Attackers as Instructors: Using Container Isolation to Reduce Risk and Understand Vulnerabilities

Yunsen Lei[1], Julian P. Lanson[1], Craig A. Shue[1], and Timothy W. Wood[2]

[1] Worcester Polytechnic Institute {ylei3,jplanson,cshue}@wpi.edu
[2] George Washington University timwood@gwu.edu

Abstract. To achieve economies of scale, popular Internet destinations concurrently serve hundreds or thousands of users on shared physical infrastructure. This resource sharing enables attacks that misuse permissions and affect other users. Our work uses containerization to create "single-use servers" which are dynamically instantiated and tailored for each user's permissions. This isolates users and eliminates attacker persistence. Further, it simplifies analysis, allowing the fusion of logs to help defenders localize vulnerabilities associated with security incidents. We thus mitigate attacks and convert them into debugging traces to aid remediation. We evaluate the approach using three systems, including the popular WordPress content management system. It eliminates attacker persistence, propagation, and permission misuse. It has low CPU and latency costs and requires linear memory consumption, which we reduce with a customized page merging technique.

1 Introduction

Internet servers are designed to handle many clients simultaneously. These servers use multiple processes or threads of execution to balance requests and make effective use of computing resources. Unfortunately, this model intermingles processing from many clients within a single execution context. When these servers have security defects, attackers can exploit the vulnerabilities to gain unauthorized access, modify the server's content, and harm other current or future users [31].

Exacerbating this problem, when a server accesses other resources, such as databases, it is often configured with a super-set of all privileges associated with the server's users. This can lead to "confused deputy" attacks [13], wherein an adversary exploits a vulnerability to cause an application server to misuse its authority when interacting with a resource provider. SQL injection attacks, which are estimated to be used in nearly two-thirds of all web attacks [3], are a common form of confused deputy attack.

In this work, we propose a "single-user server" model where each incoming client gets directed to its own isolated container. We explore a set of research questions: *How can this single-use server model limit attack propagation, persistence, and privilege escalation? Can containerization provide low enough overheads to support a large number of concurrent users? To what extent can we* *improve the resource consumption of the approach? What impact can the single-use server model have on attack reconstruction and analysis?*

The first two research questions lead to novel contributions in container management and access control. Our single-use server model places every application server in its own Docker container with permissions tailored to the associated end user. When a client first connects to a server, it has anonymous user privileges and tightly constrained access to backend resources, such as a database. When a user authenticates, our approach automatically alters the permissions associated with the container to match the privileges associated with the authenticated user. Since the permissions for each application server and container are tailored, they do not have the elevated privileges necessary to enact a confused deputy attack. The container approach provides isolation and the destruction of a container upon the client's disconnection eliminates attacker persistence.

The third research question leads to novel contributions to memory deduplication. Our approach, called Focused Kernel Same-page Merging (FKSM), actively merges two container's processes if they run the same programs.

The final research question leads to novel contributions in attack analysis and localization. To detect access violations, we create monitoring infrastructure for communication between clients and the application server as well as between the application server and any back-end resources. This monitoring also enables forensic reconstruction of attacks. In this work, we:

- **Design a Single-Use Server (SuS) approach** that includes the components needed for authentication, container management, and the collection of forensics for arbitrary applications (Section 3). *Our design improves security by cleanly separating the untrusted execution environments for individual users from each other and from the control plane that routes, authorizes, and monitors requests.*
- **Implement a Single-Use Web Server** using a novel combination of lightweight containers and network middleboxes to support three web application services, including the popular WordPress platform (Section 4). *Our implementation demonstrates that applications can be ported with minimal codebase modifications. We also enable fine-grained permissions to be safely enforced by proxy middleboxes. We further develop a memory deduplication approach that saves 26% of memory for each container while the merge time is only a fraction of the state-of-the-art UKSM [40] approach.*
- **Evaluate the security and performance** of our SuS implementation (Sections 5 and 6). *We find the containerization approach prevents several exploits against vulnerable versions of WordPress without requiring application software patches. It incurs less than 5% CPU overhead, needs only 2GB of RAM to run 100 concurrent containers, and shows only a 20% increase in response time when running 100 concurrent containers.*
- **Reconstruct attacker steps** by leveraging the per-user logging enabled by SuS (Section 5.2). *When exploring a known CVE attack on our SuS WordPress system, we find that a back-tracing workflow can quickly localize the search space for debugging and remediation, reducing the search space from thousands of files to only two functions.*

2 Background and Related Work

In this section, we review work in the most related areas and discuss how our approach is different from various perspectives.

Security through Isolation: Parno et al. proposed CLAMP to protect LAMP-stack websites [27]. CLAMP assigns individual users to isolated VMs running copies of the web server code. Users can upgrade their VM's permissions via a separate, trusted authentication portal. Unfortunately, CLAMP provides only 42% of the performance of native operations. The CLAMP authors acknowledged significant impediments to the practical deployment of such a system and did not complete an analysis of VM start-up on normal operation, citing the significant overhead of VM start-up and limitations of delta virtualization. In contrast, in our work, we designed and implemented customized memory improvements and performed an end-to-end evaluation, including on-demand server generation. Our result shows that SuS incurs modest overheads and achieves greater scalability. Taylor [36] introduces a software-defined networking (SDN) controller to demultiplex users' traffic and guide their packets to isolated VMs. The Taylor work lacks the resource restriction component present in CLAMP, but adds attack attribution. Our SuS model improves performance and scalability; further, it uses log fusion to reconstruct attack steps, which was not previously explored.

Radiatus, by Cheng et al. [6], builds off CLAMP and introduces more stringent security measures. The result of such a design is a web development framework that requires developers to use their API to create an application. Porting an existing application to use Radiatus thus requires re-implementation. Our SuS model aims to provide strong security isolation and can be deployed on widely used web applications like WordPress and HotCRP with minimal code base modification ($\leq$ 50 lines of code).

Lighter Weight Virtualization: Some Internet services use lightweight virtualization like containers to facilitate fast deployment, fine-grained scaling, and component failure isolation [34, 35]. Prior work has also sought ways to reduce the resource consumption and cost of starting and running applications with these technologies [15, 2]. Serverless computing platforms benefit from these lightweight virtualization technologies. SuS is different from the serverless model. Our work takes a user-based view of the application and constrains the user's behavior based on the functions and resources the user is supposed to access. We focus on security and forensics aspects.

