Radiatus: Strong User Isolation for Scalable Web Applications

Raymond Cheng†, Will Scott†, Paul Ellenbogen†, Jon Howell‡, Thomas Anderson†

†*University of Washington,* ‡*Microsoft Research*

Abstract

Web applications are a frequent target of successful attacks. The damage is amplified by the fact that application code is responsible for security enforcement in most web frameworks. In this paper we design and implement Radiatus, a web framework where all application-specific computation running on the server is executed within a sandbox with the privileges of the end-user. By strongly isolating users we protect user data and service availability from application vulnerabilities.

To make Radiatus practical on modern web applications, we introduce a distributed capabilities system to protect data at scale across the many distributed services that compose a modern web application. We show how this model protects applications from a large class of vulnerabilities, without compromising performance.

1 Introduction

Web sites are routinely broken into, resulting in frequent service disruptions and massive leakage of private information. In current web services, individual bugs often lead to wide-scale compromise because the server-side application logic is part of the trusted computing base (TCB). Existing web applications are structured as monolithic controllers which must interpret user permissions in order to dynamically assemble pages for a user. Because a single process mediates sharing between users, compromises allow attackers nearly unimpeded access to all of the information available to the service. Data compromises of this nature have remained the largest class of web application vulnerabilities for the full decade of OWASP (Open Web Application Security Project) vulnerability reports [11].

Because code is executed on behalf of the service, rather than as the user, remote code execution vulnerabilities are particularly devastating. For example, attackers in 2014 were able to write files and execute arbitrary code on Flickr servers by exploiting an injection vulnerability in a new photo books feature, allowing the attacker to manipulate or steal data from any user [58].

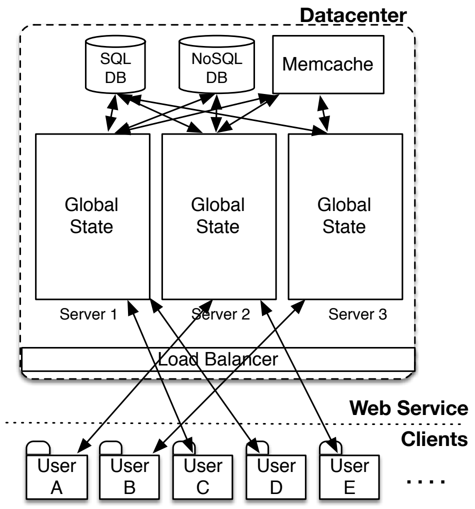
In this paper, we propose structuring web applications around *user containers*. A user's container is a strongly isolated sandbox that runs on behalf of a user. Barring compromise of the user authentication mechanism, intrusions should be contained only to the subset of site data already available to the malicious user. By executing web application code with reduced permissions, we can minimize the exploitable surface of a web service, even against unforeseen attacks.

Sandboxing users of a modern web application with reasonable performance is challenging. Generating a single page can span many layers of web servers, caches, storage systems, and coordinators, across multiple machines and data centers. A user container must strongly isolate users at every layer of the stack across the network, while supporting high levels of cross-user data sharing and application flexibility. Previous attempts to bring process isolation to server-side code have only isolated individual services (e.g. search, newsfeed, etc.) or database views [19, 39]. Other work has used encryption [49, 50] and data policies [20, 31, 47] to protect user data and guide data flows, but they typically provide no guarantees for the execution integrity of the service and limit the developer's flexibility to evolve the application. In industry, bug bounties, security audits, and monitoring can improve defenses, but these solutions become less tenable as applications grow in complexity [43].

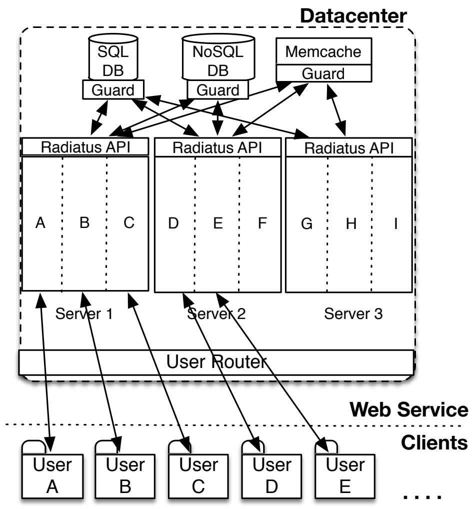
Radiatus is a web framework for organizing server-side application logic into user containers. The framework routes incoming HTTP requests via an authorization token to sandboxed processes that run code on behalf of individual users. Developers write their applications in terms of mutually distrusting users, who can only communicate through message passing protocols. Radiatus's distributed runtime uses a capability-based security system to protect access to private data, while being both storage space-efficient and horizontally scalable.

Our goal with Radiatus is to show that we can implement these changes in a way that is practical. The changes to the server are completely *transparent to the user*, who continue to access the site through an unmodified web browser. With our framework, developers can continue to use existing programming languages, distributed databases, distributed caches, content delivery networks, and infrastructure-as-a-service cloud providers. Even with this *developer flexibility*, we show that we can still effectively *protect user data from external attacks*. We also show that in spite of strongly isolating users, we can scale and achieve performance close to that of existing web frameworks. Finally, we show how enforcing data protection using capabilities fits naturally with the sharing access patterns in web applications and scales with distributed storage systems.

We have implemented the Radiatus system in 8764



(a) Layout of a traditional web service.



(b) Layout of a web service using Radiatus.

Figure 1: Current web applications provide little isolation within the context of the application runtime, leading to a large attack surface. Application logic across all machines are treated as part of the trusted computing base with access to global state. In Radiatus, applications are logically isolated into per-user containers, which run in isolated sandboxes with de-escalated privileges. Server-side code executing on behalf of a user is limited to the user's view of the database and user containers communicate through restricted message passing interfaces.

lines on top of the Node.js runtime. We have implemented three applications using Radiatus: an academic social network, a file sharing tool, and a messaging service. We have also ported Arc Forum[1] to run on our system.

While our framework can contain the damage caused by many external intrusions and exploits, we do not protect against insider threats with administrative access to site infrastructure. The framework also does not attempt to protect individual users from targeted attacks.

In the rest of the paper, we will elaborate on the following contributions:

- We introduce user containers for strongly isolating users in web applications and describe how existing web applications can be written in this model (Section 3).
- We have built the Radiatus platform with a concise API for implementing applications in the user container model and illustrate the API with three applications written in the framework (Section 4).
- We show that sandboxing users is in fact practical. Running the same application, a Radiatus web server has throughput within 98% of Apache/PHP and 63% of Node.js/Express. We also show our system can scale to a a 20-node server deployment on Amazon AWS. (Section 5).

[1]Arc Forum is the software that powers the popular Hacker News web site.

