Containing a Confused Deputy on x86: A Survey of Privilege Escalation Mitigation Techniques

Scott Brookes
Thayer School of Engineering
Dartmouth College
Hanover, NH, USA

Stephen Taylor
Thayer School of Engineering
Dartmouth College
Hanover, NH, USA

***Abstract*—The weak separation between user- and kernel-space in modern operating systems facilitates several forms of privilege escalation. This paper provides a survey of protection techniques, both cutting-edge and time-tested, used to prevent common privilege escalation attacks. The techniques are compared against each other in terms of their effectiveness, their performance impact, the complexity of their implementation, and their impact on diversification techniques such as ASLR. Overall the literature provides a litany of disjoint techniques, each of which trades some performance cost for effectiveness against a particular isolated threat. No single technique was found to effectively mitigate all known and potential attack vectors with reasonable performance cost overhead.**

Keywords*—Protection & Security; Virtualization; Kernel ROP; ret2usr; Kernel Code Implant; rootkits; Operating Systems; Privilege Escalation

I. Introduction

The modern operating system kernel is one of the most basic building blocks of any complex computing or control system. It exists to provide a controlled interface to the hardware and to protect multiple processes and users from each others' actions. In order to accomplish these tasks securely, it must operate with a higher privilege level than user processes, making it an attractive target for attackers. As security research steadily enhances the security of individual processes, the kernel is being attacked more regularly. Despite the recent increase in popularity of attacking the kernel, system designers have long recognized the need for kernel security. MULTICS [1], [2] was one of the first operating systems to take security seriously and laid the groundwork for the most popular kernel security mechanisms still used today. In particular, it defined operating system "rings", designated by processor modes, and memory segmentation and paging structures with flexible read, write, and/or execute permission bits to allow memory partitioning and protection.

Unfortunately, almost all modern operating systems share a common vulnerability: a "weak" separation between kernel- and user-space. While the operating system provides a unique address space for each process in order to isolate processes from one-another, each address space must still allow access to kernel functionality. This is generally accomplished by sharing the address space of the kernel with each process. In contrast to the rare instances of "strong" separation between kernel- and user-space (such as the 4G/4G split Linux patch [3], 32-bit XNU [4], and certain systems using the hardware facilities provided by SPARC V9 hardware [5]), this weak separation protects the kernel from unauthorized access only with the mode of operation of the processor. A process that successfully manages to operate in supervisor mode has carte blanche access to all of the code and data of the kernel.

Often assisted by the weak separation of kernel- and user-space, all of the most popular kernels have been compromised by "rootkits" that give the attacker the highest level of privilege (i.e. "root") [6]–[8]. This survey is specifically interested in privilege escalation attacks that:

- *Hijack the facilities of the kernel* to create a "confused deputy" that is acting on behalf of the attacker [9]. This does not include attacks that are correctly exercising badly designed features of the kernel [10] or attacks that operate outside of the purview of the kernel [11].
- *Persist even without a specific kernel-level bug or design flaw.* Although most rootkits do require some kernel level bug (such as a buffer overflow) to be invoked, attacks that utilize a *specific* bug such as [12]–[14] are beyond the scope of this survey. Additionally, attacks such as [15] that are enabled by a specific kernel design flaw will not be considered. These cases typically have trivial solutions.
- *Elevate local privilege to root* rather than "horizontal" privilege escalation such as [16].
- *Effect x86 Architectures.* The focus of this article is on the x86 architecture because of its wide use in data centers and workstations [17]. However, some techniques specific to ARM will be examined because they do make valuable and interesting contributions to the state of the art.

The privilege escalation attacks that fit these criteria fall into three main categories: kernel code implants [18], kernel-mode return oriented programming (ROP) [19]–[21], and return-to-user (ret2usr) attacks [22].

Kernel code implants are attacks in which the adversary manages to overwrite existing code with (or inject) arbitrary instructions into the kernel space, and then direct the kernel to execute those instructions. Well-known examples of this type of attack include exploitation of classic buffer-overflow vulnerabilities associated with system calls [23]. If an attacker

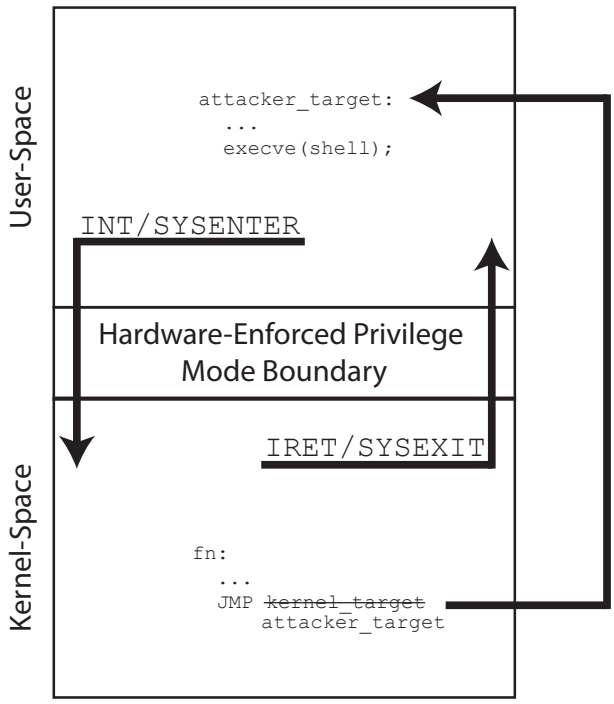

Fig. 1: Return-to-User Privilege Escalation Attack

manages to overflow a buffer on the kernel stack using some malformed arguments to a system call, it is possible to write shell-code onto the stack and overwrite a return-address so as to invoke the shell-code. This attack vector has largely been mitigated by techniques that mark the stack non-executable or provide canary code to detect overflows [24], [25] but it illustrates the core concept.

