ASHER SECURITY

`

Securing Cloud Applications Guide

Docker, Kubernetes, and Google Cloud

Part 1 – Developer Foundations

Are you responsible for the security of application containers running in the cloud?

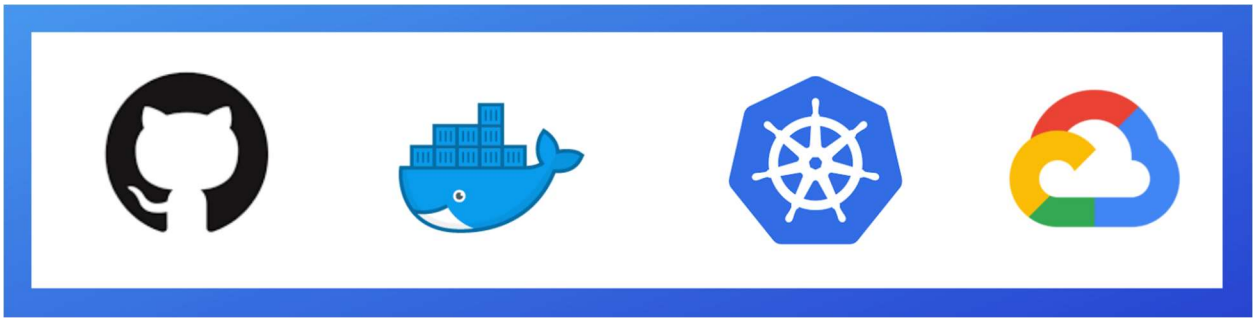
If so, you've found the right source to help you secure it. In this series, we'll focus on how to secure developed applications hosted in the cloud. We'll walk through where the code starts and follow it along the journey from development environments, to code repositories, to containers, to orchestration, all the way to the cloud platform. We'll review and recommend security controls every step of the way.

Your specific implementation may vary a little, but there's sure to be many applicable controls you can leverage to reduce the security risks of containerization and orchestration in the cloud.

The scope for this multi-step undertaking is:

- Secure the application against threats to integrity, availability, and confidentiality.
- Review the security of the programming languages and libraries.
- Secure the application being hosted in Docker containers.
- Secure the orchestration of the application performed by Kubernetes.
- Secure the application in the cloud hosted on Google Cloud Platform.

Follow along as we attempt to successfully secure this GitHub, Docker, Kubernetes, and Google Cloud Platform application implementation.



Foundation

"A journey of a thousand miles starts with a single step." – Chinese proverb.

When running container applications hosting sensitive or confidential information the journey should begin where the data is born. In this situation, we're trying to secure data within a Kubernetes environment. So we need to start with where the application is being built.

We will fast forward through the first few steps and assume this is being done without a detailed explanation:

1. Create a company culture where security is important.
2. Create solid and straightforward information security policies.
3. Create supporting data handling guides, and resources.
4. Host user security awareness training for all employees.
5. Create specific user security awareness training for special groups that have high risk. (Application developers in this case).
6. Hire talent based on qualifications and a standard questionnaire process.

At this point, you have a company that supports information security, they believe it's important. They've all been trained. You've hired some really good developers that know their stuff and understand the basics of security and can help protect against the most prevalent exploits like SQL injection. Have they heard of OWASP? Good, then let's get started.

Developer Security Group

Create a group of employees that are considered '*developers*'. Depending on the size and scope of your company you might have to break this down further into subgroups like; *application, database, SaaS, CRM, ERP, Web* and so forth.

The goal here is to manage your developer ID's in such a way that a single change will produce all the security results and impacts needed. You want the ability to remove an ID from one single security group and have all the access removed, instead of individually managing a single user ID's within development systems and applications.

Developer Environment

Work with technology leadership early and often to learn what development initiatives are being planned for the next year, and ideally as far out as you can. Your goal is to learn specifically what application are on the dock for development, where they will be hosted (internally or externally), what data the applications will host and deliver, and what programming languages will be used to achieve this.

Using this information you can agree on which of those applications will be considered a '*HIGH*' risk. You can agree on what hosting platforms will be used. You can also review and agreed what the standard environment for development looks like. For some, this is a virtual box running on their workstation. For others, this is a full-blown set of servers and databases for the specific scope of development. Learn what employees will be developing with what development tool-kits and environments, weather it is Android Developer Kit (SDK), or .NET

Studio, or Java (JDK), so you can know where to expect these development kits installed and running.

The further ahead of time you know about these initiatives the more time it will allow you to perform security reviews, build best practices guidelines, and plan for mitigation of security weaknesses.

Secondary Accounts

Next, create secondary '*development*' ID's for all the developers. Include them in the correct security groups, and provision those security groups to the appropriate resources (virtual workstations, or development servers).

I recommend creating a naming standard for development ID's, such as adding a *'D-'* at the beginning of the name. It makes it much easier to search and review for developer accounts, review access, and create whitelists if you're leveraging a SIEM with a security correlation engine. It also makes it easier on the eyes to identify non-developer accounts with developer access and privileges. Which brings me to my next point…

Remove the user's primary accounts from the development resources. This prevents an attacker pivoting from a phishing exploit to quickly owning all the application code. Keep the two roles separate. This will pay security dividends.

Remove Local Admin

Remove, or never allow, local administrative rights of the developers to their workstations. Only provision administrative access to the development resources. I know, it's hard. It's a culture change. But our culture has adapted to many changes, and this is an important one. The only administrative access the developers should have is to the development environment. This does not include the production environment.

Service Accounts

Build a policy for '*service accounts*'. Be clear stating that in production they cannot be logged in interactively and are only reserved for application services. Service accounts cannot be used by employees for the management of resources. If service accounts need to be logged into for troubleshooting and support, a ticket needs to created (or whatever your support process is). The credentials will need to be updated when the support is over, and the interactive login ability removed via policy.