Memory Deduplication: Kernel Same-page Merging (KSM) on Linux allows applications to share identical pages by comparing the page content. Previous work improves KSM in terms of scanning speed and resource utilization [33, 11, 41]. UKSM [40] improves KSM by prioritizing statically-duplicated memory regions and reducing computational cost through Adaptive Partial Hashing [12]. Our merging approach differs from these existing works in the way duplicate pages are identified. We compare our FKSM with UKSM on deduplication speed and effectiveness to show the benefit of active and strategic scanning.

Forensic Analysis of Exploits: Data records can help defenders understand, analyze and replay past events that are related to attacks. Dunlap et al. [8]

proposed Revirt, which uses checkpoint logging and roll-forward recovery to replay entire attack events. To facilitate the human understanding of collected forensics, researchers have proposed different approaches [10, 7, 32] to visualize the data. In our work, we focus on constructing execution traces in an informed way that leverages per-user isolation, facilitating visualization integration.

3 An Untrusted Application Server Design

Application servers are complex, making them ripe for attack. We create monitoring and protection components so that attacks become valuable learning opportunities for defenders to improve software without negative outcomes.

3.1 Threat Model

The SuS model is a server-side defense system aimed at preventing adversaries 1) from successfully executing any backend request above their intended privilege level and 2) from making changes to server files that would enable them to attack other users. We assume that adversaries can only access the server program's host machine via network communication and that they will attempt exploits via the packet payloads in the server program's communication protocol.

We assume the application server within a single-use container can be exploited and adversary-controlled. The adversary may arbitrarily control one or more clients and containers. We assume that the container facilitates access to information stored in one or more backend resources (e.g., in databases or file shares) but that it otherwise only stores per-user session state. For the defender, we assume that they leverage the SuS capability to configure the levels of access based on their applications and different user roles. Such configuration exercises a least-privilege principle, helping defenders mitigate exploitation against unknown vulnerabilities.

We exclude attacks that cause privileged users to misuse their legitimate privileges, such as social engineering or cross-site request forgery. Similarly, we exclude attacks against our trusted computing base (TCB), which includes the operating system, the back-end resource servers, and the SuS infrastructure components themselves (such as middleboxes and container managers). While we evaluate container scalability and performance, we exclude flooding-based denial-of-service attacks and assume defenders employ current best practices. While a trusted kernel is a common threat model assumption, and one we use as well, we recognize that efforts to escape a container and elevate privileges are possible. Given the importance of kernels and containers, we anticipate other continued efforts to improve and protect them. The scope of the SuS approach is to explore the feasibility of using lightweight virtualization to run server instances that are tailored to a single user. Our approach could be used with other lightweight isolation and virtualization technologies as needed.

3.2 Design Components

Our SuS model assumes that each server instance will support only a single client, although this could be extended to enable a container to serve a group of related users, albeit with no protection between them. We will place each application server in a separate container and examine each container for indications of compromise (IOCs) that merit further analysis. Any container that lacks an IOC is deleted once it is no longer needed by a client. Figure 1 shows the following SuS model components. We now describe these components in detail.

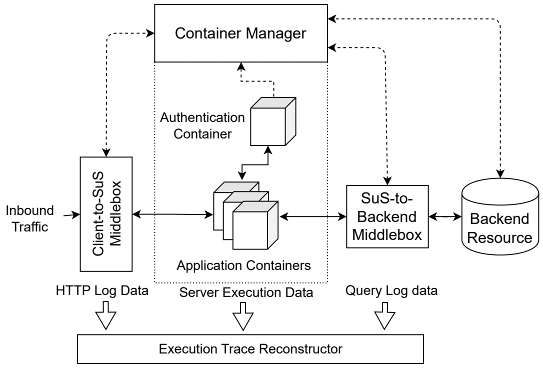

Fig. 1: Design overview of Single-use Server architecture. The middleboxes, authentication, and management components coordinate to provision SuS containers and assign them to clients, provide a means of upgrading privileges to the backend resource, and contain and analyze exploited application servers.

3.3 Application Containers

We place each application server in its own isolated execution container. These containers are instantiated by cloning an existing server. Before the container interacts with a client, we consider the container to be "pristine." While it is pristine, the container can be trusted since it is unreachable by an adversary. Once a container interacts with a client, we consider the container "contaminated" and inherently untrustworthy.

3.4 Container Management

The centralized Container Manager mostly performs management tasks including server instance configuration, container provisioning, reclamation, or freezing (if an attack is detected). Because the Container Manager knows which pair of processes are created from the same resource, it also communicates with the FKSM kernel module to initiate page deduplication.

3.5 Authentication Container and Permissions

Many application servers must identify the user associated with a given client. In our model, we cannot rely on the untrusted application to accurately report the client's authentication status. Instead, all user authentication is handled by a trusted Authentication Container, which has a minimal code base that can be more easily verified and protected. This separation of roles is somewhat similar to the Kerberos authentication model [17].

The Container Manager communicates with the Authentication Container to adjust container permissions. If the Authentication Container confirms a client's identity, the Container Manager increases privileges in the SuS-to-backend middlebox and backend infrastructure to reflect the new user permissions. In essence, the SuS container gains only the privileges associated with the connected user.

Our privilege model differs from traditional web server configurations. An application container does not need a superset of users privileges during the configuration (e.g., WordPress installation recommends granting all privileges in the WordPress database). Such a configuration has the potential risk of letting adversaries ultimately control the entire WordPress database in case of an exploited WordPress instance. In our SuS model, the same exploitation is limited to the database privileges of the account associated with the exploited container. In other words, the adversary may issue queries, but the queries will only succeed if they can be performed within the limited permissions associated with the container. We treat database server privilege errors as evidence of compromise.

3.6 Client-Side Demultiplexing and Forwarding

Our Client-to-SuS middlebox acts as a load balancer that demultiplexes clients and directs each to a separate SuS container and as a proxy that handles all encryption functionality. This keeps cryptographic keys out of the untrusted SuS container environment while letting the middlebox vet unencrypted data. It controls access and blocks client communication in the event of an access violation. It logs network traffic for forensic reconstruction. This allows defenders to pinpoint the client messages that preceded the violation, potentially revealing the vulnerability exploited in the SuS container.