2 Background

While there are many ways that web applications can be constructed, this section attempts to characterize standard design patterns found across languages and frameworks. We then discuss the threat model for web services and why existing frameworks are susceptible to attack.

2.1 The Current Web Application Model

Figure 1a illustrates the architecture of a typical medium-sized web application. When users navigate to the site in their web browser, DNS routes the request to a nearby data center running the application. A load balancer then evenly distributes incoming requests across web servers running identical copies of the application logic. Because web servers are stateless, physical resources can be dynamically scaled up or down to meet the current user demand. Each web server interacts with a variety of relational databases, NoSQL databases, and caching services to authenticate the user, fetch data, and assemble the final page for the user.

Web applications are often written as scripts that are fired in response to incoming requests. Web frameworks provide a number of useful libraries for parsing HTTP request headers and returning a page populated with content. Similarly, storage, caching, and even user authentication are implemented as libraries invoked by applica-

CWE	Description	Percent
CWE-79	Cross-site Scripting	25.9%
CWE-89	SQL Injection	22.0%
CWE-264	Improper Access Controls	7.6%
CWE-119	Buffer Overflow	6.9%
CWE-94	Code Injection	6.8%
CWE-22	Path Traversal	6.8%
CWE-20	Improper Input Validation	6.7%
CWE-200	Information Exposure	3.9%
CWE-399	Resource Management Errors	3.5%
CWE-287	Improper Authentication	2.4%

Figure 2: Top 10 classes of vulnerabilities related to web technology as reported by the National Vulnerability Database [9]. We show the percentage of web-related vulnerabilities with each classification as labeled using the standard Common Weakness Enumeration (CWE).

tion logic.

The HTTP interface intermingles authentication, user actions, and content fetches, forcing the developer to properly handle requests, administer access control and prevent leakage of information. For example, in the case of a social network, one may store a list of users and their permissions in a relational database. When a user requests a feed of recent content, the web server assembles the page by querying the database for recent content and filters the content with access control policies in another table. The web server then populates a web page template with the retrieved content and return the page back to the user.

Large services may break functionality into multiple internal services in a service-oriented architecture (SOA). For example, a scalable search service may be written and maintained by a different product group from the shopping cart service. In this case, each individual service is typically written in the same model as above. A front end web service interacts with multiple internal services on behalf of a user request.

2.2 Threat Model

In this paper, we focus on preventing attacks aimed at compromising the execution integrity of a web server from an external vantage point. We assume a malicious user can craft arbitrary network packets and send arbitrary HTTP requests to the web interface. Thus, attacks include URL interpretation attacks, server-side includes, code injection attacks, SQL injection, malicious file executions, and buffer overflows. The Web Hacking Incident database [15] reports that most attacks of this nature have led to either information leakage, service disruption, defacement, malware distribution, or some combination, with the average cost of a data breach recovery in the U.S. of around $5.4 million [48].

This defined scope represents a large portion of vulnerabilities. In Figure 2, we catalog the 31,380 vulnerabilities in the National Vulnerability Database that are related to web technologies or the systems that power them, such as SQL databases. Each vulnerability is categorized by a Common Weakness Enumeration (CWE) [5] label. The methodology likely under-reports the frequency of server-side problems; the server-side code, for most web applications, is not public, limiting the ability for outside groups to diagnose precisely why compromises occur.

In this data set, 28.1% are client-side attacks that coerce a web browser client into performing unauthorized actions. An example is cross-site scripting. Our techniques do not address these vulnerabilities, but our implementation uses industry standard content security policies [4] and CSRF tokens [11] to mitigate such attacks.

Most vulnerabilities, 69.2% in this data set, involve flaws in server-side logic, such as code injection. Our goal is to address the broad sweep of server-side attacks against server application code, to allow application code to be developed quickly without worry that it might be introducing a subtle security vulnerability. We should note that prior work has shown progress at preventing specific types of server-side attacks, such as SQL injection.

We do not address vulnerabilities involving server misconfiguration, insider attacks, social engineering, and weak cryptographic primitives, such as the backdoor in the dual elliptic curve random number generator [23]. Each of these cases are more appropriately addressed by other, complementary techniques [22, 24, 26, 30, 31, 49].

3 Radiatus Design

While introducing per-user isolation seems like an intuitively simple idea, a number of challenges make it uniquely difficult in web applications. Previous work [19, 39, 40] had proposed the use of process isolation in a single web server, but none have explored the practical demands of per-user isolation in the context of a modern web service with horizontally scaling web servers and connecting to a variety of distributed storage systems, caches, and content distribution networks.

How do you support database security for each user? Different storage backends may support completely different user models, a problem sidestepped when application code is trusted. How do you manage memory consumption and storage costs? We can give each user their own cache and storage silo, but many objects in modern web applications are shared across users, sometimes across millions of users. How do you efficiently support one-to-many communication patterns? Copying data between containers may not be feasible, and certainly adds overhead. How do you perform distributed process management? User containers need to be placed to minimize communication cost and maximize load balancing.

In this section, we describe the user container model and the techniques in Radiatus that we use to make per-user isolation practical. We have three high-level goals for the system: First, the system should interoperate with existing cloud infrastructure, storage systems, and programming tools. Second, Radiatus should provide a general framework for isolating users, such that a single server-side application vulnerability, when exploited for a single user, does not lead to compromises in data integrity or service availability for all other users. Third, the performance and scale of applications written and deployed on Radiatus should be comparable to that of existing web frameworks.

3.1 Approach

Figure 1b shows the high-level model of a Radiatus application.

User Sandbox: In Radiatus, we spawn a sandboxed process, which we call a *user container*, for each active user. All code written by the developer runs inside this protection domain with the privileges of the particular user that the container is assigned. As such, the user container can only read and modify the data owned by the user.

User Routing: Because different user containers may exist on different web servers, we introduce a *user router*, which routes incoming connections to the proper user container.

Cross-Container Communication: We expose a thin message passing interface between user containers which allows them to communicate and pass data as necessary. The framework limits which which containers can communicate with each other, bootstrapped off of existing social networks. The developer writes strict interfaces, to which all incoming messages must conform. The developer can also specify a priority level, which determines whether the message wakes up the destination or queues until the next time the user logs in.

Storage Guard: We implement a *Storage Guard*, which controls access to user data including the database, caches, and the content delivery network (CDN). Logically, each user has a storage partition, but physically the underlying data is shared. The Storage Guard intermediates requests to any storage system, and implement a distributed capabilities system to manage shared data access between users. Instances of the Storage Guard operate with little to no coordination to prevent the framework from becoming a performance bottleneck.

Distributed Process Manager: A user container needs to be active in order to support an active user session or to process incoming messages from other users. The process manager uses server load, message priority, and communication patterns to optimally determine the placement of user containers, as well as to schedule when they are suspended and resumed.

