Building Fast and Secure Web Services with OKWS

by

Maxwell Krohn

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

Author ....
Department of Electrical Engineering and Computer Science
September 2, 2005

Certified by .... [illegible] ....
M. Frans Kaashoek
Professor
Thesis Supervisor

Accepted by... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Building Fast and Secure Web Services with OKWS

by

Maxwell Krohn

Submitted to the Department of Electrical Engineering and Computer Science
on September 2, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

OKWS is a Web server specialized for secure and fast delivery of dynamic content. It provides Web developers with a small set of tools powerful enough to build complex Web-based systems. Despite its emphasis on security, OKWS shows performance improvements compared to popular systems: when servicing fully dynamic, non-disk-bound database workloads, OKWS's throughput and responsiveness exceed that of Apache 2 [3], Flash [42] and Haboob [76]. Experience with OKWS in a commercial deployment suggests it can reduce hardware and system management costs, while providing security guarantees absent in current systems. In the end, lessons gleaned from the OKWS project provide insight into how operating systems might better facilitate secure application design.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Acknowledgments

I thank the following contributors to this thesis: David Mazières, for his help throughout the OKWS project; Robert Morris and Russ Cox, for their help in debugging; Michael Walfish, Eddie Kohler, and the anonymous Usenix 2004 reviewers for their help on the original OKWS Usenix paper; Sarah Friedberg and Jeremy Stribling for proofreading; and the programmers, writers and designers at OkCupid.com—Patrick Crosby, Jason Yung, Chris Coyne, Alfred Perlstein, Christian Rudder and Sam Yagan—for adopting and improving OKWS. The *gzip* work in Appendix A was done in collaboration with abhi shelat. Work done on making changes to the underlying operating system is done as part of the Asbestos Project, a collaboration with Petros Efstathopoulos, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek and Robert Morris. Finally, I thank my advisor Frans Kaashoek.

This research was supported in part by the DARPA Composable High Assurance Trusted Systems program (BAA #01-24), under contract #N66001-01-1-8927.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many of today's dynamic Web sites are nothing more than HTTP interfaces to centralized, managed databases. As such they are gatekeepers, tasked with showing Alice only the small part of the database that pertains to her, while keeping Bob and Charlie's private data out of her reach. This task is critical and difficult in practice. One need only browse the IT headlines to get a taste for the flagrant access control failures that plague commercial Web Sites [47, 48]. Such attacks have yielded millions of e-mail addresses in some cases, and millions of credit card accounts in others. Attacks against Web sites often exploit weaknesses in popular Web servers or bugs in custom application-level logic. In theory, HTTP (or HTTPS) exposes narrow interfaces to remote clients and should not allow the litany of attacks that have surfaced over the years. In practice, emphasis on rapid deployment and performance often comes at the expense of security.

Consider the following example: Web servers typically provide Web programmers with powerful and generic interfaces to underlying databases and rely on coarse-grained database-level permission systems for access control. Web servers also tend to package logically separate programs into one address space. If a particular Web site serves its *search* and *newsletter-subscribe* features from the same machine, a bug in the former might allow a malicious remote client to select all rows from a table of subscribers' email addresses. In general, anything from a buffer overrun to an unexpected escape sequence can expose private data to an attacker. Moreover, few practical isolation schemes exist aside from running different services on different machines. A large Web site, serving thousands of different functions, might only be as strong as it weakest service.

1.1 A Solution

To plug the many security holes that plague existing Web servers, and to limit the severity of unforeseen problems, we have designed, implemented and deployed OKWS, the OK Web Server. Unlike typical Web servers, OKWS is specialized for dynamic content and is currently not well-suited to serving files from disk. It relies on existing Web servers, such as Flash [42] or Apache [3], to serve images and other static content. In deployment, we found that this separation of static and dynamic content is natural and, moreover, contributes to security.

Supporting dynamic content on Unix necessitates a trade-off between security and performance. On the one hand, the most secure Web architecture on Unix is one in which remote Web clients connect to a single dedicated server-side process. The user's data on the server, whether in memory or on disk, is only available to that one process. This architecture is impractical: storage constraints dictate that user data be aggregated into searchable databases; the operating system cannot handle too many active processes, nor can it afford to fork processes as users connect. On the other hand, most Web servers in the literature (such as Apache, Flash, and Haboob [76]) assign processes to users and services to maximize performance and without regard to security.

OKWS chooses a different point in the design space: it confines each *service* that a Web server runs to a single process, so that if there are any security problems in one service, they will not spread to others. OKWS's alignment of services and processes is an implementation of the natural principle of *least privilege* [54], beneficial in this case for both performance and security.

1.2 Contributions

The main contributions of this thesis are the design and implementation of the OKWS, and an analysis of its security, performance, and usability. We believe OKWS has carved out a new niche in the crowded Web server design space, achieving security properties that other Web servers have not, while still outperforming all the servers we tested it against. No mere academic exercise, OKWS is a real system currently used in at least one large-scale commercial Web site; details about OKWS's commercial deployment are discussed. For businesses and academics alike, OKWS is freely available under an open source license.

Though OKWS goes to great lengths to squeeze the maximum set of security properties out of the Unix interface, in some respects, it still falls short. Experience with OKWS suggests that writing secure Web servers is hard, and that the operating system should provide better security primitives to secure application programmers. This thesis sketches the more promising experimental OS designs that would make OKWS simpler and more secure, without sacrificing its current features, usability or performance.

1.3 Thesis Organization

The next two chapters examine popular Web technology, arguing that today's Web servers have a bad track record in security and do not seem to be improving. Yet, the simple design principles in Chapter 4 would go a long way to closing many Web server vulnerabilities, and Chapter 5 discusses OKWS's implementation of these principles. Chapters 6 and 7 argue that the resulting system is practical for building large systems, and Chapter 8 analyzes its performance, showing that OKWS's specialization for dynamic content helps it achieve better performance in simulated dynamic workloads than general purpose servers. Chapter 9 discusses the security achieved by the implementation, and Chapter 10 covers the overarching lessons learned from two years of experience with OKWS and explores new ideas in operating systems inspired by the work.

Chapter 2

Related work

There is a wide range of related work in Web servers—both static and dynamic. Many Web servers do not specialize in dynamic content, and none to our knowledge have achieved OKWS's security and performance properties.

Despite its problems, Apache is today's most popular popular Web Server [3]. Apache's many configuration options and modules allow Web programmers to extend its functionality with a variety of different programming languages. However, neither 1.3.x's multi-process architecture nor 2.0.x's multi-threaded architecture is conducive to process isolation. Also, its extensibility and mushrooming code base make its security properties difficult to reason about.

Highly-optimized event-based Web servers such as Flash [42] and Zeus [79] have eclipsed Apache in terms of performance. While Flash in particular has a history of outstanding performance serving static content, our performance studies here indicate that its architecture is less suitable for dynamic content. In terms of process isolation, one could most likely implement a similar separation of privileges in Flash as we have done with OKWS. However, as shown in Chapter 8, Flash's architecture might not be well-suited for dynamic content.

FastCGI [20] is a standard for implementing long-lived CGI-like helper processes. It allows separation of functionality along process boundaries but neither articulates a specific security policy nor specifies the mechanics for maintaining process isolation in the face of partial server compromise. Also, FastCGI requires the leader process to relay messages between the Web service and the remote client. OKWS passes file descriptors to avoid the overhead associated with FastCGI's relay technique.

The Haboob server studied here is one of many possible applications built on SEDA, an architecture for event-based network servers. In particular, SEDA uses serial event queues to enforce fairness and graceful degradation under heavy load. Larger systems such as Ninja [62] build on SEDA's infrastructure to create clusters of Web servers with the same appealing properties. However, performance measurements in Chapter 8 show that Haboob achieves between one fifth and one tenth the throughput of OKWS despite a strictly worse security model.

Other work has used the SFS toolkit to build static Web Servers and Web proxies [78]. Though the current OKWS architecture is well-suited for SMP machines, the adoption of *libasync-mp* would allow for finer-grained sharing of a Web workload across many CPUs.

OKWS uses events but the same results are possible with an appropriate threads library. An expansive body of literature argues the merits of one scheme over the other, and most recently, Capriccio's authors [63] argue that threads can achieve the same performance as events in the context of Web servers, while providing programmers with a more intuitive interface. Other recent work suggests that threads and events can coexist [1]. Such techniques, if applied to OKWS, would simplify stack management for Web developers.

In addition to the PHP [44] scripting language investigated here, many other Web development environments are in widespread use. Zope [81], a Python-based platform, has gained popularity due to its modularity and support for remote collaboration. CSE [25] allows developers to write Web services in C++ and uses some of the same sandboxing schemes we use here to achieve fault isolation. In more commercial settings, Java-based systems often favor thin Web servers, pushing more critical tasks to application servers such as JBoss [30] and IBM WebSphere [28]. Such systems limit a Web server's access to underlying databases in much the same way as OKWS's database proxies. Most Java systems, however, package all aspects of a system in one address space with many threads; our model for isolation would not extend to such a setting. Furthermore, our experimental results indicate significant performance advantages of compiled C++ code over Java systems.

Other work has proposed changes to underlying operating systems to make Web servers fast and more secure. The Exokernel operating system [31] allows its Cheetah Web server to directly access the TCP/IP stack, in order to reduce buffer copies allow for more effective caching. The Denali isolation kernel [77] can isolate Web services by running them on separate virtual machines.

Chapter 3

A Brief Survey of Web Server Bugs

To justify our approach to dynamic Web server design, we briefly analyze the weaknesses of popular software packages. The goal is to represent the range of bugs that have arisen in practice. Historically, attackers have exploited almost all aspects of conventional Web servers, from core components and scripting language extensions to the scripts themselves. The conclusion we draw is that a better design—as opposed to a more correct implementation—is required to get better security properties.

In our survey, we focus on the Apache [3] server due to its popularity, but the types of problems discussed are common to all similar Web servers, including IBM WebSphere [28], Microsoft IIS [38] and Zeus [79].

3.1 Apache Core and Standard Modules

There have been hundreds of major bugs in Apache's core and in its standard modules. They fit into the following categories:

Unintended Data Disclosure. A class of bugs results from Apache delivering files over HTTP that are supposed to be private. For instance, a 2002 bug in Apache's `mod_dav` reveals source code of user-written scripts [74]. A 2003 bug in a default installation of Apache Tomcat on Gentoo Linux publicly revealed authentication credentials [13]. Another recent vulnerability in Apache Tomcat causes the server to return unprocessed source code, as opposed to executing the source code and outputting the result [73]. Similarly, Tomcat's improper handling of null bytes (`%00`) and escape sequences has allowed remote attackers to view hidden directory contents [66].

A recent discovery of leaked file descriptors allows remote users to access sensitive log information [12]. On Mac OS X operating systems, a local find-by-content indexing scheme creates a hidden yet world-readable file called `.FBCIndex` in each directory indexed. Versions of Apache released in 2002 expose this file to remote clients [72]. In all cases, attackers can use knowledge about local configuration and custom-written application code to mount more damaging attacks.

Buffer Overflows and Remote Code Execution. Buffer overflows in Apache and its modules are common. Unchecked boundary conditions found recently in mod_alias and mod_rewrite regular expression code allow local attack [69]. In 2002, a common Apache deployment with OpenSSL had a critical bug in client key negotiation, allowing remote attackers to execute arbitrary code with the permissions of the Web server. The attacking code downloads, compiles and executes a program that seeks to infect other machines [65].

There have been less-sophisticated attacks that resulted in arbitrary remote code execution. Some Windows versions of Apache execute commands in URLs that follow pipe characters (`|`). A remote attacker can therefore issue the command of his choosing from an unmodified Web browser [71]. On MS-DOS-based systems, Apache failed to filter out special device names, allowing carefully-crafted HTTP POST requests to execute arbitrary code [75]. Other problems have occurred when site developers call Apache's htdigest utility from within CGI scripts to manage HTTP user authentication [10].

Denial of Service Attacks. Aside from TCP/IP-based DoS attacks, Apache has been vulnerable to a number of application-specific attacks. Apache versions released in 2003 failed to handle error conditions on certain "rarely used ports," and would stop servicing incoming connections as a result [68]. Another 2003 release allowed local configuration errors to result in infinite redirection loops [14]. In some versions of Apache, attackers could exhaust Apache's heap simply by sending a large sequence of linefeed characters [67].

3.2 Scripting Extensions to Apache

Apache's security worsens considerably when compiled with popular modules that enable dynamically-generated content such as PHP [44]. In the past two years alone, at least 13 buffer overruns have been found in the PHP core, some of which allowed attackers to remotely execute arbitrary code [16,56]. In six other cases, faults in PHP allowed attackers to circumvent its application level *chroot*-like environment, called "Safe Mode." One vulnerability exposed /etc/passwd via posix_getpwnam [9]. Another allowed attackers to write PHP scripts to the server and then remotely execute them; this bug persisted across multiple releases of PHP intended as fixes [64].