3.7 Guarding Backend Resources

The narrowly-tailored permissions the Container Manager configures for backend resources solve many Confused Deputy attacks. However, a SuS-to-backend middlebox provides fine-grained restrictions that some backend implementations cannot support. For example, an authenticated user should have `UPDATE` privileges to the application's `users` table so the user can change the associated email address or password. However, that privilege should be limited to certain rows, such as the rows for which the column `USERID` matches the authenticated user's identifier. The SuS-to-backend middlebox must be protocol-aware to perform resource access control effectively. We designed the system to easily swap between backend modules for protocols such as SQL and NFS. The modules in the middlebox must implement specific API functions that (1) parse and conditionally modify resource requests and (2) detect permission violations in request responses. The middlebox also observes any errors or responses from the backend resource and informs the Container Manager to act accordingly. The middlebox logs the communication for incident analysis of any attacks and it prevents access between potentially compromised SuS containers and the backend resources.

3.8 Constructing Execution Events

The client-side and backend-side traffic are logged by the corresponding middleboxes and associated with users' identities. With our design, this traffic is automatically separated to represent a single user's interaction with the service. While helpful, the network traffic alone is insufficient because it lacks insight into the user's interaction with the application. In Section 4.6, we describe the server profiling component that provides such detail. Since SuS logs the per-user server instances, it has a significant advantage in fusing and reconstructing logged data to facilitate the understanding of the provenance and the impact of an incident. We discuss the implementation of log integration in Section 4.7.

4 Implementation

To provide concrete examples and show evidence of generalization of our SuS design, we create implementations using three different Web applications: 1) WordPress [39], a popular Internet content management system estimated to be used in over 35% of all websites on the Internet [37]; 2) HotCRP [14], a system for managing paper submission and peer reviews for conferences; and 3) an anonymized learning management system (LMS) used in our organization for the administration and delivery of class materials. For simplicity, we focus on WordPress and simply describe where HotCRP and our LMS applications differ.

All three applications require a web server, a PHP runtime, and a database server. Since PHP is used in over 78% of popular websites [38], we focus on PHP web applications. We use the popular Docker container system to implement our containers. We test multiple versions of the WordPress software to measure the impact of the single-use server approach on attacks against versions of WordPress with known vulnerabilities. In the remainder of this section, we describe the implementation details of our SuS containers, the Container Manager, the Authentication Container, the middleboxes, and our protocols.

4.1 Container Configuration

We build a Docker container image through a "Dockerfile" for each application we need to protect in a SuS container. The application container is configured under a private network which is created through Docker's command line interface. We assign each container a unique IP and expose the necessary ports.

Under their standard deployment, most web applications must be configured to communicate external components such as server applications and resource databases. The SuS model separates server instances but does not fundamentally interrupt the data flow. Therefore, a similar configuration effort is required. Using WordPress, HotCRP, and our LMS as case studies, we find that only minor modifications are required and mainly involve the following aspects:

Authentication: The authentication logic must relocate from the untrusted container into the Authentication Container. For all three applications, we rewrite the login URL and direct the user to their assigned SuS container after successful authentication. For HotCRP, users must log in for most features and be directed to a pristine container. Our LMS application was configured as a relying party associated with a single-sign-on identity provider. Therefore, the redirection URL is encoded with the parsed SSO response.

Shared Resources: The SuS-to-Backend Middlebox (Section 4.5) can support backend resources such as databases. Some services, like HotCRP, use a mail server for message transmission. This can be handled via an external server or a shared service (which itself could be in the SuS system). For simplicity, we simply use a mail server on the container host in our experiments.

4.2 Container Manager

The Container Manager creates a thread to maintain a pool of available containers for new clients. Our pooling strategy hides latency by ensuring a container is ready when a client arrives. When stopped, the Container Manager terminates all threads and containers and removes container credentials from the database. We use a startup script as the entry point that setup control arguments for the rest of the container processes. After parameter configuration, the script then fires up the web application. The scripts receive these arguments from the Container Manager as part of the container startup process. An internal control manager handles the generation of each container's control arguments (including IP address, database credential with minimal privilege, PHP settings, etc.).

Optimizing memory usage for SuS is important. Our memory deduplicator is implemented as a kernel module in Linux and a userspace component which is part of the Container Manager. The communication between the kernel and userspace is achieved through a `ioctl` call in which the manager passes a pair of in-container process IDs that requires merging. Container processes' IDs are obtained by intercepting the Docker event interface and examining the corresponding `cgroup` directory on a container startup event. After receiving the process IDs, our kernel module's callback function scans the processes' pages. Before merging, we compute and store page metadata in a two-layered hashmap. This structure combines xxHash checksums with Blake2b checksums for each page to perform faster merge comparisons. The first layer is indexed off xxHash's first bytes, and the second layer stores the full xxHash and points to a red-black tree indexed off the Blake2b hash. The mergeable pages between two processes are first maintained in a link list and then merged using existing kernel functions.

4.3 Authentication Container

The Authentication Container operates a web page that prompts clients for credentials. Upon receiving the user's credentials, it tries to validate them and notifies the Container Manager whether the client has a new role. The Container Manager then accesses the database and appropriately upgrades the privileges of the account associated with the client's assigned SuS container. The Authentication Container finally redirects the user back to a specific URL on the appropriate SuS container. That URL encodes data that allows the script to set authentication cookies to set the user identity in the application.

This authentication model requires that the two redirect messages be cryptographically validated. We encode nonces and a message authentication code in the redirect messages to ensure the authenticity of all passed parameters. We derive the keys using information preconfigured by the Container Manager. This approach allows the Authentication Container to statelessly validate messages from any SuS container without requiring an interaction with the Container Manager. We omit the details of these messages for brevity.

