VM-based Security Overkill: A Lament for Applied Systems Security Research

[Position Paper, pre-proceedings version]

Sergey Bratus Michael E. Locasto Ashwin Ramaswamy

Sean W. Smith

ABSTRACT

Virtualization has seen a rebirth for a wide variety of uses; in our field, systems security researchers routinely use it as a standard tool for providing isolation and introspection. The extent that researchers use virtual machines has reached, if not a fever pitch, then a level of orthodoxy which makes it difficult for the collective wisdom to consider alternative approaches to protecting computation.

We suggest that, in many scenarios, virtual machines do not provide a suitable tool or appropriate security properties. We analyze the use of virtual machines in the systems security space, we highlight other work that questions the current (ab)uses of virtualization, and we present a case study of an alternative design for protecting privileged computation against malicious computation (i.e., a rootkit). The takeaway message of this paper is that "self-protection" mechanisms still represent an interesting and viable path of research. At some point, hypervisors (or whatever the lowest layer of software, firmware, or programmable hardware is) must rely on detection and protection mechanisms embedded within themselves.

> ...like most existing operating systems, Xen hasn't been designed to withstand attacks, in fact it has few if any self-protection features...what it was designed for is to provide a certain amount of isolation between different domU's...the hypervisor self-protection problem hasn't been solved; the assumption from the [Qubes] architecture document being this will be done in the future (tm). Like I mentioned in my comments, this will require some real work...

–Brad Spengler, "X11 -> Root? (Qubes square rooted)"[1]

Categories and Subject Descriptors

H.1.1 [**Models and Principles**]: Systems and Information Theory—*Value of Information*

[1] `http://lists.immunitysec.com/pipermail/dailydave/2010-August/006171.html`

NSPW 21-23 September 2010 Concord, MA USA

General Terms

Security, Measurement

Keywords

virtualization, isolation, VM

1. INTRODUCTION

Virtualization is clearly an enabling technology: providing execution containers can make computing cheaper, more mobile, easier to back up, share, and archive entire OS environments. Many variations on the theme of pure virtualization exist, from open-source CPU emulators like Bochs and QEMU through container-based systems like User-mode Linux, OpenVZ, and Zap [32] to more commercial offerings like VMWare and VirtualPC (and their open-source cousins like VirtualBox). Virtual machine management infrastructure has — notably and recently — increased the practicality of large scale, commodity distributed computing (i.e., the "cloud").

In addition to arguments involving reduced management, administration, and hardware costs, virtualization technology routinely sees service in a security context, primarily driven by the assumption that virtualization provides the best approach (for some value of "best", whether this means "most practical"; better than `jail`, `chroot`, Janus, or systrace; or something else) to providing isolation between execution containers. At an increased cost (typically), a virtual environment can provide inspection capabilities in addition to basic isolation. This kind of isolation and trapping of senstive operations presents a temptation too sweet for system security researchers to resist.

While we agree that virtualization can be quite useful in many scenarios, we observe that its level of use has approached orthodoxy in terms of the appropriate technology for composing security systems involving isolation, inspection, introspection, or behavioral analysis of execution. In short, as a field, the community seems to have found the perfect implementation form of a practical reference monitor.

1.1 Contribution

This paper attempts to question that orthodoxy. Are VM-based solutions scalable or even economically feasible to enterprise and SCADA network with respect to management and administration overhead they require? Are non-VM approaches still a viable and practical means of achieving isolation, inspection, and other forms of program behavior analysis? Specifically, in situations where sliding another security-enforcing layer such as a hypervisor/VMM might prove too costly for the platform, software might inevitabily

fall back to examining itself. We suggest that (1) such a fallback is inevitable for certain scenarios and so (2) it has to be done with appropriate engineering principles and care to make it least ad hoc as possible.

1.2 Motivation: Detecting Malicious Computation With Little Performance Impact

Our motivation to examine the possibility of non-VM approaches to software supervision, introspection, mediation, and isolation began with a consideration of how to implement anti-rootkit detection capabilities on a range of low-power and low-resource embedded platforms (e.g., mobile phones, 802.11 access points, SCADA controllers). Such devices typically do not have the resources to run an entire VM infrastructure. Furthermore, as a practical matter, using currently available commercial or F/OSS VM technology entails simultaneously running and maintaining two operating systems, with all their attendant services, libraries, and software packages, the locking down and constant patching of which presents a dramtically increased administrative burden compared with our design (a relatively simple, self-contained kernel module that has but one task: monitoring a set of hooks, which we examine in more detail in Section 3).

Rootkit programming Historically, rootkits and exploits tended to be mistakenly associated with malicious, foreign *code* introduced into the system; consequently, anti-rootkit efforts focused on detecting a *foreign body of code* one way or another. This mistaken assumption has been challenged by the original hacker research publications [13, 54, 30, 15] of flexible exploit programming techniques that involved no foreign code, and has hopefully been defeated for good by [46, 11], which made it clear to the academic community that they were dealing with a *Turing-complete programming and execution model* rather than an ad-hoc collection of obscure hacks.

Hund et al. [21] have further generalized and applied this model to rootkit programming. They stress that the essence of a rootkit was not *malicious code* but rather *malicious computation* effected either exclusively or largely with the unanticipated use of existing, trusted code integral to the exploited system.

We note that the art of co-opting and re-using existing (and therefore inherently more stable) code has been an established pattern of rootkit programming since its early days.[2] In fact, we believe that it would fair to speak of *Rootkit Design Patterns*, such as "context-based hot patching", "multi-layer interception", and others [34, 50, 33], all dedicated to *runtime manipulation of both live code and live data while lowering the probability of a crash*.

Accordingly, in our case study (Section 3) of rootkit detection, we focus on computation rather than code. In particular, we focus on *detecting violations of an observed, known invariant of the correct Linux kernel operation*.

1.3 Philosophy on Self-Monitoring and Same-layer Attacks

Essentially, our case study proposes an alternative to VM-based monitoring that uses a *ring 0* technique for catching a particular class of hostile behaviors by *ring 0* malcode. Any such technique has an intrinsic weakness: the attacker who attains the capability to arbitrarily modify kernel memory can disable or interfere with it. The VMM/hypervisor architecture has long been accepted as the principled cure for this weakness. We thus face a philosophical question: should we abandon "ring0 vs ring0" techniques entirely, and consider only those that work under the guaranteed protection of the hypervisor? Have such techniques nothing to teach us?