Even if a correct implementation of PHP were possible, it would still provide Web programmers with ample opportunity to introduce their own vulnerabilities. A canonical example is that beginning PHP programmers fail to check for sequences such as ".." in user input and therefore inadvertently allow remote access to sensitive files higher up in the file system hierarchy (*e.g.*, ../../../etc/passwd). Similarly, PHP scripts that embed unescaped user input inside SQL queries present openings for "SQL Injection." If a PHP programmer neglects to escape user input properly, a malicious user can turn a benign SELECT into a catastrophic DELETE.

The PHP manual does state that PHP scripts might be separated and run as different users to allow for privilege separation. In this case, however, PHP cannot run as an Apache module, and the system requires a new PHP process forked for every incoming connection. This isolation strategy compromises performance, and is seldom used in practice.

Chapter 4

Design

If we assume that bugs like the ones discussed in Chapter 3 are inevitable when building a large system, the best remedy is to limit the effectiveness of attacks when they occur. This section presents four simple guidelines for protecting sensitive site data in the worst-case scenario, in which an adversary remotely gains control of a Web server and can execute arbitrary commands with the Web server's privileges. We also present OKWS's design, which follows the four security guidelines without sacrificing performance.

Throughout, we assume a cluster of dynamic Web servers and database machines connected by a fast, firewalled LAN. Site data is cached at the Web servers and persistently stored on the database machines. The site's static data can be served from outside of the firewall since static content servers typically do not require access to sensitive site data. This type of configuration, exemplified in Figure 4.1, is common for high-volume Web sites. With this server setup, the primary security goals are to prevent intrusion into the firewalled cluster and to prevent unauthorized access to site data.

4.1 Four Practical Security Guidelines

(1) Server processes should be `chroot`*ed.* `chroot`ing privileged server processes is a commonly-accepted though infrequently-applied Unix security technique.[1] The rationale behind `chroot` is that privileged processes, if compromised, would do less damage if they could not access sensitive parts of the file system. In particular, secure systems can use `chroot` to hide local passwords, private keys, startup scripts, commonly-executed executable images (such as the kernel) and local configuration files from privileged servers. If a privileged server cannot access these files due to `chroot`, then a compromised privileged server cannot learn their secrets or sully their integrity.

From a more technical perspective, OS-level `chroot`-jails ought to hide all `setuid`[2] executables from the Web server, to prevent compromised processes from escalating priv-

[1] `chroot` is a system call, available only to the superuser, that changes an application's idea of what the file system root is. Calling `chroot` (*dir*) renders only the files and directories under *dir* accessible to the process [52].

[2] If a file is set to `setuid`, then any process that executes that file has its user ID set to the file's owner [52]. Typically, *su* is owned by the superuser and is marked `setuid`, allowing non-privileged users to execute it and become privileged.

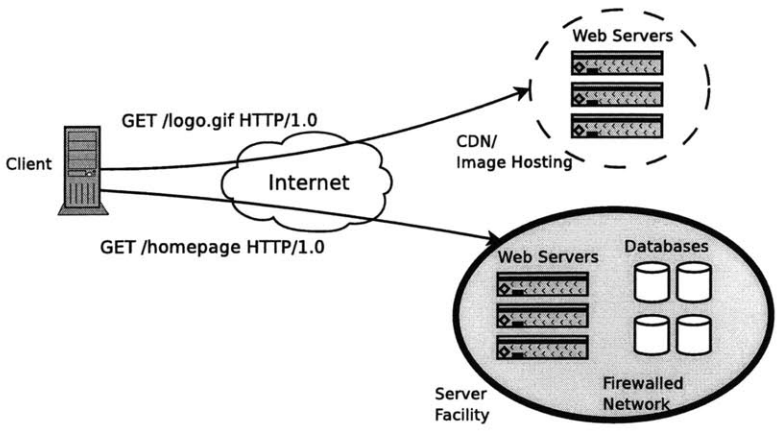

Figure 4.1: An example server setup, in which Web clients download HTML data from a firewalled clusted, and then are redirected to a content distribution network (CDN) or external image servers for downloading graphics.

ileges (examples include the *ptrace* and *bind* attacks mentioned in [32]). Privilege escalation is possible without `setuid` executables but requires OS-level bugs or race conditions (such as a recently discovered flaw in Linux's *mremap* system call [70]) that are typically rarer.

Moreover, developers should design their application around the `chroot` call as opposed to blindly applying a `chroot` container ex-post facto. Consider the Apache example. Once `chrooted`, Apache still needs access to configuration files, source files, and binaries that correspond to services the site administrator wants to make available; these files, in turn, are valuable to attackers who can compromise the server. For instance, PHP scripts often include the username and plaintext password used to gain access to a MySQL database. If an attacker can control Apache, he can read PHP scripts and wreak havoc by controlling the databases those PHP scripts access. OS-enforced policy ought to hide even application-specific source, binary and configuration files from running Web servers.

(2) Server processes should run as unprivileged users. To do otherwise—to run a server process as a privileged user— opens to the door to significant damage, even if the process is *chroot*ed. A privileged, *chroot*ed process can bind to a well-known network port. A privileged process might also interfere with other system processes, especially those associated with the Web server, by tracing their system calls, sending them signals or tampering with their binaries so that they exhibit unintended behavior on future executions.

(3) Server processes should have the minimal set of database access privileges necessary to perform their task. Separate processes should not have access to each other's databases. Moreover, if a Web server process requires only row-wise access to a table, an adversary who compromises it should not have the authority to perform operations over the entire table.

(4) A server architecture should separate independent functions into independent processes. An adversary who compromises a Web server can examine its in-memory data

structures, which might contain soft state used for user session management, or possibly secret tokens that the Web server uses to authenticate itself to its database. With control of a Web server process, an adversary might hijack an existing database connection or establish a new one with the authentication tokens it acquired. Though more unlikely, an attacker might also monitor and alter network traffic entering and exiting a compromised server.

The important security principle here is to limit the types of data that a single process can access. Site designers should partition their global set of site data into small, self-contained subsets, and their Web server ought to align its process boundaries with this partition.

If a Web server implements principles (1) through (4), and if there are no critical kernel bugs, an attacker cannot move from vulnerable to secure parts of the system. The same defenses also protect careless Web programmers from their own mistakes. For example, if a server architecture denies a successful attacker access to `/etc/passwd`, then a programmer cannot inadvertently expose this file to remote clients. Similarly, if a successful attacker cannot arbitrarily access underlying databases, then even a broken Web script cannot enable SQL injection attacks.

4.2 OKWS Design

We designed OKWS with these four principles in mind. OKWS provides Web developers with a set of libraries and helper processes so they can build Web services as independent, stand-alone processes, isolated almost entirely from the file system. The core libraries provide basic functions of receiving HTTP requests, accessing data sources, composing an HTML-formatted response, responding to HTTP requests, and logging the results to disk. A process called OK launcher daemon, or *okld*, launches custom-built services and relaunches them should they crash. A process called OK dispatcher, or *okd*, routes incoming requests to appropriate Web services based on URL. A helper process called *pubd* provides Web services with limited read access to configuration files and HTML template files stored on the local disk. Finally, a dedicated logger daemon called *oklogd* writes log entries to disk. Figure 4.2 summarizes these relationships.

This architecture allows custom-built Web services to meet our stated design goals:

(1) OKWS *chroot*s all services to a remote jail directory. Within the jail, each process has just enough access privileges to read shared libraries upon startup and to dump core upon abnormal termination. The services otherwise never access the file system and lack the privileges to do so.

(2) Each service runs as a unique non-privileged user.

(3) OKWS interposes a structured RPC interface between the Web service and the database and uses a simple authentication mechanism to align the partition among database access methods with the partition among processes.

(4) Each Web service runs as a separate process. The next section justifies this choice.

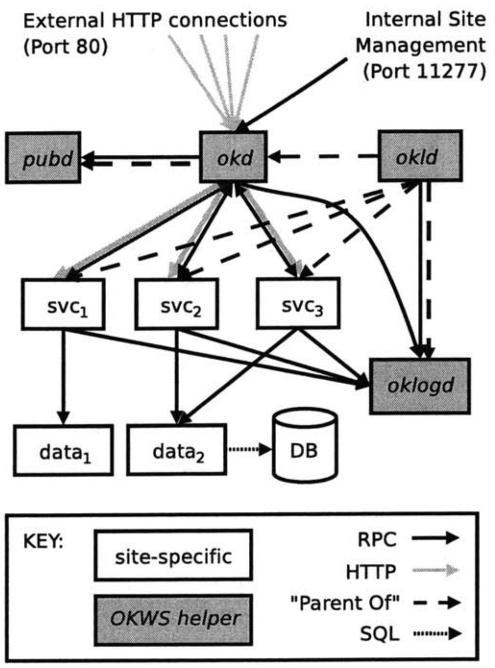

Figure 4.2: Block diagram of an OKWS site setup with three Web services (svc$_1$, svc$_2$, svc$_3$) and two data sources (data$_1$, data$_2$), one of which (data$_2$) is an OKWS database proxy.

4.3 Process Isolation

Unlike the other three principles, the fourth, of process isolation, implies a security and performance tradeoff since the most secure option—one Unix process per *external user*—would be problematic for performance. OKWS's approach to this tradeoff is to assign one Unix process per *service*.

Our approach is to view Web server architecture as a dependency graph, in which the nodes represent processes, services, users, and user state. An edge (a, b) denotes b's dependence on a, meaning an attacker's ability to compromise a implies an ability to compromise b. The crucial design decision is thus how to establish dependencies between the more abstract notions of services, users and user states, and the more concrete notion of a process.

Let the set S represent a Web server's constituent services, and assume each service accesses a private pool of data. Two services are defined as distinct if they access disjoint pools of data—whether in-memory soft state or database-resident hard state. A set of users U interacts with these services, and the interaction between user u_j and service s_i involves a piece of state $t_{i,j}$. If an attacker can compromise a service s_i, he can compromise state $t_{i,j}$ for all j; thus $(s_i, t_{i,j})$ is a dependency for all j. Compromising state also compromises the corresponding user, so $(t_{i,j}, u_j)$ is also a dependency.

Let $P = \{p_1, \ldots, p_k\}$ be a Web server's pool of processes. The design decision of how to allocate processes reduces to where the nodes in P belong on the dependency graph. In the Apache architecture [3], each process p_i in the process pool can perform the role of any service s_j. Thus, dependencies (p_i, s_j) exist for all j. For the Flash Web server [42], each process in P is associated with a particular service: for each p_i, there exists s_j such

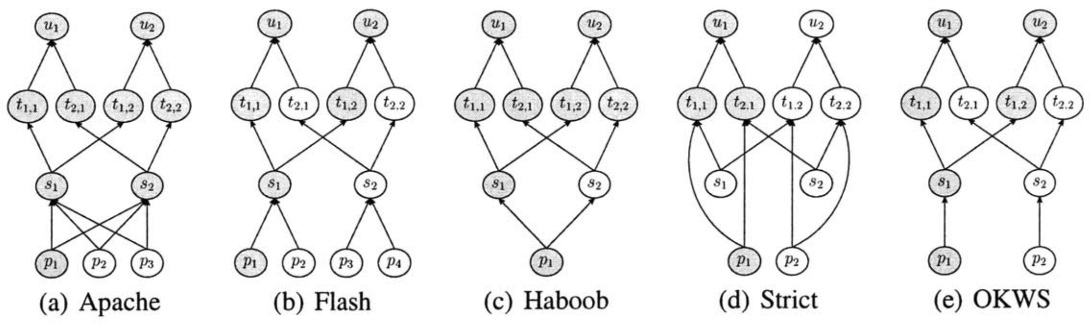

Figure 4.3: Dependency graphs for various Web server architectures.

that (p_i, s_j) is a dependency. The size of the process pool P is determined by the number of concurrent active HTTP sessions; each process p_i serves only one of these connections. Java-based systems like the Haboob Server [76] employ only one process; thus $P = \{p_1\}$, and dependencies (p_1, s_j) exist for all j.

Figures 4.3(a)-(c) depict graphs of Apache, Flash and Haboob hosting two services for two remote users. Assuming that the "dependence" relationship is transitive, and that an adversary can compromise p_1, the shaded nodes in the graph show all other vulnerable entities.

This picture assumes that the process of p_1 is equally vulnerable in the different architectures and that all architectures succeed equally in isolating different processes from each other. Neither of these assumptions is entirely true, and we will return to these issues in Section 9.2. What is clear from these graphs is that in the case of Flash, a compromise of p_1 does not affect states $t_{2,1}$ and $t_{2,2}$. For example, an attacker who gained access to u_i's search history $(t_{1,i})$ cannot access the contents of his inbox $(t_{2,i})$.

A more strict isolation strategy is shown in Figure 4.3(d). The architecture assigns a process p_i to each user u_i. If the attacker is a user u_i, he should only be able to compromise his own process p_i, and will not have access to state belonging to other users u_j. The problem with this approach is that it does not scale well on current operating systems. A Web server would either need to fork a new process p_i for each incoming HTTP request or would have a large pool of mostly idle processes, one for each currently active user (of which there might be tens of thousands).

