Single-Use Servers: A Generalized Design for Eliminating the Confused Deputy Problem in Networked Services

by

Julian P. Lanson

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

May 2020

APPROVED:

____

Professor Craig A. Shue, Major Thesis Advisor

____

Professor Lorenzo DeCarli, Thesis Reader

____

Professor Craig E. Wills, Head of Department

Abstract

Internet application servers are currently designed to maximize resource efficiency by servicing many thousands of users that may fall within disparate privilege classes. Pooling users into a shared execution context in this way enables adversaries not only to laterally propagate attacks against other clients, but also to use the application server as a "confused deputy" to gain escalated privileges against sensitive backend data. In this work, we present the Single-use Server (SuS) model, which detects and defeats these attacks by separating users into isolated, containerized application servers with tailored backend permissions. In this model, exploited servers no longer have unfettered access to the backend data or other users. We create a prototype implementation of the SuS model for the WordPress content management system and demonstrate our model's ability to neutralize real-world exploits against vulnerable WordPress versions. We find that the SuS model achieves a high level of security while minimizing the amount of code modification required for porting an application server. In our performance evaluation, we find that the CPU and latency overheads of the SuS model are very low, and memory consumption scales linearly. We generalize the SuS model to be applicable to a wide range of application server and backend resource pairs. With our modularized codebase, we port IMAP, a widely-used mail retrieval protocol, to the SuS model and find that doing so requires minimal effort.

Acknowledgements

First and foremost, I would like to thank Dr. Craig Shue for his guidance and mentorship not only during my graduate studies, but throughout my entire time at WPI. Having been my undergraduate professor, Scholarship For Service coordinator, MQP advisor and now thesis advisor, a great deal of my growth as a computer scientist and security researcher has occurred under his tutelage. For that alone I am exceedingly grateful; that I can consider him a friend is a bonus.

I would like to thank my fellow ALAS research group member Yunsen Lei for his contributions to the design and implementation of this work, and Dr. Timothy Wood of George Washington University for his useful comments and suggestions regarding the performance evaluation. I would also like to thank my thesis reader Dr. Lorenzo DeCarli for his excellent feedback on my work, and Beckley Schowalter for tackling my thesis drafts with her mastery of the English language. I also want to extend my thanks to the ALAS research group at large, for their constructive responses and questions during my research presentations.

I want to express my sincerest gratitude to my parents. This work is a culmination of years of their support and dedication; thanks to them I have always had everything I needed to succeed. I also want to thank my sister Cassandra for being a best friend, and my wonderful roommates Peter, Anna, Hannah, and Alicea, who are my family away from home.

Finally, I am very appreciative of the Scholarship For Service program for supporting me financially and enabling me to pursue a graduate degree. It is no stretch to say that without SFS, this thesis would likely not exist.

This material is based upon work supported by the National Science Foundation under Grants No. 1503742 and 1814402.

Contents

List of Figures

List of Tables

1 Introduction

In the modern Internet, the number of clients requesting remote access to resources is orders of magnitude larger than the number of application servers available to mediate that access. We therefore say that the Internet follows a "many-to-one" architecture. While efficient from a computing resource perspective, pooling users into a shared execution context in this way has significant security consequences for both the server's backend data and the clients themselves. Should a malicious user find a vulnerability in the application server, the user may compel the server to perform actions that go above his or her intended privilege set. This manipulation allows the user either to set up mechanisms to attack subsequent users (called lateral propagation) or directly read or modify important backend data. The latter case is an illustration of the so-called Confused Deputy problem, first coined by Hardy in 1988 [1].

The Confused Deputy problem arises implicitly from the Internet's many-to-one architecture, and it is inherently a problem of attribution. Each application server must be able to operate on behalf of users with a wide variety of permission levels, so it must have high privilege over the backend resource. From the perspective of the backend resource, all requests are thus made by a single highly-privileged entity who must be obeyed. There is no mechanism in place to attribute a particular request from the application server to a particular user. For example, it is impossible to tell whether a request to delete all data was initiated by a legitimate administrator or by a malicious user who has escalated their privileges by exploiting the application server (see Figure 1).

Our approach to eliminating both the Confused Deputy problem and lateral propagation is simple. Users are isolated into ephemeral, private execution contexts (SuS containers) with a copy of the application server that has privileges against the backend resource tailored to match the intended privilege set of the user against the application server. Even if a malicious user is able to induce his or her application server to make an unauthorized request against the backend, the server itself will not have sufficient privileges to execute the request, and execution will be denied. This eliminates the Confused Deputy problem. Furthermore, malicious changes made by the user to files within his or her execution context do not propagate to other users' execution contexts, and they are permanently lost as the execution context is torn down after the user's session ends. Hereafter we refer to the private execution contexts as Single-Use Server (SuS) containers, and to the overall system design as the SuS model.

The SuS model has many security benefits. By tailoring permissions at the backend, it is simple to apply a desired high-level security policy by deciding what type of actions should be permitted to a particular class of user (e.g. "Users with role X should be able to perform Y action against Z resource"). Importantly, this eliminates the need to rely on the application server's codebase to implement the security policy correctly. In other words, the SuS model creates a vulnerability-tolerant environment around the application server that detects when the actual access control system does not match the desired one. Furthermore, since there is a one-to-one relationship between users and application servers, it is simple to attribute a request denied by the backend to a particular offending user. The SuS model can then freeze the adversary's SuS container to prevent continued operation. Then, via forensic analysis of the SuS container and associated network traces, security analysts should be able to discover the vulnerability in the application server, and how the adversary exploited it, without the adversary ever benefiting from revealing this information.

Other works have previously employed a design based on user containerization as a method for attack attribution and preventing the Confused-Deputy problem (see Section 2.2). We improve upon these works in the following ways:

- **Generalization**: Other works have limited the scope of their defense to LAMP-architecture webservers, where website and webserver code function as the application server and a MySQL database functions as the backend resource. Although many networked services share a similar server/backend architecture that could be protected by a user-isolation approach, the protocols, authentication models, and backend access control systems that the services employ are often

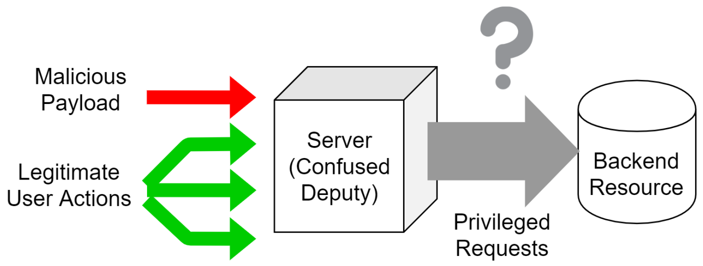


Figure 1: Illustration of the Confused Deputy Problem. In this scenario, the application server executes all requests on users' behalf with the same high level of privilege, regardless of what the security policy says a particular user can and cannot do. It is therefore impossible for the backend resource to distinguish between requests made by legitimate users and requests resulting from exploitation of the application server. This renders the server a "Confused Deputy."

very different. We implement a system that is modular enough to be applied to a variety of application servers, and we demonstrate this attribute by applying it to the IMAP mail retrieval protocol.

- **Exploit Detection and Response**: Our system includes mechanisms for detecting privilege escalation (based on responses from the backend resource) and freezing offending containers.
- **System Evaluation**: We perform a practical evaluation of security and performance. Other papers' security evaluations are purely hypothetical, whereas we research and apply real-world exploits that fit our threat model to demonstrate the SuS model defending against them. Similarly, our scalability evaluation focuses on end-user experience instead of using micro-benchmarks and counting number of operations per second.

The rest of this report is organized as follows. In Section 2, we provide necessary background and more closely examine the aforementioned related works. In Section 3, we develop a prototype implementation of the SuS model that has a similar design to that of previous work. We port a simple blog on the popular web deployment platform WordPress to the SuS model and discuss the engineering required to do so. In the first half of Section 4, we research and compile a set of exploits against the WordPress blog from the Common Vulnerability and Exposures (CVE) database that fit our threat model. We apply each of these exploits to both a control server and the SuS-protected server and demonstrate that SuS is effective at neutralizing the attacks. In the second half of this section, we evaluate the performance and scalability of the ported WordPress blog. In Section 5, we abstract out the application-specific aspects of our prototype and modularize the system to make minimal assumptions about the type of backend resource, the type of access control, and how requests are made by the application server to the backend. We demonstrate our generalized system by applying it to the IMAP mail retrieval protocol and show that it can be ported to the SuS model with little effort. Finally, we conclude and discuss future work in Section 6.

2 Background and Related Work

In this section, we provide some background on virtualization and containerization, how they operate, and how their security models differ. We discuss privilege escalation attacks and their relation to the Confused Deputy problem, and then elaborate on works that attempt to thwart these attacks.

2.1 Virtualization and Containerization

Virtualization and containerization are two strategies for enabling process isolation, resource multiplexing and scalable deployment. A virtual machine, or VM, is a complete computing environment that is emulated on virtual hardware inside another host (the applications that manage VMs and perform the hardware emulation are known as hypervisors). This hardware-level virtualization means that the VM's OS, filesystem, processes, users, etc. are all completely distinct from the host's. Because of this, VMs have a "sand-boxing" effect on the operations performed inside of them. If an external adversary compromises the VM, a user can simply delete it and install a fresh one without ever having exposed sensitive data to the attacker.

Containers, by contrast, are groups of processes running within a particular directory on the host's filesystem that have been configured to have a limited view of the rest of the machine. Whereas processes in one VM run on a completely separate kernel from those of another, containerization offers a simpler, logical process separation in which containerized processes run on the same OS. For this reason, containers offer better performance over VMs at the cost of a larger trusted computing base (TCB). As we will explore in Section 2.2.3, the use of per-client VMs caused some older works in privilege-escalation prevention to suffer from scalability concerns. More recent works, particularly Radiatus [2], use containers as a lightweight alternative.

The fundamental mechanisms that enable containerization have been around for decades; however, tools for container management have only recently become mature enough for wide-scale use. With this maturation, the use of containers has grown significantly. Here we will explain how containers are built and deployed in the popular container management toolset Docker [3].

Accomplishing container-based process isolation requires a few different features of the Linux kernel [4]. First, a command such as `chroot` or `pivot_root` changes a particular process's root directory to a specificed subdirectory of the host's root directory. So long as the `chroot`ed process does not gain root privileges, any files outside of this subdirectory become invisible and inaccessible to it. For restricting the process's view of non-file resources (user lists, network devices, process lists, message queues, etc.), new namespaces are created for the process and applied. Finally, adding the process, or set of processes, to a `cgroup` enables control over their usage of computational resources (CPU time, memory, etc.). These three kernel features work in concert to produce the basic concept of a container.

The usefulness of containers would be limited without a robust container management system, such as Docker. Docker introduces features that abstract away all of the low-level work from the developer and simplify building, deploying, modifying, and performing version-control for containers. Deploying a container starts with a container image, which is a particular state of a container filesystem that is used as a blueprint. To build a container image, developers write a Dockerfile, which specifies a base image (often that of a particular OS and version) and then a set of commands to apply to that image (such as copying in files, installing software, etc.). By executing the Dockerfile, the commands are applied and the resulting filesystem is saved with a user-supplied tag. For instance, someone deploying a webserver might use an Ubuntu 18.04 base image, install `apache2`, copy in a directory containing the desired web content, and change some file permissions. The resulting image is that of a webserver ready to be started.

Docker uses the UnionFS filesystem service to represent modifications to a container's filesystem as a set of layers. Thus, every instruction in a Dockerfile is represented as a layer on top of a base image. The final layer added by the Dockerfile becomes the base layer for the new image. Then, an arbitrary number of containers can be spawned from this image to run concurrently, making impermanent edits (also represented as layers) to their version of the image filesystem.

2.1.1 System Interaction

Containers are designed for process isolation, but for a variety of use cases, interaction with the wider system is necessary. Docker containers can be designated to have access to certain directories in the host filesystem outside of their `chroot` jail. The older technique for providing external directory access is called a bind mount, in which a directory on the host is directly mounted onto the container's filesystem. For security, the mounted directory can be set as read-only. Docker's newer, preferred method for giving access to a directory uses Docker volumes [5]. When a directory is shared with a container using a volume, that directory is first copied into Docker's storage directory. The new storage sub-directory is the one shared with the container. Since this content is managed by Docker, volumes allow for safer sharing between containers, encryption, backups, and a variety of other features.

Since containers lend themselves to running application servers, Docker also provides robust networking tools. Deployers can create their own subnets and decide what IP address to assign to their containers, or let Docker assign the address itself on a default subnet. For security, containers on different subnets cannot communicate by default. Docker also provides an option for container ports to be directly mapped to host ports at container start time; in this case a small `docker-proxy` process is spawned for each port to handle the packet forwarding.

2.2 Privilege Escalation Defense

In an attack against a system, privilege escalation refers simply to the unauthorized and unintended increase in an adversary's privilege level. It is sometimes unclear when a Confused Deputy is at play in a privilege escalation; here we attempt to clarify. Hardy, who first coined the phrase "Confused Deputy" in a 1988 paper, offers only the following: "the [program] serves two masters and carries some authority from each to perform its duties" [1]. While not a complete definition, this excerpt makes the important distinction that the entity being exploited must serve more than one user with disparate privilege sets. To be considered a Confused Deputy, we submit that an entity must satisfy the following criteria:

1. The entity must mediate access to a backend resource which is clearly distinct from itself.
2. The entity must serve multiple users from more than one privilege class. This will necessitate that the entity's privilege over the backend be a superset of all user privileges.