We do not believe so, for two reasons. First, in practice, solutions to software security problems rarely have the luxury of restarting from scratch even when a better design is available and well–understood. In particular, whereas a principled new design would make undesirable behaviors impossible, a practical security solution is constrained to instrumenting existing systems to detect and counteract such behaviors.

A classic example is provided by network stack vulnerabilities: being "immersed" into the protected network, a NIDS stack can be just as susceptible to packet-based attacks as the target systems [20]. Nevertheless, the community does not reject NIDS as a concept on this basis. Likewise, host-based firewalls, (for example, so-called "personal firewalls"), despite their history of vulnerabilities, continue to be adopted.

Second, the study of instrumentation for certain classes of undesirable events, even when undertaken in the presence of theoretically cleaner designs that would obviate the need for such instrumentation, often shows the way toward more efficient and economical designs. Practical OS security engineering offers an an impressive gallery of ideas that started as "hacks" (OpenWall, PaX, LIDS, BSD Jails and Linux VServer) and ended up informing industrial solutions like ExecShield, GrSecurity hardening patches, LSM, SELinux, and AppArmour.

Finally, although one could argue that the components necessary to support VMMs now exist in commodity hardware, and are thus readily available, using these capabilities entails warping legacy systems around them, *i.e.*, either explicitly rewriting kernels (paravirtualization), or writing device emulation layers: a heavier proposition than instrumentation-based self-analysis. We next consider some additional shortcomings of the VMM-based monitoring philosophy.

2. WHAT'S WRONG WITH VIRTUALIZATION?

As organizations increase their adoption of virtualization environments, and with the current industry focus on information security, it is natural to wonder just how a virtualization framework might pull double duty by improving a security posture as well as easing management burden and infrastructure costs. Of course, merely running a virtualized environment does not automatically entail a guarantee of increased security. As we discuss below, even the basic isolation properties of a VM framework are questionable; it remains entirely unclear whether a VMM can get the job of isolation (a job that OS designers have grappled with for years) correct, especially in the absence of hardware primitives for this purpose.[3] We also agree with the sentiment expressed in recent work [5, 10, 23, 40, 16] (some of it our own) that current VM implementations actually use a flawed event trapping framework that fails to capture events of natural and significant interest to security policies.

2.1 "Perfect" Designs

The history of military thought is replete with designs of "overkill" weapons (e.g., Figure 1,2, and 3) meant to be overwhelmingly powerful or precise, but in reality fall far short of their presumed promise, either in the prototype tests or (much to the wielders' disappointment) on the actual battlefield.

Such designs tend to be founded on a crisply formulated, solid, well-established principle or conventional wisdom – right up until

[2] See, e.g., THC's "(nearly) Complete Linux Loadable Kernel Modules" rootkit HOWTO, 1999.

[3] `http://kerneltrap.org/OpenBSD/Virtualization_Security`

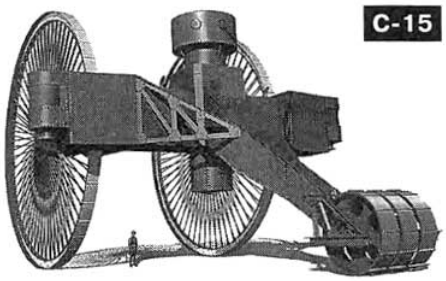

Figure 1: *"Tsar Tank"*, Russia. The nearly 30 feet tall, 40 feet wide prototype carrying 3 cannons and other weaponry was abandoned after unsuccessful 1915 field tests.

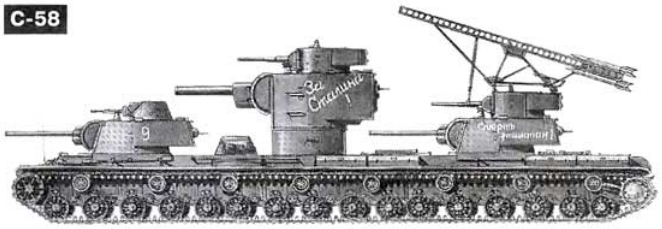

Figure 2: *Superheavy WWII tank design*. Comparable projects included 1000 metric tons, 15,000 horse power vehicles.

reality presents its bill, showing that the accepted theory got to be so neat by ignoring some crucial practical consideration.

We believe that such a situation is forming in academic research aimed at defeating rootkits – due to a popular misinterpretation of the initial and well-deserved success of VM-based research prototypes.

A series of influential papers [14, 25, 17, 4, 18] (among others) demonstrated the power of running the protected software – including monolithic OS kernels such as Linux and Windows – within a virtual machine, which was capable of intercepting and completely mediating all accesses to all relevant OS objects and data structures.

The underlying host system in its entirety essentially played the role of a *trusted reference monitor* for the guest system. In practice, the host OS is augmented with a reference monitor component loaded with the "protection map" of the guest's internals, derived through either static or dynamic analysis of the latter, or both.

Enforcing a desired security invariant through VM-based complete mediation of a system that can, at any point, be transparently stopped and examined for its violations, is a powerful technique. It appears, however, to have become a "gold standard" of invariant/policy enforcement research, with nothing less being considered as worthy of researchers' or readers' time. This view, in our opinion, ignores the following several important practical circumstances we relate below.

2.2 Five Examples of Emergeant VM Complexity

The following five points all possess an underlying theme: that of burgeoning complexity based on forces such as the need or desire to add features or create communications channels. The first two points deal with the forces driving the addition of features and

Figure 3: *"Tsar Cannon", 1586*. At 35-inch caliber and nearly 40 metric tons, it is reportedly the largest howitzer ever built. It never shot the 2-ton cannon ball in the picture; there is no record of it ever being used in battle.

emergence of complexity in the "underneath" layers of a VM-based architecture. The latter two examples discuss the pressure to create complexity within and between execution containers, especially as it relates to information flow control. The middle example finishes the set by dealing with the forces driving the creation of complexity at the interface or boundary between a VM container implementation and the execution guests within this container.

This point cannot be overstated: central to this paper is the assertion that despite the benefits of VM-based approaches, they have very real challenges derived from the forces driving an increase in their complexity, *but these challenges are routinely ignored* when considering VM-based approaches in the abstract as the principled cure for a security problem.

