Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems

Citation: Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In Proceedings of the Second Asia-Pacific Workshop on Systems (APSys '11). ACM, New York, NY, USA, , Article 5 , 5 pages.

As Published: http://dx.doi.org/10.1145/2103799.2103805

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/73008

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Linux kernel vulnerabilities: State-of-the-art defenses and open problems

Haogang Chen   Yandong Mao   Xi Wang   Dong Zhou[†]
Nickolai Zeldovich   M. Frans Kaashoek

MIT CSAIL   [†]Tsinghua University

ABSTRACT

Avoiding kernel vulnerabilities is critical to achieving security of many systems, because the kernel is often part of the trusted computing base. This paper evaluates the current state-of-the-art with respect to kernel protection techniques, by presenting two case studies of Linux kernel vulnerabilities. First, this paper presents data on 141 Linux kernel vulnerabilities discovered from January 2010 to March 2011, and second, this paper examines how well state-of-the-art techniques address these vulnerabilities. The main findings are that techniques often protect against certain exploits of a vulnerability but leave other exploits of the same vulnerability open, and that no effective techniques exist to handle semantic vulnerabilities—violations of high-level security invariants.

1. INTRODUCTION

An OS kernel is a part of the trusted computing base (TCB) of many systems. Vulnerabilities in the kernel itself can allow an adversary to bypass any kernel protection mechanisms, and compromise the system, such as gaining root access. Much research has gone into mitigating the effects of kernel vulnerabilities, but kernel vulnerabilities, and more importantly, kernel exploits, are still prevalent in Linux. This paper investigates where the research community may want to focus its attention, by analyzing past Linux kernel vulnerabilities, categorizing them, evaluating what defensive techniques might have been used to prevent them, and speculating on what the remaining open problems are.

Surprisingly few studies have been performed to understand the types of kernel vulnerabilities that occur in practice. A study by Arnold et al. [3] argues that every kernel bug should be treated as security-critical, and must be patched as soon as possible. Mokhov et al. explore how kernel programmers patch known vulnerabilities [19]. Christey and Martin report on vulnerability distributions in CVE [8]; our study is also based on CVE and our findings are consistent with that study, but ours focuses only on kernel vulnerabilities. Neither of the studies shed light on what techniques could be used to prevent unknown vulnerabilities from being exploited. In this paper, we present a case study of Linux kernel vulnerabilities discovered from January 2010 to March 2011. We categorize these vulnerabilities by the kind of programming mistake the developers made, and the impact it has on security.

Based on this list of kernel vulnerabilities, we perform a second case study, by examining how effective techniques proposed by researchers might be at mitigating vulnerabilities in the Linux kernel. These techniques include runtime mechanisms such as code integrity checks [22], software fault isolation [6, 15], and user-level device drivers [5], as well as bug-finding tools and static analysis [1, 2]. This examination is not empirical, but is based purely on our understanding of the techniques. Nonetheless, we believe it can be helpful in identifying what classes of vulnerabilities can already be solved, and what open problems remain.

The findings of our two studies are as follows. First, we find that there are 10 common classes of kernel vulnerabilities in Linux, which may lead to attacks ranging from arbitrary memory modifications to information leaks to denial-of-service attacks. Second, we find that about $2/3$ of the vulnerabilities are in kernel modules or drivers, and that conversely, $1/3$ of the bugs are found in the core kernel. Third, we find that no single existing technique can prevent all kernel vulnerabilities, and that a technique often mitigates some exploits of a vulnerability but does not address other exploits of the same vulnerability. Fourth, there are certain classes of vulnerabilities that are not addressed at all. For example, semantic vulnerabilities, where high-level security invariants are violated, are difficult to catch with state-of-the-art techniques that focus mostly on memory safety and code integrity.