We find that these criteria are supported by the definition of a Confused Deputy provided by the Common Weakness Enumeration (CWE) database [6].

A Confused-Deputy-based privilege escalation attack must make use of the above two properties to exploit or coerce the privileged entity into making valid but unintended requests to the backend. It is important that the attack targets a vulnerability in the entity; using an application server to exploit a kernel-level vulnerability and gain privileges does not qualify as a Confused-Deputy-based attack. In this case, the OS is both the entity being exploited and the backend resource, thus the first criterion is not met. A strong example of an OS-level Confused-Deputy-based attack is the BluePill rootkit [7] that exploits the OS to insert an ultra-thin hypervisor between itself and the hardware, allowing the attacker to monitor input and control the OS's operation invisibly. Here, the OS (which mediates access to hardware) is used as a deputy to install the hypervisor beneath itself.

In the literature review, we will distinguish between three general threat models. In the first model, defenses remove vulnerabilities from the application server's code, but ultimately trust both the OS and the modified application. Defenses in the second threat model use virtualization, trusting neither the application code nor the OS. In the third model, the OS is trusted to aid in the defense against an untrusted application server.

2.2.1 Work Treating Both the Application and OS as Trusted

In the first threat model, both the application and OS are part of the TCB. Although the application server code may be instrumented, hardened or analyzed for vulnerabilities, once the server's execution is started its code is trusted. In their survey on server-side defenses against application-level privilege escalation attacks, Li and Xue outlined three strategic paradigms [8]; all works in the first two paradigms, and some in the third, fall into this threat model.

The first paradigm involves developing secure server programs. Provos et al. describe a program structure called "privilege separation," in which the unprivileged and privileged pieces of a program are split into two processes [9]. The unprivileged process must request the privileged process to perform actions on its behalf, such as opening files, and is given access to those files by passing file descriptors. Since web applications cannot be easily rewritten to follow this structure, other works present special web frameworks that are designed to enforce security policies and track information flow. Some of these works require code to be written in security-typed languages [10, 11], while others check assertions and follow the flow of data from within an existing, unhardened language's runtime [12]. Unlike our work, most works in this field can only be applied to existing applications through non-trivial rewrites.

The second paradigm uses offline analysis of the server code base to detect vulnerabilities in advance. Sun et al. [13] build a web application sitemap and, leveraging implicit access control assumptions written into the code (e.g., performing a redirection if privilege levels are too low), find scenarios in which an unprivileged user can directly access a privileged PHP page. MACE is a tool that takes a slightly different approach, finding inconsistencies in the "authentication context" around each SQL query in the PHP code [14].

Li and Xue's third paradigm involves using a run-time protection scheme to catch exploit attempts and neutralize them. "Taint-tracking" is a traditional approach in this paradigm that involves tagging untrusted user input as it arrives at the system and preventing unsanitized inputs from being used in sensitive operations. Examples include work by Su and Wasserrmann [15] and Chin and Wagner [16]. This approach involves instrumenting either the application code itself, or the underlying runtime; therefore, the threat model must still trust these components. When implemented correctly, this approach is successful at preventing attacks based on poor input sanitation and validation. However, it does not prevent lateral propagation since any malicious changes to the application server or host system can still affect subsequent users.

2.2.2 Work Treating Both the Application and OS as Untrusted

Work in this threat model aims to give security guarantees regardless of whether the application server code and underlying OS are free of vulnerabilities. To achieve such guarantees often requires virtualization. For instance, in NICKLE, Riley et al. use a virtual memory manager to maintain a shadow copy of kernel-level pages [17]; by comparing loaded memory to the second copy via hash values, memory integrity can be guaranteed. Works such as SVA and KCoFI require porting the OS to a new language in exchange for control flow integrity and memory safety guarantees [18, 19]. SecVisor makes use of modern processors' ability to virtualize hardware to insert itself as a hypervisor between the OS and the hardware (the same technique used by BluePill) [20]. It then introduces an extra layer of translation between "guest" and "real" physical memory, allowing it to add extra memory security features.

As is common in security research, the more security offered by these systems, the more performance is impacted. Works like SVA do not have an excessive runtime overhead, but require significant upfront work in porting the OS. A more detailed description and analysis of these works and others can be found in a survey paper by Brookes and Taylor [21].

Client isolation is a different strategy with this threat model. Falling into Li and Xue's third paradigm, this strategy solves both privilege escalation and lateral propagation. Parno et al. were the first to introduce work in this area with CLAMP, a system for protecting LAMP stack websites [22]. In CLAMP, individual users are assigned to isolated VMs each running a copy of the web server code, and a MySQL proxy restricts each context's access to specific parts of the database. Upon logging into a separate, trusted authentication portal, a user can upgrade their VM's privilege level with fine granularity. A query restriction mechanism ensures that even if a VM is compromised, it does not have unfettered access to the database and its sensitive data.

Taylor [23] also focuses on user isolation, and introduces a software-defined networking (SDN) controller to perform the client traffic demultiplexing. The Taylor work lacks the resource restriction component present in CLAMP, but adds attack attribution. Furthermore, Taylor's SDN-based solution is not inherently tied to a particular protocol. Taylor finds that VMs are too heavy-weight to implement client isolation on a production level.

Since CLAMP and Taylor put users in VMs, neither the application server nor the OS on which it runs is trusted. However, unlike the previously described works, the hardware virtualization itself is not a key part of the defense; it is used as a technique for enabling user isolation. As we will see in Section 2.2.3, other works operate in the same field using containers as a light-weight alternative.

2.2.3 Work Treating Only the Application as Untrusted

This threat model assumes that vulnerabilities will exist in the application code, and a runtime environment that tolerates them is built around the server's execution context. The OS is involved as part of the defense and so remains part of the TCB. Common techniques for detecting and preventing low-level privilege escalation attacks include stack canaries, address space layout randomization, and non-executable stack pages. They fall into this category since an exploitation of the OS could sabotage their effectiveness. More recent work examines how to make control flow hijacking more difficult by inserting extra instructions into program code at compile time [24, 25], or how to defeat ROP attacks by removing instructions with the return opcode embedded in them [24]. Brookes and Taylor compare the security of these works and more in their survey paper [21].

The SuS model that we present is a logical progression from the CLAMP and Taylor works described in Section 2.2.2, combining tools for robust resource restriction and attack attribution with a design that is focused on general networked services. Because our defense uses containers instead of VMs to address the performance and scalability issues in these prior works, the OS must be trusted and our threat model differs from theirs.

Another work in user isolation that uses containers is Radiatus by Cheng et al. [2], which introduces even more stringent security measures than SuS or CLAMP. But whereas Radiatus places restrictions on the design of the program it defends, requiring significant code restructuring, our SuS model aims to provide reasonable security with minimal codebase modification. Furthermore, Radiatus is explicitly designed for database backends, while we endeavor to produce a protocol-agnostic design.

At the OS level, QubesOS is the tool that most closely matches the containerization approach [26]. QubesOS is designed to run all processes (in isolation or in groups) inside dedicated VMs; this way, exploiting any one application does not give immediate access to all other components of the system.

3 Application-Specific Implementation

This section describes the intial design and implementation of the SuS model. We begin by discussing our threat model, then the high-level design and function of each component of the SuS model. We then examine each component in detail and describe our experience implementing them for WordPress, a popular website deployment and content management tool [27]. WordPress is estimated to be used in over 35% of all websites on the Internet [28]; its widespread use and long list of known vulnerabilities make it an ideal candidate for porting to the SuS model.

3.1 Threat Model

The SuS model is a server-side application-level defense system aimed at 1) preventing adversaries from successfully executing any backend request above their privilege level, as defined by the server's intended security policy and 2) preventing adversaries from making changes to files on the server that would enable them to attack other users. To demonstrate an attack against which the SuS

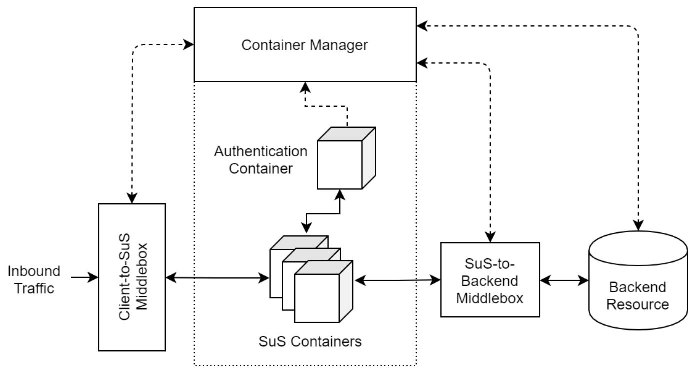


Figure 2: Design overview of the SuS model. The four trusted entities (Client-to-SuS Middlebox, Container Manager, Authentication Container, SuS-to-Backend Middlebox) coordinate to provision SuS containers and assign them to clients, provide a means of upgrading privileges to the backend resource, and enable the neutralization and analysis of exploited application servers running in SuS containers.

model is ineffective, we propose a scenario in which a user has permission to create a new row in a SQL table and attempts to populate that row with a string (such as JavaScript code) that acts maliciously when fetched by another user. Since no permissions have been violated, the SuS model is incapable of preventing such an attack; defense falls to proper application-level input sanitation. We also do not consider attacks which have the goal of disrupting the availability of the server application, such as DoS (denial-of-service) attacks, and simply expect that defenders will employ current best practices. Experiments show the SuS model's running performance is roughly equivalent to the control, but container start-up costs could be a DoS concern that would need to be addressed in the deployment plan.

We assume that the application server being encapsulated in a SuS container facilitates user access to a backend resource where data is stored, and we also assume that one or more of these containerized servers may be exploited by adversaries. We assume that adversaries can only access the application server's host machine via network communication and will attempt application-level exploits via the payload of one or more packets in the application server's communication protocol. Because we use containers instead of VMs, the OS is part of our trusted computing base, and we cannot prevent attacks against it such as buffer overflows and return-oriented programming attacks. Similarly, attacks against the hardware and SuS infrastructure are out of scope. Importantly, the SuS model is incapable of detecting attacks above the application-level, such as social engineering or CSRF, in which a privileged user is coerced into misusing his or her privileges.

3.2 High-level Design

The basis of the SuS design is the notion that the Confused Deputy problem could be solved if each user operated within an application server that only had enough privileges over the backend resource to perform any action the user is intended to be able to perform. This way the server would be simply unable to execute any commands above the user's privilege level, even if an adversary exploited a vulnerability in the server code. An overview of the SuS model design is shown in Figure 2.

To assign users to unique copies of the application server, we need a virtualization and process-isolation technology. We choose to use Docker containers because of their performance benefits over VMs and mature set of API-based utilities. Once we have a Docker image of the webserver that SuS containers can be spawned from, we need a manager process to handle provisioning, starting and destroying containers, and maintaining their state. Each container needs a unique set of credentials against the backend database so that container privileges can be altered independently of one another. The manager process that installs and removes credentials with each container, and upgrades container privileges when prompted, is called the Container Manager.

To demultiplex user requests and forward each one to an appropriate SuS container, we need a front-end proxy, which we call the Client-to-SuS middlebox (CSM). When a new user makes a request, the CSM must retrieve a valid IP address for an unused container from the Container Manager and store a mapping of that user to the retrieved IP. All subsequent traffic from that user must be directed to the mapped container.

At some point during the session, a user may wish to authenticate to upgrade his or her privilege level. Since the code within the SuS container is untrusted, we require an external portal to which clients can be redirected to perform authentication. This portal is trusted to report the requisite information for increasing privilege level to the Container Manager when a login occurs. We refer to this entity as the Authentication Server.

The Container Manager controls privilege levels by changing the permissions of each container's database credentials over the tables in the database. As Parno et al. note in CLAMP, there are scenarios in which the backend resource's native access control mechanisms are insufficient to implement a desired security policy. For instance, it may be desired to allow each authenticated user to perform an UPDATE on their own row within the `wp_users` table, but not any others. MySQL's grant system allows privileges to be granted only on a per-table basis, leaving room for adversaries to make use of escalated privileges. Thus, we need a middlebox, called the SuS-to-Backend middlebox (SBM), to sit between the SuS containers and backend and to "scope" queries to the appropriate rows for the user's privilege level.

Under this privilege control model, a container should never be denied a request against the backend due to insufficient privileges unless it has been exploited to perform an unauthorized operation. We can therefore easily detect exploited containers by monitoring query responses for such denied requests. Since the SBM is already proxying backend traffic, we can simply incorporate this mechanism into it. Once the SBM finds indication of a misbehaving container, it can notify the Container Manager to freeze the container and have the CSM block subsequent traffic to that container.

3.3 Container Manager

The Container Manager has two main responsibilities. First, it must handle all of the startup and destruction of containers, and keep track which ones are running. Second, it must listen for notifications or requests from the other trusted components of the SuS model and respond to them by performing actions.