1. Reliance on virtual machines does not take into account the need for remote management of protected platforms.

Many kinds of systems (most prominently embedded systems but also "headless" compute cluster systems) require remote management. Embedded systems based on commodity software need protection as much or more that desktops (e.g., [2, 7]).

In fact, as Dan Geer convincingly argued in [19], the design choice between equiping an embedded system with a remote management interface or releasing it without one is an *evolutionary* choice – whereas remote management interfaces increase the attack surface and risk remote exploitation, the opposite choice often mean loss[4] of an entire released generation of devices to a change in their deployment environments or to a new un-anticipated attack. Clearly, leaving the VM environment without a remote management interface is unlikely to be a popular option.

With the importance of remote management in mind, VM-based rookit protection of a guest OS presents considerable challenges that, in our opinion, significantly reduce its appeal. Namely, *if the host OS is to be remotely managed, where should the management network stack, including the management application itself reside?*

[4] At least, for their control components, which will need to be physically replaced, likely incurring prohibitive labor costs, possibly making the entire product a financial loss.

In practice, remote administration applications are built on top of complex environments that include not only a fully functional TCP/IP stack, a web server, but also a CGI scripting platform, which are notoriously hard to secure due to lack of a well-defined security model.

Interfaces based on command lines and SSH are somewhat better, but still require a full TCP/IP stack to function, and likely also depend on implementations of SSL, DNS, and other standard Internet host functionality to maintain its global connectivity – all not only theoretically susceptible to attack, but quite actively exploited in the past.

We can, of course, imagine a highly hardened dedicated control network stack that mitigates most of these issues, but so far economic pressures have unceasingly driven vendors to least-effort commodity software for actual product development and deployment, and this trend is unlikely to change. Moreover, a developer willing to invest all the necessary effort into a *specialized secure VM host* platform to secure a *commodity OS* to support an application might spend the same effort much more profitably developing a specialized non-virtualized platform in the first place!

2. Seeing double: twice the maintenance fun.

Securing a remote management interface to the host system is not the only practical challenge: there is the management itself, such as applying security updates and handling any underlying network changes (which are not infrequent in enterprise environments). In the situation when *both* the protected guest software and the protecting host OS are commodity, security updates will likely be required for both, especially considering the network-facing management functionality of the host system.

With an appropriate automated update infrastructure, the effort needed for doubling the number of maintained systems might not be doubled, even though in the case when the host and guest commodity systems are not the same, *two* infrastructures would be needed to support them.

However, whenever *update testing* is required prior to deployment – an expected procedure in mission-critical environments such as SCADA systems – the labor costs of such testing cannot be substantially reduced. Indeed, even if an organization purchases management services from a third party, that party will incur them anyway for each software item, and will have to pass them to the customer.

3. The temptation of guest-to-host API.

So far we considered the protective host system and the VM(M)-protected guest one as separate non-communicating entities, the guest OS and its applications written to be unaware of running in a virtualized environment, even though this illusion may be neither perfect nor intended to be perfect, i.e., has no goal of suppressing any virtualization-revealing "red pills" (as coined by [42]).

However, in practice the symbiotic relationship between the protector and the protected presents an almost insurmountable temptation to developers: that of guest–host communications.

Indeed, the typical guest's *raison d'etre* is implementing some complex and valuable functionality, which implies that it is likely to undergo some complex and important events or state changes. These events or state changes are best described and detected within the guest's internal logic, but the host system might need to be notified of them – from the guest. An important example is export of data formatted by the guest and parsed by the host, for administration, management or auditing, e.g., sharing of high-value or configuration files across filesystems to avoid the cost managing several independent copies, migration of process or data from an overly loaded cluster/cloud node, or attempts at guest recovery without losing all of its state through a reboot.

Theoretically, any guest–host interfaces are a bad idea because it breaks the platform's security model, in particular, complicates its isolation assumptions. However, the ongoing record of VM escape attacks (e.g.,[49, 27]) shows that the temptation of adding such interfaces never wanes despite previous adverse experience, and may even be essential to "cloud" environments.

In the light of the above analysis of practical system management challenges, it should be clear that interactions between the host VMM layer and the protected guest through dedicated interfaces will appear as a tempting way of addressing these challenges. However, this will mean passing and parsing of complex inputs, possibly crafted by the compromised guest in the host – with the usual disastrous consequences for the host's security.

4. Information Flow Policy.

The use of virtualization to support isolation is really only half a solution. If you take a "deny by default" approach, then you are probably at least as interested in what safe interactions the virtualization framework enables. As Bellovin pointed out in his October 2006 CACM article "Virtual Machines, Virtual Security," [5] virtual containers need to transfer and exchange information, and what current virtualization platforms seem to lack is a systematic way to talk about the flow of information across the isolation boundaries between virtual containers. He further pointed out that, even if this capability existed, policy writing is still a hard problem that isn't very amenable to automation.

Note that the general form of this problem (controlling information flow between task units) exists both within the context of traditional operating systems (process, filesystem, and memory space isolation) and in a virtualization environment. Even if a policy framework for controlling information flows between virtual containers (e.g., guest OSs) existed, then all the hard work falls on whoever undertakes the task of policy engineering: specifying the many ways that two or more threads of execution in a fluid set of guests might affect some amorphous set of resources across those guests. The details matter here: the type of events a virtualization system observes has an impact on the performance of the system designed to measure them and make security-related decisions about them.

5. Resource Provider or Reference Monitor?.

From a certain point of view, the amalgam of virtualization technology and information security techniques represents a rather strange blend. What does emulation or multiplexing of physical devices have to do with the enforcement of a wide variety of security properties?

As we mention above, the easiest answer (and the most traditional one) is that virtualization provides an effective means of isolating execution environments; virtualization seems like a natural way to provide isolation between execution containers. Complete isolation, however, is the exception rather than the rule (as Bellovin hints [5]), and customizing the communication between such containers presents a challenge. Thus, even the "obvious" security application of virtualization is fraught with difficulty.

Unfortunately, it appears that little thought has been given to what the best way is to combine the twin roles of resource provider and reference monitor within a single virtualization framework. As a result, virtualization environments can find themselves attempting to measure security-relevant properties of a system in ways that are both creative and convoluted. In essence, the set of events that

are interesting from a security viewpoint[5] are not necessarily the set of events that the virtualization framework was built to intercept and observe with a minimal performance impact. Karger and Safford's article [23] details the I/O complexities of most of the popular approaches to providing virtualization.

