Developing Single Use Server Containers

A Major Qualifying Project
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science
in
Computer Science
by

Christopher Bell

Kamil Gumienny

Nicholas Odell

March 6, 2020

APPROVED:

Professor Craig A. Shue, Project Advisor

Abstract

Security breaches and attacks are a major concern for service providers on the Internet, as the associated monetary and reputational costs are detrimental and, in some cases, unrecoverable. A more secure method for hosting web applications, rather than the traditional many-to-one client server setup with unprotected database access, is necessary to mitigate such risks. One security concern for service providers is persistent malware. Traditional hosting systems typically utilize a single or, in the case of more popular services, several large public-facing servers with which clients can connect. Such a setup allows adversaries to potentially impact every individual attempting to use the resource. The second security concern is data theft. Web server software is typically running alongside more sensitive components used by the application, such as a database holding confidential user information. The web server acts as a gateway to the backend systems. Privilege escalation attacks, such as a confused deputy attack [1], aim to compromise systems with this configuration where they attempt to manipulate the web application, an entity with "privileged" access to backend resources, into carrying out commands on the database with the intention of exfiltrating, modifying, or deleting data.

This project designs and implements a one-to-one client server model with database protection to defend against both the persistent and confused deputy attacks mentioned. We test our system to collect and analyze usage metrics in order to determine the feasibility of the system in production.

Our system uses WPI's InstructAssist (IA), an online course management system, to test the monetary, resource, and technical feasibility of creating individual instances of a web server for each authenticated client. This system is made possible by leveraging the capabilities of Docker containers, a lightweight option for creating virtualized systems or applications. Virtualization refers to the process of running software behind a layer of abstraction on top of the host operating system or hardware, separating it from the host system. Unlike virtual machines, containers share a hypervisor with the underlying operating system. This reduces the cost of creating, destroying, and rebooting container instances, which can be completed in seconds, thus making recovering from an attack or removing persistent threats a simple and quick operation.

To defend against confused deputy attacks, each of the system's containers have predetermined access privileges to the back-end database determined by the container's authenticated user's role, such as "student", "teaching assistant" and "professor," within IA. This restricts data retrieval and operations on tables within the database. Additionally, our implementation includes a query restrictor in the form of a MySQL proxy that adds restrictions to database queries in order to prevent malicious adversaries from tricking their container into fetching, modifying, or deleting unrelated data.

Utilizing virtualization to create a one-to-one client-server model introduces many new

protective measures for the benefit of web applications. While it helps make applications more secure overall, it especially protects against privilege escalation attacks, denial of service attacks, and data theft attacks. Our testing confirms that our system protects against an SSH attack by exposing one container. The attack shuts down that individual container without impacting the service of others or the host system. From our latency testing we found that containerizing Lighttpd results in a 391 ms delay for users as compared to 139 ms when running Lighttpd on the server. With the addition of our Proxy to the container running Lighttpd, the delay increases to an average of 535 ms. Results shown throughout testing prove our system has the potential to operate well in a production environment. One such result being the measured CPU load of the system when performing stress tests, which primarily remains below 35% when testing with 40-users and below 80% when testing with 80-users (Appendices I and J).

Contents

Chapter 1

Introduction

Web applications are streamlined to serve thousands of users from a single system to conserve computing resources such as processors, memory, and disk space. Many of such systems, like those involved in banking or authentication, handle sensitive user data that is entrusted to the application by the users. These systems are attractive targets to malicious actors who are looking to disrupt a service or to compromise them in such a way as to gain unrestricted access to sensitive data. With global Internet usage logging at nearly half of the world's population in 2016, at 3.4 billion users and rising, the incentive for disrupting or compromising shared infrastructure with sensitive data by adversaries is a prominent threat for service providers [2]. The Economist evaluated that data is now even more valuable than oil, hence the incentive for malicious adversaries to attack these systems [3]. A study by the University of Maryland found that cyber-attacks are carried out nearly every 39 seconds [4]. Not only are such attacks progressively more common, the cost of recovering from one is also becoming increasingly expensive. It is estimated that the average cost of a data breach in 2020 will exceed $150 million [5]. Our project aims to defend against such attacks by assisting in addressing web server infrastructure vulnerabilities stemming from resource sharing.

Web applications are one of the systems against which adversaries can leverage privilege escalation attacks because they are exposed to the open Internet and therefore publicly accessible. Finding systems to attack is simple as there are automated tools used for querying systems for possible vulnerabilities. Once an adversary finds a vulnerability, they can easily research how to attack the system or download tools to carry out the attack as well.

Internet hosts and service providers are tasked with preventing unauthorized access to their infrastructure. The issue is that these large and complex systems can be compromised by exploiting a single vulnerability, one that can manifest in a myriad of different places. Possible sources of vulnerabilities include outdated software, weak passwords, mismanaged permissions,

or unresolved program bugs. By gaining access to a system and obtaining permissions reserved for higher-level users, an adversary can perform what is referred to as a vertical privilege escalation attack. This allows them to carry out commands reserved for the administrator, such as copying data or planting malware to steal it. Adversaries may attempt to obtain the highest level permissions, such as "root" in a Linux-based system or "Administrator" in Windows, to perform a crippling attack or even hide one. Privilege escalation attacks are so prevalent that in a survey to 1,000 different IT decision makers in the United States and the United Kingdom, 74% of those whose companies were breached admitted privilege escalation was involved [6].

Privilege escalation is a gateway for obtaining a privileged position from which an adversary can perform another attack. As the highest privileged user on a machine, an adversary may access, steal, and/or modify data. The adversary can also perform a persistent attack by planting malware such as spyware, ransomware, remote access backdoors, and keyloggers. These attacks are especially expensive to recover from as malware was evaluated to be the most expensive kind of attack costing an average of $2.6 million per company per breach [7].

Possible examples of consequences to being the victim of a privilege escalation attack can be illustrated through two hypothetical, although realistic, scenarios. In our first scenario, an adversary finds a way to escalate themselves within an online shopping system to root level. At this point, the adversary can do things such as deleting orders from customers' accounts, duplicating private information such as addresses and payment methods for future use or sale from the e-merchant's database, or even bringing the system down to prevent orders from being placed and users from accessing their accounts. This is an example of a system privilege escalation attack where an adversary gets access to the back-end system of a publicly shared resource. In our second scenario, the adversary finds a way to execute SQL queries on an online course management application's database, similar to the one we are using for our project. With this capability they can view and change grades, delete user information, and/or modify assignments or their submissions. An exploit in an educational application that allows an adversary to modify the database constitutes a FERPA violation against the application's owner, typically the school. Such an attack would damage the school's reputation and result in significant financial damages. This is an example of privilege escalation where the adversary does not necessarily gain access into the back-end server, but is able to trick a privileged entity (the application) to execute commands in an unintended way to achieve the same result. Our goal is to prevent these attacks by implementing a system with sufficient security in place so that, with regards to our scenarios, customers can log into an e-merchant's shop with certainty that they are safe to enter their credentials, and students who access their classes online are sure of the integrity of their submitted assignments or grades.

Privilege escalation attacks are difficult to prevent on traditional systems, as adversaries need only to find a single exploit to allow them to access the rest of the system. Properly

protecting every possible attack vector directly is inconceivable. A system administrator's best line of defense against any kind of attack is to keep software up to date, manage permissions appropriately, and check system logs regularly. This defense, however, can only mitigate the chances of a successful attack. A newly discovered vulnerability, often known as a zero-day exploit, can still affect a server that has been patched for every vulnerability in the system or application.

Our project changes the approach to web application hosting. Our team uses an architecturally different approach capable of reducing the threat of privilege escalation as a whole so that, even if a vulnerability is found, the back-end system is still protected and other users are not impacted by an attack.

We achieve this by using virtualization to separate the application from the sensitive resources it uses, such as a database. Virtualization can be leveraged for security because it allows system administrators to hide components behind a layer of abstraction separating the backend components from the virtual application or system using them. Thus virtualization between the host operating system and the application itself creates a two-way barrier. In the event of a crash or breach in a virtual system, only that specific instance would be impacted and not any other instance nor the underlying host. Virtualization between the VM and the operating system also means that if an application or component of the operating system is vulnerable, it would be inaccessible from the virtual application due to the abstraction barrier.

Our system uses the fact that virtualization through containers (specifically Docker containers) is capable of achieving the one-to-one client-server model necessary to prevent attacks via privilege escalation or persistent malware. Despite their security benefits stemming from virtualization, containers are traditionally marketed to application developers because of their portability. They allow an instance of a system or application, referred to as an image, to be designed or modified on one machine and then transferred and executed on another without having to accommodate for the differences in the machine's software nor the underlying operating system (so long as it can run the Docker containerization engine). This simplifies moving an application from a development environment to a testing or production environment as the system or application within the container runs seamlessly across different hosts. This capability is appealing to organizations developing software as they create a run-anywhere platform for the systems or applications being developed. Additionally, containers are capable of spreading out their active services across multiple physical machines, thus scaling dynamically and naturally on distributed systems.

Using containers for virtualization is similar to using virtual machines (VMs) as they achieve many similar effects. The two differ with regards to the hypervisor location/abstraction level. Containers share a hypervisor with the underlying operating system as opposed to having their

own instance of one like VMs. Containers are therefore smaller, and capable of starting, stopping, and restarting much quicker than VMs. This allows developers to make changes to the software inside of a container image (a saved representation of an application or system set up to run inside a container), or the container configuration as a whole, and redeploy the container shortly thereafter. We decided to use this particular technology because it would allow us to the minimize resource usage that is necessary for virtualizing a large number of server instances.

Docker containers also provide security benefits over traditional VMs outside of the scope of virtualization. They allow for automated rollouts and rollbacks which, when combined with their lightweight nature, makes shutting down and restarting containers computationally inexpensive operation. This is essential for updating and applying changes such as security patches to a system without causing a significant downtime. The oversight of the containers' status can be handled automatically by the Docker daemon, which monitors each container's health, as defined by the system administrator. This means that containers which have stopped functioning properly (potentially due to being compromised by an attacker) can be stopped and replaced by a fresh container. Further, resource limits for each container can be manually defined to make sure each can only use enough as should be necessary [8]. To achieve our second security goal of preventing privilege escalation attacks, we employ a Query Restrictor, a MySQL proxy that scans queries being sent to the database holding user and class information. Access to back-end resources by any one of the running containers will have immutable permissions set at the start of the session which, when combined with a Query Restrictor, will dynamically manage the data available and returned to a user from the back-end. The system components and containers will be managed by a script running on one of WPI's servers (Appendix A).

The security provided by virtualization and the versatility and the undemanding nature of containers makes them viable candidates for achieving a one-to-one client-server architecture and the associated security benefits. Our team's goal for this Major-Qualifying Project (MQP) is to host WPI's InstructAssist class management software using this model. Thus, each user session will be have its own instance of the web server in a container, preventing denial of service attacks by attacking a single point of failure and data theft attacks by employing a Query Restrictor.

Implementing this system may come with potential drawbacks that our group aims to evaluate. Firstly, price is one variables we will take into consideration. Docker is available as two versions: a free version for general-use and a paid one for enterprise operations. Our team must determine if the enterprise version of Docker would be necessary to achieve our goals. We also test the implementation's feasibility from a resource usage perspective. Despite containers being less reliant on resources than virtual machines, they will require more resources to run than a single a web server due to the quantity of servers running simultaneously as each user has their own instance. There may also be additional latency for users because connections are being managed

by a proxy and database queries are all scanned by the MySQL proxy before sending data. Our biggest concern is how resources scale based on the number of users. Technical feasibility is the last variable we measure using our system. As we prepare InstructAssist to run within a container, we document and report the complexity of the setup to determine the difficulty of porting a web application from a single server configuration to ours.

Leveraging virtualization to create a one-to-one client-server model provides many new protective measures for potentially vulnerable web applications against privilege escalation attacks, denial of service attacks, and data theft attacks. The components we describe act as an adaptable model which can be implemented to work with a multitude of applications in order to strengthen their security. While more definitive testing should be done and future improvements made, the metrics we collect show that the system can operate well in a production environment as well as properly defend against the aforementioned attacks. There is a latency increase when running our system that uses containers and a proxy as opposed to running Lighttpd traditionally on a system and the resource usage is much higher. Despite the latency and performance scaling, we conclude that the system could run in a production environment.

