Building Secure High-Performance Web Services with OKWS

Maxwell Krohn, *MIT Computer Science and AI Laboratory*
krohn@csail.mit.edu

Abstract

OKWS is a toolkit for building fast and secure Web services. It provides Web developers with a small set of tools that has proved powerful enough to build complex systems with limited effort. 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 [23] and Haboob [44]. Experience with OKWS in a commercial deployment suggests it can reduce hardware and system management costs, while providing security guarantees absent in current systems.

1 Introduction

Most dynamic Web sites today maintain large server-side databases, to which their users have limited access via HTTP interfaces. Keeping this data hidden and correct is critical yet difficult. Indeed, headlines are replete with stories of the damage and embarrassment remote attackers can visit on large Web sites.

Most attacks against Web sites exploit weaknesses in popular Web servers or bugs in custom application-level logic. 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. As a result, a flaw in one service can ripple through an entire system.

To plug the many security holes that plague existing Web servers, and to limit the severity of unforeseen problems, we introduce OKWS, the OK Web Server. Unlike typical Web servers, OKWS is specialized for dynamic content and is not well-suited to serving files from disk. It relies on existing Web servers, such as Flash [23] or Apache [3], to serve images and other static content. We argue (in Section 5.4) that this separation of static and dynamic content is natural and, moreover, contributes to security.

What OKWS does provide is a simple, powerful, and secure toolkit for building dynamic content pages (also known as *Web services*). OKWS enforces the natural principle of *least privilege* [27] so that those aspects of the system most vulnerable to attack are the least useful to attackers. Further, OKWS separates privileges so that the different components of the system distrust each other. Finally, the system distrusts the Web service developer, presuming him a sloppy programmer whose errors can cause significant damage. Though these principles are not novel, Web servers have not generally incorporated them.

Using OKWS to build Web services, we show that compromises among basic security principles, performance, and usability are unnecessary. To this effect, the next section surveys and categorizes attacks on Web servers, and Section 3 presents simple design principles that thwart them. Section 4 discusses OKWS's implementation of these principles, and Section 5 argues that the resulting system is practical for building large systems. Section 6 discusses the security achieved by the implementation, and Section 7 analyzes its performance, showing that OKWS's specialization for dynamic content helps it achieve better performance in simulated dynamic workloads than general purpose servers.

2 Brief Survey of Web Server Bugs

To justify our approach to dynamic Web server design, we briefly analyze the weaknesses of popular software packages. Our 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 exten-

sions 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 [14], Microsoft IIS [19] and Zeus [47].

2.1 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 [42]. A recent discovery of leaked file descriptors allows remote users to access sensitive log information [7]. 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 [41]. 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 many modules are common. Unchecked boundary conditions found recently in `mod_alias` and `mod_rewrite` regular expression code allow local attack [39]. 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 [36].

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 [40]. On MS-DOS-based systems, Apache failed to filter out special device names, allowing carefully-crafted HTTP POST requests to execute arbitrary code [43]. Other problems have occurred when site developers call Apache's `htdigest` utility from within CGI scripts to manage HTTP user authentication [6].

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 [38]. Another 2003 release allowed local configuration errors to result in infinite redirection loops [8]. In some versions of Apache, attackers could exhaust Apache's heap simply by sending a large sequence of linefeed characters [37].

2.2 Scripting Extensions to Apache

Apache's security worsens considerably when compiled with popular modules that enable dynamically-generated content such as PHP [25]. In the past two years alone, at least 13 critical buffer overruns have been found in the PHP core, some of which allowed attackers to remotely execute arbitrary code [9, 28]. 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`[5]. 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 [35].

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 could not run as an Apache module, and the system would require a new PHP process forked for every incoming connection. This isolation strategy is at odds with performance.

3 Design

If we assume that bugs like the ones discussed above 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 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 primary security goals are to prevent intrusion and to prevent unauthorized access to site data.

3.1 Practical Security Guidelines

(1) Server processes should be chroot*ed.* After compromising a server process, most attackers will try to gain control over the entire server machine, possibly by installing "back doors," learning local passwords or private keys, or probing local configuration files for errors. At the very least, a compromised Web server should have no access to sensitive files or directories. Moreover, an OS-level jail ought to hide all `setuid` executables from the Web server, since many privilege escalation attacks require such files (examples include the *ptrace* and *bind* attacks mentioned in [17]). Privilege escalation is possible without `setuid` executables but requires OS-level bugs or race conditions that are typically rarer.