We do not want the act of spawning a container to delay the Container Manager's ability to respond to a message from another SuS model component. Thus, we split the Container Manager into two main threads. The ContainerPoolWatcher thread is the smaller of the two and has exclusive control over the creation of containers. While the pool of available containers is less than the maximum size, it will continually generate new containers and add their information to the `live_containers` dictionary and `unassigned_containers` list, both of which are mutex-protected. When creating a container, the ContainerPoolWatcher retrieves a database username and IP address from objects that manage these values to ensure no duplicates. The IP belongs to the address range of a Docker network we created, thus allowing us to control the address each container is assigned. A database password is randomly generated, and the database credentials are installed in the MySQL database with appropriate permissions for an unauthenticated client. Then the `docker run` command is executed to create and start the container.

The main program thread creates the ContainerPoolWatcher thread, waits for the pool of ready containers to reach max capacity, and then creates sockets for communicating with the CSM, Authentication Server, and the SBM. Using the `select()` function, the main thread sleeps until one or more of these sockets is ready for processing. Later, we discuss each message type and response action with their appropriate SuS component.

When the Container Manager is terminated via SIGINT, we close gracefully by registering a function with `signal()`. We notify the ContainerPoolWatcher to stop spawning containers via a `Condition` variable, and send a message to a fourth socket, that the main thread `select`s on, to interrupt its sleep. The sockets are closed, and the main thread iterates through `live_containers` to despawn each container and remove its database credentials.

3.3.1 Container Configuration

For our SuS container implementation, we run WordPress on Lighttpd (pronounced "lighty"), a lightweight, single-threaded webserver [29]. We write a Dockerfile to build a image of a containerized webserver that is appropriate for use in the SuS model. The Dockerfile handles installing Lighttpd and PHP7 on Alpine Linux. It then copies the WordPress code directory and Lighttpd configuration files into their proper locations within the container directory structure, and changes the permissions of the files to belong to the `lighttpd` process. Port 80 is also exposed so that the container can receive traffic forwarded by the CSM.

Before starting the webserver software, we need to set some unique per-container configurations (namely, database credentials). We therefore instruct the Dockerfile to execute a `start.sh` script on container startup instead of starting the Lighttpd server directly. When calling the `docker run` command, any additional arguments passed are forwarded to the `start.sh` script, which then uses them to overwrite temporary values for the database credentials in `wp-config.php`. After this, the script finally executes the `lighttpd` webserver process.

3.4 Authentication

When a user logs in, the Container Manager must be notified so that it may record his or her role and upgrade the container's backend permissions. However, since the code running within each SuS container is outside of the SuS model's trusted computing base, a SuS container must not directly perform its own user's authentication or report the user's identity and role to the Container Manager. Authentication in the SuS model therefore requires that the containerized software supports a pluggable authentication module (PAM) mechanism so that login decision-making can be performed remotely by the Authentication Server. As of version 5.1.1, WordPress does not natively support such a mechanism nor were we able to locate a satisfactory plugin, which left us the task of building a mechanism ourselves.

This authentication mechanism involves two messages. The first is a redirect from the SuS container to the Authentication Server's login portal; this must also establish the identity of the SuS container so the Container Manager can eventually be notified of the permission set to upgrade. We select the database username as such an identifier, and we encode it as a query parameter in the redirect URL. However, to prevent spoofing, we need a system by which an untrusted entity can prove it is passing the correct identifier. Our solution (shown in Figure 3) is as follows.

The Container Manager and the Authentication Container are configured with a symmetric key. When each SuS container is created, the Container Manager constructs a random salt value in addition to the database credentials. Using the salt and the SuS container's database username, the Container Manager creates an HMAC output that will be used as the container key. The Container Manager passes both the salt and the container key into the SuS container. When the SuS container redirects the client to the Authentication Container, the SuS container includes its database username, the salt, and an HMAC covering the request as a token. Since the Authentication Container has the container's database username, salt, and the key it shares with the Container Manager, it can reconstruct the appropriate container key. It can then use the supplied message and container key to generate the same HMAC output provided by the SuS container as a token. If the outputs match, the Authentication Container knows the SuS container's information is valid.

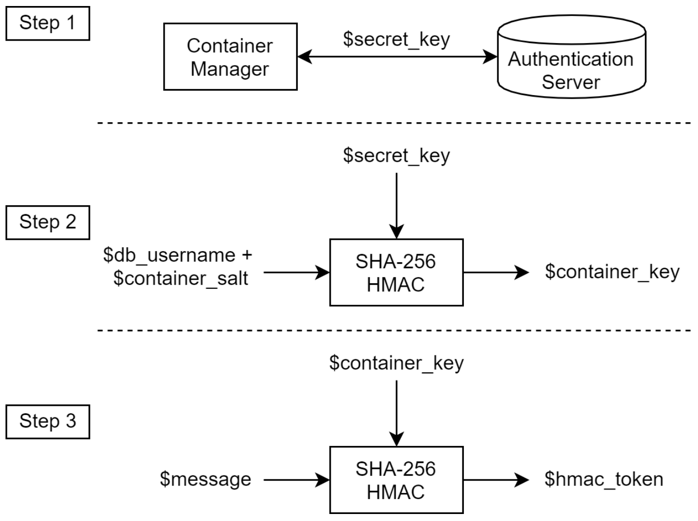


Figure 3: The cryptographic scheme used to validate data sent between the Authentication Container and SuS containers. This scheme enables the Authentication Container to validate messages from SuS containers without ever directly sharing keys.

If a malicious agent attempts to supply a different database username, the Authentication Manager would generate an incorrect container key and the resulting tokens would not match.

The second message is a redirect from the Authentication Server to the SuS container that indicates to the SuS container that a successful remote login has occurred. The redirect contains data such as the appropriate post-login destination URL, the user's site ID, and a cryptographic token (e.g., `[SITE_IP]/auth-redirect.php?data=redirect_url;user_id;salt&token=[F]`). The HMAC token for the second message is signed by the Authentication Server with the container's key.

Enabling this authentication scheme is the only part of incorporating WordPress into the SuS model that requires modification of the WordPress codebase. We implement the authentication as follows. First, we add two variable declarations to `wp-config-base.php`, namely `HMAC_KEY` and `HMAC_SALT`. We then modify `start.sh` to accept additional command line arguments and write them as the definitions of the aforementioned variables using `sed`.

To perform the first redirect, we change the `wp_login_url()` function in `wp-includes/general-template.php` to point to the address and port of the authentication server. This way, all "Login" buttons on the site automatically point to the Authentication Server. This function also calculates the HMAC token and appends the requisite query parameters to the redirect URL.

We create the Authentication Server as a bare-bones PHP login page running on Lighttpd. The login form includes hidden values retrieved from the redirect message's query parameters, so they can be passed to the POST functionality after the login submission button is clicked. Once the Authentication Server verifies the HMAC token is valid, it checks the credentials entered by the user against the `wp_users` table of the WordPress database. For this, we simply copied the password hashing PHP script over from the WordPress codebase. Then, the Authentication Server notifies the Container Manager, via socket, of the database username of the container that just logged in and what role the container now has. Finally, the Authentication Server constructs the second redirect URL and redirects the user.

When the user is redirected to their original container, we need to convince WordPress to treat an unknown user (that it has not seen log in) as if they had logged in as the user with ID X. So, before the user can reach the desired destination URL (typically a role-specific console), we need to stop at a waypoint page on which cookies can be set for the user. This way, when the user finally reaches his or her desired destination page, the browser will provide cookies that the SuS container recognizes. We accomplish this as follows. First, we create a new script in the WordPress codebase called `auth-redirect.php` and configure the Authentication Server to point to that resource when performing the second redirect. In `wp-includes/functions.php`, we register a callback `set_auth_redirect_cookies` to fire at the `after_setup_theme` stage of the bootstrapping process with the `add_action()` WordPress function. In this function, we check if the current page is `auth-redirect.php`. If so, we extract the data from the query parameters, verify the HMAC, and then manually call the `wp_set_auth_cookie()` function using the passed data. Finally, when the `auth-redirect.php` code executes, it retrieves the desired destination URL and redirects the user to that address. In the 302 HTTP redirect response returned by `auth-redirect.php`, the cookies set in our callback function are delivered to the user, ready for use in the following request.

3.5 Client-to-SuS Middlebox

The CSM is a multi-threaded HTTPS-to-HTTP proxy written in C++ that performs additional per-packet processing. The design and implementation of the CSM was a collaboration with another graduate student, Yunsen Lei. The main thread of the proxy creates a server socket and listens for new connections. SSL objects for these connections are passed via `epoll` to a dispatcher thread which spawns a proxy worker thread for each one. We use the lightweight library axTLS to handle SSL connections.

In order to demultiplex user connections, the CSM needs a unique identifier from the user that can mapped to a SuS container IP address. Because each user's browser may maintain multiple concurrent SSL connections to a website, we cannot use an SSL ID for this task. Instead, we introduce the concept of a `SUS_DEMULT_COOKIE`, which users must supply when requesting resources over a new SSL connection. The process of client demultiplexing does not concern the SuS containers themselves, so we do not set the demultiplex cookie inside of them; nor are they aware of the cookies' existence. Instead, the cookie field is silently inserted into HTTP responses and removed from HTTP requests by the proxy threads of the CSM.

When a user makes a request for the first time, he or she will not have a `DEMULT_COOKIE`; this causes the CSM to request a new container IP from the Container Manager. Upon receiving the Container Manager's response, the worker thread generates a new `DEMULT_COOKIE` and maps it to the container's IP. The HTTP request is then forwarded to the SuS container; once a response is received, the worker thread inserts the cookie in the response header before forwarding it on to the client. Subsequent proxy threads handling the same user need only extract the `DEMULT_-COOKIE` from the request header and retrieve the corresponding IP address from the map object. As per the HTTP specification, proxies must insert additional headers (e.g. `X-Forwarded-Proto`) for each intercepted request, so extracting the demultiplex cookie from each request does not introduce additional overhead.

Since the CSM is already positioned between the clients and SuS containers, it also has the responsibility of detecting periods of inactivity and notifying the SuS Manager to reap containers. Thus, the CSM's `COOKIE_MAP_ENTRY` dictionary also maps demultiplex cookies to the last timestamp when a request was forwarded to that container. This field is updated by proxy threads each time a request is received. A separate reaper thread sleeps for a configurable period of time and, on wakeup, iterates over the dictionary to find containers that have not been active since the last wakeup. It then sends the IP addresses of these containers to the Container Manager.

Eventually, a container may be frozen by the Container Manager (see Section 3.6). At this point, the Container Manager also notifies the CSM to block all subsequent traffic to the frozen container. The CSM maintains a thread that listens for such messages from the Container Manager and adds passed IPs to a drop list. Before forwarding an HTTP request to a SuS container, each proxy thread consults the drop list. If the destination IP is on the drop list, the proxy thread closes the client's SSL connection and terminates.

3.6 SuS-to-Backend Middlebox

As a foundation for the SBM, we write a simple multi-threaded TCP proxy in C++ to sit between the SuS containers and the MySQL server. The SBM is responsible for intercepting queries and appending additional WHERE clauses as needed to implement a particular row-level security policy. For instance, we may desire that a non-admin user have UPDATE privileges over the row in `wp_users` that matches his or her user ID, but not on any other row.

To allow for flexible configuration, we introduce the concept of a *ResourceRestrictTable*, a 3-dimensional array that maps the *(role, resource, query type)* for each MySQL query to a particular WHERE clause, called an access predicate, that must be appended to the query. This table is represented as a series of `.csv` files (one for each role) that are loaded into the program at startup. The middlebox also maintains a UserContext dictionary that maps IP addresses to user information (e.g., user id, role). For each database query, the SBM extracts the resource (table), and access type (e.g., SELECT, INSERT). It also retrieves the container user's role from the UserContext dictionary, or uses a default role if the user has not logged in previously. Using these values, the SBM retrieves the access predicate. An access predicate may have a variable; for example, "ID = :user_id." In this case, the SBM inserts the corresponding value from the UserContext dictionary entry before appending it to the query. Once an access predicate has been added, query header information is updated to reflect the new message length. In this way, we silently restrict the type of data to which each user has access, without their knowledge.

To support this functionality, the SBM must be notified of a SuS container's role and user ID when a user logs in so the proper access predicates are applied for the user. Thus, once the Authentication Server notifies the Container Manager that a user has logged in, the Container Manager forwards some of this information to the SBM. The SBM has a thread that blocks on a socket read from the Container Manager, waiting for new information. When a message arrives, the thread parses it and adds an entry to the UserContext dictionary. Similarly, the Container Manager notifies the SBM when a container has been despawned, so the entry may be removed.

We note that instead of forwarding modified requests, it would be possible for the SBM to simply monitor for requests that attempt to access improper table rows. It would consider such requests as indications of compromise. However, this design significantly increases the complexity of the SBM, as the middlebox must understand both the rows each user is authorized to access and the meaning of existing WHERE clauses in the request. With our design, the SBM remains oblivious to this information and simply applies additional WHERE clauses to requests that match its criteria.

A function of the backend middlebox that is not present in CLAMP is the monitoring of query responses for error messages that indicate a compromised SuS container and notifying the Container Manager to freeze the offending instance. We accomplish this by searching responses for the 1142 MySQL error code, which occurs when a client makes a request for which they had insufficient privileges. Upon detecting such an error, the SBM messages the Container Manager to freeze the offending container.