Chapter 2

Background

In this chapter we discuss the technologies we explore and utilize in making our Single-use-Server system and how they mitigate various security vulnerabilities related to web applications. Virtualization is the technology responsible for achieving our security goals because it is able to separate an instance of a web server from the host system running services in the backend, thus protecting those services. We highlight the two methods we can use to achieve virtualization, virtual machines and containers, and and elaborate on their respective pros and cons. We explain our reasoning for using containers and go in-depth on how they achieve virtualization and security. Additionally we explore the security features of Docker, the software platform we use for hosting and managing containers in our project. Lastly, we review the vulnerabilities we are defending our web application against with this project and speculate on how virtualization and Docker can defend against them.

2.1 Virtualization

Virtualization in computing is the process of running a virtual system on top of the underlying hardware. It is achieved by "hiding" the virtual system behind a layer of abstraction [9]. This abstraction is typically achieved at one of two levels. The first of these levels is on top of the hardware itself which is referred to as hypervisor-based virtualization. With this approach, the virtual system separates from the host system entirely by using its own operating system. The hypervisor-based approach is commonly used by applications like VirtualBox and VMWare. The second level of virtualization, container-based virtualization, can be achieved on top of the host operating system and, like its name suggests, is used by containers. This means that the virtual system does not use its own operating system and instead shares the host's kernel [10].

These two levels of abstraction have their own distinct use cases. Unlike the container-based approach, the hypervisor-based one allows for more flexibility in designing the virtual environment. For example, users can add custom kernel modules without affecting the host system. VMs also offer more security since sharing a kernel with the host is marginally riskier than using a dedicated one. Lastly, VMs do not have restrictions on what systems and applications they can run based on the host operating system. Containers on the other hand are restricted to using the same operating system as the host. The container-based approach may be a better approach if kernel-level modifications and operating system are not a factor for the system designer. Its virtualization performs better with regards to resource usage. It allows more environments to run alongside each other as each one does not require the memory necessary for running a separate operating system. In terms of CPU usage, the lack of a separate operating system means less overhead for the processor(s). From the perspective of the host operating system, each container appears simply as another process [10].

2.2 Containers

The idea behind containers dates back to around the 1970's with Unix V7. Despite being realized decades ago, the technology became a popular virtualization option recently with the help of Docker, a company offering a container-based development platform founded in 2013 [11, 12]. Containers are a packaging mechanism that abstract applications or systems from the system's environment. Containers simply encapsulate the binaries, libraries, and packages required for the application or systems individual purpose, while the kernel features are still handled by the underlying host's operating system [8]. As previously mentioned, the ability to create a run-anywhere platform for organization developing software was the driving force behind Docker's rise in popularity.

Our team's choice to work with Docker is based on its pricing, ease-of-use, community support, and superior functionality to that of other providers, such as Mesos Containerizer or CoreOS rkt. Docker is offered both as a free service or as an enterprise edition with features assisting in large-scale container deployment along with dedicated support. The installation of the software is easy and does not require any complicated configuration from the user, allowing them to start developing immediately after installation. Docker also comes with container management features including the ability to manage the resources allocated to each container such as network bandwidth, memory, and processing power.

Sysdig, a software company offering a DevOps tool used for container monitoring, revealed that Docker was the most popular container platform in 2018 hosting 83% of container deployments and again in 2019 hosting 79% of container deployments that were monitored by their

software [13, 14]. Because Docker has the largest userbase of all container platforms, our team will be more likely to troubleshoot problematic scenarios. Docker also offers technologies and services built to enhance the platform, such as Docker Hub, the world's largest library and community for container images, and Docker Swarm, a Docker container orchestration tool used to manage a distributed rollout of containers [15, 16].

Despite all of its appealing features, there are drawbacks to using Docker requiring our consideration. Firstly, there is no shared concept of a "sysadmin" between containers, making the configuration of communication between different servers complicated. Docker is still under heavy development, meaning that developers can expect to deal with the hassle of needing to update their Docker software often to keep up-to-date. To build on this point, there are security issues that are continually being discovered and addressed due to the software's young age [17]. A major drawback with regards to our performance testing is inaccurate application performance statistics from within a container due to skewed data resulting from the resources and workload of the host system. The data our team collects will be specific to our configuration, thus developers looking to replicate our system may collect substantially different results across deployments. Lastly, containers have a slightly weaker security than virtual machines because of the shared kernel space. If an adversary is able to access the kernel with privileged capabilities, they can "break out" of their container and gain access to resources reserved for the host operating system. Despite these drawbacks, Docker still proved to be our group's best option for our project.

2.3 Container Security

In this section we elaborate on how containers achieve virtualization despite sharing an operating system with the host. We include an overview of the six requirements that much be satisfied for container instances to be isolated and how Docker achieves through namespaces, capabilities, and cgroups.

2.3.1 Isolation

In order for operating system-level virtualization to work appropriately, Thanh Bui of Aalto University School of Science lists the six requirements that must be achieved [10]:

Process isolation: Container instances are seen as processes to the operating system. Docker is able to wrap each container into a namespace by using the PID to limit its permissions and visibility. Namespaces are hierarchical, so containers cannot see other processes unless they are the container's child processes. Namespaces separate identifier tables within the kernel global resources. Thus, filesystems, partitions, processes, stacks, and other components are divided

up and separated between containers and hosts, restricting the container's access to resources [18, 10].

Filesystem isolation: The host and container's filesystems must be protected from unauthorized access across the abstraction layer. This goal is also achieved through mount namespaces. Containers are restricted to mounting events that have an impact inside the container. Filesystems within the kernel are allocated for containers but they are not namespaced. If containers were given traditional access to these filesystems, they would be able to inherit the host's filesystem, a vulnerability we describe later in this section. Docker's solution is to restrict write access to filesystems and prevent containers from remounting them. Containers employ copy-on-write mechanisms allowing multiple containers to run the same image simultaneously, sharing that image's filesystem, but internally copying and then writing files that need to be modified. Additionally, Docker restricts the CAP_SYS_ADMIN capabilities, which is the equivalent of giving a process root privilege inside the kernel, and instead provides capabilities for achieving the least privilege needed to operate.

Device isolation: Containers must be restricted in their access to devices, such as memory or terminal access, to prevent attacks on the host operating system. By using the device whitelist controller to limit the amount of devices a container can access, it prevents it from creating new device nodes and also restricts the container process from using a pre-made device found within the container to communicate with the kernel. Containers could hypothetically bypass these restrictions if ran as privileged.

IPC isolation: Namespaces are used to limit containers from inter-process communication. Each container runs in its own IPC namespace preventing it from recognizing and communicating with other processes as well as stopping malicious actors from using their container to manage other applications through their APIs.

Network isolation: Docker uses network namespaces to give each container its own IP address, routing table, and network devices. The intention is to prevent a container from eavesdropping on the network communication from the host or other containers. When a new container is created, it receives its own virtual Ethernet interface and is connected to Docker's default Ethernet bridge with other containers, allowing each container to send out packets. Docker's approach to network isolation does not filter packets which makes ARP spoofing and MAC flooding attacks possible.

Limiting of resources: Docker uses cgroups to limit system resources. This can be configured for groups of containers or for individual containers. Cgroups restrict its member(s) to only use a predefined amount of a resource. The idea behind cgroups is to prevent one process from starving another by misusing resources [18, 10].

In summary, the OS kernel uses namespaces, cgroups, and capabilities to secure containers

and achieve virtualization. Without each one of them, vulnerabilities can be exploited inside a container or on the host system by an adversary who can bypass the abstraction layer. For instance, using purely filesystems to achieve filesystem isolation has resulted in successful escapes from inside of containers into the host filesystem. The attack is carried out as follows: first, the adversary finds a way to access the kernel and collects the kernel symbols. They then loop back to the parent of the process, typically the host operating system. Finally they mount the host operating system's filesystem as their container's filesystem which allows the container to operate within the same filesystem scope as the host [19].

Capabilities are another major factor is securing containers. Before capabilities were introduced, containers would have to be run with root privileges to achieve certain functionality, such as opening a network socket [18]. If an adversary has root access with control of a container, they could use superuser capabilities such as CAP_SYS_MODULE to insert or remove kernel modules. For this reason, Docker disables a large number of Linux capabilities by default to prevent adversaries from having root access while stile providing containers with enough privilege to carry out their task. Similar concerns were the reason behind the development of different kernel security systems which are discussed later in this chapter.

2.3.2 Policies

Docker uses multiple forms of access control policies to isolate processes. These policies are specifications on how data can be accessed, who is allowed to access it, and under which circumstances it can be accessed. Access control policies can apply to processes or systems as a whole. They can be broken down in to Role Based Access Control (RBAC), Discretionary Access Control (DAC), and Mandatory Access Control (MAC) [20]. Regardless of the mechanism used, an access control policy is only considered safe if no information can be released to an unauthorized person. Our group will be hosting Docker in an Ubuntu 16.04 system and, as such, we will focus on the security features and policies Linux offers for virtualization.

RBAC is a policy enforced by the system with which a user can only execute a task if they have the permissions associated with the file or task. One popular form of RBAC is role-based access with administrator specified rules. The role can be specified based on user credentials or by location. An example of a location based policy is RuBaC which uses network addresses, the domain, and network protocol to determine which users are granted access and at which level to a service [21]. If an employee were to receive a promotion and change offices to one with a network scheme correlating to higher privileged database access, they would automatically have their privileges modified without database or user access reconfiguration [21]. RuBaC is an example of policies for users, however the same concept is applied to processes within a system –

a processes' role is first evaluated before being permitted to execute an action.

DAC is an access control mechanism that associates an owner and groups with permissions over an object. With DAC there is a notion that every file has an owner. The owner has access control of a file and only they are allowed to determine who else is allowed to read or modify the file. This mechanism alone cannot prevent unauthorized access or modifications to a file due to the possibility of the owner account being compromised [20].

MAC has the notion that a user can only access a file or execute a command if they have a permission level equal-to or greater than the object or command requires. MAC addresses the possibility of the owner account being compromised, the issue undermining DAC, by controlling access between processes and system objects. This access policy's rules are enforced by the kernel and Docker integrates two classes of policy enforcement. The first, type enforcement, uses labels to mark containers and restrict them from reading and writing from any other process. The second, multi-category security, is used to prevent containers from modifying other containers. Together they are able to isolate containers from each other. Additional security can be achieved through third party kernel security systems. AppArmor is one such system. It uses mandatory access controls but instead of users, it limits the scope to individual processes [10].

A research group has suggested implementing a vulnerability prevention mechanism through policies regarding Docker images, the applications loaded inside containers. An analysis of both official and community images on Docker Hub, Docker's online repository, found that images contained 180 vulnerabilities, as reported in the Common Vulnerabilities and Exposure's (CVE) website, on average. Another finding was that many of the official images are typically not updated for over 100 days at a time. The lack of updates allows vulnerabilities to quickly propagate from official parent images to community and personal images [22].

As a method of achieving layered security and partially addressing the previous issue, the researchers have suggested that Docker images should come packaged with MAC kernel security modules. The authors of the research suggested using SELinux specifically. This would allow a more fine grain method for deciding which applications and users can read, append, or modify certain files. The restricted access would not be modifiable by users or applications unless they gained root privileges and modified the kernel, the authority enforcing MAC. SELinux modules would enforce the principle of least privilege for the specific applications packaged in an image. That way if a vulnerability within the image is found and exploited, the modules would prevent it from being leveraged [23, 24].

Despite all of these precautions, there are Docker vulnerabilities that external to virtualization or access control policies. "Shadow Containers" are compromised containers that are able to retain data between restarts. Malicious code is conserved and executed despite the notion that persistent data should be erased on container restarts with Docker. The vulnerability was possible

because Docker for Windows exposed its API via TCP, a setting enabled by default, allowing an attacker to execute commands remotely via the Docker daemon. Attackers exploited this vulnerability to write their own shutdown script, one allowing them to save data across shutdowns, hence creating Shadow Containers [25].

Ultimately, Docker takes many precautions to prevent vulnerabilities, however their software is still very much an underdeveloped technology with many lingering security issues and therefore cannot not be blindly trusted to provide users with absolute security.

2.3.3 Privilege Escalation

Privilege management is clearly an area of key concern when it comes to container-based virtualization. A container's permission level at run-time translates to its capabilities outside the container as well because it shares a kernel with the host system. There are three privilege-related attacks worth noting: horizontal escalation, vertical escalation, and confused deputy.

A horizontal escalation attack occurs when an adversary is able to access other accounts with the same privilege level as their own. As an analogy, this is equivalent to an employee of a company gaining access to the accounts of their fellow coworkers. In doing so, this employee is also able to access their coworkers' personal information, resulting in confidentiality issues and giving the employee the opportunity to manipulate and use said personal information for their own personal gain.