2. LINUX KERNEL VULNERABILITIES

Figure 1 categorizes the 141 Linux kernel vulnerabilities published on the CVE list from January 2010 to March 2011 and the type of attacks that can exploit the vulnerability. Despite their diversity, most of these vulnerabilities fall into 10 categories, based on the kind of programming mistake the developers made, as listed in the first column. Since each vulnerability can often be exploited in several ways, we further categorize the exploits in several *attack* classes. *Memory corruption* typically allows an adversary to perform arbitrary operations in the kernel, and is thus a superset of other types of attacks, such as *policy violation*, *DoS*, *information disclosure*, and others. Therefore if a vulnerability leads to a memory-corruption attacks, we do not count it under other attack classes. The rest of this section discusses the types of vulnerabilities we have found.

Missing pointer checks. The kernel omits `access_ok` checks or misuses "faster" operations such as `__get_user`, which does not validate the value of user-provided pointers or index variables to ensure that they point to user-space memory only. These bugs enable unprivileged processes to read from or write to arbitrary kernel memory locations, leading to memory corruption (CVE-2010-4258),

Vulnerability	Mem. corruption	Policy violation	DoS	Info. disclosure	Misc.
Missing pointer check	6	0	1	2	0
Missing permission check	0	15	3	0	1
Buffer overflow	13	1	1	2	0
Integer overflow	12	0	5	3	0
Uninitialized data	0	0	1	28	0
Null dereference	0	0	20	0	0
Divide by zero	0	0	4	0	0
Infinite loop	0	0	3	0	0
Data race / deadlock	1	0	7	0	0
Memory mismanagement	0	0	10	0	0
Miscellaneous	0	0	5	2	1
Total	32	16	60	37	2

Figure 1: Vulnerabilities (rows) vs. possible exploits (columns). Some vulnerabilities allow for more than one kind of exploit, but vulnerabilities that lead to memory corruption are not counted under other exploits.

Vulnerability	Total	core	`drivers`	`net`	`fs`	`sound`
Missing pointer check	8	4	3	1	0	0
Missing permission check	17	3	1	2	11	0
Buffer overflow	15	3	1	5	4	2
Integer overflow	19	4	4	8	2	1
Uninitialized data	29	7	13	5	2	2
Null dereference	20	9	3	7	1	0
Divide by zero	4	2	0	0	1	1
Infinite loop	3	1	1	1	0	0
Data race / deadlock	8	5	1	1	1	0
Memory mismanagement	10	7	1	1	0	1
Miscellaneous	8	2	0	4	2	0
Total	141	47	28	35	24	7

Figure 2: Vulnerabilities (rows) vs. locations (columns).

information disclosure (CVE-2010-0003), DoS (CVE-2010-2248), or privilege escalation (CVE-2010-3904 and CVE-2010-3081) if the process controls what data to write.

Missing permission checks. The kernel performs a privileged operation without checking whether the calling process has the privilege to do so. A vulnerability in this category results in a violation of a kernel security policy. The attacks that can exploit this vulnerability depend on what the security policy is, ranging from arbitrary code execution (CVE-2010-4347), privilege escalation (CVE-2010-2071 and CVE-2010-1146), to overwriting an append-only file (CVE-2010-2066 and CVE-2010-2537).

Buffer overflow. The kernel incorrectly checks the upper or lower bound when accessing a buffer (CVE-2011-1010), allocates a smaller buffer than it is supposed to (CVE-2010-2492), uses unsafe string manipulation functions (CVE-2010-1084), or defines local variables which are too large for the kernel stack (CVE-2010-3848). The attacks that can exploit this vulnerability are memory-corruption (for writes) or information-disclosure (for reads) attacks. An adversary can mount privilege-escalation attacks by overwriting nearby function pointers and subverting the kernel's control flow integrity.

Integer overflow. The kernel performs an integer operation incorrectly, resulting in an integer overflow, underflow, or sign error. The adversary can trick the kernel into using the incorrect value to allocate or access memory, allowing similar attacks as allowed by "buffer overflow" vulnerabilities. For example, overflow after multiplication can cause the kernel to allocate a smaller-than-needed buffer (CVE-2010-3442); underflow after subtraction can cause memory corruption beyond the end of a buffer (CVE-2010-3873); and sign errors during comparison can bypass bounds checking and cause information disclosure (CVE-2010-3437).