An adversary can still do damage without control of the Web server machine. The configuration files, source files, and binaries that correspond to the currently running Web server contain valuable hints about how to access important data. For instance, PHP scripts often include the username and plaintext password used to gain access to a MySQL database. OS-enforced policy ought to hide these files from running Web servers.

(2) Server processes should run as unprivileged users. A compromised process running as a privileged user can do significant damage even from within a *chroot*ed environment. It might bind to a well-known network port. It might also interfere with other system processes, especially those associated with the Web server: it can trace their system calls or send them signals.

(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 functionality 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. By incorporating these principles, a Web server design assumes that processes will be compromised and therefore prevents uncompromised processes from performing unsafe operations, even when extended by careless Web developers. 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.

3.2 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 functionality for 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. 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 1 summarizes these relationships.

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

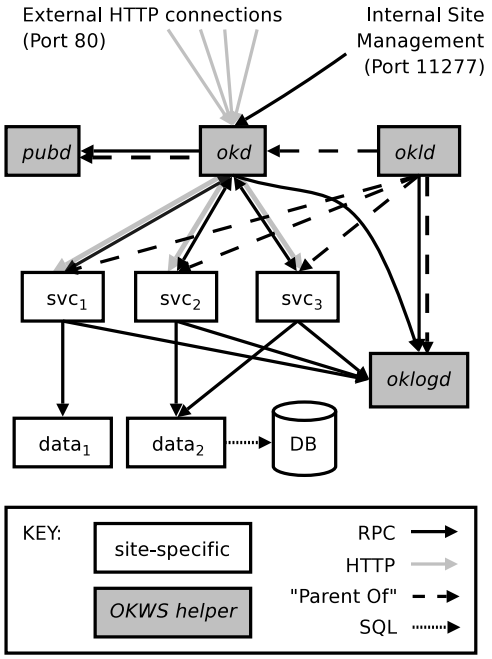


Figure 1: Block diagram of an OKWS site setup with three Web services (svc_1, svc_2, svc_3) and two data sources ($data_1$, $data_2$), one of which ($data_2$) is an OKWS database proxy.

(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.

3.3 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*; we now justify this selection.

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.

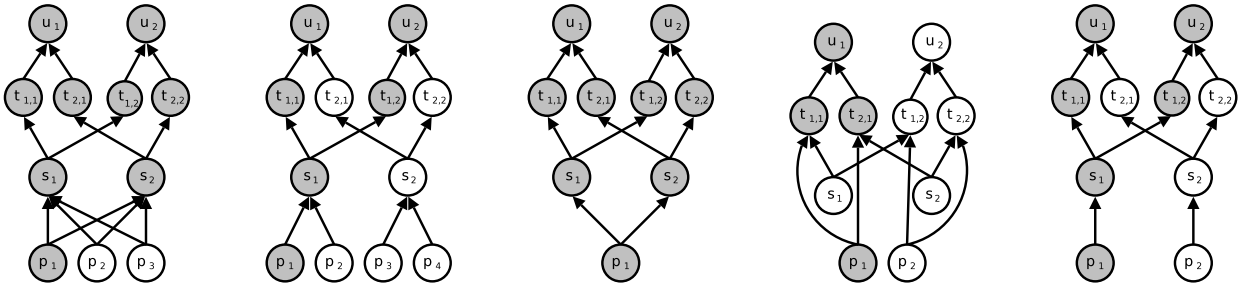
Let the set S represent a Web server's cons[illegible] services, and assume each service accesses a private pool of data. (Two application-level services that share data would thus be modelled by a single "service".) A set of users U interacts with these services, and the interaction between user u_j 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 Flash [3], each process in P is associated with a particular service: for each p_i, there exists s_j such 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 [44] employ only one process; thus $P = \{p_1\}$, and dependencies (p_1, s_j) exist for all j.

Figures 2(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 6.2. 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 2(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 compro-



(a) Apache (b) Flash (c) Haboob (d) Strict (e) OKWS

Figure 2: Dependency graphs for various Web server architectures.

mise 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. 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 2(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.

4 Implementation

OKWS is a portable, event-based system, written in C++ with the SFS toolkit [18]. It has been successfully tested on Linux and FreeBSD. In OKWS, the different helper processes and site-specific services shown in Figure 1 communicate among themselves with SFS's implementation of Sun RPC [32]; they communicate with external Web clients via HTTP. Unlike other event-based servers [23, 44, 47], OKWS exposes the event architecture to Web developers.