On the other hand, a vertical escalation attack involves an adversary gaining access to an account with more privileges than their own, such as one of an administrator or root user. Depending on how powerful the compromised account is, the adversary may be able to steal credentials, upload and run malware, or manipulate the system. Even after enacting the attack, the adversary can obscure their actions by erasing any logs or data keeping track of their actions, potentially leaving the victim incapable of recognizing that any malicious activity ever occurred.

Finally, a confused deputy attack is a type of privilege escalation attack where privileged program is tricked by a user or another program on behalf of a user into misusing its authority. The most prevalent confused deputy attack is a SQL injection attack which we elaborate on in the "Restricting Database Queries" section later in this Chapter. Some other examples of confused deputy include cross-site request forgery (CSRF), clickjacking, and FTP bounce attacks. In both CSRF and clickjacking, the user is tricked into visiting websites where malicious actions are executed and, in the worst case scenario, infect the user with malware. The FTP bounce attack is different in that it is an attack where the user is tricked into giving the adversary access to TCP ports. In all of these scenarios, the outcome can be catastrophic as the adversary is able to attack using a wide variety of methods ranging from tricking the user into downloading malware to

having the user log into a fake website in an attempt to steal their credentials.

2.4 Virtualization and Security

Virtualization, for security purposes, is the result of separating different applications into their own virtual spaces. When a vulnerability originating in one service is exploited in one machine, it is much less likely to infect another because of a layer of abstraction separating them [26]. New ways of leveraging virtualization through containerization are continually being developed. From a containerized environment for running untrusted applications, like Windows Sandbox[27] or Bromium[28], to an entire OS built around containers to ensure high levels of protection, such as Qubes OS, more and more people are turning to containers to address security and scalability problems [29].

Windows Sandbox benefits from its lack of data persistence and high security via isolation from the host. However Windows Sandbox differs in that its functionality is based around providing a safe environment to run untrusted applications. Due to its purpose, it discards everything on the device when the application is closed. Plus, it is based in the host's system while our web server implementation retains relevant information and operates within its own system and is not wholly dependant on the status of the user's system [27].

Bromium is an example of a company utilizing containers in a unique way for security. Bromium defends against malware through virtualization. When a user makes a potentially dangerous action, such as visiting a web page, opening a document, or downloading an attachment, Bromium creates a micro-VM where the process is then run and contained within without noticeably affecting the user. When the user ends their interaction and closes the task, the micro-VM is destroyed as well as the process contained within it [28].

Qubes also has security through isolation in the form of "qubes" – securely isolated compartments. Therefore, if one qube is compromised, it would not affect the others, as is the intention with our containerized servers. However, there is no separate physical machine needed for Qubes either and it serves the purpose of a highly-secure operating system [29]. Another approach for achieving security through virtualization includes breaking an individual application up into containerized segments. Passe breaks an application into individually sandboxed processes and limits communications between them dynamically to prevent adversaries from using the weakest link, whether it be a component or communication channel to access restricted resources [30].

2.5 Restricting Database Queries

Web development has become a relatively easy process thanks to both the ease of use of modern web server systems and publicly available guides on Internet. The side effect is that developers are creating applications that do not have appropriate security protocols in place to protect their applications and their backend solutions for persistent data storage. One attack designed to exploit weak security protocols is the SQL injection attack. This attack is used to manipulate a database by entering database management code into an input form field. Despite the attack being named SQL injection, it is not limited to just the SQL database management language. When the website goes to enter the data into the database, it executes the command the adversary entered instead [31].

Imperva, a data security service provider for web applications, released statistics on SQL injection attacks against its client's applications. Although Imperva does not release the number of websites it protects, it claims that there are millions of attacks per day and over 80% of the web applications they protect get attacked every month with SQL injection [32]. The organization also provided information on how to prevent the attack, which includes methods such as sanitizing input to make sure it is not a SQL command and limiting database permissions so if a query is made, it is not executed [32].

SQL injection attacks are a type of confused deputy attack. The adversary confuses the deputy, the web application with privileged database permissions, to execute some command on the database. InstructAssist uses sanitization to prevent SQL commands from being executed. Another precaution that some web applications use is a SQL proxy that modifies a query based on some conditions. SQL proxies are traditionally connected to multiple databases at a time and therefore they are used for load balancing servers or sending queries to the appropriate database.

Chapter 3

Implementation

In this chapter we describe the design of our Single-use-Server infrastructure by expanding on our design decisions and intentions, detailing each of the components, and providing a walk-through of the system's internal functionality. We complement the information with a state chart with every component to illustrate the communication channels and a state chart for initialization and user functionality available in Appendix A and B respectively.

Our goal was to design a system that was not specific to one application but instead "portable" for use with any web application. For the purpose of our initial design, we use InstructAssist (IA), a class management tool used by Computer Science professors at Worcester Polytechnic Institute (WPI). This was the perfect candidate for our implementation as our Advisor was actively teaching classes so by using this web application we would have the chance to test it and collect data from a class of students. Additionally, InstructAssist uses different access types, such as professors, students, and teaching assistants. With this design we can test if our system is able to properly defend against attacks trying to exploit the system using different privileges and access types.

With a web application in mind we design the architecture of the system. The final design for the system uses four main components: a Proxy, an Authentication (Auth) container, a Query Restrictor (QR), and the Single-use-Server (SuS) manager. In summary, the Proxy manages the communication between users and their containers, the Auth Container handles IA's authentication, the QR scans and modifies SQL queries depending on a user's permission level, and lastly the SuS Manager manages all container instances. Once we confirm that the implementation operated properly, we plan to test it in a production environment.

3.1 Planning the Container Configuration

The first step in our design process involves creating the individual container images. This process involves selecting a base image, modifying its configuration files, and adding our web application.

Our team has access to a virtual environment hosting IA and its database which itself holds a schema for a single class instance. InstructAssist is running using Apache2 inside of this environment. Despite having all of the configuration files that we would need to run IA in an Apache image, we decide to switch the server in favor of a Lighttpd server. Lighttpd is a lightweight web server with minimal resources usage. Resource usage is an important factor for our design as there may be over 100 containers actively running at a time. One such case where this could occur with respect to InstructAssist would be moments prior to a due date for a submission. Other factors that we consider in choosing a server include the ability to host a web application written in PHP, write output logs for us analyze usage data and/or debug the application, and handle user sessions using cookies.

The configuration files IA uses with Apache are rules that map URLs to filesystem locations. We port the rules written for these mappings to Lighttpd. We then test IA inside of Lighttpd in our virtual environment. Unfortunately some of the rules IA needs are not directly transferable from Apache. After rewriting them, the application works exactly as it has using Apache. With the configuration files ready we move the server system into a container image.

The first step in creating our custom IA image is accessing the Docker Hub to download the official, unmodified Lighttpd image. We create a dockerfile that makes all of the necessary changes to the image including opening up the correct ports and moving the preconfigured configuration files inside the image. By using a dockerfile, we can update the Lighttpd image without having to repeat the process of configuring the image from scratch as the script executes all of the changes on an update. We also write three other scripts to make modifications to the image. One of these script packages our IA code into a compressed file, moves it to the appropriate place inside the image, and extracts the files to their appropriate place once inside. The next script writes data into the container to personalize its permissions for the user logging in. The last of these scripts modifies settings to get the image to send logs to the docker daemon so we can follow logs inside the container with the command "docker logs *containerID*".

With a Lighttpd configuration capable of correctly running IA inside of a container, the next step is to modify IA code itself to prepare it for the image.

3.2 Understanding InstructAssist Authentication and Database Communication

InstructAssist is a PHP web application that connects to a database with student data. For conformity reasons, IA takes two precautions regarding authentication. Firstly, IA makes sure that a user is authenticated before being able to navigate to any page of the application, even ones that are not class-specific. Secondly, it handles authentication through WPI's Central Authentication Service (CAS), a Kerberos authentication system, rather than having its own local system.

When a user connects to IA, they are sent to CAS to authenticate and then back to the application. IA has to confirm that the authentication at CAS was successful. To be able to do this, IA appends a client's IP address alongside other session variables and a nonce as query parameters to a CAS redirect from IA. CAS returns a purported username, those same variables, and a hash-based message authentication code (HMAC). The HMAC is constructed using SHA256, a one way hashing function that takes all of variables it was provided, arranges them in a particular order, and computes the hash algorithm with a secret key that is shared between IA and CAS. If the user authenticates successfully, they are redirected back to IA from CAS with the user's WPI username, the variables used in the hashing algorithm without the secret key, and an HMAC stored as query parameters. To confirm the authentication, IA has to recreate the HMAC that CAS provides using its copy of the secret key. If IA is able to compute the same HMAC CAS returns, the purported username is indeed that of an authenticated WPI user.

In our containerized version, we modify the way IA handles authentication. We remove the call to CAS from within IA itself and assign that duty to a separate entity in our system, the Auth container. We do this because we did not want any unauthenticated clients to interact with an IA container instance before authentication to avoid tampering. Ultimately, this ends up being the only change we make to IA to use it with the SuS system.

Additionally, the legacy IA system uses the IP address as an HMAC variable. IA stores the user's HMAC it receives from CAS in a session variable alongside the variables that are used to create it, including the user's IP address which IA updates dynamically. This check occurs on every page load. Therefore, legacy IA is able to detect when a user switches IP addresses as the recreated HMAC would not match the original. If the user's IP address changes, they are asked to reauthenticate with CAS. With this system, users are able to use their session cookies for a whole day before having to re-authenticate.

We modify the process for creating the HMAC in our system from that of legacy IA's. In our implementation we can not check the IP because the user's IP address is now hidden behind the abstraction provided by our proxy. Our system instead uses a "demultiplexing" cookie, an identifier cookie unique to each user. An explanation on how the system uses it can be found in

the "Auth Container" subsection later in this Chapter.

3.3 Development Environments

Our design and testing platforms are virtual machines running an identical environment to that of legacy IA. In specific, the machine is running the x86_64 bit Ubuntu 16.04.6 LTS on top of two 64-bit CPU cores clocked at 2 GHz with 2 GB of memory.

An exact copy of the legacy environment was essential for configuring the containerized version of the software. By using an environment with the same configuration as the Legacy IA server we are able to collect logs and compare the data without having to account for differences in hardware. In this environment, we also have an empty database running the same version of MySQL as Legacy IA and using identical table schemas. The one major difference between the Legacy version and ours is that our database only contains data for a single test class as opposed to the many gigabytes of data on Legacy IA's database. Although we did not collect any data regarding read speeds from each of the databases, there may be a marginal time delay between fetching data from the two databases.

Within the design environment, our team plans to experiment with different technologies and versions of the system in order to achieve our final result. While we are building our system, we document every change and maintain a "SuS playbook," a document with a set of instructions for converting IA from its existing framework to ours, a list of packages that we install for our system to run, and instructions on how to configure the components in our system prior to running. Documenting the implementation steps is essential to being able to replicate this project for other systems. With proper documentation we hope to develop a guide for converting any arbitrary web application using the one-to-many server architecture to the SuS-model.

When we finally have a working system in our design environment, we will configure it inside the secondary testing environment by following the steps in our playbook. Our goal is to ensure we documented everything correctly so our advisor will be able to recreate the system in his production environment when we are ready to test it with students.

3.4 SuS Components and Workflow

In this section we take a low-level look at the Proxy, SuS manager, Auth Container, and Query Restrictor. We then describe the interactions between each of the four components. Of the four components, the Proxy and SuS manager are specific to the SuS system whereas the Auth Container and Query Restrictor are both implementation specific components for InstructAssist.

Should an application use authentication with a Kerberos authentication system or a SQL database, it can take advantage of these two systems.

3.4.1 Proxy

The Proxy handles communication with both new and previously authenticated clients, and handles IP association requests from the Auth Container. The Proxy uses software defined networking (SDN) to allow implementation-specific forwarding rules and traffic handling. The Proxy is responsible for identifying where all network traffic should be routed within our system, whether it is to the Auth container or to an active IA container. Ultimately this routing depends on how far along a user is in the authentication process. In general, the proxy will receive requests from the client, make some necessary modifications depending on the request, forward the request to the proper destination, and then wait for and forward the response to the client. In order to do so, the proxy maintains a mapping of demux cookies and the IA container IP that they should be sent to (or none, in which case it is sent to the Auth Container by default). There are three cases the Proxy encounters in our system, which depend on the presence of a demux cookie in the client's request's headers, and the state of the demux-cookie-container-IP mapping.

