Radiatus: a Shared-Nothing Server-Side Web Architecture

Raymond Cheng[†], William Scott[†], Paul Ellenbogen[*], Jon Howell[‡], Franziska Roesner[†], Arvind Krishnamurthy[†], Thomas Anderson[†]

†University of Washington, ∗Princeton, ‡Google

Abstract

Web applications are a frequent target of successful attacks. In most web frameworks, the damage is amplified by the fact that application code is responsible for security enforcement. In this paper, we design and evaluate Radiatus, a *shared-nothing* web framework where application-specific computation and storage on the server is contained within a sandbox with the privileges of the end-user. By strongly isolating users, user data and service availability can be protected from application vulnerabilities.

To make Radiatus practical at the scale of modern web applications, we introduce a distributed capabilities system to allow fine-grained secure resource sharing across the many distributed services that compose an application. We analyze the strengths and weaknesses of a shared-nothing web architecture, which protects applications from a large class of vulnerabilities, but adds an overhead of 60.7% per server and requires an additional 31MB of memory per active user. We demonstrate that the system can scale to 20K operations per second on a 500-node AWS cluster.

Categories and Subject Descriptors D.4.6 [*Security and Protection*]: Access Controls

Keywords web application, security, isolation

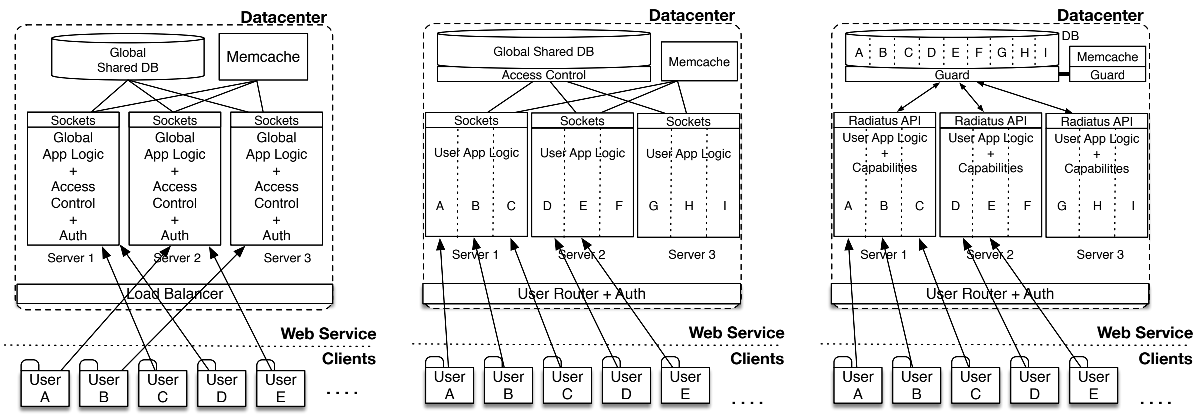
SoCC '16 October 05-07, 2016, Santa Clara, CA, USA.
© 2016 Copyright held by the owner/author(s). ISBN 978-1-4503-4525-5/16/10. . . .
DOI: http://dx.doi.org/10.1145/2987550.2987571

1. Introduction

Web sites are routinely broken into, resulting in frequent service disruptions and massive leakage of private information. With the current architecture of most web services, wide-scale compromise is all too easy because the server-side application logic is part of the trusted computing base (TCB). Existing web applications are structured as monolithic controllers with access to all user data, interpreting user permissions in order to dynamically assemble pages for a user. Thus, 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[17].

A natural approach to securing web applications is to deprivilege the code into sandboxed processes for individual services (e.g. search and newsfeed) [31, 54] or for individual users [55, 62]. Even if the code execution environment is isolated, previous attempts continue to assume a global data model to maximize compatibility with existing web frameworks. Applications are written assuming full access to a single shared database across all users, requiring that the developer iteratively restrict global data policies [32, 62, 69] to fit the application, potentially using information flow control [46, 61].

In this paper, we investigate an alternative data security model in an end-to-end *shared-nothing* web architecture. The web platform already treats the browser as a per-user isolated sandbox running untrusted code. We extend this into the server and database, where application code is run in a strongly isolated sandbox containing its own logical data partition with the privileges of the logged-in user. By default, no state is shared between users. Developers write their applications in terms of mutually distrusting users, who can only communicate through messages. For aggregate analytics across users, developers use standard differential privacy techniques [42, 57] to remove the need to store large amounts of raw data. Radiatus's distributed runtime uses a capability-based security system to protect access to pri-



(a) Layout of a traditional web service. (b) Layout of prior sandboxed services. (c) Layout of a Radiatus web service.

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. Previous attempts to sandbox services or user application logic still depend on specifying a correct global data security policy to protect a global database. In Radiatus, we extend isolation into the storage layer in an end-to-end shared-nothing architecture. Both application logic and storage for each user runs in sandboxes with de-escalated privileges, which communicate through a restricted message passing interface.

vate data, while being both storage space-efficient and horizontally scalable. Capability-based security can be a more tractable approach for the data sharing patterns of web applications across internal services compared to group-based data policies. Barring compromise of the user authentication mechanism, intrusions are contained to the subset of data already available to the malicious user.

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 isolate users at every layer of the stack, while supporting cross-user data sharing and application flexibility. Our goal with Radiatus is to show that we can implement a shared-nothing web architecture in a way that is practical today with scale, cost, and performance within a factor of existing shared-everything web frameworks. The changes to the server are completely *transparent to the user*, who continues to access the site through an unmodified web browser. Similarly, developer should be able to continue using existing programming languages, distributed databases, distributed caches, content delivery networks, and infrastructure-as-a-service cloud providers.

In order to evaluate the strengths, weaknesses, and performance implications of our architecture, we have implemented a Node.js-based web framework in 8764 lines of code, called Radiatus, and three applications: an academic social network, a file sharing tool, and a messaging service. The difficulty of porting applications can vary depending on the application workload. For example, we found it much easier to port applications that were written to be self-deployed or federated. We describe our experiences porting Arc Forum, the engine behind Hacker News, 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. Radiatus is complementary to other web security-related work, including encryption [45, 65, 66], language-based security [36, 38, 56], and bug-finding [30, 43, 68, 71, 78].