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 1 summarizes the runtime configuration of OKWS.

4.1 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`.

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.

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

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.

4.2 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 [11] 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.

Upon startup, *okd* reads the OKWS configuration file (`/etc/okws_config`) to construct its dispatch table. It inherits two file descriptors from *okld*: one for logging, and one for RPC control messages. *okd* then listens on the RPC channel for *okld* to send it the server side of the child services' HTTP and RPC connections (see Section 4.1, Step 6). *okd* receives one such pair for each service launched. The HTTP connection is the sink to which *okd* sends incoming HTTP requests from external clients after successful demultiplexing. Note that *okd* needs access to *oklogd* to log Error 404 and Error 500 messages.

okd also plays a role as a control message router for the child services. In addition to listening for HTTP connections on port 80, *okd* also 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 4.4 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.

4.3 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.

For security, *oklogd* runs as an unprivileged user in its own *chroot* environment. Thus, a compromised *okd* or Web service cannot maliciously overwrite or truncate log files; it would only have the ability to fill them with "noise."

4.4 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*).

5 OKWS In Practice

Though its design is motivated by security goals, OKWS provides developers with a convenient and powerful toolkit. Our experience suggests that OKWS is suitable for building and maintaining large commercial systems.

5.1 Web Services

A Web developer creates a new Web service as follows:

1. Extends two OKWS generic classes: one that corresponds to a long-lived service, and one that corresponds to an individual HTTP request. 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.

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.

At present, OKWS requires Web developers to program in C++, using the same SFS event library that undergirds all OKWS helper processes and core libraries. 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 3 demonstrates these facilities.

5.2 Asynchronous Database Proxies

OKWS provides Web developers with a generic library for translating between asynchronous RPC and any given blocking client library, in a manner similar to Flash's helper processes [23], and "manual calling automatic" in [1]. OKWS users can thus simply implement *database proxies*: asynchronous RPC front-ends to standard databases, such as MySQL [21] or Berkeley DB [29]. Our libraries provide the illusion of a standard asynchronous RPC dispatch routine. Internally, these proxies are multi-threaded and can block; the library handles synchronization and scheduling.

Database proxies employ a small and static number of worker threads and do not expand their thread pool. The intent here is simply to overlap requests to the underlying data source so that it might overlap its disk accesses and

```
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 3: 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.

benefit from disk arm scheduling.

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, 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.

Finally, 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.

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 [24] and likewise transparently support both parsed and prepared SQL styles. Figure 4 shows a simple database proxy built with this library.

5.3 Real-World Experience

The author and two other programmers built a commercial Web site using the OKWS system in six months [22]. 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. Our current implementation uses 34 Web services and 12 database proxies.

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 iterative computations—like match computations—load instantaneously.

Our Web cluster currently 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 ModPerl/Apache servers [20] to accommodate less traffic. It ultimately was abandoned due to insufficient software tools and prohibitive hardware and hosting expenses [30].

5.4 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 [11]. The servers that host static content for OKWS, however, can enable HTTP keep-alive as usual.

We note that 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].

In our commercial deployment, we host a cluster of OKWS and database machines at a local colocation facility; we require hands-on hardware access and a network configured for our application. We serve static content from leased, dedicated servers at a remote facility where bandwidth is significantly cheaper.

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

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

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

Figure 4: 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.

6 Security Discussion

In this section we discuss OKWS's security benefits and shortcomings.

6.1 Security Benefits

(1) The Local Filesystem. An OKWS service has almost no access to the file system when execution reaches custom 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.

(2) Other Operating System 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.

(3) Database Access. As described, all database access in OKWS is achieved through RPC channels, using independent authentication mechanisms. As a result, an attacker who gains control of an OKWS web service can only interact with the database in a manner specified by the RPC protocol declaration; he does not have generic SQL client access. Note that 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 particular service is compromised, it can establish a new connection to a remote RPC database proxy; however, 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 2.2. We expect future versions of OKWS to enforce the same invariants at compile time.

(4) Process Isolation and Privilege Separation. 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 [26]. 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 functionality across multiple address spaces.

6.2 Security Shortcomings

The current implementation of OKWS supports only C++ for service development. OKWS programmers

```
<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 5: PHP version of the null service

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 more susceptible to buffer overruns and stack-smashing attacks. Future versions of OKWS might make these attacks less likely by supporting higher-level programming languages such as Python or Perl.