Kernel return-oriented programming (ROP) attacks defeat the use of a non-executable stack by using a payload, not of code directly on the stack, but of carefully crafted stack-frames that direct computation through a series of gadgets found in normal kernel code [19]–[21]. Research has shown that even small programs are likely to contain the gadgets necessary to generate a ROP Turing machine controlled only by a carefully crafted payload delivered to the stack [26]. All operating systems are large and complex enough to guarantee that the necessary gadgets will be present. As a result, an attacker with the appropriate knowledge can perform arbitrary computation using a ROP payload.

A return-to-user attack is enabled directly by weak kernel- and user-space separation. In this attack, illustrated in Figure 1, a user-controlled target associated with some kernel-code branch is set to an address in the normal user-space code. The compromised branch creates a path of execution that leaves kernel-code, entering user-code, without changing the CPU privilege level from supervisor mode to user mode. This attack results in the execution of user-controlled code with kernel-level privileges. Although hardware extensions such as Intel's SMEP [27] aim to mitigate this threat, these extensions are only slowly being adopted by operating systems and SMEP bypass techniques have already been demonstrated [28], [29].

Unfortunately, mitigation techniques for privilege escalation do not operate in isolation and it is important that they do not undermine other security features. For instance, it is easy to inadvertently inhibit techniques for enhancing security using non-determinism. This general class of technique was initially described by Cohen [30] and Forrest [31]. In the years since these seminal papers, many have explored the idea further. A recent survey of the area was presented in [32]. Address Space Layout Randomization (ASLR) is one of the most widely used applications of this technique. First implemented by the Linux PaX team [33], many other operating systems have implemented some form of ASLR including Mac OS X [34], Windows [35], and others. ASLR loads distinct memory regions including main program code, libraries, and the stack and heap at random locations within a program's virtual address space making it difficult to predict code entry points. More fine-grained techniques for diversifying the memory layout of a process [36], [37] require even more flexibility than traditional ASLR.

In Summary, this paper surveys the primary technologies presented in the literature to mitigate privilege escalation. It provides a comparative analysis based on their effectiveness, performance impact, and implementation complexity. It also specifically considers whether the technologies provide sufficient flexibility to coexist with state of the art address space layout randomization techniques. ASLR is chosen to provide a window to whether the techniques presented "play nicely" with other kernel security efforts because it is has widespread application on real systems and requires flexibility in order to be implemented fully. Section II examines techniques based on hypervisors and virtualization while the remaining techniques are discussed in section III. These techniques are compared and contrasted in section IV. Finally, some proposals are more accurately described as architectures than techniques. These are not directly comparable to the primary methods because they involve a dramatic paradigm shift. These approaches are briefly reviewed in Section V.

II. Mitigation Techniques based on Virtualization

Virtualization has dramatically changed the face of computing, not simply in terms of security and the way individual users interact with computers, but also by enabling cloud computing by allowing virtual machines to be migrated between servers. By adding a layer to the standard software stack, known as a hypervisor [38] or Virtual Machine Monitor (VMM) [39], an abstraction layer is introduced to isolate the operating system kernel from the hardware. In many ways, the hypervisor is to an operating system what an operating system is to a user process - serving to protect virtual machines from each other just as a kernel isolates user processes. The following approaches use virtualization as a means to deliver security guarantees to the kernel.

A. NICKLE

NICKLE [40] provides memory integrity to kernel code and thereby denies the execution of kernel code implants. It uses a VMM to maintain a "shadow" copy of memory that is verified when any kernel-code is loaded. This is achieved by comparing the memory to be loaded against a pre-computed cryptographic hash of the "clean" code distributed by the

manufacturer or developer of the code. At boot time, a known clean copy of the kernel is loaded into the shadow memory and whenever a kernel module is loaded at runtime, it is verified and added to the shadow memory.

With the integrity of the shadow memory guaranteed by off-line a priori cryptographic hashes of trusted code, NICKLE can ensure that no unauthorized kernel code is executed by directing all memory accesses targeting kernel code to retrieve from the shadow memory rather than from regular memory. Although no attempt is made to deny an attacker from modifying or injecting kernel code, kernel-mode execution is contained within trusted memory.

This is achieved transparently to the operating system kernel, allowing for commodity operating systems to be executed with NICKLE with no modification of kernel code. Additionally, NICKLE permits the mixing of kernel code and data within memory pages; this distinguishes NICKLE from many alternative approaches that require code and data to be loaded onto unique pages.