The rest of the paper elaborates on our contributions:

- We describe the Radiatus shared-nothing architecture for strongly isolating users in web applications and describe how existing web applications can be written in this model (§ 3).
- We have built the Radiatus platform and illustrate the Radiatus API with three applications written in the framework (§ 4).
- We show that sandboxing users prevents large-scale exploitation of the most severe web-related vulnerabilities of 2014. We quantify the performance impact of Radiatus, including using it on a 500-node deployment on Amazon AWS (§ 5).

2. Background

There are many ways that web applications can be constructed. This section attempts to characterize standard design patterns found across languages and frameworks, and then discuss why they 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, DNS

```
if (isset($_COOKIE['ari_auth'])) {
  $buf = unserialize(stripslashes($_COOKIE['ari_auth']));
  list($data,$chksum) = $buf;
}
```

Figure 2: FreePBX (v≤2.9.0.9), a VoIP server, improperly sanitizes the `ari_auth` cookie before calling `unserialize` in htdocs_ari/includes/login.php. Because `unserialize` can import arbitrary PHP objects, this vulnerability can be exploited to execute arbitrary code (Sept. 2014).

routes the request to a nearby data center. A load balancer then distributes incoming requests across web servers running identical copies of the application. Because servers are stateless, physical resources can be dynamically scaled up or down to meet demand.

The HTTP interface intermingles authentication, user actions, and content fetches. The developer must properly handle requests, administer access control and prevent leakage of information. Developer speed is a critical issue, but expediency comes at the cost of an increased number and severity of bugs. 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, filtering the content with access control policies in another table, and populating a web page template.

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

Because code is executed on behalf of the service, rather than as the user, remote code execution vulnerabilities are particularly devastating. For example, Figure 2 shows a subtle code injection vulnerability in FreePBX, an open source VoIP server, as exploited on Sept. 2014 [60]. An improperly sanitized HTTP cookie, `ari_auth`, passed into `unserialize()` allows an attacker to import arbitrary PHP objects into the context. Code injection vulnerabilities have led to countless data leaks and service disruptions in web applications [15, 22, 25, 39, 76]. 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 [76].

2.2 Global Data Security Policies

Existing databases allow developers to specify access control policies, typically at the granularity of a database table or collection. Prior work expands on this primitive to provide the ability to write fine-grained access predicates [8, 62] and information flow control policies [46, 61]. Other frameworks bind policies to the data [32, 69] which follows the data as it propagates through the system. Data policies can be explicitly defined by the developer, or inferred through monitoring real access patterns at runtime [31].

CWE	Description	Percent
CWE-20	Improper Input Validation	6.7%
CWE-22	Path Traversal	6.8%
CWE-79	Cross-site Scripting*	25.9%
CWE-89	SQL Injection	22.0%
CWE-94	Code Injection	6.8%
CWE-119	Buffer Overflow	6.9%
CWE-189	Numeric Errors	1.7%
CWE-200	Information Exposure	3.9%
CWE-264	Improper Access Controls	7.6%
CWE-287	Improper Authentication	2.4%
CWE-352	Cross-Site Request Forgery*	2.2%
CWE-399	Resource Management Errors	3.5%
	Other (Server-Side)	3.6%

Figure 3: Most common vulnerabilities related to web technology as reported by the National Vulnerability Database [15]. Each is labeled using the standard Common Weakness Enumeration (CWE). *Client-side attacks that coerce browsers into performing unauthorized actions; the others involve server compromise.

The development pattern for these systems typically start with global access to the database, as the developer iteratively restricts access to define a proper security policy. Previous attempts at per-user sandboxes for code execution also use this data model to optimize for compatibility with existing applications [62]. In this paper, we investigate the practicality of a shared-nothing architecture, where user partitions are empty to start and require message passing to share data between users. Our goal is not to advocate for a shared-nothing architecture, but to analyze the cost and experience, showing that a system based on message passing can be practical for certain applications that need the potential security benefits.

2.3 Threat Model

We focus on preventing attacks aimed at compromising a web server from an external vantage point. We assume a malicious user can craft arbitrary network packets and send arbitrary requests to the server. This includes URL interpretation attacks, server-side includes, code injection attacks, SQL injection, malicious file executions, and buffer overflows. The Web Hacking Incident database [22] reports that attacks of this nature have led to information leakage, service disruption, defacement, and malware distribution.

We catalog the 31,380 vulnerabilities in the National Vulnerability Database [15] that are related to web technologies or the systems that power them, such as SQL databases. Each vulnerability comes categorized with a Common Weakness Enumeration (CWE) [4] label. The methodology likely under-reports the frequency of server-side problems; for most web applications, the server-side code 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, such as cross-site scripting and cross-site request forgery. Radiatus is not aimed at these vulnerabilities, but our implementation uses industry standard content security policies [3] and CSRF tokens [17] to mitigate such attacks.

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

We do not address server misconfiguration, insider attacks, social engineering, or weak cryptographic primitives. Each of these are better addressed by other, complementary techniques [34, 37, 41, 45, 46, 65].

3. Radiatus Design

While introducing per-user isolation seems like an intuitively simple idea, a number of challenges make it uniquely difficult for web applications. Previous work [31, 54, 55] has proposed process isolation in a single web server, but none have explored the practical demands of per-user isolation in the context of a modern scalable web service using a variety of distributed storage systems, caches, and content distribution networks.

For example, how do you support per-user database security? Different storage backends have 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 users may not be feasible, and certainly adds overhead. How do you perform distributed container management? User containers need to be placed to minimize communication cost and maximize load balancing.

3.1 Goals

In this section, we describe the user container model and the techniques that we use to make per-user isolation practical.

- **Strong Isolation:** Radiatus should provide a general framework for isolating users, such that server-side application vulnerabilities do not compromise data integrity or service availability for other users.
- **Minimal Overhead:** We should support each additional user with minimal overhead in performance and cost compared to existing web frameworks.
- **Scalable:** The scale and performance of applications written and deployed on Radiatus should be comparable to that of existing web frameworks.
- **Interoperability:** The system should interoperate with existing cloud infrastructure, storage systems, and tools.

3.2 Approach

Figure 1c shows the high-level model of a Radiatus application. We move developer's code into a protection domain that runs on behalf of the user. Attackers that exploit a vulnerability in that code are limited to the containers they have credentials to access.

Sandbox Users: 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 given user. As such, the user container can only read and modify data that is owned by the user. In practice, we use Linux containers, which provides memory, filesystem, and fault isolation between containers. We leverage existing techniques to apply resource limits to user processes.