The user container model improves web security by introducing a number of security properties unseen in most web frameworks. First, we move developer's code into a protection domain that runs on behalf of the user, instead of the service with full access to the web site. Following the principle of least privilege, attackers that exploit an application vulnerability are limited to the user containers they have credentials to access. We reduce the trusted computing base of the web server to the user router, login infrastructure, Storage Guard, and the sandboxing mechanism (e.g. OS processes/hypervisor). These components are written once and shared across all Radiatus web applications.

In practice, many types of multi-user web applications, such as online banking, office productivity, and online commerce, can be expressed in this model. Although Radiatus imposes constraints on the execution environment, it does not prohibit developers from building arbitrary applications. To the developer, Radiatus makes sharing and communication patterns explicit, placing logical boundaries between users.

3.2 Online Social Network Example

To illustrate the *user container model*, we have built a number of applications on top of Radiatus. Blizi (http://blizi.radiatus.io) is an academic social network that allows authors to post papers and solicit reviews from other users. The application also allows an author to privately share paper drafts and reviews with certain individuals. The intent is to allow limited dissemination without violating anonymous conference reviewing, as might occur when papers are posted to Facebook or the Web. We have started to organize one of our seminars around this tool.

Figure 3 shows the workflow of sharing a paper draft to a peer for review. A user container acts as the server-side agent for each user, in a shared-nothing architecture. The container manages the user's private data and capabilities to access data that has been shared with that user. When a user visits the site, the application code running in the container retrieves the data necessary to assemble the desired page.

Consider the scenario where Alice shares a paper with Bob. When Alice uploads the PDF to her user container, the application uses the storage interface to store the paper contents into a MongoDB database. It also caches

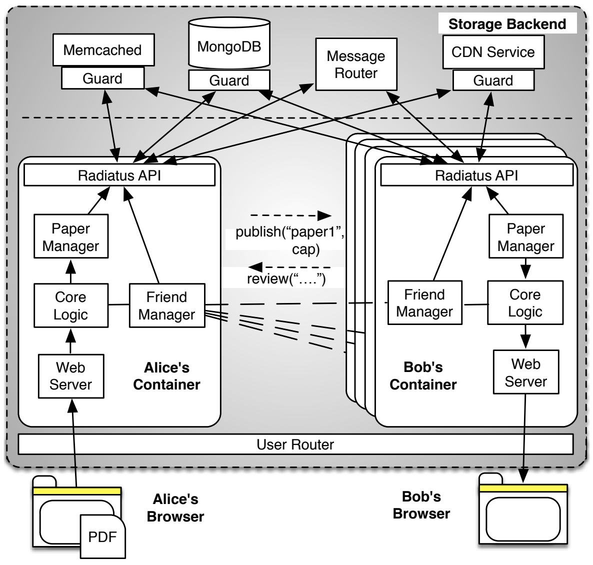


Figure 3: Workflow of uploading and sharing a paper on Blizi, an academic social network. Each user container acts in isolation and stores data in a private location. Alice and Bob communicate using a typed message passing interface, through which they share papers, reviews, and comments. User containers can privately share data, such as reviews, to select individuals.

metadata in a Memcached cluster.

This request returns a capability giving Alice (and only Alice) the ability to retrieve the paper. Using the cross-container communication interface, Alice can send the capability and an invitation to review the paper to Bob. This transferable capability gives Bob read-only access to that snapshot of the paper. If Alice makes changes to the paper, she would need to send another capability to Bob for him to see the revision. As we will see later, that capabilities refer to immutable data is important for system scalability. In a similar fashion, Bob can send reviews and comments back to Alice, or share them with other users on Alice's distribution list.

To support a newsfeed of public papers, we have a special *service user* to which other users can submit their public papers. The service user stores the newsfeed of recent papers and computes an index over all public papers. To search over the set of papers viewable by a particular user, Blizi code in the user's container computes a personal index over the papers to which it has access.

Using capabilities allows us to store each PDF only once in the entire web application. User containers are live when their respective user is online and when they must process a critical message, such as to resend a lost paper. Most messages, such as an invitation to review, are queued until the next time a user comes online. By isolating server-side application logic, an attack that compromises a user container would only affect that user's data and data to which she has been granted access.

3.3 Process Management

A process manager keeps track of the web server on which each user container is running. The process manager suspends any container when they become inactive. When a user container needs to be initiated for a particular user, the process manager load balances across servers with spare capacity. The process manager also attempts to co-locate user containers that frequently communicate with each other.

In the common case, user containers are sandboxed processes that share a kernel. For users with strong security preferences, we can reserve an entire virtual machine to their user container. Because each user container runs the same application code, they can be created in advance and dynamically assigned. Containers benefit from being forked with copy-on-write memory semantics. Because it takes 88.7ms to start a process in our implementation, the system also maintains a pool of instantiated but unconfigured processes. In order to reduce the latency of the first request by a new user, the process manager assigns new requests to one of these processes and spawns new processes as the pool is depleted.

3.4 User Routing

The user router is analogous to a load balancer for existing websites. Current load balancers proxy connections to servers with spare capacity, optionally also terminating the TLS connection. Likewise, the Radiatus user router looks for a session cookie in the HTTP headers of incoming requests. The cookie uniquely identifies the user and his/her user container. This form of authentication is common in nearly all web applications today. We use standard authentication techniques to verify users. Once identified, the user router will route requests to the proper user container.

3.5 Cross-Container Communications

Users can send messages to each other by addressing messages with their unique user IDs. Message routers forward these messages to the destination, where handlers defined by the developer specify how to interpret these messages. If a peer is not online, by default messages to that peer are put on a persistent queue. Radiatus also supports *wake-on message* policies for high priority messages (e.g. ones that impact user interfaces).

Radiatus maintains an access control list to limit which user containers can communicate with which other containers to prevent an attacker from crawling the site and to slow virus propagation. Communication is prohibited by default, but the implementation allows the developer to use existing social networks in place of the ACL; we currently support Google, Facebook, and XMPP buddy lists. The web application running in a container can also request additions (with mutual consent) or deletions from this list.

We introduced two optimizations to relieve stress on our message router. First, messages between two user containers on the same machine are directly routed to each other, bypassing the network. Second, we batch messages from (and to) different user containers on the same node to reduce overhead. With these optimizations, we were able to support our 20-node server deployment with a single message router. We plan to add distributed message queues, such as in Apache Kafka [2], as the need for further scaling arises.

Our message-passing system fits the growing use of event-driven programming for web development, similar to channels in Go [6], event emitters in Node.js [10], and Scala's actor model [1]. As with these systems, event-driven programming in Radiatus comes with a cost: added complexity in managing long chains of actions. We describe developer experiences more fully in Section 5.1.

