such mixed pages exist, they should likely remain executable. If KIS is located in the hypervisor, then the mixed mutant pages may be set non-executable. This would reduce the size of the kernel until the page-adjacent approved code was activated thereby activating the excessive code at the same time.

The Poly^2 approach is to remove unnecessary code once and never reintroduce it. This occurs at compile time. If an analogous run-time approach is used, then we would need to calculate offline the kernel code required by the applications that will be running on the system and deactivate the unneeded instructions at run-time. An alternative approach is to deactivate code liberally, leaving only essential components activated, and reactivate code on an as-needed basis. Deciding what code to reactivate is described later.

2) KIS Security Monitor: The purpose of the security monitor is to deactivate and reactivate code and log related events. KIS may be activated by system exceptions such as a page fault or debug interrupt previously described or by a message passing. Additionally, it may itself be monitoring the system for specific conditions and trigger code to be activated or deactivated based on predefined events.

The security monitor can be used passively for anomaly *detection*. If the deactivated code should not be executed but is, then the event is logged and used as part of an intrusion detection analysis. It is passive in that, in this mode, it has a permissive re-inclusion policy that allows for the code to

[1] http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/

be reactivated. In this mode KIS would act primarily as a detection mechanism rather than a prevention mechanism.

KIS may also take a more active role. If the deactivated code should not be executed but is, then the event is logged and used as part of an intrusion detection analysis. If no reactivation policy is available, then KIS would terminate the process. If a reactivation policy is available that permits reactivation, then the security monitor reactivates the code.

Various deactivation/reactivation policies may be constructed. We have devised two examples of such policies. The first is based on an event model; the second on a control flow graph technique such as the one described by [9].

One example of an event driven policy is the following: for a kernel that supports LKM, at system boot many modules are loaded automatically to service the hardware present in the system. Suppose that this system is a server system in a data center that never needs any new hardware hot-plugged. Once the system is booted, the module loading code can be deactivated. A "modules loaded" message could be passed to KIS or KIS could be configured to detect that event and deactivate the module loading code.

A second example of a reactivation policy is *on-demand* activation. Suppose that after system boot-up is complete all non-essential kernel code is deactivated. The code is then reactivated on an as-needed basis. A permissive approach is to reactivate code on-demand. In this scenario, KIS verifies that the first instruction fetched from the mutant, function or page, is the first instruction byte of a function body.

Though permissive, this is a minimally intrusive approach that allows the system to run nearly unfettered while still preventing ROP and JOP attacks from reusing instructions originally found in the mutants. In the case that the kernel memory is not visible to the attacker, for example /dev/kmem access is prevented in the Linux kernel, then the ROP and JOP attacks would be formulated against the offline version of the kernel code, the code that contains all of the original mutant function instructions. Online, any attack that depends on a mutant instruction would fail. This approach is described further and evaluated in Sections V and VI.

The previously described permissive on-demand activation policy can be constrained further using control-flow analysis. Offline, a control-flow graph is generated for the kernel. The graph describes all parent-child relationships between code regions and all valid control flow entry points. For example, if code is deactivated at the function level, then the first time that code is fetched, it should be on a function boundary and at least one parent function should already be activated. If the code is deactivate at the page level, then the entry control-flow validation would work similarly. However, for code that crosses page boundaries, additional entry points would be required.

3) KIS Location: The kernel is intended to run with all of its instructions intact. If essential instructions are removed, then a kernel fault will likely destabilize if not destroy the kernel operation. If, for example, KIS is placed in the system kernel, the page fault interrupt handler must be able to service page faults at all times. If KIS is placed outside of the system and code may be reactivated on demand, then the number of essential components may be minimal.

Out-of-the-box security mechanisms, such as those described in [**?**], [**?**, ds] are tamper-resistant against guest-based attacks. In-kernel security mechanisms are subject to vulnerabilities of the kernel that it is trying to protect. In situations where virtualization is is used, such as in a cloud hosting environment, KIS should be located in the hypervisor for maximum security benefits. However, in situations where virtualization is not appropriate, such as resource constrained mobile devices, KIS should be located in the kernel.

Out-of-the-box security mechanisms must be designed carefully as not to impose too many virtual machine exits (VM exits). VM exits are computationally expensive. One advantage to an in-kernel KIS design is that it can be more intrusive and code can be deactivated on a per process basis.