Limited Interfaces: Existing web services expose a single large HTTP interface for authentication, user actions, and content fetches. Radiatus splits this interface into three with access restricted by least privilege. Any user can authenticate with the *user authenticator*. Once equipped with an authorization token, the *user router* forwards a user's request to their own de-privileged containers; requests are never directly processed by privileged code. We expose a *cross-container message* interface between user containers to facilitate data sharing. User containers can only communicate with mutual consent and access to this interface is blocked by default.

Passive Containers: To minimize memory overhead, containers are offline by default. Applications are written using an event-based programming model. A distributed container manager determines placement, suspends, and resumes user containers as necessary to process incoming requests.

Distributed Capabilities: Logically, each user has a storage partition, but physically the underlying data is shared and stored on commodity databases. A *storage guard*, intermediates access to the database and enforces access control for user data. We provide capabilities for scalable fine-grained access control across disparate database systems, while deduplicating common data between users.

Minimize trusted computing base: When processing requests from the browser or between containers, we expect developers to employ *defensive programming*, treating communicating parties as untrusted entities. In addition to these message checks, our trusted computing base consists of a user authenticator, user router, storage guard, Radiatus runtime, and container mechanism (e.g. OS processes/hypervisor). These components are written once and shared across all Radiatus web applications.

3.3 Example Application

In this section, we walk through the typical lifecycle of Radiatus applications. A user container acts as the server-side agent for each user, in a shared-nothing architecture. The

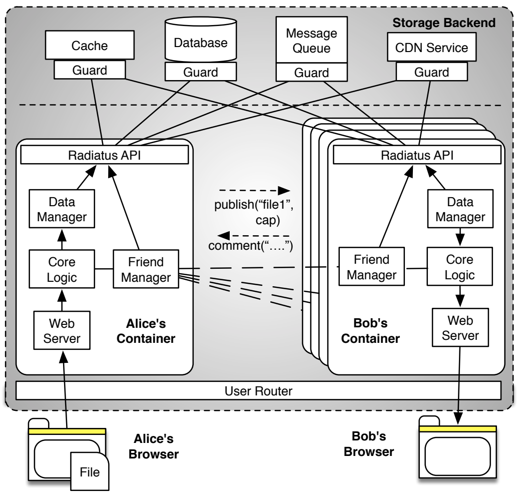


Figure 4: Workflow of uploading and sharing files in the user container model. Each user container acts in isolation and stores data in a private location. Alice and Bob communicate using a typed message passing interface, through which they share files and messages.

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. Because user containers run identical application logic, our system maintains a pool of instantiated but unconfigured containers, which are lazily bound when users log in and subsequently destroyed when they log out. This technique reduces the latency of the first request by a new user.

Figure 4 shows the workflow of sharing a file in a Radiatus application. Consider the scenario where Alice shares a file with Bob. When Alice logs in, a user container is assigned to her. Alice uploads the file to her user container and the application uses the storage interface to store the file and metadata. The storage request returns a capability giving Alice (and only Alice) the ability to retrieve the paper. Using the cross-container communication interface, Alice's container can send the capability in a message to Bob's container. If Bob is currently offline, this message is stored in a persistent queue until Bob logs in.

Capabilities are transferable and provide read-only access to an immutable snapshot of data. If Alice makes changes to the file, she would need to send another capability to Bob for him to see the revision. As we will see later, the fact that capabilities refer to immutable data is important for system scalability and capability revocation.

3.4 Container Management and User Routing

A container manager keeps track of the web server on which each user container is running. The container manager suspends containers when they become inactive. When a user container needs to be initiated, the manager chooses an appropriate server. The container manager also attempts to co-locate user containers that frequently communicate with each other.

```
type: {
  title: 'string',
  author: 'string',
  timestamp: 'number',
  pdf: 'buffer'
},
rate: 100,
priority: 'wake'
```

Figure 5: Example of a declared message type. Developers must specify the type, rate limit, and priority of all messages across the cross-container message interface.

Like a traditional load balancer, the Radiatus user router proxies connections to servers, optionally terminating the TLS connection. The user router looks for a session cookie in the HTTP request, uniquely identifying the user. If the router instance has never seen the user before, it will query the container manager for the host server of the user container and caches this information. Subsequent requests from this user are then forwarded to the proper user container. Even if an application contains an exploitable vulnerability, network requests from the attacker will only be routed to the attacker's container. User routers are horizontally scaled as necessary to meet traffic demands.

3.5 Cross-Container Communications

In order to support offline users, messages between user containers are persisted in a distributed message queue, like Apache Kafka [1] and Amazon SQS [2], until the recipient's user container wakes up. Developers can also specify *wake-on message* policies for high priority messages (e.g. ones that impact the user interface).

By default, containers are disconnected. Developers use `addPeer/removePeer` calls in the API to specify permitted communication channels. Depending on the application, developers may require bilateral consent before accepting messages (e.g. friend requests), or they may allow one-way consent (e.g. chat messages). Limiting the connectedness of the container graph makes it more difficult for the attacker to crawl the site by slowing virus propagation.

We introduce two optimizations to relieve stress on our message queue. First, messages between two user containers on the same machine are directly routed to each other, bypassing the network. Second, we batch messages to different user containers on the same node to reduce overhead.

As with incoming handling web requests, developers are expected to employ defensive programming when writing message interfaces, treating communicating parties as untrusted entities. While it is possible for vulnerabilities to exist in this code (cf. CWE-20), Radiatus employs a defense-in-depth strategy to mitigate the risk of widespread exploitation. Attackers must first compromise a user container to even have access to the cross-container interface. They must

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, msg, msgType)	Send a message
registerHandler(handler)	Handles incoming messages
addPeer(userId)	Add a peer
removePeer(userId)	Remove a peer

Figure 6: Radiatus APIs for interacting with storage and other containers. The storage system exposes a logically isolated user partition.

then find an exploit that provides control over the neighbor container in a way that can be propagated.

Because containers run on server hardware, we can use three additional techniques to limit attacks:

Typed interfaces: Developers must declare the messages and protocol for cross-container communication [50]. The system checks all messages at runtime, denying non-conforming messages. Figure 5 shows an example of a declared message type.

