Exploiting Concurrency Vulnerabilities in System Call Wrappers

Robert N. M. Watson
Computer Laboratory
University of Cambridge
robert.watson@cl.cam.ac.uk

Abstract

System call interposition allows the kernel security model to be extended. However, when combined with current operating systems, it is open to concurrency vulnerabilities leading to privilege escalation and audit bypass. We discuss the theory and practice of system call wrapper concurrency vulnerabilities, and demonstrate exploit techniques against GSWTK, Systrace, and CerbNG.

1 Introduction

System call interposition is a kernel extension technique used to augment operating system security policies without modifying the underlying code. It is widely used in research systems and commercial anti-virus software despite research suggesting security and reliability problems. Garfinkel [14], Ghormley [15], and the author [23] describe the potential for concurrency vulnerabilities in wrapper systems. However, these discussions are cursory – the vulnerability of wrappers to concurrency attacks deserves further exploration. We investigate vulnerabilities and exploit techniques for real-world systems, demonstrating that inherent concurrency problems lead directly to exploitable vulnerabilities. We conclude that addressing these systemic vulnerabilities requires rethinking security extension architecture.

We first introduce concurrency in operating system kernels and the system call wrapper technique. We then discuss the structure of concurrency vulnerabilities, the applicability of concurrency attacks to wrappers, and practical exploit techniques. We investigate privilege escalation and audit bypass vulnerabilities in three system call interposition systems, Generic Software Wrappers Toolkit (GSWTK) [12], Systrace [20], and CerbNG [7]. Finally, we explore deployed mitigation techniques and architectural solutions to these vulnerabilities.

All experiments and measurements were performed on a 3.2 GHz Intel Xeon.

2 Kernels and Concurrency

Operating system kernels are highly concurrent programs, consuming concurrency services internally and offering them to applications. Most desktop and server systems support multiprocessing and threading, as do many embedded systems, traditional bastions of minimalism.

In the operating system kernel, hardware sources of concurrency are interrupts and multiprocessing. Kernels provide internal threading facilities to kernel subsystems (file systems, network stacks, etc) and expose concurrency to applications via processes, threading, signals, and asynchronous I/O. Application writers employ these to mask I/O latency and exploit hardware parallelism. Concurrency is a fundamental operating system feature, and must be considered carefully in any security services.

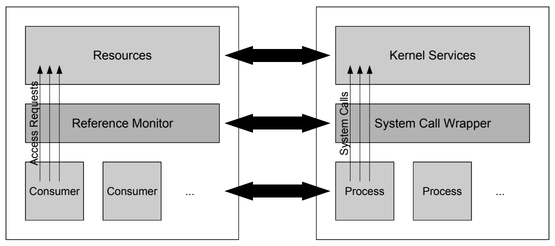

Figure 1: Misleading visual congruence of the reference monitor and system call wrapper models.

3 Wrappers for Security

System call interposition imposes a reference monitor on kernel services by intercepting system calls (Fig. 1). Anderson [4] states that a reference monitor must be tamper proof, always invoked, and small enough to subject to analysis and tests to assure correctness. System call wrappers appear to meet these criteria: they run in the kernel's protection domain, are invoked in the system call path, and avoid complex modifications to the kernel.

Further, wrappers see the UNIX API, allowing some wrapper packages to be portable across multiple operating systems. System call wrappers are compiled into the kernel or loaded as a module, hooking the system call trap handler (Fig. 2). We adopt terminology from GSWTK:

- The *precondition* hook occurs prior to passing control to the kernel, allowing the wrapper to inspect or substitute arguments before the kernel sees them.

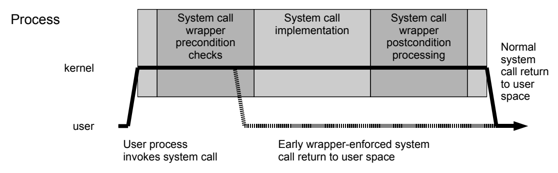
Process
kernel
user
System call wrapper precondition checks
System call implementation
System call wrapper postcondition processing
Normal system call return to user space
User process invokes system call
Early wrapper-enforced system call return to user space

Figure 2: System call wrappers implement precondition and postcondition processing around the system call.

