Make Least Privilege a Right (Not a Privilege)

Maxwell Krohn*, Petros Efstathopoulos†, Cliff Frey*, Frans Kaashoek*, Eddie Kohler†,
David Mazières‡, Robert Morris*, Michelle Osborne‡, Steve VanDeBogart† and David Ziegler*
*MIT †UCLA ‡NYU
asbestos@scs.cs.nyu.edu

ABSTRACT

Though system security would benefit if programmers routinely followed the *principle of least privilege* [24], the interfaces exposed by operating systems often stand in the way. We investigate why modern OSes thwart secure programming practices and propose solutions.

1 INTRODUCTION

Though many software developers simultaneously revere and ignore the principles of their craft, they reserve special sanctimony for the *principle of least privilege*, or *POLP* [24]. All programmers agree in theory: an application should have the minimal privilege needed to perform its task. At the very least, five *POLP requirements* must be followed: (1) split applications into smaller protection domains, or "compartments"; (2) assign exactly the right privileges to each compartment; (3) engineer communication channels between the compartments; (4) ensure that, save for intended communication, the compartments remain isolated from one another; and (5) make it easy for anyone to audit the intended separation.

Unfortunately, modern operating systems make these requirements onerous, dangerous, or impossible to apply. In our experience (detailed in Section 2.2), building least-privileged software is cumbersome and labor-intensive: following POLP feels more like an abuse of the operating system's interface than a judicious use of its features. Most programmers spare themselves these difficulties by reverting to monolithic, over-privileged application designs. Such neglect exposes machines to attacks both old and new, from remote attacks on privileged servers to "install attacks" (exploiting users' willingness to run high-privilege installers to infect machines with malware). We cannot write bug-free applications or prevent honest users from occasionally executing malicious code. Instead, our best hope is to contain the damage of evil code by resurrecting POLP.

In this paper, we examine some ways that current OSes discourage development of least-privilege applications (Section 2), then propose OS design ideas that might encourage it instead. A first approximation of a POLP-friendly system is one based on *capabilities*, discussed in Section 3. Though capabilities have historically flummoxed application designers, we present a more familiar interface, based on the Unix file system. In Section 4, we discuss shortcomings in this proposed design: system weaknesses might still allow vulnerabilities to spread, and process-sized compartments are too coarse-grained. We then propose a solution based on *decentralized mandatory access control* [17]. The end result is a new operating system called *Asbestos*.

2 LESSONS FROM CURRENT SYSTEMS

Administrators and programmers can achieve POLP by pushing the features in modern Unix-like operating systems, but only partially, and with important practical drawbacks.

2.1 `chroot`ing or `jail`ing Greedy Applications

Because Unix grants privilege with coarse granularity, many Unix applications acquire more privileges than they require. These "greedy applications" can be tamed with the `chroot` or `jail` system calls. Both calls confine applications to *jails*, areas of the file system that administrators can configure to exclude setuid executables and sensitive files. FreeBSD's `jail` goes further, restricting a process's use of the network and interprocess communication (IPC). System administrators with enough patience and expertise can `chroot` or `jail` standard servers such as Apache [1], BIND [3] and sendmail [26], though the process resembles stuffing an elephant into a taxicab.

Even when possible, the `chroot`/`jail` approach faces more fundamental drawbacks:

Jails are heavyweight. The jailed file system must contain copies of system-wide configuration files (such as `resolv.conf`), shared libraries, the run-time linker, helper executable files, and so on. Maintaining collections of duplicated files is an administrative difficulty, especially on systems with many jailed applications.

Jails are coarse-grained. Running a process in a jail is similar to running it on its own virtual machine. Two jailed applications can share files only if one's namespace is a superset of the other, or if inefficient workarounds are used, such as NFS-mounting a local file system.

Jails require privilege. Unprivileged users may not call `chroot` or `jail`.[1] Jails are therefore ill-suited for containing the many untrusted applications that should not have privileges, such as executable email attachments or browser plugins.

Finally, `chroot` or `jail`'s *ex post facto* imposition of security is no substitute for POLP-based design. For example, a typical dynamic content Web server (such as Apache with PHP [18]) runs many logically unrelated scripts within the same address space. A vulnerability in any one script exposes all other scripts to attack, regardless of whether the server is jailed.