Resource limits: Each container is subject to strict limits on resources (e.g. CPU, memory, network), to prevent attacks from launching denial-of-service attacks.

Anomaly detection and eviction: While we have not yet implemented this feature, we could use machine learning to build a steady state model of expected container behavior, because application communication patterns are hard-coded into the application logic. Anomalous resource usage or communication patterns can trigger an operator alert for manual review. A dashboard gives the operator controls to pause containers, partition the network and evict users as necessary to preserve the health of the system. Many of the techniques, such as real-time notifications, high-speed event monitoring, and policy scripts, are borrowed from decades of research in intrusion detection systems [63].

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. A storage guard instance is co-located with each database entry point, intercepts all requests, and tags each record with the owner. Storage guard implementations must be adapted to communicate with each type of database (e.g. SQL, NoSQL). 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. We assume the developer will use a heterogeneous set of diverse databases. While we could have used the built-in access control provided by the database, synchronizing fine-grained access control policies across databases could easily become a bottleneck.

3.6.1 Distributed Capabilities

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 directly pass the capability that gives Bob access to the photo. The capability is a cryptographic hash of the content plus a random nonce $H(data, nonce)$, which acts as a self-certifying name proving read-only access to immutable data [44].

When a user stores a value, $set(k, v)$, the user container adds a random nonce, n to the value and computes the hash, $H(v,n)$. The container then sends a request to the storage guard to persist the ownership metadata, $(user, k) \rightarrow H(v,n)$, the capability, as well as the content, $H(v,n) \rightarrow v$.

Capabilities are then self-certifying and transferable. Containers can send the capability, $H(v,n)$, to other containers, which can use it to retrieve the data from the database. The storage guard will only return content if the capability has been registered by the owner. Radiatus also uses Memcached to cache metadata, such as which keys a user owns, to accelerate data fetches.

This mechanism helps satisfy our original goals of scalable isolation with minimal overhead, with these properties:

- Because the capability provides proof of access, any storage guard can independently verify a capability, allowing the database and application to scale independently.
- 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.
- Transferring capabilities is cheap, regardless of the content size or number of users with access.
- If a capability is granted to a malicious user, they cannot destroy the owner's data.

To support revocation, e.g., to remove an ill-advised tweet, the owner's container can delete both the capability and the data value from the store. This invalidates any outstanding capabilities to the data and prevents future retrieval. To revoke a capability from a specific user, e.g., on a change to a friend list, the owner's container picks a new nonce m, computes the new capability $H(v,m)$, installs it, distributes the capability to the new set of friends, and then deletes the old capability mapping. Of course, a corrupted friend's account could have already retrieved and stored a copy of the data or leaked it to the tabloids. Revocation only prevents later access.

We have implemented storage guards for MongoDB and Memcached, which 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) [27] 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 give a user container access for any table (or object-relational) data for which the user 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 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 Internal Services

To simplify development and improve modularity, a number of web applications are designed with internal structure: a stateless frontend web server that coordinates calls to a mixture of backend application, caching, and storage services. For example, an e-commerce site may operate different internal web services for their shopping cart, user recommendations, personalized search, notifications, and customer support. As with frontend servers, these internal services often mix application logic with access control and privacy enforcement, raising the vulnerability of user data to compromise.

With Radiatus, internal services can exist within containers, with user credentials transparently propagating from the frontend to the backend services. User containers are not always appropriate or necessary. Some backend services are general purpose, such as a distributed configuration or lock manager. Others require access to site-wide data, such as computing trending topics in Twitter. By sandboxing frontend code, Radiatus provides defense in depth protection for these internal services. More fundamentally, these services need to be treated as part of the trusted computing base: carefully designed with a narrow interface that is hardened against attack.

In a few cases, internal services can be treated by the rest of the system as an untrusted user. An example is a service to aggregate content for a public newsfeed. For this, Radiatus supports the notion of a *service user* for shared computation. A service user encompasses a unit of aggregate computation on behalf of the service. Service users are addressable like normal users, but their containers run code on behalf of the service. Since service users communicate with normal user containers, the service developer must apply defensive programming with the assumption that user containers can be compromised, and vice versa. For example, one could apply differential privacy libraries for privacy-preserving data collection.

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 unique sandbox. A *user router* routes incoming requests to user containers. A *storage guard* mediates calls to the storage systems by checking capabilities and translates the request to the database-specific interface. Lastly, *cross-container messaging* is supported by a message queuing system and a distributed container manager.

We have implemented Radiatus as a web framework in 8764 lines of code on the Node.js runtime [14], where each user is allocated a Docker container [5] running a separate Node.js process. We inject stubs for each of the Radiatus APIs and block any other interfaces normally provided by Node.js. The Storage Guard interposes on user storage requests to expose a partitioned NoSQL database, but internally uses MongoDB and memcached.

While our implementation uses Docker, the Radiatus design is compatible with other virtualization technologies. Depending on the operating environment, performance, and security requirements, developers can choose a virtualization technique that works for them, from OS-level virtualization [5, 13, 16, 21] to full virtual machines [12, 26]

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

Figure 7: Lines of code to implement each application.

Our message-passing system fits the growing use of event-driven programming for web development, similar to channels in Go [9], event emitters in Node.js [14], and Scala's actor model [18]. 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 next.

4.2 Applications

In order to explore the expressiveness of our Radiatus framework, we used it to build a number of collaborative applications. 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. Figure 7 shows the number of lines of code for each application we developed using Radiatus.

Academic Social Network: Blizi 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.

File Sharing: FileDrop allows a user to upload files to their user container. When a friend is granted access to a file, the friend's container can retrieve it from the storage service using the file's capability. The application can then serve the file to the friend's browser.

Chat Messaging: 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.

4.3 Porting Existing Applications

We provide a simple tool for bundling existing Node.js libraries in Radiatus user containers. However, not all applications can be easily ported, such as those that use direct 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 container 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 [47], 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. Because data is persisted to files, rather than a database, the port required no changes to Arc Forum, and 188 lines of changes to our user router and container runtime.

5. Evaluation

Our evaluation asks the following questions to understand the security and performance of Radiatus. How do user containers prevent existing classes of attacks (§ 5.1)? Does the Radiatus implementation provide acceptable performance given the added overhead of user containers (§ 5.2)? What is the incremental cost per user (§ 5.3)?

