Whaler: statically analyzing Docker Compose configuration files

Tristen Nollman
School of Computer Science
Stanford University
Stanford, California 94305
Email: tnollman@cs.stanford.edu

Olayinka Adekola
School of Computer Science
Stanford University
Stanford, California 94305
Email: oadekola@stanford.edu

***Abstract*—Docker is a tool that is widely used for self-hosting - indicative through the hundreds of millions of weekly downloads for self-hosted services on DockerHub. Our research aims to assess the security posture of Docker for self-hosting. Our objective is twofold: to determine if individuals are securely deploying Docker containers and, if not, to provide a tool that allows them to understand how their own Docker configuration may be vulnerable along with steps they can take towards creating a more secure self-hosted environment.**

Our overarching goal, is for people to have the ability to leverage the positive aspects of using Docker (portability, reproducibility, and isolation) in a way that keeps their sensitive information secure.

1. Introduction

Docker is an exceptional tool that allows users to deploy containerized applications, nearly independent of their computing platform. Consequently, people have been increasingly using Docker for self-hosting - wherein they leverage Docker's versatility for hosting various services ranging from multimedia streaming and mesh VPNs to identity & access management and file-synchronization among numerous others categories.

However, Docker has default container configuration settings that while effective for Docker's original purpose - building, testing, and deploying applications quickly - raise significant security concerns when used for self-hosting purposes. This is of particular concern for the self-hosting community because often times the self hosted data consists of personal information or grants access to a user's host platform which typically is less security-oriented than a production environment at a company. There is simply less structure required for people self-hosting to actually deploy their services than there is for maintaining a complex ecosystem of production systems at a company or organization.

Given this, a victim can unknowingly deploy misconfigured, vulnerable containers that leave them susceptible to attack. An attacker could gain access to a victim's personal data, gain control over their containers, or even gain control over their host platform and enact harm through manipulating, deleting, or ransoming their personal information.

We suspect that even though Docker configuration risks are known and documented there is a gap between acknowledged risks and actual deployment practices leading to people deploying vulnerable Docker containers. This could be due to a number of reasons: the technical acumen needed to understand Docker's documentation, or a general lack of awareness of Docker's documentation, or a misunderstanding of the severeness of a particular configuration option. Moreover, in an effort to simply get a self-hosting setup deployed, people may choose the straightforward, default option rather than a more involved strategic and secure alternative. We think discerning this reason is important but isn't in the scope of this paper.

In order to empirically evaluate the prevalence of people deploying vulnerable Docker containers, we created a tool that analyzes a Docker Compose configuration file and reports the number of potential configuration vulnerabilities. We use this tool to evaluate the configuration files for a selection of the top one hundred popular self-hosted images. Then to gain a larger scope of the space, we use the tool to analyze over three thousand Docker user configuration files from GitHub repositories associated with the top one-hundred images. We confirm that people are indeed creating vulnerable configuration files that rely on Docker's default container configurations settings, as well as Docker Compose's default behavior.

It seems unlikely that Docker will change their default container configuration settings to address any of these configuration issues since the needs of all Docker users will not overlap significantly depending on what services are being deployed. Thus we extend our measurement tool to provide understandable warnings for a given Docker Compose file, as well as some sensible solutions that could work in a large percentage of use cases for self-hosted services. While this won't directly fix the misconfiguration vulnerabilities that people run into, we maintain that this can be a first step towards people deploying Docker containers for self-hosting in a more secure way.

2. Background

2.1. Docker

Docker is a platform for developing, shipping, and running applications in containers. Containers are lightweight, portable, and self-sufficient units that encapsulate all the dependencies and configurations needed to run a software application. With this, Docker also provides various levels of isolation via features from the Linux Kernel such as namespaces and cgroups. Some types of isolation Docker provides are: process isolation, file-system isolation, network isolation, resource isolation, and dependency isolation, all of which are key benefits of using Docker when deploying services.

Aside from isolation, Docker also makes it easy and fast to deploy services. If a container has a problem, it is easy to simply restart the container, or even have multiple instances of a container image running simultaneously. Since each application is containerized, there is a high-level of portability which make migrating environments much easier than if an application was installed 'bare-metal'.