Uninitialized data. The kernel copies the contents of a kernel buffer to user space without zeroing unused fields, thus leaking potentially sensitive information to user processes, such as variables on the kernel stack (CVE-2010-3876). This category has 29 vulnerabilities, the highest of all categories. A direct attack using this vulnerability results in unintended information disclosure. However, vulnerabilities in this category may enable other attacks, such as attacks that require knowing the exact address of some kernel data structure, private kernel keys, or other kernel randomness.

Memory mismanagement. This category includes vulnerabilities in kernel memory management, such as extraneous memory consumption (CVE-2011-0999), memory leak (CVE-2010-4249), double free (CVE-2010-3080), and use-after-free errors (CVE-2010-4169 and CVE-2010-1188). For the vulnerabilities that we examined, an adversary can mount DoS attacks by exploiting them, although in general arbitrary memory corruption may be possible.

Miscellaneous. There are other types of vulnerabilities that usually result in either process crashes, kernel panics, or hangs, such as null pointer dereferences, divide by zeros (CVE-2011-1012 and CVE-2010-4165), infinite loops (CVE-2011-1083 and CVE-2010-1086), deadlocks (CVE-2010-4161), and data races (CVE-2010-4526 and CVE-2010-4248).

One observation from the Figure 1 is that buffer and integer overflows are the top threats to the kernel's integrity: 78% of memory corruption exploits are caused by these two vulnerabilities. This observation is consistent with the report by Christey and Martin [8].

Figure 2 shows the distribution of the vulnerabilities in the Linux kernel source code tree, namely, the code statically linked into the kernel image ("core"), device drivers (`drivers`), network protocols (`net`), file systems (`fs`), and the sound subsystem (`sound`). We observe that a non-trivial portion ($1/3$) of vulnerabilities are located in the core kernel, while $2/3$ are in loadable kernel modules. Less than 20% of vulnerabilities that we examined are in device drivers.

3. STATE-OF-THE-ART PREVENTION

We examined several state-of-the-art kernel security tools to see how many vulnerabilities they can prevent. Figure 3 summarizes the results. The rest of the section discusses each tool and the vulnerabilities or attacks that it can prevent. Our examination mainly focuses on tools that target the OS kernel, and is based on our understanding of these techniques.

3.1 Runtime tools

Software fault isolation. BGI [6] is a tool to isolate kernel modules with support for controlled sharing between kernel and modules. BGI provides a memory access control list (ACL) for each module. Programmers using BGI set the ACLs, granting or revoking a module's privileges as it invokes various functions in the core kernel (e.g., granting access when allocating memory, and revoking access when freeing memory). In this way, BGI can prevent a vulnerable module from overwriting kernel memory that it shouldn't have access to, such as double-free bugs and some buffer overflows, but allow access to kernel memory that it should have access to.

A major shortcoming of BGI is that it handles only vulnerabilities inside a module, *and* certain attacks that attempt to cross the boundary of the buggy module. As Figure 2 shows, $1/3$ of all vulnerabilities are in the core kernel, and would not be handled by BGI. Moreover, some exploits occur entirely *within* a module, and thus would not be handled by BGI either. For example, if there is a buffer overflow vulnerability in a file system module, the attacker could tamper with the module's internal data, trick the module into executing code that only applies to `setuid` binaries, and gain root privilege. Isolating modules from one another would not solve this problem.