5.1 Security Analysis

Radiatus is designed to reduce vulnerability of user data to exploits that take advantage of bugs in web server application code. To evaluate this, we analyzed all of the vulnerabilities in the National Vulnerability Database (NVD) with a maximum severity score of 10.0 from 2014. The Common Vulnerability Scoring System (CVSS) is an open industry standard, which reserves the 10.0 score for the most severe vulnerabilities that fit 6 criteria: (1) remotely exploitable, (2) low barrier to access, (3) requires no authentication, (4) total information disclosure, (5) complete loss of system integrity, and (6) leads to total loss of availability of the attacked resource. Out of all 7316 software vulnerabilities reported in 2014, only 233 received this score. Of these 233, we analyzed the 40 that involved server-side web software. Many web applications are proprietary software running on managed infrastructure, and bugs in that software are likely to be under-reported. As a consequence, we expect Radiatus to help more cases than those reported in this section.

For each reported bug, we attempted to understand how the vulnerability affects the web application if the software with the vulnerability was translated into the Radiatus model. Figure 8 lists each vulnerability, its original impact, and its impact in Radiatus.

- Arbitrary: exploitation leads to arbitrary code execution, service disruption, and information disclosure;
- Data leak: bug can be leveraged to leak arbitrary information, but not run code;
- Prevented: bug cannot manifest in the target system;
- Sandbox: exploitation is limited to a single user's sandboxed container;
- Auth: app developers delegate the responsibility of implementing authentication to Radiatus;

CVE ID	Short Description	Original Impact	Impact in Radiatus
	Code Injection and Buffer Overflow		
CVE-2014-0294$^{\spadesuit}$	MS Forefront 2010 improper email parsing	Arbitrary	Sandbox
CVE-2014-0474$^{\spadesuit}$	Django improper type conversion	Data leak	Sandbox
CVE-2014-0650$^{\spadesuit}$	Cisco Secure ACS allows arbitrary shell commands	Arbitrary	Sandbox
CVE-2014-0787*	WellinTech KingSCADA buffer overflow	Arbitrary	Sandbox
CVE-2014-2866$^{\clubsuit}$	PaperThin CommonSpot uses client-side JavaScript for access restrictions	Arbitrary	Sandbox
CVE-2014-3496$^{\blacklozenge}$	OpenShift Origin executes arbitrary shell commands in URL	Arbitrary	Sandbox
CVE-2014-3791*	Easy File Sharing Web Server buffer overflow in cookie parsing of vfolder.php	Arbitrary	Sandbox
CVE-2014-3804 (+5)$^{\blacklozenge}$	AlienVault OSSIM executes arbitrary commands with crafted requests	Arbitrary	Sandbox
CVE-2014-3829$^{\blacklozenge}$	Centreon Enterprise Server executes arbitrary commands from command_line variable	Arbitrary	Sandbox
CVE-2014-3913*	Ericom AccessNow Server buffer overflow in AccessServer32.exe	Arbitrary	Sandbox
CVE-2014-3915$^{\blacklozenge}$	Tivoli Storage Manager executes arbitrary commands	Arbitrary	Sandbox
CVE-2014-4121$^{\spadesuit}$	Microsoft .NET improperly parses internationalized resource identifiers	Arbitrary	Sandbox
CVE-2014-6321*	Schannel in Microsoft Windows Server executes arbitrary code via crafted packets	Arbitrary	Sandbox
CVE-2014-7192$^{\clubsuit}$	Node.js eval injection in syntax-error package	Arbitrary	Sandbox
CVE-2014-7205$^{\clubsuit}$	Node.js eval injection in internals.batch() of lib/batch.js	Arbitrary	Sandbox
CVE-2014-7235$^{\spadesuit}$	FreePBX executes arbitrary code via ari_auth cookie in htdocs_ari/includes/login.php	Arbitrary	Sandbox
CVE-2014-7249*	Allied Telesis buffer overflow via crafted HTTP POST request	Arbitrary	Sandbox
CVE-2014-8361$^{\spadesuit}$	miniigd SOAP service executes arbitrary code via crafted NewInternalClient request	Arbitrary	Sandbox
CVE-2014-8661 (+1)$^{\blacktriangle}$	SAP CRM executes arbitrary commands via unspecified vectors	Arbitrary	Unclear
CVE-2014-9190*	Schneider Electric Wonderware InTouch Access Anywhere Server buffer overflow	Arbitrary	Sandbox
CVE-2014-9371$^{\spadesuit}$	ManageEngine Desktop Central MSP executes arbitrary code via crafted JSON object	Arbitrary	Sandbox
	Path Traversal		
CVE-2014-0598$^{\sharp}$	Novell Open Enterprise Server allows directory traversal	Unclear	Prevented
CVE-2014-0754$^{\blacksquare}$	SchneiderWEB allows directory traversal	Data leak	Prevented
CVE-2014-2863 (+1)$^{\sharp}$	PaperThin CommonSpot allows absolute path traversal	Unclear	Prevented
CVE-2014-3914$^{\flat}$	Tivoli Storage Manager allows arbitrary filesystem access	Arbitrary	Sandbox
CVE-2014-7985$^{\flat}$	EspoCRM allows remote include/execute via install/index.php	Arbitrary	Sandbox
CVE-2014-9373$^{\flat}$	ManageEngine NetFlow Analyzer executes arbitrary code via .. in the filename	Arbitrary	Sandbox
	Improper Authentication		
CVE-2014-0648$^{\diamondsuit}$	Cisco Secure ACS improperly enforces admin access	Arbitrary	Auth
CVE-2014-2075§	TIBCO Enterprise Administrator improperly enforces admin access	Arbitrary	Auth
CVE-2014-2609$^{\diamondsuit}$	Java Glassfish Admin Console in HP Executive Scorecard doesn't check authentication	Arbitrary	Auth
CVE-2014-8329$^{\heartsuit}$	Schrack Technik microControl stores sensitive information publicly in ZTPUsrDtls.txt	Arbitrary	Auth
	SQL Injection		
CVE-2014-3828$^{\infty}$	Centreon Enterprise Server SQL injection	Data leak	Sandbox
CVE-2014-5503$^{\infty}$	Sophos CyberoamOS SQL injection in Guest Login Portal	Data leak	Sandbox

Figure 8: Security analysis of all 40 web-related vulnerabilities in the National Vulnerability Database from 2014 with the highest severity score. In most cases, an attacker would be able to arbitrarily affect the service or access data. In Radiatus, we either prevent these attack entirely, restrict compromise within the user container sandboxes, or are mitigated by the delegation of authentication logic to Radiatus, which can be shared and independently audited.