In this area, we have previously (1) identified the problem of designing an efficient event trapping system of use for both security policy enforcement and virtualization [10] and (2) considered new hardware support for fine-grained information flow labeling and access control that did not involve virtualization [9].

While the suggestion that the design of current virtualization solutions is actually a hindrance to providing security solutions may not sit well with folks interested in touting a particular virtualization solution's security capabilities, we argue that the community has a unique opportunity to make sure that VM platforms are designed to do the things we are asking them to do. Now is also a good time to note that the stunning complexity of VMM I/O subsystems, the performance hacks therein, and the backdoor management interface all suggest that even the basic isolation story rests on somewhat shaky ground — a reasonable basis for suggesting that alternative approaches may deserve some research.

We find ourselves at a unique point in time: we can try to identify the right design for doing these two disparate tasks at once, or we can muddle through by abusing a framework meant for resource multiplexing rather than program supervision. In either case, we still must balance the tradeoff between the virtualization frameworks I/O architecture and subsystems and the trustworthiness of the reference monitor. Ironically, as we depend on VM frameworks to implement more security functionality, these systems become less trustworthy even as they become more trusted.

3. CASE STUDY: DETECTING MALICIOUS COMPUTATION WITH AUTOSCOPY

Our case study focuses on protecting privileged code (i.e., an OS kernel) from rootkit operation. This kind of protection normally involves the use of a virtual machine underneath the victim or target kernel. We explore some design considerations involved in a novel kind of self-monitoring system we call *Autoscopy*. Due both to space restrictions and our desire not to distract from the main "NSPW" point of the paper, we only provide an overview of Autoscopy here; we refer the interested reader to our technical reports [37, 38]) for more detailed design and performance information.

Attacks aimed at subverting kernel control flow (such as those meant to embed a rootkit) present one of the hardest malware threats to defend against. With Autoscopy, we suggest a new anomaly–based approach to detecting attempts to subvert OS control flow via the introduction of *malicious computation* patterns (rather than other aspects of malcode or rootkits at rest). Autoscopy's monitoring infrastructure lives in the kernel itself and is not solely provided by a "layer below" component like a PCI card, a hypervisor, or a VM host OS.

Autoscopy's design manifests from our desire to convey only a moderate impact on system performance. Consequently, we do not use virtual machines to provide a monitoring environment. Instead, we show how to leverage an existing kernel instrumentation framework, `kprobes`, to identify attack vectors from *within* the kernel rather than *underneath* it. In our experiments, Autoscopy imposes an overhead ranging from about 2% to 5% (20% in one special case). Autoscopy can train on a compromised kernel without a reduction in detection ability, and it detected all rootkits that we experimented with.

[5]And this set depends on what type of "security" you're interested in measuring...from integrity of control flow or data items to information flow to authorization and access control.

3.1 Autoscopy Design

Autoscopy discovers function pointers that are potentially vulnerable to rootkits by searching for them in kernel memory and inserting a `kprobe` that verifies that they are called in a "normal" fashion. The `kprobes` framework is a mechanism originally developed by IBM researchers and introduced in the Linux kernel version 2.6.9 that allows the execution of arbitrary C code at any point during the kernel's execution.

The primary goal of our detection system is to distinguish between kernel hook functions and rootkit hook functions, both of which use function pointers present in the kernel's data sections. Since rootkits endeavor to hide, they must maintain an interface consistent with kernel operation and the function that they interpose on. Hooks typically invoke the original function so as to mimic the hooked function's original behavior [26]. Preserving the original control flow helps a rootkit avoid simpler forms of detection.

A hook function calling another hook function is exactly the kind of behavior we wish to detect. This behavior defines the pattern our detection system will look for as it uses the hook table we built during the training phase. Since both hook functions are indirect CALL instructions, and the arguments passed to them are identical, we report this anomaly as a symptom of a rootkit interposing on standard control flow patterns. Using `kprobes` on the functions identified during our training phase, we look for this pattern during each probe handler. We put a list of the hooks (and runtime targets) we found on our website[6].

3.2 Detection Overview

During training, if we determine that the hook function is in an LKM (this address would change across reboots), we first put a probe on the caller's CALL instruction. When this probe is hit during the testing phase, we disassemble the CALL and obtain the address of the target (callee) function. Given the hook function (either via this method or via the training phase), we insert a probe on the given address and wait until the function is called during kernel execution. Note that if the function is never called, our system imposes no overhead. If the function is called, our probe handler then:

1. Determines whether the function was called through a pointer hook by performing reverse disassembly to verify whether this was indeed an indirect call. In the case of a direct call, we return.

2. Performs the general detection phase (for details see our technical reports [37, 38]).

3. Handles the case when a rootkit may already be present on the system when autoscopy is installed.

4. Propagates context information to other probe handlers to support the detection in Step 3.

5. Caches necessary probes to reduce probe creation and deletion overhead.

[6]`http://www.cs.dartmouth.edu/~pkilab/autoscopy`

3.3 Attacks on Autoscopy

We now consider a few of the possible attacks that a rootkit might conduct against autoscopy and detail our countermeasures. We reiterate that since autoscopy operates at the same privilege level as the attacker, by definition, we cannot prevent the attacker from overwriting kernel text or any self-modification of code. We thus rely on other approaches to guard the kernel text (taking breakpoint instructions into account): AMD/Intel's No-Execute (NX) bit, SWATT [45], and similar systems. We emphasize that, by themselves, these approaches cannot detect even moderately sophisticated rootkits that employ some form of dynamic hook modification or control-flow redirection. We see these tools as building blocks to help systems such as Autoscopy that are directed towards more devious rootkits than the ones that just modify static text or data. We do not see this assumption as a limitation; rather, our system is aimed at detecting and defeating the more advanced threats that have evolved to *not* rely on modifying kernel text.

Rootkits might attempt to subvert autoscopy in a number of ways:

1. **By modifying the hook to point to the original callee, then calling a function that uses this hook to redirect flow to the orginal callee, and finally reverting the hook back to the rootkit:** autoscopy can foil this attack, since it recognizes the two indirect calls even though they might potentially be situated far apart on the stack.

2. **By constructing CALL instructions manually, i.e. saving registers and flags, pushing <args, return IP, current FP> on the stack, and executing a JMP to the target:** We detect this in the probe handler by verifying that the last executed instruction within every function on the stack is indeed a CALL instruction before determining the nature of the call.