4.4 Client-to-SuS Middlebox

The main task for the client-to-SuS container middlebox is determining the appropriate container for each client. We embed the user information in an HTTP cookie called `SUS_DEMULTIPLEX_COOKIE` to perform the client demultiplexing. We implement the middlebox using Python's `asnycio` library and use its API functions to handle the TLS termination. The middlebox can thus parse the HTTP request to extract the user information within the `Cookie` header. After parsing the HTTP header, it updates an internal mapping structure with the relation between the user information and its assigned SuS container IP. The validity of the `SUS_DEMULTIPLEX_COOKIE` determines whether a new container request is needed. After a SuS container is purged for inactivity, the middlebox removes the corresponding internal mapping. Upon receiving the first response from the SuS container, the middlebox rewrites the HTTP response in a `Set-Cookie` field with appropriate cookie value and expiration to ensure the client sends subsequent requests with the cookie value. It then encrypts the response message and sends it back to the client.

We also implement a second cookie named `SUS_LOG_COOKIE`, which is used only on the server side to uniquely name each request for logging and event reconstruction purposes (See Section 4.6 and Section 4.7).

4.5 SuS-to-Backend Middlebox

The SuS architecture ensures that the database can enforce the table-level constraint by itself. Fine-grained query scoping requires configuration similar to prior work [16]. In our work, we create a proxy middlebox between the SuS containers and the MySQL database. As described in Section 3.7, one task that middlebox performs is query scoping, which limits table access to certain rows. We define a *ResourceRestrictTable* that maps the tuple *(Role, Resource, Access Type)* to an *Access Predicate*. An Access Predicate is an extra limitation that can be applied to the query by appending it to the query's `WHERE` clause or an assertion to check the presence of a specific row selector in the `WHERE` clause. The middlebox also maintains a *UserContext* dictionary that maps container IP to user information (e.g., user_id, role). For each database query, the middlebox first retrieves the user's role based on the container IP. Then it extracts the resource (table) and access type (e.g., `SELECT`, `INSERT`). The middlebox retrieves the Access Predicate using the above three values. An access predicate may have a variable, as in the case `ID = :user_id`. In this case, the middlebox inserts the corresponding value from the UserContext dictionary entry before appending it to the query. The modified query is then sent to the MySQL database, and the response is forwarded to the container from which the query originated as usual. This silently restricts the data available to each user. The middlebox monitors and logs MySQL server responses for permission violations and regards any such error messages as an indication that the container has been compromised. Upon detecting such an error, the middlebox issues a request to freeze the container.

4.6 Tracing Application Execution

We implement a PHP extension that leverages request hooks to mark the start and end of a request and log important contexts such as URLs and cookies. The

SUS_LOG_COOKIE cookie (inserted by the Client-to-SuS middlebox as mentioned in Section 4.4) is extracted at the request_start hook function. In addition to the request information, we also leverage the function execution hook to record function execution information, including the function name, the function call site (the file and line at which the function is called), and the function's parameter values. In this hook, a SUS_LOG_COOKIE value will be propagated if the function execution is part of the request handling. Because PHP handles requests synchronously, this propagation is scoped by the request_start and request_end hook functions.

The profiling extension is application-independent and can be loaded and unloaded through the PHP runtime's configuration file when PHP processes start. We found that accessing a complete list of function parameter values can incur significant overhead. Therefore, we only obtain the first three elements' values for composite-type parameters. In addition, we limit the parameter tracing depth when a composite-type parameter contains other composite-types.

Since the profiling extension runs within each SuS container, the profiling data may be tainted. An attacker with control of the SuS container may manipulate the profiling extension to provide false data. We leverage Linux's auditd from outside the container to implement rules that monitor ptrace and accesses to PHP's configuration directory. These rules can effectively detect an attacker's attempt to subvert the profiling modules through code injection and module replacement. Previous work explores syscall semantic reconstruction for interpreted program [18, 5]. Tracing syscalls from outside the container can enable legitimacy estimates of the PHP execution trace. Mismatches between the syscall traces and PHP traces could themselves be indicators of container compromise.

4.7 Integrating Execution Traces

As mentioned in Section 3.8, logs from different users can easily be separated because of the single-use design. Our system generates: (1) an HTTP log from the Client-to-SuS middlebox, (2) PHP execution logs from profiling modules, and (3) a resource query log from the SuS-to-backend middlebox. These logs depict an interaction from different perspectives and, when integrated, constitute an execution trace of the whole event.

The first step in constructing the trace is parsing unstructured PHP logs into per-request call graphs. We implement a syntactical parser based on php-ast [30] to locate the user-defined function's definition (a script that defines the function and the line ranges of the implementation) and functions that will be called in the global scope. This statically-learned information is combined with our profiling data to construct the graph. The former allows us to determine the calling relation between two function log entries (e.g., whether function A is called within the block of function B). The latter allows us to identify the root node of a chain of function execution. The resulting request call graph is essentially a tree that starts with a root node named RINIT describing the request. Each child of the RINIT node is a global scope function followed by subsequent function calls and ends with PHP sink functions such as mysqli_query. To link the HTTP log to the request call graph, we only need to match the

`SUS_LOG_COOKIE` cookie. This approach allows us to link accurately even with the server application's URL rewriting. To link the call graph with the resource log query, we match the query string with the PHP function's parameter.

Table 1: CVEs and defenses considered in our security evaluation.

Category	CVE	Vulnerability Description	SuS Attack Mitigation
Single-user Instances	2012-3578 [20]	Input type validation failure enables script file upload and execution of arbitrary SQL queries and database credential leak	At SuS container startup, each `wp-config` PHP file is written with a database user of minimum privilege
Table-Level Privilege Constraints	2021-24182 [24]	Union-based SQL injection on `wp_tutor_quiz_question_answers`	Deny access to sensitive tables for unauthenticated or limited users (e.g., student roles)
	2021-24183 [25]	Union-based SQL injection on `wp_tutor_quiz_question`	
	2018-19207 [1]	Allows update to `wp_setting` table to register new admin account	Limit update access to the `wp_setting` table to administrator users
Row-Level Query Scoping	2019-9879 [21]	Privilege escalation exploit allows unauthenticated user to register new admin user	Query scoping prevents `wp_capability` used as the row selector when updating `wp_usermeta`
	2020-13693 [23]	Privilege-escalation exploit allows unauthenticated user to change `wp_usermeta` table to register with `bbp_keymaster` role	
	2019-9880 [22]	Allows unauthenticated user to retrieve all user information in `wp_user` table	Query scoping adds row selector to limit user access to only their own data
	2009-2762 [19]	Input validation failure allows reset of administrator's password for account hijack or account-level denial-of-service	

5 Security Evaluation

To evaluate the security benefits of the single-use server, we first consider the attacker's goal and common techniques in exploiting the confused deputy. Normally, attackers aim to access resources beyond what the application is designed for or what a user is allowed. This often requires the attacker to inject specific queries or misuse existing queries in the application code. For the injection case (the first three cases in Table 1), we consider two common attack vectors: file uploads and SQL injection. For the query misuse cases, we examine a set of attacks of this type (The fourth to eighth cases in Table 1).