`

Entitlement Review

Create a process for user and asset entitlement reviews. I personally like to see a regularly scheduled process complimented by a random sampling process. The goal is to have someone review a list of assets and ensure only users that should have rights to those systems are currently provisioned. The process should also look for backdoor accounts, locally created accounts, and accounts added to the local admin group that should not be there. The process should be rotated and shifted to a sampling of user's accounts, instead of assets. Walkthrough the group membership of the user account and ensure the appropriate provisioning is setup.

Concerns

A common argument is that developers need administrative access to production boxes to support them. If the application is tested through a proper development life-cycle, there should be a low risk of production issues.

Create a break-the-glass security process for when there is a production issue. This process allows developers access to the production systems in the case of a production outage. Add this credential to your SIEM engine for alerts when it's used. Ensure that when the account activity is alerted on there is a production incident in progress. Test this process beforehand to ensure it works.

The same can be said for service account access. You can store the service account credentials in a break-the-glass procedure if you feel it necessary for production support.

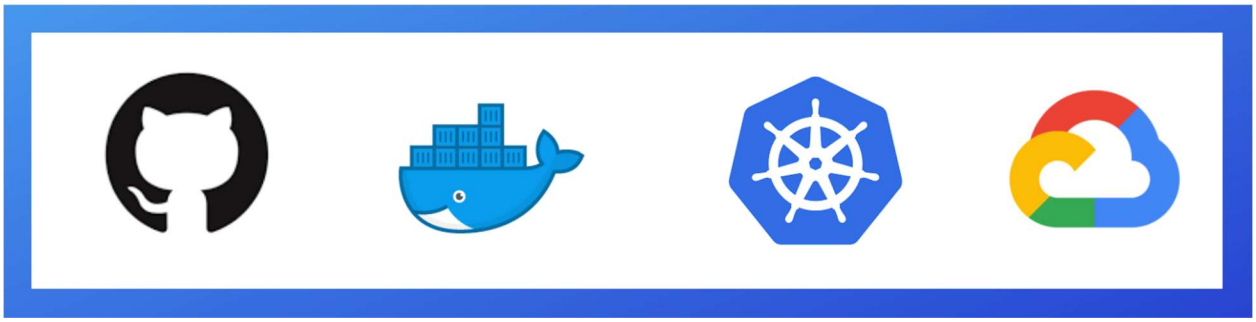
Conclusion

That covers the fundamentals of starting off with a secure development life-cycle. With these core security tenants in place, we can move forward in our security journey to protect our application in GitHub before it moves to Docker, and Kubernetes in the cloud.

In the next part of this series, we'll focus on development languages and libraries. How can you review and ensure the security of what the developers want to use to build what they need to deliver?

Part 2: Development Languages & Libraries

In the previous section, we covered the foundation for security. The business supporting security, and everyone getting trained. Developers created unique accounts and the development environment is in place and ready to go.



In this section, we focus on the programming languages and associated libraries.

What programming languages will they develop with?

Relationship

The most important thing to have in place is a relationship between security and developers. This can be a formally scheduled meeting every quarter. Maybe this is an informal coffee date. Maybe this is an information security manager sitting right next to the manager of the development team. (*I like that idea*).

A relationship needs to be in place and support continual conversation and agreement. Developers have a hard job. They need to create something out of nothing. They need to take the vision of the company and make it appear. They need to lower company costs and improve client experiences. They need to do it all while worrying about bandwidth, workloads, bottlenecks, read and write times, availability and reports.

Security needs to come alongside them, listen to what they are trying to do and start laying the paver stones so that they have a clear and visibility runway.

Security needs to know what the developer's objectives and goals are. What their priorities and initiatives are so that we can understand what tools, languages, and resources they need to deliver that solution in a secure manner. We need to get ahead of it and provide the resources, instead of waiting until their done and then telling them they did it wrong and it doesn't meet our security requirements.

By learning ahead of time what they're going to develop, we can start the process of providing a secure roadmap. Only when developers do not communicate what they're working on, and security learns after the fact, do we have some level of right to point out how it was done wrong.

Languages & Libraries

Now that you know what will be developed and what they want to use to create it, you need to decide what your security approach will be approving languages and libraries.

Libraries

For those that don't know what a code library is I want to explain at a very basic level before moving forward. With development languages like Java, Python, and .NET (among others) you'll have the core of the development language that comes wrapped in a binary distribution. You install this on your development system or endpoint. This is where the basic coding or programming is done and often referred to as the 'environment'.

For Java it's the Java Development Kit (JDK), for Python it's a combination of a text editor and a runtime environment. And with .NET it is .NET Studio.

Libraries are extensions of the core package and binary that can be imported to increase the capabilities, functionalities, and extend the language. Libraries may include instructions, templates, and pre-written code.

Notice that last part? 'Pre-written code'. That's where you'll need to focus. as the old saying goes.

"A good developer writes code, a great developer steals it."

So, if you allow the use of libraries or pre-written code, you'll need to create a review and approval process.

I have my personal preference, but I also understand the same process doesn't work for everyone. Each business is unique.

The two methods are 'pre-approving' and 'post-approving'. Let's take a look at the advantages and disadvantages of each.

1. Pre-Approving

This approach takes the intake from developers on which languages they want to use and requires security go research that development language and the associated libraries and produce an '*approved*' list. This is essentially a formal list of what can be used.

Sometimes this can be very easy. For example, you learn the team wants to develop with Java. You can do some basic research and make a statement like Java 6 is not allowed, but Java 7 is approved. You also have the option of approaching like a policy statement such as,

"Current development code needs to be used. All development must be using current releases, and nothing older than twelve months. If a serious vulnerability or exploit is reported, it must be patched and remediated within thirty days. "

It becomes more difficult when it's a language like Python, and how and when existing code can be used from their favorite Internet sites.