- Unclear: bug report did not specify enough details to make a determination.

In most cases, the original impact allowed an attacker to arbitrarily affect the service or access data. In Radiatus, we either prevent the attack entirely, restrict compromise to the user containers for which the attacker has user credentials, or require applications to delegate authentication logic to Radiatus, which can be shared and independently audited.

5.1.1 Code Injection

The majority of high severity vulnerabilities involved code injection, which allows an attacker to execute arbitrary code with the privileges of the server process and affect the state of any user. In Radiatus, packets are only routed to a user container if the attacker has proper credentials, limiting the scope of the attack.

- 6 (*) consist of stack-based buffer overflow vulnerabilities, which can be remotely exploited by sending crafted network requests.
- 7 ($^{\spadesuit}$) allow arbitrary code execution due to improper sanitization of inputs to components, such as JSON parsers.
- 3 ($^{\clubsuit}$) involve code injection vulnerabilities in JavaScript (e.g. calling *eval(...)* in Node.js).
- 9 ($^{\blacklozenge}$) are vulnerabilities that allow a remote attacker to run shell commands with the privileges of the server process.
- 2 ($^{\blacktriangle}$) involve code injection via unspecified vectors.

5.1.2 Path Traversal

Similar in nature to shellcode injection, 7 vulnerabilities involved path traversal bugs (e.g. by adding ../ in the requested resource). When exploited these bugs can allow an attacker to read, write, and execute files from the filesystem with the privileges of the server process. Radiatus confines all application code in per-user sandboxes.

- 1 ($^\blacksquare$) allows remote attackers to read arbitrary files via crafted HTTP packets.
- 3 ($^\flat$) consist of variations where an attacker can execute arbitrary commands by using '../' to navigate to shell commands in input parameters.
- 3 ($^\sharp$) are known path traversal vulnerabilities with unknown impact.

5.1.3 Improper Authentication

4 vulnerabilities involved improper authentication checks. Most web frameworks require developers to write custom authorization logic. Flaws in this logic can easily lead to privilege escalation. In Radiatus, developers delegate authentication checks and enforcement to Radiatus, which can be independently audited and shared across all applications.

- 1 ($^\heartsuit$) stored sensitive credentials in publicly accessible resources, which attackers can use to obtain privileged access.
- 2 ($^\diamondsuit$) do not require authentication to access a privileged interface.
- 1 (§) does not require authentication to issue arbitrary commands as an administrator.

5.1.4 SQL Injection

2 vulnerabilities ($^\infty$) involve SQL injection, allowing an attacker to exfiltrate data. 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. SQL injection has become less common with the prevalence of parameterized client libraries. The number of SQL injection vulnerabilities in the National Vulnerability Database has fallen from its peak of 1476 in 2008 to 261 in 2014.

5.1.5 Other Vulnerabilities

Cross-site scripting: The most common web vulnerability is cross-site scripting. No cross-site scripting vulnerability in 2014 had a severity score exceeding 8.0. Radiatus is not aimed at these vulnerabilities, but our implementation addresses them by using industry-standard CSP Policies [3].

Cross-container communication: While Radiatus makes it harder for wide-scale compromise of an application, it is still possible to use cross-container messaging as a new attack vector if the attacker can gain access to it (e.g. by exploiting a code injection vulnerability in the user container). We restrict which containers can communicate, enforce typed interfaces, detect anomalous behavior, restrict resource consumption, and evict bad actors from the system. Browsers [10, 11], operating systems [50], and cloud providers [2] use similar techniques to protect communications between mutually distrusting isolated processes.

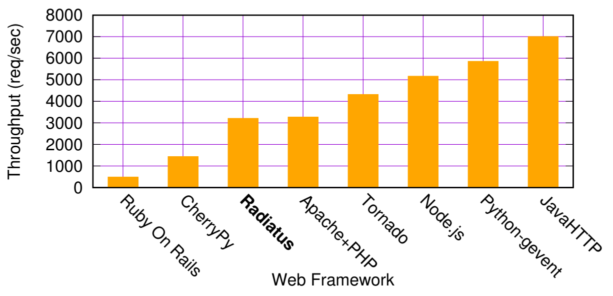


Figure 9: Comparison of single web server performance using the Siege Benchmark to make 1000 parallel connections at a time. Radiatus remains competitive with other frameworks despite handling requests in isolated user containers.

5.2 Performance

Radiatus is designed to achieve better security at modest overhead. We evaluated the performance of user containers on Amazon Web Services using r3.large EC2 instances (2 CPU, 15GB memory, 32GB SSD, $0.175/hr in 2015). We stress test the performance of a single web server and the overall throughput of a web service over 500 servers.

Memory Overhead: In Radiatus, each user container is a sandboxed process, running a separate copy of Node.js, the Radiatus runtime library, and the web application. 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 memory consumption across 100 user containers running our benchmark suite and found the average container to consume 30.5MB. As such, each memory-optimized r3.large EC2 instance was able to support around 490 processes before swapping.

Throughput Microbenchmark: 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 was an HTML page displaying a simple counter of how many replies had been served so far. This experiment disables caching while stress testing the HTTP request handler. While Radiatus performs additional routing to send requests to the proper container, our system performs comparably to existing frameworks and better than some popular frameworks, such as Ruby on Rails. Because the Radiatus router was written in Node.js, the microbenchmark shows a maximum overhead of 60.7% over our baseline. We expect the relative overhead to be less in a real web application, but we did not directly test that effect.