However, there are some things that are concerning for self-hosting:

- By default Docker (when used with Docker Compose as is commonly done) will put your containers on a default network allowing for inter-container communication which undermines the idea of isolation for each container. If one container becomes compromised due to a vulnerability in the application, or in Docker itself, then the attacker can easily find out what other containers are on the same network, which by default will be all other containers defined within the Docker Compose file.
- By default Docker will effectively bypass any IP-table based firewall such as the commonly used UFW (Universal Firewall). This is due to the fact that when a container receives a port mapped from the host, it will create a NAT rule in the NAT table. Since NAT rules are triggered prior to IP Filter rules, this means an outside observer can see any port that is mapped into a container despite a user's firewall claiming to deny all incoming connections. This is documented in the Docker documentation, and is easily verifiable with some simple port scans. This is a big issue since users may operate under the assumption that their firewall is indeed denying all incoming connection requests, but in fact incoming connections to any port mapped into a container will appear as open to an outside observer and allow connections to be made.
- By default the Docker daemon will run as the root user. By extension, this means by default containers will run with their own user set as root within the container, meaning anyone with access to the daemon, or one of the containers, could effectively have root access to -at least- parts of the host system. This is because of how docker is installed by default on Linux. While there is an option to install Docker in a 'root-less' configuration, it is not well supported and documentation is sparse. We therefore believe it is unlikely the 'average' user will install it this way for self-hosting. The default option will likely remain the most popular and well supported option.
- By default Docker will give a container read and write access to any volume mapped into the container. This is definitely convenient, but violates the principle of least privilege, as well as potentially undermining the file-system isolation that Docker is supposed to bring to the table. Coupled with the previous behavior of running containers as a root user, this could lead to catastrophic consequences for users self-hosting valuable information using Docker.

In section 4.1 we explain further how and why these manifest as potential vulnerabilities when using Docker (and Docker Compose) for self-hosting.

2.2. Self-Hosting

Self-hosting refers to the practice of hosting and managing your own online services, applications, or infrastructure instead of relying on third-party providers. In a self-hosted environment, you take on the responsibility of setting up, configuring, and maintaining the necessary software (and sometimes hardware) to run the services you need. This can include websites, email servers, file storage, communication tools, or any other online application.

People have many reasons for wanting to self-host services today. A non-exhaustive list includes enhanced privacy, control over infrastructure and data, customization, as well as learning opportunities. Self-hosting enables users to take back control of their computing from companies offering services instead of software. The difference between a service and software in this context is that software is something potentially written by someone else which you can run on your own (if it is free software), while a service is something provided to you but ultimately controlled by someone else [1].

While self-hosting may sometimes include maintaining your own hardware infrastructure, the wide availability of cloud-hosting providers has made it easy for people to deploy their own services on other people's hardware. This opens up many possibilities for users, but also creates a new landscape for securing these services since some assumptions may no longer hold. One example may be that a user's home router has port-forwarding off by default, and thus the machine the service is deployed to is inaccessible from the public internet. This may not be the case when services are deployed on a cloud provider's infrastructure such as AWS or Linode, and may even vary from provider to provider. Therefore, for the purposes of this paper we address the issues with the assumption that users are hosting their own services using cloud-infrastructure.

A non-exhaustive list of popular services to self-host include:

- Multimedia streaming
- Remote (”Cloud”) storage
- File backups
- VPN Services
- DNS Services
- File synchronization
- Document management
- Photo galleries & backups
- Personal libraries
- Federated identities (’Fediverse’)
- Version control systems (git)
- Personal ’wiki’ websites
- Budgeting & accounting software

2.3. Clarifying Terminology

We make repeated use of terminology borrowed from the Docker documentation, as well as general self-hosting terminology. To prevent confusion, we define many of these terms here.

An image is is a template with instructions for creating a container. A running container is a single instance of the image it is built from.

A container is a runnable instance of an image. It can be created, started, stopped, moved, or deleted. A container is defined by two things: the image it is built from, as well as its configuration options provided when it is created or started. These options include any port mapping, volume mapping, network creation, etc that needs to be done. This is most of what is inside a Docker Compose file.

A volume is a layer of persistent storage that containers may be provided via their configuration. This storage type is managed completely by Docker, and are therefore not dependent on the host machine’s directory structure and OS the way that bind-mounts are. This is the preferred method of persistent storage according to Docker’s documentation [2].

A bind-mount is an alternative method of persistent storage with Docker. This method maps a directory from the host machine to the given container. These are dependent on the host OS and its directory structure, and is therefore not managed completely by Docker.

A configuration file in this paper is simply a Docker Compose file. A Docker Compose, or hereby configuration file, is a structured file which defines potentially many containers. The majority of the configuration file goes to setting various configuration options, such as port mapping, volume or bind mounts, network options, image selection, etc.