OKWS does not implement the strict isolation strategy but instead associates a single process with each individual service, shown in Figure 4.3(e). As a result OKWS achieves the same isolation properties as Flash but with a process pool whose size is independent of the number of concurrent HTTP connections. OKWS thus requires significantly fewer processes than Flash under heavy load.

Chapter 5

Implementation

OKWS is a portable, event-based system, written in C++ with the SFS toolkit [37]. It has been successfully tested on Linux and FreeBSD. In OKWS, the different helper processes and site-specific services shown in Figure 4.2 communicate among themselves with SFS's implementation of Sun RPC [61]; they communicate with external Web clients via HTTP. Unlike other event-based servers [42, 76, 79], OKWS exposes the event architecture to Web developers. We could most likely have achieved equivalent security and performance results through the use of cooperative user-level threading, or perhaps through a hybrid scheme [1].

To use OKWS, an administrator installs the helper binaries (*okld*, *okd*, *pubd* and *oklogd*) to a standard directory such as `/usr/local/sbin`, and installs the site-specific services to a runtime jail directory, such as `/var/okws/run`. The administrator should allocate two new UID/GID pairs for *okd* and *oklogd* and should also reserve a contiguous user and group ID space for "anonymous" services. Finally, administrators can tweak the master configuration file, `/etc/okws_config`. Table 5.1 summarizes the runtime configuration of OKWS.

5.1 okld

The root process in the OKWS system is *okld*—the launcher daemon. This process normally runs as superuser but can be run as a non-privileged user for testing or in other cases when the Web server need not bind to a privileged TCP port. When *okld* starts up, it reads the configuration file `/etc/okws_config` to determine the locations of the OKWS helper processes, the anonymous user ID range, which directories to use as jail directories, and which services to launch. Next, *okld* launches the logging daemon (*oklogd*) and the demultiplexing daemon (*okd*), and *chroot*s into its runtime jail directory. It then launches all site-specific Web services. The steps for launching a single service are:

1. *okld* requests a new Unix socket connection from *oklogd*.
2. *okld* opens 2 socket pairs; one for HTTP connection forwarding, and one for RPC control messages.
3. *okld* calls `fork`.

process	*chroot* jail	run directory	uid	gid
okld	/var/okws/run	/	root	wheel
pubd	/var/okws/htdocs	/	www	www
oklogd	/var/okws/log	/	oklogd	oklogd
okd	/var/okws/run	/	okd	okd
svc_1	/var/okws/run	/cores/51001	51001	51001
svc_2	/var/okws/run	/cores/51002	51002	51002
svc_3	/var/okws/run	/cores/51003	51003	51003

Table 5.1: An example configuration of OKWS. The entries in the "run directory" column are relative to "*chroot* jails".

4. In the child address space, *okld* picks a fresh UID/GID pair $(x.x)$, sets the new process's group list to $\{x\}$ and its UID to x. It then changes directories into `/cores/`x.

5. Still in the child address space, *okld* calls `execve`, launching the Web service. The new Web service process inherits three file descriptors: one for receiving forwarded HTTP connections, one for receiving RPC control messages, and one for RPC-based request logging. Some configuration parameters in `/etc/okws_config` are relevant to child services, and *okld* passes these to new children via the command line.

6. In the parent address space, *okld* sends the server side of the sockets opened in Step 2 to *okd*.

Upon a service's first launch, *okld* assigns it a group and user ID chosen arbitrarily from the given range (*e.g.*, 51001-51080). The service gets those same user and group IDs in subsequent launches. It is important that no two services share a UID or GID, and *okld* ensures this invariant. The service executables themselves are owned by root, belong to the group with the anonymous GID x chosen in Step 4 and are set to mode `0410`.

These settings allow *okld* to call `execve` after `setuid` but disallow a service process from changing the mode of its corresponding binary. *okld* changes the ownerships and permissions of service executables at launch if they are not appropriately set. The directory used in Step 4 is the only one in the jailed file system to which the child service can write. If such a directory does not exist or has the wrong ownership or permissions, *okld* creates and configures it accordingly.

okld catches `SIGCHLD` when services die. Upon receiving a non-zero exit status, *okld* changes the owner and mode of any core files left behind, rendering them inaccessible to other OKWS processes. If a service exits uncleanly too many times in a given interval, *okld* will mark it broken and refuse to restart it. Otherwise, *okld* restarts dead services following the steps enumerated above.

okld runs as super-user but unlike other OKWS daemons does not drop its privileges. However, it has many fewer potential vulnerabilities in that it does not listen for any explicit messages—RPC, HTTP or otherwise. Rather, it responds only to `SIGCHLD` signals.

5.2 okd

The *okd* process accepts incoming HTTP requests and demultiplexes them based on the "Request-URI" in their first lines. For example, the HTTP/1.1 standard [21] defines the first line of a GET request as:

GET /⟨*abs_path*⟩?⟨*query*⟩ HTTP/1.1

Upon receiving such a request, *okd* looks up a Web service corresponding to *abs_path* in its dispatch table. If successful, *okd* forwards the remote client's file descriptor to the requested service. If the lookup is successful but the service is marked "broken," *okd* sends an HTTP 500 error to the remote client. If the request did not match a known service, *okd* returns an HTTP 404 error. In typical settings, a small and fixed number of these services are available—on the order of 10. The set of available services is fixed once *okd* reads its configuration file at launch time. These restrictions would prove inconvenient for a traditional static Web server, because site maintainers frequently add, rename and delete static content. In our setting, we assume that site developers add, rename and delete Web services infrequently.

Upon startup, *okd* reads the OKWS configuration file (`/etc/okws_config`) to construct its dispatch table. *okd* inherits two file descriptors from *okld*: one is a socket that connects to *oklogd*, the other is a pipe across which *okld* will send it control messages. *okd* uses its connection to the logger to log errors messages such as error 404 (file not found) or error 500 (internal server error). *okd* uses the pipe from *okld* to listen for information about Web services that are starting up. When *okld* starts up a service, it creates two socket pairs, one for control messages, and one for HTTP connections. *okld* gives one half of these connections to the service (see Section 5.1, Step 6). It sends the other half of these socket pairs to *okd* over the *okld*- to - *okd* pipe.

Thus, *okd* receives two file descriptors from *okld* for each service launched. *okd* uses one of these sockets to send service control messages. It uses the other socket to pass file descriptors after successfully demultiplexing incoming HTTP requests.

okd typically is launched with supervisor privileges but drops them (via `setuid`) after binding to its listen port and *chroot*ing into its designated runtime jail directory. For testing purposes, *okd* can run as a non-privileged user, assuming it does not need to bind to a privileged port (such as a TCP port greater than 1024). If the given configuration uses a local *pubd*, then *okd* must also launch *pubd* before dropping its privileges.

In addition to listening for HTTP connections on port 80, *okd* listens for internal requests from an administration client. It services the two RPC calls: REPUB and RELAUNCH. A site maintainer should call the former to "activate" any changes she makes to HTML templates (see Section 5.4 for more details). Upon receiving a REPUB RPC, *okd* triggers a simple update protocol that propagates updated templates.

A site maintainer should issue a RELAUNCH RPC after updating a service's binary. Upon receiving a RELAUNCH RPC, *okd* simply sends an EOF to the relevant service on its control socket. When a Web service receives such an EOF, it finishes responding to all pending HTTP requests, flushes its logs, and then exits cleanly. The launcher daemon, *okld*, then catches the corresponding `SIGCHLD` and restarts the service.

5.3 oklogd

All services, along with *okd*, log their access and error activity to local files via *oklogd*—the logger daemon. Because these processes lack the privileges to write to the same log file directly, they instead send log updates over a local Unix domain socket. To reduce the total number of messages, services send log updates in batches. Services flush their log buffers as they become full and at regularly-scheduled intervals.

Because only *oklogd* writes to the log file, we expect performance improvements over multi-process architectures, in which atomic appends to log files can only be guaranteed through kernel-level synchronization or spin-locks. For a busy Webserver, we would expect frequent contention over log file resources. But batched logging via *oklogd* has two important downsides. First, logs are not locally ordered unless *oklogd* sorts and merges batches from different processes. To avoid the additional computation and data manipulation involved with a run-time sort, OKWS defers the responsibility of completely ordering the log files to the offline tools.

Second, and more important, if a Web service crashes, its buffered batch of log entries is lost. In the deployments we have seen, we do not anticipate the loss of a few seconds of log entries to be a cause for concern. In our current implementation, the cause of the crash is discernible from the core dump that the service produces upon abnormal termination. In fact, for large HTTP POSTs that trigger server crashes, a core dump contains more information about the offending request than a single HTTP log line. Future versions of OKWS might ship log entries to *oklogd* via memory-mapped files (one for each service); in this case, the logger can access a service's cached log entries after it has crashed.

For security, *oklogd* runs as an unprivileged user in its own *chroot* environment. Also, OKWS runs *oklogd* as a different user from the *okd* user. The combination of these two methods ensures that a compromised *okd* or Web service cannot maliciously overwrite or truncate log files; it would only have the ability to fill them with "noise".

5.4 pubd

Dynamic Web pages often contain large sections of static HTML code. In OKWS, such static blocks are called HTML "templates"; they are stored as regular files, can be shared by multiple services and can include each other in a manner similar to Server Side Includes [4].

OKWS services do not read templates directly from the file system. Rather, upon startup, the publishing daemon (*pubd*) parses and caches all required templates. It then ships parsed representations of the templates over RPC to other processes that require them. *pubd* runs as an unprivileged user, relegated to a jail directory that contains only public HTML templates. As a security precaution, *pubd* never updates the files it serves, and administrators should set its entire *chroot*ed directory tree read-only (perhaps, on those platforms that support it, by mounting a read-only *nullfs*).

Administrators in a cluster setting might chose to run *pubd* locally on an NFS-mounted template directory. *pubd* also runs as a network service so that a single *pubd* instance can serve a cluster of machines.

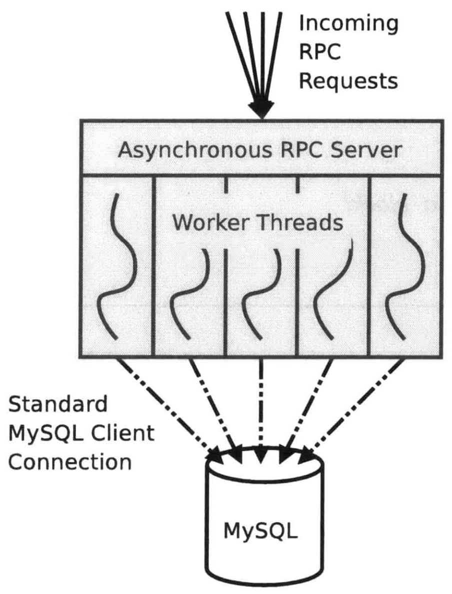

Figure 5.1: A typical Database Proxy for a MySQL database, configured with five worker threads.

5.5 Asynchronous Database Proxies

OKWS's library implementation is agnostic to particular flavors of threading libraries and currently allows the site developer to chose from one of three possibilities. For instance, a site developer would choose kernel threads when implementing a database proxy for interfacing with an embedded database such as Berkeley DB and would choose cooperative user threads for a socket-based database such as MySQL. The general architecture is reminiscent of Flash's helper processes [42], and "manual calling automatic" in Adya's work [1].

A database proxy has one dispatcher thread, and a pool of worker threads. The dispatcher thread receives incoming RPC requests and delegates it to one of its ready worker threads, or queuing it if all threads are busy. Based on the given RPC, the worker thread queries the database, perhaps blocking. Upon completion, it stores its response to a memory region shared among threads, and sends a small "I am finished" message to the dispatch thread over a pipe. The dispatch thread then responds to the Web server's RPC with the appropriate result. This configuration is shown in Figure 5.1.

Database proxies employ a static number of worker threads and do not expand their thread pool. A site maintainer should tune the database proxy's concurrency factor to find the minimum number of threads that allow the database to perform at maximum throughput. The intent here is simply to overlap requests to the underlying data source so that it might overlap its disk accesses and benefit from disk arm scheduling. An unnecessarily large thread pool might be bad for performance, since databases like MySQL often require significant per-thread state. A reasonable choice is 5 threads per database disk.

For security, database proxies ought to run on the database machines themselves. Such a configuration allows the site administrator to "lock down" a socket-based database server,

```
// can only occur at initialization time
q = mysql->prepare ("SELECT age,name FROM tab WHERE id=?");

id = 1; // get ID from client
user_t u;
stmt = q->execute (id); // might block!
stmt->fetch (&u.age, &u.name);
reply (u);
```

Figure 5.2: Example of database proxy code with MySQL wrapper library. In this case, the Web developer is loading SQL results directly into an RPC XDR structure. Error conditions ignored for brevity. The `user_xdt_t` structure is defined in the protocol file in Figure 5.3.

so that only local processes can execute arbitrary database commands. All other machines in the cluster—such as the Web server machines—only see the structured, and thus restricted, RPC interface exposed by the database proxy. Database proxies employ a simple mechanism for authenticating Web services. After a Web service connects to a database proxy, it supplies a 20-byte authentication token in a login message. The database proxy then grants the Web service permission to access a set of RPCs based on the supplied authentication token.