Code integrity. SecVisor [22] enforces code integrity for the kernel. A thin hypervisor layer authenticates all code before that code is allowed to execute in kernel mode. SecVisor is effective at preventing code injection attacks. However, although it may defend against common exploits that are unaware of SecVisor's existence, persistent adversaries could still mount attacks by corrupting important kernel data, or by hijacking control flow to existing kernel code. It has been shown that it is possible to use static analysis to combine existing code sequences to perform arbitrary computation [23]. Thus, SecVisor makes exploits harder to write, but does not prevent an adversary from exploiting vulnerabilities, which is why we list zeros in the "SecVisor" column in Figure 3.

User-level drivers. SUD [5] runs device drivers at user level and prevents vulnerabilities in the driver from affecting the rest of the kernel. This turns most vulnerabilities in the driver into a denial-of-service attack that crashes the driver itself; a separate recovery mechanism, such as shadow drivers [24], is needed to mitigate the DoS impact. Note, however, that only a small fraction of kernel security vulnerabilities come from device drivers (20% in Figure 2) and that user-level drivers don't address the other vulnerabilities.

Memory tagging. Memory tagging systems, such as Raksha [10], can detect when kernel code misuses untrusted inputs (from user processes or from the network), preventing an adversary from mounting code injection attacks or otherwise taking over control flow. As with SecVisor, this prevents certain types of exploits that rely on taking over kernel control flow, but doesn't address many other vulnerabilities in Figure 2.

Uninitialized memory tracking. Two systems specifically address the problem of kernel code leaking sensitive data through uninitialized memory. Kmemcheck [20] is a runtime tool in the Linux kernel that detects uninitialized memory vulnerabilities for a given workload. Kmemcheck tracks initialization status of each memory byte, with the help of the kernel memory allocator. Kmemcheck cannot detect reading of uninitialized data from the stack. Secure deallocation [17] (`SD`), on the other hand, periodically zeros out the kernel stack to reduce information leaks from uninitialized stack variables, but does not address dynamically-allocated objects. Neither of these two tools can guarantee that they find and prevent all information disclosure bugs, although they make the bugs more difficult to exploit.

3.2 Compile-time tools

In principle, code analysis tools can pinpoint vulnerabilities so that they can be fixed once and for all by developers, and ideally prove the absence of any vulnerabilities in code. Many such tools have been used to find and fix a wide range of security problems in the Linux kernel [1, 2, 4, 13, 21]. One limitation of most static analysis tools is the large number of false positives. Thus, since it is not productive for programmers to filter out these false positives and fix all real vulnerabilities, almost no static analysis tool can prove the absence of vulnerabilities of any type in the Linux kernel, and the tools are largely used for bug-finding. To reduce the number of false positives, many static analysis tools require programmers to supply annotations [1, 2].

One specific class of vulnerabilities that seem to be difficult to detect using static analysis are the semantic vulnerabilities, such as missing permission check bugs. On the other hand, several tools have been effective at finding potential null dereference, buffer overflow, deadlock, and infinite loop bugs [1, 2, 9, 11, 26].

4. OPEN PROBLEMS

The examination in the previous section suggests there are several unaddressed challenges facing researchers in dealing with vulnerabilities found in the Linux kernel. First, vulnerabilities are present in almost all parts of the kernel, including device drivers, kernel modules, and core kernel code. Solutions that focus on only one part of the kernel, such as kernel modules, are insufficient by themselves.

Second, many runtime tools focus on preventing a certain class of *exploits*, as opposed to preventing *any* exploit of a certain class of vulnerabilities. The difference is important, since an adversary that knows of a given vulnerability is free to choose any exploit that will work on the target system. Thus, defense mechanisms that still allow a vulnerability to be exploited in some way are of limited use.

Third, many vulnerabilities continue to stem from the fact that Linux is written in an unsafe language. While it might be a good idea to write any new kernel in a type-safe language, we must contend with C, if we want to handle existing Linux code. Similarly, alternative kernel designs, such as HiStar [27] or Minix [16], can significantly reduce the amount of trusted code in the kernel, but these techniques cannot be incrementally applied to Linux. Systems like Overshadow [7] and Proxos [25] can also avoid trusting the entire Linux kernel, for applications that require only a subset of the kernel's functionality, but this approach has not yet been shown to work for all applications. How to incrementally provide type safety and module isolation for large monolithic kernels written in an unsafe language remains an open topic.