We explore SuS effectiveness against attacks by classifying these exploits based on the SuS feature that stops or mitigates the attack. For each vulnerability, we apply an exploit to a test environment, both with our SuS model and without it. With the SuS model, the defender is required to configure the enforcement policies, while the control uses a shared server of the same software.

5.1 Evaluation: Real-Word Vulnerabilities

Single-user Instances For a server deployed using SuS model, file upload vulnerabilities are naturally mitigated because any uploaded script is only accessible within the attacker's container. Further, the uploaded script can only execute database queries within the limited permissions granted to that container (e.g., credentials saved in files like `wp-config.php`). We used CVE-2012-3578 [20] to evaluate SuS's effectiveness. As expected, the attack was successful in the shared server scenario. Our script, which aimed to delete WordPress accounts, failed in the SuS model since the container lacked the necessary permissions.

Table-Level Privilege Constraints Since each SuS container is configured with a unique database user account, we can configure different table access privileges based on the identity associated with the client. This prevents confused-deputy attacks since it eliminates privileged access that must be granted on a shared sever. For SQL injection attacks, queries which are manipulated to access any sensitive tables, such as `mysql.user`, are denied. In Table 1, we select three representative CVEs and show how SuS is configured to address these attacks.

Row-Level Query Scoping This class includes attacks in which a malicious user takes advantage of permissions that they were intentionally given in order to access or modify data that is disallowed by the security policy. This typically happens when the backend resource's native access control system is too coarse-grained to properly implement the desired policy. Our SuS-to-Backend middlebox enforces access control at the row level. We evaluate such a control's effectiveness through 4 different CVEs, as shown in Table 1.

5.2 Case Study: Exploring Execution Traces

While SuS can block the requests that trigger security exceptions, this alone does not help a security analyst to identify the root cause of an exploit. To illustrate how SuS logging can guide the process of localizing vulnerabilities, we consider a case study. We explore the *GdprOptions* vulnerability (CVE-2018-19207) and analyze the data. Using the trace construction workflow from Section 4.7, we construct a graph of relevant events from each HTTP request and SQL interaction. As mentioned above, the *GdprOptions* vulnerability is prevented by our SuS model because the `UPDATE` query needed to change the WordPress site settings exceeds the permissions associated with the client. This allows us to prune our analysis to graphs that contain the denied update query. The result, depicted in Figure 2, shows progression from an HTTP POST request (the first block) to `/wp-admin/admin-ajax.php`, and a series of PHP function calls that end with a denied SQL query (the last block).

In the trace, the PHP execution starts in the WordPress codebase and eventually enters the code base of a plugin (`gdpr142/Includes /Ajax.php`). After execution of the `processAction`, the query is prepared and processed by a sequence of query-related functions until it is issued by the `_do_query` function. Since the query is simply passed through those helper functions unmodified, it suggests that the issue originates in or near the `processAction` or

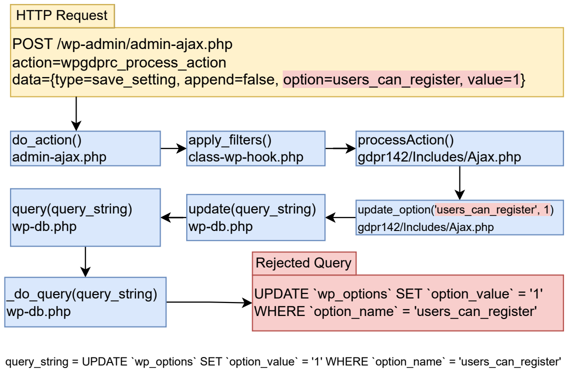
HTTP Request
POST /wp-admin/admin-ajax.php
action=wpgdprc_process_action
data={type=save_setting, append=false, option=users_can_register, value=1}
do_action()
admin-ajax.php
apply_filters()
class-wp-hook.php
processAction()
gdpr142/Includes/Ajax.php
update_option('users_can_register', 1)
gdpr142/Includes/Ajax.php
update(query_string)
wp-db.php
query(query_string)
wp-db.php
_do_query(query_string)
wp-db.php
Rejected Query
UPDATE `wp_options` SET `option_value` = '1'
WHERE `option_name` = 'users_can_register'
query_string = UPDATE `wp_options` SET `option_value` = '1' WHERE `option_name` = 'users_can_register'

Fig. 2: The pruned trace shows how the HTTP request (yellow) is handled by PHP functions (blue) and leads to the rejected SQL query (red). For readability, we use the string "query_string" as shorthand to represent the full SQL query that appears as parameters in the PHP nodes.

`update_option` functions. The documented patch to the vulnerability confirmed that the `processAction` was indeed the cause [26]. With such data, a defender can remedy the issue by patching the software or removing the plugin.

In a non-SuS system, this exploit might not be noticed for days or weeks, at which point logs may be overwhelming to analyze, and there will be no clear trail back to the request that triggered the exploit. In contrast, SuS's user separation allows the query to be rejected, signaling the need for immediate analysis. In this particular example, there are a total of 802 functions and 12 script files that are accessed between the HTTP request and the SQL query (assuming the analyst can identify the malicious request). SuS allows an analyst to quickly narrow the potential cause of the vulnerability to only two functions (`processAction` and `update_option`) within a single file (`gdpr142/Includes/Ajax.php`), out of the 1000+ PHP scripts in a WordPress installation.

We also conducted the same attack reconstruction analysis for other CVEs described in Table 1 as well. The log reduction benefit applies to other CVEs too; reconstructed traces average 18 functions with an average total of 1188 lines.