3.7 Extended Sessions

Although not integral to the SuS model, we also introduce the concept of an extended session. In HTTP, cookies with longer expiration times are often used to let users return to websites after periods of inactivity without re-authenticating. With the current SuS model, a user's SuS container is reaped after a sufficient period of inactivity, forcing re-authentication upon his or her return.

We introduce the concept of a `SUS_SESSION_ID` to track users' authentication status and prevent containers that time-out from being reaped. At container startup, the Container Manager's `ContainerPoolWatcher` thread generates a random string for the new container's session ID and includes it as an extra argument to the `docker run` command. The `start.sh` and `wp-config.php` scripts are modified to include this value as a new variable in WordPress's configuration. The `set_-auth_redirect_cookies` callback is modified to manually set an additional cookie with a value equal

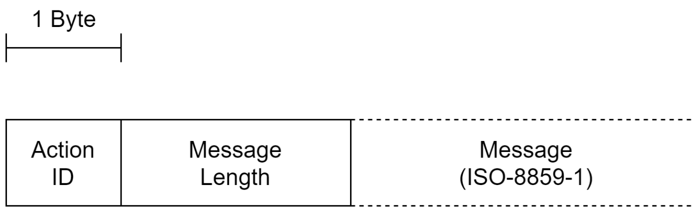


Figure 4: Message format used by SuS Model components.

to the session ID, which matches the expiration time of WordPress's authentication cookies. When the Container Manager is notified by the Authentication Server that a user has logged in, it sets some authentication context in the container's entry of `live_containers`. Eventually, the SuS container times out due to inactivity, and the Container Manager is notified to reap it. Since authentication context has been set, the Container Manager does not despawn the container but simply halts its execution using `docker pause`.

When a user makes a request to the CSM, they may or may not have a valid `DEMULT_COOKIE` value. If the cookie is not valid, the CSM searches the header for a `SUS_SESSION_ID` cookie. If one is found, it is included in the body of the message the CSM sends to the Container Manager requesting a container IP. The Container Manager may use this value to search `live_containers` for a match. If one is found, the Container Manager returns that SuS container's IP address and restarts the container using `docker start`. If no match is found, it simply returns the next IP address in the `unassigned_containers` list.

Our experience shows that a paused webserver container utilizes an insignificant portion of CPU resources, making this approach feasible. In IPv4, leaving a container running long-term per-user eventually leads to address exhaustion. By transitioning to IPv6, this ceases to be a concern, since there are a significant number of available addresses. Furthermore, extended sessions cannot be used to enable DoS attacks because only authenticated users are eligible to use their functionality. By ensuring that each client can only have one dormant container at a time, to be able to generate a large enough volume of stopped containers to impact performance, an adversary would need to have already compromised credentials for a large portion of the service's user-base.

3.8 Communication Protocol

To simplify message parsing and help polish the SuS model's distinct components into a cohesive tool, we standardize the format of messages in the SuS model as shown in Figure 4. This effort was a collaboration with Yunsen Lei.

The message header consists of 3 bytes. The first byte contains the message's action ID. The second two bytes of the header indicate the length of the subsequent message bytes. Some messages do not require a body, such as a container IP request from the CSM that does not contain a session ID. See Table 1 for a list of action types and corresponding message bodies. For a production-level tool, the communication protocol would be changed to support message acknowledgement (such as using TCP instead of UDP sockets) and authenticity checks.

3.9 Discussion

The SuS Model seeks to maximize security while minimizing modifications to existing code. We only made functional changes to the existing code for a PAM mechanism and session ID support. Table 2 details the number of lines changed to each WordPress file to enable PAM functionality.

Action ID	Action Type	Recipient	Message Content
0x01	Get Container IP	Container Manager	Session ID or none.
		CSM	Container IP address
0x02	Reap Container	Container Manager	Container IP address
0x03	Freeze/Block Container	Container Manager	Container IP address
		CSM	
0x04	Login Update	Container Manager	Container IP address, user ID, role.
		SBM	

Table 1: The action ID and corresponding contents for socket-based communication between SuS components.

File Name	Lines Added	Lines Modified
`wp-config.php`	3	2
`auth-redirect.php`	8	0
`general-template.php`	3	1
`functions.php`	14	0
Total	28	3

Table 2: The number of lines added and modified in the WordPress codebase to incorporate it into the SuS model

4 Evaluation

Here, we evaluate the security and performance of the SuS design and implementation from Section 3.

4.1 Security

In Section 3.1, we described the threat model for SuS containerization. The following classes of attacks are relevant to this system:

- **Persistent Codebase Modification**: As a form of lateral movement, attackers may inject malicious code into the application server (or other programs on the host) that attacks subsequent benign users when they attempt to access the service. The SuS model is effective against this class of attack. We emphasize that attacks such as client-side XSS that are predicated on data stored in the backend being maliciously rendered client-side are not in this class.

- **Privilege Escalation**: The SuS model is effective against the exploitation of logic flaws in the server program codebase that would enable an unprivileged adversary to make privileged requests.

- **Privilege Misuse**: Sometimes the backend resource's existing access control system is too coarse-grained for some desired policies, allowing for otherwise-isolated users to access and modify each other's data. The SuS-to-Backend middlebox enables SuS to defeat this class of attack.

- **Other Attacks**: Social engineering, CSRF, client-side XSS, hardware attacks, container-breakout attacks, attacks targeting the kernel, and attacks against the backend resource software are out of the scope of SuS threat model.

We now test the SuS model's effectiveness. We have selected a representative set of common vulnerablitities and exploits (CVEs) fitting these classes to be executed against WordPress versions with known vulnerabilities. Figure 5 provides an overview of the CVEs and the class in which they belong. As a control, each exploit is run against an unmodified, containerized WordPress site with

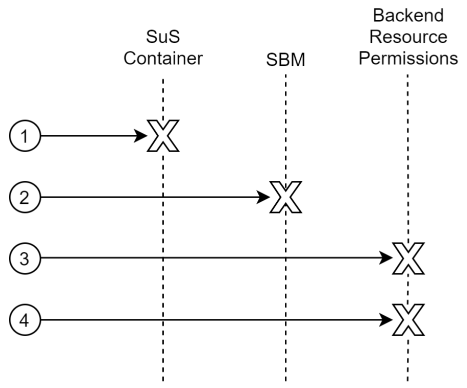


Figure 5: We evaluate the approach's ability to prevent attacker persistence via the `PhpUpload` attack (1), misuse of existing backend resource privileges via the `PasswordReset` attack (2), and confused-deputy-based privilege escalation via the `GdprOptions` (3) and `PhpUpload` (4) attacks. In all cases, the SuS approach blocked the attacks, which succeeded in the control group.

an appropriate version, allowing us to demonstrate the exploit's effect in the absence of the SuS model. We then run the same exploit against a modified WordPress codebase running inside a SuS environment to show not only that the attack is automatically neutralized, but that the container responsible for attempting the attack is frozen by the Container Manager.

4.1.1 Persistent Codebase Modification

CVE-2012-3578 describes a vulnerability (hereafter referred to as *PHPUpload*) in version 2.2.13.1 and eariler of the FCChat widget plugin for WordPress. This plugin includes a form in the `Upload.php` file that allows a user to upload an image file. By submitting a PHP file with the extension ".php.gif", an attacker can pass the plugin's file extension checks, and his or her malicious, arbitrary PHP script will be saved to a known directory inside the WordPress directory. Then, to execute the script, the attacker directly navigates to the file's location via a URL. In this attack, not only is the WordPress codebase maliciously and persistently modified, but the attacker can access the WordPress database credentials in the `wp-config.php` file to execute arbitrary MySQL commands against the backend data.

We expect that the fundamental operation of the SuS model is adequate defense against codebase modification. When a container image for a SuS-ified server program is built, the master server program *P* is copied into the image in binary or script form. All SuS containers for a single server application are spawned from this pre-built container image, so each container has a local copy of the server program *P'* derived from *P*. Any modification an adversary makes to *P'* exists only inside their container instance, and *P'* ceases to exist upon termination of the container. Thus changes to *P'* do not reflect in *P* or any other user's local program copy. Defending the database against the MySQL commands of the uploaded script is similarly fundamental to SuS. If permissions for each type of container user are set correctly in the Container Manager, an unauthenticated user should be easily caught trying to make privileged queries.

To explore our hypotheses, we run an experiment in which we execute this attack against WordPress 2.8.2 with the vulnerable FCChat plugin version in a SuS environment. We write a script for upload that uses credentials from the `wp-config.php` to delete all users in the `wp_users` table. Upon the adversary's navigation to `https://[IP]/wp-content/plugins/f cchat/html/Upload.php?id=1`, the Client-to-SuS Middlebox retrieves a container IP from the Container Manager and generates a `SUS_DEMULT_COOKIE` for the session. The adversary uploads the disguised PHP file and navigates to it directly. In the control group without the SuS approach, the attack succeeds.

However, in the SuS case, because the basic permission set for a new container does not include performing DELETEs on the `wp_users` table, the backend credentials hijacked by the adversary do not provide that capability. The SuS-to-Backend middlebox observes that the query is denied by the MySQL server and notifies the Container Manager to freeze the container. The script's execution is thus prevented. To prove that the malicious file's presence does not persist, we attempt to navigate to the files's path from another browser without the adversary's `SUS_DEMULT_COOKIE` and receive an HTTP 404 "not found" response. This is expected, because without the same `SUS_DEMULT_COOKIE`, the request is treated as coming from a new user, and thus is directed to a new container which is unaffected by changes to the adversary's container.

4.1.2 Privilege Escalation

CVE-2018-19207 describes a vulnerability (hereafter referred to as *GdprOptions*) in version 1.4.2 of the Van Ons GDPR Compliance plugin for WordPress [30]. The plugin's internal `save-setting` action fails to check the user's privilege level before using the HTTP request content to make configuration changes. As a result, unprivileged users can submit arbitrary option names and values to overwrite existing values in the `wp_options` table. With this capability, an adversary can set the default user role to administrator and allow users to register independently (by default, only an administrator can register new users). Then, after registering a new user, the adversary could remove the current site administrator and take control of all of the data in the backend database. Since the flaw in the WordPress plugin code enables unprivileged users to make privileged requests, this is a classic example of a Confused Deputy privilege escalation attack.

In our test without the SuS approach, the attack succeeds. However, when the SuS manager implements site permission policies, the attack fails. Unauthenticated users should not have `UPDATE` privileges over the `wp_options` table, so we configure the Container Manager not to endow new containers with this capability. We run the *GdprOptions* exploit against a SuS-ified verison of WordPress 5.1.1 with the aforementioned plugin version. In the first phase of the exploit, the adversary requests the WordPress main page to extract an AJAX security token. When this first request is made, the HTTPS proxy retrieved a container IP from the Container Manager and assigned a new `SUS_DEMULT_TOKEN` that maps the user to the container. The second phase of the exploit attempts to make a POST request changing the `wp_options` table values. The SuS-to-Backend middlebox intercepts an "UPDATE denied" message in transit from the MySQL server to the container (as a response to the POST request) and notifies the Container Manager to freeze the container.

4.1.3 Coarse-Grained Privilege Misuse

This attack class includes those by malicious users who take advantage of permissions that were intentionally given to access or modify data that the security policy did not intend. This can happen when the backend resource's native access control system is too coarse-grained to properly implement the desired policy (for instance, MySQL's lack of row-level permissions). This type of attack only exists because the SuS model restricts the privileges of each application server. In a typical WordPress configuration, the server already has full permissions over the database and thus unrestricted access to the data.

CVE-2009-2762 describes a vulnerability (hereafter referred to as *PasswordReset*) in WordPress version 2.8.2. The `wp-login.php` script fails to perform a proper sanity check on the type of the `key` query parameter for the reset password action. When `key` is set to an empty array instead of a string, such as in a GET request to `http://[DOMAIN_NAME]/wp-login.php?action=rp&key[]=`, the password of the first user in the `wp_users` table is reset automatically. This vulnerability is more powerful if the first user is an administrator and the adversary can access the password reset email. However, even if this condition is not met, the exploit can be triggered intermittently to repeatedly force a password reset, resulting in an account-level denial-of-service attack against a particular user. We confirm that the attack succeeds when SuS is not in place.

To prevent unauthenticated clients from exploiting this vulnerability, it is sufficient for the Container Manager to provision new containers without the ability to execute `UPDATE` queries against the `wp_users` table. However, authenticated users need the ability to modify their own information in `wp_users`, so we cannot prevent them from altering the content of other rows simply by using MySQL's table-level access control. With the SBM, we can prevent this attack with the following access predicate: `resourceRestrictTable ['subscriber'] ['UPDATE'] ['wp_users'] = "ID = :user_id"`. We empirically show that this defense is effective against *PasswordReset*.