The database proxy libraries provide the illusion of a standard asynchronous RPC dispatch routine. Internally, these proxies are multi-threaded and can block; the library hides synchronization and scheduling details. To facilitate development of OKWS database proxies, we wrapped MySQL's standard C library in an interface more suitable for use with SFS's libraries. We model our MySQL interface after the popular Perl DBI interface [43] and likewise transparently support both parsed and prepared SQL styles. Figure 5.2 shows a simple database proxy built with this library.

5.6 Building A Web Service

A Web developer creates a new Web service as follows:

1. Extends two OKWS generic classes: one that corresponds to a long-lived service (`ok_service_t`), and one that corresponds to a short-lived, individual HTTP request (`ok_client_t`). Implements the `init` method of the former and the `process` method of the latter.

2. Runs the source file through OKWS's preprocessor, which outputs C++ code.

3. Compiles this C++ code into an executable, and installs it in OKWS's service jail.

4. Adds the new service to `/etc/okws_config`.

5. Restarts OKWS to launch.

The resulting Web service is a single-threaded, event-driven process.

```
struct user_t {
  string name<30>;
  int age;
};

program USER_PROG {
        version USER_PROG_1 {
                user_t
                GET_USER(unsigned) = 1;
        } = 1;
} = 10000;
```

Figure 5.3: `user.x`: the RPC protocol specification file used in client/server examples for this section. The program has one RPC, `GET_USER`, which takes as input an `unsigned` user ID and outputs a `user_t` structure.

The OKWS core libraries handle the mundane mechanics of a service's life cycle and its connections to OKWS helper processes. At the initialization stage, a Web service establishes persistent connections to all needed databases. The connections last the lifetime of the service and are automatically reopened in the case of abnormal termination. Also at initialization, a Web service obtains static HTML templates and local configuration parameters from *pubd*. These data stay in memory until a message from *okd* over the RPC control channel signals that the Web service should refetch. In implementing the `init` method, the Web developer need only specify which database connections, templates and configuration files he requires.

The `process` method specifies the actions required for incoming HTTP requests. In formulating replies, a Web service typically accesses cached soft-state (such as user session information), database-resident hard state (such as inbox contents), HTML templates, and configuration parameters. Because a Web service is implemented as a single-threaded process, it does not require synchronization mechanisms when accessing these data sources. Its accesses to a database on behalf of all users are pipelined through a single asynchronous RPC channel. Similarly, its accesses to cached data are guaranteed to be atomic and can be achieved with simple lightweight data structures, without locking. By comparison, other popular Web servers require some combination of *mmap*'ed files, spin-locks, IPC synchronization, and database connection pooling to achieve similar results.

Native Web service development in OKWS is with C++, with the same SFS event library that undergirds all OKWS helper processes and core libraries. (See Section 7 for a description of higher-level language support.) To simplify memory management, OKWS exposes SFS's reference-counted garbage collection scheme and high-level string library to the Web programmer. OKWS also provides a C++ preprocessor that allows for Perl-style "heredocs" and simplified template inclusion. Figure 5.4 demonstrates these facilities.

Developers using the OKWS system develop a standalone server process for each discrete Web service they wish to deploy. In many cases, different services share many of the same library routines, but should be separated into as many different processes as possible while still allowing for effective caching. For instance, a Web site's *mail* and *search* fea-

```
void my_srvc_t::process ()
{
  str color = param["color"];
  /*o
    print (resp) <<EOF;
<html>
 <head>
  <title>${param["title"]}</title>
 </head>
EOF
    include (resp, "/body.html",
             { COLOR => ${color}});
  o*/
  output (resp);
}
```

Figure 5.4: Fragment of a Web service programmed in OKWS. The remote client supplies the title and color of the page via standard CGI-style parameter passing. The runtime templating system substitutes the user's choice of color for the token `COLOR` in the template `/body.html`. The variable `my_svc_t::resp` represents a buffer that collects the body of the HTTP response and then is flushed to the client via `output()`. With the `FilterCGI` flag set, OKWS filters all dangerous metacharacters from the `param` associative array.

tures access and cache almost entirely disjoint pools of data—perhaps mesage headers in the former case and the results to the most common search results in the latter case. They might, however, use the same codebase to effect a "look-and-feel" common to the whole Website. They might also use local RPC servers to share small pieces of state, such as user session information.

5.7 Code Size

Currently, OKWS is approximately 25,000 lines of C++, FLEX and YACC code. About 9200 lines of code implement the HTTP templating libraries, 6200 lines implement our asynchronous HTTP libraries, 3000 lines round out the helper processes, and 1700 lines implement the database proxy libraries.

Chapter 6

OkCupid.com: OKWS In Action

The author and two other programmers built a commercial Web site using the OKWS system in six months [41]. We were assisted by two designers who knew little C++ but made effective use of the HTML templating system. The application is Internet dating, and the site features a typical suite of services, including local matching, global matching, messaging, profile maintenance, site statistics, and picture browsing. Almost a million users have established accounts on the site, and at peak times, thousands of users maintain active sessions. The current implementation uses 34 Web services and 12 database proxies.

For developers accustomed to higher level languages and direct programming semantics, SFS's continuation-style semantics took some time to acquaint to. We also have been frustrated at times by OKWS's long compile times (OKWS and SFS together make extensive use of C++ templating). Finally, certain types of programming—database accesses in particular—take more lines of code in OKWS as they would in a system like PHP, sometimes by factors of two or three.

Otherwise, we have found the system to be usable, stable and well-performing. In the absence of database bottlenecks or latency from serving advertisements, OKWS feels very responsive to the end user. Even those pages that require heavy iteration—like match computations—load instantaneously. In terms of stability, we've gone weeks at a time without the core OKWS processes crashing, and when they do, a core dump is usually enough to reconstruct our error.

OkCupid.com's Web cluster consists of four load balanced OKWS Web server machines, two read-only cache servers, and two read-write database servers, all with dual Pentium 4 processors. We use multiple OKWS machines only for redundancy; one machine can handle peak loads (about 200 requests per second) at about 7% CPU utilization, even as it *gzip*s most responses. A previous incarnation of this Web site required six Mod-Perl/Apache servers [39] to accommodate less traffic. It ultimately was abandoned due prohibitive hardware and hosting expenses [59].

6.1 Separating Static From Dynamic

OKWS relies on other machines running standard Web servers to distribute static content. This means that all pages generated by OKWS should have only absolute links to external static content (such as images and style sheets), and OKWS has no reason to support keep-alive connections [21]. The servers that host static content for OKWS, however, can enable HTTP keep-alive as usual.

Serving static and dynamic content from different machines is already a common technique for performance reasons; administrators choose different hardware and software configurations for the two types of workloads. Moreover, static content service does not require access to sensitive site data and can therefore happen outside of a firewalled cluster, or perhaps at a different hosting facility altogether. Indeed, some sites push static content out to external distribution networks such as Akamai [2]. The OkCupid deployment hosts a cluster of OKWS and database machines at a local colocation facility, where hands-on hardware access and custom configuration is possible. OkCupid serves static content from leased, dedicated servers at a remote facility where bandwidth is significantly cheaper.

6.2 Compression

To reduce bandwidth costs and to improve perceived site responsiveness, OKWS responds to clients queries with *gzip*-encoded responses whenever possible. OKWS's current approach is to encode the static HTML portions of responses with the densest possible compression, and to encode dynamically-generated portions at lower compression levels. In practice, this means that *pubd* compresses HTML templates as it reads them off disk, and sends both plain and compressed versions to the OKWS services. Dynamic output from OKWS services passes through a faster *gzip* filter, and the OKWS libraries perform the required checksums and concatenation. A sample August 2005 trace shows that 82% of client browsers support *gzip* encoding. Using level 2 and 9 compression for dynamic and static elements, respectively, OkCupid's Web servers compress outgoing documents to about 37% of their original size. Over all Web requests, OKWS's compression strategy gives a 48% savings in bandwidth. See Appendix A for more details and future work concerning compression.

Chapter 7

Python Support in OKWS

A recent addition to OKWS is support for Web services built in Python [50], a high level interpreted language favored by Web serving systems like Zope [81]. There are some important advantages to writing Web applications in Python as opposed to C++: faster prototyping, no compilation, and less required programming expertise are just a few. In practice, experienced programmers might write a Web site's most CPU-intensive and frequently-accessed services in C++, but a team of junior programmers might implement less popular services in an interpreted language like Python. Indeed, a Web server like OKWS needs to incorporate some higher level language support to be useful to commercial Web sites.

The choice of Python as an interpreted language was largely one of taste: Ruby and Perl are similarly popular among programmers, also have a large repository of useful libraries and would have provided the same core features.

Four important goals guided the introduction of Python into OKWS:

1. **Minimal changes to the underlying OKWS code base.** Because OKWS is relatively stable, production software, we did not want to introduce regressions into the system and therefore hoped to leave most of the current code intact.

2. **Maximal reuse of exiting OKWS code.** One possibility for a Python port would be to implement services as pure Python processes, reimplementing in Python the Unix protocols that C++ services use on startup, and REPUB. However, this approach would require any future changes to the OKWS protocol to be made in two places: the C++ classes and also the Python clone. We hoped to avoid this situation by reusing as much of the OKWS libraries as possible.

3. **Decent performance.** Though a slowdown is inevitable when moving from compiled C/C++ code to an interpreted language such as Python, decent performance should still be possible.

4. **Preservation of all security properties.** Processes running OKWS services written in Python should be isolated from other Python and C++ OKWS services. Moreover, Python interpreters should be unavailable to compromised Web services within the jail.

7.1 Approach

A hybrid C++ and Python approach satisfies all of these goals. The rough idea is one new OKWS service written in C++ that acts as generic glue code. It negotiates with *okd*, *pubd*, *oklogd* and *okld* as normal but calls into a Python interpreter at the `init` and `process` methods. Python Web developers then implement `init` and `process` *in Python*, and the generic glue service implements the mechanics of moving data into and out of the Python environment.

This approach satisfies Requirements 1 through 3 listed above. First, OKWS need not be modified at all to accommodate the Python glue service, since it has the same interface into the OKWS system as native C++ Web services. Second, any changes made to the initialization or REPUB protocol would be automatically reflected in the Python glue service after recompiling and relinking, since to OKWS, the Python glue service is indistinguishable from regular C++ services. Finally, some of the more computationally-expensive parts of the HTTP request cycle — such as the core select loop or *gzip*ping output — are still handled in the C++ libraries and therefore will still perform well. Naturally, the application logic, which might be doing iterative tasks like formatting HTML tables, will be in Python and therefore slow when compared to compiled code.

A final detail to consider is database interaction: Python services need to access OKWS database proxies over asynchronous RPC, much in the same way as native C++ services. One approach to asynchronous RPC proxies is to implement the RPC transport and XDR marshalling/unmarshalling routines in Python. This reimplementation of a feature already available in the SFS libraries would contradict implementation goals 1 and 2. It might also be slow.

But the hybrid or "glue" approach is also possible here. When a service needs to make a database call, it calls a Python method that calls into the SFS RPC libraries, registering a user-specified Python callback. An RPC compiler generates code to handle marshalling a Python-accessible data representation into the standard XDR wire representation, and the SFS libraries handle the details of sending the RPC as normal. When a response comes back from the database, the SFS libraries call into autogenerated unmarshalling routines to make the data available to Python. The SFS libraries then call a standard C++ callback, which will in turn fire the Python callback that the Web service registered earlier, calling back into the Python interpreter.

7.2 Implementation

7.2.1 Wrappers for the SFS Libraries

The first implementation detail for OKWS Python support is the "glue" between the SFS libraries and Python, so that Python programs can access SFS's core event loop, its asynchronous network event dispatch, and its RPC libraries. Figure 7.1 shows a code sample that uses the Python interface to the SFS libraries. This code sample is a client that corresponds to the database server seen in Figure 5.2. It makes a connection to the database server with `sock.connect()`, gets an SFS-style asynchronous TCP transport at line 18,

```
import sfs.core
import sfs.arpc
import socket
import xdr.user as user

def cb (err, res):
    if err != 0:
        print "RPC error: " + err
    else:
        print "Username: " + res.name + "; age: " + res.age
    sfs.core.exit ()

port = 30888
uid = 1

sock = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
sock.connect (("127.0.0.1", port))
x = sfs.arpc.axprt_stream (sock)
client = sfs.arpc.aclnt (x, user.user_prog_1 ())
client.call (user.GET_USER, uid, cb)

sfs.core.amain ()
```

Figure 7.1: A sample Python application using SFS libraries

gets an RPC client that will appropriate marshall and unmarshall data at line 19, makes the RPC call with `client.call()`, and finally, calls into the SFS select loop at line 22. When the database has replied with a response, the callback `cb()` is called, with the first argument being an error code, and the second argument the data sent back in response. As line 10 shows, the data is now accessible as standard Python classes.