2. Post-Approving

In this security philosophy, you are giving the developers permission to use whatever they want.

This privilege comes with a price.

A famous quote applicable here, yet used out of context,

"The cost of freedom is always high."

– JFK

That price is, the developers are ultimately responsible for the security of their own code and applications.

Security will act as the auditor and inspector, but if they are ultimately responsible for the security of it.

The best way to implement this post-approval process is by having a strong relationship with developers, providing them training and resources on security exploits, having strong upper management support and security culture, and lastly having a security 'ninja' on the development team.

A security ninja is a security champion. They are a coder with security training. They do the hard application library security lifting, so you don't have to. They take the security boundaries and authority given to them and they do the best they can with it.

Your role in this philosophy is two-fold:

1. **Train** – Provide them with all the training on exploits, security, and best coding practices that you can find.

2. **Verify** – Create a system process and toolset to verify what they are creating is secure. This starts with the highest risk applications and works its way down. For larger organizations, this might be a dynamic code analysis in line during the development process. For smaller companies, this might be a third-party code testing tool like Veracode (or similar).

Summary

The criticality of having a strong relationship with the development team cannot be underestimated. Have a process of learning what they are going to be working on, and what tools they will be using.

Develop a security philosophy that will pre-approve language and libraries, or that will support trust and verify methodologies.

Build a relationship with the development team that reduces friction, and can enable them to do deliver great products securely.

If you've found a system that works for you, I invite you to share it in the comments section.

Next

In the next section, we will look at application repositories and the importance of storing code in a secure location. For the scope of our project we will examine moving our code into GitHub and what security best practices can be put in place to reduce the risk of secrets being exposed, the secure code being accessed, and the integrity of the application being impacted.

Part 3: Application Repository & Storage

In the last two parts, we've covered the basic foundation of a good information security program and how it supports a development team. We walked through setting up environments, accounts, and equipping the developers with training and resources.

From there we discussed the criticality of relationships and knowing what languages and libraries developers will be using. We then discussed the advantages and disadvantages of the different security philosophies on how to review the languages.

Tenants of Security: Code Responsibilities

1. Use company provisioned repositories (not individual)

2. Unique ID's that are directly tied to company ID.

3. Two-factor authentication enabled on external repositories such as GitHub

4. Logging and auditing enabled with central log collection

5. No secret storage in cloud repositories

Application Code Storage

Now that we finally have applications being developed, we need to turn our focus on how and where the application and code development is being stored.

What are the risks?

At this point in the application development lifecycle, the application is not hosting secure data. It hasn't been migrated to production yet. But there is still risk that exists early on in the development cycle.

1. **Accountability** - We need a system that tracks who worked on the code. Who updated it? What did they update?

2. **Version Tracking** - If an updated application causes a production failure and needs to go back a version, we need to know what versions we have, which one was last, and we need to know it's good.

3. **Integrity** - We need to know the code is what we think it is. If we have a good version, that has been security tested and approved and works in production, we need to know if it's been altered or tampered with.

The ultimate risk here is an unauthorized or malicious attacker editing production code so that they can access, or manipulate data in the application when it's in production. The other risk is

injecting production code with malicious code that could create an outage, or impact clients using the application. For example, injecting ransomware code in the company application that external clients use.

Take for example the historically popular way of developing applications. The developers would just copy and paste their code to a file share available on the network. The file share was typically open all and no logging was being done to track who accessed it when. The code was copied locally to endpoints sometimes and copied back when done. When two people worked on it at the same time, conflicts could exist.

How can we protect against this?

By leveraging a secure application repository.

What is an application repository?

It is a secure location to store application code that provides user access controls, version control, and code integrity through the use of hash values.

There two popular locations to store code;

- *Internally*
- *Cloud*

The most popular application code repository for internally is Jenkins.

The most popular code repository for the cloud is GitHub.

Scope for Security

Using our scope for this security project, we're trying to *secure an application that will be developed internally, stored in the cloud, and then hosted in the cloud on Google Cloud Platform, leveraging Docker containers on Kubernetes. This application will be transmitting, computing, and storing confidently and sensitive data.*

So we'll save the Jenkins conversation for a later time and focus on how to host application development code in GitHub securely.

Just for clarification, there is Git, and GitHub. Git is version control and can be run locally independent for version tracking. It is included in GitHub. GitHub is available on the web as GitHub.com and there is also GitHub Enterprise which is an on-prem solution. We're focusing on the cloud.

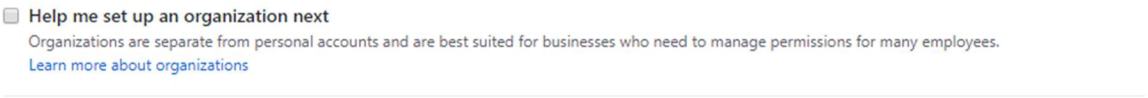
Secure Code in GitHub

- Company GitHub
- Business Accounts
- Two Factor Authentication
- Sessions
- Notifications
- SSH Keys
- Entitlement Reviews
- Stored Secretes

