Preliminary Design Document

Dustin Allen Daniel Armas Rithik Dhakshnamoorthy
Jacob Plotz Luke Samuel Sandoval Jereme Saunders

October 2025

Contents

1 Executive Summary

Imagine you have an epiphany: you need to practice SQL injection or test a newly learned web-security technique, and you need an intentionally vulnerable web application right now. For most students and web-security practitioners, the next hour is spent searching through GitHub, cloning repositories, troubleshooting dependency errors, configuring Dockerfiles, and resolving version mismatches before any real security testing can begin. The setup process becomes the main barrier, not the learning itself.

Our Web App Deployment Tool (WADT) project addresses this problem by providing a streamlined platform for deploying intentionally and unintentionally vulnerable web applications with minimal effort. WADT offers a unified toolchain that removes the complexity typically associated with configuring, running, and managing these applications. Instead of navigating obscure documentation or manually assembling Docker environments, users can select a target application and launch it through a standardized, reliable workflow.

WADT is built on a robust open-source stack designed for extensibility, reproducibility, and ease of maintenance. Applications are containerized and orchestrated using Docker, allowing each vulnerable web app to run in a consistent sandboxed environment. This ensures that students, instructors, or security professionals can expect consistent and predictable behavior across deployments and platforms. The technical architecture supports a modular catalog of apps, enabling new vulnerable applications to be added without redesigning the system.

The objectives of this project are threefold. First, WADT seeks to reduce friction in cybersecurity education by allowing users to immediately interact with target applications

rather than waste time on environment setup. Second, it aims to create a scalable base where an expanding library of intentionally vulnerable applications can be deployed through a common interface. Third, the project emphasizes security, reliability, and user experience. These are critical aspects when dealing with tools meant to support penetration testing and vulnerability analysis.

The technical approach integrates a Django-based backend for orchestration logic and user-facing functionality, a React frontend for intuitive interaction, and Docker-driven infrastructure for application deployment. This layered design ensures that WADT remains maintainable, testable, and capable of evolving as the catalog of applications grows. Each component of the system is selected to balance performance, usability, and long-term sustainability.

In summary, WADT transforms the tedious process of launching vulnerable web applications into a fast, accessible, and standardized experience. By removing setup barriers, the platform empowers learners and practitioners to focus on what truly matters: developing hands-on skills in real-world web security.

2 Project Description

2.1 Overview

The Web App Deployment Tooling (WADT) project allows students and instructors to browse a catalog of ready-to-deploy vulnerable webapps and manage them through a graphical interface. WADT is the first step in the learning pipeline that ensures students and instructors are on the same page in minimal time. Their efforts can shift to investigating the app itself, while the container-based architecture of WADT ensures they have reproducible sandboxed builds available at the click of a button.

2.2 The Setup Timesink Problem

Cybersecurity students want to practice exploiting and patching vulnerable webapps. When they sit down to practice, their time is eaten up by installation and setup before they can start hacking.

Cybersecurity instructors want to provide labs for their students. These labs require research, setup, installation, and testing. The instructor must take the time to guide each student through the same prerequisite steps before the real learning can begin.

Both parties are capable of following a list of instructions to set up a target webapp. These skills are not the desired target of practice. However, requiring setup work on a

per-lab basis means wasting learning time.

2.3 Project Goals and Objectives

WADT aims to eliminate redundant setup work by providing a standardized environment where vulnerable web apps can be deployed instantly in isolated containers. This ensures reproducibility, simplifies lab setup, and allows both instructors and students to focus on the learning objectives.

3 Personal Motivations

3.1 Dusitn Allen

3.2 Daniel Armas

3.3 Rithik Dhakshnamoorthy

3.4 Jacob Plotz

3.5 Luke Samuel Sandoval

I chose to take on the challenge of Dockerizing our vulnerable web applications because containerization has become a foundational skill in modern software engineering and cybersecurity. As I approach graduation and prepare to enter industry, I want hands-on experience with the same tooling used to deploy, isolate, and scale real-world systems. This project offers a direct way to build those skills while contributing something genuinely useful to students who need reliable practice environments.

There is a clear need for this work. Today, students often spend more time configuring machines, troubleshooting dependency issues, and reconciling outdated instructions than actually practicing security concepts. By packaging intentionally vulnerable applications into clean, reproducible containers, we can remove the setup tax that slows down learning and give students an environment where they can immediately focus on exploitation, testing, and experimentation.

I am also motivated by the opportunity to apply container orchestration techniques in a realistic, production-minded setting. The project's structure and access to an AWS instance provide an environment that balances experimentation with real operational constraints. This allows me to explore scalability, automation, and reliability in a way that mirrors the workflows found in industry platforms.

My technical vision includes building a curated catalog of vulnerable applications, each defined with a lightweight deployment manifest that simplifies onboarding new apps. By using Python bindings and the Docker Engine REST API, the system will expose one-click lifecycle actions such as deploy, restart, stop, reset, and scheduled expiration after 24 hours. Overall, we will implement structured logging for container events to support auditing and instructor oversight, as well as a frontend UI that makes these operations intuitive.

The sponsor's long-term vision of as enabling network configuration from the UI and integrating vulnerability checks align with my interest in creating tooling that is both educational and operationally sound. Longer term, I hope to incorporate role-based access controls, per-session launch tokens, and a small registry of approved base images to maintain reproducibility and security. Ultimately, my motivation is to build a system where a first-time user can launch a vulnerable web application in seconds without wrestling with configuration, while I gain practical experience with the technologies and workflows I expect to use in my career.

3.6 Jereme Saunders

One of my primary goals heading into senior design was to push myself into areas where I had less experience and to expose myself to technologies and practices that I knew would be valuable in my future career. I considered several projects that offered opportunities to learn new skills, but this one stood out. It was a brand-new project with plenty of room for creativity, and it presented multiple opportunities to work with unfamiliar technologies in a meaningful way. The focus on security and the chance to gain experience with Docker made it especially appealing and directly applicable to future software development work.

Although I had learned about security in various classes, I had very little practical experience with it being a central focus of a project. In most of my previous work, security often took a back seat to other priorities. This project, however, is inherently tied to security due to its nature as an educational tool designed around vulnerable web applications. That made it an ideal opportunity to gain first-hand experience in a field that is becoming increasingly important as people become more reliant on technology.

Another major factor in my decision was the involvement of Docker, as this was the only project pitch that was explicitly working with it. I had zero experience working with Docker, but I have heard of it and learned of the increasing popularity of it with development in a professional setting. This project felt like the perfect chance to become familiar with a technology that I will almost certainly encounter again after graduation.

I was also drawn to the idea behind the project itself. I liked the idea of a project that was

made by students for other students. It also helped that the sponsor was a recent graduate who had worked with other senior design teams. That familiarity with the process made the project feel approachable, and I felt much more comfortable committing to a project with a sponsor who understood both the technical and academic expectations. Choosing a senior design project is a long-term commitment, and knowing I would be working with someone experienced in the process made the decision to work on this project much easier.

4 System Pipeline Diagram

The following activity diagram presents the high-level overview for users using WADT. The diagram demonstrates the main interaction loop that enables users to move from authentication to hands-on practice

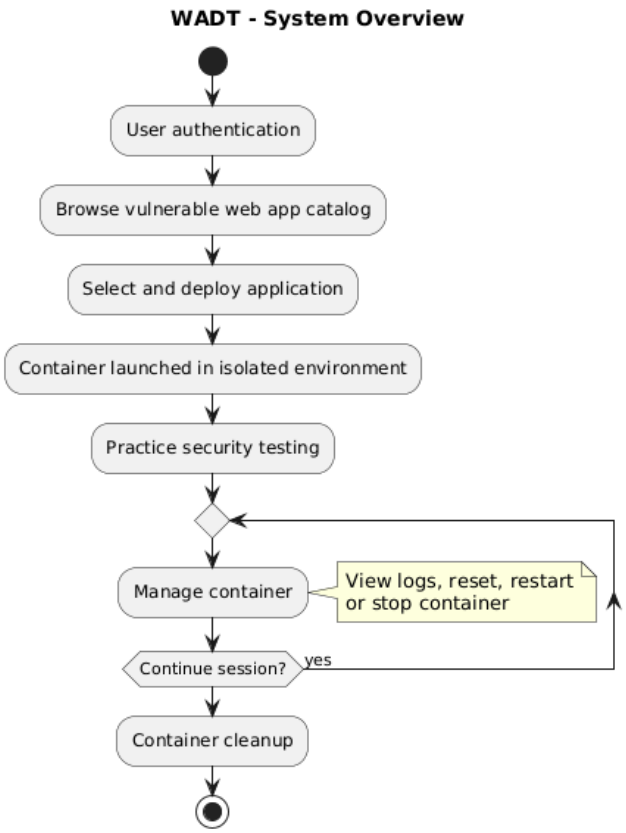

Figure 1: System Overview

This emphasizes WADT's primary goal of eliminating setup overhead while providing a streamlined and intuitive interface. Once a user has been authenticated, they can deploy any available applications in a single step and immediately begin testing. Container management

involves providing users the capability to reset or inspect containers as needed without restarting the entire process. When the session ends, either by a user manually stopping or automatic timeout after 24 hours, all resources are cleaned up to maintain system integrity.

More detailed activity diagrams will follow in the System Architecture section to provide more substantial details of the notable activities, namely for backend deployment of containers, interactions with containers, and container cleanup. These diagrams have been separated to provide clarity.

5 Minimum Viable Product

5.1 Scope and Limitations

WADT is intended to be used in an instructional setting as part of a lab. While all code developed for the project is free and open-source, the site and the AWS instance it runs on are not accessible to the general public. The team members and sponsor are the only individuals expected to access the site and instance while it is in early development.

Later in development, our users will access both the WADT site and the instance through their instructor similar to how we have accessed these two resources. We expect that users (assisted by instructors) are capable of reproducing our instructions for accessing the instance and authenticating to access the site when we provide these documents.

The current scope of WADT does not provision for multiple concurrent users at a time with containers managed on a per-identity basis. Each container may be associated with one and the same identity, under the assumption that students will actively collaborate and share management of the containers.

5.2 Hosting and Security Considerations

Hosting vulnerable applications on a public, internet-facing server is considered a bad practice. Special precautions must be taken to ensure that malicious actors cannot compromise the instance through a vulnerable container, and that the containers themselves are not compromised and used for criminal purposes.

A viable solution may be to avoid hosting the site at all. It is technically possible for the site to be kept local-only and require each student to download and run the project from their own machine. However, this would largely defeat the purpose of the project as it would require its own setup and installation steps.

Another option may be to install and configure the project local-only on the AWS instance but avoid exposing the instance to the public internet. Each student would be required to

connect to the instance through a VPN. The preceding two solutions are dead simple to the point of being below the level of sophistication expected of a 9-month project.

A more feasible solution would be to host the site and add an extremely restrictive security configuration. The instance could be configured to only accept incoming traffic from and send outgoing traffic to a small number of IPs or an IP range explicitly listed in a whitelist. This solution is ideal if the machines used in the lab have IP addresses that fall within a specific range or are unlikely to change over a semester-long course. Under this scheme, the target webapp containers would be blocked from making requests to third parties. Discretion would be left up entirely to the students (a rogue student would have the ability to change the whitelist).

5.3 Vulnerable App Selection and Containerization

In order to give our users a variety of options to choose from, we want to select a diverse set of vulnerable web applications. There is an existing registry of vulnerable web applications known as the OWASP Vulnerable Web Applications Directory (VWAD). This registry contains a variety of web applications that are intentionally designed to be vulnerable for educational purposes. We will select at least 7 web applications from this registry that cover a range of vulnerabilities and technologies.

The web applications listed in this registry record the technologies used as well as the specific vulnerabilities they contain. This information will help us ensure that we are providing a comprehensive and varied set of applications for our users to work with.

Creating a quality docker image for each web application is a non-trivial task. We will need to ensure that each image is reliable (it always deploys when required) and reproducible (the application source code and initial state are immutable and consistent). This may involve creating custom Dockerfiles, configuring the applications correctly, and testing the images thoroughly.

5.4 Technical Debt and Maintainability

While we may assume one identity controls all containers, it may still be smarter in the long run to design for the possibility of multiple concurrent users, even if this isn't made possible by our hosting configuration.

Since "make it support multiple users" is an excellent goal if this project returns to Senior Design, we will design for the possibility of this extension by programmatically tracking container IDs and associating them with the user's identity and actions. This design decision

also makes it easier for us to manage state and tell the user "You cannot stop a container you haven't deployed!" even in the single-user version of WADT.

As a stretch goal, future teams that work on this project would be able to extend our API instead of rewriting it.

5.5 "Do One Thing Well"

WADT is a stage in a learning pipeline. Pipelines are made possible by small, focused applications that do one thing well and rely on standard interfaces. WADT is an abstraction for managing containers. It is not an abstraction over the process of finding API endpoints, making requests, patching, and testing. Therefore, WADT should use conventional input and output formats and get out of the way of the user's preferred tools.

If all the user needs is a URL, provide the URL in an easy-to-copy format. Nothing else.

5.6 Definition of Done

- At least 7 target vulnerable webapps are listed on the site with icons, names, and descriptions. An image for each webapp has been created and tested for reliability (always deploys when required) and reproducibility (application source code and initial state are immutable and consistent). Each target app passes automated tests ensuring that the desired vulnerable endpoints are confirmed reachable.

- Container limits. Each container is allotted a specific amount of compute resources and network bandwidth. Each container is only kept running for a maximum of 24 hours from the last user interaction, at which point it will be automatically downed.

- Status indicator. Each webapp indicates its state. State indication matches actual container behavior (does not indicate stop when container with specified ID is still running, etc.).

- Deploy feature. Users can deploy any of the target webapps. Deploy is accompanied by connection information, assisting users in making requests to the vulnerable application.

- Stop feature. Users can stop any webapp. Stop only functions for containers which are currently running.

- Restart feature. Users can restart each webapp. "Restart" in this context means stop the webapp the student has been working on patching and start it again.

- Reset feature. "Reset" means to stop the version of the app the student has been working on, and deploy the original image before any modifications were made.

- Instructor controls. Site admins can view logs with the status of each container. Each deploy, stop, and restart action is logged. Admins have the option of deploy, stop, and restart, and their actions override the wishes of users (while preventing race conditions and state inconsistency). Instructors have the option to "disable" a container, meaning that the container leaves control of the students and only the instructors can manage it. When a container is disabled, students are unable to make any state changes to the container.