Unfortunately, NICKLE requires the off-line computation of cryptographic hashes for any code that may be executed; this poses a significant logistical issue for maintaining NICKLE on real systems and adds additional vulnerabilities associated with protection and distribution of hash values. NICKLE imposes a "minimal to moderate impact on system performance, relative to that of the respective original VMMs" averaging 1%-5% [40].

B. SecVisor

SecVisor [41] is an alternative virtualization technology leveraging hardware facilities to virtualize physical memory associated with modern processors. By utilizing this additional layer of translation from "guest physical" to "real physical" memory addresses, additional hardware memory protections can be enforced. This capability typically provides additional flexibility in creating memory access security; namely, any combination of read, write, and execute permissions can be allowed or denied on a particular page of memory [42].

SecVisor uses physical memory virtualization to mark only one of kernel- and user-space executable at a time. When a violation of security rules is detected, the protections can be swapped if the CPU has indeed changed privilege level, but are otherwise denied. This defeats ret2usr attacks by preventing unauthorized processor mode switches as shown in Figure 2. Additionally, the same virtualization allows SecVisor to enforce standard W$\oplus$X rules on all kernel code pages that the user has approved. This mitigates the possibility of a kernel code implant by verifying that all executable kernel code is non-writable and has been approved for execution by the user.

The security benefits of SecVisor are packaged in a tiny VMM that provides a small attack surface: only 4092 lines of source code in total. Unfortunately, SecVisor does have several weaknesses. The kernel running on top of SecVisor must guarantee that it does not share code and data on a single page. Additionally, the kernel has to be modified to cooperate with SecVisor by issuing VMCALLs to designate that it is loading or unloading kernel code. Finally, it imposes an overhead as high as 97% due to the additional translation

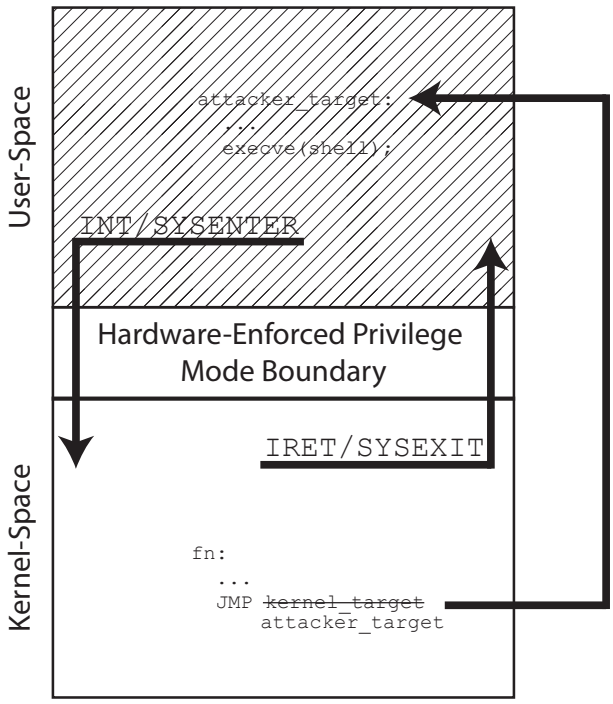

Fig. 2: SecVisor's Protection Against ret2usr Attacks

required by the virtualization of physical memory. For a full discussion of performance overhead costs, the interested reader should consult [41].

It is worth noting that [43] discovered two different bugs in the SecVisor implementation that allowed an attacker to violate rules that SecVisor claimed to enforce. Although these were implementation rather than design issues, and easily remedied, it is clear that even in a small code base security properties are difficult to reason about and correctly enforce.

C. SVA

The Secure Virtual Architecture (SVA) [44] is a set of architecture independent instructions that allow an operating system to interact with hardware. A kernel is ported to use these instructions, similar to porting a kernel to any new hardware architecture. Offline, an SVA compiler produces SVA byte-code from the kernel source code. This compiler has advanced features to provide memory safety and control-flow integrity at compile-time, similar to "safe" programming languages such as Java. The byte-code is distributed to users and executed on top of a virtualized SVA interpreter that performs the final step of translating to native target-dependent machine code.

The effort required to port an operating system to execute on SVA and the large performance cost are balanced by a promise of a substantial increase in security. Guaranteed memory safety and control-flow integrity deny common methods used to initiate ret2usr, kernel ROP, and kernel code implant attacks. An important point is that SVA does not set out to deny these attacks explicitly. Instead, it attempts to deny the vulnerabilities that enable these forms of attack, such as buffer overflows. Unfortunately, the infrastructure needed to support

SVA presents a significant hurdle. In addition to porting a kernel to a new architecture, SVA imposes restrictions on the kernel's memory allocation mechanisms that are likely to require modifications in kernel subsystems such as `kmalloc`. The performance cost is high, measured at approximately 50% on average, but at times reaching a 4-fold reduction.

D. KCoFI

Kernel Control Flow Integrity (KCoFI) [45] leverages the mechanics of the SVA implementation discussed previously, but offers only control flow integrity. Specifically, KCoFI ensures that function calls always enter at the beginning of some function's code, and that all returns from a particular function target the location of a possible call site. In order to prevent user-space applications from imitating the labels that KCoFI uses to validate branches, allowable address transitions are restricted to those within a certain pre-defined "kernel" range of virtual addresses. This limits the capabilities of advanced load-time randomization schemes. KCoFI also provides advanced treatment for the issues that make control flow integrity particularly difficult in the context of operating systems. In particular, it takes special care to handle interrupts, signals, DMA/devices, incomplete branch target information at compile-time, and page faults.