We use the administrator console for WordPress to create a new user with the role of subscriber and an ID of 2. Before the attack, the `user_pass` field of the row with ID 1 in `wp_users` is `$P$B3VzGAHiiqb6BM72IkP9721u.KvO/p/`. To perform the attack, we first navigate to the site main page to get assigned a container, then log in as the non-admin user at the Authentication Container. The Authentication Container verifies that the credentials are correct and notifies the Container Manager that our container belongs to a user with the "subscriber" role. The Container Manager then updates this container's privilege to include performing updates on `wp_users` and relays the role update to the SBM along with the user ID. Then, we navigate to the aforementioned malicious URL as the authenticated subscriber. The SBM intercepts and modifies the resulting `UPDATE` request with the access predicate so the new request is ``UPDATE `wp_users` SET `user_pass` = '$P$BczMNW7LPQ4JiZZLu9JMk7HKnnqRNoO', `user_activation_key` = '' WHERE `ID` = 1 AND `ID` = 2``. Since no row in the table could possibly match both the query's `WHERE` clauses, the query response indicates that 0 rows were affected. We then re-check the row with ID 1 in `wp_users` and verify that the `user_pass` field is unchanged. This silently renders the attack ineffective. Since no table-level permissions were violated by the attacker, the SBM does not notify the Container Manager to freeze the container.

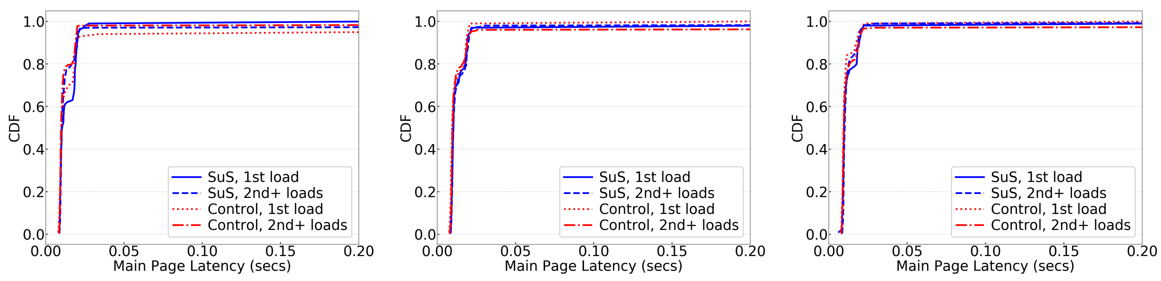
4.2 Performance

In addition to meeting its security promises, the SuS model needs to satisfy concerns about performance overheads and scalability. In this section, we examine the overheads associated with the SuS model's middleboxes, memory consumption, CPU usage, and overall latency, as well as how these individual metrics scale with number of concurrent users.

Unless otherwise stated, our performance evaluation compares the SuS-ified WordPress 5.1.1 blog against a containerized, unmodified, and shared WordPress blog that acts as the control. Our evaluation setup includes two VMs running on physically separate hosts. The first VM executes bot scripts that emulate user traffic directed at the second VM. The second VM runs either the SuS-ified webserver or the control server for each trial.

We provision both VMs with 16 GB of RAM to ensure that memory never becomes a bottleneck in the experiments. The bot VM is provisioned with 12 CPU cores, but we limit the server VM to a single core so that we can study CPU usage more effectively and prevent the SuS model from gaining an unfair advantage due to its larger number of concurrent processes. The server VM has a 40 GB SSD hard drive, and its vCPU process is pinned to a single core on the physical host. The host has 192 GB of RAM, 20 Intel(R) Xeon(R) Silver 4114 2.20GHz CPUs, and 21 TB of hard drive space, configured with RAID. The containers used in the performance experiments are not configured to enforce any CPU or memory limits.

Although our original SuS WordPress implementation uses the Lighttpd webserver, for our performance evaluation we use the Apache webserver on both SuS and control containers unless otherwise stated. In our initial testing, we found that Lighttpd's single-threaded, single-processed architecture strongly favors the SuS model because of the inherent multiprocessing associated with spawning multiple instances of the same server. When Lighttpd blocks on query responses from the MySQL server, no other user requests can be processed in the meantime. Thus, user requests are processed serially instead of in parallel, which is much more problematic for the control than the SuS scenario since each container in the latter only processes requests for a single user. In the SuS scenario,



(a) 10 concurrent users (b) 20 concurrent users (c) 40 concurrent users

Figure 6: CDFs of 100 home page load trials of a 1KB HTML page with 10, 20, and 40 concurrent users in SuS and control scenarios. Some outlier data points from both the SuS and Control experiments were removed from each graph for readability.

when a `lighttpd` process blocks, the scheduler can easily swap to another SuS container's `lighttpd` process. We therefore switch to Apache, which has a multiprocessed architecture, and find that this offers a computational advantage to neither scenario.

Each experiment adheres to the following general structure. The experiment is run as a script on the bot VM and consists of a number of independent trials. In each trial, the script first SSHes into the server VM and builds up the necessary infrastructure. For the control, this is as simple as starting the webserver container. For the SuS scenario, this includes starting all necessary processes and waiting for the pool of containers to fill up. Once the preparation is complete, the trial script starts any necessary bot scripts, each of which infinitely loop between fetching a random page from a small pool of options, using `wget`, and sleeping for 2 seconds. Then, the main metric collection bot is started, which performs some task and terminates. Finally, the other bots are gracefully terminated and the infrastructure is torn down on the server VM's side.

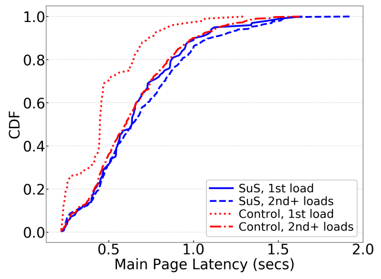
Since background container spawning and middlebox overheads are examined separately, these features are disabled during the other experiments. To demultiplex users without the CSM, we add a `-p` flag to the `docker run` command to map the SuS container's port 80 to an external port on the VM and let Docker's internal mechanisms handle the traffic forwarding. Each bot created for SuS trials is therefore assigned different ports to connect to on the VM.

4.2.1 Scalability: Latency and CPU

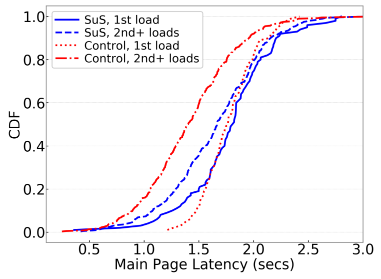
In our first evaulation stage, we compare the CPU usage and main page load latency of the SuS model against the control server. We test how these measurements scale by simulating 10, 20, and 40 distinct users. To accurately monitor the CPU during each trial, we record the usage of the server VM's vCPU on the physical host every 0.5 seconds using the `top` command. We perform each trial 100 times, and in each trial the main collection bot times how long it takes to fetch the main page and all associated resources, then repeats this 5 times for a total of 500 data points.

We first perform a microbenchmark by downloading a roughly 1KB static HTML page; the results of this experiment are shown in Figure 6. For all 3 user counts, more than 93% of page loads take less than 0.03 milliseconds. The results also show that the latency overhead of the SuS model is almost entirely nonexistent. We note that the microbenchmark is not representative of the complexity of a real website, and so we repeat the experiment using WordPress and loading its main page. WordPress has a complex bootstrapping process whenever a page is loaded which executes many PHP scripts and multiple database queries; it is well-known as a heavy-weight application [31].

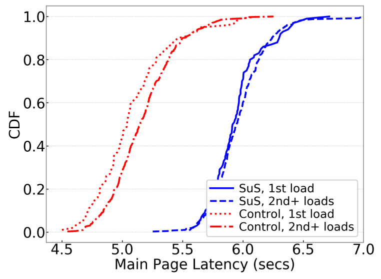
The latency results of the WordPress scalability experiments are shown in Figure 7. We also show the distribution of average per-trial CPU usage in Figure 8. We can see from the latency



(a) 10 concurrent users



(b) 20 concurrent users



(c) 40 concurrent users

Figure 7: CDFs of 100 home page load trials in WordPress 5.1.1 with 10, 20, and 40 concurrent users in SuS and control scenarios. Up to 2 outlier data points from both the SuS and Control experiments were removed from each graph for readability.

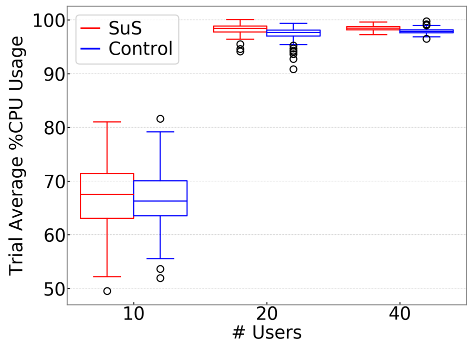


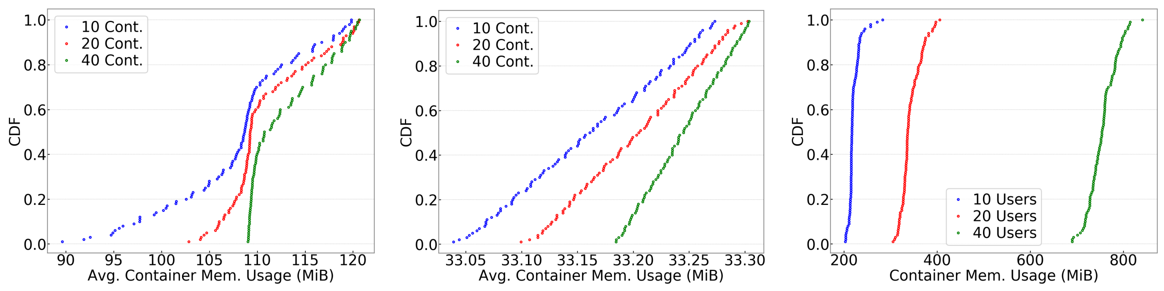
Figure 8: Comparison of per-trial average CPU usage between SuS and the control at various user counts.

graphs that at 10 and 20 users, the SuS model is competitive with the control. However, as CPU resources become scarce, differences begin to appear, as shown in the 40-user scenario. Due to the heavy-weight nature of WordPress, CPU resources start becoming very constrained at only 20 users for both the SuS and control; from there on, CPU becomes a large bottleneck in the server's user request handling. This bottleneck affects both the SuS and control latencies, but SuS is affected more, as shown by the roughly 1-second extra delay in the SuS model latency for 40 users.

From these results, we conclude the following. The CPU overhead introduced by the SuS model is light: at 10 users, the SuS model's median CPU usage is 67.5%, barely 1.2% over that of the control. However when CPU becomes constrained, the SuS model suffers more than the control. This is likely due to the fact that SuS must spend precious CPU cycles, which could otherwise be used servicing client requests, to do context switches between containers. We anticipate that in a real-world scenario, companies will already be provisioned against CPU exhaustion, making the SuS model's performance under such a bottleneck less of a concern.

4.2.2 Memory

To examine the memory overhead of the SuS model, we maintain an experimental structure that is largely the same as in Section 4.2.1. We record average memory consumption under the same user counts in 5 trials. For each trial, the main collection bot loads the WordPress main page 40 times and then terminates. We record the memory consumption on the server VM once per second using the `docker stats` command, and use the output of the `free` command to validate the data it returns. We use only the last 20 data points collected from each trial to sample only the stabilized memory consumption, for a total of 100 data points per server type and user count configuration.



(a) SuS with Apache (b) SuS with Lighttpd (c) Control

Figure 9: CDFs of 100 average per-container memory samples with 10, 20, and 40 concurrent users in SuS and control scenarios. In the control scenario there is only one container, so each data point represents the total memory consumption of the system. For the SuS model, average memory consumption must be multiplied by the number of users to calculate the total memory usage.

The results of this experiment are shown in Figure 9. The median average per-container consumption for SuS containers with WordPress running on Apache was between 109 MB and 111 MB. This corresponds to a five-fold increase in total memory consumption in the 10-user scenario over the control, and a nearly six-fold increase in consumption in the 40-user scenario. However, we had originally used Lighttpd in our implementation in Section 3.3.1; we switched to the much heavier-weight Apache server for the performance evaluation to give the control server a fair chance in our latency and scalability assessment. Since each application server process does less total work in the SuS model than in the control, the SuS model lends itself to using lightweight versions of the application server inside its containers. Thus, we also perform the experiment with SuS containers running Lighttpd. With this modification, the SuS model's average per-container memory consumption becomes about 33 MB, which is a reduction factor of more than 3. Compared to the control, the Lighttpd-based SuS model uses 1.5x the total memory consumption in the 10-user scenario, and 1.8x in the 40-user scenario.

While these overhead factors are manageable, we believe that they may be reduced even further. The file data each SuS container uses is practically identical, making it a prime candidate for copy-on-write memory strategies such as KSM [32] that would help with memory de-duplication. We note that while this type of approach could potentially create vulnerabilities that allow for unauthorized data-reads between containers, the host OS is part of our trusted computing base and exploits against it are out of the scope of our threat model.

4.2.3 Container Spawning

In this section, we examine the latency and computational overheads associated with spawning containers, and the effect of spawning on page load latency when peformed while users are accessing SuS containers. As part of the scalability assessment performed in Section 4.2.1, we wrapped the `docker run` commands executed by the Container Manager in a `time` command to measure the total time it takes to spawn a container. An analysis of these measurements is recorded in Table 3. Overall, the median container spawn time was roughly 2.3 seconds.