Security Controls

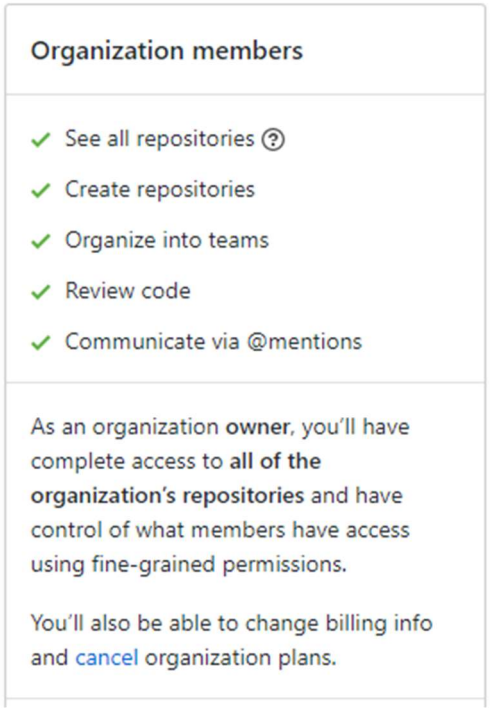
Company GitHub

First, set up a GitHub.com account for your organization. Start the sign-up process, and during the initial account creation process, you will be presented an option to create an 'organization'. Yes, do that. That is like setting up an application repository just for your business. You control who has access, what accounts will be invited, you control the security policy, and you control the code.



Business Accounts

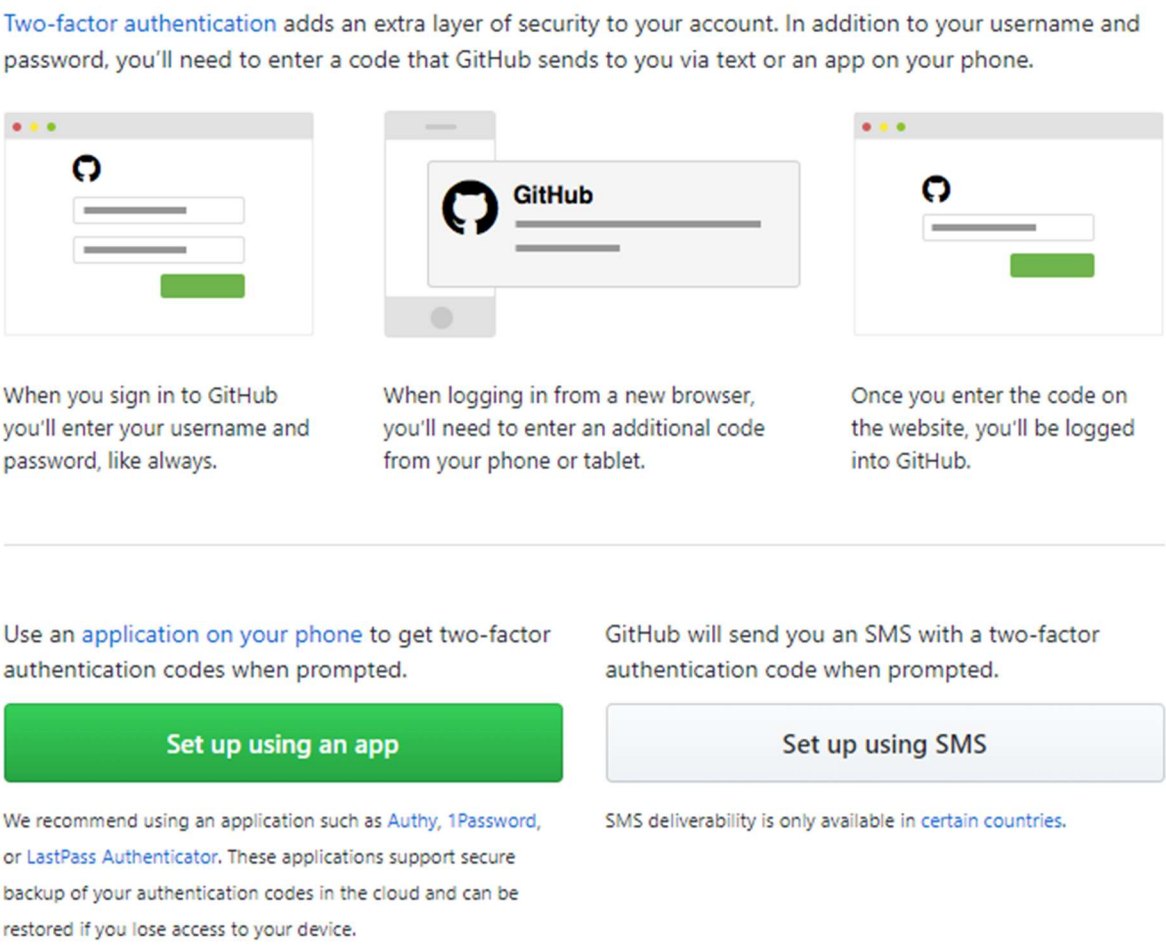
When you're done setting up your 'organization' in GitHub you can then 'invite' contributors. Only invite employees using the work email of your organization. So for us, we only invite 'johnny@ashersecurity.com' and not 'johnny@yahoo.com'. This further allows you to control who has access.



Two-Factor Authentication

Another benefit of setting up the organization is that you have control over the security policy. Use this privilege to enable two-factor authentication. A good security policy should include two-factor authentication for any cloud or external resource.

Two-factor auth is free and has several options on what you can use as your second factor including applications or SMS messaging. My favorite is Google Authenticator. It's free, easy to use, and easy to setup.



Sessions

By default, the '***Sessions'*** are turned on. This is a very valuable tool that can show you the source of the current login sessions. You can use the resource to help validate account connections are coming from valid user locations.

Notifications

There are some options for '***watching'***, which really means receiving notifications when people are added or join a team and such, but the thing I want to bring your attention to is a recently new feature being included in GitHub called '**Security Alerts**'

Here you can be made of aware of security vulnerabilities discovered by GitHub, in your dependencies. This is a very cool feature you'll want to enable. There are two options to receive

these notifications, one for every vulnerability, or a summary of vulnerabilities. I recommend the latter. If you choose this option, you can select how frequently it is emailed.