To run this code, the programmer first compiles `user.x` (as shown in Figure 5.3) with a new Python-aware RPC compiler. The RPC compiler outputs header and source code files in C++, which represent data structures, methods, and functions that will eventually be made accessible to the Python interpreter. It also outputs marshalling and unmarshalling routines that convert the Python-compatible structures to and from wire representation. Compiling these source files with a standard C++ compiler, and linking them into a shared object yields a resource that can be imported into Python, as seen in line 4 of Figure 7.1. Now, the programmer is ready to run the given Python code with the standard Python interpreter.

7.2.2 OKWS Glue

With an appropriate Python interface into the SFS libraries, the rest of the task is completed with straightforward glue code. An example of how Python in OKWS is used is shown in Figure 7.2. Like the C++ interface, the Python interface requires the programmer to subclass two virtual base classes: `service` and `client`. Under the covers, these two classes are actually Python wrappers around the C++ base classes `ok_service_t` and ok_client_t, respectively. Within these two classes, the programmer fills in code for service.__init__ and client.process.

In this particular example, the programmer is using the service.__init__ method to acquire an RPC connection to a database on line 27. The client.process method is, as usual, where the HTTP connection is actually handled. In this example, the programmer checks the user's request for the uid variable and then makes a database query with that value as input. When the query returns, control is transfered to the callback example_client.cb(), and the database's response is pasted into the given template (/get_user.html) by the publishing system. The zbuf object used to collect the response and send it out to the client via self.output() is just a wrapper around the OKWS smart *gzip*er mentioned in Section 6.2.

To start this script, as in line 30, the programmer must furnish the underlying libraries with the first command line argument sys.argv[1], which *okld* uses to pass configuration parameters to the Web service. These options are parsed and acted upon in the service class's __init__() method, which calls into the OKWS libraries. The launch() call on line 31 starts the standard service setup negotiation with *okd*. Finally, as in Figure 7.1, the Python script calls sfs.core.amain() to immediately enter into the SFS select loop.

7.3 Launching Python Services and Security

Recall from Section 5.1, that the OK launcher daemon (*okld*) starts each OKWS service process as its own anonymous user; furthermore, *okld* assigns each executable ownership and permissions so that other services cannot read the file, and no service can write to it. The same ideas should apply to Python OKWS services, but the situation is complicated by the fact that each Python service requires two files: the Python interpreter itself, and the script to be executed. By goal 4, we must ensure that no OKWS Python scripts can read the text of any other script, or of any C++ service, and moreover, that no C++ service can read the code of Python scripts, and that no C++ service can execute the Python interpreter.

If we consider all of these constraints, the more convenient solutions to jailing the Python environment are insufficient. For instance, one approach is to copy the Python interpreter into the jail, owned by root and belonging to group pysvc, and set to mode 0550. Then, all Python scripts in the jail are owned by unique anonymous users, and each anonymous user is a member of pysvc. Python scripts are set to mode 0400 so that only the owner of the script can read it and execute it with the interpreter. The problem, of course, is that a compromised Python script can change the mode of its script and write to it, so that any damage to the system can persist across reboots.

OKWS adopts a safer albeit clunkier approach:

1. For each Python service, *okld* picks a fresh UID/GID pair ($x.x$) as it does for C++ services.

2. *okld* copies the Python interpreter (python) to a new interpreter python-x, changes its owner to root, its group to x, and it access mode to 0550.

```
import okws.objs
from okws.client import client
from okws.service import service
import sfs.core
import sfs.arpc
import xdr.user as user

class example_client (client):

    def cb (self, err, res):
        buf = okws.objs.zbuf ()
        self.pub.include (buf, "/get_user.html",
                          { "name" : res.name, "age" : res.age } )
        self.output (buf)

    def process (self):
        uid = self.cgi.lookup (key="uid")
        self.service.db.call (user.GET_USER, int (uid), self.cb)

class example_service (service):

    def make_newclient (self):
        return example_client ()

    def __init__ (self, servicename, parameters):
        service.__init__ (self, servicename, parameters)
        self.db = self.add_db (port=30888, host="127.0.0.1",
                               prog=user.user_prog_1 ())

svc = example_service (sys.argv[0], sys.argv[1])
svc.launch ()
sfs.core.amain ()
```

Figure 7.2: An example OKWS service implemented in Python. No exceptions are caught for brevity.

3. *okld* changes the owner of the Python script to `root`, its group to x, and its access mode to `0440`.

4. *okld* sets its user ID to x after forking, but before executing the Python interpreter with the required script.

Thus, OKWS's approach in the case of Python scripts mimics its approach for C++ scripts seen in Section 5.1. An unfortunate complication is that a fresh copy of the interpreter must be made for each script.

7.4 Jailing Python

Coercing the Python interpreter to run inside a `chroot` jail is an onerous task. All Python libraries and the shared libraries they link against must be copied over. The Python interpreter itself requires many shared libraries, some of which it `dlopens` and hence are not reported by running `ldd` on the executable. To locate all shared libraries Python needs, one must examine Python's text segment while the interpreter is running, perhaps using the `lsof` utility [15]. Finally, up-to-date versions of SFS/OKWS Python glue and compiled Python XDR modules must also be copied into OKWS's jail.

The Python 2.4 distribution and its required libraries consumes about 100 MB of disk space in the jail on FreeBSD 5.4. The Python interpreter itself, copied over once per service, consumes about 1MB of disk space. Appendix B details the mundane mechanics of running Python in a `chroot` environment. This code gives a good flavor of Unix's hostility toward secure application development.

Chapter 8

Performance Evaluation

OKWS design limits the process pool to a small and fixed size—one per service. To evaluate, we examined OKWS's performance as a function of the number of active service processes. We also present and test the claim that OKWS can achieve high throughputs relative to other Web servers because of its smaller process pool and its specialization for dynamic content.

8.1 Testing Method

Performance testing on Web servers usually involves the SPECweb99 benchmark [60], but this benchmark is not well-suited for dynamic Web servers that disable Keep-Alive connections and redirect to other machines for static content. We therefore devised a simple benchmark that better models serving dynamic content in real-world deployments, which we call the *null service benchmark*. For each of the platforms tested, we implemented a *null service*, which takes an integer input from a client, makes a database SELECT on the basis of that input, and returns the result in a short HTML response (see Figure 8.1). Test clients make one request per connection: they connect to the server, supply a randomly chosen query, receive the server's response, and then disconnect.

```
<html><head><title>Test Result</title></head>
<body>
<?
  $db = mysql_pconnect("okdb.lcs.mit.edu");
  mysql_select_db("testdb", $db);
  $id = $HTTP_GET_VARS["id"];
  $qry = "SELECT x,y FROM tab WHERE x=$id";
  $result = mysql_query("$qry", $db);
  $myrow = mysql_fetch_row($result);
  print("QRY $id $myrow[0] $myrow[1]\n");
?>
</body>
</html>
```

Figure 8.1: PHP version of the null service

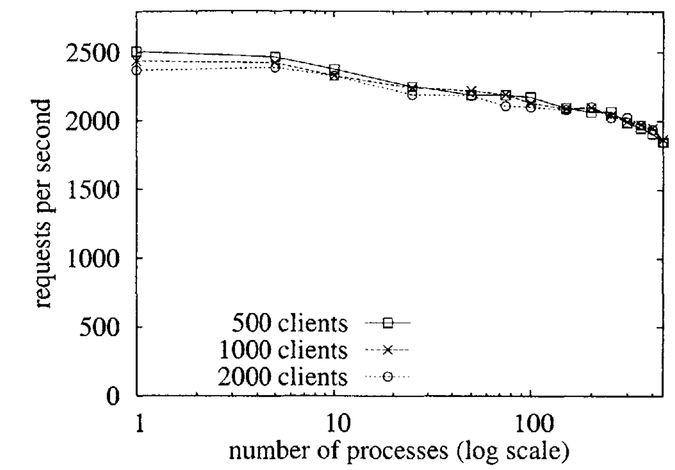

Figure 8.2: Throughputs achieved in the process pool test

8.2 Experimental Setup

All Web servers tested use a large database table filled with sequential integer keys and their 20-byte SHA-1 hashes [23]. We constrained our client to query only the first 1,000,000 rows of this table, so that the database could store the entire dataset in memory. The database used was MySQL version 4.0.16.

All experiments used four FreeBSD 4.8 machines. The Web server and database machines were uniprocessor 2.4GHz and 2.6GHz Pentium 4s respectively, each with 1GB of RAM. Our two client machines ran Dual 3.0GHz Pentium 4s with 2GB of RAM. All machines were connected via fast Ethernet, and there was no network congestion during our experiments. Ping times between the clients and the Web server measured around 250 μs, and ping times between the Web server and database machine measured about 150 μs.

The test clients uses the OKWS and SFS libraries. There was no resource strain on the client machines during our tests.

8.3 OKWS Process Pool Tests

We experimentally validated OKWS's frugal process allocation strategy by showing that the alternative—running many processes per service—performs worse. We thus configured OKWS to run a single C++ service as a variable number of processes, and collected throughput measurements (in requests per second) over the different configurations. The test client simulated either 500, 1,000 or 2,000 concurrent remote clients in the different runs of the experiment.

Figure 8.2 summarizes the results of this experiment as the number of processes varied between 1 and 450. We attribute the general decline in performance to increased context-switching, as shown in Figure 8.3. In the single-process configuration, the operating system must switch between the null service and *okd*, the demultiplexing daemon. In this configuration, higher client concurrency implies fewer switches, since both *okd* and the null service

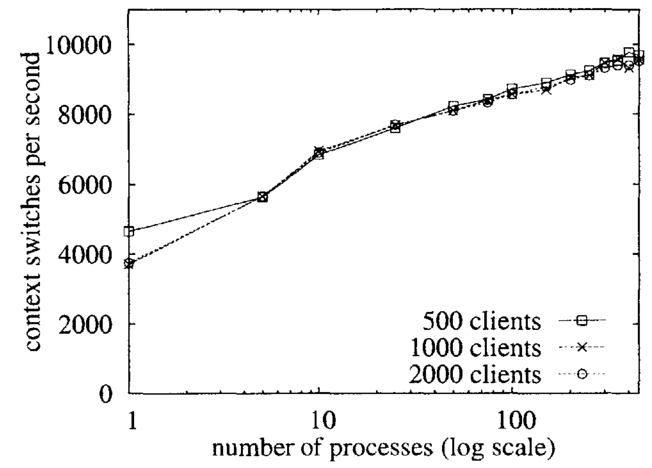

Figure 8.3: Context switching in the process pool test

have more outstanding requests to service before calling *sleep*. This effect quickly disappears as the server distributes requests over more processes. As their numbers grow, each process has, on average, fewer requests to service per unit of time, and therefore calls *sleep* sooner within its CPU slice.

The process pool test supports our hypothesis that a Web server will consume more *computational* resources as its process pool grows. Although the experiments completed without putting *memory* pressure on the operating system, memory is more scarce in real deployments. The null service requires about 1.5MB of core memory, but our experience shows real OKWS service processes have memory footprints of at least 4MB, and hence we expect memory to limit server pool size. Moreover, in real deployments there is less memory to waste on code text, since in-memory caches on the Web services are crucial to good site performance and should be allowed to grow as big as possible.

8.4 Web Server Comparison

The other Web servers mentioned in Section 4.3—Haboob, Flash and Apache—are primarily intended for serving static Web pages. Because we have designed and tuned OKWS for an entirely dynamic workload, we hypothesize that when servicing such workloads, it performs better than its more general-purpose peers. The experiments in this section test this hypothesis.

Haboob is Java-based, and we compiled and ran it with FreeBSD's native JDK, version 1.3. We tested Flash v0.1a, built with `FD_SETSIZE` set high so that Flash reported an ability to service 5116 simultaneous connections. Also tested was Apache version 2.0.47 compiled with multi-threading support and running PHP version 4.3.3 as a dynamic shared object. We configured Apache to handle up to 2000 concurrent connections. OKWS ran in its standard configuration, with a one-to-one correspondence between processes and services. All servers enabled HTTP access logging, except for Haboob, which does not support it.