As part of our implementation of the SuS design, the Container Manager maintains a "pool" of containers that are ready to be assigned to new users. As this pool is depleted, it is replenished in the background by the ContainerPoolWatcher thread. For previous tests, the replenishing feature was disabled to isolate the variables we were trying to test. Here, we measure the impact of pool replenishing on end-user experience by running the 10-user scalability assessment procedure from Section 4.2.1 with the Container Manager constantly spawning containers in the background. We opt to measure only the 10-user scenario since even without pool replenishing, the CPU is very

User count	# Samples	Median (sec.)	Std. dev.
10	1000	2.25	0.309
20	2000	2.31	0.248
40	4000	2.35	0.185

Table 3: Analysis of container spawning elapsed time across various user counts, taken during the scalability assessment.

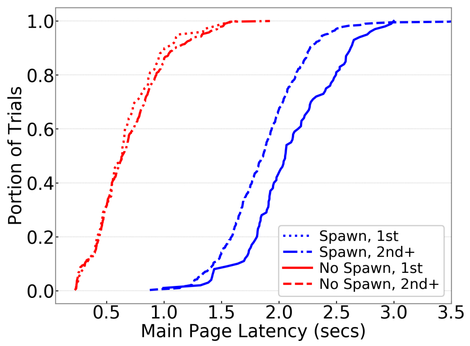


Figure 10: Main page load latency with 10 concurrent users in the SuS model with and without background container spawning.

constrained for higher user counts.

The latency results of the experiment are shown in Figure 10. Clearly, background container spawning has a significant impact on the user experience, raising the home page load latency by roughly 1.5 seconds. Whereas the 10-user median CPU usage in the scalability assessment was 67.5%, with container spawning the median CPU usage is 99.1%. Thus, Docker container startup is an expensive operation which must be done intelligently to minimize impact on latency.

There are a number of strategies that may ameliorate or hide the cost of container spawning. Most simply, we may accept that container spawning is CPU intensive and allocate an extra core to the server VM. By pinning the `dockerd`, `containerd`, and the Container Manager processes to this core, we may perform container spawning without detracting from SuS containers' time on the CPU. Another strategy orthogonal to this is intelligent selection of container spawning times. We could rewrite the Container Manager to make the ContainerPoolWatcher a separate process that shares memory with the main process, and greatly increase the pool watcher's `nice` value to make the scheduler deprioritize it. We could add a monitoring thread that waits for times of low CPU-usage to initiate a container spawn, so as to minimize the effect on end users. Finally, emerging virtualization technologies promise much shorter boot times than what we have reported with Docker; Amazon's Firecracker claims boot times of 125 milliseconds [33]. In the near future, the SuS model's Docker containers could be replaced entirely.

4.2.4 Middleboxes

In our final experiment, we examine the overheads associated with the use of our CSM and SBM middleboxes. For this experiment, we again load the WordPress homepage 5 times from our collection bot: first with no middleboxes, then with the CSM only, and finally with both middleboxes. We perform each sub-experiment 50 times, with only one SuS container and no additional traffic generation bots.

The results of our experiment are shown in Figure 11. The average overhead of enabling both middleboxes, compared to having neither, is 0.13s across the 250 data points. Middleboxes are often used in production environments with minimal impact on latency; we believe that with optimization,

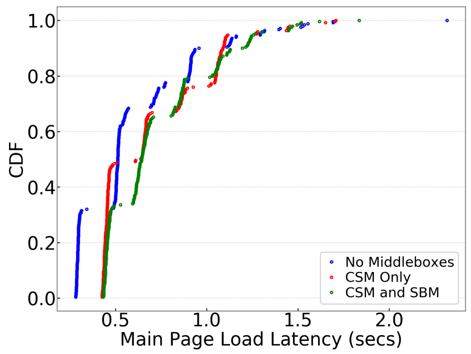


Figure 11: CDF of main page load times on a single SuS container with and without SuS middlebox components. We have not found an adequate explanation for the step-function-like quality of the graph.

our SuS middleboxes could achieve similar performance results.

4.2.5 Discussion

Here we summarize our findings from the performance evaluation of the SuS model.

Scalability and CPU: We find that the SuS model's computation overhead, compared to the control, is insignificant. When CPU is unconstrained, the latency overhead of the SuS model is similarly slight. When CPU becomes a bottleneck, page load latency under SuS suffers more than the control due to the need for context switching between containers.

Memory: Memory usage for both SuS and the control scale with the number of users. SuS does so by a higher factor, and so benefits from the use of lightweight application servers that may not be adequate for the control. Copy-on-write memory sharing may help further reduce SuS memory overhead.

Container Spawning: Although much better than VM boot performance, the act of spawning a Docker container is still somewhat slow and quite CPU-intensive. We have outlined strategies for ameliorating or hiding this overhead.

Middleboxes: Middlebox overheads are small and could be further reduced with optimizations.

5 Generalization

In Section 3, we created a basic implementation of the SuS model for a WordPress blog running on the Lighttpd webserver. Although some of the functionality of our implementation is specific to this use-case, much of the SuS infrastructure is not and can be abstracted into a configurable tool. In this section, we restructure and modularize the SuS components to arrive at a generalized model that should greatly simplify the process of applying this defense tool to various application servers. To evaluate our generalized model, we convert a popular mail retrieval protocol, IMAP, to operate with the SuS model and show that it can be done without further redesign of the SuS components.

5.1 SuS-to-Backend Middlebox

We now work to abstract away the application-specific components of the SBM. We show our analysis of the SBM's features in Table 4. Based on this analysis, we can see that although the SBM operates on resource requests and responses, it need not be concerned with how they are are transported or intercepted. The SBM can be decoupled from these mechanisms, allowing it to operate with many

Universal	Application-specific
Interact with Container Manager	Message transport protocol
ResourceRestrictTable	Request protocol & access predicates
Maintain user info from Container Manager	Roles, resources, user types
Respond to denied request	Request modification method
	Denied request error format

Table 4: Breakdown of functions of the original SBM.

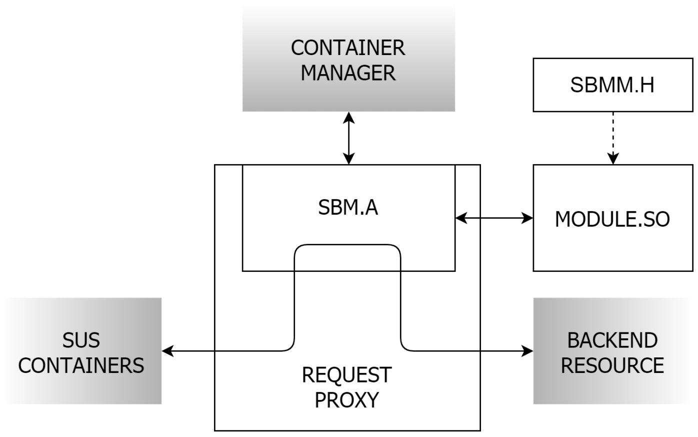


Figure 12: The SBM is implemented as a `.a` static library that can be compiled into proxy programs that are positioned between the SuS containers and the backend resource (database, remote filesystem, etc.). The SBM itself relies on a set of API calls defined in `sbmm.h` that must be implemented on a per-request-protocol basis. The API implementation is compiled into a `.so` dynamic library that the SBM can load without recompilation. In this way, the SBM allows API modules to be swapped between successive executions simply by changing a value in its configuration file.

different types of proxies (TCP, UDP, syscall interception, etc.). Similarly, the SBM should offload most of the analysis of the requests and responses to user-defined functions. The universal SBM functionality can exist as a mediating layer between the interception and modification layers. The SBM should enforce the existence of a table (known as the *ResourceRestrictTable*) that maps each request's *(role, resource, access type)* tuple to an access predicate, but not make any assumptions about the table's indexes or content. Finally, since we want to support many different application protocol types, the SBM should declare an API that developers can implement for their particular use case. For ease of development, the SBM will ideally allow users to swap between the API implementations easily and to write a module without needing to recompile the SBM.

To accomplish these goals, we design the modularized SBM as shown in Figure 12. We implement the SBM as a `.a` static library which can be included in the compilation of a program that proxies requests between SuS containers and the backend resource. The SBM library in turn loads a `.so` dynamic library based on a field specified in the configuration file; this dynamic library handles the protocol-specific request parsing and modification. The library must implement the API functions defined in `sbmm.h` (SuS-to-Backend Middlebox Module). With this program structure, the request-level functionality of the SBM can be altered between executions simply by changing which request module is loaded at runtime.

5.1.1 Request/Response Interception

The SBM library exposes a small API to the proxy, of which the important functions are `sbm_manipulateRequest(RequestContext&, buffer *)` and `sbm_monitorResponse(RequestContext&)`. When receiving a new request, the proxy simply needs to elevate it to the SBM using `sbm_mainpu-`

lateRequest() and forward the (potentially) modified result. Similarly, the proxy elevates responses via `sbm_monitorResponse()` to allow the SBM to check for request denial. Since the SBM cannot anticipate the format of requests, requests must be handled as variable-length arrays of raw bytes. These arrays are encapsulated in a contextual object that allows the proxy to specify the source of the request.

```
typedef struct
{
    void *data;
    uint32_t len;
} buffer;

class RequestContext
{
    buffer request;
    std::string src;
    std::map<std::string, std::string> contextDict;
}
```

5.1.2 SBM Library Internals

When the SBM is initialized via a function call from the proxy, it first loads a configuration file. This file includes the API module name, information for connecting to the Container Manager, and the list of roles, resources, and access types that will be used in the *ResourceRestrictTable*. The dynamic library specified by the API module name is loaded via a `dlopen()` function call and `dlsym()` is used to get each function pointer and ensure the API is completely defined. As with Section 3.6, the *ResourceRestrictTable* is populated by a `.tsv` file for each role that maps access types and resources to access predicates. In this way, the SBM controls the table object but allows the developer to control its content.

The SBM also creates a socket for interacting with the Container Manager and a separate thread for listening for login updates. Both the main thread and listener thread can share the socket, because only the former writes to it (when a request denial response is found), and only the latter reads from it. The listener thread uses the messages from the Container Manager to build up a dictionary of contextual information for each user, indexed by the container IP address. This dictionary is used by the API module to fill out variables in the access predicates when modifying requests.

5.1.3 Request Modification and Response Monitoring

During the course of execution, the proxy will elevate a request to the SBM as a `RequestContext` object via the `sbm_manipulateRequest()` function. At this point, the SBM accesses the user-context dictionary keyed by the `RequestContext.src` field to retrieve the user's role and any additional information. This data is then copied into the `RequestContext.contextDict`. If no such entry exists in the user context dictionary, the SBM inserts the default role specified in the configuration file. Next, the SBM passes the RequestContext object to the module API functions `getAccessType()` and `getResource()`, as these values are application-specific and determined by the content of the request. With the role, resource, and access type, the SBM then retrieves the proper access predicate from the *ResourceRestrictTable*. The `RequestContext` object is then sent along with the access predicate to the API `modifyRequest()` function. In this function, the access predicate is applied to the request. The resulting `buffer` structure is finally returned to the proxy program.

Once a response is elevated to the SBM via `sbm_monitorResponse()`, it is forwarded to the `isPermissionsErrorMessage()` API function. This allows the API module to specify whether a

Universal	Application-specific
Interact with Container Manager	User communication protocol
Container Manager communication protocol	User identifier value
User identifier mapped to IP address	Container prune strategy
Block user requests	

Table 5: Breakdown of functions of the original CSM.

response indicates that the associated request was denied due to insufficient privileges. Based on the function's return value, the SBM will use the response destination IP to notify the Container Manager to stop the associated container.

In this way, the SBM achieves high flexibility while still providing some foundational structure. With the SBM's access predicates, it would be feasible to completely supplant the backend resource's built-in permission control system and perform all access control using the *ResourceRestrictTable* (we considered doing this with our MySQL backend in the WordPress implementation). However, as the SBM becomes more powerful, it has the potential to become another Confused Deputy in the communication pipeline. If that were to happen, the SuS model would only complicate the problem instead of solving it. Thus, it is important that the backend resource's built-in access control system be used to the extent possible.

5.2 Client-to-SuS Middlebox

To generalize the CSM, we must abstract out code, as we did with the SBM, that is specific to the protocol with which clients communicate with their SuS containers. Since we are focused on protecting networked services, we *can* assume that, as the front-end proxy, the CSM will use sockets. In Table 5, we present our analysis of the functions of the CSM. Due to time constraints, this section describes work that has not been implemented; however, this section provides a comprehensive blueprint for future implementation.

As with the SBM, we implement the CSM as a `.a` static library which can be compiled into a proxy program to make the proxy "CSM-enhanced." At startup, the proxy must call the CSM library's `init()` function which, is establishes the socket connection used for communication with the Container Manager. This function sets up a separate thread belonging to the CSM library that listens for messages from the Container Manager about which IP to block.

Aside from startup and teardown procedures, the CSM library exposes three main functions that need to be called by the proxy. The `csm_retrieve_ip()` function takes a client identifier string and session ID and returns the associated container IP (for WordPress, the user identifier was the `SUS_DEMULT_COOKIE`). The `csm_ip_blocked()` function takes the container IP address and returns a `true` value if the Container Manager has specified that traffic to the container should be blocked because it is frozen. Finally, `csm_prune_container` is used to notify the Container Manager that a container with the specified IP can be reclaimed since the session has finished.