- Student-facing logging. Student-facing logs must provide a window into the container's behavior, without burdening the student with unnecessary Docker-specific details. For students, no single action in the UI should require scrolling the view of the logs (everything resulting from one action should fit on one view). All major interactions through the UI should be logged with timestamps. The logger must preserve history from the moment the container is started to the moment it is manually downed or auto-downed after 24 hours. In the event of an error resulting from a failed attempt to patch a vulnerability, the student must receive the full error message.

- Instructor-facing logging. Must show timestamped logs of all actions taken through the UI by students. Must show container status changes. Uptime for each container must be available to instructors.

5.7 Stretch Goals

- Instructor-facing controls to add new containers. The admin interface provides a method to upload a custom Dockerfile and add that image to the catalog. Instructors will be able to add container metadata (name, description, tags, category, etc.). This new container will function identically to the containers built-in to the site.

- Automatic patch checking. User communicates which API endpoints they wish to patch, and the site automatically checks periodically if that endpoint is still vulnerable. Status reports are generated showing which of a set of endpoints are still reachable at the time the report is generated.

- UI allows editing of network settings per-container. Users can customize port number on a per-container basis through the UI.

- System resource usage display. Admin UI displays, in addition to container uptime, CPU, memory, and network usage for each container.
- Custom colorschemes. Make site compatible with widely used color scheme file formats (minimum: .Xresources, .itermcolors, Windows settings.json, and .yml used for GNOME terminal and xfce4 terminal). Students will be able to use the same colors for WADT and for their terminal.
- Custom interactions window. Users can issue (arbitrary) commands to the container through the site. Users may edit files with a text editor through the site. This stretch goal is extremely ambitious. The security vulnerabilities potentially resulting from exposing a shell to a container are numerous and severe. Additionally, embedding a full terminal emulator inside of a webapp is a difficult task.

5.8 Tech Stack

6 Research

6.1 Hosting Provider

6.1.1 DigitalOcean

DigitalOcean droplets are small-sized virtual machines that are billed on a monthly basis. The team has previous experience using DigitalOcean droplets for hosting, making a Basic droplet a potential choice. However, since WADT is a SaaS application, in DigitalOcean's own terms a Basic droplet may not suffice. Additional tiers of compute and memory would impose an unfair cost on the student group.

6.1.2 BuyVM

BuyVM virtual machines are cheaper than a DigitalOcean droplet, but lack the support and convenient admin UI of DigitalOcean or AWS. BuyVM would allow maximum control and potentially a relatively unique hosting stack comprised of a hardened *BSD distribution.

However, this route was not chosen for the first iteration of WADT due to the high learning curve.

6.1.3 AWS

AWS EC2 was recommended by our sponsor and provided and paid for by our sponsor. This made it an attractive option for the team, even if the pricing model creates risks and additional considerations regarding extensive network traffic. AWS benefits from extensive documentation and high configurability. Ultimately, a single instance was chosen to start with the option of scaling to additional instances as usage of the project grows.

6.2 Host Operating System

6.2.1 OpenBSD

OpenBSD is a security-focused UNIX system. While using a security-focused system for a cybersecurity-focused project is a good idea, the relative scarcity of documentation meant a steep learning curve for the team.

Considering a *BSD distribution was quickly invalidated as the team determined that Docker is a Linux-specific application. Docker relies on Linux kernel namespaces as part of its core design, meaning that to use a *BSD distribution would require dramatic changes in other parts of the project, effectively invalidating WADT's original purpose. A BSD-based project would make use of Jails instead of Docker. Since users expect to manipulate Docker containers with WADT, changing to BSD would require a change in our MVP, making it more than a mere technical decision.

6.2.2 CentOS

CentOS is a Linux distribution which is used internally for the labs run by our sponsor. For the purposes of our users, the main functional difference they will experience between CentOS and other choices of distribution is the package manager. Key project pieces (Docker, NGINX, etc.) are available for every Linux distribution the team considered, the only difference between them being familiarity.

It was decided to optimize for our developer team's productivity in the beginning, and since no member had previous experience with CentOS, it was decided to postpone its adoption. WADT can be migrated to CentOS for lab instructors and students once its core features are developed.

6.2.3 Amazon Linux

Amazon Linux is a distribution provided by Amazon specifically for EC2 instances. Its primary advantage for the team would be easy integration with IaC tools like Terraform and CloudFormation.

Amazon Linux is not ideal for either developers or users in terms of previous experience and familiarity. It is not used internally within labs and was not used for testing and early development.

6.2.4 Ubuntu

Ubuntu was chosen to promote a fast ramp-up and make it easy for every member of the team to contribute. Ubuntu is an ideal choice, since it supports all elements of the project tech stack and is widely-used.

Specifically, Ubuntu by default enforces an externally-managed Python environment which effectively necessitates using virtual environments to manage project dependencies. This ensures no developers have conflicts between project-specific packages and system-wide packages and reduces dependency conflicts.

PostgreSQL is also available in Ubuntu's default repositories, simplifying project setup when compared to CentOS or Amazon Linux.

6.3 General Security Concerns

Security was one of the biggest concerns as hosting intentionally vulnerable applications inherently has many risks that need to be narrowed down and handled before pushed to a live server. Docker has some very strong security primitives that will be utilized to help the application be as secure as possible. Linux namespaces will be used to help ensure that each container has its own isolated process tree, filesystem, and network stack, ensuring that the containers are self-contained and unaware of the host system. A dedicated, isolated Docker bridge network will be used for each student's session. Containers will not be attached to the default bridge or the host network. Using this will aid in preventing these containers from scanning or attacking other services on the host machine/other containers. This essentially enforces a "deny by default" policy for communication within the containers and system itself.

One of the other main concerns was the containers not shutting down on time or a student potentially forgetting to shut down the container itself. Not only would this potentially lead to high overhead costs, it would also potentially leave an opening for a security breach to happen. Due to this, a life time limit will be implemented on the containers to automatically

run a stop command to ensure that the containers do not run for extended periods of time. This will help us cleanup our resources along with limiting the potentials for misuse.

6.4 Incoming Requests

Arbitrary command execution vulnerabilities are present for a number of selected target vulnerable webapps.

Suppose for a case example, the Damn Vulnerable Web Application (DVWA) selected by the team. DVWA includes a PHP remote code execution lab. Since arbitrary command execution is built-in to the target webapp, the slightest misconfiguration of the underlying Docker container will compromise the server.

Example exploitation steps:

1. Misconfigure Docker containers. In this example, a user uses a DVWA compose file with `privileged: true` and `pid: "host"` configured for each container
2. User starts DVWA with `docker compose up` with root privileges
3. User configures DVWA built-in security level to low using environment variables, config file, or webpage
4. User accesses command injection vulnerability in browser at /vulnerabilities/exec
5. User enters `localhost; ls /dev` to verify injection vulnerability. Devices on the host are listed. In this instance, the shell runs as user www-data, meaning that exploitation possibilities are roughly the same as a non-root user on the host. In the event of a privilege escalation vulnerability, the user may gain root access
6. For this demonstration, the user runs a fork bomb

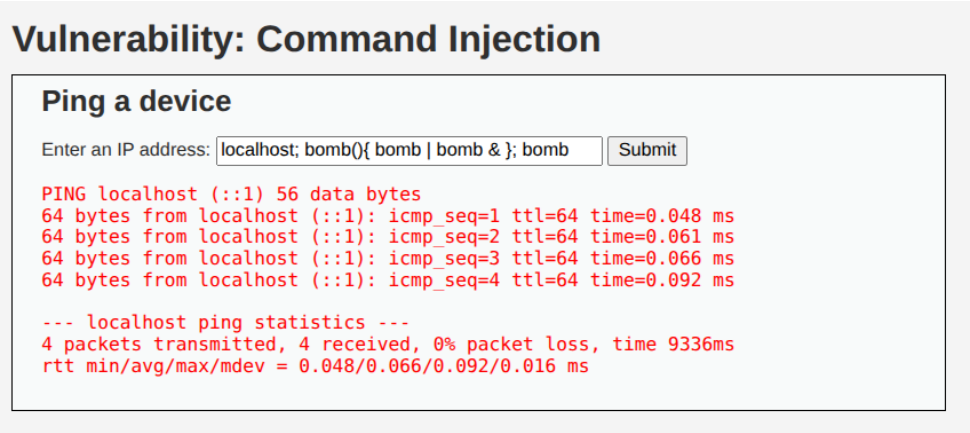

Figure 2: Command injection - fork bomb setup

7. A significant slowdown occurs on the host system. In principle, a user could perform this attack on the WADT site to take it offline

Figure 3: Command injection - fork bomb execution

6.5 Outgoing Requests

In addition to concerns that users could compromise the WADT site through any one of its containers, there is the additional concern that the containers could be manipulated to perform malicious outgoing requests. Containers, if compromised, could participate in DDoS, botnets, or be used to distribute illegal content, to name a few examples.

This problem is especially difficult to solve since blocking all outgoing requests will break any dependencies that target vulnerable webapps have on external services.

While a simple whitelist based on a search could whitelist each domain that a given target vulnerable app depends on, it would be easily exploitable by users. Given that users need write access to application source code in order to apply certain kinds of bugfixes, users would be able to add whatever external domains they wish, and therefore have them whitelisted.

As an example, the OWASP Juice Shop webapp makes external requests to CloudFlare to provide cookie consent forms. Disabling all outgoing requests and responses would break this dependency. In principle, individual apps cannot be relied on to provide an exhaustive list of the third-party domains they attempt to access. Whitelisting all domains that appear in a regex search has a few notable issues:

- Easily exploitable. Any form of write access to the source code of the application means full control of the whitelist
- Not dynamic. Any attempts to patch vulnerabilities by modifying source code which contacts third parties will be blocked until the whitelist is manually updated

- Not complete. Users may still make malicious requests to the domains that the application already requires

These problems are still an open issue for the team until a better solution is discovered.

6.6 Cross-Site Scripting (XSS)

XSS created problems for the original scope of WADT. These problems were discovered when researching target vulnerable webapps. OWASP hosts a Juice Shop instance that is shared between all users on the public internet. For this instance, all XSS challenges are disabled because of the potential consequences of XSS for users.

The Juice Shop project has taken the time already to detect container-based environments and disable the "danger zone" challenges to prevent XSS attacks on other users. However, for other target projects, or for real vulnerabilities, the demarcation of vulnerabilities into "safer" and "dangerous" is not preemptively done. A full audit of each selected application of the WADT project is outside of the scope of Senior Design.

As a result of these constraints, the WADT project is limited to a single-user scope. In principle, XSS attacks on other users cannot occur if only one user is active on the system. In this scenario, the user would have to deliver a malicious payload to the target vulnerable app that would harm their own machine. However, in practice, small groups of students are going to collaborate and be supervised by an instructor. The level of trust placed in the students with the potential for delivering malicious payloads to the target apps is similar to the level of trust to grant access to the original instance, meaning that this problem requires a social rather than technical solution.

6.7 2-Factor Authentication

In addition to previous security methods, it is desirable to add 2-factor authentication upon login. There are multiple approaches which solve the problem.

6.7.1 pyotp + pass + pass-otp

This solution implements TOTP (Time-based One Time Password) authentication, a form of 2-factor authentication, that relies on SSH access to the AWS instance which runs the site. This restriction ensures that the only users who can manipulate containers through the site *already* have full server access as a prerequisite.

The setup steps are as follows:

1. Use `pyotp` to generate a shared secret

2. Display the shared secret as a URI to the user on the WADT webpage. Other display options include QR code

3. User logs in to the instance over SSH

4. User uses `pass` with the `pass-otp` extension to store the URI

The usage steps are as follows:

1. User logs in to the instance over SSH

2. User runs `pass otp [secret-name]` to generate a one-time sign in code

3. User enters sign in code (or copies it) to the WADT site

4. User completes signin

While this solution is minimal, it relies on existing libraries and services for the core parts of TOTP, sidestepping documented issues with roll-your-own TOTP like floating point imprecision when calculating the time input.

This solution is also compatible with existing, more standardized authentication providers like Microsoft Authenticator, making it a sensible addition to the project.

All users who access WADT may be required to have SSH access to the underlying server as a prerequisite if it is ensured that only the `pass` program on the server is configured for MFA.

6.7.2 Traditional Authentication Providers

Traditional auth providers are available, most notably Microsoft Authenticator and Google Authenticator, that can provide the same service as `pass-otp` for users. However, since these traditional providers are not linked to the underlying hosting, they do not provide the same kind of security that the previous solution affords.

Traditional 2FA providers are primarily concerned with verifying the identity of users. Microsoft authenticator in and of itself will verify that the individual attempting a sign-on is the identity they claim to be. However, a solution with `pass-otp` also incidentally verifies that the individual attempting sign-on has access to the resources the WADT project is looking to protect.

6.8 Containerization Tools

6.8.1 OCI Compliance

The Open Container Initiative (OCI) maintains a group of three specifications which containerization tools are expected to follow to be compliant to the same standards as Docker. Since Docker was taken as a baseline technology for the project, only other OCI-compliant containerization tools were considered.

6.8.2 Podman

Podman is a drop-in replacement for Docker which is both daemonless and rootless. It differs in its underlying architecture: Podman is pod-based. A pod is an infra container combined with a number of regular containers.

For comparison, plain Docker functions interacts with plain containers, not pods. Docker Compose can be used to run multiple containers together for a project.

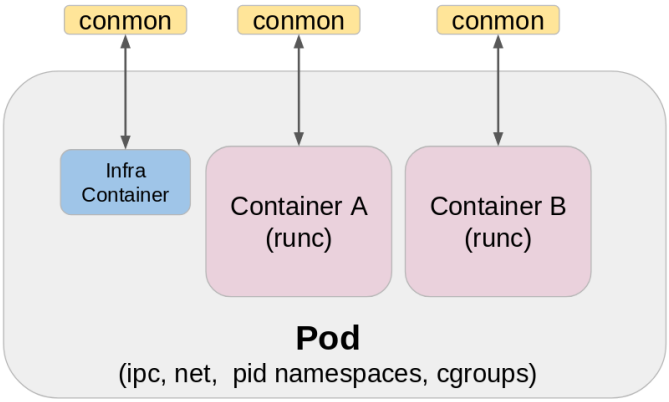

Figure 4: Pod architecture [3]

Podman's daemonless design means that it does not require a running background process (daemon) to function. Instead, Podman relies on systemd. This is in contrast to Docker, which requires a daemon with root privileges.

Podman is also rootless, meaning that system root privileges are not required by any part of Podman to run containers. This feature is the biggest draw of Podman over Docker for a security-focused project. With Podman, WADT would never require root privileges to run containers or be forced to interact with a privileged daemon process.