For example, consider the following process-level deactivation scenario. Suppose a sandboxed shell process (SSHELL) is used for launching all processes subjected to kernel minimization. The SSHELL has all non-essential kernel code pages marked as non-executable. When a process is created via a fork, the virtual memory page table is copied but the physical memory is not, thereby propagating the non-executable kernel code pages to all child processes. The KIS-enabled kernel has a modified page-fault handler. When an instruction is fetched from a mutant page, the KIS-assisted page-fault handler will detect if the fault was caused by an NX permissions violation. If so, then KIS will determine whether or not to allow the page to be reactivated based on some previously described reactivation policy. If allowed, then the mutant page is reactivated and the page table entry is set to executable for the duration of the process and all spawned processes. All processes, such as threads and spawned processes, that share this process's page table have the same activated kernel code. Unlike previously described designs, including Poly^2 that are system-level, this design is process-level; though the kernel is mapped into each process, SSHELL ancestor processes will not have access to all of the kernel code.

V. Implementation

We implemented our run-time kernel minimization security monitor, KIS, using an Intel i5-2410M 2.30GHz processor. Both our host and guest systems run an *unmodified* version of Ubuntu (11.04 2.6.38-8-generic). The host runs the 64 bit version of Ubuntu (x86_64) and the guest runs the 32 bit version (i686). We added our security monitor to the Linux Kernel Virtual Machine (KVM) version kvm-kmod-2.6.38-rc7.

Our run-time kernel minimization implementation closely resembles the permissive on-demand method described previously in Section IV-B1. KIS is placed in a Linux KVM hypervisor and protects a single guest running as a QEMU/KVM virtual machine. Guest instructions are deactivated at the function level. Functions are reactivated when its first function instruction is executed. Though this technique is permissive, it prevents ROP and JOP that reuse instructions directly

preceding free-branch instructions such as return instructions. Such instructions, that are suitable for reuse, do not occur in the function preamble. As a result ROP and JOP gadgets will *not* naturally trigger on-demand function activation.

KIS uses virtual machine introspection to intercept specific guest events. The first event intercepted is the completion of system startup. For the Linux kernel, we purposefully chose a point in the boot sequence when the kernel is fully loaded into memory and has already patched itself. This occurs right before the kernel function named init_post is executed. We refer to this event as the INIT event. The functions that are executed prior to the INIT event are not protected. Many of those functions are part of the ELF file section named .init.text and are deallocated shortly after booting completes and are therefore likely not part of an ROP/JOP attack.

When the INIT event occurs, our security monitor takes control. It reaches into the virtual machine and deactivates *all* kernel functions. To deactivate functions it replaces the first byte of every function with a INT3 byte (0xCC). When the INT3 byte is executed by the guest at some later time, it causes a software interrupt to occur. This interrupt is raised to the hypervisor and our security monitor handles the interrupt. If the interrupt is caused by the first byte of a function, then the original byte is replaced. Because INT3 is an interrupt and not an exception, when the virtual machine resumes, it tries the instruction again. This time however, the original instruction is in place and the function executes normally. For this prototype the function remains activated for the duration of the system.

VI. Evaluation

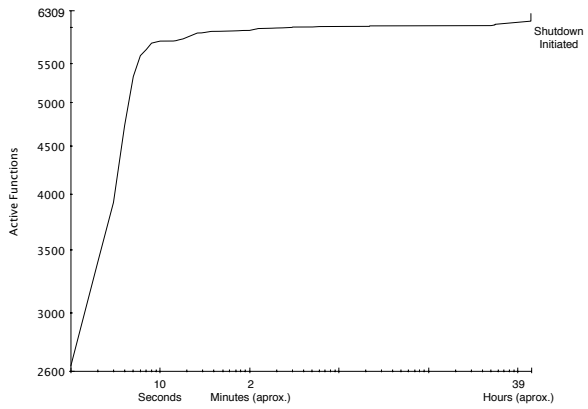

Fig. 1. Reactivation of functions over time (log scale)

To evaluate our design, we configured the protected guest with a common LAMP web server stack (Linux, Apache, MySQL, and PHP). We installed all of these packages from the Ubuntu packaging system. To emulate common usage, we configured and installed a popular blogging web application named WordPress and configured all relevant LAMP modules.