Although not required, we anticipate that the proxy will be set up in a multi-threaded fashion so each thread corresponds to one user connection. Here we present a pseudo-code example of how the proxy's client handler function can be "enhanced" using the CSM library:

```
function client_socket_handler()
{
    string container_ip;
    string client_id;
    string session_id;
    socket container_socket;

    while true
    {
        read whole message from user;

        if container_ip is not set
        {
            client_id = extract client identifier;
            container_ip = csm_retrieve_ip(client_id, session_id);
            container_socket = connect to container;
        }

        if csm_ip_blocked(container_ip)
        {
            break out of while loop;
        }

        forward message to container;
        wait for response from container;
        forward response to user;
    }

    ...

    csm_prune_container(container_ip);
}
```

Internally, the CSM maintains two data structures: a map of client IDs to container IPs and an array of blocked container IPs. Since both may be accessed by any of the proxy threads, as well as the CSM's internal Container Manager communication thread, both of these data structures are mutex-protected.

When a proxy thread makes a call to the `csm_retrieve_ip()` function, it provides the client identifier and an optional session ID. If the client identifier matches an existing key in the map structure, then the corresponding IP address is returned immediately. If no such key exists, the CSM must build a message (which may include a session ID) to request an IP from the Container Manager. Once an IP address is received, the mapping is added as a new entry to the map structure. Once the proxy issues a `csm_prune_container()` request for that container IP, the mapping is removed and a prune notification message is sent to the Container Manager.

An IP block message is the only message that may be received unprompted from the Container Manager. The CSM library's internal thread exists to listen for these messages, parse the IP address inside of them, and add the value to the blocked container IP array.

5.2.1 Pruning Strategies

Here, we briefly touch on when the proxy program should prune a container. In applications where a client connects using only a single TCP or SSL connection, we can take a client-side connection termination as indication that the client intends to end his or her session with the container. However, as we found with our WordPress implementation in Section 3.5, stateless protocols such as UDP and HTTP make it difficult to know when a session has ended. After each individual request or packet sent, the client may suddenly decide to stop interacting with the system without making any protocol-level notification to the server.

The proxy may be able to make use of application-specific information (such as an HTTP request for the URL associated with the logout button) to guess the user's intention to cease interaction with their SuS container and thus initiate pruning. However, these application-level heuristics are

Universal	Application-specific
Spawn/despawn container	Install/remove credentials
Select/maintain IP address	Upgrade privileges
Maintain cont. ID and role	Generate extra container configuration
Own component communication ports	`docker run` extra arguments
Freeze/stop container	Forward extra login context to SBM
Build/unpack component messages	Maintain/honor session ID
Close container/ports gracefully	

Table 6: Breakdown of functions of the original Container Manager

not guaranteed to cover all scenarios. Thus, as was the case with WordPress, the proxy will need modification to expire containers by time-stamping each request and noting which containers have passed a certain threshold. In this work, we make the design decision to leave this functionality outside of the CSM.

5.3 Container Manager

The CSM and SBM are modularized by abstracting SuS functionality into an API that is compiled into separate proxy code. The purpose of this compilation is to avoid making any limiting assumptions about the communication medium of the application server being protected. No matter the protocols are in use, we must still designate a process to manage the container information and handle buildup and teardown. During the modularization process, the Container Manager's codebase therefore remains mostly intact as a standalone program. We present our analysis of the functionality of the Container Manager in Table 6.

To maintain the existing codebase structure while allowing developers to inject additional functionality at various stages, we adopt a callback function system. At initialization time, developer-defined functions are registered with specific events via a callback manager. When an event occurs during execution, the callback manager is instructed to execute or "fire" the functions registered with that event. These functions may perform side-effects (such as installing user credentials), modify passed arguments, or return values to be used in subsequent execution.

For flexibility and cohesion with the new callback system, the container context object maintained by the Container Manager is re-defined as follows:

```
class ContainerContext():
    def __init__(self, container_id, container_ip, configs):
        self.container_id = container_id
        self.container_ip = container_ip
        self.status = ContainerState.READY
        self.role = None
        self.configs = configs
```

The only information about each container that the Container Manager "owns" are its ID, IP address, status, and site role. All application-specific information about a container is stored in the user-defined dictionary `configs`; for our WordPress implementation, that includes the database username/password, session ID, and the HMAC salt and key. Although a user's role is an application-specific value, we want to emphasize that upgrading privilege as a response to confirmed authentication is an essential component of the SuS model. Thus we design the container context object to enforce the use of roles.

In the following subsections, we address how our design supports each of the application-specific features listed in Table 6.

5.3.1 Additional Container Configuration Generation

We do not anticipate a need for developers to control the IP address assigned to each SuS container, as IP address selection should be irrelevant to operation of the SuS model. Therefore, in our system design, the Container Manager retains control over this configuration. We do anticipate that developers will need to assign some additional configuration based on the application being secured; this need is satisfied by the aforementioned user-defined dictionary `configs`.

To allow developers to populate this dictionary, we insert a callback action at the beginning of the `container_spawn` function of the `ContainerPoolWatcher` called *spawn_gen_config*. To functions that register with this callback, we pass the IP address of the container and an empty dictionary. In Python, objects such as dictionaries are passed by reference; thus changes made to the dictionary within each callback function are visible to the Container Manager. The populated dictionary is passed as an argument to subsequent callback functions and eventually is used in the creation of the container's `ContainerContext` object.

5.3.2 Retrieving Extra Arguments For `docker run`

As in our WordPress implementation, we anticipate that developers will want to pass a subset of the configuration values generated in *spawn_gen_config* callbacks to the container on startup in order to modify the behavior of the containerized code. Passing backend resource credentials and cryptographic information are straightforward use-cases for this functionality.

To allow developers to specify a set of arguments to append to the `docker run` command executed by the `ContainerPoolWatcher`, we add the *spawn_get_params* callback action. Functions registered with this action are passed the container IP, `configs` dictionary, and an empty list object which the functions populate to specify the desired arguments. With a loop, we then append any values in the resulting list to the string containing the `docker run` command before executing it.

5.3.3 Installing Credentials and Performing Side-effects

At some point during the container spawning/despawning process, the developer may need to perform side-effects, particularly installing/removing credentials from the backend resource. To handle any general use-cases that correspond with container spawning and despawning, we add the following callback actions to the callback manager: *spawn_pre*, *spawn_post*, *spawn_pre*, and *despawn_post*. All of these actions pass the container's IP address and `configs` dictionary, but only *spawn_post* and *despawn_pre* are passed the container ID, since they occur while the container is running.

For our WordPress implementation, we perform database credential installation during *spawn_pre* and removal during *despawn_post*, since these actions do not require the container to be running.

5.3.4 Upgrading Privileges

While it is important for the Container Manager to know which users have logged in, we expect developers to handle upgrading privileges upon a confirmed authentication. We therefore create the *login* callback action that fires after the Container Manager receives a message from the Authentication Server. We require that the Authentication Manager's login notifications begin with the container's IP address and new role, as those are relevant to the Container Manager. However, we also anticipate that developers may need the Authentication Server to send additional information for application-specific responses to login; for our WordPress implementation, that includes the time of login for determining the expiration time of a session ID. Thus, a function registered with the *login* action is passed the container's IP address, ID, the user's new role, `configs` dictionary, and any additional information appended to the received login notification message.

5.3.5 Forwarding Information to the SBM

Upon receiving a login notification from the Authentication Server, the Container Manager passes the received IP address and role information to the SBM so that it may apply the correct access predicates from the *ResourceRestrictTable*. We anticipate that developers will need to forward additional information in these messages to fill in variables within the access predicates (see Section 3.6).

We therefore alter the *login* callback action to expect functions registered with it to return a string value containing the additional information the developer wants forwarded to the SBM. After the string value is appended to the IP and role in the message, and the appropriate message header is prepended, the information is forwarded.

For our WordPress implementation, we use this feature to supply the SBM with the site user ID associated with each container, which is used as a variable to control which rows of the `wp_posts` table an authenticated user can modify.

5.3.6 Handling Session IDs

Although we do not anticipate many application servers (other than webservers) needing session IDs, we make them a full-fledged feature of the SuS model. Session IDs are used to influence the decisions of the Container Manager at two stages of execution. First, when the CSM requests a container IP address, a provided session ID may influence the Container Manager to return the IP address corresponding to that session ID instead of a fresh one. Second, when the CSM notifies the Container Manager to prune a container for inactivity, the existence of an active session ID may cause the Container Manager simply to stop the container instead of destroying it.

To handle IP address retrieval, we add a callback action to the `container_get()` function called *get_session*. This function receives the passed session ID and is expected to return an IP address. If the return value is `None`, we assume either the developer has decided that the session ID is expired or the session ID is invalid or nonexistent. In this case, we continue on to retrieving an IP address from the `unassigned_containers` list.

To handle container pruning, we insert a callback action after the sanity checks in the `container_stop()` function called *stop_session*. Functions registered with this callback are expected to return `True` if the developer wishes the container to be saved; otherwise, `False` or `None`. Based on the result of firing the callback action, we either call `docker stop` or invoke the Container Manager's `container_despawn()` function.

5.4 Applying the Modularized System

With the SuS components rewritten to abstract out protocol-specific behavior, we test the SuS model's modularity by applying it to another networked service. The service we chose is the Internet Message Access Protocol (IMAP), used for retrieving user mail from a remote server. E-mail is a popular form of communication, so the IMAP protocol is widely used today. IMAP is also useful for testing our system's modularity because it involves a different type of backend resource than web applications. Whereas WordPress's sensitive data was stored in a MySQL database, IMAP's email data is stored using the Linux filesystem. An IMAP server is also unlike a webapp in that users are required to log in before accessing any other functionality of the system, so there is no need for cookies to remember a user's data beyond a single session.

In the following subsections, we describe the functioning of the IMAP server and how it is ported to the SuS model.

5.4.1 Background

The IMAP protocol is one of two widely-used protocols for retrieving mail from a remote server. Unlike its counterpart, POP3, IMAP does not delete the email from the remote server after it has been locally downloaded. Emails, drafts, and local sub-folders are synchronized across a user's devices. Both IMAP and POP3 can be configured to work one of two different storage structures: *flat-file* and *maildir*. In the flat-file system, each user's entire mailbox is represented as a single file. In the maildir system, each email is represented as a separate file, and each user has a directory with three subdirectories: `/tmp`, `/new`, and `/cur`. Mail transfer protocols such as SMTP and LMTP use the `/tmp` directory during mail transfer for storing partially-written email files. Once the transfer is complete, the email file is moved into `/new`. The `/new` directory holds all complete email files that have not yet been read by the user. Once the user logs in and checks his or her inbox, the emails in `/new` are moved to `/cur` with the rest of the user's previous mail. For our purposes, we opted for the maildir storage system over flat-file because representing emails as individual files within well-defined directories, instead of lines within a single file, allows us to make better use of the existing Linux file access control mechanisms.

5.4.2 Containerizing IMAP

To prepare IMAP for integration into the SuS model, we need a container image with an IMAP server that defers its authentication to an external, trusted entity that will operate as the SuS Authentication Server. We begin by installing Postfix and Dovecot, two widely-used Linux mail server programs for SMTP and IMAP/POP3, on the host machine. We opt to use a virtual domain (abc.com) and virtual users, so that we may aggregate all maildirs in a single superdirectory. This decision also gives us better scalability, as we would otherwise have to assign a dedicated Linux user account for every mail account we wanted to set up.

To resolve the virtual domain and its mail server to IP addresses, we create a local authoritative DNS server with MX, NS, and A records for the abc.com zone. We then point the host to first query this local DNS server before attempting to resolve elsewhere.