Although Podman is intended as a drop-in replacement for Docker, there are minor differences which ultimately led to the team's selection of Docker instead. Docker can perform

pulls from DockerHub with short image names, while Podman requires additional configuration to do so. So while `docker pull postgres` will run successfully, `podman pull postgres` will throw an error.

These small differences are likely to create cognitive friction for users of WADT.

6.8.3 Docker

Docker is a widely-used set of container tools. A number of features made Docker the ideal choice for WADT.

- Official, extensive SDK with available low-level API access
- Images already available for numerous applications. While any OCI-compliant container engine can run these images, in practice they are built and tested for Docker
- DockerHub integration

6.9 Programming Language and Web Framework

6.9.1 Python and Flask

Flask is a micro web framework for Python. Since it's a micro framework, it lacks certain key features the team finds valuable for developing. Primarily, there is no built-in admin interface for interacting with the project's database given a model (it must be added using Flask Admin).

WebSockets are a key part of the project to provide live status updates for each container. Unfortunately, the support for WebSockets with Flask is relatively dated. The Flask-Websockets repository was archived last year, and the tutorial for Flask-SocketIO, an alternative library, was last updated in 2014.

The ecosystem of extensions to Django was found to be more actively developed. Since the project is timeboxed to a 9-month time frame, the "batteries included" features of Django were chosen over Flask.

6.9.2 Python and Django

Django is a web framework for Python which was chosen for its sensible architecture and useful features. Django's ORM and admin panel made configuring the database a one-day job, while its migrations feature ensures that any changes are tracked along with the project.

Django's MVC architecture and template system enforces clear separation of concerns and makes prototyping fast and easy. Combined with its powerful ORM, simple views can be implemented in a few lines of code.

```
def index(request):
    container_catalog = Container.objects.order_by("name")
    context = {"container_catalog": container_catalog}
    return render(request, "wadtapp/index.html", context)
```

Figure 5: Django queries and template system make simple pages simple to write.

6.10 Docker Integration

6.10.1 Overview

Docker integration lies at the core of the Web App Deployment Tooling (WADT) project. Every feature from container deployment to logging, resetting, and shutting down a container relies on Docker's ability to build and manage isolated environments quickly and consistently. While previous sections discuss why Docker was chosen over alternatives like Podman, this section focuses on how Docker is practically integrated into our system architecture and development workflow.

At a high level, Docker serves as the layer that bridges our backend application logic and the intentionally vulnerable web applications being deployed for users. When a user interacts with the WADT frontend, such as clicking "Deploy," "Stop," or "Reset", these requests travel through our Django-based API to the Docker engine via the python-on-whales library. The backend then performs the necessary Docker operations, returning live container states, logs, and connection details to the frontend. This structure allows us to abstract away all of Docker's complexity while maintaining full control and observability over the containerized environments.

Docker integration most importantly ensures reproducibility. Each vulnerable web application is built from a deterministic Dockerfile and versioned image, which guarantees that students and instructors see identical environments across sessions. This is vital in cybersecurity labs where even small configuration differences could lead to inconsistent or misleading results. By leveraging Docker's lightweight image system, we achieve repeatable deployments without sacrificing speed, isolation, or complicated library version overhead.

6.10.2 Integration with Backend (Django + Python-on-Whales)

The WADT backend, built with Django, uses the python-on-whales library to communicate directly with the Docker engine. Python-on-whales is a high-level, Pythonic interface to Docker that wraps the standard Docker CLI, allowing developers to invoke container operations using clean Python syntax while retaining access to the full range of Docker functionality.

When a user initiates a deployment from the frontend, Django processes the API request and uses python-on-whales to start a container from the appropriate image. Each container is instantiated from a predefined image stored locally on the AWS instance. Metadata such as the container ID, image name, network configuration, and timestamps are recorded in the project's database, ensuring synchronization between Django's logic layer and Docker's runtime state.

One major reason for selecting python-on-whales over the traditional Docker SDK docker-py is its built-in thread safety and high-level abstractions. WADT often handles multiple container deployments simultaneously—one per vulnerable web application, and in the future, potentially one per user session. Python-on-whales manages Docker operations safely in multi-threaded or asynchronous contexts, preventing race conditions and socket contention that can occur when using the lower-level SDK. This makes it ideal for our multi-user educational environment, where concurrency is common.

The backend uses the library to manage all stages of the container lifecycle, including creation, execution, stopping, and log retrieval. Example commands include:

```
from python_on_whales import docker

# Create and start a container
container = docker.run("bkimminich/juice-shop", detach=True, name="student_1")

# Retrieve logs
logs = docker.logs(container)

# Stop and remove container
docker.stop(container)
docker.remove(container)
```

These operations are wrapped in Django view functions or background tasks, ensuring consistent control flow and error handling. Additionally, since python-on-whales mirrors the actual Docker CLI syntax, developers can more easily reason Docker commands and debug

them both in our local environment and in our production environment.

6.10.3 Security Considerations

Security within this integration layer is enforced by restricting which Docker operations Django is permitted to execute. Containers are never launched with privileged flags or mounted host directories, and Docker's network isolation ensures that containers cannot access the host or each other unless explicitly configured. This keeps intentionally vulnerable applications securely contained while preserving functionality for testing and instruction.

6.10.4 Container Lifecycle Management

Every container deployed through WADT follows a clearly defined lifecycle managed by the backend. Understanding this lifecycle is crucial for maintaining predictable behavior, preventing resource exhaustion, and ensuring a secure sandboxed experience for users.

1. **Creation:** When a user selects a vulnerable web application and clicks "Deploy," Django instructs Docker to create a new container from the corresponding image. Environment variables, network parameters, and resource limits (CPU, memory, bandwidth) are configured during this stage.

2. **Execution:** Once created, the container is started in detached mode. The user is provided with access details, such as connection URLs and exposed ports. Docker isolates the container using Linux namespaces and cgroups, ensuring strong separation from the host.

3. **Logging:** All container logs are streamed through the backend using the docker.logs() function. These logs are relayed to both student and instructor dashboards, allowing visibility into application behavior, debugging messages, and exploitation attempts.

4. **Timeout and Auto-Stop:** Containers are automatically stopped after a set period of inactivity (e.g., 5 hours). This prevents long-running containers from consuming excessive resources or remaining exposed unnecessarily. The backend periodically polls Docker for container statuses to enforce these limits.

5. **Reset and Cleanup:** Users can reset their environment at any time, triggering Django to stop and remove the existing container and redeploy a clean instance. When containers are stopped or deleted, associated volumes and networks are also removed to avoid leftover data or stale configurations.

This lifecycle model ensures reliability and safety. It keeps container environments limited in lifespan, prevents resource leaks, and allows users to repeatedly practice deployments and exploitations in clean, reproducible environments.

6.10.5 Future Improvements and Stretch Goals

While Docker currently provides the foundation for container management in WADT, several future enhancements could further increase performance, scalability, and flexibility.

A major planned improvement is the implementation of per-user dynamic container allocation. Currently, containers are managed centrally, but expanding this to associate containers with specific user sessions will allow simultaneous isolated environments for multiple students. Python-on-Whales concurrency support and thread-safe design make this a feasible upgrade path.

Another enhancement would involve adopting Docker Compose or Kubernetes for orchestration. Docker Compose could simplify management of multi-container environments, while Kubernetes would enable automatic scaling, health monitoring, and load balancing—features especially useful for larger educational deployments. Since python-on-whales supports Compose commands directly, this transition could be made incrementally without rewriting backend logic.

Finally, the team could also automate image rebuilding and vulnerability patching. A nightly rebuild process could use docker.build() to refresh base images, integrate new security updates, and revalidate application configurations. Combined with continuous secret scanning and monitoring, these improvements will make WADT more robust and sustainable as an educational security platform.

We also could host a wider array and wider number of vulnerable web applications by automating the Dockerfile creation process. By programmatically generating Dockerfiles from application metadata, we could rapidly expand the catalog of available targets without manual intervention. This would greatly enhance the learning opportunities for students and ease the work of whoever wants to contribute a vulnerable web application to our platform.

In summary, Docker integration—implemented through python-on-whales is central to WADT's architecture. It enables safe, concurrent, and repeatable container management while providing a clear, maintainable Python interface for future growth.

6.11 python-on-whales

Python-on-Whales is a clean, one-to-one bridge between the Docker command-line interface (CLI) and Python. It acts as a direct wrapper, letting developers control Docker with familiar

Python syntax instead of juggling terminal commands. In short — if you know Python, you already know how to use Python-on-Whales.

What makes it stand out is its balance between simplicity and full Docker coverage. It mirrors nearly every Docker feature while offering Pythonic conveniences and type safety.

Some of its main features include:

- Support for the latest docker features
- Support for Docker stack, services, and Swarm
- Progress bars and progressive outputs when like: pulling, pushing, etc
- Support for other CLI commands such as `docker cp`
- SSH support for remote daemons.
- Contains a fully typed API
- All docker objects and the docker client are safe to use with multithreading and multiprocessing

Python-on-Whales basically turns docker automation into something readable, testable, and smooth to integrate with larger python projects, no weird bash script required. More details from python-on-whales can be found on their official website [2].

6.12 Go, and Go-sdk

Go is an open-source programming language developed by Google, built for efficiency in cloud computing, networking, and large-scale distributed systems. It's commonly used for developing command-line interfaces (CLIs), backend web services, and site reliability tools.

AWS SDK for Go (Go-SDK) is a software development kit that integrates Go applications with Amazon Web Services. It provides Go developers with the ability to interact directly with AWS APIs, making it useful for managing cloud resources and automating deployments.

Since our project uses AWS for hosting, we briefly considered using Go with the AWS SDK for backend operations. However, we ultimately decided against it, as Go wasn't part of our primary backend stack. Sticking with our existing setup allowed us to keep the development process simpler and avoid unnecessary integration overhead.

6.12.1 docker-py

`docker-py` has two problems that make it a poor choice for the project:

- Lack of built-in support for Docker Compose. This means additional libraries must be installed for a number of the selected target vulnerable apps
- Lack of guarantee of thread-safety for core data structures. The DockerClient class provided by `docker-py` is not documented as thread-safe, and appears to store mutable state that affects program execution. Compared to python-on-whales, which makes a guarantee of thread-safety for the core client data structure, docker-py requires more overhead to manage

6.13 Vulnerable Web Application Selection

6.13.1 Selection Criteria

The goal of the Web App Deployment Tooling (WADT) project is to provide a catalog of intentionally and unintentionally vulnerable web applications that together expose students to a broad, representative set of real-world security issues. We are aiming to maximize educational value while remaining practical to deploy and maintain. We selected target applications according to the following criteria:

- **Vulnerability breadth:**
  Each application should expose multiple common vulnerability classes (for example, the OWASP Top 10) so students can practice exploiting and patching different kinds of flaws.
- **Technology diversity:**
  The set should span several technology stacks (e.g., JavaScript/Node, Python/Django, PHP, LAMP, PostgreSQL) to expose students to language and framework-specific security issues and tooling.
- **Educational maturity:**
  Preference for well-documented, actively maintained projects with clear lab-friendly features (hints, adjustable difficulty, reset functionality).
- **Container friendliness:**
  Applications must be reliably containerizable and suitable for running in isolated, resource-constrained environments.

- **Practical relevance:**
  The application scenarios should reflect realistic application classes (API-driven services, single-page apps, e-commerce platforms, CMS, legacy stacks) so students learn transferable skills.

6.13.2 Intentional vs. Unintentional Vulnerabilities

Intentional (Designed) Vulnerabilities

Intentional or designed vulnerable applications are created specifically to teach security concepts. They are instrumented to contain a wide variety of exploitable flaws, often include challenge scaffolding or hints, and are typically packaged to be easy to reset and reproduce. Examples in our catalog (e.g., OWASP Juice Shop, DVWA, PyGoat) fall into this category. These apps prioritize discoverability and repeatability of vulnerabilities so students can quickly attack and patch vulnerabilities to varying extents.

Unintentional (Real-world) Vulnerabilities

Unintentional vulnerabilities are real production defects discovered in mainstream software: they were not created for teaching but were found and disclosed (often with CVEs). Including intentionally insecure apps alongside examples of unintentional vulnerabilities (e.g., a historical path traversal or file upload RCE) gives students exposure to the kinds of high-impact bugs that affect real services. These examples teach vulnerability discovery techniques, incident response workflows (patching, CVE analysis, key rotation), and the ethical responsibilities of disclosure and remediation. Although these vulnerabilities may be harder to exploit or reset, they provide invaluable context for understanding the stakes of security in practice. Hints and path to reproduce vulnerabilities will be offered to aid students.

6.13.3 Selected Applications

6.13.4 OWASP Juice Shop

Overview and stack

OWASP Juice Shop is a modern intentionally insecure web application implemented with a Node.js/Express backend and an Angular frontend. It is designed as a single-page application that emulates contemporary web development patterns.

Educational value

Juice Shop covers a very wide set of vulnerabilities (many items from the OWASP Top 10) in a modern application context: authentication/authorization flaws, insecure direct object references, broken access control, XSS, CSRF, injection issues, and client-side security problems. Because it is SPA-based, it is particularly good for teaching students about the interaction between frontend frameworks and backend APIs, token storage, and client-side attack surfaces.

Deployment notes and Docker considerations

For WADT we will prefer pinned image tags or build images from a known commit to ensure reproducibility. Typical deployment will require exposing the application port and mapping any data volume used for persistence. Resource usage is moderate; containers may be run with constrained CPU and memory limits without impairing lab exercises.

Why chosen

Juice Shop provides a modern, realistic lab with a polished UI and a comprehensive set of built-in challenges. It is ideal for intermediate-to-advanced students who need practice against up-to-date web stacks.

6.13.5 PyGoat

Overview and stack

PyGoat is a Django-based intentionally vulnerable web application implemented in Python and Django. It is generally smaller in scope than other applications but focuses on server-side vulnerabilities common to Python web frameworks.

Educational value

PyGoat is useful for students who want to explore issues in Python/Django applications specifically: insecure ORM usage, improper authentication/authorization, template injection, and misconfigured middleware. Its smaller codebase makes it easier for students to read the source, trace control flow, and implement fixes.

Deployment notes and Docker considerations

PyGoat can be containerized with a lightweight image based on an official Python base image. The database backend for PyGoat is typically SQLite or a lightweight DB by default, but

we may run it against PostgreSQL in a secondary container (or a single container with the DB bundled).

Why chosen

PyGoat focuses on server-side Python/Django security, offering a clear environment for students to practice code-level remediation and gain familiarity with a widely-used backend framework.

6.13.6 Damn Vulnerable Web Application (DVWA)

Overview and stack

DVWA is a classic PHP/MySQL vulnerable web application widely used in education. It intentionally exposes a set of vulnerabilities and allows the security level to be adjusted to tune difficulty.