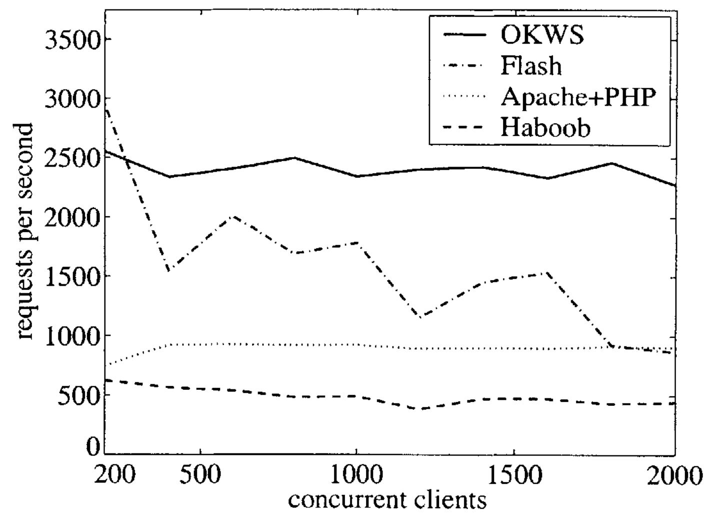

Figure 8.4: Throughputs for the single-service test

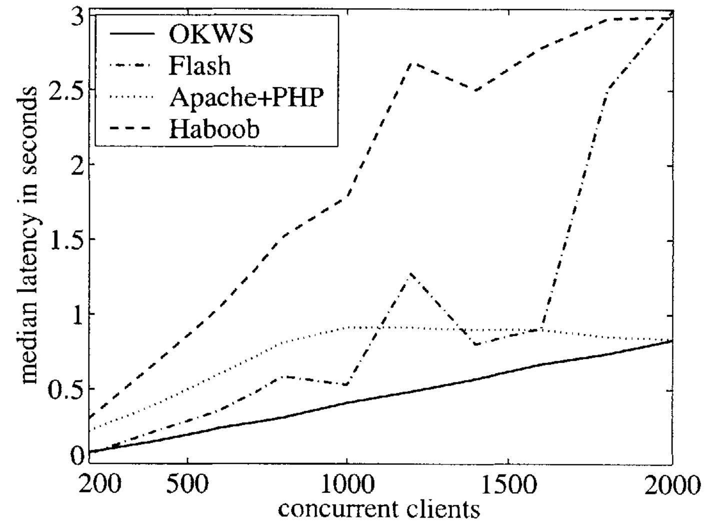

Figure 8.5: Median latencies in the single-service test

8.4.1 Single-Service Workload

In the single-service workload, clients with negligible latency request a dynamically generated response from the null service. This test entails the minimal number of service processes for OKWS and Flash and therefore should allow them to exhibit maximal throughput. By contrast, Apache and Haboob's process pools do not vary in size with the number of available services. We examined the throughput (Figure 8.4) and responsiveness (Figure 8.5) of the four systems as client concurrency increased. Figure 8.6 shows the cumulative distribution of client latencies when 1,600 were active concurrently. Of the four Web servers tested, Haboob spent the most CPU time in user mode and performed the slowest. A likely cause is the sluggishness of Java 1.3's memory management.

When serving a small number of concurrent clients, the Flash system outperforms the others; however, its performance does not scale well. We attribute this degradation to

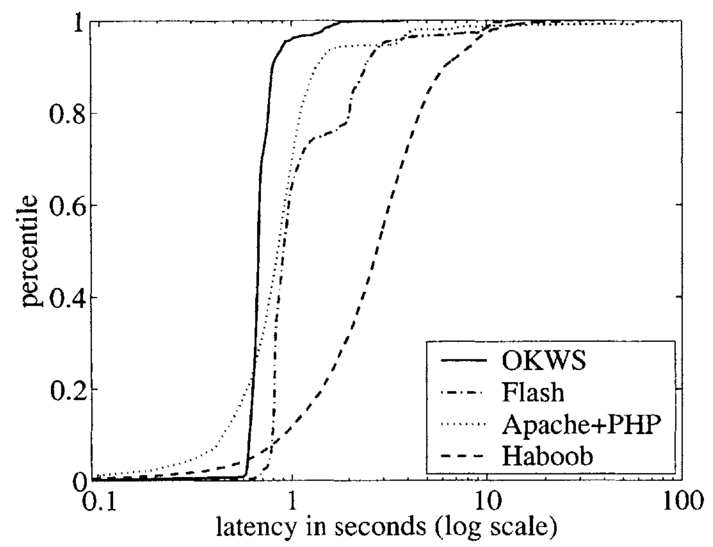

Figure 8.6: Client latencies for 1,600 concurrent clients in the single-service test

Flash's CGI model: because custom-written Flash helper processes have only one thread of control, each instantiation of a helper process can handle only one external client. Thus, Flash requires a separate helper process for each external client served. At high concurrency levels, Flash ran a large number of processes (on the order of 2000), starving itself of required OS resources. Flash also puts additional strain on the database, demanding one active connection per helper—thousands in total. A database pooling system might mitigate this negative performance impact. Flash's results were noisy in general, and we can best explain the observed non-monotonicity as inconsistent operating system (and database) behavior under heavy strain.

Apache achieves 37% of OKWS's throughput on average. Its process pool is bigger and hence requires more frequent context switching. When servicing 1,000 concurrent clients, Apache runs around 450 processes, and context switches about 7500 times a second. We suspect that Apache starts queuing requests unfairly above 1,000 concurrent connections, as suggested by the plateau in Figure 8.5 and the long tail in Figure 8.6.

Apache with PHP makes frequent calls to the *sigprocmask* system call to serialize database accesses among kernel threads within a process. In addition, Apache makes frequent (and unnecessary) file system accesses, which though serviced from the buffer cache still entail system call overhead. OKWS can achieve faster performance because of a smaller process pool and fewer system calls.

8.4.2 Many-Service Workload

In attempt to model a more realistic workload, we investigated Web servers running more services, serving more data, as experienced by clients over the WAN. Null services on all platforms were modified to send out an additional 3000 bytes of text with every reply (larger responses would have saturated the Web server's access link in some cases). We made ten uniquely-named copies of the new null service to model ten distinct services. Finally, the client was modified to pause an average of 75 ms between establishing a connection and

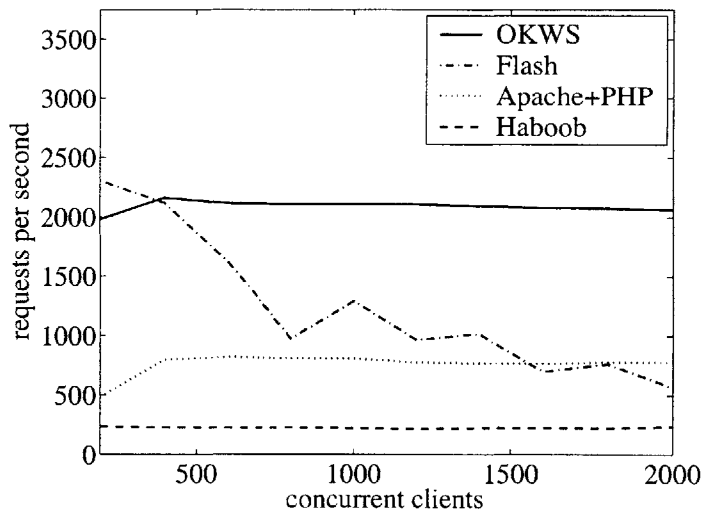

Figure 8.7: Throughputs for the many-service workload

	Haboob	Apache	Flash	OKWS
1 Service	490	895	1,590	2,401
10 Services	225	760	1,232	2,089
Change	−54.0%	−15.1%	−22.5%	−13.0%

Table 8.1: Average throughputs in connections per second

sending an HTTP request.

Figure 8.7 shows the achieved throughputs, and Table 8.1 compares these results to those results observed in the single-service workload. Haboob's performance degrades most notably, probably because the many-service workload demands more memory allocations. Flash's throughput decreases by 23%. We observed that for this workload, Flash requires even more service processes, and at times over 2,500 were running. When we switched from the single-service to the many-service configuration, the number of OKWS service processes increased from 1 to 10. The results from Figure 8.2 show this change has little impact on throughput.

Chapter 9

Security Analysis

9.1 Security Benefits

The security of OKWS is evaluated in terms of its implementation of the design goals in Chapter 4.

Design Goal 1: Minimal Filesystem Privileges. An OKWS service has almost no access to the file system when execution reaches site-specific service code. If compromised, a service has write access to its coredump directory and can read from OKWS shared libraries. Otherwise, it cannot access `setuid` executables, the binaries of other OKWS services, or core dumps left behind by crashed OKWS processes. It cannot overwrite HTTP logs or HTML templates. Other OKWS services such as *oklogd* and *pubd* have more privileges, enabling them to write to and read from the file system, respectively. However, as OKWS matures, these helpers should not present security risks since they do not run site-specific code.

Design Goals 2 and 4: Separation of Privileges. Because OKWS runs logically separate processes under different user IDs, compromised processes (with the exception of *okld*) do not have the ability to *kill* or *ptrace* other running processes. Similarly, no process save for *okld* can bind to privileged ports.

OKWS is careful to separate the traditionally "buggy" aspects of Web servers from the most sensitive areas of the system. In particular, those processes that do the majority of HTTP parsing (the OKWS services) have the fewest privileges. By the same logic, *okld*, which runs as superuser, does no message parsing; it responds only to signals. For the other helper processes, we believe the RPC communication channels to be less error-prone than standard HTTP messaging and unlikely to allow intruders to traverse process boundaries.

Process isolation also limits the scope of those DoS attacks that exploit bugs in site-specific logic. Since the operating system sets per-process limits on resources such as file descriptors and memory, DoS vulnerabilities should not spread across process boundaries. We could make stronger DoS guarantees by adapting "defensive programming" techniques [51]. Qie *et al.* suggest compiling rate-control mechanisms into network services, for dynamic prevention of DoS attacks. Their system is applicable within OKWS's architecture, which relegates each service to a single address space. The same cannot be said for those systems that spread equivalent functions across multiple address spaces.

Design Goal 3: Restricted Database Access. As described, all database access in OKWS is achieved through RPC channels, using independent authentication tokens. As a result, the attacker does not gain generic SQL Client access but instead gains access in the manner specified by an RPC protocol declaration. This is a stronger restriction than simple database permission systems alone can guarantee. For instance, on PHP systems, a particular service might only have SELECT permissions to a database's USERS table. But with control of the PHP server, an attacker could still issue commands like SELECT * FROM USERS. With OKWS, if the RPC protocol restricts access to row-wise queries and the keyspace of the table is sparse, the attacker has significantly more difficulty "mining" the database.[1]

OKWS's separation of code and privileges further limits attacks. If a service is compromised, it can attempt a new connection to any remote RPC database proxy, but it will have little success. Because the service has no access to source code, binaries, or *ptraces* of other services, it knows no authentication tokens aside from its own.

Finally, OKWS database libraries provide runtime checks to ensure that SQL queries can be prepared only when a proxy starts up, and that all parameters passed to queries are appropriately escaped. This check insulates sloppy programmers from the "SQL injection" attacks mentioned in Section 3.2. We expect future versions of OKWS to enforce the same invariants at compile time.

9.2 Security Shortcomings

The current implementation of OKWS supports C++ for service development. OKWS programmers should use the provided "safe" strings classes when generating HTML output, and they should use only auto-generated RPC stubs for network communication. however, OKWS does not prohibit programmers from using unsafe programming techniques and can therefore be made susceptible to buffer overruns and stack-smashing attacks. OKWS programmers can choose Python services if they fear vulnerabilities commonly associated with lower-level languages.

However, higher-level languages are no cure-all for OKWS or any other Web server. Many high-performance implementations of Python, Perl, PHP and Java use lower-level libraries written in C or C++. Errors in these libraries can make the programs that link against them vulnerable. For example, a recent bug in `java.util.zip` pushes some Java Virtual Machine implementations [11] to crash or exhibit undefined behavior. A recent bug in `zlib` shows that many higher-level languages (like Python, Perl and PHP) are similarly vulnerable [8].

Another shortcoming of OKWS is that an adversary who compromises an OKWS service can gain access to in-memory state belonging to other users. Developers might protect against this attack by encrypting cache entries with a private key stored in an HTTP cookie

[1]Similar security properties are possible with a standard Web server and a database that supports stored procedures, views, and roles.

on the client's machine. Encryption cannot protect against an adversary who can compromise and passively monitor a Web server.

Finally, independent aspects of the system might be vulnerable due to a common bug in the core libraries.

Chapter 10

Discussion and Directions for Future Research

OKWS can be seen as a concerted effort to apply the principle of least privilege (POLP) to a Unix Web server. OKWS attempts to uphold POLP, but experience detailed here shows this attempt 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. As we draw lessons from our experience with OKWS, we can ask if changes to the operating system could allow programmers to hold fast to privilege separation without all of the unpleasant implementation details. Additionally, might better OS support address the security weaknesses discussed in the previous chapter? We consider some changes to the Unix operating system—in order of increasing severity—to answer these questions.

10.1 Restricting the Unix System Call Interface

One of the main difficulties with OKWS's 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 [7, 58] favor the additive one. But Unix file descriptors are in fact capabilities. By hobbling system calls sufficiently—either through system call interposition [26, 49] 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 Sections 5.1 and 7.3. By disabling all unneeded privileges, the operating system could enforce privilege separation by default.

A benefit of Unix's capability-like system calls is that they 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 recompile an untrustworthy process to interpose on it.