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 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.

7 Performance Evaluation

In designing OKWS we decided to limit its process pool to a small and fixed size. In our evaluation, we tested the hypothesis that this decision has a positive impact on performance, examining 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.

7.1 Testing Methodology

Performance testing on Web servers usually involves the SPECweb99 benchmark [31], but this benchmark is not well-suited for dynamic Web servers that disable KeepAlive 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 5). 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.

7.2 Experimental Setup

All Web servers tested use a large database table filled with sequential integer keys and their 20-byte SHA-1 hashes [12]. 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. Our database 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.

We implemented our test client using the OKWS libraries and the SFS toolkit. There was no resource strain on the client machines during our tests.

7.3 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 service as a variable number of processes, and collected throughput measurements (in requests per second) over the different configurations. The test client was configured to simulate either 500, 1,000 or 2,000 concurrent remote clients in the different runs of the experiment.

Figure 6 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 7. 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 have more outstanding requests to service before calling

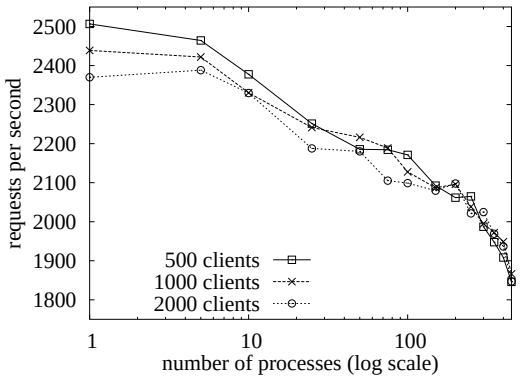


Figure 6: Throughputs achieved in the process pool test

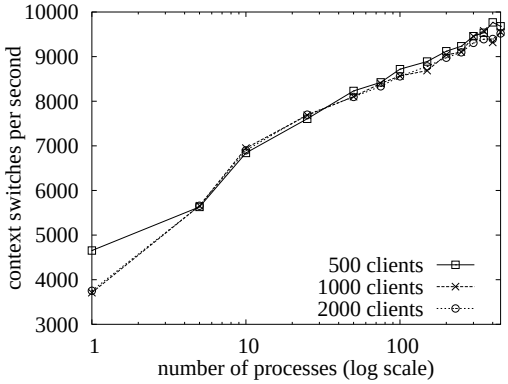


Figure 7: Context switching in the process pool test

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.

7.4 Web Server Comparison

The other Web servers mentioned in Section 3.3—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. Our 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. We ran OKWS in its standard configuration, with a one-to-one correspondence between processes and services.

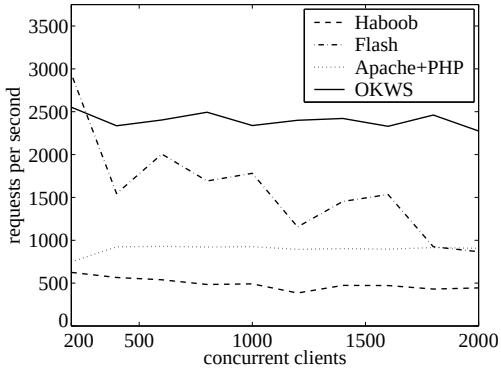


Figure 8: Throughputs for the single-service test

We enabled HTTP access logging on all systems with the exception of Haboob, which does not support it. All systems used persistent database connections.

7.4.1 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) and responsiveness (Figure 9) of the four systems as client concurrency increased. Figure 10 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 servicing a small number of concurrent clients, the Flash system outperforms the others; however, its per-

formance does not scale well. We attribute this degradation to 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, we noted a large number of running processes (on the order of 2000) and general resource starvation. 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 9 and the long tail in Figure 10.

In our configuration, 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.

7.4.2 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. We modified our null services 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 10 uniquely-named copies of the new null service, convincing the Web servers that they were serving 10 distinct services. Finally, our clients were modified to pause an average of 75 ms between establishing a connection and sending an HTTP request. We ran the experiment from 200 to 2000 simultaneous clients, and observed a graph similar in shape to Figure 8.