There remains the risk of a developer introducing a vulnerability in a message handler, which may lead to wider compromises in the system. Radiatus provides a defense-in-depth strategy to mitigate this risk. We allow developers to specify security policies that define the required schema of messages, a rate limit, and priority level. Note that attacks must first compromise a user container to even have access to this interface, which will then be limited by the types and rate of messages it can send. Because Radiatus can monitor and block any messages between users, we can effectively quarantine any user by removing their communication ability when we detect misbehavior.

3.6 Storage Access

The Storage Guard layer provides access control to protect back-end storage systems. In our shared-nothing architecture, each user reads and writes into their own logical partition. Regardless of the number of users that persist the same content, or share the same content to their friends, the physical storage system de-duplicates content to store a single copy of each unique data value.

A Storage Guard instance is co-located with each database entry point, intercepts all requests, and tags each record with the owner. For example in our MongoDB deployment, an instance of the Storage Guard is run in front of each `mongos` query router. As the MongoDB cluster grows, there can be many Storage Guards and query routers independently coordinating distributed operations over the database, itself partitioned over many `mongod` database shards.

Key-Value Storage	
Name	Description
get(key)	Get a key
set(key, value)	Set a key/value
remove(key)	Remove a key
enumerate()	Return all keys
clear()	Clear partition

Cross-Container Communication	
Name	Description
send(userId, message)	Send a message
registerHandler(handler)	Handles incoming messages

Figure 4: Radiatus APIs for interacting with storage and other containers. The storage system exposes a logically isolated user partition.

3.6.1 Strawman Approach: Centralized ACLs

Database systems typically come with their own user management and access control mechanisms. One approach would be to implement a global monitor that coordinates between these disparate access control mechanisms, creating user IDs for each database. Such a monitor would translate Radiatus user storage requests into reads, writes and access control list modifications in the respective databases.

A practical challenge to this approach is that many of the NoSQL storage systems popular with web developers lack a consistent user model or only support access control on a coarser-grain than per user. If a particular website uses multiple databases, e.g., one specialized for photos, another for tweets, and a third for metadata, the implementation complexity of managing user permissions in Radiatus would be immense.

Further, such a system would be fundamentally unscalable. When Alice shares a photo with Bob, she would first contact the monitor and change the access permissions for that photo to include read permission for Bob. Because Bob can request the photo from any MongoDB query router, the new ACL must be persisted at all query routers to properly enforce this new access pattern. As the rate of sharing increases, this mechanism could quickly become a scalability bottleneck.

3.6.2 Distributed Capabilities

Instead, Radiatus uses distributed capabilities to encode access control in the existing communication patterns of the application. For example, when Alice notifies Bob about a new photo, Alice can pass the capability that gives Bob access to the photo. Any Storage Guard can directly verify this capability, allowing the database and application to scale independently.

We have implemented a Storage Guard for both MongoDB and Memcached, key-value systems where a document, record, or blob is the major unit that gets stored. In Figure 4, we show the interface that is exposed to the user container. In practice, the capability is a cryptographic hash of the content itself, which acts as a self-certifying name used for read-only access to a snapshot of the data [29]. This mechanism affords us a number of desirable properties. User containers are a shared nothing architecture, with automatic deduplication at the Storage Guard. Regardless of the number of users that persist the same content, or share the content with their friends, the database only needs to store one copy of every unique data value, deduplicated by content hash. Because the Storage Guards do not need to coordinate to operate, the security system does not affect the scalability of the underlying storage system.

Sharing an image with a friend is thus as simple as sending a small capability, requiring negligible amounts of communication or storage overhead as shown in Section 5. When a user stores a value, $set(k, v)$, the user container first computes the hash of the value. The container then sends a request to the Storage Guard to persist the ownership metadata, $(user, k) \rightarrow H(v)$, as well as the content, $H(v) \rightarrow v$. Containers can then pass the capability, $H(v)$, to other containers, who can retrieve it directly from the database. In Radiatus, we also use Memcached to cache metadata, such as which keys a user owns, to accelerate data fetches.

Because capabilities represent snapshots, rather than the continuing right to read updates, we expect revocation to be rare. To support revocation, we allow owners to delete data values from the blob store. For example if Alice uploads a photograph and shares it with her friends, she can delete the photo content; this invalidates any outstanding capabilities to the photo and prevents future retrieval. Of course, a corrupted friend's account could have already retrieved and leaked the photo to the tabloids; our aim is only to prevent an attacker from gaining access to everyone's data.

MongoDB and Memcached have been sufficient for the applications that we have built to date. Other NoSQL and key-value systems can be supported similarly. We next describe how capabilities would interact with other types of storage systems; these are not part of our current implementation.

Object-Relational Mapping (ORM) Object-relational mapping (ORM) [18] is a common programming model that allows developers to persist objects in relational databases. For example, it is natural to write an object-oriented program where an instance of an AddressBook class stores an array of Record instances. ORM libraries provide synchronization primitives to convert these objects into representations which are compatible with a relational database. ORM is the default programming model for many popular web frameworks including Django, Ruby on Rails, and PHP. In this case, the Radiatus Storage Guard functions identically as when in front of an SQL database, described next. Objects are serialized and hashed before persisted to the database.

Relational Databases We want to allow the user container access for any table (or object-relational) data for which the user is either the owner or holds the matching capability. To do this, we configure every table in the database with two extra columns to store the owner of the row and a hash of its contents (the capability). On an `INSERT` operation, the Storage Guard automatically populates the *owner* and *capability* columns. Subsequent requests to `UPDATE` a row are allowed if the user is the owner; this also modifies the hash value, ensuring that each capability is valid only for a particular data snapshot.

For queries, the user container sends the Storage Guard a list of its capabilities; these lists can be cached for efficiency. The results of simple `SELECT` queries can be post-processed to ensure only rows that the user has permission to access are returned, with the owner and hash value stripped off. More complex queries involving `JOIN` need to be prepended with a `SELECT` operation to check and strip off the owner/capability.

Content Distribution Networks (CDN) Many modern CDNs provide a programming interface for adding and removing content from the network. As such, we can create a Storage Guard that uses similar techniques. We treat the CDN as a blob store, which stores a single copy of every published piece of content. A NoSQL database is used to store user ownership metadata. Capabilities can then be embedded in a unique URL to be linked from HTML pages.

3.7 Analytics and Search

In order to support shared computation, we support the notion of a *service user*. A service user encompasses a unit of aggregate computation on behalf of the service. For example, the developer may want to collect aggregate statistics on page views or create a search index of public content. Service users are addressable like normal users, but their containers run code on behalf of the service.

We can use existing techniques to either limit leakage of information using differential privacy [40, 41] or distribute computation over private information across user containers where the data resides [33, 40]. These systems outline how to build privacy-preserving applications, such as personalized advertising, in a way that is compatible with Radiatus and limit how much information is centralized by the developer.