Although these challenges are unaddressed, others have identified them too. The rest of this section expands on several challenges that have received less attention.

4.1 Semantic vulnerabilities

Figure 3 shows that none of previous research addresses "missing permission checks" vulnerabilities. Unfortunately, these vulnerabilities can easily be exploited to gain privilege. Figure 4 shows a patch to CVE-2010-2071, in which the `btrfs` module forgets to check the owner of a file when setting file permissions. Thus, an adversary can gain write access to any file in the volume. Fur-

Vulnerability	BGI	SecVisor	SUD	Raksha	kmemcheck	SD
Missing pointer check	0	0	3	0	0	0
Missing permission check	0	0	1	0	0	0
Buffer overflow	1	0	1	0	0	0
Integer overflow (D)	0	0	1	0	0	0
Integer overflow (I)	0	0	1	0	0	0
Integer overflow (E)	0	0	3	0	0	0
Uninitialized data	0	0	13	0	1	23
Null dereference	11	0	3	0	0	0
Divide by zero	2	0	0	0	0	0
Infinite loop	0	0	1	0	0	0
Data race / deadlock	0	0	1	0	0	0
Memory mismanagement	1	0	1	0	0	0
Miscellaneous	0	0	0	0	0	0

Figure 3: Number of vulnerabilities that existing runtime tools can prevent. For "integer overflow", "D" means that the vulnerability can only lead to DoS attack, "I" means that the vulnerability can only lead to information disclosure, and "E" means that the vulnerability can lead to root privilege escalation, thus any other type of attacks. If a tool can prevent "integer overflow (E)", then for the same vulnerability, it is not credited for "integer overflow (I)" or "integer overflow (D)".

```
--- a/fs/btrfs/acl.c
+++ b/fs/btrfs/acl.c
@@ -160,3 +160,6 @@ static int btrfs_xattr_acl_set(...
        int ret;
        struct posix_acl *acl = NULL;

+       if (!is_owner_or_cap(dentry->d_inode))
+               return -EPERM;
+
```

Figure 4: Patch for CVE-2010-2071 in `btrfs`.

thermore, the adversary can obtain root access by replacing legal `setuid` executables with malicious ones. Ironically, an identical vulnerability (CVE-2010-1641) was discovered in the `gfs2` module 15 days earlier. Similar vulnerabilities exist in other file systems as well. For example, in CVE-2010-2066, the `ext4` module allows local users to overwrite an append-only file. An attacker can exploit this vulnerability to tamper with audit logs.

In another two semantic vulnerabilities, the kernel permissively exports a sensitive interface to all users, allowing unprivileged users to alter crucial system state. In CVE-2010-1146, for example, the `reiserfs` driver does not prevent the `.reiserfs_priv` directory from being opened, which contains meta-data that should be private to the file system. As a result, unprivileged users could tamper with the exposed meta-data and directly modify ACLs belonging to other users, including root, or set the `CAP_SETUID` extended attribute on a malicious executable to gain privilege.

Another example is CVE-2010-4347, in which the `acpi` module creates a `custom_method` file in `debugfs` with writable permission for all users. The file exports an interface that allows users to define custom methods for the ACPI module to call. Therefore, an unprivileged user could modify the kernel's control flow by writing a specially-crafted value to this file, and gain root privileges.

These vulnerabilities are difficult to check for and defend against, because the security policies are not explicitly stated, and it is usually the programmer's responsibility to enforce them by manually inserting the right checks in each code path. In contrast, the policy for enforcing memory or type safety can often be inferred from the code in a semi-automated fashion.