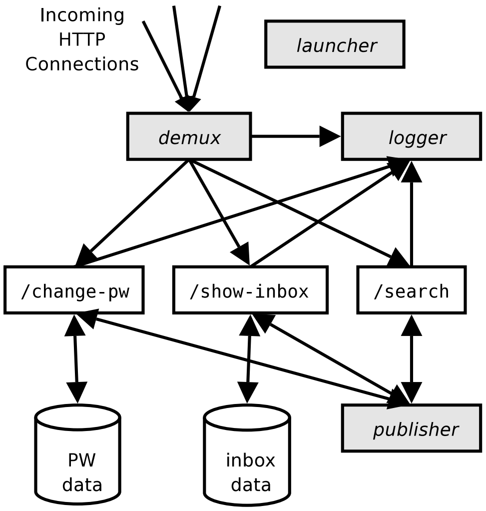

Figure 1: Block diagram of the OKWS system. Standard processes are shaded, while site-specific services and databases are shown in white. The privileged *launcher* process launches the *demux*, *publisher*, *logger* and the site-specific services. The databases shown might either be running locally, or on different machines.

2.2 Ad-Hoc Privilege Separation

True privilege separation is possible on Unix through a collection of ad-hoc techniques. For instance, our POLP-based OK Web Server (OKWS) [12] uses a pool of worker processes to sequester each logical function of the site (e.g. `/show-inbox`, `/change-pw`, and `/search`) into its own address space. The *demux*, a small, unprivileged process, accepts incoming HTTP requests, analyzes their first lines, and forwards them to the appropriate workers using file descriptor passing. Workers then respond to clients directly. A privileged *launcher* process starts this process suite, ensuring that processes are jailed into empty subtrees of the file system, and that they do not have the privileges to interact with one another. Finally, since workers' `chroot` environments prohibit them from accessing the root file system directly, they write HTTP log entries and read static HTML content via small, unprivileged helper processes: the *logger* and the *publisher*, respectively. Figure 1 shows a block diagram of a simple OKWS configuration.

The goal of this design is to separate application logic into disjoint compartments, so that any local vulnerability (especially in site-specific worker processes) cannot spread. In particular, workers cannot send each other signals or trace each other's system calls, they cannot access each other's databases, they cannot alter any executable or library, and they cannot access each other's coredumps. Unfortunately, achieving these natural requirements complicates OKWS. Its launcher must:

1. Establish a `chroot` environment, with the correct file system permissions, that contains the appropriate shared libraries, configuration files, run-time linker, and worker executables.
2. Obtain unused UID and GID ranges on the system.
3. Assign the *i*th worker its own UID u_i and GID g_i.
4. Allocate a writable coredump directory for each UID.
5. Change the *i*th worker's executable to have owner `root`, group g_i, and access mode `0410`.
6. Call `chroot`.
7. For each worker process i: kill all processes running as user u_i or group ID g_i; fork; change user ID to u_i and group ID to g_i; `chdir` into the dedicated dump directory; and call `exec` on the correct executable.

The `chown` call in Step 5, the `chroot` call in Step 6, and the `setuid` call in Step 7 all require privileged system access, so the launcher must run as root. Unix offers no guarantees of an atomic UID reservation (as required in Step 2) or race-free file system permission manipulations (as required throughout). Even ignoring these potential security problems, this design requires involved IPC to coordinate worker and helper processes.

Other systems use similar techniques to solve related problems. Examples include remote execution utilities such as OpenSSH [23] and REX [10], and mail transfer agents such as qmail [2] and postfix [21]. Considering these applications and others, a trend emerges: in each instance, the intricate mechanics of privilege separation are invented anew. To audit the exact security procedures of these applications, one must comb tens of thousands of lines of code, each time learning a new system. Even automated tools that separate privileged operations [5] require root access.

2.3 A User-Level POLP Library?

At first glance, a user-level POLP library might seem able to abstract the security-related specifics of applications like OKWS, qmail, and so on. One such example of this approach is found in the Polaris system for Windows XP [30], which applies POLP to virus-prone client applications like Web browsers and spreadsheets[2] via `chroot`-like compartments. Such solutions have three drawbacks. First, they require privileged access to the system. Second, libraries must work around the lack of good OS support for sharing across compartments: since jailed processes work with copies of files, synchronization schemes are required to reconcile copies after changes. (For example, Polaris email plug-ins run in a jail with a copy of the attachment; a persistent "synchronizer" process updates the original if the plug-in changes the copy.) Finally, we suspect that POLP techniques used in more complicated servers such as OKWS do not generalize well. As evidence, both OKWS and REX, an ssh-like login facility, use the same libraries (the SFS toolkit [16]) but share little security-related code. This comes as no surprise since the two have different security aims: OKWS hides most of the file system, while REX exposes it to authorized users; OKWS must support millions of possible users, while REX serves only those with login access to a given machine; application designers can extend OKWS with site-specific code, while REX runs unmodified. Fitting both application types into one general template seems a tall order.