By verifying all branches at run-time, while the processor is in kernel mode, KCoFI manages to deny each of the three primary privilege escalation techniques described in this survey. Unfortunately, as with SVA, there is a large performance cost. Although the average performance impact on a standard application was 13%, worst-case costs up to 3.5-fold were reported. In addition, the method shares the SVA framework and therefore also requires porting the OS to a new "architecture," and pre-compiling the kernel and all of its modules with specialized SVA compilers.

E. SBCFI

State-based control-flow integrity (SBCFI) [46] provides course grained control-flow integrity for the operating system kernel. It sets itself apart from traditional control-flow integrity solutions, such as [47], in two ways. First, it implements monitoring externally from the kernel, in a hypervisor. Additionally, it assumes that attackers will generate persistent control-flow violations, therefore necessitating that kernel state is checked only periodically. Consequently, its introspection techniques allow SBCFI to detect any attack that persistently modifies the kernel's known control-flow graph.

The authors of [46] argue that trading strict security rules for performance by using SBCFI instead of complete CFI is acceptable because SBCFI will still detect most rootkits. In particular, they examined 25 rootkits found "in the wild" on Linux and found that all but one were detected by SBCFI. They suggest that attacker goals such as packet-sniffing or keystroke logging demand persistent rather than transient control-flow changes.

Unfortunately, SBCFI focuses on detection rather than prevention. This, combined with the focus on only persistent control-flow changes, leaves many avenues open to the attacker. SBCFI verifies the state of the kernel by checking a pre-computed hash of the kernel code and checking all function pointers stored in the kernel heap to verify that nothing has been changed. These checks would not detect a process that has achieved escalated privilege via a ret2usr attack or a kernel ROP payload. Overall, SBCFI manages to effectively deny persistent modifications to the kernel control-flow graph with minimal performance costs of less than 1% on average. However, it fails to address the general threat associated with privilege escalation.

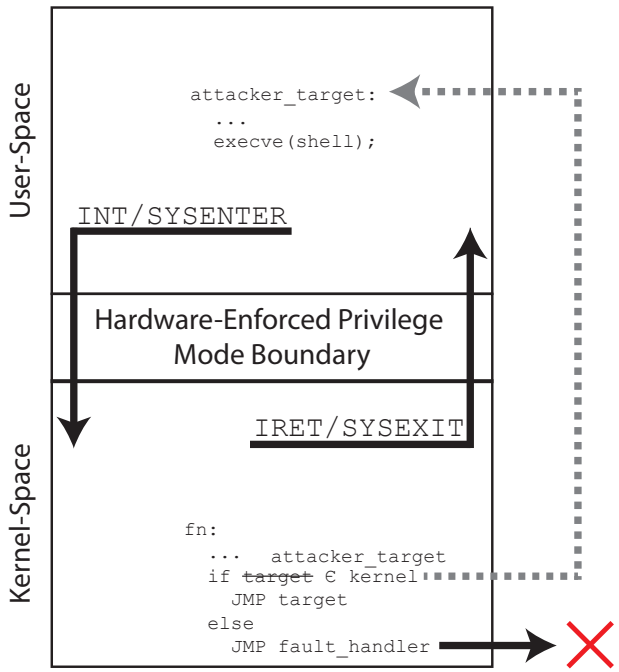

Fig. 3: kGuard's Protection Against ret2usr Attacks

III. Other Techniques

A. kGuard

kGuard [22] aims to deny ret2usr attacks by inserting guards on the kernel's control-flow at compile time as shown in Figure 3. On the x86 platform, the `call`, `jmp`, and `ret` instructions all redirect control-flow and therefore are vulnerable to being hijacked in order to redirect kernel execution into user-controlled code. kGuard places an inline check before any of these instructions. The checks are provided in two different forms depending on whether the target address is stored in a register or in memory. The checks simply verify that the branch target lies within kernel-space. Unfortunately, if an attacker controls the target of two branches, he can direct the first to jump directly to the second branch, bypassing the kGuard check completely. Since the second branch is in kernel-space, the check on the first branch would allow the control transfer. To avoid this attack, kGuard includes a compile-time code diversification mechanism that makes it difficult for the attacker to locate the address of the second branch.

One of the most significant advantages of kGuard is that, as a purely compile-time technique, it is portable to any operating system on any target hardware. It does not require any special hardware or impose many restrictions on the implementation of kernel features. Additionally, its average performance cost

is low at approximately 1%, making it deployable on existing systems. Unfortunately, kGuard does suffer from a variety of weaknesses. Although its simplicity lends itself to easy deployment, it is unable to protect against kernel-code implants or kernel-ROP. Although these are outside of the scope of kGuard, a kernel code implant could be used to create a ret2usr attack by implanting an unguarded jump into a user-space region. Therefore, another technique must be used in combination with kGuard to deny the possibility of a ret2usr scenario. This quickly increases in complexity and performance cost as multiple techniques need to be deployed on the same system. Additionally, kGuard's inline checks verify that the target of a control-flow transfer lies in kernel-space only by checking that it falls within a predefined range. This limits the capacity for deploying advanced code randomization during the loading of the kernel.