If the client is new, and thus lacks a demux cookie (or posses an expired or otherwise invalid demux cookie), a new one, in the form of a unique, randomly-generated 32-byte alphanumeric value, will be created for them. The new cookie will be added to the mapping initially with no destination IP, causing the Auth Container to be used as the destination. The new client's request will then have its demux cookie, as well as an encoded form of their intended destination URL on the server, added as the GET parameters 'clientid' and 'destination', respectively, before being forwarded to the Auth Container. These parameters are needed so that the Auth container knows the demux cookie of the client when it receives the forwarded request, and so that the Auth container knows where to redirect the client back to when authentication is complete.

If the client already has a valid demux cookie, but have not yet successfully authenticated, they will still simply be forwarded to the Auth Container. The request modifications mentioned in the previous case will also still occur. This case occurs most frequently upon returning from WPI CAS to have their authentication confirmed by the Auth container, but can also occur if they previously attempted authentication and failed, or had logged out of the system.

If the client has a valid demux cookie, and has successfully authenticated (i.e. the proxy has received a demux-cookie-container-IP association from the Auth Container and added it to the cookie mapping), they will then have their request forwarded to the proper container based on the IP in the mapping. The client's requests will no longer be modified in this case, and the server container and client will effectively be communicating directly with each other.

The only other behavior of the proxy occurs in the case of a logout request. This case is detected by the proxy when the user navigates to the logout page in IA through the web interface, resulting in a 'logout=1' GET parameter in their request. When a logout request is detected, the proxy will set the mapping for the client's demux cookie back to undefined, modify their request's path to the URL of the Auth Container's logout confirmation page, and forward the modified request to the Auth Container. From this point on, further requests will again be modified and sent to the Auth Container, as described in the second case above.

The proxy is written in C++ and compiled in the containerized environment on startup via docker-compose. It opens up the HTTPS port, 443, and uses public key cryptography via axTLS to encrypt and decrypt traffic sent between clients and the proxy. Proxy-container communication is done locally via the internal docker virtual network over unencrypted HTTP, as this network is inaccessible outside of the system. In the case that further security is desired, these communications could be upgraded to use HTTPS as well. Communication with the Auth Container is performed similarly, but the proxy will only accept Auth Container requests from the hard-coded IP reserved for the Auth Container. This communication is necessary so that the Auth Container can inform the proxy of new demux-cookie-container-IP mappings.

3.4.2 SuS Manager

The Single-use-Manager manager orchestrates the containers in our system and communicates with the Auth container. The entirety of this component is four files written in Python and compatible with Python 3+. Alongside these files, the SuS manager needs a file containing a list of users. This file allows the SuS manager to initialize containers for every user, keep track of who has permission to use the system, and generate files with SQL scrips for creating custom SQL users with appropriate permissions in MySQL and deleting them.

Besides adding a file with a list of users, there are two other requirements for running the SuS manager. The first of these is to edit the configuration file, one of the system's four python files. Users must edit variables the SuS manager needs to run including: Docker network information, database credentials and related table information for administrators, the SuS manager's IP address, IP addresses and ports on which it opens TCP sockets with the Auth and QR containers, and the number of containers to spawn on start-up. The other requirement for running the module is to download the netaddr python library using pip3. This module generates a list of available IP addresses for our containers.

A Docker network is necessary prior to starting the SuS manager so the containers can spawn on a network with no other devices or applications. To create a network we must specify the driver, subnet, and gateway options, set the network as attachable, and assign it an alias. The

alias of the network is written into the configuration file and the SuS manager references during container initialization. The IP address and subnet for the SuS manager itself is also required as the component uses socket communication and needs to know the where to opening them so it can communicate information with the Auth and QR containers.

The rest of the logic for the SuS manager is split between the remaining three files: monitor.py, container.py, and database.py. Monitor.py has the "main" function and thus starts the rest of the system. Container.py initializes the containers and monitors the pools of available IP addresses and active containers. Lastly, database.py is responsible for generating text files containing the SQL scripts that create and grant permissions for, as well as delete, our custom SQL users.

The first operation the SuS manager executes is making sure no other containers are running. It runs a shell command to query Docker for any currently running containers and if any exist, it executes another shell command to close them. This is done as a precaution should the SuS manager run into an error that it can not handle and hence closes prior to destroying all container instances. Because the containers initialize on the same subnet between runs, new containers have a chance of being initialized with an IP address in use by a container spawned from a previous instance of the SuS manager running. This process ensures this scenario does not happen.

After ensuring there are no running containers, monitor.py initializes the container manager object in container.py. The container manager creates a list of IP addresses for containers and randomize them. The manager object uses them afterwards when it creates a container-watcher object. The container-watcher object spawns in "dummy" containers with IP addresses the container-manager passes during initialization. Dummy containers are containers running Lighttpd images with IA that is not yet initialized with user-specific variables such as MySQL and IA user credentials in its configuration file. The container-watcher object creates its own thread that ensures an appropriate number of containers are available at all times. The number of containers the container-watcher initially creates however is held in the config file previously mentioned. Once the dummy containers are all ready the container-watcher pauses until it gets an interrupt to spawn more containers. The container manager now goes through the user list, a comma separated value (CSV) file, and reads in a username and its permission level row by row. For each user in the file, the container-watcher generates a random MySQL username and password and calls on database.py. There, the system writes SQL commands to text file that, when we execute in a MySQL server, create the account with the given username and grants it the appropriate commands the user is restricted to based on their permission level. Next, the container manager passes the randomly generated username and password of the MySQL user as well as the user's WPI username into one of the uninitialized containers through a shell script that directly modifies the internal IA configuration file inside the container. Once the script finishes, the SuS manager removes the container at that IP address from the list of uninitialized containers

and the container is ready for its respective user. To achieve this, monitor.py enters an endless loop of waiting for a connection over a TCP socket from the Auth container requesting the IP of a container for a particular user. When a connection establishes, the SuS manager creates a new thread to handle the communication with the Auth Container. The Auth container sends a username of an authenticated WPI user attempting to access an IA container. If the user is authorized to use the system and has a container ready, the SuS manager responds with the IP address of a container and closes the socket. Otherwise, the SuS manager sends back an error message that the Auth container is responsible for handling. This is typically the result of a user not being authorized to use the system.

The MySQL user that is written into each container has tailored SELECT, INSERT, UPDATE, and/or DELETE commands on every class database table depending on the user's permission level. These limitations are still fairly broad, however, and do not entirely restrict the user's queries on a row or column level, which is the ideal level of restriction. Thus, we plan to use a query restrictor to ensures these constraints, as we explain later in this Chapter. As previously mentioned, the configuration file also asks for the credentials to the root user for the MySQL database. The SuS manager passes these credentials into the container the system administrator will use to avoid creating another root MySQL user.

The SuS manager can be shut down by issuing the interrupt signal from the terminal. When the process is interrupted it enters "interrupted" mode and shuts down all of the IA containers, the Auth container, and the Proxy. The SuS manager prints a text file with commands to delete every MySQL user while making the users that we then execute in the MySQL database. By removing all of the containers and deleting the temporary MySQL users we ensure that all connections to the database are removed.

3.4.3 Auth Container

The Auth container is responsible for correctly routing users to the WPI Kerberos authentication system, ensuring authentication is successful, and communicating with the Proxy and SuS manager. The Auth container is composed of three PHP files.

As previously mentioned, the Auth container utilizes the authentication system from InstructAssist. We isolate the authentication mechanism from the Legacy IA code which allows us to assign each user a cookie that both authenticates them and allows our system to identify which container belongs to them. Removing the authentication component from IA was the only change in code we make to make our single-use-server system compatible with IA.

When a user navigates to our server, the Proxy checks for a cookie with a "demultiplexing" (demux) value. If a cookie is there, it checks to see if there is an association between the cookie

and an IP address of a container. If the cookie is not there, the proxy creates a 32 character alphanumeric value and routes the user to the Auth container, appending the value as a query argument to the redirect. In the Auth container, we create a cookie with the demux cookie value alongside a session variable, a session cookie storing the association between the demux value and the page the user is trying to access in their original request to the system, such as the class calendar. We then forward the user to the WPI Central Authentication System (CAS) where they enter their WPI user credentials.

The WPI CAS service has restrictions on what applications are able to access it in order to prevent misuse. Therefore, we have to route the call to CAS through legacy IA as it is one of CAS's approved applications and our system is not. The Legacy IA server redirects the user with the query parameters CAS assigns to the request back to our server. Despite an extra hop, the authentication works the same as it would with direct access to CAS.

When Legacy IA forwards the user back to our system the Proxy checks for the demux cookie and sees that a demux cookie is set for the user but there is no container IP address to demux value association. The Proxy again appends the demux cookie value as a query parameter and forwards the user to the Auth container. This time, the user goes to the CAS authentication helper page where we extract the query parameters. The query parameters include a username, the IP address of the CAS caller, the time, a nonce value, an HMAC value, and finally the demux cookie value. The Auth Container confirms the authentication as described in the "Understanding InstructAssist Authentication and Database Communication" section.

With the authentication complete, the system has to determine if the user is able to use our system. The Auth container cannot query the database directly so it opens a TCP socket with the SuS Manager and requests the IP address of a container assigned to the WPI username of the successfully authenticated user. If the SuS Manager responds with an error, the user is not able to use the system and is met with a web page notifying them that the system is only for certain classes and advising them to to contact our advisor if they are seeing the page as an error. Otherwise, the Auth container gets a response from the SuS manager with the IP address of that user's container. The Auth Container saves the address, and closes the socket with the SuS manager. It next opens a TCP socket with the Proxy so it can instruct it to route the connection with the demux value stored in the authenticated user's cookie to the IP address of their container, thus creating the Container IP to demux value association.

There are concerns regarding using a single value as a key to a container. It impacts the duration of cookie validity as a user could pose as another by copying one string. It would be unlikely to guess this value, however, as is it composed of 32 alphanumeric characters. That is, each character composing the value can be a number or lowercase character. Blindly guessing the correct value on the first try is a one in $1.53 * 10^{54}$ chance. Given these odds, we believe we

protect the user from a demux value forgery for the duration of the cookie's validity, which is currently set to be the duration of the session.

3.4.4 Query Restrictor

The Query Restrictor (QR) is a protective measure we implemented in order to further protect the MySQL database through restricting the responses sent back depending on the user sending the query. Externally, the QR is seen as simply an extension of the database and interacted with accordingly, but is actually a proxy between the web application and its associated MySQL database. Thus, when a containerized instance of IA makes a query to the database, it is instead sending the query to the QR, which returns a restricted version of the database's response. By having the QR as the intermediary in the transaction, we can further ensure that the container is not querying data on behalf of a malicious actor attempting a confused deputy attack.

One of the tasks the SuS Manager is responsible for when initializing containers is writing in database users into the IA configuration files which have limited access to tables in the database. However, this restriction is not strict enough to prevent a confused deputy attack by itself. One example that spotlights these security issues is the following: every container has access to SELECT from the table containing grades because every student has grades and should be able to view them. Despite the fact that this table contains extremely sensitive data, it is one that every container has the ability to query. We needed a way to protect this data as unauthorized access would, again, result in a FERPA violation. Leveraging a Query Restrictor proved to be the most ideal solution to such an issue.

When initially designing the QR, we explored two main options: a proxy acting as a database extension that IA would communicate with via socket communication or a man-in-the-middle program that would sniff packets being sent to the database and manipulate the packet's data if restriction was found to be needed. While both options would theoretically work in our project, we decided on moving forward with development on the proxy-based QR. Due to the similarities of the two, the decision to choose the proxy was primarily due to its likeness to the proxy, which was already constructed at this point in time. Similarly to the proxy, the QR is also written in Python and utilizes mysql.connector to connect to the IA MySQL database.

Fundamentally, the QR's primary function is as its name implies: to appropriately restrict the data returned in the MySQL responses based on the user. In doing so, adversaries are prevented from viewing data outside the scope of their permission level and accidental users are returned specific information associated with them rather than all data, including that of other users, within a table. This base functionally revolves around the usage of including a WHERE clause within the query. In MySQL, the clause "allows you to specify a search condition for the rows returned

by a query" [33]. However, to properly place the WHERE clause within the query in order to maintain the integrity of the MySQL query, the QR must correctly parse the query and insert the proper data.