2.4 Unix as a Capability System

One of the main difficulties with ad-hoc privilege separation is that starting with a privileged process and subtracting privileges is more cumbersome and error-prone than starting with a totally unprivileged process and adding privileges. Unix-like operating systems in general favor the subtractive model, while capability-based operating systems [4, 28] favor the additive one. But Unix file descriptors are in fact capabilities. By hobbling system calls sufficiently—either through system call interposition [7, 22] or small kernel modifications—we can emulate those semantics of capability-based operating systems that enable privilege separation.

The idea is to allow calls that use already-opened file descriptors (such as `read`, `write`, and `mmap`), but shut off all "sensitive" system calls, including those that create new capabilities (such as `open`), assign capabilities control of named resources (such as `bind`), and perform file system modifications, permissions changes, or IPC without capabilities (such as `chown`, `setuid`, or `ptrace`). In OKWS, the launcher could apply such a policy to the worker processes, which only require access to inherited or passed file descriptors. The launcher could run without privilege, and would no longer navigate the system call sequence seen in Section 2.2. By disabling all unneeded privileges, the operating system could enforce privilege separation by default.

This works because Unix's capability-like system calls are *virtualizable*. Processes are usually indifferent to whether a file descriptor is a regular file, a pipe to another process, or a TCP socket, since the same `read` and `write` calls work in all three cases. In practical terms, virtualization simplifies POLP-based application design. Splitting a system into multiple processes often involves substituting user-space helper applications for kernel services; for instance, OKWS services write log entries to the *logger* instead of a Unix file. With virtualizable system calls, user processes can mimic the kernel's interface; programmers need not rewrite applications when they choose to reassign the kernel's role to a process.

More important, virtualizable system calls enable *interposition*. If an untrustworthy process asks for a sensitive capability, a skeptical operator can babysit it by handing it a pipe to an interposer instead. The interposer allows harmless queries and rejects those that involve sensitive information. If the kernel API is virtualizable, then the operator need not even recompile the untrustworthy process to interpose on it.

Unfortunately, most Unix system calls resist virtualization. Some do not involve any capability-like objects; others use hard-wired capabilities hidden in the kernel, such as "current working directory" and "file system root". User-level emulation of these problematic calls—which include `open`—is messy, if not impossible; but scrapping `open` in the name of POLP seems unlikely to compel the average programmer.

3 OPERATING SYSTEM SUPPORT FOR POLP

With the lessons from Unix, we can imagine a POLP-friendly operating system interface, in which all system calls are capability-based and virtualizable like `read` and `write`. Adding universal virtualization support to a Unix-like capability system would cover all five POLP requirements. With capabilities, application programmers can split their program into isolated compartments (#1 and #4), granting each compartment only the privileges necessary to complete its task (#2). With virtualization, programmers use standard interfaces and libraries for communication between these compartments (#3), and auditors can understand this communication by interposing at the interfaces (#5). This section presents a hypothetical design for such a system, which we'll call *Unestos*.

3.1 Unestos Design

In Unestos, interactions between a process and other parts of the system take the form of *messages* sent to *devices*. Devices include processes and system services as well as hardware drivers. Messages follow the outline "perform operation O on capability C, and send any reply to capability R." The kernel forwards this message to the device that originally issued C. There are a small number of operation types, as in NFS [25] and Plan 9's 9P [19]: `LOOKUP`, `READ`, `WRITE`, and so forth. The message types and their associated syntax are conventions; the kernel only enforces or interprets those messages sent to kernel devices. Requests and replies are sent and received asynchronously.

This design aids virtualization. All of a process's interactions with the system—whether with the kernel or other user applications—take the same form, explicitly involve capabilities, and shun implicit state. Consider, for example, the Unix call `open("foo")`. This call in Unestos would translate to a message that a process P sends to the file server device FS:

$$P \rightarrow \langle C_{\mathrm{CWD}}, \text{LOOKUP}, \texttt{"foo"}, C_P \rangle \rightarrow FS.$$

The first argument is a capability C_{CWD} that identifies P's current working directory. The second is the command to perform, the third represents the arguments, and the fourth is the capability to which the file system should send its response. Since Unestos makes explicit the CWD state hidden in the Unix system call, either the file server or a user process masquerading as the file server can answer the message.