4 Implementation

4.1 Radiatus Framework

We have implemented the Radiatus web framework as a collection of various software components. A *container runtime* hosts a number of user containers on a server, each isolated in a sandboxed process. A *user router* routes incoming requests to the appropriate server and user container. A *storage guard* mediates calls to the storage systems by checking capabilities and subsequently translates the request to the database-specific interface. Lastly, *cross-container messaging* is supported by a message queuing system and a distributed process manager.

The source code for these projects can be found on GitHub (https://github.com/freedomjs). Radiatus extends on *freedom.js*, an existing peer-to-peer JavaScript framework. Both frameworks execute on the Node.js JavaScript runtime [10]. We leverage built-in Node.js support for process management and context isolation to construct sandboxed, unprivileged JavaScript processes. Each process runs its own instance of the V8 JavaScript engine, which is used as the user container to handle incoming requests for a single user. We inject stubs for each of the Radiatus APIs and block any other interfaces normally provided by Node.js.

4.2 Applications

In order to explore the expressiveness of our Radiatus framework, we used it to build a number of collaborative applications. These applications demonstrate the expressiveness of the framework API and show that user containers can serve as a form of modularity to organize functionality.

Academic Social Network: (http://blizi.radiatus.io) Blizi is an academic social network that serves as an example of how various social interactions work in Radiatus. The application allows a user to share a paper publicly or privately to specific individuals (Section 3.2).

File Sharing: (http://filedrop.radiatus.io) FileDrop allows a user to upload files to their user container. When a friend is granted access to the file, their user container can retrieve the file using the cross-container messaging system and then serve the file to the friend's browser.

Chat Messaging: (http://chat.radiatus.io) The chat application uses the cross-container messaging system to relay chat messages between people. In this particular example, we wrote a custom authentication manager that automatically assigns everyone a pseudonym and registers them on a global buddy list. When Alice sends a message to Bob, it is delivered from Alice's browser to Alice's user container, then to Bob's user container, and finally down to Bob's browser.

Radiatus fits well with the wide range of web applications that involve interacting users, including productivity software, games, social networking, e-commerce, and media. Because Radiatus is a server-side web framework, developers are unrestricted in how they design client-side user interfaces.

4.3 Porting Existing Applications

Not all applications can be easily ported to run Radiatus. We provide a simple build tool for compiling existing Node.js libraries to be used in Radiatus sandboxes. Some functionality may be broken in this process, such as in the case of filesystem access. While individual components of an existing Node.js web application can be ported using the same tool, any application logic that requires global access to state must be rewritten to exist within a restricted user container.

Arc Forum: Because the Radiatus process manager works with operating systems processes, we can port applications written in other languages, subject to the same limitations above. We ported the Arc Language Forum [32], the application behind the popular Hacker News web application, to the user container model. The forum is written in Arc, a dialect of the Lisp programming language that includes a built-in web server and libraries for generating HTML. The forum application provides a social news web application using these language primitives.

In Arc Forum, all data is persisted to disk using files storing Lisp lists. There are three subdirectories that store application state. *./profile/* stores a file for each user profile. *./story/* stores a file for each submitted story or comment. *./vote/* stores a file for each user's voting history.

For implementation simplicity, we choose not to completely re-architect Arc Forum, aiming instead for providing most of the benefits of Radiatus with minimal code changes. Instead of using an existing Storage Guard, we set up separate subdirectories with different permissions for each user container. We modified the user router to change the HTTP request routing behavior. Requests to render a page are directed to the proper user container as expected. When a user posts a new story or comment, it is replayed across all user containers. This configuration provides strong guarantees for isolating attacks against the code to render a page; this is especially important as any anonymous user can request to render a page. However, vulnerabilities in the handler for story posts can still affect a wide range of users. Because the application only allows authenticated users to post stories, we rely on manual detection to audit and disable misbehaving accounts.

5 Evaluation

Our evaluation of Radiatus asks the following questions to understand the security, performance, and ease of development. What effort is required to develop an application using this framework (Section 5.1)? How do user containers prevent existing classes of attacks (Section 5.2)? Does the Radiatus implementation provide acceptable performance with the added overhead of user containers (Section 5.3)?

5.1 Developer Experiences

Figure 5 shows the number of lines of code for each application. Server-side logic represents the portion of code run within a user container, which is coded against Radiatus APIs. Client-side user interfaces are written using standard HTML5 and are reusable with any web framework. In each of these cases, it took more effort to define the user interface than the server-side logic. For example in Blizi, our user interface included dynamic data-driven rendering. Blizi user containers simply persisted data uploaded from the user interface and sent messages notifying other appropriate users of changes. This complexity metric does not include the Radiatus runtime, including the login infrastructure, storage guard, and cross-container message router, which is common across applications.

We asked seven undergraduate students to build software on top of the framework, including Blizi, and describe their experiences and challenges. Most of the challenges they encountered were rooted in the complexity of event-driven programming and handling distributed failures. One example was that the action of sharing a paper with others involved uploading the paper to the user container, persisting it to disk, and sending a message to some set of peers. Failures anywhere on this chain of dependent events needs to be properly logged and handled. While many web frameworks are event-driven, Radiatus requires more messages to be sent compared to a traditional framework. We provide a number of tools, such as JavaScript promises, to structure some of this complexity.

Application	Blizi	FileDrop	Chat
Total LOC	2958	614	285
Server-side LOC	870	219	133
User Interface LOC	2088	395	152

Figure 5: Number of lines of code to implement each application. For these applications, the majority of code resided in user interfaces.

5.2 Security Analysis

To evaluate how Radiatus mitigates wide-scale exploitation of web vulnerabilities, we constructed four worst-case attack scenarios shown in Figure 6. For the attacks where we have access to the software, we download the vulnerable versions. For the others, we have to introduce our own vulnerability in Blizi to fit the description. Because many web applications are proprietary software running on managed infrastructure, the Common Weakness Enumeration [5] suspects that many vulnerabilities such as JavaScript *code injection* are heavily under-reported.

For the *database injection* attack, the attacker sends a malicious database command, which the web application relays to the database. While this attack commonly occurs in the wild as SQL injection, new NoSQL databases are equally vulnerable with their own query languages. In our evaluation, we use a vulnerable version of MongoDB and inject code through our web application.

For the *code injection* attack, we inject JavaScript that is evaluated by the web application itself. Various studies have demonstrated the feasibility of this attack [53], and in our evaluation, we introduce a vulnerability such that our web application improperly uses *eval(...)*.

For the *buffer overflow* attack, we link one of our applications to use a vulnerable LibYAML parser library. Because Node.js applications commonly links against native libraries to be used by JavaScript, these applications are still vulnerable to buffer overflow attacks.