- The *postcondition* hook occurs prior to returning control to user space, allowing the wrapper to track and log results, transform return values, etc.

Wrappers perform policy checks with system call number and arguments, and access to kernel data structures such as process information. On access control failure, they may abort the call, transform arguments, modify credentials, log events, or cause other side effects. By substituting argument and return values, system call wrappers can change system object name spaces visible to application; e.g,, a wrapper can redirect file open requests or change the IP address bound by an application. Policy sources include compiled policies (LOMAC [11]), policy configuration languages (GSWTK, CerbNG), and even message passing to user processes (Systrace).

4 Concurrency Vulnerabilities

Concurrency vulnerabilities are present wherever improper software concurrency control can lead to a violation of security policy. They may originate from a failure to properly plan for concurrence, implementation errors in concurrency control, or a mismatch between the expectations of the implementer and run-time conditions.

Abbott et al describe a range of operating system vulnerability categories in the RISOS report [2], including concurrency vulnerabilities resulting from asynchronous validation and inadequate synchronization. Bisbey and Hollingworth [5] identify the importance of consistency over time for protection information, the challenges of changes in object name binding over time, general serialization errors, and interrupted atomic operations, in which incorrect assumptions about the effects of concurrency on operations lead to vulnerability. Fithen [10] lists possible results of concurrency bugs:

- Deadlock: threads become permanently blocked.
- Loss of information: information is overwritten by another thread.
- Loss of integrity information: information written by multiple threads may be arbitrarily interlaced.
- Loss of liveness: unbalanced access to shared resources leads to performance problems.

General concurrency bugs may lead to vulnerability where they cause incorrect implementation of a desired security property. Security-related outcomes range from denial of service to violated access control properties, such as corruption or leaking of sensitive data. Protection flaws for data used in access control itself, such as credentials, permissions, and security labels, will be of greatest value to attackers. Fithen observes that most races are time-of-check, time-of-use vulnerabilities. Several such attacks are explored in past research, especially incorrect use of race-prone file APIs in UNIX [6], and application signal reentrance [25].

5 Concurrency Attacks on Wrappers

As highly-concurrent software protecting critical data, operating system kernels are fertile ground for the discovery of concurrency vulnerabilities. In contrast to the assumption of atomic system calls made in previous considerations of race conditions [8], the key to our approach is non-atomicity between the kernel and system call wrappers. We have identified and successfully exploited three forms of concurrency vulnerability in wrappers:

- Synchronization bugs in wrapper logic leading to incorrect operation, e.g., improper locking of data.
- Lack of synchronization between the wrapper and the kernel in *copying* system call arguments, such that arguments processed by the wrapper and the kernel differ. We describe these as *syntactic* race conditions, as the attacker changes literal values.
- Lack of synchronization between the wrapper and the kernel in *interpreting* system call arguments. We describe these as *semantic* race conditions, as the attacker manipulates the interpretation of values.

The latter two forms involve not a wrapper in isolation, but rather its failure in composition with the service it protects. We focus on syntactic race conditions which do not depend on kernel and wrapper internals, and are hence portable across wrapper frameworks and operating systems. We found that the most frequently identifiable and exploitable vulnerabilities fell into three categories:

- *Time-of-check-to-time-of-use* (TOCTTOU) vulnerabilities, in which access control checks are non-atomic with the operations they protect, allowing an attacker to violate access control rules.

- *Time-of-audit-to-time-of-use* (TOATTOU) vulnerabilities, in which the trail diverges from actual accesses as a result of non-atomicity, violating accuracy requirements. This allows an attacker to mask activity, avoiding IDS triggers.
- *Time-of-replacement-to-time-of-use* (TORTTOU) vulnerabilities, unique to wrappers, in which attackers modify system call arguments after a wrapper has replaced them but before the kernel has accessed them, violating the security policy.

We are not aware of prior research on the topic of TOATTOU and TORTTOU vulnerabilities.