Unfortunately, most Unix system calls resist virtualization. Some system calls 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". Exact user-level emulation of these problematic calls—which include `open` and `accept`—is messy, if not impossible.

10.2 Alternatives to Unix

From above, we claim that a step in the right direction for secure applications is to hobble the richness of the Unix interface to expose only capability-based system calls to applications. Such a scheme would give OKWS a stronger assurance of isolation between its processes but might not simplify implementation due to bad support for virtualization.

An alternative to Unix is a hypothetical OS dubbed "Unestos" [33]. 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 [55] and Plan 9's 9P [45]: `LOOKUP`, `READ`, `WRITE`, and so forth. 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:

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

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.

10.2.1 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 [58] favor global, persistent naming, but persistence has proven cumbersome to kernel and application designers [57].

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 [46], 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 sends and receives messages to and from 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.

10.2.2 OKWS Under Unestos

We now consider what OKWS might look like on Unestos. Similar to before, the application suite consists of a *okld*, *okd* 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). *pubd* is no longer needed.

okld 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 *okd*'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 (which perhaps only exists in memory), 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 *okd*. 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.

In this hypothetical operating system, the interface exposed to applications feels like the familiar Unix namespace (with added flexibility for unprivileged, fine-grained jails). In reality, an application's system interactions are entirely defined by its capabilities, and Unestos behaves like a capability system for the purposes of security analysis.

10.3 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 separate services in the current OKWS. Though a user session involves many different processes (such as *okd*, databases and worker processes), a policy for separating users should be achievable with a few stanzas of privileged code, as opposed to hidden authorization checks scattered throughout the system. 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 [19].

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 [40]: 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 *event-processes*.

Under Asbestos, OKWS behaves as follows: *okd* 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, *okd* creates a compartment for U; if U is returning, then *okd* reassigns U to its previous compartment. It then forwards U's connection to the appropriate event-process of the appropriate worker. When handling U's request, the event-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 event-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 event-process, and speak on behalf of a different user. Finally, since *okd* 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.

OKWS implemented on Asbestos provides the end-to-end security properties that Web site administrators care most about: that the system cannot leak data from one user to another. Though site-specific service code is often in flux and touched by many developers, a correct Asbestos kernel guarantees that however flawed the service code is, large-scale data theft is very unlikely.

Chapter 11

Conclusion

This thesis described in detail the design and implementation of the OKWS system. The important results of this investigation are twofold. First, OKWS represents a significant improvement over the status quo in terms of security and performance, and its commercial uses shows that the system is no mere academic exercise. The key performance improvements in OKWS stem from its specialization for dynamic content. The key security improvements result from a design focused on separation and isolation, as opposed to an ex-post facto application of Unix security techniques to an otherwise monolithic and privileged application. OKWS has pushed as much as possible on the Unix interface to achieve useful security properties without sacrificing performance.

The second important result begins where OKWS falls short: a complex implementation and less-than-perfect security guarantees. If OKWS represents a concerted effort to build a secure Unix application from scratch but still cannot provide needed end-to-end security guarantees (such as keeping Alice from Bob's data and vice-versa), perhaps it is time to look beyond Unix to an operating system whose API is friendlier to secure application developers. Using the lessons from OKWS, this thesis proposes a rough sketch of what that operating system (dubbed "Asbestos") might look like: a system in which privileges are explicitly added, not assumed and then subtracted; in which all user-kernel and user-user interactions involve capabilities, or explicit rights; in which systems calls are virtualizable, so that user-user communication resembles user-kernel communication; in which IPC is auditable; in which data can be tracked as it flows through the system.

Availability

Visit `www.okws.org` for documentation and source code, available under a GPL license.

Appendix A

Speeding Up *gzip* Compression for Dynamic Webservers

Most currently active Web clients such as Mozilla/Firefox and Microsoft Internet Explorer support HTTP/1.1 *gzip* compression [18, 22]. A Web server can therefore transparently send HTML output to a client compressed via *gzip*. Older clients are still supported, since newer clients advertise their ability to accept gzip-encoded transfers in their initial request. There are obvious advantages for both client and server; they both conserve bandwidth and they both enjoy lower latency. Moreover, since network bandwidth is expensive relative to computational power, a cost-effective Web server should compress its output whenever possible.

Compression of *static* HTML content on the server side is easy to imagine: the server simply maintains two copies of each document — one compressed, and one expanded— and answers client requests appropriately. For *dynamic* HTML content, the state-of-the art is for the Web server to naïvely compress each response on-the-fly. Compression is not cheap in terms of computation, and in many cases, compressing each response on the fly significantly curtails server throughput. However, given a simple observation—that even *dynamically* generated HTML documents are often composed primarily of *static* HTML components— there is an opportunity for a Web server to cache work spent on compression, and to stitch several cached components together for each request. We develop this idea in more detail by first describing the generic *gzip* algorithm, and then by proposing a cacheable implementation thereof.

A.1 The *gzip* Algorithm

gzip [18] is a 2-layer compression scheme. The top layer uses LZ77-compression [80], that looks for repeated substrings within a document. When the compressor finds a second instance of a string, it inserts a "backpointer" that points back to the original instance. If it finds multiple earlier instances of the same string, it selects the instance that yields the *longest* possible match — in general, this heuristic improves compression ratios. A requirement of the standard *gzip* decompressors built into most browsers is that these backpointers can point no more than 32K characters into the past. This requirement helps clients decom-

```
<html>
 <head>
  <title>Foo Page</title>
 <head>
<body>
 <? for ($i = 0; $i < 10; $i++)
   print "bar $i ";
 ?>
</body>
</html>
```

Figure A.1: A simple fragment of PHP code.

press with only a 32K memory window. If the compressor finds no earlier instances of a string, it simply outputs the character itself. At the bottom layer, *gzip* feeds all backpointers and non-matches (*i.e.*, single characters) through a Huffman encoder. It tacks on a 32-bit CRC checksum, and the compression is complete.

A naïve implementation of *gzip* checks every character position against a number of character positions linear in the size of the scan window. The standard *gzip* tool uses this method, in concert with 3-character Rabin fingerprinting.[1] Though aggressively optimized, the standard *gzip* implementation[2] absorbs ASCII text at about 19MB/sec and outputs compressed data at about 4MB/sec on a 2.4 GHz Pentium IV. A call graph profile shows that only 10% of computational resources are dedicated to Huffman encoding, while LZ77 computation occupies a large majority of the remaining cycles.

A.2 Cacheable Compression

An improvement on the standard *gzip* algorithm is *cacheable gzip compression*. The goal of this algorithm is to speed up the LZ77 phase of *gzip* compression, by performing potentially expensive precomputations on the HTML fragments that are likely to recur. The first idea is that a Web server splits each outgoing Web response into a series of texts:

$$\mathcal{R} = (S_1, M_1, S_2, M_2, \ldots M_n, S_n)$$

A text M_i is a *mutable* text, and might be unique to a given HTML response. A text S_i is a *static* text, and is likely to repeat in many HTML responses. Given enough access to the internals of one's Web server, one can readily tell which texts are static, and which are mutable, without expensive techniques such as "shingling." For instance, a typical PHP script [44] might look something like Figure A.1. The "static" text in this example is all of the HTML not within the special PHP tags (given by `<?`... `?>`). All of the other data is mutable. Thus, with sufficient integration with PHP's internals, a compression filter can

[1]The same is true for the `zlib` library [24] used by most popular Web servers.

[2]`gzip-1.2.4`, compiled with `gcc-2.95.4`

```
Cacheable-LZ77 on input $(\mathcal{R} = (S_1, M_1, \ldots M_n, S_n))$
1  foreach $X \in \mathcal{R}$ do
2      If $X$ is static, then append $X$ to $\mathcal{W}$.
3      Trim $\mathcal{W}$ so that its total length is less than 32K
4      $o \leftarrow 1$
5      while $o \leq ||X||$ do
6          $\langle j, o^*, l \rangle \leftarrow$ LongestMatch$(X, o, \mathcal{W})$
7          $p \leftarrow o - o^* + ||S_n|| + ||M_n|| + \cdots + ||M_{j+1}|| + ||S_j||$
8          Output backpointer $\langle p, l \rangle$
9          $o \leftarrow o + l$
10     done
11 done
```

Figure A.2: A simple, incomplete *cacheable LZ77* algorithm. Several important corner cases are elided for clarity. The notation $||M_n||$ gives the length of string M_n.

receive input in the form given above — a series of demarcated static and mutable texts.

It is now possible to formulate a rough approximation of the proposed cacheable compression algorithm. Given any text — mutable or static — called X. Assume the existence of a routine LongestMatch, that takes as input the text X, a starting offset o into that text, and window of texts $\mathcal{W} = (S_1, \ldots, S_m)$ to compare X against. The routine returns a triple $\langle j, o^*, l \rangle$, with the property that $S_j[o^*, \ldots, o^* + l - 1] = X[o, \ldots, o + l - 1]$. That is, the substring of text S_j, starting at offset o^* of l characters long, exactly matches the given text X, starting at offset o, for l characters. Moreover, this l is maximal for the given context window $\mathcal{W}$.[3] We assume for simplicity that LongestMatch$(X, o, (X))$ is handled properly — that is, all offsets o^* output are bounded below o. This is a natural constraint, since *gzip* does not accommodate forward pointers.

A simple cacheable LZ77 algorithm is shown in Figure A.2, which makes use of the abstract algorithm LongestMatch just defined. Note that only static blocks are considered as part of the compression window $\mathcal{W}$. Assuming that *static* content composes a majority of *dynamic* HTML pages, this omission has a minimal effect on compression ratios. Also note that Figure A.2 elides certain corner cases for clarity — such as offsets without matches and comparing a block X against itself.

By far, the most expensive part of the Cacheable-LZ77 algorithm are the repeated calls to LongestMatch on line 6. Thus, the two important performance goals are to call LongestMatch as infrequently as possible (through cache and reuse of earlier return values) and to make LongestMatch as inexpensive as possible. In pursuit of the former goal, a real implementation of Cacheable-LZ77 should cache the results of all calls to LongestMatch for static content blocks S_i. That is, when the first argument to LongestMatch is a static

[3] If two equally long matches are found, the string further right is preferred, since it reduces backpointer length.

block S_i, all of the arguments are likely to be seen again: (1) the set of all possible static blocks S is assumed to be small for a given Web site (on the order of 1,000) and that all blocks are known to the Web server at boot time; (2) the context window $\mathcal{W}$ is a subset of $\mathcal{S}$, and only a small number of these subsets are used in practice; and (3) the offset o follows a predictable pattern based on S_i and $\mathcal{W}$, since the LongestMatch computation is deterministic. Furthermore, the output of LongestMatch does not depend on the ordering of texts within the compression window $\mathcal{W}$. Thus, Web servers can reorder static blocks in HTML responses, without diminishing their ability to cache compression.

The mutable texts M_i do not enjoy the same cacheability as the static texts S_i, as they recur infrequently, if at all. For compression of mutable texts, servers have no choice but to compute LongestMatch as efficiently as possible. Efficient LongestMatch computation via suffix trees is the ultimate goal for achieving fast, cacheable compression. The challenge is to do so without hoarding memory, which is scarce on busy Web servers. We leave this challenge to future work.

A.3 Related Work

There is a rich body of literature dedicated to suffix trees, and compressing their in-memory representations. Gusfield's book [27] provides an overview of the state-of-the-art as of 1997. Since then, Kurtz [34] has discovered an efficient implementation for single texts that consumes $10m$ memory for ASCII texts on average. It is not readily apparent, however, how to map these techniques to trees for multiple texts. Other scheme such as suffix arrays [36] and suffix binary search trees [29] trade-off lookup speed for memory savings. Suffix arrays do not meet our requirements, as there is not an efficient way to phase out the compression window $\mathcal{W}$ while traversing the suffix array. If specialized for cacheable *gzip* compression, suffix binary search trees would most likely incur the same overhead of storing the table D.

Another approach is to compress suffix trees into *directed acyclic word graphs*, known as DAWGs [6]. The idea is that if a node x has a suffix link to a node y, and the subtrees of x and y are isomorphic, then there is no need to store both trees; rather, the suffix tree can contain an edge from x to y, breaking the tree property of the graph. In the context of multiple texts though, this compression technique is of little use — one will see very few such subtree isomorphisms in practice.

Other work has addressed the *gzip* compression problem directly. Larsson discovered a construction for sliding window suffix trees with direct application to LZ77 compression [35]. Later, Sadakane and Imai argued that this construct is not practical in terms of speed, memory usage, or compression ratios. Instead, they suggest hashing and sorting techniques based on suffix arrays [53].

Another approach to the problem is to modify the HTTP standard and client browsers, to minimize the data sent over and active HTTP session [5, 17]. While most likely more efficient in terms of compression and server resources than the *gzip* method discussed here, these approaches face obvious deployment barriers; our proposed scheme, by contrast, is transparent to clients.