3. **By modifying its stack to remove the rootkit's call frame, then executing a JMP to the target:**. By analyzing the operand of each call instruction that we disassemble and verifying that the call operand corresponds to the actual function that was called, we can detect such an attack. Since the rootkit's hook function is now basically "missing" on the stack, but the caller's CALL operand would still contain the address of the rootkit hook, autoscopy would be able to detect this attack.

4. **By creating an alternate stack that omits the rootkit's call frame, modifying the stack pointer to point to the new stack, and executing a jump to the original callee:** This is just an elaborate version of the previous attack, and can be detected in the same way as outlined before.

5. **Executing a code duplication attack:** A rootkit could potentially bring in a copy of functions it intends to subvert or hook, and then return to those functions (which would not have been monitored by Autoscopy) rather than the native kernel function. We believe that such an attack is impractical given the constraints of writing a working kernel rootkit (we expand on reasons why below).

3.4 Code-duplication Attack

Since Autoscopy's detection of malicious computation is based on catching flow control deviations from those of known good kernel code paths, and, more specifically, on catching the diverted ("hooked") codepath as it rejoins the inteded flow, one concern is that the attacker could bring along enough code duplicating the kernel's own but not profiled by Autoscopy's learning phase to avoid detection.

While this is theoretically possible — and in fact influential publications like SubVirt [24] postulated that a rootkit might bring with itself an entire OS image [43] — we believe that it contradicts a steady trend in actual rootkit engineering, and is not, in practice, a credible threat, for the following reasons.

First, such low-level rootkits must faithfully duplicate the underlying hardware under all circumstances. High fidelity replication of hardware behavior, however, is quite difficult and error-prone [16] due to various discrepancies in CPUs, MMUs, TLBs, and timer chips.

Second, for a non-SubVirt-style rootkit, injecting "enough" duplicate code to never return to an intended kernel code path would require covering the lower layers of the kernel, below the well-defined VFS and DDK interfaces. If we think, e.g., of a system call going through the `sys_call_table`, and then calling file system-specific functions by pointer through the VFS dispatch tables and then on to calling lower, driver-level interfaces, then, having hijacked a call on the way down, a "code-duplicating" rootkit would need to also supply all of the code encountered on the way up.

This means that the code-duplicating attacker must provide code that exactly matches the state created by the hijacked kernel path on its way down by his duplicated code that handles it on the way up. This requires matching the kernel build pretty closely, and, at the very least, burdens the attacker with extra information gathering (just like daemon banners, kernel version and configuration can be trivially faked by defenders, a classic defence trick recommended by, e.g., the SANS Institute). The price of error is a kernel panic, which is directly at odds with the primary purpose of a rootkit: stealth. In fact, kernel panics have led directly to the discovery of installed rootkits in real life scenarios [29].

To a practically-minded attacker, this presents a significant engineering problem, further exacerbated by the (forced) movement of rootkits into the lower layers of the kernel to hide from rootkit detectors, exemplified by [34, 33, 50], further illustrated by examples of in-depth subverting specific Linux subsystems such as the network stack, memory management, and the scheduler [8, 12, 51], as well as in Rutkowska's Linux rootkit work [41]. This downward migration is forced by the improvements in the defender's understanding of the hijacking opportunities presented by the standard kernel APIs and de-facto DDKs and the corresponding evolution of detection tools.

However, it is of crucial importance to observe that these "deeper" rootkit designs stick to well-defined developer interfaces rather than attempting arbitrary hooking – since rootkit stealth requires reliability, and software reliability under diverse software and hardware versions can only be achieved via observing interface contracts, malware or not.

Thus, from the rootkit engineering point of view, code duplication in the lower layers of the kernel is neither attractive nor trivial. In fact, irrespective of the purpose (attack, defense, or enhancing functionality), injecting code in this manner presents a difficult challenge! Not surprisingly, hot patching a running system to inject new code and state is known to be a difficult research and engineering problem, and even cold patching of well-understood faults can impose substantial testing costs and lengthy (weeks or months) quality assurance requirements on vendors to ensure stability and reliability.

4. RESEARCH QUESTION

One major theme that arose from the reviewer's comments on the submission version of this paper was a desire for more direct comparison of the relative performance and efficacy of approaches to measurements of security-related events. Karger and Safford [23]

have studied this question, and they conclude that:

> Modern I/O architectures are quite complex, so keeping a virtual machine monitor (VMM), or hypervisor, small is difficult. Many current hypervisors move the large, complex, and sometimes proprietary device drivers out of the VMM into one or more partitions, leading to inherent problems in complexity, security, and performance.

In their article, they review a variety of virtual machine architectures and illustrate both the performance challenges and resulting complexity arising from attempting to deal with I/O management and interception in each virtual machine architecture.

In his 2009 ASIAN keynote [22], Karger goes on to say:

> It is widely believed that the use of a virtual machine monitor (VMM) is at least as secure, if not more secure than separate systems. A recent Information Week survey reports that 55% of responding business technology professionals believe that a system running in a virtual machine is as safe as physical servers and 20% believe it safer than physical servers...in reality, the security of a single system running in a virtual machine can never be as secure as that single system running in its own dedicated physical hardware...because there are more lines of code that must be correct, the VMM case always has more opportunity for exploitable flaws.

4.1 Future Experiments

It would be of interest to conduct experiments that evaluate the relative performance impact of several different approaches (such as those reviewed by Karger and Safford in their article) to virtual machine security — keeping in mind that some of these architectures are used with different security goals in mind (e.g., securing the hypervisor from guest OSs, securing a guest OS from vulnerable applications, securing guests from each other, securing a VM host from another VM host). We did not conduct such experiments during the time from notification of acceptance to pre-proceedings, although we agree that they would be a valuable basis for further research in this area. Whereas Karger and Safford provide a model-based argument for why performance and efficacy might degrade, their argument can be supplemented with an experimental evaluation of different scenarios.

Most research papers (see Section 5) seem to claim either very low performance overhead ($< 10\%$) or a significant (order of magnitude or more) slowdown, depending on the invasiveness of the instrumentation. Direct comparisons might be difficult to draw, however, given the sometimes differing goals of these systems.