6 Exploit Techniques

Concurrency vulnerabilities exist where there is inadequate synchronization of a shared resource leading to violation of security policy. The first step in locating concurrency vulnerabilities is to identify resources relevant to access control, audit, or other security functionality that are accessed concurrently across a trust boundary. Relevant resources include file system objects, shared memory, and sockets, as well as indirectly accessed kernel objects, such as nodes/inodes and kernel buffers. We will use process memory holding system call arguments, which is accessed by the user process, wrapper, and kernel. User memory is accessed from the kernel with copying functions, e.g. the BSD `copyin()` and `copyout()`, which validate addresses and page memory as needed.

Direct arguments are a passed as stack variables or registers, and contain values such as process IDs and pointers; they are copied in by the system call trap handler. Wrappers consume the same instance of these arguments as the kernel, so are not subject to syntactic races.

Indirect arguments are referenced by pointers, often passed as direct arguments, and copied on-demand by kernel services: for example, file paths are copied and resolved by `namei()`. Indirect arguments are copied after the precondition hook, so wrappers copy them independently from the kernel, opening a race window between the two copy operations. These races are limited to indirect arguments, many of which are security-critical, such as socket addresses, file paths, arguments to `ioctl()` and `sysctl()`, group ID lists, resource limits, and I/O data.

6.1 Concurrency Approaches

Concurrent program execution on UNIX occurs via signals, asynchronous I/O, threads, and processes. Operations in a single process necessarily support shared memory; processes may shares memory using `minherit()`, `rfork()`, and `clone()`, explicit shared memory and, debugging interfaces. We share memory across processes by inheritance, as other methods are not consistent across systems. Prior work has considered races between user processes, but we are interested in races between user threads and the kernel itself. This requires the user thread and kernel to run concurrently, which is possible through interleaved scheduling or hardware parallelism.

6.2 Racing on Uniprocessor Systems

On uniprocessor (UP) systems, the attacker must cause the kernel to yield to a user thread between execution of the system call and wrapper preconditions and postconditions. Yielding may occur voluntarily (a thread requests blocking I/O on a socket or disk) or involuntarily (a kernel thread accesses memory resulting in a page fault from disk). Both may be used to race with system call wrapper preconditions and postconditions.

Page faults on indirect system call arguments are effective in opening up race windows. The resulting wait on disk I/O provides a scheduling window of several million instruction cycles, more than enough time for an attacking thread to replace the contents of a memory page. On most systems it is easy to arrange for user memory to be paged to disk, either swap (if configured) or a memory mapped file, by increasing memory pressure.

Initially, we believed that this technique was limited to system calls with multiple indirect arguments, such as `rename()`. We were able to successfully exploit this case by paging the `rename()` target path to disk, allowing the source path to be replaced between check and use. On reflection, we realized that copy operations themselves are non-atomic, sleeping part-way through if user data spans multiple pages, allowing even system calls with a single argument to be attacked. This works well as the last byte of many indirect arguments is not essential: strings are nul-terminated and many data structures contain padding. Page faults may also be used to attack postconditions, subject to the limitation that it is possible to force a page fault on each page only once during most system calls.

Voluntary thread sleeps also prove useful: during a TCP `connect()`, the calling thread will wait in kernel for a TCP ACK to confirm the connection, allowing a user thread to execute after the arguments have been copied in by the kernel to attack on postcondition auditing.

6.3 Racing on Multiprocessor Systems

We consider any system with parallelism to be multiprocessor (MP), including SMT and multicore systems. UP systems are vulnerable to races, but require manipulating kernel scheduling via limited yield opportunities. On MP systems, races between user and kernel threads can be exploited without relying on kernel sleeping, as user threads may run simultaneously other CPUs. Inter-CPU races are narrower, as they occur at memory speed (10K-100K cycles) rather than disk or network speed (>1M cycles).

We use two approaches to identify timing for argument replacement. In the case of argument copies without replacement, a binary search for the Time Stamp Counter (TSC) length of the race window can be performed by inspecting the results of the system call being raced with. In the case of argument copies with replacement by the wrapper, it is possible to simply spin watching for the replacement, then modify the argument.