Achieved throughputs are shown in Table 2 and are compared to the 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

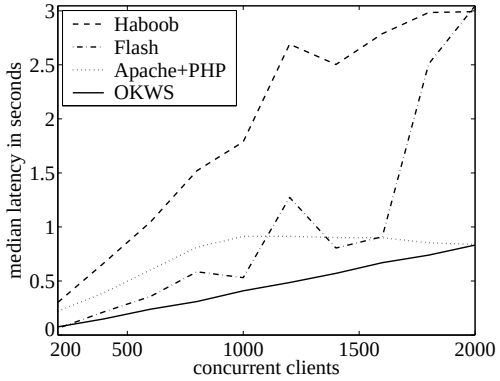


Figure 9: Median Latencies

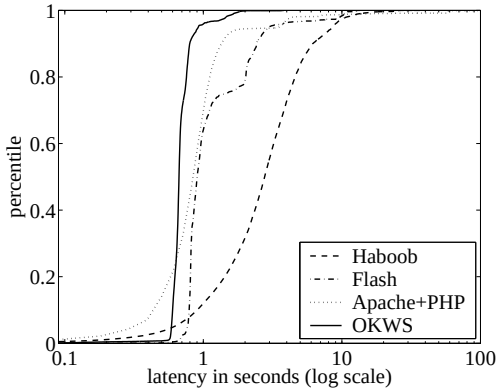


Figure 10: Client latencies for 1,600 concurrent clients

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 6 show this change has little impact on throughput. We can better explain OKWS's diminished performance by arguing that larger HTTP responses result in more data shuffling in user mode and more pressure on the networking stack in kernel mode. The same explanation applies for Apache, which experienced a similar performance degradation.

8 Related work

Apache's [3] 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.

	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 2: Average throughputs in connections per second

Highly-optimized event-based Web servers such as Flash [23] and Zeus [47] 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.

FastCGI [10] 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 [33] build on SEDA's infrastructure to create clusters of Web servers with the same appealing properties.

Other work has used the SFS toolkit to build static Web Servers and Web proxies [46]. 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 [34] 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 [25] scripting language investigated here, many other Web development environments are in widespread use. Zope [48], a Python-based platform, has gained popularity due to its modularity and support for remote collaboration. CSE [13] 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 [15] and IBM WebSphere [14]. 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 [16] 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 [45] can isolate Web services by running them on separate virtual machines.

9 Summary and Future Work

OKWS is a toolkit for serving dynamic Web content, and its architecture fits naturally into a compelling security model. The system's separation of processes provides reasonable assurances that vulnerabilities in one aspect of the system do not metastasize. The performance results we have seen are encouraging: OKWS derives significant speedups from a small and fixed process pool, lightweight synchronization mechanisms, and avoidance of unnecessary system calls. In the future, we plan to experiment with high-level language support and better resilience to DoS attacks. Independent of future improvements, OKWS is stable and practical, and we have used it to develop a popular commercial product.

Acknowledgments

I am indebted to David Mazières for his help throughout the project, and to my advisor Frans Kaashoek for help in preparing this paper. Michael Walfish significantly improved this paper's writing and presentation. My shepherd Eddie Kohler suggested many important improvements, Robert Morris and Russ Cox assisted in debugging, and the anonymous reviewers provided insightful comments. I thank the programmers, designers and others at OkCupid.com—Patrick Crosby, Jason Yung, Chris Coyne, Christian Rudder and Sam Yagan—for adopting and improving OKWS, and Jeremy Stribling and Sarah Friedberg for proofreading. This research was

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

Availability

OKWS is available under an open source license at `www.okws.org`.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. 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] Bugtraq ID 4606. SecurityFocus. `http://www.securityfocus.com/bid/4606/info/`.

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

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

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

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

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

[11] 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.

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

[13] T. Gchwind and B. 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.

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

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

[16] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. 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.

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

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

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

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

[21] MySQL. `http://www.mysql.com`.

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

[23] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX*, Monterey, CA, June 1999. USENIX.

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

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

[26] X. Qie, R. Pang, and L. 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.

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

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

[29] Sleepycat Software. `http://www.sleepycat.com`.

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

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

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

[33] J. R. van Berhen, E. A. Brewer, N. Borisova, M. C. an Matt Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A framework for network services. In *Proceedings of the 2002 USENIX*, Monterey, CA, June 2002. USENIX.

[34] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003. ACM.

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

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

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

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

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

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

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

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

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

[44] M. Welsh, D. Culler, and E. 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.

[45] A. Whitaker, M. Shaw, and S. 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.

[46] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX*, San Antonio, TX, June 2003. USENIX.

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

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

Notes

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