SSH Keys

If your developers are submitting code via the terminal, considering documenting an easy to follow process to enable SSH key creation. This will remove the requirement to provide authentication when a push or commit is applied. (Making it easier on the developers makes it easier on security).

Security Policy

The last two security recommendations cannot be configured with GitHub and instead need to policy and process within your business.

Entitlement Review

Setup a process, and schedule it regularly, when someone reviews all the GitHub active users and verifies, they should currently have the access and privileges that they have. The goal is to implement a complementary governance process that prevents terminated users, and external contributors, to keep their rights long after they should have been removed.

Secrets

Lastly, add to your security policy that secrets should not be stored in GitHub. It's easy to do and happens all the time. It's typical to have credentials stored within configuration files of code. These should be removed, or replaced with placeholder values before committing code.

In addition to the policy and user awareness, compliment your current security process of entitlement review with sampling and searching for credentials. This can be difficult, but if you have a common programming language that is used, reference the typical configuration files used for credential storage and search these locations.

Summary

I hope these security best practices are helpful for you securing your application and development code as it continues its journey out to the cloud platform into Docker containers.

In the next section, we will focus on how to secure code in Docker. The level of technical depth will increase to achieve the capabilities required for this process.

Part 4: Docker Containers

So far, our internally developed application has made its way securely to an application repository and is finally ready to be built into a Docker image, saved, and deployed as a production application running in a Docker container.

One mistake here can lead to a serious security impact.

Introduction

Docker is an amazing tool, but it comes with a lot of risks.

At the core of the risk, is not a Docker flaw, but more to do with what happens when we work on highly technical implementations with new technologies. It's a complex solution. It is hard to get an application running in the cloud in a Docker container successfully. It takes a lot of knowledge and depth of understanding just to get Docker to do what you want it to do.

Security often takes a back seat when product complexity is high. Security is placed on the back burner and the default values and configurations are accepted. Good security is not included in the production rollout. Just getting the Docker container running in production is defined as 'success'.

We can do better. As security professionals, we can help educate and equip. By learning about Docker security and becoming familiar with the risks, and defensive steps, we can raise this standard of what '*success*' looks like.

In this article, we focus on how to build and deploy Docker images and containers securely. We start by reviewing the risks, then we'll examine common vulnerabilities and attack methodologies and propose security standards, tools, and methods to protect your Docker implementations from exploits.

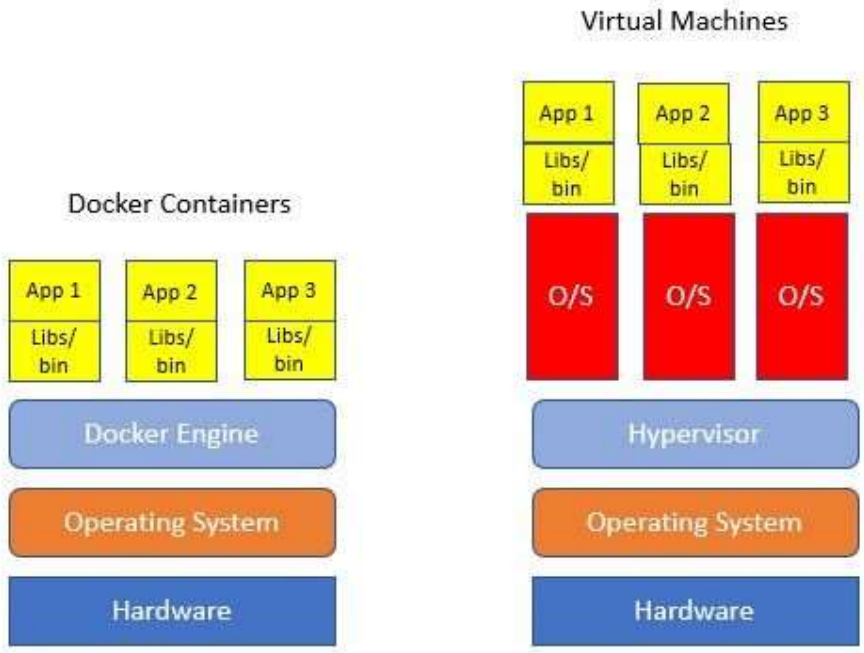
We're going deep and we're getting more technical as we go up the stack. I've tried to keep it as brief as possible to provide the understanding and provided links to additional articles for those that want more detail.

What is Docker?

Before we get too excited, let's first review what Docker is. Docker is a software platform that allows an application to be 'containerized'. Essentially this means you can run an application without having to support the full operating system. Within the container are the application, the libraries, and any other dependencies it needs. That's it. No operating system. The container is able to run without an operating system the system kernel for processing without a full operating system installation.

This means the application containers can be small, and lightweight. Because of their size, they can be easily movable and shareable. In addition, they take much fewer hardware resources. They can be self-contained with all their dependencies, which means they will run successfully across systems without all the proper dependencies and software versions being installed. And finally, they don't require all the additional software and applications to be installed and configure to get them to run.

The best analogy I can think of is it's like bringing your lunch to school with you when you were a kid. Did you carry a lunchbox? I did, G.I. Joe. The non-Docker way would be having to go to the dining room in your house, and sit at your table in your home, and pick the things you wanted to eat off the table and put them on your plate. Then you can have your lunch meal. But all these things had to be in place and available first. The Docker way is just taking the sandwich, chips, apple, and milk and put it in a container of your lunch box. Now it's mobile and doesn't require the dependencies of the dining room or kitchen table.



`

Containers vs Images

Let's take a moment to discuss the difference between a container and an image.