For example, a comparison between a system like Autoscopy (2..5% overhead) and a system like SecVisor is hardly apples-to-apples; in one instance, self-protection of a commodity OS on top of bare metal has almost negligible cost, whereas a hypervisor concerned with protecting the kernel integrity of guest OSs has a performance impact that exceeds 100% (in relation to Xen – and so, even slower than bare metal) in many cases (see Figure 12 in the SecVisor paper [44]). In any event, our arguments in Section 2 focus on additional costs beyond performance concerns. In other words, we believe that self-monitoring has a substantial performance edge, but for those concerned that it may be vulnerable to attacks or subversion, it has several other advantages over different forms of VM-based monitoring. Even this way of framing the question is not quite satisfactory to critics, who naturally want both zero performance impact as well as impregnable security and who seem ready to settle for VM-based solutions that are neither zero performance impact nor provide anything close to unassailable security.

Two main challenges seem to exist in the space of protecting low-level or highly-privileged code. First, the embedding of self-monitoring must in some way attempt to protect itself from malware (rather, malicious computation) executing at the same level of privilege. Thus, self-monitoring systems (like Autoscopy), in order to gain acceptance from the community, must provide an argument as to why malicious computation will find it difficult (i.e., genuinely hard or impossible, not just laborious) to (1) detect, (2) disable, or (3) blind self-monitoring. In essence, *good* self-monitoring instrumentation should be looking at execution events or data artifacts which malicious computation **must** use or modify in order to gain control.

Second, a race exists here: the detection granularity depends to a certain extent on performance considerations; should malware be able to use or modify critical state before detection occurs, then malware has a potential avenue of attack — it has established itself at the same level of privilege. Depending on the architecture of the system, however, malicious computation might have to perform additional actions (which may or not be noticed by defensive instrumentation) to succeed in disabling the monitoring mechanism itself. Self-monitoring should have some way of controlling this race if it is to be effective.

5. RELATED WORK

While virtualization serves as an enabler for security solutions, it does not necessarily function as a security provider without some careful thought about the design and management of the systems, processes, and people surrounding it.

Security Applications of Virtualization Projects involving aspects of virtual machines and security range from those that show how a VM or VM framework can provide or enhance security functionality intrinsically to those that use VMs as containers to form part of a larger security system. The former type of project looks at what functionality can be added to the VM framework's code to implement things like access control, trusted computing [6], isolation, malware reverse engineering[7], virus scanning, network content filters, information flow analysis, and host or network anomaly detection.

The latter type of project employs to VMs to provide a convenient disposable container to examine the execution of a guest application or OS. Some examples of this use include opening potentially infected emails [48, 47] or web pages [36], testing out patches or other software fixes, and recording application state for replay [14].

Rootkit Detection Traditional rootkit detection approaches like chkrootkit[8] and Rootkit Hunter[9] tend to look for threat-specific information: either known rootkit binaries or known alterations of system binaries, configuration files, or system state (*i.e.,* a network interface set in promiscuous mode). While these tools provide some assurance against basic kernel or user–level rootkits and provided a vital defense during the advent of such rootkits, their general approach is similar to signature-based virus scanning.

Most current approaches to kernel rootkit detection monitor the "known good state" of some static data. Microsoft's RootkitRe-

[7] `http://bitblaze.cs.berkeley.edu/`

[8] `http://www.chkrootkit.org/`

[9] `http://www.rootkit.nl/projects/rootkit_hunter.html`

vealer[10] compares the results of API calls against on-disk registry and file data to detect manipulation of this data by a rootkit in the middle. Other current approaches examine static portions of the kernel text segment (*i.e.,* kernel code) to detect rootkits that attempt to overwrite existing kernel functions. Such approaches typically check cryptographic hashes of kernel text memory ranges against a whitelist of hash values (a mechanism similar to Tripwire[11]). As one example, Petroni *et al.* [35] suggest using a bus-mastering PCI card to periodically monitor the contents of kernel memory via a DMA-like mechanism. Finally, identifying potential kernel *hooks* (control flow transfer points that might enable a rootkit to interpose on kernel execution) for defensive purposes is one interesting avenue of recent research [53, 55] because it enables guarding these locations.

Control flow integrity (CFI) [1] ensures that runtime program execution paths conform to a Control Flow Graph (CFG) learned via static source code analysis. Petroni and Hicks propose State-Based CFI [31]; their approach validates the data structures in the kernel instead of tracking individual execution branches. This approach allows the inspecting process to be external to the system (*e.g.,* in a VMM or on a PCI card) but opens a window during which an ephemeral rootkit might be deployed. SBCFI can also incur close to 40% overhead on a typical machine running Xen (but a major portion of the overhead is attributed to Xen itself).

Kruegel, Robertson, and Vigna [28] propose the use of symbolic execution to analyze LKMs *before* they are loaded into the kernel. This approach, however, requires a static whitelist of valid memory regions and kernel symbols in order to distinguish malicious and benign LKMs. Building such a whitelist can present a challenge for large, constantly evolving systems.

The NICKLE system [39] implements a shadow memory controlled by a VMM for the entire region of kernel memory in the guest machine. On guest boot, all known authenticated kernel instructions are copied into the shadow memory. At runtime, each kernel instruction fetch is verified by comparing the shadow memory maintained by the VMM with the actual physical memory at that location. Differences indicate the presence of a rootkit. The difficulty with such shadow memory schemes is how to handle LKMs: manual certification of an LKM via code signing does not guaranee the absence of malcode in the LKM. It is possible that the large body of work done to handle untrusted modules applies here, but it remains an open problem how to integrate these techniques with current production operating systems.

Paladin [3] divides the system into two *protected zones*. Paladin safeguards these zones from illegal accesses by malware. The *Memory Protected Zone* consists of the kernel hook tables and other static regions of the kernel that need to be protected from rootkits, while the *File Protected Zone* consists of system binaries and libraries that need to be prevented against modifications. Given the specifications of these zones, Paladin uses a VMM to monitor write accesses across the system for validity. Any time Paladin detects an invalid access, it identifies and kills the entire process subtree for the process that invoked the offending write. A Paladin driver resides on the guest machine to facilitate this process and interacts with the monitor to aid detection. More fundamentally, monitoring writes to specified locations does not prevent rootkits from hijacking function hooks within data structures because these locations are *meant* to be overwritten, and malcode can use existing kernel code to execute the overwrite, thus avoiding detection based on the "origin" of an instruction.