A service is simply a container with a name. Services are used in Docker Compose files, and is essentially just a list of containers (which are themselves just a collection of configuration options, and an image).

3. Related Work

There has been a vast amount of research done focusing on Docker. A significant amount of this research has focused on evaluating Docker’s effectiveness within various compute paradigms. The research in this space has spanned from evaluating Docker as an edge compute platform where researchers propose that the containerization properties are sufficient for reducing latency and enhancing user experience [7], to using Docker in high performance compute applications where researchers compare Docker to other VMs [8]. Researchers have also studied Docker’s integration into cloud computing [9] and the economics of cloud computing using Docker [10]. While Docker generally provides deployment efficiency, there are certain drawbacks to using Docker.

More specifically, in the security space there has been a lot of research into the vulnerabilities Docker faces. Researchers have tested the general security of Docker, finding that Docker’s ecosystem needs more enhanced security protocols and practices to guard against threats [11]. There has also been work to define the different threat models that Docker faces [12] from DoS attacks [12] and threat detection frameworks that Docker can use [13].

In the image security space, researchers have looked into DockerHub images hosted on DockerHub and found that official and community images contain vulnerabilities or haven’t been updated [14]. They have also looked into overall security of DockerHub finding run commands that leak host files, malicious images, and a general lack of software patching [15].

The above is all important work that aims for more secure use of Docker. However, we maintain that we are the first to explore what vulnerable configurations look like in real life and propose an easy to use tool that allows individuals to check that their Docker configuration file for secureness before running.

The remainder of the paper is organized as follows: In Section 4 we describe our methodology for analyzing Docker configuration files. Section 5 describes our results from the analysis. Finally in Section 6 we discuss further work that can be done.

4. Methodology

In order to test our hypothesis, we broke the problem down into three parts: creating a tool to measure the prevalence of potentially vulnerable Docker Compose files, using that tool to measure configuration files given by services as example configurations, and using that tool to measure how often Docker Compose files are misconfigured via files from GitHub.

4.1. Measurement Tool Metrics

The measurement tools aims to analyze configuration files and indicate whether or not a given file contains what we consider a potential vulnerability. Since configuration

files are written in YAML, they are easy to parse programmatically.

More concretely, the tool looks for the following:

Implicit mapping of IP address 0.0.0.0. By default, whenever a user maps a port in a configuration file (or via the command line), if no explicit IP address is also provided then Docker will implicitly map 0.0.0.0 in the NAT rules that is generates. This clearly allows for any external actor to create a connection to the container on any of the ports it has been mapped. We chose to measure this because Docker bypasses a user's provided IP-tables IP filter rules since the NAT rule it generates and adds to the systems NAT table will trigger prior to any IP filter rules in the filter table [3].

We consider this an issue because while this is documented, this is never actually mentioned in any of the getting-started materials from Docker, or from image maintainers for popular self-hosted software. Therefore, it is feasible for a user to use this configuration option while believing their firewall configuration is protecting their host machine and their container while in reality their containers are completely visible to outside observers over the network.

In order to check for this in a configuration file, we simply enumerate each service within the configuration file, and check if it has a 'ports' key within it. If so, we then check to see if an explicit IP was given with the port mapping. If not, then we know that 0.0.0.0 is implicitly mapped, as so we flag that service (which is itself just a container with a name!).

We also offer a recommendation to use a mesh VPN, such as Tailscale, in order to acquire a secure static IP address to use for explicit mapping. This has the benefit of being very simple compared to something like a reverse proxy, and prevents users from needing to open their containers up to the public internet. While this solution will not work in all cases, we believe it will work in a majority of cases for self-hosted software.

Whether any networks are created or specified for containers. We chose to measure this since Docker Compose defaults to a general 'default' network that all other containers without specified networks will also share. This allows for containers to see and communicate with one another without having to explicitly link them together. We consider this inter-container communication an issue because it violates the principle of least by allowing containers to view the existence of, as well as interface with, other containers on the default network. This could be exploiting for lateral movement within a compromised environment. Given the nature of self-hosting often dealing with private personal data, this could lead to an attacker gaining access to sensitive information that other containers on the default network handle.

There are cases where multiple services need to share the same network, for example if a user wanted to route a container's traffic through a designated VPN container. However, we assert that the shared network should be separate from the default network and well defined within the configuration file to prevent communication with unwanted that are configured within the same Docker Compose file.