When the QR receives a query, it must first determine whether restriction is needed to be performed or not on said query. Based on IAs' database interactions, the QR specifically inspects to see if the query is a SELECT statement, since SELECT statements are commonly used by IA to return data to the user. If the query does not contain SELECT, it does not add any restrictions as no data is returned to the user aside from a message signifying that the database action had been completed. After confirming that a query contains SELECT, the QR then checks if the table being queried is defined within a list of table where restriciton is needed. Some tables within IA lack fields that support user-based restriction, therefore a dictionary is present in the QR code linking the names of tables where restriction is possible with the fields used by that table for restriction, such as "userID" or "username." Throughout the majority of the tables in IA, a user's ID can be leveraged as the best method to restrict a table by, although the field names referring to one's ID can vary.

When the query passed both criteria, it is then parsed for key MySQL clauses. If a WHERE clause is already present, the QR simply appends the new restricted field value to it through the use of AND. If a WHERE clause is not found, the QR will then create its own WHERE clause and checks to see where it should be placed within the query. A tuple of MySQL words relating to clauses is used and, based on the order of clauses defined by MySQL syntax, the query is parsed until it reads a clause that traditionally follows WHERE or reaches the end of the query itself. In the first scenario, the query is split, and the new WHERE clause is added between the two substrings to form a new query, which is then passed on to the database. In the second scenario, the WHERE clause is appended to the end of the query and then sent to the database.

The QR determines which queries it modifies based on the user-specific data the SuS manager transfers over during the initialization process. The SuS manager passes the QR the user's username, id, and permission level. As a result, the QR can effectively restrict any restrictable table within IA using one of the three fields.

3.4.5 System Workflow Summary

When an unauthenticated user navigates to our system, the Auth Container receives the initial GET request via the Proxy and redirects the client to WPI's Central Authentication Server (CAS) to authenticate. Once the client authenticates, a redirect brings them back to our system, and the proxy sends them to the Auth Container again. The Auth Container confirms the authentication and requests a container from the SuS Manager by passing it the user's WPI username. The SuS

Manager receives the WPI username from the Auth Container and finds the IP address of the IA container instance for that user through its username-to-IP association map. When the SuS Manager finishes, it sends a response back to the Auth Container with the IP of the container. The Auth container then informs the Proxy about the container IP so that future requests from that client will be forwarded to that container and redirects the client back to the Proxy. From this point until the user's demux cookie expires, all traffic from the client will be sent directly to their IA container where they can use the web application as usual. Behind the scenes, the Query Restrictor is filtering data to ensure only the necessary information makes it to the container from the database. A complete overview of all system components and communications is provided in Appendix A in the form of a chart.

3.5 Implementation in Production Environment

Once all the components of the system are operating properly and we are able to replicate the system in a secondary environment following our playbook, we give our advisor instructions and code for installing the system in his production environment. There he recreates the system using our playbook and runs his own tests to make sure the system is indeed working and no steps of the configuration or components of the system expose vulnerabilities before running the system with the production database.

The SuS manager originally ran with direct access to the database. Security concerns were raised regarding giving a script access to execute privileged queries. Vulnerabilities, whether in our code or in packages we were using, could result in a myriad of different issues ranging from database slowness to data being exposed. Our advisor is responsible for ensuring the database remains stable for the other services relying on the data it stores. To mitigate these risks, we made minor changes to our SuS manager to have it read from the text file containing user information as mentioned in the previous section. This was the only change we made to prepare the system for production testing.

With this method we lost some functionality. With a direct database connection, the SuS manager can spawn and despawn containers dynamically as users are authenticated and the SuS manager can check the production database for authorized users. This functionality allows changes to the list of authorized users without having to restart the system with a new user CSV. Choosing to take this route of container orchestration would, however, result is a longer delay between authentication and access to a container for the users as the container creation process takes a few seconds but it would maximize resource usage for the system administrator as the containers are not all spawned and waiting at the same time.

By planning the modularity of the all of our system's components and by commenting inside

of each component explaining functionality, system administrators can make changes to revert the system to utilize a direct database easily.

3.6 Testing Containerized Server Implementation

We plan to use our system in an offering of Professor Craig Shue's CS3013 Operating Systems class. We expect that students may experience a minor delay by connecting to the containerized version of IA. This may be a possibility under heavy traffic should the Virtual Machine be running too many container instances consecutively.

Should students experience any kind of disruption connecting to or using IA with our system or decide to opt-out of the study entirely, they can use legacy IA which is running in tandem to our system. The legacy IA and the containerized version share a connection to the same database. This means that any work a user completes using our team's version will be accessible on the legacy server and vice versa, thus preventing data loss or syncing issues.

To collect logs from the production test, we run a TOP command that print system resource use to a file every five seconds. We filter the output of the command for processes related to docker and timestamp them. These logs include resource usage of the CPU and memory and how much of each is in use by Docker containers. The results from these tests are highlighted in the results section.

3.6.1 Launch Issues

Prior to testing the implementation with the classroom environment, we found a major issue preventing us from granting access to students. When our advisor ran the system, it appeared as though the MySQL daemon seized up for both our system and the sever hosting legacy IA, thus making it inoperable. After analyzing the incident, it was still unclear whether the issue was a fault of a bug in our code or whether it was an external problem.

Due to this uncertainty, we exercise caution by again running the system during a period of low student activity and only allow our team to interact with our system. In doing so, we hope to realize and fix any bugs we may have introduced during our original test. In addition, if the MySQL daemon seized up yet again, we could analyze the logs surrounding the incident to hopefully make more steps to discover the cause. Our goal is to test the system and ensure it would be stable enough to expand the amount of testers until we can allow access to the class, as originally planned.

Chapter 4

Results

In order to test the feasibility of the system, we ran a series of latency, stress tests, and security tests in order to analyze its effectiveness and to gather useful metrics. The latency and stress tests are accomplished by running a python script locally that simulates a predefined number of users navigating to different pages within their containers in varying versions of the environment. The code for such testing uses the Python Requests package to time how long it takes to receive a response from the web server. For each client we simulate, we create a separate thread so that requests are sent concurrently. We run three latency tests and two stress tests as well as two security-related test on our system.

4.1 Latency Testing

The first latency test is testing delay without a container by running Lighttpd on the machine hosting our project. This test acts as our benchmark for seeing how much latency containers and the Proxy add to the end-to-end delay. The second test is testing the delay with the web server in a container. The third and last latency test involves testing the web server in a container that can only be queried through the proxy to see the total end-to-end delay a user will experience. This testing helps us understand how long of a delay users using our system can expect and also help us understand which parts of the system cause the most delay. The Python program we use to measure end-to-end latency queries the web server and writes the time it takes to get a response for each query sent into a CSV file that we analyze.

The results of the three tests do depict a significant change in latency based on the the web servers configuration. We determine this based on the average round trip time, which is found by calculating the average load times for various IA web pages. The average round trip time we found after our first test where we use a non-containerized implementation of IA is 139 ms. The

results of the second test in which we run IA in a container results in an average round trip time of 391 ms. In the third test where we use the proxy to communicate with a containerized instance of IA the latency is an average of 535 ms. When analyzing the greatest individual round trip times respective to each test, there is a dramatic increase. The longest round trip time is 764 ms in the first test, 8,266 ms in the second test, and 10,834 ms in the third test. While the majority of the round trip page load times are fairly low, we found a select number of pages that result in large round trip times that alone contribute to the increasingly large averages. The containerization and the implementation of a proxy do significantly hamper the performance of the system. Based on these results and our own experience, we conclude that users should be able to interact with our implementation despite a longer delay than one from a tradition web server without debilitating performance issues.

4.2 Stress Testing

By running a plethora of stress tests upon our system, we hoped to mimic what would normally occur if this implementation were to be used in the real world. For our stress tests, we simulated one click per second for forty containers and then eighty containers for twenty minutes for each. The stress testing allows us to collect logs on the usage of six major components: the system's physical memory, virtual memory, CPU usage, and CPU load as well as the status of the containers and Docker. After testing, we compile the logs containing the test data, convert them into .csv files for easy viewing and manipulation, and finally utilize programs, such as Microsoft Excel, to create data visualizations.

4.2.1 Physical Memory

When analyzing the physical memory usage, we specifically focus on the amount of free memory, the amount of memory being used, and the amount allocated to buffers.

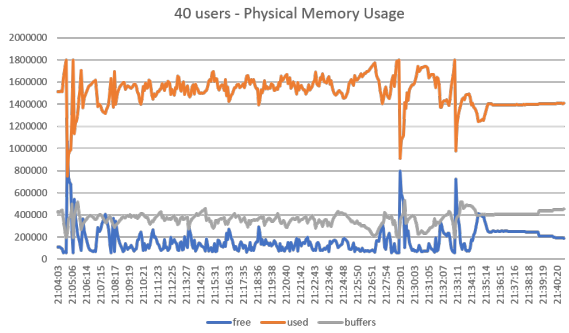


Fig. 1. Visualization of the physical memory results collected after simulating one click per second per container for twenty minutes on a set of forty test containers.

The effects of the stress test on the physical memory of the client, at least in the case of the forty user simulation, results in a fairly stable outcome as seen in Figure 1. While occasionally "spiking" down, causing the line depicting the amount of free memory to "spike" up in accordance, the line representing the amount of used memory stays mainly level throughout the test at 1,500,000. Similarly, the buffer memory also stays relatively stable at approximately 400,000.

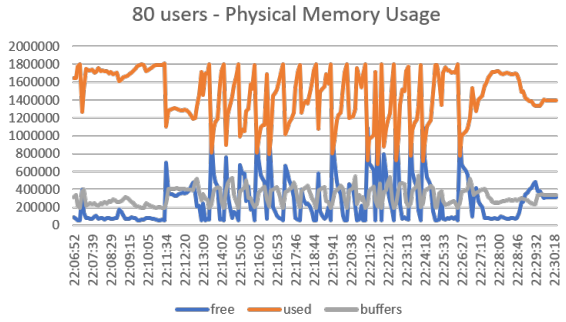


Fig. 2. Visualization of the physical memory results collected after simulating one click per second per container for twenty minutes on a set of eighty test containers.

As Figure 2 indicates, the eighty user result, when compared to the forty user results, are much more erratic. The amount of memory used dips significantly from 1,800,000 to 800,000 to the point where the amount of free memory becomes equal to it a multitude of times. The amount of buffer memory does seem to variate more than in the forty user test as well, but the valleys and peaks seem to still average out around 400,000.

4.2.2 Virtual Memory

Like with the physical memory analysis, we are analyzing the virtual memory graphs in order to look for trends that develop over time involving the amount of used and cached memory and the resulting amount of free memory left.

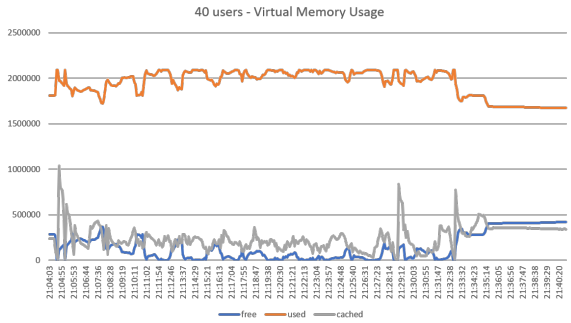


Fig. 3. Visualization of the virtual memory results collected after simulating one click per second per container for twenty minutes on a set of forty test containers.

In the case of the 40 user stress test as seen in Figure 3, the amount of memory used seems to remain level around 2,000,000 with a slight increase halfway through the test and a slight decrease closer to the end. The cached memory is shown to waver throughout the test, peaking as high as 1,000,000 near the beginning of the test and then decreasing to around 250,000 for the bulk of the remainder excluding a few other spikes later on. Interestingly, after 21:35:14, the used and cached amounts level out and remain constant until the end of the test.

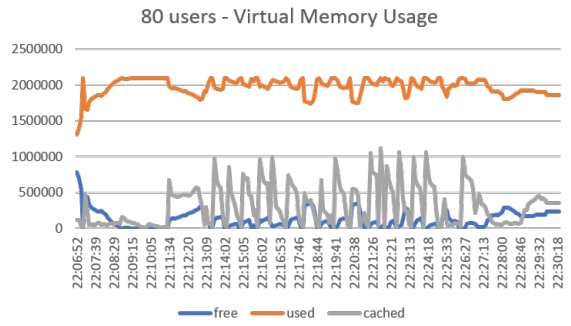


Fig. 4. Visualization of the virtual memory results collected after simulating one click per second per container for twenty minutes on a set of eighty test containers.