Docker Image: Is a lightweight stand-alone, executable software package that includes everything needed to run an application such as the code, libraries, configurations, and other dependencies. The image is an application package stored as a software file in a Docker format.

Docker Container: An instance of the running image. It is the standard unit of software running in the compute environment.

Risks

Let's quickly cover the risks that Docker is susceptible to.

Availability:

- Removal of code or applications that will impact production.
- System resources can be exhausted and cause failure.

Integrity:

- Modification of application code.
- Malicious software being inserted in the application.
- Backdoors created.

Confidentiality:

- Sensitive and confidential data capture and exfiltration.
- Privilege escalation.

Threats & Exploits

1. Privilege Escalation

Said simply, by default a root user inside the container equals a root user on the host system.

Often times a process (PID) inside the container will require elevated root privileges to run correctly. As we've witnessed historically from other systems if one process needs it the whole system will commonly be configured to use it. So, we quickly escalate from a single process to everything running as root.

By default, any process running as root on the container also has root on the host machine. Take for example a mounted directory that supports containers writing data out to a system drive. Docker volumes are a way to provide persistent storage to Docker containers. This mounted drive exists on the host system. Because it performs application 'writes' instead of being given specific rights, it's set up to run as root. That means the ID on the container can run as root on the host mounted drive. If this drive has access to the password file, system configuration, or sensitive data this is a big security problem.

Visit this article for more information on privilege escalation using volume mounts:

https://www.electricmonk.nl/log/2017/09/30/root-your-docker-host-in-10-seconds-for-fun-and-profit/

2. Backdoors

Just as it's easier to steal code than write code, it's easier to find and download an existing Docker image from the Internet than it is to create one from scratch. Just as we've read in the news about malicious apps being found on the Apple Store, so too can Docker applications be modified to host malicious code.

There are many websites hosting already created images that serve the purpose. Now you can just search on Google for '*Apache Docker Image*' and find a free site to download it. The challenge of interrogating these sites and the images they host is extremely difficult and time-consuming. There isn't a tool available yet like VirusTotal for Docker images hosted on the Internet.

The most popular way to quickly make a malicious image is to use Dockerscan. You can quickly edit a good image with a malicious backdoor.

3. Vulnerabilities

There are vulnerabilities in Docker. They can come in several layers, the kernel, the Docker layer or in the applications and dependencies being hosted in the Docker container. If an application has a vulnerability that can be exploited (always assume it does), then an attacker has the potential of owning the running Docker container.

Two popular vulnerabilities that have posed a large risk to Docker containers are:

- *ShellShock:* vulnerability in a bash shell and can be used to exploit https, SMTP, ssh
- *Dirty Cow*: a vulnerability that allows privilege escalation

For a list of current Docker vulnerabilities you can visit:

https://www.cvedetails.com/vulnerability-list/vendor_id-13534/product_id-28125/Docker-Docker.html

Security Controls

Below is my personal compiled list of security controls that I use when I'm helping a client build a secure cloud deployment using Docker containers. I invite you (beg you) to add any other security controls you can think of to the comments section so we can all benefit from it. This is a growing space and I definitely don't know everything.

But here's what I do know, ...

1. Least Privilege

Work with developers to ensure that not everything in the container is running as root. If anything is running as root you need a detailed explanation, document it, and make sure you're doing everything to remediate it. If it cannot be running in a different privilege profile, then ensure its specific only to the process (PID) that is required. In addition, see what detection controls can be implemented to detect if something leverages this credential in a way that is not expected.

2. Namespaces

Combine a feature of Docker containers called 'namespaces' with least privilege.

Namespaces allow isolation between containers and hosts. Docker has six different namespaces available:

1. PID namespace for process isolation
2. NET namespace for managing network interfaces
3. IPC namespace for managing access to IPC resources

4. MNT namespace for managing filesystem mount points
5. UTS namespace for isolating kernel and version identifiers
6. User ID (user) namespace for privilege isolation

3. cgroups

Docker cgroups is a feature that allows you to limit the access processes and containers have to the system host resources such as PCU, RAM, IOPS, and the network.

Layering cgroups on your known good image can prevent additional processes from running, such as a malicious backdoor. Create a policy that only allows for the three known good processes in your image. Any additional process will be denied from running in the container.

cgroups can allow be used to prevent an attack against the availability of the host, such a denial of service attack that causes the Docker container to run away with system resources. A cgroup will limit how much host resources the container can consume.

4. Application-Specific Host Volumes

If the application does need to use the host for persistent storage, work with the system administrator to create application-specific volumes the Docker containers can mount to save and process data. This will reduce the risk of mounting a sensitive volume that could gain host sensitive information such as passwords or user log data.

5. Docker Registry

A Docker registry is a system for storing and distributing Docker images. Docker Hub (**https://hub.docker.com/**) is the public registry available to everyone on the Internet.

For storing your companies Docker images set up an internal and secure Docker registry. Limit the access to only those that should have it, including the scanning tools. Also prevent external, public, and Internet access to the registry.

Consider adding an additional layer of security to prevent outbound access to public Docker registries. If user security awareness training isn't doing the job, consider blacklisting (blocking) public Docker registries that you are concerned might host malicious Docker images.

For more information on how to set up a private Docker registry:

https://docs.docker.com/registry/deploying/

6. Docker Bench Security Standards

As with other technologies that have been released, the security community has worked to develop a list of best security practices called a benchmark. CIS is the most popular source for security benchmarks.

In conjunction with CIS, the 'Docker Bench Security' was created.

This tool is a script that checks your Docker deployments for security best practices and vulnerabilities. It leverages the CIS standards for best practice setup and configuration and is a great tool to include in your security program. It will check for things like TLS, Daemon running, ETC ownership, containers using trusted base images. To learn more about Docker Bench Security check out:

https://github.com/docker/docker-bench-security

7. Docker Vulnerability Scans

Ensure your Docker environment is being included in your vulnerability management program. Include network ranges of Docker containers to detect assets that were not included in your asset management tool.