We check this in a similar way to the previous issue, except this requires two checks instead of one. In order to create a network in the configuration file, a user must declare a 'networks' key outside of the 'services' list. This is where the definition and configuration of the network happens. Each network defined here will have a name, which will be referenced within the configuration of any container within the 'services' list that will be on that network. Therefore, if our tool does not detect any network defined, or if no service (container) references a defined network, we will mark this as a found issue and warn the user. We also suggest to the user to consider creating more isolated networks to prevent containers from communicating unless they explicitly need to.

Whether a service defines a user and group for its container to run as. This is the user and group ID that is running processes within the container. We chose to measure this since Docker will default to running containers as a root user if one is not provided in the container configuration. We consider this an issue since any capabilities, bind mounts, or volumes given to a container will effectively lose all access control since the root user may do whatever they like. This is in contrast to when a user and group is specified, which can allow the administrator to restrict file-system, resources, etc via normal user access controls in Linux.

We check this in a similar way as the previous two. We enumerate each service and check for either a 'uid:gid' key within a service, or for the PUID and PGID environment variables being defined within the 'environment' section of the service. If neither the 'uid:gid' nor the 'environment' key exists for a service, we know that no user or group was mapped to the container, and thus the default docker user is used.

Whether overlapping host directories are mapped to multiple services with default read/write permissions. We chose to include this because some services within the self-hosted community work in tandem, and require shared access to host directories. A good example would be a multimedia manager service alongside a multimedia streaming service. One manages the files, while the other needs to be able to read and stream them to clients. A notable property of this example is that only one of the two services needs write access, while the other only needs read access.

We consider this an issue because it could lead to data corruption if two containers make conflicting changes to the same files mapped from the host, as well as the ability for an attacker to indirectly affect another container which shares a host directory with the already-pwned container. An example of this could be a Denial-of-Service by simply deleting important files in the shared directory.

We check this by enumerating each service and checking for a 'volumes' key. If none are present, we know that there cannot be any overlapping volumes with other services in the Docker Compose file. However, if the key is present, we then enumerate all the volumes for that service, and add it to

a set of seen host directories. This allows us to check future volumes on other services for overlap by simply checking if we have already seen a given host directory. If we have, then we flag these two services, and warn the user about which volumes overlap with read/write permissions. If two services have overlapping volumes, but only one of the services has write access and the rest have read only access, we do not flag it. This is essentially a way to prevent a multi-writer scenario by trying to keep it to a single writer, multiple reader scheme.

4.2. Measuring Prevalence of Image Maintainers Providing Vulnerable Example Configuration Files

We wanted to understand how image maintainers communicate to users how to use their Docker images. To do this, we first identified 100 popular images for self-hosting by using the 'awesome self-hosted' GitHub repository, which lists popular self-hosted software listed by category, alongside DockerHub and GitHub's own image repository to see how many total and weekly downloads the image had. A majority of our chosen images had more than 10,000 weekly downloads and some had closer to 1 million weekly downloads. The few images that had less than this were often niche alternatives to overwhelmingly popular software within the same category, but were still downloaded often enough that we felt they were worth including. None of our chosen images had less than 1000 weekly downloads according to DockerHub or GitHubs image repository. A notable limitation was that some of our chosen software did not have an actively maintained docker image, so we were unable to evaluate their example configurations, or real uses of the image in the next step. This effectively brought down our pool of images from 100 to about 97. Once we had identified all images we wished to investigate, we ran our tool against their example configurations, and checked the results against our own manual evaluation of the example configurations. We saw that our manual evaluations lined up with our tools results exactly, which was promising for the next phase.

4.3. Measuring Prevalence of Vulnerable Configuration Files on GitHub

For this we set a target of downloading 100 Docker Compose files for each image we wished to investigate. This would have given us a pool of 10,000 configuration files to scan. In order to obtain the Docker Compose files, we searched on GitHub for any 'docker-compose.yaml' (or '.yml') file which contained the name of the image we were investigating. For example, if we are trying to collect 100 configuration files which have the 'Jellyfin' Multimedia software in them, we queried GitHub for all Docker Compose files containing the string 'jellyfin'.

However, when we actually began collecting the configuration files from GitHub, it became clear that some of the images we had chosen did not have any results for our queries within public repositories on GitHub. Exactly 21 of our chosen images returned no configuration files on GitHub. To combat this, we collected more than 100 configuration files from really popular software images.