Macrobenchmarks: In order to stress test the system at scale and evaluate the performance of the cross-container messaging system, we set up a cluster of 500 virtual machines (VMs) supporting 180,000 user containers, communicating through the AWS Simple Queuing Service. As a point of comparison, Wikipedia in 2010 had 205 Apache

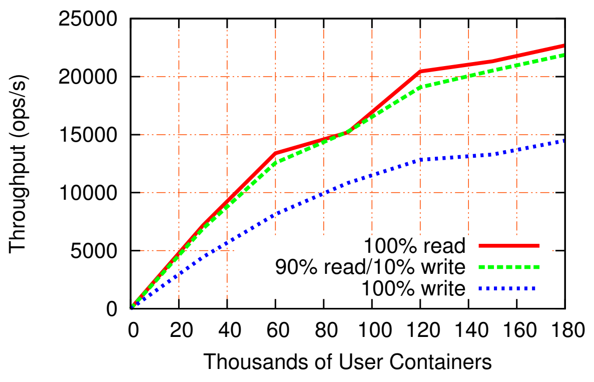


Figure 10: Aggregate throughput with different workloads across a 500-node cluster. We scale the number of user containers across our cluster, sending messages to each other via AWS Simple Queuing Service.

web servers to support 414M readers and 100K active editors per month, with 2000 HTTP requests per second [24]. In our benchmark, we stress tested our file-sharing application under varying workloads. A read request consisted of a request to the storage guard and a response containing an item from the user's personal storage. A write request consisted of sending a message to a peer container through the message router. Each user container performs either a read or write request every 4 to 6 seconds. Figure 10 shows the aggregate throughput of the system with various workloads. In Radiatus, the global performance can be bottlenecked by both the database and the message queue. Developers will need to properly balance these resources to fit the application workload.

5.3 Cost Estimation

The largest incremental cost of scaling a web service in Radiatus is the memory overhead of running user containers for each active user. As reported in § 5.2, a single user container consumes 30.5MB of memory when running our benchmark suite. Using average DRAM prices in 2015 [6] and an average server life of 3 years, we can estimate the incremental memory cost of an active user to be \$0.007/year. To include the additional cost of CPU overhead, power, and space, we use current EC2 pricing (2 cores @\$0.175/hour). Assuming 1000 requests/day for each user and using the differential CPU cost of handling requests in Radiatus versus Node.js from Figure 9, brings the total cost to \$0.008/user/year.

In order to estimate the potential cost of scaling such a system up to traffic levels seen by some of the biggest web apps today, we can use Little's Law and publicly reported numbers from the Facebook Newsroom [7]. On average, Facebook supports 26 million concurrent users on the site, for an estimated additional annual cost of \$200K. Note that this cost will multiply in a service-oriented architecture, depending on the number of internal services using Radiatus. Compared to the average cost of a data breach recovery in the U.S. (\$5.4M [64]), Radiatus may be appropriate for security-conscious web applications with sensitive high-value data.

6. Related Work

System	Cross-site Scripting	SQL Injection	Improper Access Controls	Buffer Overflow	Code Injection	Path Traversal	Improper Input Validation	Information Exposure	Resource Management	Improper Authentication
Radiatus	✗	✓	✓	✓	✓	✓	✓	✗	✓	✗
Process Isolation [31, 54, 55, 62]	✗	✓	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✗	✓[1]	✗
IFC [40, 46, 53, 61, 77]	✗	✗[2]	✓	✗[2]	✗[2]	✗	✗[2]	✓	✗	✗[2]
Encryption [45, 65, 66]	✗	✓[3]	✓	✗[3]	✗[3]	✗	✓[3]	✓	✗	✓[3]
Monitoring / Firewall [29, 52, 59, 70, 74]	✗	✓[4]	✗	✓[4]	✓[4]	✗	✓[4]	✗	✓[4]	✗
Browser Isolation [49, 51, 58, 72]	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗

Figure 11: Categories of vulnerabilities mitigated by web application security techniques. [1]Radiatus expands on these works to make per-user isolation practical at scale. [2]Invalid flows can be blocked, but potential for compromise still exists. [3]While mitigated by data encryption, bugs of this type remain possible. (e.g deletion through SQL injection.) [4]Intrusion detection systems use heuristics to deny requests, but potential for compromise remains.

Server-side Frameworks: In Figure 11, we categorize server-side web security solutions. These impose code structure limits in exchange for security guarantees.

OKWS [54] and Passe [31] 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. Passe also introduces a mechanism for automatically generating a security policy by monitoring accesses during normal operation. Many web services now use a service-oriented architecture [19] for a variety of reasons beyond security. πBox [55] introduces a per-user sandbox that spans a mobile app and web server; it interposes on all communication between users, with the goal of providing a end-to-end privacy-preserving mobile-cloud platform. CLAMP [62] was the first to introduce per-user sandboxes for server-side code execution, spawning a new virtual machine for each user session. In order to more easily port existing web applications written in a shared-everything model, they required that the developer specify an access control policy that properly limited each user's view of the database. Specifying a correct data policy can easily become intractable when specifying policies across tables (e.g. link tables for many-to-many relationships) and across inter-operating internal services. None of these systems were evaluated beyond a single machine deployment. Radiatus is the first to apply a capability-based security model in a shared-nothing architecture for web workloads. We evaluate the strengths, weaknesses, and performance implications of this model for security-conscious web applications.

Information flow control (IFC) can be used to limit the ability of a corrupted application to exfiltrate information [53, 77]. Hails [46] uses IFC to track privacy violations when third-party applications run on data provided by a web service. PHP Aspis [61] uses IFC to guard against injection attacks and DBTaint [40] tracks IFC across different applications. Other web frameworks [32, 69], attach fine-grained security policies on data. These frameworks assume the existing centralized model of web development. While IFC systems can block invalid flows, it does not prevent service disruption or all forms of exfiltration.

CryptDB [65], Mylar [66] and homomorphic encryption [45] have been proposed as ways to perform certain computations over encrypted data. Consequently even if a service gets compromised, users can rest assured that their data is safe. These systems place greater limits on behavior and evolution than does Radiatus.

A few projects have also explored variants of partitioning server-side application logic. BStore [33], Lockr [73] and RemoteStorage [20] provide mechanisms for application logic to be detached from storage, allowing storage to be provided by a third party.

Server-side Monitoring and Code Analysis: A variety of black box techniques have been proposed to detect attack signatures [63, 67, 74], block known attack vectors [23, 59, 70], replay attacks [29, 52], and recover state after attacks [34, 35], without modifying the server-side code. However, recent studies [28] show that black box testing miss many important vulnerabilities in the wild. Other techniques analyze source code for vulnerabilities [30, 68, 71, 78], or use symbolic execution to detect violations of an application specification [36, 43].

Client-side Browser Security:

Weak isolation is recognized as an important security problem in web browsers. A variety of browsers [10, 11, 48, 49, 58, 75] and client-side JavaScript libraries [51, 72] have explored isolation techniques for web applications and were influential in the Radiatus design. Because Radiatus is a server-side framework, it is complementary to client-side isolation.