The effect of the eighty user stress test on the virtual memory seems to result in a more erratic version of the forty user version. As depicted in Figure 4, the amount of memory used wavers closer and closer to 1,500,000 and the spikes of the cached memory present in the alternative test is even more pronounced in this one, with the memory usage peaking at a little over 1,000,000.

4.2.3 CPU Usage

The CPU usage is an important metric to keep track of when it comes to developing applications. In order for our implementation to be a success in the eyes of the users, it would have to obviously be fast and responsive and looking at and analyzing the CPU usage can provide an insight into determining just that.

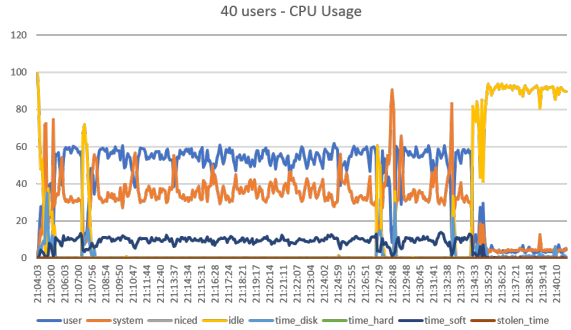


Fig. 5. Visualization of the CPU usage results collected after simulating one click per second per container for twenty minutes on a set of forty test containers.

When looking at the results gathered from the forty user test in regards to CPU usage as seen in Figure 5, three main values stand out: the user level CPU usage, the system level CPU usage, and the time spent servicing software interrupts, shown by time_soft. System level CPU usage and user level CPU usage refer to the percentage of CPU utilization that occurred at the kernel level and user level respectively. The graphs shows that the user level CPU usage barely under 60%. Underneath is the system level CPU usage clocking in it approximately 38% and periodically, around every three minutes, spiking, causing a drop in user level CPU usage. Finally, the time_soft takes up the little remaining CPU usage leftover and stays relatively stable at around 10%. From 21:34:33 to the end however, the percentage of time the CPU was idle skyrockets and all other usage outlets drastically decrease as a result.

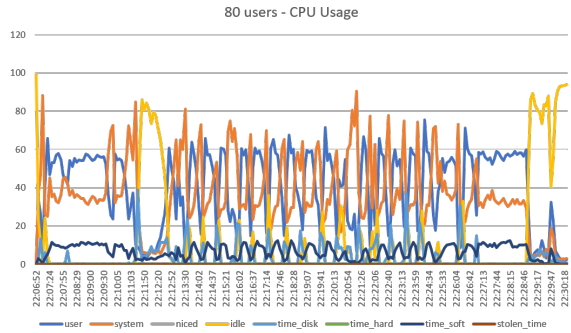


Fig. 6. Visualization of the CPU usage results collected after simulating one click per second per container for twenty minutes on a set of eighty test containers.

Like most of the other visualizations mentioned thus far, the eighty user version of the CPU usage shown in Figure 6 is similar to the forty person version aside from more exaggerated features. The user and system CPU usage fluctuate tremendously and exchange percentages with one another around every thirty seconds. The idle CPU usage still overcomes all others at the end of the test however.

4.2.4 CPU Load

CPU loads were another significant metric that was tracked throughout our stress testing process. The primary fields for analysis are the one, five, and fifteen minute averages. When referring to the averages, we specifically mean the number of processes awaiting or being executed within a given period of time.

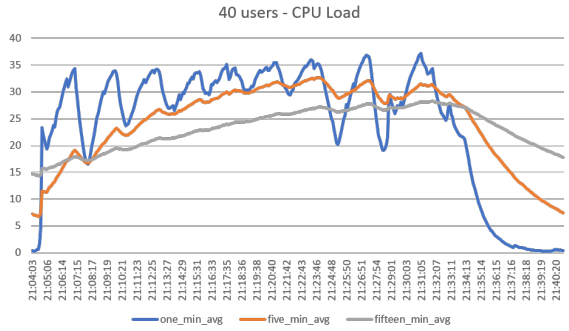


Fig. 7. Visualization of the CPU load results collected after simulating one click per second per container for twenty minutes on a set of forty test containers.

For the forty user stress test results depicted in Figure 7, the one minute average first increases, from zero to twenty-four, and then continues to form in a wave-like pattern varying from higher values such as thirty-five to low values like twenty while the size of the waves decreases and the values generally increase until 21:25:46. Similarly, the five minute and fifteen minute results also reflect this by increasing to around that same point in time, although lacking a wave-like composition. From 21:33:11 on wards, the averages drastically decrease.

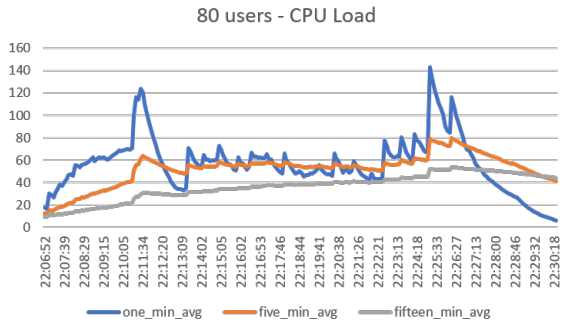
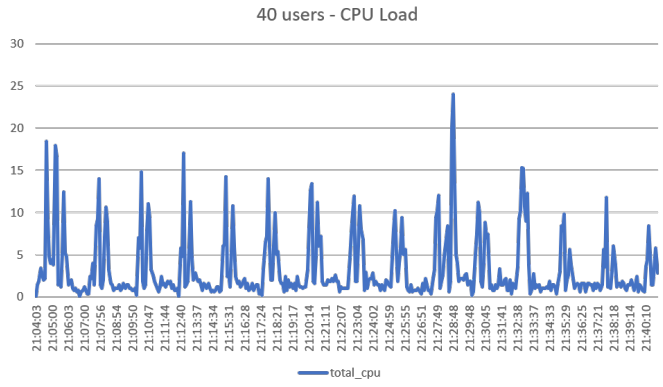


Fig. 8. Visualization of the CPU load results collected after simulating one click per second per container for twenty minutes on a set of eighty test containers.

Interestingly, the results visualized in Figure 8 for the eighty user stress test are quite different that the results shown for the forty user test. While still erratically wavy, the one minute average has two large spikes at 22:11:34 and 22:25:33, where it reaches 120 and 140 respectively. Outside of those two isolated incidents, the one minute average wavers around 60. The five minute and fifteen minute averages increase overall but experience smaller spikes where the one minute average experiences the larger ones already mentioned.

4.2.5 Containers

Since containers are fundamental to our system, in order to analyze the containers, we take note of the stress tests' effects on several features in order to synthesize a more comprehensive analysis. Specifically, we analyzed the containers' CPU load and memory usage, virtual memory usage, and resident and shared memory usage.



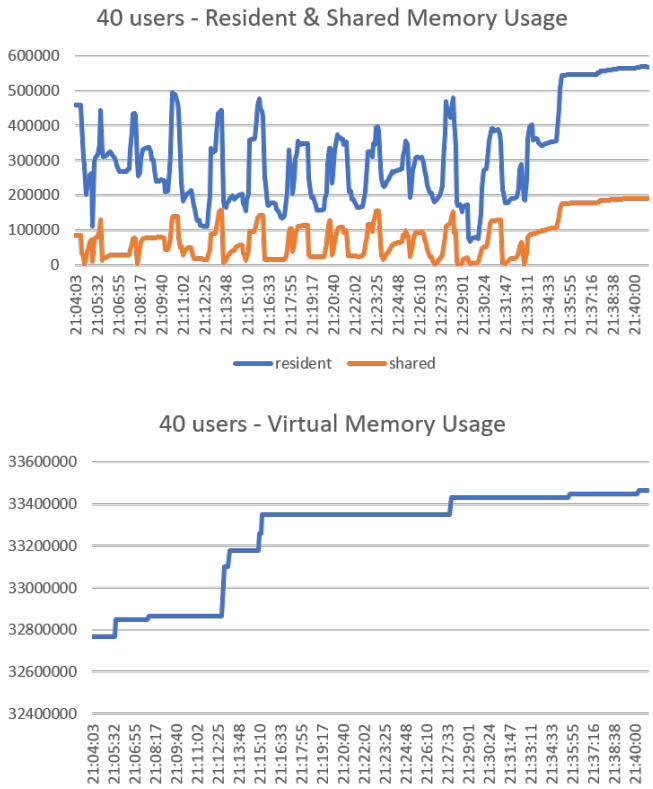


Fig. 9. Visualization of the container results collected after simulating one click per second per container for twenty minutes on a set of forty test containers.

For the stress test of forty users shown in Figure 9, the amount of total memory used closely aligns with the amount used by the total CPU. The total CPU ranges from twenty to two percent while the total memory used normally around 3 percent greater. The two match each others rises and falls until the end of the stress test, where the total memory rises and plateaus around twenty-five percent.Similarly, the resident and shared memory usage reflect each other but the shared memory usage lies, around 50,000, significantly lower than the resident memory usage, at 300,000. Lastly, the virtual memory usage simply incrementally climbs as the stress test progresses, starting beneath 32,800,000, and ending above 33,400,000.

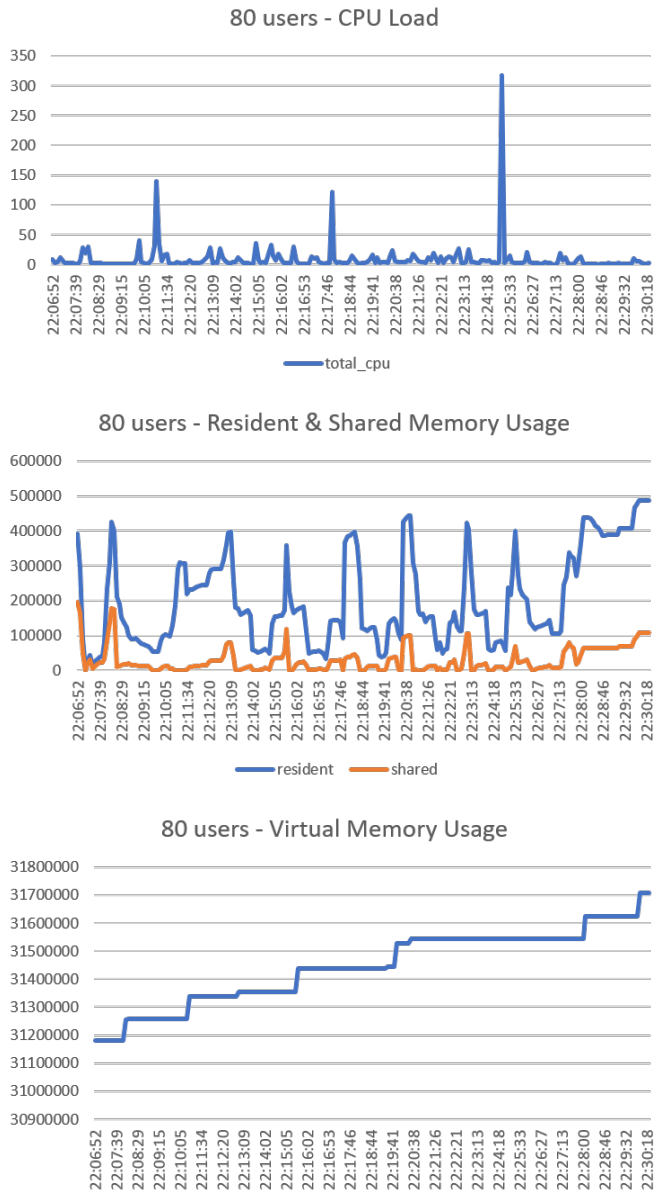


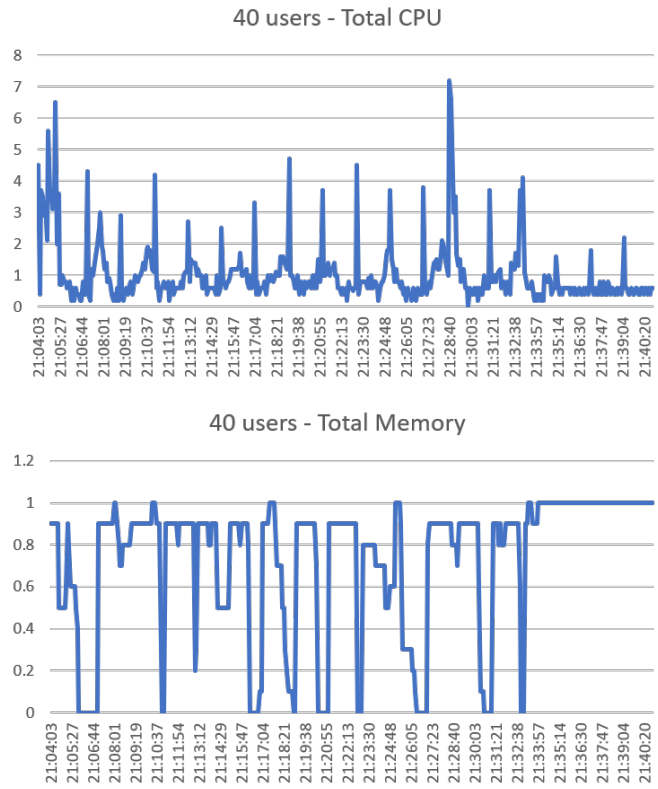
Fig. 10. Visualization of the container results collected after simulating one click per second per container for twenty minutes on a set of eighty test containers.