Additionally, 3 of our chosen images had names that were too generic for our queries to be meaningful (one example was a software called 'Send' by Mozilla). This meant that in total, 24 of our chosen images could not be evaluated this way. Combined with the 3 images from the previous stage which had no maintained Docker image, we now only had 73 images we were able to collect configuration files on from GitHub. In total, we were able to collect 3076 Docker Compose files relating to our chosen images. While this is only 30% of our initial goal, we still feel that it is large enough to suggest trends in how users are configuring their Docker containers for self-hosted software.

Of the 3076 collected configuration files, our tool detected that 3038 of them were 'valid', meaning they were in the standard Docker Compose form. The ones which were marked invalid either had large syntactical errors, or were simply including other Docker Compose files for each service.

Once we had all the configuration files downloaded and sorted by which image they were queried for, we began running our tool against them which analyzed them and kept track of which files had which issues flagged. This allowed us to keep track of the percentage of files which had a given issue for each service, as well as overall.

We decided to use these public configuration files found on GitHub because we believe they give a good representation of how people are using Docker in the real world to deploy these self-hosted services. In order to measure the gap between the acknowledged risks Docker has, and the actual risk of users' deployments, we needed real users' configuration files. While an alternative approach could have been to create a survey asking for users' to submit their configuration files to our data set, we felt that would be too self-selective, and that we could get a broader range of data by querying public GitHub repositories.

Additionally we can't proclaim to know what ever user's configuration file looks like. However, we do believe that the configuration files people posted on GitHub can provide a representation of what is happening in the real world.

5. Results

5.1. Top 100 Self-Hosting DockerHub Configurations

After evaluating the example configurations for each image provided by the image maintainer we saw that no image had an example configuration that passed all 4 of our checks. The most that any image passed was 3 out of 4 checks. That image was the Gitea image. Gitea is a self-hosted version control software, so the fact that it's example configuration passed the most checks makes intuitive sense given that version control systems are often used by developers who

may have more experience using tools like Docker than the average user deploying self-hosted services. The majority of images passed 2 out of 4 checks. We also noted whenever a sample configuration noted that a user should change certain settings or look into the documentation for certain settings. Even with this, no image had higher than a 3/4 success rate.

One thing we did notice was that images which were maintained by LinuxServer.io had better example configurations, as well as warnings and documentation for users on how to actually change the example configuration to be suited for real world use. Despite this, even these images failed to pass all of our checks, and also failed to mention the potential risks associated with all the configuration settings we check. They may have mentioned two or three of them, but never all four. The one that was never mentioned across any example configurations and their documentation was the implicit IP mapping done when mapping host ports to containers. We found this disappointing since this could be the biggest vulnerability for someone deploying these services on cloud infrastructure like Linode.

Overall, the example configurations failed to meet our bar for all 4 issues we were looking at. Some of the images' documentation even incorrectly explained some configuration settings like the host to container port mapping, which could confuse users who do not have experience with Docker or deploying software services. The best example configurations and corresponding documentation consistently came from LinuxServer.io maintained images. Although these did not adequately explain the risks involved with all of the settings we looked at, the documentation was nicely standardized across all the different images that they maintain.

5.2. Github

After running our tool across all 3000 configuration files we obtained from GitHub, we saw clear trends in how people were writing their configurations. Figure 1 shows the percentage of tested files which had a given issue as well as the percentage of tested files which did not have the issue present. We can clearly see that around $\frac{3}{4}$ of tested Docker Compose files had the implicit IP mapping issue, while $\frac{2}{3}$ possessed the default network issue. Furthermore, only $\frac{1}{4}$ of the tested files contained the volume issue, and 97% of tested files failed to map a user/group ID to their containers.

Although we cannot be certain that all of these configuration files are used in a default docker environment in which the default user within containers will be root, we assert that only a minority of the tested files, if any, would likely use a root-less installation of docker given its lack of support, and more involved setup process.

It is interesting that this issue was overwhelmingly prevalent in the configuration files since it is definitely the easiest setting to configure. Any non-root (or password protected sudo account) would provide better security than the default, and would obey the principle of least privilege more. If a container with a properly mapped user and group

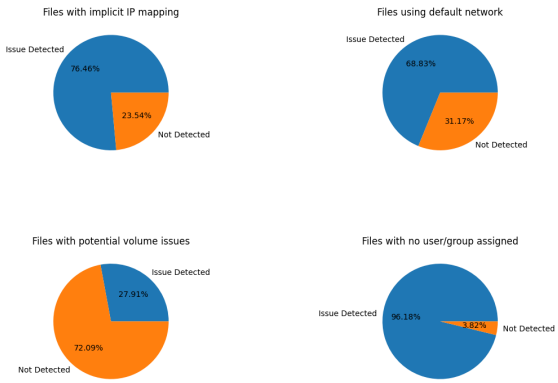

Figure 1. Percentage of files flagged vs not separated by issue type

were compromised, this would help limit the amount of damage that could be done by an attacker. Specifically, the attacker would only be able to do whatever the mapped user/group ID could do within the container or on the host system (if any host capabilities were given to the container).