7. Conclusion

Modern trends in OS-level containers, cost of memory, and elastic cloud computing make it an opportune time to revisit per-user isolation and study the costs at scale. Radiatus provides an alternative model for web application design offering increased security over existing frameworks. User containers are a lightweight mechanism to strongly isolate users within a web application. We show that it is practical to provide per-user isolation, while offering performance competitive with existing web frameworks, at modest cost per user. While application design with Radiatus is different from traditional frameworks, we show that our APIs are expressive enough to support many of the web applications today.

The web platform already treats the browser as a per-user isolated container running potentially untrusted code. Leveraging this design pattern on the server provides a structured approach to isolation, offering the same containment we expect from our own machines, mobile applications, and multi-tenant data centers.

Acknowledgments

This work was supported by the National Science Foundation (CNS-1318396 and CNS-1420703).

References

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

[2] Amazon Web Services. `https://aws.amazon.com`

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

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

[5] Docker. `https://www.docker.com`

[6] Average selling price of DRAM 1Gb equivalent units from 2009 to 2017. `http://www.statista.com/statistics/298821/dram-average-unit-price/`

[7] Facebook Newsroom. `https://newsroom.fb.com/company-info/`

[8] Firebase. `https://www.firebase.com`

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

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

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

[12] Kernel Virtual Machine. `http://www.linux-kvm.org/page/Main_Page`

[13] lmctfy. `https://github.com/google/lmctfy`

[14] Node.js. `http://nodejs.org/`

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

[16] Open Container Initiative. `http://www.opencontainers.org/`

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

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

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

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

[21] Linux-VServer. http://linux-vserver.org/

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

[23] greensql. http://www.greensql.com/ 2009.

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

[25] F. Almroth and M. Karlsson. How we got read access on Googles production servers. http://blog.detectify.com/post/82370846588/

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37 (5):164–177, 2003.

[27] D. Barry and T. Stanienda. Solving the Java object storage problem. *Computer*, 31(11):33–40, 1998.

[28] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[29] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 607–618. ACM, 2010.

[30] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 575–586. ACM, 2011.

[31] A. Blankstein and M. J. Freedman. Automating isolation and least privilege in web services. In *IEEE Symposium on Security and Privacy (SP)*, 2014.

[32] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. GuardRails: A data-centric web application security framework. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.

[33] R. Chandra, P. Gupta, and N. Zeldovich. Separating web applications from user data storage with bstore. 2010.

[34] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 101–114. ACM, 2011.

[35] R. Chandra, T. Kim, and N. Zeldovich. Asynchronous intrusion recovery for interconnected web services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 213–227. ACM, 2013.

[36] A. Chaudhuri and J. S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 585–594. ACM, 2010.

[37] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 227–238. ACM, 2011.

[38] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 97–106. IEEE, 2005.

[39] S. Curtis. Barclays: 97 percent of data breaches still due to SQL injection. http://news.techworld.com/security/3331283/

[40] B. Davis and H. Chen. Dbtaint: cross-application information flow tracking via databases. In *2010 USENIX Conference on Web Application Development*, 2010.

[41] C. Dixon, T. E. Anderson, and A. Krishnamurthy. Phalanx: Withstanding multimillion-node botnets. In *NSDI*, volume 8, pages 45–58, 2008.

[42] Ú. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1054–1067. ACM, 2014.

[43] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, pages 143–160, 2010.

[44] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. In *OSDI*, pages 13–13. USENIX Association, 2000.

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

[46] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 47–60, 2012. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387886.

[47] P. Graham and R. Morris. Arc forum. http://arclanguage.org/forum 2008.

[48] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *HotOS*, volume 7, pages 1–7, 2007.

[49] J. Howell, B. Parno, and J. Douceur. Embassies: Radically refactoring the web. *NSDI*, 2013.

[50] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2): 37–49, 2007.

[51] L. Ingram and M. Walfish. Tingram2012treehousoreehouse: Javascript sandboxes to help web developers help themselves. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[52] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 193–206. USENIX Association, 2012.

[53] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 321–334, 2007.

[54] M. N. Krohn. Building secure high-performance web services with OKWS. In *USENIX Annual Technical Conference, General Track*, pages 185–198, 2004.

[55] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. πbox: A platform for privacy-preserving apps. In *NSDI*, pages 501–514, 2013.

[56] R. A. McClure and I. H. Krüger. Sql dom: compile time checking of dynamic sql statements. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 88–96. IEEE, 2005.

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

[58] J. Mickens and M. Dhawan. Atlantis: robust, extensible execution environments for web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 217–231. ACM, 2011.

[59] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. Spyproxy: Execution-based detection of malicious web content. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, number 3, pages 1–16. USENIX Association, 2007.

[60] National Vulnerability Database. Vulnerability summary for cve-2014-7235. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7235` 2014.

[61] I. Papagiannis, M. Migliavacca, and P. Pietzuch. PHP Aspis: using partial taint tracking to protect against injection attacks. In *2nd USENIX Conference on Web Application Development*, page 13, 2011.

[62] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *30th IEEE Symposium on Security and Privacy*, pages 154–169. IEEE, 2009.

[63] V. Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999. URL `http://www.icir.org/vern/papers/bro-CN99.pdf`.

[64] Ponemon Institute. 2013 Cost of a Data Breach Study: Global Analysis. `http://www.ponemon.org/`

[65] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[66] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *USENIX Symposium of Networked Systems Design and Implementation*, 2014.

[67] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[68] M. Samuel, P. Saxena, and D. Song. Context-sensitive autosanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 587–600. ACM, 2011.

[69] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Usenix Security*, 2012.

[70] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 601–614. ACM, 2011.

[71] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *USENIX Security Symposium*, 2011.

[72] J. Terrace, S. Beard, and N. P. K. Katta. JavaScript in JavaScript (js.js): Sandboxing third-party scripts. 2012.

[73] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: better privacy for social networks. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, pages 169–180. ACM, 2009.

[74] G. Vigna, W. Robertson, V. Kher, and R. A. Kemmerer. A stateful intrusion detection system for world-wide web servers. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*. IEEE, 2003.

[75] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 1–16. ACM, 2007.

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

[77] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[78] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: automated detection and quantification of side-channel leaks in web application development. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 595–606. ACM, 2010.