B. Return-less Kernels

Recall that Kernel ROP attacks requires "return" instructions in order to move from one gadget to another. In [48] the author utilizes "return indirection," introducing additional jumps at compile-time to disrupt this mechanism and defeat kernel ROP attacks. This approach uses a pre-computed table of all legal return addresses. Rather than pulling a return address from the stack and jumping to it at the end of a function, this method reads an address from the specified index in the return address table. If this table is trusted, the attacker could only modify which legal return address is used. It is assumed that most gadgets begin in a location other than a legal return address, and as a result this technique defeats the possibility of an attacker to craft a malicious payload.

In addition to introducing return indirection, [48] introduces compiler modifications to avoid instructions with an embedded "return" opcode. On an architecture such as x86, variable length instructions make it possible to read different instructions if the instruction pointer is offset some distance into an opcode. Without taking care at compile-time to avoid these scenarios, an attacker could still create gadgets by indexing the instruction pointer at unintended positions in the middle of the intended instruction.

The idea of using a return-less kernel is a clearly beneficial. It effectively mitigates a very particular risk with reasonable overhead, assessed at approximately 6%. Unfortunately, it does require modification to the kernel source. Functionality provided by compiling a higher-level language, such as C, does not need to be modified, but any functionality defined in assembly language must be manually modified to follow return-less principles. Since the kernel interfaces with hardware directly, there is a non-trivial level of assembly code included in most kernel implementations.

C. PaX

One of the first kernel-hardening efforts was implemented on Linux by the PaX team [49] circa 2000. UDEREF [50] utilizes segmentation to create a stricter separation between kernel- and user-space (denying ret2usr), while the PAGEXEC and Restricted `mprotect()` features essentially generate and enforce typical W$\oplus$X security rules on kernel code and data to mitigate kernel code implants.

PaX is valuable as a case study in hardening kernels. Unfortunately, it is less valuable as a mechanism for protecting modern kernels on today's hardware. Its protection mechanisms were based on Linux-specific software mechanisms (such as `mprotect()`) and x86-32-specific hardware features (such as segmentation). Additionally, the performance cost was significant, according to [22]. PaX-reported data about performance cost was available at the time of writing.

D. Sprobes and TZ-RKP

Sprobes [51] and TZ-RKP [52] both utilize the ARM TrustZone [53] hardware facilities included in modern ARM processors. TrustZone is a hardware-protected context that can run tangentially to the regular operation of the processor. The hardware disables the normal processor context from accessing anything within the "secure world" created by TrustZone and transitions between the regular context and TrustZone's secure context are limited by hardware to a small well-specified interface.

Sprobes [51] utilizes TrustZone by installing an introspection handler in the secure world and installing, at load- or run-time, special instructions that invoke the secure world at predetermined points in the execution of the kernel. When one of these probes is executed, control transfers to the secure world in which kernel state can be interrogated, control flow or memory contents verified, or any other number of actions can be taken. Furthermore, restrictions are placed on the normal world's ability to manipulate the virtual memory settings of the processor. The requirement that these systems be updated by the secure world guarantees that a kernel cannot manipulate virtual memory in order to bypass the probes.

TrustZone-based Real-time Kernel Protection (TZ-RKP) [52] is a similar approach that forces vital control operations involving the virtual memory layer to be routed through the secure world. TZ-RKP forgoes the probes provided by Sprobes, but takes a more extreme approach by limiting the kernel's control over important system state such as virtual memory. TZ-RKP forces all attempts to control virtual memory and other hardware resources through the secure world, providing a mechanism to verify any changes to the system state. With a controlled and static system state, it is easier to make claims about what an attacker may do to manipulate the kernel state.

Both [52] and [51] are built on the TrustZone architecture. The hardware underlying their implementation allows each to be implemented with a reasonable performance cost (typically 10%). TrustZone is also attractive because it manages to avoid the "turtles all the way down" problem in which software layer x is protected by introducing software layer $x-1$, which simply becomes the new target for attackers and instantiates the same problem again. Traditional virtualization can be criticized for this problem, but TrustZone holds itself off to the side of layer x rather than existing underneath it.

Unfortunately, TrustZone is an ARM-specific technology. Although ARM is used extensively in mobile and embedded applications, the x86 architecture continues to dominate desktop and server applications. Although these techniques are interesting, their utility is limited by a reliance on specialized hardware.