The next two most prevalent issues were the implicit IP mapping, and default network use. These are probably the two most complex settings of the four we investigated, since a 'proper' secure configuration will vary not only by each service, but also by how the user intends to use the service. For example, if these were deployed on a home network with no port forwarding enabled, then the implicit IP mapping would pose little threat unless an attacker had access to the user's local network. However, if the user deployed these configurations within a cloud environment like on a Virtual Private Server instance which has an assigned public IP address, then it could be much more worrisome and potentially lead to a compromise of the users system and personal data. The same applies for the explicit network definitions and configuration, although we believe that even in a home network setup, it is still worthwhile to separate services onto different Docker networks to prevent inter-container communications. After all, defense should be in depth using multiple layers of security mechanisms.

The most surprising result we found was that the volume configurations seemed to be mostly okay, with only about 28% of tested files being flagged. After thinking about it more, this makes intuitive sense because most services do not need to share any data with other services. The notable exceptions within our data set were Sonarr, Radarr, and Transmission. Sonarr and Radarr are services which will automatically manage shows (for Sonarr) and movies (for Radarr) by automatically downloading media via p2p protocols like bit-torrent, or usenet, as well as managing a media library more generally, like renaming media files and folders to match a unified naming scheme, or fetching subtitles for existing media on the system. These services often share

host directories with a download client, like Transmission which is a bit-torrent client. We see this reflected in our data, since most of the flagged files for the volume issue come from Sonarr, Radarr, and Transmission. This suggests users are in fact allowing these services to share access to the same host directory, which is the intended behavior of the services.

Overall, we see that the implicit IP mapping, default Docker network, and user/group ID mapping are the most prevalent issues of the 4 that our tool currently checks for. Not only that, but they are all three **extremely** prevalent in the wild, with well over $\frac{2}{3}$ of configuration files being flagged for each issue, and in the case of the user/group IDs, nearly 100% of configurations were flagged.

5.3. Case Study: Mock Servers

After collecting our results on the large set of configuration files from GitHub, we decided to setup a test environment with some of our selected images. We wanted to deploy the services on cloud infrastructure, and chose Linode as our provider given their popularity within the self-hosted Reddit community. Our VPS (Virtual Private Server) was assigned a static public IP address, and was running Ubuntu server 23.04. We configured our firewall rules to deny all incoming connections via UFW, as well as disabled root SSH login, and password-based SSH login. All of these are considered basic security practices anyone should follow when setting up a new server.

After setting up the host environment and downloading Docker, we began writing our own configuration files. We chose to have two: one which followed the example configurations from the images we chose to include, with minor changes, as well as a second one with a more thought out configuration taking into account the issues we have identified earlier. The goal of this exercise was to try and identify simple and easy solutions to each issue that we could add to our tool so that users more easily fix their configurations after using our tool to evaluate them.

For the user/group ID problem, the solution was trivial. We added some messages to our tool telling the user why they should assigned a user/group to their containers, as well as giving an example of how to do it. We also added the option for our tool to automatically add these changes to the users configuration file if they wanted. By default, our tool assigned PUID and PGID 1000 to the container, since that is the first non-root user/group ID assigned on a Linux system, and thus will give the containers the same privilege level as the normal user. However, if the user wishes, our tool can also assigned any other PUID/PGID for them, but we warn them that they must create the user and verify their IDs prior to deploying the services.

The default network issue was also mostly trivial. Since we have no hope of providing a single-click solution for everyone's network needs, we settled for verbose messaging. Our tool is now able to not only warn a user of which services are on the default network, but also provides brief explanations on why it is important to separate their services onto different Docker networks. After these explanations, we offer a few examples on how to set them up. We provide an example for having only a single service on its own network, as well as how to setup a network designed for multiple services to share in the event that some services need to communicate with one another.