6 Performance Evaluation

Scalability and performance are key considerations for the SuS model. Since containers can use shared read-only mount points, and the remaining temporary file write space is needed in shared servers anyway, disk utilization is not a significant concern for the SuS model. However, we must explore what additional CPU and memory resources, if any, would be required by allocating separate server instances for each client and isolating them in separate containers. We must also explore the latency overheads associated with directing traffic to the appropriate container, logging its interactions, and enforcing permissions associated with those containers. We explore each of these topics in turn.

All our SuS containers run within a virtual machine with 16 GB of RAM, an allocation of 4 host CPU cores, and a 40 GB virtual hard drive. The VM runs on a physical host with 192 GB of RAM, 20 cores running at 2.20GHz, and 21 TB of hard drive space, configured with RAID. Our containers are not configured to use or enforce any CPU or memory limits. This configuration allows a comparison with the performance results associated with CLAMP [27].

6.1 RAM Usage

Before comparing memory usage between the SuS and the shared server model, we explore multiple server configuration options to ensure a fair comparison. We use WordPress as an example to show the impact of these options and determine the best choice for each model. We first configured different web servers using PHP with a static pool. For the shared and SuS server, the PHP worker pool size is set to the maximum number of concurrent users and one per container, respectively. The memory usage is calculated through Linux's `free` command. In the shaded portion of Table 2, we show the memory usage ratio (i.e., $\frac{SuS}{Shared}$) of the SuS model versus the shared server model. Our experiment shows that the shared server only uses a subset of the configured workers and achieves memory sharing that the SuS model does not.

Table 2: The ratio of memory ($\frac{SuS}{Shared}$) used by WordPress in the SuS model verses the shared server model across varying numbers of concurrent users in 10 trials of experiments. The shaded results omit copy-on-write sharing while FKSM uses such page sharing in the kernel.

Server Configuration	Concurrent Users	10	25	50	100
`Apache` PHP-FPM	original approach	3.94	5.01	6.97	8.1
	Our FKSM	2.72	3.37	4.71	5.39
`Nginx` PHP-FPM	original approach	4.13	5.08	5.67	7.37
	Our FKSM	2.85	3.47	3.86	5.0
`Lighttpd` PHP-FPM	original approach	4.02	4.96	5.48	6.9
	Our FKSM	2.89	3.33	4.65	5.59

For subsequent experiments, we select `nginx` with PHP-FPM as the default configuration for SuS because its relative lightweight and server popularity. Using PHP-FPM with a single PHP worker, we pre-spawn a fixed number of SuS containers and use web clients to interact with the servers. We measure active containers' memory usage while serving pages to clients. In Table 3, we see that the per-container memory usage decreases as the number of concurrent clients increases. This is likely the result of amortizing fixed costs.

The application server's basic properties play a significant role in the overall memory usage and the practical deployability of SuS. The 2.02 GBytes for 100 concurrent users can be easily handled by modern web servers. The web server hosting the LMS has ample memory and can easily scale to support the more

Table 3: Per-container memory usage (in MiB) with active clients across three applications using `nginx` and PHP-FPM. Results averaged over 10 trials.

Concurrent Users SuS Application		10	25	50	100
memory usage in MiB	WordPress	31.25	30.02	29.21	28.37
	HotCRP	27.80	27.71	27.55	27.16
	LMS	23.12	22.90	21. 14	20.70

than 1,000 active users in the system. Likewise, the HotCRP service can handle one hundred concurrent users with less than 3 GByte of RAM, which may meet the needs of most conferences. For high-volume websites such as WordPress, when considering our FMSK improvement, the memory usage will reduce to be relatively the same as the LMS application. But even without further optimization, compared with CLAMP's VM-based approach, each SuS container's memory usage is only half as much as a VM-based Webstack's (64 MB).

Table 4: The average merge time (t_m) and per-container memory saving (m_s) comparison between our FKSM and the UKSM approach across 10 trials with varying container counts. UKSM requires parameter tuning for best performance; the default works better in workloads with < 25 containers.

Concurrent Users KSM Approach		10	25	50	100
t_m **(secs)**	Our FKSM	1.24	3.24	8.51	18.56
	UKSM [40]	95.75	88.5	105.25	143.25
m_s **(MiB)**	Our FKSM	8.92	9.84	10.02	10.03
	UKSM [40]	4.98	5.30	5.14	5.00

We compare UKSM with our own Focused Kernel Same-page Merging (FKSM) approach, in which the container manager actively initiates the merging requests and records merge completion time and the average memory saved for each container. In contrast, UKSM constantly runs in the background without a clear merge completion point. Therefore, to make a fair comparison, we record the memory statistics for 10 minutes and choose the time needed for UKSM to complete 80% of memory savings for its performance statistics. We show these results in Table 4. FKSM focuses on mergeable pages only between pairs of processes, enabling quick merging. In contrast, UKSM makes multiple rounds of local and global samplings and may not discover all merging opportunities (resulting in 48% less memory saving on average). In the shared region of Table 2, we show the FKSM savings, which average around 30%. We examine how web retrievals can lead to unique states to process requests. We found that the memory usage will increase by around 2MB for both page-sharing approaches. Our approach still saves 26% of memory per container. We also found that when ASLR is enabled, the memory saving of FKSM and UKSM is significantly reduced to less than 5%. While the original UKSM paper [40] reports 39% of memory saving for containers, we found that this result is only achievable with a fully duplicated LAMP stack, where most saving is attributed to duplicated MySQL processes. In our settings, multiple containers share a single database with different credentials.

Table 5: The CPU usage (in percent) for both the shared and Single-use Server across 10 trials. For both deployments, the first load on WordPress triggers a bootstrapping process that causes the CPU usage to be higher than the second (and subsequent) page loads. In SuS, the first page load also causes the Container Manager to assign a container to the newly-connected user.

Concurrent Users Configuration		10	25	50	100
First Load	Shared	21.01%	23.19%	31.49%	41.33%
	SuS	56.85%	77.02%	85.94%	93.62%
Second Load	Shared	4.36%	10.10%	17.53%	34.83%
	SuS	16.26%	22.40%	27.60%	39.50%

Table 5 shows the median CPU usage (obtained using `mpstat`) for a four-core system with each tested case across 10 trials using the same container configuration as our per-container memory usage experiment with WordPress as the SuS-hosted application. The CPU usage difference between SuS and a shared server appears related to the processes needed to fork and isolate containers. For SuS, each container has a complete process set. For the shared server setup, it only needs to run a single server application and shared PHP worker processes. On the initial load, the 100 concurrent users each cause the accessed PHP scripts to be compiled on SuS, whereas in the shared server, a single compilation suffices due to PHP's OPcache [28]. This likewise explains the closer results for SuS and the shared server models on subsequent loads.