We found that race window length varied based across wrapper systems. Races on arguments in GSWTK, which runs only in kernel, were often between 5K and 15K cycles. Systrace passes control to a user process, which performs optional copies in and out of the target process, opening race windows of over 100K. The order of magnitude difference in race window size did not, however, lead to measurable differences in attack cost: we had a 100% success rate in exploiting races across packages.

7 Exercising Real Vulnerabilities

7.1 Generic Software Wrappers Toolkit

GSWTK is a kernel access control system that allows task-specific system call wrappers to inspect and modify arguments and return values. Wrappers are written using a C language extension with integrated SQL database support. GSWTK is available as a third party package on the Solaris, FreeBSD, BSD/OS, and Linux platforms; we employed GSWTK 1.6.3 on FreeBSD 4.11. A variety of wrappers are available, from access control policies to intrusion detection systems.

We were able to successfully substitute values used in both precondition access control and postcondition auditing and intrusion detection on UP with paging (Fig. 3), and on MP systems from a second processor. After experimentally validating the approach on a subset of wrappers, we inspected the remaining wrappers shipped with GSWTK. Of 23 wrappers available for UNIX or all platforms, 16 had one ore more vulnerabilities (Table 1). Also of interest is Ko's work on sequence-based intrusion detection [18], as it illustrates the potential impact of TOATTOU vulnerabilities. Investigation revealed vulnerabilities in several intrusion detection wrappers.

7.2 Systrace

Systrace is an access control system that allows user processes to control target processes by inspecting and modifying system call arguments and return values. Systrace

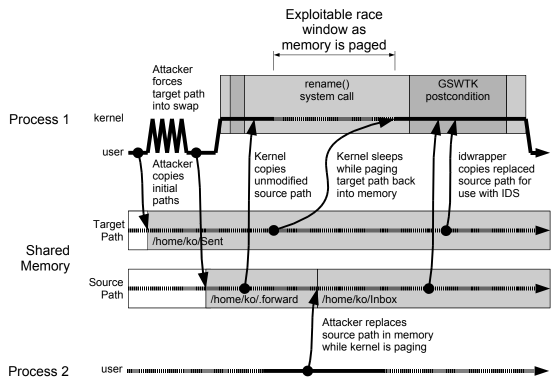
Exploitable race window as memory is paged
Attacker forces target path into swap
rename() system call
GSWTK postcondition
Process 1
kernel
user
Attacker copies initial paths
Kernel copies unmodified source path
Kernel sleeps while paging target path back into memory
idwrapper copies replaced source path for use with IDS
Target Path
Shared Memory
/home/ko/Sent
Source Path
/home/ko/.forward
/home/ko/Inbox
Attacker replaces source path in memory while kernel is paging
Process 2
user

Figure 3: Processes employ paging to force `copyin()` in `rename()` to sleep so that the process can use a TOATTOU attack on an intrusion detection wrapper.

ships in the OpenBSD and NetBSD operating systems, with ports to Mac OS X, FreeBSD, and Linux. For this work, we used Systrace on NetBSD 3.1, 4.0 (Jan. 2007), and OpenBSD 4.0. As Systrace is a programmable policy system, we used two policies: Sudo monitor mode [19] and Sysjail [16]. We bypassed protections in both, violating access control policy and audit trail integrity.

7.2.1 Sudo

Sudo [19] is a widely used privilege management tool allowing users to run authorized commands with the rights of another user. The prerelease version of Sudo includes a monitor mode based on Systrace that audits commands executed by Sudo-derived processes. Sudo intercepts `execve()`, which accepts a program path, command line arguments, and environmental variables as indirect arguments, and thus vulnerable to attack. Due to a user space policy source, Systrace requires additional context switches to make access control decisions, leading to larger race windows. With Sudo on MP systems, the window for `execve()` arguments was over 430K cycles. We were able to successfully exploit this vulnerability, replacing command lines so that they were incorrectly logged, masking all further attacker activity in the audit trail.

7.2.2 Sysjail

Sysjail [16] is port of the FreeBSD jail containment facility [17] using the Systrace framework for NetBSD and OpenBSD. Sysjail attaches to all processes in the jail, validating and in some cases rewriting system call arguments to maintain confinement. Sysjail is of particular interest as it is intended to contain processes running with root privilege, increasing exposure in the event of vulnerability.