Educational value

DVWA is particularly well-suited for beginners and intermediate students because exercises are simple to understand and reproduce: SQL injection, cross-site scripting (XSS), command injection, file inclusion, and authentication bypasses. The adjustable security level makes DVWA a flexible tool for progressive lab assignments, starting from smaller exploits and advancing to realistic variations.

Deployment notes and Docker considerations

DVWA is typically deployed as a LAMP-style container or with separate containers for PHP and MySQL. Because it is lightweight, DVWA is easy to run on constrained hosts. We must ensure default credentials and debug features are documented and reset between sessions.

Why chosen

DVWA's long-standing presence in security education, its adjustable difficulty, and its simplicity make it an ideal entry point for students learning web vulnerabilities and exploitation basics.

6.13.7 Ghost CMS (Unintentionally Vulnerable Example)

Overview and stack

Ghost is a widely used open-source blogging and publishing platform built on Node.js. It represents a realistic production-grade content management system (CMS) that students might encounter in the wild.

Vulnerability detail

A historical path traversal vulnerability (CWE-22) was disclosed for Ghost (CVE-2023-32235). Affected versions prior to 5.42.1 allowed an attacker to read arbitrary files on the web server (for example, system configuration files) without authentication by exploiting an improper input path normalization issue.

Educational value

Including Ghost with this specific CVE gives students a real-world case study in how a seemingly small file-path handling bug can lead to critical information disclosure. Labs can cover:

- How path traversal vulnerabilities arise (unsafe file path concatenation, insufficient input validation).
- How to reproduce the issue in a controlled environment using crafted requests.
- Incident response steps: responsible disclosure, patching strategy, and post-exposure mitigation (rotate secrets, review logs).
- Differences between exploiting intentional educational vulnerabilities and analyzing disclosed CVEs in production software.

Deployment notes and Docker considerations

Because this is a production-grade application, resource and configuration considerations (storage for themes, filesystem access, and user content directories) must be handled carefully to avoid leaking host state. For lab purposes, the container will be run in an isolated network with seeded content and no external persistence unless explicitly required.

Why chosen

Ghost demonstrates a realistic, unintentional vulnerability in a contemporary platform and teaches practical incident-handling skills in addition to exploitation techniques.

6.13.8 PrestaShop (Unintentionally Vulnerable Example)

Overview and stack

PrestaShop is an open-source PHP e-commerce platform used for online stores. It is representative of a PHP-based web applications that manages business workflows.

Vulnerability detail

A high-severity vulnerability (CVE-2023-3882) affecting PrestaShop versions such as 8.1.1-apache was disclosed that allows arbitrary file write and potentially remote code execution. Typical exploitation involves a 'Zip Slip' or unsafe file upload path that permits an authenticated or misconfigured flow to write a webshell or malicious file. This enables arbitrary command execution on the server.

Educational value

PrestaShop with this CVE illustrates real-world risks in file handling and upload features on e-commerce platforms. Lab objectives include:

- Demonstrating how improper validation on file uploads and archive extraction can lead to arbitrary file write and RCE.
- Teaching safe file handling patterns (validate file names, use safe extraction libraries, enforce strict upload directories with no execute permissions).
- Practicing mitigation steps for production systems: patching, credential rotation, integrity checks, and rollbacks.

Deployment notes and Docker considerations

For lab reproducibility we will ensure database seeding is deterministic. Because this scenario can involve writeable filesystem areas, containers will be carefully sandboxed (no host mounts, strict filesystem permissions) to avoid any risk to the host.

Why chosen

PrestaShop is a high-impact, real-world example of how file handling vulnerabilities in feature-heavy web applications can lead to full server compromise. It complements our intentional-vulnerability apps by demonstrating production failures.

6.13.9 vAPI

Overview and stack vAPI (Vulnerable Adversely Programmed Interface) is a self-hosted lab that shows common API security issues from the OWASP Top 10. It uses PHP and MySQL, built with the Laravel framework. The project includes Postman collections for testing and supports Docker Compose and Helm for deployment.

Educational value vAPI helps users learn about real-world API vulnerabilities like broken access control, data leaks, and insecure input handling. Each issue is set up as a hands-on challenge, making it useful for students, ethical hackers, and security testers. Postman files make it easy to follow and repeat tests. Some vulnerabilities stimulated in vAPI are lack of resources, injection flaws, and broken authentication.

Deployment notes and Docker considerations vAPI comes with a Docker Compose setup that runs both the Laravel app and MySQL database. Users should set up the `.env` file with correct database settings and match passwords across files. Using volumes helps save logs and exploit data. After setup, the app runs at `http://localhost/vapi/`.

Why chosen vAPI was chosen because it focuses on API-specific security issues, which are common in modern apps. It's easy to deploy, well-documented, and recognized by the security community. It fits WADT's goal of offering practical, container-based labs for learning and testing.

6.13.10 NoSQL Injection Vulnerable App (NIVA)

Overview and stack NIVA is a simple web application built to demonstrate NoSQL injection vulnerabilities in MongoDB. It uses Java as the backend language and connects to a MongoDB database using the official driver. The app allows users to search for contact records by email, and includes both secure and insecure query implementations to show how different coding practices affect security.

Educational value This app is designed to help users understand how NoSQL injection works and why it matters. It shows how attackers can manipulate query inputs to access or modify data they shouldn't be able to. By comparing safe and unsafe query methods, learners can see the impact of poor input validation and insecure query construction. NIVA is useful for students, ethical hackers, and developers who want hands-on experience with NoSQL security issues. Its vulnerabilities are Data extraction, NoSQL Injection, and Privilege escalation.

Deployment notes and Docker considerations NIVA is easy to run using Docker. A prebuilt image is available on Docker Hub, and the app can be launched with a single command: `docker run -p 8080:8080 aabashkin/niva`. Once running, it's accessible at `http://localhost:8080`, where users can test both secure and insecure endpoints. No extra setup is required, but you can add volumes if you want to save logs or test data. The app runs as a standalone container, making it simple to reset and reuse in different testing scenarios.

Why chosen We chose NIVA because it focuses on NoSQL injection, which is a type of vulnerability that's becoming more common as NoSQL databases grow in popularity. The app is lightweight, easy to deploy, and clearly shows the difference between secure and insecure code. Its Docker compatibility and minimal setup make it a great fit for WADT's goal of providing practical, container-based labs for learning and experimentation.

6.13.11 VulnerableApp-facade

Overview and stack VulnerableApp-facade is a Java-based tool that helps manage and launch multiple vulnerable applications from one place. It's built using Spring Boot and provides a web interface and API to control apps like OWASP Juice Shop. Instead of running each app separately, this facade lets users start, stop, and switch between them easily. It acts like a central dashboard for security labs.

Educational value This app has vulnerabilities like SQL injection, weak login flows, and session handling. This tool is useful for learners, trainers, and testers who want to explore different types of web vulnerabilities without setting up each app manually. By combining several vulnerable apps into one interface, it can save time and helps users to focus on learning. It also supports tracking progress and organizing exercises, which is helpful in classroom or workshop settings. Users can compare how different apps handle similar vulnerabilities and learn from the differences.

Deployment notes and Docker considerations VulnerableApp-facade supports Docker and includes a Docker Compose file to run the facade along with selected vulnerable apps. Users can choose which apps to enable by editing the configuration file. The default port is `9090`, and the web interface is available at `http://localhost:9090`. The setup is straightforward, and no customization is needed to get started.

Why chosen We chose VulnerableApp-facade because it makes working with multiple vulnerable apps much easier. Instead of setting up each app manually, users can launch everything from one place. It's simple to use, works well with Docker, and supports a wide range of popular security labs. This makes it a great fit for WADT's goal of building a flexible and user-friendly environment for learning, testing, and teaching web application security.

6.13.12 BodgeIt Store

Overview and stack The BodgeIt Store is a purposely insecure web application created to help people learn about web security. It's written in Java and uses JSP (JavaServer Pages) for the frontend. The app runs on a servlet container like Apache Tomcat and uses an in-memory database, which means it doesn't need any external database setup. It simulates a basic online store with features like user login, product listings, and a shopping cart. These features are intentionally built with security flaws to make them easier to test and exploit.

Educational value BodgeIt Store includes common web vulnerabilities like SQL injection, XSS, CSRF, and IDOR, all exposed by default for testing. This app is great for beginners who want to practice finding and understanding common web vulnerabilities. It includes examples of issues like cross-site scripting (XSS), SQL injection, cross-site request forgery (CSRF), and insecure direct object references. There are also hidden pages and functions that can be discovered through testing. One helpful feature is the scoring page, which shows which vulnerabilities have been found, making it useful for self-paced learning or classroom challenges. It's a good starting point for anyone new to ethical hacking or penetration testing.

Deployment notes and Docker considerations It is easy to run using Docker, which makes the app simple to set up and reset. A prebuilt Docker image is available on Docker Hub. To start the app, users can run the command: `docker run -p 9090:8080 psiinon/bodgeit`. After that, the app will be available at `http://localhost:9090 /bodgeit`. Since it uses an in-memory database, all data is lost when the container stops, which is useful for resetting

the environment between tests. If needed, users can attach volumes to store logs or other data. No extra configuration is required, making it quick to get started.

Why chosen We chose the BodgeIt Store because it's simple and easy to use, especially for people who are just starting to learn about web security. Even though it's no longer actively updated, it still works well and covers many of the most common web vulnerabilities. Its Docker support makes it easy to include in WADT's container lab environment. Overall, it's a usable tool for teaching and practicing basic security testing skills.

6.13.13 Damn Vulnerable Restaurant

Overview and stack Damn Vulnerable RESTaurant is a Python-based API training app built with FastAPI and PostgreSQL. It includes REST and GraphQL endpoints and simulates a restaurant system with user roles and data access.

Educational value The app teaches API security through hands-on challenges. Users explore vulnerabilities like broken authentication, insecure access control, and data exposure while progressing through game-like levels. It also has vulnerabilities such as Broken Function Level Authorization (BFLA), Broken Object Level Authorization (BOLA), and Server-Side Request Forgery (SSRF).

Deployment notes and Docker considerations It supports Docker Compose and includes scripts for developer and hacker modes. The API runs on port `8091`, with Swagger and Redoc docs available. Data persists across sessions, and volumes can be added for logging.

Why chosen We chose this app for its modern stack, interactive design, and strong focus on API-specific flaws. It fits WADT's goal of providing practical, containerized labs for security learning.

6.13.14 Deployment and catalog-level considerations

For each selected application, we plan to:

- **Prefer reproducible images:**
  Use images from known commits to ensure identical lab environments across sessions.
- **Use ephemeral containers by default:**
  Configure deployments so containers can be reset or destroyed and recreated from a clean image to avoid state drift between student sessions.

- **Manage persistence intentionally:**
  When persistent state is required (for example, seeded content in Ghost or database state for PrestaShop), provide documented seed fixtures and automated reset scripts.

- **Resource planning:**
  Assign resource limits (CPU, memory) to each container type and test concurrent runs to validate the AWS instance sizing for class workloads.

- **Documentation and lab guides:**
  Provide short, focused lab instructions and expected learning outcomes for each app so instructors can map exercises to teaching objectives.

6.13.15 Summary

Our chosen catalog (OWASP Juice Shop, PyGoat, DVWA, Ghost CMS, and PrestaShop) provides a balanced mix of intentionally crafted educational targets and real-world, unintentional vulnerabilities. This mixed approach allows students to practice both guided exploitation and code-level fixes in pedagogical environments, while also learning to analyze, reproduce, and respond to actual CVEs found in production software. The container-based WADT architecture and the python-on-whales backed orchestration are well-suited to host this catalog in a reproducible, isolated, and instructor-friendly manner.

6.14 Asynchronous Support

6.14.1 Celery

Celery is a task queue that's compatible with Django. While it is capable of efficiently handling the asynchronous message-passing needed to send real-time status messages for each container, it involves many features that aren't needed for the project.

Celery requires the choice of a message transport (another dependency and another source of complexity). Its performance features are excellent, even beyond the scope of the MVP. "Millions of tasks a minute" [1] are not likely to happen within the project scope.

Due to the high complexity associated with a sophisticated solution for concurrent computation, the team opted for a simpler solution, Django Channels.

6.15 Django Channels

Django Channels is a project that will take Django and extend its abilities beyond HTTP so it may handle websockets, chat protocols, IoT protocols, and more. Django channels

comprise of multiple packages:

- Channels: the Django integration layer
- Daphne: the HTTP and Websocket termination server
- asgierf: the base ASGI library, and channels redis: the Redis channel layer backend.
- channels-redis: a Redis-based backend for message passing

In our project, Django Channels is used to eliminate the need for the client to repeatedly poll the server for updates. Instead, it keeps a live connection open, enabling real-time, two-way communication through WebSockets. Rather than following the traditional request-response cycle — where the client constantly sends "any updates?" messages — Django Channels allows the server to push updates to the client as they happen. This drastically cuts down on network overhead and server load, since the client only receives data when something actually changes.

By maintaining an open, event-driven connection, Django Channels creates a faster, more responsive interaction between the client and server. The result is smoother, real-time feedback and an overall better user experience.

6.16 Backend Architecture

6.16.1 Django MVC

what's meant here is using django's built in stuff for rendering HTML templates with whatever data you want.

6.16.2 API-only

what's meant here is using Django to create an API that just sends and receives JSON and doesn't take responsibility for rendering pages, which is what we're actually doing.

6.17 API Testing

API testing is an essential step in the development process. API testing serves multiple functions, and through testing will be a necessity throughout the development process. The most relevant types of testing for WADT are:

- Functionality Testing. This is the most important function of API testing. It answers the simple question, Does the API work as intended? For our project, this includes verifying that container deployment, stopping, restarting, resetting, and logging functions work correctly and return the expected information.

- Security Testing. Given that our application manages containers running intentionally vulnerable web applications, security testing is crucial. We need to ensure that the API itself cannot be exploited to bypass container restrictions and compromise our AWS server

- Interoperability Testing. Our API communicates with Docker through the Python Docker SDK. If there is a mismatch between SDK versions or Docker versions, this could cause errors when attempting to manage containers. It is important that we thoroughly test this to make sure these interactions work as expected.

It is important that we use the right technologies for such an important step in development. The two technologies we considered for API testing are cURL and Postman. Both come with their own benefits and disadvantages.

6.17.1 Postman

Postman is a popular API testing tool with one of its main selling points being its graphical user interface and collaboration tools. The interface simplifies the process of creating, sending, and managing API requests. This makes it especially useful for our projects' collaborative development.