3.2 Naming and Managing Capabilities

When an Unestos process P_1 launches a child process P_2, it typically grants P_2 a number of capabilities, ranging from directories on the file system to opened network connections. How can P_2 then access these capabilities? Traditional capability systems such as EROS favor global, persistent naming, but persistence has proven cumbersome to kernel and application designers [27].

Instead, we advocate a per-process, Unix-style namespace. Under Unestos, P_1 makes capabilities available to P_2 as files in P_2's namespace. Suppose P_1's namespace contains a tree of files and directories under `/secret`, and P_1 wishes to grant P_2 access to files under `/secret/bob`. As in Plan 9 [20], P_1 can mount `/secret/bob` as the directory `/home` in P_2's namespace. Unlike in Plan 9, the state implicit in the per-process namespace is handled at user level, and the kernel only traffics in messages sent to capabilities. For example, when the process P_2 opens a file under `/home`, the user level libraries translate the directory `/home` to some capability C. The kernel sees a LOOKUP message on C.

3.3 OKWS Under Unestos

We now consider what OKWS might look like on Unestos. Similar to before, the application suite consists of a *launcher*, *demux* and worker processes. Under Unestos, the logger process simply enforces append-only access to a log file, and might be useful for many applications (much like `syslogd` on today's systems). No publisher process is needed.

The launcher starts each process with an empty namespace (and thus no capabilities), then augments their namespaces as follows:

- In the *logger*'s namespace, mounts a logfile on `/okws/log`.
- In the *demux*'s namespace, mounts TCP port 80 on `/okws/listen`. For each worker process *i*, makes a socket pair and connects one end to `/okws/worker/`*i*.
- In worker process *i*'s namespace, mounts the other end of the above socket pair to `/okws/listen`. Mounts a connection to the logger on `/okws/log`. Mounts a read-only capability to the root HTML directory on `/www`.
- In all namespaces, makes required shared libraries available under `/lib`.

The launcher then launches all processes as before.

Under Unix, the launcher had to carefully construct jails, physically copying over files and invoking custom helper applications like the publisher and logger to limit file system access. Unestos, by contrast, lets the launcher expose capabilities to child processes at arbitrary points in their namespaces. Each child receives a synthetic file system perfectly suited to its task.

Moreover, all capabilities available to the Unestos OKWS processes are virtualizable. Workers accept connections on `/okws/listen` regardless of whether they originate from the kernel's TCP stack or the *demux*. Similarly, logging might be to a raw file or through a logging process that enforces append-only behavior; worker processes are oblivious to the difference.

3.4 Discussion

So far, the proposed system features no individually novel ideas; rather, it finds a new point in the OS design space amenable to secure application construction. Similar effects might be possible with message-passing microkernels, or unwieldy system call interposition modules. But in Unestos, the security primitives are few and simple, for both the kernel and application developer. Although the interface exposed to applications feels like the familiar Unix namespace (with added flexibility for unprivileged, fine-grained jails), an application's system interactions are entirely defined by its capabilities, and Unestos behaves like a capability system for the purposes of security analysis.

4 FINE-GRAINED POLP WITH MAC

Though we believe Unestos is an improvement over the status quo, it still falls short of enabling the high-level, end-to-end security policies we seek. Applications in Unestos can only express security policies in terms of *processes*, but processes often access many different types of data on behalf of different users. A security policy based on processes alone can therefore conflate data flows that ought to be handled separately. For example, OKWS on Unestos achieves the policy that data from a `/change-pw` process cannot flow to a corrupted `/show-inbox` process; but the policy says nothing about whether user U's data within `/show-inbox` can flow to user V, meaning an attacker who compromises `/show-inbox` might be able to read an arbitrary user's private e-mail.

Of course, a much better policy for OKWS would be that "only user U can access user U's private data". We would like to separate users from one another, much as we separated services in Section 3. Though a user session involves many different processes (such as the *demux*, databases[3], and worker processes), a policy for separating users should be achievable with a few stanzas of privileged code. This section extends Unestos to a new system, *Asbestos*, whose kernel uses flexible mandatory access control primitives to enforce richer end-to-end security policies. We are currently designing and building Asbestos as a full operating system for x86 machines.

4.1 Complete Isolation

One possible approach to better isolation, which we call *complete isolation*, is to prohibit server-side processes from speaking for multiple users. The server must be prepared to run a process for every service–user pair; trusted code in *demux* would route traffic accordingly, and various isolation schemes (such as capabilities) could prevent these processes from communicating. More drastic separation is possible with virtual machines [11, 32] so that each machine can only speak for one user.