Looking at the eighty user test results shown in Figure 10, the virtual memory usage and resident and shared memory usage mimic what was seen in the forty user test. However, there is a large difference shown in the CPU load and memory usage data. The total CPU usage spikes three times, at around 22:11:34, 22:17:46, and 22:25:33, and peaks at approximately 150, 125, and 315 respectively. Since seemingly no significant action was taken during the test to our

knowledge and because the data outside those three spikes seems similar to that shown in the forty user test, we theorize that these spikes may be inaccurate outliers, potentially spawned from faulty readings or unrelated processes.

4.2.6 Docker

Another vital component behind our system is the Docker process itself. We recorded various metrics but focused on analyzing the difference between the total CPU, the resident energy usage, and total memory metrics between the two stress tests.



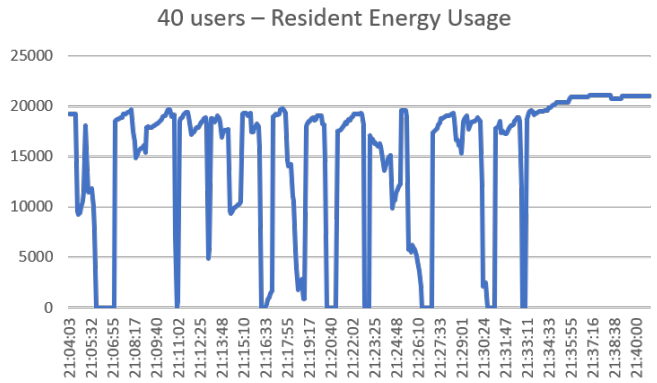
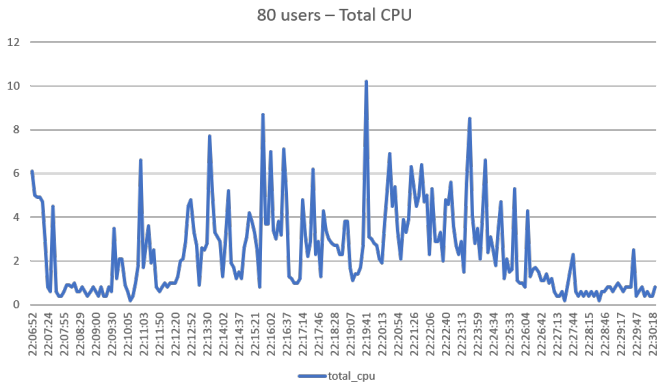


Fig. 11. Visualization of the Docker results collected after simulating one click per second per container for twenty minutes on a set of forty test containers.

In the case of the forty user stress test, Figure 11 shows the resident energy usage and total memory usage following a similar trend. The resident energy usage erratically peaks around 20,000 and falls to zero throughout the test with the total memory follows a similar variation in its own usage. The total CPU's data quickly rises and falls, primarily remaining below one but occasionally spiking to values such as four or even seven every couple seconds.



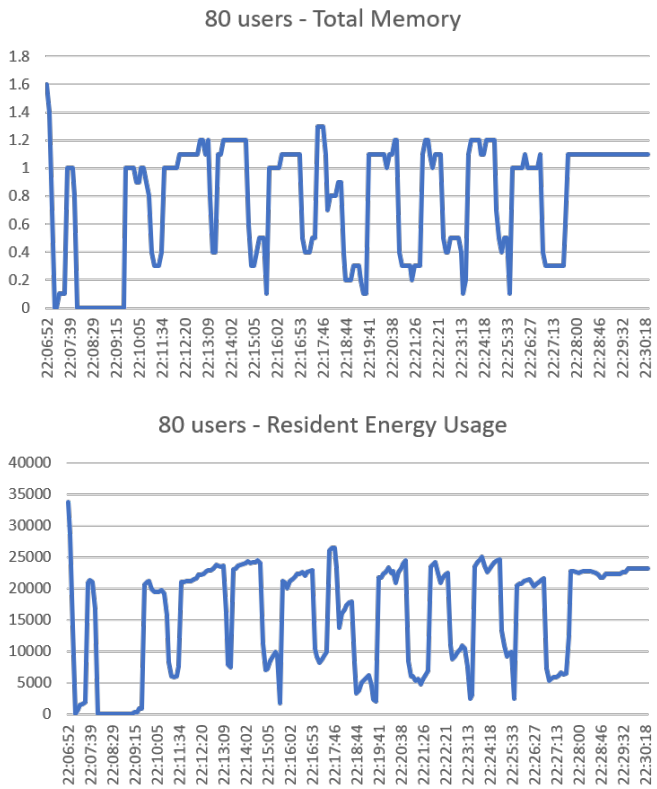


Fig. 12. Visualization of the Docker results collected after simulating one click per second per container for twenty minutes on a set of eighty test containers.

The results of the eighty user test remains largely unchanged aside from the total CPU data, as indicated in Figure 12. The total memory and resident energy usage still reflect one another and remain consistent with that in the forty user test. However, the CPU values experience a significant increase, going from peak values of four and seven to values closer to eight and ten. Despite these increased values and an overall slight increase to all CPU values in the middle of the test, the values still frequently appear at values around one.

4.3 Security Testing

Since this single-use containerized implementation's primary purpose revolves around heightening security, ensuring that it fulfills said purpose was an essential aspect of our testing procedure. We accomplished this is two ways: testing the effectiveness of a Secure Socket Shell (SSH) attack on the system and testing our QR to verify that it was correctly restricting MySQL queries sent to it.

4.3.1 Testing against SSH Attacks

The SSH protocol enables users to transfer data over networks, but can also be leveraged maliciously to gain access to protected user accounts and information. Potentially, such attacks can even result in sites becoming entirely compromised. However, our system provides a solution by mitigating SSH attacks that could normally shutdown entire sites by restricting users to their own individual container. Therefore, only the container the adversary is assigned would be affected and the rest would still maintain functionality. In order to see if our system worked as planned, we checked the verify that it effectively mitigates SSH attacks performed on the containerized IA.

To run the test, we simulated an adversary connecting via SSH from within our system; in doing so, we assume that an adversary would be able to connect similarly from the public internet if the containerized IA was available publicly. The "adversary" was able to successfully SSH in and compromised the inside of their container. However, when testing the other containers running while the attack was performed, we found that, outside of the adversary's container, no other containers were affected. This coincides with the planned functionality of our implementation and thus proved the implementation's SSH security to be successful in mitigating attacks.

4.3.2 Testing the Query Restrictor

Databases, which can hold large amounts of valuable and sensitive data, can be high-priority targets for adversaries to attack. Due to the fact that our system frequently interacts with a database of its own, we decided that running security tests on the QR was necessary and would provide a useful insight into any flaws adversaries could exploit to maliciously again access.

While developing the QR and before it was connected IA, a varied collection of MySQL queries were performed on it to not only guarantee that it would correctly process queries sent to it, but to also help defend against malformed queries and attacks. Due to the time given to complete the system and because IA primarily uses simple queries, more complex MySQL queries, such as nested queries, were not involved in testing. While we were able to develop the QR to the point where it was restricting the majority of queries appropriately, when linking the QR to IA, we experienced an issue. IA's communication with the MySQL database checks the initial response to confirm that it is indeed reaching out a MySQL database. Unfortunately, we were unable to adapt the QR to conform with this process and thus were unable to completely test the QR when integrated into the system. While the testing performed in development looks promising, it is currently unknown how secure it is when running alongside all the other components.

Chapter 5

Related Work

Implementing a barrier between a web server and the operating system with back-end resources while implementing per-user isolation is a concept that has been conceptualized and implemented before using different technologies.

5.0.1 Single-Use Server Patent

In 2010, a patent was issued for a web system infrastructure with a single-use server mechanic that achieves per-process isolation. The web server is comprised of a series of virtual servers to service and then hand off requests. Each of these servers shuts down and rebuilds after it completes its task. When a request is sent to the server, it is first processed by a public-facing "incoming server". That server passes the request it receives to a virtual incoming client request server. Once it passes that data, the incoming server shuts down and rebuilds. The incoming client request server queries the request, retrieves the application/information, and passes it to an outbound server before also rebuilding. Lastly, the outbound server sends information back to the client and rebuilds. Every time the public facing incoming server rebuilds, it uses a slightly different configuration to confuse attackers. Between a request being sent and the server rebuilding, there are only a few seconds where it can be corrupted [34].

5.0.2 SuS with Virtual Machines

The patented system described above does not, however, incorporate per-user isolation. A more direct competitor to our implementation would be using VM's to comprise the virtualization barrier. As we have previously mentioned, the inclusion of an operating system and kernel makes VMs slow and resource-intensive. A single-use server system using VMs has been attempted and analyzed in the master's thesis *Leveraging Software-Defined Networking and Virtualization for*

a One-to-One Client-Server Model by Curtis R. Taylor. Through the use of software-defined networking and virtual machine virtualization, Taylor created a one-to-one client-server model resulting in a paradigm capable of stopping any ongoing attacks, preventing persistent attacks, and analyzing an attack to find how the attacker tried to compromise a virtual server [35].

5.0.3 CLAMP Stack

A similar approach to Taylor's has been tested using a LAMP stack which is an acronym for its components, a Linux operating system, Apache web server, MySQL database, and a back-end written in PHP. In 2009, a group of researchers added confidentiality through virtualization to the stack, dubbing it the CLAMP stack. The research team extracted and isolated the authentication logic and data access control logic from the web application. The system's dispatcher was used to check the SSL session ID for applications using HTTPS or a cryptographic session cookie to determine the correct web stack for each authenticated request. If neither is mapped, the user is routed to complete the authentication process. Once the user is authenticated they have an instance of the web stack initialized, a process that takes roughly 36 seconds. We believe that with our implementation using containers, we can drastically reduce this latency providing a quicker transition from authentication to a server instance. Ultimately, our project looks to build on this idea using different technologies.

5.0.4 Radiatus

Lastly, Radiatus is a project sharing many commonalities to our work. It uses a "User Router" to check for authentication cookies and forwards connections to their appropriate containers. If a user is not authenticated, the user is forwarded to a secure authentication entity. Upon successful authentication the user is routed to their own Docker container. The image in the container is Radiatus, a Node.js application that can be modified to run some web application. Data is scanned by a storage guards to make sure it is relevant to the user before making it into the container as well. They key differences between our implementations is that Radiatus is limited to Node.js, whereas our goal is to create a Single-use-Server system that works with any web server. The Radiatus implementation was evaluated monetarily and the researchers concluded it cost an additional $0.0008 per year per user using this system [36].

Chapter 6

Future Work

6.1 Production Testing

While we were able to get detailed results for lab-based testing of our system, the issues encountered when attempting to run it in the production environment were not able to be fixed within the scope of this project. The main prohibitive issue encountered was that some site content was not being correctly forwarded to the client, resulting in some images and documents not loading properly or at all. We were unable to reproduce these issues in our testing environment, and thus unable to properly investigate what may have been going wrong, as we did not have ready access to develop in the secure production environment. This means that we could not get results detailing the performance of the system when scaling to support a large number of users. We were thus also unable to determine whether the larger scale of the production environment was the cause of these issues, or whether it was an environment-related difference we had not anticipated. Further investigation on the performance of such an architecture at scale is therefore necessary for evaluating its feasibility in production.

6.2 A More Generalized Implementation

This project's goal was to evaluate the feasibility and effectiveness of our proposed new one-to-one client-server architecture, and we did so with one application, WPI's InstructAssist. However, there is potential for this system to be implemented for a wide variety of other applications.