Some of the main advantages of using Postman include:

- User-Friendly Interface: Postman has its own application with a straightfoward interface, making it easy to quickly set up and execute API tests without extensive configuration.

- Collaborative Workspaces: Postman offers shared workspaces where tests can be easily shared with other team members, ensuring consistent and repeatable testing across the development team.

- Scripted Testing with JavaScript: Postman allows automated test scripts to be written in JavaScript, enabling the creation of reusable testing routines and smoother integration.

- CI/CD Pipeline Integration: Postman tests can be run within CI/CD pipelines, allowing for automated testing as part of the deployment process.

- Cost: The basic version of Postman is free to use many helpful features to suit our current testing needs.

6.17.2 cURL

cURL is another tool used for testing REST APIs. Unlike Postman, which provides a graphical user interface, cURL operates entirely from the command line. This makes it better for scripting and automating tests.

Some advantages of using cURL include:

- No setup required or cost. cURL is preinstalled on most UNIX-based systems, so there is no need for additional configuration prior to testing.
- More versatility. It supports protocols beyond HTTP and HTTPS, such as FTP, FTPS, SCP, and SFTP. However, these are not likely to be needed for our application.
- Efficiency Using the command line can be faster than using an interface, particularly for quick testing or repeated API calls.
- Automation and scripting. cURL can easily be integrated into shell scripts and CI/CD pipelines to more efficiently confirm that our Docker containers and backend APIs are functioning properly.

6.17.3 Summary: cURL and Postman in Our Testing Workflow

Overall, cURL complements Postman well. While Postman is best at collaborative testing and visualization, cURL is better suited for automation of tests as well as integration with the CI/CD pipeline. Together, they provide more flexibility for our testing purposes. It is possible that we use both, however, we will primarily use cURL, as it better integrates with other API documentation tools.

6.18 API Documentation

API documentation is often an overlooked aspect of software development, yet it is one of the most important. Thorough and accurate documentation provides clear communication between the API development team and anyone who needs to interact with the API. This includes front-end developers, testers, and even future developers so they can properly understand the API functions. Without proper documentation, using an API can feel like trying to build a piece of furniture out of the without an instruction manual. While it is technically possible, it can be slow, confusing, and prone to error.

Comprehensive API documentation saves time and preserves sanity, by providing a formal description of how each endpoint works, what data it expects, and what responses it returns. This is especially important for complex APIs, where multiple endpoints interact with each other. For our project, maintaining detailed API documentation will also make it easier for future teams to continue development and troubleshoot any issues in a more efficient way.

We considered two tools for documenting our APIs, Hopscotch and SwaggerHub. Both providesda standardized way to describe REST APIs. However, they differ in cost, accessibility, and integration capabilities within our existing testing structure.

6.18.1 Hoppscotch

Hoppscotch is a free and open-source platform for testing and documenting APIs. It is entirely browser-based, making it easy for all team members to access without any need for additional installation. One of Hoppscotch's biggest advantages for our project is its ability to integrate with our API testing tools such as cURL.

As we test our APIs, Hoppscotch will allow us to document each endpoint directly within the same environment, which can be very convenient. Hoppscotch can also generate cURL commands for each request. This makes it simple to copy those commands into scripts for automated or mode advanced testing in the future. This allows for the ability to keep our documentation and testing closely aligned throughout the development process, rather than writing APIs and documenting at the end. Because cURL can be used in our CI/CD pipeline, this integration also helps to ensure that what we have documented remains consistent with what is being actively tested.

Endpoints can also be grouped into collections which help to organize the different APIs, rather than having one large list of different endpoints. Any documentation, alongside any additional notes, can be easily shared with its Share feature.

6.18.2 SwaggerHub

SwaggerHub is well-known platform for designing and documenting APIs, which incorporates the OpenAPI Specification or OAS. This specification provides a standardized format for describing REST APIs. These descriptions are defined within a YAML or JSON document and are used to define all aspects of the API, such as the endpoints and their corresponding response formats, status codes and example payloads. The goal of these OAS schemas is to provide a structure that can clearly be understood by anyone reading it.

SwaggerHub offers additional advanced features such as version control, various collaboration tools, and even automatic documentation generation. It can also be integrated with

CI/CD tools to keep API documentation in synch with live deployments.

While SwaggerHub does offer many helpful features, it does have a significant drawback, its high cost. While a 30-day free trial is available, the Team plan, which includes the collaboration features, costs $29 per month.

6.18.3 HoppScotch or SwaggerHub?

Both Hoppscotch and SwaggerHub provide suitable options for our API documentation needs. While SwaggerHub offers more professional-grade collaboration and versioning tools, it also comes with a high monthly cost. This is a long-term project, so we would be required to maintain documentation well beyond the free trial period and thus would have to pay $29 per month for the Team package. This high cost makes SwaggerHub not feasible in the long term, as we are college students trying to avoid unnecessarily high out of pocket costs. Hoppscotch by contrast, is free, yet it still provides the tools we need to properly document our APIs. It also can integrate directly with cURL, allowing us to import cURL commands directly into Hoppscotch. Additionally, it supports real-time documentation alongside testing. This makes Hoppscotch the more practical tool for our API documentation.

6.19 API Performance Testing

6.20 Implementing Webhooks

Webhooks are one-way calls that originate from the backend and notify another system when a specific event has occurred. They differ from traditional API requests in that they are push-based rather than pull-based—instead of the frontend constantly asking for updates, the backend automatically sends information when something changes. For instance, when a user makes a purchase on Amazon, an API call first records the transaction in the backend's database; once the payment is verified, a webhook is sent to confirm the successful purchase and update the user interface in real time.

In our project, webhooks serve as infrastructure-driven callbacks—functions triggered by the underlying container orchestration system. These callbacks are sent to our backend whenever key container events occur, such as when a container is created, becomes healthy, or is terminated. The frontend then receives the corresponding status updates through WebSockets or Server-Sent Events (SSE), ensuring the interface reflects container activity immediately without the need for repeated polling requests. Each webhook carries structured information relevant to the event (for example, the container name, event type, and associated identifiers), which the backend processes before relaying to the user interface.

Security is a central consideration in implementing webhooks. Every webhook request includes an HMAC (Hash-based Message Authentication Code) signature—a cryptographic hash that verifies the integrity and authenticity of the message. The payload also contains a timestamp, allowing the backend to reject outdated or replayed requests. All webhook traffic is transmitted over HTTPS to protect data in transit. Additionally, minimal error logs are maintained for debugging purposes, with sensitive or privileged information redacted to prevent accidental exposure.

Together, these mechanisms make webhooks a reliable and secure method for synchronizing system state across components. By leveraging webhooks alongside real-time channels like WebSockets, our frontend remains responsive and accurate, reflecting infrastructure changes the moment they occur while maintaining strong security guarantees.

6.21 Secrets Management

6.21.1 Overview

Secrets management refers to the secure handling of sensitive information such as API keys, passwords, encryption keys, and database credentials used throughout an application's infrastructure. Without proper management, secrets stored in code or config files can expose vulnerabilities to attackers. Proper secrets management minimizes these risks by ensuring that secrets are secure, access is restricted, and exposure is quickly detected and remediated.

In our project, secrets management is especially important because we host intentionally vulnerable applications on an Amazon Web Services (AWS) cloud environment. Any leaked credentials, tokens, or access keys could allow malicious users to gain control of containers, the host instance, or associated AWS resources. Therefore, we must adopt proactive scanning and detection measures as part of our security workflow. After all, if the whole point of this application is to teach students about vulnerable web applications, ours has to be secure.

6.21.2 TruffleHog

TruffleHog is an open-source security tool designed to detect secrets and credentials within source code repositories, config files, and commit histories. It scans files for high-entropy strings (values that look like random keys) and known credential patterns (such as AWS access keys, GitHub tokens, or private keys). It can be used locally or integrated into CI/CD pipelines to automatically flag and block commits that contain sensitive information. This allows our secrets management to be proactive rather than reactive. We want our application to be secure from the start, rather than trying to patch vulnerabilities after they are discovered.

TruffleHog's main features include:

- **Pattern-based and entropy-based scanning:** Detects both known secret formats and unknown high-entropy strings that might be secrets.
- **Version-history scanning:** Inspects the entire Git commit history, preventing exposure from older commits.
- **Extensive detector library:** Includes hundreds of predefined detectors for cloud providers, APIs, and services.
- **Integration support:** Can be run manually or automatically in CI/CD workflows to enforce security policies before deployment.

6.21.3 Importance for WADT

For the Web App Deployment Tooling (WADT) project, TruffleHog helps prevent accidental credential leakage in our public or shared codebase. Because our team uses AWS for hosting and relies on Docker for containerization, we routinely work with environment variables, API keys, and access credentials that could cause significant damage if exposed.

Integrating TruffleHog allows us to:

- Automatically scan our Git repository before merges or deployments.
- Detect hard-coded credentials or tokens in Django configuration files, Dockerfiles, or environment files.
- Prevent compromised access to our AWS instance or container registry.
- Enforce best practices for secure development across the team.

6.21.4 Integration Plan

We plan to integrate TruffleHog as part of our CI/CD security checks:

1. **Local Pre-Commit Hook:** Developers will install a pre-commit hook that runs TruffleHog before code is committed. If any secrets are detected, the commit will be blocked until the secret is removed or replaced with an environment variable.
2. **Pipeline Integration:** TruffleHog will also run in our CI/CD pipeline to scan each pull request or branch before merging into main. This ensures that no secrets are introduced even if local checks are bypassed.

3. **Periodic Scans:** Scheduled scans of the repository history will be conducted to ensure that no historical commits contain exposed secrets. If any are found, they will be invalidated, rotated, and removed using Git history rewriting tools.

6.21.5 Best Practices and Maintenance

- All secrets (AWS credentials, Django secret keys, database passwords) will be stored in environment variables, never directly in code.
- A .env file will be used locally and excluded from version control through the use of a .gitignore file.
- Compromised keys will be immediately rotated, and TruffleHog scan reports will be reviewed regularly to verify compliance.
- Access to secrets will be limited to authorized team members only, following the principle of least privilege.

6.21.6 Summary

By integrating TruffleHog into our development workflow, we create a proactive defense layer against one of the most common and dangerous security oversights—hard-coded secrets. This integration not only protects our AWS infrastructure and project data but also helps enforce professional software-security practices within the development team.

6.22 CI/CD Pipeline

6.23 Chosen API Tools

For API implementation, some of the biggest considerations are what will potential students need to have access too and what will make this project the easiest to utilize. Along with this we also had to ensure that interfacing with Docker would be seamless to ensure that we could control the container instances while minimizing the potential risks of security issues. Due to this the project was decided to run on Python as Docker SDK for python was the most feature-rich and had the best integration with the Docker engine. With the integration this project will have much easier implementation of the core operations needed to control the containers i.e. start, stop, logs, restart, etc.

For the framework some of the biggest concerns were ensuring that authentication, admin control and security all had proper components available, which is why the final decision

was to use Django as our framework for the API. One of the biggest motivators was that Django already has built in authentication features. This allows for users to have secure registration, login, and allows the application to have session management features including user identification without having the need to build up the authorization system from scratch. Django also offers a built-in admin panel along with object relational mapper will help associate users with the containers they are utilizing, and manage user data leading to simpler database operations. REST architecture is also being used as it is the industry standard for web API.

6.24 API Considerations

Some other considerations were the usage of web-sockets for our communication protocols as they would provide continuous streaming of live logs or container stats. This was also considered due to the concern of "heisenbugs", race conditions and other overhead related issues such as our client initialization with each user who may be using the application. After research the decision to not utilize web-sockets was that the REST API would be able to handle this much better with less complications of writing and implementing these systems ourselves. The research also provided insight on Docker itself. The realization was that rather than initializing multiple clients when we have multiple users, instead initializing a single shared client would be the best option. This is because Docker already uses different request threads which will help us prevent race conditions or any "heisenbugs" that were anticipated at first. Using this Docker client management we also avoid high overhead costs as the clients are only created once the app boots up, it is inherently designed to be shared across threads while keeping user data separate, and prevents the need for database space being taken.

7 Database

When choosing our database we have a few options:

- MongoDB
- MySQL
- MariaDB
- PostgreSQL

7.1 MongoDB

MongoDB is a document-oriented NoSQL database that stores data in flexible, JSON-like objects called BSON. It's great when you need to move fast and don't want to commit to a strict table structure — data can evolve as the project does. MongoDB supports ACID transactions and comes with solid security features like role-based access control and built-in encryption. It can also scale horizontally, letting data spread across multiple servers for speed and redundancy.

For our project, though, MongoDB's flexibility was more than we needed. We wanted something with stronger relational consistency and a clear schema, since our data follows predictable relationships. MongoDB shines when the data structure constantly changes; ours doesn't. So we passed on it.

7.2 MySQL

MySQL is the classic open-source SQL database — structured, reliable, and familiar. It organizes data into tables, rows, and columns, all defined by schemas that make relationships clear and enforce consistency. It's been the backbone of web apps for decades, and for good reason.

That said, MySQL starts to stumble when handling massive datasets or tons of concurrent users. Stored procedures are clunky, and scaling beyond a single node takes work. It's good, but not great for what we're building. We needed something with stronger concurrency handling and more flexibility under pressure.

7.3 MariaDB

MariaDB is basically MySQL's smarter, faster cousin. It was forked from MySQL after Oracle bought it out, keeping full compatibility but improving performance and adding more modern features. It handles queries faster, offers more storage engines, and supports parallel replication for better scaling. It's a solid choice for web-based apps that expect high traffic.

We considered it seriously — it performs better than MySQL across the board — but PostgreSQL still beat it out in terms of handling complex queries and maintaining data integrity. MariaDB is fast, but PostgreSQL is deep.

7.4 PostgreSQL

PostgreSQL ended up being our pick. It's open-source, built for reliability, and supports everything from basic relational data to advanced types like JSONB and arrays. It follows SQL standards closely but still gives room for flexibility. Transactions are fully ACID-compliant, and its concurrency control (MVCC) keeps things smooth even when multiple operations hit at once.

Performance-wise, PostgreSQL can scale well and handle complex queries without breaking a sweat. It's secure by default, supports role-based permissions, and encrypts traffic between the server and clients. For our project — managing user data, container metadata, and system events — PostgreSQL just makes sense. It gives us structure without limiting what we can do later.

7.5 Container Identity Management

the problem of figuring out which container is which. container name and container id are both guaranteed to be unique by docker, but which one we need to use depends on context.

7.6 Docker Compose

Docker compose requires a different Python library than just single containers, which means if any webapps we want on WADT use compose, we need a way to use a different library to control them.