Finally, we need a method to prevent the standalone Postfix process from accessing mail that has already been written to the maildir. Following an online tutorial [34], we set up a non-containerized Dovecot process running only the Local Mail Transfer Protocol (LMTP). Then, we configure Postfix to pass the received mail to Dovecot via Unix socket. Dovecot then delivers the mail to the correct directory under ownership of `dovecot-user`. Relevant pieces of our configuration files for Postfix, Dovecot-LMTP, and containerized Dovecot are shown below.

```
alias_maps = hash:/etc/aliases
alias_database = hash:/etc/aliases

# Configuration for virtual domains and LMTP delivery
masquerade_domains = abc.com
virtual_mailbox_domains = abc.com
virtual_mailbox_base = /home/vmail
virtual_mailbox_maps = hash:/etc/postfix/virtual
virtual_minimum_uid = 1000
virtual_uid_maps = static:6001
virtual_gid_maps = static:6002
virtual_transport = lmtp:unix:private/dovecot-lmtp
```

```
protocols = lmtp
mail_location = maildir:~/
listen = *
```

```

# Which directory to deposit mail into, based on domain and username
userdb {
	driver = static
	args = uid=dovecot-user gid=dovecot-user home=/home/vmail/%d/%n
}

# LMTP Configuration
service lmtp {
	unix_listener /var/spool/postfix/private/dovecot-lmtp {
		group = postfix
		mode = 0600
		user = postfix
	}
}
protocol lmtp {
	postmaster_address = postmaster@sus-ubuntu-2.cs.wpi.edu
}
service lmtp {
	user = dovecot-user
}
```

```
protocols = imap
mail_location = maildir:~/
auth_mechanisms = plain

passdb {
	driver = passwd-file
	args = /etc/dovecot/passwd
}

userdb {
	driver = static
	args = uid=6000 gid=6002 home=/var/mail/mail_mount
}
```

Next, we need to restrict the system data that each IMAP user is able to access. It is intuitive that a particular user should only be able to access his or her own maildir from within the containerized IMAP server, and we should therefore mount only that directory. Barring a hypervisor breakout, each user would be rendered totally incapable of accessing other users' files. However, this line of thought leads straight to the following paradox: we cannot know which maildir needs to be mounted until the user has provided login credentials to the running IMAP container, but `docker` can only mount volumes at the time of container instantiation. Thankfully, we are not the first to notice the usefulness of dynamic volume mounting; an online article by Jerome Petazzoni describes a method for accomplishing this [35]. The technique involves entering the container to manually create the device containing the entire host filesystem, mounting the entire filesystem from within the container, and then mounting the desired subdirectory from that mount into its final desired location. This cannot be accomplished solely with `docker exec`; using that command causes the namespace with which one enters the container to prevent performing mountings for security reasons. So we use a non-Docker tool called `nsenter` to enter the container with a desired namespace that does not have such restrictions. Based on Petazzoni's article, we wrote a script to take in a mail username and dynamically mount the corresponding maildir into the given container ID (shown below).

```
#!/bin/sh

# Requested username and container ID originating request supplied as args
CONTAINERID=$1
MAILUSER=$2

# Lookup the PID associated with the container, needed for nsenter
CONTAINERPID=‘docker inspect --format {{.State.Pid}} $CONTAINERID‘

# Create the device for the filesystem root in the container
docker exec -d $CONTAINERID mknod --mode 0600 /dev/sus--ubuntu--2--vg-root b 252 0

# Create requisite folders
docker exec -d $CONTAINERID mkdir -p /tmpmnt
docker exec -d $CONTAINERID mkdir /var/mail/mail_mount

# Mount the fileystem root device to /tmpnt
sudo nsenter --target $CONTAINERPID --mount --uts --ipc --net --pid mount
    /dev/sus--ubuntu--2--vg-root /tmpmnt

# Mount the desired directory from within /tmpmnt to mail_mount dir
sudo nsenter --target $CONTAINERPID --mount --uts --ipc --net --pid mount --bind
    /tmpmnt/home/vmail/abc.com/$MAILUSER /var/mail/mail_mount

# Cleanup
sudo nsenter --target $CONTAINERPID --mount --uts --ipc --net --pid umount /tmpmnt
docker exec -d $CONTAINERID rm -r /tmpmnt
```

With this mechanism, we can mount the proper directory once a user has authenticated. As part of the SuS model, however, we cannot trust the containerized IMAP server's native authentication to accurately report which user has logged in. This is because the container becomes untrusted as soon as a user connects to it. Dovecot v2.3 has native support for PAM, and can be configured to send a JSON string containing a user login and password hash to a remote HTTP server for login approval. Dovecot's PAM system does not supplant its native authentication, but it can be configured to launch before or after. The PAM configuration in the SuS IMAP server is shown below.

```
auth_policy_server_url = http://172.17.0.1:8080/
auth_policy_hash_nonce = SwarmsOfSuperSuspiciousServers
auth_policy_server_timeout_msecs = 5000
auth_policy_hash_mech = sha256
auth_policy_request_attributes = login=%{requested_username} pwhash=%{hashed_password}
    remote=%{rip} device_id=%{client_id} protocol=%s
auth_policy_reject_on_fail = yes
auth_policy_check_before_auth = yes
auth_policy_check_after_auth = no
auth_policy_report_after_auth = no
auth_policy_hash_truncate = 0
```

We create a simple HTTP server in Python3 that verifies the username/password-hash pair sent by Dovecot, and accepts or rejects the login based on a file of the correct password hashes. This acts as the Authentication Server in the SuS model for IMAP. After a set of credentials are verified to be correct, but before a response has been sent to the container, the Authentication Server invokes the dynamic volume mounting script with the given username and a reverse-searched container ID.

Consequently, the maildir is always loaded into the container before the Dovecot instance attempts to access it.

Because Dovecot's PAM mechanism does not replace its native authentication, we still need to provide a file of password hashes to the containerized server. Since we have been using password-hash-file-based authentication, this would allow each SuS IMAP server to have access to the whole password hash file, with undesirable security implications. However, since we already know at the Authentication Server which username the client is attempting login as, we can simply split the password hash file into a set of files with only one username/hash entry per file. The relevant password file is then copied into the container at the same time we perform the maildir mount. This method prevents a user from ever having access to other users' login information, even if the container is totally compromised.

Now we have a containerized IMAP server ready for use in the SuS model. Building the SuS IMAP server is much simpler than building one for WordPress. For one, Dovecot already has a remote authentication mechanism, so we do not need to invent one. Also, all clients have the same role and must log in before they can access any sensitive data. Finally, the permission control boils down to directory access, which we can control easily via bind mounting.

5.4.3 Complete System

We now explain how the modularized SuS components are combined with the IMAP containers to create a complete system. Due to time constraints, the work described in the section is not implemented.

- **Client Demultiplexing**: The CSM library is compiled into an SSL-to-TCP proxy, and the SSL ID used by each client becomes the identifier used to distinguish them.
- **Container Pruning**: Since the SSL protocol is not stateless, we use connection termination as a catalyst for container pruning in the CSM.
- **Container Spawning and Despawning**: No credentials need to be installed or removed in the backend resource. Our user script for the Container Manager registers a callback function with the `spawn_gen_config` action. When a container is going to be spawned, we generate a nonce for the IMAP server to use when hashing user-supplied credentials to the Authentication Server. As in the WordPress implementation, this hash nonce is based on the container IP and a shared key, so information supplied to the Authentication Server cannot be spoofed.
- **Container Freezing**: SuS containers running IMAP can be considered compromised when the internal `dovecot` process attempts to read or write files outside the associated user's mail directory. Thus, the SBM is compiled into a program that monitors log files of the `auditd` service for improper read/write syscalls made based on the calling PID and the destination file's path. We therefore need to know the PID associated with each IMAP process, so information is recorded in each container's context object after the spawn occurs during the `spawn_post` action.
- **Query Scoping**: With IMAP, the backend resource is a filesystem, so the "requests" are simply read/write syscalls against specific files. For simplicity, we compile the SBM library into a program that monitors log files; thus, the SBM is not positioned as a "man in the middle" to scope queries. If this functionality is desired, we may be able to compile the SBM into a loadable kernel module that replaces the relevant syscalls and can approve or deny them.
- **Login Handling**: Our HTTP Authentication Server is extended to communicate role updates to the Container Manager process. The function we register with the Container Manager's `login` callback action forwards the container IP, username, and associated `dovecot` PID to the SBM.

- **Extended Sessions**: IMAP does not call for the use of extended sessions marked by session IDs.

We have now arrived at a generalized SuS model that can be applied to a wide range of server/backend pairs. The Container Manager, SBM, and CSM are modularized so they can be applied in different contexts by changing configurations instead of rewriting them on a per-application basis. To demonstrate this capability, we port `dovecot`, an IMAP protocol server, to use the generalized SuS model and do so without any further changes to the SuS components.

6 Conclusion

The goal of this work is to create a generalized model for protecting application servers against lateral propagation and Confused-Deputy-based privilege escalation attacks. Internet application servers are currently designed to maximize resource efficiency by handling many users that fall within disparate privilege classes. This co-hosting of users allows adversaries to attack other clients or exploit privilege escalation vulnerabilities to read and modify sensitive backend data. Our proposed solution, the SuS model, separates each user into an isolated execution context with tailored backend permissions so that these attack classes are thwarted. We design our system to achieve a high level of security while minimizing the amount of code modification required for porting an application server to SuS.

In the first phase of this work, we define a set of SuS components that have well-defined roles and responsibilities for enabling user separation, trusted authentication, state maintenance, and compromise detection. We implement these components for the popular web deployment tool WordPress. We also create a containerized version of the WordPress codebase, using Docker, that is appropriate for use in the SuS model.

In the second phase, we select three real-world exploits against WordPress that are representative of our threat model and show that the SuS model is effective at detecting and neutralizing them. We go on to perform a comprehensive evaluation of the SuS model's performance with a focus on end-user experience. We find that in an unconstrained system, the CPU and latency overheads of the SuS model are low. Further, we find that memory consumption and container spawning CPU usage are non-trivial overheads, and we outline a set of strategies for ameliorating them that will be examined in future work.

In the final phase, we arrive at a generalized version of the SuS model by modularizing the SuS components. We find that the two SuS middleboxes can be effectively rewritten as libraries for proxy programs to abstract away protocol-specific details while still implementing SuS-specific features such as query scoping, client demultiplexing, and communication with the container manager. We also find that the container manager process can offer flexibility to deployers while maintaining its original structure by adopting a callback system. We then evaluate our modularization efforts by applying the SuS model to the IMAP mail retrieval protocol, which has a very different privilege model than WordPress. Although some of the work in this phase is theoretical, due to time constraints, we show that IMAP can be ported to the SuS model with little effort.

References

[1] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, 1988.

[2] R. Cheng, W. Scott, P. Ellenbogen, J. Howell, F. Roesner, A. Krishnamurthy, and T. Anderson, "Radiatus: A shared-nothing server-side web architecture," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 237–250, 2016.

[3] "Docker." `https://www.docker.com/`. Accessed February 6th, 2020.

[4] "Demystifying containers - part i: Kernel space." `https://medium.com/@saschagrunert/demystifying-containers-part-i-kernel-space-2c53d6979504`. Accessed April 28th, 2020.

[5] "Docker docs: Use volumes." `https://docs.docker.com/storage/volumes/`. Accessed April 28th, 2020.

[6] "Cwe-441: Unintended proxy or intermediary ('confused deputy')." `https://cwe.mitre.org/data/definitions/441.html`. Accessed April 29th, 2020.

[7] "Blue pill: The first effective hypervisor rootkit." `https://www.zdnet.com/article/blue-pill-the-first-effective-hypervisor-rootkit/`. Accessed April 27th, 2020.

[8] X. Li and Y. Xue, "A survey on server-side approaches to securing web applications," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 54, 2014.

[9] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation.," in *USENIX Security Symposium*, 2003.

[10] S. Chong, K. Vikram, A. C. Myers, *et al.*, "Sif: Enforcing confidentiality and integrity in web applications.," in *USENIX Security Symposium*, pp. 1–16, 2007.

[11] B. J. Corcoran, N. Swamy, and M. Hicks, "Cross-tier, label-based security enforcement for web applications," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 269–282, 2009.

[12] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 291–304, 2009.

[13] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications.," in *USENIX Security Symposium*, vol. 64, 2011.

[14] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan, "Mace: Detecting privilege escalation vulnerabilities in web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 690–701, ACM, 2014.

[15] Z. Su and G. Wassermann, "SQLCheck: The essence of command injection attacks in web applications," in *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, 2006.

[16] E. Chin and D. Wagner, "Efficient character-level taint tracking for Java," in *Proceedings of the 2009 ACM Workshop on Secure Web Services*, pp. 3–12, 2009.

[17] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 1–20, Springer, 2008.

[18] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pp. 351–366, 2007.

[19] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *2014 IEEE Symposium on Security and Privacy*, pp. 292–307, IEEE, 2014.

[20] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pp. 335–350, 2007.

[21] S. Brookes and S. Taylor, "Containing a confused deputy on x86: A survey of privilege escalation mitigation techniques," *International Journal of Advanced Computer Science and Applications*, 2016.

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

[23] C. R. Taylor, "Leveraging software-defined networking and virtualization for a one-to-one client-server model," Master's thesis, Worcester Polytechnic Institute, 2014.

[24] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with" return-less" kernels," in *Proceedings of the 5th European conference on Computer systems*, pp. 195–208, 2010.

[25] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: lightweight kernel protection against return-to-user attacks," in *21st {USENIX} Security Symposium ({USENIX} Security 12)*, pp. 459–474, 2012.

[26] "Qubes os: A reasonably secure operating system." `https://www.qubes-os.org/`. Accessed April 27th, 2020.

[27] "Wordpress." `https://www.wordpress.org/`. Accessed November 18th, 2019.

[28] "Usage statistics and market share of WordPress." `https://w3techs.com/technologies/details/cm-wordpress`. Accessed February 6th, 2020.

[29] "LigHTTPD." `https://www.lighttpd.net/`. Accessed February 6th, 2020.

[30] "Privilege escalation flaw in wp gdpr compliance plugin exploited in the wild." `https://www.cisecurity.org/advisory/a-vulnerability-in-wordpress-wp-gdpr-compliance-plugin-could-allow-for-arbitrary-code-execution_2018-127/`. Accessed December 10th, 2019.

[31] "Wordpress PHP performance benchmark." `https://www.savvii.com/blog/wordpress-php-performance-benchmark-2019/`. Accessed April 17th, 2020.

[32] "Kernel same-page merging." `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/chap-ksm`. Accessed April 20th, 2020.

[33] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *USENIX NSDI*, pp. 419–434, 2020.

[34] "Building an email server on ubuntu linux, part 3." `https://www.linux.com/topic/networking/building-email-server-ubuntu-linux-part-3/`. Accessed April 28th, 2020.

[35] "Attach a volume to a container while it is running." `https://jpetazzo.github.io/2015/01/13/docker-mount-dynamic-volumes/`. Accessed April 28th, 2020.