For the *access control* attack, we send requests to access content that a user should not have access to. This vulnerability has been widely reported in the news (e.g. Snapchat in 2013 [13]), but researchers typically do not have access to the vulnerable source code. To facilitate this attack, we introduce a vulnerability to improperly check access control before sending content in our own application.

For each of these attacks, we run the application with 100 legitimate users, each reading, writing, and sharing content. We then introduce an attacker, who crafts ma-

Attack Type	Reference	Description	Affected Containers	
			No Sandboxes	Radiatus
Database Injection	CVE-2013-1892	MongoDB does not properly validate requests to the nativeHelper function in SpiderMonkey	All	None
Code Injection	CWE-95	Software incorrectly neutralizes code syntax before using the input in a dynamic evaluation call (e.g. "eval")	All	Attacker
Buffer Overflow	CVE-2013-6393	LibYAML executes arbitrary code via crafted tags in a YAML document, which triggers a heap-based buffer overflow	All	Attacker
Improper Access Control	CAPEC-58	Missing access control for certain HTTP commands, leading to RESTful privilege elevation	All	Attacker

Figure 6: Security vulnerabilities constructed in our evaluation setup. For further details on specific software versions affected, see the respective reference number. Without Radiatus sandboxing and database protections, exploiting any one vulnerability would lead to compromised service integrity for any user. Radiatus effectively sanitizes commands to the database and contains an attacker to their own user container.

licious requests to exploit each of the specified vulnerabilities from the network. For each vulnerability, we run the attack once with no sandboxing or database protection, and once in Radiatus. In Figure 6 we show the reach of each attack scenario.

For the database injection and access control attacks, an attacker could steal arbitrary data from the database. With Radiatus, the storage guard filters JavaScript from all commands going into the database and similarly filters all outgoing data for values not belonging to the authenticated user. For the code injection and buffer overflow attacks without sandboxing, an attacker has access to global state and could manipulate arbitrary user sessions. With Radiatus, our sandboxed user containers prevent the injected code from touching any state not belonging to the user.

Vulnerabilities can also arise on the cross-container messaging interface. In order to exploit these vulnerabilities, an attacker would first need to compromise an accessible user container from the network interface. The attack would then need to spread through the social graph of user containers in order to affect all users. As part of a defense-in-depth strategy, we can specify strict message formats, blacklisted regular expressions, rate limit, and alert systems for these messages. Compromised user containers can be reset without affecting other users. Combined with suspending misbehaving users, Radiatus represents a substantial improvement over existing web frameworks, where a single exploit can immediately affect all users. We also benefit from the fact that attacks along the cross-container messaging interface occur in a controlled data center environment. We leave to future work how information flow control, encryption, and stricter data policies can layer additional security guarantees on the system.

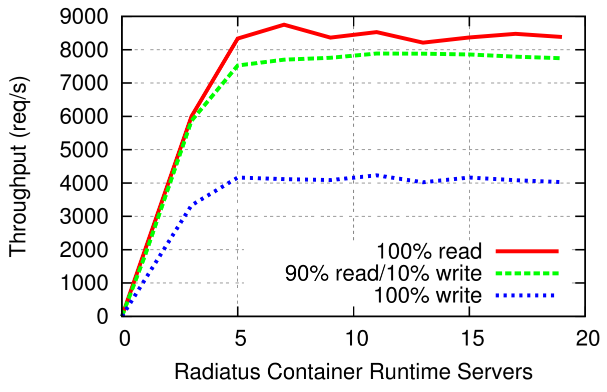


Figure 7: Aggregate throughput with different workloads across a 20-node cluster. We scale the number of servers running the container runtime with a single message router and storage guard. At around 5 runtime servers, the system is bottlenecked by the message router and storage guard.

5.3 Performance

We evaluated the performance of user containers on Amazon Web Services, with r3.large EC2 instances (2 CPU cores, 15GB memory, 32GB SSD, $0.175/hr in 2014). We stress test the performance of a single web server, as well as the overall throughput of a web service consisting of 20 servers.

Distributed Performance:

We set up a cluster of 20 virtual machines (VMs) on Amazon AWS to evaluate the performance of a coordinated deployment. Up to 19 VMs were used to host container runtimes and 1 VM was dedicated to running the message router, storage guard, and database. Figure 7 shows the aggregate throughput of the system with various simulated workloads. A read request consisted of a request to the storage guard and a response containing

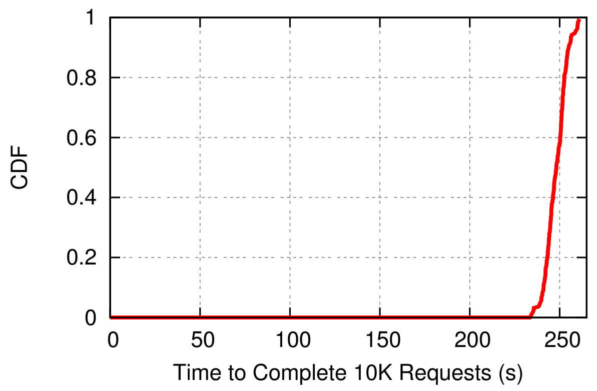


Figure 8: Cumulative distribution of completion times for 10,000 requests in each of 1900 user containers. The median completion time of 248 seconds represents 40.32 requests per second per container.

an item from the user's personal storage. A write request consisted of sending a message to a peer container through the message router, which then stores the value to their partition. In each of the data points, each server simulated 1000 user containers, each making back-to-back requests.

The graph grows linearly until around 5 container runtimes simulating 5000 active users. At this point, the message router and storage guard quickly become the bottleneck. This setup represented sufficient capacity for our medium-sized website deployments. As a point of comparison, Wikipedia in 2010 had 205 Apache web servers to support 414 million readers and 100 thousand active editors per month, as well as averaging 2000 HTTP requests per second [16]. We leave to future work the effort of scaling the message router and storage guard to support larger deployments.

Figure 8 shows the breakdown of completion time for user containers across 19 servers to finish 10,000 requests at a 9:1 read/write ratio. The graph shows minimal spread with a median completion time of 248 seconds or 40.32 requests per second per container. As most applications will likely require far fewer than 40 requests per second per user, we should be able to support more servers and active users as long as it fits the capacity of the message router and storage guard. Note that these numbers do not account for inactive users.