7.6.1 Benefits for Vulnerable App Deployment

Docker Compose is pretty much valuable when deploying vulnerable applications consisting of multiple interconnected components, such as a web server, a database, and a proxy. Compose makes it easy to spin up and manage lots of web apps, which helps us quickly test and reset vulnerable setups.

- Service grouping – Compose enables us to define and launch multiple services as a single unit. This is ideal for vulnerable app stacks that require coordinated startup of frontend, backend, and supporting components.
- Network isolation – Each Compose project creates its own virtual network, isolating services from the host and other containers. This containment reduces the risk of unintended interactions and supports controlled exploit testing.

- Volume sharing – Shared volumes allow services to store and access data that remains on restarts of containers. We use them to collect logs, exploit payloads, and forensic traces without modifying the container internals.
- Environment reproducibility – The `docker-compose.yml` file serves as a portable blueprint for the entire application stack. It ensures consistent deployment across different machines, operating systems, and CI/CD pipelines.
- Simplified teardown and reset – With a single command, we can stop and remove all services, networks, and volumes. This makes it easy to reset the environment between test runs or restore a clean state after exploitation.

7.6.2 Compose Workflow

Compose operates in three main stages:

- Definition – Services are declared in a YAML file, specifying images, ports, variables, volumes, and dependencies.
- Build and Up – The `docker compose up` command builds images and starts all services in the correct order.
- Teardown – The `docker compose down` command stops and removes containers, networks, and volumes, which allows to have a clean resets.

7.7 Docker Architecture

Docker gives a containerization framework that enables us to deploy vulnerable web applications with putting all their dependencies into isolated, secure environments. Most of the tools in Docker explains how we utilize them in our deployment pipeline.

7.7.1 Core Components

The Docker architecture consists of these key elements:

- Docker Engine – The runtime that builds and runs the containers.
- Images – Read-only files that contain everything needed to run an application. We create custom images using Dockerfiles and choose lightweight, secure base images to reduce risk.

- Containers – Containers are active instances of images. Each one runs in isolation using Linux namespaces or cgroups to safely execute and run vulnerable applications.
- Volumes – Volumes gives persistent storage for logs and payloads. We use them to examine attack traces without changing the container itself.

7.7.2 Benefits for Vulnerable App Deployment

The Docker architecture offers several advantages for deploying and managing intentionally vulnerable applications:

- Fast setup and teardown – Containers can be launched or removed in seconds, enabling rapid iteration and testing of exploit scenarios.
- Safe and isolated experimentation – Vulnerable apps run in sandboxed environments, allowing controlled execution of payloads without affecting the host system.
- Scalability for labs – Multiple instances of the same app can be launched with different configurations for parallel testing.
- Cross-platform reproducibility – Docker ensures consistent behavior across Windows and Linux, reducing platform-specific bugs and simplifying collaboration.
- Seamless CI/CD and cloud integration – Containerized apps can be easily integrated into automated pipelines and deployed to cloud platforms for scalable testing.
- Resource control – Docker allows limiting CPU, memory, and network usage, which helps simulate constrained environments or prevent abuse during testing.

7.7.3 Security Isolation

Docker uses built-in Linux features to keep containers separate and safe. Namespaces and cgroups help in isolating processes and controlling resource usage. We also improve security by removing extra permissions and applying system call filters to block risky operations. This helps prevent containers from affecting the host system. Containers also run with their own network stack, which limits exposure to external threats unless explicitly configured.

7.7.4 Container Management Process

Our deployment pipeline handles every stage of container management. We build images from Dockerfiles, launch containers with secure configurations, and use volumes in order to

retain logs and exploit traces. Containers are stopped and removed cleanly, enabling quick resets between test cycles.

8 Prototypes

8.1 ProtoWADT v0

The first prototype was created to explore controlling Docker containers programmatically and gain familiarity with Django. This prototype was timeboxed to a 1-day sprint to prohibit overanalysis and decision paralysis.

The scope was intentionally basic and focused on only programmatically controlling one container, instead of the usual 7-10. It ran local-only, depended on a specific vulnerable app (DVWA), and was intended to be thrown away.

ProtoWADT v0 was also a vital exploration of some of the invalid assumptions the team had made previously about our tech stack. Different libraries were depended on than originally predicted as a result of this exploration. This prototype also began the thinking which led to further design exploration and investigation of hidden complexity in the next prototype.

The prototype's definition of done required that at least the core states of Docker containers could be manipulated through API endpoints. Start, Restart, Logs, Stop, and Down were initially required.

A crucial oversight of this prototype was the lack of a database, which hid further problems with state and storage that were not discovered until the second prototype. The decision to store important data in memory for a local-only demo sidestepped issues of data corruption, handling asynchronous connections, and scaling that are discussed in more detail in **??** and **??**.

8.1.1 Programmatic Control

The first goal of v0 was to programmatically control Docker containers with a suitable programming language. For Python, two main libraries exist that fit this purpose: python-on-whales and docker-py. Broadly, docker-py is intended for applications that run as a single container, and python-on-whales is intended for applications that run as a group of containers with Docker Compose as well as single-container applications. Since DVWA specifically uses Docker Compose to run multiple containers, python-on-whales was selected for the prototype.

First, a short demo was created that essentially listed and used the main parts of the python-on-whales API that the project would depend on. python-on-whales has essentially a one-to-one correspondence beteween its API and the Docker CLI command structure, meaning that finding these endpoints was often as easy as typing the command with Python syntax.

```
docker compose up --detach
```

Figure 6: `docker` command syntax

```
docker.compose.up(detach=True)
```

Figure 7: python-on-whales API

The key failures of this prototype related to the scope. A standard Python program has a synchronous top-down control flow, meaning that a given sequence of method calls to a single DockerClient object in memory will occur as expected. However, a web server aiming to provide the same service cannot use this model of execution.

Instead, the DockerClient object cannot be shared in memory between REST API endpoints without mechanisms for data persistence (serialization and storage in the session, cache, database, etc.) and data integrity (preventing race conditions or invalid states).

At a deeper level, HTTP is a stateless protocol. Specific to the project, shared state between requests needs to be managed by a separate system server-side. Sharing data between views as a global variable, such as providing a global DockerClient for all views to access, is not a sufficient solution due to the potential for data corruption.

A WSGI server typically assigns each request to a worker process or thread, meaning that sharing global variables in memory would break down upon deployment. While python-on-whales guarantees that their DockerClient object does not store any state that will affect program execution, docker-py does not make a similar guarantee. In the end, the potential for the "share global variables in memory" approach is bad design which may or may not work depending on the Docker library used.

8.1.2 Django

The second goal of v0 was to understand how Django is used to build webapps. A basic demo site was created with views to control the DVWA container. The completion criteria involved creating URLs for each Docker action and writing a view to achieve each. Completion was

manually tested by monitoring container state with Docker Desktop while using the written views. Start, restart, stop, down, and logs views were required.

Each of these endpoints was simple due to the lack of complexity of the scope of the prototype. More complex requests for continuous information like container CPU usage were not considered. As a result, the prototype v0 didn't reveal much about how to write the views of our project, but did teach the team how to structure a Django project overall.

Gaps in the team's understanding, resulting from an insufficiently detailed prototype, were (partially) corrected by the second which was developed.

8.2 ProtoWADT v1

Crucially, the second prototype included a database. This inclusion revealed a series of issues with the initial design of the project that needed to be corrected to reach our MVP.

A database was developed for the prototype with an accompanying ERD that was sent to the team.

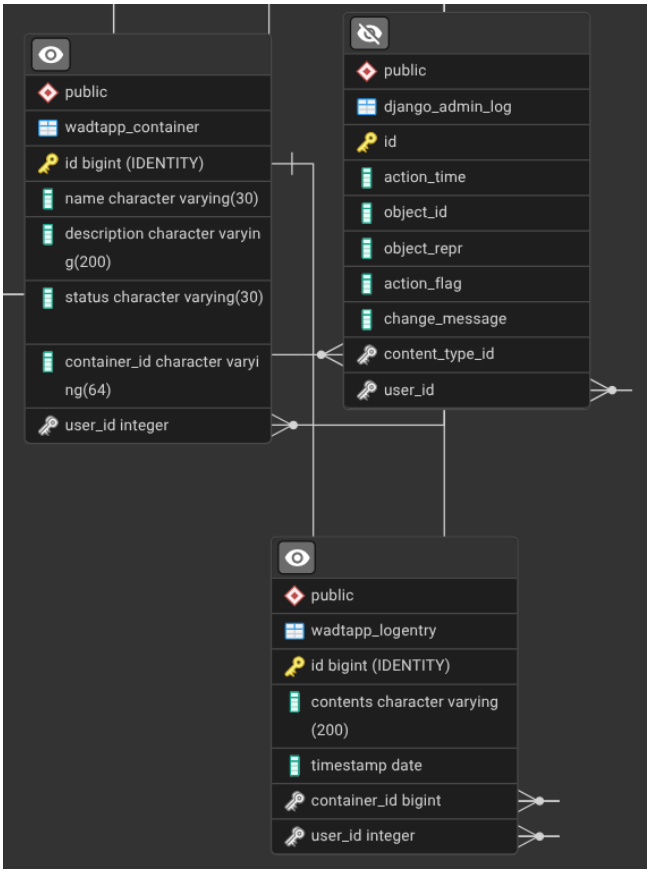

Figure 8: Relevant portions of prototype v1 ERD

9 Frontend

9.1 Frontend Framework

9.1.1 React

The front-end of our application is built using React. React employs a virtual Document Object Model, or DOM, to more efficiently manage updates to the UI. Only the components that have changed are updated, rather than needing to re-render the entire page to make changes. This makes the application faster and more responsive, which is an important quality for any web application.

React provides several benefits that we found attractive:

- Reusability: Components can be reused across different parts of the application, improving consistency and helping us avoid the DRY principle as much as possible. This will also make it easier for future development teams to extend or modify the project.
- Flexibility and Maintainability: Individual components can be updated or replaced without affecting the rest of the system, allowing for simpler debugging, testing, and any future changes.
- Organization and Readability: React promotes self-contained components, leading to more structured and readable code. This will make it much easier for future us to understand code if we need to refer to it at some point in the future.

9.1.2 Drawbacks of React

While this React offers many benefits, it does present some challenges that we have to consider. According to our research, large applications can become quite messy if the different components are not managed and organized carefully. However, given the relatively limited scope of our project and our attention to good design practices, we do not expect this to be a significant concern. Also, the emphasis on reusability can sometimes limit customization, but for our needs, we find the benefits of React far outweigh these potential drawbacks.

9.1.3 React-Bootstrap

In order to speed up and simplify the design and process, we incorporated React-Bootstrap, which is a library built on top of Bootstrap 5. Our sponsor is already familiar with Bootstrap, so adopting React-Bootstrap also gives us an additional source of guidance and expertise if needed during development.

9.1.4 JavaScript vs. Typescript with React

React primarily uses JavaScript, but it also supports TypeScript, which we chose for this project. TypeScript extends JavaScript by adding some additional support which offers several advantages.

- Compile time error checking. Catches errors at compile time rather than runtime, speeding up the process of finding and fixing bugs.
- Improved readability: Type annotations make the code easier to understand for both our team and others in the future.
- Easier maintainability: As the project grows, TypeScript will help maintain consistency across components and reduce the risk of introducing hard to find errors.

9.2 CSS Libraries

When it comes to frontend work, part of the problem is making sure that a website looks appealing enough to attract users, providing a more enjoyable coding experience for the developer and ensure pleasurable user experience.

With this in mind, we wanted to choose a professional CSS library that provides the user with good readability and accessibility. There were a lot of options for CSS libraries, but we narrowed the options down to three: Bootstrap, Tailwind and Chakra UI.

9.2.1 Bootstrap

This one was our first consideration, as bootstrap is a very popular CSS library. Bootstrap and REACT have their own merged CSS library that makes coding the project smoother called 'REACT-Bootstrap'. REACT-Bootstrap utilizes the bootstrap visuals with REACT type programming to ensure a coherent coding experience.

Bootstrap's reviews are overall positive, with extensive praise on the use of child themes. Its REACT-Bootstrap also gets pretty high reviews, being praised for it's developer friendly nature. The only major complaint on Bootstrap is it's lack of flexibility, requiring the developer to edit the CSS code themselves to achieve the desired look.

There are also custom themes for bootstrap to use that can be found online.

One example is 'Bootswatch', which provides free themes for bootstrap that can be easily implemented into the project. Utilizing these themes can help us achieve a more unique look for our project, and possibly implement different themes for the user to choose from.

9.2.2 Tailwind

Tailwind is another popular CSS library containing many good options for CSS design. Through research, we found that tailwind operated very differently from bootstrap, as it is a utility-first CSS library. This means that instead of having pre-designed components like bootstrap, tailwind gives a lot of freedom to the developer to design their own components using CSS.

Tailwind reviews are good, with 63% of the reviews on their website being 5 star. A complaint that we have noticed when it came to tailwind is that while it is versatile and useful, there is a learning curve. It suggested that those who don't have extensive knowledge on CSS will struggle a little to utilize tailwind's components.

Unlike Bootstrap which has it's own REACT library, tailwind does not have one. Instead, tailwind has it's own third party library called Tailwind UI that integrates tailwind with REACT. However, this third party library is locked behind a paywall, requiring the developer to pay to use it. Our research concludes that tailwind is extremely versatile for CSS design, however requires a learning curve.

9.2.3 Chakra UI

Chakra UI also came across our radar as a popular CSS library. On Chakra's website, there was a page that showcased the professional websites made using Chakra UI. This was a nice way to look at what a fully developed website using Chakra UI can do.

Chakra UI's reviews are mostly positive, with emphasis on good developer experience. A downside to Chakra UI is a lack of certain components (one example is the search bar component), as well as good ways to change the themes of the components.

Chakra UI is also developer friendly with Figma with their Chakra UI + Figma Kit. We found it interesting that there was a Figma page that showed us exactly what to use for certain components. This was a nice touch that made us consider Chakra UI more seriously since we used Figma for our design mockups.

9.2.4 What we chose

After weighing all considerations, our team selected **REACT-Bootstrap** as our primary CSS framework. It offers a familiar structure, excellent documentation, and seamless integration with React, enabling us to focus on functionality without getting bogged down in complex setup processes.

Additionally, having a developer with prior Bootstrap experience allowed us to accelerate onboarding and maintain consistency across our components. While Tailwind and Chakra

UI both provide unique advantages — creative freedom and modern theming, respectively — React-Bootstrap strikes the most effective balance between stability, usability, and developer productivity.