There are several vulnerability management solutions that can provide vulnerability feedback for Docker. Once open source vulnerability tool specifically for Docker is Clair.

This is an open-source static analysis scan of vulnerabilities in Docker containers.

https://coreos.com/clair/docs/latest/

Another service is VeraCode's SourceClear.

https://www.youtube.com/watch?v=G4l6l9zseBg

8. Immutable Images

Docker images and containers should be immutable. That means they should never change.

Docker allows ensuring this by providing cryptographic hash values assigned to each image and each container. These values are called 'digests' in Docker. You can get a list of the values with the command:

docker images --digests

By monitoring the known good digest values, you can quickly detect and remediate any container or image that has been changed.

Doing this on a small scale can be done manually and with a routine governance or audit process. Doing this on a large scale will require a commercial solution or additional internally developed monitoring tool.

(I've had the opportunity to talk to a couple of people lately that have very good ideas on how to do this but don't want to be called out publicly. So, if you want some ideas, ping me directly.)

Security Policy

Here are some policies you should review and consider putting in place to help increase security.

Secrets Management

Ensure secretes are not stored in Docker environment variables, or in configuration files.

Instead, have a security policy that states secrets should be stored in approved solutions. I will discuss this much more in the next article on Kubernetes orchestration, as this is used more for secretes management instead of Docker. If you are using Docker Swarm for orchestration you should research how to manage secrets using Docker Swarm.

Docker Remote API

Allows the interaction with Docker containers remotely. When the API is exposed over the network, it exposes the daemon to anyone with network access.

Authentication is not required by default.

It's possible a malicious attacker can remotely connect to the API, gain access with no authentication, and then pivot and escalate privileges. A reverse shell can be set up.

Exposing the Docker Remote API is even considered a security finding and can be detected using some vulnerability detection tools such as Tenable:

https://www.tenable.com/plugins/nessus/124029

Create a policy against enabling the Remote API service without a security review.

Dangling volumes

This is a situation where a mounted volume resists available even after a container that was using the volume has been stopped, or terminated.

If there is sensitive data on these volumes like secretes the containers were using, those can be exposed to a potential attacker.

Digital Ocean provides a great Cheat Sheet:

https://www.digitalocean.com/community/tutorials/how-to-remove-docker-images-containers-and-volumes

Create a policy that requires the regular review of mounted volumes. There should be a process to review, remove, and clean up volumes.

CAP_SYS_MODULE

When this module is enabled by running the container with this capability, the container user has the ability to escape from the container by installing a kernel module onto the host's kernel.

TwistLock blog has a video:

https://www.twistlock.com/labs-blog/escaping-docker-container-using-waitid-cve-2017-5123/

--privileged flag

There is only a handful of privileges a default container has.

When a container is run with *"--privileged*" flag it will give full privilege capabilities to the container.

An attacker who is inside the container can take advantage of these capabilities to escape the container and gain a foothold on the host.

For more information on the privilege flag:

http://obrown.io/2016/02/15/privileged-containers.html

Ensure you're policy includes restricting running containers with the privilege flag. If it's required, it needs to be reviewed by security and a risk acceptation process might be invoked.

Docker Provided Defense

Apparmor

Linux security module that protects Docker containers from malicious threats. It's a Linux security tool that can be used for Docker by using a specific AppArmor policy.

You can use an AppArmor policy to further reduce the risk of container exploitation by blocking write access to mounted volumes that contain sensitive data or secrets.

I recommend leveraging the tool when the Docker production containers do require access to mounted volumes that were restricted by policy. If a security exception is made, then this tool can be used to dial in on what writes can be used on certain folders and files.

For example, if the Docker container does require a mounted volume that has the Shadow file on it, you can invoke the AppArmor policy that blocks read and write access to this specific file by the container.

seccomp

Filters syscalls from Docker containers to the host. It acts as a service firewall. Just like AppArmor you can create a policy, in the form of a JSON file. By default, nothing is blocked. But you can block things like 'CHMOD' to prevent a container user from changing rights.

Capabilities

Linux accounts have a bunch of rights that can be broken down into *'capabilities*'. Docker has implemented a system that is relational to the Linux capabilities inside Docker. Here is an example below:

Linux	Docker
CAP_CHOWN	CHOWN
CAP_NET_ADMIN	NET_ADMIN
CAP_SETUID	SETUID
CAP_SYSADMIN	SYSADMIN

In a Docker container shell, you can check the capabilities with the command 'capsh'

If you have a user that needs higher privileges, such as 'root', you can still help reduce the risk an attacker exploiting the rights by leveraging the Docker capabilities to restrict the user rights in a granular way.

Additional Security Controls

Here is an additional security control available to Docker, that I am not necessarily all on board with due to complexity, and not the right amount of risk reduction return for the investment.

Docker Content Trust: Allow the digital signing of Docker images to verify and ensure the publisher.

Conclusion

Docker is an amazing technology, but it also poses a lot of risks. By applying the same old security philosophies and methodologies to today's technologies we can exponentially reduce the risks to the business.

Next

Next, we will review our orchestration platform Kubernetes. We will look at what the risks are and how to build security processes and controls to reduce them.

Part 5: Kubernetes

The purpose of this guide is to equip individuals responsible for security with a methodology for securing internally developed applications being deployed to the cloud.

In attempts to be specific in these efforts, we're considering steps we can take to secure an application that uses the Git repository, Docker containers, Kubernetes orchestration, and hosted on the Google Cloud Platform.

Review

So far, we've covered the application code being developed according to security policy, the secure development environment, and how it's stored in an approved application repository using Git.