[10] `http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx`

[11] `www.tripwire.org`

One closely related piece of work is the HookSafe system [52]. HookSafe maintains a shadow list of hooks in a protected page; the system is based on the assumption that hooks rarely change (i.e., they are typically read many times during system execution but written only a relatively few number of times).

The novelty of Autoscopy's approach is more clearly seen when we contrast it with alternative rootkit detection paradigms: a comparison that also helps clarify Autoscopy's design philosphy. Since the essential purpose of a rootkit is to *deceive* the system administrator or his monitoring software, we can classify anti-rootkit measures according to the kinds of deceptions they aim to counter.

For example, HookMap [53] profiles kernel execution paths used by a set of reporting utilities the user believes are most likely to be targeted by rootkit deception. Thus, HookMap's design proceeds from this choice of *likely targets of deception*, which are then profiled dynamically. SBCFI [31] is an example of a different approach – it analyzes and validates the kernel's use of global variables, *i.e.,* it targets deviations from the *normal patterns in (global) data use*, learned via static analysis. Both systems rely on an *external virtual machine monitor* (VMM) to capture and mediate events of interest that must comply with their representations of the normal behavior. For these systems, relying on a VMM proves essential because x86 does not provide effective interception capabilities for their respective sets of mediated events. In contrast, Autoscopy adopts a different approach to self–monitoring control flow, by means that push but do not exceed the limits of hardware's ability to mediate control flow events.

6. CONCLUSION

We suggest that the systems security community should refrain from premptorially dismissing non-VM approaches to isolation and inspection, precisely because virtual machines offer a tantalizing computing primitive. We argue that other ways of assessing trustworthy behavior and measuring security properties exist, and we examined a case study of one way to provide such a monitoring mechanism at the same privilege level as the malicious computation it tries to detect. Such an approach seems risky – and to those steeped in the comforts of a VM mindset, this new kernel-level self-monitoring mechanism seems wrong. Such a viewpoint says: "but the problem is already solved by putting the kernel and everything above it in a VM!" and dismisses the arguments as to why one might *not* want to depend on a VMM as "handwaving."

With such a viewpoint, we risk avoiding the natural evolution of technical approaches. Indeed, why explore other ways of doing things if one way already exists and "solves the problem"? And of course a new way has to be proven all-around superior before it deserves to be communicated. From a certain point of view, there is always something that appears to "solve the problem." Why Perl when shell and grep are already there? Why use Ruby when you have Perl? The Linux kernel was dead on arrival according to certian sages (monolithic kernels being so "yesterday") and so on.

The community runs the danger of failing to realize that virtual machines, despite their practicality in some situations, are not the sole method of implementing a reference monitor. Furthermore, this mechanism has its own risks, shortcomings, and drawbacks. For example, with a VMM, one must effectively provision *two kernels*, one of which may be difficult to manage remotely except by way of the upper-level kernel it protects, or through something like an extra "OS in the chipset/network card/SMM", and all this extra complexity is assumed to come for free.

Justification Statement

This work takes a position that questions an existing paradigm: the use of virtualization to support systems security. Are security and virtualization *really* a natural match? Virtual machines have recently revived for a number of applications. Somewhat troubling to us, virtual machines have also been seen in the systems security community as the most practical form of a reference monitor, and have been pressed into service in any number of ways to provide isolation and introspection.

A Virtual Crutch This paper argues that such a paradigm is injurious to the systems security community: we've become addicted to a crutch that is not suitable for all situations, and it is furthermore unclear that the many forms of virtualization are the correct tool for providing isolation in terms of security properties. Virtual machines are unfit and cumbersome for security tasks for a number of reasons that we examine in the paper. Virtualization is a crutch: it may add stability, but slows down an otherwise healthy system. One whole problem with the VM-based approach is that they typically ignore the incumbant complexity of managing the VM infrastructure.

We note that the main contribution of this paper is the argument dealing with the appropriateness of virtual machines as a security technique. In other words, the "NSPW" contribution is *not* the relatively well-polished consideration of the Autoscopy system (our case study), but rather an exploration of whether VM alternatives are a feasible and useful avenue of research that should not suffer a premature death simply because the collective wisdom assumes VM methods are always appropriate.

Modifications

We have modified this paper in the following ways from the submission version:

- expand "conclusion.tex" with text dealing with the perceived attitude of the VM-only paradigm
- add spengler quote from dailydave mailing list
- added two sentences to abstract dealing with takeway message
- significantly trimmed down case study of Autoscopy
- added "Research Question" section in response to reviewer comments. We hope to continue expanding this section through and after the workshop with attendee comments

7. REFERENCES

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2005).

[2] BAEK, K.-H., BRATUS, S., SINCLAIR, S., AND SMITH, S. Attacking and Defending Networked Embedded Devices. In *2nd Workshop on Embedded Systems Security (WESS)* (Salzburg, Austria, October 2007).

[3] BALIGA, A., CHEN, X., AND IFTODE, L. Paladin: Automated Detection and Containment of Rootkit Attacks. In *Technical Report DCS-TR-593, Rutgers University, Department of Computer Science* (2006).

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In 19^{th} *ACM Symposium on Operating Systems Principles (SOSP)* (October 2003).

[5] BELLOVIN, S. M. Virtual Machines, Virtual Security. *Communications of the ACM 49*, 10 (October 2006).

[6] BERGER, S., CACERES, R., GOLDMAN, K. A., PEREZ, R., SAILER, R., AND VAN DOORN, L. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security Symposium* (2006).

[7] BICKFORD, J., O'HARE, R., BALIGA, A., GANAPATHY, V., AND IFTODE, L. Rootkits on smart phones: attacks, implications and opportunities. In *HotMobile '10: Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications* (New York, NY, USA, 2010), ACM, pp. 49–54.

[8] BIOFORGE. Hacking the Linux Kernel Network Stack. Phrack 61–0x0d.

[9] BRATUS, S., JOHNSON, P., LOCASTO, M. E., RAMASWAMY, A., AND SMITH, S. W. The Cake is a Lie: Privilege Rings as a Policy Resource. In *Proceedings of the* 2^{nd} *ACM Workshop on Virtual Machine Security (VMSec) held in conjunction with ACM CCS 2009* (October 2009).