Single Machine Performance:

We profiled the performance of a single machine under load to determine how Radiatus restricts performance. In Radiatus, each user container is a sandboxed process running application logic with its own Node.js context. When scaling the number of active users on a single machine, memory quickly becomes a bottleneck in the context of a single server. We measured the average memory

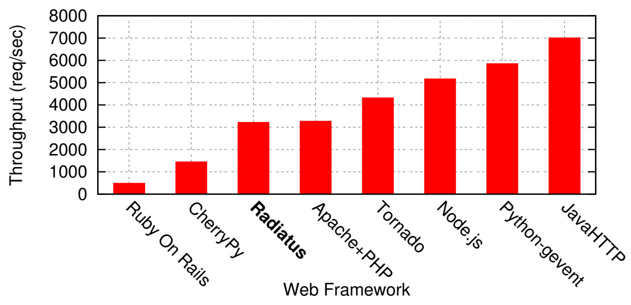


Figure 9: Comparison of single web server performance using the Siege Benchmark to make 1000 parallel connections at a time. Radiatus remains competitive with other frameworks despite handling requests in isolated user containers.

consumption of each process across 100 user containers to be 10.6MB. As such, each memory-optimized r3.large EC2 instance was able to support around 1400 processes before swapping.

Figure 9 shows the serving performance of a number of web frameworks for generating simple dynamic web pages. The serving performance data was collected using the Siege load testing tool, which simulates 100 users making HTTP requests in parallel. The page response in all cases was chosen to be a simple counter of how many requests had been made so far. This experiment prevents caching, while stress testing the HTTP request handler. Radiatus performs comparably to existing frameworks and better than some very popular frameworks, such as Ruby on Rails. We expect the practical overhead of Radiatus over Node.js to be much less when the response is backed by an actual web application.

6 Related Work

Radiatus is not the first attempt at strengthening security in web apps, and is inspired by the existing corpus of web security research. A number of prior techniques can be used in conjunction with Radiatus to further layer defense in depth.

Server-side Web Security: There have been a number of proposals to secure data integrity in the server. CryptDB [49], Mylar [50] and homomorphic encryption [30] have been proposed as ways to encrypt data and perform certain computations over encrypted data. Consequently even if a service gets compromised, users can rest assured that their data is safe.

A few projects have also explored variants of partitioning server-side application logic. BStore [21], Lockr [55] and RemoteStorage [12] provide mechanisms for application logic to be detached from storage, allowing storage primitives for a web application to be fulfilled by a third party.

Denali [59] and Xen [17] are paravirtualization techniques that can be used to isolate different web services on the same host with reasonable performance. Apache virtual host isolation [3] allows an Apache installation to host different web services.

OKWS [39] and Passe [19] introduce process isolation within an individual web application, providing protection boundaries between naturally isolated services (e.g. search), but stops short of per-user isolation. In fact, many web services now use a service-oriented architecture [14] for a variety of reasons beyond security. πBox [40] introduces a per-user sandbox that spans a mobile app and web server that interposes on all communication between users to enable fine-grained privacy management. Radiatus expands on these works to make per-user isolation practical at scale, supporting database security, data deduplication, 1-to-many communications patterns, and distributed process management.

Information flow control (IFC) can be used to limit the ability of a corrupted application to exfiltrate information. These techniques have also been applied to web frameworks to track malicious data flows. Hails [31] used IFC to track privacy violations when third-party applications run on data provided by a web service. PHP Aspis [47] used IFC to guard against injection attacks and DBTaint [25] introduced mechanisms to perform IFC across different applications. HiStar [61] and Flume [38] are operating systems that similarly use IFC to police data flows.

Other web frameworks, such as Excalibur [52] and GuardRails [20], have been introduced to attach fine-grained security policies on data. These frameworks help manage the complexity of managing access control, but under the exiting centralized model of web development. Logging techniques have also been applied to databases [22], allowing administrators to restore state to a point of known integrity after an intrusion.

Our focus in this paper is to create a sandboxing mechanism that scales well in modern web applications. These mechanisms are compatible with Radiatus and represent layers in a defense-in-depth strategy. We leave it to future work to explore how encryption, differential privacy, logging, IFC, and fine-grained data policies affect Radiatus's performance and security guarantees.

Client-side Browser Security: Weak isolation has long been recognized as an important security problem in web browsers. Improper isolation between different applications and principles can lead to data leaks, poor user experience, and unstable browser runtimes.

A variety of browsers [7, 8, 35, 36, 42, 57] and client-side JavaScript libraries [37, 54] have explored various isolation techniques for web applications and were influential in the Radiatus design. Because Radiatus is a server-side framework, it is complementary to the security provided by client-side isolation.

Distributed Operating Systems: While we address new challenges in providing security in web apps, we must recognize the decades of research in building distributed operating systems. Many prior efforts have recognized the need to isolate users and applications [27, 28, 34, 44–46, 51, 56, 60]

7 Conclusion

Radiatus provides an alternative model for web application design offering increased security over existing frameworks. In this paper, we have presented the user container abstraction as a lightweight mechanism to strongly isolate users within a web application. Testing our implementation of Radiatus, we find that it offers performance competitive with existing web frameworks, while adding an important layer of isolation between users. While the design of Radiatus applications is different from those in a monolithic controller, we present a set of APIs in our implementation which are expressive enough to support many of the classes of applications in use today. Finally, we present and evaluate our implementation of Radiatus and offer guidance on how processes can be leveraged for efficient isolated user containers.

The web platform already treats the browser as a per-user isolated container running potentially untrusted code. This abstraction has developed into a growing and powerful runtime for a growing diversity of applications. Leveraging these same design patterns on the server provides a structured approach to isolation, offering the same containment we would expect from our own machines, mobile applications, and multi-tenant data centers.

References

[1] Akka Actor Model. `http://akka.io/`

[2] Apache Kafka. `https://kafka.apache.org/`

[3] Apache Virtual Host. `https://httpd.apache.org/docs/2.2/vhosts/`

[4] Content Security Policy 1.0. `http://www.w3.org/TR/CSP/`

[5] CWE/SANS Top 25 Most Dangerous Software Errors. `http://cwe.mitre.org/top25/`

[6] Go Language Specification. `http://golang.org/ref/spec`

[7] Google Chrome Multi-process Architecture. `http://blog.chromium.org/2008/09/multi-process-architecture.html`

[8] IE8 and Loosely-Coupled IE (LCIE). http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx

[9] National Vulnerability Database. https://nvd.nist.gov/

[10] Node.js. http://nodejs.org/

[11] Open Web Application Security Project. https://www.owasp.org

[12] RemoteStorage. http://remotestorage.io/

[13] Snapchat Security Advisory. http://gibsonsec.org/snapchat/

[14] Understanding Service-Oriented Architecture. http://msdn.microsoft.com/en-us/library/aa480021.aspx

[15] Web hacking incident database. http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Database

[16] Wikimedia Foundation Annual Report. http://upload.wikimedia.org/wikipedia/commons/4/48/WMF_AR11_SHIP_spreads_15dec11_72dpi.pdf 2011.

[17] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.