One possible approach for dealing with these vulnerabilities is based on the observation that there are common interfaces where high-level policies can be checked, or where the policies can be inferred from. For example, in Linux, many file systems implement the VFS interface, and each file system internally is supposed to perform the same set of permission checks (e.g., the permissions on a file can be changed only by the file's owner or by root). A kernel developer could annotate the VFS interface with these high-level policies, and use some type of analysis to check whether every implementing file system performs sufficient checks. Alternatively, an automated analysis could try to infer the set of policy checks needed, by comparing many file system implementations to each other, similar to the idea of detecting bugs as deviant behavior [14].

Another possible approach to dealing with these bugs may be to reason about privilege separation in terms of user or application principals, instead of kernel module principals (as most current kernel isolation systems have done [5, 6, 15, 16]). While user boundaries may not correspond to clean boundaries in kernel code, if one can enforce isolation in terms of user privileges, one may not need to reason about vulnerable modules, as in Loki [28].

4.2 Denial-of-service vulnerabilities

Few previous pieces of work address vulnerabilities that lead to only denial-of-service (DoS) attacks, possibly because DoS attacks do not harm the data integrity of the kernel. However, as pointed out by an earlier study [3], any kernel weakness can turn out to be a security threat. For example, the `econet` privilege escalation exploit involved three separate vulnerabilities that were not originally considered to be security-critical [12], including one DoS vulnerability.

Dealing with denial-of-service vulnerabilities in kernel code is difficult, because it requires not only detecting and preventing an adversary from crashing the kernel, but also requires continuing the kernel's execution to properly perform operations on behalf of other users. This can be particularly difficult if the denial-of-service vulnerability is triggered while a kernel thread is holding locks, or otherwise cannot be terminated cleanly without violating kernel invariants. One approach to ensuring the kernel continues to operate despite denial-of-service vulnerabilities is to use shadow drivers [24] or recovery domains [18], but no general-purpose techniques for ensuring availability despite these vulnerabilities are available.

5. SUMMARY

This paper's study of Linux kernel vulnerabilities suggests that we have a long way to go in making existing OS kernels secure. First, from January 2010 to March 2011, 141 vulnerabilities in the Linux kernel were discovered, many of which have serious exploits. Second, state-of-the-art defense techniques address only a small subset of them. Third, some of the unaddressed vulnerabilities, such as semantic bugs, pose challenging research problems.

Acknowledgments

We thank the anonymous reviewers for their feedback. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089. The opinions in this paper don't necessarily represent DARPA or official US policy.

References

[1] Smatch. `http://smatch.sourceforge.net/`.

[2] Sparse. `http://sparse.wiki.kernel.org/`.

[3] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg. Security impact ratings considered harmful. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, Switzerland, May 2009.

[4] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy*, Berkeley, CA, May 2002.

[5] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, June 2010.

[6] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008.

[8] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. `http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf`, 2007.

[9] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proceedings of the 2006 ACM Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[10] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 19th Usenix Security Symposium*, San Jose, CA, July 2008.

[11] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[12] N. Elhage. CVE-2010-4258: Turning denial-of-service into privilege escalation. `http://blog.nelhage.com/2010/12/cve-2010-4258-from-dos-to-privesc/`, 2010.

[13] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[15] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[16] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[17] C. Jim, P. Ben, G. Tal, and R. Mendel. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the 14th Usenix Security Symposium*, Baltimore, MD, August 2005.

[18] A. Lenharth, V. S. Adve, and S. T. King. Recovery domains: an organizing principle for recoverable operating systems. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, March 2009.

[19] S. A. Mokhov, M.-A. Laverdiere, and D. Benredjem. Taxonomy of Linux kernel vulnerability solutions. *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*, 2008.

[20] V. Nossum. Getting started with kmemcheck. `http://www.mjmwired.net/kernel/Documentation/kmemcheck.txt`.

[21] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, March 2011.

[22] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[23] 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*, Alexandria, VA, October–November 2007.

[24] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[25] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[26] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *Proceedings of the 32rd ACM Symposium on Principles of Programming Languages*, Long Beach, CA, January 2005.

[27] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[28] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.