Complete isolation has several drawbacks. First, scalability is a challenge: a process for each service–user pair implies either a CPU-intensive fork-accept-exit model or a memory-intensive large server pool. Second, with no kernel support for tracking data flow, processes are completely responsible for their own access-control checks. The initial check happens at *demux*; a subsequent check is required when each per-user process accesses the database. With each additional process that speaks on behalf of multiple users comes additional access control checks. If application programmers forget or misapply any of these checks, the system can leak sensitive data to attackers.

Finally, complete isolation fails if two processes that were intended to be isolated from each other can communicate with any common third process. The system therefore implicitly trusts all running processes to refrain from enabling unintended communication.

4.2 Decentralized, Fine-Grained MAC

A more principled, and reliable approach to managing data flow is possible with mandatory access control (MAC). The Asbestos operating system proposes a decentralized, fine-grained version of MAC to solve the security problems inherent in an OKWS-like system. Similar to traditional MAC, Asbestos assigns devices on the system to *compartments*, which form a partially-orderable lattice. If device *A* sends device *B* a message, and they are in the same compartment, they remain so after delivery. If *A*'s compartment is strictly higher than *B*'s, then receiving a message from *A* pushes *B* into *A*'s compartment. If *A* and *B*'s compartments are incomparable, or *A*'s compartment is strictly less than *B*'s, then message delivery fails. With compartments, Asbestos tracks all devices that have accessed a given datum, whether directly or via proxy.

We propose two important modifications to traditional MAC-based operating systems. First, decentralization [17]: processes can create their own compartments on the fly, so that a Web server can associate each remote user with her own compartment. Second, compartments apply at the fine-grained level of individual memory pages, so that a single process can act on behalf of mutually distrustful users without fear of leaking data among them. Taken together, these two modifications allow application designers to dynamically partition a process's virtual address space into compartmentalized *subprocesses*.

Under Asbestos, OKWS behaves as follows: *demux* peaks into user *U*'s incoming TCP connection, authorizing *U* based on session state or login information in the HTTP headers. If *U* is logging on for the first time, *demux* creates a compartment for *U*; if *U* is returning, then *demux* reassigns *U* to its previous compartment. It then forwards *U*'s connection to the appropriate sub-process of the appropriate worker. When handling *U*'s request, the sub-process can access virtual memory pages and devices available to *U*'s compartment; for instance, it might access session state cached on the worker process or a database process trusted to store data for all users. If the sub-process errantly accesses data in *V*'s compartment, the read or write will fail, since *U* and *V* occupy incomparable compartments.

Once a worker has finished serving *U*, it can restore its memory and register state to a saved checkpoint, and is then safe to enter a different sub-process, and speak on behalf of a different user. Finally, since *demux* created the user compartments, it can sanction trusted *declassifiers* to traverse them. For example, it might authorize a trusted statistics collector to comb all pages in a worker's virtual address space, regardless of compartment.

5 RELATED WORK

Asbestos proposes the marriage of previous ideas in systems: the capability-based operating system [4, 13, 28, 33], the per-process name space [20], the virtualizable kernel interface (the logical extension of system-call interposition libraries [7, 22]), and decentralized MAC [17].

Naturally, other operating systems predating Asbestos meet related design goals or offer similar features. Message-based operating systems such as L4, Amoeba, V, Chorus and Spring can isolate system services by running them as independent, user-level processes and provide natural support for interposition through message-based interfaces [14]; Trusted Mach in particular views message-passing from a security perspective [6]. But ports in microkernel systems are coarse as capabilities go; for instance, a process can have a capability for the file server but not for a particular directory. For POLP, application programmers need arbitrary collections of specific capabilities; in this respect, the microkernels of yesteryear do not fit the bill.

The Flask System applies MAC to the Fluke Microkernel [29]. Many of Flask's design principles have found a modern incarnation in SELinux [15], which, like TrustedBSD [31], adds mandatory access control to popular Unix systems. In both, static policy files dictate which resources applications might access, and how processes can interact with one another. Such systems are attractive because they preserve the POSIX interface to which many programmers are accustomed. However, their policy extension model, which is based on privileged files and kernel modules, appears to fall short of the uniformly-analyzable policy extensions decentralized labels can support.