Conceivably, any authentication-requiring, web-based application making use of a server and a database could be converted to, and benefit from, this architecture. While other projects, such as the aforementioned Radiatus, require significant changes to all aspects of the original application (and have limitations on the type of application used), our model only requires the

successful containerization of the existing application, and the isolation of authentication into a separate container. While our project is built specifically for the InstructAssist application, a more general version of the Proxy, SuS Manager, and Query Restrictor could be written to handle any containerizable application. Such a system could have configuration options to allow it to adapt to application-specific differences. For example, the logout functionality of the proxy, currently specific to InstructAssist, could be made configurable and able to be set on a per-application basis. One could imagine a system as robust and configurable as other, less specialized, container orchestration systems such as Docker Swarm and Kubernetes. This remains to be explored.

6.3 Working with Distributed Systems

This project worked on a relatively small-scale web application, for which a centralized single webserver and database on one VM was enough to handle the maximum load it could expect in normal operation. Large-scale production web service architectures, however, are often distributed, utilizing numerous physical and virtual hosts needed to scale to massive numbers of clients. While our implementation made use of a single SuS manager, proxy, and Auth Container, it could certainly be packaged within a larger container orchestrator, such as Docker Swarm or Kubernetes. Within such a framework, there could be multiple proxy, Auth Container, and Query Restrictor instances, with load balancing done across them. The Proxies and Auth Containers would likely each need some sort of database to synchronize state such as cookie mappings, authentication status, and other details, and the SuS manager would need to be modified to interface with the container orchestration software, but the architecture lends itself well to being adapted for this use.

6.4 Further Use of Docker Features

Two features of the Docker Container Engine which we mentioned, but which are not utilized in our implementation, are resource usage and health monitoring.

Docker can put CPU and RAM usage limitations on containers, which can prevent malicious use of the container to hog resources and slow down the system for other users. This can also be used to ensure that each user's container gets a fair amount of resource usage when at peak use times for the system. Additionally, resource usage stats can be gathered live, meaning that an additional feature could be added to restart containers which are using too many resources, as this may indicate it has been hijacked for malicious purposes.

Container health monitoring uses user-defined health checks to get a measure of whether the container is functioning properly. As health checks can be any arbitrary command run on the

container, one could set them up to ensure that the server is running and functioning properly, and/or that no processes are running that should not be. This can help limit the capabilities of a user hijacking a container for malicious use, and even further limit the effectiveness of persistent attacks, as such an attack could be detected and cleared with a container restart with well-written health checks.

Chapter 7

Conclusion

In this paper, we describe our container-based one-to-one client-server model as well as outline the process of how to implement such a model for added web application security. The use of virtualization in the form of single-use Docker containers bolsters the model's security capabilities, especially against privilege escalation attacks and persistent malware. A Query Restrictor is also included to provide further protection to connected databases to ensure that any malicious, malformed, or unauthorized commands sent to said databases are ignored. Ultimately, these protections effectively form an adaptable model that can be altered to defend a variety of web applications against privilege escalation attacks, denial of service attacks, and data theft attacks.

While we were unable to test our model within a more realistic context, such as having students within a computer science class utilize it as part of their day-to-day lives, our IA implementation shows promise based on the data collected from stress testing. While more definitive testing should be made, the metrics collected have shown that the system has the potential to work in a production-level environment. However, due to issues around the Proxy and QR implementation, such metrics do not account for some delay we predict to exist in the model once it is fully constructed and running.

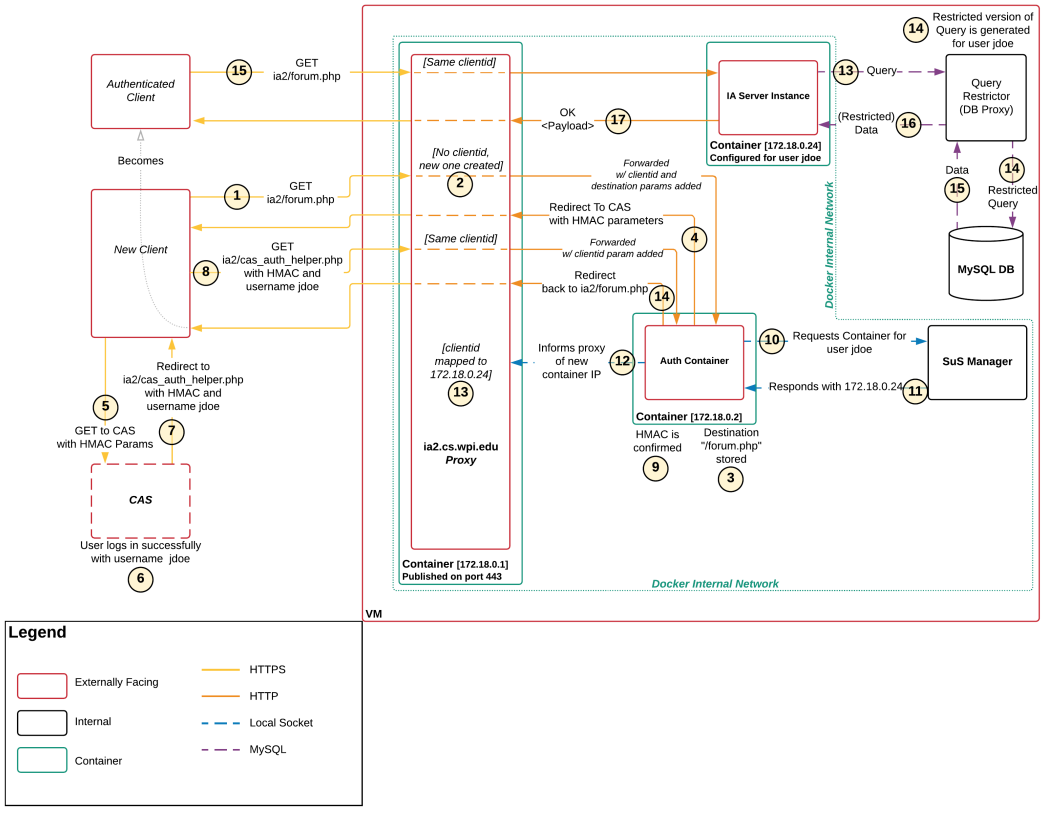
There are also many advancements that were made throughout the development of the project that can be more improved upon by future work. As mentioned, some opportunities for additional improvements lie within further leveraging the functionality of Docker containers, with two examples being setting CPU and RAM limitations and adding container health monitoring. Other potential improvements could be expanding upon the logic within the QR to handle more complex queries and to improve upon its filtering capabilities or changing the QR from acting as a database proxy to instead act as a middleman between users' containers and the database in order to manipulate the packets being sent based on the restrictions set.

Leveraging virtualization to create a one-to-one client-server model expands upon the its

cybersecurity potential and makes a swath of new protective measures available for vulnerable web applications. The fundamental components described within this paper act as an adaptable model which can be implemented to work with any web application and provide protection against often costly attacks which are all too common throughout the web today. Further work in this topic can yield promising results in the field of cybersecurity and could ultimately change the way web applications are developed in the near future.

Appendix A

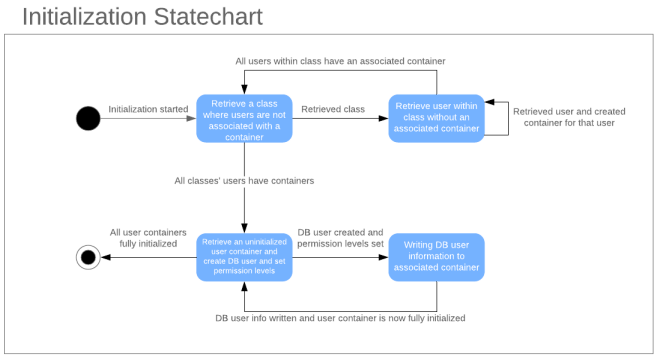
System Architecture Diagram



An overview of the new architecture, and the handling of a user's request when navigating to ia2.cs.wpi.edu/forum.php, going through authentication in the process.

Appendix B

State Chart Diagram



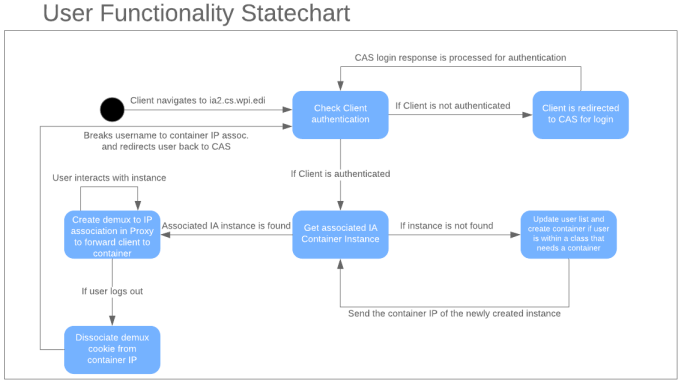


Diagram showing the states and actions taken during the initialization process and the standard user functionality.

Bibliography

[1] N. Hardy, "The confused deputy (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, pp. 36–38, 1994.

[2] M. Roser, H. Ritchie, and E. Ortiz-Ospina, "Internet," *Our World in Data*, July 2015.

[3] "The world's most valuable resource is no longer oil, but data, The world's most valuable resource is no longer oil, but data," *The Economist*.

[4] "Hackers Attack Every 39 Seconds."

[5] S. Smith, "Cybercrime will cost businesses over $2 trillion by 2019," *Hacktivism Professionalising and Going After Bigger Targets*, May 2015.

[6] "Privileged credential abuse is involved in 74% of data breaches," Feb. 2019.

[7] "For enterprises, malware is the most expensive type of attack," Mar. 2019.

[8] "What are Containers?," Nov. 2018.

[9] "What is virtualization?."

[10] T. Bui, "Analysis of docker security," tech. rep., Aalto University School of Science, Jan. 2015.

[11] "About Docker - Management & History."

[12] "Free Infographic: A history of #containerization technology!," Dec. 2016.

[13] E. Carter, "Sysdig 2019 container usage report: New kubernetes and security insights," Oct 2019.

[14] E. Carter, “2018 docker usage report.,” May 2018.

[15] “Docker Hub.”

[16] “Swarm mode overview,” Dec. 2019.

[17] “Docker Community Edition or Docker Enterprise Edition - Everything You Need to Know,” Dec. 2018.

[18] R. Chandramouli, “Security assurance requirements for linux application container deployments,” Nov 2018.

[19] Z. Jian and L. Chen, “A defense method against docker escape attack,” Mar 2017.

[20] J. R. V. Winkler, *Securing the Cloud: Cloud Computer Security Techniques and Tactics*. Syngress Publishing, 2011.

[21] V. Hu, D. Ferraiolo, and R. Kuhn, “Assessment of access control systems,” Sep 2006.

[22] R. Shu, X. Gu, and W. Enck, “A study of security vulnerabilities on docker hub,” Mar 2017.

[23] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi, Dec 2015.

[24] “Main page,” Nov 2017.

[25] T. Spring, “Attack uses docker containers to hide, persist, plant malware,” Jul 2017.

[26] T. Garfinkel and A. Warfield, “What virtualization can do for security,” Dec 2007.

[27] “Windows Sandbox,” Dec. 2018.

[28] “Bromium.”

[29] “Qubes OS: A reasonably secure operating system.”

[30] A. Blankstein and M. Freedman, “Automating isolation and least privilege in web services,” pp. 133–148, 05 2014.

[31] R. Chandrashekhar, M. Mardithaya, S. Thilagam, and D. Saha, “Sql injection attack mechanisms and prevention techniques,” in *Advanced Computing, Networking and Security* (P. S. Thilagam, A. R. Pais, K. Chandrasekaran, and N. Balakrishnan, eds.), (Berlin, Heidelberg), pp. 524–533, Springer Berlin Heidelberg, 2012.

[32] O. Nakar and J. Azaria, “Sql injection attacks: So old, but still so relevant. here’s why (charts): Imperva,” Sep 2019.

[33] “Mysql where.”

[34] “Single use server system,” May 2010.

[35] C. Taylor, “Leveraging Software-Defined Networking and Virtualization for a One-to-One Client-Server Model,” *Masters Theses (All Theses, All Years)*, Apr. 2014.

[36] R. Cheng, W. Scott, P. Ellenbogen, J. Howell, F. Roesner, A. Krishnamurthy, and T. Anderson, “Radiatus: a shared-nothing server-side web architecture,” pp. 237–250, 10 2016.