[18] BARRY, D., AND STANIENDA, T. Solving the Java object storage problem. *Computer 31*, 11 (1998), 33–40.

[19] BLANKSTEIN, A., AND FREEDMAN, M. J. Automating isolation and least privilege in web services. In *IEEE Symposium on Security and Privacy (SP)* (2014).

[20] BURKET, J., MUTCHLER, P., WEAVER, M., ZAVERI, M., AND EVANS, D. GuardRails: A data-centric web application security framework. In *Proceedings of the 2nd USENIX Conference on Web Application Development* (2011).

[21] CHANDRA, R., GUPTA, P., AND ZELDOVICH, N. Separating web applications from user data storage with bstore.

[22] CHANDRA, R., KIM, T., SHAH, M., NARULA, N., AND ZELDOVICH, N. Intrusion recovery for database-backed web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 101–114.

[23] CHECKOWAY, S., FREDRIKSON, M., NIEDERHAGEN, R., GREEN, M., LANGE, T., RISTENPART, T., BERNSTEIN, D. J., MASKIEWICZ, J., AND SHACHAM, H. On the practical exploitability of dual ec drbg in tls implementations.

[24] CHEN, E. Y., BAU, J., REIS, C., BARTH, A., AND JACKSON, C. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), ACM, pp. 227–238.

[25] DAVIS, B., AND CHEN, H. Dbtaint: cross-application information flow tracking via databases. In *2010 USENIX Conference on Web Application Development* (2010).

[26] DIXON, C., ANDERSON, T. E., AND KRISHNAMURTHY, A. Phalanx: Withstanding multimillion-node botnets. In *NSDI* (2008), vol. 8, pp. 45–58.

[27] DOUGLIS, F., AND OUSTERHOUT, J. Transparent process migration: Design alternatives and the Sprite implementation. *Software: Practice and Experience 21*, 8 (1991), 757–785.

[28] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE JR., J. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (December 1995), pp. 251–266.

[29] FU, K., KAASHOEK, M. F., AND MAZIERES, D. Fast and secure distributed read-only file system. In *OSDI* (2000), USENIX Association, pp. 13–13.

[30] GENTRY, C. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[31] GIFFIN, D. B., LEVY, A., STEFAN, D., TEREI, D., MAZIÈRES, D., MITCHELL, J. C., AND RUSSO, A. Hails: protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), pp. 47–60.

[32] GRAHAM, P., AND MORRIS, R. Arc forum. http://arclanguage.org/forum 2008.

[33] GUHA, S., CHENG, B., AND FRANCIS, P. Privad: practical privacy in online advertising. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011), USENIX Association, pp. 13–13.

[34] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. MINIX 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev. 40*, 3 (July 2006), 80–89.

[35] HOWELL, J., JACKSON, C., WANG, H. J., AND FAN, X. MashupOS: Operating system abstractions for client mashups. In *HotOS* (2007), vol. 7, pp. 1–7.

[36] HOWELL, J., PARNO, B., AND DOUCEUR, J. Embassies: Radically refactoring the web. *NSDI* (2013).

[37] INGRAM, L., AND WALFISH, M. Tingram2012treehousoreehouse: Javascript sandboxes to help web developers help themselves. In *Proceedings of the USENIX Annual Technical Conference* (2012).

[38] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, pp. 321–334.

[39] KROHN, M. N. Building secure high-performance web services with OKWS. In *USENIX Annual Technical Conference, General Track* (2004), pp. 185–198.

[40] LEE, S., WONG, E. L., GOEL, D., DAHLIN, M., AND SHMATIKOV, V. πbox: A platform for privacy-preserving apps. In *NSDI* (2013), pp. 501–514.

[41] MCSHERRY, F. D. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2009), ACM, pp. 19–30.

[42] MICKENS, J., AND DHAWAN, M. Atlantis: robust, extensible execution environments for web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 217–231.

[43] MICROSOFT PATTERNS AND PRACTICES. Code Review. http://msdn.microsoft.com/en-us/library/ff648637.aspx.

[44] MULLENDER, S. J., VAN ROSSUM, G., TANANBAUM, A., VAN RENESSE, R., AND VAN STAVEREN, H. Amoeba: A distributed operating system for the 1990s. *Computer 23*, 5 (1990), 44–53.

[45] NIGHTINGALE, E. B., HODSON, O., MCILROY, R., HAWBLITZEL, C., AND HUNT, G. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 221–234.

[46] OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. The Sprite network operating system. *Computer 21*, 2 (1988), 23–36.

[47] PAPAGIANNIS, I., MIGLIAVACCA, M., AND PIETZUCH, P. PHP Aspis: using partial taint tracking to protect against injection attacks. In *2nd USENIX Conference on Web Application Development* (2011), p. 13.

[48] PONEMON INSTITUTE. 2013 Cost of a Data Breach Study: Global Analysis. `http://www.ponemon.org/`

[49] POPA, R. A., REDFIELD, C., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 85–100.

[50] POPA, R. A., STARK, E., HELFER, J., VALDEZ, S., ZELDOVICH, N., KAASHOEK, F., AND BALAKRISHNAN, H. Building web applications on top of encrypted data using Mylar. In *USENIX Symposium of Networked Systems Design and Implementation* (2014).

[51] RAMESH, K. Design and development of minix distributed operating system. In *Proceedings of the ACM Sixteenth Annual Conference on Computer Science* (1988), ACM, p. 685.

[52] SANTOS, N., RODRIGUES, R., GUMMADI, K. P., AND SAROIU, S. Policy-sealed data: A new abstraction for building trusted cloud services. In *Usenix Security* (2012).

[53] SULLIVAN, B. Server-side JavaScript injection. *Black Hat USA* (2011).

[54] TERRACE, J., BEARD, S., AND KATTA, N. P. K. JavaScript in JavaScript (js.js): Sandboxing third-party scripts.

[55] TOOTOONCHIAN, A., SAROIU, S., GANJALI, Y., AND WOLMAN, A. Lockr: better privacy for social networks. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies* (2009), ACM, pp. 169–180.

[56] VINTER, S. T., AND SCHANTZ, R. E. The cronus distributed operating system. In *Proceedings of the 2nd Workshop on Making Distributed Systems Work* (New York, NY, USA, 1986), EW 2, ACM, pp. 1–3.

[57] WANG, H. J., FAN, X., HOWELL, J., AND JACKSON, C. Protection and communication abstractions for web browsers in MashupOS. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 1–16.

[58] WEI, W. Flickr vulnerable to SQL Injection and Remote Code Execution Flaws. `http://thehackernews.com/2014/04/flickr-vulnerable-to-sql-injection-and.html?m=1`

[59] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the Denali isolation kernel. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 195–209.

[60] WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS) 12*, 1 (1994), 3–32.

[61] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 263–278.