Type safety is another way to enforce operating system security. Coyotos combines capabilities with language-level verification techniques [27]. Singularity combines strong isolation with a type-safe ABI [8]. At user level, the Java Sandbox uses customizable policies to specify an applet's access rights; dynamic sandboxing shows these policies can be automatically produced [9].

6 CONCLUSION

Asbestos aims to combine decentralized MAC and capabilities to make POLP convenient, practical and effective for applications like OKWS. We have no proof that other applications would similarly benefit from Asbestos, but we are optimistic. Asbestos provides simple, flexible, and fine-grained mechanisms for achieving the five important POLP requirements without sacrificing performance.

ACKNOWLEDGMENTS

The authors thank Lee Badger, Butler Lampson, Mike Walfish and the reviewers. This work was supported by DARPA grants MDA972-03-P-0015 and FA8750-04-1-0090, and by joint NSF Cybertrust/DARPA grant CNS-0430425. David Mazières and Robert Morris are supported by Sloan fellowships.

REFERENCES

[1] The Apache Software Foundation. Apache. `http://www.apache.org`.

[2] D. J. Bernstein. qmail. `http://cr.yp.to/qmail.html`.

[3] Internet Systems Consortium. Berkeley Internet Name Daemon. `http://www.isc.org/sw/bind`.

[4] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *USENIX Workshop on Microkernels and Other Kernel Architectures*. USENIX, 1992.

[5] D. Brumley and D. X. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72. USENIX, 2004.

[6] T. Fine and S. E. Minear. Assuring distributed trusted mach. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, page 206, Washington, DC, USA, 1993. IEEE Computer Society.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[8] G. C. Hunt and J. R. Larus. Singulairty design motivation. Technical Report MSR-TR-2004-105, Microsoft Corporation, Dec. 2004.

[9] H. Inoue and S. Forrest. Anomaly intrusion detection in dynamic execution environments. In *NSPW '02: Proceedings of the 2002 workshop on New security paradigms*, pages 52–60. ACM Press, 2002.

[10] M. Kaminsky, E. Peterson, D. B. Giffin, K. Fu, D. Mazières, and M. F. Kaashoek. REX: Secure, extensible remote execution. In *Proceedings of the 2004 USENIX*, pages 199–212, Boston, MA, June–July 2004. USENIX.

[11] P. Karger, M. Zurko, D. Bonin, A. Mason, and C. Kahn. A retrospective on the VAX VMM security kernel. *Transactions on Software Engineering*, 17(11):1147–1165, 1991.

[12] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX*, Boston, MA, June–July 2004. USENIX.

[13] H. Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[14] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[15] P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of Ottawa Linux Symposium 2001*, June 2001.

[16] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX*, pages 261–274. USENIX, June 2001.

[17] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 129–142, Saint-Malo, France, October 1997. ACM.

[18] PHP: Hypertext processor. `http://www.php.net`.

[19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[20] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. In *Proceedings of the 5th ACM SIGOPS Workshop*, Mont Saint-Michel, 1992.

[21] Postfix. `http://www.postfix.org`.

[22] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–271, Washington, DC, August 2003.

[23] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington, D.C., August 2003.

[24] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[25] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119–130, Portland, OR, 1985. USENIX.

[26] The Sendmail Consortium. Sendmail. `http://www.sendmail.org`.

[27] J. S. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In G. Klein, editor, *Proc. NICTA Formal Methods Workshop on Operating Systems Verification*, Sydney, Australia, 2004. NICTA Technical Report 0401005T-1, National ICT Australia.

[28] J. S. Shapiro, J. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. Symposium on Operating Systems Principles*, pages 170–185, 1999.

[29] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *In Proceedings of the Eighth USENIX Security Symposium*, August 1999.

[30] M. Stiegler, A. H. Karp, K.-P. Yee, and M. Miller. Polaris: Virus safe computing for windows XP. Technical Report HPL-2004-221, December 2004.

[31] R. N. M. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 15–28. USENIX Association, 2001.

[32] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[33] M. V. Wilkes and R. M. Needham. *The Cambidge CAP Computer and its Operating System*. North Holland, 1979.

NOTES

[1] Were it not for this prohibition, unprivileged users could use control of the `chroot`ed top-level directory to elevate privileges. The attack is to make a new directory `/tmp/foo`, hard link from `/tmp/foo/su` to the system `su`, write a new password file `/tmp/foo/etc/passwd`, call `chroot` on `/tmp/foo`, and then call `su` from within the jail.

[2] Polaris appears not as well-suited for larger servers.

[3] We assume for simplicity that databases run locally, though all concepts discussed can generalize to distributed deployments.