Appendix B

Jailing Python for OKWS

What follows is the actual code needed to make Python run inside of OKWS's jail. Though jailing Python is possible, it does not seem natural on the Unix platform.

```
"""PyOKWS Jail Utilities

Copy the system Python and all associated libraries into the
OKWS runtime jail.
"""

import sys
import os
import re
import os.path
import string

from okws.setup.config import ConfigParser

# for reading python configuration stuff, and also for copying
# files and stuff over from the global directories to inside the jail.
from distutils.dir_util import mkpath, copy_tree
from distutils.file_util import copy_file, write_file
from distutils.sysconfig import get_python_lib

__version__ = 0.1

def find_files_by_ext (root, ext):
    """Find files with the given extensions, recursively, starting with
    the given directory root"""
    from os import listdir
    from os.path import join, isdir, islink, splitext
    ls = listdir (root)
    files = []

    # add the leading '.' if not already there
    if ext[0] != ".":
        ext = "." + ext;

    for f in ls:
        fullname = join (root, f)
        if isdir (fullname):
            files += find_files_by_ext (fullname, ext)
        elif (splitext (fullname))[1] == ext:
            files.append (fullname)
            pass
        pass
    return files

#find_file_by_ext ()

def ldd (file):
    """Given a dynamically linked executable, collect a list of all
    .so files that it links to."""
    from os import popen
    p = popen ("ldd " + file, "r")
    rxx = re.compile ("(\S+) => (\S+)")
    sofiles = []
    line = p.readline ()
    while line != "":
        m = rxx.search (line)
        if m:
            sofiles.append (m.group (2))
            pass
        line = p.readline ()
        pass
    return sofiles

#ldd ()


def my_join (a,b):
    import os.path
    s = os.path.sep # '/' for mostly anything reasonable ;)
    if a[-1] == s:
        if b[0] != s:
            ret = a + b
        else:
            ret = a + b[1:]
    else:
        if b[0] == s:
            ret = a + b
        else:
            ret = a + s + b
    return ret

#
# call lsof to figure out which dynamic libraries python loaded
# dynamically.   oh bother, what a pain.
#
# XXX - this is probably platform specific. bummer.
#
def lsof_me ():
    import os
    p = os.popen ("lsof", "r")
    # the first line tells us the name of the columns (i hope...)
    line = p.readline ()
    cols = line.split ()

    # make an index of column names
    col_names = {}
    for i in range (0, len (cols)):
        col_names[cols[i]] = i
        pass

    ret = []
    while line != "":
        s = line.split ()
        if s[col_names["COMMAND"]] == "python" and \
           s[col_names["FD"]] == "txt":
            ret.append (s[col_names["NAME"]])
            pass
        line = p.readline ()
        pass
    return ret


class Jailer:
    """A class that takes care of moving Python's world into a given
    chroot jail. All specification of jails and options is given with
    the initilization function."""

    def __init__ (self, jaildir=None, okws_config=None, servicebin=None,
                  verbose=False, dry_run=False):

        self.jaildir = jaildir
        self.okws_config = okws_config
        self.servicebin= servicebin
        self.verbose = verbose
        self.dry_run = dry_run

        self._config ()
        return
    # __init__()

    def _config (self):

        attrs =   [ "jaildir", "servicebin"]

        for a in attrs:
            if getattr (self, a) is None:

                # reads an okws_config file, taking the standard
                # one at /etc/okws_config if given parameter
                # 'None'
                p = ConfigParser (self.okws_config)
                (_, self.cfg_file_nvtab) = p.parse ()
                break
            pass

        # fill in missing v
        for a in attrs:
            if getattr (self, a) is None:
                setattr (self, a, self.cfg_file_nvtab[a])
                pass
            pass
        return
    # config ()

    def get_lib_dirs (self):
        return [ get_python_lib (standard_lib=True) ]
    # get_lib_dirs ()

    def run (self):
        """Given an initialized OKWS/Python Jailer, run it and actually
        do the require FS operations."""
        dst = self.jaildir
        srcs = self.get_lib_dirs ()
        sos = []
        done = {} # keep track of those already done

        # copy the trees and also find all .so files in the trees
        for s in self.get_lib_dirs ():
            if done.get (s):
                continue
            else:
                done[s] = True

            copy_tree (src=s, dst=my_join(dst, s),
                       verbose=self.verbose,
                       dry_run=self.dry_run)
            sos += find_files_by_ext (s, "so")
            pass

        # keep track of all .so files on the system the python
        # .so files link to, and watch out for duplicates
        solibs =   {}

        for s in sos:
            for l in ldd (s):
                solibs[l]= True
            pass

        # also, don't forget about the python executable itself!
        for l in ldd (sys.executable):
            solibs[l] = True
            pass

        # also, don't forget about those libraries python loaded
        # dynamically that it didn't tell the dynamic linker about!
        for l in lsof_me ():
            solibs[l] = True
            pass

        for s in solibs.keys ():
            if not os.path.isfile (s):
                print "Skipping non-file (did lsof misreport?): " + s
            else:
                (d,f) = (os.path.dirname (s), os.path.basename (s))
                mkpath (name=my_join (dst,d), verbose=self.verbose,
                        dry_run=self.dry_run)
                copy_file (src=s, dst=my_join (dst, s),
                           verbose=self.verbose, dry_run=self.dry_run)

        # copy over the python executable, too
        d = os.path.dirname (sys.executable)
        jail_bin_dir = my_join (dst, d)
        mkpath (name=jail_bin_dir, verbose=self.verbose)
        copy_file (src=sys.executable, dst=jail_bin_dir, verbose=self.verbose)
    # run ()

# Run the jailer with default arguments
j = Jailer ()
j.run ()
```

Bibliography

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proceedings of the 2002 USENIX*, Monterey, CA, June 2002. USENIX.

[2] Akamai Technologies, Inc. `http://www.akamai.com`.

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

[4] Apache Tutorial: Introduction to Server Side Includes. `http://httpd.apache.org/docs/howto/ssi.html`.

[5] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *1997 USENIX Annual Technical Conference*, pages 289–303, 1997.

[6] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas J. The smallest automaton recognizing the subwords of a text, 1985.

[7] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *USENIX Workshop on Microkernels and Other Kernel Architectures*. USENIX, 1992.

[8] Bugtraq ID 14162. SecurityFocus. `http://www.securityfocus.com/bid/14162/info`.

[9] Bugtraq ID 4606. SecurityFocus. `http://www.securityfocus.com/bid/4606/info/`.

[10] Bugtraq ID 5993. SecurityFocus. `http://www.securityfocus.com/bid/5993/info/`.

[11] Bugtraq ID 7109. SecurityFocus. `http://www.securityfocus.com/bid/7109/info/`.

[12] Bugtraq ID 7255. SecurityFocus. `http://www.securityfocus.com/bid/7255/info/`.

[13] Bugtraq ID 7768. SecurityFocus. `http://www.securityfocus.com/bid/7768/info/`.

[14] Bugtraq ID 8138. SecurityFocus.
`http://www.securityfocus.com/bid/8138/info/`.

[15] Gary Burd. Running python CGI in OpenBSD httpd chroot jail.
`http://burd.info/gary/2003/11/running-python-cgi-in-openbsd-httpd.html`.

[16] CERT® Coordination Center. `http://www.cert.org`.

[17] Mun Choon Chan and Thomas Y. C. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *INFOCOM*, pages 117–125, 1999.

[18] Peter Deutsch. *Gzip File Format Specification Version 4.3.* Internet Network Working Group RFC 1952, 1996.

[19] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[20] Open Market. Fastcgi. `http://www.fastcgi.com`.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol — HTTP/1.1.* Internet Network Working Group RFC 2616, 1999.

[22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol — HTTP/1.1.* Internet Network Working Group RFC 2616, 1999.

[23] FIPS 180-1. *Secure Hash Standard.* U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[24] Jean-Loup Gailly and Greg Roelofs. zlib, 1993. `http://www.gzip.org/zlib`.

[25] Thomas Gchwind and Benjamin A. Schmit. CSE — a C++ servlet environment for high-performance web applications. In *Proceedings of the FREENIX Track: 2003 USENIX Technical Conference*, San Antonio, TX, 2003. USENIX.

[26] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

[27] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[28] IBM corporation. IBM websphere application server. `http://www.ibm.com`.

[29] Robert W. Irving and Lorna Love. The suffix binary search tree and suffix AVL tree. *Journal of Discrete Algorithms*, 1(5-6):387–408, 2003.

[30] JBoss Group. `http://www.jboss.org`.

[31] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997. ACM.

[32] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003. ACM.

[33] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (to appear)*, Santa Fe, NM, June 2005.

[34] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.

[35] N. Jesper Larsson. Extended application of suffix trees to data compression. In J. A. Storer and M. Cohn, editors, *Proceedings of the Data Compression Conference*, pages 190–199, Snowbird, UT, 1996. IEEE Computer Society Press.

[36] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computation*, 22(5):935–948, 1993.

[37] David Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX*. USENIX, June 2001.

[38] Microsoft Corporation. IIS. `http://www.microsoft.com/windowsserver2003/iis/default.mspx`.

[39] mod_perl. `http://perl.apache.org`.

[40] Andrew C. Myers and Barbara 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.

[41] OkCupid.com. `http://www.okcupid.com`.

[42] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX*, Monterey, CA, June 1999. USENIX.

[43] Perl DBI. `http://dbi.perl.org`.

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

[45] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[46] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. In *Proceedings of the 5th ACM SIGOPS Workshop*, Mont Saint-Michel, 1992.

[47] Kevin Poulsen. Car shoppers' credit details exposed in bulk. *SecurityFocus*, September 2003. `http://www.securityfocus.com/news/7067`.

[48] Kevin Poulsen. Ftc investigates petco.com security hole. *SecurityFocus*, December 2003. `http://www.securityfocus.com/news/7581`.

[49] Niels Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, pages 257–271, Washington, DC, August 2003.

[50] The Python Programming Language. `http://www.python.org`.

[51] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, October 2002. USENIX.

[52] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[53] Kunihiko Sadakane and Hiroshi Imai. Improving the speed of LZ77 compression by hashing and suffix sorting. *TIEICE: IEICE Transactions on Communications/ Electronics/Information and Systems*, 2000.

[54] Jerome H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, 1975.

[55] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119–130, Portland, OR, 1985. USENIX.

[56] SecurityFocus. `http://www.securityfocus.com`.

[57] Jonathan S. Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a verified, general-purpose operating system kernel. In Gerwin Klein, editor, *Proc. NICTA Formal Methods Workshop on Operating Systems Verification*, Sydney, Australia, 2004. NICTA Technical Report 0401005T-1, National ICT Australia.

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

[59] The SparkMatch service. Previously available at http://www.thespark.com.

[60] Standard performance evaluation corporation. the specweb99 benchmark. http://www.spec99.org/osg/web99/.

[61] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Network Working Group, August 1995.

[62] J. Robert van Berhen, Eric A. Brewer, Nikita Borisova, Michael Chen an Matt Welsh, Josh MacDonald, Jeremy Lau, Steve Gribble, and David Culler. Ninja: A framework for network services. In *Proceedings of the 2002 USENIX*, Monterey, CA, June 2002. USENIX.

[63] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003. ACM.

[64] Vulnerability CAN-2001-1246. SecurityFocus. http://www.securityfocus.com/bid/2954/info/.

[65] Vulnerability CAN-2002-0656. SecurityFocus. http://www.securityfocus.com/bid/5363/info/.

[66] Vulnerability CAN-2003-0042. SecurityFocus. http://www.securityfocus.com/bid/6721/info/.

[67] Vulnerability CAN-2003-0132. SecurityFocus. http://www.securityfocus.com/bid/7254/info/.

[68] Vulnerability CAN-2003-0253. http://www.securityfocus.com/bid/8137/info/.

[69] Vulnerability CAN-2003-0542. SecurityFocus. http://www.securityfocus.com/bid/8911/info/.

[70] Vulnerability CAN-2004-0077. SecurityFocus. http://www.securityfocus.com/bid/9686/info.

[71] Vulnerability CVE-2002-0061. SecurityFocus. http://www.securityfocus.com/bid/4435/info/.

[72] Vulnerability Note VU117243. CERT. http://www.kb.cert.org/vuls/id/910713.

[73] Vulnerability Note VU208131. CERT. http://www.kb.cert.org/vuls/id/208131.

[74] Vulnerability Note VU91073. CERT. http://www.kb.cert.org/vuls/id/910713.

[75] Vulnerability Note VU979793. CERT.
`http://www.kb.cert.org/vuls/id/979793`.

[76] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001. ACM.

[77] Andrew Whitaker, Marianne Shaw, and Steve D. Gribble. Scale and performance in the denali isolation kernel. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, October 2002. USENIX.

[78] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX*, San Antonio, TX, June 2003. USENIX.

[79] Zeus Technology Limited. Zeus Web Server. `http://www.zeus.co.uk`.

[80] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[81] The Zope Corporation. `http://www.zope.org`.

[illegible]