Finally, we focused on the implicit IP mapping. Our less-secure configuration file was allowing us to see which ports were open for containers via basic port scans as we explained earlier. However, actually coming up with viable solutions for users was harder. Oftentimes, users want to be able to access their services from outside their own network, and given that our environment is on cloud infrastructure, we had to have that access for the services to be usable at all. One popular option for this is to setup a reverse proxy container and have it handle the public internet side of things. However, reverse proxies become complex very quickly if you have any more than a few services deployed since you not only need to keep up with the general configuration, but also handle issuing and renewing SSL certificates in order to have HTTPS enabled. Couple this with needing a domain to be able to actually make this useful and things started to feel a bit too complex for us to suggest this method.

Instead, we began to test mesh VPN's like the one Tailscale provides. These essentially just create wireguard or openVPN nodes, and manage the public keys between the nodes for the user. In practice, this create a flat network between devices that have the mesh VPN running, and only users which are authenticated with the VPN can even see that the service is listening on a port. We tested this by creating a mesh VPN network with Tailscale for our server. Tailscale assigns each machine on a user's tailnet (what they call the mesh network it creates) an unchanging static IP. We then plugged that IP address into the port mappings in our Docker Compose file, so that the containers were only listening on the VPN IP address and port.

We verified that this prevents outside actors from seeing active ports mapped to containers by running a port scan on the server again. This time, we saw no open ports! This meant that our firewall was now working as intended for all other incoming connections, and that Docker was only routing connections to the Tailscale IP address to our containers.

Given how easy this method was to setup, and the high level of security it provides, we decided this was going to be what we suggested to users who use our tool and detect the implicit IP issue. This method has the benefit of being super easy to setup compared to a reverse proxy, as well as being easy to automate. A user can provide our tool with the Tailscale IP address of their server and our tool will automatically fix all the port mappings to explicitly assign this IP. Of course this may not work for every single use case. One example would be a DNS server, which will need to listen on more than just the Tailscale IP if it is meant to be used by many people. To mitigate this issue, we allow the users to confirm each change we make to the port assignments so that they may prevent the tool from assigning the Tailscale IP to services that actually do need

to map to 0.0.0.0.

It is important to note that while we used Tailscale here, this method would work with any mesh VPN network. Tailscale even has an open source implementation of their control server called Headscale which can be run in Docker. All of this together made us believe this was the best suggestion we could give to users of our tool.

For volume management, we simply emit warnings of overlapping volumes, and suggest to the user to either separate the host directories if possible, or try to make some of them read only to enforce the single-writer/multi-reader paradigm. Since our results indicated this was much less prevalent of an issue than the other three issues, we spent less time on it here.

Overall, our tool is now able to not only emit warnings when it finds potentially vulnerable configurations, but can also suggest fixes, provide examples of those fixes, and in some instances, automatically fix the issue for the user. We believe that with this tool, users can quickly move toward a more secure Docker environment.

6. Discussion

During our investigations we came across some anecdotal evidence that these issues have already had real world consequences for user, especially the implicit IP mapping done by docker. Specifically, we found a GitHub issue thread related to this exact topic. The thread had been open since 2019, and multiple users claimed that they have had their databases stolen and environments attacked due to these ports being observable by outside actors. It was made clear that Docker had no plans to change the behavior of the Docker daemon and how it creates NAT table rules, which makes sense given the variety of use cases Docker needs to support. This lead us to wonder how many other users may have this specific configuration issue.

Another thing worth mentioning is that all of our data and investigations relies on the assumption that Docker is the containerization platform being used. Some of these issues, like the user/group ID mapping, are not an issue with other container platforms such as Podman since they do not run as root by default. It is unclear whether any of the other issues persist with drop-in docker replacements. By restricting our investigation to Docker Compose files, we hoped to ensure that the configurations we tested were actually used with the Docker platform and not a drop-in subsistute.

Overall, we found that 3 of the 4 issues we identified are widespread among users configuration files. We also found ways to enhance our static analysis tool to be able to provide helpful suggestions and automated fixes when possible. Our hope is that people interested in self-hosting will be able to use our tool to bring their Docker configurations to a more secure state.

For further work, it would be interesting to evaluate these issues in a broader environment than just self-hosted software, such as production software deployments, or software development pipelines which make heavy use of Docker. Another avenue for further research would be to collect more configuration files from GitHub across more self-hosted software. This could provide a better indication of use patterns, as well as let us identify patterns across categories of software rather than just across all the tested images like we have done here. It could lead to better suggestions and automated solutions based on the type of software being deployed.