After configuring the web server, we rebooted the system with KIS protection enabled. At boot, KIS disabled 28,828 kernel functions, and 94% of the functions that were eventually reactivated were activated within the first 1 minute after deactivation. We allowed the web server to run for 39 hours. Upon shutdown, 160 functions were activated. A plot of these numbers is illustrated in Figure 1.

Of the 28,828 functions deactivate, only 6,309 were reactivated. 78% of the kernel functions were not needed by this web server workload. 72% of the instruction bytes were not needed by this web server workload. KIS can therefore effectively remove nearly 3/4 of the instructions that could be reused by ROP and JOP attacks.

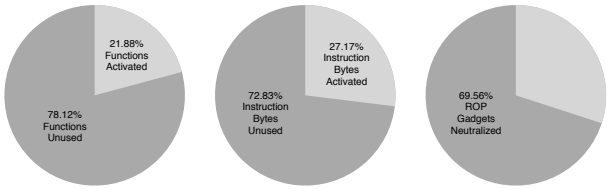

Fig. 2. KIS keeps it simple

To evaluate the effectiveness of KIS against attack, we used an ROP exploitation tool named ROPgadget v4.0.3 [**?**]. ROPgadget found 92 reusable gadgets in the Linux kernel ELF file (vmlinux-2.6.38-8-generic). For our evaluation, with 6,309 kernel functions activated, KIS would prevent 69.56% of the gadgets from executing. Any attack that depends on one of the 64 deactivated gadgets will fail to run on our web server's kernel. Figure 2 illustrates these numbers.

VII. Discussion

The on-demand function activation technique described in Sections IV-B1 and V assumes that all valid calls to a function, set the program counter to the address of the first instruction byte for the function. As a result features such as function nesting, a non-standard C feature found in GCC, are not supported unless the parent function is first activated [**?**]. Similarly, according to the C standard, nonlocal jumps from a callee function are permitted when the jump destination is located in the calling function. If C programming standards are followed, the target function of a nonlocal jump will have already been previously activated. It is possible however, that valid machine code could be created that defies our assumptions and that valid programs would be prevented from executing as intended as a result of KIS.

We have demonstrated that it is possible to reduce the kernel code significantly for one specific well-defined workload. We have demonstrated that for this specific workload and for one known set of ROP exploits, that kernel security is improved. It is possible that some workloads will activate significantly more of the kernel code. It is also possible that other ROP techniques may produce significantly more attack possibilities.

If an attacker understood and detected the on-demand function activation technique described here, he may be able to directly or indirectly cause valid calls to a set of functions that contain code segments necessary for a specific return-oriented attack.

VIII. Summary

We introduced two novel techniques for run-time kernel minimization. We showed that these techniques can be an effective defense against kernel-based ROP attacks. Further, we demonstrated that for a common Linux web server workload, the distributed, unmodified Linux kernel tested was far too large; the workload demanded only 27.17% (by bytes) of the shipped kernel code to run.

Code injection prevention and authentication techniques provide a strong defense against attacks that require foreign code to run in kernel space. However, these techniques are insufficient on their own to defend against attacks that reuse existing kernel code. Combining the former with run-time kernel minimization significantly improves the security of a commodity operating system kernel.

References

[1] D. M. Stanley, Z. Deng, D. Xu, R. Porter, and S. Snyder, "Guest-transparent instruction authentication for self-patching kernels," *Proceedings of Military Communications Conference (MILCOM)*, October 2012.

[2] E. Bryant, J. Early, R. Gopalakrishna, G. Roth, E. Spafford, K. Watson, P. William, and S. Yost, "Poly2 paradigm: a secure network service architecture," in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, dec. 2003, pp. 342 – 351.

[3] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[4] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*, 2nd, Ed. Prentice Hall, 1987.

[5] D. Golub, R. Dean, A. Forin, A. Dean, R. Forin, and R. Rashid, "Unix as an application program," in *In USENIX 1990 Summer Conference*, 1990, pp. 87–95.

[6] R. Campbell, G. Johnston, and V. Russo, "Choices (class hierarchical open interface for custom embedded systems)," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 3, pp. 9–17, Jul. 1987.

[7] C. Pu, H. Massalin, and J. Ioannidis, "The synthesis kernel," *Computing Systems*, vol. 1, pp. 11–32, 1988.

[8] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 1–20.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009.