TABLE I: Summary of Examined Attack Mitigation Methods

Project	Kernel Code Implant	Kernel ROP	ret2usr	Typical Reported Performance Cost	Maximum Reported Performance Cost
NICKLE [40]	✓	✗	✗	1-5%	19.03%
KCoFI [45]	✓	✓	✓	13%	3.5×
SVA [44]	✓	✓	✓	50%	4×
SecVisor [41]	✓	✗	✓	20%	97%
SBCFI [46]	✓	✗	✗	<1%	13%
kGuard [22]	✗	✗	✓	1%	23.5%
PaX [49], [50]	✓	✗	✓	No Data	No Data
Return-less Kernel [48]	✗	✓	✗	6%	17.32%
Sprobes [51]	✓	✓	✓	10%	10%
TZ-RKP [52]	✓	✗	✓	3%	7.65%

TABLE II: Further Characteristics of Examined Methods

Project	x86-64 Compatible	(M)emory and/or (C]ontrol [F)low Integrity	Code-Diversity Compatible	LoC
NICKLE [40]	✓	M	✓	932
KCoFI [45]	✓	CF	✗	5579
SVA [44]	✓	M & CF	✓	No Data
SecVisor [41]	✓	M	✓	4092
SBCFI [46]	✓	CF	✓	No Data
kGuard [22]	✓	CF	✗	1000
PaX [49], [50]	✗	M & CF	✗	No Data
Return-less Kernel [48]	✓	CF	✓	2100
Sprobes [51]	✗	M & CF	✓	No Data
TZ-RKP [52]	✗	M	✓	No Data

IV. Comparison

Table I summarizes and compares the techniques discussed in the previous sections on the basis of their ability to mitigate privilege escalations and their expected cost:

- *Kernel Code Implant/Kernel ROP/ret2usr*: Does this technique mitigate the risk of privilege escalation associated with these particular attack vectors?
- *Typical/Maximum Performance Cost*: What is the typical and worst-case reported performance costs?

The performance costs listed represent only the maximum performance cost and an estimated average used only to illustrate differences between the techniques. In some cases these come from micro-benchmarks corresponding to small code segments, in other cases they come from macro-benchmarks corresponding to full applications. For the estimated average, they are often a mix of these tests. Each of the techniques offers thorough performance cost analyses that could not be summarized in a simple table. Interested readers should consult the original paper for each technique for a more complete treatment.

Table II compares the techniques on the basis of general observations regarding their operation:

- *x86-64 compatible*: Most desktop and server-class systems use the 64-bit x86 architecture. Is the technique viable with the hardware provided by the x86-64 hardware?
- *Memory and/or Control Flow Integrity*: Which is the primary mechanism by which the tool delivers its security guarantees?
- *Code-Diversity Compatible*: Is the technique sufficiently flexible to allow for advanced fine-grained address space layout randomization techniques?
- *Code Size*: How many lines of code (LoC), as a measure of the attack surface presented, are used in the implementation of the technique as presented?

It is clear from Table I that while KCoFI and SVA offer the most protection against the three different techniques associated with privilege escalation, they also come with dramatically more performance overhead than the other methods. This conforms to expectations in that the more thorough the security measure, the higher its performance impact. Sprobes and TZ-RKP appear exceptional as they enjoy the lowest performance costs and strong security claims. Unfortunately, each utilizes the ARM TrustZone architecture and consequently are unavailable on the Intel x86 architecture. Additionally, vulnerabilities have already been discovered in some TrustZone hardware implementations [54].

V. Paradigm-Shift Techniques

The techniques compared in Tables I and II each provide a modification to some part of the conventional kernel design, implementation, or build process that mitigates a particular threat. There are a few approaches, however, that attempt to offer similar security benefits by redefining the security paradigm rather than simply patching the status quo best practices. This radical departure from the current state of the art means that they cannot easily be compared to the previously described techniques. In all cases, it also means that they have not yet been widely accepted.

A. Microkernels

The idea of a microkernel departs from the standard "monolithic" kernel architecture by emphasizing a small codebase for the operating system kernel. There have been several examples of microkernels presented in the literature such as Mach [55], Minix [56], L4 [57], QNX [58], Bear [59], and many others.

All microkernels aim to minimize the source code in order to decrease the likelihood of vulnerabilities [60]. Additionally, a small code base allows for the possibility of using formal analysis and formal verification techniques [61], [62]. In order to keep the microkernel small, core functionality such as device drivers are migrated into user level processes. Additionally, many microkernels use message-passing for all communication between two processes or a process and the kernel. This provides a more easily verified and secured narrow interface between components.

By exporting core functionality, such as device drivers, into user-space microkernels struggle to offer the same levels of performance as monolithic kernels. Consequently, they have yet to replace monolithic kernels in common applications on commodity hardware.

B. ExoKernel

The ExoKernel [63] suggests redefining the nature of the kernel entirely. Rather than providing abstractions that the application developer can use to access hardware, the ExoKernel provides only the thinnest possible layer to manage the multiplexing of hardware resources. Therefore, the ExoKernel circumvents tasks normally reserved for the kernel such as buffering network communications, interrupt or exception handling, virtual memory management, and other normal kernel functions. Instead, each individual application must define its own abstractions to handle these tasks.

Although likely to offer more security for a system overall, the ExoKernel appears significantly complicate application development. Many of the tasks that a secure kernel can provide to protect all processes, such as virtual memory management, become the responsibility of the application developer. This is likely to make individual applications less secure since application programmers may lack the technical sophistication to interact directly with hardware, interrupts, atomicity, and concurrency. These central parts of the operating system exist to provide applications with well-defined interfaces to this complex functionality. The ExoKernel eliminates those interfaces by design.