Looking forward for our static analysis, a valuable extension would be to check for mishandling of secrets, such as API keys and passwords, which presently we do not check at all. All the security in the world can become moot if you give away your secrets by accidentally pushing them to a public repository like GitHub or GitLab.

7. Conclusion

Docker is a tool that is widely used for self-hosting. While there exists documentation on how Docker containers can be vulnerable, we wanted to discern whether or not individuals in the self hosting community were deploying secure Docker containers. After analyzing over three thousand Docker configuration files (with our tool) we conclude that based on our empirical analysis people are frequently deploying vulnerable Docker containers. This is of special concern given that we chose to focus our research on the self hosting community - where self hosted data is usually sensitive. Our analysis reveals that there is indeed a wide gap between acknowledged risks and actual deployment practices.

In efforts to begin closing that gap we extended our measurement tool to offer reasonable solutions and automated fixes, in order for people to deploy more secure containers. We acknowledge that our analysis has limitations, however, we think this is a solid step forward in fostering more secure deployment landscapes.

References

[1] "Who Does That Server Really Serve? - GNU Project - Free Software Foundation," Gnu.org, 2010. https://www.gnu.org/philosophy/who-does-that-server-really-serve.html

[2] "Use volumes," Docker Documentation, Jan. 04, 2019. https://docs.docker.com/storage/volumes/

[3] "Networking overview," Docker Documentation, Apr. 23, 2021. https://docs.docker.com/network/

[4] S. Sultan, I. Ahmad and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," in IEEE Access, vol. 7, pp. 52976-52996, 2019, doi: 10.1109/ACCESS.2019.2911732.

[5] Souppaya, M., Morello, J. and Scarfone, K. (2017) Application Container Security Guide [Preprint]. doi:10.6028/nist.sp.800-190.

[6] Patra, M.K. et al. (2022b) 'Docker Security: Threat model and best practices to secure a Docker container', 2022 IEEE 2nd International Symposium on Sustainable Energy, Signal Processing and Cyber Security (iSSSC) [Preprint]. doi:10.1109/isssc56467.2022.10051481.

[7] B. I. Ismail et al., "Evaluation of Docker as Edge computing platform," 2015 IEEE Conference on Open Systems (ICOS), Melaka, Malaysia, 2015, pp. 130-135, doi: 10.1109/ICOS.2015.7377291.

[8] M. T. Chung, N. Quang-Hung, M. -T. Nguyen and N. Thoai, "Using Docker in high performance computing applications," 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), Ha-Long, Vietnam, 2016, pp. 52-57, doi: 10.1109/CCE.2016.7562612.

[9] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," in IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, Sept. 2014, doi: 10.1109/MCC.2014.51.

[10] K. Kumar and M. Kurhekar, "Economically Efficient Virtualization over Cloud Using Docker Containers," 2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), Bangalore, India, 2016, pp. 95-100, doi: 10.1109/CCEM.2016.025.

[11] T. Combe, A. Martin and R. Di Pietro, "To Docker or Not to Docker: A Security Perspective," in IEEE Cloud Computing, vol. 3, no. 5, pp. 54-62, Sept.-Oct. 2016, doi: 10.1109/MCC.2016.100.

[12] A. Tomar, D. Jeena, P. Mishra and R. Bisht, "Docker Security: A Threat Model, Attack Taxonomy and Real-Time Attack Scenario of DoS," 2020 10th International Conference on Cloud Computing, Data Science and Engineering (Confluence), Noida, India, 2020, pp. 150-155, doi: 10.1109/Confluence47617.2020.9058115.

[13] D. Huang, H. Cui, S. Wen and C. Huang, "Security Analysis and Threats Detection Techniques on Docker Container," 2019 IEEE 5th International Conference on Computer and Communications (ICCC), Chengdu, China, 2019, pp. 1214-1220, doi: 10.1109/ICCC47050.2019.9064441.

[14] Rui Shu, Xiaohui Gu, and William Enck. 2017. A Study of Security Vulnerabilities on Docker Hub. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY '17). Association for Computing Machinery, New York, NY, USA, 269–280. https://doi.org/10.1145/3029806.3029832

[15] Liu, P. et al. (2020). Understanding the Security Risks of Docker Hub. In: Chen, L., Li, N., Liang, K., Schneider, S. (eds) Computer Security – ESORICS 2020. ESORICS 2020. Lecture Notes in Computer Science(), vol 12308. Springer, Cham. https://doi.org/10.1007/978-3-030-58951-6-13