[10] BRATUS, S., LOCASTO, M. E., RAMASWAMY, A., AND SMITH, S. W. Traps, Events, Emulation, and Enforcement: Managing the Yin and Yang of Virtualization-based Security. In *Proceedings of the* 1^{st} *ACM Workshop on Virtual Machine Security (VMSec) held in conjunction with ACM CCS 2008* (October 2008).

[11] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of CCS 2008* (Oct. 2008), P. Syverson and S. Jha, Eds., ACM Press, pp. 27–38.

[12] BUFFER. Hijacking Linux Page Fault Handler. Phrack 61:07.

[13] DESIGNER, S. Getting around non-executable stack (and fix). Bugtraq mailing list, August 1997.

[14] DUNLAP, G. W., KING, S., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)* (February 2002).

[15] DURDEN, T. Bypassing PaX ASLR protection. *Phrack 59*, 5 (July 2002).

[16] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility is not Transparency: VMM Detection Myths and Realities. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–6.

[17] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In 10^{th} *ISOC Symposium on Network and Distributed Systems Security (SNDSS)* (February 2003).

[18] GARFINKEL, T., ROSENBLUM, M., AND BONEH, D. Flexible OS Support and Applications for Trusted Computing. In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems* (May 2003), pp. 145–150.

[19] GEER, D. Keynote, source boston conference. `http://www.sourceconference.com/2008/sessions/dan-geer-keynote.html`, March 2008.

[20] HANDLEY, M., PAXSON, V., AND KREIBICH, C. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the USENIX Security Conference* (2001).

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

[22] KARGER, P. A. Securing virtual machine monitors: what is needed? In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (New York, NY, USA, 2009), ACM, pp. 1–2.

[23] KARGER, P. A., AND SAFFORD, D. R. I/O for Virtual Machine Monitors: Security and Performance Issues. *IEEE Security and Privacy Magazine 6*, 5 (2008), 16–23.

[24] KING, S. T., CHEN, P. M., WANG, Y.-M., VERBOWSKI, C., WANG, H. J., AND LORCH, J. R. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2006).

[25] KING, S. T., DUNLAP, G., AND CHEN, P. Operating System Support for Virtual Machines. In *Proceedings of the General Track: USENIX Annual Technical Conference* (June 2003).

[26] KONG, J. *Designing BSD Rootkits: An Introduction to Kernel Hacking*. No Starch Press, April 2007.

[27] KORTCHINSKY, K. Cloudburst: Hacking 3D (and Breaking Out of VMware. BlackHat USA, July 2009.

[28] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the* 20^{th} *Annual Computer Security Applications Conference (ACSAC)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 91–100.

[29] LOCASTO, M. E., BURNSIDE, M., AND BETHEA, D. Pushing Boulders Uphill: The Difficulties of Network Intrusion Recovery. In *USENIX Large Installation and Systems Administration Conference* (2009).

[30] NERGAL. Advanced return-into-lib(c) exploits (PaX case study). *Phrack 58*, 4 (December 2001).

[31] NICK L. PETRONI, J., AND HICKS, M. Automated Detection of Persistent Kernel Control-flow Attacks. In *Proceedings of the* 14^{th} *ACM conference on Computer and Communications Security (CCS)* (New York, NY, USA, 2007), ACM, pp. 103–115.

[32] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the* 5^{th} *Symposium on Operating Systems Design and Implementation (OSDI 2002)* (December 2002), pp. 361–376.

[33] PALMERS. 5 Short Stories about execve (Advances in Kernel Hacking II). Phrack 59–0x05. Team TESO.

[34] PALMERS. Sub proc_root Quando Sumus (Advances in Kernel Hacking). Phrack 58–0x06.

[35] PETRONI, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pp. 179–194.

[36] PROVOS, N., MAVROMMATIS, P., RAJAB, M. A., AND MONROSE, F. All Your iFRAMEs Point to Us. In *USENIX Security Symposium* (2008).

[37] RAMASWAMY, A. Detecting kernel rootkits. Tech. Rep. TR2008-627, Dartmouth College, Computer Science, Hanover, NH, September 2008.

[38] RAMASWAMY, A. Autoscopy: Detecting Pattern-Searching Rootkits via Control Flow Tracing. Tech. Rep. TR2009-644, Dartmouth College, Computer Science, Hanover, NH, May 2009.

[39] RILEY, R., JIANG, X., AND XU, D. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *RAID* (2008).

[40] ROSCOE, T., ELPHINSTONE, K., AND HEISER, G. Hype and Virtue. In *Proceedings of the* 11^{th} *Workshop on Hot Topics in Operating Systems (HOTOS XI)* (May 2007).

[41] RUTKOWSKA, J. Invisible things (papers and presentations). http://invisiblethings.org/papers.html.

[42] RUTKOWSKA, J. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`, November 2004.

[43] RUTKOWSKA, J. Subverting Vista Kernel For Fun And Profit. SyScan/BlackHat 2006 presentation, July 2006.

[44] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), ACM, pp. 335–350.

[45] SESHADRI, A., PERRIG, A., DOORN, L. V., AND KHOSLA, P. Swatt: Software-based attestation for embedded devices. In *In Proceedings of the IEEE Symposium on Security and Privacy* (2004).

[46] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007* (Oct. 2007), S. De Capitani di Vimercati and P. Syverson, Eds., ACM Press, pp. 552–61.

[47] SIDIROGLOU, S., IOANNIDIS, J., KEROMYTIS, A. D., AND STOLFO, S. J. An Email Worm Vaccine Architecture. In *Proceedings of the* 1^{st} *Information Security Practice and Experience Conference (ISPEC)* (April 2005).

[48] SIDIROGLOU, S., AND KEROMYTIS, A. D. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security* (June 2003), pp. 220–225.

[49] SKOUDIS, E., AND LISTON, T. Virtual Machine Security Issues. SANS FIRE Conference, July 2007.

[50] STEALTH. Kernel Rootkit Experiences. Phrack 61–0x0e.

[51] UBRA. Process Hiding and The Linux Scheduler. Phrack 63:18.

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

[53] WANG, Z., JIANG, X., CUI, W., AND WANG, X. Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[54] WOJTCZUK, R. Defeating solar designer non-executable stack patch. Bugtraq mailing list, 1998.

[55] YIN, H., LIANG, Z., AND SONG, D. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the* 15^{th} *Annual Network and Distributed System Security Symposium (NDSS)* (February 2008).