C. Unikernels

Unikernels trade flexibility for security and performance by running a single process within a single address space [64]. Eliminating the requirement to support multiple processes and/or multiple users simplifies the code base required to implement a unikernel and reduces the overhead required to complete a single unit of useful work. Several examples have been deployed alongside virtualization technologies in cloud applications [65]–[67]. Despite their proven usefulness for providing fast, highly focused applications, unikernels don't, in isolation, provide protection from most of the attack vectors discussed in this paper. Additionally, in order to support the multiple-user multiple-job paradigm that conventional applications require to operate effectively, they require a hypervisor for scheduling and other process-management type tasks. In a sense, this is simply asking the hypervisor to act as an operating system and the same issues with conventional operating system design will simply move one layer deeper in the software stack.

VI. Conclusion

Each of the techniques examined in this survey makes valuable contributions to the security of modern operating systems. Those that offer the most comprehensive security suffer from high performance costs or specialty hardware requirements. On the other hand, many mitigate a specific, focused risk to kernel security while suffering only a small performance cost. Unfortunately, there is no single solution that offers both acceptable performance and comprehensive security coverage on the popular x86 platform. The impact of combining the techniques to improve coverage is not well understood in terms of complexity, performance, or security. This survey has also examined techniques that, rather than presenting incremental improvements on the status quo, attempt to dramatically redefine the notion of an operating system. These techniques also suffer from nontrivial performance costs in addition to the logistical challenges associated with a paradigm shift.

Overall, the kernel developer has a wide variety of techniques to choose from, but must balance individual strengths in privilege escalation prevention with the associated penalties in performance and complexity. The authors believe that future work aimed at mitigating privilege escalation will continue to have performance issues without some change in the underlying hardware or kernel design paradigms. Modern commodity operating systems are so highly developed that there is unlikely to be some technique hiding in a dark corner that will not decreasing perforance by requiring extra work.

Notice

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. This material is based on research sponsored by DARPA under agreement number: FA8750-11-2-0257.

References

[1] R. C. Daley and J. B. Dennis, "Virtual memory, processes, and sharing in multics," *Communications of the ACM*, vol. 11, no. 5, pp. 306–312, 1968.

[2] F. J. Corbató and V. A. Vyssotsky, "Introduction and overview of the multics system," in *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, ser. AFIPS '65 (Fall, part I). New York, NY, USA: ACM, 1965, pp. 185–196. [Online]. Available: http://doi.acm.org/10.1145/1463891.1463912

[3] I. Molnar, "4G/4G split on x86, 64 GB RAM (and more) support," July 2003. [Online]. Available: https://lwn.net/Articles/39283/

[4] D. Keuper, "Xnu: a security evaluation," December 2012. [Online]. Available: http://essay.utwente.nl/62852/

[5] R. McDougall and J. Mauro, *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Pearson Education, 2006.

[6] K. Way, "Lastore-daemon in deepin 15 results in privilege escalation," February 2016. [Online]. Available: https://www.exploit-db.com/exploits/39433/

[7] J.-J. Khalife, "MS15-010/CVE-2015-0057 win32k Local Privilege Escalation," December 2015. [Online]. Available: https://www.exploit-db.com/exploits/39035/

[8] rebel, "issetugid() + rsh + libmalloc osx local root," July 2015. [Online]. Available: https://www.exploit-db.com/exploits/38371/

[9] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, Oct. 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=54289.871709

[10] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Information Security*, ser. Lecture Notes in Computer Science, M. Burmester, G. Tsudik, S. Magliveras, and I. Ili, Eds. Springer Berlin Heidelberg, 2011, vol. 6531, pp. 346–360. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18178-8_30

[11] Kdm, "NTIllusion: A portable Win32 userland rootkit," 62. [Online]. Available: http://phrack.org/issues/62/12.html

[12] "CVE-2016-0728," January 2016. [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2016-0728

[13] "CVE-2013-2094," May 2013. [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2013-2094

[14] metasploit, "Chkroot local privilege escalation," November 2015. [Online]. Available: https://www.exploit-db.com/exploits/38775/

[15] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 957–972. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671286

[16] "CWE-639: Authorization Bypass Through User-Controlled Key," September 2008. [Online]. Available: https://cwe.mitre.org/data/definitions/639.html

[17] T. P. Morgan, "x86 servers dominate the datacenter - for now," June 2015. [Online]. Available: http://www.nextplatform.com/2015/06/04/x86-servers-dominate-the-datacenter-for-now/

[18] A. Lineberry, "Malicious code injection via/dev/mem," 2009.

[19] E. Buchanan, R. Roemer, S. Savage, and H. Shacham, "Return-Oriented Programming: Exploitation without Code Injection," 2008. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

[20] 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. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313

[21] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 383–398. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855792

[22] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection Against Return-to-user Attacks," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 39–39. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362832

[23] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, November 1996. [Online]. Available: http://www.phrack.com/issues.html?issue=49&id=14

[24] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267549.1267554

[25] C. Cowan, "Non-executable stack," 1997.