Wrapper	Description	Vulnerabilities
`callcount`	Count system calls	None: relies on the system call number.
`conwatch`	Track IP connections by processes.	Postcondition TOATTOU race on `connect()` and `bind()` allows masking address/port used.
`dbfencrypt`	Encrypt files with '$' in their names; prevent rename so that encryption policy on a file cannot be changed.	Postcondition TOCTTOU race allows incorrect name in policy check; precondition TORTTOU races on I/O write unencrypted data and bypass rename checks.
`dbexec`	Authorize execution of programs based on a pathname database.	Precondition TOCTTOU race allows bypass by substituting the name during the wrapper check.
`dbsynthetic`	Synthetic file system sandbox substituting pathnames from a database.	Precondition TORTTOU race bypasses path replacement; postcondition TORTTOU race leaks true paths
`life`	Prints the process life cycle.	Precondition TOATTOU race replaces `exec()` paths.
`noadmin`	Deny all privileged operations.	None: relies on the kernel's process credential.
`aks.wr`	Audit file operations	Pre/postcondition TOATTOU races avoid audit.
`seq-kernel.wr`	Sequence-based intrusion detection	None: relies on the system call number.
`imapd.wr`	Detect anomalous access by `imapd`.	Postcondition TOATTOU path races prevent alerts.

Table 1: Selection of concurrency vulnerabilities in GSWTK and ID Wrappers

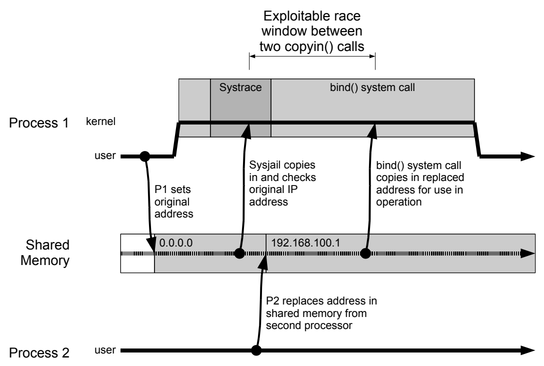
Exploitable race window between two copyin() calls
Systrace
bind() system call
Process 1
kernel
user
P1 sets original address
Sysjail copies in and checks original IP address
bind() system call copies in replaced address for use in operation
Shared Memory
0.0.0.0
192.168.100.1
P2 replaces address in shared memory from second processor
Process 2
user

Figure 4: Race to bypass protections from a second processor by replacing the IP address between check and use.

Sysjail handles several indirect arguments, including IP addresses passed to `bind()`. It enforces two constraints: the address must be configured for the jail or it must be `INADDR_ANY`, in which case it will be replaced with the jail's address. By racing with the Sysjail, we are able to replace the IP accepted by Sysjail with another IP address, bypassing network confinement (Fig. 4).

7.3 CerbNG

CerbNG is a third-party security framework for FreeBSD 4.8 similar to GSWTK. It allows rule-based control of system calls, checking and modifying arguments and return values, changing process properties, and logging events. We successfully exploited TOATTOU and TOCTTOU races in rules shipped with the system, replacing command lines in `log-exec.cb`, which audits `execve()`, generating incorrect audit trails. It employs several virtual memory defenses discussed in Section 8.1.

8 Preventing Wrapper Races?

System call wrapper races can lead to partial or complete bypass of access control and audit. To address this, concurrency must be properly managed. We consider proposed solutions in three areas: those that retain the wrapper architecture but modify wrapper systems to mitigate attacks, those that retain the wrapper architecture but modify to the OS kernel, and those that entirely abandon the wrapper approach in extending kernel security.

8.1 Mitigation Techniques

Lee Badger has suggested a weak consistency approach: detect and mitigate exploitative changes in kernel state via a postcondition, taking remedial action if a violation has occurred. We believe that this approach faces challenges from the complex side effects of some system calls (e.g., `connect()` and `unlink()`); detecting inconsistency faces the same atomicity issues as other postconditions.