To avoid the PHP bootstrapping and compiling process, as mentioned above, we configured the Container Manager to perform this bootstrapping when creating a SuS container to prime it. Our implementation handles this by sending a pre-provision request to the fresh-started SuS container. We note that such a process can also be configured by the Opcache preload [29] feature but requires specifying the scripts in the correct dependency order to compile. Our request-based approach avoids this requirement.

6.2 Page Retrieval Times

We examine and compare the latency between SuS and Shared server using WordPress, which is known to be a heavyweight application [4]. We generate concurrent client requests using multiple instances of `wget` to get WordPress's main page (each main page access requires 7 different assets files and triggers 22 unique MySQL queries). While website complexity varies in practice, this experiment compares the workload's impact across different server configurations.

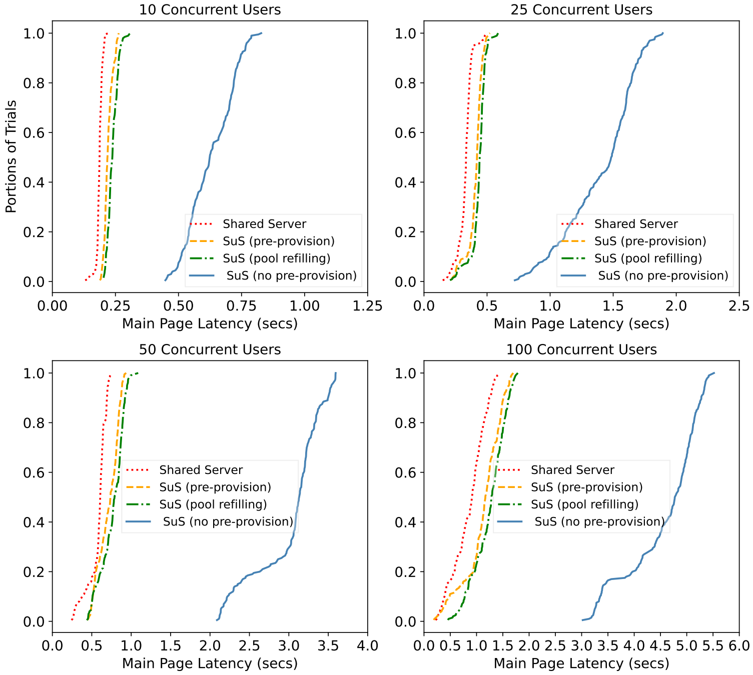

Fig. 3: CDFs of 200 home page load trials in WordPress 5.1.1. with 10, 25, 50, 100 concurrent users in SuS and shared server ("control") scenarios.

Figure 3 shows page load times under different settings. As we mentioned above, one overhead is the multiple script compilation process. Given this, the first two settings for SuS are with and without pre-provision. In addition, we consider a third pool refilling setting where the container manager maintains a pool watcher thread to ensure sufficient available containers.

From Figure 3, we see that when using pre-provisioned containers, the load times for the SuS and shared server models are similar, from 10 to 50 concurrent users. The SuS model becomes slower at 100 concurrent users. We believe there are two main sources of delay. First, the SuS model requires the CPU to spend extra cycles to context switch between processes, which is saved for shared servers because of sharing worker processes across requests. Second, our Client-to-SuS middlebox must request a new container from the Container Manager for the first request from a new client, adding initial latency and load. For the pool refilling setting, we observed only a minor impact on the page load time (characterized by the difference between the second (orange) and third (green) lines in Figure 3).

In CPU usage and page load time, SuS has results close to a shared server. When memory is not the bottleneck, SuS is able to achieve similar throughput as a shared server. This significantly outperforms the VM-based approach in CLAMP, which had only 42% of the throughput of a shared server [27].

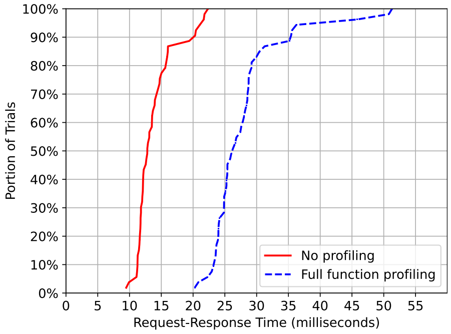

Fig. 4: Profiling affects page load time. Unless applied to production traffic, this logging overhead would occur in *post-hoc* analysis.

6.3 PHP Profiling Overhead

Our PHP profiling module adds extra runtime procedures to obtain the function's execution context. Then it asynchronously sends the collected traces to a profiling data receiver. The profiling overhead does not affect each asset retrieval, only the request handled by PHP. This experiment measures the overhead by comparing the PHP request's round trip time with and without the module.

Figure 4 indicates that the profiling adds around 10ms delay on individual PHP requests. WordPress makes many function calls (around 25,000 user-defined functions for each PHP request on average). Given this statistic, our profiling module adds less than 1μs for each function call. The PHP profiling overhead does not need to affect production traffic since analysts can disable this functionality in normal usage and only enable it in *post hoc* analysis in which the profiling module is enabled in a pristine container, and the previously logged HTTP request and back-end resource requests are replayed. Tools like TCPreplay [9] enable such event-based traffic replaying. Accordingly, defenders may choose whether to enable the feature for live traffic or only in incident response. Since our system prunes unrelated interactions, the replay logs may be small.

7 Conclusion

Our work introduces SuS containers that prevent adversaries from exploiting vulnerabilities in front-end Internet servers. These protections require only small code base alterations. Overheads introduced by the containerization approach are limited, with 2GB RAM sufficient for 100 containers and only a 5% increase in CPU consumption. The memory consumption of the approach is practical in some settings. Our FKSM saves 26% of memory for active containers. High-volume servers may benefit from future work in copy-on-write container cloning. This approach captures logs at the middleboxes and execution engine. The approach can allow analysts to reconstruct an incident and localize a vulnerability.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1814402 and 1814234.