[26] "Microgadgets: Size does matter in turing-complete return-oriented programming," in *Presented as part of the 6th USENIX Workshop on Offensive Technologies*. Berkeley, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/woot12/workshop-program/presentation/Homescu

[27] S. Fischer, "Supervisor mode execution protection," 2011, nSA Trusted Computing Conference and Exposition. [Online]. Available: https://www.ncsi.com/nsatc11/presentations/wednesday/emerging_technologies/fischer.pdf

[28] D. Rosenburg, "SMEP: What is it, and How to Beat it on Linux." June 2011. [Online]. Available: http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/

[29] keegan, "Attacking Hardened Linux Systems with Kernel JIT Spraying," June 2011. [Online]. Available: http://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html

[30] F. B. Cohen, "Operating system protection through program evolution," *Computers & Security*, vol. 12, no. 6, pp. 565–584, 1993.

[31] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, ser. HOTOS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 67–. [Online]. Available: http://dl.acm.org/citation.cfm?id=822075.822408

[32] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *Security and Privacy (SP), 2014 IEEE Symposium on*, May 2014, pp. 276–291.

[33] "Address Space Layout Randomization," PaX Team, Tech. Rep., 2001. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[34] "OS X Mavericks Core Technologies Overview," Apple, Tech. Rep., October 2013. [Online]. Available: http://www.apple.com/media/us/osx/2013/docs/OSX_Mavericks_Core_Technology_Overview.pdf

[35] O. Whitehouse, "An Analysis of Address Space Layout Randomization on Windows Vista," Symantec, Tech. Rep., 2007.

[36] M. Kanter and S. Taylor, "Attack Mitigation through Diversity," in *Military Communications Conference, MILCOM 2013 - 2013 IEEE*, Nov 2013, pp. 1410–1415.

[37] ——, "Diversity in Cloud Systems Through Runtime and Compile-time Relocation," in *Proceedings of the 13th IEEE Conference on Technologies for Homeland Security*, 2013, pp. 396–402.

[38] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/945445.945462

[39] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 9, pp. 34–45, Sep. 1974. [Online]. Available: http://dx.doi.org/10.1109/MC.1974.6323581

[40] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87403-4_1

[41] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 335–350, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294294

[42] *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C*, Intel, June 2014.

[43] J. Franklin, A. Seshadri, N. Qu, S. Chaki, and A. Datta, "Attacking, repairing, and verifying secvisor: A retrospective on the security of a hypervisor," Carnegie Mellon University, Tech. Rep., 2008.

[44] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 351–366. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294295

[45] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *Security and Privacy (SP), 2014 IEEE Symposium on*, May 2014, pp. 292–307.

[46] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 103–115. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315260

[47] U. Erlingsson and F. B. Schneider, "Sasi enforcement of security policies: A retrospective," in *Proceedings of the 1999 Workshop on New Security Paradigms*, ser. NSPW '99. New York, NY, USA: ACM, 2000, pp. 87–95. [Online]. Available: http://doi.acm.org/10.1145/335169.335201

[48] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 195–208. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755934

[49] PaX, "Homepage of the PaX Team," 2013. [Online]. Available: http://pax.grsecurity.net

[50] B. Spengler, "uderef," 2007. [Online]. Available: https://grsecurity.net/~spender/uderef.txt

[51] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," *arXiv preprint arXiv:1410.7747*, 2014.

[52] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection

from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 90–102. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660350

[53] "Building a Secure System using TrustZone Technology," ARM, Tech. Rep. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[54] D. Rosenberg, "Qsee trustzone kernel integer over flow vulnerability," 2014.

[55] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for unix development," 1986, pp. 93–112.

[56] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 80–89, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151374.1151391

[57] D. Potts, S. Winwood, and G. Heiser, "Design and implementation of the l4 microkernel for alpha multiprocessors," 2002.

[58] D. Hildebrand, "An architectural overview of qnx," in *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*. Berkeley, CA, USA: USENIX Association, 1992, pp. 113–126. [Online]. Available: http://dl.acm.org/citation.cfm?id=646405.759105

[59] C. Nichols, M. Kanter, and S. Taylor, "Bear – a resilient kernel for tactical missions," in *Military Communications Conference, MILCOM 2013 - 2013 IEEE*, Nov 2013, pp. 1416–1421.

[60] R. K. Pandey and V. Tiwari, "Article: Reliability issues in open source software," *International Journal of Computer Applications*, vol. 34, no. 1, pp. 34–38, November 2011, full text available.

[61] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Formal verification of a microkernel used in dependable software systems," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, B. Buth, G. Rabe, and T. Seyfarth, Eds. Springer Berlin Heidelberg, 2009, vol. 5775, pp. 187–200. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04468-7_16

[62] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560537

[63] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 251–266. [Online]. Available: http://doi.acm.org/10.1145/224056.224076

[64] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Queue*, vol. 11, no. 11, p. 30, 2013.

[65] A. Bratterud, A.-A. Walla, P. E. Engelstad, K. Begnum *et al.*, "Includeos: A minimal, resource efficient unikernel for cloud services," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 250–257.

[66] A. Kivity, D. Laor, G. Costa, P. Enberg, N. HarEl, D. Marti, and V. Zolotarov, "Osvoptimizing the operating system for virtual machines," in *2014 usenix annual technical conference (usenix atc 14)*, 2014, pp. 61–72.

[67] J. Martins, M. Ahmed, C. Raiciu, and F. Huici, "Enabling fast, dynamic network processing with clickos," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 67–72.