9.3 Themes

Since we decided to use React-Bootswatch for our CSS library, the goal was to find themes that actually fit our project's personality. We wanted a setup that not only looked clean but gave users the option to tweak the website's vibe to their liking. We ended up utilizing bootswatch, a theme library made for bootstrap and can be utilized with react-bootstrap. Bootswatch made that easy—it's built for Bootstrap, works smoothly with React-Bootstrap, and comes loaded with theme options we could build around.

Our design direction aimed to capture a sleek, "hacker-style" aesthetic—something that felt sharp and modern without overcomplicating the visuals. The frontend team tested several themes that balanced readability and contrast while keeping the interface lightweight.

Here are some theme designs that we decided on (note subject to change until due date)

- Bootswatch Cyborg:
- Bootswatch Lux:
- Bootswatch Zephyr:
- Bootswatch Pulse:

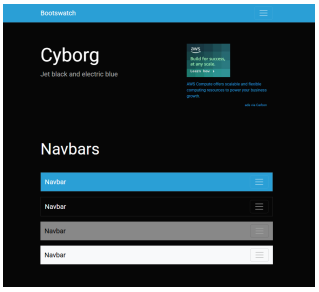

Figure 9:

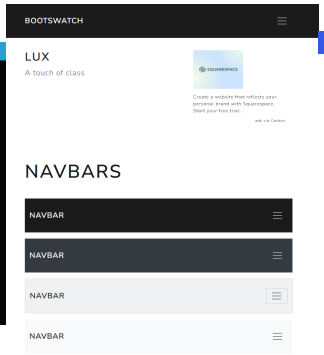

Figure 10:

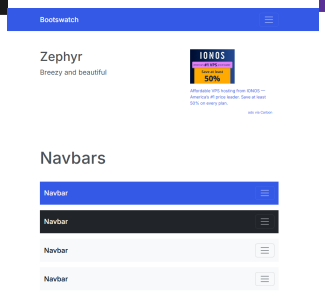

Figure 11:

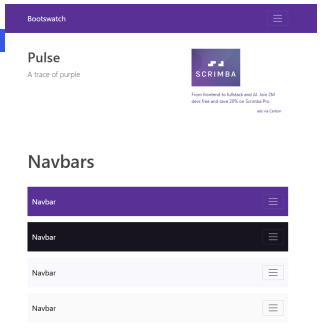

Figure 12:

Since Bootstrap isn't exactly known for wild creativity, we relied on Bootswatch to inject variety while keeping the codebase clean. When a specific design wasn't available, we customized the Bootswatch SCSS files to match our desired color palette and maintain a consistent style across components.

9.4 Wireframing

Wireframing is an essential component of front-end development; it serves as the blueprint from which all design and implementation decisions follow. Like all blueprints, its primary goal is to serve as a medium of communication. Communication is the foundation of any collaborative effort, and wireframing provides the first stage of communication for the front-end team. It is important that the frontend team uses a tool that will allow us to work together on designs to ensure we are on the same page before implementation. It is much easier to tackle a large problem with a clear objective in mind prior to implementation. This will improve efficiency and lead to a better overall result. It is also important that we have the flexibility to make changes and add to designs as we progress further into the project and gain a better understanding of our end goals.

While we are the first group to work on this project, it is entirely possible that future Senior Design teams will continue our work. Therefore, documenting and wireframing all of our designs serves as an important organizational tool, not only for our current team but also for any future teams to draw inspiration from or build upon our designs.

9.4.1 Decision Criteria

There are many wireframing tools available, so it was important to establish clear decision criteria to guide our final decision. Since this project focuses on building an educational web application, practicality and clarity were our main priorities rather than highly complex or flashy design features. The major factors we considered included:

- Ease of Use. In order to begin implementation as soon as possible, we prioritized researching tools that are intuitive, beginner-friendly and have a well-established resource for working with the tool.
- Collaboration Features. The ability to work together both in real time and asynchronously through sharing feedback easily was essential. Effective collaboration ensures smooth communication across the team and aligns with the overall goal of efficient group work.
- Cost. As students, we aimed to minimize additional expenses and preferred tools that offered strong free versions or educational discounts
- Flexibility. We wanted tools that could support our needs without requiring excessive plug-ins or extraneous tools to work sufficiently.

9.4.2 Figma

Figma was the first technology we considered, as it is one of the most widely used and well-supported wireframing and UI design tools available today. A notable feature of Figma is its emphasis on real-time collaboration. Multiple team members can work on or view the same design simultaneously, which allows for instant feedback and streamlined communication. Figma's commenting feature also makes it easy to communicate asynchronously, which is very helpful as scheduling meetings at the same time can be difficult throughout the week.

Sharing designs is also extremely easy with Figma, as links can be shared with anyone, including non-front-end team members, for feedback or review of designs. Figma also offers a large collection of community resources, like templates, plug-ins, and tutorials, which help users get acclimated to the software quickly. While Figma is capable of supporting more high-level designs, its basic functionality is straightforward and practical for our project.

Our main concern with Figma was its potential cost. While there is a free version, we were unsure if the free version would be overly limiting. Fortunately, we found that the free plan provided all the core features we needed for the project. The paid version, priced at around 15 dollars per month, could be considered later if premium features became necessary. The month-to-month payments would be manageable if deemed necessary. Figma also works well on all major operating systems, so it would be easily accessible for everyone on the team. Overall, Figma met nearly all our criteria as it is collaborative, user-friendly, and cost-effective given our current scope.

9.4.3 Canva

Another tool we considered was Canva. Although Canva is not a traditional UI/UX design platform, it can be used effectively for wireframing, especially for projects with simpler design requirements, such as this. Canva's biggest strength is its ease of use. It has an easy-to-use drag-and-drop interface. It also has a large variety of templates, icons, and pre-made design elements, making it extremely approachable for beginners. This allows team members to quickly create clear layouts without needing extensive design experience with the tool.

Canva also supports collaborative editing, where multiple users can work on a design at the same time and leave comments similar to Figma. Files are stored in the cloud, making them easily accessible and shareable across different devices, giving it a great amount of flexibility. Canva works on all major operating systems through most web browsers, and it also offers desktop and mobile apps for added flexibility.

In terms of cost, Canva offers a free version that includes most core features, while the premium version costs 13 dollars per month, making it the cheapest of the options considered,

if premium features were deemed necessary. Canva also provides educational access for free to students, which could make it even more accessible for our team.

However, Canva does have a major limitation in terms of flexibility. Its heavy use of pre-made templates, while convenient, can be restrictive when trying to create or experiment with more complex designs. This could become a drawback if future teams wish to implement more advanced UI layouts in the future. Despite this, Canva's simplicity and focus on clarity make it a strong option for an educational web application like ours, where functional, easy-to-understand designs are the top priority.

9.4.4 What we chose

After evaluating the most practical tools for our needs, our team ultimately selected Figma as our primary technology for design prototyping. It offered the most balanced combination of collaboration features, accessibility, and functionality. Allowing our front-end team to efficiently plan, iterate, and communicate interface designs.

Figma's collaboration tools, and ease of sharing designs and information made it especially effective for the needs of this project. Its easy-to-use interface lowered the learning curve for members who were new to working with the tool. Additionally, its extensive library of community resources and templates enabled us to quickly build and refine wireframes that align with our project's goals and communicate the front-end vision with the rest of the team.

9.5 Figma Designs

Below are several of the mock-up pages designed in Figma along with a discussion of their purpose within the application. While these designs are subject to change as implementation of the project progresses, the goals and requirements for each page will likely remain the same.

9.5.1 Login Page

A login page is necessary as both students and instructor users will need different levels of access and capabilities. User accounts will also allow students to save their work and return to it later. Although containers will automatically stop after 24 hours, this approach will provide students the flexibility to work in intervals rather than needing to complete everything in a single session.

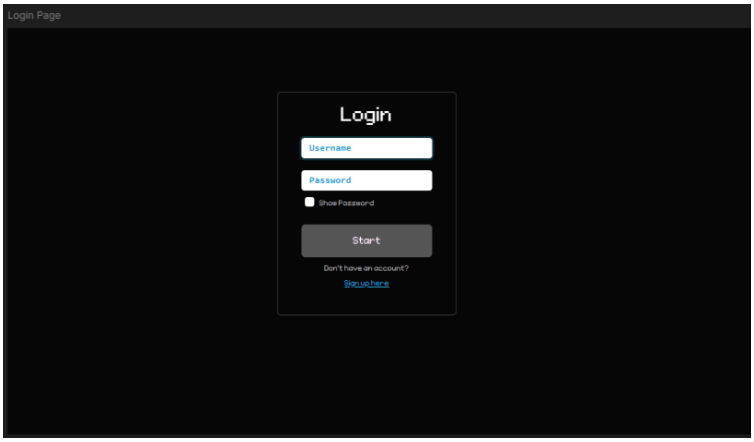

Figure 13: Example login page design

9.5.2 Vulnerable Applications Dashboard

Students need a clear and easily accessible way to work with vulnerable applications. Therefore, the webapp will have a dashboard which will contain all available vulnerable applications. Each listed application will include a description of the application alongside the relevant skills that are being tested when attempting to patch any vulnerabilities. The front-end team will work with the sponsor and docker team to rank the difficulty as well.

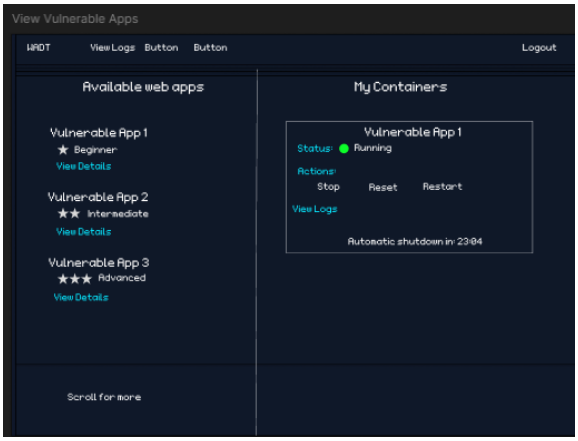

Figure 14: Example dashboard design

9.5.3 Container Logs

The container logs page is crucial, as it will provide all of the information needed to resolve any issues related to any relevant container actions or events. Logs will be maintained for the entire lifespan of each container until it is stopped manually or automatically. Since feedback is critical for an educational tool, the logs page must present information clearly and simply.

Only relevant troubleshooting information will be included to avoid overwhelming the user.

Figure 15: Example Container Logs

The designs shown above represent only some of the pages created throughout the planning of this project. Over the course of development, many of our original ideas have been adjusted, or expanded as we learned more about the requirements and how each page would function within the application. Figma has been especially helpful in this process, as it allows us to quickly draft new ideas and revise our designs without needing to commit to coding right away. While not every design can be included in this document, the mock-ups we have shown highlight the core pages necessary for the application and reflect the overall direction of the project.

10 Ethical Concerns and Considerations

WADT will host intentionally vulnerable applications, which come with its own unique ethical and security challenges. We have a responsibility to protect our users, and systems from foul play and misuse. The most prominent ethical consideration for this project re-

volves around limiting potential harm to protect our users in order to promote responsible cybersecurity education.

10.1 Data Privacy

WADT serves as an educational tool for students, rather than as commercial application, therefore it is unnecessary to store significant amounts of personal information. The only personal information needed will be a username and password. Other information such as credit card information, address, or other personal contact information will never be stored by WADT. Additionally, logs will exclude any identifiable information beyond what is required for container management system audits. We will store minimal information to ensure that if an account is compromised for any reason, there will be minimal impact for the user.

10.2 Responsible Handling of Vulnerable Applications

We will host vulnerable applications with exploitable vulnerabilities by design. Given this, we have an ethical responsibility to ensure these vulnerabilities cannot be exploited and used for harm. These applications will be used only for educational purposes within a controlled environment. In order to prevent unethical use of our systems, we will have strict controls to mitigate any risks that come with such hosting. Our AWS instance will not be publicly accessible and will only accept or send out traffic to a small number of white-listed IP addresses for the intended students and instructors. All vulnerable applications will be deployed using OCI-compliant container tooling and images to further reduce risk of exploitation of a contained vulnerability. We will also implement automated shutdowns of containers to reduce exposure and mitigate the ability of vulnerable containers to interact with external environments.

10.3 Information Logging

Our application will maintain logs of container actions for both students and instructors such as deployment, stoppage, and resetting. These logs will be limited only to container relevant events or actions. These logs will serve two primary functions.

- Feedback. Provide clear feedback to students on their interactions with containers.
- Oversight. Provide visibility to instructors into container usage and potential misuse.

Our logs will only capture information required for necessary functions and will exclude excessive information such as private user information, or unrelated system information.

Logging will only be used as a tool for debugging, learning, or ensuring proper usage

11 The "WADTtastic" Future In Store

The WADT project is the team's opportunity to support cybersecurity education and make the greatest of all makeable things, a useful tool. We want to see the project revolutionize the workflow of our target market and make jumping into a lab as quick and easy as one-click shipping.

Tedium is the worst thing that smart people can be subjected to, and WADT gets rid of some of it for students and the instructor. We hope that by making it easier to get started, students can retain their motivation for the problems they're really after.

We also hope that in creating a free and open-source tool, our same users will become our contributors as they envision new changes to WADT that we couldn't have imagined ourselves. We have strived to create a maintainable codebase with low technical debt, so that future Senior Design groups or users may pick up our project again. Ideally, others may use what they have learned to make something neat, as we are doing.

And rounding up our hopes for WADT, the most important vision has come from our project sponsor. We know that he will put our MVP in the hands of real users and appreciate the objective which he has given to the group. We have built up a great deal of confidence in our sponsor as a long-term project owner who can carry the project forward to future teams of developers, or develop it further into a more available webapp that can serve more users.

12 System Architecture

12.1 Use Case Diagram

The use case diagram illustrates the interactions between the two primary actors, students and instructors, inside the web application system. Students can browse available vulnerable web applications, deploy containers with automatic 24-hour timeouts, and manage their containers through stop, restart, and reset operations. Students also have access to timestamped logs specific to their container activities. Instructors inherit all student capabilities, but have additional privileges which include, access to more comprehensive instructor logs, the ability to view statuses across all student containers, disabling containers to restrict student access, and overriding student actions with race condition prevention.