Dawidek has experimented with marking memory pages holding system call arguments read-only during system calls. If implemented properly, this prevents argument races, but violates concurrent programming assumptions. Legitimate multithreaded processes may store concurrently accessed data in the same memory page as arguments, and will suffer ill effects such as unexpected faults.

VM protection is non-trivial as all mappings of a physical page must be protected. One interesting case involves memory-mapped files: systems with unified VM/buffer caches must prevent writes via I/O system calls, not just mapped memory. Protecting pages is also insufficient: the address space must be protected to prevent the unmapping of protected pages and remapping with writable ones. We found several vulnerabilities in CerbNG's VM protections, including incorrect write protection of pages, and race windows while copying arguments.

Provos provides similar facilities in Systrace, copying indirect arguments into the "stack gap", a reserved area of process memory, allowing wrappers to substitute indirect arguments of greater size than the original argument. He has also suggested that this may be used to resist shared memory attacks as the stack gap area is unique to each process. This protection is not effective with threads, as threads share a single address space. Experimentation on OpenBSD indicates that the stack gap mapping can be replaced with shared memory accessible to other processes even in the non-threaded case. This approach causes additional data copies for any protected arguments.

Ghormley addresses argument races in the SLIC interposition framework via in-kernel buffers that extend the user address space. Each thread caches regions of the address space copied by the extension or kernel; future accesses will be from the cache, preventing further modification by user threads. This approach requires replacing the kernel copy routines so that the kernel, not just wrappers, use the cache. As cache buffers are not forced to page size, the false sharing effects of page protections are avoided; however, this approach imposes a significant performance penalty, as all indirect arguments must be copied and cached in kernel memory.

VM and caching schemes make processing indirect arguments that are read and written in a single system call (such as POSIX asynchronous I/O) more tricky. None of the systems with protections were able to handle this case correctly, although this had limited impact as none of the sample policies controlled affected system calls.

These mitigation techniques suffer serious correctness and performance problems. VM and caching protect only against syntactic vulnerability, as they prevent the attacker from replacing arguments and do not synchronize with kernel services. Fundamentally, system call wrappers are not architecturally well-placed to synchronize with the kernel, as this conflicts with clean separation from the kernel.

8.2 Message Passing Systems

In order to maintain the system call interposition model without resorting to mitigation techniques, kernel operation must be changed. One possibility is to move to a message-passing model, in which system call arguments are bundled and delivered to the kernel at once rather than being copied on-demand. This approach would not eliminate semantic race conditions, but would eliminate syntactic race conditions by allowing wrappers to inspect the same argument values as the kernel. The disadvantage to this model is that it requires the complete layout of arguments to be available to the trap handler; currently, this knowledge is distributed across many layers of the kernel. Garfinkel's Ostia [13] and Seaborn's Plash [21] both implement message-passing approaches in which access to the file system name space must occur "by proxy" via a monitor process, avoiding argument and name space copying races, but allowing further accesses occur directly using a passed file descriptor. VM mitigation schemes may be gradually extended to approximate the message passing paradigm, although provide less clean implementations than systems designed with message passing in mind, such as Mach [3].

8.3 Integrating Security and Concurrency

A more flexible, if more complex approach, is to eliminate race conditions between security extensions and the kernel by integrating security checks with the kernel itself. Invocations of security extensions occur throughout the kernel, atomically with respect to use of the object they control. For example, access control checks on a process operation would be performed while holding locks on the process to prevent changes in associated context.

As system call interposition was developed to avoid OS modification, this may seem contradictory; however, the move to open source systems and the adoption of security extensions has driven the creation of security frameworks by vendors. The approach has been adopted by FLASK in SELinux, SEBSD, and SEDarwin [22], the TrustedBSD MAC Framework in FreeBSD and Mac OS X [23], `kauth` in NetBSD and Mac OS X [1, 9], and Linux Security Modules [24]. The degree of integration varies across systems: at one extreme, the TrustedBSD MAC Framework asserts object locks at each entry from the kernel, allowing policies to rely on kernel locks to protect associated access control checks. At the other, the `kauth` framework allows upcalls to a user process, which precludes holding some locks over checks.