References

1. Cybersecurity Help: Privilege escalation in gdpr compliance plugin for wordpress. `https://www.cybersecurity-help.cz/vdb/SB2018111101`
2. Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., Popa, D.M.: Firecracker: Lightweight Virtualization for Serverless Applications. In: USENIX NSDI (2020)
3. Akamai: Web Attacks and Gaming Abuse. State of the Internet **5**(3), 1–30 (2019), `https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-web-attacks-and-gaming-abuse-report-2019.pdf`
4. Budding, R.J.: Wordpress PHP performance benchmark. `https://www.savvii.com/blog/wordpress-php-performance-benchmark-2019/` (2019)
5. Bulekov, A., Jahanshahi, R., Egele, M.: Saphire: Sandboxing PHP applications with tailored system call allowlists. In: USENIX Security Symposium (2021)
6. Cheng, R., Scott, W., Ellenbogen, P., Howell, J., Roesner, F., Krishnamurthy, A., Anderson, T.: Radiatus: A shared-nothing server-side web architecture. In: ACM Symposium on Cloud Computing (2016)
7. Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., van Wijk, J.J.: Execution trace analysis through massive sequence and circular bundle views. J. Syst. Softw. **81**(12), 2252–2268 (2008). https://doi.org/10.1016/j.jss.2008.02.068
8. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: Revirt: Enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev. **36**(SI), 211–224 (2003). https://doi.org/10.1145/844128.844148
9. Fred Klassen: Tcpreplay-github. `https://github.com/appneta/tcpreplay`
10. Frei, A., Rennhard, M.: Histogram matrix: Log file visualization for anomaly detection. In: IEEE Conference on Availability, Reliability and Security (2008)
11. Garg, A., Mishra, D., Kulkarni, P.: Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments. In: ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (2017)
12. Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A.C., Varghese, G., Voelker, G.M., Vahdat, A.: Difference engine: Harnessing memory redundancy in virtual machines. Communications of the ACM **53**(10), 85–93 (Oct 2010)
13. Hardy, N.: The confused deputy: (or why capabilities might have been invented). SIGOPS OS Rev. **22**(4), 36–38 (Oct 1988)
14. Kohler, E.: Hotcrp software. `https://github.com/kohler/hotcrp`
15. Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., Huici, F.: My VM is Lighter (and Safer) Than Your Container. In: ACM Symposium on Operating Systems Principles (2017)
16. Mehta, A., Elnikety, E., Harvey, K., Garg, D., Druschel, P.: Qapla: Policy compliance for database-backed systems. In: USENIX Security (2017)
17. Neuman, B.C., Ts'o, T.: Kerberos: An authentication service for computer networks. IEEE Communications Magazine **32**(9), 33–38 (1994)
18. Nisi, D., Bianchi, A., Fratantonio, Y.: Exploring Syscall-Based semantics reconstruction of android applications. In: USENIX RAID Symposium (2019)
19. NIST: Cve-2009-2762. `https://nvd.nist.gov/vuln/detail/CVE-2009-2762`

20. NIST: Cve-2012-3578. `https://nvd.nist.gov/vuln/detail/CVE-2012-3578`
21. NIST: Cve-2019-9879. `https://nvd.nist.gov/vuln/detail/CVE-2019-9879`
22. NIST: Cve-2019-9880. `https://nvd.nist.gov/vuln/detail/CVE-2019-9880`
23. NIST: Cve-2020-13693. `https://nvd.nist.gov/vuln/detail/CVE-2020-13693`
24. NIST: Cve-2021-24182. `https://nvd.nist.gov/vuln/detail/CVE-2021-24182`
25. NIST: Cve-2021-24183. `https://nvd.nist.gov/vuln/detail/CVE-2021-24183`
26. Oexman, D.: Changeset for wp-gdpr-compliance. `https://plugins.trac.wordpress.org/changeset/1970366/wp-gdpr-compliance` (2018)
27. Parno, B., McCune, J.M., Wendlandt, D., Andersen, D.G., Perrig, A.: Clamp: Practical prevention of large-scale data leaks. In: IEEE Security and Privacy (2009)
28. PHP Devs.: OPcache. `https://www.php.net/manual/en/book.opcache.php`
29. PHP Devs.: Preloading manual. `https://https://www.php.net/manual/en/opcache.preloading.php`
30. Popov, N.: Extension exposing php 7 abstract syntax tree. `https://github.com/nikic/php-ast`
31. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All your iframes point to us. In: USENIX Security. pp. 1–15. USENIX, USA (2008)
32. Puentes, M.A.: PEGASUS: Powerful, Expressive, Graphical Analyzer for the Single-Use Server. Thesis, Worcester Polytechnic Institute (May 2021)
33. Raoufi, M., Deng, Q., Zhang, Y., Yang, J.: Pagecmp: Bandwidth efficient page deduplication through in-memory page comparison. In: IEEE Computer Society Annual Symposium on VLSI (2019)
34. Salah, T., Jamal Zemerly, M., Chan Yeob Yeun, Al-Qutayri, M., Al-Hammadi, Y.: The evolution of distributed systems towards microservices architecture. In: IEEE International Conference for Internet Technology and Secured Transactions (2016)
35. Stubbs, J., Moreira, W., Dooley, R.: Distributed systems of microservices using docker and serfnode. In: IEEE International Workshop on Science Gateways (2015)
36. Taylor, C.R.: Leveraging Software-Defined Networking and Virtualization for a One-to-One Client-Server Model. Master's thesis, WPI (2014)
37. W3Techs: Usage statistics and market share of WordPress. `https://w3techs.com/technologies/details/cm-wordpress` (2020)
38. W3Techs: Usage statistics of server-side programming languages for websites. `https://w3techs.com/technologies/overview/programming_language` (2020)
39. wordpress.org: Wordpress. `https://www.wordpress.org/` (2003)
40. Xia, N., Tian, C., Luo, Y., Liu, H., Wang, X.: UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling. In: USENIX Conference on File and Storage Technologies (2018)
41. You, L., Li, Y., Guo, F., Xu, Y., Chen, J., Yuan, L.: Leveraging array mapped tries in ksm for lightweight memory deduplication. In: IEEE NAS Conference (2019)