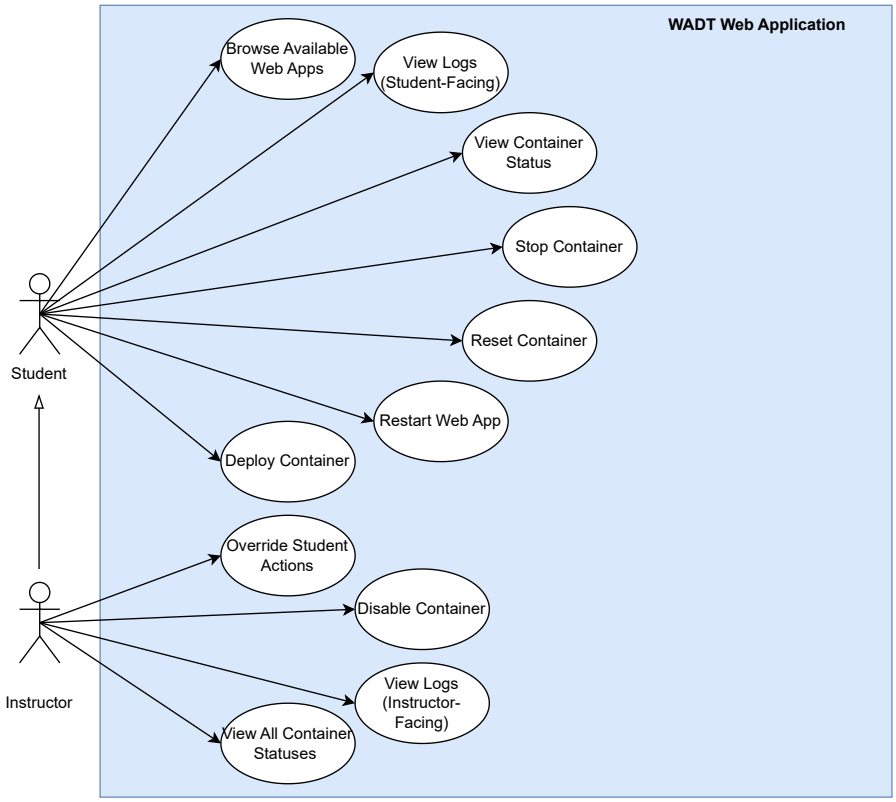

Figure 16: WADT Web Application Use Case Diagram

12.2 Deploying Containers

The activity diagram details the process behind deployment of a container, from catalog selection through backend processing.

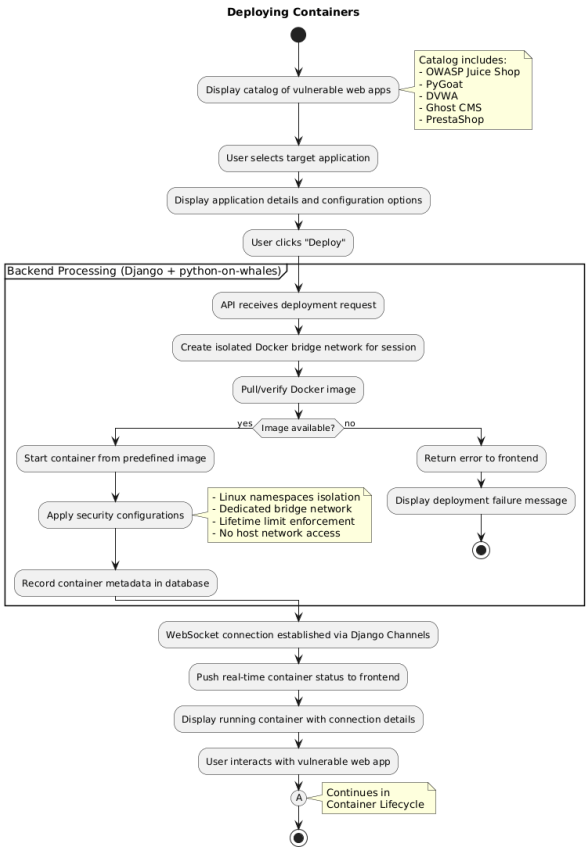

Figure 17: Container Deployment Activity Diagram

The backend applies security configurations automatically during deployment, ensuring each container runs in an isolated network with enforced lifetime limits. If an image is unavailable, the system returns an error rather than proceeding with a partial deployment.

12.3 Interactions with Containers

The figure below illustrates the management actions available to users during an active session.

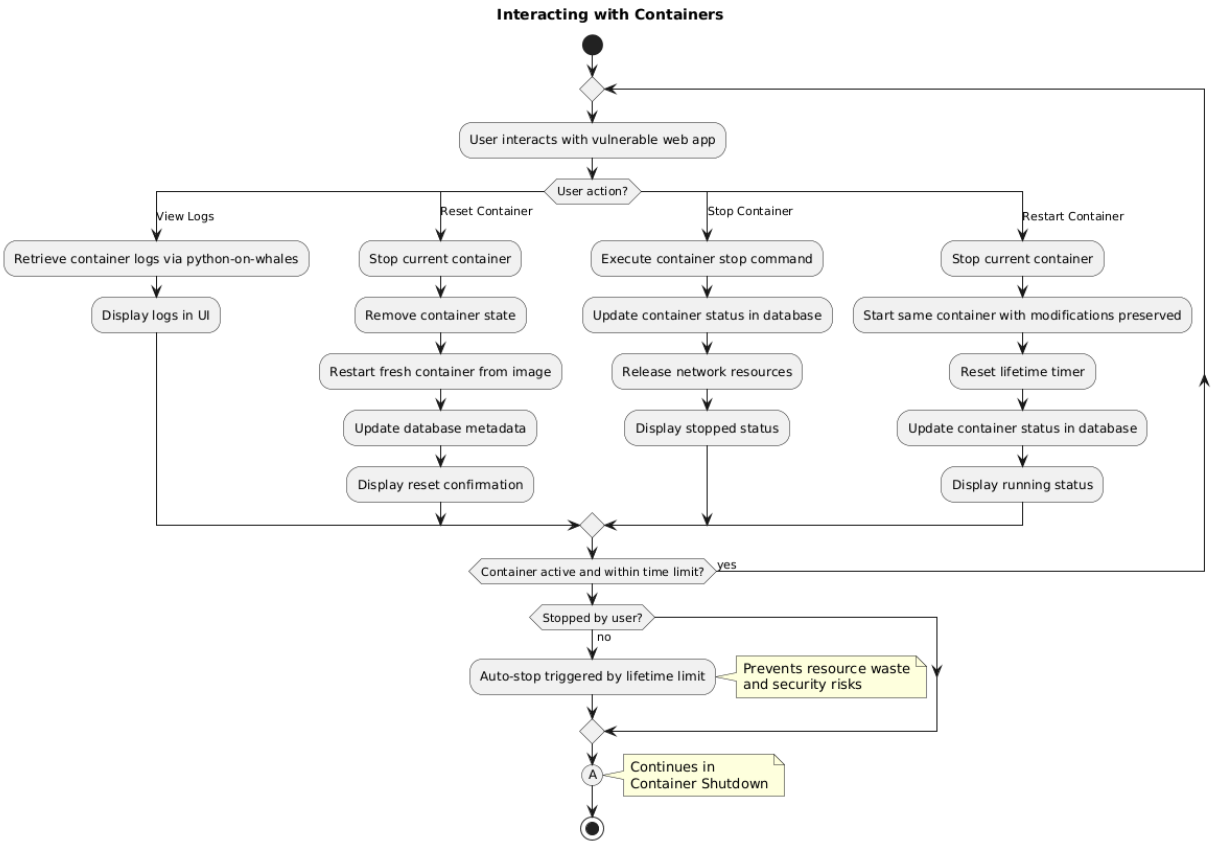

Figure 18: Interacting with Containers

During an active session, users can view logs, reset to a clean state, restart with any changes made preserved, or stop containers entirely. The system monitors lifetime limits in the background and triggers automatic shutdown if a container exceeds its allocated time.

12.4 Container Cleanup Process

The following figure describes the cleanup process that occurs when a container session ends.

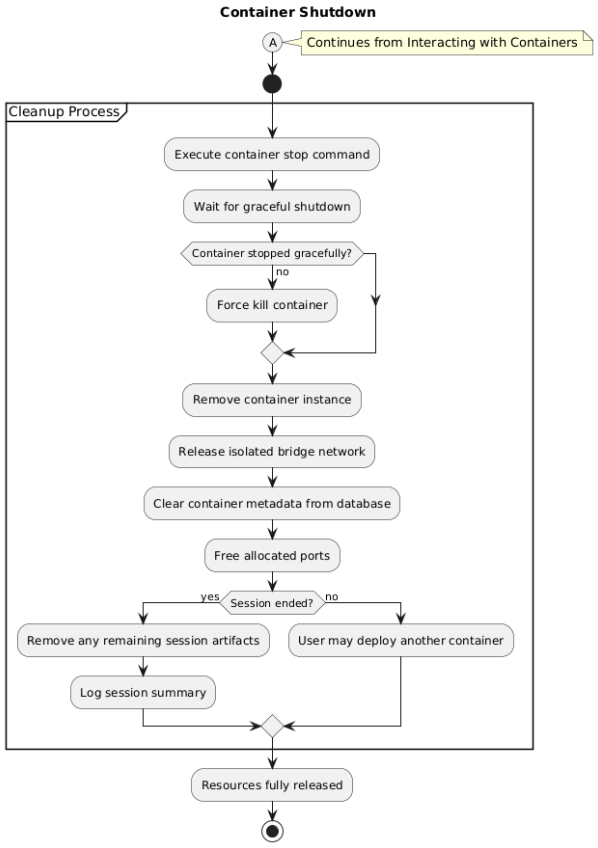

Figure 19: Container Cleanup

The system attempts a graceful shutdown before forcing termination if necessary. All associated resources, network bridges, database metadata, and allocated ports are released to maintain system integrity for subsequent deployments.

13 Timeline and Milestones

13.1 Project Timeline

The following timeline outlines the weekly tasks and objectives throughout Senior Design I and II.

Senior Design I

- **Week 1 (Sept 9 – Sept 15):** Meet team, contact sponsor, set up communication channels

- **Week 2 (Sept 16 – Sept 22):** Meet sponsor, write Team Contract, assign roles, and gather requirements
- **Week 3 (Sept 23 – Sept 29):** Set up Jira, conduct research on Docker integration, database, and security concerns
- **Week 4 (Sept 30 – Oct 6):** Write Early Status Report and present research to sponsor to set up AWS instance
- **Week 5 (Oct 7 – Oct 13):** Set up GitHub, create initial UI design mockups, and start Preliminary Design Document
- **Week 6 (Oct 14 – Oct 20):** Continue working on Preliminary Design Document, start initial coding and setting up installations, packages, and organizing the codebase
- **Weeks 7–8 (Oct 21 – Nov 3):** Finish Preliminary Design Document, create additional UI mockups for the rest of the pages, and get initial API endpoints working
- **Weeks 9–10 (Nov 4 – Nov 17):** Implement UI for base website, finish APIs for base website, and work on Final Design Document
- **Week 11 (Nov 18 – Nov 24):** Final Design Document and plan work that needs to be done between semesters

Senior Design II

- **Week 1 (Jan 12 – Jan 18):** Reconvene with sponsor and review SD1 feedback
- **Weeks 2–3 (Jan 19 – Feb 1):** Begin implementation of container management features (deploy, stop, restart, reset) and establish stable frontend-backend integration
- **Weeks 4–5 (Feb 2 – Feb 15):** Deploy and test first vulnerable web application, validate UI interactions and backend functionality
- **Weeks 6–8 (Feb 16 – Mar 8):** Expand system to include all planned web applications, begin integration testing and security validation
- **Weeks 9–10 (Mar 9 – Mar 22):** Performance testing, get feedback from sponsor, implement stretch goals
- **Weeks 11–13 (Mar 23 – Apr 12):** Implement instructor features, final touches, and work on Final Design Document
- **Weeks 14-15 (Apr 13- Apr 27):** Final presentation and Design Document

13.2 Project Milestones

The following milestones represent the major deliverables and functional achievements of the project. Reaching milestones will help us ensure we reach our Definition of Done and have all of the required features and functions.

Senior Design I

- **Oct 3:** Submit Early Status Report
- **Oct 31:** Submit Preliminary Design Document
- **Nov 17:** Build functional web app with basic features implemented
- **Nov 24:** Submit Final Design Document

Senior Design II

- **Feb 8:** Container management system fully implemented with fully working deploy, stop, restart, and reset functions, container resource limits enforced and accurate status information
- **Mar 1:** First vulnerable web application deployed and fully functional
- **Mar 22:** All 7 web applications integrated
- **Apr 26:** Instructor features implemented and system testing
- **Apr 27:** Final presentation and Final Design Document delivered

14 What's Next?

Our group is the first to work on the Web App Deployment Tool, and while the project has a relatively small set of core features, there is a lot of room for additional features. Once we begin testing the application and receiving feedback, the relevance of additional features will become clearer. By the end of Senior Design II, we will have a functional website with all of the core features, but the project is designed so that future groups can continue building on what we have created. As implementation progresses, we plan to add extra features whenever possible, based on feedback and any additional ideas that emerge as the application becomes more complete. This will create a strong foundation for later groups to extend.

Cybersecurity education evolves constantly, especially as new vulnerabilities, technologies, and industry practices emerge. With that in mind, there is a lot of potential to expand

the catalog of vulnerable applications beyond the initial seven we will include. Future groups will be able to update the tool to introduce new applications, new vulnerability types, and new skills for students to practice. Our team is essentially starting the process and establishing the structure, leaving plenty of room for refinement as more students and instructors interact with the application. While we will make updates based on early feedback, the most meaningful insights will come from long-term use in educational settings. Over time, future teams together with the sponsor will be able to use that feedback to set new goals and evolve the project further.

There are also additional features outside our core requirements that later teams could explore. One major possibility is adding automated checks to determine whether a vulnerable application has been successfully patched. Even if this isn't part of our team minimum viable product, another group could create a companion tool to evaluate student work more thoroughly. Our focus is on giving students the environment they need to work on vulnerabilities, but a complementary system that provides automated feedback without requiring instructor involvement would make the tool even more useful. The overarching goal of this project is to make the learning process more efficient, and giving students fast, and thorough feedback would help them learn more independently. We are supplying the core environment, but there is significant room to elevate the project by expanding the amount of guidance, assessment, and feedback available to students.

References

[1] *Introduction to Celery*. URL: `https://docs.celeryq.dev/en/stable/getting-started/introduction.html`. (accessed: 10.31.2025).

[2] Gabriel de Marmiesse. *Python on Whales*. URL: `https://gabrieldemarmiesse.github.io/python-on-whales/`. (accessed: 10.31.2025).

[3] Podman: Managing pods and containers in a local container runtime. *Brent Baude*. URL: `https://developers.redhat.com/blog/2019/01/15/podman-managing-containers-pods#`. (accessed: 10.31.2025).