Integrated kernel security frameworks do not eliminate the problem of concurrency vulnerabilities entirely but they do provide tools to avoid race conditions innate to the system call interposition approach.

9 Conclusion

In this paper, we have explored concurrency vulnerabilities in system call interposition security extensions, arguing that correctness with respect to concurrency is not only important in preventing inconvenient deadlocks, but also critical to access control and audit. We have demonstrated that several wrapper systems suffer from common classes of concurrency vulnerability allowing privilege escalation and bypass of intrusion detection. These vulnerabilities derive from the fundamental architectural separation of the wrapper and native kernel synchronization strategies – the same structural separation that leads to a deceptively appealing similarity to an idealized reference monitor. We have also demonstrated that many deployed mitigation solutions suffer from vulnerabilities, as well as semantic or performance degradation, and that architectural solutions require much tighter integration of security with the kernel.

References

[1] Kernel Authorization. Technical Note TN2127, Apple Computer, Inc., 2007. `http://developer.apple.com/technotes/tn2005/tn2127.html`.

[2] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security Analysis and Enhancements of Computer Operating Systems. Technical Report NBSIR 76-1041, National Bureau of Standards, April 1976.

[3] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. Technical report, August 1986.

[4] James P. Anderson. Computer Security Technology Planning Study. Technical report, Electronic Systems Division, Air Force Systems Command, Hanscom Field, Bedford, MA 01730, October 1972.

[5] Richard Bisbey and Dennis Hollingworth. Protection Analysis: Final Report. Technical Report ISI/SR-78-13, Information Sciences Institute, University of Southern California, May 1978.

[6] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[7] Pawel Jakub Dawidek and Slawomir Zak. CerbNG: system firewall mechanism, 2007. `http://cerber.sourceforge.net/`.

[8] Drew Dean and Alan J. Hu. Fixing Races for Fun and Profit: How to use `access(2)`. In *Proc. 13th USENIX Security Symposium*, August 2004.

[9] Elad Efrat. kauth: kernel authorization framework. In *NetBSD Kernel Developer's Manual*. January 2007. `http://www.netbsd.org/~elad/recent/man/kauth.9.html`.

[10] William L. Fithen. Follow the Rules Regarding Concurrency Management. *BuildSecurityIn*, October 2005. `https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/guidelines/332.html`.

[11] Timothy Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proc. 2000 IEEE Symposium on Security and Privacy*, 2000.

[12] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proc. 1999 IEEE Symposium on Security and Privacy*, May 1999.

[13] T. Garfinkel, B. Pfa, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Internet Society 2003*, 2003.

[14] Tal Garfinkel. Traps and Pitfalls: Practical Problems in in System Call Interposition Based Security Tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[15] Douglas P. Ghormley, David Patrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proc. USENIX Annual Technical Conference (NO 98)*, June 1998.

[16] Kristaps Johnson and Maikls Deksters. sysjail: systrace userland virtualization, 2007. `http://sysjail.bsd.lv/`.

[17] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd International SANE Conference*, 2000.

[18] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and Countering System Intrusions Using Software Wrappers. In *Proc. 9th Usenix Security Symposium*, August 2000.

[19] Todd C. Miller. Sudo, 2007. `http://www.gratisoft.us/sudo/`.

[20] Niels Provos. Improving Host Security with System Call Policies. In *Proc. 12th USENIX Security Symposium*, Washington, DC, August 2003.

[21] Mark Seaborn. Plash: tools for practical least privilege, 2007. `http://plash.beasts.org/`.

[22] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Anderson, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. 8th USENIX Security Symposium*, August 1999.

[23] Robert Watson, Brian Feldman, Adam Migus, and Chris Vance. Design and Implementation of the TrustedBSD MAC Framework. In *Proc. Third DARPA Information Survivability Conference and Exhibition (DISCEX), IEEE*, April 2003.

[24] Chris Wright, Crispan Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proc. 11th USENIX Security Symposium*, August 2002.

[25] Michal Zalewski. Delivering Signals for Fun and Profit: Understanding, exploiting and preventing signal-handling related vulnerabilities. 2001. `http://www.bindview.com/Services/Razor/Papers/2001/signals.cfm`.