The code is being deployed to Docker containers for production. The docker containers are being orchestrated by Kubernetes.

Securing Kubernetes

Kubernetes Tenants of Security

1. Two Levels of encryption

2. Control user permissions

Kubernetes Secrets

What are secrets?

Secrets are any type of information that can be used to grant privileges, escalate privileges, read or access data like confidential information. This includes encryption keys, passwords, API keys and other pieces of credential information.

Risks

Kubernetes is the orchestration engine. If you have permission to the cluster control, you can get access to the Google Cloud Platform, the master nodes, the identity and access system, the application code, and control containers.

In Kubernetes, secrets are stored in etcd and are not encrypted.

They are encoded with Base64. It's very easy to derive the secrets from Base64, and there are readily available decoders available on the web. This means *Kubernetes secrets are not secure by default.*

We often find that permissions assigned within Kubernetes are overly permissive. This means that too many people have access to the information, and an attacker with any level of foothold can easily gain the information stored here.

Anyone with etcd access or backups of the system and etcd will have access to these secrets.

These secrets can be used to get dashboard access, administrative privileges, take over containers, upload alternative code, and exfiltrate data.

To reduce this risk two things can be done:

1. Encryption
2. Diligent permission controls

Encryption

When encryption is applied to the secrets it becomes much more difficult for an attacker or malicious party to obtain the credentials.

One of our security tenants for cloud applications hosting sensitive or confidential data is to always apply two layers of encryption.

By default, Google Cloud Platform is encrypted at rest. This is great but does not apply the level of security needed as this only prevents an attacker without credentials from reading the data in a stored state. If an attacker does have credentials to the application, Docker, or Kubernetes, they can read data within the application and there is a chance they can read the secrets.

So, by enabling '*application-layer*' security, you can further reduce this risk by encrypting the secrets.

Restriction

In this situation, the secrets are encrypted before they leave the application. The application is the only part of the system that will ever see the plaintext credentials, and they will not be exposed to users or saved to stored locations.

Granularity

This also provides application granularity as you can leverage multiple encryption keys to encrypt different sets of secrets. This can allow role-based user access, system access, or separation of duties.

Transmission

The data remains encrypted during transmission. Remembering our scope of security for this project, it was to protect confidential and sensitive data at rest, and in transit. The Google Cloud Platform will provide the protection at rest, but this additional layer of encryption will provide additional protection for transit.

Kubernetes Versions with Encryption

Version 1.7 was the first attempt of allowing encryption of secrets. It used the Kubernetes API to encrypt the secrets before getting stored and written to etcd. This encryption process takes place on the Kubernetes master node where the key is stored.

So if an attacker is on the master node, or can pivot to the master node, the attacker can also access the encryption and decryption keys of the secrets.

Version 1.10 allows configuration of encryption using an external party and key providers. This is a huge step towards providing great security.

In version 1.10 you can configure application-layer encryption using Google Cloud Platform, HashiCorp, or any other third-party encryption provider.

This external third-party encryption service is referred to as a '*key management store*' or KMS.

In this situation, instead of the Kubernetes master node providing and applying encryption, the master node will call the external API (KMS) to apply encryption to the secrets before they are stored.

The external API interrogates the Kubernetes master node to ensure it has the privileges to call this public service, and this parties keys, and if successful provides the Kubernetes master node with a token to access the key store. The master node presents the authentication token to the external API. The external API encrypts the data and feeds it back to the master node, which then writes the encrypted secrets to storage.

The encryption keys are stored by an external third party, and not within the Kubernetes cluster.

An attacker would have to compromise the container, etcd, the master node, and the third party KMS provider to gain access to the decryption keys.

Third-Party KMS:

- Google Cloud Platform:
- **https://github.com/GoogleCloudPlatform/k8s-cloudkms-plugin**

- Azure
- **https://github.com/Azure/kubernetes-kms**

- HashiCorp
- **https://www.hashicorp.com/resources/vault-secret-management-kubernetes**

User Permissions

User permissions can be secured by leveraging Kubernetes *'role-based authentication controls*', referred to as RBAC.

For this discussion I'll refer directly to the Kubernetes documentation located here:

https://kubernetes.io/docs/reference/access-authn-authz/rbac/

This article does a really good job of explaining, so I'll keep my explanation short.

You can use RBAC to create a 'role'. You assign the role permissions such as "get", "list", "update". You also assign the role to a Kubernetes namespace that its permissions are specific to. You can also assign the role specifically to resources within the namespace such as 'pods'.

As you can see this gets specific and granular very quickly.

Once your role is created you use the 'RoleBinding' to assign users or groups to the newly created role.

They also have 'cluster roles' that are allowed across the whole cluster. These cannot be tied to namespaces and are used specifically for cluster-wide scopes, like persistent storage.

I recommend looking at the user roles you need within Kubernetes and architect your RBAC strategy. In enterprise-size companies, the Kubernetes administrators are often separated from the application developers.

Consider outside services and resources that will need to authenticate such as databases and persistent storage. Additionally, consider what security tools you want to authenticate.

Security logging is a great use case for implementing RBAC. Historically the security logging service account has been granted administrative privileges to assets. Using RBAC you can create a role that can *"get*" the resource of "*logs*".

For a complete list of 'verbs' and 'resources' that can be used to create roles, you'll want to refer to the Kubernetes official documentation for the API version you are using. For example, if you're using API version 1.15:

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.15/

Summary

The orchestration control for your containerized applications is a high-risk asset.

There have already been public cases of companies being compromised due to default values and credentials in their Kubernetes configuration (*look up Tesla if interested*).

`

Taking time to consider adding additional encryption protection and leveraging RBAC for user roles will greatly decrease the risk of your Kubernetes environment being compromised.